Structuri de Date

Seminar 7

Problema 1:

Se dau o listă de N cuvinte și o propoziție. Se cere ca toate cuvintele din propoziție care încep cu un cuvânt din listă (sunt formate prin adăugarea unui sufix la un cuvânt din lista dată) să fie înlocuite cu cuvântul respectiv.

Exemplu: L = [“pisi”, “manca”, “dat”], P = “I-am dat mancare pisicii”

=> “I-am dat manca pisi”

Soluţia 1:

Iau fiecare cuvânt din propoziție și îl verific cu fiecare cuvânt din listă. Pentru a verifica un cuvânt din propoziție cu toate cuvintele din listă, complexitatea timp este **O(cuv_lista1 + cuv_lista2 + ... + cuv_listaN) = O(I)**, unde **I** este suma lungimilor cuvintelor din listă. Pentru a face acest lucru cu toate cuvintele din propoziție obținem complexitatea timp **O(cuv_prop1 * I + cuv_prop2 * I + ... + cuv_propM * I) = O(I*L)**, unde L este lungimea propoziției (suma lungimilor cuvintelor din propoziție).

Soluţia 2:

Putem sorta cuvintele din listă și, pentru fiecare cuvânt din propoziție, să facem căutare binară în listă.

Soluţia 3:

Punem toate cuvintele din listă într-un trie, apoi căutăm fiecare cuvânt din propoziție în trie. Complexitatea timp va fi O(L+I) unde L este lungimea propoziției (suma lungimilor cuvintelor din propoziție), **I** este suma lungimilor cuvintelor din listă.

Problema 2:

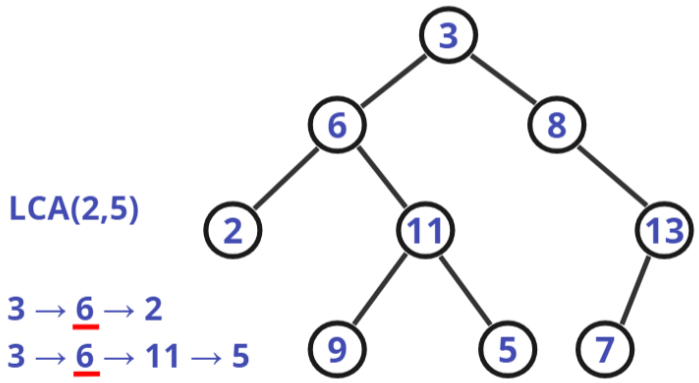
Se dă un arbore binar (simplu, NU de căutare) sub următoarea formă:

```
struct TreeNode {
     int val;
     TreeNode *left;
     TreeNode *right;
     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 };
```

Se cere să se găsească cel mai de jos strămoș comun (Lowest Common Ancestor) pentru două noduri date.

Soluţia:

Vom explica soluția pe un exemplu.



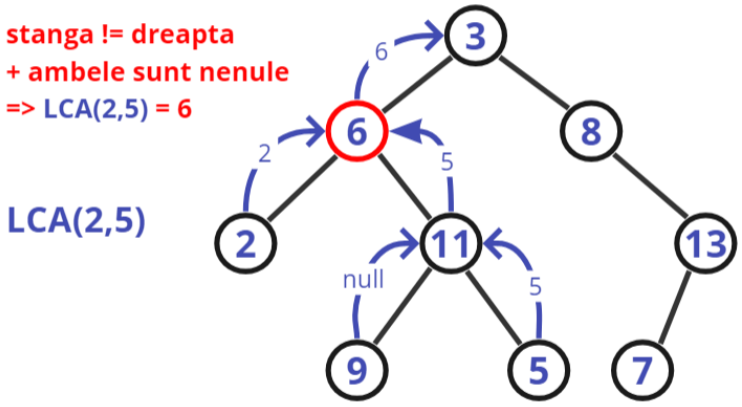
Observăm că dacă ne uităm la drumurile din rădăcină la cele două noduri pentru care vrem să găsim LCA, acestea sunt comune până la un punct. Ultimul nod comun din cele două drumuri este LCA-ul căutat, deoarece acela este punctul în care cele două drumuri o iau în părți diferite.

Brutul acestei soluții funcționează și se găsește aici, însă nu ne place pentru că are nevoie de spațiu auxiliar pentru stocarea celor două drumuri.

Pentru a îmbunătăți soluția, putem face observația că LCA este nodul pentru care p se află în stânga și q se află în dreapta. Putem construi o funcție recursivă care, pentru un nod dat, verifică dacă p sau q se află în arborele cu rădăcina în nodul respectiv. Apoi, când căutăm LCA, dacă suntem într-un nod pentru care p și q sunt în același subarbore (stâng sau drept), coborâm în fiul respectiv și reluăm recursiv operația. Dacă p și q sunt în subarbori diferiți, cum am spus, înseamnă că nodul curent este LCA. Implementarea este mai sugestivă și se găsește aici.

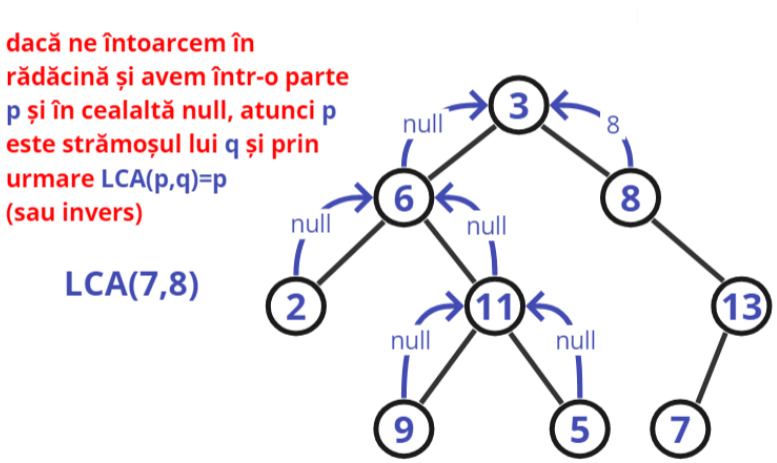
Această soluție repetă de mai multe ori aceiași pași atunci când caută nodurile p și q. Putem îmbunătăți acest lucru integrând funcția de căutare în funcția de LCA. Practic, vom căuta nodurile p și q, și când le vom găsi le vom returna. LCA va fi nodul pentru care subarborele stâng va returna p și subarborele drept va returna q (sau invers).

Dacă un nod returnează p, înseamnă că p a fost găsit în subarborele cu rădăcina în nodul respectiv (analog pentru q, exemplu nodul 11 returnează 5 pentru că q = 5 se găsește în



subarborele lui). Dacă un nod returnează un nod diferit de p și q, nodul acela este LCA și îl pasăm mai departe spre rădăcină.

Implementarea pentru soluția aceasta este atât mai eficientă, cât și mult mai scurtă (5 linii) și se găsește aici.



Complexitate timp: O(N) - unde N este numărul de noduri

Spațiu auxiliar: O(H) stivă - unde H este înălțimea arborelui

Problema 3:

*Se dă un vector de numere întregi. Răspundeți la întrebări de tipul "Care este suma elementelor din vector dintre indicii **x** și **y**?".*

Soluția 1:

Deoarece avem doar query-uri, fără update-uri (nu trebuie să facem modificări vectorului inițial), putem folosi sume parțiale. Fie **S** vectorul de sume parțiale, unde **S[i]** reține suma numerelor de la începutul vectorului până la indicele **i**. Calcularea acestui vector se face printr-o simplă parcurgere a numerelor. Suma dintre doi indici **x** și **y** este **S[y] - S[x – 1]** (din suma până la **y**, scădem ce este înainte de **x**).

Complexitate timp: O(N) preprocesare, O(1) query

Spaţiu auxiliar: O(N) preprocesare

Cod soluţie: https://pastebin.com/W6ZtmCab

Soluţia 2:

Putem folosi *şmenul lui Batog/sqrt decomposition*, calculând minimul pe bucăţi din vector de lungime **sqrt(n)**. Exemplu:

Avem "bucket-uri" de dimensiuni **sqrt(n)** (în exemplu, **sqrt(10) ≈ 3**).
Range_Min[i] = minimul pe "bucket-ul" **i**, unde **0 <= i <= n/sqrt(n)**

Indice	0	1	2	3	4	5	6	7	8	9
Valoare	3	9	2	8	5	3	8	7	6	11
Range_Min	2			3			6			11

query(2,6) = min(Valoare[2], Range_Min[1], Valoare[6])

Complexitate timp: O(N) preprocesare, O(sqrt(N)) query

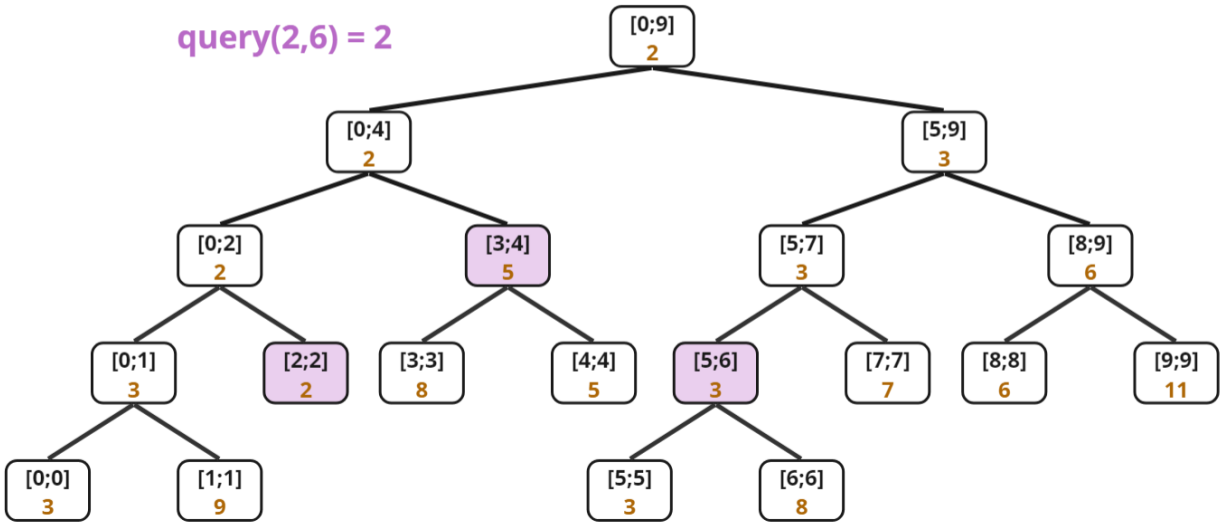
Spaţiu auxiliar: O(sqrt(N))

Cod soluţie: https://pastebin.com/5RUrVpVi

Soluţia 3:

Putem folosi un arbore de intervale, reţinând minimul pe fiecare interval. Exemplu:

Indice	0	1	2	3	4	5	6	7	8	9
Valoare	3	9	2	8	5	3	8	7	6	11



Complexitate timp: O(N) construire arbore, O(log N) query

Soluţia 4:

Putem adapta soluţia de la RMQ (range minimum query) să folosească sumă în loc de minim. Exemplu:

SumK[i] = suma elementelor dintre indicii **[i; 2^K - 1]**

Indice	0	1	2	3	4	5	6	7	8	9
Valoare	3	9	2	8	5	3	8	7	6	11
Sum2	12	11	10	13	8	11	15	13	17	11
Sum4	22	24	18	24	23	24	32	24	17	11
Sum8	45	48	50	48	40	35	32	24	17	11

query(1,6) = sum4[1] + sum2[5]
[1,4] [5,6]

query(2,9) = sum8[2]
[2,9]

query(3,6) = sum4[3]
[3,6]

Complexitate timp: O(N log N) preprocesare, O(log N) query

Spaţiu auxiliar: O(N log N)

Cod soluţie: https://pastebin.com/jjBf4Aep

Pe scurt, câteva întrebări de la examene/seminarii din anii anteriori:

1. Înălţimea unui arbore binar de căutare cu **N** elemente este între _ şi _.
2. Se dau **N** numere. Determinaţi câte triplete pitagoreice sunt printre ele.
3. Se dau mai multe perechi de numere **L** şi **R** ($L <= R <= 10^6$) şi se cere să spuneţi pentru fiecare, câte numere prime sunt în intervalul **[L;R]**.
4. Se dă un şir de **N** cuvinte. Se cere să se răspundă la query-uri de tipul "Câte anagrame ale cuvântului **X** sunt în şir?" (Bonus: Putem avea şi update-uri de tipul "Eliminaţi cuvântul de la indicele **y**.")
5. Pe o structură de date avem o operaţie numită Bla. Ştim că **N** utilizări au timpul **O(N log N)**. Care este timpul amortizat pentru o operaţie Bla? Care este timpul maxim pentru o singură operaţie Bla?

Răspunsuri:

1. **log N, (N – 1)**
2. Un triplet **(a, b, c)** de numere se numeşte triplet pitagoreic dacă $a^2 + b^2 = c^2$. În $O(N^2)$ punem într-un hash valorile $(a^2 + b^2)$ pentru toate perechile posibile **(a, b)**. Parcurgem numerele şi verificăm, pentru fiecare număr **c**, dacă există în hash-ul calculat anterior.
3. Folosim ciurul lui Eratostene (acesta are complexitate **O(N*log(log N))**) pentru precalcularea numerelor prime, şi sume parţiale pentru a reţine **P[i]** = câte numere mai mici decât **i** sunt prime. Răspunsul va fi **(P[R] - P[L-1])**.
4. Pentru preprocesare, sortăm fiecare cuvânt (**O(lungime_cuvant)** dacă folosim Count Sort) şi punem toate cuvintele într-un trie – **O(N * lungime_cuvant)**. Pentru a găsi numărul de anagrame ale unui cuvânt **X**, sortăm cuvântul (tot cu Count Sort) şi căutăm în trie în **O(lungime_cuvant)**. Alternativ, în loc de trie putem folosi un hash. Ambele variante permit şi update-uri uşor.
5. **O(log N), O(N log N)**