**Tutoriat 1**

Cuprins:

1. Discuții sortări
2. Introducere scurtă în Containerii **S**tandard **T**emplate **L**ibrary (Vector/Stack/Queue)
   1. Vector
   2. Stack & Queue

**Discuții sortări**

În practică, putem clasifica mai multe tipuri de sortări: elementare, prin comparație sau prin numărare.

Cele **elementare** (**Bubble Sort**, **Selection Sort** și **Insertion Sort**) sunt cele mai lente, dar și cele mai ușor de implementat. Cele **prin comparație** (**Quick Sort**, **Merge Sort**, **Heap Sort**, **Intro Sort**) se disting prin faptul că la fiecare pas avem un pivot față de care să ne raportăm sau 2 sau mai multe elemente pe care să le verificăm și interschimbăm (dacă e necesar). Cele **prin numărare** (**Bucket Sort**, **Radix Sort**, **Counting Sort**) implică menținerea unei structuri de date auxiliare care să memoreze numărul de apariții sau să definească o categorie în care putem introduce mai multe valori cu aceeași proprietate (bucket-uri).

În funcție de tipul și numărul de date care trebuie procesate, putem face anumite alegeri optime, raportându-ne la alegerea tipului de sortare utilizat:

* numere naturale; mai puțin de 10^6 — **Counting Sort**
* șiruri de caractere / numere în diferite baze; mai puțin de 10^18 — **Radix Sort**
* numere întregi — **Bucket Sort**

**ATENȚIE!**

* Folosirea unor **structuri de date auxiliare** pentru memorarea datelor, **influențează negativ complexitatea-spațiu** a algoritmului. De exemplu, **Quick Sort** poate avea o **complexitate-timp** mai mare decât **Merge Sort**. Totuși, pentru că **Merge Sort** folosește un vector auxiliar pentru memorarea subvectorilor, acesta, este, în practică, **mai ineficient decât Quick Sort**, care nu are nevoie de memorie suplimentară.
* **Quick Sort** pierde din eficiență în funcție de **alegerea pivotului**. NU luăm primul / ultmul element din vector ca pivot. Optim este să luăm mediana din 3, 5, 7, mediana medianelor (<https://en.wikipedia.org/wiki/Median_of_medians>) sau un element random ca pivot.
* **Shell Sort** pierde din eficiență în funcție de **alegerea gap-ului**. (<https://en.wikipedia.org/wiki/Shellsort#Gap_sequences>)
* Majoritatea sortărilor optime ating un timp de **O(nlogn)**, ceea ce noi căutăm.
* Se regăsesc și **sortări hibride** (**Tim Sort** — sortarea din Python, **Intro Sort** — sortarea din STL) care presupun folosirea mai multor tipuri de sortări, în funcție de datele rămase. De exemplu: Tim Sort începe cu Merge Sort și trece în Insertion Sort când numărul de elemente scade sub o anumită limită.
* Eu, recomand în practică (NU la temele și cerințele care vă cer explicit tipuri de sortări) folosirea funcției de sortare implicită **sort()**, la care să adăugați o funcție de tip bool care să vă compare elementele după criterii alese de voi sau o lambda-expresie. De exemplu:

bool compara(long long a, long long b){return a\*a\*0.5 < b\*b\*0.5;}

……………………………………………………………………………………………..

sort(a, a+n, compara);

**Introducere scurtă în Containerele STL**

Containerele STL sunt doar clase de structuri de date dinamice (își (re)alocă spațiu singure, la nevoie) deja scrie în C++ (Container = Clasă). E mană cerească să aveți la îndemână funcționalități care v-ar lua timp prețios să le scrieți și verificați de bug-uri. Deci, prinde bine să vă familiarizați cu ele fiindcă o să vă facă viața mai ușoară la implementarea temelor de laborator la SD, cât și la cele de OOP și mai ales la interviuri.

**vector**

Am vorbit puțin despre operațiile de bază ale container-ului vector, acoperind o mare parte din articolul acesta: [C++ vector - scurt tutorial | www.pbinfo.ro](https://www.pbinfo.ro/articole/23705/cpp-vector-scurt-tutorial).

Deși este menționat și în articolul de mai sus, doresc să pun accentul pe cum este alocată memoria pentru un conainter, mai ales pentru vector.

La declararea container-ului, compilatorul alocă un spațiu de lucru pe acesta (o putere de-a lui 2 \* (mărimea tipului de date în bytes)). După multe push\_back-uri, când adăugăm în structura noastră un element care o face să dea pe afară, structura este mutată în altă zonă de memorie și i se alocă de două ori mai mult spațiu.

Din cauza acestui fapt, uneori e posibil să avem spike-uri destul de măricele în performanță deoarece dintr-un push\_back în O(1) ne trezim cu o realocare în O(n). De asemenea, acel O(n) din realocare are o constantă destul de măricică în spate, mutarea unui container implică crearea unei referințe noi de memorie, apelări de constructori și metode și consumă mult timp. Implicit, ne place să (re)alocăm de cât mai puține ori containeri STL.

Aveți grijă la declarări de containeri în recursivitate. De pildă, avem următorul subprogram care rulează Merge sort (luat din [acest](https://github.com/PlatDrake2875/5-Sorting-Algorithms-in--C-Plus-Plus) repo):

vector<long long> temp;  
  
void Merge\_Sort(vector<long long> &A, int left, int right)  
{  
 if(left == right)  
 return;  
 int mid = (left + right) / 2;  
 Merge\_Sort(A, left, mid);  
 Merge\_Sort(A, mid + 1, right);  
  
 for(int i = left, j = mid + 1, k = 0; i <= mid || j <= right;)  
 if(j > right || (i <= mid && A[i] < A[j]))  
 temp[k++] = A[i++];  
 else  
 temp[k++] = A[j++];  
 for(int k = left; k <= right; ++k)  
 A[k] = temp[k - left];  
}

void Merge\_Sort(vector<long long> &A, int left, int right)  
{  
 if(left == right)  
 return;  
 int mid = (left + right) / 2;  
 Merge\_Sort(A, left, mid);  
 Merge\_Sort(A, mid + 1, right);

vector<long long> temp(right – left + 1);

for(int i = left, j = mid + 1, k = 0; i <= mid || j <= right;)  
 if(j > right || (i <= mid && A[i] < A[j]))  
 temp[k++] = A[i++];  
 else  
 temp[k++] = A[j++];  
 for(int k = left; k <= right; ++k)  
 A[k] = temp[k - left];  
}

Vă invit să comparați cele două bucăți de cod să vă convingeți că diferența de viteză este colosală.

**Stack & Queue**

Am implementat stiva și coada în versiunea clasică și în cea cu STL.

Găsiți toate implementările în Folder-ul Tutoriat 1.

TODO: O să mai adăugăm materiale pentru implementări și animații pentru structuri de date.