**Динамично оптимиране +**

**Динамично оптимиране. Задача за разделянето**

*Преслав Наков*

*preslav@rila.bg*

*сп. Byte/Bulgaria, 11/1999*

Често за решаването на една задача се оказва удачно тя да бъде разбита на по-малки подзадачи, които се решават по-лесно. По-малките подазачи често са по-прости за решаване и дават възможност за съсредоточване върху някои детайли и частни случаи, които не са така очевидни при изходната задача. Освен това подходящото разбиване ни дава съответно рекурсивно решение. Съществуват най-общо две ефективни алгоритмични схеми, основаващи се на разбиване на изходната задача на подзадачи: *динамично оптимиране* и *разделяй и владей*. В програмирането римският принцип “*Разделяй и владей*” най-често се свежда до разделяне на изходната задача на две подзадачи с еднаква или приблизително еднаква размерност. На практика няма никакви ограничения върху броя на подзадачите, както и върху тяхната размерност, като единственото изискване, което се поставя, е никои две подзадачи да не се пресичат и решаването им да дава възможност за директно получаване на решение на изходната задача. От своя страна динамичното оптимиране е свързано най-често с премахване на единствен елемент, а подзадачите му се задължително се пресичат. По-долу въпросът ще бъде разгледан по-строго и ще бъдат приведени две необходими условия за прилагане на метода.

Една възможна дефиниция на динамичното оптимиране е намиране оптимума (минимум или максимум) на функция, имаща за аргумент друга функция. По-общо казано динамичното оптимиране би могло да се разглежда като начин на мислене или програмиране, при който, чрез намиране на оптимални решения за част или всички подмножества на дадено множество, може да се намери оптималното решение за цялото множество.

Горната дефиниция би могла да се стори на читателя прекалено математическа и в известна степен абстрактна. В действителност става въпрос за проста програмистка техника, подобна на “разделяй и владей”: задачата се разбива на подзадачи, те от своя страна отново се разбиват на подзадачи и т.н. до достигане на достатъчно прости задачи, които могат да се решат директно. След това решенията на подзадачите се комбинират по подходящ начин, така че да се получи решение н изходнта задача. За разлика от “разделяй и владей” обаче, тук не се изисква множествата на подзадачите да са непресичащи се, което означава, че динамичното оптимиране е по-обща техника от “разделяй и владей”. Отказът от изискването за непресичане на подмножествата от задачи обаче води до проблеми. В общия случай са налице полиномиален брой подзадачи на изходната, но в процеса на декомпозиция се оказва че едни и същи задачи се решават няколко пъти, при което получаваме екпоненциална сложност на алгоритъма. Типичен пример в това отношение е следната функция за пресмятане числата на Фибоначи:

int fib(int n) {

if (n < 2) return n;

else return fib(n-1) + fib(n-2); }

Какво може да се направи, за да се ускори функция, подобна на горната? Едно очевидно решение е да се въведе таблица на всички вече пресметнати стойности на функцията. Всеки път, когато трябва да пресметнем fib(n) за някоя конкретна стойност на *n*, първо ще проверяваме, дали вече не сме я записали в таблицата и едва тогава, в случай че задчата все още не е била решавана, ще извършваме съответните пресмятания. Така достигаме до втора, чисто “програмистка”

***Дефиниция:*** *Запълването на таблица с резултатите от решенията на вече решените подзадачи с цел избягване на излишните пресмятания се нарича* ***динамично програмиране****.*

Макар основният принцип на динамичното оптимиране да е приложим и при задачи с единствено решение, като тази за намиране на *n*-тото число на Фибоначи, основното му приложение си остава решаването на оптимизационни задачи. Става въпрос за задачи с множество решения. На всяко решение се съпоставя някакво число с помощта на предварително дефинирана функция. Целта е да се намери такова решение, за което функцията приема своята оптимална (максимална или минимална) стойност. Прието е тази функция да се нарича *целева функция*. Възможно е задачата да има повече от едно оптимални решения, като в общия случай прилагането на динамичното оптимиране ще ни даде някое, но точно едно от тях.

Предимствата на динамичното оптимиране често се превъзнасят. Отсега обаче трябва да бъде ясно, че “безплатен обяд” няма. В основата на динамичното оптимиране стои решаването на подзадачи на изходната задача с по-малка размерност и запазване на вече пресметнатите резултати, т.е. печелим скорост срещу памет. В някои случаи (например числата на Фибоначи) е необходима константна памет, докато в други случаи необходимата памет може да бъде линейна (задача за раницата), квадратична (задачата за умножението на матрици), а понякога и още по-голяма.

Следва да се отбележи още, че динамичното оптимиране невинаги е приложимо. От една страна невинаги решението на изходната задача може да се получи чрез комбиниране на резултатите от решаването на част или всички нейни подзадачи. От друга страна, дори и когато такова комбиниране е възможно, може да се окаже, че броят на подзадачите, които следва да се разгледат е недопустимо голям. Към това следва да прибавим и липсата на ясен критерий, характеризиращ задачите, които могат да бъдат решени с помощта на описания метод: оказва се, че за редица задачи, стандартните алгоритми се оказват далеч по-ефективни от динамичното оптимиране, а за други — методът изобщо не е приложим.

Въпреки това съществуват две важни необходими условия за прилагане на метода: *оптимална подструктура на решението* и *припокриване на подзадачите*. Оптималната подструктура на решението означава, че оптималното решение на изходната задача може да бъде намерено като комбинация на *оптималните* решения на подзадачите. Обикновено изпълнението на този критерий ни води до намиране на подходящо “естествено” подпространство на подзадачите, които следва да бъдат разгледани. Така например при решаването на задачата за умножението на матриците, разгледана по-долу, ще видим, че е достатъчно да се разглеждат само подпоследователности на изходната последователност от матрици, а не произволни нейни подмножества. Това води до силно ограничаване множеството на подзадачите, откъдето и до по-голяма ефективност на метода. Второто необходимо условие за прилагане на динамичното оптимиране е припокриването на подзадачите. Динамичното оптимиране умее да се възползва умело от припокриването на подзадачите, пресмятайки решението на всяка от тях само веднъж, силно ограничавайки по този начин действителния брой решени подзадачи. Колкото по-рядко се налага действително решаване на нова подзадача, толкова по-ефективен е методът. Липсата на припокриване на подзадачите е сигурен индикатор, че прилагането на динамичното оптимиране е неподходящо. В такъв случай е добре да се опита с метода “разделяй и владей”, който, в случай че е приложим, ще даде по-добри резултати.

Често обектите, които се разглеждат при решаване на дадена задача, представляват множества с наредба на елементите, която можем да определим условно като наредба от тип “отляво-надясно”. Примери за такива обекти са символни низове, различни комбинаторни конфигурации (напр. пермутации, комбинации и др.), листата на наредени дървета за търсене, точки в върху права, върхове на многоъгълник и др. В тези случаи обикновено динамичното оптимиране води до ефективно решение.

Обикновено задачите от динамично оптимиране се решават итеративно отдолу-нагоре, т.е. първо се решават тривиалните подзадачи, след това тези, чиито подзадачи вече са решени и т.н. Основен недостатък на този подход е, че изисква решаване на *всички* подзадачи с размерност, по-малка от тази на изходната задача, при което често се пресмятат решенията на подзадачи, чието решаване не е нужно за решаване на изходната. По-долу ще разгледаме конкретни примери. Обикновено това не е толкова опасно и не води до съществено забавяне. Понякога обаче премахването на излишните пресмятания може да доведе до значително увеличение на скоростта. В такъв случай се прилага специален вариант на динамичното оптимиране, известен в англоезичната литература като *memoization*. Идеята е да се използва рекурсивна функция, при което да се обърне редът на пресмятане на решенията на подзадачите: отгоре-надолу. Преди да се извърши обръщение към функцията, масивът, в който пазим стойностите на решенията на подзадачите, се инициализира със специална стойност-индикатор, указваща че решението все още не е било пресмятано. Рекурсивната функция извършва пресмятане, само ако съответният елемент на масива съдържа въпросния индикатор, след което записва там резултата. В противен случай резултатът се взема наготово от таблицата. Така се пресмятат само тези подзадачи, които реално могат да участват в оптималното решение.

Динамичното оптимиране често се подценява, или по-точно — недооценява. Сред някои програмисти битува мнението, че то е сложно и объркано. В действителност става въпрос за една от най-простите и същевременно ефективни алгоритмични техники. Веднъж разбрал основните му принципи, за читателя ще бъде по-лесно откриването и съставянето на собствени схеми, отколкото откриването им литературата. Тъй като подобни знания се усвояват най-лесно на базата на конкретни примери, ще разгледаме една задача използваща посочения метод\*.

*(\*) Б.р.: Предложената задача е задачата с библиотеката от ЗМС’2000, която я има във файла bibliot.pdf, затова сметнахме за ненужно публикуването и тук.*

***Литература:***

*[Бонев-1996]* Бонев Ст., “Итерация или рекурсия — предимства и недостатъци”, сп. “PC&MAC World” 4/1996.

*[Калихман-1979]* Калихман И., М. Войтенко, “Динамическое программирование в примерах и задачах”, “Высшая школа”, Москва, 1979.

*[Brassard,Bratley–1987]* Brassard G., Bratley P., Fundamentals of Algorithmics, Prentice-Hall, 1996.

*[Cormen,Leiserson,Rivest–1998]* Cormen T., Leiserson C., Rivest R., Introduction to Algorithms, The MIT Press, 1998.

*[Parberry-1995]* Parberry I., “Problems on Algorithms”, Prentice-Hall, 1995.

*[Sedgewick-1990]* Sedgewick R., “Algorithms in C”, Addison-Wesley Publishing Company, New York, 1990.

*[Skiena-1997]* Skiena S., “The Algorithm Design Manual”, Springer-Telos, 1997.

**Задача C1. Максимална сума /НОИ 2009 – 1 кръг/**

Дадена е редица от *n* числа *a*1, *a*2, …, *an*. Да се напише програма **maxk**, която определя максималната сума на *k* последователни члена на редицата.

**Вход:** От първия ред на стандартния вход се въвеждат числата *n* и *k*. От втория ред се въвеждат числата от редицата.

**Изход:** На стандартния изход да се изведе търсената максимална сума.

**Ограничения:** 1 ≤ *k* ≤ *n* ≤ 1000, – 9999 ≤ *ai* ≤ 9999, *i* = 1, 2, …, *n*

**ПРИМЕР: Вход:** 6 2 **Изход:** 8

4 -2 3 5 1 3

Променяме масива по следния начин: запазваме стойността на първия елемент, а всеки следващ елемент е сумата от предходния и себе си, т.е a[i-1] + a[i]. Така i-тия елемент на масива ще представлява сума на елементите от a[0] до a[i]. И тогава, за да намерим сумата на k поредни елемента трябва от a[i] да извадим a[i-k].

#include <iostream>

using namespace std;

int main()

{

int a[1000], i, maxsum, n, k;

cin >> n >> k;

for (i =0; i<n; i++)

cin >> a[i];

for (i = 1;i<n; i++)

a[i] = a[i]+a[i-1];

maxsum = a[k-1];

for (i=k; i<=n-1;i++)

if ((a[i]-a[i-k])>maxsum)

maxsum = a[i]-a[i-k];

cout << maxsum <<endl;

return 0;}

**ЗАДАЧА C1. РОЗИ /НОИ 2011 – 2 кръг, Автор: Евгений Василев/**

Розовите храсти в градината ви са подредени в M реда и N колони. Метеоролозите прогнозират слани и това застрашава розовата реколта. За щастие имате голямо квадратно полиетиленово покривало, с което можете да направите импровизирана оранжерия и да спасите от измръзване част от храстите. Размерите на полиетилена са такива, че да обхванат K съседни реда и толкова колони от храстите. Колко рози ще загубите, ако знаете количеството им за всеки от храстите и разположите покривалото оптимално (т.е. да спасите максимално количество от измръзване)? "Редовете" и "колоните" разделят правоъгълника на единични квадратчета; квадратът е със страни, успоредни на страните на правоъгълника и покрива само цели квадратчета.

**Вход**: Програмата ви roses чете от първия ред на стандартния вход целите числа M, N и K. От всеки от следващите M реда се четат по N цели числа – броят на розите за поредния храст от реда.

**Изход**: На един ред на стандартния изход да се изведе търсеното количество измръзнали рози.

**Ограничения**: 0 < K < M, N ≤ 1000 (в 40% от тестовете M, N ≤ 100, в 70% от тях M, N ≤ 500). Няма храст с повече от 1000 рози.

**Пример Вход: Изход: 12**

3 4 2

1 1 1 2

2 3 4 1

2 1 9 2

**Пояснение**: Оцветеният квадрат в примера илюстрира мястото на защитното покривало.

**Решение: Бавно решение:** Въвеждаме в двумерен масив и едновременно с това пресмятаме сумата на всички клетки в масива. С два цикъла от 1 до m-k+1, и от 1 до n-k+1обхождаме всички клетки, които могат да бъдат начало на квадратна област с размер k. За всяка клетка a[i][j] пресмятаме сумата на клетките, попадащи в квадратната област – за целта използваме още два цикъла – и намираме максималната от всички суми. Това решение ще работи за около 60 т.

**Бързо решение чрез динамично оптимиране:** По време на въвеждане на стойностите в масива ще пресмятаме стойността на всяка клетка, така че тя да съдържа сумата на всички клетки в масива, които са наляво и нагоре от нея, включително и самата нея. Това става като зададем стойността на всяка клетка да е a[i][j] = a[i][j]+ a[i-1][j] + a[i][j-1] - a[i-1][j-1]; След това вече пресмятане на сума на клетките в правоъгълна зона се свежда само до сумиране на две клетки и изваждане на други две:  
s = a[i][j] - a[i-k][j] - a[i][j-k] + a[i-k][j-k];. Още можем да подобрим решението като направим входа и изхода чрез scanf и printf .

**//бавно решение**

#include <iostream>

using namespace std;

int a[1001][1001], m, n, k, maxs=0, sum=0, s,i, j;

int main()

{

cin>>m>>n>>k;

for (i=1; i<=m; i++)

for(j=1; j<=n; j++)

{

cin>>a[i][j];

sum=sum+a[i][j];

}

for (i=1; i<=m-k+1; i++)

for(j=1; j<=n-k+1; j++)

{

s=0;

for (int r=i;r<i+k;r++)

for (int q=j;q<j+k;q++)

s=s+a[r][q];

if (s>maxs)maxs=s;

}

cout << sum - maxs <<endl;

return 0;

}**//бързо решение**

#include <iostream>

using namespace std;

int a[1001][1001], m, n, k, maxs=0, s, i, j;

int main()

{

cin>>m>>n>>k;

for (i=1; i<=m; i++)

for(j=1; j<=n; j++)

{

cin>>a[i][j];

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

}

for (i=k; i<=m; i++)

for(j=k; j<=n; j++)

{

s = a[i][j] - a[i-k][j] - a[i][j-k] + a[i-k][j-k];

if (s > maxs) maxs = s;

}

cout << a[m][n]- maxs <<endl;

return 0;

}

[INFOMAN](http://infoman.musala.com/welcome/main.html) [брой 18](http://infoman.musala.com/magazine/archive/issue18/index.html)

/\*

Задача 5. Билети (давана и на ЗМС'97 8-9 клас)

Условие:

Даден е път между точките 1 и N (минаващ през 2, 3, ..., N-1 в този ред)

разстоянието между всеки две от тях има определена цена и може да бъде

изминавано само от по-малката към по-голямата. Да се намери такава серия от

прехвърляния, че да може да се измине разстоянието от 1 до N за минимална

цена.

Решение:

Ще използваме метода на динамичното оптимиране. Таблицата ни ще бъде

dist[i][j], (където i е началната, а j крайната точки.) Рекурентната връзка

ще е:

dist[1][j] = min( dist[1][k] + dist[k][j] ), за k = 2, ..., j - 1

За да намерим обратния път, за всяка точка j, ще помним във from[j] от коя

друга точка k сме дошли, и тъй като възможността е само една за всяка точка,

това ще ни гарантира запомнянето на обратния път. За да възстановим самия

обратен път, ще е нужно само да обходим таблицата отзад напред. За съжаление

по този начин намираме пътя в обратен ред и само за обръщането му са ни

нужни стека path и указателя pp.

Числата с които работим са цели, защото всяко число с фиксирана запетая може

да се представи като цяло, а компютърът извършва по-бързо и по-коректно

целочислени операции. Самото представяне на числа с 2 символа след запетаята

става така: F \* 100 = I където F е число с фиксирана запетая, а I цяло число.

\*/

#include <stdio.h>

#define MAX 100

double dist[5][5] = {{ 0, 0, 0, 0, 0 },

{ 0, 0, 3, 9, 20 },

{ 0, 0, 0, 7, 16 },

{ 0, 0, 0, 0, 10 },

{ 0, 0, 0, 0, 0 }};

int from[MAX];

int path[MAX], pp;

int n = 4;

main()

{

int i, j;

double v;

for( i = 1; i <= n; from[i++] = 1 );

for( i = 3; i <= n; i++ )

for( j = 2; j < i; j++ )

if( dist[1][i] > ( v = dist[1][j] + dist[j][i] ))

dist[1][i] = v, from[i] = j;

for( path[0] = n, pp = 1, i = n; i != 1;)

path[pp++] = i = from[i];

printf( "%d", --pp );

for( i = pp; i; i-- )

printf( "\n%d %d", path[i], path[i - 1] );

}

**Задача за раницата**

Дадена е раница с вместимост *M* килограма и *N* предмета, всеки от които се характеризира с две числа - тегло *mi* и стойност *ci*. Да се избере такова множество от предмети, чиято сумарна стойност е максимална, а сумата от теглата не надвишава *M*.

Дефинираме рекурентна целева функция:

*F*(0) = 0;  *F*(*i*) = max { *cj* + *F*(*i - mj*),  *j =* 1, 2, ..., *N*,  *mj ≤ i* }, *i >* 0

Методът на динамичното оптимиране изисква последователно пресмятане на стойностите на *F*(*i*), като за това пресмятане се използват вече пресметнатите стойности за по-малки *i*.

Да разгледаме примера:

N = 8;  
index  1  2  3  4  5  6  7  8  
m[8] 3, 7, 6, 1, 2, 4, 5, 5  
c[8] 5, 3, 9, 1, 1, 2, 5, 2  
M = 8;  
  
Fn[0] = 0;  
Fn[1] = max{c[4]+Fn[0]} = 1;   
 set[1][4]=1; set[1] = {0,0,0,1,0,0,0}  
  
Fn[2] = max{c[4]+Fn[1], c[5]+Fn[0]} = 1  
 set[2][5]=1; set[2] = {0,0,0,0,1,0,0}  
  
Fn[3] = max{c[1]+Fn[0], c[4]+Fn[2], c[5]+Fn[1]} =   
 max{5 +0, 1 +1, 1 +1} = 5  
 set[3][1]=1; set[3] = {1,0,0,0,0,0,0}  
  
Fn[4] = max{c[1]+Fn[1], c[4]+Fn[3], c[5]+Fn[2], c[6]+Fn[0]} =   
 max{5 +1, 1 +5, 1 +1, 2 +0} = 6  
 set[4][1]=1; set[4] = {1,0,0,1,0,0,0}   
  
Fn[5] = max{c[1]+Fn[2],c[4]+Fn[4],c[5]+Fn[3],c[6]+Fn[1],c[7]+Fn[0],c[8]+Fn[0]} =   
 max{5 +1, 1 +6, 1 +5, 2 +1, 5 +0, 2 +0} = 6  
 set[5][1]=1; set[5] = {1,0,0,0,1,0,0}  
  
Fn[6] = max{c[1]+Fn[3],c[3]+Fn[0],c[4]+Fn[5],c[5]+Fn[4],c[6]+Fn[2],c[7]+Fn[2],c[8]+Fn[1]} =   
 max{5 +5, 9 +0, 1 +5, 1 +6, 2 +1, 5 +1, 2 +1} = 9  
 set[6][3]=1; set[6] = {0,0,1,0,0,0,0}  
  
Fn[7] = max{c[1]+Fn[4],c[2]+Fn[0],c[3]+Fn[1],c[4]+Fn[6],c[6]+Fn[5],c[7]+Fn[2],c[8]+Fn[2]} =   
 max{5 +6, 5 +0, 9 +1, 1 +9, 2 +6, 5 +1, 2 +1} = 10  
 set[7][3]=1; set[7] = {0,0,1,1,0,0,0}  
  
Fn[8] = max{c[1]+Fn[5],c[2]+Fn[1],c[3]+Fn[2],c[4]+Fn[7],c[6]+Fn[4],c[7]+Fn[3],c[8]+Fn[3]} =   
 max{5 +6, 5 +1, 9 +1, 1 +10, 2 +6, 5 +1, 2 +1} = 10  
 set[8][3]=1; set[8] = {0,0,1,0,1,0,0}

#include <iostream>

using namespace std;

int N, M, m[1000], c[1000];

int Fn[1000];

bool set[1000][1000];

void calculate()

{ int maxValue, maxIndex, i, j;

for (i=1; i<=M; i++)

{ maxValue = maxIndex = 0;

for (j=1; j<=N; j++)

if (m[j]<=i && !set[i-m[j]][j])

if (c[j] + Fn[i-m[j]] > maxValue)

{ maxValue = c[j] + Fn[i-m[j]];

maxIndex = j;

}

if (maxIndex > 0)

{ Fn[i] = maxValue;

for(int k=0;k<=N-1;k++) set[i][k]=set[i-m[maxIndex]][k];

set[i][maxIndex] = 1;

}

if (Fn[i] < Fn[i-1])

{ Fn[i] = Fn[i-1];

for(int k=0;k<=N-1;k++) set[i][k], set[i-1][k];

}

}

cout << "Max value: " << Fn[M]<<endl;

cout<<"Taken things: ";

for (int i=1; i<=N; i++)

if (set[M][i]) cout << i << " ";

cout << endl;

}

int main()

{

cin >> M >> N;

for (int i=1; i<=N; i++) cin >> m[i] >> c[i];

calculate();

return 0;

}

/\*

Вход:

7

8

3 5

7 3

6 9

1 1

2 1

4 2

5 5

5 2

Изход: 10

Взети предмети: 3 и 4

\*/

**Разбиване на число**

Дадено е естествено число n. Под разбиване на числото n ще разбираме всяко множество от естествени числа, чиято сума е n. Например всички възможни разбивания за 4 са: 4, 3+1, 2+2, +1+1+2, 1+1+1+1.

Задачата е да се намери броя на всички разбивания на естественото число n.

Разглеждаме по-общата задача – да определим броя S(n, m) на всички разбивания на числото n на сума от числа, ненадминаващи m. Когато решим тази задача, търсеният брой разбивания на n ще бъде S(n,n).

Разписваме разбиванията на първите 6 числа на събираеми, ненадминаващи определена стойност m, за да открием зависимост.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Число** | **Сума от събираеми, ненадминаващи m=** | | | | | |
| **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 1 | 1 | 1 | 1 | 1 | 1 |
| **2** | 2 | 2  1+1 | 2  1+1 | 2  1+1 | 2  1+1 | 2  1+1 |
| **3** | 3 | 3  1+2 | 3  1+2  1+1+1 | 3  1+2  1+1+1 | 3  1+2  1+1+1 | 3  1+2  1+1+1 |
| **4** | 4 | 4  1+3  2+2 | 4  1+3  2+2  1+1+2 | 4  1+3  2+2  1+1+2  1+1+1+1 | 4  1+3  2+2  1+1+2  1+1+1+1 | 4  1+3  2+2  1+1+2  1+1+1+1 |
| **5** | 5 | 5  1+4  2+3 | 5  1+4  2+3  1+1+3  1+2+2 | 5  1+4  2+3  1+1+3  1+2+2  1+1+1+2 | 5  1+4  2+3  1+1+3  1+2+2  1+1+1+2  1+1+1+1+1 | 5  1+4  2+3  1+1+3  1+2+2  1+1+1+2  1+1+1+1+1 |
| **6** | 6 | 6  1+5  2+4  3+3 | 6  1+5  2+4  3+3  1+1+4  1+2+3  2+2+2 | 6  1+5  2+4  3+3  1+1+4  1+2+3  2+2+2  1+1+1+3  1+1+2+2 | 6  1+5  2+4  3+3  1+1+4  1+2+3  2+2+2  1+1+1+3  1+1+2+2  1+1+1+1+2 | 6  1+5  2+4  3+3  1+1+4  1+2+3  2+2+2  1+1+1+3  1+1+2+2  1+1+1+1+2  1+1+1+1+1+1 |

В следващата таблица отразяваме броя на разбиванията:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Число** | **Брой разбивания** | | | | | |
| **от 1 число** | **от 2 числа** | **от 3 числа** | **от 4 числа** | **от 5 числа** | **от 6 числа** |
| **1** | 1 | 1 | 1 | 1 | 1 | 1 |
| **2** | 1 | 2 | **2** | 2 | 2 | 2 |
| **3** | 1 | 2 | 3 | 3 | 3 | 3 |
| **4** | 1 | 3 | 4 | 5 | 5 | 5 |
| **5** | 1 | **3** | **5** | 6 | 7 | 7 |
| **6** | 1 | 4 | 7 | 9 | 10 | 11 |

От втората таблица се вижда, че първи ред и първа колона имат стойност 1.

Числата вдясно от главния диагонал са равни на числата по главния диагонал.

Числата по главния диагонал са с 1 по-големи от предходните им.

Числата под главния диагонал са сума от стойностите на две клетки – тази вляво от текущата, и тази същата колона, но на ред (n-m) , т.е  
а[5][3] = a[5][2]+a[2][3].

Обобщаваме получените резултати в следната формула:

S(n, m) =

Задачата се свежда до попълване на масив a[n][n] по зададената формула и извеждането на елемента a[n][n].

Решение:

#include <iostream>

using namespace std;

int n, a[1001][1001];

void calculate()

{

for (int i=1; i<=n; i++) a[i][1]=a[1][i]=1;

for (int i=2; i<=n; i++)

for (int j=2; j<=n; j++)

if (i==j) a[i][j]=1+a[i][j-1];

else if (i<j) a[i][j]=a[i][i];

else a[i][j]=a[i][j-1]+a[i-j][j];

}

int main()

{

cin >> n;

calculate();

cout << a[n][n] << endl;

return 0;

}

Решение на по-общата задача – да се намери броя на всички разбивания на числото n на сума от числа, ненадминаващи m.

#include <iostream>

using namespace std;

int n, m, a[1001][1001];

void calculate()

{

for (int i=1; i<=n; i++) a[i][1]=a[1][i]=1;

for (int i=2; i<=n; i++)

for (int j=2; j<=m; j++)

if (i==j) a[i][j]=1+a[i][j-1];

else if (i<j) a[i][j]=a[i][i];

else a[i][j]=a[i][j-1]+a[i-j][j];

}

int main()

{

cin >> n >> m;

calculate();

cout << a[n][m] << endl;

return 0;

}

## bgcoder.com – Telerik Algo Academy – 27-28.04.2012 – Динамично оптимиране

## Задача 3 – Скоби

### Дадена е последователност от символите "(", ")" и "?". Използвайки тази последователност, напишете програма, която намира броя възможни начини, с които могат да се заместят всеки от въпросителните знаци със скоба "(" или ")" с което да получите валиден израз от скоби. Ако вземете валиден математически израз и премахнете от него всичко друго освен скобите ще получите валиден израз от скоби. За пример: има точно 2 валидни израза от скоби с дължина точно 4: ()() и (()).

### Вход: Входните данни се четат от стандартния вход (конзолата). Последователността от скоби и въпросителни знаци ще бъде дадена на единствения ред от входа. Входните данни ще са винаги валидни и в описания формат.

### Изход: Изходните данните трябва да се изведат на стандартния изход (конзолата). На единствения ред на стандартния изход трябва да изведете броя валидни изрази от скоби.

### Ограничения: Дължината на последователността от символи е между 1 и 80, включително.

* Броят възможности за образуване на валидни изрази винаги е между -1018 и 1018.
* Разрешено време за работа на програмата: 0.10 секунди. Разрешена памет: 16 MB.

### Примери

|  |  |  |
| --- | --- | --- |
| **Примерен вход** | **Примерен изход** | **Обяснение** |
| ????(? | 2 | Решенията са: ()()**(**) и (())**(**) |
| ()(? | 1 | Има само 1 решение и то е: **()(**) |
| ?????? | 5 | Има 5 възможни решения: ((())), (())(), ()(()), (()()) и ()()() |

**Анализ**: Валиден израз от скоби е такъв израз, за който, във всеки един момент затварящите скоби не са повече от отварящите. Обхождаме низа и във всеки момент пазим информация за броя на символите до момента и разликата между отварящите и затварящите скоби в него. Тъй като параметрите са 2, ще използваме таблица за представянето на тази информация. Елементът DP[i][j] на масива ще съдържа информация за броя валидни изрази с дължина i, за които разликата от броя отварящи и броя затварящи е j.

Ще изразим текущото състояние чрез предишни състояния, чийто брой вече сме изчислили.

Низ с дължина 0 и разлика 0 е единствен – празния низ, затова DP[0][0] = 1

Низ с дължина 0 и всяка друга разлика няма – всички останали елементи на ред 0 са 0.

За всички останали елементи от ред 1 нататък важи следното:

Ако текущият символ е ‘(‘: DP[i][j] = DP[i-1][j-1] /броя изрази с дължина i е равен на броя изрази с дължина i-1, на които разликата на отварящите и затварящите скоби е с 1 по-малка./

Ако текущият символ е ‘)’: DP[i][j] = DP[i-1][j+1] /броя изрази с дължина i е равен на броя изрази с дължина i-1, на които разликата на отварящите и затварящите скоби е с 1 по-голяма./

Ако текущият символ е ‘?’: DP[i][j] = DP[i-1][j-1] + DP[i-1][j+1] /сумата от горните две/.

Тук трябва да обърнем специално внимание на елемента в първа колона, т.к. индекс j-1 става по-малък от 0, затова го пресмятаме отделно и след това обхождаме останалите елементи в реда.

Отговорът е в клетка DP[n][0] – броя на всички правилни изрази с дължина n и разлика на отварящи и затварящи скоби 0.

#include <iostream>

using namespace std;

long long DP[100][100];

int main()

{

string s;

cin >> s;

int n = s.length();

DP[0][0] = 1;

for(int i=1; i<=n; i++)

{

if (s[i-1]=='(' )DP[i][0] = 0;

else DP[i][0] = DP[i-1][1];

for (int j=1; j<=n; j++)

if (s[i-1]=='(') DP[i][j] = DP[i-1][j-1];

else if (s[i-1]==')') DP[i][j] = DP[i-1][j+1];

else DP[i][j] = DP[i-1][j-1] + DP[i-1][j+1];

}

cout << DP[n][0] << endl;

return 0;

}

Maycamp Arena - Състезание 1 (23.10.2009 - 26.10.2009) – Сребърна дивизия

**Задача 2. Лабиринт**

Имаме лабиринт под формата на пирамида, който е на N нива номерирани с числата от 1 до N, като ниво i има i клетки. Във всяка от клетките има карти със записано върху тях число. Всички карти в една клетка имат записано едно и също число. Имаме следната игра, която може да се играе от произволен брой играчи, които играят последователно:

Всеки играч влиза в клетката на върха на лабиринта (ниво 1) и трябва да излезе през основата (ниво N), преминавайки последователно през всяко едно от нивата на лабиринта. Всяка клетка има точно два изхода, водещи до клетките разположени диагонално на нея от долното ниво. Всеки играч е длъжен да вземе по точно една карта от всяка една от клетките, през които преминава. Печели играчът, чиято сума на събраните карти е най-висока.

Преди да започне играта, всеки играч получава план на лабиринта, в който е описано какви са числата върху картите в отделните клетки.

Ето един примерен план на лабиринта.

7 (Ниво 1)

/ \

3 8

/ \ / \

8 1 0

/ \ / \ / \

2 7 4 4

/ \ / \ / \ / \

4 5 2 6 5 (Ниво N)

Даден е лабиринт с N нива, напишете програма, която намира най-високия резултат, който може да бъде постигнат от някой от играчите.

**Вход:**

Програмата чете от стандартния вход.

На първия ред е записано едно цяло число N (1 <= N <= 350). Следват N реда (от 1 до N), описващи, числата в клетките от всяко едно ниво на лабиринта. На i-я ред са записани i числа, разделени с интервал, съответстващи на i-тото ниво (Всяко число е в интервала от 1 до 99).

**Изход:**

На стандартния изход трябва да бъде отпечатан най-големият резултат, който някой от играчите може да постигне по правилата на играта.

**Пример:**

**Вход:**

5

7

3 8

8 1 0

2 7 4 4

4 5 2 6 5

**Изход:**

30

**Обяснение на примера**:

1 Maycamp Arena - Състезание 1 (23.10.2009 - 26.10.2009) – Сребърна дивизия 2

Най-високият резултат, който може да се постигне за дадения пример става по следния начин:

7

\*

3 8

\*

8 1 0

\*

2 7 4 4

\*

4 5 2 6 5

**Време за изпълнение 1 секундa.**

**Ограничение на паметта: 256MB**

**Размер на стека: 1MB**