

1. Alapfogalmak

Az eddig vizsgált algoritmusok analízise a legrosszabb eset analízisén alapultak. Egy inputon futtattuk és az input hosszának/méretének függvényében egy korlátot adtunk a futási időre.

Sokszor egy algoritmus összetett műveleteket ismétel sokszor. Ha csak egy összetett művelet legrosszabb eset analízisét végezzük el és ezt megszorozzuk a művelet elvégzésének számával, akkor „rosszul járhatunk”. Jobb, ha a művelet átlagos lépésszámával dolgozunk. Ha az algoritmus sokszor végrehajt egy műveletet, akkor nem zavaró ha egyszer sok lépést végez el, míg máskor lényegesen kevesebbet. A teljes algoritmus futási ideje az átlagos lépésszámtól függ.

Ha a fenti „filozófia” szerint elemzünk egy ismételt műveletet, akkor amortizált analízisről beszélünk. A fogalmat legjobban példákon keresztül megismerni.

2. Példák

1. példa: Adott az $0^n = (0, 0, \dots, 0)$ n -hosszú bitsorozat. Vehetjük úgy is mint a 0 szám kettes számrendszerbeli felírásának számjegyei. $2^n - 1$ -szer növeljük meg a bitsorozat által kódolt természetes számot 1-gyel és a megnövelt szám n hosszú (esetleges kezdeti 0-kkal felduzzasztott) kettes számrendszerbeli felírását számoljuk ki. Minden növelés költsége az előző sorozathoz képest megváltoztatott bitek száma lesz. Mi a teljes algoritmus összköltsége?

A feladat nem túl érdekes, de elemzése tanulságos lesz. Ha sorozatunk $01^{n-1} = 011\dots 11$, akkor növelése után elérjük az $10^{n-1} = 100\dots 00$ sorozatot, Ennek költsége n , a lehető legnagyobb költség. Így a $2^n - 1$ növelés összeköltsége becsülhető $(2^n - 1)n$ -nel.

Gondolkodjunk másképp. Tegyük fel, hogy egy számjegy megváltoztatásához valójában ki kell fizetnünk 1 \$-t. Ha a legrosszabb esetre készülnénk fel, akkor n \$-t kell magunkkal vinnünk egy növeléshez. Ha azonban jó gazdasági érzékkel rendelkezünk, akkor 2 \$-ral boldogulunk. Az egyik \$-unk „0-ról 1-re írás” költsége, a másik \$-unk „1-ről 0-ra írás” költsége. Ha egy 0-t 1-esre írunk át, akkor a hozott 2 \$ egyikével ki tudjuk fizetni és az átírt 1-esnek ott tudjuk hagyni a második \$-t mind leendő átírási költséget.

Mi van, ha az átírás nem csak az utolsó 0 megváltoztatása 1-re. Akkor sincs baj. A fenti gondolatmenet alapján számjegysorozatunk minden 1-ese a leendő 0-ra írásának költségét „letétben tartja”. Az általános „update” a számjegyek egy végső 1-blokkjának 0-kkal való felülírása és az előtte álló 0-s 1-esé írása. Az 1-esek 0-ra írásának költsége ott van letétben, végül a blokk előtti 0 átírását a hozott 2 \$-ból fedezzük, a maradék 1 \$-t letétként hagyjuk az új 1-es számjegynél. Így a $2^n - 1$

művelethez $2(2^n - 1)$ \$-ra van szükségünk. Ez fedezi az összes költséget. Sőt az algoritmus végén ott van $1^n = 11 \dots 11$ szám mindegyik 1-esén 1 \$ letét. Tehát a számolási költség pontosan $2(2^n - 1) - n$.

Összefoglalva. Minden növelést 2 költséggel számoltunk el. Tettük ez annak ellenére, hogy minden második növelésünk 1 költségű volt. Ezekben a növelésekben a költség felhasználása az átírásért való fizetés és letét képzése. Általában a korábbi letétekkel és a hozott 2 költséggel gazdálkodtunk (fizettünk átírásainkért és újabb letéteket képeztünk). A növelés *amortizációs költsége* 2. A fenti interpretációt a *bankár értelmezésének* nevezzük.

Van egy másik értelmezés is, a *fizikus értelmezése*: Az aktuális szám 1-eseinek számát egy potenciálnak fogjuk fel. Tehát kezdetben 0 potenciálról indulunk. Minden lépésben változtatjuk a potenciált. Ha a potenciált növeljük, akkor azért is fizetnünk kell. Ha viszont potenciált veszünk, akkor ez fizetési eszköt ad nekünk. 2 költséggel boldogulunk. A költségből 1-et a potenciál növelésére fordítunk, a másiból és esetleg a meglévő potenciál egy részéből (a potenciál csökkenni fog) fizetjük az átírási költséget. Csak a nyelvezet és így a gondolkodásmód más, de ugyanazt a matematikai ötletet „kódoljuk”.

2. példa: Adott n pont a koordináta-síkon: $(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n)$ úgy, hogy x -koordinátáik szigorúan monoton növekvők. Határozzuk meg az input ponthalmaz konvex burkát (mely pontok, milyen körszerűen rendezett sorrendben szerepelnek a konvex borkon). A konvex buroknak lesz egy x koordinátára nézve minimális csúcsa és egy maximális csúcsa. Ez a két csúcs a konvex burok kerületét két ívre osztja. Ezeket (természetes módon) a ponthalmazunk felső és alsó konvex burkolójának nevezzük. A két rész ami együtt (a két extrémális pont átfedésével) kiadja a teljes konvex burkot. A két burkoló szimmetrikus szerepet játszik. Elegendő azt tárgyalnunk, hogyan határozható meg a felső burkoló. Ezt részletezzük.

Vegyük az első i pont halmazát és tegyük fel ismerjük

$$\mathcal{F}_i : E = P_1, P_{i_2}, P_{i_3}, \dots, P_{i_{\ell-1}}, P_i$$

felső burkolójukat (ℓ jelöli a felső burkoló csúcsainak számát). Szúrjuk be a P_{i+1} pontot ebbe a felső burkolóba. Ezt a beszúrást ismételtetjük az algoritmus során.

A beszúrás nyilvánvaló: Az új felső burkoló az előző felső burkoló egy kezdőszelete lesz. Ez a kezdőszelet $U = P_{i_k}$ -ban végződik, majd jön a P_{i+1} csúcs. Tehát az új felső burkoló:

$$\mathcal{F}_i : E = P_1, P_{i_2}, P_{i_3}, \dots, P_{i_{k-1}}, U = P_{i_k}, P_{i+1}.$$

Az U pont úgy azonosítható, hogy a régi felső burkoló egyetlen U csúcs teljesíti a

$$m(P_{i_{k-1}}P_{i+1}) > m(UP_{i+1}) \leq m(P_{i_{k+1}}P_{i+1})$$

feltételt, ahol $m(AB)$ az A és B pontok által meghatározott egyenes meredeksége.

A beszúrásra/ $U = P_{i_k}$ megtalálására több lehetőségünk van.

I) Például tippelhetünk egy P_{i_j} csúcsra (az egyszerűség kedvéért tegyük fel, hogy $1 < j < \ell$). Három lehetőségünk van:

(a) $m(P_{i_{j-1}}P_{i+1}) > m(P_jP_{i+1}) \geq m(P_{i_{j+1}}P_{i+1})$. Ekkor eltaláltuk a korrekt U -t.

(b) $m(P_{i_{j-1}}P_{i+1}), m(P_{i_{j+1}}P_{i+1}) > m(P_jP_{i+1})$. Ekkor a keresett U indexe (mint indexelt P) nagyobb mint j .

(c) $m(P_{i_{j-1}}P_{i+1}), m(P_{i_{j+1}}P_{i+1}) \geq m(P_jP_{i+1})$. Ekkor a keresett U indexe (mint indexelt P) kisebb mint j .

Erre alapítva egy bináris keresést végezhetünk. $\mathcal{O}(\log \ell) = \mathcal{O}(\log i)$ kérdéssel megtaláljuk a keresett P_j -t. Minden „kérdés” költsége $\mathcal{O}(1)$ aritmetikai és összehasonlítási lépés. Tehát a beszűrés költsége $\mathcal{O}(\log i)$. A teljes költség $\mathcal{O}(\log 1 + \log 2 + \dots + \log(n-1)) = \mathcal{O}(n \log n)$.

II) Egy sokkal naívabb próbálkozás, hogy teszteljük P_{i_ℓ} csúcsot. Ha ez nem jó U szerepére, akkor teszteljük a $P_{i_{\ell-1}}$ csúcsot. Ha ez nem jó U szerepére, akkor teszteljük a $P_{i_{\ell-2}}$ csúcsot. És így tovább. Talán meglepő, de ez az előzőnél jobb algoritmust ad. Mi ennek az oka? Ezt az amortizált analízis válaszolja meg.

Az $U = P_{i_k}$ csúcs megtalálása után a $P_{i_{k+1}}, P_{i_{k+2}}, \dots, P_{i_{\ell}} = P_i$ csúcsok kiesnek a felső burkolóból. Sőt, soha nem is kerülhetnek vissza. Azaz a naív algoritmus minden sikertelen tesztje egy csúcs kiesésével jár. Legyen $c = \mathcal{O}(1)$ a költsége egy tesztnek. A P_{i+1} csúcs beszűrésát $2c$ amortizációs költséggel számoljuk el. Ebből c egy „bekerülési költség”. Ezt akkor fizetjük ki az aktuális tesztre, amikor megtaláljuk az U csúcsot. A másik c költség a bekerült P_{i+1} csúcs „kikerülési költsége”, amit letétbe helyezünk a csúcsnál. Ezt akkor fizetjük ki (amennyiben P_{i+1} nincs a felső burkoló csúcsai között), amikor egy későbbi pont teszteli őt sikertelenül. Tehát minden sikertelen tesztet egy korábbi csúcs letéti összegéből fizetünk. Az amortizáció költség $2c = \mathcal{O}(1)$. A teljes költség $\mathcal{O}(n)$.

3. példa: Dinic-algoritmus. Adott egy \vec{G} irányított gráf két kitüntetett csúccsal: s és t . A gráf rendelkezik a következő tulajdonsággal:

(D) minden éle rajta van egy legrövidebb \vec{st} úton.

Az algoritmus során rendre egy e élt kapunk az aktuális gráfból, amit elhagyunk, majd a gráf további ritkítását végezzük el (ha szükséges), hogy a (D) tulajdonság továbbra is fennáljon. Az éleket addig kapjuk, amíg gráfunk ki nem ürül.

Az első naív gondolat: $|E|$ sok élt kaphatunk törlésre, egy törlés hatása lehet akkora, hogy $\mathcal{O}(|E|)$ költséggel jár. Lássuk mit ad az amortizációs analízis.

A (D) tulajdonság miatt gráfunk szintezett:

$$V(G) = S_0 \dot{\cup} S_1 \dot{\cup} \dots \dot{\cup} S_{\ell-1} \dot{\cup} S_\ell,$$

ahol $S_0 = \{s\}$, $S_\ell = \{t\}$ és minden él valamely i -re S_i -ből S_{i+1} -be vezet.

Legyen $e = \vec{xy}$ az aktuális törlendő él ($x \in S_i$ és $y \in S_{i+1}$). Az e él elhagyása további ritkítást nem igényel, ha y -ba vezet e -től eltérő él is, és x -ből vezet ki e -től eltérő él is.

Ha x kifoka 1, akkor a korábbi szintek közt lesz olyan él ami már nem lesz rajta \vec{st} úton. Például az összes x -be futó él ilyen, ezeket el kell hagyni. Ha azon csúcsok, amikből vezetett él x -hez 1 kifokúak voltak, akkor továbbgyűrűzzük a hatás visszafelé.

Hasonló a hatás, ha y befoka 1. Ekkor az y -ből kifutó élek válnak feleslegessé (nem lesznek rajta \vec{st} úton). Törlésüknek lesz további hatása: előre felé (a nagyobb indexű szintek felé) gyűrűzhet.

Algoritmusunk a fenti gondolatmenet naív végrehajtása: e törlése után követjük a törlés hatását. Előre és hátra is rendre teszteljük, hogy a megfelelő (lásd fenn) fok 1 vagy nagyobb. Ha valamelyik fok 1, akkor a törlendő élek listáját (ami

kezdetben egyetlen e élből állt) bővítjük (lásd fenn). Majd tovább törölünk: az aktuális törlendő élet töröljük, ennek hatásának vizsgálatával. Ha a hatás „lecseng”, akkor a következő élt kezeljük.

Az analízis remélhetőleg már nem okoz gondot. Egy e él esetén $\mathcal{O}(1)$ költsége annak, hogy töröljük gráfunkból és felismerjük gyűrűzik-e előre vagy hátra az él elhagyásának hatása. Ezt a költséget e -re hárítjuk. Ha az x -be befutó élek (vagy y -ből kifutó élek) is a törlendő élek listájára kerülnek, akkor az ezekkel kapcsolatos költségeket már az aktuális él viseli. Az egész algoritmus $\mathcal{O}(|E|)$ lépéssel megvalósítható.

4. példa: Az utolsó példánk a Fibonacci-kupacok tárgyalása. Ez különösen fontos volt az amortizációs analízis fontosságának felismerésében. Ezt a példát külön tárgyaljuk.