CURS PREGATIRE ADMITERE UBB 2023

RECAPITULARE CLASA A XI-A + EXTRA

BACKTRACKING

- Backtracking este un procedeu de programare care pentru a determina rezultatul unei probleme, genereaza toate variantele posibile.
- Este important de precizat ca faptul ca genereaza toate variantele posibile, il face un algoritm foarte lent, care nu poate fi folosit pentru date de intrare foarte mari.
- Informaticienii il folosesc foarte mult pentru matematica, putand sa genereze toate Permutarile, Combinarile, Aranjamentele, Partitiile etc.
- Desi pare un algoritm care se foloseste mult pe siruri, acesta poate fi implementat si pe matrici pentru a determina numarul total de drumuri intre 2 pozitii sau alte probleme de acest tip

PERMUTARI

Acest algoritm afiseaza toate permutariile multimii {1,2,…,n}

Complexitatea este n * n! (n! numarul de permutari si n va “costa” afisarea)

```cpp
#include <iostream>
using namespace std;

// generarea permutarilor

int n; // elementele vor fi 1, 2, 3, ... , n -> n este in limita a 10 elemente
int x[11], P[11];

void afis(){
    for(int i = 1; i <= n; ++i)
        cout << x[i] << ' ';
    cout << "\n";
}

void back(int k){ // complexitate: n * n!
    for(int i = 1; i <= n; ++i)
        if(!P[i]){
            P[i] = 1;
            x[k] = i;
            if(k == n)
                afis();
            else back(k + 1);
            P[i] = 0;
        }
}

int main(){
    cin >> n;
    back(1);
    return 0;
}
```

COMBINARI

- Acest algoritm genereaza toate combinarile de n luate cate k
- Regula de generare a combinarilor, pentru a avea cea mai buna complexitatea este ca mereu x[i] > x[j], oricare ar fi 1 <= j < i

```cpp
#include <iostream>
using namespace std;

// generarea combinarilor de n luate cate k

int n, k; // datele de intrare
int x[11];

void afis(){
    for(int i = 1; i <= k; ++i)
        cout << x[i] << ' ';
    cout << '\n';
}

void back(int pas){
    for(int i = x[pas - 1] + 1; i <= n - (k - pas); ++i){
        x[pas] = i;
        if(pas == k)
            afis();
        else back(pas + 1);
    }
}

int main(){
    cin >> n >> k;
    back(1);
    return 0;
}
```

```cpp
#include <fstream>
using namespace std;

ifstream cin("aranjamente.in");
ofstream cout("aranjamente.out");

int n, k, P[21], x[21];

void afis()
{
    for (int i = 1; i <= k; ++i)
        cout << x[i] << ' ';
    cout << "\n";
}

void back(int pas)
{
    for (int i = 1; i <= n; ++i)
        if (!P[i])
        {
            P[i] = 1;
            x[pas] = i;
            if (pas < k)
                back(pas + 1);
            else
                afis();
            P[i] = 0;
        }
}

int main()
{
    cin >> n >> k;
    back(1);
    return 0;
}
```

ARANJAMENTE

- Acest algoritm genereaza toate aranjamentele de n luate k.
- De aceasta data nu avem conditia de la combianri.
- Complexitatea este aproximativ k * (n ^ k)

PROBLEMA "PLATA"

Cerința

Se consideră `n` tipuri de bancnote, cu valorile `v[1] v[2] ... v[n]`, ordonate strict crescător. Pentru fiecare tip de bancnote se știe numărul de bancnote disponibile `c[1] c[2] ... c[n]`. Se cere să se determine o modalitate de a plăti integral o sumă dată `S` cu bancnotele disponibile, astfel încât să se folosească cel puțin o bancnotă de fiecare tip.

Date de intrare

Programul citește de la tastatură numerele `n` și `S`, apoi valorile `v[1] v[2] ... v[n]` ale bancnotelor și apoi `c[1] c[2] ... c[n]`.

Date de ieșire

Programul va afișa pe ecran `n` numere, reprezentând o modalitate de plată a sumei `S`. Fiecare număr `x[i]` va reprezenta numărul de bancnote de valoarea `x[i]` folosite pentru plata sumei `S`.

Restricții și precizări

- `1 ≤ n ≤ 6`
- `1 ≤ S ≤ 1000`
- `1 ≤ v[i] ≤ 100`
- `1 ≤ c[i] ≤ 10`
- oricare variantă corectă de plată a sumei `S` va fi luată în considerare
- pentru toate seturile de date există soluție

REZOLVARE:

- Avand in vedere faptul ca backtrackingul este un algoritm extrem de lent, codarea unui algoritm efficient backtracking presupune eliminarea functiilor de tip “verifica” sau “valid”
- Acest mod il putem face minimizand numarul de pasi al algoritmului si verificand la fiecare pas validitatea problemei, nu doar la final.

```cpp
#include <iostream>
using namespace std;

int n, S, v[7], c[7], x[7];
bool ok = false;

void afis(){
    for(int i = 1; i <= n; ++i)
        cout << x[i] + 1 << ' ';
    ok = true;
}

void back(int k, int sum){
    for(int i = 0; i <= c[k] && sum + v[k] * i <= S && !ok; ++i){
        sum += i * v[k];
        x[k] = i;
        if(k == n && sum == S)
            afis();
        else if(k < n)
            back(k + 1, sum);
        sum -= i * v[k];
    }
}

int main(){
    cin >> n >> S;
    for(int i = 1; i <= n; ++i)
        cin >> v[i];
    for(int i = 1; i <= n; ++i){
        cin >> c[i];
        c[i]--;
        S -= v[i];
    }
    back(1, 0);
    return 0;
}
```

PROBLEMA "PARANTEZE"

Cerinţa

Se dă un număr natural par `n`. Generați toate șirurile de `n` paranteze rotunde care se închid corect.

Date de intrare

Fişierul de intrare `paranteze.in` conţine pe prima linie numărul `n`.

Date de ieşire

Fişierul de ieşire `paranteze.out` va conţine pe fiecare linie câte un șir de `n` paranteze rotunde care se închid corect. Șirurile vor fi afișate în ordine lexicografică, considerând paranteza deschisa `(` mai mică decât paranteza închisă `)`.

Restricţii şi precizări

- `1 ≤ n ≤ 20`, număr natural par

REZOLVARE

```cpp
#include <fstream>
using namespace std;

ifstream cin("paranteze.in");
ofstream cout("paranteze.out");

int n;
char s[21];

void afis(){
    cout << s + 1 << '\n';
}

void back(int k, int cnt_d, int cnt_i){
    if(cnt_d < n / 2){
        s[k] = '(';
        back(k + 1, cnt_d + 1, cnt_i);
    }
    if(cnt_i < cnt_d && k < n){
        s[k] = ')';
        back(k + 1, cnt_d, cnt_i + 1);
    }
    else if(cnt_i < cnt_d && k == n){
        s[k] = ')';
        afis();
    }
}

int main(){

    cin >> n;
    back(1, 0, 0);

    return 0;
}
```

```cpp
#include <fstream>
using namespace std;

ifstream fin("paranteze.in");
ofstream fout("paranteze.out");

int x[21], n;

void Afisare(){
    for(int i = 1; i <= n; ++i)
        if(x[i] == 1)
            fout << '(';
        else fout << ')';
    fout << '\n';
}

int Valid(int k){
    int pd = 0, pi = 0;
    for(int i = 1; i <= k; ++i)
        if(x[i] == 1)
            pd++;
        else pi++;
    if(pi > pd || pd > n/2)
        return 0;
   return 1;
}

void back(int k){
    for(int i = 1; i <= 2; ++i){
        x[k] = i;
        if(Valid(k)){
            if(k == n)
                Afisare();
            else back(k + 1);
        }
    }
}

int main(){
    fin >> n;
    back(1);
    return 0;
}
```

PROBLEMA "BACK_ME"

Cerința

Se citesc două numere naturale `n` și `m`. Afișați în ordine lexicografică toate cuvintele care sunt formate din `n` litere `E` și `m` litere `M` cu proprietatea că nu există mai mult de două litere `M` alăturate și nici mai mult de două litere `E` alăturate.

Date de intrare

Programul citește de la tastatură numerele `n` și `m`, separate prin spații.

Date de ieșire

Programul va afișa pe ecran pe linii separate cuvintele cerute.

Restricții și precizări

- `0 ≤ n ≤ 17`, `0 ≤ m ≤ 17`
- `0 ≤ n + m ≤ 28`
- Dacă nu există cuvinte care să respecte condițiile, atunci se va afișa `IMPOSIBIL`.

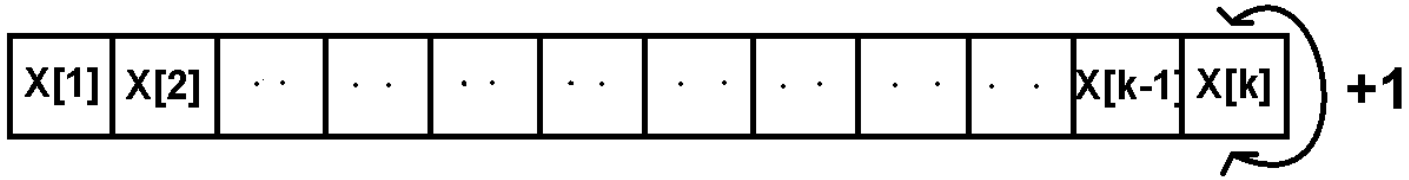
REZOLVARE

- Similar cu problema paranteze.
- Evitam generarea unor solutii gresite, astfel scazand mult complexitatea.
- Complexitatea unor astfel de solutii este chiar numarul de solutii finale, ceea ce duce la timpul de executie sa fie direct influentat de afisare si nu de algoritm.

```cpp
#include <iostream>
using namespace std;

int n, m, x[30];
bool ok = false;

void afis(){
    for(int i = 1; i <= n + m; ++i)
        (x[i]) ? cout << 'M' : cout << 'E';
    cout << "\n";
    ok = true;
}

void back(int k, int cntE, int cntM){
    if(k > n + m)
        afis();
    if(cntE > 0){
        if(k == 1 || x[k - 1] == 1)
            x[k] = 0, back(k + 1, cntE - 1, cntM);
        else if(x[k - 1] == 0 && (x[k - 2] == 1 || k == 2))
            x[k] = 0, back(k + 1, cntE - 1, cntM);
    }
    if(cntM > 0){
        if(k == 1 || x[k - 1] == 0)
            x[k] = 1, back(k + 1, cntE, cntM - 1);
        else if(x[k - 1] == 1 && (x[k - 2] == 0 || k == 2))
            x[k] = 1, back(k + 1, cntE, cntM - 1);
    }
}

int main(){
    cin >> n >> m;
    back(1, n, m);
    if(!ok)
        cout << "IMPOSIBIL";
    return 0;
}
```

MODUL DE GENERARE A BACKTRACKING-ULUI

- Se modifica ultima variabila pana nu se mai poate modifica cu valori mai mari.
- Se reseteaza ultima valoare, se modifica penultima valoare si din nou continua modificarea ultimei variabile pana la limita superioara.
- Se reseteaza ultima valoare, se incrementeaza penultima si continua procesul pana prima variabila ajunge la limita superioara.



EXEMPLU

- Permutarea multimii {1, 2, 3, 4}:
  - Initial, pornim de la 1,2,3,4
  - Pe ultima pozitie, nu mai putem pune o valoare mai mare, asa ca modificam penultima valoare de la 3, la 4 si obtinem permutarea 1,2,4,3
  - Pe penultima valoare nu mai putem modifica valoarea, asa ca pe antepenultima o facem 3 si rezulta sirul 1,3,2,4. Acum putem din nou sa modificam penultima valoare in 4 si avem permutarea 1,3,4,2
  - ETC.

DRUM IN MATRICE

```cpp
#include <iostream>
using namespace std;

// problema soarece

int n, m, xs, ys, xb, yb, cnt_drumuri;
bool a[11][11], P[11][11];

int dx[]={0,0,-1,1};
int dy[]={-1,1,0,0};

bool inmat(int x, int y){
    return x >= 1 && x <= n && y >= 1 && y <= m;
}

void back(int x, int y){
    if(x == xb && y == yb)
        cnt_drumuri ++;
    for(int d = 0; d <= 3; ++d){
        int xnou = dx[d] + x;
        int ynou = dy[d] + y;
        if(inmat(xnou, ynou) && P[xnou][ynou] == 0 && a[xnou][ynou] == 0)
            P[xnou][ynou] = 1, back(xnou, ynou), P[xnou][ynou] = 0;
    }
}
```

```cpp
int main(){
    cin >> n >> m;
    for(int i = 1; i <= n; ++i)
        for(int j = 1; j <= m; ++j)
            cin >> a[i][j];
    cin >> xs >> ys >> xb >> yb;
    P[xs][ys] = 1;
    back(xs, ys);
    cout << cnt_drumuri;
    return 0;
}
```

METODA GREEDY

- Metoda Greedy este o metoda folosita in algoritmica care presupune ca la fiecare posibilitate de a alege o continuare, sa se aleaga metoda cea mai favorabil la acel moment.
- Greedy = (in traducere din engleza) lacom.
- Metoda Greedy este o metoda foarte generala care nu duce mereu la un rezultat final corect, dar exista si multe probleme care se rezolva correct prin aceasta metoda. Decizia de a putea folosii acest concept sta in mainile programatorului.

METODA GREEDY

- Metoda Greedy este un concept si nu un algoritm, acest lucru este foarte important de inteles pentru a nu incerca folosirea unui sablon la acest capitol.
- Daca la Backtracking si la alte capitole existau niste algoritmi care se putea modifica sa mearga pe toate problemele, aici lucrurile stau usor diferit, intrucat modul de abordare al problemelor ar trebui sa fie unul natural.
- Cea mai usoara metoda de a concepe un algoritm de acest gen este simularea metodei naturale de a rezolva problema aceasta pe hartie.

METODA GREEDY

- Una dintre cele mai importante faze a rezolvarii unei probleme prin metoda Greedy este proiectarea algoritmului. Asa cum spuneam, aceasta metoda este una dintre cele mai naturale metode, intrucat presupune punerea in cod a rationamentului uman

- In general, se citeste problema si se incearca o rezolvare in cap a acesteia. Daca se ajunge la rezultatul final correct prin rationamentul uman, abordarea a fost una buna si vom incerca punerea in cod a acestuia.

UN EXEMPLU RELEVANT

- La aceasta problema, e natural ca pentru a maximiza numarul de masini care se repara, sa sortam masinile crescator dupa timpii de reparere si astfel sa vedem cate masini putem repara in timpul dat.

Cerinţa

Cunoscând timpul necesar pentru repararea fiecărei maşini, scrieţi un program care calculează numărul maxim de maşini care pot fi reparate într-un interval de timp `T`.

Date de intrare

Pe prima linie a fişierului `masini.in` se găsec două numere naturale `n` şi `T` separate printr-un singur spaţiu, reprezentând numărul de maşini din curtea atelierului auto şi timpul total în care se va lucra.

Pe linia a doua, separate prin câte un spaţiu, se găsesc `n` numere naturale t_1, t_2, ..., t_n, reprezentând timpii necesari pentru repararea fiecărei maşini.

Date de ieşire

Pe prima linie a fişierului `masini.out` se va găsi un număr natural `k`, reprezentând numărul maxim de maşini care pot fi reparate.

Restricţii şi precizări

- `1 < n, T <= 1000`
- numerele de pe a doua linie a fişierului de intrare vor fi mai mici sau egale cu `100`

Exemplu

`masini.in`

```
5 10
6 2 4 8 2
```

`masini.out`

```
3
```

Explicaţie

SOLUTIA:

```cpp
#include <fstream>
using namespace std;

ifstream cin("masini.in");
ofstream cout("masini.out");

// Greedy

int n, t;
int a[1001];

void sortare(int a[], int n){
    bool este_sortat = false;
    while(!este_sortat){
        este_sortat = true;
        for(int i = 1; i < n; ++i)
            if(a[i] > a[i+1]){
                este_sortat = false;
                swap(a[i], a[i+1]);
            }
    }
}
```

```cpp
int main(){
    cin >> n >> t;
    for(int i = 1; i <= n; ++i)
        cin >> a[i];
    sortare(a, n);
    int i = 1;
    while(t - a[i] >= 0 && i <= n)
        t -= a[i], i++;
    cout << i - 1;
    return 0;
}
```

PROBLEMA "SPECTACOLE"

Cerința

La un festival sunt programate `n` spectacole. Pentru fiecare se cunoaște momentul de început și momentul de sfârșit, exprimate prin numere naturale. Un spectator dorește să urmărească cât mai multe spectacole în întregime.

Determinați numărul maxim de spectacole care pot fi urmărite, fără ca acestea să se suprapună.

Date de intrare

Fişierul de intrare `spectacole.in` conţine pe prima linie numărul `n`. Pe fiecare dintre următoarele `n` linii se află câte două numere naturale `X Y`, reprezentând momentul de început și momentul de sfârșit al unui spectacol.

Date de ieşire

Fişierul de ieşire `spectacole.out` va conţine pe prima linie numărul `S`, reprezentând numărul maxim de spectacole care pot fi urmărite, fără să se suprapună.

Restricţii şi precizări

- `1 ≤ n ≤ 100`
- momentele de început și sfârșit ale spectacolelor sunt numere naturale mai mici decât `1.000.000`
- pentru fiecare spectacol, `X < Y`
- dacă momentul de început al unui spectacol și momentul de sfârșit al altui spectacol coincid, pot fi urmărite ambele spectacole

SOLUTIE

- Vom sorta intervalele dupa ora de final. De ce? Pentru ca cu cat ies mai repede de la spectacolul i (ca timp), am oportunitatea de a alege din cat mai multe spectacole ulterioare.
- Exact cum am proceda si in viata reala, de asta se numeste GREEDY.

```cpp
#include <fstream>
using namespace std;

ifstream cin("spectacole.in");
ofstream cout("spectacole.out");

// Metoda Greedy

struct spectacol{
    int start, finish;
}a[101];

int n;

void sortare_spectacole(spectacol a[], int n){
    for(int i = 1; i < n; ++i)
        for(int j = i + 1; j <= n; ++j)
            if(a[i].finish > a[j].finish)
                swap(a[i], a[j]);
}

int main(){
    cin >> n;
    for(int i = 1; i <= n; ++i)
        cin >> a[i].start >> a[i].finish;
    sortare_spectacole(a, n);
    int cnt = 1;
    int ora_fin = a[1].finish;
    for(int i = 2; i <= n; ++i)
        if(ora_fin <= a[i].start){
            cnt++;
            ora_fin = a[i].finish;
        }
    cout << cnt;
    return 0;
}
```

SEAMANA CU CEVA CUNOSCUT ACEASTA GRILA DIN 2021?

15. Se dă o mulțime ***S***, care conține ***n*** intervale specificate prin capătul stâng s_i și capătul drept d_i ($s_i < d_i \; \forall \; i = 1...n$). Se dorește determinarea unei submulțimi $S' \subseteq S$ de ***m*** elemente, astfel încât să nu existe două intervale în ***S'*** care se intersectează și ***m*** să aibă cea mai mare valoare posibilă.
Care dintre următoarele strategii rezolvă corect problema?

A. Se sortează intervalele din mulțimea ***S*** crescător după capătul stâng. Se adaugă primul interval din șirul sortat în ***S'***. Se parcurg celelalte elemente din șir în ordinea sortată și când se întâlnește un interval care nu se intersectează cu intervalul care a fost adăugat ultima oară în ***S'***, se adaugă și acesta în ***S'***.
B. Se sortează intervalele din mulțimea ***S*** crescător după capătul drept. Se adaugă primul interval din șirul sortat în ***S'***. Se parcurg celelalte elemente din șir în ordinea sortată și când se întâlnește un interval care nu se intersectează cu intervalul care a fost adăugat ultima oară în ***S'***, se adaugă si acesta în ***S'***.
C. Se sortează intervalele din mulțimea ***S*** crescător după lungimea intervalului. Se adaugă primul interval din șirul sortat în ***S'***. Se parcurg celelalte elemente din șir în ordinea sortată și când se întâlnește un interval care nu se intersectează cu intervalul care a fost adăugat ultima oară în ***S'***, se adaugă și acesta în ***S'***.
D. Se sortează intervalele din mulțimea ***S*** crescător după numărul de intervale din ***S*** cu care se intersectează. Se adaugă primul interval din șirul sortat în ***S'***. Se parcurg celelalte elemente din șir în ordinea sortată și când se întâlnește un interval care nu se intersectează cu intervalul care a fost adăugat ultima oară în ***S'***, se adaugă și acesta în ***S'***.

PROGRAMARE DINAMICA

- Programarea dinamica este un concept informatic care presupunerea calcularea a noi valori bazandu-ne pe valori precalculate.
- De asemenea, poate fi considerate si ca fiind o metoda de divizare a unei probleme in subprobleme, care rezolvandu-le sa ne duca la rezolvarea problemei initiale.
- Multe concept informatice se bazeaza pe programare dinamica, dar aceasta nu este predate prea amanuntit in scoli pentru ca nu este un capitol care sa intre la bac sau admitere.
- Un exemplu de programare dinamica pe care l-ati folosit fara sa stiti ca este programare dinamica este calculul sumelor partiale.

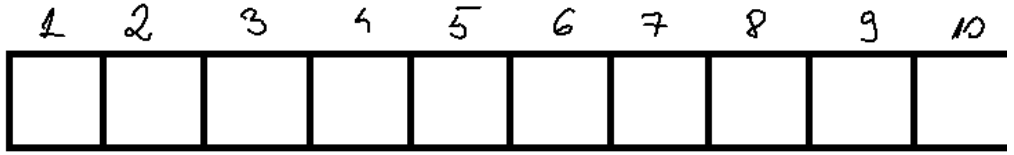
PROGRAMARE DINAMICA

- Conceptul de baza a programarii dinamice este divizarea problemei intr-o subproblema.
- De exemplu, daca ma intereseaza sa stiu o anumita proprietate despre un numar n, este sufficient sa determin modul in care foarte general pot raspunde la aceiasi proprietate referindu-se la o pozitie “i”, cuprinsa intre 1 si n
- Desigur, acest lucru poate suna destul de abstract, dar idea de baza este sa generalizam problema.
- In practica, ce vrea sa spuna conceptul e ca daca ne dorim sa calculam un rezultat pentru un intreg, trebuie sa calculam mai intai pentru toate subproblemele problemei.

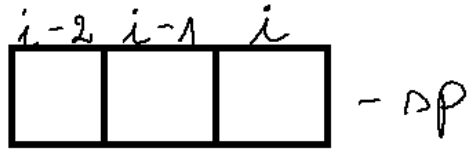
UN EXEMPLU RELEVANT

- Alaturat vedem cum se poate calcula intr-un mod dinamic suma elementelor unui vector.
- La fiecare pas, ne bazam pe alte valori calculate inainte si astfel, continuam calculul intr-un mod corect.

```
15      sp[1] = a[1];
16      for(int i = 2; i <= n; ++i)
17          sp[i] = sp[i-1] + a[i];
18      cout << sp[n];
```



Daca ne intereseaza suma elementelor vectorului A, definit mai sus, vom calcula suma celor n elemente. Subproblema problemei ar fi sa analizam aceasta situatie din prisma unui i oarecare din intervalul [1,n]. Astfel, oricare ar fi i, vom stii suma primelor i elemente. Suma primelor n elemente este suma primelor n-1 + elementul de la pozitia n. Astfel sp[i] = sp[i-1] + A[i] - formula calculului sumelor partiale

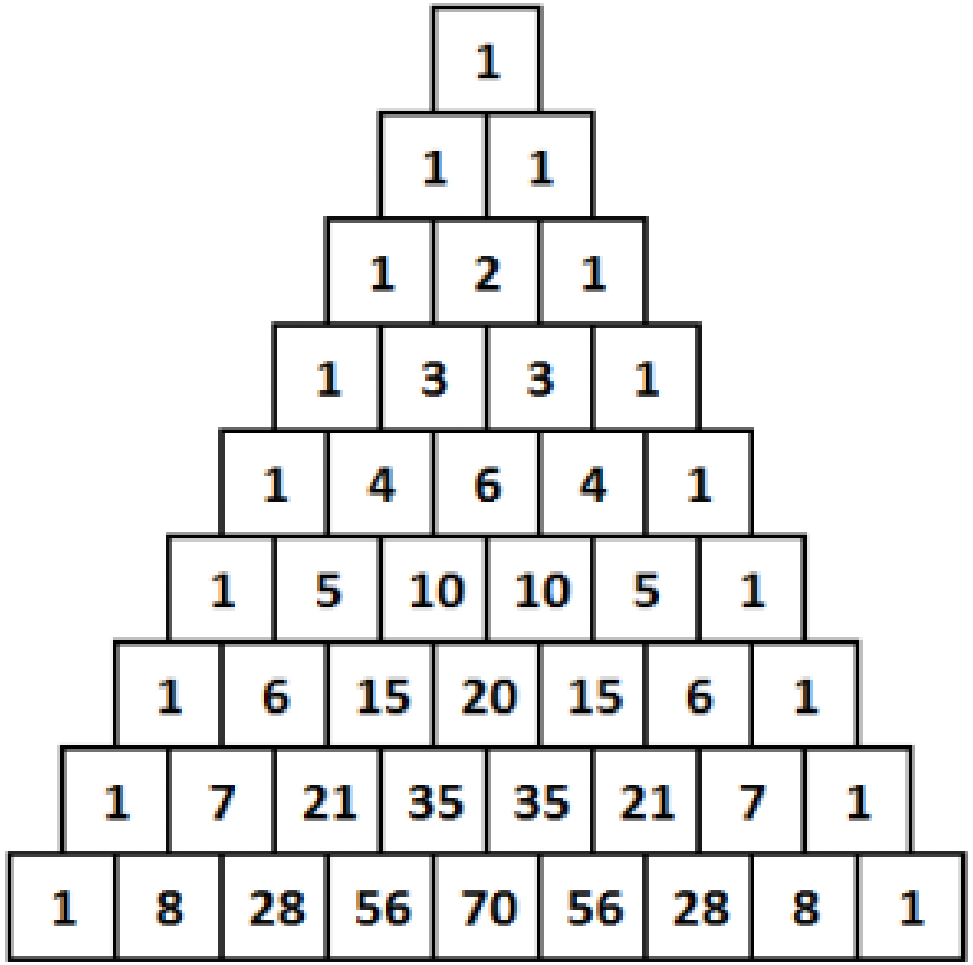


sp[i-1] memoreaza suma primelor i-1 valori din sir

sp[i] este egal cu suma primelor i-1 elemente + elementul de la pozitia i (a[i] + sp[i-1])

TRIUNGHIUL LUI PASCAL

- Triunghiul lui Pascal este un triunghi mathematic care se auto-genereaza dupa o recursive bine stabilita.
- Acesta calculeaza Combinari de n luate cate k dupa o formula clasica: C(n, k) = C(n-1,k-1) + C(n-1,k)
- Cum arata acest triunghi?



METODA RECURSIVA DE A GENERA C(N, K) CU TRIUNGHIUL LUI PASCAL

- Problema e numarul foarte mare de pasi, asemeni problemei fotbal.
- Din acest punct de vedere, o solutie mai buna ar fi programare dinamica, calculand intreg triunghiul din prima.

```cpp
#include <iostream>
using namespace std;

int C(int n, int k){
    if(k == 0 || n == k)
        return 1;
    return C(n-1, k) + C(n-1, k-1);
}

int main(){
    cout << C(5, 3);
    return 0;
}
```

SOLUTIA CU PROGRAMARE DINAMICA SI MATRICE

- Problema numarului mare de pasi, REZOLVATA!
- Totusi, avand in vedere ca folosim o matrice, complexitatea spatiu ar putea sa creasca foarte mult.
- Observam ca la fiecare calcul de noi elemente, folosim doar ce am calculat pe linia anterioara, de aceea vom putea implementa aceasta problema cu ajutorul unui singur vector.

```
1   #include <iostream>
2   using namespace std;
3
4   int c[10][10];
5
6   int C(int n, int k){
7       for(int i = 0; i <= n; ++i)
8           c[i][0] = c[i][i] = 1;
9       for(int i = 2; i <= n; ++i)
10          for(int j = 1; j < i; ++j)
11              c[i][j] = c[i-1][j-1] + c[i-1][j];
12      return c[n][k];
13  }
14
15  int main(){
16      int n, k;
17      cin >> n >> k;
18      cout << C(n, k);
19      return 0;
20  }
```

SOLUTIA CU DINAMICA PE VECTOR

- Aceasta este cea mai buna varianta posibila.
- Nu exista imbunatatiri posibile.
- Desigur, daca vrem, o putem scrie pe aceasta recursiv si obtinem o solutie cel putin la fel de buna.

```cpp
#include <iostream>
using namespace std;

void pascal(int niv, int a[]){
    int level = 1;
    a[1] = a[2] = 1;
    while(level < niv){
        level++;
        a[level + 1] = 1;
        for(int i = level; i >= 2; --i)
            a[i] = a[i] + a[i-1];
    }
}

int main(){
    int a[1001], n, k;
    cin >> n >> k;
    pascal(n, a);
    cout << a[k];
    return 0;
}
```

VARIANTA RECURSIVA BUNA

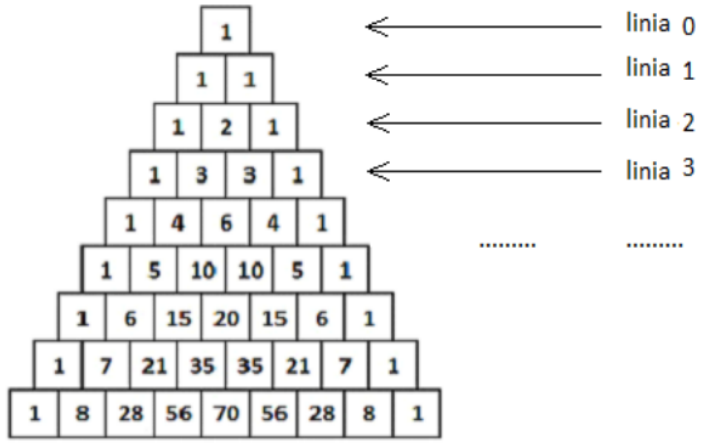
```cpp
#include <iostream>
using namespace std;

void pascal(int niv, int a[]){
    if(niv == 1){
        a[1] = a[2] = 1;
        return ;
    }
    pascal(niv - 1, a);
    for(int i = niv; i >= 2; --i)
        a[i] += a[i-1];
    a[niv + 1] = 1;
}

int main(){
    int a[1001], n, k;
    cin >> n >> k;
    pascal(n, a);
    cout << a[k];
    return 0;
}
```

POATE SEAMANA CU CEVA CUNOSCUT…

<u>Subiectul I</u> (50 puncte)

1. Triunghiul lui Pascal (20 puncte)

Triunghiul lui Pascal este un triunghi isoscel cu mai multe linii orizontale formate din numere naturale astfel: laturile egale conțin doar cifra 1, iar fiecare număr de pe o linie ***n*** reprezintă suma celor două numere vecine de pe linia superioară ***n*** – 1, pentru ***n*** > 1. Liniile sunt numerotate de sus în jos începând de la 0, ca în figura alăturată:



Scrieți un subalgoritm care generează numerele aflate pe linia ***r*** ($2 \leq r \leq 32$), *fără a folosi structuri de date bidimensionale*. Parametrul de intrare este numărul natural ***r***, iar parametrul de ieșire va fi șirul numerelor de pe linia ***r***.

SCLM (SUBSIR CRESCATOR DE LUNGIME MAXIMA)

Cerinţa

Se dă un șir cu `n` elemente, numere naturale. Determinați un cel mai lung subșir crescător al șirului.

Date de intrare

Fişierul de intrare `sclm.in` conţine pe prima linie numărul `n`, iar pe a doua linie `n` numere naturale separate prin spaţii, reprezentând elementele șirului.

Date de ieşire

Fişierul de ieşire `sclm.out` va conține pe prima linie numărul `L`, reprezentând lungimea maximă a unui subșir crescător. A doua linie va conține `L` numere cu valori între `1` și `n`, ordonate crescător, reprezentând indicii elementelor din șirul dat care dau subșirul crescător de lungime maximă.

Restricţii şi precizări

- `1 ≤ n ≤ 1000`
- numerele de pe a doua linie a fişierului de intrare vor fi cel mult egali cu `1.000.000`
- orice șir de indici care duce la subșir crescător de lungime maximă este corect

SOLUTIE:

```cpp
#include <fstream>
using namespace std;

ifstream cin("sclm.in");
ofstream cout("sclm.out");

int n;
int a[1001], t[1001];
int lmax[1001], rez;
int pozrez;

void drum(int val){
    if(val == 0)
        return ;
    drum(t[val]);
    cout << val << ' ';
}
```

```cpp
int main(){
    cin >> n;
    for(int i = 1; i <= n; ++i)
        cin >> a[i];

    for(int i = 1; i <= n; ++i){
        int lm = 0, poz = 0;
        for(int j = i - 1; j >= 1; --j)
            if(a[j] <= a[i] && lmax[j] > lm)
                lm = lmax[j], poz = j;
        lmax[i] = lm + 1;
        t[i] = poz;
        if(rez < lmax[i])
            rez = lmax[i], pozrez = i;
    }

    cout << rez << '\n';

    drum(pozrez);

    return 0;
}
```

INTERSCHIMBAREA A 2 VARIABILE

```
39  void swap4(int &x, int &y){
40      int aux = x;
41      x = y;
42      y = aux;
43  } // interschimbarea prin metoda clasica
```

```
21  void swap1(int &x, int &y){
22      x = x + y;
23      y = x - y;
24      x = x - y;
25  } // interschimbarea prin adunare
```

```
4   void swap2(int &x, int &y){
5       x = x - y;
6       y = x + y;
7       x = y - x;
8   } // interschimbare prin scadere
```

```
33  void swap3(int &x, int &y){
34      x = x * y;
35      y = x / y;
36      x = x / y;
37  } // interschimbarea prin inmultire
```

INTERSCHIMBAREA A 2 VARIABILE

```
45  void swap5(int &x, int &y){
46      swap(x, y);
47  } // interschimbarea prin swap
```

```
49  void swap6(int &x, int &y){
50      x = x ^ y;
51      y = x ^ y;
52      x = x ^ y;
53  } // interschimbarea pe biti (XOR)
```

CARE DINTRE FUNCTIILE DE MAI JOS, INTERSCHIMBA A SI B?

A.
```
void F(int& a, int& b){
      a-=b;
      b+=a;
      a=b-a;
}
```

B.
```
void F(int& a, int& b){
      a=a+b;
      b=a-b;
      a=a-b;
}
```

C.
```
void F(int& a, int& b){
      a=a/b;
      b=a*b;
      a=b/a;
}
```

D.
```
void F(int& a, int& b){
      a=a*b;
      b=a/b;
      a=a/b;
}
```

4. Fie subalgoritmul **prelucrare(x, n)** definit mai jos, care primește ca și parametru un șir **x** cu **n** numere reale nenule (x[1], x[2], …, x[n]) și numărul întreg **n** ($1 \leq n \leq 10000$).
Operatorul **/** reprezintă împărțirea reală (ex. **3/2=1,5**).

```
Subalgoritm prelucrare(x, n):
    p ← 1
    Pentru k ← 1, n - 1 execută
        p ← p + 1
        Pentru i ← 1, n - 1 execută
            Dacă x[i] > x[i + 1] atunci
                x[i] ← x[i] * x[i + 1]
                x[i + 1] ← x[i] / x[i + 1]
                x[i] ← x[i] / x[i + 1]
            SfDacă
        SfPentru
    SfPentru
    n ← p
SfSubalgoritm
```

Care dintre următoarele afirmații descriu modificarea aplicată șirului **x** în urma apelului subalgoritmului `prelucrare(x, n)`?

A. Elementele șirului **x** vor rămâne nemodificate
B. Elementele șirului **x** vor fi în ordine descrescătoare
C. Elementele șirului **x** vor fi în ordine crescătoare
D. Numărul **n** este decrementat cu o unitate

VA MULTUMESC PENTRU ATENTIE!