МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, програмної інженерії та комп'ютерних наук
Кафедра інформаційних технологій

Пояснювальна записка

до кваліфікаційної роботи
другого (магістерського) рівня

на тему **РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ОПТИМІЗАЦІЇ МНОЖЕННЯ ПОСЛІДОВНОСТІ МАТРИЦЬ ЗА ДОПОМОГОЮ ДИНАМІЧНОГО ПРОГРАМУВАННЯ**

Виконав: студент 2 курсу, групи ІПЗ
спеціальності
121 Інженерія програмного забезпечення
Новочинський Є.Г.

Керівник Григор'єва Т.І.

Рецензент Т.М. Соловська

Одеса – 2023 р.

Д О В І Д К А

кафедри ІТ про виконану магістерську роботу

студента 2 курсу ФКПІ та КН групи ІПЗ

Новочинського Євгена Григоровича

на тему Розробка програмного забезпечення оптимізації множення послідовності матриць за допомогою динамічного програмування

Висновок нормоконтролера пояснювальна записка до кваліфікац. роботи вик. з урахуванням вимог ДСТУ, робота виконана згідно внутрішнього положення ДНТУ

Нормоконтролер асист. каф. ІТ ______ 15.12.2023 Келемешева І.В.
(науковий ступінь, вчене звання, посада) (підпис, дата) (і. б. прізвище)

Висновок відповідального за наявність плагіату згідно з сертифікатом ID 1015677894 унікальність роботи підтверджено

Відповідальна особа асист. каф. ІТ ______ 15.12.2023 Келемешева І.В.
(науковий ступінь, вчене звання, посада) (підпис, дата) (і. б. прізвище)

Попередня експертиза (захист) магістерської роботи
(бакалаврської роботи чи магістерської роботи)

студ. Новочинський Є.Г. проведена “ 12 ” 12 2023 р.
(прізвище і.б.)

Висновки Виконана розробка відповідає завданню. У магістерській роботі проведено аналіз видів використання динамічного програмування для задач отримання рішення множення матриць. Тема роботи є актуальною, перевагою отриманих рішень в результаті виконання завдання є можливість скоротити застосування методів динамічного програмування. Виконується процес покрокового вирішення задачі. Робота магістра відповідає вимогам стандартам та рекомендується до захисту

Члени комісії ______ д.т.н., проф. Стрелковська І.В.
(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

______ к.т.н., доц. Григор'єва Т.Г.
(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

______ к.т.н., доц. Горбачов В.Є.
(підпис) (науковий ступінь, вчене звання, посада, прізвище і.б.)

МІЖНАРОДНИЙ ГУМАНІТАРНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, програмної інженерії та комп'ютерних наук
Кафедра інформаційних технологій
Освітній ступінь магістр
Галузь знань 12 Інформаційні технології
Спеціальність 121 Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
к.т.н., доц.
_____ Т.І. Григор'єва
"25." 09 2023 року

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ РОБОТУ

Новочинському Євгену Григоровичу

1. Тема роботи Розробка програмного забезпечення оптимізації множення послідовності матриць за допомогою динамічного програмування
керівник роботи Григор'єва Тетяна Ігорівна, к.т.н., доцент
затверджені наказом закладу вищої освіти від 25.09.2023 р. № 1957
зі змінами від 04.12.2023 р. № 3102
2. Строк подання студентом роботи 11.12.2023 р.
3. Вихідні дані до роботи
1 C++
2 Алгоритми динамічного програмування
3 Алгоритми множення матриць

4. Зміст розрахунково-пояснювальної записки
1 Динамічне програмування
2 Множення матриць
3 Сукупність динамічного програмування
4 Алгоритми, приклади та рішення задач
5 Тестування

5. Перелік графічного матеріалу (з зазначенням обов'язкових креслень)

Слайд 1 – Основні характеристики роботи
Слайд 2 – Динамічне програмування
Слайд 3 – Рішення завдання динамічним програмуванням
Слайд 4 – Множення ланцюжка матриць
Слайд 5 – Рекурсивне рішення
Слайд 6 – Приклад множення матриць

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Вступ	25.09.2023-01.10.2023	вик
2	Динамічне програмування	02.10.2023-08.10.2023	вик
3	Множення матриць	09.10.2023-15.10.2023	вик
4	Сукупність динамічного програмування	16.10.2023-05.11.2023	вик
5	Алгоритми, приклади та рішення задач	06.11.2023-12.11.2023	вик
6	Тестування	13.11.2023-19.11.2023	вик
7	Висновки та рекомендації	20.11.2023-26.11.2023	вик
8	Перелік джерел посилання	27.11.2023-03.12.2023	вик
9	Додаток А. Перелік копій демонстраційного матеріалу	04.12.2023-10.12.2023	вик

Студент _____ Є.Г. Новочинський
(підпис)

Керівник роботи _____ Т.І. Григор'єва
(підпис)

ВІДГУК

наукового керівника на кваліфікаційну

магістерську роботу

на тему «**Розробка програмного забезпечення для задачі оптимального перемноження послідовності матриць за допомогою динамічного програмування**»

студента Новочинського Є. Г.

Спеціальність: 121 Інженерія програмного забезпечення

Оскільки, динамічне програмування призначено для підвищення ефективності обчислень при вирішенні задач шляхом їх декомпозиції на відносно прості підзадачі, цей метод є важливим інструментом у різних галузях для розв'язання складних задач оптимізації, що свідчить про актуальність теми магістерської роботи.

В роботі проведено аналіз умов, переваг та недоліків використання динамічного програмування та розробленно програмне забезпечення для задачі оптимального перемноження послідовності матриць за допомогою динамичного програмування.

Результати дослідження представлені у тезах доповідей IX Всеукраїнської науково-практичної конференції студентів, аспірантів та молодих учених «Гуманітарний і інноваційний ракурс професійної майстерності: пошуки молодих вчених».

В роботі було проведено тестування не всіх розглянутих задач, що можна віднести до недоліків магістерської роботи. Вказане зауваження не знижує цінності виконаної роботи.

В цілому магістерська робота студента Новочинського Є. Г. відповідає вимогам до кваліфікаційних робіт магістрів і заслуговує на оцінку "добре". Студент Новочинський Є.Г. заслуговує присвоєння кваліфікації магістр з інженерії програмного забезпечення за заявленою спеціальністю 121 «Інженерія програмного забезпечення».

Науковий керівник:
завідувачка кафедри Інформаційних технологій,
кандидат технічних наук, доцент Т.І.Григор'єва

РЕЦЕНЗІЯ

на кваліфікаційну роботу другого (магістерського) рівня здобувача
Новочинського Євгена Григоровича

на тему «Розробка програмного забезпечення оптимізації множення послідовності матриць за допомогою динамічного програмування»

Актуальність теми магістерської роботи підтверджується тим, що динамічне програмування є важливим інструментом для розв'язання складних задач оптимізації.

У магістерській роботі проведено аналіз умов використання динамічного програмування для задач отримання рішення множення матриць. Актуальність теми безсумнівна, адже іноді перебір допустимих рішень в результаті виконання завдання вдається істотно скоротити застосуванням методів динамічного програмування. Виконується процес покрокового вирішення завдань, коли на кожному кроці з безлічі допустимих рішень обирається одне рішення, воно оптимізує цільову функцію. При цьому кожна з підзадач вирішується тільки один раз і її рішення запам'ятовується в спеціальній таблиці.

До недоліків роботи слід віднести:

– доцільно було б розглянути більше прикладів рішення задач за допомогою динамічного програмування,

– треба було більше приділити уваги функціональному тестуванню, та тестуванню інтерфейсу користувача.

Проте, зазначені недоліки не знижують цінності виконаної роботи.

У цілому, кваліфікаційна робота Новочинського Є.Г. відповідає вимогам до випускних кваліфікаційних робіт здобувачів другого (магістерського) рівня та заслуговує оцінки «добре».

Студент Новочинського Є.Г. заслуговує присвоєння кваліфікації магістр з інженерії програмного забезпечення за заявленою спеціальністю 121 Інженерія програмного забезпечення.

Рецензент,
к.т.н., доцент кафедри
комп'ютерних наук

І.М. Соловська

UNICHECK
by Turnitin

Ім'я користувача:
Анна Серединко

ID перевірки:
1015995243

Дата перевірки:
11.12.2023 23:55:20 MSK

Тип перевірки:
Doc vs Internet + Library

Дата звіту:
11.12.2023 23:59:35 MSK

ID користувача:
100001433

Назва документа: Новочинський_диплом

Кількість сторінок: 64 Кількість слів: 8660 Кількість символів: 60808 Розмір файлу: 1.82 MB ID файлу: 1015677894

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

28.4%
Схожість

Найбільша схожість: 11% з Інтернет-джерелом (https://stud.wiki/programming/2c0b65635b2ac69b4c53a88521216c26_0.h.

27.9% Джерела з Інтернету | 973 ..... Сторінка 66

3.48% Джерела з Бібліотеки | 27 ..... Сторінка 70

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи | 15

Підозріле форматування | 16 сторінок

РЕФЕРАТ

Пояснювальна записка до магістерської роботи:: 43 с., 29 рис., 10 табл., 2 додатка, 10 джерел.

Метою роботи є оптимізація послідовності множення матриць за допомогою динамічного програмування. В процесі застосування парадигми динамічного програмування для розв'язання складної задачі оптимального перемноження послідовності матриць, були визначені ключові етапи та вирішені основні завдання. Для реалізації рішення задачі обрано програмне забезпечення Microsoft Visual Studio, лінійка продуктів компанії Microsoft, що включає інтегроване середовище розробки програмного забезпечення, модулі тестування, пошуку помилок і ряд інших інструментальних засобів.

Проведено аналіз умов використання динамічного програмування для задач, які для отримання рішення використовують множення матриць.

ДИНАМІЧНЕ ПРОГРАМУВАННЯ, МНОЖЕННЯ МАТРИЦЬ, ОПТИМІЗАЦІЯ, MICROSOFT, VISUAL STUDIO, СКРИПТИ, РОЗРОБКА ПРИКЛАДНОЇ ПРОГРАМИ, ПРОЕКТУВАННЯ ДІАГРАМ, C++

ЗМІСТ

ВСТУП.....10
РОЗДІЛ 1. Динамічне програмування.....11
1.1 Поняття динамічного програмування 11
1.2 Рекурсивні рішення 12
1.3 Послідовне обчислення та створення індексів.....14
РОЗДІЛ 2. Множення матриць.....18
РОЗДІЛ 3. Сукупність динамічного програмування.....29
3.1 Динамічне програмування за профілем.....29
3.2 Завдання про замощення доміно 29
РОЗДІЛ 4. Алгоритми, рішення та приклади задач.....34
4.1 Задача "Сходи".....34
4.2 Алгоритм Нідлмана-Вунша.....35
РОЗДІЛ 5. Тестування.....44
5.1 Побудова діаграм.....44
5.2 Множення матриць 49
ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ.....52
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ.....53
Додаток А ПЕРЕЛІК КОПІЙ ДЕМОНСТРАЦІЙНОГО МАТЕРІАЛУ.....54
Додаток Б ЛІСТИНГ КОДУ.....58

ВСТУП

Іноді перебір допустимих рішень в результаті виконання завдання вдається істотно скоротити застосуванням методів динамічного програмування. Динамічне програмування – це процес покрокового вирішення завдань, коли на кожному кроці з безлічі допустимих рішень вибирається одне рішення, оптимізує цільову функцію. У методі гілок і меж завдання ділиться на перетинаючи підзадачі, рішенням вихідної задачі є найкраще з рішень підзадач. Динамічне програмування застосовують тоді, коли у завдання є однакові підзадачі. При цьому кожна з підзадач вирішується тільки один раз і її рішення запам'ятовується в спеціальній таблиці. Це дозволяє не шукати вирішення одних і тих же підзадач багаторазово, при необхідності відповідь береться з таблиці. В основі алгоритму, заснованого на динамічному програмуванні, зазвичай лежить рекурентне співвідношення, що зв'язує оптимальні значення цільових функцій підзадач.

Матриці в математиці – один з найважливіших об'єктів, що мають прикладне значення. Матриці проникли майже в усі галузі людської діяльності. Вони використовуються в математиці, у фізиці, в техніці, в економіці, в теорії управління, статистики, інших областей науки і знань. Для вирішення різних практичних завдань використовується програмне забезпечення, наприклад, Microsoft Excel, MathCad, Maple, яке дозволяє маніпулювати цілими масивами без виконання обчислень вручну. Множення матриць – одна з основних операцій над матрицями, ключовими особливостями будь-якого матричного множення є: кількість рядків і стовпців, в початкових матрицях, і правило, як елементи матриць утворюють нову матрицю. Тому оптимізація послідовності множення матриць за допомогою динамічного програмування є актуальною задачею прикладного значення.

1 ДИНАМІЧНЕ ПРОГРАМУВАННЯ

1.1 Поняття динамічного програмування

Динамічне програмування – метод вирішення задачі шляхом її розбиття на кілька однакових підзадач, рекурентно пов'язаних між собою. Найпростішим прикладом будуть числа Фібоначчі – щоб обчислити деяке число в цій послідовності, нам потрібно спершу обчислити третє число, склавши перші два, потім четвертий таким же чином на основі другого і третього, і так далі.

Рішення завдання динамічним програмуванням містить Залежність елементів динаміки один від одного. Така залежність може бути прямо дана в умови (так часто буває, якщо це завдання на числові послідовності). В іншому випадку ви можете спробувати дізнатися якийсь відомий числовий ряд (на кшталт тих же чисел Фібоначчі), обчисливши перші кілька значень вручну. Якщо вам зовсім не пощастило - доведеться думати. Також значення початкових станів. В результаті довгого розбиття на підзадачі вам необхідно звести функцію або до уже відомих значеннях (як у випадку з Фібоначчі — заздалегідь визначені перші два члена), або до задачі, розв'язуваної елементарно.

Є й інші підходи до реалізації динаміки. Розберемо їх на прикладі наступної задачі:

Обчислити n-й член послідовності, заданої формулами:

$a_{2n} = a_n + a_{n-1}$,

$a_{2n+1} = a_n - a_{n-1}$,

$a_0 = a_1 = 1$.

Рішення:

Тут нам дано і початкові стану ($a_0 = a_1 = 1$), і залежності. Єдина складність, яка може виникнути - розуміння того, що 2n – умова парності числа, а 2n + 1 – непарності. Іншими словами, нам потрібно перевіряти, парно чи число, і вважати його в залежності від цього за різними формулами.

1.2 Рекурсивні рішення

У програмуванні рекурсія – виклик функції чи процедури з неї самої (звичайно з іншими значеннями вхідних параметрів) безпосередньо чи через інші функції (наприклад, функція А викликає функцію В, а функція В – функцію А). Кількість вкладених викликів функції чи процедури називається глибиною рекурсії.

Ціль рекурсивного визначення об'єкта в тім, що таке кінцеве визначення здатне описувати нескінченно велике число об'єктів. За допомогою ж рекурсивної програми можливо описати нескінченне обчислення, причому без явних повторень частин програми.

Існує спеціальний тип рекурсії, називаний «хвостовою рекурсією». Інтерпретатори і компілятори функціональних мов програмування, що підтримують оптимізацію коду (вихідного та/або такого, що виконується), реалізують хвостову рекурсію в обмеженому обсязі пам'яті за допомогою ітерацій.

Варто уникати надлишкової глибини рекурсії, бо це може викликати переповнення стека викликів (рис. 1.1).

Очевидна реалізація складається в написанні наступного методу:

```
private static int f (int n) {
      if (n == 0 || n == 1) return 1; // Перевірка на початкове значення

      if (n% 2 == 0) {// Перевірка на парність
            return f (n / 2) + f (n / 2-1); // Обчислюємо по формулі для парних
індексів,
                        // посилаючись на попередні значення
      } Else {
            return f ((n-1) / 2) -f ((n-1) / 2-1); // Обчислюємо по формулі для непарних
                              // індексів, посилаючись на попередні значення
      }
}
```

Рисунок 1.1 – Приклад рекурсивного рішення

І вона відмінно працює, але є нюанси. Якщо ми захочемо обчислити f (12), то метод буде обчислювати суму f (6) + f (5). У той же час, f (6) = f (3) + f (2) і f (5) = f (2) -f (1), тобто значення f (2) ми будемо обчислювати двічі. Порятунок від цього є - мемоізація (кешування значень).

Рекурсивне рішення з кешуванням значень

Ідея мемоізації (тобто, кешування значень) дуже проста – раз обчислюючи значення, ми заносимо його в якусь структуру даних. Перед кожним обчисленням ми перевіряємо, чи є обчислюється значення в цій структурі, і якщо є, використовуємо його. Як структури даних можна використовувати масив, заповнений флаговими значеннями. Якщо значення елемента за індексом N дорівнює значенню прапора, значить, ми його ще не обчислювали. Це створює певні труднощі, тому що значення прапора не повинен належати безлічі значень функції, яка не завжди очевидно. Особисто я вважаю за краще використовувати хеш-таблицю – всі дії в ній виконуються за O (1), що дуже зручно. Однак, при великій кількості значень два числа можуть мати однаковий хеш, що, природно, породжує проблеми. У такому випадку варто використовувати, наприклад, червоно-чорне дерево.

Для вже написаної функції f(int) кешування значень (рис. 1.2) буде виглядати наступним чином:

```
private static HashMap <Integer, Integer> cache = new HashMap <Integer, Integer> ();

private static int fcashe (int n) {
if (! cache.containsKey (n)) {// Перевіряємо, чи знаходили ми дане значення
cache.put (n, f (n)); // Якщо немає, то знаходимо і записуємо в таблицю
}
return cache.get (n);
}
```

Рисунок 1.2 – Приклад рекурсивного рішення з кешуванням значень

Таке написання коду позбавляє від величезного числа операцій. Але це потребує зайвих витрат пам'яті.

1.3 Послідовне обчислення та створення індексів

Тепер повернемося до того, з чого почали – рекурсія працює повільно. Чи не занадто повільно, щоб це приносило дійсні неприємності в реальному житті, але на змаганнях зі спортивного програмування кожна мілісекунда на рахунку.

Метод послідовного обчислення підходить, тільки якщо функція посилається виключно на елементи перед нею – це його основний, але не єдиний мінус. Наше завдання цій умові задовольняє.

Суть методу в наступному: ми створюємо масив на N елементів і послідовно заповнюємо його значеннями (рис. 1.3). Ви, напевно, вже здогадалися, що таким чином ми можемо обчислювати в тому числі ті значення, які для відповіді не потрібні. У значної частини завдань на динаміку цей факт можна опустити, так як для відповіді часто бувають потрібні якраз все значення. Наприклад, при пошуку найменшого шляху ми не можемо не обчислювати шлях до якоїсь точки, нам потрібно переглянути всі варіанти. Але в нашій задачі нам потрібно обчислювати приблизно log2 (N) значень (на практиці більше), для 9223372036854775807-го елемента (MaxLong / 10) нам буде потрібно 172 обчислення.

```
private static int f (int n) {
if (n <2) return 1;
int [] fs = int [n]; // Створюємо масив для значень
fs [0] = fs [1] = 1; // Задаємо початкові стану

for (int i = 2; i <n; i ++) {
if (i% 2 == 0) {// Перевіряємо парність
fs [i] = fs [i / 2] + fs [i / 2-1];
} Else {
fs [i] = fs [(i-1) / 2] + fs [(i-1) / 2-1]
}
}

return fs [n-1];
}
```

Рисунок 1.3 – Приклад послідовного обчислення

Ще одним мінусом такого підходу є порівняно велика витрата пам'яті.

Створення стека індексів

Зараз нам належить, по суті, написати свою власну рекурсію. Ідея полягає в наступному – спочатку ми проходимо «вниз» від N до початкових станів, запам'ятовуючи аргументи, функцію від яких нам потрібно буде обчислювати.

Потім повертаємося «вгору», послідовно обчислюючи значення від цих аргументів, в тому порядку, який ми записали (рис. 1.4).

Залежності обчислюються наступним чином:

```
LinkedList <Integer> stack = new LinkedList <Integer> ();
stack.add (n);
{
LinkedList <Integer> queue = new LinkedList <Integer> ();
// Зберігаємо індекси, для яких ще не обчислені залежності
queue.add (n);
int dum;
while (queue.size ()> 0) {// Поки є що обчислювати
dum = queue.removeFirst ();
if (dum% 2 == 0) {// Перевіряємо парність
if (dum / 2> 1) {// Якщо обчислена залежність не належить початковим станам
stack.addLast (dum / 2); // Додаємо в стек
queue.add (dum / 2); // Зберігаємо, щоб обчислити подальші залежності
}
if (dum / 2-1> 1) {// Перевіряємо приналежність до початкових станів
stack.addLast (dum / 2-1); // Додаємо в стек
queue.add (dum / 2-1); // Зберігаємо, щоб обчислити подальші залежності
}
} Else {
if ((dum-1) / 2> 1) {// Перевіряємо приналежність до початкових станів
stack.addLast ((dum-1) / 2); // Додаємо в стек
queue.add ((dum-1) / 2); // Зберігаємо, щоб обчислити подальші залежності
}
if ((dum-1) / 2-1> 1) {// Перевіряємо приналежність до початкових станів
stack.addLast ((dum-1) / 2-1); // Додаємо в стек
queue.add ((dum-1) / 2-1); // Зберігаємо, щоб обчислити подальші залежності
}
}
```

Рисунок 1.4 – Приклад створення стека індексів

Отриманий розмір стека є і саме скільки обчислень нам буде потрібно зробити. Саме так я отримав згадане вище число 172.

Тепер ми по черзі витягаємо індекси і обчислюємо для них значення за формулами – гарантується, що всі необхідні значення вже будуть обчислені (рис. 1.5). Зберігати будемо як раніше – в хеш-таблиці:

```
HashMap <Integer, Integer> values = new HashMap <Integer, Integer> ();

values.put (0,1); // Важливо додати початкові стану в таблицю значень
values.put (1,1);

while (stack.size ()> 0) {
int num = stack.removeLast ();

if (! values.containsKey (num)) {// Цю конструкцію ви повинні пам'ятати з абзацу
про кешування
if (num% 2 == 0) {// Перевіряємо парність
int value = values.get (num / 2) + values.get (num / 2-1); // Обчислюємо значення
values.add (num, value); // Розміщуємо його в таблицю
} Else {
int value = values.get ((num-1) / 2) -values.get ((num-1) / 2-1); // Обчислюємо
значення
values.add (num, value); // Розміщуємо його в таблицю
}
}
```

Рисунок 1.5 – Приклад обчислення індексів

Всі необхідні значення обчислені, залишилося тільки написати потрібну функцію (рис. 1.6):

```
return values.get(n);
```

Рисунок 1.6 – Необхідний рядок у коді

Звичайно, таке рішення набагато більш трудомістке, однак це того варто.

Висновки до першого розділу

У першому розділі були розглянуті деякі моменти стосовно динамічного програмування. Були наведені приклади рішень та задач. Розглянули потрібні нам формули та алгоритми.

2 МНОЖЕННЯ МАТРИЦЬ

Перемноження ланцюжка матриць.

Наступний приклад динамічного програмування – алгоритм дозволяє вирішити задачу про перемножуванні ланцюжка матриць. Постановка завдання така: нехай є послідовність, що складається з n матриць, і потрібно обчислити їх добуток A1A2 ... An.

Цей вираз можна обчислити, використовуючи в якості підпрограми стандартний алгоритм множення пар матриць. Але для цього спочатку необхідно розставити дужки, щоб все неоднозначності в порядку перемноження. Порядок добутку матриць повністю визначений дужками, якщо добуток є або окремої матрицею, або взятим в дужки добуток двох послідовностей матриць, в якому порядок перемноження повністю визначений дужками. Матричне множення має властивість асоціативності, тому результат не залежить від розстановки дужок. Наприклад, якщо задана послідовність матриць (A1, A2, A3, A4) (рис. 2.1), то спосіб обчислення їх добутку можна повністю визначити за допомогою дужок п'ятьма різними способами:

(A1 (A2 (A3 A4))),
(A1 ((A2 A3) A4)),
((A1 A2) (A3 A4)),
((A1 (A2A3)) A4),

Рисунок 2.1 – Послідовність матриць

Від того як розставлені дужки при перемножуванні послідовності матриць, може сильно залежати час, витрачений на обчислення добутку. Спочатку розглянемо, як визначити вартість добутку двох матриць. Нижче наводиться

псевдокод стандартного алгоритму (рис. 2.2). Атрибути rows і columns означають кількість рядків і стовпців матриці:

```
Matrix_Multiply (A, B)
if columns [A] ≠ rows [B]
then error "Несумісні розміри"
else for i ← 1 to rows [A]
do for j ← 1 to columns [B]
do C [i, j] ← 0
for k ← 1 to columns [A]
do C [i, j] ← C [i, j] + A [i, k] • B [k, j]
return C
```

Рисунок 2.2 – Алгоритм добутку двох матриць

Матриці A і B можна множити, тільки якщо вони сумісні: кількість стовпців матриці A має збігатися з кількістю рядків матриці B. Якщо A – це матриця розміру p * q, а B - матриця розміру q * r, то в результаті їх перемноження вийде матриця З розміру p * Час обчислення матриці C переважно визначається кількістю скалярних добутків, яке виконується в рядку 7. Ця кількість одно pqr. Отже, вартість множення матриць буде виражатися в кількості множень скалярних величин.

Щоб проілюструвати, як розстановка дужок при перемножуванні декількох матриць впливає на кількість виконуваних операцій, розглянемо приклад, в якому перемножуються три матриці (A1, A2, A3). Припустимо, що розміри цих матриць рівні 10 * 100, 100 * 5 і 5 * 50 відповідно. Перемноживши матриці в порядку, заданому виразом ((A1A2) A3), необхідно виконати 10 • 100 • 5 = 5 000 скалярних множень, щоб знайти результат добутку A1A2 (при цьому вийде матриця розміром 10 * 5), а потім - ще 10 • 5 • 50 = 2 500 скалярних множень, щоб помножити цю матрицю на матрицю A3. Всього виходить 7 500 скалярних множень. Якщо

обчислювати результат в порядку, заданому виразом (A1 (A2A3)), то спочатку знадобиться виконати 100 • 5 • 50 = 25 000 скалярних множень (при цьому буде знайдена матриця A2 A3 розміром 100 * 50), а потім ще 10 • 100 • 50 = 50 000 скалярних множень, щоб помножити A1на цю матрицю. Всього виходить 75 000 скалярних множень. Таким чином, для обчислення результату першим способом знадобиться в 10 разів менше часу.

Завдання про перемножуванні послідовності матриць (matrix-chain multiplication problem) можна сформулювати так: для заданої послідовності матриць (A1, A2, ..., An), в якій матриця Ai, i = 1,2, ..., n має розмір pi-1 * pi, за допомогою дужок слід повністю визначити порядок множень в матричному добутку A1A2 ... An, при якому кількість скалярних множень зведеться до мінімуму.

Зверніть увагу, що саме перемножування матриць в задачу не входить. Наша мета – визначити оптимальний порядок множення. Зазвичай час, витрачений на перебування оптимального способу множення матриць, з лишком окупається, коли виконується саме множення (як це було в розглянутому прикладі, коли вдалося обійтися всього 7 500 скалярними множеннями замість 75 000).

Перший етап: структура оптимальної розстановки дужок.

Перший етап застосування парадигми динамічного програмування - знайти оптимальну допоміжну підструктуру, а потім з її допомогою сконструювати оптимальне рішення задачі по оптимальних рішень допоміжних завдань. Оптимальна допоміжна підструктура для даного завдання. Припустимо, що в результаті оптимальної розстановки дужок послідовність AiAi + 1 ... Aj розбивається на підпослідовності між матрицями Ak і Ak + 1.Тогда розстановка дужок в "префіксной" підпослідовності AiAi + 1 ... Ak теж повинна бути оптимальною. Тому що якби існував більш економний спосіб розстановки дужок в послідовності AiAi + 1 ... Aj, то його застосування дозволило б перемножити матриці AiAi + 1 ... Aj ще ефективніше, що суперечить припущенню про оптимальність початкової розстановки дужок. Аналогічно можна прийти до висновку, що розстановка дужок в підпослідовності матриць Ak + 1Ak + 1 ... Aj, що

виникає в результаті оптимальної розстановки дужок в послідовності AiAi + 1 ... Aj, також повинна бути оптимальною.

Тепер за допомогою нашої оптимальної допоміжної підструктури покажемо, що оптимальне рішення задачі можна скласти з оптимальних рішень допоміжних завдань. Ми вже переконалися, що для вирішення будь-якої нетривіальної задачі про оптимальний добуток послідовності матриць всю послідовність необхідно розбити на підпослідовності і що кожне оптимальне рішення містить в собі оптимальні рішення підзадач. Іншими словами, рішення повної завдання про оптимальному перемножуванні послідовності матриць можна побудувати шляхом розбиття цього завдання на дві підзадачі - оптимальну розстановку дужок в підпослідовність AiAi + 1 ... Ak і Ak + 1Ak + 2 ... Aj. Після цього знаходяться оптимальні рішення підзадач, з яких потім складається оптимальне рішення повною завдання. Необхідно переконатися, що при пошуку способу множення матриць враховуються всі можливі розбиття - тільки в цьому випадку можна бути впевненим, що знайдене рішення буде глобально оптимальним.

Другий етап: рекурсивне рішення.

Далі, рекурсивно визначимо вартість оптимального рішення в термінах оптимальних рішень допоміжних завдань. У задачі про перемножуванні послідовності матриць в якості допоміжної завдання вибирається завдання про оптимальну розстановку дужок в підпослідовності Ai Ai + i ... Aj при $1 \le i \le j \le n$.

Шлях m [i, j] - мінімальна кількість скалярних множень, необхідних для обчислення матриці Ai..j. Тоді в повній завданню мінімальна вартість матриці A1..n дорівнює m [1, n].

Рекурсивне визначення величини m [i, j] відбувається наступним чином. Якщо i = j, то завдання стає очевидною: послідовність складається всього з однієї матриці Ai..j = Aj, і для обчислення добутку матриць не потрібно виконувати жодних скалярних множень. Таким чином, при i = 1,2, ..., n m [i, i] = 0. Щоб обчислити m [i, j] при i <j, скористаємося властивістю підструктури оптимального рішення, дослідженим на першому етапі. Припустимо, що в результаті оптимальної

розстановки дужок послідовність Ai Ai + i ... Aj розбивається між матрицями Ak і Ak + 1, де i ≤ k <j.

Тоді величина m [i, j] дорівнює мінімальній вартості обчислення приватних добутків Ai..k і Ak + 1..j плюс вартість множення цих матриць один на одного. Якщо згадати, що кожна матриця A і має розміри pi-1 * pi, то неважко зрозуміти, що для обчислення добутку матриць Ai..kAk + 1..j знадобиться pi-1pkpj скалярних множень у формулі 2.1. Таким чином, отримуємо:

$$m [i, j] = m [i, до] + m [до + i, j] + pi\text{-}1pkpj , \qquad (2.1)$$

У цьому рекурсивному рівнянні передбачається, що значення до відомо, але насправді це не так. Для вибору цього значення за все мається j - i можливостей, а саме - k = i, i + 1, ..., j – 1 у формулі 2.2. Оскільки в оптимальній розстановці дужок необхідно використовувати одне з цих значень k, все, що потрібно зробити – перевірити всі можливості і вибрати серед них кращу. Таким чином, рекурсивне визначення оптимальної розстановки дужок в добутку Ai Ai + 1 ... Aj набирає вигляду:

$$m[i, j]=\begin{cases} 0 \text{ при } \&i=j \\ \min\limits_{i\le k<j}\{m[i, k]+m[k+1, j]+p_{i-1}p_kp_j\} \text{ при } i<j , \end{cases} \qquad (2.2)$$

Величини m [i, j] рівні вартостям оптимальних рішень потрібних допоміжних завдань. Щоб легше було простежити за процесом побудови оптимального рішення, позначимо через s [i, j] значення k, при якому послідовність AiAi + 1 ... Aj розбивається на дві підпослідовності в процесі оптимальної розстановки дужок у формулі 2.3. Таким чином, величина s [i, j] дорівнює значенню k, такому що:

$$m [i, j] = m [i, k] + m [k + 1, j] + pi\text{-}1pkpj , \qquad (2.3)$$

Третій етап: обчислення оптимальної вартості.

Важливе спостереження, яке можна зробити на даному етапі, полягає в тому, що у нас відносно мало підзадач: по одній для кожного вибору величин i і j, що задовольняють нерівності $1 \le i \le j \le n$, тобто всього $(n|2) + n = \theta (n2)$. У рекурсивном алгоритмі кожна допоміжна задача може неодноразово зустрічатися в різних гілках рекурсивного дерева.

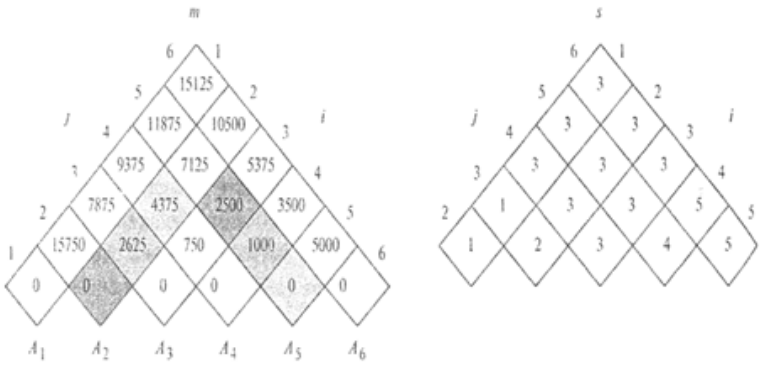
Така властивість перекриття допоміжних підзадач – друга відмінна риса застосовності методу динамічного програмування.

Замість вирішення рекуррентного співвідношення виконаємо третій етап парадигми динамічного програмування і обчислимо оптимальну вартість шляхом побудови таблиці в висхідному напрямку. В описаній нижче процедурі передбачається, що розміри матриць A і рівні pi-1 * pi (i = 1,2, ..., n). Вхідні дані представляють собою послідовність p = (p0, p1, ..., pn) довжина даної послідовності дорівнює length [p] - n + 1. У процедурі використовується допоміжна таблиця m [l..n, l..n] для зберігання вартостей m [i, j] і допоміжна таблиця s [l..n, l..n], в яку заносяться індекси k, при яких досягаються оптимальні вартості m [i, j]. Таблиця s буде використовуватися при побудові оптимального рішення.

Щоб коректно реалізувати вихідний підхід, необхідно визначити, за допомогою яких записів таблиці будуть обчислюватися величини m [i, j]. З рівняння видно, що вартість m [i, j] обчислення добутку послідовності j - i + 1 матриць залежить тільки від вартості обчислення послідовностей матриць, що містять менш j - i + 1 матриць. Іншими словами, при k = i, i + 1, ..., j -1 матриця Ai..k є добуток k - i + 1 <j - i + 1 матриць, а матриця Ak + 1..j - твір j - k <j - i + 1 матриць (рис. 2.3), (рис. 2.4). Таким чином, в ході виконання алгоритму слід організувати заповнення таблиці m в порядку, відповідному рішенню завдання про розстановку дужок в послідовності матриць зростаючої довжини:

```
Matrix_Chain_Order(p)
n ← length[p] — 1
for i ←1 to n
do m[i,i] ← O
for l ← 2 to n
do for i ← l to n – l + 1
do j ←i + l - 1
m[i, j] ← ∞
for k ← i to j – 1
do q ← m[i, k] + m [k +1, j] + pi-1pkpj
if q < m[i, j]
then m[i, j] ← q
s [i, j] ←k
return m и s .
```

Рисунок 2.3 – Процедура Matrix_Chain_Order для n = 6.



Матриця	Розмірність
A_1	30 × 35
A_2	35 × 15
A_3	15 × 5
A_4	5 × 10
A_5	10 × 20
A_6	20 × 25

Рисунок 2.4 – Таблиці m і s, обчислені процедурою Matrix_Chain_Order для n = 6.

На рисунку 2.4 описаний вище процес проілюстрований для ланцюжка, що складається з n = 6 матриць, розміри яких дорівнюють: A1- 30 * 35, A2 - 35 * 15, A3 - 15 * 5,A4 - 5 * 10, A5 - 10 * 20, A6 - 20 * 25. Оскільки величини m [i, j] визначені тільки при i ≤ j, використовується тільки частина таблиці m, розташована над її головною діагоналлю. Те ж можна сказати і про таблицю s. На рисунку 2.2 таблиця повернута так, щоб її головна діагональ була розташована горизонтально. У нижній частині малюнка наведено список матриць, що входять в послідовність. На цій схемі легко знайти мінімальну вартість m [i, j] перемноження часткової послідовності матриць AiAi + 1 ... Aj. Вона знаходиться на перетині ліній, що йдуть від матриці A і вправо і вгору, і від матриці Aj - вліво і вгору.

У кожній горизонтальній рядку таблиці містяться вартості перемноження приватних послідовностей, що складаються з однакової кількості матриць. У процедурі Matrix_Chain_Order рядки обчислюються від низу до верху, а елементи в кожному рядку - зліва направо. Величина m [i, j] обчислюється за допомогою добутків pi-1pkpj для k = i, i + 1, ..., j - 1 та величин внизу зліва і внизу праворуч від m [i, j]. З таблиці m видно, що мінімальна кількість скалярних множень, необхідних для обчислення добутку шести матриць, так само m [1,6] = 15 125. Для пояснення наведемо приклад. При обчисленні елемента m [2,5] у формулі 2.4 використовувалися пари елементів, що йдуть від матриць A2 і A5 і мають однаковий відтінок фону:

$$\begin{cases} m[2,5] = \min \\ m[2,2]+m[3,5]+p_1p_2p_5=0+2500+35*15*20=13000, \\ m[2,3]+m[4,5]+p_1p_3p_5=2625+1000+35*5*20=7125, \\ m[2,4]+m[5,5]+p_1p_4p_5=4375+0+35*10*20=11375 \\ =7125. \end{cases} \quad (2.4)$$

Нескладний аналіз структури вкладених циклів в процедурі Matrix_Chain_Order показує, що час її роботи становить O (n3). Глибина вкладення циклів дорівнює трьом, а індекси в кожному з них (l, i і k) приймають не більше n -

1 значень. Таким чином, процедура Matrix_Chain_Order набагато ефективніше, ніж метод перебору і перевірки всіляких способів розстановки дужок, час роботи якого експоненціально залежить від кількості перемножуєте матриць.

Четвертий етап: конструювання оптимального рішення.

Незважаючи на те, що в процедурі Matrix_Chain_Order визначається оптимальна кількість скалярних множень, необхідних для обчислення добутку послідовності матриць, в ньому не показано, як саме перемножуються матриці. Оптимальне рішення нескладно побудувати за допомогою інформації, що зберігається в таблиці s. У кожному елементі s [i, j] зберігається значення індексу k, де при оптимальній розстановці дужок в послідовності AiAi + 1 ... Aj виконується розбиття. Таким чином, нам відомо, що оптимальне обчислення добутку матриць A1..n виглядає як A1..s [1, n] As [1, n] + 1..n.Ці приватні добутки матриць можна обчислити рекурсивно, оскільки елемент s [1, з [1, п]] визначає матричне множення, яке виконує останнім при обчисленні A1..s [1, n], as [s [1, n] + 1, n] – останнім множенням при обчисленні As [1, n] + 1..n. Наведена нижче рекурсивна процедура виводить оптимальний спосіб розстановки дужок в послідовності матриць (Ai, Ai + 1, ..., Aj), по таблиці s, отриманої в результаті роботи процедури Matrix_Chain_Order, і індексам i i j (рис. 2.5). В результаті виклику процедури Print_Optimal_Parens (s, l, n) виводиться оптимальна розстановка дужок у послідовності (A1, A2, ..., An):

```
Print_Optimal_Parens (s, i, j)
If i = j
then print "A" i
else print "("
Print_Optimal_Parens (s, i, s [i, j})
Print_Optimal_Parens (s, s [i, j] + 1, j)
print ")"
```

Рисунок 2.5 – Оптимальна розстановка дужок у послідовності

У прикладі, проілюстрованому на рисунку 2.4 в результаті виклику процедури Print_Optimal_Parens (s, 1,6) виводиться рядок ((A1 (A2A3)) ((A4A5) A6)).

Ще один невеличкий приклад (рис. 2.6), де показано множення матриць, формула 2.5, 2.6 та 2.7. Отже задача стоїть таким чином:

$$M = M_1 * M_2 * \ldots * M_n \tag{2.5}$$

A_j матриця r_{i-1}, рядки та стовпці: $r_{i-1} * r_i$.

$$A=B*C,\ a_{i,j}= \sum_{k=1}^{q} b_{i,k} * c_{k,j} \tag{2.6}$$

Для того, щоб перемножити p*q та q*r необхідні операції p*q*r.

$$M= M_1*M_2*M_3*M_4= M_1*\left(M_2*(M_3*M_4)\right)=\left(M_1*(M_2*M_3)\right)*M_4 \tag{2.7}$$

Рішення:

Візьмемо $m_{i,j}$ – мінімальна вартість розрахунку і формулу 2.8:

$$M_j*M_{i+1}*\ldots* M_j,\ 1 \le i \le j \le n \tag{2.8}$$

Далі записуємо формулу 2.9, виводимо множення, (2.10) та мінімальну вартість, (2.11):

$$m_{i,j}= \begin{Bmatrix} 0,\ i = j \\ \min_{i \le k < j}\left(m_{i,j}+m_{k,j}+r_{i-1}*r_k*r_j\right),\ i < j \end{Bmatrix} \tag{2.9}$$

Записуємо множення:

$$M_i*M_{i+1}*\ldots* M_j = (M_i*M_{i+1}*\ldots* M_k) * (M_{k+1}*M_{k+2}*\ldots* M_j) \tag{2.10}$$

Виводимо мінімальну вартість:

$m_{i,j}$ мінімальна вартість $M'= M_i*M_{i+1}*\ldots*M_k$

$m_{i,j}$ мінімальна вартість $M' = M_i * M_{i+1} * \ldots * M_j$

$r_{i-1} * r_k * r_j$ вартість $M' * M''$ (2.11)

```
for(i=1;i<=n;i++) m[i][i]=0;
for(l=1;l<=n-1;l++)
{
 for(i=1;i<=n-l;i++)
 {
  j=i+l;
  m[i][j]=mmin(i,j);
 }
}
```

```
int dpexpr()
{
 return(m[i][k] + m[k+1][j] + r[i-1] * r[k] * r[j]);
}

int mmin(int i, int j)
{
 for(k=i+1;k<j;k++) mi=min(mi,dpexpr(i,k,j));
 return(mi);
}
```

Рисунок 2.6 – Алгоритм множення

Приклад задачі множення матриць(табл. 2.1):

$M = M_1 * M_2 * M_3 * M_4$, r_j: 10, 20, 50, 1, 1000.

Таблиця 2.1– Множення матриць

$m_{1,1} = 0$	$m_{2,2} = 0$	$m_{3,3} = 0$	$m_{4,1} = 0$
$m_{1,2} = 10000$	$m_{2,3} = 1000$	$m_{3,4} = 5000$	-
$m_{1,3} = 1200$	$m_{2,4} = 3000$	-	-
$m_{1,4} = 2200$	-	-	-

Висновки до другого розділу

У другому розділі було розглянуто множення ланцюжка матриць. Були наведені приклади коду, формули. Розглянуто етапи рішень стосовно множення матриць.

3 СУКУПНІСТЬ ДИНАМІЧОГО ПРОГРАМУВАННЯ

3.1 Динамічне програмування за профілем

Динамічне програмування за профілем – це спосіб оптимізації перебору кількості варіантів за допомогою динамічного програмування, коли один з вимірів невелика.

До більшості олімпіадних завдань обмеження журі підбирає за принципом «якомога більше». Тобто щоб будь-які розумні реалізації правильного рішення проходили, а все інше – ні. Коли зустрічається завдання з маленькими обмеженнями, це означає, що або автор навмисно збиває Вас з правильного шляху, або дійсно це завдання вирішується якимось (оптимізованим) перебором. Динамічне програмування за профілем - одна з таких оптимізацій. Часто в таких завданнях справа відбувається на прямокутної таблиці, одна з розмірностей якої досить мала (не більше 10). Як і в звичайному динамічному програмуванні потрібно перевірити існування, порахувати кількість способів, вартість і т.д. Асимптотика алгоритму, заснованого на цій ідеї, є експоненціальною тільки по одній розмірності, а по другий - лінійна або навіть краще.

3.2 Завдання про замощення доміно

Дана таблиця m * n, потрібно знайти кількість способів повністю замостити її неперекриваючих кісточками доміно (прямокутниками 2 * 1 і 1 * 2). Нехай m, n $\leq$ 10. У процесі замощення кожна клітина таблиці буде мати одне з двох станів: покрита якийсь Доміно чи ні. Щоб запам'ятати стан клітин одного стовпчика, достатньо однієї змінної P типу integer. Покладемо i-й біт P рівним 1, якщо i-а зверху клітина даного стовпця зайнята, 0 - якщо вільна. Будемо говорити в такому випадку, що P - бітова карта нашого стовпчика.

Дамо визначення базової лінії і профілю.

Базовою лінією називатимемо вертикальну пряму, що проходить через вузли таблиці. Можна занумерувати всі базові лінії, по черзі зліва направо, починаючи з нуля, таким чином, базова лінія з номером i - це пряма, що відсікає перші i стовпців від всіх інших (якщо такі є). Через bi будемо позначати базову лінію з номером i (рис. 3.1).

Стовпці пронумеруємо так: зліва від bi буде стовпець з номером i. Стовпець з номером i знаходиться між bi-1 і bi:

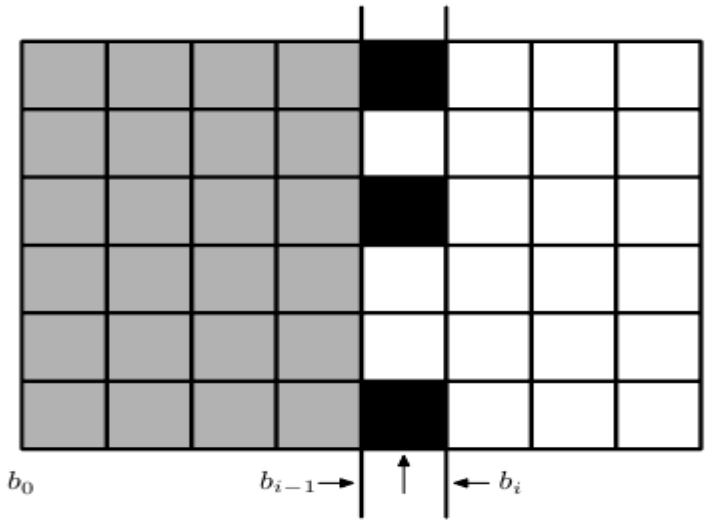


Рисунок 3.1 – Таблиця для задачі

Профіль буде таким: 1001012 = 1 + 4 + 32 = 37 (зайняті перша + третя + шоста клітини).

Профілем для базової лінії з номером i (bi) будемо називати бітову карту для стовпця з номером i при тому, що всі клітини зліва від bi-1 вже покриті, в i-м стовпці немає вертикальних доміно і вважається, що праворуч від bi немає покритих клітин.

Перша умова виникає від бажання вважати кількість способів поступово, спочатку розглядаючи перший стовпець, потім другий за умови, що перший вже заповнили і т.д. Друге і третє вводяться для того, щоб один і той же спосіб покриття ні пораховано більше одного разу. Точний зміст цих умов буде розкритий пізніше.

Для кожного профілю p1 базової лінії bi визначимо безліч профілів p2 базової лінії bi + 1 (рис. 3.2), які можуть бути з нього отримані. На таблицю, відповідну p1 можна класти доміно тільки двох типів:

- горизонтальні доміно, які перетинають bi (тобто діляться їй навпіл);
- вертикальні доміно, які лежать зліва від bi:

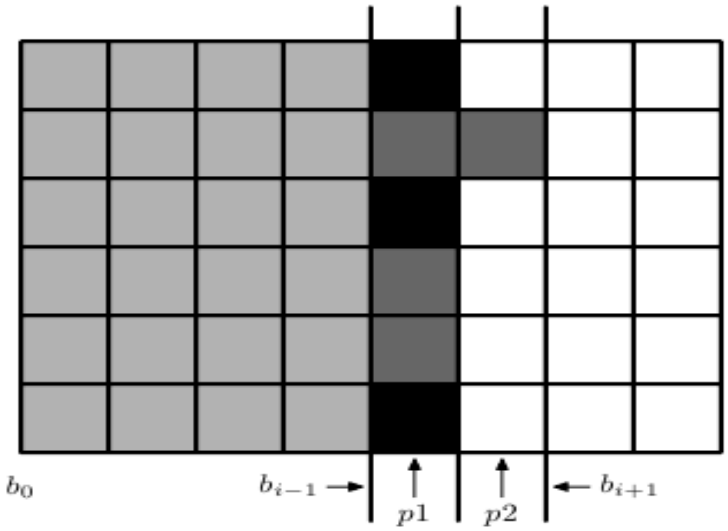


Рисунок 3.2 – Вигляд доміно в таблиці

p1 = 37, p2 = 2; неважко помітити по рисунку 3.2, що з p1 можна отримати p2.

Таким чином, d [37, 2] = 1.

Також нові доміно не повинні перекриватися один одним. Вони повинні покривати тільки незайняті клітини і кожна клітина i-го стовпця повинна бути покрита.

Бітова карта стовпця i + 1 і буде можливим профілем p2.

Очевидно, що отриманий таким чином профіль p2 дійсно задовольняє всім умовам, що накладається на профіль. Нехай d [p1, p2] - кількість способів з профілю p1 (для bi) отримати p2 (для bi + 1). Очевидно, для даної задачі про доміно це число може відрізнятися в основному тільки значеннями d [p1, p2]. Зауважимо, що всього профілів 2n: від 00 ... 02 = 0 до 11 ... 12 = 2n-1.Тому в даному випадку матриця D матиме розмір 2n * 2n.

Нехай тепер a [i, p] - кількість способів таким чином розташувати доміно, що p- профіль для bi (таким чином, всі клітини лівіше bi-1 покриті).

Початкові значення (i = 1):

a [1, 0] = 1;

a [1, p2] = 0, p2 = 1, ..., 2n-1.

Загальна формула (i > 1):

$$a[i, p_1] = \sum_{p_2=0}^{2^n-1} a[i-1, p_2\, d[t, p_1] \qquad (3.1)$$

Зауважимо, що для базової лінії номер 1 існує єдиний профіль (тобто бітова карта, що задовольняє умовам профілю) – карта незаповненого стовпчика.

Відповідь на питання завдання буде записаний в a [m + 1,0]. Помилкою було б вважати правильною відповіддю число a [m, 2n - 1], так як в цьому випадку не враховується можливість класти вертикальні доміно в останньому стовпчику.

Здавалося б, забутий ще один тип доміно, які можуть брати участь при формуванні нового профілю, а саме повністю лежать в стовпці i + 1. Справа в тому, що якщо дозволити їх, то деякі способи замощення будуть вважатися більш ніж один раз. Наприклад, нехай n = 2, m = 2. Тоді d '[0] [3] = 2, так як можна покласти або дві вертикальні доміно, або дві горизонтальні. Аналогічно, d '[3] [3] = 1 (можна покласти одну вертикальну). У підсумку маємо неправильну відповідь 3.

$$D' = \begin{pmatrix} 1 & 0\,0 & 2 \\ 0 & 0\,1 & 0 \\ 0 & 0\,0 & 1 \\ 1 & 0\,0 & 1 \end{pmatrix},\ A' = \begin{pmatrix} 1\,1\,3 \\ 0\,0\,0 \\ 0\,0\,0 \end{pmatrix}.$$

Якщо слідувати цьому правилу отримання з одного профілю іншого, то можна переконатися у вірності обчислень.

D і A при n = 2, m = 4:

$$D = \begin{pmatrix} 1 & 0\,0 & 1 \\ 0 & 0\,1 & 0 \\ 0 & 1\,0 & 0 \\ 1 & 0\,0 & 0 \end{pmatrix},\ A = \begin{pmatrix} 1 & 1\,2 & 3\,5 \\ 0 & 0\,0 & 0\,0 \\ 0 & 0\,0 & 0\,0 \\ 0 & 1\,1 & 2\,3 \end{pmatrix}$$

Таким чином, замостити Доміно таблицю 2 * 4 можна 5 способами, а таблицю 2 * 2 - двома. Якщо дивитися a [m, 2n - 1], то отримаємо 2 і 1. Очевидно, що таблицю 2 * 2 можна замостити двома, а не одним способом: пропущений варіант, коли

кладуть дві вертикальні доміно. У разі 2 * 4 пропущені три замощення - всі випадки, коли останній рядок покритий вертикальної Доміно.

Існує два способи для обчислення D. Перший полягає в тому, щоб для кожної пари профілів p1 і p2 перевіряти, чи можна з p1 отримати p2 описаним способом. Для всіх профілів p2, які при цьому виходили в наступному стовпці, між іншим d = [p1, p2] = 1. У більшості випадків цей спосіб більш економічний, так що логічно використовувати саме його.

У третьому розділі був розглянутий спосіб оптимізації перебору кількості варіантів за допомогою динамічного програмування, задачі про доміно та потрібні формули з поясненнями.

4 АЛГОРИТМИ, РІШЕННЯ ТА ПРИКЛАДИ ЗАДАЧ

4.1 Задача «Сходи»

Приклад завдання формулюється так: у маленького хлопчика є набір з різних N кубиків ($5 \leq N \leq 500$) (рис. 4.1).

З цих кубиків можна скласти не мало різні сходів. Сходи мають ступені різного розміру, які прямують у порядку зростання цього розміру, сходи не може мати дві однакові ступені. Кожна драбина повинна мати мінімум два ступені, і кожна ступінь повинен складатися мінімум з одного кубика. На малюнку наведено приклади сходів для N = 11 і N = 5:

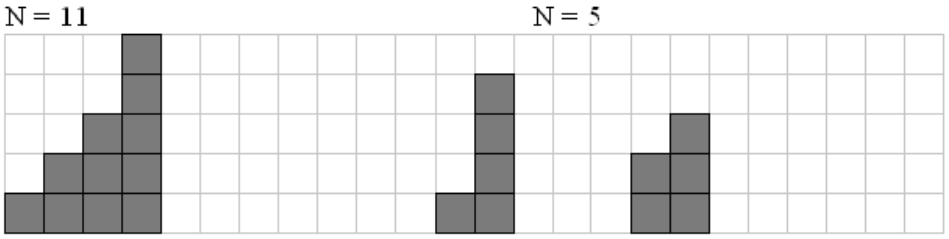


Рисунок 4.1 – Приклади сходів для N = 11 і N = 5

$$d[n][k]=\sum_{j=1}^{k-1} d[n-k][j] \quad (4.1)$$

n- кількість кубиків, k- висота сходів, j - висота k-1 сходинки. Якщо остання драбинка висоти k то передостання повинна бути висоти j де j від 1 до k - 1

Другий спосіб: d [n] [k] - кількість сходів з n кубиків в яких ширина дорівнює k якщо видалимо нижній шар сходи вийде сходи або тієї ж ширини (якщо перша сходинка була висоти більше ніж 1) або сходи шириною на одиницю менше (якщо перша сходинка була висоти 1).

$$d[n][k]= d[n-k][k-1] + d[n-k][k] \quad (4.2)$$

4.2 Алгоритм Нідлмана - Вунша

Алгоритм Нідлман - Вунша – це алгоритм, який використовується в біоінформатики для вирівнювання білкових або нуклеотидних послідовностей. Це було одне з перших додатків динамічного програмування для порівняння біологічних послідовностей. Алгоритм був розроблений Солом Б. Нідлманом і Крістіаном Д. Вунша і опублікований в 1970 році. Алгоритм по суті ділить більшу проблему (наприклад, повну послідовність) на серію дрібніших проблем і використовує рішення дрібніших проблем для пошуку оптимальне рішення більшої проблеми. Його також іноді називають алгоритмом оптимального відповідності та методом глобального вирівнювання. Алгоритм Нідлман - Вунша досі широко використовується для оптимального глобального вирівнювання, особливо коли якість глобального вирівнювання має першорядне значення. Алгоритм привласнює оцінку кожного можливого вирівнювання, і мета алгоритму - знайти всі можливі вирівнювання, що мають найвищий бал.

Приклад з молекулярної біології. Молекули ДНК, що містять генетичну інформацію - це довгі слова з чотирьох букв (А, Г, Ц, Т) (рис. 4.2), (рис. 4.3). В процесі еволюції, в результаті мутацій, послідовності змінюються, одна буква може замінитися на іншу, випасти, а може додатися нова. Наскільки схожі два фрагмента, яким найменшим числом перетворень можна один з них отримати з іншого? Формулювання завдання. Дано два слова (довжини M і N), що складаються з літер А, Г, Ц, Т. Знайти підпослідовність найбільшої довжини, що входить в те й інше слово:

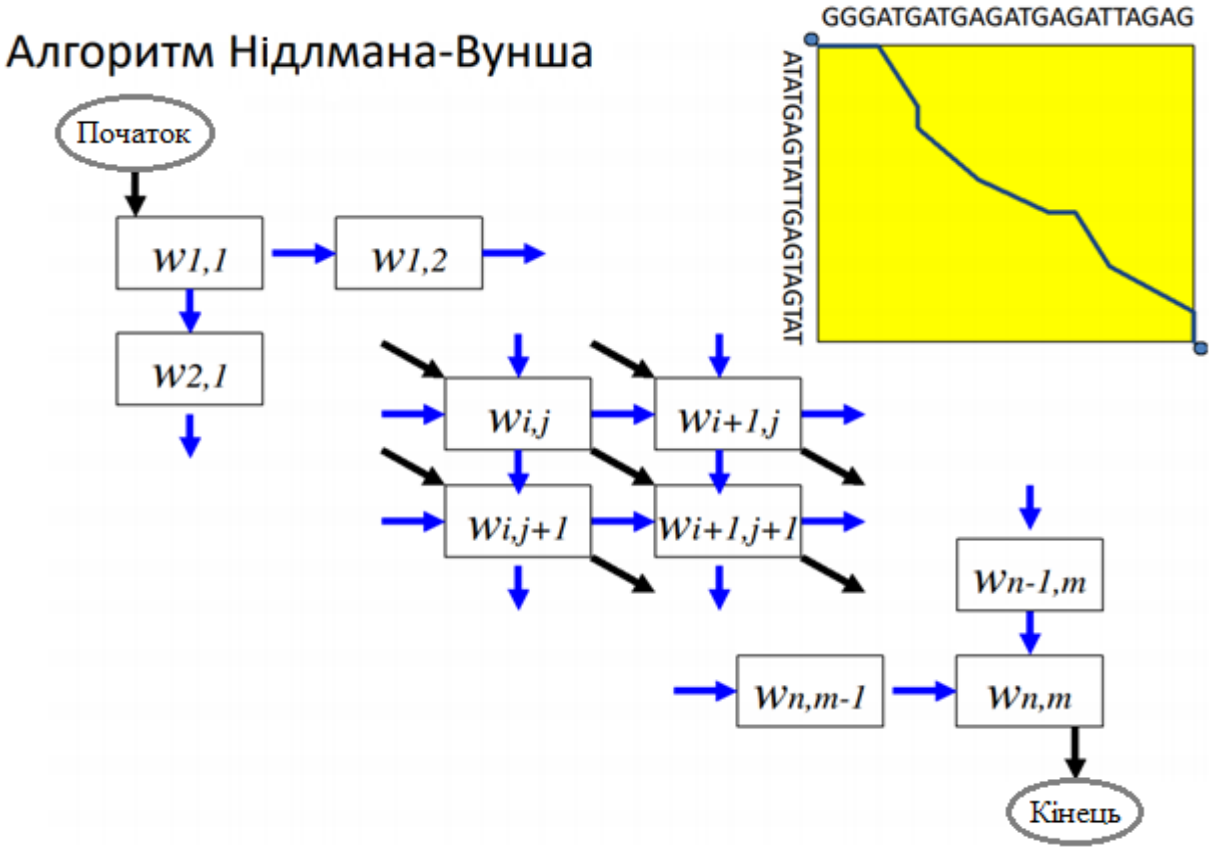


Рисунок 4.2 – Алгоритм Нідлмана-Вунша

Needleman-Wunsch

match = 1 mismatch = -1 gap = -1

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7
G	-1	1	0	-1	-2	-3	-4	-5
A	-2	0	0	1	0	-1	-2	-3
T	-3	-1	-1	0	2	1	0	-1
T	-4	-2	-2	-1	1	1	0	-1
A	-5	-3	-3	-1	0	0	0	-1
C	-6	-4	-2	-2	-1	-1	1	0
A	-7	-5	-3	-1	-2	-2	0	0

Рисунок 4.3 – Попарне вирівнювання послідовностей Нідлмана-Вунша

Побудова сітки

Спочатку побудуйте сітку, наприклад як показано на малюнку 1 вище. Почніть перший рядок у верхній частині третього стовпця і почніть другий рядок на початку третього рядка. Заповніть залишилися заголовки стовпців і рядків, як показано в табл. 4.1. У сітці ще не повинно бути чисел:

Таблиця 4.1 – Побудова сітки

	G	C	A	T	G	C	U
G							
A							
T							
T							
A							
C							
A							

Вибір системи підрахунку ваги(score)

Потім вирішите, як оцінювати кожну окрему пару букв. Використовуючи наведений вище приклад, одним з можливих кандидатів на вирівнювання може бути:

- 12345678;
- GCATG-CU;
- G-ATTACA.

Букви можуть збігатися, не збігатися або збігатися з пропуском (видалення або вставка (indel)):

- Відповідність: дві букви в поточному індексі збігаються;
- Невідповідність: дві букви в поточному індексі різні;
- Відступ (INsertion або DELetion): найкраще вирівнювання включає вирівнювання однієї літери з пропуском в іншому рядку.

Кожному з цих сценаріїв присвоюється бал, і сума балів всіх пар становить оцінка всього кандидата на вирівнювання. Існують різні системи для виставлення

оцінок; деякі з них описані в розділі Системи підрахунку очок нижче. На даний момент буде використовуватися система, яка використовується Needleman і Wunsch де є відповідність: +1 та невідповідність або Indel: -1;

Для наведеного вище прикладу оцінка вирівнювання буде дорівнює 0:

- GCATG-CU
- G-ATTACA
- + - ++ −− + - ->1 * 4 + (−1) * 4 = 0.

Заповнення таблиці

Почніть з нуля в другому рядку, другому стовпці. Переміщайтеся по осередках рядок за рядком, підраховуючи бали для кожного осередку. Оцінка розраховується шляхом порівняння оцінок осередків, сусідніх з лівої, верхній або верхній лівій (діагоналлю) осередки, і додавання відповідної оцінки за збіг, невідповідність або відступ. Обчисліть бали кандидатів для кожної з трьох можливостей: Шлях від верхньої або лівого вічка є пару з відступом, тому візьміть бали лівої і верхньої осередків і додайте бали для відступів до кожної з них. Діагональний шлях представляє збіг / невідповідність, тому візьміть оцінку лівої верхньої діагональної осередки і додайте оцінку збігу, якщо відповідні підстави (літери) в рядку і стовпці збігаються, або оцінку для невідповідність, якщо вони цього не роблять.

Підсумкова оцінка для осередку є найвищою з трьох оцінок кандидата. З огляду на, що для другого рядка немає «верхніх» або «верхніх лівих» осередків, тільки існуюча зліва осередок може використовуватися для розрахунку оцінки кожного осередку. Отже, -1 додається для кожного зсуву вправо, оскільки це є відступ від попередньої оцінки. Це призводить до того, що перший рядок дорівнює 0, -1, -2, -3, -4, -5, -6, -7 (табл. 4.2). Те ж саме відноситься до на одну, так як можна використовувати тільки існуючий рахунок над кожною клітинкою. Таким чином, підсумкова таблиця виглядає так:

Таблиця 4.2 – Додавання оцінок

		G	C	A	T	G	C	U
	0	-1	-2	-3	-4	-5	-6	-7

G	-1							
A	-2							
T	-3							
T	-4							
A	-5							
C	-6							
A	-7							

Перший випадок з існуючими оцінками у всіх трьох напрямках – це перетин наших перших букв (в даному випадку G і G) (табл. 4.3). Навколишні осередки знаходяться нижче:

Таблиця 4.3 – Перетини G і G

		G
	0	-1
G	-1	1

У таблиці 4.3 є три можливих суми кандидатів:

- Діагональний лівий верхній сусід має оцінку 0. З'єднання G і G є збігом, тому додайте оцінку для match: 0 + 1 = 1 (табл. 4.4);
- У верхнього сусіда оцінка -1, і перехід звідти є відступ, тому додайте оцінку для indel: (-1) + (-1) = (-2);
- Лівий сусід також має оцінку -1, являє собою відступ і також дає (-2).

Найвищий кандидат дорівнює 1 і вводиться в осередок:

Таблиця 4.4 – Оцінки з’єднань G і G

		G

Продовження таблиці 4.4

	0	-1
G	-1	1

У таблиці 4.4 місця, які дали найвищий бал кандидата, також повинні бути записані. На завершеною діаграмі на рисунку 1 вище це представлено у вигляді стрілки від осередку в рядку і стовпці 3 до осередку в рядку і стовпці 2.

У наступному прикладі крок по діагоналі для X і Y (табл. 4.5) є невідповідність:

Таблиця 4.5 – Діагоналі для X і Y

		G	C
	0	-1	-2
G	-1	1	X
A	-2	Y	

X:

- Верх: (-2) + (- 1) = (-3);
- Лівий: (+1) + (- 1) = (0);
- Вгорі зліва: (-1) + (- 1) = (-2).

Y:

- Вгорі: (1) + (- 1) = (0);
- Зліва: (-2) + (- 1) = (-3);
- Зліва вгорі: (-1) + (- 1) = (-2).

І для X, і для Y найвищий бал дорівнює нулю (табл. 4.6):

Таблиця 4.6 – Розрахунки балів для X та Y

		G	C
	0	-1	-2
G	-1	1	0
A	-2	0	

Найвищий бал кандидата може бути отриманий двома або всіма сусідніми осередками (табл. 4.7):

Таблиця 4.7 – Розрахунок найвищого балу кандидата

	T	G
T	1	1
A	0	X

- Верх: (1) + (- 1) = (0);
- Зліва вгорі: (1) + (- 1) = (0);
- Зліва: (0) + (- 1) = (-1).

В даному випадку, всі напрямки, які набрали найбільшу кількість балів кандидата, повинні бути відзначені як можливі вихідні осередки на готової діаграмі на рисунку 1, наприклад в осередку в рядку і стовпці 7. Заповнення таблиці таким чином дає оцінки або всіх можливих кандидатів на вирівнювання, оцінка в осередку в правому нижньому кутку представляє оцінку вирівнювання для кращого виравніванія.

Відстеження стрілок назад до вихідної точки

Відзначте шлях від осередку в правому нижньому кутку до осередку в верхньому лівому кутку, слідуючи напрямку стрілок. Цим шляхом послідовність будується за наступними правилами:

Діагональна стрілка представляє збіг чи невідповідність, тому буква стовпця і буква рядки вихідної осередки будуть вирівняні.

А горизонтальна або вертикальна стрілка позначає відступ. Горизонтальні стрілки вирівнюють пробіл ("-") з буквою рядка ("бічна" послідовність), вертикальні стрілки вирівнюють пробіл з буквою стовпця ("верхня" послідовність).

Якщо є кілька стрілок для вибору, вони представляють собою розгалуження трас. Якщо дві або більше гілки належать шляхах від нижнього правого до верхнього лівого кута комірки, вони однаково життєздатні. В цьому випадку відзначте шляху як окремі кандидати на вирівнювання. Дотримуючись цих правил, кроки для одного можливого кандидата на вирівнювання на рисунку 4.1.2 наступні:

U → CU → GCU → -GCU → T-GCU → AT-GCU → CAT-GCU → GCAT-GCU
A → CA → ACA → TACA → TTACA → ATTACA → -ATTACA → G-ATTACA ↓ (гілка) → TGCU → .. . → TACA → ...

Базові схеми підрахунку ваги(score)

У найпростіших схемах підрахунку очок просто вказується значення для кожного збігу, невідповідності і відступу. У наведеному вище покроковому

керівництві використовується match = 1, mismatch = -1, indel = -1. Таким чином, чим нижче оцінка вирівнювання, тим більше відстань редагування, для цієї системи оцінок потрібен високий бал. Інша система оцінки може бути коли відповідність = 0, Indel = 1 та невідповідність = 1.

Для цієї системи оцінка вирівнювання буде являти собою відстань редагування між двома рядками. Для різних ситуацій можуть бути розроблені різні системи підрахунку очок, наприклад, якщо прогалини вважаються дуже поганими для вашого вирівнювання, ви можете використовувати систему підрахунку очок, яка сильно карає прогалини, наприклад, коли Match = 0, Mismatch = 1 та Indel = 10.

Матриця подібності

Більш складні значення атрибутів систем підрахунку очок не тільки для типу зміни, але і для букв, які використовуються. Наприклад, збігом між A і A може бути присвоєно 1, але збігом між T і T може бути присвоєно 4. Тут (за умови використання першої системи підрахунку очок) більше значення надається збігом Ts, ніж As, тобто збігу Ts вважається більш важливим для вирівнювання. Це зважування на основі літер також застосовується до збігів, які не є ними(тобто незбігів).

Для подання всіх можливих комбінацій букв і їх результуючих оцінок використовується матриця подібності. Матриця подібності (табл. 4.8) для самої базової системи представлена як:

Таблиця 4.8 – Усі можливі комбінації та оцінки букв

	A	G	C	T
A	1	-1	-1	-1
G	-1	1	-1	-1
C	-1	1	1	-1
T	-1	-1	-1	1

Кожна оцінка являє собою перехід від однієї з букв, відповідних осередку, до іншої. Отже, тут представлені всі можливі збіги і видалення (для алфавіту ACGT). Зверніть увагу, що всі збіги йдуть по діагоналі, також не потрібно заповнювати всю таблицю, тільки цей трикутник, тому що оцінки взаємні. = (Оцінка для A → C =

Оцінка для C → A) (табл. 4.9). При реалізації наведеного вище правила TT = 4 створюється наступна матриця подібності:

Таблиця 4.9 – Оцінка для ACGT

	A	G	C	T
A	1	-1	-1	-1
G	-1	1	-1	-1
C	-1	1	1	-1
T	-1	-1	-1	4

Статистично побудовані різні оціночні матриці, які надають вагу різних дій, відповідним для конкретний сценарій. Наявність зважених оціночних матриць особливо важливо при вирівнюванні білкових послідовностей через різної частоти виникнення різних амінокислот. Існує два широких сімейства скорингових матриць, кожна з подальшими змінами для конкретних сценаріїв:

- PAM;
- BLOSUM.

Висновки до четвертого розділу

У четвертому розділі була розглянута невеличка задача «Сходи». Був більш детально розібраний алгоритм Нідлмана-Вунша. Були наведені приклади його алгоритму, пояснення з таблицями, побудови матриць.

5 ТЕСТУВАННЯ

5.1 Побудова діаграм

Відношення між акторами «Адміністратор» та «Юзер» (рис. 5.1):

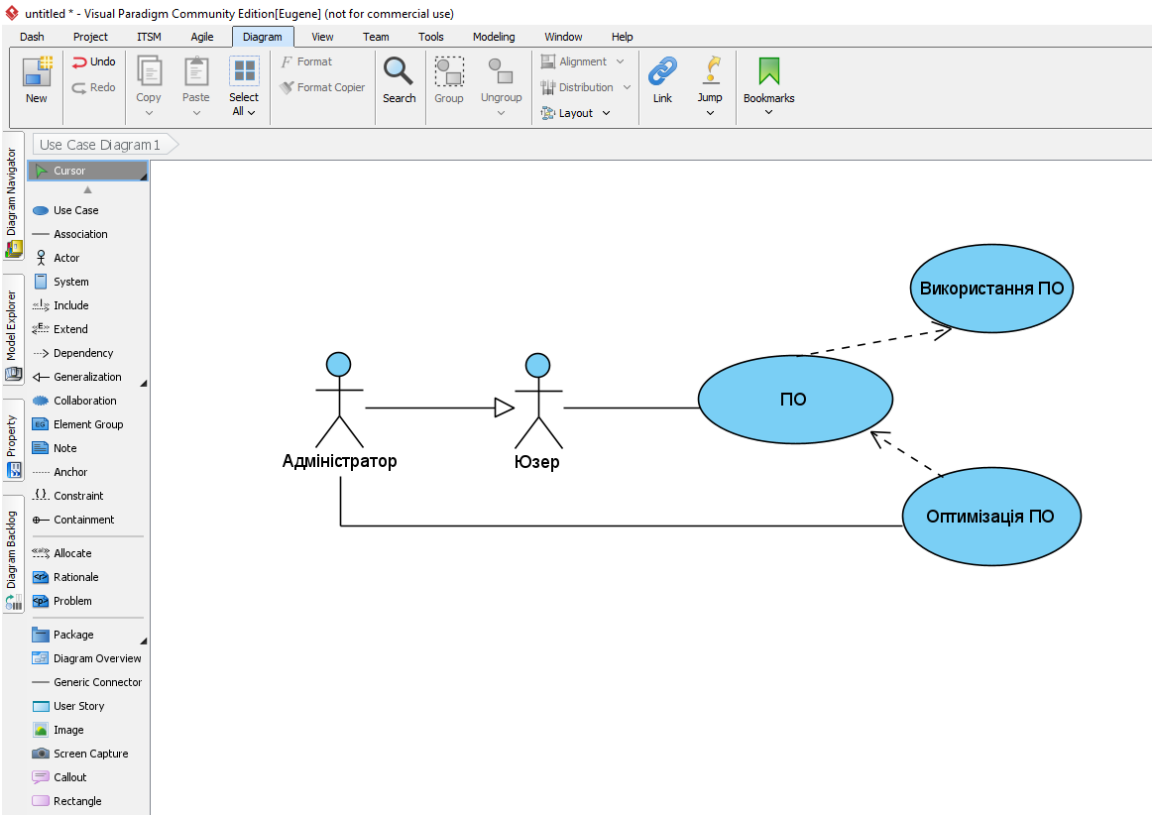


Рисунок 5.1 – Побудова UML-діаграми прецедентів

На рисунку 5.2 показано два класи: клас «Дані юзера» — це залежний клас-клієнт та клас «ПО» – є клас-постачальник у цій залежності:

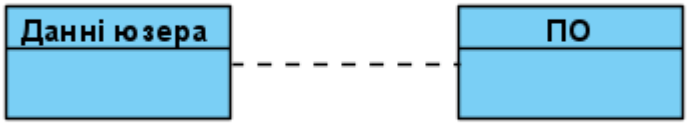


Рисунок 5.2 – Приклад відношення залежності

Відношення агрегації (рис. 5.3) має місце між декількома класами у тому випадку, якщо один з класів є таким, що включає в себе як складові частини деякі інші класи:

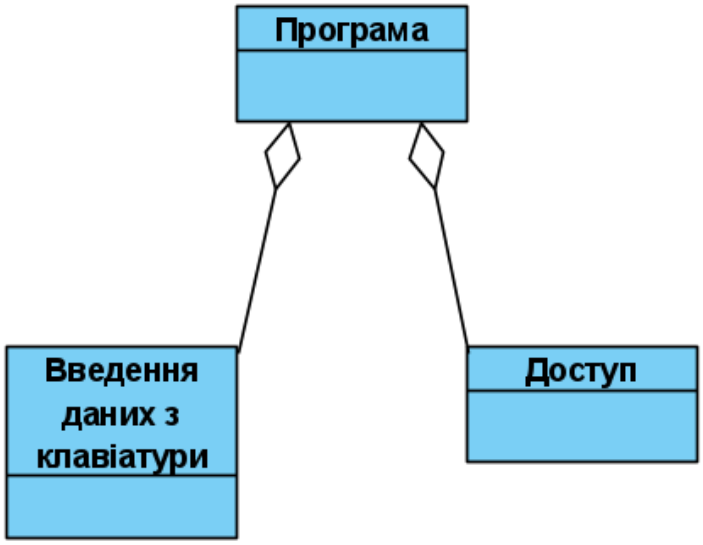


Рисунок 5.3 – Приклад діаграми класів для відношення агрегації

Графічно відношення композиції зображується суцільною лінією, один з кінців якої є зафарбованим усередині ромбом (рис. 5.4). Цей ромб вказує на той з класів, який є класом-композицією:

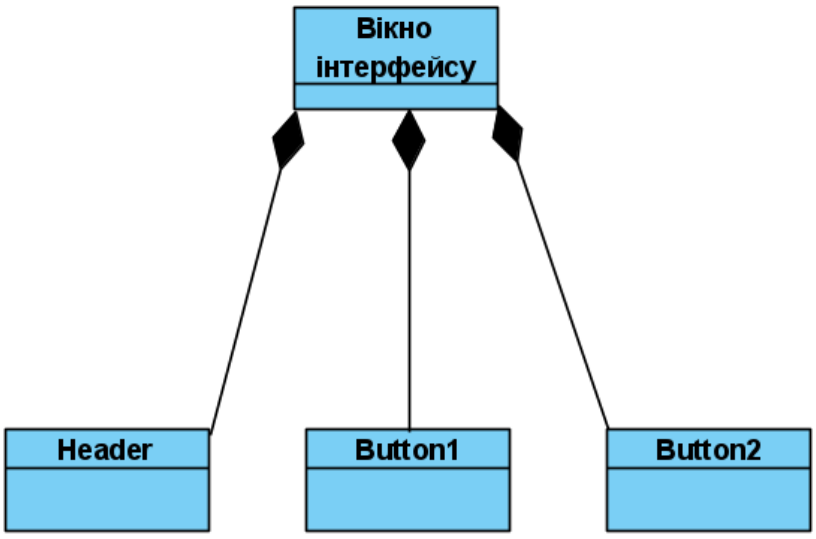


Рисунок 5.4 – Приклад діаграми класів для відношення композиції

Щодо діаграми класів відношення узагальнення (рис. 5.5), то вона описує ієрархічну структуру класів і вказує на спадкоємність їх властивостей і поведінки:

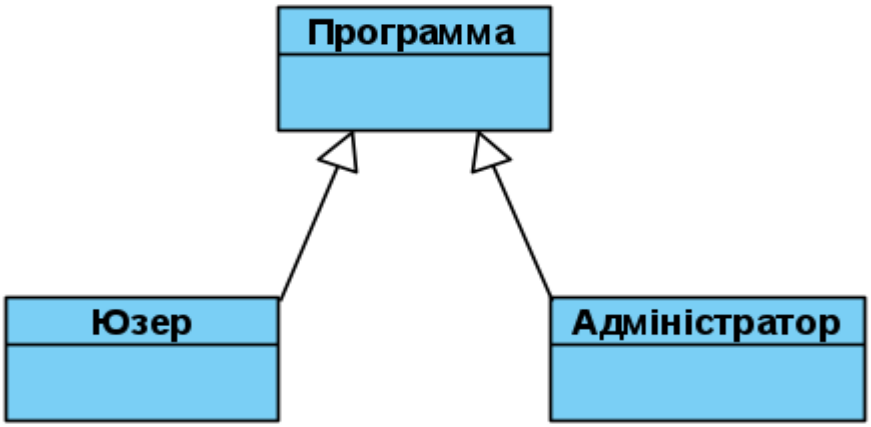


Рисунок 5.5 – Приклад діаграми класів для відношення узагальнення

Об’єкт є окремим представником певного класу, і кожен об’єкт має унікальну власну назву та конкретні значення своїх атрибутів. Невеличкий приклад з алгоритмом, який дозволяє вирішити задачу про перемножуванні ланцюжка матриць (рис. 5.6). Постанова завдання така: нехай є послідовність, що складається з n матриць, і потрібно обчислити множення A1A2 ... An:

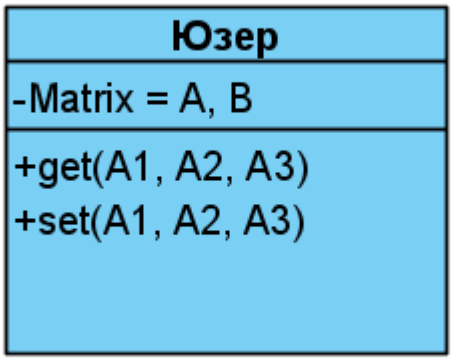


Рисунок 5.6 – Приклад зображення окремого об’єкта на діаграмі

На рисунку 5.7 показано початковий стан деякої «Програми» та перехід у стан «start», що відповідає появі на екрані комп’ютера потрібного діалогового вікна:

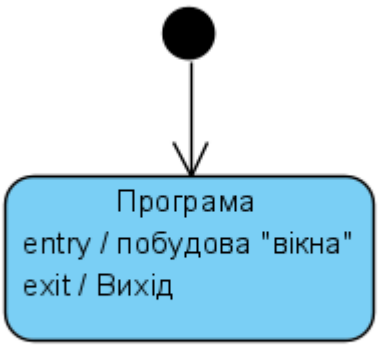


Рисунок 5.7 – Графічне зображення окремого стану з секцією опису внутрішніх дій

Діаграма станів (рис. 5.8) для прикладу розробки програмного забезпечення(ПО) процедури використання алгоритмів матриць:

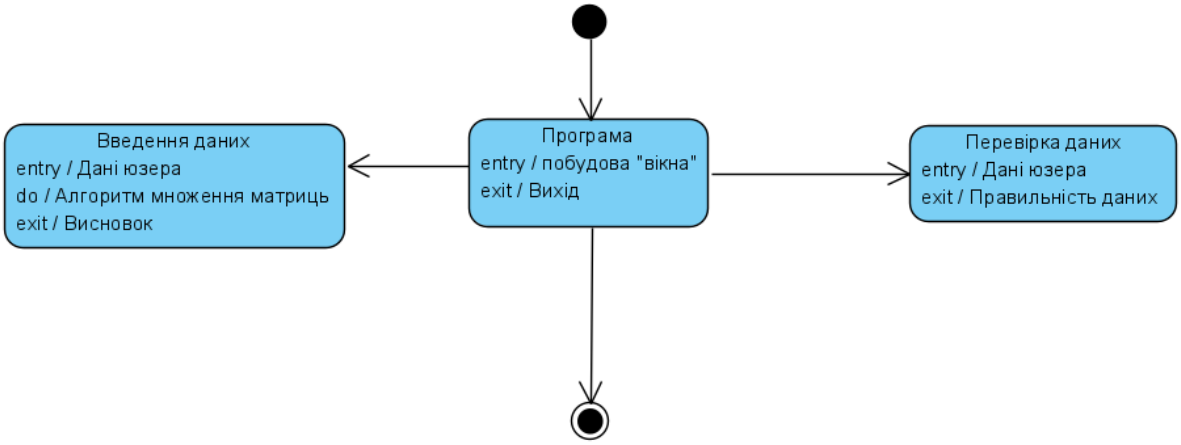


Рисунок 5.8 – Приклад побудови діаграми кінцевого автомата

Кожне перебування на діаграмі діяльності (рис. 5.9) відповідає виконанню деякої елементарної операції, а перехід у наступний стан спрацьовує лише при

завершенні цієї операції в попередньому стані. І для невеличкої наочності – діаграма комунікації (рис. 5.10):

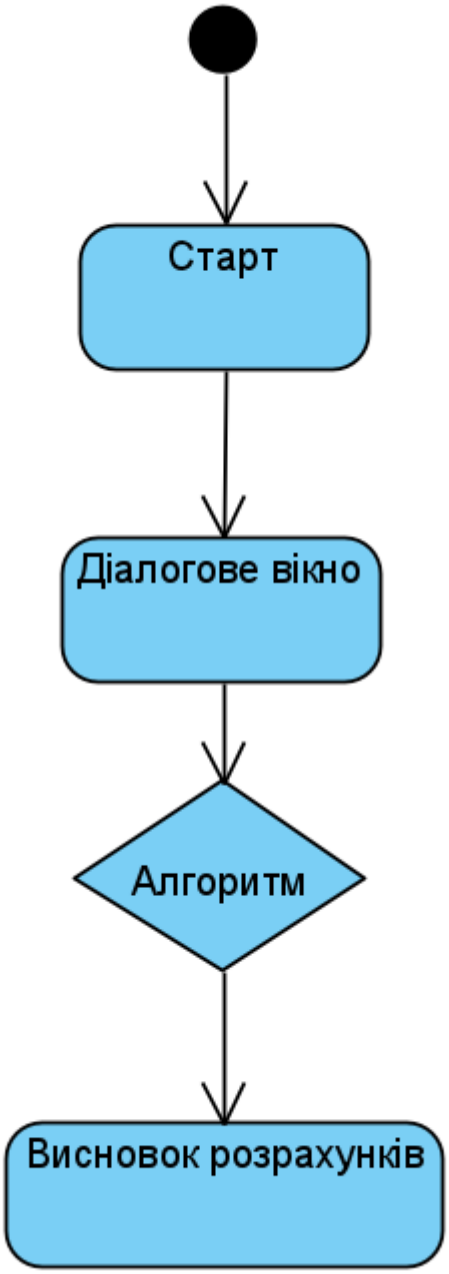


Рисунок 5.9 – Зображення основних графічних елементів діаграми дії(невеличке альтернативне розгалуження)

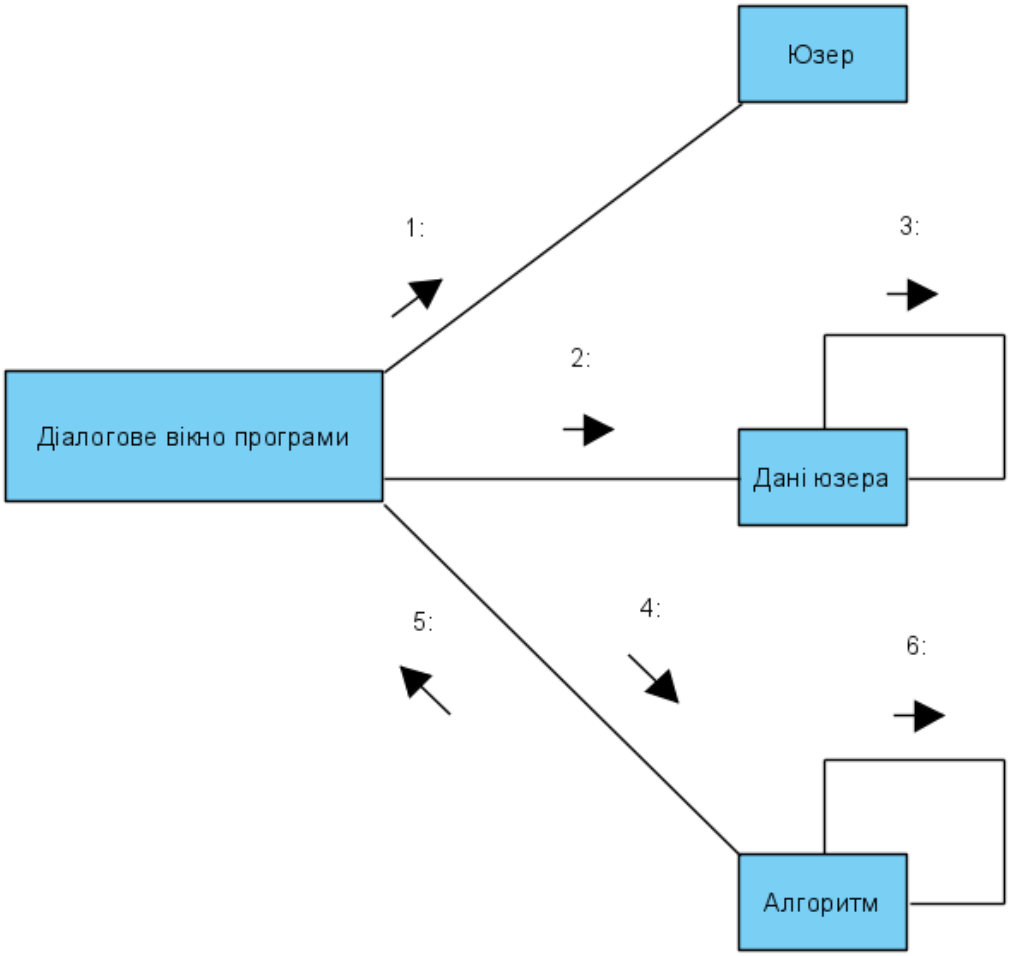


Рисунок 5.10 – Приклад побудови діаграми комунікації

5.2 Множення матриць

Приклад 1:

```
vector<_int64> a(n);
vector< vector<_int64> > p(n,a),q(n,a);
for(i=1;i<n;++i)
p[i][i]=1;
_int64 s;
for(i=3;i<n;++i)
for(j=1;j<i;++j)
{
s=0;
for(k=1;k<j;++k)
```

```
s+=p[i-j][k];
p[i][j]=s;
}
s=0;
for(j=1;j<n-1;++j)
s+=p[n-1][j];
fout<<"Q="<<s<<endl;
q[1][1]=1;
for(i=2;i<n;++i)
for(j=1;j<i;++j)
q[i][j]=q[i-j][j-1]+q[i-j][j];
s=0;
for(j=1;j*j<2*i;++j)
s+=q[n-1][j];
```

Лістинг 5.1 –Побудова матриці

Результати роботи програми:

N = 32

Q = 389

Q1 = 389

N – це значення яке вводиться з клавіатури Q, Q1 – результат роботи програми.

Приклад 2:

```
vector< vector<unsigned long long> > cache;
if (!*cached_value)
 *cached_value = compress(coins, n, i - 1);
 return *cached_value;
 }
vector<unsigned> coins;
 coins.push_back(1);
 coins.push_back(3);
 coins.push_back(5);
```

```
 coins.push_back(10);
 //coins.push_back(50);
unsigned n;
 cout<<"enter the amount- ";
 while (cin >> n)
 {
 unsigned long long solutions = solve_v3(n, coins);
if (solutions > 1)
 cout << "There are " << solutions << " ways to produce " << n << " cents change.\n";
 else
 cout << "There is only 1 way to produce " << n << " cents change.\n";
 }
return 0;
}
```

Лістинг 5.2 – Приклад задачі про «сдачу»

```
enter the amount-  3
There are 2 ways to produce 3 cents change.
5
There are 3 ways to produce 5 cents change.
10
There are 8 ways to produce 10 cents change.
```

Рисунок 5.11 – Результати роботи програми

Приклад 3

```
bool uspeshno_zamostil(unsigned cur_variant)
 {
 unsigned temp_cur_variant = cur_variant;
 kletki_clear();
for(int cur_cost_domino = 1; cur_cost_domino <= vsego_kostey_domino_;
 ++cur_cost_domino)
 {
 int variant_bin_dig = temp_cur_variant % 2;
 temp_cur_variant /= 2;
```

```
 const int VPRAVO = 0;
 const int VNIZ = 1;
kletki_[first_empty_kletka_i][first_empty_kletka_j] = cur_cost_domino;
 switch(variant_bin_dig)
 {
 case VPRAVO:
case VNIZ:
 {
 int& kletka_snizu_ot_pustoy
 = kletki_[first_empty_kletka_i + 1][first_empty_kletka_j];
 if(kletka_snizu_ot_pustoy != EMPTY_KLETKA) return false;
 kletka_snizu_ot_pustoy = cur_cost_domino;
 break;
 }
bool get_first_empty_kletka
```

Лістинг 5.3 – Приклад задачі про доміно

У п’ятому розділі були розглянуті побудови діаграм стосовно матриць. Були наведені приклади, лістинги коду.

ВИСНОВКИ ТА РЕКОМЕНДАЦІЇ

У магістерській роботі проведено дослідження можливостей підвищення ефективності матричних обчислень при вирішенні прикладних задач за допомогою динамічного програмування.

Результати досліджень, виконаних в роботі дозволили встановити, що:

1 Проведено аналіз умов, переваг та недоліків використання динамічного програмування. Показано, що метод динамічного програмування призначено для підвищення ефективності обчислень, при вирішенні задач шляхом їх декомпозиції на відносно прості підзадачі.

2 Матричні моделі реальних систем та процесів є одним з найпоширеніших представлень даних, які можна і потрібно складати і множити та вирішувати таким чином різні завдання.

3 Виконано порівняння кількості операцій множення матриць для різних послідовностей операцій множення. Отримані розрахунки демонструють перевагу той або іншої послідовності операцій множення в рази та десятки разів.

4 Використання метода динамічного програмування дозволяє отримувати не тільки зменшення кількості операцій, а також швидкий засіб формування запису послідовності множення матриць за рахунок рекурентних відносин.

Результати досліджень дозволяють рекомендувати використання динамічного програмування для оптимізації послідовності множення матриць при вирішенні задач в математиці, у фізиці, в техніці, в економіці, інших областях науки і знань.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1 Динамічне програмування / С. М. Окулов, О. А. Пестов. БИНОМ. Лабораторія знань, 2012.-296 с.

2 Динамічне програмування / Беллмана Р. Изд-во іноз. літ., 2020. 400 с.

3 Математичне програмування в прикладах і завданнях / Акуліч І.Л. 2021, 319 с.

4 Основи динамічного програмування / Габасит Р., Кирилова Ф. М. Изд-во БГУ, 2019. - 262 с.

5 Динамічне програмування / URL: https://tproger.ru/articles/dynprog-starters/

6 Рекурсія. Введення у програмування. Головні елементи / URL: https://ru.hexlet.io/courses/introduction_to_programming/lessons/recursion/theory_unit

7 Алгоритми у біоінформатиці. Вирівнювання послідовностей / URL: http://bioinformaticsinstitute.ru/sites/default/files/algorithms_in_bioinformatics.pdf

8 Алгоритм Нідлмана-Вунша / URL https://wikichi.ru/wiki/Needleman–Wunsch_algorithm

9 Динамічне програмування / URL: https://ua5.org/osnprog/1907 dynamichne-programuvannya.html

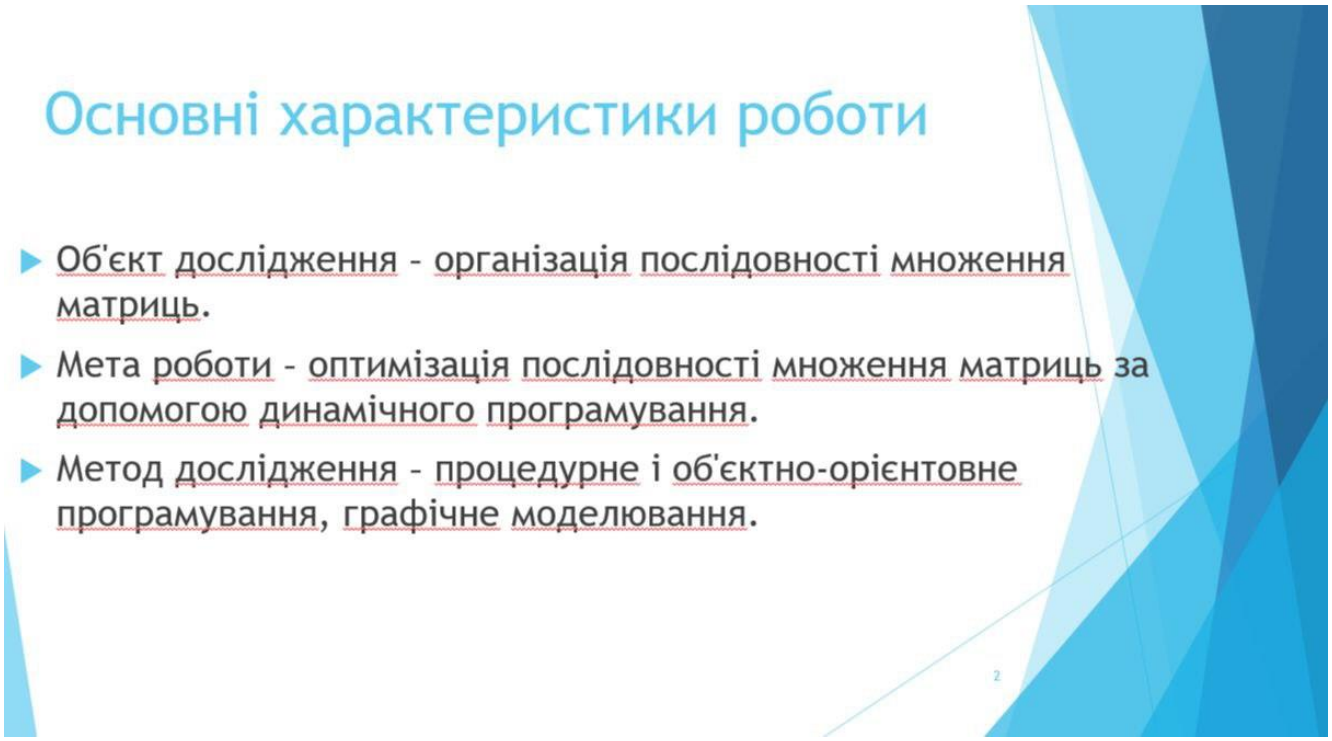
10 A. Lew, H. Mauch. Dynamic Programming: A Computational Tool. // Springer-Verlag Berlin Heidelberg. 2007. – 378 p.

11 Міца О.В., Оришич С.С., Дуло В.В. Аналіз ефективності динамічного підходу розв'язання задачі комівояжера. / XI Міжнародна наукова Інтернет-конференція "Національна безпека у фокусі викликів глобалізаційних процесів в економіці". 7-8 грудня 2021 року, Ukraine-Slovakia. С. 20-22.

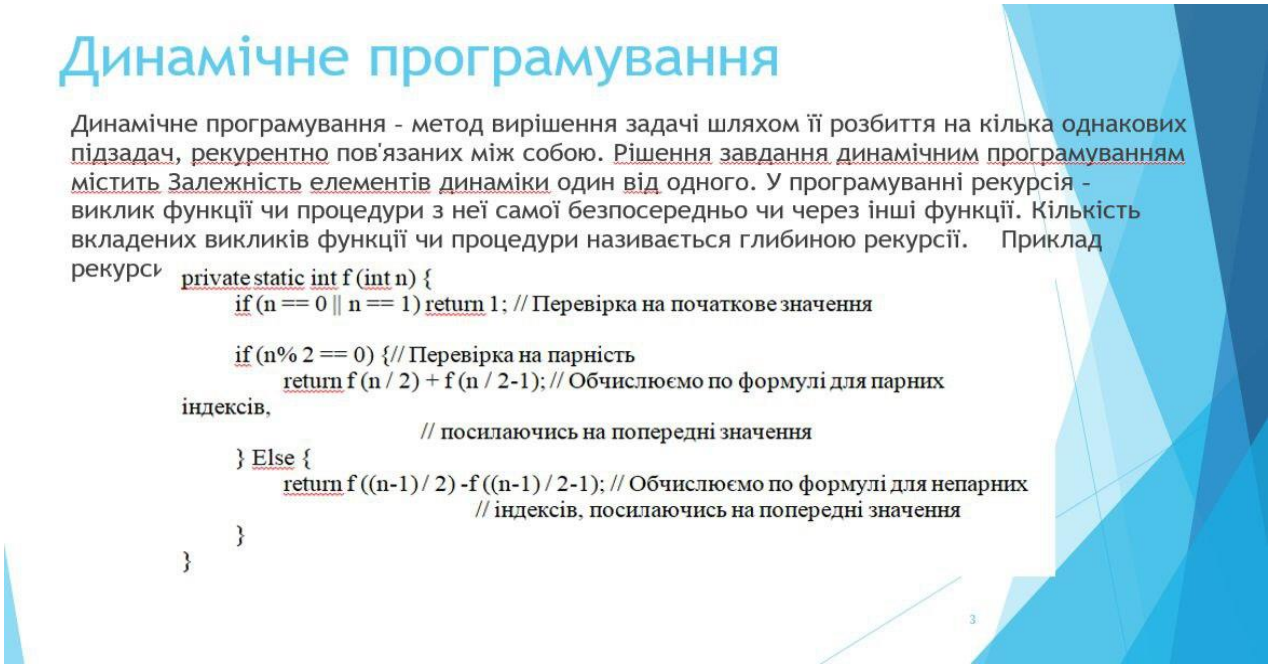
ДОДАТОК А

ПЕРЕЛІК КОПІЙ ДЕМОНСТРАЦІЙНОГО МАТЕРІАЛУ

Слайд 1 – ОСНОВНІ ХАРАКТЕРИСТИКИ РОБОТИ



Слайд 2 – РІШЕННЯ ЗАВДАННЯ ДИНАМІЧНИМ ПРОГРАМУВАННЯМ



Слайд 3 – РІШЕННЯ ЗАВДАННЯ ДИНАМІЧНИМ ПРОГРАМУВАННЯМ

Рішення завдання динамічним програмуванням

- Рішення завдання динамічним програмуванням повинно містити:

1) Залежність елементів динаміки один від одного.

2) Значення початкових станів.

Слайд 4 – МНОЖЕННЯ ЛАНЦЮЖКА МАТРИЦЬ

Множення ланцюжка матриць

- Завдання про порядок множення матриць - класична задача динамічного програмування, в якій дана послідовність матриць {\ displaystyle A_ {1}, A_ {2}, \ dots, A_ {n}} і потрібно мінімізувати кількість скалярних операцій для обчислення їх добутку.

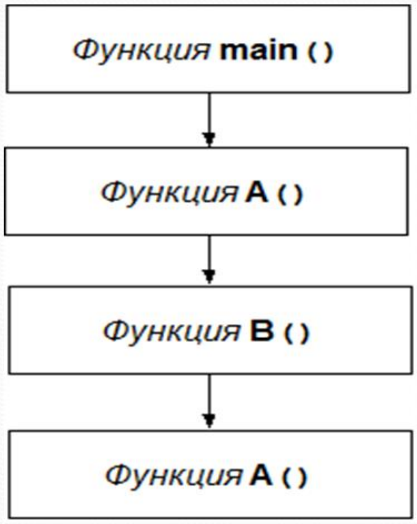
A1 • A2 • A3 • ... • An

(A1 (A2 (A3 A4))),
(A1 ((A2 A3) A4)),
((A1 A2) (A3 A4)),
((A1 (A2A3)) A4),

Слайд 5 – РЕКУРСИВНЕ РІШЕННЯ

Рекурсивне рішення

- У програмуванні рекурсія — виклик функції чи процедури з неї самої (звичайно з іншими значеннями вхідних параметрів) безпосередньо чи через інші функції (наприклад, функція A викликає функцію B, а функція B — функцію A). Кількість вкладених викликів функції чи процедури називається глибиною рекурсії.



Слайд 6 – ПРИКЛАД МНОЖЕННЯ

Приклад множення

- $M = M_1 * M_2 * \ldots * M_n$
- Для того, щоб перемножити p*q та q*r необхідні операції p*q*r:
- $M = M_1 * M_2 * M_3 * M_4 = M_1 * (M_2 * (M_3 * M_4)) = (M_1 * (M_2 * M_3)) * M_4$
- Рішення:
- Візьмемо $m_{i,j}$ – мінімальна вартість розрахунку і
- $M_j * M_{i+1} * \ldots * M_j$, $1 \le i \le j \le n$
- Записуємо множення:
- $M_i * M_{i+1} * \ldots * M_j = (M_i * M_{i+1} * \ldots * M_k) * (M_{k+1} * M_{k+2} * \ldots * M_j)$
- Виводимо мінімальну вартість:
- $m_{i,j}$ мінімальна вартість $M' = M_i * M_{i+1} * \ldots * M_k$
- $m_{i,j}$ мінімальна вартість $M' = M_i * M_{i+1} * \ldots * M_j$
- $r_{i-1} * r_k * r_j$ вартість $M' * M''$

ДОДАТОК Б

ЛІСТИНГ КОДУ

Приклад 1

```
#include<vector>
#include<iostream>
#include<fstream>
using namespace std;
int main(){
int i,j,n,k;
ifstream fin("input.txt");
ofstream fout("output.txt");
fout<<"N=";
fin>>n;
fout<<n<<endl;
++n;
vector<_int64> a(n);
vector< vector<_int64> > p(n,a),q(n,a);
for(i=1;i<n;++i)
p[i][i]=1;
_int64 s;
for(i=3;i<n;++i)
for(j=1;j<i;++j)
{
s=0;
for(k=1;k<j;++k)
s+=p[i-j][k];
p[i][j]=s;
```

```
}
s=0;
for(j=1;j<n-1;++j)
s+=p[n-1][j];
fout<<"Q="<<s<<endl;
q[1][1]=1;
for(i=2;i<n;++i)
for(j=1;j<i;++j)
q[i][j]=q[i-j][j-1]+q[i-j][j];
s=0;
for(j=1;j*j<2*i;++j)
s+=q[n-1][j];
fout<<"Q1="<<s-1<<endl;
system("pause");
return 0;
}
```

Приклад 3

```
#include <iostream>
#include <cmath>
#include <map>
#include <iomanip>
class T_doska
{
 typedef std::map<int, int> T_horizontal;
 typedef std::map<int, T_horizontal> T_kletki;
 static const int EMPTY_KLETKA = -1;
int dim_;
 int count_;
 double vsego_kostey_domino_;
```

```
 double variant_max_;
 T_kletki kletki_;
public:
 T_doska(int dim)
 : dim_(dim),
 count_(0),
 vsego_kostey_domino_(dim_ * dim_ / 2),
 variant_max_(pow(static_cast<double>(2), vsego_kostey_domino_))
 {}
void count_zamostit_domino()
 {
 for(unsigned cur_variant = 0; cur_variant < variant_max_; ++cur_variant)
 {
 if(uspeshno_zamostil(cur_variant))
 {
 ++count_;
 print_kletki();
 }
 }
 std::cout << "Всего получили "
 << count_
 << " вариантов замощения доски "
 << dim_
 << " x "
 << dim_
 << " костями домино."
 << std::endl;
 }
private:
void print_kletki()
```

```
{
std::cout << "Вариант № "
<< count_
<< std::endl;
for(int i = 0; i < dim_; ++i)
{
for(int j = 0; j < dim_; ++j)
{
if(kletki_[i][j] == EMPTY_KLETKA)
{
std::cout << std::setw(4)
<< "";
}
else
{
std::cout << std::setw(4)
<< kletki_[i][j];
}
}
std::cout << std::endl
<< std::endl;
}
}
bool uspeshno_zamostil(unsigned cur_variant)
{
unsigned temp_cur_variant = cur_variant;
kletki_clear();
for(int cur_cost_domino = 1; cur_cost_domino <= vsego_kostey_domino_;
++cur_cost_domino)
{
```

```
int variant_bin_dig = temp_cur_variant % 2;
temp_cur_variant /= 2;
const int VPRAVO = 0;
const int VNIZ = 1;
//Ищем первую пустую клетку доски.
int first_empty_kletka_i;
int first_empty_kletka_j;
if(!get_first_empty_kletka(first_empty_kletka_i, first_empty_kletka_j))
{
return false;
}
kletki_[first_empty_kletka_i][first_empty_kletka_j] = cur_cost_domino;
switch(variant_bin_dig)
{
case VPRAVO:
{
int& kletka_sprava_ot_pustoy
= kletki_[first_empty_kletka_i][first_empty_kletka_j + 1];
if(kletka_sprava_ot_pustoy != EMPTY_KLETKA) return false;
kletka_sprava_ot_pustoy = cur_cost_domino;
break;
}
case VNIZ:
{
int& kletka_snizu_ot_pustoy
= kletki_[first_empty_kletka_i + 1][first_empty_kletka_j];
if(kletka_snizu_ot_pustoy != EMPTY_KLETKA) return false;
kletka_snizu_ot_pustoy = cur_cost_domino;
break;
}
```

```
 }//switch(variant_bin_dig)
 }//for(int cur_cost_domino = 1; cur_cost_domino <= vsego_kostey_domino_;
 return true;
 }
bool get_first_empty_kletka
 (
 int& first_empty_kletka_i,
 int& first_empty_kletka_j
 )
 {
 for(int i = 0; i < dim_; ++i)
 {
 for(int j = 0; j < dim_; ++j)
 {
 if(kletki_[i][j] == EMPTY_KLETKA)
 {
 first_empty_kletka_i = i;
 first_empty_kletka_j = j;
 return true;
 }
 }
 }
 return false;
 }
void kletki_clear()
 {
 for(int i = 0; i < dim_; ++i)
 {
 for(int j = 0; j < dim_; ++j)
 {
```

```
 kletki_[i][j] = EMPTY_KLETKA;
 }
 }
 }
};
int main()
std::locale::global(std::locale(""));
 std::cout << "Введите длину стороны доски: ";
 int doska_dim;
 std::cin >> doska_dim;
 T_doska doska(doska_dim);
 doska.count_zamostit_domino();
system("pause");
 return 0;
```