Дніпропетровський ліцей інформаційних технологій при  
Дніпропетровському національному університеті

Кафедра Інформатики

**КУРСОВА РОБОТА**

**Тема:** Класифікація задача динамічного програмування за обраними методами їх вирішення

**Виконав:**

Ліцеїст 10-Г класу

*Мунтян Даниїл Миколайович*

**Керівники:**

*Ентін Йосиф Абрамович*

**Робота допущена до захисту**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  
Ентін Йосиф Абрамович

Дніпропетровськ

(2017)

ЗМІСТ

[ВСТУП 3](#_Toc476757952)

[Тема роботи 3](#_Toc476757953)

[Актуальність теми 3](#_Toc476757954)

[РОЗДІЛ 1. ПОНЯТТЯ ДИНАМІЧНОГО ПРОГРАМУВАННЯ, ОСНОВНІ ЙОГО ПРИНЦИПИ 4](#_Toc476757955)

[1.1 Визначення та сутність 4](#_Toc476757956)

[1.2 Основні етапи формування, історія 5](#_Toc476757957)

[1.3 Загальна методика динамічного програмування 6](#_Toc476757958)

[1.4 Задачі на рекурентні співвідношення 8](#_Toc476757959)

[РОЗДІЛ 2. ОСНОВНИЙ ПРИНЦИП ТА МЕТОД РЕАЛІЗАЦІЇ 9](#_Toc476757960)

[2.1 Вступні зауваження 9](#_Toc476757961)

[2.2 Множина задач, що вирішуються, функція обчислення 10](#_Toc476757962)

[2.3 Граф залежності задач 11](#_Toc476757963)

[РОЗДІЛ 3. КЛАСИФІКАЦІЯ ЗАДАЧ ДИНАМІЧНОГО ПРОГРАМУВАННЯ ЗА ОБРАНИМИ МЕТОДАМИ ЇХ ВИРІШЕННЯ 14](#_Toc476757964)

[3.1 Задачі пошуку оптимальної кількості і суми 14](#_Toc476757965)

[3.1.1 Шляхом від оптимізації до результату, сума 14](#_Toc476757966)

[3.1.2 Максимальна кількість елементів сприятливих рішень 14](#_Toc476757967)

[3.1.3 Оптимальна кількість елементів у визначеному порядку 15](#_Toc476757968)

[3.1.4 Шляхом від результату до оптимізації 15](#_Toc476757969)

[3.2 Задачі пошуку оптимального шляху 16](#_Toc476757970)

[3.3 Задачі на рекурентну формулу 16](#_Toc476757971)

[РОЗДІЛ 4. РІШЕННЯ ДЕЯКИХ ПРИВЕДЕНИХ ЗАДАЧ 18](#_Toc476757972)

[РОЗДІЛ 5. РОЗБІР ПРОГРАМНОГО ІНТЕРФЕЙСУ ПРОГРАМИ 27](#_Toc476757973)

[ВИСНОВКИ 31](#_Toc476757974)

[ВИКОРИСТАНІ ДЖЕРЕЛА 32](#_Toc476757975)

ВСТУП

Тема роботи

Динамічне програмування – дуже перспективний та широко застосовуваний розділ математики, зустрічається майже в усіх галузях науки. З ним доводиться мати справу в технологіях, в програмування, в різних питаннях фізики, біології, в економіці (часто у задачах на оптимальне розподілення ресурсів), тощо. В ньому запропоновано розділити велику задачу на маленькі підзадачі, та вирішивши усіх їх, прийти до відповіді на все завдання.

## **Актуальність теми**

Актуальність теми зумовлена тим, що зараз оптимізаційні методи використовуються дуже часто, спроба вирішити такі завдання рекурсивно або перебором призводить до підвищення часу виконання роботи програми, а її швидкість має вагоме значення. Не менше використання методів динамічного програмування ми зустрічаємо на олімпіадах з програмування, дуже часто для вирішення задачі на оптимізацію на олімпіадах дуже незручно вирішувати повним перебором, більше того, це займає набагато більше часу, за рахунок великої кількості операцій. Тому дуже корисно буде ознайомитися з типовими задачами цього способу та, що не менш важливо, методами їх вирішення, бо коли людина зрозуміє декілька загальних методів розв’язання задач динамічного програмування, вона буде орієнтуватися у цьому розділі набагато краще.

**Головна мета роботи:** Головною метою роботи є дослідження декількох методів рішення типових задач динамічного програмування та класифікація цих задач за методами їх вирішення.

**Постановка задач.** В моїй роботі розглянуто декілька задач динамічного програмування та алгоритми їх розв’язку. Треба виділити головні методи їх вирішення, тобто знайти спільний прийом або спосіб реалізації задач, та згрупувати їх по цим методам. Також продемонструвати роботу алгоритмів вирішення цих задач.

# **РОЗДІЛ 1. ПОНЯТТЯ ДИНАМІЧНОГО ПРОГРАМУВАННЯ, ОСНОВНІ ЙОГО ПРИНЦИПИ**

## **1.1 Визначення та сутність**

На практиці часто доводиться зустрічатись з випадками, коли метою (ціллю) оптимізації є встановлення найкращої послідовності тих чи інших робіт (виробничих операцій, етапів будівництва різних споруд тощо). З подібною метою зустрічаються при розв’язанні задач так званого динамічного програмування

Динамічне програмування – розділ математики, який присвячено теорії і методам розв’язання багатокрокових задач оптимального управління. Динамічне програмування є одним з розділів оптимального програмування. Часто у «динамічному програмуванні» мають на увазі багатоступеневість – розподілення управління на ряд послідовних етапів, ступенів, кроків, що відповідають, як правило, різним моментам часу. Таким чином, говорячи про визначення «динамічного програмування», слід зазначити, що під словом «програмування» в даному випадку розуміють прийняття рішень, планування, а слово «динамічне» вказує на значення часу та порядку виконання операцій в методах цього розділу.

Розглянемо приклад такого процесу. Нехай планується діяльність групи підприємств на N років. Тут кроком є один рік. На початку 1-го року на розвиток підприємств виділяються кошти, які повинні бути якось розподілені між цими підприємствами. В процесі їх функціонування виділенні кошти частково витрачаються. Кожне підприємство за рік приносить певний дохід, який залежить від вкладених коштів. На початку року наявні кошти можуть перерозподілятися між підприємствами: кожному з них виділяється якась частка коштів. Ставиться питання: як на початку кожного року розподілити наявні кошти між підприємствами, щоб сумарний дохід від всіх підприємств за N років був максимальним? Це типова задача динамічного програмування, яка може зустрітися в реальному житті, в якій розглядається керований процес – функціонування групи підприємств. Планування кожного року має проводитися з урахуванням загальної вигоди, одержуваної по завершенні всього процесу, що і дозволяє оптимізувати кінцевий результат за обраним критерієм. Таким чином, динамічне програмування в широкому сенсі є оптимальним управлінням процесами за допомогою зміни керованих параметрів на кожному році.

Загалом, динамічне програмування є водночас і методом математичної оптимізації і методом комп’ютерного програмування. В обох контекстах воно використовує підхід спрощення пошуку розв’язку складної задачі, розбиттям її на простіші задачі, часто методом рекурсії.

## **1.2 Основні етапи формування, історія**

Річард Ернест Беллман

В основі методу динамічного програмування лежить принцип оптимальності, вперше сформульований Річардом Ернестом Беллманом, американським математиком у 1953 році – яким би не був стан системи пісня певної кількості кроків, на найближчому кроці потрібно вибирати управління так, щоб воно, в сукупності з оптимальним керуванням, на всіх наступних кроках приводило до оптимального результату. Тобто при кожному крокові вибирається таке управління, що повинне привести до оптимального виграшу. Слово динамічне було обране Беллманом, тому що звучало більш переконливо і краще підходило для передачі того факту, що проблема [оптимального управління](https://uk.wikipedia.org/w/index.php?title=%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B5_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D1%96%D0%BD%D0%BD%D1%8F&action=edit&redlink=1), яку він розв'язував цим методом, має аспект залежності від часу. Слово «програмування» в словосполученні «динамічне програмування» в дійсності до «традиційного» [програмування](https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) (написання тексту програм) майже ніякого відношення не має. Це використання таке саме як і в словосполученнях [лінійне програмування](https://uk.wikipedia.org/wiki/%D0%9B%D1%96%D0%BD%D1%96%D0%B9%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F) та [математичне програмування](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F), які фактично є синонімами для [математичної оптимізації](https://uk.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D1%87%D0%BD%D0%B0_%D0%BE%D0%BF%D1%82%D0%B8%D0%BC%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F). Тут воно означає оптимальну послідовність дій, оптимальну програму для отримання розв'язку задачі. Наприклад, певний розклад подій на виставці чи в театрі теж називають програмою.

Відзначивши внесок Беллмана, його ім’ям назвали рівняння Беллмана – основну формулу динамічного програмування, яка інтерпретує задачу оптимізації в рекурсивній формі. Рівняння Беллмана є достатньою умовою для оптимальності, асоційованою з математичним методом оптимізації, базується звісно на принципі оптимальності Беллмана. Являє собою диференційне рівняння у приватних похідних з початковими умовами, заданими для останнього моменту часу для функції Беллмана, яка виражає мінімальне/максимальне значення критерію оптимізації, яке може бути досягнуто за умови «еволюції» системи з поточного стану в деякий кінцевий. Його принцип дозволяє вирішувати або осмислювати багато цікавих задач, що виникають у реальному житті, особливо застосовувані в економіці.

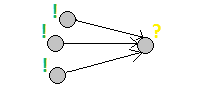
## **1.3 Загальна методика динамічного програмування**

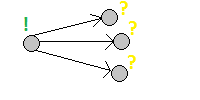
З точки зору математичної оптимізації, динамічне програмування полягає в спрощенні знаходження загального оптимального розв’язку, шляхом пошуку розв’язків в підзадачах, отриманих розбиттям задачі на послідовні проміжки часу.

Ідея і метод динамічного програмування найбільше пристосова­ні до дискретних задач, якими в більшості є задачі управління.

Доволі часто на олімпіадах зустрічаються задачі, що провокують до застосовування алгоритмів перебору. Але простий підрахунок числа варіантів перебору переконує у неефективності такого метода. Тому треба використовувати принцип динамічного програмування. Суть у тому, що для вирішення поставленої задачі, вирішуються схожа, але більш мала задача. При цьому здійснюється перехід до більш простих і так далі, поки не доходить до тривіальної задачі. Як видно, можна скористатися властивостями рекурсії, але це може привести до швидкого переповнення стеку. Якщо використовувати рекурсію, то треба подбати про оптимізацію рекурсивних переходів, наприклад, не обчислювати одне й теж значення декілька разів, зробити так зване відсікання. Щоб не виконувати додаткову роботу, можна відмовитися від рекурсії, і вирішити задачу навпаки – спочатку вирішити тривіальну задачу, а потім переходити к більш складним. Другий спосіб вирішення задач динамічного програмування, тобто без рекурсії, використовується набагато частіше, і при рішенні, і при роз’ясненні матеріалу в посібниках.

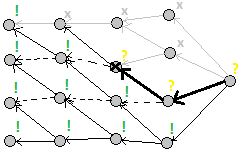
Щоб успішно вирішити задачу динамічного програмування потрібно спочатку визначити початкові значення/стани; потім обрати формулу перерахунку, є три порядки перерахунку: прямий, зворотній та лінивий.

1) Прямий порядок: стан послідовно перераховується, виходячи з вже порахованих.



2) Зворотній порядок: оновлюються всі стани що залежать від поточного стану

3) «Лінивий» прохід: рекурсивна функція перерахунку динаміки, тобто за допомогою першого мною описаного способу рішення задач динамічного програмування. Нагадує обхід у глибину по ациклічному графу, де ребра – це залежності між станами, а вершини – це стани.



## **1.4 Задачі на рекурентні співвідношення**

У всіх посібниках, статтях, які я зустрічав, пишуть, що динамічне програмування це рекурентні співвідношення. Це помилково, бо саме динамічне програмування полягає у оптимізації, тобто в рішенні обов’язково повинна бути функція оптимізації (max/min). Багато задач на рекурентні співвідношення приписують динамічному програмуванню. У своїй курсовій роботі я все таки розглядав задачі цього типу, для демонстрації роботи тільки рекурентних співвідношень в задачі, тобто без оптимізації.

Наприклад, класична задача про Числа Фібоначчі, всім відома формула:

F[i] = F[i-1] + F[i-2]. Як бачимо, в ній немає функції min або max.

Тобто це не є динамічним програмуванням, цю задачу просто як приклад дають спочатку, щоб зрозуміти сутність динамічного програмування, тобто щоб людина зрозуміла, що поточне значення виражається через попередні.

Рішення задачі динамічного програмування повинно мати в собі не тільки рекурентне співвідношення, а ще й функцію оптимізації. Приведемо приклад саме динамічного програмування, наприклад задача «Платні сходи», яку я розбирав у програмі. Формула перерахунку масиву сум (масиву динаміки):

D[i] = min( D[i-1], D[i-2] ) + a[i];

Отже, в задачах динамічного програмування на кожному кроці алгоритм повинен «вирішувати» що робити, для оптимізації. В «Платних Сходах» він вирішує, з якої сходинки краще наступити на поточну: з сходинки (і-1) або ж з (і-2). Також цей факт використовується при відновленні відповіді, тобто щоб вказати, як потрібно діяти людині, щоб заплатити якомога найменше грошей: ми запам’ятовуємо з якої сходинки ми потрапили у поточну (за допомогою допоміжного масиву «предків»), а потім з кінцевої йдемо по цьому масиву поки не дійдемо до низу сходинок. Як бачимо, ще одна особливість динамічного програмування в тому, що алгоритм повинен вирішувати якесь питання, а не просто підсумовувати за допомогою рекурентної формули. Це і є принципом оптимальності.

# **РОЗДІЛ 2. ОСНОВНИЙ ПРИНЦИП ТА МЕТОД РЕАЛІЗАЦІЇ**

## **2.1 Вступні зауваження**

Отже, у чому полягає суть метода динамічного програмування? Нехай дана задача. Чи можемо ми однозначно сказати о використовуванні метода динамічного програмування? Питання можна продовжувати.

По-перше, на наш погляд, точних конкретних і достатніх умов можливості рішення конкретної задачі за допомогою динамічного програмування не сформульовано. Із цього слідує, що у кожному конкретному випадку нам треба шукати можливість рішення даної задачі названим методом.

По-друге, у самої загальної постановці метод не містить якоїсь новизни. У твердженні о розбитті задачі на підзадачі, потім підзадач – на наступний рівень підзадач і т.д., а також формуванні рішення початкової задачі з рішень підзадач немає чогось принципового нового – це відомий принцип структуризації, та він в якомусь виді застосовується та використовується з давніх давен. Запам’ятовування результатів рішення підзадач у деякій структурі даних та рішення кожної задачі один раз (ключова відмінність від перебору) – це, звісно, чудово, але чи це технічний прийом, і в чи в цьому суть? На мій погляд, «корінь» відмінності метода в іншому. Всі підзадачі, будь-якого рівня розбивки, та, звісно, сама початкова задача мають одну «топологію» (наприклад, якщо людина не знає, як пояснити щось, вона вводить іншу незрозумілість і через неї пояснює першу незрозумілість). Великий шар, менше шар та ще менше – вже зовсім шарик, але «топологія» в них єдина.

Таким чином, якщо задача допускає розбивку на підзадачі с тією ж самою «топологією», то вона вирішується методом динамічного програмування. Наприклад, задача на знаходження квадрата з максимальною площею – задача на метод динамічного програмування (ми можемо розбити її на підзадачі одної топології), а якщо б задача стояла не в знаходженні квадрата максимальної площі, а в знаходження прямокутника максимальної площі, то це вже не динамічне програмування.

## **2.2 Множина задач, що вирішуються, функція обчислення**

Нехай V = {v1, v2, …, vn} – множина задач, від рішення яких опосередковано чи на пряму може залежати рішення початкової задачі. Початкова задача теж може входити в цю множину. Можливі ситуації, коли для конкретних вхідних даних потрібно вирішити тільки невелику кількість задач. Множина же V містить всі задачі. Розберемо на прикладі.

Знаходження біноміальних коефіцієнтів(числа сполучень). З формули  слідує, що знаходження залежить від знаходження для . Остання умова відсікає непотрібні підзадачі, наприклад обчислення і так далі. Так, множина задач

При рішенні задачі методом динамічного програмування потрібно знайти спосіб формування нових рішень на основі вже найдених. Це повинно бути деяке правило, що визначає, яким образом рішення задачі виражається через рішення її підзадач. В силу того, що задачі з множини V мають однакову структуру, у нас є підстава вважати, що це правило буде загальне.

Введемо нове поняття. Назвемо обчислюваної функцією f відображення множини задач у множині відповідних рішень. Тобто f(v) – це рішення задачі . Таким чином, ми зв’язали знаходження рішень с обчисленням значень деякої функції.

Навіщо це потрібно? Бо спосіб формування нових рішень звичайно записується як рекурентне співвідношення. Приведемо приклад (тут за нього можна взяти і числа Фібоначчі).

1. Числа Фібоначчі. Якщо , то очевидно, що рекурентне співвідношення для обчислення – це

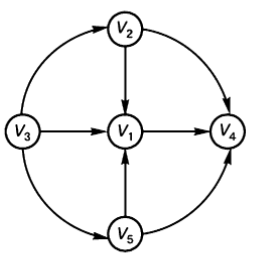
2) Задача Черепашка. Множина задач

має на увазі, що f() – це шлях до клітини (і,j). З розбору цієї задачі видно, що достатньо обчислювати і зберігати тільки суму елементів, а шлях відновлюється після отримання оптимального значення вартості шляху. Отже, f() – це сума елементів на оптимальному шляху до клітини (і, j), і рекурентне співвідношення має вид:

Звичайно в таких задачах убирають v у визначенні функції обчислення. Тобто заміть пишуть f[i][j]. Так, в задачі Черепашка f[i][j] – це максимальна сума елементів на шляху від клітини (1,1) до клітини (i, j). Надалі ми не будемо при рішенні задач описувати множину V, вона очевидним образом слідує з визначення обчислюваної функції.

## **2.3 Граф залежності задач**

У неявному виді взаємозв’язок завдань, прописана в рекурентних співвідношеннях, говорить про порядок їх рішення. Введемо поняття граф залежностей задач. У цього графа вершини це задачі, а ребра – залежності. Наявність ребра (u, v) означає, що рішення задачі v залежить від рішення задачі u.

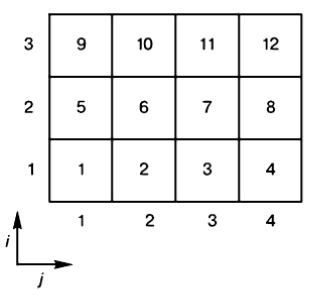
Граф цей потрібен тому що дозволяє визначити, яким образом організувати порядок обчислювань.

Наприклад, нехай граф залежностей представлений так, як на малюнку. Єдина задача, яке не залежить від інших, це v3, саме з неї необхідно починати покроковий процес. Знаючи v3, можемо вирішити v5, v2. Продовжуючи аналогічно, отримаємо наступний порядок обчислення:

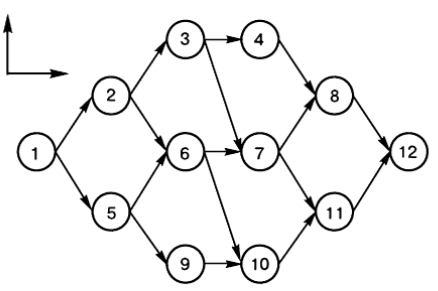
v3→v5→v2→v1→v4, або v3→v2→v5→v1→v4.

Виникає питання, а як обчислити задач, в які не входять ні одне ребро? Дійсно, оскільки в цьому графі немає циклів, існує одна чи більше вершин, в які не входять ребра. В цьому випадку відповідні задачі називаються базовими. Саме на їх основі будуть вирішуватися всі задачі що залишилися, і буде обчислюватися відповідь. Такі базові задачі називаються початковими умовами.

Проілюструємо використання графа залежностей на прикладі задачі Мишка та зернятка, проілюстрованої у програмі. Нехай дано поле (на рисунку), мишка може рухатися вверх або вправо, треба пройти з (1,1) до (3,4). На рисунку цифрами позначено НЕ кількість зернят, а номера задач, які потребують рішення: v =



Таким чином, треба вирішити 12 задач. Граф залежностей задач показаний на рисунку нижче (зліва верху, на рисунку позначено напрям рух Мишки – вверх і вправо).



Бачимо, що задачі можна вирішувати в одній з трьох послідовностей:

1. (1,2,3,4), (5,6,7,8), (9,10,11,12);

2. (1,5,9), (2,6,10), (3,7,11), (4,8,12);

3. (1), (2,5), (3,6,9), (4,7,10), (8,11), (12);

При зміні умови, наприклад, додавати можливість мишці ходити по діагоналі, граф буде змінюватися, але всі вони вирішуються методом динамічного програмування. Відмінність у тому, що залежності між задачами різні, що визначає різну логіку формування елементів матриці, в якій зберігаються результати рішення підзадач.

# **РОЗДІЛ 3. КЛАСИФІКАЦІЯ ЗАДАЧ ДИНАМІЧНОГО ПРОГРАМУВАННЯ ЗА ОБРАНИМИ МЕТОДАМИ ЇХ ВИРІШЕННЯ**

## **3.1 Задачі пошуку оптимальної кількості і суми**

Задач динамічного програмування, що вирішуються даним методом дуже багато. Тому я вирішив внести цей метод як окремий.

### **3.1.1 Шляхом від оптимізації до результату, сума**

До цього методу я відніс такі задачі: «Трикутник», «Платні сходи», «Купівля квитків».

Від оптимізації до результати означає те, що ми йдемо поступово, вирішуємо найменші задачі, і доходимо да загальної. При цьому, коли ми йдемо ми додаємо значення, тобто потрібно знайти оптимальну суму.

Наприклад, візьмемо «Купівлю квитків», ми йдемо, заповнюючи масив сум, і значення в ньому завжди буде дорівнювати сумі якихось значень початкового масиву. При цьому не забуваємо про оптимальність, в рішенні цієї задачі на кожній ітерації маємо приймати рішення: скільки білетів поточна людина повинна взяти задля оптимальності часу.

Теж саме з «Платними сходами», йдемо, додаючи, вирішуючи з якої сходи потрібно піднятись на поточну, та з «Трикутником», на кожній ітерації, на кожному крокові ми оптимізуємо, обираємо кращий шлях від вершини до поточної точки, оптимізуємо спочатку а потім отримаємо результат, ось що значить «від оптимізації до результату»

### **3.1.2 Максимальна кількість елементів сприятливих рішень**

Як демонстрацію цього методу, я обрав задачу «Максимальний падпаліндром».

Згідно з умовою задачі, треба знайти підпаліндром найбільшої довжини, тому в назві метода є «максимальна кількість елементів». «Сприятливих рішень» тому, що на кожному крокові ми обираємо найбільшу з попередніх рішень, та при цьому підпаліндром, як і питається в умові.

### **3.1.3 Оптимальна кількість елементів у визначеному порядку**

До цього методу рішення я додав: «Максимальний загальний підрядок», «Найбільша зростаюча підпослідовність».

«Оптимальна кількість», тому що в умовах задач питається про відповідь найбільшої довжини, тобто оптимальної довжини, оптимальної кількості елементів.

Як в задачі про загальний підрядок, так і в задачі підпослідовності, ми маємо строго дотримуватися порядку рішення. Тобто в задачі про найбільшу зростаючу підпослідовність d[i] – це довжина найбільшої зростаючої підпослідовності, послідовності від 0-ого елементу до i-го, у визначеному порядку. Тобто в даному випадку у зростаючому порядку, може бути і в спадному, це не принципово, але порядок повинен бути, бо інакше задача не буде мати сенс. А у максимальному загальному підрядку взагалі очевидно, що порядок визначений, тому що дані рядки, рядки читаються зліва направо, і логічно, що загальний підрядок теж буде читатися зліва направо, і в першому рядку і в другому.

### **3.1.4 Шляхом від результату до оптимізації**

Метод «оборотній» 3.1.1, тут ми повинні знати результат, і на кожному крокові його використовувати, тобто він на кожній ітерації відіграє свою роль, зазвичай роль «рамки», я додав до цього методу дві задачі: «Задача про рюкзак» та «Калькулятор».

Візьмемо спочатку задачу про рюкзак. Згадаємо її рекурентну формулу:

A[k][s] = max (A[k-1][s], A[k-1][s-w[k]] + p[k])

Тут k від 1 до N, а s – від 1 до W (місткість рюкзака). Тут s – це саме результат. Звісно, на кожному кроці він збільшується, АЛЕ він ставить рамку, тобто максимальне значення маси, яку ми можемо взяти з поточних k предметів. Тому ми, знаючи результат, доходимо до кінцевого значення (k = N, s = W), і вже маємо оптимізацію, тобто знаємо, що треба класти в рюкзак а що ні. Отже, на кожній ітерації ми знали результат, і від нього віднімали масу поточного предмету, бо логічно, що якщо ми кладемо до рюкзака, до місткість його зменшується.

Аналогічно з калькулятором, маємо рекурентну формулу:

A[i] = min(A[i-1], A[i/2], A[i/3]) + 1;

Тобто на кожній ітерації ми знаємо результат (це число i), і обираємо звідки ми до нього прийшли, з трьох випадків, для яких ми теж знаємо відповідь.

На мій погляд, цей метод дуже цікавий, тому ще неординарний.

## **3.2 Задачі пошуку оптимального шляху**

Особливість цього методу в тому, що створюється двовимірний масив, розмірами тими ж, що і вихідний, з цінами. І у масиві сум обчислюється поступово рішення для всіх підзадач: оптимальний шлях від (1,1) до (i, j), де i від 1 до N, а j від 1 до M.

Я взяв дві типові задачі: «Маршрут» та «Мишка і зернятка». В першій з них ми повинні пройти від (1,1) до (N,M) оптимальним шляхом, а саме мінімальний. В другій ми повинні пройти від (N, 1) до (M, 1) максимальним шляхом, при рішенні другої я «перевернув» масив, для зручності, але рішення фактично теж саме, тільки функції оптимальності різні.

Ще часто задачі на такий метод називають задачами динамічного програмування на дошці. Бо використовуються двовимірні масиви.

## **3.3 Задачі на рекурентну формулу**

Як я вже писав, розглядаю в моїй роботі задачі на рекурентну формулу тільки тому, що часто помилково зустрічаються як приклад динамічного програмування. Це просто приклад використання рекурентної формули БЕЗ використання оптимізації, притаманній динамічному програмуванні.

Обрав наступні задачі: «Стрибунець», «Хід конем», «Трикутник Паскаля».

У задачі «Стрибунець», як і у «Хід конем» йде підрахунок способів. Як вже писалося, особливість цього методу в тому, що немає функцій оптимальності. І йде прямий підрахунок, шляхом додавання, або, в деяких задачах і множення значень.

У «Трикутнику Паскаля» йде прямий підрахунок для кожної комірки суми попередніх над поточною. Я взяв цю задачу, тому що вона дуже типова, та вважається класикою використання рекурентних співвідношень. Також це дуже швидкий спосіб обчислення чисел сполучення. Я розглядав цю задачу у пункті 2.2

Взагалі, задач на рекурентну формулу дуже і дуже багато, часто на олімпіадах з програмування зустрічаються задачі, для рішення яких потрібно рекурентну формулу виводити самостійно.

# **РОЗДІЛ 4. РІШЕННЯ ДЕЯКИХ ПРИВЕДЕНИХ ЗАДАЧ**

Приводити рішення всіх задач, які я розглядав у програмі я не буду, але декілька цікавих розберу.

Задача купівля квитків

Цю задачу вирішуємо методом динамічного програмування. Тобто пробуємо поступово збільшувати розмір черги. Нехай у нас в черзі стоїть тільки один чоловік. Так як кожній людині потрібен один і тільки один білет (навіть якщо 2 або 3 білета купити швидше), то відповіддю буде просто число a[1].

Тепер додаємо до черги наступну людину, тепер у черзі 2 людини. Вони можуть купити білети роздільно, тоді час покупки буде a[1] + a[2]. Крім того, вони можуть об'єднатися і купити білети разом, при цьому купити два білета по умові задачі може тільки перша людина. З двох можливостей ми обираємо таку, яка займе менше часу та запам'ятовуємо час в елементі d[2]. Таким чином, d[2] - мінімальний час покупки білетів з чергою з 2 людей.

Додаємо третю людину. Розглянемо варіанти тепер. Якщо третя людина купляє білет самостійно, то перші два, очевидно, не залежать від нього. Тоді ми уже вирішили задачу, бо ми тільки що знайшли мінімальний час покупки білетів для черги з двох людей, воно зберігається у d[2]. Загальний час покупки в цьому випадку буде d[2] + a[3]. Припустимо, друга і третя людина вирішують купити білети разом. Відповіддю на цей випадок буде b[2] + a[1] (два білети купляє друга людина та перша купляє білет самостійно). Нарешті, домовитися зможуть і всі три людини, це буде вже третій випадок, тоді вони куплять білети за c[1] - саме стільки часу займе купівля трьох білетів у першого стоячого в черзі чоловіка. Напишемо кращий варіант в d[3] - це буде мінімальний час покупки білетів з чергою із 3 перших людей.

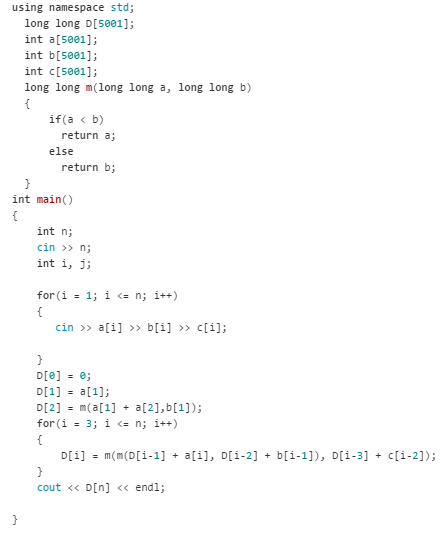
Далі будемо діяти абсолютно аналогічно. Додаємо до черги четверту людину, напишемо формулу для d[4], d[4] = min(d[3] + a[4], d[2] + b[3], d[1] + c[2]). Продовжуючи міркувати аналогічно, у загальному вигляді отримуємо наступну рекурентну формулу:

d[i] = min(d[i-1] + a[i], d[i-2] + b[i-1], d[i-3] + c[i-2])

Відповідь до задачі буде зберігатися у d[n].

Ця задача явний приклад роботи динамічного програмування. Тобто вирішуємо повну задачу, використовуючи рішення більш менших підзадач.

В даній задачі я розмір масивів вказав 5001, але це не принципово, тобто в коді можна встановити і N. Код:



Задача максимальний підпаліндром

Нашу задачі можна перефразувати так: дано рядок, необхідно знайти довжину найбільшого паліндрома, який можна отримати викреслюванням деяких букв з даного рядка. Дана задача є дуже відомої на олімпіадах та є прикладом динамічного програмування на рядках.

Позначимо даний рядок через S, тобто його символи - S[i], 1 <= i <= n (довжина рядка). Будемо розглядати можливі підрядки даного рядка с i-го по j-ий символ, позначимо їх через S(i,j) (наприклад, для рядка "QWERTY", S(2,4) = ERT, рахуємо від 0). Максимальні паліндроми для підрядків будемо записувати у квадратний масив D: D[i,j] - найбільший максимальний підпаліндром, який можна отримати з підрядка S(i,j).

Тепер заповнимо головну діагональ масиву D буквами нашого рядка. Будемо заповнювати його по діагоналям від головної до правого верхнього кута. Тобто D[i,j] - максимальний підпаліндром підрядка S(i,j). Почнемо з вирішення простих задач, тобто коли будемо знаходити максимальний підпаліндром підрядка S(i,i+1), в цьому випадку, якщо ці два символи однакові, то ці два символи і є максимальний підпалиндром S(i,i+1), якщо ж вони різні, то обираємо будь який з них, і так по всій діагоналі. При наступній ітерації, ми вже будемо знаходити максимальний підпалиндром рядка S(i,i+2), його ми знаходимо за рахунок того, що знайшли максимальний підпаліндром з підрядків S(i,i+1) та S(i+1,i+2). Далі робимо по аналогії, і отримуємо наступну рекурентну формулу:

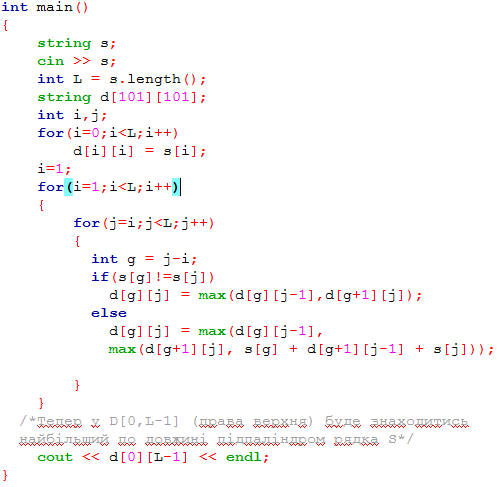
D[g,j] = max(D[g,j-1],D[g+1,j]), якщо S[g] != S[j]

D[g,j] = max(D[g,j-1],D[g+1,j],s[g] + D[g+1,j-1] + s[j]), якщо S[g] = S[j],

де 0 < i < S.Length; i <= j < S.Length;

Після заповнення D, у комірці [0][L-1] будемо мати відповідь до задачі.

Розміри масивів ввів довільно. Код програми:



Задача про рюкзак

Задача пакування рюкзака - задача комбінаторної оптимізації. Цю задачу, як і більшість задач динамічного програмування можливо вирішити жадібним алгоритмом або перебором, але більш точним та надійним методом рішення цієї задачі є динамічне програмування.

Нехай w[i] - маса i-го предмета, p[i] - вартість i-го предмета

Нехай A[k,s] це максимальна вартість предметів , які можна укласти в рюкзак місткості s, якщо можна використовувати тільки перші k предметів, тобто {n1,n2,...,nk}, назвемо цей набір допустимих предметів для A[k,s]

A[k,0] = 0; Це очевидно, можна брати будь-які з перших k предметів але місткість рюкзака дорівнює 0.

A[0,s] = 0; Жодний предмет брати не можна (k=0), тому максимальна вартість дорівнює 0.

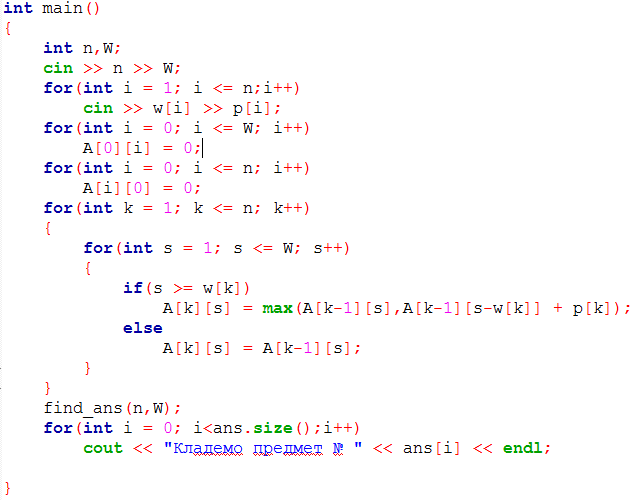
Тепер складемо рекурентне відношення в загальному випадку. Якщо предмет k не потрапив в максимальний рюкзак маси s, то максимальний рюкзак буде складений тільки з предметів с номерами 1, ..., k-1, отже, A(k,s) = A(k-1,s)

Якщо ж в максимальний рюкзак включимо предмет s, то маса тих, що залишилися предметів не перевищує s-w[k], а від додавання предмета s загальна вартість рюкзака збільшується на p[k], тобто A[k,s] = A[k-1,s-w[k]] + p[k];

Тепер з двох "рішень" нам треба обрати оптимальніший (в цьому і полягає оптимальність даної задачі). Отже, маємо наступну рекурентну формулу:

A[k,s] = max(A[k-1,s], A[k-1,s-w[k]] + p[k]);

Шукана вартість набора дорівнює A[N,W], так як треба знайти максимальну вартість рюкзака, де всі предмети припустимі та місткість рюкзака W. Код:



Відновлення відповіді треба для демонстрації рішення, тобто відповідь на запитання: "Які предмети треба взяти, щоб отримати максимум вартості?".

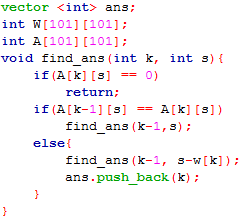
Будемо визначати, чи входить i-ий предмет в шуканий набір. Починаємо з елемента A[N,W. Для цього порівнюємо A[k,s] зі наступними значеннями:

1) Максимальна вартість рюкзака с тій самою місткістю та набором допустимих предметів {n1,n2,...,n(k-1)}, тобто A[k-1,s]

2) Максимальна вартість рюкзака с місткістю w[i] менше і набором допустимих елементів {n1,n2,...n(k-1)} плюс вартість p[i], тобто A[k-1,s-w[k]] + p[k]

Будемо відновлювати відповідь рекурсивним варіантом, за допомогою вектора ans;

Код:



Тепер у ans буде зберігатися індекси тих елементів, які треба взяти для оптимізації цін згідно з умовою.

Задача маршрут

Ця задача є класикою динамічного програмування. Жодне друкований або електронний посібник не обходиться без розбору цієї задачі.

Розглянути всі можливі варіанти і порахувати їх неможливо. Але можливо звести задачу до аналогічної.

Нехай нам відомий «мінімальний» шлях для всіх клітинок дошки окрім останньої, правої нижньої. Всі потрібні маршрути проходять через одну з клітинок, суміжними з поточною, їх всього дві (верхня та ліва). Мінімальний же маршрут проходить через ту, у якої значення динаміки менше.

Нехай a[i][j] – початковий масив, тобто «дошка», яка подається на вхід. d[i][j] – масив динаміки, що має в собі ціну оптимального шляху від [1][1] до [i][j]. У підсумку будемо мати значення d[N][N] – відповідь до задачі.

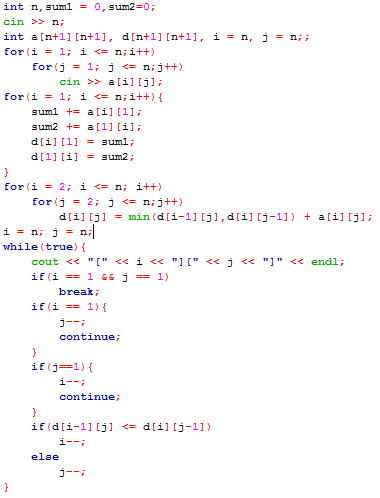
Очевидно, що у масиві динаміки будь-яка комірка лівого першого стовпця буде мати в собі суму усіх комірок до неї, починаючи з [1][1], бо в [i][1] комірку ми можемо попасти тільки з [i-1][1], тому що комірка [i][1-1] не існує, тобто зліва масив вже закінчується. Аналогічно і перший рядок масиву динаміки: ми можемо прийти тільки зліва, зверху вже масив закінчується. Тому ми заповнюємо у масиві динаміки перший стовпець та перший рядок в першу чергу, перед тим як будемо використовувати рекурентну формулу.

А рекурентну формулу не важко вивести, бо ми можемо прийти зверху та зліва, тобто обрати нам треба мінімум з цих варіантів і додати поточний, тобто додати a[i][j]. Маємо:

d[i][j] = min( d[i-1][j], d[i][j-1] ) + a[i][j];

Звісно і = 2…N, j = 2...N.

Задля відновлення відповіді будемо йти від d[N][N] до d[1][1], на кожному крокові будемо перевіряти що менше: d[i-1][j] чи d[i][j-1] і обираючи найменше, зменшувати на одиницю і або j. А якщо ми дійшли до першого рядка або до першого стовпця, то ми просто йдемо «напряму» до [1][1], тобто зменшуємо на одиниці відповідно j або і. Паралельно цьому, виводимо поточні індекси, тобто [i][j], що буде означати, що комірка [i][j] входить до оптимального маршруту. Реалізація:



Задача трикутник Паскаля

Трикутник Паскаля – це невичерпне джерело математичних «радощів». Наприклад обчислення чисел розміщення із n по k найефективніше робити за допомогою цього трикутника.

Нехай d[i][j] – масив динаміки, тобто це і є масив, в якому ми будемо зберігати трикутник. Також, якщо ми захочемо отримати такий трикутник:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 1 |  |  |  |
|  |  | 1 |  | 1 |  |  |
|  | 1 |  | 2 |  | 1 |  |
| 1 |  | 3 |  | 3 |  | 1 |

То у масив d[i][j] буде виглядати так:

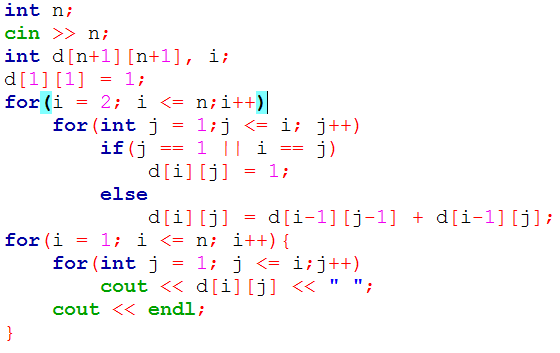
|  |  |  |  |
| --- | --- | --- | --- |
| 1 |  |  |  |
| 1 | 1 |  |  |
| 1 | 2 | 1 |  |
| 1 | 3 | 3 | 1 |

Тобто якщо в трикутнику в нас поточне значення дорівнює сумі верхнього правого і верхнього лівого, то у масиві це буде сума верхнього лівого і верхнього «прямо» над поточним. Тобто маємо таку формулу:

d[i][j] = d[i-1][j-1] + d[i-1][j];

Очевидно, що i = 2…N, j = 1…і; Та звісно d[1][1] = 1, за умовою.

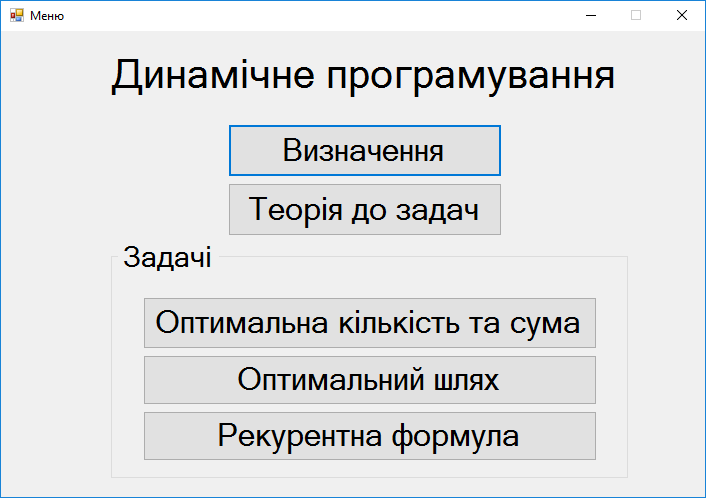
Код:



# **РОЗДІЛ 5. РОЗБІР ПРОГРАМНОГО ІНТЕРФЕЙСУ ПРОГРАМИ**

Можливість зміни розмірів вікон у програмі заблокована

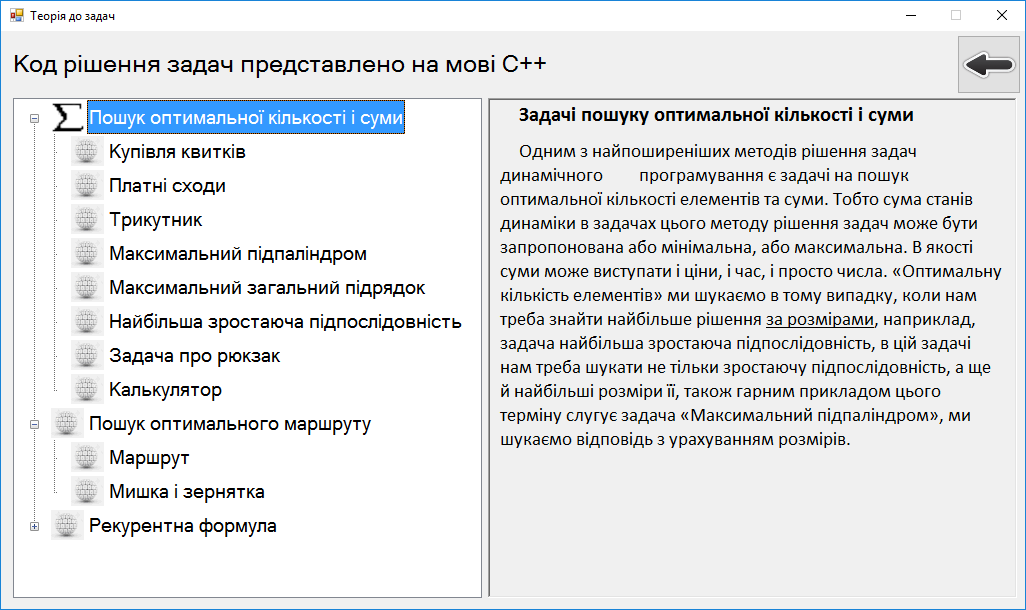
Спочатку відкриється меню програми:



У «Визначення» коротко описується поняття, принцип динамічного програмування. Приведено приклад порядків та код до них (на числах Фіб.)



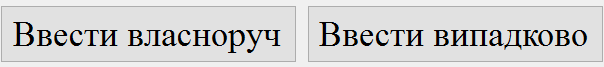
Розділ «Теорія до задач» представляє собою пояснення, теорію та реалізацію на мові C++ задач, які були представлені у програмі.



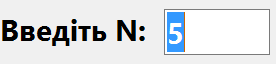
Розділ «Задачі» має три пункти: «Оптимальна кількість та сума», «Оптимальний шлях», «Рекурентна формула».

Всі надписи з назвами задач містять реалізацію та візуалізацію відповідних задач.

В деяких з них мається можливість ввести власноруч або випадково:



Також в кожній з них є можливість ввести вхідні дані:

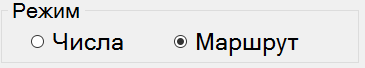


Якщо потребується більше даних, у тих задачах будуть ще поля для вводу.

У тих задачах, в яких потрібно, будуть пояснення до кольорів, використаних на формі, наприклад:



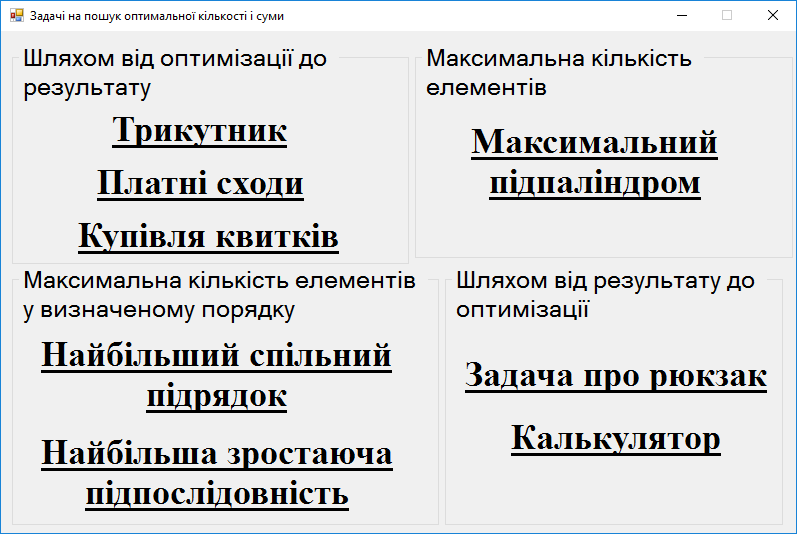
Також в деяких задачах може бути налаштований «Режим», якщо режим буде налаштований на «Числа», то у початковому масиві, на вході, будуть стояти ті вхідні дані, які були спочатку, якщо ж на «Маршрут», то буде показаний маршрут, або порядок дій на початковому масиві, тобто на шляху маршруту чисел не буде, будуть тільки вказівки для оптимального шляху:



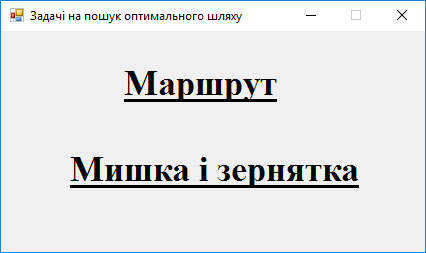
В деяких розділах буде можливість повернутися до попереднього по ієрархії вікна:



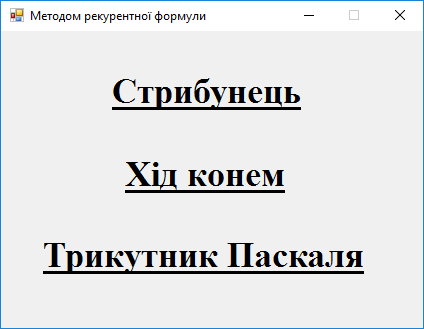
У «Оптимальна кількість та сума» відкриється наступне вікно, з класифікацією, описаної раніше:



У «Оптимальний шлях», має вікно з назвами двох задач не відповідний метод рішення:

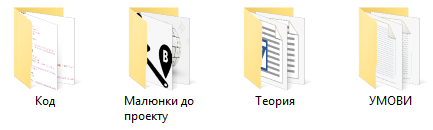


У розділі «Рекурентна формула» відкриється вікно з відповідними задачами:



ОБОВ’ЯЗКОВО

В папці з .exe файлом повинні знаходитися такі файли та папки:



# **ВИСНОВКИ**

Я розібрав не всі методи рішення, які є в динамічному програмуванні, їх більше, наприклад, метод рішення задач за допомогою профіля мають дуже широке застосування, їх можна віднести до окремого методу. Але самі базові задачі, методи динамічного програмування, які найчастіше зустрічаються на олімпіадах з програмування розібрані в моєї роботі.

Динамічне програмування дуже добре пристосоване для вирішення завдань оптимізації. Існує дуже багато задач на динамічне програмування, дуже різні за змістом і типом, але все ж таки можна більшість задач класифікувати за методом їх рішення, побачити спільні ризи у реалізації алгоритмів їх розв’язку.

Труднощі виконання цієї роботи полягали у пошуку алгоритму відновлення відповіді в деяких задачах, у знаходженні загального методу рішення групи задач, у вирішенні деяких задач, бо динамічне програмування не має загального алгоритму, і практично всі задачі мають свої особливості рішення;

# **ВИКОРИСТАНІ ДЖЕРЕЛА**

* Беллман Р. Динамическое программирование. Изд-во иностранной литературы, 1960.
* Динамическое программирование / Окулов С.М. / Пестов О.А. – Электронное издательство. 2012 год
* Лежнев, А.В. Динамическое программирование в экономических задачах: учеб. пособие / Лежнев А.В. - Москва: Бином, 2010.
* Сайт <http://informatics.mccme.ru>
* Сайт <https://www.e-olymp.com/ru/>
* Сайт <https://habrahabr.ru>
* Акулич И.Л. Глава 4. Задачи динамического программирования // Математическое программирование в примерах и задачах