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.

$TreeSearch(T.root, k)$ // $VersioneRicorsiva$

```
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:

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