

Introduzione

Problema

Quesito di cui si richieda ad altri o a sé stessi la soluzione, partendo di solito da elementi noti.

Un problema computazionale: problema risolvibile mediante algoritmi, espresso in generale come corrispondenza tra input e output.

Istanza

particolare occorrenza del problema data da una specifica configurazione degli input

Algoritmo

Def: *Un insieme ordinato di operazioni/istruzioni elementari, non ambigue, ed effettivamente computabili che, quando eseguito su certi dati in ingresso (input), produce un risultato (output) arrestandosi in tempo finito.*

Un modo per risolvere un problema. Esistono due diversi momenti relativi ad essi:

- specifica / rappresentazione
 - implementazione: algoritmo rappresentato in forma eseguibile (programma)
- esecuzione: un esecutore segue quanto indicato dall'algoritmo su un'istanza del problema
 - L'esecuzione di un algoritmo è il processo di risoluzione del problema

Più algoritmi possono risolvere lo stesso problema, problema del confronto e scelta dell'algoritmo migliore, e uno stesso algoritmo può essere rappresentato in modi diversi.

L'**informatica** è la disciplina che cerca di fornire il fondamento scientifico a vari argomenti, l'**algoritmo** è una sequenza finita di passi formali (non ambigui ed eseguibili dalla macchina) che trasformano un input in un output invece un **programma** è la rappresentazione di un algoritmo comprensibile dalla macchina.

Requisiti/proprietà

- *Ordinamento delle operazioni:* un algoritmo dovrebbe avere una struttura dove è chiaro l'ordine di esecuzione dei suoi passi.
- *Non ambiguità:* l'esecutore deve poter interpretare le istruzioni in modo univoco.
- *Istruzioni effettivamente computabili:* l'esecutore deve essere in grado di eseguire le operazioni indicate.
- *Finitezza:* l'algoritmo dovrebbe consistere in un numero finito di passi, richiedere un numero finito di input/risorse, e terminare in tempo finito.

Programmazione imperativa

In questo stile, gli algoritmi sono espressi come sequenze di istruzioni che il computer deve eseguire e che modificano lo stato del programma; coerente con l'architettura di von Neumann e con il modo con cui i computer funzionano a livello hardware.

Questo tipo di paradigma si concentra sul “come” invece che sul “cosa” (paradigmi dichiarativi).

Caratteristiche

- *Correttezza*: un algoritmo è corretto se, per ogni istanza del problema, termina con l'output corretto.
- *Generalità*: un algoritmo dovrebbe applicarsi a una tipologia di problemi, e non solamente a specifiche istanze.
- *Determinismo*: partendo dagli stessi input, si dovrebbero ottenere gli stessi output.
- *Efficienza*: l'esecuzione dell'algoritmo dovrebbe avere un costo accettabile.

Programmazione strutturata

Paradigma di programmazione basato sull'uso di costrutti di controllo del flusso e blocchi di codice.

Teorema di Böhm-Jacopini: *qualsiasi programma può essere scritto usando tre tipi di strutture di controllo: sequenza, selezione e iterazione.*

Si usa lo pseudo codice per astrarre da dettagli specifici dei linguaggi di programmazione e per semplificare la notazione e migliorare la leggibilità.

Non esiste nessuno standard per la sintassi dello pseudocodice, ognuno può scriverselo come vuole.

Algoritmi ricorsivi

Un algoritmo viene detto ricorsivo quando nel suo corpo richiama se stesso.

Principio di induzione:

Data una proprietà P . Se $P(0)$ è vera e $P(n) \Rightarrow P(n + 1)$ per ogni n , allora $P(n)$ è vera per ogni n .

$$(\forall P)[P(0) \wedge (\forall k \in N)(P(k) \Rightarrow P(k + 1))] \Rightarrow (\forall n \in N)[P(n)]$$

Principio per dimostrazioni:

1. *caso base*: si dimostra che $P(0)$ o $P(1)$ è vera
2. *passo induttivo*: assumendo $P(n)$ vera, si dimostra che anche $P(n + 1)$ lo è

Funzioni ricorsive

Una funzione ricorsiva corretta ha le seguenti caratteristiche:

- include uno o più casi base in cui il valore della funzione viene calcolato senza invocazione ricorsiva;
- includa una o più invocazioni ricorsive della funzione, tipicamente con argomenti che progressivamente “avvicinano” ai casi base.

Divide-et-impera

Con questa tecnica dividiamo il problema in sottoproblemi più facili e li risolviamo uno a uno e li combiniamo insieme per risolvere il problema iniziale.

Nella creazione di un algoritmo ricorsivo bisogna pensare prima al caso base e successivamente al passo della ricorsione

Tipi di ricorsione

1. **Diretta**: la procedura richiama direttamente se stessa;
2. **Indiretta**: la procedura invoca un'altra procedura che richiama (direttamente o indirettamente) la procedura originaria;
3. **Lineare**: la procedura include una sola chiamata ricorsiva (es. fattoriale);
4. **Multipla**: la procedura include più di una chiamata ricorsiva (es. Fibonacci) se ci sono solo due chiamate è detta *binaria*;
5. **Mutua**: caso particolare di ricorsione indiretta dove la procedura A chiama una procedura B che chiama nuovamente la procedura A in modo diretto (es. pari/dispari);
6. **Coda**: caso particolare di ricorsione lineare in cui la chiamata ricorsiva è l'ultima istruzione della procedura;
7. **Annidata (innestata)**: la chiamata ricorsiva ha come argomento un'ulteriore chiamata ricorsiva (es. funzione di Ackermann);
8. **Infinita**: quando non vi è (riduzione del problema che porta a) caso base gestito senza ricorsione.

Esempi

Lineare:

```
1 # linear recursion (non-tail)
2 def fact(n):
3     if n <= 0: return 1
4     return n * fact(n-1)

1 # linear recursion (tail)
2 def fact_tailrec(n, acc=1):
3     if n <= 1: return acc
4     return fact_tailrec(n-1, n*acc)

1 # linear recursion (tail)
2 def min_rec(lst):
3     if len(lst) == 2: return min(lst[0], lst[1])
4     if len(lst) == 1: return lst[0]
5     if len(lst) <= 0: raise Exception("empty list")
6     return min_rec(lst[1:] if lst[0] > lst[-1] else lst[0:-1])
```

Multipla

Require: $n \geq 1$

```
1: function FIB( $n$ )
2:   if  $n \leq 2$  then return 1
3:   else return FIB( $n - 1$ ) + FIB( $n - 2$ )
```

```
1 def fib(n):
2     if n <= 2: return 1
3     return fib(n-2) + fib(n-1)
```

Mutua

```
1 # mutual recursion
2 def even(n):
3     if n==0: return True
4     return odd(n+(-1 if n>0 else +1))
5
6 def odd(n):
7     if n==0: return False
8     return even(n + (-1 if n>0 else +1))
```

Efficienza degli algoritmi

Caratterizzare l'efficienza è un aspetto importante dell'analisi e progettazione di algoritmi, esistono due nozioni fondamentali:

- efficienza in tempo
- efficienza in spazio

Tempi di esecuzione

τ (Tau) è la funzione che serve per identificare il tempo di esecuzione, determinata dalla natura e struttura dell'algoritmo.

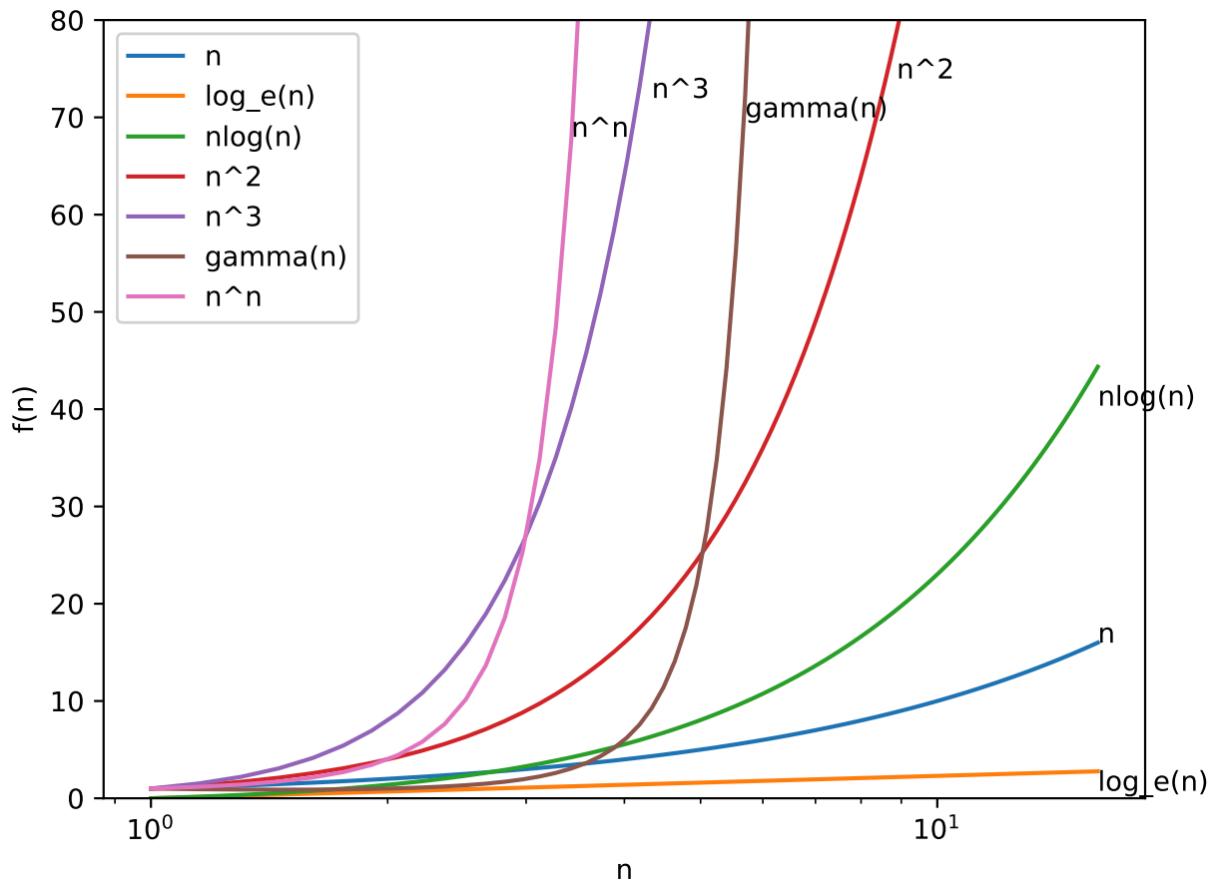
La τ dipende dalla dimensione n dell'input, dalle istanze dell'input (se dobbiamo valutare un algoritmo di ordinamento il modo in cui l'array ci arriva è essenziale) e dall'esecutore.

In sostanza:

“La variazione del tempo d'esecuzione al variare della dimensione n dell'input può essere caratterizzata da una funzione $\tau(n)$ ”

Alcune forme frequenti di $\tau(n)$ sono:

- **meno che lineare:** $\log_e(n), \log_2(n), \sqrt{n} \rightarrow$ all'aumento di n il tempo di esecuzione aumenta lentamente
- **lineare:** $k \cdot n$
- **più che lineare:** $n * \log(n), n^k$ (polinomiale), k^n (esponenziale), $n!, n^n \rightarrow$ all'aumento di n il tempo di esecuzione aumenta velocemente



Per una dimensione n esistono tre casi:

1. **Best case:** le configurazioni della struttura dati di ingresso d che danno luogo al tempo minimo;
2. **Average case:** le configurazioni della struttura dati di ingresso d ritenute “normali” (e.g., più frequenti in pratica);
3. **Worst case:** le configurazioni della struttura dati di ingresso d che danno luogo al tempo massimo.

La prima casistica non è rilevante, invece:

- la *complessità worst-case* serve per capire i limiti di applicabilità pratica permettendo di fare ragionamenti sulla safety (controlli real-time);
- la *complessità average-case* è difficile da valutare essendo non semplice caratterizzare input tipici.

Andamento asintotico

Essendo che $\tau(n)$ è influenzato dal hardware o dal linguaggio di programmazione si vuole astrarre da questi aspetti, infatti non ci interessa quanto tempo ci mette un determinato computer rispetto ad un altro ma bisogna capire l'andamento del tempo asintoticamente quindi al crescere dell'input.

Anche con tre tempi di esecuzione differenti (essendo fatti su hardware diversi) essi potrebbero essere accomunati dallo stesso andamento.

Modello di calcolo

Durante lo studio degli algoritmi si considera un modello di calcolo standard chiamato **RAM** (*Random Access Machine*), cioè un sistema mono-processore (istruzioni in sequenza) dove le istruzioni semplici richiedono tempo costante.

Con $T(n)$ denotiamo il tempo stimato per l'esecuzione dell'algoritmo nella RAM (stima tempo effettivo $\tau(n)$).

Complessità computazionale

È l'ordine di grandezza della funzione $T(n)$

Ordini di infiniti

Una funzione $f(n)$ tale che $\lim_{n \rightarrow c} f(n) = \infty$ è infinita per $n \rightarrow c$.

Due funzioni $f(n)$ e $g(n)$ sono **infiniti simultanei** se entrambe sono infinite per $n \rightarrow c$.

Quindi se abbiamo funzioni con dei tempi di esecuzione $n \log(n)$ o n^3 con n che tende all'infinito avremo solo funzioni tendenti ad esso e non saremo in grado di capire quale ha tempi di esecuzione migliori. Quindi per confrontare diversi infiniti dobbiamo determinare quale “*tende all'infinito più rapidamente*” andando a studiare il limite del rapporto ∞/∞ (forma indeterminata).

Le casistiche sono:

Una funzione $f(n)$ è, rispetto a $g(n)$

- **infinito di ordine superiore** se: $\lim_{n \rightarrow c} \frac{f(n)}{g(n)} = \infty$
- **infinito di ordine inferiore** se: $\lim_{n \rightarrow c} \frac{f(n)}{g(n)} = 0$
- **infinito dello stesso ordine** se: $\lim_{n \rightarrow c} \frac{f(n)}{g(n)} = \ell \neq 0$
- **non confrontabile** se non esiste $\lim_{n \rightarrow c} \frac{f(n)}{g(n)}$

Analisi asintotica

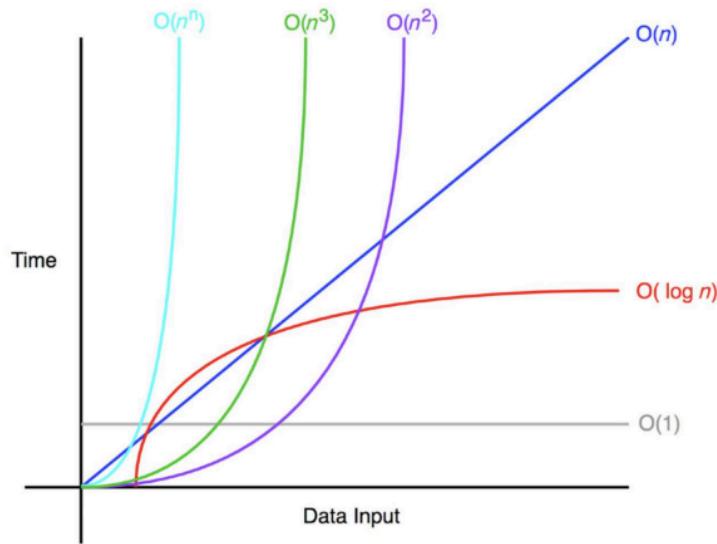
Abbiamo capito che il tasso di crescita del tempo stimato di esecuzione $T(n)$ di un algoritmo da una semplice caratterizzazione della relativa efficienza e calcolare il tempo esatto non serve visto che per input grandi le costanti moltiplicative e i termini di ordine inferiore del tempo effettivo di esecuzione sono trascurabili ed solo la forma della funzione di n che definisce l'andamento.

Efficienza asintotica | Def:

È quella che si studia per dimensioni di n tali per cui solo l'ordine del tasso di crescita è rilevante

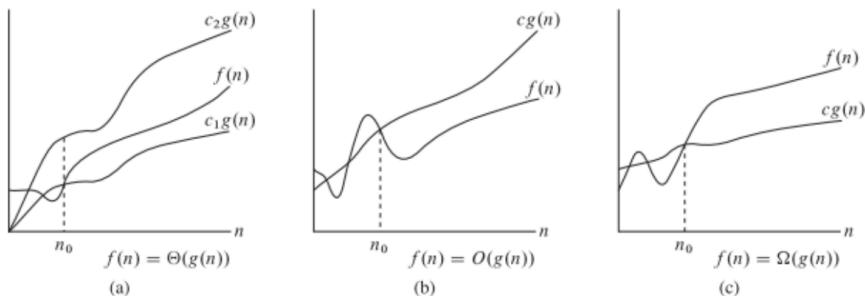
Tempo di esecuzione asintotico | Def:

Approssimazione del tempo d'esecuzione di un algoritmo mediante una "più semplice" funzione di simile ordine di crescita



Notazione asintotica

Notazione standard per caratterizzare l'efficienza in tempo di un algoritmo



Si suddividono in altre 3 notazioni che sono:

- **O [O-grande]:** data una funzione (di confronto) $g(n)$:

$O(g(n))$ individua tutte le funzioni $f(n)$ tale per cui esiste una costante reale maggiore di 0 (c, n_0) tale per cui questa condizione è verificata:

$$0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \quad (f(n) \text{ compresa tra } 0 \text{ e } c \cdot g(n) \text{ per tutti i valori di } n > n_0).$$

Quindi possiamo individuare un certo valore n_0 di dimensioni del problema e una certa costante moltiplicativa tale per cui dal punto n_0 individuato, la funzione $g(n)$ (moltiplicata per un fattore a scelta) va a dominare dall'alto la nostra funzione.

In sostanza:

$g(n)$ è un limite superiore o upper bound asintotico per $f(n)$
 $f(n)$ asintoticamente cresce come o meno di $g(n)$

Esempio:

Dimostrare che $n^2 \in O(n^3)$

$$\begin{array}{ccc} \downarrow & & \downarrow \\ f(n) & & g(n) \end{array}$$

Dobbiamo usare la definizione di O , quindi esiste una costante c, n_0 tale per cui $f(n)$ è maggiorata da $c \cdot g(n)$ e questa cosa è valida per ogni $n \geq n_0$.

Applichiamo la disequazione che la formula ci da:

$$\begin{aligned} 0 \leq f(n) \leq cg(n) &\Rightarrow 0 \leq n^2 \leq c \cdot n^3 \\ &\Downarrow \\ 0 \leq 1 \leq c \cdot n &\Rightarrow c \geq \frac{1}{n} \end{aligned}$$

- **Ω [Omega-grande]:** data una funzione $g(n)$:

$\Omega(g(n))$ individua tutte le funzioni $f(n)$ tale per cui esiste una costante reale maggiore di 0 (c, n_0) tale per cui questa condizione è verificata:

$0 \leq g(n) \leq f(n), \forall n \geq n_0$ ($g(n)$ compresa tra 0 e $f(n)$ per tutti i valori di $n > n_0$).

In sostanza:

$g(n)$ è un limite inferiore o lower bound asintotico per $f(n)$

e

$f(n)$ asintoticamente cresce come o più di $g(n)$

- **Θ [Theta-grande]:** data una funzione $g(n)$:

$\Theta(g(n))$ individua tutte le funzioni $f(n)$ tale per cui esiste una costante reale maggiore di 0 (c_1, c_2, n_0) tale per cui questa condizione è verificata:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$$

Quindi $f(n)$ da un certo punto in poi (n_0) è compresa tra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$

In sostanza:

$g(n)$ è un limite asintotico stretto o tight bound per $f(n)$

e

$g(n)$ e $f(n)$ hanno lo stesso ordine di grandezza

NB: se $f(n)$ appartiene a **O-grande** di n^2 allora sicuramente apparterrà anche a **O-grande** n^3 , se invece $f(n)$ appartiene a **Theta-grande** di n^2 sicuramente non apparterrà ne a **Theta-grande** di n ne a **Theta-grande** di n^3 .

Proprietà e osservazioni

- **Relazioni:** una funzione f appartiene a $\Theta(g(n))$ se solo se la funzione appartiene anche a $O(g(n))$ e $\Omega(g(n))$;
- **Regola della somma:** se una funzione $T(n)$ appartiene a $\Theta(f(n)) + \Theta(g(n))$ allora possiamo semplificare considerando Theta-grande come il massimo delle due funzioni [$\Theta(\max(f(n), g(n)))$];

- **I termini di grado inferiore non interessano:** se abbiamo $T(n) = n^3 + 999n^2$ ci interessa valutare solo quello con il grado maggiore, essendo gli altri “contenuti” in esso;

Complessità costrutti

Per ogni costrutto semplice abbiamo le seguenti complessità:

- **Istruzioni semplici:** qui ci sono assegnamenti e operazioni aritmetiche/logiche/relazionali; tutti hanno $\Theta(1)$.
- **Sequenza di istruzioni semplici:** uguale a sopra ($\Theta(1)$).
- **Costrutti selettivi:** con condizione pari a $\Theta(1)$:
 - $O(\max(T_{\text{then}}, T_{\text{else}}))$;
 - $\Omega(\min(T_{\text{then}}, T_{\text{else}}))$;
 - Caso peggiore: $\Theta(\max(T_{\text{then}}, T_{\text{else}}))$;
 - Caso maggiore: $\Theta(\min(T_{\text{then}}, T_{\text{else}}))$.
- **Costrutti iterativi:**
 - *for* per $k(n)$ iterazioni e contando che $T_{\text{init}} = T_{\text{cond}} = T_{\text{inc}} = \Theta(1)$
 - $O(T_{\text{init}} + k(n) \cdot (T_{\text{cond}} + T_{\text{body}} + T_{\text{inc}}) + T_{\text{cond}}) = O(k(n)T_{\text{body}})$;
 - $\Omega(T_{\text{init}} + k(n) \cdot (T_{\text{cond}} + T_{\text{body}} + T_{\text{inc}}) + T_{\text{cond}}) = \Omega(k(n)T_{\text{body}})$;
 - Caso peggiore: $\Theta(k_{\text{worst}}(n) T_{\text{body,worst}})$;
 - Caso maggiore: $\Theta(k_{\text{best}}(n) T_{\text{body,best}})$.
 - *while* e in funzione del numero min e max di iterazioni ($k_{\text{min}}, k_{\text{max}}$)
 - $O(k_{\text{max}} T_{\text{body}})$ e $\Omega(k_{\text{min}} T_{\text{body}})$;
 - Caso peggiore: $\Theta(k_{\text{max}} \cdot T_{\text{body,worst}})$;
 - Caso maggiore: $\Theta(k_{\text{min}} \cdot T_{\text{body,best}})$.

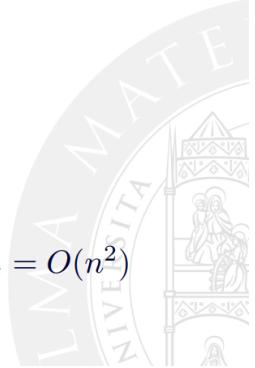
Esempio

Bubble Sort

```
1 def bubble_sort(v):
2     swap = True
3     n = len(v)
4     i = 0
5     while swap and i < n-1:
6         swap = False
7         for j in range(n-1-i):
8             if v[j] > v[j+1]:
9                 swap = True
10                v[j], v[j+1] = v[j+1], v[j]
11                i += 1
```

- array già ordinato: 1 sola iterazione del while
- array contrordinato: n iterazioni del while
- array non ordinato (caso più frequente)

- $T_{while,best} = 1 \cdot T_{for}$ $T_{while,worst} = n \cdot T_{for}$
- $T_{for} = n(\Theta(1)) = \Theta(n)$
- $T_{bs,best} = T_{while,best} = T_{for} = \Theta(n)$
- $T_{bs,worst} = T_{while,worst} = nT_{for} = n\Theta(n) = \Theta(n^2)$
- Possiamo dire che $T_{bs} = \Omega(n)$ (lineare nel caso migliore) e $T_{bs} = O(n^2)$ (quadratico nel caso peggiore).
 - anche $\Omega(1)$ (più che costante) e $O(n^3)$ (meno che cubico)



Ricorrenze

Il modo di calcolare la complessità degli algoritmi visto precedentemente non è applicabile alle funzioni ricorsive, infatti per calcolare la complessità, dobbiamo considerare anche il costo della chiamata ricorsiva andando a ottenere **equazioni ricorrenti**.

Ora distinguiamo due contributi:

- $f(n)$: tempo di tutte le istruzioni che **non contengono** chiamate ricorsive;
- $T(k)$: tempo derivante dalle chiamate ricorsive, invocate su $k < n$ (istanze più piccole del problema—cf. divide-et-impera).

Metodi di risoluzione delle ricorrenze

Esistono tre metodi principali:

1. **Metodo iterativo:** si espande l'equazione fino ad arrivare a una espressione in funzione di n e costanti;

```

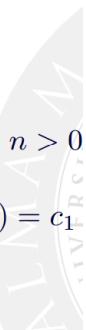
1: function FACTORIAL( $n$ )
2:   if  $n = 0$  then return 1
3:   else return  $n \times \text{FACTORIZATION}(n-1)$ 

```

$$T(n) = \begin{cases} c_1 & n = 0 \\ T(n-1) + c_1 & n > 0 \end{cases}$$

- Applicando il metodo iterativo:

$$\begin{aligned} T(n) &= T(n-1) + c_1 = T(n-2) + 2c_1 = \dots \\ &= T(n-k) + kc_1 \end{aligned}$$



- La ricorrenza si chiude per $k = n$ dove abbiamo $T(n-k) = T(0) = c_1$
- Dunque

$$T(n) = (n+1)c_1 = \Theta(n)$$

```

1: function FIB( $n$ )
2:   if  $n \leq 2$  then return 1
3:   else return FIB( $n-1$ ) + FIB( $n-2$ )

```

$$T(n) = \begin{cases} c_1 & n \leq 2 \\ T(n-1) + T(n-2) + c_2 & n > 2 \end{cases}$$

- Si noti come la formula della ricorrenza sia identica alla formula del calcolo
 Il metodo iterativo non aiuta particolarmente

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c_2 = 2T(n-2) + T(n-3) + 2c_2 \\ &= 3T(n-3) + 2T(n-4) + 4c_2 = 5T(n-4) + 3T(n-5) + 7c_2 \\ &= 8T(n-5) + 5T(n-6) + 12c_2 = 13T(n-6) + 8T(n-7) + 20c_2 = \dots \end{aligned}$$

- Però possiamo trovare un limite inferiore sfruttando $T(n) \geq T(n-1)$

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ &\geq 2T(n-2) \geq 4T(n-4) \geq 8T(n-6) \geq \dots \\ &\geq 2^k T(n-2k) \geq 2^{\lfloor n/2 \rfloor} c_1 \end{aligned}$$

- Occorre un numero di chiamate ricorsive almeno esponenziale in n : $\Omega(2^{\lfloor n/2 \rfloor})$
 $2^{\lfloor n/2 \rfloor} = O(2^n)$ ma $2^{\lfloor n/2 \rfloor} \neq \Theta(2^n)$ poiché $2^{\lfloor n/2 \rfloor} \neq \Omega(2^n)$ (**Esercizio: provarlo**)

- Metodo della sostituzione: in questo caso: $T(n)$ appartiene a $O(f(n))$ quindi si effettua un “guess” sulla possibile soluzione e si utilizza l’induzione matematica per dimostrare la correttezza della soluzione;

```

1: function FIB(n)
2:   if n ≤ 2 then return 1
3:   else return FIB(n-1) + FIB(n-2)

```

$$T(n) = \begin{cases} c_1 & n \leq 2 \\ T(n-1) + T(n-2) + c_2 & n > 2 \end{cases}$$

- Cerchiamo di dimostrare $T(n) = O(2^n)$
- Assumiamo $T(n') \leq c2^{n'}, \forall n' < n$
- Cerchiamo di dimostrare $T(n) \leq c2^n$
 - Partiamo dalla formula della ricorrenza: $T(n) = T(n-1) + T(n-2)$
 - Per ipotesi induttiva $T(n-1) \leq c2^{n-1}$ e $T(n-2) \leq c2^{n-2}$
 - Quindi: $T(n) \leq c2^{n-1} + c2^{n-2} + c = c(2^{n-1} + 2^{n-2} + 2^0) \leq c2^n$
- **(Altro tentativo - caso negativo)** Con l'assunto $T(n') \leq cn', \forall n' < n$ otterremmo $T(n) = c(n-1) + c(n-2) + c = c(n-1 + n-2 + 1) = c(2n-2)$ ma per nessun valore di c si ha $2cn - 2c \leq cn \rightarrow n \leq 2$ (!!)

3. **Metodo dell'esperto:** basato sul Master Theorem, per algoritmi della forma $T(n) = aT(n/b) + f(n)$ (divide-et-impera).

Esempi

```

1: function FIB(n)
2:   if n ≤ 2 then return 1
3:   else return FIB(n-1) + FIB(n-2)

```

$$T(n) = \begin{cases} c_1 & n \leq 2 \\ T(n-1) + T(n-2) + c_2 & n > 2 \end{cases}$$

- 💡 Si noti come la formula della ricorrenza sia identica alla formula del calcolo
- ⚠ Il metodo iterativo non aiuta particolarmente

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + c_2 = 2T(n-2) + T(n-3) + 2c_2 \\
&= 3T(n-3) + 2T(n-4) + 4c_2 = 5T(n-4) + 3T(n-5) + 7c_2 \\
&= 8T(n-5) + 5T(n-6) + 12c_2 = 13T(n-6) + 8T(n-7) + 20c_2 = \dots
\end{aligned}$$

- Però possiamo trovare un limite inferiore sfruttando $T(n) \geq T(n-1)$

$$\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 \\
&\geq 2T(n-2) \geq 4T(n-4) \geq 8T(n-6) \geq \dots \\
&\geq 2^k T(n-2k) \geq 2^{\lfloor n/2 \rfloor} c_1
\end{aligned}$$

- ➔ Occorre un numero di chiamate ricorsive **almeno esponenziale in n :** $\Omega(2^{\lfloor n/2 \rfloor})$

Problema della ricerca

In molte applicazioni è necessario ricercare un elemento all'interno di una struttura dati senza saperne la posizione, o non potendovi accedere direttamente.

La formulazione del problema della ricerca si basa su:

- **modalità di accesso** (casuale, sequenziale, ...);
- **rappresentazione della posizione dell'elemento** (es. indice, puntatore, ...);

- **rappresentazione della proprietà dell'elemento** (es. valore, predicato sul valore, posizione...);

Usando un accesso di tipo **casuale** l'accesso ad un elemento è costante indipendentemente dall'posizione $\Theta(1)$ invece usando l'accesso **sequenziale** tutto dipende dalla distanza tra l'elemento desiderato e l'ultimo elemento acceduto (tra x_i e x_j è $\Theta(|i - j|)$).

Negli esempi successivi si userà un **array** quindi una struttura dati con le seguenti proprietà:

- **modalità d'accesso**: accesso casuale;
- **rappresentazione posizione**: attraverso indice
- **rappresentazione proprietà**: mediante una nozione di uguaglianza *equal* che deve essere:
 - **riflessiva**: $equal(a, a)$;
 - **simmetrica**: $equal(a, b) \Rightarrow equal(b, a)$;
 - **transitica**: $equal(a, b) \wedge equal(b, c) \Rightarrow equal(a, c)$.

Def | Ricerca su sequenza indicizzata

Data una sequenza indicizzata di elementi a_1, \dots, a_n e un valore x da ricercare, si vuole trovare un indice i tale che $equal(a_i, x)$ è vera.

Tipi di ricerca

Lineare

l'idea è quella di esaminare in sequenza tutti gli elementi della struttura dati, confrontandoli con l'elemento desiderato:

```

1: function LINEAR-SEARCH(array, n, elem)
2:   i  $\leftarrow 0$ 
3:   while i < n do
4:     if equal(array[i], elem) then return i
5:     i  $\leftarrow i + 1$ 
return -1
```

Questo algoritmo ha una crescita lineare nel caso medio/peggiore, per quanto riguarda l'analisi della complessità:

- **Caso migliore**: $k = 1$: $T_{best}(n) = \Theta(1) + \Theta(1) = \Theta(1)$
- **Caso peggiore**: $k = n$: $T_{worst}(n) = \Theta(n) + \Theta(1) = \Theta(n)$
- **Caso medio**: $k = n/2$: $T_{avg}(n) = \frac{n}{2}\Theta(1) + \Theta(1) = \Theta(\frac{n}{2}) + \Theta(1) = \Theta(n)$

Minimo/massimo

Adattando l'algoritmo precedente, in quanto la proprietà non è più verificabile in modo indipendente per ogni elemento, possiamo fare una ricerca del minimo/massimo in una lista non ordinata di numeri; l'idea dietro è quella di tenere traccia del risultato parziale.

Require: $n > 0$

```
1: function BEST-SEARCH(array, n, better)
2:   currBest  $\leftarrow 0$ 
3:   i  $\leftarrow 1$ 
4:   while i  $< n$  do
5:     if better(array[i], currBest) then
6:       currBest  $\leftarrow$  array[i]
7:   return currBest
```

▷ Precondizione

▷ Il candidato iniziale è il primo elemento
▷ Considero gli elementi dopo il primo

▷ Tengo traccia del “nuovo” elemento preferito

Per qualunque coppia di oggetti *x* e *y* dev'essere definita una relazione Booleana *better* (*x*, *y*) che dice se *x* dev'essere preferito a *y*

Binaria

In alcuni casi la struttura dati d'ingresso può essere già ordinata, quindi possiamo assumere che gli elementi dell'array siano ordinati rispetto a una relazione d'ordine *less*, dove il primo elemento è minore rispetto al successivo, o *equal* dove due elementi adiacenti sono uguali.

Utile per il gioco guess a number o un dizionario

In questo algoritmo per trovare un numero all'interno della struttura dati (ordinata) servirà tenere traccia della porzione dell'array da esaminare usando due indici, *from* e *to*, che tengono traccia del primo e ultimo elemento utile. In questo modo si andrà a scartare la porzione di array che non ci interessa.

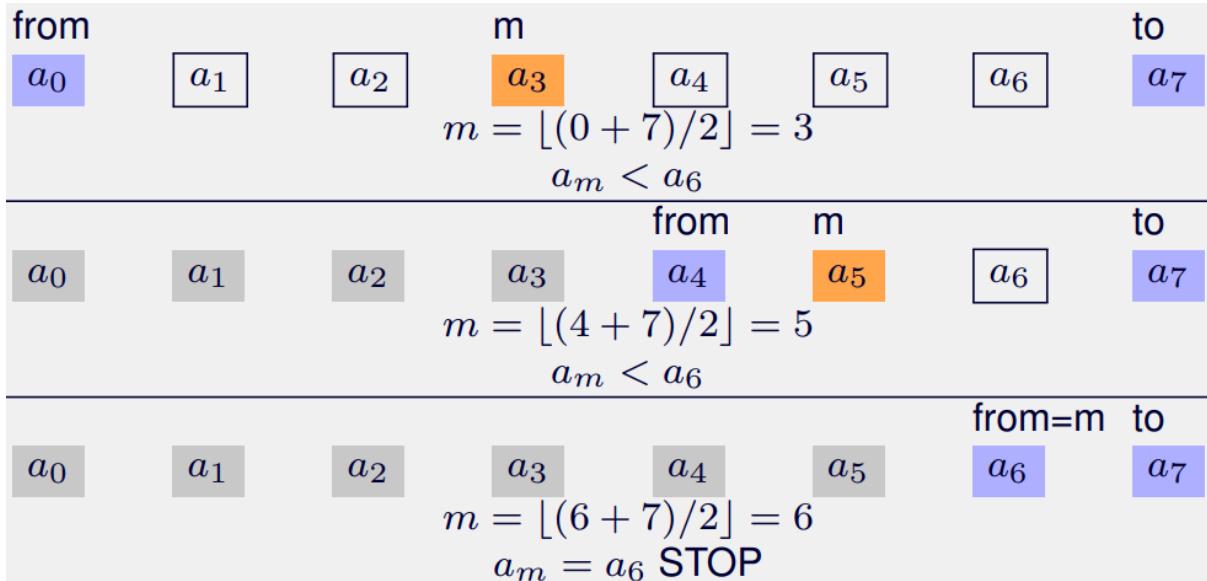
Ad ogni iterazione sceglieremo un elemento utile, detto **pivot** (x_m), contenuto fra *from* e *to*, verifichiamo se il pivot è l'elemento da ricercare, se sì terminiamo altrimenti restringiamo il campo:

- *less(elem, x_m)*: poniamo *to* = *m* - 1;
- $\neg less(elem, x_m)$: poniamo *from* = *m* + 1.

Dobbiamo anche considerare che, se l'elemento desiderato non è contenuto nell'array, la riduzione della porzione da esaminare si ridurrà all'insieme vuoto.

E per scegliere un elemento utile nella porzione *from..to* conviene usare la seguente formula: $(from + to)/2 = from/2 + to/2$.

Visualizzazione del procedimento



L'algoritmo in pseudocodice è il seguente:

```

1: function BINARY-SEARCH(array, from, to, elem)
2:   while from  $\leq$  to do
3:     m  $\leftarrow$   $\lfloor (\text{from} + \text{to})/2 \rfloor$ 
4:     if equal(array[m], elem) then return m
5:     else if less(array[m], elem) then
6:       from = m + 1
7:     else
8:       to = m - 1
return -1

```

L'analisi della complessità denota che nel caso generale $T(n) = k\theta(1)$ e nel migliore $T_{best}(n) = \Theta(1)$. Il caso peggiore è quando l'elemento non viene trovato quindi $T_{worst} = \log_2(n)\theta(1) = \Theta(\log_2(n))$ con una crescita logaritmica.

Per interpolazione / Interpolation search

La ricerca per interpolazione è una variante migliorata della ricerca binaria, che si basa sull'interpolazione [*metodo per individuare nuovi punti del piano cartesiano a partire da set finito di punti dati*].

In questo algoritmo, per cercare di indovinare dove sarà l'elemento da ricercare, si usa la formula:

$$x = x_0 + \frac{(x_1 - x_0) \cdot (y - y_0)}{y_1 - y_0}$$

Dove:

- Gli indici da considerare sono quelli dell'intervallo $[x_0, x_1]$
- La formula offre una “stima” di dove è plausibile si possa trovare il target da trovare considerando:

- $(x_1 - x_0)$: la grandezza dell'intervallo considerato;
- $(y - y_0)$: la differenza tra il target da trovare e il più piccolo esaminato;
- $(y_1 - y_0)$: la differenza tra il più grande e il più piccolo nell'intervallo esaminato;

Esempio di esecuzione [caso sfortunato]

from					m			to
$a_0 = 15$	$a_1 = 15$	$a_2 = 15$	$a_3 = 15$	$a_4 = 15$	$a_5 = 15$	$a_6 = 16$	$a_7 = 20$	
					$m = 0 + \lfloor (7 - 0) \times (16 - 15) / (20 - 15) \rfloor = \lfloor 7/5 \rfloor = 1$			
from								to
$a_0 = 15$	$a_1 = 15$	$a_2 = 15$	$a_3 = 15$	$a_4 = 15$	$a_5 = 15$	$a_6 = 16$	$a_7 = 20$	
					$m = 2 + \lfloor (7 - 2) \times (16 - 15) / (20 - 15) \rfloor = 2 + \lfloor 5/5 \rfloor = 3$			
from								to
$a_0 = 15$	$a_1 = 15$	$a_2 = 15$	$a_3 = 15$	$a_4 = 15$	$a_5 = 15$	$a_6 = 16$	$a_7 = 20$	
					$m = 4 + \lfloor (7 - 4) \times (16 - 15) / (20 - 15) \rfloor = 4 + \lfloor 3/5 \rfloor = 4$			
from								to
$a_0 = 15$	$a_1 = 15$	$a_2 = 15$	$a_3 = 15$	$a_4 = 15$	$a_5 = 15$	$a_6 = 16$	$a_7 = 20$	
					$m = 5 + \lfloor (7 - 5) \times (16 - 15) / (20 - 15) \rfloor = 5 + \lfloor 2/5 \rfloor = 5$			
from								to
$a_0 = 15$	$a_1 = 15$	$a_2 = 15$	$a_3 = 15$	$a_4 = 15$	$a_5 = 15$	$a_6 = 16$	$a_7 = 20$	

L'algoritmo in pseudocodice è il seguente:

```

1: function INTERPOLATION-SEARCH(array, from, to, elem)
2:    $x_0 \leftarrow from ; x_1 \leftarrow to ; y \leftarrow elem$ 
3:   while  $x_0 \leq x_1$  do
4:      $y_0 \leftarrow array[x_0] ; y_1 \leftarrow array[x_1]$ 
5:     if  $y_1 - y_0 = 0$  then
6:       if equal(array[ $x_0$ ], y) then return  $x_0$ 
7:       else return  $-1$ 
8:      $x \leftarrow x_0 + \lfloor (x_1 - x_0) \times (y - y_0) / (y_1 - y_0) \rfloor$                                  $\triangleright$  Interpolazione
9:     if  $x > x_1 \vee x < x_0$  then return  $-1$ 
10:    if equal(array( $x$ ), y) then return  $x$                                           $\triangleright$  N.B.: potrebbe non essere il più piccolo
11:    if less(array[ $x$ ], y) then                                                  $\triangleright$  Restringiamo la porzione da esaminare
12:       $x_0 = x + 1$ 
13:    else
14:       $x_1 = x - 1$ 
return  $-1$ 

```

Per quanto riguarda l'analisi della complessità:

- **Caso migliore:** $O(1)$
- **Caso medio:** $O(\log \log n)$
- **Caso peggiore:** $O(n)$

Strutture dati di base

Array dinamici

Si superano i problemi degli array classici come l'impossibilità di modificarlo a run-time o operazioni simili che hanno complessità $\Theta(n)$.

L'allocazione in memoria può essere:

- **Statica**: il ciclo di vita del programma e dell'oggetto è uguale (creati/distrutti insieme);
- **Automatica**: l'oggetto viene de/allocato automaticamente all'uscita/entrata di un contesto;
- **Dinamica**: l'oggetto viene allocato e deallocated su richiesta (malloc, free).

In maniera simile una **struttura dati** è *statica* se ha dimensione fissa o *dinamica* se varia a run time.

Recap C:

In `<stdlib.h>` sono definite le seguenti funzioni:

- `void *malloc(size_t sz)`: Alloca un blocco di memoria di `sz` byte e restituisce il puntatore al blocco allocato (o `NULL` in caso di fallimento)
- `void *free(void *ptr)`: Dealloca un blocco di memoria precedentemente allocato con `re/malloc`
 - `free(NULL)`: non fa nulla
 - **double free**: comportamento indefinito
- `void *realloc(void *ptr, size_t new_sz)`: Rialloca l'area di memoria indicata (precedentemente allocata con `malloc/realloc` e non liberata)
 - In caso di fallimento (ad es. non c'è abbastanza memoria libera), ritorna `NULL`
 - Può **espandere/contrarre** il blocco esistente ($\Theta(1)$) o **allocare un nuovo blocco** (copiando e liberando il precedente) ($\Theta(n)$): la parte eccedente ha contenuto indefinito

In linguaggi nuovi come Python gli array dinamici sono già presenti sotto forma di liste, in C questo non è vero.

Per realizzarle abbiamo bisogno di funzioni per gestire dinamicamente la memoria, la cosiddetta **riallocazione**, e tenere traccia della dimensione corrente.

La riallocazione è un'operazione costosa ($\Theta(n)$) e quindi evitiamo di farla ogni cambio di dimensione, a questo punto distinguiamo tra:

- **Capacità**: numero max di elementi contenuti;
- **Dimensione/Lunghezza**: numero di elementi correnti all'interno della struttura dati.

```
1 typedef int TInfo;
2
3 struct SDArray {
4     TInfo* item;
5     int capacity; // memory capacity
6     int size;      // the dynamic array has size elements in it
7 };
8
9 typedef struct SDArray DArray;
```

Per lavorare sulle due variabili distinguiamo fra:

- **Ridimensionamento:** lavora sulla size;
- **Riallocazione:** lavora sulla capacity, operazione che viene fatta meno di frequente e solo quando la size è piena.

Se abbiamo una **espansione** vuol dire che: $size < newSize$, ma c'è bisogno di ridimensionare con riallocazione solo se: $capacity < newSize$. Con una **contrazione** invece abbiamo che: $newSize < size$ e la riallocazione è necessaria.

Ora però dobbiamo capire quanta capacità supplementare va allocata o tollerata in queste due fasi.

Ridimensionamento con espansione lineare

Distinguiamo in riallocazione e contrazione:

- in caso di necessità di **riallocazione** ($capacity < newSize$), riserviamo un numero fisso di elementi in più Δ_{grow} rispetto alla dimensione richiesta:

$$capacity < newSize \Rightarrow newCapacity \leftarrow newSize + \Delta_{grow}$$

- in caso di contrazione, riduciamo la capacità solo se la differenza tra la capacità attuale e la nuova lunghezza richiesta supera una soglia Δ_{shrink} :

$$capacity - newSize > \Delta_{shrink} \Rightarrow newCapacity \leftarrow newSize + \Delta_{grow}$$

Ricordarsi di controllare che $\Delta_{grow} \leq \Delta_{shrink}$

Esempi con $\Delta_{grow} = 5$, $\Delta_{shrink} = 10$, $size = 0$, $capacity = 10$:

- `resize(da, 9)` \Rightarrow $size \leftarrow 9$
- `resize(da, 12)` \Rightarrow $size \leftarrow 12$ $new_capacity \leftarrow 12 + 5 = 17$
- `resize(da, 10)` \Rightarrow $size \leftarrow 10$
- `resize(da, 6)` \Rightarrow $size \leftarrow 6$ $new_capacity \leftarrow 6 + 5 = 11$

Ridimensionamento con espansione geometrica

In questo caso invece di usare una capacità supplementare fissa Δ_{grow} si usa una capacità supplementare che è proporzionale alla lunghezza richiesta $newSize$ di un fattore $\phi_{grow} > 1$.

In caso di contrazione riduciamo la capacità quando: $(capacity/newSize) > \phi_{shrink} > 1$, quindi i possibili frangenti sono:

1. Se $capacity < newSize$ allora $newCapacity \leftarrow \phi_{grow} \cdot newSize$
2. Se $(capacity/newSize) > \phi_{shrink}$ e con $\phi_{shrink} \geq \phi_{grow}$ allora
 $newCapacity \leftarrow \phi_{grow} \cdot newSize$
3. nessuna riallocazione

Esempi con $\phi_{grow} = 2.5$, $\phi_{shrink} = 3$, $size = 0$, $capacity = 10$:

- `resize(da, 9)` $\implies size \leftarrow 9$
- `resize(da, 12)` $\implies size \leftarrow 12$ $new_capacity \leftarrow 2.5 \cdot 12 = 30$
- `resize(da, 10)` $(30/10) = 3 > 3 \implies size \leftarrow 10$
- `resize(da, 6)` $(30/6) = 5 > 3 \implies size \leftarrow 6$ $new_capacity \leftarrow 2.5 \cdot 6 = 15$

Complessità

Complessità spaziale

- La quantità di memoria allocata è pari a $capacity \times sizeof(TInfo)$
- Poiché sappiamo $capacity > size = n$, dobbiamo stabilire un limite superiore per c come funzione di n per poter definire la **complessità spaziale**
- **Espansione lineare:** $capacity \leq n + \Delta_{grow}$, quindi
 $S(n) = \Theta(c) = \Theta(n + \Delta_{grow}) = \Theta(n)$
- **Espansione geometrica:** $capacity \leq \phi_{grow} \cdot n$, quindi
 $S(n) = \Theta(c) = \Theta(\phi_{grow} \cdot n) = \Theta(n)$

Complessità temporale del ridimensionamento con espansione lineare

- Si consideri una singola applicazione di `resize(da, n)`
 - **Caso migliore:** non occorre nessuna riallocazione, ergo $\Theta(1)$
 - **Caso peggiore:** riallocare un array lungo n ha costo $\Theta(n)$
 - Contributi: (1) allocazione nuovo blocco di memoria, $\Theta(1)$; (2) copia del contenuto dal vecchio al nuovo blocco, $\Theta(n)$; e (3) deallocazione vecchio blocco, $\Theta(1)$
- Per una **successione di append di 1 elemento fino ad aver lunghezza n**
 - Poiché quando $capacity < new_size$ si effettua $new_capacity \leftarrow new_size + \Delta_{grow}$, si ha **una riallocazione (di costo lineare) ogni Δ_{grow} ridim. unitari**
 - Per arrivare a dim. n abbiamo **n ridim. unitari** con $k = n/\Delta_{grow}$ riallocazioni.
Il costo della i -esima riallocazione è $\Theta((i-1)\Delta_{grow})$
 - $T_n \text{ append}(n) = \sum_{i=1}^k \Theta((i-1) \cdot \Delta_{grow}) = \Theta(\frac{k(k-1)}{2} \Delta_{grow}) = \Theta(\frac{n(n-1)}{2\Delta_{grow}}) = \Theta(n^2)$
 - 💡 Nota: la complessità non differisce da un algoritmo di ridimensionamento che effettui ogni volta la riallocazione; tuttavia, la costante di proporzionalità viene divisa per Δ_{grow} e quindi il tempo effettivo può essere notevolmente minore
 - **Complessità ammortizzata** (su n ridimensionamenti): $\Theta(n^2)/n = \Theta(n)$

Complessità temporale del ridimensionamento con espansione geometrica

- Ad ogni riallocazione la capacità precedente viene moltiplicata per ϕ_{grow}
- Supponendo all'inizio $capacity = 1$, dopo k riallocazioni abbiamo ϕ_{grow}^k elementi
- Per arrivare ad n occorrono $k = \lceil \log_{\phi_{grow}}(n) \rceil$ riallocazioni
- Il costo della i -esima riallocazione è $\Theta(\phi_{grow}^{i-1})$
- Degli n ridimensionamenti:
 - k comporteranno una riallocazione con costo totale di
 $\sum_{i=1}^k \Theta(\phi_{grow}^{i-1}) = \Theta((\phi_{grow}^k - 1)/(\phi_{grow} - 1)) = \Theta(n)$
 - Poiché $\phi_{grow}^k \approx n$ e $\sum_{i=0}^n q^i = (q^{n+1} - 1)/(q - 1)$
 - E gli altri $n - k$ con costo totale $(n - k)\Theta(1) = \Theta(n)$
 - Dunque abbiamo $T_n \text{ append} = \Theta(n) + \Theta(n) = \Theta(n)$
 - Molto meglio rispetto all'espansione lineare!
- **Complessità ammortizzata** (su n ridimensionamenti): $\Theta(n)/n = \Theta(1)$

Pile / Stack

Struttura dati con accesso di tipo LIFO, l'ultimo elemento che entra è il primo ad uscire. Le operazioni possibili sono:

- *create* e *destroy*;
- *push*: metti in cima;
- *pop*: togli dalla cima;
- *top*: vedi cosa c'è in cima;
- *isempty* e *isfull*: controllo vuota/piena.

Per implementare una pila è sufficiente tenere traccia del numero di elementi presenti e da questo dato si ricava anche la posizione di inserimento o prelievo. L'implementazione risulta facile tramite array (statico o dinamico):

- **push**
 1. $a_n \leftarrow x$ (dove x è l'elemento da inserire)
 2. $n \leftarrow n + 1$
- **pop**
 1. $x \leftarrow a_{n-1}$ (dove x è l'elemento prelevato)
 2. $n \leftarrow n - 1$
- **top**: $x \leftarrow a_{n-1}$
- **isempty**: $n = ?$

Dove n tiene traccia del numero di elementi e allo stesso tempo della prossima posizione di inserimento

Coda / Queue

Struttura dati con modalità di accesso FIFO, il primo elemento che entra è il primo ad uscire.

Utile se vogliamo processare una sequenza di richieste in base all'ordine di arrivo e con un'attesa non infinita, possiamo usare i **buffer** se o il tasso di arrivo di richieste (producer speed) non corrisponde alla capacità di soddisfarle (consumer speed) o il tasso di arrivo di richieste (producer speed) non corrisponde alla capacità di soddisfarle (consumer speed).

Le operazioni possibili sono:

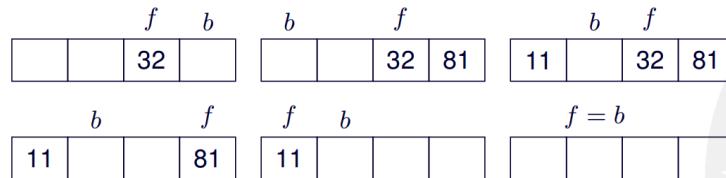
- *create* e *destroy*;
- *add* o *enqueue*: metti in coda o accodamento;
- *remove* o *dequeue*: togli dalla coda;
- *front*: vedi cosa c'è in cima (1° elemento);
- *isempty* e *isfull*: controllo vuota/piena.

Per implementare una coda è sufficiente tenere traccia del numero di elementi presenti e da questo dato si ricava anche la posizione di inserimento o prelievo.

Tenendo traccia della testa (f), il prelievo si attua semplicemente con lo spostamento in avanti di f .

Si può sempre usare un array statico o dinamico e per evitare problemi di complessità la soluzione è implementare una **coda circolare** dove l'array viene interpretato come struttura **ciclica** (dove la posizione successiva all'ultima è quella di indice 0) e per incrementare gli indici è comodo usare la capacity:

Enqueue: $b = (b + 1) \% \text{capacity}$ | Dequeue: $f = (f + 1) \% \text{capacity}$



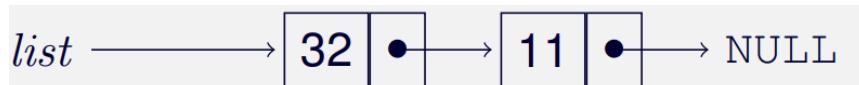
Liste dinamiche

Utili per rappresentare collezioni di elementi organizzati linearmente e per lavorare con collezioni che consentano di svolgere in modo efficiente le operazioni di ricerca, inserimento, e cancellazione.

Def | Lista: struttura dati che immagazzina un insieme di elementi in ordine

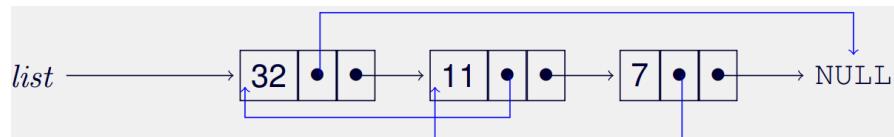
Esistono diversi tipi di liste:

- **Linked list:** formata da nodi, ogni nodo contiene l'informazione e il riferimento all'elemento successivo.

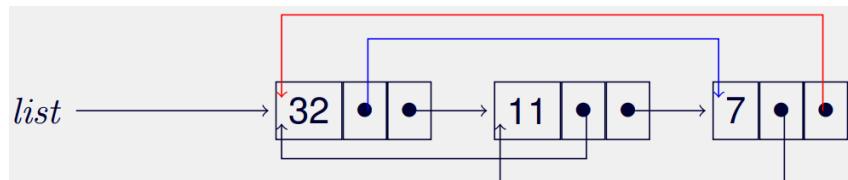


L'ordine degli elementi è dato dai collegamenti esistenti fra gli elementi e non dal piazzamento in memoria, il primo elemento è la *testa* e l'ultimo è la *coda*.

- **Doubly-linked list:** ogni nodo contiene anche un collegamento all'elemento precedente; tipicamente sono noti sia il riferimento alla testa sia il riferimento alla coda.



- **Circular list:** una linked list dove l'elemento in coda è collegato all'elemento in testa, il collegamento può essere singolo o doppio.



Possiamo distinguere tre nozioni di ordinamento nelle liste:

1. **Ordine strutturale:** l'ordine risultante dai collegamenti fra i nodi;
2. **Ordine logico:** l'ordine esistente tra gli elementi sulla base di una relazione d'ordine;
3. **Ordine fisico:** l'ordine degli elementi in memoria.

Def | Lista ordinata: una lista dove l'ordine strutturale corrisponde all'ordine logico

Implementazione

L'implementazione può avvenire su array statici o dinamici dove ogni elemento dell'array è un nodo della lista e il collegamento è dato dall'indice dell'elemento successivo. Con gli array statici c'è il problema di quanta memoria allocare e nei dinamici della rimozione dei nodi e della frammentazione della memoria contigua associata all'array dinamico.

Normalmente l'implementazione avviene tramite allocazione dinamica dei nodi, si definisce la lista come struttura dati ricorsivamente definita:

```
1 typedef struct list {
2     TInfo val;
3     struct list *next;
4 } list;
```

Una *list* può essere **NULL**, se è vuota, oppure un nodo che contiene un **value** e il collegamento **next** alla sottolista rimanente:

```
1 list *list_create(TInfo val, list *t) {
2     list *r = (list *)malloc(sizeof(list));
3     if(r == NULL) return NULL;
4     r->val = val;
5     r->next = t;
6     return r;
7 }
```

Creazione della lista: step-by-step

```
1 struct list *L = list_create(2017, NULL); // [2017,NULL]
```



```
1 L = list_create(5, L); // lista [5,[2017,NULL]]
```



```
1 L = list_create(15, L); // lista [15,[5,[2017,NULL]]]
```



💡 Si noti come la costruzione porti ad aggiungere elementi **in testa** alla lista

Stessa espressione in una sola riga

```
1 struct list *L = list_create(15, list_create(5, list_create(2017, NULL)));
2 // l'ordine delle chiamate e' da quella piu' interna
```

Ora vediamo una carrellata di operazioni utili per lavorare con le liste:

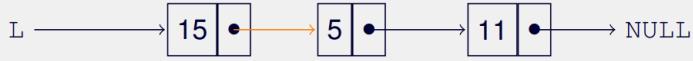
Accesso agli elementi

```
1 struct list *L = list_create(15, list_create(5, list_create(2017, NULL)));
2 printf("%d\n", L->next->val); // stampa il secondo elemento di L
3 L->next->next->val = 11; // modifica del terzo elemento => [15,5,11]
```



Aggiunta elemento

```
1 L->next = list_create(25, L->next); // aggiunta elemento => [15,25,5,11]
```



DISTRUZIONE

```
1 void list_destroy(list *L) {
2     if (L != NULL) {
3         list_destroy(L->next);
4         L->next = NULL; /* non necessario... */
5         free(L);
6     }
7 }
```

Stampa

```
1 void list_prinr(list *L) {
2     if(L == NULL) {
3         printf("()");
4         return;
5     }
6     printf("(");
7     printf("%d,", L->val);
8     list_prinr(L->next);
9     printf(")");
10 } // time complexity: Θ(n)
```

Controllo uguaglianza

```
1 int list_equal(list *L1, list *L2) {
2     if (L1 == NULL || L2 == NULL) {
3         return L1 == NULL && L2 == NULL;
4     } else {
5         return L1->val == L2->val && list_equal(L1->next, L2->next);
6     }
7 }
```

Calcolo lunghezza

```
1 int list_length(list *L) {
2     if (NULL == L) {
3         return 0;
4     } else {
5         return 1 + list_length(L->next);
6     }
7 }
```

Inserimento (in posizione arbitraria)

```
1 list *list_insert(list *L, int pos, TInfo newvalue) {
2     if(pos == 0) { return list_create(newvalue, L); }
3     list *curr = L;
4     int i = 0;
5     for(; curr != NULL && i < pos - 1; i++, curr = curr->next);
6     if(i == pos - 1 && curr != NULL) {
7         curr->next = list_create(newvalue, curr->next);
8     }
9     return L;
10 }
```

L'inserimento ha complessità $\Theta(1)$ nel caso migliore e $\Theta(n)$ nel peggiore/medio. Le stesse complessità valgono per rimozione e ricerca.

Liste vs Array dinamici

Operazione	Dynamic array	Linked list
Accesso ad inizio/fine	$\Theta(1)$	$\Theta(1)$
Accesso per posizione	$\Theta(1)$	$\Theta(n)$
Inserimento in testa	$\Theta(n)$	$\Theta(1)$
Inserimento in coda	$\Theta_{\text{amort}}(1), \Theta_w(n)$	$\Theta(1)$
Inserimento in posizione	$\Theta(n)$	$\Theta(n)$

Tabelle hash

Create per supportare ricerche con complessità $\Theta(1)$, ecco la definizione:

Una tabella (anche chiamata mappa o dizionario o array associativo) è un insieme di coppie chiave-valore (elementi) $E_i = (K_i, V_i)$. Le chiavi sono prese da un insieme U (universo delle chiavi), gli elementi da un insieme V .

In questo caso la chiave identifica un certo elemento, ma esistono strutture dati dove ad ogni chiave può corrispondere più di un elemento, la **multi-mappa**.

Operazioni possibili:

- $\text{insert}(key, elem)$: inserimento della coppia (k,e) alla mappa;
- $\text{get}(k)$: restituzione dell'elemento associato alla chiave k;
- $\text{delete}(k)$: rimozione elemento associato alla chiave.

A indirizzamento diretto

Si fa uso di un **vettore** e usa **chiavi numeriche** il cui valore è da interpretarsi come indice del vettore.

Implementazione: array $a = [a_0, a_1, \dots, a_{m-1}]$ di m elementi, dove all'indice i corrisponde la coppia (i, V_i)

- **Terminologia:** gli slot dell'array sono spesso chiamati **bucket**

0	?
1	?
...	...
$m - 1$?

- $\text{insert } (\mathbf{k}, \mathbf{e}) : a[k] \leftarrow e$
 - $\text{get } (\mathbf{k}) : x \leftarrow a[k]$
 - $\text{delete } (\mathbf{k}) : a[k] \leftarrow \text{null}$
- 💡 Tutte $\Theta(1)$ in quanto l'accesso per indice in un array ha complessità costante

Fattore di carico: $n = n/m$ per n chiavi/elementi memorizzati, un esempio:

Se usiamo un codice da 4 cifre possiamo usare un array di 10000 elementi e usare il codice come chiave.

Avendo 500 elementi, il fattore di carico è: $n = 500/10000 = 0.05 = 5\%$

A indirizzamento indiretto

Riduce l'occupazione di memoria avendo comunque un accesso efficiente. La strategia è usare la funzione di hash $h: U \rightarrow [0, 1, \dots, m - 1]$ che fa corrispondere ad ogni chiave k appartenente a U la posizione nell'array in cui l'informazione associata è memorizzata.

La dimensione m può non coincidere con $|U|$.

Quando due chiavi (k_1 e k_2) hanno lo stesso valore hash ($h(k_1) = h(k_2)$) si verifica una **collisione**.

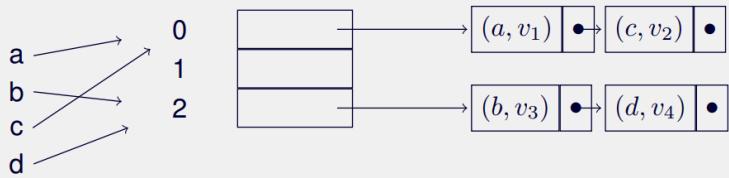
Se una funzione (h) non causa collisioni, cioè è iniettiva, si chiama **hash perfetto**.

$$\forall k_1, k_2 \in U, k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

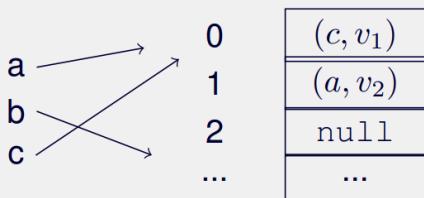
Si dice **hash perfetto minimale** un hash (h) con immagine da 0 a $(|U| - 1)$.

Per la gestione delle collisioni esistono due metodi:

- **Liste di collisione** (aka. **chaining** o **indirizzamento chiuso** → **chained hashtable**): gli elementi collidenti sono contenuti in **liste esterne alla tabella**



- **Indirizzamento aperto** (aka **linear probing**): tutti gli elementi sono contenuti nella tabella; eventuali collisioni sono gestite con l'**occupazione di posizioni successive**



Una buona funzione di hash deve essere facile da calcolare e distribuire in modo uniforme le chiavi sulle posizioni della tabella.

Con la distribuzione uniforme le liste di collisione devono avere lunghezza $n/m = 2$, con complessità di accesso worst-case: $\Theta(n/m)$; senza si avrebbe $\Theta(l_{max})$ con l_{max} = lunghezza della lista di collisione più lunga.

Ogni chiave, secondo l'**uniformità semplice**, deve avere la stessa probabilità di vedersi assegnata una qualsiasi posizione ammissibile indipendentemente da altri valori hash già assegnati; il requisito di uniformità semplice è difficile da verificare.

Funzioni di hash

Il **metodo della divisione** è semplice e veloce, la formula è la seguente:

$$h(k) = k \cdot mod \cdot m$$

Conviene evitare certi valori di m , come le potenze di 2, occorre rendere h dipendente da tutti i bit della chiave per questo m dovrebbe essere un numero primo non troppo vicino ad una potenza del due.

Il **metodo della moltiplicazione** ha la seguente formula:

$$h(k) = \lfloor m \cdot (k \cdot A - [k \cdot A]) \rfloor$$

Per sapere cosa vuol dire il simbolo: $\lfloor \rfloor$

Consiste in due passi:

1. si moltiplica la chiave k per una certa costante A , $0 < A < 1$, estraendo la parte frazionaria $k \cdot A$;
2. moltiplico la parte frazionaria per m e prendo la parte intera inferiore del risultato.

$k \cdot A - [k \cdot A]$ è la parte frazionaria di $k \cdot A$ (si suggerisce: $A \approx (\sqrt{5} - 1)/2$)

In questo metodo m non è critico.

Implementazione hashtable

Chained hashtable

Una **chained hashtable** (aka separate chaining o closed addressing hash table) è una impl. di tabelle ad indirizzamento indiretto che usa liste di collisione.

Per implementarli si usa un vettore di bucket, dove ogni bucket vi è una linked list:

Definizioni preliminari

```
1 typedef int TKey;
2 typedef int TValue;
3 typedef struct TInfo { TKey key; TValue value; } TInfo;
4 int equal(TInfo a, TInfo b) { return a.key == b.key; } // NB: only key
5 // ...
6 typedef struct list { TInfo val; struct list *next; } list;
```

Struttura dati + interfaccia di programmazione

```
1 typedef struct HashTable {
2     list** bucket;
3     int nbuckets;
4 } HashTable;
5 // ...
6 HashTable *hashtable_create(int buckets);
7 void hashtable_insert(HashTable*, TKey key, TValue val);
8 void hashtable_delete(HashTable*, TKey key);
9 TValue *hashtable_search(HashTable*, TKey key);
```

Esempi di implementazioni: creazione e distruzione

```
1 HashTable *hashtable_create(int nbuckets) {
2     HashTable *h = (HashTable*) malloc(sizeof(HashTable));
3     if(h == NULL) return NULL;
4     h->bucket = (list**) malloc(sizeof(list*) * nbuckets);
5     if(h->bucket == NULL) { free(h); return NULL; }
6     h->nbuckets = nbuckets;
7     for(int i = 0; i < nbuckets; i++) {
8         h->bucket[i] = NULL;
9     }
10    return h;
11 }
12
13 void hashtable_destroy(HashTable* h) {
14     if(h == NULL) return;
15     for(int i = 0; i < h->nbuckets; i++) {
16         list_destroy(h->bucket[i]);
17     }
18     free(h->bucket);
19     h->nbuckets = 0;
20     free(h);
21 }
```

Esempi di implementazioni: ricerca

```
1 TValue *hashtable_search(HashTable* h, TKey key) {
2     list* l = h->bucket[hashtable_hash(h, key)];
3     for(; l != NULL; l = l->next) {
4         if(l->val.key == key) { return &l->val.value; }
5     }
6     return NULL;
7 }
```

Per quanto riguarda la complessità tutte le operazioni sono $\Theta(1)$ tranne la *list_search* con complessità $O(k)$ (k sta per la lunghezza della lista di collisione).

Se la funzione di hash produce una distribuzione uniforme delle chiave, si avrà una lunghezza media della lista n/m e avremo $\Theta_{avg}(n) = \Theta(n/m)$ (n : numero elementi in tabella, m : numero bucket).

Con un numero di bucket proporzionale al massimo numero di elementi previsti:

$$\Theta_{avg}(n) = \Theta(n/m) = \Theta(n/(cn)) = \Theta(c) = \Theta(1).$$

Esempi di implementazioni: inserimento e rimozione

```

1 void hashtable_insert(HashTable* h, TKey key, TValue val) {
2     TInfo info = { key = key, val = val };
3     unsigned int hash = hashtable_hash(h, key);
4     if(!hashtable_search(h, key)) {
5         h->bucket[hash] = list_create(info, h->bucket[hash]);
6     }
7 }
8
9 void hashtable_delete(HashTable* ht, TKey key) {
10    unsigned int h = hashtable_hash(ht, key);
11    TInfo ikey = { key = key };
12    ht->bucket[h] = list_delete_value(ht->bucket[h], ikey);
13 }
```

Anche qui abbiamo $\Theta_{avg}(n/m)$ e $\Theta_{worst}(n)$ della ricerca.

Esempi di implementazioni: visita

```

1 void hashtable_print(HashTable* h, int include_empty_buckets) {
2     printf("{");
3     for(int i = 0; i < h->nbuckets; i++) {
4         if(include_empty_buckets || h->bucket[i] != NULL) {
5             printf("\t[%d] ", i);
6             list_print(h->bucket[i]);
7             printf("\n");
8         }
9     }
10    printf("}\n");
11 }
```

La complessità per $T_{avg} = T_{worst} = \Theta(n)$, ma se $m \gg n$ allora avremo; $T_{worst} = \Theta(m + n)$

Ora vediamo diverse funzioni di hash per diversi tipi:

Funzione di hash per interi

```
1 unsigned int hash_int(TKey k) { return (unsigned int) k; }
```

Funzione di hash per stringhe

```

1 unsigned hash_str(TKey key) {
2     unsigned int h = 0;
3     for(int i = 0; key[i] != '\0'; i++) {
4         h = h * 33 + key[i];
5     }
6     return h;
7 }
```

Per tipi arbitrari conviene interpretare la struttura dati come sequenza di byte, e di applicare un algoritmo simile a quello mostrato per le stringhe; **però** per valori concettualmente

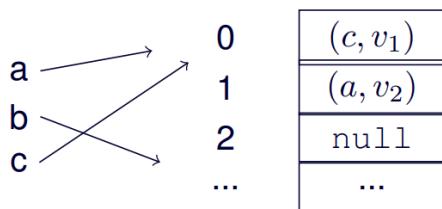
equivalenti occorre produrre lo stesso hashcode e si interpreti tutto il contenuto della struttura dati: se questa usa puntatori, si consideri il contenuto dell'area puntata (e non il puntatore).

Probing

Una **probing / open addressing hashtable** è una implementazione di tabelle ad indirizzamento indiretto che memorizza le coppie chiave-valore direttamente nei bucket dell'array sottostante e usa la tecnica dell'ispezione/probing lineare per risolvere le collisioni.

Idea: se si vuole aggiungere una chiave k nel bucket $h(k)$, ma in questo vi è già una chiave k' (ovvero è $h(k) = h(k')$), allora vengono esaminati in successione i bucket $h(k) + 1, h(k) + 2, \dots$ fino a quando non viene trovato un bucket libero

- se si raggiunge la fine del vettore, si ricomincia dall'inizio



Quando $n \approx m$ il probing causa collisioni aggiuntive e si raccomanda che il fattore di carico sia sotto una soglia di carico λ : $n/m < \lambda < 1$, se si supera λ la tabella va ridimensionata e ricalcolare l'hash di tutti gli elementi (*re-hasing*).

Set

È una struttura dati che memorizza una collezione di elementi distinti; quindi senza duplicati e senza un ordine particolare.

Sono mutabili e ogni elemento del set deve essere hashable.

[RECAP]

Caso peggiore

	Ricerca	Accesso posiz.	Visita	Inserim.	Rimoz.
Array	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Array ordinati	$O(\log n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Liste dinamiche	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Tabelle hash	$O(n)$	N/A	$O(m+n)$	$O(n)$	$O(n)$

Caso medio

	Ricerca	Accesso posiz.	Visita	Inserim.	Rimoz.
Array	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$\Theta(\log n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Liste dinamiche	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Tabelle hash	$\Theta(1)$	N/A	$O(n)$	$\Theta(1)$	$\Theta(1)$

Algoritmi di ordinamento

Def | Sorting: Data una sequenza x_1, x_2, \dots, x_n , l'ordinamento di tale sequenza consiste nel determinare una sua permutazione x'_1, x'_2, \dots, x'_n tale che $x'_1 \leq x'_2 \leq \dots \leq x'_n$

L'ordinamento può essere **interno**, la struttura è interamente contenuta in memoria centrale, **esterno**, la struttura è memorizzata in memoria secondaria.

Un'ulteriore suddivisione è:

- **ord. sul posto:** detto anche *in-place*, l'output consiste in una modifica della struttura in input (complessità spaziale di strutture dati aggiuntive al $\max O(\log(n))$);
- **ord. fuori posto:** detto anche *out-of-place*, si produce una nuova struttura in output (complessità spaziale strutture dati aggiuntive $\Omega(n)$).

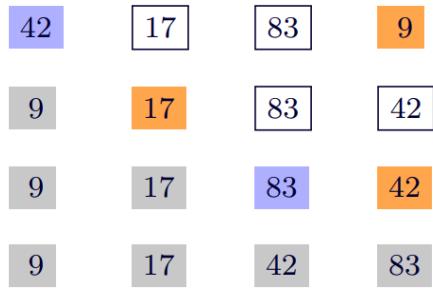
L'ordinamento stabile è : quando nella sequenza finale gli elementi equivalenti mantengono lo stesso ordine relativo.

L'ordinamento per confronti: l'ordinamento è basato sul confronto di coppie di elementi.

Selection sort

Si cerca il minimo nell'array da ordinare, e si scambia con quello alla prima posizione; si seleziona il minimo nell'array restante, e si scambia con quello alla seconda posizione; e così via.

Bastano $n - 1$ iterazioni, l'ultimo sarà ordinato per forza, vediamo un esempio:



Notazione

- Blu: fronte del sotto-array da ispezionare
- Arancione: elemento a valore minimo nel sotto-array
- Grigio: parte già ordinata

In pseudo codice:

```

1: function SELECTION-SORT(array)
2:   for i  $\leftarrow 0..(n - 2)$  do
3:     minId  $\leftarrow \text{minIndex}(\text{array}[i..(n - 1)])           ▷ Returns index of min element in subarray
4:     if minId  $\neq i$  then swap(minId, i)               ▷ swaps array[i] with array[minId]$ 
```

La complessità è $\Theta(n^2)$, ma vediamo perchè:

$$T(n) = \sum_{i=0}^{n-2} T^i(n)$$

dove $T^i(n)$ è la complessità di una singola iterazione in funzione della var. *i* del ciclo ed essendo che *minIndex* è $\Theta(n)$ e all'iterazione *i* lavora su $n - i$ elementi abbiamo:

$$T(n) = \sum_{i=0}^{n-2} \Theta(n - i) = \Theta\left(\sum_{i=0}^{n-2} \Theta(n - i)\right) = \Theta\left(\frac{(n-1)(n+2)}{2}\right) = \Theta(n^2)$$

Alcune osservazioni sul seguente algoritmo possono essere:

- Essendo che lo swap può invertire l'ordine relativo di elem. uguali, **non è stabile!**;
- La complessità calcolata precedentemente vale anche per il caso migliore.

Insertion sort

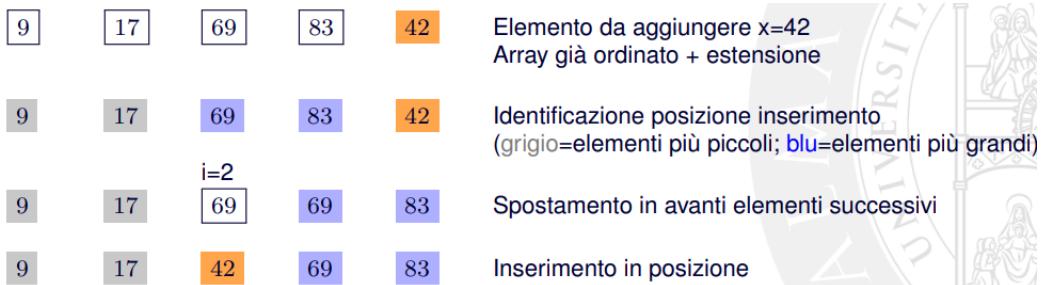
Data una sequenza a_0, \dots, a_{n-1} già ordinata e un nuovo valore x , si vuole ottenere una nuova sequenza ordinata di $n + 1$ elementi che contenga gli elementi iniziali e x .

Per semplicità dividiamo il problema in due parti

↓

1. *Si inizia riducendo il problema all'inserimento in ordine:*

Lo scopo è trovare la posizione dove mettere x , per farlo troviamo la posizione dove si trova il primo elemento più grande, poi spostiamo tutti i numeri da quella posizione a destra di uno e inseriamo il nuovo valore nel posto che si è liberato.

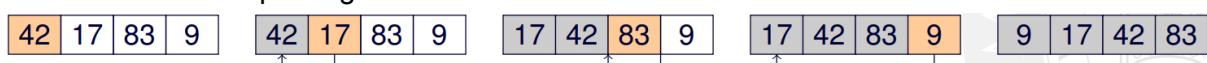


2. L'algoritmo in se invece lavora nel seguente modo:

Si tiene traccia di due porzioni dell'array:

1. la parte iniziale, che è già ordinata;
2. la parte finale, da ordinare.

Si fa un ciclo che itera sulla seconda parte ed ogni volta prende il prossimo elemento della parte non ordinata e tramite *l'inserimento in ordine* lo mette nella parte ordinata nel posto giusto.



```

1: function INSERT-IN-ORDER(array, n, x)    ▷ The array is 0-indexed and has n+1 buckets
2:   pos ← n    ▷ I.e. the array has room for the element to be inserted, possibly at index n
3:   while pos > 0 ∧ greater(array[pos - 1], x) do      ▷ ① ③ Find the position of insertion
4:     pos ← pos - 1
5:   for i ← (n - 1) downto pos do ▷ ② Shift to the right the greater elements (RtL iteration)
6:     array[i + 1] = array[i]
7:   array[pos] ← x                                ▷ Insertion
8: function INSERTION-SORT(array)
9:   for i ← 1..(n - 1) do                      ▷ Consider subsequent elements for insertion
10:    insert-in-order(array, i, array[i])

```

Un'altra implementazione utilizza due indici: uno punta all'**elemento da ordinare** e l'altro all'**elemento immediatamente precedente**. Se l'elemento puntato dal **secondo indice** è maggiore di quello a cui punta il **primo indice**, i due elementi vengono scambiati di posto; altrimenti il **primo indice** avanza. Il procedimento è ripetuto finché si trova nel punto in cui il valore del **primo indice** deve essere inserito. Il *primo indice* punta *inizialmente al secondo elemento dell'array*, il *secondo inizia dal primo*. L'algoritmo così tende a spostare man mano gli elementi maggiori verso destra.

```

function insertionSortIterativo(array A)
  for i ← 1 to length[A] do
    value ← A[i]
    j ← i-1
    while j >= 0 and A[j] > value do
      A[j + 1] ← A[j]
      j ← j-1
    A[j+1] ← value;

```

L'implementazione, dentro il *while*, sposta a destra di uno il valore che è in posizione *j* (maggiore dell'elemento da swappare) e poi diminuisce *j*, in questo modo sposta tutti fino a che non abbiamo un posto libero per la variabile da swappare (il ciclo si interrompe o se arriviamo in fondo o finchè non siamo arrivato in un punto dove il valore di sinistra non è maggiore).

Prima dello swap ci sarà un elemento ripetuto poi in quella posizione mettiamo l'elemento minore.

La complessità varia nei due algoritmi:

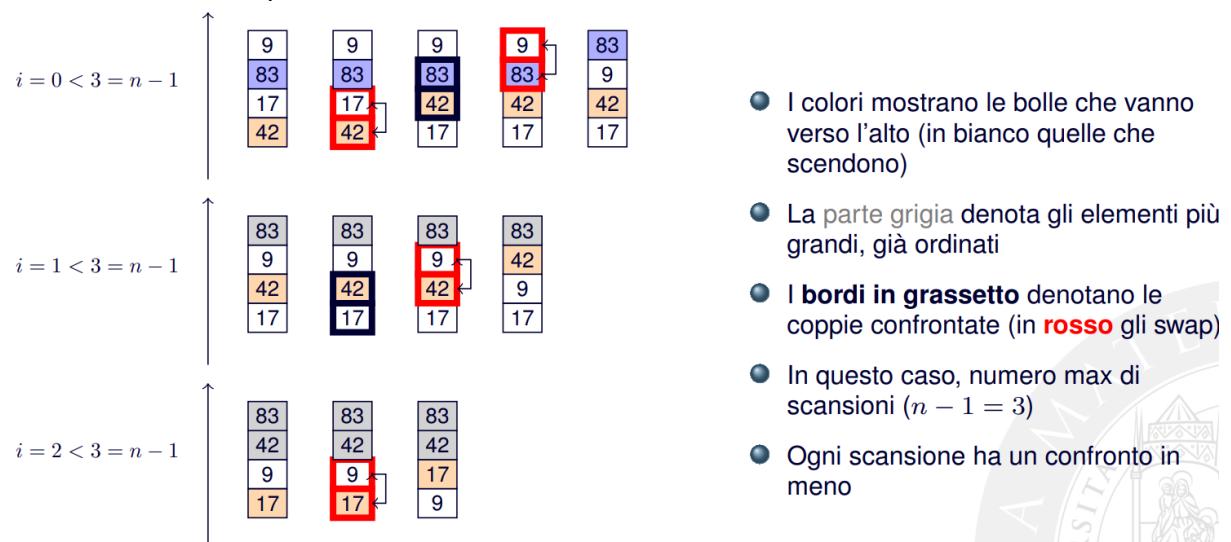
- *insert-in-order*: con p la posizione di inserimento
 - $T_{iio} = T_{while} + T_{for} = (n - p)\Theta(1) + (n - p)\Theta(1) = \Theta(n - p)$
 - **Best case** [$p = n$]: $\Theta(1)$;
 - **Worst case** [$p = 0$]: $\Theta(n)$.
- *insertion-sort*:
 - $T_{is} = (n - 1)T_{iio}$
 - **Best case**: $\Theta(n)$;
 - **Worst case**: $\Theta(n^2)$.

Bubble sort

L'idea è di spostare verso la fine dell'array gli elementi che hanno valore più alto di quelli adiacenti, si esaminano tutte le coppie adiacenti e se non sono localmente ordinati si scambiano di posto, lo scambio va registrato e si interrompe il ciclo solo quando non si registrano più scambi.

Si produrrà un array ordinato in al massimo $n - 1$ scansioni e per produrre un ordinamento stabile basta non scambiare i valori adiacenti con ugual valore.

Visualizzazione del procedimento:



Traducendolo in pseudo codice otteniamo:

```

1: function BUBBLE-SORT(array, n)
2:   swap  $\leftarrow 1$ 
3:   i  $\leftarrow 0$ 
4:   while swap = 1  $\wedge$  i < n - 1 do:                                 $\triangleright$  Need n - 1 iterations at most
5:     swap  $\leftarrow 0$ 
6:     for j  $\in$  [0..(n - i - 2)] do                                $\triangleright$  No need to consider the last i items
7:       if array[j] > array[j + 1] then
8:         SWAP(array[j], array[j + 1])
9:         swap  $\leftarrow 1$ 
10:    i  $\leftarrow i + 1$ 

```

Il ciclo interno produce una complessità di $\Theta(n - i)$, invece con il ciclo esterno eseguito k volte:

$$T(n) = \sum_{i=0}^{k-1} \Theta(n - 1) = \Theta\left(\sum_{i=0}^{k-1} n - \sum_{i=0}^{k-1} i\right) = \Theta\left(nk - \frac{k(k-1)}{2}\right) = \Theta\left(\frac{k(2n-k+1)}{2}\right)$$

Con il **caso migliore** (array ordinato e $k = 1$) abbiamo $\Theta(n)$ e con il **peggiore** ($k = n - 1$) abbiamo $\Theta(n^2)$.

Esiste la variante **bidirectional bubble sort** che alterna scansioni verso l'alto a scansioni verso il basso per evitare che rimangano elementi piccoli alla fine che rallentano l'algoritmo.

Merge sort

Riduzione ricorsiva del problema dell'ordinamento al problema della fusione di array ordinati.

Def | Fusione array ordinati: date due sequenze ordinate di n e m elementi rispettivamente, a_0, \dots, a_{n-1} e a_0, \dots, a_{m-1} , si vuole produrre una nuova sequenza ordinata di $n + m$ elementi, c_0, \dots, c_{n+m-1} , che contenga tutti gli elementi delle due sequenze di partenza.

Per funzionare si procede per decomposizione ricorsiva, quindi:

1. **divide**: dividiamo la sequenza da ordinare in due parti;
2. **impera**: ad ogni parte si applica ricorsivamente l'ordinamento;
3. **combine**: si fa il merge delle due parti.

Visualizzato:

<i>m</i>							
19	42	7	38	26	71	54	
19	42	7	38	26	71	54	
7	19	42	26	38	54	71	

7	19	26	38	42	54	71	
---	----	----	----	----	----	----	--

Divisione in due parti attorno al centro.

Ordinamento ricorsivo delle due parti.

Fusione delle due parti ordinate.

Lo pseudocodice derivante è:

Require: a and b are sorted ; r is of length $n_a + n_b$

Ensure: at the end, r holds all the elements of a and b in sorted order

```

1: function MERGE( $a, n_a, b, n_b, r$ )
2:    $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$             $\triangleright$  cursore array  $a$ ; cursore array  $b$ ; cursore array  $r$ 
3:   while  $i < n_a \wedge j < n_b$  do
4:     if  $a[i] \leq b[j]$  then
5:        $r[k] = a[i]; i \leftarrow i + 1; k \leftarrow k + 1$ 
6:     else
7:        $r[k] = b[j]; j \leftarrow j + 1; k \leftarrow k + 1$ 
8:     while  $i < n_a$  do
9:        $r[k] = a[i]; i \leftarrow i + 1; k \leftarrow k + 1$ 
10:    while  $j < n_b$  do
11:       $r[k] = b[j]; j \leftarrow j + 1; k \leftarrow k + 1$ 
return  $r$ 

```

Merge

Dentro il primo while controllo le due parti con due contatori, chi è minore lo metto nell'array che poi sarà quello finale e aumento il contatore solo di quella parte (e quello dell'array finale ovviamente). Guardando la figura sotto, nella penultima fila verde faccio così:

1. Fra 3 e 9 chi è minore? 3 quindi lo metto nell'array e aumento il contatore del suo array;
2. Fra 27 e 9? 9, lo metto nell'array finale e aumento il suo contatore;
3. Fra 27 e 10?
4. E così via.

Per finire riempio le caselle dell'array finale non toccate perchè fuori dal conteggio dei sotto array con il resto degli elementi.

La complessità sarà: $\Theta(n_a + n_b)$

Require: $temp$ is a support array of length n

```

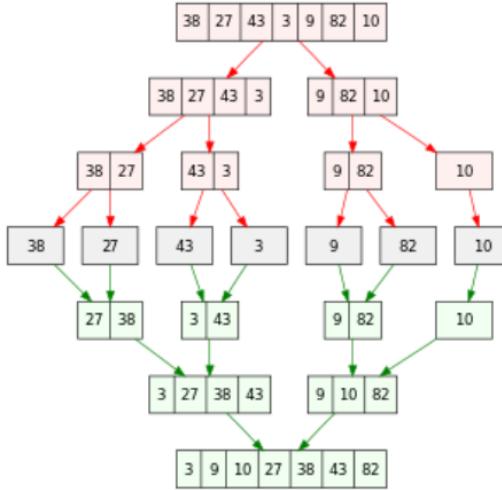
1: function MERGE-SORT( $a, n, temp$ )
2:   if  $n \leq 1$  then return            $\triangleright$  Caso base: spezziamo ricorsione su array vuoto/singleton
3:    $m \leftarrow \lfloor n/2 \rfloor$ 
4:    $left \leftarrow a; n_{left} = m$ 
5:    $right \leftarrow a + m; n_{right} = n - m$        $\triangleright a + m$  da interpretarsi à la C (aritm. puntatori)
6:   merge-sort( $left, n_{left}, temp$ )
7:   merge-sort( $right, n_{right}, temp$ )
8:   merge( $left, n_{left}, right, n_{right}, temp$ )       $\triangleright$  Complessità  $\Theta(n_{left} + n_{right}) = \Theta(n)$ 
9:   for  $i \in 0..n - 1$  do                   $\triangleright$  Complessità  $\Theta(n)$ 
10:     $a[i] \leftarrow temp[i]$ 

```

Merge sort

Ricordarsi che l'algoritmo lavora in *out-of-place* (array di appoggio), come complessità si parla di: $\Theta(n \log n)$.

Visualizzando i passaggi:



Quick sort

Vogliamo le performance del *merge-sort* lavorando **in-place**, per farlo dividiamo in due sottosequenze indipendenti in modo da ordinarle indipendentemente l'un l'altra.

Def | Problema del partizionamento: *data una sequenza a_0, \dots, a_{n-1} e scelto un pivot (x) vogliamo ottenere una sequenza dove tutti gli elementi che precedono x sono minori o uguali di x , e dove tutti gli elementi che seguono x sono maggiori o uguali di x .*

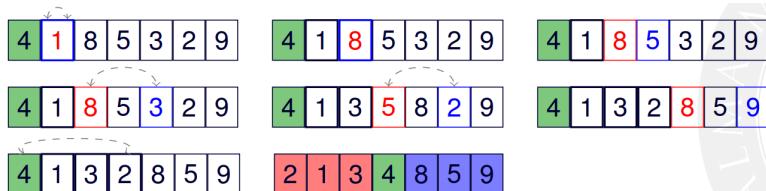
Prima definiamo l'algoritmo che divide l'array in due in base al pivot scelto:

```

1: function PARTITION( $a, n$ )
2:    $pivotIndex \leftarrow 0$                                  $\triangleright$  Prendiamo il primo elemento come pivot
3:    $k \leftarrow 1$                                           $\triangleright$  Cursore sotto-array di elementi minori del pivot
4:   for  $i \in [1, n - 1]$  do                          $\triangleright$  Cursore array complessivo
5:     if  $a[i] < a[pivotIndex]$  then
6:       SWAP( $a[i], a[k]$ )     $\triangleright$  Gli elementi più piccoli del pivot sono messi in posizioni  $k$  successive,
  mentre  $i$  scandaglia tutto l'array.
7:        $k \leftarrow k + 1$ 
8:     swap( $a[pivotIndex], a[k - 1]$ )  $\triangleright$   $k$  sarebbe la prossima posizione di un elemento minore del pivot.
  Quindi  $k - 1$  è l'ultima posizione. Vi mettiamo il pivot e ne restituiamo l'indice.
9:   return  $k - 1$ 

```

Complessità $\Theta(n)$, visualizzando i passaggi:



Ora passiamo all'algoritmo vero e proprio:

```

1: function QUICK-SORT( $a, n$ )
2:   if  $n < 2$  then
3:      $k \leftarrow \text{partition}(a, n)$            $\triangleright$  Viene determinato il pivot (e la sua posizione fissata)
4:     quick-sort( $a, k$ )                    $\triangleright$  Cf. aritmetica puntatori
5:     quick-sort( $a + k + 1, n - k - 1$ )   $\triangleright$  Cf. aritmetica puntatori

```

La complessità si divide in:

- **best case**: $\Theta(n \log n)$;
- **worst case**: $\Theta(n^2) \Rightarrow$ se il pivot è il min/max.

Il *quick sort* prima partizione e poi ricorre (contrario rispetto al *merge sort*)

Complessità a confronto

Nel caso generale, nessun algoritmo di ordinamento per confronto può avere complessità computazionale nel caso peggiore inferiore a $\Theta(n \log n)$.

Esistono algoritmi di ordinamenti che non si basano sul confronto tra coppie, ma su altre operazioni. Spesso sfruttano proprietà specifiche della rappresentazione del tipo di dato degli elementi della sequenza, e quindi non sono di applicabilità generale.

Sceglieremo un algoritmo rispetto ad un altro in base a:

- **Necessità ordinamento sul posto**;
- **Complessità spaziale**;
- **Efficienza effettiva**;
- Certi algoritmi potrebbero funzionare bene per **specifiche istanze del problema**;
- La complessità asintotica è rappresentativa dell'onere per $n \rightarrow \infty$, ma per bassi valori di n , un algoritmo $\Theta(n^2)$ potrebbe essere più veloce di uno $\Theta(n \log n)$.

Algoritmo	Tempo (best)	Tempo (avg)	Tempo (worst)	Spazio (worst)
SELECTION SORT	$\Theta(n^2)$		$\Theta(n^2)$	$\Theta(1)$
INSERTION SORT	$\Theta(n)$		$\Theta(n^2)$	$\Theta(1)$
BUBBLE SORT	$\Theta(n)$		$\Theta(n^2)$	$\Theta(1)$
MERGE SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
QUICK SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	$\Theta(n)^1$

Grafi e alberi

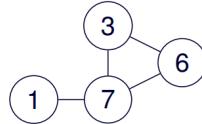
Grafi

Un grafo è un insieme di nodi (o vertici, node) e collegamenti (o archi, edge) tra nodi, formalmente:

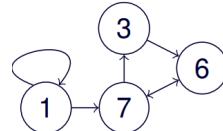
Grafo è $G = (V, E)$ con V che è l'insieme dei nodi ed E è l'insieme di archi ($V \times V$)

I grafi possono essere:

- **non orientato**: un arco è equivalente ad un altro, non ci sono versi, i collegamenti sono detti edge;



- **orientato:** un arco non è equivalente ad un altro, c'è un verso, i collegamenti sono detti arc.



se è presente una doppia freccia vuol dire che la coppia è presente due volte

Esempio di rappresentazione:

- $V = \{1, 3, 7, 6\}$
- $E = \{(1, 1), (1, 7), \dots\}$

Definizioni

Un nodo (u) si dice **adiacente** ad un altro nodo (v) se esiste un arco che li collega (u, v).

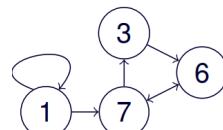
Il **grado** di un nodo prende due diverse definizioni, se il grafo è orientato, il grado, è il numero dei suoi archi incidenti altrimenti è il numero di archi entranti e uscenti ad esso.

Il **cammino** è una sequenza di nodi tali che esiste un arco tra ogni coppia consecutiva di nodi, dove la sua lunghezza è data dal numero di archi percorsi per raggiungere il nodo finale partendo da quello iniziale.

Esiste anche il **ciclo** ovvero un cammino che torna al punto di inizio; un grafo senza cicli è detto **aciclico**.

Se ogni coppia di un grafo è collegata si dice **grafo connesso**; un sottografo massimale in cui ogni coppia di nodi è connessa da un cammino è detto **componente connessa**.

Rappresentazione



I grafi possono essere rappresentati tramite:

- **liste di adiacenza:** per ogni nodo, si elencano i nodi adiacenti:

$$\begin{aligned} 1: [1, 7] & \quad 6: [7] \\ 3: [1, 6] & \quad 7: [3] \end{aligned}$$

Utile per rappresentare i grafi **sparsi** (quando i numeri di archi sono minori del numero di coppie possibili).

- **matrice di adiacenza:** usiamo una matrice per indicare se è presente un arco per ogni coppia di nodi:

	1	3	6	7
1	1	0	0	1
3	1	0	1	0
6	0	0	0	1
7	0	1	0	0

Utile per rappresentare i grafi densi (quando i numeri di archi sono circa lo stesso numero delle coppie possibili).

Graph traversal

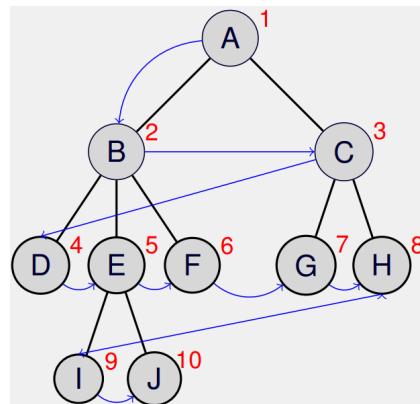
Il problema della visita sistematica di tutti i nodi di un grafo.

Def | Graph traversal: Dato un grafo $G = (V, E)$ e un nodo $r \in V$ (chiamato sorgente o radice), si visiti ogni nodo del grafo raggiungibile da r , con il vincolo che ogni nodo deve essere visitato una sola volta.

Per ovviare a questo problema sono nati due algoritmi principali:

Breadth-First Visit (BFV)

Si inizia visitando i vicini del sorgente, poi i vicini dei vicini, e così via; praticamente si fanno prima tutti quelli vicini ad un nodo, poi si passa ad un altro.



Per implementarlo teniamo traccia dei nodi già visitati e del loro padre, possiamo usare un coda (FIFO) e accodare i nodi adiacenti non ancora visitati.

- **INPUT:** un grafo (G) e un nodo radice $root$;
- **OUTPUT:** possiamo ritornare un nodo che soddisfa un predicato (passato in input [f]) e si parla di **BFS** o un'annotazione e dei nodi visitati e del loro rispettivo parent.

```

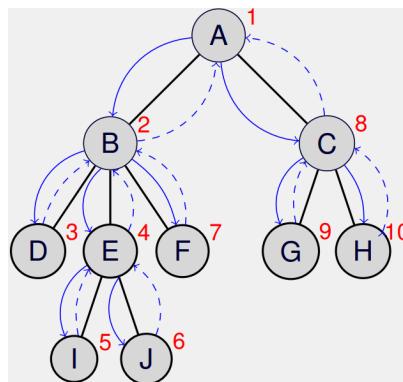
1: function BFV( $G, root, f$ )
2:   Let  $Q$  be an empty queue
3:    $root.visited \leftarrow \top$ 
4:   ENQUEUE( $Q, root$ )
5:   while  $Q$  not empty do
6:      $v \leftarrow \text{DEQUEUE}(Q)$ 
7:     if  $f(v)$  then return  $v$ 
8:     for  $w$  in  $\text{NEIGHBORS}(G, v)$  do
9:       if  $\neg w.visited$  then
10:         $w.visited \leftarrow \top$ 
11:         $w.parent \leftarrow v$ 
12:        ENQUEUE( $Q, w$ )

```

▷ To avoid re-visitation

Depth-First Visit (DFV)

Prima di passare a un secondo vicino, si attraversano tutti i nodi raggiungibili dal primo vicino



Per implementarlo dobbiamo fare in modo che prima di completare la visita del nodo corrente, ricorsivamente si visiti in ordine ogni nodo adiacente non ancora visitato.

- **INPUT:** un grafo (G) e un nodo radice $root$;
- **OUTPUT:** possiamo ritornare un nodo che soddisfa un predicato (passato in input [f]) e si parla di **DFS** o un'annotazione e dei nodi visitati e del loro rispettivo parent.

```

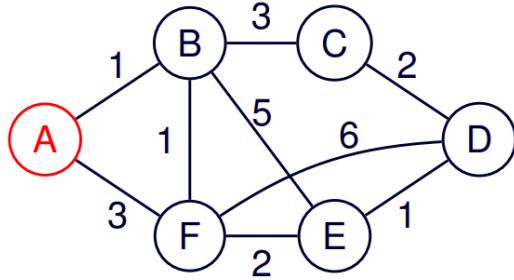
1: function DFV( $G, root, f$ )
2:    $root.visited \leftarrow \top$ 
3:   for  $w$  in  $\text{NEIGHBORS}(G, root)$  do
4:     if  $\neg w.visited$  then
5:       DFV( $G, w, f$ )                                ▷ Visita ricorsiva del sottoalbero con radice  $w$ 
6:     if  $f(v)$  then return  $v$ 

```

Dijkstra

Un algoritmo per risolvere il problema a del cammino di costo minimo, cioè:

dato un grafo pesato e un nodo sorgente, determinare per ogni nodo il cammino a costo minimo di percorrenza (dove il costo di un cammino è dato dalla somma dei pesi degli archi percorsi).



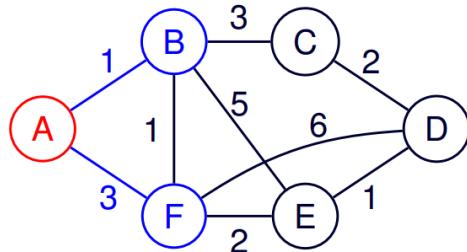
Implementazione in pseudo codice:

```

1: function DIJKSTRA( $G = (V, E)$ ,  $src$ )
2:   for  $v \in V$  do
3:      $v[visited] \rightarrow \perp$                                  $\triangleright$  Initialisation
4:      $v[prev] \rightarrow None$                                 $\triangleright$  Keep track if node has been visited yet
5:      $v[dist] \rightarrow \infty$                               $\triangleright$  Keep track of parent node of shortest path
6:    $src[dist] \rightarrow 0$                                   $\triangleright$  Keep track of cost of shortest path
7:   while  $\exists v \in V : v[visited] = \perp \wedge v[dist] \neq \infty$  do     $\triangleright$  Until all nodes have been visited and at least
       one node has distance less than  $\infty$  (note: at the beginning, only the source)
8:      $v \leftarrow$  node not yet visited at minimum distance
9:      $v[visited] \rightarrow \top$ 
10:    for  $nbr \in neighbours(v)$  do
11:       $d \leftarrow v[dist] + distance(v, nbr)$ 
12:      if  $d < nbr[dist]$  then                          $\triangleright$  Update only if the new parent decreases the path cost
13:         $nbr[dist] \leftarrow d$ 
14:         $nbr[prev] \leftarrow v$ 

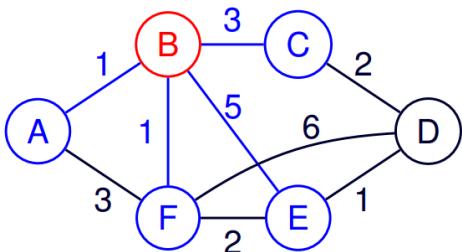
```

Procedimento visualizzato:

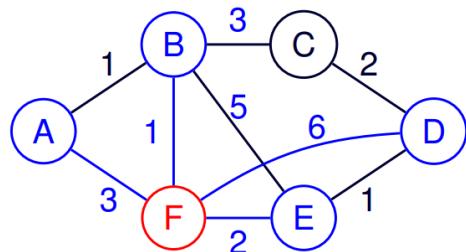


Nodi	<i>dist</i>	<i>prev</i>	<i>neighbours</i>
A	0	?	B, F
B	∞	?	A, C, E, F
C	∞	?	B, D
D	∞	?	C, E, F
E	∞	?	B, D, F
F	∞	?	A, B, E, D

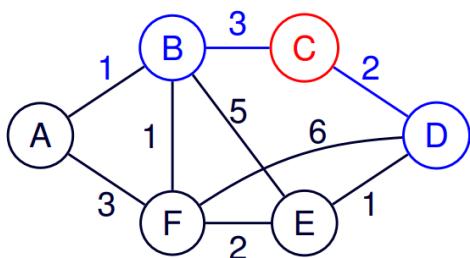
Nodi	<i>dist</i>	<i>prev</i>	<i>neighbours</i>
A	0	?	B, F
B	1	A	A, C, E, F
C	∞	?	B, D
D	∞	?	C, E, F
E	∞	?	B, D, F
F	3	A	A, B, E, D



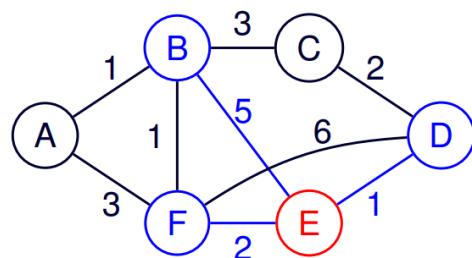
Nodi	dist	prev	neighbours
A	0	?	B, F
B	1	A	A, C, E, F
C	∞	X B	B, D
D	∞	?	C, E, F
E	∞	X B	B, D, F
F	2	X B	A, B, E, D



Nodi	dist	prev	neighbours
A	0	?	B, F
B	1	A	A, C, E, F
C	4	B	B, D
D	∞	X F	C, E, F
E	4	X F	B, D, F
F	2	B	A, B, E, D



Nodi	dist	prev	neighbours
A	0	?	B, F
B	1	A	A, C, E, F
C	4	B	B, D
D	∞	X C	C, E, F
E	4	F	B, D, F
F	2	B	A, B, E, D



Nodi	dist	prev	neighbours
A	0	?	B, F
B	1	A	A, C, E, F
C	4	B	B, D
D	∞	X E	C, E, F
E	4	F	B, D, F
F	2	B	A, B, E, D

Alberi binari di ricerca [BST]

Per i grafi indiretti un albero è un grafo indiretto connesso aciclico, e per i grafi diretti un albero è un grafo diretto aciclico dove vi è un nodo radice senza archi entranti (nessun nodo padre) e tutti gli altri nodi hanno un singolo arco entrante (un solo nodo padre).
Una foresta è un insieme di alberi.

Definizioni

Radice: unico nodo dell'albero senza archi entranti (ovvero, senza padre) (A)

Foglia: nodo senza archi uscenti (D,I,J,G,H)

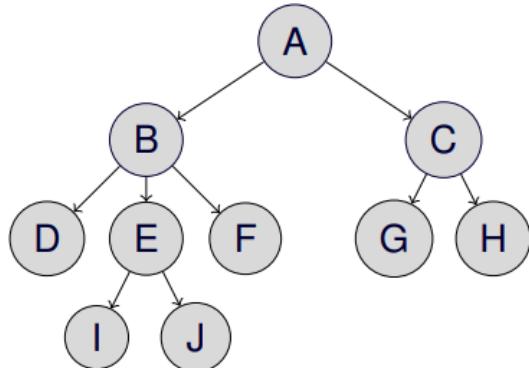
Padre: nodo con un arco uscente verso un altro nodo (figlio) oppure nodo che non è foglia o nodo radice di un sottoalbero non vuoto (A,B,C,E)

Figlio: nodo con un arco entrante (B,C,D,E,F,G,H,I,J)

Discendente: nodo figlio, o il figlio di un discendente

Antenato: nodo padre, o il padre di un antenato

Nodo interno: nodo che non è né radice né foglia (B,C,E)



Profondità di un nodo [D(n)]: numero di archi tra la radice e il nodo, ovviamente la radice ha profondità 0;

Altezza di un nodo H(n): num di archi del percorso più lungo tra un nodo e una foglia;

Profondità/altezza di un albero: l'altezza del nodo radice, o equivalentemente, la profondità massima raggiunta dai nodi dell'albero.

Un albero con al massimo due figli è detto **binario**, si dice **ordinato** se, per ogni nodo n , il suo valore x non è minore del valore di tutti i nodi del sottoalbero sinistro, e non è maggiore del valore di tutti i nodi del sottoalbero destro.

Un livello si dice **completo** se il livello precedente è completo e ogni nodo del livello precedente ha entrambi i figli.

Invece parlando di albero binario, si dice che è:

- **bilanciato:** se per ogni nodo le altezze dei due sottoalberi differiscono al più di 1;
- **perfettamente bilanciato:** se tutte le foglie hanno la stessa profondità;
- **completo:** se è perfettamente bilanciato e tutti i nodi interni hanno grado 2;
- **quasi completo:** se tutti i livelli tranne al più l'ultimo sono completi, quindi è bilanciato e tutti i nodi interni hanno grado 2.

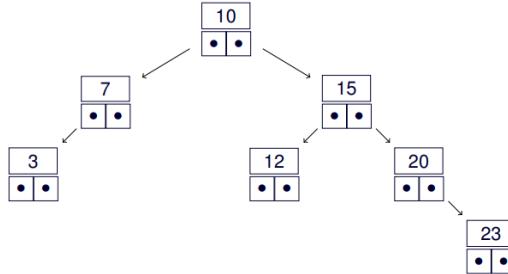
Un albero binario di n nodi si dice ottimo se ha altezza $h = \lfloor \log_2 n \rfloor$

e

un albero binario ottimo ha altezza minima rispetto a tutti gli alberi che possono rappresentare lo stesso insieme di dati

Implementazione

```
1 typedef int TInfo;
2 typedef struct SNode {
3     TInfo info;
4     struct SNode *left;
5     struct SNode *right;
6 } TNode;
7 typedef TNode* TBinaryTree;
```

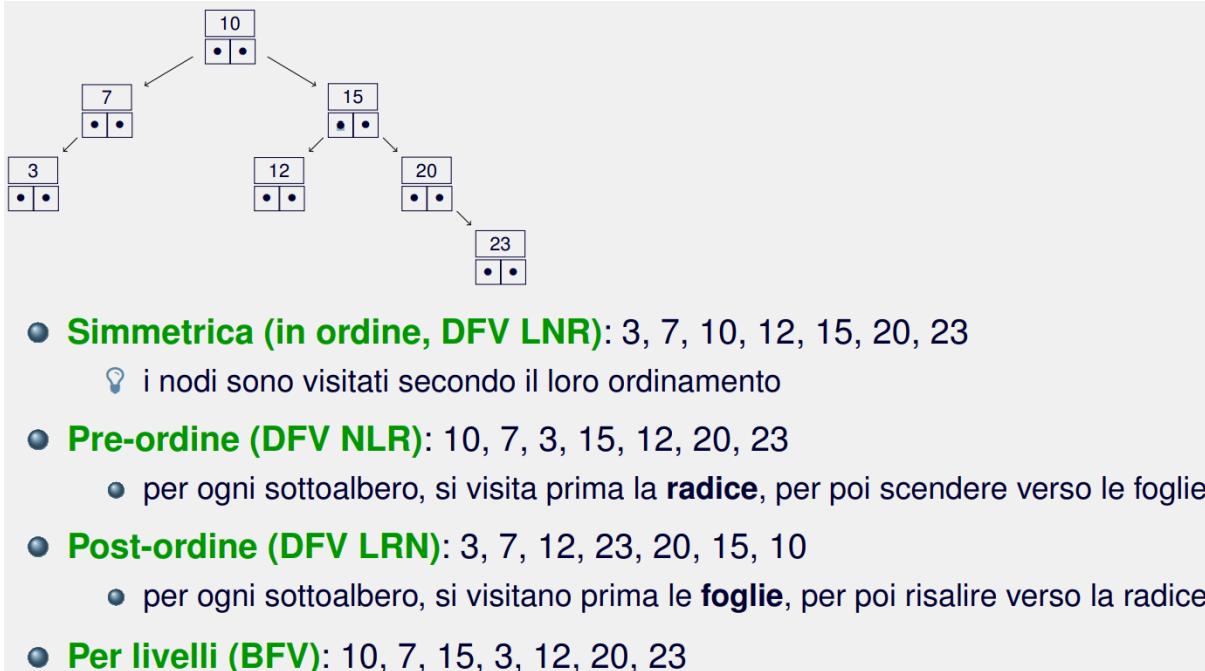


Operazioni

```
1 // Operations on nodes
2 TNode *node_create(TInfo value);
3 void node_destroy(TNode* node);
4
5 // Creation and destruction of btrees
6 TBinaryTree binarytree_create();
7 TBinaryTree binarytree_destroy();
8
9 // Operations on btrees
10 void binarytree_visit(TBinaryTree tree, void (*f)(TInfo));
11 TNode *binarytree_search(TBinaryTree tree, TInfo value);
12 TBinaryTree binarytree_insert(TBinaryTree tree, TInfo info);
13 TBinaryTree binarytree_delete(TBinaryTree tree, TInfo info);
14 bool binarytree_is_empty(TBinaryTree tree);
```

Abbiamo varie tipologie di **visita** di un BST:

- **Depth-First Visit (DFV)**: visita simmetrica in ordine
 1. Visita simmetrica del sottoalbero sinistro;
 2. Visita del nodo radice;
 3. Visita simmetrica del sottoalbero destro;
- **Depth-First Visit (DFV)**: Visita in pre-ordine
 1. Visita del nodo radice;
 2. Visita in pre-ordine del sottoalbero sinistro;
 3. Visita in pre-ordine del sottoalbero destro;
- **Depth-First Visit (DFV)**: Visita in post-ordine
 1. Visita in post-ordine del sottoalbero sinistro;
 2. Visita in post-ordine del sottoalbero destro;
 3. Visita del nodo radice;
- **Breadth-First Visit (BFV)**



```

1 void binarytree_visit(TBinaryTree tree, void (*f)(TInfo)) {
2     if(tree != NULL) {
3         binarytree_visit(tree->left, f);
4         f(tree->info);
5         binarytree_visit(tree->right, f);
6     }
7 }
```

Complessità: $\Theta(n)$

Per la **ricerca** in un BST possiamo usare l'approccio divide-et-impera:

- **Divide:** la ricerca si può dividere nella ricerca nelle diverse parti della struttura dati;
- **Impera:**
 - **caso base:** l'albero è vuoto (NULL) o la radice contiene il valore cercato;
 - **caso ricorsivo:** si assume applicazione corretta dell'algoritmo a sottoalbero;
- **Combina:** se non ci si trova nei casi base, si confronta il valore del nodo con il valore da cercare e lì si sceglie se cercare nel sottoalbero sinistro o destro.

```

1 TNode *binarytree_search(TBinaryTree t, TInfo v) {
2     if(t == NULL || equal(v, t->info)) return t;
3     return binarytree_search(less(v, t->info) ? t->left : t->right, v);
4 } // ricorsione diretta, lineare, in coda
```

Complessità [worst case]: $\Theta(\log n)$

Per la **ricerca del /min/max** dobbiamo tenere conto che il più piccolo elemento è il nodo più a sinistra, infatti per l'ordinamento i padri hanno sempre valore maggiore dei nodi del sottoalbero sinistro.

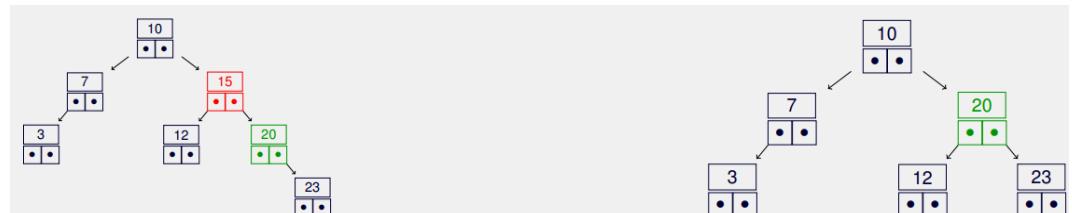
```
1 TNode *binarytree_search_min(TBinaryTree tree) {  
2     if(tree == NULL || tree->left == NULL) return tree;  
3     return binarytree_search_min(tree->left);  
4 }
```

Per l'**inserimento** dobbiamo ricordare di mantenere la proprietà di ordinamento, trattiamo l'inserimento come foglia: riduciamo e il problema all'inserimento nel sottoalbero sinistro o destro.

```
1 TBinaryTree binarytree_insert(TBinaryTree tree, TInfo info) {  
2     if(tree == NULL) {  
3         return node_create(info);  
4     } else {  
5         if(less(info, tree->info) || equal(info, tree->info) ) {  
6             tree->left = binarytree_insert(tree->left, info);  
7         } else {  
8             tree->right = binarytree_insert(tree->right, info);  
9         }  
10    }  
11    return tree;  
12 }
```

La rimozione di un nodo si distingue in:

1. **il nodo da rimuovere è una foglia:** basta eliminare la foglia;
 2. **il nodo da rimuovere ha un solo figlio:** basta sostituire il collegamento al nodo con il collegamento al figlio del nodo;
 3. **il nodo da rimuovere ha due figli:** sostituzione del nodo da cancellare con il max del sottoalbero sinistro o con il min del sottoalbero destro.



Rimozione di un nodo in un BST: implementazione C

```

1 TBinaryTree binarytree_delete(TBinaryTree tree, TInfo info) {
2     if(tree == NULL) return tree;
3     if(equal(tree->info, info)) {
4         if (tree->left != NULL && tree->right != NULL) {
5             TBinaryTree min = binarytree_search_min(tree->right);
6             tree->info = min->info;
7             tree->right = min->right;
8             node_destroy(min);
9             return tree;
10        } else if(tree->left == NULL && tree->right == NULL) {
11            node_destroy(tree);
12            return NULL;
13        } else {
14            TBinaryTree result = tree->left ? tree->left : tree->right;
15            node_destroy(tree);
16            return result;
17        }
18    } else {
19        if(less(tree->info, info)) {
20            tree->right = binarytree_delete(tree->right, info);
21        } else {
22            tree->left = binarytree_delete(tree->left, info);
23        }
24    }
25    return tree;
26}

```

Complessità

Operazione	Best	Average	Worst
Inserimento	$\Theta(1)$	$\Theta(h)$	$\Theta(h)$
Ricerca (elem. presente)	$\Theta(1)$	$\Theta(h)$	$\Theta(h)$
Ricerca (elem. assente)	$\Theta(h)$	$\Theta(h)$	$\Theta(h)$
Cancellazione	$\Theta(1)$	$\Theta(h)$	$\Theta(h)$
Visita	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Varianti BST

Un'importante classe di BST è data dei BST auto-bilancianti (self-balancing): sfruttano operazioni di ribilanciamento a fronte di modifiche (inserimenti, cancellazioni) al BST, per garantirne un buon bilanciamento (e quindi prevenire sviluppo eccessivo in altezza che, come visto, impatta la complessità di varie operazioni); tra le tecniche di ribilanciamento vi è quella basata su rotazioni dell'albero, che cambiano la struttura senza impattare sull'ordine degli elementi.

Alcuni esempi sono:

- **Red-Black Trees**: i nodi hanno un “bit” di colore (“rosso” o “nero”) che viene utilizzato per vincolare/riorganizzare il BST;
- **Adelson-Velsky and Landis (AVL) Trees**: sono più rigidamente bilanciati rispetto ai Red-Black Tree; ovvero, hanno un più stretto limite d’altezza.

B-tree: generalizzano i BST auto-bilancianti, ammettendo nodi con più di 2 figli; sono particolarmente utili per sistemi che leggono/scrivono grossi blocchi di dati