ALGORITMI DE SORTARE

1. SHELL SORT

- Bazat pe Insertion Sort, dar permite interschimbarea elementelor de pe poziții mult mai depărtate, în funcție de gap-ul ales
- Ca idee, setăm **h** – dimensiunea gap-ului și divizăm vectorul în subvectori ce conțin intervale de dimensiune h; sortăm subvectorii folosind Insertion Sort; repetăm pașii până când vectorul e în întregime sortat
- Complexitatea algoritmului depinde de alegera gap-urilor https://en.wikipedia.org/wiki/Shellsort#Gap_sequences
- **Timp: Average:** $O(n \log n)$ **Worst Case:** $O(n^2)$

2. QUICK SORT

- Algoritm Divide-et-Impera, eficient în practică, dar eficiența depinde de algerea pivotului
- **Divide**: se împarte vectorul în doi subvectori în funcție de un pivot **x**, astfel încât elementele din subvectorul din stânga sunt ≤ **x** ≤ elementele din subvectorul din dreapta
- **Impera**: se sortează recursiv cei doi subvectori
- Cel mai eficient pivot: mediana din 3 (luăm 3 elemente din vector: primul element, cel din mijloc și pe ultimul și le sortăm), mediana (adică elementul din mijlocul vectorului), mediana din 5, mediana din 7, mediana medianelor, pivot random
- Cel mai ineficient pivot: primul/ultimul element al vectorului
- **Timp: Average:** $O(n \log n)$ **Worst Case:** $O(n^2)$

3. MERGE SORT

- Algoritm Divide-et-Impera
- **Divide**: se împarte vectorul în jumătate, și se sortează fiecare parte
- **Impera**: se sortează recursiv cei doi subvectori
- Recursivitatea se oprește când am vectori de lungime 1 sau 2
- Algoritmul de Merging -> creez un vector temprorar **v_aux**; iterez prin cele 2 jumătăți sortate de la stânga la dreapta; copiez în **v_aux** cel mai mic dintre cele 2 elemente
- **Timp:** $O(n \log n)$

4. COUNTING SORT

- Optim când avem de sortat date mici
- Se implementează cu un vector de frecvență **fr[max]** în care reținem numărul de apariții pentru fiecare valoare, iar pentru afișare parcurgem vectorul de frecvență și afișăm **i** de **fr[i]** ori
- **Timp:** $O(n + \max$) unde **n** = numărul de elemente, **max** = valoarea maximă citită
- **Spațiu:** $O(\max$)

5. BUCKET SORT

- Elementele vectorului sunt distribuite în bucket-uri (reprezentate ca liste înlănțuite) după anumite criterii
- Avem **v** - vectorul inițial, **b** - vectorul de bucketuri; iterăm prin **v** și adăugăm fiecare element în bucket-ul corespunzător; sortăm fiecare bucket; iterăm prin fiecare bucket și adăugăm elementele înapoi în **v**
- Inserare în bucket: cel mai uzual este să împărțim la o valoare stabilită și în funcție de câtul împărțirii să inserăm în bucket-ul corespunzător
- Sortare bucket: aplicăm recursiv Bucket Sort sau folosim o sortare simplă
- **Timp: Average:** $O(n + k)$ **Worst Case:** $O(n^2)$ unde **k** = numărul de bucket-uri
- **Spațiu:** $O(n + k)$

6. **RADIX SORT**
   - Folosit în special pentru sortarea șirurilor de caractere
   - Bucket Sort generalizat pentr numere mari; **B** - baza în care considerăm numerele (2, 2^4, 2^16, 10, 10^2, 10^4); presupunem că **v** – vectorul de sortat conține valori din mulțimea **{0, .. , B-1}** pe care le vom sorta după fiecare cifră; bucket-urile – cifrele numerelor => necesare --- baza în care sunt scrise numerele --- bucketuri
   - **Timp:** $O(n \log \max)$ unde **max** = numărul maxim de cifre dintr-un număr
   - **Spațiu:** $O(n + k)$ unde **b** = baza în care sunt scrise numerele
   - **LSD (Least Significant Digit)** --- iterativ, rapid
   - **MSD (Most Significant Digit)** --- recursiv, similar Bucket Sort

7. **TIM SORT**
   - Algoritm hibrid (Merge + Insertion); sortarea din Python
   - Începe cu Merge și trece în Insertion dacă numărul de elemente scade sub o anumită limită

8. **INTRO SORT(INTROSPECTIVE SORT)**
   - Algoritm hibrid (Quick + Heap + Insertion); sortarea din STL
   - Începe cu Quick și trece în Heap dacă nivelul recursivității crește peste $O(\log n)$, apoi trece în Insertion dacă numărul de elemente sortate scade sub o anumită limită

ALEGERI OPTIME

- Până la 10^6 numere ----> **COUNT SORT**
- Până la 10^8 numere ----> **RADIX SORT**
- Numere întregi ----> **BUCKET SORT**

❗ **OBSERVAȚIE:** Merge Sort este mai ineficient decât Quick Sort, deoarece folosește memorie auxiliară

CLASIFICARE

Elementari	Prin comparație	Prin numărare
Insertion Sort $O(n^2)$	Quick Sort $O(n \log n)$	Bucket Sort
Selection Sort $O(n^2)$	Merge Sort $O(n \log n)$	Counting Sort
Bubble Sort $O(n^2)$	Heap Sort $O(n \log n)$	Radix Sort
	Intro Sort $O(n \log n)$	