AISD lista 4

Dominik Szczepaniak

August 31, 2024

1 Zadanie 1

Jednocześnie przy liczeniu DP będziemy jeszcze trzymać drugi co do wielkości element oprócz pierwszego.

Drugi oczywiście liczymy tak, że bierzemy drugi najmniejszy z poprzednich opcji i dodajemy do niego wartośc z aktualnego elementu.

Na końcu znajdziemy minimum z pierwszego i drugiego elementu, a później znowu przejdziemy przez dp z pierwszych elementów i dp z drugich elementów i weźmiemy minimum które nie jest minimum globalnym (wcześniej znalezionym).

Jeśli istnieją dwa najmniejsze wyniki to muszą mieć jakieś miejsce w tablicy w którym się kończą, więc jeśli będziemy pamiętać dwa wyniki, to niezależnie gdzie sie skończą dostaniemy oba.

2 Zadanie 2

skip

3 Zadanie 3

Podciąg (substring) to ciąg znaków które muszą występować w oryginalnej kolejności, czyli nie może być między nimi przerwy.

Podsekwencja (subsequence) to ciąg znaków które mogą mieć spacje między sobą.

Znaleźć taki najdłuższy wyraz który zawiera w sobie ciągi X i Y.

Na pewno X + Y będzie dobrym wynikiem.

Kiedy można to zmniejszać? Jeśli jakaś początkowa część X zawiera się w Y albo na odwrót.

Czyli jeśli X = "abcdggg" i Y = "ggdfgh" to możemy pozbyć sie dwóch g z Y i mieć :

"abcdgggdfgh"

W takim razie musimy sobie znaleźć jaka część Y zawiera się w X albo jaka część X zawiera się w Y. Możemy to zrobić binarnie.

Jeśli to jest podciąg to u gory

Jeśli podsekwencja to musimy znaleźć najdłuższy wspólny podciąg (longest common subsequence). W $O(n^2)$. później po prostu możemy dodać pozostałe litery z X i Y.

Jak znaleźć LIS?

```
    int L[m + 1][n + 1];

for (int i = 0; i <= m; i++) {
    for (int j = 0; j <= n; j++) {
        if (i == 0 || j == 0)
            L[i][j] = 0;

        else if (X[i - 1] == Y[j - 1])
            L[i][j] = L[i - 1][j - 1] + 1;

        else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
    }
}

return L[m][n];
```

4 Zadanie 4

Niech dp[i][j] oznacza pesymistyczny koszt szukania na przedziale [i, j]

Jesli wielkosc przedzialu jest rowna 1 to koszt to c[i]

Jesli wielkosc przedzialu jest rowna 2 to koszt to c[i] + c[j]

Jesli przedzial jest wiekszy niz 2, to
dp[i, j] = min i <= k <= j (max(dp[i, k], dp[k, j]) + c[k])
Jeśli będziemy to liczyć ręcznie to obliczymy całość w $O(n^3)$.

Zauważmy, że nie chcemy mieć sytuacji gdzie jeden z dp jest o wiele większy od drugiego bo bierzemy maxa. W takim razie chcemy zrównoważyć, aby dp[i, k] było mniej więcej równe dp[k, j] - chcemy wziąć takie k które minimalizuje bezwzględną różnice.

Możemy to robić tak, że bierzemy strzelamy w sobie jakieś miejsce:
L y R, gdzie L i R to przedział.
Teraz mamy możliwe sytuacje:
L = R - trafiliśmy idealnie
L > R - musimy przesunąć k w lewo, bo wtedy R sie zwiększy, a L zmniejszy
L < R - musimy przesunąć k w prawo bo wtedy R się zmniejszy a L zwiększy.

Zauważmy, że możemy to miejsce szukać binarnie. Musimy tylko szybko odpowiadać na pytania o sume na przedziale - no ale z nazwy robimy to sumami przedziałowymi.

W takim razie jak jesteśmy na miejscu y to:

```
def krok(y):
    start = i
    end = j
    while(start < end):
        y = (i + j) / 2
        L = suma_przedzial[y] - suma_przedzial[i-1]
        R = suma_przedzial[j] - suma_przedzial[y-1]
        if(L < R):
            start = y
        else if(L > R):
            end = y
        else:
            start = y
            end = y
```

chuj wie co dalej nie pamietam tegogowna lol
jakies drzewo przedzialowe sie tu wpierdalalo ale co dokladnie to idk

5 Zadanie 5

Hirschberg's algorithm

https://www.cs.nthu.edu.tw/ wkhon/algo09/tutorials/tutorial-hirschberg.pdf

Jesli liczymy LCS w klasyczny sposob, to mozemy zastosowac ten sam sposob na liczenie od prawej strony, tylko wtedy bierzemy z prawego dolnego rogu lub maxa z prawej lub dolu.

Teraz mozemy robic tak, ze robimy divide and conquer na jednym z napisow (niech bedzie tym co jest pionowo w tablicy). Liczymy najpierw od gory do dolu dla gornej czesci a nastepne od dolu do gory dla dolnej czesci. Nastepnie w polowie szukamy najwiekszej sumy po przekatnej. Wtedy robimy tak, ze liczymy rekurencyjnie dla obu podtablic ktore nam wyjda.

Czyli obrazujac na przykladzie:

Niech nasza cala tablica wyglada mniej wiecej tak:

011112

111222

211110

111110

Wtedy max jest na przekatnej 2 i 1 (suma 3)

Odcinamy w nastepujacy sposob:

01111xx

11122xx

xxxxx10

xxxxx10

I liczymy odpowiednio rekurencje dla powstalych tablic znowu je dzielac w polowie.

Bazowy krok to gdy wysokosc jest rowna 1. Wtedy szukamy dowolnej litery ktora sie zgadza lub jesli takiej nie ma to zwracamy pusty string. Nastepnie laczymy wyniki.

Czemu jest O(n*m)?

Najpierw liczymy calosc w O(nm)

Pózniej liczymy 1/2 * O(nm)

Pozniej 1/4*O(nm)

Mamy wiec $O(nm) * \sum_{i=0}^{\inf}(1/2)^i = O(nm) * 2 = O(nm)$

Czemu miejsca jest liniowo?

Poniewaz rekurencje mozemy liczyc tak, ze najpierw pamietamy dla calego wyniku 4 linijki (dla gornej polowy i dolnej). Nastepnie liczymy rekurencyjnie

gorna czesc rekurencji, ktora tez potrzebuje 4 linijki, ale nie musimy pamietac wyniku dla dolnej czesci, wiec wykorzystujemy te 4 linijki. Gdy skonczymy liczyc gorna czesc rekurencji mamy string ktory zwrocila gorna czesc i liczymy dolna czesc, to tez zwroci string i nastepnie je dwa laczymy. Czyli zawsze pamietamy 4 linijki, czyli 4n = O(n), no i dodatkowo jakies dane o stringach oraz miejscach gdzie zaczyna sie rekurencja, ale one sa ograniczone i tak przez O(n), poniewaz mozemy miec co najwyzej n roznych rekurencji, bo jesli jest wysokosc 1 to konczymy.

6 Zadanie 6

Podzbiór jest niezależny jeśli nie ma żadnych dwóch takich wierzchołków które są połączone krawędzią.

dla każdego wierzchołka chcemy mieć maksymalny wynik dla jego poddrzewa jeśli

a) wybralibyśmy ten wierzchołek

w takim przypadku bierzemy sume z dzieci dla przypadku jeśli byśmy ich nie wybrali.

b) nie wybralibyśmy go

w takim przypadku bierzemy sume z dzieci dla lepszego z dwóch przypadków.

```
def licz(i, p):
    if dzieci[i].size() == 0:
        dp[i][0] = w[i]
        dp[i][1] = 0
    else:
        for j in dzieci:
            licz(j, i)
        dp[i][0] = w[i] + sum(dp[j][1] for j in dzieci[i])
        dp[i][1] = max(dp[j][0], dp[j][1] for j in dzieci[i])
```

wynik będzie w max(dp[korzen][0], dp[korzen][1])

jeśli mamy znaleźć dokładnie jakie wierzchołki wybraliśmy to dodatkowo do dp trzymamy sobie binary string który nam mówi które wierzchołki wybraliśmy, jeśli wybraliśmy to 1, jeśli nie to 0. z każdym kolejnym wyborem aktualizujemy string który wzięliśmy od synów.

Dlaczego działa? Ponieważ rozpatrujemy dp w którym mamy jakieś

drzewo i rozszerzamy je o jakiś nowy wierzchołek. W takim przypadku albo weźmiemy ten wierzchołek albo go nie weźmiemy. W obu przypadkach wiemy jaki będzie najlepszy wynik.

Złożoność:

O(n) - dla każdego wierzchołka wejdziemy do niego tylko raz.

7 Zadanie 7

a) Trzymamy 6 kubełków - najdłuższe podsłowa z LCS 'a', 'aa', 'aaa', 'aaab', 'aaabb', reszta

Jak trafiamy na nową literę to przechodzimy po wszystkich kubełkach i jeśli litera jest a to przesuwamy wszystkie rzeczy z kubełka 'a' do 'aa', z 'aa' do 'aaa' i z pustych do 'a'.

Jak trafiamy na b to z 'aaa' do 'aaab' i z 'aaab' do 'aaabb'.

Gdy trafiamy na jakakolwiek inna literke to w kazdym kubelku dopisujemy 1.

8 Zadanie 8

Jesli znamy sume dwoch podzbiorow to trzeciego tez.

W takim razie chcemy sprawdzic wszystkie mozliwosci wyboru dwoch zbiorow.

Nastepnie chcemy uzyc memoizacji aby nie liczyc dwa razy tego samego.

Kod:

```
bool CanBuild3EqualSums(int sumA, int sumB, int[] numbers, int totalSum
{
    if (memo == null)
        memo = new Dictionary<(int, int, int), bool>();

    int sumC = totalSum - sumA - sumB;

    var key = (sumA, sumB, numbers.Length);
    if (memo.ContainsKey(key))
        return memo[key];

    if (numbers.Length == 0)
```

```
        return sumA == sumB && sumB == sumC;

    bool result = CanBuild3EqualSums(sumA + numbers[0], sumB, numbers[1
                || CanBuild3EqualSums(sumA, sumB + numbers[0], numbers[1
                || CanBuild3EqualSums(sumA, sumB, numbers[1..], totalSum

    memo[key] = result;
    return result;
}
```

Jaka jest zlozonosc? sumA oraz sumB moga przyjmowac O(nC) mozliwych wartosci.

Sprawdzamy to dla kazdej liczby (n+1)

Czyli mamy:

$O((nC)^2 * (n+1)) = O(n^3 * C^2)$

9 Zadanie 9

Mamy LCS gdzie

f[i][j] = f[i-1][j-1] + 1 jeśli a[i] == b[j]

max(f[i][j-1], f[i-1][j]) jeśli a[i] != b[j]

Oraz mamy LIS gdzie:

f[i] = max(f[j] + 1, 1) j < i, a[j] < a[i]

W LCIS chcemy miec f[i][j] jako LCIS dla pierwszych i elementów z a i końcu w elemencie b[j].

Jeśli a[i] != b[j] to nie możemy wybrać a[i], bo ma się kończyć w b[j], więc b[j] musi być wybrane, czyli

f[i][j] = f[i-1][j]

Jeśli a[i] == b[j] to sprawdzamy wszystkie podciągi które kończą się z b[k] (k < j) i sprawdzamy czy możemy dodać a[i] (czyli b[j])

f[i][j] = max(f[i-1][k] + 1) dla każdego k < j, b[k] < a[i]

f[i][j] = f[i-1][j] jeśli a[i] != b[j]

max(f[i-1][k] + 1, k<j, b[k] < a[i]) jeśli a[i] == b[j]

Ponieważ możemy pamiętać największy f[i-1][k] to możemy to zrobić w złożoności $O(n^2)$.

```
for (int i = 1; i <= n; ++i)
{
    int fmax = 0, pos = 0;
    for (int j = 1; j <= m; j++)
    {
        f[i][j] = f[i - 1][j];      // Exclude a[i] from LCIS
        pre[i][j] = pre[i-1][j];
        if (a[i] == b[j])
        {   // Add a[i] to LCIS
        // fmax is the maximum value of the O(n^3) solution f[i-1][
            if (f[i][j] < fmax + 1)
            {
                f[i][j] = fmax + 1;
                // Prepare for the output path, record the index of
                // previous element of the LCIS ending with b[j] am
                // first j elements of b, which is also an element
                // first i elements of a.
                pre[i][j] = pos;
            }
        }
        if (b[j] < a[i])
        {
            if (f[i - 1][j] > fmax)
            {
            // fmax is the maximum value of the O(n^3) solution f[i
                fmax = f[i - 1][j];
                pos = j;
            }
        }
    }
}
```

ograniczenie k nic nie zmienia

10 Zadanie 10

Każda prosta zawiera dwa punkty które symbolizują przecięcie z l' i l".

$p1 -> p(x_{p_1}, y_{p_1}) oraz k(x_{k_1}, y_{k_1})$

$p2 -> p(x_{p_2}, y_{p_2}) oraz k(x_{k_2}, y_{k_2})$

$\vdots$

Proste p_1i p_2 przecinają się gdy:

$x_{p_1} < x_{p_1} oraz x_{k_1} > x_{k_2}$

lub

$x_{p_1} > x_{p_2} oraz x_{k_1} < x_{k_2}$

W takim razie wartości y nas w ogóle nie interesują.

Wartości dokładne x też są bez znaczenia, liczy się tylko ich porządek.

Zapiszmy teraz wszystkie pary przecięć jako pary (x_{p_1}, x_{k_1})

Ponieważ posortujemy wszystkie te pary względem pierwszego przecięcia zawsze będzie zachodzić, że ta po lewej jest mniejsza od tej po prawej.

Teraz jedyne co wystarczy zrobić to policzyć najdłuższy rosnący podciąg.

Robimy taki dp, że dp[i] oznacza najmniejszy element którym kończy się podciąg długości i.

Na początku dp[0] = -inf a reszta +inf

Jak idziemy po a[i] i chcemy wstawić a[i] na miejsce d[l] to d[l-1] $<$ a[i] oraz a[i] $<$ d[l]

Możemy to zrobić dwoma pętlami w $O(n^2)$

Dwie obserwacje:

1) Tablica d jest posortowana.

2) a[i] może zupdatować co najwyżej 1 wartość w d (bo może być tylko jedno miejsce w tablicy d, gdzie d[l-1] $<$ a[i] $<$ d[l])

Więc możemy znaleźć sobie to miejsce za pomocą binsearcha w O(nlogn).

Odwarzamy trzymając drugą tablicę która mówi skąd przyszliśmy (indeks)

b) Jeśli chcemy zapytać się ile jest LIS o podanej długości to wystarczy

$O(n^2)$:

```
int countLIS(vector<int>& arr) {
    int n = arr.size();

    // Vector to store the length of the
    // LIS ending at each element
```

```
vector<int> lis(n, 1);

// Vector to store the number of LIS
// of that length ending at each element
vector<int> count(n, 1);

// Variable to track the length of
// the longest LIS found
int maxLen = 1;

for (int i = 1; i < n; i++) {
    for (int prev = 0; prev < i; prev++) {
        if (arr[i] > arr[prev]) {

            // If a longer subsequence is found,
            // update the lis and reset the count
            if (lis[i] < lis[prev] + 1) {
                lis[i] = lis[prev] + 1;
                count[i] = count[prev];
            }

            // If another subsequence of the
            // same length is found, add to the count
            else if (lis[i] == lis[prev] + 1) {
                count[i] += count[prev];
            }
        }
    }
    maxLen = max(maxLen, lis[i]);
}

// Sum up counts of subsequences that
// have the maximum length
int res = 0;
for (int i = 0; i < n; i++) {
    if (lis[i] == maxLen) {
        res += count[i];
    }
```

```
    }

    return res;
}
```

W dp[i] trzymamy wszystkie takie wartości które były kiedyś liczbą którą kończył się LIS o długości i+1. Te wartości są trzymane jako pierwszy element pary. Drugi element pary reprezentuje liczbę możliwych opcji dla których LIS o długości i+1 kończy się elementem większym lub równym niż pierwszy element pary. Na tych wartościach robimy sumy prefiksowe.

Jeśli chcemy wiedzieć, ile mamy możliwości zakończenia najdłuższego rosnącego podciągu (LIS) długości m wartością y, wystarczy wykonać wyszukiwanie binarne dla indeksu i w tablicy dyn[m-1], gdzie pierwsza część pary jest ściśle mniejsza niż y. Wtedy liczba opcji to:

dyn[m-1].back().second - dyn[m-1][i-1].second: Różnica między liczbą LISów długości m-1 kończących się dowolną wartością a liczbą kończącą się wartością ściśle mniejszą niż y (przechowywaną w indeksie i-1).

dyn[m-1].back(): Przypadek, gdy i jest równe 0, co oznacza, że w dyn[m-2] nie ma elementów ściśle mniejszych niż y. W takim przypadku wszystkie możliwe wartości można dołączyć do LIS, więc używamy po prostu całkowitej liczby LISów długości m-1 (przechowywanej w dyn[m-1].back()).

Weźmy przykład:
[1, 4, 2, 5, 3]

i = 0
dp[1] = 1, 1
i = 1
binsearchem wyszukujemy miejsce 2
dp[2] = 4, 1
i = 2
binsearchem wyszukujemy miejsce 2
dp[2] = 4, 1, 2, 1
i = 3
binsearchem wyszukujemy miejsce 3

binsearchem wyszukujemy miejsce zerowe (czyli większe od każdego elementu w historii) w dp[2]
więc bierzemy sumę prefiksową z całej tablicy
dp[3] = 5, 2
i = 4
binsearchem wyszukujemy miejsce 3
dp[3] = 5, 2, 3, 1

to jaką optymilizacje robimy to to, że zamiast trzymać tylko nasze wystąpienia w drugim elemencie trzymamy wystąpienia nasze i wszystkich mniejszych.

Czyli w dp[2] mamy 4, 2, 2, 1 i wtedy jak bierzemy 5 to będziemy mieć po prostu 5, 2

Później jak weźmiemy 3, to mamy, że 3 jest większa od 2 i mniejsza od 4, więc weźmiemy wartość z 2 i dodamy wynik z 5:
dp[3] = 5, 2, 3, 2+1=3 = 5, 2, 3, 3

Wynik to dp[3].back().second()

```
int findNumberOfLIS(vector<int>& nums) {
    if (nums.empty())
        return 0;

    vector<vector<pair<int, int>>> dp(nums.size() + 1);
    int max_so_far = 0;
    for (int i = 0; i < nums.size(); ++i) {
        // bsearch insertion point
        int l = 0, r = max_so_far;
        while (l < r) {
            int mid = l + (r - l) / 2;
            if (dp[mid].back().first < nums[i]) {
                l = mid + 1;
            } else {
                r = mid;
            }
        }
```

```
            // bsearch number of options
            int options = 1;
            int row = l - 1;
            if (row >= 0) {
                int l1 = 0, r1 = dp[row].size();
                while (l1 < r1) {
                    int mid = l1 + (r1 - l1) / 2;
                    if (dp[row][mid].first < nums[i]) {
                        r1 = mid;
                    } else {
                        l1 = mid + 1;
                    }
                }

                options = dp[row].back().second;
                options -= (l1 == 0) ? 0 : dp[row][l1 - 1].second;
            }


            dp[l].push_back({nums[i], (dp[l].empty() ? options : dp[l].l
            if (l == max_so_far) {
                max_so_far++;
            }
        }

        return dp[max_so_far-1].back().second;
    }
```