ANALIZĂ AMORTIZATĂ

Responsabil: Radu Iacob (2017), Darius-Florentin Neațu (2016)

Noiembrie 2017

CUPRINS

1 ANALIZA AMORTIZATĂ

1.1 De ce o studiem?

Există operații, aplicate asupra unor structuri de date, a căror timp de execuție variază foarte mult pe parcursul unui program. De exemplu, este posibil ca operația aplicată să fie ineficientă într-o anumită situație, care să apară însă relativ rar pe parcursul execuției programului. In acest context este interesant să investigăm timpul de execuție pentru întreaga secventă de operații de același tip din algoritm.

La structuri de date ați implementat **hash tables** care permiteau permiteau efectuarea eficientă, în $O(1)$, a operațiilor de căutare si inserare de elemente. Această afirmație se bazează pe presupunerea că numărul de elemente stocate este comparabil cu numărul de buckets (liste/vectori), altfel complexitatea pentru căutare este $O(n)$. În practică, dacă condițiile de mai sus sunt îndeplinite, vom spune ca implementarea de hash table are $O(1)$ **costul amortizat** pentru operațiile de cautare, respectiv inserare.

1.2 Ce este analiza amortizată?

Definiție 1. Analiza amortizată este o metoda de a analiza o secvență de n operatii, din perspectiva resurselor consumate (timp/memorie), pentru cazul cel mai defavorabil.

Folosind analiza amortizată, putem demonstra că pentru o operație individuală din secvență costul mediu este mic, chiar dacă câteva operații din secvență pot fi costisitoare.

Analiza amortizată diferă de analiza medie probabilistică (average-case analysis) prin faptul că rezultatele sunt **exacte** (noțiunea de probabilitate nu este inclusă). Analiza amortizată **garantează** că performanța medie pe operație în cazul cel mai defavorabil este cea calculată.

1.3 Ce NU este analiza amortizată?

NU este o tehnică de design pentru algoritmi. Dacă vi se dă o problemă, nu începeți prin a spune: "Pot să aplic tiparul general al unui algoritm de analiză amortizată și să îl modific pentru a obține soluție la problema mea." **NU** putem folosi analiza amortizată pentru a proiecta un algoritm.

NU este o modalitate de a reface un algoritm existent. Analiza amortizată nu schimbă în nici un fel algoritmul, ci este doar un mod mai precis de evaluare a timpului cel mai defavorabil de execuție a unei secvențe de operații (mult mai precis decât dacă am însuma toți timpii worst-case pentru toate operațiile din secvență).

2 CONTOR BINAR (BINARY COUNTER)

Fie un contor binar pe k biți, care numără crescător începând de la 0. Vom folosi notația $A[0..k-1]$ pentru a descrie un șir de k biți, cu precizarea că bitul cel mai puțin semnificativ (least significant bit - LSB) este $A[0]$, iar bitul cel mai semnificativ (most significant bit - MSB) este $A[k-1]$.

Dacă x este numărul curent arătat de contor, atunci $x = \sum_{i=0}^{k-1} A[i] * 2^i$. Inițial avem $x = 0$, deci $A[i] = 0$.

Pentru a adăuga 1 (modulo 2^k) la valoarea contorului, vom folosi următorul pseudocod.

```
function Inc(A) {
        // i = rangul binar curent
        i = 0;

        // daca adun 1, trebuie sa propag transportul
        // pana cand gasesc un zero
        // 0 + 1 = 1 si nu mai am transport
        while (i < A.length && A[i] == 1) {
                A[i] = 0;
                i++;
        }

        // am gasit un zero daca nu am facut overflow
        if (i < A.length) {
                A[i] = 1;
        }
}
```

Care este costul amortizat pentru o operație de incrementare?

3 TEHNICI DE CALCUL

Analiza amortizată este un subiect acoperit în multe publicații. Recomandăm să consultați "Introduction to algorithms" [1], respectiv "Introducere în Analiza Algoritmilor" [2]. Acest concept este explicat pe scurt și în "Algorithm Design: Foundations, Analysis and Internet Examples" [3].

Cele mai comune 3 tehnici folosite în analiza amortizată sunt: analiza agregată, metoda creditelor și metoda potențialului.

3.1 Analiza agregată (aggregate analysis)

În **analiza agregată**, vom arăta că pentru oricare secvență de **n** operații, în cel **worst-case time** este în total $T(n)$.

Definiție 2 (Cost amortizat - metoda agregată). În cel mai rău caz, costul mediu sau **costul amortizat** pe operație este $\frac{T(n)}{n}$.

ATENȚIE: Acest cost amortizat se aplică fiecărei operații din secvență, chiar dacă sunt mai multe tipuri de operații, iar unele au cost real mai mare. Celelate două metode studiate pot asocia costuri diferite pentru fiecare operație.

3.1.1 *Contor binar*

Observăm că atunci când facem o operație de **Inc** nu toți biții se modifică (se ia o decizie și se stabilește care biți se modfică la fiecare pas).

- A[0] se modifică după fiecare iterație $\rightarrow$ din 2^0 în 2^0 pași
- A[1] se modifică după fiecare două iterații $\rightarrow$ din 2^1 în 2^1 pași
- A[2] se modifică după fiecare patru iterații $\rightarrow$ din 2^2 în 2^2 pași
- ....
- A[j] se modifică după 2^j incrementări

Pentru a calcula numărul total de operații rescriem astfel:

- A[0] se modifică de $\frac{n}{2^0}$ ori
- A[1] se modifică de $\frac{n}{2^1}$ ori
- A[2] se modifică de $\frac{n}{2^2}$ ori
- ....
- A[j] se modifică de $\frac{n}{2^j}$ ori

Adunând precedentele k relații ($0 \leqslant j \leqslant k-1$), obținem:

$$T(n) = \sum_{j=0}^{k-1} \frac{n}{2^j} = n \sum_{j=0}^{k-1} \frac{1}{2^j} < n \sum_{j=0}^{\infty} \frac{1}{2^j} = 2n$$

Prin urmare, **costul total amortizat** este $T(n) \leqslant 2n$, adică $T(n) = O(n)$.
Pentru o operație **costul amortizat** este $\frac{T(n)}{n} \leqslant 2$, adică $O(1)$.

3.2 Metoda creditelor (accounting method)

În tehnica de analiză amortizată numită **metoda creditelor**, asignăm diferite costuri (sume) fiecărei operații, unele având un cost mai mic sau mai mare față de costul real.

Definiție 3 (Cost amortizat - metoda creditelor). Costul amortizat al unei operații este suma cu care a fost marcată o operație.

Dacă costul amortizat al unei operații depășește costul real, diferența o retinem într-o structură specială numită **credit**. Creditele ne ajută să compensăm operațiile ulterioare al căror cost amortizat este mai mic decât costul real.

Putem vedea costul amortizat al unei operații ca fiind împărțit în costul actual și credit (care poate fi depozitat sau folosit).

Operații diferite pot avea cost amortizat diferit. Trebuie **să alegem** costul amortizat al unei operații cu foarte mare atenție. Dacă vrem să arătăm, folosind această tehnică, că în cel mai rău caz costul mediu pe operație este mic, trebuie să ne asigurăm că întotdeauna costul total amortizat al secvenței de operații este o limită superioară a costului total real.

Pentru orice secvență de n operații obținem:

$$\sum_{i=1}^{n} \hat{c}_i \geqslant \sum_{i=1}^{n} c_i$$

- n = numărul de operații din secvența dată
- c_i = **costul real** al operației cu numărul i din secvență
- $\hat{c}_i$ = **costul amortizat** al operației cu numărul i din secvență

Cu aceste notații putem defini și creditul pentru o operație:

- $credit_i$ = creditul asociat operației i
- $credit_i = \hat{c}_i - c_i$
- $credit_i \geqslant 0 \iff$ "depunem" bani în cont
- $credit_i < 0 \iff$ "retragem" bani din cont

Creditul total acumulat în structura de date este diferența între costul total amortizat și costul total real. Acesta trebuie să fie mereu pozitiv (nu putem cheltui mai mult decât avem).

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i \geqslant 0$$

3.2.1 *Contor binar*

Considerăm același contor binar pe k biți despre care s-a vorbit anterior.

Facem observația că operația **Inc** schimbă doar starea unor biți (tranziții 0 în 1 și 1 în 0). Astfel definim următoarele operații:

- **set** = reprezintă o tranziție din 0 **în** 1 a unui bit (oarecare)
- **reset** = reprezintă o tranziție din 1 **în** 0 a unui bit (oarecare)

Pentru cele două operații avem costurile reale $c_{set} = c_{reset} = 1$. În ideea că poate pot plăti în avans prin operația set, strângând credite pe care să le pot consuma în operația reset, aleg următoarele costuri amortizate:

- $\hat{c}_{set} = 2$
- $\hat{c}_{reset} = 0$

Operația **Inc** schimbă un singur bit de 0 în 1 (face o singură operație set), iar toți biții mai puțin semnificativi decât acesta, sunt făcuți 0 (se aplică pe toți operația reset).

De fiecare dată când fac un **set**, ar trebui să platesc 2 credite (costul amortizat ales de mine), dar costul real este 1, astfel încât consum un credit pentru operația efectivă de set, iar un credit îl pun în cont; acest credit va fi folosit atunci când același bit va trece din 1 în 0 (reset).

Se observă că, în fiecare moment, în cont există un credit pentru fiecare bit de 1 din număr. Putem afirma că nu plătim nimic în plus pentru un reset (fiecare bit de 1 are creditul lui deja - "își plătește singur"). Prin urmare, costul de resetare, care este plătit în while, este acoperit de creditele din cont.

Deoarece numărul de credite din cont este egal cu numărul de biți de 1 din număr, iar numărul nu poate avea "un număr negativ de biți de 1" , rezultă mereu că în cont avem o sumă nenegativă. Deci **am ales corect** costurile amortizate pentru cele două operații.

O operație **Inc** este compusă din maximum o operație de set (la overflow nu face set - putem spune că vom reseta contorul) și câteva operații de reset. Cele din urmă au cost amortizat 0, deci o operație de **Inc** are costul amortizat cel mult 2.

Evident, obținem aceeași concluzie: pentru o operație **costul amortizat** este $O(1)$.

3.3 Metoda potențialului (potential method)

În loc să reprezentăm munca făcută în plus ca și credite acumulate, **metoda potențialului** de analiză amortizată reprezintă munca ca "energie potențială" sau "potențial", care poate fi eliberată pentru a plăti operații. Asociem potențialul mai degrabă cu structura de date ca și un întreg (nu mai considerăm obiectele ca fiind individuale).

Vom face următoarele notații:

- D_0 = structura de date inițială, pe care vom executa n operații
- c_i = **costul real** al operației i
- D_i = **structura de date** care rezultă după aplicarea operației i asupra structurii de date D_{i-1}
- $\phi(D_i)$ = **funcție de potențial** care asociază fiecărei structuri D_i un număr real, numit potențialul ("energia") structurii de date

- $\hat{c}_i$ = **costul amortizat** al operației i în raport cu funcția de potențial dată

Asemănator legilor fizicii, variația de potențial este cea care modifică costul ("starea") curent: costul amortizat al unei operații este costul real la care se adaugă variația de potențial.

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

Însumând cele n egalități obținem relația care ne arată dependența între costul total amortizat și costul total real.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

Precedenta relație ne arată că dacă vrem să majorăm costul real folosind costul amortizat atunci $\phi(D_n) \geqslant \phi(D_0)$.

În practică nu știm întotdeauna de la început câte operații se vor executa. Dacă impunem ca $\phi(D_i) \geqslant \phi(D_0)$, $\forall i \leqslant n$, atunci garantăm că, la fel ca la metoda creditelor, plătim în avans.

De obicei, alegem $\phi(D_0) = 0$ și doar demonstrăm că $\phi(D_i) \geqslant 0$

Definiție 4 (Cost amortizat - metoda potențialului). Dacă s-a definit o funcție de potențial, pentru structura de date, care este inițial zero și întotdeauna nenegativ, costul amortizat al unei operații este costul real la care se adaugă diferența de potențial.

3.3.1 *Contor binar*

Pentru a alege o funcție de potențial pentru problema contorului, ne gândim cum "curge energia". Se pornește de la 0, se tot adaugă 1 și numărul de biți de 1 crește ("energia crește"), iar din când în când (mai exact cand numărul curent este de forma $2^m - 1$), toți biți se resetează ("energia este consumată") și se rămâne cu un singur bit de 1 (cel setat).

Observând această analogie, alegem funcția de potențial

$$\phi(D_i) = b_i$$

În continuare voi descrie notațiile folosite.

- $\phi(D_i)$ = potențialul contorului după aplicarea operației cu numărul i
- $\phi(D_0)$ = potențialul inițial al contorului (înainte de prima operație)
- b_i = numărul de biți de 1 după aplicarea operației cu numărul i
- t_i = numărul de biți de 1 care sunt facuți 0 de către operația i
- c_i = costul real al operației i
- $\hat{c}_i$ = costul amortizat al operației i

Ca să putem aplica metoda, funcția aleasă de noi trebuie se respecte condiția

$$\phi(D_i) \geqslant \phi(D_0)$$

.

Presupunem că numărătoarea începe de la 0, deci $b_0 = 0$, deci și $\phi(D_0) = 0$. Atunci este suficient să arătăm că $\phi(D_i) \geqslant 0$. Condiția este mereu verificată întrucât mereu avem un număr nenegativ de biți de 1.

Pentru a calcula costul amortizat aplicăm formula menționată în teorie.

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1})$$

Deoarece operația i resetează t_i biți, apoi face un set, costul real al operației i este

$$c_i = 1 + t_i$$

(durează t_i unități să reseteze t_i biți; durează 1 unitate să seteze 1 bit).

În continuare calculăm variația de potențial.

Numărul de biți de 1 de la pasul curent, este numărul de biți de la pasul anterior, din care scădem numărul de biți resetați la pasul curent, apoi adăugăm și bitul setat la acest pas.

$$b_i = b_{i-1} - t_i + 1$$

Relația anterioară merge doar dacă $b_i > 0$. Dacă $b_i = 0$ ($i > 0$), înseamnă că la pasul anterior toți cei k biți erau 1 ($b_{i-1} = k$) și la pasul curent toți au fost făcuți 0 ($t_i = k$). Nu se mai setează bit la acest pas (vezi condiția de la finalul ciclului). În acest caz particular avem

$$b_i = b_{i-1} - t_i < b_{i-1} - t_i + 1$$

Pentru o operație oarecare i, avem următoarea relație

$$b_i \leqslant b_{i-1} - t_i + 1 \iff \phi(D_i) \leqslant \phi(D_{i-1}) - t_i + 1$$

Din precedenta relație putem scoate variația de potențial (necesară în formula de cost amortizat).

$$\phi(D_i) - \phi(D_{i-1}) \leqslant 1 - t_i$$

Înlocuim rezultatele obținute pentru a afla costul amortizat.

$$\hat{c}_i = c_i + \phi(D_i) - \phi(D_{i-1} = (1 + t_i) + \phi(D_i) - \phi(D_{i-1}) \leqslant (1 + t_i) + (1 - t_i) = 2$$

Costul amortizat pentru operația **Inc** este 2. Prin urmare aceasta are cost O(1) amortizat.

4 EXERCIȚII PROPUSE

4.1 O problemă oarecare

Să presupunem că avem de efectuat o succesiune de n operații pe o structură de date în care operația cu numărul i are asociat costul c_i de mai jos.

$$c_i = \begin{cases} i, & \text{dacă i este putere a lui 2} \\ 1, & \text{altfel} \end{cases}$$

Folosiți metoda agregării și/sau metoda creditelor pentru a determina costul amortizat per operație.

4.2 Stivă cu backup (backup stack)

Să presupunem că avem de efectuat o succesiune de n operații pe o stivă a cărei dimensiune niciodată nu depașeste o valoare k. După fiecare k operații, facem o copie (backup) întregii stive.

Calculați, folosind metoda creditelor, costul a n operații pe stivă, incluzând operațiile de backup.

Hint: Se poate obține costul total $O(n)$ dacă se aleg costurile amortizate corespunzătoare.

4.3 Vector redimensionabil (resizable array)

Un vector redimensionabil este o structură de date care stochează o secvență de elemente și suportă operațiile de mai jos.

- **AddToEnd(D, x)**: adaugă elementul x la sfârșitul secvenței
- **LookUp(D, k)**: returnează al k-lea element din secvență sau NULL dacă lungimea curentă a secvenței este mai mică decât k

4.4 Vector extensibil (extendable array)

Un vector extensibil este o structură de date care stochează o secvență de elemente și suportă operațiile de mai jos.

- **AddToFront(D, x)**: adaugă elementul x la începutul secvenței
- **AddToEnd(D, x)**: adaugă elementul x la sfârșitul secvenței
- **LookUp(D, k)**: returnează al k-lea element din secvență sau NULL dacă lungimea curentă a secvenței este mai mică decât k

Descrieți o simplă structură de date care implementează algoritmul cerut. Primele două operații trebuie să aibă un cost amortizat $O(1)$, operația **LookUp** are

worst-case O(1). De asemenea, structura voastră ar trebui să ocupe un spațiu $\theta(n)$, unde n este lungimea curentă a numărului de element din subsecvență.

4.5 O structură de date nouă

Folosind ideea de la Insertion Sort vrem să construim o structură de date are să suporte executarea în mod eficient a operațiilor SEARCH si INSERT.

Operația de SEARCH se execută în $O(\log(n))$ pe un vector sortat (folosing căutare binară). Din păcate, operația INSERT rulează în $O(n)$, întrucât în cel mai defavorabil caz suntem nevoiți să shiftăm toate elementele pentru a face loc pentru cel nou.

Presupunem că dorim să suportăm o secvență den operații SEARCH și INSERT. Fie $< n_{k-1}n_{k-2}...n_0 >$ reprezentarea binară a lui n (unde $k = [\log(n+1)]$).

În loc să folosim un singur vector sortat de dimensiune n (numit **v**), vom folosi k vectori sortați, numiti $v_0, v_1, ..., v_{k-1}$, care respectă relația următoare.

$$|v_i| = \begin{cases} i, & \text{dacă } n_i = 1 \\ 0, & \text{dacă } n_i = 0 \end{cases}$$

Se observă că, deși fiecare vector este sortat, **nu** există o relație particulară între elementele din vectori diferiți.

Operația de SEARCH se realizează făcând căutare binară.

Operația de INSERT se realizează ca în următorul pseudocod, complexitatea fiind $O(n)$.

```
function Insert(A[1..k], x) {
        // vector care contine doar elementul
        newArray = {x};

        i = 0;
        while (A[i] is NOT empty) {
                // apeleaza subrutina de la Merge Sort
                newArray = merge(A[i], newArray);

                A[i] = empty;
                i++;
        }

        A[i] = newArray;
}
```

Se cere să se determine raspunsul pentru următoarele cerințe:

- Complexitatea operației SEARCH (worst-case).
- Costul amortizat pentru o secvență de n operații de tip INSERT, utilizând metoda potențialului.

- Dați exemple de structuri de date învățate care oferă un timp de rulare mai bun pentru ambele operații.

Hint: $\phi(D_i) = \sum_{j=0}^{k-1} (k-j) n_j 2^j$

BIBLIOGRAFIE

[1] Charles E Leiserson, Ronald L Rivest, Clifford Stein, and Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[2] Cristian A Giumale. *Introducere în analiza algoritmilor: teorie şi aplicaţie*. Polirom, 2004.

[3] Michael T Goodrich and Roberto Tamassia. *Algorithm design: foundation, analysis and internet examples*. John Wiley & Sons, 2006.