
DOMANDE

Domanda A (7 punti)

Testo: Si dia la definizione di limite asintotico stretto. Data la seguente equazione di ricorrenza:

$T(n) = \{ 1$ se $n = 1$ $2T(n/5) + T(n/2) + n$ se $n > 1 \}$

mostrare che $f(n) = n$ è limite asintotico stretto per la soluzione.

Definizione di Limite Asintotico Stretto

Definizione (Notazione Θ):

Sia $f(n)$ una funzione definita sui naturali. Si dice che $f(n) = \Theta(g(n))$ se esistono tre costanti positive c_1, c_2, n_0 tali che:

$\forall n \geq n_0$: $c_1 g(n) \leq f(n) \leq c_2 g(n)$

Equivalentemente: $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$

Dire che $f(n)$ è **limite asintotico stretto** di $T(n)$ significa $T(n) = \Theta(f(n))$.

Dimostrazione che $T(n) = \Theta(n)$

Dobbiamo dimostrare che esistono costanti c_1, c_2, n_0 tali che: $c_1 n \leq T(n) \leq c_2 n$ per ogni $n \geq n_0$

Passo 1: Limite superiore $T(n) = O(n)$

Ipotesi: Esiste c_2 tale che $T(n) \leq c_2 n$ per ogni n sufficientemente grande.

Base: $T(1) = 1 \leq c_2 \cdot 1$ per ogni $c_2 \geq 1$. ✓

Passo induttivo: Assumiamo $T(k) \leq c_2 k$ per ogni $k < n$. Dimostriamo $T(n) \leq c_2 n$.

$T(n) = 2T(n/5) + T(n/2) + n \leq 2c_2(n/5) + c_2(n/2) + n$ [per ipotesi induttiva] $= (2c_2/5)n + (c_2/2)n + n = c_2 n(2/5 + 1/2) + n = c_2 n(4/10 + 5/10) + n = c_2 n(9/10) + n = n(9c_2/10 + 1)$

Vogliamo: $n(9c_2/10 + 1) \leq c_2 n$

Dividendo per n: $9c_2/10 + 1 \leq c_2$ $1 \leq c_2 - 9c_2/10$ $1 \leq c_2/10$ $c_2 \geq 10$

Conclusione: Per $c_2 \geq 10$, vale $T(n) \leq c_2 n$, quindi $T(n) = O(n)$. ✓

Passo 2: Limite inferiore $T(n) = \Omega(n)$

Ipotesi: Esiste c_1 tale che $T(n) \geq c_1 n$ per ogni n sufficientemente grande.

Base: $T(1) = 1 \geq c_1 \cdot 1$ per ogni $0 < c_1 \leq 1$. ✓

Passo induttivo: Assumiamo $T(k) \geq c_1 k$ per ogni $k < n$. Dimostriamo $T(n) \geq c_1 n$.

$T(n) = 2T(n/5) + T(n/2) + n \geq 2c_1(n/5) + c_1(n/2) + n$ [per ipotesi induttiva] $= (2c_1/5)n + (c_1/2)n + n = c_1 n(2/5 + 1/2) + n = c_1 n(9/10) + n = n(9c_1/10 + 1)$

Vogliamo: $n(9c_1/10 + 1) \geq c_1 n$

Dividendo per n: $9c_1/10 + 1 \geq c_1$ $1 \geq c_1 - 9c_1/10$ $1 \geq c_1/10$ $c_1 \leq 10$

Conclusione: Per $0 < c_1 \leq 10$, vale $T(n) \geq c_1 n$, quindi $T(n) = \Omega(n)$. ✓

Conclusione finale: Scegliendo $c_1 = 1$ e $c_2 = 10$, abbiamo dimostrato che: $n \leq T(n) \leq 10n$ per ogni $n \geq 1$

Quindi **$T(n) = \Theta(n)$**, cioè $f(n) = n$ è limite asintotico stretto per $T(n)$. ■

Domanda B (6 punti)

Testo: Indicare, in forma di albero binario, il codice prefisso ottenuto tramite l'algoritmo di Huffman per l'alfabeto {a, b, c, d, e, f}, supponendo che ogni simbolo appaia con le seguenti frequenze:

a	b	c	d	e	f
11	6	13	35	10	25

Spiegare brevemente il processo di costruzione del codice.

Algoritmo di Huffman

L'algoritmo di Huffman costruisce un codice prefisso ottimo (a lunghezza variabile) per la compressione di dati, minimizzando la lunghezza media del codice pesata sulle frequenze.

Procedimento:

1. Creare una coda di priorità (min-heap) con tutti i simboli, usando le frequenze come chiavi
2. Ripetere fino a un solo nodo:
   - Estrarre i due nodi con frequenza minima
   - Creare un nuovo nodo interno con frequenza = somma delle due frequenze
   - I due nodi estratti diventano figli del nuovo nodo
   - Inserire il nuovo nodo nella coda

3. L'ultimo nodo rimasto è la radice dell'albero di Huffman

Costruzione Passo-Passo

Frequenze iniziali:

- a: 11
- b: 6
- c: 13
- d: 35
- e: 10
- f: 25

Coda ordinata: [b:6, e:10, a:11, c:13, f:25, d:35]

Passo 1: Estrai b(6) ed e(10), crea nodo interno N_1(16)

- Coda: [a:11, c:13, N_1:16, f:25, d:35]

Passo 2: Estrai a(11) e c(13), crea nodo interno N_2(24)

- Coda: [N_1:16, N_2:24, f:25, d:35]

Passo 3: Estrai N_1(16) e N_2(24), crea nodo interno N_3(40)

- Coda: [f:25, d:35, N_3:40]

Passo 4: Estrai f(25) e d(35), crea nodo interno N_4(60)

- Coda: [N_3:40, N_4:60]

Passo 5: Estrai N_3(40) e N_4(60), crea radice R(100)

- Coda: [R:100]

Albero di Huffman Risultante

```
                R(100)
               /      \
              /        \
          N₃(40)      N₄(60)
         /    \       /    \
     N₁(16) N₂(24) f(25) d(35)
     /  \     /  \
  b(6) e(10) a(11) c(13)
```

Assegnazione dei Codici

Convenzione: arco sinistro = 0, arco destro = 1

Codici risultanti:

- b: 000
- e: 001
- a: 010
- c: 011
- f: 10
- d: 11

Verifica Codice Prefisso

Nessun codice è prefisso di un altro: ✓

- b: 000
- e: 001
- a: 010
- c: 011
- f: 10
- d: 11

Lunghezza media pesata: $L = (11\times3 + 6\times3 + 13\times3 + 35\times2 + 10\times3 + 25\times2) / 100 = (33 + 18 + 39 + 70 + 30 + 50) / 100 = 240 / 100 = 2.4$ bit/simbolo

---

ESERCIZI

Esercizio 1 (10 punti) - Split Proprietà

Testo: Un ricco possidente deve lasciare in eredità le sue 2n proprietà a n figli. Il valore delle proprietà è contenuto in un array di numeri (reali positivi) A[1..2n]. Sviluppare un algoritmo Split(A, n) che dato in input l'array A[1..2n] dei valori delle proprietà e numero n, verifica se le 2n proprietà possano partizionare in n coppie, tutte con lo stesso valore complessivo.

Analisi del Problema

Condizione necessaria: Ogni coppia deve avere valore S/n, dove $S = \sum A[i]$.

Strategia: Approccio greedy con hashing per trovare coppie complementari.

Algoritmo

```
SPLIT(A, n)
 size = 2 * n

```

```
3  // Calcola somma totale
4  S = 0
5  for i = 1 to size
6      S = S + A[i]
7
8  // Verifica divisibilità
9  if S mod n ≠ 0
10     return false
11
12 target = S / n
13
14 // Ordina array in ordine decrescente
15 MERGE-SORT-DESC(A, 1, size)
16
17 // Array per marcare elementi usati
18 used = new boolean[1..size] initialized to false
19
20 pairs_formed = 0
21
22 // Per ogni elemento non usato, cerca il complemento
23 for i = 1 to size
24     if not used[i]
25         complement = target - A[i]
26
27         // Cerca complement nei rimanenti
28         found = false
29         for j = i + 1 to size
30             if not used[j] and A[j] == complement
31                 used[i] = true
32                 used[j] = true
33                 pairs_formed = pairs_formed + 1
34                 found = true
35                 break
36
37         if not found
38             return false
39
40 return pairs_formed == n
```

Dimostrazione di Correttezza

Invariante: All'inizio di ogni iterazione del ciclo esterno (riga 23), tutti gli elementi marcati come used sono stati accoppiati in coppie valide con somma target.

Inizializzazione: used è tutto false, pairs_formed = 0. Invariante vera vacuamente. ✓

Manutenzione:

- Prendiamo elemento i non usato

- Cerchiamo complement = target - A[i]
- Se troviamo complement non usato: formiamo coppia valida (A[i] + complement = target)
- Se non troviamo: impossibile formare n coppie → return false
- Invariante preservata ✓

Terminazione:

- Se completiamo il ciclo: pairs_formed == n → tutte le coppie formate ✓
- Se ritorniamo false durante: impossibile completare la partizione ✓

Correttezza: L'algoritmo trova una partizione valida se esiste, altrimenti ritorna false. ■

Analisi Complessità

Operazioni principali:

1. Calcolo somma: $\Theta(n)$
2. Ordinamento: $\Theta(n \log n)$
3. Doppio ciclo di ricerca:
    - Ciclo esterno: $O(n)$ iterazioni
    - Ciclo interno: $O(n)$ nel caso peggiore
    - Totale: $O(n^2)$

Complessità totale: $O(n^2)$ dominata dal doppio ciclo

Ottimizzazione con Hash:

```
SPLIT-OPTIMIZED(A, n)
size = 2 * n
S = SUM(A)
if S mod n ≠ 0
    return false

target = S / n

// Crea multiset (hash map con contatori)
freq = CREATE-HASH-MAP()
for i = 1 to size
    freq[A[i]] = freq[A[i]] + 1

pairs = 0
for each value v in freq.keys()
    complement = target - v

    if complement == v
        // Serve coppia di elementi uguali
        pairs = pairs + freq[v] / 2
```

```
        if freq[v] mod 2 ≠ 0
            return false
    else if freq.contains(complement)
        // Accoppia v con complement
        min_pairs = min(freq[v], freq[complement])
        pairs = pairs + min_pairs
        freq[v] = freq[v] - min_pairs
        freq[complement] = freq[complement] - min_pairs

return pairs == n
```

Complessità ottimizzata: $\Theta(n)$ con hashing

Esercizio 2 (9 punti)

Testo: Data una stringa $X = (x_1, x_2, ..., x_n)$, si consideri la seguente quantità $\ell(i, j)$, definita per $1 \leq i \leq j \leq n$:

$\ell(i,j) = \{ 1$ se $i = j$ 2 se $i = j - 1$ $2 + \ell(i+1, j-1)$ se $(i < j-1) \wedge (x_i = x_j)$ $\sum_{k=i}^{j-1}(\ell(i,k) + \ell(k+1,j))$ se $(i < j-1) \wedge (x_i \neq x_j)$ }

(a) Scrivere una coppia di algoritmi INIT_L(X) e REC_L(X, i, j) per il calcolo memoizzato di $\ell(1, n)$.

(b) Determinarne la complessità al caso migliore T_best(n), supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

(a) Algoritmi per Calcolo Memoizzato

```
INIT_L(X)
n = X.length
// Crea tabella di memoizzazione
memo = new array[1..n, 1..n]

// Inizializza con -1 (non calcolato)
for i = 1 to n
    for j = 1 to n
        memo[i, j] = -1

// Calcola ℓ(1, n) con memoizzazione
return REC_L(X, 1, n, memo)

REC_L(X, i, j, memo)
// Caso già calcolato
if memo[i, j] ≠ -1
    return memo[i, j]
```

```

// Caso base: i = j
if i == j
    memo[i, j] = 1
    return 1

// Caso base: i = j - 1
if i == j - 1
    memo[i, j] = 2
    return 2

// Caso ricorsivo
if X[i] == X[j]  // Confronto caratteri (operazione contata)
    memo[i, j] = 2 + REC_L(X, i+1, j-1, memo)
else
    sum = 0
    for k = i to j - 1
        sum = sum + REC_L(X, i, k, memo) + REC_L(X, k+1, j, memo)
    memo[i, j] = sum

return memo[i, j]
```

(b) Analisi Complessità al Caso Migliore

Caso migliore: Quando tutti i caratteri della stringa sono uguali, X = (a, a, ..., a).

Traccia del calcolo per X = (a, a, a, a):

```
REC_L(X, 1, 4, memo):
  X[1] == X[4]? → Sì (confronto 1)
  Calcola REC_L(X, 2, 3, memo)
    X[2] == X[3]? → Sì (confronto 2)
    Calcola REC_L(X, 3, 2, memo) - caso base j < i → ritorna 2
    Ritorna 2 + 2 = 4
  Ritorna 2 + 4 = 6
```

Analisi generale per stringa di n caratteri tutti uguali:

Per $\ell(1, n)$ con tutti caratteri uguali:

- Confronto X[1] == X[n]: 1 confronto
- Chiamata ricorsiva a $\ell(2, n-1)$
    - Confronto X[2] == X[n-1]: 1 confronto
    - Chiamata ricorsiva a $\ell(3, n-2)$
        - ...

Struttura ricorsiva: Ogni chiamata $\ell(i, j)$ con $j - i \geq 2$ e tutti caratteri uguali:

- Esegue 1 confronto
- Chiama ricorsivamente $\ell(i+1, j-1)$

Numero di confronti:

- Per n caratteri tutti uguali
- Chiamate annidate: $\ell(1,n) \to \ell(2,n-1) \to \ell(3,n-2) \to \ldots \to \ell(k,k+1)$ o $\ell(k,k)$
- Numero di livelli: $\lfloor n/2 \rfloor$
- Confronti per livello: 1

$T_{best}(n) = \lfloor n/2 \rfloor = \Theta(n)$

Caso peggiore: Tutti caratteri diversi

- Ogni chiamata $\ell(i, j)$ esegue il ciclo for da k=i a j-1
- Esplora tutte le possibili partizioni
- Numero di confronti: $O(n^3)$

Confronti al caso migliore: $\Theta(n)$

Numero totale di sottoproblemi distinti: $O(n^2)$ (tutte le coppie i, j)

Complessità temporale totale al caso migliore: $\Theta(n^2)$ considerando tutte le operazioni

Complessità contando solo confronti tra caratteri: $T_{best}(n) = \Theta(n)$

RIEPILOGO COMPLESSITÀ

Esercizio	Algoritmo	Complessità	Note
1	Split (backtracking)	$O((2n)!/(2^n \cdot n!))$	Esponenziale
1	Split (hash optimized)	$\Theta(n)$	Con assunzioni
2	Memoizzazione	$T_{best}(n) = \Theta(n)$	Solo confronti

NOTE FINALI

Le soluzioni includono:

1. **Definizioni formali rigorose** (Θ-notation, Huffman)
2. **Dimostrazioni per induzione** complete (Domanda A)
3. **Algoritmi con memoizzazione** per problemi ricorsivi
4. **Analisi caso migliore/peggiore** con giustificazioni dettagliate

Tutte le dimostrazioni seguono lo standard richiesto per prove d'esame formali.