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:

INSERTION-SORT(A)

```
1  for j ← 2 to A.length
2      key ← A[j]
3      i ← j − 1
4      while i > 0 AND A[i] > key
5          A[i + 1] ← A[i]
6          i ← i − 1
7      A[i + 1] ← key
```

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
3      smallest ← j
4      for i ← j + 1 to n
5          if A[i] < A[smallest]
6              smallest ← i
7      exchange A[j] ↔ A[smallest]
```

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
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  for i ← 1 to n − 1
 4      L[i] ← A[p + i − 1]
 5  for j ← 1 to n2
 6      R[i] ← A[q + j]
 7  L[n1 + 1] ← ∞
 8  R[n2 + 1] ← ∞
 9  i ← 1
10  j ← 1
11  for k ← p to r
12      if L[i] ≤ R[j]
13          A[k] ← L[i]
14          i ← i + 1
15      else
16          A[k] ← R[j]
17          j ← j + 1
```

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
2      q ← Partition(A, p, r)
3      QuickSort(A, p, q − 1)
4      QuickSort(A, q + 1, r)
```

```
Partition(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4      if A[j] ≤ x
5          i ← i + 1
6          Exchange A[i] ↔ A[j]
7  Exchange A[i + 1] ↔ A[r]
8  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:

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

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

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
3      exchange A[i] ↔ A[Random(i, n)]
```

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", mentre l'algoritmo "quicksort" rimane uguale. Lo pseudocodice è:

RandomizedPartition(A, p, r)

```
1  i ← RANDOM(p, r)
2  EXCHANGE A[r] ↔ 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)
```

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
2        C[i] ← 0
3  for j ← 1 to A.lenght
4        C[A[j]] ← C[A[j]] + 1
5  for i ← 1 to k
6        C[i] ← C[i] + C[i − 1]
7  for j ← A.lenght downto 1
8        B[C[A[j]]] ← A[j]
9        C[A[j]] ← C[A[j]] − 1
```

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
2        "usa ordinamento stabile per ordinare l'array A sulla cifra i"
```

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.

Costo → Definisco il fattore di caricamento $\alpha = \frac{n}{m}$, ovvero il numero medio di elementi in ogni lista collegata, con n numero di elementi memorizzati e m dimensione della tabella hash.

Caso Peggiore → tutte le n chiavi nello stesso slot, cioè si ha una singola lista di dimensione n:

$$\Theta(1) + \Theta(n) = \Theta(n)$$

Caso Medio → Impongo come ipotesi di avere una funzione hash uniforme semplice (cioè per ogni elemento ognuno degli m slot è ugualmente probabile come hash). Definisco n_j, dimensione della j-esima lista concatenata. Il suo valore atteso sarà $E[n_j] = \alpha = \frac{n}{m}$. Inoltre il calcolo della funzione hash avviene in $O(1)$.

Ricerca Senza Successo → È necessario ricercare fino alla fine di ogni lista, quindi aggiungendo il costo del calcolo dell'hash avrò:

$$\Theta(\alpha + 1)$$

Ricerca Con Successo → Devo compiere un'analisi probabilistica sul numero di elementi esaminati $n_x + 1$ ("+1" poiché esamino anche l'elemento che sto cercando x). Ottengo che il costo medio è:

$$\Theta(2 + \tfrac{\alpha}{2} - \tfrac{\alpha}{2n}) = \Theta(1 + \alpha)$$

$$\text{Se } n = O(m) \implies \alpha = O(1) \implies \Theta(1)$$

Indirizzamento Aperto → Memorizza tutte le chiavi nella tabella hash, ogni slot contiene una chiave o NIL. La funzione hash è nella forma $h(k, i) = (h'(k) + i) \mod m$ (è una delle possibili forme). Gli pseudocodici di inserimento e ricerca sono:

HashInsert(T, k)

```
1  i ← 0
2  repeat
3        j ← h(k, i)
4        if T[j] = NIL
5              T[j] ← k
6              return j
7        else
8              i ← i + 1
9  until i = m
10 error "hash table overflow"
```

HashSearch(T, k)

```
1  i ← 0
2  repeat
3       j ← h(k, i)
4       if T[j] = k
5            return j
6       i ← i + 1
7  until T[j] = NIL OR i = m
8  return NIL
```

Per quanto riguarda la cancellazione si può sostituire il valore da cancellare con "DEL" che viene visto dalla ricerca come una chiave diversa da quella cercata e dall'inserimento come uno slot vuoto (il costo non dipenderà più da α).

Hash Uniforme → Ciascuna chiave ha la stessa probabilità che generi una delle $m!$ possibili sequenze di esplorazione. È difficile da implementare, quindi si approssima garantendo che la sequenza di esplorazione sia una delle $m!$ permutazioni. Per questo è necessario utilizzare funzioni hash ausiliari $h'() : U \to \{0, 1, ..., m-1\}$.

Esplorazione Lineare → La funzione hash, data la chiave k e il numero di esplorazione i, è nella forma:

$$h(k,i) = (h'(k) + ci) \mod m$$

Si hanno solo m possibili sequenze e non $m!$. Si rischia di avere "**clustering primario**" cioè lunghe sequenze di slot occupati.

Esplorazione Quadratica → La funzione hash, data la chiave k e il numero di esplorazione i, è nella forma:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$$

c_1, c_2 devono essere costanti e $\neq 0$ e vanno scelti affinchè si abbia una permutazione completa (ES: $c_1 = \frac{1}{2}, c_2 = \frac{1}{2}$).Si rischia di avere "**clustering secondario**", cioè se $h'(k) = h'(k')$ con $k \neq k'$ allora le due chiavi avranno la stessa sequenza di esplorazione.

Doppio Hash → Si va a definire il passo dell'esplorazione lineare usando una seconda funzione hash, la forma è:

$$h(k,i) = (h'_1(k) + h'_2(k)i) \mod m$$

h'_2 deve essere scelto affinché sia "primo" rispetto ad m (non devono avere fattori comuni), per garantire che la sequenza di esplorazione una permutazione completa (ES: $h'_1(k) = k \mod m$, $h'_2(k) = 1 + (k \mod m')$). Si hanno $\Theta(m^2)$ diverse sequenze di esplorazione (molto meglio di m).

Metodo delle Moltiplicazioni → Data una costante $A : 0 < A < 1$ la funzione hash è:

$$\boldsymbol{h(k) = \lfloor m(kA \mod 1) \rfloor}$$

dove $kA \mod 1$ = "parte frazionaria di kA". Nonostante questo metodo sia più lento di quello delle divisioni, il valore di m non è critico. La sua implementazione in base 2 è molto semplice infatti basta moltiplicare k e A per poi selezionare dal risultato i p bit più significativi della parte frazionaria ($p = \log_2 m$). Anche la scelta di A è importante (Knuth → $A \approx \frac{\sqrt{5}-1}{2}$).

Hash Randomizzato → Supponendo che la scelta delle chiavi sia affidata ad un avversario sleale, tutte le chiavi potrebbero essere inviate nello stesso slot. In questo caso sarebbe vantaggioso scegliere randomicamente la funzione la funzione hash da usare ogni volta che si inizia una nuova tabella (non ad ogni chiave inserita).

Hash Universale → La funzione hash ha la forma:

$$h_{a,b} = ((a \cdot k + b) \mod p) \mod m$$

dove p è un numero primo t.c. $\forall k\ 0 \leq k \leq p-1$ e dove a, b sono due numeri naturali t.c. $0 \leq a \leq p-1$ e $0 \leq b \leq p$. In questo modo per ogni diverso m si ha una famiglia di $p(p-1)$ diverse hash.

Hash Perfetto → Si basa sull'ipotesi di avere un insieme di chiavi statico. Provo diverse funzioni hash fino a trovarne una che non genera collisioni. La complessità è $O(1)$ nel caso peggiore. Se m è troppo grande o se devo provare la funzione hash troppe volte posso usare due livelli di hash:

1° livello → è un hash con concatenamento.

2° livello → creo tabelle hash per ogni lista concatenata $T[i]$ del 1° livello di dimensione n_i^2 (dove n_i è il numero di elementi nella i-esima lista concatenata). Sul 2° livello devo provare varie funzioni hash affinchè non si abbiano collisioni (la probabilità di averne è meno di metà della funzione hash perfetto ad un livello).

ANALISI AMMORTIZZATA

Si studia il tempo per eseguire una sequenza di operazioni (diverse) su una struttura dati. In questo caso non si fa una media su una distribuzione di input ma considero sempre il costo medio del caso peggiore per una sequenza di n operazioni.

Tabelle Dinamiche → Data una tabella hash di dimensione m con n che varia, ho che se:

$\alpha = \frac{n}{m} = 1$ → rialloco la tabella, stavolta con dimensione $m' > m$, e copio gli oggetti;

$\alpha = \frac{n}{m} \leq 0.5$ → alloco la tabella, con dimensione $m' < m$, e copio gli oggetti.

È molto utile avere una tabella dinamica in quanto riduce lo spazio inutilizzato e permette di avere un costo ammortizzato per un'operazione di $O(1)$.

$DynamicTableInsert(T, x)$

```
1  if T.size = 0
2       "alloca tabella T.size con 1 elemento"
3       T.size ← 1
4  if T.num = T.size
5       "alloca NT (new table) con 2 · T.size elementi"
6       "inserisci gli elementi di T.table in NT"
7       free T.table
8       T.table ← NT
9       T.size ← 2T.size
10 "inserisci x in T.table"
11 T.num ← T.num + 1
```

Costo

Supponendo di compiere solo inserimenti avrò che costo della i-esima operazione $c_i = 1$ se la tabella è non piena, mentre $c_i = i$ se la tabella è piena (copio i elementi). Quindi $c_i = O(i)$ in generale. Invece nel caso peggiore, considerando n operazioni di costo c_i avrò che il costo complessivo è $O(n^2)$. Infine per compiere un'analisi esatta (considerando solo le espansioni della tabella che realmente avvengono) avrò che il costo per n operazioni è $< 3n$.

ALBERI BINARI DI RICERCA

È una struttura dati spesso usata come dizionario e che supporta operazioni dinamiche. Esegue le operazioni base in $O(h)$ (dove h è l'altezza dell'albero). Viene rappresentato con una struttura dati collegata T dove ogni nodo è un oggetto.

Campi → $root$ (punta alla radice dell'albero), key (il valore del nodo), $left$ (punta al figlio sinistro), $right$ (punta al figlio destro), p (punta al padre($T.root$ non ha padre)).

Proprietà → se y è nel sottoalbero sinistro di x allora $y.key \leq x.key$, se y è nel sottoalbero destro di x allora $y.key \geq x.key$,

$TreeInsert(T, z)$

```
1  y ← NIL        /Tiene traccia del padre
2  x ← T.root
3  while x ≠ NIL
4       y ← x
5       if z.key < x.key
6            x ← x.left
7       else x ← x.right
8  z.p ← y
9  if y = NIL
10      T.root ← z        /L'albero T era vuoto
11 elseif z.key < y.key
12      y.left ← z
13 else y.right ← z
```

Costo

Il costo dipende esclusivamente dall'altezza h dell'albero, quindi il costo è $O(h)$. Nel caso migliore $h = \log_2 n$ mentre nel caso peggiore $h = n$. L'altezza dipende soltanto dall'ordine di inserimento delle chiavi.

$TreeSearch(T.root, k)$ $\quad VersioneRicorsiva$

```
1  if x = NIL OR k = x.key
2       return x
3  if k < x.key
4       return TreeSearch(x.left,k)
5  else return TreeSearch(x.right,k)
```

Costo

Anche in questo caso il costo dell'algoritmo dipende esclusivamente dall'altezza dell'albero, infatti il costo è $O(h)$.

```
TreeSearch(x, k)      VersioneIterativa
1  while x ≠ NIL AND k ≠ x.key
2        if k < x.key
3              x ← x.left
4        else x ← x.right
5  return x
```

```
TreeMinimum(x)
1  while x.left ≠ NIL
2        x ← x.left
3  return x
```

La chiave minima si trova sempre nel nodo più a sinistra. **Costo** $\rightarrow O(h)$

```
TreeMaximum(x)
1  while x.right ≠ NIL
2        x ← x.right
3  return x
```

La chiave massima si trova sempre nel nodo più a destra. **Costo** $\rightarrow O(h)$

Attraversamento di un Albero Binario → Per gli alberi binari è possibile effettuare un'esplorazione per stampare direttamente i valori ordinati (grazie alle sue proprietà). Esistono 3 modalità di "attraversamento" possibili, cioè:

```
InorderTreeWalk(x)
1  if x ≠ NIL
2        InorderTreeWalk(x.left)
3        Stampa x.key
4        InorderTreeWalk(x.left)
```

Per le altre due versioni, Preorder e Postorder, lo pseudocodice è lo stesso con l'unica differenza che la stampa del valore avviene rispettivamente prima o dopo le due chiamate ricorsive a "TreeWalk".

Costo

Intuitivamente il costo è $\Theta(n)$ in quanto ogni nodo viene visitato una sola volta, ma è possibile dimostrare che il costo è $O(n)$.

Successore/Predecessore di una Chiave → Il successore di x è la più piccola chiave $> x.key$ mentre il predecessore di x è la più grande chiave $< x.key$. E' anche possibile che $x.key$ sia il massimo/minimo dell'albero e che quindi non abbia successore/predecessore. Possiamo modellizzare il problema come:

- **x ha il sottoalbero destro non-vuoto** → Il successore di x è il minimo del sottoalbero destro di x;
- **x ha il sottoalbero destro vuoto** → Il successore di x è l'antenato più prossimo di x che ha come figlio sinistro è anch'esso antenato di x.

Lo pseudocodice è:

```
TreeSuccessor(x)
1  if x.right ≠ NIL
2        return TreeMinimum(x.right)
3  y ← x.p
4  while y ≠ NIL AND x = y.right
5        x ← y
6        y = y.p
7  return y
```

Per "TreePredecessor" l'algoritmo è lo stesso, basta cambiare "x.right" con "x.left".

Costo

Il costo dipende anche in questo caso esclusivamente dell'altezza infatti è $O(n)$.

Trapianto di Sottoalberi → Un trapianto sostituisce il sottoalbero con radice u con quello di radice v (N.B. Non fa uno scambio). Lo pseudocodice è:

```
Transplant(T, u, v)
1  if u.p = NIL
2        T.root ← v        /u era la radice di T
3  elseif u = u.p.left
4        u.p.left ← v
5  else u.p.right ← v
6  if v ≠ NIL
7        v.p ← u.p
```

Cancellazione → Per effettuare la cancellazione di una chiave z da un albero, possiamo trovarci un due casi, cioè:

- **z non ha un figlio** → cancello z e aggiusto il puntatore del padre di z.
- **z ha un figlio** → cancello z e faccio in modo che il padre di z punti al figlio di z.
- **z ha due figli** → devo trovare il successore di z che chiamo y che sarà il minimo del sottoalbero destro di z e riordinare il sottoalbero, aggiustando i collegamenti con il padre di z.

```
TreeDelete(T, z)
1  if z.left = NIL
2        Transplant(T,z,z.right)
3  elseif z.right = NIL
4        Transplant(T,z,z.right)
5  else
6        y ← TreeMinimum(z.right)
7        if y.p ≠ z
8              Transplant(T,y,y.right)
9              y.right ← z.right
10             y.right.p ← y
11       Transplant(T,z,y)
12       y.left ← z.left
13       y.left.p ← y
```

Chiavi duplicate → Se si hanno delle chiavi duplicate da inserire in un ABR nell'algoritmo "TreeInsert" è necessario aggiungere dei controlli, cioè:

- aggiungere $if(z.key = x.key)$... prima della riga 5;
- aggiungere $if(z.key = y.key)$... prima della riga 11.

Ci sono dunque diversi modi di assegnare chiavi duplicate come: fare un lista concatenata con valori uguali, aggiungere un flag booleano nel nodo x e assegnare la chiave duplicata a $x.left$ o $x.right$ a seconda del valore del flag (che cambia ad ogni vista al nodo x) oppure assegnare casualmente la chiave duplicata ad $x.left$ o $x.right$.

Notazione Parentesizzata → È possibile rappresentare un ABR tramite una notazione alternativa. Ad esempio, avendo un albero composto da x, $x.left$ e $x.right$ posso scriverlo come:

$$x(x.left, x.right)$$

Ponendo $x.left = y$ e $x.right = z$ e supponendo che entrambi abbiano solo un figlio sinistro posso scrivere:

$$x(y(y.left,\ ), z(z.left,\ ))$$

ALBERI AVL

Gli alberi AVL sono alberi quasi bilanciati", infatti ,per qualsiasi nodo, le altezze dei due sottoalberi differiscono al più di 1. Questi alberi aggiungono anche un fattore di bilanciamento ad ogni nodo. Le sue proprietà principali sono:

Proprietà ereditate dagli ABR → gli alberi AVL ereditano tutte le proprietà degli ABR;

Altezza dei Nodi → ad ogni nodo si aggiunge un parametro "altezza" che indica appunto il massimo numero di archi dal nodo considerato ad una foglia;

Fattore di Bilanciamento → ad ogni nodo x si aggiunge un fattore di bilanciamento $|BF(x)| \leq 1$ e si calcola facendo $BF(x) = x.left.h - x.right.h$, è cioè la differenza tra l'altezza sottoalbero sinistro e l'altezza del sottoalbero destro.

Sentinella T.NIL → le foglie, ovvero i nodi ad altezza 0, non hanno una chiave e vengono rappresentate come un unico nodo, collegato a tutti i nodi ad altezza 1.

Altezza Massima → l'altezza massima di un AVL con n nodi interni è $\mathbf{2\log_2(n+1)}$;

Teorema : Chiamo $n(h)$ il numero minimo di nodi che un albero di altezza h deve avere. Per definizione avrò che: $n(h) \geq 2^{h/2} - 1$ per ogni $h > 1$.

Dimostrazione :

h=1 → $n(1) = 1 > 2^{1/2} - 1 = \sqrt{2} - 1$

h=2 → $n(2) = 2 > 2^{1/1} - 1 = 1$

h≥2 → (per induzione) suppongo che i due sottoalberi differiscano di un nodo, quindi uno avrà altezza $h-1$ e l'altro $h-2$. Applicando i la formula dei nodi minimi data dal teorema visto sopra avrò che:

$$n(h) \geq \underbrace{1}_{\text{nodo esaminato}} + \underbrace{n(h-1) + n(h-2)}_{\text{numero nodi sottoalberi}} =$$

$$= \underbrace{1}_{\text{nodo esaminato}} + \underbrace{2^{h-1/2} - 1 + 2^{h-2/2} - 1)}_{\text{numero nodi sottoalberi}} =$$

$$= (2^{-1/2} + 2^{-1})2^{h/2} - 1 =$$

$$= \frac{1}{2\sqrt{2}} 2^{h/2} - 1 > 2^{h/2} - 1.$$

Ora ricavando h rispetto ad n ottengo:

$$n + 1 \geq 2^{h/2} \implies \log_2(n+1) = h/2 \implies h = 2\log_2(n+1) \quad \text{C.V.D.}$$

Gli AVL ereditano anche tutte le operazioni tipiche degli ABR, cioè MIN, MAX, SUCCESSOR, PREDECESSOR e SEARCH, ma il costo diventa dipendente dal numero di nodi infatti si eseguono in $O(log_2\ n)$. Ci sono invece altre operazioni, come inserimento e cancellazione, che rischiano di violare le proprietà dell'AVL. A questo scopo si introducono le ROTAZIONI che permettono di ripristinare, se usate in modo corretto, il bilanciamento dei nodi. Lo pseudocodice è:

$LeftRotate(T, x)$

```
1  y ← x.right        /assegna y, che verrà ruotato insieme ad x
2  x.h ← MAX(x.left.h,x.right.h) + 1
3  if y.left ≠ T.NIL
4        y.left.p ← x        /collega x e il sottoalbero sinistro di y
5  y.p ← x.p        /collega il padre di x come padre di y
6  if x.p = T.NIL        /collega il padre di x a y come padre
7        T.root ← y
8  elseif x = x.p.left
9        x.p.left ← y
10 else
11       x.p.right ← y
12 y.left ← x        /collega x come figlio di y
13 y.h ← MAX(y.left.h,y.right.h) + 1
14 x.p ← y        /collega y come padre di x
```

AVL$Insert(T, z)$

```
1  y ← T.NIL        /tiene traccia del padre
2  x ← T.root
3  while x ≠ T.NIL        /scorre fino a trovare la foglia dove inserire x
4        y ← x
5        if z.key < x.key
6              x ← x.left
7        else
8              x ← x.right
9  z.p ← y        /assegna il padre alla chiave da inserire
10 if y = T.NIL        /Assegna z come il giusto figlio di y
11       T.root ← z
12 elseif z.key < y.key
13       y.left ← z
14 else
15       y.right ← z
16 z.left ← T.NIL        /Crea le due foglie NIL di z
17 z.right ← T.NIL
18 z.h ← 1
19 AVLInsertFixup(T,z)        /chiamata alg. che aggiusta il bilanciamento
```

Quando si inserisce/cancella una nuova foglia z bisogna aggiustare il bilanciamento di qualche x antenato di z. Infatti esisterà almeno un nodo x per cui $-2 \leq BF(x) \leq 2$, violando cioè le regole degli AVL. Esistono quindi due possibilità: Posso dividere il problema in 4 sottocasi:

- **se $BF(x) = 2$** → il sottoalb. sinistro è due livelli più alto di quello destro;
  - **z inserito nel sottoalb. x.left.left** → allora avrò che $BF(x.right) = 1 \implies$ RightRotate su x;
  - **z inserito nel sottoalb. x.left.right** → allora avrò che $BF(x.left) = -1 \implies$ LeftRotate su y poi RightRotate su x.
- **se $BF(x) = 2$** → il sottoalb. destro è due livelli più alto di quello sinistro;
  - **z inserito nel sottoalb. x.right.left** → allora avrò che $BF(x.right) = 1 \implies$ LeftRotate su x;
  - **z inserito nel sottoalb. x.right.right** → allora avrò che $BF(x.left) = -1 \implies$ RightRotate su y poi LeftRotate su x;

Lo pseudocodice è:

AVL$Insertfixup(T, x)$

```
1  x ← x.p
2  while x ≠ T.NIL
3        x.h ← MAX(x.left.h,x.right.h) + 1
4        if (x.left.h − x.right.h) = 2
5              if (x.left.left.h − x.left.right.h) = −1
6                    LeftRotate(T,x.left)
7              RightRotate(T,x)
8              x ← x.p
9        elseif (x.left.h − x.right.h) = −2
10             if (x.right.left.h − x.right.right.h) = 1
11                   RightRotate(T,x.right)
12             LeftRotate(T,x)
13             x ← x.p
14       x ← x.p
```

Costo
Il costo di AVLInsertFixup è $O(1)$ per ogni iterazione, essendo $O(\log_2 n)$ iterazioni (corrispondenti al numero di livelli dell'albero) il costo totale di AVLInsertFixup è $O(\log_2 n)$. Il costo di AVLInsert è $O(\log_2 n)$ fino all'esecuzione di AVLInsertFixup. Il costo complessivo è comunque $O(\log_2 n)$.
Correttezza con I.C.
"All'inizio di ogni iterazione, in AVL c'è al massimo una violazione del bilanciamento di AVL" (non è necessario dimostrarla).

STRUTTURE DATI AUMENTATE

Raramente progetto una struttura dati da zero, ma spesso ho bisogno di aggiungere informazioni e quindi anche nuove operazioni ad una struttura dati esistente. L'aspetto fondamentale delle strutture dati aumentate è quello di aggiungere nuove operazioni senza perdere di efficienza.

- -

Statistiche d'Ordine Dinamiche

Voglio aggiungere le funzioni:

OS-Select(x,i) → ritorna il puntatore al nodo che contiene la i-esima chiave più piccola del sottoalb. con radice x;

OS-Rank(T,x) → ritorna la posizione (rango) di x nell'ordine determinato da un attraversamento Inorder di T.

Posso usare l'attraversamento Inorder, ma questo comporta un aumento del costo (che diventa $O(n)$) oppure posso aumentare la struttura aggiungendo ***x.size*** cioè il numero di nodi nel sottoalb. con radice x (includo x ma escludo le foglie NIL). Avrò dunque che:

$$x.size \leftarrow x.left.size + x.right.size + 1$$

Gli pseudocodici sono:

$OS - Select(x, i)$

```
1  r ← x.left.size + 1
2  if i = r
3        return x
4  elseif i < r
5        return OS-Select(x.left, i)
6  else
7        return OS-Select(x.right, i − r)
```

Costo
Ad ogni chiamata ricorsiva scende di un livello nell'albero, che ha $O(\log_2 n)$ livelli, quindi il costo complessivo è $O(\log_2 n)$.
Correttezza
Poiché è ricorsivo in coda posso considerarlo come un alg. iterativo e usare l'I.C. per dimostrarne la correttezza:
"Prima di ogni iterazione l'i-esimo valore si trova nel sottoalb. con radice in x".
Dato r rango di x avrò che:

- se $i = r \rightarrow$ ritorna $x \rightarrow$ OK;
- se $i < r \rightarrow$ i-esimo elemento più piccolo è nel sottoalb. sinistro;
- se $i > r \rightarrow$ i-esimo elemento più piccolo è nel sottoalb. destro $\rightarrow$ sottraggo gli r elementi che precedono quelli nel sottoalb. destro di x.

$OS - Rank(T, x)$

```
1  r ← x.left.size + 1
2  y ← x
3  while y ≠ T.root
4        if y = x.p.right
5              r ← x.p.left.size + 1
6        y ← x.p
7  return r
```

Costo
y sale di un livello ad ogni iterazione quindi anche in questo caso il costo è $O(\log_2 n)$.
Correttezza con I.C.
"All'inizio di ogni iterazione del ciclo ho che $r =$ Rank$(x.key, y)$ "

- -

Attributo Size

Suppongo di aggiungere alla struttura degli AVL un attributo "$size$". L'obbiettivo è quello di mantenere il costo della struttura, cioè $O(\log_2(n))$. Per farlo, senza aggiungere una funzione per aggiustare la "$size$" dei sottoalberi posso direttamente farlo dopo Inserimento e Cancellazione. Inoltre anche le rotazioni modificano "$size$".

Inserimento :

- incrementa "$size$" di ogni nodo visitato;
- nel fixup modifica "$size$" dopo le rotazioni.

Dopo LEFTROTATE
...
$y.size \leftarrow x.size$
$x.size \leftarrow x.left.size + x.right.size + 1$
Dopo RIGHTROTATE
...
$x.size \leftarrow y.size$
$y.size \leftarrow y.left.size + y.right.size + 1$

Cancellazione :

- decrementa "$size$" dal nodo cancellato a $T.root$;
- nel fixup modifica "$size$" dopo le rotazioni (senza entrare nel dettaglio).

Come Aumentare una Struttura Dati

Teorema
Se si aumenta un albero AVL con un campo f, dove $x.f$ dipende solo da informazioni in x, $x.left$, $x.right$ allora si possono mantenere i valori di f in tutti i nodi in $O(\log_2(n))$.
Dimostrazione
La modifica di $x.f$ si propaga solo agli antenati di x, che vengono aggiornati al costo di $O(1)$ ciascuno, in totale $O(\log_2(n))$.
Applicazione

Inserimento :

- aggiusto la statistica f in tutti i nodi visitati (non sempre posso farlo in scendendo ma posso sempre farlo salendo);
- aggiusto la statistica f dopo tutte le rotazioni sui nodi x,y,padre (costo O(1)) e se necessario la aggiusto anche su tutti i predecessori (costo complessivo $O(log_2(n))$).

Cancellazione :

- aggiusto la statistica f in tutti gli antenati del nodo rimosso;
- aggiusto la statistica dopo tutte le rotazioni (al massimo 3) sui nodi x,y,padre (costo O(1)) e se necessario la aggiusto anche su tutti i predecessori (costo complessivo $O(log_2(n))$).

Alberi di Intervalli

Servono a gestire un insieme di intervalli. Si aumenta quindi una struttura dati aggiungendo l'attributo $x.int$ che a sua volta ha attributi $int.low$ e $int.high$.
Uso per comodità gli alberi AVL e impongo $x.key$ come l'estremo inferiore dell'intervallo (affinché un InorderTreeWalk stampi gli intervalli in ordine rispetto all estremo inferiore), dunque $x.int$ conterrà solo l'estremo superiore dell'intervallo. Inoltre aggiungo anche un attributo $x.max = \max(x.int,\ x.left.max,\ x.right.max)$ che rappresenta il massimo estremo superiore presente nel sottoalbero con radice x. Come fatto anche per l'attributo $size$, posso aggiornare $x.max$ durante la discesa per l'inserimento in tempo $O(1)$ per ogni rotazione che effettuo.

Sovrapposizione di intervalli : Si dice che due intervalli i, j si sovrappongono sse $i.low \leq j.high$ AND $j.low \leq i.high$ mentre non si sovrappongono sse $i.low > j.high$ OR $j.low > i.high$.

Sviluppo di nuove operazioni : Utilizzando gli attributi dei nodi appena visti è possibile sviluppare nuove operazioni come ad esempio la ricerca. Lo pseudocodice è:

$IntervalSearch(T, i)$
1 $x \leftarrow T.root$
2 **while** $x \neq T.$ NIL AND i non si sovrappone a $x.int$
3 **if** $x.left \neq T.$ NIL AND $x.left.max \geq i.low$
4 $x \leftarrow x.left$
5 **else**
6 $x \leftarrow x.right$
7 **return** x

Costo
Ho costruito la struttura dati affinché il costo sia $O(\log_2 n)$
Correttezza
L'algoritmo restituisce un nodo il cui intervallo si sovrapponga con i oppure restituisce $T.$ NIL se non ci sono nodi il cui intervallo si sovrapponga con i. In altre parole se la ricerca va a destra (sinistra) allora c'è un intervallo che si sovrappone con i nel sottoalb. destro (sinistro) oppure non esiste un intervallo che si sovrapponga con i.
(Vedi dimostrazione ...)

PROGRAMMAZIONE DINAMICA

Fino a questo momento ho usato la logica "Divide et Impera" poiché scomponendo un problema ottenevo sottoproblemi indipendenti. La programmazione dinamica invece si occupa di problemi che hanno sottoproblemi non indipendenti, comuni a più parti del problema. Questi sottoproblemi vengono salvati in una tabella per essere riutilizzati più volte. Spesso viene usata nell'ambito dell' ottimizzazione. La procedura per la risoluzione di questo tipo di sottoproblemi è:

- individuare la struttura della soluzione ottima;
- definire ricorsivamente la soluzione ottima;
- calcolare la soluzione ottima in modo bottom-up (dai sottoproblemi più semplici a quelli più complessi);
- costruire una soluzione ottima del problema richiesto.

Taglio delle Aste

Supponiamo che il prezzo di un'asta dipenda dalla sua lunghezza, un'asta lunga n può essere tagliata in pezzi più corti che hanno valore diverso. L'obbiettivo è trovare i tagli che permettano di massimizzare il ricavato.
Per risolvere questo problema utilizzando la programmazione dinamica devo:

- Individuare la struttura della soluzione ottima:
  $\boldsymbol{r_n = r_i + r_{n-i}}$ ← suddivido il problema in due sottoproblemi dove r_n è il ricavo ottimo che voglio ottenere, e r_i, r_{n-i} sono i due ricavi parziali ottimi ottenuti da un taglio in posizione i. La soluzione si ottiene quindi calcolando $\boldsymbol{\max\ (p_n, r_1 + r_{n-1}, ..., r_{n-1} + r_1)}$;
  $\boldsymbol{r_n = p_i + r_{n-i}}$ ← scompongo il problema in una costante p_i, ovvero il prezzo di un primo pezzo di asta tagliata in i e il ricavato ottimo della restante parte dell'asse. La soluzione si ottiene calcolando $\boldsymbol{\max_{1 \leq i \leq n} (p_i + r_{n-i})}$.

- Definire ricorsivamente la soluzione ottima:
  Usando la 2° formulazione Posso definire un algoritmo ricorsivo che risolva il problema. Lo pseudocodice è:

  $CutRod(p, n)$
  1 **if** $n = 0$
  2 **return** 0
  3 $q \leftarrow -\infty$
  4 **for** $i \leftarrow 1$ **to** n
  5 $q \leftarrow \max\ (q, p[i] + \text{CUTROD}(p, n - i))$
  6 **return** q

  Costo
  Considerando tutti i possibili tagli dell'asta, ovvero 2^{n-1}, il costo del problema è esattamente $T(n) = \Theta(2^n)$ poiché si risolvono tutti i sottoproblemi, anche quelli che si presentano più volte.

- Calcolare la soluzione ottima (2 possibili metodi):
  Memorizzo i problemi risolti in una tabella affinché possano essere riutilizzati nel caso si presentassero più di una volta.

  Top-Down : Risolvo il problema di dimensione n e ricorsivamente risolvo i sottoproblemi più piccoli. Prima di effettuare la ricorsione verifico se nella tabella è già presente la soluzione del problema che voglio risolvere. Lo pseudocodice è:

  $MemoizedCutRod(p, n)$
  1 Sia $r[0, ..., n]$ un nuovo array
  2 **for** $i \leftarrow 0$ **to** n
  3 $r[i] \leftarrow -\infty$
  4 **return** MEMOIZEDCUTRODAUX (p,n,r)

  $MemoizedCutRodAux(p, n, r)$
  1 **if** $r[n] \geq 0$
  2 **return** r[n]
  3 **if** $n = 0$
  4 $q = 0$
  5 **else**
  6 $q \leftarrow -infty$
  7 **for** $i \leftarrow 1$ **to** n
  8 $q \leftarrow \max\ (q, p[i] + \text{MEMOIZEDCUTRODAUX}(p, n - i, r))$
  9 $r[n] \leftarrow q$
  10 **return** q

Costo
Ogni sottoproblema è risolto una sola volta, avrò in totale $n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$ iterazioni. Il costo asintotico è quindi $\Theta(n^2)$.

Bottom-Up : Risolvo i sottoproblemi partendo dai più piccoli, in modo che arrivando alla dimensione n tutti i problemi di dimensione $< n$ sono già stati risolti e memorizzati. Lo pseudocodice è:

BottomUpCutRod(p, n)
```
1  Sia r[0, ..., n] un nuovo array
2  r[0] ← 0
3  for j ← 1 to n
4        q ← −infty
5        for i ← 1 to j
6              q ← max (q, p[i] + r[j − i])
7        r[j] ← q
8  return r[n]
```

Costo
Svolgo due cicli annidati, avrò in totale $n + (n-1) + (n-2) + \cdots + 1 = \frac{n(n+1)}{2}$ iterazioni. Il costo asintotico è quindi $\Theta(n^2)$.

Gli algoritmi appena visti tengono traccia solo del massimo ricavo ma non dei tagli da effettuare per ottenerlo. Aggiungo un'array s che memorizzi i tagli ottimi. Lo pseudocodice è:

ExtendedBottomUpCutRod(p, n)
```
1  Siano r[0, ..., n] ed s[0, ..., n] due nuovi array
2  r[0] ← 0
3  for j ← 1 to n
4        q ← −infty
5        for i ← 1 to j
6              if q < p[i] + r[j − i]
7                    q ← p[i] + r[j − i]
8                    s[j] ← i
9        r[j] ← q
10 return (r, s)
```

PrintCutRodSolution(p, n)
```
1  (r, s) ← ExtendedBottomUpCutRod(p,n)
2  while n > 0
3        print s[n]
4        n ← n − s[n]
```

- -

LCS - Longest Common Sequence

Date 2 sequenze $X =< x_1, ..., x_m >$ e $Y =< y_1, ..., y_n >$ trovare una sottosequenza comune ad X, Y di lunghezza massima. Una sottosequenza è una sequenza a cui tolgo 0 o più elementi, infatti $Z =< 1, ..., z_k >$ è una sottosequenza di X se $\exists$ una sequenza di indici $I =< 1, ..., k >$ tali che $i_j > i_h$ se $j > h \ \forall j, h = 1, ..., k$ e $x_{i_j} = z_j \ \forall j = 1, ..., k$.
Per trovare la LCS tra X e Y potrei usare l'algoritmo "forza-bruta" analizzando tutte le possibili sottosequenze ma il costo sarebbe $\Theta(n \cdot 2^m)$ (molto costoso). Posso in alternativa ricorrere alla programmazione dinamica:

- Individuare la struttura della soluzione ottima:

  Notazione $\rightarrow$ Definisco il prefisso di una sequenza X, detto X_i, ovvero i primi i elementi della sequenza X.
  Teorema: Date X, Y due sequenze e Z una loro qualunque LCS, avrò che:
  1) se $x_m = y_n \implies z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1};
  2) se $x_m \neq y_n \implies$ se $z_k \neq x_m$, Z è una LCS di X_{m-1} e Y;
  3) se $x_m \neq y_n \implies$ se $z_k \neq y_n$, Z è una LCS di X e Y_{n-1}.

- Definire ricorsivamente la soluzione ottima:
  Posso distinguere due casi:

  $\boldsymbol{x_m = y_n}$ $\rightarrow$ devo risolvere un solo sottoproblema, ovvero trovare una LCS di X_{m-1} e Y_{n-1};
  $\boldsymbol{x_m \neq y_n}$ $\rightarrow$ devo risolvere due sottoproblemi, ovvero trovare una LCS di X_{m-1} e Y e una LCS di X e Y_{n-1}-. La più lunga delle due sarà la LCS di X e Y.

  Utilizzo una matrice c per memorizzare la LCS, $c[i, j]$ sarà la lunghezza della LCS di X_i e Y_j. L'obbiettivo è trovare $c[m, n]$ ovvero la lunghezza della LCS di X e Y.
  $$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i-1, j-1] + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max(c[i, j-1],\ c[i-1, j]) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$ Il vantaggio di questa formulazione è che al massimo devo risolvere due sottoproblemi mentre per il taglio delle aste li consideravo tutti. Inoltre, anche in questo caso, memorizzo i sottoproblemi comuni per evitare di calcolarli più di una volta.

- Calcolare la lunghezza della soluzione ottima:
  Creo un algoritmo che compila le tabelle b, che indica la direzione che ho seguito per risolvere i sottoproblemi (se $b[i, j] =\nwarrow$ ho esteso la LCS di un carattere), e c, che indica passo passo la lunghezza della LCS. Lo psudocodice è:

  $LCS - Length(X, Y)$
```
1  m ← X.length
2  n ← Y.length
3  Siano b[1, ..., m ; 1, ..., n] e c[0, ..., m ; 0, ..., n] due nuove tabelle
4  for i ← 1 to m
5        c[i, 0] ← 0
6  for j ← 0 to n
7        c[0, j] ← 0
8  for i ← 1 to m
9        for j ← 1 to n
10             if x_i = y_j
11                   c[i, j] ← c[i − 1, j − 1] + 1
12                   b[i, j] ← " ↖ "
13             elseif c[i − 1, j] ≥ c[i, j − 1]
14                   c[i, j] ← c[i − 1, j]
15                   b[i, j] ← " ↑ "
16             else
17                   c[i, j] ← c[i, j − 1]
18                   b[i, j] ← " ← "
19 return b, c
```

- Costruire una soluzione ottima:
  Usando le 2 tabelle b e c posso scrivere un algoritmo che individui e stampi la LCS. Lo pseudocodice è:

  $Print - LCS(b, X, i, j)$
```
1  if i = 0 OR j = 0
2        return
3  if b[i, j] = " ↖ "
4        Print-LCS(b,X,i-1,j-1)
5        print x_i
6  elseif b[i, j] = " ↑ "
7        Print-LCS(b,X,i-1,j)
8  else
9        Print-LCS(b,X,i,j-1)
```

- -

Edit Distance

L'obbiettivo è quello di trasformare la stringa X nella stringa Y usando solo operazioni elementari, come Inserimento, Cancellazione e Sostituzione di un carattere. Definisco $X =< x_1, ..., x_i >$ e $Y =< y_1, ..., y_j >$. Voglio minimizzare il numero delle operazioni per trasformare una stringa nell'altra, il loro numero minimo è la distanza tra le 2 stringhe. L'idea generale è quella di usare una tabella c per memorizzare i costi di tutte le successive operazioni, infatti,supponendo di conoscere il costo dell'ultima operazione, basterà aggiungere il costo della successiva. Esistono 6 casi distinti:

Copia $\rightarrow$ allora era $x_i = y_j$ e il sottoproblema rimanente è $X_{i-1} \rightarrow Y_{j-1}$ quindi il costo sarà $c[i, j] = c[i-1, j-1] +$ "costo copia";
Sostituzione $\rightarrow$ allora era $x_i \neq y_j$ e il sottoproblema rimanente è $X_{i-1} \rightarrow Y_{j-1}$ quindi il costo sarà $c[i, j] = c[i-1, j-1] +$ "costo sostituzione";
Scambio $\rightarrow$ allora era $x_i = y_{j-1}$ e $y_j = x_{i-1}$ e il sottoproblema rimanente è $X_{i-2} \rightarrow Y_{j-2}$ quindi il costo sarà $c[i, j] = c[i-2, j-2] +$ "costo scambio";
Cancellazione $\rightarrow$ allora non ho restrizioni su X, Y e il sottoproblema rimanente è $X_{i-1} \rightarrow Y_j$ quindi il costo sarà $c[i, j] = c[i-1, j] +$ "costo cancellazione";
Inserimento $\rightarrow$ allora non ho restrizioni su X, Y e il sottoproblema rimanente è $X_i \rightarrow Y_{j-1}$ quindi il costo sarà $c[i, j] = c[i, j-1] +$ "costo inserimento";
Casi Base $\rightarrow$ se $i = 0$ o $j = 0$ allora X_0 o Y_0 è una stringa vuota, quindi posso trasformare $X \rightarrow Y$ con j inserimenti o i cancellazioni. I rispettivi costi saranno:
$c[0, j] = j \cdot$ "costo inserimento"
$c[i, 0] = i \cdot$ "costo cancellazione"

Quindi usando le formule precedenti:

$$c[i,j] = \min \begin{cases} c[i-1, j-1] + \text{cost}(\text{COPY}) & \text{se } x[i] = y[j] \\ c[i-1, j-1] + \text{cost}(\text{REPLACE}) & \text{se } x[i] \neq y[j] \\ c[i-2, j-2] + \text{cost}(\text{TWIDDLE}) & \text{se } x, y \geq 2 \text{ AND } x[i-1] = y[j] \\ c[i-1, j] + \text{cost}(\text{DELETE}) & \text{sempre} \\ c[i-1, j] + \text{cost}(\text{INSERT}) & \text{sempre} \end{cases}$$

Scrivo quindi un algoritmo che usa le due tabelle c, dei costi, e op delle operazioni. Lo pseudocodice è:

```
EditDistance(x, y)
 1  m ← X.length
 2  n ← Y.length
 3  Siano c[0,...,m ; 0,...,n] e op[0,...,m ; 0,...,n] due nuove tabelle
 4  for i ← 0 to m
 5       c[i,0] ← i · cost(DELETE)
 6       op[i,0] ← "DELETE"
 7  for i ← 0 to m
 8       c[0,j] ← j · cost(INSERT)
 9       op[0,j] ← "INSERT"
10  for i ← 1 to m
11       for j ← 1 to n
12            c[i,j] ← ∞
13            if x_i = y_j
14                 c[i,j] ← c[i−1,j−1] + cost(COPY)
15                 op[i,j] ← "COPY"
16            if x_i ≠ y_j AND c[i−1,j−1] + cost(REPLACE) < c[i,j]
17                 c[i,j] ← c[i−1,j−1] + cost(REPLACE)
18                 op[i,j] ← "REPLACE"
19            if i,j ≥ 2 AND x_i = y_{j−1} AND y_j = x_{i−1} AND ...
20                 ... AND c[i−2,j−2] + cost(TWIDDLE) < c[i,j]
21                 c[i,j] ← c[i−2,j−2] + cost(TWIDDLE)
22                 op[i,j] ← "TWIDDLE"
23            if c[i−1,j] + cost(DELETE) < c[i,j]
24                 c[i,j] ← c[i−1,j] + cost(DELTE)
25                 op[i,j] ← "DELETE"
26            if c[i,j−1] + cost(INSERT) < c[i,j]
27                 c[i,j] ← c[i,j−1] + cost(INSERT)
28                 op[i,j] ← "INSERT"
29  return c, op
```

Costo
Il costo dell'EditDistance in termini di tempo è $\Theta(n \cdot m)$, dovuto ai due cicli annidati, lo stesso è il costo di spazio di memoria che infatti è $\Theta(n \cdot m)$.
Usando la tabella op restituita da EditDistance posso creare un algoritmo che restituisca la serie di operazioni. Lo pseudocodice è:

```
OpSequence(op, i, j)
 1  if i = 0 AND j = 0
 2       return
 3  if op[i,j] = "COPY" OR op[i,j] = "REPLACE"
 4       i' ← i − 1
 5       j' ← j − 1
 6  elseif op[i,j] = "TWIDDLE"
 7       i' ← i − 2
 8       j' ← j − 2
 9  elseif op[i,j] = "DELETE"
10       i' ← i − 1
11       j' ← j
12  else      /Ovvero op[i,j]="INSERT"
13       i' ← i
14       j' ← j − 1
15  OpSequence(op, i', j')
16  print(op[i,j])
```

Gli utilizzi della EditDistance sono molti, ad esempio: correzione automatica di parole errate (individuate grazie al contesto), suggerimenti di ricerca o correzione ortografica. Al questo scopo è spesso necessario confrontare una stringa con un insieme molto grande di stringhe, come ad esempio un dizionario o tutte le parole sul web, ma questo implicherebbe un costo proibitivo. Esiste modo per ridurre l'insieme dei candidati che esamino e si chiama "Intersezione di n-gram". Consiste nell'associare alla parola di partenza numerose sequenze di caratteri che abbiano una EditDistance minore di una certa soglia. Dal dizionario si estraggono solo parole che contengano abbastanza di quegli "n-gram". Per avere una sovrapposizione di n-gram normalizzata (utile per confronti molto grandi) posso usare il coefficente di Jaccard:
$JC = \frac{|X \bigcap Y|}{|X \bigcup Y|}$ con X, Y insiemi di dimensione diversa.

ALGORITMI ELEMENTARI PER GRAFI

Dato un grafo $G = (V, E)$, dove V è l'insieme dei vertici ed E è l'insieme degli archi, posso definire il **Grado** $u.degree$ di un vertice come il numero di archi che convergono in esso.

Grafo Diretto → E' un grafo i cui archi sono coppie di vertici ordinati;
Grafo Indiretto → E' un grafo i cui archi sono coppie di vertici non ordinati;
Cammino → Un cammino di lunghezza k è una sequenza di vertici $< v_0, v_1, ..., v_k >$ tale che $(v_{i-1}, v_i) \in E$ per $i = 0, ..., k$.
  Ciclo → in un grafo orientato, un cammino $< v_0, v_1, ..., v_k >$ forma un ciclo se $v_0 = v_k$ e il cammino contiene almeno un arco;
  Grafo Aciclico → un grafo che non contiene nessun ciclo è detto aciclico.
Componenti Connesse → è una classe di equivalenza dei vertici che verifica la relazione "è raggiungibile da";
Grafo Fortemente Connesso → un grafo viene detto fortemente connesso se due vertici qualsiasi sono raggiungibili l'uno dall'altro.

- -

Rappresentazione di Grafi

Un grafo $G = (V, E)$ questo può essere rappresentato in due modi:
Liste di Adiacenza → uso un vettore adj di $|V|$ liste, una per nodo. Ogni lista $adj[u]$ contiene i nodi raggiungibili dall'u-esimo nodo (per i grafi indiretti considero gli archi in entrambe le direzioni). Il peso dell'arco o qualsiasi attributo aggiuntivo viene memorizzato nella lista.
  Spazio → $\Theta(V + E)$
  Tempo → elencare tutti i nodi adiacenti ad u costa $\Theta(u.degree)$ mentre determinare se $(u, v) \in E$ costa $O(u.degree)$.
Matrice di Adiacenza → uso una matrice $|V| \times |V|$ dove l'elemento $a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{se } (i,j) \notin E \end{cases}$
  Spazio → $\Theta(V^2)$
  Tempo → elencare tutti i nodi adiacenti ad u costa $\Theta(V)$ mentre determinare se $(u, v) \in E$ costa $O(1)$.

- -

Visita in Ampiezza

Scopre tutti i vertici raggiungibili da un nodo sorgente $s \in V$. Come input ho il grafo G e il nodo sorgente s mentre come output avrò $\forall v \in V$ la distanza $v.d$ da s a v e il predecessore $v.\pi = u$ di v ovvero il penultimo nodo nel cammino da s a v. L'insieme degli archi $\{v.\pi, v\} talechev \neq s$ forma un albero. Per la Visita in Ampiezza uso una coda "FIFO" Q con Enqueue e Dequeue. Si può mostrare che i valori di d presenti in Q sono al massimo 2 e se sono due, quelli più piccoli sono stati inseriti prima, quindi saranno i primi ad uscire. Lo pseudocodice è:

```
BFS(G, s)
 1  for ogni vertice u ∈ G.V − {s}
 2       u.color ← WHITE
 3       u.d ← ∞
 4       u.πgetNIL
 5  s.color ← GRAY       /Imposto la sorgente
 6  s.d ← 0
 7  s.π ← NIL
 8  Q ← ∅       /inizializzo la coda FIFO
 9  Enqueue(Q, s)
10  while Q ≠ ∅
11       u ← Dequeue(Q)       /estrae il nodo corrente e lo salva
12       for ogni v ∈ G.adj[u]       /esplora i nodi adiacenti a quello corrente
13            if u.color = WHITE
14                 u.color ← GRAY
15                 v.d ← u.d + 1
16                 v.π ← u
17                 Enqueue(Q, v)
18       u.color ← BLACK       /imposta il nodo corrente come esplorato
```

Costo
Il tempo di esecuzione complessivo sarà $O(V + E)$ poiché Enqueue e Dequeue costano $O(1)$, ogni nodo è messo nella coda al massimo una volta e ogni arco viene esplorato al massimo una volta se il grafo è diretto (o al massimo due volte sei il grafo è indiretto). Inoltre si usa O e non Θ perché alcuni nodi potrebbero non essere esplorati.

- -

Visita in Profondità

Esplora sistematicamente ogni arco, anche ri-iniziando da nodi diversi. Come input ho il grafo G mentre come output avrò $\forall v \in V$ il tempo di scoperta $v.d$, il tempo di fine $v.f$ e il predecessore $v.\pi$. I tempi di scoperta e di fine rispettano la relazione $1 \leq v.d < v.f \leq 2\,|V|$. Lo pseudocodice è:

```
DFS(G)
```

```
1  for ogni vertice u ∈ G.V
2      u.color ← WHITE
3      u.π ← NIL
4  time ← 0
5  for ogni vertice u ∈ G.V      /usa tutti i nodi come sorgente
6      if u.color = WHITE
7          DFS-Visit(G, u)
```

$DFS - Visit(G, u)$

```
1  time ← time + 1      /aggiorna il tempo corrente
2  u.d ← time      /inizia l'esplorazione del nodo corrente
3  u.color ← GRAY
4  for ogni v ∈ G.adj[u]      /esplora tutti i nodi raggiungibili dal corrente
5      if v.color = WHITE
6          v.π ← u
7          DFS-Visit(G, v)
8  u.color ← BLACK      /conclude l'esplorazione del nodo corrente
9  time ← time + 1      /aggiorna il tempo corrente
10 attribuf ← time
```

Costo
Esplora ogni nodo e ramo quindi il costo totale sarà $\Theta(V + E)$.

Teorema delle Parentesi (Successori e predecessori) → durante una DFS di $G = (V, E)$, $\forall$ coppia di vertici (u, v) è soddisfatta solo una delle seguenti condizioni:

- Gli intervalli $[u.d, u.f]$ e $[v.d, v.f]$ sono completamente disgiunti (cioè non si sovrappongono) allora **nessuno dei due è discendente/predecessore dell'altro** in un albero DF;
- L'intervallo di v è completamente contenuto in quello di u allora ***v* è un discendente di *u*** in un albero DF;
- L'intervallo di u è completamente contenuto in quello di v allora ***u* è un discendente di *v*** in un albero DF.

Corollario: v è un discendente di u nella foresta DF $\Leftrightarrow$
$$u.d < v.d < v.f < u.f.$$

Foresta DF → Quando scopro un vertice v durante l'ispezione di $adj[u]$ ho $v.\pi = u$. Il sottografo con radice u forma una foresta DF contenente almeno un albero DF. Quando esploro l'arco (u, v) di un albero DF, il nodo u è grigio e il nodo v è bianco.

Teorema del Cammino Bianco → In una forsesta DF di un grafo $G = (V, E)$ il vertice v è un discendente di u $\Leftrightarrow$al tempo $u.d$ esiste un cammino bianco da u a v in G fatto solo di nodi bianchi (eccetto u che è appena diventato grigio).

Classificazione degli Archi → Durante la DFS posso classificare gli archi del grafo:
- **T** → se il ramo (u, v) è un ramo della foresta DF;
- **F** → se v è un discendente di u ma non fa parte della foresta DF;
- **B** → se u è un discendente di v;
- **C** → qualsiasi altro arco tra i nodi dello stesso albero o di alberi diversi.

Posso anche classificare gli archi durante la DFS utilizzando i colori dei vertici. Durante la visita del ramo (u, v) se $v.color$ è:
- **WHITE** → il ramo è T;
- **GRAY** → il ramo è B;
- **BLACK** → il ramo è F o C:
  - $u.d < v.d$: il ramo è F (v è un discendente di u);
  - $u.d > v.d$: il ramo è C.

Teorema → Se il grafo $G = (V, E)$ è indiretto $\implies$ la foresta DF ho solo archi T e B.
LEMMA: un grafo diretto è **aciclico** $\Leftrightarrow$una visita DFS di G non genera archi B.

Ordinamento Topologico

L'ordinamento topologico di un grafo aciclico (sia diretto che indiretto) $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici tale che se $(u, v) \in E$ allora da qualche parte nell'ordinamento u appare prima di v (è diverso dall'ordinamento di un insieme di numeri). Non è possibile effettuare un ordinamento lineare su un grafo ciclico. E' utile per gestire oggetti che hanno un ordine parziale, ad esempio:
$$a > b \text{ e } b > c \implies a > c.$$
Ma potrei anche non sapere con precisione l'ordinamento totale:
$$a > c \text{ e } b > c \implies a > b \text{ o } a < b.$$
E' comunque possibile trovare un ordinamento totale anche da un ordinamento parziale, usando la DFS. Lo pseudocodice è:

$TopologicalSort(G)$

```
1  Chiama DFS(G) per calcolare i tempi di completamento
   v.f per ogni vertice v
2  Completata l'ispezione di ogni vertice, lo inserisce in testa
   ad una lista concatenata
3  return lista concatenata di vertici
```

Costo
Il tempo complessivo è $\Theta(V + E)$.
Correttezza
Mostriamo che se $(u, v) \in E$ allora $v.f < u.f$. Quando esploro (u, v) di che colore è e v?
- **Grigio** → impossibile altrimenti (u, v) sarebbe un arco B(back) $\implies$ contraddizione con ipotesi (no grafi ciclici);
- **Bianco** → allora v diventa discendente di u $\implies u.d < v.d < v.f < u.f$ (per th. Parentesi) $\implies v.f < u.f$;
- **Nero** → allora v è terminato ma u ancora no $\implies v.f < u.f$

Componenti Fortemente Connesse

Dato il grafo diretto $G = (V, E)$una componente fortemente connessa (SCC) di G è un insieme massimale di vertici $C \subseteq V$ tale che per ogni $u, v \in C$ esistono entrambi i cammini "$u \to v$" e "$v \to u$". Uso la matrice di adiacenza G e la sua trasposta G^{T} per ottenere singolarmente le componenti connesse del grafo $G = (V, E)$.

Trasposta di $G \to G^{\mathrm{T}} = (V, E^{\mathrm{T}})$, ovvero inverto la direzione di tutti gli archi del grafo. Se rappresento il grafo con la matrice di adiacenza il tempo è lineare altrimenti, se lo faccio con le liste di adiacenza il costo diventa $\Theta(V + E)$. Inoltre G e G^{T} hanno le stesse SCC.
Definisco anche il grafo delle componenti fortemente connesse $G^{SCC} = (V^{SCC}, E^{SCC})$ che ha un vertice per ogni SCC di G e un arco solo se c'è un arco che connette le SCC in G.

Lemma G^{SCC} è un DAG (grafo diretto aciclico).
L'algoritmo che restituisce il G^{SCC} è il seguente:

$Strongly - Connected - Components(G)$

```
1  Chiama DFS(G)      /per calcolare v.f di ogni vertice v
2  Calcola G^T
3  Chiama DFS(G^T)      /i tempi u.f sono considerati in ord. decresc.
4  Genera come output le radici di ogni SCC generata da
   DFS(G^T) come foresta DF.      /In pratica crea un grafo G^SCC
```

Costo Il tempo complessivo è $\Theta(V + E)$.

STRUTTURE DATI PER INSIEMI DISGIUNTI

Questa pratica è anche nota come UnionFind. Serve a gestire una collezione $S = S_1, ..., S_k$ di **insiemi disgiunti dinamici**. Ogni insieme è rappresentato da un **rappresentante**, non è rilevante quale sia, ma devo necessariamente essere sempre lo stesso. Le operazioni che implementano questa tecnica sono:
Make-Set(x) → crea un nuovo insieme $S_i = x$ e lo aggiunge ad S;
Union(x, y) → elimina gli insiemi disgiunti di x, y da S e ci aggiunge la loro unione: $S \leftarrow S - S_x - S_y + (S_x \bigcup S_y)$;
Find-Set(x) → ritorna il rappresentante dell'insieme che contiene x.

Componenti Connesse

In un grafo $G = (V, E)$i vertici u e v sono nella stessa componente connessa $\Leftrightarrow$c'è un cammino tra di loro. La componente connessa è una classe di equivalenza che risponde alla domanda "E' raggiungibile da?". Un grafo **indiretto** può essere connesso, mentre un grafo **diretto** può essere fortemente connesso. L'algoritmo che trova le componenti connesse é:

$Connected - Components(G)$

```
1  for ogni vertice v ∈ G.V
2      Make-Set(v)
3  for ogni arco (u, v) ∈ G.E
4      if Find-Set(u) ≠ Find-Set(v)
5          Union(u, v)
```

Gli insiemi sono implementati con le **liste concatenate**:
ogni insieme è rappresentato come una lista concatenata diversa e ogni elemento della lista contiene elemento, puntatore al rappresentante e puntatore all'elemento successivo. **Operazioni:**
- Make-Set(x) → crea una lista con un solo elemento;
- Union(x, y) → è la parte costosa dell'algoritmo (accodare una lista lunga ad una lista corta può richiedere molte operazioni)
  - **Soluzione:** Unione Pesata → tra la due liste da unire si accoda sempre la più corta alla più lunga.
- Find-Set(x) → ritorna il puntatore al rappresentante.

Costo
Utilizzando l'Unione Pesata per una sequenza di m operazioni su n elementi il costo è $O(m + n \log_2 n)$

- -

Heap

Un heap è un'albero binario quasi completo. l'altezza di un nodo è il numero di archi nel cammino più semplice dal nodo ad una foglia. L'altezza dell'heap, che è uguale all'altezza della radice è $\Theta(\log_2 n)$. L'heap può essere implementato usando un'array:

Radice dell'albero : è il primo elemento cioè $A[1]$;
Padre dell'i-esimo elemento : il padre di $A[i]$ è $A[i/2]$;
Figlio sx dell'i-esimo elemento : il figlio sx di $A[i]$ è $A[2 \cdot i]$
Figlio sx dell'i-esimo elemento : il figlio dx di $A[i]$ è $A[2 \cdot i + 1]$

Esistono due tipi di heap:
max-heap $\rightarrow \forall$ nodo i, eccetto la radice $A[\text{PARENT}(i)] \geq A[i]$
min-heap $\rightarrow \forall$ nodo i, eccetto la radice $A[\text{PARENT}(i)] \leq A[i]$
Da qui in poi useremo sempre il **max-heap** per comodità, infatti il nodo con chiave massima è sempre la radice.
Serve un algoritmo che conservi le proprietà del max-heap, ovvero che aggiusti l'heap dopo l'inserimento del nodo i. Lo pseudocodice è:

$Max - Heapify(A, i)$

```
1  l ← LEFT(i)
2  r ← RIGHT(i)
3  if l ≤ A.heap − size AND A[l] > A[i]
4        massimo ← l
5  else
6        massimo ← i
7  if r ≤ A.heap − size AND A[r] > A[massimo]
8        massimo ← r li if massimo ≠ i
9        scambia A[i] ↔ A[massimo]
10       MAX-HEAPIFY(A, massimo)
```

Costo
L'heap è alto $O(\log_2 n)$ poiché è un albero quasi completo. Il costo ad ogni livello è costante (3 confronti e max 2 scambi) $\Longrightarrow$ il costo complessivo è intuitivamente $O(\log_2 n)$.
Correttezza
Si può ottenere con l'invariante di ciclo (perché è ricorsivo in coda).

- -

Cammini minimi