Algoritmi
e
Strutture Dati

v0.3.0

Luca De Franceschi Università degli studi di Padova

Diario delle modifiche

Autore	**Versione**	**Data**	**Descrizione**
Luca De Franceschi	0.5.0	13/06/2014	Inserito capitolo analisi ammortizzata con due esercizi, incrementata sezione metodo dell'integrale
Luca De Franceschi	0.4.0	11/06/2014	Inserito capitolo analisi complessità con metodo dell'integrale e metodo dell'esperto
Luca De Franceschi	0.3.0	11/06/2014	Inserita spiegazione metodo di sostituzione
Luca De Franceschi	0.2.0	11/06/2014	Inserita teoria su programmazione dinamica
Luca De Franceschi	0.1.0	11/06/2014	Creata struttura del documento

Indice

1 Stima della complessità di un algoritmo

Identifichiamo dei casi base, studiando la complessità degli algoritmi noti.

1. Le operazioni elementari, messe al di fuori dei cicli, e che riguardano l'uso di variabili hanno complessità costante c_i, aprossimabile a 0 nello studio della complessità asintotica;
2. Da *insertion-sort* si vede che un *for i = 2 to n* ha complessità pari a n. Constatiamo dunque che un ciclo for che va dall'indice 1 all'indice n avrà complessità $c_i(n+1)$;
3. Tutte le operazioni elementari che compaiono all'interno di un ciclo for di complessità $c_i(n+1)$ hanno complessità $c_i n$;
4. Per i cicli annidati in altri cicli a complessità è data da $c_i \sum_{j=x}^{n}(c_j)$, dove gli estremi della sommatoria sono gli estremi del del ciclo esterno.

Per valutare la complessità si scrive l'equazione $T(n)$ sommando tutte le complessità. Per studiare le sommatorie si utilizza il **metodo dell'integrale**.

1.1 Metodo dell'integrale

Se $f(x)$ è una funzione **non decrescente**:

$$\int_a^{b+1} f(x)\mathrm{d}x \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b+1} f(x)\mathrm{d}x$$

Se $f(x)$ è una funzione **non decrescente**:

$$\int_a^{b+1} f(x)\mathrm{d}x \leq \sum_{i=a}^{b} f(i) \leq \int_{a-1}^{b} f(x)\mathrm{d}x$$

1.1.1 Integrali immediati

- $\int [f(x)]^\alpha f'(x)\mathrm{d}x = \frac{[f(x)]^{\alpha+1}}{(\alpha+1)} + C \qquad \alpha \neq 1$
- $\int \frac{f'(x)}{f(x)}\mathrm{d}x = \log|f(x)| + C$
- $\int f'(x)e^{f(x)}\mathrm{d}x = e^{f(x)} + C$
- $\int f'(x)a^{f(x)}\mathrm{d}x = a^{f(x)}\log_a e + C$

1.1.2 Integrazione per parti

Siano f, g, g' continue nell'intervallo $[a, b]$ e sia γ una primitiva di f, allora:

$$\int f(x)g(x)\mathrm{d}x = \gamma(x)g(x) - \int \gamma(x) + g'(x)\mathrm{d}x$$

1.1.3 Serie aritmetica e geometrica

- Serie aritmetica: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$;
- Serie geometrica: $\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}$ $q \neq 1$

1.2 Andamento asintotico

Una volta ottenuta una funzione che rappresenta la complessità dell'algoritmo ci può interessare prendere in esame l'andamento asintotico della medesima. Per farlo introduciamo le seguenti notazioni:

- **"O" grande**: date due funzioni $f(n)$ e $g(n)$ si dice che $f(n)$ è "O" grande di $g(n)$ se esiste un $c > 0$ e un h_0 tali che:

  $$f(n) \leq cg(n)$$

  per $n \geq h_0$ (**limite asintotico superiore**). In pratica l'ordine di crescita di $f(n)$ è non superiore a quello di $g(n)$;

- **"Ω" grande**: date $f(n)$ e $g(n)$ si dice che $f(n)$ è "Ω" grande di $g(n)$ se esiste una costante $c > 0$ e un h_0 tale che:

  $$f(n) \geq cg(n)$$

  per $n \geq h_0$ (**limite asintotico inferiore**). In pratica l'ordine di crescita di $f(n)$ è non inferiore a quello di $g(n)$;

- **"Θ" grande**: date $f(n)$ e $g(n)$ si dice che $f(n)$ è "Θ" grande di $g(n)$ se ci sono costanti positive c_1, c_2 e un h_0 tali che:

  $$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

  per $n \geq h_0$ (**limite asintotico stretto**). In pratica diciamo che se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$ allora è vero anche che $f(n) = \Theta(g(n))$

Di una funzione non ci interessa la sua forma ma il suo comportamento asintotico. Spesso è possibile determinare dei limiti asintotici calcolando il limite di un rapporto:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

In base al risultato di questo limite ho tre casi:

1. Ottengo un valore costante $k > 0$: in questo caso $f(n)$ è dello stesso ordine di $g(n)$, e dunque:

   $$\forall \epsilon > 0, \exists h_0 | h \geq h_0 : k - \epsilon \leq f(n)/g(n) \leq k + \epsilon$$

   ponendo:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Dunque concludo dicendo che $f(n) = \Theta(g(n))$;

2. Il limite tende a ∞: $f(n) = \Omega(g(n))$;
3. Il limite tende a 0: $f(n) = O(g(n))$.

1.3 Metodo dell'esperto

Per risolvere le ricorrenze il primo metodo da utilizzare è il **metodo dell'esperto**. Se la ricorrenza è espressa nella forma:

$$T(n) = aT(n/b) + f(n)$$

e se $a \geq 1$ e $b < 1$ allora:

1. Tolgo eventuali arrotondamenti;
2. Calcolo $\log_b a$ e calcolo il limite: $\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}}$;

A questo punto, in base al valore del limite ho 3 possibili casi:

1.3.1 Caso 2

Se il limite è **finito** e diverso da zero:

$$f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \log n)$$

1.3.2 Caso 1

Se il limite è **uguale a zero** devo trovare un valore $\epsilon > 0$ per il quale risulta finito il limite:

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a - \epsilon}} = k$$

Se lo trovo allora posso affermare che:

$$f(n) = O(n^{\log_b a - \epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$$

1.3.3 Caso 3

Se il limite è $\int$ allora devo trovare un $\epsilon > 0$ per il quale risulti:

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a + \epsilon}} \neq 0$$

Se lo trovo allora devo studiare l'equazione:

$$af(n/b) \leq k(f(n))$$

se trovo un $k < 1$ allora posso concludere che:

$$f(n) = \Omega(n^{\log_b a + \epsilon}) \Rightarrow T(n) = \Theta(f(n))$$

1.4 Metodo di sostituzione

Se non riesco ad applicare il metodo dell'esperto allora devo utilizzare il **metodo di sostituzione**.
Per capire il metodo di sostituzione proviamo a risolvere il seguente esercizio:

La ricorrenza $T(n) = 4T(n/2) + n^2 \log n$ si può risolvere con il metodo dell'esperto? Giustificare la risposta. Se la risposta è negativa usare il metodo di sostituzione per dimostrare che $T(n) = O(n^2 \log^2 n)$.

Anzitutto vediamo i dati a disposizione:

$$a = 4, b = 2$$
$$f(n) = n^2 \log n$$
$$g(n) = n^{\log_b a} = n^{\log_2 4} = n^2$$

Calcoliamo ora il limite:

$$\lim_{n \to +\infty} \frac{n^2 \log n}{n^2} = \infty$$

Da cui deduco che:

$$f(n) = \Omega(n^2)$$

Potrei dunque essere nel caso 3. Devo trovare un

$$\epsilon > 0$$

tale che:

$$\lim_{n \to +\infty} \frac{n^2 \log n}{n^{2+\epsilon}} \neq 0$$

Ma mi accorgo subito che la cosa è impossibile, in quanto il denominatore, incrementando l'esponente, crescerà molto più velocemente rispetto al numeratore, per cui avrò sempre un valore tendente allo zero. Da questa considerazione deduco che la ricorrenza **non è risolvibile con il metodo dell'esperto**.

Procedo dunque con la sostituzione. Proviamo $T(n) = O(n^2 \log^2 n)$.
Assumiamo che per un'opportuna costante $C > 1$ e $\forall x < n$ sia verificata la disuguaglianza $T(x) \leq C(x^2 \log^2 x)$ e dimostriamo che vale anche per n:

$$\begin{aligned} T(n) = 4T(n/2) + n^2 \log n &\leq 4C(n/2)^2 \log^2(n/2) + n^2 \log n \\ &= Cn^2(\log n - 1)^2 + n^2 \log n \\ &= Cn^2(\log^2 n - 2 \log n + 1) + n^2 \log n \\ &= Cn^2 \log^2 n - 2Cn^2 \log n + Cn^2 \log n + Cn^2 + n^2 \log n \\ &= Cn^2 \log^2 n - (C-1)n^2 \log n - Cn^2(\log n - 1) \end{aligned}$$

Ora applico una **maggiorazione**:

$$\leq Cn^2 \log^2 n$$

Dunque ho dimostrato che: $T(n) = O(n^2 \log^2 n)$

2 Programmazione dinamica

In maniera del tutto generale la programmazione dinamica può essere descritta nel seguente modo:

1. Identifichiamo dei **sottoproblemi** del problema originario e utilizziamo una *tabella* per memorizzare i risultati intermedi;
2. Inizialmente vanno definiti i **valori iniziali** di alcuni elementi della tabella, corrispondenti a sottoproblemi più semplici;
3. Al generico passo, avanziamo in modo opportuno sulla tabella calcolando il valore della soluzione di un sottoproblema in base alla soluzione di sottoproblemi precedentemente risolti (corrispondenti ad elementi della tabella precedentemente calcolati);
4. Alla fine restituiamo la soluzione del problema originario, che è stato memorizzato in un particolare elemento della tabella.

La programmazione dinamica è usata normalmente per **problemi di ottimizzazione**, il termine "programmazione" si riferisce al metodo tabulare, non alla scrittura di codice.

La programmazione è applicabile con vantaggi se:

- Gode della proprietà di **sottostruttura ottima**: una soluzione si può costruire a partire da soluzioni ottime di sottoproblemi;
- Il numero di sottoproblemi distinti è molto minore del numero di soluzioni possibili tra cui cercare quella ottima, altrimenti c'è la **ripetizione di sottoproblemi**, ovvero se il numero di sottoproblemi distinti è molto minore del numero di soluzioni possibili tra cui cercare quella ottima, allora uno stesso sottoproblema deve comparire molte volte come sottoproblema di altri sottoproblemi.

2.1 Ordine di calcolo delle soluzioni dei sottoproblemi

Bottom-up: le soluzioni dei sottoproblemi del problema in esame sono già state calcolate. È il metodo migliore se per il calcolo della soluzione globale servono le soluzioni di tutti i sottoproblemi.

Top-down: è una procedura ricorsiva che dall'alto scende verso il basso. È la soluzione migliore se per il calcolo della soluzione globale servono soltanto alcune delle soluzioni dei sottoproblemi.

3 Algoritmi golosi

Si parla sempre di **problemi di ottimizzazione**.

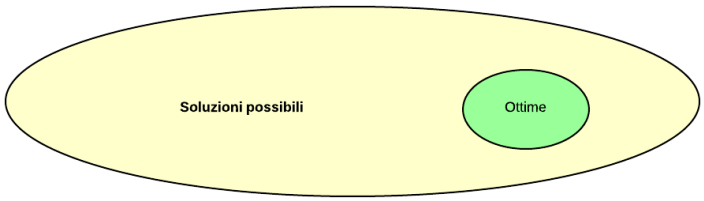


Figura 1: Soluzioni di un problema

Se utilizziamo l'**enumerazione esaustiva**:

- Si generano tutte le soluzioni possibili;
- Si calcola il costo di ciascuna di esse;
- Se ne seleziona una di ottima.

Questo metodo è chiaramente efficace ma comporta **tempi esponenziali**, perchè l'insieme delle soluzioni è generalmente enorme.
Un **algoritmo goloso** sceglie sempre una soluzione **localmente ottima**:

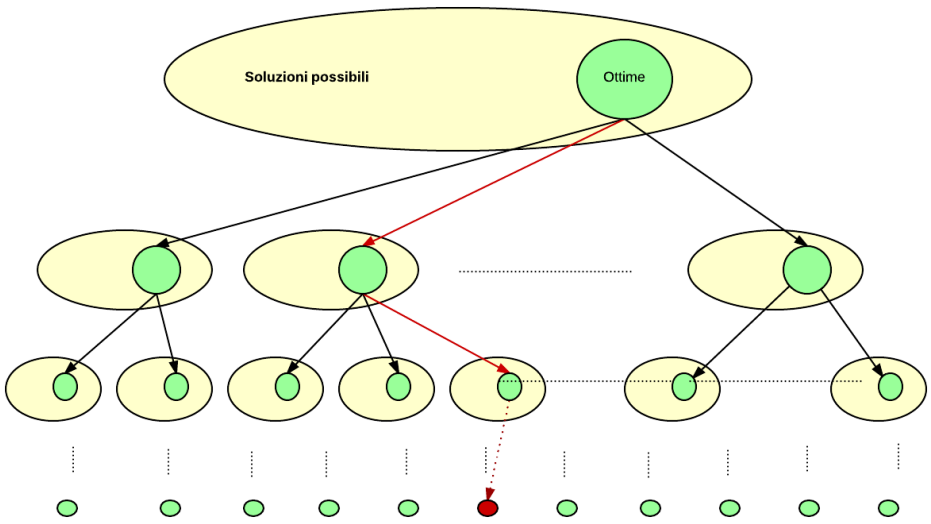


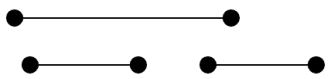
Figura 2: Schema di un algoritmo goloso

1. Ogni volta si fa una scelta che sembra migliore localmente;
2. In questo modo per alcuni problemi si ottiene una soluzione globalmente ottima.

3.1 Problema della scelta di attività

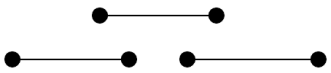
Un esempio tipico è quello del **problema della scelta di attività**.
Supponiamo di avere a disposizione n attività $a_1, a_2, ..., a_n$ che utilizzano la stessa risorsa (ad esempio un'aula). Ciascuna attività ha un tempo di inizio s_i e un tempo di fine f_i, con $0 \leq s_i < f_i < \infty$.
a_i occupa la risorsa nell'intervallo $[s_i, f_i)$.
a_i e a_j sono **compatibili** se gli intervalli $[s_i, f_i)$ e $(s_j, f_j]$ sono disgiunti.
Strategie golose:

- Scegliere l'attivitò che inizia per prima:



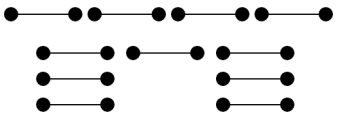
  Non funziona!

- Scegliere l'attività che dura meno tempo:



  Non funziona!

- Scegliere l'attività incompatibile con il minor numero di altre attività:



  Non funziona!

La strategia che funziona è quella di scegliere l'attività che **termina per prima**.

Listato 1: Algoritmo di selezione delle attività

```
ActivitySelector(a,s,f,n) // f1 ≤ f2 ≤ ... ≤ fn
        A = {a1}, k=1
        for m=2 to n
                if s[m] >= f[k]
                        A = A "unito" {am}, k = m
        return A
```

Sappiamo che durante tutta l'esecuzione dell'algoritmo esiste sempre una soluzione ottima contenente le attività scelte fino a quel momento. L'algoritmo

è goloso, perchè ad ogni passo sceglie l'attività che termina prima. Questa scelta è **localmente ottima**, perchè è quella che lascia più tempo a disposizione per le attività successive.

3.2 Problema dello zaino frazionario

Dati n tipi di merce, $M_1, M_2, ..., M_n$ in quantità $q_1, q_2, ..., q_n$ e con costi unitari $c_1, c_2, ..., c_n$ si vuole riempire uno zaino di capacità Q in modo che il contenuto abbia costo massimo.

Listato 2: Algoritmo goloso dello zaino frazionario

```
RiempiZaino(q, c, n, Q) // c_1 ≥ c_2 ≥ c_3 ≥ ... ≥ c_n
       Spazio = Q
       for i=1 to n
              if Spazio >= q[i]
                     Z[i] = q[i], Spazio = Spazio - Z[i]
              else
                     Z[i] = Spazio, Spazio = 0
       return Z
```

3.3 Problema del pulmino con numero fissato di posti

Un'azienda di una grande città ha diverse agenzie. Su richiesta sindacale l'azienda ha istituito un servizio sperimentale di trasporto per i dipendenti che utilizza un solo pulmino con 10 posti. Al mattino il pulmino effettua un percorso prefissato che passa per tutte le agenzie raccogliendo gli n dipendenti che hanno fatto richiesta nei punti del percorso per loro più comodi per portarli alle rispettive agenzie. Per ogni dipendente è noto il punto di partenza $s[i]$ e il punto di arrivo $f[i]$. Essendo il servizio sperimentale non ci si aspetta che tutte le richieste possano essere soddisfatte.
Descrivere un algoritmo goloso che determina un'assegnazione dei posti del pulmino che permette di trasportare il massimo numero di dipendenti.
Naturalmente lo stesso posto può essere utilizzato da più dipendenti, purchè il loro tragitti non si sovrappongano.
Soluzione: Ordiniamo i dipendenti per i punti di arrivo, dal più vicino al più lontano. La scelta golosa consiste poi nell'assegnare a ogni dipendente il posto che si è liberato per ultimo, in modo da massimizzare il numero di dipendenti.

Listato 3: Algoritmo goloso del pulmino

```
Pulmino(s, f, n, m) // PRE: f_1 ≤ f_2 ≤ ... ≤ f_n
       for j=1 to m
              t_j = 0
```

```
t_0 = -1
for i=1 to n
        h = 0
        for j=1 to m
                if s_i ≥ t_j and t_j ≥ t_h then h = j
                // A_h Ã¨ il posto che si libera per ultimo tra
                     quelli in cui Ã¨ possibile far sedere una
                     persona.
                // Se h == 0 in nessuno degli m posti si puo' far
                     sedere una persona
        J[i] = h
        if h ≠ 0 then t_h = f_i
// POST: J[1...n] Ã¨ una programmazione ottima del massimo
    numero di persone che si possono trasportare con m posti.
// J[i] == j ≠ 0 significa che la persona non puo' sedersi in
    nessuno degli m posti
return J
```

4 Analisi ammortizzata

Si consideri il tempo richiesto per eseguire, nel **caso pessimo**, un'intera sequenza di operazioni. Se le operazioni costose sono relativamente meno frequenti allora il costo richiesto per eseguirle può essere ammortizzato con l'esecuzione delle operazioni meno costose.

4.1 Metodo dell'aggregazione

Si basa sul concetto di **costo ammortizzato**: data una sequenza di n istruzioni aventi complessità $O(f(n))$, il costo ammortizzato della singola operazione si ottiene dividendo la complessità totale per il numero di istruzioni.

$$\hat{c} = \frac{O(f(n))}{n}$$

4.2 Metodo degli accantonamenti (o dei crediti)

Si caricano le operazioni meno costose di un costo aggiuntivo. Il costo aggiuntivo viene assegnato come **credito prepagato** a certi oggetti nella struttura dati. I crediti accumulati saranno usati per pagare le operazioni più costose su tali oggetti.
Il costo ammortizzato delle operazioni meno costose è il costo effettivo aumentato del costo aggiuntivo.
Il costo ammortizzato delle operazioni più costose è il costo effettivo diminuito del credito prepagato utilizzato.
Il costo artificiale fornisce un **limite superiore** al costo ammortizzato.

4.3 Metodo del potenziale

Si associa alla struttura dati D un **potenziale** $\Phi(D)$ tale che la modifica della struttura dati dovuta alle operazioni meno costose comporti un aumento del potenziale, mentre le operazioni meno costose lo facciano diminuire.
Il costo ammortizzato è quindi la **somma algebrica** del costo effettivo e della variazione di potenziale. D_i è la struttura dati dopo la c-esima operazione e c_i + il costo dell'i-esima operazione.

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

Il costo ammortizzato di una sequenza di n operazioni è:

$$\begin{aligned}\hat{c} = \sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} [c_i + \Phi(D_i) - \Phi(D_{i-1})] \\ &= c + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

Se la variazione $\Phi(D_n) - \Phi(D_0)$ del potenziale relativo all'esecuzione di tutta la sequenza non è negativa allora il costo ammortizzato $\hat{c}$ è una **maggiorazione** del costo reale c.
Altrimenti un valore $\Phi(D_n) - \Phi(D_0)$ negativo deve essere compensato con un aumento adeguato del costo ammortizzato delle operazioni.

4.4 Esercizio 1

Si vuole realizzare un contatore binario usando un array $A[0...k]$ per memorizzare i $k+1$ bit $b_k...b_1b_0$ della rappresentazione binaria del valore x del contatore. Si vuole che il contatore possa iniziare anche con un valore maggiore di zero. A tale scopop si vuole che, oltre all'operazione **increment**, che aumenta di 1 il valore del contatore, sia prevista anche un'operazione iniziale **SET(A,n)**, che inizializza ad n il valore del contatore. Usare il **metodo di aggregazione** per dimostrare che le operazioni di una sequenza costituita da una set seguita da m increment, con $m = \Omega(k)$ hanno costo ammortizzato costante.

Soluzione:Vediamo anzitutto gli pseudo-codici delle due funzioni:

```
Increment(A)
        i = 0
        while i<=k and A[i]==1
                A[i] = 0
                i++
        if i<=k
                A[i] = 1
```

```
Set(A,n) // O(k)
        // PRE: A azzerato
        while i<=k and n>0
                A[i] = n%2
                n = n/2    // preso per difetto
                i++
```

Sia $\Phi(A) =$"numero di bit impostati a 1". Il costo di una increment è $c = 1 + t$, dove $t \geq 0$ è il numero di bit trasformati in 1 da 0. Sia A_0 lo stato iniziale del contatore azzerato ed A_1 il suo stato dopo aver eseguito una Set(A,n). Supponiamo di effettuare m esecuzioni di increment per valutare l'analisi ammortizzata:

$$\Delta\Phi = \Phi(A_m) - \Phi(A_1) = t - 1$$

Il numero di bit 1 rispetto ad A_1 varia di $-t + 1$ in A_m. Dunque aggiungiamo k alla formula per calcolare $\hat{c}$, distribuendolo a tutte le operazioni;

$$\hat{c} = c + \Phi(A_n) - \Phi(A_0) + \frac{k}{m} = 1 + t - t + 1 + \frac{k}{m}$$

Alla fine il costo ammortizzato è $O(1 + \frac{k}{m})$, $m = \Omega(k), m \geq k$.

$$\frac{O(k)+O(1+\frac{k}{m})}{m+1} = O(\frac{k+1+k/m}{m+1}) \leq O(\frac{k+1+k/k}{k+1}) = O(\frac{k+2}{k+1}) = O(1)$$

4.5 Esercizio 2

Si vuole realizzare un timer usando un array A per memorizzare i $k+1$ bit $b_k, ..., b_1, b_0$ della rappresentazione binaria del valore del timer. Le operazioni previste per un timer sono *Set(A,n)* che carica il timer ad n secondi e *Decrement(A)* che diminuisce di un secondo il valore del timer. L'operazione Set si può eseguire solamente quando il timer è azzerato mentre l'operazione Decrement si può eseguire soltanto quando il timer ha valore maggiore di 0. Scrivere le due funzioni Set e Decrement ed analizzarne la complessità ammortizzata.

Suggerimento: Memorizzare l'indice m del bit 1 più significativo ($m = -1$ se tutti i bit sono 0, $m = k$ se tutti i bit sono a 1) ed usare come funzione potenziale il numero di bit uguali a 0 che precedono $A[m]$ più due volte il numero di bit che seguono $A[m]$ (che sono tutti 0), ossia
$\Phi = \sum_{i=0}^{m-1}(1-b_i) + 2(k-m)$.

Soluzione: Vediamo anzitutto gli pseudo-codici delle due operazioni:

```
Set(A,n) // tutti i bit a 0, n<2^{k+1}
        k = A.lenght
        i = 0 // parto dal bit meno significativo
        while n>0 and i<=k
                A[i] = n%2 // il resto della divisione per 2
                n = n/2  // divisione intera per difetto
                i++
        if i>k
                error "underflow"
        else
                A.m = i-1
```

```
Decrement(A)
        i = 0
        while A[i] == 0
                A[i] = 1
                i++
        A[i] = 0 // ora devo mettere apposto l'n
        if A.m == i
                A.m = A.m-1
```

Analisi ammortizzata di Set:

$$\hat{c} = c + \Delta\Phi$$

$$\Delta\Phi = \Phi + \Phi' = 2(k-m) + \sum_{i=0}^{m-1}(1-b_i) - 2(k+1) =$$

$$= 2k - 2m + \sum_{i=0}^{m-1}(1-b_i) - 2k - 2$$

$$\hat{c} \leq m + 1 - 2m + m - 2 \leq -1$$

Il costo è costante quindi ho concluso la dimostrazione.

Analisi ammortizzata di Decrement

$c = c + 1$, proporzionale a $t - 1$, dove t è il numero di bit trasformati in 0.

$$\hat{c} = c + \Delta\Phi$$

$$\Delta\Phi = \Phi - \Phi' = 2(k-m) + \sum_{i=0}^{m-1}(1-b_i) - 2(k-m') + \sum_{i=0}^{m'-1}(1-b_i) =$$

$$-2m + 2m' - t + 1$$

m ed m' possono al massimo diversi tra loro di 1, quindi:

$$\Delta\Phi \leq 2 - t + 1 = 3 - t$$

$$\hat{c} = t + 1 + 3 - t = 4$$

Ottengo un valore costante, inoltre:

$$\Phi_0 = 2(k+1)$$

In questo caso perchè il costo ammortizzato vada bene deve essere maggiore o uguale del costo reale.
Prendiamo le operazioni di segno:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}$$

$$\sum_{i=1}^{n}\hat{c}_i = \sum_{i=1}^{n}\hat{c}_i + \Phi_n + \Phi_0 \leq \sum_{i=0} n(\hat{c}_i - \frac{\Phi_0}{n})$$

$$\hat{c}_i = c_i + \frac{\Phi_0}{n}$$

Devo distribuire, il costo ammortizzato è quindi:

$$\hat{c} = 4 + \frac{2(k+1)}{n}$$

Se $n = \Omega(k) \Rightarrow \hat{c} = O(1)$