

1_Introduzione algoritmi.

Cominciamo dal principio: la definizione di informatica proposta dall'ACM (Association for Computing Machinery):
"L'informatica è la scienza degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione e applicazione (programmazione vera e propria, secondo modulo)."

Quello di algoritmo, dunque, è un concetto fondamentale, centrale per l'informatica.

Un **algoritmo** è "una sequenza di comandi elementari ed univoci che terminano in un tempo finito ed operano su strutture dati".

Un comando è **elementare** quando non può essere scomposto in comandi più semplici; è **univoco** quando può essere interpretato in un solo modo.

Ad esempio, domandiamoci se la seguente ricetta per fare un uovo al tegamino sia o no un algoritmo:

1. Rompere un uovo in padella. Non è né elementare né univoco in quanto, dopo aver rotto l'uovo (cosa che già si può fare in molti modi, non tutti utili allo scopo), si deve separare il guscio dal resto.
2. Cuocere l'uovo. Non è elementare: l'attività di cottura si può scomporre nell'accensione del fornello, nell'aggiunta del condimento e del sale, prevede il controllo della cottura, eccetera.

Se un algoritmo è ben specificato, chi (o ciò che) lo esegue non ha bisogno di pensare, deve solo eseguire con precisione i passi elencati nell'algoritmo, nella sequenza in cui appaiono.

E infatti un calcolatore non pensa, esegue pedissequamente tutte le operazioni elencate negli algoritmi pensati (ossia progettati) da un essere umano.

Se si verifica un errore e il risultato è sbagliato, l'errore non è del calcolatore ma di chi ha progettato l'algoritmo!

Per risolvere i problemi abbiamo, ovviamente, bisogno di gestire i relativi dati. A tal fine dovremo definire le opportune **strutture dati**: strumenti necessari per organizzare e memorizzare i dati veri e propri, semplificandone l'accesso e la modifica, specificando il modo in cui interagire con queste strutture dati.

Algoritmi e strutture dati sono strettamente collegati, vanno di pari passo.

Non esiste una struttura dati che sia adeguata a ogni problema (ognuna ha le sue caratteristiche), per cui è necessario conoscere proprietà, vantaggi e svantaggi delle principali strutture dati in modo da poter scegliere di volta in volta quale sia quella più adatta (o più facilmente adattabile) al problema.

Il progetto o la scelta della struttura dati da adottare nella soluzione di un problema è un aspetto fondamentale per la risoluzione del problema stesso, al pari del progetto dell'algoritmo.

Perciò, gli algoritmi e le strutture dati fondamentali vengono sempre studiati e illustrati assieme.

Scrivere un buon algoritmo si basa sulla corretta struttura dati.

Dalla definizione di algoritmo ci dobbiamo ricordare che questo termina in un tempo finito.

Affinché un algoritmo sia utilizzabile, deve concludersi e produrre il suo output entro un tempo "ragionevole".

Un aspetto fondamentale che va affrontato nello studio degli algoritmi è la loro **efficienza**, cioè la quantificazione delle loro esigenze in termini di tempo e di spazio, ossia tempo di esecuzione e quantità di memoria richiesta.

Efficienza: quantificare in modo preciso il tempo e lo spazio per risolvere un dato problema, quando il tempo è il minor possibile per il problema, l'algoritmo è efficiente.

L'**efficienza** è qualcosa da perseguire sempre perché:

- I calcolatori sono molto veloci, ma non infinitamente veloci;
- La memoria è economica e abbondante, ma non è né gratuita né illimitata.

Nel corso illustreremo il concetto di **costo computazionale** degli algoritmi, in termini di numero di operazioni elementari e quantità di spazio di memoria necessari in funzione della dimensione dell'input. Per costo computazione degli algoritmi intendiamo una funzione che dipende dal numero di operazioni elementari che compongono l'algoritmo e anche dalla quantità di spazio, di memoria necessaria per risolvere il problema.

Esempio:

La maggiore velocità di V riesce a bilanciare la minore efficienza dell'algoritmo IS?

Confrontiamo il tempo di esecuzione di IS sul calcolatore V con quello di MS sul calcolatore L.

Problema: ordinare $n=10^6$ numeri interi;

- Il calcolatore **V** (veloce) effettua **10^9** operazioni/ sec
- Il calcolatore **L** (lento) effettua **10^7** operazioni/ sec
- L'algoritmo **IS** (Insertion Sort) richiede **$2 n^2$** operazioni;
- L'algoritmo **MS** (Merge Sort) richiede **$50 n \log n$** operazioni.

$$\text{Tempo di } V(\text{IS}) = \frac{2(10^6)^2 \text{ istruzioni}}{10^9 \text{ istruzioni al secondo}} = 2000 \text{ secondi} = 33 \text{ minuti}$$

$$\text{Tempo di } L(\text{MS}) = \frac{50 \cdot 10^6 \log 10^6 \text{ istruzioni}}{10^7 \text{ istruzioni al secondo}} = 100 \text{ secondi} = 1,5 \text{ minuti}$$

Cioè, la risposta è no.

Se mettiamo insieme il calcolatore veloce con l'algoritmo lento e il calcolatore lento con l'algoritmo veloce, quale sarà il più veloce? Si bilanceranno?

Aumentiamo la dimensione dell'input, portandola da **10^6** a **10^7** :

$$\text{Tempo di } V(\text{IS}) = \frac{2(10^7)^2 \text{ istruzioni}}{10^9 \text{ istruzioni al secondo}} = 2 \text{ giorni}$$

$$\text{Tempo di } L(\text{MS}) = \frac{50 \cdot 10^7 \log 10^7 \text{ istruzioni}}{10^7 \text{ istruzioni al secondo}} = 20 \text{ minuti}$$

Cioè, indipendentemente dall'aumento di velocità dei calcolatori prodotto dagli avanzamenti tecnologici, **l'efficienza degli algoritmi è un fattore di importanza cruciale**.

Riuscire a progettare algoritmi veloci ed efficienti è molto importante.

Il **problem solving** è un'attività che ha lo scopo di raggiungere una soluzione a partire da una situazione iniziale. È quindi un'attività creativa, di natura essenzialmente progettuale, ed in questo risiede la sua difficoltà.

Approccio al problem solving:

- analisi del problema:** lettura approfondita della situazione iniziale, comprensione e identificazione del problema;
- esplorazione degli approcci possibili:** identificazione delle metodologie di soluzione tra i metodi noti;
- selezione di un approccio:** scelta dell'approccio migliore;
- definizione dell'algoritmo risolutivo:** identificazione dei dati e progettazione della sequenza di passi elementari da applicare su di essi;
- riflessione critica:** a problema risolto, ripensamento delle fasi della soluzione proposta per identificare eventuali criticità e possibili migliorie.

Quali problemi?

In questo corso restringiamo la nostra attenzione ai **problemi computazionali**, problemi cioè che richiedono di descrivere in modo automatico una specifica relazione tra un insieme di valori in input e il corrispondente insieme di valori in output.

Un algoritmo è corretto se, per ogni (input) istanza di un problema computazionale, termina producendo l'output corretto.

In tal caso diremo che l'algoritmo risolve il problema

Esempio di problema computazionale, definizione del problema: ordinare n numeri dal più piccolo al più grande:

Input (anche detto *istanza del problema*):

Sequenza di n numeri a_1, a_2, \dots, a_n ;

Output:

Permutazione a'_1, a'_2, \dots, a'_n della sequenza di input t.c.

$a'_1 \leq a'_2, \dots, \leq a'_n$.

Per poter valutare l'efficienza di un algoritmo è necessario **analizzarlo**, cioè quantificare le risorse che esso richiede per la sua esecuzione, **senza che tale analisi sia influenzata da una specifica tecnologia** che, inevitabilmente, col tempo diviene superata.

Se ho un problema e devo risolverlo, con il problem solving posso produrre più di un algoritmo, devo vedere qual è quello più efficiente e ho bisogno di essere su una macchina che sia equilibrata per tutti gli algoritmi.

Vogliamo confrontare degli algoritmi indipendentemente dalle macchine che stiamo utilizzando, per fare questo si usa la **Random Access Machine** (modello RAM) che è indipendente dalle caratteristiche tecniche di uno specifico calcolatore reale. (astrarre dalla tecnologia)

La RAM è quindi una macchina astratta, la cui validità e potenza concettuale risiede nel fatto che non diventa obsoleta con il progredire della tecnologia.

Nel modello RAM:

- esiste un singolo processore, che esegue le operazioni sequenzialmente (una dopo l'altra);
- esistono delle operazioni elementari, l'esecuzione di ciascuna delle quali richiede per definizione un tempo costante. (Es.: operazioni aritmetiche, letture, scritture, salto condizionato, ecc.);
- esiste un limite (dell'input) alla dimensione di ogni valore memorizzato ed al numero complessivo di valori utilizzati (il max valore rappresentabile in memoria **non** può superare 2 elevato al numero di bit della parola (32 o 64)).

Se è soddisfatta l'ipotesi che ogni dato in input sia minore di un valore: $K = 2^{\text{numero di bit della parola di memoria}}$
ciascuna operazione elementare sui dati del problema verrà eseguita in un tempo costante se sono memorizzati in due addendi nelle parole della memoria (rivedere 41m).

In tal caso si parla di misura di **costo uniforme**.

Tale **criterio non è sempre realistico** perché, se un dato del problema è più grande di k , esso deve comunque essere memorizzato, ed in tal caso si useranno **più parole di memoria**.

Di conseguenza, anche le operazioni elementari su di esso dovranno essere reiterate per tutte le parole di memoria che lo contengono, e quindi richiederanno un **tempo che non è più costante**: Calcolo scientifico e misura di costo logaritmico (non sarà affrontato qui).

Esempio:

Calcolo della potenza di 2, calcola 2^n .

```
def PotenzaDi2(n)
x = 1
for i in range(n):
    x = x*2
return x
```

(ciclo, reiterato n volte, che calcola il valore 2^n)

Il tempo di esecuzione totale è **proporzionale ad n**:

- si tratta di un **ciclo eseguito n volte**;
- ad ogni iterazione del ciclo si compiono due operazioni, ciascuna delle quali ha **costo unitario**:
 1. l'incremento del contatore i ;
 2. il calcolo del nuovo valore di x.

Affinché tutti siamo in grado di comprendere un algoritmo, è necessario utilizzare una descrizione:

- il più formale possibile;
- indipendente dal linguaggio che si intende usare;

Pseudocodice

(di cui ripareremo...)

2_Notazione asintotica.

- Vogliamo valutare l'efficienza di un algoritmo, così da poterlo confrontare con algoritmi diversi che risolvono lo stesso problema.
- Lo facciamo in termini di **costo computazionale**, ovvero del tempo di esecuzione di un algoritmo e delle sue necessità in termini di memoria.

In generale, questa valutazione è complicata e contiene spesso dettagli inutili che vorremmo ignorare, per cui ci limitiamo a una visione più astratta e si valuta su quello che possiamo chiamare "tasso di crescita", cioè la velocità con cui il tempo di esecuzione cresce all'aumentare della dimensione dell'input. Poiché per piccole dimensioni dell'output il tempo usato è comunque poco, tale valutazione è più interessante quando la dimensione dell'input è grande, per questo si parla di efficienza asintotica degli algoritmi.

- Prediligiamo il tempo di esecuzione all'occupazione di memoria.

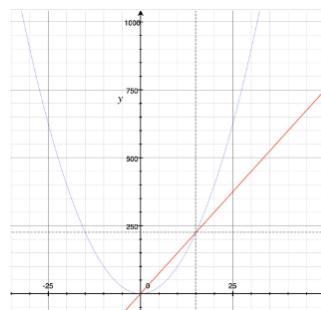
In matematica la notazione asintotica permette di confrontare il tasso di crescita del tempo impiegato (comportamento asintotico) di una funzione nei confronti di un'altra. Più il tasso di crescita è lento, migliore è l'algoritmo che stiamo progettando

Si vogliono confrontare le 2 funzioni in termini di quanto velocemente crescono quando n va a $+\infty$.

Per valori piccoli di n la funzione blu è più piccola, ma da un certo punto la blu va più veloce a ∞ .
La funzione più efficiente è quella rossa.

$$f(n)=15n+1$$

$$g(n)=n^2$$



In informatica, il **calcolo asintotico** è utilizzato per analizzare la complessità di un algoritmo. In particolar modo, per stimare quanto aumenta il tempo al crescere della dimensione n dell'input.

Ci aspettiamo che queste funzioni vadano all'infinito e ci andranno con dei **tempi e velocità diverse**.

Esempio: un retta va all'infinito più lentamente rispetto ad una parabola.

Esistono tre notazioni asintotiche:

Notazione asintotica O :

La notazione O grande è il **limite superiore** asintotico

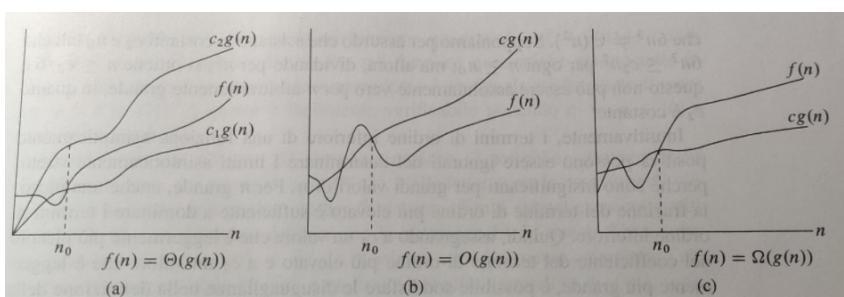
Notazione asintotica Ω :

La notazione Ω è il **limite inferiore** asintotico

Notazione asintotica Θ :

La notazione Θ è il **limite asintotico stretto**

Tale valutazione ha senso quando la dimensione dell'input è sufficientemente grande. Per questo si parla di efficienza asintotica degli algoritmi.



L'idea è che se si riesce a schiacciare la nostra funzione f tra una funzione Ω dal basso e una funzione O dall'alto ho esattamente l'indicazione di quanto velocemente la funzione va all'infinito.

Se la funzione è la stessa parliamo di Θ (teta)

Notazione O – O grande (1)

Date due funzioni $f(n), g(n) \geq 0$ si dice che

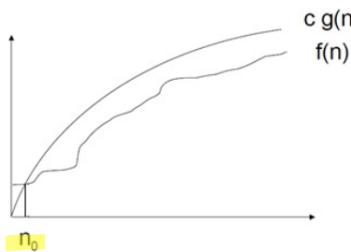
$f(n)$ è in $O(g(n))$

se esistono due costanti c ed n_0 tali che

$0 \leq f(n) \leq c g(n)$ per ogni $n \geq n_0$

Le funzioni rappresentano un tempo
impiegato dal nostro algoritmo.

Anche se $f(n)$ è molto strana,
sappiamo che $g(n)$ all'inizio
sia più piccola, la funzione
 $g(n)$ da un certo n_0 in poi si
troverà sempre sopra.



In $O(g(n))$
troviamo tutte
le funzioni
che risultano
«dominate»
dalla
funzione $g(n)$

Notazione O – O grande (2)

Esempio. $f(n) = 3n + 3$

$f(n)$ è in $O(n^2)$ in quanto, posto $c = 6$:

$$cn^2 \geq 3n + 3 \text{ per ogni } n \geq 1.$$

Ma $f(n)$ è anche in $O(n)$ in quanto:

$$\begin{aligned} cn \geq 3n + 3 \text{ per ogni } n \geq 1 \text{ se } c \geq 6, \\ \text{oppure per ogni } n \geq 3 \text{ se } c \geq 4. \end{aligned}$$

💡 data $f(n)$, esistono **infinite funzioni** $g(n)$ per cui $f(n)$ risulta in $O(g(n))$.

Vogliamo determinare la funzione $g(n)$ che meglio approssima la funzione $f(n)$ dall'alto o la più piccola $g(n)$ tale che $f(n)$ sia $O(g(n))$.

Notazione O – O grande (3)

Esempio. Sia $f(n) = n^2 + 4n$

$f(n)$ è in $O(n^2)$ in quanto:

$$\begin{aligned} cn^2 \geq n^2 + 4n \text{ per ogni } n \text{ se } c \geq 5 \\ \text{oppure per ogni } n \geq 4/(c-1) \text{ se } c > 1. \end{aligned}$$

$$\frac{cn^2}{n^2} \geq \frac{n^2 + 4n}{n^2}$$

$$c \geq 1 + \frac{4}{n}$$

Notazione O – O grande (4)

Esempio. Sia $f(n)$ un polinomio di grado m :

$$f(n) = \sum_{i=0}^m a_i n^i, \text{ con } a_m > 0$$

Dimostriamo, per induzione su m , che $f(n)$ è in $O(n^m)$

Caso base:

$m = 0$: $f(n) = a_0$, quindi $f(n)$ è una costante, cioè in $O(n^0) = O(1)$ per ogni n e per ogni $c \geq a_0$.

Notazione O – O grande (5)

Esempio (segue)

Ipotesi induttiva:

$\sum_{i=0}^k a_i n^i$ è un $O(n^k)$ per ogni $k < m$
cioè esiste una costante c' tale che $\sum_{i=0}^k a_i n^i \leq c' n^k$.

Passo induttivo:

Dobbiamo dimostrare che

$f(n) = \sum_{i=0}^m a_i n^i$ è in $O(n^m)$
cioè che esiste una costante c tale che:
 $\sum_{i=0}^m a_i n^i \leq c n^m$.

Notazione O – O grande (6)

Esempio (segue)

Si osservi che $f(n)$ si può scrivere come:

$$f(n) = \sum_{i=0}^m a_i n^i = a_m n^m + \sum_{i=0}^{k-1} a_i n^i = a_m n^m + h(n)$$

con $k < m$ e che, per l'ipotesi induttiva:

$$h(n) \leq c' n^k$$

Ora:

$$f(n) = a_m n^m + h(n) \leq a_m n^m + c' n^k \leq a_m n^m + c' n^m = (c' + a_m) n^m.$$

Ponendo $c = c' + a_m$ si ha la tesi.

Notazione O – O grande (7)

Esempio. Sia $f(n) = \log n$. $f(n)$ è in $O(\sqrt{n})$.

Più in generale, $\log^a n = O(n^{1/b})$ per ogni $a, b \geq 1$

Cioè:
un poli-logaritmo è dominato da una qualunque radice

Notazione O – O grande (8)

Notazione O – O grande (9)

Esempio. Sia $f(n) = n^{1/a}$. $f(n)$ è in $O(n)$ per ogni $a \geq 2$

Più in generale, $n^{1/a} = O(n^b)$ per ogni $a, b \geq 1$

Cioè:

Una radice è dominata da un qualunque polinomio

Esempio. Sia $f(n) = n^a$. $f(n)$ è in $O(2^n)$ per ogni $a \geq 1$

Più in generale, $n^a = O(b^n)$ per ogni $a \geq 1$, e $b \geq 2$

Cioè:

Un polinomio è dominato da un qualunque esponenziale

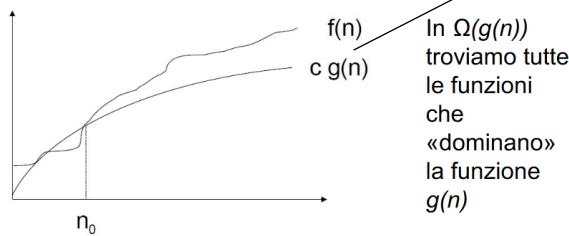
Notazione Ω - Omega (1)

Date due funzioni $f(n), g(n) \geq 0$ si dice che

$f(n)$ è in $\Omega(g(n))$

se esistono due costanti c ed n_0 tali che

$f(n) \geq c g(n)$ per ogni $n \geq n_0$



Va all'infinito più lentamente di f , da un certo punto in poi (n_0)

Notazione Ω - Omega (2)

Esempio. Sia $f(n) = 2n^2 + 3$

$f(n)$ è in $\Omega(n)$ in quanto $2n^2 + 3 \geq cn$ per qualunque n se $c = 1$

Ma $f(n)$ è anche in $\Omega(n^2)$ in quanto

$2n^2 + 3 \geq cn^2$ per ogni n , se $c \leq 2$
 $2n^2 - cn^2 \geq -3$
 $n^2(2 - c) \geq -3$
 vero se $c \leq 2$

Notazione Ω - Omega (4)

Esempio. Sia $f(n) = \sqrt{n}$. $f(n)$ è in $\Omega(\log n)$.

Più in generale, $n^{1/b} = \Omega(\log^a n)$ per ogni $a, b \geq 1$

Cioè:
 una radice domina qualunque poli-logaritmo

Notazione Ω - Omega (3)

Esempio. Sia $f(n)$ un polinomio di grado m :

$$f(n) = \sum_{i=0}^m a_i n^i, \text{ con } a_m > 0$$

La dimostrazione che $f(n)$ è in $\Omega(n^m)$ è analoga alla dimostrazione che $f(n)$ è in $O(n^m)$ e perciò viene lasciata come esercizio.

Notazione Ω - Omega (5)

Esempio. Sia $f(n) = 2^n$. $f(n)$ è in $\Omega(n^a)$ per ogni $a \geq 1$

Più in generale, $b^n = \Omega(n^a)$ per ogni $a \geq 1$, e $b \geq 2$

Cioè:
 Un esponenziale domina un qualunque polinomio

Notazione O e Ω - considerazioni.

Abbiamo visto che in entrambe le notazioni O e Ω , per ogni funzione $f(n)$ sia possibile trovare più funzioni $g(n)$.

In effetti **O($g(n)$)** e **$\Omega(g(n))$** sono **insiemi di funzioni**, e dire "f(n) è in O($g(n)$)" oppure "f(n) = O($g(n)$)" ha il significato di "f(n) appartiene a O($g(n)$)".

Abbiamo visto che data una funzione $f(n)$ io posso trovare tante funzioni sopra, tante funzioni sotto, ma poiché i limiti asintotici ci servono per stimare con la maggior precisione possibile il costo computazionale di un algoritmo, vorremmo trovare – fra tutte le possibili funzioni $g(n)$ – quella che più si avvicina a $f(n)$.

Per questo cerchiamo la più piccola funzione $g(n)$ per determinare O e la più grande funzione $g(n)$ per determinare Ω .

Una volta che trovo le due funzioni, se queste, sono la stessa funzione allora trovo Θ .

La definizione che segue formalizza questo concetto intuitivo.

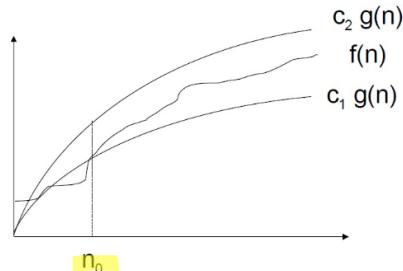
Notazione Θ – Teta (1)

Date due funzioni $f(n), g(n) \geq 0$ si dice che

$f(n)$ è in $\Theta(g(n))$

se esistono tre costanti C_1, C_2 ed n_0 tali che

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$



Notazione Θ – Teta (3)

Se ho due log con base diversa, vanno all' ∞ con la stessa velocità?

Esempio. Dimostrare che $f(n) = \log_a n = \Theta(\log_b n)$ per ogni $a, b > 0$.

Basta usare la formula per il cambio di base dei logaritmi:

$$\log_a n = \log_b n \cdot \frac{\log_a b}{\log_b a}$$

Il cambio di base è dunque asintoticamente irrilevante e per questo nella notazione asintotica la base del logaritmo viene spesso omessa.

Notazione Θ – Teta (2)

Sia O che Ω della stessa funzione $g \rightarrow \Theta$

Esempio. Sia $f(n) = 3n + 3$

$f(n)$ è in $\Theta(n)$ ponendo, ad esempio:

$$c_1 = 3, c_2 = 4, n_0 = 3$$

Infatti:

$$3n \leq 3n + 3 \leq 4n \text{ per } n \geq 3 \\ 3 \leq 3 + 3 \leq 4 \Rightarrow 3 \leq 12 \Rightarrow 3 \leq 12 \leq 12$$

Notazione Θ – Teta (4)

Esempio. Sia $f(n)$ un polinomio di grado m :

$$f(n) = \sum_{i=0}^m a_i n^i, \text{ con } a_m > 0$$

La dimostrazione che $f(n)$ è in $\Theta(n^m)$ discende dall'aver dimostrato che

$$\sum_{i=0}^m a_i n^i \text{ è sia in } O(n^m) \text{ che in } \Omega(n^m)$$

Calcolo della notazione asintotica tramite limiti.

- se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k > 0$ allora $f(n) = \Theta(g(n))$; Vanno all'infinito con la stessa.
- se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ allora $f(n) = O(g(n))$ ma $f(n) \neq \Theta(g(n))$; f(n) va più veloce all' ∞ .
- se $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ allora $f(n) = \Omega(g(n))$ ma $f(n) \neq \Theta(g(n))$. g(n) va più veloce all' ∞ .

Ovviamente, quando il limite non esiste, questo metodo non si può usare e bisogna procedere diversamente.

Algebra della notazione asintotica.

Per semplificare il calcolo del costo computazione asintotico degli algoritmi si possono sfruttare delle semplici regole che spieghiamo, di cui facciamo esempi e dimostriamo.

Regole sulle costanti moltiplicative.

1A: Per ogni $k > 0$ e per ogni $f(n) \geq 0$,

se $f(n)$ è in $O(g(n))$ allora anche $k f(n)$ è in $O(g(n))$.

1B: Per ogni $k > 0$ e per ogni $f(n) \geq 0$,

se $f(n)$ è in $\Omega(g(n))$ allora anche $k f(n)$ è in $\Omega(g(n))$.

1C: Per ogni $k > 0$ e per ogni $f(n) \geq 0$,

se $f(n)$ è in $\Theta(g(n))$ allora anche $k f(n)$ è in $\Theta(g(n))$.

Informalmente, queste tre regole si possono riformulare dicendo che: le costanti moltiplicative si possono ignorare.

Regole sulla commutatività con la somma.

La somma è commutativa rispetto alla notazione asintotica.

2A: Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $O(g(n))$ e $d(n)$ è in $O(h(n))$

allora $f(n)+d(n)$ è in $O(g(n)+h(n)) = O(\max(g(n), h(n)))$.

Se prendo la funzione g e h più grande, quella che va più veloce all'infinito, allora posso sostituire la somma con la più grande delle 2.

max: quella che va più velocemente.

2B: Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $\Omega(g(n))$ e $d(n)$ è in $\Omega(h(n))$

allora $f(n)+d(n)$ è in $\Omega(g(n)+h(n)) = \Omega(\max(g(n), h(n)))$.

2C: Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $\Theta(g(n))$ e $d(n)$ è in $\Theta(h(n))$

allora $f(n)+d(n)$ è in $\Theta(g(n)+h(n)) = \Theta(\max(g(n), h(n)))$.

Informalmente: le notazioni asintotiche commutano con l'operazione di somma.

Regole sulla commutatività col prodotto.

3A: Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $O(g(n))$ e $d(n)$ è in $O(h(n))$

allora $f(n)d(n)$ è in $O(g(n)h(n))$.

3B: Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $\Omega(g(n))$ e $d(n)$ è in $\Omega(h(n))$

allora $f(n)d(n)$ è in $\Omega(g(n)h(n))$.

3C: Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $\Theta(g(n))$ e $d(n)$ è in $\Theta(h(n))$

allora $f(n)d(n)$ è in $\Theta(g(n)h(n))$.

Informalmente: le notazioni asintotiche commutano con l'operazione di prodotto.

Esempi di applicazione delle regole.

Esempio 1

Trovare il limite asintotico stretto per $f(n) = 3n^2 + 7$.

$$3n^2 = \Theta(n^2) \text{ e } 7 = \Theta(n^0) \text{ quindi } 3n^2 + 7 = \Theta(n^2).$$

Esempio 2

Trovare il limite asintotico stretto per $f(n) = 3n2^n + 4n^4$

$$3n2^n + 4n^4 = \Theta(n)\Theta(2^n) + \Theta(n^4) = \Theta(n2^n) + \Theta(n^4) = \boxed{\Theta(n2^n)} = \Theta(\max\{\dots\})$$

Quella che va più veloce all'infinito è l'esponenziale con la n davanti.

Esempio 3

Trovare il limite asintotico stretto per $f(n) = 2^{n+1}$

$$2^{n+1} = \boxed{2}2^n = \Theta(2^n).$$

Costante, può essere ignorata.

Esempio 4

Trovare il limite asintotico stretto per $f(n) = 2^{2n}$

$$2^{2n} = 2^n 2^n = \Theta(2^n)\Theta(2^n) = \Theta(2^{2n}).$$

💡 le costanti moltiplicative si possono ignorare solo se non sono all'esponente.

Esempio 5

Trovare il limite asintotico stretto per $f(n) = \log n + 8 \cdot 2^{n \log n} + 3$

$$2^{n \log n} \Rightarrow (2^{\log n})^n \Rightarrow n^n$$

$$\begin{aligned} \log n + 8 \cdot 2^{n \log n} + 3 &= \log n + 8n^n + 3 \\ &= \Theta(\log n) + \Theta(1) \Theta(n^n) + \Theta(1) = \boxed{\Theta(n^n)}. \end{aligned}$$

Quella che va più veloce all'infinito è l'esponenziale.

più grande

Dimostrazione della regola 1A.

Regola:

Per ogni $k > 0$ e per ogni $f(n) \geq 0$,

se $f(n)$ è in $O(g(n))$

allora anche $k f(n)$ è in $O(g(n))$.

Dimostrazione

Per ipotesi, $f(n)$ è in $O(g(n))$ quindi esistono due costanti c ed n_0 tali che:

$$f(n) \leq cg(n) \text{ per ogni } n \geq n_0$$

Ne segue che:

$$kf(n) \leq kc g(n)$$

Questo prova che, prendendo kc come nuova costante c' e mantenendo lo stesso n_0 , $kf(n)$ è in $O(g(n))$.

CVD

Dimostrazione della regola 2A.

Regola:

Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $O(g(n))$ e $d(n)$ è in $O(h(n))$

allora $f(n)+d(n)$ è in $O(g(n)+h(n)) = O(\max(g(n), h(n)))$.

Dimostrazione

Se $f(n)$ è in $O(g(n))$ e $d(n)$ è in $O(h(n))$ allora esistono quattro costanti c' e c'' , n'_0 ed n''_0 tali che:

$$f(n) \leq c'g(n) \text{ per ogni } n \geq n'_0 \text{ e } d(n) \leq c''h(n) \text{ per ogni } n \geq n''_0$$

Allora:

$$f(n) + d(n) \leq c'g(n) + c''h(n) \leq \max(c', c'') (g(n) + h(n))$$

per ogni $n \geq \max(n'_0, n''_0)$ n_0

Da ciò segue che $f(n) + d(n)$ è in $O(g(n)+h(n))$.

Infine:

$$\max(c', c'') (g(n) + h(n)) \leq 2 \max(c', c'') \max(g(n), h(n)).$$

Ne segue che $f(n) + d(n)$ è in $O(\max(g(n), h(n)))$.

CVD

Dimostrazione della regola 3A.

Regola:

Per ogni $f(n), d(n) > 0$,

se $f(n)$ è in $O(g(n))$ e $d(n)$ è in $O(h(n))$

allora $f(n)d(n)$ è in $O(g(n)h(n))$.

Dimostrazione

Se $f(n)$ è in $O(g(n))$ e $d(n)$ è in $O(h(n))$ allora esistono quattro costanti c' e c'' , n'_0 ed n''_0 tali che:

$$f(n) \leq c'g(n) \text{ per ogni } n \geq n'_0 \text{ e } d(n) \leq c''h(n) \text{ per ogni } n \geq n''_0$$

Allora:

$$f(n)d(n) \leq c'c''g(n)h(n) \text{ per ogni } n \geq \max(n'_0, n''_0)$$

Da ciò segue che $f(n)d(n)$ è in $O(g(n)h(n))$.

CVD

Le dimostrazioni delle altre regole, che coinvolgono le notazioni Ω e Θ , sono lasciate per esercizio.

Immettere esercizi, informazioni del libro.

Costanti, logaritmi, radici, polinomi, esponenziali.

3_Costo computazionale.

Tra i criteri di misura di costo uniforme e costo logaritmico ci concentriamo sempre sulla **misura di costo uniforme**. La **misura di costo logaritmico** si usa soltanto quando si fa calcolo scientifico; quindi, quando i dati hanno un numero di cifre significative che è estremamente grande, quando non entra in una parola di memoria.
Le funzioni che esprimono il costo computazionale di un qualsiasi algoritmo saranno delle funzioni particolari: devono essere **positive** e sarà sempre **monotona non decrescente** al crescere della variabile indipendente (dimensione input). Al crescere della dimensione del problema noi avremo che la funzione non può diminuire. Potrebbe non essere vero per valori piccoli o estremamente piccoli, ma dovrà essere vera definitivamente da un certo n_0 in poi.

Vediamo ora come *calcolare effettivamente il costo computazionale* di un algoritmo, adottando il criterio della misura di costo uniforme.

Nota: è ragionevole pensare che il costo computazionale, inteso come funzione che rappresenta il tempo di esecuzione di un algoritmo, sia una funzione monotona non decrescente(cioè che cresce o resta uguale) della dimensione dell'input.

Trovare questo parametro (che rappresenta la dimensione dell'input) è, di solito, abbastanza semplice:

- in un *algoritmo di ordinamento* esso sarà il numero di dati da ordinare; (vettori con insieme di dati da ordinare)
Non importa se questi dati sono piccoli o grandi, ma è importante la quantità.
- in un *algoritmo che lavora su una matrice* sarà il numero di righe e di colonne (quindi, 2 parametri);
- in un *algoritmo che opera su alberi* sarà il numero di nodi, ecc.

In altri casi, invece, l'individuazione del parametro non è banale.

In ogni caso, è necessario stabilire quale sia la variabile, parametro (o le variabili) di riferimento prima di accingersi a calcolare il costo.

Nella stragrande maggioranza dei casi, quando abbiamo a che fare con una struttura dati, in cui vengono memorizzati i dati, diremo che la dimensione dell'input è data dalla quantità dei dati.

La notazione asintotica (valida da un certo n_0 in poi, per valori grandi) viene sfruttata pesantemente per il calcolo del costo computazionale degli algoritmi, quindi - in base alla definizione stessa – tale costo computazionale potrà essere ritenuto valido solo asintoticamente.

In effetti, esistono degli algoritmi che per dimensioni dell'input relativamente piccole hanno un certo comportamento, mentre per dimensioni maggiori un altro.

Pseudocodice.

Per poter valutare il tempo computazionale di un algoritmo, esso deve essere formulato in un modo che sia chiaro, sintetico e non ambiguo (non si deve prestare a più di un'interpretazione).

Si adotta il cosiddetto **pseudocodice**, che è una sorta di linguaggio di programmazione “informale”:

- si usano, come nei linguaggi di programmazione, i costrutti di controllo (for, if then else, while, ecc.);
- si può usare il linguaggio naturale per specificare alcune operazioni;
- si omettono dettagli (ad es. la gestione degli errori), per esprimere solo l'essenza della soluzione.

Non c'è una definizione precisa e specifica di pseudocodice, è una sequenza di istruzioni che vengono date con un linguaggio abbastanza simile a un linguaggio naturale, ma con costrutti di linguaggi di programmazione e con le regole specificate sopra. Quindi, non esiste una notazione universalmente accettata per lo pseudocodice.

In questo corso (come nel libro di testo) useremo spesso i costrutti del Python ma, a volte, potremo usare altre convenzioni intuitive, ad esempio il simbolo \neq per verificare che il contenuto di 2 variabili sia differente.

Costo delle istruzioni.

- Le **istruzioni elementari**: hanno costo **$\Theta(1)$** ;

- operazioni aritmetiche (+, -, \times , $/$, ...),
- lettura del valore di una variabile,

- assegnazione di un valore a una variabile (`ciao = 5`),
- valutazione di una condizione logica su un numero costante di operandi,
- stampa del valore di una variabile, ecc.)

- L'istruzione:

*costante:
se sommo due valori, se ho dati da 10
o dati da milioni ha lo stesso costo*

*if (condizione): costo condizione
+
 se condizione
 istruzione1 } max tempo tra le 2 istruzioni,
 false else:
 istruzione2 caso peggiore*

ha costo pari a:

1. il costo di verifica della condizione (di solito costante), da considerare sempre. Quando si scrive un if si chiede se c'è un valore, se 2 valori sono uguali,... (di solito confronto)
2. più il max tra i costi di istruzione1 e istruzione2, noi siamo interessati al caso peggiore;

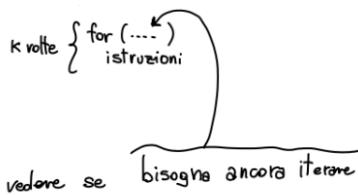
- Le **istruzioni iterative** (cicli for, while e repeat) hanno un costo pari alla somma dei costi di ciascuna delle iterazioni (compreso il costo di verifica della condizione).
- Se tutte le iterazioni hanno lo stesso costo, allora il costo dell'iterazione è pari al prodotto del costo di una singola iterazione per il numero di iterazioni. la condizione viene valutata una volta in più rispetto al numero delle iterazioni, poiché l'ultima valutazione, che dà esito negativo, è quella che fa terminare l'iterazione.

Se il codice viene eseguito n volte il costo sarà:

$$(n + 1)c + ni$$

Dove → c: costo condizione, i: costo iterazione.

Quello che si deve ricordare è che:



Il **costo della condizione** deve essere tenuto in conto una volta in più rispetto ai numeri di cicli che noi eseguiamo. Questo non ci crea problemi, perché anche come l'if, generalmente il costo della condizione è costante; quindi, aggiungere una costante una volta di più o di meno non crea molte differenze.

Il **costo dell'algoritmo** nel suo complesso è pari alla **somma dei costi delle istruzioni che lo compongono**.

Un dato algoritmo potrebbe avere tempi di esecuzione (e quindi costo computazionale) diversi a seconda dell'input. Se io cambio l'input, l'algoritmo può avere un tempo di esecuzione differente.

- casi migliore e peggiore (input particolarmente vantaggioso e, rispettivamente, svantaggioso, ai fini del costo computazionale dell'algoritmo).

Tempo minore: caso migliore | tempo maggiore: caso peggiore.

A parità di dimensione dell'input, fissata una certa dimensione, scopriamo il caso migliore o peggiore.

Dipendenza del costo dall'input.

Per avere un'idea di quale sia il tempo di esecuzione di un algoritmo, a prescindere dall'input: **caso peggiore**. (cioè la situazione che porta alla computazione più onerosa).

Allo stesso tempo, però, vorremmo essere il più precisi possibile e quindi, nel contesto del caso peggiore, cerchiamo di calcolare il costo in termini di notazione asintotica Θ .

Laddove questo non sia possibile, essa dovrà essere approssimata per difetto (tramite la notazione Ω) e per eccesso (tramite la notazione O).

Esempio: calcolo del massimo.

Calcolo del massimo in un vettore disordinato contenente n valori.

def Trova_Max (A):

```

    vettore
    n=len(A)
    max=A[0] # elemento iniziale
    for i in range (1,n):
        if A[i] > max:
            max ← A[i]
    return max

```

$\Theta(1)$ indica un **valore costante**.

$\Theta(1)$: assegnazione

$\Theta(1)$: per la condizione: $i < n-1$

eseguito $(n-1)$ iterazioni più $\Theta(1)$ volte

$\Theta(1)$: confronto costante if: max tro then ed else, qui c'è solo then

$\Theta(1)$ $\Theta(1) + \Theta(1) = \Theta(4)$

$\Theta(1)$; $\Theta(4)$ fuori dal ciclo: $\Theta(4) + \Theta(4) = \Theta(8)$

Detto $T(n)$ il costo computazionale di questo algoritmo: $T(n) = \Theta(1) + [(n-1)\Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$

$\Theta(n) - \Theta(4) + \Theta(2) = \Theta(4) + \Theta(n) + \Theta(4) = \max^5$ teta

Esempio: somma dei primi n interi.

```

def Calcola_Somma_1(n):
    somma = 0
    for i in range (1,n+1):
        somma += i
    return somma

```

$\Theta(1)$: assegnazione costo verifica condizione + incremento i
 $\Theta(1) + \Theta(1) = \Theta(2)$

n iterazioni più $\Theta(1)$

$\Theta(1)$: assegnazione
 $\Theta(1)$

$$T(n) = \Theta(1) + [n \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(n)$$

$\Theta(n) + \Theta(\frac{n}{2}) = \Theta(\frac{n}{2}) + \Theta(\frac{n}{2}) + \Theta(\frac{n}{2}) = \max \Theta$

Osserviamo, tuttavia, che lo stesso problema può essere risolto in modo ben più efficiente come segue:

```
def Calcola_Somma_2(n):  
    somma = n*(n+1)/2  
    return somma
```

Il costo della funzione è, ovviamente, $\Theta(1)$, costo decisamente migliore rispetto al $\Theta(n)$ precedente.

Quando progetto un algoritmo, devo trovare quello più efficiente e non solo quello che risolve il problema.

Esempio: valutazione di un polinomio.

Valutazione del polinomio $\sum_{i=0}^n a_i x^i$ nel punto $x = c$. \rightarrow dimensione input

for i in range(len(A)):

```
[potenza = 1  
termine polinomio i-esima  
for j in range(i):  
    calcolo la potenza i-esima  
    potenza = c*potenza  
    3θ(1) = θ(4)  
somma = somma + A[i]*potenza  
  
return somma
```

Somma = 2

n iterazioni più Θ(1)

Θ(1)

i iterazioni più $\Theta(1)$

$\Theta(1)$

$3\Theta(1) = \Theta(4)$

$\Theta(1)$

array lungo $n+1$

$$T(n) = \Theta(1) + \sum_{i=1..n} (\Theta(1) + \Theta(i) + \Theta(1)) + \Theta(1) = \dots$$

$$T(n) = \underline{\Theta(1)} + \sum_{i=1..n} (\underline{\Theta(1)} + \underline{\Theta(i)} + \underline{\Theta(1)}) + \overline{\Theta(1)} =$$

$$= \Theta(1) + \sum_{i=1..n} (\Theta(1) + \Theta(i)) =$$

$$= \Theta(1) + \sum_{i=1..n} \Theta(1) + \sum_{i=1..n} \Theta(i)$$

$$= \Theta(1) + \Theta(n) + \Theta(n^2)$$

8

avendo usato che

avendo usato che $\sum_{i=1 \dots n} \Theta(i) = n(n+1)/2 = \Theta(n^2)$

Solo riscrivendo in modo più oculato lo pseudocodice, il medesimo problema può essere risolto in modo più efficiente come segue:

```

def Calcola_Polinomio (A, c) :
    somma = A[0]           Θ(1)
    potenza = 1            Θ(1)
    for i in range(1, len(A)) :   n iterazioni più Θ(1)
        potenza = c*potenza      Θ(1)
        somma = somma + A[i]*potenza   Θ(1)
    return somma             Θ(1)

```

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

Il costo computazionale di questa funzione è, ovviamente, $\Theta(n)$, costo decisamente migliore rispetto al $\Theta(n^2)$ precedente.

Costi computazionali e tempi di esecuzione.

Cerchiamo di capire quanto sono grandi effettivamente i tempi di esecuzione di un algoritmo in funzione del suo costo computazionale.

Ipotizziamo di disporre di un sistema di calcolo in grado di effettuare una operazione elementare in un nanosecondo (10⁹ operazioni al secondo), e supponiamo che la dimensione del problema sia $n = 10^6$ (un milione):

- costo $O(n)$ - tempo di esecuzione: 1 millesimo di secondo;
- costo $O(n \log n)$ - tempo di esecuzione: 20 millesimi di secondo;
- costo $O(n^2)$ - tempo di esecuzione: 1000 secondi = 16 minuti e 40 secondi.

Che succede se il costo computazionale cresce esponenzialmente, ad esempio quando è $O(2n)$?

Anche solo su un input di dimensione $n = 100$, l'eventuale algoritmo richiederebbe per la sua soluzione ben $1,26 \cdot 10^{21}$ secondi, cioè circa $3 \cdot 10^{13}$ anni.

un algoritmo con costo esponenziale serve a poco, infatti l'avanzamento tecnologico, seppur formidabile, non è in grado di rendere abbordabile un tale problema.

Infatti, supponiamo di avere un calcolatore estremamente potente che riesce a risolvere un problema di dimensione $n = 1000$, avendo costo computazionale $O(2n)$, in un determinato tempo T , quale dimensione $n' = n + x$ del problema riusciremmo a risolvere nello stesso tempo utilizzando un calcolatore mille volte più veloce?

$$T = \frac{2^{1000} \text{ operazioni}}{10^k \text{ operazioni al secondo}} = \frac{2^{1000+x} \text{ operazioni}}{10^{k+3} \text{ operazioni al secondo}}$$

Si ha quindi:

$$\frac{2^{1000+x}}{2^{1000}} = 2^x = \frac{10^{k+3}}{10^k} = 10^3 = 1000$$

Ossia

$$x = \log 1000 \approx 10$$

Dunque, con un calcolatore mille volte più veloce riusciremmo solo a risolvere, nello stesso tempo, un problema di dimensione 1010 anziché di dimensione 1000!

Cioè: un algoritmo con costo esponenziale serve a poco oggi e servirà a poco domani.

In effetti esiste un'importantissima branca della teoria della complessità che si occupa proprio di caratterizzare i cosiddetti problemi intrattabili, ossia quei problemi il cui costo computazionale è tale per cui essi non né saranno mai risolvibili per dimensioni realistiche dell'input.

Il problema della ricerca.

Nell'informatica esistono alcuni problemi particolarmente rilevanti, poiché essi:

- si incontrano in una grande varietà di situazioni reali;
- appaiono come sotto problemi da risolvere nell'ambito di problemi più complessi.

Uno di questi problemi è la **ricerca di un elemento in un insieme di dati** (ad es. numeri, cognomi, basi di dati di tutti gli studenti della sapienza, anagrafe comune, ecc.).

Definiamo un po' più formalmente tale problema (computazione) descrivendone l'input e l'output:

- **Input:** un **array A** di n valori (interi, stringhe, ecc.) ed un **valore v** omogeneo rispetto ai valori memorizzati;
- **Output:** un **indice i** tale che $A[i] = v$, oppure un particolare valore **null** o **-1** se il valore v non è presente nell'array.

Ricerca sequenziale.

Un primo semplice algoritmo è basato su questa idea:

- Prendo l'array A;
- Ispezioniamo uno alla volta gli elementi dell'array;
- Confrontiamo ciascun elemento con v;
- Restituiamo il risultato, interrompendoci appena (non) trovato v.

Vediamo il caso in cui v non è contenuto nell'array:



Parto da 10, lo confronto con 27, non sono uguali e vado avanti finché non esaurisco l'array oppure finché non trovo il valore cercato.

```
def Ricerca (A;v):  
    i = 0  
    while ((i<len(A)) and (A[i]!=v)):  
        i += 1  
        if (i < len(A)):  
            return i  
        else:  
            return -1
```

Sostanzialmente nell'algoritmo abbiamo un indice i. Esco dal while in ogni caso. Quando uscirò se i sarà maggiore della lunghezza dell'array l'elemento che cerchiamo non sarà presente(e tornerò -1) altrimenti sarà presente e lo ritornerò.

```
Funzione Ricerca (A;v)  
i = 1  
while ((i<len(A)) and (A[i]!=v)) Θ(1) + al più n volte:  
    i += 1. Θ(1)  
    if (i < len(A)): return i Θ(1)  
    else: return null Θ(1)
```

Questo algoritmo ha un *costo computazionale* di:

Il costo dell'algoritmo dipende dall'input e avremo:

- **Θ(n)** nel **caso peggiore** (quando, cioè, v non è contenuto nel vettore: devo scorrere l'intero array)
- **Θ(1)** nel **caso migliore** (quando v viene incontrato per primo e il ciclo while non viene mai eseguito)

Poiché non abbiamo trovato una stima del costo che sia valida per tutti i casi, diremo che il costo computazionale dell'algoritmo (in generale, non nel caso peggiore) è un **O(n)** o **Ω(1)**, per evidenziare il fatto che ci sono input in cui questo valore viene raggiunto, ma ci sono anche input in cui il costo è minore.

Sia il caso peggiore che caso migliore corrispondono a una dimensione dell'input che è comunque n.

Quando distinguiamo tra caso migliore e caso peggiore dobbiamo mantenere la stessa identica dimensione dell'input.

Il caso peggiore che chiamo T(n) sarà \geq del caso migliore e \leq del caso peggiore.

Quando non è possibile determinare un valore stretto per il costo computazionale, perché i casi migliore e peggiore si discostano, possiamo domandarci quale sia il costo computazionale dell'algoritmo nel **caso medio**: media di tutti i possibili costi e vediamo se il caso medio sia più vicino al caso peggiore e al caso migliore. Non è sempre fattibile ed è laborioso.

Facciamo l'ipotesi che v possa apparire con uguale probabilità in qualunque posizione, ossia che: stiamo assumendo che v lo troviamo nell'array e lo possiamo trovare in qualsiasi posizione (equiprobabile). Se v si trova in posizione i -esima il costo dell'algoritmo sarà $\theta(i)$.

$$P(v \text{ si trova in } i\text{-esima posizione}) = \frac{1}{n}$$

Allora il numero medio di iterazioni del ciclo è dato da:

$$\frac{1}{n} \sum_{i=1}^n i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Il numero medio di iterazioni è dato dal costo di tutti i costi possibili diviso tutte le possibilità.

Nel caso speciale in cui l'elemento sicuramente esiste in media avrà un costo computazionale dell'ordine di $n/2$. A livello di crescita asintotica ci avviciniamo al caso peggiore: $\Theta(n)$.

Un'ipotesi alternativa è la seguente: supponiamo che tutte le possibili $n!$ permutazioni della sequenza di n numeri siano **equiprobabili** (disordinato su una delle n permutazioni).

Di queste, ve ne saranno un certo numero nelle quali v appare in **prima** posizione, un certo numero nelle quali v appare in **seconda** posizione, v in **i -esima** posizione.

Il numero medio di iterazioni sarà quindi:

$$\begin{aligned} \text{Numero medio di iterazioni} &= \\ &= \sum_{i=1}^n i \frac{\text{numero di permutazioni in cui } v \text{ è in posizione } i}{\text{numero totale di permutazioni}} \end{aligned}$$

Ora, il numero di permutazioni nelle quali v appare nella i -esima posizione è uguale al numero delle permutazioni di $(n-1)$ elementi, cioè $(n-1)!$, dato che fissiamo solo la posizione di uno degli n elementi.

Quindi:

$$\text{Numero medio di iterazioni} = \sum_{i=1}^n i \frac{(n-1)!}{n!} = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$$

💡 Facendo due ipotesi di equi probabilità diverse, abbiamo trovato lo stesso risultato.

Questa non è una dimostrazione.

Se il nostro input è abbastanza equiprobabile ci aspettiamo che: il costo computazionale del caso medio sia un $\Theta(n)$.

Abbiamo fatto l'ipotesi forte che l'elemento v esiste sempre all'interno del vettore, questo non è sempre; quindi, il nostro costo potrà essere soltanto peggio.

In mancanza di queste ipotesi nel nostro input, non abbiamo possibilità di dimostrare un valore per il caso medio del costo computazionale, ma questo ci aiuta ad avere un'idea.

⚠ ATTENZIONE:

L'operatore `in` del Python che verifica se un elemento è contenuto in un oggetto di tipo list utilizza una ricerca sequenziale, perciò ha costo computazionale $O(n)$.

Ad esempio, il seguente frammento di codice:

```
...
if v in A:
    print('presente')
else:
    print('assente')
...
```

ha costo $O(n)$ nonostante non comprenda (apparentemente!) alcuna istruzione iterativa!

Cioè l'`in` è una specie di funzione che utilizza la ricerca sequenziale.

Ricerca binaria.

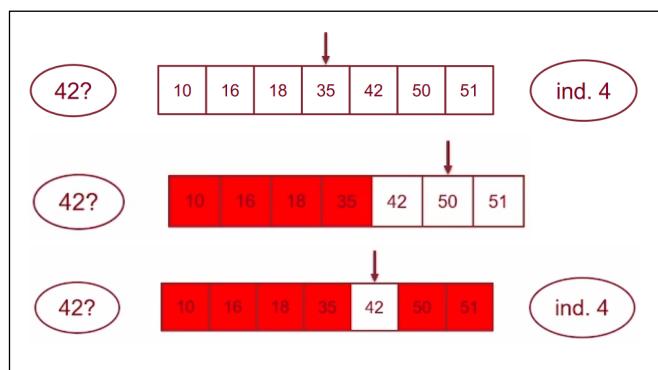
Ma nella vita di tutti i giorni, non cerchiamo così: l'algoritmo della ricerca binaria funziona come segue:

- Si ispeziona l'**elemento centrale** della sequenza:
- se esso è uguale al valore cercato ci si ferma;
- se il valore cercato è più piccolo si prosegue nella sola metà inferiore della sequenza, altrimenti nella sola metà superiore.

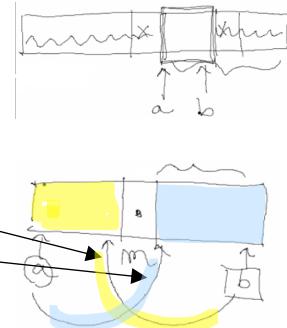
Così, ad ogni iterazione si **dimezza** il numero degli elementi su cui proseguire l'indagine ed il numero di iterazioni cresce come **$\log n$** e quindi cresce con estrema lentezza.

Perciò, ad esempio, per trovare un elemento in una sequenza ordinata di un miliardo di valori bastano circa 30 Iterazioni (\log di un miliardo di valori) !

Vediamo un esempio nel quale l'elemento cercato (42) è presente.



```
def Ricerca_binaria(A, v):
    a, b = 0, len(A) - 1
    m = (a+b)//2
    while (A[m] != v):
        if (A[m] > v):
            b = m - 1
        else:
            a = m + 1
        if a > b:
            return -1
        m = (a+b)//2
    return m
```



Ovviamente questo vale se il mio array è già ordinato.

Il ciclo `while` viene eseguito al più $\Theta(\log n)$ volte; pertanto, il costo computazionale è:

- **$\Theta(\log n)$** nel **caso peggiore** (l'elemento non c'è);
- **$\Theta(1)$** nel **caso migliore** (l'elemento si trova al primo colpo).

Poiché caso migliore e caso peggiore non hanno lo stesso costo, valutiamo il costo computazionale del **caso medio...**

Assunzioni:

- il numero di elementi è una potenza di 2 (per semplicità di calcolo, ma è facile vedere che questa assunzione non modifica in alcun modo il risultato finale);
- v è presente nella sequenza (altrimenti si ricade nel caso peggiore);
- tutte le posizioni di v fra 1 e n sono equiprobabili.

Quante siano le posizioni $n(i)$ raggiungibili alla i -esima iterazione?

- con una iterazione si raggiunge la sola posizione $n/2$, cioè **$n(1)=2^0=1$** ;
 - con esattamente due iterazioni si raggiungono le due posizioni $\frac{n}{4}$ e $3\frac{n}{4}$, cioè **$n(2)=2^1=2$** ;
 - con esattamente tre iterazioni si raggiungono le quattro posizioni $\frac{n}{8}, 3\frac{n}{8}, 5\frac{n}{8}, 7\frac{n}{8}$, cioè **$n(3)=2^2=4$** ;
- e così via.



In generale, la ricerca binaria esegue i iterazioni se v si trova in una delle $n(i) = 2^{i-1}$ posizioni raggiungibili con esattamente tale numero di iterazioni.

La probabilità che l'elemento da trovare si trovi su una delle $n(i)$ posizioni toccate dalla i -esima iterazione è $n(i)/n$ (ossia $2^{i-1}/n$), perciò il numero medio di iterazioni è:

$$\text{numero medio di iterazioni} = \sum_{i=1}^{\log n} \frac{n(i)}{n} = \frac{1}{n} \sum_{i=1}^{\log n} i 2^{i-1}$$

$$\text{dove } \sum_{i=1}^k i 2^{i-1} = (k-1)2^k + 1$$

per cui:

$$\frac{1}{n} ((\log n - 1)2^{\log n} + 1) = \log n - 1 + \frac{1}{n}$$

Ossia, il numero medio di iterazioni si discosta per meno di un confronto dal numero massimo di iterazioni!

Il caso medio siamo più vicini al caso peggiore che al caso migliore.

Si riesce a fare meno di $\log n$? Vediamo questo con una soluzione di biologia.

Ricerca costante.

Problema: Data una sequenza di DNA S ed un certo frammento F, F è presente in S?

Ibridizzazione: processo che unisce due eliche singole di DNA in un'unica elica doppia (frammenti di singole eliche si attraggono e formano un'elica doppia se essi sono uno il complementare dell'altro).

Sonda: piccolo frammento di elica singola di DNA che ha una sequenza nota ed è fluorescente.

L'ibridizzazione di una sonda in un frammento di DNA sconosciuto (= un singolo passo computazionale su una sorta di ipotetica macchina molecolare) evidenzia la presenza della sequenza complementare a quella della sonda nel frammento.

Cioè: algoritmo di ricerca con costo costante se consideriamo un diverso modello computazionale...

Esercizi per casa: vedere 2:00- 2:08.

La ricorsione.

Consideriamo una diversa formulazione dell'algoritmo di ricerca binaria, nella quale sono volutamente tralasciati i dettagli per catturarne l'essenza:

```
Ricerca_binaria (A, v)
  se A è vuoto
    restituisci -1
  ispeziona l'elemento A[centrale]
  se esso è uguale a v
    restituisci il suo indice
  se v < A[centrale]
    esegui Ricerca_binaria (metà sinistra di A, v)
  se v > A[centrale]
    esegui Ricerca_binaria (metà destra di A, v)
```

```
Ricerca_binaria (A, v)
  se A è vuoto
    restituisci -1
  ispeziona l'elemento A[centrale]
  se esso è uguale a v
    restituisci il suo indice
  se v < A[centrale]
    esegui Ricerca_binaria (metà sinistra di A, v)
  se v > A[centrale]
    esegui Ricerca_binaria (metà destra di A, v)
```

L'aspetto cruciale di questa formulazione risiede nel fatto che l'algoritmo risolve il problema "riapplicando" sé stesso su un sotto problema (una delle due metà dell'array). L'input deve cambiare ad ogni chiamata della funzione per arrivare al caso base.

Questa tecnica si chiama **ricorsione**.

Funzioni matematiche ricorsive.

Una funzione matematica è detta **ricorsiva** quando la sua definizione è espressa in termini di sé stessa.

Esempio: $n! = n * (n - 1)!$ Se $n > 0$
 $0! = 1$

prima riga: meccanismo di calcolo ricorsivo

seconda riga: caso base

💡 una funzione matematica ricorsiva deve sempre avere un **caso base**!

Algoritmi ricorsivi.

Nel campo degli algoritmi vi è un concetto del tutto analogo, quello degli algoritmi ricorsivi:

un **algoritmo** è detto **ricorsivo** quando è espresso in termini di sé stesso.

Un algoritmo ricorsivo ha sempre queste proprietà:

- la soluzione del problema complessivo è costruita risolvendo (ricorsivamente) uno o più sotto problemi di dimensione minore e combinando poi queste soluzioni;
- la successione dei sotto problemi, che sono sempre più piccoli, deve sempre convergere ad un sotto problema che costituisca un caso base (detto anche condizione di terminazione), incontrato il quale la ricorsione termina.

Esempio: **Calcolo del fattoriale di un intero dato n** $\rightarrow n! = n \cdot (n-1)!$

Abbiamo una versione iterativa e la versione ricorsiva che ci interessa.

```
def Fattoriale_Iter (n): // n: intero non negativo
    fatt = 1
    for i in range(1, n+1):
        fatt = fatt*i
    return fatt
```

```
def Fattoriale_Ric (n): // n : intero non negativo
    if (n == 0) return 1
    return n*Fattoriale_Ric (n - 1)
```

- Se una funzione ricorsiva non ha un caso base, la sua esecuzione genererà una catena illimitata di chiamate ricorsive che non terminerà mai.

💡 Accertiamoci che il caso base non possa essere mai “saltato” durante l’esecuzione.

- Nulla vieta di prevedere più di un caso base, basterà assicurarsi che **uno tra i casi base sia sempre e comunque incontrato**, cioè per qualsiasi input si va a finire in un caso base.

Esempio: **algoritmo ricorsivo per la ricerca sequenziale su n elementi:**

v è l’elemento che dobbiamo cercare.

- ispezioniamo l’n-esimo elemento;
- se non è l’elemento cercato risolviamo il problema ricorsivamente sui primi (n – 1) elementi.

```
caso base ←
def Ricerca_seq_ric(A, v, n=len(A)-1):
    if (A[n]==v): return n
    if (n==0) return -1
    else return Ricerca_seq_ric(A, v, n - 1)
```

Ci possiamo fermare già da subito, cioè il primo elemento ispezionato (cioè l’ultimo in questo caso) è quello che cercavamo oppure nel caso peggiore la funzione sarà richiamata quanto è lungo l’array.

Ad ogni sottopasso considero il sotto array.

Possiamo partire anche dal primo elemento e non dall’ultimo, ma cambierebbero alcune cose.

Esempio: **algoritmo ricorsivo per la ricerca binaria su n elementi:**

- ispezioniamo l’elemento centrale del vettore;
- se non è l’elemento cercato risolviamo il problema ricorsivamente sulla prima oppure sulla seconda metà del vettore a seconda dell’elemento centrale.

```
def Ricerca_bin_ric (A, v, i_min=0, i_max=len(A)):
    if (i_min > i_max): return -1 → Array dimensione 0
    m =(i_min + i_max)//2
    if (A[m] = v): return m
    elif (A[m] > v):
        return Ricerca_bin_ric (A, v, i_min, m - 1)
    else:
        return Ricerca_bin_ric (A, v, m + 1, i_max)
```

Anche nella formulazione ricorsiva della ricerca binaria ogni nuova chiamata ricorsiva riceve un sotto-problema da risolvere la cui dimensione è circa la metà di quello originario quindi, come nella versione iterativa, si giunge molto rapidamente al caso base.

Nel caso migliore avremo un costo $\Theta(1)$, nel caso peggiore un $\Theta(\log n)$, in generale abbiamo:

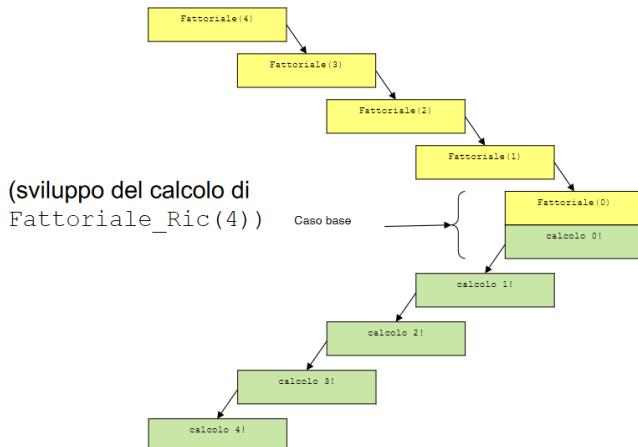
Da ciò deriva un costo computazionale che, come vedremo, è **$O(\log n)$** nel caso peggiore come per la ricerca binaria iterativa.

Quando implemento un determinato algoritmo sia nella forma iterativa che nella forma ricorsiva il costo computazionale è lo stesso.

La ricorsione.

Ripassiamo come si sviluppa il procedimento di calcolo effettivo di una funzione ricorsiva sull'esempio del fattoriale:

```
def Fattoriale_Ric (n)
    if (n == 0): return 1
    return n*Fattoriale_Ric (n - 1)
```



Quando eseguo la funzione Fattoriale_Ric(4) richiamo subito la funzione Fattoriale_Ric(3), in questo passaggio c'è una certa quantità di informazioni che è copiata da una parte, lasciata in attesa.

Il Fattoriale_Ric(3) richiamo Fattoriale_Ric(2), stessa cosa... fino a quando non arriviamo a Fattoriale_Ric(0). Siamo giunti al caso base e quindi la funzione si chiude e ritorna il valore 1.

A questo punto viene richiamata dalla memoria l'ultima funzione che era stata aperta, cioè Fattoriale_Ric(1) e qui il valore è 1.

Qui chiudo anche questa funzione e ritorno 1*1.

Ora riprendo Fattoriale_Ric(2) che n=2, e Fattoriale_Ric(1) = 1, quindi la funzione viene chiusa e ritorno 2*1.

Si continua così fino all'ultima chiamata che restituisce il valore corretto.

Le *chiamate ricorsive* in generale **richiedono tantissima memoria**.

💡 qualsiasi problema risolvibile con un algoritmo ricorsivo può essere risolto anche con un algoritmo iterativo.

In quali situazioni è consigliabile utilizzare un algoritmo ricorsivo anziché uno iterativo?

- **meglio ricorsivo** quando si può *formulare la soluzione del problema in un modo che è chiaro ed aderente alla natura del problema stesso*, mentre la soluzione iterativa è molto più complicata o addirittura non evidente;
- **meglio iterativo** quando *esiste una soluzione iterativa altrettanto semplice e chiara*;
- **meglio iterativo** quando *l'efficienza è un requisito primario*.

Ogni funzione, sia essa ricorsiva o no, richiede per la sua esecuzione una certa quantità di memoria, per:

- caricare in memoria il suo codice;
- passare i parametri e vengono copiate;
- memorizzare i valori delle sue variabili locali.

Quindi le *funzioni ricorsive* in generale hanno **maggiori esigenze**, in termini di **memoria**, delle funzioni non ricorsive (dette anche funzioni iterative).

(costo computazione in termini di spazio)

Per inciso, questa è la ragione per la quale una funzione **ricorsiva mal progettata**, ad es. nella quale la condizione di terminazione non si incontra mai, esaurisce con estrema rapidità la memoria disponibile con la conseguente inevitabile terminazione forzata dell'esecuzione. (problemi di eliminazione forzata)

Esempio: **calcolo dell'n-esimo numero di Fibonacci**.

La definizione dei numeri di Fibonacci è:

- $F(0) = 0$
- $F(1) = 1$
- $F(i) = F(i - 1) + F(i - 2)$ se $i > 1$



$F(i)$: somma dei due precedenti valori di Fibonacci

Esiste una formula chiusa per il calcolo (formula di Binet), che però introduce *errori numerici* dovuti agli inevitabili arrotondamenti causati dai calcoli in virgola mobile:

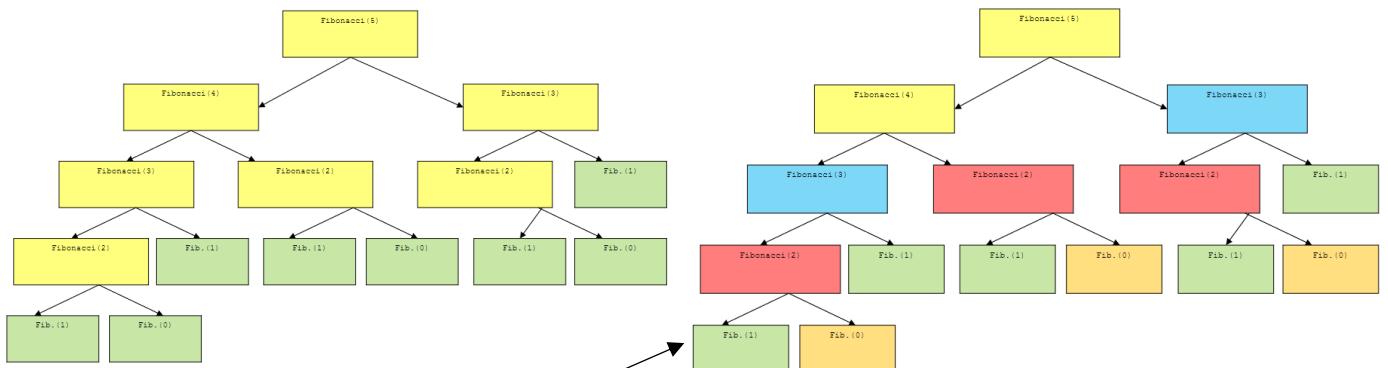
$$F(n) = \frac{\Phi^n}{\sqrt{5}} - \frac{(1-\Phi)^n}{\sqrt{5}} \text{ con } \Phi = \frac{1+\sqrt{5}}{2}$$

... quindi impostiamo il calcolo mediante la seguente funzione ricorsiva:

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

caso base

⚠ nel corpo della funzione, ci sono due chiamate ricorsive, non una sola come nel caso della ricerca binaria. Questo si riflette in una più complessa articolazione delle chiamate di funzione: (costo computazionale più alto).



Osservazione 1: uno stesso calcolo viene ripetuto più volte 🤦

Osservazione 2: L'occupazione dello spazio di memoria, legata alle chiamate di funzione, è molto alta:

- Il numero totale delle chiamate effettuate dall'inizio alla fine dell'elaborazione cresce molto velocemente con n.
- Inoltre, quando si arriva a ciascun caso base vi è una catena di chiamate "aperte" lungo il percorso che va dalla chiamata iniziale al caso base in questione.

Osservazione 3: Per risolvere un problema di dimensione n si devono risolvere due sotto problemi di dimensione ben poco inferiore (rispettivamente, n-1 ed n-2).

La versione iterativa della funzione per il calcolo del numero di Fibonacci:

```

def Fibonacci_iter(n):
    if (n <= 1): return n
    fib_prec_prec = 0
    fib_prec = 1
    for i in range(2,n+1):
        fib_prec_prec, fib_prec= fib_prec,
                                         fib_prec_prec + fib_prec
    return fib_prec
  
```

ha **costo** in tempo pari a $\Theta(n)$ ed in [spazio pari a $\Theta(1)$] → 3 variabili intere.

Qual è il costo computazionale della versione ricorsiva?

Funzione Fibonacci (n)	
if ($n \leq 1$)	$\leftarrow \Theta(1)$
return n	$\leftarrow \Theta(1)$
return ($\text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$)	$\leftarrow T(n-1) + T(n-2)$

Ottieniamo un costo $T(n) = \Theta(1) + T(n-1) + T(n-2)$ che è simile alla formula che definisce i numeri di Fibonacci, per cui prevediamo sia un **costo esponenziale**. Ma come lo dimostriamo? Con le **equazioni di ricorrenza** (in seguito) Nei casi in cui n è 0 oppure 1 allora il costo computazionale è $\Theta(1)$.

Dati in input due interi n e k, calcolare la potenza k-esima di n.

💡 possiamo basarci sul fatto che $n^k = n \cdot n^{k-1}$ ed $n^0 = 1$

def Enne_alla_Kappa (n, k)	
if (k=0): return 1	$\Theta(1) + \Theta(1)$
return n*Enne_alla_Kappa (n, k-1)	$\Theta(1) + T(n, k-1)$

$n^k = n \cdot n \cdot n \dots \cdot n$ per k volte.

Costo:

Se $k=0$: $T(n,k)=\Theta(1)$

Se $k>0$: $T(n,k)=\Theta(1)+T(n,k-1)$

💡 n è un parametro inutile ai fini del costo computazionale, se cambia non succede niente al costo computazionale, quindi $\rightarrow T(0) = \Theta(1)$, $T(k) = \Theta(1) + T(k - 1)$, in questo caso non possiamo eliminare $\Theta(1)$.

Dato in input un array di n interi (con indici da 1 a n), calcolare la somma dei suoi elementi.

💡 la somma è pari al valore di un elemento più la somma calcolata ricorsivamente sul resto dell'array.

In questo caso partiamo dall'ultimo elemento per evitare di portarci un indice in più, ma possiamo anche partire dal primo elemento. Non utilizzare slicing perché avrebbe un costo alto.

Se $n=1$ restituisco l'ultimo elemento, altrimenti restituisco la somma dell'ultimo elemento più la chiamata ricorsiva.

Ad ogni chiamata ricorsiva c'è il valore che viene ritornato in quel momento, come abbiamo visto per il fattoriale, per questo alla fine avremo il risultato corretto.

```
def SommaVet (V, n)
    if (n==1) return V[n]
    return (V[n] + SommaVet (V, n-1))
```

Caso base:

$$T(1) = \Theta(1)$$

Costo:

Se $n=1$: $T(n)=\Theta(1)$

Se $n>1$: $T(n)=\Theta(1)+T(n-1)$

Da questo algoritmo ci aspettiamo un costo di $\Theta(n)$.

Tutte queste equazioni di ricorrenza le teniamo da parte e impareremo i metodi per risolverle. Con le equazioni di ricorrenza avremo i veri costi.

Dato in input un array di n interi, trovare il minimo.

💡 il minimo è pari al minore fra il valore di un elemento e il minimo trovato ricorsivamente sul resto dell'array.

```
def MinVet (V, n=len(V))
    if (n==1):
        return V[n]
    else :→ return Min(V[n], MinVet (V, n-1))
```

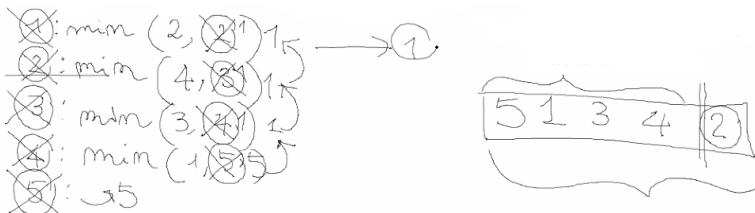
OSSERVAZIONE:

Non è lo stesso
eliminare l'elemento
più a destra e quello
più a sinistra...

Costo:

Se $n=1$: $T(n)=\Theta(1)$

Se $n>1$: $T(n)=\Theta(1)+T(n-1)$



n in questo caso è 5.

Nella prima chiamata calcolo il minimo tra l'ultimo elemento (2) e il risultato di una nuova chiamata.

La seconda chiamata entra con $n=4$, calcolo il minimo tra 4 e la chiamata ricorsiva 3. Così fino alla quinta chiamata che entra con $n=1$, in questo caso per l'if ritorno l'elemento $V[n]$ che è 5.

Il controllo ripassa alla chiamata quarta la quale al posto della chiamata 5 si ritrova il valore ritornato, quindi 5. A questo punto viene calcolato il minimo tra 1 e 5, che è 1. Si chiude la quarta chiamata e si passa alla terza chiamata che al posto della quarta chiamata si ritrova il valore ritornato 1. Quindi si ricalcola il minimo tra 3 e 1 che è 1. Si chiude questa chiamata e si va alla chiamata seconda dove avviene la stessa cosa.

CI ritroveremo alla prima chiamata con un ritorno di 1 che è il minimo.

Dato in input un array di n interi, verificare se è **palindromo**.

Esempio di vettore palindromo:

1	5	4	8	4	5	1
---	---	---	---	---	---	---

💡 un array è palindromo se i suoi estremi sono uguali e il resto del vettore è anch'esso palindromo.

Nota: se n è dispari c'è un elemento centrale, altrimenti no; questa differenza va gestita.

Se n è pari i due indici si scavalcano, se n è dispari andranno a convergere sullo stesso elemento centrale.

Non funziona se partiamo dall'elemento centrale.

```
def Palin (V, i_min=0, i_max=len(A)-1)
    if (i_max <= i_min):
        return true
    if (V[i_min] != V[i_max]):
        return false
    return ((V[i_min]==V[i_max])AND Palin(V, i_min+1,i_max-1))
```

<-Caso base

💡 dim dell'input $n = i_{\text{max}} - i_{\text{min}} + 1$

Costo:

Se $n \leq 1$: $T(n) = \Theta(1)$

Se $n > 1$: $T(n) = \Theta(1) + T(n-2)$

Se i due elementi già dal primo ed ultimo elementi non sono uguali allora sicuramente l'array non è palindromo, altrimenti possiamo continuare nei valori più interni.

Nell'algoritmo possiamo anche eliminare: if ($V[i_{\text{min}}]$ diverso da $V[i_{\text{max}}]$): return false e tenere il return sotto completo, altrimenti possiamo tenere l'if ed eliminare nel return $V[i_{\text{min}}] == V[i_{\text{max}}]$.

Dato in input un array di n interi, stampare le **chiavi dall'ultima alla prima**, ossia nell'ordine:

$V[n-1] V[n-2] V[n-3] \dots V[2] V[1] V[0]$

💡 stampiamo l'ultima chiave e chiamiamo ricorsivamente la funzione sul resto dell'array, gestendo opportunamente il secondo parametro.

```
def StampaArray (V, n: len(A)-1)
    stampa (V[n])
    if (n > 1)
        StampaArray (V, n-1)
```

Costo: Se $n=1$: $T(n) = \Theta(1)$
Se $n > 1$: $T(n) = \Theta(1) + T(n-1)$

Caso base $\rightarrow n = 1$.

E se invece vogliamo stamparle **dalla prima all'ultima**?

💡 dobbiamo fare in modo che le stampe avvengano solo durante la risalita dalle chiamate ricorsive, dopo aver incontrato il caso base per $n = 1$. Basta scambiare l'ordine della stampa rispetto alla chiamata ricorsiva:

```
def StampaArray_2 (V, n: len(A)-1)
    if (n > 1):
        StampaVet (V, n-1)
        stampa (V[n])
```

OSSERVAZIONE:

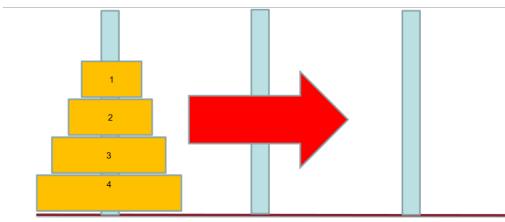
Non va bene stampare
una chiave e poi
richiamare sulla parte
destra dell'array...

Costo: Se $n=1$: $T(n) = \Theta(1)$
Se $n > 1$: $T(n) = \Theta(1) + T(n-1)$

La ricorsione – esercizio 7: Torri di Hanoi (1).

In questo classico gioco si deve spostare una torre di dischi da un piolo ad un altro, con questi vincoli:

- In una mossa si può muovere un solo disco.
- È vietato spostare un disco più grande sopra un disco più piccolo. Quello più piccolo su quello più grande.

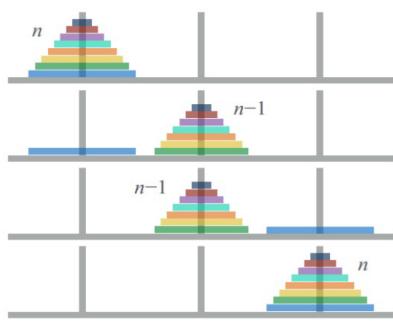


Strategia ricorsiva: per spostare una torre di n dischi, dal piolo i al j , si deve:

1. Spostare la torre degli $n-1$ dischi più piccoli da i a k ;
2. Spostare il disco n -esimo (il più grande) da i a j ;
3. Spostare la torre di $n-1$ dischi da k a j .

I passi 1. e 3. sono adatti all'uso della **ricorsione**, il passo 2 invece lo consideriamo un'operazione **elementare**.

💡 per spostare una torre di n dischi bisogna spostare due torri di $n-1$ dischi, il che ci fa capire che il numero di mosse cresce esponenzialmente con n .



[Disegno di Ozalp Babaoglu, Università di Bologna]

```
def SpostaTorreH (n, i, j)
    if (n=0) return
    SpostaTorreH (n-1, i, 3-(i+j))
    SpostaDisco (i,j)
    SpostaTorreH (n-1, 3-(i+j), j)
```

- 📌 1. I pioli sono numerati da 0 a 2;
- 2. I dischi sono numerati dal più piccolo (1) al più grande (n)
- 3. La funzione SpostaTorreH, quindi, agisce sui dischi dall'alto.

Equazione di ricorrenza dello pseudocodice → $T(n) = \theta(1) + 2T(n - 1)$ (costo esponenziale)

Equazioni di ricorrenza.

Il costo computazionale di un algoritmo ricorsivo è sempre un'equazione di ricorrenza.

- Valutare il costo computazionale di un algoritmo ricorsivo è, in generale, più laborioso che nel caso degli algoritmi iterativi.
- Infatti, la natura ricorsiva della soluzione algoritmica dà luogo a una funzione di costo che, essendo strettamente legata alla struttura dell'algoritmo, è anch'essa ricorsiva. (ad esempio $T(n) = \theta(1) + T(n - 1)$)
- Trovare la funzione di costo ricorsiva è piuttosto immediato. Essa però deve essere risolta, altrimenti il costo asintotico non può essere quantificato, e questa è la parte meno semplice.
- La *funzione matematica ricorsiva che esprime il costo* è anche detta [equazione di ricorrenza](#).

Esempio: costo computazionale della ricerca sequenziale ricorsiva:

```
Funzione Ricerca_seq_ric(A: vet.; v: int.; n: int.)
    if (A[n] = v)
        return true
    else
        if (n = 1)
            return false
        else
            return Ricerca_seq_ric (A, v, n-1)
```

Caso base

In generale: $T(n) = \theta(1) + T(n - 1)$, Caso base: $T(1) = \theta(1)$

Anche se avessimo solo: $\text{return Ricerca_seq_ric (A, v, } \uparrow \text{n-1)}$ avremo anche in questo caso il $\Theta(1)$ per il return e per l'assegnamento n-1.

- La parte generale dell'equazione di ricorrenza che definisce $T(n)$ deve essere sempre costituita dalla somma di almeno due addendi, di cui almeno uno contiene la **parte ricorsiva** (nell'esempio $T(n-1)$) mentre uno rappresenta il **costo computazionale di tutto ciò che viene eseguito al di fuori** della chiamata ricorsiva (non ricorsivo).
- Anche se questa parte dovesse essere un solo confronto, il suo costo non può essere ignorato, altrimenti si otterrebbe che la funzione ha un costo computazionale indipendente dalla dimensione del suo input, e questa è data da quanto illustrato nel caso base. Non possiamo dire che ciò sia impossibile, ma è molto improbabile.
- ◆ Deve sempre essere presente un **caso base**.

Esistono alcuni *metodi utili* per **risolvere** le equazioni di ricorrenza, che illustreremo:

- Metodo **iterativo**;
- Metodo di **sostituzione**;
- Metodo **dell'albero**;
- Metodo **principale**;

Non tutti i metodi si riescono ad usare in ogni occasione; perciò, sono tutti importanti e tutti da imparare.

Metodo iterativo.

- **Idea:** sviluppare l'equazione di ricorrenza (sostituirla dentro sé stessa) ed esprimere come somma di tanti termini dipendenti da n e dal caso base.
- **Difficoltà:** maggiore quantità di calcoli algebrici rispetto agli altri metodi.

Esempio: Equazione relativa alla ricerca seq. ricorsiva

$$\begin{aligned} T(n) &= T(n-1) + \Theta(1) \\ T(1) &= \Theta(1) \end{aligned}$$

$$\begin{aligned} T(n) &= T(1) + (n-1)\Theta(1) = \\ &= \Theta(1) + (n-1)\Theta(1) = \\ &= n\Theta(1) \end{aligned}$$

• $T(n) = T(n-1) + \Theta(1)$, ma $T(n-1) = T(n-2) + \Theta(1)$

Sostituiamo:

• $T(n) = T(n-2) + \Theta(1) + \Theta(1)$, ma $T(n-2) = T(n-3) + \Theta(1)$

• $T(n) = T(n-3) + \Theta(1) + \Theta(1) + \Theta(1)$, ma $T(n-3) = T(n-4) + \Theta(1)$

• $T(n) = T(n-4) + \Theta(1) + \Theta(1) + \Theta(1) + \Theta(1)$, ecc.

fino a ottenere: $T(n) = T(n-k) + k\Theta(1)$ finché $n-k=1$

$$T(n) = n\Theta(1) = \Theta(n).$$

I $\Theta(1)$ non si possono semplificare, non si possono considerare come una costante.

Esempio: Equazione relativa alla ricerca binaria ricorsiva

$$\begin{aligned} T(n) &= T(n/2) + \Theta(1) \\ T(1) &= \Theta(1) \end{aligned}$$

• $T(n) = T(n/2) + \Theta(1)$ ma $T(n/2) = T(n/2^2) + \Theta(1)$

Sostituiamo:

• $T(n) = T(n/2^2) + \Theta(1) + \Theta(1)$ ma $T(n/2^2) = T(n/2^3) + \Theta(1)$

• $T(n) = T(n/2^3) + \Theta(1) + \Theta(1) + \Theta(1)$ ma...

$T(n) = T(n/2^k) + k\Theta(1)$ finché $n/2^k = 1 \Leftrightarrow n = 2^k \Leftrightarrow k = \log_2 n$

Quando $k = \log n$ si ha $n/2^k = 1$, quindi per tale valore di k ci fermiamo ed otteniamo:

$$T(n) = \Theta(1) + \log n \Theta(1) = \Theta(\log n).$$

Esempio: Possiamo riscrivere un algoritmo di ricerca `seq` ricorsiva diverso dal precedente (e meno furbo!):

```
def Ric_seq_ric2(A, v, i_min=0, i_max=len(A)):
    if (i_max < i_min): return -1; ← Caso base
    if (i_max==i_min)
        if (A[m]==v): return m
        else return -1;
    m=(i_max+i_min)//2
    return (Ricerca_seq_ric2 (A, v, i_min, m-1)
            OR
            Ricerca_seq_ric2 (A, v, m+1, i_max))
costo:   T(n) = 2T(n/2) + Θ(1)
         T(1) = Θ(1)
```

Metodo iterativo (5)

Esempio: Equazione relativa alla ricerca sequenziale ricorsiva (seconda versione):

$$T(n) = 2T(n/2) + \Theta(1)$$

$$T(1) = \Theta(1)$$

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(1) \text{ ma } T(n/2) = 2T(n/2^2) + \Theta(1) \\ \text{Sostituendo: } &= 2[2T(n/2^2) + \Theta(1)] + 2\Theta(1) = 2^2T(n/2^2) + 2\Theta(1) + \Theta(1) = \\ T(n) &= 2[2T(n/2^2) + \Theta(1)] + \Theta(1) = 2^2T(n/2^2) + 2\Theta(1) + \Theta(1) = \\ &= 2^k T(n/2^k) + \sum_{i=0..k-1} 2^i \Theta(1) = \dots \quad \frac{m}{2^k} = 1 \Rightarrow m = 2^k \Rightarrow k = \log_2 n \\ \dots \text{prosegua finché } &k = \log_2 n \dots \\ &= 2^{\log_2 n} \Theta(1) + \sum_{i=0.. \log_2 n - 1} 2^i \Theta(1) = 2^{\log_2 n} \Theta(1) + \Theta(\log_2 n) \\ &= \Theta(n) \end{aligned}$$

$$(2^{\log_2 n} - 1)/(2-1)$$

A volte le cose sono un po' più complicate...

Esempio: Equazione relativa al calcolo dell' n -esimo numero di Fibonacci: $T(n) = T(n-1) + T(n-2) + \Theta(1)$
 $T(0)=T(1) = \Theta(1)$

Provo ad usare il metodo iterativo:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + \Theta(1) = T(n-2) + 2T(n-3) + T(n-4) + 3\Theta(1) = \\ &= T(n-3) + 3T(n-4) + 3T(n-5) + T(n-6) + 6\Theta(1) = \dots \end{aligned}$$

Se ne devono sostituire sempre il doppio.

Quando non riusciamo ad ottenere un'espressione che sappiamo gestire possiamo usare le *maggiorazioni e minorazioni*:

Segue Esempio: Equazione numeri di Fibonacci:

- $T(n) = T(n-1) + T(n-2) + \Theta(1)$
- $T(0)=T(1) = \Theta(1)$

A partire da $T(n)=T(n-1)+T(n-2) + \Theta(1)$ e da $T(n-1) \geq T(n-2)$:

- $T(n) \leq 2T(n-1) + \Theta(1)$: questa diseguaglianza ci permetterà di derivare un limite superiore O ;
- $T(n) \geq 2T(n-2) + \Theta(1)$: questa diseguaglianza ci permetterà di derivare un limite inferiore Ω .

Da un certo punto in poi.

$$\begin{aligned} T(n) &\leq 2T(n-1) + \Theta(1) \leq 2^2T(n-2) + 2^1\Theta(1) + \Theta(1) \leq \\ &\leq 2^3T(n-3) + 2^2\Theta(1) + 2^1\Theta(1) + \Theta(1) \leq \dots \end{aligned}$$

$$\leq 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i \Theta(1)$$

Il procedimento si ferma quando $n-k=1$, ossia $k=n-1$ per cui otteniamo:

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

e dunque troviamo che $T(n) = O(2^n)$.

$$\begin{aligned} T(n) &\geq 2T(n-2) + \Theta(1) \geq 2^2T(n-4) + 2^1\Theta(1) + \Theta(1) \geq \dots \\ &\geq 2^3T(n-6) + 2^2\Theta(1) + 2^1\Theta(1) + \Theta(1) \geq \dots \\ &\geq 2^k T(n-2k) + \sum_{i=0}^{k-1} 2^i \Theta(1) \quad m-2k = \frac{m}{2} \Rightarrow 2k = m - \frac{m}{2} \Rightarrow k = \frac{m}{2} \\ \text{Il procedimento si ferma quando } &n-2k=0 \text{ (se } n \text{ è pari; per } n \text{ dispari il procedimento termina quando } n-2k=1 \text{ e differisce per alcuni dettagli, ma il comportamento asintotico non cambia), ossia } k = n/2 \text{ per cui otteniamo:} \\ T(n) &\geq 2^{n/2} \Theta(1) + \sum_{i=0}^{\frac{n}{2}-1} 2^i \Theta(1) = 2^{n/2} \Theta(1) + (2^{n/2}-1)\Theta(1) = \Theta(2^{n/2}) \end{aligned}$$

Anche se non siamo in grado di trovare una funzione asintotica precisa (Θ), possiamo comunque concludere che il calcolo dei numeri di Fibonacci con una tecnica ricorsiva richiede un tempo esponenziale in n , visto che

$$k_1 2^{n/2} \leq T(n) \leq k_2 2^n$$

per opportune costanti k_1 e k_2 .

Metodo di sostituzione.

- **Idea:** provare a fare un'ipotesi sull'equazione e sostituirla.
- si ipotizza una soluzione per l'equazione di ricorrenza data;
- si verifica (dimostrando per induzione) se essa "funziona".

- Difficoltà:

- si deve trovare la funzione più vicina alla vera soluzione, perché tutte le funzioni più grandi (se stiamo cercando O) o più piccole (se stiamo cercando Ω) funzionano.
- In effetti questo metodo serve soprattutto nelle dimostrazioni mentre si consiglia di utilizzarlo nella pratica.

Esempio: Equazione relativa alla ricerca seq. ricorsiva

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

- Ipotizziamo la soluzione $T(n) = cn$ per una costante opportuna c .
- $T(n) = cn = c(n-1) + \Theta(1)$, cioè $c = \Theta(1)$;
- $T(1) = c = \Theta(1)$,
- Non è detto che le due costanti nascoste nella notazione asintotica siano le stesse. Per questa ragione, è necessario eliminare la notazione asintotica dall'equazione di ricorrenza, così che le costanti non rimangano "nascoste" nella notazione, conducendo a risultati errati

$$\text{et} = cn - c + \Theta(1) \\ c = \Theta(1)$$

segue Esempio: Equazione relativa alla ricerca seq. ricorsiva

- $T(n) = T(n - 1) + c$
- $T(1) = d$ per due costanti c e d fissate.

- Ipotizziamo la soluzione $T(n) = O(n)$, ossia $T(n) \leq kn$ dove k è una costante che va ancora determinata.
- **N.B.** non si può sperare in una soluzione esatta, ma possiamo solo maggiorare o minorare.

- Sostituiamo nel caso base: $k > T(1) = d$

$T(1) \leq k$; poiché sapevamo che $T(1) = d$, la diseguaglianza è soddisfatta se e solo se $k \geq d$.

- Sostituiamo nella formulazione ricorsiva:

$T(n) \leq k(n-1) + c = kn - k + c \leq kn$; vera se e solo se $k \geq c$.

- La soluzione $T(n) \leq kn$ è corretta per tutti i valori di k tali che $k \geq c$ e $k \geq d$. Poiché un tale k esiste sempre, una volta fissati c e d , $T(n)$ è in $O(n)$.

HP INDOTT
 $T(m-i) \leq k(m-i)$

- Che cosa sarebbe successo se avessimo ipotizzato $T(n) \leq kn^2$?
- Anche questa soluzione è corretta per tutti i valori di k tali che $k \geq c$ e $k \geq d$.
- A noi interessa stimare $T(n)$ asintoticamente tramite la funzione più piccola, e questo è spesso un obiettivo difficile.

- Per rendere il nostro risultato stretto, ipotizziamo ora la soluzione $T(n) = \Omega(n)$, ossia $T(n) \geq hn$ dove h è una costante che va ancora determinata.
- In modo assolutamente analogo al caso O , sostituiamo nel caso base ottenendo $h \leq d$,
- Sostituiamo nella formulazione ricorsiva dell'equazione di ricorrenza ottenendo $T(n) \geq h(n-1) + c = hn - h + c \geq hn$ vera se e solo se $h \leq c$.
- Deduciamo dunque che $T(n)$ è un $\Omega(n)$ poiché è possibile trovare un valore h ($h \leq c$, $h \leq d$) per il quale si ha $T(n) \geq hn$.

Dalle due soluzioni: $T(n) = O(n)$ e $T(n) = \Omega(n)$, si ottiene ovviamente che $T(n) = \Theta(n)$.

C.B. $m=1$ $d=T(1) \geq h$ vera
 $\forall h \leq d$

HP IND $T(m-i) \geq k(m-i) \Rightarrow$
 $T(n) \geq h(m-1) + c = hm - h + c \geq hn$
 $-h + c \geq 0 \Leftrightarrow h \leq c$

Esempio.

- $T(n) = 2T(n/2) + \Theta(1)$
- $T(1) = \Theta(1)$

Dobbiamo eliminare per prima cosa la notazione asintotica:

- $T(n) = 2T(n/2) + c$ per due costanti positive **fissate** c e d
- $T(1) = d$

Ipotizziamo la soluzione $T(n) \leq kn$ per qualche k da **determinare** e sostituiamo:

- nel caso base: $d \leq k$
- nel caso generale: $T(n) \leq 2k(n/2) + c = kn + c$ che non è MAI $\leq kn$...

Vuol dire che l'ipotesi è errata? non sempre:

HP IND. $T(n/2) \leq \frac{kn}{2}$

$\rightarrow kn + c \leq \frac{kn}{2}$ mai vera!!

HP. $T(n/2) \leq \frac{kn-h}{2}$

Ipotizziamo la nuova soluzione $T(n) \leq kn-h$ per qualche k ed h da **determinare** e sostituiamo:

- nel caso base: $d \leq k-h$ vera $\forall h, k.t.c. [k-h > d]$
- nel caso generale: $T(n) \leq 2(kn/2 - h) + c = kn - 2h + c \leq kn-h$ sse $h \geq c$.

Da qui deduciamo che $T(n) = O(n)$.

Poi dobbiamo dimostrare anche che $T(n) = \Omega(n)$

PER ESERCIZIO

Metodo dell'albero.

Idea:

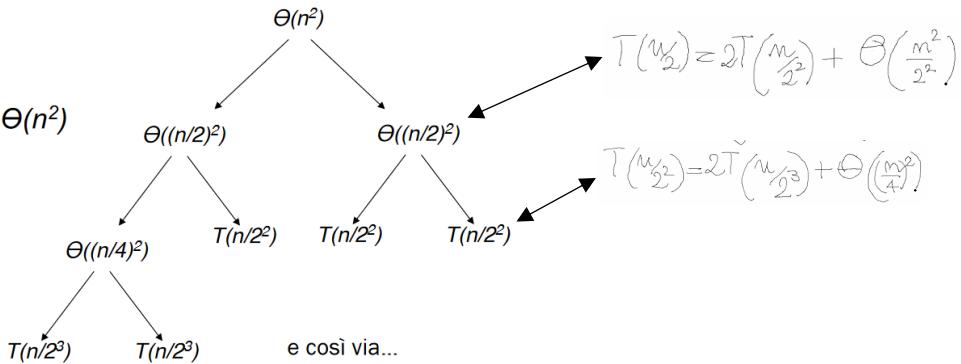
- rappresentare graficamente lo sviluppo del costo computazionale dell'algoritmo, in modo da poterla valutare con maggior facilità rispetto al metodo iterativo.

Dificoltà:

- come il metodo iterativo: ad un certo punto dobbiamo generalizzare al caso k e fare calcoli algebrici complessi.

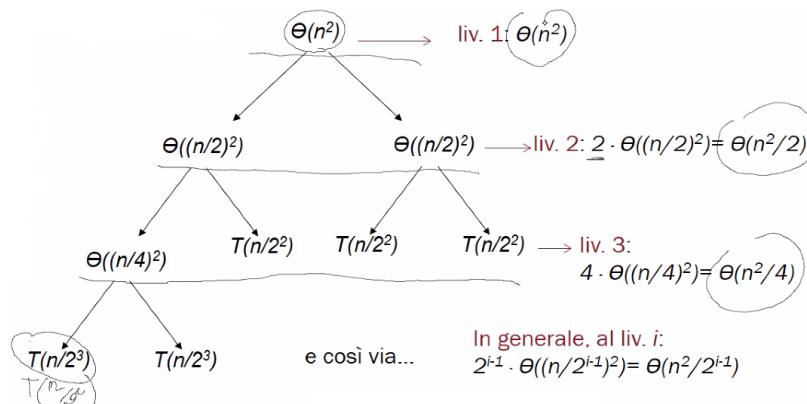
Esempio:

- $T(n) = 2T(n/2) + \Theta(n^2)$
- $T(1) = \Theta(1)$



Ognuno dei nodi rappresenta una chiamata ricorsiva.

Una volta completato l'albero, il costo computazionale è dato dalla somma dei contributi di tutti i livelli (cioè le "righe" in cui sono disposti i nodi) di cui è costituito l'albero.



In questo caso il numero di livelli dell'albero ha un valore tale che $n/2^{i-1} = 1$, ossia $i-1 = \log n$ da cui $i = \log n + 1$.

$$\sum_{i=0}^k \frac{1}{2^i} = O(1) \quad \text{con } k = \log n \Rightarrow m = 2^k \Rightarrow i = \log n$$

Sommiamo i contributi di tutti i livelli:

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

Metodo principale.

Idea:

- avere una "ricetta" meccanica per risolvere un'equazione di ricorrenza

Dificoltà:

- funziona solo quando l'equazione è della forma: $T(n) = aT(n/b) + f(n)$ con $T(1) = \Theta(1)$, perciò non funziona sempre.

Enunciato del teorema principale:

Dati $a \geq 1$, $b > 1$, una funzione asintoticamente positiva $f(n)$ ed un'equazione di ricorrenza di forma $T(n) = aT(n/b) + f(n)$, $T(1) = \Theta(1)$ vale che:

- Se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche costante $\varepsilon > 0$ allora $T(n) = \Theta(n^{\log_b a})$
- Se $f(n) = \Theta(n^{\log_b a})$ allora $T(n) = \Theta(n^{\log_b a} \log n)$
- Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche costante $\varepsilon > 0$ e se $a f(n/b) \leq c f(n)$ per qualche costante $c < 1$ e per n sufficientemente grande, allora $T(n) = \Theta(f(n))$.

Il teorema principale ci dice che in ciascuno dei tre casi vengono confrontati fra loro $f(n)$ e $n^{\log_b a}$ ed il costo computazionale è governato dal maggiore dei due:

- se **(caso 1)** il più grande dei due è $n^{\log_b a}$, allora il costo è $\Theta(n^{\log_b a})$;
- se **(caso 3)** il più grande dei due è $f(n)$, allora il costo è $\Theta(f(n))$;
- se **(caso 2)** sono uguali, allora si moltiplica $f(n)$ per un fattore logaritmico.

Si noti che “più grande” e “più piccolo” in questo contesto significa **polynomialmente** più grande (o più piccolo), data la presenza all'esponente di ε . In altre parole, $f(n)$ deve essere asintoticamente più grande (o più piccola) rispetto a $n^{\log_b a}$ di un fattore n^ε per qualche $\varepsilon > 0$.

In effetti fra i casi 1 e 2 vi è un intervallo in cui $f(n)$ è più piccola di $n^{\log_b a}$, ma non polynomialmente.

Analogamente, fra i casi 2 e 3 vi è un intervallo in cui $f(n)$ è più grande di $n^{\log_b a}$, ma non polynomialmente.

↓
In questo caso non si può utilizzare il metodo principale.

Esempi.

$T(n) = 9T(n/3) + \Theta(n)$ e $T(1) = \Theta(1)$ <ul style="list-style-type: none"> • $a = 9$, $b = 3$ • $f(n) = \Theta(n)$ • $n^{\log_b a} = n^{\log_3 9} = n^2$ <p>Poiché $f(n) = \Theta(n^{\log_3 9 - \varepsilon})$ con $\varepsilon = 1$, siamo nel caso 1, per cui $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.</p>	$T(n) = T(\frac{2}{3}n) + \Theta(1)$ e $T(1) = \Theta(1)$ <ul style="list-style-type: none"> • $a = 1$, $b = 3/2$ • $f(n) = \Theta(1)$ • $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$ <p>Poiché $f(n) = \Theta(n^{\log_b a})$ siamo nel caso 2, per cui $T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$.</p>
$T(n) = 3T(n/4) + \Theta(n \log n)$ e $T(1) = \Theta(1)$ <ul style="list-style-type: none"> • $a = 3$, $b = 4$ • $f(n) = \Theta(n \log n)$ • $n^{\log_b a} = n^{\log_4 3} \approx n^{0.7}$ <p>Poiché $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ con, ad esempio, $\varepsilon = 0.2$, siamo nel caso 3 se possiamo dimostrare che $3^{\frac{n}{4}} \log \frac{n}{4} \leq c n \log n$, per qualche $c < 1$ ed n abbastanza grande. Ponendo $c = \frac{3}{4}$ otteniamo:</p> $3^{\frac{n}{4}} \log \frac{n}{4} \leq \frac{3}{4} n \log n$ <p>che è vera, quindi $T(n) = \Theta(n \log n)$.</p>	$T(n) = 2T(n/2) + \Theta(n \log n)$ e $T(1) = \Theta(1)$ <ul style="list-style-type: none"> • $a = 2$, $b = 2$ • $f(n) = \Theta(n \log n)$ • $n^{\log_b a} = n^{\log_2 2} = n$ <p>Ora, $f(n) = \Theta(n \log n)$ è asintoticamente più grande di $n^{\log_b a} = n$, ma non polynomialmente più grande. Infatti, $\log n$ è asintoticamente minore di n^ε per qualunque valore di $\varepsilon > 0$. Di conseguenza non possiamo applicare il metodo del teorema principale.</p>

Il problema dell'ordinamento: Ordinamento naif.

Il problema dell'ordinamento degli elementi di un insieme è un problema molto ricorrente in informatica poiché ha un'importanza fondamentale per le applicazioni: lo si ritrova molto frequentemente come sotto problema nell'ambito dei problemi reali.

Si stima che una parte rilevante del tempo di calcolo complessivo consumato nel mondo sia relativa all'esecuzione di algoritmi di ordinamento.

Algoritmi di ordinamento.

Un **algoritmo di ordinamento** è un algoritmo capace di ordinare gli elementi di un insieme sulla base di una certa relazione d'ordine, definita sull'insieme stesso.

Per semplicità di trattazione, supponiamo che gli n elementi da ordinare siano numeri interi e siano contenuti in un array i cui indici vanno da 0 ad $n-1$.

Tuttavia, nei problemi reali, i dati da ordinare sono ben più complessi: in generale essi sono strutturati in record, cioè in gruppi di informazioni non sempre omogenee relative allo stesso soggetto, e si vuole ordinarli rispetto ad una di tali informazioni (ad esempio, il Codice Fiscale).

Esistono diversi algoritmi di ordinamento.

Dapprima illustreremo gli algoritmi più semplici ma ci renderemo conto che, volendo migliorare l'efficienza, sarà necessario introdurre idee più elaborate.

Gli algoritmi *semplici* (costo $\Theta(n^2)$) che illustreremo sono:

Insertion sort	Selection sort	Bubblesort
----------------	----------------	------------

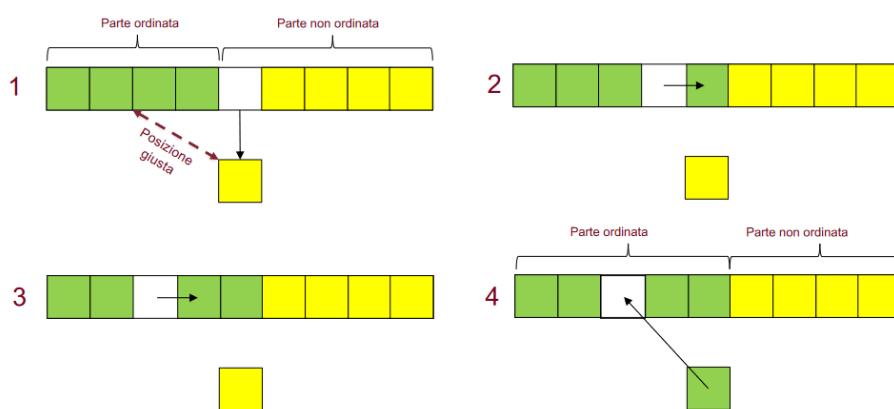
Gli algoritmi *più evoluti* (costo computazionale migliore) che vedremo sono:

Mergesort	Quicksort	Heapsort
-----------	-----------	----------

Insertion sort.

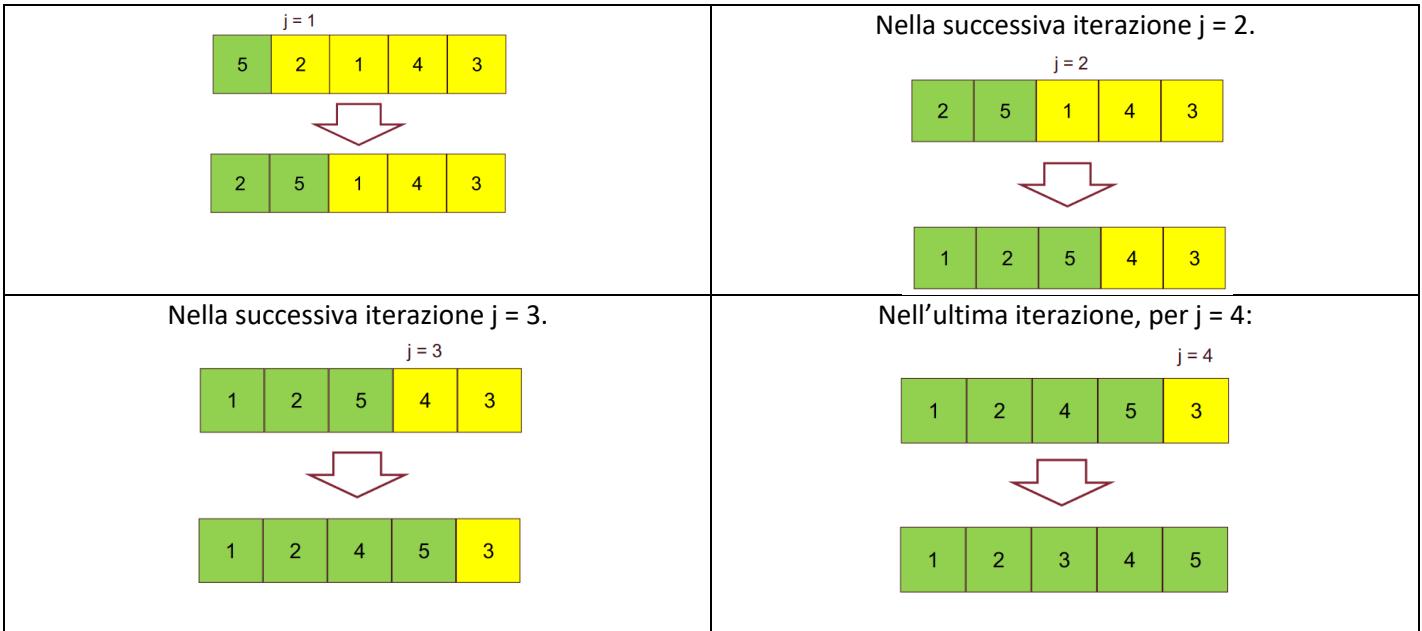
- Gli elementi da ordinare sono inizialmente contenuti in un array. Per ogni $i=1, \dots, n-1$:
- Si estrae l'elemento della posizione i , così da liberare la sua posizione corrente.
- Si spostano di una posizione verso destra tutti gli elementi alla sua sinistra (già ordinati) che sono maggiori di esso.
- Si inserisce l'elemento nella posizione che si è liberata.
- INVARIANTE: ad ogni passo i , gli elementi con indice $< i$ (a sinistra) sono già ordinati, mentre quelli con indice $> i$ (a destra) sono ancora da processare.

L'algoritmo procede per iterazioni, aumentando ad ogni iterazione la dimensione della porzione di sinistra dell'array, che è ordinata.



Prima di esaminare il codice dell'algoritmo vediamo un esempio.

Un array di un solo elemento è già ordinato, quindi si estrae per primo il secondo elemento. L'indice usato nel codice per l'elemento da sistemare è j .



Lo pseudocodice dell'algoritmo è il seguente:

```
def Insertion_Sort(A)
for j in range(1, len(A)) :           ( $n-1$ )  $\Theta(1) + \Theta(1)$ 
    x = A[j]                          $\Theta(1)$ 
    i = j - 1                          $\Theta(1)$ 
    while ((i >= 0) and (A[i]) > x)    $t_j \Theta(1) + \Theta(1)$ 
        A[i+1] = A[i]                  $\Theta(1)$ 
        i = i - 1                      $\Theta(1)$ 
        A[i+1] = x                    $\Theta(1)$ 
```

Costo computazionale:

$$T(n) = \sum_{j=2}^n (\Theta(1) + t_j \Theta(1) + \Theta(1)) + \Theta(1)$$

Il numero di iterazioni del while interno, t_j , può andare:

- da un minimo di 1 per ogni j (se ogni $x > A[j-1]$)
- a un massimo di j per ogni j (se ogni $x < A[1]$)

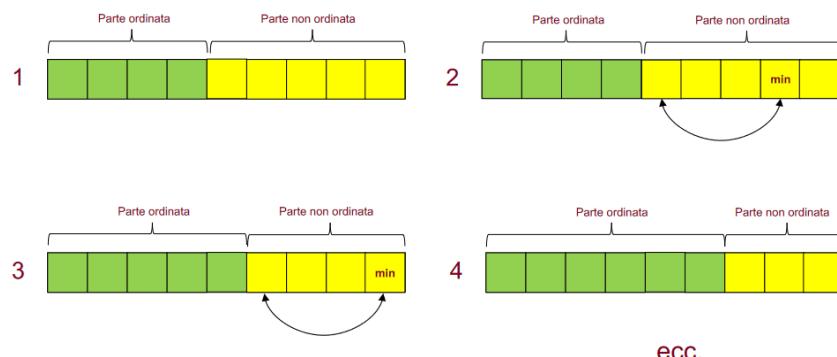
Quindi caso migliore e caso peggiore differiscono:

$$\text{Caso peggiore: } T(n) = \sum_{j=2}^n (\Theta(1) + \Theta(j)) =$$

$$\text{Caso migliore: } T(n) = (n-1)\Theta(1) = \Theta(n) = \Theta\left(\frac{n(n+1)}{2} - 1\right) = \Theta(n^2)$$

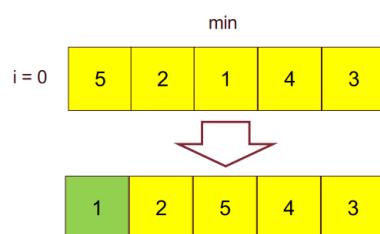
Selection sort.

- Cerca il minimo dell'intero array e lo mette in prima posizione (con uno scambio);
- Cerca il nuovo minimo nell'array restante, cioè nelle posizioni dalla seconda all'ultima incluse, e lo mette in seconda posizione (con uno scambio);
- Cerca il minimo nelle posizioni dalla terza all'ultima incluse e lo mette in terza posizione;
- e così via...



Prima di esaminare il codice dell'algoritmo vediamo un esempio.

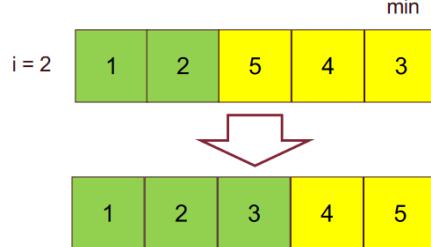
L'indice usato nel codice per l'elemento da sistemare è i .



Per $i = 1$ il minimo, in questo esempio, viene rimesso nella sua posizione:



Successiva iterazione, per $i = 2$:



E così via. In questo esempio le ultime due iterazioni rimettono ciascuno dei due elementi al proprio posto.

Lo pseudocodice dell'algoritmo è il seguente.

```
def Selection_Sort(A)
for i in range(len(A)-1):
    m = i
    for j in range(i+1, len(A)):
        if (A[j] < A[m]):
            m = j
    A[m], A[i] = A[i], A[m]
```

$(n - 1)\Theta(1) + \Theta(1)$
 $\Theta(1)$
 $(n - i)\Theta(1) + \Theta(1)$
 $\Theta(1)$
 $\Theta(1)$
 $\Theta(1)$

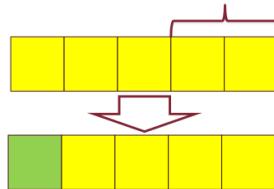
Costo computazionale:

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

In questo algoritmo non c'è differenza fra i costi di caso migliore e peggiore.

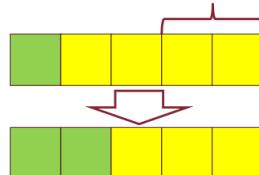
Bubble sort.

Funziona per fasi. Ogni fase ispeziona una dopo l'altra, da destra a sinistra, ogni coppia di elementi adiacenti e, se l'ordine dei due elementi non è quello giusto, essi vengono scambiati.



La prima «passata» scende fino alla prima posizione e sistema il minimo nella posizione corretta.

La seconda passata si ferma alla seconda posizione e sistema il secondo minimo.



E così via per le passate (iterazioni) successive.

Lo pseudocodice dell'algoritmo è il seguente.

```
def Bubble_Sort(A[1..n])
    for i in range(len(A)-1):           n Θ(1) + Θ(1)
        for j in range(n, i, -1):       (n - i) Θ(1) + Θ(1)
            if (A[j] < A[j - 1])      Θ(1)
                A[j], A[j - 1] = A[j-1], A[j]   Θ(1)
```

Costo computazionale:

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

Anche in questo algoritmo non c'è differenza fra i costi di caso migliore e peggiore.

In rete sono disponibili diverse pagine che offrono l'animazione di vari algoritmi di ordinamento. Queste sono particolari.

Insertion sort: <https://www.youtube.com/watch?v=EdIKIf9mHk0>

Selection sort: <https://www.youtube.com/watch?v=0-W8OEwLebQ>

Bubble sort: <https://www.youtube.com/watch?v=semGJAJ7i74>

La complessità dell'ordinamento.

I tre algoritmi di ordinamento appena visti hanno tutti un costo computazionale asintotico che cresce come il quadrato del numero di elementi da ordinare.

Domanda 1: si può fare di meglio? Risposta: sì, se troviamo un algoritmo X con costo minore.

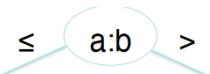
Domanda 2: Se sì, quanto meglio si può fare? Risposta: Non sapremo, chi ci assicura che non si possa fare ancora meglio del nuovo algoritmo X?

Come si fa a stabilire un limite di costo computazionale al di sotto del quale nessun algoritmo di ordinamento basato su confronti fra coppie di elementi possa andare?

- Esiste uno strumento adatto allo scopo, l'**albero di decisione**, che permette di rappresentare tutte le strade che la computazione di uno specifico algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.
- Nel caso degli algoritmi di ordinamento basati su confronti, ogni test effettuato ha due soli possibili esiti (ad es.: minore o uguale, oppure no).

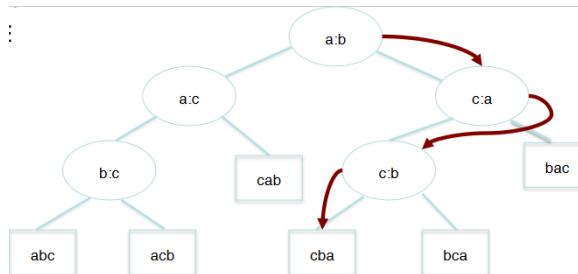
L'albero di decisione relativo a un qualunque algoritmo di ordinamento basato su confronti ha queste proprietà:

- è un **albero binario** che rappresenta tutti i possibili confronti che vengono effettuati dall'algoritmo; ogni nodo interno (ossia un nodo che non è una foglia) ha esattamente due figli;
- ogni nodo interno rappresenta un singolo confronto, ed i due figli del nodo sono relativi ai due possibili esiti di tale confronto;



- ogni foglia rappresenta una possibile soluzione del problema, la quale è una specifica **permutazione** della sequenza in ingresso.

Esempio: albero di decisione dell'Insertion sort su 3 elementi a, b, c.



Eseguire l'algoritmo corrisponde a scendere dall'radice dell'albero alla foglia che contiene la permutazione che costituisce la soluzione; la discesa è governata dagli esiti dei confronti che vengono via via effettuati.

La lunghezza (= numero degli archi) di tale cammino rappresenta il numero di confronti necessari per trovare la soluzione.

La lunghezza del percorso più lungo dalla radice ad una foglia (altezza dell'albero binario) **rappresenta il numero di confronti che l'algoritmo deve effettuare nel caso peggiore**.

Determinare una limitazione inferiore all'altezza dell'albero di decisione relativo a qualunque algoritmo di ordinamento basato su confronti equivale a trovare una limitazione inferiore al tempo di esecuzione nel caso peggiore di qualunque algoritmo di ordinamento basato su confronti.

Cerchiamo di determinare tale limitazione:

- **Osservazione 1:** dato che la sequenza di ingresso può avere una qualunque delle sue permutazioni come soluzione, l'albero di decisione deve contenere nelle foglie tutte le permutazioni della sequenza in ingresso, che sono $n!$ per un problema di dimensione n .
- **Osservazione 2:** un albero binario di altezza h non può contenere più di 2^h foglie.

Ogni volta che passo dal livello sopra al livello sotto raddoppio i nodi.

Segue che l'altezza h dell'albero di decisione di qualunque algoritmo di ordinamento basato su confronti deve essere tale per cui:

$$2^h \geq n! \text{ ossia } h \geq \log n!$$

Valutiamo $\log n!$ (per semplicità assumiamo che n sia pari).

$$\begin{aligned} \log n! &= \log(n*(n-1)*(n-2)*(n-3)\dots*2*1)= \\ &= \sum_{i=1}^n \log i = \\ &= \sum_{i=1}^{n/2} \log i + \sum_{i=n/2+1}^n \log i \geq \\ &\geq 0 + \sum_{i=\frac{n}{2}+1}^n \log \frac{n}{2} = \frac{n}{2} \log n - \frac{n}{2} = \Theta(n \log n) \end{aligned}$$

Siccome abbiamo visto che $h \geq \log n!$
e che $\log n! = \Theta(n \log n)$
ciò significa che

$$h = \Omega(n \log n)$$

Quanto detto si riassume nel seguente: **Teorema**.

Il costo computazionale di qualunque algoritmo di ordinamento basato su confronti è $\Omega(n \log n)$.

Esercizio svolto.

Nell'algoritmo di Insertion sort è possibile ricercare la posizione in cui inserire l'elemento i-esimo tramite la ricerca binaria. Come cambia il calcolo del costo computazionale dell'algoritmo?

Soluzione.

Per ogni j da 1 ad n-1, l'algoritmo di Insertion Sort in versione standard:

- scorre la parte ordinata di array confrontando ciascun elemento con x
- trova la posizione corretta di x
- sposta tutti gli elementi che sono a destra per fare posto ad x.

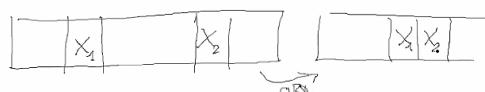
Consideriamo lo pseudocodice modificato dell'Insertion Sort:

```
def Insertion_Sort(A):
    for j in range(1, len(A)):
        x = A[j]
        k = RicercaBin(A, x)
        /* x dovrebbe stare in pos.k
        i = j - 1
        while ((i>=0) and (i>=k))
            A[i+1] = A[i]
            i = i - 1
        A[k] = x
```

$(n-1)\Theta(1) + \Theta(1)$
 $\Theta(1)$
 $O(\log j)$
 $\Theta(1)$
 $t_j\Theta(1) + \Theta(1)$
 $\Theta(1)$
 $\Theta(1)$
 $\Theta(1)$

- Il contributo della ricerca binaria non si sostituisce a t_j , ma si aggiunge ad esso.
- Anche se troviamo la posizione di x più velocemente, dobbiamo comunque spostare gli elementi a destra...
- Pertanto, il costo computazionale *non migliora!*

Algoritmo di ordinamento stabile se, in caso ci siano più valori uguali nel vettore, questi manterranno lo stesso ordine dopo l'ordinamento.



Riusciamo a progettare degli algoritmi che richiedono costo computazionale uguale proprio a $\Theta(n \log n)$ e sono, quindi, ottimi?

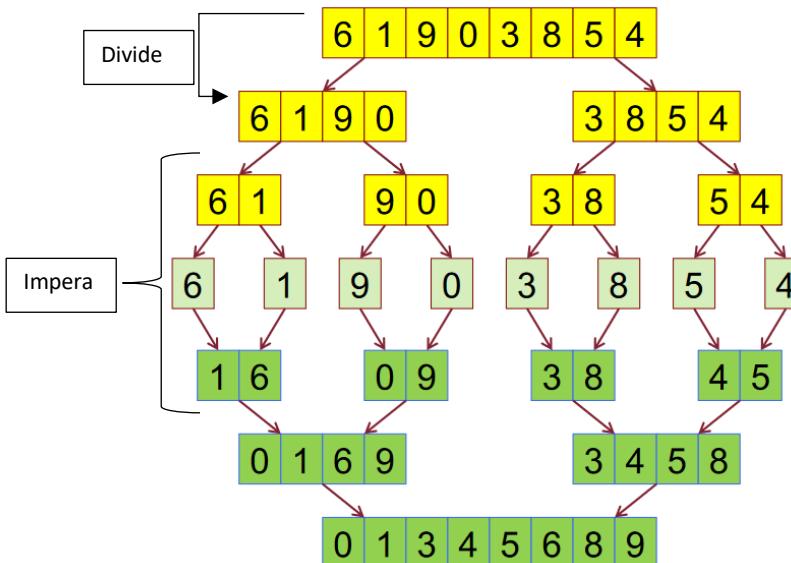
Merge sort.

L'algoritmo **merge sort** (ordinamento per fusione) è un algoritmo ricorsivo che adotta una tecnica algoritmica detta **divide et impera**. Essa può essere descritta come segue:

- il problema complessivo si suddivide in sotto problemi di dimensione inferiore (**divide**);
- i sotto problemi si risolvono ricorsivamente (**impera**);
- le soluzioni dei sotto problemi si compongono per ottenere la soluzione al problema complessivo (**combina**).

L'approccio dell'algoritmo (ordinare n oggetti) Merge Sort è il seguente:

- **divide**: la sequenza di n elementi viene divisa in due sottosequenze di $n/2$ elementi ciascuna;
- **impera**: le due sottosequenze di $n/2$ elementi vengono ordinate ricorsivamente;
- **passo base**: la ricorsione termina quando la sottosequenza è costituita di un solo elemento, per cui è già ordinata;
- **combina**: le due sottosequenze – ormai ordinate – di $n/2$ elementi ciascuna vengono “fuse” in un'unica sequenza ordinata di n elementi.



- **divide**: la sequenza di n elementi viene divisa in due sotto-sequenze di $n/2$ elementi ciascuna;
- **impera**: le due sotto-sequenze di $n/2$ elementi vengono ordinate ricorsivamente;
- **passo base**: la ricorsione termina quando la sotto-sequenza è costituita di un solo elemento, per cui è già ordinata;
- **combina**: le due sotto-sequenze – ormai ordinate – di $n/2$ elementi ciascuna vengono “fuse” in un'unica sequenza ordinata di n elementi.

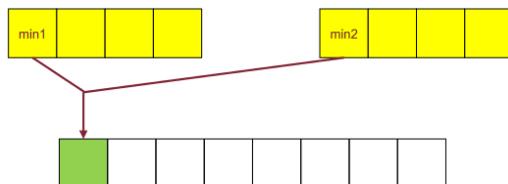
```
def Merge_sort (A, ind_primo, ind_ultimo):
    if (ind_primo < ind_ultimo)
        ind_medio = (ind_primo+ind_ultimo)//2
        Merge_sort (A, ind_primo, ind_medio)
        Merge_sort (A, ind_medio + 1, ind_ultimo)
        Fondi (A, ind_primo, ind_medio, ind_ultimo) T(n)
    else
        S(n) Θ(1)
```

$$T(n) = \Theta(1) + 2T(n/2) + S(n) \quad \text{dove } S(n) = \text{costo di Fondi}()$$

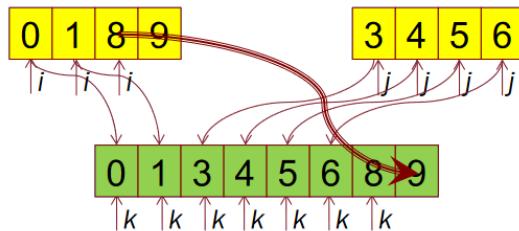
$$T(1) = \Theta(1)$$

Funzionamento della funzione Fondi():

- sfrutta il fatto che le due sottosequenze sono ordinate;
- il minimo della sequenza complessiva non può che essere **il più piccolo fra i minimi delle due sottosequenze** (se essi sono uguali, scegliere l'uno o l'altro non fa differenza);
- dopo aver eliminato da una delle due sottosequenze tale minimo, la proprietà rimane: il prossimo minimo non può che essere il più piccolo fra i minimi delle due parti rimanenti delle due sottosequenze.



Esempio di funzionamento della funzione Fondi():



Dopo aver ricopiato anche l'ultimo elemento il vettore complessivo risulta ordinato.

```

def Fondi (A,ind_primo,ind_medio,ind_ultimo):
    i,j = indice_primo, indice_medio+1
    B=[]
    while ((i≤ind_medio) and (j≤ind_ultimo)):
        if (A[i]≤A[j]):
            B.append(A[i])
            i += 1
        else:
            B.append(A[j])
            j += 1
    while (i≤ind_medio) //il primo sottovettore non è terminato
        B.append(A[i])
        i += 1
    while (j≤ind_ultimo) //il secondo sottovettore non è terminato
        B.append(A[j])
        j += 1
    for i in range(len(B)):
        A[primo+i] = B[i]

```

Valutiamo il costo computazionale della funzione Fondi():

- inizializzazione delle variabili: $\Theta(1)$;
- primo ciclo while
 - ogni iterazione ha costo $\Theta(1)$ e incrementa di 1 l'indice i oppure l'indice j. Quindi il costo del while varia da un minimo di $n/2$ a un massimo di n , ossia è $\Theta(n)$.
- secondo e terzo while (mai eseguiti entrambi):
 - si ricopia nel vettore B l'eventuale “coda” di una delle due sottosequenze: $O(n)$;
- copia del vettore B nell'opportuna porzione del vettore A: $\Theta(n)$.

Dunque, il costo $S(n)$ della funzione Fondi() è:

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

Quindi il costo computazionale del Merge Sort è:

<pre> def Merge_sort (A,ind_primo,ind_ultimo): if (ind_primo < ind_ultimo) ind_medio = (ind_primo+ind_ultimo)//2 Merge_sort (A, ind_primo, ind_medio) Merge_sort (A, ind_medio + 1, ind_ultimo) Fondi (A, ind_primo, ind_medio, ind_ultimo) </pre>	$T(n)$ $\Theta(1)$ $\Theta(1)$ $T(n/2)$ $T(n/2)$ $\Theta(n)$
---	---

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ T(1) &= \Theta(1) \end{aligned}$$

OSSERVAZIONE.

L'operazione di fusione non si può fare “in loco”, cioè aggiornando direttamente il vettore A, senza incorrere in un aggravio del costo.

Infatti, in A bisognerebbe fare spazio via via al minimo successivo, ma questo costringerebbe a spostare di una posizione tutta la sottosequenza rimanente per ogni nuovo minimo, il che costerebbe $\Theta(n)$ operazioni elementari per ciascun elemento da inserire, facendo lievitare quindi il costo computazionale della fusione da $\Theta(n)$ a $\Theta(n^2)$.

Ciò a sua volta risulterebbe nell'equazione di ricorrenza:

$$T(n) = 2 T(n/2) + \Theta(n^2)$$

la cui soluzione, come sappiamo, è:

$$T(n) = \Theta(n^2)$$

Algoritmo estremamente elegante, ma non usato perché nel caso peggiore $\Theta(n^2)$.

Esercizio svolto.

Nonostante Merge Sort funzioni in tempo $\Theta(n \log n)$ mentre Insertion Sort in $O(n^2)$, i fattori costanti sono tali che l'Insertion Sort è più veloce del Merge Sort per valori piccoli di n . Quindi, ha senso usare l'Insertion Sort dentro il Merge Sort quando i sottoproblemi diventano sufficientemente piccoli.

- Si consideri una modifica del Merge Sort in cui il caso base si applica ad una porzione del vettore di lunghezza k , che viene ordinata usando Insertion Sort.
- Le porzioni vengono combinate usando il meccanismo standard di fusione.
- Si determini il valore di k come funzione di n per cui l'algoritmo modificato ha lo stesso tempo di esecuzione asintotico del Merge Sort.

Soluzione. Il codice che realizza tale versione è il seguente.

```
def Merge_Insertion (A, k, primo, ultimo, dim):
    if dim>k:
        medio = (primo+ultimo)//2
        Merge_Insertion (A,k, primo, medio, medio-primo+1)
        Merge_Insertion (A,k,medio+1, ultimo, ultimo-primo)
        Fondi(primo, medio, ultimo)
    else: InsertionSort(primo, ultimo)
```

La chiamata iniziale sarà:

$\text{Merge_Insertion } (A, k, \cancel{x}, \cancel{x}, n)$

L'equazione di ricorrenza che lo caratterizza è la seguente:

$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(k) = \Theta(k^2)$$

Risolviamola col metodo iterativo:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) = \\ &= 2[2T(n/2^2) + \Theta(n/2^1)] + \Theta(n/2^0) = \\ &= 2[2[2T(n/2^3) + \Theta(n/2^2)] + \Theta(n/2^1)] + \Theta(n/2^0) = \\ &= 2^3T(n/2^3) + 2^2\Theta(n/2^2) + 2^1\Theta(n/2^1) + 2^0\Theta(n/2^0) \\ &\quad \dots \\ &= 2^h T(n/2^h) + \sum_{i=0}^{h-1} 2^i \Theta\left(\frac{n}{2^i}\right) \end{aligned}$$

Ci fermiamo incontrando il caso base, il che succede quando $n/2^h = k$, ossia $2h=n/k \Rightarrow h = \log n/k$

Sostituendo tale valore nell'espressione precedente otteniamo:

$$\begin{aligned} T(n) &= 2^{\log n/k} T(n/2^{\log n/k}) + \sum_{i=0}^{\log n/k - 1} 2^i \Theta\left(\frac{n}{2^i}\right) = \\ T(k) &= \underbrace{\frac{n}{k} \Theta(k^2)}_{\Theta(nk)} + \Theta\left(n \log \frac{n}{k}\right) = \\ &= \Theta(nk) + \Theta\left(n \log \frac{n}{k}\right) = \\ &= \Theta(nk) + \Theta(n \log n - n \log k) \end{aligned}$$

Se $k = O(\log n)$ otteniamo:

$$T(n) = O(n \log n) + \Theta(n \log n - n \log \log n) = \Theta(n \log n)$$

Quicksort.

L'algoritmo **Quicksort** (ordinamento veloce) ha costo $O(n^2)$ nel caso peggiore, ma nella pratica è spesso la *soluzione migliore per grandi valori di n* perché:

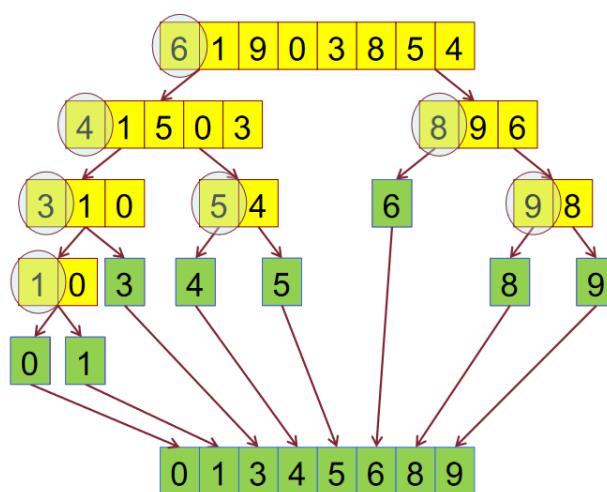
- il suo tempo di esecuzione atteso è $\Theta(n \log n)$;
- i fattori costanti nascosti sono molto piccoli;
- permette l'ordinamento “in loco”.

Riunisce i vantaggi del Selection sort (ordinamento in loco) e del Merge sort (ridotto tempo di esecuzione).

Ha però lo svantaggio dell'elevato costo computazionale nel caso peggiore.

Anche l'algoritmo **Quicksort** è un algoritmo *ricorsivo* che adotta una tecnica algoritmica detta **divide et impera**:

- **divide**: nella sequenza di n elementi si seleziona un pivot (valore di riferimento). La sequenza viene quindi divisa in due sottosequenze: quella degli elementi *minori o uguali del pivot*, e quella degli *elementi maggiori o uguali del pivot*;
- **impera**: le due sottosequenze vengono ordinate ricorsivamente;
- **passo base**: la ricorsione procede fino a quando le sottosequenze sono costituite da un solo elemento;
- **combina**: non occorre.



- **divide**: nella sequenza di n elementi seleziona un **pivot**. La sequenza viene quindi divisa in due sottosequenze: quella degli elementi minori o uguali del pivot, e quella degli elementi maggiori o uguali del pivot;
- **impera**: le due sottosequenze vengono ordinate ricorsivamente;
- **passo base**: la ricorsione procede fino a quando le sottosequenze sono costituite da un solo elemento;
- **combina**: non occorre

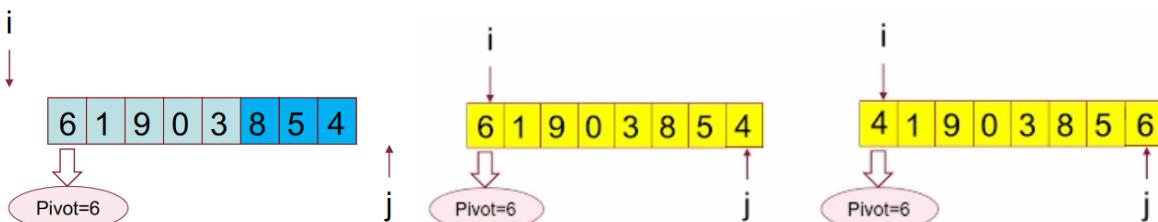
Lo pseudocodice del Quicksort è il seguente:

```
def Quick_sort (A, ind_primo, ind_ultimo)
    if (ind_primo < ind_ultimo):
        ind_medio=Partiziona(A,ind_primo,ind_ultimo)
        Quick_sort (A,ind_primo,ind_medio)
        Quick_sort (A,ind_medio+1,ind_ultimo)
```

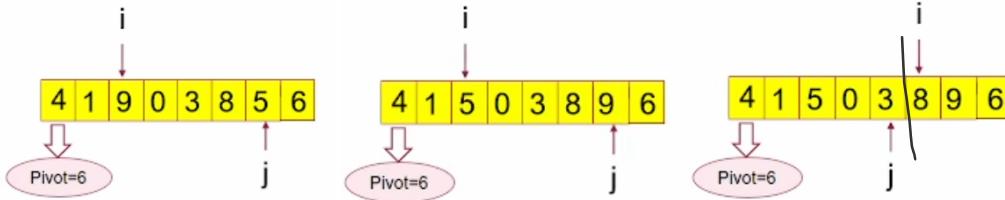
In questa implementazione *ind_medio* è l'indice dell'estremo superiore della porzione di sinistra (quella contenente elementi minori o uguali del pivot). Il suo valore è **sempre** compreso fra 1 ed ($n-1$).

Prima di vedere il codice della funzione Partiziona, vediamone il funzionamento su un esempio.

Ho due indici i e j ; in questo caso da i ci dovrebbero essere elementi più piccoli del pivot e j elementi più grandi. Se trovo su i qualcosa che è più grande del pivot lo scambio con j e così viceversa.



Ora che ho scambiato 4 e 6 faccio scorrere i miei indici.



i è andati avanti perché 1 è più piccolo di 6.

Ora abbiamo finito.

Tra 3 e 8 c'è la linea di demarcazione che mi separa gli elementi piccoli da quelli grandi.

```
Funzione Partiziona (A: vettore; ind_primo, ind_ultimo:
intero)
pivot ← A[indice_primo] //scelta arbitraria
i ← indice_primo - 1;
j ← indice_ultimo + 1
while true
repeat
    j ← j - 1
until A[j] ≤ pivot
repeat
    i ← i + 1
until A[i] ≥ pivot
if (i < j) scambia A[i] e A[j]
else return j
```

In effetti, ad esempio, nelle dispense, si procede in modo diverso, prendendo l'ultimo elemento come pivot... Anche questo codice è un po' diverso, ma la filosofia è la stessa...

```
def Partiziona(A, ind_primo, ind_ultimo):
    pivot = A[ind_ultimo]
    i = ind_primo-1

    print(A)
    print("Pivot = " + str(pivot))

    for j in range(ind_primo, ind_ultimo):
        if (A[j] <= pivot):
            i = i+1
            A[i], A[j] = A[j], A[i]

        print("j = " + str(j) + " , i = " + str(i))
    print(A)
    A[ind_ultimo], A[i+1] = A[i+1], A[ind_ultimo]
    print("i = " + str(i))
    print(A)
    print()
    return i+1 # posizione corretta del pivot
```

Analizziamo il costo computazionale della funzione *Partiziona()*.

- Le prime tre istruzioni costano $\Theta(1)$.
- Il while ha un costo $O(n)$ più un costo pari alla somma dei costi di ciò che avviene al suo interno, che è $\Theta(n)$ poiché:
 - ciascuna iterazione di ognuno dei due repeat costa $\Theta(1)$ e avvicina di una posizione un indice all'altro;
 - quindi complessivamente si effettuano $\Theta(n)$ iterazioni dei due repeat;
 - terminati i due repeat si trova un if ed un possibile scambio di elementi, $\Theta(1)$

Dunque, il costo di *Partiziona()* è $\Theta(n)$.

NOTA: il valore j restituito da *Partiziona()* vale:

- 1 se il pivot è più piccolo degli altri elementi;
- $(n-1)$ se il pivot è più grande degli altri elementi.

Questo significa che almeno un elemento tra i due ci deve essere per forza, non ci possono essere 0 elementi da una parte e tutti gli altri dall'altra parte.

Valutiamo ora il costo computazionale del Quicksort:

```
Funzione Quick_sort (A, ind_primo, ind_ultimo)
if (ind_primo < ind_ultimo):          Θ(1)
    ind_medio=Partiziona(A,ind_primo,ind_ultimo) Θ(n)
    Quick _sort (A,ind_primo,ind_medio)           T(k)
    Quick _sort (A, ind_medio+1,ind_ultimo)       T(n-k)
```

$$T(n)=T(k)+T(n-k)+\Theta(n)$$

$$T(1)=\Theta(1)$$



non sappiamo risolverla con
alcuno dei metodi studiati...

Non si riesce a risolvere con i metodi iterativi, dell'albero,...

Possiamo facilmente derivare la soluzione (eliminando k) per due situazioni, il caso migliore e quello peggiore.

- Caso **migliore**: è quello in cui, ad ogni passo, la dimensione dei due sottoproblemi è identica. (cioè $k=n/2$) Cioè i due sotto array creati hanno la stessa dimensione.

L'equazione di ricorrenza diventa:

(meglio da sperare)

$$T(n) = 2T(n/2) + \Theta(n) \text{ che ha soluzione } T(n) = \Theta(n \log n)$$

- Caso **peggiore**: è quello in cui, ad ogni passo, la dimensione di uno dei due sottoproblemi da risolvere è 1. Cioè la situazione in cui nella creazione dei due sotto array ce n'è uno con dimensione 0.

(k=1 or k=n-1) L'equazione di ricorrenza diventa:

$$T(n) = T(n-1) + \Theta(n) \text{ che ha soluzione } T(n) = \Theta(n^2)$$

Caso peggiore e caso migliore **non** coincidono.

Valutiamo ora il *costo computazionale nel caso medio*, nell'ipotesi che il valore del pivot suddivida con uguale probabilità $1/(n-1)$ la sequenza da ordinare in due sotto-sequenze di dimensioni k ed $n-k$, per tutti i valori di k tra 1 ed $n-1$.

Il valore medio si ottiene con la somma dei valori possibili diviso tutte le possibilità.

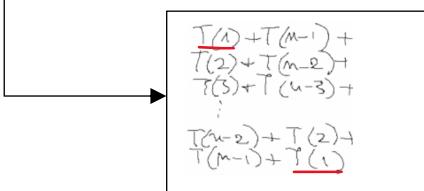
$$T(n) = \frac{1}{n-1} \left[\sum_{k=1}^{n-1} (T(k) + T(n-k)) \right] + \Theta(n)$$

$n-1$ possibili divisioni.

Ora, per ogni valore di i e $k = 1, 2, \dots, n-1$ il termine $T(k)$ compare **due volte** nella sommatoria, la prima quando $k=i$ e la seconda quando $k=n-i$.

Valutiamo dunque il valore di:

$$T(n) = \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + \Theta(n)$$



Utilizziamo il **metodo di sostituzione**, e quindi:

- eliminiamo innanzi tutto la notazione asintotica:

$$\begin{aligned} T(n) &= \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + \Theta(n) &\longrightarrow T(n) &= \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + hn \\ T(1) &= \Theta(1) &T(1) &= k \end{aligned}$$

- Per ragioni che saranno chiare tra breve (per il $\log(1)=0$), calcoliamo anche (mettendo 2 al posto di n):

$$T(2) = \frac{2}{2-1} \sum_{q=1}^1 T(1) + 2h = \frac{2}{1} k + 2h = 2k + 2h.$$

- Ipotizziamo ora la soluzione: **$T(n) \leq an \log n$**

- Sostituiamo la soluzione innanzi tutto nel *caso base*.

Visto che $\log 1 = 0$, non possiamo utilizzare $T(1)$ e sostituiamo quindi in $T(2)$, ottenendo:

$$T(2) = 2k + 2h \leq 2a \log 2 = 2a$$

che è vera per a opportunamente grande ($a \geq k + h$).

- Per il passo induttivo possiamo scrivere:

$$\text{HP: } T(m) \leq am \log m, \forall m < n$$

$$T(n) = \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + hn \leq \frac{2}{n-1} \sum_{q=1}^{n-1} (aq \log q) + hn = \frac{2a}{n-1} \sum_{q=1}^{n-1} (q \log q) + hn$$

Valutiamo ora la sommatoria $\sum_{q=1}^{n-1}(q \log q)$, che spezziamo in due:

$$\sum_{q=1}^{n-1}(q \log q) = \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1}(q \log q) + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1}(q \log q)$$

$\leq \log \frac{n}{2} = \log n - 1$

$\leq \log n$

Dunque, possiamo scrivere:

$$\begin{aligned} \sum_{q=1}^{n-1}(q \log q) &\leq \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q (\log n - 1) + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q \log n = \\ &= (\log n - 1) \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q = \\ &= \log n \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q \leq \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\frac{n}{2}-1} q \end{aligned}$$

Nota: l'ultima diseguaglianza è vera perché $\lceil \frac{n}{2} \rceil - 1 \geq \frac{n}{2} - 1$

Ora,

$$\begin{aligned} \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\frac{n}{2}-1} q &= \log n \frac{(n-1)n}{2} - \frac{1}{2} \left(\frac{n}{2} - 1 \right) \frac{n}{2} = \\ &= \frac{1}{2} n(n-1) \log n - \frac{1}{4} \left(\frac{n}{2} - 1 \right) n \leq \\ &\leq \frac{1}{2} n(n-1) \log n - \frac{1}{4} \left(\frac{n}{2} - 1 \right) (n-1) = \\ &= (n-1) \left(\frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right) \end{aligned}$$

Ricapitolando:

$$\sum_{q=1}^{n-1}(q \log q) \leq \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\frac{n}{2}-1} q \leq (n-1) \left(\frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right)$$

Sapendo quindi che:

$$T(n) \leq \frac{2a}{n-1} \sum_{q=1}^{n-1}(q \log q) + hn$$

e che:

$$\sum_{q=1}^{n-1}(q \log q) \leq (n-1) \left(\frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right)$$

possiamo scrivere:

$$T(n) \leq \frac{2a}{n-1} (n-1) \left(\frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right) + hn = an \log n - \frac{an}{4} + \frac{a}{2} + hn$$

Scegliendo a sufficientemente grande si ha che $hn - \frac{an}{4} + \frac{a}{2} \leq 0$ e per tale a si avrà:

$$T(n) \leq an \log n - \frac{an}{4} + \frac{a}{2} + hn \leq an \log n$$

Il che ci permette di dimostrare che $T(n) = O(n \log n)$.

Poiché già sappiamo che $T(n) = \Omega(n \log n)$ perché il costo del caso migliore limita superiormente il costo computazionale degli altri casi, abbiamo che nel caso medio il Quicksort ha un costo computazionale:

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

Osservazioni.

L'analisi ora fatta è valida nell'ipotesi che il valore del pivot sia equiprobabile e, quando questo è il caso, il **quicksort** è considerato l'**algoritmo ideale per input di grandi dimensioni**.

A volte però l'ipotesi di equiprobabilità non è soddisfatta (ad esempio quando i valori in input sono "poco disordinati") e le prestazioni dell'algoritmo degradano.



Per ovviare a tale inconveniente si possono adottare delle tecniche volte a **randomizzare** la sequenza da ordinare, cioè volte a disgregarne l'eventuale regolarità interna.

Tali tecniche mirano a **rendere l'algoritmo indipendente dall'input**, e quindi consentono di ricadere nel caso medio.

Randomizzare: introduco un qualcosa che rende causale le cose.

Alcune di tali tecniche sono:

- prima di avviare l'algoritmo, alla sequenza da ordinare viene applicata una permutazione degli elementi generata casualmente;
- l'operazione di partizionamento sceglie casualmente come pivot il valore di uno qualunque degli elementi della sequenza anziché sistematicamente il valore di quello più a sinistra.

Così evitiamo che il costo dell'algoritmo dipenda dall'ordinamento della sequenza in input.

Questa fase di randomizzazione può evitare di farci cadere nel caso peggiore.

Heapsort.

L'algoritmo **Heapsort** è un algoritmo di ordinamento piuttosto *complesso* che esibisce ottime caratteristiche:

- come *Mergesort* ha un costo computazionale di $O(n \log n)$ anche nel caso peggiore ma non consente di ordinare in loco.

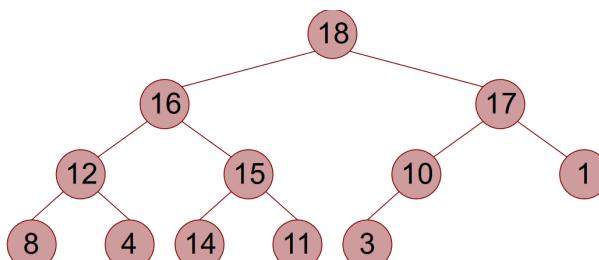
- come *Selection sort* ordina in loco.

L'algoritmo che riesce a riassumere tutte le caratteristiche positive per l'ordinamento: ordina in loco e ha costo computazionale $n \log n$ sia nel caso migliore che caso peggiore.

Sfrutta una opportuna organizzazione dei dati, ossia una struttura dati, che garantisce una o più specifiche proprietà, il cui mantenimento è **essenziale** per il corretto funzionamento dell'algoritmo.

↓
Struttura dati **Heap**

Uno **heap** è un *albero binario completo o quasi completo*, ossia un albero binario (sempre due figli) in cui tutti i livelli sono pieni, tranne l'ultimo, i cui nodi sono addensati a sinistra...



Con l'ulteriore proprietà che la chiave di ogni nodo è **maggior o uguale** alla chiave dei suoi figli (proprietà di ordinamento verticale).

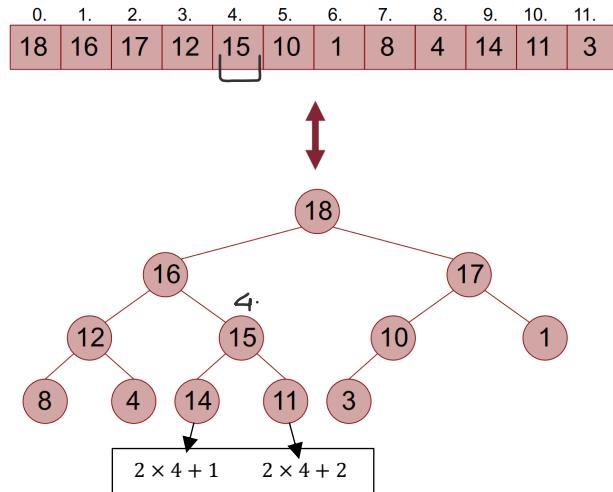
↑
Struttura dati *astratta*: definisco le proprietà che deve avere la struttura dati.

Struttura dati *concreta*: struttura dati astratta che viene memorizzata fisicamente in memoria.

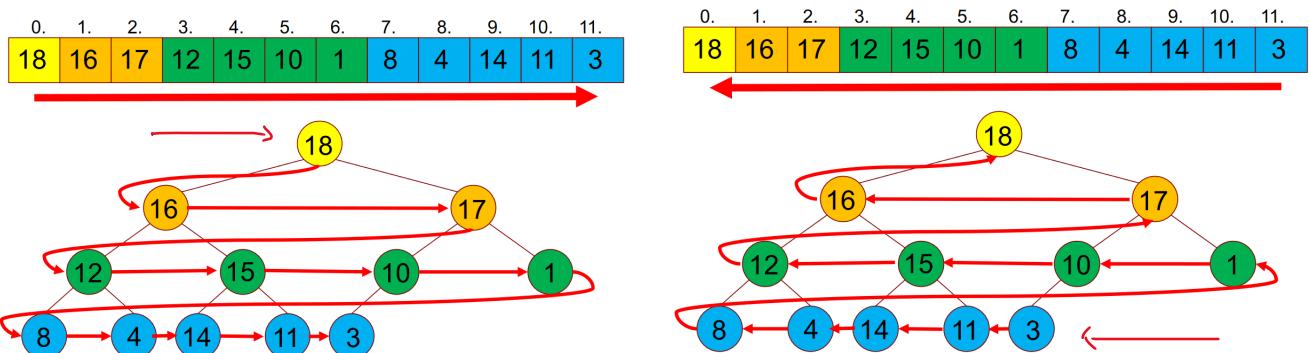
↓
Il modo più naturale per memorizzare uno heap è utilizzare:

- un array A, con indici che vanno da 0 fino al numero di nodi dell'heap, **heap_size**, diminuito di 1; gli elementi possono infatti essere messi in corrispondenza con i nodi dell'heap:
 - il vettore è **riempito a partire da sinistra**; se contiene più elementi del numero **heap_size** di nodi dell'albero, allora i suoi elementi di indice > **heap_size** non fanno parte dell'heap;
 - ogni nodo dell'albero binario corrisponde a uno e un solo elemento dell'array A;
 - la radice dell'albero corrisponde ad A[0];

- il **figlio sinistro** del nodo che corrisponde all'elemento $A[i]$, se esiste, corrisponde all'elemento $A[2i+1]$: $\text{left}(i) = 2i+1$;
- il **figlio destro** del nodo che corrisponde all'elemento $A[i]$, se esiste, corrisponde all'elemento $A[2i+2]$: $\text{right}(i) = 2i+2$;
- il **padre** del nodo che corrisponde all'elemento $A[i]$ corrisponde all'elemento $A[\lfloor(i-1)/2\rfloor]$: $\text{parent}(i) = \lfloor(i-1)/2\rfloor$.



Si noti che scorrere il vettore da sinistra a destra corrisponde a muoversi sull'albero per livelli, dall'alto verso il basso e da sinistra a destra in ciascun livello.



Simmetricamente, scorrere il vettore da destra a sinistra corrisponde a muoversi sull'albero per livelli, dal basso verso l'alto e da destra a sinistra in ciascun livello.

Proprietà:

- Poiché lo heap ha tutti i livelli completamente pieni tranne al più l'ultimo, la sua **altezza** è $\Theta(\log n)$.
- Con questa implementazione, la proprietà di ordinamento verticale implica che per tutti gli elementi tranne $A[0]$ (poiché esso corrisponde alla radice dell'albero e quindi non ha genitore) vale: $A[i] \leq A[\text{parent}(i)]$.
- L'elemento **massimo** risiede nella radice, quindi può essere trovato in tempo $O(1)$.

L'algoritmo Heapsort si avvale di due funzioni ausiliarie, necessarie per il suo corretto funzionamento:

- Funzione **Heapify** (aggiusta l'heap);
- Funzione **Buildheap** (costruisce l'heap).

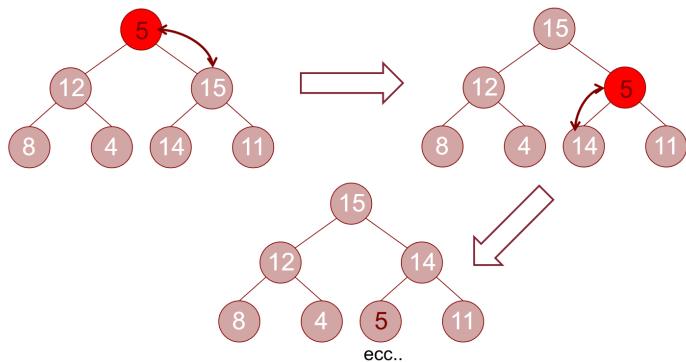
Illustreremo tali funzioni prima di descrivere l'algoritmo Heapsort.

La funzione **Heapify**.

Heapify ha lo scopo di *mantenere la proprietà di heap*, sotto l'ipotesi che nell'albero su cui viene fatta lavorare sia garantita la proprietà di heap per entrambi i sottoalberi (sinistro e destro) della radice. Di conseguenza l'unico nodo che può violare la proprietà di heap è la radice dell'albero, che può essere minore di uno o di entrambi i figli.

La funzione opera sulla radice confrontandola coi suoi figli e, se necessario, la scambia col maggiore di suoi figli. Dopo lo scambio si verifica se la violazione si sia trasferita sul figlio scambiato e, se necessario, si ripete ricorsivamente l'operazione su tale nodo.

Esempio:



```
def Heapify (A, i, heap_size)
    L=left(i); R=right(i); indice_max=i
    if ((L < heap_size) and (A[L] > A[i])):
        indice_max=L
    if ((R < heap_size) and (A[R] > A[indice_max])):
        indice_max=R
    if (indice_max ≠ i)
        A[i], A[indice_max]=A[indice_max], A[i]
        Heapify (A, indice_max, heap_size)
```

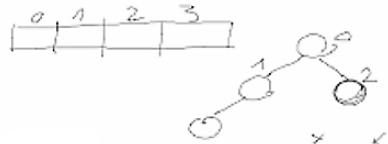
$\Theta(1)$

$T(n')$

$T(n) = \Theta(1) + T(n')$, dove n' è il numero di nodi del sottoalbero che ha più nodi.

Tutti gli oggetti hanno la proprietà di ordinamento verticale, tranne la radice. Se la proprietà non è rispettata, prendo il max dei figli e lo scambio. Ripeto il procedimento finché non arriviamo all'heap corretto.

$L < heap_size$ e $R \leq heap_size$ mi servono per vedere se i figli del nodo che sto considerando sono nell'albero.

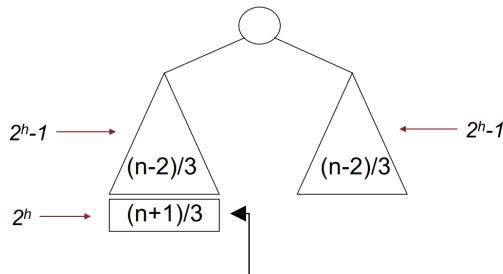


Ad esempio, i figli di 2, che saranno nell'indici 5 e 6 non sono presenti nell'albero, per questo faccio il controllo.

A questo punto all'ultimo controllo if: se il massimo non è nella radice lo vado a scambiare.

A questo punto richiamo ricorsivamente nell'albero del figlio della radice.

I sottoalberi della radice non possono avere più di $2n/3$ nodi, situazione che accade quando l'ultimo livello è pieno esattamente a metà:



C'è ancora un livello parziale in cui però le foglie si interrompono nella parte sx. Parte più sbilanciata possibile.

Quindi se ci mettiamo nel caso peggiore (lo scambio viene fatto tra la radice e la radice del sottoalbero a sinistra che ha più nodi) l'equazione di ricorrenza diventa $T(n)=T(2/3 n)+\Theta(1)$, che ha soluzione: $T(n)=O(\log n)$
O e non Θ perché non è detto che devo fare tutti i passi ricorsivi.

Il risultato non è sorprendente perché ad ogni passo scendiamo di un livello. Dato che i livelli dell'heap sono dell'ordine di $\log n$, stiamo facendo un'operazione costante al più per ogni livello.

La funzione Buildheap.

Buildheap serve per *trasformare qualunque vettore contenente n elementi in uno heap*, chiamando ripetutamente Heapify sugli opportuni nodi dello heap.

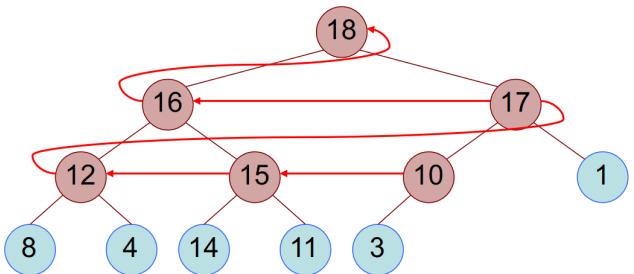
Osservazioni:

- poiché Heapify presuppone che entrambi i sottoalberi della radice siano heap (tranne la radice), deve essere chiamata scorrendo l'albero per livelli *dal basso verso l'alto* (quindi, sull'array, da destra a sinistra);
- ogni foglia è già uno heap, quindi basta chiamare Heapify a partire dal nodo interno più a destra, che ha indice:

$$\lfloor n/2 \rfloor$$

Se parto dal basso, piano piano l'aggiusto salendo verso l'alto. Le foglie sono tante quanti sono i nodi interni.

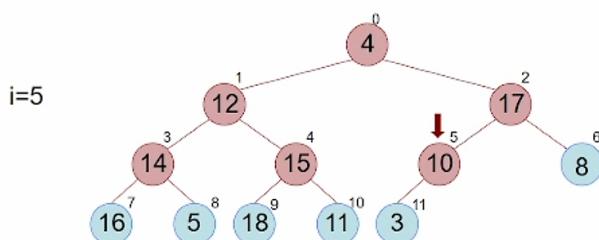
In questo esempio, Buildheap chiama
Heapify sui nodi con indice
5, 4, 3, 2, 1, 0:



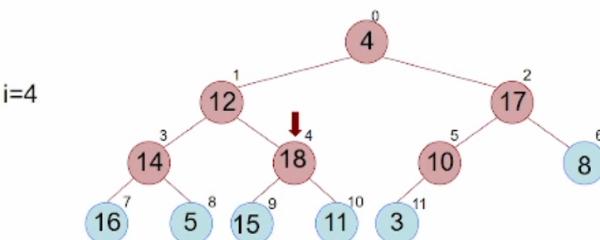
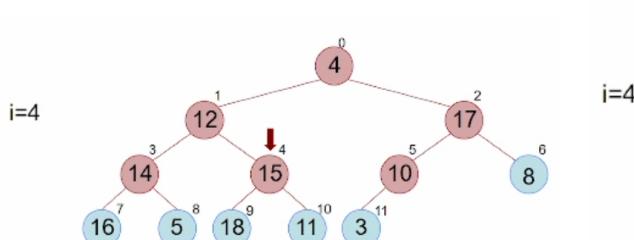
```
def Build_heap (A) :
    for i in reversed(range(len(A)//2)) :
        Heapify (A, i, heap_size)
```

Vediamo un esempio:

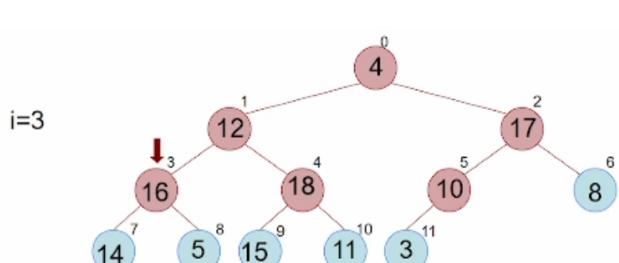
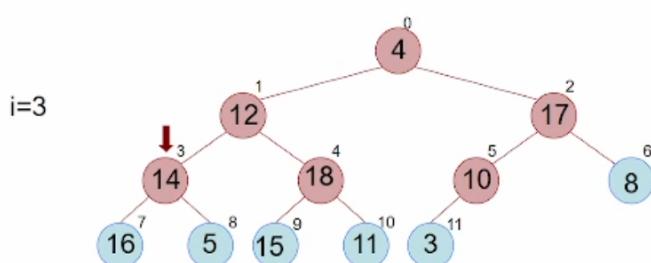
Parto dall'indice 5, quello è già un heap quindi non devo fare nulla.



L'indice 4 non rispetta le specifiche per un heap e scambio 15 con 18.

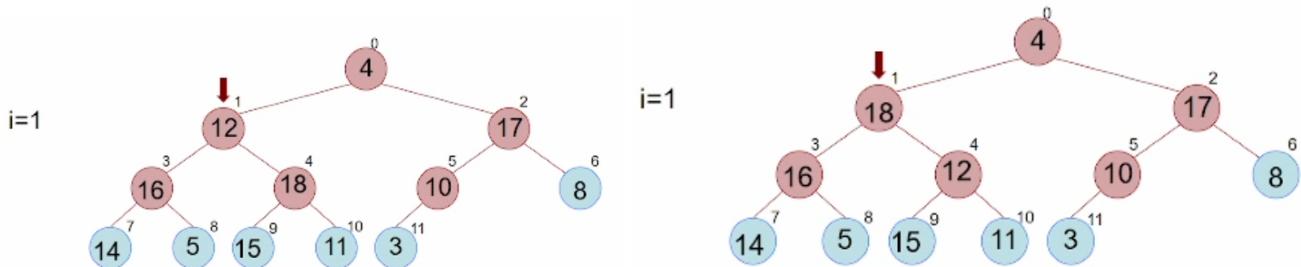


Vado avanti e scambio anche il 14 con il 16 nell' i=3.

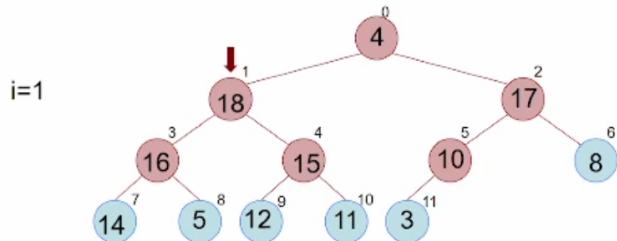


i = 2, rispetta l'heap e non faccio niente.

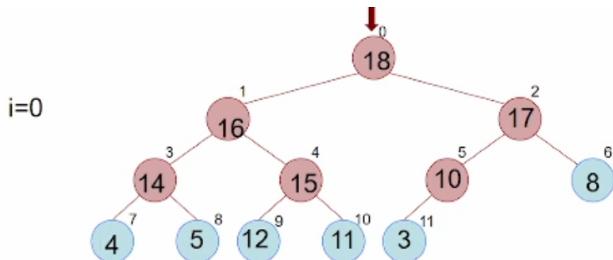
$i = 1$, non rispetta quindi scambio 12 con 18:



Ora però mi accorgo che non è soddisfatto $i = 4$ e scambio 12 con 15:



Ora sostituisco la radice e faccio altri cambi con 16 e 4, e 4 e 14.



A questo punto siamo arrivati a un heap.

La funzione Build_heap effettua $\Theta(n)$ chiamate di Heapify, che sappiamo avere ciascuna costo $O(\log n)$, quindi:

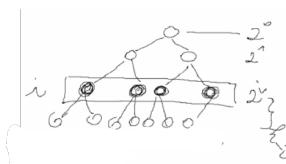
$$T(n) = O(n \log n)$$

Con un calcolo più accurato si può mostrare che $T(n)=\Theta(n)$

Mostriamo che $T(n)=O(n)$.

- Il tempo richiesto da Heapify applicata ad un nodo che è radice di un albero di altezza h è $O(h)$ per quanto già detto;
- il numero di nodi che sono radice di un sottoalbero di altezza h è al massimo:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil$$



Quindi possiamo scrivere:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

$$\text{Ricordando che } \sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

Otteniamo:

$$T(n) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O\left(\frac{n}{2} \sum_{h=0}^{\infty} \left(\frac{1}{2}\right)^h\right) = O\left(\frac{n}{2} 2\right) = O(n)$$

Se $|x| < 1$:

$$\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$$

Vediamo ora il funzionamento di Heapsort.

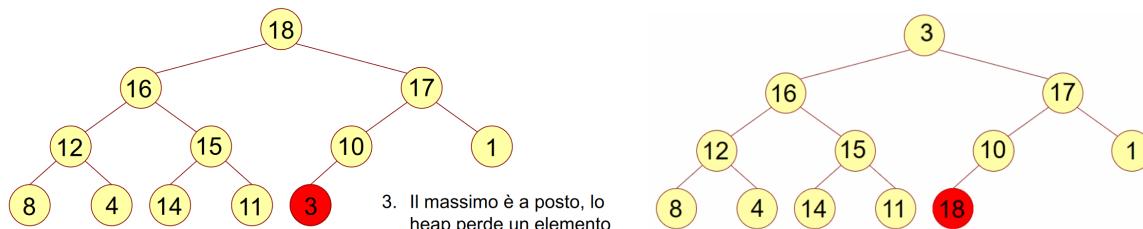
- Trasforma un vettore A di dimensione n in un heap (di n nodi), mediante Build_heap.
- Ora il max del vettore è in A[0] e, per metterlo nella corretta posizione dell'ordinamento, scambiarlo con A[n-1].
- La dim. dell'heap viene ridotta ad (n - 1), e:
 - i due sottoalberi della radice sono ancora degli heap.
 - solo la nuova radice (ex foglia più a destra) può violare la proprietà del nuovo heap di dimensione (n - 1).
- Ripristina la proprietà di heap sui residui (n - 1) elementi con Heapify;
- Scambia il nuovo max A[0] col penultimo elemento;
- Riapplica il procedimento riducendo via via la dimensione dell'heap a (n - 2), (n - 3), ecc., fino ad arrivare a 2.

Prima iterazione, Heap_size= 12



1. Scambio del massimo:

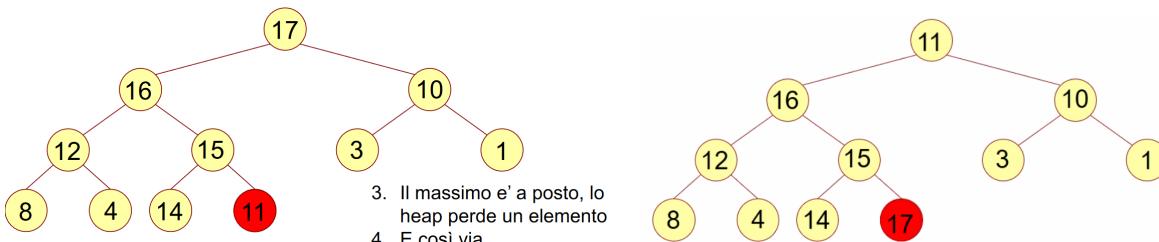
2. Lavoro di Heapify (su 11 elementi):



Seconda iterazione, Heap_size= 11



2. Lavoro di Heapify (su 10 elementi):



Vediamo ora lo pseudocodice di Heapsort:

```
def Heapsort (A):
    Build_heap(A)
    for x in reversed(range(1, len(A))) : (n-1) iteraz. + Θ(1)
        A[0], A[x]=A[x], A[0]           Θ(1)
        Heapify(A, 0, x)                O(log n)
```

$$T(n) = \Theta(n) + (n-1)(\Theta(1) + O(\log n)) + \Theta(1) = O(n \log n)$$

Ordinamenti lineari.

Algoritmi lineari.

- Abbiamo dimostrato il teorema che asserisce che ogni algoritmo di ordinamento **che opera per confronti** ha un costo computazionale di $\Omega(n \log n)$.
- Com'è possibile, allora, avere degli algoritmi di ordinamento di costo computazionale lineare?
- Si possono fare ipotesi aggiuntive che ci permettano di **evitare** che l'algoritmo sia basato sui **confronti**.
Riassumendo: ordinare n elementi in ordine di n senza confrontarli tra di loro e senza scambi.

Counting Sort.

- Ipotesi:** ciascuno degli n elementi da ordinare è un intero di valore compreso in un intervallo $[0..k]$
- Il costo computazionale è di $\Theta(n+k)$. Se $k = O(n)$ allora l'algoritmo ordina n elementi in tempo lineare, cioè $\Theta(n)$.
- Idea:** fare in modo che il valore di ogni elemento della sequenza determini direttamente la sua posizione nella sequenza ordinata.

Scorrere l'array e contare gli elementi. Una volta fatto il conteggio, sulla base di quest'ultimo posso direttamente posizionare ciascun elemento nella posizione corretta.

Esempio:

A:

0	6	7	2	5	6	1	0	4	4	1	6
---	---	---	---	---	---	---	---	---	---	---	---

$n=12, k=7$

C:

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

- 1) Scorrendo A conteggiamo in C il numero di occorrenze di ciascun valore

Nota: $\sum_{i=1}^k C[i] = n$

C:

0	1	2	3	4	5	6	7
2	2	1	0	2	1	3	1

- 2) Scorrendo C ricopiamo in A ciascun indice di C tante volte quanto è il valore in C di quell'indice

A:

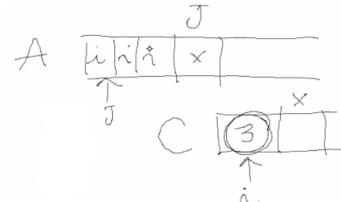
0	0	1	1	2	4	4	5	6	6	6	7
---	---	---	---	---	---	---	---	---	---	---	---

$C[i] = j$. In A ci sono j occorrenze di i .
Se sommo tutti i valori in C ottengo n .

```
def Counting_sort (A):
    k=max(A)
    Θ(n)
    n=len(A)
    Θ(1)
    C[0]* (k+1)
    Θ(k)
    for j in range(n):
        n volte
        C[A[j]] +=1
        Θ(1)
    //C[i] ora contiene il numero di elementi uguali a i
    j = 0
    Θ(1)
    for i in range(k):
        Σki=0
        C[i] volte
        while (C[i] > 0)
            A[j]=i
            Θ(1)
            j+=1
            Θ(1)
            C[i]-=1
            Θ(1)

Poiché Σki=0 C[i] = n,
```

$$T(n) = \Theta(n) + \Theta(k) + \Theta(1) + n \Theta(1) + \sum_{i=0}^k C[i] \Theta(1) = \Theta(k+n)$$



Lo pseudocodice appena illustrato è adeguato solamente se non vi sono dati satellite. Infatti, il ciclo che riscrive il vettore A di fatto ne sovrascrive i dati.

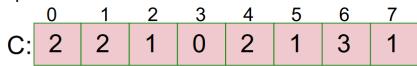
Se ci dovessero essere dati satelliti, oltre ai vettori A e C si deve introdurre un nuovo vettore B di n elementi, che alla fine, conterrà la sequenza ordinata, e si deve usare uno schema leggermente più complesso...

- Dopo aver conteggiato gli elementi in C, si fa una seconda passata su C, da sinistra a destra a partire da C[2], nella quale a ogni elemento C[i] si somma il precedente;
- alla fine di tale fase:
 - C[i] indica la posizione corretta, nel vettore ordinato, per l'elemento di valore pari a i più a destra fra quelli contenuti nel vettore A;
 - C[i] - C[i-1] indica quanti elementi ci sono con valore pari a C[i]
- si scorre quindi il vettore A da destra a sinistra e:
 - si copia in B ogni elemento A[j]=k nella posizione giusta che è C[k];
 - si decrementa di 1 il valore C[k].

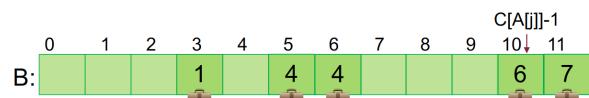
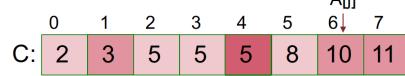
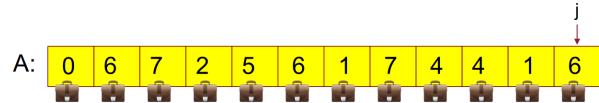
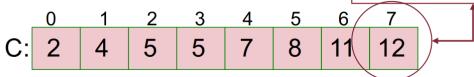
Esempio:



1) Prima passata: come nell'algoritmo precedente



2) Seconda passata: partendo da C[1], ad ogni elemento si somma il precedente



E così via...

```
Funzione Counting_sort_con_Dati_Satellite (A)
    k=max(A); n=len(A)
    C[0]* (k+1); B=[0]*n
    for j in range(n)
        C[A[j]]+=1 // in C[i] ora c'è il num. di elem. = i
    for i in range(1,k)
        C[i]+=C[i-1] // in C[i] ora c'è il num. di elem. ≤ i
    for j in range(n,-1)
        B[C[A[j]]]=A[j]
        C[A[j]]-=1
    return B
```

Il cui costo è chiaramente $T(n) = \Theta(k) + \Theta(n)$

Bucket Sort.

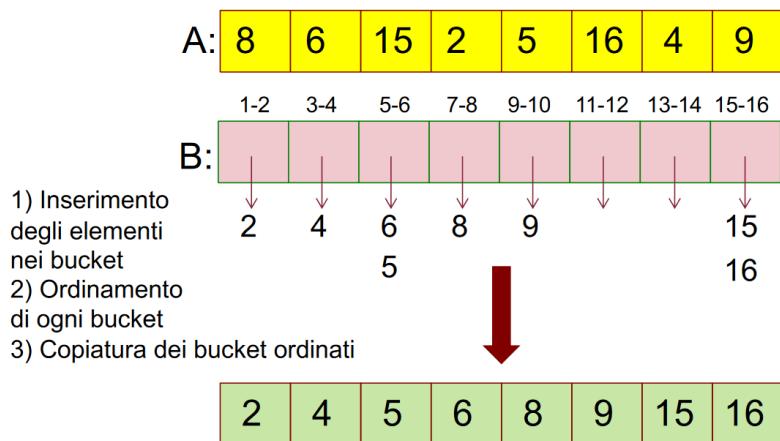
- **Ipotesi:** gli n elementi da ordinare sono **distribuiti** in modo uniforme nell'intervallo $[1..k]$, senza alcuna ipotesi su k .
- Il costo computazionale medio è di $\Theta(n)$.
- **Idea:** dividere l'intervallo $[1..k]$ in n sottointervalli di uguali dimensioni k/n , detti bucket, e distribuire i valori nei bucket.

In media un valore ogni intervallo.

- Poiché gli elementi in input sono uniformemente distribuiti, non ci si aspetta che molti elementi cadano nello stesso bucket.

Esempio:

$n=8, k=16$



```
Funzione Bucket_Sort (A; k, n: interi)
for i=1 to n                         nΘ(1)
    inserisci A[i] nella lista B[    ]
    for i=1 to n                       $\sum_{i=1}^n$ (costi di ordinamento)
        ordina la lista B[i] con insertion_sort
    concatena le liste B[1] B[2] ... B[n]   Θ(n)
                                            in quest'ordine
    Copia la lista unificata in A
```

- I costi di ordinamento dipendono dalla lunghezza delle liste.
- Ci sono n bucket ed n valori; per l'ipotesi di equiprobabilità, in media ci sarà un numero costante di valori in ogni bucket; quindi, la lista $B[i]$ ha in media lunghezza costante.
- Quindi, in media $T(n) = \Theta(n)$.

Strutture dati fondamentali: Insiemi dinamici ed operazioni su di essi.

Strutture dati.

- In un linguaggio di programmazione, un **dato** è un valore che una variabile può assumere.
 - Un **tipo di dato astratto** è un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori.
 - Le **strutture dati** sono particolari tipi di dato, caratterizzate dall'organizzazione dei dati, più che dal tipo di dato stesso.
-
- Una struttura dati è quindi composta da:
 - un **modo sistematico** di organizzare i dati;
 - un **insieme di operatori** che permettono di manipolare la struttura;
-
- Le strutture dati possono essere:
 - **lineari** o **non lineari** (a seconda che esista o no una sequenzializzazione: se i dati possono essere messi in sequenza, array è lineare)
 - **statiche** o **dinamiche** (a seconda che possano variare la dimensione nel tempo, statica non è possibile variare la dimensione, dinamica invece può farlo: può accrescere o restringere a seconda delle necessità)
 - **omogenee** o **disomogenee** (rispetto ai dati contenuti, omogenee: tutti i dati dello stesso tipo, non omogenee: possono contenere dati di diverso tipo).

L'array è statico, ma i moderni linguaggi di programmazione cercano di superare i limiti delle strutture dati; perciò, si può cambiare dimensione con delle funzioni interne.

Insiemi dinamici.

Una struttura dati serve a **memorizzare** e **manipolare** **insiemi dinamici**, cioè insiemi i cui elementi possono variare nel tempo in funzione delle operazioni compiute dall'algoritmo che li utilizza.

Gli elementi dell'insieme dinamico (spesso chiamati anche **record**) possono essere piuttosto complessi e contenere ciascuno più di un dato "elementare". In tal caso è abbastanza comune che essi contengano:

matricola	data di nascita
cognome	
nome	
esami sostenuti	

- una **chiave**, utilizzata per **distinguere un elemento da un altro** nell'ambito delle operazioni di manipolazione dell'insieme dinamico; normalmente i valori delle chiavi fanno parte di un insieme totalmente ordinato (ad. es., sono numeri interi);
- ulteriori dati, detti **dati satellite**, che sono relativi all'elemento stesso ma non sono direttamente utilizzati nelle operazioni di manipolazione dell'insieme dinamico.

In altri casi, ogni elemento contiene solo la chiave e quindi coincide con essa.

Nel seguito ci riferiremo sempre a questa situazione semplificata, a meno che non venga esplicitamente specificato il contrario.

Le **tipiche operazioni** che si compiono su un **insieme dinamico S** (e quindi sulla struttura dati che ne permette la gestione), che si suppone totalmente ordinabile, si dividono in due categorie:

- **operazioni di interrogazione**: non modificano i dati;
- **operazioni di modifica** : cambiano i dati;

Tipici esempi di operazioni di **interrogazione**, che non modificano la consistenza dell'insieme dinamico, sono:

- **Search(S, k)**: recuperare l'elemento con chiave di valore k, se è presente in S, valore speciale nullo altrimenti;
- **Min(S)/Max(S)**: recuperare il minimo/massimo valore presente in S;

- **Predecessor(S, k)/Successor(S, k)**: recuperare l'elemento presente in S che precederebbe/seguirebbe quello di valore k (di cui supponiamo di conoscere la locazione) se S fosse ordinato;

Tipici esempi di operazioni di **manipolazione**, che invece modificano la consistenza dell'insieme dinamico, sono:

- **Insert(S, k)**: inserire un elemento di valore k in S: proprietà struttura dati da mantenere nell'inserimento;
- **Delete(S, k)**: eliminare da S l'elemento di valore k (di cui supponiamo di conoscere la locazione). Anche in questo caso si devono mantenere le proprietà delle strutture dati.

Le *differenti strutture dati* che vedremo, se da un lato hanno in comune la capacità di memorizzare insiemi dinamici, dall'altro differiscono anche profondamente fra loro per le *proprietà* che le caratterizzano.

Sono proprio le proprietà della struttura dati ad essere l'elemento determinante nella scelta da effettuare quando si deve progettare un algoritmo per risolvere un problema.

Un esempio lo abbiamo già incontrato: la struttura dati Heap, senza la quale l'algoritmo Heapsort non potrebbe nemmeno essere pensato.

Arrays.

Prima di procedere allo studio di alcune interessanti strutture dati, vediamo il costo computazionale delle principali operazioni su insiemi dinamici quando questi vengano memorizzati su semplici **array**, *disordinati* o *ordinati*.

Osservazione.

L'array è considerato come una **struttura statica**: con alcuni linguaggi di programmazione (ad esempio in Python) sembra possibile variarne dinamicamente la dimensione, ma ciò viene consentito solo "simulando" la struttura dati array con una struttura dati dinamica.

Search(S, k):

- array *disordinato*: bisogna scorrere l'array: $O(n)$ → perché nel caso migliore lo trovo subito (tempo costante), nel caso peggiore dobbiamo scorrere tutto l'array ($\Theta(n)$).
- array *ordinato*: ricerca binaria: $O(\log n)$

Min(S)/Max(S):

- array *disordinato*: bisogna scorrere l'array: $\Theta(n)$, non c'è né caso migliore né caso peggiore.
- array *ordinato*: primo o ultimo elemento: $\Theta(1)$

Predecessor(S,k), Successor(S,k):

- array *disordinato*: bisogna scorrere l'array e memorizzare l'elemento più vicino che segue o precede: $\Theta(n)$.
- array *ordinato*: elemento precedente o seguente: $\Theta(1)$.

Nelle operazioni di manipolazione l'array disordinato procede meglio.

Insert(S,k)

- array *disordinato*: inserimento nella prima posizione libera: $\Theta(1)$.
- array *ordinato*: ricerca della posizione, scorrimento a destra degli elementi maggiori ed inserimento: $O(n)$, caso migliore e peggiore.

Delete(S,k)

- array *disordinato*: eliminazione e scambio con l'ultimo elemento (dato che non è ordinato): $\Theta(1)$.
- array *ordinato*: eliminazione e scorrimento a sinistra per coprire lo spazio lasciato (splitto a sx tutto): $O(n)$.

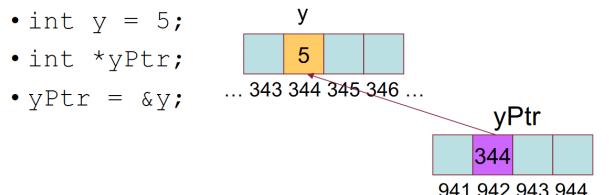
Riassumendo:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k) Successor(S,k)	Insert(S,k)	Delete(S,k)
Vettore qualunque	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(1)$
Vettore ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$

Già da questo semplice confronto delle due strutture dati più semplici in assoluto, si può dedurre come non abbia senso dire che una struttura è migliore di un'altra: le strutture dati possono essere più o meno adatte ad un certo algoritmo e sta al buon progettista disegnare un algoritmo efficiente usando la struttura dati più adatta. Per le operazioni di interrogazione è più efficiente avere un array ordinato, per quelle di manipolazione conviene avere un array disordinato.

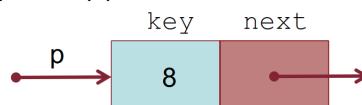
Liste puntate semplici.

- Un **puntatore** è una *variabile* che **assume** come *valore* un indirizzo di memoria.
- Il nome di una variabile fa quindi riferimento ad un valore direttamente, mentre un puntatore lo fa indirettamente.
- Esempio:



Ogni *elemento di lista* è un record a due campi:

- campo **key**: contiene l'informazione vera e propria;
- campo **next**: contiene il puntatore che consente l'accesso all'elemento successivo; nel caso dell'ultimo elemento della lista contiene un apposito valore null (o none) (visualizzato col simbolo \).



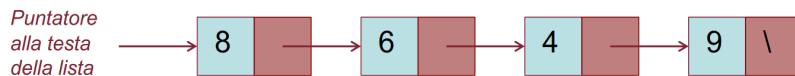
Nello pseudocodice viene utilizzata questa sintassi:

- $p->key$ indica il *contenuto* del campo **key** dell'elemento puntato da p .
- $p->next$ indica il *contenuto* del campo **next** dell'elemento puntato da p , ossia l'*indirizzo* dell'elemento successivo (o None se esso è l'ultimo elemento).

La *lista puntata semplice* è una struttura dati nella quale gli **elementi sono organizzati in successione**.

Proprietà specifiche delle liste:

- l'accesso avviene sempre ad una estremità della lista, per mezzo di un **puntatore** alla testa della lista;
- ogni elemento contiene un puntatore che consente l'accesso all'elemento successivo;
- è permesso solo un accesso sequenziale agli elementi.



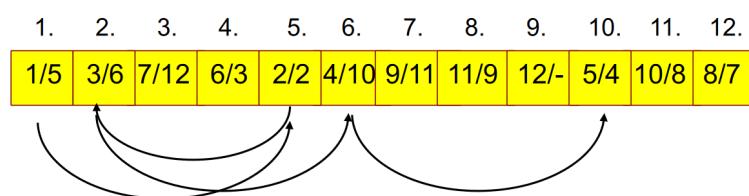
Un accesso diretto (casuale) ed accesso sequenziale.

Gli arrays, ordinati e no, godono di un **accesso diretto**:

per accedere ad un dato basta conoscerne la posizione nell'array (cioè l'indice).

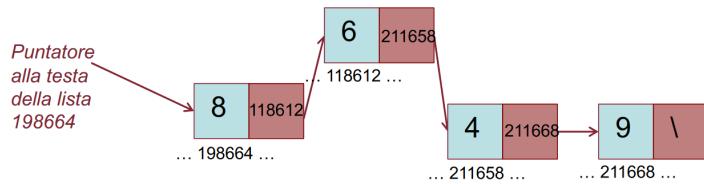
Segue che l'accesso a qualsiasi dato in un array ha un costo $\Theta(1)$.

Questo non è più vero nel solo specialissimo caso in cui ogni dato rappresenta l'elemento che lo segue nell'ordine:



In una **lista puntata**, la successione degli elementi viene implementata mediante un collegamento esplicito da un elemento ad un altro (di norma tramite un puntatore). È possibile quindi solo **l'accesso sequenziale**.

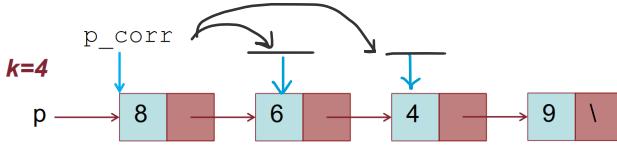
Segue che l'accesso a qualsiasi dato in una lista ha un costo proporzionale alla sua posizione, nel caso peggiore $\Theta(n)$.



```

Funzione Search (p: puntatore alla lista; k: valore)
p_corr = p
while ((p_corr != NULL) and (p_corr-> key != k))
    p_corr = p_corr-> next
return p_corr

```

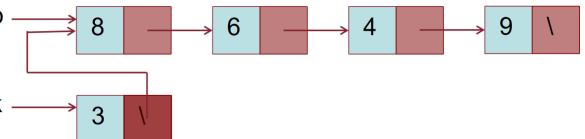


Il costo computazionale della ricerca è $O(n)$.

```

Funzione Insert_in_testa (p: puntatore alla lista;
k: punt. all'elem. da inserire)
if (k != None)
    k->next = p
    p = k
return p

```



Il costo computazionale dell'inserimento è $\Theta(1)$.

Siccome p è la testa della lista deve poi diventare uguale a k e poi viene tornato.

Nella funzione `Insert_in_testa` abbiamo preso come parametro il puntatore k all'elemento da inserire.

In generale, questo va creato, ci deve essere a monte un'**allocazione di memoria** per nuovi oggetti!!!

Questo è il trucco per far sì che la lista sia una struttura dati dinamica.

Tale creazione avviene tramite una **allocazione di memoria**.

Oltre ad inserire in testa, possiamo pensare di inserire in una posizione qualsiasi, ad esempio dopo la posizione indicata da un puntatore:

```

Funzione Insert_Dopo_d (p: punt. testa;
k: punt. a elem. da ins., d: punt. dopo cui ins.)
if d != None
    k->next ← d->next;
    d->next ← k;
    return p
else return None

```



```

Funzione Delete (p: punt. alla lista;
k: puntat. all'elem. da cancellare)
if (k != NULL) // se k=NULL non c'è niente da cancellare
    if k = p // cancel. 1° elem
        p ← p->nextp; free(k); return p
    p_corr ← p
    while (p_corr-> next ≠ k)
        p_corr ← p_corr-> next // qui, p_corr punta
        // all'elem. che precede k
    p_corr-> next ← k-> next; (free(k));
    return p

```

p_corr a 6 non a 4.

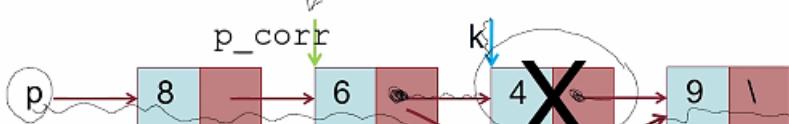
$\Theta(1)$ if

$\Theta(1)$

$\leq n-1$

$\Theta(1)$

$\Theta(1)$



Il costo computazionale della ricerca è $O(n)$.

L'operazione opposta all'allocazione è quella che libera la memoria fisicamente, oltre che logicamente.

Nei moderni linguaggi di programmazione viene fatta automaticamente, ogni qual volta una zona di memoria non sia più referenziata da alcun puntatore.

Le liste sono strutture dati inerentemente ricorsive.

Pertanto, tutti gli algoritmi proposti possono essere implementati sia in versione iterativa che ricorsiva.

Vediamo la cancellazione:

Funzione Delete_Ric(p: punt. alla lista;

k: puntat. all'elem. da canc.)

if $p==k$

$p=p->next$

free(k)

else

$p->next=Delete_Ric(p->next, k)$

return p;



Altro esempio:

Funzione Delete_Ric(p: punt. alla lista; T(n))

k: puntat. all'elem. da canc.)

if $p==k$

$p=p->next$

free(k)

else

$p->next=Delete_Ric(p->next, k)$

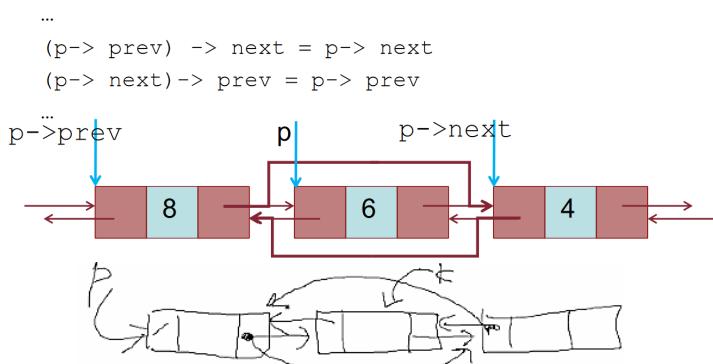
return p;

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

$$T(0)=\Theta(1)$$

Liste doppiamente puntate.

Alcuni problemi riscontrati nella lista semplice (ad esempio complessità lineare nella cancellazione) possono essere risolti organizzando la struttura dati in modo che da ogni suo elemento si possa accedere sia all'elemento che lo segue che a quello che lo precede nella lista, quando essi esistono. Tale struttura dati si chiama lista doppia o lista doppiamente concatenata o lista doppiamente puntata:



Riassumendo, per le liste puntate semplici e doppie il costo computazionale delle varie operazioni è il seguente:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k)	Insert(S,k)	Delete(S,k)
Lista semplice	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Lista doppia	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

- Esercizio.** Scrivere una funzione RICORSIVA che, preso in input un intero n , restituisce il puntatore ad una lista concatenata di n nodi contenenti nell'ordine i valori $n, n-1, n-2, \dots, 1$.

$T(n)$ def creaLista_Ric(n):
 $\Theta(1)$ if $n==0$: return None
 $\Theta(1)$ p=Nodo(n)
 $\Theta(n)$ p.next=creaLista_Ric(n-1)
 $\Theta(1)$ return p



$$\{ T(n)=T(n-1)+\Theta(1) \}$$

$$\cdot T(0)=\Theta(1)$$

Costo computazionale: $\Theta(n)$

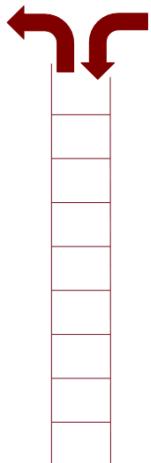
Pila.

La **pila** è una struttura dati che esibisce un comportamento **LIFO** (*Last In First Out*).

In altre parole, la pila ha la proprietà che gli elementi vengono prelevati dalla pila nell'ordine inverso rispetto a quello col quale vi sono stati inseriti.

La pila può essere visualizzata come una *pila di piatti*: ne aggiungiamo uno appoggiandolo sopra quello in cima alla pila, e quando dobbiamo prenderne uno preleviamo quello più in alto.

Un esempio di utilizzo di questa struttura dati è la pila di sistema, con la quale vengono tra l'altro gestite le chiamate a funzione (ricorsive e non).



Su una pila sono definite solo due operazioni:

- **l'inserimento** (che di norma viene chiamata Push);
- **l'estrazione** (che di norma viene chiamata Pop).

Non è previsto né scandire gli elementi di una pila né eliminare elementi con mezzi diversi dalla Pop.

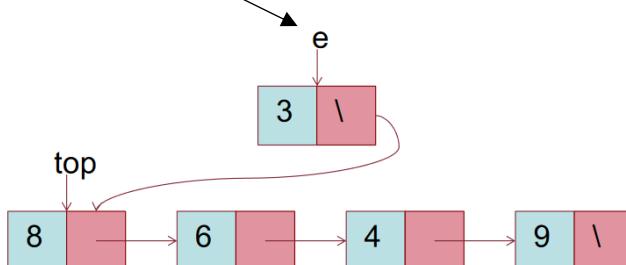
La particolarità che garantisce la proprietà LIFO è che le operazioni Push e Pop operano sulla **stessa estremità** della pila (attraverso il puntatore **top**).

Pila implementata con le liste:

```
fun Push(top: punt.; e: punt. all'elem. da inserire)
```

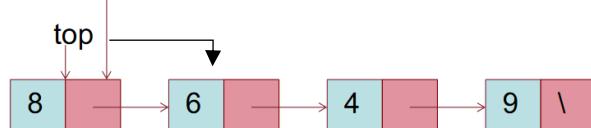
```
    e->next = top  
    top = e  
    return top
```

top = e; perché e diventa il primo elemento, cioè e.



Pila implementata con le liste (segue):

```
fun Pop (top: puntatore)  
    if (top==None) //la coda è vuota  
        scrivi "Errore: coda vuota" e return None  
    e = top  
    top = e->next  
    e->next=None  
    e return e, top
```



Il costo computazionale di entrambe le operazioni, Push e Pop, è $\Theta(1)$.

Coda.

La **coda** è una struttura dati che esibisce un comportamento **FIFO** (*First In First Out*).

In altre parole, la coda ha la proprietà che gli elementi vengono da essa prelevati esattamente nello stesso ordine col quale vi sono stati inseriti.

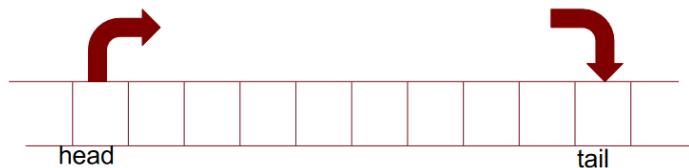
La coda può essere visualizzata come una coda di persone in attesa ad uno sportello ed uno dei suoi più classici utilizzi è la gestione della coda di stampa, in cui documenti mandati in stampa prima vengono stampati prima.

Su una coda sono definite solo due operazioni:

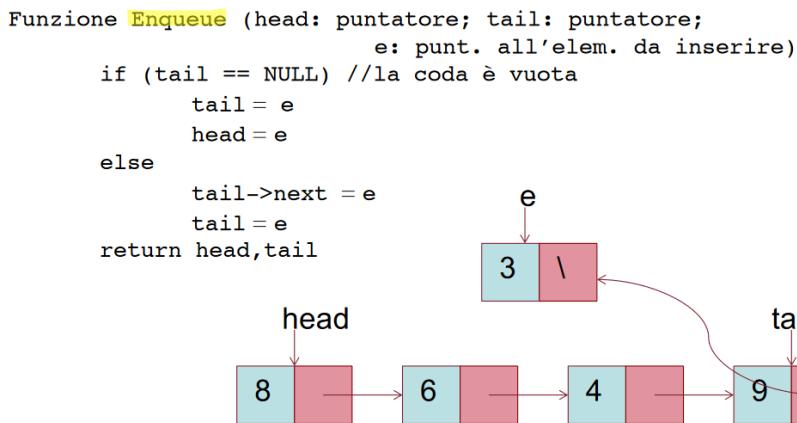
- l'**inserimento** (che di norma viene chiamata *Enqueue*)
- l'**estrazione** (che di norma viene chiamata *Dequeue*).

Di solito non si scandiscono gli elementi di una coda né si eliminano elementi con mezzi diversi dalla *Dequeue*.

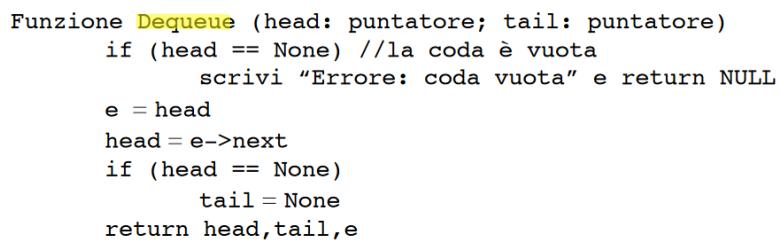
La particolarità che garantisce la proprietà FIFO è che l'operazione *Enqueue* opera su una estremità della coda (*tail*) e la *Dequeue* opera sull'altra estremità (*head*)



Coda implementata con le liste:



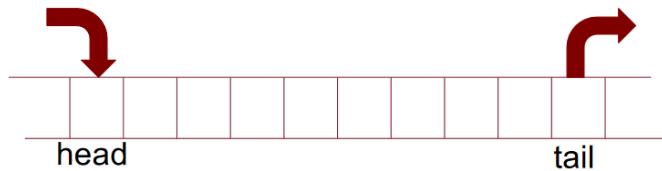
Coda implementata con le liste (segue):



Coda implementata con le liste (segue):

Il **costo computazionale** di entrambe le operazioni, *Enqueue* e *Dequeue*, è $\Theta(1)$.

Si noti che se decidessimo di estrarre dalla coda ed inserire in testa, sulla struttura dati così come mostrata (senza campo prec) l'operazione di *Dequeue* richiederebbe costo $\Theta(n)$.



Coda implementata con gli *arrays*:

Le code possono essere realizzate mediante arrays se il numero massimo di elementi è noto a priori.

Problema: a seguito di ripetute operazioni di Enqueue e Dequeue, gli elementi della coda si spostano via via verso una delle due estremità del vettore, e quando la raggiungono non vi è apparentemente più spazio per i successivi inserimenti.

Soluzione: gestire il vettore in modo circolare, considerando cioè il primo elemento come successore dell'ultimo.

Code con priorità.

- La coda con priorità è una *variante della coda*.
- Come nella coda, l'inserimento avviene ad un'estremità e l'estrazione avviene all'estremità opposta.
- A differenza della coda, la posizione di ciascun elemento non dipende dal momento in cui è stato inserito, ma dal valore di una determinata grandezza, detta priorità, la quale in generale è associata ad uno dei campi presenti nell'elemento stesso.
- Quindi, gli elementi di una coda con priorità sono collocati in ordine crescente (o decrescente, a seconda dei casi) rispetto alla grandezza considerata come priorità.

Esempi:

- La priorità è associata al valore numerico del campo key, quando un nuovo elemento avente key = x viene inserito in una coda con priorità (crescente) esso viene collocato come predecessore del primo elemento presente in coda che abbia key $\geq x$.
- In questo senso, un array ordinato è una coda con priorità, e la priorità coincide con la chiave.
- Anche la struttura dati heap è una coda con priorità rispetto alla stessa priorità.
- Anche una coda può essere intesa come una coda con priorità, ma qui la priorità è il maggior tempo di permanenza nella struttura dati.
- La pila è una coda con priorità dettata dal minor tempo di permanenza nella struttura.

La coda con priorità presenta un potenziale pericolo di starvation (attesa illimitata): un elemento potrebbe non venire mai estratto, se viene continuamente scavalcato da altri elementi di priorità maggiore che vengono via via immessi nella struttura dati.

Pile e code.

In entrambe le strutture, si può verificare se esse sono vuote, implementando le funzioni di *PilaVuota* e *CodaVuota*.

Esse prendono come parametro la struttura dati e restituiscono **True** se essa è vuota e **False** altrimenti.

Chiaramente, è possibile effettuare un'estrazione (Dequeue o Pop) solo se la struttura relativa non è vuota.

Si può anche verificare se una delle due strutture dati sia piena, tramite le funzioni di *PilaPiena* e *CodaPiena*.

Analogamente, esse prendono come parametro la struttura dati e restituiscono **True** se essa è piena e

False altrimenti.

Ha senso effettuare questo controllo SOLO se la pila o la coda sono implementate su un array.

Infatti, come è noto, una lista non potrà mai essere piena.

Alberi.

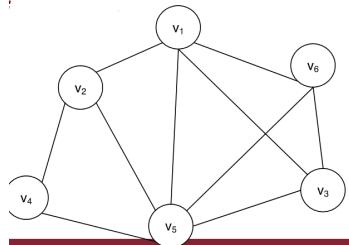
L'**albero** è una struttura dati estremamente versatile, utile per modellare una grande quantità di situazioni reali e progettare le relative soluzioni algoritmiche.

Abbiamo già incontrato la struttura ad albero (in particolare ad albero binario) varie volte, ma l'abbiamo sempre considerata in modo intuitivo.

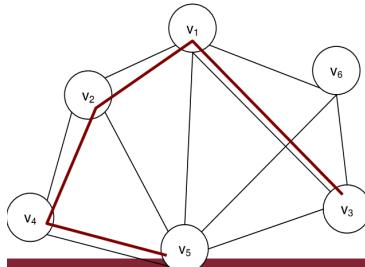
Per dare la definizione formale di albero è necessario prima fornire alcune definizioni relative ad un'altra struttura dati, il **grafo**:

Un **grafo** $G = (V, E)$ è costituito da una coppia di insiemi:

- un insieme finito **V** dei **nodi**, o **vertici**;
- un insieme finito **E** $\subseteq V \times V$ di **coppie non ordinate di nodi**, dette **archi** o **spigoli**.

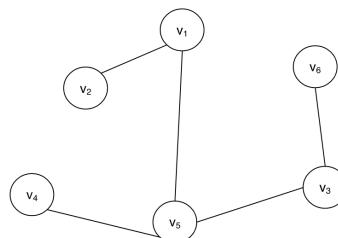


- Un **cammino** in un grafo $G = (V, E)$ è una sequenza (v_1, v_2, \dots, v_k) di nodi distinti di V tale che (v_i, v_{i+1}) sia un arco di E per ogni $1 \leq i \leq k-1$.
- Se nel cammino (v_1, v_2, \dots, v_k) i nodi v_k e v_1 coincidono, si parla di **ciclo**.
- Un grafo G è **connesso** se, per ogni coppia di nodi (u, v) , esiste un cammino tra u e v .
- Un grafo G è **aciclico** se non contiene cicli.



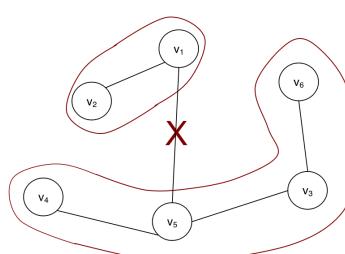
Definizione:

Un **albero** è un **grafo** $G = (V, E)$ connesso e aciclico.



Lemma.

Sia $G = (V, E)$ un grafo connesso aciclico; eliminando da G un arco qualsiasi, G si disconnette, cioè si suddivide in due grafi $G1 = (V1, E1)$ e $G2 = (V2, E2)$, entrambi connessi e aciclici.

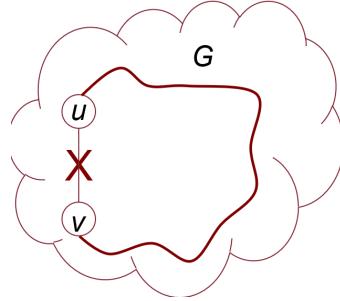


Dimostrazione. Per assurdo, dopo l'eliminazione dell'arco $e = (u, v)$ il grafo rimane connesso.

Cioè, nel nuovo grafo esiste un cammino da u a v .

Ma allora, nel grafo originario G , tale cammino, con e , forma un ciclo, contro l'ipotesi che G sia aciclico.

Infine, banalmente entrambe le componenti generate devono rimanere connesse e acicliche.



Caratterizzazione per gli alberi.

Teorema. Sia $G = (V, E)$ un grafo. Le seguenti due affermazioni sono equivalenti:

1. G è connesso e aciclico (in altre parole, G è un albero).
2. G è connesso ed $|E| = |V| - 1$. (numero archi = numero nodi -1)

Dim. 1. \rightarrow 2. (Assumo che 1 sia vero e voglio dimostrare che 2 è vero)

Dimostreremo per induzione che, se G è connesso e aciclico, allora $|E| = |V| - 1$.

Passo base: se $|V| = 1$ oppure $|V| = 2$ l'affermazione è banalmente vera.

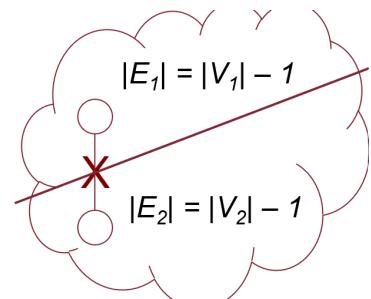
Passo induttivo: rimuovendo un arco qualsiasi, per il lemma provato precedentemente, il grafo G si disconnette in due grafi $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, entrambi connessi e aciclici.

Per essi vale l'hyp induttiva:

- $|V| = |V_1| + |V_2|$
- $|E| = |E_1| + |E_2| + 1 =$
- $= |V_1| - 1 + |V_2| - 1 + 1 = |V| - 1$

Abbiamo dimostrato che il numero di archi è uguale al numero di nodi -1.

segue dim. G è connesso ed $|E| = |V| - 1 \Rightarrow G$ è connesso e aciclico



2. \Rightarrow 1. (Assumo che 2 sia vero e voglio dimostrare che 1 è vero)

$G = (V, E)$ connesso, con $|V|=n$ e con $|E| = |V| - 1$ per assurdo contenga un ciclo $v_1, v_2, \dots, v_k, v_1$.

Consideriamo il grafo $G_k = (V_k, E_k)$ costituito dal solo ciclo.

In esso abbiamo: $|E_k| = |V_k|$

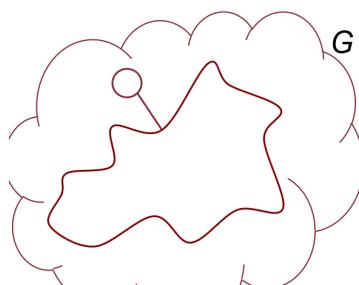
Se $k=n$ assurdo.

Se $k < n$ esiste un nodo v_{k+1} connesso a G_k tramite un arco $\Rightarrow G_{k+1}$ con $|E_{k+1}| = |V_{k+1}|$.

Si prosegue fino a G_n per cui:

$V = V_n$ ed $E \supseteq E_n \Rightarrow |E| \geq |E_n|$

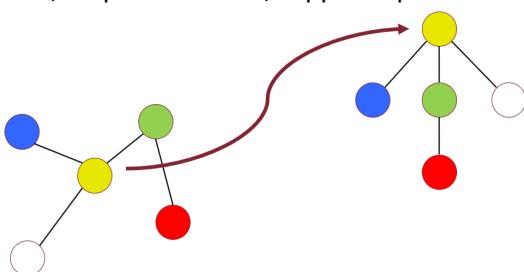
Per ipotesi $|E| = |V| - 1$, per cui $|V| - 1 = |E| \geq |E_n| = |V_n| = |V|$. Assurdo.



Alberi radicati.

Alberi radicati: vi si distingue un nodo particolare tra gli altri, detto **radice**.

L'albero radicato si può rappresentare in modo tale che i cammini da ogni nodo alla radice seguano un percorso dal basso verso l'alto, come se l'albero venisse, in qualche modo, "appeso" per la radice.



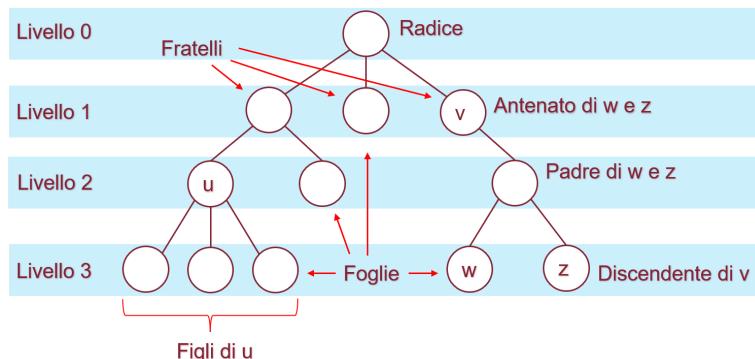
In un albero radicato:

- i nodi sono organizzati in **livelli**, numerati in ordine crescente allontanandosi dalla radice (di norma la radice è posta a livello zero);
- l'**altezza** di un albero radicato è la lunghezza del più lungo cammino dalla radice ad una foglia; un albero di altezza h contiene $(h + 1)$ livelli, di norma numerati *da 0 ad h* .

In un *albero radicato*:

- Dato un qualunque nodo v di un albero radicato che non sia la radice, il primo nodo che si incontra sul (- l'unico) cammino da v alla radice viene detto **padre di v**
- nodi che hanno lo stesso padre sono detti **fratelli** e la radice è l'unico nodo che non ha padre
- ogni nodo sul cammino da v alla radice viene detto **antenato di v**
- tutti i nodi che ammettono v come padre sono detti **figli di v** , ed i nodi che non hanno figli sono detti **foglie**;
- tutti i nodi che ammettono v come antenato vengono detti **discendenti di v** .

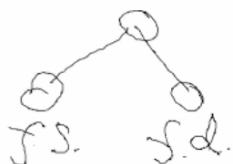
Alberi radicati (esempio, $h = 3$)



Un albero radicato si dice **ordinato** se attribuiamo un qualche ordine ai figli di ciascun nodo, nel senso che se un nodo ha k figli, allora vi è un figlio che viene considerato primo, uno che viene considerato secondo, ..., uno che viene considerato k -esimo.

Una particolare sottoclassificazione di alberi radicati ordinati è quella degli **alberi binari**, che hanno la particolarità che ogni nodo ha al più **due figli**.

Poiché sono alberi ordinati, i due figli di ciascun nodo si distinguono in **figlio sinistro** e **figlio destro**.

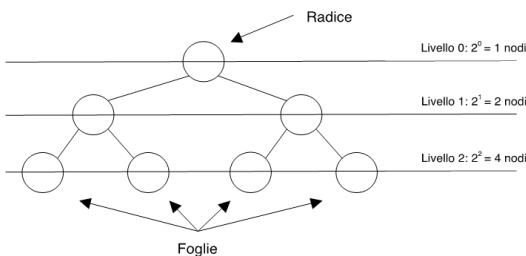


Albero binario.

Un albero binario nel quale tutti i livelli contengono il massimo numero possibile di nodi è chiamato **albero binario completo**.

Se invece tutti i livelli tranne l'ultimo contengono il massimo numero possibile di nodi mentre l'ultimo livello è riempito completamente da sinistra verso destra solo fino ad un certo punto, l'albero è chiamato **albero binario quasi completo**.

In un albero binario completo di altezza h :



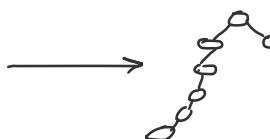
- il numero delle foglie è 2^h
- il numero dei nodi interni è $\sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1$
- il numero totale dei nodi è $2^h + 2^h - 1 = 2^{h+1} - 1$.

Di conseguenza, l'altezza h di un albero binario completo è calcolabile come segue:

Il numero totale dei nodi è: $n = 2^{h+1} - 1$, da cui segue che: $\log(n + 1) = h + 1$ e quindi: $h = \log(n + 1) - 1 = \log((n + 1)/2)$

Per questo $h = \Theta(\log n)$ vale solo per alberi binari completi.

$= \Theta(n)$ nel caso in cui non sono completi, sbilanciato.



In generale $\rightarrow h = \Omega(\log n)$ e $h = \Theta(n)$

Rappresentazione in memoria.

Memorizzazione tramite record e puntatori:

Il modo più naturale di rappresentare e gestire gli alberi binari è per mezzo dei puntatori.

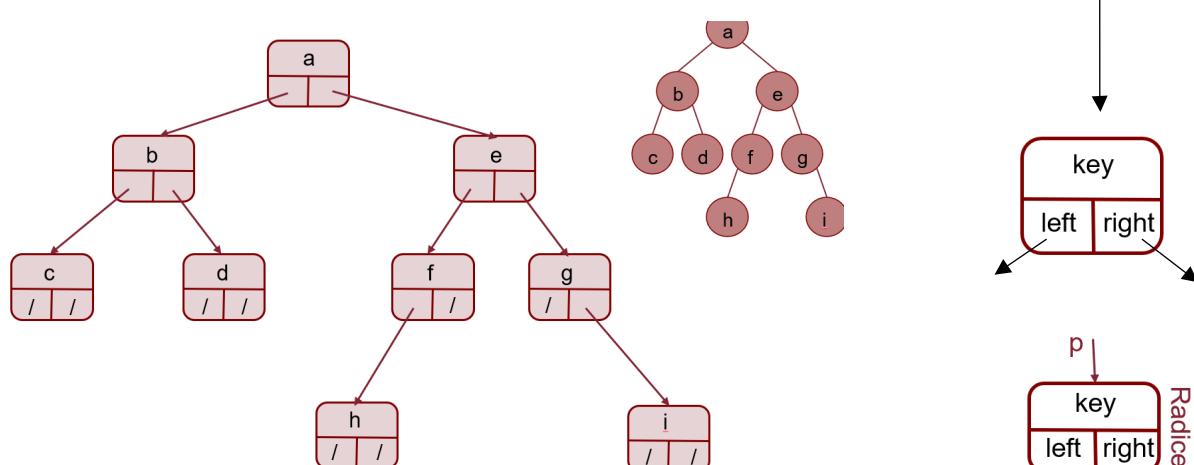
Ogni singolo nodo è costituito da un record contenente:

key: le opportune informazioni pertinenti al nodo stesso;

left: il puntatore al figlio sinistro (oppure **NULL** se il nodo non ha figlio sinistro);

right: il puntatore al figlio destro (oppure **NULL** se il nodo non ha figlio destro);

L'albero viene acceduto per mezzo del puntatore alla radice.



Ha tutti i vantaggi e l'elasticità delle strutture dinamiche basate sui puntatori (si possono inserire nuovi nodi, spostare dei nodi, ecc.)

Ma ne presenta svantaggi moltiplicati: l'unico modo per accedere all'informazione memorizzata in un nodo è scendere verso di esso partendo dalla radice e poi spostandosi di padre in figlio, ma non è chiaro se ad ogni passo si debba andare verso il figlio sinistro o verso il figlio destro.

Rappresentazione posizionale:

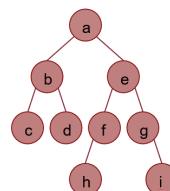
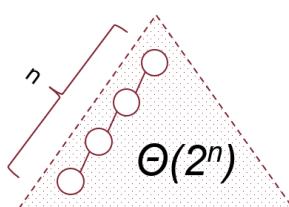
Lo abbiamo già discusso in merito allo heap.

I nodi vengono memorizzati in un array, nel quale la radice occupa la posizione di indice 0 ed i figli sinistro e destro del nodo in posizione i si trovano rispettivamente nelle posizioni $2i-1$ e $2i$. (secondo me $2i+1$, $2i+2$)

Svantaggi rispetto alla gestione mediante puntatori:

richiede di conoscere in anticipo la massima altezza h dell'albero e, una volta noto tale valore, richiede l'allocazione di un array in grado di contenere un albero binario completo di altezza h (quindi un array contenente $2^{h+1} - 1$ elementi); a meno che l'albero non sia abbastanza "denso" di nodi, si verifica uno spreco di memoria:

Albero «degenero»:



0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.
a	b	e	c	d	f	g	-	-	-	-	h	-	-	i

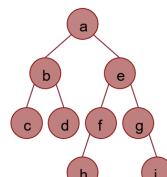
Vettore dei padri:

Costituito da un array P in cui ogni elemento è associato ad un nodo dell'albero.

Introducendo una biiezione tra gli n nodi dell'albero e gli indici $0, \dots, n-1$, l'elemento $P[i]$ dell'array P contiene l'indice del padre del nodo i nell'albero.

Questo metodo di memorizzazione funziona senza alcuna modifica anche per alberi non necessariamente binari, in cui cioè ogni nodo può avere un numero qualunque di figli.

Vettore dei padri (segue):



	0	1	2	3	4	5	6	7	8
array delle chiavi	a	b	c	d	e	f	g	h	i
array P	/	0	1	1	0	4	4	5	6

Confronto fra le strutture dati: come trovare il padre di un nodo

Struttura a puntatori:

- al momento non siamo in grado di farlo: dobbiamo accedere all'albero tramite il puntatore alla sua radice, ma poi non possiamo scorrere la struttura come fosse una lista, perché non sappiamo se dirigerci a destra o a sinistra → **visite**.

Rappresentazione posizionale:

- il padre del nodo i è banalmente in posizione $\left\lfloor \frac{i-1}{2} \right\rfloor \rightarrow \Theta(1)$

Vettore dei padri:

- L'indice del padre di ogni nodo i è memorizzato direttamente nell'elemento $P[i]$ dell'array → $\Theta(1)$

Confronto fra le strutture dati: determinare se il nodo abbia 0, 1 o 2 figli

Struttura a puntatori:

- si verifica se i campi left e right siano settati a None oppure no → $\Theta(1)$

Rappresentazione posizionale:

- si vede se gli elementi di indice $2i-1$ e $2i$ sono settati a '-' oppure no → $\Theta(1)$

Vettore dei padri:

- Si deve scorrere l'intero array P e contarvi il numero di occorrenze dell'elemento i (l'indice del nodo di cui vogliamo contare i figli) → $\Theta(n)$

Confronto fra le strutture dati: determinare la distanza di un nodo dalla radice

Struttura a puntatori:

- come nel caso della ricerca del padre di un nodo → **visite**.

Rappresentazione posizionale:

- il livello del nodo i (e quindi la sua distanza dalla radice) è banalmente $\lfloor \log(i+1) \rfloor \rightarrow \Theta(1)$

Vettore dei padri:

- vettore dei padri: a partire da i risaliamo di padre in padre passando per $P[i], P[P[i]], P[P[P[i]]],$ ecc. fino a giungere alla radice → $\Theta(h)$

Esercizio.

Dire quant'è la massima lunghezza di un array che è necessario allocare per poter memorizzare sempre un albero con n nodi con la rappresentazione posizionale.

Soluzione.

Un albero **di n nodi** può avere, nel caso peggiore, altezza **($n-1$)**, nel caso sia un albero degenere (nel quale ogni nodo ha un solo figlio);

per memorizzare un albero di **altezza h** con la notazione posizionale è necessario predisporre un vettore di $2^{h+1} - 1$ elementi;

dunque, è necessario un array di $2^n - 1$ elementi.

Visite di alberi.

Un'operazione basilare sugli alberi è **l'accesso a tutti i suoi nodi**, uno dopo l'altro, al fine di poter effettuare una specifica operazione (che dipende ovviamente dal problema posto) su ciascun nodo.

Leggere tutte le informazioni di ogni nodo.

Tale operazione sulle liste si effettua con una semplice iterazione, ma sugli alberi la situazione è più complessa dato che la loro struttura è ben più articolata, si deve scegliere se andare a sx o a dx.

L'accesso progressivo a tutti i nodi di un albero si chiama **visita dell'albero**.

Facendo riferimento all'ordine col quale si accede ai nodi dell'albero, è evidente che esiste più di una possibilità.

Nel caso degli *alberi binari*, nei quali i figli di ogni nodo (e quindi i sottoalberi) sono al massimo due, e volendo comportarsi nello stesso modo su tutti i nodi, abbiamo tre diverse visite:

- **visita in preordine (preorder)**: il **nodo** è visitato prima di proseguire la visita nei suoi sottoalberi;

Prima il nodo, poi figli sinistri poi destri. Ricorsivamente si ripete sul sottoalbero di sx e dx.

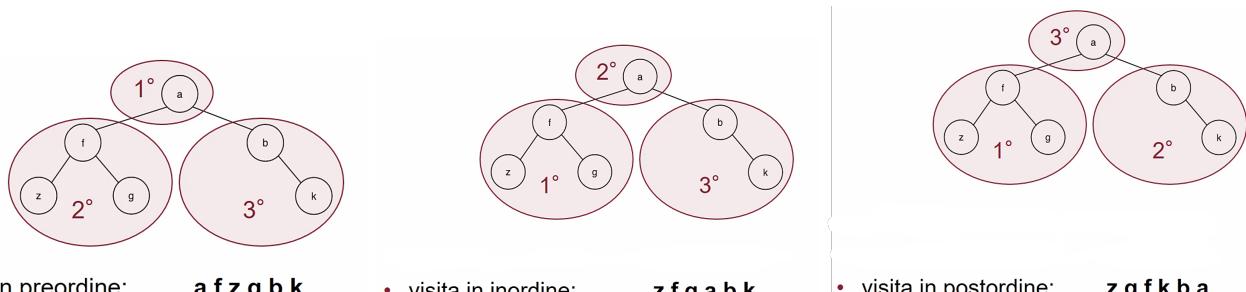
- **visita inordine (inorder)**: il **nodo** è visitato dopo la visita del sottoalbero sinistro e prima di quella del sottoalbero destro;

Prima sinistra, il nodo e poi figlio destro, ricorsivamente nei sottoalberi.

- **visita postordine (postorder)**: il **nodo** è visitato dopo entrambe le visite dei sottoalberi.

Prima figlio sinistro, poi destro e infine nodo.

Visitare un nodo significa posizionarsi sul nodo, leggere l'informazione e spostarsi sui figli.



Se l'albero è memorizzato tramite record e puntatori ed è quindi dato tramite il puntatore *p* alla sua radice:

```
def Visita_preordine (p) :
    if (p != None)
        accesso al nodo e operazioni conseguenti
        inordine
        Visita_preordine (p->left)
        Visita_preordine (p->right)
    postordine
    return
```

Le altre visite sono analoghe.

L'unica differenza fra i tre casi è la posizione della pseudo-istruzione relativa all'accesso al nodo per effettuarvi le operazioni desiderate.

Costo computazionale delle visite.

È, ovviamente, **lo stesso per tutte e tre**.

Esso varia al variare della struttura dati utilizzata per memorizzare l'albero. Nel caso di memorizzazione tramite **record e puntatori**, detto *k* il numero di nodi del sottoalbero sinistro della radice e a destra *n-k-1* nodi, l'equazione di ricorrenza è:

$$T(n) = T(k) + T(n - k - 1) + \Theta(1)$$

$$T(0) = \Theta(1) \quad \rightarrow p == \text{None}$$

Non siamo in grado di risolvere questa equazione con gli strumenti dati, quindi: **metodo di sostituzione**.

Facciamoci un'idea della possibile soluzione...

Costo computazionale delle visite – caso migliore.

Si verifica quando l'albero è completo, poiché la suddivisione tra le due chiamate ricorsive è la più equa possibile. In tal caso, se h è il numero dei suoi livelli, si ha $n = 2^{h+1} - 1$ ed entrambi i sottoalberi di ogni nodo sono completi. L'equazione, quindi, diviene: $T(n) = 2T(n/2) + \Theta(1)$ che ricade nel caso 1 del teorema principale, ed ha quindi soluzione: $T(n) = \Theta(n^{\log 2}) = \Theta(n)$

Costo computazionale delle visite – caso peggiore.

Si verifica quando $k = 0$ (oppure, simmetricamente, quando $n - k - 1 = 0$, cioè quando l'albero è sbilanciato da una parte, o tutto a dx o tutto a sx), per il quale si ottiene: $T(n) = T(n-1) + \Theta(1)$ che ha banalmente, ad esempio col metodo iterativo, la soluzione: $T(n) = n \Theta(1) = \Theta(n)$

Costo computazionale delle visite – caso generale.

Il costo computazionale nel caso generale è limitato inferiormente da quello del caso migliore, quindi è $\Omega(n)$, e superiormente da quello del caso peggiore, quindi è $O(n)$.

Di conseguenza, anch'esso è $\Theta(n)$.

Dimostriamolo formalmente, risolvendo l'equazione con il *metodo di sostituzione*.

1. Eliminiamo prima la notazione asintotica:

- $T(n) = T(k) + T(n-1-k) + c$
- $T(1) = d = T(0)$

per due costanti positive c e d note.

2. Tentiamo la soluzione $T(n) \leq an$, dove a è una costante da determinare.

Passo base: $d = T(1) \leq a$ vera sse $a \geq d$

Passo induttivo: $T(n) \leq ak + a(n-1-k) + c = a(n-1) + c = an - a + c \leq an$ vera sse $a \geq c$

Abbiamo dimostrato che $T(n) = O(n)$

Si dimostra analogamente che $T(n) \geq bn$, quindi che $T(n) = \Omega(n)$. Ne segue che $T(n) = \Theta(n)$

Domanda: avremmo potuto scrivere l'equazione di ricorrenza in termini di h ?

In altre parole, è giusto scrivere questo?

- $T(h) = 2T(h-1) + \Theta(1)$
- $T(1) = \Theta(1)$

Questa equazione ha costo $T(h) = \Theta(2^h)$ ma non siamo in grado di mettere in relazione esatta n ed h in un albero binario qualsiasi; sappiamo solo che $h = O(n)$, che porterebbe a $T(n) = O(2^n)$, certamente corretto ma inutile!

Applicazioni delle visite.

Le **visite** sono estremamente utili per *ispezionare l'albero e dedurne delle proprietà*.

A seconda delle proprietà che si vuole esaminare può essere più utile una delle tre visite considerate.

Esempio: **conteggio del numero dei nodi**. (con rappresentazione record e puntatori)

```
def Calcola_n (p) :  
    if (p != None) :  
        num_l = Calcola_n (p->left)  
        num_r = Calcola_n (p->right)  
        num = num_l + num_r + 1           #accesso al nodo  
        return num  
    return 0
```

N.B. segue la filosofia della visita in postordine! Al posto delle 4 istruzioni nell'if possiamo scrivere:

```
return 1+Calcola_n(p->left)+Calcola_n(p->right)
```

L'approccio della visita postordine è molto utilizzato per ottenere informazioni sull'albero.

Esempio: *ricerca in un albero*.

```
def Cerca (p):  
    if (p ≠ None):  
        if p->info== k return TRUE  
    else:  
        if Cerca (p->left,k)==TRUE:  
            return TRUE  
        else: return Cerca (p->right,k)  
    return FALSE
```

N.B. segue la filosofia della visita in preordine!

Esercizio proposto: modificare il codice in modo da restituire il puntatore al nodo che contiene la chiave k oppure none se la chiave k non è presente.

Esempio: *calcolo dell'altezza dell'albero* (stabiliamo che un albero costituito di un singolo nodo ha altezza zero).

```
def Calcola_h (p):  
    if (p==None): return -1 #albero vuoto  
    if (p->left==None AND p->right==None): return 0  
    h = max{Calcola_h(p->left),Calcola_h(p->right)}  
    return h + 1
```

N.B. segue la filosofia della visita in postordine! Sx, dx e poi il +1 per il nodo.

Esempio: *conteggio dei nodi presenti nel livello k* (la radice è a livello 0).

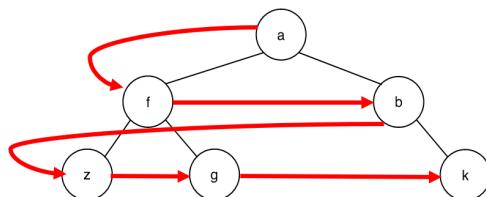
```
def Conta_k (p; k,i: interi):  
    if (p == None): return 0 #albero vuoto  
    if (k==i): return 1  
    k_left = Conta_k(p->left,k,i+1)  
    k_right = Conta_k(p->right,k,i+1)  
    return k_left + k_right
```

La chiamata iniziale nel main() sarà nk = Conta_k(radice, k, 0)

N.B. segue la filosofia della visita in postordine; il costo computazionale è $\Theta(\text{numero di nodi che si trovano ad un livello } \leq k)$.

Visita per livelli.

E se volessimo accedere ai nodi per livelli, dalla radice in giù?



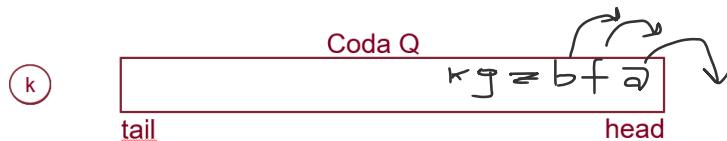
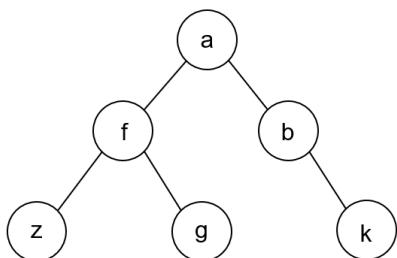
Nessuna delle visite ricorsive che abbiamo illustrato per gli alberi implementati mediante puntatori permette di farlo. È necessario utilizzare una **coda d'appoggio**, nella quale inserire opportunamente i nodi, estraendoli poi per visitarli. La medesima tecnica, con alcuni ulteriori accorgimenti, permetterà (come vedremo più avanti) di realizzare un importante tipo di visita su grafi.

Per semplicità, supponiamo che l'implementazione della coda sia fatta in modo tale da inserire ed estrarre direttamente puntatori a nodi dell'albero. Si lascia come esercizio tale implementazione.

Vogliamo stampare le chiavi dei nodi visitandoli per livelli.

```
def Visita_per_livelli (r; head, tail):
    if (r==None): return
    Enqueue(head, tail, r)
    while (!CodaVuota(head))
        p = Dequeue(head, tail)
        print(p->key)
        if (p->left!=None): Enqueue(head, tail, p->left)
        if (p->right!=None): Enqueue(head, tail, p->right)
    return
```

```
def Visita_per_livelli (r; head, tail):
    if (r==None): return
    Enqueue(head, tail, r)
    while (!CodaVuota(head))
        p = Dequeue(head, tail)
        print (p->key)
        if (p->left!=None)
            Enqueue(head, tail, p->left)
        if (p->right!=None)
            Enqueue(head, tail, p->right)
    return
```



Stampa: a f b z g k

La caratteristica fondamentale di questa implementazione è che vengono inseriti nella coda tutti i nodi di livello i prima di inserire anche un solo nodo di livello $(i + 1)$.

Poiché l'ordine di estrazione è lo stesso (la coda è una struttura FIFO) e il lavoro sul nodo (la stampa) segue immediatamente l'estrazione, il risultato di scandire l'albero per livelli è raggiunto.

Il **costo computazionale** è $\Theta(n)$ perché per ognuno degli n nodi si effettuano:

- una Enqueue, costo $\Theta(1)$;
- una Dequeue, costo $\Theta(1)$;
- un numero costante di operazioni elementari, $\Theta(1)$.

Ad ogni estrazione viene fatta un'estrazione dalla coda perciò n estrazioni.

Esercizio svolto.

Esercizio. Scrivere la visita in pre-ordine di un albero dato tramite vettore dei padri P , supponendo di avere una funzione `Trova_Figli(P, v, sx, dx)` che restituisce in `sx` e `dx` i figli di v (oppure zero se il figlio non c'è).

Soluzione.

```
def Pre_Order_Vett_Padri (P, ind_radice):
    sx = 0; dx = 0;
    #visita qui il nodo corrisp. a ind_radice;
    Trova_Figli(P, ind_radice, sx, dx);
    if sx!=0: Pre_Order_Vett_Padri(P,sx);
    if dx!=0: Pre_Order_Vett_Padri(P,dx);
```

Costo computazionale:

Osserviamo che Trova_Figli(P, v, sx, dx) ha un costo di $\Theta(n)$, indipendentemente da quale sia v , per cui $T(n)$ non decresce spostandosi verso i sottoalberi.

Per questo non è possibile scrivere l'equazione di ricorrenza (o meglio, non è possibile scriverla semplicemente come sappiamo fare).

Procediamo quindi con questo ragionamento:

- per ogni nodo si fa lavoro $\Theta(n)$, da cui $T(n) = \Theta(n^2)$.

Esercizio svolto.

Esercizio:

Dato un albero binario non vuoto memorizzato tramite vettore dei padri, creare uno identico memorizzato tramite notazione posizionale. Calcolare il costo computazionale.

IDEA.

Prima di tutto identifichiamo la radice (unico elemento con None nell'array P - funzione Trovaradice()).

Poi, per ogni nodo sistemato in posizione i , identifichiamo i suoi figli - funzione Trova_figli() e li sistemiamo nelle posizioni $2i$ e $2i+1$ tramite la ricorsione.

Supponiamo che la funzione Trova_figli() restituisca due valori, **left**, **right**, con gli indici di ciascuno dei due figli (oppure None per ogni figlio che non esiste). Si assume per convenzione che il primo indice trovato da Trova_figli() sia quello del figlio sinistro.

Trova_radice(P) effettua una scansione del vettore P dei padri e restituisce l'indice dell'unico elemento a None, che è quello relativo alla radice.

Trova_figli($P; i$) cerca le posizioni dei figli del nodo collocato in posizione i . Effettua una scansione dell'array P da sinistra a destra:

Il primo indice h trovato (se esiste) per cui $P[h] = i$ viene assegnato a left; se non viene trovato si assegna None a left;

Il secondo indice k trovato (se esiste) per cui $P[k] = i$ viene assegnato a right; se non viene trovato si assegna None a right;

SistemaNodo(), che ora vedremo, è ricorsiva:

Copia un nodo dall'array delle chiavi (vettore dei padri, sottovettore B) all'array posizionale (A);

Chiama ricorsivamente sé stessa sui due figli del nodo appena sistemato.

Nella funzione chiamante si avrà:

```
def SistemaNodo (A,B,P;indA,indB) :  
    if (indA < n) :  
        A[indA]=B[indB]  
        {left,right} = Trova_Figli (P,indB)  
        if (left!=None) : SistemaNodo (A,B,P,2*indA-1,left)  
        if (right!=None) : SistemaNodo (A,B,P,2*indA,right)  
    return
```

Array A: array risultato, con l'albero nella notazione posizionale

Array B: chiavi dei nodi nella memorizzazione con vettore dei padri

Array P: padri dei nodi nella memorizzazione con vettore dei padri

indA: indice che si muove sull'array A (memorizzazione posizionale)

indB: indice che si muove sugli array B e P (memorizzazione con vettore dei padri)

P è il vettore dei padri in cui sono memorizzati gli indici dei padri; ind è l'indice del nodo di cui si vogliono trovare i figli.

```

def Trova_Figli(P; ind):
    num_figli= 0
    left=None
    right=None
    for i in range len(P):
        if (P[i]==ind)
            num_figli=num_figli+1
            if (num_figli == 1): left=i
            else: right=i
    return {left,right}

```

Costo computazionale: $\Theta(n)$

N.B. Si può facilmente modificare questa funzione perché funzioni in $O(n)$.

Il costo computazionale è dato dalla somma dei costi di `Trova_radice()` e `SistemaNodo()`.

1. `Trova_radice()` ha banalmente costo $\Theta(n)$
2. `SistemaNodo()` è di fatto una visita, nella quale però il lavoro su ciascun nodo ha costo $\Theta(n)$ perché bisogna sempre scandire l'intero array P per trovare i due figli di un nodo. Poiché la visita fa tale lavoro (sempre sull'intero array P) per ciascun nodo, il suo costo è $T(n) = \Theta(n^2)$

Quindi il costo computazionale è

$$T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Dizionari.

Un **dizionario** è una *struttura dati* che permette di gestire un insieme dinamico di dati, che di norma è un insieme totalmente ordinato, tramite queste tre sole operazioni:

- **insert**: si inserisce un elemento;
- **search**: si ricerca un elemento;
- **delete**: si elimina un elemento.

Fra le strutture dati che abbiamo descritto, le uniche che supportano in modo semplice (anche se non efficiente) tutte queste tre operazioni sono gli array, le liste concatenate e le liste doppie.

Infatti:

- le code (con o senza priorità, inclusi gli heap) e le pile non consentono né la ricerca né l'eliminazione di un elemento arbitrario;
- negli alberi, l'eliminazione di un elemento comporta la disconnessione di una parte dei nodi dall'altra e quindi è un'operazione che in genere richiede delle successive azioni correttive.

Quando l'esigenza è quella di realizzare un dizionario, ossia una struttura dati che rispetti la definizione data sopra, si ricorre quindi a soluzioni specifiche.

Qui ne illustriamo tre:

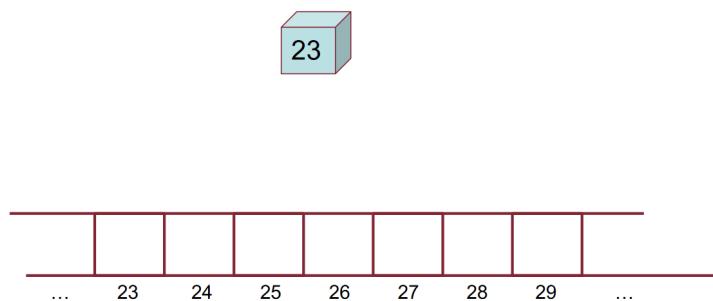
- **tabelle ad indirizzamento diretto** (array);
- **tabelle hash** (array con proprietà particolari);
- **alberi binari di ricerca** (particolari tipi di alberi).

Assunzioni e nomenclatura:

- U : **insieme universo dei valori** che le chiavi possono assumere; U è costituito da valori interi;
- m : **numero delle posizioni** a disposizione nella struttura dati;
- n : **numero degli elementi** da memorizzare nel dizionario; i valori delle chiavi degli elementi da memorizzare sono tutti diversi fra loro.

Tabelle ad indirizzamento diretto.

È un vettore nel quale ogni indice corrisponde al valore della chiave dell'elemento da memorizzare in tale posizione.



Nell'ipotesi $n \leq |U| = m$ un array A di m posizioni assolve perfettamente al compito di dizionario, e per giunta con grande efficienza. Infatti, tutte e tre le operazioni hanno costo computazionale **$\Theta(1)$** :

```
Funzione Insert_Indirizz_Diretto (A; k: chiave da ins.)
    A[k] ← dati dell'elemento di chiave k
    return

Funzione Search_Indirizz_Diretto (A; k: chiave da cerc.)
    return(dati dell'elemento di A con indice k)

Funzione Delete_Indirizz_Diretto (A; k: chiave da canc.)
    A[k] ← None
    return
```

Purtroppo, le cose non sono così semplici nel caso dei problemi reali, poiché:

- l'insieme U può essere enorme, tanto grande da rendere impraticabile l'allocazione in memoria di un array A di sufficiente capienza;
- il numero delle chiavi effettivamente utilizzate può essere molto più piccolo di $|U|$: in tal caso vi è un rilevante spreco di memoria, in quanto la maggioranza delle posizioni dell'array A resta inutilizzata...

Perciò, si ricorre spesso a differenti implementazioni dei dizionari, a meno che non ci si trovi nelle condizioni che permettono l'uso dell'indirizzamento diretto.

Ad esempio, si pensi al caso dei *Codici Fiscali*.

Un CF è costituito da 8 lettere (più una lettera di controllo) e 7 cifre. Considerando un alfabeto di 26 lettere si hanno:

$$26^8 * 10^7 \approx 2 * 10^{18}$$

Mentre ci sono circa $6 * 10^7$ cittadini. Anche considerando che non tutte le lettere e le cifre vengono usate in ogni posizione, la sproporzione è enorme.

Perciò, si ricorre spesso a differenti implementazioni dei dizionari, a meno che non ci si trovi nelle condizioni che permettono l'uso dell'indirizzamento diretto.

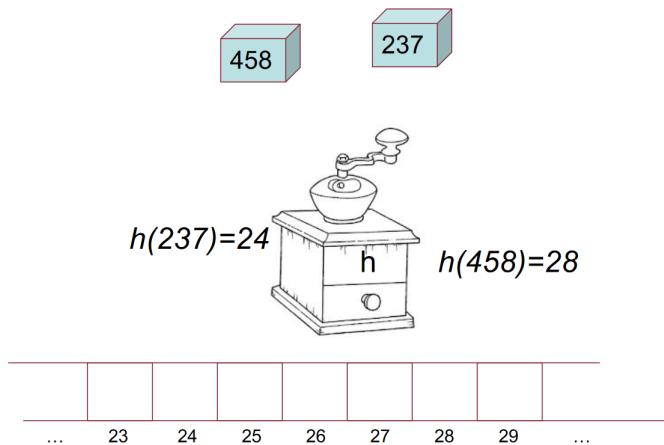
Tabelle hash.

Vi si ricorre quando l'insieme U dei valori che le chiavi possono assumere è molto grande e l'insieme K delle chiavi da memorizzare effettivamente è invece molto più piccolo di U.

Idea: utilizzare di nuovo un array di m posizioni.

Problema: non è possibile mettere direttamente in relazione la chiave con l'indice corrispondente, poiché le possibili chiavi sono molte di più rispetto agli indici.

Soluzione: Si definisce una opportuna funzione h, detta *funzione hash*, che viene utilizzata per calcolare la posizione di un elemento sulla base del valore della sua chiave.



Problema: anche se le chiavi da memorizzare sono meno di m, non si può escludere che due chiavi $k_1 \neq k_2$ siano tali per cui $h(k_1) = h(k_2)$, ossia che la funzione hash restituisca lo stesso valore per entrambe le chiavi, che quindi andrebbero memorizzate nella stessa posizione della tabella.

Tale situazione viene chiamata **collisione**.

NON c'è modo di evitare le collisioni; possiamo solo evitarle il più possibile e, altrimenti, risolverle.

Una buona funzione hash deve essere tale da rendere il più possibile equiprobabile il valore risultante dall'applicazione della funzione: tutti i valori fra 0 ed $(m - 1)$ dovrebbero essere prodotti con uguale probabilità. In altre parole, la funzione dovrebbe far apparire come "casuale" il valore risultante, disregando qualunque regolarità della chiave. Inoltre, la funzione deve essere deterministica: se applicata più volte alla stessa chiave deve fornire sempre lo stesso risultato.

La situazione ideale è quella in cui ciascuna delle m posizioni della tabella è scelta deterministicamente con la stessa probabilità: ipotesi di uniformità semplice della funzione hash.

Sorvoliamo su come ottenere una funzione hash con l'ipotesi di uniformità semplice...

L'ipotesi di uniformità semplice minimizza il numero di collisioni ma queste possono avvenire...

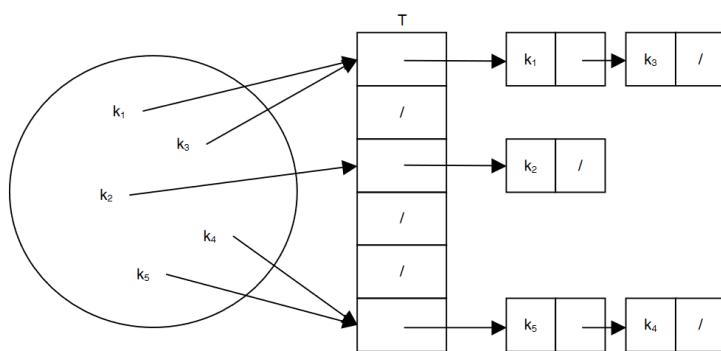
Infatti, per quanto bene sia progettata la funzione hash, è impossibile evitare del tutto le collisioni perché se $|U| > m$ è inevitabile che esistano chiavi diverse che producono una collisione.

Dobbiamo, quindi, risolverle.

Liste di trabocco.

Questa tecnica prevede di inserire tutti gli elementi le cui chiavi mappano nella stessa posizione in una lista concatenata, detta **lista di trabocco**.

La cella della tabella, invece di contenere dati, contiene un puntatore, che è la testa della lista concatenata. Se ho un solo oggetto, la lista ha un solo elemento, altrimenti la lista sarà costituita da più elementi.



Operazioni elementari:

```
Funz. Insert_Liste_Trabocco (A; k: chiave da ins.)  
    inserisci i dati dell'elemento di chiave k  
    nella lista puntata da A[h(k)]  
    return
```

Vado ad inserire in testa.

Costo computazionale: $\Theta(1)$, anche nel caso peggiore, con l'inserzione in testa alla lista.

```
Funz. Search_ListeTrabocco (A; k: chiave da cercare)  
    ricerca k nella lista puntata da A[h(k)]  
    if k è presente  
        then: return puntatore all'elemento contenente k  
    else: return None
```

Costo computazionale: $O(\text{lunghezza della lista puntata da } A[h(k)])$ il che, nel caso peggiore, diviene $O(n)$ quando tutti gli n elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel caso medio (con l'h.p di uniformità semplice) è $O(n/m)$.

$n/m = \alpha$ viene detto **fattore di carico** della tabella e ci indica quanto è piena la tabella.

```
Funz. Delete_ListeTrabocco (A; k: chiave da canc.)  
    cancella i dati dell'elem. di chiave dalla lista  
    puntata da A[h(k)]  
    return
```

Costo computazionale: dipende dall'implementazione delle liste di trabocco e valgono, pertanto, tutte le osservazioni fatte per il costo dell'operazione di cancellazione nelle liste concatenate.

Indirizzamento aperto.

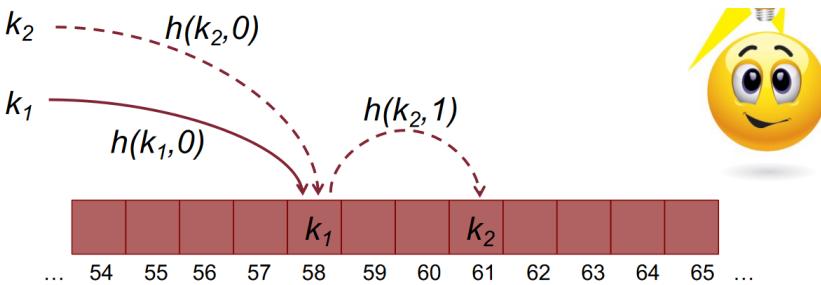
Questa tecnica prevede di inserire tutti gli elementi direttamente nella tabella, senza far uso di strutture dati aggiuntive.

Essa è applicabile quando:

- m è maggiore o uguale al numero n di elementi da memorizzare (quindi il fattore di carico non è mai maggiore di 1);
- $|U| >> m$ (numero di celle a disposizione della tabella).

Idea: invece di avere una lista concatenata per ogni posizione dell'array, inseriamo solo all'interno dell'array stesso, calcolando (nei modi che vedremo fra breve) la sequenza delle posizioni da esaminare.

La funzione hash dipende ora da 2 parametri: la chiave k e il numero di collisioni già trovate.



Inserisco il primo oggetto, se trovo una collisione (già c'è un valore in quella posizione), reitero il procedimento, tramite la stessa funzione, ma cambiando il parametro relativo all'iterazione.

Inserimento.

Se la posizione iniziale relativa alla chiave k è occupata, si scandisce la tabella fino a trovare una posizione libera nella quale l'elemento con chiave k può essere memorizzato.

La scansione è guidata da una sequenza di funzioni hash ben determinata: $h(k, 0)$, $h(k, 1)$, ..., $h(k, m - 1)$.

Costo computazionale: $O(n)$ (lunghezza della sequenza che è necessario scandire) il che, nel **caso peggiore**, diviene $O(n)$ quando tutti gli n elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel **caso medio** (in caso di uniformità semplice) è $1/(1-\alpha)$ dove $\alpha = n/m \leq 1$ è il fattore di carico della tabella.

Ricerca.

Si scandisce la tabella mediante la stessa sequenza di funzioni hash utilizzata per l'inserimento fino a quando si incontra l'elemento cercato oppure una casella vuota, deducendo che l'elemento non è presente.

Costo computazionale nel caso di **ricerca senza successo**:

Come per l'inserimento: nel caso peggiore $O(n)$, ma nel caso medio (quando la funzione hash gode di uniformità semplice) $1/(1-\alpha)$.

dove $\alpha = n/m$ è il fattore di carico della tabella.

Costo computazionale nel caso di **ricerca con successo**: nell'ipotesi di uniformità semplice, il numero di accessi atteso per una ricerca con successo è:

$$\frac{1}{\alpha} \log_e \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

dove $\alpha = n/m$ è il **fattore di carico**.

Esempio: con una tabella piena al 50% il numero di accessi atteso è meno di 3,387; con una tabella piena al 90% il numero di accessi atteso è meno di 3,67; tutto questo indipendentemente dalle dimensioni della tabella!

Cancellazione.

Questa operazione è piuttosto critica. Infatti:

- se si lascia la casella vuota, ciò impedisce di recuperare qualunque elemento sia stato memorizzato in caselle successive nella sequenza di funzioni hash (una ricerca è senza successo quando si incontra una casella vuota...).
- se si marca con un apposito valore deleted la casella da cui è stato cancellato l'elemento, il costo computazionale della ricerca non dipende più esclusivamente dal fattore di carico poiché è influenzata anche dal numero delle posizioni precedentemente occupate da elementi e successivamente marcate.

Per queste ragioni di solito la **cancellazione non è supportata con l'indirizzamento aperto**.

Hashing doppio.

Non è facile garantire l'uniformità semplice.

Una tecnica che, pur non realizzandola, nella pratica ci si avvicina molto è l'hashing doppio.

IDEA: Usare due diverse funzioni hash, una per determinare l'accesso iniziale alla tabella e l'altra per determinare il passo di scansione:

$$h(k, i) = [h_1(k) + i \cdot h_2(k)] \bmod m, \quad i = 0, 1, \dots, m - 1$$

La bontà di questo metodo risiede nel fatto che, se le due funzioni sono ben progettate, è estremamente improbabile che due chiavi $k_1 \neq k_2$ producano una collisione su entrambe le funzioni hash (nel qual caso le due chiavi scandirebbero l'intera tabella nello stesso modo, situazione indesiderabile).

Alberi binari di ricerca.

Sono basati sulla struttura dati albero.

Questi riescono a garantire una ricerca che sia il costo dell'altezza.

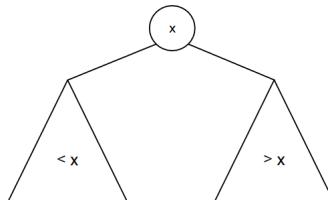
Mentre negli alberi normali, parto dalla radice e vado sia a dx che a sx perché non so dov'è il mio elemento.

Negli alberi binari di ricerca, entrando all'interno dell'albero, non ho bisogno di perlustrare sia la parte dx o sx, ma ci sarà un modo per essere indirizzati a sx o dx.

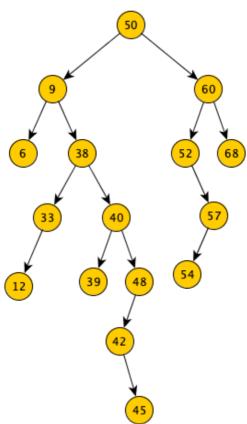
ABR.

Un **albero binario di ricerca (ABR)** è un albero nel quale vengono mantenute le seguenti proprietà:

- ogni nodo contiene una chiave;
- il valore della chiave contenuta in ogni nodo è maggiore della chiave contenuta in ciascun nodo nel suo sottoalbero sinistro (se esiste);
- il valore della chiave contenuta in ogni nodo è minore della chiave contenuta in ciascun nodo del suo sottoalbero destro (se esiste);



Un esempio:



Gli ABR sono strutture dati che supportano tutte le operazioni già definite in relazione agli insiemi dinamici più alcuni altre.

Operazioni di interrogazione:

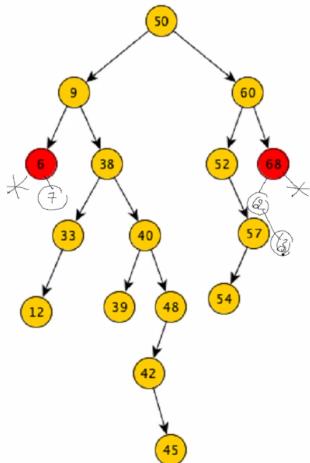
- **Search(T, k)**: restituisce un puntatore all'elemento con chiave di valore k in T se questo è presente, None altrimenti;
- **Minimum(T)/Maximum(T)**: restituisce un puntatore all'elemento in T con minima/massima chiave;
- **Predecessor(T, p)/Successor(T, p)**: restituisce un puntatore all'elemento in T con la chiave che precederebbe/seguirebbe in una sequenza ordinata la chiave contenuta nel nodo puntato da p .

Operazioni di manipolazione:

- **Insert**(T, k): inserisce un elemento di chiave k in T;
- **Delete**(T, p): elimina da T l'elemento puntato da p.

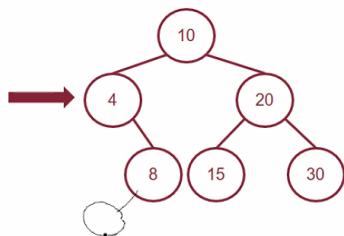
Un **ABR** può essere usato sia come dizionario che come coda di priorità: il **minimo** è sempre nel **nodo più a sinistra** (che non ha un figlio a sx, ma ne può avere uno a dx), il **massimo** in **quello più a destra** (che può avere figlio sx, ma non può avere figlio dx).

Si noti che il nodo più a sinistra non necessariamente è una foglia: può anche essere il nodo più a sinistra che non ha un figlio sinistro; analoga considerazione per il nodo più a destra.



Per elencare tutte le chiavi in ordine crescente basta compiere una visita inordine.

6 9 12 33 38 39 40 42 45 48 50 52 54 57 60 68



Nell'elemento disegnato non possiamo inserire 3 perché dato che 8 è figlio di 4, ci possono essere solo elementi più grandi di 4.

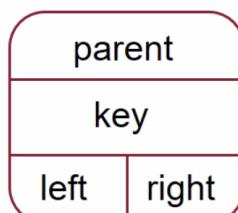
Dunque, un ABR può anche essere visto come una struttura dati su cui eseguire un **algoritmo di ordinamento**, costituito di due fasi:

1. inserimento di tutte le n chiavi da ordinare in un ABR, inizialmente vuoto;
2. visita inordine dell'ABR appena costruito.

Il costo computazionale di tale algoritmo sarà: $T(\text{costruzione ABR}) + T(\text{visita}) = T(\text{costruzione ABR}) + \Theta(n)$

Più avanti determineremo il tempo necessario alla costruzione di un ABR.

Nel seguito assumiamo che l'ABR sia memorizzato tramite record a puntatori e che ciascun nodo abbia, oltre ai soliti puntatori ai figli dx e sx, anche un puntatore al padre. (utile per risalire agli antenati)



Ricerca in un albero binario di ricerca.

Il procedimento di ricerca su un ABR è concettualmente simile alla ricerca binaria ed è molto semplice: si esegue una discesa dalla radice che viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

Quando si trova in un nodo il valore ricercato si termina con successo restituendo il puntatore al nodo. Se invece si arriva ad una foglia che non lo contiene si termina senza successo restituendo NULL. Si osservi che ogni qual volta si scende verso destra (sinistra), automaticamente si decide di escludere l'intero sottoalbero di sinistra (destra), e quindi lo spazio di ricerca si riduce via via che si discende verso le foglie.

```
Funzione ABR_search_ricorsiva (p: puntatore all'albero; k: chiave)
    if ((p = NULL) or (key[p] = k))
        return p
    if (k < key[p])
        return ABR_search (left[p], k)
    else
        return ABR_search (right[p], k)

def ABR_searchRic(p,k):
    if (p == None) or (p->key = k):
        return p
    if (k < p->key):
        return ABR_searchRic(p->left, k)
    else:
        return ABR_searchRic(p->right, k)
```

Per quanto riguarda il costo computazionale della funzione di ricerca, intuitivamente, dato che la ricerca percorre un singolo cammino dalla radice ad una foglia, ci aspettiamo che esso sia legato all'altezza h dell'albero.

Più precisamente, la funzione ricorsiva compie un numero costante di operazioni sulla radice dell'albero e poi richiama sé stessa sulla radice di uno dei due sottoalberi, che al massimo ha altezza $h - 1$; possiamo quindi scrivere:

$$T(h) = \Theta(1) + T(h - 1) = \Theta(h)$$

nel caso peggiore, che è quello della ricerca senza successo che termina in una foglia a distanza massima dalla radice.

Nel caso base, che si ha quando $h=0$ (caso in cui la chiave cercata sia nella radice), il tempo di esecuzione è costante, cioè $T(0) = \Theta(1)$.

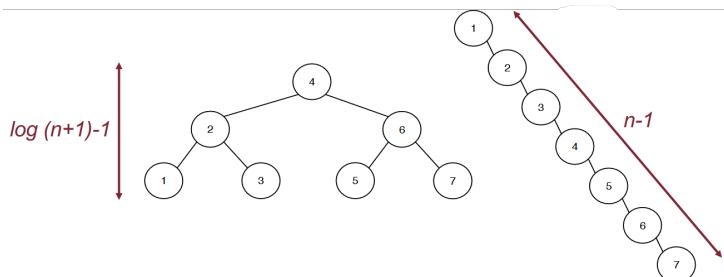
La ricerca su un ABR può essere espressa in modo semplice anche in forma iterativa:

```
Funzione ABR_search_iterativa (p: puntatore all'albero; k: chiave)
    while ((p ≠ NULL) and (key[p] ≠ k))
        if (k < key[p])
            p ← left[p]
        else
            p ← right[p]
    return p
```

In questo caso, il costo computazionale dipende dal numero di volte che viene eseguito il ciclo while. Nel caso peggiore (ricerca senza successo che termina in una foglia a distanza massima dalla radice) esso viene eseguito un numero di volte pari all'altezza dell'albero e quindi il costo è, ovviamente come per la versione ricorsiva, $O(h)$.

Considerando che la ricerca su un ABR ricorda molto da vicino la ricerca binaria già descritta nel par. 3.2, per la quale il costo computazionale è $O(\log n)$, sorge spontanea una domanda: come mai non riusciamo a garantire un costo logaritmico anche per la ricerca su un ABR?

La ragione risiede nel fatto che l'ABR non include fra le sue proprietà alcunché in relazione alla sua altezza. Ad esempio, entrambi gli ABR illustrati nella figura seguente sono legittimi (permessi) :



Chiaramente, nell'ABR di sinistra l'altezza è $\Theta(\log n)$ mentre in quello di destra (detto albero degenere) è $\Theta(n)$; la stessa differenza fra tali due situazioni quindi si manifesta nel costo della ricerca nel caso pessimo.

Questo ci fa capire che, se vogliamo garantire che il costo computazionale della ricerca su ABR sia limitata superiormente da un logaritmo, dovremo preoccuparci di introdurre qualche tecnica che ci permetta di tenere sotto controllo la crescita dell'altezza: questo tipo di tecniche prendono il nome di tecniche di **bilanciamento in altezza** e saranno descritte nel seguito.

Analogamente al Quicksort, è possibile dimostrare che il comportamento degli ABR nel caso medio è molto più vicino al caso migliore del caso peggiore:

definendo un albero binario di ricerca costruito in modo casuale con n chiavi come quello che si ottiene inserendo in ordine casuale le chiavi in un albero inizialmente vuoto, dove ciascuna delle $n!$ permutazioni delle chiavi di input ha la stessa probabilità, vale:

Teorema. L'altezza attesa di un albero binario di ricerca costruito in modo casuale con n chiavi tutte distinte è $O(\log n)$.

Di conseguenza, una strategia che, in media, costruisce un albero bilanciato per un insieme fisso di elementi consiste nel permutare in modo casuale gli elementi e poi nell'inserire gli elementi in quell'ordine nell'albero.

Questa tecnica non può essere usata se non abbiamo tutti gli elementi contemporaneamente, perché ad esempio (come spesso accade) riceviamo gli elementi uno alla volta...

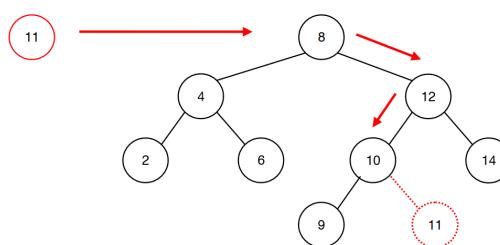
Quindi: se vogliamo garantire che il costo computazionale della ricerca su ABR sia limitata superiormente da un logaritmo, dovremo preoccuparci di mettere in campo qualche tecnica che ci permetta di tenere sotto controllo la crescita dell'altezza: bilanciamento in altezza (di cui ripareremo).

Inserimento in un albero binario di ricerca.

Anche il procedimento di inserimento su un ABR è molto semplice: si esegue una discesa che, come nel caso della ricerca, viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

Quando si arriva al punto di voler proseguire la discesa verso un puntatore vuoto (NULL) allora, in quella posizione, si aggiunge un nuovo nodo contenente la chiave da inserire.

Il padre di tale nuovo nodo potrebbe essere una foglia (entrambi i suoi figli sono NULL), ma, più in generale, è un nodo a cui manca il figlio corrispondente alla direzione presa.



Per convenienza introduciamo una modifica alla struttura dati utilizzata per rappresentare un nodo che, sebbene non strettamente necessaria per l'inserzione, lo sarà per poter gestire l'eliminazione di un nodo.

In ogni nodo, oltre ai due puntatori ai figli sinistro e destro, aggiungiamo un campo in cui viene memorizzato il puntatore al padre (NULL nella radice dell'albero). Chiameremo parent tale campo.

Supponiamo, come già fatto in altri casi analoghi, che sia disponibile un nodo già costruito contenente nel campo key il valore della chiave e nei campi parent, left e right il valore NULL.

Così come nel caso della ricerca, anche qui il costo computazionale dipende dal numero di volte in cui viene eseguito il ciclo while. Esso viene eseguito un numero di volte che è al massimo pari all'altezza dell'albero e quindi il costo è $O(h)$.

Lo pseudocodice (iterativo) è il seguente:

```

def ABR_insert (p,k):
    y,x=None,p
    z=NodoABR(k)
    while (x != None)           #y punta sempre al padre di x
        y = x                   #discesa alla prima pos. disponib.
        if z->key < x->key:
            x = x->left
        else:
            x = x->right
        if (y==None)             #se albero inizialmente vuoto
            p = z
        else:
            if (z->key < y->key):
                y->left = z
            else:
                y->right = z
            z->parent = y
    return p
  
```

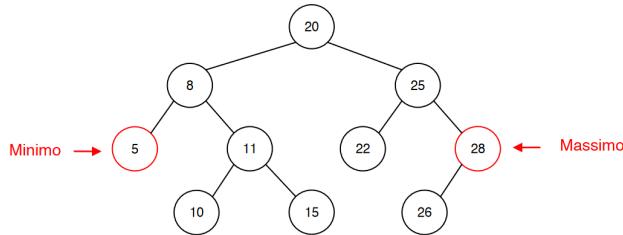
ciclo eseguito al max h volte quindi $T(h)=O(h)$

#collegam. padre - nodo da inser.
#p potrebbe essere cambiato

Ricerca di minimo, massimo, predecessore e successore.

Consideriamo il problema di reperire il minimo e il massimo dei valori contenuti in un ABR. La soluzione è semplice: poiché, come già detto, il **minimo** | **massimo** si trova nel nodo più a sinistra, per trovarlo si scende sempre a **sinistra** | **destra** a partire dalla radice. Ci si ferma quando si arriva a un nodo che non ha figlio sinistro (destro): quel nodo contiene il minimo (massimo).

Si lascia lo pseudocodice delle due funzioni come esercizio.

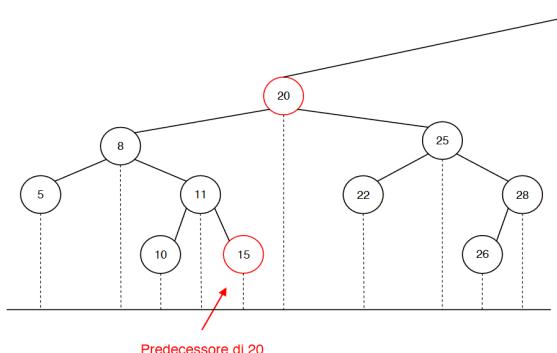


Entrambe queste operazioni richiedono di scendere dalla radice lungo un singolo cammino, e quindi hanno costo computazionale limitato superiormente dall'altezza dell'albero: $O(h)$.

Più interessante è il problema di determinare il predecessore o il successore di una chiave k contenuta nell'ABR.

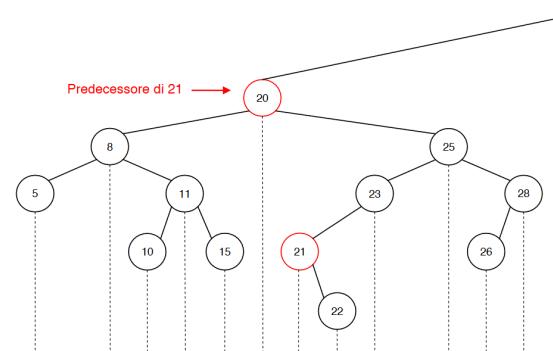
Ricordiamo che per :

- **predecessore** si intende il nodo dell'albero contenente la chiave che precederebbe immediatamente k se le chiavi fossero ordinate;
 - il **successore** è il nodo che contiene la chiave che seguirebbe immediatamente k se le chiavi fossero ordinate; pertanto, la ricerca del predecessore (successore) non ha nulla a che fare con relazioni padre-figlio sull'albero.
- Concentriamoci dapprima sulla ricerca del predecessore.



Caso 1: se la chiave k è contenuta in un nodo che ha un sottoalbero sinistro (ad es. $k=20$ nella figura della pagina precedente), allora il predecessore di k è il massimo delle chiavi contenute nel sottoalbero sinistro, e quindi il nodo che la contiene è quello più a destra del sottoalbero sinistro.

Caso 2: viceversa, se il nodo che contiene k non ha sottoalbero sinistro (ad es. $k=21$ nella figura che segue) vuol dire che esso è il nodo più a sinistra di un certo sottoalbero, e quindi il minimo di tale sottoalbero. Per trovare il predecessore di k bisogna quindi risalire alla radice di quel sottoalbero, il che significa salire a destra finché è possibile. Una volta giunti nella radice del sottoalbero, si risale (con un singolo passo di salita a sinistra) a suo padre che è il predecessore di k . Si noti che è sempre possibile fare tale ultimo passo, a meno che k non sia il minimo.



Una situazione perfettamente simmetrica esiste per il problema di trovare il successore di un nodo. Entrambe tali operazioni richiedono una discesa lungo un singolo cammino a partire dalla radice oppure una singola risalita verso la radice. Per entrambe le funzioni il costo computazionale è limitato superiormente dall'altezza dell'albero: $O(h)$.

Come si fa a sapere se il passo di risalita dal nodo x al padre di x avviene salendo verso destra o verso sinistra?

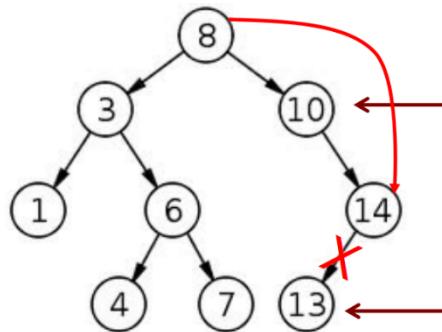
Bisogna controllarlo esplicitamente con un test, ad ogni passo, prima di salire:

- if ($x = x->parent->left$) allora la risalita sarà verso destra;
- if ($x = x->parent->right$) allora la risalita sarà verso sinistra.

Cancellazione:

Per eliminare un nodo in un ABR:

- **Caso 1:** se il nodo è una foglia lo si elimina, ponendo a None l'opportuno campo nel nodo padre;
- **Caso 2:** se il nodo ha un solo figlio lo si "cortocircuita", cioè si collegano direttamente fra loro suo padre e il suo unico figlio, indipendentemente che sia destro o sinistro;



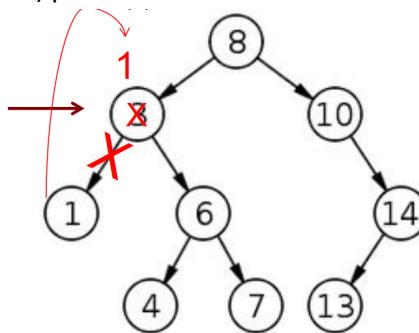
Problema: Se la chiave da eliminare è contenuta in un nodo che ha entrambi i figli è necessario riaggiustare l'albero dopo l'eliminazione per evitare che si disconnetta, il che produrrebbe di conseguenza due ABR separati.

Riaggiustare l'albero significa trovare un nodo da collocare al posto del nodo che va eliminato, così da mantenere l'albero connesso e da garantire il mantenimento della proprietà fondamentale degli ABR.

Tale nodo può quindi essere solamente il predecessore o il successore del nodo da eliminare.

Caso 3: se il nodo ha entrambi i figli lo si sostituisce col predecessore (o col successore), che va quindi tolto (ossia eliminato) dalla sua posizione originale (operazione che ricade in uno dei due casi precedenti).

N.B. Per trovare il predecessore (o il successore) del nodo da cancellare siamo sicuramente nel caso 1 dell'algoritmo per la ricerca del predecessore (o successore) perché il nodo da cancellare ha due figli.



Esercizio. Progettare un algoritmo che, dati due ABR T_1 e T_2 , rispettivamente con n_1 e n_2 nodi, ed altezza h_1 ed h_2 , dia in output un ABR che contiene tutti gli n_1+n_2 nodi. Fare le opportune osservazioni sul costo computazionale e sull'altezza dell'ABR risultante, come funzione di h_1 e h_2 .

Soluzione. Inseriamo nell'albero con il maggior numero di nodi (senza perdere di generalità sia esso T_1) i nodi dell'altro albero uno ad uno.

Sapendo che l'operazione di inserimento in un ABR ha un costo dell'ordine dell'altezza dell'albero si ha, nel caso peggiore, che il costo computazionale è:

$$O(h_1) + O(h_1+1) + O(h_1+2) + \dots + O(h_1 + n_2 - 1) = n_2 O(h_1) + O(1 + 2 + \dots + n_2 - 1)$$

Poiché nel caso peggiore $O(h_1) = O(n_1)$ ed $O(1 + 2 + \dots + n_2) = O(n_2^2)$, si ottiene che il costo di questo approccio è $O(n_1 n_2) + O(n_2^2) = O(n_1 n_2)$, essendo per ipotesi $n_1 > n_2$.

Osservazione: questo procedimento è corretto perché un albero binario di ricerca non è necessariamente bilanciato, e quindi poco importa che l'altezza dell'albero risultante possa diventare $O(h_1 + n_2)$.

Esercizio svolto (1)

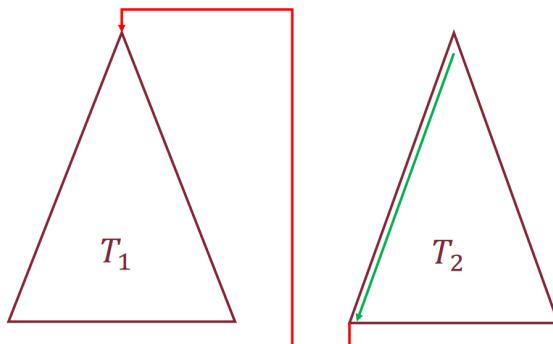
Esercizio. Progettare un algoritmo che, dati due ABR T_1 e T_2 , rispettivamente con n_1 ed n_2 nodi, ed altezza h_1 ed h_2 , dia in output un ABR che contiene tutti gli n_1+n_2 nodi, assumendo che tutti gli elementi in T_1 siano minori di quelli in T_2 .

Soluzione. Oltre alla soluzione dell'esercizio precedente, proponiamo un altro approccio per tentare di sfruttare l'ipotesi...

“Appendiamo” l'albero T_1 come figlio sinistro del minimo di T_2 . Questo si può sempre fare facilmente perché il minimo di un ABR è il nodo più a sx che non possiede figlio sx, pertanto è sufficiente:

- settare un puntatore sulla radice di T_2 ,
- scendere verso il figlio sx finché esso esista
- giunti al nodo che non ha figlio sx (che è il minimo dell'albero), agganciarlo a sx la radice di T_1 .

Il costo è dominato dal costo della ricerca del minimo, cioè da $O(h_2)$.



Esercizio. Scrivere lo pseudocodice dell'inserimento di una nuova chiave z in un ABR memorizzato mediante la notazione posizionale.

Assunzioni:

- L'array contiene n posizioni (indici da 0 a $n-1$);
- Nell'array si usa il simbolo '-' per denotare un nodo dell'albero mancante.

```
def ABR_insert (A, z):
    n=len(A)
    x=0
    while (x < n) AND (A[x] != '-'):      #discesa
        if z < A[x]:
            x = 2*x-1
        else:
            x = 2*x
    if (x < n)
        A[x] = z
    return
```

Costo computazionale: $O(h)$. Nota: $h = \Omega(\log n)$ e $h = O(n)$

Alberi rosso-neri.

Bilanciamento dell'altezza.

Un ABR di altezza h contenente n nodi supporta le operazioni fondamentali dei dizionari in tempo $O(h)$, ma purtroppo non si può escludere che h sia $O(n)$, con conseguente degrado delle prestazioni.

Viceversa, tutte le operazioni restano efficienti se si riesce a garantire che l'altezza dell'albero sia piccola, in particolare sia $O(\log n)$.

Per raggiungere tale scopo (altezza logaritmica) esistono varie tecniche, dette di **bilanciamento**. Il modo per garantire un'altezza logaritmica..

Le tecniche di bilanciamento sono tutte basate sull'idea di riorganizzare la struttura dell'albero se essa, a seguito di un'operazione di inserimento o di eliminazione di un nodo, viola determinati requisiti.

In particolare, il requisito da controllare è che, per ciascun nodo dell'albero, l'altezza dei suoi due sottoalberi non sia "troppo differente".

Ciò che rende non banali queste tecniche è che si vuole aggiungere agli ABR una proprietà (il bilanciamento) senza peggiorare il costo computazionale delle operazioni.

In letteratura vi sono vari approcci. Noi ne descriveremo uno.

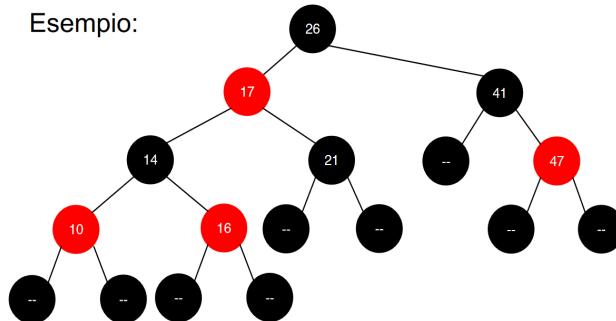
Un **albero rosso-nero (RB-albero)** è un ABR i cui nodi hanno un campo aggiuntivo, il **colore**, che può essere solo **rosso** o nero.

Alla struttura si aggiungono foglie **fittizie** (che non contengono chiavi) per fare in modo che tutti i nodi "veri" dell'albero abbiano esattamente due figli.

Un RB-albero è un ABR che soddisfa le seguenti **proprietà aggiuntive**:

1. ciascun nodo è rosso o nero;
2. ciascuna foglia fittizia è nera;
3. se un nodo è rosso (nodo interno) i suoi figli sono entrambi neri;
4. ogni cammino da un nodo a ciascuna delle foglie del suo sottoalbero contiene lo stesso numero di nodi neri.
5. La radice è sempre nera.

Esempio:



Nota: per non sprecare spazio in memoria. l'insieme delle foglie fittizie può essere sostituito da un unico oggetto, cui puntano tutti i puntatori che indicano una foglia fittizia (sentinella).

Proprietà: nessun cammino dalla radice ad una foglia può essere lungo più del doppio di un cammino dalla radice ad una qualunque altra foglia.

Dim.

- il numero di nodi neri è il medesimo lungo tutti i cammini dalla radice ad una qualunque foglia (proprietà 4);
- il numero di nodi rossi lungo un cammino non può essere maggiore del numero di nodi neri lungo lo stesso cammino (proprietà 3);
- un cammino che contiene il massimo numero possibile di nodi rossi non può essere lungo più del doppio di un cammino composto solo di nodi neri.

La b-altezza di un nodo x (black height di x , $bh(x)$) è il numero di nodi neri sui cammini dal nodo x (non incluso) alle foglie sue discendenti (è uguale per tutti i cammini, proprietà 4).

La b-altezza di un RB-albero è la b-altezza della sua radice.

NOTA: se un albero rispetta le proprietà 1-4 ma ha la radice rossa (e quindi non rispetta la proprietà 5), basta

cambiare il colore della radice in nero per ottenere un RB-albero valido. La sua b-altezza risulterà aumentata di 1.

Lemma

Il sottoalbero radicato in un qualsiasi nodo x contiene almeno $2bh(x)-1$ nodi interni.

Dimostrazione

Ragioniamo per induzione sulla distanza di x dalle foglie, sia essa $d(x)$.

Se $d(x) = 0$, allora x è una foglia per cui $bh(x) = 0$.

Quindi il suo sottoalbero contiene almeno $2bh(x) - 1 = 20 - 1 = 1 - 1 = 0$ nodi interni.