**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.1

*la Analiza și Proiectarea Algoritmilor*

A efectuat:

st. gr. TI-216 Vlașițchi Ștefan

A verificat:

asist. univ. Andrievschi-Bagrin Veronica

Chişinău - 2022

**Lucrare de laborator nr 1**

**Tema**: Metoda divide et impera.

**Scopul lucrării:**

1. Studierea metodei divide et impera.

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

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.

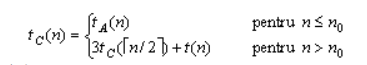
2. Rezolvarea succesivă şi independentă a fiecăruia din aceste subcazuri.

3. Recompunerea subsoluţiilor astfel obţinute pentru a găsi soluţia cazului iniţial. Să presupunem că avem un algoritm A cu timp pătratic. Fie c o constantă, astfel încât timpul pentru a rezolva un caz de mărime n este tA(n) ≤ cn2 . Să presupunem că este posibil să rezolvăm un astfel de caz prin descompunerea în trei subcazuri, fiecare de mărime ⎡n/2⎤. Fie d o constantă, astfel încât timpul necesar pentru descompunere şi recompunere este t(n) ≤ dn. Folosind vechiul algoritm şi ideea de descompunere-recompunere a subcazurilor, obţinem un nou algoritm B, pentru care:

tB(n) = 3tA(⎡n/2⎤)+t(n) ≤ 3c((n+1)/2)2+dn = 3/4cn2+(3/2+d)n+3/4c

Termenul 3/4cn2 domină pe ceilalţi când n este suficient de mare, ceea ce înseamnă ca algoritmul B este în esenţă cu 25% mai rapid decât algoritmul A. Nu am reuşit însă să schimbăm ordinul timpului, care rămâne pătratic.

Putem să continuăm în mod recursiv acest procedeu, împărţind subcazurile în subsubcazuri etc. Pentru subcazurile care nu sunt mai mari decât un anumit prag n0, vom folosi tot algoritmul A. Obţinem astfel algoritmul C, cu timpul



tC(n) este în ordinul lui n lg 3 . Deoarece lg 3 ≅ 1,59, înseamnă că de această dată am reuşit să îmbunătăţim ordinul timpului.

Algoritmul formal al metodei divide et impera:

**funcţion** divimp(x) {returnează o soluţie pentru cazul x}

**if** x este suficient de mic **then** **return** adhoc(x) {descompune x în subcazurile x1, x2, …, xk}

**for** i ← 1 **to** k **do** yi ← divimp(xi) {recompune y1, y2, …, yk în scopul obţinerii soluţiei y pentru x}

return y.

unde adhoc este subalgoritmul de bază folosit pentru rezolvarea micilor subcazuri ale problemei în cauză (în exemplul nostru, acest subalgoritm este A).

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. Ce înseamnă însă “suficient de mic”?

In exemplul precedent, cu toate ca valoarea lui n0 nu influenţează ordinul timpului, este influenţata însă constanta multiplicativa a lui n lg 3 , ceea ce poate avea un rol considerabil în eficienţa algoritmului. Pentru un algoritm divide et impera oarecare, chiar dacă ordinul timpului nu poate fi îmbunătăţit, se doreşte optimizarea acestui prag în sensul obţinerii unui algoritm cât mai eficient. Nu exista o metodă teoretică generală pentru aceasta, pragul optim depinzând nu numai de algoritmul în cauză, dar şi de 2 particularitatea implementării. Considerând o implementare dată, pragul optim poate fi determinat empiric, prin măsurarea timpului de execuţie pentru diferite valori ale lui n0 şi cazuri de mărimi diferite.

In general, se recomandă o metodă hibridă care constă în a a) determinarea teoretică a formei ecuaţiilor recurente; b) găsirea empirică a valorilor constantelor folosite de aceste ecuaţii, în funcţie de implementare.

Revenind la exemplul nostru, pragul optim poate fi găsit rezolvând ecuaţia

tA(n) = 3tA(⎡n/2⎤) + t(n)

Empiric, găsim n0 ≅ 67, adică valoarea pentru care nu mai are importanţă dacă aplicăm algoritmul A în mod direct, sau dacă continuăm descompunerea. Cu alte cuvinte, atâta timp cât subcazurile sunt mai mari decât n0, este bine să continuăm descompunerea. Dacă continuăm însă descompunerea pentru subcazurile mai mici decât n0, eficienţa algoritmului scade.

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 Ω(log n), uneori chiar în Ω(n).

**Codul programului in C++**

**Mergesort.cpp**

#include <iostream>

using namespace std;

int iter=0;

void merge(int \*,int, int , int );

void merge\_sort(int \*arr, int low, int high)

{

    int mid;

    if (low < high){

        //divide the array at mid and sort independently using merge sort

        mid=(low+high)/2;

        merge\_sort(arr,low,mid);

        merge\_sort(arr,mid+1,high);

        //merge or conquer sorted arrays

        merge(arr,low,high,mid);

    }

}

// Merge sort

void merge(int \*arr, int low, int high, int mid)

{

    int i, j, k, c[50];

    i = low;

    k = low;

    j = mid + 1;

    while (i <= mid && j <= high) {

        if (arr[i] < arr[j]) {

            c[k] = arr[i];

            k++;

            i++;

            iter++;

        }

        else  {

            c[k] = arr[j];

            k++;

            j++;

            iter++;

        }

    }

    while (i <= mid) {

        c[k] = arr[i];

        k++;

        i++;

        iter++;

    }

    while (j <= high) {

        c[k] = arr[j];

        k++;

        j++;

        iter++;

    }

    for (i = low; i < k; i++)  {

        arr[i] = c[i];

        iter++;

    }

}

// read input array and call mergesort

int main()

{

    int myarray[30], num;

    cout<<" Lungimea sirului:";

    cin>>num;

    cout<<"Introdu "<<num<<" elemene pentru a fi sortate:";

    for (int i = 0; i < num; i++) { cin>>myarray[i];

    }

    merge\_sort(myarray, 0, num-1);

    cout<<"Sir sortat/n";

    for (int i = 0; i < num; i++)

    {

        cout<<myarray[i]<<"\t";

    }

    cout<<iter;

}

**Quiksort.cpp**

#include <iostream>

using namespace std;

int iter=0;

void swap(int arr[] , int pos1, int pos2){

    int temp;

    temp = arr[pos1];

    arr[pos1] = arr[pos2];

    arr[pos2] = temp;

}

int partition(int arr[], int low, int high, int pivot){

    int i = low;

    int j = low;

    while( i <= high){

        if(arr[i] > pivot){

            i++;

        }

        else{

            swap(arr,i,j);

            i++;

            j++;

        }

        iter++;

    }

    return j-1;

}

void quickSort(int arr[], int low, int high){

    if(low < high){

    int pivot = arr[high];

    int pos = partition(arr, low, high, pivot);

    quickSort(arr, low, pos-1);

    quickSort(arr, pos+1, high);

    }

}

int main()

{

    int n ;

    cout << " Lungimea sirului:";

    cin>>n;

    int arr[n];

    cout <<"Introdu sirul:";

    for( int i = 0 ; i < n; i++){

        cin>> arr[i];

    }

    quickSort(arr, 0 , n-1);

    cout<<"Sir sortat: ";

    for( int i = 0 ; i < n; i++){

        cout<< arr[i]<<"\t";

    }

    cout<<iter;

}

**Analiza grafica a algoritmilor:**

**Concluzie**

Scopul final al lucrări a fost în analiză și perceperea teoretică a algoritilor de a stabili al n-lea număr din șirul Fibonacci.Pentru această am utilizat trei algoritme diferite pentru a identifica cel mai eficient algoritm care va folosi un număr minim de iterații printand același rezultat.Folosind un simplu tablou am putut identifica dar și analiză rezultatele fiecărui algoritm.În final am format unele concluzii precum că,metodă recursiva este cea mai puțin eficientă, căci această calculează de prea multe ori pe când metodă prin formulă să dovedit a fi cea mai eficientă.Pe de altă parte algoritmul iterativ este unul mediu acesta nu a depășit cu mult numarulde iterații.Întrun final putem spune că cele două metode iterativă și prin formulă sau dovedit a fi cel mai eficiente pentru a află al n-lea număr a lui Fibonacci.