Univerzitetni programerski maraton 2024
2. kolo — rešitve nalog

Tomaž Hočevar
tomaz.hocevar@fri.uni-lj.si

4. april 2024

Bančni račun

Naloga zahteva samo implementacijo preverjanja po navodilih. Obsega predvsem manipulacijo nizov, za kar je najbrž najbolj prikladen jezik Python. Bodite pozorni na vse možne čudne vhodne podatke, ki lahko vsebujejo poljubne presledke, črke na sredini številke bančnega računa, izrazito prekratke številke, itd. Število, ki ga morate izračunati po modulu 97, je preveliko za 64-bitne številske tipe, zato morate računati rezultat po modulu sproti, uporabljati velika števila, ali pa uporabiti Python, kjer te težave nimate.

Koreni

Izbrati moramo koren drevesa, da bo njegova višina čim nižja. Pravzaprav moramo najti vse take korene. Ker je drevo dovolj majhno, lahko preverimo vseh N možnih korenov. Za vsak koren s poljubnim preiskovanjem drevesa izračunamo oddaljenosti vseh vozlišč od jega in tako dobimo globino drevesa. Časovna zahtevnost rešitve je $O(N^2)$.

Obstaja tudi hitrejša rešitev v $O(N)$ časa. Pravzaprav iščemo center/središče drevesa. Listi bodo vedno najbolj oddaljeni in prispevali ena k globini, zato jih lahko vse odrežemo in iščemo center obrezanega drevesa. To ponavljamo, dokler nam ne ostaneta največ dve vozlišči. Odgovor bo tako sestavljen iz enega ali dveh vozlišč.

Pozorni bodite na drevesa z dvema ali samo z enim vozliščem.

Knjižna zbirka

Podana imamo seznama števil a in b. Zanima nas število permutacij seznama b, da bo povsod veljalo $a_i > b_i$. Razmislek si poenostavimo z urejanjem seznamov a in b naraščajoče, kar ne spremeni rezultata. Element b_n lahko postavimo na katerokoli izmed zadnjih j mest, da še velja $a_{n-j} > b_n$. Naslednji element b_{n-1} lahko podobno postavimo na katerokoli izmed zadnjih k mest, kar poleg prejšnjih j vključuje morda še kakšnega. Eno izmed teh mest pa je že zasedeno, torej imamo za prvi dve števili $j \cdot (k-1)$ možnosti (ne glede na to, katero mesto je bilo zasedeno s prvim številom). Z enakim razmislekom obravnavamo preostala števila v seznamu b.

Število mest j v urejenem seznamu a, ki so večja od elementa b_i, lahko iščemo vsakič znova z bisekcijo, s čimer dobimo časovno zahtevnost $O(n \log n)$. Lahko pa upoštevamo dejstvo, da se ob vedno manjših številih b_i veča (ali ostane enako) tudi pripadajoče število mest j, zato lahko te lokacije poiščemo z enim prehodom in dobimo časovno zahtevnost $O(n)$.

Regularni izrazi

Zgenerirati je treba prvih 1000 elementov zaporedja, ki je definirano z regularnim izrazom. Najprej moramo razčleniti podan izraz v obvladljivo rekurzivno/gnezdeno/drevesno obliko, kjer vsako vozlišče predstavlja unijo ali stik ali elementarni izraz. Nato lahko za vsako vozlišče od listov proti korenu zgeneriramo prvih 1000 elementov zaporedja (več jih v vmesnih korakih zagotovo ne bomo potrebovali).

Unija je enostavna. Združimo sezname in ohranimo prvih 1000 elementov. Stik je bolj kompleksen. Vseh kombinacij je preveč, da bi lahko zgenerirali vse. Tudi če jih generiramo postopno, moramo paziti, saj je lahko zadnji element v stiku prazen, zato bi lahko zgenerirali eksponentno veliko delnih rezultatov, ki ne bi prispevali k nobenemu dejanskemu rezultatu. Generiramo jih lahko postopoma od leve proti desni z dodajanjem novega seznama do sedaj zgeneriranim kombinacijam in vedno ohranjamo samo prvih 1000 zgeneriranih, saj jih več kasneje ne bomo potrebovali.

Lahko pa smo bolj zviti in uporabimo kakšno od že obstoječih funkcionalnosti. Vhodni niz lahko preoblikujemo in uporabimo v Pythonu funkcijo `eval`. Definiramo tudi operatorje za unijo in stik, ki pa morajo podpirati leno evalvacijo, da se rezultati generirajo sproti in ne v celoti vnaprej, saj je tak rezultat prevelik.

Zlivanje

Seznam nizov moramo zliti v leksikografsko čim manjši rezultat. Začnimo z lažjim problemom, kjer zlivamo samo dva niza A in B. Če se razlikujeta v prvem znaku, bomo očitno najprej vzeli manjšega. Dilema pa nastane, če imata enak prvi znak. Recimo, da je drugi znak manjši v nizu A. Potem se nam splača najprej vzeti znak iz prvega niza, da bomo čim prej dosegli ta manjši znak. V splošnem se lahko seveda ujema večja predpona.

En niz je lahko tudi v celoti predpona drugega. Temu primeru se izognemo tako, da dodamo vsem nizom na konec znak, ki je večji od Z-ja, kar očitno ne spremeni rešitve, ker ga ne bomo nikoli uporabili, nizi pa sedaj ne bodo predpona kakšnega drugega.

Recimo sedaj, da neki skupni predponi v prvem nizu sledi znak a, v drugem pa b (recimo, da je $a < b$). Dokažemo lahko, da bo v optimalnem zlitem rezultatu vedno nastopal znak a pred znakom b (sicer bi lahko ravno obrnili izbire iz prvega in drugega niza in dobili boljšo rešitev). To lahko dosežemo tako, da izberemo prvi znak iz leksikografsko manjšega niza, v prejšnjem primeru torej A (ker je $a < b$). Recimo, da bi v optimalni rešitvi (kjer se a pojavi pred b) najprej izbrali znak iz drugega niza. Potem lahko najdemo neko predpono rezultata, ki vključuje enako število znakov iz obeh nizov (ta predpona se bo nahajala pred pojavitvijo a-ja) in brez škode zamenjamo izbire iz prvega in drugega niza.

Če imamo opravka s seznamom N nizov (s skupino dolžino D), moramo na vsakem koraku izbrati leksikografsko najmanjšega med njimi in ga skrajšati za en znak. Primerjave takih pripon lahko delamo učinkovito, če vse nize zložimo skupaj v en daljši niz ter nad njim zgradimo priponsko polje (suffix array) v $O(D \log^2 D)$ časa. Za vsako pripono lahko sedaj v konstantnem času ugotovimo, kje v priponskem polju se nahaja (kako "majhna" je). Ker na vsakem od D korakov hkrati primerjamo N takih pripon, lahko njihove indekse iz priponskega polja hranimo v prioritetni vrsti, kar nam k časovni zahtevnosti doda še člen $O(D \log N)$.

Črtni diagram

Problem lahko modeliramo z grafom. Vsaki črti ustreza vozlišče, dve vozlišči pa sta povezani, če se pripadajoča grafa kje sekata. V takem grafu nas zanima minimalno število barv (vsaka barva ustreza svojemu diagramu), s katerimi lahko pobarvamo vozlišča, da nobeni dve sosednji (sekajoči se) ne bosta iste barve. Na žalost to ni bistveno lažji problem.

Lahko pa problem modeliramo malo drugače. Črte formirajo delno urejeno množico (partially ordered set), kjer vsako črto povežemo s črtami, ki se v celoti nahajajo nad njo. Naloga sprašuje po najmanjšem število verig (chains) v tej delno urejeni množici. Ta problem se modelira z dvodelnim grafom, kjer vsaka črta ustreza vozlišču na levi in desni strani dvodelnega grafa, povezave iz leve na desno stran pa predstavljajo relacijo "nad". V tem grafu nas zanima največje ujemanje (maximum matching). Predstavljamo si lahko, da je na začetku vsaka črta na svojem diagramu, dodatek povezave v ujemanje pa združi pripadajoči črti na isti diagram (leva se nahaja neposredno pod desno).

Klasično iskanje največjega ujemanja v takem grafu z n vozlišči in $m = O(n^2)$ povezavami ima lahko časovno zahtevnost $O(n^3)$, kar je prepočasi. Iskanje ujemanja lahko pohitrimo tako, da najprej izvedemo neko vrsto hitrega požrešnega ujemanja in šele nato poskusimo povečati preostanek s klasičnim postopkom iskanja alternirajočih poti. Druga možnost pa je, da uporabimo Hopcroft-Karpov algoritem s časovno zahtevnostjo $O(m\sqrt{n}) = O(n^{2.5})$.