ESERCIZI ARRAY CON DIVIDE ET IMPERA

Guida completa con soluzioni per esame

DEFINIZIONI FONDAMENTALI

Paradigma Divide et Impera

1. **Divide**: Suddividi problema in sottoproblemi più piccoli
2. **Impera**: Risolvi ricorsivamente i sottoproblemi
3. **Combina**: Unisci soluzioni per ottenere soluzione problema originale

Algoritmi classici

- MergeSort: O(n log n)
- QuickSort: O(n log n) medio, O(n²) peggiore
- QuickSelect: O(n) medio
- Ricerca binaria: O(log n)

CATEGORIA 1: MERGESORT

ES 1.1 - MergeSort classico

Traccia: Ordinare array A[p..r] con divide et impera

Soluzione:

```
MergeSort(A, p, r)
1. if p < r
2.     q = ⌊(p+r)/2⌋
3.     MergeSort(A, p, q)
4.     MergeSort(A, q+1, r)
5.     Merge(A, p, q, r)

Merge(A, p, q, r)
1. n1 = q - p + 1
2. n2 = r - q
3. create arrays L[1..n1+1] and R[1..n2+1]
4. for i = 1 to n1
5.     L[i] = A[p+i-1]
6. for j = 1 to n2
```

```
7.     R[j] = A[q+j]
8. L[n1+1] = +∞  // sentinella
9. R[n2+1] = +∞
10. i = 1
11. j = 1
12. for k = p to r
13.    if L[i] ≤ R[j]
14.        A[k] = L[i]
15.        i = i + 1
16.    else
17.        A[k] = R[j]
18.        j = j + 1
```

Complessità: T(n) = 2T(n/2) + Θ(n) = Θ(n log n)
Spazio: O(n) per array ausiliari

ES 1.2 - MergeSort senza array ausiliari (alternativo)

Traccia: MergeSort che alterna tra due array per evitare copie multiple

Soluzione:

```
MergeSort(A)
1. n = A.length
2. allocate B[1..n]
3. MergeSortAlt(A, B, 1, n, true)

MergeSortAlt(A, B, p, r, destA)
// destA=true → ordina in A, destA=false → ordina in B
1. if p < r
2.     q = ⌊(p+r)/2⌋
3.     MergeSortAlt(A, B, p, q, not destA)
4.     MergeSortAlt(A, B, q+1, r, not destA)
5.     if destA
6.         Merge(B, A, p, q, r)  // source=B, target=A
7.     else
8.         Merge(A, B, p, q, r)  // source=A, target=B
9. else
10.    if not destA
11.        B[p] = A[p]

Merge(S, T, p, q, r)  // merge da Source a Target
1. i = p
2. j = q+1
3. for k = p to r
4.     if i ≤ q and (j > r or S[i] ≤ S[j])
5.         T[k] = S[i]
```

```
6.          i = i + 1
7.      else
8.          T[k] = S[j]
9.          j = j + 1
```

Complessità: O(n log n)
Vantaggio: Evita copie inutili alternando target

CATEGORIA 2: QUICKSORT E PARTITIONING

ES 2.1 - QuickSort classico

Traccia: Ordinare array usando partitioning

Soluzione:

```
QuickSort(A, p, r)
1. if p < r
2.      q = Partition(A, p, r)
3.      QuickSort(A, p, q−1)
4.      QuickSort(A, q+1, r)

Partition(A, p, r)
// Usa A[p] come pivot
1. i = p − 1
2. j = r + 1
3. x = A[p]
4. while true
5.      repeat j = j − 1 until A[j] ≤ x
6.      repeat i = i + 1 until A[i] ≥ x
7.      if i < j
8.          A[i] ⇔ A[j]
9.      else
10.         return j
```

Complessità:

- Caso migliore/medio: O(n log n)
- Caso peggiore: O(n²) se pivot sempre minimo/massimo

ES 2.2 - QuickSort con pivot randomizzato

Traccia: Evitare caso peggiore con scelta casuale pivot

Soluzione:

```
RandomizedPartition(A, p, r)
1. i = Random(p, r)
2. A[p] ⇔ A[i]
3. return Partition(A, p, r)

RandomizedQuickSort(A, p, r)
1. if p < r
2.     q = RandomizedPartition(A, p, r)
3.     RandomizedQuickSort(A, p, q−1)
4.     RandomizedQuickSort(A, q+1, r)
```

Complessità attesa: O(n log n)

ES 2.3 - Tripartition (3-Way QuickSort)

Traccia: Gestire array con molti duplicati efficientemente

Soluzione:

```
Tripartition(A, p, r)
// Partiziona in 3: <x, =x, >x
1. i = p − 1
2. k = p
3. j = r
4. x = A[p]
5. while k ≤ j
6.     if A[k] < x
7.         i = i + 1
8.         A[i] ⇔ A[k]
9.         k = k + 1
10.    else if A[k] > x
11.        A[k] ⇔ A[j]
12.        j = j − 1
13.    else
14.        k = k + 1
15. return (i+1, j)  // ritorna [i+1..j] = zona uguale a x

QuickSort3Way(A, p, r)
1. if p < r
2.     q1, q2 = Tripartition(A, p, r)
3.     QuickSort3Way(A, p, q1−1)
4.     QuickSort3Way(A, q2+1, r)
```

Complessità: O(n) se tutti elementi uguali, O(n log n) altrimenti

CATEGORIA 3: SELECT (k-esimo elemento)

ES 3.1 - QuickSelect

Traccia: Trovare k-esimo elemento più piccolo senza ordinare tutto l'array

Soluzione:

```
QuickSelect(A, p, r, k)
1. if p = r
2.     return A[p]
3. q = Partition(A, p, r)
4. if k = q
5.     return A[q]
6. else if k < q
7.     return QuickSelect(A, p, q-1, k)
8. else
9.     return QuickSelect(A, q+1, r, k)
```

Complessità:

- Caso medio: O(n)
- Caso peggiore: O(n²)

Ricorrenza caso medio:

- T(n) = T(n/2) + Θ(n) = O(n)

CATEGORIA 4: RICERCA BINARIA E VARIANTI

ES 4.1 - Ricerca binaria classica

Traccia: Cercare valore k in array A[p..r] ordinato crescente

Soluzione:

```
BinarySearch(A, p, r, k)
1. if p ≤ r
2.     q = ⌊(p+r)/2⌋
3.     if A[q] = k
4.         return q
5.     else if A[q] < k
6.         return BinarySearch(A, q+1, r, k)
7.     else
8.         return BinarySearch(A, p, q-1, k)
```

```
9. else
10.    return 0  // non trovato
```

Complessità: T(n) = T(n/2) + c = O(log n)

ES 4.2 - Ricerca elemento > x (primo elemento maggiore)

Traccia: In array ordinato, trovare minimo indice i tale che A[i] > x

Soluzione:

```
over(A, p, r, x)
1. if r - p < 0
2.     return r + 1
3. else
4.     q = ⌊(p+r)/2⌋
5.     if A[q] ≤ x
6.         return over(A, q+1, r, x)
7.     else
8.         return over(A, p, q-1, x)
```

Complessità: O(log n)

Correttezza (induzione su n = r-p+1):

- Base (n=0): ritorna r+1 correttamente
- Passo: se A[q]≤x tutti elementi in [p,q] sono ≤x → ricorre su [q+1,r]
  se A[q]>x il minimo è in [p,q-1] o è q stesso

ES 4.3 - Ricerca indice stabile (A[i] = i)

Traccia: Array ordinato crescente con elementi distinti. Trovare i tale che A[i] = i

Soluzione:

```
stab(A, p, r)
1. if p ≤ r
2.     q = ⌊(p+r)/2⌋
3.     if A[q] = q
4.         return q
5.     else if A[q] > q
6.         return stab(A, p, q-1)
7.     else
8.         return stab(A, q+1, r)
```

```
9. else
10.    return 0
```

Complessità: O(log n)

Proprietà chiave:

- Se A[i] > i allora A[j] > j per ogni j > i
- Se A[i] < i allora A[j] < j per ogni j < i

Dimostrazione: Array ordinato distinti → differenza tra indici e valori monotona

---

ES 4.4 - Ricerca GAP

Traccia: Dato A[1..n] ordinato con A[n]-A[1] ≥ n, trovare gap dove A[i+1]-A[i] > 1

Proprietà:

```
Lemma: Se A[n]−A[1] ≥ n allora esiste gap in A
Dimostrazione: Somma gap ≥ A[n]−A[1] ≥ n
               Con n−1 posizioni di gap → almeno un gap > 1
```

Soluzione:

```
gap(A, p, r)
1. if p = r − 1
2.     return p  // gap in posizione p
3. else
4.     q = ⌊(p+r)/2⌋
5.     if A[q] − A[p] ≥ q − p + 1
6.         return gap(A, p, q)
7.     else
8.         return gap(A, q, r)
```

Complessità: T(n) = T(n/2) + Θ(1) = O(log n)

Correttezza: Una delle due metà ha differenza ≥ lunghezza → contiene gap

---

CATEGORIA 5: PROBLEMI SPECIALI

ES 5.1 - Contare inversioni

Traccia: Coppia (i,j) è inversione se i<j ma A[i]>A[j]. Contare inversioni.

Soluzione:

```
CountInversions(A, p, r)
1. if p < r
2.     q = ⌊(p+r)/2⌋
3.     left = CountInversions(A, p, q)
4.     right = CountInversions(A, q+1, r)
5.     cross = MergeCount(A, p, q, r)
6.     return left + right + cross
7. else
8.     return 0

MergeCount(A, p, q, r)
// Come Merge ma conta inversioni cross-boundary
1. count = 0
2. i = p, j = q+1, k = p
3. create temp[p..r]
4. while i ≤ q and j ≤ r
5.     if A[i] ≤ A[j]
6.         temp[k] = A[i]
7.         i = i + 1
8.     else
9.         temp[k] = A[j]
10.        count = count + (q - i + 1)  // inversioni!
11.        j = j + 1
12.    k = k + 1
13. // Copia rimanenti
14. copy remaining elements...
15. copy temp to A
16. return count
```

Complessità: O(n log n)

ES 5.2 - Massimo in array

Traccia: Trovare massimo con divide et impera

Soluzione:

```
Max(A, p, r)
1. if p = r
2.     return A[p]
3. else
4.     q = ⌊(p+r)/2⌋
5.     m1 = Max(A, p, q)
6.     m2 = Max(A, q+1, r)
```

```
7.      if m1 < m2
8.          return m2
9.      else
10.         return m1
```

Complessità: T(n) = 2T(n/2) + c = O(n)

COUNTING SORT (non divide et impera, ma importante)

ES 6.1 - Counting Sort

Traccia: Ordinare array A[1..n] con valori in [0..k]

Soluzione:

```
CountingSort(A, k)
1. create C[0..k] = all zeros
2. create B[1..n]
3. // Conta frequenze
4. for i = 1 to n
5.      C[A[i]] = C[A[i]] + 1
6. // Calcola posizioni cumulative
7. for i = 1 to k
8.      C[i] = C[i] + C[i-1]
9. // Posiziona elementi in ordine
10. for i = n downto 1
11.     B[C[A[i]]] = A[i]
12.     C[A[i]] = C[A[i]] - 1
13. return B
```

Complessità: O(n + k)
Stabile: Mantiene ordine relativo elementi uguali

ANALISI COMPLESSITÀ RICORRENZE

Ricorrenze comuni divide et impera:

Ricorrenza	Soluzione	Esempio
T(n) = T(n/2) + c	O(log n)	Ricerca binaria
T(n) = 2T(n/2) + c	O(n)	Max ricorsivo
T(n) = T(n/2) + cn	O(n)	QuickSelect medio

Ricorrenza	Soluzione	Esempio
T(n) = 2T(n/2) + cn	O(n log n)	MergeSort
T(n) = T(n-1) + cn	O(n²)	QuickSort peggiore

TEMPLATE RISOLUZIONE DIVIDE ET IMPERA

Step 1: Identificare schema

- Problema si scompone in sottoproblemi simili?
- Esiste punto di divisione naturale (es. metà array)?
- Soluzioni sottoproblemi si combinano facilmente?

Step 2: Definire ricorsione

```
Solve(A, p, r)
1. if caso_base
2.     return soluzione_diretta
3. // DIVIDE
4. q = punto_divisione
5. // IMPERA
6. sol1 = Solve(A, p, q)
7. sol2 = Solve(A, q+1, r)
8. // COMBINA
9. return combine(sol1, sol2)
```

Step 3: Analizzare complessità

- Scrivi ricorrenza: T(n) = aT(n/b) + f(n)
- Applica Master Theorem se possibile
- O controlla con substitution method

CATEGORIA 5: COUNTING SORT

ES 5.1 - Counting Sort Completo

Problema: Ordinare array A[1..n] con chiavi intere in range [0..k-1] in tempo lineare.

Caratteristiche:

- **Non basato su confronti**: usa valore chiavi come indici
- **Stabile**: preserva ordine relativo elementi con stessa chiave

- **Complessità**: O(n + k) tempo, O(n + k) spazio
- **Limitazione**: richiede k = O(n) per essere efficiente

Pseudocodice:

```
CountingSort(A, B, n, k)
    // A: array input [1..n]
    // B: array output [1..n]
    // k: range chiavi [0..k-1]

    allocate C[0..k-1]

    // Fase 1: Conta occorrenze di ogni chiave
    for i = 0 to k-1
        C[i] = 0
    for j = 1 to n
        C[A[j]]++

    // Fase 2: Calcola posizioni cumulative
    for i = 1 to k-1
        C[i] = C[i-1] + C[i]

    // Fase 3: Posiziona elementi (dal fondo per stabilità)
    for j = n downto 1
        B[C[A[j]]] = A[j]
        C[A[j]]--

    return B
```

Dimostrazione Correttezza:

Fase 1: Dopo primo ciclo, C[i] = numero occorrenze chiave i in A

- **Invariante**: Dopo j iterazioni, C[i] conta occorrenze di i in A[1..j]

Fase 2: Dopo secondo ciclo, C[i] = numero elementi ≤ i

- **Invariante**: C[i] = |{j : A[j] ≤ i}|
- Questo indica posizione finale dell'ultimo elemento con chiave i

Fase 3: Scorre A dal fondo, posiziona ogni elemento in B

- **Invariante**: Elementi in B[k+1..n] sono correttamente ordinati e stabili
- **Stabilità**: Scorrere dal fondo preserva ordine originale per elementi uguali

Esempio Esecuzione:

Input: A = [2, 5, 3, 0, 2, 3, 0, 3]
k = 6 (range 0..5)

Fase 1 - Conta:

```
C[0] = 2  (due 0)
C[1] = 0  (nessun 1)
C[2] = 2  (due 2)
C[3] = 3  (tre 3)
C[4] = 0  (nessun 4)
C[5] = 1  (un 5)
```

Fase 2 - Cumulative:

```
C[0] = 2
C[1] = 2 (2+0)
C[2] = 4 (2+2)
C[3] = 7 (4+3)
C[4] = 7 (7+0)
C[5] = 8 (7+1)
```

Significato: elementi con chiave 3 occupano posizioni 5,6,7

Fase 3 - Posizionamento (dal fondo):

```
j=8: A[8]=3 → B[7]=3, C[3]=6
j=7: A[7]=0 → B[2]=0, C[0]=1
j=6: A[6]=3 → B[6]=3, C[3]=5
j=5: A[5]=2 → B[4]=2, C[2]=3
j=4: A[4]=0 → B[1]=0, C[0]=0
j=3: A[3]=3 → B[5]=3, C[3]=4
j=2: A[2]=5 → B[8]=5, C[5]=7
j=1: A[1]=2 → B[3]=2, C[2]=2
```

Output: B = [0, 0, 2, 2, 3, 3, 3, 5]

Analisi Complessità:

- Fase 1: Θ(k) + Θ(n) = Θ(n+k)
- Fase 2: Θ(k)
- Fase 3: Θ(n)
- **Totale**: Θ(n+k)
- **Spazio**: Θ(n+k) per array B e C

Stabilità:

Counting Sort è **stabile** perché:

1. Scorre A dal fondo (j = n downto 1)
2. Per chiavi uguali, l'elemento più a destra in A viene posizionato più a destra in B
3. Questo preserva l'ordine relativo

Dimostrazione: Se A[i] = A[j] con i < j:

- j viene processato prima (loop dal fondo)
- A[j] ottiene posizione C[A[j]]
- C[A[j]] viene decrementato
- Quando A[i] viene processato, ottiene C[A[i]] < posizione di A[j]
- Quindi posizione(A[i]) < posizione(A[j]) ✓

Confronto Algoritmi Stabili:

- Insertion Sort: O(n²), stabile
- Merge Sort: O(n log n), stabile
- Counting Sort: O(n+k), stabile
- Radix Sort: O(d(n+b)), stabile
- Heap Sort: O(n log n), **NON stabile**
- Quick Sort: O(n log n) medio, **NON stabile**

ES 5.2 - Counting Sort Semplificato

Variante senza dati satellite (solo chiavi):

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

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

    // Ricostruisci array ordinato direttamente
    idx = 1
    for i = 0 to k-1
        while C[i] > 0
            A[idx] = i
            idx++
            C[i]--

    return A
```

Vantaggi: Risparmia spazio (no array B), opera in-place
Svantaggi: **NON stabile** (perde ordine relativo)

Quando usare:

- ✓ Usare Counting Sort quando k = O(n)
- ✓ Richiesta stabilità → versione completa
- ✗ Evitare quando k >> n (spreco memoria)

CATEGORIA 6: RADIX SORT

ES 6.1 - Radix Sort Completo

Problema: Ordinare n numeri interi con d cifre in base b.

Idea: Ordinare cifra per cifra usando algoritmo stabile (es. Counting Sort)

Due Approcci:

1. **LSD** (Least Significant Digit): dalla cifra meno significativa
2. **MSD** (Most Significant Digit): dalla cifra più significativa

Usiamo LSD (più comune):

Pseudocodice:

```
RadixSort(A, n, d, b)
    // A: array di n numeri
    // d: numero di cifre
    // b: base numerica (es. 10 per decimale)

    for i = 1 to d
        // Ordina A rispetto alla i-esima cifra
        // usando algoritmo STABILE (Counting Sort)
        CountingSortOnDigit(A, n, i, b)

    return A

CountingSortOnDigit(A, n, digit, b)
    // Counting Sort modificato per estrarre i-esima cifra
    allocate C[0..b-1]
    allocate B[1..n]

    // Conta occorrenze cifra i-esima
    for j = 0 to b-1
        C[j] = 0
    for j = 1 to n
        d = getDigit(A[j], digit, b)
        C[d]++
```

```
    // Cumulative
    for j = 1 to b-1
        C[j] = C[j-1] + C[j]

    // Posiziona (dal fondo per stabilità)
    for j = n downto 1
        d = getDigit(A[j], digit, b)
        B[C[d]] = A[j]
        C[d]--

    // Copia B in A
    for j = 1 to n
        A[j] = B[j]

getDigit(num, digit, base)
    // Estrae i-esima cifra (1 = meno significativa)
    return (num / base^(digit-1)) mod base
```

Correttezza - Invariante Principale:

Invariante: Dopo i iterazioni, array A è ordinato rispetto alle i cifre meno significative.

Dimostrazione per Induzione:

Base (i=1): Dopo prima iterazione, A ordinato per cifra meno significativa ✓

Passo Induttivo (i → i+1):

- **Ipotesi**: A ordinato rispetto a cifre 1..i
- Ordiniamo rispetto a cifra i+1 con algoritmo **stabile**
- Consideriamo due numeri x, y in A:

  Caso 1: cifra(i+1) di x < cifra(i+1) di y
  → Counting Sort posiziona x prima di y ✓

  Caso 2: cifra(i+1) di x > cifra(i+1) di y
  → Counting Sort posiziona x dopo y ✓

  Caso 3: cifra(i+1) di x = cifra(i+1) di y
  → Counting Sort **stabile** preserva ordine relativo
  → Per ipotesi induttiva, x e y erano ordinati per cifre 1..i
  → Ordine preservato ✓

Conclusione: Dopo d iterazioni, A ordinato per tutte le cifre → A ordinato ✓

La stabilità è CRUCIALE per la correttezza!

Esempio Esecuzione:

Input: A = [170, 045, 075, 090, 002, 024, 802, 066]
d = 3 cifre, b = 10 (decimale)

Iterazione 1 (cifra unità):
Ordina per: [0, 5, 5, 0, 2, 4, 2, 6]

```
A = [170, 090, 002, 802, 024, 045, 075, 066]
```

Iterazione 2 (cifra decine):
Ordina per: [7, 9, 0, 0, 2, 4, 7, 6]

```
A = [002, 802, 024, 045, 066, 170, 075, 090]
```

Iterazione 3 (cifra centinaia):
Ordina per: [0, 8, 0, 0, 0, 1, 0, 0]

```
A = [002, 024, 045, 066, 075, 090, 170, 802]
```

Output ordinato: [002, 024, 045, 066, 075, 090, 170, 802] ✓

Analisi Complessità:

- **Ogni iterazione**: O(n + b) per Counting Sort sulla cifra
- **d iterazioni**: d × O(n + b)
- **Totale**: **Θ(d(n + b))**

Quando è efficiente:

- Se d = O(1) e b = O(n) → Θ(n) **lineare**!
- Esempio: n numeri a 32-bit, base $b=2^{32}$ → d=1, tempo O(n)

Scelta della base b:

- Base piccola (b=2): più iterazioni (d grande)
- Base grande ($b=2^{32}$): meno iterazioni (d=1) ma Counting Sort più costoso
- **Ottimo**: b = Θ(n) → Θ(d) iterazioni di O(n)

Confronto con altri algoritmi:

Algoritmo	Tempo	Spazio	Stabile	Note
Comparison-based	Ω(n log n)	-	-	Limite inferiore
Counting Sort	O(n+k)	O(n+k)	✓	k deve essere O(n)
Radix Sort	O(d(n+b))	O(n+b)	✓	d cifre, base b

Algoritmo	Tempo	Spazio	Stabile	Note
Merge Sort	O(n log n)	O(n)	✓	Sempre O(n log n)

Vantaggi Radix Sort:

- ✓ Può essere O(n) se d costante
- ✓ Stabile
- ✓ Prevedibile (no caso peggiore)

Svantaggi:

- ✗ Funziona solo con chiavi numeriche/stringhe
- ✗ Richiede memoria aggiuntiva
- ✗ Costanti moltiplicative alte

ES 6.2 - Radix Sort per Stringhe

Applicazione: Ordinare stringhe di lunghezza fissa.

```
RadixSortStrings(S, n, m)
    // S: array di n stringhe
    // m: lunghezza fissa stringhe

    for i = m downto 1
        // Ordina per i-esimo carattere (stabile)
        CountingSortOnChar(S, n, i)

    return S
```

Esempio:

```
Input: ["DOG", "CAT", "BAT", "RAT"]
m = 3

Iter 1 (3° car): ["CAT", "BAT", "RAT", "DOG"]  // T,T,T,G
Iter 2 (2° car): ["CAT", "BAT", "RAT", "DOG"]  // A,A,A,O
Iter 3 (1° car): ["BAT", "CAT", "DOG", "RAT"]  // B,C,D,R

Output: ["BAT", "CAT", "DOG", "RAT"]
```

Complessità: Θ(m(n + Σ)) dove Σ = dimensione alfabeto

CHECKLIST ESAME

✓ MergeSort: O(n log n) tempo, O(n) spazio, stabile
✓ QuickSort: O(n log n) medio, O(n^2) peggiore, NON stabile
✓ QuickSelect: O(n) medio
✓ Ricerca binaria: O(log n) solo su array ordinati
✓ Partition posiziona pivot in posizione finale
✓ Tripartition utile con duplicati
✓ Counting Sort: O(n+k), stabile, richiede k=O(n)
✓ Radix Sort: O(d(n+b)), stabile, d cifre base b
✓ Stabilità: preserva ordine relativo elementi uguali
✓ Algoritmi stabili: Insertion, Merge, Counting, Radix
✓ Algoritmi NON stabili: Quick Sort, Heap Sort

ERRORI COMUNI DA EVITARE

❌ Usare ricerca binaria su array non ordinato
❌ Dimenticare caso base in ricorsione
❌ Confondere QuickSort con MergeSort (uno in-place, altro no)
❌ Pensare QuickSort sempre O(n log n) (peggiore O(n^2))
❌ Non considerare stabilità quando richiesta
❌ Usare Counting Sort con range k >> n (inefficiente)
❌ Dimenticare che Partition modifica l'array