# Tema Analiza Algoritmilor

## - Subiectul 1: Sortare -

### Algoritmi: QuickSort, MergeSort, HeapSort

Facultatea de Automatica si Calculatoare

Universitatea Politehnica Bucuresti

Găvan Adrian-George

324CA

# Cuprins

# 1. Introducere

Tema aleasa de mine pentru a realiza un studiu comparativ asupra principalilor algoritmi este “Subiectul 1 – Sortare”. Cei 3 algoritmi care trebuie implementati pentru acest studiu comparativ sunt algoritmii “QuickSort”, “MergeSort” si “HeapSort”.

Sortarea este una dintre cele mai des intalnite subprobleme in programare. Aceasta constituie o parte esentiala din numeroasele procese de prelucrare a datelor. Operatia de sortare este efecutata frecvent de catre oameni in viata de zi cu zi, cum ar fi sortarea programelor TV favorite, ordonarea hainelor, prioritatilor, anumitor dosare etc.

Sortarea reprezinta o metoda prin care elementele unei multimi sunt aranjate intr-o anumita ordine, numerica sau lexicografica, pe baza valorilor unor chei. Sortarea este una dintre cele mai raspandite operaţii care se executa pe calculatoarele de astazi iar de eficienta ei depind performantele altor algoritmi care opereaza asupra unor colectii ce trebuie sa fie în prealabil ordonate (precum algoritmii de cautare sau interclasare).

Chiar daca exista o multitudine de algoritmi de sortare, nu se poate spune ca unul dintre acesti algoritmi este cel mai bun. Majoritatea algoritmilor de sortare sunt mai buni decat altii in anumite cazuri, de exemplu in functie de datele de intrare, daca se doreste ca sortarea sa se efectueze mai rapid cu un cost de memorie utilizata mai mare sau daca este putina memorie pusa la dispozitie, ceea ce inseamna ca trebuie utilizat un algoritm ce foloseste putina memorie, chiar daca este mai lent.

Aplicatii practice in care este folosita sortarea:

- QuickSort este folosit pentru scoruri sportive (scorurile sunt organizate pe baza raportului wins-losses).
- MergeSort este folosit in baze de date (bazele de date folosesc un MergeSort extern pentru a sorta cantitati de date prea mari pentru a fi incarcate efectiv in memorie).
- HeapSort este folosit pentru citirea codurilor de bare de pe cardurile de plastic (serviciul iti permite sa comunici cu baza de date pentru a rula continuu verificari si a intoarce date precum cine este cel mai activ user etc.).
- BubbleSort este folosit in domeniul TV pentru a sorta diferitele canale de televiziune.
- RadixSort este folosit de eBay pentru a ne permite sa sortam inregistrarile in functie de suma licitata curent.

In continuare, voi prezenta cativa dintre cei mai importanti algoritmi de sortare: QuickSort (cu 3 variante de optimizare a pivotului: primul element, elementul din mijloc si un element random), MergeSort si HeapSort. Voi compara algoritmii din punct de vedere al timpului de rulare pe cazul favorabil, mediu si defavorabil si din punct de vedere al spatiului pe cazul cel mai defavorabil. Seturile de teste vor fi variate, continand diferite tipuri de date (Int, String si un tip de data custom), dimensiuni diferite (de la 2.000 la 20.000.000 de intrari), iar testele vor trata si cazurile favorabile si defavorabile.

Am ales acesti 3 algoritmi pentru ca sunt foarte utilizati, sunt eficienti si fiecare dintre acestia face parte dintr-o categorie diferita de sortare: QuickSort – Partitionare, HeapSort – Selectie, MergeSort – Interclasare.

## 2. Prezentarea si implementarea solutiilor

Problema: Se da un vector de dimensiune n si se cere vectorul sortat.

Sortarea este procesul prin care elementele unei colectii sunt rearanjate astfel incat acestea sa se afle intr-o anumita ordine. In cazul de fata, cazul favorabil ar fi ca vectorul dat sa contine elementele deja sortate, iar cazul defavorabil ar fi ca elementele sa fie sortate in ordine inversa.

Voi detalia fiecare din cei 3 algoritmi alesi pentru rezolvarea problemei, explicand cum functioneaza fiecare dintre acestia si explicand punctele cheie ale solutiilor.

### 2.1 QuickSort

QuickSort este un algoritm de sortare conceput de catre C.A.R. Hoare in 1962. Este un algoritm recursiv, in-place si se bazeaza pe principiul “divide et impera”. Majoritatea oamenilor spun ca acest algoritm este in esenta un algoritm in-place mai rapid decat MergeSort, pentru ca necesita un spatiu auxiliar mai mic si cazul cel mai defavorabil O($n^2$) poate fi usor evitat prin alegerea corecta a pivotului. Algoritmul este simplu in teorie, dar mai greu de implementat. Cercetatorii din domeniul informaticii au avut nevoie de ani de zile pentru a obtine o implementare practica a acestui algoritm. In continuare se cauta variante de imbunatatire a performantelor, fie prin marirea numarului de pivoti, fie prin determinarea pivotului prin metoda medianei in 3, 5, 7... elemente. Un lucru important de mentionat este faptul ca QuickSort nu este un algoritm stabil (un algoritm este stabil daca 2 chei cu aceeasi valoare apar in vectorul sortat in aceeasi ordine in care apareau si in datele de intrare). Pentru baze mari de date se prefera MergeSort, acest algoritm fiind unul stabil.

Algoritmul recursiv este format din 4 pasi:

- Daca este un singur element sau niciun element in vector, se returneaza imediat.
- Se alege un element din array ca si “pivot”. Alegerea pivotului este esentiala pentru eficienta algoritmului. Sunt mai multe modalitati de alegere a pivotului: elementul cel mai din stanga, elementul cel mai din dreapta, elementul din mijloc sau un element random si multe altele mai avansate. In implementarea mea, pentru realizarea studiului comparativ, am ales 3 variante pentru pivot: cel mai din stanga element, elementul din mijloc si un element random.
- Partitionarea: se rearanjeaza elementele vectorului in asa fel incat pivotul va fi pus pe pozitia corecta a vectorului sortat, toate elementele mai mici decat pivotul vor fi puse in partea stanga a pivotului si toate elementele mai mari decat pivotul vor fi puse in partea dreapta a pivotului. Cu alte cuvinte vectorul este impartit in 2

parti(partitii), o parte ce contine elementele mai mici decat pivotul si o parte care contine elementele mai mari decat pivotul. Acest pas este punctul cheie al algoritmului QuickSort.

- Se sorteaza cele 2 partitii apelandu-se recursiv algoritmul pe fiecare partitie.

## 2.2 MergeSort

MergeSort este un algoritm de sortare conceput de John von Neumann in 1945. Acest algoritm, precum QuickSort, este un algoritm recursiv bazat pe principiul "divide et impera", dar spre deosebire de QuickSort acest algoritm este stabil. Algoritmul imparte vectorul ce trebuie sortat in 2 vectori de dimensiuni egale, se apeleaza recursiv algoritmul pe fiecare din cele 2 jumatati ale vectorului si apoi se interclaseaza cele 2 jumatati sortate pentru a forma vectorul sortat. MergeSort este mai incet in practica decat QuickSort pentru ca ocupa mai multa memorie (utilizeaza o memorie auxiliara de complexitate O(n), iar alocarea si dezalocarea memoriei ocupa timp). Mai mult decat atat, complexitatea medie pentru ambele este O(n*logn) dar constantele difera, iar pentru QuickSort este mai usor de evitat cazul defavorabil prin alegerea pivotului corespunzator. De asemenea, QuickSort este "cache-friendly" cand este folosit pentru vectori. MergeSort este mai bun pentru baze mari de date, pentru ca este un algoritm stabil si este usor de adaptat pentru a lucra cu liste inlantuite si cu liste foarte mari de pe disc-uri de stocare (interne, externe sau in retea).

Algoritmul recursiv este format din urmatorii pasi:

- Se imparte vectorul in 2 vectori de dimensiuni egale. NU se foloseste pivot ca la QuickSort.
- Se aplica recursiv algoritmul pe fiecare dintre cei 2 subvectori.
- Atunci cand marimea subvectorului este 0 sau 1 acesta este considerat sortat.
- Se interclaseaza subvectorii sortati intr-un singur vector. Interclasarea (Merge-ul) este un punct cheie al solutiei. Se itereaza prin elementele celor 2 subvectori si se verifica daca elementul din primul subvector este mai mic decat elementul din al doilea subvector. Daca elementul este mai mic se adauga in vectorul sortat elementul din primul subvector, iar daca nu, se va adauga in vectorul sortat elementul din cel de-al doilea subvector. La final, se copiaza elementele ramase dintr-un subvector (daca exista elemente ramase) la finalul vectorului sortat.

## 2.3 HeapSort

Algoritmul HeapSort a fost conceput de catre J.W.J. Williams in 1964. Pentru a descrie algoritmul, trebuie mai intai intelese notiunile de Complete Binary Tree, Binary Heap si cum este un Binary Heap reprezentat ca un vector.

Un Complete Binary Tree (arbore binar complet) este un arbore binar in care fiecare nivel este plin (posibil in afara de ultimul nivel) si toate nodurile sunt cat mai la stanga posibil. Un Binary Heap (Heap binar) este un arbore binar complet in care nodurile au urmatoarea

proprietate: valoarea din orice nod parinte este mai mare (respectiv mai mica) decat valoarea din nodurile copiiilor sai. Daca nodul parinte este mai mare (respectiv mai mic) heap-ul se numeste Max Heap (respectiv Min Heap). Cum un Binary Heap este un arbore binar complet, este usor de reprezentat sub forma unui vector (care este eficient ca si spatiu). Elementele din vector sunt ordonate dupa urmatoarea regula: daca nodul parinte este la indexul I, pozitia copilului stang va fi la 2*I + 1 si pozitia copilului drept va fi la pozitia 2*I+2 (in cazul in care indexarea incepe de la 0).

HeapSort-ul este cel mai lent algoritm de sortare cu complexitate O(n*logn), dar spre deosebire de QuickSort si MergeSort are avantajul ca nu este recursiv. Acest lucru favorizeaza folosirea algoritmului pentru seturi mari de date (de milioane de intrari). De asemenea are si avantajul ca in cazul cel mai dezavantajos, complexitatea temporala este tot O(n*logn). HeapSort este un algoritm care “impaca” viteza cu consumul relativ mic de memorie. Este un algoritm in-place, care nu este stabil. HeapSort este mai incet in practica decat MergeSort pentru ca dureaza mai mult reconstruirea Heap-ului dupa fiecare extragere a elementului din varf.

Algoritmul este format din urmatorii pasi:

- Se construieste Heap-ul sub forma de vector (in cazul nostru Max Heap).
- In acest moment, cel mai mare element va fi in radacina Heap-ului. Se va interschimba ultimul element cu radacina si se va micsora dimensiunea Heap-ului cu 1 (scoatem cel mai mare element). In final, se va reconstrui arborele cu elementele 1..(n-1), astfel incat cel mai mare element sa fie radacina. Pasul de reconstruire este punctul cheie al algoritmului. Procesul de reconstruire se face din partea de jos a arborelui spre varf, pentru ca procesul de reconstruire se poate aplica unui nod daca copii sai au fost reconstruiti (adica copii sai contin cele mai mari elemente din subarbori). Se verifica daca copilul stang este mai mare decat root-ul sau daca copilul drept este mai mare decat copilul stang sau root-ul. Daca unul din copii este mai mare, se interschimba cele 2 elemente si se reconstruieste subarborele.
- Se aplica pasii de mai sus cat timp dimensiunea arborelui este mai mare ca 1.

## 2.4 Paralela intre cei 3 algoritmi

| Algoritm | Structura date | Complexitate temporala | | | Complexitate spatiala pentru cel mai defavorabil caz |
|---|---|---|---|---|---|
| | | Favorabil | Mediu | Defavorabil | |
| **QuickSort** | Vector | O(n log(n)) | O(n log(n)) | O(n^2) | O(n) |
| **MergeSort** | Vector | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(n) |
| **HeapSort** | Vector | O(n log(n)) | O(n log(n)) | O(n log(n)) | O(1) |

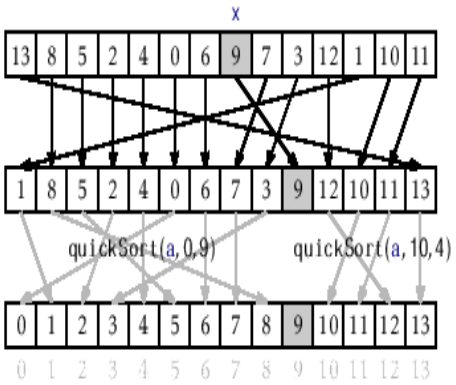
# 3. Complexitate si testare

## 3.1 Complexitate

### 3.1.1 Complexitate QuickSort

Acest subcapitol contine analiza complexitatii algoritmului QuickSort. In implementarea solutiilor, conform cerintei, am utilizat 3 variante de optimizare ale algoritmului, cu 3 pivoti diferiti (mentionati in descrierea algoritmului).

```
/* Functia ia primul element ca si pivot, pune pivotul in pozitia corecta
  din vectorul sortat, plaseaza toate elementele mai mici decat pivotul
  la stanga acestuia si toate elementele mai mari la dreapta. */
template<typename Type>
int partitionPrim (Type* array, int low, int high) {
    Type pivot = array[low];
    low--;
    high++;
    while(low <= high) {
        low++;
        while(array[low] < pivot) {
          low++;
        }
        high--;
        while(array[high] > pivot) {
          high--;
        }
        if(low >= high) {
          break;
        }
        swap(array[low], array[high]);
    }
    return high;
}

/* Functia care implementeaza Quick Sort.*/
template<typename Type>
void quickSortPrim(Type* array, int left, int right) {
    if (left < right) {
        /* Elementul de la pozitia part_index se va afla in pozitia
          corecta din vectorul sortat */
        int part_index = partitionPrim(array, left, right);

        /* Se sorteaza elementele de dinainte de part_index si de dupa. */
        quickSortPrim(array, left, part_index);
        quickSortPrim(array, part_index + 1, right);
    }
}
```




O forma generala pentru complexitatea temporala a algoritmului poate fi descrisa de relatia:

T(n) = T(k) + T(n-k-1) + $\theta$(n); (cele 2 functii T reprezinta cele 2 apeluri recursive)

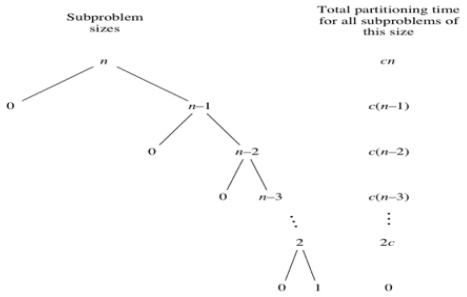
Complexitatea temporala pentru cel mai defavorabil caz:

Cel mai defavorabil caz are loc atunci cand procesul de partitionare alege pivotul ca fiind cel mai mare sau cel mai mic element. In acest caz, relatia generala va avea forma:

T(n) = T(1) + T(n-1) + $\theta$(n), recursivitatea facandu-se pe o singura ramura.

Observam ca T(n−1) = T(1) + T(n−2) + $\theta$(n - 1), inlocuind n cu (n-1) in prima relatie. Substituind termenii se ajunge la relatia: T(n) = T(1) + (n−1)*T(1) + $\theta$( $\sum_{i=0}^{n-2}(n-2)$) = n * T(1) + $\theta$ (n * (n − 2) − (n − 2) * (n − 1) / 2) => Complexitatea algoritmului in cel mai rau caz este O($n^2$).

Complexitatea se poate calcula foarte simplu si cu metoda arborelui de recurenta:




Adunand complexitatile termenilor liberi se ajunge la relatia:

O(n)+O(n−1)+O(n−2)+ ... +O (2)=O(n+(n−1)+(n−2)+ ... +2) = O ((n+1)*(n/2)−1) = O($n^2$)

Complexitatea temporala in cazul mediu a algoritmului este: O(n*logn).

Complexitatea temporala pentru cel mai favorabil caz:

Complexitatea temporala pentru cel mai favorabil caz are loc atunci cand pivotul pe care il alegem imparte la fiecare pas algoritmul in 2 jumatati perfect balansate, astfel incat forma generala devine: T(n) = 2T(n/2)+ $\theta$(n).

Observam ca T(n/2) = 2T(n/4) + $\theta$(n/2), inlocuind n cu n/2 in prima relatie =>

$2^1$ * T(n/$2^1$) = $2^2$T(n/$2^2$) + $2^1\theta$(n/$2^1$). Relatia de recurenta poate fi scrisa:

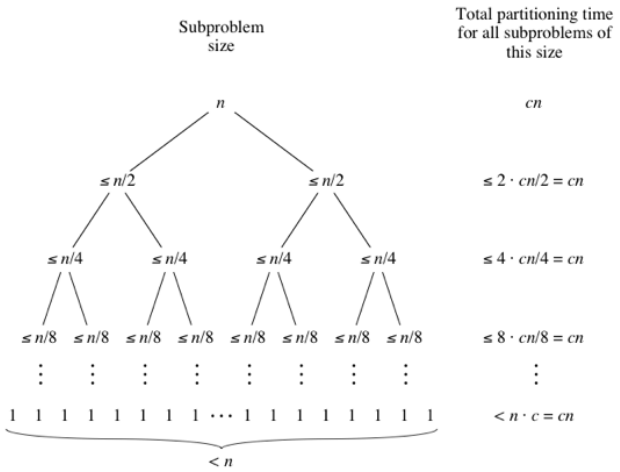
$2^k$T(n/$2^k$) = $2^{k+1}$T(n/$2^{k+1}$) + $2^k\theta$(n/$2^k$).

Folosind metoda iterativa (substituind fiecare functie in parte si adunandu-le) se obtine relatia: T(n) = (h + 1) * $\theta$(n).

n/2*h = 1 => h = logn (recurenta se duce pana in T(1), cand se ajunge la inaltimea “arborelui de recurenta”, notata h).

Folosind cele 2 relatii de mai sus => T(n) = (logn + 1)*$\theta$(n) => Complexitatea in cazul favorabil este $\theta(n * logn)$.

Complexitatea se poate calcula foarte simplu si cu metoda arborelui de recurenta:




Adunand complexitatile termenilor liberi se ajunge la relatia:

O(n)+O(n)+O(n)+ ... +O(n) = (h+1)*O(n) = (logn + 1)*$O$(n) = $O(n * logn)$.

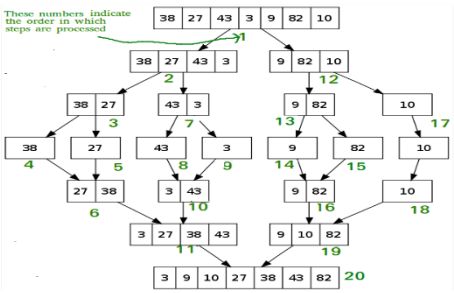
Complexitatea spatiala in cazul cel mai defavorabil este O(n). Este un algoritm “in-place”, partitia facandu-se in O(1) complexitate spatiala, dar daca se ia in considerare memoria utilizata de stiva pentru apelurile recursive avem defapt O(n) in cel mai rau caz, pentru ca se va apela recursiv algoritmul pentru fiecare element al vectorului, pana cand recursivitatea va ajunge la vectori de 1 element.

### 3.1.2 Complexitate MergeSort

Acest subcapitol contine analiza complexitatii algoritmului MergeSort, un algoritm stabil (spre deosebire de QuickSort) bazat pe principiul “divide et impera”.

```
template<typename Type>
void merge(Type* array, int left, int middle, int right) {
    int i = 0, j = 0, k = 0;
    int n1 = middle - left + 1;
    int n2 =  right - middle;
    // Se creeaza 2 array-uri auxiliare si se pun datele in ele
    //Type HalfL[n1], HalfR[n2];
    Type* HalfL = new Type[n1];
    Type* HalfR = new Type[n2];
    for (i = 0; i < n1; i++) {
        HalfL[i] = array[left + i];
    }
    for (j = 0; j < n2; j++) {
        HalfR[j] = array[middle + 1+ j];
    }
    // Se pun elementele din cele 2 array-uri inapoi in vector.
    i = 0; // Indexul pentru primul subvector (HalfL).
    j = 0; // Indexul pentru al doilea subvector (HalfR).
    k = left; // Indexul pentru vectorul interclasat.
    while (i < n1 && j < n2) {
        if (HalfL[i] <= HalfR[j]) {
            array[k] = HalfL[i];
            i++;
        } else {
            array[k] = HalfR[j];
            j++;
        }
        k++;
    }
    // Se copiaza elementele ramase din HalfL (daca mai sunt).
    while (i < n1) {
        array[k] = HalfL[i];
        i++;
        k++;
    }
    // Se copiaza elementele ramase din HalfR (daca mai sunt).
    while (j < n2) {
        array[k] = HalfR[j];
        j++;
        k++;
    }
    delete[] HalfL;
    delete[] HalfR;
}
```

```
/* left si right sunt idecsii pentru subvectorii
   vectorului ce trebuie sortat */
template<typename Type>
void mergeSort(Type* array, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        // Se sorteaza cele 2 jumatari.
        mergeSort(array, left, middle);
        mergeSort(array, middle + 1, right);

        merge(array, left, middle, right);
    }
}
```



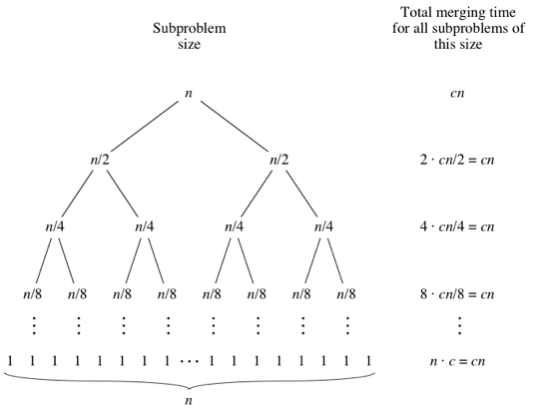

Un lucru foarte important de mentionat despre acest algoritm este faptul ca in toate cele 3 cazuri (cazul defavorabil, mediu sau favorabil) complexitatea temporala este O(n*logn). Acest lucru reiese din faptul ca algoritmul mereu imparte vectorul dat in 2 subvectori de dimensiuni egale si dureaza un timp liniar pentru a unii cele 2 jumatati (operatia de interclasare – “merge”).

Complexitatea temporala a algoritmului va avea mereu relatia de recurenta:

T(n) = 2T(n/2)+ $\theta$(n)

Aplicand acceasi metoda ca in cazul favorabil de la QuickSort (descris anterior, in capitolul 3.1.1) => complexitatea temporala a algoritmului este O(n*logn).

Complexitatea se poate calcula foarte simplu si cu metoda arborelui de recurenta:




Adunand complexitatile termenilor liberi se ajunge la relatia:

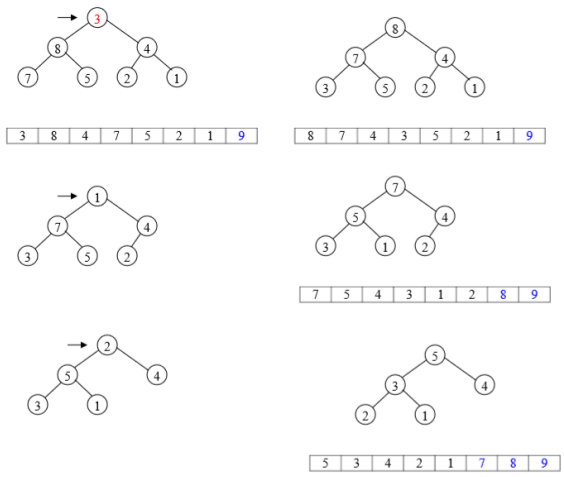
O(n)+O(n)+O(n)+ ... +O(n) = (h+1)*O(n) = (logn + 1)*$O(n) = O(n * logn)$.

Complexitatea spatiala in cel mai rau caz este O(n), pentru a putea fi realizat merge-ul subvectorilor. Chiar daca la fiecare apel recursiv se creeaza 2 subvectori, acestia nu coexista, utilizandu-se doar O(n) memorie auxiliara.

### 3.1.3 Complexitate HeapSort

Acest subcapitol contine analiza complexitatii algoritmului HeapSort.

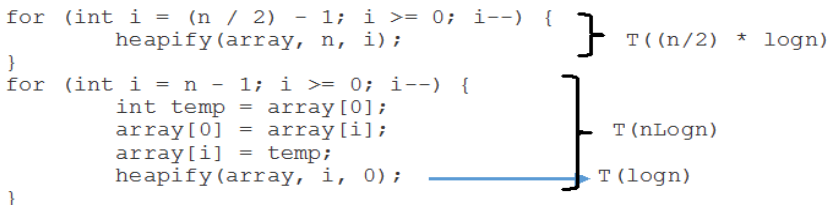
```
// Functia de reordonare a heap-ului.
template<typename Type>
void heapify(Type* array, int n, int i) {
    int largest = i;  // Largest va fi root
    int left = 2 * i + 1;  // left = 2*i + 1
    int right = 2 * i + 2;  // right = 2*i + 2

    // Cazul cand copilul stang este mai mare ca root.
    if (left < n && array[left] > array[largest]) {
        largest = left;
    }

    // Cazul cand copilul drept este mai mare ca root.
    if (right < n && array[right] > array[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(array[i], array[largest]);

        // Se apeleaza recursiv heapify pentru subtree.
        heapify(array, n, largest);
    }
}

// Functia pentru Heap Sort.
template<typename Type>
void heapSort(Type* array, int n) {
    // Se construieste heap-ul.
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(array, n, i);
    }

    // Se extrag elementele unul cate unu si se rearanjeaza heap-ul.
    for (int i = n - 1; i >= 0; i--) {
        // Se muta roo-ul la final.
        swap(array[0], array[i]);

        heapify(array, i, 0);
    }
}
```




Un lucru foarte important de mentionat despre acest algoritm este ca complexitatea spatiala este O(1), utilizand putina memorie. Chiar daca este mai lent decat QuickSort si MergeSort, acest algoritm “impaca” viteza cu consumul relativ mic de memorie.

Complexitatea temporala in fiecare caz (cazul defavorabil, mediu sau favorabil) este O(n*logn). Complexitatea temporala pentru crearea unui Heap este O(n), iar complexitatea toala a algoritmului este O(n*logn).

Metoda “heapify” verifica daca proprietatea heap-ului este mentinuta in O(logn).




Conform imaginii de mai sus (corespunde codului meu, eu folosind functia “swap” pentru interschimbare), deducem relatia de recurenta pentru HeapSort:

T(n) = T((n/2) * logn) + T(n * logn) = 2 * T(n*logn) = O(n*logn) => Complexitatea temporala a algoritmului HeapSort este O(n*logn).

Complexitatea temporala poate fi si intuita. Heap-ul se construieste in O(n) si se vor face n-1 extrageri. La fiecare extragere se va apela metoda “heapify” pentru a vedea daca proprietatea Heap-ului se respecta, metoda “heapify” avand complexitatea O(logn). Din acestea rezulta ca complexitatea algoritmului HeapSort va fi:

O(n) + (n-1) * O(logn) = O(n*logn);

Complexitatea spatiala in cel mai defavorabil caz este O(1) (cum am mentionat si la inceputul subcapitolului), nefolosindu-se vectori auxiliari, HeapSort fiind un algoritm in-place.

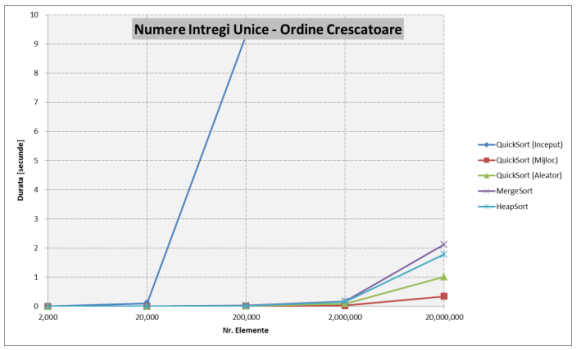
## 3.2 Testarea si analiza algoritmilor

Acest subcapitol contine analiza si datele rezultate in urma testarii efective a algoritmilor. Testarea a fost realizata pe o masina virtuala cu un sistem de operare Linux Ubuntu 17.04. Masina virtuala a avut o memorie RAM de 4GB si a fost rulata pe un calculator cu un sistem de operare Windows 10, 8GB RAM si procesoz Intel Core i7.

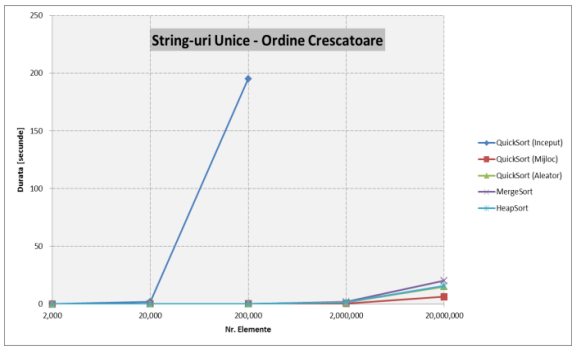
Pentru testare s-au folosit 60 de teste cu dimensiuni si tipuri de date diferite. Testele au fost generate cu ajutorul programului “Spawner” (se gaseste si in arhiva aferenta lucrarii), testele de dimensiuni pana intr-un milion de intrari au fost editate cu ajutorul programului “Microsoft Excel”, iar cele de dimensiuni mai mari au fost editate cu ajutorul programului “Microsoft Access”. De asemenea, am folosit programul “EditPat Lite 7” pentru a putea deschide si vizualiza fisierele (Notepad++ functiona foarte greu pentru fisierele cu dimensiuni foarte mari). Pentru implementarea algoritmilor s-a folosit limbajul C++.

Cele 60 de teste folosite pentru testarea programului contin intrari de tip “int”, “string” si un tip de data custom (“int” si “string”). Pentru fiecare tip de date au fost create 20 de teste. Acestea trateaza urmatoarele 4 cazuri: vector cu elemente unice sortate crescator (favorabil), vector cu elemente unice sortate descrescator (defavorabil), vector cu elemente unice random (adica nesortat), vector cu elemente duplicate random (tot nesortat, dar contine foarte multe duplicate). Pentru fiecare caz in parte, am creat cate 5 teste de dimensiuni diferite: 2.000 de intrari, 20.000 intrari, 200.000 intrari, 2.000.000 intrari, 20.000.000 intrari. Cu alte cuvinte, avem 5 teste pentru fiecare caz in parte => 20 de teste pentru un tip de date. Cum noi avem 3 tipuri de date => un total de 60 de teste. Programul pentru testare poate fi gasit in arhiva aferenta lucrarii. Algoritmii au fost scrisi in fisiere header si sunt apelati de catre Main.cpp. La rulare, utilizatorului i se cere sa introduca fisierul de intrare, de iesire si algoritmul ce trebuie testat.

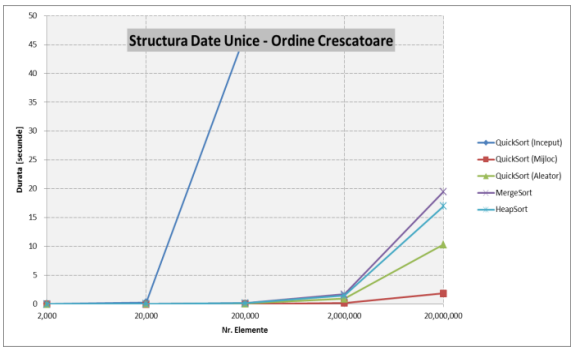
### 3.2.1 Teste cu elemente unice ordonate crescator



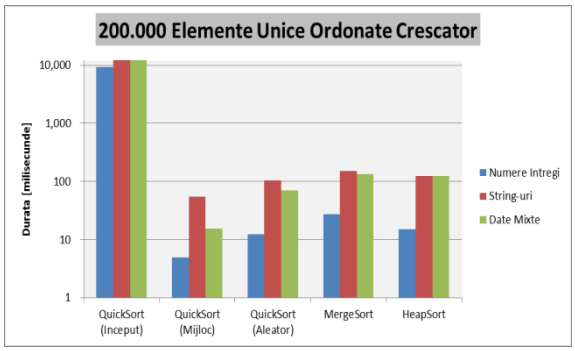

| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 1,125 | 28 | 73 | 181 | 156 |
| 20,000 | 105,873 | 260 | 1,007 | 1,621 | 1,502 |
| 200,000 | 9,312,599 | 4,991 | 12,381 | 27,535 | 15,200 |
| 2,000,000 | | 28,725 | 93,416 | 173,679 | 165,028 |
| 20,000,000 | | 345,004 | 1,011,997 | 2,125,999 | 1,786,153 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 23,868 | 427 | 987 | 1,211 | 745 |
| 20,000 | 1,867,835 | 3,890 | 13,704 | 17,864 | 10,741 |
| 200,000 | 195,204,671 | 54,727 | 105,410 | 151,950 | 125,371 |
| 2,000,000 | | 534,258 | 1,337,221 | 1,749,300 | 1,368,378 |
| 20,000,000 | | 6,217,203 | 15,130,769 | 20,105,064 | 15,587,186 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 2,596 | 136 | 607 | 1,412 | 874 |
| 20,000 | 237,618 | 1,383 | 9,901 | 18,289 | 12,849 |
| 200,000 | 46,832,516 | 15,499 | 69,865 | 135,331 | 124,707 |
| 2,000,000 | | 165,338 | 892,194 | 1,689,985 | 1,458,524 |
| 20,000,000 | | 1,850,589 | 10,305,770 | 19,460,338 | 16,975,820 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Numere Intregi | 9,312,599 | 4,991 | 12,381 | 27,535 | 15,200 |
| String-uri | 195,204,671 | 54,727 | 105,410 | 151,950 | 125,371 |
| Date Mixte | 46,832,516 | 15,499 | 69,865 | 135,331 | 124,707 |

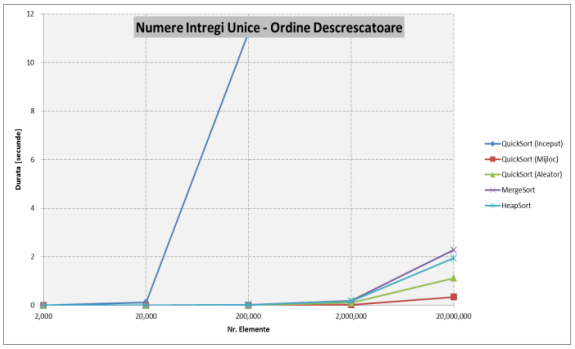
Fiecare tabel contine durata algoritmilor in microsecunde pentru fiecare tip de date pentru cazul in care elementele sunt unice si sortate crescator. Fiecare tabel are atasat si un grafic.

Daca ne uitam la fiecare tabel in parte observam un lucru foarte interesant. Pentru fiecare tip de data si pentru orice dimensiune a fisierului de intrare, QuickSort cu pivotul elementul din mijloc a avut timpii cei mai buni (colorati cu verde in tabele) acesta fiind de altfel si cazul cel mai favorabil pentru acest pivot, dar in acelasi timp, QuickSort-ul ce isi alege ca pivot cel mai din stanga element a avut cele mai slabe rezultate (colorate cu rosu), fiind un caz defavorabil pentru acesta. Putem afirma nu doar ca QuickSort (Inceput) a avut rezultate mai slabe, ci timpii cresteau asa de mult incat algoritmul nu a putut fi testat decat pentru testele cu maxim 200.000 de intrari (ar fi durat mult prea mult). Din aceste lucruri reise clar ca alegerea pivotului este un element cheie pentru QuickSort. QuickSort (Aleator) a obtinut rezultate evident mai bune decat QuickSort (Inceput), dar nici pe departe la fel de bune precum QuickSort (Mijloc). HeapSort si MergeSort au avut rezultate comparabile, diferentele mari incepand sa fie vizibile la testele cu 20.000.000 de intrari (HeapSort avand timpi mai buni cu pana la 5 secunde la datele de tip “string”).

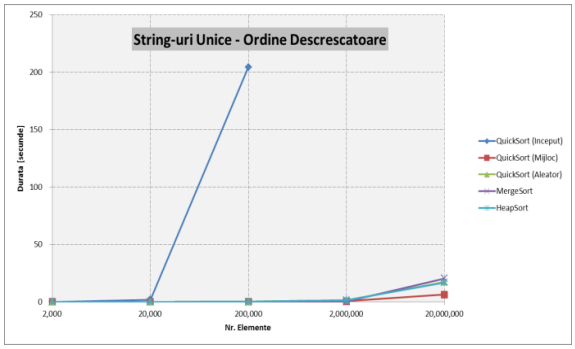
Conform rezultatelor, cel mai mult timp a durat sortarea elementelor de tip “string”, indiferent de algoritm. Datele “StructCustom” au fost sortate mai intai dupa “int” si apoi dupa “string”.

Performante: QuickInceput << MergeSort < HeapSort < QuickAleator < QuickMijloc

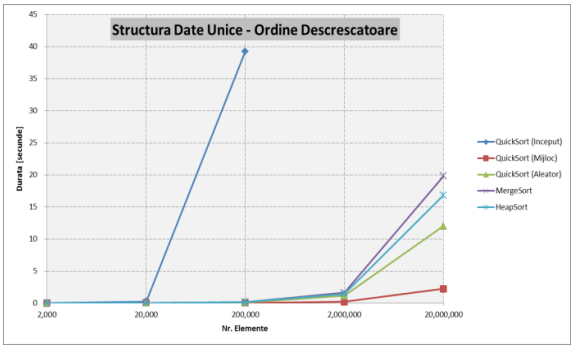
### 3.2.2 Teste cu elemente unice ordonate descrescator



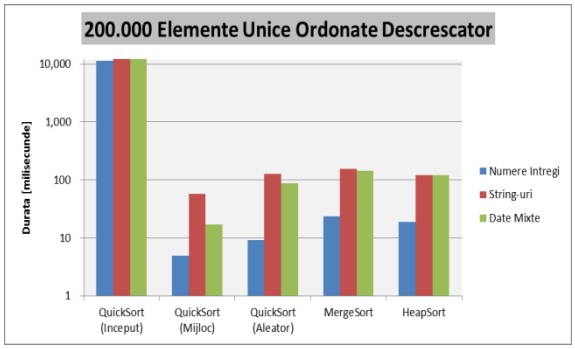

| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 1,259 | 41 | 93 | 187 | 134 |
| 20,000 | 120,773 | 260 | 979 | 1,931 | 1,730 |
| 200,000 | 11,242,383 | 4,928 | 9,227 | 23,367 | 18,966 |
| 2,000,000 | | 29,989 | 103,381 | 182,143 | 175,285 |
| 20,000,000 | | 346,345 | 1,123,915 | 2,282,442 | 1,937,365 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 25,525 | 409 | 1,138 | 1,357 | 729 |
| 20,000 | 1,877,587 | 4,494 | 13,570 | 13,523 | 10,886 |
| 200,000 | 204,356,781 | 58,231 | 127,425 | 154,132 | 121,565 |
| 2,000,000 | | 538,458 | 1,569,637 | 171,003 | 1,408,602 |
| 20,000,000 | | 6,237,817 | 17,235,708 | 20,389,650 | 16,960,103 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 2,255 | 172 | 654 | 1,333 | 813 |
| 20,000 | 212,480 | 1,725 | 10,979 | 16,419 | 10,924 |
| 200,000 | 39,250,151 | 17,022 | 86,768 | 142,754 | 121,697 |
| 2,000,000 | | 196,430 | 1,141,159 | 1,596,127 | 1,444,028 |
| 20,000,000 | | 2,226,870 | 12,007,793 | 19,831,414 | 16,798,038 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Numere Intregi | 11,242,383 | 4,928 | 9,227 | 23,367 | 18,966 |
| String-uri | 204,356,781 | 58,231 | 127,425 | 154,132 | 121,565 |
| Date Mixte | 39,250,151 | 17,022 | 86,768 | 142,754 | 121,697 |

Duratele de executie ale algoritmilor in microsecunde pentru fiecare tip de date (pentru cazul in care elementele sunt unice si sortate descrescator) se gasesc in tabele si graficele aferente.

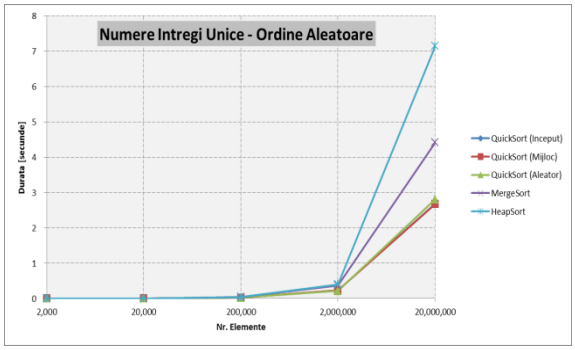
Rezultatele sunt foarte asemanatoare cu cele din subcapitolul anterior, in cazul elementelor unice sortate crescator. QuickSort (Inceput) are rezultate “dezastruoase” comparativ cu ceilalti algoritmi (neputand fii testat decat pentru testele de pana in 200.000 de intrari), testele reprezentand si un caz defavorabil, iar QuickSort (Mijloc) are din nou rezultatele cele mai bune.

Toti timpii sunt asemanatori cu timpii din cazul elementelor sortate crescator, chiar si pentru MergeSort si HeapSort, pentru care un vector deja sortat in ordinea corecta este un caz favorabil. Acest lucru poate fi justificat de faptul ca algoritmii au complexitatea O (n * logn) in toate cazurile. Daca tot vorbim de complexitati, prin faptul ca QuickSort (Inceput) nu a putut fi testat pentru toate testele confirma faptul ca complexitatea acestuia pentru un caz nefavorabil este O($n^2$).

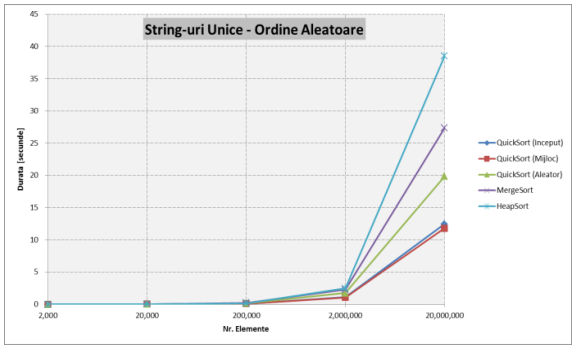
Conform rezultatelor, cel mai mult timp a durat sortarea elementelor de tip “string”, indiferent de algoritm, iar cel mai bun timp a fost inregistrat pentru sortarea datelor de tip “int” (diferenta este de ordinul zecilor).

Performante: QuickInceput << MergeSort < HeapSort < QuickAleator < QuickMijloc

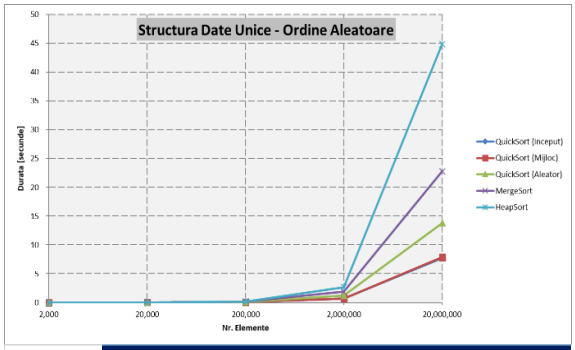
### 3.2.3 Teste cu elemente unice in ordine aleatoare



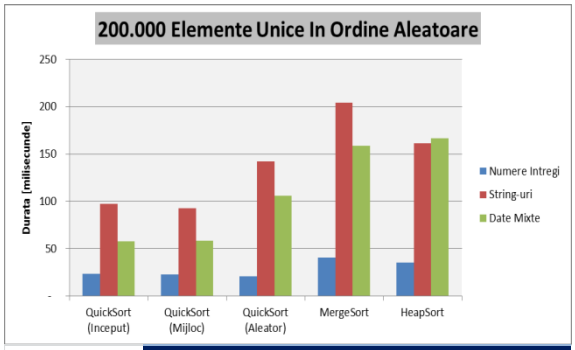

| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 166 | 165 | 153 | 302 | 178 |
| 20,000 | 1,721 | 2,072 | 1,799 | 3,537 | 2,274 |
| 200,000 | 23,708 | 23,032 | 20,499 | 40,712 | 35,588 |
| 2,000,000 | 229,832 | 233,176 | 217,838 | 370,934 | 397,669 |
| 20,000,000 | 2,681,491 | 2,674,121 | 2,804,712 | 4,417,094 | 7,142,021 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 682 | 728 | 1,252 | 1,421 | 843 |
| 20,000 | 9,759 | 8,637 | 17,594 | 20,684 | 11,529 |
| 200,000 | 97,240 | 92,701 | 142,134 | 203,969 | 161,626 |
| 2,000,000 | 1,081,346 | 1,050,858 | 1,748,898 | 2,272,716 | 2,470,235 |
| 20,000,000 | 12,438,008 | 11,786,196 | 19,845,622 | 27,333,373 | 38,499,447 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 470 | 451 | 710 | 1,308 | 976 |
| 20,000 | 4,616 | 5,634 | 10,813 | 13,029 | 13,919 |
| 200,000 | 57,841 | 58,122 | 106,235 | 158,653 | 166,296 |
| 2,000,000 | 660,007 | 696,305 | 1,204,226 | 1,865,169 | 2,687,494 |
| 20,000,000 | 7,741,740 | 7,839,935 | 13,799,598 | 22,775,115 | 44,885,568 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Numere Intregi | 23,708 | 23,032 | 20,499 | 40,712 | 35,588 |
| String-uri | 97,240 | 92,701 | 142,134 | 203,969 | 161,626 |
| Date Mixte | 57,841 | 58,122 | 106,235 | 158,653 | 166,296 |

Tabelele si graficele contin durata algoritmilor in microsecunde pentru cazul in care elementele sunt unice si aleatoare.

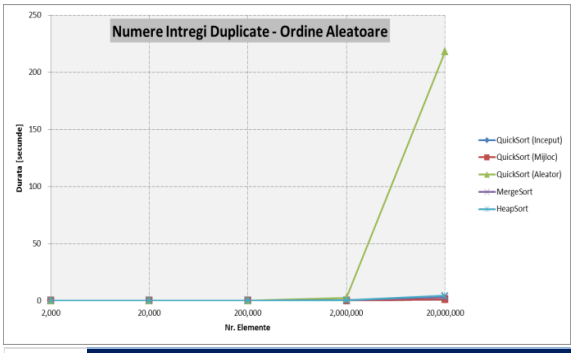
In ceea ce priveste performanta algoritmilor pe testele cu elemente unice in ordine aleatoare, putem afirma ca algoritmul QuickSort intrece algoritmii MergeSort si HeapSort in toate testele. QuickSort este mai rapid, dar alegerea pivotului avantajeaza anumite cazuri. De exemplu, QuickSort (Aleator) a avut cele mai bune rezultate pentru testele de tip “int”, QuickSort(Mijloc) a avut cele mai bune rezultate pentru datele de tip “string”si QuickSort(Inceput) a avut cele mai bune rezultatele pentru tipurile de date mixte.

Se poate realiza si o comparatie intre MergeSort si HeapSort. Se observa ca pentru testele de dimensiuni mici, HeapSort este mai rapid decat MergeSort, dar pentru testele de dimensiuni foarte mari, MergeSort se comporta mult mai bine (pentru tipul de date custom, pentru cel mai mare test sunt diferente de 12 secunde).

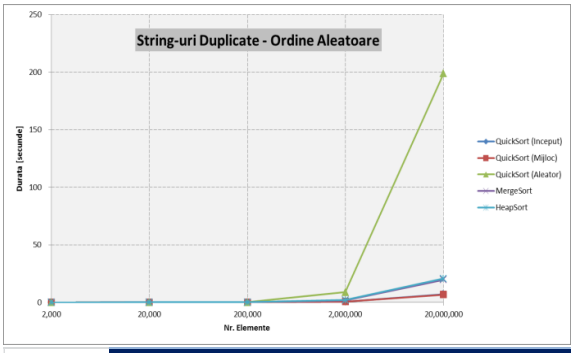
Un lucru imporant de mentionat este faptul ca toti algoritmii au avut timpi mai mari de executie pentru testele de tip “string”, in afara de HeapSort care a avut un timp foarte mare pentru datele custom (un timp de executie de 2 ori mai mare fata de urmatorul cel mai lent algoritm).

Performante: Depinde de caz, dar QuickSort (indiferent de pivotul ales in aceste cazuri) este mai rapid decat MergeSort si HeapSort, iar HeapSort este cel mai lent dintre algoritmi pentru volume mari de date.

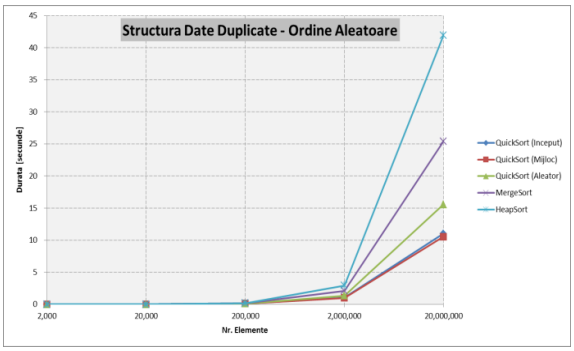
### 3.2.4 Teste cu elemente duplicate in ordine aleatoare



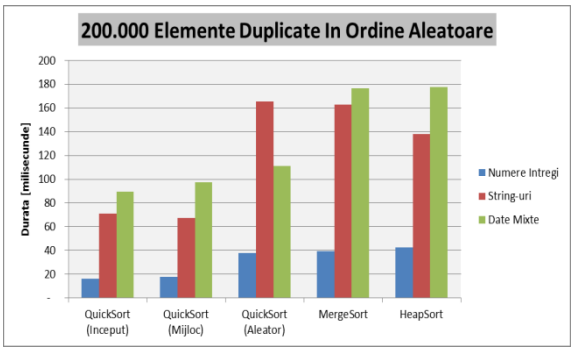

| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 234 | 157 | 149 | 303 | 198 |
| 20,000 | 1,564 | 1,607 | 1,590 | 4,193 | 2,260 |
| 200,000 | 16,087 | 17,646 | 37,993 | 39,456 | 42,739 |
| 2,000,000 | 124,924 | 130,277 | 2,280,329 | 301,428 | 316,472 |
| 20,000,000 | 1,325,604 | 1,322,526 | 218,194,469 | 3,190,836 | 4,384,170 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 690 | 650 | 831 | 1,335 | 876 |
| 20,000 | 6,975 | 7,461 | 13,167 | 20,211 | 12,622 |
| 200,000 | 70,917 | 67,219 | 165,655 | 162,923 | 137,818 |
| 2,000,000 | 640,446 | 630,968 | 8,763,958 | 1,744,812 | 1,940,112 |
| 20,000,000 | 7,141,300 | 6,813,361 | 198,808,904 | 19,610,007 | 20,666,855 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Nr. Elemente | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| 2,000 | 487 | 507 | 871 | 1,074 | 860 |
| 20,000 | 6,597 | 8,868 | 10,905 | 13,649 | 13,726 |
| 200,000 | 89,565 | 97,444 | 111,028 | 176,833 | 177,863 |
| 2,000,000 | 1,036,195 | 989,640 | 1,341,716 | 2,096,069 | 2,935,519 |
| 20,000,000 | 11,039,615 | 10,535,630 | 15,568,971 | 25,416,650 | 41,967,451 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Numere Intregi | 16,087 | 17,646 | 37,993 | 39,456 | 42,739 |
| String-uri | 70,917 | 67,219 | 165,655 | 162,923 | 137,818 |
| Date Mixte | 89,565 | 97,444 | 111,028 | 176,833 | 177,863 |

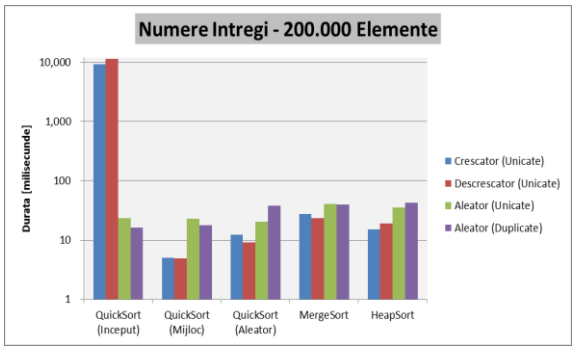
In fiecare tabel avem durata algoritmilor in microsecunde pentru fiecare tip de date pentru cazul in care elementele sunt duplicate si aleatoare.

In aceste teste, QuickSort (Inceput) si QuickSort (Mijloc) au rezultate asemanatoare, fiind cei mai rapizi algoritmi. Un lucru diferit fata de celalalte subcapitole este faptul ca pe date de tip “int” si pe date de tip “string”, cand sunt multe date de intrare si in vector sunt foarte multe elemente duplicate, algoritmul QuickSort (Aleator) are rezultate foarte slabe, fiind mult mai lent chiar si fata de MergeSort si HeapSort, inclusiv pe testele de dimensiuni mari. In ceea ce priveste tipul de date custom, ca si in anteriorul subcapitol, HeapSort are cele mai slabe rezultate, mai ales pe testele de dimensiune mare.

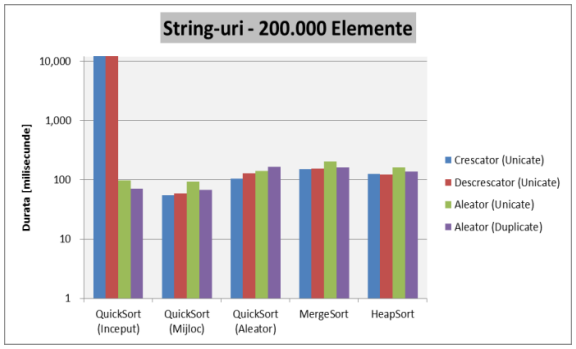
Un lucru important de observat este faptul ca MergeSort are mereu cei mai mari timpi de executie (pentru aceste teste) pentru testele cu putine intrari (2.000 si 20.000), deci nu ar fi o alegere buna daca avem putine elemente de sortat si foarte multe duplicate.

Performante: Depinde de caz. Pentru cazuri in care avem putine elemente trebuie evitata folosirea MergeSort. Pentru date de dimensiuni mari de tip “int” sau “string” trebuie evitata folosirea QuickSort cu pivot random, iar pentru date de tip custom trebuie evitat HeapSort-ul (mai ales daca sunt foarte multe intrari). Cei mai buni algoritmi au fost QuickSort (Inceput) si QuickSort(Middle), avand timpi asemanatori.

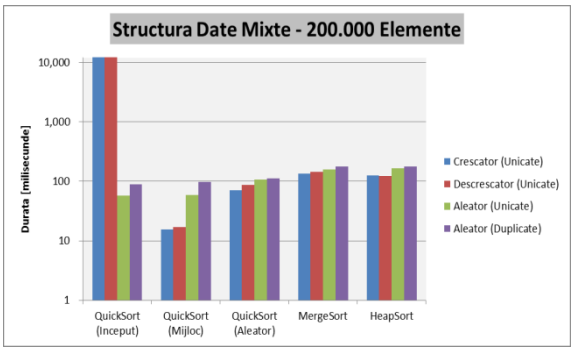
### 3.2.5 Analiza pe tipuri de date pentru 200.000 de elemente




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Crescator (Unicate) | 9,312,599 | 4,991 | 12,381 | 27,535 | 15,200 |
| Descrescator (Unicate) | 11,242,383 | 4,928 | 9,227 | 23,367 | 18,966 |
| Aleator (Unicate) | 23,708 | 23,032 | 20,499 | 40,712 | 35,588 |
| Aleator (Duplicate) | 16,087 | 17,646 | 37,993 | 39,456 | 42,739 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Crescator (Unicate) | 195,204,671 | 54,727 | 105,410 | 151,950 | 125,371 |
| Descrescator (Unicate) | 204,356,781 | 58,231 | 127,425 | 154,132 | 121,565 |
| Aleator (Unicate) | 97,240 | 92,701 | 142,134 | 203,969 | 161,626 |
| Aleator (Duplicate) | 70,917 | 67,219 | 165,655 | 162,923 | 137,818 |




| | Durata [microsecunde] | | | | |
|---|---|---|---|---|---|
| Ordonare Initiala | QuickSort (Inceput) | QuickSort (Mijloc) | QuickSort (Aleator) | MergeSort | HeapSort |
| Crescator (Unicate) | 46,832,516 | 15,499 | 69,865 | 135,331 | 124,707 |
| Descrescator (Unicate) | 39,250,151 | 17,022 | 86,768 | 142,754 | 121,697 |
| Aleator (Unicate) | 57,841 | 58,122 | 106,235 | 158,653 | 166,296 |
| Aleator (Duplicate) | 89,565 | 97,444 | 111,028 | 176,833 | 177,863 |

Pentru a analiza algoritmii din inca un punct de vedere, am ales sa analizez cum se comporta toti algoritmii pe fiecare tip de date, in functie de ordinea elementelor din vector si unicitatea lor, pentru testele mele “medii” de 200.000 de elemente.

Analizand cele 3 tabele si graficele lor aferente, reiese clar faptul ca pentru cazurile limita (elemente sortate crescator sau descrescator), indiferent de tipul de date, QuickSort (Inceput) este de departe cel mai ineficient algoritm, dar in acelasi timp, QuickSort (Mijloc) este cel mai eficient algoritm.

Pentru datele de tip “int” cel mai bun algoritm este QuickSort, dar pivotul trebuie ales cu mare grija, in functie de datele de intrare. MergeSort este cel mai ineficient cand datele sunt unice si aleatoare, iar HeapSort este cel mai ineficient cand datele sunt aleator ordonate si duplicate.

Pentru datele de tip “string” cel mai bun algoritm este de departe QuickSort (Mijloc). Cand avem elemente aleatoare unice, cel mai ineficient este MergeSort. Cand avem elemente aleatoare duplicate, cel mai ineficient este QuickSort (Random).

Pentru datele de tip “StructCustom” (datele mixte), QuickSort (Mijloc) este cel mai eficient in cazurile limita, iar QuickSort (Inceput) este cel mai eficient in cazul in care elementele sunt aleatoare.

Daca analizam toate tabelele din subcapitolele anterioare, putem observa faptul ca desi MergeSort nu este niciodata cel mai eficient, are cele mai mici variatii de timp in functie de ordinea de intrare a elementelor (chiar daca sunt cazuri limita sau sunt duplicate).

HeapSort si MergeSort nu sunt cei mai eficienti algoritmi in niciunul din cazuri.

# 4. Concluzii

In acest raport am analizat 3 algoritmi de sortare: QuickSort (cu 3 modalitati de alegere a pivotului: cel mai din stanga element, elementul din mijloc si un element random), MergeSort si HeapSort.

Sortarea este una dintre cele mai des intalnite subprobleme in programare. Aceasta constituie o parte esentiala din numeroasele procese de prelucrare a datelor, iar din cauza ca in zilele noastre se lucreaza cu cantitati imense de date, eficienta algoritmilor de sortare este o problema de actualitate.

In urma multitudinii de teste pe care le-am efectuat si in urma complexitatilor determinate, putem afirma urmatorul lucru: nu exista un algoritm perfect de sortare. Fiecare algoritm are avantaje si dezavantaje si este mai bun in unele cazuri decat altele.

Algoritmul care s-a dovedit a fi cel mai potrivit in cele mai multe din cazuri a fost algoritmul QuickSort care alege ca pivot elementul din mijloc. MergeSort are avantajul ca este un algoritm stabil, iar timpii lui de executie au cele mai mici variatii in functie de ordinea elementelor de intrare. HeapSort are complexitatea spatiala in cel mai defavorabil caz O(1), deci daca avem o cantitate limitata de memorie HeapSort poate fi o alternativa.

In multe cazuri QuickSort este rapid indiferent de pivotul ales, dar totusi trebuie sa alegem pivotul cu mare grija, pentru ca altfel putem intampina mari probleme (cazurile cand QuickSort (Inceput) avea timp extrem de mari si nici macar nu a putut fi testat). HeapSort este ineficient pe cantitati mari de date (aproape mereu a fost cel mai ineficient algoritm pe testele cu multe intrari, in afara de cazurile limita). Exceptand cazurile limita pentru QuickSort, MergeSort a fost mereu cel mai ineficient algoritm cand sunt putine elemente de sortat.

In ceea ce priveste tipurile diferite de date, trebuie sa ii dezamagesc pe cei ce sustin ideea ca algoritmii de sortare au acelasi timp de executie pentru orice tip de date. Conform testelor efectuate, timpii difera extraordinar de mult, cea mai mare diferenta fiind intre timpii pentru sortarea datelor de tip “int” si datelor de tip “string”. Alegerea algoritmului potrivit trebuie sa tina seama si de tipul de date ce trebuie sortate.

In urma analizei, recomandarile mele personale sunt in principal urmatoarele:

- Alegerea atenta a pivotului pentru QuickSort. Din testele mele, un pivot adecvat ar fi elementul din mijloc, iar un pivot inadecvat ar fi primul element, pentru ca poate aparea cazul defavorabil O($n^2$).
- Daca nu sunt limitari de memorie, cel mai bun algoritm (in cele mai multe cazuri) din cele mai multe puncte de vedere este QuickSort cu pivot element din mijloc (dintre algoritmii analizati in acest raport).
- Evitarea folosirii HeapSort pentru cantitati foarte mari de date. Este o solutie buna in momentul in care nu aveti multa memorie la dispozitie.
- Evitarea folosirii MergeSort pentru cantitati mici de date. Daca nu se stie cum arata datele de intrare acest algoritm este o optiune, pentru ca are cele mai mici variatii in privinta timpului de executie in functie de ordinea elementelor de intrare.

In opinia mea, “Sortarea” este unul dintre cele mai importante subiecte din domeniul programarii. Chiar daca nu pare o problema de o importanta majora, foarte multe probleme mari au performanta lor strans legata de eficienta algoritmilor de sortare.

Cu siguranta in viitor vor aparea algoritmi tot mai performanti pentru sortarea diferitelor tipuri de date, ceea ce va contribui in continuare la dezvoltarea multor domenii stiintifice si la procesarea rapida a volumelor tot mai mari de date.

# Bibliografie


- Analiza Algoritmilor (curs) (AA) - Prof. Șt. TRĂUȘAN-MATU
- Analiza Algoritmilor (seminar) (AA) – Stefania Budulan
- www.academia.edu/31480109/ANALIZA_COMPARATIVĂ_A_COMPLEXITĂȚII_ALGORITMILOR_DE_SORTARE
- Parallel Sorting Algorithms din revista “Informatica Economica”, Asist. Felician ALECU
- http://staff.cs.upt.ro/~chirila/teaching/upt/id12-sda/lectures/ID-SDA-Cap3.pdf
- http://opendatastructures.org/ods-java/11_1_Comparison_Based_Sorti.html
- www.gdeepak.com/thesisme/Thesis-Selection%20of%20Best%20Sorting%20Algorithm%20for%20a%20Particular.pdf
- http://scanftree.com/Data_Structure/time-complexity-and-space-complexity-comparison-of-sorting-algorithms
- http://warp.povusers.org/SortComparison/index.html
- https://stackoverflow.com/
- http://www.geeksforgeeks.org/
- https://www.hackerearth.com/practice/algorithms/sorting/
- http://z-sword.blogspot.ro/2014/02/complexity-of-sorting.html
- https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html