

МЕТОДИЧНІ ВКАЗІВКИ ДЛЯ ВИКОНАННЯ ПРОЄКТНИХ ЗАВДАНЬ
З ДИСЦИПЛІНИ
«АЛГОРИТМИ ТА ОПТИМІЗАЦІЯ ПРОГРАМНИХ РІШЕНЬ»

Бичков О.С., Меркулова К.В., Жабська Є.О., Духновська К.К., Зубик Л.В,
Марцафей А.С.

Київ -2026

Зміст

Вступ	22
Нормативні документи для виконання робіт	23
Що читати в якій ситуації	24
П'ять правил, які варто прочитати одразу	25
Як влаштована методичка.....	26
Календар виконання та оцінювання робіт	26
Академічна доброчесність	27
Що ви матимете наприкінці	27
1. Загальні положення	28
2. Мета, завдання та очікувані результати проєктної роботи	30
2.1. Що студент повинен навчитися робити	31
2.2. Реалізація алгоритмів	32
2.3. Тестування як частина розробки.....	32
2.4. Експериментальне порівняння алгоритмів	33
2.5. Відтворюваність результатів	34
2.6. Очікуваний кінцевий результат	35
3. Організація роботи над проєктом і робота в команді	36
3.1. Зона відповідальності.....	37
3.2. Рівень складності зони	38
3.3. Зріз 0 — старт проєкту	39
3.4. Репозиторій проєкту	39
Вимоги до ведення репозиторію	40
Лог рішень	40
3.5. Робота з гілками.....	41
3.6. Issue як маленька задача.....	41
3.7. Commit	42
3.8. Code review	42
3.9. Інтеграція компонентів	43
3.10. Хто відповідає за інтеграцію	44
3.11. Комунікація в команді.....	44
3.12. Як працювати з помилками	45

3.13. Що робити, якщо студент не встигає	45
3.14. Використання штучного інтелекту	46
3.15. Що повинно бути готово після організаційного етапу	46
4. Алгоритми і структури даних, які використовуються у проєктах.....	47
4.1. Масиви, списки, словники та множини.....	48
Масив або список.....	48
Словник	48
Множина.....	48
4.2. Черга з пріоритетами.....	49
Де це потрібно у проєктах	49
Що повинен розуміти студент.....	50
4.3. Жадібні алгоритми	50
Простий приклад.....	50
Чому тоді жадібні алгоритми потрібні	51
Де використовується	51
Головне завдання студента.....	51
4.4. Динамічне програмування	51
Приклад	52
Чому це точний алгоритм	52
Де використовується у проєктах	52
Проєкт 1	52
Проєкт 2	52
Проєкт 3	53
Що повинен уміти студент	53
4.5. Рюкзак 0/1.....	53
Чому ця модель така корисна	54
4.6. Багатоваріантний рюкзак	54
4.7. Алгоритм Гаффмана.....	55
Префіксність.....	55
Як будується дерево	56
Складність	56
4.8. Канонічний Гаффман	56

Повнота коду.....	57
Навіщо це потрібно	57
Важливий крайній випадок.....	57
Де використовується	57
4.9. LZ77	57
Елемент потоку	58
Greedy LZ77.....	58
Lazy greedy	58
ДП-розбір.....	58
Де використовується	58
4.10. Хеш-таблиці та пошук збігів	58
4.11. Перцептрон.....	59
Нейрон	59
Навчання.....	59
Навіщо перцептрон у проєкті	60
Важливе правило	60
4.12. Train, validation і holdout	60
4.13. Macro-F1	61
4.14. Генетичний алгоритм	61
Популяція	62
Основні операції	62
Елітизм.....	62
З чим порівнювати.....	62
4.15. Хеш-функції, CRC і HMAC	62
CRC	62
Криптографічний хеш	63
HMAC	63
4.16. AES-CTR.....	63
Дуже важливе правило.....	63
4.17. KDF і розділення ключів.....	64
KDF виконується один раз.....	64
4.18. Encrypt-then-MAC	64

Чому стиснення йде перед шифруванням.....	65
4.19. Канонічний формат і строгий parser	65
4.20. Алгоритм як частина системи	65
4.21. Що потрібно знати перед переходом до реалізації	66
5. Загальна методика виконання проєкту	67
5.1. Крок 1. Прочитати умову повністю	68
5.2. Крок 2. Окремо прочитати специфікацію формату	69
5.3. Крок 3. Переглянути видані файли	70
5.4. Крок 4. Побудувати карту системи.....	70
5.5. Крок 5. Узгодити інтерфейси між модулями.....	71
5.6. Крок 6. Зробити найпростішу робочу версію	72
5.7. Крок 7. Спочатку перевірити happy path	73
5.8. Крок 8. Додати перший алгоритм	73
5.9. Крок 9. Кожен алгоритм спочатку тестується окремо	74
5.10. Крок 10. Використовувати маленькі приклади	74
5.11. Крок 11. Порівнювати з простим еталоном	75
5.12. Крок 12. Додати negative tests.....	75
5.13. Крок 13. Не ловити всі помилки через except Exception.....	76
5.14. Крок 14. Додати логування.....	77
5.15. Крок 15. Додати точні перевірки інваріантів.....	77
5.16. Крок 16. Тільки після стабільної бази додавати ML	77
5.17. Крок 17. Спочатку зробити baseline для ML.....	78
5.18. Крок 18. Не торкатися holdout.....	78
5.19. Крок 19. Генетичний алгоритм додавати тільки після готової fitness.....	79
5.20. Крок 20. Перед GA зробити random search	79
5.21. Крок 21. Шифрування додавати після правильного формату.....	79
5.22. Крок 22. Криптографічні параметри не вигадуються	80
5.23. Крок 23. Після кожного великого етапу запускати весь набір тестів	80
5.24. Крок 24. Додати автоматичний запуск тестів	81
5.25. Крок 25. Зібрати наскрізний тракт системи	81
5.26. Крок 26. Виміряти базову метрику	81
5.27. Крок 27. Вимірювати один фактор за раз.....	82

5.28. Крок 28. Зберігати результати експериментів	82
5.29. Крок 29. Не видаляти невдалі результати	83
5.30. Крок 30. Перед чекпоінтом зробити clean run	83
5.31. Крок 31. Не виправляти вручну результати	83
5.32. Крок 32. Кожен етап повинен мати критерій готовності	84
5.33. Рекомендована загальна послідовність	84
5.34. Головне правило цього розділу	85
6. Покрокове тестування проєкту	85
6.1. Які види тестів потрібні	85
Unit tests	86
Round-trip tests	86
Edge-case tests	86
Negative tests	86
Integration tests	86
Regression tests	86
Experimental tests	86
Acceptance tests	86
6.2. Починаємо з unit tests	86
6.3. Тестуйте не тільки великі приклади	87
6.4. Тестування структур даних	87
6.5. Тестування алгоритму Гаффмана	88
Перша перевірка — round-trip	88
Друга перевірка — префіксність	88
Третя перевірка — канонічність	88
Четверта перевірка — вироджений алфавіт	88
П'ята перевірка — некоректні таблиці	88
6.6. Тестування LZ77	89
Окремий negative test	89
6.7. Тестування greedy	90
6.8. Тестування динамічного програмування	90
6.9. Перевіряйте і значення, і відновлене рішення	90
6.10. Тестування багатоваріантного рюкзака	91

6.11. Тестування перцептрона	91
Тест forward pass	91
Тест loss	91
Тест backpropagation.....	92
Тест навчання.....	92
6.12. Не тестуйте модель на train	92
6.13. Тестування train/valid/holdout протоколу	92
6.14. Тестування генетичного алгоритму	93
Генерація популяції.....	93
Selection	93
Crossover	93
Mutation.....	93
Elitism.....	93
6.15. GA потрібно порівнювати з random search	94
6.16. Тестування криптографії	94
6.17. Тестування HMAC.....	94
Правильні дані	95
Змінити один байт	95
Змінити tag.....	95
Невірний пароль	95
6.18. Перевірка порядку HMAC і decrypt	95
6.19. Тестування nonce	95
6.20. Round-trip як головний інтеграційний тест	96
6.21. Edge cases.....	96
6.22. Boundary tests	96
6.23. Negative tests.....	97
6.24. Погані дані не повинні викликати необроблені винятки.....	97
6.25. Fuzz testing.....	98
6.26. Integration tests.....	98
6.27. Regression tests	99
6.28. Не видаляйте старі тести після виправлення	99
6.29. Experimental tests.....	99

6.30. Правило чесного експерименту	100
6.31. Вимірювання часу	100
6.32. Warm-up	100
6.33. Тестування використання ресурсів	101
6.34. Перевірка інваріантів після повного запуску	101
6.35. Як організувати каталог тестів	101
6.36. Назва тесту повинна пояснювати, що перевіряється	102
6.37. Один тест — одна зрозуміла причина падіння	102
6.38. Автоматичні тести повинні запускатися однією командою.....	102
6.39. Покриття тестами	103
6.40. Тести перед pull request.....	103
6.41. Тести після merge	103
6.42. Що потрібно показувати викладачу під час проміжної перевірки	103
1. Команду запуску тестів.....	104
2. Результат.....	104
3. Один простий unit test	104
4. Один edge case.....	104
5. Один negative case.....	104
6. Один integration test	104
7. Порівняльний experiment	104
6.43. Журнал тестування	104
6.44. Коли компонент можна вважати протестованим	105
6.45. Коли весь проєкт можна вважати протестованим.....	105
6.46. Фінальний clean test.....	105
6.47. Тестовий чек-лист перед задачею.....	106
6.48. Головне правило тестування	106
7. Покрокова задача проєкту	106
7.1. Що означає «здати етап»	107
7.2. Зріз 0 — старт проєкту	108
7.3. Що повинно бути готово до Зрізу 0.....	108
7.4. Розподіл зон на Зрізі 0.....	109
7.5. Що показувати викладачу на Зрізі 0	109

7.6. Критерій готовності Зрізу 0	110
7.7. Чекпоінт 1 — перша робоча система.....	110
7.8. Що означає мінімальна робоча версія	111
7.9. Що повинно бути готово до Чекпоінту 1	111
7.10. Що повинно бути в репозиторії на Чекпоінті 1	112
7.11. Що демонструвати на Чекпоінті 1	112
7.12. Що не потрібно робити до Чекпоінту 1.....	113
7.13. Критерій готовності Чекпоінту 1	113
7.14. Чекпоінт 2 — алгоритмічна частина.....	113
7.15. Що повинно бути готово до Чекпоінту 2	114
7.16. Порівняння greedy і точного алгоритму	114
7.17. Перевірка DP на Чекпоінті 2	114
7.18. Перцептрон на Чекпоінті 2	115
7.19. Що демонструвати на Чекпоінті 2	115
7.20. Що повинно бути в репозиторії на Чекпоінті 2	116
7.21. Критерій готовності Чекпоінту 2	116
7.22. Чекпоінт 3 — повна інтегрована система	116
7.23. Що повинно бути готово до Чекпоінту 3	117
7.24. Перевірка криптографії на Чекпоінті 3	117
7.25. Генетичний алгоритм на Чекпоінті 3.....	117
7.26. GA проти random search	118
7.27. Основна метрика проєкту	118
7.28. Що демонструвати на Чекпоінті 3	119
7.29. Після Чекпоінту 3 не починайте переписувати систему з нуля.....	119
7.30. Тиждень інтеграційних та атаквальних тестів.....	120
7.31. Що робити після знайденої помилки.....	120
7.32. Підготовка фінальної версії.....	120
7.33. Що повинно бути у фінальному репозиторії	121
7.34. README фінальної версії.....	121
7.35. Фінальний clean run	122
7.36. Фінальний тестовий набір.....	122
7.37. Фінальні експерименти	122

7.38. Що здається викладачу фінально.....	123
7.39. Що команда повинна вміти показати без підготовки	123
7.40. Індивідуальна готовність	124
7.41. Не вчіть код напам'ять.....	124
7.42. Що вважати готовим фінальним проєктом.....	124
7.43. Короткий маршрут задачі.....	125
7.44. Головне правило покрокової задачі.....	126
8. Оформлення результатів і підготовка звіту	126
8.1. Звіт пишеться протягом семестру.....	126
8.2. Рекомендована структура звіту.....	127
8.3. Постановка задачі	128
8.4. Архітектура системи	128
8.5. Для кожного компонента пояснюйте його роль.....	129
8.6. Опис алгоритму не повинен бути копією підручника	129
8.7. Псевдокод краще за великі фрагменти коду.....	129
8.8. Формули повинні бути пояснені словами.....	130
8.9. Обчислювальна складність.....	130
8.10. Не потрібно приховувати спрощення.....	130
8.11. Розділ тестування	131
8.12. Покажіть, як перевірялася правильність	131
8.13. Експериментальна частина — одна з головних.....	132
8.14. Перед кожним експериментом сформулюйте питання	132
8.15. Описуйте умови експерименту	133
8.16. Один експеримент — одна зміна	133
8.17. Не показуйте лише найкращий запуск	134
8.18. Графіки повинні відповідати на питання	134
8.19. Не використовуйте 3D-графік там, де достатньо двох ліній.....	134
8.20. Аналіз важливіший за самі числа.....	135
8.21. Нульовий виграш — теж результат	135
8.22. Негативний результат теж не потрібно приховувати	135
8.23. Ablation study.....	136
8.24. Не приписуйте алгоритму те, чого не вимірювали	136

8.25. Розділ «Відомі обмеження»	136
8.26. Висновки.....	137
8.27. Внесок учасників команди.....	137
8.28. Якщо робота виконувалася спільно	138
8.29. Посилання на код.....	138
8.30. Результати повинні генеруватися програмою	138
8.31. Зберігайте сирі результати.....	138
8.32. Результати повинні відповідати фінальній версії коду.....	139
8.33. Як оформлювати таблиці результатів.....	139
8.34. Не дублюйте одну інформацію тричі	139
8.35. Скриншоти	139
8.36. Формат чисел	140
8.37. Одиниці вимірювання	140
8.38. Округлення.....	141
8.39. Відтворюваність звіту	141
8.40. Використання зовнішніх джерел.....	141
8.41. Використання ІІІ.....	142
8.42. Перед здачею звіту перевірте узгодженість.....	142
8.43. Перевірте, чи можна зрозуміти звіт без автора	142
8.44. Рекомендований обсяг	142
8.45. Чек-лист готовності звіту.....	143
8.46. Головне правило підготовки звіту	143
9. Спеціальні методичні вказівки до Проєкту 1	144
«Захищений канал телеметрії з обмеженою пропускнуою здатністю»	144
9.1. Загальна постановка задачі.....	144
9.2. Головна ідея проєкту.....	145
9.3. Архітектура Проєкту 1	145
9.4. Що студент отримує від викладача	146
9.5. Перший етап — навчитися читати кадри.....	147
9.6. Перший важливий тест	147
9.7. Другий етап — мінімальний пакет без стиснення	148
9.8. FORMAT_SPEC є контрактом	148

9.9. Третій етап — кодування Гаффмана	149
9.10. Канонічний Гаффман	149
9.11. Обов'язкові крайні випадки для Гаффмана	149
9.12. Четвертий етап — рюкзак 0/1	150
Звідки береться вага	150
Звідки береться цінність	150
Звідки береться місткість	150
9.13. Спочатку зробіть brute force	151
9.14. Реалізація DP	151
9.15. Не забудьте про фактичний розмір пакета	151
9.16. П'ятий етап — greedy baseline	152
9.17. Головний експеримент Проєкту 1	152
9.18. Шостий етап — планувальник черги	153
9.19. Сьомий етап — підключення моделі каналу	153
9.20. Три сценарії каналу	153
9.21. Transmission budget	154
9.22. Восьмий етап — класифікатор	154
Поле <code>critical</code> не є міткою класу	154
Ознак одного кадру недостатньо	154
9.23. Не починайте MLP з повного датасету	155
9.24. Train, valid і holdout	155
9.25. Перевірка класифікатора	156
9.26. Як класифікатор впливає на систему	156
9.27. Просунутий прогноз наступного кадру	157
9.28. Дев'ятий етап — криптографічний захист	157
9.29. Виведення ключів	157
9.30. AES-256-CTR	158
9.31. HMAC перевіряється першим	158
9.32. Обов'язкові negative tests для захисту	158
9.33. Обов'язковий експеримент з порядком операцій	159
9.34. Десятий етап — основна метрика Q	159
9.35. Не оптимізуйте лише стиснення	160

9.36. Вимірюйте Q до оптимізації.....	160
9.37. Одинадцятий етап — генетичний алгоритм	160
9.38. Fitness	161
9.39. Оператори GA	161
9.40. Перевірка елітизму	161
9.41. GA проти random search	162
9.42. Дванадцятий етап — наскрізний тракт	162
9.43. Перевірка на трьох сценаріях	162
9.44. Приховані seed каналу.....	163
9.45. Що потрібно дослідити експериментально.....	163
9.46. Дублювання критичних кадрів.....	164
9.47. Модуль атаки	164
9.48. Аналіз трафіку.....	164
9.49. Покрокове тестування Проєкту 1	165
9.50. Які помилки потрібно спеціально спробувати викликати.....	165
9.51. Що потрібно показати на Чекпоінті 1	166
9.52. Що потрібно показати на Чекпоінті 2	167
9.53. Що потрібно показати на Чекпоінті 3	167
9.54. Фінальні приймальні перевірки	167
9.55. Що потрібно мати у фінальних результатах	168
9.56. Які питання студент повинен уміти пояснити.....	168
9.57. Типова неправильна стратегія.....	169
9.58. Головний критерій успіху Проєкту 1	169
9.59. Короткий маршрут студента	169
9.60. Головна порада	171
10. Спеціальні методичні вказівки до Проєкту 2	172
«Розумний захищений архіватор файлів»	172
10.1. Постановка задачі	172
10.2. Що означає «розумний архіватор»	173
10.3. Що отримує команда.....	174
10.4. Сім типів даних.....	174
10.5. Перший етап — простий архів без стиснення	175

10.6. Перший головний тест	175
10.7. Почніть з одного тому	176
10.8. Формат архіву є контрактом	176
10.9. Зовнішній заголовок тому	177
10.10. Другий етап — таблиця файлів	177
10.11. Безпечні імена файлів	177
10.12. Третій етап — розбиття файла на блоки	178
10.13. Спочатку протестуйте блоки без стиснення	179
10.14. Четвертий етап — LZ77	179
10.15. Спочатку реалізуйте decoder	179
10.16. Жадібний LZ77	180
10.17. Лінива жадібність	180
10.18. Обов'язкове порівняння LZ77	180
10.19. Межі LZ77	181
10.20. Важлива межа DIST	181
10.21. П'ятий етап — Huffman	182
10.22. Алфавіти Huffman	182
10.23. Канонічний Huffman	182
10.24. Максимальна довжина коду	183
10.25. Рівність Крафта	183
10.26. Вироджений випадок Huffman	183
10.27. Шостий етап — NO_COMPRESSION	183
10.28. Не використовуйте просте правило «висока ентропія = не стискати» ...	184
10.29. Сьомий етап — оптимальний LZ77-розбір	184
10.30. Фіксовані вартості DP	185
10.31. Як тестувати DP-розбір	185
10.32. Восьмий етап — пакування в томи	185
10.33. Вага блока	186
10.34. Місткість першого тому	186
10.35. Важливий момент з таблицею файлів	186
10.36. DP-рюкзак для тому	187
10.37. Greedy для пакування	187

10.38. Якщо блок не поміщається	187
10.39. Порожній архів	187
10.40. Дев'ятий етап — класифікатор типу файла	188
10.41. Спочатку перевірте ознаки	188
10.42. Протокол навчання	188
10.43. Навіщо класифікатор потрібен архіватору	189
10.44. Десятий етап — криптографічний захист	189
10.45. KDF	190
10.46. PBKDF2 виконується один раз на архів	190
10.47. AES-256-CTR	190
10.48. Nonce не можна повторювати і між різними архівами	191
10.49. HMAC	191
10.50. CRC32 і HMAC виконують різні ролі	191
10.51. Багатотомний архів.....	192
10.52. Порядок подання томів не має значення.....	192
10.53. Відсутній том	192
10.54. Одинадцятий етап — метрика Q.....	193
10.55. Найважливіша умова метрики.....	193
10.56. Спочатку baseline.....	193
10.57. Дванадцятий етап — генетичний алгоритм.....	194
10.58. GA не повинен ламати правильність	194
10.59. GA проти random search	194
10.60. Елітизм.....	194
10.61. Тринадцятий етап — наскрізний тракт	195
10.62. Покрокове тестування Проекту 2.....	195
10.63. Обов'язкові маленькі тести	196
10.64. Обов'язкові negative tests	196
10.65. Контрольовані коди помилок	197
10.66. Еталонний архів	197
10.67. Що показати на Чекпоінті 1.....	198
10.68. Що показати на Чекпоінті 2.....	198
10.69. Що показати на Чекпоінті 3.....	199

10.70. Фінальні приймальні перевірки	199
10.71. Які експерименти обов'язково провести.....	199
10.72. Що вимірювати для різних типів файлів.....	200
10.73. Не соромтеся режиму NO_COMPRESSION	200
10.74. Типова помилка — оптимізувати до перевірки правильності	201
10.75. Типова помилка — стиснути весь файл одним великим блоком	201
10.76. Типова помилка — довіряти архіву	201
10.77. Що студент повинен уміти пояснити	202
10.78. Короткий маршрут Проєкту 2	202
10.79. Головна порада	204
11. Спеціальні методичні вказівки до Проєкту 3	205
«Розумна захищена система пріоритетної реплікації журналів подій»	205
11.1. Постановка задачі	205
11.2. Чому не можна просто зробити три копії всього	206
11.3. Головна ідея Проєкту 3	206
11.4. Загальна архітектура	207
11.5. Що отримує команда.....	207
11.6. Модель відмов уже готова.....	208
11.7. Чому replication.py не можна змінювати	208
11.8. Перший етап — навчитися читати один запис	209
11.9. Перший тест	209
11.10. Перевірка полів	209
11.11. Другий етап — найпростіша репліка без стиснення.....	210
11.12. Формат LOG3	210
11.13. Спочатку реалізуйте payload без криптографії.....	211
11.14. Основний інваріант payload.....	211
11.15. Порожній журнал.....	211
11.16. Третій етап — Huffman	211
11.17. Канонічний Huffman.....	212
11.18. Коли вмикати NO_COMPRESSION	212
11.19. Четвертий етап — класифікація подій	213
11.20. Важливість часової історії	213

11.21. Мінімальні часові ознаки.....	213
11.22. Поле <code>critical</code> не є правильною відповіддю для <code>classifier</code>	213
11.23. Протокол навчання.....	214
11.24. Не підбирайте модель під <code>holdout</code>	214
11.25. Як <code>classifier</code> входить у реплікацію.....	214
11.26. П'ятий етап — обчислення ваги події.....	215
11.27. Не змішуйте вагу і цінність.....	215
11.28. Шостий етап — багатоваріантний рюкзак.....	216
11.29. Вартість кількох копій.....	216
11.30. Корисність кількох копій.....	216
11.31. Чому <code>factors</code> кумулятивні.....	216
11.32. Стан <code>DP</code>	217
11.33. Складність <code>DP</code>	217
11.34. Як тестувати <code>DP</code>	217
11.35. Перевіряйте не тільки <code>utility</code>	218
11.36. Сьомий етап — <code>greedy baseline</code>	218
11.37. Чому <code>greedy</code> не гарантує оптимум.....	218
11.38. Головний алгоритмічний експеримент.....	218
11.39. Вікно реплікації.....	219
11.40. Восьмий етап — вибір вузлів.....	219
Чому $R_{\max} = 3$	219
ДП не знає про місткість окремих вузлів.....	219
11.41. Вузли мають різну надійність.....	220
11.42. Копії на різних вузлах.....	220
11.43. Жадібний планувальник вузлів.....	220
11.44. Порядок операцій має значення.....	221
11.45. Дев'ятий етап — штатний сценарій.....	221
11.46. Деградований сценарій.....	221
11.47. Аварійний сценарій.....	221
11.48. Не оптимізуйте під відкритий <code>seed</code>	222
11.49. Десятий етап — криптографічний захист.....	222
11.50. Виведення ключів.....	222

11.51. PBKDF2 один раз на сеанс.....	223
11.52. AES-256-CTR	223
11.53. HMAC перед decrypt.....	223
11.54. Обов'язкові negative tests	223
11.55. Контрольовані коди помилок	224
11.56. Відновлення після відмов	224
11.57. Дублікати — це нормально	224
11.58. Одинадцятий етап — метрика R	225
11.59. Метрика S	225
11.60. Метрика T	225
11.61. Загальна метрика	226
11.62. Чому не можна оптимізувати лише R.....	226
11.63. Дванадцятий етап — генетичний алгоритм.....	226
11.64. Не запускайте GA до готової системи.....	226
11.65. Fitness	227
11.66. GA проти random search	227
11.67. Елітизм.....	227
11.68. Тринадцятий етап — наскрізний тракт	227
11.69. Як тестувати повну систему	228
11.70. Покроковий порядок тестів	228
11.71. Обов'язкові маленькі приклади	229
11.72. Boundary tests для DP	229
11.73. Перевірка зменшеної віддачі	229
11.74. Що показати на Чекпоінті 1	230
11.75. Що показати на Чекпоінті 2.....	230
11.76. Що показати на Чекпоінті 3.....	231
11.77. Базові фінальні приймальні перевірки	231
11.78. Вплив кількості копій.....	231
11.79. Не робіть висновок «3 копії завжди краще»	232
11.80. Вплив classifier	232
11.81. Вплив DP	232
11.82. Вплив GA.....	233

11.83. Обов'язковий експеримент із криптографією	233
11.84. Модуль атаки	233
11.85. Просунутий аналіз метаданих	233
11.86. Просунуті контрольні точки журналу	234
11.87. Крос-сумісність.....	234
11.88. Типова помилка — змішати дві задачі	234
11.89. Типова помилка — вважати другу копію другим незалежним предметом	235
11.90. Типова помилка — рахувати лише успішні записи в бюджеті.....	235
11.91. Типова помилка — робити три копії на одному вузлі.....	235
11.92. Типова помилка — використовувати severity як готовий клас.....	236
11.93. Типова помилка — оптимізувати на одному seed.....	236
11.94. Що потрібно зберігати в результатах	236
11.95. Що студент повинен уміти пояснити	236
11.96. Короткий маршрут Проєкту 3	237
11.97. Головна порада	239
12. Додатки.....	240
Додаток А. Шаблон README.md	240
Додаток Б. Короткий шаблон META-USAGE.md.....	244
Додаток В. Шаблон підсумкового звіту	245
Додаток Г. Шаблон розділу «Постановка задачі».....	246
Додаток Д. Шаблон опису алгоритму	246
Додаток Е. Шаблон опису експерименту.....	247
Додаток Ж. Приклад експерименту «DP проти greedy».....	248
Додаток З. Шаблон журналу експериментів.....	249
Додаток И. Мінімальний машинозчитуваний журнал експериментів	250
Додаток К. Шаблон журналу знайдених помилок	251
Додаток Л. Чек-лист Зрізу 0	252
Додаток М. Чек-лист Чекпоінту 1	252
Додаток Н. Чек-лист Чекпоінту 2	252
Додаток О. Чек-лист Чекпоінту 3	253
Додаток П. Чек-лист перед pull request	253

Додаток Р. Чек-лист code review	253
Додаток С. Чек-лист алгоритму DP	254
Додаток Т. Чек-лист greedy	254
Додаток У. Чек-лист Huffman	254
Додаток Ф. Чек-лист LZ77	255
Додаток Х. Чек-лист MLP	255
Додаток Ц. Чек-лист GA	255
Додаток Ч. Чек-лист криптографії	256
Додаток Ш. Чек-лист parser	256
Додаток Щ. Чек-лист negative testing	257
Додаток Ю. Чек-лист експерименту	257
Додаток Я. Чек-лист фінального звіту	257
Додаток АА. Чек-лист фінального clean run	258
Додаток АБ. Чек-лист готовності до захисту	258
Додаток АВ. Чек-лист команди перед захистом	259
Додаток АГ. Рекомендована структура фінального репозиторію	259
Додаток АД. Рекомендована структура experiments/	260
Додаток АЕ. Рекомендована структура results/	260
Додаток АЄ. Короткий шаблон фінальної демонстрації	260
Додаток АЖ. Що робити, якщо під час захисту знайдено помилку	261
Додаток АЗ. Головний фінальний чек-лист	261
Підсумок	262
Карта документів	262
Хто що видає і як часто змінюється	262
Де шукати відповідь	263
Що лишається у викладача	264
Правило пріоритету	265
Два режими читання	265
Перелік рекомендованої літератури	265

Вступ

Ця методичка призначена студентам, які виконують семестрову проектну роботу з дисципліни «Алгоритми і структури даних», і викладачам, які цю роботу супроводжують і оцінюють.

Вона не замінює лекції й не переказує підручник. Тут майже немає теорії алгоритмів у звичному вигляді: доведення оптимальності Гаффмана, виведення рекурсій динамічного програмування і аналіз складності ви знайдете в курсі лекцій. Ця книга відповідає на інше питання:

Що саме робити щодня протягом тринадцяти тижнів, щоб наприкінці мати працездатну, перевірену систему і вміти пояснити кожне прийняте в ній рішення.

У лабораторній роботі алгоритм існує сам по собі. Студент реалізує чергу з пріоритетами, здає її і переходить до наступної теми. Алгоритм працює на підготовлених даних, у відриві від будь-якої системи, і питання «навіщо він тут» просто не виникає.

Проект влаштований інакше. Кожен алгоритм у ньому є компонентом, від якого залежать інші компоненти. Помилка в кодуванні Гаффмана виявляється не в тесті на Гаффмана, а в тому, що архів не відкривається. Неправильно порашована вага в задачі про рюкзак не дає повідомлення про помилку: вона тихо псує підсумкову метрику. Перцептрон, навчений на неправильно розділених даних, показує чудові числа, які нічого не означають.

Саме тому в проекті оцінюється не лише те, що програма запускається. Оцінюється, чи правильно сформульована задача, чи доречно обрані алгоритми, чи перевірені крайні випадки, чи можна відтворити результати і чи вміє автор пояснити власне рішення.

Є ще одна відмінність, менш очевидна. У лабораторній роботі є правильна відповідь. У проекті її часто немає: два алгоритми можуть дати однаковий результат, складніший підхід може програти простішому, а покращення в одному місці може погіршити систему в цілому. Розібратися, чому саме так сталося, і є основним змістом роботи.

Пропонуються три проекти:

№	Назва	Предметна область
1	Захищений канал телеметрії з обмеженою пропускною здатністю	передавання даних за втрат і вузького каналу
2	Розумний захищений архіватор файлів	стиснення і зберігання

№	Назва	Предметна область
3	Розумна захищена система пріоритетної реплікації журналів подій	надійність і резервування

Предметні задачі різні, але набір тем і логіка роботи однакові. У кожному варіанті є аналіз даних, стиснення, оптимізаційна задача з динамічним програмуванням, жадібний алгоритм для порівняння, криптографічний захист і автоматичне налаштування параметрів. Команда обирає один варіант.

Проекти рівноцінні за обсягом і оцінюються за однаковими критеріями. Різниця лише в тому, які саме труднощі вийдуть на перший план: у першому варіанті це поведінка системи за втрат, у другому — алгоритми пошуку збігів, у третьому — розподіл обмеженого ресурсу між подіями різної важливості.

Методичка — не єдиний документ, з яким ви працюєте. Їх кілька, вони мають різний статус, і плутати їх не варто.

Ця книга описує метод, а не завдання. Тут ви не знайдете ні тексту умови свого варіанта, ні специфікації формату, ні жодного конкретного числа — порога, коефіцієнта чи розміру вікна. Усе це видає викладач окремим комплектом на початку семестру і не входить до методички навмисно. Умова змінюється щороку, специфікація — при зміні версії формату, а числа взагалі свої для кожної команди. Метод роботи при цьому лишається тим самим, тому книга і матеріали живуть окремим життям і оновлюються незалежно.

Практичний наслідок простий. Методичку ви отримуєте один раз і користуєтеся нею як довідником; комплект матеріалів отримуєте на початку роботи і зобов'язані звірятися саме з ним щоразу, коли йдеться про конкретну вимогу або конкретне значення. Якщо в тексті цієї книги трапляється число, воно наведене лише як приклад для ілюстрації. Робоче значення завжди береться з виданого вам `team_config.json`.

Нормативні документи для виконання робіт

За ними виставляють бали, на них посилаються у спірних випадках. Вони не тлумачаться на власний розсуд.

Документ	Що визначає	Хто видає
Положення про оцінювання	критерії, ваги, формулу підсумку, множник за доброчесність	кафедра
Умова проекту	що саме треба зробити у вашому варіанті, зони відповідальності, приймальні перевірки	викладач, для варіанта

Документ	Що визначає	Хто видає
FORMAT_SPEC	формат даних із точністю до біта, коди відмови	викладач, для варіанта
CHANNEL_SPEC або REPLICATION_SPEC	модель середовища: втрати, бюджети, параметри	викладач, для варіанта
team_config.json	числа саме для вашої команди: коефіцієнти метрики, пороги, seed	викладач, для команди

Окремо стоять видані дані та еталонні файли: вибірки, робочий корпус, правильні й навмисно пошкоджені зразки, очікувані коди відмов. Це не текст, а перевірка: ваша система зобов'язана прочитати правильні зразки і відхилити пошкоджені саме з тими кодами, які записано у файлі очікувань.

Методичка пояснює, як працювати, у якому порядку реалізовувати, що і як тестувати, які помилки типові. Вона нічого не додає до вимог і нічого з них не скасовує.

Тому за будь-якої розбіжності діє просте правило:

Нормативний документ важливіший за методичку. Якщо ви бачите розбіжність, повідомте викладача, а не обирайте зручніший варіант.

FORMAT_SPEC — це контракт. Він або виконується точно, або не виконується. Поле в два байти не можна записати чотирма, навіть якщо ваш власний читач це розуміє.

Методичка натомість містить рекомендації, які можна виконати по-різному: порядок реалізації, спосіб організації тестів, структуру звіту. Тут є простір для рішень, і за нього ніхто не карає.

Змішувати ці два режими читання шкідливо в обидва боки. Студент, який тлумачить специфікацію, отримує несумісну систему. Студент, який сприймає кожну пораду методички як жорстку вимогу, витрачає час на непотрібну формальність.

Числа теж розділені навмисно. Розділи цієї книги посилаються на team_config.json, а не наводять конкретні значення. Коефіцієнти метрики, пороги і seed змінюються від семестру до семестру і від команди до команди, а методика роботи — ні.

Що читати в якій ситуації

Ситуація	Куди дивитися
«Що взагалі треба зробити?»	умова свого варіанта
«Скільки байтів у цьому полі?»	FORMAT_SPEC

Ситуація	Куди дивитися
«Яке значення порога у нас?»	<code>team_config.json</code>
«З чого почати і що робити далі?»	розділ 5 цієї книги
«Як це протестувати?»	розділ 6 і чек-листи додатків
«Скільки за це балів?»	Положення про оцінювання
«Чому мій розв’язок не читається чужою системою?»	FORMAT_SPEC і еталонні файли
«Я не знаю алгоритм»	Підручник з АСД Бичкова О.С. (шукай на сайті кафедри)

П’ять правил, які варто прочитати одразу

Решта книги розгортає ці правила докладно. Але якщо з усього тексту запам’ятати лише п’ять речей, то саме ці.

1. Спочатку найпростіша правильна версія. Не починайте з найцікавішого алгоритму. Почніть із наскрізного тракту, який працює погано, але працює цілком: дані входять, зберігаються чи передаються, відновлюються побайтово точно. Тоді кожен наступний алгоритм можна виміряти окремо, і в будь-який момент у вас є працездатна система.
2. Формат даних не вигадується. Ви отримаєте специфікацію, описану з точністю до біта, і набір еталонних файлів. Ваша система зобов’язана їх правильно прочитати. Власний передавач і власний приймач можуть мати однакову помилку і чудово розуміти один одного — саме тому справжньою перевіркою є чужі дані, а не власні.
3. Вимірюється вплив, а не гарантується користь. Динамічне програмування, перцептрон або генетичний алгоритм цілком можуть не дати виграшу на ваших даних. Це не помилка і не знижує оцінку. Знижує оцінку неспроможність пояснити, чому виграшу немає. Чесно виміряний нуль із поясненням цінніший за красивий графік без розуміння.
4. Оцінювання переважно індивідуальне. Зі ста балів проєкту лише тридцять припадають на спільну якість продукту. Решта сімдесят — це особистий внесок: власна зона відповідальності, історія роботи в репозиторії, результативність рев’ю і персональний захист. Тому «бути в команді» недостатньо, а надмірні зусилля одного учасника не рятують решту.
5. Розуміння важливіше за код. На захисті питають не «що ви написали», а «чому саме так». Студент, який не може пояснити фрагмент зі свого модуля, фактично не виконав цю частину роботи, незалежно від того, звідки цей фрагмент узявся.

Як влаштована методичка

Розділи	Про що
1–2	загальні положення, мета і очікувані результати
3	організація роботи в команді, зони відповідальності, Зріз 0
4	алгоритми і структури даних, які використовуються у проєктах
5	загальна методика виконання: тридцять два кроки від умови до готової системи
6	покрокове тестування
7	покрокова здача: чекпоінти, фінальна версія, захист
8	оформлення результатів і підготовка звіту
9–11	спеціальні вказівки до кожного з трьох проєктів
12	додатки: шаблони, чек-листи, приклади

Читати підряд не обов'язково. Розумний порядок такий: розділи 1–3 перед початком роботи, розділ 4 як довідник під час реалізації, розділ 5 як план дій, розділ свого варіанта (9, 10 або 11) відкритий постійно, розділи 6–8 у міру наближення до відповідних етапів, додатки — щоразу перед здачею етапу.

Календар виконання та оцінювання робіт

Проєкт розрахований на тринадцять тижнів і має чотири контрольні точки.

Етап	Тиждень	Головний зміст
Зріз 0	1	зони відповідальності, рівні складності, план, репозиторій
Чекпоінт 1	4	наскрізний тракт без стиснення і кодування Гаффмана
Чекпоінт 2	8	динамічне програмування проти жадібного, класифікатор
Чекпоінт 3	11	шифрування, генетичний алгоритм, повна система
Взаємні перевірки	12	крос-тести і атаки між командами
Захист	13	фінальна демонстрація

Семестрова оцінка складається з двох частин:

Семестрова оцінка (0–60) = Підсумок проєкту (0–100) × 0,4 +
Лабораторні та ДЗ (0–20)

Проєктна складова вважається складеною за умови не менше 20 балів із 40.

Проміжні перевірки існують не для того, щоб збільшити кількість контрольних робіт. Вони потрібні, щоб помилка, зроблена на четвертому тижні,

не встигла прорости в стиснення, шифрування і всі експерименти до тринадцятого.

Академічна доброчесність

Використання інструментів генерації коду, зокрема систем штучного інтелекту, дозволене. Але за трьох умов: воно задеклароване у файлі `META-USAGE.md`, студент може пояснити обрану стратегію їх застосування і розуміє зміст кожного отриманого фрагмента.

Доброчесність діє як множник до підсумку проєкту, а не як окремий критерій. Це означає, що високі бали за інші пункти її не компенсують: недекларований згенерований код зменшує підсумок удвічі, а видавання чужої роботи за власну обнуляє його.

Правило, яке варто тримати в голові:

Можна користуватися інструментом, але не можна передати йому відповідальність за розуміння власного проєкту.

Що ви матимете наприкінці

Не набір файлів із кодом, а завершену систему: вона виконує задачу свого варіанта, правильно поводить себе на виданих даних, контрольовано відмовляється працювати з пошкодженими, має автоматичні тести, `README`, за яким запускається з нуля, виміряні результати експериментів і звіт, у якому пояснено, чому кожне рішення прийнято саме так.

І, що важливіше за сам продукт, — досвід доведення великої програмної роботи від умови до захисту без втрати контролю над нею.

Редагування, форматування, пошук помилок виконано із застосуванням штучного інтелекту.

1. Загальні положення

Проектна робота з дисципліни «Алгоритми і структури даних» — це не просто велике програмне завдання наприкінці семестру. Її головна мета — навчити застосовувати алгоритми не окремо, а як частини реальної програмної системи.

Під час лабораторної роботи зазвичай розглядається один алгоритм або одна структура даних. Наприклад, студент реалізує чергу з пріоритетами, динамічне програмування, дерево Гаффмана або жадібний алгоритм. У проєкті ситуація інша. Тут потрібно зрозуміти, де саме цей алгоритм потрібен, яку задачу він розв'язує, з чим його можна порівняти і як перевірити, що він справді працює правильно.

Саме тому кожен проєкт побудований як невелика завершена програмна система. Вона складається з кількох пов'язаних частин:

- аналіз вхідних даних і визначення їхнього типу або стану;
- стиснення;
- вибір і пакування даних в обмежений обсяг — саме тут працює динамічне програмування і його порівняння з жадібним алгоритмом;
- захист даних;
- автоматичне налаштування параметрів усієї системи;
- інтеграція, тестування і вимірювання якості.

Розподіл цих частин між учасниками команди фіксується на початку роботи як індивідуальні зони відповідальності.

У межах дисципліни пропонуються три варіанти проєктної роботи:

- захищена система передавання телеметричних даних;
- розумний захищений архіватор файлів;
- система пріоритетної реплікації журналів подій.

Формат даних у проєкті не вигадується, а видається. До початку роботи команда отримує специфікацію формату, описану з точністю до біта, і набір еталонних файлів, які її система зобов'язана правильно прочитати. Це принципова умова: завдяки їй роботи різних команд можна перевіряти однаково, а системи двох команд можуть обмінюватися даними між собою. Вигадувати власний формат не потрібно і не дозволяється.

Предметні задачі у цих проєктах різні, але логіка роботи над ними однакова. У кожному випадку студентам потрібно пройти шлях від постановки задачі до готової системи:

зрозуміти задачу

→ розбити її на частини

→ вибрати алгоритми

→ реалізувати базову версію

→ протестувати окремі компоненти

- інтегрувати їх
- провести експерименти
- порівняти алгоритми
- підготувати фінальну версію
- захистити прийняті рішення

У проєктній роботі оцінюється більше, ніж те, що програма «запускається». Оцінюється також:

- чи правильно сформульована задача;
- чи доречно обрані алгоритми;
- чи коректно вони реалізовані;
- чи перевірені крайні випадки;
- чи можна відтворити результати;
- чи вміє студент пояснити власне рішення;
- чи проведено порівняння різних підходів;
- чи зрозуміло, чому один алгоритм працює краще або гірше за інший.

Тому у проєкті важливий не лише кінцевий результат, а і процес отримання цього результату.

Проєкт виконується командою. Рекомендований склад команди — від чотирьох до шести студентів. При цьому кожен учасник повинен мати власну зону відповідальності і вміти пояснити і загальну архітектуру проєкту, і ту частину, яку він реалізовував самостійно.

Проєкт розрахований на 13 тижнів і виконується поетапно. Робота не повинна відкладатися до останніх тижнів. Саме тому передбачено чотири контрольні точки, під час яких перевіряється не готовий проєкт цілком, а його окремі частини:

Точка	Тиждень	Що перевіряється
Зріз 0	1	зони відповідальності, рівні складності, репозиторій, план
Чекпоінт 1	4	наскрізний тракт без стиснення і кодування Гаффмана
Чекпоінт 2	8	динамічне програмування, жадібні алгоритми, класифікатор
Чекпоінт 3	11	шифрування, генетичний алгоритм, повний набір таблиць звіту

Тижні 12 і 13 відведено на взаємні перевірки між командами і на захист.

Такий підхід має важливу перевагу: якщо помилка з'явилася на ранньому етапі, її можна виправити до того, як на неправильному рішенні буде побудована вся система.

Наприклад, якщо студент неправильно реалізував формат даних, немає сенсу одразу додавати до нього стиснення, шифрування і генетичний алгоритм. Спочатку потрібно домогтися правильної базової передачі або відновлення даних.

Тому протягом усієї роботи діє просте правило:

Спочатку зробіть найпростішу правильну версію. Потім поступово додавайте складні алгоритми.

Це правило є одним із головних у всій методиці виконання проєкту.

2. Мета, завдання та очікувані результати проєктної роботи

Головна мета проєктної роботи — навчитися застосовувати алгоритми і структури даних для розв’язання складних прикладних задач.

При цьому важливо не просто знати назву алгоритму або вміти відтворити його псевдокод. Студент повинен навчитися відповідати на більш практичні питання:

- навіщо цей алгоритм потрібен у конкретній системі;
- які дані він отримує на вході;
- який результат повинен повернути;
- яка його обчислювальна складність;
- які існують крайні випадки;
- з яким іншим алгоритмом його варто порівняти;
- у яких ситуаціях він працює добре;
- у яких ситуаціях його використання не дає виграшу.

Наприклад, динамічне програмування у межах курсу не розглядається лише як абстрактна формула виду

$DP[i] = \dots$

У проєкті воно повинно розв’язувати реальну задачу.

У різних варіантах це може бути:

- вибір даних, які варто передати через обмежений канал;
- вибір блоків, які потрібно розмістити у томі архіву;
- вибір кількості резервних копій для подій різної важливості.

Так само жадібний алгоритм потрібен не сам по собі. Він використовується як простіший і швидший підхід, з яким можна порівняти точний алгоритм.

Написати обидва варіанти замало. Треба провести експеримент і відповісти на питання:

Наскільки точний алгоритм кращий за жадібний і чи завжди ця різниця достатньо велика, щоб виправдати додаткові обчислення?

Такі питання перетворюють реалізацію алгоритму на повноцінне дослідження.

2.1. Що студент повинен навчитися робити

Після виконання проєктної роботи студент повинен уміти розкласти складну задачу на окремі підзадачі.

Це означає, що перед початком програмування потрібно побачити структуру майбутньої системи.

Наприклад:

вхідні дані

- аналіз
- вибір алгоритму
- обробка
- оптимізація
- захист
- збереження або передавання
- перевірка результату

Кожен блок повинен мати зрозумілий вхід, вихід і відповідальність.

Студент також повинен уміти обирати структури даних відповідно до задачі.

Наприклад:

- черга з пріоритетами використовується там, де постійно потрібно вибирати найкращий або найменший елемент;
- словники і хеш-таблиці — для швидкого пошуку;
- масиви і списки — для послідовної обробки;
- дерева — для побудови кодів;
- таблиці динамічного програмування — для збереження проміжних оптимальних результатів.

Ще одна важлива навичка — вміння оцінювати складність алгоритму.

Студент повинен не просто сказати, що алгоритм «повільний» або «швидкий», а вміти пояснити це через кількість операцій.

Наприклад:

$O(n)$

$O(n \log n)$

$O(n \cdot W)$

$O(n^2)$

При цьому оцінка складності повинна бути пов'язана з реальною задачею.

Наприклад, якщо використовується рюкзак зі складністю $O(n \cdot W)$, потрібно розуміти, що таке n і що таке W саме у цьому проєкті.

2.2. Реалізація алгоритмів

Навчальні алгоритми курсу реалізуються студентами самостійно. Це принципова вимога.

Мета проєкту полягає не в тому, щоб знайти готову функцію в бібліотеці, а в тому, щоб зрозуміти роботу алгоритму на практиці.

Тому перелік поділений на три частини, і він однаковий для всіх варіантів проєкту.

Реалізується самостійно (це предмет вивчення дисципліни):

- кодування Гаффмана, включно з канонічними кодами й обмеженням довжини коду;
- пошук збігів LZ77, якщо його передбачає варіант;
- динамічне програмування;
- жадібні алгоритми, з якими порівнюється динамічне програмування;
- генетичний алгоритм;
- багатошаровий перцептрон разом із навчанням.

Дозволено брати з бібліотек (службові операції, які не є предметом дисципліни):

- AES, SHA-256, HMAC і функції виведення ключа з пароля;
- робота з файлами, мережею і часом;
- побудова графіків;
- засоби автоматичного тестування.

Заборонено в робочому тракті системи:

- готові архіватори і компресори (zlib, lzma, bzip2 та подібні);
- бібліотеки машинного навчання (TensorFlow, PyTorch, scikit-learn та подібні).

Викликати заборонені засоби можна лише як еталон для порівняння у звіті, з явною позначкою в коді. Наприклад, порівняти власний архіватор із zip рівня 9 дозволено; використати zip усередині власного архіватора — ні.

Використання інструментів генерації коду, зокрема ШІ, регулюється окремо: воно дозволене за умови декларування у файлі `META-USAGE.md` і здатності студента пояснити кожен згенерований фрагмент. Докладніше про це — у розділі про академічну доброчесність.

2.3. Тестування як частина розробки

Ще одна мета проєктної роботи — навчити студента перевіряти програму систематично.

Тестування не повинно виконуватися лише наприкінці.

Кожен новий компонент спочатку перевіряється окремо.

Наприклад:

реалізували Гаффмана

→ перевірили Гаффмана

реалізували ДП

→ перевірили ДП

реалізували шифрування

→ перевірили шифрування

після цього

→ об'єднали компоненти

Не рекомендується одразу підключати п'ять нових компонентів, а потім намагатися знайти, у якому саме з них виникла помилка.

Окремо повинні перевірятися крайні випадки.

Наприклад:

- порожні дані;
- рівно один елемент;
- вироджений алфавіт: усі дані складаються з одного символу;
- обсяг даних, менший за мінімальну одиницю обробки;
- обсяг даних, більший за доступну оперативну пам'ять, тобто потреба в потоковій обробці;
- вичерпання обмеженого ресурсу, заданого умовою варіанта: бюджету часу передавання, місткості пакета або розміру тому;
- некоректний формат вхідних даних;
- навмисно пошкоджені дані.

У хорошому проєкті тест повинен підтверджувати правильну роботу і водночас показувати, що програма правильно відмовляється працювати з некоректними даними.

2.4. Експериментальне порівняння алгоритмів

У кожному проєкті є алгоритми, які потрібно реалізувати і порівняти між собою.

Це одна з найважливіших частин роботи.

Типовий експеримент має виглядати так:

однакові вхідні дані

→ алгоритм А

→ результат А

ті самі дані
→ алгоритм В
→ результат В

порівняння

Порівнювати алгоритми на різних даних некоректно.

Для вимірювання часу цього замало: потрібні ще й однакові умови запуску. Один замір часу нічого не означає, бо на нього впливають фонові процеси. Час слід міряти не менше ніж за п'ять повторних запусків і наводити середнє значення разом із розкидом, а обидва алгоритми запускати на одній машині в одному сеансі.

Наприклад, якщо порівнюються динамічне програмування і жадібний алгоритм, обидва повинні отримати абсолютно однаковий набір вхідних даних.

Можна порівнювати:

- якість результату;
- розмір даних;
- час виконання;
- використану пам'ять;
- кількість збережених важливих подій;
- коефіцієнт стиснення;
- інші показники, визначені умовою проєкту.

Результат експерименту не зобов'язаний показувати, що складніший алгоритм завжди кращий.

Цілком нормально отримати ситуацію, коли два алгоритми дали однаковий результат.

У такому випадку студент повинен пояснити, чому саме на цих даних різниці немає.

Це також є правильним науковим результатом.

2.5. Відтворюваність результатів

Готовий проєкт повинен бути відтворюваним.

Це означає, що інша людина повинна мати можливість:

1. отримати код;
2. прочитати README;
3. встановити необхідні залежності;
4. запустити програму;
5. запустити тести;
6. отримати той самий результат.

Слова «той самий» тут означають буквальну рівність, а не приблизну схожість. На однакових вхідних даних і однакових значеннях `seed` система повинна давати побайтово однаковий результат від запуску до запуску. Це перевіряється безпосередньо: видані еталонні файли звіряються побайтово.

Єдиний виняток — генетичний алгоритм за різних значень `seed`. Там результати відрізняються, і порівнювати їх треба як статистичні: за середнім і розкидом за кількома запусками, а не за одним числом.

Не повинно бути ситуації:

«На моєму ноутбучі воно працює, але я не знаю чому».

Усі важливі параметри повинні бути зафіксовані.

Про `seed`. Значення `seed` не обирається командою довільно: воно видається викладачем разом із комплектом даних і є індивідуальним для кожної команди. Саме тому числові результати різних команд не збігаються, і саме тому їх не можна просто переписати одна в одній. Якщо в системі використовується власна випадковість, наприклад у генетичному алгоритмі, її `seed` теж має бути зафіксований і записаний у звіт.

Про умови вимірювання. Відтворюваність стосується і даних, і умов. Офіційні вимірювання часу та підсумкової метрики виконуються:

- на одній машині або в одному контейнері для всіх команд, бо інакше швидший ноутбук дає перевагу в оцінці;
- на прихованих значеннях `seed`, які команді не видаються, з усередненням за кількома прогонами.

Відкриті значення `seed` у виданому комплекті призначені для налагодження. Підбирати рішення під конкретний відкритий `seed` безглуздо: на офіційному вимірюванні дані будуть інші.

Так само підсумкова якість класифікатора визначається не на тих даних, які команда бачила під час навчання, а на прихованій вибірці, що зберігається у викладача.

Якщо проводиться експеримент, потрібно записати:

- які дані використовувалися;
- які параметри було встановлено;
- який алгоритм запускався;
- який результат було отримано.

2.6. Очікуваний кінцевий результат

Наприкінці семестру команда повинна мати не просто набір файлів із кодом. Мова реалізації обирається командою і фіксується на Зрізі 0; жодна конкретна мова умовою не нав'язується.

Результатом є завершена, протестована і документована програмна система.

Вона повинна:

- виконувати основну задачу свого варіанта;
- правильно працювати на виданих тестових даних;
- проходити базові приймальні тести;
- коректно обробляти помилкові дані;
- містити реалізації необхідних алгоритмів;
- містити автоматичні тести;
- мати README з інструкцією запуску;
- містити результати експериментів;
- мати підготовлений звіт;
- бути готовою до демонстрації на захисті.

При цьому студент повинен бути готовий пояснити що зроблено і чому це зроблено саме так.

Саме це є головним результатом проєктної роботи з дисципліни «Алгоритми і структури даних».

3. Організація роботи над проєктом і робота в команді

Проєктна робота виконується командою, тому важливо від самого початку правильно організувати спільну роботу.

Головна помилка командних проєктів — ситуація, коли один студент робить майже все, а решта підключається лише наприкінці. Такий підхід поганий не тільки з погляду справедливості. Він ще й робить проєкт нестійким: якщо одна людина не встигає, хворіє або неправильно зрозуміла задачу, зупиняється вся команда.

Але є ще одна причина, суто практична. Оцінювання проєкту здебільшого індивідуальне. Зі ста балів проєкту лише тридцять припадає на спільну якість продукту й однакові для всієї команди. Решта сімдесят — це особистий внесок кожного: дотримання дедлайнів, ведення репозиторію, виконання власної зони, результативність рев'ю, складність зони і персональний захист.

Тому студент, який «був у команді», але не має власної зони, історії комітів і не може відповісти за свою частину на захисті, не отримає балів за чужу роботу. І навпаки: команда не постраждає через те, що хтось один зробив більше.

Тому проєкт потрібно відразу розділити на зрозумілі частини, а кожному учаснику надати власну зону відповідальності.

У більшості випадків команда складається з 4–6 студентів. Для команди з п'яти осіб зручно розподілити роботу приблизно так:

- один студент відповідає за основний алгоритм стиснення або кодування;
- другий — за динамічне програмування та жадібні алгоритми;
- третій — за аналіз даних або перцептрон;

- четвертий — за захист даних і генетичний алгоритм;
- п'ятий — за інтеграцію компонентів, тестування і загальний запуск системи.

Це не є жорстким правилом. Розподіл залежить від конкретного проєкту та складу команди.

Головне правило таке:

У кожного учасника повинна бути власна частина системи, за яку він відповідає і яку він може пояснити на захисті.

3.1. Зона відповідальності

Зона відповідальності — це частина проєкту, яку студент реалізує, тестує, документує і захищає.

Наприклад, зона може називатися:

- «Стиснення»;
- «Пакування» або «Розбір потоку» — залежно від варіанта, саме тут працює динамічне програмування;
- «Аналіз даних» або «Прогноз»;
- «Захист»;
- «Оптимізація»;
- «Інтеграція».

Точні назви зон і їхній зміст наведені в таблиці зон відповідальності кожного варіанта проєкту. Вигадувати власні назви не потрібно: зона обирається з цієї таблиці, бо саме за нею визначається рівень складності й перелік перевірок.

Не рекомендується формулювати зону надто загально.

Наприклад:

«Я відповідаю за Python»

не є нормальною зоною відповідальності.

Так само небажано:

«Я допомагаю всім».

У такому випадку незрозуміло, що саме студент створив самостійно і що можна перевірити на захисті.

Гарна зона має чіткий результат.

Наприклад:

«Я реалізую канонічний алгоритм Гаффмана, кодування і декодування блоків та автоматичні тести для крайніх випадків».

або:

«Я реалізую точний алгоритм динамічного програмування для вибору даних і порівнюю його з жадібним алгоритмом».

Таку роботу легко перевірити.

3.2. Рівень складності зони

Для кожної зони може бути визначено кілька рівнів складності:

- базовий;
- просунутий;
- експертний.

Рівень обирається на початку роботи.

Базовий рівень містить обов'язкову частину, достатню для повноцінної роботи системи.

Просунутий рівень передбачає додатковий алгоритм, складнішу оптимізацію або розширений експеримент.

Експертний рівень містить завдання, які виходять за межі основного курсу або потребують більш глибокого алгоритмічного аналізу.

Рівень визначає стелю оцінки за складність і повноту індивідуальної зони:

Рівень	Що це означає	Стеля оцінки за зону
базовий	зона виконана повністю і якісно	0,8
просунутий	додано механізми просунутого рівня з таблиці зон	0,9
експертний	додано дослідницьку або оптимізаційну складову	1,0

Важливо правильно оцінити свої сили, і тут допомагають самі числа. Повністю виконаний базовий рівень дає 0,8 і не карається. А частково виконана експертна зона оцінюється приблизно в 0,4–0,5, тобто нижче за повністю виконану базову. Отже обирати експертний рівень лише тому, що він виглядає складніше, не вигідно навіть арифметично.

Практична порада: беріть просунутий рівень, якщо впевнені, що закриєте базовий до другого чекпоінту. Експертний має сенс лише за реального запасу часу.

З рівнем пов'язаний і склад перевірок. Приймальні перевірки поділені на базові, просунуті й експертні. Командна оцінка функціональної коректності рахується тільки за базовими, тож повністю базовий проєкт отримує за неї повний бал. Просунуті й експертні перевірки підтверджують рівень конкретної індивідуальної зони.

Рівень складності фіксується на Зрізі 0. Надалі він може бути знижений за погодженням з викладачем на першому чекпоінті з відповідним записом, але не може бути підвищений заднім числом.

3.3. Зріз 0 — старт проєкту

Перший етап роботи називається Зріз 0. Це ще не перевірка програмного коду.

На цьому етапі команда повинна показати, що вона зрозуміла задачу і має реалістичний план її виконання.

До Зрізу 0 команда повинна:

1. обрати варіант проєкту і мову реалізації;
2. прочитати постановку задачі;
3. ознайомитися зі специфікацією формату;
4. переглянути видані дані та еталонні файли;
5. розділити систему на основні модулі;
6. розподілити зони відповідальності;
7. визначити рівень складності кожної зони;
8. зафіксувати склад приймальних перевірок: усі базові плюс ті просунуті й експертні, що відповідають заявленим рівням зон;
9. створити репозиторій;
10. скласти початковий план роботи за чекпоінтами;
11. визначити основні ризики.

На Зрізі 0 викладач оприлюднює банк питань до захисту. Він відкритий від самого початку: підготовка до захисту не є окремою роботою наприкінці семестру, а супроводжує реалізацію.

Наприклад, команда може одразу помітити:

- що формат даних складний;
- що один алгоритм ще ніхто не реалізовував;
- що потрібне точне побайтове відновлення;
- що певна частина залежить від результату іншої.

Це і є ризики, які потрібно зафіксувати.

Важливо:

Зріз 0 — це не формальність. Добре зроблений план на початку економить багато часу наприкінці.

3.4. Репозиторій проєкту

З першого тижня проєкт повинен зберігатися в системі контролю версій.

Рекомендовано використовувати Git.

Репозиторій повинен бути зрозумілим авторам, викладачу і будь-якій іншій команді.

Приблизна структура може виглядати так:

```
project/  
└─ src/
```

```
├── tests/
├── experiments/
├── docs/
├── README.md
├── META-USAGE.md
└── requirements.txt
```

Конкретна структура може відрізнятися.

Наприклад, у Python-проєкті може використовуватися `pyproject.toml` замість `requirements.txt`.

Головне — щоб код, тести, експерименти і документація не були змішані в одному каталозі.

Не рекомендується структура:

```
project/
├── final.py
├── final2.py
├── final_new.py
├── final_really_final.py
├── test.py
├── test2.py
└── data123.txt
```

Через кілька тижнів навіть автори такого проєкту перестають розуміти, який файл є актуальним.

Вимоги до ведення репозиторію

Ведення репозиторію оцінюється окремо, за п'ятьма пунктами:

1. щонайменше один змістовний коміт на тиждень від кожного учасника;
2. коміти прив'язані до issue;
3. README з інструкцією запуску, за якою проєкт стартує з нуля;
4. лог рішень — не менше п'яти записів;
5. розробка не ведеться безпосередньо в `main`.

Лог рішень

Це файл у каталозі `docs/`, у якому фіксуються прийняті технічні рішення у форматі «проблема → варіанти → обране рішення → чому». Наприклад:

Проблема. Точний розбір працює 40 секунд на файлі 10 МБ, це не вкладається в ліміт. Варіанти. (1) обмежити глибину пошуку збігів; (2) простіша евристика замість точного алгоритму; (3) розбити дані на вікна і застосувати точний алгоритм у межах вікна. Рішення. Варіант 3, вікно 1 МБ. Чому. Втрата якості 0,4 %, прискорення в 12 разів. Варіант 2 давав втрату 2,1 %.

Лог рішень не є формальністю. Він виконує одразу три функції: закриває пункт вимог до репозиторію, дає готові відповіді на захисті і є доказом самостійності роботи, бо містить помилки і тупики, яких у запозиченому коді не буває.

3.5. Робота з гілками

Кожен учасник команди повинен працювати у своїй гілці.

Наприклад:

```
main
feature/huffman
feature/dp
feature/mlp
feature/security
feature/integration
```

Гілка `main` повинна містити тільки версію, яка проходить основні тести.

Не рекомендується безпосередньо змінювати `main`, особливо коли над проєктом одночасно працює кілька людей.

Типовий порядок роботи:

```
створити issue
→ створити гілку
→ написати код
→ додати тести
→ зробити commit
→ створити pull request
→ пройти review
→ merge у main
```

Такий процес спочатку здається повільнішим, ніж просто пересилати файли в Telegram.

Насправді через кілька тижнів він економить значно більше часу.

3.6. Issue як маленька задача

Велику задачу потрібно розбивати на маленькі.

Наприклад, не варто створювати issue:

«Зробити стиснення».

Це надто велика задача.

Краще:

- реалізувати побудову частот символів;
- реалізувати дерево Гаффмана;
- отримати довжини кодів;

- побудувати канонічні коди;
- реалізувати encoder;
- реалізувати decoder;
- додати тест для одного символу;
- додати тест для порожнього входу.

Кожен такий пункт має зрозумілий результат.

Це дозволяє бачити реальний прогрес проєкту.

3.7. Commit

Commit повинен відповідати одній зрозумілій зміні.

Гарні повідомлення:

`add canonical Huffman encoder`

`fix empty block decoding`

`add DP vs greedy benchmark`

`validate HMAC before decryption`

Погані повідомлення:

`fix`

`new`

`111`

`final`

`final_final`

Кожен commit має бути прив'язаний до issue: у більшості систем для цього достатньо вказати номер задачі в повідомленні, наприклад `add canonical Huffman encoder (#14)`.

За історією commit повинно бути можливо зрозуміти, як розвивався проєкт.

Не варто робити один commit наприкінці семестру з усім проєктом.

У такому випадку неможливо побачити процес роботи.

3.8. Code review

Перед об'єднанням великої зміни її повинен переглянути інший член команди. Це не побажання: результативність рев'ю оцінюється окремо.

Зараховуються лише результативні рев'ю: знайдено суттєву проблему або запропоновано обґрунтоване покращення, автор відреагував, виправлення

перевірено. Три таких рев'ю дають повний бал. Косметичні зауваження про пробіли і назви змінних не рахуються.

Практично це означає, що кожен учасник має розібратися хоча б у двох чужих зонах. Зручний розподіл — рев'ю за колом: кожен перевіряє зміни сусіда за таблицею зон.

Review не означає:

«Я подивився, все нормально».

Потрібно перевірити хоча б кілька речей:

- чи зрозумілий код;
- чи немає очевидних помилок;
- чи є тести;
- чи правильно обробляються крайні випадки;
- чи не порушено формат даних;
- чи не дублюється вже існуючий код.

Особливо корисно, коли компонент перевіряє студент, який його не писав.

Автор часто знає, як його програма «повинна працювати», тому може не помітити очевидної для іншої людини проблеми.

3.9. Інтеграція компонентів

Командний проєкт — це не набір п'яти незалежних лабораторних робіт.

Наприкінці всі модулі повинні працювати разом.

Тому інтеграцію не можна залишати на останній тиждень.

Наприклад, одна частина системи може повертати:

```
list[int]
```

а інша очікувати:

```
bytes
```

Обидві частини окремо працюють правильно, але разом — ні.

Щоб цього не сталося, необхідно рано погодити інтерфейси між модулями.

Наприклад:

```
compress(data: bytes) -> bytes
```

```
decompress(data: bytes) -> bytes
```

або:

```
select(items, budget) -> list[int]
```

Інтерфейс повинен бути максимально простим.

Чим менше один модуль знає про внутрішню реалізацію іншого, тим легше їх поєднати.

3.10. Хто відповідає за інтеграцію

Добре, якщо одна людина має окрему зону «Інтеграція».

Але це не означає, що вона повинна самостійно виправляти весь чужий код.

Її завдання:

- визначити спільні інтерфейси;
- стежити за сумісністю компонентів;
- підтримувати інтеграційні тести;
- збирати наскрізний тракт системи;
- стежити, щоб `main` залишався робочим;
- перевіряти, що система правильно читає видані еталонні файли, а не лише власні дані.

Останній пункт важливий: система, яка розуміє тільки те, що сама створила, формату не відповідає. Саме читання еталонних файлів є базовою перевіркою, а взаємна сумісність із системою іншої команди — просунутим рівнем зони «Інтеграція».

Якщо модуль Гаффмана не проходить свої тести, виправляти його повинен автор модуля Гаффмана.

Інтегратор не повинен перетворюватися на людину, яка наприкінці переписує весь проєкт.

3.11. Комунікація в команді

Команда повинна регулярно коротко обговорювати стан роботи.

Не обов'язково проводити довгі формальні зустрічі.

Достатньо, щоб кожен учасник міг відповісти на три питання:

1. Що я зробив?
2. Що я роблю зараз?
3. Що мені заважає?

Наприклад:

Реалізував `canonical Huffman` і три тести. Зараз пишу `decoder`.

Проблема — не до кінця зрозумів вироджений випадок одного символу.

Це гарне повідомлення.

А повідомлення:

«Я ще роблю»

майже не дає інформації команді.

Якщо проблема блокує роботу, про неї потрібно повідомляти одразу.

Не потрібно чекати три тижні, а потім на чекпоінті казати:

«У мене не працювало, але я думав, що сам розберусь».

Своєчасне повідомлення про ризик є нормальною частиною командної роботи.

3.12. Як працювати з помилками

Помилка в програмі — це нормальна частина розробки.

Погано не те, що помилка виникла.

Погано, якщо команда:

- приховує її;
- не може відтворити;
- не знає, коли вона з'явилася;
- виправляє її випадковими змінами;
- не додає тест після виправлення.

Правильний процес:

помилка знайдена

- створити мінімальний приклад
- додати тест, який відтворює помилку
- виправити код
- переконатися, що тест проходить
- переконатися, що старі тести не зламалися

Це один із найважливіших практичних навиків, які потрібно отримати під час проєкту.

3.13. Що робити, якщо студент не встигає

Якщо учасник бачить, що його зона виявилася занадто складною, потрібно повідомити про це команду і викладача якомога раніше.

Можливі рішення:

- знизити заявлений рівень зони до базового;
- перенести частину роботи;
- розбити задачу на менші;
- попросити іншу людину провести review;
- змінити внутрішній план команди.

Зниження рівня оформлюється записом за погодженням з викладачем і робиться до другого чекпоінту. Це нормальний робочий крок, а не покарання: повністю виконана базова зона оцінюється вище, ніж наполовину виконана експертна.

Найгірший варіант — нічого не говорити до останнього тижня.

У проєктній роботі важливо виконувати задачі і вчасно повідомляти про ризики.

3.14. Використання штучного інтелекту

Якщо під час роботи використовуються системи штучного інтелекту, це потрібно робити усвідомлено.

ШІ може допомогти:

- пояснити алгоритм;
- запропонувати тест;
- знайти можливий крайній випадок;
- допомогти зрозуміти повідомлення про помилку;
- перевірити документацію;
- запропонувати варіант структури коду.

Але студент повинен розуміти код, який потрапляє до репозиторію.

На захисті відповідь:

«Це написав ChatGPT, я не знаю, як воно працює»

означає, що студент фактично не виконав цю частину роботи.

Тому діє просте правило:

Можна використовувати інструмент, але не можна передавати йому відповідальність за розуміння власного проєкту.

Використання таких інструментів дозволене за трьох обов'язкових умов.

Умова 1. Декларація. У репозиторії ведеться файл `META-USAGE.md`, де записуються застосовані інструменти, ключові запити до них, отримані фрагменти, внесені зміни і спосіб перевірки результату.

Умова 2. Володіння стратегією. На захисті студент пояснює, чому обрав саме такий спосіб застосування інструмента, і порівнює його з альтернативами, що впливають із матеріалу дисципліни.

Умова 3. Розуміння отриманого. Студент пояснює зміст кожного згенерованого фрагмента, як перевіряв його коректність і складність, і як цей фрагмент можна використати далі.

Наслідки недотримання серйозні. Академічна доброчесність діє як множник до підсумкової оцінки проєкту, а не як окремий критерій, тому високі бали за інші пункти її не компенсують. Недекларований згенерований код прирівнюється до запозичення без атрибуції і зменшує підсумок удвічі. Видавання чужої роботи за власну обнуляє проєкт.

Запис у `META-USAGE.md` варто робити одразу, а не наприкінці семестру. Файл, заповнений в останній день одним рядком, не дозволяє виконати умови 2 і 3 і сам по собі знижує оцінку за захист.

3.15. Що повинно бути готово після організаційного етапу

Після Зрізу 0 команда ще не зобов'язана мати готову програму.

Але вона повинна мати чітку основу для подальшої роботи.

До цього моменту мають бути готові:

- репозиторій;
- склад команди;
- зони відповідальності;
- рівні складності;
- список основних модулів;
- погоджені інтерфейси між ключовими модулями;
- початкові issue;
- календарний план за чекпоінтами;
- список ризиків;
- зафіксований склад приймальних перевірок;
- обрана мова реалізації;
- базовий README.

Після цього можна переходити до реалізації.

І тут починається наступний важливий етап:

не будувати одразу всю систему, а створити найпростішу робочу версію і поступово її ускладнювати.

4. Алгоритми і структури даних, які використовуються у проєктах

У трьох проєктах використовуються різні алгоритми, але важливо побачити між ними спільну логіку.

Ми не вчимо алгоритм заради самого алгоритму.

Нас цікавить інше:

Яку конкретну проблему в системі цей алгоритм розв'язує?

Наприклад, черга з пріоритетами потрібна не тому, що вона є у програмі курсу, а тому, що під час побудови дерева Гаффмана нам постійно потрібно швидко брати два найменш частотні елементи.

Динамічне програмування потрібне не тому, що це «важлива тема», а тому, що в задачі є багато варіантів вибору, і потрібно знайти найкращий.

Жадібний алгоритм потрібен не лише як окремий метод, а як простіший конкурент точному алгоритму.

Генетичний алгоритм використовується там, де параметрів багато і повний перебір стає занадто дорогим.

Перцептрон потрібен для того, щоб система могла автоматично робити вибір на основі даних.

Тому в цьому розділі кожен алгоритм будемо розглядати за простою схемою:

що це таке

- яку задачу розв'язує
- як працює
- де використовується у проєктах
- що саме має реалізувати студент

4.1. Масиви, списки, словники та множини

Перед тим як переходити до складніших алгоритмів, потрібно правильно використовувати базові структури даних.

Часто помилка в проєкті виникає не через неправильний складний алгоритм, а через невдалий вибір структури даних.

Масив або список

Список зручно використовувати, коли дані потрібно зберігати у певному порядку.

Наприклад:

- послідовність блоків файла;
- послідовність подій журналу;
- список пакетів;
- список кандидатів для оптимізації;
- історія значень fitness у генетичному алгоритмі.

Якщо нам часто потрібен елемент за номером, список є природним вибором.

Наприклад:

```
blocks[i]  
events[i]  
population[i]
```

Словник

Словник використовується тоді, коли потрібно швидко знайти значення за ключем.

Наприклад:

```
frequency[byte]  
file_by_name[name]  
node_by_id[id]  
code_length[symbol]
```

Замість того щоб щоразу проходити весь список у пошуку потрібного елемента, можна отримати його майже відразу за ключем.

Множина

Множина корисна, коли потрібно швидко перевірити:

Чи вже зустрічався цей елемент?

Наприклад:

- чи був уже використаний попсе;
- чи є дубльований номер блока;
- чи повторюється індекс тому;
- чи вже оброблена подія;
- чи є дубльований символ у таблиці кодів.

Це особливо корисно для перевірки коректності даних.

4.2. Черга з пріоритетами

Черга з пріоритетами схожа на звичайну чергу, але елементи виходять із неї не в порядку надходження.

Першим вибирається елемент із найкращим пріоритетом.

Наприклад, якщо маємо:

A: 15
B: 3
C: 8
D: 1

і менше число означає вищий пріоритет, порядок вилучення буде:

D
B
C
A

Для ефективної реалізації черги з пріоритетами часто використовується бінарна купа.

Основні операції:

<code>insert(x)</code>	додати елемент
<code>extract_min()</code>	забрати найменший
<code>peek()</code>	подивитися найменший

Для купи основні операції працюють приблизно за:
 $O(\log n)$

Де це потрібно у проєктах

Найважливіше застосування — алгоритм Гаффмана.

Під час побудови дерева ми багато разів виконуємо одну й ту саму операцію:

1. знайти два символи з найменшою частотою;
2. об'єднати їх;

3. повернути новий вузол назад;
4. повторити.

Саме для цього дуже добре підходить черга з пріоритетами.

Що повинен розуміти студент

Потрібно вміти пояснити:

- чому звичайний список тут менш зручний;
- чому купа дає кращу складність;
- які операції виконуються найчастіше;
- яка загальна складність побудови дерева.

4.3. Жадібні алгоритми

Жадібний алгоритм на кожному кроці робить вибір, який здається найкращим прямо зараз.

Він не перебирає всі можливі майбутні варіанти.

Наприклад, якщо потрібно взяти предмети в рюкзак, жадібний алгоритм може щоразу брати предмет із найбільшим відношенням:
цінність / вага

Це швидко.

Але не завжди оптимально.

Простий приклад

Нехай місткість рюкзака:

$$W = 10$$

Предмети:

A: вага 6, цінність 13	відношення 2,17
B: вага 5, цінність 10	відношення 2,00
C: вага 5, цінність 10	відношення 2,00

Жадібний алгоритм бере A, бо в нього найбільше відношення цінності до ваги.

Після цього залишається місткість 4, і ні B, ні C уже не вміщуються:
цінність = 13

Але оптимальний вибір — узяти B і C:

$$5 + 5 = 10 \leq 10$$

цінність = 20

Тобто локально хороший вибір не обов'язково дає глобально найкращий результат.

Зверніть увагу, що відношення в цьому прикладі різні. Це важливо: якби всі три предмети мали однакове відношення, результат жадібного алгоритму залежав би від довільного порядку сортування, і приклад нічого не доводив би.

Чому тоді жадібні алгоритми потрібні

Тому що вони часто:

- прості;
- швидкі;
- потребують мало пам'яті;
- дають хороший результат.

У проєктах жадібний алгоритм дуже часто використовується як baseline.

Тобто як проста початкова стратегія, з якою потім порівнюється складніший алгоритм.

Де використовується

У Проєкті 1 жадібний алгоритм може визначати порядок або вибір даних для передавання.

У Проєкті 2:

- жадібний LZ77 бере найдовший збіг;
- жадібний рюкзак сортує блоки за відношенням цінності до ваги.

У Проєкті 3 жадібно можна вибирати наступну копію події за найбільшим відношенням:

приріст корисності / вартість

Головне завдання студента

Не просто реалізувати greedy.

Потрібно порівняти його з точним алгоритмом і відповісти:

Де greedy програє?

і:

Де він дає майже такий самий результат, але значно швидше?

4.4. Динамічне програмування

Динамічне програмування використовується тоді, коли складну задачу можна розбити на багато повторюваних підзадач.

Основна ідея:

Не розв'язувати одну й ту саму підзадачу багато разів.

Замість цього її результат зберігається.

Приклад

У задачі рюкзака можна визначити:
 $DP[i][w]$

як найкращу цінність, яку можна отримати, використовуючи перші i елементів і місткість w .

Тоді для кожного елемента є два варіанти:
не брати його

або:
взяти його

і вибрати кращий результат.

Спрощено:
 $DP[i][w] =$
 $\max(\$
 $DP[i-1][w],$
 $value[i] + DP[i-1][w-weight[i]]$
 $\)$

якщо елемент вміщується.

Чому це точний алгоритм

На відміну від greedy, динамічне програмування розглядає всі суттєві варіанти станів.

Тому для правильно сформульованої задачі воно знаходить оптимум.

Ціна цього — більша кількість обчислень і пам'яті.

Де використовується у проєктах

Проєкт 1

ДП допомагає вибрати, які телеметричні дані передати через обмежений канал.

Є:
цінність
вартість
обмежений бюджет

Це природна задача типу рюкзака.

Проєкт 2

ДП використовується у двох різних місцях.

Перше — пакування блоків у том архіву.

Потрібно вибрати блоки максимальної сумарної цінності при обмеженій місткості тому.

Друге, на просунутому рівні, — оптимальний LZ77-розбір.

Тут стан можна визначити як:

$C[i]$

— мінімальна вартість закодувати дані від позиції i до кінця.

Проект 3

Для кожної події можна обрати:

0 копій

1 копію

2 копії

3 копії

Кожен варіант має:

вартість

корисність

i є загальний бюджет.

Це багатоваріантний рюкзак.

Що повинен уміти студент

Студент повинен вміти:

1. визначити стан;
2. визначити переходи;
3. записати базовий випадок;
4. пояснити порядок заповнення таблиці;
5. відновити саме рішення, а не тільки його цінність;
6. оцінити складність.

Особливо важливо:

Перед написанням коду потрібно спочатку словами пояснити, що означає кожен стан DP.

Якщо студент не може пояснити, що означає:

$DP[i][w]$

то майже напевно реалізація буде складною для налагодження.

4.5. Рюкзак 0/1

Задача рюкзака є одним із центральних прикладів динамічного програмування у цих проектах.

Є набір елементів.

Для кожного:

вага w_i

цінність v_i

Є місткість:

W

Потрібно максимізувати:

$\sum v_i$

за умови:

$\sum w_i \leq W$

У варіанті 0/1 кожен елемент можна:

або взяти один раз

або не взяти

Чому ця модель така корисна

Багато реальних задач можна записати саме так.

Наприклад:

Чи варто передати цей пакет?

Чи варто покласти цей блок у перший том?

Чи варто створити ще одну копію важливої події?

Тобто «предмет» не обов'язково є фізичним предметом.

Це може бути:

- пакет;
- файл;
- блок;
- подія;
- репліка.

4.6. Багатоваріантний рюкзак

У Проєкті 3 використовується трохи складніша версія.

Для однієї події можна вибрати не лише:

0 або 1

а:

0

1

2

3 копії

Для кожного варіанта є своя вартість і корисність.

Наприклад:

0 копій $\rightarrow$ 0

1 копія $\rightarrow$ 10

2 копії $\rightarrow$ 15

3 копії $\rightarrow$ 18

Корисність зростає, але кожна наступна копія дає менший додатковий виграш.

Це називається зменшуваною віддачею.

Ми хочемо уникнути ситуації, коли система витратить весь бюджет на одну подію.

Саме тому друга і третя копії мають меншу додаткову цінність.

4.7. Алгоритм Гаффмана

Алгоритм Гаффмана будує префіксний код, у якому часті символи зазвичай отримують коротші коди, а рідкі — довші.

Наприклад:

A зустрічається дуже часто

B рідше

C дуже рідко

можна отримати:

A $\rightarrow$ 0

B $\rightarrow$ 10

C $\rightarrow$ 11

Префіксність

Важлива властивість:

жоден код не є початком іншого.

Наприклад:

0

10

11

можна однозначно декодувати.

A набір:

0
01
11

поганий, бо 0 є префіксом 01.

Як будується дерево

1. порахувати частоти;
2. створити вузол для кожного символу;
3. покласти вузли в чергу з пріоритетами;
4. взяти два найменші;
5. об'єднати;
6. повернути новий вузол;
7. повторювати, поки не залишиться один корінь.

Саме тут використовується купа.

Складність

Якщо маємо k різних символів:

$O(k \log k)$

для побудови дерева через priority queue.

4.8. Канонічний Гаффман

У реальному форматі незручно зберігати саме дерево.

Замість цього можна зберігати тільки довжини кодів.

Наприклад:

A → 1
B → 2
C → 3
D → 3

Якщо всі використовують один і той самий канонічний алгоритм, самі біти кодів можна відновити однозначно:

A → 0
B → 10
C → 110
D → 111

Це і є канонічний Гаффман.

Повнота коду

Набір довжин не може бути довільним. Він має задовольняти рівність Крафта точно:

$$\sum 2^{(-L)} = 1$$

Для наведеного прикладу: $1/2 + 1/4 + 1/8 + 1/8 = 1$. Усе гаразд.

А набір $A \rightarrow 2, B \rightarrow 3, C \rightarrow 3, D \rightarrow 3$ дає $1/4 + 3/8 = 0,625$. Це означає, що частина кодового простору не використана, дерево неповне, і строгий parser такий набір відхилить. Єдиний виняток — вироджений випадок з одним символом, описаний нижче.

Навіщо це потрібно

Менше службових даних.

Також дві різні реалізації можуть читати файли одна одною.

Важливий крайній випадок

Якщо є тільки один символ, дерево формально вироджується.

У наших форматах домовляємося:

довжина = 1

код = 0

Це потрібно перевіряти окремим тестом.

Де використовується

У всіх проектах, де потрібно стискати або компактно кодувати потоки даних.

Особливо важливий у Проєкті 2, де Гаффман працює після LZ77.

4.9. LZ77

LZ77 використовує повтори в уже прочитаній частині даних.

Замість того щоб багато разів записувати один і той самий фрагмент, можна сказати:

Повернися назад на d байтів і скопіюй 1 байтів.

Наприклад:

abcabcabcabc

можна частково описати як:

abc

назад на 3, взяти 9

У цьому прикладі відстань $d = 3$ менша за довжину $l = 9$. Це нормально і навіть типово: посилання може перекриватися з даними, які воно копіює.

Але звідси випливає обов'язкова вимога до декодера: копіювати треба побайтово, по одному байту за раз, дописуючи кожен байт у вихідний буфер одразу. Спроба скопіювати весь фрагмент одним блоком дасть неправильний результат саме на перекритті. Це одна з найпоширеніших помилок у реалізаціях LZ77, і на неї варто мати окремий тест.

Елемент потоку

У спрощеному вигляді LZ77 створює два типи елементів:
літерал

або:
посилання (distance, length)

Greedy LZ77

Найпростіша стратегія:

У кожній позиції взяти найдовший доступний збіг.

Це швидко, але не завжди дає найкращий загальний код.

Lazy greedy

Іноді корисно подивитися на одну позицію вперед.

Якщо наступна позиція дає значно кращий збіг, зараз можна записати один літерал і почати посилання пізніше.

ДП-розбір

Ще сильніший варіант — розглядати всі корисні збіги й вибирати глобально найкращу послідовність літералів і посилань.

Це вже задача динамічного програмування.

Де використовується

LZ77 є центральним алгоритмом стиснення в Проєкті 2.

4.10. Хеш-таблиці та пошук збігів

Наївний LZ77 може бути дуже повільним.

У кожній позиції можна переглядати багато попередніх байтів у пошуку збігів.

Для прискорення використовуються структури даних для пошуку.

На базовому рівні можна використовувати словники або хеші коротких послідовностей.

Наприклад, для кожних трьох байтів:

abc

зберігати список попередніх позицій, де вони зустрічалися.

Тоді не потрібно перевіряти все вікно.

Можна одразу перейти до кількох кандидатів.

На експертному рівні можливі складніші структури:

- суфіксний масив;
- суфіксне дерево;
- суфіксний автомат.

Але вони не є обов'язковими для базового проєкту.

4.11. Перцептрон

Перцептрон у цих проєктах використовується не як окрема «магічна нейромережа», а як алгоритм прийняття рішення за ознаками.

Наприклад, маємо ознаки файла:

ентропія

частка текстових символів

автокореляція

гістограма

довжина рядків

На основі цих значень потрібно визначити:

текст

код

структуровані дані

растр

виконуваний файл

вже стиснуті дані

...

Нейрон

У найпростішому вигляді:

$$z = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

Потім застосовується функція активації.

У багатошаровій мережі вихід одного шару стає входом наступного.

Навчання

Типовий цикл:

forward

→ обчислити помилку

→ backpropagation

→ змінити ваги

→ повторити

Навіщо перцептрон у проєкті

Щоб система могла автоматично вибирати стратегію.

Наприклад:

якщо дані вже схожі на стиснуті

→ не стискати

якщо файл текстовий

→ дозволити певні параметри

якщо подія аномальна

→ дати їй більший пріоритет

Важливе правило

У межах навчального проєкту основна логіка перцептрона реалізується самостійно.

Тобто недостатньо викликати готовий `MLPClassifier`.

Студент повинен розуміти:

- forward pass;
- функцію втрат;
- backpropagation;
- оновлення ваг;
- train/valid/holdout.

4.12. Train, validation i holdout

Щоб чесно оцінити модель, дані розділяються.

`train`

→ навчання

`validation`

→ вибір параметрів

`holdout`

→ фінальна оцінка

У цих проєктах `holdout` студентів не видається взагалі. Команда отримує `train` і відкритий `valid`, а прихована вибірка зберігається у викладача і використовується один раз для підсумкової оцінки.

Так зроблено навмисно. Правило «використати тестові дані лише раз» неможливо проконтролювати: якщо файл із мітками лежить у команди, ніщо не заважає сто разів підбирати модель під нього. Тоді результат перестає бути

чесною оцінкою якості й показує лише те, наскільки добре модель запам'ятала конкретні дані.

Під час розробки роль «фінальної перевірки» виконує `valid`, і саме за ним обирається модель.

4.13. Macro-F1

Для класифікатора недостатньо лише асигурації.

Особливо якщо важливо добре розпізнавати всі класи.

Для кожного класу обчислюються `precision` і `recall`.

Потім:

$$F1 = 2 \cdot \text{precision} \cdot \text{recall} / (\text{precision} + \text{recall})$$

Якщо `precision + recall = 0`, тобто клас жодного разу не був передбачений і жодного разу не вгаданий, за угодою вважається `F1 = 0`. Це не виняткова ситуація: саме так поводить себе класифікатор, який ігнорує рідкісний клас, і саме за це `macro-F1` його карає.

`Macro-F1` — це середнє `F1` для всіх класів:

`Macro-F1 =`

$$(F1_1 + F1_2 + \dots + F1_k) / k$$

Тобто кожен клас має однакову вагу.

Це не дозволяє отримати хороший результат просто за рахунок одного великого класу.

4.14. Генетичний алгоритм

Генетичний алгоритм використовується для пошуку хороших параметрів системи.

Наприклад, потрібно підібрати:

розмір вікна пошуку

мінімальну довжину збігу

розмір блока

поріг відмови від стиснення

Повний перебір усіх комбінацій може бути занадто великим.

Тоді кожен набір параметрів розглядається як хромосома.

На базовому рівні достатньо трьох параметрів, наприклад:

[32768, 3, 16384]

↑ ↑ ↑
| | розмір блока
| мінімальна довжина збігу
розмір вікна пошуку

На просунутому рівні хромосома розширюється: до неї додаються архітектура мережі, глибина пошуку збігів і прапорці стратегій за типами даних.

Популяція

Створюємо багато хромосом.

Для кожної обчислюємо:

fitness

тобто наскільки добре працює система з такими параметрами.

Потім:

1. вибираємо кращих;
2. схрещуємо;
3. мутуємо;
4. створюємо нове покоління;
5. повторюємо.

Основні операції

Студент повинен розуміти:

- представлення хромосоми;
- функцію fitness;
- selection;
- crossover;
- mutation;
- elitism.

Елітизм

Кілька найкращих рішень переходять у наступне покоління без змін.

Тому найкращий fitness не повинен ставати гіршим.

Це дуже зручний тест коректності.

З чим порівнювати

Генетичний алгоритм обов'язково потрібно порівняти з random search при однаковій кількості обчислень fitness.

Інакше не можна зрозуміти, чи справді ГА дав користь.

4.15. Хеш-функції, CRC і HMAC

Ці три поняття схожі тим, що створюють коротке значення з великих даних.

Але призначення різне.

CRC

CRC добре знаходить випадкові помилки.

Наприклад:
пошкодився файл
→ CRC змінився

Але CRC не є криптографічним захистом.
Зловмисник може змінити файл і перерахувати CRC.

Криптографічний хеш

Хеш типу SHA-256 використовується там, де потрібне стійке криптографічне відображення даних.

НМАС

НМАС використовує секретний ключ.

Тому зловмисник без ключа не може просто змінити дані й створити правильний новий тег.

У проєктах НМАС використовується для перевірки автентичності та цілісності.

4.16. AES-CTR

AES — блочний симетричний шифр.

У режимі CTR він використовується як генератор потоку ключових байтів.

Ідея:

counter

→ AES

→ псевдовипадковий блок

→ XOR з даними

Для наступного блока counter змінюється.

Дуже важливе правило

Для одного ключа не можна повторно використовувати той самий nonce.

Якщо:

key однаковий

nonce однаковий

то створюється той самий keystream.

Це серйозна криптографічна помилка.

Тому в проєктах nonce повинен бути унікальним відповідно до правил конкретної специфікації: у Проєкті 1 — для кожного пакета, у Проєкті 2 — для кожного тому.

Ще одна деталь, у якій легко помилитися: специфікації задають нумерацію блоків лічильника, і вона починається з одиниці, а не з нуля. Помилка тут не

завадить власній системі працювати сама із собою, але зробить її несумісною з еталонними файлами.

4.17. KDF і розділення ключів

Пароль користувача не варто напряду використовувати як AES-ключ.

Для цього застосовується KDF.

У проєктах використовується PBKDF2-HMAC-SHA256.

З пароля і salt отримується довгий ключовий матеріал.

Потім він розділяється:

`K_enc`

`K_mac`

Один ключ використовується для AES.

Другий — для HMAC.

Не потрібно використовувати один і той самий ключ для обох задач.

KDF виконується один раз

PBKDF2 навмисно зроблений повільним: у наших специфікаціях це 100 000 ітерацій. Саме повільність і захищає від перебору паролів.

Але сіль однакова для всього сеансу або всього архіву, отже ключі теж однакові. Тому KDF потрібно виконати один раз на початку і зберегти отримані ключі, а не перераховувати їх для кожного пакета чи тому.

Якщо цього не зробити, система формально працюватиме правильно, але KDF почне домінувати у часі виконання і зіпсує вимірювання швидкодії, яке входить до підсумкової метрики.

4.18. Encrypt-then-MAC

У захищеному форматі важливий порядок операцій.

Правильна загальна схема:

дані

→ стиснення

→ шифрування

→ HMAC

Під час читання:

отримали дані

→ перевірили HMAC

→ тільки після цього розшифрували

Причина проста. Якщо HMAC неправильний, дані вже вважаються недовіреними.

Немає сенсу передавати їх у складний parser або decoder.

Це одночасно і правильна криптографічна схема, і хороша інженерна практика.

Чому стиснення йде перед шифруванням

Порядок «стиснути, потім зашифрувати» теж не довільний. Добре зашифровані дані статистично не відрізняються від випадкових: у них немає ні повторів для LZ77, ні перекошеного розподілу байтів для Гаффмана. Тому стиснути шифротекст практично неможливо — коефіцієнт стиснення буде близький до одиниці.

Це не теоретичне міркування, а вимірюваний факт, і в проєкті передбачено відповідний експеримент: обидва порядки застосовуються до тих самих даних, а результат наводиться в таблиці звіту.

4.19. Канонічний формат і строгий parser

У всіх трьох проєктах результат залежить від алгоритму і від точного формату даних.

Parser повинен бути строгим.

Він не повинен думати:

«Мабуть, автор мав на увазі ось це».

Якщо специфікація каже:

`reserved = 0`

а отримано:

`reserved = 1`

файл потрібно відхилити.

Якщо довжина не збігається — відхилити.

Якщо `code length` неправильна — відхилити.

Якщо НМАС неправильний — відхилити.

Такий підхід називають:

`fail closed`

тобто за сумнівних даних система відмовляється від роботи, а не намагається «вгадати».

4.20. Алгоритм як частина системи

Найважливіша ідея цього розділу полягає в тому, що алгоритми не існують окремо.

Наприклад, у Проєкті 2 маємо:

дані

→ LZ77

→ Huffman

→ пакування

→ AES

→ HMAC

Якщо LZ77 працює неправильно, Гаффман може ідеально стиснути неправильний потік.

Якщо Huffman правильний, але format writer записує неправильні довжини, архів не відкриється.

Якщо стиснення правильне, але потім повторився, система функціонально працює, але криптографічно небезпечна.

Тому кожен алгоритм потрібно перевіряти двома способами:
окремо

i:

у складі всієї системи

4.21. Що потрібно знати перед переходом до реалізації

Перед тим як починати програмування основної системи, студент повинен уміти відповісти хоча б на такі питання:

Про структури даних

- Де потрібен список?
- Де словник?
- Де множина?
- Де priority queue?

Про greedy

- Який локальний вибір робиться?
- Чому він може бути неоптимальним?
- З чим його будемо порівнювати?

Про DP

- Що означає стан?
- Який перехід?
- Який базовий випадок?
- Яка складність?

Про Huffman

- Чому часті символи мають коротші коди?
- Навіщо priority queue?
- Що таке canonical Huffman?

- Що робити з одним символом?
Про LZ77
- Що таке distance і length?
- Чим greedy відрізняється від lazy?
- Чому DP-розбір може бути кращим?
Про MLP
- Які ознаки подаються на вхід?
- Що прогнозує модель?
- Для чого потрібні train, valid і holdout?
Про GA
- Що є хромосомою?
- Що є fitness?
- Як працюють mutation і crossover?
- Для чого elitism?
Про захист
- Для чого НМАС?
- Чим CRC відрізняється від НМАС?
- Для чого salt?
- Чому поперше не можна повторювати?
- Чому НМАС перевіряється до розшифрування?

Якщо ці відповіді зрозумілі, можна переходити до наступного етапу — побудови проєкту крок за кроком.

5. Загальна методика виконання проєкту

Робота над проєктом стає значно простішою, якщо не намагатися одразу побудувати всю систему.

Найгірша стратегія — відкрити умову, побачити в ній багато алгоритмів і почати одночасно реалізовувати все:

Гаффман

- + динамічне програмування
- + перцептрон
- + шифрування
- + генетичний алгоритм
- + формат
- + інтеграція

У такому випадку дуже швидко стає незрозуміло:

- що вже працює;
- що ще не працює;

- де саме з'явилася помилка;
 - який компонент зламав інший.
- Тому проєкт потрібно будувати шарами.
Основний принцип такий:

Спочатку найпростіша правильна система. Потім — поступове ускладнення.

Усі три проєкти виконуються за приблизно однаковою логікою.

1. зрозуміти формат і дані
2. зробити мінімальний round-trip
3. додати базові алгоритми
4. протестувати їх окремо
5. інтегрувати
6. додати оптимізацію
7. додати ML / GA
8. додати захист
9. провести експерименти
10. довести систему до фінальної версії

5.1. Крок 1. Прочитати умову повністю

Перед написанням першого рядка коду потрібно прочитати всю постановку задачі.

Не тільки вступ.

Не тільки свою зону.

Не тільки те, що «схоже на мою частину».

Потрібно побачити систему цілком.

Під час першого читання не треба намагатися запам'ятати всі деталі.

Достатньо відповісти на п'ять питань:

1. Що приходить на вхід?
2. Що повинно бути на виході?
3. Які основні етапи обробки?
4. Які алгоритми обов'язкові?
5. Як буде перевірятися правильність?

Наприклад, для архіватора:

вхід:

набір файлів

вихід:

захищений архів

після розпакування:
ті самі файли побайтово

Для системи реплікації:
вхід:
потік подій

вихід:
зашифровані репліки на вузлах

після відмов:
важливі події повинні відновлюватися

Коли ця схема зрозуміла, можна переходити далі.

5.2. Крок 2. Окремо прочитати специфікацію формату

У проєктах є точна специфікація формату.

Її потрібно сприймати не як довідковий текст, а як контракт.

Наприклад, якщо `FORMAT_SPEC` каже:

поле займає 2 байти
`big-endian`

не можна записати:
4 байти
`little-endian`

і сказати:

«Але мій decoder це читає».

Ваш `encoder` і `decoder` можуть бути однаково неправильними.

Тому формат перевіряють не власним кодом, а еталонними даними.

Перед реалізацією рекомендується виписати собі коротку схему:

HEADER
↓
PAYLOAD
↓
DATA
↓
TAG

і для кожної секції:

поле
розмір
тип

порядок байтів
обмеження

Це допомагає уникнути великої кількості дрібних помилок.

5.3. Крок 3. Переглянути видані файли

Не потрібно одразу писати код.

Спочатку подивіться, що саме вам видано.

Наприклад:

```
team_config.json
train/
valid/
corpus/
reference/
expected_failures.json
```

Потрібно зрозуміти призначення кожного файла.

Особливо важливо не плутати:

```
train
valid
holdout
```

Holdout у нормальній роботі студент не бачить.

Якщо ж у вас є відкриті reference-файли, потрібно зрозуміти:

- які з них правильні;
- які навмисно пошкоджені;
- який результат очікується.

5.4. Крок 4. Побудувати карту системи

Перед кодом корисно намалювати просту схему.

Наприклад:

```
input
↓
parser
↓
algorithm
↓
encoder
↓
crypto
↓
output
```

Або:
events
↓
classifier
↓
priority
↓
DP
↓
replication
↓
encryption

Ця схема не повинна бути красивою.

Її задача — показати:

які блоки існують і як дані між ними проходять.

Для кожного блока потрібно вписати:

вхід
вихід
відповідальність

Наприклад:

Huffman encoder

input:
bytes

output:
compressed bytes + code lengths

відповідальність:
лише стиснення

Такий підхід дуже допомагає не змішувати все в одну велику функцію.

5.5. Крок 5. Узгодити інтерфейси між модулями

До того як різні студенти почнуть писати код, потрібно домовитися, як модулі будуть взаємодіяти.

Наприклад:

compress(data: bytes) -> bytes

або:

`classify(features) -> int`

або:

`select(items, budget) -> list[int]`

Якщо цього не зробити, один студент може повернути:

`dict`

інший очікувати:

`list`

а третій передавати:

`bytes`

Усі модулі окремо можуть працювати, але інтеграція провалиться.

Тому інтерфейси узгоджуються до реалізації.

5.6. Крок 6. Зробити найпростішу робочу версію

Це один із найважливіших етапів.

На цьому кроці складні алгоритми ще не потрібні.

Потрібно зробити найпростіший варіант системи, який уже проходить базовий шлях:

`input`

→ `process`

→ `output`

→ `restore`

Для архіватора це може бути:

файл

→ записати без стиснення

→ прочитати

→ отримати той самий файл

Для телеметрії:

пакет

→ записати

→ передати

→ прочитати

→ отримати ті самі дані

Для реплікації:

подія

→ створити одну репліку

- зберегти
- прочитати

На цьому етапі головний тест:
`original == restored`

Побайтово.

Не:

«Виглядає однаково».

Не:

«JSON схожий».

А саме:

`bytes_before == bytes_after`

5.7. Крок 7. Спочатку перевірити happy path

Happy path — це найпростіший правильний сценарій.

Наприклад:

один файл
коректні дані
достатньо ресурсу
правильний пароль
без пошкоджень

Поки happy path не працює, немає сенсу переходити до складних крайніх випадків.

Послідовність:

найпростіший правильний приклад

→ стабільна робота

→ кілька правильних прикладів

→ тільки потім помилки

5.8. Крок 8. Додати перший алгоритм

Після того як базовий тракт працює, можна додавати алгоритми.

Але по одному.

Наприклад:

спочатку Huffman

Після його інтеграції:

усі старі тести повинні проходити

Тільки після цього:

додаємо DP

Потім:
знову всі тести

Так само далі.
Це називається поступовою інтеграцією.

5.9. Крок 9. Кожен алгоритм спочатку тестується окремо

Перед тим як підключити алгоритм до системи, потрібно перевірити його ізольовано.

Наприклад, Huffman:
input bytes
→ encode
→ decode
→ ті самі bytes

DP:
малий набір
→ DP
→ порівняти з повним перебором

Greedy:
відомий приклад
→ перевірити порядок вибору

AES:
known test vector
→ результат повинен збігатися

MLP:
малий synthetic dataset
→ loss повинна зменшуватися

GA:
best fitness після кількох поколінь
не гірший за початковий

5.10. Крок 10. Використовувати маленькі приклади

Під час налагодження не потрібно одразу запускати систему на 100 МБ даних.

Малі приклади кращі.

Наприклад:
AAAAAA

або:

abcsabcsabc

або:

3 предмети

budget = 10

Малий приклад можна перевірити вручну.

Якщо алгоритм дав:

17

ви можете самі порахувати, чи це правильно.

На великому наборі це майже неможливо.

5.11. Крок 11. Порівнювати з простим еталоном

Для складного алгоритму дуже корисно мати простий, але повільний варіант.

Наприклад:

DP

vs

повний перебір на $n = 10$

Якщо результати збігаються на сотнях маленьких випадкових прикладів, це сильна перевірка.

Так само:

optimized search

vs

naive search

або:

fast parser

vs

simple reference parser

5.12. Крок 12. Додати negative tests

Після правильних сценаріїв потрібно перевірити неправильні.

Наприклад:

пошкодити один байт

Очікування:

архів відхилено

Інші приклади:

wrong password

bad magic

truncated file
invalid length
duplicate block
invalid Huffman table
unknown flag
missing volume
duplicate volume

Тут дуже важливо:

неправильні дані не повинні призводити до випадкового падіння програми.

Погано:

IndexError
struct.error
UnicodeDecodeError

Добре:

E_PAYLOAD
E_HMAC
E_CODE

або інший передбачений код.

5.13. Крок 13. Не ловити всі помилки через `except Exception`

Іноді студент бачить падіння і робить:

```
try:  
    ...  
except Exception:  
    return "error"
```

Це погана практика.

Так можна приховати реальну помилку в програмі.

Наприклад:

IndexError

може означати баг у вашому алгоритмі, а не погані вхідні дані.

Краще:

- явно перевіряти межі;
- явно перевіряти довжини;
- явно перевіряти типи;
- повертати конкретну помилку.

5.14. Крок 14. Додати логування

На ранніх етапах корисно мати простий лог.

Наприклад:

```
read header
verified HMAC
decoded 8 blocks
restored file a.txt
```

Але лог не повинен містити:

- пароль;
- секретний ключ;
- приватні дані;
- весь binary payload без потреби.

Лог потрібен для розуміння:

на якому кроці система перебуває.

5.15. Крок 15. Додати точні перевірки інваріантів

Інваріант — це правило, яке завжди повинно бути істинним.

Наприклад:

```
volume_count >= 1
```

або:

```
coded_len == raw_len
для NO_COMPRESSION
```

або:

```
sum(cost) <= budget
```

або:

```
HMAC перевірено до decrypt
```

або:

```
replica nodes distinct
```

Такі правила потрібно перевіряти прямо в коді або тестах.

Чим більше важливих інваріантів зафіксовано, тим легше знайти помилку.

5.16. Крок 16. Тільки після стабільної бази додавати ML

Перцептрон не повинен бути першим компонентом проєкту.

Спочатку система повинна працювати без нього.

Наприклад:

```
strategy = fixed
```

і тільки після цього:
`strategy = model.predict(features)`

Це дає можливість відповісти на важливе питання:

Що саме покращив ML?

Якщо система без ML не працює, це неможливо виміряти.

5.17. Крок 17. Спочатку зробити baseline для ML

Перед навчанням MLP потрібно мати простий baseline.

Наприклад:

`random classifier`

або:

`majority class`

Тоді можна оцінити:

чи модель взагалі навчилася чомусь корисному

Орієнтовні значення базових ліній залежать від кількості класів:

три класи → випадковий вибір дає macro-F1 близько 0,33

сім класів → випадковий вибір дає macro-F1 близько 0,16

Мажоритарний класифікатор зазвичай гірший за випадковий, бо повністю ігнорує малі класи.

Якщо для семи класів вийшло:

MLP = 0,18

випадковий = 0,16

це практично ненавчена модель, і додавати її до системи ще рано.
Конкретний поріг F1_min для вашого варіанта заданий у `team_config.json`.

5.18. Крок 18. Не торкатися holdout

У цих проєктах holdout командою навіть не отримується: прихована вибірка зберігається у викладача. Тому крок формулюється точніше — не намагатися відтворити або замінити holdout.

Процес виглядає так:

`train`

→ навчання

`valid`

→ вибір моделі та параметрів

holdout (у викладача)

→ фінальна оцінка, один раз

Типова помилка — використати `valid i` як набір для підбору параметрів, і як остаточну оцінку якості, а потім записати це число у звіт як підсумкове. Воно таким не є: модель обиралася саме за ним, тому оцінка завищена.

У звіті `valid-результат` наводиться як робочий орієнтир, і поруч зазначається, що підсумкове значення визначає викладач на прихованій вибірці. Розбіжність між цими числами є нормальною і сама по собі не є помилкою.

5.19. Крок 19. Генетичний алгоритм додавати тільки після готової `fitness`

GA не може працювати без надійної функції `fitness`.

Тому порядок:

спочатку система

↓

потім метрика

↓

потім перевірка метрики

↓

тільки після цього GA

Якщо `fitness` рахується неправильно, GA буде дуже добре оптимізувати неправильну функцію.

Це одна з найнеприємніших помилок.

5.20. Крок 20. Перед GA зробити `random search`

До запуску GA корисно реалізувати простий `random search`.

Наприклад:

100 випадкових конфігурацій

Це дає `baseline`.

Тоді GA і `random search` запускаються з однаковим бюджетом:

100 `fitness evaluations`

vs

100 `fitness evaluations`

І тільки після цього можна чесно порівнювати результат.

5.21. Крок 21. Шифрування додавати після правильного формату

Crypto не повинна приховувати помилки `parser`.

Якщо одразу все зашифрувати, кожна помилка виглядатиме однаково:

«Чомусь не відкривається».

Незрозуміло, чи справа в неправильному ключі, чи в неправильно записаній довжині поля, чи в помилці кодера.

Тому на ранньому етапі корисно мати внутрішній налагоджувальний режим, у якому навантаження серіалізується і читається без шифрування. Тоді помилки формату видно одразу.

Важливе застереження. Такий режим є тимчасовим інструментом розробки, а не варіантом формату. Специфікація не передбачає незашифрованих пакетів або томів, тому:

- файли, створені в цьому режимі, не є коректними за форматом і не здаються як результат;
- у фінальній версії режим або вилучається, або лишається прихованим прапорцем, недоступним у звичайному запуску;
- усі приймальні перевірки виконуються лише на повністю зашифрованих даних.

Порядок після налагодження формату звичайний:

payload

→ AES

→ HMAC

5.22. Крок 22. Криптографічні параметри не вигадуються

Якщо специфікація задає:

AES-256-CTR

PBKDF2-HMAC-SHA256

100000 iterations

nonce = 12 bytes

саме це і потрібно реалізувати.

Не треба вирішувати:

«Я зроблю AES-CBC, він теж нормальний».

У межах інтероперабельного формату правильний алгоритм — той, який визначений специфікацією.

5.23. Крок 23. Після кожного великого етапу запускати весь набір тестів

Типова ситуація:

додали новий алгоритм

новий тест проходить.

Але старі три зламалися.

Тому після кожної суттєвої зміни:

run all tests

Це називається regression testing.

Виправлення одного компонента не повинно ламати інші.

5.24. Крок 24. Додати автоматичний запуск тестів

Бажано, щоб тести запускалися однією командою.

Наприклад:

```
pytest
```

або:

```
python -m unittest
```

або інша команда, описана в README.

Ще краще — запускати тести автоматично після push або pull request.

Тоді команда одразу бачить:

```
green
```

або:

```
red
```

5.25. Крок 25. Зібрати наскрізний тракт системи

Коли основні частини готові, потрібно запустити всю систему від початку до кінця.

Наприклад:

```
input
```

```
→ classifier
```

```
→ DP
```

```
→ compression
```

```
→ encryption
```

```
→ storage
```

```
→ recovery
```

```
→ verification
```

Після цього перевіряється:

```
original == recovered
```

та всі додаткові інваріанти.

5.26. Крок 26. Виміряти базову метрику

Перед оптимізацією потрібно виміряти початковий результат.

Наприклад:

```
Q_baseline = ...
```

і зберегти його.

Потім можна послідовно вмикати компоненти:

baseline
+ Huffman
+ DP
+ MLP
+ GA

і дивитися:
як змінилася Q

Це називається ablation study.

5.27. Крок 27. Вимірювати один фактор за раз

Якщо одночасно змінити:
алгоритм
block size
MLP
GA parameters

а результат покращився, неможливо зрозуміти чому.
Тому хороший експеримент:
все однакове
змінюється тільки одна річ

Наприклад:
greedy
vs
DP

за тих самих даних і параметрів.

5.28. Крок 28. Зберігати результати експериментів

Не покладатися на пам'ять.
Для кожного важливого запуску потрібно зберігати:

date
commit
seed
parameters
dataset
result
time

Наприклад:
commit: a13fd2
seed: 2001

mode: DP
Q: 0.681
time: 3.21 s

Тоді через три тижні можна відтворити результат.

5.29. Крок 29. Не видаляти невдалі результати

Якщо експеримент не дав виграшу, це теж результат.

Наприклад:

```
greedy = 6233 bits  
DP      = 6233 bits
```

Це не означає:

«експеримент неправильний».

Потрібно пояснити, чому на цих даних жадібний вибір уже дає оптимальний розбір. Наприклад, для регулярних даних із довгими періодичними повторами найдовший збіг у кожній позиції одночасно є і глобально найкращим вибором, тому кращого розбору просто не існує.

Такий самий нульовий виграш очікуваний на даних, у яких збігів практично немає: там і жадібний алгоритм, і точний однаково записують майже все літералами.

Саме такі пояснення часто показують краще розуміння алгоритмів, ніж красивий графік.

5.30. Крок 30. Перед чекпоінтом зробити clean run

Перед задачею не потрібно запускати проєкт із середовища, де:

- вже є потрібні файли;
- вже створені каталоги;
- залишилися старі результати;
- вручну встановлені невідомі залежності.

Потрібно зробити чистий запуск.

Умовно:

```
новий каталог  
→ clone repository  
→ install  
→ run tests  
→ run project
```

Якщо це працює, проєкт відтворюваний.

5.31. Крок 31. Не виправляти вручну результати

Неприпустима ситуація:

програма створила output.json

після чого студент відкрив файл і вручну виправив:
2 → 3

Фінальний результат повинен бути отриманий програмою.

Якщо потрібна ручна зміна — це означає, що система ще не завершена.

5.32. Крок 32. Кожен етап повинен мати критерій готовності

Дуже корисне правило:

Крок завершений не тоді, коли «я багато над ним працював», а тоді, коли виконаний конкретний критерій.

Наприклад:

Huffman готовий:

encode(decode(x)) працює на всіх unit tests

DP готовий:

на малих задачах збігається з brute force

crypto готове:

known vector + tamper test проходять

integration готова:

повний round-trip працює

Це значно спрощує керування проектом.

5.33. Рекомендована загальна послідовність

У скороченому вигляді весь процес можна представити так:

1. Прочитати умову
2. Прочитати FORMAT_SPEC
3. Переглянути дані
4. Побудувати архітектурну схему
5. Узгодити інтерфейси
6. Реалізувати мінімальний round-trip
7. Додати unit tests
8. Додати базовий алгоритм
9. Додати negative tests
10. Додати DP / greedy
11. Провести перше порівняння
12. Додати ML
13. Додати crypto
14. Зібрати наскрізний тракт

15. Виміряти базову метрику
16. Додати GA
17. Провести експерименти і порівняння
18. Провести regression testing
19. Зробити clean run
20. Підготувати чекпоінт
21. Перейти до наступного етапу

5.34. Головне правило цього розділу

Під час роботи над проєктом постійно ставте собі питання:

Який найменший наступний крок я можу зробити так, щоб після нього система залишалася правильною і протестованою?

Не:

«Як мені одразу завершити весь проєкт?»

А:

«Який наступний компонент я можу додати без втрати контролю над системою?»

Саме такий підхід перетворює великий і складний семестровий проєкт на послідовність невеликих, зрозумілих задач.

6. Покрокове тестування проєкту

Тестування — це не фінальний етап після завершення програмування.

У хорошому проєкті тестування починається разом із першими рядками коду і супроводжує розробку до самого захисту.

Головна ідея проста:

Кожен новий компонент спочатку перевіряється окремо, потім у зв'язці з іншими, а після цього — у повній системі.

Не потрібно чекати, поки буде готовий увесь проєкт.

Якщо студент написав алгоритм Гаффмана, його вже можна тестувати.

Якщо реалізовано parser — його вже можна перевіряти на правильних і неправильних даних.

Якщо готовий DP — його вже можна порівнювати з повним перебором на малих прикладах.

Такий підхід значно спрощує пошук помилок.

6.1. Які види тестів потрібні

У межах проєкту рекомендується використовувати кілька рівнів тестування.

1. unit tests
2. round-trip tests
3. edge-case tests
4. negative tests
5. integration tests
6. regression tests
7. experimental tests
8. final acceptance tests

Кожен рівень відповідає на своє питання.

Unit tests

Чи правильно працює окремий алгоритм або функція?

Round-trip tests

Чи можна перетворити дані і потім відновити їх без втрат?

Edge-case tests

Що відбувається на крайніх значеннях?

Negative tests

Чи правильно програма відхиляє некоректні дані?

Integration tests

Чи працюють різні модулі разом?

Regression tests

Чи не зламала нова зміна те, що раніше працювало?

Experimental tests

Який алгоритм кращий і за яких умов?

Acceptance tests

Чи виконує готова система вимоги проєкту?

6.2. Починаємо з unit tests

Unit test перевіряє маленьку частину програми.

Наприклад:

`canonical_codes(lengths)`

або:

`select_dp(items, budget)`

або:
`derive_keys(password, salt)`

Unit test не повинен запускати всю систему.
Його задача — перевірити одну конкретну властивість.

6.3. Тестуйте не тільки великі приклади

Для алгоритмів дуже важливі маленькі приклади.
Наприклад, для Гаффмана:

A

AAAAAA

AB

ABVCCC

Для рюкзака:
 $W = 5$

A: $w=2$, $v=3$

B: $w=3$, $v=4$

C: $w=4$, $v=5$

На таких прикладах відповідь легко перевірити вручну.
Це дуже корисно.
Якщо тест використовує 10 000 елементів і дав неправильний результат, знайти причину важко.
Якщо тест використовує три елементи — помилка часто видно одразу.

6.4. Тестування структур даних

Якщо студент сам реалізує структуру даних, її потрібно перевірити окремо.
Наприклад, для купи:

1. вставити один елемент;
2. вставити кілька елементів;
3. вилучити мінімальний;
4. перевірити правильний порядок;
5. перевірити однакові пріоритети;
6. перевірити порожню купу.

Можна створити тест:

```
insert 5
insert 1
insert 3
```

очікування:

```
extract_min → 1
extract_min → 3
extract_min → 5
```

6.5. Тестування алгоритму Гаффмана

Для Гаффмана потрібно перевірити кілька різних властивостей.

Перша перевірка — round-trip

```
data
→ Huffman encode
→ Huffman decode
→ restored
```

```
i:
data == restored
```

Друга перевірка — префіксісність

Жоден код не повинен бути префіксом іншого.

Третя перевірка — канонічність

Якщо задані довжини кодів, відновлені коди повинні точно відповідати правилам FORMAT_SPEC.

Четверта перевірка — вироджений алфавіт

Наприклад:
AAAAAAAAAAAAAAAAAA

Є лише один символ.

У такому випадку не повинно бути:

```
code length = 0
```

Якщо формат задає:

```
length = 1
code = 0
```

саме це і потрібно отримати.

П'ята перевірка — некоректні таблиці

Наприклад:

- довжина 0;
- довжина більша за дозволену;
- дубльований символ;
- порушена рівність Крафта;
- символ поза дозволеним алфавітом.

Parser повинен їх відхиляти.

6.6. Тестування LZ77

Для LZ77 корисно починати з дуже простих рядків.

Наприклад:

abcsabcsabcsabc

або:

AAAAAAAAAAAAA

Потрібно перевірити:

- літерали;
- посилання;
- правильність `distance`;
- правильність `length`;
- максимальну довжину збігу;
- максимальну відстань;
- перекриття: випадок `distance < length`.

Останній пункт заслуговує окремого тесту. Рядок AAAAAAAAAAAAAA кодується як один літерал A і посилання `distance = 1, length = 11`, тобто посилання копіює дані, які саме зараз дописуються. Декодер зобов'язаний копіювати побайтово, дописуючи кожен байт у вихідний буфер одразу. Реалізація, яка копіює фрагмент одним блоком, на цьому тесті дасть неправильний результат, а на всіх інших тестах пройде.

Round-trip:

raw

→ LZ77 tokens

→ restore

→ raw

Окремий negative test

Посилання:

`distance > already_decoded_length`

повинно відхилятися.

6.7. Тестування greedy

Жадібний алгоритм потрібно тестувати не на тому, що він «дає хороший результат», а на тому, що він виконує саме своє правило.

Наприклад, якщо правило:

вибирати предмет із найбільшим `value / weight`

потрібно створити набір, де порядок легко передбачити.

Окремо корисно створити приклад, де greedy не оптимальний.

Це важливий тест для розуміння.

Якщо greedy завжди випадково тестується лише на даних, де він оптимальний, студент може зробити неправильний висновок, що greedy завжди знаходить оптимум.

6.8. Тестування динамічного програмування

Для DP є дуже сильний спосіб перевірки.

На малих наборах потрібно порівнювати результат із повним перебором.

Наприклад:

$n \leq 15$

Можна перебрати всі варіанти.

Тоді:

`DP_result == brute_force_result`

Це одна з найкращих перевірок правильності DP.

Рекомендовано:

1. створити 100 або більше малих випадкових наборів;
2. для кожного порахувати brute force;
3. порахувати DP;
4. порівняти результат.

Якщо всі збігаються, це значно надійніше за один великий тест.

6.9. Перевіряйте і значення, і відновлене рішення

Типова помилка:

DP правильно обчислив:

`max_value = 42`

але неправильно відновив, які елементи потрібно взяти.

Тому потрібно перевіряти обидва результати:

`value`

`i:`

selected_items

Потім:

sum(weights[selected]) <= W

i:

sum(values[selected]) == max_value

6.10. Тестування багатоваріантного рюкзака

У Проекті 3 потрібно перевіряти не тільки загальний бюджет.

Для кожної події має бути дозволений лише один варіант:

0

1

2

3 копії

Не можна одночасно вибрати:

1 копію

i

2 копії

для однієї події як два незалежні предмети.

Тест повинен перевіряти саме цю умову.

Також:

sum(cost) <= budget

6.11. Тестування перцептрона

Перцептрон потрібно тестувати поетапно.

Не починати одразу з повного датасету.

Тест forward pass

Для маленької мережі вручну задайте ваги:

x = [...]

W = [...]

b = [...]

i перевірте результат.

Тест loss

Для відомого прогнозу перевірте значення функції втрат.

Тест backpropagation

Корисний сильний тест — numerical gradient checking.

Ідея:

трохи збільшити вагу

→ подивитися, як змінився loss

і порівняти з gradient, який обчислив backpropagation.

Тест навчання

На простому synthetic dataset:

loss повинна зменшуватися

Наприклад:

epoch 1: 1.24

epoch 20: 0.61

epoch 100: 0.18

Не обов'язково монотонно на кожному кроці, але загальна тенденція повинна бути правильною.

6.12. Не тестуйте модель на train

Високий результат на train ще нічого не означає.

Потрібно окремо бачити:

train

valid

Якщо:

train F1 = 0.99

valid F1 = 0.45

це ознака перенавчання.

6.13. Тестування train/valid/holdout протоколу

Потрібно перевірити сам процес.

У цих проєктах holdout командою не отримується: прихована вибірка зберігається у викладача і використовується один раз.

Правильно:

train

→ навчання

valid

→ вибір моделі та параметрів

holdout (у викладача)
→ один фінальний запуск

Неправильно:
valid
→ підбір параметрів
→ той самий valid
→ записали результат у звіт як підсумковий

Так valid починає виконувати обидві ролі одночасно, і отримане число завищене: модель обиралася саме за ним.

Тому перевіряти треба сам процес: чи розділені набори, чи вибір моделі спирається лише на valid, і чи не потрапляють дані valid у навчання. Останнє легко перевірити автоматично: перетин множин файлів або записів має бути порожнім.

6.14. Тестування генетичного алгоритму

ГА потрібно тестувати по частинах.

Генерація популяції

Перевірити:

- правильну кількість особин;
- допустимі межі генів;
- відсутність некоректних хромосом.

Selection

На штучному наборі з відомими fitness перевірити, що кращі особини мають більший шанс бути вибраними.

Crossover

Перевірити, що діти складаються з генів батьків відповідно до правила.

Mutation

Перевірити:

- що мутація іноді відбувається;
- що значення не виходять за допустимі межі.

Elitism

Один із найважливіших тестів:
best_final >= best_initial

якщо fitness максимізується.

Якщо елітизм реалізований правильно, найкраще рішення не повинно зникнути.

6.15. GA потрібно порівнювати з random search

Для чесного тесту:

GA:

100 fitness evaluations

i:

Random Search:

100 fitness evaluations

Не можна:

GA 1000 запусків

vs

Random 20 запусків

і робити висновок, що GA кращий.

6.16. Тестування криптографії

Криптографічні алгоритми не варто перевіряти лише власним:

encrypt

→ decrypt

Чому?

Якщо encryption і decryption мають однакову помилку, round-trip може пройти.

Тому потрібні known test vectors.

Наприклад, для AES:

known key

known plaintext

known ciphertext

Результат повинен збігатися побайтово. Готові контрольні вектори для AES-256 наведені в стандарті FIPS-197 і в публікації NIST SP 800-38A; брати їх треба звідти, а не з власної реалізації.

6.17. Тестування HMAC

Мінімум чотири перевірки.

Правильні дані

verify → OK

Змінити один байт

verify → FAIL

Змінити tag

verify → FAIL

Невірний пароль

verify → FAIL

Важливо, що результатом є саме контрольована відмова з передбаченим кодом, а не спотворені дані на виході. Невірний пароль дає інший K_mac, тому перевірка HMAC не проходить, і до розшифрування справа взагалі не доходить.

6.18. Перевірка порядку HMAC і decrypt

Це окремий тест.

Потрібно переконатися, що при неправильному HMAC система не переходить до декодування payload.

Правильний порядок:

header

→ HMAC verify

→ decrypt

→ parse

Не:

decrypt

→ parse

→ HMAC

6.19. Тестування nonce

У CTR потрібно перевіряти унікальність nonce.

Для одного ключа:

nonce_i != nonce_j

для різних томів або повідомлень, відповідно до вимог формату: у Проєкті 1 nonce унікальний для кожного пакета, у Проєкті 2 — для кожного тому.

У багатотомному архіві також потрібно перевірити:

salt однакова в межах архіву

nonce різні

6.20. Round-trip як головний інтеграційний тест

Для всіх трьох проєктів round-trip є одним із центральних тестів.

Загальна форма:

```
original data
→ full system
→ stored/transmitted representation
→ recovery
→ restored data
```

Після цього:

```
original == restored
```

Цей тест потрібно запускати:

- на одному об'єкті;
- на кількох;
- на великому наборі;
- на крайніх випадках.

6.21. Edge cases

Крайні випадки — це дані на межі допустимого.

Типові приклади:

```
empty input
one byte
one symbol
maximum field value
minimum field value
zero budget
exact budget
very long name
one block
maximum block
```

Крайні випадки часто знаходять помилки, які не видно на звичайних даних.

6.22. Boundary tests

Особливо корисно тестувати значення:

```
limit - 1
limit
limit + 1
```

Наприклад, якщо максимальна відстань:
32768

потрібно перевірити:
32767
32768
32769

Очікування:
32767 → valid
32768 → valid
32769 → reject

6.23. Negative tests

Negative test спеціально подає неправильні дані.

Це не «поганий тест».

Навпаки, він перевіряє безпечну поведінку системи.

Типові мутації:

змінити magic
змінити version
обрізати файл
додати зайвий байт
змінити reserved
пошкодити HMAC
змінити CRC
пошкодити Huffman table
повторити block id
видалити volume
дублювати volume

6.24. Погані дані не повинні викликати необроблені винятки

Якщо parser отримав некоректний файл, допустимий результат:
E_PAYLOAD

або:
E_CODE

або інший передбачений код.

Небажаний результат:

IndexError
KeyError
struct.error
UnicodeDecodeError

Це означає, що перевірка меж або структури була недостатньою.

6.25. Fuzz testing

Додатково можна використовувати простий fuzz-підхід.

Не обов'язково застосовувати спеціальний framework.

Можна:

1. взяти правильний файл;
2. випадково змінювати байти;
3. обрізати;
4. вставляти зайві дані;
5. запускати parser багато разів.

Головна перевірка:

parser не зависає

не падає необробленим exception

не повертає неправильні дані

Це дуже корисно для parser і decoder.

6.26. Integration tests

Після unit tests потрібно поєднувати модулі.

Наприклад:

LZ77

→ Huffman

потім:

LZ77

→ Huffman

→ format writer

→ format parser

потім:

compression

→ encryption

→ HMAC

→ decryption

→ decompression

Краще інтегрувати поступово.
Не всі компоненти одразу.

6.27. Regression tests

Кожна знайдена помилка повинна перетворюватися на тест.
Наприклад, знайшли:

parser падає на порожній таблиці.

Порядок:

1. створити тест
2. переконатися, що він падає
3. виправити код
4. тест проходить
5. залишити тест назавжди

Це не дозволяє помилці повернутися пізніше.

6.28. Не видаляйте старі тести після виправлення

Іноді студент думає:

«Ця помилка вже виправлена, тест більше не потрібен».

Навпаки.

Саме після виправлення тест стає особливо цінним.

Він захищає систему від повторної появи помилки.

6.29. Experimental tests

У проєктах є перевірки, які не відповідають на питання:

«Правильно чи неправильно?»

Вони відповідають:

«Що краще?»

Наприклад:

greedy vs DP

compressed vs no compression

GA vs random search

MLP vs baseline

6.30. Правило чесного експерименту

Порівнювані алгоритми повинні отримувати:
однакові дані
однаковий seed
однаковий бюджет
однакове середовище

Змінюється тільки те, що досліджується.
Це називається контрольованим експериментом.

6.31. Вимірювання часу

Час потрібно вимірювати однаково.

Погано:

algorithm A на ноутбуці
algorithm B на сервері

або:

A один запуск
B середнє з 100 запусків

Краще:

не менше п'яти запусків
→ медіана або середнє
→ разом із розкидом

в одному середовищі і в одному сеансі.

Один замір нічого не означає: на нього впливають фонові процеси. Тому у звіті наводиться не одне число, а середнє або медіана з ознакою розкиду.

6.32. Warm-up

Для дуже коротких алгоритмів перший запуск може відрізнитися через:

- кеш;
- завантаження модулів;
- створення об'єктів;
- інші службові витрати.

Тому для маленьких benchmark можна зробити один або кілька warm-up запусків, а потім вимірювати.

6.33. Тестування використання ресурсів

Якщо у проєкті є бюджет, потрібно перевірити:
`used <= budget`

Для томів:
`volume_size <= max_volume_bytes`

Для реплікації:
`sum(replication_cost) <= budget`

Для каналу:
`transmitted_bits <= channel_budget`

6.34. Перевірка інваріантів після повного запуску

Після кожного full pipeline потрібно перевіряти не тільки фінальний результат.

Також:
усі IDs унікальні
усі блоки на місці
усі CRC правильні
усі HMAC правильні
усі budgets виконані
усі copies на різних вузлах

Залежно від конкретного проєкту.

6.35. Як організувати каталог тестів

Приблизно:
tests/
├─ test_huffman.py
├─ test_lz77.py
├─ test_dp.py
├─ test_greedy.py
├─ test_mlp.py
├─ test_ga.py
├─ test_crypto.py
├─ test_format.py
├─ test_negative.py
└─ test_integration.py

Не обов'язково саме так.

Але структура повинна дозволяти швидко знайти потрібний тип тестів.

6.36. Назва тесту повинна пояснювати, що перевіряється

Хороша назва:

```
test_single_symbol_huffman_roundtrip()
```

```
test_wrong_password_returns_hmac_error()
```

```
test_dp_matches_bruteforce()
```

Погана:

```
test1()
```

```
test2()
```

```
test_new()
```

Через кілька тижнів такі назви нічого не пояснюють.

6.37. Один тест — одна зрозуміла причина падіння

Якщо один тест перевіряє двадцять незалежних речей, при падінні незрозуміло, що сталося.

Краще:

```
test_bad_magic
```

```
test_bad_hmac
```

```
test_bad_version
```

```
test_truncated_payload
```

ніж:

```
test_all_bad_files
```

Хоча окремо може бути один загальний integration test для всього набору.

6.38. Автоматичні тести повинні запускатися однією командою

Наприклад:

```
pytest
```

Після запуску викладач не повинен вручну відкривати десять файлів і запускати їх по одному.

README повинен містити одну чітку команду.

6.39. Покриття тестами

Coverage не означає автоматично, що тести хороші.

Можна мати:

100% coverage

і не перевіряти жодного правильного результату.

Але coverage корисний, щоб побачити:

які частини коду взагалі ніколи не виконуються тестами.

Особливу увагу потрібно звернути на:

- error branches;
- крайні випадки;
- parser validation;
- recovery logic.

6.40. Тести перед pull request

Перед створенням pull request автор повинен:

run all tests

Не варто передавати review-коду, який уже відомо не проходить базові перевірки.

6.41. Тести після merge

Після merge у main потрібно знову запускати весь набір.

Тому що:

feature A

може працювати окремо,

і:

feature B

теж,

але після merge:

A + B

можуть конфліктувати.

6.42. Що потрібно показувати викладачу під час проміжної перевірки

Не потрібно відкривати весь код і прокручувати його.

Краще показати послідовно:

1. Команду запуску тестів

`pytest`

2. Результат

Наприклад:

`58 passed`

3. Один простий unit test

Пояснити, що він перевіряє.

4. Один edge case

Наприклад:

`empty input`

5. Один negative case

Наприклад:

`tampered HMAC`

6. Один integration test

`full round-trip`

7. Порівняльний experiment

Якщо він уже передбачений поточним чекпоінтом.

6.43. Журнал тестування

Корисно вести короткий журнал.

Наприклад:

`2026-09-10`

`commit: 8fa19c`

`tests: 42 passed`

`fixed: empty Huffman alphabet`

`2026-09-14`

`commit: 91c88a`

`tests: 48 passed`

`added: bad HMAC, wrong password`

Не потрібно перетворювати це на великий документ.

Головне — бачити історію якості системи.

6.44. Коли компонент можна вважати протестованим

Не тоді, коли:

«Я один раз запустив і воно спрацювало».

А тоді, коли перевірені:

нормальний випадок
крайній випадок
помилковий випадок
інтеграція

Для алгоритму також бажано:
порівняння з простим еталоном

6.45. Коли весь проєкт можна вважати протестованим

До фінальної задачі повинно виконуватися:

all unit tests pass
all negative tests pass
all integration tests pass
all C1 cases pass
clean run pass

і окремо:

original == restored

для всіх даних, де потрібне точне відновлення.

6.46. Фінальний clean test

Перед фінальною задачею зробіть окремий каталог.

Потім:

clone repository

далі:

install dependencies

потім:

run tests

і:

run full project

Не використовуйте старі кеші, вручну створені файли або дані, яких немає в репозиторії.

Це імітує ситуацію:

викладач отримав ваш проєкт уперше.

6.47. Тестовий чек-лист перед задачею

Перед кожним важливим чекпоінтом команда повинна відповісти «так» на такі питання:

- ☐ Усі старі тести проходять?
- ☐ Нові функції мають тести?
- ☐ Є крайні випадки?
- ☐ Є negative tests?
- ☐ Немає необроблених exceptions?
- ☐ Full round-trip проходить?
- ☐ Бюджетні обмеження виконуються?
- ☐ Формат відповідає FORMAT_SPEC?
- ☐ Результат можна відтворити?
- ☐ README містить актуальну команду запуску?

Якщо на кілька питань відповідь «ні», проєкт ще не готовий до здачі цього етапу.

6.48. Головне правило тестування

Найважливіше правило можна сформулювати так:

Не намагайтеся довести, що ваша програма працює. Намагайтеся знайти дані, на яких вона зламається.

Якщо після багатьох таких спроб програма продовжує працювати правильно, довіра до неї стає значно вищою.

Саме тому хороший тест — це не той, який підтвердив очікуваний результат.

Хороший тест — це той, який міг знайти реальну помилку.

7. Покрокова задача проєкту

Семестровий проєкт не здається один раз наприкінці семестру.

Він виконується і перевіряється поетапно.

Це зроблено не для того, щоб збільшити кількість контрольних робіт. Навпаки, проміжні перевірки потрібні для того, щоб команда не витратила кілька місяців на неправильне рішення.

Якщо помилка у форматі даних знайдена на четвертому тижні, її ще легко виправити.

Якщо та сама помилка виявиться на фінальному захисті, вона може впливати вже на:

- стиснення;
- динамічне програмування;
- шифрування;
- реплікацію;
- генетичний алгоритм;
- результати експериментів.

Тому проєкт проходить кілька контрольних етапів:

Етап	Тиждень	Головний зміст
Зріз 0	1	зони відповідальності, рівні складності, план, репозиторій
Чекпоінт 1	4	наскрізний тракт без стиснення і кодування Гаффмана
Чекпоінт 2	8	динамічне програмування проти жадібного, класифікатор
Чекпоінт 3	11	шифрування, генетичний алгоритм, повна система
Взаємні перевірки	12	крос-тести і атаки між командами
Захист	13	фінальна демонстрація

Кожен наступний етап будується на попередньому.

Головне правило:

На чекпоінт здається не обіцянка, що певна частина буде готова, а вже працюючий і перевірений результат.

7.1. Що означає «здати етап»

Для кожного етапу є три різні речі:

1. що повинно бути реалізовано;
2. що повинно бути в репозиторії;
3. що потрібно показати викладачу.

Ці речі не потрібно плутати.

Наприклад, недостатньо сказати:

«У нас уже написаний алгоритм Гаффмана».

Потрібно мати:

код
+
тести
+
результат запуску
+
можливість пояснити алгоритм

Так само недостатньо мати хороший графік у звіті, якщо команда не може показати код, який цей результат отримав.

7.2. Зріз 0 — старт проєкту

Зріз 0 проводиться на початку роботи.

На цьому етапі ще не потрібно мати готову програму.

Головна задача — показати, що команда:

- зрозуміла постановку;
- розібралася з матеріалами;
- розділила систему на частини;
- розподілила роботу;
- має реалістичний план.

До Зрізу 0 команда повинна повністю прочитати постановку свого проєкту.

Окремо потрібно прочитати видану специфікацію формату.

Не потрібно розуміти кожен байт напам'ять, але команда повинна вже знати:

що є входом;
що є виходом;
які основні модулі;
які алгоритми обов'язкові;
як буде перевірятися результат.

7.3. Що повинно бути готово до Зрізу 0

До цього моменту команда повинна створити репозиторій.

У репозиторії вже мають бути:

README.md
META-USAGE.md
початкова структура каталогів

Наприклад:

src/
tests/

experiments/
docs/

Також потрібно створити початковий список задач.

Це можуть бути issue:

реалізувати parser;
реалізувати writer;
реалізувати Huffman;
реалізувати greedy;
реалізувати DP;
реалізувати classifier;
реалізувати crypto;
реалізувати integration tests.

Не потрібно одразу створювати сотню issue.

Достатньо мати зрозумілу карту найближчої роботи.

7.4. Розподіл зон на Зрізі 0

Кожен студент повинен мати власну зону відповідальності.

Для кожної зони фіксується:

назва;
відповідальний студент;
базовий / просунутий / експертний рівень;
очікуваний результат.

Наприклад:

Зона: Динамічне програмування

Результат:

реалізовано точний алгоритм;
додані unit tests;
проведено порівняння з greedy;
визначена складність.

Такий опис значно кращий за:

«Іван робить оптимізацію».

7.5. Що показувати викладачу на Зрізі 0

Демонстрація повинна бути короткою.

Команда показує:

1. репозиторій;

2. структуру каталогів;
3. схему системи;
4. зони відповідальності;
5. заявлені рівні;
6. список основних issue;
7. план роботи;
8. основні ризики.

На цьому етапі викладач може побачити, що певний план нереалістичний.
Наприклад:

три великі модулі залежать від одного студента.

або:

інтеграція запланована лише на останній тиждень.

або:

команда ще не визначила, як модулі будуть обмінюватися даними.

Такі проблеми краще виправити саме зараз.

7.6. Критерій готовності Зрізу 0

Зріз 0 можна вважати завершеним, якщо кожен член команди може відповісти:

Що робить наша система?

Яка моя зона?

Що саме я повинен отримати на виході своєї зони?

Від кого моя частина отримує дані?

Кому вона передає результат?

Який перший тест я напишу?

Якщо на ці питання немає зрозумілої відповіді, програмувати ще зарано.

7.7. Чекпоінт 1 — перша робоча система

Перший чекпоінт є дуже важливим.

До нього команда повинна отримати мінімальну наскрізну версію системи.

Вона ще не повинна бути оптимальною.

Не повинна використовувати всі просунуті алгоритми.

Але основний шлях даних уже повинен працювати.

Загальна ідея:

input
→ basic processing
→ output
→ recovery
→ verification

7.8. Що означає мінімальна робоча версія

Для різних проєктів вона буде різною.

Але принцип один.

Для системи передавання даних:

створити пакет

→ передати
→ прочитати
→ відновити дані

Для архіватора:

файл

→ записати в архів
→ прочитати архів
→ відновити файл

Для системи реплікації:

подія

→ сформувати репліку
→ записати
→ знайти
→ відновити

Головний критерій:

`original == restored`

якщо проєкт передбачає точне відновлення даних.

7.9. Що повинно бути готово до Чекпоінту 1

До цього моменту рекомендується мати:

- базову реалізацію формату;
- writer;
- parser;
- мінімальний алгоритмічний тракт;
- перший алгоритм стиснення або кодування, якщо він є базовою частиною проєкту;

- round-trip tests;
- unit tests уже реалізованих компонентів;
- обробку кількох простих крайніх випадків.

Важливо:

На цьому етапі правильність важливіша за швидкість.

Якщо є вибір між:
повільно, але правильно

і:
дуже швидко, але іноді неправильно

для Чекпоінту 1 правильний перший варіант.

7.10. Що повинно бути в репозиторії на Чекпоінті 1

До цього моменту репозиторій уже повинен виглядати як справжній програмний проєкт.

У ньому мають бути:

src/
tests/
README.md
META-USAGE.md

а також необхідні конфігураційні файли.

README повинен уже містити:

як встановити залежності;
як запустити програму;
як запустити тести.

Команда запуску тестів повинна бути одна і зрозуміла.

Наприклад:

pytest

7.11. Що демонструвати на Чекпоінті 1

Рекомендований порядок демонстрації:

1. показати commit або tag перевірюваної версії;
2. запустити всі тести;
3. показати простий unit test;
4. показати один round-trip;
5. показати один edge case;
6. запустити мінімальну систему від початку до кінця.

Наприклад:
input.txt
→ archive
→ extract
→ restored.txt

і потім автоматично перевірити:
input.txt == restored.txt

7.12. Що не потрібно робити до Чекпоінту 1

Не потрібно намагатися вже завершити:

- складний MLP;
- повний генетичний алгоритм;
- усі експертні алгоритми;
- всі графіки;
- фінальний звіт.

Якщо базовий формат ще нестабільний, додавання цих частин тільки ускладнить налагодження.

7.13. Критерій готовності Чекпоінту 1

Перший чекпоінт завершений, якщо:
мінімальний pipeline працює;
дані відновлюються правильно;
основні unit tests проходять;
є round-trip tests;
запуск описаний у README;
результат можна відтворити з репозиторію.

7.14. Чекпоінт 2 — алгоритмічна частина

На другому чекпоінті проєкт уже повинен бути не просто робочим.

Тепер потрібно показати алгоритмічну частину дисципліни.

До цього моменту команда переходить від:

«система працює»

до:

«ми можемо пояснити, які алгоритми використовуємо і що вони дають».

7.15. Що повинно бути готово до Чекпоінту 2

На цьому етапі мають бути реалізовані основні алгоритми базового рівня. Залежно від конкретного проєкту це можуть бути:

- жадібний алгоритм;
- динамічне програмування;
- рюкзак;
- Huffman;
- LZ77;
- алгоритми вибору реплік;
- базовий перцептрон;
- інші алгоритми, передбачені умовою.

Важливо не просто написати їх.

До кожного алгоритму повинні бути тести.

7.16. Порівняння greedy і точного алгоритму

До другого чекпоінту команда зобов'язана мати перший повноцінний експеримент: таблиця порівняння жадібного і точного алгоритмів є обов'язковим артефактом цього чекпоінту, а не побажанням.

Наприклад:

greedy

vs

DP

Обидва алгоритми повинні отримувати однакові дані.

Порівнюються:

результат;

час;

за потреби пам'ять.

Студент повинен бути готовий відповісти:

Де DP дав кращий результат?

Де результат однаковий?

Наскільки DP повільніший?

Чому?

7.17. Перевірка DP на Чекпоінті 2

Дуже бажано показати не лише великий приклад.

Потрібно мати автоматичну перевірку:

DP == brute force

на малих задачах.

Це сильніше доводить правильність реалізації.

Наприклад:

100 випадкових малих задач

→ brute force

→ DP

→ усі результати збігаються

7.18. Перцептрон на Чекпоінті 2

Якщо базовий перцептрон входить до вашого варіанта, до цього етапу вже повинні бути:

обчислення ознак;

forward;

навчання;

validation;

метрика якості.

Не потрібно мати найкращу можливу модель.

Потрібно показати, що навчання працює.

Наприклад:

baseline macro-F1 = ...

MLP macro-F1 = ...

І пояснити різницю.

7.19. Що демонструвати на Чекпоінті 2

Рекомендований порядок:

1. запустити весь набір тестів;
2. показати тест DP проти brute force;
3. показати greedy;
4. показати DP;
5. запустити обидва на однакових даних;
6. показати результати порівняння;
7. показати MLP або інший алгоритмічний компонент, якщо він уже входить у цей етап;
8. показати, що full pipeline після додавання алгоритмів не зламався.

7.20. Що повинно бути в репозиторії на Чекпоінті 2

Крім коду й тестів, уже повинен з'явитися каталог експериментів.

Наприклад:

experiments/

У ньому можуть бути:

greedy_vs_dp.json

classifier_results.json

timing.csv

або інший машинно створений формат.

Не потрібно вручну переписувати результати в красиві файли.

Результати повинні бути отримані програмою.

7.21. Критерій готовності Чекпоінту 2

Другий чекпоінт можна вважати завершеним, якщо:

основні алгоритми реалізовані;

для них є unit tests;

DP перевірено;

greedy працює;

проведено хоча б одне коректне порівняння;

full pipeline проходить;

результати експерименту відтворюються.

7.22. Чекпоінт 3 — повна інтегрована система

Третій чекпоінт — це вже майже готовий проєкт.

До цього моменту не повинно залишатися великих частин типу:

«Шифрування ми ще не починали».

або:

«GA додамо перед захистом».

Після третього чекпоінту повинні залишитися переважно:

- виправлення;
- додаткові тести;
- повторні експерименти;
- оформлення;
- підготовка до захисту.

7.23. Що повинно бути готово до Чекпоінту 3

До цього моменту базова система повинна включати всі обов'язкові компоненти конкретного проекту.

Зокрема, за наявності у варіанті:

- повний алгоритмічний pipeline;
- стиснення;
- DP;
- greedy;
- MLP;
- криптографічний захист;
- HMAC;
- контроль цілісності;
- генетичний алгоритм;
- автоматичні тести;
- вимірювання основної метрики.

7.24. Перевірка криптографії на Чекпоінті 3

Не потрібно демонструвати криптографію словами.

Потрібно показати тести.

Наприклад:

correct password → success

wrong password → reject

original archive → success

tampered archive → reject

correct HMAC → decrypt

incorrect HMAC → decrypt не запускається

Також потрібно бути готовим пояснити:

- навіщо KDF;
- навіщо salt;
- чому ключі шифрування та HMAC розділені;
- чому nonce не можна повторювати;
- чому HMAC перевіряється до розшифрування.

7.25. Генетичний алгоритм на Чекпоінті 3

До цього моменту GA повинен уже працювати на реальній функції fitness.

Потрібно показати:

початкова популяція;
кілька поколінь;
best fitness;
фінальне рішення.

Окремо перевіряється елітизм.

Якщо fitness максимізується:
`best_final >= best_initial`

7.26. GA проти random search

Обов'язкове порівняння повинно бути чесним.

Наприклад:

GA:
`200 fitness evaluations`

Random Search:
`200 fitness evaluations`

Потім порівнюється найкращий результат.

Не потрібно доводити, що GA обов'язково переміг.

Якщо random search виявився таким самим або кращим, це теж результат.

Потрібно пояснити можливу причину.

7.27. Основна метрика проєкту

До третього чекпоінту основна метрика вже повинна обчислюватися автоматично.

Не вручну в калькуляторі.

Потрібна команда або сценарій:

`run evaluation`

→ $Q = \dots$

або відповідна метрика вашого проєкту.

Рекомендується вже мати:

`baseline`

`vs`

`current`

щоб бачити, що саме змінилося за час роботи.

При цьому важливо розрізняти два числа. Значення, яке команда отримує в себе, є робочим орієнтиром. Офіційне значення метрики обчислює викладач — в

однаковому для всіх середовищ, а там, де це передбачено умовою варіанта, ще й на прихованих значеннях `seed` з усередненням за кількома прогонами.

Тому підбирати рішення під конкретний відкритий `seed` не має сенсу, а розбіжність між власним і офіційним числом є нормальною.

7.28. Що демонструвати на Чекпоінті 3

Рекомендований порядок:

1. `clean test run`;
2. `full pipeline`;
3. `negative test`;
4. алгоритмічне порівняння;
5. `MLP result`;
6. `crypto test`;
7. `GA result`;
8. основна метрика;
9. список залишкових проблем.

Останній пункт дуже важливий.

Не потрібно робити вигляд, що проблем немає.

Якщо відомо:

певний алгоритм повільний на великих даних,
це потрібно записати.

7.29. Після Чекпоінту 3 не починайте переписувати систему з нуля

Це типова небезпечна помилка.

Команда бачить, що архітектура неідеальна, і вирішує:

«Давайте все перепишемо за два тижні».

У більшості випадків результат стає гіршим.

Після Чекпоінту 3 пріоритет:

стабільність

- тести
- виправлення
- вимірювання
- документація

а не масштабний `rewrite`.

7.30. Тиждень інтеграційних та атакуювальних тестів

Перед фінальною задачею система повинна пройти перевірки не лише власними авторами.

Це особливо важливо для форматів даних.

Ці перевірки проводяться на дванадцятому тижні, після третього чекпоінту.

Інша команда може спробувати:

- прочитати ваш файл;
- передати вам свій файл;
- пошкодити ваш файл;
- змінити поля;
- подати неправильні дані.

Так знаходяться помилки, які автори не помічали.

Окремо проводиться криптоаналітична вправа. AES не атакується і атакувати його не потрібно. Для цієї частини викладач видає окремий набір даних, зашифрованих навмисно слабо, і мета вправи — показати, що генетичний алгоритм є практичним інструментом пошуку там, де простір ключів досить малий.

Крос-сумісність перевіряється на базовому форматі. Командні розширення, які специфікація не описує, у взаємних перевірках не беруть участі: чужа система не зобов'язана їх розуміти і повинна відхилити такий файл із передбаченим кодом помилки.

7.31. Що робити після знайденої помилки

Якщо інша команда знайшла баг:
не виправляйте його тільки вручну.

Правильний процес:

bug
→ issue
→ test reproducing bug
→ fix
→ test passes
→ regression suite

Тест повинен залишитися в проєкті.

7.32. Підготовка фінальної версії

Після завершення основної розробки потрібно зафіксувати версію, яка піде на захист.

Наприклад:
`final-v1.0`

або інший тег, визначений правилами курсу.

Після цього не рекомендується безконтрольно змінювати код.

Якщо знаходиться критична помилка, створюється нова версія:

`final-v1.1`

7.33. Що повинно бути у фінальному репозиторії

Фінальний репозиторій повинен містити все необхідне для запуску.

Наприклад:

```
src/  
tests/  
experiments/  
docs/  
README.md  
META-USAGE.md  
requirements.txt
```

або:

`pyproject.toml`

залежно від структури проєкту.

Не повинно бути необхідності шукати:

«ще один файл у чаті».

або:

«датасет у Google Drive».

Якщо він потрібний для запуску, README повинен точно пояснювати, звідки його взяти.

7.34. README фінальної версії

README — це інструкція для людини, яка бачить проєкт уперше.

Він повинен відповідати на питання:

Що це?

Як встановити?

Як запустити?

Як запустити тести?

Як запустити експерименти?

Які дані потрібні?

Де основні результати?

README не повинен містити лише:

«Проект з алгоритмів».

7.35. Фінальний clean run

Перед здачею обов'язково перевірити проєкт у чистому середовищі.

Послідовність:

новий каталог

→ clone

→ checkout final tag

→ install

→ tests

→ full run

Команда повинна переконатися, що не використовуються випадково:

- локальні файли;
- кеші;
- старі результати;
- секретні налаштування;
- бібліотеки, яких немає в залежностях.

7.36. Фінальний тестовий набір

Перед захистом запускаються:

unit tests

edge cases

negative tests

integration tests

acceptance tests

В ідеальному випадку все запускається однією командою або невеликою кількістю чітко описаних команд.

7.37. Фінальні експерименти

Фінальні числові результати потрібно отримувати з фінального commit/tag.

Не можна використовувати в звіті:

результат, отриманий три тижні тому на іншій версії коду,

якщо його неможливо повторити зараз.

Для кожного важливого результату потрібно знати:

```
commit/tag;  
seed;  
dataset;  
parameters;  
result.
```

7.38. Що здається викладачу фінально

Фінально команда надає:

- посилання на репозиторій;
- точний тег або commit;
- README;
- вихідний код;
- автоматичні тести;
- результати експериментів;
- звіт;
- META-USAGE;
- інші матеріали, передбачені конкретним варіантом.

Якщо потрібен окремий архів із проєктом, він повинен відповідати тому самому commit, який вказаний у репозиторії.

7.39. Що команда повинна вміти показати без підготовки

На фінальному запуску викладач може попросити:
запустити тести;
запустити один конкретний тест;
створити новий приклад;
зіпсувати один байт;
змінити budget;
змінити seed;
показати greedy;
показати DP;
показати результат classifier;
перезапустити GA.

Тому не потрібно будувати демонстрацію як театральний сценарій, який працює лише при натисканні кнопок у певному порядку.

Проєкт повинен бути справді працездатним.

7.40. Індивідуальна готовність

Командна задача не означає, що всі отримують однаковий індивідуальний результат автоматично.

Кожен студент повинен знати свою зону.

Він повинен уміти:

- знайти свій код;
- пояснити алгоритм;
- запустити свої тести;
- пояснити складність;
- пояснити один експеримент;
- пояснити один крайній випадок.

7.41. Не вчіть код напам'ять

На захисті не потрібно пам'ятати кожен рядок.

Потрібно розуміти:

- яка задача;
- який алгоритм;
- який стан;
- який інваріант;
- яка складність;
- як перевіряється результат.

Якщо студент це розуміє, конкретний рядок коду завжди можна знайти.

7.42. Що вважати готовим фінальним проєктом

Проєкт можна вважати готовим, якщо:

- основна задача виконується;
- дані відновлюються правильно;
- усі базові acceptance tests проходять;
- немає необроблених помилок на відомих negative cases;
- результати відтворюються;
- README працює;
- експерименти проведені;
- кожен учасник знає свою частину.

Проєкт не обов'язково повинен бути ідеальним.

Семестровий проєкт — це навчальна робота.

Але він повинен бути закінченим.

Краще мати:

простішу
але стабільну
протестовану
зрозумілу
систему

ніж:
дуже складну
але напівготову
і нестабільну

7.43. Короткий маршрут задачі

Усю систему контрольних етапів можна запам'ятати так:
ЗРІЗ 0

Що ми будуємо і хто що робить?

↓

ЧЕКПОІНТ 1

Чи працює найпростіша система?

↓

ЧЕКПОІНТ 2

Чи правильно працюють основні алгоритми?

↓

ЧЕКПОІНТ 3

Чи працює повна інтегрована система?

↓

ФІНАЛЬНА ПЕРЕВІРКА

Чи витримує система тести та помилкові дані?

↓

ЗАХИСТ

Чи можемо ми пояснити, що зробили і чому?

7.44. Головне правило покрокової здачі

Не потрібно на кожному чекпоінті намагатися показати якомога більше. Потрібно показати те, що вже завершено і перевірено.

Тому краще сказати:

«Ці чотири компоненти готові, для них є тести, ось результат».

ніж:

«Ми почали дванадцять компонентів, але поки жоден повністю не працює».

Саме послідовне накопичення невеликих завершених результатів і приводить команду до готового фінального проєкту.

8. Оформлення результатів і підготовка звіту

Звіт про проєкт — це не переказ програмного коду.

Його задача інша.

Звіт повинен дати можливість людині, яка не брала участі в розробці, зрозуміти:

- яку задачу розв'язувала команда;
- як побудована система;
- які алгоритми використано;
- чому було обрано саме їх;
- як перевірялася правильність;
- які експерименти проведено;
- який результат отримано;
- які рішення виявилися вдалим, а які — ні.

Тому хороший звіт відповідає не стільки на питання:

«Що написано в коді?»

скільки на питання:

«Що ми зробили, чому зробили саме так і що показали результати?»

8.1. Звіт пишеться протягом семестру

Не потрібно починати писати звіт за два дні до захисту.

До цього моменту вже важко згадати:

- чому було прийнято певне рішення;
- які параметри використовувалися в першому експерименті;
- який результат був до оптимізації;

- який баг було знайдено;
 - чому від певної ідеї відмовилися.
- Тому звіт краще створювати поступово.
Приблизно так:

Зріз 0

→ постановка задачі та архітектура

Чекпоінт 1

→ базова реалізація і перші тести

Чекпоінт 2

→ алгоритми та їх порівняння

Чекпоінт 3

→ повна система, GA, ML, захист

Фінал

→ узагальнення результатів і висновки

Тоді наприкінці семестру звіт не пишеться з нуля, а лише приводиться до завершеного вигляду.

8.2. Рекомендована структура звіту

Звіт може мати таку структуру:

1. Постановка задачі
2. Архітектура системи
3. Використані алгоритми і структури даних
4. Реалізація основних компонентів
5. Методика тестування
6. Експериментальне дослідження
7. Результати
8. Аналіз результатів
9. Відомі обмеження
10. Висновки
11. Внесок учасників команди
12. Використані джерела

Конкретний проєкт може потребувати додаткових підрозділів.

Але загальна логіка повинна залишатися простою:

задача

→ рішення

- перевірка
- результат
- висновки

8.3. Постановка задачі

Перший розділ звіту повинен коротко пояснити, що саме створювала команда.

Не потрібно копіювати всю умову проєкту.

Достатньо своїми словами описати:

- вхідні дані;
- результат;
- основні обмеження;
- критерій якості.

Наприклад:

Система отримує набір файлів і повинна сформувати захищений багатотомний архів. Після розпакування всі файли мають бути відновлені побайтово. Додатково система повинна мінімізувати розмір архіву та час обробки.

Цього достатньо, щоб читач зрозумів задачу.

8.4. Архітектура системи

Після постановки задачі потрібно показати загальну схему системи.

Наприклад:

```
input
↓
classification
↓
compression
↓
optimization
↓
encryption
↓
output
```

або більш детально:

```
файли
→ LZ77
→ Huffman
```

- пакування
- AES
- HMAC
- архів

Схема повинна показувати рух даних.

Не потрібно малювати десятки класів і функцій.

На цьому рівні важливо побачити основні компоненти і зв'язки між ними.

8.5. Для кожного компонента пояснюйте його роль

Недостатньо написати:

«У системі використовується динамічне програмування».

Потрібно пояснити:

Для чого саме воно використовується.

Наприклад:

Динамічне програмування використовується для вибору набору блоків, який максимізує сумарну цінність при обмеженій місткості тому.

Після такого речення вже зрозуміло, навіщо цей алгоритм існує в системі.

8.6. Опис алгоритму не повинен бути копією підручника

Не потрібно переписувати кілька сторінок теорії про алгоритм Гаффмана.

Краще пояснити його в контексті своєї роботи.

Наприклад:

Після LZ77 отримується послідовність символів і посилань. Для зменшення розміру їх представлення використовується канонічне кодування Гаффмана. Часті символи отримують коротші коди, а в архіві зберігаються лише довжини кодів.

Далі можна навести основні кроки алгоритму.

8.7. Псевдокод краще за великі фрагменти коду

У звіті не потрібно вставляти функцію на 150 рядків.

Код уже є в репозиторії.

Якщо потрібно пояснити алгоритм, краще використати псевдокод.

Наприклад:

```
for each item:  
    if weight[item] <= capacity:
```

```
choose better of:
    skip item
    take item
```

Псевдокод показує ідею без службових деталей Python.

8.8. Формули повинні бути пояснені словами

Якщо у звіті є:

$DP[i][w]$

потрібно одразу пояснити:

$DP[i][w]$ — максимальна сумарна цінність, яку можна отримати, використовуючи перші i елементів при доступній місткості w .

Не залишайте формулу без пояснення.

Особливо важливо пояснити:

- що означає стан;
- що означає кожен параметр;
- що максимізується або мінімізується.

8.9. Обчислювальна складність

Для основних алгоритмів потрібно навести оцінку складності.

Наприклад:

Huffman: $O(k \log k)$

але після цього пояснити:

k — кількість різних символів.

Або:

Knapsack DP: $O(n \cdot W)$

де:

n — кількість блоків-кандидатів,
 W — місткість тому.

Без пояснення змінних запис $O(n \cdot W)$ мало що говорить про конкретну систему.

8.10. Не потрібно приховувати спрощення

Навчальний проєкт не зобов'язаний використовувати найскладніший відомий алгоритм.

Якщо використано простішу реалізацію, це нормально.

Наприклад:

Для пошуку збігів LZ77 використано обмежений список кандидатів замість суфіксного дерева.

Після цього потрібно пояснити:

Це спрощує реалізацію, але збільшує час пошуку для великих даних.

Це хороший інженерний аналіз.

8.11. Розділ тестування

У звіті не потрібно перелічувати всі сотні тестів.

Краще описати стратегію тестування.

Наприклад:

`unit tests`

- `edge cases`
- `negative tests`
- `integration tests`
- `acceptance tests`

Потім навести кілька характерних прикладів.

Наприклад:

- порожній файл;
- один символ;
- неправильний HMAC;
- відсутній том;
- DP проти brute force;
- повний round-trip.

8.12. Покажіть, як перевірялася правильність

Для різних алгоритмів можуть бути різні еталони.

Наприклад:

`DP`

- порівнювався з brute force

`AES`

- `known test vector`

`encoder/decoder`

- `round-trip`

classifier
→ valid macro-F1

Це значно сильніше, ніж просто написати:

«Алгоритм протестовано».

8.13. Експериментальна частина — одна з головних

Проект повинен показувати не тільки те, що алгоритми працюють.

Потрібно дослідити:

Що вони дають?

Наприклад:

greedy
vs
DP

або:
без стиснення
vs
Huffman

або:
random search
vs
GA

або:
baseline classifier
vs
MLP

8.14. Перед кожним експериментом сформулюйте питання

Не просто:

«Ми запустили DP».

А:

Чи дає DP кращий результат за greedy і якою ціною за часом?

Або:

Чи дає перцептрон кращий вибір стратегії, ніж фіксоване правило?

Або:

Чи знаходить GA кращі параметри, ніж випадковий пошук при однаковому бюджеті?

Це і є дослідницьке питання.

8.15. Описуйте умови експерименту

Для відтворюваності потрібно зафіксувати:

dataset
seed
parameters
commit
environment
number of runs

Не обов'язково записувати все це в кожному абзаці.

Можна створити окремий підрозділ:

«Умови експериментів».

8.16. Один експеримент — одна зміна

Якщо хочемо перевірити вплив DP, інші параметри повинні залишатися однаковими.

Правильно:

same data
same block size
same seed

greedy

vs

DP

Неправильно:

greedy + block=4096

vs

DP + block=16384 + інший seed

У другому випадку незрозуміло, що саме спричинило різницю.

8.17. Не показуйте лише найкращий запуск

Якщо алгоритм залежить від випадковості, один результат може бути випадково хорошим.

Тому для GA або MLP бажано проводити кілька запусків.

Наприклад:

```
run 1: ...  
run 2: ...  
run 3: ...
```

Потім можна навести:

середнє
медіану
мінімум
максимум

залежно від задачі.

8.18. Графіки повинні відповідати на питання

Не потрібно будувати графік лише тому, що «у звіті має бути графік».

Хороший графік допомагає побачити закономірність.

Наприклад:

generation
→ best fitness

показує роботу GA.

Або:

budget
→ result quality

показує вплив доступного ресурсу.

Графік повинен мати:

- назву;
- підпис осі X;
- підпис осі Y;
- одиниці вимірювання, якщо вони є.

8.19. Не використовуйте 3D-графік там, де достатньо двох ліній

Чим складніша візуалізація, тим важче її читати.

Мета графіка — зробити результат зрозумілішим.

Не красивішим.

8.20. Аналіз важливіший за самі числа

Припустимо, отримано:
greedy = 13532
DP = 13532

Не потрібно писати:

«DP не дав покращення».

І переходити далі.

Потрібно подумати:

Чому?

Можливо, для цих даних greedy вже робить оптимальні вибори.
Це важливий результат.

8.21. Нульовий виграш — теж результат

У науковому або інженерному експерименті не існує вимоги:

«Складніший алгоритм зобов'язаний перемогти».

Можна отримати:

DP improvement = 0%

або:

ML predictor improvement ≈ 0

або:

GA $\approx$ random search

Це не помилка, якщо експеримент проведено правильно.

Але потрібно пояснити можливу причину.

8.22. Негативний результат теж не потрібно приховувати

Можлива ситуація:

нова оптимізація

→ система стала повільнішою

Це теж корисна інформація.

Наприклад:

Використання складнішого пошуку дало додатково 0,3% стиснення, але збільшило час роботи у 4,7 раз. Тому для базової конфігурації цей режим не використовується.

Це сильний висновок.

8.23. Ablation study

Корисно показати, як змінюється система при поступовому додаванні компонентів.

Наприклад:

```
baseline
↓
+ Huffman
↓
+ DP
↓
+ classifier
↓
+ GA
```

Для кожного етапу вимірюється основна метрика.

Тоді можна побачити:

який компонент реально дав найбільший внесок.

8.24. Не приписуйте алгоритму те, чого не вимірювали

Якщо було виміряно тільки час, не можна написати:

«Алгоритм використовує менше пам'яті».

Пам'ять не вимірювалася.

Якщо не було експерименту, краще написати:

«Очікується менше використання пам'яті через...»

і чітко позначити, що це теоретичне припущення.

8.25. Розділ «Відомі обмеження»

У хорошому звіті повинно бути чесно написано, що система не вміє або де вона працює погано.

Наприклад:

- дуже великі файли обробляються повільно;

- classifier гірше розпізнає два певні класи;
- GA потребує багато запусків fitness;
- точний DP стає дорогим при великому бюджеті;
- експертний механізм не реалізований.

Це не знижує якість звіту.

Навпаки, показує, що команда розуміє межі власного рішення.

8.26. Висновки

Висновок не повинен бути повторенням змісту.

Погано:

«Було реалізовано Huffman, DP, MLP, GA...»

Це список виконаних робіт.

Краще:

«На тестовому корпусі DP давав кращий результат за greedy для структурованих даних, але майже не відрізнявся на випадкових. Найбільший вплив на загальну метрику дало стиснення, тоді як додаткова оптимізація параметрів за допомогою GA дала менший приріст».

Це вже висновок із результатів.

8.27. Внесок учасників команди

У звіті потрібно чітко вказати внесок кожного учасника.

Не:

Усі робили все.

А приблизно:

Студент 1:

Huffman, format encoder, unit tests.

Студент 2:

greedy, DP, експеримент greedy vs DP.

Студент 3:

MLP, feature extraction, classifier evaluation.

При цьому внесок повинен відповідати історії репозиторію та заявленим зонам відповідальності.

8.28. Якщо робота виконувалася спільно

Це нормально.

Наприклад:

`integration tests:`

Студент 2 + Студент 5

або:

архітектура:

вся команда

Не потрібно штучно приписувати кожен файл лише одній людині.

Але особиста зона кожного повинна залишатися зрозумілою.

8.29. Посилання на код

У тексті звіту можна вказувати:

`src/compression/huffman.py`

або конкретний модуль.

Не потрібно вставляти весь код.

Викладач за потреби відкриє його в репозиторії.

8.30. Результати повинні генеруватися програмою

Числові результати, які потрапляють у звіт, бажано отримувати автоматично.

Наприклад:

`python experiments/run_dp_vs_greedy.py`

створює:

`results/dp_vs_greedy.csv`

а вже на його основі будується графік.

Це значно краще, ніж вручну переносити числа з консолі.

8.31. Зберігайте сирі результати

Крім графіка:

`ga_fitness.png`

корисно зберігати:

`ga_fitness.csv`

або:

ga_fitness.json

Тоді результат можна перевірити або перебудувати.

8.32. Результати повинні відповідати фінальній версії коду

Дуже важливе правило:

Усі фінальні таблиці, графіки і метрики повинні бути відтворюваними з фінального commit або tag.

Якщо після експерименту код змінився, потрібно переконатися, що результат усе ще актуальний.

8.33. Як оформлювати таблиці результатів

У таблицю потрібно включати тільки ті показники, які допомагають порівнювати варіанти.

Наприклад:

Алгоритм
Результат
Час

Не потрібно створювати таблицю з двадцяти колонок, якщо п'ятнадцять із них не обговорюються.

Після таблиці обов'язково потрібен короткий текст:

Що з неї видно?

8.34. Не дублюйте одну інформацію тричі

Якщо числовий результат уже є в таблиці, не потрібно переписувати всі числа в текст.

Текст повинен давати аналіз.

Наприклад:

«DP перевершив greedy у чотирьох із семи груп, найбільша різниця спостерігалася для структурованих даних».

Читач за потреби подивиться точні числа в таблиці.

8.35. Скриншоти

Скриншот консолі не є основним способом представлення результатів.

Не рекомендується вставляти сторінки типу:

Screenshot 1: terminal
Screenshot 2: terminal
Screenshot 3: terminal

Краще:

- таблиця;
- графік;
- короткий log;
- текстовий результат.

Скриншот доречний лише тоді, коли він справді демонструє щось, що важко показати інакше.

8.36. Формат чисел

Потрібно використовувати однакову точність.

Наприклад:

0.812

0.746

0.903

а не:

0.812345128

0.7

0.903112314778

Надмірна кількість знаків після коми створює ілюзію точності, якої експеримент може не мати.

8.37. Одиниці вимірювання

Не пишіть:

time = 24

Незрозуміло:

24 s?

24 ms?

24 min?

Пишіть:

24.1 s

Так само для:

bytes
KiB
MiB
bits
%

8.38. Округлення

Розрахунки можна виконувати з повною точністю.

Для представлення результату значення округлюються розумно.

Наприклад:

`macro-F1 = 0.873`

зазвичай достатньо.

8.39. Відтворюваність звіту

До фінального захисту команда повинна мати можливість показати:

Ось команда, яка створює цей результат.

Наприклад:

`python experiments/run_all.py --seed 2001`

Після запуску отримуються:

`results/`

з даними для звіту.

Це дуже хороший рівень організації роботи.

8.40. Використання зовнішніх джерел

Якщо в роботі використовувалися:

- стаття;
 - книга;
 - документація;
 - сторонній алгоритмічний опис,
- на нього потрібно послатися.

Не потрібно ставити посилання на очевидні речі в кожному реченні.

Але джерело алгоритму або нетривіального технічного рішення повинно бути зрозумілим.

8.41. Використання ШІ

Якщо системи штучного інтелекту використовувалися під час роботи, це відображається відповідно до правил дисципліни та у META-USAGE.md.

Сам факт використання ШІ не замінює авторства студента.

Студент усе одно відповідає за:

- правильність коду;
- правильність формул;
- правильність експериментів;
- правильність висновків.

8.42. Перед здачею звіту перевірте узгодженість

Перед фінальною задачею потрібно пройти по звіту і запитати:

Чи всі назви алгоритмів однакові?

Чи однаково позначені змінні?

Чи збігаються числа в тексті й таблицях?

Чи актуальні графіки?

Чи відповідають вони final tag?

Чи немає опису функцій, яких уже немає в коді?

Після кількох тижнів активної розробки такі розбіжності виникають дуже легко.

8.43. Перевірте, чи можна зрозуміти звіт без автора

Корисний тест:

Дайте звіт іншому студенту, який не працював над вашим модулем.

Попросіть його відповісти:

- що робить система;
- навіщо використовується DP;
- що порівнювалося;
- який алгоритм виявився кращим;
- які є обмеження.

Якщо він не може цього зрозуміти, звіт потрібно зробити яснішим.

8.44. Рекомендований обсяг

Не потрібно намагатися зробити звіт максимально великим.

Хороший звіт — не найдовший.

Він повинен містити достатньо матеріалу, щоб пояснити:

задачу
алгоритми
реалізацію
тестування
експерименти
висновки

але не повинен перетворюватися на копію підручника або роздруківку вихідного коду.

8.45. Чек-лист готовності звіту

Перед фінальною задачею перевірте:

- ☐ Постановка задачі зрозуміла?
- ☐ Є схема системи?
- ☐ Пояснено роль основних алгоритмів?
- ☐ Наведено складність?
- ☐ Описано тестування?
- ☐ Є порівняльні експерименти?
- ☐ Умови експериментів зафіксовані?
- ☐ Є baseline?
- ☐ Є аналіз результатів?
- ☐ Негативні результати не приховані?
- ☐ Описані обмеження?
- ☐ Вказаний внесок кожного студента?
- ☐ Числа відповідають final tag?
- ☐ Результати можна відтворити?
- ☐ META-USAGE актуальний?

8.46. Головне правило підготовки звіту

Звіт не повинен переконувати читача словами:

«Наша система дуже хороша».

Він повинен дозволити читачу самому побачити це з результатів.

Тому найсильніша структура пояснення:

ми поставили питання

- провели контрольований експеримент
- отримали результат
- пояснили результат
- зробили висновок

Саме так код перетворюється на завершену інженерну роботу.

9. Спеціальні методичні вказівки до Проєкту 1

«Захищений канал телеметрії з обмеженою пропускнуою здатністю»

Проєкт 1 моделює ситуацію, у якій група апаратів або датчиків передає телеметрію на наземну станцію.

Канал зв'язку при цьому неідеальний.

Він має обмежену пропускну здатність, може втрачати пакети, створювати затримки, а сам трафік може бути доступний сторонньому спостерігачу.

Тому задача полягає не просто в тому, щоб передати всі дані.

Потрібно навчитися відповідати на значно цікавіше питання:

Що саме передавати, у якому порядку, як стискати, як захищати і як не втратити найважливіші дані, коли ресурсу не вистачає?

Саме тому в одному проєкті поєднуються:

- перцептрон;
- кодування Гаффмана;
- рюкзак 0/1;
- жадібні алгоритми;
- криптографічний захист;
- генетичний алгоритм;
- моделювання втрат у каналі.

9.1. Загальна постановка задачі

Система отримує потік телеметрії.

Кожен кадр містить вимірювання або стан апарата.

У телеметрії можуть бути:

- плавні сигнали, наприклад висота або температура;
- різкі сигнали, наприклад вібрація;
- дискретні стани;
- випадковий шум;
- критичні кадри, втрата яких особливо небажана.

Передавач повинен:

отримати телеметрію

→ проаналізувати кадри

→ визначити їх пріоритет

→ стиснути

→ сформувати пакети

→ визначити порядок передавання

→ зашифрувати

- додати контроль автентичності
- передати через модель каналу

Приймач виконує зворотний шлях:
отримати пакет

- перевірити НМАС
- розшифрувати
- розібрати формат
- декодувати дані
- відновити телеметрію

Модель каналу студент не реалізує.

Команді видається готовий файл `channel.ru`, однаковий для всіх команд. Його не потрібно і не можна змінювати. Це важливо, тому що всі команди повинні перевіряти свої алгоритми в однаковій моделі втрат і затримок.

9.2. Головна ідея проєкту

Якби канал був нескінченно швидким і ніколи нічого не втрачав, задача була б простою:
передати все

Але в реальному сценарії це не завжди можливо.

Тому система повинна робити вибір.

Наприклад, є десять кадрів, але поточний пакет дозволяє передати лише п'ять.

Тоді потрібно відповісти:

Які п'ять кадрів найважливіші?

Саме тут з'являється задача рюкзака.

Кожен кадр має:

вартість

тобто кількість бітів, необхідних для його передавання,

і:

цінність

тобто його важливість.

Далі система повинна знайти набір кадрів з найбільшою сумарною цінністю, який поміщається в пакет.

9.3. Архітектура Проєкту 1

Логіку системи можна уявити так:

телеметрія
↓
класифікація кадрів
↓
призначення пріоритетів
↓
стиснення
↓
пакування
↓
планування відправлення
↓
AES-256-CTR
↓
HMAC
↓
канал
↓
приймач
↓
відновлена телеметрія

Кожен блок відповідає окремій алгоритмічній задачі.
Головне — не намагатися будувати весь цей ланцюг одночасно.

9.4. Що студент отримує від викладача

Команда отримує готовий набір матеріалів.

До нього входять:

- FORMAT_SPEC.md;
- CHANNEL_SPEC.md;
- team_config.json;
- channel.py;
- train.csv;
- valid.csv;
- session.csv;
- session.bin;
- еталонні пакети;
- пошкоджені пакети;
- файл з очікуваними кодами помилок.

Викладач окремо зберігає:

- `holdout.csv`;
- приховані seed моделі каналу.

Студенти їх не отримують.

Це зроблено навмисно.

`valid.csv` можна використовувати для налагодження моделі.

`A holdout.csv` використовується викладачем лише для фінальної оцінки класифікатора.

Так само відкритий seed каналу можна використовувати під час розробки, але офіційне значення Q рахується на кількох прихованих seed.

9.5. Перший етап — навчитися читати кадри

Не потрібно починати з нейронної мережі.

Спочатку потрібно переконатися, що ви правильно читаєте телеметрію.

Створіть функцію, яка читає один кадр.

Потім:

прочитати 1 кадр

→ перевірити всі поля

Далі:

прочитати 10 кадрів

і лише потім:

прочитати весь `session.bin`

На цьому етапі потрібно перевірити:

- довжину кадру;
- типи полів;
- допустимі діапазони;
- порядок байтів;
- обробку кінця потоку.

9.6. Перший важливий тест

Для одного кадру повинно виконуватися:

`frame`

→ `serialize`

→ `parse`

→ `restored_frame`

i:

```
frame == restored_frame
```

Якщо цей тест не працює, переходити до стиснення ще рано.

9.7. Другий етап — мінімальний пакет без стиснення

Наступний крок — створити найпростішу версію пакета.

Поки без:

- MLP;
- Huffman;
- DP;
- GA.

Просто:

кілька кадрів

→ запис у пакет

→ читання пакета

→ відновлення кадрів

Це дозволяє окремо перевірити формат.

На цьому етапі потрібно навчитися правильно записувати:

- заголовок;
- кількість кадрів;
- службові поля;
- payload.

9.8. FORMAT_SPEC є контрактом

Формат пакета не можна змінювати на власний розсуд.

Якщо специфікація задає:

`big-endian`

треба використовувати `big-endian`.

Якщо поле має:

2 байти

не можна використовувати 4.

Якщо зарезервоване поле повинно бути нульовим, ненульове значення має призводити до відмови.

Власний передавач і власний приймач можуть мати однакову помилку.

Тому справжня перевірка — це робота з еталонними пакетами викладача.

9.9. Третій етап — кодування Гаффмана

Після того як пакет без стиснення працює, додається Гаффман.

Не інтегруйте його одразу в повну систему.

Спочатку перевірте окремо:

bytes

→ Huffman encode

→ Huffman decode

→ bytes

Потім потрібно додати:

- підрахунок частот;
- чергу з пріоритетами;
- побудову дерева;
- визначення довжин кодів;
- канонічні коди;
- bit writer;
- bit reader.

9.10. Канонічний Гаффман

У пакет не потрібно записувати все дерево.

Замість цього передаються довжини кодів.

За ними приймач повинен побудувати ті самі канонічні коди.

Саме тому алгоритм побудови кодів повинен бути детермінованим.

Перевірте:

encoder lengths

==

decoder lengths

і після цього:

decoded == original

9.11. Обов'язкові крайні випадки для Гаффмана

Потрібно перевірити:

порожній потік

один символ

два символи

усі символи приблизно рівноймовірні

Особливо важливий вироджений алфавіт з одним символом.
Він входить до базових приймальних перевірок проєкту.

9.12. Четвертий етап — рюкзак 0/1

Тепер можна переходити до однієї з центральних задач проєкту.

Є набір кадрів. Кожен має вагу `weight_i` і цінність `value_i`. Є місткість пакета `W`.

Але ці три величини не беруться зі стелі, і саме тут студенти найчастіше помиляються.

Звідки береться вага

Наївне визначення «вага кадру — це його розмір після стиснення» некоректне. Розмір залежить від таблиці Гаффмана, а таблиця залежить від того, які кадри обрано. Виникає циклічна залежність, і задача стає невизначеною.

Тому порядок жорсткий:

1. взяти вікно кандидатів (`candidate_window_frames`, базово 64 кадри)
2. побудувати ОДНУ таблицю Гаффмана за частотами байтів УСІХ кандидатів
3. `weight_i` = довжина 12 байтів кадру в бітах за цією таблицею
4. розв'язати рюкзак
5. закодувати обрані кадри ТІЄЮ САМОЮ таблицею

Після кроку 2 таблиця не змінюється, тому ваги фіксовані й рюкзак коректний.

Зверніть увагу: усі кадри мають однакові 12 байтів, але ваги в них різні. Кадр із рідкісними байтами коштує більше біт. Якби ваги були однакові, задача виродилася б у просте сортування за пріоритетом.

Звідки береться цінність

`value_i` = пріоритет(`клас_i`) + `critical_бонус` × [`critical_i` == 1]

Числа задає `team_config.json`, секція `пріоритети_кадрів`. Вони однакові для всіх команд, тому розв'язки різних команд можна порівнювати.

Звідки береться місткість

$W = 8 \times (\text{max_payload_bytes} - 14 - 2 \times \text{table_nsym})$

де `max_payload_bytes` = `max_packet_bytes` - 43 - 32, а `2 × table_nsym` — розмір таблиці кодів. Точні формули наведені у `FORMAT_SPEC.md`, розділ 3.5.

Вікно кандидатів обмежене не випадково: таблиця Гаффмана росте разом із кількістю різних байтів, і за зовеликого вікна вона перестав вміщатися в пакет, а W стає від'ємним.

Потрібно вибрати підмножину кадрів так, щоб:
 $\text{sum}(\text{weight_i}) \leq W$

і при цьому:
 $\text{sum}(\text{value_i})$

була максимальною.

9.13. Спочатку зробіть brute force

Перед DP бажано створити повний перебір для малих задач.

Наприклад:

$n \leq 15$

Тоді можна перевірити:

$DP == \text{brute_force}$

на багатьох випадкових маленьких наборах.

Це одна з найкращих перевірок правильності рюкзака.

9.14. Реалізація DP

Визначте стан:

$DP[i][w]$

як максимальну цінність, яку можна отримати з перших i кадрів при доступному бюджеті w .

Тоді в кожній точці є два варіанти:

не брати кадр

або:

взяти кадр

якщо він поміщається.

Потрібно зберегти найкращу цінність і навчитися відновлювати набір вибраних кадрів.

9.15. Не забудьте про фактичний розмір пакета

Важлива помилка — вважати, що вся місткість пакета доступна тільки для кадрів.

Насправді пакет має службові поля.

Тому доступний бюджет повинен рахуватися відповідно до `FORMAT_SPEC.md` і `team_config.json`.

Не можна оптимізувати рюкзак для одного W , а потім створювати пакет більшого розміру.

Фінальна перевірка:

`actual_packet_size <= max_packet_bytes`

9.16. П'ятий етап — greedy baseline

Після точного DP потрібно реалізувати простий greedy.

Наприклад:

`sort by value / weight`

після чого брати кадри, поки вони поміщаються.

Цей алгоритм не є гарантовано оптимальним для рюкзака 0/1.

Саме це і потрібно дослідити.

9.17. Головний експеримент Проєкту 1

Порівняйте:

DP

vs

greedy

на тих самих наборах.

Виміряйте:

- сумарну цінність;
- час роботи;
- різницю в результаті.

Не потрібно доводити, що DP завжди набагато кращий.

Можлива ситуація:

DP = 100

greedy = 100

Тоді потрібно не відмахнутися словом «випадково», а пояснити причину.
Наприклад:

На цьому наборі предмети з найбільшим відношенням цінності до ваги заповнюють місткість без залишку, тому жадібний вибір збігається з оптимальним.

або:

Ваги кадрів у цьому вікні майже однакові, тому впорядкування за відношенням збігається з упорядкуванням за цінністю, і жадібний алгоритм дає оптимум.

Саме таке пояснення показує розуміння алгоритму.

9.18. Шостий етап — планувальник черги

Після формування пакетів потрібно визначити порядок їх передавання.

Тут використовується жадібна евристика.

Наприклад, можна враховувати:

- максимальний пріоритет пакета;
- кількість критичних кадрів у ньому;
- сумарну цінність;
- розмір пакета, а отже час, який він забере з бюджету.

Окремих дедлайнів у пакетів немає. Недоставка настає або через втрату в каналі, або через вичерпання `transmission_budget_s`. Саме бюджет і робить порядок відправлення значущим.

Потрібно пам'ятати:

Планувальник є евристикою.

Не потрібно називати його оптимальним, якщо оптимальність не доведена.

У самому проєкті порядок відправлення прямо розглядається як евристичний, а greedy-рюкзак — як baseline для порівняння з точним DP.

9.19. Сьомий етап — підключення моделі каналу

Тепер пакет можна передавати через `channel.py`.

До цього моменту система повинна вже правильно працювати у спокійному сценарії.

Починайте саме з нього.

Послідовність:

спокійний сценарій

→ робочий сценарій

→ аварійний сценарій

Не починайте одразу з аварійного.

9.20. Три сценарії каналу

У проєкті передбачені три типи умов:

спокійний

— втрат практично немає, ресурс достатній;
робочий

— є втрати, змінні затримки і обмеження;
аварійний

— великі втрати, вузький канал і жорсткий бюджет часу.

Ці сценарії потрібні для того, щоб побачити:

як алгоритм поводить себе не лише в ідеальних умовах.

9.21. Transmission budget

Крім ширини каналу, сценарій задає часовий бюджет.

Пакет, який фізично не встигає передатися до завершення бюджету, не передається.

Це означає, що порядок пакетів має значення.

Якщо спочатку відправити багато маловажливих пакетів, критичні дані можуть залишитися в кінці й не встигнути пройти.

Саме тому планування тут є реальною частиною задачі.

9.22. Восьмий етап — класифікатор

Тепер можна додати MLP.

Його базове завдання — класифікувати кадр.

Класи:

0 — норма

1 — аномалія

2 — перехідний режим

Результат класифікації використовується для визначення пріоритету.

Поле **critical** не є міткою класу

Кадр має окреме поле **critical**. Воно означає важливість кадру для доставки, а не його клас, і задається правилами, не пов'язаними з класом. Частка кадрів із **critical = 1** приблизно однакова в усіх трьох класах.

Тому подавати **critical** на вхід мережі можна: витоку мітки це не дає. Але й розраховувати, що воно саме по собі розв'яже задачу, не варто — класифікатор, побудований лише на цьому полі, дає рівень випадкового вгадування.

Ознак одного кадру недостатньо

Це базова вимога, а не порада. Клас 1 розпізнається переважно за миттєвою вібрацією, але клас 2 відрізняється від норми тільки за поведінкою в часі:

нестабільністю mode і дрейфом висоти та температури. Модель, що бачить лише поля одного кадру, порогу F1_min не досягає.

Мінімальний набір ознак:

Δaltitude, Δtemp	різниці із попереднім кадром
vibration	модуль вібрації
ковзне відхилення vibration і altitude по вікну	
частота зміни mode у вікні	

Розмір вікна близько восьми кадрів працює добре.

9.23. Не починайте MLP з повного датасету

Спочатку перевірте:

forward pass

на маленькому прикладі.

Потім:

loss

Потім:

backpropagation

Потім:

навчання на невеликому наборі

і тільки після цього переходьте до `train.csv`.

9.24. Train, valid i holdout

Правильный порядок:

train.csv

→ навчання

valid.csv

→ вибір параметрів

holdout.csv

→ фінальна оцінка викладачем

Студент не отримує `holdout.csv`.

Ще одна обов'язкова частина протоколу — кілька ініціалізацій. Навчання повторюється щонайменше тричі з різними початковими вагами, і модель обирається за `macro-F1` на `valid.csv`.

Це не формальність. Одна й та сама мережа на тих самих даних дає розкид `macro-F1` від 0,30 до 0,90 залежно від початкових ваг: інколи навчання збігається в поганий локальний мінімум і мережа згортається до мажоритарного класу. Вибір за валідацією прибирає цю залежність.

Це захищає фінальну оцінку від підгонки моделі.

9.25. Перевірка класифікатора

Не достатньо показати:
`accuracy = ...`

Основною метрикою є `macro-F1`.

Також потрібно подивитися матрицю помилок.

Вона допомагає відповісти:

Які класи модель плутає?

Наприклад:
`аномалія`
`↔`
`перехідний режим`

можуть виявитися складнішими для розділення, ніж нормальний режим.

9.26. Як класифікатор впливає на систему

MLP не повинен існувати окремо від проєкту.

Його вихід повинен реально впливати на:
`priority(frame)`

а отже — на:
`DP`

і:
`scheduler`

Тому корисний експеримент:
`фіксовані пріоритети`
`vs`
`пріоритети від classifier`

і порівняння Q .

9.27. Просунутий прогноз наступного кадру

На просунутому рівні MLP може прогнозувати наступне значення телеметрії.

Тоді замість самого значення кодується залишок:

`residual = actual - predicted`

Якщо сигнал плавний, залишки можуть мати меншу ентропію.

Але для випадкового шуму цього може не бути.

Тому потрібно виміряти:

`entropy(original)`

i:

`entropy(residual)`

а не просто припустити, що прогноз завжди допомагає.

9.28. Дев'ятий етап — криптографічний захист

Криптографію додавайте тоді, коли формат і стиснення вже працюють.

Порядок обробки:

стиснення

→ шифрування

→ HMAC

У базовому форматі використовуються:

PBKDF2-HMAC-SHA256

AES-256-CTR

HMAC-SHA256

з окремими ключами `K_enc` і `K_mac`.

9.29. Виведення ключів

Пароль не використовується безпосередньо як AES-ключ.

Спочатку:

`password + salt`

→ PBKDF2-HMAC-SHA256, 100 000 ітерацій

→ 64 bytes

PBKDF2 навмисно повільний, і саме це захищає від перебору паролів. Але сіль однакова для всього сеансу, отже ключі теж. Тому виведення виконується один раз на початку, а отримані ключі кешуються. Перерахунок для кожного

пакета не зламає систему, але KDF почне домінувати у часі й зіпсує компоненту T метрики Q.

Після цього:

first 32 bytes

→ K_enc

last 32 bytes

→ K_mac

Використовувати один і той самий ключ для AES та HMAC не можна.

9.30. AES-256-CTR

У режимі CTR формується блок:

nonce || counter

де counter починається з 1.

Для кожного пакета nonce повинен бути унікальним у межах одного ключа.

Це дуже важливо.

Не можна зробити:

packet 1 → nonce X

packet 2 → nonce X

з тим самим ключем.

Повторне використання nonce у CTR є серйозною криптографічною помилкою.

9.31. HMAC перевіряється першим

На стороні приймача порядок:

прочитати зовнішній заголовок

→ перевірити базову структуру

→ перевірити HMAC

→ тільки після цього decrypt

Не потрібно розшифровувати пошкоджений пакет.

Це одна з базових вимог безпеки проєкту.

9.32. Обов'язкові negative tests для захисту

Потрібно перевірити:

неправильний пароль

змінений байт ciphertext

змінений HMAC

обрізаний пакет

неправильний MAGIC

ненульове поле reserved

Очікуваний результат — контрольована відмова з передбаченим кодом, а не просто виняток. Коди визначені у `FORMAT_SPEC.md`:

Ситуація	Код
зіпсований байт, зіпсований тег, невірний пароль	<code>E_HMAC</code>
обрізаний пакет	<code>E_LENGTH</code>
зіпсований MAGIC або version	<code>E_MAGIC</code>
ненульове reserved, невідомі біти flags	<code>E_PAYLOAD</code>
некоректна таблиця кодів	<code>E_CODE</code>

Кожен із цих випадків має відповідний файл у виданому наборі пошкоджених пакетів, а очікуваний код записаний у `expected_failures.json`.

Не:

`IndexError`

або:

`UnicodeDecodeError`

9.33. Обов’язковий експеримент з порядком операцій

Потрібно порівняти:

`compress`

→ `encrypt`

із:

`encrypt`

→ `compress`

Другий варіант повинен стискатися значно гірше, тому що зашифровані дані виглядають майже випадковими.

Це не просто теоретичне питання.

Потрібно реально виміряти розмір.

9.34. Десятий етап — основна метрика Q

Система оцінюється не лише за одним показником.

У проєкті використовуються три компоненти:
S – ефективність передавання;
R – частка відновлених критичних кадрів;
T – швидкодія.

Підсумкова метрика:
 $Q = \alpha S + \beta R + \gamma T$

де:
 $\alpha + \beta + \gamma = 1$

Точні коефіцієнти беруться з `team_config.json`.

9.35. Не оптимізуйте лише стиснення

Можна дуже сильно зменшити кількість переданих бітів, але при цьому втратити багато критичних кадрів.

Тоді система погана.

Так само можна дублювати все по десять разів і майже не втрачати критичні дані, але повністю перевищити ресурс каналу.

Тому оптимізується саме баланс.

9.36. Вимірюйте Q до оптимізації

Перед GA потрібно мати:
Q_baseline

Наприклад:
fixed parameters
→ Q_baseline

Після цього:
GA parameters
→ Q_GA

І лише тоді можна говорити про покращення.

9.37. Одинадцятий етап — генетичний алгоритм

GA використовується для підбору параметрів системи.

На базовому рівні хромосома містить три параметри.

Наприклад:

block_size
compression_threshold
priority_weight

Кожна хромосома задає одну конфігурацію системи.

9.38. Fitness

Fitness має бути пов'язана з реальною метрикою системи.

Найприродніше:

`fitness = Q`

або функція, прямо визначена умовою.

Перш ніж запускати GA, переконайтеся, що:

`fitness(configuration)`

працює правильно сама по собі.

9.39. Оператори GA

Базова реалізація повинна містити:

population
selection
crossover
mutation
elitism

Не намагайтеся одразу реалізувати п'ять типів мутацій і складну адаптацію.

Спочатку зробіть просту правильну версію.

9.40. Перевірка елітизму

Якщо Q максимізується:

`best_final >= best_initial`

Це одна з базових sanity checks GA.

Але це не означає:

GA обов'язково має бути кращим за ручні параметри.

У проєкті прямо вимагається лише коректне порівняння і пояснення результату.

9.41. GA проти random search

Порівнювати потрібно при однаковому бюджеті.

Наприклад:

GA:

200 fitness evaluations

Random Search:

200 fitness evaluations

Потім порівняти найкращі Q.

Якщо random search переміг, не приховуйте це.

Поясніть можливу причину.

9.42. Дванадцятий етап — наскрізний тракт

Після інтеграції всіх базових компонентів система повинна проходити:

session

→ classify

→ prioritize

→ compress

→ pack

→ schedule

→ encrypt

→ HMAC

→ channel

→ verify

→ decrypt

→ decode

→ recover

→ evaluate

Після цього автоматично обчислюється Q.

9.43. Перевірка на трьох сценаріях

Повний прогін потрібно виконати для:

спокійного

робочого

аварійного

сценаріїв.

Для кожного зберігайте щонайменше:

S
R
T
Q

Не дивуйтеся, якщо найкращі параметри для спокійного і аварійного сценаріїв різні.

Це нормальний результат.

9.44. Приховані seed каналу

Під час розробки команда має відкритий seed.

Але не потрібно будувати алгоритм, який використовує знання конкретної майбутньої послідовності втрат.

Офіційне Q викладач обчислює на п'яти прихованих seed і усереднює результати.

Алгоритм має бути загальним.

Не:

«Я знаю, що пакет №17 буде втрачено».

А:

«У цьому пакеті високий ризик втрати важливих даних за загальною стратегією».

9.45. Що потрібно дослідити експериментально

У фінальному звіті Проекту 1 особливо важливі такі порівняння:

DP

vs

greedy

без classifier

vs

з classifier

без GA

vs

з GA

compress → encrypt

vs

encrypt → compress

спокійний
vs
робочий
vs
аварійний сценарій

Ці експерименти дозволяють побачити внесок кожного алгоритмічного блоку.

9.46. Дублювання критичних кадрів

На просунутому рівні можна дублювати важливі кадри в сусідніх пакетах.
Це має підвищувати:

R

але збільшує:

B_sent

Тому питання не:

Чи підвищує дублювання надійність?

Це очевидно.

Питання:

Чи виправдовує приріст R додаткові біти?

Саме це потрібно виміряти.

9.47. Модуль атаки

Окремий модуль використовується під час тижня атак.

Базова криптоаналітична задача працює зі спеціально підготовленими слабкими шифрами.

AES атакувати не потрібно.

Мета — використати GA для пошуку ключа до навмисно слабкої схеми.

Окремо перевіряється цілісність:

змінити пакет

→ передати приймачу

Коректний приймач повинен відхилити його.

9.48. Аналіз трафіку

На просунутому рівні можна дослідити, чи можна визначити тип передаваних даних тільки за:

- розмірами пакетів;
- частотою;
- часовими характеристиками.

При цьому payload не розшифровується.

Потім можна додати padding і перевірити:
аccuracy атаки до padding

і:

аccuracy атаки після padding

а також виміряти вартість padding у бітах.

9.49. Покрокове тестування Проєкту 1

Рекомендований порядок тестів:

1. frame serialize/parse
2. packet без compression
3. Huffman round-trip
4. canonical Huffman
5. packet round-trip
6. DP vs brute force
7. greedy
8. DP vs greedy
9. classifier
10. scheduler
11. channel calm
12. channel working
13. channel emergency
14. AES known vector
15. HMAC
16. tamper tests
17. наскрізний тракт
18. Q
19. GA
20. GA vs random

Не переходьте до наступних пунктів, якщо попередні нестабільні.

9.50. Які помилки потрібно спеціально спробувати викликати

Створіть тести для:

empty stream

single frame

zero packet capacity

frame larger than available packet payload

wrong password

bad HMAC

truncated packet

bad Huffman table

invalid class

invalid header

channel drops many packets

Система повинна або правильно працювати, або контрольовано відмовлятися.

9.51. Що потрібно показати на Чекпоінті 1

До першого чекпоінту:

формат пакета

→ працює

передача без складної оптимізації

→ працює

Huffman

→ працює

еталонні пакети

→ читаються

baseline Q

→ виміряно

Головне питання Чекпоінту 1:

Чи є вже проста наскрізна система?

9.52. Що потрібно показати на Чекпоінті 2

До другого чекпоінту:

DP

greedy

DP vs greedy

scheduler

три сценарії каналу

classifier

На цьому етапі вже потрібно показати перші алгоритмічні експерименти.

9.53. Що потрібно показати на Чекпоінті 3

До третього чекпоінту:

AES

HMAC

KDF

повинні бути інтегровані.

Також:

GA

→ працює

GA vs random

→ порівняно

Q

→ рахується автоматично

і наскрізний тракт повинен бути стабільним.

9.54. Фінальні приймальні перевірки

Базова система повинна, зокрема:

- правильно обробляти один кадр;
- не падати на порожньому потоці;
- повністю відновлювати спокійний сценарій;
- виконувати заданий поріг втрат критичних кадрів у робочому сценарії;
- стабільно працювати в аварійному;

- правильно відновлювати канонічні коди;
- правильно обробляти один символ;
- мати DP не гірший за greedy за цінністю;
- класифікувати всі кадри;
- відхиляти змінені пакети;
- відхиляти неправильний пароль і обрізані дані;
- читати еталонні пакети.

9.55. Що потрібно мати у фінальних результатах

Рекомендується автоматично зберігати результати:

```
results/
├── dp_vs_greedy.csv
├── channel_scenarios.csv
├── classifier_metrics.json
├── ga_history.csv
├── ablation.csv
└── final_metrics.json
```

Назви можуть бути іншими.

Головне — щоб результати можна було повторити.

9.56. Які питання студент повинен уміти пояснити

До фінального захисту потрібно розуміти:

Чому greedy-рюкзак не гарантує оптимум?

Що означає $DP[i][w]$?

Чому Гаффман є жадібним алгоритмом?

Чому канонічний Гаффман дозволяє не передавати дерево?

Навіщо MLP у цьому проєкті?

Чому прогноз може допомагати для плавних сигналів і не допомагати для шуму?

Чому HMAC перевіряється до decrypt?

Чому CTR nonce не можна повторювати?

Чому K_{enc} і K_{mac} різні?

Що саме оптимізує GA?

Чому GA порівнюється з random search?

Який компонент найбільше вплинув на Q?

9.57. Типова неправильна стратегія

Не робіть так:
спочатку MLP
→ потім GA
→ потім crypto
→ наприкінці спробувати з'єднати

Правильніше:
format
→ round-trip
→ compression
→ DP/greedy
→ channel
→ classifier
→ crypto
→ GA
→ full evaluation

9.58. Головний критерій успіху Проєкту 1

У цьому проєкті не перемагає команда, яка:

передала найбільше даних.

І не команда, яка:

отримала найкраще стиснення.

І не команда, яка:

майже нічого не втратила, дублюючи все.

Потрібно знайти розумний баланс між:
кількістю переданих бітів

збереженням критичної інформації

і:
часом роботи

Саме цей баланс і відображає метрика Q.

9.59. Короткий маршрут студента

Усю роботу над Проєктом 1 можна запам'ятати так:
КРОК 1
Навчитися читати телеметрію

↓

КРОК 2

Зробити пакет без стиснення

↓

КРОК 3

Додати Huffman

↓

КРОК 4

Додати DP-рюкзак

↓

КРОК 5

Додати greedy і порівняти

↓

КРОК 6

Додати scheduler

↓

КРОК 7

Пропустити через channel.py

↓

КРОК 8

Навчити classifier

↓

КРОК 9

Додати AES + HMAC

↓

КРОК 10

Навчитися рахувати Q

↓

КРОК 11

Додати GA

↓

КРОК 12

Провести всі експерименти

↓

КРОК 13

Пройти negative tests

↓

КРОК 14

Зробити clean run і захист

9.60. Головна порада

Не намагайтеся одразу зробити «розумний захищений оптимальний канал».

Спочатку зробіть:

простий канал, який правильно передає один кадр.

Потім десять.

Потім пакет.

Потім стиснення.

Потім вибір.

Потім втрати.

Потім захист.

Потім оптимізацію.

Саме така послідовність робить складний Проєкт 1 зрозумілою серією невеликих задач.

10. Спеціальні методичні вказівки до Проєкту 2

«Розумний захищений архіватор файлів»

Проєкт 2 присвячений створенню власного архіватора.

Але це не просто програма, яка бере кілька файлів і записує їх в один файл.

Архіватор повинен одночасно розв'язувати кілька задач:

- аналізувати тип вхідних даних;
- знаходити повтори;
- стискати дані;
- вирішувати, коли стиснення взагалі не потрібне;
- оптимально розміщувати блоки у томах;
- захищати архів криптографічно;
- перевіряти його цілісність;
- відновлювати всі файли побайтово точно.

Тому основне питання цього проєкту можна сформулювати так:

Як побудувати архіватор, який добре стискає різні типи файлів, не витрачає час на беззмістовне стиснення, правильно розподіляє дані між томами і при цьому гарантує точне та безпечне відновлення?

Саме в цій задачі поєднуються LZ77, Гаффман, динамічне програмування, жадібні алгоритми, перцептрон, генетичний алгоритм і криптографічний захист.

10.1. Постановка задачі

На вході система отримує один або кілька файлів.

На виході потрібно отримати архів, який може складатися з одного або кількох томів.

Загальна схема:

файли
↓
аналіз
↓
розбиття на блоки
↓
LZ77
↓
Huffman
↓
пакування в томи
↓
AES-256-CTR

↓
HMAC
↓
архів

Розархіватор виконує зворотний шлях:
томи архіву

↓
перевірка структури
↓
перевірка HMAC
↓
розшифрування
↓
розбір блоків
↓
Huffman decode
↓
LZ77 decode
↓
склеювання файлів
↓
перевірка CRC32
↓
оригінальні файли

Найважливіша вимога:

Після розпакування кожен файл повинен точно збігатися з вихідним побайтово.

Якщо хоча б один байт відновлено неправильно, архіватор не виконав свою основну задачу.

10.2. Що означає «розумний архіватор»

Різні типи файлів поводяться при стисненні по-різному.

Наприклад, текст часто добре стискається.

У вихідному коді багато повторів.

У структурованих даних часто повторюються ключі, назви полів і роздільники.

Растрове зображення може мати сильну локальну структуру.

А вже стиснутий ZIP, JPEG або MP3 часто майже немає сенсу стискати ще раз.

Тому архіватор повинен не просто застосовувати один і той самий алгоритм до всього.

Він повинен уміти відповісти:

Що це за файл?

і після цього:

Яку стратегію до нього застосувати?

10.3. Що отримує команда

Команді видаються матеріали для розробки і перевірки.

Зокрема:

FORMAT_SPEC_P2.md
team_config.json
train/
valid/
corpus/
reference/

У каталозі `reference/` містяться:

- еталонні архіви;
- JSON-описи очікуваного вмісту;
- спеціально пошкоджені архіви;
- очікувані коди помилок.

Також команді видається робочий корпус із сімома типами даних.

Прихована вибірка `holdout/` студентам не видається.

Вона використовується викладачем для фінальної перевірки класифікатора.

10.4. Сім типів даних

У проєкті використовуються такі групи:

- 0 – природний текст
- 1 – вихідний код
- 2 – структуровані дані
- 3 – растр без стиснення
- 4 – виконувані файли
- 5 – вже стиснуті дані
- 6 – штучні дані

Кожна група потрібна не випадково.

Вона перевіряє іншу властивість архіватора.

Наприклад:

текст

→ багато статистичної надлишковості

код

→ довгі повтори

растр

→ локальні закономірності

JPEG / ZIP

→ практично відсутній додатковий виграш

нули

→ дуже сильне стиснення

випадковий шум

→ майже нестисливі дані

10.5. Перший етап — простий архів без стиснення

Не починайте з LZ77.

Не починайте з Huffman.

Не починайте з MLP.

Спочатку потрібно навчитися:

файл

→ записати в архів

→ прочитати архів

→ відновити файл

На цьому етапі блок можна зберігати без стиснення.

Для цього в форматі існує режим:

NO_COMPRESSION

Тоді:

coded_len = raw_len

а дані блока зберігаються як звичайні сирі байти.

10.6. Перший головний тест

Для одного файла:

original

→ archive

→ extract
→ restored

і після цього:
original == restored

Побайтово.
Після цього потрібно перевірити:
кілька файлів

і лише потім:
багатотомний архів

10.7. Почніть з одного тому

Поки одне томне представлення не працює стабільно, не потрібно реалізовувати багатотомність.

Спочатку:
1 файл
→ 1 том

потім:
кілька файлів
→ 1 том

і тільки після цього:
кілька файлів
→ кілька томів

10.8. Формат архіву є контрактом

Архів використовує формат SARC.

Точне розташування полів визначено в FORMAT_SPEC_P2.md.

Не потрібно дублювати всю специфікацію в коді у вигляді випадкових чисел.

Краще мати зрозумілі константи:
MAGIC = b"SARC"
VERSION = 1

і окремі функції для:
write_header
read_header
write_payload
read_payload

10.9. Зовнішній заголовок тому

Кожен том починається з відкритого транспортного заголовка.
Він містить службову інформацію, необхідну ще до розшифрування.
Наприклад:

- magic;
- версію;
- індекс тому;
- загальну кількість томів;
- salt;
- nonce;
- довжину ciphertext.

Цей зовнішній заголовок не потрібно плутати з таблицею файлів.

Імена файлів, їхні розміри, CRC32, тип і стратегія знаходяться в зашифрованій частині архіву.

10.10. Другий етап — таблиця файлів

Після читання базового заголовка потрібно реалізувати таблицю файлів.
Для кожного файла зберігаються:

ім'я
оригінальний розмір
CRC32
тип файла
стратегія

Дуже важливо:

orig_size

i:

CRC32

стосуються всього файла, навіть якщо його блоки розміщені в різних томах.

10.11. Безпечні імена файлів

Розархіватор не повинен дозволяти архіву записувати файл куди завгодно у файловій системі.

Наприклад, ім'я:

../secret.txt

не повинно прийматися.

Так само небезпечні:

..\secret.txt
/absolute/path
C:evil.txt

У базовій версії дозволяється лише безпечне ім'я файла без шляху.

Тобто:

- не порожнє;
- не . і не ..;
- без /;
- без \;
- без ::;
- без нульового байта.

Це потрібно перевіряти під час читання архіву, до будь-якого запису на диск, і відхиляти архів із кодом E_PAYLOAD.

Ім'я з архіву керує тим, куди розархіватор запише дані, тому воно є недовіреними даними — такими самими, як і будь-яке інше поле формату.

10.12. Третій етап — розбиття файла на блоки

Великі файли потрібно розбивати на блоки.

Наприклад:

```
file  
↓  
block 0  
block 1  
block 2  
...
```

Кожен блок має власний:

```
block_seq  
raw_len  
coded_len  
flags
```

Це дуже важливо для багатотомного архіву.

Один файл може початися в першому томі, продовжитися у другому і завершитися у третьому.

Після розпакування блоки склеюються за:

ім'я файла + block_seq

10.13. Спочатку протестуйте блоки без стиснення

Перед LZ77 зробіть:

```
file
→ split blocks
→ store raw blocks
→ read blocks
→ join
```

Перевірте:

```
joined == original
```

Це відокремлює помилки багатоблокового формату від помилок алгоритму стиснення.

10.14. Четвертий етап — LZ77

Тепер можна переходити до алгоритму стиснення.

LZ77 шукає повтори у вже прочитаних даних.

Замість:

```
abcsabcsabcsabc
```

можна частково передати:

```
abc
```

а далі:

```
повернутися назад
і скопіювати повтор
```

Елементи потоку бувають двох типів:

```
literal
```

або:

```
match(distance, length)
```

10.15. Спочатку реалізуйте decoder

Це часто корисна стратегія.

Напишіть простий decoder для послідовності:

```
literal
match
literal
...
```

і вручну створіть кілька token-наборів.

Наприклад:

```
lit 'a'  
lit 'b'  
lit 'c'  
match distance=3 length=6
```

Очікування:

```
abcsabcsabc
```

Тільки після того, як decoder стабільний, переходьте до пошуку збігів.

10.16. Жадібний LZ77

Базова стратегія:

у поточній позиції взяти найдовший знайдений збіг.

Наприклад:

```
position i  
→ search window  
→ longest match  
→ emit match
```

Якщо придатного збігу немає:

```
emit literal
```

10.17. Лінива жадібність

Другий базовий варіант — lazy greedy.

Ідея:

Іноді краще не використовувати хороший збіг зараз, якщо наступна позиція має значно кращий.

Тоді:

```
position i  
→ знайдений match  
→ подивитися i+1
```

Якщо в i+1 значно кращий збіг:

```
emit literal at i
```

i використовувати match наступним.

10.18. Обов'язкове порівняння LZ77

Потрібно виміряти:

greedy
vs
lazy greedy

На однакових файлах.
Для кожного варіанта зберігайте:

- розмір потоку;
- час;
- кількість літералів;
- кількість посилань.

Не потрібно очікувати, що lazy завжди переможе.
Потрібно пояснити результат.

10.19. Межі LZ77

У базовому форматі:

MIN_MATCH = 3
MAX_MATCH = 258
WINDOW = 32768

Ці межі є частиною формату.

Decoder повинен відхилити некоректне посилання.

Наприклад:

distance = 0

або:

distance > 32768

або посилання за межі вже відновленого буфера.

10.20. Важлива межа DIST

Для вікна:

32768

максимальний допустимий код відстані дорівнює:

15

Тому алфавіт DIST:

0...15

а не 0...16.

Це хороший приклад того, чому граничні значення потрібно перевіряти окремо.

Тести:

distance = 32767 → valid
distance = 32768 → valid
distance = 32769 → reject

10.21. П'ятий етап — Huffman

LZ77 створює потік літералів і посилань.

Після цього значення потрібно закодувати Huffman-кодом.

У форматі використовуються три окремі таблиці:

LIT
LEN
DIST

Це дозволяє враховувати різну статистику для різних типів символів.

10.22. Алфавіти Huffman

У базовій версії:

LIT: 0...256

де:

0...255

— байти,

а:

256

— спеціальний маркер «далі йде посилання».

Для довжин:

LEN: 0...255

Для відстаней:

DIST: 0...15

Parser повинен перевіряти, що символ не виходить за межі свого алфавіту.

10.23. Канонічний Huffman

У файл записуються не самі бітові коди, а довжини.

За ними decoder однозначно відновлює канонічні коди.

Перевіряйте окремо:

lengths
→ canonical codes

Потім:
encode
→ decode
→ original

10.24. Максимальна довжина коду

У форматі максимальна довжина Huffman-коду:
15 біт

Якщо звичайне дерево дало більшу довжину, потрібно застосувати length-limited Huffman.

Не можна просто:
обрізати код

або:
злити два символи

Тоді decoder уже не зможе однозначно відновити дані.

10.25. Рівність Крафта

Для більш ніж одного символу довжини повинні формувати повний префіксний код.

Тобто:
 $\sum 2^{-L} = 1$

Якщо рівність не виконується, таблиця некоректна.

Parser повинен повернути помилку, а не намагатися «виправити» її.

10.26. Вироджений випадок Huffman

Якщо є лише один символ:
length = 1
code = 0

Це потрібно перевірити окремим тестом.

Наприклад, файл:
AAAAAAAAAAAAAAAAAAAA

є дуже хорошим тестом.

10.27. Шостий етап — NO_COMPRESSION

Не всі дані потрібно стискати.

Якщо стислий блок виходить не вигідним, потрібно зберегти його без стиснення.

Тоді:

`NO_COMPRESSION = 1`

i:

`coded_len = raw_len`

Це особливо важливо для:

- випадкового шуму;
- JPEG;
- ZIP;
- MP3;
- інших уже стиснутих даних.

Практичне правило вибору просте і надійне: закодуйте блок звичайним шляхом і порівняйте результат із сирими даними. Якщо закодований блок не коротший, запишіть блок у режимі `NO_COMPRESSION`. Так рішення спирається на фактичний результат, а не на прогноз.

10.28. Не використовуйте просте правило «висока ентропія = не стискати»

Це надто груба евристика.

Наприклад, деякі растрові дані можуть мати високу байтову ентропію, але сильні локальні повтори.

LZ77 усе одно може добре їх стиснути.

Тому правильніше реально оцінювати:

`compressed_size`

i:

`raw_size`

а не приймати рішення лише за одним показником ентропії.

10.29. Сьомий етап — оптимальний LZ77-розбір

Це вже просунутий рівень.

Greedy приймає рішення локально.

DP дивиться на весь хвіст.

Визначимо:

`c[i]`

як мінімальну вартість закодувати файл від позиції `i` до кінця.

Тоді:

```

C[i] =
min(
    cost_literal + C[i+1],
    cost_match(d,l) + C[i+1]
)

```

по всіх допустимих збігах.

10.30. Фіксовані вартості DP

У базовій постановці не потрібно будувати циклічну залежність:
Huffman depends on LZ77 parse

і водночас:
LZ77 parse depends on Huffman codes

Тому використовуються фіксовані вартості:
literal = 9 bits

```

match(d) =
17 + max(0, bit_length(d - 1) - 1)

```

Це робить DP однозначним.

10.31. Як тестувати DP-розбір

На малих рядках можна виконати повний перебір усіх допустимих розборів.

Тоді:

```

DP_cost == brute_force_cost

```

Після цього порівнюйте:

```

greedy
lazy
DP

```

на реальному корпусі.

Просунутий DP повинен давати вартість не гіршу за greedy за тією самою моделлю вартостей.

10.32. Восьмий етап — пакування в томи

Після кодування всі блоки вже мають відомі розміри.

Тепер їх потрібно розподілити між томами.

Це ще одна задача рюкзака.

Кожен блок має:

weight

i:

value

10.33. Вага блока

Для пакування потрібно врахувати не лише `coded_len`.

Кожен блок також потребує запису в таблиці блоків.

Тому:

`weight(block) = coded_len + 15`

байтів.

Ці 15 байтів — службовий запис блока.

10.34. Місткість першого тому

Не вся максимальна довжина тому доступна блокам.

Потрібно відняти:

- 41 байт зовнішнього заголовка;
- 32 байти НМАС;
- службові поля `payload`;
- таблицю файлів.

Тому для поточного набору кандидатів:

`W =`

`max_volume_bytes`

- 41
- 32
- 6
- `candidate_file_table_size`

Саме це `W` використовується в рюкзаку.

10.35. Важливий момент з таблицею файлів

Перед оптимізацією тому визначається множина всіх файлів, яким належать блоки-кандидати.

Їхня таблиця резервується до запуску рюкзака.

Не можна після вибору блоків раптово змінити таблицю і отримати іншу місткість.

Тоді оптимізаційна задача перестає бути тією, яку ви розв'язували.

10.36. DP-рюкзак для тому

Кожен блок можна:
ВЗЯТИ

або:
НЕ ВЗЯТИ

у поточний том.
Потрібно максимізувати сумарну цінність за обмеження:
 $\text{sum}(\text{weight}) \leq W$

Решта блоків переходить до наступного тому.

10.37. Greedy для пакування

Реалізуйте також простий baseline:
sort by value / weight

і послідовно додавайте блоки.
Порівнюйте:
DP
vs
greedy

для першого тому.
Базова вимога:
 $\text{value_DP} \geq \text{value_greedy}$

10.38. Якщо блок не поміщається

Якщо навіть один блок не може поміститися у порожній том, це не повинно породжувати напівзаписаний пошкоджений архів.

Пакувальник повинен контрольовано завершитися помилкою місткості.
Наприклад:
E_CAPACITY

Так само, якщо сама службова таблиця не вміщується в том.

10.39. Порожній архів

У Проекті 2 порожній вхід є допустимим.
Результат:
1 том

```
3:
file_count = 0
block_count = 0
```

Після розпакування отримується порожній набір файлів.
Це хороший крайній тест.

10.40. Дев'ятий етап — класифікатор типу файла

Тепер можна додавати перцептрон.
Для кожного файла обчислюється набір ознак.
Наприклад:

- ентропія байтів;
- гістограма;
- частка друкованих символів;
- характеристики рядків;
- автокореляція;
- ентропія біграм.

Мережа повертає один із семи класів.

10.41. Спочатку перевірте ознаки

До навчання MLP переконайтеся, що ознаки мають сенс.
Наприклад:

```
text
→ printable fraction high

random noise
→ entropy high

zeros
→ entropy low
```

Якщо ознаки обчислюються неправильно, нейромережа не виправить проблему.

10.42. Протокол навчання

Використовується:
train/
→ навчання

valid/

→ вибір моделі

holdout/

→ фінальна оцінка викладачем

Модель обирається за macro-F1 на validation.

Потрібно виконати щонайменше кілька незалежних ініціалізацій, а не залишати випадково найперший запуск.

10.43. Навіщо класифікатор потрібен архіватору

MLP не повинен бути декоративним компонентом.

Його клас повинен впливати на стратегію.

Наприклад:

already compressed

→ prefer NO_COMPRESSION

text

→ normal LZ77/Huffman

raster

→ інші параметри пошуку

Після цього можна виміряти:

fixed strategy

vs

classifier strategy

10.44. Десятий етап — криптографічний захист

Криптографія додається після того, як payload стабільно формується без шифрування.

Порядок:

compress

→ build payload

→ encrypt

→ HMAC

Не:

encrypt

→ compress

10.45. KDF

Використовується:
PBKDF2-HMAC-SHA256

з:
100000 iterations

і 16-байтовою salt.

Отримуються 64 байти:
first 32 → K_enc
last 32 → K_mac

10.46. PBKDF2 виконується один раз на архів

Усі томи одного архіву використовують одну salt.

Тому:
PBKDF2

не потрібно повторювати для кожного тому.

Правильна схема:
password + salt
→ derive keys once
→ use same K_enc/K_mac for all volumes

Це важливо і для швидкодії, і для коректного вимірювання метрики.

10.47. AES-256-CTR

Для кожного тому використовується окремий nonce.

У межах одного ключа:
nonce_i != nonce_j

для різних томів.

Зручна схема:
archive_nonce_prefix
||
volume_index

CTR counter всередині тому починається з 1.

10.48. Nonce не можна повторювати і між різними архівами

Якщо створюється новий архів, він повинен отримати нову випадкову salt і новий nonce-префікс.

Не можна постійно використовувати:

`salt = fixed`

`nonce = fixed`

у робочому архіваторі.

Фіксовані значення допустимі лише в контрольних тест-векторах.

10.49. HMAC

Кожен том має власний HMAC.

Перевірка виконується:

до decrypt

Правильний порядок:

`validate header`

→ `validate length`

→ HMAC

→ `decrypt`

→ `parse`

Якщо HMAC неправильний:

STOP

Payload не розбирається.

10.50. CRC32 і HMAC виконують різні ролі

HMAC відповідає за криптографічну цілісність і автентичність тому.

CRC32 використовується після склеювання всіх блоків, щоб перевірити правильність відновлення конкретного файлу.

Тобто:

HMAC

→ чи можна довіряти тому

CRC32

→ чи правильно відновлено файл

10.51. Багатотомний архів

Для всіх томів одного архіву повинні бути узгоджені:
volume_count
salt

Індекси повинні утворювати:
0, 1, ..., volume_count - 1

без:

- пропусків;
- дублікатів.

Nonce кожного тому має бути унікальним.

10.52. Порядок подання томів не має значення

Розархіватор може отримати:
part2
part0
part1

Він повинен сам упорядкувати їх за:
volume_index

і правильно відновити файли.

Не можна покладатися на порядок файлів у каталозі.

10.53. Відсутній том

Якщо:
volume_count = 3

а є лише:

0
2

архів неповний.

Розархіватор повинен відхилити його.

Так само при дублюванні:

0
1
1
2

Передбачений код відмови для цих випадків — E_VOLUMES. Він охоплює чотири ситуації:

- кількість наявних томів не дорівнює `volume_count`;
- якийсь `volume_index` трапляється двічі;
- `volume_index` $\geq$ `volume_count`;
- різні томи оголошують різне `volume_count`, тобто належать різним архівам.

Окремо перевіряється послідовність блоків кожного файла: значення `block_seq` мають утворювати суцільний ряд від нуля без пропусків і повторів, інакше E_PAYLOAD.

10.54. Одинадцятий етап — метрика Q

Для архіватора враховуються:

S

— ефективність стиснення,
i:

T

— швидкодія.

Підсумково:

$$Q = \alpha S + \beta T$$

де:

$$S = B_{\text{orig}} / (B_{\text{orig}} + B_{\text{arc}})$$

i:

$$T = t_{\text{ref}} / (t_{\text{ref}} + t)$$

10.55. Найважливіша умова метрики

Якщо хоча б один файл відновлено неправильно:

$$Q = 0$$

Немає сенсу отримувати надзвичайне стиснення, якщо архів пошкоджує дані.

Тому правильність завжди має вищий пріоритет за оптимізацію.

10.56. Спочатку baseline

Перед GA обов'язково виміряйте:

Q_baseline

на фіксованих параметрах.

Наприклад:

```
window = ...  
min_match = ...  
threshold = ...
```

Після цього вже можна оцінити, чи GA щось покращив.

10.57. Дванадцятий етап — генетичний алгоритм

На базовому рівні хромосома може містити три параметри:

```
window size  
minimum match length  
NO_COMPRESSION threshold
```

Кожна особина — одна конфігурація архіватора.

Її fitness визначається через реальний запуск на корпусі.

10.58. GA не повинен ламати правильність

Якщо певна хромосома приводить до того, що файл не відновлюється:
fitness

повинен бути найгіршим.

Не можна дозволяти GA «покращувати стиснення» шляхом втрати даних.

10.59. GA проти random search

Порівняння виконується за однакового числа fitness evaluations.

Наприклад:

```
GA = 200 evaluations  
Random = 200 evaluations
```

Порівнюється найкраще значення Q.

10.60. Елітизм

При максимізації:

```
best generation 10  
>=  
best generation 0
```

Це проста перевірка того, що найкраща особина не втрачається.

10.61. Тринадцятий етап — наскрізний тракт

Після інтеграції:

files

- features
- classification
- split blocks
- LZ77
- Huffman
- NO_COMPRESSION decision
- knapsack volumes
- payload
- AES
- HMAC
- volumes

Під час розпакування:

volumes

- validate
- HMAC
- decrypt
- parse
- decode Huffman
- decode LZ77
- join blocks
- CRC
- files

10.62. Покрокове тестування Проєкту 2

Рекомендований порядок:

1. file read/write
2. one raw block
3. one-volume raw archive
4. multiple raw files
5. block splitting
6. LZ77 decoder
7. greedy LZ77
8. lazy LZ77
9. Huffman
10. LZ77 + Huffman
11. NO_COMPRESSION

12. multi-volume without crypto
13. knapsack DP
14. greedy packing
15. DP vs greedy
16. classifier
17. AES
18. HMAC
19. CRC
20. multi-volume crypto
21. negative tests
22. Q
23. GA
24. full corpus

10.63. Обов'язкові маленькі тести

Створіть окремі тести для:

empty archive

empty file

one-byte file

all-zero file

one-symbol Huffman alphabet

file shorter than MIN_MATCH

file larger than volume

two-volume archive

volumes passed in reverse order

10.64. Обов'язкові negative tests

Потрібно перевірити:

bad magic

bad version

wrong ciphertext_len

bad HMAC

wrong password
bad CRC
invalid Huffman lengths
invalid DIST symbol
distance > 32768
truncated volume
missing volume
duplicate volume
unsafe filename

10.65. Контрольовані коди помилок

Некоректний архів не повинен завершувати програму випадковим Python exception.

Добре:

E_MAGIC
E_LENGTH
E_HMAC
E_PAYLOAD
E_CODE
E_CRC

Погано:

IndexError
struct.error
UnicodeDecodeError

Користувач повинен отримувати зрозумілу контрольовану відмову.

10.66. Еталонний архів

Особливо важливий тест — еталонний архів викладача.

Ваш розархіватор повинен правильно відкрити його навіть тоді, коли він був створений не вашим кодом.

Це перевіряє:

- endian;
- canonical Huffman;

- LZ77;
- bit order;
- KDF;
- CTR;
- HMAC;
- структуру payload.

Власний encoder/decoder можуть мати однакову помилку.

Еталонний архів таку помилку виявляє.

10.67. Що показати на Чекпоінті 1

До першого чекпоінту повинні працювати:
архів без складної оптимізації

Huffman

round-trip

еталонний архів

baseline Q

Головне питання:

Чи вже можна створити архів і побайтово правильно його розпакувати?

10.68. Що показати на Чекпоінті 2

До другого чекпоінту:
greedy LZ77

lazy LZ77

порівняння їхніх розмірів

DP-рюкзак

greedy packing

DP vs greedy

classifier macro-F1

Просунутий DP-розбір демонструється лише командами, які його заявили.

10.69. Що показати на Чекпоінті 3

До третього чекпоінту повинні бути інтегровані:

PBKDF2

AES-256-CTR

HMAC

CRC32

а також:

GA

GA vs random search

Q

Повний корпус повинен правильно архівуватися і відновлюватися.

10.70. Фінальні приймальні перевірки

Базовий проєкт повинен показати, що:

- один текстовий файл відновлюється точно;
- порожній і одnobайтовий файли підтримуються;
- всі сім груп корпусу відновлюються;
- дуже повторювані дані добре стискаються;
- випадковий шум переходить у NO_COMPRESSION, а не беззмістовно «стискається»;
- вже стиснуті файли можуть зберігатися без стиснення;
- canonical Huffman відновлюється правильно;
- один символ обробляється правильно;
- greedy і lazy LZ77 працюють;
- DP-рюкзак першого тому не гірший за greedy;
- змінений архів відхиляється;
- неправильний пароль відхиляється;
- багатотомний архів відновлюється незалежно від порядку томів;
- відсутній і дубльований томи відхиляються;
- еталонні архіви розпаковуються.

10.71. Які експерименти обов'язково провести

Для Проєкту 2 особливо корисні:

greedy LZ77

vs

lazy LZ77

DP packing

vs

greedy packing

compression

vs

NO_COMPRESSION

classifier strategy

vs

fixed strategy

GA

vs

random search

compress → encrypt

vs

encrypt → compress

10.72. Що вимірювати для різних типів файлів

Не потрібно дивитися лише на загальне Q.

Корисно окремо побачити поведінку на:

text

source code

structured data

raster

executable

already compressed

synthetic

Тоді можна зрозуміти, де алгоритм справді сильний, а де ні.

10.73. Не соромтеся режиму NO_COMPRESSION

Якщо для JPEG отримано:

raw = 100000 bytes

compressed = 100800 bytes

правильний висновок не:

«Наш compressor поганий».

Правильний висновок:

«Цей файл уже майже не містить надлишковості, тому його потрібно зберегти без повторного стиснення».

Уміння не запускати непотрібний алгоритм також є частиною розумної системи.

10.74. Типова помилка — оптимізувати до перевірки правильності

Не починайте GA, поки:
round-trip

не проходить стабільно.

Не починайте DP-розбір, поки:
greedy LZ77

не декодується правильно.

Не починайте multi-volume crypto, поки:
single-volume crypto

не працює.

Кожен наступний рівень повинен будуватися на стабільному попередньому.

10.75. Типова помилка — стиснути весь файл одним великим блоком

Це може:

- споживати багато пам'яті;
- ускладнювати пакування в томи;
- погано працювати на великих файлах.

Тому дані обробляються блоками.

Архіватор повинен бути готовим до ситуації, коли файл більший за доступну оперативну пам'ять.

10.76. Типова помилка — довіряти архіву

Все, що читається з архіву, потрібно вважати потенційно неправильним.

Не можна робити:

`buffer = bytearray(raw_len)`

без перевірки `raw_len`.

Не можна використовувати:

`file_index`

без перевірки меж.

Не можна декодувати Huffman-символ поза алфавітом.

Не можна записувати файл з небезпечним ім'ям.

Parser повинен бути строгим.

10.77. Що студент повинен уміти пояснити

На захисті потрібно розуміти:

Як LZ77 знаходить повтори?

Чому greedy LZ77 може бути неоптимальним?

Що змінює lazy greedy?

Що означає `C[i]` у DP-розборі?

Чому вартості DP зафіксовані?

Чому потрібні три Huffman-дерева?

Як працює canonical Huffman?

Чому DIST має діапазон `0...15`?

Чому вже стиснутий файл краще іноді не стискати?

Що є вагою блока у рюкзаку?

Чому таблиця файлів враховується до DP-пакування?

Чому PBKDF2 виконується один раз на архів?

Чому попсо різний для кожного тому?

Чому HMAC перевіряється до decrypt?

Чим CRC32 відрізняється від HMAC?

Що саме оптимізує GA?

10.78. Короткий маршрут Проєкту 2

Усю роботу можна запам'ятати так:

КРОК 1

Один файл без стиснення

↓

КРОК 2

Кілька файлів без стиснення



КРОК 3

Розбиття на блоки



КРОК 4

LZ77 greedy



КРОК 5

LZ77 lazy



КРОК 6

Huffman



КРОК 7

NO_COMPRESSION



КРОК 8

Багатотомність



КРОК 9

DP-рюкзак і greedy



КРОК 10
Класифікатор

↓

КРОК 11
AES + HMAC + CRC

↓

КРОК 12
Метрика Q

↓

КРОК 13
GA

↓

КРОК 14
Negative tests

↓

КРОК 15
Повний корпус

↓

КРОК 16
Clean run і захист

10.79. Головна порада

Не намагайтеся відразу написати власний аналог ZIP або 7-Zip.

Почніть із дуже простої задачі:

взяти один файл, записати його у правильний формат і отримати назад ті самі байти.

Коли це працює, додайте блоки.

Потім LZ77.

Потім Huffman.

Потім томи.

Потім оптимізацію.

Потім класифікатор.

Потім захист.

Потім GA.

Саме така послідовність дозволяє не втратити контроль над складною системою і зрозуміти роль кожного алгоритму.

11. Спеціальні методичні вказівки до Проєкту 3

«Розумна захищена система пріоритетної реплікації журналів подій»

Проєкт 3 присвячений задачі збереження журналів подій у розподіленій системі.

Уявімо, що є кілька серверів, edge-вузлів або автономних станцій. Вони постійно створюють події:

- звичайні повідомлення;
- попередження;
- аномалії;
- критичні події;
- переходи системи до аварійного стану.

Було б добре просто зберегти кожен подію у трьох копіях.

Але ресурс обмежений.

На вузлах не вистачає місця, операції запису іноді завершуються невдало, а в аварійному режимі значна частина спроб реплікації може бути втрачена.

Тому виникає головне питання:

Які події потрібно зберігати, скільки копій кожної події створити і на які вузли їх записати, щоб після відмов вдалося відновити максимум важливої інформації?

Саме ця задача робить Проєкт 3 алгоритмічно цікавим.

11.1. Постановка задачі

На вході система отримує потік записів журналу.

Кожний запис має фіксований розмір і містить інформацію про подію.

Система повинна:

отримати події

→ проаналізувати їх

→ визначити важливість

- стиснути
- вирішити кількість копій
- визначити порядок реплікації
- зашифрувати
- записати копії на вузли

Після моделювання відмов потрібно виконати зворотну задачу:
отримати доступні репліки

- перевірити НМАС
- розшифрувати
- декодувати
- усунути дублікати
- відновити журнал
- визначити, які критичні події збережено

Архітектурно система складається з агента журналу, планувальника реплікації та відновлювача. Модель вузлів і відмов студент не створює: вона видається готовим файлом `replication.py`.

11.2. Чому не можна просто зробити три копії всього

Якби ресурс був необмеженим, рішення було б простим:
кожна подія
→ 3 копії

Але кожна додаткова копія:

- займає місце;
- витрачає загальний бюджет реплікації;
- потребує часу;
- може сама бути втрачена під час запису.

Якщо витратити весь ресурс на дублювання перших подій, для пізніших подій бюджету може вже не залишитися.

Тому система повинна розподіляти ресурс, а не просто копіювати все.

11.3. Головна ідея Проекту 3

Для кожної події потрібно вирішити:

- 0 копій
- 1 копія
- 2 копії
- 3 копії

Це вже не звичайна задача рюкзака 0/1.

У нас для одного об'єкта є кілька допустимих варіантів.

Тому використовується багатоваріантний рюкзак.

При цьому друга і третя копії мають давати додаткову надійність, але їхня користь не повинна зростати лінійно.

Інакше алгоритм міг би витратити майже весь бюджет на кілька дуже важливих подій.

11.4. Загальна архітектура

Систему можна уявити так:

потік журналу

↓

класифікація

↓

визначення цінності

↓

Huffman

↓

вибір 0/1/2/3 копій

↓

планування вузлів

↓

AES-256-CTR

↓

HMAC

↓

replication.py

↓

відмови

↓

відновлення

Окремо генетичний алгоритм налаштовує параметри системи.

11.5. Що отримує команда

Для виконання проєкту команда отримує:

FORMAT_SPEC_P3.md

REPLICATION_SPEC.md

team_config.json

replication.py

train.csv

valid.csv
session.csv
session.bin
reference/

У reference/ містяться:

- еталонні правильні репліки;
- їхні описи;
- спеціально пошкоджені репліки;
- очікувані коди відмов.

Команди не видаються:

holdout.csv
official_eval.json
gen_project3.py

Фінальна якість класифікатора перевіряється на прихованому holdout.csv, а офіційне Q — на прихованих seed моделі реплікації.

11.6. Модель відмов уже готова

Не потрібно реалізовувати випадкові відмови вузлів самостійно.

Використовується:

replication.py

У системі є три вузли:

A
B
C

Для кожного задаються:

- місткість;
- імовірність невдалого запису;
- середня довжина пачки відмов;
- оцінка надійності.

Є також глобальний бюджет усіх операцій реплікації.

11.7. Чому replication.py не можна змінювати

Якщо кожна команда самостійно реалізує модель відмов, результати перестануть бути порівнянними.

Одна команда може зробити модель «м'якшою», інша — складнішою.

Тому модель однакова для всіх.

Студент розробляє:

алгоритм прийняття рішення в заданому середовищі, а не саме середовище.

11.8. Перший етап — навчитися читати один запис

Запис журналу в базовому форматі займає рівно:
16 bytes

Він містить:

```
timestamp_ms  
source_id  
event_code  
metric_value  
category  
severity  
critical  
record_flags
```

Точні типи та допустимі діапазони задає `FORMAT_SPEC_P3.md`.

11.9. Перший тест

Почніть з одного запису:

```
record  
→ serialize  
→ parse  
→ restored
```

Після цього:

```
record == restored
```

Не переходьте до реплікації, поки базове представлення одного запису не працює.

11.10. Перевірка полів

Parser повинен перевіряти допустимі значення.

Наприклад:

```
source_id
```

має бути в допустимому діапазоні.

Так само:

category
severity
critical
record_flags

Не потрібно приймати явно некоректний запис і сподіватися, що подальші алгоритми «якось розберуться».

11.11. Другий етап — найпростіша репліка без стиснення

Тепер створіть одну репліку з одного або кількох записів.

На цьому етапі можна використовувати:

NO_COMPRESSION

Тобто:

records
→ payload
→ replica
→ payload
→ records

Поки без MLP, DP і GA.

11.12. Формат LOG3

Базова репліка має вигляд:

HEADER
|
CIPHERTEXT
|
HMAC

Заголовок займає:

43 bytes

і містить:

- LOG3;
- версію;
- replica_group_id;
- batch_seq;
- salt;

- nonce;
- ciphertext_len.

Формат є обов'язковим для всіх команд.

11.13. Спочатку реалізуйте payload без криптографії

Зручно спочатку зробити внутрішнє навантаження окремо:

records

→ encode_payload

→ decode_payload

→ records

І лише після цього:

payload

→ AES

→ HMAC

Це значно спрощує налагодження.

11.14. Основний інваріант payload

Для базового формату один запис завжди складається з 16 байтів.

Тому:

`symbol_count = 16 × record_count`

Якщо це не виконується, payload некоректний.

Parser повинен його відхилити.

11.15. Порожній журнал

Порожній журнал є важливим крайнім випадком.

Система не повинна:

crash

лише тому, що:

`record_count = 0`

Порожня репліка повинна або коректно підтримуватися відповідно до формату, або оброблятися контрольованою логікою без необроблених винятків.

11.16. Третій етап — Huffman

Коли сире кодування працює, додайте канонічний Huffman.

Послідовність:

- 16-byte records
 - byte stream
 - frequencies
 - Huffman lengths
 - canonical codes
 - bitstream

На приймачі:

- lengths
 - canonical codes
 - bitstream decode
 - bytes
 - records

11.17. Канонічний Huffman

У репліку передаються:

- symbol
- +
- code length

Саме дерево не передається.

Decoder за довжинами відновлює ті самі коди.

Максимальна довжина коду:

15

Якщо отримано один символ:

length = 1

code = 0

Ці правила визначені специфікацією.

11.18. Коли вмикати NO_COMPRESSION

Як і в Проєкті 2, стиснення не завжди вигідне.

Якщо службова таблиця та Huffman-потік дають більший результат, ніж сирі записи, можна використати:

NO_COMPRESSION

Тоді головне правило:

decode(raw)

==

original

11.19. Четвертий етап — класифікація подій

Тепер можна перейти до MLP.

Базове завдання — класифікувати кожну подію в один із трьох класів:

- 0 — штатна
- 1 — аномальна
- 2 — перехід до інциденту

Клас впливає на цінність події для задачі реплікації.

11.20. Важливість часової історії

У цьому проєкті недостатньо дивитися лише на один запис.

Наприклад, перехід до інциденту може проявлятися не в одному великому значенні, а в послідовності змін:

події частішають

category частіше змінюється

змінюються джерела

severity поступово зростає

Тому ознаки обов'язково повинні містити інформацію про попереднє вікно подій.

11.21. Мінімальні часові ознаки

Точний набір задається в `team_config.json`.

Типові ознаки включають:

`delta_metric`
`events_per_window`
`category_changes`
`source_changes`
`severity_mean`

Не потрібно механічно додавати сотні ознак.

Спочатку перевірте, чи правильно рахуються базові.

11.22. Поле `critical` не є правильною відповіддю для `classifier`

Дуже важливий момент:

`critical`

означає важливість події для зберігання.

Це не клас.

Не можна робити:

```
critical = 1
```

→ anomaly

як готове правило.

Клас і critical формуються різними механізмами.

11.23. Протокол навчання

Правильний процес:

```
train.csv
```

→ навчання

```
valid.csv
```

→ вибір моделі

```
holdout.csv
```

→ фінальна оцінка викладачем

Навчання повторюється щонайменше з трьох ініціалізацій.

Обирається модель із кращим macro-F1 на validation.

11.24. Не підбирайте модель під holdout

holdout.csv студент не отримує.

Це зроблено для того, щоб фінальний macro-F1 залишався незалежною оцінкою.

Налаштовуйте:

- кількість нейронів;
 - learning rate;
 - кількість епох;
 - регуляризацію;
- за train і valid.

11.25. Як classifier входить у реплікацію

Результат MLP повинен впливати на:

```
value_i
```

Тобто важливіші класи отримують більшу базову цінність.

Додатково:

`critical = 1`

додає бонус.

Базова формула:

```
v_i =  
priority(class_i)  
+  
critical_bonus × [critical_i = 1]
```

Точні значення задає `team_config.json`.

11.26. П'ятий етап — обчислення ваги події

Перед DP потрібно знати вартість зберігання.

Для одного вікна записів спочатку будується одна таблиця Huffman.

Після цього для події i :

```
w_i =  
sum(code_length(byte))
```

по всіх 16 байтах запису.

Тобто w_i — кількість бітів, необхідних для однієї стиснутої копії цієї події.

Таблиця будується до розв'язання рюкзака і після цього не змінюється. Це принципово. Якби таблицю перебудовували за фактично обраними подіями, виникла б циклічна залежність: розміри залежали б від вибору, а вибір — від розмірів, і задача стала б невизначеною.

Звідси випливає наслідок, який легко пропустити: таблиця будується за частотами байтів усіх записів вікна, включно з тими, для яких потім буде обрано $r_i = 0$. Саме ця таблиця записується в репліку, навіть якщо частина подій до неї не потрапила. Інакше фактичний розмір репліки не збігатиметься з тим, який врахувало ДП.

11.27. Не змішуйте вагу і цінність

Це різні поняття.

`weight`

відповідає на питання:

Скільки ресурсу потрібно?

А:

`value`

відповідає:

Наскільки важливо зберегти цю подію?

Саме співвідношення цих двох величин визначає складність оптимізації.

11.28. Шостий етап — багатоваріантний рюкзак

Для кожної події:

$$r_i \in \{0, 1, 2, 3\}$$

Потрібно максимізувати:

$$\sum u_i(r_i)$$

за умови:

$$\sum \text{cost}_i(r_i) \leq W$$

де W — бюджет реплікації.

11.29. Вартість кількох копій

Для r копій:

$$\text{cost}_i(r) = r \times w_i$$

Тобто три копії займають приблизно втричі більше ресурсу, ніж одна.

11.30. Корисність кількох копій

Корисність задається кумулятивно:

$$u_i(0) = 0$$

$$u_i(1) = v_i$$

$$u_i(2) = v_i \times \text{second_copy_factor}$$

$$u_i(3) = v_i \times \text{third_copy_factor}$$

Наприклад, якщо фактори:

$$1.55$$

$$1.85$$

то друга копія не подвоює користь, а третя не потроює її.

Це і є зменшувана віддача.

11.31. Чому factors кумулятивні

Якщо:

$$u(2) = 1.55v$$

це загальна корисність двох копій, а не приріст другої копії.

Тому приріст другої копії:

$$\Delta u_2 = 1.55v - 1.0v = 0.55v$$

А приріст третьої:

$$\Delta u_3 = 1.85v - 1.55v = 0.30v$$

Саме так видно зменшувану віддачу.

11.32. Стан DP

Один зі зручних варіантів:

$DP[i][w]$

— максимальна корисність для перших i подій при бюджеті w .

Для кожної події перебираються чотири варіанти:

$$r = 0$$

$$r = 1$$

$$r = 2$$

$$r = 3$$

i вибирається кращий допустимий.

11.33. Складність DP

Якщо:

- n — кількість подій;
- W — дискретизований бюджет;
- $R_{\max} = 3$,

складність приблизно:

$$O(n \times W \times R_{\max})$$

Оскільки $R_{\max}$ малий і фіксований, основний вплив мають n і W .

11.34. Як тестувати DP

Для малого числа подій можна зробити повний перебір.

Для кожної події є чотири варіанти.

Наприклад для:

$$n = 6$$

кількість комбінацій:

$$4^6 = 4096$$

це легко перебрати.

Потім:

DP_utility == brute_force_utility

11.35. Перевіряйте не тільки utility

Після DP потрібно перевірити:

$\sum \text{cost}_i(r_i) \leq W$

i:

$r_i \in \{0, 1, 2, 3\}$

для кожної події.

Також відновлена конфігурація повинна давати саме ту utility, яку записано в таблиці DP.

11.36. Сьомий етап — greedy baseline

Після точного DP реалізується жадібний алгоритм.

Він дивиться не на загальну utility, а на наступну можливу копію.

Для неї рахується:

$\Delta u / \text{cost}$

Наприклад:

- перехід $0 \rightarrow 1$;
- $1 \rightarrow 2$;
- $2 \rightarrow 3$.

На кожному кроці вибирається найвигідніший приріст.

11.37. Чому greedy не гарантує оптимум

Greedy робить локально найкращий вибір.

Але набір локально хороших копій може зайняти бюджет так, що пізніше не поміститься більш вигідна комбінація.

Тому потрібно знайти або побудувати контрприклад.

Це важлива частина захисту.

11.38. Головний алгоритмічний експеримент

Порівняйте:

DP

vs

greedy

На однакових вікнах.

Виміряйте:

- utility;
- використані біти;
- час роботи.

Базова вимога:

`utility_DP >= utility_greedy`

11.39. Вікно реплікації

Рішення приймається не обов'язково одразу для всього журналу.

У конфігурації задається:

`replication_window_records`

У базовому комплекті використовується вікно з 48 записів.

Принцип роботи:

взяти вікно

→ класифікувати

→ стиснути

→ вирішити кількість копій

→ передати планувальнику

→ перейти до наступного вікна

11.40. Восьмий етап — вибір вузлів

DP відповідає на питання:

Скільки копій створити?

Але залишається інше питання:

Куди саме їх записати?

Це вже задача планувальника.

Чому $R_{\max} = 3$

Вузлів рівно три, а кілька копій мають сенс лише на різних вузлах. Тому більше ніж три копії однієї події не дають додаткової незалежності, і верхня межа $r_i = 3$ є не довільним обмеженням, а прямим наслідком кількості вузлів.

ДП не знає про місткість окремих вузлів

Це найважливіше застереження цього етапу. ДП оптимізує за одним глобальним бюджетом W . Місткості окремих вузлів A, B, C у нього не входять.

Тому розв'язок ДП є верхньою оцінкою: він каже, скільки копій варто зробити за наявного ресурсу, але не гарантує, що планувальник фізично

розмістити їх усі. Може виявитися, що для події призначено три копії, а вільне місце лишилося тільки на одному вузлі.

Система має передбачати, що частину призначених копій розмістити не вдасться. Планувальник у такому разі:

- розміщує копії за спаданням цінності, щоб недоотримали ресурс найменш важливі події;
- фіксує, скільки копій із призначених реально записано.

У звіті варто навести розбіжність між призначеним і розміщеним: це показує, наскільки обмеження вузлів впливає на результат, і є хорошим матеріалом для захисту.

11.41. Вузли мають різну надійність

Вузли:

A
B
C

можуть мати:

- різну місткість;
- різну ймовірність відмови;
- різну довжину пачок відмов.

Тому не завжди правильно просто послідовно писати:

A
A
A

11.42. Копії на різних вузлах

Дві копії на одному ненадійному вузлі не еквівалентні двом копіям на незалежних вузлах.

Якщо вузол недоступний, обидві копії можуть бути втрачені одночасно.

Тому під час планування важливо використовувати незалежність вузлів.

На цьому й побудована модель реплікації.

11.43. Жадібний планувальник вузлів

Базова евристика може враховувати:
приріст очікуваної відновлюваності

/

вартість запису

з поправкою на:

- надійність вузла;
- його залишкову місткість.

Це евристика.

Не потрібно називати її оптимальним алгоритмом.

11.44. Порядок операцій має значення

У моделі `store()` навіть невдала спроба може витратити ресурс.

Якщо операція поміщається в бюджет, байти вважаються витраченими незалежно від того, чи сам запис завершився успішно.

Тому:

які копії йдуть першими

має реальний вплив.

Критичні дані, залишені на кінець, можуть не отримати ресурсу.

11.45. Дев'ятий етап — штатний сценарій

Перший повний тест робіть у штатному сценарії.

Тут:

- вузли стабільні;
- місткість велика;
- бюджет достатній.

Очікується, що критичні події повинні бути відновлені повністю.

Це найпростіший інтеграційний сценарій.

11.46. Деградований сценарій

Після штатного переходьте до деградованого.

Тут:

- частина операцій запису не проходить;
- виникають короткі пачки відмов;
- бюджет уже обмежений.

Система повинна показати, що пріоритетна реплікація дає користь.

11.47. Аварійний сценарій

Найскладніший сценарій:

- висока частота відмов;
- довші пачки BAD;

- жорсткий глобальний бюджет.
Тут неможливо ідеально зберегти все.
Саме в цьому режимі особливо видно якість DP і планувальника.
Три сценарії та їхня логіка визначені REPLICATION_SPEC.md.

11.48. Не оптимізуйте під відкритий seed

Під час розробки є відкритий seed.

Він потрібен, щоб помилки можна було відтворити.

Але не можна будувати логіку типу:

«Я вже знаю, що десята операція на вузлі В буде невдалою».

Офіційна оцінка виконується на п'яти інших прихованих seed.

Алгоритм повинен використовувати статистичну інформацію, а не знання майбутнього.

11.49. Десятий етап — криптографічний захист

Після стабільного кодування репліки додається захист.

Порядок:

compress

→ encrypt

→ HMAC

У базовому форматі жорстко задані:

PBKDF2-HMAC-SHA256

AES-256-CTR

HMAC-SHA256

з окремими ключами.

11.50. Виведення ключів

З пароля і salt:

PBKDF2

→ 64 bytes

Потім:

K_enc = first 32

K_mac = last 32

Не використовуйте один ключ для двох задач.

11.51. PBKDF2 один раз на сеанс

Для однієї salt ключі виводяться один раз.

Не потрібно виконувати 100000 ітерацій для кожної маленької репліки окремо.

Це лише погіршить час і спотворить компонент T.

11.52. AES-256-CTR

Counter формується:

```
nonce(12)
||
uint32_BE(i)
```

де:

i starts at 1

Nonce не повинен повторюватися для одного K_enc.

11.53. HMAC перед decrypt

Правильний порядок приймання:

```
minimum length
→ magic/version
→ exact length
→ HMAC
→ decrypt
→ strict payload parse
```

Якщо HMAC неправильний:

STOP

Розшифровування не виконується.

11.54. Обов'язкові negative tests

Потрібно спеціально перевірити:

bad magic

bad version

truncated replica

bad HMAC

modified ciphertext
wrong password
reserved != 0
invalid Huffman table

11.55. Контрольовані коди помилок

Формат передбачає коди:

E_SHORT
E_MAGIC
E_LENGTH
E_HMAC
E_PAYLOAD
E_CODE

Повний перелік із точними умовами наведено у `FORMAT_SPEC_P3.md`, а очікуваний код для кожної виданої пошкодженої репліки — у файлі `expected_failures.json` з каталогу `reference/`.

Пошкоджена репліка не повинна завершуватися випадковим:
`IndexError`

або:
`struct.error`

11.56. Відновлення після відмов

Після завершення сценарію потрібно отримати множину записів, які вижили хоча б на одному вузлі.

Саме це є фінальним результатом реплікації.

Після цього:

`surviving records`
→ `deduplicate`
→ `sort/reconstruct`
→ `restored log`

11.57. Дублікати — це нормально

Якщо одна подія має три копії і всі три пережили сценарій, у відновленому журналі вона повинна з'явитися один раз.

Тому потрібен стабільний ключ ідентичності запису, і він має бути визначений заздалегідь, а не обраний на ходу.

Це не дрібниця: від ключа прямо залежить R. Якщо два різні записи випадково вважати одним, частина подій зникне з відновленого журналу і R буде занижене. Якщо навпаки, одну подію порахувати двічі, R буде завищене.

Ключ задає `FORMAT_SPEC_P3.md`. Порівнювати всі 16 байтів запису як ознаку ідентичності недостатньо надійно: два різні події теоретично можуть збігтися побайтово, і тоді одна з них зникне.

11.58. Одинадцятий етап — метрика R

Одна з найважливіших метрик:

R

— частка критичних подій, які вдалося відновити.

Якщо було:

100 critical

а відновлено:

93

то:

R = 0.93

11.59. Метрика S

S показує ефективність використання ресурсу:

S =

B_ref

/

(B_ref + B_stored)

Чим більше бездумних копій створює система, тим більший B_stored.

Тобто надійність не є безкоштовною.

11.60. Метрика T

Час також враховується:

T =

t_ref

/

(t_ref + t)

Тому надзвичайно точний, але дуже повільний алгоритм може програвати більш збалансованому.

11.61. Загальна метрика

Фінальна метрика:

$$Q = \alpha S + \beta R + \gamma T$$

де:

$$\alpha + \beta + \gamma = 1$$

Точні значення задає `team_config.json` і не змінює протягом семестру.

Якщо система аварійно завершується або не відновлює жодної критичної події:

$$Q = 0$$

11.62. Чому не можна оптимізувати лише R

Можна підвищити R, якщо:

усе копіювати 3 рази

Але тоді дуже швидко закінчиться бюджет.

У результаті пізніші події взагалі не потраплять у сховище.

Тому система повинна шукати баланс між:

надійністю

ресурсом

часом

11.63. Дванадцятий етап — генетичний алгоритм

Після того як Q стабільно обчислюється, можна додавати GA.

На базовому рівні хромосома має кілька параметрів.

Наприклад:

```
replication_window_size  
NO_COMPRESSION threshold  
second_copy_factor
```

11.64. Не запускайте GA до готової системи

GA не виправляє помилки.

Якщо Q рахується неправильно, GA дуже ефективно знайде параметри, які оптимізують неправильну метрику.

Тому порядок:

```
full system
```

```
→ Q
```

- tests
- GA

11.65. Fitness

На базовому рівні:
`fitness = Q`

або точна функція, визначена умовою.

Якщо конфігурація приводить до некоректного відновлення:
`fitness`

повинен бути найгіршим.

11.66. GA проти random search

Порівняння повинно мати однаковий обчислювальний бюджет.

Наприклад:

`GA = 200 evaluations`

`Random Search = 200 evaluations`

Не порівнюйте алгоритми, якщо одному дали в десять разів більше спроб.

11.67. Елітизм

За елітизму:

`best_final >= best_initial`

якщо `fitness` максимізується.

Це хороший базовий sanity test.

Водночас GA не зобов'язаний перемогти random search на кожному запуску.

11.68. Тринадцятий етап — наскрізний тракт

Фінальний шлях:

`session`

→ `features`

→ `MLP`

→ `value`

→ `Huffman`

→ `DP copy counts`

→ `greedy node placement`

→ `AES`

- HMAC
- replication.py
- failures
- surviving replicas
- verify
- decrypt
- decode
- deduplicate
- evaluate Q

Саме цей запуск потрібно вміти виконати однією зрозумілою командою.

11.69. Як тестувати повну систему

Запускайте сценарії в такому порядку:
штатний

потім:
деградований

потім:
аварійний

Не починайте налагодження з найскладнішого сценарію.

11.70. Покроковий порядок тестів

Рекомендовано:

1. record serialize/parse
2. one-record raw replica
3. empty log
4. NO_COMPRESSION
5. Huffman
6. canonical Huffman
7. replica round-trip
8. MLP forward
9. classifier training
10. value calculation
11. DP vs brute force
12. greedy vs DP
13. node scheduler
14. штатний scenario
15. деградований scenario

- 16. аварійний scenario
- 17. AES known vector
- 18. HMAC tamper
- 19. wrong password
- 20. full recovery
- 21. Q
- 22. GA
- 23. GA vs random

11.71. Обов'язкові маленькі приклади

Окремо перевірте:

- 1 record
- 0 records
- 1 critical record
- 2 records with same weight
- budget exactly equals cost
- budget one unit smaller
- all events critical
- no events critical

11.72. Boundary tests для DP

Особливо корисно:

- $W = 0$
- $W = \text{min_cost} - 1$
- $W = \text{min_cost}$
- W дуже великий

У великому бюджеті алгоритм може дозволити більше копій.
У нульовому — жодної.

11.73. Перевірка зменшеної віддачі

Для однієї події переконайтеся:

$$u(1) > u(0)$$

$u(2) > u(1)$

$u(3) > u(2)$

але:

$u(2) - u(1)$

$>$

$u(3) - u(2)$

для типових параметрів.

Так студент краще бачить сенс другої і третьої копії.

11.74. Що показати на Чекпоінті 1

До першого чекпоінту повинні працювати:

record format

replica format

NO_COMPRESSION

Huffman

round-trip

reference replicas

Головне питання:

Чи можемо ми записати журнал у базову репліку і точно відновити його?

11.75. Що показати на Чекпоінті 2

До другого чекпоінту:

classifier

macro-F1 на valid

value calculation

DP 0/1/2/3 copies

greedy baseline

DP vs greedy

replication.py

і перші результати для трьох сценаріїв.

11.76. Що показати на Чекпоінті 3

До третього чекпоінту повинні бути інтегровані:

PBKDF2

AES-256-CTR

HMAC

Також:

Q

GA

GA vs random

full recovery after failures

11.77. Базові фінальні приймальні перевірки

Проект повинен показати, що:

- один запис відновлюється побайтово;
- порожній журнал не викликає аварії;
- NO_COMPRESSION працює;
- canonical Huffman відновлюється з довжин;
- один символ обробляється правильно;
- DP не гірший за greedy за utility;
- у штатному сценарії критичні події відновлюються;
- у деградованому виконується заданий поріг;
- classifier проходить F1_min;
- змінений ciphertext або HMAC відхиляється;
- неправильний пароль і пошкоджений формат обробляються контрольовано;
- еталонні пакети читаються.

11.78. Вплив кількості копій

Обов'язково дослідіть окремо:

max 1 copy

max 2 copies

max 3 copies

S
R
Q

Для кожного режиму виміряйте:

Це покаже реальну ціну надійності.

11.79. Не робіть висновок «3 копії завжди краще»

Три копії можуть дати більший R.

Але вони також:

- використовують більше ресурсу;
- можуть витіснити інші події;
- збільшують час.

Тому правильне питання:

У яких умовах третя копія ще виправдана?

11.80. Вплив classifier

Проведіть ablation:
без MLP

проти:
з MLP

Наприклад, без classifier можна використовувати фіксовані цінності.

Порівняйте Q.

Якщо MLP майже нічого не покращив, це теж корисний результат.

11.81. Вплив DP

Так само:
greedy only

проти:
DP

За utility і за фінальним Q.

Можлива ситуація:

DP знаходить трохи кращий набір, але витрачає значно більше часу.

Саме такі trade-off і потрібно аналізувати.

11.82. Вплив GA

Порівняйте:
manual parameters

GA parameters

random search parameters

за однакових даних.

11.83. Обов'язковий експеримент із криптографією

Як і в інших проєктах, порівняйте:
compress
→ encrypt

i:
encrypt
→ compress

Зашифрований потік повинен стискатися значно гірше.

11.84. Модуль атаки

У базовому завданні GA може використовуватися для відновлення ключа спеціального слабкого навчального шифру.

AES атакувати не потрібно.

Також проводиться перевірка цілісності:
modify replica byte
→ recovery system

Очікування:
reject

11.85. Просунутий аналіз метаданих

На просунутому рівні можна спробувати визначити характер активності лише за:

- розміром реплік;
- частотою їх створення;
- часовими патернами.

Payload при цьому не розшифровується.

Після цього можна додати padding і оцінити:

attack quality before
vs
attack quality after

i вартість захисту в байтах.

11.86. Просунуті контрольні точки журналу

На просунутому рівні можна оптимізувати розміщення повних snapshot.

Ідея:

snapshot
→ delta records
→ snapshot
→ delta records

Занадто багато snapshot дорого.

Занадто мало — збільшує вартість відновлення.

Тому використовується DP:

```
F[j] =  
min_i(  
    F[i]  
    + Snapshot(i, j)  
    + RecoveryCost(i, j)  
)
```

Це окрема розширена алгоритмічна задача.

11.87. Крос-сумісність

На просунутому рівні базову репліку однієї команди повинен уміти читати відновлювач іншої.

Але це перевіряється тільки для базового формату.

Командні розширення типу:

PREDICTION
MULTI_COPY_META

не зобов'язані бути сумісними між командами.

11.88. Типова помилка — змішати дві задачі

У Проекті 3 є дві різні оптимізації:
скільки копій?

i:

на які вузли?

Перша вирішується DP.

Друга — планувальником.

Не потрібно змішувати їх в одну незрозумілу функцію на сотні рядків.

11.89. Типова помилка — вважати другу копію другим незалежним предметом

Для однієї події не можна незалежно вибрати:

1 core option

i:

2 core option

одночасно.

Це взаємовиключні варіанти.

Для кожної події обирається одне значення:

r_i

із 0, 1, 2, 3.

11.90. Типова помилка — рахувати лише успішні записи в бюджеті

У моделі спроба запису, яка вже пройшла перевірку бюджету, витрачає ресурс навіть тоді, коли вузол перебуває у BAD і фактична копія втрачена.

Тому не можна після невдалого запису просто повернути витрачений бюджет.

11.91. Типова помилка — робити три копії на одному вузлі

Кілька копій мають сенс лише тоді, коли вони підвищують незалежність збереження.

Три дублікати на одному вузлі мало допомагають проти відмови самого вузла.

Тому планувальник повинен враховувати розподіл копій.

11.92. Типова помилка — використовувати severity як готовий клас

Високий severity може бути корисною ознакою.

Але клас залежить також від часової структури.

Особливо:

```
class 2
```

— перехід до інциденту.

Тому модель, яка просто копіює severity, не виконує справжню задачу.

11.93. Типова помилка — оптимізувати на одному seed

Якщо один алгоритм випадково добре працює на одному сценарії відмов, це ще не означає, що він кращий.

Офіційна оцінка усереднюється за п'ятьма прихованими seed.

Тому власні експерименти також корисно повторювати на кількох seed.

11.94. Що потрібно зберігати в результатах

Рекомендовано:

```
results/  
├── dp_vs_greedy.csv  
├── copy_count_experiment.csv  
├── scenarios.csv  
├── classifier_metrics.json  
├── ga_history.csv  
├── ablation.csv  
└── final_Q.json
```

Так результати легко відтворити.

11.95. Що студент повинен уміти пояснити

На захисті потрібно розуміти:

Чому реплікація взагалі є оптимізаційною задачею?

Чим багатоваріантний рюкзак відрізняється від 0/1?

Що означає $DP[i][w]$?

Чому utility другої і третьої копії має зменшувану віддачу?

Чому factors є кумулятивними?

Як працює greedy за $\Delta u / cost$?

Наведіть приклад, де greedy програє DP.

Чому копії на різних вузлах кращі за копії на одному?

Чому порядок `store()` має значення?

Чому `critical` не є класом?

Навіщо MLP потрібне в задачі реплікації?

Чому HMAC перевіряється до decrypt?

Чому поперше не можна повторювати?

Що саме оптимізує GA?

Що відбудеться, якщо бюджет реплікації зменшити на 30 %?

11.96. Короткий маршрут Проєкту 3

Усю роботу можна запам'ятати так:

КРОК 1

Навчитися читати один запис

↓

КРОК 2

Створити просту репліку без стиснення

↓

КРОК 3

Додати Huffman

↓

КРОК 4

Зробити classifier

↓

КРОК 5

Розрахувати value i weight

↓

КРОК 6

Реалізувати DP 0/1/2/3 copies

↓

КРОК 7

Реалізувати greedy baseline

↓

КРОК 8

Порівняти DP і greedy

↓

КРОК 9

Реалізувати планувальник вузлів

↓

КРОК 10

Підключити replication.py

↓

КРОК 11

Перевірити три сценарії

↓

КРОК 12

Додати AES + HMAC

↓

КРОК 13

Реалізувати відновлення і дедуплікацію

↓

КРОК 14

Навчитися рахувати Q

↓

КРОК 15

Додати GA

↓

КРОК 16

Провести експерименти

↓

КРОК 17

Negative tests

↓

КРОК 18

Clean run і захист

11.97. Головна порада

Не починайте Проект 3 з думки:

«Нам потрібно одразу побудувати розумну відмовостійку розподілену систему».

Почніть із значно простішої задачі:

Створити один правильний 16-байтовий запис, записати його в одну репліку і точно отримати назад.

Потім навчіться стискати.

Потім визначати важливість.

Потім вибирати кількість копій.

Потім розподіляти їх між вузлами.

Потім моделювати відмови.

Потім захищати.

І тільки після цього — оптимізувати.

Саме така послідовність перетворює складну задачу реплікації на набір невеликих і зрозумілих алгоритмічних кроків.

12. Додатки

Цей розділ містить готові шаблони, які можна використовувати під час виконання проєкту.

Їх не потрібно копіювати буквально.

Команда може змінювати структуру під свій проєкт.

Головне — зберегти логіку:

що робимо

- як запускаємо
- як тестуємо
- що отримали
- як це відтворити

Додаток А. Шаблон README.md

README — це перший файл, який відкриває викладач або інший студент.

Він повинен дозволити людині, яка ніколи раніше не бачила проєкт, зрозуміти:

- що це за система;
- як її встановити;
- як запустити;
- як запустити тести;
- як повторити основні експерименти.

Рекомендований шаблон:

Назва проєкту

Короткий опис системи у 2–4 реченнях.

Команда

- Студент 1 — зона відповідальності
- Студент 2 — зона відповідальності
- Студент 3 — зона відповідальності
- Студент 4 — зона відповідальності
- Студент 5 — зона відповідальності

Що робить система

Коротко описати:

- вхідні дані;

- основну обробку;
- вихідний результат.

Архітектура

input

- component 1
- component 2
- component 3
- output

Коротко пояснити роль кожного компонента.

Основні алгоритми

- алгоритм 1 — для чого використовується;
- алгоритм 2 — для чого використовується;
- алгоритм 3 — для чого використовується.

Вимоги

- Python ...
- операційна система ...
- необхідні бібліотеки ...

Встановлення

```
```bash
git clone ...
cd ...
python -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```
```

або актуальні команди вашого проєкту.

Запуск програми

```
```bash
python ...
```

```

Приклад:

```
```bash
python main.py --input data/...
```
```

Запуск тестів

```
```bash
pytest
```
```

або інша команда.

Запуск окремого тесту

```
```bash
pytest tests/test_dp.py
```
```

Запуск експериментів

```
```bash
python experiments/run_all.py
```
```

або окремо:

```
```bash
python experiments/run_dp_vs_greedy.py
python experiments/run_ga.py
```
```

Вхідні дані

Описати:

- де вони знаходяться;
- який формат;

- які файли потрібні для запуску.

Результати

Основні результати зберігаються в:

```
```text  
results/
```
```

Наприклад:

```
```text  
results/dp_vs_greedy.csv
results/ga_history.csv
results/final_metrics.json
```
```

Відтворення фінального результату

1. checkout final tag;
2. встановити залежності;
3. запустити тести;
4. запустити основний experiment;
5. отримати фінальні метрики.

Наприклад:

```
```bash  
git checkout final-v1.0
pytest
python experiments/run_final.py
```
```

Відомі обмеження

- ...
- ...
- ...

Використання ШІ

Див. файл:

```
```text
META-USAGE.md
```
```

Додаток Б. Короткий шаблон META-USAGE.md

Файл META-USAGE .md потрібен для прозорого опису використання систем штучного інтелекту під час роботи.

Не потрібно записувати кожне коротке питання.

Важливо зафіксувати суттєве використання.

Шаблон:

META-USAGE

Використані інструменти

Наприклад:

- ChatGPT
- GitHub Copilot
- інший інструмент

Для чого використовувалися

Приклад 1

Задача:

пояснення алгоритму канонічного Гаффмана.

Що було використано:

пояснення принципу та приклад.

Що зроблено студентом:

алгоритм реалізовано самостійно, перевірено за FORMAT_SPEC та еталонними даними.

Приклад 2

Задача:

пошук крайніх випадків для parser.

Що було використано:
список можливих negative tests.

Що зроблено студентом:
тести адаптовано до формату проєкту та реалізовано самостійно.

Що не передавалося ШІ

Наприклад:

- приховані дані;
- паролі;
- секретні ключі;
- закриті матеріали викладача.

Відповідальність

Команда перевірила весь код, формули, тести і результати та несе відповідальність за їх правильність.

Додаток В. Шаблон підсумкового звіту

Рекомендована структура:
ТИТУЛЬНА СТОРІНКА

1. Постановка задачі
2. Архітектура системи
3. Алгоритми і структури даних
4. Реалізація
5. Тестування
6. Експериментальне дослідження
7. Результати
8. Аналіз результатів
9. Відомі обмеження
10. Висновки
11. Внесок учасників
12. Використані джерела
13. Додатки

Додаток Г. Шаблон розділу «Постановка задачі»

Можна використовувати таку структуру:
Метою проєкту є створення ...

На вхід система отримує ...

Система повинна ...

Основними обмеженнями є ...

Якість системи оцінюється за ...

Основними алгоритмічними задачами є ...

Не потрібно копіювати повну умову.

Потрібно коротко описати задачу своїми словами.

Додаток Д. Шаблон опису алгоритму

Для кожного основного алгоритму зручно використовувати однакову структуру.

Назва алгоритму

1. Задача

Яку проблему він розв'язує у нашому проєкті?

2. Вхід

Які дані отримує?

3. Вихід

Що повертає?

4. Ідея

Коротко пояснити принцип.

5. Псевдокод

...

6. Основний інваріант

Що залишається правильним під час роботи алгоритму?

7. Складність

Час:

$O(\dots)$

Пам'ять:

$O(\dots)$

8. Крайні випадки

...

9. Як перевіряли

...

10. З чим порівнювали

...

Додаток Е. Шаблон опису експерименту

Кожен важливий експеримент бажано оформляти однаково.
Назва експерименту

Мета:

...

Дослідницьке питання:

...

Гіпотеза:

...

Алгоритм А:

...

Алгоритм В:

...

Дані:

...

Параметри:

...

Seed:

...

Commit/tag:

...

Кількість запусків:

...

Метрики:

...

Результат:

...

Висновок:

...

Додаток Ж. Приклад експерименту «DP проти greedy»

Назва:

Порівняння DP та жадібного алгоритму

Питання:

Чи дає DP кращий результат за greedy і якою ціною за часом?

Дані:

100 тестових наборів.

Умови:

обидва алгоритми отримують однакові дані та однаковий бюджет.

Метрики:

- objective value;
- час виконання;
- відносна різниця.

Seed:

2001

Результат:

...

Висновок:

...

Додаток 3. Шаблон журналу експериментів

Журнал не повинен бути складним.

Його задача — дозволити через кілька тижнів зрозуміти, звідки взявся певний результат.

Можна вести файл:

experiments/log.md

Шаблон одного запису:

2026-09-10 – DP vs greedy

Commit:

``a81f20c``

Seed:

``2001``

Dataset:

``corpus/test_01``

Parameters:

- budget = ...
- block_size = ...

Run:

```
```bash
python experiments/run_dp_vs_greedy.py
```
```

Result:

- DP = ...
- Greedy = ...
- DP time = ...
- Greedy time = ...

Observation:

DP дав кращий результат на ..., але ...

Next:

Перевірити ...

Додаток II. Мінімальний машинозчитуваний журнал експериментів

Для автоматичної обробки можна зберігати JSON:

```
{
  "date": "2026-09-10",
  "commit": "a81f20c",
  "seed": 2001,
  "dataset": "test_01",
  "algorithm": "dp",
  "parameters": {
    "budget": 1000
  },
  "metrics": {
    "value": 812,
    "time_s": 0.35
  }
}
```

Або CSV:

date,commit,seed,dataset,algorithm,value,time_s
2026-09-10,a81f20c,2001,test_01,dp,812,0.35
2026-09-10,a81f20c,2001,test_01,greedy,784,0.01

Додаток К. Шаблон журналу знайдених помилок

Корисно окремо фіксувати суттєві bugs.
BUG-014

Дата:

...

Опис:

Decoder падає на виродженому Huffman-алфавіті.

Мінімальний приклад:

```
```text  
AAAAAAAAA
```
```

Очікування:

round-trip проходить.

Фактичний результат:

...

Причина:

...

Виправлення:

...

Regression test:

```
`test_single_symbol_huffman`
```

Commit:

...

Головне правило:

Кожна знайдена суттєва помилка повинна після виправлення залишити після себе regression test.

Додаток Л. Чек-лист Зрізу 0

- ☐ Прочитана повна умова проєкту
- ☐ Прочитана специфікація формату
- ☐ Переглянуті видані дані
- ☐ Створено репозиторій
- ☐ Створено README
- ☐ Створено META-USAGE
- ☐ Визначено архітектуру
- ☐ Визначено основні модулі
- ☐ Погоджено інтерфейси
- ☐ Розподілено зони відповідальності
- ☐ Зафіксовано рівень кожної зони
- ☐ Створено початкові issue
- ☐ Складено план роботи за чекпоінтами
- ☐ Визначено основні ризики
- ☐ Обрано варіант проєкту і мову реалізації
- ☐ Зафіксовано склад приймальних перевірок
- ☐ Отримано банк питань до захисту

Додаток М. Чек-лист Чекпоінту 1

- ☐ Є мінімальний наскрізний тракт
- ☐ Основний формат реалізовано
- ☐ Writer працює
- ☐ Parser працює
- ☐ Є round-trip
- ☐ Є unit tests
- ☐ Є edge-case tests
- ☐ Еталонні дані читаються
- ☐ README містить актуальний запуск
- ☐ Усі тести запускаються однією командою
- ☐ Базову метрику виміряно
- ☐ Поточний commit/tag зафіксовано

Додаток Н. Чек-лист Чекпоінту 2

- ☐ Основні алгоритми базового рівня реалізовано
- ☐ Greedy працює

- ☐ DP працює
- ☐ DP перевірено на малих задачах
- ☐ Є порівняння DP vs greedy
- ☐ Є вимірювання часу
- ☐ Реалізовано classifier, якщо він входить у проєкт
- ☐ Є macro-F1 на valid
- ☐ Наскрізний тракт після інтеграції проходить
- ☐ Результати експериментів збережені
- ☐ Старі тести не зламані

Додаток О. Чек-лист Чекпоінту 3

- ☐ Усі базові компоненти інтегровані
- ☐ Crypto працює
- ☐ HMAC перевіряється до decrypt
- ☐ Negative tests проходять
- ☐ GA працює
- ☐ Є elitism test
- ☐ GA порівняно з random search
- ☐ Основна Q рахується автоматично
- ☐ Є результати ablation
- ☐ Є майже готовий звіт
- ☐ Усі відомі критичні bugs закрито

Додаток П. Чек-лист перед pull request

- ☐ Код запускається
- ☐ Нові функції мають тести
- ☐ Усі старі тести проходять
- ☐ Немає debug print
- ☐ Немає паролів або ключів у коді
- ☐ Немає великих випадкових файлів у commit
- ☐ Назви зрозумілі
- ☐ Помилки обробляються контрольовано
- ☐ README оновлено, якщо змінився запуск
- ☐ Issue пов'язаний із PR

Додаток Р. Чек-лист code review

Під час review іншого модуля перевірте:

- ☐ Чи зрозуміло, що робить код?
- ☐ Чи відповідає він умові?

- [] Чи відповідає FORMAT_SPEC?
- [] Чи правильно обробляються межі?
- [] Чи є unit tests?
- [] Чи є negative tests?
- [] Чи немає прихованого дублювання коду?
- [] Чи не змішані різні відповідальності?
- [] Чи правильно оцінена складність?
- [] Чи немає очевидної можливості аварійного завершення?

Додаток С. Чек-лист алгоритму DP

- [] Чітко визначено стан
- [] Визначено базовий випадок
- [] Визначено всі переходи
- [] Визначено порядок обчислення
- [] Відновлюється саме рішення
- [] Перевіряється бюджет
- [] DP порівнюється з brute force на малих задачах
- [] Є boundary tests
- [] Вказана часова складність
- [] Вказана просторова складність

Додаток Т. Чек-лист greedy

- [] Чітко визначено локальне правило
- [] Tie-breaking детермінований
- [] Є тест правильного порядку вибору
- [] Є контрприклад, де greedy не оптимальний
- [] Greedy порівнюється з точним алгоритмом
- [] Вимірюється час
- [] Не заявляється оптимальність без доведення

Додаток У. Чек-лист Huffman

- [] Правильно рахуються частоти
- [] Priority queue працює
- [] Побудова дерева детермінована
- [] Canonical codes будуються правильно
- [] Є тест одного символу
- [] Є тест двох символів
- [] Перевіряється максимальна довжина
- [] Перевіряється префіксність

- [] Перевіряється рівність Крафта, якщо потрібна форматом
- [] Encode → decode дає original
- [] Еталонний файл читається

Додаток Ф. Чек-лист LZ77

- [] Literal працює
- [] Match працює
- [] MIN_MATCH враховано
- [] MAX_MATCH враховано
- [] Максимальна distance перевіряється
- [] Немає посилання за межі буфера
- [] Є тест повторюваного рядка
- [] Є тест випадкових даних
- [] Greedy працює
- [] Lazy greedy працює
- [] Round-trip проходить

Додаток Х. Чек-лист MLP

- [] Ознаки обчислюються правильно
- [] Ознаки нормалізовані, якщо потрібно
- [] Forward pass перевірено
- [] Loss перевірено
- [] Backpropagation перевірено
- [] Навчання реально зменшує loss
- [] Train і valid не змішані
- [] Holdout не відтворюється і не замінюється власною вибіркою
- [] Ознаки включають вікно історії, а не лише поточний запис
- [] Є щонайменше кілька ініціалізацій
- [] Модель вибрана за valid
- [] Macro-F1 рахується правильно
- [] Є confusion matrix
- [] Є baseline

Додаток Ц. Чек-лист GA

- [] Визначено хромосому
- [] Межі кожного гена правильні
- [] Fitness перевірена окремо
- [] Population генерується правильно
- [] Selection працює

- [] Crossover працює
- [] Mutation не виходить за межі
- [] Elitism працює
- [] `best_final >= best_initial`
- [] Є фіксований seed для відтворення
- [] GA порівнюється з random search
- [] Однакова кількість fitness evaluations
- [] Збережено history поколінь

Додаток Ч. Чек-лист криптографії

- [] Використовується алгоритм зі специфікації
- [] KDF має правильні параметри
- [] `K_enc` і `K_mac` різні
- [] Salt створюється правильно
- [] Nonce не повторюється
- [] CTR counter починається з правильного значення
- [] AES перевірено known test vector
- [] HMAC рахується над правильними даними
- [] HMAC перевіряється до decrypt
- [] HMAC порівнюється constant-time
- [] Wrong password дає контрольовану відмову
- [] Змінений ciphertext відхиляється
- [] Секретні ключі й пароль не пишуться в log
- [] KDF виконується один раз, ключі кешуються
- [] Порядок «стиснути → зашифрувати» перевірено вимірюванням

Додаток Ш. Чек-лист parser

- [] Перевіряється minimum length
- [] Перевіряється magic
- [] Перевіряється version
- [] Перевіряються всі довжини
- [] Перевіряються reserved поля
- [] Перевіряються невідомі flags
- [] Перевіряються індекси
- [] Перевіряються діапазони значень
- [] Зайві байти не приймаються
- [] Обрізані дані не викликають crash
- [] Некоректні дані дають контрольований error code

Додаток Щ. Чек-лист negative testing

- ☐ bad magic
- ☐ bad version
- ☐ truncated input
- ☐ wrong length
- ☐ unknown flag
- ☐ reserved != 0
- ☐ bad HMAC
- ☐ wrong password
- ☐ invalid code table
- ☐ invalid index
- ☐ duplicate object
- ☐ missing object
- ☐ corrupted payload
- ☐ extra bytes
- ☐ unsafe filename, якщо проєкт працює з файлами

Не всі пункти застосовуються до кожного проєкту.
Вибирайте ті, що відповідають вашому формату.

Додаток Ю. Чек-лист експерименту

Перед тим як включити experiment у звіт:

- ☐ Є чітке питання?
- ☐ Є baseline?
- ☐ Змінюється лише один фактор?
- ☐ Алгоритми отримують ті самі дані?
- ☐ Seed зафіксовано?
- ☐ Commit зафіксовано?
- ☐ Параметри зафіксовано?
- ☐ Кількість запусків достатня?
- ☐ Метрики визначені до запуску?
- ☐ Сирі результати збережені?
- ☐ Графік можна перебудувати?
- ☐ Є пояснення результату?

Додаток Я. Чек-лист фінального звіту

- ☐ Постановка задачі написана своїми словами
- ☐ Є схема системи
- ☐ Пояснено роль кожного основного алгоритму

- ☐] Є псевдокод ключових алгоритмів
- ☐] Вказана складність
- ☐] Описано тестування
- ☐] Є edge cases
- ☐] Є negative tests
- ☐] Є основні порівняльні експерименти
- ☐] Є baseline
- ☐] Є аналіз, а не лише числа
- ☐] Описані негативні результати
- ☐] Описані обмеження
- ☐] Є внесок кожного студента
- ☐] Є посилання на джерела
- ☐] Результати відповідають final tag
- ☐] META-USAGE актуальний

Додаток АА. Чек-лист фінального clean run

Перед здачею:

- ☐] Створено чистий каталог
- ☐] Репозиторій клоновано заново
- ☐] Checkout final tag
- ☐] Створено нове virtual environment
- ☐] Встановлено залежності тільки з файлів проєкту
- ☐] Усі unit tests проходять
- ☐] Усі integration tests проходять
- ☐] Negative tests проходять
- ☐] Основний проєкт запускається
- ☐] Експерименти запускаються
- ☐] Результати відтворюються
- ☐] Немає залежності від локальних файлів автора
- ☐] Немає залежності від IDE

Додаток АБ. Чек-лист готовності до захисту

Кожен студент окремо повинен перевірити:

- ☐] Я знаю свою зону відповідальності
- ☐] Я можу знайти свій код
- ☐] Я можу запустити свої тести
- ☐] Я можу пояснити алгоритм без читання коду
- ☐] Я можу пояснити його складність
- ☐] Я знаю хоча б один крайній випадок

- [] Я знаю хоча б одну знайдену помилку
- [] Я можу пояснити, як її виправили
- [] Я можу пояснити свій експеримент
- [] Я можу пояснити отриманий результат
- [] Я знаю обмеження свого рішення

Додаток АВ. Чек-лист команди перед захистом

- [] Final tag створено
- [] README актуальний
- [] Усі тести проходять
- [] Усі результати отримані з final tag
- [] Звіт фінальний
- [] META-USAGE фінальний
- [] Усі issue чекпоінтів закриті або пояснені
- [] Відомі обмеження записані
- [] Демонстрацію перевірено на чистому середовищі
- [] Кожен учасник знає свою частину

Додаток АГ. Рекомендована структура фінального репозиторію

Приблизно:

```
project/
├── src/
├── tests/
├── experiments/
├── results/
├── docs/
│   └── decisions.md      лог рішень (не менше п'яти записів)
├── data/
├── README.md
├── META-USAGE.md
├── requirements.txt
└── pyproject.toml
```

Не обов'язково використовувати всі каталоги.

Наприклад, requirements.txt і pyproject.toml одночасно можуть бути не потрібні.

Головне — щоб структура була зрозумілою.

Додаток АД. Рекомендована структура experiments/

experiments/

- run_baseline.py
- run_dp_vs_greedy.py
- run_classifier.py
- run_ga.py
- run_ablation.py
- run_final.py

Рекомендується, щоб:
run_final.py

дозволяв відтворити основні фінальні показники.

Додаток АЕ. Рекомендована структура results/

results/

- baseline.json
- dp_vs_greedy.csv
- classifier_metrics.json
- confusion_matrix.csv
- ga_history.csv
- ablation.csv
- final_metrics.json

Ці файли бажано генерувати автоматично.

Додаток АЄ. Короткий шаблон фінальної демонстрації

Демонстрацію краще будувати не як довгу презентацію коду, а як послідовність перевірок.

1. Показати final tag
2. Запустити:
pytest
3. Показати:
усі тести проходять
4. Запустити:
мінімальний правильний приклад

5. Запустити:
один negative test
6. Показати:
основний алгоритмічний experiment
7. Показати:
фінальну метрику Q
8. Коротко показати:
results/
9. Відкрити:
один модуль за питанням викладача

Додаток АЖ. Що робити, якщо під час захисту знайдено помилку

Не потрібно намагатися приховати її.

Правильна поведінка:

1. відтворити проблему;
2. визначити, який компонент її спричинив;
3. пояснити очікувану поведінку;
4. запропонувати тест;
5. пояснити, як саме потрібно виправити код.

Якщо студент може правильно проаналізувати нову помилку, це також показує розуміння системи.

Додаток АЗ. Головний фінальний чек-лист

Перед тим як сказати:

«Проект готовий»,

команда повинна відповісти «так» на п'ять головних питань.

1. ЧИ ПРАЦЮЄ?

[] Основна задача виконується.

2. ЧИ ПРАВИЛЬНО ПРАЦЮЄ?

[] Результат перевірений автоматичними тестами.

3. ЧИ ПРАВИЛЬНО ПАДАЄ?

[] Некоректні дані обробляються контрольовано.

4. ЧИ МОЖНА ПОВТОРИТИ?

[] Інша людина може отримати той самий результат з README.

5. ЧИ МОЖЕМО ПОЯСНИТИ?

[] Кожен учасник розуміє свою частину.

Якщо всі п'ять відповідей позитивні, проєкт справді близький до завершеного.

Підсумок

Усі шаблони й чек-листи цього розділу мають одну мету — не створити додаткову бюрократію, а допомогти команді не забути важливі речі.

Проєкт повинен пройти простий шлях:

зрозуміти

→ реалізувати

→ перевірити

→ виміряти

→ пояснити

→ відтворити

Якщо цей ланцюг виконується, студент демонструє повний цикл роботи з алгоритмічною задачею — від постановки до перевіреного інженерного результату.

Карта документів

Довідкова сторінка. Тримайте її під рукою: за три місяці роботи згадати, у якому саме документі шукати відповідь, буває важче, ніж знайти саму відповідь.

Хто що видає і як часто змінюється

| Документ | Хто видає | Область дії | Оновлення |
|--------------------------|-----------|--------------|--------------------------|
| Положення про оцінювання | кафедра | усі варіанти | при зміні правил кафедри |

| Документ | Хто видає | Область дії | Оновлення |
|----------------------------------|-----------|--------------|-----------------------------------|
| Умова проєкту | викладач | один варіант | щороку |
| FORMAT_SPEC | викладач | один варіант | при зміні версії формату |
| CHANNEL_SPEC / REPLICATION_SPEC | викладач | один варіант | разом із моделлю середовища |
| channel.py / replication.py | викладач | один варіант | разом зі специфікацією середовища |
| team_config.json | викладач | одна команда | щосеместру, індивідуально |
| Дані та reference/ | викладач | одна команда | щосеместру, індивідуально |
| Методичка | кафедра | усі варіанти | раз на кілька років |
| Репозиторій, звіт, META-USAGE.md | команда | своя робота | постійно |

Де шукати відповідь

| Питання | Документ |
|--|--|
| Що взагалі треба зробити? | умова свого варіанта |
| Які зони відповідальності існують і які в них рівні? | умова свого варіанта, таблиця зон |
| Які приймальні перевірки я маю пройти? | умова свого варіанта |
| Скільки байтів у полі, який порядок байтів? | FORMAT_SPEC |
| Який код відмови повернути? | FORMAT_SPEC, розділ про коди |
| Як обчислюються ваги для рюкзака? | FORMAT_SPEC, розділ про місткість і ваги |
| Які фіксовані вартості для ДП? | FORMAT_SPEC і team_config.json |
| Як влаштована модель втрат чи відмов? | CHANNEL_SPEC / REPLICATION_SPEC |
| Яке значення порога, коефіцієнта, seed у нас? | team_config.json |

| | |
|--|---|
| Питання | Документ |
| Який очікуваний результат для пошкодженого зразка? | reference/expected_failures.json |
| З чого почати і що робити далі? | методичка, розділ 5 |
| Як це протестувати? | методичка, розділ 6 і чек-листи додатків |
| Що показати на чекпоінті? | методичка, розділ 7 і розділ свого варіанта |
| Як оформити звіт? | методичка, розділ 8 |
| Що специфічного саме в моєму варіанті? | методичка, розділ 9, 10 або 11 |
| Скільки за це балів? | Положення про оцінювання |
| Чому мій результат не читається чужою системою? | FORMAT_SPEC і еталонні зразки |

Що лишається у викладача

Частина матеріалів команді не видається, і це зроблено навмисно.

| | |
|------------------------------------|--|
| Матеріал | Навіщо прихований |
| Прихована вибірка holdout | підсумкова якість класифікатора має вимірюватися на даних, яких модель не бачила; правило «використати один раз» неможливо проконтролювати, якщо файл лежить у команди |
| Приховані значення seed середовища | інакше можна підлаштувати рішення під конкретну послідовність втрат чи відмов замість того, щоб робити загальний алгоритм |
| Генератор комплекту | містить готові реалізації Гаффмана, LZ77, ДП і криптографічного шару, тобто саме те, що студент має написати сам |

Відкриті значення `seed` у виданому комплекті призначені для налагодження. Офіційні вимірювання виконуються в однаковому для всіх середовищі і, де це передбачено умовою варіанта, на прихованих `seed` з усередненням.

Правило пріоритету

Положення про оцінювання

↑

Умова проєкту

↑

FORMAT_SPEC та інші специфікації

↑

Методичка

Документ, розташований вище, важливіший. Методичка нічого не додає до вимог і нічого з них не скасовує.

Якщо ви помітили розбіжність між документами, повідомте викладача. Обирати зручніший варіант не можна: на захисті питатимуть за нормативним документом.

Два режими читання

Це найчастіше джерело непорозумінь, тому варто зафіксувати окремо.

Специфікація — контракт. Вона або виконана точно, або не виконана. Поле в два байти не можна записати чотирма, навіть якщо ваш власний читач це розуміє. Тлумачити її на власний розсуд не можна.

Методичка — рекомендації. Порядок реалізації, організація тестів, структура звіту допускають різні рішення. Тут є простір для вибору, і за нього ніхто не карає.

Плутати ці режими шкідливо в обидва боки. Хто тлумачить специфікацію, отримує систему, несумісну з еталонними даними. Хто сприймає кожну пораду методички як жорстку вимогу, витрачає час на непотрібну формальність.

Перелік рекомендованої літератури

1. Бичков О.С. Алгоритми і структури даних. Електронне видання. 2963 стор.