**Тема: Динамічне програмування**

**1. Динамічне програмування з одним параметром: кількість способів розв’язань**

**Задача про коника**

На числовій прямій сидить коник, який може стрибати вправо на одну або на дві одиниці. Спочатку коник знаходиться в точці з координатою 0. Визначте кількість різних маршрутів коника, що приводять його в точку з координатою n.

**Розв’язання**

Позначимо кількість маршрутів коника, що ведуть в точку з координатою n, як . Тепер навчимося обчислювати функцію . Насамперед зазначимо, що (це вироджений випадок, існує рівно один маршрут з точки 0 в точку 0 ‒ він не містить жодного стрибка), *,* . Як обчислити ? У точку коник може потрапити двома способами – з точки за допомогою стрибка довжиною 2 і з точки стрибком довжини 1. Тобто число способів потрапити в точку n дорівнює сумі числа способів потрапити в точку і , що дозволяє виписати рекурентне співвідношення: , правильне для всіх .

**Рекурсивне розв’язання**

Тепер ми можемо оформити рішення цієї задачі у вигляді рекурсивної функції на мові Free Pascal:

function f(n: longint): longint;

begin

if n < 2

then f := 1

  else f := f(n - 1)+f(n - 2);

end;

Але при спробі обчислити розв’язання цієї функції для вже не дуже великих n, наприклад, для n = 40, виявиться, що ця функція працює вкрай повільно – 263 ms, на комп’ютері з тактовою частотою процесора в 2,2 ГГц, а при n = 50 виконання програми здійснюється за 32 350 ms. Отже, час роботи функції зі збільшенням зростає експоненціально, тобто таке рішення є неприйнятним за складністю. Причина цього полягає в тому, що при обчисленні рекурсивної функції підзадачі, для яких обчислюється розв’язання, «перекриваються». Тобто для того, щоб обчислити нам потрібно викликати і .

**Нерекурсивне розв’язання**

Насправді нескладно побачити, що значення рекурсивної функції в даному випадку будуть збігатися з числами Фібоначчі, бо обчислюються за тими ж рекурентним співвідношенням. А для обчислення чисел Фібоначчі можна використовувати цикл, а не рекурсію – наступне число Фібоначчі визначається, як сума двох попередніх.

Приклад на мові Pascal:

uses Dos;

var f: array [0..100] of qword;

i, n:longint;

hp,he,hvp,hve,sp,se,msp,mse:word;

s:qword;

begin

readln (n);

//Засікання часу початку роботи алгоритму

GetTime(hp,hvp,sp,msp);

//Виконання власне алгоритму

for i:= 1 to n do

f[i]:= 0;

f[0]:= 1;

f[1]:= 1;

for i:= 2 to n do

f[i]:= f[i - 1] + f[i - 2];

writeln(f[n]);

//Засікання закінчення роботи алгоритму і виведення

// результату у мілісекундах

GetTime(he,hve, se,mse);

s:=mse-msp+100\*(se-sp+60\*(hve-hvp+60\*(he-hp)));

writeln(s);

end.

Складність такого рішення буде . Складність обчислення зменшується за рахунок того, що для кожного проміжного i значення обчислюється один раз і зберігається в списку, щоб згодом використовувати це значення кілька разів для обчислення і .

Такий прийом називається динамічним програмуванням. **Динамічним програмуванням** **називається метод розв’язування задач, при якому задача розбивається на підзадачі, відповіді до яких записуються в структуру даних і використовуються при розв’язуванні задачі.** Для розв’язання в кожній з підзадач використовується один і той же метод.

Ідеї цього методу належать російському математику А. А. Маркову, їх розвивали американські вчені Д. Уальд, Р. Айзекс, Р. Беллман. Власне термін «динамічне програмування» належить Річарду Беллману. Значний внесок у створення загального формалізму послідовного аналізу варіантів даного методу належить київській школі математиків на чолі з В. С. Михалевичем (схема формалізації Михалевича, метод «київський віник»).

Одним із способів розв’язання задач, для яких не можна побудувати оптимальний розв’язок на кожному кроці є перевірка всіх можливих послідовностей розв’язків. Динамічне програмування значно зменшує кількість варіантів‚ що слід перевірити‚ за рахунок виключення послідовностей розв’язків‚ які не можуть бути оптимальними.

Оптимальної послідовності розв’язків досягають завдяки явному зверненню до **принципу оптимальності:**

**оптимальна послідовність розв’язків має таку властивість, що, незважаючи на початковий стан та розв’язок в початковий момент, серед розв’язків, що залишились, завжди міститься оптимальна послідовність розв’язків щодо стану‚ який утворився після прийняття першого розв’язку.**

Нехай − це стан задачі, який утворюється після вибору розв’язку , . Нехай буде оптимальною послідовністю розв’язків відповідно до стану задачі . Тоді, згідно з принципом оптимальності‚ оптимальну послідовність розв’язків відповідно до визначають як найкращу послідовність розв’язків .

Для більшості задач рекурсивна природа задання принципу опти-мальності приводить до рекурентного типу співвідношень. Алгоритми динамічного програмування розв’язують ці рекурентні рівняння, щоб знайти оптимальний розв’язок заданої задачі.

Формулювати рекурентні співвідношення динамічного програмування можна за допомогою використання двох підходів перегляду варіантів розв’язань: **прямого та оберненого**.

Нехай − змінні, що характеризують шукану послідовність розв’язків.

У прямому перегляді побудова розв’язку формулюється в термінах оптимальної послідовності розв’язків для . Для оберненого перегляду формулювання тверджень щодо розв’язку здійснюють у термінах оптимальної послідовності розв’язків для .

Якщо рекурентні співвідношення сформульовані з використанням прямого підходу, тоді рівняння розв’язують за технікою «перегляду назад» - починаючи з останнього розв’язку. Якщо співвідношення сформульовані з використанням оберненого підходу ‒ їх розв’язують за технікою «перегляду вперед».

Завдяки використанню принципу оптимальності послідовності роз-в’язків, що містять підпослідовності, які не є оптимальними, здебільшого не розглядаються. Тому алгоритми динамічного програмування, як правило, мають поліноміальну оцінку.

Інша важлива риса динамічного програмування полягає у тому, що оптимальні розв’язки підзадач можна зберігати в такому вигляді, який запобігає перерахуванню їх значень у разі подальших використань цих підзадач. Здебільшого для цього використовують зберігання у структурах (одновимірний чи двовоимірний масив, різного роду списки). Використання структурних значень робить природним перетворення рекурсивних рівнянь в ітеративні програми.

За оберненого підходу побудова загального розв’язку із розв’язків підзадач меншої розмірності, обчислення йде від менших підзадач до більших, і відповіді запам’ятовуються в структурі даних.

Перевага використання структур даних полягає в тому, що, як тільки будь-яку підзадачу розв’язано, її відповідь заносять у структуру і ніколи вже не обчислюють знову.

Для аналізу застосовності метода перевіряють умови:

**1. У задачі можна виділити однотипні підзадачі різних розмірів.**

**2. Серед виділених підзадач є тривіальні, які мають малий розмір і очевидне розв’язання.**

**3. Оптимальні розв’язання підзадачі більшого розміру можна побудувати із оптимальних розв’язань менших підзадач.**

**4. Одні і ті ж нетривіальні менші підзадачі використовуються при розв’язуванні різних великих підзадач.**

**5. Кількість різних підзадач розумна і для запам’ятовування результатів не потрібно багато пам’яті.**

Перевірка 3 пункта – головний етап в аналізі застосовності динамічного програмування до задачі.

Основні кроки динамічного програмування:

**1. Задача розбивається на певну кількість підзадач.**

**Підзадача –** це та ж сама задача, але з меншою кількістю аргументів, або ж тією кількістю аргуметрів, але з меншими їх значеннями.

**2. Записуються рекурентні співвідношення, що виражають розв’язання задачі через підзадачі.**

**3. Здійснюється пошук оптимального розв’язання кожної підзадачі і формування його у вигляді структури даних.**

**4. Розв’язування задачі будується на основі оптимальних розв’язків підзадач.**

Відшукання розв’язку здійснюється так: спочатку здійснюється вибір останнього в часі розв’язання, потім у зворотному порядку вибирається решта розв’язань аж до початкового.

Динамічне програмування використовує ті ж рекурентні співвідношення, що і рекурсивне розв’язання, але на відміну від рекурсії в динамічному програмуванні значення обчислюються в циклі і зберігаються в структурі даних, наприклад, одновимірному масиві. При цьому заповнення структури даних йде від менших значень до більших, тоді як в рекурсії ‒ навпаки, рекурсивна функція викликається для великих значень, а потім викликає сама себе для менших значень.

**Модифікація задачі про коників**

Нехай коник стрибає на одну, дві або три одиниці праворуч. Необхідно обчислити кількість способів потрапити в точку . У точку коник може потрапити трьома способами – з точки за допомогою стрибка довжиною 3, з точки за допомогою стрибка довжиною 2 і з точки стрибком довжини 1. Тоді у рекуррентному співвідношенні з’явиться ще один доданок: . А початкові значення для обчислення функції повинні складатися з трьох чисел: , , . При цьому розв’язання зміниться не сильно:

var f: array[0..100] of longint;

i, n: longint;

begin

readln(n);

fillchar(f,sizeof(f),0);

f[0] := 1;

f[1] := 1;

f[2] := 2;

for i := 3 to n do

f[i] := f[i - 1] + f[i - 2] + f[i - 3];

writeln(f[n]);

end.

**Інша модифікація задачі про коників**

Ще раз модифікуємо задачу. Нехай деякі точки є «забороненими» для коника і він не може стрибати в них. «Карта» заборонених точок задається за допомогою масиву . У точку з номером *i* коник не може стрибати якщо , коли ж , то дана точка є дозволеною для коника. Як і в попередній задачі, необхідно знайти кількість маршрутів в точку .

У даному випадку також доведеться модифікувати рекурентне співвідношення: якщо , то : якщо точка «заборонена», то кількість способів потрапити в цю точку дорівнює 0, оскільки немає жодного допустимого маршруту, який закінчується в цій точці. Якщо ж , то значення обчислюється за тими ж рекурентним співвідношенням, що і раніше. Отримуємо наступне розв’язання:

var f, map: array[0..100] of longint;

i, n: longint;

begin

readln(n);

for i := 1 to n do

read(map[i]);

f[0] := 1;

f[1] := map[1]\*f[0];

f[2] := map[2]\*(f[1] + f[0]);

for i := 3 to n do

f[i] := map[i]\*(f[i - 1] + f[i - 2] + f[i - 3]);

writeln(f[n]);

end.

Тут використовується дещо інший код для обчислення суми для того, щоб граничні значення і також можна було обчислити за допомогою цього коду в основному циклі, а не перед ним.

**ДИНАМІЧНЕ ПРОГРАМУВАННЯ З ОДНИМ ПАРАМЕТРОМ: НАЙКРАЩИЙ СПОСІБ**

**ЗАДАЧА ПРО КОНИКІВ З ВАРТІСТЮ**

Нехай коник стрибає на одну або дві точки вперед, а за стрибок в кожну точку необхідно заплатити певну вартість, різну для різних точок. Вартість стрибка в точку *i* задається значенням списку *Price*. Необхідно знайти мінімальну вартість маршруту коника з точки в точку .

Нехай ‒ мінімальна вартість шляху з в . Виведемо рекурентне співвідношення для . Щоб потрапити в точку ми повинні потрапити в точку або , мінімальні вартості цих маршрутів дорівнюватимуть і відповідно. До них доведеться додати значення за стрибок в точку . Але з двох точок і потрібно вибрати той маршрут, який має найменшу вартість. Отримали рекурентне співвідношення:

.

Обчислити значення цільової функції також краще за допомогою методу динамічного програмування, а не рекурсії:

var c, price: array [0..100] of longint;

   i, n: longint;

function min(a,b:longint):longint;

begin

if a<b

then min:=a

else min:=b;

end;

begin

readln(n);

for i: = 1 to n do

read(price[i]);

for i: = 0 to n do

c[i]: = 0;

c[1]: = price[1];

for i: = 2 to n do

c[i]: = min(c[i - 1], c[i - 2]) + price[i];

writeln(c[n]);

end.

Після виконання цього циклу в списку буде записана мінімальна вартість маршруту для всіх точок від до .

**ВІДНОВЛЕННЯ ВІДПОВІДІ**

Але крім знаходження найменшої вартості маршруту, зрозуміло, хотілося б знайти і сам маршрут мінімальної вартості. Таке завдання називається задачею «відновлення відповіді».

Для відновлення відповіді будемо для кожної точки запам’ятовувати номер точки , з якої коник потрапив в точку , якщо він буде пересуватися по шляху мінімальної вартості. Тобто ‒ це точка, що передує точці з номером на шляху мінімальної вартості (також кажуть, що ‒ це масив попередників). Як визначити ?

Якщо , то коник потрапив в точку з точки , тому інакше . Модифікуємо алгоритми обчислення значень цільової функції, одночасно обчислюючи значення .

Приклад програми на мові Free Pascal:

var c, price, prev: array[0..100] of longint;

  j, i, n: longint;

begin

readln(n);

 for i := 1 to n do

    read(price[i]);

 for i := 0 to n do

    c[i] := 0;

 c[1] := price[1];

prev[1] := 0;

 for i := 2 to n do

   if c[i - 1] < c[i - 2]

then begin

      c[i] := c[i - 1] + price[i];

      prev[i] := i - 1;

end

else begin

      c[i] := c[i - 2] + price[i];

      prev[i] := i - 2;

   end;

...

end.

Тепер для відновлення шляху необхідно почати з точки n і переходити від кожної точки до її попередника, поки шлях не дійде до початкової точки з номером 0. Номери всіх вершин будемо додавати в список (масив) *Path*.

  i := n;

  j := 1;

  while i > 0 do

  begin

    Path[j] := i;

    i := Prev[i];

    inc(j);

  end;

  Path[j] := 0;

  for i := j downto 2 do

write(Path[i], ‘ ‘);

writeln(Path[1]);

У кінці в список *Path* додається початкова вершина з номером 1, яка не була оброблена в основному циклі, а потім весь список *Path* розгортається в зворотному порядку (бо вершини додаються в зворотному порядку, від кінцевої до початкової).

Але можна обійтися і без списку Prev. Ми в будь-який момент можемо визначити, з якої точки коник прийшов в точку i, якщо порівняємо і . Тому рішення про те, до якої вершині переходити при відновленні відповіді можна приймати безпосередньо при відновленні відповіді, порівнявши і

Шлях відновлюється через ту вершину, для якої значення C буде менше.

var c, price, path: array[-1..100] of longint;

  j, i, n, prev: longint;

function min(a,b:longint):longint;

begin

if a<b

then min:=a

else min:=b;

end;

begin

  readln(n);

  for i := 1 to n do

    read(price[i]);

  for i := 1 to n do

    c[i] := 0;

  c[-1] := -1;

c[0] := 0;

  c[1] := price[1];

  for i := 2 to n do

    c[i] := min(c[i - 1], c[i - 2]) + price[i];

  i := n;

  j := 1;

  path[j] := i;

  while i > 0 do

  begin

    inc(j);

    if c[i - 1] < c[i - 2]

then

  prev := i - 1

    else

      prev := i - 2;

    path[j] := prev;

    i := prev;

  end;

  path[j] := 0;

  for i := j downto 1 do

    write(path[i], ‘ ‘);

writeln(path[0]);

end.

**2. Динамічне програмування з двома параметрами: таблиці**

Розглянемо задачі, схожі на задачі **динамічного програмування з одним параметром**, але замість одновимірних задач на рух по прямій будуть розглядатися переміщення в двовимірному просторі ‒ наприклад, переміщення на шахівниці або на аркуші паперу в клітинку.

**Підрахунок числа маршрутів**

Розглянемо шахівницю в лівому верхньому кутку якої знаходиться король. Король може рухатися тільки вправо, вниз або по діагоналі вправо-вниз на одну клітку. Необхідно визначити кількість різних маршрутів короля, що приводять його в правий нижній кут.

Співставимо кожному полю шахівниці її координати , де *i* буде позначати номер рядка на дошці, *j* ‒ номер стовпця. Нумерувати рядки будемо зверху вниз, стовпці ‒ зліва направо, нумерація починається з 0. Тоді початкове положення короля буде поле .

Позначимо через кількість способів прийти з поля в поле . У поле можна прийти з трьох полів ‒ зліва з , зверху з і по діагоналі з . Тому число маршрутів, які ведуть в поле дорівнює числу маршрутів з усіх її попередників, а саме:

Окремо потрібно задати значення для граничних полів, тобто коли або . У результаті вийде таблиця заповнена наступним чином:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 3 | 5 | 7 | 9 |
| 1 | 5 | 13 | 25 | 41 |
| 1 | 7 | 25 | 63 | 129 |
| 1 | 9 | 41 | 129 | 321 |

Для заповнення цієї таблиці і підрахунку числа маршрутів можна використовувати наступний фрагмент програми, в якому спочатку створюється двовимірна таблиця, потім заповнюються крайні поля (перший стовпець і перший рядок), потім заповнюються інші елементи таблиці за допомогою наведеного вище рекуррентного співвідношення. У даному прикладі n ‒ число рядків, m ‒ число стовпців на дошці.

…

fillchar(f,sizeof(f),0);

for i:=1 to n do

f[i, 1]:=1;

for j:=1 to m do

f[1, j]:=1;

for i:=1 to n do

for j:=1 to m do

f[i, j]:= f[i, j-1] + f[i-1, j] + f[i-1, j-1];

На цьому прикладі можна скласти загальний план розв'язання задачі методом динамічного програмування. Його можна використовувати для розв’язання будь-яких задач за допомогою динамічного програмування:

1. Записати те, що потрібно знайти в задачі, як цільову функцію від деякого набору аргументів (числових, рядкових або ще яких-небудь).

2. Звести розв’язання задачі для довільного набору параметрів до розв’язання аналогічних підзадач для інших наборів параметрів (як правило, з меншими значеннями параметрів). Якщо завдання нескладне, то корисно буває виписати явне рекурентне співвідношення, що задає значення функції для даного набору параметрів.

3. Задати початкові значення функції, тобто ті набори аргументів, при яких задача є тривіальною і можна явно вказати значення функції.

4. Створити масив (або іншу структуру даних) для зберігання значень функції. Як правило, якщо функція залежить від одного цілочисельного параметра, то використовується одновимірний масив, для функції від двох цілочисельних параметрів - двовимірний масив і т. д.

5. Організувати заповнення масиву з початкових значень, визначаючи черговий елемент масиву за допомогою виписаного на кроці 2 рекуррентного співвідношення або алгоритму.

Для заповнення першого рядка і першого стовпця таблиці ми використовували «спеціальну» формулу, що відрізняється від загального випадку. Але в деяких завданнях зручніше буває всі значення обчислювати за допомогою однієї і тієї ж формули, а для граничних значень функції ввести спеціальні «фіктивні» елементи. У цій задачі можна зробити так само ‒ введемо спеціальну «облямівку» з одного фіктивного стовпчика ліворуч і одного фіктивного рядка зверху таблиці.

Для того, щоб значення в решті таблиці обчислювалися за загальною формулою, в усі комірки окантування потрібно записати число 0, крім комірки (0,0), в яку буде записано значення 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 3 | 5 | 7 | 9 |
| 0 | 1 | 5 | 13 | 25 | 41 |
| 0 | 1 | 7 | 25 | 63 | 129 |
| 0 | 1 | 9 | 41 | 129 | 321 |

Тепер у всіх інших комірках таблиці значення можуть бути обчислені за загальною формулою: , а фрагмент програми може виглядати так:

fillchar(f,sizeof(f),0);

f[0, 0]:=1;

for i:=1 to n do

for j:=1 to m do

f[i, j]:= f[i, j-1] + f[i-1, j] + f[i-1, j-1];

**Маршрут найменшої вартості**

Тепер розв’яжемо задачу про знаходження маршруту мінімальної вартості з лівого верхнього кута в правий нижній, вважаючи що для кожної клітини вказана вартість проходу через цю клітину. Відразу ж будемо вважати, що таблиця забезпечена «облямівкою», тому початкова клітина матиме індекси , кінцева клітина ‒ , а рядок і стовпець з індексом 0 будуть відноситися до фіктивного окантування.

Якщо вважати, що вартість проходу через клітину записана в окремому списку , то позначивши через вартість найкоротшого шляху з початкової клітини в клітину отримаємо рекурентне співвідношення:

При обчисленні граничних значень (в першому стовпці і першому рядку) необхідно враховувати тільки клітини з цього стовпчика і цього рядка (але не з попереднього стовпчика і попереднього рядка). Це зручно реалізувати, якщо заповнити попередній рядок і попередній стовпець «облямівкою», записавши для комірок окантування значення функції C рівними деякому дуже великому числу. Позначати це число як INF. За значення INF слід взяти число, яке більше, ніж максимально допустиме в задачі число. Воно може бути записано в таблиці C. А в кут «окантовки» потрібно записати число 0: C[0,0]:= 0.

Тоді фрагмент розв’язання методом динамічного програмування запишеться:

Const inf=100000000000000;

…

Fillchar(c,sizeof(c),0);

For i:=1 to n+1 do c[I,0]:=inf;

For j:=1 to m+1 do c[0,j]:=inf;

For i:=1 to n+1 do

For j:=1 to m+1 do

C[I,j]:=min(c[I,j-1],c[i-1,j-1])+price[I,j];

Література:

1. Алгоритмы: построение и анализ, 3-е изд. / Т. Х. Кормен, И. И. Лейзер-сон, Р. Л. Ривест, К. Штайн. – М.: Изд-ский дом «Вильямс», 2013. – C. 392-447.
2. Ахо А. Структуры данных и алгоритмы. / А. Ахо, Д. Хопкрофт, Д. Уль-ман. – М: Вильямс, 2007. – С. 280-288.
3. Дасгупта С. Алгоритмы. / С. Дасгупта, Х. Пападимитриу, У. Вазирани; Пер. с англ. под ред. А. Шеня. ‒ М.: МЦНМО, 2014. ‒ С. 156-175.
4. Котов В.М. Информатика. Методы алгоритмизации. 8-9 классы. / В.М. Котов, И.А. Волков, А.И. Лапо. – Мн.ИГП Нар.асвета, 2000. – С. 176-235.
5. Окулов С. М. Программирование в алгоритмах. – М.: БИНОМ. Лаборатория знаний, 2002. – C. 96-104.

При написанні матеріалу за основу взято матеріал курсу Дениса Кірієнка із сайту foxford.ru/wiki/informatika/