ALGORITMO

Un algoritmo è una sequenza finita di operazioni, anche dette istruzioni, che riceve un input e restituisce un output al fine di risolvere un determinato problema.

PROBLEMA DELL'ORDINAMENTO

Data in input una sequenza di numeri n numeri $< a_1, a_2, ..., a_n >$ si vuole ottenere in output una permutazione tale che:

$$< a_1^{'}, a_2^{'}, ..., a_n^{'} > \text{ con } a_1^{'} \leq a_2^{'} \leq ... \leq a_n^{'}$$

Il tipo di algoritmo usato per l'ordinamento dipende da fattori come: numero di elementi, ordinamento iniziale, tipo di memorizzazione usata, ecc...

CORRETTEZZA DI UN ALGORITMO

Un algoritmo viene detto "corretto" se ad ogni istanza di input termina con l'output giusto.

Induzione Matematica: Supponendo di voler dimostrare che la proprietà $P(n)$ vale per qualsiasi $n \in \mathbb{N}$. Definisco l'insieme universo $U = n \in \mathbb{N}$ t.c. vale $P(n)$. Allora:

- **Passo Base** → dimostro che vale $P(1)$ o $P(0)$;
- **Passo Induttivo** → suppongo che $P(n)$ valga per un generico n. Posso dunque concludere che U coincide con $\mathbb{N}$.

Invariante di Ciclo: formulo un'affermazione che deve essere verificata in 3 diversi momenti:

- **Inizializzazione** → l'I.C. deve essere vera prima della prima iterazione del ciclo;
- **Conservazione** → l'I.C. deve essere vera prima della successiva iterazione del ciclo;
- **Conclusione** → l'I.C. al termine del ciclo deve fornire una condizione che permetta di verificare se l'output dell'algoritmo è corretto.

ANALISI ASINTOTICA

Il calcolo asintotico è usato per analizzare la complessità di un algoritmo ovvero per stimare quanto tale complessità aumenta all'aumentare della dimensione dell'input. Uso la funzione $T(n)$ in quanto la risorsa considerata è il tempo (misurato in operazioni elementari). Allora:

Notazione O → $g(n)$ è un limite asintotico superiore per $f(n)$ ed è definito come:

$$\boldsymbol{O(g(n))} = \{f(n) : \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ t.c. } \boldsymbol{0 \leq f(n) \leq c \cdot g(n)} \ \forall n \geq n_0\}$$

Notazione o → è definito come:

$$\boldsymbol{o(g(n))} = \{f(n) : \forall c \in \mathbb{R}, \exists n_0 > 0 \in \mathbb{N} \text{ t.c. } \boldsymbol{0 \leq f(n) < c \cdot g(n)} \ \forall n \geq n_0\}$$

Notazione Ω → $g(n)$ è un limite asintotico inferiore per $f(n)$ ed è definito come:

$$\boldsymbol{\Omega(g(n))} = \{f(n) : \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ t.c. } \boldsymbol{0 \leq c \cdot g(n) \leq f(n)} \ \forall n \geq n_0\}$$

Notazione ω → è definito come:

$$\boldsymbol{\omega(g(n))} = \{f(n) : \forall c \in \mathbb{R}, \exists n_0 > 0 \in \mathbb{N} \text{ t.c. } \boldsymbol{0 \leq c \cdot g(n) < f(n)} \ \forall n \geq n_0\}$$

Notazione Θ → $g(n)$ è un limite asintoticamente stretto per $f(n)$ ed è definito come:

$$\boldsymbol{\Theta(g(n))} = \{f(n) : \exists\, c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} \text{ t.c. } \boldsymbol{0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)} \ \forall n \geq n_0\}$$

Teorema :

$f(n) = \Theta g(n)$ SSE $f(n) = Og(n)$ e $f(n) = \Omega g(n)$

Proprietà :

- **Transitiva** : $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n)) \implies f(n) = \Theta(h(n))$ (vale anche per O, o, Ω, ω);
- **Riflessiva** : $f(n) = \Theta(f(n))$ (vale anche per O, Ω);
- **Simmetria** : $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$;
- **Simmetria Trasposta** :

$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
$f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$

CLASSI DI COMPLESSITA'

Per categorizzare la complessità degli algoritmi faccio riferimento alla crescita di funzioni semplici. Le classi sono:

- $\boldsymbol{O(1)}$ → complessità costante;
- $\boldsymbol{O(kn)}$ con $k < 1$ → complessità sottolineare (ES: ricerca sequenziale);
- $\boldsymbol{O(n)}$ → complessità lineare (ES:ricerca sequenziale);
- $\boldsymbol{O(n \cdot \ln n)}$ → (ES: algoritmi di ordinamento ottimi);
- $\boldsymbol{O(n^k)}$ con $k \geq 2$ → (ES: BubbleSort con $O(n^2)$);
- $\boldsymbol{O(k^n)}$ → complessità esponenziale;

ALGORITMI ITERATIVI - INSERTION SORT

È uno degli algoritmi iterativi più semplici. Ordina "in place" ed è efficiente per insiemi quasi ordinati. Lo pseudocodice è:

InsertionSort(A)

```
1: for j ← 2 to A.lenght do
2:    key ← A[j]
3:    i ← j − 1
4:    while i > 0 AND A[i] > key do
5:       A[i + 1] ← A[i]
6:       i ← i − 1
7:    end while
8:    A[i + 1] ← key
9: end for
```

Correttezza con I.C.

"All'inizio di ogni iterazione del ciclo *for*, il sottoarray $A[1, \ldots, j-1]$ è ordinato ed è formato dagli stessi elementi che erano originariamente in $A[1, \ldots, j-1]$, ma ordinati".

Costo

- **Caso Ottimo** (Array Ordinato): $O(n)$ → non si entra mai nel *while* ma si esegue ogni volta il confronto, al contrario si esegue sempre il *for*.
- **Caso Peggiore** (Array Ordinato al Contrario): $O(n^2)$ → si esegue il *while* per il numero massimo di volte, il costo del *for* è irrilevante rispetto a quello del *while*.
- **Caso Medio** (Array Disordinato): $O(n^2)$ → il costo dipende dall'ordine ma statisticamente il *while* verrà eseguito soltanto la metà delle volte rispetto al caso peggiore, il costo rimane quadratico.

ALGORITMI ITERATIVI - SELECTION SORT

È un algoritmo molto semplice, opera "in place". Il suo obbiettivo è quello di trovare il minimo all'interno dell'array e inserirlo nella sequenza ordinata fino a qual momento. Lo pseudocodice è:

SelectionSort(A)

```
1: n ← A.lenght
2: for j ← 1 to n − 1 do
3:    smallest ← j
4:    for i ← j + 1 to n do
5:       if A[I] < A[smallest] then
6:          smallest ← i
7:       end if
8:    end for
9:    exchange A[j] ↔ A[smallest]
10: end for
```

Correttezza con I.C.

- **Ciclo Interno** → "All'inizio della i-esima iterazione del *for* interno, $A[min]$ è minore o uguale di ogni elemento di $A[j, \ldots, i-1]$ cioè $\forall j \in [j, \ldots, i-1]$ si ha che $A[min] \leq A[j]$".
- **Ciclo Esterno** → "All'inizio di ogni iterazione del ciclo *for* esterno, il sottoarray $A[1, \ldots, j-1]$ è ordinatoe composto solo dagli elementi più piccoli dell'array A".

Costo

In questo caso è sempre $O(n^2)$ perché, sia nel caso migliore sia nel caso peggiore, i due cicli *for* vengono comunque eseguiti. L'unica riga che può non essere eseguita è la 6, che è trascurabile al fine del calcolo del costo.

COMPLESSITÀ ALGORITMI RICORSIVI

Equazione di Ricorrenza

Si usa per stimare il tempo di esecuzione di algoritmi ricorsivi.

Si fa rifermento alla divisione e combinazione di vari sottoproblemi. Infatti l'equazione è:

$$\mathbf{T(n)} = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(\frac{n}{b}) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

Se $n \leq c$ con c una costante, si ha il caso base; altrimenti si divide in a sottoproblemi di dimensione $\frac{1}{b}$. Questo implica un costo $D(n)$ di divisione del problema e un costo $C(n)$ di combinazione dei sottoproblemi. I metodi di risoluzione di tale equazione sono:

Metodo di Sostituzione → Ipotizzo una soluzione e la dimostro con un'induzione matematica:
1. indovino una soluzione e formulo un'ipotesi induttiva;
2. sostituisco nell'equazione di ricorrenza le epressioni $T(\cdot)$;
3. dimostro che è valida anche per il caso base.

La soluzione per una ricorrenza può essere:
- **Esatta** → se la ricorrenza è formata da una funzione esatta allora anche la soluzione sarà sempre esatta;
- **Asintotica** → mi riconduco ad una soluzione esatta, ipotizzo per prima cosa che la soluzione $T(n)$ sia per esempio un O di qualche funzione e poi cerco le costanti che verificano l'ipotesi induttiva (quelle presenti nella definizione di O).

Metodo dell'Albero di Ricorsione → È un metodo grafico per arrivare alla soluzione. L'albero deve sempre condurre ad un'equazione esatta. I nodi rappresentano i costi dei vari sottoproblemi e permette di aver un'idea del costo complessivo di tutte le esecuzioni dell'algoritmo. Si sommano i costi di tutti i vari sottolivelli per poi fare una somma complessiva di tali sottolivelli.

Metodo dell'Esperto → Sia $T(n)$ una funzione definita sui naturali dalla ricorrenza $T(n) = aT(\frac{n}{b}) + f(n)$ con $a \geq 1, b > 1$ costanti e $f(n) > 0$ asintoticamente, allora $T(n)$ può essere limitata nei seguenti modi:
- **caso 1** → se $f(n) = O(n^{\log_b a - \epsilon})$ per qualche costante $\epsilon > 0$ allora: $\mathbf{T(n) = \Theta(n^{\log_b a})}$
- **caso 2** → se $f(n) = \Theta(n^{\log_b a})$ allora: $\mathbf{T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)}$
- **caso 3** → se $f(n) = \Omega(n^{log_b a + \epsilon})$ per qualche costante $\epsilon > 0$ e $f(n)$ t.c. $a\, f(\frac{n}{b}) \leq c\, f(n)$ per qualche costante $c < 1$ e $\forall n \geq n_0$, allora: $\mathbf{T(n) = \Theta(f(n))}$
- **Condizione di Regolarità:** Devo assicurarmi che quando scendo nell'albero $f(\cdot)$ diventa più piccola.

Albero di Ricorsione + Esperto → Combinando i metodi precedentemente descritti ottengo:
- **caso 1** → se il costo cresce dalla radice alle foglie ("Costo Dominato dalle Foglie"), allora: $\mathbf{T(n) = \Theta(n^{\log_b a})}$
- **caso 2** → se il costo è circa lo stesso in ogni livello, allora: $\mathbf{T(n) = \Theta(n^{\log_b a} \cdot \log_2 n)}$
- **caso 3** → se il costo decrementa dalla radice alle foglie ("Costo Dominato dalla Radice"), allora: $\mathbf{T(n) = \Theta(f(n))}$

ALGORITMI RICORSIVI - MERGE SORT

Divide ricorsivamente l'array da ordinare in due sottoarray di uguale lunghezza. Una volta arrivato alla lunghezza unitaria combina ricorsivamente i sottoarray ordinandoli. Lo pseudocodice è:

MergeSort(A,p,r)

```
1: if p < r then
2:    q ← ⌊(p + r)/2⌋
3:    MergeSort(A,p,q)
4:    MergeSort(A,q+1,r)
5:    Merge(A,p,q,r)
6: end if
```

Merge(A,p,q,r)

```
1: n1 ← q − p + 1
2: n2 ← r − q
3: for i ← 1 to n1 do
4:    L[i] ← A[p + i − 1]        {Crea il sottoarray sinistro}
5: end for
6: for j ← 1 to n2 do
7:    R[j] ← A[q + j]            {Crea il sottoarray destro}
8: end for
9: L[n1 + 1] ← ∞
10: R[n2 + 1] ← ∞
11: i ← 1
12: j ← 1
13: for k ← p to r do
14:    if L[i] ≤ R[j] then
15:       A[k] ← L[i]
16:       i ← i + 1
17:    else
18:       A[k] ← R[j]
19:       j ← j + 1
20:    end if
21: end for
```

Costo

Ho che il costo di divisione è $D(n) = \Theta(1)$ mentre il costo di combinazione è $C(n) = \Theta(n)$ Inoltre ad ogni ricorrenza successiva l'algoritmo risolve 2 problemi di dimensione $\frac{n}{2}$. Allora l'equazione di ricorrenza sarà:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(\frac{n}{2}) + \Theta(1) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Poiché divido sempre il problema in 2 sottoproblemi di uguale dimensione è facile risolvere il problema con il metodo dell'albero di ricorrenza. Ogni livello ha costo $\Theta(n)$ ed essendo il numero di livelli $\log_2 n$ avrò che il costo del problema è:

$$\mathbf{T(n)} = \Theta(n \cdot \log_2 n) + \Theta(n) + \Theta(1) = \mathbf{\Theta(n \cdot \log_2 n)}$$

ALGORITMI RICORSIVI - QUICKSORT

Divide ricorsivamente l'array da ordinare in due sottoarray grazie all'uso di due indici (ordina sul posto) grazie all'algoritmo "partition" per poi riordinarli e combinarli grazie all'algoritmo "quicksort". Lo pseudocodice è:

QuickSort(A,p,r)

```
1: if p < r then
2:    q ← Partition(A,p,r)
3:    QuickSort(A,p,q-1)
4:    QuickSort(A,q+1,r)
5: end if
```

Partition(A,p,r)

```
1: x ← A[r]
2: i ← p − 1
3: for j ← p to r − 1 do
4:    if A[j] ≤ x then
5:       i ← i + 1
6:       exchange A[i] ↔ A[j]
7:    end if
8: end for
9: exchange A[i + 1] ↔ A[r]
10: return i + 1
```

Costo

Partition → ha costo fisso $\Theta(n)$.

Quicksort → il costo dipende dal partizionamento dei sottoarray. Infatti:
- **Caso migliore** (sottoarray bilanciati) → ogni sottoarray ha dimensione $\leq \frac{n}{2}$ quindi il costo sarà $\Theta(\log_2 n)$;
- **Caso peggiore** (sottoarray sbilanciati) → i due sottoarray sono di dimensione 0 e $n - 1$ quindi il costo complessivo sarà $\Theta(n^2)$;
- **Caso Medio** → supponendo una suddivisione

parzialmente sbilanciata (ad esempio 9 a 1) avremo un costo:

$$T(n) = T(\tfrac{9n}{10}) + T(\tfrac{n}{10}) + \Theta(n) = O(n \cdot \log_2 n)$$

CENNI DI PROBABILITÀ

Definisco S come lo spazio degli eventi, ovvero come un insieme di eventi E (esiti di esperimenti). Allora la probabilità P che un evento si verifichi è:

$$\boldsymbol{P(E)} = \tfrac{\text{casi favorevoli}}{\text{casi possibili}}$$

Proprietà:

1. $P(E) \geq 0 \quad \forall$ evento E;
2. $P(S) = 1$;
3. $P(A \cup B) = P(A) + P(B)$ se A e B sono mutuamente esclusivi;
3. $P(A \cup B) = P(A) + P(B) - P(A \cap B)$ se A e B sono compatibili (cioè hanno eventi in comune).

Probabilità Composta:

Eventi Indipendenti → Due eventi sono indipendenti se il realizzarsi di uno non influenza la probabilità di realizzarsi dell'altro. La probabilità che si realizzino entrambi è:

$$\boldsymbol{P(A \cap B) = P(A) \cdot P(B)}$$

Eventi Dipendenti → Due eventi sono dipendenti se il realizzarsi di uno influenza la probabilità di realizzarsi dell'altro. Indico con $P(B|A)$ la probabilità che si verifichi B supponendo che si sia verificato A, allora la probabilità che si realizzino entrambi è:

$$\boldsymbol{P(A \cap B) = P(A) \cdot P(B|A)}$$

Probabilità Condizionata:

$$\boldsymbol{P(A|B) = \tfrac{P(A \cap B)}{P(B)}}$$

Teorema di Bayes → Poiché $A \cap B = B \cap A$ ho che:

$$\boldsymbol{P(A \cap B) = P(B)P(A|B) = P(A)P(A|B)}$$

Variabile Aleatoria → È l'insieme di tutti i possibili risultati che possono verificarsi in un esperimento.

Distribuzione di Probabilità → Elenca le probabilità di verificarsi di ogni valore della variabile aleatoria a cui si riferisce. La somma di tutte le probabilità di una distribuzione di probabilità è sempre pari a 1 (cioè $\sum P(X) = \sum_i P(x_i) = 1$).

Valore Atteso → È il valore medio che mi aspetto da una lunga serie di osservazioni, ed è definito come:

$$\boldsymbol{\mu = E[X] = \sum_{i=1}^{n} x_i \cdot P(x_i)}$$

Linearità :

$$\boldsymbol{E[X + Y] = E[X] + E[Y]};$$
$$g(x) = ax \implies \boldsymbol{E[aX] = aE[X]}.$$

VARIABILI CASUALI INDICATRICI

Dato uno spazio dei campioni S e un evento A, si definisce variabile casuale indicatrice:

$$\boldsymbol{I\{A\}} \text{ oppure } \boldsymbol{X_A} = \begin{cases} 1 & \text{se si verifica A;} \\ 0 & \text{se non si verifica A} \end{cases}$$

Lemma → $E[X_a] = Pr\{A\}$.

ALGORITMI RANDOMIZZATI

Poiché non posso conoscere l'ordine in cui l'algoritmo riceverà gli input, non posso, allo stesso modo, conoscere la probabilità delle $n!$ possibili permutazioni. Uso l'algoritmo "random" per forzare l'ordine casuale. Lo pseudocodice è:

RandomizeInPlace(A)

```
1: n ← A.lenght
2: for i ← 1 to n do
3:     excange A[i] ↔ A[Random(i, n)]
4: end for
```

Costo:

$\Theta(1)$ per ogni iterazione, quindi $\Theta(n)$ in totale.

ALGORITMI RANDOMIZZATI - QUICKSORT

Cerco di avvicinarmi il più possibile all'ipotesi di caso medio, per farlo randomizzo la scelta del pivot q per l'algoritmo "partition". Lo pseudocodice è:

RandPartition(A,p,r)

```
1: i ← Random(p,r)
2: exchange A[r] ↔ A[i]
3: return Partition (A,p,r)
```

RandQuickSort(A,p,r)

```
1: if p < r then
2:     q ← RandomizedPartition(A,p,r)
3:     RandomizedQuickSort(A,p,q-1)
4:     RandomizedQuickSort(A,q+1,r)
5: end if
```

Costo

Il tempo di esecuzione atteso di RandomizedQuickSort è $O(n \log_2 n)$ in quanto statisticamente rispecchia il caso medio di QuickSort.

ALBERO DI DECISIONE

È un albero binario che rappresenta i possibili confronti fatti da un algoritmo su un input di una specifica dimensione. La radice rappresenta le prima due posizioni confrontate. Si procede a cascata analizzando tutti i possibili casi. Le foglie rappresenteranno infine tutte le possibili combinazioni degli input.

NOTA BENE La radice e i nodi intermedi sono strutturati come ("a":"b"). I confronti si effettuano sulla possibilità che sulla posizione "a" vi sia un valore $\leq$ / $>$ di quello in posizione "b". Non si fa dunque riferimento allo specifico valore ma solo alla relazione che lega la coppia di posizioni.

Lemma → Ogni albero binario di altezza h ha al massimo 2^h foglie.

Teorema → Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log_2 n)$ (cioè nel "caso migliore").

ORDINAMENTO IN TEMPO LINEARE

Gli algoritmi visti fino ad ora non raggiungono un costo minore di $\Theta(n \log_2 n)$, è possibile fare di meglio? In ogni caso si ha un costo di $\Omega(n)$ per esaminare tutti gli input.

ALGORITMI ITERATIVI - COUNTING SORT

È un algoritmo di ordinamento che non si basa sui confronti, ma richiede due array di supporto per effettuare l'ordinamento. Inoltre può ordinare solo numeri naturali. Lo pseudocodice è:

CountingSort(A,B,k)

```
1: for i ← 0 to k do
2:     C[i] ← 0
3: end for
4: for j ← 1 to A.lenght do
5:     C[A[j]] ← C[A[j]] + 1
6: end for
7: for i ← 1 to k do
8:     C[i] ← C[i] + C[i − 1]
9: end for
10: for j ← A.lenght downto 1 do
11:     B[C[A[j]]] ← A[j]
12:     C[A[j]] ← C[A[j]] − 1
13: end for
```

Costo

Il costo di CountingSort è $\Theta(n + k)$, che diventa $\Theta(n)$ se $k = O(n)$. Imponendo la dimensione massima di k come condizione iniziale ottengo un ordinamento in tempo lineare.

NOTA BENE CountingSort è stabile (se due valori sono uguali, il primo nell'array in input rimane primo nell'array di output).

ALGORITMI ITERATIVI - RADIX SORT

È un algoritmo di ordinamento che usa il concetto controintuitivo di ordinare le singole cifre a partire dalla meno significativa. Necessita inoltre di riceve in ingresso in numero di cifre "d". Lo pseudocodice è:

RadixSort(A,d)

```
1: for i ← 1 to d do
```

2: "usa ordinamento stabile per ordinare l'array A sulla cifra i"
3: **end for**

Correttezza con I.C.
"Prima della i-esima iterazione i numeri sono ordinati sulla base della $i-1$esima cifra meno significativa".
Costo
Supponendo di usare CountingSort come algoritmo di ordinamento stabile, avrò che il costo complessivo è $\Theta(d(n+k))$ ed utilizzando la condizione $k = O(n)$ diventa $\Theta(d \cdot n)$.
Bilanciare Parole e Cifre
Supponendo di dover ordinare n parole di b bit divise in cifre da r bit, ho che il costo del CountingSort sarà $\boldsymbol{\Theta(\frac{b}{r}(n+2^r))}$. In particolare scegliendo $r \approx \log_2 n$ ho che il costo (sostituendo) diventa $\boldsymbol{\Theta(\frac{b \cdot n}{\log_2 n})}$ (meglio di $\Theta(n)$).

HASHING

Per la realizzazione di dizionari efficienti si usano le tabelle Hash che hanno tempo di ricerca atteso $O(1)$ mentre hanno $\Theta(n)$ nel caso peggiore. Dato un universo delle chiavi $U = \{0, 1, \ldots, m-1\}$ esistono diversi modi per implementarle, ovvero:

Indirizzamento Diretto → spesso il numero di chiavi memorizzate K è molto minore dell'insieme delle chiavi U, si spreca quindi molto spazio nella tabella. Per risolvere questo problema si usano le funzioni Hash $h(k)$ memorizzando il valore in $T[h(k)]$ e non in $T[k]$.

Metodo delle Divisioni → una valida funzione Hash è quella data dal metodo delle divisioni, ovvero:

$$\boldsymbol{h(k) = k \mod m}$$

Un buon criterio di scelta per la variabile m è un numero primo non troppo vicino ad una potenza del 2 (se si usa una potenza del 2 si rischia di raggruppare i valori sulla base dei loro valori meno significativi).

Risoluzione delle Collisioni → le funzioni Hash non sono iniettive quindi dovrò risolvere le collisioni che si creeranno, esistono 2 metodi:

Concatenamento → tutti gli elementi con lo stesso Hash sono memorizzati in una lista concatenata.

Indirizzamento Aperto →
