Structuri de Date

Seminar 4

Problema 1:

Enunț complet: https://leetcode.com/problems/last-stone-weight/description/

Rezumat: Avem N pietre de diverse greutăți. La fiecare pas luăm cele mai grele două pietre si le lovim una de alta: piatra mai mică se va distruge complet, iar piatra mai mare se va micșora cu o greutate egală cu greutatea pietrei mai mici. Dacă pietrele au greutăți egale, ambele sunt distruse. Repetăm până rămânem cu cel mult o piatră. Calculați greutatea pietrei rămase (0 în caz că nu a rămas niciuna).

Soluția 1:

Sortăm pietrele și le eliminăm pe cele mai grele două. Introducem în vector diferența lor dacă e cazul. Repetăm până la final (pentru fiecare piatră = de N ori).

Complexitate timp: $O(N^2 * \log N)$ - sortare (N log N) de N ori

```
int lastStoneWeight(vector<int>& stones) {
    while (stones.size() > 1) {
        // ma doare că sortez ca să găsesc maximul
        sort(stones.begin(), stones.end());

        auto last_stone = stones.back();
        stones.pop_back();

        auto second_last_stone = stones.back();
        stones.pop_back();

        int difference = abs(last_stone - second_last_stone);
        if (difference)
            stones.push_back(difference);
    }

    return stones.size() == 1 ? stones[0] : 0;
}
```

Soluţia 2:

Găsim prin parcurgere cele mai mari două pietre, le scoatem din vector şi inserăm diferenţa lor dacă e cazul. Repetăm pentru fiecare piatră.

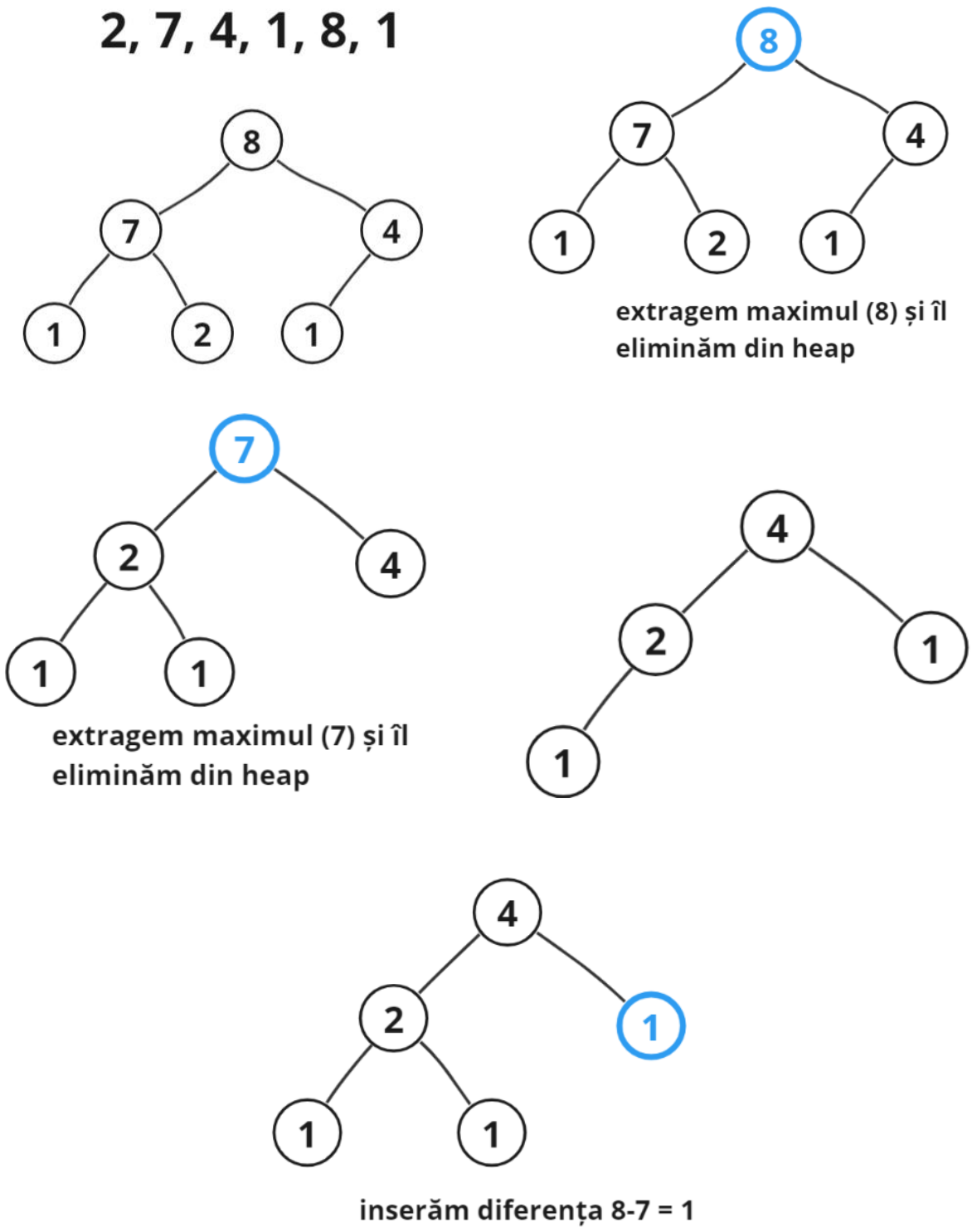
Complexitate timp: $O(N^2)$ - găsire maxime O(N), de N ori

```
int lastStoneWeight(vector<int>& stones) {
    while (stones.size() > 1) {
        int max_stone = 0;
        int second_max_stone = 0;
        int max_stone_index = -1;
        int second_max_stone_index = -1;

        for (int i = 0; i < stones.size(); ++i) {
            if (stones[i] > max_stone) {
                second_max_stone = max_stone;
                second_max_stone_index = max_stone_index;
                max_stone = stones[i];
                max_stone_index = i;
            }
            else if (stones[i] > second_max_stone) {
                second_max_stone = stones[i];
                second_max_stone_index = i;
            }
        }

        stones.erase(stones.begin() + max_stone_index);
     // daca max_stone se gasea in stanga lui second_max_stone, indexul
     // lui second_max_stone va scadea cu 1 dupa eliminarea lui max_stone
        if (max_stone_index < second_max_stone_index)
            --second_max_stone_index;
        stones.erase(stones.begin() + second_max_stone_index);

        int difference = abs(max_stone - second_max_stone);
        if (difference)
            stones.push_back(difference);
    }

    return stones.size() == 1 ? stones[0] : 0;
}
```

Soluţia 3:

Putem îmbunătăţi modul în care găsim cele două maxime. O modalitate este un heap de maxim. Extragem şi eliminăm maximul de două ori şi inserăm diferenţa lor o dată.

Complexitate timp: O(N * log N) - de N ori, eliminăm maximul din heap de două ori (2*log N) și inserăm element nou în heap (log N); de asemenea, O(N) construire heap inițial



Repetăm până rămânem cu un singur element (sau niciunul, după caz).

Implementare folosind priority_queue:

```
int lastStoneWeight(vector<int>& stones) {
   priority_queue<int> stones_heap(stones.begin(), stones.end());

   while (stones_heap.size() > 1) {
       int max_stone = stones_heap.top();
       stones_heap.pop();

       int second_max_stone = stones_heap.top();
       stones_heap.pop();

       int difference = abs(max_stone - second_max_stone);
       if (difference)
           stones_heap.push(difference);
   }

   return stones_heap.size() == 1 ? stones_heap.top() : 0;
}
```

Implementare folosind heap din STL:

```
int lastStoneWeight(vector<int>& stones) {
    make_heap(stones.begin(), stones.end());

    while (stones.size() > 1) {
        int max_stone = stones.front();
        pop_heap(stones.begin(), stones.end());
        stones.pop_back();

        int second_max_stone = stones.front();
        pop_heap(stones.begin(), stones.end());
        stones.pop_back();

        int difference = abs(max_stone - second_max_stone);
        if (difference) {
            stones.push_back(difference);
            push_heap(stones.begin(), stones.end());
        }
    }

    return stones.size() == 1 ? stones.front() : 0;
}
```

Problema 2:

Enunț complet: https://leetcode.com/problems/find-k-th-smallest-pair-distance/submissions/1199509749/

Rezumat: Avem N numere și un număr K. Considerăm toate perechile de elemente posibile folosind cele N numere și pentru fiecare pereche luăm diferența dintre cele două numere. Dacă ordonăm crescător diferențele, ne interesează să găsim a K-a diferență.

Soluția 1:

Punem toate diferențele într-un vector (de dimensiune N*(N-1)/2) și îl sortăm. Returnăm al K-lea element din vectorul sortat.

Complexitate timp: O(N^2 * log N)

Complexitate spațiu: O(N^2)

```
int smallestDistancePair(vector<int>& nums, int k) {
    vector<int> differences;

    for (int i = 0; i < nums.size(); ++i)
        for (int j = i + 1; j < nums.size(); ++j)
            differences.push_back(abs(nums[i] - nums[j]));

    sort(differences.begin(), differences.end());

    return differences[k - 1];
}
```

Soluția 2:

Putem crea din nou același vector de diferențe în O(N^2), dar să găsim al K-lea element altfel decât sortând. Spre exemplu, folosind metoda Quickselect putem obține o complexitate totală de O (N^2) - mai bine decât sortarea de la prima soluție. Vom prezenta astfel de metode mai jos.

Problema K-th element: Avem M numere nesortate și vrem să găsim al K-lea cel mai mic element.

- **Heapify – O(M + K * log M):** Punem numerele într-un heap de minim – **O(M)** – și facem pop (ștergem minimul) de K ori – **O(K * log M)**
- **Quickselect – O(M):** Foarte similar cu *Quicksort*. Este evidențiat în exemplul de mai jos, unde folosim ca pivot mijlocul (problema alegerii pivotului este aceeași ca la Quicksort).

3 6 **2** 9 1 5 **K = 5**

pivot

Toate elementele mai mici decât 2 le mutăm în stânga lui 2, iar pe cele mai mari decât 2 în dreapta lui 2.

1 **2** 3 6 9 5 **K = 5**

Am găsit astfel ce poziție ar ocupa 2 în vectorul sortat. Această poziție este 2 (poz[2]=2). K=5>2, așadar vom repeta procedeul în dreapta lui 2.

1 2 3 **6** 9 5 **K = 5**

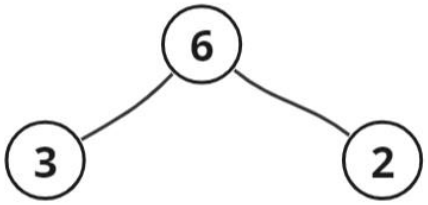
pivot

1 2 3 5 **6** 9 **K = 5**

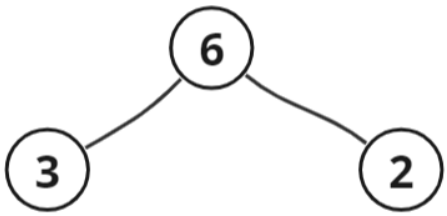
Am găsit ce poziție ar ocupa 6 în vectorul sortat. Această poziție este 5 (poz[6]=5). K=5, așadar am găsit elementul căutat: al 5-lea cel mai mic element în vector este 6.

- **Heap cu K elemente – O(M * log K):** Avem un heap de maxim în care adăugăm inițial primele K elemente. Parcurgem vectorul de la poziția (K+1) încolo și dacă elementul curent este mai mic decât maximul din heap (vârful), eliminăm maximul (este evident că acesta nu mai poate fi în cele mai mici K elemente, deoarece are deja (K-1) elemente mai mici decât el și tocmai am mai găsit unul) și inserăm elementul curent. La final, în vârful heap-ului se va găsi al K-lea cel mai mic element, deoarece un heap de maxim cu K elemente înseamnă practic că există (K-1) numere mai mici decât numărul din vârful heap-ului.

3 6 2 9 1 5 **K = 3**

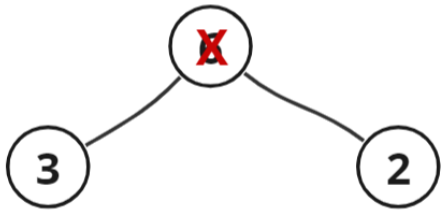


3 6 2 **9** 1 5 **K = 3**



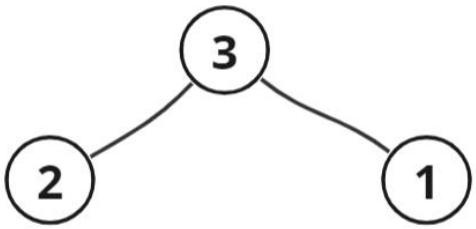
9 > 6 => există cel puțin K elemente mai mici decât 9, așadar nu prezintă interes

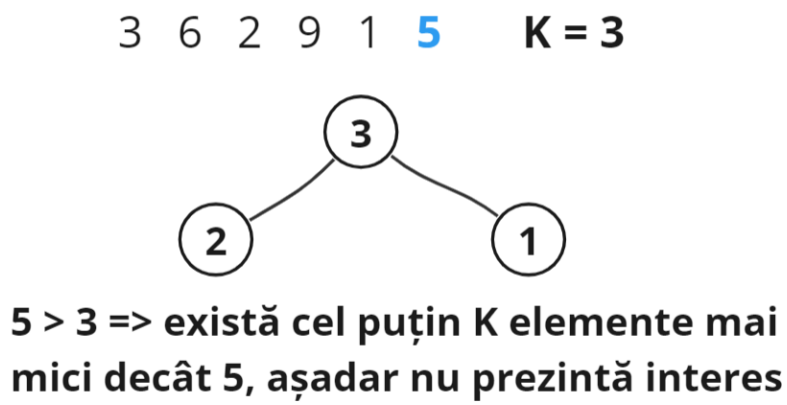
3 6 2 9 **1** 5 **K = 3**

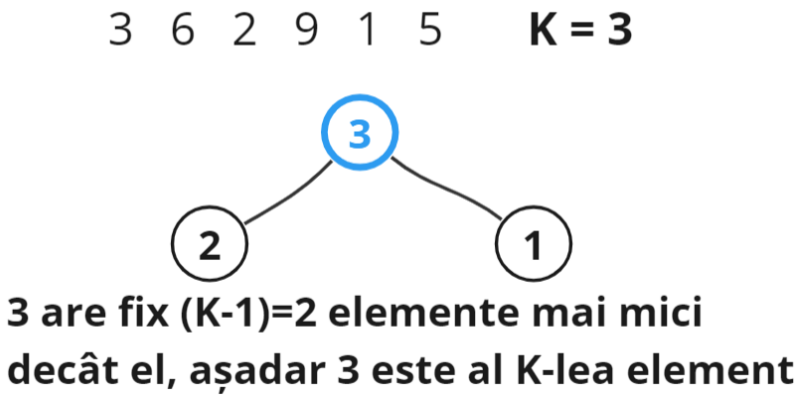


1 < 6 => eliminăm maximul (6) și inserăm elementu curent (1)

3 6 2 9 **1** 5 **K = 3**







Soluţia 3:

Sortăm vectorul inițial și folosim căutare binară pe intervalul [0,X], unde X este valoarea maximă pe care o pot lua numerele. Pentru o valoare fixată Y, căutăm pentru fiecare valoare din vector cu câte numere mai mici decât el are suma mai mică decât Y. Acest lucru se face cu un sistem de doi pointeri în timp liniar.

Complexitate timp: O(N * (log N + log Max)) - unde Max este valoarea maximă a numerelor; sortare – O(N log N), sistem 2 pointeri – O(N log Max)

```cpp
// verifică dacă există cel puțin K perechi de numere care au
// diferența mai mică sau egală cu parametrul difference
bool check_k_smaller_pairs(vector<int>& nums, int difference, int k) {

    // pairs_found = câte perechi cu diferența <= K am găsit
    int j = 0, pairs_found = 0;

    for (int i = 0; i < nums.size() && pairs_found < k; ++i) {
        // creștem j până ajungem la numere cu diferența <= K
        // reminder: vectorul nums este sortat
        while (j < i && nums[i] - nums[j] > difference)
            ++j;

        // dacă avem j .. i și nums[i]-nums[j]<=difference
        // atunci nums[i]-nums[l]<=difference pt orice l din [j+1 ; i-1]
        pairs_found += (i - j);
    }

    return pairs_found >= k;
}

int smallestDistancePair(vector<int>& nums, int k) {
    sort(nums.begin(), nums.end());

    // folosim căutare binară pentru a găsi diferența
    // care are (K-1) diferențe mai mici decât ea
    int left = 0, right = 1e6;
    while (left <= right) {
        int mid = (left + right) / 2;

        if (check_k_smaller_pairs(nums, mid, k))
            right = mid - 1;
        else
            left = mid + 1;
    }
    return left;
}
```