**ПЕРМУТАЦИИ. ОСНОВНИ СВОЙСТВА.**

**КОМБИНАТОРНИ КОНФИГУРАЦИИ. КОДИРАНЕ И ДЕКОДИРАНЕ.**

СП-2007-Комбинаторика, Програми =++ Алгоритми, Г. Момчева и др.

**1. Представяне на комбинаторните конфигурации**

Голяма част от състезателните задачи предполагат намирането на оптимално решение. В подобни ситуации често се налага да се изследват всички възможни решения и от тях да се избере най-доброто. Това най-често се прави чрез генериране на комбинаторни конфигурации. Важно е да се разбере обаче, че ако се търси броят на комбинаторни обекти, то тяхното генериране и едновременно изброяване е много лош (бавен) подход за решаването им и почти винаги може да се намери формула, която дава отговора много по-бързо.

**Задача 1:** Даден ни е граф, както и 5 специални върха от него. Искаме да минем през тях в такъв ред, че пътят, който ще извървим, да е минимален.

**Алгоритъм**: Намираме най-късите пътища между всички двойки върхове в графа чрез алгоритъма на Флойд. След това генерираме всички възможни пермутации на петте върха (те са общо 5! = 120). За всяка една пермутация изчисляваме дължината на пътя, който би се образувал, ако минем през върховете в конкретния ред, и избираме най-късия.

**Задача 2**: Всички са чували за играта, в която е дадена матрица *N*x*N* и в тази матрица стоят *N\*N - 1* квадратчета, номерирани с числата от *1* до *N\*N - 1* и има едно празно. Иска се чрез разместване квадратчетата да се подредят в правилния ред, т.е.:

(1, 2, ... N-1, N)

(N+1, N+2 ... 2\*N-1, 2\*N)

... ... ... ... ...

((N-2)\*N+1, (N-2)\*N+2 ... (N-2)\*N + N-1, (N-2)\*N + N)

((N-1)\*N+1, (N-1)\*N+2 ... (N-1)\*N + N-1, X)

където X е празното квадратче, което трябва да остане в последната кутийка – т.е. тази с индекс [N][N].

**Алгоритъм**: Тази задача, макар и замаскирано, е от раздел *графи* и се решава с BFS, като върховете на графа са отделните размествания на кутийките в матрицата. Интересно е обаче как отбелязваме даден връх като посетен. Нареждайки редовете на матрицата един до друг, можем да си представим, че получаваме вектор, чийто координати са числата от 1 до *N\*N-1*, разместени в даден ред. Така получаваме пермутация, която можем да кодираме в число между 0 и (*N\*N - 1*)! - 1 и много бързо да проверяваме дали даден връх е посетен или не.

Техниките от комбинаториката идват на помощ и при решаването на много други задачи като генериране на всички разбивания на число като сума/произведение от числа, разбиване на число като сума от дадени числа, разбиване на множества и т.н.

**Основни операции над комбинаторни конфигурации**

* Генериране
* Кодиране
* Декодиране

**2. Генериране на трите основни вида комбинаторни конфигурации**

**2.1. Генериране на пермутации**

**I начин – рекурсивен:** Ще използваме следния рекурсивен алгоритъм: всеки един от елементите поставяме на първа позиция, след което в останалите *n-1* позиции разполагаме последователно всички възможни пермутации на останалите *n-1* елемента. Функцията generatePermutations има едни параметър – позицията на която ще поставяме елемента. С вътрешен цикъл поставяме на позицията node всички елементи, които още не са използвани в текущата пермутация и продължаваме рекурсивно за следващата позиция. За да определим дали едни елемент е използван, използваме масив visited. Дъното на рекурсията е при node = n+1, тогава всички елементи са поставени и пермутацията е готова за отпечатване. Този алгоритъм генерира пермутациите лексикографски.

**Програмна реализация:**

#include <iostream>

#define MAX 32

using namespace std;

int n, visited[MAX], pos[MAX];

void print()

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

cout << pos[i] << " ";

cout << pos[n] << endl;

}

void generatePermutations(int node)

{

if(node == n+1)

{

print();

return;

}

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

if(!visited[i])

{ visited[i] = 1;

pos[node] = i;

generatePermutations(node+1);

visited[i] = 0;

}

}

int main()

{ cin >> n;

generatePermutations(1);

return 0;

}

**II начин – итеративен** **със сложност O(n):**

Описание на алгоритъма

Ще отбелязваме пермутaцията с P, а елементите й с pi

Първата пермутация е 1 2 3

Повтаряме следното

Обхождаме масива P отдясно-наляво

Търсим индекс i: p i < p i+1

Търсим индекс j: p i > p j , p j = min(p i+1, … p n)

Разменяме pi с pj

Обръщаме реда на елементите p i+1 ,…., p n

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
|  |  | i | j |
| 1 | 2 | 4 | 3 |
|  | i |  | j |
| 1 | 3 | 2 | 4 |
|  |  | i | j |
| 1 | 3 | 4 | 2 |
|  | i | j |  |
| 1 | 4 | 2 | 3 |

…

#include <iostream>

#define MAX 32

using namespace std;

int n, visited[MAX], pos[MAX];

void print()

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

cout << pos[i] << " ";

cout << pos[n] << endl;

}

bool nextPerm(int n, int \*perm)

{

int i = n;

while (i > 0)

{ i--;

int j = n;

while (j > i)

{ j--;

if (perm[i] < perm[j])

{ swap(perm[i], perm[j]);

i++;

j = n-1;

while (i < j)

{

swap(perm[i], perm[j]);

i++;

j--;

}

return true;

}

}

}

return false;

}

int main()

{

int perm[100];

int n;

cin >> n;

for (int i = 0; i < n; i++) perm[i] = i;

do

{ for (int i = 0; i < n; i++) cout << perm[i] << " ";

cout << endl;

}while (nextPerm(n, perm));

return 0;

}

**III начин – с функция next\_permutation от STL:**

#include<iostream>

#include<algorithm>

using namespace std;

int main()

{

int i, n, P[20];

cin>>n;

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

cin>>P[i];

do {

for (i=1; i<=n; i++) cout<<P[i]<<' ';

cout<<endl;

}

while (next\_permutation(P+1, P+n+1));

}

**IV начин –** пермутациите с размер (k+1) се получават от пермутациите с размер k, като отново използваме механизма на рекурсията. В този случай обаче не се налага да се използва допълнителен масив, за да означаваме използваните елементи. Алгоритъмът е следният:

1. При n = 1 се генерира единствена пермутация

2 Нека (p1,p2, p3…pk) е пермутация с размер k. Поставяме (k+1)-вия елемент съответно на първа, втора, трета и т.н. до (k+1)-ва позиция

Генериране на пермутации с вмъкване на n-ти елемент на всяко възможно място в пермутация от n-1 елемента

Пример

|  |  |  |
| --- | --- | --- |
| 1 | 12  21 | 312  132  123  321  231  213 |

Следващата програмна реализация /Наков/ е **модификация** на дадения алгоритъм:

#include <stdio.h>

#define MAXN 100

const unsigned n = 6;

unsigned a[MAXN];

void print(void)

{ unsigned i;

for (i = 0; i < n; i++) printf("%u ", a[i] + 1);

printf("\n");

}

void permut(unsigned k)

{ unsigned i, swap;

if (k == 0) print();

else {

permut(k - 1);

for (i = 0; i < k - 1; i++) {

swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;

permut(k - 1);

swap = a[i]; a[i] = a[k-1]; a[k-1] = swap;

}

}

}

int main(void) {

unsigned i;

printf("\n\n");

for (i = 0; i < n; i++) a[i] = i;

permut(n);

system("pause");

return 0;

}

**Забележка**: Ако дадена оптимизационна задача се свежда до генериране на пермутации, то е възможно да прекъснем текущото генериране, ако е сигурно, че няма да доведе до оптимално решение. Тогава преди рекурсивното извикване на функцията слагаме if и условието.

**2.2. Генериране на комбинации**

Идеята за генерирането на комбинации е същата като тази при пермутациите. За да получим обаче пермутациите без повторение, не е нужно да пазим visited[], а просто ще отсичаме рекурсията докъдето ни трябва - *for(int i = pos[node-1]+1; i <= n-k+node; i++)*. Право на това действие ни дава фактът, че генерираме комбинациите лексикографски. Следва пълна реализация на генерирането на комбинации от *n* елемента *k*-ти клас.

**I начин – рекурсивен:**

#include <iostream>

#define MAX 32

using namespace std;

int n, visited[MAX], k, pos[MAX];

void print()

{for(int i = 1; i < k; i++)

cout << pos[i] << " ";

cout << pos[k] << endl;

}

void generateCombinations(int node)

{if(node == k+1)

{ print();

return;

}

for(int i = pos[node-1]+1; i <= n-k+node; i++)

{ pos[node] = i;

generateCombinations(node+1);

}

}

int main()

{

cin >> n >> k;

generateCombinations(1);

return 0;

}

**II начин – итеративен:**

Лексикографски алгоритъм за генериране на комбинации {1, 2, 3, 4, 5}

Ще генерираме комбинации от по три елемента

1, 2, 3

1, 2, 4

1, 2, 5

1, 3, 4

1, 3, 5

1, 4, 5

2, 3, 4

2, 3, 5

2, 4, 5

3, 4, 5

**Описание на алгоритъма**

Отдясно наляво търсим елемент, който не е достигнал максимума (за позицията си). Ако намерим увеличаваме го с едно, а следващите след него са следващите естествени числа.

#include <iostream>

#define MAX 32

using namespace std;

int n, visited[MAX], k, pos[MAX];

void print()

{for(int i = 1; i < k; i++)

cout << pos[i] << " ";

cout << pos[k] << endl;

}

bool nextComb(int n, int k, int \*comb)

{int i = k - 1;

for (;i >= 0, comb[i] == n - k + i + 1; i--) ;

if (i < 0) return false;

comb[i]++;

for (i++; i < k; i++) comb[i] = comb[i - 1] + 1;

return true;

}

int main()

{int comb[100];

int n = 0, k = 1;

while (k > n || n < 1) cin >> n >> k;

for (int i = 0; i < k; i++) comb[i] = i + 1;

do

{

for (int i = 0; i < k; i++) cout << comb[i] << " ";

cout << endl;

}while (nextComb(n, k, comb));

return 0;

}

**2.3. Генериране на вариации**

Идеята по нищо не се различава от досега споменатите, като правим малки промени заради дефиницията на *вариация*:

**I начин – рекурсивен:**

#include <iostream>

#define MAX 32

using namespace std;

int n, pos[MAX], k, visited[MAX];

void print()

{for(int i = 1; i < k; i++)

cout << pos[i] << " ";

cout << pos[k] << endl;

}

void generateVariations(int node)

{if(node == k+1)

{ print();

return;

}

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

if(!visited[i])

{ visited[i] = 1;

pos[node] = i;

generateVariations(node+1);

visited[i] = 0;

}

}

int main()

{ cin >> n >> k;

generateVariations(1);

return 0;

}

**II начин – итеративен:**

#include <iostream>

#define MAX 32

using namespace std;

int n, pos[MAX], k, visited[MAX];

void print()

{for(int i = 1; i < k; i++)

cout << pos[i] << " ";

cout << pos[k] << endl;

}

bool nextVar(int n, int k, int \*var)

{int i = k - 1;

for (;i >= 0, var[i] == n; i--) var[i] = 1;

if (i < 0) return false;

var[i]++;

return true;

}

int main()

{int var[100];

int n = 0, k = 1;

while (k > n || n < 1) cin >> n >> k;

for (int i = 0; i < k; i++) var[i] = 1;

do

{ for (int i = 0; i < k; i++) cout << var[i] << " ";

cout << endl;

}while (nextVar(n, k, var));

return 0;

}

Ако искаме да генерираме вариации с повторение, просто изключваме действието на масива visited[]:

void generateVariations(int node)

{if(node == k+1)

{ print();

return;

}

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

{ pos[node] = i;

generateVariations(node+1);

}

}

**3. Алгоритъм за генериране на следваща пермутация**

1. Разглеждаме дадената пермутация отзад напред и намираме първата двойка числа, наредени в нарастващ ред. Нека ai-1 < ai. За пермутацията 563421 това са числата 3<4.

2. Разглеждаме всички числа от ai до края и намираме най-малкото от тях, което обаче е по-голямо от ai-1. Нека го означим с ak. За пермутацията 563421 това е числото 4.

3. Разменяме ai-1 и ak. Пермутацията 563421 става 564321

4. Сортираме елементите от ai до an в нарастващ ред и това е търсената пермутация. В разглеждания пример пермутация става 564123.

**В STL има функция next\_permutation(начало, край);**

**Програмна реализация:**

\*\*\*\*

**4. Кодиране**

Ще покажем как кодираме пермутации. Аналогично на генерирането, алгоритмите за кодиране и декодиране на останалите комбинаторни конфигурации са много близки до тези за пермутациите. Кодиране означава на всяка пермутация да се съпостави уникално естествено число, така че по него да може да се възстанови пермутацията /декодиране/. На всяка пермутация с *n* елемента можем да съпоставим уникално число, което е между *0* и *n!-1*. Алгоритъм за кодиране на дадена пермутация може да се направи по следната схема:

1. Нека *perm[]* е масив с *n* елемента, съдържащ пермутацията, която ще кодираме, а *p[]* е масив с *n* елемента такъв, че *p[i-1]* = *i* за *i* = *1*, *2*, ... *n-1*, *n*

2. Нека *pos* = *result* = *0*

3. Ако *pos < n*:

3.1. Изключваме елемента, заемащ *tmp*-тата позиция в *p[],* а на *tmp* присвояваме *i*, където *i* е индекс, за който *perm[pos]* == *p[i]*

3.2. Присвояваме *res* = *res \* (n-pos) + tmp*

3.3 *pos*++ и преход към стъпка 3

4. Извеждаме *res*.

**Програмна реализация**

#include <iostream>

using namespace std;

int codePermutation(int n, int \*perm)

{int \*p = new int[n+1], res = 0, tmp;

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

p[i-1] = i;

for(int pos = 0; pos < n; pos++)

{ tmp = 0;

while(perm[pos] != p[tmp])

tmp++;

res = res \* (n-pos) + tmp;

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

p[i-1] = p[i];

}

return res;

}

int main()

{int n, \*perm;

cin >> n;

perm = new int[n];

for(int i = 0; i < n; i++) cin >> perm[i];

cout << (codePermutation(n, perm)) << endl;

return 0;

}

Тест: На пермутацията *(5, 4, 3, 2, 1)* тази програма ще върне резултат *119*.

**5. Декодиране**

Декодиране означава по дадено число да се намери пермутацията, която има този номер в лексикографски генерираните пермутации с дължина е *n*. Нека е дадено число *num,* което е кода на пермутацията, която трябва да възстановим. Тогава алгоритъмът за декодиране е следният:

1. Нека *k* = *n - 1*, *p[]* е масив с *n* елемента такъв, че *p[i-1]* = *i* за *i* = *1*, *2*, ... *n-1*, *n*

2. Докато *k* >= *0*:

2.1. *m = n - k*

2.2. *perm[k] = num % m*

2.3. Ако *k > 0*, тогава *num /= m*

2.4. k--;

3. Присвояваме *k* = *0*. Докато *k < n* повтаряме:

3.1. m = perm[k]

3.2. perm[k] = p[m]

3.3. Ако *k < n*, тогава *for(int i = m+1; i < n; i++) p[i-1] = p[i]*

3.4. k++

**Програмна реализация**

#include <iostream>

using namespace std;

void decodePermutation(int num, int n, int \*perm )

{

int m, k = n, \*p = new int[n];

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

p[i-1] = i;

do

{ m = n - k + 1;

perm[k-1] = num % m;

if(k > 1)

num /= m;

}while(--k > 0);

k = 0;

do

{

m = perm[k];

perm[k] = p[m];

if(k < n)

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

p[i-1] = p[i];

}while(++k < n);

}

int main()

{

int n, num, \*perm;

cin >> n >> num;

perm = new int[n];

decodePermutation(num, n, perm);

for(int i = 0; i < n - 1; i++)

cout << perm[i] << " ";

cout << perm[n-1] << endl;

return 0;

}

**Тест**: n = 5, num = 119 => (5, 4, 3, 2, 1)

**Програмна реализация на кодиране и декодиране /Наков – файл codeperm.c/**

#include <stdio.h>

#define MAXN 100

const unsigned n = 6;

unsigned perm[MAXN] = { 5, 3, 6, 4, 2, 1 };

const unsigned long code = 551;

unsigned long codePerm(unsigned n, unsigned perm[])

{ unsigned p[MAXN], i, pos;

unsigned long r, result;

result = 0;

for (i = 0; i < n; i++) p[i] = i + 1;

for (pos = 0; pos < n; pos++) {

r = 0;

while (perm[pos] != p[r]) r++;

result = result \* (n - pos) + r;

for (i = r + 1; i < n; i++) p[i - 1] = p[i];

}

return result;

}

void decodePerm(unsigned long num, unsigned n, unsigned perm[])

{ unsigned long r, m, k;

unsigned i, p[MAXN];

for (i = 0; i < n; i++) p[i] = i + 1;

k = n;

do {

m = n - k + 1;

perm[k - 1] = num % m;

if (k > 1) num /= m;

} while (--k > 0);

k = 0;

do {

m = perm[k]; perm[k] = p[m];

if (k < n)

for (i = m + 1; i < n; i++) p[i - 1] = p[i];

} while (++k < n);

}

int main() {

unsigned i;

printf("Kodirane na permutacia: %lu \n", codePerm(n, perm));

printf("Decodirane na permutaciata, otgovarqshta na chisloto %lu: ", code);

decodePerm(code, n, perm);

for (i = 0; i < n; i++) printf("%u ", perm[i]);

printf("\n");

return 0;

}

**Декодиране /Шиков/**

**Дадени са числа n и k. Да се намери пермутацията с k елемента, съответстваща на номер n в лексикографската наредба на пермутациите.**

1. Колко пермутации с k елемента започват с 1? Отг. (k-1)! Толкова са и пермутациите започващи с 2, 3 и т.н. Затова намираме p = n / (k-1)!. Това означава че са направени пермутациите с 1, 2, …, p. Следва пермутация с начало (p+1).

2. Пресмятаме остатъка r при деление на n на (k-1)! Сега вече разглеждаме пермутация с k-1 елемента с номер r.

3. Само за първата пермутация важи това, че тя започва с (p+1). Ако числата са подредени в масив, премахваме (p+1)я, който вече сме използвали. В следващата пермутация елементът ще е (p+1)я, но измежду останалите.

Пример: k=6, n=250. Общ брой пермутации: 6!=720.

(k-1)! = 5! = 120

n/120 = 250/120 = 2, r=10 => пермутацията започва с 3 => пермутацията е 3\_ \_ \_ \_ \_.. Премахваме от елементите и те остават 1, 2, 4, 5, 6.

Сега търсим пермутация с 5 елемента и то десетата от тях, т.к. n=10

n/(k-1)! = 10/4! = 10/24 = 0, r = 10 => на втора позиция поставяме 1

=> n=10, k=4 => n/(k-1)! = 10/3! = 10/6 =1 => на трета позиция поставяме второто останало число измежду 2, 4, 5, 6 => поставяме 4

n=4, k = 3 => n/(k-1)! = 4/2! = 4/2 = 2 => поставяме третото останало число ….

3 1 4 5 2 6

**Кодиране и декодиране /Момчева/**

За числата 1,2, 3, 4, 5, 6

a. кое число попада на 124та позиция?

b. коя е позицията на числото 321546?

Разгледайте следното разлагане

124 - 1 = 123,

= 1\*5! + 1\*2! + 1\*1!

= ((((1\*5 + 0)\*4 + 0)\*3 + 1)\*2 + 1)\*1 + 0,

^ ^ ^ ^ ^ ^

<--> 100110,

<--> 211221,

При това разлагане получаваме редица от числа, която ще използваме за получаване на пермутацията, отговаряща на номер 124:

Вземаме 2то най-малко число от множеството от числа {1, 2, 3, 4, 5, 6 }. Това е 2.

Вземаме 1то най-малко число от множеството от числа {1, 3, 4, 5, 6 }. Това е 1.

Вземаме 1то най-малко число от множеството от числа {3, 4, 5, 6 }. Това е числото 3.

Вземаме 2то най-малко число от множеството от числа {4, 5, 6 }. Това е числото 5.

Вземаме 2то най-малко число от множеството от числа {4, 6 }. Това е числото 6.

Вземаме 1то най-малко число от множеството от числа {4 }. Това е числото 4.

И получаваме числото 213564.

b. За да намерим номера по пермутация, обръщаме реда на описаните вече стъпки:

3,2,1,5,4,6 <--> 321211,

<--> 210100,

<--> ((((2\*5 + 1)\*4 + 0)\*3 + 1)\*2 + 0)\*1 + 0,

= 266,

= 267 - 1,

Тогава пермутацията 321546 е с номер 267.

**Зад. 1.** Да се намерят всички десетцифрени телефонни номера, в които всяка цифра участва точно по веднъж и такива, че, сумата на цифрите да е равна на 20.

Упътване. Можем да генерираме всички телефонни номера чрез пермутации. Така, ако при поставянето на първите няколко цифри в линейното нареждане забележим, че сумата надвишава 20, то няма смисъл да продължаваме с по-нататъшно генериране.

**Зад. 2. Сума Нула**

Нека са дадени числата a1, a2, …, an. Да се поставят операции „+” и „-” между всеки две числа, така че резултатът от пресмятането на получения израз да е 0.

Например, ако са дадени числата от 1 до 8, то някои възможни решения на задачата са:

1+2+3+4-5-6-7+8=0

1+2+3-4+5-6+7-8=0

Решение: Ще генерираме всички възможни вариации с повторение на n-1 елемента от втори клас т.е. всевъзможните наредени (n-1)-торки, съставени от 0 и 1 (което отговаря на знак „+” или „-” пред съответното число). За всяка такава (n-1)-торки ще проверяваме дали е решение на задачата.

#include <stdio.h>

#include <math.h>

const unsigned n = 8; /\* Брой числа в редицата \*/

int a[] = { 1, 2, 3, 4, 5, 6, 7, 8 };/\* Редица \*/

int sum = 0; /\* Търсена сума \*/

void checkSol(void)

{ unsigned i;

int tempSum = 0;

for (i = 0; i < n; i++) tempSum += a[i];

if (tempSum == sum) { /\* намерено е решение => отпечатваме го \*/

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

if (a[i] > 0) printf("+%d ", a[i]);

else printf("%d ", a[i]);

printf(" = %d\n", tempSum);

}

}

void variate(unsigned i)

{ if (i >= n) {

checkSol();

return;

}

a[i] = abs(a[i]); variate(i + 1);

a[i] = -abs(a[i]); variate(i + 1);

}

int main(void)

{ variate(0);

system( "pause" );

return 0;

}

**Зад. 3. Сума Нула /с операции +, - и \*)**

Нека са дадени числата a1, a2, …, an. Да се поставят операции „+”, „-” и „\*” между всеки две числа, така че резултатът от пресмятането на получения израз да е 0.

**Зад. 4. БАНКА**

Достъп до сейфа на една банка има само управителят й. Сейфът се отваря с **K**-числов код, в който са включени всички числа от **1** до **K**, записани в някакъв ред. Числата не се повтарят. Спазвайки политиката за сигурност на фирмата управителят не си е записал кода и след командировка в чужбина изведнъж осъзнава, че го е забравил. Като бивш състезател по информатика си спомня само, че кодът е с номер, задаващо число от интервала **[N,M]** и разликата на тези две числа не е по-голяма от **1000000**. Номер на код е поредният номер на кода, ако те са подредени в нарастващ (лексикографски) ред.

***Пример***За **K = 3**

**Номера Кодове**

1 1 2 3

2 1 3 2

3 2 1 3

4 2 3 1

5 3 1 2

6 3 2 1

За да не стане объркване при записа на едноцифрени и многоцифрени числа, между отделните числа има интервал. Напишете програма **banka**, която по дадени **N** и **M** извежда всички кодове с номера от **N** до **M**, които управителят на банката ще опита, за да отвори сейфа.

***Вход:***От стандартния вход се въвеждат: на първия ред число **K**; на вторият ред **N** и **M**, отделени с интервал.

***Изход:***На стандартния изход се извеждат на отделни редове кодовете с номера от зададения интервал.

***Ограничения:* 2 < K < 13, 0 < N < M < 400000000, 0 < M – N ≤ 1000000**

***Пример***

**Вход Изход**

3 1 3 2

2 4 2 1 3

2 3 1

**Упътване:** Ако генерираме всички от първата до номер N без да ги извеждаме, а изведем тези от N до M, ще изпълним 400 млн.пъти, а от ограничението M – N ≤ 1000000, следва чеможе да ги ограничим до 1 млн, ако генерираме директно М-тата пермутация и изведем следващите до N пермутации. Задачата се свежда до генериране на пермутация по зададен номер.

**Решение:**

#include <iostream>

#include <algorithm>

using namespace std;

int main () {

int k, n, m, a[1000], b[1000], fact [13]={1}, p=1, i, j=0, r, N;

cin>>k>>n>>m;

N=n;

for ( i = 0; i<k; i++) a[i]=i+1;

for ( i = 1; i<=k; i++) fact[i]=fact[i-1]\*i;

r = k;

while (r>0)

{

int d=n/fact[r-1];

n = n%fact[r-1];

if (n==0) {d--; n = fact[r-1];}

int br=0;

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

{

if (a[i]!=0) br++;

if (br==d+1)break;

}

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

a[i]=0;

r--;

}

n=N; if (m>fact[k]) m = fact[k];

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

{

for (int i=0; i<k-1; i++) cout <<b[i]<<" ";

cout<<b[k-1]<< endl;

next\_permutation (b, b+k) ;

n++;

}

return 0;

}

**Зад. 5. Кутии**

В една редица са наредени N кутии (1<=N<=20). Разполагаме с А броя червени топчета и В броя сини (0<=А<=15, 0<=B<=15). Червените, както и сините топчета, са неразличими помежду си. Топчетата могат да се поставят в кутиите. Разрешено е в една кутия да се поставят по няколко

топчета от двата вида или само от единия вид. Възможно е някои кутии да остават празни. Не е задължително всичките топчета да се поставят в кутиите. Съставете програма която намира по колко различни начина може да се осъществи описаното поставяне нa топчетата в кутиите.

РЕШЕНИЕ:

Поставената задача изисква преброяването на някакви комбинаторни конфигурации което обикновено не може да бъде извършено с генерирането на всички възможности поради времевото ограничение. Тогава можем да дефинираме функция която ни дава броя на различните поставяния на А червени топчета и B сини топчета в N кутии по гореописания начин:

A B

F(N,A,B) = ä ä F(N-1,A-i,B-j)

i=0 j=0

F(0,A,B) = 1

Верността на функцията е очевидна. На всяка стъпка тя избира колко от червените топчета да постави в кутията N (i броя), и колко от сините топчета да постави в същата кутия (j броя). Броя на поставянията се получава като сума от различните поставяния на всяка стъпка. Разбира се рекурсивна функция изчисляваща функцията F би била прекалено бавна. Затова можем да използваме динамично оптимиране и да пазим в таблица стойността на функцията. Тази стойност запазваме веднага след като сме изчислили стойността на функцията при определени три параметъра. Когато

повторно ни се наложи да изчисляваме функцията при тези три параметъра стойнотта можем да вземем от таблицата, защото тази стойност вече я имаме. Сложността на алгоритъма за изброяване е O(N\*A\*B), защото всяка стойност на функцията се изчислява само веднъж, а броя на различните параметри N, A и B, които можем да предадем на функцията е N\*A\*B. Самото изчисляване е реализирано с рекурсивна функция, но може да се направи и итеративно.

Важно е да се забележи, че стандартния тип int, не може да "побере" отговора на задачата. Затова е нужно да си реализираме собствени функции за работа с дълги числа. Те разбира се е нужно да поддържат само най-простата операция - събиране. Реализацията и е по-стандартния начин

за събиране на две числа, с пазене на пренос и намиране на текуща цифра на сбора. Възможността за предефиниране на оператори в С++, ни предоставя много удобен начин за реализиране на дългите числа, при коeто синтаксиса за работа с тях не се различава от синтаксиса за работа с обикновения тип int например.

Petko Minkov

\*/

#include <fstream.h>

#include <string.h>

#define MAXD 25

#define MAXN 21

#define MAXAB 16

#define max(a, b) ((a) > (b) ? (a) : (b))

class large {

public:

int dc; // брой на цифрите

char d[MAXD]; // самите цифри

large();

large(int);

large operator +(large);

void writeln();

};

large::large()

{ memset(d, 0, sizeof(d));

dc = 0;

}

large::large(int val)

{ memset(d, 0, sizeof(d));

dc = 0;

while(val)

{d[++dc] = val % 10;

val /= 10;

}

}

large large::operator +(large b)

{ int i, carry, v;

large res;

carry = 0;

for(i=1;i<=max(b.dc, dc);i++)

{ v = d[i] + b.d[i] + carry;

res.d[++res.dc] = v % 10;

carry = v / 10;

}

if(carry)

res.d[++res.dc] = carry;

return res;

}

void large::writeln()

{ int i;

for(i=dc;i>=1;i--)

cout << int(d[i]);

cout << endl;

}

/\* program data \*/

int n, a, b;

large t[MAXN][MAXAB][MAXAB];

void readdata()

{ ifstream fin("boxes.cpp");

fin.ignore(80, '\n');

fin >> n >> a >> b;

fin.close();

}

large calc(int n, int a, int b)

{ int i, j;

large sum;

if(n == 0)

return 1;

if(t[n][a][b].dc)

return t[n][a][b];

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

for(j=0;j<=b;j++)

sum = sum + calc(n - 1, a - i, b - j);

t[n][a][b] = sum;

return sum;

}

void main()

{ readdata();

calc(n, a, b).writeln();

}