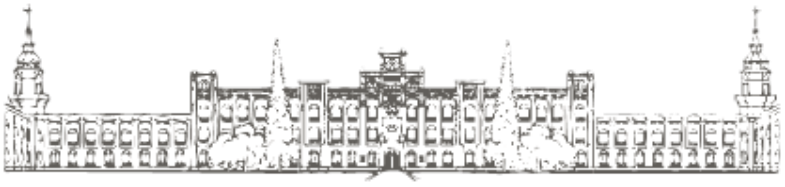


МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Курсова робота з освітнього компоненту
з дисципліни «Технології паралельних обчислень»
Тема: Алгоритм розв’язання задачі про пакування рюкзака методом динамічного програмування та його паралельна реалізація мовою програмування Java

Керівник:
Кандидат технічних наук,
старший викладач
Дифучин Антон Юрійович

«Допущено до захисту»
_____

«___» _____ 2023 р.

Захищено з оцінкою
_____

Члени комісії:

_____

_____

Виконавець:

Дмитрієва Ірина Ігорівна

студент групи ІТ-01

залікова книжка № ІТ-0107

_____

«23» травня 2023 р.

Інна СТЕЦЕНКО

Антон ДИФУЧИН

Київ – 2023

Міністерство освіти та науки України

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Кафедра Інформатики та Програмної Інженерії

Дисципліна Технології паралельних обчислень

Спеціальність Інженерія програмного забезпечення

Курс III Група ІТ-01 Семестр 6

ЗАВДАННЯ
на курсову роботу студента

Дмитрієвої Ірини Ігорівни

(прізвище, ім’я, по батькові)

1.Тема роботи Алгоритм розв’язання задачі про пакування рюкзака методом динамічного програмування та його паралельна реалізація мовою програмування Java

2.Термін здачі студентом закінченої роботи «23» травня 2023 р.

3.Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці): зміст, реферат, ключові слова, вступ, розділ 1 (опис алгоритму та його відомих паралельних реалізацій), розділ 2 (розробка послідовного алгоритму та аналіз його швидкодії), розділ 3 (вибір програмного забезпечення для розробки паралельних обчислень та його короткий опис), розділ 4 (розробка паралельних обчислень алгоритму з використанням обраного програмного забезпечення: проектування, реалізація, тестування), розділ 5 (дослідження ефективності паралельних обчислень алгоритму (порівняльний аналіз швидкості обчислень та дослідження впливу параметрів алгоритму на показники ефективності)), висновки, список використаних джерел, додатки (результати тестування послідовного та паралельного алгоритмів, лістинг коду, схема взаємодії паралельних процесів).

4. Дата видачі завдання «17» лютого 2022 р.

РЕФЕРАТ

Метою курсової роботи є проведення дослідження алгоритму розв’язання задачі про пакування рюкзака та його паралельна реалізація мовою Java для дослідження ефективності розпаралелізації процесів.

Задача пакування рюкзака (Knapsack problem) — задача комбінаторної оптимізації, мета якої визначити як укласти якомога більше цінних речей в рюкзак, за умови, що загальний обсяг (або вага) всіх предметів, здатних помістится в рюкзак, обмежений.

Інша, більш загальна назва задачі – задача завантаження.

Задача часто виникає при розподілі ресурсів, коли наявні фінансові обмеження, і вивчається в таких областях, як комбінаторика, інформатика, теорія складності, криптографія, прикладна математика.

Актуальність задачі пакування рюкзака (завантаження) полягає в тому, що ця задача та різні її модифікації широко застосовуються на практиці в прикладній математиці, криптографії, економіці, логістиці, при вирішенні великої кількості промислових задач:

- пошук оптимального завантаження різних транспортних засобів: літаків, човнів, автомобілів, залізничних вагонів (розмістити вантаж так, щоб зайняти якомога менше місця за площею, об’ємом, або обрати за вагою);
- пошуку варіантів оптимального розміщення товарів на складі (площа, об’єм);
- кроєння різних матеріалів (тканини, сталеві листи, плитні меблеві матеріали тощо): вибір оптимальної схеми розкрою матеріалів з метою зменшення кількості відходів.

Існуючі алгоритми, на практиці, здатні розв'язати задачі досить великих розмірів. Однак, унікальна структура задачі, а також той факт, що вона присутня як підзадача у більших, загальніших проблемах, робить її важливою для наукових досліджень.

Об’єктом дослідження курсової роботи є алгоритм розв’язання задачі про пакування рюкзака методом динамічного програмування та його паралельна реалізація.

У роботі буде розроблена реалізація послідовного та паралельного алгоритмів з використанням мови програмування Java, виконані експериментальні дослідження послідовних та паралельних обчислень алгоритму під час яких проведений порівняльний аналіз швидкості обчислень та дослідження впливу параметрів алгоритму на показники ефективності.

Ключові слова: ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ДИНАМІЧНЕ ПРОГРАМУВАННЯ, БАГАТОПОТОКОВІСТЬ, ПАКУВАННЯ РЮКЗАКА, АЛГОРИТМИ, МОВА ПРОГРАМУВАННЯ JAVA

ЗМІСТ

ВСТУП

У загальному плані під паралельними обчисленнями розуміються процеси обробки даних, в яких одночасно можуть виконуватися декілька операцій комп'ютерної системи.

У паралельних обчисленнях велика задача розбивається на менші підзадачі, кожна з яких розв'язується окремим процесором або ядром незалежно від інших (паралельно). Це дозволяє зменшити час виконання задачі, оскільки кожен процесор (ядро) працює над своєю частиною завдання одночасно.

За останні кілька десятиліть кількість даних та складність алгоритмів зросла настільки, що багатоядерні процесори та GPU є незамінними інструментами для багатьох обчислювальних задач.Більшість сучасних комп'ютерних систем мають багатоядерні процесори, що дозволяють виконувати паралельні обчислення на рівні апаратного забезпечення навіть на персональних комп’ютерах. Але більш складні задачі виконуються паралельно на таких засобах, як комп'ютерні кластери, суперкомп'ютери, мультипроцесорні системи та інші.

Крім того, паралельні обчислення можуть бути виконані на рівні програмного забезпечення, використовуючи різноманітні технології, такі як бібліотеки для паралельних обчислень, мови програмування з підтримкою паралельних обчислень та системи кластеризації.

Для ефективної роботи паралельних обчислень використовуються спеціальні алгоритми та програми, які розподіляють завдання між процесорами та зберігають правильну послідовність виконання операцій.

Обчислення даної курсової роботи будуть проводитись на шестиядерному процесорі Intel Core i5-11400.

Основною метою паралельних обчислень є зменшення часу на рішення задачі або розв’язання більш складних задач (обробка більшого обсягу даних) за той самий проміжок часу, використовуючи більше процесорів або ядер.

Актуальність паралельних обчислень полягає в тому, що вони дозволяють розв'язувати складні задачі, що вимагають великої обчислювальної потужності, швидко та ефективно.

Паралельні обчислення можуть бути застосовані в багатьох областях науки та промисловості, включаючи наступні:

- наука про матеріали та інженерія: паралельні обчислення дозволяють виконувати складні симуляції молекулярної динаміки та обробки зображень, що вимагають великої обчислювальної потужності;
- фінансові ринки: паралельні обчислення дозволяють аналізувати великі обсяги фінансових даних та прогнозувати тенденції на фінансових ринках;
- інтернет-технології: паралельні обчислення дозволяють швидко обробляти великі обсяги даних, що генеруються веб-серверами та додатками;
- машинне навчання: паралельні обчислення дозволяють тренувати нейромережі та моделі машинного навчання на великих обсягах даних, що дозволяє досягти кращої точності та ефективності;
- космічні дослідження: паралельні обчислення дозволяють обробляти великі обсяги даних, що отримуються з космічних апаратів та обчислювати складні математичні моделі погоди та клімату.

Також, паралельні обчислення можуть бути застосовані в багатьох інших областях, таких як медицина, біологія, енергетика, транспорт, виробництво тощо. Наприклад, у медицині паралельні обчислення можуть допомогти при аналізі генетичних даних, у транспорті - при оптимізації маршрутів транспортних засобів, а в енергетиці - при моделюванні та проектуванні систем електропостачання.

У багатьох випадках, застосування паралельних обчислень дозволяє не тільки зменшити час виконання обчислювальних завдань, але й зменшити

вартість обчислень та збільшити їх ефективність. Наприклад, замість того, щоб купувати десятки однакових комп'ютерів, можна скористатись паралельними обчисленнями та використати велику кількість доступної обчислювальної потужності.

Отже, паралельні обчислення є незамінним інструментом для вирішення складних обчислювальних задач в багатьох галузях науки та промисловості. Вони дозволяють ефективно використовувати ресурси та скорочувати час виконання завдань, що робить їх незамінними для більшості сучасних обчислювальних задач.

В рамках виконання курсової роботи необхідно виконати наступні завдання:

1) Виконати розробку паралельного алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування за допомогою мови програмування Java.

2) Забезпечити зручне введення даних для початку обчислень.

3) Виконати тестування алгоритму та довести коректність результатів обчислень.

4) Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.

5) Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше ніж 1,2.

6) Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.

1 ОПИС ПОСЛІДОВНОГО АЛГОРИТМУ ТА ЙОГО ВІДОМИХ ПАРАЛЕЛЬНИХ РЕАЛІЗАЦІЙ

1.1 Опис задачі

Задача про пакування рюкзака є однією з найвідоміших задач комбінаторної оптимізації. Вона полягає в тому, щоб запакувати максимальну вагу предметів в обмежений за об'ємом рюкзак.

Вхідними даними задачі є:

- в наявності N речей і рюкзак місткістю W;
- річ з номером і характеризується розміром (вагою) w_i і цінністю c_i;
- додаткова умова: w_i – додатні цілі числа.
- Потрібно вибрати деяку підмножину речей так, щоб:
- сумарний розмір обраних речей не перевищував W;
- сумарна вартість обраних речей була якомога більше.

Задачу про пакування рюкзака можна розв’язати різними методами:

- застосуванням наближеного методу розв’язання – жадібного алгоритму (*greedy algorithm*) [1], який забезпечує результат, наближений до оптимального, але не обов'язково оптимальний;
- перебором всіх варіантів набора з N предметів. Часова складність такого рішення – $O(2^N)$ [2];
- методом зустрічі всередині (*meet-in-the-middle*). Складність рішення $O(2^{N/2}N)$ [2][3];
- методом гілок і меж (*branch-and-bound algorithm*)[4];
- методом динамічного програмування, реалізація якого є завданням курсової роботи. Часова складність алгоритму дорівнює O(NW). Алгоритм має дві переваги: швидкість та те, що він не вимагає сортування змінних, та недолік: вимагає порівняно багато пам'яті (що робить його не прийнятним для розв'язання великих задач)[5].

1.2 Опис послідовного алгоритму розв’язання задачі методом динамічного програмування

1.2.1 Основна ідея:

Ідея методу динамічного програмування полягає в тому, щоб розбити задачу на менші підзадачі та використати результати розв’язання цих підзадач для розв'язання більш складної задачі.

Для задачі про пакування рюкзака підзадачею буде прийняття рішення, чи варто додати певний предмет до рюкзака чи ні.

Розв'язування задачі про пакування рюкзака за допомогою методу динамічного програмування полягає в побудові таблиці, в якій зберігається найбільша вага, яку можна запакувати, враховуючи попередні предмети та об'єм рюкзака.

1.2.2 Кроки алгоритму:

1. Створити двовимірний масив A розміром (N+1) x (W+1), де N - кількість предметів, W - максимальний об’єм рюкзака (максимальна вага, яку можна помістити в рюкзак).

Елемент масиву (i, j) буде містити максимальну вагу, яку можна запакувати, включаючи перші і предметів, якщо об'єм рюкзака дорівнює j.

2. Заповнити перший рядок та першу колонку масиву нулями, оскільки при вагах предметів, рівних нулю, вага, яку можна запакувати, також буде нульовою.

3. Пройтися по всіх комбінаціях предметів та об'ємів рюкзака від 1 до N та від 1 до W відповідно.

4. Якщо вага поточного предмета менша або дорівнює поточному об'єму рюкзака, тоді вирахувати максимальну вагу, яку можна запакувати, враховуючи поточний предмет та об'єм, що залишився в рюкзаку – як максимальну з ваг:

- предмета, що ми збираємось додати до рюкзака (вага його плюс максимальна вага, яку можна запакувати з врахуванням попередніх предметів і об'єму, що залишився в рюкзаку);

- максимальної ваги, яку можна запакувати без додавання і-го предмета (вага максимальної ваги, яку можна запакувати, з врахуванням попередніх предметів і об'єму, що залишився в рюкзаку без додавання і-го предмета).

A[i][j] = max(A[i-1][j], A[i-1][j-w[i-1]] + c[i-1])

де w[i-1] - вага і-го предмета, c[i-1] - вартість і-го предмета.

5. Якщо вага поточного предмета більша за поточний об'єм рюкзака, тоді вага, яку можна запакувати, дорівнює вазі, яку можна запакувати без додавання поточного предмета:

A[i][j] = A[i-1][j]

6. Результат (максимальна вага, яку можна запакувати) буде розміщена в останньому елементі A[N][W].

1.2.3 Відновлення набору предметів, що відібрані в рюкзак

Визначатимемо, чи входить N_i предмет у набір.

Починаємо з елемента A(i, j), де i = N, j = W.

Для цього порівнюємо A(i, j) з такими значеннями:

Максимальна вартість рюкзака з такою ж місткістю та набором допустимих предметів $\{n_1,n_2,\ldots,n_{i-1}\}$, тобто A(i-1,j)

Максимальна вартість рюкзака з місткістю на w_i менше та набором допустимих предметів $\{n_1,n_2,\ldots,n_{i-1}\}$ плюс вартість c_i, тобто $A(i-1,j-w_i)+c_i$

Порівнюватимемо A(i, j) з A(i−1, j). Якщо вони рівні, тоді n_i не входить до набору, що шукається, інакше входить.

1.2.4 Псевдокод

```
for j = 0 to W    //initialize upper row and left column of table 0
  A[0][i] = 0
for i = 0 to N
  A[i][0] = 0
```

```
for i = 1 to N    // sort through all items
  for j = 1 to W  // sort through each i all possible capacities
    if j >= w[i]  // current item fits in backpack, put it in?
      A[i][j] = max(A[i - 1][j], A[i - 1][j - w[i]] + p[i]) //yes
    else
      A[i][j] = A[i - 1][j]                                  // no
```

1.2.5 Приклад роботи алгоритму

Нехай максимальна місткість рюкзака W = 14, кількість предметів N = 12, їхні вартості та маси наведені в таблиці 1.1:

№	Умовна назва	Вартість p, грн.	Вага w, кг
1.	Sleek Aluminum Bag	11	5
2.	Enormous Silk Shoes	10	8
3.	Incredible Concrete Computer	14	11
4.	Ergonomic Leather Table	12	15
5.	Sleek Marble Table	1	15
6.	Lightweight Aluminum Bottle	13	2
7.	Incredible Linen Bag	10	7
8.	Synergistic Plastic Lamp	8	13
9.	Rustic Leather Lamp	2	10
10.	Intelligent Aluminum Car	8	9
11.	Heavy Duty Linen Bottle	3	9
12.	Practical Leather Coat	12	14

Табл. 1.1 – Приклад набору предметів, з яких ми обираємо речі в рюкзак

У наведеній нижче таблиці 1.2 вказано значення заповненого масиву A[k+1][W+1].

	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7	j=8	j=9	j=10	j=11	j=12	j=13	j=14
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=1	0	0	0	0	0	11	11	11	11	11	11	11	11	11	11
i=2	0	0	0	0	0	11	11	11	11	11	11	11	11	21	21

i=3	0	0	0	0	0	11	11	11	11	11	11	14	14	21	21
i=4	0	0	0	0	0	11	11	11	11	11	11	14	14	21	21
i=5	0	0	0	0	0	11	11	11	11	11	11	14	14	21	21
i=6	0	0	13	13	13	13	13	24	24	24	24	24	24	27	27
i=7	0	0	13	13	13	13	13	24	24	24	24	24	24	27	34
i=8	0	0	13	13	13	13	13	24	24	24	24	24	24	27	34
i=9	0	0	13	13	13	13	13	24	24	24	24	24	24	27	34
i=10	0	0	13	13	13	13	13	24	24	24	24	24	24	27	34
i=11	0	0	13	13	13	13	13	24	24	24	24	24	24	27	34
i=12	0	0	13	13	13	13	13	24	24	24	24	24	24	27	34

Таблиця 1.2 – Результат пошуку максимальної ваги, яку можна запакувати

1.3 Опис паралельного алгоритму

Паралельна реалізація задачі про пакування рюкзака може допомогти збільшити швидкість обчислення, особливо для великих значень N та W.

Ідея полягає в тому, щоб розділити проблему на підзадачі, що можуть бути розраховані незалежно. Один з підходів полягає у тому, щоб розділити масиви w та c на рівні частини та розподілити їх між процесами. Кожен процес обчислює свою частину масиву A, а потім ці значення об'єднуються в один A.

Одна з основних складнощів при паралельній реалізації полягає у тому, щоб забезпечити правильну роздачу підзадач між процесами та правильне об'єднання результатів.

Також потрібно враховувати, що надмірна паралелізація може призвести до збільшення часу виконання, оскільки комунікація між процесами може займати більше часу, ніж обчислення.

Існує кілька способів реалізації паралельної версії задачі про пакування рюкзака методом динамічного програмування з використанням різних технологій паралелізму.

Вибір конкретного підходу залежить від потреби, доступності технології та мови програмування, яка використовується.

Висновок: Розробка алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування та його паралельна реалізація важливі, мають велике практичне значення і тому дослідження, що проводиться в рамках курсової роботи, є актуальними.

2 РОЗРОБКА ПОСЛІДОВНОГО АЛГОРИТМУ ТА АНАЛІЗ ЙОГО ШВИДКОДІЇ

Нижче наведено послідовну реалізацію мовою Java алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування.

2.1 Розробка класів

З метою модульної організації даних та структурування програми розроблені класи **Item** та **ItemsWarehouse**, які дозволяють створювати та управляти сховищем предметів, генерувати випадкові предмети, зберігати та завантажувати предмети з файлу та отримувати деталі про окремі предмети, інтерфейс **PackBackpack**, який визначає метод pack() розміщення предметів у рюкзаку та клас **Dynamic**, який реалізує інтерфейс **PackBackpack** за допомогою методу динамічного програмування.

Основні елементи коду:

Клас **ItemsWarehouse** представляє сховище предметів. Основні методи класу включають наступне:

- **ItemsWarehouse(int countItems, int maxWeight, int maxPrice)** – конструктор, який генерує новий набір предметів з заданою кількістю, максимальною вагою та максимальною ціною;
- **ItemsWarehouse(String filePath)** – конструктор, який створює набір предметів, завантажений з файлу, шлях до якого вказується параметром filePath;
- **getItems()** – повертає список предметів у сховищі;
- **getItem(int id)** – повертає предмет з заданим ідентифікатором id;
- **getWeight(int id)** – повертає вагу предмета з заданим ідентифікатором id;

- **getPrice(int id)** – повертає ціну предмета з заданим ідентифікатором id;
- **getName(int id)** – повертає назву предмета з заданим ідентифікатором id.
- **generate(int count, int maxWeight, int maxPrice)** – генерує новий набір з count предметів з випадковими назвами, вагою (до maxWeight) та ціною (до maxPrice);
- **saveItems(String path)** – серіалізує та зберігає список предметів у файл, вказаний параметром path;
- **loadItems(String filePath)** – завантажує та десеріалізує список предметів з файлу, вказаного параметром filePath;
- **printItems()** – виводить деталі кожного предмета у сховищі.

Клас **Item** представляє окремий предмет. Він має наступні методи:

- **Item(int id, String name, int weight, int price)** – конструктор, який ініціалізує предмет з заданим ідентифікатором id, назвою name, вагою weight та ціною price;
- **getId()** – повертає ідентифікатор предмета;
- **getName()** – повертає назву предмета;
- **getPrice()** – повертає ціну предмета;
- **getWeight()** – повертає вагу предмета.

Лістинг коду класів **Item**, **ItemsWarehouse, PackBackpack** наведений у додатку А.

Клас **Dynamic** реалізує інтерфейс **PackBackpack** і має наступні поля:

- **itemsWarehouse** - об'єкт типу ItemsWarehouse, який представляє склад предметів;

- **countItems** – кількість предметів у складі;.
- **maxW** – максимальна вага (місткість) рюкзака;
- **totalPrice** – загальна ціна розміщених предметів у рюкзаку;
- **totalWight** – загальна вага розміщених предметів у рюкзаку;

Конструктор **Dynamic(ItemsWarehouse itemsWarehouse, int maxW)** приймає об'єкт складу предметів та максимальну вагу рюкзака і ініціалізує відповідні поля.

Метод **pack()** виконує розміщення предметів у рюкзаку за допомогою методу динамічного програмування. Спочатку виводиться повідомлення про використання методу динамічного програмування. Далі створюється двовимірний масив A, який представляє таблицю для збереження проміжних результатів.

Метод **fillTable(A)** заповнює цю таблицю. Потім викликається метод **findSelectedItems(A)**, який знаходить список обраних предметів за найкращою комбінацією. На основі результатів викликається метод **printResult(selected)**, який виводить результати вибору предметів для рюкзака.

Метод **fillTable(int[][] A)** заповнює двовимірний масив A за допомогою алгоритму динамічного програмування.

Для ітерації по предметам використовуються індекси від 1 до **countItems**, а для ітерації по вагам – індекси від 0 до **maxW**.

На кожній ітерації перевіряється, чи можна додати поточний предмет до рюкзака. Якщо вага поточного предмета більше, ніж максимальна вага рюкзака, то предмет не можна додати до рюкзака, тому значення відповідного елементу масиву A просто копіюється з попереднього рядка.

Якщо ж вага поточного предмета менше або дорівнює максимальній вазі рюкзака, то розглядаються два випадки: додати поточний предмет до рюкзака або не додавати його. Ми обчислюємо максимальну вартість, яку можна отримати в обох випадках і зберігаємо її відповідному елементу масиву A.

Таким чином, внутрішній цикл обчислює найкращу вартість розміщення предметів у рюкзаку для кожної можливої комбінації предметів та місткостей рюкзака. Якщо вага поточного предмета менша або рівна місткості рюкзака, тоді обчислюється вартість включення цього предмета або виключення його залежно від того, який варіант дає більшу загальну вартість.

На основі цього методу знаходиться максимальна вартість, яку можна отримати з обмеженням на вагу рюкзака, та список обраних предметів за найкращою комбінацією. Результатом є заповнена таблиця А.

Діаграма класів, задіяних у послідовній реалізації алгоритму наведена на рис. 2.1.

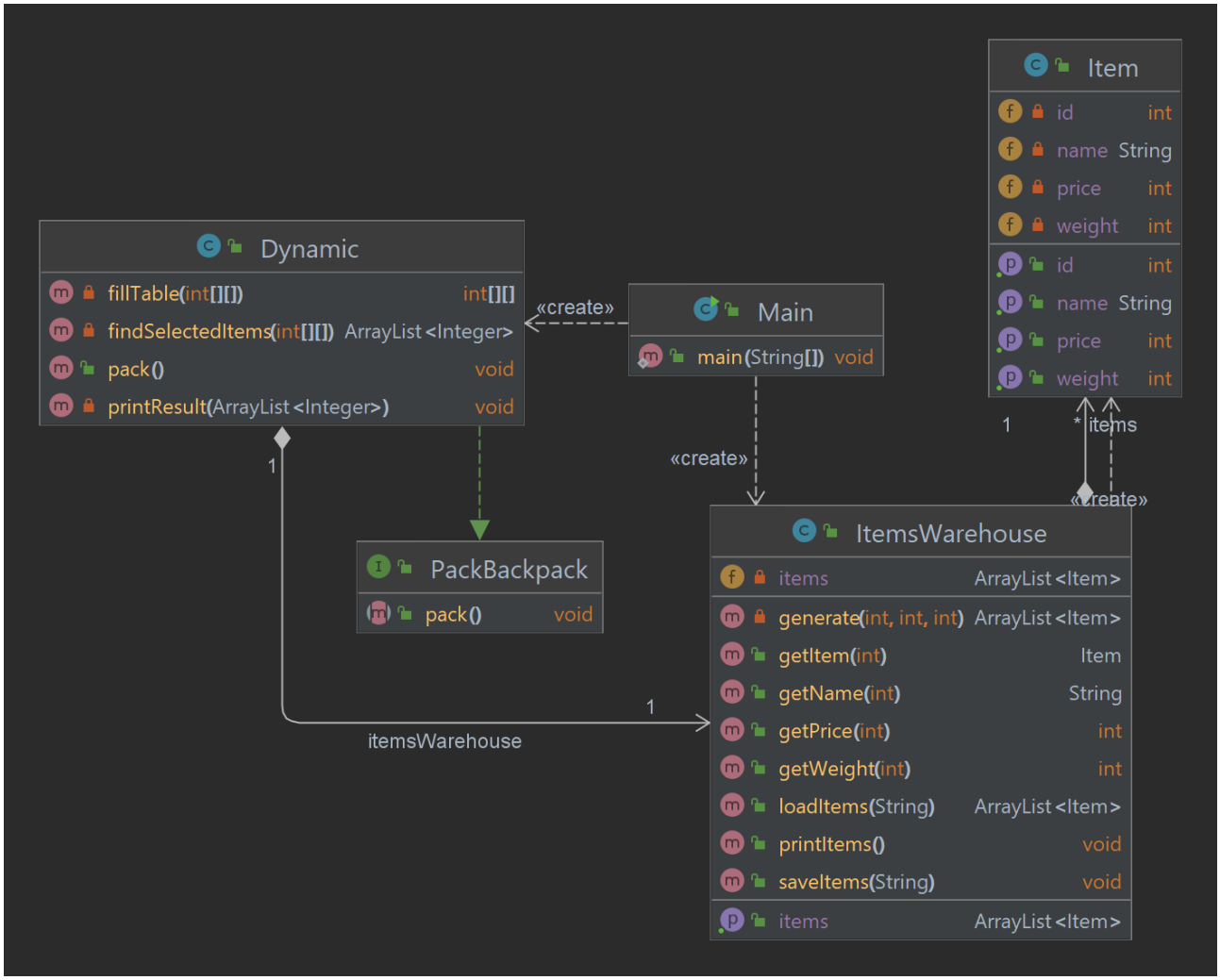


Рис. 2.1 – Діаграма класів послідовної реалізації алгоритму

Псевдокод реалізації алгоритму представлений на рисунку 2.2.

```
class Dynamic implements PackBackpack {
  itemsWarehouse : ItemsWarehouse
  countItems : int       // number of items
  maxW : int             // maximum weight (capacity) of the backpack
  totalPrice : int = 0
  totalWight : int = 0

  constructor(itemsWarehouse : ItemsWarehouse, maxW : int) {
    this.itemsWarehouse = itemsWarehouse
    countItems = itemsWarehouse.getItems().size()
    this.maxW = maxW
  }

  method pack() {
    A : 2D int array = new int[countItems + 1][maxW + 1]   // Table
    A = fillTable(A)
    selected : ArrayList<int> = findSelectedItems(A)
    printResult(selected)
  }

// Filling table A to find optimal solution
  method fillTable(A : 2D int array) : 2D int array {


    for i from 1 to countItems {   // i is the item index
      for j from 1 to maxW {      // j is the backpack capacity

// Weight of the item is less than or equal to the backpack capacity
        if itemsWarehouse.getWeight(i - 1) <= j {

          // Price of including the current item
          includePrice : int = itemsWarehouse.getPrice(i - 1) + A[i - 1][j
- itemsWarehouse.getWeight(i - 1)]
         // Price of excluding the current item
          excludePrice : int = A[i - 1][j]
           if includePrice > excludePrice {
            A[i][j] = includePrice   // Select the item to include
          } else {
            A[i][j] = excludePrice   // Select the item to exclude
          }
        } else {
// Weight of the item is greater than the backpack capacity, skip the item
          A[i][j] = A[i - 1][j]
        }
      }
    }
    return A
  }

  method findSelectedItems(A : 2D int array) : ArrayList<int> {
    selectedItems : ArrayList<int> = new ArrayList<int>()
    n : int = countItems
    capacity : int = maxW
    while n > 0 and capacity > 0 {
      if A[n][capacity] != A[n - 1][capacity] {
        selectedItems.add(n - 1)  // Add the index of the selected item
        totalPrice += itemsWarehouse.getPrice(n - 1)
        totalWight += itemsWarehouse.getWeight(n - 1)
        capacity -= itemsWarehouse.getWeight(n - 1)
```

```
            }
            n -= 1
          }
          return selectedItems
        }

        method printResult(result : ArrayList<int>) {
          print("Items in the backpack: ")
          result.reverse()
          for num in result {
            print(num + 1 + " : " + itemsWarehouse.getName(num) + " " +
itemsWarehouse.getWeight(num) + " kg " + itemsWarehouse.getPrice(num) + " UAH")
          }
          print("Weight = " + totalWight + " Price = " + totalPrice)
        }
      }
```

Рис. 2.2 – Псевдокод реалізації послідовного алгоритму

Лістинг коду програми, що реалізує послідовний алгоритм, наведений в додатку В.

Результати заповнення таблиці A для вихідних даних, наведених в розділі 1, показано на рис. 2.3, результати роботи програми – на рис. 2.4.

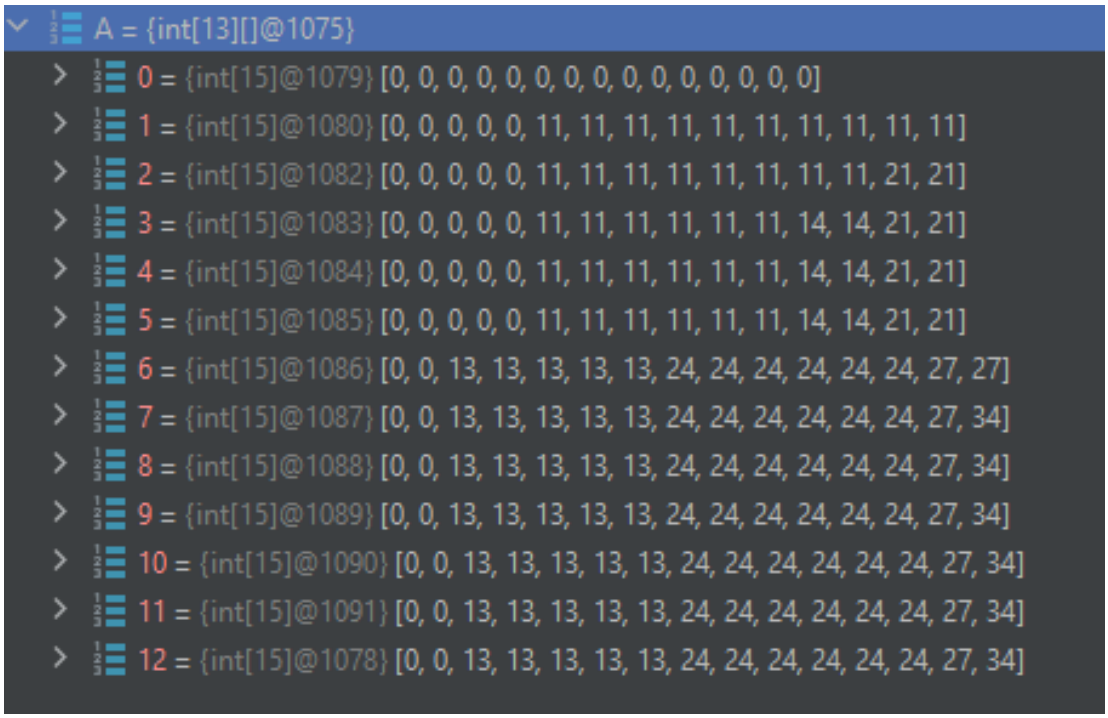


Рис. 2.3 – Результати заповнення програмою таблиці A

```
Вихідні дані:
        1  Sleek Aluminum Bag                          11 грн.    5  кг
        2  Enormous Silk Shoes                         10 грн.    8  кг
        3  Incredible Concrete Computer                14 грн.   11  кг
        4  Ergonomic Leather Table                     12 грн.   15  кг
        5  Sleek Marble Table                           1 грн.   15  кг
        6  Lightweight Aluminum Bottle                 13 грн.    2  кг
        7  Incredible Linen Bag                        10 грн.    7  кг
        8  Synergistic Plastic Lamp                     8 грн.   13  кг
        9  Rustic Leather Lamp                          2 грн.   10  кг
       10  Intelligent Aluminum Car                     8 грн.    9  кг
       11  Heavy Duty Linen Bottle                      3 грн.    9  кг
       12  Practical Leather Coat                      12 грн.   14  кг

Послідовна реалізація алгоритму пакування рюкзака
за допомогою методу динамічного програмування:
countItems =      12  capacity =   14 -  час виконання:       161000 наносекунд
Зібраний рюкзак:
        1  Sleek Aluminum Bag                          11 грн.    5  кг
        6  Lightweight Aluminum Bottle                 13 грн.    2  кг
        7  Incredible Linen Bag                        10 грн.    7  кг
Вага = 14 Вартість  = 34
```

Рис. 2.4 – Результати виконання програми, що реалізує послідовний алгоритм пакування рюкзака

Перевірка коректності роботи програми показує, що результати роботи програми співпадають з результатами, розрахованими у розділі 1 та наведеними в таблиці 1.2.

2.2 Аналіз швидкодії послідовного алгоритму

Для аналізу швидкодії з використанням розробленої програми були підготовлені файли на диску із тестовими даними (рис. 2.5) та проведено серію експериментів.

Лістинг коду підготовки вихідних даних для кількості предметів **countItems** = 10, 100, 1000, 10000, 100000, 1000000, діапазону ваги предметів від 1 до 40, діапазону вартостей від 1 до 100, а також запуску тестів для місткості рюкзака **capacity** = 25 та кількості предметів **countItems** = 10, 10, 100 показаний на рис. 2.5.

```
ItemsWarehouse itemsWarehouse;
Dynamic dynamic;

int capacity = 25;
int maxWeight = 40;
int maxPrice = 100;
itemsWarehouse = new ItemsWarehouse( countItems: 10,maxWeight,maxPrice);
itemsWarehouse.saveItems( path: "d:\\items-10-40-100.dat");
itemsWarehouse = new ItemsWarehouse( countItems: 100,maxWeight,maxPrice);
itemsWarehouse.saveItems( path: "d:\\items-100-40-100.dat");
itemsWarehouse = new ItemsWarehouse( countItems: 1000,maxWeight,maxPrice);
itemsWarehouse.saveItems( path: "d:\\items-1000-40-100.dat");
itemsWarehouse = new ItemsWarehouse( countItems: 10000,maxWeight,maxPrice);
itemsWarehouse.saveItems( path: "d:\\items-10000-40-100.dat");
itemsWarehouse = new ItemsWarehouse( countItems: 100000,maxWeight,maxPrice);
itemsWarehouse.saveItems( path: "d:\\items-100000-40-100.dat");
itemsWarehouse = new ItemsWarehouse( countItems: 1000000,maxWeight,maxPrice);
itemsWarehouse.saveItems( path: "d:\\items-1000000-40-100.dat");

itemsWarehouse = new ItemsWarehouse( filePath: "d:\\items-10-40-100.dat");
dynamic = new Dynamic(itemsWarehouse, capacity);
dynamic.pack();

itemsWarehouse = new ItemsWarehouse( filePath: "d:\\items-10-40-100.dat");
dynamic = new Dynamic(itemsWarehouse, capacity);
dynamic.pack();

itemsWarehouse = new ItemsWarehouse( filePath: "d:\\items-100-40-100.dat");
dynamic = new Dynamic(itemsWarehouse, capacity);
dynamic.pack();
```

Рис. 2.5 – Лістинг коду підготовки вихідних даних та запуску тестів

Результат роботи програми з підготовки даних та запуску тестів показаний на рис. 2.6.

```
Об'єкти були серіалізовані та збережені у файлі d:\items-10-40-100.dat
Об'єкти були серіалізовані та збережені у файлі d:\items-100-40-100.dat
Об'єкти були серіалізовані та збережені у файлі d:\items-1000-40-100.dat
Об'єкти були серіалізовані та збережені у файлі d:\items-10000-40-100.dat
Об'єкти були серіалізовані та збережені у файлі d:\items-100000-40-100.dat
Об'єкти були серіалізовані та збережені у файлі d:\items-1000000-40-100.dat

Послідовна реалізація алгоритму пакування рюкзака
за допомогою методу динамічного програмування:
countItems =      10  capacity =   25 -  час виконання:       141800 наносекунд
Зібраний рюкзак:
        2  Sleek Bronze Bottle                          87 грн.    7  кг
        7  Small Wooden Bench                          71 грн.    3  кг
        9  Awesome Wooden Pants                        97 грн.    6  кг
Вага = 16 Вартість  = 255

Послідовна реалізація алгоритму пакування рюкзака
за допомогою методу динамічного програмування:
countItems =      10  capacity =   25 -  час виконання:        68100 наносекунд
Зібраний рюкзак:
        2  Sleek Bronze Bottle                          87 грн.    7  кг
        7  Small Wooden Bench                          71 грн.    3  кг
        9  Awesome Wooden Pants                        97 грн.    6  кг
Вага = 16 Вартість  = 255

Послідовна реалізація алгоритму пакування рюкзака
за допомогою методу динамічного програмування:
countItems =     100  capacity =   25 -  час виконання:       435300 наносекунд
Зібраний рюкзак:
        3  Mediocre Bronze Gloves                       72 грн.    4  кг
       10  Sleek Steel Keyboard                        90 грн.    6  кг
       14  Fantastic Aluminum Bench                    99 грн.    7  кг
       22  Rustic Marble Table                         37 грн.    2  кг
       36  Heavy Duty Copper Watch                     40 грн.    2  кг
       99  Small Bronze Bottle                         74 грн.    3  кг
Вага = 24 Вартість  = 412
```

Рис. 2.6 – Результат роботи програми з підготовки даних та запуску тестів

Експеримент показав, що при першому запуску тесту на 10 елементів час виконання дорівнював 141800 нс, а при другому (з тими ж параметрами) – 65100 нс.

Зазначений результат пояснюється тим, що в Java існує поняття "теплого" і "холодного" запуску методів. При першому виконанні методу, JVM (Java Virtual Machine) виконує деякі операції, які можуть займати додатковий час, такі як завантаження класів, ініціалізація статичних полів, оптимізація коду та інші налаштування. Це може призвести до затримки у виконанні методу і виглядає, що час першого виконання більший. При повторному виконанні методу, JVM може використовувати кеш інструкцій та результатів оптимізації, що дозволяє йому прискорити виконання. Також, при повторному виклику методу, дані можуть бути уже завантажені в кеш процесора, що також сприяє збільшенню швидкості виконання. Отже, різниця в часі виконання між першим і повторним викликами методу може бути пов'язана з процесом ініціалізації та оптимізації, який відбувається лише під час першого виклику.

У зв’язку з цим, в подальших експериментах перший тест буде виконуватися двічі, а результат першого запуску буде ігноруватися.

Крім того, з метою зменшення обсягу консольного виводу, а також його впливу на час виконання алгоритму, в подальших експериментах інформація про результати відбору предметів не виводиться (на час тестування код виводу на консоль результатів відбору був за коментований).

Результати експериментів при зміні кількості предметів показаний на рис. 2.7.

```
countItems =      10  capacity =   25 -  час виконання:       275900 наносекунд
countItems =      10  capacity =   25 -  час виконання:       134300 наносекунд
countItems =     100  capacity =   25 -  час виконання:       493000 наносекунд
countItems =    1000  capacity =   25 -  час виконання:      5422700 наносекунд
countItems =   10000  capacity =   25 -  час виконання:     18939900 наносекунд
countItems =  100000  capacity =   25 -  час виконання:     32560100 наносекунд
countItems = 1000000  capacity =   25 -  час виконання:    348808000 наносекунд
```

Рис. 2.7 – Результат виконання тестів при збільшенні кількості предметів.

Відмінність у результатах тестів, показаних на рис. 2.6 та рис. 2.7, пояснюється різними умовами, в яких вони запускалися (повторно не виконувався код з підготовки та збереження даних).

У зв’язку з внесенням віртуальною машиною Java похибок у час виконання алгоритму, що особливо відчувається на малих обсягах тестових даних, для зручності аналізу в подальшому час буде підраховуватися в мілісекундах, кількість речей менше 100 аналізуватися не буде.

Результати експериментів при зміні кількості предметів та обсягу рюкзаку показані в таблиці 2.1.

Для кожної групи параметрів виконувалася серія по 3 виміри та розрахований усереднений час виконання ($t_{усер.}$). У зв’язку з тим, відхилення вимірюних значень від усереднених виявилися незначними, в подальшому для кожної групи параметрів буде виконуватися лише одне вимірювання.

Результати експериментів при зміні кількості предметів, параметрів обсягу рюкзаку, допустимих діапазонів ваги та вартості предметів показані в таблицях 2.1, 2.2 та 2.3.

	Ємність рюкзака (capacity)											
	10		25		50		75		100		200	
Кільк. речей	t, мс	$t_{усер.}$	t, мс	$t_{усер.}$	t, мс	$t_{усер.}$	t, мс	$t_{усер.}$	t, мс	$t_{усер.}$	t, мс	$t_{усер.}$
100	0	0	1	1	2	2	3	3	4	4	7	5
	0		1		3		3		4		4	
	1		1		2		3		5		5	
1000	2	2	5	4	6	10	11	13	13	13	23	20
	2		3		10		15		14		16	
	2		3		13		12		12		22	
10000	9	9	18	19	20	27	23	25	18	19	29	26
	8		19		29		26		19		22	
	11		21		31		26		19		28	
100000	18	21	31	31	57	60	72	71	97	111	226	227
	27		30		61		70		112		225	
	18		32		62		71		123		231	
1000000	211	207	368	357	610	624	819	811	1075	1083	2141	2111
	200		344		611		803		1074		2097	
	209		358		652		810		1099		2095	

Табл. 2.1 – maxWeight = 40, maxPrice = 100

		Ємність рюкзака (capacity)								
		10	25	50	75	100	200	1000	10000	10000
Кількість речей	100	1	1	2	3	3	6	19	51	157
	1000	2	4	6	11	8	13	24	120	1228
	10000	10	16	21	20	23	20	109	1224	OME
	100000	36	27	54	64	85	201	1040	11458	OME
	1000000	204	323	511	775	949	1861	9747	OME	OME

Табл. 2.2 – maxWeight = 200, maxPrice = 100 **(**OME – OutOfMemoryError)

		Ємність рюкзака (capacity)								
		10	25	50	75	100	200	1000	10000	10000
Кількість речей	100	0	2	2	2	3	5	17	40	148
	1000	1	3	6	8	10	16	22	115	1567
	10000	7	17	21	22	32	21	117	1259	OME
	100000	18	26	54	67	88	204	1120	11205	OME
	1000000	151	318	540	738	996	1904	10670	OME	OME

Табл. 2.2 – maxWeight = 200, maxPrice = 20000

З отриманих даних видно, що час роботи програми збільшується при збільшенні кількості речей та при збільшенні ємності рюкзака.

Параметри речей, що містяться у рюкзаку, на час роботи програми практично не впливають.

2.3 Розв’язання задачі про пакування рюкзака методом перебору

Програмна реалізація розв’язання задачі про пакування рюкзака методом перебору виконана у класі **Bruteforce** поза межами завдання курсової роботи.

Мета розробки класу **Bruteforce** – практична перевірка коректності послідовної та паралельної реалізації методу динамічного програмування.

Основна функціональність класу полягає у пошуку оптимального розміщення предметів у рюкзаку методом перебору (brute force).

Основні елементи класу **Bruteforce**:

- **itemsWarehouse**: об'єкт типу ItemsWarehouse, який представляє склад предметів;
- **countItems** – кількість предметів у складі;
- **maxW** – максимальна вага (місткість) рюкзака;
- **totalPrice** – загальна ціна розміщених предметів у рюкзаку;
- **totalWight** – загальна вага розміщених предметів у рюкзаку;
- **maxSelection** – числове представлення комбінації предметів з найвищою ціною.

Конструктор **Bruteforce(ItemsWarehouse itemsWarehouse, int maxW)** приймає об'єкт складу предметів та максимальну вагу рюкзака і ініціалізує відповідні поля.

Метод **pack**() виконує пошук оптимальної комбінації предметів для рюкзака. За допомогою циклу перебираються всі можливі комбінації предметів, і для кожної комбінації розраховується загальна вага (wight) та ціна (price). Якщо вага не перевищує максимальну вагу рюкзака, то порівнюється ціна з попередньою оптимальною комбінацією, і якщо ціна вища, то оновлюються значення **totalPrice**, **totalWight** та **maxSelection**.

У кінці методу виводяться результати вибору предметів для рюкзака. За допомогою поразрядних операцій перевіряється, які предмети входять у вибрану комбінацію, та виводяться їх вага, ціна та індекси.

Клас також містить два приватних методи: **statePrice**(long selection) та **stateWeight**(long selection), які розраховують вагу та ціну для заданої комбінації предметів. Вони використовуються у методі **pack**() для розрахунку ваги та ціни комбінацій.

Лістинг коду програми, що реалізує розв’язання задачі методом перебору, наведений в додатку Г.

Результат виконання програми для вихідних даних, наведених у розділі 1, зображений на рисунку 2.8.

```
Вихідні дані:
        1  Sleek Aluminum Bag                          11 грн.    5  кг
        2  Enormous Silk Shoes                        10 грн.    8  кг
        3  Incredible Concrete Computer               14 грн.   11  кг
        4  Ergonomic Leather Table                    12 грн.   15  кг
        5  Sleek Marble Table                          1 грн.   15  кг
        6  Lightweight Aluminum Bottle                13 грн.    2  кг
        7  Incredible Linen Bag                       10 грн.    7  кг
        8  Synergistic Plastic Lamp                    8 грн.   13  кг
        9  Rustic Leather Lamp                         2 грн.   10  кг
       10  Intelligent Aluminum Car                    8 грн.    9  кг
       11  Heavy Duty Linen Bottle                     3 грн.    9  кг
       12  Practical Leather Coat                     12 грн.   14  кг

Метод перебору: використовуємо виключно для контролю на невеликих обсягах даних.
Зібраний рюкзак:
        1  Sleek Aluminum Bag                          11 грн.    5  кг
        6  Lightweight Aluminum Bottle                13 грн.    2  кг
        7  Incredible Linen Bag                       10 грн.    7  кг
Вага = 14 Вартість  = 34
```

Рис. 2.8 – Результат розв’язання задачі методом перебору

Результати роботи програми співпадають з результатами, розрахованими у з використанням методу динамічного програмування та наведеними на рисунку 2.4, що підтверджує коректність роботи програми.

3 ВИБІР ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ РОЗРОБКИ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ ТА ЙОГО КОРОТКИЙ ОПИС

3.1 Огляд мов програмування для розробки паралельних обчислень

Для розробки паралельних обчислень з урахуванням поставлених задач та обмежень можна використовувати різні мови програмування, кожна з яких має свої властивості.

Ці мови програмування мають різні підходи до паралельних обчислень і надають різні інструменти та бібліотеки для керування багатопотоковими програмами. Вибір мови програмування залежить від конкретних вимог проекту, обмежень, навичок програміста або команди.

До найбільш популярних мов програмування, які широко використовуються для вирішення задач паралельних обчислень, належать:

1) C та C++ – мови програмування, які надають прямий доступ до пам'яті та дозволяють використовувати низькорівневі операції. Вони широко використовуються для паралельного програмування, зокрема за допомогою бібліотек, таких як OpenMP і MPI. Ці бібліотеки дозволяють використовувати багатопотоковість та розподілені обчислення в програмах на C/C++. [6]

2) Python – високорівнева мова програмування з багатим екосистемою бібліотек. Для паралельних обчислень в Python використовуються бібліотеки, такі як multiprocessing, threading, concurrent.futures та інші. Вони дозволяють створювати та керувати багатопотоковими програмами в Python. [7]

3) Go – мова програмування, що розроблена з урахуванням паралельного та розподіленого програмування. Вона має вбудовану підтримку багатопотоковості, засоби синхронізації та канали для обміну даними між потоками. Go надає простий та ефективний спосіб створення багатопотокових програм. [8]

4) Scala – це мова програмування, що працює на JVM, і має потужну підтримку паралельних обчислень. Вона має вбудовані конструкції для

створення та керування потоками, а також бібліотеку Akka, яка надає акторну модель паралельного програмування. [9]

5) Мова програмування Java та віртуальна машина Java (JVM) надають потужні засоби для паралельних обчислень. Java є однією з найпопулярніших мов програмування для паралельних обчислень, має вбудовану підтримку багатопотоковості, багато інструментів та бібліотек для керування потоками та синхронізації, а також має кілька особливостей, які роблять її привабливою для розробки паралельних програм. [10]

3.2 Властивості віртуальної машини Java (JVM) для підтримки паралельних обчислень

Java та віртуальна машина Java (JVM) мають декілька властивостей, які роблять їх основними рішеннями для програмування паралельних обчислень.

Віртуальна машина Java (JVM) відіграє важливу роль у підтримці паралельних обчислень у мові програмування Java. Ось деякі властивості JVM, які сприяють паралельним обчисленням:

1) Менеджмент пам'яті: JVM відповідає за керування пам'яттю та виділення ресурсів між потоками. Вона автоматично вирішує питання виділення та звільнення пам'яті для об'єктів, забезпечуючи ефективне використання ресурсів. Крім того, JVM використовує механізми збору сміття для автоматичного видалення непотрібних об'єктів та відновлення вільної пам'яті.

2) Планування потоків: JVM має вбудований планувальник потоків, який визначає порядок виконання потоків в мультипотоковій програмі. Він вирішує питання розподілу обчислювальних ресурсів між потоками, включаючи виконання потоків на різних процесорах або ядрах процесора. Планувальник потоків може використовувати різні алгоритми планування, такі як round-robin або пріоритетне планування, для оптимального використання ресурсів системи.

3) Модель пам'яті: JVM має свою власну модель пам'яті, яка визначає, як потоки взаємодіють з пам'яттю та як змінні синхронізуються. Модель пам'яті JVM гарантує атомарність певних операцій та впорядкованість дій потоків, що сприяє правильній синхронізації та спільному доступу до даних.

4) Багатопотокова безпека: JVM надає механізми безпеки багатопотокових обчислень. Вона гарантує, що операції над об'єктами та змінними виконуються атомарно та безпечно у багатопотоковому середовищі. Це включає механізми блокування та синхронізації, що дозволяють уникнути гонок за даними та інших помилок, пов'язаних з багатопотоковістю.

5) Інструменти та бібліотеки: JVM має багатий набір інструментів та бібліотек для підтримки паралельних обчислень. Наприклад, Java Concurrency API надає класи та інтерфейси для керування потоками, синхронізації, паралельного виконання завдань та обміну даними між потоками. Ці інструменти спрощують розробку багатопотокових програм та допомагають ефективно використовувати ресурси системи.

3.3 Властивості мови Java у паралельному програмуванні

Виконання коду Java у віртуальній машині Java (JVM) може бути розподілено на багато потоків, що дозволяє виконувати обчислення паралельно. Ось кілька способів, якими можна досягти паралельного виконання коду Java у JVM:

1) Платформово-незалежність: Java є мовою програмування, яка компілюється в байт-код, що виконується на JVM. Це дозволяє програмам, написаним на Java, працювати на будь-якій платформі, де є встановлена відповідна версія JVM. Це дає змогу розробникам легко розподіляти та виконувати багатопотокові програми на різних обчислювальних пристроях та операційних системах.

2) Багатопотоковий підхід (Threads): Java має вбудовану підтримку для роботи з потоками. JVM дозволяє створювати багато потоків виконання у програмі Java, а Java має вбудовану підтримку багатопотоковості, що дозволяє використовуючи клас Thread або імплементуючи інтерфейс Runnable створювати та керувати багатьма потоками в одній програмі. Кожен потік виконує окрему частину коду паралельно з іншими потоками. Це дає можливість виконувати різні обчислювальні завдання паралельно, збільшуючи продуктивність програми. При цьому можна використовувати синхронізацію та механізми взаємодії між потоками, щоб забезпечити відповідну синхронізацію та обмін даними між потоками.

3) Модель пам'яті Java: Java має специфікацію моделі пам'яті, яка визначає порядок доступу до пам'яті та поведінку потоків. Ця модель пам'яті забезпечує багато корисних гарантій, таких як атомарність операцій та впорядкованість доступу до пам'яті, що допомагає уникнути проблем з багатопотоковою безпекою та забезпечити правильну синхронізацію.

4) Пул потоків (Thread Pool): в програмі, написанный мовою Java, можно використовувати пул потоків для керування паралельним виконанням завдань. В JVM існує пакет java.util.concurrent, який надає класи, такі як ExecutorService та ThreadPoolExecutor, для створення та керування пулом потоків. Замість створення нового потоку для кожної задачі можно використовувати пул потоків для повторного використання і керування доступними потоками. Пул потоків має фіксовану кількість потоків або динамічно налаштовується, він може автоматично розподіляти завдання між доступними потоками. Завдання можуть подаватися на виконання у пул потоків, і вони будуть виконуватися паралельно, використовуючи доступні потоки.

5) Бібліотеки паралельного програмування: Java має багато вбудованих бібліотек та інструментів для підтримки паралельного програмування, таких як java.util.concurrent та java.util.concurrent.atomic. Ці бібліотеки містять різноманітні класи і інструменти для синхронізації, координації і керування паралельними задачами, роблять розробку паралельних програм на Java більш

зручною та ефективною. Так, java.util.concurrent.atomic містить класи, що дозволяють виконувати атомарні операції над певними типами даних, такими як цілі числа (AtomicInteger), лонги (AtomicLong), булеві значення (AtomicBoolean) тощо. Це дозволяє забезпечити атомарність операцій і уникнути гонок за даними при одночасному доступі з різних потоків. Крім того, Java надає клас Semaphore, CountDownLatch та інші механізми взаємовиключення, які полегшують координацію та синхронізацію роботи між потоками.

6) Синхронізація (Synchronization): Java надає механізми для синхронізації доступу до спільних ресурсів у багатопотоковому середовищі. Використовуючи ключове слово synchronized можно блокувати об'єкт або метод, щоб забезпечити взаємовиключення між потоками і уникнути проблем, таких як гонки за ресурсами (race conditions). Java надає монітори як вбудований механізм синхронізації та координації роботи потоків. Також можно використовувати блоки synchronized для виконання операцій атомарно, тобто безперервно, незалежно від інших потоків. Крім того, Java надає класи Lock та Condition, які дають більш гнучкий підхід до синхронізації та керування потоками.

7) Паралельні колекції: починаючи з Java 8, в мову програмування Java були введені паралельні версії деяких стандартних колекцій, таких як ArrayList і HashMap. Паралельні колекції дозволяють одночасний доступ та зміну даних у багатопотоковому середовищі. Ці колекції можуть використовуватися для паралельної обробки даних без необхідності власноручної синхронізації.

8) Паралельні стріми: в Java 8 було введено нову функціональність - паралельні стріми (parallel streams). Стріми в Java дозволяють виконувати послідовні операції над потоком даних, таких як фільтрація, мапування та зведення. За допомогою методу parallel(), послідовний стрім може бути перетворений у паралельний стрім, що дозволить Java автоматично розподілити операції над елементами стріму між доступними потоками. Це дозволяє ефективно обробляти великі потоки даних паралельно.

9) Виключення та обробка помилок: Java має потужну систему виключень, яка допомагає управляти помилками та непередбаченими ситуаціями у багатопотокових програмах. Можно використовувати блок try-catch для обробки виключень та забезпечення правильного виконання коду в різних потоках. Крім того, Java надає інструменти, такі як Thread.UncaughtExceptionHandler, які дозволяють вам обробляти неперехоплені виключення та вживати відповідних заходів для відновлення програми.

Ці властивості роблять мову програмування Java та віртуальну машину Java (JVM) привабливими для розробки паралельних обчислень, дозволяючи створювати ефективні, переносимі та безпечні багатопотокові програми.

Таким чином, відповідно до завдання на курсову роботу, а також виходячи з властивостей для паралельної реалізації алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування обрано мову програмування Java.

4 РОЗРОБКА ПАРАЛЕЛЬНОГО АЛГОРИТМУ З ВИКОРИСТАННЯМ ОБРАНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ: ПРОЕКТУВАННЯ, РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ

4.1 Повторне використання коду послідовного алгоритму

Для забезпечення коректності порівняння часу виконання послідовного та паралельного алгоритмів, а також з метою забезпечення одного з принципів SOLID – DRY (don’t repeat yourself), частина коду, розробленого для реалізації послідовного алгоритму, буде використана для реалізації паралельного алгоритму.

Для виконання паралельного алгоритму будемо використовувати розроблені класи та методи:

- клас **Item** – для створення об’єкту, що відповідає одному предмету, геттерів та методу друку відомостей про предмет;
- клас **ItemsWarehouse** – для створення (генерації) наборів (сховищ) з предметів, збереження наборів на диску, повторного використання наборів шляхом читання з диску;
- класс **KnapSack** – для створення об’єкту, що відображає набір параметрів рюкзака: **capacity** – ємність рюкзака, **totalPrice** – вартість предметів у рюкзаку, **totalWight** – загальна вага предметів у рюкзаку, **packedItems** – набір предметів у рюкзаку, а також друку відомостей про рюкзак;
- інтерфейс **PackBackpack**, який визначає метод **pack**() розміщення предметів у рюкзаку;
- статичний метод **Helper.findSelectedItems** – для відновлення список обраних предметів з таблиці з результатами обчислень;
- метод **pack**() – код буде модифікований для забезпечення роботи у декількох потоках. При цьому основний алгоритм формування таблиці методом динамічного програмування залишається без змін (рис. 2.2).

4.2 Основні характерні ознаки паралельного алгоритму пакування рюкзака методом динамічного програмування

Мета розпаралелення алгоритму полягає у використанні паралельних обчислювальних ресурсів (таких, як багатоядерні процесори) для прискорення виконання обчислювальних завдань.

Цілями розпаралелення алгоритму є:

- збільшення продуктивності шляхом виконання більшої кількості обчислювальних завдань одночасно. Використання багатоядерних процесорів дозволяє розподілити завдання між різними обчислювальними одиницями і виконувати їх паралельно.
- зменшення часу виконання за рахунок розбиття обчислювальної задачі на менші частини і їх паралельного виконання.

Алгоритм реалізує паралельне виконання алгоритму пакування рюкзака з використанням багатопоточності. Загалом, паралельне виконання алгоритму досягається шляхом розподілу обчислювальної роботи між різними потоками, які паралельно виконують розрахунки для різних частин рюкзака.

Паралельний алгоритм вимагає фіксованої обміну даними між потоками та синхронізації.

Основні характеристики алгоритму, що розробляється:

1) Паралельність

Алгоритм має використовувати паралельні обчислення з використанням пулу потоків. Кожен предмет обробляється в окремому потоці, що дозволяє виконувати обчислення паралельно та прискорює процес пакування рюкзака.

2) Розбиття на частини

Ємність рюкзака розбивається на частини для обробки в окремих потоках. Кожний потік відповідає за обчислення результатів для своєї частини рюкзака.

Це дозволяє розділити обчислення між потоками та збільшити ефективність алгоритму.

3) Синхронізація

Використовується об'єкт CountDownLatch для синхронізації потоків та забезпечення того, що всі потоки завершать свою роботу перед продовженням виконання. Це дозволяє коректно злити результати обчислень з різних потоків.

4) Обмін даними

Дані обмінюються між потоками через спільний двовимірний масив dt, де зберігаються результати обчислень для кожного предмета та вміщуваної місткості рюкзака. Кожен потік має доступ до цього масиву та може записувати значення відповідних елементів.

5) Залежності поточних обчислень від попередніх

У динамічному програмуванні пакування рюкзака, кожен поточний розрахунок залежить від попередніх обчислень. Це забезпечує ефективне використання попередньо обчислених результатів для побудови оптимального рішення.

Залежність поточних обчислень від попередніх можна виразити рекурентною формулою. Давайте розглянемо загальну формулу для динамічного програмування пакування рюкзака.

Нехай dt - двовимірний масив, де dt[i][j] представляє максимальну сумарну цінність, яку можна отримати з вагою не більше j за умови, що враховані перші i предметів.

Формула для обчислення dt[i][j] буде залежати від двох можливостей:

Якщо вага items[i] (вага предмета i) менша або рівна j, тобто items[i].weight <= j, то ми можемо включити цей предмет до рюкзака. Тоді максимальна цінність буде складатися з цінності цього предмета та максимальної цінності,

яку ми можемо отримати з вагою не більше j - items[i].weight за умови, що враховані перші i-1 предметів.

Формула: **dt[i][j] = items[i].value + dt[i-1][j - items[i].weight].**

Якщо вага items[i] більша за j, тобто items[i].weight > j, то ми не можемо включити цей предмет до рюкзака. Тоді максимальна цінність буде такою ж, як і в попередньому кроці, тобто dt[i][j] = dt[i-1][j].

Таким чином, для обчислення поточного dt[i][j] нам необхідні значення dt[i-1][j] та dt[i-1][j - items[i].weight]. Ці значення були обчислені на попередніх кроках.

Така залежність дозволяє ефективно використовувати попередні результати обчислень і побудувати оптимальний розподіл предметів в рюкзаку.

Разом з цим, залежність поточних обчислень від попередніх у динамічному програмуванні пакування рюкзака ускладнює процес паралелізації алгоритму. Оскільки кожен поточний розрахунок залежить від попередніх результатів, необхідно забезпечити правильний порядок виконання цих обчислень.

Один із способів паралелізації алгоритму полягає у розподілі роботи між потоками таким чином, щоб кожен потік обчислював свою частину масиву dt. В цьому випадку, перед початком обчислень, потрібно забезпечити, щоб усі попередні результати, необхідні для поточних розрахунків, були доступні.

Одним з можливих підходів є використання динамічного планування завдань. При цьому кожному потоку призначається окремий діапазон індексів i, для яких він виконуватиме обчислення. Крім того, для кожного потоку потрібно забезпечити доступ до необхідних результатів попередніх обчислень.

Наприклад, можна використовувати синхронізаційні механізми, такі як CountDownLatch, для контролю потоків та забезпечення правильного порядку виконання. Кожен потік може чекати на завершення обчислень потрібних для його діапазону i-1, перш ніж розпочати свої обчислення для діапазону i. Після

завершення обчислень потрібних для певного діапазону, потік може перейти до наступного діапазону і.

У випадку реалізації паралельного алгоритму пакування рюкзака з використанням бібліотеки java.util.concurrent, залежності даних потоків такі:

1) Залежність даних між потоками виникає на етапі розділення роботи між потоками. Пул потоків (ExecutorService) використовується з фіксованою кількістю потоків (**numThreads**). Кожен потік обробляє окрему частину рюкзака (**partLeftIndex** і **partRightIndex**), але всі вони працюють зі спільним масивом dt.

2) У функції **calculate** відбувається паралельне обчислення цінностей для підмножини предметів, відповідно до вказаного діапазону частини рюкзака (**partLeftIndex** і **partRightIndex**). Кожен потік працює зі своєю власною частиною масиву dt, але він може залежати від результатів обчислень, виконаних попередніми потоками.

3) Для синхронізації завершення потоків використовується **CountDownLatch (latch)**. Після завершення обчислень у кожному потоці, він викликає **latch.countDown()**, зменшуючи лічильник. Головний потік викликає **latch.await()** для очікування завершення всіх потоків перед продовженням.

Таким чином, дані потоків залежать від обчислень, виконаних попередніми потоками, і вони спільно працюють зі спільним масивом dt, розділеним на частини для кожного потоку.

Однак, ефективність паралелізації може залежати від розміру задачі, кількості доступних потоків та характеристик апаратного забезпечення. Навіть за наявності залежності даних між обчисленнями, паралельна реалізація може прискорити алгоритм пакування рюкзака за рахунок виконання обчислень на різних частинах масиву dt одночасно.

4.3 Проектування алгоритму

Паралельний алгоритм буде релізовувати наступну послідовність дій:

1) Створити пул потоків (ExecutorService) з фіксованою кількістю потоків.

2) Ініціалізувати необхідні змінні, такі як **partCapacity** (часткова ємність для кожного потоку) та двовимірний масив **dt**, який використовується для збереження результатів обчислень.

3) У зовнішньому циклі пройти по номерам предметів (i), від 1 до **countItems** та створити об'єкт CountDownLatch з кількістю потоків **numThreads**. Цей об'єкт буде використовуватися для синхронізації потоків і забезпечення їх одночасного завершення.

4) У вкладеному циклі пройти по стадіях (**stage**) від 0 до **numThreads** – 1, кожна стадія представляє окремий потік обчислення.

У цьому циклі визначити змінні **itemIndex**, **partLeftIndex** та **partRightIndex**, які відповідають поточному предмету, лівій та правій межам частини рюкзака для поточного потоку.

5) Викликати метод **submit** пулу потоків та передати лямбда-вираз для паралельного виконання (виконати метод **calculate**, який обчислить значення в масиві **dt** для заданого піддіапазону).

6) Викликати метод **latch.await()**, що заблокує поточний потік до завершення всіх потоків, які виконують обчислення для поточного предмета. Це забезпечить правильний порядок виконання обчислень.

7) Продовжити зовнішній цикл до обробки всіх предметів.

Після завершення зовнішнього циклу (завершення всіх обчислень), зупинити пул потоків зупиняється методом **pool.shutdown()**.

8) Дочекатися завершення виконання всіх потоків у пулі за допомогою **pool.awaitTermination().**

9) Після завершення виконання всіх потоків викликати метод **Helper.findSelectedItems()**, який відновить список вибраних предметів на основі масиву dt та інформації про предмети у **itemsWarehouse**.

10) Визначити час виконання алгоритму.

11) Вивести результати на екран, включаючи кількість потоків (**numThreads**), кількість предметів (**countItems**), максимальну вагу рюкзака (**capacity**) та час виконання (**executionTime**).

Псевдокод алгоритму наведений на рис. 4.1

```
method pack():
  // Create a thread pool for parallel computations
  pool = Executors.newFixedThreadPool(numThreads)

  // Calculate the partial capacity for each thread
  partCapacity = (capacity + 1) / numThreads

  // Initialize the array to store calculation results
  dt = new int[countItems + 1][capacity + 1]

  // Start measuring the execution time
  startTime = current_time_millis()

  // Iterate over the items
  for i = 1 to countItems:
    // Create a latch to synchronize threads
    latch = new CountDownLatch(numThreads)

    // Divide the capacity range among threads
    for stage = 0 to numThreads - 1:
      // Determine the current item, left index, and right index
      itemIndex = i
      partLeftIndex = stage * partCapacity

      // Calculate the right index based on the thread stage
      if stage == numThreads - 1:
        partRightIndex = capacity
      else:
        partRightIndex = partLeftIndex + partCapacity - 1

      // Submit a task for parallel execution
      pool.submit(() -> {
        calculate(dt, latch, itemIndex, partLeftIndex, partRightIndex)
      })

    // Wait for all threads to finish their calculations
    latch.await()

  // Shutdown the thread pool
  pool.shutdown()

  // Wait for all threads to terminate
  pool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS)

  // Find the selected items for the knapsack
  knapSack = Helper.findSelectedItems(dt, itemsWarehouse, capacity)

  // Measure the total execution time
  endTime = current_time_millis()
  executionTime = endTime - startTime

  // Print the results
  print("Parallel\t", numThreads, "\t", countItems, "\t", capacity, "\t",
executionTime)


method calculate(dt, latch, itemIndex, partLeftIndex, partRightIndex):
```

```
        // Perform calculations in parallel for a subset of the knapsack capacity range
        for j = partLeftIndex to partRightIndex:
          if itemsWarehouse.getWeight(itemIndex - 1) <= j:
            includePrice = itemsWarehouse.getPrice(itemIndex - 1) + dt[itemIndex - 1][j
- itemsWarehouse.getWeight(itemIndex - 1)]
            excludePrice = dt[itemIndex - 1][j]
            if includePrice > excludePrice:
              dt[itemIndex][j] = includePrice
            else:
              dt[itemIndex][j] = excludePrice
          else:
            dt[itemIndex][j] = dt[itemIndex - 1][j]

        // Signal the completion of the thread's calculations
        latch.countDown()
```

Рис. 4.1 – Псевдокод алгоритму паралельного алгоритму пакування рюкзака методом динамічного програмування.

4.4 Реалізація алгоритму

Код реалізації паралельного алгоритму пакування рюкзака методом динамічного програмування наведений у додатку Г.

Клас **ParallelDynamic** імплементує інтерфейс **PackBackpack** та представляє паралельну версію алгоритму заповнення рюкзака.

Клас **ParallelDynamic** має наступні змінні:

- **itemsWarehouse** – екземпляр класу ItemsWarehouse, що представляє сховище предметів.
- **countItems** –кількість предметів у сховищі (отримується з itemsWarehouse).
- **capacity** – максимальна вага рюкзака.
- **numThreads** – кількість потоків для паралельних обчислень (значення за замовчуванням – 1).

Конструктор **ParallelDynamic** приймає екземпляр **itemsWarehouse**, максимальну вагу **capacity** і кількість потоків **numThreads**. Він ініціалізує змінні екземпляра відповідними значеннями.

Метод **pack()** є перевизначеним методом з інтерфейсу **PackBackpack**, він є основним в реалізації алгоритму та виконує заповнення рюкзака.

У методі **pack()** створюється пул потоків (ExecutorService pool), який використовується для паралельних обчислень.

Обчислення починаються з визначення часткової вмістимості рюкзака для кожного потоку (**partCapacity**). Також створюється двовимірний масив dt для збереження результатів обчислень.

Потім виконується два вкладені цикли. Перший цикл проходиться по номерах предметів (i), а другий цикл - по потоках (**stage**).

У другому циклі обчислюються межі рюкзака для кожного потоку (**partLeftIndex** і **partRightIndex**). Задача потоку полягає в обчисленні значень для певного діапазону місткостей рюкзака.

Потоки відправляються в пул для паралельного виконання, викликаючи метод **calculate**(). Цей метод виконує фактичні обчислення заповнення рюкзака для визначеного діапазону.

Після відправлення всіх потоків в пул, виконується очікування завершення всіх потоків за допомогою **latch.await().**

Після завершення всіх обчислень пул потоків закривається (**pool.shutdown()**), і виконується очікування завершення виконання всіх потоків (**pool.awaitTermination()**).

На основі результатів обчислень створюється об'єкт **KnapSack** за допомогою методу **Helper.findSelectedItems()**. Цей об'єкт містить список вибраних предметів для заповнення рюкзака. (Лістинг коду класів **Knapsack**, **Helper** наведений у додатку Б.)

На кінцевому етапі виводиться час виконання паралельного алгоритму пакування рюкзака (executionTime) та за необхідності – викликається метод knapSack.printResult() друку вмісту рюкзака (при проведенні аналізу часу виконання з різними параметрами метод не викликається).

4.5 Тестування алгоритму

Для перевірки коректності буде виконане порівняння результатів розрахунків з використанням паралельного алгоритму та інших алгоритмів. Критерієм коректності вважається співпадання результатів.

Для невеликих значень кількості предметів та ємності рюкзака будуть використані розрахунки наведеного в розділі 2 алгоритму пакування рюкзака методом перебору (клас **Bruteforce,** додаток Г).

Лістинг виклику коду, що реалізує алгоритм пакування методом перебору наведений на рисунку 4.2.

```
public class Main {
    public static void main(String[] args) {
        int capacity = 14;
        ItemsWarehouse itemsWarehouse;
        itemsWarehouse = new ItemsWarehouse( filePath: "d:\\items12.dat");
        System.out.println("Вихідні дані:");
        itemsWarehouse.printItems();
        Bruteforce bruteforce = new Bruteforce(itemsWarehouse, capacity);
        bruteforce.pack();
```

Рис. 4.2 – Лістинг виклику коду, що реалізує алгоритм пакування методом перебору

Результат роботи алгоритму пакування методом перебору наведений на рисунку 4.3.

```
Вихідні дані:
        1  Sleek Aluminum Bag                          11 грн.    5  кг
        2  Enormous Silk Shoes                         10 грн.    8  кг
        3  Incredible Concrete Computer                14 грн.   11  кг
        4  Ergonomic Leather Table                     12 грн.   15  кг
        5  Sleek Marble Table                           1 грн.   15  кг
        6  Lightweight Aluminum Bottle                 13 грн.    2  кг
        7  Incredible Linen Bag                        10 грн.    7  кг
        8  Synergistic Plastic Lamp                     8 грн.   13  кг
        9  Rustic Leather Lamp                          2 грн.   10  кг
       10  Intelligent Aluminum Car                     8 грн.    9  кг
       11  Heavy Duty Linen Bottle                      3 грн.    9  кг
       12  Practical Leather Coat                      12 грн.   14  кг

Метод перебору: використовуємо виключно для контролю на невеликих обсягах даних.
Зібраний рюкзак:
        1  Sleek Aluminum Bag                          11 грн.    5  кг
        6  Lightweight Aluminum Bottle                 13 грн.    2  кг
        7  Incredible Linen Bag                        10 грн.    7  кг
countItems =      12  capacity =   14 -  час виконання:          12 мс
Вага = 14 Вартість  = 34

Process finished with exit code 0
```

Рис. 4.3 – Результат розв’язання задачі методом перебору

Для великої кількості даних використовуються результати розрахунків за допомогою наведеного в розділі 2 послідовного алгоритму пакування рюкзака методом динамічного програмування (клас **Dynamic,** додаток В).

Код, що реалізовує паралельний алгоритм, виконується з використанням різної кількості потоків (1, 2, 3, 4, 8, 16).

Для забезпечення можливості обробки програмою великої кількості даних, аби уникнути помилки “OutOfMemoryError” (рис. 4.4), в IDE IntelliJ IDEA в опції “Build and run” конфігурації проекту додамо параметр “ -Xmx18g” (рисунок 4.5). Альтернативно зазначений параметр можно використати як опцію командного рядка.

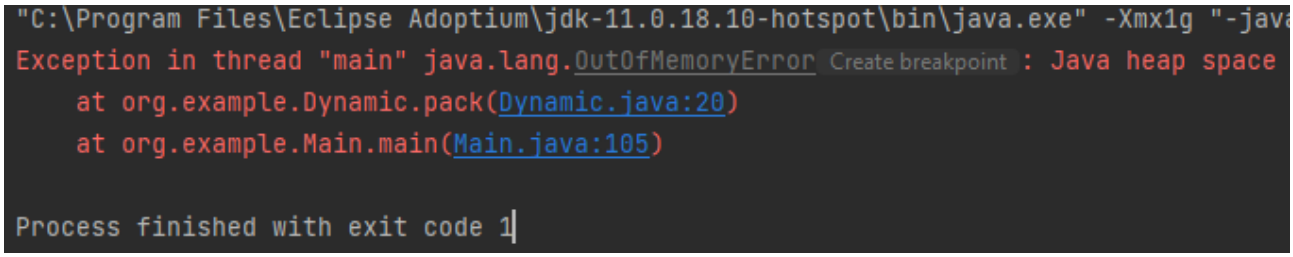


Рис. 4.4 – Помилка недостачі доступного обсягу оперативної пам'яті для виконання

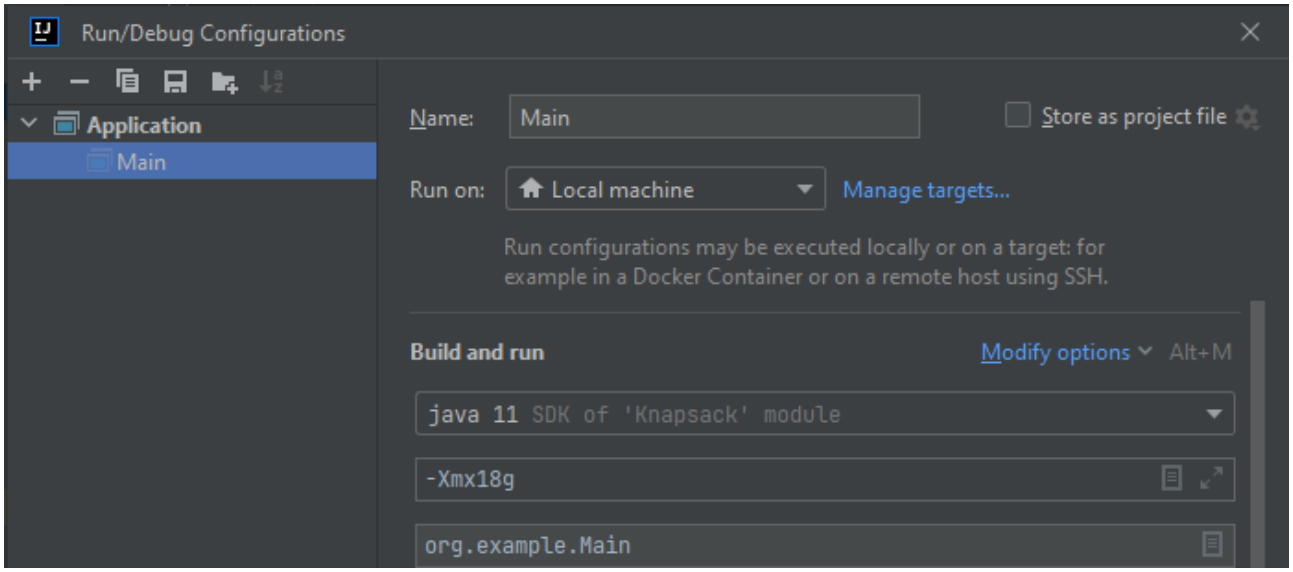


Рис. 4.5 – Налаштування обсягу доступної програмі пам’яті розміром 18 гб

Параметр -Xmx18g встановлює максимальний обсяг пам'яті (heap memory) розміром 18 гігабайт (гб), яка доступна програмі Java для зберігання об'єктів, даних та інших ресурсів, які використовуються під час виконання.

Лістинг виклику коду, що реалізує послідовний та паралельний алгоритми пакування для 1 млн. предметів та ємності корзини 1500 наведений на рисунку 4.5.

```java
public class Main {
    public static void main(String[] args) {
        int capacity = 1500;
        ItemsWarehouse itemsWarehouse;
        Dynamic dynamic;
        itemsWarehouse = new ItemsWarehouse( filePath: "d:\\items-1000000-200-20000.dat");
        dynamic = new Dynamic(itemsWarehouse, capacity);
        dynamic.pack();
        ParallelDynamic parallelDynamic = new ParallelDynamic(itemsWarehouse, capacity,  numThreads: 1);
        parallelDynamic.pack();
        parallelDynamic = new ParallelDynamic(itemsWarehouse, capacity,  numThreads: 2);
        parallelDynamic.pack();
        parallelDynamic = new ParallelDynamic(itemsWarehouse, capacity,  numThreads: 3);
        parallelDynamic.pack();
        parallelDynamic = new ParallelDynamic(itemsWarehouse, capacity,  numThreads: 4);
        parallelDynamic.pack();
        parallelDynamic = new ParallelDynamic(itemsWarehouse, capacity,  numThreads: 8);
        parallelDynamic.pack();
        parallelDynamic = new ParallelDynamic(itemsWarehouse, capacity,  numThreads: 16);
        parallelDynamic.pack();
```

Рис. 4.5 – Лістинг виклику коду, що реалізує послідовний та паралельний алгоритми

Файл items-1000000-200-20000.dat з вхідними даними був сформований та записаний на диск d:\.

Результат роботи послідовного алгоритму (початок та завершення виводу списку відібраних предметів) наведений на рисунку 4.6.

```
"C:\Program Files\Eclipse Adoptium\jdk-11.0.18.10-hotspot\bin\java.exe" -Xmx18g
 Послідовна               1000000          1500                14407
Зібраний рюкзак:
      403  Incredible Plastic Bench                    15019 грн.    1  кг
      527  Aerodynamic Concrete Plate                  14414 грн.    1  кг
      763  Sleek Aluminum Clock                        15850 грн.    1  кг
     1480  Incredible Steel Car                        18265 грн.    1  кг
     1786  Practical Wooden Coat                       16326 грн.    1  кг
     3958  Incredible Steel Wallet                     16679 грн.    1  кг
   997712  Mediocre Wool Bottle                        19919 грн.    1  кг
   998180  Incredible Concrete Gloves                  14324 грн.    1  кг
   998274  Gorgeous Granite Shirt                      14528 грн.    1  кг
   998306  Intelligent Linen Watch                     18568 грн.    1  кг
   999267  Lightweight Leather Wallet                  19478 грн.    1  кг
   999552  Gorgeous Bronze Chair                       15565 грн.    1  кг
Вага = 1500 Вартість  = 25573663
```

Рис. 4.6 – Результат роботи послідовного алгоритму

Результат роботи паралельного алгоритму для 1, 4 та 16 процесів наведений на рисунку 4.7.

```
 Паралельна        1       1000000          1500                25186
Зібраний рюкзак:
      403  Incredible Plastic Bench                    15019 грн.    1  кг
      527  Aerodynamic Concrete Plate                  14414 грн.    1  кг
      763  Sleek Aluminum Clock                        15850 грн.    1  кг
     1480  Incredible Steel Car                        18265 грн.    1  кг
     1786  Practical Wooden Coat                       16326 грн.    1  кг
     3958  Incredible Steel Wallet                     16679 грн.    1  кг
   997712  Mediocre Wool Bottle                        19919 грн.    1  кг
   998180  Incredible Concrete Gloves                  14324 грн.    1  кг
   998274  Gorgeous Granite Shirt                      14528 грн.    1  кг
   998306  Intelligent Linen Watch                     18568 грн.    1  кг
   999267  Lightweight Leather Wallet                  19478 грн.    1  кг
   999552  Gorgeous Bronze Chair                       15565 грн.    1  кг
Вага = 1500 Вартість  = 25573663
 Паралельна        2       1000000          1500                26128
Зібраний рюкзак:
      403  Incredible Plastic Bench                    15019 грн.    1  кг
      527  Aerodynamic Concrete Plate                  14414 грн.    1  кг
      763  Sleek Aluminum Clock                        15850 грн.    1  кг
     1480  Incredible Steel Car                        18265 грн.    1  кг
     1786  Practical Wooden Coat                       16326 грн.    1  кг
     3958  Incredible Steel Wallet                     16679 грн.    1  кг
   997712  Mediocre Wool Bottle                        19919 грн.    1  кг
   998180  Incredible Concrete Gloves                  14324 грн.    1  кг
   998274  Gorgeous Granite Shirt                      14528 грн.    1  кг
   998306  Intelligent Linen Watch                     18568 грн.    1  кг
   999267  Lightweight Leather Wallet                  19478 грн.    1  кг
   999552  Gorgeous Bronze Chair                       15565 грн.    1  кг
Вага = 1500 Вартість  = 25573663
 Паралельна       16       1000000          1500                27728
Зібраний рюкзак:
      403  Incredible Plastic Bench                    15019 грн.    1  кг
      527  Aerodynamic Concrete Plate                  14414 грн.    1  кг
      763  Sleek Aluminum Clock                        15850 грн.    1  кг
     1480  Incredible Steel Car                        18265 грн.    1  кг
     1786  Practical Wooden Coat                       16326 грн.    1  кг
     3958  Incredible Steel Wallet                     16679 грн.    1  кг
   997712  Mediocre Wool Bottle                        19919 грн.    1  кг
   998180  Incredible Concrete Gloves                  14324 грн.    1  кг
   998274  Gorgeous Granite Shirt                      14528 грн.    1  кг
   998306  Intelligent Linen Watch                     18568 грн.    1  кг
   999267  Lightweight Leather Wallet                  19478 грн.    1  кг
   999552  Gorgeous Bronze Chair                       15565 грн.    1  кг
Вага = 1500 Вартість  = 25573663

Process finished with exit code 0
```

Рис. 4.7 – Результат роботи паралельного алгоритму

Порівняння результатів роботи програми, що реалізовує алгоритми пакування рюкзака методом перебору та методом динамічного програмування (паралельна реалізація) на невеликих обсягах даних, а також послідовний та паралельний алгоритма на великих обсягах даних та для різної кількості потоків показує, що відібрані предмети, вага та вартість співпадають.

Отже, паралельний алгоритм пакування рюкзака методом динамічного програмування розроблений та реалізований мовою програмування Java коректно.

5 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ПАРАЛЕЛЬНИХ ОБЧИСЛЕНЬ АЛГОРИТМУ

Проведемо аналіз швидкодії та ефективності паралельної реалізації алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування.

Для цього виміряємо та порівняємо час виконання послідовної та паралельної реалізацій алгоритму на різних об’ємах даних (різній кількості предметів та обсязі рюкзака), а також на різній кількості потоків для паралельної реалізації.

Випробування алгоритмів будемо проводити з діапазоном кількості предметів від 1000 до 1000000, обсязі рюкзака від 1000 до 100000 та кількості потоків 2, 4, 8 (процесор має 8 ядер).

З метою забезпечення лаконічності представлення результатів виклик методу друку набору елементів **knapSack.printResult()** відключемо.

Результати випробувань для 100000 предметів та діапазону об’ємів рюкзака від 1000 до 45000 наведені на рисунку 5.1, для 10000 предметів та діапазону об’ємів рюкзака від 1000 до 100000 – на рисунку 5.2, для 1000 предметів та діапазону об’ємів рюкзака від 1000 до 100000 – на рисунку 5.3.

```
"C:\Program Files\Eclipse Adoptium\jdk-11.0.18.10-hotspot\bin\java.exe"
Реалізація     Threads        Items         Capacity              T,мс
 Послідовна                  100000         1000                  1120
 Паралельна        2         100000         1000                  2854
 Паралельна        4         100000         1000                  1984
 Паралельна        8         100000         1000                  2799
 Послідовна                  100000         2000                  1935
 Паралельна        2         100000         2000                  2946
 Паралельна        4         100000         2000                  2792
 Паралельна        8         100000         2000                  2994
 Послідовна                  100000         3000                  2601
 Паралельна        2         100000         3000                  3415
 Паралельна        4         100000         3000                  2928
 Паралельна        8         100000         3000                  3424
 Послідовна                  100000         4000                  3385
 Паралельна        2         100000         4000                  3729
 Паралельна        4         100000         4000                  3880
 Паралельна        8         100000         4000                  3947
 Послідовна                  100000         5000                  4293
 Паралельна        2         100000         5000                  4262
 Паралельна        4         100000         5000                  4578
 Паралельна        8         100000         5000                  4332
 Послідовна                  100000         7000                  6038
 Паралельна        2         100000         7000                  5190
 Паралельна        4         100000         7000                  5767
 Паралельна        8         100000         7000                  5202
 Послідовна                  100000         10000                  7053
 Паралельна        2         100000         10000                  6428
 Паралельна        4         100000         10000                  5969
 Паралельна        8         100000         10000                  6073
 Послідовна                  100000         20000                 15694
 Паралельна        2         100000         20000                 10401
 Паралельна        4         100000         20000                  9653
 Паралельна        8         100000         20000                  9229
 Послідовна                  100000         30000                 21039
 Паралельна        2         100000         30000                 13825
 Паралельна        4         100000         30000                 12660
 Паралельна        8         100000         30000                 12897
 Послідовна                  100000         40000                 27481
 Паралельна        2         100000         40000                 17404
 Паралельна        4         100000         40000                 15589
 Паралельна        8         100000         40000                 16290
 Послідовна                  100000         45000                 34879
 Паралельна        2         100000         45000                 24621
 Паралельна        4         100000         45000                 20468
 Паралельна        8         100000         45000                 18221

Process finished with exit code 0
```

Рис. 5.1 – Результати випробувань паралельного алгоритму для 100000 предметів

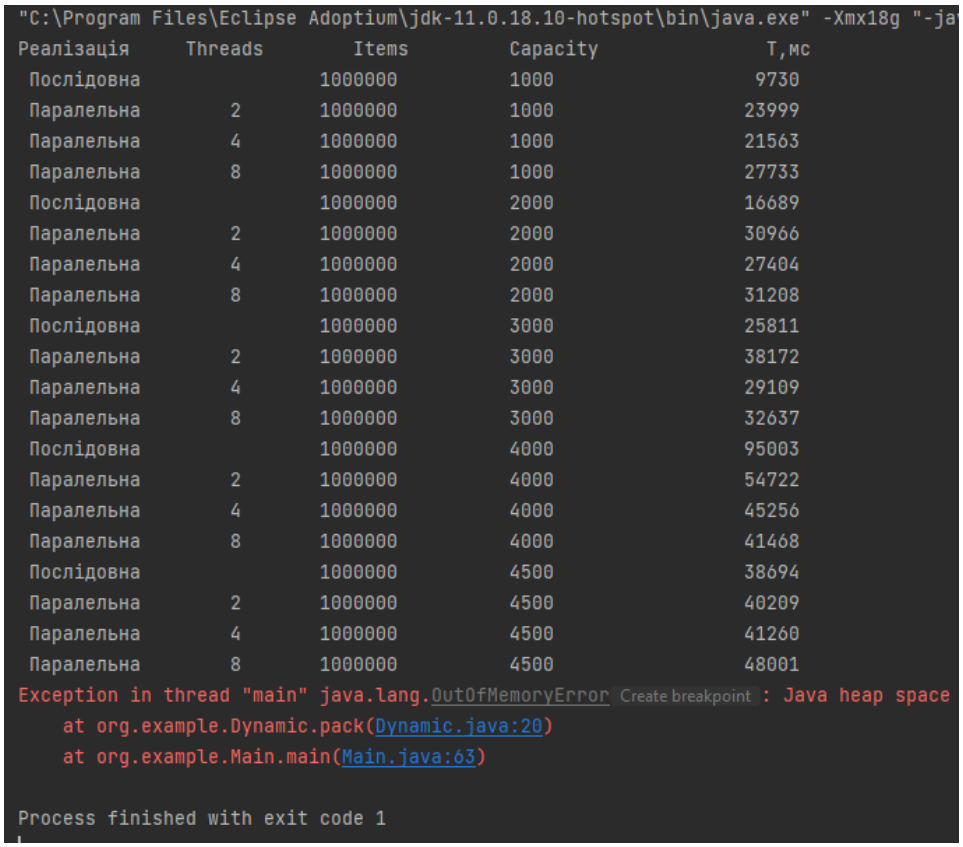
Реалізація	Threads	Items	Capacity	T,мс
Послідовна		10000	1000	163
Паралельна	2	10000	1000	477
Паралельна	4	10000	1000	409
Паралельна	8	10000	1000	358
Послідовна		10000	2000	218
Паралельна	2	10000	2000	368
Паралельна	4	10000	2000	278
Паралельна	8	10000	2000	303
Послідовна		10000	3000	224
Паралельна	2	10000	3000	339
Паралельна	4	10000	3000	258
Паралельна	8	10000	3000	314
Послідовна		10000	4000	340
Паралельна	2	10000	4000	403
Паралельна	4	10000	4000	388
Паралельна	8	10000	4000	389
Послідовна		10000	5000	424
Паралельна	2	10000	5000	428
Паралельна	4	10000	5000	418
Паралельна	8	10000	5000	431
Послідовна		10000	7000	567
Паралельна	2	10000	7000	504
Паралельна	4	10000	7000	538
Паралельна	8	10000	7000	510
Послідовна		10000	10000	608
Паралельна	2	10000	10000	532
Паралельна	4	10000	10000	551
Паралельна	8	10000	10000	503
Послідовна		10000	20000	1211
Паралельна	2	10000	20000	883
Паралельна	4	10000	20000	967
Паралельна	8	10000	20000	920
Послідовна		10000	30000	2101
Паралельна	2	10000	30000	1452
Паралельна	4	10000	30000	1264
Паралельна	8	10000	30000	1484
Послідовна		10000	40000	2879
Паралельна	2	10000	40000	1884
Паралельна	4	10000	40000	1634
Паралельна	8	10000	40000	2000
Послідовна		10000	50000	3385
Паралельна	2	10000	50000	2555
Паралельна	4	10000	50000	1993
Паралельна	8	10000	50000	2165
Послідовна		10000	60000	4455
Паралельна	2	10000	60000	2828
Паралельна	4	10000	60000	2199
Паралельна	8	10000	60000	2115
Послідовна		10000	70000	5071
Паралельна	2	10000	70000	3071
Паралельна	4	10000	70000	2551
Паралельна	8	10000	70000	2446
Послідовна		10000	80000	5752
Паралельна	2	10000	80000	3618
Паралельна	4	10000	80000	2627
Паралельна	8	10000	80000	2805
Послідовна		10000	90000	6176
Паралельна	2	10000	90000	3973
Паралельна	4	10000	90000	2891
Паралельна	8	10000	90000	3077
Послідовна		10000	100000	6917
Паралельна	2	10000	100000	4206
Паралельна	4	10000	100000	3228
Паралельна	8	10000	100000	3093

Рис. 5.2 – Результати випробувань параллельного алгоритму для 10000 предметів

Реалізація	Threads	Items	Capacity	T,мс
Послідовна		1000	1000	40
Паралельна	2	1000	1000	105
Паралельна	4	1000	1000	54
Паралельна	8	1000	1000	76
Послідовна		1000	2000	112
Паралельна	2	1000	2000	53
Паралельна	4	1000	2000	52
Паралельна	8	1000	2000	61
Послідовна		1000	3000	71
Паралельна	2	1000	3000	48
Паралельна	4	1000	3000	55
Паралельна	8	1000	3000	53
Послідовна		1000	4000	36
Паралельна	2	1000	4000	54
Паралельна	4	1000	4000	58
Паралельна	8	1000	4000	63
Послідовна		1000	5000	42
Паралельна	2	1000	5000	53
Паралельна	4	1000	5000	59
Паралельна	8	1000	5000	53
Послідовна		1000	7000	59
Паралельна	2	1000	7000	65
Паралельна	4	1000	7000	64
Паралельна	8	1000	7000	61
Послідовна		1000	10000	68
Паралельна	2	1000	10000	59
Паралельна	4	1000	10000	75
Паралельна	8	1000	10000	83
Послідовна		1000	20000	122
Паралельна	2	1000	20000	85
Паралельна	4	1000	20000	88
Паралельна	8	1000	20000	82
Послідовна		1000	30000	211
Паралельна	2	1000	30000	114
Паралельна	4	1000	30000	107
Паралельна	8	1000	30000	136
Послідовна		1000	40000	260
Паралельна	2	1000	40000	193
Паралельна	4	1000	40000	150
Паралельна	8	1000	40000	141
Послідовна		1000	50000	337
Паралельна	2	1000	50000	255
Паралельна	4	1000	50000	155
Паралельна	8	1000	50000	181
Послідовна		1000	60000	410
Паралельна	2	1000	60000	232
Паралельна	4	1000	60000	158
Паралельна	8	1000	60000	168
Послідовна		1000	70000	424
Паралельна	2	1000	70000	243
Паралельна	4	1000	70000	248
Паралельна	8	1000	70000	263
Послідовна		1000	80000	569
Паралельна	2	1000	80000	325
Паралельна	4	1000	80000	237
Паралельна	8	1000	80000	205
Послідовна		1000	90000	545
Паралельна	2	1000	90000	363
Паралельна	4	1000	90000	207
Паралельна	8	1000	90000	217
Послідовна		1000	100000	635
Паралельна	2	1000	100000	326
Паралельна	4	1000	100000	328
Паралельна	8	1000	100000	337

Рис. 5.3 – Результати випробувань параллельного алгоритму для 1000 предметів

Результати випробувань для 1000000 предметів та діапазону об’ємів рюкзака від 1000 до 4500 наведені на рисунку 5.4



```
"C:\Program Files\Eclipse Adoptium\jdk-11.0.18.10-hotspot\bin\java.exe" -Xmx18g "-ja
Реалізація     Threads        Items         Capacity              T,мс
 Послідовна               1000000         1000                  9730
 Паралельна        2      1000000         1000                 23999
 Паралельна        4      1000000         1000                 21563
 Паралельна        8      1000000         1000                 27733
 Послідовна               1000000         2000                 16689
 Паралельна        2      1000000         2000                 30966
 Паралельна        4      1000000         2000                 27404
 Паралельна        8      1000000         2000                 31208
 Послідовна               1000000         3000                 25811
 Паралельна        2      1000000         3000                 38172
 Паралельна        4      1000000         3000                 29109
 Паралельна        8      1000000         3000                 32637
 Послідовна               1000000         4000                 95003
 Паралельна        2      1000000         4000                 54722
 Паралельна        4      1000000         4000                 45256
 Паралельна        8      1000000         4000                 41468
 Послідовна               1000000         4500                 38694
 Паралельна        2      1000000         4500                 40209
 Паралельна        4      1000000         4500                 41260
 Паралельна        8      1000000         4500                 48001
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space
    at org.example.Dynamic.pack(Dynamic.java:20)
    at org.example.Main.main(Main.java:63)

Process finished with exit code 1
```

Рис. 5.4 – Результати випробувань параллельного алгоритму для 1000000 предметів

Під час виконання експериментів діяли обмеження на обсяг даних, що оброблюються: при виконанні програми з параметром -Xmx18g (на комп’ютері встановлено 22 Гб оперативної пам’яті) для обсягу даних, що визначається формулою countItems x capacity та перевищує значення 4500000000 виникає помилка OutOfMemoryError.

З отриманих результатів видно, що переваги паралельного алгоритму над послідовним спостерігаються при збільшенні обсягу рюкзака, що пояснюється описаним в пункті 4.3 алгоритмом розділення даних на потоки.

Задане у завданні на курсову роботу прискорення паралельного алгоритму над послідовним більше, ніж 1.2, досягається незалежно від кількості предметів у сховищі для об’єму рюкзаку більшому, ніж 10000 (таблиця 5.1).

Реалізація	Кільк. потоків	Ємність рюкзака	Час, мс	Прискорення
Послідовна	1	1000	163	
Паралельна	2	1000	477	0,34
Паралельна	4	1000	409	0,40
Паралельна	8	1000	358	0,46
Послідовна	1	2000	218	
Паралельна	2	2000	368	0,59
Паралельна	4	2000	278	0,78
Паралельна	8	2000	303	0,72
Послідовна	1	3000	224	
Паралельна	2	3000	339	0,66
Паралельна	4	3000	258	0,87
Паралельна	8	3000	314	0,71
Послідовна	1	4000	340	
Паралельна	2	4000	403	0,84
Паралельна	4	4000	388	0,88
Паралельна	8	4000	389	0,87
Послідовна	1	5000	424	
Паралельна	2	5000	428	0,99
Паралельна	4	5000	418	1,01
Паралельна	8	5000	431	0,98
Послідовна	1	7000	567	
Паралельна	2	7000	504	1,13
Паралельна	4	7000	538	1,05
Паралельна	8	7000	510	1,11
Послідовна	1	10000	608	
Паралельна	2	10000	532	1,14
Паралельна	4	10000	551	1,10
Паралельна	8	10000	503	1,21
Послідовна	1	20000	1211	
Паралельна	2	20000	883	1,37
Паралельна	4	20000	967	1,25
Паралельна	8	20000	920	1,32
Послідовна	1	30000	2101	
Паралельна	2	30000	1452	1,45
Паралельна	4	30000	1264	1,66
Паралельна	8	30000	1484	1,42

Продовження таблиці 5.1.

Реалізація	Кільк. потоків	Ємність рюкзака	Час, мс	Прискорення
Послідовна	1	40000	2879	
Паралельна	2	40000	1884	1,53
Паралельна	4	40000	1634	1,76
Паралельна	8	40000	2000	1,44
Послідовна	1	50000	3385	
Паралельна	2	50000	2555	1,32
Паралельна	4	50000	1993	1,70
Паралельна	8	50000	2165	1,56
Послідовна	1	60000	4455	
Паралельна	2	60000	2828	1,58
Паралельна	4	60000	2199	2,03
Паралельна	8	60000	2115	2,11
Послідовна	1	70000	5071	
Паралельна	2	70000	3071	1,65
Паралельна	4	70000	2551	1,99
Паралельна	8	70000	2446	2,07
Послідовна	1	80000	5752	
Паралельна	2	80000	3618	1,59
Паралельна	4	80000	2627	2,19
Паралельна	8	80000	2805	2,05
Послідовна	1	90000	6176	
Паралельна	2	90000	3973	1,55
Паралельна	4	90000	2891	2,14
Паралельна	8	90000	3077	2,01
Послідовна	1	100000	6917	
Паралельна	2	100000	4206	1,64
Паралельна	4	100000	3228	2,14
Паралельна	8	100000	3093	2,24

Табл. 5.1 Розрахунок прискорення параллельного алгоритму для 10000 предметів

Графіки залежності прискорення від обсягу рюкзака для 10000 предметів зображені на рисунку 5.5.

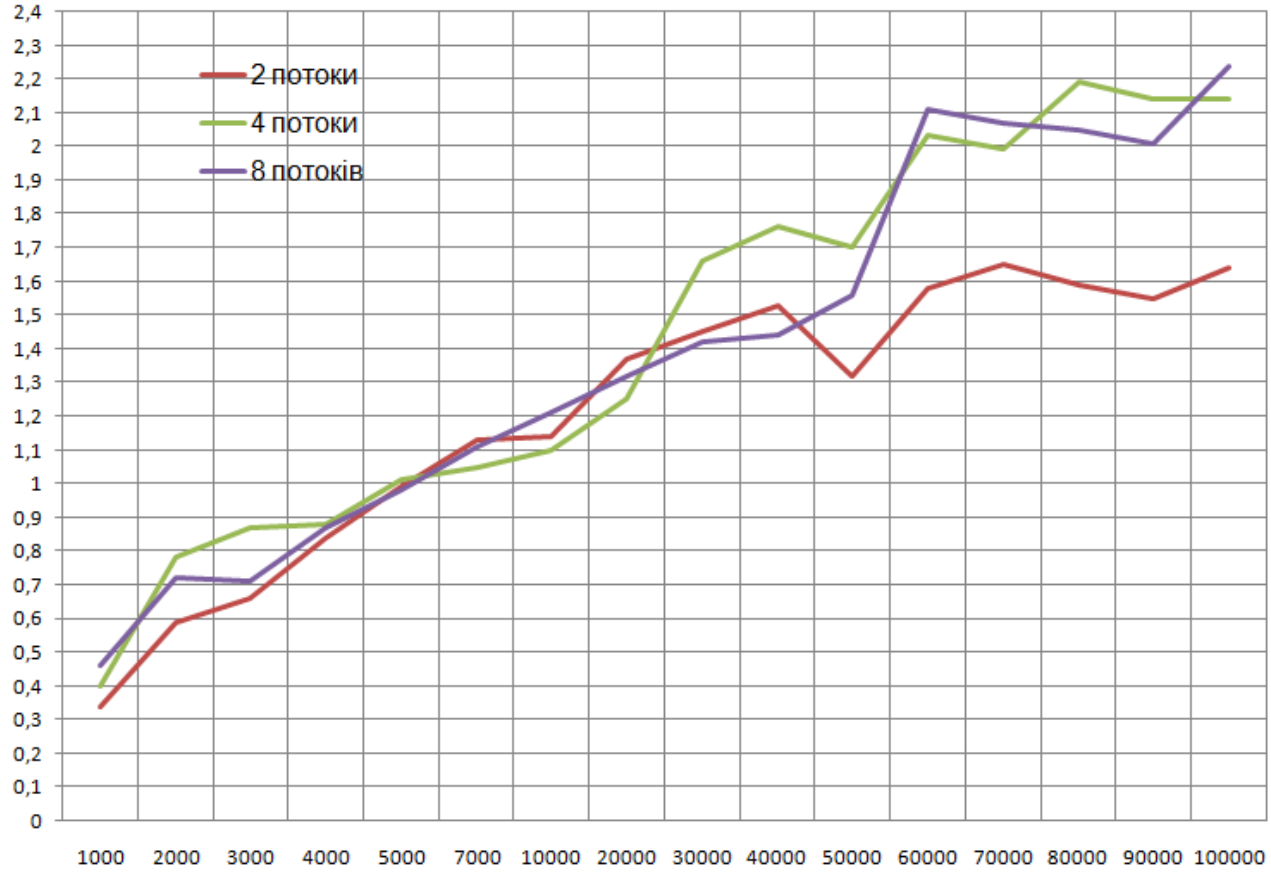


Рис. 5.5 – Графік залежності прискорення від обсягу рюкзака

Досягнутий результат вважається гарним, оскільки основною метою распаралелювання було зменшення часу розрахунків. Оскільки ємність рюкзака – це один з двох параметрів, від яких суттєво залежить час розрахунків, то саме при його збільшенні є виправданим застосовувати паралельний алгоритм, який прискорює розрахунки.

При малих обсягах даних вибір алгоритму для розрахунків є несуттєвим, оскільки загальний час розрахунків на сучасному комп’ютері надто малий.

ВИСНОВКИ

У даній курсовій роботі була реалізована паралельна версія алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування.

За результатами роботи можна зробити деякі висновки щодо ефективності виконання паралельних обчислень.

Обрана мова програмування Java відмінно підходить для розробки програмного забезпечення, що реалізує задачі з багатопотоковістю.

У результаті проведених досліджень дійшла до висновку, що використання паралельних процесів взмозі суттєво прискорити обчислення з великими обсягами даних та допомогти у розв’язанні більш складнішіх задач. Дослідження були проведені на алгоритмі пакування рюкзака методом динамічного програмування і при паралельній реалізації алгоритму на великих обсягах даних отримано реальний результат, який підтверджує цей висновок.

Разом з цим, слід відмітити, що розв’язання задачі про пакування рюкзака методом динамічного програмування вимагає обчислення значень великої кількості проміжних результатів, які залежать від попередніх обчислень. Оскільки кожне обчислення залежить від результатів попереднього кроку, паралельне вирішення цієї задачі ускладнюється, а в окремих випадках може бути недоцільним і непрактичним. У загальному випадку, распаралелювання задачі про пакування рюкзака методом динамічного програмування характеризується наступними проблемами:

1) Залежність від попередніх обчислень: У методі динамічного програмування, для обчислення значення в кожній клітинці таблиці потрібно мати значення з попередніх клітинок. Це створює залежність між обчисленнями, що робить складним розпаралелювання цих обчислень.

2) Синхронізація та координація: Розпаралелювання вимагає синхронізації та координації між потоками/процесами, щоб вони могли обмінюватись необхідними даними та результатами. У задачі про пакування рюкзака це може

створювати додаткову складність та затрати на управління потоками/процесами.

3) Масштабування: Розмір задачі про пакування рюкзака залежить від кількості предметів та максимальної ваги рюкзака. Якщо задача є достатньо великою, розпаралелювання може призвести до надмірної витрати ресурсів на керування потоками/процесами без суттєвого збільшення продуктивності.

Таким чином, для загального випадку задачі про пакування рюкзака методом динамічного програмування розпаралелювання може бути складним і неефективним.

Однак, варто зазначити, що для деяких варіантів задачі про пакування рюкзака можливі розпаралелені підходи. Наприклад, якщо задача може бути розділена на незалежні підзадачі, то розпаралелювання може бути доцільним, і реалізація алгоритму в даній курсовій роботі з досягнутими значеннями прискорень більше ніж 1.2 (а в окремих випадках – більше 2) – тому підтвердження.

Основне завдання курсової роботи – паралельна реалізація мовою програмування Java алгоритму розв’язання задачі про пакування рюкзака методом динамічного програмування з прискоренням понад 1.2, цей пункт виконаний, отже алгоритм є успішним.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Knapsack problem using Greedy-method in Java – [Електронний ресурс] / Sanskar Dwivedi – 2023. – Режим доступу до ресурсу: https://www.codespeedy.com/knapsack-problem-using-greedy-method-in-java/
2. Meet in the middle – [Електронний ресурс] / GeeksForGeeks – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/meet-in-the-middle/
3. What is meet in the middle algorithm w.r.t. competitive programming? – [Електронний ресурс] / S. A. Kumar – 2023. – Режим доступу до ресурсу: https://www.quora.com/What-is-meet-in-the-middle-algorithm-w-r-t-competitive-programming
4. 0/1 Knapsack using Branch and Bound – [Електронний ресурс] / GeeksForGeeks – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/
5. 0/1 Knapsack Problem – [Електронний ресурс] / GeeksForGeeks – 2023. – Режим доступу до ресурсу: https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/
6. What Is Parallel Programming and Multithreading? – [Електронний ресурс] / Richard Bellairs – 2023. – Режим доступу до ресурсу: https://www.perforce.com/blog/qac/multithreading-parallel-programming-c-cpp
7. Python Documentation – [Електронний ресурс] / Python Software Foundation – 2023. – Режим доступу до ресурсу: https://docs.python.org/3/library/threading.html
8. Concurrency in modern programming languages: Golang – [Електронний ресурс] / Deepu K Sasidharan – 2023. – Режим доступу до ресурсу: https://deepu.tech/concurrency-in-modern-languages-go/
9. Scala Multithreading – [Електронний ресурс] / JavaTPoint – 2023. – Режим доступу до ресурсу:

https://www.javatpoint.com/scala-multithreading

10. Java Documentation – [Електронний ресурс] / Oracle – 2023. – Режим доступу до ресурсу:
https://docs.oracle.com/javase/tutorial/essential/concurrency/
11. Comparison of Parallel Algorithms for the 0-1 Knapsack Problem on Networked Computers / – Rebecca A Hunt, A Thesis in the Field of Information Technology for the Degree of Master of Liberal Arts in Extension Studies, 2005.
12. Dynamic programming parallel implementations for the knapsack problem – [Електронний ресурс] / Rumen Andonov, Frédéric Raimbault, Patrice Quinton – 2006. – Режим доступу до ресурсу:
https://inria.hal.science/inria-00074634/document

ДОДАТКИ

Повний код усієї роботи знаходиться на GitHub по посиланню https://github.com/DmIrina/Knapsack-Problem--Parallel-Dynamic

Додаток А. Лістинг коду класів PackBackpack, Item, ItemWarehouse,

```
public interface PackBackpack {
  void pack();
}
```

```
import java.io.Serializable;

public class Item implements Serializable {
  private int id;
  private String name;
  private int weight;
  private int price;

  public Item(int id, String name, int weight, int price) {
    this.id = id;
    this.name=name;
    this.price=price;
    this.weight=weight;
  }

  public int getId() {
    return id;
  }

  public String getName() {
    return name;
  }

  public int getPrice() {
    return price;
  }

  public int getWeight() {
    return weight;
  }

  public void printItem() {
    System.out.printf("%9d %-45s %4d грн. %4d кг\n", (getId() + 1), getName(),
getPrice(), getWeight());
  }
}
```

```
import com.github.javafaker.Faker;

import java.io.*;
import java.util.ArrayList;
import java.util.Random;

public class ItemsWarehouse implements Serializable {
  private ArrayList<Item> items;

  public ItemsWarehouse(int countItems, int maxWeight, int maxPrice) {
    items = generate(countItems, maxWeight, maxPrice);
  }
```

```
  public ItemsWarehouse(String filePath) {
    this.items = loadItems(filePath);
  }

  public ArrayList<Item> getItems() {
    return items;
  }

  public Item getItem(int id) {
    return items.get(id);
  }

  public int getWeight(int id) {
    return items.get(id).getWeight();
  }

  public int getPrice(int id) {
    return items.get(id).getPrice();
  }

  public String getName(int id) {
    return items.get(id).getName();
  }

  private ArrayList<Item> generate(int count, int maxWeight, int maxPrice) {
    ArrayList<Item> array = new ArrayList<>();
    Faker faker = new Faker();
    Random randomWeight = new Random();
    Random randomPrice = new Random();

    for (int i = 0; i < count; i++) {
      String name = faker.commerce().productName();
      int weight = randomWeight.nextInt(maxWeight) + 1;
      int price = randomPrice.nextInt(maxPrice) + 1;
      array.add(new Item(i, name, weight, price));
    }
    return array;
  }

  public void saveItems(String path) {
    try {
      FileOutputStream fileOut = new FileOutputStream(path);
      ObjectOutputStream objectOut = new ObjectOutputStream(fileOut);
      objectOut.writeObject(items);
      objectOut.close();
      fileOut.close();
      System.out.println("Об'єкти були серіалізовані та збережені у файлі " +
path);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public ArrayList<Item> loadItems(String filePath) {
    try {
      FileInputStream fileIn = new FileInputStream(filePath);
      ObjectInputStream objectIn = new ObjectInputStream(fileIn);
      ArrayList<Item> deserializedItems = (ArrayList<Item>)
objectIn.readObject();
      objectIn.close();
      fileIn.close();
      return deserializedItems;
    } catch (Exception e) {
```

```
      e.printStackTrace();
    }
    return null;
  }

  public void printItems() {
    for (Item obj : items) {
      // Item k = obj;
      System.out.printf("%9d %-45s %4d грн. %4d кг\n", (obj.getId() + 1),
obj.getName(), obj.getPrice(), obj.getWeight());
      // obj.printItem();
    }
  }
}
```

Додаток Б. Лістинг коду класів Knapsack, Helper

```
import java.util.ArrayList;
import java.util.Collections;

public class KnapSack {
  private int totalPrice;
  private int totalWight;
  private ArrayList<Integer> packedItems;
  private int capacity;

  public KnapSack(int capacity, int totalPrice, int totalWight,
ArrayList<Integer> packedItems) {
    this.capacity = capacity;
    this.totalWight = totalWight;
    this.totalPrice = totalPrice;
    this.packedItems = packedItems;
  }

  public void printResult(ItemsWarehouse itemsWarehouse) {
    System.out.println("Зібраний рюкзак:");
    for (int num : packedItems) {
      itemsWarehouse.getItem(num).printItem();
    }
    System.out.println("Вага = " + totalWight + " Вартість = " + totalPrice);
  }
}
```

```
import java.util.ArrayList;
import java.util.Collections;

public class Helper {
  // знайти список вибраних предметів
  public static KnapSack findSelectedItems(int[][] dt, ItemsWarehouse
itemsWarehouse, int capacity) {
    int totalPrice = 0;
    int totalWight = 0;
    ArrayList<Integer> packedItems = new ArrayList<>();
    int n = itemsWarehouse.getItems().size();
    while (n > 0 && capacity > 0) {
      if (dt[n][capacity] != dt[n - 1][capacity]) {
        packedItems.add(n - 1); // Додавання індексу вибраного предмета
        totalPrice += itemsWarehouse.getPrice(n - 1);
        totalWight += itemsWarehouse.getWeight(n - 1);
        capacity -= itemsWarehouse.getWeight(n - 1);
      }
      n--;
    }
    Collections.reverse(packedItems);
    return new KnapSack(capacity, totalPrice, totalWight, packedItems);
  }
}
```

Додаток В. Лістинг коду роботи послідовного алгоритму

```
public class Dynamic implements PackBackpack {
 private ItemsWarehouse itemsWarehouse;
 private int countItems;
 private int capacity;   // max weight of knapsack

 public Dynamic(ItemsWarehouse itemsWarehouse, int capacity) {
  this.itemsWarehouse = itemsWarehouse;
  countItems = itemsWarehouse.getItems().size();
  this.capacity = capacity;
 }
 @Override
 public void pack() {
  System.out.println("\nПослідовна реалізація алгоритму пакування рюкзака
методом динамічного програмування:");
  long startTime = System.currentTimeMillis();
// таблиця динамічного програмування
  int[][] dt = new int[countItems + 1][capacity + 1];
  // Заповнення таблиці A для знаходження оптимального рішення
  for (int i = 1; i <= countItems; i++) {  // i - номер предмета
   for (int j = 1; j <= capacity; j++) { // j - місткості рюкзака
// Вага предмета менша за вміщувану в рюкзак
    if (itemsWarehouse.getWeight(i - 1) <= j) {
     // Вартість включення поточного предмета
     int includePrice = itemsWarehouse.getPrice(i - 1) + dt[i - 1][j -
itemsWarehouse.getWeight(i - 1)];
// Вартість виключення поточного предмета
     int excludePrice = dt[i - 1][j];
     if (includePrice > excludePrice) {
// Вибір включення предмета
      dt[i][j] = includePrice;
     } else {
// Вибір виключення предмета
      dt[i][j] = excludePrice;
     }
    } else {
// Вага предмета більша за вміщувану ємність, пропускаємо предмет
     dt[i][j] = dt[i - 1][j];
    }
   }
  }

  KnapSack knapSack = Helper.findSelectedItems(dt, itemsWarehouse, capacity);
  long endTime = System.currentTimeMillis();
  long executionTime = endTime - startTime;
  System.out.printf(" Послідовна    %7d   %4d   %12d\n", countItems, capacity,
executionTime);
  knapSack.printResult(itemsWarehouse);
 }
}
```

Додаток Г. Лістинг коду реалізації розв'язання задачі методом перебору

```
public class Bruteforce implements PackBackpack {
  private ItemsWarehouse itemsWarehouse;
  private int countItems;
  private int maxW;      // max weight of knapsack
  int totalPrice = 0;
  int totalWight = 0;
  long maxSelection = 0;

  public Bruteforce(ItemsWarehouse itemsWarehouse, int maxW) {
    this.itemsWarehouse = itemsWarehouse;
    countItems = itemsWarehouse.getItems().size();
    this.maxW = maxW;
  }

  @Override
  public void pack() {
    long startTime = System.currentTimeMillis();
    long countCombination = (long) Math.pow(2, countItems);

    // пошук максимуму перебором
    // біти selection показують, чи присутній предмет у рюкзаку
    // (1 - беремо, 0 - ні: 00011 - беремо 2 предмета справа)
    for (long selection = 0; selection < countCombination; selection++) {
      int price = statePrice(selection);
      int wight = stateWeight(selection);
      if (wight <= maxW) {
        if (totalPrice < price) {
          totalPrice = price;
          totalWight = wight;
          maxSelection = selection;
        }
      }
    }

    System.out.println("\nМетод перебору: використовуємо виключно для контролю
на невеликих обсягах даних.");
    System.out.println("Зібраний рюкзак:");
    long numberOfBit = 1;
    for (int i = 0; i < countItems; i++) {
      if ((numberOfBit & maxSelection) > 0) {
        itemsWarehouse.getItem(i).printItem();
      }
      numberOfBit <<= 1;
    }
    long endTime = System.currentTimeMillis();
    long executionTime = endTime - startTime;
    System.out.printf("countItems = %7d capacity = %4d - час виконання: %12d
мс\n", countItems, maxW, executionTime );
    System.out.println("Вага = " + totalWight + " Вартість = " + totalPrice);
  }

  // рахуємо вартість усіх обраних предметів для варіанту state
  // (selection = 3 = 00011 - вказує на 2 останні предмети)
  private int statePrice(long selection) {

// перший (молодший) біт = перший предмет
    long numberOfBit = 1;
```

```
    int price = 0;
    for (int i = 0; i < countItems; i++) {
      if ((numberOfBit & selection) != 0) {

// поразрядної операції "I" -

// кожен біт у результуючому виразі буде встановлений,

// якщо він встановлений в обох операндах.
        price += itemsWarehouse.getPrice(i);
      }

// зсув вліво на 1 біт - переходимо до наступного
      numberOfBit <<= 1;                }
    return price;
  }

  private int stateWeight(long selection) {
    long numberOfBit = 1;
    int weight = 0;
    for (int i = 0; i < countItems; i++) {
      if ((numberOfBit & selection) != 0) {
        weight += itemsWarehouse.getWeight(i);
      }
      numberOfBit <<= 1;
    }
    return weight;
 }

}
```

Додаток Ґ. Лістинг коду роботи паралельного алгоритму пакування рюкзака методом динамічного програмування

```
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class ParallelDynamic implements PackBackpack{
 private ItemsWarehouse itemsWarehouse; // Сховище предметів
 private int countItems;     // Кількість предметів
 private int capacity;     // Максимальна вага рюкзака
 private int numThreads = 1;   // Кількість потоків за замовчуванням

 public ParallelDynamic(ItemsWarehouse itemsWarehouse, int capacity, int
numThreads) {
  this.itemsWarehouse = itemsWarehouse;
  this.countItems = itemsWarehouse.getItems().size();
  this.capacity = capacity;
  this.numThreads = numThreads;
 }

 @Override
 public void pack() {

 // System.out.println("\nПаралельна реалізація алгоритму пакування рюкзака");
  // Створення пулу потоків для виконання паралельних обчислень
  ExecutorService pool = Executors.newFixedThreadPool(numThreads);
  long startTime = System.currentTimeMillis();
// часткова ємність для кожного потоку
  int partCapacity = (capacity + 1) / numThreads;
// Масив для збереження результатів обчислень
  int[][] dt = new int[countItems + 1][capacity + 1];

  for (int i = 1; i <= countItems; i++) {   // i - номер предмета
// Лічильник для синхронізації потоків
   CountDownLatch latch = new CountDownLatch(numThreads);
   for (int stage = 0; stage < numThreads; stage++) {// цикл по потоках
    int itemIndex = i;  // Номер поточного предмета
// Ліва межа частини рюкзака для поточного потоку
    int partLeftIndex = stage * partCapacity;
    int partRightIndex;
    if (stage == numThreads - 1) {
// Права межа останньої частини рюкзака
     partRightIndex = capacity;
    } else {
// Права межа для поточної частини рюкзака
     partRightIndex = partLeftIndex + partCapacity - 1;
    }

    pool.submit(() -> {
     calculate(dt, latch, itemIndex, partLeftIndex, partRightIndex);
    });
   }
   try {
    latch.await();  // Очікування завершення всіх потоків
   } catch (InterruptedException e) {
    e.printStackTrace();
   }
  }
  pool.shutdown();
```

```
  try {
   pool.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
  } catch (InterruptedException e) {
   e.printStackTrace();
  }


// відновити список вибраних предметів
  KnapSack knapSack = Helper.findSelectedItems(dt, itemsWarehouse, capacity);
  long endTime = System.currentTimeMillis();
  long executionTime = endTime - startTime;
  System.out.printf(" Паралельна  %2d  %7d   %4d   %12d\n", numThreads,
countItems, capacity, executionTime);
  knapSack.printResult(itemsWarehouse); // друк змісту рюкзака
 }

 private void calculate(int[][] dt, CountDownLatch latch, int itemIndex, int
partLeftIndex, int partRightIndex) {
  // обчислювальна робота виконується паралельно у пулі потоків
// j - місткості рюкзака
  for (int j = partLeftIndex; j <= partRightIndex; j++) {
// Вага предмета менша за вміщувану в рюкзак
   if (itemsWarehouse.getWeight(itemIndex - 1) <= j) {
    int includePrice = itemsWarehouse.getPrice(itemIndex - 1) + dt[itemIndex -
1][j - itemsWarehouse.getWeight(itemIndex - 1)];
// Вартість без включення поточного предмета
    int excludePrice = dt[itemIndex - 1][j];
    if (includePrice > excludePrice) {
     dt[itemIndex][j] = includePrice;          // покласти предмет
    } else {
     dt[itemIndex][j] = excludePrice;  // не класти предмет
    }
   } else {
// Вага предмета більша за вміщувану місткість, пропускаємо предмет
    dt[itemIndex][j] = dt[itemIndex - 1][j];
   }
  }
  latch.countDown();
 }
}
```

ЗАВДАННЯ

1. Виконати огляд існуючих реалізацій алгоритму, послідовних та паралельних, з відповідними посиланнями на джерела інформації (статті, книги, електронні ресурси). Зробити висновок про актуальність дослідження.

2. Виконати розробку послідовного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Дослідити швидкодію алгоритму при зростанні складності обчислень та зробити висновки про необхідність паралельної реалізації алгоритму.

3. Виконати розробку паралельного алгоритму у відповідності до варіанту завдання та обраного програмного забезпечення для реалізації. Забезпечити зручне введення даних для початку обчислень.

4. Виконати тестування алгоритму, що доводить коректність результатів обчислень.

5. Виконати дослідження швидкодії алгоритму при зростанні кількості даних для обчислень.

6. Виконати експериментальне дослідження прискорення розробленого алгоритму при зростанні кількості даних для обчислень. Реалізація алгоритму вважається успішною, якщо прискорення більше 1,2.

7. Зробити висновки про переваги паралельної реалізації обчислень для алгоритму, що розглядається у курсовій роботі, та програмних засобів, які використовувались.