ESERCIZI SUPPLEMENTARI - ARRAY E DIVIDE-ET-IMPERA

1. ANAGRAMMA SU ALFABETO BINARIO

Problema: Verificare se due stringhe x e y su alfabeto {0,1} sono anagrammi (stessi caratteri, ordine diverso).

Algoritmo

```
Anagramma(x, y)
    if x.length != y.length
        return false

    diff[0..1] = [0, 0]

    for i = 1 to x.length
        diff[x[i]]++    // incrementa contatore carattere in x
        diff[y[i]]--    // decrementa contatore carattere in y

    if diff[0] == 0 and diff[1] == 0
        return true
    else
        return false
```

Correttezza

Invariante: `diff[c]` = numero di occorrenze di carattere c in x[1..i] - numero di occorrenze in y[1..i]

- Se x e y sono anagrammi, ogni carattere appare stesso numero di volte
- Quindi diff[0] = diff[1] = 0 alla fine
- Se diff[c] ≠ 0 per qualche c, x e y non sono anagrammi

Complessità

Θ(n) - un solo passaggio sulle stringhe

Generalizzazione

Per alfabeto di dimensione k: usare array diff[0..k-1], complessità Θ(n+k).

---

2. REVERSE COUNTING SORT

Problema: Ordinare array A[1..n] con chiavi in [0..k-1] in ordine **decrescente** usando Counting Sort.

Algoritmo

```
RevCountingSort(A, B, n, k)
    allocate C[0..k-1]

    // Fase 1: conta occorrenze
    for i = 0 to k-1
        C[i] = 0
    for j = 1 to n
        C[A[j]]++

    // Fase 2: somme cumulative INVERSE
    for j = k-2 downto 0
        C[j] = C[j] + C[j+1]

    // Fase 3: posiziona elementi
    for j = n downto 1
        B[C[A[j]]] = A[j]
        C[A[j]]--

    return B
```

Modifica chiave

Counting Sort standard: `for j=1 to k-1: C[j]=C[j-1]+C[j]`
Reverse Counting Sort: `for j=k-2 downto 0: C[j]=C[j]+C[j+1]`

Correttezza

- Somme cumulative inverse: C[j] = numero di elementi ≥ j
- Questo posiziona elementi grandi prima di quelli piccoli
- L'algoritmo rimane stabile (scorre A dal fondo)

Complessità

O(n + k)

- Fase 1: Θ(k + n)
- Fase 2: Θ(k)
- Fase 3: Θ(n)
- Totale: O(n + k)

---

3. DOUBLE - VERIFICA RELAZIONE 2x

Problema: Dato array A[1..n] ordinato crescente, verificare se esiste coppia (i,j) tale che A[j] = 2·A[i].

Algoritmo

```
double(A, n)
    i = 1
    j = 1

    while (i <= n) and (j <= n)
        if 2*A[i] == A[j]
            return (i, j)
        else if 2*A[i] > A[j]
            j++
        else
            i++

    return (0, 0)  // non trovato
```

Correttezza

Invariante: ∀i' ∈ [1, i). ∀j' ∈ [1, j). 2·A[i'] ≠ A[j']

- **Inizializzazione**: Nessuna coppia esplorata, invariante vera
- **Mantenimento**:
  - Se 2·A[i] = A[j], trovato → return
  - Se 2·A[i] > A[j], incrementa j (A[j] troppo piccolo)
  - Se 2·A[i] < A[j], incrementa i (A[i] troppo piccolo)
- **Conclusione**: Se while termina senza trovare, non esiste coppia

Complessità

O(n) - lineare

- Numero massimo iterazioni: 2n
- i e j aumentano monotonamente
- Ogni iterazione: O(1)

---

4. MIN CON PERMUTAZIONI

Problema: Dati due array A[1..n] e B[1..n] che sono permutazioni uno dell'altro, trovare il minimo confrontando solo elementi di A con B.

Algoritmo

```
min(A, B, n)
    i = 1
    j = 1

    while i <= n and j <= n
        if A[i] <= B[j]
            j++
        else
            i++

    if i <= n
        return A[i]
    else
        return B[j]
```

Correttezza

Invariante: Il minimo globale non è in A[1..i-1] né in B[1..j-1]

- Quando A[i] ≤ B[j], incrementa j (B[j] non è minimo)
- Quando A[i] > B[j], incrementa i (A[i] non è minimo)
- Alla fine, il primo tra A[i] o B[j] ancora non escluso è il minimo

Complessità

Θ(n)

- Massimo 2n iterazioni (una per escludere ogni elemento)
- Ogni iterazione: O(1)

---

5. CHECKSUM TRE ARRAY

Problema: Dati tre array A[1..n], B[1..n], C[1..n] di interi distinti, verificare se esistono indici i, j, k tali che A[i] + B[j] = C[k].

Algoritmo

```
checkSum(A, B, C, n)
    MergeSort(B, 1, n)
    MergeSort(C, 1, n)

    for i = 1 to n
        j = 1
```

```
        k = n
        found = false

        while j <= n and k >= 1 and not found
            if A[i] + B[j] == C[k]
                return true
            else if A[i] + B[j] < C[k]
                j++
            else
                k--

    return false
```

Correttezza

Invariante esterno: Se found = false, per ogni i' < i e qualunque j', k': A[i'] + B[j'] ≠ C[k']

Invariante interno: Per i fissato, se found = false:

- ∀j' < j: A[i] + B[j'] < C[k]
- ∀k' > k: A[i] + B[j] > C[k']

Complessità

O(n²)

- Ordinamento: O(n log n) per B e C
- Ciclo esterno: n iterazioni
- Ciclo interno: O(n) per ogni i (j cresce, k decresce)
- Totale: O(n log n + n²) = O(n²)

---

6. BI-ORDINATO

Problema: Array A[1..n] è bi-ordinato se esiste k tale che A[1..k] crescente e A[k..n] decrescente.

Algoritmo

```
isBiOrdinato(A, n)
    // Trova massimo
    maxIdx = 1
    for i = 2 to n
        if A[i] > A[maxIdx]
            maxIdx = i

    // Verifica crescente prima di maxIdx
```

```
    for i = 1 to maxIdx-1
        if A[i] > A[i+1]
            return false

    // Verifica decrescente dopo maxIdx
    for i = maxIdx to n-1
        if A[i] < A[i+1]
            return false

    return true
```

Correttezza

- Se A è bi-ordinato, il massimo è in posizione k
- A[1..k] crescente ⇔ ∀i ∈ [1,k): A[i] ≤ A[i+1]
- A[k..n] decrescente ⇔ ∀i ∈ [k,n): A[i] ≥ A[i+1]

Complessità

Θ(n) - tre scansioni lineari

---

7. FIX - INDICE FISSO

Problema: Dato array A[1..n] ordinato crescente con elementi distinti, trovare indice i tale che A[i] = i.

Algoritmo Divide-et-Impera

```
Fix(A, p, r)
    if p > r
        return 0  // non trovato

    q = floor((p + r) / 2)

    if A[q] == q
        return q
    else if A[q] > q
        return Fix(A, p, q-1)  // cerca a sinistra
    else
        return Fix(A, q+1, r)  // cerca a destra

Fix(A)
    return Fix(A, 1, A.length)
```

Correttezza

Proprietà chiave: Se A[q] > q, allora ∀i > q: A[i] > i

Dimostrazione: A è ordinato con elementi distinti, quindi:

- A[q+1] ≥ A[q] + 1 > q + 1
- Per induzione: A[i] ≥ A[q] + (i-q) > q + (i-q) = i

Analogamente, se A[q] < q, allora ∀i < q: A[i] < i.

Complessità

Θ(log n)

- Ricorrenza: T(n) = T(n/2) + c
- Soluzione: T(n) = Θ(log n)

---

8. SUBSEQ - SOTTOSEQUENZA

Problema: Verificare se X[1..m] è sottosequenza di Y[1..n] (caratteri di X appaiono in Y nello stesso ordine).

Algoritmo Ricorsivo

```
subseq(X, Y, m, n)
    if m == 0
        return true  // stringa vuota è sottosequenza
    if n == 0
        return false  // Y vuota, X non vuota

    if X[m] == Y[n]
        return subseq(X, Y, m-1, n-1)
    else
        return subseq(X, Y, m, n-1)
```

Correttezza

- **Caso base**: X vuota è sottosequenza di qualsiasi Y
- **Caso ricorsivo**:
  - Se X[m] = Y[n], verifica se X[1..m-1] è sottosequenza di Y[1..n-1]
  - Se X[m] ≠ Y[n], verifica se X[1..m] è sottosequenza di Y[1..n-1]

Complessità

O(n)

- Caso peggiore: m = 1, scorre tutto Y
- Ogni chiamata ricorsiva decrementa n di 1
- Massimo n chiamate ricorsive

---

9. ALT - ARRAY ALTERNANTE

Problema: Verificare se array è alternante (no elementi contigui uguali, e per ogni i ≤ n-2: ai < ai+1 > ai+2 oppure ai > ai+1 < ai+2).

Algoritmo Ricorsivo

```
alt(A, n)
    if n <= 1
        return true
    if n == 2
        return A[1] != A[2]

    // Determina direzione iniziale
    if A[1] < A[2]
        return altRec(A, 2, n, "up")
    else if A[1] > A[2]
        return altRec(A, 2, n, "down")
    else
        return false  // A[1] == A[2]

altRec(A, i, n, dir)
    if i == n
        return true

    if A[i] == A[i+1]
        return false

    if dir == "up"
        if A[i] > A[i+1]
            return altRec(A, i+1, n, "down")
        else
            return false
    else  // dir == "down"
        if A[i] < A[i+1]
            return altRec(A, i+1, n, "up")
        else
            return false
```

Correttezza

Invariante: direzione alterna correttamente fino a posizione i

Complessità

O(n) - una visita dell'array

---

10. MONETE FALSE CON BILANCIA

Problema: Date n monete con una falsa (pesa meno), trovare la moneta falsa usando una bilancia a due piatti.

Algoritmo Divide-et-Impera

```
trovaFalsa(S)
    if |S| == 1
        return S[1]  // unica moneta rimasta

    // Dividi in 3 gruppi
    k = floor(|S| / 3)
    S1 = S[1..k]
    S2 = S[k+1..2k]
    S3 = S[2k+1..|S|]

    risultato = confronta(S1, S2)  // usa bilancia

    if risultato == "S1 più leggero"
        return trovaFalsa(S1)
    else if risultato == "S2 più leggero"
        return trovaFalsa(S2)
    else  // "uguale"
        return trovaFalsa(S3)
```

Correttezza

- Un confronto elimina almeno 2/3 delle monete
- La moneta falsa è sempre nel gruppo più leggero (o in S3 se S1=S2)

Complessità

O(log n)

- Ricorrenza: $T(n) = T(n/3) + c$
- Soluzione: $T(n) = \Theta(\log_3 n) = \Theta(\log n)$

Limite Inferiore

Ω(log n) - albero decisionale

- Ogni confronto divide spazio ricerca in ≤ 3 parti
- Servono almeno $\lceil \log_3 n \rceil$ confronti
- Algoritmo è ottimo

Generalizzazione

Se non si sa se falsa pesa più o meno:

- Dividere in S1, S2, S3 con |S1| = |S2|
- Se S1 = S2, falsa in S3 (peso incognito)
- Se S1 ≠ S2, falsa nel più leggero o più pesante
- Servono informazioni aggiuntive per risolvere

RIEPILOGO COMPLESSITÀ

Esercizio	Complessità	Note
Anagramma	$\Theta(n)$	Lineare
RevCountingSort	$O(n+k)$	Counting sort inverso
Double	$O(n)$	Due puntatori
Min permutazioni	$\Theta(n)$	Due puntatori
CheckSum	$O(n^2)$	Loop nidificati
Bi-ordinato	$\Theta(n)$	Scansioni lineari
Fix	$\Theta(\log n)$	Divide-et-impera
Subseq	$O(n)$	Ricorsivo lineare
Alt	$O(n)$	Ricorsivo lineare
Monete false	$O(\log n)$	Ottimo