

# **APPUNTI ALGORITMI E STRUTTURE DATI 2023-2024**

**Professore:**  
Antonio Virdis

**Autore:**  
Cambria Gabriele Domenico

# Algoritmi

Un **algoritmo** è un procedimento che descrive una sequenza di passi ben definiti, finalizzato a risolvere un dato problema.

Un algoritmo ha queste proprietà:

- Insieme finito di istruzioni
- Ha un input e un output;
- Ogni istruzione deve essere **ben definita** ed eseguibile **in un tempo finito** da un **agente di calcolo**;
- Utilizza memoria per salvare risultati intermedi;

L'algoritmo e il programma sono due cose **differenti**:

- Il programma è basato sugli algoritmi, che costituiscono l'*essenza computazionale*
- Gli algoritmi sono slegati da programmi specifici.

## Complessità Computazionale

Dato un determinato problema ci possono essere diversi algoritmi che riescono a risolverlo.

Ogni volta che abbiamo un algoritmo abbiamo diverse proprietà, che dipendono dalla richiesta del problema.

Problema	Istanza	Dimensione Istanza
Modello di Calcolo	Algoritmo	Correttezza

A partire da queste proprietà possiamo ricavare:

Complessità Temporale	Complessità Temporale <i>Worst Case</i>	Efficienza
Numero di operazioni eseguite per individuare il risultato data un'istanza	Numero <i>massimo</i> di operazioni eseguite per individuare il risultato data un'istanza particolarmente "sconveniente"	"Velocità" dell'algoritmo a risolvere il problema (rispetto a cosa però?)

# Algoritmo di Euclide per la ricerca dell'MCD

## *Metodo della sottrazione:*

```
In [ ]: int MCD(int x, int y){  
    while(x != y){  
        if(x > y)  
            x -= y;  
        else  
            y -= x;  
    }  
    return x;  
}
```

In questo caso dato in input i valori:  $\begin{cases} x = 30 \\ y = 21 \end{cases}$  si riesce a arrivare alla soluzione in 5 passi

x	y
30	21
9	21
9	12
9	3
6	3
3	3

## *Metodo del resto:*

```
In [ ]: int MCD(int x, int y){  
    while(y != 0){  
        int k = x;  
        x = y;  
        y = k%y;  
    }  
    return x;  
}
```

Con gli stessi input di prima con questo algoritmo la soluzione è raggiungibile in 4 passi.

x	y
30	21
21	9
9	3
3	3
3	0

# Ricerca Numeri Primi minori di $n$

Dei modi in brute force per riuscire nell'impresa possono essere:

- Divido ogni numero precedente per i suoi numeri precedenti. E' primo solo se ha come divisori 1 e se stesso.
- Divido ogni numero precedente per i suoi numeri precedenti fino a  $\sqrt{n}$ . E' primo solo se ha come divisori 1 e se stesso.

Analizzando il problema possiamo ricavare che:

<b>Problema</b>	Trovare i numeri primi minori di $n$
<b>Istanza</b>	$0 \rightarrow n$
<b>Dimensione Istanza</b>	$n$
<b>Modello di Calcolo</b>	Insieme di caselle contenenti i numeri
<b>Algoritmo</b>	Strategia di calcolo <b>comprendibile</b> e <b>compatta</b> dei numeri primi. Permette di descrivere la sequenza di operazioni sul modello di calcolo per istanze generiche
<b>Correttezza</b>	Strategia eseguibile per istanze generiche

Una volta trovato un algoritmo andremo a calcolarne la *complessità computazionale (generica e WC)* e l'*efficienza*.

## Setaccio di Eratostene

L'algoritmo funziona in questo modo:

1. Elencare tutti i numeri fino a  $n$ ;
2. Partendo dal primo numero primo (2), cancellare dall'elenco tutti i suoi multipli a partire dal quadrato ( $2^2$ );
3. Ripetere il procedimento con i numeri seguenti non ancora cancellati minori di  $n$ .

2	3	-4	5	-6	7	-8	-9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

```
In [ ]: int numPrimi(int n){
    bool primi[n];
    // Inizializzo la matrice
    primi[0] = primi[1] = false;
    for(int i = 2; i < n; i++)
        primi[i] = true;

    int i = 2;
    for(; i*i < n; i++){
        // Scorro fino al primo successivo
        while(!primi[i])
            i++;
        // A partire dal quadrato elimino tutti i multipli minori di n
        for(int k = i*i; k < n; k += i)
            primi[k] = false;
    }

    for(int j = 2; j < n; j++){
        if(primi[j])
            cout << j << endl;
    }
}
```

## Complessità di un Algoritmo

La complessità di un algoritmo è rappresentata da una funzione (sempre positiva) che associa alla **dimensione** del problema il **costo** della sua soluzione.

- Il *costo* può essere inteso come tempo, spazio(memoria),...
- La *dimensione* dipende dai dati in input

Il confronto tra due algoritmi si effettua attraverso il confronto tra le relative funzioni di complessità.

La complessità può essere intesa secondo diversi punti di vista:

- Tempo di esecuzione (*BC*, *AC*, *WC*)
- Memoria (*in-place*, *not in-place*)

Queste funzioni infatti misurano l'efficienza di un algoritmo come proprietà intriseca, separandola dal linguaggio di sviluppo o dal calcolatore che lo esegue, diversamente dal **Profiling** (che misura gli indici di prestazione a *runtime*).

A parità di dimensione, istanze diverse in uno stesso algoritmo potrebbero necessitare di tempi diversi, perciò quando si calcolano le *funzioni di costo* si differenzia tra:

- *Best Case (BC)*, caso più "favorevole", agisce da "limite inferiore"
- *Average Case (AC)*, caso medio
- *Worst Case (WC)*, caso più "sfavorevole", agisce da "limite superiore"

Le funzioni di costo si indicano così:

$T_P(n)$ : complessità del tempo di esecuzione del programma  $P$  al variare di  $n$ .

Le funzioni di costo sono basate su un modello di calcolo. Per questo esempio prenderemo un modello che assegna ad *ogni operazione elementare costo 1*.

---

## Tipi di Complessità

La complessità può essere intesa secondo diversi punti di vista:

- Tempo di esecuzione ( $BC$ ,  $AC$ ,  $WC$ )
- Memoria (*in-place*, *not in-place*)

```
In [ ]: int max(int a[], int n){  
    int max = a[0];      // 1  
    for(int i = 1;       // 1  
        i < n;          // 1 per ogni iterazione + controllo finale  
        i++){           // 1 per ogni iterazione  
        if(max < a[i]) // 1 per ogni iterazione  
            max = a[i];  
        /*  
            WC: 1 per ogni iterazione  
            AC: 1 per metà delle iterazioni  
            BC: non entro mai nell'if  
        */  
    }  
    return max;          // 1  
}
```

$$WC: T_P(n) = 1 + 1 + (n - 1)(1 + 1 + 1 + 1) + 1 + 1 = 4n$$

$$AC: T_P(n) = 1 + 1 + (n - 1)(1 + 1 + 1) + \frac{n-1}{2}(1) + 1 + 1 = \frac{7}{2}n + \frac{1}{2}$$

$$BC: T_P(n) = 1 + 1 + (n - 1)(1 + 1 + 1) + 1 + 1 = 3n + 1$$

## Confronto Asintotico

Per analizzare l'efficienza indipendente dalle dimensioni specifiche di un istanza, si esegue un **Analisi Asintotica**.

Prendiamo per esempio tre programmi diversi:  $P, Q, R$  che hanno come  $WC$  (a parità di input) rispettivamente:

- $T_P(n) = 2n^2$
- $T_Q(n) = 100n$
- $T_R(n) = 5n$

Possiamo notare che:

- $n \geq 50 \Rightarrow T_Q(n) \leq T_P(n)$ 
  - $T_Q(n)$  ha complessità **minore o uguale** a  $T_P(n)$ , ma non vale il contrario;
- $n \geq 3 \Rightarrow T_R(n) \leq T_P(n)$

- $T_R(n)$  ha complessità **minore o uguale** a  $T_P(n)$ , ma non vale il contrario;
- $\forall n \Rightarrow T_R(n) \leq T_Q(n)$ 
  - $T_R(n)$  ha complessità **minore o uguale** a  $T_Q(n)$ ;
- $\forall n \Rightarrow T_Q(n) \leq 20T_R(n)$ 
  - $T_Q(n)$  ha complessità **minore o uguale** a  $T_R(n)$ ;

Queste ultime due considerazioni implicano che  $T_Q(n)$  e  $T_R(n)$  hanno **la stessa complessità**.

---

## **Limite Asintotico superiore**

Per esprimere il *limite asintotico superiore* si utilizza la notazione  $O$  grande:

$$f(n) \text{ è di ordine } O(g(n)) \Leftrightarrow \begin{cases} \exists n_0 \in \mathbb{Z} \\ \exists c > 0 \end{cases} \quad | \quad \forall n \geq n_0 : \quad f(n) \leq cg(n)$$

La scritta " $f(n)$  è di ordine  $O(g(n))$ " è spesso riassunta nelle due seguenti notazioni:

- $f(n)$  è  $O(g(n))$
- $f(n) \in O(g(n))$

Tornando agli esempi fatti prima possiamo dire che:

- $T_Q(n) \in O(T_P(n))$ ,  $[n_0 = 50, c = 1]$  oppure  $[n_0 = 1, c = 50]$
- $T_R(n) \in O(T_P(n))$ ,  $[n_0 = 3, c = 1]$
- $T_R(n) \in O(T_Q(n))$ ,  $[n_0 = 1, c = 1]$
- $T_Q(n) \in O(T_R(n))$ ,  $[n_0 = 1, c = 20]$
- $T_P(n) \notin O(T_Q(n))$
- $T_P(n) \notin O(T_R(n))$

$O(f(n))$  indica proprio l'insieme delle funzioni di ordine  $O(f(n))$ , perciò:

- $O(n) = \{ \text{costanti}, n, 4n, 300n, 100 + n, \dots \}$
- $O(n^2) = O(n) \cup \{ n^2, 300n^2, n + n^2, \dots \}$

## **Regola dei fattori costanti**

$$\forall k > 0, \quad O(f(n)) = O(kf(n))$$

## **Regola della somma**

$$f(n) \in O(g(n)) \quad \Rightarrow \quad f(n) + g(n) \in O(g(n))$$

## **Regola del Prodotto**

$$\begin{cases} f(n) \in O(f_1(n)) \\ g(n) \in O(g_1(n)) \end{cases} \quad \Rightarrow \quad f(n)g(n) \in O(f_1(n)g_1(n))$$

## Altre Regole

- **Regola transitività:**  $\begin{cases} f(n) \in O(g(n)) \\ g(n) \in O(h(n)) \end{cases} \Rightarrow f(n) \in O(h(n))$
- $\forall k \Rightarrow k \in O(1)$
- $m \leq p \Rightarrow n^m \in O(n^p)$
- Un polinomio di grado  $m \in O(n^m)$

## Classi di Complessità

Le classi di complessità più comuni e più famose sono le seguenti, in ordine di complessità crescente:

- $O(1)$ , costante
- $O(\log n)$ , logaritmica
- $O(n)$ , lineare
- $O(n \log n)$ ,  $n \log n$
- $O(n^2)$ , quadratica
- $O(n^3)$ , cubica
- ...
- $O(n^p)$ , polinomiale
- $O(2^n)$ , esponenziale
- $O(n^n)$ , esponenziale
- $O(n!)$ , fattoriale

Ciò significa che ogni funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale:

$$\begin{cases} \forall k \\ \forall a > 1 \end{cases} \Rightarrow n^k \in O(a^n)$$

---

## Limite Asintotico Inferiore

Mentre la notazione  $O$  ci da un *limite asintotico superiore*, che rappresenta come si comporta la nostra funzione nel peggior caso

La notazione  $\Omega$  ci da il *limite asintotico inferiore*, ovvero ci assicura che non è possibile trovare algoritmi che risolvino il problema con complessità minore di una data funzione.

Per esprimere il *limite asintotico inferiore* si utilizza la **notazione  $\Omega$  grande**:

$$f(n) \in \Omega(g(n)) \Leftrightarrow \begin{cases} \exists n_0 \in \mathbb{Z} \\ \exists c > 0 \end{cases} \quad | \quad \forall n \geq n_0 : f(n) \geq cg(n)$$

Alcuni esempi:

- $2n^2 + 3n + 2 \in \Omega(n)$  [ $n_0 = 1, c = 1$ ]
  - $n^2 + 3n + 2 \in \Omega(n^2)$  [ $n_0 = 1, c = 1$ ]
  - $n^2 \in \Omega(2n^2)$  [ $n_0 = 1, c = \frac{1}{2}$ ]
  - $n^2 \in \Omega(100n)$  [ $n_0 = 101, c = 1$ ]
- 

## **Limite Asintotico Stretto**

Per esprimere il *limite asintotico stretto* si utilizza la **notazione  $\Theta$  grande**:

$$\begin{aligned} f(n) \in \Theta(g(n)) &\Leftrightarrow \begin{cases} f(n) \in O(g(n)) \\ f(n) \in \Omega(g(n)) \end{cases} \\ f(n) \in \Theta(g(n)) &\Leftrightarrow \begin{cases} \exists n_0 \in \mathbb{Z} \\ \forall c_1 > 0 \\ \forall c_2 > 0 \end{cases} \quad | \quad \forall n > n_0 : c_1g(n) \leq f(n) \leq c_2g(n) \end{aligned}$$

$f(n) \in \Theta(g(n))$  se  $f(n)$  e  $g(n)$  hanno lo stesso ordine di complessità.

Alcuni esempi:

- $2n^2 + 3n + 2 \begin{cases} \in \Omega(n) \\ \notin O(n) \end{cases} \Rightarrow \notin \Theta(n)$
- $\begin{cases} 2n^2 + 3n + 2 \in \Omega(n^2) \\ 2n^2 + 3n + 2 \in O(n^2) \end{cases} \quad [n_0 = 1, c = 7] \quad [n_0 = 1, c = 1] \Rightarrow 2n^2 + 3n + 2 \in \Theta(n^2)$

Alcuni teoremi sono:

- $f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$
  - $f(n) \in \Theta(g(n)) \Rightarrow g(n) \in \Theta(f(n))$
  - Sia per  $\Omega$  che per  $\Theta$  valgono le regole dei *valori costanti*, *del prodotto* e *della somma*
  - Un polinomio di grado  $m \in \Theta(n^m)$
  - $O, \Omega$  e  $\Theta$  sono **relazioni transitive**
- 

## **Complessità reale**

La complessità computazionale asintotica si traduce in differenze di tempo *pratiche*.

Queste differenze si riscontrano soprattutto quando i valori di riferimento sono grandi.

Un esempio può essere visto con gli algoritmi di ordinamento con 1.000.000 di elementi:

- Algoritmo  $O(n^2)$ : più di un minuto
- Algoritmo  $O(n \log n)$ : 0.063 secondi circa

# Complessità algoritmi

Nella seguente sezione utilizzeremo la seguente leggenda:

- $Q[]$ : costrutto del linguaggio → *Classi di Complessità*
- $V$ : costanti
- $I$ : variabili
- $E$ : espressione
- $C$ : comando

Il nostro costrutto sarà:

- $Q[V] = Q[I] = O(1)$
- $Q[E_1 \text{ op } E_2] = Q[E_1] + Q[E_2]$
- $Q[I[E]] = Q[E]$
- $Q[I[E]] = Q[E]$
- $Q[I[E_1] = E_2;] = Q[E_1] + Q[E_2]$
- $Q[\text{return } E;] = Q[E]$
- $Q[\text{if}(E) C] = Q[E] + Q[C]$
- $Q[\text{if}(E) C_1 \text{ else } C_2] = Q[E] + Q[C_1] + Q[C_2]$
- $Q[\text{for}(E_1; E_2; E_3)C] = Q[E_1] + Q[E_2] + (Q[C] + Q[E_2] + Q[E_3])O(g(n))$   
( $g(n)$  numero di iterazioni)
- $Q[\text{while}(E) C] = (Q[C] + Q[E])O(g(n)) + Q[E]$
- $Q[C_1 \dots C_N] = Q[C_1] + \dots + Q[C_N]$
- $Q[f(E_1, \dots, E_N)] = Q[E_1] + \dots + Q[E_N] + O(f(n))$

---

## Selection Sort

```
In [ ]: void exchange(int& a, int& b){  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
void selectionSort(int v[], int n){  
    for(int i = 0; i < n-1; i++){ // O(n)  
        int min = i;  
        for(int j = i + 1; j < n; j++){ // O(n)  
            if(v[j] < v[min])  
                min = j;  
        }  
        exchange(v[i], v[min]);  
    }  
}
```

Studio il *Selection Sort* come **numero di cicli**:

- $T_{exc}(n) \in O(1)$ , tutte operazioni costanti

- $T_{ss}(n) \in O(n^2)$ , le operazioni nel ciclo interno hanno costo  $O(n)$ , e vengono iterate con un costo  $O(n)$ , perciò il costo totale è  $O(n * n) = O(n^2)$

Se voglio studiare il **numero di confronti**:

- $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2)$

Se voglio studiare il **numero di scambi**:

- E' sempre  $O(n)$

Se voglio studiare l'**utilizzo di memoria**

- $O(n)$ , inoltre la funzione è *in-place* in quanto non utilizza ulteriore memoria non costante.
- 

## Bubble Sort

```
In [ ]: void exchange(int& a, int& b){
    int tmp = a;
    a = b;
    b = tmp;
}

int bubbleSort(int v[], int n){
    for(int i = 0; i < n-1; i++){                         // O(n)
        for(int j = n - 1; j >= i + 1; j--){           // O(n)
            if(v[j] < v[j-1])
                exchange(v[j], v[j-1]);
        }
    }
}
```

Studio il *Bubble Sort* come **numero di cicli**:

- $T_{exc}(n) \in O(1)$ , tutte operazioni costanti
- $T_{bs}(n) \in O(n^2)$ , le operazioni nel ciclo interno hanno costo  $O(n)$ , e vengono iterate con un costo  $O(n)$ , perciò il costo totale è  $O(n * n) = O(n^2)$

Se voglio studiare il **numero di confronti**:

- $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2)$

Se voglio studiare il **numero di scambi**:

- Nel WC è  $O(n^2)$
- Nel BC è  $O(1)$

Se voglio studiare l'**utilizzo di memoria**

- $O(n)$ , inoltre la funzione è *in-place* in quanto non utilizza ulteriore memoria non costante.
- 

## Inseriton Sort

Scorro il vettore finché gli elementi sono ordinati.

Quando trovo un elemento non in ordine, torno indietro fino a inserirlo nel punto corretto.

```
In [ ]: void exchange(int& a, int& b){
    int tmp = a;
    a = b;
    b = tmp;
}

void insertionSort(int v[], int n){
    int mano = 0;
    int occhio = 0;
    for(int i = 0; i < n - 1; i++){
        mano = v[i];
        occhio = i - 1;
        // finché l'elemento è in posizione errata
        while(occhio >= 0 && v[occhio] > mano){
            // sposto il valore nella posizione accanto
            v[occhio + 1] = v[occhio];
            // vado indietro di una
            occhio--;
        }
        // inserisco il valore
        v[occhio + 1] = mano;
    }
}
```

Studio l'*Insertion Sort* come **numero di cicli**:

- $T_{exc}(n) \in O(1)$ , tutte operazioni costanti
- $T_{is}(n) \in O(n^2)$ , le operazioni nel ciclo interno hanno costo  $O(n)$ , e vengono iterate con un costo  $O(n)$ , perciò il costo totale è  $O(n * n) = O(n^2)$

Se voglio studiare il **numero di confronti**:

$$\bullet \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2)$$

Se voglio studiare il **numero di scambi**:

$$\bullet \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in O(n^2), \text{ nel peggior caso effettuo ad ogni confronto uno scambio}$$

Se voglio studiare l'**utilizzo di memoria**

- $O(n)$ , inoltre la funzione è *in-place* in quanto non utilizza ulteriore memoria non costante.

---

## Algoritmi di ordinamento e limiti asintotici

Oltre che agli algoritmi, i limiti asintotici possono essere applicati alla complessità di un problema, e indicano la più bassa complessità con la quale quel problema può essere risolto.

Ad esempio gli *algoritmi di ordinamento* hanno come limite superiore minimo:

- Basati sul confronto (intuitivamente):  $O(n)$ , necessito di scorrere almeno una volta tutti gli elementi
- Basati sul confronto (più basso):  $O(n \log n)$  [*merge sort, quicksort*]

In generale un algoritmo ha tre casi nei quali vale la pena studiarlo data un'istanza generica  $I$ :

- **Best Case (BC)**: miglior caso, numero minimo di passaggi
  - $T_{best} = \min_{I_n}(\text{tempo}(I))$
- **Average Case (AC)**: caso medio, numero medio di passaggi
  - $T_{average} = \sum_{I_n}(\text{tempo}(I) * p(I))$
- **Worst Case (WC)**: peggior caso, numero massimo di passaggi
  - $T_{worst} = \max_{I_n}(\text{tempo}(I))$

---

## Ricerca Binaria

Per la ricerca di un dato valore in un vettore **ordinato**.

Questa è un assunzione che facciamo all'inizio.

```
In [ ]: bool binarySearch_it(int v[], int x, int l, int r){  
    int m = (l+r)/2;  
  
    while(v[m] != x){  
        if(x < v[m])  
            r = m - 1;  
        else  
            l = m + 1;  
        if(l > r)  
            return false;  
        m = (l+r)/2;  
    }  
    return true;  
}
```

Ipotizzando che i numeri abbiano tutti la stessa probabilità di apparire nel vettore, studio i casi:

- BC =  $O(1)$
- WC =  $O(\log n)$

La complessità peggiore è  $\log n$  perché tutte le operazioni nella funzione sono  $O(1)$ , eccezion fatta quelle all'interno del ciclo.

Per capire quante iterazioni esso compie, dobbiamo tenere a mente che  $a$  e  $b$  ad ogni iterazione si avvicinano della metà della distanza, perciò, nel momento in cui essi combaciano, ho diviso per 2 esattamente  $x$  volte ( $x$  numero di step totali compiuti):

$$\frac{n}{2^x} = 1 \Rightarrow n = 2^x \Rightarrow x = \log_2 n = \log n$$

---

---

## Programmi Ricorsivi

Per comprendere gli algoritmi ricorsivi, ci si può basare su strutture derivanti da altre discipline (matematica, basi di dati, fisica, ...).

Ad esempio, il **fattoriale** (di un numero maggiore di 0) può essere visto:

- *Approccio Iterativo:*  $n! = 1 * 2 * 3 * \dots * n$
- *Approccio Ricorsivo:*  $n! = n * (n - 1)!$

```
In [ ]: int fattoriale_iter(int n){  
    if(n < 0)  
        return -1;  
  
    int out = 1;  
    for(int i = 2; i <= n; i++)  
        out *= i;  
    return out;  
}  
  
int fattoriale_ric(int n){  
    if(n < 0)  
        return -1;  
    if(n == 1 || n == 0)  
        return 1;  
    return n * fattoriale_ric(n-1);  
}
```

Gli algoritmi ricorsivi **devono rispettare 3 regole**:

- Individuare i *casi base (default case)* in cui la funzione è definita immediatamente
- Effettuare le chiamate ricorsive su un insieme più "piccolo" di dati
- Fare in modo che alla fine di ogni sequenza di chiamate ricorsive, si ricada **SEMPRE** in uno dei *casi base*.

## Pari e MCD ricorsivi

```
In [ ]: bool pari(int x){
    if(x == 0)
        return true;
    if(x == 1)
        return false;

    return pari(x-2);
}

// Algoritmo di Euclide (differenza)

int MCD(int x, int y){
    if (x == y)
        return x;
    if(x < y)
        return MCD(x, y-x);
    return MCD(x-y, y);
}

// Algoritmo di Euclide (resto)
int MCD(int x, int y){
    if (x == y)
        return x;
    if(x < y)
        return MCD(x, y%x);
    return MCD(x%y, y);
}
```

## Strutture dati ricorsive

Esistono alcune strutture dati che per loro natura possono essere considerate ricorsive:

- **Lista:**
  - **NULL** è una Lista
  - L'elemento che è seguito da una lista è una lista.

```
In [ ]: int lenght(elem* p){
    if(p->nextEl == nullptr)
        return 0;
    return 1 + lenght(p->nextEl);
}

int howMany(elem* p, int x){
    if(p == nullptr)
        return 0;
    if(p->nextEl == nullptr)
        return 0;
    return (p->val == x) + howMany(p->nextEl, x);
    //? Sfrutto il fatto che se la condizione è vera
    //? il true viene convertito in 1
}
```

```

bool belongs(elem* p, int x){
    if(p == nullptr)
        return false;
    if(p->var == x)
        return true;
    return belongs(p->next, x);
}

void eliminaCoda(elem* &p){
    if(p == nullptr)
        return;
    if(p->next == nullptr){
        delete p;
        p = nullptr;
        //? Si riflette sul puntatore del penultimo elemento
        //? perché passato per riferimento
    }
    else
        eliminaCoda(p->next);
}

void inserisciCoda(elem* &p, int x){
    if(p == nullptr){
        p = new elem;
        p->var = x;
        p->next = nullptr;
    }
    else
        inserisciCoda(p->nextEl, x);
}

void append(elem* &l1, elem* l2){
    if(l1 == nullptr)
        l1 = l2;
    else
        append(l1->nextEl, l2);
}

elem* append(elem* l1, elem* l2){
    if(l1 == nullptr)
        return l2;

    l1->nextEl = append(l1->next, l2);
    return l1;
}

```

## Definizioni Utili

- **Principio di induzione naturale:**
  - Data una proprietà  $P$  sui naturali:  
Se  $P(0)$  valida  $\Rightarrow \forall n : P(n)$  vera  $\Rightarrow P(n + 1)$  vera
- **Principio di induzione completa:**
  - Data una proprietà  $P$  sui naturali:  
Se  $P(0)$  valida  $\Rightarrow \forall n : \forall m < n \rightarrow P(m)$  vera  $\Rightarrow P(n + 1)$  vera

- **Principio di induzione ben fondata:**

- Data una proprietà  $P$  sui naturali.

Se

$P(x)$  valida  $\wedge x$  minimale di  $S \Rightarrow \forall E \in S : \forall m < n \rightarrow P(m)$  vera  $\Rightarrow P(n)$  vera

---

## Complessità algoritmi ricorsivi

In [ ]:

```
int fact(int x){
    if(x == 0) return 1;      // O(1)

    return x*fact(x-1);      // O(1) + T(n-1)
}
```

La complessità di un codice del genere sarebbe:

- $T(0) = a$
- $T(n) = b + T(n - 1)$

Nelle funzioni ricorsive la *complessità* si chiama: **Relazione di Ricorrenza**, ed è anch'essa ricorsiva.

Per risolvere esistono diversi metodi, un esempio è il **metodo di sostituzione**:

- $T(0) = a$
- $T(1) = b + T(0) = b + a$
- $T(2) = b + T(1) = b + b + a = 2b + a$
- $T(3) = b + T(2) = b + 2b + a = 3b + a$
- $T(n) = b + T(n - 1) = b + (n - 1)b + a = nb + a$

La nostra funzione ha quindi complessità  $O(n)$

## Selection Sort Ricorsivo

In [ ]:

```
/// v = vettore
/// m = lunghezza
/// i = posizione primo elemento che devo controllare
void r_selectionSort(int* v, int m, int i = 0){
    if(i == m-1)
        return;
    int min = i;
    for(int j = i + 1; j < m; j++)
        if(v[j] < v[min])
            min = j;
    exchange(v[i], v[min]);

    r_selectionSort(v, m, i+1);
}
```

$$T_{base} = T(1) = a$$

$$T_{generica} = T(n) = bn + T(n - 1)$$

Utilizzo il metodo della sostituzione:

- $T(1) = a$
- $T(2) = 2 * b + T(0) = 2b + a$
- $T(3) = 3 * b + T(1) = 3b + 2b + a$
- $T(4) = 4 * b + T(2) = 4b + 3b + 2b + a$
- $T(n) = nb + T(n - 1) = (n + (n - 1) + \dots + 2)b + a = (\frac{n(n-1)}{2} - 1)b + a$

Quindi:  $T(n) \in O(n^2)$

## Quick Sort

Il *quick sort* è un algoritmo ricorsivo che lavora su una **porzione di array** compresa tra due valori *inf* e *sup*:

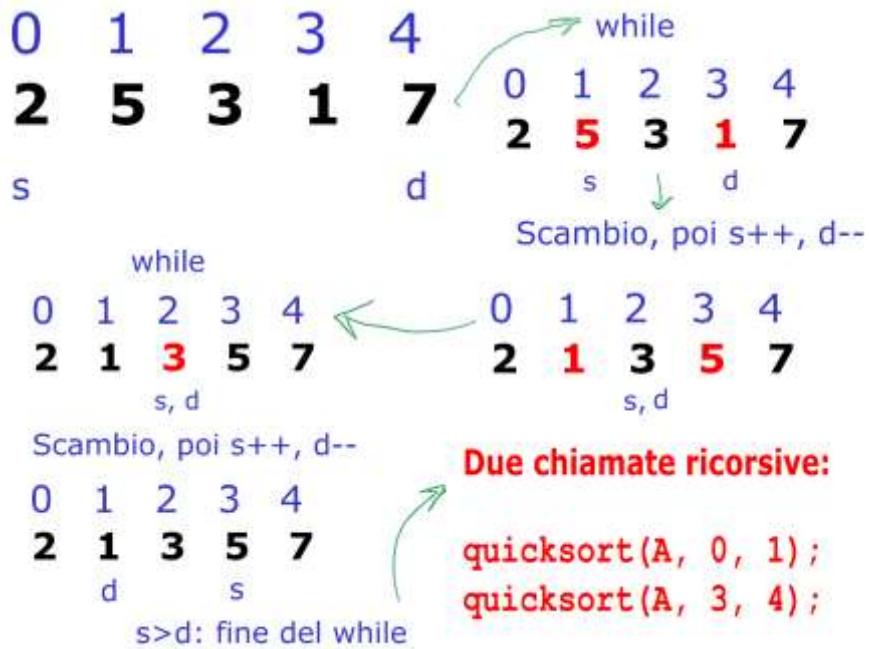
- Sceglie un *perno*;
- Divide l'array in due parti:
  - Prima parte: elementi  $\leq$  perno
  - Seconda parte: elementi  $\geq$  perno
- Chiama *quickSort()* sulla prima parte (se contiene almeno due elementi)
- Chiama *quickSort()* sulla seconda parte (se contiene almeno due elementi)

```
In [ ]: void quickSort(int v*, int inf = 0, int sup = n - 1){
    int perno = v[(sup + inf)/2];
    int s = inf;
    int d = sup;
    while(s < d){
        while(v[s] < perno)
            s++;
        while(v[d] > perno)
            d--;
        if(s > d)
            break;
        exchange(v[s], v[d]);
        s++;
        d--;
    }

    if(inf < d)
        quickSort(A, inf, d);
    if(sup > s)
        quickSort(A, s, sup);
}
```

Uno schema può essere il seguente:

- $v = [2, 5, 3, 1, 7]$
- $inf = 0, sup = 4, perno = 3$



Successivamente viene richiamata sui vettori:

- $v_1 = [2, 1]$
- $v_2 = [5, 7]$

La singola chiamata ha costo massimo:

- $T_{base} = T(1) = a$
- $T_{worst} = T(n) = nb + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = nb + 2T\left(\frac{n}{2}\right)$

Utilizzando il metodo della sostituzione:

- $T(1) = a$
- $T(2) = 2b + 2T(1) = 2b + 2a$
- $T(4) = 4b + 2T(2) = 4b + 2(2b + 2a) = 4b + 4b + 4a = (2^2 * 2)b + 4a$
- $T(8) = 8b + 2T(4) = 8b + 2(8b + 4a) = 8b + 8b + 8b + 8a = (2^3 * 3)b + 8a$
- $T(16) = 16b + 2T(8) = 16b + 2(16b + 8b + 8a) = 16b + 16b + 16b + 16b + 16a =$
- $T(n) = (n \log n)b + na$

Cio significa che il nostro algoritmo:  $T_w(n) \in O(n \log n)$

E' possibile dimostrare, attraverso versioni "randomizzate" del *quickSort* (dove il perno è scelto in modo casuale) che anche nell'**average case** (supponendo che le permutazioni dei valori siano tutte equiprobabili) la complessità è uguale a quella nel caso peggiore, ovvero  $O(n \log n)$ , seppur con una costante  $c$  modificata.

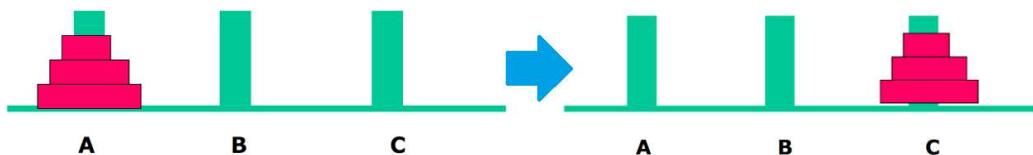
## Torre di Hanoi

La *torre di hanoi* è un gioco dove sono presenti 3 paletti e 1 torre di  $n$  cerchi, ognuno con un diametro sempre più piccolo.

Lo scopo del gioco è quello di spostare la torre dal paletto sorgente ( $A$ ) a quello destinatario ( $C$ ) usando un paletto ausiliario ( $B$ )

Le regole sono le seguenti:

- Si può spostare solamente un cerchio alla volta
- Non è possibile impilare un cerchio di diametro maggiore sopra ad uno già posizionato.

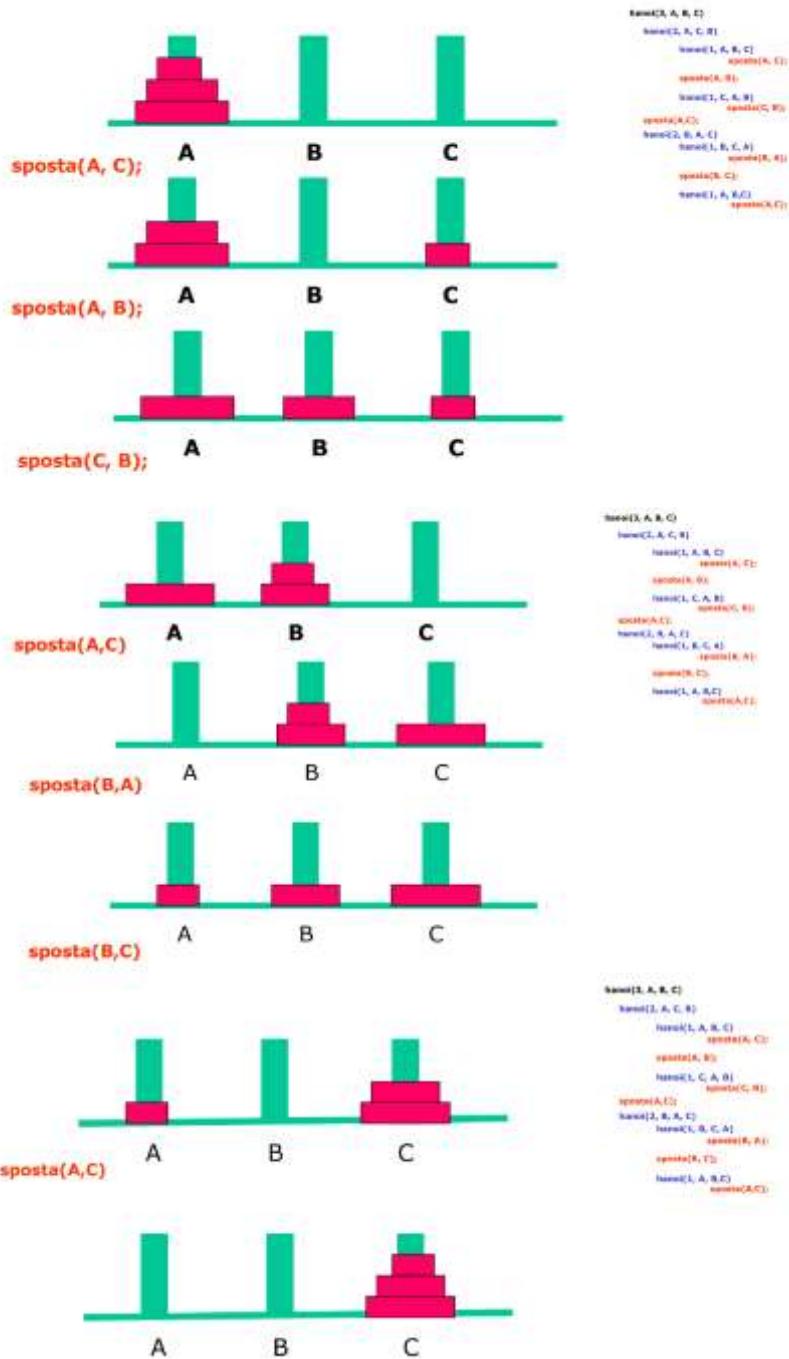


L'algoritmo ricorsivo che ci permette di risolvere il gioco è il seguente:

- Se  $n = 1 \Rightarrow A \rightarrow C$
- Altrimenti:
  - Trasferisci la torre degli  $n - 1$  cerchi più piccoli da  $A \rightarrow B$  utilizzando  $C$  come ausiliario
  - $A \rightarrow C$
  - Trasferisci la torre degli  $n - 1$  cerchi più piccoli da  $B \rightarrow C$  utilizzando  $A$  come ausiliario

```
In [ ]: void hanoi(int* n, pal A, pal B, pal C){  
    if(n==1)  
        sposta(A, C);  
    else{  
        hanoi(n-1, A, C, B);  
        sposta(B, C);  
        hanoi(n-1, B, A, C);  
    }  
}
```

Uno schema dei passaggi è il seguente:



Ipotizzando che  $sposta(x, y) \in O(1)$ , la singola chiamata ha costo massimo:

- $T_{base} = T(1) = a$
- $T_{generico} = T(n) = b + 2T(n - 1)$

Utilizzando il metodo della sostituzione:

- $T(1) = a$
- $T(2) = b + 2T(1) = b + 2a$
- $T(3) = b + 2T(2) = b + 2(b + 2a) = 3b + 4a = (2^{3-1} - 1)b + 2^{3-1}a$
- $T(4) = b + 2T(3) = b + 2(3b + 4a) = 7b + 8a = (2^{4-1} - 1)b + 2^{4-1}a$
- $T(n) = (2^{n-1} - 1)b + 2^{n-1}a$

Ciò significa che:

$$T(n) \in O(2^n)$$

## Ricerca Lineare e Binaria Ricorsiva

```
In [ ]: bool linearSearch_ric(int* v, int start = 0, int end = m, int x){  
    if(start == end)  
        return false;  
    if(x == v[start])  
        return true;  
    else  
        return linearSearch_ric(v, start + 1, end, x);  
}  
  
bool binarySearch_ric(int* v, int start = 0, int end = m-1, int x){  
    if(start > end)  
        return false;  
    int k = (start+end)/2;  
    if(x == v[k])  
        return true;  
  
    if(x < v[k])  
        return binarySearch_ric(v, start, k - 1, x);  
    else  
        return binarySearch_ric(v, k+1, end, x);  
}  
  
bool search_ric(int* v, int start = 0, int end = m-1, int x){  
    if(start > end)  
        return false;  
    int k = (start+end)/2;  
    if(x == v[k])  
        return true;  
  
    return search_ric(v, start, k - 1, x) || search_ric(v, k+1, end, x);  
}
```

### Ricerca lineare:

- $T(1) = a$
- $T(n) = b + T(n - 1)$

Per quello che abbiamo dimostrato prima:

$$T(n) \in O(n)$$

### Ricerca binaria:

- $T(0) = a$
- $T(b) = b + T(\frac{n}{2})$

Utilizzando il metodo della sostituzione:

- $T(1) = b + a$
- $T(2) = b + T(1) = b + b + a$
- $T(4) = b + T(2) = b + 2b + a$

- $T(8) = b + T(4) = b + 3b + a$
- $T(n) = (\log n + 1)b + a$

Ciò significa che:

$$T(n) \in O(\log n)$$

### Ricerca Generica:

- $T(0) = a$
- $T(b) = b + 2T(\frac{n}{2})$

Utilizzando il metodo della sostituzione:

- $T(1) = b + a$
- $T(2) = b + 2T(1) = b + 2b + 2a$
- $T(4) = b + 2T(2) = b + 2b + 4b + 4a =$
- $T(8) = b + 2T(4) = b + 2b + 4b + 8b + 8a$
- $T(n) = (\sum_{i=1}^{\log n} 2^i)b + na = \frac{2^{\log_2 n} - 1}{2 - 1}b + na = (n - 1)b + na$

Ciò significa che:

$$T(n) \in O(n)$$

## Correlazioni di Ricorrenza

E' possibile trovare delle metodologie di ricorrenza che vengono utilizzati da diversi algoritmi a costo noto. Di seguito è possibile vederne alcune.

### **Divite et impera**

Dato un insieme  $S | \#S = n$ , è possibile dividere  $S$  in  $b$  sottoinsiemi.

Successivamente è possibile effettuare un numero di chiamate  $a \in [1, b] \cap \setminus \mathbf{N}$  per i quali risolvere il problema (volendo ancora utilizzare la stessa metodologia), per poi ricombinare il tutto.

```
In [ ]: void divideEtImpera(S{
    if(S.dim() <= m)
        // risolvi il problema
    else{
        // divido il problema in b sottoinsiemi
        divideEtImpera(S_1);
        // ...
        divideEtImpera(S_a);
        // combino i risultati
    }
}
```

E' possibile quindi associare le classi di complessità:

- $\begin{cases} T(1) = d \\ T(n) = c + aT(\frac{n}{b}) \end{cases} \Rightarrow \begin{cases} \in O(\log n), & a = 1 \\ \in O(n^{\log_b a}), & a > 1 \end{cases}$
- $\begin{cases} T(n) = d, & n \leq m \\ T(n) = cn^k + aT(\frac{n}{b}), & n > m \end{cases} \Rightarrow \begin{cases} \in O(n^k), & a < b^k \\ \in O(n^k \log n), & a = b^k \\ \in O(n^{\log_b a}), & a > b^k \end{cases}$

Da questi due casi generali possiamo ricavare i risultati che abbiamo trovato prima inserendo valori particolari per  $a, b, c, d$  e  $k$ :

- $\begin{cases} T(0) = d \\ T(n) = c + T(\frac{n}{2}) \end{cases} \Rightarrow \in O(\log n)$
- $\begin{cases} T(0) = d \\ T(n) = c + 2T(\frac{n}{2}) \end{cases} \Rightarrow \in O(n)$
- $\begin{cases} T(0) = d \\ T(n) = cn + 2T(\frac{n}{2}) \end{cases} \Rightarrow \in O(n \log n)$

## Relazioni Lineari

Come le classi di complessità degli algoritmi *divide et impera*, anche per le relazioni Lineari esistono delle regole.

La regola generale che più ci interessa è quella per distinguere la classe di un algoritmo da *polinomiale a esponenziale*.

La regola è la seguente:

$$\begin{cases} T(0) = d \\ T(n) = bn^k + \sum_{i=0}^r a_i T(n-i) \end{cases} \Rightarrow \begin{cases} \text{Polinomiale} \Leftrightarrow \text{!}\exists i \mid \begin{cases} a_i = 1 \\ a_{j \neq i} = 0 \end{cases} \\ \text{Esponenziale} \quad \exists i, j \mid \begin{cases} a_i \neq 0 \\ a_j \neq 0 \wedge i \neq j \end{cases} \end{cases}$$

Alcuni casi particolari che si incontrano spesso sono i seguenti:

- $\begin{cases} T(0) = d \\ T(n) = b + T(n-1) \end{cases} \Rightarrow \in O(n)$
- $\begin{cases} T(0) = d \\ T(n) = bn + T(n-1) \end{cases} \Rightarrow \in O(n^2)$
- $\begin{cases} T(0) = d \\ T(n) = bn^k + T(n-1) \end{cases} \Rightarrow \in O(n^{k+1})$
- $\begin{cases} T(0) = d \\ T(n) = b + 2T(n-1) \end{cases} \Rightarrow \in O(2^n)$

# Merge Sort

E' un algoritmo di ordinamento che si basa sulla filosofia del *divide et impera*.

Per ordinare un insieme si procede a dividerlo in due sottoinsiemi, e poi si ordinano i due sottoinsiemi con la stessa filosofia.

Grazie a questo ragionamento alla fine ottengo  $n$  sottoinsiemi ordinati localmente, ognuno con un solo elemento.

A questo punto mi basta riunire gli insiemi ordinati localmente in un unico insieme, inserendo i valori ottenuti in maniera ordinata

In [ ]: // Pseudocodice

```
void mergeSort(sequenza S){
    if(|S| <= 1)
        return;
    else{
        < divido S in due sottosequenze di uguale lunghezza S1, S2>
        mergeSort(S1);
        mergeSort(S2);
        <fondo S1 e S2>
    }
}
```

## Merge Sort su liste

In [ ]:

```
void split(Elem* &s1, Elem* &s2){
    if(s1 == nullptr || s1->next == nullptr)
        return;
    Elem* p = s1->next;
    // Prendo il secondo elemento
    s1->next = p->next;
    // Aggancio il primo elemento al terzo
    p->next = s2;
    // Faccio sì che p diventi la testa della lista s2
    s2 = p;
    // Adesso s2 è la testa
    split(s1->next, s2);
}

void merge(Elem* &s1, Elem* &s2){
    Elem* p = nullptr;
    if(s2 == nullptr)
        return;
    if(s1 == nullptr){
        s1 = s2;
        return;
    }

    if(s1->val <= s2->val){
        merge(s1->next, s2);
        //? Se s1 è il più piccolo, lo lascio e procedo con il next
    }
}
```

```

    else{
        merge(s2->next, s1);
        //? Altrimenti, tratto s2 come s1 e procedo con la next
        s1 = s2;
        //? Avendo trattato s2 come ordinata, la scambio
    }
}

void mergeSort(Elem*& s1){
    if(s1 == nullptr || s1->next == nullptr)
        return;

    Elel* s2 = nullptr;

    split(s1, s2);
    mergeSort(s1);
    mergeSort(s2);
    merge(s1, s2);
}

```

La complessità di questo codice:

- $T(0) = T(1) = b$
- $T(n) = T_{merge}(n) + T_{split}(n) + 2T(\frac{n}{2})$
- $T_{split} = \begin{cases} T(0) = T(1) = c \\ T(n) = c + T(n-2) \end{cases} \Rightarrow T_{split}(n) \in O(n)$
- $T_{merge} = \begin{cases} T(0) = d \\ T(n) = e + T(n-1) \end{cases} \Rightarrow T_{merge}(n) \in O(n)$

Di conseguenze abbiamo che:

- $T(0) = T(1) = b$
- $T(n) = O(n) + O(n) + 2T(\frac{n}{2})$

Ovvero:  $T(n) \in O(n \log n)$

## Merge Sort su Array

```

In [ ]: void merge(vector<int>& v, int head, int mid, int tail){
            int iSx = head, iDx = mid;
            vector<int> vTmp;
            while(iSx < mid && iDx <= tail){
                if(v[iSx] < v[iDx])
                    vTmp.push_back(v[iSx++]);
                else
                    vTmp.push_back(v[iDx++]);
            }

            while(iSx < mid)
                vTmp.push_back(v[iSx++]);

            while(iDx <= tail)
                vTmp.push_back(v[iDx++]);

            for(int i = head; i <= tail; ++i)
                v[i] = vTmp[i - head];
        }

```

```

void mergeSort(vector<int>& v, int head, int tail){
    if(head >= tail)
        return;
    int mid = (head + tail)/2;
    mergeSort(v, head, mid);
    mergeSort(v, mid + 1, tail);

    merge(v, head, mid + 1, tail);

}

```

Come complessità questo codice:

- $T(n) = c + 2T(\frac{n}{2})$

Riflettendo ci rendiamo conto che per ogni "livello" la combina esegue  $n$  spostamenti, e ho  $\log n + 1$  livelli.

Perciò il costo totale:

$$T(n) \in \Theta(n \log n)$$

Il mergeSort ha complessità  $O(n \log n)$  sia nel *Best Case* sia nell'*Average Case* sia nel *Worst Case*.

---

## Sorting Ibridi

E' possibile combinare più algoritmi di ordinamenti per migliorare le complessità.

Infatti, è possibile dimostrare che per vettori piccoli è molto probabile che il vettore sia già ordinato.

Perciò potremmo pensare di utilizzare il *merge sort* finché le dimensioni dei vettori non sono abbastanza piccole per andare a sfruttare l'alta probabilità dell'*insertion sort* di essere nel *Best Case*  $O(n)$

---

# Ordinamento Multi-Valore

Date due richieste, ordinarle secondo priorità. Se la priorità è uguale ordinare secondo l'ID

```
In [ ]: struct Richiesta{
    int id_;
    int prio_;
    Richiesta(int id, int prio): id_(id), prio_(prio){}
}

bool confrontaRichieste(Richiesta r1, Richiesta r2){
    if(r1.prio_ < r2.prio_)
        return true;

    if(r1.prio_ == r2.prio_)
        return (r1.id_ < r2.id_);

    return false;
}
```

## Limiti Inferiori per i Problemi

Un **problema** è di ordine  $\Omega(f(n))$  se non è possibile trovare un algoritmo che lo risolva con complessità minore di  $f(n)$ .

Tutti gli algoritmi che risolvono il problema, sono quindi nel migliore dei casi  $O(f(n))$ .

## Alberi di Decisione

Per gli algoritmi *basati su confronti*, dove la complessità è *proporzionale al numero di confronti effettuati*, per studiarne il limite inferiore, si applica la tecnica degli *alberi di decisione*.

Gli alberi ci permettono di rappresentare il comportamento dei nostri algoritmi.

Supponiamo di avere come input  $x_1, \dots, x_n$ . Il nostro algoritmo di confronto ad un certo punto si porrà la domanda  $x_i <? x_j$ , dove  $i, j$  sono dei valori scalari che il programma itera.

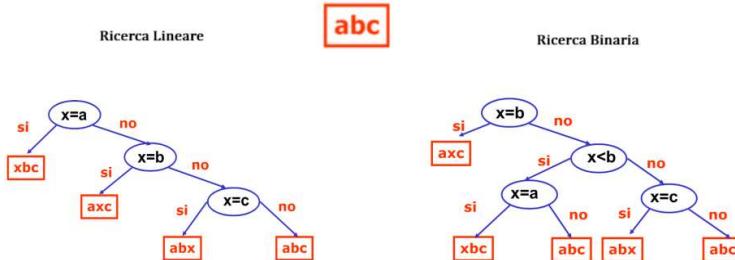
Questo significa che possiamo rappresentare la coppia  $(i, j)$  come un nodo che ha due possibili esiti (nodo binario).

Da questo ragionamento possiamo determinare un albero binario dove:

- Ogni *foglia* è una **soluzione dato un particolare assetto dei dati iniziali**
- Ogni *cammino dalla radice ad una foglia* rappresenta **un'esecuzione dell'algoritmo** (sequenza di confronti) per giungere alla soluzione relativa alla foglia.

Questa configurazione ci permette di visualizzare graficamente tutti i comportamenti del nostro algoritmo a seconda delle varie configurazioni in input.

In questo modo è possibile in maniera concreta stabilire il comportamento Migliore, Peggiore e Medio del nostro algoritmo, conteggiando il numero di nodi, ed effettuando eventuali medie aritmetiche/pesate.



Per tutti i problemi basati sul confronto con complessità porporzionale al numero di confronti effettuati:

- Ogni algoritmo che risolve un problema che ha **s** soluzioni, ha un albero di decisione corrispondente con almeno **s** foglie.
- Fra tutti gli alberi di decisione per il particolare problema con **s** soluzioni:
  - L'albero di decisione che **minimizza la lunghezza massima** dei percorsi (minimizza l'altezza massima) fornisce un **limite inferiore** al numero di confronti che un algoritmo effettua per risolvere il problema nel *caso peggiore*.
  - L'albero di decisione che **minimizza la lunghezza media** dei percorsi (minimizza l'altezza media delle foglie) fornisce un **limite inferiore** al numero di confronti che un algoritmo effettua per risolvere il problema nel *caso medio*.
  - L'albero di decisione che **minimizza la lunghezza migliore** dei percorsi (minimizza l'altezza minima per una qualsiasi foglia) fornisce un **limite inferiore** al numero di confronti che un algoritmo effettua per risolvere il problema nel *caso migliore*.

Alcune considerazioni sugli alberi binari con  $k$  livelli:

- Ha al massimo  $2^k$  foglie (ne ha quel numero quando è *bilanciato*).
- Se ha  $s$  foglie, ha almeno  $\log s$  livelli (ne ha quel numero quando è *bilanciato*)

Perciò gli **Alberi Binari Bilanciati** minimizzano sia il **WC** che il **AC**:

- $\log(s(n))$  livelli

Quindi, dato un insieme di  $n$  elementi, quanti possibili soluzioni posso avere?

Se ho  $n$  elementi, posso avere al massimo  $n!$  permutazioni.

Perciò avrò al massimo  $n!$  foglie.

Il cammino medio e massimo in un albero bilanciato è:

$\log n! \approx n \log n$  (formula di Stirling)

Perciò non posso avere algoritmi (basati sul confronto) che hanno soluzione migliore di  $O(n \log n)$ . Perciò:

- *MergeSort* è ottimo
- *QuickSort* è ottimo, nel caso medio

## ***Algoritmi di ordinamento con complessità minore di $O(n \log n)$***

E' possibile per un problema di ordinamento fare meglio di  $O(n \log n)$ ? Se è basato su confronto no, ma se non lo fosse?

E' il caso degli algoritmi:

- Counting Sort
- Radix Sort

### ***Counting Sort***

E' un algoritmo che ordina una sequenza di **interi**.

Si può utilizzare quando si conoscono i valori minimo e massimo degli elementi da ordinare.

Per ogni valore presente nell'array, si contano gli elementi con quel valore utilizzando un array ausiliario avente come dimensione l'intervallo dei valori.

Successivamente si ordinano i valori tenendo conto dell'array ausiliario.

Dato in input il vettore:

$$A = [7, 7, 4, 4, 7, 5, 4, 7, 4, 5, 1, 1, 0, 1]$$

Ho  $n = 14$  elementi, dove il numero massimo è 7 e il numero minimo è 0, perciò:  $k = 8$

$$[0, 1, 2, 3, 4, 5, 6, 7]$$

$$\& C = [1, 3, 0, 0, 4, 2, 0, 4]$$

Scorro adesso C e stampo ogni posizione il numero di volte contenute nell'array

```
In [ ]: void countingSort(int A[], int k, int n){  
    int C[k + 1];  
  
    for(int i = 0; i < k+1; ++i) //O(k)  
        C[i] = 0;  
    for(int i = 0; i < n; ++i) //O(n)  
        ++C[A[i]];  
  
    for(int i = 0, j = 0; i < n; ++i){ //O(n)  
        while(C[i] > 0){  
            A[j] = i;  
            --C[i];  
            ++j;  
        }  
    }  
  
}
```

Questo algoritmo non è basato sui confronti, e ha complessità:

$$T_n \in O(n + k)$$

Tuttavia l'algoritmo non lavora *in-place*, e ha bisogno una memoria che cresce come  $O(k)$

## Radix Sort

Ordina una sequenza di **int**eri.

Si può usare quando si conosce la lunghezza massima (intesa come numero di cifre)  $d$  dei numeri da ordinare

Si eseguono  $d$  cicli ripartendo, in base alla  $d$ -esima cifra, i numeri in  $k$  contenitori, dove  $k$  sono i possibili valori di una cifra, e rileggendo il risultato con un determinato ordine.

Il radix sort:

- Non è basato su confronti
- E' fondamentale partire dalla cifra meno significativa
- Ha necessità di memoria ausiliaria
- Ha costo:  $T_n \in O(d * (n + k))$  dove
  - $d$  è lunghezza delle sequenze
  - $k$  è il numero dei possibili valori di ogni cifra
- E' molto conveniente quando  $d \ll n$

```
In [ ]: int pow(int e){  
    int out = 1;  
    while(e > 0){  
        out *= 10;  
        --e;  
    }  
  
    return out;  
}
```

```

int getChypher(int i, int pos){
    int tmp;
    int pow10 = pow(pos);

    tmp = i % pow10;

    if(tmp < pow10/10)
        return 0;
    while(tmp >= 10){
        tmp /= 10;
    }

    return tmp;
}

void radixSort(vector<int>& A, int maxCypher){

    vector<vector<int>> B (10);
    int tmp;

    for(int j = 1; j <= maxCypher; ++j){
        for(int i = 0; i < A.size(); ++i){
            tmp = getChypher(A[i], j);
            B[tmp].push_back(A[i]);
        }
    }

    A.clear();

    for(int i = 0, k = 0; i < B.size(); ++i, k = 0){
        while( !(B[i]).empty() ){
            A.push_back(B[i].front());
            (B[i]).erase((B[i]).begin());
        }
    }

}
}
}

```

# Teoria dei numeri

Per alcune operazioni, la complessità per eseguirle è calcolata prendendo come misura il **numero di cifre** che compongono il numero.

Per esempio:

- Addizione  $\in O(n)$
- Moltiplicazione (scuola elementare)  $\in O(n^2)$

E' possibile andare a migliorare anche queste complessità attraverso degli **algoritmi di teoria dei numeri**

---

## Moltiplicazione Veloce

Si rappresentano i numeri nella seguente notazione:

$$A = A_s 10^{\frac{n}{2}} + A_d$$

Ad esempio:  $1342 = 13 * 10^2 + 42$ ,  $n = 4$

Allora abbiamo che:

$$AB = (A_s 10^{\frac{n}{2}} + A_d)(B_s 10^{\frac{n}{2}} + B_d) = A_s B_s 10^n + (A_s B_d + A_d B_s) 10^{\frac{n}{2}} + A_d B_d$$

Possiamo ulteriormente semplificare se notiamo che:

$$(A_s + A_d)(B_s + B_d) = (A_s B_d + A_d B_s) + A_s B_s + A_d B_d \Rightarrow$$

$$\Rightarrow (A_s B_d + A_d B_s) = (A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d$$

Troviamo quindi che la moltiplicazione di due numeri a  $n$  cifre può essere riscritta come:

$$AB = A_s B_s 10^n + ((A_s + A_d)(B_s + B_d) - A_s B_s - A_d B_d) 10^{\frac{n}{2}} + A_d B_d$$

```
In [ ]: void parteSx(int* half, int* whole, int n){
    for(int i = 0; i < n; i++){
        half[i] = whole[i];
    }
}
void parteDx(int* half, int* whole, int n){
    int end = 2 * n;
    for(int i = n; i < end; i++){
        half[i - n] = whole[i];
    }
}
```

```

void add(int* l, int* r, int* out, int n){
    int sum, resto = 0;
    for(int i = n - 1; i <= 0; i++){
        sum = l[i] + r[i] + resto;
        if(resto == 1)
            resto = 0;
        if(sum >= 10){
            resto = 1;
            sum %= 10;
        }
        out[i + 1] = sum;
    }

    out[0] = resto;
}

int moltiplicazioneVeloce(int* A, int* B, int n){
    if(n == 1)
        return A[0]*B[0];
    else{
        int* As = new int[n/2];
        int* Ad = new int[n/2];
        int* Bs = new int[n/2];
        int* Bd = new int[n/2];

        parteSx(As, A, n/2);           // O(n/2)
        parteDx(Ad, A, n/2);           // O(n/2)
        parteSx(Bs, B, n/2);           // O(n/2)
        parteDx(Bd, B, n/2);           // O(n/2)

        int* x1 = new int[n/2 + 1];
        int* x2 = new int[n/2 + 1];

        add(As, Bd, x1, n/2);         // O(n/2)
        add(Bs, Ad, x2, n/2);         // O(n/2)

        int y1 = moltiplicazioneVeloce(As, Bs, n/2);           // T(n/2)
        int y2 = moltiplicazioneVeloce(Ad, Bd, n/2);           // T(n/2)
        int y3 = moltiplicazioneVeloce(x1, x2, n/2) - y1 - y2; // T(n/2)

        return y1 * pow(10, n) + z3 * pow(10, n/2) + y2;
    }
}

```

La complessità di questo codice è:

- $T(1) = d$
- $T(n) = cn + 3T(n/2)$

Dalle classi di complessità note ricaviamo che questo è il caso:

$$\begin{cases} T(n) = d, & n \leq m \\ T(n) = cn^k + aT\left(\frac{n}{b}\right), & n > m \end{cases} \Rightarrow \begin{cases} \in O(n^k), & a < b^k \\ \in O(n^k \log n), & a = b^k \\ \in O(n^{\log_b a}), & a > b^k \end{cases}$$

Dove: 
$$\begin{cases} k = 1 \\ a = 3 \\ b = 2 \end{cases} \Rightarrow a > b^k \rightarrow 3 > 2^1$$

Perciò in nostro algoritmo avrà complessità:  $T(n) \in O(n^{\log_2 3}) = O(n^{1.59})$

---

## Serie di Fibonacci

La serie di fibonacci è una serie così definita:

$$x_n = \begin{cases} x_0 = 1 \\ x_1 = 1 \\ x_n = x_{n-1} + x_{n-2} \end{cases}$$

Questa serie venne proposta nel 1200 circa da *Lorenzo Fibonacci da Pisa* mentre studiava la dinamica delle popolazioni.

In particolare Fibonacci studiò questo problema:

- Una coppia di conigli adulti genera 2 conigli ogni anno
- Un coniglio diventa adulto dal 2° anno di vita.
- I conigli sono immortali e incestuosi

Partendo da una coppia non adulta:

- Anno 0: 1 coppia non adulta
- Anno 1: 1 coppia adulta
- Anno 2: 1 coppia adulta e 1 coppia non adulta
- Anno 2: 2 coppie adulte e 1 coppia non adulta
- Anno 3: 3 coppie adulte e 2 coppie non adulte

La serie di fibonacci è una serie che **converge**, infatti il rapporto all'infinito tra due elementi successivi è la **sezione aurea**:

$$\varphi = \lim_{n \rightarrow \infty} \frac{x_n}{x_{n-1}} = \frac{1+\sqrt{5}}{2} = 1,6183\dots$$

Questo numero compare in diversi ambiti della realtà:

- Rapporto diagonale/lato di un pentagono
- Livello di accrescimento della spirale logaritmica

Sviluppando il codice seguendo la definizione, ricaviamo questo:

```
In [ ]: int fibonacci(int n){
    if(n == 0)
        return 0;
    if(n == 1)
        return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

La complessità è la seguente:

- $T(0) = T(1) = d$
- $T(n) = b + T(n - 1) + T(n - 2)$

Perciò:  $T(n) \in O(2^n)$

Se invece non utilizziamo la definizione ricorsiva, ma sviluppiamo un codice iterativo:

```
In [ ]: int fibonacci(int n){
    int fMeno1 = 0, fMeno2 = 0, f = 1;

    for(int i = 1; i <= n; i++){
        fMeno2 = fMeno1;
        fMeno1 = f;
        f = fMeno1 + fMeno2;
    }
}
```

La complessità di questo codice è:  $T(n) \in O(n)$

Quindi l'iterazione è meglio? No, infatti un'altra versione ricorsiva:

```
In [ ]: int fibonacci(int n, int f = 0, int fMeno1 = 1){
    if(n == 0)
        return f;
    return fibonacci(n-1, f, f + fMeno1);
}
```

In questo caso, la complessità è la seguente:

- $T(0) = T(1) = d$
- $T(n) = b + T(n - 1)$

Quindi ha complessità:  $T(n) \in O(n)$

# **Standard Template Library (STL)**

E' una libreria che al suo interno ha diverse strutture dati comode, utili e efficienti.

Poiché gli oggetti sono *template*, è possibile per l'utente di sfruttarle utilizzando il tipo di dati che più ci fa comodo.

La libreria definisce:

- Contenitori: variabili che contengono i dati
- Iteratori: variabili usate per accedere agli elementi della struttura dati
- Algoritmi: permettono di valutare **con costo definito**

Le informazioni della STL si può trovare sul sito: <https://cplusplus.com>

---

## **Vector**

Il vector è una struttura dati che permette di emulare un array.

I vantaggi di utilizzare questa struttura dati è quella di mantenere il costo di accesso randomico  $O(1)$ , ma avere i vantaggi dell'allocazione dinamica.

## **Funzioni membro**

<b>(costruttore)</b>	Costruisce il vettore (funzione membro pubblica)
<b>(distruttore)</b>	Distruttore del vettore (funzione membro pubblica)
<b>operator=</b>	Assegna il contenuto (funzione membro pubblica)

## **Iteratori**

<b>begin</b>	Ritorna l'iteratore all'inizio (funzione membro pubblica)
<b>end</b>	Ritorna l'iteratore alla fine (funzione membro pubblica)
<b>rbegin</b>	Ritorna l'iteratore inverso all'inizio inverso (funzione membro pubblica)
<b>rend</b>	Ritorna l'iteratore inverso alla fine inversa (funzione membro pubblica)
<b>cbegin</b>	Ritorna il const_iterator all'inizio (funzione membro pubblica)
<b>cend</b>	Ritorna il const_iterator alla fine (funzione membro pubblica)
<b>crbegin</b>	Ritorna il const_reverse_iterator all'inizio inverso (funzione membro pubblica)
<b>crend</b>	Ritorna il const_reverse_iterator alla fine inversa (funzione membro pubblica)

# **Capacità**

<b>size</b>	Ritorna la dimensione (funzione membro pubblica)
<b>max_size</b>	Ritorna la dimensione massima (funzione membro pubblica)
<b>resize</b>	Cambia la dimensione (funzione membro pubblica)
<b>capacity</b>	Ritorna la dimensione della capacità di memoria allocata (funzione membro pubblica)
<b>empty</b>	Verifica se il vettore è vuoto (funzione membro pubblica)
<b>reserve</b>	Richiede una modifica della capacità (funzione membro pubblica)
<b>shrink_to_fit</b>	Riduce la capacità alla dimensione attuale (funzione membro pubblica)

# **Accesso agli elementi**

<b>operator[]</b>	Accede all'elemento (funzione membro pubblica), costo costante
<b>at</b>	Accede all'elemento (funzione membro pubblica)
<b>front</b>	Accede al primo elemento (funzione membro pubblica)
<b>back</b>	Accede all'ultimo elemento (funzione membro pubblica)
<b>data</b>	Accede ai dati (funzione membro pubblica)

# **Modificatori**

<b>assign</b>	Assegna il contenuto del vettore (funzione membro pubblica)
<b>push_back</b>	Aggiunge un elemento alla fine (funzione membro pubblica)
<b>pop_back</b>	Elimina l'ultimo elemento (funzione membro pubblica)
<b>insert</b>	Inserisce elementi (funzione membro pubblica)
<b>erase</b>	Cancella elementi (funzione membro pubblica)
<b>swap</b>	Scambia i contenuti (funzione membro pubblica)
<b>clear</b>	Cancella il contenuto (funzione membro pubblica)
<b>emplace</b>	Costruisce ed inserisce un elemento (funzione membro pubblica)
<b>emplace_back</b>	Costruisce ed inserisce un elemento alla fine (funzione membro pubblica)
<b>sort(begin, end)</b>	Ordina il vettore tra i due estremi. Ha complessità garantita $O(n \log(n))$
<b>sort(begin, end, comparatore)</b>	Ordina il vettore tra i due estremi attraverso un comparatore booleano. Ha complessità garantita $O(n \log(n))$

# **Allocator**

<b>get_allocator</b>	Ottiene l'allocatore (funzione membro pubblica)
----------------------	---

# **Overload funzioni non membro**

**operatori relazionali** Operatori relazionali per vettore (modello di funzione)

**swap** Scambia i contenuti dei vettori (modello di funzione)

In [ ]: `#include <vector>`

```
int main(){
    vector<int> a;
    // vettore vuoto di lunghezza 0
    vector<int> a(10, 0);
    // vettore di 10 elementi tutti inizializzati a 0
    a.sort(a.begin, a.end);
}
```

## **Heap**

La STL, nella libreria `<algorithm>`, sono già presenti i metodi:

**push\_heap()** Inserisce un elemento nell'intervallo dell'heap

**pop\_heap()** Estrae un elemento dall'intervallo dell'heap

**make\_heap()** Crea un heap dall'intervallo

**sort\_heap()** Ordina gli elementi dell'heap

**is\_heap()** Verifica se l'intervallo è un heap

**is\_heap\_until()** Trova il primo elemento che non è nell'ordine dell'heap

In [ ]: `#include <algorithm>`

```
vector<TypeInfo> v;

// Rende un Vector un Heap
make_heap(v.begin(), v.end());

// Restituisce il valore maggiore,
// Lo butta in fondo all'array,
// Riuaggusta l'heap fino a v.end() - 1
pop_heap(v.begin(), v.end());
```

## Priority Queue

E' una struttura dati dove in ogni istante viene tenuto conto dell'importanza degli elementi.

<b>empty()</b>	Verifica se il contenitore è vuoto
<b>size()</b>	Restituisce la dimensione
<b>top()</b>	Accede all'elemento in cima
<b>push()</b>	Inserisce un elemento
<b>emplace()</b>	Costruisce e inserisce un elemento
<b>pop()</b>	Rimuove l'elemento in cima
<b>swap()</b>	Scambia i contenuti

In [ ]: `#include <queue>`

```
priority_queue<int> prioQ;
prioQ.push(val);
prioQ.top();
prio.pop();
```

# **Tipo di accessi**

Esistono diversi tipi di accessi ai dati di una struttura:

- Accesso diretto: come nei vettori
  - Accesso lineare: liste (devo scorrere la lista)
- 

# **Alberi Binari**

Tipo di struttura dati estremamente ricorsiva.

La definizione è la seguente:

- Un albero vuoto (**NULL**) è un *albero binario*
- Un *nodo*  $p$  con due alberi binari  $B_s$  e  $B_d$  forma un albero binario

Secondo questa definizione:

- $p$ : è detto **radice**
- $B_s$ : è il **sottoalbero sinistro** di  $p$
- $B_d$ : è il **sottoalbero destro** di  $p$

Gli alberi binari sono una struttura **gerarchica**, dove ogni nodo è **etichettato** (ovvero contiene degli elementi).

Sono molto potenti quando vogliamo instaurare relazioni di ordine.

I nodi sono chiamati:

- *Nodo Radice*: nodo che non ha antecendenti.
- *Nodo padre*: nodo che ha almeno un discendente non **NULL**
- *Nodo figlio sx/dx*: nodo che è discendente (sx/dx) di un nodo padre
- *Foglia*: nodo che non ha discendenti
- *Nodi Antecedenti*: nodi padri rispetto ai nodi padri e così andando
- *Nodi Discendenti*: nodi figli rispetto ai nodi figli e così andando
- *Livello del nodo*: numero di antecendenti rispetto al nodo fino alla radice / numero di archi necessari per collegare un nodo alla radice
  - Il livello della radice è 0
  - Il livello di un albero **NULL** è generalmente assunto come -1
- *Livello dell'albero*: livello del nodo massimo

Proprio per la loro definizione ricorsiva, le funzioni che agiscono sugli alberi si prestano bene ad essere ricorsive.

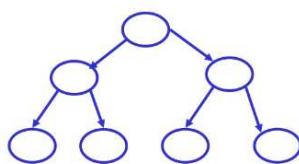
```
In [ ]: struct Node{  
    InfoType label;  
    Node* left;  
    Node* right;  
}
```

## Alberi Binari Bilanciati

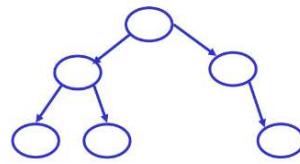
In un albero bilanciato, i nodi di tutti i **livelli** (tranne gli ultimi), hanno due figli.

In un albero bilanciato di livello  $k$  abbiamo:

- $2^{k+1} - 1$  nodi
- $2^k$  foglie



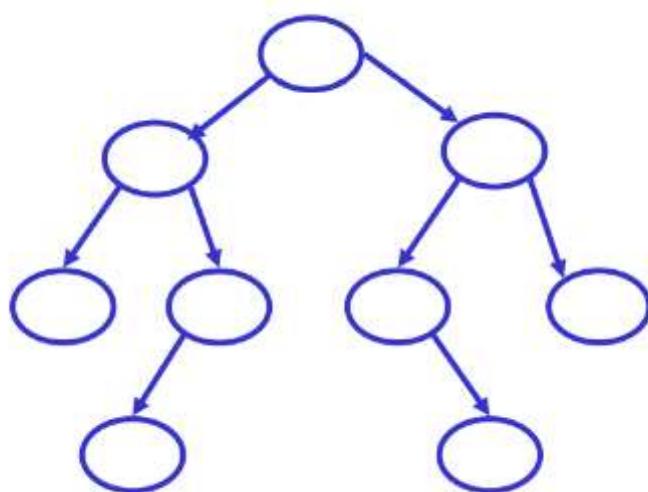
**bilanciato**



**non bilanciato**

## Alberi Binari quasi bilanciati

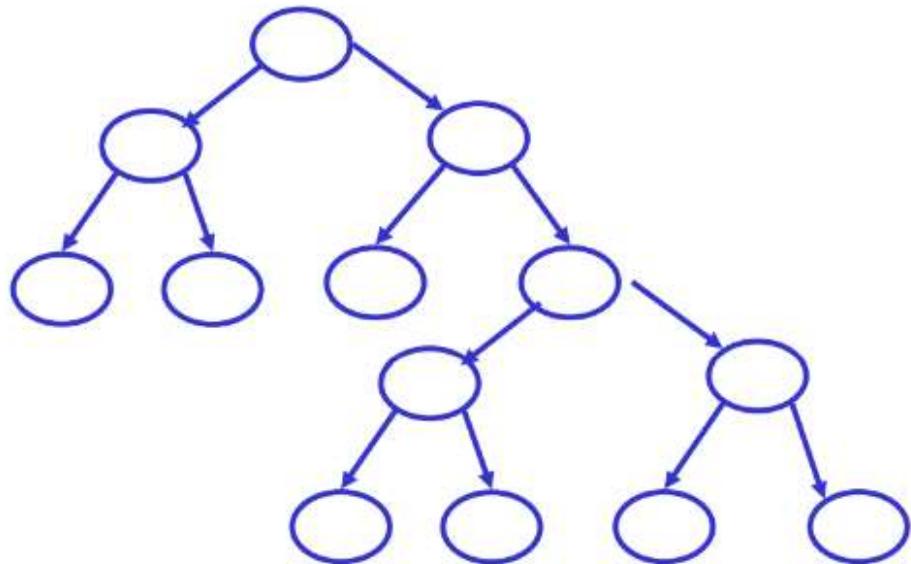
Albero che fino al penultimo livello è un albero bilanciato



## **Alberi pienamente Binari**

In un albero pienamente binario tutti i **nodi**, tranne le foglie, hanno 2 figli.

Il numero di nodi interni di un albero pienamente binario è il numero delle foglie meno 1.



## **Visite di Alberi Binari**

Le operazioni più comuni sugli alberi sono:

- Linearizzazione
- Ricerca
- Inserimento di nodi
- Cancellazione di nodi

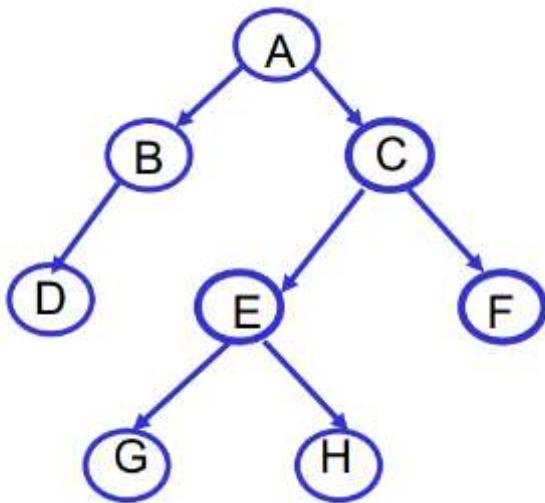
### **Linearizzazione**

Una linearizzazione di un albero è una *sequenza* contenente i nomi dei suoi nodi.

Le linearizzazioni più comuni, dette **visite**, sono tre:

- Ordine *anticipato* - **preorder**
- Ordine *differito* - **postorder**
- Ordine *simmetrico* - **inorder**

Le linearizzazioni verranno effettuate su questo albero:



### ***Preorder***

Esamina prima tutti i *Left*.

Ritornando indietro esamina i *right*, sempre dando priorità ai *Left*

```
In [ ]: void preOrder(Node* tree){
    if(tree == nullptr) // (!tree)
        return;

    //esamina tree->label;
    preOrder(tree->left);
    preOrder(tree->right);
}
```

A B D C E G H F

### ***Postorder***

Esamina prima la foglia più a sinistra, per poi esaminare il *right* dello stesso padre.  
Poi risale.

```
In [ ]: void postOrder(Node* tree){
    if(tree == nullptr)
        return;

    postOrder(tree.left);
    postOrder(tree.right);
    //esamina tree->label;
}
```

D B G H E F C

### ***Inorder***

Dato un qualsiasi nodo, considera prima il sottoalbero sinistro, poi la radice, poi il sottoalbero destro

```
In [ ]: void inOrder(Node* tree){
    if(tree == nullptr)
        return;

    inOrder(tree.left);
    //esamina tree->label;
    inOrder(tree.right);
}
```

D B A G E H C F

### Complessità visite

Complessità in funzione del numero di nodi:

- $T(0) = a$
- $T(n) = b + T(n_s) + T(n_d) \quad \wedge \quad n_s + n_d = n - 1 \quad n > 0$

Un caso particolare che possiamo prendere è quando:  $n_s = n_d$ , ovvero:

- $T(0) = a$
- $T_{s=d}(n) = b + 2T((n - 1)/2)$

Perciò  $\Rightarrow T_{s=d}(n) \in O(n)$

La complessità di una visita di un *albero bilanciato*, in dipendenza del numero di livelli (e non dei nodi):

- $T(0) = a$
- $T(k) = b + 2T(k - 1)$

Perciò  $\Rightarrow T(k) \in O(2^k)$

Per quanto possa sembrare che la complessità sia pessima, dobbiamo considerare che generalmente  $k \ll n$

## Funzioni su Alberi

Di seguito ci sono alcune funzioni sugli alberi binari, con le complessità

### Conta Nodi

Dato un albero, contare i nodi che fanno parte dell'albero.

```
In [ ]: int nodes(Node* tree){
    if(!tree)
        return 0;
    return 1 + nodes(tree->left) + nodes(tree->right);
}
```

- $T(0) = a$
- $T(n) = b + T(n_s) + T(n_d)$

Questa funzione può essere assimilata ad una *visita*.

Poiché visita tutto l'albero  $\Rightarrow T(n) \in O(n)$

### **Conta Foglie**

```
In [ ]: int leaves(Node* n){
    if(!tree)
        return 0;

    if(!tree->left && !tree->right)
        return 1;

    return leaves(tree->left) + leaves(tree->right);

}
```

- $T(0) = T(\text{foglia}) = a$
- $T(n) = b + T(n_s) + T(n_d)$

Poiché visita tutto l'albero  $\Rightarrow T(n) \in O(n)$

### **Ricerca Etichetta**

Resituire un puntatore al nodo che contiene l'etichetta. Se non presente restituire un puntatore nullo, se c'è più colte restituire la prima visitata con la visita anticipata

```
In [ ]: Node* findNode(InfoType n, Node* tree){
    if(!tree)
        return nullptr;
    if(tree->label == n)
        return tree;
    Node* a = findNode(n, tree->left);

    return (a)?: a : findNode(n, tree->right);
}
```

- $T(0) = T(\text{foglia}) = a$
- $T(n) = b + T(n_s) + T(n_d)$

Poiché nel peggiore dei casi visita tutto l'albero  $\Rightarrow T(n) \in O(n)$

## Cancella l'albero

Eliminare tutto l'albero e mettere il puntatore a **NULL**;

```
In [ ]: void deTree(Tree* &tree){
    if(!tree)
        return;

    delTree(tree->left);
    delTree(tree->right);
    delete tree;
    tree = nullptr;
}
```

$T(n) \in O(n)$ , scorre tutto l'albero

E' assimilabile ad una *postorder*

## Inserisci Nodo

Inserisce un nodo (son) come figlio di *father*, sinistro se c = 'l', destro se c = 'r'.

La funzione restituisce 1 se ha successo, 0 se fallisce.

Se l'albero è vuoto lo inserisce come radice.

Se *father* non compare o ha già un figlio in quella posizione, non modifica l'albero

```
In [ ]: int insertNode(Node* &tree, InfoType son, InfoType father, char c){
    if(!tree){
        tree = new Node;
        tree->label = son;
        tree->left = tree->right = nullptr;
        return 1;
    }

    Node* a = findNode(father, tree);

    if(!a)
        return 0;

    if(c == 'l' && !a->left){
        a->left = new Node;
        a->left->label = son;
        a->left->left = a->left->right = nullptr;
        return 1;
    }
    if(c == 'r' && !a->right){
        a->right = new Node;
        a->right->left = a->right->right = nullptr;
        return 1;
    }
    return 0;
}
```

$T(n) = O(n)$

---

## Classe BinTree

```
In [ ]: template<class InfoType>

class Bintree{
    struct Node{
        InfoType label;
        Node* left;
        Node* right;
    };
    Node* root;
    Node* findNode(InfoType, Node*);
    void preOrder(Node*);
    void postOrder(Node*);
    void inOrder(Node*);
    void delTree(Node* &);
    int insertNode(Node*&, InfoType, InfoType, char);
public:
    BinTree(){
        root = nullptr;
    };
    ~BinTree(){
        delTree(root);
    };

    // termina, slide 28 parte 2a
}
```

---

## Alberi Generici

La definizione di un albero generico:

- Un nodo  $p$  è un *albero*
- Un nodo  $p$  più una sequenza di alberi  $A_1, \dots, A_n$  è un *albero*

Oltre alle definizioni degli alberi binari abbiamo:

- **Arco(u,v)**: collegamento tra il nodo  $u$  e il nodo  $v$
- **Fratelli**: nodi che hanno lo stesso padre
- **Nodo Interno**: né radice, né foglia

A differenza degli alberi binari, non c'è più differenza tra sottoalberi *destri* e *sinistri*, ma si considerano semplicemente *sottoalberi*.

## Rappresentazione

Per rappresentarlo si può generalizzare la struttura degli alberi binari.

Sapendo che un nodo può avere al massimo  $k$  il numero di sottoalberi:

```
In [ ]: struct Node{
    InfoType label;
    Node* subNode1;
    Node* subNode2;
    ...
    Node* subNodeK;
};
```

## Visite

Anche per gli alberi generici esistono visite *preorder* e *postorder*.

Non ha invece senso parlare di *inorder*, poiché non abbiamo più la capacità di individuare un "punto di mezzo".

```
In [ ]: void preOrder(albero){
    if(!albero)
        return;

    <esamino la radice>
    foreach sottoalbero in albero{
        preOrder(sottoalbero);
    }
}

void postOrder(albero){
    if(!albero)
        return;

    foreach sottoalbero in albero{
        preOrder(sottoalbero);
    }

    <esamino la radice>
}
```

## Memorizzazione Figlio-Fratello

Gli alberi generici occupano una grande quantità di memoria, e possono diventare "ingestibili".

Per ovviare al problema della memoria si utilizza la memorizzazione figlio-fratello, che sfrutta i concetti del *Binary Tree*.

In questa memorizzazione ogni nodo ha:

- Etichetta
- Puntatore al primo figlio (sx)
- Puntatore al primo fratello (dx)

Dato un nodo:

- A sinistra abbiamo il sottoalbero figlio (livello + 1)
- A destra abbiamo i fratelli (stesso livello)

Questa memorizzazione in memoria **non è** l'albero generico, ma solamente come l'albero è salvato in memoria

```
In [ ]: struct Node{  
    InfoType label;  
    Node* son;  
    Node* brother;  
}
```

## Corrispondenze tra Viste

Quando prendiamo un albero generalizzato, e il suo trasformato abbiamo che:

- Vista preorder generalizzato  $\equiv$  Vista preorder trasformato
- Vista postorder generalizzato  $\equiv$  Vista Inorder trasformato

In tutti i casi il tempo delle visite continua ad essere **lineare** nel numero di nodi, così come *cancellazione, inserimento e ricerca*.

---

## Funzioni

### Conta Nodi e Conta Foglie

```
In [ ]: int nodes(Node* tree){  
    if(!tree)  
        return 0;  
  
    return 1 + nodes(tree->son) + nodes(tree->bro);  
}  
  
int leaves(Node* tree){  
    if(!tree)  
        return 0;  
  
    if(!tree->son)  
        return 1;  
  
    return leaves(tree->son) + leaves(tree->bro);  
}
```

## Inserisci l'ultimo figlio di un nodo

Inserisci un nodo  $F$  come ultimo nodo figlio di un nodo padre  $A$

```
In [ ]: bool insertNode(InfoType label, Node* &tree){  
    if(!tree){  
        tree = new Node;  
        tree->label = son;  
        tree->left = tree->right = nullptr;  
        return true;  
    }  
  
    if(tree->label == label)  
        return false;  
  
    return insertNode(label, tree->right);  
}  
  
bool insLastSon(InfoType son, InfoType father, Node* &tree){  
    if(!tree){  
        tree = new Node;  
        tree->label = son;  
        tree->left = tree->right = nullptr;  
        return true;  
    }  
    if(tree->label == son)  
        return false;  
  
    if(tree->label == father){  
        return insertNode(son, tree->left);  
    }  
  
    if(insLastSon(son, father, tree->left))  
        return true;  
  
    return insLastSon(son, father, tree->right);  
}
```

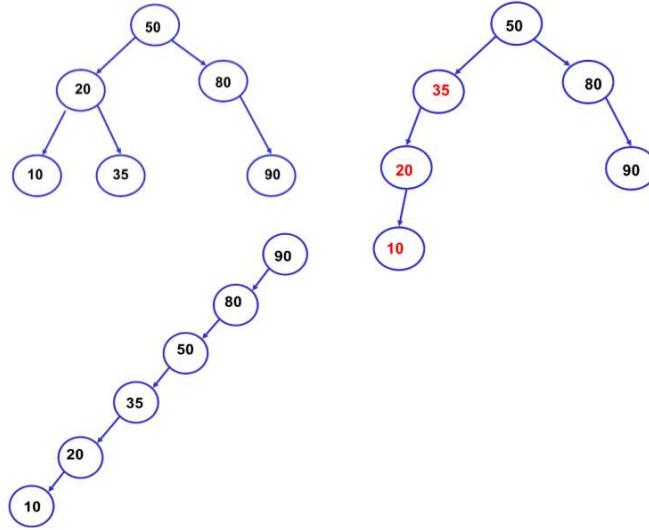
---

## Alberi Binari di Ricerca

Sono alberi binari tale che per ogni nodo  $p$ :

- I nodi del sottoalbero sinistro di  $p$  hanno **etichetta minore di  $p$**
- I nodi del sottoalbero destro di  $p$  hanno **etichetta maggiore di  $p$**

Gli alberi binari di ricerca, dato un insieme non ordinato di nodi, **non hanno un'unica forma**, possono assumere forme diverse, avendo livelli diversi, a seconda dell'ordine in cui gli elementi sono inseriti.



Alcune proprietà degli alberi binari di ricerca sono:

- Non ci sono elementi duplicati
- La *visita simmetrica* elenca le etichette in **ordine crescente**

Le operazioni più comuni sono:

- Ricerca di un nodo
- Inserimento di un nodo
- Cancellazione di un nodo

## Ricerca di un Nodo

A differenza degli alberi binari, dove non abbiamo a priori informazioni su come i nodi sono distribuiti, negli alberi binari di ricerca sappiamo che i valori inferiori sono sul sottoalbero di sx mentre i valori superiori sono a dx

```
In [ ]: Node* findNode(InfoType n, Node* tree){
    if(!tree)
        return nullptr;

    if(n == tree->label)
        return tree;

    return (n < tree->label)? findNode(n, tree->left) : findNode(n, tree->right)
}
```

- $T(0) = a$
- $T(n) = b + T(k), \quad k < n$

Se ipotizziamo che:

- $k = \frac{n}{2} \Rightarrow T(n) \in O(\log n)$  (albero bilanciato)
- $k = n - 1 \Rightarrow T(n) \in O(n)$

Nell'*average case*:  $T(n) \in O(\log n)$ .

E' importante sottolineare che la proprietà che ha più influenza nella ricerca è **il livello dell'albero**.

## Inserimento di un Nodo

Quando inserisco un nodo seguo la politica dettata dalla definizione.

Questo comporta che la funzione sia *deterministica*, ovvero:

```
In [ ]: bool insNodo(InfoType n, Node* &tree){  
    if(!tree){  
        tree = new Node;  
        tree->label = n;  
        tree->left = tree->right = nullptr;  
        return true;  
    }  
  
    if(n < tree->label)  
        return insNodo(n, tree->left);  
  
    if(n > tree->label)  
        return insNodo(n, tree->right);  
  
    return false;  
}
```

- $T(0) = a$
- $T(n) = b + T(k), \quad k < n$

Se ipotiziamo che:

- $k = \frac{n}{2} \Rightarrow T(n) \in O(\log n)$  (albero bilanciato)
- $k = n - 1 \Rightarrow T(n) \in O(n)$

Nell'*average case*, così come la ricerca:  $T(n) \in O(\log n)$ .

Riempire un ABR da 0 con  $n$  elementi ha costo:  $T(n) \in O(n^2)$

## Cancellazione di un Nodo

La cancellazione di un nodo può avere 3 esiti:

1. Se  $p$  è una foglia  $\Rightarrow$  nessun problema
2. Se  $p$  ha un sottoalbero vuoto, il padre di  $p$  viene connesso all'unico sottoalbero non vuoto di  $p$
3. Se  $p$  ha entrambi i sottoalberi non vuoti, si cerca il nodo con etichetta minore nel sottoalbero destro di  $p$ , detto *successore*, si cancella e si mette la sua etichetta in  $p$ .
  - Il *successore* è sicuramente più grande del sottoalbero sinistro, e più piccolo di quello destro.

- La cancellazione del *successore* ci garantisce di essere in uno dei primi due casi, in quanto sicuramente non ha il figlio sinistro.

```
In [ ]: // Restituisce per riferimento l'etichetta del nodo più piccolo di un albero ed

void deleteMin(Node* &tree, InfoType &m){
    if(tree->left)
        deleteMin(tree->left, m);
    else{
        m = tree->label;
        Node* tmp = tree;
        tree = tree->right;
        delete tmp;
    }
}

void removeNode(InfoType n, Node* &tree){
    if(!tree)
        return;

    if(n < tree->label)
        removeNode(n, tree->left);
    else if(n > tree->label)
        removeNode(n, tree->right);
    else if(!tree->left){
        Node* a = tree;
        tree = tree->right;
        delete a;
    }
    else if(!tree->right){
        Node* a = tree;
        tree = tree->left;
        delete a;
    }
    else
        deleteMin(tree->right, tree->label);
}
```

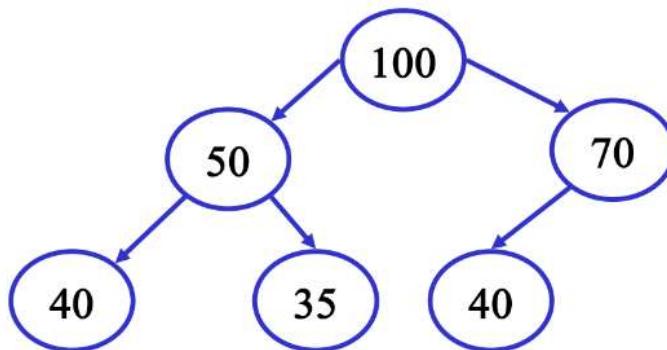
# Heap

Sono degli alberi Binari *quasi bilanciati*, ovvero che ogni livello dell'albero ha il numero massimo di nodi, tranne l'ultimo, che ha però i nodi **addossati a sx**.

Ne esistono di due tipi:

- **MaxHeap**: In ogni sottoalbero, l'etichetta della radice è **maggiore o uguale** a quella di tutti i discendenti.
- **MinHeap**: In ogni sottoalbero, l'etichetta della radice è **minore o uguale** a quella di tutti i discendenti.

Le implementazioni che vedremo sono per il **MaxHeap**.



Questo tipo di struttura dati ha la proprietà di non perdere informazioni tra le relazioni padre e figlio quando l'Heap viene *linearizzato per livelli*.

Infatti è possibile stabilire relazioni numeriche tra l'indice di un nodo  $i$  e quello dei suoi figli e del padre:

- Figlio sx:  $2i + 1$
- Figlio dx:  $2i + 2$
- Padre:  $\frac{i-1}{2}$

In questo modo, preso il nodo di un *Heap* posso non solo risalire ai figli, ma anche al padre e alla radice stessa dell'albero.

L'*Heap* ha sicuramente altezza  $\log n$ .

Inoltre l'estrazione del massimo elemento dell'*heap* ha costo  $O(1)$  per via della definizione stessa, che fa coincidere il nodo con la radice.

L'*Heap* inoltre consente l'operazione di inserimento di un nodo, conservando le proprietà.

In memoria viene memorizzato come un'array.

```
In [ ]: class Heap{
    int* h;
    int last;

    // a partire da un nodo che viola le caratteristiche:
    // lo sposto verso l'alto
    void up(int);

    // lo sposto verso il basso
    void down(int);
    void exchange(int i, int j){
        int tmp = h[i];
        h[i] = h[k];
        h[k] = tmp;
    }
public:
    Heap(int);
    ~Heap();

    static void down(int*, int, int);
    static void extract(int*, int&);

    void insert(int);
    int extract();
};
```

## Metodi Private

```
In [ ]: Heap::Heap(int n){
    h = new int[n];
    last = -1;
}

Heap::~Heap(){
    delete[] h;
}

/// Se non sono la radice, Controllo se sono più grande del padre.
/// Se lo sono mi scambio e controllo ricorsivamente la nuova posizione
/// Ha complessità O(log(n)), ovvero il numero di livelli.
void Heap::up(int i){
    if(i > 0){
        if(h[i] > h[(i-1)/2]){
            exchange(i, (i-1)/2);
            up((i-1)/2);
        }
    }
}

/// Se ho un solo figlio (sono l'ultimo dell'array)
/// Controllo semplicemente se il figlio è maggiore del padre
/// eventualmente faccio lo scambio
/// Altrimenti, se ho entrambi i figli invece
/// Cerco prima il più grande
/// se il figlio è maggiore del padre faccio lo scambio.
/// In quest'ultimo caso controllo che la nuova posizione mantenga le proprietà
/// Ha complessità O(log(n)), ovvero il numero di livelli
```

```

int Heap::down(int i){
    int son = 2*i + 1;

    if(son == last){
        if(h[son] > h[i])
            exchange(i, last);
    }
    else if(son < last){
        if(h[son] < h[son + 1])
            son++;

        if(h[son] > h[i]){
            exchange(i, son);
            down(son);
        }
    }
}

```

## Metodi Public

```

In [ ]: // Inserire un elemento nella prima posizione libera dell'array
// Successivamente lo si fa risalire per rispettare le proprietà dell'Heap
void Heap::insert(int label){
    h[++last] = label;

    up(last);

}

/// Restituisco il primo elemento dell'array
/// Poi metto l'ultimo elemento al posto della radice e decremento last
/// Infine faccio scendere l'elemento per mantenere la proprietà dello heap
int Heap::extract(){
    int output = h[0];

    h[0] = h[last--];
    down(h[0]);
}

```

## Code di Priorità

Gli Heap sono particolarmente indicati per l'implementazione del tipo di dato astratto **Coda con Priorità**.

Una coda di priorità è una coda in cui gli elementi contengono l'informazione e un intero che ne definisce la priorità.

In caso di estrazione, l'elemento da estrarre deve essere quello con maggiore priorità.

## Heap Sort

E' un algoritmo che ordina un vettore sfruttando le proprietà dello *Heap*.

Ridefinendo le funzioni `down` e `insert`, affinché possano lavorare anche su array di interi generici:

```
In [ ]: void buildHeap(int* A, int n){
    // i = n/2 - 1, è l'ultimo nodo non foglia
    for(int i = n/2 - 1; i >= 0; --i)
        down(A, i, n);
}

void heapSort(int* A, int n){
    buildHeap(A, 5);
    int* B = new int[n];
    for(int i = n - 1; i >= 0; --i){
        B[i] = extract(A, 0, n);
    }

    delete[] A;
    A = B;
}
```

E' possibile dimostrare che la complessità della `buildHeap`  $\in O(n)$ .

La `extract`, eseguita  $n$  volte, ha invece complessità complessiva  $O(n \log n)$

# **Hashing**

I metodi di *hashing* sono degli strumenti per ottimizzare e migliorare i limiti inferiori di diversi algoritmi.

## **Hashing e Ordinamento**

Habbiamo già dimostrato le complessità per la ricerca di un elemento in un insieme:

- Linear Search:  $O(n)$  nel Worst Case,  $O(\frac{n}{2})$  nel Average Case
- Binary Search:  $O(\log n)$  nel Worst e nell'Average Case (l'insieme deve però essere ordinato)

E' possibile migliorare questi limiti inferiori, non basandosi più sul confronto, ma sull'*hashing*:

### **hashSort**

E' un metodo di ricerca di un elemento generico in un'array, non basato sui confronti.

E' molto efficiente, avendo complessità di ricerca che può diventare nell'Average Case persino  $O(1)$ .

Per permettere ciò viene definita una funzione *hash*, che trasforma ogni elemento in un indice di un nuovo array.

$h(\text{funzione hash}) : \text{InfoType} \rightarrow \text{indici}$

$$x \rightarrow h(x) \rightarrow \begin{pmatrix} 0 \\ 1 \\ \vdots \\ k-1 \end{pmatrix}$$

La funzione deve essere biunivoca, poiché:

- Iniettività  $\Rightarrow$  ad ogni elemento associo un solo indice, così da poter fare *backtracking* univoco
- Suriettività  $\Rightarrow$  per occupare solamente la memoria necessaria, e non avere dei "buchi" vuoti.

L'unico svantaggio di questo algoritmo è lo spazio in memoria, che alloca una memoria  $O(k)$ , dove  $k$  indica il numero di elementi distinti nella nostra *hash table*, che deve essere necessariamente maggiore o uguale alla cardinalità dell'insieme di partenza.

```
In [ ]: bool hashSearch(InfoType *A, int n, InfoType x){
    return (A[h(x)] == x);
}
```

La funzione  $h(x)$  è una funzione *hash* che mappa ad  $x$  la sua immagine-indice.

## Metodo Hash senza accesso diretto

Granatire l'iniettività provoca però una crescita esponenziale nella quantità di memoria necessaria per salvare la *hash-table*.

Perciò si rilascia l'iniettività e si permette che due elementi diversi abbiano lo stesso indirizzo hash.

Questo comporta però collisioni  $h(x_1) = h(x_2)$ , che vanno a rendere problematico l'inserimento degli elementi.

Alcuni metodi per risolvere questi problemi:

## Metodo Hash ad indirizzamento Aperto

- Funzione hash modulare:  $h(x) = x \% k$ , dove  $k$  è la dimensione dell'array
- Legge di scansione lineare: se non si trova l'elemento al suo posto, lo cerca linearmente nelle posizioni successive
- Inserimento a scansione lineare permanente

Questa scelta va a generare un **agglomerato**: gruppo di elementi adiacenti con indirizzi hash diversi da  $h(x)$

```
In [ ]: const int EMPTY = -1;

const int k;

int h(int x){
    return x % k;
}

bool hashSearch(InfoType *A, int x){
    int i = h(x);
    for(int j = 0; j < k; ++j){
        int pos = (i+j) % k;
        if(A[pos] == EMPTY)
            return false;
        if(A[pos] == x)
            return true;
    }
    return false;
}
```

```

bool hashInsert(InfoType *A, int x){
    int i = h(x);

    for(int j = 0; j < k; ++j){
        int pos = (i+j) % k;
        if(A[pos] == EMPTY){
            A[pos] = x;
            return true;
        }
    }

    return false;
}

```

### Cancellazione di un elemento

Se nel nostro metodo hash volessimo cancellare un elemento, avremmo la necessità di inserire al suo posto un valore diverso da quello di *no-element* (-1).

Se non facessimo in così, quando andremmo a cercare un elemento che dovrebbe trovarsi in quella posizione, ma è stato spostato poiché era occupata, il valore -1 (dovuta all'eliminazione), ci restituirebbe che quell'elemento non è presente.

Inserendo invece un valore diverso, potremmo continuare a cercarlo nelle posizioni successive.

```

In [ ]: const int EMPTY = -1;
          const int DELETED = -2;

          const int k;

          bool hashInsert(InfoType *A, int x){
              int i = h(x);

              for(int j = 0; && j < k; ++j){
                  int pos = (i+j) % k;
                  if(A[pos] == EMPTY || A[pos] == DELETED){
                      A[pos] = x;
                      return true;
                  }
              }

              return false;
}

          bool hashSearch(InfoType *A, int x){
              int i = h(x);

              for(int j = 0; j < k; ++j){
                  int pos = (i+j) % k;
                  if(A[pos] == EMPTY)
                      return false;
                  if(A[pos] == x)
                      return true;
              }

              return false;
}

```

La complessità delle *hash-table ad indirizzamento aperto* dipende dall'agglomerato, perciò possiamo andare a cambiare come vado a scansionare gli elementi quando le posizioni sono occupate:

- Scansione Lineare:
  - $(x; j) = (h(x) + cost * j) \bmod k$
- Scansione Quadratica:
  - $(x; j) = (h(x) + cost * j^2) \bmod k$
  - Questa scansione permette di diradare le collisioni e gli agglomerati, per evitare collisioni multiple, tuttavia questa scelta rischia di far fallire l'inserimento anche se l'array non fosse pieno.
  - Se  $k = 2^l$  allora siamo sicuri di mappare tutte le posizioni
  - Se  $k \neq 2^l$  allora abbiamo un riempimento di circa il 50%

Il tempo medio di ricerca (numero di confronti) dipende da:

- **Fattore di carico:** rapporto  $\alpha = \frac{n}{k} \leq 1$ , numero medio di elementi per ogni posizione;
  - $O(k) \rightarrow O(1)$
- **Legge di scansione:** migliora con la scansione quadratica ed altre più sofisticate;
- **Uniformità della funzione hash:** genera gli indici con uguale probabilità.

Possiamo stimare il numero medio di accessi:  $\leq \frac{1}{1-\alpha}$

$\frac{n}{M}$	scansione lineare	scansione quadratica
10%	1.06	1.06
50%	1.50	1.44
70%	2.16	1.84
80%	3.02	2.20
90%	5.64	2.87

Nell'inserimento aperto, a seguito di molti inserimenti/cancellazioni, il tempo di ricerca viene degradato a causa degli agglomerati, perciò è necessario "risistemare" periodicamente l'array, per riottimizzarlo.

## Metodi di Concatenazione

Traduco le collisioni, inserendo per ogni posizione una *lista* di elementi.

Questo traduce la nostra *hash-table* in un array di puntatori, di dimensione  $k \leq n$ . Gli elementi che collidono su  $i$  diventano elementi della lista in posizione  $A[i]$

Per definizione gli agglomerati sono del tutto evitati.

Il costo di effettuare un inserimento, effettuato in testa, è  $O(1)$ .

Il costo di effettuare una cancellazione, nel peggiore dei casi, è di  $O(n)$

In generale la complessità di ricerca è  $O(\alpha)$

---

## HashTable (Dizionario)

Un dizionario è una struttura dati che associa ad ogni informazione una chiave **key**, che permette di raggiungere un'informazione tramite una funzione *hash*.

Supporta:

- Ricerca
- Inserimento
- Cancellazione

Tramite i dizionari andiamo a creare una struttura dati estremamente efficiente per ricerca, inserimento e cancellazione, (tendente a  $O(1)$ ).

In un dizionario il dominio degli input è **eterogeneo**, è possiede una cardinalità maggiore di quella della memoria disponibile. Ad ogni elemento viene associata una posizione proprio grazie ad una funzione *hash()*.

## Hashing di Interi

Per progettare una tabella **Hash** bisogna innanzitutto la struttura dati alla base a seconda del fatto che ci serva o meno espandere dinamicamente il numero di valori di output.

Nel nostro caso utilizzeremo una dimensione fissa.

Va poi specificata la chiave da utilizzare per ogni valore, nel nostro caso prendiamo:

- Interi  $> 0$
- Chiave coincide con il valore
- 0 indica un elemento vuoto

Per la funzione di HASH utilizziamo la *funzione modulo*.

```
In [ ]: class HashTable{
    int* table_;
    int size_;

    const int EMPTY = 0;

    int hash(int);
public:
    HashTable(int);
    ~HashTable();

    bool insert(int);
    void print();
};
```

```

// Metodi Privati

int HashTable::hash(int key){
    return key % size_;
}

// Metodi Pubblici

HashTable::HashTable(int size){
    size_ = size;
    table_ = new int[size];

    cout << size_ << endl;
    memset(table_, 0, size*sizeof(int));
}

HashTable::~HashTable(){
    delete[] table_;
    table_ = nullptr;
}

bool HashTable::insert(int key){
    int hashKey = hash(key);

    if(table_[hashKey] == EMPTY){
        table_[hashKey] = key;
        return true;
    }

    return false;
}

void HashTable::print(){
    for(int i = 0; i < size_; ++i)
        cout << table_[i] << "\t";

    cout << endl;
}

```

Tuttavia questa implementazione non risolve le collisioni.

Per poterle gestire esistono più metodi, quello che svilupperemo in seguito è il metodo delle **Liste di Trabocco**.

Il nostro vettore di partenza non sarà più un vettore di interi, bensì un vettore di liste, ognuna avente la seguente struttura:

```

In [ ]: struct Elek{
    int val;
    Elek* next;
    Elek* pred;

    Elek(int v, Elek* n, Elek* p): val(v), next(n), pred(p) {}

};

```

```

class HashTable{
    Elem** table_;
    int size_;

    int hash(int);
public:
    HashTable(int);
    ~HashTable();

    bool insert(int);
    void print();
};

HashTable::HashTable(int size){
    size_ = s;
    table = new Elem*[size_];

    for(int i = 0; i < size_; ++i)
        table[i] = nullptr;

}

```

Per implementare la nostra classe consideriamo l'inserimento in testo di un nuovo elemento  $O(1)$ .

L'implementazione sarà la seguente:

```

In [ ]: bool HashTable::insert(int key){
    int hashKey = hash(key);

    Elem* tmp = table[hashKey];
    table[hashKey] = new Elem(Key, tmp, nullptr);
    if(tmp != nullptr)
        tmp->pred = table[hashKey];
}

```

## Hashing di Stringhe

La potenza delle Tabelle Hash, è il fatto che la **key** possa essere non necessariamente un numero, ma anche un carattere.

Infatti prendendo una stringa come valore, posso riconfigurare la mia funzione *hash()* in base alla prima lettera della stringa:

```

In [ ]: int hash(string key){
    int index = key[0] % size_;
}

```

Anche in questo caso posso sviluppare una tabella hash con **liste di trabocco**.

Questa scelta però si rileva poco efficente, in quanto avremo molte parole che iniziano con la stessa lettera, ed altre lettere che invece tendenzialmente non appariranno mai.

Mantenendo la politica delle **liste**, posso modificare la mia funzione *hash*:

```
In [ ]: int hash(string key){  
    int index = 0;  
  
    for(int i = 0; i < key.length(); ++i)  
        index = (index + key[i]) % size;  
}
```

## Funzioni Hash

Non esiste una funzione hash migliore in maniera assoluta, ma dipende fortemente dal tipo di applicazione.

Nelle applicazioni di indexing è fondamentale l'**uniformità**.

Le funzioni "migliori" per l'indexing si basano sulla rappresentazione binaria, alcuni esempi sono "*MurmurHash*", "*CityHash*", "*FarmHash*",...

---

## <map>

Nella **STL** è implementata una struttura dati chiamata **MAP** così definita:

```
In [ ]: map<key_T , obj_T> table;
```

Ogni elemento è definito da una coppia (*pair*) *<key, obj>* che possono essere di qualsiasi tipo.

Questa struttura dati permette l'indicizzazione tra parentesi quadre delle chiavi.

```
In [ ]: table["uno"] = "valore uno";  
table.find("uno");
```

# **Programmazione Dinamica**

Sono delle azioni di programmazione che dividono un problema in sottoproblemi, permettendo di enumerare le soluzioni di tutti i sottoproblemi per poi utilizzarli per risolvere il problema più complesso.

Si può utilizzare quando non è possibile sfruttare il metodo *divide et impera* per uno dei due seguenti casi:

- Non si sa quali sottoproblemi risolvere
- Non è possibile partizionare l'insieme in sottoinsiemi disgiunti

Il metodo consiste nel **risolvere tutti i sottoproblemi** a partire dal basso, conservando via via i risultati così da poterli utilizzare successivamente (strategia *bottom-up*).

La complessità del metodo dipenderà dal numero di sottoproblemi.

Si può utilizzare quando abbiamo:

- **Sottostrutture ottime**: soluzione ottima del problema  $\Rightarrow$  soluzione ottima dei sottoproblemi
- **Sottoproblemi comuni**: algoritmo ricorsivo richiederebbe di risolvere lo stesso sottoproblema più volte

## **Più lunga sottosequenza comune (PLSC)**

Date due sequenze:

- $\alpha = \alpha_1 \dots \alpha_i \dots \alpha_m$
- $\beta = \beta_1 \dots \beta_j \dots \beta_n$

Vogliamo prendere la più lunga sequenza  $\beta_{j_1} \dots \beta_{j_k}$  che appare anche in  $\alpha$ , potendo rimuovere caratteri  $\alpha_i$ .

La lunghezza delle PLSC delle sequenze  $\alpha_1 \dots \alpha_i$  e  $\beta_1 \dots \beta_j$  data dalla funzione:

$$L(i, j)$$

Sicuramente possiamo dire che:

- Se una delle due sequenze è vuota (o entrambe) la lunghezza è 0
  - $L(0, 0) = L(i, 0) = L(0, i) = 0$
- Se gli elementi in ultima posizione sono uguali, allora la lunghezza della stringa è 1 + quella della sottostringa precedente
  - $L(i, j) = L(i - 1, j - 1) + 1, \quad \alpha_i = \beta_j$
- Se gli ultimi elementi sono diversi, la lunghezza sarà il massimo tra la lunghezza tra la sottosequenza con  $\alpha$  più corta e quella con  $\beta$  più corta
  - $L(i, j) = \max(L(i, j - 1), L(i - 1, j)), \quad \alpha_i \neq \beta_j$

Abbiamo considerato delle sottosequenze generiche:

- $i \leq m$
- $j \leq n$
- Se la sequenza  $\chi$  è una PLSC tra  $\alpha$  e  $\beta$ , allora conterrà sicuramente una PLSC tra le due sotto-sequenze

```
In [ ]: int lenght(char* a, char* b, int i, int j){
    if(i == 0 || j == 0)
        return 0;
    if(a[i] == b[j])
        return lenght(a,b, i-1, j-1) + 1;
    else
        return max(lenght(a,b, i, j-1), lenght(a,b,i-1, j));
}
```

La relazione di ricorrenza ha costo:

$$T(k) = b + 2T(k - 1) \in O(2^k), \quad k = \min(m, n)$$

L'algoritmo in **programmazione dinamica**, costruisce invece  $L(i, j)$  a partire dagli indici più piccoli:  
 $L(0, 0), \dots, L(0, n)$

$$L(1, 0), \dots, L(1, n) \dots L(m, 0), \dots, L(m, n)$$

```
In [ ]: const int m = 7;
const int n = 6;

int L[m+1], L[n+1];
int quickLength(char* a, char* b){
    for(int j = 0; j <= n; ++j){
        L[0][j] = 0;
    }
    for(int i = 1; i <= m; ++i){
        L[i][0] = 0;
        for(int j = 1; j <= n; ++j){
            if(a[i] != b[j])
                L[i][j] = max(L[i][j-1], L[i-1][j]);
            else
                L[i][j] = L[i-1][j-1] + 1;
        }
    }
    return L[m][n];
}
```

Uno schema di cosa fa questo algoritmo:

	c	b	a	b	a	c	
a							
b							
c							
a							
b							
b							
a							

	c	b	a	b	a	c	
a	0	0	0	1	1	1	
b	0	0	1	1	2	2	
c	0	1	1	1	2	2	
a	0	1	1	2	2	3	
b	0	1	2	2	3	3	
b	0	1	2	3	3	3	
a	0	1	2	3	3	4	

La complessità di questo algoritmo è:

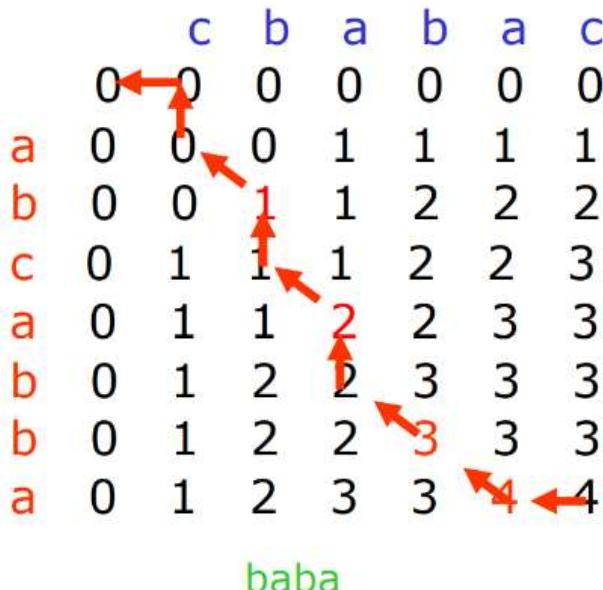
$$T_n \in O(nm)$$

Possiamo inoltre fare *back-tracking* e trovare una delle stringhe più lunghe:

```
In [ ]: void print(int **L, char* a, char* b, int i = m, int j = n){
    if(i == 0 || j == 0)
        return;

    if(a[i] == b[j]){
        print(a, b, i-1, j-1);
        cout << a[i];
    }
    else if(L[i][j] == L[i-1][j])
        print(a, b, i-1, j);
    else
        print(a, b, i, j-1);
}
```

Uno schema di cosa fa questo algoritmo:



La complessità di questo algoritmo è:

$$T_n \in O(n + m) \quad (\text{graficamente è il caso in cui ci spostiamo a scalini}).$$

---

## Algoritmi Greedy

Sono algoritmi dove la soluzione ottima si ottiene mediante una certa sequenza di scelte. In ogni punto dell'algoritmo, viene scelta la strada che in quel momento **sembra la migliore**.

La scelta locale dovrebbe essere in accordo con la scelta globale.

Gli algoritmi greedy seguono un metodo *top-down*, e fornisce un'ottima euristica.

# Codici di Compressione

Abbiamo:

- **Alfabeto:** insieme di caratteri
- **Codice Binario:** assegna ad ogni carattere una stringa binaria
- **Codifica del testo:** sostituisce ad ogni carattere del testo il corrispondente codice binario
- **Decodifica:** restituisce il testo originario.

Il modo in cui andiamo a codificare ogni carattere può essere variabile ('a' = 1 bit, 'b' = 4 bit ...)

	a	b	c	d	e	f
frequenza	45	13	12	16	9	5
Codice a lunghezza fissa	000	001	010	011	100	101
Codice a lunghezza variabile	0	101	100	111	1101	1100

La scelta della frequenza variabile si basa sul concetto di diminuire il numero di bit utilizzati per gli elementi più frequenti.

Data la stringa "abc", la codifichiamo:

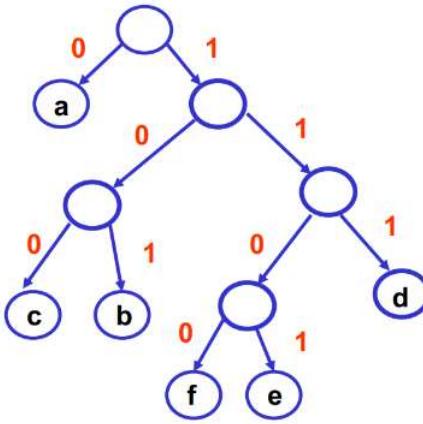
- Lunghezza fissa: "000 001 010" (9 bit)
- Lunghezza variabile: "0 101 100" (7 bit)

La decodifica tuttavia può essere più complicata con la lunghezza variabile, e necessita che le codifiche siano fatte con *codici prefissi*.

Ogni codice **non può** quindi essere il prefisso di un altro codice.

La potenza sta nel fatto che i *codici prefissi* possono essere rappresentati tramite **alberi bianri**. La rappresentazione ottima si ottiene quando si ha un **Albero Pienamente Binario** (ogni nodo ha 2 figli).

L'albero avrà tante foglie quante sono i caratteri dell'alfabeto, e la decodifica non fa altro che cercare possibili cammini dalla radice alla foglia per ogni carattere conosciuto.



## Codici Di Huffman

Il problema di partenza è il seguente:

*"Dato un alfabeto e la frequenza dei suoi caratteri, costruire un codice ottimo (che minimizza la lunghezza in bit delle codifiche)"*

L'algoritmo gestisce una foresta di alberi in questo modo:

- All'inizio ogni albero ha un solo nodo, che contiene la frequenza di un carattere dell'alfabeto.
- Poiché ho necessità di estrarre il minimo ad ogni passo, è conveniente utilizzare la struttura **minheap**.
- Ad ogni passo quindi:
  - Prendo i due alberi con frequenza minore e li fono:
    - La nuova radice ha come valore la somma delle radici
    - Le vecchie radici diventano figli della nuova radice
- Inserisco il nuovo albero nel *minheap* che avrà ora dimensione  $n - 1$

Il ciclo ha  $n$  iterazioni dove ogni iterazione ha costo  $O(\log n)$   $\Rightarrow O(n \log n)$

```
In [ ]: struct NodeH{
    char symbol;
    int freq;
    NodeH* sx;
    NodeH* dx;
}

Node* huffman(Heap H, int n){
    for(int i = 0; i < n - 1; ++i){
        Node* tmp = new NodeH();
        t->sx = H.extract();
        t->dx = H.extract();
        t->freq = t->sx->freq + t->dx->freq;
        H.insert(t);
    }

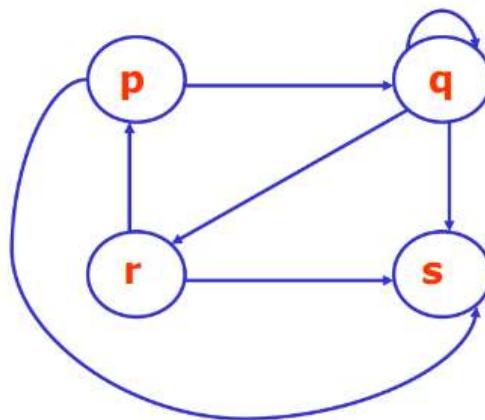
    return H.extract();
}
```

# Grafi

## Grafi Orientati

Un grafo **orientato** è così definito:

- $G = (N, A)$
- $N$ : Insieme di Nodi
- $A \subseteq N \times N$ : Insieme di Archi (coppie ordinate di nodi)



Se  $(p, q) \in A$ , diciamo che  $p$  è predecessore di  $q$  e  $q$  è successore di  $p$ .

Un grafo orientato possiede:

- $n = |N|$  Numero di Nodi
- $m = |A|$  Numero di Archi

Un grafo con  $n$  nodi ha al massimo  $m^2$  archi.

Altre proprietà di un grafo ordinato sono:

- **Cammini:** un cammino è una sequenza di nodi collegati da archi. La sua lunghezza è il numero degli archi
  - $(n_1, \dots, n_k), k \geq 1 \quad | \quad \exists (n_i, n_{i+1}) \in A, \quad 1 \leq i < k$
- **Ciclo:** cammino che inizia e finisce sullo stesso nodo
- **Grafo Aciclico:** un grafo che non contiene cicli.

Per implementare un grafo si possono sfruttare due idee:

## Liste di adiacenza

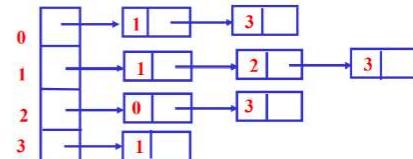
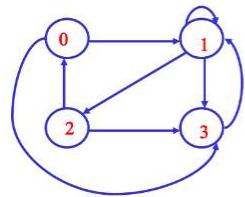
La struttura è simile a quelli degli alberi.

Viene definito un array con dimensione uguale al numero dei nodi.

Ogni elemento dell'array rappresenta un nodo con i suoi successori.

```
In [ ]: struct Node{
    InfoType label;
    Node* next;
}

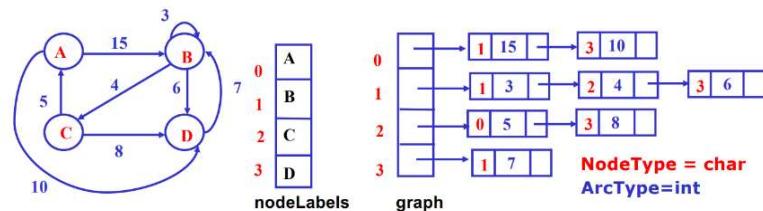
Node *graph[N]
```



Oltre a rappresentare informazioni relative ai nodi, posso andare ad etichettare gli archi:

```
In [ ]: struct Node{
    int NodeLabel;
    ArchType arcLabel;
    Node* next;
}

Node* graph[N]
NodeType nodeLabels[N];
```



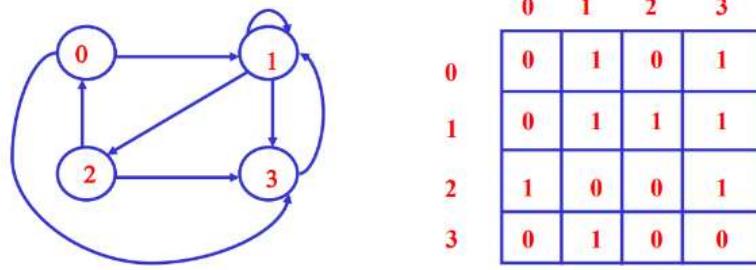
## Matrici di Adiacenza

Consideriamo una matrice di grandezza  $n \times n$ .

L'elemento della matrice di indici  $(i, j)$  è **false** se esiste un arco dal nodo  $i$  al nodo  $j$  e **false** altrimenti

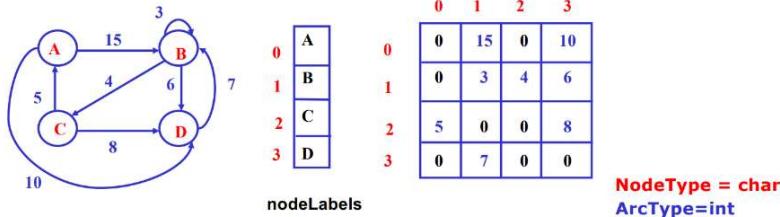
```
In [ ]: struct Node{
    InfoType label;
    bool connected;
};

Node graph[N][N];
```



Anche in questo caso posso etichettare gli archi:

```
In [ ]: ArcType graph[N][N];
NodeType nodeLabels[N];
```



## Visite sui grafi orientati

Sui grafi si implementano le visite, esattamente come negli alberi, quella che vediamo noi è la seguente:

### Visita in Profondità

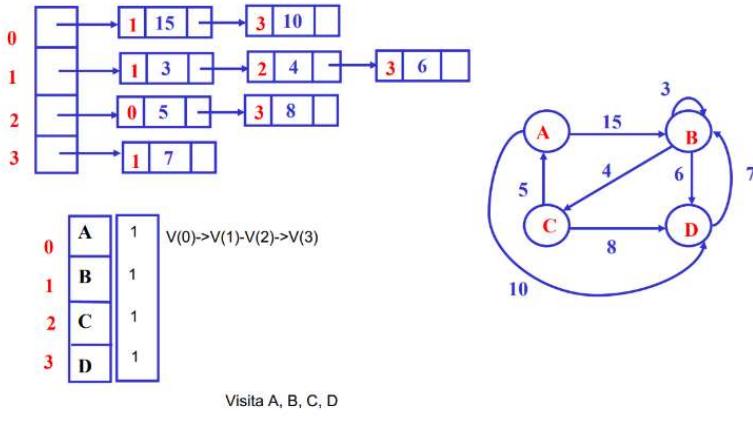
Una visita in profondità è una visita che percorre tutto il grafo.

Partendo da un nodo, si verifica se lo abbiamo già visitato o meno, successivamente si procede ricorsivamente con tutti gli archi uscenti.

```
In [ ]: void NodeVisit(Node* n){
    if(n == nullptr)
        return;
    <esamino il nodo>
    <marco il nodo>
    <per ogni successore non marcato chiamo NodeVisit()>;
}

void DepthVisit(Node* graph){
    for(int i = 0; i < n; ++i){
        NodeVisit(graph[i]);
    }
}
```

La complessità di questa visita è  $O(n + m)$ .



## Classe per i grafi orientati

```
In [ ]: const int N;
class Graph{
    struct Node{
        int nodeNumber;
        Node* Next;
    };
    Node* graph[N];
    NodeType nodeLabels[N];
    bool mark[N];

    void NodeVisit(int i){
        mark[i] = true;

        <esamina NodeLabels>

        Node* g;
        int j;
        for(Node* g = graph[i]; g != nullptr; g = g->next){
            j = g->nodeNumber;
            if(!mark[j])
                NodeVisit(j);
        }
    }
public:
    void depthVisit(){
        for(int i = 0; i < N; ++i){
            mark[i] = 0;
        }
        for(int i = 0; i < N; ++i){
            if(!mark[i])
                NodeVisit(i);
        }
    }
}
```

## Algoritmi sui grafi orientati

Sui grafi è possibile implementare diversi algoritmi:

## Ricerca Arco

La ricerca di un arco si implementa così:

```
In [ ]: // Implementazione di Liste di Adiacenza

bool searchArch(Node* &graph, int i, int j){
    for(Node* q = graph[i], q != nullptr; q = q->next){
        if(q->label == j)
            return true;
    }

    return false;
}

// Implementazione Matrice di Adiacenza

bool searchArch(Node* &graph, int i, int j){
    return (graph[i][j] != 0);
}
```

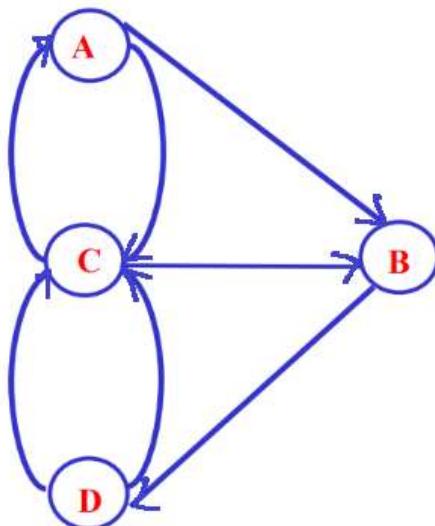
## Multi-Grafi Orientati

Una variante dei grafi ordinati sono i *multi-grafi orientati*.

Un *multi-grafo* è definito così:  $(N, A)$

- $N$ : Insieme di Nodi
- $A$ : **multi-insieme di coppie** di nodi.

In un multi-grafo per due nodi  $n_i, n_j$  possono esistere infiniti cammini separati.

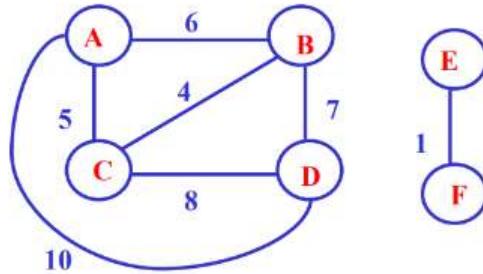


Ci sono 2 archi che collegano i nodi C e D

## Grafi Non Orientati

Un grafo **orientato** è così definito:

- $G = (N, A)$
- $N$ : Insieme di Nodi
- $A \subseteq N \times N$ : Insieme di Archi



Se  $(p, q) \in A$  e  $p \neq q$ , diciamo che  $p$  e  $q$  sono adiacenti.

Se un grafo ha  $n$  nodi, avrà al massimo  $\frac{n(n-1)}{2}$  archi.

In un grafo:

- **Cammino**: in un grafo non orientato è una sequenza di nodi  $(n_1, \dots, n_k)$ ,  $k \geq 1$ , tali che  $(n_i, n_{i+1})$  sono adiacenti per  $1 \leq i < k$
- **Ciclo**: un ciclo è un cammino che inizia e termina nello stesso nodo e non ha ripetizioni, eccetto l'ultimo nodo.
- **Grafo connesso**: un grafo non orientato è connesso se esiste un cammino fra due nodi qualsiasi del grafo.

L'implementazione è la medesima dei grafi orientati, prestando attenzione al fatto che se esiste l'arco  $(p, q)$  deve esistere anche quello  $(q, p)$ .

---

## Albero di Copertura

Definiamo innanzitutto la *componente connessa*: come un sottografo connesso (in cui tutti i nodi possono raggiungersi tra loro).

Si dice *componente connessa massimale* quando nessun nodo è connesso ad un'altra componente connessa (consideriamo tutti gli archi)

Un **Albero di copertura** è quindi definito come l'insieme delle **componenti massimali acicliche**.

Dati diversi alberi di copertura, posso scegliere il **Minimo Albero di Copertura**, ovvero quello che ha la somma dei pesi minima.

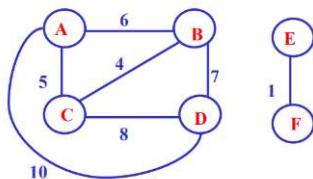
## Algoritmo di Kruskal

Per trovare il minimo albero di copertura possiamo utilizzare l'algoritmo di Kruskal, che è un algoritmo *greedy*.

Funziona così:

```
In [ ]: void Kruskal(Node*& graph){
    <ordino gli archi in ordine crescente>

    foreach a in archiCrescente do
        if(<a connette due componenti non connesse>){
            <scelgo a>
            <unifico le componenti>
        }
}
```



(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)  
{A} {B} {C} {D} {E} {F}

(C) (D) (F)

(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)  
{A} {B} {C} {D} {E, F}

(A) (B) (E)  
(C) (D) (F)

(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A} {B, C} {D} {E, F}

(A) (B) (E)  
(C) (D) (F)

(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A, B, C} {D} {E, F}

(A) (B) (E)  
(C) (D) (F)

(E,F) (B,C) (A,C) (A,B) (D,B) (C,D) (A,D)

{A, B, C, D} {E, F}

Lunghezza: 17

# Algoritmo di Dijkstra

L'algoritmo di Dijkstra è un algoritmo che risolve il problema del *cammino minimo tra due nodi di un grafo*.

L'algoritmo ha due richieste:

- Grafi orientati
- Pesi positivi

L'algoritmo, dato un grafo e un nodo di partenza, trova **tutti i cammini minimi** da quel nodo a tutti gli altri nodi.

Si basa su un implementazione *greedy*:

- Il cammino tra il nodo di partenza e se stesso viene impostato a 0
- I cammini minimi con tutti gli altri nodi vengono inizialmente posti a  $\infty$  per poi essere successivamente aggiornati con stime via via più precise tramite un ciclo
- Ad ogni iterazione del ciclo viene "sistemato" un nuovo nodo, ovvero stabilizzo il cammino minimo per raggiungerlo aggiornando gli altri cammini
- Alla fine tutti i nodi vengono sistematati.

L'algoritmo lavora su:

- 2 vettori con  $n$  elementi:
  - $dist[A]$ : distanza dal nodo di partenza in quel momento
  - $pred[A]$ : associa ad ogni nodo il predecessore

Per ogni nodo  $A$ ,  $dist[A]$  contiene la lunghezza dal punto di partenza, mentre  $pred[A]$  il predecessore in quel cammino.

I nodi sono divisi in 2 gruppi:

- Nodi Sistematici:
  - Hanno valore  $dist[A]$  e  $pred[A]$  definitivi:
    - $dist[A]$  contiene il minimo cammino per raggiungere il nodo  $A$
    - $pred[A]$  contiene il nodo precedente sul cammino minimo
- Nodi da Sistemare ( $Q$ ):
  - Hanno valori in  $dist[A]$  e in  $pred[A]$  non ancora certi e da aggiustare

Inizialmente nessun nodo è sistemato, quindi  $Q$  contiene tutti i nodi, compreso quello di partenza  $p_0$ .

Eseguo quindi un ciclo, dove ad ogni passo:

- Considero sistemato tra i nodi di  $Q$  il nodo con  $dist(A)$  minore e lo tolgo.
- aggiorno  $pred$  e  $dist$  per gli immediati successori.

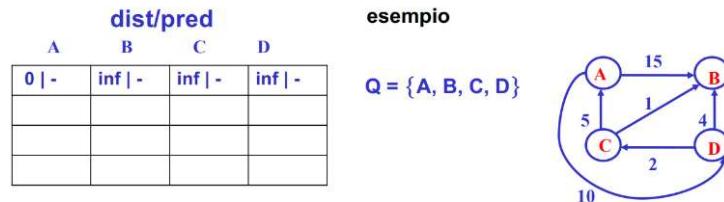
Termino quando  $Q$  contiene un solo nodo.

```
In [ ]: Q = N;

<foreach Node p in Q where != p0:>           // O(n)
    dist(p) = infinito;
    dist(q) = vuoto;

dist(p0) = 0;

while(Q contiene più di un nodo){
    <estraggo da q il nodo con minima dist(p)>
    foreach q successore di p in Q{
        lenpq = lunghezzaArco(p, q);
        if(dist(p) + lenpq < dist(q)){
            dist(q) = dist(p) + lqp;
            pred(q) = p;
            <re-inserisco in Q il nodo q modificato>
        }
    }
}
```



estraggo A:  $\text{dist}(A)=0$

$\text{dist}(A)+|(A,B)| < \text{dist}(B)$   
 $0 + 15 < \text{inf}.$

$\text{dist}(A)+|(A,D)| < \text{dist}(D)$   
 $0 + 10 < \text{inf}.$

A	B	C	D
0   -	15   A	inf   -	10   A

$$Q = \{B, C, D\}$$

estraggo D:  $\text{dist}(D)=10$

$\text{dist}(D)+|(D,B)| < \text{dist}(B)$   
 $10 + 4 < 15$

$\text{dist}(D)+|(D,C)| < \text{dist}(C)$   
 $10 + 2 < \text{inf}.$

A	B	C	D
0   -	14   D	12   D	10   A

$$Q = \{B, C\}$$

estraggo C:  $\text{dist}(C)=12$

$\text{dist}(C)+|(C,B)| < \text{dist}(B)$   
 $12 + 1 < 14$

A	B	C	D
0   -	13   C	12   D	10   A

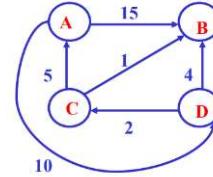
$$Q = \{B\}$$

da A a B:  $A \rightarrow D \rightarrow C \rightarrow B$     lung=13

da A a C:  $A \rightarrow D \rightarrow C$     lung=12

da A a D:  $A \rightarrow D$     lung=10

Nodo scelto	Q	A	B	C	D
	A, B, C, D	0   -	i   -	i   -	i   -
A	B, C, D	0   -	15/A	i   -	10/A
D	B, C	0   -	14/D	12/D	10/A
C	B	0/-	13/C	12/D	10/A



dist/pred

Poiché estraggo sempre il nodo con  $\text{dist}[A]$  minima, è conveniente memorizzare l'insieme Q in un **minHeap**. Perciò:

- Estrazione nodo minima distanza  $\Rightarrow O(\log n)$
- Re-inserimento nodo modificato:  $\Rightarrow O(\log n)$

Nel ciclo while eseguo  $n$  iterazioni, dove ogni iterazione ha complessità:

$O(\log n + \frac{m}{n} \log n)$  (dove ipotizziamo che da ogni nodo esca un numero uguale di archi)

$$\Rightarrow \frac{m}{n}).$$

La complessità totale del nostro algoritmo sarà quindi:

$$T(n, m) \in O(n \log n + m \log n)$$

---

## Applicazioni dei Grafi

I grafi sono applicati in tantissimi problemi.

### Graph Coloring

Dato un numero  $K$  di colori, associare ad ogni nodo di un grafo un colore tale che, per nessun nodo adiacente abbia lo stesso colore.

Il *graph coloring* è un problema che ha come soluzione migliore una soluzione **esponenziale**, inoltre spesso non è nemmeno detto che sia possibile trovare una soluzione.

Un applicazione del *graph-coloring* si trova nel **sudoku** (i colori corrispondono in questo caso ai numeri da 1 a 9)

### Page Rank

L'algoritmo di *page-rank* è uno degli algoritmi utilizzati da *Google* per restituire una serie di risultati a partire da una stringa.

L'idea si basa su restituire all'utente medio le pagine alle quali potrebbe essere più interessato.

Cerca quindi di stimare il comportamento di un "random surfer" che naviga il web basandosi sulle connessioni (*link*) fra le pagine.

Per fare questo, l'algoritmo di page rank rappresenta:

- Pagine web → grafo (*webgraph*)
- Link → arco

Google rilasciò un *webgraph* nel 2002 contenente:

- 5.105.309 archi
- 675.713 nodo

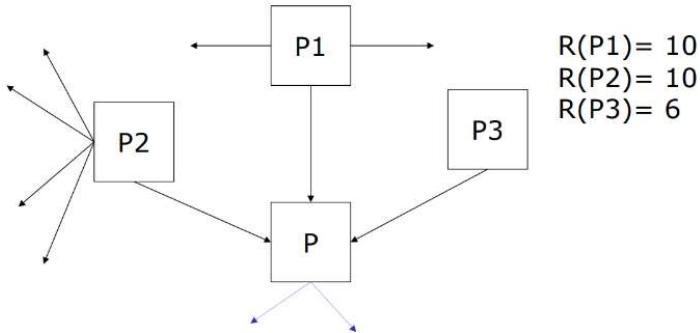
Il grafo che contiene le pagine di *Wikipedia*:

- 165.206.104 archi
- 6.625.370 nodi

L'algoritmo calcola il **rango** di una pagina web  $P : R(P)$ , dove il rank dipende dalle altre pagine che hanno un arco uscente verso  $P$ , secondo la formula:

$$R(P) = \sum_{Q \rightarrow P} \frac{R(Q)}{|Q|}$$

$|Q|$  numero di link uscenti da  $Q \rightarrow P$



$$R(P) = 10/3 + 10/5 + 6/1 = 11,3$$

Il calcolo viene fatto calcolando il rango dei nodi in modo iterativo, utilizzando la matrice di adiacenza e partendo da un valore del rango uguale per tutti i nodi. Inizialmente una stima affidabile richiedeva un tempo di calcolo di qualche giorno.

La formula necessita comunque alcuni aggiustamenti per garantire la convergenza dell'algoritmo:

- **Nodi pozzo:** nodi senza archi uscenti
- **Cicli:** rank che dipende da se stesso

## Graph Database

Il concetto di grafo può essere applicato anche per la creazione di Database.

I graph database adottano vertici e archi per memorizzare relazioni esplicite tra entità.

Mentre nei database relazionali le connessioni sono rappresentati tramite valori di attributo comune (**chiave**). I collegamenti/connettori possono essere trovate tramite l'utilizzo di **JOIN**, che tendono a diventare computazionalmente molto costose qualora si stia gestendo una grande quantità di dati.

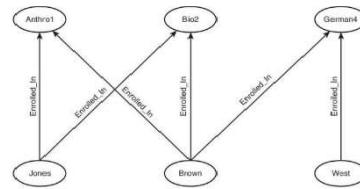
La rappresentazione tramite *graph database* invece permette un'interrogazione più rapida.

Students	
123	Jones
278	Brown
789	West
.	.
.	.
.	.

Enrollment	
123	Astro1
123	Bio2
278	Bio2
278	Astro1
789	German4
.	.
.	.
.	.

Courses	
Astro1	Intro. to Anthropology
Bio2	Evolutionary Biology
German4	German Literature
.	.
.	.
.	.

Database Relazionale

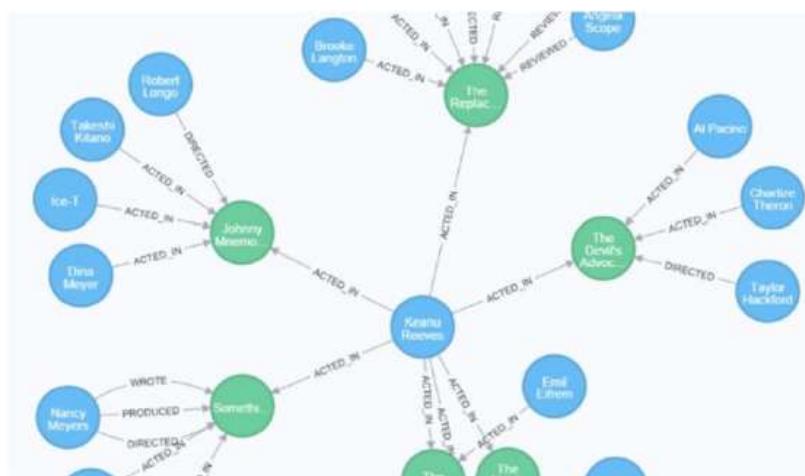


Graph Database

Avendo un database che contiene film e attori, se volessi cercare tutti gli attori che hanno recitato con Keanu Reeves, adottando un modello relazionale:

```
In [ ]: SELECT p2.personName, m1.movieName
FROM people p1
JOIN actors a1 ON (p1.personId = a1.personId)
JOIN movies m1 ON (a1.movieId = m1.movieId)
JOIN actors a2 ON (a2.movieId = m1.movieId)
JOIN people p2 ON (p2.personId = a2.personId)
WHERE p1.personName = "Keanu Reeves"
```

In un *graph database* è sufficiente, dato il nodo che identifica "Keanu Reeves", visitare i nodi dei film adiacenti e successivamente visitare tutti i nodi entranti che sono attori.



# Cenni alla NP-completezza

L'analisi della complessità computazionale di un algoritmo che risolve un problema può essere ampliata al problema stesso.

Il nostro focus si sposta sulla domanda: "Dato un problema, qual'è il miglior modo oggettivo con il quale posso risolverlo?"

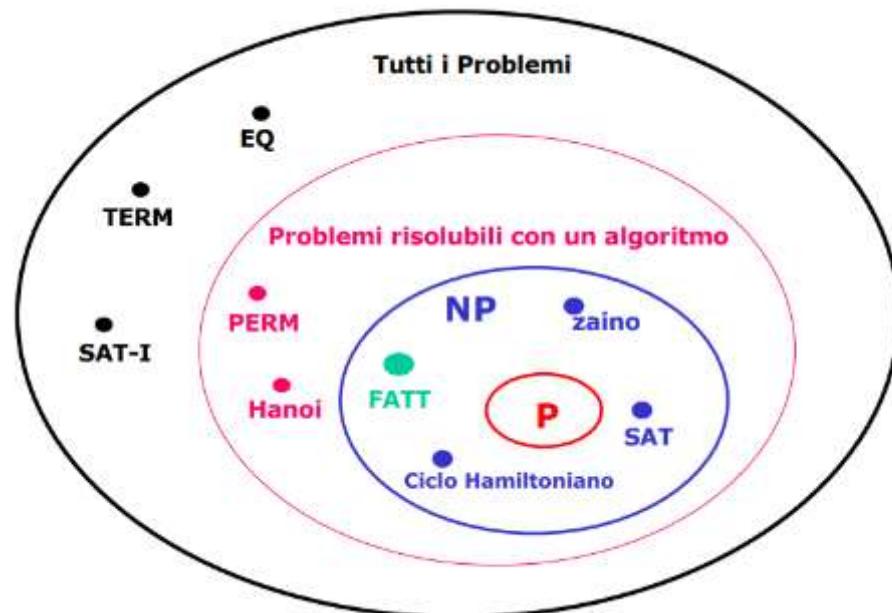
Immaginando un insieme contenente tutti i problemi, si separano innanzitutto tra quelli che possono essere risolvibili con algoritmi e quelli che non possono esserlo.

Tra i problemi non risolvibili con algoritmi troviamo:

- **TERM**: decidere la terminazione di un programma su un input;
- **SAT-I**: soddisfacibilità di una formula nella logica del I° ordine (con operatori  $\forall$  e  $\exists$ );
- **EQ**: Decidere l'equivalenza di due programmi.

Tra quelli invece risolvibili con algoritmi li classifichiamo nel seguente modo:

- **Problemi Esponenziali**: problemi che hanno complessità esponenziale sia se deterministici che non, sia la verifica (torre di Hanoi, permutazioni)
- **Problemi NP**: problemi deterministici che hanno algoritmo di verifica di soluzione polinomiale
- **Problemi "Difficili" (NP-completi)**: problemi NP dei quali non siamo sicuri se abbiano o meno soluzione esponenziale o polinomiale
- **Problemi P**: problemi che hanno soluzione polinomiale



Alcuni esempi di problemi NP:

## Problema dello zaino

"Dato uno zaino con capacità massima e  $n$  oggetti, dove ogni oggetto ha un peso e un valore, ottimizzare il riempimento dello zaino, determinare il numero di oggetti di ogni tipo in modo tale che il peso sia minore o uguale alla capacità dello zaino massimizzando il valore totale."

## Problema del Commesso Viaggatore

"Dato un insieme di città collegate da strade, trovare il percorso di minore lunghezza che un commesso viaggiatore deve seguire per visitare tutte le città una e una sola volta per poi tornare alla città di partenza."

## Cammino e ciclo Hamiltoniano

Dato un multigrafo, definiamo:

- *Cammino Hamiltoniano*: un cammino che tocca tutti i nodi una e una sola volta
- *Ciclo Hamiltoniano*: un ciclo che tocca tutti i nodi una e una sola volta

Un grafo è Hamiltoniano se possiede un ciclo Hamiltoniano.

Il problema dei cicli Hamiltoniani è simile, ma profondamente diverso e più complesso, al problema dei *cicli Euleriani*:

- Dato un multigrafo, trovare un ciclo che passa per tutti gli archi una e una sola volta

Esiste per i cicli euleriani un teorema: "*Un multigrafo non orientato contiene un ciclo Euleriano se e solo se gli archi che partono da ogni nodo sono in numero pari*"

Perciò esiste una soluzione polinomiale per questi ultimi

## Soddisfattibilità di una formula logica (SAT)

*SAT*: soddisfattibilità di una formula nella logica dei predicati

Data una formula  $F$  con  $n$  variabili, trovare, se esiste, una combinazione di valori booleani che, SE assegnati alle variabili di  $F$ , la rendono vera

$$\begin{aligned} F = (x \wedge !x) \vee (y \wedge !y), \quad n = 2, & \quad \text{non risolvibile} \\ F = (x \wedge !y) \vee (y \wedge !x), \quad n = 2, & \\ \text{risolvibile: } x = 0, y = 1 & \end{aligned}$$

---

Tutti questi problemi hanno come soluzione sicura quella di provare tutte le possibili combinazioni, che sono  $2^n$ .

# Teoria della NP-completezza

La teoria della NP-completezza classifica un insieme di problemi difficili.

Si applica a problemi decisionali, che hanno complessità minore o uguale a quella del rispettivo problema non decisionale.

Se il problema decisionale è difficile, a maggior ragione lo sarà il problema non decisionale corrispondente.

Alcune traduzioni sono:

- **Compresso Viaggiatore:** dato un intero  $k$ , esiste un ciclo senza ripetizione di nodi minore di  $k$
  - **Zaino:** dato un valore  $v$ , esiste un riempimento dello zaino con valore maggiore o uguale a  $v$
  - **Cicli Hamiltoniani:** Dato un grafo, esiste un ciclo Hamiltoniano
  - **Cicli Euleriani:** Dato un grafo, esiste un ciclo Euleriano
  - **Formula Logica:** Data una formula, esiste un assegnamento alle variabili che rende vera la formula
- 

## Algoritmi Non Deterministici

Un algoritmo non deterministico comprende il nuovo comando **choice( $I$ )**, dove  $I$  è un insieme.

**choice( $I$ )** sceglie un elemento causuale dall'insieme  $I$ .

Gli algoritmi non deterministici permettono di determinare risultati solo grazie alla fortuna.

Per ogni algoritmo non deterministico è possibile trovarne uno deterministico che lo simula, esplorando tutto lo spazio delle soluzioni (solitamente in numero esponenziale), fino a trovare un successo.

---

## Insieme $P$ e $NP$

Definiamo quindi i due insiemi:

- $P$ : insieme di tutti i problemi decisionali risolubili in tempo **Polinomiale** con un algoritmo Deterministico
  - Ricerca
  - Ordinamento

- Ciclo Euleriano
- $NP$ : insieme di tutti i problemi decisionali risolubili in tempo **Polinomiale** con un algoritmo Nondeterministico
  - Ricerca
  - Ordinamento
  - Fattorizzazione
  - Soddisfacibilità
  - Zaino
  - Commesso viaggiatore
  - Cicli Hamiltoniani
  - Formula Logica

Inoltre i problemi  $NP$ , **data una soluzione**, permettono di verificare in tempo *polinomiale* la sua correttezza (certificato)

Per dimostrare che un problema  $R \in NP$ , si dimostra che la verifica di una soluzione di  $R$  è data in tempo polinomiale.

- Ciclo Hamiltoniano  $\rightarrow O(n)$
- Formula Logica  $\rightarrow O(n)$

Abbiamo quindi due classi di problemi:

- $P$  = problemi decisionali **facili da risolvere**
- $NP$  = problemi decisionali **facili da verificare**

Da qui nasce uno dei 7 problemi del millennio, la cui soluzione (in un senso o nell'altro) ha un premio di 1 milione di dollari:

"Sicuramente sappiamo che  $P \subseteq NP$ , ma  $P = NP?$ "

La soluzione di questo problema collassa su un sottoinsieme particolare di  $NP$ , che chiamiamo *NPcompleto*.

I problemi *NP-completi* derivano dalla *riducibilità*, che è un metodo per convertire l'istanza di un problema  $P1$  in un'istanza di un problema  $P2$  e utilizzare la soluzione di quest'ultimo per ottenere la soluzione di  $P1$ .

La *riducibilità* si dice **polinomiale** se un problema  $P1$  si riduce in tempo polinomiale a un problema  $P2$ , ciò significa che ogni soluzione di  $P1$  può ottenersi deterministicamente in tempo polinomiale da una soluzione di  $P2$

$$P1 \leq P2 \quad \text{oppure} \quad P1 \prec P2$$

La conseguenza della riducibilità è:

$$\begin{cases} P1 \leq P2 \\ P2 \text{ è risolubile in tempo polinomiale} \end{cases} \Rightarrow P1 \text{ è risolubile in tempo polinomiale}$$

Da questa conseguenza definiamo l'insieme **NP-completo**:

$$\begin{cases} R \in NP \\ \forall Q \in NP \rightarrow Q \prec R \end{cases}$$

Alcuni problemi **NP-completi** sono:

- **SAT**
- **Compresso Viaggatore**
- **Zaino**
- **Ciclo Hamiltoniano**

In particolare il *Teorema di Cook* ci dice che:

$$\forall R \in NP : R \prec SAT$$

Perciò **SAT** è il più difficile dei problemi ovvero, ridefinisco **NP-completo**:

$$\begin{cases} R \in NP \\ R \prec SAT \end{cases}$$

I problemi **NP-completi** sono difficili tanto quanto SAT, perciò sono i più difficili tra i problemi NP, e la soluzione di uno di loro in tempo polinomiale dimostrerebbe che  $P = NP$

Per dimostrare che un problema  $R$  è NP-completo:

- Dimostrare che  $R \in NP$ : trovare un algoritmo di verifica polinomiale
  - Dimostrare che esiste un problema NP-completo che si riduce a R
- 

## Fattorizzazione in fattori primi

Il problema della fattorizzazione in numeri primi, come tutti i problemi di teoria dei numeri, dipende dal numero di cifre.

La verifica di una soluzione richiede semplicemente di moltiplicare i fattori dati:  
 $O(n^{\log_2 3})$  oppure  $O(n^2)$

Trovare due fattori particolari di un numero invece richiede decine di anni di tempo di calcolo  $O(2^n)$ . E la crittografia moderna si basa proprio su questa realtà.

La fattorizzazione è quindi sicuramente un problema NP, quasi sicuramente non NP-completo (*NP-intermediate*).

E' praticamente impossibile scomporre un numero di 200 o più cifre con gli algoritmi attuali.

La dimostrazione che  $P = NP$  avrebbe come conseguenza la fattorizzazione in tempo polinomiale di qualsiasi numero e quindi il collasso della moderna crittografia a chiave pubblica.

Va sottolineato che esiste un algoritmo polinomiale che risolve **FATT**, chiamato *algoritmo di Shor*(1994) che è però basato su **Quantum Computing**.

---

# **Problemi Difficili**

Per affrontare problemi difficili si utilizzano alcune tecniche quali:

- Algoritmi di approssimazione
- Algoritmi probabilistici
- Reti Neurali
- Quantum Computing

# Programmazione ad Oggetti

Dalla programmazione ad oggetti possiamo trarre i seguenti vantaggi:

- **Incapsulamento**: permettere l'accesso solo a determinati aspetti della classe, nascondendo altri aspetti o le implementazioni stesse (*information hiding*);
- **Decomposizione**: una classe può contenere altri parametri o altre classi, permettendo una creazione modulare;
- **Riuso**: capacità di riutilizzare funzioni e classi in nuove funzioni e classi;
- **Manutenzione**;
- **Affidabilità**.

## Funzioni e Classi Modello - Templates

I *template* sono un meccanismo che permettono al programmatore di scrivere classi e funzioni indipendentemente dal tipo di dato che verrà implementato, permettendo di concentrarsi solamente sulla logica.

I *template* rientrano nel concetto della *meta-programmazione*, ovvero la possibilità di applicare lo stesso codice a tipi diversi, *parametrizzando i tipi utilizzati*.

Questo ci permette:

- L'indipendenza degli algoritmi dai dati a cui si applicano (`std::algorithm::sort`)
- L'indipendenza dei contenitori dal tipo del loro contenuto (`<vector>`)

```
In [ ]: // Funzioni Modello

int max(int x, int y){
    return (x > y)? x : y;
}

double max(double x, double y){
    return (x > y)? x : y;
}
```

Le due funzioni nell'esempio hanno **la stessa definizione** con *tipi diversi*.

Il costrutto del *template* ovvia a questo problema:

```
In [ ]: template<class type>

type max(type x, type y){
    return (x > y)? x : y;
}

int main(){
    int b;
    double c;
    int array[2] = {3, 4};
```

```

b = max(3, b);
// type = int -> max<int>(int, int)
// OK

c = max(3.6, c);
// type = double -> max<double>(double, double)
// type

c = max(3, c);
// ERRORE
// non si deduce se type = int oppure type = double
}

// a tempo di compilazione viene creato:

max<int>(int x, int y){
    return (x > y)? x : y;
}

max<double>(double x, double y){
    return (x > y)? x : y;
}

```

A tempo di compilazione, il *template* viene convertito nel codice opportuno per gestire il tipo di variabili.

```

In [ ]: template<class type>

void primoP(type *x){
    type y = x[0];
    cout << y << endl;
}

void primo(type x){
    cout << x[0] << endl;
}

void primoParticolare(type x){
    type y = x[0];
    cout << y << endl;
}

int main(){
    int array1[2] = {3, 4};
    int array2[2] = {3.6, 6.6};

    primoP(array1);
    /* OK, type = int

    primo(array2);
    /* OK, type = double*

    primoParticolare(array1);
    /*! ERRORE, type = int*
    /*! int *y = x[0]; -> non si può convertire un int in int*
}

```

```
In [ ]: template <class type1, class type2>

type1 max(type1 x, type2 y){
    return (x > y)? x : y;
}

int main(){
    int b;
    double c;

    b = max(3, b);
    /* OK
    /* type1 = int, type2 = int

    c = max(3, c);
    /* OK
    /* type1 = int, type2 = double
}
```

```
In [ ]: template<class type1, class type2, class type3>

type1 max(type2 x, type3 y){
    return (x>y)? x : y;
}

int main(){
    int b;
    double c;

    b = max(3, c);
    /* type2 = int
    /* type3 = double
    //! type1 = int/double -> ERRORE

}
```

Oltre a utilizzare i parametri in maniera implicita, possiamo anche utilizzarli in maniera esplicita:

```
In [ ]: template<class type>

type max(type x, type y){
    return (x > y)? x : y;
}

int main(){
    double d;
    cout << max<int>(4, 5.5) << endl;
    //? chiama max<int>(int, int)

    cout << max<double>(3, 5.5) << endl;
    //? chiama max<double>(double, double)

    d = max<int>(3, 6.5);
    //? chiama max<int>(int, int) e poi converte il risultato in double
```

```
    cout << d << endl;
}
```

```
5
5.5
6
```

```
In [ ]: template<class type1, class type2, class type3>

type1 fun(type2 x, type3 y){
    // ...
}

int main(){
    fun<int>(9.8, 10);
    /* type1 = int
     * type2 = double
     * type3 = int

    fun<int, double>(9, 'c');
    /* type1 = int
     * type2 = double
     * type3 = char

    fun<int, double, double>(10, 9.9);
    /* type1 = int
     * type2 = double
     * type3 = double

    fun(9, 8);
    /*! type1 = ?
     * type2 = int
     * type3 = int
}
```

Inoltre posso specificare mettere dei parametri non-tipo nel *template*.

Questo permette di creare istanze diverse di una stessa funzione al variare del parametro non-tipo.

```
In [ ]: template<int n, double m>

void funzione(int x = n){
    int y = m;
    int array[n];
    // ...
}

int main(){
    funzione<1+2, 4>(8);
    /*
     * n = 3
     * m = 4
     * funzione<3, 4>(int)
     * ---
}
```

```

* funzione(int x = 3){
*   int y = 4;
*   int array[3];
* }
*/
funzione<2,2>(9);
/*
* n = 2
* m = 2
* funzione<2, 2>(int)
* ---
* funzione(int x = 3){
*   int y = 4;
*   int array[3];
* }
*/
}

```

```

In [ ]: template<int n, class type>

int gt(type x){
    return x > n;
}

int main(){
    cout << gt<50+6>(101) << endl;
/*
* n = 56
* type = int (implicita)
*/

    cout << gt<8, double>(7) << endl;
/*
* n = 8
* type = double (esplicita)
*/
}

}

```

101

8

Si può anche applicare a variabili statiche:

```

In [ ]: template<class type>

type maxT(type x, type y){
    static int a = 0;
    a++;
    cout << a << endl;
    return (x > y)? x : y;
}

```

```

int main(){
    cout << maxT<int>(101, 102) << endl;
    cout << maxT<int>(101, 203) << endl;
    cout << maxT<double>(10.2, 293.2) << endl;
}

```

```

1
202
2
203
1
293.2

```

Posso definire anche le classi tramite template.

In questo caso però i parametri sono **sempre esplicativi**.

Inoltre, l'implementazione non in-place di funzioni o membri di una classe che vanno a utilizzare il template, devono essere necessariamente locati nello stesso file, non è possibile la loro definizione in un file esterno.

In aggiunta, è necessario esplicitare ogni volta il tipo di template che stiamo utilizzando.

```

In [ ]: template<class type>

class stack{
    int size;
    type *p;
    int top;
public:
    stack(int n){
        size = n;
        p = new type[n];
        top = -1;
    };

    ~stack(){
        delete[] p;
    };

    bool empty(){
        return top == -1;
    };

    int full(){
        return (top == size - 1);
    };

    bool push(type s){
        if(this->full())
            return false;
        p[+top] = s;
        return true;
    };

    bool pop(type& s);
}

```

```

template<class type>
bool stack<type>::pop(type& s){
    if(this->empty())
        return false;
    s = p[top--];
    return true;
}

int main(){
    stack<int> s1(10), s2(20);
    stack<double> s3(200);
    stack<char> s4(23);

    s1.push(3);
    s3.push(3.32);
    s4.push('a');
}

```

In [ ]: template<class type, int size>

```

class stack{
    type *p;
    int top;
public:
    stack(){
        p = new type[size];
        top = -1;
    };

    ~stack(){
        delete[] p;
    };

    bool empty(){
        return top == -1;
    };

    int full(){
        return (top == size - 1);
    };

    bool push(type s);

    bool pop(type& s);
}

template<class type, int size>
bool stack<type, size>::push(type s){
    if(this->full())
        return false;

    p[+top] = s;
    return true;
}

```

```

template<class type, int size>
bool stack<type, size>::pop(type& s){
    if(this->empty())
        return false;
    s = p[top--];
    return true;
}

int main(){
    stack<int, 200> stk1;
    stack<int, 300> stk2;
    stack<int, 100>* ptr;

    ptr = &stk1;
    //! ERRORE, tipi diversi
    //! stack<int, 100> != stack<int 200>
}

```

## Derivazione - Inheritance

La derivazione o ereditarietà (*inheritance*) consente di trasmettere un insieme di caratteristiche comuni da una classe **base** (*padre*) ad una **derivata** (*figlio*), senza che ciò comporti una duplicazione del codice, offrendo allo stesso tempo l'opportunità di **adattare** o **estendere** il comportamento a casi d'uso speciali.

La classe **derivata/figlio**: eredita i membri *public* e *protected* della **base/padre**, mantenendo la stessa specifica.

La sezione *protected* di una classe si comporta:

- Come la *public* per le classi derivate;
- Come la *private* per il resto del codice.

I membri *private* della classe padre **continuano a non essere visibili per nessuno**.

La classe figlio oltre ad avere accesso ai membri del padre, può anche sviluppare delle funzioni o dei membri propri, ai quali il padre non può accedere.

Attraversando i livelli gerarchici dall'alto in basso significa spostarsi da un livello di astrazione *generico* ad altri più *specifici*.

```

In [ ]: class uno{
    int priv;
protected:
    int prot;
public:
    int x;
};

class due : public uno{
public:
    int y;
}

```

```

void f(){
    prot = 10;
    x = 5;
    y = 6;
}
}

int main(){
    uno* t = new uno;
    due* s = new due;
    s->f();
    cout << s->x << endl;

    s->priv = 10;
    //! ERRORE
    //! priv è membro privato di uno, non è ereditato

    cout << s->prot << endl;

    t->prot = 101;
    //! ERRORE
    //! prot è membro protected, perciò non è possibile accedervi
}

```

5

10

Il tipo di derivazione che stiamo effettuando è una **derivazione public**:

- Deriviamo la parte *protected* mantenendola tale
- Deriviamo la parte *public* mantenendola tale

Un altro tipo di derivazione possibile è la **derivazione protected**:

- Deriviamo la parte *protected* mantenendola tale
- Deriviamo la parte *public* rendendola *protected*

Un ultimo tipo di derivazione è la **derivazione private**:

- Deriviamo la parte *protected* rendendola *private*
- Deriviamo la parte *public* rendendola *private*

In [ ]:

```

class persona{
public:
    string nome;
    int eta;
};

class studente : public persona{
public:
    int esami;
    int matricola;
};

```

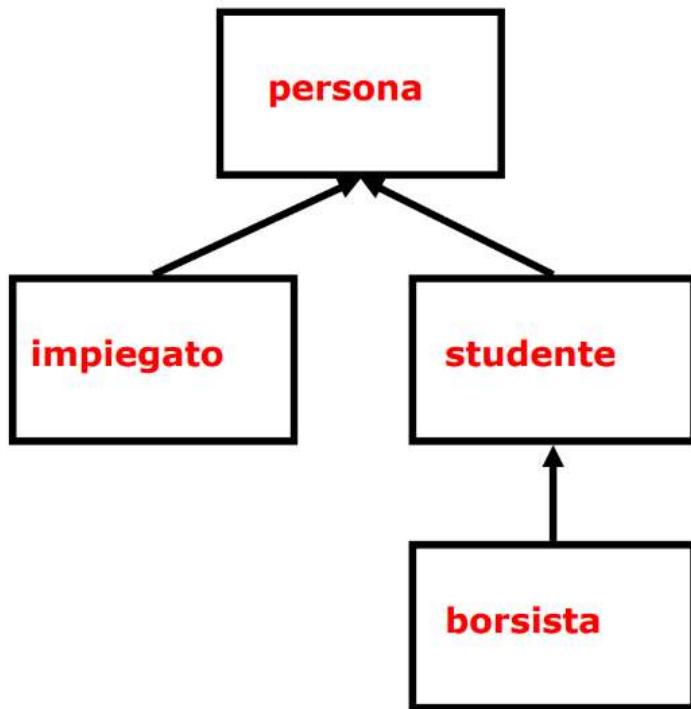


```
In [ ]: class borsista : public studente{
public:
    int borsa;
    int durata;
};

int main(){
    borsista b;
    borsista* borsistaPointer = &b;
    b.borsa = 500;
    borsistaPointer->esami = 33;
    b.eta = 22;
}
```



```
In [ ]: class impiegato: public persona{
public:
    int livello;
    int stipendio;
};
```



Posso assegnare *sottotipi* a *supertipi* (figli a padri), perdendo però così le specializzazioni.

La compatibilità si riflette anche sui puntatori. Infatti puntatori di classi padri potranno puntare classi figlie, seppur avendo solamente accesso ai membri presenti nella classe padre.

```

In [ ]: int main(){
    persona p;
    persona *personPointer;

    impiegato i;

    studente s;
    studente *studentPointer;

    borsista b;
    borsista *borsistaPointer;

    p = s;
    /* conversione implicita persona->studente ammessa
    personPointer = &p;

    studentPointer = &s;

    s = p;
    //! ERRORE
    //! Supertipo assegnato a sottotipo
    s = i;
    //! ERRORE
    //! Supertipo assegnato a sottotipo

    p = b;
    /* conversione implicita borsista->persona ammessa
  
```

```

s = b;
/* conversione implicita borsista->studente ammessa

personPointer = studentPointer;
/* conversione implicita ammessa, il puntatore a persona
/* può puntare ad una sua specializzazione, ma potrà accedere
/* solamente ai membri presenti anche in "persona"
}

```

Di conseguenza potrò accedere a:

- personPointer->nome, personPointer->eta
- studentPointer->nome, studentPointer->eta, studentPointer->esami, studentPointer->matricola

personPointer->esami è un errore.

```

In [ ]: class persona{
public:
    string nome;
    int eta;

    void chiSei(){
        cout << nome << "\t" << eta << endl;
    }
};

class studente : public persona{
public:
    int esami;
    int matricola;

    void quantiEsami(){
        cout << esami << endl;
    }
};

class borsista : public studente{
public:
    int borsa;
    int durata;
};

int main(){
    persona *p;
    studente *s;
    borsista *b;

    // ...

    p->chiSei();
    s->chiSei();
    b->chiSei();
}

```

```

s->quantiEsami();
b->quantiEsami();

p->quantiEsami();
//! ERRORE
}

```

La visibilità nelle classi ereditarie funziona come la visibilità in blocchi annidati.

Ciò significa che una classe figlio ed una classe padre possono condividere i nomi di una variabili/funzioni dando loro significati diversi.

Quando non è specificato diversamente, la classe ereditata utilizzerà la propria versione (se presente).

Per utilizzare la versione della classe padre, va specificato tramite operatore di visibilità:

```

In [ ]: class studente : public persona{
public:
    int matricola;
    int esami;
};

class persona : public studente{
public:
    int borsa;
    int durata;
    int esami;
}

int main(){
    studente *s = new studente;
    borsista *b = new borsista;

    b->esami = 4;
    //? borsista::esami, esami di borsista

    b->studente::esami = 5
    //? esami di studente

    cout << "Esami borsa: " << b->esami << endl;

    s = b;

    cout << "Esami studente: " << s->esami << endl;
}

```

*Esami borsa: 4*

*Esami studente: 5*

In questo caso gli esami studente risultano 5 poiché siamo andati a modificare il membro esami relativo all'eredità dovuta da Studente, che è stata copiata nell'assegnazione.

```
In [ ]: class persona{
public:
    string nome;
    int eta;

    void chiSei(){
        cout << nome << "\t" << eta << endl;
    };

    void f(int n){
        cout << "BIG PP" << endl;
    };
};

class studente : public persona{
public:
    int matricola;
    int esami;

    void chiSei(){
        cout << nome << "\t" << eta;
        cout << "\t" << matricola << "\t" << esami << endl;
    };

    void f(){
        cout << "smoll pp" << endl;
    }
};

int main(){
    studente s;
    s.nome = "anna";
    s.eta = 18;
    s.matricola = 634534;
    s.esami = 4;

    s.chiSei();

    s.persona::chiSei();

    s.f();

    s.f(6);
    //! ERRORE
    //! In studente non è presenta una funzione f(int)

    s.persona::f(6);
}
```

anna 18 634534 4  
 anna 18  
 smoll pp  
 BIG PP

Quando costruisco una nuova classe, viene costruita **prima** la parte *base*, e successivamente quella *derivata*.

Quando chiamo il costruttore di una classe derivata infatti, prima chiamerò i costruttori dei livelli più alti della gerarchia, per poi continuare nella chiamata dei costruttori procedendo a cascata fino ad arrivare a quello che vogliamo ottenere.

Nel caso di costruttori di default, questa azione avviene in automatico.

Nel caso in cui non ci sia un costruttore di *default*, va **necessariamente** specificato il caso in cui volessimo chiamare i costruttori *custom*.

```
In [ ]: class uno{
protected:
    int a;
public:
    uno(){
        int a = 5;
        cout << "uno default" << a << endl;
    };
    uno(int x){
        a = x;
        cout << "uno custom " << a << endl;
    };
};

class due : public uno{
    int b;
public:
    due(int x){
        b = x;
        cout << "due custom" << b << endl;
    };
    due(int x, int y) : uno(y){
        b = x;
        cout << "due custom" << b << endl;
    }
};

class tre: public due{
    int c;
public:
    tre(int x){
        c = x;
        cout << "tre custom" << c << endl;
    }
};

int main(){
    due obj2(8);
    due objN2(7, 9);

    tre obj3(10);
    //! ERRORE
    //! La classe 'due' non ha costruttore di default, e io non ho specificato q
}
```

```
uno default 5  
due custom 8  
uno custom 9  
due custom 7
```

La chiamata dei distruttori è inversa rispetto a quella dei costruttori, ovvero distruggo prima i campi *derivati* e poi i campi *base*, in una gerarchia dal basso verso l'alto.

```
In [ ]: class uno{  
public:  
    uno(){  
        cout << "creo 1" << endl;  
    };  
    ~uno(){  
        cout << "rompo 1" << endl;  
    };  
};  
  
class due : public uno{  
public:  
    due(){  
        cout << "creo 2" << endl;  
    };  
    ~due(){  
        cout << "rompo 2" << endl;  
    };  
};  
  
int main(){  
    uno* u = new uno;  
  
    cout << endl;  
    due* d = new due;  
  
    cout << endl;  
    delete u;  
  
    cout << endl;  
    delete d;  
}
```

*creo 1*

*creo 1*  
*creo 2*

*rompo 1*

*rompo 2*  
*rompo 1*

Le classi derivate ereditano anche i *membri statici*, continuando ad andare a modificare sempre lo stesso membro, come se la derivazione non ci fosse.

```
In [ ]: class uno{
public:
    static int quantiA;
    uno(){
        cout << "A = " << ++quantiA << endl;
    };
};

uno::quantiA = 0;

class due : public uno{
public:
    static int quantiB;
    due(){
        cout << "B = " << ++quantiB << endl;
    };
};

due::quantiB = 0;

int main(){
    uno u1;
    uno u2;

    cout << endl;
    due d1;

    cout << endl;
    due d2;
}
```

A = 1

A = 2

A = 3

B = 1

A = 4

B = 2

---

## Funzioni Virutali

Le *virtual functions* vengono utilizzate in caso di ridefinizioni di funzioni membro, venendo chiamate tramite puntatore.

La scelta della versione della funzione da chiamare è effettuata a tempo di compilazione, e riflette sempre l'implementazione del tipo del puntatore.

```
In [ ]: class studente{
    int esami;
    int matricola;
public:
    studente(int e, int m){
        esami = e;
        matricola = m;
    };

    int qualeMatricola(){
        return matricola;
    };

    void chiSei(){
        cout << "sono uno studente" << endl;
    };
};

class borsista : public studente{
    int borsa;
public:
    borsista(int e, int m, int b) : studente(e, m){
        borsa = b;
    };

    void chiSei(){
        cout << "sono un borsista" << endl;
    };
};

int main(){
    studente *s = new studente(4, 123456);
    borsista *b = new borsista(3, 654321, 50000);
    studente* b1 = b;

    s->chiSei();
    b->chiSei();
    b1->chiSei();      //oggetto borsista, ma puntatore a studente
}
```

| sono uno studente  
| sono un borsista  
| sono uno studente

Tramite le *funzioni virtuali*, in una gerarchia di classi, il metodo/funzione da chiamare **ATTRaverso Puntatori** viene scelta *dinamicamente a tempo di esecuzione*.

Infatti la funzione chiamata non è più dettata dal tipo del puntatore, ma dal tipo dell'oggetto puntato.

La definizione **virtual** si propaga ed ha effetto **su tutta la gerarchia**.

Nell'istanza di un oggetto, le funzioni **virtual**, non sono altro che un puntatori a funzioni, che puntano le versioni correnti delle funzioni.

```
In [ ]: class studente{
    int esami;
    int matricola;
public:
    studente(int e, int m){
        esami = e;
        matricola = m;
    };

    int qualeMatricola(){
        return matricola;
    };

    void virtual chiSei(){
        cout << "sono uno studente" << endl;
    };
};

class borsista : public studente{
    int borsa;
public:
    borsista(int e, int m, int b) : studente(e, m){
        borsa = b;
    };

    //? Non è necessario rispecificare virtual
    void virtual chiSei(){
        cout << "sono un borsista" << endl;
    };
};

int main(){
    studente *s = new studente(4, 123456);
    borsista *b = new borsista(3, 654321, 50000);
    studente* b1 = b;

    s->chiSei();
    b->chiSei();
    b1->chiSei(); // puntatore a studente, ma l'oggetto puntato è borsista
}
```

*sono uno studente*

*sono un borsista*

*sono un borsista*

```
In [ ]: class studente{
    int esami;
    int matricola;
public:
    studente(int e, int m){
        esami = e;
        matricola = m;
    };

    int getMatricola{
        return matricola;
    };
};
```

```

        void virtual chiSei(){
            cout << "Sono uno studente" << endl;
        };
    };

    class borsista : public studente{
        int borsa;
    public:
        borsista(int e, int m, int b) : studente(e, m){
            borsa = b;
        };

        void chiSei(){
            cout << "Sono un borsista" << endl;
        };
    };

    void stampa(studente* s){
        s->chiSei();
        cout << "Matricola: " << s->getMatricola << endl;
    }

    int main(){
        studente* s[2];
        s[0] = new studente(7, 777777);
        s[1] = new borsista(10, 888888, 500000);

        for(int i = 0, i < 2, ++i)
            stampa(s[i]);
    }
}

```

*Sono uno studente*

*Matricola: 777777*

*Sono un borsista*

*Matricola: 888888*

```

In [ ]: class uno{
public:
    void f(){
        cout << "Uno" << endl;
    };
};

class due : public uno{
public:
    void virtual f(){
        cout << "Due" << endl;
    };
};

class tre: public due{
public:
    void f(){
        cout << "Tre" << endl;
    };
};

```

```

class dueBis: public uno{
public:
    void f(){
        cout << "DueBis" << endl;
    };
};

int main(){
    uno* u = new uno();
    uno* d = new due();
    uno* dB = new dueBis();
    uno* t = new tre();

    u->f();
    d->f();
    dB->f();
    t->f();
}

```

*Uno*

*Due*

*Uno*

*Tre*

I costruttori **NON POSSONO** essere resi **virtual**.

Tuttavia è possibile farlo per i distruttori, così da poter eseguire **delete** anche da puntatori di classi "più generiche".

```

In [ ]: class uno{
public:
    uno() {};
    virtual ~uno(){
        cout << "rompo 1" << endl;
    };
};

class due : public uno{
public:
    due() {};
    ~due(){
        cout << "rompo 2" << endl;
    };
};

class unoB{
public:
    uno() {};
    ~uno(){
        cout << "rompo 1" << endl;
    };
};

```

```

class dueB : public unoB{
public:
    due();
    ~due(){
        cout << "rompo 2" << endl;
    };
};

int main(){
    uno* u = new due;
    unoB* uB = new dueB;

    delete u;
    cout << endl;
    delete uB;
}

```

*Rompo 2*

*Rompo 1*

*Rompo 1*

## Classi Astratte e Polimorfismo

Servono come classe base nelle derivazioni.

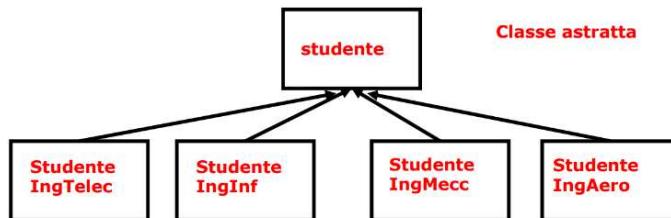
Viene specializzata nelle classi derivate.

Definisce una **interfaccia unica** verso le applicazioni.

Non viene definita completamente, in quanto contiene *almeno* una funzione **virtuale pura**.

Una **funzione virtuale pura** è una funzione (ereditata o no) senza definizione:  
 $F(\dots) = 0$

In una classe astratta non è possibile istanziare oggetti.



```
In [ ]: class studente{
    const int matricola;
    int esami;
public:
    studente(int m): matricola(m){
        esami = 0;
    };
    void virtual chiSei() = 0;      /* Funzione Virtuale Pura
};

class studenteIngInf{
    //...
public:
    studenteIngeInf(int m) : studente(m){
        // ...
    };

    void chiSei(){
        cout << "Studente Ingegneria Informatica" << endl;
    };
};

class studenteIngAero{
    //...
public:
    studenteIngeInf(int m) : studente(m){
        // ...
    };

    void chiSei(){
        cout << "Studente Ingegneria Aerospaziale" << endl;
    };
};

int main(){
    studente s;
    //! ERRORE - studente classe astratta

    studente *s;

    studente* studenti[3];
    studenti[0] = new studenteIngInf(777777);
    studenti[1] = new studenteIngAero(777777);

    for(int i = 0; i < 2; ++i){
        studenti[i].chiSei();
    }
}
```

*Studente Ingegneria Informatica*

*Studente Ingegneria Aerospaziale*

## Classi Modello Astratte

Sono classi astratte basate su **template**:

```
In [ ]: template<class T>
class uno{
    T a;
public:
    uno(T x){
        a = x;
        cout << a << endl;
    };
};

template<class type>
class dueT: public uno<type>{
    type b;
public:
    dueT(type x, type y): uno<type>(x){
        b = y;
        cout << b << endl;
    };
};

class due : public uno<int>{
    int b;
public:
    due(int x, int y) : uno<int>(x){
        b = y;
        cout << b << endl;
    };
};

template<class tipo1, class tipo2>
class dueTT: public uno<tipo1>{
    tipo2 b;
public:
    due(tipo1 x, tipo2 y): uno<tipo1>(x){
        b = y;
        cout << b << endl;
    };
};

int main(){
    dueT<int> obj(1, 2);
    // due<int>::due<int>(1, 2)
    // uno<int>::uno<int>(1)
    due obj2(3, 4);
    // uno<int>::uno<int>(3)
    // due::due(3, 4)
    dueTT<int, double> obj3(5, 6.6);
    // uno<int>::uno<int>(5)
    // due<int, double>::due<int, double>(5, 6.6)
    dueTT<int, int> obj4(7, 8);
    // uno<int>::uno<int>(7)
    // due<int, int>::due<int, int>(7, 8)

}
```

# Gestione delle Eccezioni

Le **eccezioni** sono gli errori che nascono a tempo di esecuzione (*segmentation fault, divisioni per 0, errori di tipizzazione nell'inserimento dei dati ...*), che non è possibile rilevare dalla compilazione.

Fino ad adesso, ogni qualvolta queste eccezioni venissero sollevate, il nostro programma **terminava**, con un determinato codice di errore che rappresentava l'eccezione.

Esite tuttavia un metodo formale e ben definito per poter gestire queste eccezioni a *runtime*.

Questo metodo introduce una **netta separazione** tra il codice che rileva l'eccezione e il codice che lo gestisce.

Questo metodo è quello dei blocchi **try{}catch()**.

```
In [ ]: // Codice indipendente

try{
    // Codice con possibili errori a runtime
    throw esprssione1;      // Lancio Eccezione1

    // Codice con possibili errori a runtime
    throw espressioneN;     // Lancio EccezioneN
}
catch(tipo1 e){
    // Gestisco il caso in cui sia stato sollevata
    // L'eccezione1
}
// ...
catch(tipoN e){
    // Gestisco il caso in cui sia stato sollevata
    // L'eccezioneN
}

// Altro codice indipendente
```

Se viene lanciata un'eccezione (**throw**), l'esecuzione del blocco **try** viene interrotta.

Le eccezioni lanciate durante l'esecuzione del blocco **try** sono gestite dai **gestori (catch)**: la gestione è scelta in base al tipo dell'eccezione lanciata.

Dopo la gestione dell'eccezione, l'esecuzione prosegue normalmente con altro codice che *non considerano* le altre clausole **catch**.

Se non vengono lanciate eccezioni l'esecuzione prosegue ignorando le clausole **catch**.

Se l'eccezione lanciata non viene gestita da nessun blocco **catch**, il programma termina con errore.

Le eccezioni possono essere lanciate soltanente durante l'esecuzione di un blocco **try**.

```
In [ ]: void div(int x, int y){
    try{
        if(y == 0)
            throw "divisione per 0";           // Eccezione const char*
        cout << x/y << endl;
    }
    catch(const char* p){                     // catch Eccezione const char*
        cout << p << endl;
    }

    cout << "fine div" << endl;
}

void divRotto(int x, int y){
    try{
        if(y == 0)
            throw 0;                      // Eccezione int*
        cout << x/y << endl;
    }
    catch(const char* p){                     // catch Eccezione const char*
        cout << p << endl;
    }

    cout << "fine div" << endl;
}

void positive_div(int x, int y){
    try{
        if(y == 0) throw 0;
        if((x < 0 && y > 0) || (x > 0 && y < 0)) throw 1;
        cout << x/y << endl;
    }
    catch(int e){
        if(e == 0)
            cout << "divisione per 0" << endl;
        else
            cout << "risultato negativo" << endl;
    }

    // Oppure

    try{
        if(y == 0) throw '0';
        if((x < 0 && y > 0) || (x > 0 && y < 0)) throw 1;
        cout << x/y << endl;
    }
    catch(char p){
        cout << "divisione per 0" << endl;
    }
    catch(int e){
        cout << "risultato negativo" << endl;
    }

    cout << "Fine div_pos" << endl;
}
```

```

int main(){
    int x,y;
    cin >> x >> y;

    div(x,y);

    divRotto(x,y);
    cout << "fine main" << endl;
}

```

In:

10 5

Out:

2  
*fine div*  
*fine main*

---

In:

10 0

Out:

*divisione per 0*  
*fine div*  
*fine main Programma Termina perché l'eccezione non è gestita*

```

In [ ]: void div(int x, int y){
    if(y == 0)
        throw "divisione per 0";

    cout << x/y << endl;
}

int main(){
    int x, y;
    cin >> x >> y;

    div(x, y);
    //! ERRORE - Non ammesso
    try{
        div(x, y);
    }
    catch(const char* p){
        cout << p << endl;
    }
}

```

Non sono ammesse conversioni implicite (tranne da sottotipo a sopratipo).

L'eccezione viene gestita a *runtime* esaminando i gestori nell'ordine in cui compaiono, a partire dal blocco più recente incontrato.

Viene scelto il primo con argomento corrispondente all'eccezione lanciata.

Esiste inoltre una clausola **catch** indicato dal carattere (...), che gestisce qualsiasi eccezione. E' buona prassi inserirla come ultima opzione.

```
In [ ]: void f(int x){
    if(x == 0) throw 0;
    if(x > 100) throw 'a';
}

void g(int x){
    try{
        f(x);
    }
    catch(int){
        cout << "eccezione da g" << endl;
    }
}

int main(){
    try{
        int x;
        cin >> x;
        g(x);
    }
    catch(int){
        cout << "eccezione da main" << endl;
    }
    catch(...){
        cout << "eccezione da main non prevista" << endl;
    }
}
```

In:

```
| 0
```

Out:

```
| eccezione da g
```

---

In:

```
| 1010
```

Out:

```
| eccezione da main non prevista
```

Dato che le eccezioni non sono altro che tipi, possiamo creare delle classi ad-hoc che rappresentano le nostre eccezioni.

```
In [ ]: class eccezioneStack{
    int e;
public:
    eccezione(int n){
        e = n;
    };
    void print(){
        if(e == 0)
            cout << "stack pieno" << endl;
        else
            cout << "stack vuoto" << endl;
    };
};

class Stack{
    int *v;
    int size_;
    int top;
public:
    Stack(int n){
        size_ = n;
        v = new int[size_];
        top = -1;
    };

    Stack& push(int n){
        if(top == size_)
            throw eccezioneStack(0);
        v[top++] = n;
        return *this;
    };

    void pop(int& n){
        if(top == -1)
            throw new eccezioneStack(1);
        n = v[top--];
    };
}

int main(){
    Stack s(10);
    for(int i = 0; i < n; i++){
        s.push(1);
    }
    try{
        s.push(10);
    }
    catch(eccezioneStack e){
        e.print();
    }
    catch(eccezioneStack *e){
        e->print();
        delete e;
    }
}
```

E' inoltre possibile creare classi eccezione astratte, sulle quali derivo classi eccezione.

```
In [ ]: class StackEcc{
public:
    void msg(){
        cout << "attenzione :";
    };
    void virtual print() = 0;
};

class StackFull : public StackEcc{
    int e;
public:
    StackFull(int n){
        e = n;
    };
    void print(){
        msg();
        cout << e << "non inserito" << endl;
    };
};

class StackEmpty: public StackEcc{
public:
    StackEmpty(){}
    void print(){
        msg();
        cout << "stack vuoto" << endl;
    };
};

Stack& Stack::push(int n){
    if(top == size_)
        throw new StackFull(n);
    v[top++] = n;
    return *this;
}

void Stack::pop(int &n){
    if(top == -1)
        throw new StackEmpty();
    n = v[top--];
}

int main(){
    try{
        Stack s(2);
        s.push(10).push(0).push(676);
    }
    catch(StackEmpty* e){
        e->print();
        delete e;
    }
    catch(StackFull* e){
        e->print();
        delete e;
    }
    catch(...){
        cout << "Boh qualcosa è andato storto" << endl;
    }
}
```

# **Buona Programmazione**

## **Tecniche di Debug**

Il debug è l'arte di rimuovere i bug dal codice.

Esistono diversi metodi di debug:

- **Debug tramite testo:** inserimento di frasi (imprecazioni) nel testo per vedere quel'è l'ultima visualizzata;
- **Debug visuale:** vedere a schermo il valore di tutte le variabili coinvolte per monitorarne il comportamento;
- **Debug tramite Debugger (*GDB, DDD*):** tecniche per eseguire il codice passo passo;
- **Debug tramite Compilatore:** sfruttare il compilatore;
- **Debug tramite Valgrind:** tool di analisi della memoria. Permette di salvarci dal *segmentation fault*.

---

Attraverso operazioni di "debug", è possibile verificare anche il numero di volte che si entra in determinate parti di codice, così da verificare più efficacemente la complessità di un codice (**Profiling**).

---

## **Lettura Dati in Input**

La maggior parte dei programmi viene eseguita con valori che non sono conosciuti a tempo di compilazione.

Per fare ciò si utilizzano delle funzioni e dei metodi già implementati in C++.

- Scrittura file da Tastiera: `std::cin`
- Lettura file da Tastiera: `std::cout`
- Lettura, scrittura su file: `std::fstream`

Tuttavia noi sostituiremo lo stream di input da tastiera con quello da file.

Per fare ciò, da *Unix*, il comando per la redirezione è il seguente:

`./eseguibile <file`

# Comandi Base Unix

Sono comandi per sistemi operativi eseguibili tramite il terminale (linea di comando).

Qualsiasi comando se preceduto dalla parola **man** ne farà vedere il funzionamento.

## Comandi di Navigazione

- **setxkbmap** *it*: seleziona il template della tastiera che si vuole utilizzare
- **ls**: (*list*) Lista dei file e cartelle presenti nella cartella corrente (*directory*)
- **pwd**: (*Path Within Directory*) nome e path della cartella corrente
- **cd**: (*Change Directory*), permette di accedere ad una sottocartella della cartella corrente
- **cd ..** : ritorno alla cartella padre

## Comandi di manipolazione file

- **mkdir** *cartella*: (*Make Directory*) creare una cartella
- **touch** *file*: crea un nuovo file nuovo
- **cp/mv**: copia/taglia la cartella da una root ad un'altra
- **rm** *file*: (*remove*) rimuovere un file in maniera **definitiva**
- **rm -R** *cartella*: rimuovere una cartella, il comando **-R** indica l'esecuzione in maniera ricorsiva per cancellare tutte le eventuali sottocartelle e file.
- **nano**: editare il file direttamente dalla linea di comando
- **mousepad** : edita il file tramite applicazione *mousepad*
- **gedit** : edita il file tramite applicazione *gedit*

## Comandi per compilare/eseguire

In ordine:

- **g++ [opzioni] -o eseguibile sorgente.cpp** : compilazione
- **./eseguibile**: esecuzione (se si è nella stessa cartella)
  - **./eseguibile < file**: lettura di dati in input da file e non da stream
  - **time ./eseguibile**: restituisce il tempo di un esecuzione di file
  - **./eseguibile < fileInput | diff - fileOutput**: confronta l'output restituito dagli input da file con un set di valori presenti nel file di output. Se non sono presenti differenze non stampa niente.

## **Esecuzione di più linee di codice in un unico file**

I file contenenti linee di codice in Unix sono file .sh.

Il template da seguire è il seguente:

*File .sh*

```
#!/bin/bash  
-- comandi vari
```

*Terminale*

```
chmod +x mio_script.sh  
.mio_script.sh
```

---

## **Flag di Compilazione**

Sono dei flag che ci possono aiutare

- *g++ -w ...*: mostra i warning a tempo di compilazione
- 

## **Gestione Segmentation Fault**

Un errore comune e inevitabile quando si utilizza memoria dinamica, è il **segmentation fault**.

Per riuscire ad affrontare questa iaga si può utilizzare il tool **valgrind**, che è una specie di "babysitter della memoria":

- Controlla gli accessi
- Conta gli accessi

Oltre a fare ciò **valgrind** ci fornisce dati su:

- Numero di volte che una funzione viene chiamata
- Tempo medio di esecuzione per ogni funzione