Structuri de Date

Seminar 3

Problema 1:

Enunț complet: https://leetcode.com/problems/find-the-difference/description/

Rezumat: Avem două șiruri de caractere, S și T. T este obținut prin amestecarea aleatorie a caracterelor lui S, și adăugarea unei litere în plus la o poziție oarecare. Găsiți care a fost litera adăugată.

Exemplu: S = “pisica”, T = “cipixsa” => litera în plus = ‘x’

Soluția 1:

Folosim doi vectori de frecvență, unul pentru fiecare șir, și vedem unde diferă.

Complexitate timp: O(N+Σ), N = numărul de caractere dintr-un șir, Σ = valoarea maximă din alfabet (în cazul limbii engleze, Σ = 26)

Complexitate spațiu: O(Σ)

S = “pisica” T = “cipixsa”

literă	a	c	i	p	s	x
apariții_S	1	1	2	1	1	0
apariții_T	1	1	2	1	1	1

răspuns: 'x'

Pentru simplitate, am pus doar literele relevante în tabel. Celelalte au toate valoarea 0 în ambele șiruri.

Soluția 2:

Folosim un singur vector de frecvență, la care adunăm fiecare literă care apare în S și scădem fiecare literă care apare în T. La final, vedem ce literă are valoarea −1.

Complexitate timp: O(N+Σ)

Complexitate spaţiu: O(Σ)

<u>Dacă în loc de vector folosim un hash, vom obține:</u>

Complexitate timp: O(N) (fără Σ)

Complexitate spaţiu: O(N)

S = "pisica" **T = "cipixsa"**

literă	a	c	i	p	s	x
apariții	1	1	2	1	1	0

Parcurgem S și contorizăm aparițiile

S = "pisica" **T = "cipixsa"**

literă	a	c	i	p	s	x
apariții	0	0	0	0	0	-1

Parcurgem T și scădem aparițiile

Alternativ, în loc să scădem pentru T, putem aduna atât pentru S, cât şi pentru T, şi să vedem ce literă are număr impar de apariţii.

Soluţia 3:

Putem folosi operaţia pe biţi XOR (^). Facem XOR între toate literele din S şi T, şi rezultatul va fi litera căutată.

Complexitate timp: O(n)

Complexitate spaţiu: O(1)

Proprietate: x ^ x = 0 ∀ x (observăm în tabel că atunci când doi biţi sunt la fel, XOR între ei va fi 0, aşadar un număr XOR-at cu el însuşi va fi 0); De asemenea, un număr XOR-at cu 0 va rămâne neschimbat.

b1	b2	b1 XOR b2
0	0	0
0	1	1
1	0	1
1	1	0

S = "pisica" T = "cipixsa"

p ^ i ^ s ^ i ^ c ^ a ^ c ^ i ^ p ^ i ^ x ^ s ^ a

=

p ^ p ^ i ^ i ^ s ^ s ^ i ^ i ^ c ^ c ^ a ^ a ^ x

=

(p ^ p) ^ (i ^ i) ^ (s ^ s) ^ (i ^ i) ^ (c ^ c) ^ (a ^ a) ^ x

=

0 ^ 0 ^ 0 ^ 0 ^ 0 ^ 0 ^ x

=

x

Soluţia 4:

Putem aduna codurile ASCII ale literelor lui S, şi să scădem codurile ASCII ale literelor lui T. Rezultatul va fi fix litera căutată.

Complexitate timp: O(n)

Complexitate spaţiu: O(1)

S = "pisica" T = "cipixsa"

'p' + 'i' + 's' + 'i' + 'c' + 'a' - 'c' - 'i' - 'p' - 'i' - 'x' - 's' - 'a' = 'x'

Problema 1.5:

Putem extinde problema 1 să folosească cuvinte în loc de caractere.

Exemplu: S = "Pisica", "prinde", "soareci" T = "prinde", "soareci", "multi", "pisica" => "multi"

Soluţiile ce folosesc vectori de frecvenţă nu mai funcţionează. În schimb, funcţionează în continuare soluţia cu hash-uri.

Complexitate timp: O(N*L) - unde e L e lungimea alfabetului

(se pot obține și **O(input)** si **O(S)** unde S e lungimea șirurilor de intrare)

Problema 2:

Enunț complet: https://leetcode.com/problems/4sum-ii/

Rezumat: Se dau 4 vectori de lungime N ce conțin numere întregi (inclusiv negative). Însumând câte un element din fiecare vector, vrem să obținem suma 0. Găsiți toate combinațiile posibile de astfel de numere și returnați numărul lor.

Mai exact, căutăm toate (i, j, k, l) cu proprietatea că nums1[i] + nums2[j] + nums3[k] + nums4[l] = 0.

Exemplu: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
1. (0, 0, 0, 1) → nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1+(-2)+(-1)+2 = 0
2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2+(-1)+(-1)+0 = 0

=> răspuns = 2

Soluția 1:

Folosim patru for-uri:

```
int fourSumCount(vector<int>& nums1,
                 vector<int>& nums2,
                 vector<int>& nums3,
                 vector<int>& nums4) {
    int count = 0;

    for (auto num1 : nums1)
        for (auto num2 : nums2)
            for (auto num3 : nums3)
                for (auto num4 : nums4)
                    if (num1 + num2 + num3 + num4 == 0)
                        ++count;

    return count;
}
```

Complexitate timp: $O(N^4)$

Soluția 2:

Sortăm vectorul nums4. Cu ceilalţi trei vectori facem trei for-uri şi obţinem numărul **x = nums1[i] + nums2[j] + nums3[k]**. Căutăm numărul **(0-x)** în nums4 folosind căutare binară.

Complexitate timp: O(N³ * log N)

```
int fourSumCount(vector<int>& nums1,
                 vector<int>& nums2,
                 vector<int>& nums3,
                 vector<int>& nums4) {
    int count = 0;

    sort(nums4.begin(), nums4.end());

    for (auto num1 : nums1)
        for (auto num2 : nums2)
            for (auto num3 : nums3) {
                int x = num1 + num2 + num3;
                if (binary_search(nums4, 0 - x) != -1)
                   ++count;


    return count;
}
```

Soluţia 3:

Punem elementele din nums4 într-un hash şi facem din nou trei for-uri ca mai sus, căutând în hash numărul **(0-x)**, unde **x = nums1[i] + nums2[j] + nums3[k].**

Complexitate timp: O(N³)

Soluţia 4:

Luăm toate combinaţiile posibile de elemente din nums1 şi elemente din nums2 şi punem într-un hash sumele **nums1[i]+nums2[j]**. Parcurgem cu două for-uri nums3 şi nums4 şi căutăm în hash numărul **0-(nums3[k]+nums4[l])**.

Complexitate timp: O(N²)

```cpp
int fourSumCount(vector<int>& nums1,
                 vector<int>& nums2,
                 vector<int>& nums3,
                 vector<int>& nums4) {
    int count = 0;
    unordered_map<int, int> hash;

    for (auto num1 : nums1)
        for (auto num2 : nums2)
            ++hash[num1 + num2];

    for (auto num3 : nums3)
        for (auto num4 : nums4)
            count += hash[0 - (num3 + num4)];

    return count;
}
```

Problema 3:

Enunț complet: https://leetcode.com/problems/substring-with-concatenation-of-all-words/description/

Rezumat: Avem un șir S și niște cuvinte de lungimi egale între ele. Vrem să găsim pozițiile din S unde apare concatenarea tuturor cuvintelor date, în orice ordine.

Soluția 1:

Fie L lungimea fiecărui cuvânt din listă (au lungimi egale). Pentru fiecare indice i, parcurgem de la i din L în L și verificăm dacă fiecare subșir **[i..i+L]**, **[i+L+1..i+2*L]**, ... este un cuvânt din listă, care nu a mai fost găsit (sau nu i-au fost găsite toate aparițiile).

Complexitate timp: O(N*M*L), unde N este lungimea lui S, M este numărul de cuvinte, L este lungimea unui cuvânt

```cpp
vector<int> findSubstring(const string& s, vector<string>& words) {
    vector<int> result;

    int s_length = s.length();
    int num_words = words.size();
    int word_length = words[0].length();
    int total_length = num_words * word_length;

    // reținem de câte ori apare fiecare cuvânt în lista de cuvinte
    unordered_map<string, int> word_count;
    for (const string& word : words)
        ++word_count[word];

    for (int i = 0; i <= s_length - total_length; ++i) {
        // needed[x] = câte apariții ale lui x mai trebuie să găsim
        unordered_map<string, int> needed = word_count;

        // verificam cuvintele de la indicii
        // i, i + word_len, i + 2*word_len ... i + num_words*word_len
        int j;
        for (j = 0; j < num_words; ++j) {
            // al j-lea cuvânt de lungime word_length, de la cu indexul i
            string word = s.substr(i + j * word_length, word_length);

            // cuvântul nu se află în listă, sau a fost găsit deja
            if (!needed[word]) break;

            // marcăm că am găsit o apariție a cuvântului
            --needed[word];
        }

        // s-au găsit toate cuvintele din listă
        if (j == num_words)
            result.push_back(i);
    }

    return result;
}
```

Soluția 2:

Va fi util să ne gândim la algoritmul Rabin-Karp.

Hash("bar") = 'b' * P^2 + 'a' * P + 'r' * P^0 , P nr. prim.

Parcurgem șirul S și identificăm cuvintele din words (acest lucru este posibil doar pentru că toate au aceeași lungim) folosind hash-ul calculat. Folosim tehnica **sliding window** ca să ne deplasăm din L în L (unde L este lungimea unui cuvânt din listă). Cu această soluție putem ajunge la o **complexitate timp O(N+M*L)**, unde N este lungimea lui S, M este numărul de cuvinte, L este lungimea unui cuvânt.