📊 PANORAMICA ALGORITMI

Algoritmo	Caso Peggiore	Caso Medio	Caso Migliore	Spazio	Stabile	In-place
InsertionSort	Θ(n²)	Θ(n²)	Θ(n)	O(1)	✓	✓
MergeSort	Θ(n log n)	Θ(n log n)	Θ(n log n)	O(n)	✓	✗
QuickSort	Θ(n²)	Θ(n log n)	Θ(n log n)	O(log n)	✗	✓
HeapSort	Θ(n log n)	Θ(n log n)	Θ(n log n)	O(1)	✗	✓
CountingSort	Θ(n+k)	Θ(n+k)	Θ(n+k)	O(n+k)	✓	✗
RadixSort	Θ(d(n+k))	Θ(d(n+k))	Θ(d(n+k))	O(n+k)	✓	✗

Legenda:

- **k** = range valori (CountingSort)
- **d** = numero di cifre (RadixSort)
- **Stabile** = mantiene ordine relativo elementi uguali
- **In-place** = usa O(1) spazio aggiuntivo

🔵 INSERTION SORT (Incrementale)

Idea

Come ordinare carte da gioco: inserisci ogni elemento nella posizione corretta tra quelli già ordinati.

Pseudocodice

```
InsertionSort(A, n):
    for j = 2 to n:
        key = A[j]
        i = j - 1
        while i > 0 AND A[i] > key:
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key
```

Invariante di Ciclo

Invariante: All'inizio di ogni iterazione del ciclo for, il sottoarray `A[1..j-1]` contiene gli elementi originali di `A[1..j-1]` ma in ordine crescente.

Dimostrazione:

- **Inizializzazione:** j=2 → A[1..1] è banalmente ordinato
- **Conservazione:** Se A[1..j-1] ordinato, inserendo A[j] correttamente → A[1..j] ordinato
- **Terminazione:** j=n+1 → A[1..n] ordinato ✓

Complessità

Caso migliore (array già ordinato): Θ(n)

- Ciclo while mai eseguito
- Solo n-1 confronti

Caso peggiore (array ordinato decrescente): Θ(n²)

```
T(n) = Σ(j=2 to n) j = 2+3+...+n = n(n+1)/2 - 1 = θ(n²)
```

Caso medio: Θ(n²)

- Mediamente metà degli elementi del sottoarray ordinato vengono spostati

Quando Usarlo

- Array piccoli (n < 50)
- Array quasi ordinati
- Come ottimizzazione di QuickSort per sottoproblemi piccoli

🔵 MERGE SORT (Divide et Impera)

Idea

1. **Divide:** Dividi array in due metà
2. **Impera:** Ordina ricorsivamente le due metà
3. **Combina:** Fondi le due metà ordinate

Pseudocodice

```
MergeSort(A, p, r):
    if p < r:
        q = ⌊(p+r)/2⌋
        MergeSort(A, p, q)
```

```
        MergeSort(A, q+1, r)
        Merge(A, p, q, r)

Merge(A, p, q, r):
    n1 = q - p + 1
    n2 = r - q
    L[1..n1+1], R[1..n2+1]

    for i = 1 to n1:
        L[i] = A[p+i-1]
    for j = 1 to n2:
        R[j] = A[q+j]

    L[n1+1] = ∞
    R[n2+1] = ∞

    i = 1
    j = 1
    for k = p to r:
        if L[i] ≤ R[j]:
            A[k] = L[i]
            i = i + 1
        else:
            A[k] = R[j]
            j = j + 1
```

Correttezza (Invariante Merge)

Invariante: All'inizio di ogni iterazione del ciclo for, il sottoarray `A[p..k-1]` contiene i k-p elementi più piccoli di L e R in ordine crescente, e `L[i]` e `R[j]` sono i più piccoli elementi non ancora copiati.

Complessità

Ricorrenza: $T(n) = 2T(n/2) + \Theta(n)$

Soluzione (Master Theorem, caso 2):

- a=2, b=2, f(n)=Θ(n)
- n^($\log_2 2$) = n^1 = n = f(n)
- **T(n) = Θ(n log n)** per TUTTI i casi

Ottimizzazioni

1. **Allocazione singola:** Allocare L e R una sola volta all'inizio
2. **Eliminare sentinelle:** Verificare fine array esplicitamente
3. **Ibrido con InsertionSort:** Per sottoarray piccoli (k < 50), usare InsertionSort

---

🔵 QUICK SORT (Divide et Impera + Randomizzato)

Idea

1. **Partition:** Scegli pivot, riorganizza array in ≤pivot e >pivot
2. **Divide:** Ordina ricorsivamente le due parti

Pseudocodice

```
QuickSort(A, p, r):
    if p < r:
        q = Partition(A, p, r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)

Partition(A, p, r):
    x = A[r]  // pivot
    i = p - 1
    for j = p to r-1:
        if A[j] ≤ x:
            i = i + 1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
    return i + 1
```

Invariante Partition

All'inizio di ogni iterazione del ciclo for:

1. A[p..i] ≤ x (pivot)
2. A[i+1..j-1] > x
3. A[r] = x

Versione Randomizzata (Preferibile)

```
RandomizedPartition(A, p, r):
    i = Random(p, r)
    swap(A[i], A[r])
    return Partition(A, p, r)

RandomizedQuickSort(A, p, r):
    if p < r:
        q = RandomizedPartition(A, p, r)
        RandomizedQuickSort(A, p, q-1)
        RandomizedQuickSort(A, q+1, r)
```

Complessità

Caso peggiore: $\Theta(n^2)$

- Partition sempre sbilanciato (es. array già ordinato)
- $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Caso migliore: $\Theta(n \log n)$

- Partition sempre bilanciato
- $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

Caso medio (versione randomizzata): $\Theta(n \log n)$

- Dimostrazione complessa basata su albero delle ricorrenze
- Profondità attesa = $O(\log n)$
- Lavoro per livello = $O(n)$

Ottimizzazione: 3-Way Partition (Tripartition)

Per gestire molti duplicati:

```
Partition3Way(A, p, r):
    x = A[r]
    i = p - 1  // fine elementi < x
    j = r      // inizio elementi > x
    k = p      // elemento corrente

    while k < j:
        if A[k] < x:
            i = i + 1
            swap(A[i], A[k])
            k = k + 1
        else if A[k] > x:
            j = j - 1
            swap(A[k], A[j])
        else:
            k = k + 1

    swap(A[j], A[r])
    return (i+1, j)  // restituisce coppia di indici
```

🔵 HEAP SORT

Vedi file `04_alberi_heap_ABR_RBTree.md` per dettagli su Heap.

Pseudocodice

```
HeapSort(A, n):
    BuildMaxHeap(A, n)
    for i = n downto 2:
        swap(A[1], A[i])
        heapsize = heapsize - 1
        MaxHeapify(A, 1)
```

Complessità

- **BuildMaxHeap:** O(n)
- **n-1 chiamate a MaxHeapify:** O((n-1) log n)
- **Totale:** Θ(n log n) per TUTTI i casi

Proprietà

- **In-place:** O(1) spazio aggiuntivo
- **NON stabile:** L'ordine relativo può cambiare
- **Deterministico:** Nessuna randomizzazione

🟢 COUNTING SORT (Non basato su confronti)

Idea

Per array con valori interi in range [0..k]:

1. Conta occorrenze di ogni valore
2. Calcola posizioni finali
3. Posiziona elementi

Pseudocodice

```
CountingSort(A, B, n, k):
    // A: input, B: output, k: max valore
    C[0..k] = array di zeri

    // Conta occorrenze
    for j = 1 to n:
        C[A[j]] = C[A[j]] + 1

    // Posizioni cumulative
    for i = 1 to k:
        C[i] = C[i] + C[i-1]

    // Posiziona elementi
```

```
    for j = n downto 1:
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

Complessità

- **Tempo:** Θ(n + k)
- **Spazio:** O(n + k)

Quando Usarlo

- k = O(n) → tempo Θ(n)
- Valori interi in range piccolo
- Come subroutine di RadixSort

Proprietà Importante

Stabile: Mantiene ordine relativo (scorrimento da n a 1)

🟢 RADIX SORT (Non basato su confronti)

Idea

Ordina numeri di d cifre ordinando dalla cifra meno significativa (LSD) a quella più significativa.

Prerequisito

Algoritmo di ordinamento stabile per le singole cifre (es. CountingSort).

Pseudocodice

```
RadixSort(A, n, d):
    for i = 1 to d:
        // Usa algoritmo stabile per ordinare su cifra i
        CountingSortOnDigit(A, n, i)
```

Complessità

- d cifre, ogni cifra in base k
- CountingSort per cifra: Θ(n + k)
- **Totale: Θ(d(n + k))**

Ottimizzazione

Se numeri sono b-bit e ordiniamo r bit alla volta:

- d = ⌈b/r⌉ passate
- k = 2^r possibili valori per cifra
- **Tempo:** Θ((b/r)(n + 2^r))

Ottimo r: r = ⌈log n⌉ → Θ(bn/log n)

Esempio

```
Numeri a 3 cifre decimali (d=3, k=10):
Input:  329, 457, 657, 839, 436, 720, 355

Dopo digit 1: 720, 355, 436, 457, 657, 329, 839
Dopo digit 2: 720, 329, 436, 839, 355, 457, 657
Dopo digit 3: 329, 355, 436, 457, 657, 720, 839
```

📐 LIMITI INFERIORI

Ordinamento Basato su Confronti

Teorema: Qualunque algoritmo di ordinamento basato solo su confronti richiede Ω(n log n) confronti nel caso peggiore.

Dimostrazione (Albero delle Decisioni):

- n! possibili permutazioni (foglie dell'albero)
- Altezza minima albero binario con n! foglie: $h \geq \log_2(n!)$
- $\log_2(n!) = \Theta(n \log n)$ (formula di Stirling)
- **Quindi:** h = Ω(n log n) confronti necessari

Scambi di Elementi Contigui

Teorema: Ordinare con solo scambi di elementi adiacenti richiede $\Omega(n^2)$ scambi nel caso peggiore.

Dimostrazione:

- **Inversione:** coppia (i,j) con i < j e A[i] > A[j]
- Caso peggiore: n(n-1)/2 inversioni (array decrescente)
- Ogni scambio adiacente rimuove al più 1 inversione
- **Quindi:** $\Omega(n^2)$ scambi necessari

🎯 SCELTA ALGORITMO - GUIDA PRATICA

Usa InsertionSort se:

- n < 50
- Array quasi ordinato
- Serve algoritmo stabile e in-place

Usa MergeSort se:

- Serve garantire Θ(n log n) sempre
- Serve algoritmo stabile
- Spazio O(n) accettabile
- Dati su disco (external sorting)

Usa QuickSort se:

- Serve velocità in pratica (costanti migliori)
- Caso medio Θ(n log n) sufficiente
- Randomizzazione accettabile
- In-place desiderato

Usa HeapSort se:

- Serve garantire Θ(n log n) sempre
- Serve in-place (O(1) spazio)
- Stabilità non necessaria

Usa CountingSort se:

- Valori interi in range piccolo (k = O(n))
- Serve Θ(n)
- Spazio O(n+k) accettabile

Usa RadixSort se:

- Numeri con cifre fisse
- Serve Θ(n) (quando d piccolo)
- Serve algoritmo stabile

💡 TRUCCHI PER L'ESAME

Correttezza con Invarianti

1. Identifica invariante di ciclo
2. Dimostra: Inizializzazione, Conservazione, Terminazione
3. Invariante deve essere vero PRIMA di ogni iterazione

Complessità

- **InsertionSort:** conta operazioni nel ciclo interno
- **Divide et Impera:** scrivi ricorrenza + Master Theorem
- **QuickSort:** caso medio richiede analisi probabilistica

Stabilità

Stabile: InsertionSort, MergeSort, CountingSort, RadixSort **NON stabile:** QuickSort, HeapSort

Common Mistakes

- ❌ Confondere caso migliore con caso medio
- ❌ Dimenticare il costo di BuildMaxHeap è O(n)
- ❌ Non verificare condizione di regolarità in Master Theorem caso 3
- ❌ Assumere QuickSort sempre Θ(n log n) (è caso medio, non peggiore!)