Universitatea Tehnică a Moldovei
Facultatea Calculatoare Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

RAPORT

La lucrarea de laborator nr. 2

TEMA: „ Analiza algoritmilor de sortare ”

Disciplina: Analiza și proiectarea algoritmilor

A efectuat studentul: gr.SI-201 Ivanova Evghenia
A verificat: asistentul universitar Buldumac Oleg

Chișinău 2021

Scopul lucrării : Studierea metodei Devide et Impera. Analiza și implementarea algoritmilor bazați pe metoda Devide et Impera și anume a algoritmilor Mergesort, Quicksort și unul la alegere.

Sarcina : Studiază noțiunile teoretice despre metoda Devide et Impera. Implementează algoritmii Mergesort, Quicksort și unul la alegere. Efectuează analiza empirică a algoritmilor propuși.

Considerații teoretice :

Devide et Impera este o tehnică de elaborare a algoritmilor care constă în :

1. Descompunerea cazului ce trebuie rezolvat într-un număr de subcazuri mai mici ale aceleași probleme.
2. Rezolvarea succesivă și independentă a fiecărui subcaz
3. Combinarea soluțiilor astfel obținute pentru a găsi soluția cazului initial

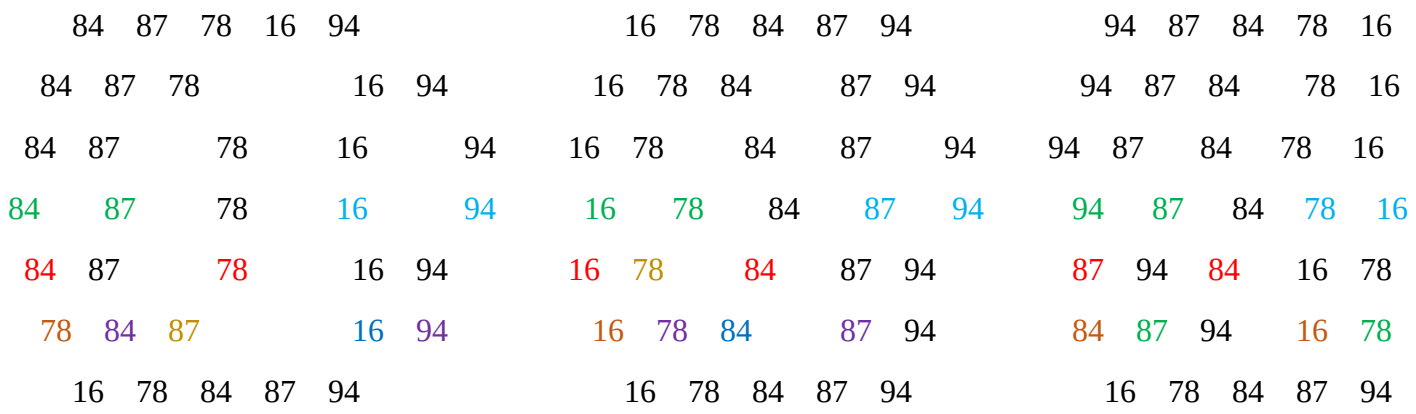
În tehnica Divide et Impera, în urma împărțirilor succesive în subprobleme, se ajunge în situația că problema curentă nu mai poate fi împărțită în subprobleme. O asemenea problemă se numește **problemă elementară** și se rezolvă în alt mod – de regulă foarte simplu.

Divide et Impera admite de regulă o implementare recursivă – rezolvarea problemei constă în rezolvarea unor subprobleme de același tip.

MergeSort

Reprezintă un algoritm de sortare prin interclasare care se bazează pe următoarea idee: pentru a sorta un tabel cu N elemente îl împărțim în două tabele pe care le sortăm separat și le intrclasăm. Este o metodă de sortare care folosește strategia de baza "divide et impera", conform căreia problema se descompune în alte două subprobleme de același tip și după rezolvarea lor rezultatele se combină. Algoritmul sortează elementele în ordine crescătoare. Tabelul se împarte în n/2, după aceea tabelele se împart în jumatate, tot așa până când tabelele formate au mai puțin sau cel mult de k elemente (în cazul nostru k = 2 - este cel mai ușor să compari 2 elemente) .

Exemplu



QuickSort

În practică algoritmul de sortare cel mai rapid este Quicksort numit sortare rapidă, care folosește partiționarea ca idee de bază. Este mai rapid decât orice altă metodă de sortare simplă, se execută bine pentru tabele mari, dar ineficient pentru cele mici. Strategia de bază folosită este "divide et impera", pentru că este mai ușor de sortat două tabele mici, decât una mare. Algoritmul este ușor de implementat, lucrează destul de bine în diferite situații și consumă mai puține resurse decât orice altă metodă de sortare. Necesită numai în jur de NlogN operații în cazul general pentru a sorta N elemente. Metoda QuickSort presupune găsirea poziției finale pe care o ocupă elemenetul de pe prima poziție comparându-l cu elementele din cealaltă partiție a tabelului, acest algoritm realizându-se până când partiția are 1 element. Potrivit algoritmului, fiecare element este comparat cu pivotul, adică operațiunea este de O(N), tabelul este divizat în două părți, fiecare parte este divizată iarăși în două.Dacă fiecare parte este împărțită aproximativ în jumătate, va rezulta log2N împărțiri. Deci timpul de execuție al Quicksortului în caz mediu este de O(N log2N), iar în caz nefavorabil $O(N^2)$.

84	87	78	16	**94**
84	87	78	**16**	94
16	**84**	87	78	94
16	78	84	87	94

16	78	84	87	**94**
16	78	84	**87**	94
16	78	**84**	87	94
16	**78**	84	87	94
16	78	84	87	94

94	87	84	78	**16**
16	**94**	87	84	78
16	87	84	**78**	94
16	78	**87**	84	94
16	78	84	87	94

TimSort

Timsort este un algoritm de sortare hibrid stabil, derivat din sortarea prin îmbinare și sortarea prin inserție, conceput pentru a funcționa bine pe multe tipuri de date din lumea reală. A fost implementat de Tim Peters în 2002 pentru a fi utilizat în limbajul de programare Python. Algoritmul găsește subsecvențe ale datelor care sunt deja ordonate (rulate) și le folosește pentru a sorta restul mai eficient. Acest lucru se realizează prin fuzionarea rulărilor până când sunt îndeplinite anumite criterii. Tim Sort are o complexitate temporală *liniară în* cel mai bun caz, care poate fi testată folosind o listă de numere identice.
Pe scurt, Timsort face 2 lucruri incredibil de bine: arată o performanță excelentă pe matrice cu structură internă preexistentă fiind capabil să mențină o sortare stabilă. Pentru a obține o sortare stabilă, ar trebui să arhivați elementele din listă cu numere întregi și să le sortați ca o matrice de tupluri.

42	68	35	1	70
42	35	68	1	70
35	42	68	1	70
1	35	42	68	70

1	35	42	68	70
1	35	42	68	70
1	35	42	68	70

70	68	42	35	1
68	70	42	35	1
68	42	70	35	1
42	68	70	35	1
42	68	70	1	35
1	35	42	68	70

Codul programului :

```
#include <bits/stdc++.h>

using namespace std;

int  size, kM = 0, kMS = 0, kMD = 0, kQ = 0, kQS = 0, kQD = 0, kT = 0, kTS = 0, kTD = 0, RUN = 3;

time_t beginM, endM, beginMS, endMS, beginMD, endMD, beginQ, endQ, beginQS, endQS, beginQD, endQD, beginT, endT, beginTS, endTS, beginTD, endTD;


void merge(int arr[], int left, int mid, int right){

   int subArrayOne = mid - left + 1, subArrayTwo = right - mid, leftArray[subArrayOne], rightArray[subArrayTwo], indexSubArrayOne = 0, indexSubArrayTwo = 0, indexMergeArray = left;

   for(int i = 0; i < subArrayOne; i++)

      leftArray[i] = arr[left + i];

   for(int j = 0; j < subArrayTwo; j++)

      rightArray[j] = arr[mid + 1 + j];

   while(indexSubArrayOne < subArrayOne && indexSubArrayTwo < subArrayTwo){

      if(leftArray[indexSubArrayOne] < rightArray[indexSubArrayTwo]){

         kM++;

         //cout << "\n " << kM << ". " << leftArray[indexSubArrayOne] << " < " << rightArray[indexSubArrayTwo];

         arr[indexMergeArray++] = leftArray[indexSubArrayOne++];

      }

      else{

         kM++;

         //cout << "\n " << kM << ". " << leftArray[indexSubArrayOne] << " < " << rightArray[indexSubArrayTwo];

         arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];

      }

   }

   while(indexSubArrayOne < subArrayOne)

      arr[indexMergeArray++] = leftArray[indexSubArrayOne++];

   while(indexSubArrayTwo < subArrayTwo)
```

```
        arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
}

void mergeS(int arr[], int left, int mid, int right){
    int subArrayOne = mid - left + 1, subArrayTwo = right - mid, leftArray[subArrayOne],
rightArray[subArrayTwo], indexSubArrayOne = 0, indexSubArrayTwo = 0, indexMergeArray = left;
    for(int i = 0; i < subArrayOne; i++)
        leftArray[i] = arr[left + i];
    for(int j = 0; j < subArrayTwo; j++)
        rightArray[j] = arr[mid + 1 + j];
    while(indexSubArrayOne < subArrayOne && indexSubArrayTwo < subArrayTwo){
        if(leftArray[indexSubArrayOne] < rightArray[indexSubArrayTwo]){
            kMS++;
            //cout << "\n " << kMS << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
        }
        else{
            kMS++;
            //cout << "\n " << kMS << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
        }
    }
    while(indexSubArrayOne < subArrayOne)
        arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
    while(indexSubArrayTwo < subArrayTwo)
        arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
}

void mergeD(int arr[], int left, int mid, int right){
```

```
    int subArrayOne = mid - left + 1, subArrayTwo = right - mid, leftArray[subArrayOne],
rightArray[subArrayTwo], indexSubArrayOne = 0, indexSubArrayTwo = 0, indexMergeArray = left;
    for(int i = 0; i < subArrayOne; i++)
        leftArray[i] = arr[left + i];
    for(int j = 0; j < subArrayTwo; j++)
        rightArray[j] = arr[mid + 1 + j];
    while(indexSubArrayOne < subArrayOne && indexSubArrayTwo < subArrayTwo){
        if(leftArray[indexSubArrayOne] < rightArray[indexSubArrayTwo]){
            kMD++;
            //cout << "\n " << ++kMD << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
        }
        else{
            kMD++;
            //cout << "\n " << kMD << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
        }
    }
    while(indexSubArrayOne < subArrayOne)
        arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
    while(indexSubArrayTwo < subArrayTwo)
        arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
}


void Mergesort(int arr[], int left, int right){
    if(left < right){
        auto mid = (left + right) / 2;
        Mergesort(arr, left, mid);
        Mergesort(arr, mid + 1, right);
```

```
        merge(arr, left, mid, right);
    }
}

void MergesortS(int arr[], int left, int right){
    if(left < right){
        auto mid = (left + right) / 2;
        MergesortS(arr, left, mid);
        MergesortS(arr, mid + 1, right);
        mergeS(arr, left, mid, right);
    }
}

void MergesortD(int arr[], int left, int right){
    if(left < right){
        auto mid = (left + right) / 2;
        MergesortD(arr, left, mid);
        MergesortD(arr, mid + 1, right);
        mergeD(arr, left, mid, right);
    }
}

void swap(int* a, int* b){
    int t = *a;
    *a = *b;
    *b = t;
}

int partition(int arr[], int low, int high){
    int pivot = arr[high], i = (low - 1);
```

```
    for(int j = low; j <= high - 1; j++){
        kQ++;
        //cout << "\n " << kQ << ". " << arr[j] << " > " << pivot;
        if(arr[j] <= pivot) swap(&arr[++i], &arr[j]);
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

int partitionS(int arr[], int low, int high){
    int pivot = arr[high], i = (low - 1);
    for(int j = low; j <= high - 1; j++){
        kQS++;
        //cout << "\n " << kQS << ". " << arr[j] << " > " << pivot;
        if(arr[j] <= pivot) swap(&arr[++i], &arr[j]);
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

int partitionD(int arr[], int low, int high){
    int pivot = arr[high], i = (low - 1);
    for(int j = low; j <= high - 1; j++){
        kQD++;
        //cout << "\n " << kQD << ". " << arr[j] << " > " << pivot;
        if(arr[j] <= pivot) swap(&arr[++i], &arr[j]);
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
void Quicksort(int arr[], int low, int high) {
   if(low < high) {
      int pi = partition(arr, low, high);
      Quicksort(arr, low, pi - 1);
      Quicksort(arr, pi + 1, high);
   }
}

void QuicksortS(int arr[], int low, int high) {
   if(low < high) {
      int pi = partitionS(arr, low, high);
      QuicksortS(arr, low, pi - 1);
      QuicksortS(arr, pi + 1, high);
   }
}

void QuicksortD(int arr[], int low, int high) {
   if(low < high) {
      int pi = partitionD(arr, low, high);
      QuicksortD(arr, low, pi - 1);
      QuicksortD(arr, pi + 1, high);
   }
}

void insertionSort(int arr[], int left, int right){
   for(int i = left + 1; i <= right; i++){
      int temp = arr[i], j = i - 1;
      while(j >= left && arr[j] > temp){
         kT++;
```

```
            //cout << "\n " << kT << ". " << arr[j] << " > " << temp;
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}

void insertionSortS(int arr[], int left, int right){
    for(int i = left + 1; i <= right; i++){
        int temp = arr[i], j = i - 1;
        while(j >= left && arr[j] > temp){
            kTS++;
            //cout << "\n " << kTS << ". " << arr[j] << " > " << temp;
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}

void insertionSortD(int arr[], int left, int right){
    for(int i = left + 1; i <= right; i++){
        int temp = arr[i], j = i - 1;
        while(j >= left && arr[j] > temp){
            kTD++;
            //cout << "\n " << kTD << ". " << arr[j] << " > " << temp;
            arr[j+1] = arr[j];
            j--;
        }
```

```
        arr[j+1] = temp;
    }
}


void mergeT(int arr[], int left, int mid, int right){
    int subArrayOne = mid - left + 1, subArrayTwo = right - mid, leftArray[subArrayOne],
rightArray[subArrayTwo], indexSubArrayOne = 0, indexSubArrayTwo = 0, indexMergeArray = left;
    for(int i = 0; i < subArrayOne; i++)
        leftArray[i] = arr[left + i];
    for(int j = 0; j < subArrayTwo; j++)
        rightArray[j] = arr[mid + 1 + j];
    while(indexSubArrayOne < subArrayOne && indexSubArrayTwo < subArrayTwo){
        if(leftArray[indexSubArrayOne] < rightArray[indexSubArrayTwo]){
            kT++;
            //cout << "\n " << kT << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
        }
        else{
            kT++;
            //cout << "\n " << kT << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
        }
    }
    while(indexSubArrayOne < subArrayOne)
        arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
    while(indexSubArrayTwo < subArrayTwo)
        arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
}
```

```
void mergeTS(int arr[], int left, int mid, int right){
   int subArrayOne = mid - left + 1, subArrayTwo = right - mid, leftArray[subArrayOne],
rightArray[subArrayTwo], indexSubArrayOne = 0, indexSubArrayTwo = 0, indexMergeArray = left;
   for(int i = 0; i < subArrayOne; i++)
      leftArray[i] = arr[left + i];
   for(int j = 0; j < subArrayTwo; j++)
      rightArray[j] = arr[mid + 1 + j];
   while(indexSubArrayOne < subArrayOne && indexSubArrayTwo < subArrayTwo){
      if(leftArray[indexSubArrayOne] < rightArray[indexSubArrayTwo]){
         kTS++;
         //cout << "\n " << kTS << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
         arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
      }
      else{
         kTS++;
         //cout << "\n " << kTS << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
         arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
      }
   }
   while(indexSubArrayOne < subArrayOne)
      arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
   while(indexSubArrayTwo < subArrayTwo)
      arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
}


void mergeTD(int arr[], int left, int mid, int right){
   int subArrayOne = mid - left + 1, subArrayTwo = right - mid, leftArray[subArrayOne],
rightArray[subArrayTwo], indexSubArrayOne = 0, indexSubArrayTwo = 0, indexMergeArray = left;
   for(int i = 0; i < subArrayOne; i++)
```

```
        leftArray[i] = arr[left + i];
    for(int j = 0; j < subArrayTwo; j++)
        rightArray[j] = arr[mid + 1 + j];
    while(indexSubArrayOne < subArrayOne && indexSubArrayTwo < subArrayTwo){
        if(leftArray[indexSubArrayOne] < rightArray[indexSubArrayTwo]){
            kTD++;
            //cout << "\n- " << kTD << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
        }
        else{
            kTD++;
            //cout << "\n- " << kTD << ". " << leftArray[indexSubArrayOne] << " < " <<
rightArray[indexSubArrayTwo];
            arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
        }
    }
    while(indexSubArrayOne < subArrayOne)
        arr[indexMergeArray++] = leftArray[indexSubArrayOne++];
    while(indexSubArrayTwo < subArrayTwo)
        arr[indexMergeArray++] = rightArray[indexSubArrayTwo++];
}


void Timsort(int arr[], int n){
    for(int i = 0; i < n; i+=RUN)
        insertionSort(arr, i, min((i+RUN-1),(n-1)));
    for(int size = RUN; size < n; size = 2*size){
        for(int left = 0; left < n; left += 2*size){
            int mid = left + size - 1,
            right = min((left + 2*size - 1),(n-1));
```

```
            if(mid < right) mergeT(arr, left, mid, right);
        }
    }
}

void TimsortS(int arr[], int n){
    for(int i = 0; i < n; i+=RUN)
        insertionSortS(arr, i, min((i+RUN-1),(n-1)));
    for(int size = RUN; size < n; size = 2*size){
        for(int left = 0; left < n; left += 2*size){
            int mid = left + size - 1, right = min((left + 2*size - 1),(n-1));
            if(mid < right) mergeTS(arr, left, mid, right);
        }
    }
}

void TimsortD(int arr[], int n){
    for(int i = 0; i < n; i+=RUN)
        insertionSortD(arr, i, min((i+RUN-1),(n-1)));
    for(int size = RUN; size < n; size = 2*size){
        for(int left = 0; left < n; left += 2*size){
            int mid = left + size - 1, right = min((left + 2*size - 1),(n-1));
            if(mid < right) mergeTD(arr, left, mid, right);
        }
    }
}

void printArray(int arr[], int size){
    for(int i = 0; i < size; i++)
        cout << arr[i] << " ";
```

```
}

int main(){
    cout << "\tSize of table ";
    cin >> size;
    int arrM[size], arrQ[size], arrT[size];
    for(int i = 0; i < size; i++)
        arrM[i] = rand() % 100 +1;
    for(int i = 0; i < size; i++){
        arrQ[i] = arrM[i];
        arrT[i] = arrM[i];
    }

    //     Mergesort
    cout << "\n\tMethod Mergesort :\n Table random\t\t";
    //printArray(arrM, size);
    auto beginM = chrono::high_resolution_clock::now();
    Mergesort(arrM, 0, size - 1);
    auto endM = chrono::high_resolution_clock::now() - beginM;
    cout << "\n Nr of steps: " << kM << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endM).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrM, size);
    auto beginMS = chrono::high_resolution_clock::now();
    MergesortS(arrM, 0, size - 1);
    auto endMS = chrono::high_resolution_clock::now() - beginMS;
    cout << "\n Nr of steps: " << kMS << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endMS).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrM, size);
    cout << "\n Descending\t\t";
    sort(arrM, arrM + size, greater<int>());
```

```
    //printArray(arrM, size);
    auto beginMD = chrono::high_resolution_clock::now();
    MergesortD(arrM, 0, size - 1);
    auto endMD = chrono::high_resolution_clock::now() - beginMD;
    cout << "\n Nr of steps: " << kMD << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endMD).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrM, size);

    //     Quicksort
    cout << "\n\n\tMethod Quicksort :\n Table random\t\t";
    //printArray(arrQ, size);
    auto beginQ = chrono::high_resolution_clock::now();
    Quicksort(arrQ, 0, size - 1);
    auto endQ = chrono::high_resolution_clock::now() - beginQ;
    cout << "\n Nr of steps: " << kQ << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endQ).count()*1e-9  << "\n Table sorted\t\t";
    //printArray(arrQ, size);
    auto beginQS = chrono::high_resolution_clock::now();
    QuicksortS(arrQ, 0, size - 1);
    auto endQS = chrono::high_resolution_clock::now() - beginQS;
    cout << "\n Nr of steps: " << kQS << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endQS).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrQ, size);
    cout << "\n Descending\t\t";
    sort(arrQ, arrQ + size, greater<int>());
    //printArray(arrQ, size);
    auto beginQD = chrono::high_resolution_clock::now();
    QuicksortD(arrQ, 0, size - 1);
    auto endQD = chrono::high_resolution_clock::now() - beginQD;
    cout << "\n Nr of steps: " << kQD << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endQD).count()*1e-9 << "\n Table sorted\t\t";
```

```
    //printArray(arrQ, size);


    //     Timsort
    cout << "\n\n\tMethod Timsort :\n Table random\t\t";
    //printArray(arrT, size);
    auto beginT = chrono::high_resolution_clock::now();
    Timsort(arrT, size);
    auto endT = chrono::high_resolution_clock::now() - beginT;
    cout << "\n Nr of steps: " << kT << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endT).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrT, size);
    auto beginTS = chrono::high_resolution_clock::now();
    TimsortS(arrT, size);
    auto endTS = chrono::high_resolution_clock::now() - beginT;
    cout << "\n Nr of steps: " << kTS << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endTS).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrT, size);
    cout << "\n Descending\t\t";
    sort(arrT, arrT + size, greater<int>());
    //printArray(arrT, size);
    auto beginTD = chrono::high_resolution_clock::now();
    TimsortD(arrT, size);
    auto endTD = chrono::high_resolution_clock::now() - beginTD;
    cout << "\n Nr of steps: " << kTD << "\n Timpul de executie: " <<
chrono::duration_cast<chrono::nanoseconds>(endTD).count()*1e-9 << "\n Table sorted\t\t";
    //printArray(arrT, size);
}
```

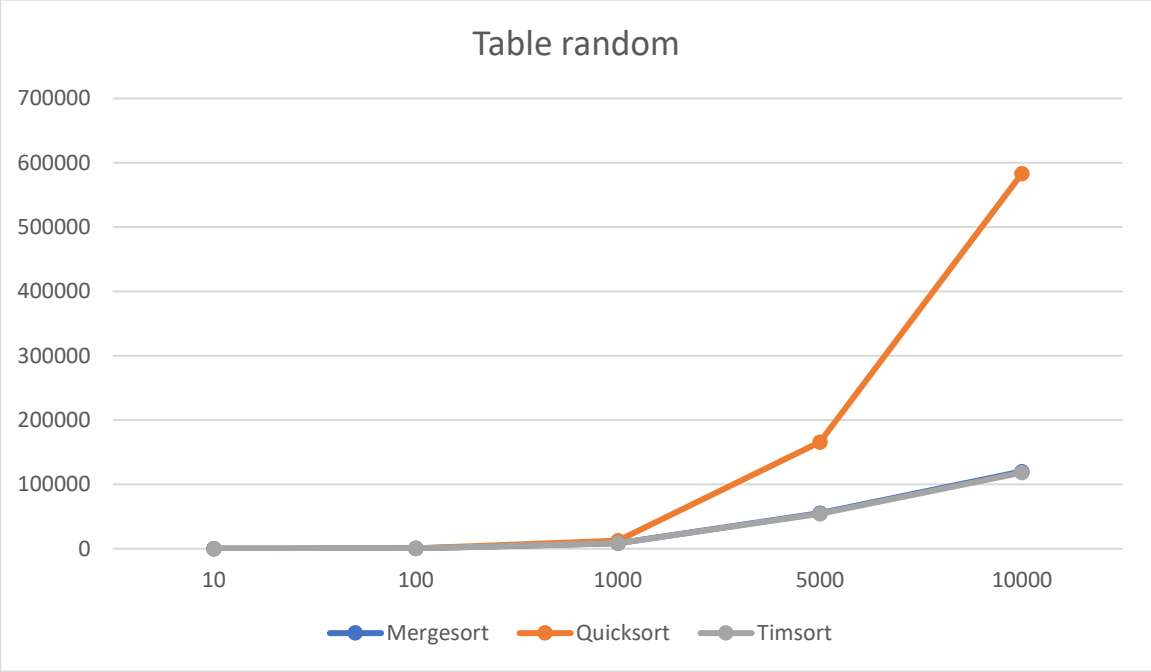
Execuția programului :

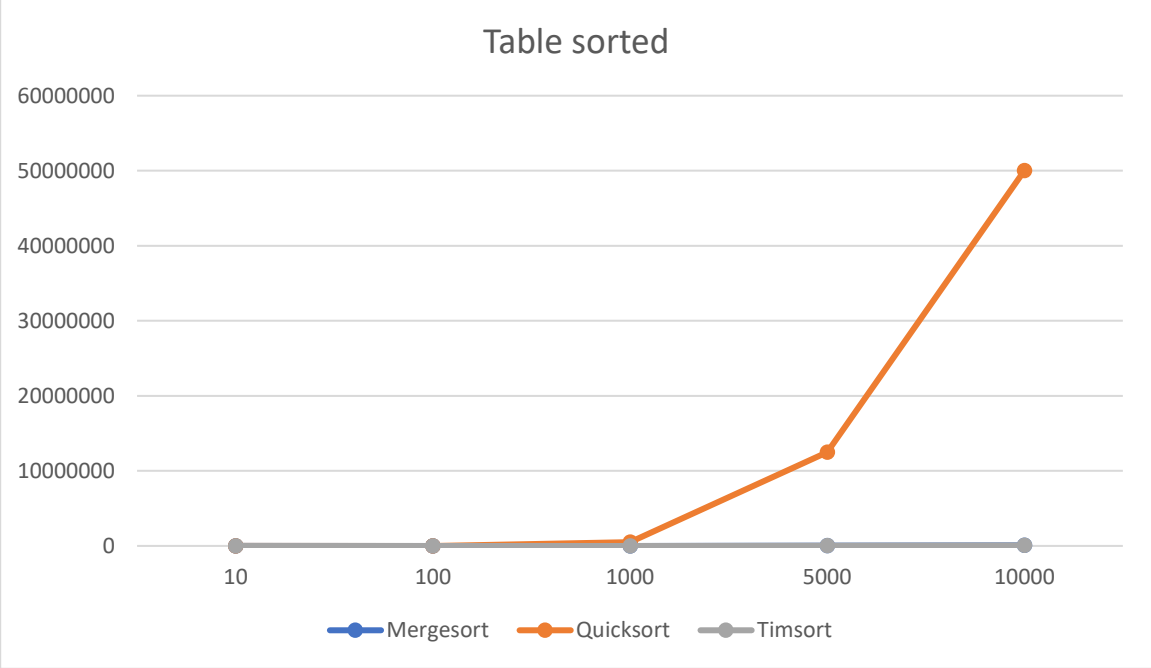
```
        Size of table 10000

       Method Mergesort :
Table random
Nr of steps: 120140
Timpul de executie: 0.001006
Table sorted
Nr of steps: 74733
Timpul de executie: 0.00132
Table sorted
Descending
Nr of steps: 64608
Timpul de executie: 0.001314
Table sorted

       Method Quicksort :
Table random
Nr of steps: 583218
Timpul de executie: 0.004837
Table sorted
Nr of steps: 49995000
Timpul de executie: 0.243322
Table sorted
Descending
Nr of steps: 13364247
Timpul de executie: 0.072064
Table sorted

       Method Timsort :
Table random
Nr of steps: 118329
Timpul de executie: 0.001002
Table sorted
Nr of steps: 72521
Timpul de executie: 0.003001
Table sorted
Descending
Nr of steps: 57029
Timpul de executie: 0.001433
Table sorted
```

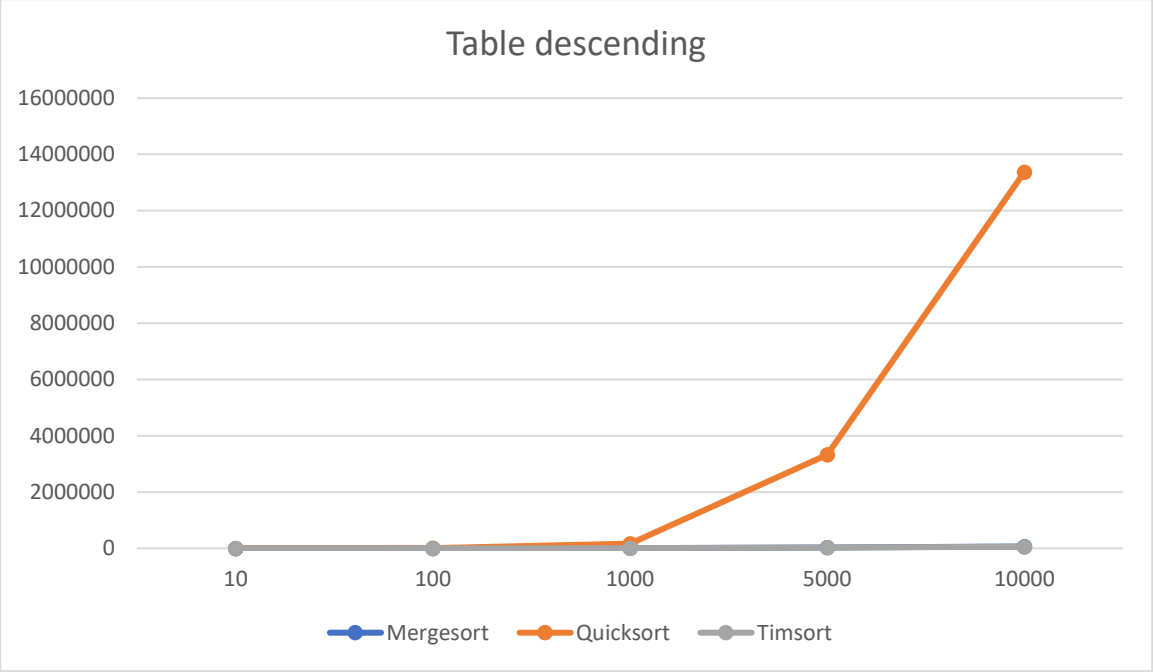
Tabelul cu datele de ieșire :

	10	**100**	**1000**	**5000**	**10000**
Mergesort	24 0.00000169s 19 0.000001779s 15 0.00000128s	545 0.00001497s 366 0.00001442s 316 0.00001272s	8705 0.00016727s 5760 0.00012969s 4932 0.00011603s	55128 0.001346s 34473 0.000499s 29804 0.000789s	120140 0.001006s 74733 0.00132s 64608 0.001314s
Quicksort	25 0.000000896s 45 0.000000896s 45 0.000000938s	592 0.00000928s 4950 0.00005351s 3374 0.00002757s	12694 0.00013304s 499500 0.00318766s 160971 0.00076963s	165971 0.000996s 12497500 0.06608s 3322621 0.020298s	583218 0.004837s 49995000 0.243322s 13364247 0.072064s

Timsort	22 0.000000612s 12 0.000000213s 17 0.000000569s	573 0.00001266s 352 0.0000035s 316 0.00001174s	8621 0.00009359s 5440 0.00003057s 4111 0.00006401s	54183 0.000997s 33099 0.000225s 26089 0.000268s	118329 0.001002s 72521 0.003001s 57029 0.001433s







	Complexitatea în timp	**Complexitatea în spațiu**	**Stabil / Instabil**	**Spațiul de memorie**	**Recursiv/Non-Recursiv**
MergeSort	În toate cazurile $O(nlogn)$	$O(n)$	Stabil	Not – In -place (nu sortează în loc)	Recursiv
QuickSort	$O(nlogn)$ $O(nlogn)$ $O(n^2)$	$O(1)$	Instabil	In -place	Recursiv
TimSort	$O(n)$ $O(nlogn)$ $O(nlogn)$	$O(n)$	Stabil	Not – In -place	Recursiv

Concluzia :

În cadrul acestui laborator am analizat și am implementat 3 algoritmi de sortare după paradigma Devide et Impera. Pentru o analiză eficientă am rulat acești algoritmi pentru 3 cazuri : pentru un tablou random, sortat crescător și descrescător.

MergeSort este un algoritm care necesită multă memorie. Un avantaje al acestui față de QuickSort că lucrează extrem de bine cu listele.

Quicksort este o metodă bună în caz general, dar nu și în caz nefavorabil când este preferabil folosirea a 3 indici de impărțire. Randomizarea este o idee importantă și folositoare, o unealtă generală pentru a îmbunătăți algoritmul. Quicksort este sensibil la ordinea datelor de intrare. Nu este o metodă stabilă. Dezavantajul algoritmului este că, e recursiv. Necesită în jur de N^2 de operații în caz nefavorabil. Este fragil, o simplă greșeală în implementare poate cauza o executare greșită.

TimSort este avantajos față de Quicksort pentru sortarea referințelor de obiecte sau indicatori, deoarece acestea necesită o indirectă costisitoare a memoriei pentru a accesa date și a efectua comparații.