# ANALIZA ALGORITMILOR

# -TEMA1-

# SORTARI

# CUPRINS

# 1.Punctul de plecare.

## 1.1. Motivatie

Am ales sa vorbesc despre sortari pe de o parte, datorita folosirii acestora in foarte multe aplicatii de actualitate, iar pe de alta parte datorita, diferitelor implementari ce urmaresc anumite “taskuri”(complexitate mica pe un set de date mare, usurinta in intelegerea algoritmului in sine).

## 1.2.Descrierea problemei

Sortarile ,in general au acelasi scop. Punerea intr-o anumita ordine a unor elemente de un anumit tip specific.(Ex : ordonarea notelor unor elevi in ordine crescatoare). Insa intervine problema urmatoare. Exista posibilitatea ca datele noastre de intrare sa fie foarte mari sau foarte mici..De asemenea, pot exista cazuri cand datele de intrare pot fi ,,aproape sortate”( **Sorting “almost sorted” lists.**) s.a.m.d . Avand in vedere aceste situatii, algoritmi existenti de sortare obtin o complexitate adecvata( mai mica de $O(n^2)$) in anumite situatii. Prin urmare, problema se pune astfel : ,, Cum as putea sorta, cat mai efficient, un sir de date random ?”

## 1.3 Specificarea solutiilor alese

In aceasta lucrarea vor fi tratati 3 algoritmi de sortare diferiti, cu complexitati diferite ce lucreaza cu tipuri de date de diferite dimensiuni. Prin aceasta comparatie se va stabili aplicabilitatea diferitor algoritmi.

Se da un sir de N numere si se pune problema sortarii acestora in ordine crescatoare. Inputurile au diferite proprietati care ne ajuta in alegerea algoritmului cu performantele cele mai bune.

Daca datele noastre de intrare sunt repetitive (ex: 1 4 13 1 4 14), se foloseste **CountingSort**. Acest algoritm creeaza un vector de frecventa **arrayFreq** al elementelor date ca input. Odata creat, se parcurge vectorul si se insumeaza fiecare element cu frecventa precedenta,. Se creeaza apoi un vector **newArray** cu dimensiunea egala cu frecventa maxima.In final,se parcurge din nou inputul iar fiecare element va fi pus in **newArray** pe pozitia egala cu frecventa din **arrayFreq,**de la pozitia egala cu elementul citit. Ulterior se decrementeaza frecventa si se trece mai departe.Acest algoritm este foarte eficient,avand o complexitate de O(n + k), unde **n** este numarul de elemente, iar **k** este intervalul in care se gasesc numerele din input. Daca elementele sunt dintr-un interval al carui maxim superior este foarte mare, vor aparea probleme de efecienta din cauza memoriei(Ex: 10 100 12 10 10 22 -Se aloca un vector cu multe elemente **nule**) si a informatiei inutile din vectorl **arrayFreq**( 0 reprezinta informatie inuitla).

Daca datele noastre de intrare sunt aproape sortat(ex: 1 2 3 5 4 6 7 ) se foloseste **ShellSort.** Este un algoritm asemanator cu **InsertionSort**, insa este mult mai eficient deoarece elementele nu vor mai fi verificate unul cate unul. Se va alege un **gap**, ce reprezinta din cat in cat verificam elemente(Spre ex: daca **gap** = 4 , vom verifica elementele din 4 in 4 incepand de pe pozitia 0.).Practic se vor face comparatii intre elemente de pozitia **i,** si **i+gap**.Daca **i** > **i+gap** se va face **swap** pe cele doua elemente.In caz contrar se trece mai departe. Algoritmul se repeat pana cand **gap** va fi egal cu 0. Initial acesta este egal cu **n/2,** unde **n** reprezinta numarul de elemente din vector ce trebuie sortat.La urmatoarea iteratie, **gap** va fi egal cu **previous_gap/ 2**, unde

**previous_gap** reprezinta **gap** de la pasul anterior. Algoritmul ofera o complexitate de O(nlogn) in the best case, iar in the worst case ofera o complexitate de $O(n^2)$ (greu de atins deoarece acest algoritm pleaca de la InsertionSort ce are complexitate $O(n^2)$ si scade treptat avand in vedere ca **gap** nu este egal cu 1 ci cu n/2).

Daca datele de intrare difera, fiind pur si simplu un set de numere random, se prefera folosirea unui algoritm *hibird*, **IntroSort.**Un algoritm hibribd are la baza mai multi algoritmi ,iar in functie de datele de intrare se aplica un algoritm anume mai intai. Acesta are la baza **QuickSort**, **HeapSort** si **InsertionSort.** Mai intai sortarea este facuta cu **QuickSort.**Daca recursivitatea depaseste o anumita limita stabilita la intrarea, se trece **HeapSort**, deoarece se doreste evitarea complexitatii $O(n^2)$ a **QuikSor**t pe **worst case**. Cand numarul de elemente de sortat este foarte mic, se aplica **InsertionSort** . Apar astfel 3 situatii :

- Daca partitia este mai mare decat limita impus(2*log(n)) **IntroSort** trece la **HeapSort**
- Daca partita este mult prea mica, **QuicSort** trece la **InsertionSort**
- Daca partitia este sub limita ,dar nu este prea mica atunci se va face **QuickSort**

## 1.4. Evaluarea solutiei alese

Pentru cei 3 algoritmi propusi se vor testa urmatoarele : **bestCase**, **midCase** si **worstCase.** Datele vor fi alese pentru a determina un runtime pentru fiecare algoritm.De asemenea, se va testa si acelasi set de date pentru a descoperi care algoritm este cel mai fiabil pentru un set de date random. Datele de intrare vor avea dimensiuni diferite pentru a putea trasa un graphic al eficientei pentru fiecare algoritm..