Universitatea Tehnică a Moldovei

Catedra Informatica Aplicată

**RAPORT**

Lucrarea de laborator nr.2

*Analiza și proiectarea algoritmilor*

A efectuat:

st. gr. TI-154 Suruceanu Valentin

A verificat: Bagrin Veronica

lect.,univ.

Chişinău 2016

Tema: Metoda divide et impera

Scopul lucrării:

1. Studierea metodei divide et impera.

2. Analiza şi implementarea algoritmilor bazaţi pe metoda divide et impera.

Note de curs:

1. **Tehnica divide et impera**

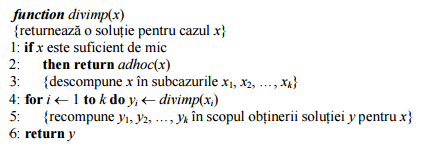
Divide et impera este o tehnica de elaborare a algoritmilor care constă în:

1. Descompunerea cazului ce trebuie rezolvat într-un număr de subcazuri mai mici ale aceleiaşi probleme.

2. Rezolvarea succesivă şi independentă a fiecăruia din aceste subcazuri.

3. Combinarea subsoluţiilor astfel obţinute pentru a găsi soluţia cazului iniţial.

Algoritmul formal al metodei divide et impera:



unde *adhoc* este subalgoritmul de bază folosit pentru rezolvarea micilor subcazuri ale problemei în cauză.

Un algoritm divide et impera trebuie să evite descompunerea recursivă a subcazurilor “suficient de mici”, deoarece, pentru acestea, este mai eficientă aplicarea directă a subalgoritmului de bază.

Observăm că metoda divide et impera este prin definiţie recursivă. Uneori este posibil să eliminăm recursivitatea printr-un ciclu iterativ. Implementată pe o maşina convenţionala, versiunea iterativă poate fi ceva mai rapidă (in limitele unei constante multiplicative). Un alt avantaj al versiunii iterative ar fi faptul că economiseşte spaţiul de memorie. Versiunea recursiva foloseşte o stivă necesară memorării apelurilor recursive. Pentru un caz de mărime n, numărul apelurilor recursive este de multe ori în (log n), uneori chiar în (n).

1. **Algoritmi de sortare „divide et impera”**
   1. **Mergesort (sortarea prin interclasare)**

Algoritmul mergesort ilustrează perfect principiul divide et impera:

- divide problema în subprobleme;

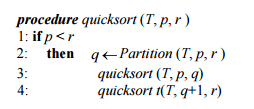
- stăpâneşte subproblemele prin rezolvare;

- combină soluţiile subproblemelor.

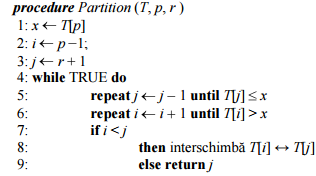
In algoritmul mergesort, suma mărimilor subcazurilor este egală cu mărimea cazului iniţial. Această proprietate nu este în mod necesar valabilă pentru algoritmii divide et impera. Este esenţial ca subcazurile să fie de mărimi cât mai apropiate (sau, altfel spus, subcazurile să fie cât mai echilibrate).

**2.2. Quicksort (sortarea rapida)**

Algoritmul de sortare quicksort, inventat de Hoare în 1962, se bazează de asemenea pe principiul divide et impera. Spre deosebire de mergesort, partea nerecursivă a algoritmului este dedicata construirii subcazurilor şi nu combinării soluţiilor lor.



Mai rămâne să concepem un algoritm de partiţionare (pivotare) cu timp liniar, care să parcurgă tabloul T o singură dată. Putem folosi următoarea procedură



Intuitiv, ne dăm seama că algoritmul quicksort este ineficient, dacă se întâmpla în mod sistematic ca subcazurile să fie puternic neechilibrate. Operaţia de pivotare necesită un timp în (n). 7

Dacă elementele lui T sunt distincte, cazul cel mai nefavorabil este atunci când iniţial tabloul este ordonat crescător sau descrescător, fiecare partiţionare fiind total neechilibrată.

**Listengul Programului în limbajul C++**

import java.util.\*;

import java.lang.Math;

import java.util.Scanner;

class sarcina2 {

public static int a[] = new int[100000];

static long startTime,stopTime,duration;

static long iteratii=0;

public static void main (String[] args) {

Scanner input = new Scanner(System.in);

System.out.printf("Introduceti numarul de elemente al vectorului: ");

int n = input.nextInt();

System.out.printf("\n1.MergeSort\n2.qSort\n3.insertSort\n$: ");

int optSortare = input.nextInt();

switch (optSortare) {

case 1: {

System.out.printf("\nMergeSort:\n1.Cazul cel mai favorabil\n2.Cazul mediu\n3.Cazul cel mai defavorabil\n$: ");

int optCase = input.nextInt();

switch (optCase) {

case 1: {

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

a[i] = 10000+i;

}

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

mergeSort(0,n-1);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

case 2: {

randVector(n);

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

mergeSort(0,n-1);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

case 3: {

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

a[i] = 100000-i;

}

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

mergeSort(0,n-1);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

}

break;

}

case 2: {

System.out.printf("\nqSort:\n1.Cazul cel mai favorabil\n2.Cazul mediu\n3.Cazul cel mai defavorabil\n$: ");

int optCase = input.nextInt();

switch (optCase) {

case 1: {

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

a[i] = 10000+i;

}

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

qSort(0,n-1);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

case 2: {

randVector(n);

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

qSort(0,n-1);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

case 3: {

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

a[i] = 100000-i;

}

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

qSort(0,n-1);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

}

break;

}

case 3: {

System.out.printf("\ninsertSort:\n1.Cazul cel mai favorabil\n2.Cazul mediu\n3.Cazul cel mai defavorabil\n$: ");

int optCase = input.nextInt();

switch (optCase) {

case 1: {

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

a[i] = 10000+i;

}

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

insertSort(n);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

case 2: {

randVector(n);

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

insertSort(n);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

case 3: {

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

a[i] = 100000-i;

}

displayVector(n);

System.out.printf("\nVectorul sortat: \n");

startTime = System.nanoTime();

insertSort(n);

stopTime = System.nanoTime();

duration = stopTime - startTime;

displayVector(n);

System.out.printf("\n");

System.out.println("Iteratii: "+iteratii+"\tTimp: "+duration+"nS");

break;

}

}

break;

}

}

}

// algoritmul merge sort ---------------------------------------->

public static void interclasare ( int p, int m, int q) {

int i = p, j=m+1, k=0;

int b[] = new int[100000];

while (i<=m && j<=q) {

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

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

else

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

iteratii++;

}

while (i<= m) {

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

iteratii++;

}

while (j<=q) {

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

iteratii++;

}

for (i=p; i<=q; i++) {

a[i] = b[i-p];

iteratii++;

}

}

public static void mergeSort (int p, int q) {

if (q>p) {

int m= (p+q)/2;

mergeSort(p,m);

mergeSort(m+1,q);

interclasare(p,m,q);

}

}

//----------------------------------------------------------------<

public static void randVector (int n) {

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

a[i] = (int)(Math.random()\*10000+1);

}

}

public static void displayVector (int n) {

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

System.out.printf("%d\t",a[i]);

}

}

//Algoritmul sortarea rapida -------------------------------------->

public static int divide (int p, int q) {

int st=p, dr = q, x=a[p];

while (st<dr) {

iteratii++;

while (st<dr && a[dr]>=x) {

dr--;

iteratii++;

}

a[st] = a[dr];

while (st<dr && a[st]<=x) {

st++;

iteratii++;

}

a[dr] = a[st];

}

a[st] = x;

return st;

}

public static void qSort (int p,int q) {

int m = divide(p,q);

if (m-1 > p) qSort(p,m-1);

if (m+1 < q) qSort(m+1,q);

}

//------------------------------------------------------------------<

//algoritmul bubble sort-------------------------------------------->

public static void insertSort(int n) {

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

int v=a[i];

int poz;

for (poz = i; poz-1>=0 && a[poz-1]>v; poz--) {

a[poz] = a[poz-1];

iteratii++;

}

iteratii++;

a[poz]=v;

}

}

//------------------------------------------------------------------<

}

**Rezultatele obținute**

Cazul mediu (numărul de iterații)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **F/n** | **10** | **50** | **100** | **1000** |
| **Quiksort** | 36 | 333 | 826 | 14492 |
| **Mergesort** | 43 | 335 | 771 | 10975 |
| **Shellsort** | 37 | 378 | 1387 | 88996 |

Diagrama cazului mediu (numarul de iterații)

Cazul favorabil (numărul de iterații)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **F/n** | **10** | **50** | **100** | **1000** |
| **Quiksort** | 90 | 1607 | 5875 | 514991 |
| **Mergesort** | 86 | 670 | 1542 | 21950 |
| **Shellsort** | 59 | 480 | 1589 | 90998 |

Diagrama cazului mediu (numărului de iterații)

Cazul nefavorabil (numarul de iterații)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **F/n** | **10** | **50** | **100** | **1000** |
| **Quiksort** | 144 | 2881 | 10924 | 1015490 |
| **Mergesort** | 129 | 1005 | 2313 | 32925 |
| **Shellsort** | 81 | 582 | 1791 | 93000 |

Diagrama cazului nefavorabil (numărul de iterații)

Cazul mediu (timpul de execuție)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **F/n** | **10** | **50** | **100** | **1000** |
| **Quiksort** | 0 | 0 | 0,001 | 0 |
| **Mergesort** | 0 | 0 | 0 | 0 |
| **Shellsort** | 0 | 0 | 0 | 0 |

Diagrama cazul mediu (timpul de execuție)

9

Tabelul cazului favorabil (timpul de execuții)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **F/n** | **10** | **50** | **100** | **1000** |
| **Quiksort** | 0 | 0 | 0 | 0 |
| **Mergesort** | 0 | 0 | 0,005 | 0,001 |
| **Shellsort** | 0 | 0 | 0 | 0 |

Diagrama cazul favorabil (timpul de execuție)

Tabelul cazului nefavorabil (timpul de execuții)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **F/n** | **10** | **50** | **100** | **1000** |
| **Quiksort** | 0 | 0 | 0 | 0 |
| **Mergesort** | 0 | 0,001 | 0,003 | 0,002 |
| **Shellsort** | 0 | 0 | 0 | 0 |

Diagrama cazului nefavorabil (tempul de execuții)

**Concluzii :**

În urma analizării algoritmilor de sortare MergeSort , QuickSort și ShellSort toate fiind bazate pe tehnica “divide et impera”. Am analizat cazurile favorabile atunci cînd tabloul este deja sortat crescator , nevaforabile atunci cînd tabloul este sortat descrescator si medie atunci cînd avem un tablou random , am observant ca pentru fiecare algoritm in parte nu conteaza cum e sortat tabloul.

Cel mai eficient algoritm dupa interatii este Mergesort iar dupa timpul de execuții