

Algoritmi e Strutture Dati

Autore: *Antonio Iovine*





April 8, 2025

Contents

1	Introduzione	3
2	Concetti Fondamentali	3
2.1	Cos'è un Algoritmo?	3
2.2	Complessità	3
2.3	O grande	4
2.3.1	Dimostrazione:	5
2.4	Il caso migliore Ω	5
2.4.1	Dimostrazione:	5
2.5	Il caso medio Θ	5
3	Algoritmi di Sorting	5
3.1	Bubble Sort	6
3.1.1	Codice:	6
3.2	Insertion Sort	6
3.2.1	Codice	6
3.3	Selection Sort	7
3.3.1	Codice	7
3.4	Problema 1	8
4	Equazioni di Ricorrenza	8
4.1	Fattoriale	8
4.1.1	Prova inversa e tecniche avanzate	9
4.2	Master Theorem	10
5	Algoritmi di Sorting Ricorsivi	11
5.1	Merge Sort	11
5.1.1	Codice	11
5.1.2	Equazione di Ricorrenza	13
5.2	Quick Sort	13
5.2.1	Codice	14
5.2.2	Equazione di Ricorrenza	14
5.3	Quick Sort Randomized	14

6	Introduzione alle Strutture Dati: Heap	15
6.1	proprietà di un Heap	15
6.1.1	Proprietà derivate: Alberi completi	15
6.2	Heap Sort	16
6.2.1	Codice	16
6.3	Problema 2: Limiti inferiori	17
7	Algoritmi di Ordinamento $\in O(n)$	17
7.1	Counting Sort	18
7.1.1	Codice	18
7.2	Radix Sort	18
7.2.1	Codice	19
7.3	Bucket Sort	19
7.3.1	Codice	19
8	Problema della selezione	20
9	Selezione del massimo e del minimo	21
9.1	Select Min/Max	21
9.1.1	Codice	21
9.1.2	Trovare il minimo e il massimo in meno di $O(n)$	21
9.2	Esempio di Immagine	21

Legenda colori

-  Teoremi / Definizioni / Dimostrazioni
-  Esempi
-  Esercizi
-  Codice

1 Introduzione

Questo PDF vuole essere una spiegazione più o meno dettagliata di alcuni concetti fondamentali riguardanti gli algoritmi. Prima di iniziare è sempre bene fare una premessa, un questo PDF userò principalmente la Java notation quindi sia codice che notazione per i grafi saranno in Java. quindi si avrà un codice simil java e si preferirà una notazione del tipo $u.d$ piuttosto che $d[u]$.

Ovviamente l'intento principale non è quello di capire il codice ma il ragionare sul perchè funzioni e come funzioni. Ergo se non si conoscesse java non è un problema.

2 Concetti Fondamentali

2.1 Cos'è un Algoritmo?

Un algoritmo è una sequenza finita di passi ben definiti per risolvere un problema, o almeno è la definizione formale, ma poche menate, la domanda che ci poniamo ora è cosa ci faccio con un algoritmo?

La risposta è: un po' quel che voglio, se riesco a capire concretamente come funziona un algoritmo posso applicarlo come più mi fa comodo, riesco a decidere quel è il migliore in base alla situazione, e soprattutto, cosa non banale, se quel problema è risolvibile o meno. e da qui vi è la prossima domanda, come faccio a capire quale algoritmo è il migliore per il mio problema?

2.2 Complessità

Per dire che un algoritmo è migliore di un altro bisogna prima confrontarli, ma come fare? beh usando la complessità.

la complessità è uno strumento che ci permette di confrontare due algoritmi in base al tempo e allo spazio che impiegano per risolvere un problema. la complessità si divide in due parti:

- Complessità temporale: indica il tempo che un algoritmo impiega per risolvere un problema.
- Complessità spaziale: indica lo spazio che un algoritmo impiega per risolvere un problema.

La complessità temporale però, non viene espressa in secondi, bensì in numero di operazioni elementari che un algoritmo deve compiere per risolvere un problema. Perché? perchè il tempo di esecuzione di un algoritmo dipende da molti fattori, come la macchina su cui viene eseguito, il linguaggio di programmazione, ecc. Quindi per confrontare due algoritmi è meglio usare il numero di operazioni elementari.

Ma se avessi un algoritmo che riordina i numeri naturali da 1 a n in ordine crescente, ma sono fortunato e me li danno già riordinati, cosa faccio con la complessità?

per questo esistono tre notazioni principali:

- O : indica il caso peggiore

- Ω : indica il caso migliore
- Θ : indica il caso medio

Il caso più importante è quello peggiore, data anche la legge di Murphy.

Esempio: Moltiplicazioni di matrici/vettori,

Supponiamo di avere due vettori, quindi matrici monodimensionali, di lunghezza n , e vogliamo eseguire una moltiplicazione tra le due, data una classica moltiplicazione tra vettori è banale dire che la complessità sia $O(n)$, moltiplico l'elemento $V_i \cdot W_i$ e sommo il risultato, ma se volessi fare una moltiplicazione tra matrici?

Codice: Definisco due matrici A e B di dimensione $n \times m$ e $m \times l$.

```

1  i, j, k = 1
2  for( i → n ) {
3      for( j → l ) {
4          C[i][j] = 0;
5          for( k → m ) {
6              C[i][j] += A[i][k] * B[k][j];
7          }
8      }
9  }
```

Lo so è tnto, ma andando per gradi si riesce a fare qualsiasi cosa. Parto dal for a riga 5, verrà eseguito un numero di volte pari a m poichè vado da $k = 1$ a m (raggiungo già in ordini di grandezza, altrimenti sarebbe $5m + 1$), Ora guardo il for a riga 3 il quale verrà eseguito l volte, e infine il for a riga 1 che verrà eseguito n volte. quindi la complessità totale sarà $O(n \cdot m \cdot l)$.

2.3 O grande

Mi dispiace ma un pò di matematica serve sempre, questa è la definizione formale di O grande:

$$f \in O(g)$$

$$\exists_{c>0} \exists_{n_0} \forall_{n \geq \bar{n}} \quad f(n) \leq c \cdot g(n)$$

Tutto questo per dire che dopo una certa \bar{n} la funzione g sarà sempre più grande di f , e quindi posso dire che f è $O(g)$.

Cosa significa? prendendo l'esempio della moltiplicazione tra matrici, la complessità è $O(n \cdot m \cdot l)$, ma se dicessi che la complessità fosse $O(2 \cdot n \cdot m \cdot l)$ sarebbe comunque corretto, impreciso, ma corretto, ecco perchè se la mia funzione g è maggiore della mia f allora si può dire che f è $O(g)$.

2.3.1 Dimostrazione:

dimostro che se $f_1 \in O(g)$ e $f_2 \in O(g)$ allora $f_1 + f_2 \in O(g)$

- $\forall_{n > \bar{n}} \quad f_1(n) \leq c^1 \cdot g(n)$
 - $\forall_{n > \bar{n}} \quad f_2(n) \leq c^2 \cdot g(n)$
- $$f_1(n) + f_2(n) \leq c \cdot g(n)$$

prendendo: $c = (c^1 + c^2)$ e $\bar{n} = \max(\bar{n}_1, \bar{n}_2)$

Come esercizio avanzato si può dimostrare che $f_1 \cdot f_2 \in O(g)$

2.4 Il caso migliore Ω

Come per l'O grande anche il caso migliore ha una definizione formale:

$f \in \Omega(g)$

$$\cdot \exists_{c > 0} \exists_{n_0} \forall_{n \geq \bar{n}} \quad f(n) \geq c \cdot g(n)$$

2.4.1 Dimostrazione:

Vorrei dimostrare che $f \in O(g) \leftrightarrow g \in \Omega(f)$

- $\forall_{n > \bar{n}} \quad f(n) \geq c \cdot g(n)$

$$\frac{f(n)}{c} \leq g(n) = \frac{1}{c} \cdot f(n) \leq g(n) = c' \cdot f(n) \leq g(n) \quad \text{con } c' = \frac{1}{c}$$

2.5 Il caso medio Θ

Per Θ la definizione è un pò diversa:

Supponiamo di avere un algoritmo tale che $A \in O(f)$ e $A \in \Omega(f)$ allora $A \in \Theta(f)$, Cosa significa? che la complessità di A è sia minore che maggiore di f, quindi è uguale a f, cioè il mio algoritmo è il migliore possibile per quel problema. (salvo alcune eccezioni).

3 Algoritmi di Sorting

Di cosa si sta parlando? algoritmi i quali riescono a ordinare (input) una sequenza di oggetti su cui è definita una relazione d'ordinamento, in una (output) permutazione di quella sequenza.

Definizione: Si dice **in loco** un algoritmo di ordinamento che non richiede spazio extra per l'ordinamento, ma utilizza solo lo spazio dell'array di input.

Definizione: Si dice **stabile** un algoritmo di ordinamento che mantiene l'ordine relativo degli elementi uguali.

3.1 Bubble Sort

Il Bubble Sort è un algoritmo di ordinamento molto semplice, ma anche molto inefficiente. L'idea di base è quella di scorrere l'array da sinistra a destra, confrontando due elementi adiacenti e scambiandoli se non sono ordinati. Questo processo viene ripetuto fino a quando non ci sono più scambi da fare.

3.1.1 Codice:

```
1 void bubbleSort(int[] A) {  
2     int n = A.length;  
3     boolean swapped;  
4     do {  
5         swapped = false;  
6         for (int i = 1; i < n; i++) {  
7             if (A[i - 1] > A[i]) {  
8                 int temp = A[i - 1];  
9                 A[i - 1] = A[i];  
10                A[i] = temp;  
11                swapped = true;  
12            }  
13        }  
14    } while (swapped);  
15 }
```

Complessità:

- La complessità temporale del Bubble Sort è $O(n^2)$ nel caso peggiore.
- La complessità spaziale è $O(1)$ costante, essendo in-place. (non richiede spazio extra)

3.2 Insertion Sort

Come il bubbleSort anche l'insertionSort è un algoritmo di ordinamento molto semplice, ma più efficiente del bubbleSort.

Perché usare un algoritmo inefficiente quando ne esiste uno più efficiente? Semplice, l'insertion ha delle proprietà che lo rendono molto utile e a volte molto più efficiente di algoritmi che hanno complessità molto minore

L'idea di base è quella di mantenere una parte dell'array ordinata e una parte disordinata. Ad ogni passo, si prende un elemento dalla parte disordinata e lo si inserisce nella parte ordinata, spostando gli elementi più grandi di esso a destra.

3.2.1 Codice

```

1 void insertionSort(int[] A) {
2     int n = A.length;
3     for (int i = 1; i < n; i++) {
4         int key = A[i];
5         int j = i - 1;
6         while (j >= 0 && A[j] > key) {
7             A[j + 1] = A[j];
8             j--;
9         }
10        A[j + 1] = key;
11    }
12 }

```

L'Insertion Sort ha delle proprietà che lo rendono molto utile in specifici contesti. Una di queste proprietà è che se l'array è già ordinato, la complessità temporale sarà $O(n)$, rendendolo molto efficiente per array quasi ordinati.

Inoltre, questo algoritmo è stabile, il che significa che l'ordine dei duplicati non cambia, preservando l'ordine relativo degli elementi uguali.

Infine, l'Insertion Sort è molto efficiente per ordinare piccole sequenze di numeri, rendendolo una scelta eccellente per piccoli dataset o come parte di algoritmi più complessi.

Complessità:

- La complessità temporale dell'Insertion Sort è $O(n^2)$ nel caso peggiore . $O(n)$ se ordinato
- La complessità spaziale è $O(1)$ costante, essendo in-place.
- Algoritmo è stabile

3.3 Selection Sort

Qui il concetto alla base è molto semplice: si cerca il minimo elemento dell'array e lo si scambia con il primo elemento. Poi si cerca il minimo elemento del sottoarray rimanente e lo si scambia con il secondo elemento, e così via.

3.3.1 Codice

```

1 void selectionSort(int[] A) {
2     int n = A.length;
3     for (int i = 0; i < n - 1; i++) {
4         int minIndex = i;
5         for (int j = i + 1; j < n; j++) {
6             if (A[j] < A[minIndex]) {
7                 minIndex = j;
8             }
9         }
10        int temp = A[i];
11        A[i] = A[minIndex];

```

```

12     A[minIndex] = temp;
13 }
14 }

```

Complessità:

- La complessità temporale del Selection Sort è $O(n^2)$ nel caso peggiore.
- La complessità spaziale è $O(1)$ costante, essendo in-place.
- Algoritmo non è stabile

3.4 Problema 1

Luca e Paolo chiacchierano del più e del meno, quando a un certo punto Luca chiede:

“Quanti anni hanno i tuoi tre figli?”

Paolo, un po’ infastidito, risponde:

“Il prodotto delle loro età è 36.”

Luca ci pensa su, ma dopo un attimo dice perplesso:

“Mi serve un altro indizio.”

Paolo, sempre più seccato, aggiunge:

“La somma delle loro età è uguale al numero civico di questa casa.”

Luca riflette ancora, ma scuote la testa:

“Non mi basta, dammi un altro indizio!”

Paolo, ormai rassegnato, dice:

“Va bene... il più grande ha gli occhi azzurri.”

Luca rimane in silenzio per qualche secondo, poi annuisce e sorride:

“Ah, ora ho capito quanti anni hanno!”

Questo problema sembra complesso, ma con un pò di ragionamento laterale, si riesce a risolvere. (Soluzione a fine PDF)

4 Equazioni di Ricorrenza

Definizione: Un’equazione di ricorrenza è un’equazione che definisce una funzione in termini di se stessa.

Sono un modo per descrivere la complessità di un algoritmo ricorsivo, ad esempio merge sort. Sono utili per capire quanto tempo e spazio richiede un algoritmo e per confrontare algoritmi diversi.

Poniamo ora il problema di calcolare il fattoriale dei primi n numeri naturali, quindi $f(n) = n!$.

4.1 Fattoriale

```

1 int fattoriale(int n) {
2     if (n == 0) return 1;
3     else return n * fattoriale(n - 1);

```


Questo codice, se pur semplice, ha una complessità temporale non banale da calcolare. Per fortuna esistono l'equazioni di ricorrenza per facilitarci il lavoro e rendere il tutto semplice,

- Iniziamo con esprimere la funzione fattoriale senza averla svolta neanche una volta $\Leftarrow T(n)$
- Una volta averla svolta una volta avremo $T(n-1) + 1$ cioè le volte che devo eseguire la funzione + le volte che ho eseguito la funzione
- Andando avanti $T(n-2) + 1 + 1$ e così via fino ad arrivare a $T(0)$
- Quando avremmo eseguito la funzione n volte avremmo $T(0) + n$ con $T(0) = 1$ quindi $T(n) = T(n-1) + 1 = T(n-2) + 2 = \dots = n + 1$

* $T(0) = 1$ è dato dal caso base della funzione

Concludendo, anche se ora sembra banale dire che la funzione $\in O(n+1)$ che per ordini di grandezza è $O(n)$, ma non è sempre così semplice, ecco perché esistono le equazioni di ricorrenza.

4.1.1 Prova inversa e tecniche avanzate

Sia $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n \in O(n \cdot \log(n))$

Con:

$$T(n) = \begin{cases} \text{costante} & \text{se } n < a \\ 2T(\lfloor \frac{n}{2} \rfloor) + n & \text{se } n \geq a \end{cases}$$

Provare che la funzione appartenga a $O(n \cdot \log(n))$ è un po' più complesso, ma non impossibile.

Iniziamo col dire che affermare che qualcosa appartenga a $O(n \cdot \log(n))$ significa dire che $T(n) \leq c \cdot n \cdot \log(n)$ *Definizione di O grande 2.3

da qui possiamo iniziare a sostituire la funzione $T(n)$ con la sua definizione, quindi:

$$\begin{aligned} T(n) &\leq 2c \lfloor \frac{n}{2} \rfloor \cdot \log(\lfloor \frac{n}{2} \rfloor) + n \\ &\leq c \cdot n \cdot \log(\frac{n}{2}) + n \\ &\leq c \cdot n \cdot \log(n) - c \cdot n \cdot \log(2) + n \quad \text{se } n - c \cdot n \cdot \log(2) \leq 0 \\ &\leq c \cdot n \cdot \log(2) \end{aligned}$$

Questo è vero se e solo se:

$$\begin{aligned} 0 &\geq n - c \cdot n \cdot \log(2) \\ c &\geq \frac{n}{n \cdot \log(2)} \\ c &\geq \frac{1}{\log(2)} \end{aligned}$$

Il metodo della prova inversa o della sostituzione valida l'ipotesi quando esistono dei valori di c che validano la disequazione che ho costruito.

Ad esempio, Sia $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + 1 \in O(n)$

$$\begin{aligned} T(n) &\leq c \lfloor \frac{n}{2} \rfloor + c \lceil \frac{n}{2} \rceil + 1 \\ &\leq c(\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil) + 1 \\ &\leq c \cdot n + 1 \end{aligned}$$

Essendo che: $c \cdot n + 1 \not\leq c \cdot n$ questa funzione non appartiene a $O(n)$.

Anche se potrebbe appartenere a $O(n \pm b)$ con b costante di ordine inferiore,

$$\begin{aligned} T(n) &\leq c \lfloor \frac{n}{2} \rfloor - b + c \lceil \frac{n}{2} \rceil - b + 1 \\ &\leq c(\lfloor \frac{n}{2} \rfloor + \lceil \frac{n}{2} \rceil) + 1 - 2b \\ &\leq c \cdot n + 1 - 2b \end{aligned}$$

In questo caso abbiamo che $c \cdot n + 1 - 2b \leq c \cdot n - b$ è verificata quando $b \geq 1$ perché $1 - b \geq 0$

così abbiamo dimostrato che questa funzione appartiene a $O(n - b)$ che, per ordine di grandezza, appartiene a $O(n)$.

Ovviamente non è sempre possibile usare questo trucchetto, a volte la supposizione iniziale potrebbe essere semplicemente sbagliata.

4.2 Master Theorem

Se prendessimo un'equazione di ricorrenza del tipo: $T(n) = a \cdot T(\frac{n}{b}) + f(n)$ potremmo risolverla considerando i seguenti casi:

- se $f(n) \in O(n^{\log_b a - \epsilon})$ allora $T(n) \in \Theta(n^{\log_b a})$
- se $f(n) \in \Theta(n^{\log_b a})$ allora $T(n) \in \Theta(f(n) \cdot \log(n))$
- se $f(n) \in \Omega(n^{\log_b a + \epsilon})$ allora $T(n) \in \Theta(f(n))$

Esempio: $T(n) = 9T(\frac{n}{3}) + n$

- $a = 9$
- $b = 3$
- $f(n) = n$

Quindi si avrà che $n^{\log_b(a)} = n^{\log_3(9)} = n^2$ $n \in O(n^{2-\epsilon}) \xrightarrow[3^\circ \text{ caso}]{} T(n) \in \Theta(n^2)$

1) Esercizi:

1. $T(n) = T(n-1) + n \in O(n^2)$
2. $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$
3. $T(n) = 4T(\frac{n}{2}) + n^3$
4. $T(n) = 4T(\frac{n}{2}) + n^{\log_2(4)}$

Soluzione a fine PDF

5 Algoritmi di Sorting Ricorsivi

Ora che abbiamo capito come studiare la complessità di un algoritmo ricorsivo, possiamo anche studiare gli algoritmi di sorting ricorsivi.

In genere questo tipo di algoritmi sono più complessi da implementare, ma più efficienti rispetto agli algoritmi iterativi.

Infatti, storicamente sono stati molto disprezzati poiché riempivano le memorie con i loro stack, ma ora che le memorie sono enormi e gli algoritmi iterativi sono stati superati in efficienza, sono tornati di moda.

5.1 Merge Sort

Ed ecco che uno di quei algoritmi ricorsivi super di gran lunga la complessità di un algoritmo iterativo, al modico prezzo di un po di spazio in memoria per le chiamate ricorsive, il Merge Sort.

L'idea di base è quella di dividere l'array in due metà, ordinare ricorsivamente ciascuna metà e poi unire le due metà ordinate.

difficile da visualizzare? beh non è un problema, basta pensare a un mazzo di carte, lo divido in due mazzi, ordino i due mazzi e poi unisco i due mazzi ordinati.

5.1.1 Codice

```
1 void mergeSort(int[] A, int left, int right) {
2     if(left < right) {
3         int mid = (left + right) / 2; //per difetto
4         mergeSort(A, left, mid);
5         marge(A, left, mid, right);
6     }
7 }
```

```
1 void marge(int[] A, int left, int mid, int right){
2     int i = 1;
3     int j = left + 1;
4     int k = mid + 1;
```

```

5      int[] B = new int[right - left + 1];
6      while(j <= mid || k <= right) {
7          if( j <= mid && (k < right || A[j] < A[k])) {
8              B[i] = A[j];
9              j++;
10         } else {
11             B[i] = A[k];
12             k++;
13         }
14         i++;
15     }
16 }

```

Ok, so che sembra tanto, ma non è così complesso come sembra, basta capire il concetto di base e il resto viene da se.

La funzione mergeSort è la funzione principale che divide l'array in due metà e chiama ricorsivamente se stessa per ordinare ciascuna metà. La funzione merge è quella che unisce le due metà ordinate.

La funzione merge prende tre argomenti: l'array da ordinare, gli indici di inizio e fine dell'array. La funzione merge crea un array temporaneo B per memorizzare gli elementi ordinati e poi copia gli elementi ordinati nell'array originale.

Alla fine avremmo l'array B ordinato, e lo copiamo nell'array originale, oppure ritorniamo semplicemente B

Complessità:

- La complessità temporale del Merge Sort è $O(n \cdot \log(n))$ nel caso peggiore.
- La complessità spaziale è $O(n)$, poichè richiede spazio extra per l'array temporaneo.
- Algoritmo non è stabile, ma può essere reso stabile con piccole modifiche.

5.1.2 Equazione di Ricorrenza

L'equazione di ricorrenza per il Merge Sort è: $T(n) = 2T(\frac{n}{2}) + n$

il $2T(\frac{n}{2})$ rappresenta la divisione in due metà, e il fatto che su tutte e due le metà chiamo il mergeSort, mentre il n rappresenta il costo di unione delle due metà ordinate.

La soluzione di questa equazione di ricorrenza è $T(n) \in O(n \cdot \log(n))$ poichè rientra nel caso 2 del teorema dell'esperto, se si vuole fare la contro prova basti usare l'altro metodo e assumere che $O(n \log n)$ sia la complessità su cui fare la prova.

*Se ti stai chiedendo se copiando l'array B nell'array originale si avrebbe un costo maggiore, ti consiglio di andare a rileggere la definizione di ordine di grandezza, poichè la copia ha complessità $O(n)$, la complessità finale sarebbe $O(2n \cdot \log(n))$ che, per ordine di grandezza ritornerebbe ad essere $O(n \cdot \log(n))$

5.2 Quick Sort

Un altro algoritmo ricorsivo molto efficiente è il Quick Sort, molto efficiente ma anche molto inefficiente, poichè nel caso peggiore la sua complessità si alza fino a $O(n^2)$, questo dipende principalmente da come scegliamo di partizionare l'array, ma andando per gradi.

L'idea di base è quella di scegliere un elemento come pivot e partizionare l'array in due sottoarray: uno contenente gli elementi minori del pivot e l'altro contenente gli elementi maggiori del pivot. Poi si ordina ricorsivamente ciascun sottoarray. Alla fine, si ottiene un array ordinato unendo i due sottoarray ordinati con il pivot.

5.2.1 Codice

```
1 void quickSort(int[] A, int left, int right) {  
2     if(left < right){  
3         int pivot = partition(A, left, right); //per  
4             difetto  
5         quickSort(A, left, pivot)  
6         quickSort(A, pivot + 1, right);  
7     }  
}
```

```
1 int partition(int[] A, int left, int right) {  
2     int pivot = A[left];  
3     int i = left - 1;  
4     int j = right + 1;  
5     while(true) {  
6         repeat j-- until A[j] <= pivot;  
7         repeat i++ until A[i] >= pivot;  
8         if(i < j)  
9             swap(A[i], A[j]); // O(1) scambia  
10                elementi  
11     else  
12     return j;  
}
```

Complessità:

- La complessità temporale del Quick Sort è $O(n \cdot \log(n))$ nel caso medio e $O(n^2)$ nel caso peggiore.
- La complessità spaziale è $O(\log(n))$ per lo stack delle chiamate ricorsive.
- Algoritmo non è stabile, ma può essere reso stabile con modifiche.

5.2.2 Equazione di Ricorrenza

L'equazione di ricorrenza per il Quick Sort è: $T(n) = T(k) + T(n - k - 1) + n$, dove k è il numero di elementi nel primo sottoarray. Nel caso medio, $k \approx n/2$, quindi $T(n) = 2T(n/2) + n$, che si risolve in $O(n \cdot \log(n))$. Nel caso peggiore, $k = 0$ o $k = n - 1$, quindi $T(n) = T(n - 1) + n$, che si risolve in $O(n^2)$.

5.3 Quick Sort Randomized

Per rendere meno probabile il caso peggiore si potrebbe aggiungere un fattore randomico alla scelta della partizione, ad esempio:

```
1 void randomizedPartitio(int[] A, int left  
    , int right) {
```

```

2         int pivot = random.nextInt(left,
3             right);
4         swap(A[pivotI], A[left]);
5         return partition(A, left, right);
    }

```

In questo modo, la scelta del pivot è casuale e quindi il caso peggiore è meno probabile. La complessità rimane $O(n \cdot \log(n))$ nel caso medio e $O(n^2)$ nel caso peggiore, ma la probabilità di raggiungere il caso peggiore è molto bassa.

6 Introduzione alle Strutture Dati: Heap

Partiamo col dire che confrontando ogni elemento di un array non potremmo mai avere una complessità inferiore a $\Theta(n)$, per abbassare ancora di più quest'asticella abbiamo bisogno di strutture specifiche che ci permettano di accedere agli elementi in modo più efficiente.

Una di queste strutture è l'Heap.

6.1 proprietà di un Heap

Per far sì che la complessità scenda sotto $\Theta(n)$ è necessario che la struttura dati abbia delle proprietà specifiche, in questo caso l'Heap ha le seguenti proprietà:

- Deve essere un albero binario completo o semicompleto
- I nodi contengono oggetti su cui è definita una relazione d'ordinamento
- Per ogni nodo il contenuto è minore o uguale al contenuto dei suoi figli

Se tutte queste proprietà sono soddisfatte, allora l'Heap è una struttura dati valida.

6.1.1 Proprietà derivate: Alberi completi

*Ma cosa significa che deve essere un albero binario completo o semicompleto?

Definizione: Un albero binario di ricerca è un albero binario in cui ogni nodo ha un valore e i valori dei nodi a sinistra sono minori del valore del nodo padre, mentre i valori dei nodi a destra sono maggiori del valore del nodo padre.

Detto in parole povere: Un albero binario completo ogni nodo deve avere necessariamente 2 figli, mentre un albero binario semicompleto ha una metà che si differenzia dall'altra per al più un figlio, ad esempio, se mi mettessi sul nodo radice e guardassi i suoi figli, vedrei che i figli del figlio destro sono al massimo $n+1$ rispetto ai figli del figlio sinistro.

- le foglie di un albero completo sono $foglie = \lceil \frac{nodi}{2} \rceil$
- la profondità di un albero completo è $\lceil \log_2(n) \rceil$

*Nota: per essere completo un albero binario può anche avere tutti i nodi con 0 figli, poichè un singolo nodo soddisfa le proprietà di un albero binario completo.

6.2 Heap Sort

L'Heap Sort è un algoritmo di ordinamento che utilizza la struttura dati Heap per ordinare gli elementi. L'idea di base è quella di costruire un Heap a partire dall'array da ordinare e poi estrarre gli elementi dall'Heap in ordine crescente.

L'algoritmo funziona in due fasi:

- Costruzione dell'Heap: si costruisce un Heap a partire dall'array da ordinare. Questo richiede $O(n)$ tempo.
- Estrazione degli elementi: si estrae l'elemento minimo dall'Heap e lo si inserisce nell'array ordinato. Questo richiede $O(n \cdot \log(n))$ tempo.

6.2.1 Codice

```
1 void buildHeap(int[] A) { // costruzione dell'heap
2     int size = A.length;
3     for (int i = size / 2 ; i > 0; i--) { // per difetto
4         heapify(A, size, i);
5     }
6 }
```

```
1 void extractMax(int[] A) { // estrazione dell'elemento
    massimo
2     int size = A.length;
3     for(int i = size - 1; i > 0; i--) {
4         swap(A[0], A[i]); // scambia l'ultimo elemento
5         heapify(A, i, 0); // ricostruisce l'heap
6     }
7 }
```

```
1 void heapify(int[] A, int size, int i) { // ricostruzione
    dell'heap
2     int longest;
3     int left = 2 * i; // figlio sinistro / left[i]
4     int right = 2 * i + 1; // figlio destro / right[i]
5     if(left < size && A[left] > A[i])
6         longest = left;
7     else
8         longest = i;
9     if(right < size && A[right] > A[longest])
10        longest = right;
11    if(longest != i) {
12        swap(A[i], A[longest]); // scambia i nodi
13        heapify(A, size, longest); // ricostruisce l'heap
14    }
15 }
```


Complessità:

- La complessità temporale dell'Heap Sort è $O(n \cdot \log(n))$ nel caso peggiore.
- La complessità spaziale è $O(1)$ costante, essendo in-place.
- Algoritmo non è stabile, ma può essere reso stabile con modifiche.
- La costruzione dell'Heap richiede $O(n)$ tempo, mentre l'estrazione degli elementi richiede $O(n \cdot \log(n))$ tempo.

6.3 Problema 2: Limiti inferiori

Prima di affrontare gli algoritmi di ordinamento con complessità $O(n)\log(n)$, vorrei fare una digressione sui limiti inferiori e su come quasi ogni problema possa essere ricondotto a un albero decisionale. Da qui nasce il problema dei pesi: Problema 2: Abbiamo 12 sfere apparentemente identiche, ma una di esse ha un peso diverso (potrebbe essere più leggera o più pesante, ma non sappiamo quale). L'obiettivo è individuare la sfera anomala e determinare se è più leggera o più pesante, utilizzando esclusivamente una bilancia a due piatti e un massimo di tre pesate. Come possiamo risolvere il problema? *Soluzione a fine PDF

Ma invece di buttarci subito a capofitto nella soluzione, proviamo a capire se è possibile risolverlo. Il che è molto facile da capire: $\log_3(25)$ la base del logaritmo indica le possibilità che abbiamo, e il numero 25 indica il numero di casi che abbiamo per determinare quale sia la sfera anomala.

il risultato del logaritmo è ≈ 2.9 , questo numero ci dice che con 3 pesate è possibile individuare la sfera anomala. Quindi si è possibile trovare la sfera anomala, poichè ci hanno fornito abbastanza pesate. Ma questo a cosa serve? capendo quale sia il limite inferiore di qualcosa possiamo anche capire quale sia l'asticella che non può essere superata.

Limite inferiore degli algoritmi di ordinamento per confronto

Se ponessimo nello stesso modo del problema un qualsiasi algoritmo di ordinamento, avremmo che il numero di confronti necessari per ordinare n elementi è almeno $\log_2(n!)$. Questo è dovuto al fatto che ci sono $n!$ permutazioni possibili di n elementi e ogni confronto può eliminare al massimo la metà delle permutazioni rimanenti. Quindi, il numero minimo di confronti necessari per ordinare n elementi è $\log_2(n!)$, e, avendo presente alcune proprietà dei logaritmi e dei fattoriali, possiamo semplificare in $O(n \cdot \log(n))$.

Ma come detto in precedenza, non è sempre sufficiente un $O(n \cdot \log(n))$, ma volendo scendere sotto questa soglia abbiamo bisogno di avere una struttura dati oppure dei vincoli specifici che ci permettano di farlo.

7 Algoritmi di Ordinamento $\in O(n)$

Per far sì che un algoritmo scenda sotto il limite inferiore di $O(n \cdot \log(n))$, come detto in precedenza, dobbiamo porre qualche vincolo.

7.1 Counting Sort

Il Counting Sort è un algoritmo di ordinamento che funziona bene quando gli elementi da ordinare sono numeri interi compresi in un intervallo limitato. L'idea di base è quella di contare il numero di occorrenze di ciascun elemento e poi costruire l'array ordinato a partire da questi conteggi.

L'idea di base è quella di contare il numero di occorrenze di ciascun elemento e poi costruire l'array ordinato a partire da questi conteggi.

7.1.1 Codice

```
1 void countingSort(int[] A, int k) { // con K numero massimo
2     int[] count = new int[k + 1]; // array di conteggio
3     int[] output = new int[A.length]; // array di output
4     for(int i = 1; i <= k; i++) {
5         count[i] = 0; // inizializza l'array di conteggio
6     }
7     for(int i = 0; i < A.length; i++) {
8         count[A[i]]++; // conta le occorrenze
9     }
10    for(int i = 1; i <= k; i++) {
11        count[i] += count[i - 1]; // accumula le occorrenze
12    }
13    for(int i = A.length - 1; i >= 0; i--) {
14        output[count[A[i]] - 1] = A[i];
15        count[A[i]]--; // decrementa il conteggio
16    }
```

Complessità:

- La complessità temporale del Counting Sort è $O(n + k)$, dove n è il numero di elementi da ordinare e k è il valore massimo degli elementi.
- La complessità spaziale è $O(k)$, poichè richiede spazio extra per l'array di conteggio.
- Algoritmo stabile, poichè mantiene l'ordine relativo degli elementi con lo stesso valore.
- Funziona bene quando k è relativamente piccolo rispetto a n , altrimenti la complessità spaziale può diventare elevata.

7.2 Radix Sort

Il Radix Sort è un algoritmo di ordinamento che funziona bene quando gli elementi da ordinare sono numeri interi o stringhe di caratteri.

L'idea è quella di ordinare gli elementi in base a ciascuna cifra o carattere, partendo dalla cifra meno significativa (o dal carattere meno significativo) e procedendo verso la cifra più significativa (o il carattere più significativo).

L'algoritmo utilizza il Counting Sort come sottoprocedura per ordinare gli elementi in base a ciascuna cifra o carattere.

7.2.1 Codice

```
1 void radixSort(int[] A) {
2     int max = getMax(A); // trova il valore massimo
3     for(int exp = 1; max / exp > 0; exp *= 10) {
4         countingSort(A, exp); // ordina in base alla cifra
           corrente
5     }
6 }
```

Complessità:

- La complessità temporale del Radix Sort è $O(n \cdot k)$, dove n è il numero di elementi da ordinare e k è il numero di cifre (o caratteri) massime.
- La complessità spaziale è $O(n + k)$, poichè richiede spazio extra per l'array di conteggio e l'array di output.
- Algoritmo stabile, poichè mantiene l'ordine relativo degli elementi con lo stesso valore.
- Funziona bene quando k è relativamente piccolo rispetto a n , altrimenti la complessità spaziale può diventare elevata.

7.3 Bucket Sort

Il Bucket Sort è un algoritmo di ordinamento che funziona bene quando gli elementi da ordinare sono distribuiti uniformemente in un intervallo limitato.

L'idea è quella di suddividere gli elementi in un certo numero di "secchi" (o bucket) e poi ordinare ciascun secchio separatamente. Alla fine, si uniscono i secchi ordinati per ottenere l'array finale ordinato.

7.3.1 Codice

```
1 void bucketSort(int[] A, int k) { // massimo dell'array
2     List<List<Integer>> buckets = new ArrayList<>(k);
3     for(int i = 0; i < k; i++) {
4         buckets.add(new ArrayList<>()); // crea i secchi
5     }
6     for(int i = 0; i < A.length; i++) {
7         int index = A[i] / k; // trova il secchio corretto
8         buckets.get(index).add(A[i]); // aggiunge l'elemento
           al secchio
9     }
10    for(int i = 0; i < k; i++) {
11        Collections.sort(buckets.get(i)); // ordina ciascun
           secchio
12    }
13 }
```

Complessità:

- la complessità temporale del Bucket Sort è $O(n + \kappa)$ dato da n numero di elementi da ordinare, κ numero massimo di elementi in un secchio, dato che l'inserimento nei bucket è $\Theta(1)$ e, se distribuiamo uniformemente gli elementi in ogni bucket, la complessità finale è $O(n)$.
- La complessità spaziale è $O(n + k)$, poichè richiede spazio extra per i secchi e l'array di output.
- Algoritmo stabile, poichè mantiene l'ordine relativo degli elementi con lo stesso valore.
- nel caso peggiore, cioè se distribuisco non uniformemente gli elementi, posso arrivare ad una complessità di $O(n^2)$, ma questo è molto raro, una probabilità, se si strutturano bene i secchi, di $\frac{1}{n^{n-1}}$

8 Problema della selezione

Il problema è definito da due concetti, un input e un output. L'input è una sequenza di oggetti su cui è definita una relazione d'ordinamento, con indice i compreso tra 1 e n , l'output è l'oggetto che occupa la posizione i -esima nella sequenza ordinata.

Sappiamo che esistono limiti superiori e inferiori per cui un algoritmo risiede in un dato intervallo, per quanto riguarda gli algoritmi di ordinamento i vari limiti sono già stati trattati, ma , non ho mai parlato degli algoritmi di selezione.

Un limite superiore di questi algoritmi è $O(n \cdot \log(n))$, dato che l'algoritmo deve ordinare un array di dimensione, al più, n .

Un limite inferiore è $\Omega(n)$, dato che l'algoritmo deve esaminare ogni elemento dell'array almeno una volta.

come esempio abbiamo:

```
1  int select(int[] A, int start, int end, int index){
2      if(start == end) return A[start];
3
4      int partition = partition(A, start, end);
5      // arrotondato per difetto
6      int pivot = partition - start + 1;
7
8      if(index == pivot)
9          return select(A, start, partition, index);
10     else
11         return select(A, partition + 1, end, index -
12                        pivot);
13 }
```

Possiamo notare che nel caso peggiore l'algoritmo ha una complessità $T(n) = T(n-1) + n$, che si risolve in $\Theta(n^2)$, invece, nel caso medio la complessità è $\Theta(n)$, poichè la partizione divide l'array in due metà, cioè $T(n) = n + T(\frac{n}{2})$.

9 Selezione del massimo e del minimo

Ora che abbiamo introdotto il concetto di selezione, possiamo complicare un pò le cose e, invece di selezionare un elemento a caso, selezionare il massimo e il minimo.

9.1 Select Min/Max

In tempo lineare è possibile trovare il minimo o il massimo di un array:

9.1.1 Codice

```
1  int minimo(int[] A) {
2      int min = A[0];
3      for(int i = 1; i < A.length; i++)
4          if(A[i] < min)
5              min = A[i];
6      return min;
7  }
```

*Nota: trovare il minimo equivale a trovare $\text{select}(A, 0)$ e trovare il massimo equivale a trovare $\text{select}(A, n-1)$.

È facile vedere che la complessità è lineare, ma, se volessimo andare al di sotto di questa soglia? è possibile?

9.1.2 Trovare il minimo e il massimo in meno di $O(n)$

Per trovare il massimo (o il minimo) elemento di un array di grandezza n , bisogna fare almeno $n-1$ confronti, poichè, bisogna confrontare ogni elemento con l'elemento massimo (o minimo) corrente. Di conseguenza, non è possibile avere un algoritmo per la ricerca del massimo (o minimo) in cui c'è un elemento che non "perde"

9.2 Esempio di Immagine

Soluzione Problema 1

$$\exists_{a,b,c} \exists_x \mid a \cdot b \cdot c = 36 \wedge a_1 + b_1 + c_1 = x = a_2 + b_2 + c_2 \wedge a_1 > b_1 \geq c_1$$

Dobbiamo trovare tre numeri il cui prodotto sia 36. Tuttavia, ci sono molte combinazioni possibili, quindi abbiamo bisogno di un ulteriore indizio. Sfortunatamente, nemmeno conoscere il numero civico è sufficiente, perché Luca ha dovuto chiedere ancora un altro suggerimento. Questo significa che, tra tutte le combinazioni possibili, ce ne sono almeno due che hanno la stessa somma. Infatti, le coppie (2,2,9) e (1,6,6) soddisfano entrambe la condizione, poiché la loro somma è 13. A questo punto, l'ultimo indizio diventa decisivo: Paolo specifica che solo uno dei suoi figli è più grande degli altri due. Tra le due combinazioni, solo (2,2,9) soddisfa questa condizione, perché in (1,6,6) ci sarebbero due figli della stessa età. La risposta corretta è quindi 2,2,9.

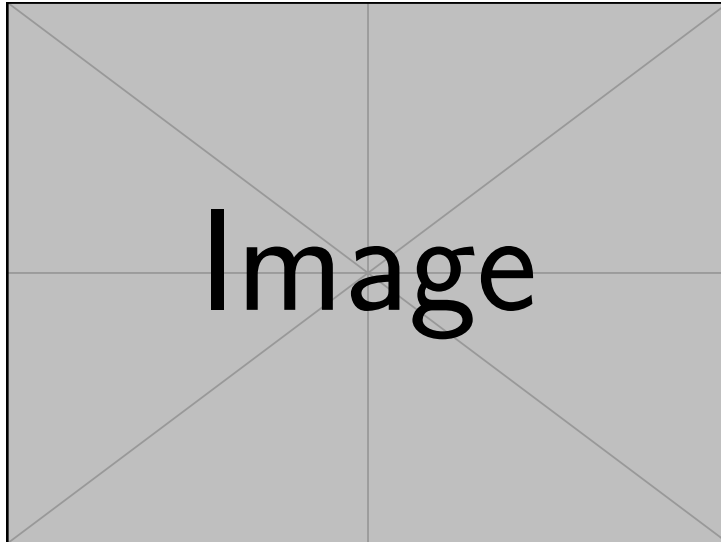


Figure 1: Esempio di immagine

Soluzione Esercizi (1)

1) $T(n) = T(n-1) + n \in O(n^2)$

$$\begin{aligned} T(n) &\leq c(n-1)^2 + n \\ &\leq c(n^2 - 2n + 1) + n \\ &\leq cn^2 - 2cn + c + n \end{aligned}$$

Ora avendo trovato $c \cdot n^2$ mi basta trovare quando $-2cn + c + n \leq 0$

$$\begin{aligned} 0 &\geq -2cn + c + n \\ 0 &\geq n - 2cn + c \\ 2c &\geq n + c \\ c &\geq \frac{n}{2} + \frac{1}{2} \end{aligned}$$

Quindi la funzione appartiene a $O(n^2)$ se $c \geq \frac{n}{2} + \frac{1}{2}$, quindi la funzione appartiene a $O(n^2)$

2) $T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n$

Per questa funzione è molto più semplice, basta applicare il teorema dell'esperto, o master theorem, e vedere che $T(n) \in O(n)$, basti vedere che $n^{\log_4(3)+\epsilon} = n$ per chiunque stia leggendo, questa cosa è una bestemmia nero su bianco, ma è molto semplice da capire, per favore non replicatela

3) $T(n) = 4T(\frac{n}{2}) + n^3$

Anche qui basta il Teorema dell'esperto il che ci riconduce al caso 3 del teorema, quindi $T(n) \in O(n^3)$

4) $T(n) = 4T(\frac{n}{2}) + n^{\log_2(4)}$

caso 2 del teorema, quindi $T(n) \in O(n^{\log_2(4)} \cdot \log(n)) \rightarrow O(n^2 \cdot \log(n))$

Soluzione Problema 2

Per risolvere il problema delle 12 sfere (dove una sfera ha un peso diverso, senza sapere se è più leggera o più pesante) con 3 pesate, seguiamo questi passaggi:

Pesata 1:

- Dividi le 12 sfere in 3 gruppi:
 - Gruppo A: {1,2,3,4}
 - Gruppo B: {5,6,7,8}
 - Gruppo C: {9,10,11,12}
- Poni il Gruppo A sul piatto sinistro e il Gruppo B sul piatto destro.
- **Se la bilancia è in equilibrio:**
 - Le sfere dei Gruppi A e B sono normali.
 - L'anomalia è nella sfera appartenente al Gruppo C.
- **Se la bilancia è sbilanciata:**
 - L'anomalia si trova tra le sfere dei Gruppi A e B.
 - L'indicazione della bilancia fornisce un primo indizio sulla natura dell'anomalia (più pesante o più leggera).

Pesata 2:

- **Se la Pesata 1 era in equilibrio:**
 - Confronta 3 sfere normali (da uno dei gruppi già confermati, ad esempio dal Gruppo A) con 3 sfere del Gruppo C.
 - **Se in equilibrio:** l'anomalia è la sfera non pesata del Gruppo C.
 - **Se sbilanciata:** l'anomalia è tra le 3 sfere pesate del Gruppo C.
- **Se la Pesata 1 era sbilanciata:**
 - Esegui una pesata strategica scambiando alcune sfere sospette con altre.
 - L'obiettivo è restringere il campo delle sfere sospette e ottenere ulteriori indicazioni sulla loro natura.

Pesata 3:

- Utilizza la terza pesata per confrontare le sfere rimanenti sospette.
- Confronta una o due sfere sospette con sfere normali (già confermate) per determinare esattamente:
 - Quale sfera è l'anomala.
 - Se l'anomalia consiste in un peso maggiore o minore rispetto alle altre.