DOMANDE

Domanda A (7 punti)

Testo: Dare la definizione di max-heap. Dato un insieme S di elementi, memorizzato in parte in un min-heap A e in parte in un max-heap B, entrambi non vuoti, dare un algoritmo min(A,B) per trovare il minimo di S nelle due situazioni seguenti:

- (a) ogni elemento di A è minore o uguale a ogni elemento di B;
- (b) ogni elemento di B è minore o uguale a ogni elemento di A. In entrambi i casi scrivere lo pseudo-codice e valutare la complessità.

Definizione di Max-Heap

Un **max-heap** è una struttura dati ad albero binario quasi completo che soddisfa la **proprietà di heap massimo**: per ogni nodo i diverso dalla radice vale A[PARENT(i)] ≥ A[i].

Formalmente, dato un array A[1..n] che rappresenta un max-heap:

- La struttura è un albero binario completo o quasi completo
- $\forall i \in \{2, ..., n\}$: $A[\lfloor i/2 \rfloor] \geq A[i]$
- La radice A[1] contiene l'elemento massimo

Soluzione Caso (a): ogni elemento di A ≤ ogni elemento di B

Analisi: Se ogni elemento di A è minore o uguale a ogni elemento di B, il minimo di S si trova necessariamente in A. Poiché A è un min-heap, il minimo si trova alla radice.

Algoritmo:

```
min_caso_a(A, B)
1  return A[1]
```

Complessità: $\Theta(1)$ - accesso diretto alla radice del min-heap A.

Soluzione Caso (b): ogni elemento di B ≤ ogni elemento di A

Analisi: Se ogni elemento di B è minore o uguale a ogni elemento di A, il minimo di S si trova necessariamente in B. Poiché B è un max-heap, dobbiamo individuare il minimo tra tutti gli elementi. In un max-heap di dimensione n, il minimo si trova necessariamente tra le foglie, che sono $\lceil n/2 \rceil$ elementi.

Algoritmo:

```
min_caso_b(A, B)
1  n = B.heap-size
2  min = B[⌊n/2⌋ + 1]
3  for i = ⌊n/2⌋ + 2 to n
4      if B[i] < min
5          min = B[i]
6  return min
```

Complessità:

- Il ciclo itera su tutte le foglie del max-heap B
- Numero di foglie: ⌈n/2⌉
- Complessità: Θ(n)

Domanda B (7 punti)

Testo: Si consideri il problema di selezione di attività compatibili:

- (a) Definire il problema.
- (b) Descrivere brevemente l'algoritmo ottimo GREEDY-SEL visto in classe.
- (c) Fornire un esempio di algoritmo greedy non ottimo, motivandone la non ottimalità.

(a) Definizione del Problema

Problema della Selezione di Attività Compatibili:

Dato un insieme $S = \{a_1, a_2, ..., a_n\}$ di n attività, dove ogni attività a_i ha:

- Un tempo di inizio s_i
- Un tempo di fine f_i con $s_i < f_i$

Due attività a_i e a_j sono **compatibili** se i loro intervalli temporali non si sovrappongono, cioè:

- $f_i \leq s_j$ oppure $f_j \leq s_i$

Obiettivo: Trovare un sottoinsieme $A \subseteq S$ *di cardinalità massima tale che tutte le attività in A* siano mutuamente compatibili (compatibili a coppie).

Formulazione formale:

- Input: $S = \{a_1, ..., a_n\}$ con s_i, f_i per ogni a_i
- Output: $A^* \subseteq S$ tale che:
  - $\forall a_i, a_j \in A^*$ con $i \neq j$: $f_i \leq s_j \vee f_j \leq s_i$
  - $|A^*|$ è massimo

(b) Algoritmo Ottimo GREEDY-SEL

Idea: Scegliere iterativamente l'attività compatibile che finisce prima tra quelle rimanenti.

Scelta greedy: Dato un insieme di attività compatibili con un intervallo [i, j), selezionare l'attività $a_m \in S(i,j)$ tale che $f_m = \min\{f_k : a_k \in S(i,j)\}$.

Algoritmo:

```
GREEDY-SEL(s, f, n)
// Assunzione: le attività sono ordinate per tempo di fine crescente
1  A = {a₁}
2  k = 1
3  for i = 2 to n
4      if s[i] ≥ f[k]
5          A = A ∪ {aᵢ}
6          k = i
7  return A
```

Proprietà:

1. **Sottostruttura ottima:** Se A è *una soluzione ottima per S e contiene l'attività a_m che finisce per prima, allora A* \ $\{a_m\}$ è una soluzione ottima per il sottoproblema $S' = \{a_i \in S : s_i \geq f_m\}$.
2. **Proprietà di scelta greedy:** Esiste sempre una soluzione ottima che contiene l'attività che finisce per prima.

Complessità: Θ(n log n) per l'ordinamento + Θ(n) per la scansione = Θ(n log n)

(c) Esempio di Algoritmo Greedy Non Ottimo

Strategia alternativa: Selezionare l'attività con durata minima ($f_i - s_i$) tra quelle compatibili.

Controesempio:

Consideriamo 3 attività:

- a_1: $s_1 = 0$, $f_1 = 6$
- a_2: $s_2 = 1$, $f_2 = 3$
- a_3: $s_3 = 4$, $f_3 = 7$

Durate:

- $d_1 = 6$
- $d_2 = 2$ (minima)
- $d_3 = 3$

Algoritmo greedy "durata minima":

1. Seleziona a_2 (durata minima)
2. a_1 non è compatibile con a_2 ($f_1 = 6 > s_2 = 1$)
3. a_3 non è compatibile con a_2 ($s_3 = 4 > f_2 = 3$ ma a_3 si sovrappone con a_2 in [1,3])

Risultato: A = {a_2}, |A| = 1

Soluzione ottima:

- Selezionare {a_1} oppure {a_3}
- Entrambe hanno cardinalità 1

Consideriamo un esempio migliore:

- a_1: $s_1 = 0$, $f_1 = 4$
- a_2: $s_2 = 1$, $f_2 = 2$
- a_3: $s_3 = 3$, $f_3 = 7$

Algoritmo greedy "durata minima":

- Seleziona a_2 (durata 1)
- Risultato: A = {a_2}, |A| = 1

Soluzione ottima (GREEDY-SEL):

- Seleziona a_1 (finisce per prima tra compatibili iniziali), poi a_3
- Risultato: A = *{a_1, a_3}*, |*A*| = 2

Conclusione: L'algoritmo greedy basato sulla durata minima non è ottimo perché la scelta locale (durata minima) può bloccare opportunità di selezionare più attività globalmente.

ESERCIZI

Esercizio 1 (10 punti)

Testo: Sia dato un array V[1..n] i cui valori rappresentano la variazione giornaliera del valore di un titolo azionario. È noto che il titolo è stato prima perdita, con valori sempre negativi, poi ha iniziato a oscillare in giorni consecutivi tra valori positivi e negativi, e infine si è stabilizzato su valori positivi (dunque nella sequenza non ci possono essere due giorni positivi seguiti da un negativo). Realizzare un algoritmo divide et impera Split(V) che individua il giorno in cui il titolo ha iniziato a essere stabile su valori positivi, ovvero il minimo indice i ∈ [1, n] tale che per ogni j ≥ i vale V[j] > 0. Se il titolo rimane sempre su valori positivi, ritornare 0. Ad es., se V = [-1,-2,2,-1,6,3] l'indice da tornare sarà 5, mentre invece per V= [-1,-2,-2,-1,6,-3] si ritornerà 0. Fornire lo pseudocodice di Split(V), motivarne la correttezza e individuarne la complessità. Si assuma che non ci siano valori nulli.

Analisi del Problema

Proprietà della sequenza:

1. Fase iniziale: tutti valori negativi $V[1..k_1]$
2. Fase oscillante: alternanza positivi/negativi $V[k_1+1..k_2]$
3. Fase stabile: tutti valori positivi $V[k_2+1..n]$

Obiettivo: Trovare $k_2 + 1$, il minimo indice i tale che $\forall j \geq i$: $V[j] > 0$

Osservazione chiave: La fase stabile corrisponde al più lungo suffisso di valori tutti positivi.

Strategia Divide et Impera

Dividiamo l'array a metà e verifichiamo:

1. Se tutta la metà destra è positiva → soluzione nella metà sinistra o all'inizio della metà destra
2. Se la metà destra contiene negativi → soluzione nella metà destra

Criterio di ricerca: Cercare il primo indice dopo il quale tutti gli elementi sono positivi.

Pseudocodice

```
SPLIT(V)
1  return SPLIT-REC(V, 1, V.length)

SPLIT-REC(V, left, right)
1  if left == right
2      if V[left] > 0
3          return left
4      else
5          return 0
6
7  mid = ⌊(left + right) / 2⌋
8
9  // Verifica se tutta la metà destra è positiva
10 all_positive_right = true
11 for i = mid + 1 to right
12     if V[i] ≤ 0
13         all_positive_right = false
14         break
15
16 if all_positive_right
17     // Soluzione nella metà sinistra o è mid+1
18     result = SPLIT-REC(V, left, mid)
19     if result == 0
20         return mid + 1
21     else
```

```
22         return result
23 else
24     // Soluzione nella metà destra
25     return SPLIT-REC(V, mid + 1, right)
```

Versione Ottimizzata (senza scansione lineare)

```
SPLIT-OPTIMIZED(V)
1  return SPLIT-OPT-REC(V, 1, V.length)

SPLIT-OPT-REC(V, left, right)
1  if left == right
2      if V[left] > 0
3          return left
4      else
5          return 0
6
7  if left > right
8      return 0
9
10 mid = ⌊(left + right) / 2⌋
11
12 // Caso 1: V[mid] ≤ 0 → soluzione a destra di mid
13 if V[mid] ≤ 0
14     return SPLIT-OPT-REC(V, mid + 1, right)
15
16 // Caso 2: V[mid] > 0 e V[right] ≤ 0 → soluzione tra mid e right
17 if V[right] ≤ 0
18     return SPLIT-OPT-REC(V, mid + 1, right)
19
20 // Caso 3: V[mid] > 0 e V[right] > 0 → verifica a sinistra
21 result = SPLIT-OPT-REC(V, left, mid - 1)
22 if result == 0
23     return mid
24 else
25     return result
```

Dimostrazione di Correttezza

Base induttiva: Se left == right, l'array ha un solo elemento. Se V[left] > 0, questo è l'inizio della fase stabile. Altrimenti ritorniamo 0.

Passo induttivo: Supponiamo l'algoritmo corretto per array di dimensione < n. Per un array di dimensione n:

Caso 1: V[mid] ≤ 0

- Tutti gli elementi fino a mid (incluso) non possono essere l'inizio della fase stabile

- La soluzione deve trovarsi in [mid+1, right]
- Per ipotesi induttiva, SPLIT-OPT-REC(V, mid+1, right) ritorna correttamente

Caso 2: V[mid] > 0 e V[right] ≤ 0

- Esiste almeno un negativo in [mid, right]
- La fase stabile non può iniziare prima della posizione di questo negativo
- Per ipotesi induttiva, la ricorsione in [mid+1, right] è corretta

Caso 3: V[mid] > 0 e V[right] > 0

- Tutti gli elementi da mid a right sono positivi
- Dobbiamo verificare se esistono altri positivi consecutivi prima di mid
- Se la ricorsione su [left, mid-1] ritorna 0, mid è l'inizio della fase stabile
- Altrimenti ritorniamo il risultato della ricorsione

Terminazione: Ogni chiamata riduce la dimensione del problema di almeno metà, garantendo la terminazione.

Analisi della Complessità

Relazione di ricorrenza:

- $T(n) = T(n/2) + \Theta(1)$
- Ogni chiamata ricorsiva processa metà dell'array
- Il lavoro per chiamata è costante

Soluzione (Master Theorem):

- $a = 1, b = 2, f(n) = \Theta(1)$
- $n^{\log_b a} = n^0 = 1$
- $f(n) = \Theta(n^{\log_b a}) \rightarrow$ Caso 2 del Master Theorem
- $T(n) = \Theta(\log n)$

Complessità totale: $\Theta(\log n)$

Esercizio 2 (8 punti)

Testo: Date due stringhe $X = (x_1, x_2, ..., x_m)$ e $Y = (y_1, y_2, ..., y_n)$, si consideri la seguente quantità $\ell(i,j)$, definita per ogni coppia di valori i, j con $0 \le i \le m$ e $0 \le j \le n$:

$\ell(i,j) = \{ 1$ se $i = 0$ o $j = 0$ $3\ell(i, j-1)$ se $i,j > 0$ e $x_i = y_j$ $2\ell(i-1, j-1) - \ell(i-1, j)$ se $i,j > 0$ e $x_i \neq y_j \}$

Si vuole calcolare la quantità $q = \max\{\ell(i,j) : 0 \le i \le m, 0 \le j \le n\}$.

- (a) Scrivere un algoritmo bottom-up per il calcolo di q.
- (b) Determinare la complessità esatta dell'algoritmo, supponendo che le uniche operazioni di costo unitario e non nullo siano i confronti tra caratteri.

(a) Algoritmo Bottom-Up

Struttura della soluzione:

- Matrice L[0..m, 0..n] per memorizzare i valori ℓ(i,j)
- Variabile q per tracciare il massimo

Pseudocodice:

```
COMPUTE-Q(X, Y, m, n)
1  // Inizializzazione
2  for i = 0 to m
3      L[i, 0] = 1
4  for j = 0 to n
5      L[0, j] = 1
6
7  q = 1  // Massimo iniziale (dai casi base)
8
9  // Calcolo bottom-up
10 for i = 1 to m
11     for j = n-1 downto I
12         if X[i] == Y[j]
13             L[i, j] = 3 * L[i, j-1]
14         else
15             L[i, j] = 2 * L[i-1, j-1] - L[i-1, j]
16
17         // Aggiornamento del massimo
18         if L[i, j] > q
19             q = L[i, j]
20
21 return q
```

Correttezza:

1. **Inizializzazione:** I casi base ℓ(i, 0) = 1 e ℓ(0, j) = 1 sono correttamente impostati.
2. **Calcolo iterativo:** Per ogni cella (i, j) con i, j > 0, calcoliamo ℓ(i, j) secondo la ricorrenza, utilizzando valori già calcolati.
3. **Massimo:** Tracciamo il massimo durante il calcolo, garantendo q = max{ℓ(i,j)}.

(b) Analisi della Complessità

Operazioni:

- Inizializzazione righe: Θ(m)

- Inizializzazione colonne: Θ(n)
- Ciclo principale: doppio ciclo su i e j
  - Iterazioni: m × n
  - Per ogni iterazione: 1 confronto tra caratteri (X[i] == Y[j])
  - Altre operazioni (aritmetiche, assegnamenti): non conteggiate per ipotesi

Confronti tra caratteri:

- Numero totale di confronti: m × n
- Ogni coppia (i, j) con 1 ≤ i ≤ m, 1 ≤ j ≤ n richiede esattamente 1 confronto

Complessità esatta (contando solo confronti): Θ(mn)

Complessità nel caso migliore T_best(n): Supponendo m = n per simmetria:

- Numero di confronti: n^2
- **$T_{best}(n) = \Theta(n^2)$**
