

Divide et impera și probleme diverse cu recursivitate

Metoda de programare DIVIDE ET IMPERA.

Pornim de la următorul program:

```
#include <iostream>
using namespace std;
int v[1001], i, n;

int suma(int st, int dr) {
    if (st == dr) {
        return v[st];
    } else {
        int mid = (st + dr)/2;
        return suma(st, mid) + suma(mid+1, dr);
    }
}

int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
    cout<<suma(1, n);
}
```

Să urmărim și scrierea echivalentă, dar într-un mod mai didactic:

```
#include <iostream>
using namespace std;
int v[1001], i, n;

int suma(int st, int dr) {
    if (st == dr) {
        return v[st];
    } else {
        int mid = (st + dr)/2;
        int sumaStanga = suma(st, mid);
        int sumaDreapta = suma(mid+1, dr);
        return sumaStanga + sumaDreapta;
    }
}

int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
    cout<<suma(1, n);
}
```

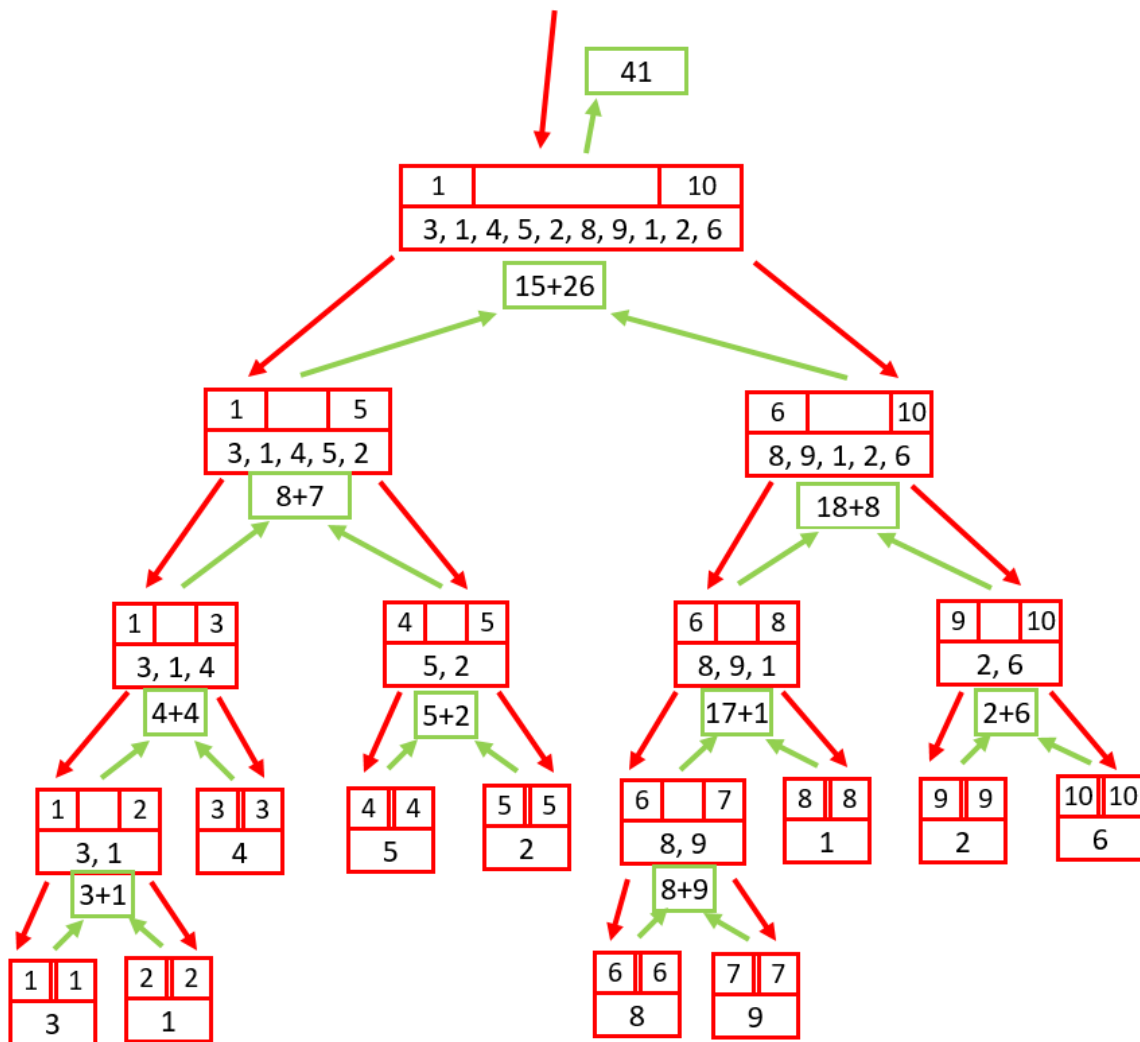
Soluțiile propuse realizează suma elementelor dintr-un tablou unidimensional, aflate într-un interval dat de indici.

Funcția recursivă este una operand, primește ca paraetri doi indici și are ca efect returnarea sumei elementelor aflate între cei doi indici inclusiv. Observăm că pentru a obține suma elementelor din tot șirul declanșăm în main un apel `suma(1, n)`.

Principiul de funcționare a subprogramului este:

- Dacă secvența este suficient de scurtă, în acest caz dintr-un singur element (`st == dr`), rezultatul se obține direct, `return v[st]`.
- Dacă secvența are cel puțin două elemente, o descompunem în două (de lungimi cât mai apropiate), iar după ce obținem rezultatele din cele două părți le adunăm și pe baza lor găsim suma din secvența dată.

Fie $n=10$, și elementele memorate pe poziții de la 1 la 10 în vectorul `v`: 3, 1, 4, 5, 2, 8, 9, 1, 2, 6.



Pentru o bună înțelegere a modului de funcționare pentru funcția recursivă de mai sus trebuie analizat atent și cu răbdare arborele de desfășurare a apelurilor recursive de mai sus.

- Elementele simbolizate cu roșu reprezintă evenimente de la coborârea în recursivitate iar cele verzi de la întoarcere.
- Tabelele roșii au înainte pe prima linie valorile lui `st` și `dr` iar pe linia a doua valorile din vector dintre pozițiile `st` și `dr`.

- Căsuțele verzi conțin valorile cu care se revine din autoapeluri și operația care se face între ele în scopul obținerii rezultatului apelului curent. Observăm că din punct de vedere cronologic calculele se fac la întoarcere și de jos în sus.

Un algoritm divide et impera se aplică la problemele care, pot fi descompuse în altele mai simple de același tip, cele mai simple și ele să se poată descompune la fel, până când se ajunge la probleme suficient de simple care să poată fi rezolvate direct. Apoi, pe baza soluțiilor subproblemelor, să se poată obține soluția problemei descompuse.

Întrucât subproblemele sunt de același tip cu problema descompusă, putem folosi recursivitatea. Cazul că problema a devenit suficient de simplă încât poate fi rezolvată direct este chiar situația când nu se mai face autoapel în recursivitate.

Iată într-un pseudocod schema de rezolvare a unei probleme prin divide et impera.

rezolvă(problema)

- *dacă problema este simplă, o rezolvăm direct*
- *dacă nu*
 - descompunem** problema în subprobleme, de același fel (de regulă două)
 - rezolvăm subproblemele** (de regulă recursiv)
 - combinăm** rezultatele subproblemelor în scopul obținerii rezultatului problemei inițiale

Pașii descriși anterior folosind comentarii în a doua soluție prezentată mai sus sunt:

```
#include <iostream>
using namespace std;
int v[1001], i, n;

int suma(int st, int dr) {
    if (st == dr) {    /// subproblema este simplă
                      /// (intervalul este de lungime 1)
                      /// o rezolv direct, fără descompunere
                      /// (suma pe un astfel de interval este chiar
                      /// valoarea din vector de pe acea poziție)
        return v[st];
    } else {
        /// descompun intervalul în cele două jumătăți ale sale
        /// după modul de scriere a formulei, dacă numărul de elemente
        /// dintre st și dr este impar, în partea din stânga
        /// va fi cu un element mai mult decât în cea din dreapta
        int mid = (st + dr)/2;

        /// rezolv cele două subprobleme prin autoapeluri recursive
        /// în cele două subintervale în care am realizat descompunerea
        int sumaStanga = suma(st, mid);
        int sumaDreapta = suma(mid+1, dr);

        /// combin rezultatele subproblemelor, adică suma pe
        /// intervalul de rezolvat o obținem prin adunarea rezultatelor
        /// subproblemelor (întoarse de autoapeluri)
```

```

        return sumaStanga + sumaDreapta;
    }
}

int main () {
    cin>>n;
    for (i=1;i<=n;i++)
        cin>>v[i];
    cout<<suma(1, n);
}

```

Așa cum am descris mai sus, de foarte multe ori descompunerea se face în două subprobleme, iar dacă natura problemei este de a procesa un interval de indici, pentru calculul locului unde se face împărțirea se calculează media aritmetică a indicilor.

Trebuie avut grijă la cazul când suma valorilor st și dr produce overflow pe tipul de date pe care se lucrează, în acest caz folosindu-se scrierea: $st + (dr - st) / 2$.

Observăm că prin modul de scriere a formulei, și prin modul de stabilire a capetelor intervalelor de la autoapel, se ajunge mereu la intervale de lungime 1 (chiar la toate cele n intervale formate din câte un element), acestea fiind frunzele arborelui de autoapeluri.

Timpul de executare al unui apel pe un interval de lungime n se calculează astfel:

- la nivelul frunzelor se fac $n/2$ adunări
- un nivel mai sus se fac $n/4$ adunări
- ...

$n/2 + n/4 + \dots$ este valoare maxim egală cu n . Așadar timpul de calcul este de ordinul lungimii intervalului.

Probleme rezolvate

1. Folosind un procedeu divide et impera, să se decidă dacă un vector de numere naturale conține cel puțin o valoare impară (pbinfo.ro, #1148).

Soluție

- Definim problema de rezolvat astfel: $verificare(st, dr)$ = funcție operand care realizează testul pentru elementele din intervalul de indici st, dr , inclusiv și care returnează 1 dacă se găsește un element impar acolo și 0 în caz contrar.
- Problema este simplă când $st==dr$ și acolo returnăm 1 dacă $v[st]$ este impar și 0 în caz contrar.
- În etapa de combinare a soluțiilor, returnăm 1 dacă cel puțin o subproblemă returnează 1 (pentru un cod mai compact putem folosi operatorul $||$ între rezultatele celor două autoapeluri).

```

#include <iostream>
using namespace std;
int n;
int v[1001];

int exista(int st, int dr) {
    if (st == dr)
        if (v[st] % 2 == 0)
            return 0;
        else
            return 1;
    else {

```

```

        int mid = (st+dr)/2;
        return exista(st, mid) || exista(mid+1, dr);
    }
}

int main () {
    cin>>n;
    for (int i=1;i<=n;i++)
        cin>>v[i];
    cout<<(exista(1, n)==1 ? "DA" : "NU");
}

```

2. Să se verifice dacă elementele unui tablou unidimensional sunt ordonate crescător (pbinfo.ro, #1152).

Soluție

- Definim problema de rezolvat astfel: $f(st, dr)$ = funcție operand care realizează testul pentru elementele din intervalul de indici st, dr , inclusiv și care returnează 1 dacă ele sunt în ordine crescătoare și 0 în caz contrar.
- Problema este simplă când $st==dr$ și acolo returnăm 1 (considerăm că o secvență formată dintr-un singur element este ordonată crescător).
- În etapa de combinare a soluțiilor, prima idee este de a returna 1 dacă ambele subprobleme dau 1 (cele două secvențe sunt ordonate crescător). Nu este suficient întrucât pot să apară situații de forma: 1, 2, 4, 2, 3, 5 (ambele jumătăți sunt ordonate, dar totuși nu este tot șirul ordonat). Mai este necesar și ca ultimul element din secvența stângă să fie mai mic sau egal decât primul din secvența dreaptă.

```

#include <iostream>
using namespace std;

int v[10010];
int n, i, suma;

int f(int st, int dr)
    if (st==dr){
        return 1;
    }
    else{
        int mid=(st+dr)/2;
        return f(st, mid) && f(mid+1, dr) && v[mid]<=v[mid+1];
    }
}

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

    if (f(1, n) == 1)
        cout<<"DA";
    else
        cout<<"NU";

    return 0;
}

```

3. Căutare. Folosind o strategie divide et impera, trebuie să decidem dacă o valoare dată x se află printre elementele unui șir.

Soluție

- Definim problema de rezolvat astfel: $f(st, dr, x)$ = funcție operand care realizează testul pentru elementele din intervalul de indici st, dr , inclusiv și care returnează 1 dacă cel puțin unul dintre ele este egal cu x și 0 în caz contrar.
- Problema este simplă când $st==dr$ și acolo returnăm 1 dacă valoarea este chiar x și 0 în caz contrar.
- În etapa de combinare a soluțiilor se face $||$ între rezultatele subproblemelor.

```
int f(int st, int dr, int x) {
    if (st==dr){
        if (v[st]==x){
            return 1;
        }
        else
            return 0;
    }
    else{
        int mid=(st+dr)/2;
        return f(st, mid, val)||f(mid+1, dr, val);
    }
}
```

4. **Căutare binară.** Dorim să decidem dacă o valoare dată x se găsește printre elementele unui șir, dar se cunoaște că șirul este sortat crescător.

Soluție

Vom folosi o strategie similară, de a împărți șirul în două părți, dar de această dată profităm de faptul că are sens să continuăm căutarea doar într-o parte, făcând astfel ca timpul de căutare să fie de ordin logaritmic.

- Semnificația antetului: $cauta(st, dr, x)$: returnează 1 dacă vreunul dintre elementele vectorului cu indici de la st la dr este egal cu x .
- La această problemă vom verifica efectiv dacă x este egal cu elementul de pe poziția calculată mid , iar dacă găsim egalitate returnăm 1.
- Astfel, autoapelurile se fac fie între st și $mid-1$ fie între $mid+1$ și dr .
- Lucrând ca mai sus, cunoaștem de la analiza variantei iterative că se poate ajunge la situații când $st > dr$, corespunzătoare cazului că x nu apare.

```
int cauta(int st, int dr, int x) {
    if (st > dr)
        return 0;
    int mid = (st+dr)/2;
    if (v[mid] == x)
        return 1;
    if (x > v[mid])
        return cauta(mid+1, dr, x);
    else
        return cauta(st, mid-1, x);
}
```

Două observații legate de rezolvarea de mai sus:

- Folosind `return` când identificăm clar cazurile de oprire evităm folosirea de mai multe ori a lui `else` (iar pe ramura `else` prezentă se ajunge dacă x este strict mai mic decât $v[mid]$).
- Dacă înlocuim `return 1` cu `return mid` facem ca funcția să ofere o poziție pe care se află x în cazul în care acesta apare în șir sau 0 în caz contrar.

5. Sortarea prin interclasare (Merge Sort)

Să se ordoneze crescător elementele unui tablou unidimensional. Prezentăm mai jos codul apoi o analiză detaliată.

```
#include <iostream>
using namespace std;
int i,n,v[100010], w[100010];

/// În arborele de recursivitate se fac log2n nivele
/// și costul de pe fiecare nivel este n deoarece
/// pe fiecare nivel sunt mai multe interclasari dar
/// un element al sirului participa la exact una dintre ele.

int interclaseaza(int st, int mid, int dr) {
    /// Întrucât primul element din dreapta s-ar putea
    /// să fie mai mic decât primul din stanga
    /// am risca să suprascriem în stanga.
    /// Pentru a evita asta vom folosi un vector auxiliar în care facem
    /// interclasarea și apoi copiem în v.
    int i = st;
    int j = mid+1;
    int k = st-1; /// Inițializăm k cu indicele anterior primului loc
                  /// unde punem în w pentru că la fiecare plasare
                  /// de element nou mai întâi se mărește k
    while (i <= mid && j <= dr) {
        if (v[i] < v[j]) {
            w[++k] = v[i++];
        } else {
            w[++k] = v[j++];
        }
    }
    for (;i<=mid;i++)
        w[++k] = v[i];
    for (;j<=dr;j++)
        w[++k] = v[j];
    /// Deocamdata am interclasat in w
    ////deci doar mutăm in v între aceiași indici.
    for (int i=st;i<=dr;i++)
        v[i] = w[i];
}

void sorteaza(int st, int dr) {
    if (st < dr) {

        int mid = (st + dr)/2;
        sorteaza(st, mid);
        sorteaza(mid+1, dr);

        interclaseaza(st, mid, dr);
        /// La apelul de mai sus gândesc:
        /// functia primeste sortată secvența din v
    }
}
```

```

        /// dintre indicii st și mid, de asemenea
        /// sortată pe cea dintre indicii mid+1, dr
        /// și va lăsa sortată toată reuniunea lor
        /// între indicii st și dr.

    }/// dacă st == dr înseamnă că e de sortat o secvență de lungime 1
    /// și nu am nimic de făcut
}

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

    sorteaza(1, n);

    for (i=1; i<=n; i++) {
        cout << v[i] << " ";
    }
    return 0;
}

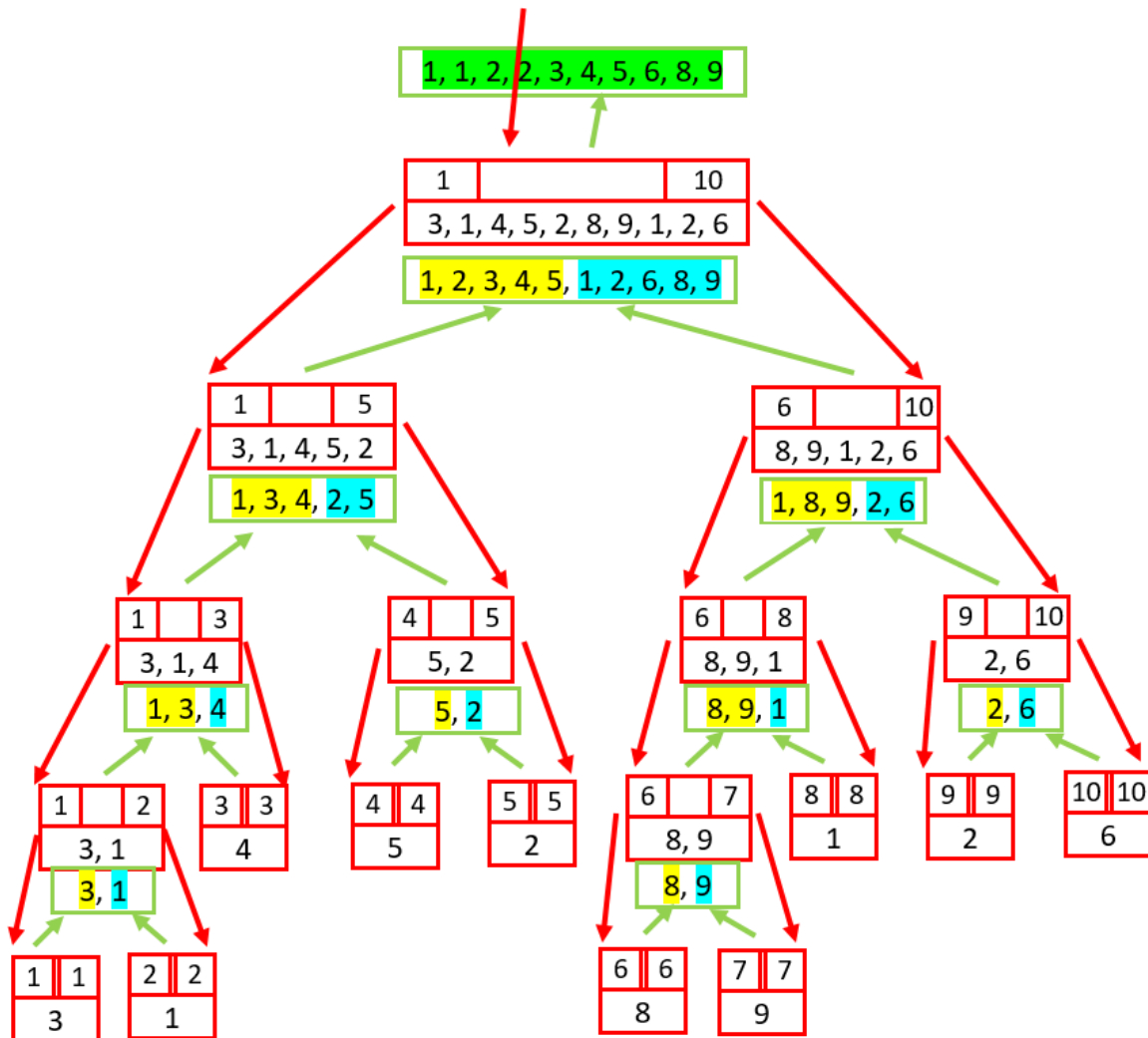
```

- Stabilim antetul funcției recursive `sorteaza(st, dr)` cu semnificația: ordonează crescător elementele vectorului aflate pe poziții cuprinse între `st` și `dr` inclusiv. Considerăm că lucrăm cu vectorul global, iar de la un autoapel la altul doar stabilim intervalul de indici pe care îl procesăm.
- Descompunem problema curentă în două subprobleme pe principiul folosit la aplicațiile anterioare.
- Dacă se ajunge la o secvență cu un singur element (`st == dr`) șirul reprezentat de ea este sortat așa că nu facem nimic (funcția `sorteaza` este procedurală și nu facem nimic pe ramura respectivă).
- La revenire considerăm că venim de jos în sus cu cele două secvențe sortate (și când se pornește de la frunze acestea reprezintă, cum am spus, secvențe sortate, deci pornim corect) așa că rămâne să le reunim pentru a obține sortată soată secvența.
- Pentru a pune împreună sortate cele două jumătăți vom folosi un algoritm de interclasare optimă, despre care știm că are timp de executare de ordinul numărului total de elemente de interclasat. Funcția `interclaseaza(st, mid, dr)` este cea care realizează această etapă, de combinare. Ea primește sortată secvența de la `st` la `mid` și tot sortată pe cea de la `mid+1` la `dr` urmând să le reunească sortate. Este esențial să observăm că nu putem lucra direct în vectorul `v` (s-ar putea ca primul element din dreapta să fie mai mic decât primul din stânga și l-ar suprascrise). Așadar folosim un vector auxiliar, `w` în care construim rezultatul interclasării și apoi îi copiem elementele, în ordinea sortată, în `v`. Este foarte important să declarăm acest vector suplimentar global pentru a evita alocarea sa în memorie la fiecare inițiere de autoapel, lucru foarte costisitor.
- La fiecare nivel al arborelui se fac așadar interclasări și observăm că pe fiecare nivel orice element al vectorului apare o singură dată și, în plus, el participă la exact o interclasare. Concluzia este că pe fiecare nivel timpul total de calcul este de ordin n . Pe de altă parte, avem număr de nivele de ordin $\log_2 n$. Obținem așadar un algoritm cu timp de calcul $n \log_2 n$, chiar dacă este nevoie de spațiu dublu de memorie.

Pentru a înțelege mai bine modul de funcționare a algoritmului, este utilă și urmărirea modului în care se avansează pe arborele de autoapeluri generat de un apel inițial `sorteaza(1, n)`;

- Cu roșu am simbolizat ce se întâmplă la împărțirea în subprobleme, deci când se coboară în arborele autoapelurilor.

- Cu verde simbolizăm ce se întâmplă la revenire. Se observă acum cum cele două subsecvențe revin ordonate și apoi ele urmează să fie interclasate.



6. **Sortarea rapidă (Quick Sort).** De asemenea, considerăm că avem de sortat crescător un șir cu n elemente.

Soluție

Funcția recursivă o vom scrie cu aceeași semnificație: `sorteaza(st, dr)` înseamnă: ordonează crescător elementele din vector aflate între pozițiile `st` și `dr` inclusiv.

Vom folosi următoarea strategie: plasăm elementul aflat inițial pe poziția `st` în secvență la locul său dacă secvența ar fi sortată și totodată lăsăm toate elementele din față sa mai mici decât el și pe toate de după el mai mari decât el. Observăm că dacă am realiza asta, trebuie sortate disjunct atât elementele din față sa cât și, similar, cele de după el.

Exemplu:

Dacă secvența de sortat ar fi formată din elementele: 5 1 8 9 3 4 6 2, elementul pe care l-am fixa la locul său ar fi 5, iar 1 2 3 4 ar trebui să ajungă în față sa iar 6 8 9 după el, adică am transforma secvența în una de forma:

4 3 1 2 **5** 8 6 9. Nu neapărat elementele din fața lui 5 ar fi în această ordine, dar important este să fie toate cele mai mici decât 5 în fața sa.

Să presupunem că avem o funcție `poz(st, dr)` care face cele descrise mai sus pentru secvența de elemente dintre indicii `st` și `dr` și care în plus returnează poziția pe care a dus elementul aflat inițial pe poziția `st` (să îl numim pivot).

Funcția principală ar fi:

```
void sorteaza(int st, int dr) {
    if (st < dr) {
        int p = poz(st, dr);
        /// aceasta funcție va duce primul element din intervalul de indici
        /// st,dr la locul lui în interval și în plus tot ce e în stânga lui
        /// va fi mai mic decât el și tot ce e în dreapta lui va fi
        /// mai mare ca el
        sorteaza(st, p-1);
        sorteaza(p+1, dr);
    }
}
```

Rămâne acum să discutăm modul de realizare a funcției `poz`.

- Pe parcursul operației, vom gestiona o pereche de indici i și j , cu $i \leq j$ astfel: pivotul este pe una dintre pozițiile i și j , elementele din stânga poziției i să fie mai mici decât pivotul, elementele din dreapta poziției j să fie mai mari decât pivotul. De exemplu, pe parcurs am putea să ne aflăm în starea:

		i			j		
3	1	5	4	3	7	6	9

Inițializarea o facem astfel: $i=st$; $j=dr$; Observăm că ea este o situație particulară dar care respectă restricțiile.

La pasul curent comparăm $v[i]$ cu $v[j]$. Întrucât sunt în ordinea cerută nu vom interschimba, dar vom scădea pe j cu 1, i rămânând pe loc (acolo unde se află pivotul).

		i		j			
3	1	5	4	3	7	6	9

Acum $v[i] > v[j]$, vom interschimba $v[i]$ cu $v[j]$, dar acum pivotul ajunge pe poziția j , deci va trebui să nu mai modificăm pe j , dar vom crește i (valoarea de pe vechea poziție a lui i este acum mai mică decât pivotul).

Analizând atent, generalizăm astfel:

- Comparăm $v[i]$ cu $v[j]$, iar dacă sunt în ordinea cerută le lăsăm așa și continuăm cu modul de avansare setat curent. Dacă le interschimbăm, facem și comutarea în în celălalt mod de avansare.
- Observăm că avansarea se face prin incrementari de forma $(i+di, j+dj)$ unde perechea (di, dj) este fie $(0, -1)$ – (adică i stă și j scade cu 1), fie $(1, 0)$ – (adică i crește cu 1 și j stă).
- O modalitate compactă de a comuta dintr-un mod în altul este următoarea:

```
aux = di;
di = -dj;
dj = -aux;
```

Secvența este asemănătoare cu aceea de la interschimbarea clasică, dar la ultimele două atribuiri expresia din dreapta se scrie cu minus. La o analiză pas cu pas se observă că se realizează corect comutarea în ambele sensuri. Astfel funcția `poz` arată astfel:

```
int poz(int st, int dr) {
    int di = 0;
    int dj = -1;
    int i = st;
    int j = dr;
    int aux;
    while (i < j) {
        if (v[i] > v[j]) {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;

            aux = di;
            di = -dj;
            dj = -aux;
        }
        i += di;
        j += dj;
    }
    return i;
}
```

Valoarea comună la care ajung `i` și `j` este aceea cu poziția finală a pivotului, fiind cea pe care o și returnăm.

Să analizăm eficiența algoritmului. Funcția `poz` face număr de pași de ordinul diferenței `dr-st` (adică dat de numărul de elemente din secvența `st, dr`). Acest lucru se probează ușor observând că la fiecare pas fie crește `i`, fie scade `j`.

În funcția recursivă, la fiecare autoapel se face deci număr de pași de ordin `n`. Rămâne deci de analizat care este numărul de autoapeluri. Cazul favorabil este când pivotul ajunge cât mai aproape de jumătate, atunci cele două secvențe în care se face autoapel ar fi jumătate din lungimea secvenței inițiale și dacă se întâmplă asta mereu ajungem la un algoritm cu timp de calcul de ordin $n \log_2 n$. Însă nu se poate întâmpla mereu așa.

Mai mult, putem fi în cazul defavorabil al șirului sortat la început crescător (și chiar descrescător), sau în cazuri în care șirul este "aproape" sortat, când se poate ajunge la număr de autoapeluri de ordin `n`. Atunci timpul total de calcul ajunge de ordin n^2 .

O îmbunătățire a performanțelor algoritmului se obține dacă așezăm elementele șirului în ordine aleatoare. În acest fel, cu cât șirul este mai lung cu atât probabilitatea ca primul element să fie mai aproape de mijloc este mai mare.

7. Problema dinți (pbinfo.ro, #842).

Enunțul pe scurt este următorul: Un fierăstrău de tip `n` este format din două fierăstraie de tip `n-1` între care se pune un dinte de înălțime `n`. Un fierăstrău de tip `1` este format doar dintr-un dinte de înălțime `1`. Dându-se `n`, afișați configurația unui fierăstrău de tip `n`. De exemplu, pentru `n=2` trebuie afișat `1 2 1`, iar pentru `n=3` trebuie afișat `1 2 1 3 1 2 1`.

Soluție.

Este cerință clasică de recursivitate cu două autoapeluri identice în care pe ramura cu autoapelul se face o singură afișare, între autoapeluri.

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

void dinti(int n) {
    if (n >= 1) {
        dinti(n-1);
        cout<<n<<" ";
        dinti(n-1);
    }
}

int n;
int main () {
    cin>>n;
    dinti(n);
}
```

8. Pattern (pbinfo.ro, #845).

Se dă un număr natural n . Să se genereze o matrice pătratică de dimensiune 2^n , după următoarele reguli:

- împărțim matricea în 4 submatrice
- cea din stânga-sus are toate elementele 1
- celelalte trei se generează similar, dar au dimensiunea 2^{n-1} .

Valoarea lui n este maxim egală cu 10.

De exemplu, pentru $n = 3$ se afișează:

1 1 1 1 1 1 1 0	O reprezentare mai sugestivă a rezultatului este:
1 1 1 1 1 1 0 0	
1 1 1 1 1 0 1 0	
1 1 1 1 0 0 0 0	
1 1 1 0 1 1 1 0	
1 1 0 0 1 1 0 0	
1 0 1 0 1 0 1 0	
0 0 0 0 0 0 0 0	

Soluție

Regula după care procedăm la pătratul dat inițial este aceeași cu cea de aplicat în pătratele mai mici în care se ajunge prin descompuneri după regulile din problemă. Observăm că și aceste pătrate mai mici sunt tot de latură putere de 2. Așadar trebuie gândită o funcție în care să scriem codul ce rezolvă un pătrat general.

Parametrii acestei funcții vor fi cei prin care identificăm pătratul curent. Putem identifica un pătrat prin două colțuri diametral opuse. Ar fi nevoie de patru parametri. În acest caz, fiind vorba de un pătrat îl putem identifica prin trei parametri: coordonatele unui colț și latura.

Vom scrie o funcție: $f(i, j, L)$ cu semnificația: rezolvă pătratul cu colțul stânga-sus i (linia), j (coloana) și cu latura L .

Observăm că problema devine simplă când ajungem la un pătrat de latură 1. În acest caz nu facem nimic, nemaîncadrându-ne în regula generală când am mai avea de împărțit.

Dacă latura este cel puțin 2 avem:

- pătratul cu colțul stânga-sus i, j și latură $L/2$ îl completăm cu 1.
- Autoapelăm în pătratele de latură $L/2$ cu colțurile stânga-sus respectiv în: $(i, j+L/2)$, $(i+L/2, j)$, $(i+L/2, j+L/2)$.

```
#include <iostream>
#include <fstream>
using namespace std;
int n, a[1030][1030], i, j;

void f(int i, int j, int L) {
    if (L > 1) {
        for (int ii = i; ii<=i+L/2-1; ii++)
            for (int jj = j; jj<=j+L/2-1; jj++)
                a[ii][jj] = 1;
        f(i+L/2, j, L/2);
        f(i, j+L/2, L/2);
        f(i+L/2, j+L/2, L/2);
    }
}

int main ()
{
    cin>>n;
    n = (1<<n);

    f(1, 1, n);

    for (i=1;i<=n; i++) {
        for (j=1;j<=n;j++)
            cout<<a[i][j]<<" ";
        cout<<"\n";
    }
}
```

Observăm că în acest caz, chiar dacă avem mai multe autoapeluri, nu este vorba despre un algoritm exponențial. Autoapelurile nu procesează același lucru de mai multe ori, ele mergând pe zone disjuncte din matricea globală. Nu există element al matricei în care să se ajungă de două ori, așadar numărul de operații este de ordinul numărului de elemente din matrice.

9. **Ridicare la putere în timp logaritm.** Se dau a și b numere naturale, se cere calcularea valorii a^b . Cele două valori sunt cel mult egale cu 10^9 . Întrucât rezultatul poate fi foarte mare, se cere doar restul împărțirii acestuia la numărul 9901.

Soluție

Varianta imediată de rezolvare este o repetiție cu b pași în care se calculează $a \cdot a \cdot a \cdot \dots \cdot a$, de b ori. Evident că soluția nu este practică, timpul de executare nefiind instant.

Soluția mai bună se bazează pe observația:

- Dacă b este par, am putea calcula valoarea $a^{b/2}$, pe care apoi o înmulțim cu ea însăși.
- Din fericire, putem folosi acest truc și dacă b este impar. Dacă vom considera $b/2$ ca fiind câtul împărțirii la 2 al lui b , calculăm, ca și mai sus $a^{b/2}$, înmulțim cu ea însăși și cu încă un a .

Astfel, pentru a calcula a^b este necesar să calculăm puterea la b înjumătățit. Dar înjumătățirea repetată a lui b duce la un algoritm cu timp de calcul de ordin $\log_2 b$.

Mai concis, avem recurența:

1, dacă $b = 0$

$a^b = a^{\frac{b}{2}} \cdot a^{\frac{b}{2}}$, dacă b este nenul și par

$a^{\frac{b}{2}} \cdot a^{\frac{b}{2}} \cdot a$, dacă b este nenul și impar

Acum traducem direct în funcție relația de recurență (având grijă de lucrul esențial, acela de a nu autoapela de două ori cu $b/2$ ci de a memora valoarea primului autoapel și de a o folosi apoi).

```
#include<iostream>
#define MOD 9901
using namespace std;

int calcul(int a, int b) {
    if (b == 0)
        return 1;
    else {
        int rant = calcul(a, b/2);
        if (b%2 == 0)
            return rant * rant % MOD;
        else
            return rant * rant % MOD * a % MOD;
    }
}

int a, b;

int main () {
    cin>>a>>b;
    cout<<calcul(a, b);
    return 0;
}
```

Prezentăm aici și o altă variantă de a calcula a^b cu număr logaritmic de operații, chiar dacă abordarea nu este recursivă. Soluția ce o vom prezenta este folosită chiar mai des în concursuri.

Pentru o explicație mai ușoară pornim de la un exemplu. Presupunem $b=41$.

Atunci, $a^{41} = a^{32+8+1} = a^{2^5+2^3+2^0} = a^{2^5} \cdot a^{2^3} \cdot a^{2^0}$.

Mai sus este scrierea lui b ca sumă de puteri de 2 conform reprezentării lui binare. Observăm că în soluție apar factori cu baza a și exponent putere de 2. Dar exact acele puteri de 2 ce apar în scrierea binară a lui b .

Așadar, pe măsură ce descompunem b în baza 2 construim și puterile corespunzătoare iar atunci când valoarea bitului curent al descompunerii lui b este 1 folosim la soluție valoarea curentă a puterii lui a .

```
#include <iostream>
```

```

#define MOD 9901
using namespace std;
int a, b, r;
int main() {
    cin>>a>>b;
    r = 1;
    while (b) {
        if (b%2 == 1) {
            r = r*a % MOD;
        }
        a = a*a % MOD;
        b/=2;
    }
    cout<<r;
    return 0;
}

```

Nu trebuie trecut în grabă peste instrucțiunea $a=a*a$. Tocmai aici este elementul cheie.

Inițial a este egală cu valoarea originală la puterea 1 (adică 2^0). După prima executare $a=a*a$, a devine valoarea originală la puterea a 2-a (adică 2^1), după încă o executare a ajunge valoarea originală la puterea a 4-a (adică la puterea 2^2), așa că se obțin toate puterile care ne interesează ale lui a original. Pe cele utile le înmulțim la rezultat.

10. **Problema turnurilor din Hanoi.** Este foarte cunoscută în literatura de specialitate și este aproape nelipsită din materialele ce descriu metoda de programare Divide et Impera. Avem trei tije verticale (stive). Pe prima tijă sunt așezate n discuri de raze distincte, în ordine descrescătoare a razelor privind de jos în sus. Se cere să se efectueze mutări încât să ducem toate discurile pe a doua tijă. Avem voie să folosim tija a treia de lucru iar la o mutare putem lua un singur disc din vârful unei stive și să îl așezăm în vârful altei stive, fără însă a plasa vreodată un disc mai mare peste unul mai mic.

Soluție

Dacă notăm tijele cu 1, 2, 3 vom folosi următoarea strategie:

Notăm $hanoi(n, a, b, c)$ ca fiind secvența de mutări necesată să luăm n discuri plasate corect pe tija a , să le ducem corect pe tija b , folosind tija c intermediară. Evident că primul disc pe care îl vom așeza în poziție finală este cel cu raza cea mai mare. În momentul realizării operației trebuie deci ca tija a să conțină doar acest disc, tija b să fie goală deci automat pe tija c să se afle toate celelalte discuri (obligatoriu descrescător de jos în sus). Observăm că în această stare se ajunge mutând mai întâi cele $n-1$ discuri din vârful tije a pe tija c , cu tija b intermediară. Așadar acesta este un autoapel de forma $hanoi(n-1, a, c, b)$. Evident că după mutarea discului mare de pe tija a pe tija b , trebuie să ducem cele $n-1$ discuri așezate corect pe tija c pe tija b folosind tija a ca intermediară. Discul pus deja pe tija b poate fi neglijat pentru că se află în poziție finală și în plus este mai mare decât toate cele care mai sunt de mutat.

Așadar:

$hanoi(n, a, b, c)$ se realizează din: $hanoi(n-1, a, c, b)$, urmat de o mutare efectivă de pe a pe b și apoi de un $hanoi(n-1, c, b, a)$. Putem considera cazul direct cel cu $n=0$ (când nu facem nicio operație) sau $n=1$ când facem doar mutarea directă de pe tija a pe tija b .

Iată o primă variantă de a scrie un program C/C++.

```

#include <iostream>
#include <fstream>

```

```

#include <deque>

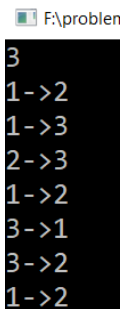
using namespace std;
int n;

void hanoi(int n, int a, int b, int c) {
    if (n == 1) {
        cout<<a<<"->"<<b<<"\n";
    } else {
        hanoi(n-1, a, c, b);
        cout<<a<<"->"<<b<<"\n";
        hanoi(n-1, c, b, a);
    }
}

int main () {
    cin>>n;
    hanoi(n, 1, 2, 3);
    return 0;
}

```

Rulând pentru $n=3$ obținem pe ecran:



```

F:\probleme
3
1->2
1->3
2->3
1->2
3->1
3->2
1->2

```

Iată și o versiune îmbogățită cu o formă de afișare a stivelor după fiecare operație, precum și rezultatul obținut la o rulare pentru $n=3$.

```

#include <iostream>
#include <fstream>
#include <deque>

using namespace std;
deque<int> S[4];
ofstream fout("date.out");
int n;

void afiseazaStiva(deque<int> s, char *msg) {
    fout<<msg;
    for (deque<int>::iterator it = s.begin();it!=s.end();it++)
        fout<<*it<<" ";
    fout<<"\n";
}

```



```

void muta(int a, int b) {
    int x = S[a].back();
    S[a].pop_back();
    S[b].push_back(x);
    afiseazaStiva(S[1], "Stiva 1: ");
    afiseazaStiva(S[2], "Stiva 2: ");
    afiseazaStiva(S[3], "Stiva 3: ");
    fout<<"\n";
}

void hanoi(int n, int a, int b, int c) {
    if (n == 1) {
        muta(a, b);
    } else {
        hanoi(n-1, a, c, b);
        muta(a, b);
        hanoi(n-1, c, b, a);
    }
}

int main () {
    cin>>n;
    for (int i=n;i>=1;i--)
        S[1].push_back(i);
    hanoi(n, 1, 2, 3);
    return 0;
}

```

Conținutul fișierului de ieșire este:

```

Stiva 1: 3 2
Stiva 2: 1
Stiva 3:

Stiva 1: 3
Stiva 2: 1
Stiva 3: 2

Stiva 1: 3
Stiva 2:
Stiva 3: 2 1

Stiva 1:
Stiva 2: 3
Stiva 3: 2 1

Stiva 1: 1
Stiva 2: 3
Stiva 3: 2

Stiva 1: 1
Stiva 2: 3 2
Stiva 3:

```

Stiva 1:
 Stiva 2: 3 2 1
 Stiva 3:

Numărul de operații necesare pentru rezolvarea problemei în cazul general cu n discuri este $2^n - 1$.

O modalitate simplă de a deduce asta este de a da altă semnificație antetului `hanoi(n, a, b, c)` și anume: numărul de mutări necesare pentru a muta cele n discuri de pe tija a pe tija b cu c tijă intermediară. Observăm că valoarea ar fi aceeași indiferent de modul de permutare a codurilor tijelor. Din codul C++ de mai sus, avem:

`hanoi(1) = 1` (mutarea directă)

`hanoi(n) = hanoi(n-1) + 1 + hanoi(n-1)`, adică `hanoi(n) = 2 * hanoi(n-1) + 1`.

Ajungem deci la termenul general pentru `hanoi(n)` ca fiind $2^n - 1$.

11. Algoritmul fill. Enunțul general al acestei probleme cu mare aplicabilitate practică este: dându-se o suprafață care are zone blocate și zone accesibile, precum și o regulă de a se face deplasare între o zonă și alta vecină liberă, să se determine toate zonele accesibile din una dată de pornire.

Formal, se dă o matrice cu n linii și m coloane, iar elementele pot fi 0 și 1. Valoarea 0 reprezintă zonă blocată. Se cunoaște că ne putem deplasa într-un element vecin dacă acolo este 1 și dacă el se află imediat deasupra, imediat sub, imediat în stânga sau imediat în dreapta celui curent. Se mai să o poziție de pornire (garantată liberă), `istart, jstart` și se cere să marcăm cu 2 elementele accesibile din aceasta. Nu este permisă părăsirea matricei.

Date de intrare	Date de ieșire
4 5	0 0 0 1 0
0 0 0 1 0	2 2 0 1 1
1 1 0 1 1	0 2 0 0 1
0 1 0 0 1	2 2 2 0 0
1 1 1 0 0	
2 1	

Ne stabilim starea curentă ca fiind prezența într-o poziție liberă, identificată prin indicii săi (i -linia și j -coloana). Ce avem de făcut de aici? Așa cum spune și problema, ne putem deplasa în una dintre pozițiile vecine dacă acestea indică tot un element din matrice și dacă acolo este tot liber (1). Acum ne vine ideea că dacă am ajunge într-un astfel de vecin ce am avea de făcut ar fi același lucru: să mergem în vecinii liberi și să reluăm procedeul în ei. Așadar putem încerca o abordare recursivă.

Trebuie să ținem cont de un lucru esențial: zona în care ne aflăm este liberă iar vecinul în care mergem trebuie de asemenea să fie o zonă liberă. Dar când aplicăm procedeul recursiv în acesta nu vom găsi iarăși liber elementul din care tocmai am ajuns în el? Acest lucru ar putea declanșa un ciclu infinit. Soluția prin care rezolvăm problema este: Primul lucru pe care îl facem în funcția recursivă la elementul curent (înainte să căutăm recursiv deplasarea în vecini) este să îl marcăm cu o valoare ce nu mai semnifică faptul că el este liber. Rezolvăm astfel două probleme: evităm repetiția infinită (condiționând astfel autoapelurile), dar obținem și marcarea elementelor traversate.

```
#include <iostream>

using namespace std;
int a[1001][1001];
int n, m, istart, jstart;

void umple(int i, int j) {
    a[i][j] = 2;
```

```

    if (i-1 >= 1 && a[i-1][j] == 1)
        umple(i-1, j);
    if (i+1 <= n && a[i+1][j] == 1)
        umple(i+1, j);
    if (j-1 >= 1 && a[i][j-1] == 1)
        umple(i, j-1);
    if (j+1 <= m && a[i][j+1] == 1)
        umple(i, j+1);
}

int main () {
    cin>>n>>m;
    for (int i=1;i<=n;i++)
        for (int j=1;j<=m;j++)
            cin>>a[i][j];
    cin>>istart>>jstart;

    umple(istart, jstart);

    for (int i=1;i<=n;i++) {
        for (int j=1;j<=m;j++)
            cout<<a[i][j]<<" ";
        cout<<"\n";
    }
    return 0;
}

```

Pe lângă faptul că este un algoritm ușor de scris, observăm că timpul de executare este de ordin $n \cdot m$ chiar dacă este vorba despre recursivitate cu mai multe autoapeluri. Acestea se fac disjunct și pentru orice element al matricei se poate june să se facă apel o singură dată (pentru că noi marcăm elementul în care intrăm ca fiind blocat).

Totuși, dacă pentru fiecare dintre cei 4 vecini ar fi și alte lucruri de realizat înainte de autoapel, sau dacă în loc de 4 vecini am avea 8 (și în diagonale, de exemplu), codul s-ar încărca. Pentru aceste cazuri se poate folosi așa numita tehnică a vectorilor de direcții, generalizând modul de tratare a vecinilor.

Astfel, la poziția curentă (i, j) putem aduna respectiv o pereche de valori (d_i, d_j) obținând vecinul ca fiind $(i+d_i, j+d_j)$. Dar cât pot fi valorile din aceste perechi în cazul problemei de mai sus? Răspunsul este: $(-1, 0)$ pentru sus, $(1, 0)$ pentru jos, $(0, -1)$ pentru stânga, $(0, 1)$ pentru dreapta.

Vom folosi d_i și d_j ca fiind doi vectori cu câte 4 componente și îi vom declara astfel (îi inițializăm la declarare).

```

int di[] = {-1, 1, 0, 0};
int dj[] = { 0, 0, -1, 1};

```

Indicii îi folosim de la 0 (când inițializăm la declarare se stochează în tablou începând cu componenta 0).

Observăm că dacă inițializăm vectorii în momentul declarării nu este necesar să scriem și dimensiunea.

Astfel, perechea $(d_i[0], d_j[0]) = (-1, 0)$ reprezintă cât trebuie adunat la (i, j) pentru a obține vecinul de sus.

Programul echivalent este următorul:

```

#include <iostream>
using namespace std;
int a[1001][1001];
int n, m, istart, jstart;
int di[] = {-1, 1, 0, 0};

```

```
int dj[] = { 0,0,-1,1};

void umple(int i, int j) {
    a[i][j] = 2;
    for (int d = 0; d <= 3; d++) {
        int iv = i + di[d];
        int jv = j + dj[d];
        if (iv >= 1 && iv <= n && jv >= 1 && jv <= m && a[iv][jv] == 1)
            umple(iv, jv);
    }
}

int main () {
    cin>>n>>m;
    for (int i=1;i<=n;i++)
        for (int j=1;j<=m;j++)
            cin>>a[i][j];
    cin>>istart>>jstart;

    umple(istart, jstart);

    for (int i=1;i<=n;i++) {
        for (int j=1;j<=m;j++)
            cout<<a[i][j]<<" ";
        cout<<"\n";
    }
    return 0;
}
```

La prima vedere elevii nu consideră neapărat acest cod ca fiind mai aerisit (în comparație cu primul), dar gândiți-vă, așa cum am apus mai sus, ce se întâmplă dacă trebuie să merg în 8 vecini și în plus dacă la trecerea în fiecare vecin ar fi mai multe operații de făcut.