Universitatea Politehnică București
Facultatea de Automatică și Calculatoare
Analiza Algoritmilor

January 16, 2018

Soluțiile problemelor de la Testul 2
Analiză Amortizată
Verificarea corectitudinii algoritmilor

---

1. (30 points) Fie următoarea problemă:

   Se dă un șir A cu N numere întregi. Pentru fiecare subsecvență de lungime K să se determine minimul, iar apoi să se calculeze suma acestor minime.

   Pentru a rezolva problema vom folosi o structură de date numită *deque* ("double-ended queue"): o structură de date care conține o secvența de elemente și permite realizarea operațiilor menționate mai jos, în timp (amortizat) constant.

D.size()	Întoarce numarul de elemente din deque
D.front()	Întoarce valoarea de pe prima poziție
D.pushFront(e)	Adaugă elementul **e** la începutul secvenței
D.popFront()	Elimină valoarea de pe prima poziție
D.back()	Întoarce valoarea de pe ultima poziție
D.pushBack(e)	Adaugă elementul **e** la sfârșitul secvenței
D.popBack()	Elimină valoarea de pe ultima poziție

   Pentru soluția prezentată mai jos:

   (a) (20 points) Care este costul amortizat al operației **OrderedDequeAddElement**? Justificați.

   (b) (10 points) Ce complexitate are funcția **Solve**? Justificați.

```

OrderedDequeAddElement(D, A, K, pos) {

    // Cat timp elementul curent este mai mic
    // decat ultimul element din deque, eliminam pozitia
    // ultimului element din deque
    while (D.size() > 0 && A[pos] <= A[D.back()]) {
        D.popBack();
    }

    // Adaugam pozitia elementului curent in deque
    D.pushBack(pos);

    // Eliminam primul element din deque daca nu va mai
    // reprezenta o pozitie valida la pasul urmator
    if (D.front() == pos - K) {
        D.popFront();
    }
}

Solve(A, N, K) {

    Deque D;
    sum = 0;

    for (i = 1; i <= N; i++) {
        OrderedDequeAddElement(D, A, K, i);

        // Salvam minimul secventei curente,
        // acesta aflandu-se pe prima pozitie din deque
        if (i >= K) sum += A[D.front()];
    }

    return sum;
}
```

Listing 1: ”Soluție Problema 1”

Soluție

Putem aborda aceasta problemă într-o manieră similară fie prin metoda creditelor, fie prin metoda potențialului. Cu prima metodă accentul cade pe a asocia un credit unei componente particulare dintr-o structură de date (în acest caz, unui element din deque) pentru a putea ”plăti” pentru o operație viitoare; Metoda potențialului în schimb exploatează proprietăți generale ale structurii de date (de exemplu, în acest caz, am putea lua în considerare numărul de elemente din deque).

Câteva observații utile legate de algoritm:

- O poziție este adăugată **o dată**, respectiv eliminată din deque **cel mult o singura dată**.
- $Deque[i] \leq Deque[j], \forall i < j < K$
- Numărul de elemente din deque este, în orice moment de timp, **cel mult K**. Dacă estimăm pesimist că vom inspecta toate elementele din deque la fiecare iterație din for-ul principal, obținem o primă complexitate pentru întregul algoritm de $\mathcal{O}(\mathcal{N} * \mathcal{K})$ (care corespunde complexitații unei soluții naive pentru problema noastră)

Vom încerca să demonstrăm că putem sa asociem un cost amortizat constant pentru o operație de tipul *OrderedDequeAddElement*.

Observăm că avem o singură instrucțiune repetitivă (liniile 7-9), restul instrucțiunilor având asociate un cost amortizat constant, conform enunțului.

Vom nota cu T numărul de iterații din while, pentru un apel oarecare al funcției OrderedDequeAddElement și cu S numărul de elemente din deque în momentul apelului.

În funcție de configurația elementelor din deque, costul unei operații poate să aibă mai multe expresii diferite:

$$c = \begin{cases} (T+1) * \hat{c}_{size()} + (T+1) * \hat{c}_{back()} + T * \hat{c}_{popBack()} + 1 * \hat{c}_{pushBack()} + 1 * \hat{c}_{front()} \\ (T+1) * \hat{c}_{size()} + (T+1) * \hat{c}_{back()} + T * \hat{c}_{popBack()} + 1 * \hat{c}_{pushBack()} + 1 * \hat{c}_{front()} + 1 * \hat{c}_{popFront()} \\ (S+1) * \hat{c}_{size()} + S * \hat{c}_{back()} + S * \hat{c}_{popBack()} + 1 * \hat{c}_{pushBack()} + 1 * \hat{c}_{front()} \end{cases}$$

Primele doua situații au loc când $T \leq S$.

Este important să observăm că nu ne interesează cum este implementata efectiv structura "Deque", respectiv care sunt costurile reale ale operațiilor (este suficient să știm că putem să alocăm costuri amortizate constante).

Pentru a ne simplifica calculele, putem să înlocuim costurile din formulă cu o limită superioară:

$\hat{c}_{op} = max(\hat{c}_{size()}, \hat{c}_{back()}, \hat{c}_{popBack()}, \hat{c}_{pushBack()}, \hat{c}_{front()}, \hat{c}_{popFront()})$

Expresia costului devine:

$$c = \begin{cases} (3 * T + 4) * \hat{c}_{op} & \text{Ștergem T elemente, inserăm unul} \\ (3 * T + 5) * \hat{c}_{op} & \text{Ștergem T + 1 elemente, inserăm unul} \\ (3 * S + 3) * \hat{c}_{op} & \text{Ștergem S elemente, inserăm unul} \end{cases}$$

Vrem să găsim un cost amortizat pentru OrderedDequeAddElement convenabil care să respecte proprietatea:

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

Vom aborda aceasta problema folosind **metoda creditelor**.

Am menționat anterior că putem asocia un credit fiecărui element din deque. Practic, vom 'acumula' credit în timpul operației push_back() și vom 'consuma' credit când efectuăm toate celelalte operații.

Vom nota cu $credit_i$ - creditul acumulat la un moment de timp i. Condiția care trebuie îndeplinită este să avem permanent credit pozitiv în 'bancă':

$credit_{i+1} = credit_i + \hat{c}_i - c_i \geq 0$

$credit_i$ este direct proporțional cu numărul de operații de tip push_back() efectuate, respectiv cu numărul de elemente din deque (S). ($credit_i = k * S$)

Prin urmare, pentru cazul general, când inserăm un element și stergem T elemente, vrem să acumulăm cel puțin $k * (S - T + 1)$ credite:

$credit_i + \hat{c}_i - c_i = k * S + \hat{c}_i - (3 * T + 4) * \hat{c}_{op} \geq k * (S - T + 1)$

$\hat{c}_i \geq k * (S - T + 1) - k * S + (3 * T + 4) * \hat{c}_{op}$

Pentru a avea o valoare convenabila pentru $\hat{c}_i$ (o constantă), propunem $k = 3 * \hat{c}_{op}$.

$\hat{c}_i \geq 3 * \hat{c}_{op} + 4 * \hat{c}_{op} \geq 7 * \hat{c}_{op}$

Prin urmare, putem alege $\hat{c}_i = 7 * \hat{c}_{op}$

Putem verifica că acumulăm credit corect și în celelalte situații:

$c = (3 * S + 3) * \hat{c}_{op} \implies credit_{i+1} = 3 * \hat{c}_{op} * S + 7 * \hat{c}_{op} - (3 * S + 3) * \hat{c}_{op} = 4 * \hat{c}_{op} \geq 3 * \hat{c}_{op}$

$c = (3 * T + 4) * \hat{c}_{op} \implies credit_{i+1} = 3 * \hat{c}_{op} * S + 7 * \hat{c}_{op} - (3 * T + 4) * \hat{c}_{op} = 3 * (S - T + 1) * \hat{c}_{op}$

Deoarece avem un cost amortizat constant pentru operația OrderedDequeAddElement(), funcția Solve() are complexitatea $\mathcal{O}(\mathcal{N})$.

Bineînteles, aceasta este doar una din soluțiile posibile. De exemplu, o alternativă ar fi fost să căutăm o funcție de potențial direct proportională cu numărul de elemente din deque.

2. (10 points) În urma analizei prin metoda potențialului, costul amortizat al oricărei operații realizate de un algoritm, aplicat pe o structură de date D, rezultă egal cu costul real al operației. Ce se poate spune despre funcția de potențial folosită?

Soluție

Conform definiției:

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

Conform enunțului:

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

Prin urmare, $\phi(D_i)$ este o constantă.

3. (30 points) Fie o secvență A de N numere întregi: $(a_1, a_2, ...a_N)$, N > 0. O subsecvență a șirului este de forma: $(a_i, a_{i+1}, ..a_j)$, cu 1 ≤ i ≤ j ≤ N. Suma subsecvenței este $a_i + a_{i+1} + .. + a_j$. Se cere să se determine subsecvența de sumă maximă.

   Fie următoarea solutie pentru problema mentionată mai sus:

```
1    maxSubArraySum(A, N) {
2        maxEndingHere = maxSoFar = A[1]
3        for (i = 2; i <= N; ++i) {
4            maxEndingHere = max(A[i], maxEndingHere + A[i])
5            maxSoFar      = max(maxSoFar, maxEndingHere)
6        }
7        return maxSoFar;
8    }
```

Listing 2: ”Subsecvență de sumă maximă”

   Demonstrați corectitudinea algoritmului specificat, folosind unul sau mai mulți invarianți la ciclare.

Înainte de iterația i:

$maxEndingHere$ = suma maximă a unei subsecvențe care se termina pe poziția i - 1.

$maxSoFar$ = suma maximă a oricărei subsecvențe $A[k_1..k_2],\ \forall k_1, k_2,\ 1 \leq k_1 \leq k_2 \leq i - 1$.

Inițializare

$maxEndingHere = maxSoFar = A[1]$ - se respectă proprietățile menționate.

Menținere

Linia 4: $maxEndingHere_i = max(A[i], maxEndingHere_{i-1} + A[i])$

Vrem sa calculăm suma maximă a unei subsecvențe care se termină pe poziția i. Avem practic doua cazuri:

- $maxEndingHere_{i-1} < 0$, prin urmare putem să ignorăm toate elementele anterioare și să includem doar elementul de pe poziția i. (altfel am obține o valoare mai mică)
- $maxEndingHere_{i-1} \geq 0$, prin urmare ne avantajează să extindem secvența de la pasul anterior cu elementul de pe poziția i.

Linia 5: $maxSoFar_i = max(maxSoFar_i, maxEndingHere_i)$

Subsecvența de sumă maximă pentru toate elementele din intervalul [1..i] este fie o subsecvență care se termină pe o poziție $k, k < i$, fie chiar subsecvența de sumă maximă care se termină pe poziția i.

Terminare

La ieșirea din for, $maxSoFar = maxSoFar_N$ - subsecvența maximă a oricărei subsecvențe $A[k_1..k_2],\ \forall k_1, k_2,\ 1 \leq k_1 \leq k_2 \leq N$.

*Am fost surprins să primesc întrebarea ”de ce este ++i și nu i++” în for. Intenția nu era să vă incurce, pur și simplu, în acest context, dacă reprezintă al treilea parametru al instrucțiunii for, cele doua expresii sunt perfect echivalente logic. (pentru tipuri primitive, un compilator modern va genera exact același cod in ambele situații)

---

4. (10 points) Dați exemplu de un invariant la ciclare pentru următorul algoritm:

```
log(x) {
    i = 0
    j = x
    while (j > 1) {
        j = j / 2
        i = i + 1
    }
    return i;
}
```

Listing 3: "Logaritm"

Considerăm ca x este un număr întreg, x >= 1.

Soluție

Era suficient să menționați un invariant corect, fără justificare.
(ex.: $x = j * 2^i$)

Invarianți la ciclare: proprietăți valabile la intrarea într-o buclă și conservate pe durata și la ieșirea din buclă.

---

5. (30 points) Se consideră tipul de date LIST, o listă generică de elemente de tipul T pentru care avem operatorii de egalitate, inegalitate și adunare. Fie următorii constructori de bază:

[] : $\rightarrow LIST$
cons(elem, list) : T × LIST → LIST

Se consideră operatorii:

1. **append** : $LIST \times LIST \rightarrow LIST$
   (A1) append([], B) = B
   (A2) append(cons(x, A), B) = cons(x, append(A, B))
2. **reverse** : $LIST \rightarrow LIST$
   (R1) reverse([]) = []
   (R2) reverse(cons(x, L)) = append(reverse(L), cons(x, []))
3. **filter** : $(T \rightarrow Bool) \times LIST \rightarrow LIST$
   (F1) filter(p, []) = []
   (F2)

$$filter(p, cons(x, L)) = \begin{cases} cons(x, filter(p, L)) & \text{if } p(x) \\ filter(p, L) & \text{altfel} \end{cases}$$

În plus, se presupune cunoscută proprietatea:

(P1) **reverse(append(A, B)) = append(reverse(B), reverse(A))**

Demonstrați următoarea proprietate prin inducție structurală:

(P2) **filter(p, L) = reverse(filter(p, reverse(L)))**

Soluție

O posibilă rezolvare prin inducție structurală după L.

Cazul de bază: L = []

Vrem să arătăm: **filter(p, []) = reverse(filter(p, reverse([]))**

filter(p, []) = [] (din F1)

filter(p, reverse([])) = filter(p, []) (din R1) = [] (din F1) (proprietatea este adevărată)

Pasul de inducție: L = cons(x, A)

Ipoteza inductivă: filter(p, A) = reverse(filter(p, reverse(A)))

Vrem să arătăm: **filter(p, cons(x, A)) = reverse(filter(p, reverse(cons(x, A))))**

Folosim o proprietate adițională:

(P3) **filter(p, append(A, B)) = append(filter(p, A), filter(p, B))**

Obs. A = append([], A) (din A1)

LHS (left-hand-side) = filter(p, cons(x, A)) = filter(p, append(cons(x, []), A)) (din A1, A2) = append(filter(p, cons(x, [])), filter(p, A)) (din P3)

$$Obs.filter(p, cons(x, [])) = \begin{cases} cons(x, filter(p, []))(dinF2) = cons(x, []) \text{ (din F1)} & \text{if p(x)} \\ filter(p, []) = [] \text{ (din F1)} & \text{altfel} \end{cases}$$

$$LHS = \begin{cases} append(cons(x, []), filter(p, A)) = LHS_1 & \text{if p(x)} \\ filter(p, A) \text{ (din A2)} = LHS_2 & \text{altfel} \end{cases}$$

RHS (right-hand-side) = reverse(filter(p, reverse(cons(x, A))))

Obs. reverse(cons(x, A)) = append(reverse(A), cons(x, [])) (din R2)

Obs. filter(p, append(reverse(A), cons(x, []))) = append(filter(p, reverse(A)),filter(p, cons(x, []))) (din P3)

RHS = append(reverse(filter(p, cons(x, []))), reverse(filter(p, reverse(A)))) (din P1)

Obs. reverse(cons(x, [])) = append(reverse([]), cons(x, [])) (din R2) = append([], cons(x, [])) (din R1) = cons(x, []) (din A1)

$$RHS = \begin{cases} append(cons(x, []), filter(p, A)) \text{ (din F1, II)} = RHS_1 & \text{if p(x)} \\ append(filter(p, A) \text{ (din F1, A2)} = LHS_2 & \text{altfel} \end{cases}$$

LHS = RHS (q.e.d)

Demonstrăm și proprietatea adițională P3:

(P3) filter(p, append(A, B)) = append(filter(p, A), filter(p, B))

Cazul de bază: L = []

Vrem să arătăm: **filter(p, append([], B)) = append(filter(p, []), filter(p, B))**

LHS = filter(p, append([], B)) = filter(p, B) (din A1) RHS = append(filter(p, []), filter(p, B)) = append([], filter(p, B)) (din F1) = filter(p, B) (din A1)

LHS = RHS

Pasul de inducție: L = cons(x, A)

Ipoteza inductivă: filter(p, append(A, B)) = append(filter(p, A), filter(p, B))

Vrem să arătăm: **filter(p, append(cons(x, A), B)) = append(filter(p, cons(x, A)), filter(p, B))**

LHS = filter(p, append(cons(x, A), B)) = filter(p, cons(x, append(A, B))) (din A2)

$$LHS = \begin{cases} cons(x, filter(p, append(A, B))) = LHS_1 & \text{if p(x)} \\ filter(p, append(A, B)) = LHS_2 & \text{altfel} \end{cases}$$

RHS = append(filter(p, cons(x, A)), filter(p, B))

$$RHS = \begin{cases} append(cons(x, filter(p, A)), filter(p, B)) = RHS_1 & \text{if p(x)} \\ append(filter(p, A), filter(p, B)) = RHS_2 & \text{altfel} \end{cases}$$

RHS$_1$ = $cons(x, append(filter(p, A), filter(p, B)))(dinA2) = LHS_1$ (din II)

LHS$_2$ = RHS_2 (din II)

---

6. Fie tipul de date T definit prin:
   1. $(11, 21) \in T$
   2. $(x, y) \in T \to (x + 2, y) \in T$
   3. $(x, y) \in T \to (-x, y) \in T$
   4. $(x, y) \in T \to (y, x) \in T$

   (a) (5 points) Câți constructori de bază nulari, externi, respectiv interni are tipul T?

   (b) (5 points) Ce valori poate lua un element care aparține tipului T?

Soluție

(a) T are 1 constructor nular (care din 0 parametri produce perechea (11, 21)) și 3 interni (cei specificați pe liniile 2, 3, 4).

(b) T conține toate perechile (x, y) cu $x, y \in Z$ și $|x|$, $|y|$ impare.