**MAX ȘI MIN PE FIECARE LINIE**

| static void MaxMinLinii(int[][] numbers, int length)  {  for(int i=0; i<length; i++)  {  //Resetam max si minim, după terminarea fiecărei linii.  int max=-99999999;  int min=99999999;    for(int j=0; j<length; j++)  {  if(numbers[i][j]>max) max=numbers[i][j];  if(numbers[i][j]<min) min=numbers[i][j];  }    System.out.println("Max Linia " + i + ": " + max);  System.out.println("Min Linia " + i + ": " + min);  }  } |
| --- |

**PRODUS SCALAR MAXIM**

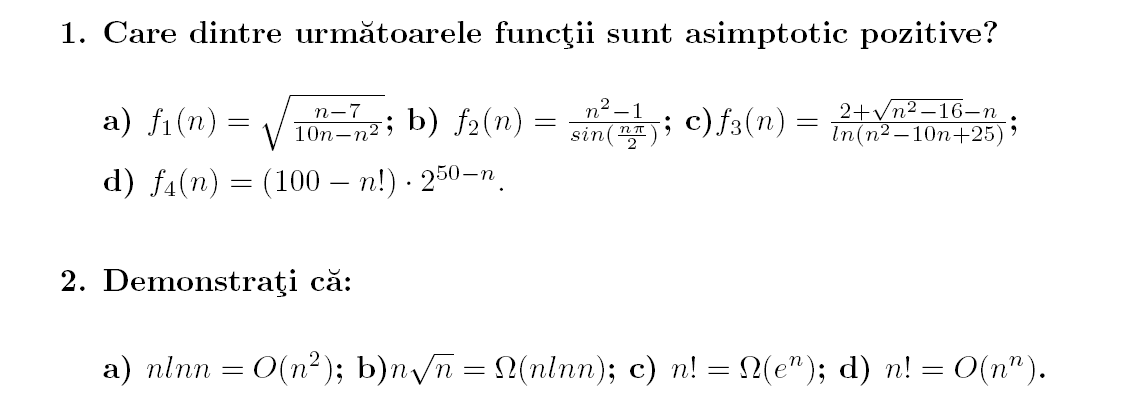
| static void SortareCrescatoare(int a[], int n)  {  int temp;    //Sortare de complexitate O(n^2), are 2 for-uri  for(int i=1; i<=n; i++)  {  for(int j=1; j<=n; j++)  {  //Cu interschimbare  if(a[i]<a[j])  {  temp=a[i];  a[i]=a[j];  a[j]=temp;  }  }  }  }  static void MAXIM2(int a[],int b[], int n, int s)  {  SortareCrescatoare(a,n);  SortareCrescatoare(b,n);  s=0;    for(int i=1; i<=n; i++){  s=s+a[i]\*b[i];  }    AFISARE(a,b,n,s);  }  //Sortând ambele tablouri **în ordine crescătoare**, se asociază **cei mai mici termeni cu cei mai mici termeni** și **cei mai mari termeni cu cei mai mari termeni**.  Acest lucru **maximizează produsul scalar**, ceea ce este corect. | static void AFISARE(int a[], int b[], int n,int s)  {  System.out.println("Suma="+s);  for(int i=1; i<=n; i++) System.out.print(a[i] + " ");    System.out.println();    for(int i=1; i<=n; i++) System.out.print(b[i] + " ");  }  public static void main(String[] args)  {  //Produs Scalar Maxim/Minim  int n,s=0,i,a[], b[];  Scanner cin=new Scanner(System.in);  System.out.print("Nr elemente=");  n=cin.nextInt();  a=new int [n+1];  b=new int [n+1];  System.out.print("Elementele din a[]: ");  for(i=1;i<=n;i++) a[i]=cin.nextInt();    System.out.print("Elementele din b[]: ");  for(i=1;i<=n;i++) b[i]=cin.nextInt();  MAXIM2(a,b,n,s);  } |
| --- | --- |

**PRODUS SCALAR MINIM**

Dacă dorim produsul scalar minim, atunci vem înmulții cel mai mare element dintr-un vector cu cel mai mic element din celălalt vector.

* Asta s-ar face sortând un vector crescător și celălalt descrescător.

| static void MINIM1(int a[],int b[], int n, int s)  {  s=0;  for(int i=1; i<=n; i++)  {  int k=i;  int m=a[i];  for(int j=i+1; j<=n; j++)  {  if(a[j]<m)  {  k=j;  m=a[j];  }  }  //Interschimbam  int temp=a[i];  a[i]=a[k];  a[k]=temp;    k=i;  m=b[i];    for(int j=i+1; j<=n; j++)  {  if(b[j]>m)  {  k=j;  m=b[j];  }  }  temp=b[i];  b[i]=b[k];  b[k]=temp;  s=s+a[i]\*b[i];  }  AFISARE(a,b,n,s);  } | import java.util.Scanner;  public class MinProdusScalar{  static void AFISARE(int a[], int b[], int n,int s)  {  System.out.println("Suma="+s);  for(int i=1; i<=n; i++) System.out.print(a[i] + " ");    System.out.println();    for(int i=1; i<=n; i++) System.out.print(b[i] + " ");  }  **//AICI AR VENII FUNCȚIILE. din stanga**  public static void main(String[] args)  {  //Produs Scalar Maxim/Minim  int n,s=0,i,a[], b[];  Scanner cin=new Scanner(System.in);  System.out.print("Nr elemente=");  n=cin.nextInt();  a=new int [n+1];  b=new int [n+1];  System.out.print("Elementele din a[]: ");  for(i=1;i<=n;i++) a[i]=cin.nextInt();    System.out.print("Elementele din b[]: ");  for(i=1;i<=n;i++) b[i]=cin.nextInt();  MINIM1(a,b,n,s);  }}  //sorteaza a[] crescator, si b[] descrescator, apoi face prod scalar minim |
| --- | --- |



1. **Care dintre urmatoarele functii sunt asimptotic pozitive?**

**a)** **(cand n este un numar mare)**

* Numaratorul (n−7) este aproximativ n
* Numitorul 10n - n^2 este dominat de termenul −n^2
* Rezulta aproximativ ca fractia este -1/n, iar radacina devine radical din -1/n care nu este definit.
* Deci **NU** este asimptotic pozitiva.

#### **b)**

* Numaratorul (n^2-1) este pozitiv pentru n>1.
* Numitorul sinus oscilează între -1 și 1, semnul depinde de n.
* Deoare sin ia valori negative pentru unele valori ale lui n, functia va fii uneori negativa, deci **NU** este asimptotic pozitiva.

#### **c)​​**

* Numărătorul este echivalent cu radical din n^2, adica aproximativ n.
* Numitorul este pozitiv, deoarece, ce se află sub (ln) este > 0 cand n este numar mare.
* Numitorul si Numaratoru sunt pozitivi => **Funcția este Asimptotic pozitiva**

#### **d)**

* Factorialul devine negativ, cand n>100
* Deci funcția nu e definită pentru n>100 => **NU** este asimptotic pozitiva

**2. Demonstrati ca:**

**a)**

* Comparam n ln(n) si n^2.
* ln(n) creste mult mai incet decat n.
* Astfel, n ln(n) < n^2 (cand n are valori mari)

**b)**

* Definitia notatiei Omega (Ω) înseamnă ca exista constante c>0 și n0>0 astfel incat f(n) >= c\* g(n).
* Comparam n radical din n CU n ln(n):
* n sqrt(n) este aproximativ = cu n^1.5
* n ln(n) creste mai lent decat n^1.5
* Deci n sqrt(n) este mereu > n ln(n) (pentru valori suficient de mari)

**c)**

* n factorial = 1\*2\*3\*4\*....\*n
* e^n = e\*e\*e\*....\*e (de n ori)
* Pentru n>3, toti termeni n! > e^n deoarece factorial creste mai repede decat e^n.

**d)**

* n factorial = 1\*2\*3\*4\*....\*n
* n^n = n\*n\*n\*...\*n
* n! este clar mai mic decat n^n, pana și pentru n=2

**Avem o multime nr int. Determina o submultime a acestuia cu suma maxima!**

| import java.util.Scanner;  class Main {  public static Scanner cin = new Scanner(System.in);  public static void AFISARE(int sumaMax, int n, int[] numere, int[] aparitii){  System.out.println("Suma maxima este -> " + sumaMax);  System.out.println("Compusa din submultimea:");  System.out.print("{");  for(int i=1; i<=n; i++){  if(aparitii[i] == 1) System.out.print(numere[i] + " ");  }  System.out.println("}");  }  public static void SUBMULTIME\_SUMA\_MAXIMA(int n, int[] numere, int sumaMax, int[] aparitii){  sumaMax=0;  for(int i=1; i<=n; i++){  //Daca e nr negativ, sau 0, sarim peste el.  if(numere[i] <= 0) continue;  sumaMax += numere[i];  aparitii[i]=1;//numărul curent apare în submulțime  }  AFISARE(sumaMax, n, numere, aparitii);  }  public static void main(String[] args) {  int n, sumaMax=0; int[] numere, aparitii;  System.out.print("n = ");  n = cin.nextInt();  numere = new int[n+1];  aparitii = new int[n+1];  System.out.print("Dati " + n + " numere: ");  for (int i=1; i<=n; i++){  numere[i] = cin.nextInt();  }  SUBMULTIME\_SUMA\_MAXIMA(n, numere, sumaMax, aparitii);  }  }  /\* Exemplu de apelare:  n = 5  Dati 5 numere: 0 3 -2 5 -4  Rezultat:  Suma maxima este -> 8  Compusa din submultimea: {3 5 } \*/ |
| --- |

**L3**

**Memorarea optima a textelor pe benzi**

Explicație:

* Definirea clasei Text – Aceasta conține textul și lungimea sa calculată automat.
* Crearea unui array de obiecte Text – Se introduc mai multe texte de diferite lungimi.
* Sortarea array-ului – Se folosește Arrays.sort() cu un Comparator care sortează după lungime.
* Afișarea rezultatului – Se afișează textele în ordinea optimă. (de la lungime minima la lungime maxima)
* Această soluție minimizează timpul mediu de citire conform cerințelor problemei.

| import java.util.Arrays;  import java.util.Comparator;  class Text {  String content;  int length;  public Text(String content) {  this.content = content;  this.length = content.length();  }  }  //folosim Arrays.sort, ca un comparator de lungimi a textelor **o(n log n)**, unde n e nr elemente din tablou  public class MagneticTapeStorage {  public static void main(String[] args) {  Text[] texts = {  new Text("Text text text text 4"),  new Text("Text 1"),  new Text("Text text text 3"),  new Text("Text text 2"),  new Text("Text text text text text 5 "),  };  // Sortare după lungimea textelor (O(n log n))  Arrays.sort(texts, Comparator.comparingInt(t -> t.length));  // Afisare ordine optima de memorare  System.out.println("Ordinea optima de memorare pe banda:");  for (Text text : texts) {  System.out.println(text.content + " (" + text.length + " octeți)");  }  }  } |
| --- |

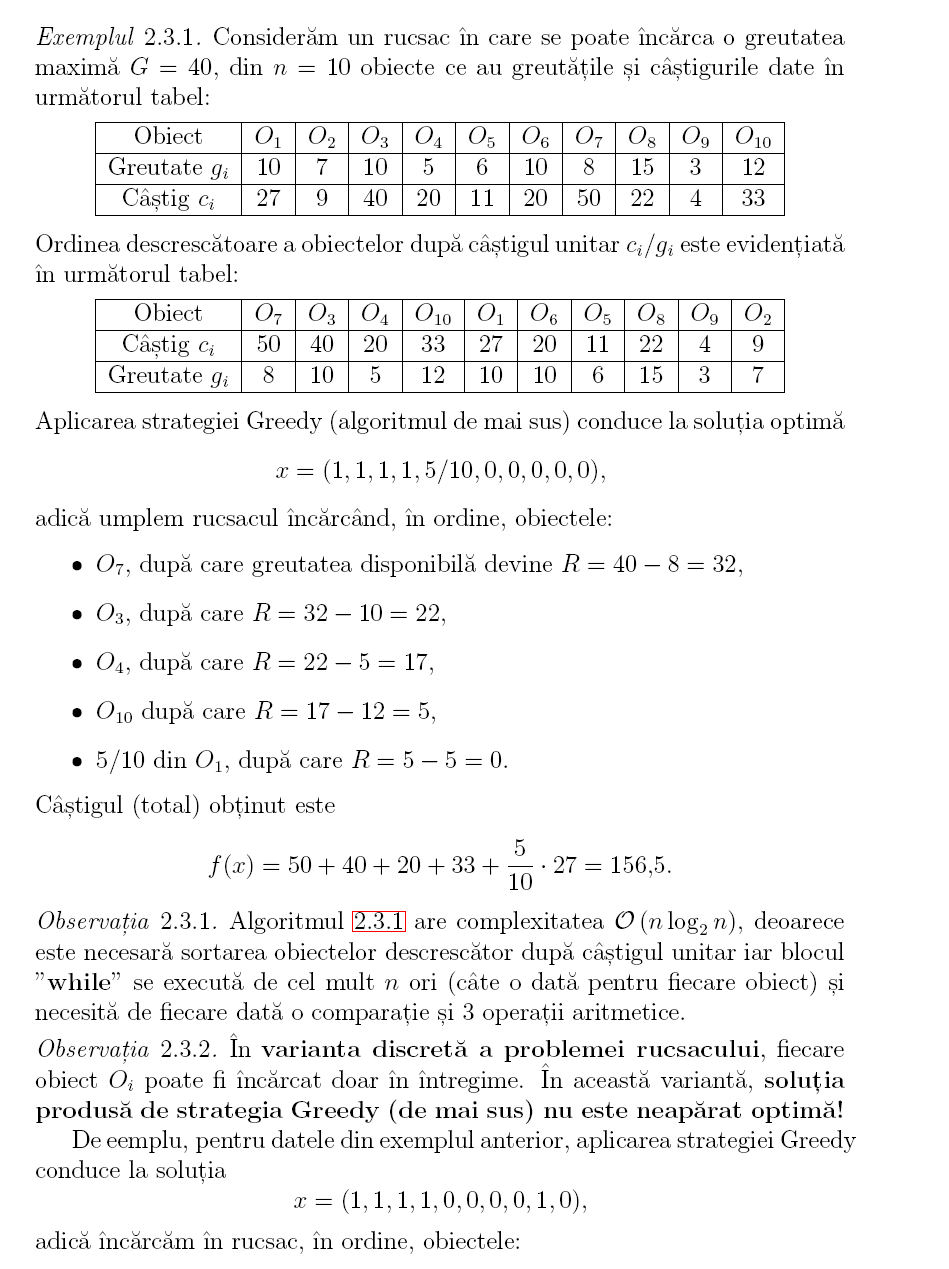
**//(La fiecare pas) La citirea unui text de pe banda, trebuie citite și textele aflate înaintea lui.**

//Acest program va lua textele si le va memora in ordinea lungimii lor, deoarece cele mai scurte texte au cea mai mica memorie. De aceea ordinea conteaza. (aceasta sortare face timpul mediu de citire să fie minim)

**Problema Rucsacului**

| import java.util.Scanner;  class L4Rucsac {      public static void SORTARE(double g[], double c[], int n) {  for (int i = 1; i <= n - 1; i++) {  for (int j = i + 1; j <= n; j++) {  if (c[i] / g[i] < c[j] / g[j]) {  double aux = g[i];  g[i] = g[j];  g[j] = aux;  aux = c[i];  c[i] = c[j];  c[j] = aux;  }  }  }  }    public static void CITIRE(double g[], double c[], int n)  {  Scanner cin = new Scanner(System.in);  System.out.println("Citire " + n +  " Greutati:");  for(int i=1; i<=n; i++)g[i]=cin.nextDouble();    System.out.println("Citire " + n +  " Costuri:");  for(int i=1; i<=n; i++)c[i]=cin.nextDouble();  }    public static void AFISARE(double g[], double c[], int n)  {  System.out.println("c[]");  for(int i=1; i<=n; i++)System.out.print(c[i] + " ");  System.out.println("");  System.out.println("g[]");  for(int i=1; i<=n; i++)System.out.print(g[i] + " ");  }    public static void RUCSAC(double G, double g[], double C, double c[],  int n, double x[])  {  C=0;  for(int i=1; i<=n; i++)x[i]=0;  double R=G;  int i=1;    while(R>0)  {  if(g[i]<=R)  {  x[i]=1;  C=C+c[i];  R=R-g[i]; | i++;  }  else{  x[i]=R/g[i];  C=C+x[i]\*c[i];  R=0;  }  }    System.out.println("Ordine optima:");  for(i=1; i<=n; i++) System.out.print(x[i] + " ");  }      public static void main(String[] args) {  double g[], c[], x[], C=0, G=40; int n;    Scanner cin = new Scanner(System.in);  System.out.print("n=");  n = cin.nextInt();    System.out.print("G:");  G = cin.nextDouble(); //40    g= new double[n+1];  c= new double[n+1];  x= new double[n+1];    CITIRE(g,c,n);  SORTARE(g,c,n);  //AFISARE(g, c, n);  RUCSAC(G,g,C,c,n,x);  }  }  **/\* REZULTAT:**  **n=10**  **G:40**  **Citire 10 Greutati:**  **10 7 10 5 6 10 8 15 3 12**  **Citire 10 Costuri:**  **27 9 40 20 11 20 50 22 4 33**  **Ordine optima:**  **1.0 1.0 1.0 1.0 0.5 0.0 0.0 0.0 0.0 0.0**  **\*/**  **Complexitate:**  Foloseste bubble sort care e O(n^2).  Citire() - o(n)  Functia RUCSAC: O(n)  În această funcție, se parcurge lista de obiecte și se adaugă fiecare obiect în rucsac (sau o fracțiune din obiect, în cazul în care nu încap întregi). |
| --- | --- |

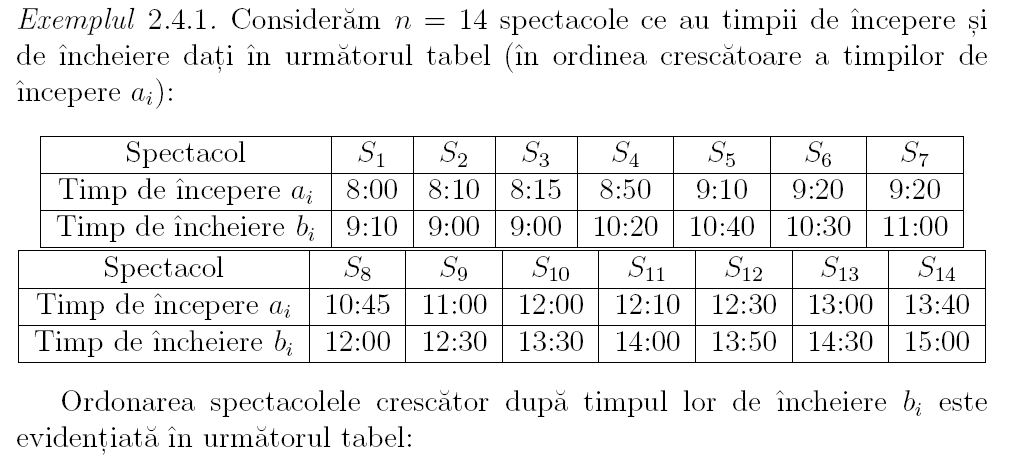
| **O7 = 50/8 = 6.25** | **O3 = 40/10 = 4** | **O4 = 20/5 = 4** | **O10 = 33/12 = 2.75** | **O1 = 27/10 = 2.7** |
| --- | --- | --- | --- | --- |
| **O6 = 20/10 = 2** | **O5 = 11/6 = 1.83** | **O8 = 22/15 = 1.46** | **O9 = 4/3 = 1.33** | **O2 = 1.28** |

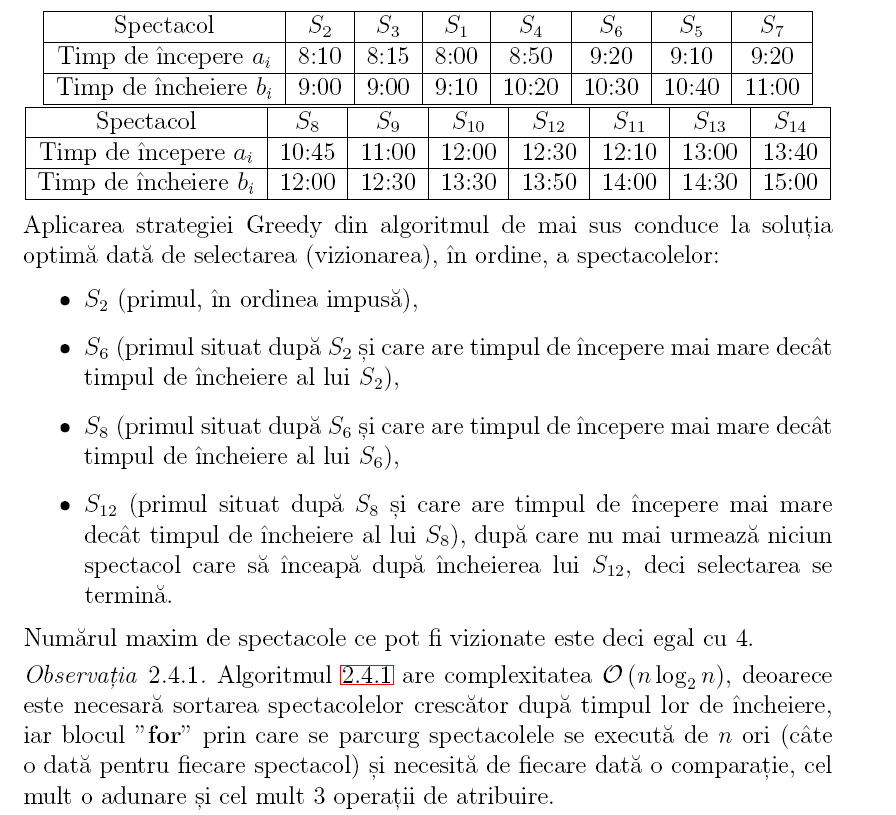
****

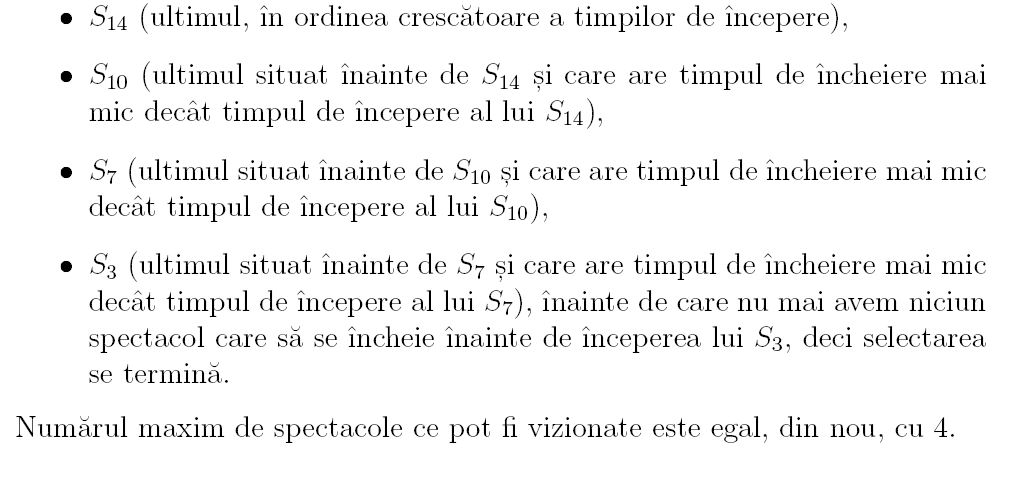
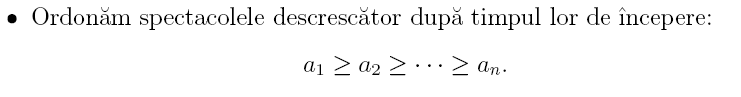
**Nr. Spectacole**

| import java.util.Scanner;  class L4Spectacole {      public static void SORTARE(double a[], double b[], int n) {  for (int i = 1; i <= n - 1; i++) {  for (int j = i + 1; j <= n; j++) {  if (b[i] > b[j]) {  double aux = a[i];  a[i] = a[j];  a[j] = aux;  aux = b[i];  b[i] = b[j];  b[j] = aux;  }  }  }  }    public static void CITIRE(double a[], double b[], int n)  {  Scanner cin = new Scanner(System.in);  System.out.println(n + " Timpuri de Incepere:");  for(int i=1; i<=n; i++)a[i]=cin.nextDouble();    System.out.println(n + " Timpuri de Incheiere:");  for(int i=1; i<=n; i++)b[i]=cin.nextDouble();  cin.close();  }    public static void AFISARE(double a[], double b[], int n)  {  System.out.println("Lista Sortata: Timpuri Incepere:");  for(int i=1; i<=n; i++)System.out.print(a[i] + " ");  System.out.println("");  System.out.println("Lista Sortata: Timpuri Incheiere:");  for(int i=1; i<=n; i++)System.out.print(b[i] + " ");  }    //Sortarea cu Bubble Sort (O(n^2)  //CITIRE o(n)  //SPECTACOLE o(n) deoarece doar se parcurge o lista | public static void SPECTACOLE(double a[], double b[], double c[], int n, int m)  {  m=0;  for(int i=1; i<=n; i++) c[i]=0;  double t=a[1]-1;    for(int i=1; i<=n; i++){  if(a[i]>t)  {  c[i]=1;  m++;  t=b[i];  }  }    System.out.println("Nr. spectacole vizionate: " + m);  System.out.println("Lista vizionarii:");  for(int i=1; i<=n; i++) if(c[i]==1) System.out.println("Incepe la: " + a[i] + " si se termina la: " + b[i]);  }      public static void main(String[] args) {  double a[], b[], c[]; int n, m=0;    Scanner cin = new Scanner(System.in);  System.out.println("");  System.out.print(" Nr spectacole=");  n = cin.nextInt();    a= new double[n+1];  b= new double[n+1];  c= new double[n+1];      CITIRE(a,b,n);  SORTARE(a,b,n);  //AFISARE(a, b, n);  SPECTACOLE(a,b,c,n,m);  cin.close();  }  }  /\*  N=14  Timp Incepere:  8 8.1 8.15 8.50 9.10 9.20 9.20 10.45 11 12 12.1 12.3 13 13.4  Timp Incheiere:  9.1 9 9 10.2 10.4 10.3 11 12 12.3 13.3 14 13.5 14.3 15  \*/ |
| --- | --- |

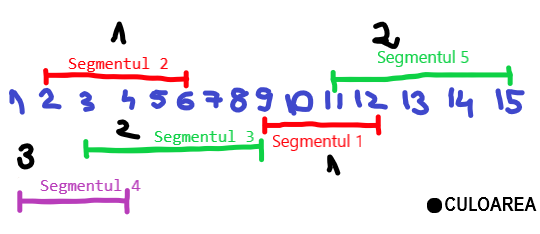
* **Ordonam spectacolele crescător după timpul lor de iıncheiere: b1 <= b2 <=... <= bn**
* Selectam primul. Spectacolul curent S(i) selectează dacă **timp începere>timp încheiere al spectacolului selectat.**

****

****

****

Nr minim de culori necesare pentru a colora “n” segmente dar 2 segmente care se intersectează sa fie colorate diferit.



Segmentul 1, 2 - Culoarea 1

Segmentul 3 - Culoarea 2

Segmentul 4,5 - Culoarea 3

**COLORAREA STEAGULUI TRICOLOR**

| import java.util.Scanner;  class Main {  static Scanner cin = new Scanner(System.in);  public static void COLORARE\_STEAG\_TRICOLOR(int n, String[] c){  //"i" reprezinta culoarea pusa pe prima pozitie.  //"j" reprezinta culoare pusa pe a 3-a pozitie.  //Dar toate culorile steagului trebuie sa fie diferite.  for(int i=1; i<=n; i++){  for(int j=1; j<=n; j++) {  //n-1 este penultima culoare (punem la mijloc)  if(i!=j && i!=n-1 && j!=n-1) System.out.println(c[i] + " " + c[n-1] + " " + c[j] );  //"n" este ultima culoare (punem la mijloc) | if(i!=j && i!=n && j!=n)System.out.println(c[i] + " " + c[n] + " " + c[j] );  //Fac cu 2 if-uri ca sa putem afisa toate solutiile.  }  }  }  public static void main(String[] args) {  int n=4; //n- nr culori  String[] c = new String[n+1];//culorile  c[1] = "rosu";  c[2] = "albastru";  c[3] = "galben";  c[4] = "alb";  COLORARE\_STEAG\_TRICOLOR(n,c);  }  } |
| --- | --- |

**Colorarea grafurilor**

| import java.util.Scanner;  class Main {  static Scanner cin = new Scanner(System.in);  static int i,muchii=10,n=6, m=5, a[][];  public static void CreareMatriceAdiacenta(int a[][], int muchie)  {  int x=0,y=0;  //System.out.println("Dati muchia " + i + ": " + "Intre nodurile: ");  System.out.println("Dati muchiile: ");  for(int i=1; i<=muchie; i++)  {  x = cin.nextInt();  y = cin.nextInt();  a[x][y]++;  a[y][x]++;  }  }  public static void AfisareMatrice(int a[][], int n)  {  System.out.println("Matrice Adiacenta:");  for(int i=1; i<=n; i++)  {  for(int j=1; j<=n; j++) System.out.print(a[i][j] + " ");  System.out.println();  }  System.out.println("- - - - - -");  }  public static void AFISARE(int vector[], int n)  {  for(int i=1; i<=n; i++) System.out.print(vector[i] + " ");  System.out.println();  } | public static void COLORARE(int a[][], int n, int m){  int k=1, x[];  x = new int[n+1];  x[1]=0;  while(k>0){  if(x[k]<m){  x[k]= x[k]+1;  if(VALID(x,k)==1)  {  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=0;  }  }  }  else{  k--;  }  }  }  public static int VALID(int x[], int k){  for(int i=1; i<=k-1; i++)  {  if(a[i][k]>=1 && x[i]==x[k])return 0;  }  return 1;  }  public static void main(String[] args) {  a = new int[n+1][n+1];  //System.out.println("Nr noduri=");  //cin.nextInt(n);  //System.out.println("Nr muschii=");  //cin.nextInt(muchii);  CreareMatriceAdiacenta(a, muchii);  AfisareMatrice(a, n);  //Asumam ca avem m=5 culori, n=6 noduri, m=10 muchii  //1 2 1 3 1 5 1 6 2 3 2 5 3 4 3 6 4 5 5 6  COLORARE(a,n,m);  }  } |
| --- | --- |

x = (x1, x2, . . . , xn), (asa arata fiecare solutia, unde xi reprezinta culoarea)

**Dame**

| import java.util.Scanner;  class HelloWorld {  public static Scanner cin = new Scanner(System.in);        public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++){  System.out.print(x[i] + " ");  }  System.out.println();  }    public static void DAME(int n){  int k=1;  int[] x;  x = new int[n+1];  x[1]=0;    while(k>0){  if(x[k]<n){  x[k]++;  if(VALID(x,k)==1){  if(k==n){  AFISARE(x,n);  k--;  }  else{  k++;  x[k]=0;  }  }  }  else k--;  }    }    public static int VALID(int[] x, int k){  for(int i=1; i<=k-1; i++){  int pozitiv = Math.abs(x[k]-x[i]);  if(x[k]==x[i] || k-i==pozitiv) return 0;  }  return 1;  }  public static void main(String[] args) {  int n;  System.out.print("n=");  n = cin.nextInt();  DAME(n);  }  } | * x = (1, 3, 5, 2, 4) * **(stochează index coloana pt fiecare linie)** * Toate modalitățile de aranjare a n dame astfel încât oricare 2 dame să nu se atace reciproc (pe linii, coloane sau diagonale). * NU pe aceeași linie * NU aceeași coloana (i!=j) * NU pe aceeași diagonală |i-j| != xi-xj * Pentru n = 2, problema nu are soluție, deoarece două dame așezate pe linii și coloane diferite se află pe aceeași diagonală, deci se atacă reciproc. * Nici pentru n = 3, problema nu are soluție, deoarece orice două dame așezate pe aceeași culoare se atacă reciproc.   **Complexitatea funcției VALID**:   * Funcția VALID verifică dacă o damă plasată în poziția curentă (x[k]) nu este pe aceeași linie, coloană sau diagonală cu alte dame deja plasate. Acest lucru presupune parcurgerea listei damei plasate deja, adică în cel mai rău caz, pentru fiecare poziție k, trebuie să parcurgă k-1 elemente. * Astfel, complexitatea funcției VALID este O(k), unde k este poziția curentă a damei pe tablă.   **Complexitatea totală**:   * Fiecare dintre cele N dame poate fi plasată într-o poziție din 1 până la N, ceea ce sugerează un număr maxim de N! (factorial de N) posibile aranjamente. * În fiecare caz, se verifică validitatea poziției cu o complexitate de O(k), unde k este indexul damei curente. * Astfel, complexitatea totală este **O(N! \* N)**, deoarece:   + Pt fiecare aranjament de dame (N!), trebuie să validăm fiecare poziție cu o complexitate de O(k). * Complexitatea algoritmului este **O(N! \* N)**, ceea ce este o complexitate factorială, ceea ce face algoritmul ineficient pentru valori mari ale lui N (de exemplu, N > 10) |
| --- | --- |

**Nebuni pe Tabla Sah**

| **//PE NEGRU**  import java.util.Scanner;  class HelloWorld {  public static Scanner cin = new Scanner(System.in);  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n-1; i++){  System.out.print(x[i] + " ");  }  System.out.println();  }    public static void NEBUNI1(int n){  int k=1, ult=2;  int[] x;  x = new int[n+1];  x[1]=0;    while(k>0){  if(x[k]<ult){  x[k]++;  if(VALID(x,k)==1){  if(k==n-1)AFISARE(x,n);  else{  k++;  if(k<=n/2)  {  x[k]=0;  ult=2\*k;  }  else{  x[k]=2\*k-n;  ult=n;  }  }  }  }  else{  k--;  if(k<=n/2)ult=2\*k;  else ult=n;  }  }    }    public static int VALID(int[] x, int k){  for(int i=1; i<=k-1; i++){  if(x[k]-x[i]==k-i) return 0;  }  return 1;  }  public static void main(String[] args) {  int n;  System.out.print("n=");  n = cin.nextInt();  NEBUNI1(n);  }  } | **//PE ALB**  class HelloWorld {  public static Scanner cin = new Scanner(System.in);  //afisare functie  public static void NEBUNI2(int n){  int k=2, ult=3;  int[] y;  y = new int[n+1];  y[2]=0;    while(k>1){  if(y[k]<ult){  y[k]++;  if(VALID(y,k)==1){  if(k==n-1)  {  y[1]=1;  AFISARE(y,n);  y[1]=n;  AFISARE(y,n);  }  else{  k++;  if(k<=(n+1)/2)  {  y[k]=0;  ult=2\*k-1;  }  else{  y[k]=2\*k-n-1;  ult=n;  }  }  }  }  else{  k--;  if(k<=n/2)ult=2\*k;  else ult=n;  }  }  }    public static int VALID(int[] y, int k){  if(y[k]==k)return 0;    for(int i=2; i<=k-1; i++){  if(y[k]-y[i]==k-i) return 0;  }  return 1;  }  public static void main(String[] args) {  int n; System.out.print("n=");  n = cin.nextInt();  NEBUNI2(n);  }  } |
| --- | --- |

| * x = (1, 4, 2, 5) | * Pe alb, primul nebun este pus ori pe poziția 1 ori pe poziția 5. * **Nu se pot afla pe aceeași diagonală.** * xₖ reprezintă linia pe care se află nebunul de pe diagonala Dₖ, pentru k = 1, n − 1. * **IMPORTANT:** Numărul maxim de nebuni ce pot fi așezați pe o tablă de șah de dimensiune n×n, n ≥ 2, astfel încât oricare doi nebuni să nu se atace reciproc este = **n − 1**. * Negru: Direcția stanga-sus -> dreapta-jos |
| --- | --- |

| **CALUL pe Tabla De Șah** (merge ca un **“L”** adică 2 poziții într-o direcție, și apoi 1 poziție în alta directie)  // Verificăm dacă noua poziție este validă (dacă nu am ieșit din tabla si mai poate fii vizitata)  //Spre exemplu x = x + dx[i]  //Si y = y + dy[i].  //Pentru i=1, calul încearca sa mearga 2 poziții la dreapta, si 1 pozitie in sus. (daca poate)  Stocam Poziția curentă în vectorul de pași  // Dacă nu am găsit mișcări valide), afisam solutia partiala.  // Dar doar daca tabla este și goala. (altfel înseamnă ca ne-am blocat.) |
| --- |

### **Algorithm prin backtracking**

1. Se încearcă să atribuim o culoare primului nod.
2. Se verifică dacă alegerea este validă (adică nu intră în conflict cu culorile nodurilor adiacente).
3. Dacă este validă, se trece la următorul nod și se repetă procesul.
4. Dacă un nod nu poate primi nicio culoare validă, se face **backtracking**: se revine la nodul anterior și se încearcă o altă culoare.
5. Se repetă procesul până când:
   * Toate nodurile au fost colorate (soluție validă).
   * S-au epuizat toate opțiunile (nu există soluție cu mm culori).

## **Problema colorarii hartilor**

Se dă o hartă formată din mai multe țări și un număr de culori disponibile. Se cere să se coloreze fiecare țară cu una dintre aceste culori, astfel încât **două țări vecine (care au frontieră comună) să nu aibă aceeași culoare**.

### **Modelarea problemei**

Putem reprezenta această problemă sub forma unui **graf neorientat**, unde:

* **Fiecare țară este un nod al grafului**.
* **Există o muchie între două noduri dacă și numai dacă țările respective au frontieră comună**.

Astfel, problema colorarii hartilor devine o **problemă de colorare a unui graf**, unde trebuie să atribuim fiecărui nod o culoare astfel încât **nodurile conectate prin muchii să aibă culori diferite**.

|  |  |
| --- | --- |

**2. Metoda Greedy**

* Metoda Greedy nu caută să determine toate soluțiile posibile și apoi să aleagă pe cea optimă conform criteriului de optimizare dat (ceea ce ar necesita în general un timp de calcul și spațiu de memorie mari), ci constă în a alege pe rând câte un element, urmând să-l "înghită" eventual în soluția optimă. De aici vine și numele metodei (Greedy = lacom).
* Astfel, dacă trebuie determinat maximul unei funcții de cost depinzând de a1, ..., an, ideea generală a metodei este de a alege la fiecare pas acel element care face să crească cât mai mult valoarea acestei funcții. Din acest motiv metoda se mai numește și a **optimului local.**
* **Optimul global** se obține prin alegeri succesive, la fiecare pas, ale optimului local, ceea ce permite rezolvarea problemelor fără revenire la deciziile anterioare (așa cum se întâmplă la metoda backtracking).
* În general metoda Greedy oferă o soluție posibilă și nu întotdeauna soluția optimă. De aceea, dacă problema cere soluția optimă, algoritmul trebuie să fie însoțit și de justificarea faptului că soluția generată este optimă.

### 

### **🔹 Complexitate**

1. **Timp de execuție și complexitate**
   * Se măsoară în funcție de **numărul de operații** efectuate în raport cu dimensiunea intrării (n).
   * Se exprimă în **notația Big O** care oferă o estimare a performanței în cel mai rău caz.
2. **Notația Big-O**  
   Reprezintă o limită superioară pentru creșterea numărului de operații în raport cu n.  
   Exemple:
   * **O(1)** – Timp constant (ex: accesarea unui element într-un array)
   * **O(log n)** – Logarithmic (ex: căutare binară)
   * **O(n)** – Liniar (ex: parcurgerea unui array)
   * **O(n log n)** – Quasilinear (ex: sortarea prin Merge Sort)
   * **O(n²), O(2ⁿ), O(n!)** – Complexități mai mari (ex: algoritmi de forță brută)
3. **Cel mai bun, mediu și cel mai rău caz**
   * **Best-case (Ω)** – Cel mai favorabil scenariu
   * **Average-case (Θ)** – Cazul mediu
   * **Worst-case (O)** – Cel mai rău caz (folosit de obicei pentru analiză)

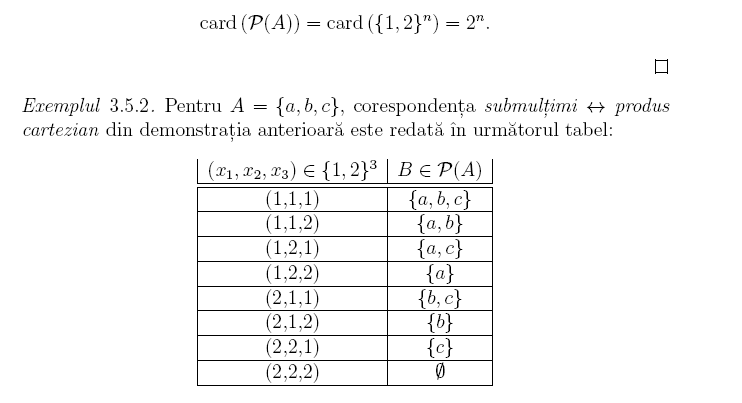
**Pattern-uri standard**

* Sortare Bubble Sort ⟶ O(n²)
* Căutare binară ⟶ O(log n)
* DFS/BFS într-un graf ⟶ O(V + E)

| **PRODUS CARTEZIAN**  import java.io.Reader;  import java.io.StringReader;  import java.util.Scanner;  class Main {  static Scanner cin = new Scanner(System.in);  static public int[] m;  static public char[][] a;  public static void PRODUS\_CARTEZIAN(int n, int[] m){  int k=1;  int[] x;  x= new int[n+1]; //tine combinatiile curente  x[1]=0; //intial primul element este 0  while(k>0)  {  if(x[k]<m[k]){ //element < lungimea sirului  x[k]++;  if(k==n)AFISARE(x,n); //ajuns la ult combinatie  else{  k++;  x[k]=0; //initializeaza urm element  }  }  else{  k--;  }  }  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(a[i][x[i]]);  System.out.println();  }  public static void SETEAZA\_LUNGIMI\_SIRURI(int n){  m = new int[n+1]; //Citeste lungimea fiecarui sir  a = new char[n+1][]; //Creaza vector char size “m”  System.out.print("Citeste toate lungimile de sir 'm': ");  for(int i=1; i<=n; i++){  m[i] = cin.nextInt();  a[i] = new char[m[i]+1];  }  //Goleste caracterul \n  cin.nextLine();  }  public static void CITESTE\_SIRURI(int n){  for(int i=1; i<=n; i++){  System.out.print("Citeste Sir" + i + " de lungime " + m[i] + ": ");  String line;  do{  line = cin.nextLine(); //citeste linia  if(line.length() < m[i])System.out.println("Linie prea scurta! Încearcă din nou."); //asigura size corect  }  while(line.length() < m[i]);  line = line.substring(0, m[i]); //Taie toate caracterele peste lungimea maximă.  for(int j=0; j<m[i]; j++){  a[i][j+1] = line.charAt(j); //Stochează la poziția corecta.  // "j" începe de la 0 deoarece String începe tot de la 0.  }  }  }  public static void main(String[] args) {  int n;  System.out.print("n=");  n=cin.nextInt();  SETEAZA\_LUNGIMI\_SIRURI(n);  CITESTE\_SIRURI(n);  PRODUS\_CARTEZIAN(n,m);  }  } | Exemplu:  n=3 //nr siruri  Citește toate lungimile de șir 'm': 221  Citeste Sir1 de lungime 2: ab  Citeste Sir2 de lungime 2: +-  Citeste Sir3 de lungime 1: с  Rezultat:  a+c  a-c  b+c  b-c  **COMPLEXITATEA**  Fiecare sir i are lungimea m[i].  Total combinatii = m[1] X m[2] X … X m[n]  Complexitatea spatiala a[][] este O(n\*m[i])   * Algoritmul este eficient în cazul unor dimensiuni mici ale șirurilor, dar devine ineficient pe măsură ce numărul de șiruri și lungimea acestora cresc exponențial. * Complexitatea exponențială în ceea ce privește numărul de combinații face acest algoritm nefezabil pentru seturi mari de date. |
| --- | --- |

**SUBMULTIMI**

| import java.util.Scanner;  class Main {  public static void SUBMULTIMI\_STANDARD(int n){  int k=1;  int[] x;  x= new int[n+1]; //stocare elemente din multime  x[1]=0; //initial primul element nu e inclus  while(k>0)  {  if(x[k]<2){  x[k]++;  if(k==n)AFISARE(x,n);//dacă am ajuns la ult  else{  k++;//mergi la urm element  x[k]=0;//initializeaza urm element  }  }  else k--;  }  }  public static void AFISARE(int[] x, int n){  //if(x[i]==1) verifica daca element inclus  //in submultimea curenta  for(int i=1; i<=n; i++)if(x[i]==1) System.out.print(i + " ");  System.out.println();  }  public static void main(String[] args) {  int n;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  SUBMULTIMI\_STANDARD(n);  }  }    Deci A are 2^3 = 8 submultimi | import java.util.Scanner;  class Main {  static public int[] a;  public static void SUBMULTIMI\_OARECARE(int[] a, int n){  int k=1;  int[] x;  x= new int[n+1];  x[1]=0;  while(k>0)  {  if(x[k]<2){  x[k]++;  if(k==n)AFISARE(x,a,n);  else{  k++;  x[k]=0;  }  }  else k--;  }  }  public static void AFISARE(int[] x, int[] a, int n){  for(int i=1; i<=n; i++)if(x[i]==1) System.out.print(a[i] + " ");  System.out.println();  }  public static void main(String[] args) {  int n;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  a = new int[n+1];  System.out.print("Citeste toate nr oarecare: ");  for(int i=1; i<=n; i++){  a[i] = cin.nextInt();  }  SUBMULTIMI\_OARECARE(a,n);  }  }  /\*Nr submultimi pt multimea cu “n” elemente este 2^n, deoarece fiecare element poate fi inclus sau exclus independent de celelalte.  =>Complexitatea temporala este O(2^n)  Algoritmul generează toate submulțimile posibile și le afișează pe fiecare. Având în vedere că numărul total de submulțimi crește exponențial cu n, algoritmul are o eficiență scăzută pentru valori mari ale lui n. |
| --- | --- |



**Aranjamente (cu repetitie)**

Nr cuvinte de lungime n peste un alfabet cu m litere = m

Pentru m = 2 și n = 3, aranjamente cu repetiție sunt, în ordine lexicografică:

(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2).

| import java.util.Scanner;  class Main {  public static void ARANJAMENTE\_CU\_REPETITII\_STANDARD(int m, int n){  int k=1;  int[] x;  x= new int[n+1];  x[1]=0;  while (k > 0) {  if (x[k] < m) {  x[k]++;  if (k == n) AFISARE(x, n);  else {  k++;  x[k] = 0; // Reset  }  } else {  k--; // Backtrack  }  }  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  System.out.print("m=");  m=cin.nextInt();  ARANJAMENTE\_CU\_REPETITII\_STANDARD(m,n);}}  }  }  //Algoritmul generează **aranjamente cu repetiții**, ceea ce înseamnă că fiecare dintre cele n poziții poate fi ocupată de oricare dintre cele m elemente disponibile. | import java.util.Scanner;  class Main {  static int[] a;  public static void ARANJAMENTE\_CU\_REPETITII\_OARECARE(int m, int n){  int k=1;  int[] x;  x= new int[n+1];  x[1]=0;  while (k > 0) {  if (x[k] <= m) {  x[k]++;  if (k == n) AFISARE(x, n);  else {  k++;  x[k] = 0; // Reset  }  } else {  k--; // Backtrack  }  }  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(a[x[i]] + " ");  System.out.println();  }  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  a = new int[n+1];  System.out.print("m=");  m=cin.nextInt();  System.out.print("Numere oarecare=");  for(int i=1; i<=n; i++){  a[i] = cin.nextInt();  }  ARANJAMENTE\_CU\_REPETITII\_OARECARE(m,n);  }  } |
| --- | --- |

**Aranjamente**

Nr cuvinte de lungime n **(cu litere distincte)** pt un alfabet cu m litere este [m]n

Pt m=4, n=2, aranjamentele sunt in ordine lexicografica:

(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (3, 4), (4, 1), (4, 2), (4, 3).

Deci avem [4]2 = 4 · 3 = 12 aranjamente.

Formula aranjamente [m]n =

| import java.util.Scanner; //**STANDARD**  class Main {  **//aranjamente de m luate cate n**  public static void ARANJAMENTE\_STANDARD(int m, int n){  int k=1;  int[] x;  x= new int[m+1];  x[1]=0;  while(k>0)  {  if(x[k]<m){ //daca val curenta < m  x[k]++;  if(VALID(x,k))  {  if(k==n)AFISARE(x,n);  else{ //daca nu a ajuns la aranj complet  k++;  x[k]=0;  }  }  } //daca valoarea curenta depaseste “m”  else k--;  }  }  public static boolean VALID(int[] x, int k){  for(int i=1;i<=k-1;i++){  if(x[k]==x[i])return false; //daca exista elemente  }//duplicate, NU este valid  return true;  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  System.out.print("m=");  m=cin.nextInt();  ARANJAMENTE\_STANDARD(m,n);  }  } | import java.util.Scanner; //**OARECARE**  class Main {  static int[] a;  public static void ARANJAMENTE1(int m, int n){  int k=1;  int[] x;  x= new int[m+1];  x[1]=0;  while(k>0)  {  if(x[k]<m){  x[k]++;  if(VALID(x,k))  {  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=0;  }  }  }  else k--;  }  }  public static boolean VALID(int[] x, int k){  for(int i=1;i<=k-1;i++){  if(x[k]==x[i])return false;  }  return true;  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(a[x[i]] + " ");  System.out.println();  }  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  System.out.print("m=");  m=cin.nextInt();  a = new int[m+1];  System.out.print("Numere oarecare=");  for(int i=1; i<=m; i++)a[i]=cin.nextInt();  ARANJAMENTE1(m,n);  }  } |
| --- | --- |

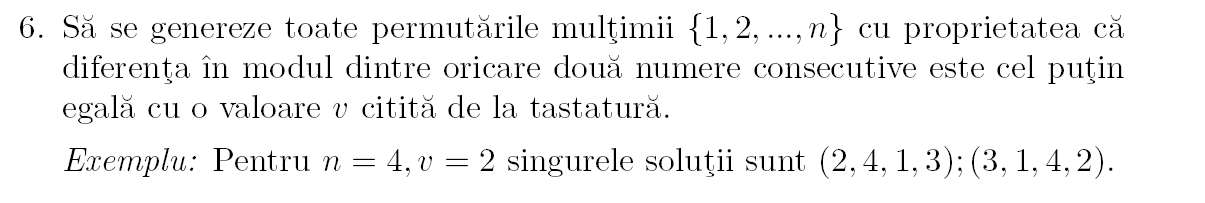
**Permutari**

Nr cuvinte de lungime ce conțin exact o data fiecare litera dintr-un alfabet cu “n” litere = n! (0! = 1)

n=3 => (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)

| import java.util.Scanner; //**STANDARD**  class Main {    public static void PERMUTARI1(int n){  int k=1;  int x[];  x= new int[n+1];  x[1]=0;    while(k>0)  {  if(x[k]<n){  x[k]++;  if(VALID(x,k)==1){  if(k==n){  AFISARE(x,n);  k--;  }  else{  k++;  x[k]=0;  }  }  }  else k--;  }  }    public static int VALID(int[] x, int k){ **//O(m)**  for(int i=1; i<=k-1; i++)if(x[k]==x[i]) return 0;  return 1;  }    public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }    public static void main(String[] args) {  int n;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();    PERMUTARI1(n);  }  }//Nr total aranjamente este:    Algo este eficient pentru valori mici “m și n”, dar devine rapid ineficient cand cresc, deoarece nr aranj crește **exponențial.** | import java.util.Scanner; //**OARECARE**  class Main {  static public int y[];    public static void PERMUTARI1(int n){  int k=1;  int x[];  x= new int[n+1];  x[1]=0;    while(k>0)  {  if(x[k]<n){  x[k]++;  if(VALID(x,k)==1){  if(k==n){  AFISARE(x,n);  k--;  }  else{  k++;  x[k]=0;  }  }  }  else k--;  }  }    public static int VALID(int[] x, int k){  for(int i=1; i<=k-1; i++)if(x[k]==x[i]) return 0;  return 1;  }    public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(y[x[i]] + " ");  System.out.println();  }    public static void main(String[] args) {  int n;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();    y = new int[n+1];  System.out.print("Citeste toate nr oarecare: ");  for(int i=1; i<=n; i++){  y[i] = cin.nextInt();  }    PERMUTARI1(n);  }  }//Complexitatea algo: **O(n \* A(m,n)**) |
| --- | --- |

**Formula combinari [m]n =**

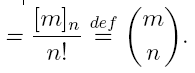


| import java.util.Scanner; //Cel mai rău caz, generează **toate perm posibile n elemente**, cu **complexitate de**: O(n!)  class Main { //In plus verificare VALID O(n) înseamnă complexitatea finală O(n! \* n)  static public int[] y; //valori citite de user  static public int v; **//prag minim dintre valori consecutive ale permutarii**  public static void PERMUTARI(int n){  int k=1;  int[] x; //ordinea elemente in permutare  x= new int[n+1];  x[1]=0;  while(k>0)  {  if(x[k]<n){ //daca val curenta < n  x[k]++;  if(VALID(x,k)==1){ //daca permutarea partiala este valida  if(k==n){ //daca am completat o permutare valida  AFISARE(x,n);  k--; //revenim pas, cautam alta solutie  }  else{  k++; //pozitia urm  x[k]=0; // Inițializăm cu 0 pentru a încerca noi valori  }  }  }  else k--; // Dacă nu mai sunt opțiuni pentru poziția curentă, revenim la pasul anterior  }  }  public static int VALID(int[] x, int k){  for(int i=1; i<=k-1; i++) if(x[k]==x[i]) return 0;  if(Math.abs(y[x[k]] - y[x[k-1]]) < v) return 0;  return 1;  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(y[x[i]] + " ");  System.out.println();  }  public static void main(String[] args) {  int n;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  y = new int[n+1];  System.out.print("Citeste toate nr oarecare: ");  for(int i=1; i<=n; i++){  y[i] = cin.nextInt();  }  System.out.print("v="); v=cin.nextInt();  PERMUTARI(n);}} |
| --- |

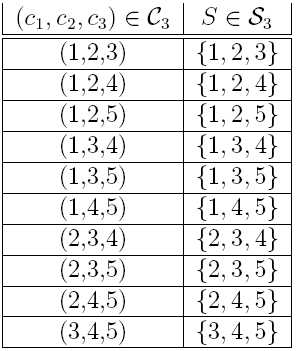
**Combinari**

Nr cuvinte STRICT crescătoare de lungime “n” cu alfabet “m” litere =

= nr submultimi cu “n” nr ale unei mulțimi cu “m” nr



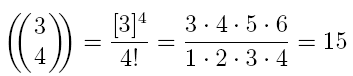
m=5 n=3



| class Main {  public static Scanner cin;    public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }    public static void COMBINARI(int m, int n){  int k=1;  int[] x = new int[n+1];  x[1]=0;    while(k>0){ //pana nu mai sunt elemente de procesat  if(x[k]<m-n+k){ //daca el. curent poate fii marit  x[k]++;  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=x[k-1];  }  }  else{  k--;  }  }  }    public static void main(String[] args) {  int m,n;  cin = new Scanner(System.in);    System.out.print("n=");  n = cin.nextInt();  System.out.print("m=");  m = cin.nextInt();  COMBINARI(m,n); }} |  |
| --- | --- |

**Combinari (cu repetitie)**

Nr cuvinte STRICT crescătoare de lungime “n” cu alfabet **(ordonat)** “m” litere =

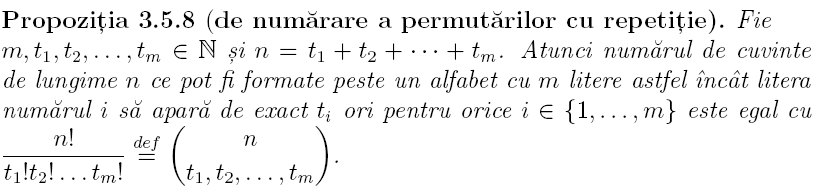
(1, 1, 1, 1), (1, 1, 1, 2), (1, 1, 1, 3), (1, 1, 2, 2), (1, 1, 2, 3),

(1, 1, 3, 3), (1, 2, 2, 2), (1, 2, 2, 3), (1, 2, 3, 3), (1, 3, 3, 3),

(2, 2, 2, 2), (2, 2, 2, 3), (2, 2, 3, 3), (2, 3, 3, 3), (3, 3, 3, 3).

| import java.util.Scanner;  class Main {  public static Scanner cin;  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }  public static void COMBINARI\_REPETITIE(int m, int n){  int k=1;  int[] x = new int[n+1];  x[1]=0;  while(k>0){  if(x[k]<m){  x[k]++;  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=x[k-1]-1;  }  }  else{  k--;  }  }  }  public static void main(String[] args) {  int m,n;  cin = new Scanner(System.in);  System.out.print("n=");  n = cin.nextInt();  System.out.print("m=");  m = cin.nextInt();  COMBINARI\_REPETITIE(m,n);  }  } | import java.util.Scanner;  class Main {  public static Scanner cin = new Scanner(System.in);  public static int[] a;  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(a[x[i]] + " ");  System.out.println();  }  public static void CITIRE(int[] a, int n){  for(int i=1; i<=n; i++)a[i] = cin.nextInt();  }  public static void COMBINARI\_REPETITIE\_OARECARE(int m, int n){  int k=1;  int[] x = new int[n+1];  x[1]=0;  while(k>0){  if(x[k]<m){  x[k]++;  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=x[k-1]-1;  }  }  else{  k--;  }  }  }  public static void main(String[] args) {  int m,n;  System.out.print("n=");  n = cin.nextInt();  System.out.print("m=");  m = cin.nextInt();  a = new int[m+1];  CITIRE(a, m);  COMBINARI\_REPETITIE\_OARECARE(m,n);  }  } |
| --- | --- |

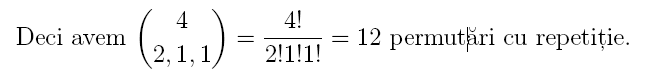
**Permutari (cu repetitie)**



Pentru m = 3, t1 = 2, t2 = t3 = 1, deci n = 4, permutarile cu repetitie sunt, in ordine lexicografica:

(1, 1, 2, 3), (1, 1, 3, 2), (1, 2, 1, 3), (1, 2, 3, 1), (1, 3, 1, 2), (1, 3, 2, 1),

(2, 1, 1, 3), (2, 1, 3, 1), (2, 3, 1, 1), (3, 1, 1, 2), (3, 1, 2, 1), (3, 2, 1, 1).

(deci litera 1 apare de 2 ori, litera 2 si 3 apare 1 data)

| import java.util.Scanner; **//DOAR STANDARD**  class Main {  public static Scanner cin;  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }  public static void CITESTE(int[] t, int n){  for(int i=1; i<=n; i++)t[i] = cin.nextInt();  }  public static void PERMUTARI\_CU\_REPETITIE\_STANDARD(int n, int[] t, int m){  n=0;  for(int i=1; i<=m; i++)n=n+t[i];  int k=1;  int[] x = new int[n+1];  x[1]=0;  while(k>0){  if(x[k]<m){  x[k]++;  int p = t[x[k]];  if(VALID(p,x,k)){  if(k==n){  AFISARE(x,n);  k--;  }  else{  k++;  x[k]=0;  }  }  }  else{  k--;  }  }  } | //**DOAR STANDARD**  public static boolean VALID(int p, int[] x, int k){  int q=0;  if(p==0)return false;  for(int i=1; i<=k-1; i++){  if(x[i]==x[k]){  q++;  if(q>=p) return false;  }  }  return true;  }  public static void main(String[] args) {  int m,n; int[] t;  cin = new Scanner(System.in);  System.out.print("n=");  n = cin.nextInt();  System.out.print("m=");  m = cin.nextInt();  t = new int[m+1];  CITESTE(t,m);  PERMUTARI\_CU\_REPETITIE\_STANDARD(n,t,m);  }  } |
| --- | --- |

**Compuneri**

m = 4, n=3

4 = 0 + 0 + 4 = 0 + 1 + 3 = 0 + 2 + 2 = 0 + 3 + 1 = 0 + 4 + 0

= 1 + 0 + 3 = 1 + 1 + 2 = 1 + 2 + 1 = 1 + 3 + 0 = 2 + 0 + 2

= 2 + 1 + 1 = 2 + 2 + 0 = 3 + 0 + 1 = 3 + 1 + 0 = 4 + 0 + 0.

| import java.util.Scanner;  class Main {  public static Scanner cin;  public static int[] nr;  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(nr[x[i]] + " ");  System.out.println();  }  public static void CITESTE(int[] t, int lungime){  for(int i=1; i<=lungime; i++)t[i] = cin.nextInt();  }  public static void COMPUNERI\_OARECARE(int m, int n){  int k=1;  int[] x = new int[n+1]; //stocheaza combinatia curenta  x[1]=-1;  int s=0; //suma totala a valorilor din combinatie  while(k>0){  if(x[k]<m-s){ // Dacă suma curentă + valoarea la indexul k este mai mică decât m  x[k]++;  if(k==n){ // Dacă am ajuns la ultima combinație  x[n]=m-s-x[k]; // Calculează valoarea rămasă pentru a completa suma m  AFISARE(x,n); // Afișează combinația curentă  }  else{  s=s+x[k]; //adauga suma  k++; //treci la urmatorul  x[k]=-1; //valoarea urm index setata la -1  }  }  else{ //daca suma curenta depaseste "m"  k--; //mergi inapoi la index anterior  s=s-x[k]; //si scade din suma totala  }  }  }  public static void main(String[] args) {  int m=4,n=3; //asumam valorile din curs  cin = new Scanner(System.in);  System.out.println("Citeste " + m + " numere oarecare: ");  nr = new int[m+1];  CITESTE(nr,m);//nr oarecare  COMPUNERI\_OARECARE(m,n);  }  } |
| --- |

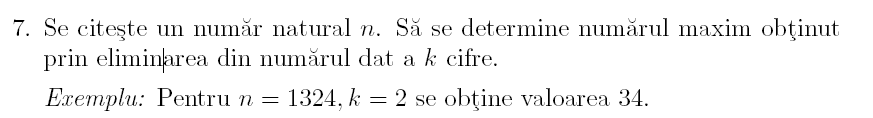
**Partitii**

m = 6

6 = 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

| import java.util.Scanner;  class Main {  public static Scanner cin;  public static int[] nr;  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(nr[x[i]] + " ");  System.out.println();  }  public static void CITESTE(int[] t, int lungime){  for(int i=1; i<=lungime; i++)t[i] = cin.nextInt();  }  public static void PARTITII(int m, int n){  int k=1;  int[] x = new int[n+1]; // stocam partitiile curente  x[1]=0;  int s=0; // Suma parțială  while(k>0){  if(k<=n-1) { //nu am ajuns la ultima parte  int temp= (m-s)/(n-k+1); //calc val max posibila  //pt element curent  if(x[k]<Math.abs(temp)){  x[k]++;  s=s+x[k];  k++;  x[k]=x[k-1]-1; //set val urm index  //pt a limita partitia  }  else{ //daca valoare depasim limita calculata  k--;  s=s-x[k]; //ne intoarcem la index anterior  }  }  else{  x[n]=m-s;  AFISARE(x,n);  k--;  s=s-x[k];  }  }  } | public static void main(String[] args) {  int m=6,n=6; //asumam valorile din curs  cin = new Scanner(System.in);  System.out.println("Citeste " + m + " numere oarecare: ");  nr = new int[m+1];  CITESTE(nr,m);//nr oarecare  for(int i=1; i<=n; i++){  System.out.println("Partitii de " + i + " numere: ");  PARTITII(m,i);  }  }  }  /\*  Citeste 6 numere oarecare:  1 2 3 4 5 6  Partitii de 1 numere:  6  Partitii de 2 numere:  1 5  2 4  3 3  Partitii de 3 numere:  1 1 4  1 2 3  2 2 2  Partitii de 4 numere:  1 1 1 3  1 1 2 2  Partitii de 5 numere:  1 1 1 1 2  Partitii de 6 numere:  1 1 1 1 1 1  \*/ |
| --- | --- |



| class Main {  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }  public static void NR\_MAXIM\_DUPA\_ELIMINARE\_CIFRE(int n, int k){  int max=0;  int temp=n;  //Calculam nr de cifre din numarul "n"  int nrCifre=0;  while(temp>0){  temp/=10;  nrCifre++;  }  int[] numar = new int[nrCifre+1];  //Le adaugam intr-un array in ordine corecta (nu inversa)  for(int i=1; i<nrCifre; i++){  numar[i]=n%10;  n/=10;  }  for(int i=1; i<=nrCifre-1; i++){  for(int j=i+1; j<=nrCifre; j++){  int suma=0, produs=1;  for(int l=1; l<=nrCifre; l++)  {  if(l!=i && l!=j){  suma+=numar[l]\*produs;  produs\*=10;  }  }  if(suma>max){  max=suma;  System.out.println(max + " este noul Maxim!");  }  }  }  }  public static void main(String[] args) {  int n=1324, k=2; //asumam valorile din curs  NR\_MAXIM\_DUPA\_ELIMINARE\_CIFRE(n,k);  }  } |
| --- |

**Cifra 1 Sa Apara de K ori**

| class Main {  public static void CIFRE\_DE\_1(int n, int k){  int nrMin=1, nrMax;  while(n>1){  n--;  nrMin\*=10;  }  nrMax=nrMin\*10;  for(int i=nrMin; i<=nrMax; i++){  if(CIFRA1\_APARE\_DE\_K\_ORI(i,k)) System.out.println(i);  }  }  public static boolean CIFRA1\_APARE\_DE\_K\_ORI(int i, int k){  int counter=0;  while(i>0){  if(i%10==1)counter++;  i=i/10;  }  if(counter==k) return true;  else return false;  }  public static void main(String[] args) {  int n=3, k=2; //asumam valorile din curs  CIFRE\_DE\_1(n,k);  }  } |
| --- |

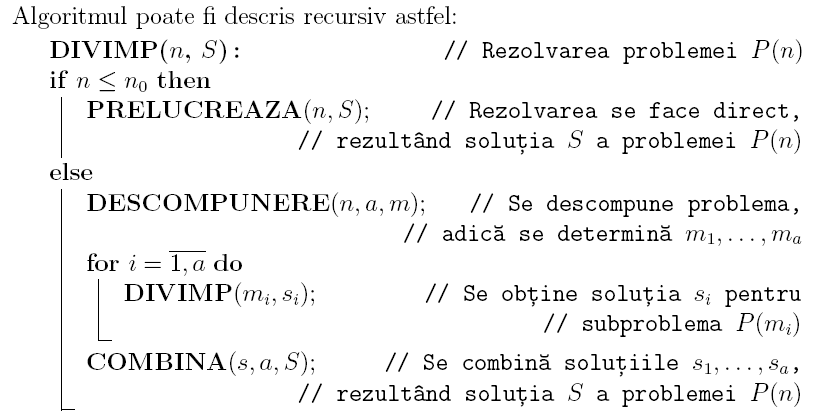
**HANOI**

| import java.util.Scanner;  class Main {  public static Scanner cin = new Scanner(System.in);    public static void AFISARE(int i, int j){  System.out.println(i + "," + j);  }    public static void HANOI(int m, int i, int j){  if(m==1)AFISARE(i,j);  else{  int k=6-i-j;  HANOI(m-1, i, k);  AFISARE(i,j);  HANOI(m-1, k, j);  }  } | public static void main(String[] args) {  int n;  cin = new Scanner(System.in);  System.out.print("n=");  n = cin.nextInt();  HANOI(n,1,2);  }}  //**COMPLEXITATE** Problema hanoi  Problema Turnurilor din Hanoi este un exercițiu clasic de recursiune.  Dacă avem m discuri, algo de mutare urmează recursiv pașii:  Mută m-1 discuri pe un stâlp auxiliar.  Mută discul rămas pe stâlpul de destinație.  Mută cele m-1 discuri de pe stâlpul auxiliar pe stâlpul de destinație.  Procesul se repetă pentru fiecare sub problemă mai mică.  Complexitatea timpului este **exponențială** în funcție de numărul de discuri **m**, adică **O(2^m)** |
| --- | --- |

**Metoda Divide et Impera**

**Metoda Divide et Impera („împarte și stăpânește”) = metodă generală a algoritmilor care are 3 etape:**

* **Divide:** Problema este împărțită în 2 sau mai multe subprobleme de același tip, dar de dimensiuni mai mici.
* **Impera:** Stăpânește subproblemele obținute. Subproblemele se rezolvă direct, dacă dimensiunea lor permite aceasta (cazuri elementare) sau, fiind de același tip, se rezolvă în mod recursiv, prin același procedeu.
* **Combină:** Se combină soluțiile tuturor subproblemelor pentru a obține soluția problemei inițiale.
* Rezolvarea unei subprobleme cu 1 element se poate face direct, fara descompunere.



**Problema turnurilor din Hanoi - O(2^n)**

* Rezolvarea unei subprobleme cu 1 element se poate face direct, fara descompunere.

**Se consideră trei tije notate cu 1, 2, 3 și n ≥ 1 discuri perforate de diametre diferite. Inițial, toate discurile se află pe tija 1, în ordinea crescătoare a diametrelor, considerând sensul de la vârf către bază.**

**Se cere mutarea discurilor pe tija 2, în aceeași ordine, utilizând tija 3 și respectând următoarele reguli:**

* La fiecare pas se mută un singur disc;
* Nu este permisă așezarea unui disc peste unul cu diametru mai mic.

**Modelarea problemei**Vom nota o mutare cu perechea (i,j)(i, j)(i,j), ceea ce semnifică faptul că se mută un disc de pe tija iii pe tija jjj (i,j∈{1,2,3}i, j \in \{1, 2, 3\}i,j∈{1,2,3}).

* Dacă n=1n = 1n=1, soluția directă este mutarea (1,2)(1, 2)(1,2).
* Dacă n=2n = 2n=2, o soluție este formată din mutările (1,3)(1, 3)(1,3), (1,2)(1, 2)(1,2), (3,2)(3, 2)(3,2).
* Dacă n>2n > 2n>2, problema se complică.

**Așadar, pentru m ≥ 2, mutarea a m discuri de pe tija i pe tija j este echivalentă cu:**

* Mutarea a m − 1 discuri de pe tija i pe tija k = 6 − i − j, utilizând pentru manevre tija j;
* Mutarea discului rămas de pe tija i pe tija j;
* Mutarea celor m − 1 discuri de pe tija k pe tija j, utilizând pentru manevre tija i.

**O problema de m ≥ 2 se rezolvarea 3 subprobleme, de același tip, de dimensiuni mai mici:**

* Două de dimensiune m − 1, care se vor rezolva, recursiv, în același mod;
* Una de dimensiune 1, care se rezolvă direct.

**Problema de Acoperire - O(n^2)**

Complexitatea est O(n^2) deoarece avem n^2 patrate. Fiecare subproblema are marimea n/2

Pentru rezolvarea problemei utilizăm următoarea strategie Divide et Impera:

**Se împarte suprafața pătrată în patru pătrate egale, formându-se patru suprafețe pătrate cu lungimea laturii 2ᵏ⁻¹.**

* Una dintre cele patru suprafețe are deja un pătrat unitar acoperit.
* Acoperim cu o placă trei pătrate unitare din centrul suprafeței mari astfel încât să acoperim câte un pătrat unitar și din celelalte trei suprafețe.
* Fiecare dintre cele patru suprafețe este acum în condițiile problemei, deci pentru fiecare suprafață se repetă procedeul de descompunere descris anterior.

| 3 | 3 | 4 | 4 | 8 | 8 | 9 | 9 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | 2 | 2 | 4 | 8 | 7 | 7 | 9 |
| 5 | 2 | 6 | 6 | 10 | 7 | 11 | **0** |
| 5 | 5 | 6 | 1 | 10 | 10 | 11 | 11 |
| 13 | 13 | 14 | 1 | 1 | 18 | 19 | 19 |
| 13 | 12 | 14 | 14 | 18 | 18 | 17 | 19 |
| 15 | 12 | 12 | 16 | 20 | 17 | 17 | 21 |
| 15 | 15 | 16 | 16 | 20 | 20 | 21 | 21 |

**Cautare Binara - O(log2 n)**

| import java.util.Scanner;  class Main {  public static Scanner cin;    public static void SORTARE(int[] a, int n){  for(int i=1; i<=n; i++){  for(int j=i+1; j<=n; j++){  if(a[i]>a[j]){  int temp = a[i];  a[i] = a[j];  a[j] = temp;  }  }  }  }    public static void AFISARE(int[] a, int n){  for(int i=1; i<=n; i++) System.out.print(a[i] + " ");  System.out.println();  }    public static int CAUTBIN\_REC(int[] A, int n, int p, int u, int x){  int m;    if(p>u) return -1;  else{  m=(p+u)/2;  if(x==A[m]) return m;  else{  if(x<A[m])return CAUTBIN\_REC(A,n,p,m-1,x);  else return CAUTBIN\_REC(A,n,m+1,u,x);  }  }  }    public static void main(String[] args) {  cin = new Scanner(System.in);  System.out.print("n=");  int n,x, poz;  n = cin.nextInt();    System.out.print("Scrie " + n + " numere:");  int[] A = new int[n+1];    for(int i=1; i<=n; i++) A[i] = cin.nextInt();    System.out.print("x=");  x = cin.nextInt();    SORTARE(A,n);  AFISARE(A,n);    poz = CAUTBIN\_REC(A,n,1,n,x);  if(poz==-1) System.out.println("Elementul nu a fost gasit.");  else{  System.out.println("A fost gasit la pozitia: " + poz);  }  System.out.println();    }  } | Arbore cautare binara n=10    **Element mediu = (n+1)/2 = 11/2 = [5,5] = 5** |
| --- | --- |

Se consideră un vector A=(a1,a2,…,an); **(Presupunem că vectorul A are elementele sortate crescător.)**

O căutare se poate încheia:

* fie fără succes, caz în care se returnează −1.
* se compara x cu elementul din mijloc a[m] unde m=(n+1)/2
* daca a[m] = x; //atunci căutarea se termina cu succes, caz în care se returnează o poziție în vector pe care se află valoarea căutată;
* daca a[m] > x // cautam in partea din stanga
* altfel a[m] < x // cautam in dreapta

**Interclasare** (2 vectori)

* Se compară elementele curente din A și B (adică aᵢ cu bⱼ), iar cel mai mic se depune în C pe poziția k.
* Crește k și indicele corespunzător vectorului din care s-a făcut depunerea (i sau j).
* Procesul continuă până când este epuizat unul dintre vectori. Elementele rămase în celălalt vector se vor adăuga, în ordinea în care se află, la sfârșitul lui C.
* Complexitate O(m+n) deoarece
  + Numărul de atribuiri în care intervin elemente ale vectorilor, numite și deplasări, notat cu Nd(m, n), este egal cu numărul de componente ale lui C, deci Nd(m, n) = m + n. (4.5.1)
  + Numărul de comparării de chei (comparării în care intervin elemente ale vectorilor), notat cu Nc(m, n), este egal cu numărul de execuții ale ciclului while numerotat cu (1). Cu fiecare astfel de comparare are loc și o atribuire pentru o componentă a vectorului C, adică o deplasare. La terminarea ciclului while numerotat cu (1) elementele unuia dintre vectorii A sau B au fost depuse integral în C, iar în celălalt vector au mai ramas “r” elemente nedepuse in C, care vor fi depuse in C dupa incheierea acestui ciclu.

| import java.util.Scanner; **//INTERCLASARE**  class Main {  public static Scanner cin;    public static void SORTARE(int[] a, int n){  for(int i=1; i<=n; i++){  for(int j=i+1; j<=n; j++){  if(a[i]>a[j]){  int temp = a[i];  a[i] = a[j];  a[j] = temp;  }  }  }  }    public static void AFISARE(int[] a, int n){  for(int i=1; i<=n; i++) System.out.print(a[i] + " ");  System.out.println();  }    public static void INTERCLASARE(int[] A, int[] B, int[] C, int m, int n){  int i=1, j=1, k=1;    while(i<=m && j<=n){  if(A[i]<=B[j]){  C[k]=A[i];  i++;  }  else{  C[k]=B[j];  j++;  }  k++;  }  while(i<=m){  C[k]=A[i];  i++;  k++;  }  while(j<=n){  C[k]=B[j];  j++;  k++;  }  } | public static void main(String[] args) {  cin = new Scanner(System.in);  int m,n;    System.out.print("n=");  n = cin.nextInt();  System.out.print("m=");  m = cin.nextInt();  int[] C=new int[m+n+1];    System.out.print("Scrie " + m + " numere:");  int[] A = new int[m+1];  for(int i=1; i<=m; i++) A[i] = cin.nextInt();    System.out.print("Scrie " + n + " numere:");  int[] B = new int[n+1];  for(int i=1; i<=n; i++) B[i] = cin.nextInt();      SORTARE(A,m);  AFISARE(A,m);    SORTARE(B,n);  AFISARE(B,n);    System.out.println();  INTERCLASARE(A,B,C,m,n);  AFISARE(C,m+n);    }  } |
| --- | --- |

**Sortare Prin Interclasare (mergesort) O(n log2 n)**

Pentru a sorta crescător vectorul A = (a1, a2, . . . , an), n ≥ 1, se interclasează succesiv secvențe ordonate crescător ale vectorului, până se obține secvența cu toate elementele sortate ale vectorului.

Se utilizează Divide et Impera:

• pentru a sorta un vector cu n componente îl împărțim în doi vectori care odată sortați se interclasează;

• un vector cu o singură componentă este DEJA sortat.

Pentru interclasare vom utiliza un vector intermediar, de lucru, B. Mai întâi se interclasează cele 2secvențe:

(ap, . . . , am) și (am+1, . . . , aq), rezultatul interclasării depunându-se în vectorul B:

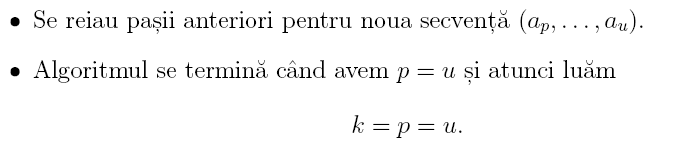
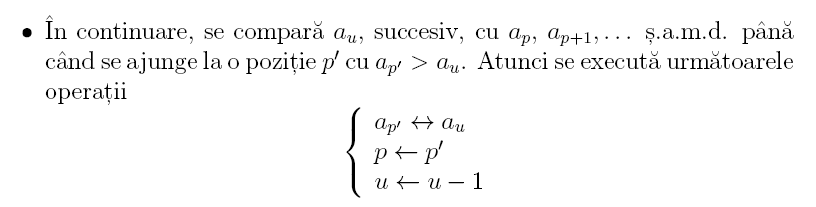
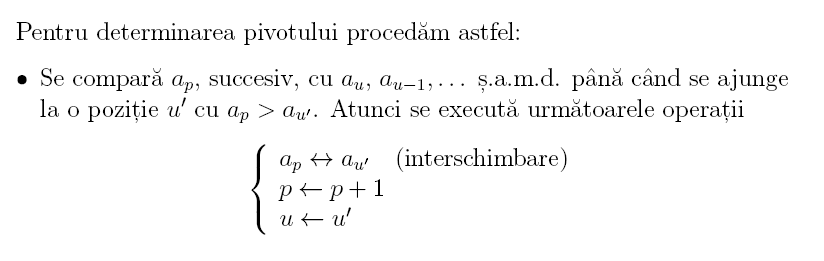
| import java.util.Scanner;  class Main {  public static Scanner cin;  public static void SORTINT(int[] a, int p, int q) {  if (p < q) {  int m = (p + q) / 2;  SORTINT(a, p, m);  SORTINT(a, m + 1, q);  INTERCLAS(a, p, q, m);  }  }  public static void AFISARE(int[] a, int n) {  for (int i = 0; i < n; i++) {  System.out.print(a[i] + " ");  }  System.out.println();  }  public static void INTERCLAS(int[] A, int p, int q, int m) {  int i = p;  int j = m + 1;  int k = 0; // Changed to start from 0  int[] B = new int[q - p + 1]; // Size should be q - p + 1  while (i <= m && j <= q) {  if (A[i] <= A[j]) {  B[k] = A[i];  i++;  } else {  B[k] = A[j];  j++;  }  k++;  }    while (i <= m) {  B[k] = A[i];  i++;  k++;  }    while (j <= q) {  B[k] = A[j];  j++;  k++;  } | // Copy back to original array  for (i = p; i <= q; i++) {  A[i] = B[i - p]; // Adjust index for B  }  }  public static void main(String[] args) {  cin = new Scanner(System.in);  System.out.print("m=");  int m = cin.nextInt();  System.out.print("Scrie " + m + " numere:");  int[] A = new int[m]; // Changed size to m  for (int i = 0; i < m; i++) { // Changed loop to start from 0  A[i] = cin.nextInt();  }  System.out.println();  SORTINT(A, 0, m - 1); // Changed parameters to match zero-based indexing  AFISARE(A, m); // Display sorted array  }  } |
| --- | --- |

1. **Împărțirea în subliste**: La fiecare pas, lista este împărțită în două jumătăți. De exemplu, dacă ai o listă de 8 elemente, o vei împărți în două subliste de 4, apoi fiecare sublistă în câte două de 2 și tot așa. De fiecare dată, numărul de divizări este de **log2(n)**, unde n este numărul de elemente. De exemplu, pentru 8 elemente, vei face 3 împărțiri (log2(8) = 3).
2. **Combinarea sublistelor**: După ce ai obținut subliste de dimensiuni mici, le combini două câte două pentru a le ordona. Procesul de combinare presupune compararea fiecărui element din sublistă, iar acest lucru durează **O(n)**, deoarece trebuie să parcurgi toată lista.

Deci, la fiecare nivel al diviziunii, trebuie să faci **O(n)** operațiuni de combinare, iar numărul de niveluri este **log2(n)**. Astfel, complexitatea totală este **O(n log2 n)**.

**Sortare rapida (quicksort) O(n^2)**

* **O(n^2)** - in cazul cel mai nefavorabil
* **O(nlog2n)** - in cazul mediu
* În procedura **PIVOT(A, p, u)** se efectuează **u - p** comparații de chei.
* Pentru orice secvență, de la “p” primul la “u” ultimul
* Prin interschimbari succesive
* Secvența de lungime 1 în urma partitionarii nu se mai prelucrează.
* Un element intre a[i] <= a[k] <= a[j]. Elementulk a[k] se numeste pivot.
* Pivotul:
  + A ajuns deja pe poziția finală, în următorul vector sortat, deoarece toate elementele la stanga lui sunt mai mici decat el, și toate la dreapta sunt mai mari.



Algoritmul de sortare rapidă (**quicksort**) este ineficient în situațiile în care se întâmplă ca, în mod sistematic, în cele mai multe apelări succesive, cele două partiții care se obțin să fie „foarte neechilibrate” (adică să aibă dimensiuni diferite, una foarte mare și cealaltă foarte mică). Cu cât dimensiunile celor două partiții sunt mai apropiate, cu atât se reduce numărul comparațiilor de chei efectuate.

Un mod de a realiza echilibrarea constă în încercarea de a obține ca pivot elementul de pe poziția mediană a secvenței de sortat.

De asemenea, în cazurile cele mai defavorabile (vector deja sortat), la vectori de dimensiuni mari, este posibilă apariția unor erori la execuție de genul **„depășirea stivei”** (*stack overflow*). Unele dintre aceste situații pot fi evitate, dacă se elimină apelurile recursive inutile.

| import java.util.Scanner; **//QUICKSORT**  class Main {  public static Scanner cin = new Scanner(System.in);  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++){  System.out.print(x[i] + " ");  }  System.out.println();  }  public static int PIVOT(int[] A, int p, int u){  int cp=0 //pas crestere al lui “p”  int cu=-1; //pas crestere al lui “u”  while(p<u){  if(A[p]>A[u]){  int temp = A[p];  A[p]=A[u];  A[u]=temp;    int m=cp; //se interschimba (cu semn schimbat)  cp=-cu; //valorile pașilor de creștere  cu=-m;  }  p=p+cp;  u=u+cu;  }  int k=p;  return k;  }  public static void QUICKSORT(int[] A, int p, int u) //sortează secvența de la primul la ultimul  {  if(p<u){  int k=PIVOT(A,p,u); //se determina pozitia pivotului  QUICKSORT(A,p,k-1);  QUICKSORT(A,k+1,u);  }  }  public static void main(String[] args) {  int n;  System.out.print("n="); n = cin.nextInt();  int[] A= new int[n+1];    System.out.print("Dati numerele: ");  for(int i=1; i<=n; i++) A[i] = cin.nextInt();    QUICKSORT(A,1,n);  AFISARE(A,n);  }  } |
| --- |

**Divide Timpera**

| import java.util.Scanner; **//MIN DIN SIR**  class Main {  public static Scanner cin = new Scanner(System.in);  public static int[] sir;  public static int MIN\_SIR(int[] sir, int p, int u)  {  if(p==u) return sir[p];    int m = (p+u)/2;  int a = MIN\_SIR(sir,p,m);  int b = MIN\_SIR(sir,m+1,u);    if(a<b) return a;  else return b;  }  public static void main(String[] args) {  int n, min;  System.out.print("n="); n = cin.nextInt();  sir= new int[n+1];    System.out.print("Dati numerele: ");  for(int i=1; i<=n; i++) sir[i] = cin.nextInt();      min=MIN\_SIR(sir,1,n);  System.out.print("Minim: " + min);  }}  import java.util.Scanner; **//CMMDC**  class Main {  public static Scanner cin = new Scanner(System.in);  public static int[] sir;  public static int CMMMDC\_SIR(int[] sir, int p, int u)  {  if(p==u) return sir[p];    int m = (p+u)/2;  int a = CMMMDC\_SIR(sir,p,m);  int b = CMMMDC\_SIR(sir,m+1,u);    while(b!=0){  int r = a%b;  a=b;  b=r;  }  return a;  }  public static void main(String[] args) {  int n, min;  System.out.print("n="); n = cin.nextInt();  sir= new int[n+1];    System.out.print("Dati numerele: ");  for(int i=1; i<=n; i++) sir[i] = cin.nextInt();  min=CMMMDC\_SIR(sir,1,n);  System.out.print("CMMDC: " + min);  }} | import java.util.Scanner; **//K\_lea cel mai mic din sir**  class Main {  public static Scanner cin = new Scanner(System.in);  public static int[] sir;  public static int K\_lea\_MINIM(int[] arr, int p, int u, int k)  {  int i = p; //i = Cate elementele sunt mai mici decat ultimul (arr[u]), care l-am ales ca un fel de pivot.  for(int j=p; j<u; j++){  if(arr[j]<=arr[u]){  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  i++;  }  }  //Muta pivot la poz corecta.  int temp=arr[i];  arr[i]=arr[u];  arr[u]=temp;  //Acest algoritm este ca o sortare, dar doar partiala,  //deoarece cand a gasit al k-lea element, se va opri din sortat.  //Verifica poz pivot:  if(i==k) return arr[i];  else if(i>k) return K\_lea\_MINIM(arr,p,i-1,k);  else return K\_lea\_MINIM(arr,i+1,u,k);  }  public static void main(String[] args) {  int n, k, min;  //if n<min[0]  System.out.print("n="); n = cin.nextInt();  sir= new int[n+1];  System.out.print("Dati numerele: ");  for(int i=1; i<=n; i++) sir[i] = cin.nextInt();  System.out.print("k="); k = cin.nextInt();  min= K\_lea\_MINIM(sir,1,n, k);  System.out.println("Al " + k + "-lea minim nr este: " + min);  }  } |
| --- | --- |

**Fibonacci**

| import java.util.Scanner;  class Main {  public static Scanner cin;  public static int t[];    public static void AFISARE(int[] a, int n){  for(int i=1; i<=n; i++) System.out.print(a[i] + " ");  System.out.println();  }    //Varianta 1  public static int F(int n){  if(n<=1)return n;  else return F(n-1) + F(n-2);  }    //Varianta 2  public static int F2(int n){  if(n<=1)return n;  else if(t[n]!=0)return t[n];  else{  t[n]=F2(n-1)+F2(n-2);  return t[n];  }  }    //Varianta 3  public static void Fib(int[] F, int n){  F[0]=0;  F[1]=1;  for(int i=2; i<=n; i++){  F[i]=F[i-1]+F[i-2];  }  AFISARE(F,n);  }    //Varianta 4  public static int F4(int n){  int a,b,c=0;  if(n<=1)return n;  else{  a=0;  b=1;  for(int i=1; i<=n; i++){  c=b+a;  a=b;  b=c;  }  return c;  }    } | public static void main(String[] args) {  cin = new Scanner(System.in);  System.out.print("n=");  int n;  n = cin.nextInt();  t = new int[n+1];  int[] F = new int[n+1];    //System.out.print("Scrie " + n + " numere:");  //int[] A = new int[n+1];  Fib(F,n);    //Exemplu apel F4  //System.out.println(F4(n));    }  } |
| --- | --- |

**Metoda programării dinamice**

Ca și metoda **Divide et Impera**, metoda **programării dinamice** implică rezolvarea unei probleme prin descompunerea în subprobleme de același tip, de dimensiuni mai mici. Însă, de această dată, subproblemele **nu sunt disjuncte** (ca la metoda **Divide et Impera**), ci se pot **suprapune**, în sensul că două subprobleme pot avea (sub)subprobleme comune.

Soluția problemei se construiește combinând soluțiile subproblemelor, numite și **subsoluții** (*soluții parțiale*).

Pentru a evita repetarea procesului de calcul necesar rezolvării aceleiași subprobleme, soluțiile subproblemelor se memorează temporar (de obicei în structuri de date de tip tablouri) pentru reutilizări la posibile viitoare apeluri ale acestor subprobleme.

Soluțiile optime ale problemelor se exprimă în funcție de soluțiile optime ale subproblemelor lor, de obicei prin intermediul unor relații de recurență, și astfel determinarea soluțiilor optime revine la scrierea și rezolvarea acestor relații de recurență.

a) Dacă relația de recurență la soluția optimă exprimă elementul S[i₁, . . . , iᵣ] în funcție de elemente următoare (lexicografic), atunci calculul acestor elemente se face în ordinea inversă, descrescătoare, a indicilor (programare dinamică în varianta înapoi.

b) Dacă relația de recurență la soluția optimă exprimă elementul S[i₁, . . . , iᵣ] în funcție de elemente anterioare (lexicografic), atunci calcul elemente se face în ordinea directă, crescătoare, a indicilor. (varianta înainte)

**Subsir crescator lungime maxima**

10, 16, 2, 8, 4, 10, 3, 4, 9, 8

Lungimea maximă 4, iar un subșir crescător de lungime maximă este, de exemplu, 2, 3, 4, 8.

Pentru memorarea soluțiilor subproblemelor utilizăm doi vectori:

L = (L[1], . . . , L[n]) și P = (P[1], . . . , P[n − 1]) având semnificația:

* **L[i]** = lungimea maximă a unui subșir crescător care începe cu elementul **aᵢ**.
* **P[i]** = indicele primului element care este succesor al lui **aᵢ** într-un subșir crescător de lungime maximă care începe cu **aᵢ**, dacă există un astfel de indice.
* **P[i] = -1**, dacă nu există un astfel de indice.
* Evident, **L[n] = 1**.

|  | **Rezolvare:**  Pentru șirul: 10, 16, 2, 8, 4, 10, 3, 4, 9, 8  Aplicam algoritmul de programare dinamica in varianta inapoi.  Avem n=10 :  **<** //aici scrie ce e la stanga  Deci, lungimea maximă a unui subșir crescător este k=4.  Iar un subsir crescator de lungime maxima este dat de indicii:  t4=9;  t3=P[t4]=8;  t2=P[t3]=6;  t1=P[t2]=2;  **Adica subsirul (5,6,7,10)** |
| --- | --- |

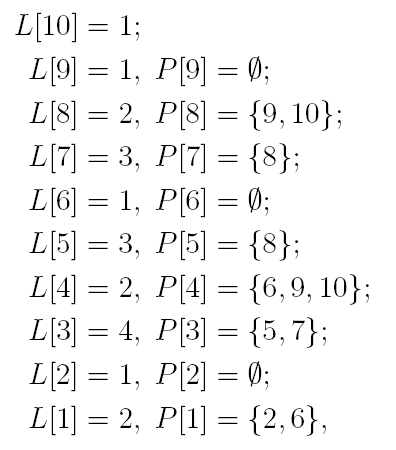
| **//SUBSIR MAX (varianta inapoi, top-down)**  import java.util.Scanner;  class Main {  public static Scanner cin;  public static int[] L, P, t, a;    public static void SUBSIRMAX1(int[] a, int n){  int[] L = new int[n+1];  int[] P = new int[n+1];  int[] t = new int[n+1];    L[n]=1;  for(int i=n-1; i>=1; i--){  L[i]=1;  P[i]=-1;    for(int j=i+1; j<=n; j++){  if(a[j]>=a[i] && (1+L[j]>L[i])){  L[i]=1+L[j];  P[i]=j;  }  }  }  int k=L[1];  t[1]=1;  for(int i=2; i<=n; i++){  if(L[i]>k){  k=L[i];  t[1]=i;  }  }  for(int j=2; j<=k; j++) t[j]=P[t[j-1]];    for(int j=1; j<=k; j++) System.out.print(a[t[j]] + " ");  }        public static void main(String[] args) {  cin = new Scanner(System.in);  System.out.print("n="); int n = cin.nextInt();  int[] a = new int[n+1];    System.out.print("Scrie " + n + " numere:");  for(int i=1; i<=n; i++) a[i] = cin.nextInt();  //10 16 2 8 4 10 3 4 9 8    SUBSIRMAX1(a,n);  }  } | **//SUBSIR MAX (varianta inainte, bottom-up)**  import java.util.Scanner;  class Main {  public static Scanner cin;  public static int[] L, P, t, a;    public static void SUBSIRMAX2(int[] a, int n){  int[] L = new int[n+1];  int[] P = new int[n+1];  int[] t = new int[n+1];    L[1]=1;  for(int i=2; i<=n; i++){  L[i]=1;  P[i]=-1;    for(int j=1; j<=i-1; j++){  if(a[j]<=a[i] && 1+L[j]>L[i]){  L[i]=1+L[j];  P[i]=j;  }  }  }    int k=L[1];  int m=1;    for(int i=2; i<=n; i++){  if(L[i]>k){  k=L[i];  m=i;  }  }    t[k]=m;  for(int j=k-1; j>=1; j--) t[j]=P[t[j+1]];  for(int j=1; j<=k; j++)System.out.print(a[t[j]] + " ");  }    public static void main(String[] args) {  cin = new Scanner(System.in);  System.out.print("n="); int n = cin.nextInt();  int[] a = new int[n+1];    System.out.print("Scrie " + n + " numere:");  for(int i=1; i<=n; i++) a[i] = cin.nextInt();  //10 16 2 8 4 10 3 4 9 8    SUBSIRMAX2(a,n);  }  } |
| --- | --- |

Instructiunea (a[j] ≥ a[i]) and (1 + L[j] > L[i])

se execută cate o data pentru fiecare pereche de indici (i, j)

Deci algoritmul are complexitatea **O(n^2)**

**Subșir lungime maximă cu memorarea tuturor drumurilo**r:



(a3 = 2, a5 = 4, a8 = 4, a9 = 9), (a3 = 2, a5 = 4, a8 = 4, a10 = 8),

(a3 = 2, a7 = 3, a8 = 4, a9 = 9), (a3 = 2, a7 = 3, a8 = 4, a10 = 8),

**Deci avem 4 solutii** (subșiruri crescătoare de lungime maximă).

|  | Aplicam algoritmul de programare dinamica in varianta inainte:  n=10  Deci lungimea maximă a unui subșir crescător este k=4  Iar un subsir crescator de lungime maxima este dat de indicii:  t4 = 9, t3 = P[t4] = 8, t2 = P[t3] = 5, t1 = P[t2] = 3,  Adica subsirul:  2, 4, 4, 9. |
| --- | --- |

**Subsir comun de lungime maxima**

a = (5, 0, 2, 1, 3, 4, 0, 2)

b = (2, 3, 0, 0, 1, 2, 5, 3, 0)

Max=4

Un subsir comun (de lungime max) este:

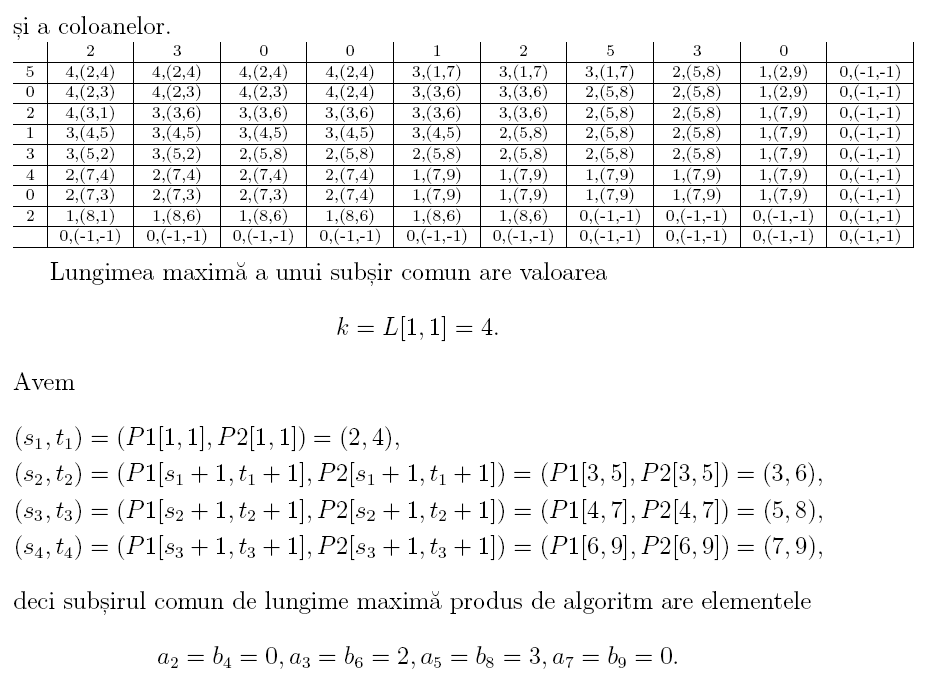
(a2, a3, a5, a7) = (0, 2, 3, 0) = (b3, b6, b8, b9).

Pentru memorarea soluțiilor folosim 2 matrici L[i,j] si P[i,j]

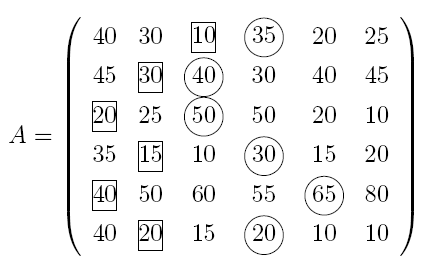
Pt. calculul fiecărui element al matricelor **L, P₁** și **P₂** sunt necesare câte 1 atribuire și cel mult 2 comparații,

deci algoritmul are complexitatea **O(mn)**.

| import java.util.Scanner;  class Main {  public static Scanner cin;  static int[][] A, S, P;  static int[] t;  static int m,n;    public static void AFISARE(int[] a, int n){  for(int i=1; i<=n; i++) if(a[i]!=0) System.out.print(a[i] + " ");  System.out.println(); }    public static void DRUMMAX1(int[] A, int m, int n){  for(int j=1; j>=1; j--)S[m][j]=A[m][j];  for(int i=m-1; i>=1; i--){  P[i][1]=0;  if(S[i+1][2]>S[i+1][1])P[i][1]=1;  S[i][1]=A[i][1]+ S[i+1][1+P[i][1]];    for(int j=2; j<=n-1; j++){  P[i][j]=-1;  if(S[i+1][j]>S[i+1][j-1])P[i][j]=0;  if(S[i+1][j+1]>S[i+1][j+P[i][j]])P[i][j]=1;  S[i][j]=A[i][j]+S[i+1][j+P[i][j]];  }  P[i][n]=-1;    if(S[i+1][n]>S[i+1][n-1]) P[i][n]=0;  S[i][n]=A[i][n]+ S[i+1][n+P[i][n]];  }  int M=S[1][1];  t[1]=1;  for(int j=2; j<=n; j++){  if(S[1][j]>M){  M=S[1][j];  t[1]=j;  }  }  for(int i=2; i<=m; i++){  t[i]=t[i-1]+P[i-1][t[i-1]];  }  System.out.println(M);  for(int i=1; i<=m; i++)System.out.print(a[i][t[i]]);  }    public static void main(String[] args) {  cin = new Scanner(System.in);  System.out.print("n="); int n = cin.nextInt();  System.out.print("m="); int m = cin.nextInt();  S=new int[m+1][n+1];  P=new int[m][n+1];  t=new int[N+1];    int[] a = new int[n+1];    System.out.print("Scrie " + n + " numere:");  for(int i=1; i<=n; i++) a[i] = cin.nextInt();  //m=n=4  DRUMMAX1(A, m, n);  }  } |
| --- |



**Drum de suma maxima/minima**



Un drum de sumă maximă este format din elementele încadrate în cercuri, având suma **240**.

Pentru aceeași matrice, un drum de sumă minimă este format din elementele încadrate în pătrate, având suma **135**.

Pentru calculul fiecărui element al matricelor **S** și **P** sunt necesare câte o atribuire și cel mult două comparații, deci algoritmul are complexitatea **O(mn)**.

| import java.util.Scanner;  public class Main  {  static int A[][];  static int m,n;  public static void DrumMax2(int A[][], int m, int n)  {  int P[][],S[][];  int t[];  int M;  S=new int[m+1][n+1];  P=new int[m][n+1];  t=new int[n+1];  for(int j=1;j<=n;j++) S[1][j]=A[1][j];  for(int i=1;i<=m-1;i++)  {  P[i][1]=0;  if(S[i][2]>S[i][1]) P[i][1]=-1;  S[i+1][1]=A[i+1][1]+S[i][1-P[i][1]];  for(int j=2;j<=n-1;j++)  {  P[i][j]=1;  if(S[i][j]>S[i][j-1]) P[i][j]=0;  if(S[i][j+1]>S[i][j-P[i][j]]) P[i][j]=-1;  S[i+1][j]=A[i+1][j]+S[i][j-P[i][j]];  }  P[i][n]=1;  if(S[i][n]>S[i][n-1]) P[i][n]=0;  S[i+1][n]=A[i+1][n]+S[i][n-P[i][n]];  }  M=S[m][1];  t[m]=1;  for(int j=2;j<=n;j++)  {  if(S[m][j]>M)  {  M=S[m][j];  t[m]=j;  }  }  for(int i=m-1;i>=1;i--)  t[i]=t[i+1]-P[i][t[i+1]];  System.out.println(M);  for(int i=1;i<=m;i++) System.out.println(A[i][t[i]]);  }  public static void main(String[] args) {  Scanner cin = new Scanner(System.in);  System.out.print("m="); m=cin.nextInt();  System.out.print("n="); n=cin.nextInt();  A=new int[m+1][n+1];  for(int i=1;i<=m;i++)  for(int j=1;j<=n;j++)  A[i][j]=cin.nextInt();  //40 30 10 35 20 25 45 30 40 30 40 45 20 25 50 50 20 10 35 15 10 30 15 20 40 50 60 55 65 80 40 20 15 20 10 10  DrumMax2(A,m,n);  }} | import java.util.Scanner;  public class Main  {  static int A[][];  static int m,n;  public static void DrumMin1(int A[][], int m, int n)  {  int P[][],S[][];  int t[];  int M;  S=new int[m+1][n+1];  P=new int[m][n+1];  t=new int[n+1];  for(int j=1;j<=n;j++) S[1][j]=A[1][j];  for(int i=1;i<=m-1;i++)  {  P[i][1]=0;  if(S[i+1][2]>S[i+1][1]) P[i][1]=1;  S[i][1]=A[i][1]+S[i+1][1+P[i][1]];  for(int j=2;j<=n-1;j++)  {  P[i][j]=-1;  if(S[i+1][j]<S[i+1][j-1]) P[i][j]=0;  if(S[i+1][j+1]<S[i+1][j+P[i][j]]) P[i][j]=1;  S[i][j]=A[i][j]+S[i+1][j+P[i][j]];  }  P[i][n]=-1;  if(S[i+1][n]>S[i+1][n-1]) P[i][n]=0;  S[i][n]=A[i][n]+S[i+1][n+P[i][n]];  }  M=S[1][1];  t[1]=1;  for(int j=2;j<=n;j++)  {  if(S[1][j]<M)  {  M=S[1][j];  t[1]=j;  }  }  for(int i=2;i<=m;i++) t[i]=t[i-1]+P[i-1][t[i-1]];  System.out.println(M);  for(int i=1;i<=m;i++) System.out.println(A[i][t[i]]);  }  public static void main(String[] args) {  Scanner cin = new Scanner(System.in);  System.out.print("m="); m=cin.nextInt();  System.out.print("n="); n=cin.nextInt();  A=new int[m+1][n+1];  for(int i=1;i<=m;i++)  for(int j=1;j<=n;j++)  A[i][j]=cin.nextInt();  //m=n=6  DrumMin1(A,m,n);  }  } |
| --- | --- |

Aplicam algoritmul de programare dinamica in varianta inainte:

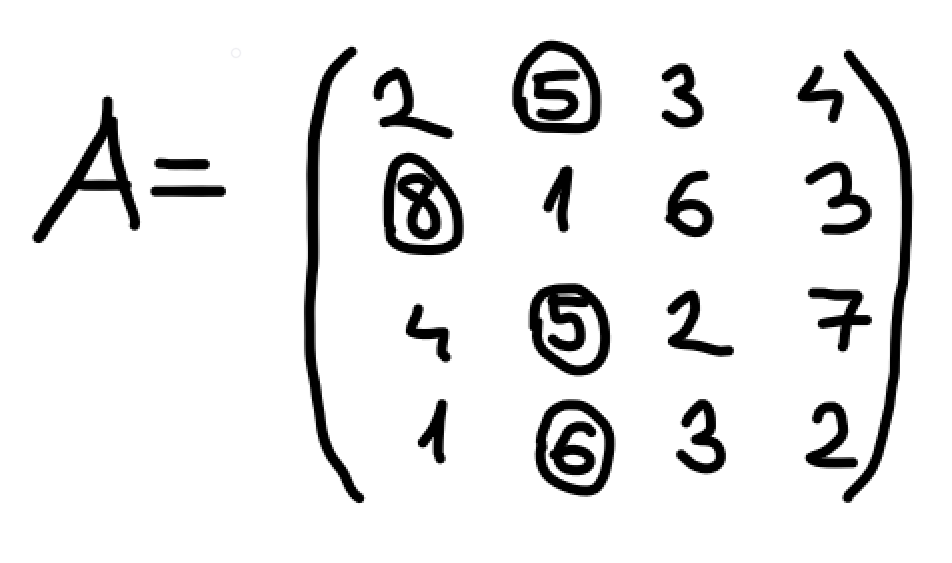
|  | **Explica:** Dacă începem cu un element de pe prima linie și coborâm pe diagonale, alegând la fiecare pas valoarea maximă dintre opțiunile posibile (adică urmărind o cale optimă care se extinde de la elementul curent până la ultima linie).  * **Varianta înainte** presupune că începem de la o stare inițială (elementele de pe prima linie) și calc pașii succesivi pentru a ajunge la soluția finală (ultima linie). * La fiecare pas, decizia de alegere (care element să alegem pe linia următoare) se bazează pe soluțiile deja calculate pentru liniile anterioare. |
| --- | --- |

Suma maxima are valoarea M=24

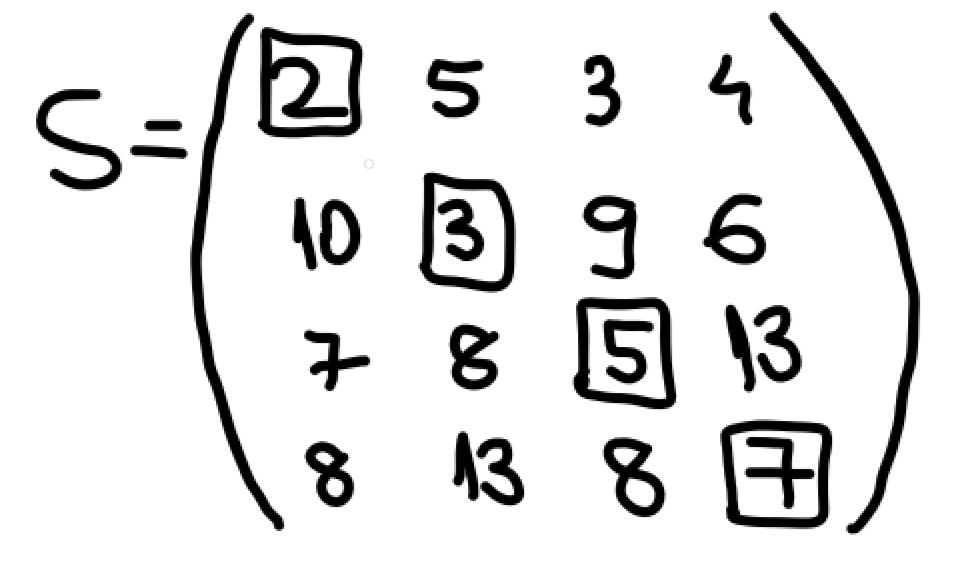
Deci drumul de suma maxima produs de algoritm are elementele:

a[4][2] = 6; a[3][2] = 5; a[2][1] = 8; a[1][2] = 5;

Incadrate in cercuri:



Aplicam algoritmul de programare dinamica in varianta inainte:

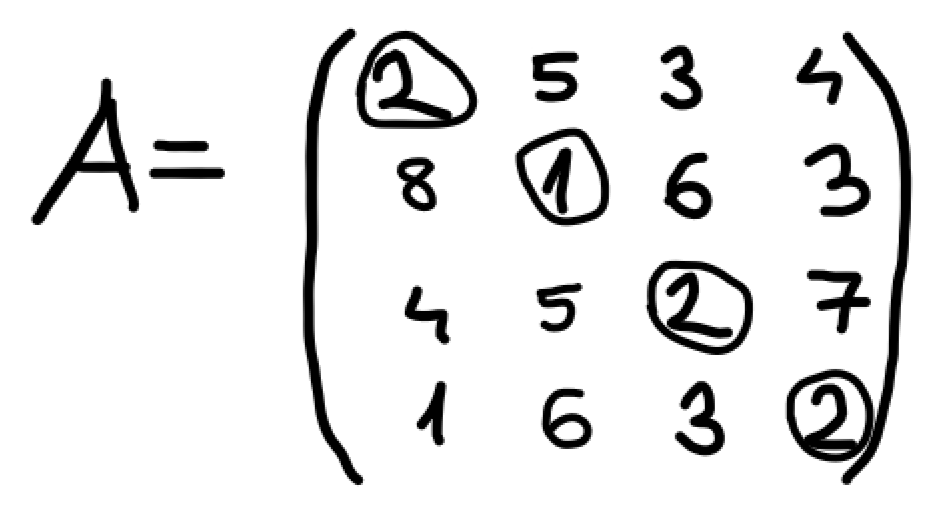


Suma minima are valoarea M=7

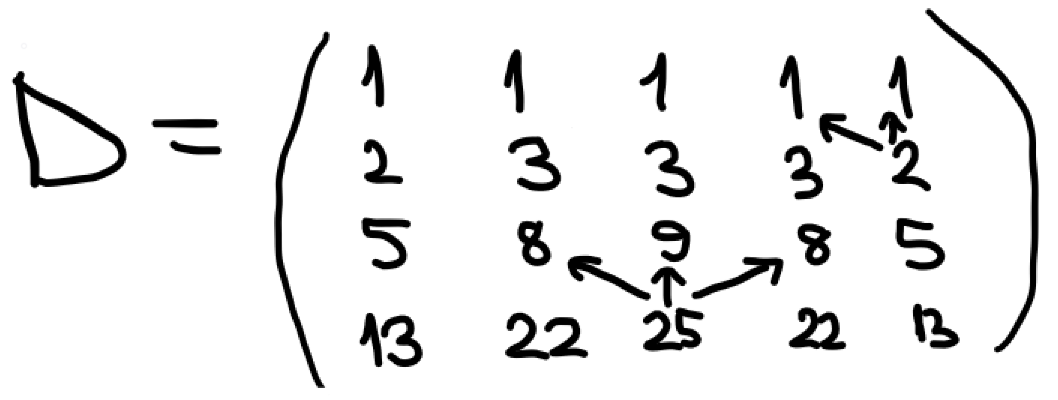
Deci drumul de suma minima produs de algoritm are elementele:

a[4][4] = 2; a[3][3] = 2; a[2][2] = 1; a[1][1] = 2;

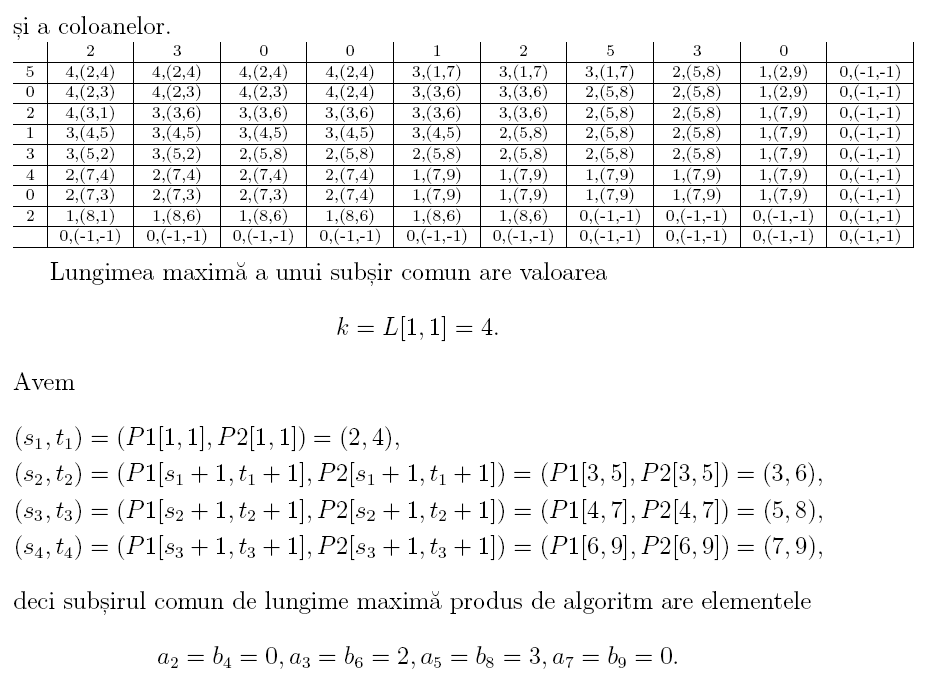
Incadrate in cercuri:



Cate drumuri (in jos sau pe diagonala in jos) exista intr-o matrice m\*n?



TOTAL: 13+22+25+22+13=95



**SUBIECTE EXAMEN**

**Subiectul 1)** Fie recurenţa Xn = 10X(n-1)-9X(n-2)+3n-2; X0 = -1; X1 = 1.

a) Calculaţi X4.

b) Descrieţi şi implementați (în C++ sau Java) un algoritm pentru calculul lui In.

c) Evaluați complexitatea şi eficienţa acestui algoritm.

1. X2 = 10 + 9 + 6 -2 = 23  
   X3 = 10\*23 - 9\*1 + 9 -2 = 228  
   X4 = 19\*228 - 9\*23 + 12 - 2 = 2083
2. S

| import java.util.Scanner;  public class Main {  public static int calculateX(int n) {  // Initializam valorile de bază  int X0 = -1, X1 = 1;  int Xn = 0, XnMinus1 = X1, XnMinus2 = X0;    // Calculăm recursiv Xn pentru n > 1  for (int i = 2; i <= n; i++) {  Xn = 10 \* XnMinus1 - 9 \* XnMinus2 + 3 \* i - 2;  XnMinus2 = XnMinus1;  XnMinus1 = Xn;  }  return XnMinus1; // Xn este stocat în XnMinus1 la final  }  public static void main(String[] args) {  Scanner scanner = new Scanner(System.in);  System.out.print("Introduceti valoarea lui n: ");  int n = scanner.nextInt();  System.out.println("Valoarea lui X" + n + " este: " + calculateX(n));  }  } |
| --- |

1. Algoritmul propus este iterativ, astfel încât complexitatea sa temporală este:

* Algoritmul parcurge o singură dată recurența pentru fiecare valoare a lui n. Astfel, complexitatea este O(n), adică liniară, deoarece calculul pentru fiecare Xn​ depinde de valoarea anterioară Xn−1​ și Xn−2, iar acești pași sunt realizați doar 1 dată.
* **Complexitatea spațială:** Algoritmul utilizează un număr constant de variabile pentru a stoca valorile lui anterioare, astfel încât complexitatea spațială este O(1), adică constantă.

**Subiectul 2) Descrieți și implementati in Java un algoritm pentru generarea tuturor cuvintelor de lungime n care pot fi formate cu litere din mulțimea {a,b,c,d,e,f} astfel incat sa contina cel mult un "b", cel putin 5 de "d", si cel putin 3 de "e". Evaluați complexitatea și eficienta acestui algoritm.**

| public class Main{  private static final char[] letters = {'a', 'b', 'c', 'd', 'e', 'f'}; // Setul de litere posibile  public static void genereazaCuvinte(int n) {  char[] word = new char[n]; // Cuvântul curent  generateCombinations(word, 0, n);  }  private static void genereazaCombinari(char[] word, int index, int n) { **// F. recursivă pt generarea toate combinații**  if (index == n) {  if (isValid(word)) System.out.println(new String(word)); **// Verificăm dacă cuvântul respectă condițiile**  return;  }  // Iterăm prin fiecare literă posibilă  for (char letter : letters) {  word[index] = letter;  genereazaCombinari(word, index + 1, n);  }  }  // Verificarea dacă un cuvânt respectă condițiile  private static boolean isValid(char[] word) {  int bCount = 0;  int dCount = 0;  int eCount = 0;  // Numărăm aparițiile literelor "b", "d" și "e"  for (char c : word) {  if (c == 'b') bCount++;  if (c == 'd') dCount++;  if (c == 'e') eCount++;  }  // Verificăm condițiile  return bCount <= 1 && dCount >= 5 && eCount >= 3;  }  public static void main(String[] args) {  int n = 8; // Exemplu de lungime a cuvântului  genereazaCuvinte(n);  }  } |
| --- |

Descrieți și implementati in Java un algoritm eficient pentru verificarea dacă un nr întreg k dat este sau nu termen al șirului (an) definit prin: an = (n+1)(n+3), n=1,2,....,1000. Evaluați complexitatea și eficient acestui algoritm.

| public class Main {  public static boolean esteInSecventa(int k) { // Func verifică dacă k este termen al șirului  for (int n = 1; n <= 1000; n++) {  int An = (n + 1) \* (n + 3);  if (An == k) {  return true;  }  }  return false;  }  public static void main(String[] args) {  int k = 35; // Exemplu nr de verificat  if (esteInSecventa(k)) System.out.println(k + " este termen al șirului.");  else System.out.println(k + " nu este termen al șirului.");  }  } |
| --- |

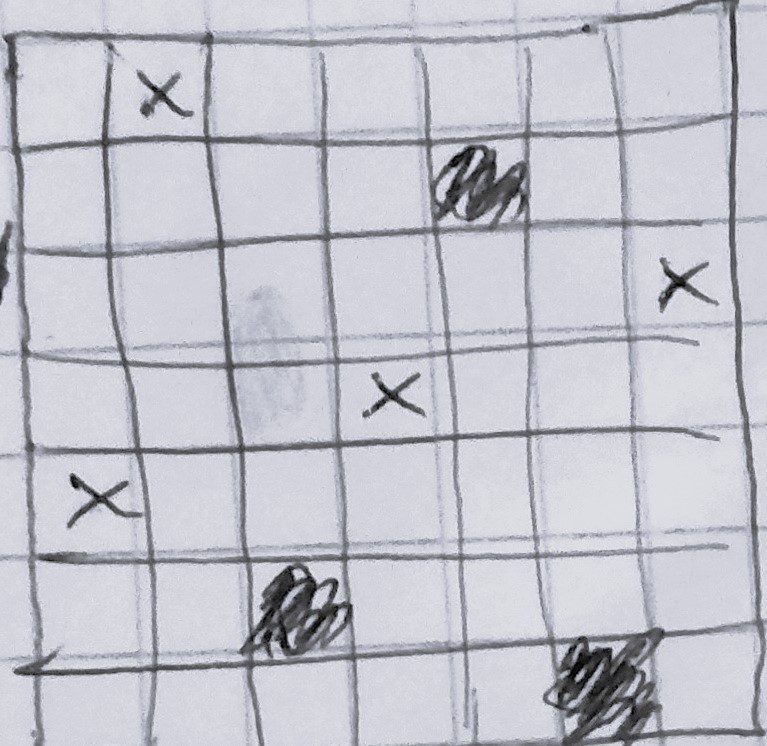
* **Complexitatea de timp**: Algoritmul parcurge valorile lui n de la 1 la 1000, iar pentru fiecare valoare de n, se efectuează 1 operație constantă (calculul termenului an). Astfel, complexitatea de timp este O(1000), ceea ce este echivalent cu O(1), având în vedere că 1000 este un număr fix și nu depinde de dimensiunea de intrare.
* **Complexitatea de spațiu**: Algoritmul folosește o cantitate constantă de memorie, deoarece doar câteva variabile sunt utilizate pentru a stoca rezultatele intermediare (fără a folosi structuri de date care cresc în dimensiune în funcție de intrare). Astfel, complexitatea de spațiu este O(1).

### **Eficiența algoritmului:**

* Algoritmul este destul de eficient, deoarece verificarea se face într-un număr fix de pași (1000 de iterări), care nu depinde de valoarea lui k, dar poate fi mărită peste 1000 dacă este necesar.

Determinati numerele a,b,c,d pentru care vectorul x=(a,5,b,c,d,3,6) este solutie a problemei celor “n” dame, pentru n=7  
**Solutie: a=2, b=7, c=4, d=1**

Cate astfel de solutii exista? **DOAR 1**



| index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Numbers | 22 | 14 | 16 | 8 | 9 | 23 | 6 | 12 | 14 | 21 | 16 | 11 | 8 | 13 |

| L[14]=1  L[13]=1  L[12]=2  L[11]=3  L[10]=4  L[9]=3  L[8]=3  L[7]=1  L[6]=5  L[5]=2  L[4]=2  L[3]=4  L[2]=4  L[1]=5 | P[13]=-1  P[12]=13  P[11]=12  P[10]=11  P[9]=12  P[8]=12  P[7]=-1  P[6]=10  P[5]=7  P[4]=7  P[3]=8  P[2]=8  P[1]=10 |
| --- | --- |

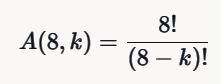
| **Descrieți și implementati** in Java un algoritm pentru generarea tuturor cuvintelor de lungime "k" care pot fi formate cu litere distincte din mulțimea { a,b,c,d,e,f,g,h,i } astel incat sa nu contina 2 vocale alaturate. Evaluati complexitatea și eficienta acestui program. |
| --- |

**Generare combinatorică** – Folosim backtracking pentru a genera toate cuvintele de lungime k cu litere distincte din mulțimea **{a, b, c, d, e, f, g, h, i}.**

**Pentru a reprezenta literele folosim numere “int” { 1, 2, 3, 4, 5, 6, 7, 8, 9 }**

**Evitarea vocalelor consecutive** – La fiecare pas al construcției unui cuvânt, verificăm dacă adăugarea unei noi vocale ar crea o secvență de două vocale consecutive.

**Explorare cu backtracking** – Explorăm toate posibilitățile, eliminând ramurile invalide imediat ce detectăm două vocale consecutive.



| import java.util.Scanner; //STANDARD  class Main { **//Am notat pe “k” cu “n”. Deci pentru m=9 si n=3, nr soluții care le avem sunt Aranj de m luate**  **//cate n - solutiile care incalca (fa calcul pe foaie, deci cam maxim 500-1000 solutii)**  public static void ARANJAMENTE\_STANDARD(int m, int n){  int k=1;  int[] x;  x= new int[m+1];  x[1]=0;  while(k>0) **//dacă vrei mai multe comments, du-te înapoi la primele pagini**  {  if(x[k]<m){  x[k]++;  if(VALID(x,k))  {  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=0;  }  }  }  else k--;  }  }  public static boolean VALID(int[] x, int k){  for(int i=1;i<=k-1;i++){  if(x[k]==x[i])return false; **//prima conditie verifica ca nu se repeta literele (ARANJ FARA REPETITIE)**  }  for(int i=1; i<=k-1; i++){ **//al 2-lea for verifica ca toate literele x[i] nu sunt vocale in acelasi timp cand litera x[i+1] de langa ea este tot consoana. daca ambele sunt vocale, solutia este invalida**  if(x[i]==1|| x[i]==5 || x[i]==7){  if(x[i+1]==1 || x[i+1]==5 || x[i+1]==7)return false;  }  }  return true;  **//Are 426 de solutii. (dar explica de ce are atatea, aratand formula aranjamentelor)**  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();}  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  System.out.print("m=");  m=cin.nextInt();  ARANJAMENTE\_STANDARD(m,n);  }} |
| --- |

* **Complexitate și eficiență:** Generarea **toate aranjamente** de lungime **k** din **9** elemente este **O(A(9, k)) = O(9! / (9-k)!)**.
* Validarea **se face în O(k)**, deci per total avem **O(A(9, k) \* k)**, ceea ce este destul de eficient pentru valori rezonabile de **k**.

**Suma Maxima/Minima Mod De Rezolvare:**

* **Inițializarea ultimei linii**: La început, copiem valorile din ultima linie a matricei în matricea auxiliară S. Acestea reprezintă suma maximă de la fiecare element de pe ultima linie, care este chiar valoarea acelui element, deoarece nu există linii sub acestea.
* **Calcularea valorilor pentru liniile anterioare**: Apoi, pentru fiecare linie de la penultima linie până la prima, calculăm suma maximă pe care o putem obține pentru fiecare element de pe acea linie. Aceasta se face adunând valoarea curentă din matricea originală cu suma maximă obținută din linia imediat următoare, din una dintre cele trei direcții posibile:
  + **Jos** (direct sub elementul curent),
  + **Jos-stânga** (dacă există),
  + **Jos-dreapta** (dacă există).
* Astfel, pentru fiecare element de pe linia curentă, alegem suma maximă posibilă dintre cele trei direcții și adăugăm valoarea elementului curent.
* **Suma maximă**: După ce am completat matricea S, suma maximă a drumului va fi găsită pe prima linie a matricei S. Aceasta poate fi obținută ca fiind cel mai mare element de pe prima linie.

**Reconstruirea drumului**: După ce am calculat matricea S, putem reconstrui drumul de sumă maximă. Începem cu cel mai mare element de pe prima linie a matricei S și, pentru fiecare pas următor, alegem elementul de pe linia următoare care contribuie la suma maximă. Mai precis, pentru fiecare element S[i][j], scădem valoarea corespunzătoare din matricea originală M[i][j] (pentru a nu mai alege acest element din nou) și apoi căutăm cel mai mare element posibil pe linia următoare (S[i+1][j], S[i+1][j-1] dacă există, sau S[i+1][j+1] dacă există).

### **Pentru minim:**

În loc să căutăm suma maximă pe fiecare pas, căutăm suma minimă. În loc de max(S[i+1][j], S[i+1][j-1], S[i+1][j+1]), alegem **minimul** dintre valorile din direcțiile disponibile: min(S[i+1][j], S[i+1][j-1],S[i+1][j+1] (dacă există)).

Reconstruirea drumului va presupune alegerea celei mai mici valori posibile de pe linia următoare și scăderea valorii curente din matricea originală pentru a nu alege același element din nou.

### **Problemă de acoperire**

* Se împarte suprafața pătrată în patru pătrate egale, formându-se patru suprafețe pătrate cu lungimea laturii 2^(k−1).
* - Una dintre cele patru suprafețe are deja un pătrat unitar acoperit.
* - Acoperim cu o placă trei pătrate unitare din centrul suprafeței mari astfel încât să acoperim câte un pătrat unitar și din celelalte trei suprafețe.
* Se reia algoritmul.

### **4.2 Problema turnurilor din Hanoi**

Orice algoritm care rezolvă problema turnurilor din Hanoi necesită cel puțin (2^n) - 1 mutări.

Se cere mutarea discurilor pe tija 2, în aceeași ordine, utilizând tija 3 și respectând următoarele reguli:

* Mutăm un singur disc la un moment dat.
* Nu putem așeza un disc mai mare peste unul mai mic.
* Folosim tija 3 ca suport temporar.

Algoritm pas cu pas pentru n discuri

Dacă avem **n discuri**, le mutăm astfel:

1. **Mutăm n−1 discuri de pe Tija 1 pe Tija 3** (folosind Tija 2 ca ajutor).
2. **Mutăm discul cel mai mare de pe Tija 1 pe Tija 2**.
3. **Mutăm cele n−1 discuri de pe Tija 3 pe Tija 2** (folosind Tija 1 ca ajutor).

**Exemplu pentru 3 discuri**

📍 Tijele: **1 (inițial), 2 (destinație), 3 (auxiliar)**

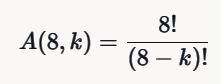
**Pas cu pas:**

1️⃣ Mutăm discul 1 de pe **1 → 2**2️⃣ Mutăm discul 2 de pe **1 → 3**3️⃣ Mutăm discul 1 de pe **3 → 3**4️⃣ Mutăm discul 3 de pe **1 → 2**5️⃣ Mutăm discul 1 de pe **2 → 1**6️⃣ Mutăm discul 2 de pe **2 → 2**7️⃣ Mutăm discul 1 de pe **1 → 2**

✅ **Toate discurile sunt acum pe Tija 2, în aceeași ordine!**

Algoritmul în care generam toate cuvintele lungime k care pot fi formate cu cifre distincte din mulțimea  
{0, 1, 2, 3, 4, 5, 6, 7} astfel incat orice 2 litere alaturate sa nu fie ambele pare, sau ambele impare.

**Complexitatea:**

Avem un total de 8 cifre și dorim să generăm aranjamente de lungime k. Numărul total de aranjamente fără repetitie este dat de:  


Deci, complexitatea de generare a tuturor aranjamentelor este **O(8! / (8-k)!)**.

**Eficiența:**

* Pt valori mici “k”: Algoritmul va fi eficient (k ≤ 4), deoarece numărul de aranjamente crește rapid pentru valori mari ale lui k.

| import java.util.Scanner; //STANDARD **//n=3 //m=8 (am renotat “k” cu “n” in acest algoritm)**  class Main { **//dacă vrei mai multe comments, du-te înapoi la primele pagini**  public static void ARANJAMENTE\_STANDARD(int m, int n){  int k=1;  int[] x;  x= new int[m+1];  x[1]=0;  while(k>0)  {  if(x[k]<m){  x[k]++;  if(VALID(x,k))  {  if(k==n)AFISARE(x,n);  else{  k++;  x[k]=0;  }  }  }  else k--;  }  }  public static boolean VALID(int[] x, int k){  for(int i=1;i<=k-1;i++){  if(x[k]==x[i])return false; //prima condiție verifica ca nu se repeta literele  }  for(int i=1; i<=k-1; i++){  if(x[i]%2==0 && x[i+1]%2==0)return false; **//daca x[i] și următorul x[i+1] ambele pare**  if(x[i]%2==1 && x[i+1]%2==1)return false; **//sau ambele alaturate sunt impare, atunci sunt invalide**  }  return true;  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i]-1 + " "); **//aici am pus un "-1" ca sa scada șirul (1...8) la (0...7)**  System.out.println(); **//pastrand functionalitatea algoritmului identica ca la laborator**  }  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  System.out.print("m=");  m=cin.nextInt();  ARANJAMENTE\_STANDARD(m,n);  }  } |
| --- |

Exemple solutii la algoritm: 0 1 2, 0 1 4 , 0 1 6, 0 3 2, 0 3 4…..

Are 96 de solutii. (dar explica de ce are atatea)

| **Algoritm Java pt generare toate scrieri S ca o suma de nr impare, in care nu conteaza ordinea dintre termeni.**  import java.util.Scanner;  class Main {  public static Scanner cin;  public static int[] nr;  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(nr[x[i]] + " ");  System.out.println();  }  public static void CITESTE(int[] t, int lungime){  for(int i=1; i<=lungime; i++)t[i] = cin.nextInt();  }  public static void PARTITII(int m, int n){  int k=1; **//pt mai multe comments, mergi la primele pagini**  int[] x = new int[n+1]; // Array pentru a stoca partitiile curente  x[1]=0;  int s=0; // Suma parțială  while(k>0){  if(k<=n-1) { // Dacă nu am ajuns la ultima parte  int temp= (m-s)/(n-k+1); // Calculează val maxima posibila  if(x[k]<Math.abs(temp)){ // Dacă val curenta < limita  x[k]++;  s=s+x[k];  k++;  x[k]=x[k-1]-1;  }  else{  k--;  s=s-x[k];  }  }  else{  x[n]=m-s;  if(VALID(x)) AFISARE(x,n);  k--;  s=s-x[k];  }  }  }  public static boolean VALID(int[] x){  for(int i=1; i<x.length; i++){  if(x[i]%2==0) return false; //dacă am găsit 1 număr par in secvența, este invalid  }  return true;  }  public static void main(String[] args) {  int m=10,n=10; //asumam valorile din curs  cin = new Scanner(System.in);  System.out.println("Citeste " + m + " numere oarecare: ");  nr = new int[m+1];  CITESTE(nr,m);//nr oarecare  for(int i=1; i<=n; i++){  System.out.println("Partitii de " + i + " numere: ");  PARTITII(m,i);  }  }  }  The number of possible combinations is **C(m, n) = m! / (n!(m-n)!)**, so the algorithm runs in **O(C(m, n))**. | **Evaluati complexitatea si eficienta acestui algoritm:**  CITESTE() - O(m)  Functia PARTITII():  **Algoritm de generare a partițiilor**:   * Partitiile sunt generate printr-un algoritm de backtracking cu o abordare bazată pe combinări, în care se explorează diferite secvențe de valori care sumate să fie egale cu m, iar numărul de părți nu depășește n. * În fiecare iterație, k este incrementat până când se ajunge la ultima parte (k == n). * Este o abordare de backtracking, dar cu condiții de validare pentru a asigura că fiecare parte este impară (valoare determinată de funcția VALID). * Generarea partițiilor este o problemă combinatorială, și în funcție de valoarea lui **m** și **n**, numărul de partiții posibile poate fi destul de mare. * Algoritmul generează toate combinațiile de numere care formează suma **m** în **n** părți. * În cel mai rău caz, numărul de combinații (partiții) este dat de **numărul de partiții ale unui număr** (denumită funcția de partiționare), care pentru valori mari ale **m** și **n** poate fi exponențial în mărimea inputului. * Estimarea aproximativă a complexității pentru generarea partițiilor este **O(2^m)** în cel mai rău caz, dar depinde foarte mult de condițiile de validare impuse de algoritm.   **Rezultat Printat**:  Citeste 10 numere oarecare:  1 2 3 4 5 6 7 8 9 10  Partitii de 1 numere:  Partitii de 2 numere:  1 9  3 7  5 5  Partitii de 3 numere:  Partitii de 4 numere:  1 1 1 7  1 1 3 5  1 3 3 3  Partitii de 5 numere:  Partitii de 6 numere:  1 1 1 1 1 5  1 1 1 1 3 3  Partitii de 7 numere:  Partitii de 8 numere:  1 1 1 1 1 1 1 3  Partitii de 9 numere:  Partitii de 10 numere:  1 1 1 1 1 1 1 1 1 1  => 10 solutii  Nr exact depinde de m și de numărul de părți n. Dacă m este mic și condițiile sunt stricte (nr impare), nr soluții va fi mic. Dacă m este mai mare și se permit mai multe părți, numărul soluțiilor va crește, dar va fi mereu < decât nr total de partiții al lui m. |
| --- | --- |

Implement Java algoritm pt generarea tuturor vectorilor de lungime k care pot fii formati din multimea

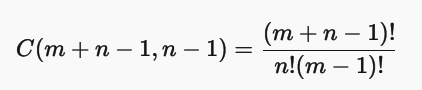
{1,2,3,4,5,6,7,8} astfel incat orice 2 elemente alaturate sa aiba suma mai mare sau egala cu 9.

| import java.util.Scanner;  class Main {  public static void ARANJAMENTE\_CU\_REPETITII(int m, int n){  int k=1;  int[] x;  x= new int[n+1];  x[1]=0;  while (k > 0) {  if (x[k] < m) {  x[k]++;  if (k == n){  if(VALID(x)) AFISARE(x, k);  }  else {  k++;  x[k] = 0; // Reset  }  } else {  k--; // Backtrack  }  }  }  public static boolean VALID(int[] x) {  int suma = 0;  for (int i = 1; i < x.length; i++) {  suma += x[i];  if(suma >= 9) return true;  }  return false;  }  public static void AFISARE(int[] x, int n){  for(int i=1; i<=n; i++)System.out.print(x[i] + " ");  System.out.println();  }  public static void main(String[] args) {  int n, m;  Scanner cin = new Scanner(System.in);  System.out.print("n=");  n=cin.nextInt();  System.out.print("m=");  m=cin.nextInt();  ARANJAMENTE\_CU\_REPETITII(m,n);}} | n=2 (am notat “k” cu “n” in acest program)  m=8  //Solutii  1 8  2 7  2 8  3 6  3 7  3 8  4 5  4 6  4 7  4 8  5 4  5 5  5 6  5 7  5 8  6 3  6 4  6 5  6 6  6 7  6 8  7 2  7 3  7 4  7 5  7 6  7 7  7 8  8 1  8 2  8 3  8 4  8 5  8 6  8 7  8 8  **Nr Solutii: 38**  Fiecare poziție din aranjament (lungime n) poate fi ocupată de oricare dintre cele m elemente.  Numărul total de aranjamente posibile este: **m^n**  În **cel mai rău caz**, toate aranjamentele sunt generate și validate, deci complexitatea este:  **O(n\*m^n)** |
| --- | --- |

**Compuneri (complexitatea)**

Acest algoritm generează **compuneri oarecare** ale unui număr mmm în nnn termeni, ceea ce înseamnă că trebuie să găsim toate modurile în care m poate fi scris ca sumă de n numere naturale (inclusiv zero).

Numărul total de compuneri ale lui mmm în nnn termeni este dat de formula combinatorică:



Astfel, **complexitatea totală a algoritmului este**: **O(n\*C(m+n−1,n−1))**

Aceasta este **exponențială** în raport cu n și m, ceea ce face algoritmul ineficient pentru valori mari ale acestora.

**Permutari (complexitatea)**

* Numărul total de permutări pentru nnn elemente este P(n)=n!P(n) = n!P(n)=n!.
* Fiecare permutare necesită verificări de validitate și posibile afișări.

În cel mai rău caz, pentru fiecare nivel kkk, trebuie să verificăm toate cele k−1k-1k−1 poziții anterioare pentru a verifica validitatea.

* Fiecare validare durează O(n) în cel mai rău caz.
* Se fac O(n!) apeluri datorită numărului total de permutări.
* Deci, complexitatea totală este **O(n! \* n)**.
* În practică, se poate aproxima drept **O(n!)**, deoarece factorialul crește mult mai repede decât termenul suplimentar O(n).