Algoritmica

Ahmad Shatti

2019 - 2020

Indice

Chapter 1

Algoritmi e notazione asintotica

1.1 Algoritmi

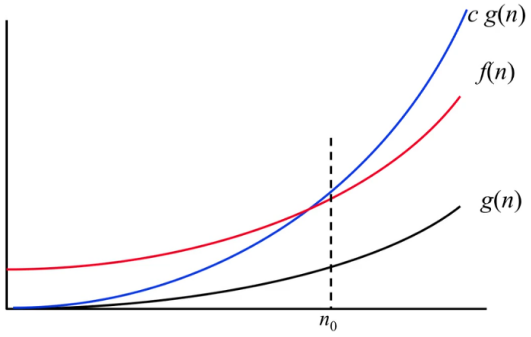
Un **algoritmo** è una procedura di calcolo definita che prende un certo valore, o un insieme di valori, come input e genera un valore, o un insieme di valori, come output. Un algoritmo quindi è una sequenza di passi computazionali che trasforma l'input in output. Per esempio, data la sequenza di input $\{31, 41, 59, 26, 41, 58\}$, un algoritmo di ordinamento restituisce come output la sequenza $\{26, 31, 41, 41, 58, 59\}$. Tale sequenza di input è detta **istanza** del problema di ordinamento. La scelta dell'algoritmo più appropriato dipende dal numero di elementi da ordinare, da eventuali vincoli sui valori degli elementi, dall'architettura dei calcolatori e dal tipo di unità di memorizzazione (memoria principale, dischi, ...). Si può valutare un algoritmo da tanti punti di vista:

- **Correttezza**, dimostrazione formale.
- **Utilizzo delle risorse**, cioè quale tipo di risorse vengono utilizzate e in quale quantità. Tipicamente le risorse che vengono contate sono *il tempo di esecuzione* e l'*utilizzo della memoria*. Il tempo di esecuzione dipende da tanti fattori come *hardware, compilatore, input, ...* quindi è inutile cronometrare il tempo del programma ma l'algoritmo progettato deve essere efficiente indipendentemente dal calcolatore. Si cerca di astrarre tutti quei problemi dell'hardware utilizzando un modello computazionale denominato **modello RAM** dove si immagina di avere la memoria principale infinita, un singolo processore, un solo programma e si calcola il tempo in unità. Il costo per effettuare una operazioni di lettura, esecuzione di una computazione, una scrittura è 1 unità di tempo.
- **Semplicità**, cioè la facilità con cui è scritto. La facilità aiuta sia chi deve utilizzare l'algoritmo sia chi lo deve mantenere.

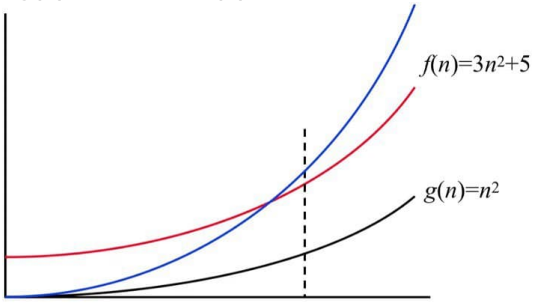
1.2 Notazione asintotica

Si utilizzano le notazioni asintotiche per descrivere i tempi di esecuzione *T(n)* degli algoritmi o a qualche altro aspetto come la quantità di spazio utilizzato. Esse sono definite in termini di funzioni il cui dominio è l'insieme dei numeri naturali $\mathbb{N}$.

- **Limite superiore asintotico**
  se esiste una costante $c > 0$ e un $n_0 > 0$ tale che la funzione $f(n) \leq c\, g(n)$ per tutti gli $n \geq n_0$ allora $g(n)$ è detto limite superiore asintotico di $f(n)$. Si scrive ***f(n) = O(g(n))***



Esempio 1. Determinare se esiste c tale che $f(n) < c\ g(n)$, cioè esiste una costante tale che da un certo punto in poi $g(n)$ domina su $f(n)$



Si prende c = 4 allora $4g(n) = 4n^2$, si deve dimostrare $4n^2 \geq 3n^2 + 5$

$$\begin{aligned} 4n^2 &= 3n^2 + n^2 \\ &\geq 3n^2 + 5 \\ &\text{per ogni } n \geq 3 \end{aligned}$$

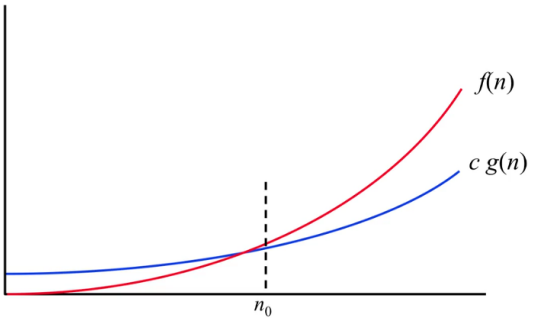
quindi c = 4 e $n_0 = 3$

Esempio 2. Mostrare che $3n^2 + 2n + 5 = O(n^2)$
Come prima si trova una c e un n_0 positivi per cui la c moltiplicata per n^2 domina sempre $f(n) = 3n^2 + 2n + 5$.

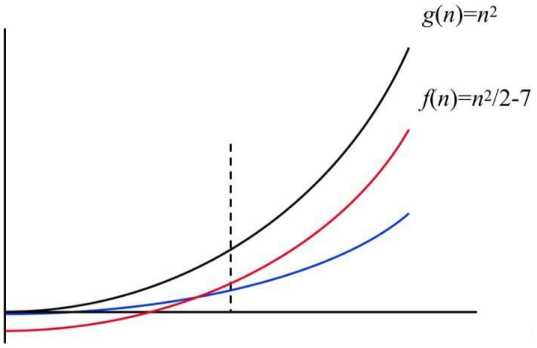
Per c = 10

$$\begin{aligned} 10n^2 &= 3n^2 + 2n^2 + 5n^2 \\ &\geq 3n^2 + 2n + 5 \\ &\text{per ogni } n \geq 1 \end{aligned}$$

- **Limite inferiore asintotico**
se esiste una costante $c > 0$ e un $n_0 > 0$ tale che la funzione $f(n) \geq c\ g(n)$ per tutti gli $n \geq n_0$ allora $g(n)$ è detto limite inferiore asintotico di $f(n)$. Si scrive $\boldsymbol{f(n) = \Omega(g(n))}$



Esempio 3. Determinare se esiste c tale che $f(n) > c\ g(n)$

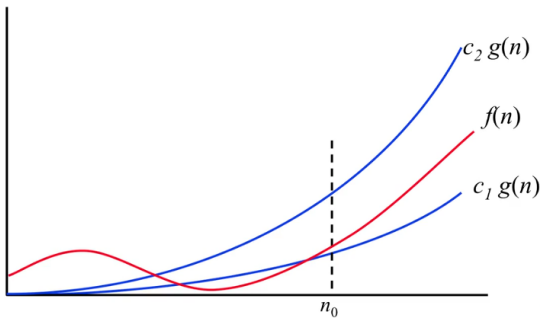


Si prende c = $\frac{1}{4}$ allora $\frac{n^2}{4} \leq \frac{n^2}{2} - 7$

$$\begin{aligned} \frac{n^2}{4} &= \frac{n^2}{2} - \frac{n^2}{4} \\ &\leq \frac{n^2}{2} - 7 \\ &\text{per ogni } n \geq 6 \end{aligned}$$

quindi c = $\frac{1}{4}$ e $n_0 = 6$

- **Limite asintotico stretto**
se esistono c_1, c_2 e n_0 positive tali che $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ *per ogni* $n \geq n_0$ allora $g(n)$ è detto limite asintotico stretto di $f(n)$. Si scrive $\boldsymbol{f(n) = \Theta(g(n))}$



1.2.1 Ulteriori notazioni asintotiche

O e Ω forniscono dei limiti che però non sempre sono significativi, per esempio

$2n = O(n)$	anche	$2n = \Omega(n)$
$2n = O(n^2)$	ma	$2n \neq \Omega(n^2)$

Il limite $2n = O(n^2)$ non è stretto. Esistono notazioni per descrivere limiti non asintoticamente stretti:

- **Limite asintotico superiore non stretto**
  per ogni $c > 0$ esiste un n_0 tale che $f(n) < c\, g(n)$ per ogni $n \geq n_0$. Si scrive ***f(n) = o(g(n))***

  Esempio 4. $2n = o(n^2)$ $o(g(n))$ sta sempre sopra per ogni *c*, mentre, $2n^2 \neq o(n^2)$ sarebbe vero solo per $c \geq 2$. Se il limite è $o(n)$ sicuramente sarà anche $O(n)$

- **Limite asintotico inferiore non stretto**
  per ogni $c > 0$ esiste un n_0 tale che $f(n) > c\, g(n)$ per ogni $n \geq n_0$. Si scrive ***f(n) =*** ω***(g(n))***

  Esempio 5. $\frac{n^2}{2} = \omega(n)$ ma $\frac{n^2}{2} \neq \omega(n^2)$

1.2.2 Teoremi sulla notazione asintotica

- $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$
- Transitività:
  i) Se $f_1 = O(f_2(n))$ e $f_2(n) = O(f_3(n))$ allora $f_1(n) = O(f_3(n))$
  ii) Se $f_1 = \Omega(f_2(n))$ e $f_2(n) = \Omega(f_3(n))$ allora $f_1(n) = \Omega(f_3(n))$
  iii) Se $f_1 = \Theta(f_2(n))$ e $f_2(n) = \Theta(f_3(n))$ allora $f_1(n) = \Theta(f_3(n))$
- Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$ allora $O(f_1(n) + f_2(n)) = O(max\{g_1(n), g_2(n)\})$
- Se $f(n)$ è un polinomio di grado d allora $f(n) = \Theta(n^d)$

1.3 Analisi del caso migliore e peggiore

L'analisi del caso migliore e peggiore dipendono dall'input:

- L'analisi del **caso migliore** è il costo minimo possibile ovvero quanto costa il programma nei migliori dei casi. Si utilizza un Ω-grande cioè il **limite inferiore** del tempo di esecuzione per un qualunque input di dimensione N
- L'analisi del **caso peggiore** è il costo massimo possibile ovvero quanto costa il programma nei peggiori dei casi. Si utilizza un O-grande cioè il **limite superiore** del tempo di esecuzione per un qualunque input di dimensione N

L'analisi del caso medio è in genere molto più difficile e a volte non è ovvio quale sia il valore medio.

1.4 Legge di Moore

La legge di Moore afferma che *le prestazioni dei calcolatori raddoppiano ogni 18 mesi*. Allora conviene concentrarsi da subito sulla **progettazione di algoritmi efficienti** piuttosto che attendere calcolatori più potenti.

Chapter 2

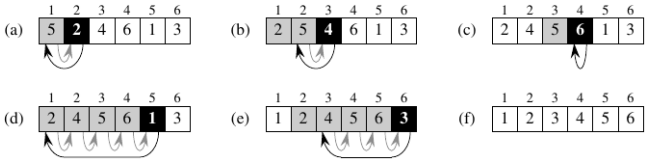
Insertion, selection e merge sort

2.1 Insertion sort

L'Insertion sort è un algoritmo efficiente per ordinare un piccolo numero di elementi. Prende una sequenza di n numeri, per esempio un array, e ordina i numeri **in place**.

Algorithm 1: costo $O(n^2)$

```
1 INSERTION-SORT(a)
2 for j ← 2 to a.lenght do
3     key ← a[j];
4     i ← j − 1;
5     while i > 0 and a[i] > key do
6         a[i + 1] = a[i];
7         i ← i − 1;
8     a[i + 1] = key;
```



dove il rettangolo nero rappresenta *key* che viene confrontato con i rettangoli grigi. Le frecce grigie mostrano i valori dell'array spostati verso destra, le frecce nere indicano dove viene spostata la chiave

Una volta scritto un algoritmo e capito che funziona, si effettua la dimostrazione formale della sua correttezza, cioè che il programma funzioni con qualsiasi input. Quando l'algoritmo presenta un ciclo, vanno dimostrate alcune proprietà, che cambiano per ogni algoritmo, per arrivare alla correttezza.

Invariante di ciclo: all'inizio di ogni iterazione for, gli elementi in *a[1...j-1]* sono gli elementi che erano in *a[1...j-1]*, ma ordinati. In altre parole al ciclo j-esimo gli elementi da 1 fino a *j-1* sono ordinati. Per verificare ciò occorrono dimostrare 3 cose:

1. **Inizializzazione**, invariante vera all'inizio del primo ciclo

   All'inizio del primo ciclo *j* vale 2, allora il sottoarray *a[1...j-1]* è formato dal solo elemento *a[1]* e quindi banalmente il sottoarray è ordinato.

2. **Mantenimento**, se vera prima di un ciclo, allora rimane vera prima della successiva iterazione

   Ogni ciclo for si spostano *a[j-1], a[j-2], ...* di una posizione a destra, fino a trovare il posto giusto per *a[j]*. Il sottoarray a[1...j] quindi è ordinato ed è formato dagli stessi elementi che originariamente erano in *a[1...j]*. Dunque l'incremento di *j* per la successiva iterazione del ciclo for preserva l'invariante di ciclo.

3. **Terminazione**, vera alla fine del ciclo

   Il for termina quando $j > n$ e quindi alla fine del ciclo $j = n + 1$ allora sostituendolo nella formula dell'invariante di ciclo si può concludere che gli elementi in *a[1...n]* sono gli iniziali ma ordinati.

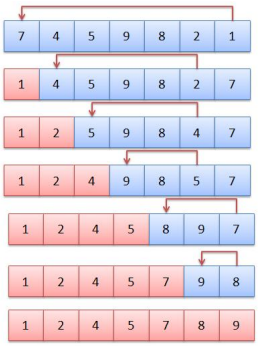
Il caso migliore è quando l'array è già ordinato dove si effettuano n operazioni: *O(n)*. Il caso peggiore è quando l'array è ordinato al contrario e allora occorrono n^2 operazione, quindi il costo è $O(n^2)$

2.2 Selection sort

La strategia nell'Insertion sort era inserire nel punto giusto la chiave, nel Selection sort si seleziona a ogni ciclo il minimo e lo si posiziona nel punto giusto.

Algorithm 2: costo $O(n^2)$

```
1 SELECTION-SORT(a)
2 n ← a.lenght;
3 for j ← 1 to n − 1 do
4     min ← j;
5     for i ← j + 1 to n do
6         if (a[i] < a[min]) then
7             min ← i;
8     swap a[j] with a[min];
```



Il costo nel caso migliore o caso peggiore è sempre $O(n^2)$ perché indipendentemente dall'input si deve effettuare la ricerca del minimo nella porzione che rimane da cercare, anche se l'array è già ordinato. In questo algoritmo quindi il costo non dipende da come è fatto l'input

2.3 Divide et impera

Esistono altri modi iterativi per affrontare il problema dell'ordinamento ma per cambiare strategia occorre la *ricorsione*. Gli algoritmi ricorsivi adottano un approccio **divide et impera**:

- **Divide**, il problema viene diviso in un certo numero di sottoproblemi, che sono istanze più piccole dello stesso problema
- **Impera**, i sottoproblemi vengono risolti in modo ricorsivo. Quando i sottoproblemi hanno una dimensione sufficientemente piccola, essi vengono risolti direttamente
- **Combina**, le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema generale

Esempio 1. Dato un array di n elementi, calcolare il min e il max. Lo pseudocodice può essere:

Algorithm 3: costo $O(n)$

```
1  MAX-MIN(a, sx, dx)
2  if (sx = dx) then                                   // Caso base 1
3      return (a[sx], a[dx])
4  if (sx = dx − 1) then                               // Caso base 2
5      if (a[sx] < a[dx]) then
6          return (a[sx], a[dx])
7      else
8          return (a[dx], a[sx])
9  cx ← (sx + dx)/2 ;                                  // Passo ricorsivo
10 (min1, max1) ← MAX − MIN(a, sx, cx);
11 (min2, max2) ← MAX − MIN(a, cx + 1, dx);
12 return (min(min1, min2), max(max1, max2))
```

Si calcolano i costi: se n = 1 si si avrà:

- 1 per `if(sx = dx)`
- 1 per `return a[sx], a[dx]`

quindi in tutto si avrà T(n) = 2. Se n = 2 allora T(n) = 4 perché ci sono:

- 1 per `if(sx = dx)`
- 1 per `if(a[sx] < a[dx])`
- 1 per `if(sx=dx-1)`
- 1 per uno dei due `return`

Il tempo per risolvere su n elementi equivale a:

- 1 per `if(sx = dx)`
- 1 per `if(sx=dx-1)`
- 1 per `cx` (**Divide**)
- 2 per min e max nel return (**Combina**)

e infine le chiamate ricorsive costano ognuno T(n/2) quindi in totale il costo **Impera** corrisponde a T(n) = T(n/2) + T(n/2) + 5. Questa equazione si chiama **equazione di ricorrenza** ed è presente in tutti gli algoritmi che contengono una chiamata ricorsiva a sé stesso. Una equazione di ricorrenza esprime il tempo di esecuzione totale di un problema di dimensione n in funzione del tempo di esecuzione per input più piccoli. Si risolve:

$$T(n) = \begin{cases} 2 & se\ n = 1 \\ 4 & se\ n = 2 \\ 2T(n/2) + 5 & negli\ altri\ casi \end{cases}$$

Build Solution	*Exspand Scratch*
$T(n) = 2T(n/2) + 5$	$T(n/2) = 2T(n/2^2) + 5$
$T(n) = 2[2T(n/2^2) + 5] + 5$	
$T(n) = 2^2T(n/2^2) + 2 \cdot 5 + 5$	$T(n/2^2) = 2T(n/2^3) + 5$
$T(n) = 2^2[2T(n/2^3) + 5] + 2 \cdot 5 + 5$	
$T(n) = 2^3T(n/2^3) + 2^2 \cdot 5 + 2 \cdot 5 + 5$	

dopo una serie di passi si arriva a capire la formula generica: $2^kT(n/2^k) + 5\sum_{i=0}^{k-1} 2^i$

ci si ferma quando si arriva al caso base cioè quando l'argomento è 1, ovvero $(n/2^k) = 1 \Rightarrow n = 2^k \Rightarrow \boxed{k = \log_2 n}$

se ora si sostituisce: $2^{\log_2 n}T(n/2^{\log_2 n}) + 5\sum_{i=0}^{\log_2 n-1} 2^i$ mettendo tutto insieme il costo è *O(n)*. Si poteva arrivare a questo risultato anche pensando che per trovare il massimo in un array si devono fare minimo n confronti perché si devono confrontare tutti. Stesso discorso per trovare il minimo in un array. Una generica equazione di ricorrenza legata ad una soluzione di tipo divide-et-impera ha sempre questa forma:

$$T(n) = \begin{cases} costante & se\ n \leq k \\ D(n) + aT(n/b) + C(n) & se\ n > k \end{cases}$$

dove D(n) è il costo per dividere, C(n) il costo per combinare i risultati e infine vi è l'impera cioè il costo per risolvere i sotto problemi: a sono le chiamate ricorsive e ogni chiamata ricorsiva ha una dimensione di (n/b)

2.4 Merge sort

Un algoritmo di ordinamento che utilizza il metodo divide et impera è il Merge sort e intuitivamente opera nel seguente modo:

- **Divide** la sequenza degli n elementi da ordinare in due sottosequenze di $n/2$ elementi ciascuna
- **Impera**, ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo Merge sort
- **Combina**, fonde le due sottosequenze ordinate per generare la sequenza ordinata

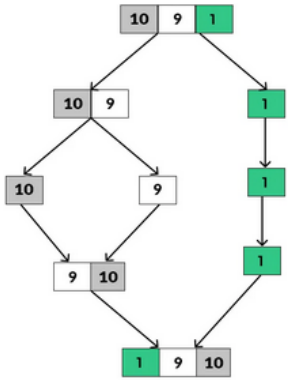
La ricorsione "tocca il fondo" quando la sequenza da ordinare ha lunghezza 1, in quel caso è già ordinato. Il punto chiave dell'algoritmo Merge sort è la fusione di due sottosequenze ordinate nel passo *combina*. Per effettuare la fusione si utilizza una procedura ausiliaria *MERGE(a, sx, cx, dx)*, dove *a* è un array e *sx, cx* e *dx* sono indici dell'array tali che $sx \leq cx < dx$. La procedura assume che i sottoarray *a[sx...cx]* e *a[cx+1...dx]* siano ordinati e li fonde per formare un unico sottoarray ordinato che sostituisce il sottoarray corrente *a[sx...dx]*.

Algorithm 4: costo $O(n)$

```
1  MERGE(a, sx, cx, dx)
2  n1 ← cx − sx + 1;
3  n2 ← dx − cx;
4  create arrays L[1...n + 1] e R[1...n2 + 1];
5  for i ← 1 to n1 do
6      L[i] = a[sx + i − 1]
7  for j ← 1 to n2 do
8      R[i] = a[cx + j]
9  L[n1 + 1] = ∞;
10 R[n2 + 1] = ∞;
11 i = 1, j = 1;
12 for k = sx to dx do
13     if L[i] ≤ R[j] then
14         a[k] = L[i];
15         i = i + 1;
16     else
17         a[k] = R[j];
18         j = j + 1
```

Algorithm 5: costo $O(n \log_2 n)$

```
1  MERGE-SORT(a, sx, dx)
2  if sx < dx then
3      cx = (sx + dx)/2;
4      MERGE − SORT(a, sx, cx);
5      MERGE − SORT(a, cx + 1, dx);
6      MERGE(a, sx, cx, dx)
```



La procedura *MERGE* impiega un tempo *O(n)* dove *n = dx - sx + 1* è il numero totale di elementi da fondere. In dettaglio la procedura opera nel modo seguente: calcola la sequenza *n1* del sottoarray *a[sx...cx]*, calcola la sequenza *n2* del sottoarray *a[cx+1...dx]*. Si ha necessita di utilizzare memoria aggiuntiva, quindi si creano due array *L* e *R* che contengono rispettivamente *a[sx...cx]*, *a[cx+1...dx]* ed in entrambi vi è un valore **sentinella** che contiene un valore speciale che si usa per semplificare il codice. Si usa ∞ come valore sentinella in modo che quando si presenta come numero, esso non può essere il numero più piccolo, ma questo accade quando tutti i valori non-sentinella sono già stati salvati nell'array di output. Le righe 11-18 mantengono la seguente invariante di ciclo:

Invariante di ciclo: all'inizio di ogni for, *a[sx...k-1]* contiene i *k-sx* elementi più piccoli di *L* e *R*, in ordine. *L[i]* e *R[i]* sono i più piccoli che non sono stati ancora copiati in *a*.

- **Inizializzazione**, all'inizio del primo ciclo *k = sx* quindi il sottoarray *a[sx...k-1]* è vuoto, poiché *i = j = 1*, *L[i]* e *R[j]* sono i più piccoli elementi nei rispettivi array tra quelli che non sono stati ancora copiati in *a*.
- **Mantenimento**, si suppone che *L[i]* $\leq$ *R[j]*, quindi *L[i]* è il più piccolo elemento non ancora copiato in *a*. Dato che *a[sx...k-1]* contiene i *k-sx* elementi più piccoli, dopo che la riga 14 ha copiato *L[i]* in *a[k]*, il sottoarray *a[sx...k]* conterrà i *k-sx+1* elementi più piccoli. Incrementando *k*, cioè aggiornando il ciclo for, e *i* si stabilisce l'invariante di ciclo per la successiva iterazione. Analogo ragionamento per *L[i]* $>$ *R[j]*
- **Terminazione**, alla fine del ciclo *k = dx +1*. Per l'invariante *a[sx...k-1]* contiene i *k - sx = dx - sx +1* elementi ordinati più piccoli di *L* e *R*. Tutti gli elementi tranne le sentinelle sono stati copiati in *a*

Adesso si può utilizzare la procedura *MERGE* come subroutine nell'algoritmo *MERGE-SORT*. Quest'ultimo ordina gli elementi nel sottoarray *a[sx...dx]*. Se *sx* $\geq$ *dx* il sottoarray ha al massimo un elemento e quindi è già ordinato, altrimenti, il passo *divide* calcola un indice *cx* che separa *a[sx...dx]* in due sottoarray. Se vi sono n $>$ 1 elementi si suddivide il tempo di esecuzione nel modo seguente:

- **Divide**, calcola *cx* ciò richiede un tempo costante
- **Impera**, si risolve in modo ricorsivo i 2 sottoproblemi, ciascuno di dimensione $n/2$, quindi si avrà $2T(n/2)$ al tempo di esecuzione
- **Combina**, il costo del *MERGE* è *O(n)*

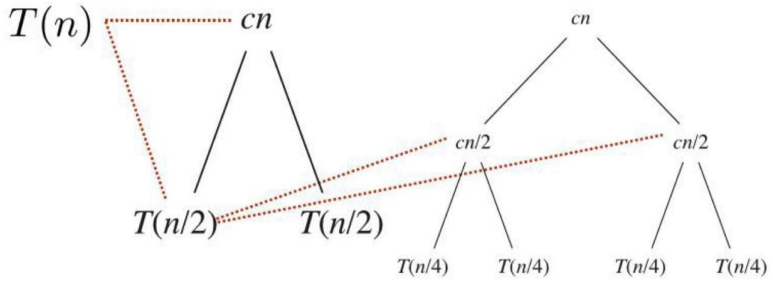
Quando si sommano le funzioni *D(n)* e *C(n)* si sta sommando una funzione *O(1)* e *O(n)* che da come risultato una funzione lineare *O(n)* che si somma a sua volta al passo *impera*, allora:

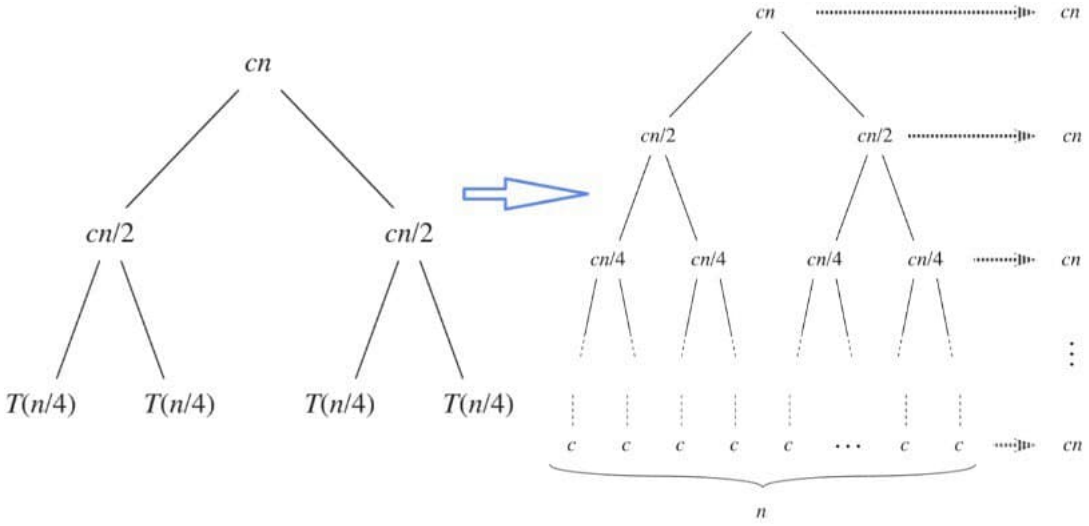
$$T(n) = \begin{cases} c & se\ n = 1 \\ 2T(n/2) + cn & se\ n > 1 \end{cases}$$

dove la costante *c* rappresenta il tempo richiesto per risolvere i problemi di dimensione 1 sia il tempo per l'elemento dell'array nei passi *divide* e *combina*.

Build Solution	*Exspand Scratch*
$T(n) = 2T(n/2) + n$	$T(n/2) = 2T(n/2^2) + n/2$
$T(n) = 2[2T(n/2^2) + n/2] + n$	
$T(n) = 2^2T(n/2^2) + n + n$	

Formula generica: $2^kT(n/2^k) + k \cdot n$. Si trova k: $(n/2^k) = 1 \Rightarrow n = 2^k \Rightarrow \boxed{k = \log_2 n}$
Si sostituisce: $2^{\log_2 n}T(n/2^{\log_2 n}) + \log_2 n \cdot n = cn + n\log_2 n$ mettendo tutto insieme il costo è $O(n \log_2 n)$.
Un altro modo per risolvere l'equazione di ricorrenza è costruire l'**albero di ricorsione** risultante:





Il termine *cn* è la radice, cioè il costo sostenuto al primo livello di ricorsione e i due sottoalberi della radice sono le due ricorrenze più piccole di *T(n/2)*. Per avere il costo totale si devono sommare ogni livello dell'albero dove ognuno costa *cn*. La somma dipende dall'altezza dell'albero che è *n*, si parte da 2^k elementi, si divide sempre per 2 e si arriva alle foglie dopo $\log_2 n$ livelli. Quindi il costo totale è $cn + cn\log_2 n$ allora $O(n \log_2 n)$.

Tabella riassuntiva:

Algoritmo	Tempo Migliore	Tempo Peggiore	Note
selection-sort	$O(n^2)$	$O(n^2)$	• lento • in-place • per piccoli insiemi (< 1K)
insertion-sort	$O(n)$	$O(n^2)$	• lento • in-place • per piccoli insiemi (< 1K)
merge-sort	$O(n \log n)$	$O(n \log n)$	• veloce • per grandi insiemi (> 1M)

Chapter 3

Master theorem

3.1 Risolvere le ricorrenze

Per risolvere le ricorrenze si è usato il metodo dello svolgimento e il metodo ad albero. Il metodo del master theorem rappresenta un *ricettario* per risolvere ricorrenze della forma $T(n) = aT(n/b) + f(n)$ dove $a \geq 1$ e $b > 1$ sono costanti e $f(n)$ è una funzione asintoticamente positiva. Il teorema presenta 3 casi, dopo aver individuato il caso di appartenenza della equazione di ricorrenza si può concludere qual è la soluzione. L'intuizione dell'applicazione del master theorem è di confrontare la funzione $f(n)$, cioè il costo per la divisione e combina, con la funzione $n^{log_b a}$ in base a chi "vince" questo confronto, si determina la soluzione. I tre casi di applicazione:

1. $f(n) = O(n^{log_b a - \epsilon})$ per qualche costante $\epsilon > 0$

   $f(n)$ cresce polinomialmente più lentamente di $n^{log_b a}$ di un fattore n^ϵ quindi la soluzione è $\boldsymbol{T(n) = \Theta(n^{log_b a})}$

2. $f(n) = \Theta(n^{log_b a} lg^k n)$ per qualche costante $k \geq 0$

   $f(n)$ e $n^{log_b a}$ crescono allo stesso modo quindi la soluzione è $\boldsymbol{T(n) = \Theta(n^{log_b a}\, lg^{k+1} n)}$

3. $f(n) = \Omega(n^{log_b a + \epsilon})$ per qualche costante $\epsilon > 0$

   $f(n)$ cresce polinomialmente più velocemente di $n^{log_b a}$ di un fattore n^ϵ e $f(n)$ soddisfa la condizione che $af(n/b) \leq cf(n)$ per qualche costante $c < 1$ e $n > n_0$ quindi la soluzione è $\boldsymbol{T(n) = \Theta(f(n))}$

Non sempre è possibile applicare il master theorem, soprattutto per il primo e il terzo caso. Entrambi prevedono un ϵ, se esso non esiste allora non si può applicare il teorema.

- **Esempio 1.** $T(n) = 9T(n/3) + n$

  a = 9
  b = 3
  f(n) = n

  Si deve fare il confronto fra n e $n^{log_3 9} = n^2$
  Essendo n^2 il più veloce si prova il primo caso: $f(n) = O(n^{2-\epsilon})$
  Si può prendere come ϵ = 1 e quindi $O(n)$
  A questo punto si può concludere che la soluzione è $T(n) = \Theta(n^2)$

- **Esempio 2.** $T(n) = T(2n/3) + 1$

  a = 1
  b = 3/2
  f(n) = 1

  Si deve fare il confronto fra 1 e $n^{log_{\frac{3}{2}} 1} = n^0 = 1$
  Si applica il secondo caso: $f(n) = \Theta(1 \cdot lg^k n)$
  Si può utilizzare $k = 0$ e quindi $\Theta(1)$
  A questo punto si può concludere che la soluzione è $T(n) = \Theta(lgn)$

- **Esempio 3.** $T(n) = 3T(n/4) + nlgn$

  a = 3
  b = 4
  f(n) = nlgn

  Si deve fare il confronto fra $nlgn$ e $n^{log_4 3}$ = $n^{0.793}$
  Si applica il terzo caso: $f(n) = \Omega(n^{log_4 3 + \epsilon})$
  Si può prendere $\epsilon \approx 0.2$ per farlo diventare 1 quindi $\Omega(n)$.
  Ora si deve verificare l'altra condizione: $af(n/b) = 3\frac{n}{b} log\frac{n}{b} \leq cnlogn$
  Si trova c sapendo che $lg(n/4)$ è sicuramente più piccolo di $lg(n)$ quindi si può prendere $c = 3/4$ e la condizione è soddisfatta. Soluzione: $T(n) = \Theta(nlgn)$

- **Esempio 4.** $T(n) = 2T(n/2) + n$ (*Merge sort*)

  a = 2
  b = 2
  f(n) = n

  Si deve fare il confronto fra n e $n^{log_2 2}$ = n
  Si applica il secondo caso: $f(n) = \Theta(nlg^k n)$
  Si può utilizzare $k = 0$ e quindi $\Theta(n)$
  A questo punto si può concludere che la soluzione è $T(n) = \Theta(nlgn)$

- **Esempio 5.** $T(n) = 2T(n/2) + nlgn$

a =	2
b =	2
f(n) =	nlgn

Si deve fare il confronto fra $nlgn$ e $n^{log_2 2} = n$
Si potrebbe applicare il caso 3: $f(n) = \Omega(n^{1+\epsilon}) = \Omega(n \cdot n^{\epsilon})$ ma non si riesce a trovare un ϵ tale $n \cdot n^{\epsilon}$ stia sotto a $nlgn$ quindi il terzo caso non si può applicare. Il caso 1 è impossibile. Allora rimane il secondo caso:
$f(n) = \Theta(nlg^k n)$ per $k = 1$ $f(n) = \Theta(nlgn)$, allora si può applicare questo caso e la soluzione è $T(n) = \Theta(nlg^2 n)$

- **Esempio 6.** $4T(n/2) + n^2/logn$

a =	4
b =	2
f(n) =	n^2/logn

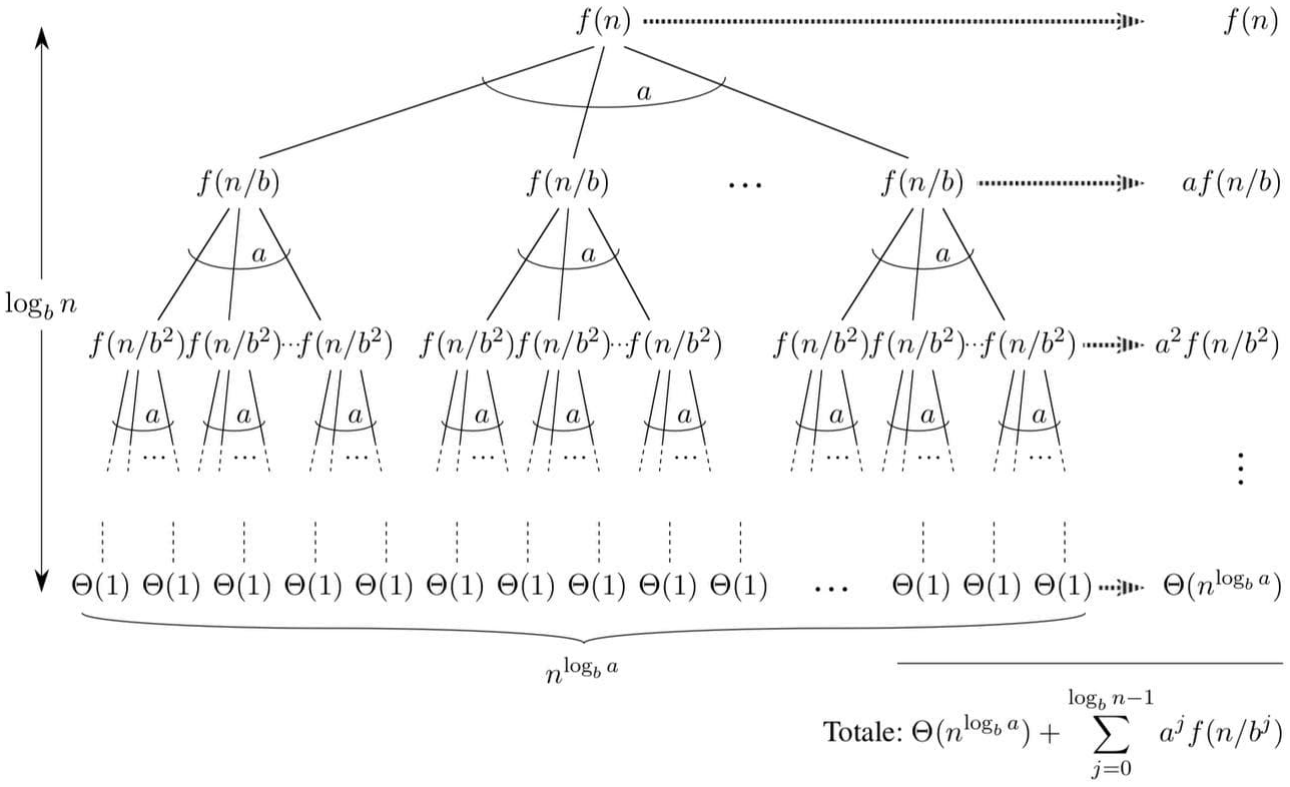
Si deve fare il confronto fra $n^2/logn$ e $n^{log_4 2} = n^2$
Il secondo caso è impossibile. $f(n)$ è più lenta quindi il terzo caso non può essere. Si prova il caso 1: $f(n) = O(n^{2-\epsilon}) = n^2/n^{\epsilon}$ Vs $n^2/logn$
n^{ϵ} sta sempre sopra a $logn$, quindi il master theorem non si può applicare

3.2 Dimostrazione master theorem

Si parte da questa equazione di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ aT(n/b) + f(n) & se\ n = b^j \end{cases}$$

il suo albero di ricorsione corrisponde a:



La radice dell'albero ha costo $f(n)$ e ha a figli, ciascuno di costo $f(n/b)$. Ognuno di questi figli ha, a sua volta, a figli con un costo di $f(n/b^2)$ quindi ci sono a^2 nodi alla distanza 2 dalla radice. In generale, ci sono a^j nodi alla distanza j dalla radice, ciascuno dei quali ha un costo $f(n/b^j)$. L'altezza dell'albero è $log_b n$. L'albero ha $a^{log_b n} = n^{log_b a}$ foglie. Il costo per un livello j di nodi interni è $a^j f(n/b^j)$ quindi il totale per tutti i livelli interni è $\sum_{j=0}^{log_b n-1} a^j f(n/b^j)$. Il costo di tutte le foglie, che è il costo per svolgere $n^{log_b a}$ sottoproblemi di dimensione 1 è $\Theta(n^{log_b a})$.

3.2.1 Lemma

Il cuore della dimostrazione è capire quanto vale la sommatoria che fa parte di un lemma diviso nei 3 casi del master theorem. Si chiama $g(n) = \sum_{j=0}^{log_b n-1} a^j f(n/b^j)$

1. **Se $f(n) = O(n^{log_b a - \epsilon})$ per qualche costante $\epsilon > 0$, allora $g(n) = O(n^{log_b a})$**

   Si conosce che $f(n) = O(n^{log_b a - \epsilon})$ quindi $f(n/b^j) = O((n/b^j)^{log_b a - \epsilon})$ ora si sostituisce in $g(n)$ e si ottiene $O(\sum_{j=0}^{log_b n - 1} a^j (n/b^j)^{log_b a - \epsilon})$. Si calcola la sommatoria:

   $$\sum_{j=0}^{log_b n - 1} a^j (\boldsymbol{n}/b^j)^{\boldsymbol{log_b a - \epsilon}}) \qquad \text{si porta fuori dalla sommatoria perché non dipende da } j$$

   $$= n^{log_b a - \epsilon} \sum_{j=0}^{log_b n - 1} a^j \boldsymbol{(1/b^j)^{log_b a - \epsilon}} \qquad \text{si distribuisce la potenza}$$

   $$= n^{log_b a - \epsilon} \sum_{j=0}^{log_b n - 1} a^j (b^\epsilon / \boldsymbol{b^{log_b a}})^j \qquad b^{log_b a} = a$$

   $$= n^{log_b a - \epsilon} \sum_{j=0}^{log_b n - 1} a^j (b^\epsilon / a)^j \qquad \text{si porta dentro } j$$

   $$= n^{log_b a - \epsilon} \sum_{j=0}^{log_b n - 1} \boldsymbol{a^j} (b^\epsilon)^j / \boldsymbol{a^j} \qquad \text{si semplifica e per } \sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

   $$= n^{log_b a - \epsilon} \left(\frac{\boldsymbol{b^{\epsilon \cdot log_b n}} - 1}{b^\epsilon - 1} \right) \qquad b^{\epsilon \cdot log_b n} = n^\epsilon$$

   $$= n^{log_b a - \epsilon} \left(\frac{n^\epsilon - 1}{\boldsymbol{b^\epsilon} - 1} \right) \qquad b \text{ e } \epsilon \text{ sono costanti}$$

   $$= \frac{n^{log_b a}}{\boldsymbol{n^\epsilon}} O(\boldsymbol{n^\epsilon}) \qquad \text{si semplifica}$$

   $$= \boxed{O(n^{log_b a})}$$

   $$\Rightarrow T(n) = \Theta(n^{log_b a}) + \sum_{j=0}^{log_b n - 1} a^j (n/b^j)$$

   $$= \Theta(n^{log_b a}) + O(n^{log_b a})$$

   $$= \Theta(n^{log_b a})$$

2. **Se $f(n) = \Theta(n^{log_b a})$, allora $g(n) = \Theta(n^{log_b a} lgn)$**

   Come prima $f(n/b^j) = \Theta((n/b^j)^{log_b a})$ ora si sostituisce in $g(n)$ e si ottiene $\Theta(\sum_{j=0}^{log_b n - 1} a^j (n/b^j)^{log_b a})$. Si calcola la sommatoria:

   $$\sum_{j=0}^{log_b n - 1} a^j (n/b^j)^{log_b a}$$

   $$\ldots \qquad \text{stessi procedimenti di prima...}$$

   $$= n^{log_b a} \sum_{j=0}^{log_b n - 1} a^j (1/a^j) \qquad \text{...fino a qui}$$

   $$= n^{log_b a} \sum_{j=0}^{log_b n - 1} 1$$

   $$= n^{log_b a} (log_b n)$$

   $$= \boxed{\Theta(n^{log_b a} lgn)}$$

$$\Rightarrow T(n) = \Theta(n^{log_b a}) + \sum_{j=0}^{log_b n-1} a^j(n/b^j)$$

$$= \Theta(n^{log_b a}) + \Theta(n^{log_b a} lgn)$$

$$= \Theta(n^{log_b a} lgn)$$

3. **Se $af(n/b) \leq cf(n)$ per qualche c $<$ 1 e per ogni n sufficientemente grande, allora $g(n) = \Theta(f(n))$**

   Si riscrive l'assunzione come $f(n/b) \leq (c/a)f(n)$ la si itera j volte per ottenere $f(n/b^j) \leq (c/a)^j f(n)$. Si sostituisce in $g(n) = \Theta(\sum_{j=0}^{log_b n-1} a^j(c/a)^j f(n))$. Si calcola la sommatoria:

$$\sum_{j=0}^{log_b n-1} a^j(c/a)^j f(n) \qquad \text{si porta dentro j}$$

$$\leq \sum_{j=0}^{log_b n-1} \boldsymbol{a^j}(c^j/\boldsymbol{a^j})f(n) \qquad \text{si semplifica}$$

$$\leq \sum_{j=0}^{log_b n-1} c^j f(n)$$

$$\leq f(n)\sum_{j=0}^{\infty} c^j \qquad \text{quando la sommatoria è infinita e } |x| < 1 \text{ allora } \sum_{k=0}^{\infty} x^k = \left(\frac{1}{1-x}\right)$$

$$= f(n)\left(\frac{1}{1-c}\right) \qquad \text{c è costante}$$

$$= \boxed{\Theta(f(n))}$$

$$\Rightarrow T(n) = \Theta(n^{log_b a}) + \sum_{j=0}^{log_b n-1} a^j(n/b^j)$$

$$= \Theta(n^{log_b a}) + \Theta(f(n))$$

$$= \Theta(f(n))$$

Esercizio 1 Siano:

- $T(n) = 7T(n/2) + n^2$ il costo di funzione di un algoritmo A, e
- $T(n) = aT(n/4) + n^2$ il costo di un'altra funzione di un algoritmo B

Qual è il massimo valore intero di a che rende B asintoticamente più veloce di A?

Algoritmo A:

a =	7
b =	2
f(n) =	n^2

n^2 Vs $n^{log_2 7} \approx n^{2.8}$
Primo caso: $O(n^{log_2 7 - \epsilon})$ con $\epsilon = 0.8 \Rightarrow T(n) = O(n^{log_2 7})$

Algoritmo B:

a =	a
b =	4
f(n) =	n^2

n^2 vs $n^{log_4 a}$

- Per $a = 16$ Caso 2 $\Rightarrow log_4 16 = log_4 4^2 = 2log_4 4 = 2 \Rightarrow n^2$ cioè $\Theta(n^2)$ con $k = 0 \Rightarrow T(n) = \Theta(n^2 logn)$
- Per $a < 16$ Caso 3 $\Rightarrow \Theta(n^2)$
- Per $a > 16$ Caso 1 $\Rightarrow O(n^{log_2 a})$

B è più veloce di A per a $<$ 49

Chapter 4

Problema della ricerca e limiti inferiori

4.1 Ricerca sequenziale

INPUT: array a di n interi e una chiave k intero
OUTPUT: sapere se k è nell'array restituendo la posizione i, se non c'è restituire -1

Algorithm 6: costo $\Theta(n)$

```
1 SEQ-SEARCH(a, n, k)
2 pos ← −1, i ← 1 ;                                   // Θ(1)
3 while i ≤ n and pos = −1 do                         // Θ(T(W))
4     if a[i] = k then
5         pos ← i
6     else
7         i ← i + 1
8 return pos ;                                        // Θ(1)
```

dove $T(W)$ è il numero di volte in cui viene eseguito il while.

- **Caso ottimo**:
  - k è in prima posizione
  - $n = 0 \vee a = 0$

  in entrambi i casi l'algoritmo non dipende dal numero di elementi di a $\Rightarrow T(n) = \Theta(1) + \Theta(1) + \Theta(1) = \Theta(1)$.
- **Caso peggiore**: k non è presente, cioè $TW = n + 1$ e allora $\Rightarrow T(n) = \Theta(1) + \Theta(n) + \Theta(1) = \Theta(n)$.
- **Caso medio**: ci sono n posizioni in cui k può trovarsi +1 per considerare l'assenza

possibili soluzioni:	-1	1	2	3	...	n
$\downarrow$						
numero confronti:	n	1	2	3	...	n

$$
\begin{aligned}
C(n) = \frac{n+1+2+3+...+n}{n+1} &= \frac{1}{n+1}\left[\left(\sum_{i=0}^{n} i\right) + n\right] && \text{per } \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \\
&= \frac{1}{n+1}\left[\left(\frac{n(n+2)+n}{2}\right)\right] \\
&= \frac{n}{2} + \frac{n}{n+1} \\
&= \Theta(n)
\end{aligned}
$$

La ricerca sequenziale è lineare nel caso peggiore e medio, ed è costante al caso ottimo.

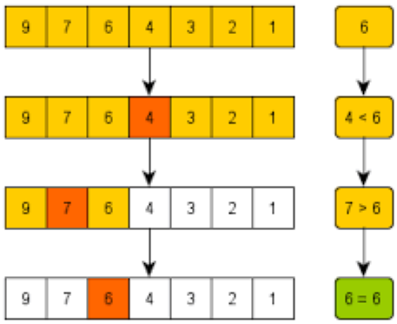
4.2 Ricerca binaria

Questo algoritmo cerca un elemento all'interno di un array che deve essere **ordinato** in ordine crescente, effettuando mediamente meno confronti rispetto ad una ricerca sequenziale, e quindi più rapidamente rispetto ad essa perché, sfruttando l'ordinamento, dimezza l'intervallo di ricerca ad ogni passaggio. La ricerca binaria opera nel modo seguente:

1. si confronta il valore da cercare con l'elemento centrale dell'array
2. se il valore da cercare è uguale all'elemento centrale, allora la ricerca termina positivamente
3. se invece la chiave è minore dell'elemento centrale, si effettua la ricerca solo sulla porzione di array a sinistra. Altrimenti si effettua la ricerca solo sulla porzione di array a destra.

Algorithm 7: costo $\Theta(log_2 n)$

```
1  BIN-SEARCH(a, sx, dx, k)
2  if (sx > dx) then
3      return -1
4  cx = (sx + dx)/2;
5  if (a[cx] = k) then
6      return cx
7  if (a[cx] < k) then
8      return BIN - SEARCH(a, sx, cx, k)
9  else
10     return BIN-SEARCH(a, cx+1, dx, k)
```



L'equazione di ricorrenza corrisponde a : $T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ T(n/2) + \Theta(1) & se\ n > 1 \end{cases}$

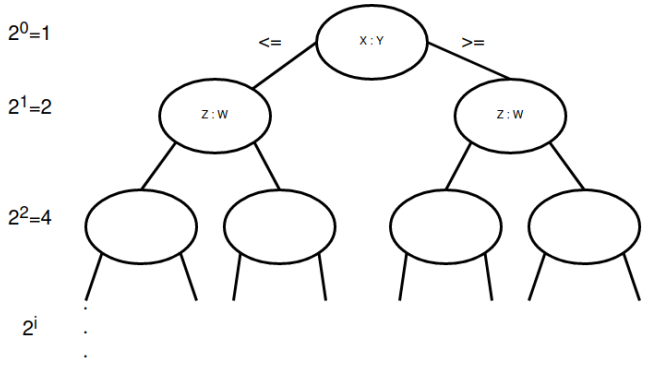
Il costo dell'impera è $T(n/2)$ nonostante ci siano 2 chiamate ricorsive nel codice perché si esegue una sola chiamata alla volta essendo all'interno di un if/else. Si risolve:

Build Solution	*Exspand Scratch*
$T(n) = T(n/2) + c$	$T(n/2) = T(n/2^2) + c$
$T(n) = T(n/2^2) + c + c$	$T(n/2^2) = T(n/2^3) + c$
$T(n) = T(n/2^3) + c + c + c$	

Formula generica: $T(n) = T(n/2^k) + k \cdot c$. Si trova k: $(n/2^k) = 1 \Rightarrow n = 2^k \Rightarrow \boxed{k = log_2 k}$
Si sostituisce $T(n) = T(n/2^{log_2 n}) + (log_2 n)c = T(1) + (log_2 n)c = \Theta(log_2 n)$

Per risolvere il problema della ricerca esiste un algoritmo migliore? Per esempio $O(loglogn)$? Per non andare a tentativi si cerca di stabilire un **limite inferiore** al problema. L'operazione principale nel problema della ricerca è l'operazione di confronto fra 2 elementi. Si considerano i due casi:

- Array non ordinato = $\Omega(n)$ perché una volta fatto il confronto $x : y$ non si può dedurre niente sugli altri elementi. Tutti gli elementi devono essere confrontati
- Array ordinato, si utilizza l'**albero della decisione**:



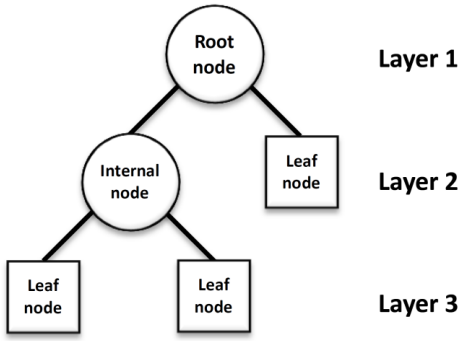
Il numero di soluzioni sono $n+1$ includendo anche la ricerca senza successo. Si deve poter distinguere sull'albero di decisione su $n+1$ possibilità

$$\begin{aligned} 2^i \geq n+1 \quad &= \quad log_2 2^i \geq log(n+1) \\ &= \quad i \geq log(n+1) \\ &= \quad i \geq \Omega(logn) \end{aligned}$$

i è il numero di confronti necessari a trovare k nel caso pessimo. Per array ordinato $\begin{cases} operazioni\ O(logn) \\ limite\ inferiore\ \Omega(logn) \end{cases}$ poiché i due limiti sono equivalenti in ordine di grandezza, si può concludere che la ricerca binaria è un algoritmo **ottimo**.

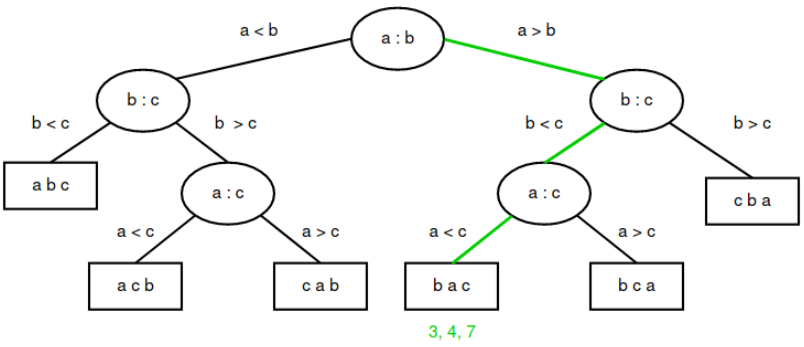
4.3 Limiti inferiori

Stabilire i limiti inferiori non è semplice. Gli ordinamenti per confronti possono essere visti con un albero di decisione che è un albero binario che rappresenta i confronti fra elementi. L'esecuzione dell'algoritmo corrisponde a tracciare un cammino semplice dalla radice dell'albero fino a una foglia. Ogni nodo interno rappresenta un confronto $x_i \leq y_j$. Quando raggiunge una foglia, l'algoritmo ha stabilito l'ordinamento. Poiché qualsiasi algoritmo di ordinamento corretto deve essere in grado di produrre ogni permutazione del suo input, una condizione necessaria affinché un ordinamento per confronti sia corretto è che ciascuna delle permutazioni di $n!$ permutazioni di n elementi appaia come una foglia dell'albero e che ciascuna di queste foglie sia raggiungibile dalla radice. La lunghezza del cammino più lungo dalla radice fino a una delle sue foglie rappresenta il numero di confronti che si svolge nel caso peggiore. Di conseguenza il numero di di confronti nel caso peggiore è uguale all'altezza dell'albero di decisione. Scoprire un algoritmo per la risoluzione di problema equivale a stabilire un limite superiore di complessità nel senso che risulta automaticamente provato come il problema sia risolubile entro i limiti di tempo e spazio stabili dall'algoritmo. La ricerca dei limiti inferiori di complessità risponde alla domanda se si possono determinare algoritmi più efficienti di quelli noti. Lo studio è spesso limitato all'ordine di grandezza, per cui si applica la notazione Ω. Quando la complessità di un algoritmo è pari al limite inferiore di complessità determinato per il problema, l'algoritmo si dice ottimo. Nel sorting si è visto:



Insertion sort $O(n^2)$ Selection sort $O(n^2)$ Merge sort $O(nlogn)$

si cerca di abbassare la complessità di questi algoritmi trovando il numero minimo di confronti necessari per ordinare un insieme di n elementi. Si costruisce l'albero di decisione tenendo conto che deve essere bilanciato, cioè non si vuole che da un ramo di arrivi subito alla soluzione e poi nell'altro ramo si devono considerare tutte le altre possibilità. Si suppone di avere 3 elementi distinti $a = 4, b = 3, c = 7$



Numero soluzioni: 3!
Ora si studia il caso su n elementi distinti, il numero di soluzioni sarà: $n!$. L'albero di decisione deve avere tante foglie da contenere tutte le possibili soluzioni cioè $2^i \geq n!$

$$\begin{aligned} 2^i \geq n! \quad &= \quad log_2 2^i \geq log(n!) \\ &= \quad i \geq log(n!) \\ &= \quad i \geq \Theta(nlogn) \end{aligned}$$

Approssimazione di Sterling $log(n!) = \Theta(nlogn)$

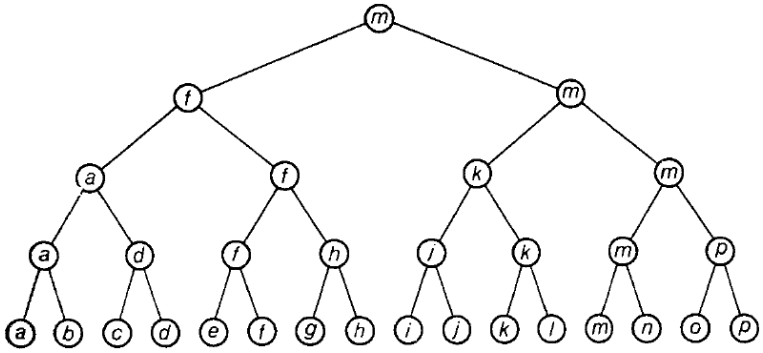
dove i rappresenta il livello delle foglie nell'albero di decisione. L'albero di decisione è associato a un algoritmo quindi i rappresenta il numero di confronti che sono necessari dall'input di partenza fino ad arrivare alla permutazione ordinata. Si può concludere che per il problema dell'ordinamento è $\Omega(nlogn)$ cioè non si può trovare un algoritmo che faccia un numero di confronti minori di $nlogn$ e quindi il Merge sort è un algoritmo ottimo. Non esiste una teoria generale per determinare i limiti inferiori, ma con alcune linee di studio si possono condurre a risultati soddisfacenti:

- **Dimensione dell'input/output**
  Si può osservare come $\Omega(n)$ sia un banale limite inferiore per operazioni che richiedono l'esame di tutti gli n elementi di un insieme, come per esempio nel sorting, ma $\Omega(n)$ è minore rispetto a quello stabilito cioè $\Omega(nlogn)$ quindi $\Omega(n)$ non è significativo.

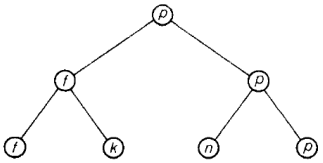
- **Eventi contabili**
  Un altro criterio che permette di stabilire un limite inferiore di complessità è legato al verificarsi di un evento e questo evento si deve ripetere per un certo numero di volte. Si suppone di cercare il massimo elemento in un insieme di n elementi.

  – **Soluzione banale**, si suppone che il massimo sia il primo elemento, e si confronta il massimo corrente con gli altri elementi. Se si trova un elemento maggiore sarà il nuovo massimo corrente. Tutti gli elementi vanno controllati, quindi il limite inferiore è $\Omega(n)$. Si presume di voler valutare *esattamente* quanti confronti sono necessari. Si può stabilire qual è l'evento: *per essere giudicato non massimo, un elemento dell'array deve perdere in almeno un confronto*. In totale gli elementi non massimi sono $n-1$ confronti, quindi occorrono $n-1$ confronti.

  – **Problema del torneo**, si hanno n giocatori e si vuole scoprire quanti confronti servono per trovare il campione. Come ipotesi si può utilizzare la **tecnica dell'avversario**: il numero di giocatori che risultano minori di un altro giocatore per transitività deve essere minima. Per esempio si suppone ci sia un incontro con un giocatore x e un giocatore y e si conosce già che ci sono 100 giocatori che hanno perso con x. Nel caso in cui x vincesse, quest'ultimo acquisirebbe un nuovo giocatore, ma nel caso in cui y vincesse allora y è maggiore di 100 altri giocatori, quindi si acquisirebbe un numero enorme di informazioni per transitività, e questo non deve succedere. L'incontro deve essere bilanciato: per esempio i giocatori x e y hanno vinto 50 incontri a testa quindi se $x > y$, il numero di giocatori acquisiti è 50 + 1, nel caso in cui $y > x$, il numero di giocatori acquisiti è sempre 50 + 1. La soluzione è rappresentata da questo albero binario completo:



Un albero binario completo con n foglie ha $n-1$ nodi interni, questo indica che il numero di confronti necessari per proclamare il vincitore con l'algoritmo del torneo è $n-1$ ed essendo uguale al limite inferiore, l'algoritmo del torneo è ottimo. Il vincitore della finale è proclamato vincitore in virtù della transitività della bravura ma se ora si volesse determinare la seconda vincitrice non è detto che la perdente della finale sia l'effettiva seconda. Per stabilire il vero secondo massimo si devono prendere tutti i valori che hanno perso contro il campione e ottenere un nuovo torneo. Questo algoritmo si chiama **problema del doppio torneo**:



* 1° torneo su n giocatori $\Rightarrow n-1$ confronti
* 2° torneo sui giocatori che hanno perso contro il campione $\Rightarrow log_2 n - 1$ confronti

I giocatori che hanno perso contro il campione sono $log_2 n$. Si indica con j gli incontri svolti dal campione e $M(j)$ il numero di giocatori che risultato minore del campione dopo j incontri. L'equazione di ricorrenza equivale a

$$M(j) = \begin{cases} 0 & se\ j = 0 \\ M(j-1) + M(j-1) + 1 & negli\ altri\ casi \end{cases}$$

ovvero $M(j-1)$ sono il numero di giocatori risultati sconfitti al tempo precedente $+1$ che rappresenta lo scontro diretto tra x e y. L'equazione non si può risolvere con il master theorem ma iterativamente:

Build Solution	*Exspand Scratch*
$M(j) = 2M(j-1) + 1$	$M(j-1) = 2M(j-2) + 1$
$M(j) = 2[2M(j-2) + 1] + 1$	
$M(j) = 2^2 M(j-2) + 2 + 1$	$M(j-2) = 2M(j-3) + 1$
$M(j) = 2^2[2M(j-3) + 1] + 2 + 1$	
$M(j) = 2^3 M(j-3) + 2^2 + 2 + 1$	

Formula generica: $2^k M(j-k) + \sum_{i=O}^{k-1} 2^i \Rightarrow 2^k M(j-k) + 2^k - 1$. Si trova k: $J - k = 0 \Rightarrow \boxed{k = j}$

Si sostituisce $2^j M(0) + 2^j - 1 \Rightarrow 2^j - 1$ quindi $M(j) = 2^j - 1$.

Per stabilire quanti incontri deve fare il campione servono $n-1$ giocatori che risultino minori del campione cioè $2^j - 1 = n - 1$ ovvero $j = log_2 n$. Gli incontri che il campione deve disputare sono $log_2 n$ e i perdenti del campione sono j. Il numero di confronti che deve essere fatto per stabilire il primo e il secondo $C(n) = n - 1 + j - 1$ cioè $n - 1$ sono gli incontri per proclamare il campione e $j - 1$ il numero di confronti tra i perdenti del campione quindi sostituendo $C(n) = n + log_2 n - 2$ l'algoritmo del doppio torneo è ottimo.

Chapter 5

Quick sort e selezione randomizzata

5.1 Quick Sort

Il Quick sort come il Merge sort è basato sul paradigma del divide et impera. Questi sono i tre passi del processo per ordinare un generico sottoarray $a[sx...dx]$

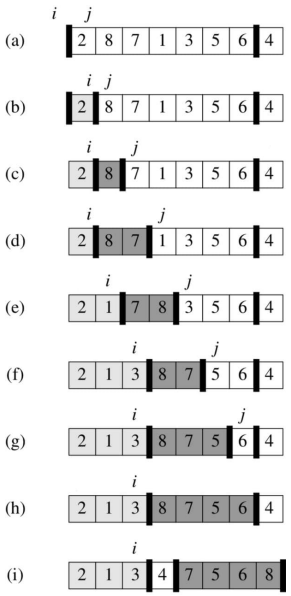
- **Divide**, si sceglie un elemento **pivot** tale da partizionare in place l'array $a[sx...dx]$ in due sottoarray $a[sx...px-1]$ e $a[px+1...dx]$ tali che ogni elemento di $a[sx...px-1]$ sia minore o uguale a $a[px]$ che, a sua volta, è minore o uguale ad ogni elemento di $a[px+1,...,dx]$. Si calcola l'indice px come parte di questa procedura di partizionamento
- **Impera**, ordinare i due sottoarray chiamando ricorsivamente Quick sort
- **Combina**, poiché i due sottoarray sono già ordinati, non occorre alcun lavoro per combinarli. L'intero array $a[sx...dx]$ è ordinato

Algorithm 8: costo caso medio $O(nlgn)$, caso peggiore $\Theta(n^2)$

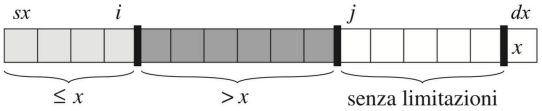
```
1 QUICK-SORT(a, sx, dx)
2 if sx < dx then
3     px ← PARTITION(a, sx, dx);
4     QUICK − SORT(a, sx, px − 1);
5     QUICK − SORT(a, px + 1, dx)
```

Algorithm 9: costo $\Theta(n)$

```
1 PARTITION(a, sx, dx)
2 x ← a[dx];
3 i ← sx − 1;
4 for j ← sx to dx − 1 do
5     if a[j] ≤ x then
6         i ← i + 1;
7         swap a[i] with a[j]
8 swap a[i + 1]with a[dx];
9 return i + 1
```



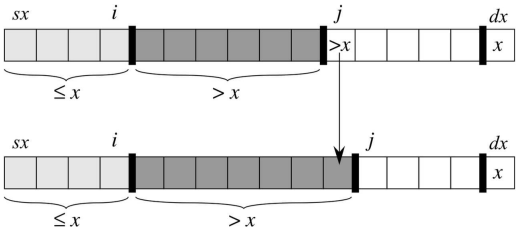
Gli elementi dell'array su sfondo grigio chiaro sono tutti nella prima partizione con valori minori o uguali a x. Gli elementi su sfondo grigio scuro sono nella seconda partizione con valori maggiori di x. Gli elementi con sfondo bianco non sono stati ancora spostati. In questo esempio l'ultimo elemento su sfondo bianco è il pivot. Il punto chiave dell'algoritmo è la procedura *PARTITION* che riaggangia il sottoarray $a[sx...dx]$ sul posto. *PARTITION* seleziona sempre un elemento $x = a[dx]$ come pivot intorno al quale partizionare il sottoarray $a[sx...dx]$. Durante l'esecuzione della procedura, l'array viene suddiviso in quattro regioni. All'inizio di ogni iterazione del ciclo for ogni regione soddisfa alcune proprietà che si enunciano sottoforma di invariante di ciclo:



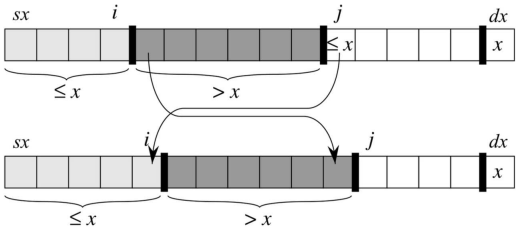
Invariante di ciclo: all'inizio di ogni iterazione del ciclo, per qualsiasi indice k dell'array,

1. Se $sx \leq k \leq i$, allora $a[k] \leq x$
2. Se $i+1 \leq k \leq j-1$ allora $a[k] > x$
3. Se $k = dx$ allora $a[k] = x$

- **Inizializzazione**, prima della iterazione del ciclo $i = sx - 1$ e $j = sx$. Non ci sono valori fra sx e i né fra $i+1$ e $j-1$, quindi le prime due condizioni sono soddisfatte. L'assegnazione alla riga 2 soddisfa la terza condizione
- **Conservazione**, ci sono due casi da considerare a seconda del risultato del test nella riga 5:
  - Se $a[j] > x$, l'unica azione da fare è incrementare j. Dopo l'incremento la condizione 2 è soddisfatta per $a[j-1]$ e tutte le altre posizioni non cambiano

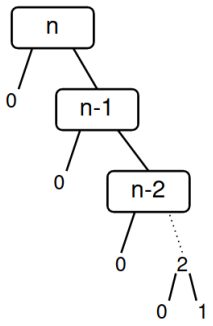


  - Se $a[j] \leq x$, viene incrementato l'indice i, vengono scambiati $a[i]$ e $a[j]$, e poi, viene incrementato l'indice j. In seguito allo scambio, adesso si ha $a[i] \leq x$ e la condizione 1 è soddisfatta. Analogamente, si ha anche $a[j-1] > x$, in quanto l'elemento che è stato spostato in $a[j-1]$ è, per l'invariante di ciclo, più grande di x

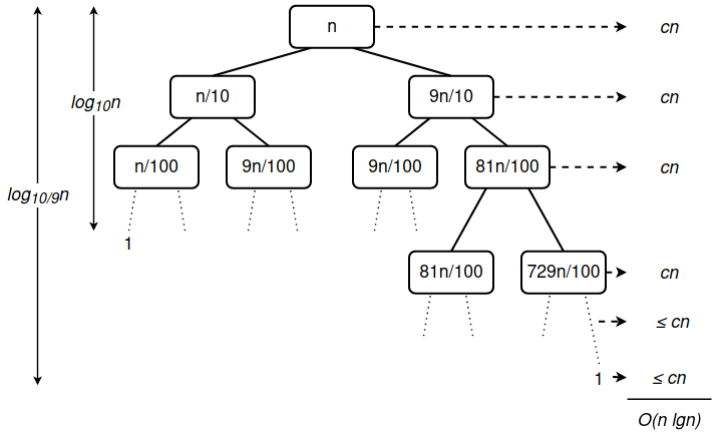


- **Terminazione**, alla fine del ciclo $j = dx$ allora pertanto ogni posizione dell'array si trova in uno dei tre insieme descritti dall'invariante

5.1.1 Prestazioni



La procedura *PARTITION* costa $\Theta(n)$, mentre, il tempo di esecuzione di Quick sort dipende da chi si sceglie come pivot: il caso migliore è quando il pivot divide a metà l'array e quindi l'equazione di ricorrenza corrisponde a $T(n) = 2T(n/2) + \Theta(n)$. Applicando il *master theorem* si trova come risultato $\Theta(nlogn)$, quindi se il partizionamento è bilanciato, l'algoritmo viene eseguito con la stessa velocità asintotica del Merge sort. Il caso peggiore si ha quando il pivot è il minimo o il massimo degli elementi dell'intervallo che si vuole ordinare. Per esempio sull'array {1, 2, 3, 4, 5, 6} se il pivot è 1, il Quick sort partiziona con 0 elementi a sinistra e $n-1$ a destra. Il secondo giro di *PARTITION* restituisce 2 quindi ci sarà una chiamata ricorsiva a sinistra di 0 elementi e $n-2$ elementi a destra, e così via. Se si immagina l'albero di ricorsione sarà tutto sbilanciato a destra. L'equazione di ricorrenza equivale a $T(n) = T(n-1) + T(0) + \Theta(n)$. Applicando il metodo della sostituzione si ottiene $T(n) = \Theta(n^2)$ allora se il partizionamento è sbilanciato il Quick sort è asintoticamente lento quanto l'Insertion sort. Inoltre il tempo di esecuzione $\Theta(n^2)$ si ha quando l'array di input è già completamente ordinato, una situazione in cui l'Insertion sort è eseguito nel tempo $O(n)$. Il tempo di esecuzione nel caso medio di Quick sort è molto più vicino al caso migliore che al caso peggiore. Si suppone che l'algoritmo di partizionamento produca sempre una ripartizione proporzionale 9 a 1, in questo caso si ottiene la ricorrenza $T(n) \leq T(9n/10) + T(n/10) + cn$. Ogni livello dell'albero ha un costo cn, finché non viene raggiunta una condizione al contorno alla profondità $log_1 0n = \Theta(lgn)$ dopo la quale i livelli hanno al massimo un costo cn. La ricorsione termina alla profondità $log_{10/9} n = \Theta(lgn)$. Il costo totale è dunque $O(nlgn)$. Pertanto con una ripartizione che intuitivamente sembrava molto sbilanciata, il tempo di esecuzione è asintoticamente uguale a quello che si ha nel caso di ripartizione esattamente a metà. La ragione è che qualsiasi ripartizione con proporzionalità costante produce un albero di ricorsione di profondità $\Theta(lgn)$ dove il costo in ogni livello è $O(n)$ quindi il tempo di esecuzione è $O(nlgn)$.



5.1.2 Versione randomizzata

Quando si esegue il Quick sort su un array di input casuale è piuttosto improbabile che i dati vengano partizionati sempre nel modo peggiore ad ogni chiamata ricorsiva. In media qualche ripartizione sarà ben bilanciata e qualche altra sarà molto sbilanciata. Per cercare di evitare sempre il caso meno buono si usa un metodo di **randomizzazione** dove anziché utilizzare $a[dx]$ come pivot si utilizza un elemento scelto a caso dal sottoarray $a[sx...dx]$. Per fare ciò si scambia l'elemento $a[dx]$ con il pivot scelto a caso. Poiché il pivot viene scelto a caso, la ripartizione dell'array di input è ben bilanciata in media.

Algorithm 10: costo $\Theta(1)$

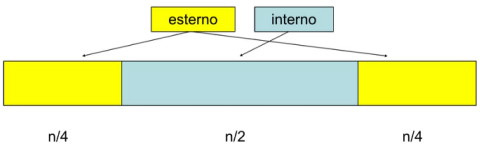
```
1 RANDOMIZED-PARTITION(a, sx, dx)
2 i ← RANDOM(sx, dx);
3 swap a[i] with a[dx];
4 return PARTITION(a, p, r)
```

Algorithm 11: costo caso medio $O(nlgn)$, caso peggiore $\Theta(n^2)$

```
1 RANDOMIZED-QUICKSORT(a, sx, dx)
2 if sx < dx then
3     px = RANDOMIZED − PARTITION(a, sx, dx);
4     RANDOMIZED − QUICKSORT(a, sx, px − 1);
5     RANDOMIZED − QUICKSORT(a, px + 1, dx);
```

Si definiscono due eventi con probabilità 1/2 ciascuno:

- Pivot esterno
- Pivot interno



si denomina con $T(n)$ il **costo medio** randomizzato e si definisce:

- A = $T(n)$ quando il pivot è esterno
- B = $T(n)$ quando il pivot è interno

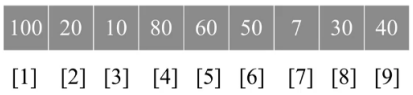
La situazione peggiore per A si verifica quando il pivot capita agli estremi della zona esterna: $T(n) = T(n-1)+O(n)$, cioè la complessità di prima nel caso peggiore. Stesso ragionamento per B, cioè la situazione più sfortunata è quando il pivot è agli estremi della zona interna: $T(n) = T(n/4) + T(3/4n) + O(n)$. Si sommano i due tempi

$$
\begin{aligned}
T(n) &\leq \frac{A}{2} + \frac{B}{2} \\
T(n) &= \frac{T(n-1) + T(n/4) + T(3/4n) + O(n)}{2} \\
2T(n) &\leq T(n) + T(n/4) + T(3/4n) + O(n) \\
T(n) &\leq T(n/4) + T(3/4n) + O(n)
\end{aligned}
$$

che è come nell'esempio del partizionamento 9 a 1, quindi si arriva alla stessa conclusione $O(nlogn)$. Nella pratica il Quick sort viene sempre utilizzata in forma randomizzata, è più efficiente e più facile da implementare rispetto al Merge sort perché quest'ultimo richiede memoria aggiuntiva. È presente in molte librerie.

5.2 Problema della selezione

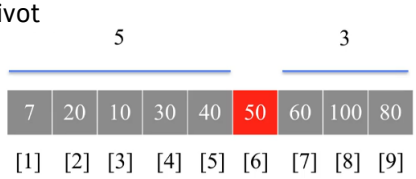
Sia dato un insieme a di n interi distinti, e un intero i, con $1 \leq i \leq n$, determinare l'elemento x che è più grande esattamente di $i-1$ altri elementi di a.



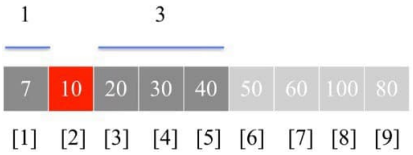
Per esempio con $i = 4$, l'output sarebbe 30 perché è esattamente maggiore di $\{20, 10, 7\}$. La soluzione banale per risolvere questo problema sarebbe quello di ordinare l'array ma l'ordinamento costa $O(nlogn)$, si vuole cercare qualcosa con un costo minore. L'approccio che si può usare è molto simile al Quick sort:

1. si sceglie un pivot a caso
2. si spostano gli elementi minori del pivot a sinistra e gli elementi maggiori a destra
3. a questo punto si conosce la cardinalità degli elementi a sinistra e la cardinalità di quelli a destra, e quindi si decida in quale parte fare la chiamata ricorsiva

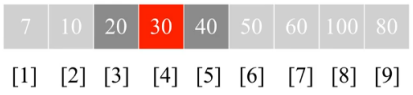
Sempre nel caso $i = 4$ e con 50 come pivot



il quarto elemento più grande non può stare a destra perché ci sono solo 3 elementi quindi si può escludere la partizione di destra e fare la chiamata ricorsiva solo a sinistra. A questo giro il pivot 10, si ripartiziona



non può essere a sinistra, ma non si sta più cercando il quarto elemento nella porzione di array che resta quindi ora i non può più essere 4 ma diventa $i = i - \#elementi\ a\ sx + 1$. Si esegue un nuovo giro con pivot 30 e si è finito perché prima del pivot si ha esattamente $i - 1 = 1$ elementi.



Il problema della selezione diventa una applicazione parziale del Quick sort

Algorithm 12: costo $\Theta(n)$

```
1  RAND-SELECT(a, sx, dx, i)
2  if sx = dx then
3      return a[sx];
4  px ← RANDOMIZED − PARTITION(a, sx, dx);
5  k ← px − sx + 1;
6  if i = k then
7      return a[px]
8  else if i < k then
9      return RANDOMIZED − SELECT(a, sx, px − 1, i)
10 else
11     return RANDOMIZED − SELECT(a, px + 1, dx, i − k)
```

Algorithm 13: costo $\Theta(1)$

```
1  RANDOMIZED-PARTITION(a, sx, dx)
2  i ← RANDOM(sx, dx);
3  swap a[i] with a[dx];
4  return PARTITION(a, p, r)
```

Come prima si immaginano due eventi: pivot interno e pivot esterno, e due diversi costi A e B. Per A il caso peggiore rimane quando è all'estremità della zona esterna cioè $T(n) = T(n-1)+O(n)$. Per B sparisce il termine $T(n/4)$ perché la ricorsione vi è solo in un lato, quindi $T(n) = T(3/4n)+O(n)$. Sommando i tempi si arriva a $T(n) \leq T(3/4n)+O(n)$ e quindi a $\Theta(n)$

Chapter 6

Moltiplicazione bit model, egizia, e di matrici

6.1 Moltiplicazione bit model

Se si deve calcolare $P = A \cdot B$ nel modello RAM (*word model*) ha costo costante. Il modello RAM parte dal presupposto che tutte le celle di memoria siano abbastanza grandi da contenere tutti gli operandi di cui ha bisogno. Se questa ipotesi non è vera, cioè si è in un ambiente in cui non è possibile effettuare la moltiplicazione in tempo costante, si va nel **bit model** dove si devono valutare le operazioni in funzione del numero delle cifre degli argomenti che partecipano alle operazioni. Si immagina di avere due interi da moltiplicare A e B:

			3	8	5	6	·
			1	3	4	5	=
		1	9	2	8	0	+
	1	5	4	2	4		+
1	1	5	6	8			+
3	8	5	6				=
5	1	8	6	3	2	0	

Il calcolo del prodotto di due numeri da n cifre A e B richiede $\Theta(n^2)$ operazioni. Si tenta di ridurre il numero di operazioni applicando il metodo divide et impera. Si suddividono i due numeri in due parti:

- $A = A_1 * 10^{n/2} + A_2$ nel caso precedente 3856 = 38 $*$ 10^2 + 56
- $B = B_1 * 10^{n/2} + B_2$ nel caso precedente 1345 = 13 $*$ 10^2 + 45

cioè si è suddiviso il numero di cifre degli elementi da moltiplicare in due parti da $n/2$ cifre ciascuna. Si esegue la moltiplicazione:

$$\begin{aligned} A * B &= (A_1 * 10^{n/2} + A_2) * (B_1 * 10^{n/2} + B_2) \\ &= A_1B_1 * 10^n + (A_1B_2 + A_2B_1) * 10^{n/2} + A_2B_2 \end{aligned}$$

Questo restituisce un algoritmo di tipo divide et impera:

Algorithm 14: costo $\Theta(n^2)$

```
1 MOLTRAPIDA4(A, B, n)
2 if n = 1 then
3     return A * B
4 x ← MOLTRAPIDA(A_1, B_1, n/2);
5 y ← MOLTRAPIDA(A_2, B_2, n/2);
6 z ← MOLTRAPIDA(A_1, B_1, n/2) + MOLTRAPIDA(A_2, B_2, n/2);
7 return x * 10^2 + z * 10^{n/2} + y;
```

L'equazione di ricorrenza equivale a:

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 4T(n/2) + \Theta(n) & se\ n > 1 \end{cases}$$

dove $\Theta(n)$ è il numero massimo di cifre del prodotto. Si utilizza il master thereom per risolvere l'equazione:

a =	4
b =	2
f(n) =	n

Si deve fare il confronto fra n e $n^{log_2 4} = n^2$
Si prova il primo caso $f(n) = O(n^{2-\epsilon})$ con $\epsilon = 1 \Rightarrow \Theta(n^2)$

Per calcolare il prodotto

$$A * B \quad = \quad A_1B_1 \cdot 10^n + (A_1B_2 + A_2B_1) * 10^{n/2} + A_2B_2$$

si considera questa formula

$$(A_1 + A_2) * (B_1 + B_2) \quad = \quad A_1B_1 + A_2B_1 + A_1B_2 + A_2B_2$$

che la si può utilizzare per calcolare la parte centrale del prodotto

$$\begin{aligned} A * B \quad &= \quad A_1B_1 * 10^n + (A_1B_2 + A_2B_1) * 10^{n/2} + A_2B_2 \\ &= \quad A_1B_1 * 10^n + ((A_1 + A_2) * (B_1 + B_2) - A_1B_1 - A_2B_2)10^{n/2} + A_2B_2 \end{aligned}$$

ora si hanno solo 3 moltiplicazioni di numeri da $n/2$ cifre. L'algoritmo diventa

Algorithm 15: costo $\Theta(n^{1.79})$

```
1 MOLTRAPIDA3(A, B, n)
2 if n = 1 then
3     return A * B
4 x ← MOLTRAPIDA(A_1, B_1, n/2);
5 y ← MOLTRAPIDA(A_2, B_2, n/2);
6 z ← MOLTRAPIDA(A_1 + A_2, B_1 + B_2, n/2) − x − y;
7 return x * 10^2 + z * 10^{n/2} + y;
```

allora l'equazione di ricorrenza corrisponde a

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 3T(n/2) + \Theta(n) & se\ n > 1 \end{cases}$$

si risolve anch'esso con il master theorem

a =	3
b =	2
f(n) =	n

Si deve fare il confronto fra n e $n^{log_2 3} \approx n^{1.79}$
Si prova il primo caso $f(n) = O(n^{2-\epsilon})$ con $\epsilon = 0.79 \Rightarrow \Theta(n^{1.79})$

Con il divide et impera si è sceso per pochi decimali da $\Theta(n^2)$ a $\Theta(n^{1.79})$

6.2 Moltiplicazione egizia

La moltiplicazione egizia è una operazione trovata all'interno del papiro di Ahmes nel 1700 a.C. e funziona in questo modo: si suppone di avere sempre due numeri interi A e B da n cifre con A $\geq$ B

Algorithm 16: costo $\Theta(n^2)$

```
1 MOLTEGIZIA(A, B)
2 P ← 0;
3 while A > 0 do
4     if A is odd then
5         P ← P + B
6     A ← A/2;
7     B ← B * 2;
8 return P;
```

Si prova con un esempio con $A = 21$ e $B = 13$

A	B	P
21	13	0 + 13
10	26	13
5	52	13 + 52
2	104	65
1	208	65 + 208
0	-	273

L'algoritmo si basa sulla seguente riflessione: se si vuole moltiplica $A \cdot B$ nel caso in cui A è pari si può considerare $A/2 \cdot 2B$. Nel caso in cui A è dispari si calcola $A/2 \cdot 2B + B$. La complessità è data dal ciclo while, si suppone che A e B sono composte da n cifre binarie: ogni volta che si divide per 2 un numero binario si elimina una cifra e siccome si esegue un while finché $A > 0$, il ciclo viene eseguito n volte. La divisione e la moltiplicazione per 2 costano ognuno $\Theta(1)$ perché in binario basta rispettivamente rimuovere la cifra più significativa, e aggiungere uno zero a destra. La somma consiste nel considerare tutte le cifre dei due operandi, e nel caso anche un riporto, quindi la somma costa $\Theta(n)$. In tutto la funzione costa $\Theta(n^2)$

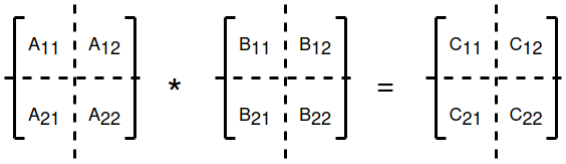
6.3 Moltiplicazioni di matrici

Si ritorna al modello RAM. Questo algoritmo viene molto usato nella grafica ed è molto importante dal punto di vista pratico. Si suppone di avere due matrici quadrate A e B

$$A = \begin{cases} a_{11} & a_{12} & \dots & a_{1n} \\ \boldsymbol{a_{21}} & \boldsymbol{a_{22}} & \boldsymbol{\dots} & \boldsymbol{a_{2n}} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{cases} \qquad B = \begin{cases} b_{11} & \boldsymbol{b_{12}} & \dots & b_{1n} \\ b_{21} & \boldsymbol{b_{22}} & \dots & b_{2n} \\ \dots & \boldsymbol{\dots} & \dots & \dots \\ b_{n1} & \boldsymbol{b_{n2}} & \dots & b_{nn} \end{cases}$$

$$C = A * B = \begin{cases} c_{11} & \dots & c_{1n} \\ \dots & c_{ij} & \dots \\ c_{n1} & \dots & c_{nn} \end{cases}$$

L'elemento generico c_{ij} della matrice risultate si ottiene con il prodotto "righe - colonne": $\sum_{k=1}^{n} a_{ik} * b_{kj}$
Si studia quanto costa calcolare c: esso è composto da n^2 elementi, per ciascun elemento si devono fare n moltiplicazioni e in seguito un certo numero di somme che sono circa n^2. In totale si devono fare n^3 moltiplicazioni e n^2 somme. Si controlla se anche in questo caso conviene usare un metodo divide et impera: si prende la matrice A e la si divide in 4 sottomatrici di nome $A_{11}, A_{12}, A_{21}, A_{22}$ di dimensioni $n/2 * n/2$. Stessa procedimento per B.



Dove le sottomatrici C sono state ottenute in questo modo:

- $C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$
- $C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$
- $C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$
- $C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$

Queste scomposizioni danno l'algoritmo di divide et impera, invece di andare calcolare direttamente tutti i j e i della matrice C, si calcola ricorsivamente

Algorithm 17: costo $\Theta(n^2)$

```
1 MOLTMAT(A, B, n)
2 if n = 1 then
3     return c ← a_11 + b_11
4 else
5     C_11 ← MOLTMAT(A_11, B_11, n/2) + MOLTMAT(A_12, B_21, n/2);
6     C_12 ← MOLTMAT(A_11, B_12, n/2) + MOLTMAT(A_12, B_22, n/2);
7     C_21 ← MOLTMAT(A_21, B_11, n/2) + MOLTMAT(A_22, B_21, n/2);
8     C_22 ← MOLTMAT(A_21, B_12, n/2) + MOLTMAT(A_22, B_22, n/2);
9 return C
```

La corrispondente equazione di ricorrenza è $T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 8T(n/2) + \Theta(n^2) & se\ n > 1 \end{cases}$
dove con n^2 elementi, la complessità della somma di due matrici è $\Theta(n^2)$. Si risolve l'equazione di ricorrenza: 1 caso del master theorem $f(n) = O(n^{3-\epsilon})$ con ϵ = 1 $\Rightarrow f(n) = \Theta(n^3)$. Anche con il divide et impera il costo non cambia.

Per migliorare la complessità si usa l'**algoritmo di Strassen** che si basa su queste considerazioni:

- $X_1 = (A_{11} + A_{21})(B_{11} + B_{22})$
- $X_2 = (A_{21} + A_{22})B_{11}$
- $X_3 = A_{11}(B_{12} - B_{22})$
- $X_4 = A_{22}(B_{21} - B_{11})$
- $X_5 = (A_{11} + A_{12})B_{22}$
- $X_6 = (A_{21} - A_{11})(B_{11} - B_{12})$
- $X_7 = (A_{12} - A_{21})(B_{21} + B_{22})$

- $C_{11} = X_1 + X_4 - X_5 - X_7$
- $C_{12} = X_3 + X_4$
- $C_{21} = X_2 + X_4$
- $C_{22} = X_1 + X_3 - X_2 + X_6$

I valori di X si ottengono con 7 prodotti $n/2 * n/2$, mentre i valori di C si ricavano con 8 somme di matrici. L'equazione di ricorrenza è

$$T(n) = \begin{cases} \Theta(1) & se\ n = 1 \\ 7(n/2) + \Theta(n^2) & se\ n > 1 \end{cases}$$

a =	7
b =	2
f(n) =	n^2

Si deve fare il confronto fra n^2 e $n^{log_2 7} \approx n^{2.81}$
Si prova il primo caso $f(n) = O(n^{2.81-\epsilon})$ con $\epsilon = 0.81 \Rightarrow \Theta(n^{2.81})$

Altri ricercatori sono riusciti ad arrivare a $n^{2.4}$. Non si sa quanto si potrebbe scendere di complessità perché nessuno ha stabilito un limite inferiore a questo problema.

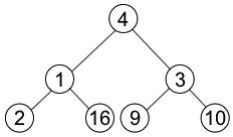
Chapter 7

Struttura dati heap

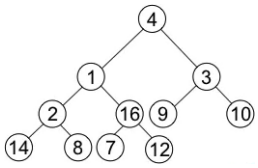
7.1 Alberi, terminologia e proprietà

Un albero è un grafo connesso, ovvero preso qualsiasi coppia di nodi esiste un cammino composto da archi che connette i due nodi, ed è **privo di cicli**. Se l'albero ha n nodi allora ha $n-1$ archi. Alberi speciali:

- **Albero binario pieno**: un albero binario in cui tutte le foglie sono allo stesso livello e tutti i nodi interni hanno grado 2. Possiede tutti i nodi possibili.

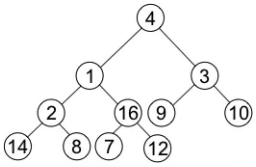


- **Albero binario completo**: un albero binario in cui ogni nodo è una foglia o ha grado esattamente 2. Tutti i livelli dell'albero sono completamente riempiti tranne l'ultimo che può essere riempito da sinistra fino a un certo punto.



Terminologia:

- **Altezza di un nodo**, numero di archi sul più lungo cammino semplice dal nodo a una foglia
- **Livello di un nodo** (o profondità), è la distanza dalla radice, espressa come numero di archi di cui è composto il cammino dalla radice al nodo. La radice è al livello zero
- **Altezza di un albero**, altezza della radice



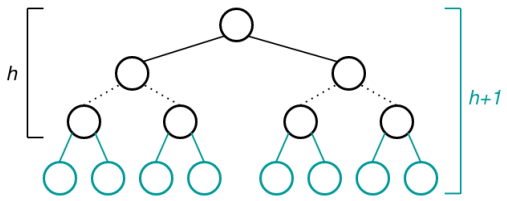
Altezza albero = 3
Altezza di (2) = 1
Livello di (10) = 2

Proprietà:

- Ci sono *al più* 2^l nodi al livello l di un albero binario
- Un albero binario di altezza h ha $N_h = 2^{h+1} - 1$ nodi, di cui $I_h = 2^h - 1$ nodi interni e $F_h = 2^h$ foglie.

  Caso base: $h = 0$, l'albero è composto da un $N_0 = 2^{0+1} - 1$ nodi, da $I_0 = 2^0 - 1$ nodi interni e$F_0 = 2^0$ foglie
  Passo induttivo: si assume che il teorema sia vero per h e si dimostra per $h+1$

$$\begin{aligned} N_{h+1} &= N_h + 2 * F_h \\ &= 2^{h+1} - 1 + 2 * 2^h \\ &= 2^{h+1} - 1 + 2^{h+1} \\ &= 2^{h+2} - 1 \end{aligned}$$



  di conseguenza $I_{h+1} = N_h = 2^{h+1} - 1$ e $F_{h+1} = 2F_h = 2^{h+1}$

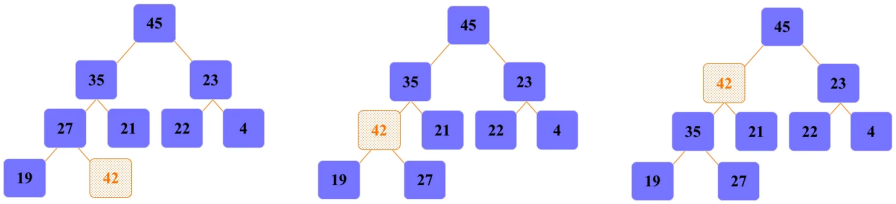
- Un albero binario con n nodi ha altezza $h = log(n+1) - 1 = O(lgn)$

$$n = 2^{h+1} - 1 \quad \Rightarrow \quad n+1 = 2^{h+1} \quad \Rightarrow \quad log(n+1) = h+1 \quad \Rightarrow \quad h = log(n+1) - 1$$

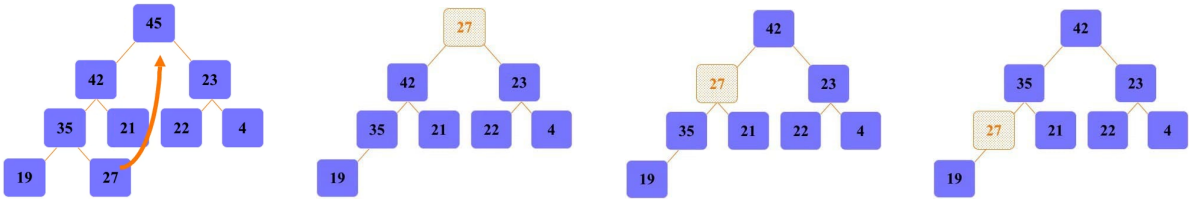
7.2 Heap

Un **heap** è una struttura dati composta da un array che si può considerare come un albero binario completo. Ogni nodo dell'albero corrisponde a un elemento dell'array. Per costruire un albero binario completo si parte dalla radice e poi si aggiungono i nodi negli altri livelli partendo da sinistra. La **proprietà di max-heap**: richiede che la chiave (valore) di ogni nodo sia maggiore o uguale della chiave dei suoi figli. La radice è il massimo elemento.

- **Aggiungere un nodo a uno heap**, si inserisce il nodo come foglia, mantenendo la struttura completa. Si spinge il nodo in alto, effettuando swap con il padre finché è necessario, cioè il nodo si ferma quando il padre è più grande o quando si è alla radice. Questo processo si chiama **reheapification upward**.

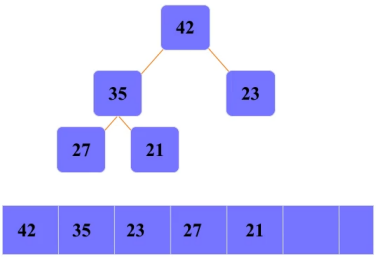


- **Rimuovere la radice di uno heap**, si elimina la radice, e si scambia con una foglia (ultimo nodo) al suo posto. Si spinge il nodo fuori posto verso il basso, scambiandolo con il suo figlio più grande finché è necessario, cioè il nodo si ferma quando è maggiore o uguale a tutti i figli o quando il nodo diventa una foglia. Questo processo si chiama **reheapification downward**.



Insieme alla proprietà di max-heap esiste l'organizzazione opposta ovvero la **proprietà di min-heap** che richiede che la chiave di ogni nodo sia minore o uguale alla chiave dei suoi figli. La radice è il minimo elemento.

7.2.1 Implementare uno heap

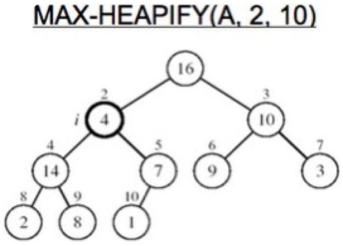


Uno heap può essere rappresentato con un array parzialmente riempito. L'unica differenza tra implementare uno heap come array o come albero è che il primo ad un certo punto si riempirà e quindi si dovranno effettuare delle operazioni per fare in modo che l'array si allarghi. La radice dell'albero va nella prima allocazione dell'albero. I dati al successivo livello vanno nelle successive locazioni. Riempire l'array per livello permette di sapere dato un nodo qual è la cella dell'array che rappresenta il padre o il figlio. I collegamenti tra nodi non sono memorizzati cioè non si usano puntatori e si sa che l'array è un albero solo per il modo in cui si manipolano i dati. Sapendo che l'array parte da 1 per conoscere l'indice del padre di 21 basta fare l'indice di quest'ultimo diviso 2 ovvero $5/2 = 2 \Rightarrow a[2] = 35$. In generale il padre di $a[i]$ è $a[i/2]$. Dualmente i figli di un nodo $a[i]$ si troveranno alla posizione $a[2i]$ e $a[2i+1]$.

7.2.2 Operazioni su uno heap

Le operazioni che si andranno a sviluppare sono

- **MAX-HEAPIFY**, conserva la proprietà di max-heap



MAX-HEAPIFY è una procedura in cui mantiene la proprietà di max-heap sistemando un elemento fuori posto. Quando viene chiamata, MAX-HEAPIFY si assume che gli alberi binari con radici in *LEFT(i)* e *RIGHT(i)* siano max-heap, ma che $a[i]$ possa essere più piccolo dei suoi figli violando così la proprietà del max-heap. MAX-HEAPIFY fa "scendere" il valore $a[i]$ nel max-heap in modo che il sottoalbero con radice di indice i diventi max-heap.

Algorithm 18: costo $O(lgn)$

```
1  MAX - HEAPIFY(a, i, n)
2  l ← LEFT(i);
3  r ← RIGHT(i);
4  if l ≤ n and a[l] > a[i] then
5      largest ← l
6  else
7      largest ← i
8  if r ≤ n and a[r] > a[largest] then
9      largest ← r
10 if largest ≠ i then
11     swap a[i] with a[largest];
12     MAX − HEAPIFY(a, largest, n)
```

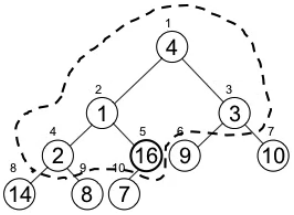
A ogni passo viene determinato il più grande tra gli elementi $a[i]$, $a[LEFT(i)]$ e $a[RIGHT(i)]$, e il suo indice viene memorizzato in *grande*. Se $a[i]$ è più grande, allora il sottoalbero con radice nel nodo i è un max-heap e la procedura termina. Altrimenti, uno dei due figli ha l'elemento più grande e $a[i]$ viene scambiato con $a[grande]$, in questo modo, il nodo i e i suoi figli soddisfano la proprietà del max-heap. Il nodo con indice *grande*, però, adesso ha il valore originale $a[i]$ e, quindi, il sottoalbero con radice in *grande* potrebbe violare la proprietà del max-heap. Di conseguenza deve essere chiamata ricorsivamente la subroutine MAX-HEAPIFY per questo sottoalbero. Il tempo di esecuzione di MAX-HEAPIFY in un sottoalbero di dimensione n con radice in un nodo i è pari al tempo $\Theta(1)$ per sistemare le relazioni fra gli elementi $a[i], a[LEFT(i)]$, e $a[RIGHT(i)]$, più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del nodo i. Se l'albero fosse pieno, cioè si effettuano le chiamate ricorsive a sinistra e a destra esattamente lo stesso numero di nodi l'equazione di ricorrenza equivale ad $T(n) = T(n/2) + \Theta(1)$. Il caso peggiore è quando a sinistra si ha un livello in più rispetto alla parte di destra ovvero l'ultima riga dell'albero è piena a metà allora il tempo di esecuzione di MAX-HEAPIFY può essere descritto da questa ricorrenza $T(n) = T(2n/3) + \Theta(1)$ che per il caso 2 del master theorem è $T(n) = O(lgn)$

- **BUILD-MAX-HEAP**, crea un max-heap da un array non ordinato

  Ciascuna foglia dell'albero sono già degli heap che verificano la proprietà di max-heap. La procedura BUILD-MAX-HEAP attraversa i restanti nodi dell'albero ed esegue MAX-HEAPIFY in ciascuno di essi.

Algorithm 19: costo $O(n)$

```
1  BUILD - MAX - HEAP(a)
2  n ← lenght[a];
3  for i ← n/2 downto 1 do
4      MAX − HEAPIFY(a, i, n)
```

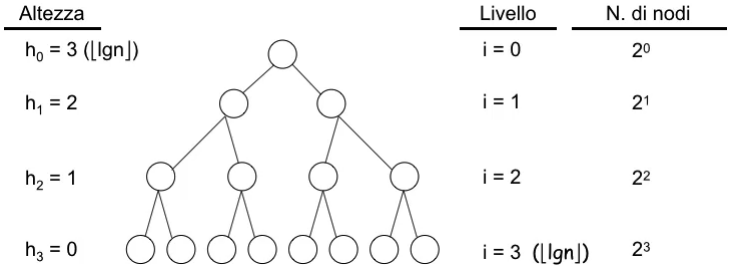


L'indice di metà rappresenta l'ultimo nodo che non è una foglia. Per verificare che BUILD - MAX - HEAP funziona correttamente si utilizza la seguente invariante di ciclo:

Invariante di ciclo: all'inizio di ogni iterazione del ciclo for, ogni nodo $i+1, i+2, ..., n$ è la radice di un max-heap.

- **Inizializzazione**, prima della prima iterazione del ciclo, $i = n/2$. Ogni nodo $n/2 + 1, n/2 + 2, ..., n$ è una foglia e, quindi, è la radice di un max-heap banale
- **Conservazione**, i figli del nodo i hanno una numerazione più alta di i. Per l'invariante di ciclo, quindi, essi sono entrambi radici di max-heap. Questa è esattamente la condizione richiesta affinchè la chiamata $MAX - HEAPIFY(a, i, n)$ renda il nodo i la radice di un max-heap. Inoltre, la chiamata MAX-HEAPIFY preserva la proprietà che tutti i nodi $i + 1, i + 2, ...n$ siano radici di max-heap. La diminuzione di i nell'aggiornamento del ciclo for ristabilisce l'invariante di ciclo per la successiva iterazione
- **Terminazione**, alla fine del ciclo $i = 0$. Per l'invariante di ciclo, ogni nodo $1, 2, ..., n$ è la radice di un max-heap, in particolare, lo è il nodo 1

Ogni chiamata di MAX-HEAPIFY costa un tempo $O(lgn)$ e ci sono $O(n)$ di queste chiamate. Quindi il tempo di esecuzione è $O(nlgn)$. Questo limite superiore, sebbene sia corretto, non è asintoticamente stretto perché il costo di MAX-HEAPIFY non è sempre n ma è proporzionale all'altezza in cui si sta chiamando la procedura. Il tempo richiesto dalla procedura MAX-HEAPIFY quando viene chiamata per un nodo di altezza h è $O(h)$



Il numero di nodi al livello i è 2^i, mentre l'altezza heap radicata al livello i è $h-1$. Quindi il tempo sommando il costo di eseguire MAX-HEAPIFY su ogni livello è $\sum_{i=0}^{h} n_i h_i$ dove $n_i h_i$ è il prodotto del numero di nodi che si ha ad un certo livello e il costo di MAX-HEAPIFY per quel livello

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{h} n_i h_i && \text{si sostituiscono i valori di } n_i \text{ e } h_i \\
&= \sum_{i=0}^{h} 2^i (h-i) && \text{moltiplica per } 2^h \text{ numeratore e denominatore, e scrivi } 2^i \text{ come } \frac{1}{2^{-i}} \\
&= \sum_{i=0}^{h} \frac{h-i}{2^h - i} 2^h && \text{si fa un cambio di variabile } k = h - i \\
&= 2^k \sum_{k=0}^{h} \frac{k}{2^k} && \text{si può limitare con la sommatoria che va da 0 a infinito perché essa è una serie nota} \\
&\leq n \sum_{k=0}^{\infty} \frac{k}{2^k} && \text{la sommatoria è al più 2} \\
&= O(n)
\end{aligned}
$$

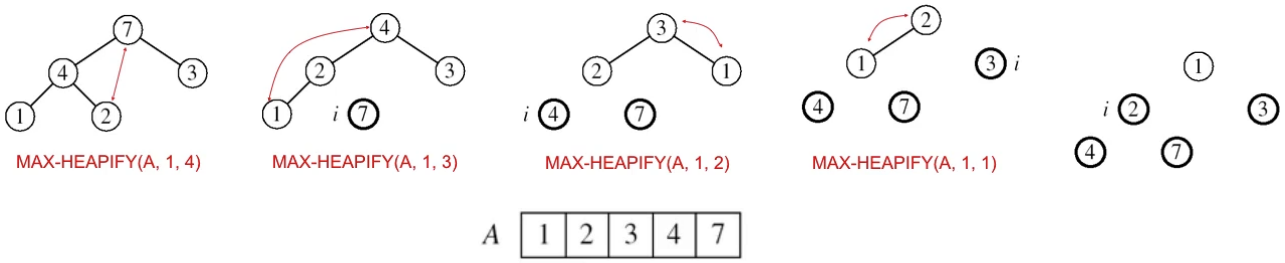
- **Heap sort**, ordina un array in place

  L'algoritmo Heap sort inizia utilizzando BUILD-MAX-HEAP per costruire un max-heap nell'array di input $a[1...n]$. Poiché l'elemento più grande dell'array è memorizzato nella radice $a[1]$ lo si scambia con l'ultimo cioè $a[n]$. Se adesso si rimuove il nodo n dall'heap, decrementando la dimensione dello heap, si nota che i figli della radice restano max-heap, ma la nuova radice potrebbe violare la proprietà del max-heap. Per ripristinare questa proprietà basta una chiamata di MAX-HEAPIFY che lascia un max-heap in $a[1...(n-1)]$. L'algoritmo Heap sort ripete questo processo per il max-heap di dimensione $n-1$ e così via fino a un heap di dimensione 2.

Algorithm 20: costo $O(nlgn)$

```
1 HEAP-SORT(a)
2 BUILD − MAX − HEAP(a);
3 for i ← a.length downto 2 do
4     swap a[1] with a[i];
5     MAX − HEAPIFY(a, 1, i − 1)
```

La procedura Heap sort impiega un tempo $O(nlgn)$, in quanto la chiamata di BUILD-MAX-HEAP impiega $O(n)$ e ciascuna delle $n-1$ chiamate di MAX-HEAPIFY impiega un tempo $O(lgn)$



A	1	2	3	4	7

7.3 Code di priorità

Una coda di priorità è una struttura dati dinamica che serve a mantenere un insieme S di elementi. Una coda di priorità possiede le seguenti proprietà:

- ogni elemento è associato ad un valore (priorità)
- la chiave con la più alta (o bassa) priorità è estratta prima

Le principali operazioni sono:

- rimuovere un elemento dalla coda
- inserire un elemento in coda

devono essere svolte facendo attenzione di conservare sempre ad ogni operazione le proprietà. Analogamente agli heap, esistono due tipi di code di priorità: **code di max-priorità** e **code di min-priorità**. Le code a priorità massima supportano le seguenti operazioni:

- $INSERT(S, x)$, inserisce l'elemento x nell'insieme S
- $MAXIMUM(S)$, restituisce l'elemento di S con la chiave più grande
- $EXTRACT - MAX(S)$, rimuove e restituisce l'elemento S con la chiave più grande
- $INCREASE - KEY(S, x, k)$, imposta a k il valore della chiave dell'elemento x assumendo che k è maggiore del valore corrente della chiave di x

Si immagina di implementare la coda con una lista collegata e si analizzano i costi:

$EXTRACT - MAX(S)$	$O(1)$	si estrae dalla testa
$INCREASE - KEY(S, x, k)$	$O(n)$	per incrementare una chiave e poi sistemare la lista
$INSERT(S, x)$	$O(n)$	si deve scandire la lista per trovare il punto giusto
$MAXIMUM(S)$	$O(1)$	è il la testa della coda

Si può utilizzare uno heap per implementare una coda di priorità con i seguenti costi:

- $HEAP - EXTRACT - MAX(S)$, scambia la radice con l'ultimo, decrementa la dimensione dello heap e invoca MAX-HEAPIFY sulla nuova radice

Algorithm 21: costo $O(nlgn)$

```
1 HEAP-EXTRACT-MAX(a, n)
2 if n < 1 then
3     error "underflow dell'heap"
4 max ← a[1];
5 a[1] ← a[n];
6 n ← n − 1;
7 MAX − HEAPIFY(a, 1, n);
8 return max
```

- $HEAP - INCREASE - KEY(S, x, k)$, imposta la chiave $a[i]$ al nuovo valore, se la proprietà max-heap non è verificata, naviga verso la radice per trovare il punto giusto per la nuova chiave. Il padre di $a[i]$ è $a[PARENT(i)]$

Algorithm 22: costo $O(nlgn)$

```
1 HEAP-INCREASE-KEY(a, key, i)
2 if key < a[i] then
3     error "la nuova chiave è più piccola di quella corrente
4 a[i] ← key;
5 while i > 1 and a[PARENT(i)] < a[i] do
6     swap a[i] with a[PARENT(i)];
7     i ← PARENT(i);
```

- $MAX - HEAP - INSERT(S, x)$, espande il max-heap aggiungendo una nuova foglia la cui chiave è $-\infty$, poi chiama HEAP-INCREASE-KEY per impostare la chiave di questo nodo al suo valore corretto e mantenere la proprietà del max-heap

Algorithm 23: costo $O(nlgn)$

```
1 MAX-HEAP-INSERT(a, key, n)
2 n ← n + 1;
3 a[n] ← −∞;
4 HEAP − INCREASE − KEY(a, key, n)
```

- $HEAP - MAXIMUM(S)$, è la radice, quindi basta fare $return\ a[1]$ con relativo costo $O(1)$

Implementare una coda di priorità con una lista è più efficiente per quanto riguarda l'estrazione del massimo ma tutte le altre operazioni sono più efficienti con lo heap senza considerare il fatto che lo heap può essere implementato con un array e quindi evitando i puntatori, allocazione della memoria dinamica, ecc...

Chapter 8

Sorting in tempo lineare

8.1 Ordinamento senza confronti

Tutti gli algoritmi di ordinamento presentati finora sono ordinamenti per confronti. In un ordinamento per confronti si usano soltanto i confronti fra gli elementi per ottenere informazioni sull'ordinamento di una sequenza di input $\{a_1, a_2, ..., a_n\}$. Ovvero, dati due elementi a_i e a_j, si eseguono uno dei test $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i \geq a_j$ o $a_i > a_j$ per determinare il loro ordine relativo. Non si può assumere i valori degli elementi o ottenere informazioni sul loro ordine in altri modi. Si suppone senza perdere di generalità che tutti gli elementi di input sono distinti. Fatta questa ipotesi, i confronti della forma $a_i = a_j$ sono inutili, e inoltre i confronti $a_i \leq a_j, a_i \geq a_j, a_i > a_j$ e $a_i < a_j$ sono tutti equivalenti perché forniscono le stesse informazioni sull'ordine relativo di a_i e a_j. Quindi si suppone che tutti i confronti abbiano la forma $a_i \leq a_j$. Per ordinare senza confronti si sfruttano eventuali proprietà note a priori degli oggetti da ordinare. Per esempio si vogliono ordinare 10 interi che vanno da 1 a 10 senza ripetizioni, in questo caso non si usano i confronti ma in prima posizione si scrive 1, in seconda posizione si scrive 2, ... fino alla decima posizione dove si scriverà 10, quindi il costo sarà lineare. Gli algoritmi di **Counting sort** e **Radix sort** si basano sulle proprietà note degli oggetti da ordinare.

8.2 Counting sort

L'algoritmo Counting sort suppone che ciascuno degli n elementi di input sia un numero intero compreso nel range 1...k (con eventuali ripetizioni o non è detto che ci siano tutti) per qualche k, e si contano le occorrenze di ogni possibile valore. Per esempio nel range 1...5 con l'array $\{4, 5, 1, 1, 3, 2, 2, 4\}$ vi sono due 4, un 5, due 1, un 3 e due 2, e questi valori vengono salvati in un array di supporto. Come ultimo passaggio si scrive nell'array di output l'1 due volte, il 2 due volte ... e così via. Nel codice di Counting sort si suppone che l'array di input è $a[1...n]$, l'array di output è $b[1...n]$ e l'array $c[0...k]$ è l'array di supporto.

Algorithm 24: costo $\Theta(n)$

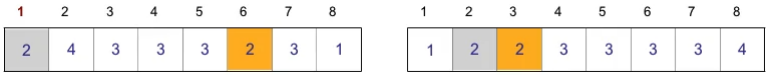
```
1  COUNTING-SORT(a, b, k)
2  for i ← 0 to k do
3      c[i] ← 0
4  for j ← 1 to a.lenght do
5      c[a[j]] ← c[a[j]] + 1                    // c[i] adesso contiene il numero di elementi uguali a i
6  for i ← 1 to k do
7      c[i] ← c[i] + c[i − 1]                   // c[i] adesso contiene il numero di elementi minori o uguali a i
8  for j ← a.lenght downto 1 do
9      b[c[a[j]]] ← a[j];
10     c[a[j]] ← c[a[j]] − 1
```

Dopo che il ciclo for (righe 2-3) inizializza a zero tutti gli elementi dell'array c, ogni elementi di input viene esaminato nelle righe 4-5. Se il valore di un elemento di input è i, si incrementa $c[i]$. Quindi, dopo la riga 5, $c[i]$ contiene il numero degli elementi input uguali a i per ogni intero $i = 0, ..., k$. Le righe 6-7 determinano, per ogni $i = 0, ..., k$ quanti elementi di input sono minori o uguali a i, mantenendo la somma corrente dell'array c. Infine le righe 8-10 inseriscono l'elemento $a[j]$ nella corretta posizione ordinata dell'array di output b. Se tutti gli elementi n elementi sono distinti, quando viene eseguita per la prima volta la riga 10, per ogni $a[j]$, il valore $c[a[j]]$ rappresenta la

posizione finale corretta di $a[j]$ nell'array di ouput, in quanto ci sono $c[a[j]]$ elementi minori o uguali a $a[j]$. Poiché gli elementi potrebbero non essere distinti, $c[a[j]]$ viene ridotto ogni volta che viene inserito un valore $a[j]]$ nell'array b. La riduzione di $c[j]]$ fa sì che il successivo elemento di input con valore uguale ad $a[j]$, se esiste, venga inserito nella posizione immediatamente prima di $a[j]$ nell'array di output.

8.2.1 Analisi Counting sort

I cicli for alle righe 2-3 e 7-8 impiegano un tempo $\Theta(k)$, e i cicli alle righe 4-5 e 8-10 impiegano un tempo $\Theta(n)$. Quindi, il totale è $\Theta(k+n)$. Di solito Counting sort viene utilizzato quando $k = O(n)$, nel qual caso il tempo di esecuzione è $\Theta(n)$. Un'importante proprietà di Counting sort è la **stabilità**: i numeri con lo stesso valore si presentano nell'array di output nello stesso ordine in cui si trovano nell'array di input. Ovvero, l'uguaglianza di due numeri viene risolta applicando la seguente regola: il numero che si presenta per primo nell'array di input sarà inserito per primo nell'array di output. Preservare l'ordine tra due numeri uguali è importante perché ad uno di essi si possono essere associate altre informazioni. Non tutti gli algoritmi di ordinamento sono stabili per esempio come l'Heap sort o Quick sort. La stabilità di Counting sort è importante anche perché spesso viene utilizzato come subroutine di Radix sort.

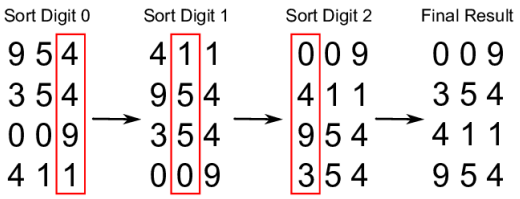


8.3 Radix sort

Il Radix sort ordina solo numeri interi si basa sull'idea di ordinare i numeri cifra per cifra. Il Radix sort ordina prima i numeri in base alla cifra *meno* significativa. I numeri vengono poi combinati in un unico insieme: i numeri del gruppo 0 precedono quelle del gruppo 1 che, a loro volta, procedono quelle del gruppo 2 e così via. Poi tutto l'insieme viene ordinato di nuovo in funzione della seconda cifra meno significativa e ricombinando in maniera analoga. Il processo continua fino a che tutti i numeri sono completamente ordinati rispetto al numeri di d cifre. Quindi, occorrono soltanto d passaggi attraverso l'insieme per completare l'ordinamento. È essenziale che gli ordinamenti delle cifre in questo algoritmo siano stabili. Il codice per Radix sort è semplice. La seguente procedura suppone che ogni elemento nell'array di a di n elementi abbia d cifre, dove la cifra 1 è quella di ordine più basso e la cifra d è quella di ordine più alto

Algorithm 25: costo $\Theta(n)$

```
1 RADIX-SORT(a, d)
2 for i ← 1to d do
3     usa un ordinamento stabile
4     per ordinare l'array a sulla cifra i
```



La correttezza di Radix sort si dimostra per induzione. Si assume che le cifre meno significative siano ordinate e si dimostra che ordinare sulla cifra i lascia l'array completamente ordinato fino alla cifra i:

- Se due cifre in posizione i sono differenti, ordinando su quelle cifre l'induzione è dimostrata. Le cifre meno significative sono irrilevanti
- Se sono uguali, i numeri sono già ordinati. Dato che si utilizza un sort stabile

L'analisi del tempo di esecuzione dipende dall'ordinamento stabile che viene utilizzato come algoritmo di ordinamento intermedio. La scelta ovvia è scegliere il Counting sort, quindi ogni passaggio su n numeri di d cifre richiede un tempo $\Theta(n+k)$, poiché ci sono d passaggi, il tempo totale di Radix sort è $\Theta(d(n+k))$

Chapter 9

Tabelle hash

9.1 Introduzione

Una compagnia telefonica vuole fornire Caller ID ai propri clienti: dato un numero di telefono, fornire il numero del chiamante:

- **Chiave**, numero di telefono
- **Elemento**, nome del chiamante

con l'assunzione che i numeri di telefono sono unici e compresi tra 0 e 10^7. Non tutti i numeri nell'intervallo sono effettivamente numeri di telefono. Un **dizionario** consiste in un insieme di coppie (chiave/elemento) dove la chiave è utilizzata per cercare l'elemento. Il dizionario può essere:

- **Dizionario ordinato**, elementi memorizzati ordinatamente rispetto alle chiave
- **Dizionario non ordinato**, elementi memorizzati senza un ordina particolare

Esistono diverse implementazioni di un dizionario:

	Insert	Search	Delete
Lista ordinata	$O(1)$	$O(n)$	$O(n)$
Alberi	$O(logn)$	$O(logn)$	$O(logn)$
Array ordinato	$O(n)$	$O(logn)$	$O(n)$
Array speciale chiave $\in \{1, ..., k\}$	$O(1)$	$O(1)$	$O(1)$

dove per *array speciale* si intende nel caso in cui si sapesse che le chiavi sono in un insieme limitato, allora, gli indici degli array sono le chiavi. Un array da 1 a k in cui la posizione i corrisponde alla chiave i. Un altro modo efficace per implementare i dizionari sono con le **tabelle hash**

9.2 Tabelle hash

La tabella hash generalizza il principio degli array speciali utilizzando le chiavi come indici della tabella. Le tabelle hash utilizzano:

- **Funzione hash**, è una funzione che prende in input una chiave e restituisce in output un indice che viene utilizzato per memorizzare la chiave all'interno della tabella.
- **Parametro load factor λ**, indica quanto è piena la tabella hash

Esistono due possibili metodologie di implementazione: tabelle a indirizzamento diretto e l'hashing.

9.2.1 Tabelle a indirizzamento diretto

L'indirizzamento diretto è una tecnica semplice che funziona bene quando l'universo delle chiave U è ragionevolmente piccolo. La tabella è un array grande quanto il numero di chiavi possibili e la funzione hash è la funzione identità dove l'indice i equivale alla chiave i. Le operazioni di dizionario sono semplici da implementare:

- Inserimento

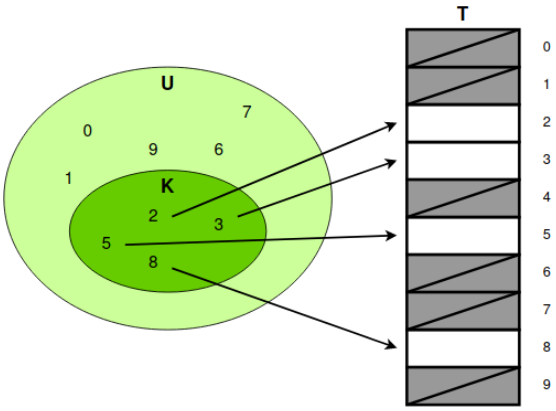
Algorithm 26: costo $O(1)$
1 DIRECT-ADDRESS-INSERT(T, x)
2 $T[x.key] \leftarrow x$

- Ricerca

Algorithm 27: costo $O(1)$
1 DIRECT-ADDRESS-SEARCH(T, k)
2 **return** $T[k]$

- Eliminazione

Algorithm 28: costo $O(1)$
1 DIRECT-ADDRESS-DELETE(T, x)
2 $T[x.key] \leftarrow NIL$

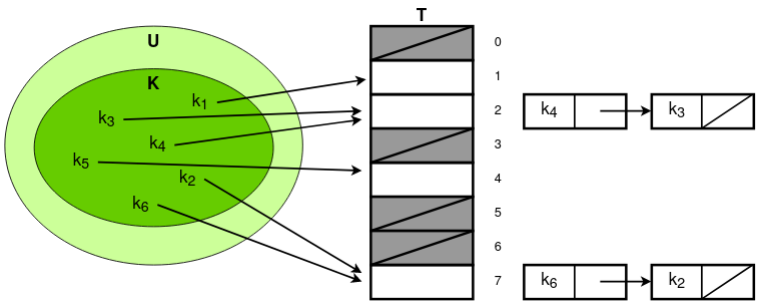


Con l'indirizzamento diretto la complessità in tempo per le tre operazione è $O(1)$, mentre la complessità in spazio è $O(size(U))$. Gli svantaggi dell'indirizzamento diretto è che se l'universo delle chiavi U è troppo grande, memorizzare una tabella T di dimensione U può essere impraticabile considerando la memoria disponibile in un normale calcolatore. Inoltre se l'insieme delle chiavi usate K è piccolo rispetto a U la maggior parte dello spazio allocato per la tavola T sarebbe sprecato.

9.2.2 Hashing

In generale si vuole avere una tabella di dimensione piccola rispetto alle chiavi e per fare ciò è fondamentale il modo in cui si sta implementando la funzione hash. Con l'indirizzamento diretto, un elemento con chiave k è memorizzato nellla cella k. Con l'hashing, questo elemento è memorizzato nella cella $h(k)$ dove h è la funzione hash. Il compito della funzione hash è ridurre l'intervallo degli indici e di conseguenza la dimensione dell'array. L'array ha dimensione m invece di U. Il problema è quando due chiavi vengono mappate nella stessa cella. Questo evento si chiama **collisione**. Se una funzione hash è ben progettata e apparentemente "casuale" può ridurre al minimo le collisioni ma evitarle completamente è impossibile. Esistono delle tecniche efficaci per risolvere i conflitti creati dalle collisioni:

- **Chaining**
  La tabella è un array di liste, si pongono tutti gli elementi che sono associati alla stessa cella in una lista concatenata. L'inserimento all'interno della lista può essere in testa o in coda. La cella j contiene un puntatore alla testa della lista di tutti gli elementi memorizzati che vengono mappati in j, se non ce ne sono, la cella j contiene la costante NIL.

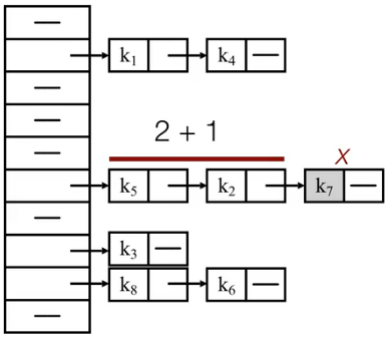


  Il tempo di esecuzione per la ricerca:

  – *Caso peggiore* è quando tutte le n chiavi sono mappate nella stessa posizione, quindi il tempo di esecuzione per scandire la lista è $\Theta(n)$ più il tempo per calcolare la funzione hash. Il caso peggiore in pratica

non accade mai, quindi, come nel Quick sort si fa l'analisi del caso medio con l'assunzione che ognuna delle chiavi ha le stesse probabilità di essere mappata in una delle posizioni della tabella (**simple uniform hashing**).

- *Caso medio*, siano n il numero di elementi nella tabella hash, m la dimensione della tabella hash e $\lambda = n/m$ il numero medio di elementi memorizzati in una lista. Per il simple uniform hashing quindi $n = n_0 + n_1 + ... + n_{m-1}$ cioè è la somma di tutte le lunghezze di tutte le liste con lunghezza attesa λ. Con questa assunzione il tempo di ricerca di una chiave k dipende dalla lunghezza di $n_{h(k)}$. Se la chiave k *non esiste*, allora le operazioni da effettuare sono accedere alla lista $h(k)$ e scorrere tutta la lista. Quindi la ricerca fallita incluso il tempo per calcolare $h(k)$ che si assume costante $\Theta(1)$, impiega $\Theta(1 + \lambda)$. Se la chiave k *esiste*, l'elemento può essere in qualsiasi posizione all'interno della lista e in quale posizione, con l'assunzione che gli inserimenti vengono fatti in testa, dipende da quanti elementi sono prima dell'elemento cercato x nella lista. Per trovare il numero atteso di elementi da esaminare prima di x si deve fare una media sugli elementi della tabella di 1 + il numero atteso di elementi aggiunti alla lista di x successivamente a x, ovvero si vuole sapere in media quanto è lunga la lista prima di x. Si chiama x_i lo i-esimo elemento inserito nella tabella, k_i la chiave di x_i e date le chiavi k_i e k_j si definisce la variabile casuale $X_{ij} = l\{h(k_i) = h(k_j)\}$ che vale 1 se le due chiavi sono uguali cioè che sia l'elemento i e j vanno nella stessa lista, altrimenti se l'evento non occorre vale 0. Nell'ipotesi di *simple uniform hashing* la probabilità che X_{ij} accada è $1/m$ e il valore atteso $E[X_{ij}] = 1/m$. Mettendo tutto insieme, cioè il numero atteso di elementi esaminati in una ricerca con successo è $E[\frac{1}{n}\sum_{i=1}^{n}(1 + \sum_{j=i+1}^{n} X_{ij})] = \Theta(1 + \lambda)$ dove la sommatoria va da $j = i + 1$ fino a n perché si devono prendere gli elementi che sono stati inseriti dopo i. Ricapitolando:



  * Se k non esiste si deve scorrere tutta la lista quindi $\Theta(1 + \lambda)$
  * Se k esiste, il conto viene fatto rispetto alla lunghezza media degli elementi che sono prima di x nella sua lista, questo numero dipende dal numero atteso di elementi che sono stati inseriti dopo x (per l'inserimento in testa) e che hanno avuto collisione. Mettendo tutto insieme si arriva a $\Theta(1 + \lambda)$

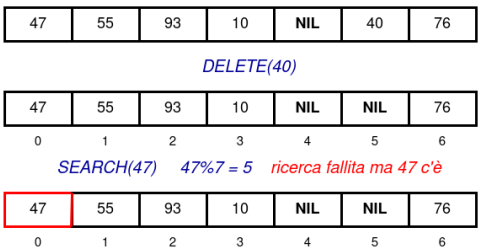
Il tempo di esecuzione per l'inserimento nel caso peggiore con l'inserimento in testa è $O(1)$. Nel caso della cancellazione, usando liste unidirezionali e conoscendo la posizione dell'elemento x da eliminare la cancellazione costa $O(1 + \lambda)$, invece, con liste bidirezionali costa $O(1)$

- **Open addressing**

Si usano array dinamici dove tutti gli elementi sono memorizzati nella tabella hash stessa, ovvero ogni cella della tabella contiene un elemento dell'insieme dinamico o la costante NIL. Per effettuare un inserimento si fanno dei tentativi (**probing**) cioè si calcola la funzione hash che restituisce un certo indice: se la posizione è occupata, si prova con un altra fino a che non si trova una posizione libera. Lo schema generale del probing:

- Primo tentativo - data la chiave k, si calcola la funzione hash $h(k)$
- Secondo tentativo - se $h(k)$ è occupata, si prova la posizione $h(k) + 1$
- Terzo tentativo - se $h(k) + f(1)$ è occupata, si prova con $h(k) + f(2)$ e così via

l'i-esimo tentativo è in posizione $h(k) + f(i)\ mod\ size$. Se i diventa $size - 1$, il probe è *fallito* cioè la tabella è piena. A seconda di come si definisce $f()$ il processo può fallire prima. Lunghe sequenze di probe sono costose. La ricerca usa la stessa sequenza di tentativi del inserimento e se trova la chiave cercata si restituisce l'elemento, altrimenti, si trova NULL e quindi la chiave non è nella tabella. La cancellazione è critica con l'open addressing: quando si cancella una chiave dalla cella i, non si può semplicemente marcare la cella come vuota inserendovi la costante NIL, perché la chiave potrebbe essere presente nella tabella ma in una posizione successiva alla cella i. Una soluzione consiste nel marcare la cella registrandovi il valore speciale DELETED, anzichè NIL



Esistono diverse tecniche di probing:

- **Linerare**, $f(i) = i$, quindi la sequenza di tentativi è $h'(k)\ mod\ m, (h'(k)+1)\ mod\ m, h'(k)+2)\ mod\ m$, e così via. Formalmente la funzione hash è $h(k,i) = (h'(k) + f(i))\ mod\ m$ dove $h'(k)$). Lo svantaggio del probing lineare è il **clustering primario** cioè tentativi ripetuti in una posizione occupata creano gruppi di posizioni occupate in quella zona della tabella. Lunghe sequenze di posizioni occupate fanno aumentare i tempi medi per la ricerca. Per ogni $\lambda < 1$, ovvero non si è riempito tutte le posizioni della tabella, il probing lineare trova sempre una posizione libera.
- **Quadratico**, si cerca di risolvere il cluster primario usando una funzione f quadratica $h(k,i) = (h'(k) + c_1 i + c_2 i^2)\ mod\ m$. Tipicamente si usa $f(i) = i^2$ con costanti ausiliarie $c_1 = 0$, $c_2 = 1$ e offset = 0, 1, 4, 9, 16, ... L'utilizzo di tutta la tabella è garantito solo con particolari valori di c_1, c_2 e m. Il probing quadratico soffre di un problema di clustering più lieve detto **clustering secondario**. Se due chiavi hanno la stessa posizione iniziale, la loro sequenza di tentativi è la stessa, dato che $h(k_1, 0) = h(k_2, 0)$ implica che $h(k_1, 1) = h(k_2, 1)$
- **Double hashing**, applica una funzione hash secondaria $h_2(k)$ quindi $f(i) = i * h_2(k)$ e la funzione hash è della forma $h(k,i) = (h_1(k) + i * h_2(k))\ mod\ m$. Per garantire che l'intera tabella venga utilizzata, il valore della seconda funzione hash $h_2(k)$ deve essere coprima rispetto alla dimensione m della tabella, ovvero, due interi sono coprimi se non hanno nessun divisore in comune eccetto 1.

Analisi open-addressing: il costo di una ricerca senza successo viene valutato come il numero atteso di tentativi prima di arrivare in una posizione che è NIL. In una ricerca senza successo, ogni tentativo tranne l'ultimo accede a una posizione occupata che non contiene la chiave cercata ovvero all'interno della posizione vi è scritto un elemento non si sta cercando o il flag DELETED. Si chiama X il numero di tentativi in una ricerca senza successo e A_i l'evento che l'i-esimo tentativo accade e che sia in una posizione occupata allora l'evento $\{X \geq i\}$ è l'intersezione degli eventi $\{A_i \cap A_2 \cap ... \cap A_{i-1}\}$. La probabilità condizionata $Pr(A_i \cap A_2 \cap ... \cap A_{i-1}) = \frac{1}{1-\lambda}$. Anche il costo per l'inserimento che richiede una ricerca senza successo e poi l'inserimento effettivo costa $\frac{1}{1-\lambda}$. Il costo di una ricerca con successo costa $\frac{1}{\lambda} ln \frac{1}{1-\lambda}$. In generale quando la tabella si riempie troppo, il tempo di ricerca medio peggiore da $O(1)$ a $O(n)$. Il proibing rende bene se $\lambda \leq 0.5$. Con il open addressing tipicamente appena la tabella è piena a metà si crea una tabella il doppio (o triplo, quadruplo, ...) più grande e si fa il refresh di tutti gli elementi nella nuova tabella.

Quindi ricapitolando per progettare una tabella hash occorre definire: una funzione hash, la dimensione della tabella e come risolvere le collisioni

9.3 Funzioni hash

Una funzione hash deve essere veloce $O(1)$ in pratica, deve distribuire i dati uniformemente e si deve utilizzare l'intera tabella, cioè non ci deve essere una posizione della tabella che non viene mai utilizzata dalla funzione hash. Degli possibili schemi sono:

- **Il metodo della divisione**, la funzione hash è $\boldsymbol{h(k) = k\ mod\ m}$. Questo metodo è molto veloce perché richiede una sola operazione di divisione. Quando si utilizza il metodo della divisione si evitano certi valori di m. Per esempio m non dovrebbe essere una potenza di 2 perché se $m = 2^p$ allora $h(k)$ rappresenta proprio i p bit meno significativi di k e questo aumenta le possibilità di collisioni. Un numero primo non troppo vicino a una potenza esatta di 2 è spesso una buona scelta per m
- **Il metodo della moltiplicazione** si svolge in due passi: si moltiplica la chiave k per una costante A, con $0 < A < 1$, e si estrae la parte frazionaria di kA cioè $kA - \lfloor kA \rfloor$; nel secondo passo si moltiplica questo valore per m, e si prende l'approssimazione inferiore del risultato ovvero $h(k) = \lfloor m(kA\ mod\ 1) \rfloor$. Un vantaggio del metodo della moltiplicazione è che m non è critico e tipicamente è una potenza di 2 perché semplifica il calcolo e l'implementazione

9.4 Hash per stringhe

Finora si sono considerate le chiavi come interi e quindi svolgere, per esempio $k\ mod\ m$ è semplice, ma se la chiave è una stringa, una possibile soluzione è interpretare i caratteri come numeri usando la tabella ASCII. Per ottenere $h(pt)$ dove 'pt' è la stringa si possono interpretare i singoli caratteri, rispettivamente 112 e 116, come un intero in base 128 ed effettuare la conversione: $h(pt) = (112, 116)_{10} = 112 * 128 + 116 = 14452 = h(14462)$. Con stringhe che possiedono troppi caratteri questa soluzione non è efficiente

Chapter 10

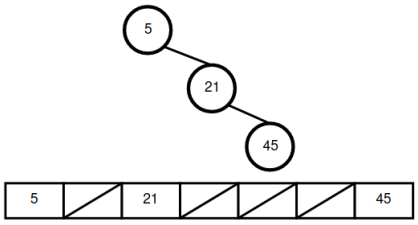
Alberi binari di ricerca e alberi AVL

10.1 Alberi binari

Gli alberi sono strutture dati che servono per rappresentare relazioni gerarchiche tra elementi detti *nodi*. Ogni albero inizia con un nodo detto *radice*. Un **albero binario** è un albero in cui ogni nodo ha al massimo due figli e ha la seguente definizione ricorsiva:

- l'albero T vuoto
- l'albero T non vuoto allora è composto da: $\begin{cases} radice \\ sotto-albero\ sinistro \\ sotto-albero\ destro \end{cases}$

Gli alberi binari, sono semplici da rappresentare in memoria. Rappresentare un albero binario in un array come in un heap ha il vantaggio dell'accesso diretto ai nodi ma ha lo svantaggio che l'altezza dell'albero deve essere nota e soprattutto si avrebbe tanto spreco di memoria. Si utilizzano liste collegate con $x.key$ per memorizzare il valore del nodo e $x.left, x.right$ per memorizzare i puntatori al figlio sinistro e al figlio destro di ogni nodo. Se $x.p = NIL$, allora x è la radice. Se il nodo x non ha un figlio sinistro, allora $x.left = NIL$, lo stesso vale per il figlio destro. $T.r$ punta alla radice dell'intero albero. Se $T.r = NIL$, allora l'albero è vuoto. Con questo schema l'altezza dell'albero non deve essere nota e non si ha nessun spreco in memoria, ma non si hanno più accesso diretto ai nodi e si utilizza memoria aggiuntiva per memorizzare riferimenti ai figli sinistro e destro.

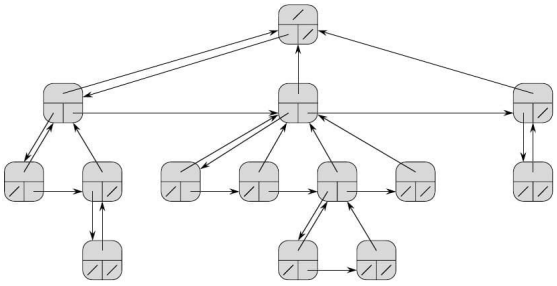


10.1.1 Alberi k-ari

Lo schema per rappresentare un albero binario può essere esteso a qualsiasi classe di **alberi k-ari** per qualche costante k: basta che si sostituiscono i puntatori $left, right$ con $child_1, ..., child_k$. Questo schema non funziona più se il numero di figli è illimitato perché non si sa in anticipo quanti puntatore allocare. Fortunatamente, esiste un altro schema che usa gli alberi binari per rappresentare gli alberi con un numero arbitrario di figli. Anziché avere un puntatore a ciascuno dei figli, ogni nodo ha soltanto due puntatori:

- $x.left - child$ punta al figlio più a sinistra del nodo x
- $x.right - siblings$ punta al fratello di x immediatamente a destra

Se il nodo x non ha figli, allora $x.left - child = NIL$. Se il nodo x è il figlio più a destra di suo padre allora $x.right - siblings = NIL$.



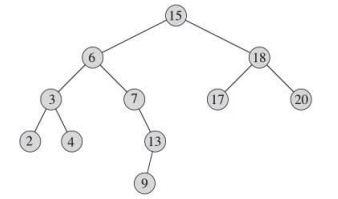
10.1.2 Visite albero binario

Si può visitare, cioè accedere a tutti i nodi dell'albero, in tre diversi modi:

- **Anticipata**, dato un sotto-albero, visita la radice e, poi ricorsivamente il figlio sinistro e il figlio destro

Algorithm 29: costo $\Theta(n)$

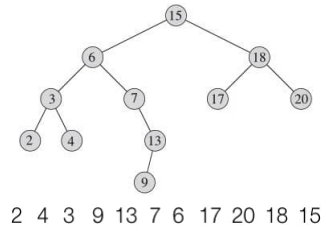
```
1 ANTICIPATA(x)
2 if x ≠ NULL then
3     print(x.key);
4     ANTICIPATA(x.left);
5     ANTICIPATA(x.right);
```



- **Posticipata**, dato un sotto-albero, visita prima ricorsivamente il figlio sinistro e il figlio destro, e infine la radice

Algorithm 30: costo $\Theta(n)$

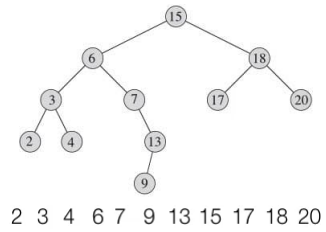
```
1 POSTICIPATA(x)
2 if x ≠ NULL then
3     POSTICIPATA(x.left);
4     POSTICIPATA(x.right);
5     print(x.key);
```



- **Simmetrica**, dato un sotto-albero, visita prima ricorsivamente il figlio sinistro, poi la radice, e infine ricorsivamente il figlio destro

Algorithm 31: costo $\Theta(n)$

```
1 SIMMETRICA(x)
2 if x ≠ NULL then
3     SIMMETRICA(x.left);
4     print(x.key);
5     SIMMETRICA(x.right);
```



10.1.3 Algoritmi ricorsivi su alberi binari

Gli alberi binari, essendo definiti in modo ricorsivo, permettono di progettare naturalmente gli algoritmi ricorsivi seguendo la metodologia del divide et impera:

- **Dimensione**, un parametro che caratterizza un albero è la sua dimensione n data dal numero di nodi in esso contenuti. La dimensione di un albero binario può essere definita ricorsivamente nel modo seguente: un albero vuoto ha dimensione 0, mentre la dimensione di un albero non vuoto è pari alla somma delle dimensioni dei suoi sotto-alberi, incrementata di 1, per includere la radice.

Algorithm 32: costo $\Theta(n)$

```
1 DIMENSIONE(u)
2 if u = NIL then
3     return 0
4 else
5     dimensioneSX ← DIMENSIONE(u.sx);
6     dimensioneDX ← DIMENSIONE(u.dx);
7     return dimensioneSX + dimensioneDX + 1;
```

La complessità equivale a una visita posticipata perché si va a contare la dimensione dell'albero sinistro, poi la dimensione dell'albero destro e poi la radice

- **Altezza dell'albero**, è la distanza massima di una foglia dalla radice in termine del numero di archi. Un albero vuoto ha altezza -1, un albero composto da un solo nodo ha altezza 1, e per un albero con nodi maggiori di due si calcola come il massimo tra il sotto-albero sinistro e il sotto-albero destro, incrementata di 1 per includere la radice. Anche in questo caso la complessità equivale a una visita posticipata

Algorithm 33: costo $\Theta(n)$

```
1 ALTEZZA(u)
2 if u = NIL then
3     return -1
4 else
5     altezzaSX ← ALTEZZA(u.sx);
6     altezzaDX ← ALTEZZA(u.dx);
7     return MAX(altezzaSX, altezzaDX) + 1;
```

10.2 Alberi binari di ricerca

Un **albero binario di ricerca**, detto anche **ABR**, è un albero binario per cui per ogni nodo x vale questa proprietà: $x.left.key \leq x.key$ *e* $x.right.key \geq x.key$. Per stampare i valori in ordine crescente si utilizza la visita simmetrica. Gli alberi binari di ricerca permettono di realizzare tutte le operazioni del dizionario (e altre) con un numero di operazioni proporzionale all'altezza dell'albero:

- **Ricerca**

Algorithm 34: costo $O(h)$

```
1 TREE-SEARCH(x, k)
2 if x ← NIL or k ← x.key then
3     return x
4 if k < x.key then
5     return TREE − SEARCH(x.left, k)
6 else
7     return TREE − SEARCH(x.right, k)
```

Algorithm 35: costo $O(h)$

```
1 ITERATIVE-TREE-SEARCH(x, k)
2 while x ≠ NIL and k ≠ x.key do
3     if k < x.key then
4         x ← x.left
5     else
6         x ← x.right
7 return x
```

La procedura inizia la sua ricerca dalla radice e segue un cammino semplice verso il basso lungo l'albero. Per ogni nodo x che incontra confronta la chiave k con $x.key$. Se le due chiavi sono uguali, la ricerca termina. Se k è minore di $x.key$, la ricerca continua nel sotto-albero sinistro di x, in quanto la proprietà degli alberi binari di ricerca implica che k non può essere memorizzato nel sotto-albero destro. Simmetricamente, se k è maggiore di $x.key$, la ricerca continua nel sotto-albero destro. La stessa procedura può essere scritta in modo iterativo "srotolando" la ricorsione in un ciclo while. Questa versione è più efficiente nella maggior parte dei calcolatori.

- **Minimo e massimo**, un elemento con chiave minima in un albero binario di ricerca può essere sempre trovato seguendo, a partire dalla radice, i puntatore $left$ dei figli a sinistra, fino a quando non viene trovato NILL.

Algorithm 36: costo $O(h)$

```
1 TREE-MINIMUM(x)
2 while x.left ≠ NIL do
3     x ← x.left;
4 return x
```

Algorithm 37: costo $O(h)$

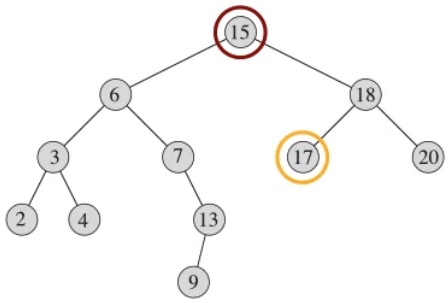
```
1 TREE-MAXIMUM(x)
2 while x.right ≠ NIL do
3     x ← x.right;
4 return x
```

La procedura restituisce un puntatore all'elemento minimo nel sotto-albero con radice in un nodo x, che si suppone diverso da NIL. Lo pseudocodice per TREE-MAXIMUM è simmetrico

- **Successore e predecessore**, dato un nodo in un albero, a volte è importante trovare il successore nell'ordine stabilito da un attraverso simmetrico. Se tutte le chiavi sono distinte, il successore di un nodo x è il nodo con la più piccola chiave che è maggiore di $x.key$. La struttura di un albero binario di ricerca consente di determinare il successore di un nodo senza mai confrontare le chiavi. La seguente procedura restituisce il successore di un nodo x in un albero binario di ricerca, se esiste, oppure NIL se x ha la chiave massima nell'albero

Algorithm 38: costo $O(h)$

```
1  TREE-SUCCESSOR(x)
2  if x.right ≠ NIL then
3      return TREE − MINIMUM(x.right)
4  y ← x.p;
5  while y ≠ NIL and x = y.right do
6      x ← y;
7      y ← y.p
8  return y
```



La procedura TREE-SUCCESSOR prevede due casi. Se il sotto-albero destro del nodo x non è vuoto, allora il successore di x è proprio il nodo più a sinistra nel sottoalbero destro, che viene trovato nella riga 3 chiamando $TREE-MINIMUM(x.right)$. D'altra parte se il sotto-albero destro del nodo x è vuoto e x ha un successore y, allora y è l'antenato più prossimo di x il cui figlio sinistro è anche antenato di x. Per esempio il successore del nodo con la chiave 13 è il nodo con chiave 15. Per trovare y, si risale l'albero partendo dal nodo x fino a che non si incontra un nodo che è il figlio sinistro di suo padre (righe 4-8). La procedura per trovare il successore è simmetrica:

Algorithm 39: costo $O(h)$

```
1  TREE-PREDECESSOR(x)
2  if x.left ≠ NIL then
3      return TREE − MAXIMUM(x.left)
4  y ← x.p;
5  while y ≠ NIL and x = y.left do
6      x ← y;
7      y ← y.p
8  return y
```

10.2.1 Inserimento e cancellazione

Le operazioni di inserimento e cancellazione modificano l'insieme dinamico rappresentato da un ABR. La struttura dati deve essere modificata per riflettere questa modifica, ma in modo tale che la proprietà degli alberi binari di ricerca resti valida.

- **Inserimento**, si suppone di voler inserire un nuovo valore v. La procedura TREE-INSERT riceve un nodo z per il quale $z.key = v, z.left = NIL, z.right = NIL$, in seguito modifica l'albero e qualche attributo di z in modo che z sia inserito in una posizione appropriata nell'albero

Algorithm 40: costo $O(h)$

```
1  TREE-INSERT(T, z)
2  y ← NIL, x ← T.root;
3  while x ≠ NIL do
4      y = x;
5      if z.key < x.key then
6          x ← x.left
7      else
8          x ← x.right
9  z.p ← y;
10 if y = NIL then
11     T.root ← z                                   // T e' vuoto
12 else if z.key < y.key then
13     y.left ← z
14 else
15     y.right ← z
```

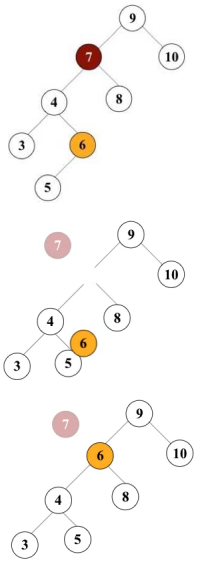
TREE-INSERT inizia dalla radice dell'albero e il puntatore x traccia un cammino semplice in discesa cercando un NIL da sostituire con l'elemento di input z. La procedura mantiene anche un puntatore *inseguitore* y che punta al padre di x. Dopo l'inizializzazione, le righe 3-8 del ciclo while spostano questi due puntatori verso il basso, andando a sinistra o a destra a seconda dell'esito del confronto fra $z.key$ e $x.key$, finché a x non viene assegnato il valore NIL. Serve un puntatore inseguitore y perché quando si arriva a trovare il NIL dove mettere z, la ricerca è andata un passo oltre il nodo che deve essere modificato. Le righe 9-15 impostano i puntatori che servono a inserire z.

- **Cancellazione**, la procedura per cancellare un nodo z da un albero binario di ricerca considera tre casi base:
  1. Se z non ha figli, si modifica suo padre $p[z]$ per sostituire z con NIL come suo figlio
  2. Se il nodo z ha un solo figlio, si eleva questo figlio in modo da occupare la posizione z nell'albero, modificando il padre z per sostituire z con il figlio di z
  3. Se il nodo z ha due figli, si deve trovare un nodo nei sotto-alberi di z e si può scegliere:
     - il minimo del sotto-albero destro (successore)
     - il massimo del sotto-albero sinistro (predecessore)

     Si sceglie la seconda strada: si cerca il predecessore, si elimina z, si imposta il figlio del predecessore come figlio del padre del predecessore e infine si porta il predecessore al posto di z. Nell'esempio si suppone di cancellare il nodo 7.

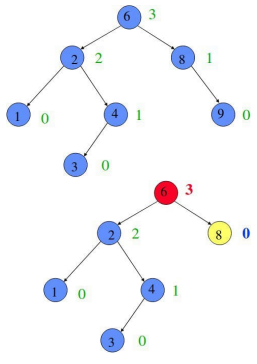
Algorithm 41: costo $O(h)$

```
1  DELETE(T, z)
2  if z.degree = 2 then                          // Caso 3
3      pred ← MAXIMUM(z.left);
4      swap z.key with pred.key;
5      z ← pred
6  f ← NULL;
7  if z.left ≠ NULL then                         // Caso 2
8      f ← z.left
9  if z.right ≠ NULL then                        // Caso 2
10     f ← z.right
11 if f = NULL then                              // Caso 1
12     stacca(z)
13 else                                          // Caso 2
14     swap z with f;
15     stacca(z)
```

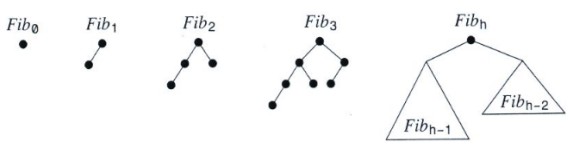


10.3 Alberi AVL

Un **albero AVL** è un albero binario di ricerca che garantisce avere un altezza di $h = O(logn)$ per n elementi memorizzati nei suoi nodi. Oltre alla proprietà di ricerca, l'albero AVL soddisfa la proprietà di essere **1-bilanciato** al fine di garantire l'altezza logaritmica: l'albero T si dice 1-bilanciato se, per ogni nodo u la differenza delle altezze dei due figli differiscono per al più di un'unità ovvero $|\boldsymbol{h(u.sx) - h(u.dx)}| \leq \mathbf{1}$. In figura un esempio di albero AVL e un albero non AVL. La connessione tra l'essere 1-bilanciato e avere altezza logaritmica non è immediata e passa attraverso gli **alberi di Fibonacci** che sono un sottoinsieme degli alberi 1-bilanciati che raggiungono altezza h con il minor numero di nodi, ovvero se si elimina un solo nodo si esce dall'insieme degli alberi 1-bilanciati. L'albero di Fibonacci Fib_h di altezza h è definito ricorsivamente:



$$Fib_h \begin{cases} per\ h = 0\ allora\ n_0 = 1\ nodo \\ per\ h = 1\ allora\ n_1 = 2\ nodi \\ per\ h > 1\ allora\ n_h = n_{h-1} + n_{h-2} + 1\ nodi \end{cases}$$



Sia n_h la dimensione di un albero di Fibonacci di altezza h, si dimostra:

1. $n_h = n_{h-1} + n_{h-2} + 1$

   ovvio, per costruzione dove 1 è la radice e, n_{h-1} e n_{h-2} sono le dimensioni dei due sotto-alberi

2. $n_h = F_{h+3} - 1$ dove per F_{h+3} si intende l'h+3-esimo della successione di Fibonacci

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
n_h	1	2	4	7	12	20	33	54	88	143	232	376	609	986	1596
F_h	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377

   si dimostra per induzione su h

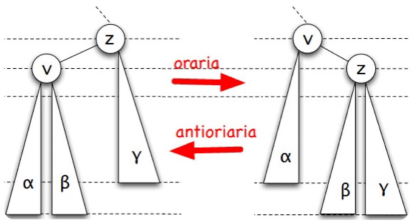
Caso base 1:	$h = 0$ allora $n_0 = 1$ e $F_{h+3} - 1 = F_{0+3} - 1 = 2 - 1 = 1$
Caso base 2:	$h = 1$ allora $n_1 = 2$ e $F_{h+3} - 1 = F_{1+3} - 1 = 3 - 1 = 2$
Ipotesi induttiva	per ogni altezza $l < h$ si ha $n_l = F_{l+3} - 1$
Passo induttivo	si dimostra che $n_h = F_{h+3} - 1$. Si sa che per costruzione $n_h = n_{h-1} + n_{h-2} + 1$ ma $h-1$ e $h-2$ sono strettamente minori di h allora si può applicare l'ipotesi induttiva $n_h = (F_{h-1+3} - 1) + (F_{h-2+3} - 1) + 1 = F_{h+2} - 1 + F_{h+1} - 1 + 1 = F_{h+3} - 1$

   Utilizzando la nota forma chiusa dei numeri di Fibonacci $F_h = \dfrac{\phi^h - (1-\phi)^h}{\sqrt{5}}$ dove $\phi = \frac{1+\sqrt{5}}{2} = 1.6180339...$ si può affermare che $F_h > \dfrac{\phi^h - 1}{\sqrt{5}}$ e che quindi, esiste una costante $c > 1$ tale che $F_h > c^h$ per $h > 2$. Pertanto $n_h = F_{h+3} - 1 \geq c^h$. Sia n il numero di nodi di un albero 1-bilanciato e n_h il numero di nodi di un albero di Fibonacci allora si ha $n \geq n_h \geq c^h$, che per la proprietà transitiva $n \geq c^h$. Facendo il logaritmo da entrambe le parti si arriva a $h \geq log_c n$ che equivale a $h = O(logn)$

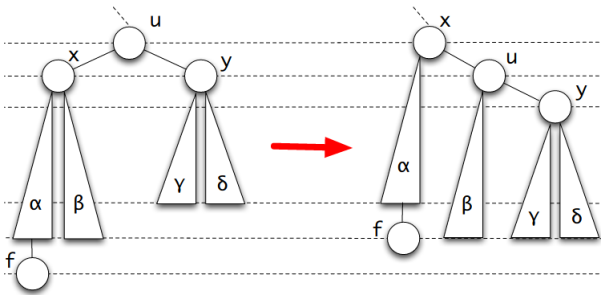
10.3.1 Dizionario implementato con AVL

Tutte le interrogazioni (ricerca, minimo, massimo, successore, predecessore) hanno costo $O(h)$ ma siccome $h = logn$ il tempo complessivo è pari a $O(logn)$ al caso pessimo. Il codice è lo stesso degli alberi binari di ricerca. Per quanto riguarda le operazioni di modifica dell'insieme dinamico:

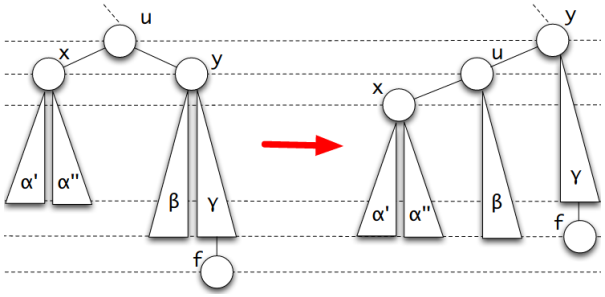
- **Inserimento**, si svolge come negli ABR ma dopo aver inserito il nodo, l'albero potrebbe aver perso la proprietà di bilanciamento. I nodi che corrono il pericolo di sbilanciamento sono gli antenati della nuova foglia. Dopo aver inserito la foglia si risale l'albero per andare a verificare quali antenati si sono sbilanciati e il controllo si arresta sul primo nodo il cui fattore di sbilanciamento è diventato 2. Questo nodo si chiama **nodo critico**. Una volta individuato il nodo critico si svolge una operazione sul nodo, chiamata **rotazione**, che ripristina l'altezza del sotto-albero radico sul nodo critico e quindi di conseguenza sistema anche tutti gli altri antenati ovvero la rotazione garantisce che l'albero diventa nuovamente 1-bilanciato. Per implementare gli alberi AVL, oltre ad avere i puntatori alla radice e ai sotto-alberi sinistro e destro, si introduce un ulteriore campo $u.altezza$. nei suoi nodi u tale che $u.altezza = h(u)$. I quattro casi possibili di sbilanciamento del nodo critico u a causa della creazione della foglia f:



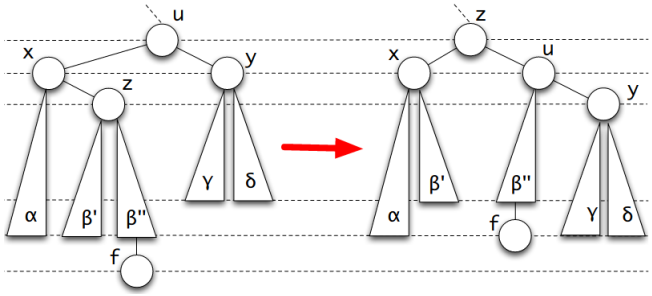
  - *Caso SS*, la foglia f appartiene al sotto-albero α radicato in $u.sx.sx$, cioè al sotto-albero *sinistro* del figlio *sinistro* del nodo critico $\Rightarrow rotazione\ oraria\ (u)$



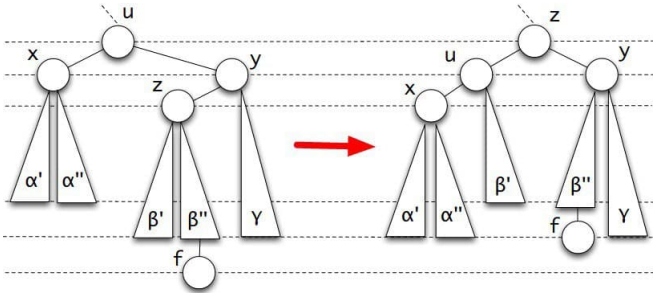
– *Caso DD*, la foglia f appartiene al sotto-albero γ radicato in $u.dx.dx$, cioè al sotto-albero *destro* del figlio *destro* del nodo critico $\Rightarrow$ $rotazione\ antioraria\ (u)$



– *Caso SD*, la foglia f appartiene al sotto-albero β radicato in $u.sx.dx$, cioè al sotto-albero *destro* del figlio *sinistro* del nodo critico $\Rightarrow$ $rotazione\ antioraria\ (u.sx) \Rightarrow rotazione\ oraria\ (u)$



– *Caso DS*, la foglia f appartiene al sotto-albero β radicato in $u.dx.sx$, cioè al sotto-albero *sinistro* del figlio *destro* del nodo critico $\Rightarrow$ $rotazione\ oraria\ (u.dx) \Rightarrow rotazione\ antioraria\ (u)$



Algorithm 42: costo $O(1)$

```
1 RUOTA-ORARIA(z)
2 v ← z.sx;
3 z.sx ← v.dx;
4 v.dx ← z;
5 z.altezza ←
    MAX(ALTEZZA(z.sx), ALTEZZA(z.dx))+1;
6 v.altezza ←
    MAX(ALTEZZA(v.sx), ALTEZZA(v.dx))+1;
7 return v
```

Algorithm 43: costo $O(1)$

```
1 RUOTA-ANTIORARIA(z)
2 z ← v.dx;
3 v.dx ← z.sx;
4 z.sx ← v;
5 v.altezza ←
    MAX(ALTEZZA(v.sx), ALTEZZA(v.dx))+1;
6 z.altezza ←
    MAX(ALTEZZA(z.sx), ALTEZZA(z.dx))+1;
7 return z
```

- **Cancellazione**, funziona come negli ABR, il problema a differenza dell'inserimento e che possono esserci più nodi critici tra gli antenati del nodo cancellato. Nel caso peggiore si deve eseguire una rotazione su ogni antenato del nodo cancellato, che siccome al massimo l'altezza è logaritmica si deve svolgere la rotazione su un numero logaritmico di nodi. Il costo della cancellazione è $O(h) = O(logn)$ più $logn$ rotazioni. Nella pratica si preferisce marcare logicamente i nodi cancellati, che vengono ignorati ai fini della ricerca. Quando una frazione costante dei nodi sono marcati come cancellati, si ricostruisce l'albero con le sole chiavi valide

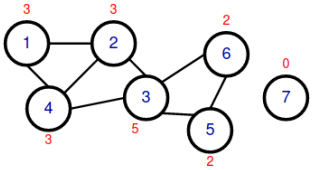
Chapter 11

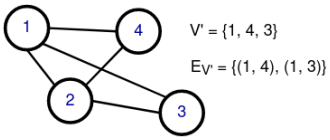
Grafi

11.1 Definizioni

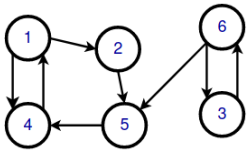
Un grafo $G = (V, E)$ è definito come una coppia di insiemi finiti di V vertici (o nodi) e E archi. Si indica con n il numero di vertici, chiamato anche *ordine del grafo* e con m il numero degli archi. Il numero dei vertici sommato il numero degli archi è detto *dimensione del grafo* $n+m = |V|+|E|$. Quando $m = O(n)$ il grafo è detto **sparso**, invece se $m = \Theta(n^2)$ il grafo è detto **denso**. Un grafo può essere:

- **Non orientato**, tra gli archi vi è una relazione biunivoca tra i nodi, ovvero non è specificata la direzione negli archi. Presi due vertici $u, v \in V$ se si ha l'arco $(u, v) \in E$ allora si dice che u e v sono **adiacenti**. Si dice inoltre che l'arco (u, v) è **incidente** su u e v. Per ogni $v \in V$ si definisce **grado** di v il numero di archi incidenti su v. Si usa la notazione $\delta(v)$. Si dice che v è un vertice *isolato* se il suo grado è 0. In figura un grafo non orientato dove in rosso si è indicato il grado dei vertici. La somma del grado di tutti i vertici del grafo è il doppio dei numeri degli archi, questo perché ogni arco incrementa di un fattore uno il grado dei vertici incidenti e quindi contribuisce alla somma per un fattore 2. Il **cammino** tra due vertici (u, v) è dato da una sequenza di vertici a due a due adiacenti (cioè collegati con un arco) tale che il primo vertice della sequenza è uguale a u, l'ultimo vertice è v, e gli altri vertici appartengono a E. Il numero di archi sul cammino è chiamato la **lunghezza** del cammino, ovvero il numero di archi che lo compongono. Un **ciclo** è un cammino che torna al vertice di partenza. Un cammino si dice **semplice** se tutti i vertici che lo compongono sono distinti, cioè non vi sono cicli. La distanza tra 2 vertici $\delta(u, v)$ è il minimo numero di archi da percorrere per spostarsi da u a v. Nel primo esempio il cammino da 1 a 5 ha lunghezza quattro, invece la distanza è pari a due. Dati due vertici $u, v \in V$ si dice che u e v sono **connessi** se esiste un cammino da u verso v. Un grafo G si dice che è connesso se ogni coppia di nodi è connessa. Un grafo è **completo** quando ogni coppia di vertici è adiacente. Si definisce **sottografo** $G' = (V', E')$ di un grafo $G = G(V, E)$ se $V' \subseteq V$, $E' \subseteq V \times V$ e $E' \subseteq E$. Il sottografo **indotto** da $V' \subseteq V$ è il grafo $G = (V', E_{V'})$ dove $E_{V'} = \{(u, v) \in E \; t.c. \; u \in V', v' \in V\}$. Una **componente connessa** è un sottografo G' di G connesso e **massimale**, dove per massimale si intende che non si può aggiungere al sottografo nessun altro vertice





- **Orientato**, per ogni arco è specificato la direzione. Una coppia generica di vertici $(u, v) \neq (v, u)$ quindi se $|V| = n$ allora $|E| \leq n(n-1)$. Per ogni vertice si indica con $\delta_u(v)$ il **grado uscente** di v, ovvero il numero di archi che escono da v. Si indica $\delta_e(v)$ il **grado entrante**, ovvero il numero di archi che entrano in v. Il **grado generale** del vertice è pari a $\delta(v) = \delta_e(v) + \delta_u(e)$. La somma dei gradi uscenti (o gradi entranti) corrisponde al numero degli archi, questo perché un arco fornisce un contributo pari a uno alla somma. Il **cammino orientato** da u a v è dato da una sequenza di vertici dove il primo vertice corrisponde a u, l'ultimo a v e gli altri altri vertici hanno archi orientati nel verso corretto. Un **ciclo orientato** è un cammino orientato che torna al vertice di partenza. Per avere un ciclo ci devono essere almeno due vertice con due archi. Due vertici u e v sono connessi se esiste un cammino orientato da u verso v. Un grafo $G = (V, E)$ orientato è **fortemente connesso** se ogni coppia ordinata di nodi è connessa, ovvero per ogni (u, v) deve esistere un cammino orientato che va da u verso v ma anche un altro cammino orientato che va da v verso u, cioè u e v sono mutualmente raggiungibili. Una **componente fortemente connessa** è un sottografo fortemente connesso e massimale. In figura un grafo orientato non fortemente connesso, per esempio non esiste un cammino orientato che va da 1 a 6. Il grafo in figura ha due componenti fortemente connesse $\{1, 2, 4, 5\}$ e $\{6, 3\}$. Un grafo si dice **aciclico** è un grafo che non contiene cicli.



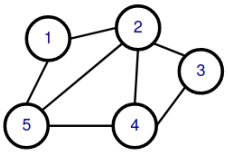
11.2 Rappresentazione in memoria

Esistono due metodi per rappresentare un grafo:

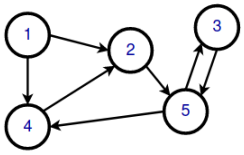
- **Matrice di adiacenza**, si suppone che i vertici siano numerati $1, 2, ...|V|$ in modo arbitrario. La rappresentazione con una matrice di adiacenza di un grafo G consiste in una matrice A di dimensione $n \times n$:

$$A[i,j] = \begin{cases} 1 \; se \; (i,j) \in E \\ 0 \; se \; (i,j) \notin E \end{cases}$$ occupa uno spazio fisso pari a $O(|V|)^2$.

  Questo metodo funziona meglio con un grafo denso perché se fosse sparso, si avrebbe molti zeri e quindi si sprecherebbe molto spazio. Per un grafo non orientato si può utilizzare solo metà matrice perché essa è simmetrica. Gli uno in un grafo orientato rappresentano solo gli archi uscenti. Il metodo con matrice può essere comoda se si deve rappresentare un **grafo pesato** cioè un grafo che associa ad ogni arco un valore detto *peso*. Se il grafo è pesato al posto degli uno si metterà il peso e si dovrà trovare anche un valore che indichi l'assenza di arco perché a questo punto lo zero potrebbe essere un peso



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

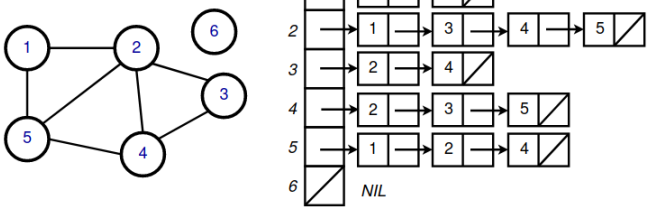


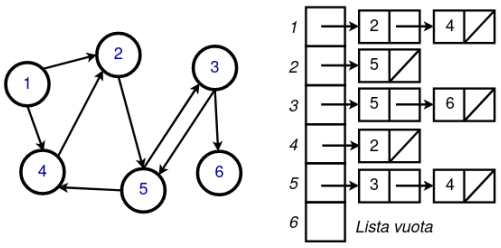
	1	2	3	4	5
1	0	1	0	1	0
2	0	0	0	0	1
3	0	0	0	0	1
4	0	1	0	0	0
5	0	0	1	1	0

- **Liste di adiacenza**, si associa a ogni vertice, la lista dei vertici adiacenti, quindi si rappresenta il grafo come un'array Adj di n liste, dove $n = |V|$. Per ogni $i \in V$

$$Adj[i] = \begin{cases} NIL \qquad se \; \delta(i) = 0 \\ lista \; di \; tutti \; vertici \; j \; t.c. \; (i,j) \in E \end{cases}$$

  Per ogni i la lunghezza della lista $Adj[i]$ è pari al grado $\delta(i)$ nei grafi non orientati, e al grado uscente $\delta_u(i)$ nei grafi orientati. Occupa uno spazio pari a $O(n+m)$ dove n è lo spazio occupato dall'array e m è il numero di archi





Analisi costo delle operazioni per le due rappresentazioni:

Operazioni	Matrice di adiacenza	Liste di adiacenza
$adiacenti(i,j)$	$\Theta(1)$	$O(\delta(i))$
$grado(i)$	$\Theta(n)$	$\Theta(\delta(i))$
$aggiungiArco(i,j)$	$\Theta(1)$	$O(\delta(i))$ o $O(\delta(i)+\delta(j))$
$rimuovereArco(i,j)$	$\Theta(1)$	$O(\delta(i))$ o $O(\delta(i)+\delta(j))$

Dati i e j per stabilire che sono adiacenti in una matrice di adiacenza basta fare un $return \; A[i,j]$, mentre in una lista di adiacenza si deve cercare j nella lista $Adj[i]$ quindi costa $O(\delta(i))$. Per calcolare il grado del vertice i, per quanto riguarda la matrice, si deve accedere alla riga i e fare la somma di tutta la riga. In una lista si deve percorrere tutta la lista del vertice i e incrementare un contatore. Aggiungere e rimuovere un arco in una matrice di adiacenza costa costante perché per quanto riguarda aggiungere un arco basta fare $A[i,j] = 1$ (e nel caso di un grafo non orientato anche $A[j,i] = 1$), invece per rimuovere un arco si fa $A[i,j] = 0$ (in un grafo non orientato anche $A[j,i] = 0$). In una lista di adiacenza aggiungere un arco costa $\Theta(1)$ se la lista non è ordinata, mentre costa $O(\delta(i))$ se si vuole mantenere le liste ordinate in un grafo orientato, e $O(\delta(i)+\delta(j))$ per un grafo non orientato. Per rimuovere un arco in grafo orientato si spende $O(\delta(i))$ poiché si rimuove j dalla lista $Adj[i]$, in un grafo non orientato si spende $O(\delta(i)+\delta(j))$ perché si elimina i da $Adj[j]$ e j da $Adj[i]$

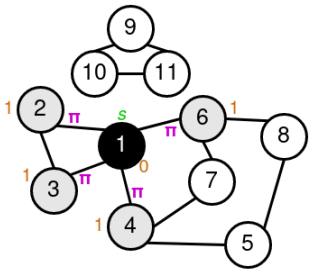
11.3 Breadth-First Search

La **BFS**, o visita in **ampiezza**, ispeziona sistematicamente tutti i suoi vertici e archi (o di quelli raggiungibili dal **vertice sorgente**). Dato un grafo $G = (V, E)$, rappresentato con liste di adiacenza ordinate per vertice crescente, e un vertice sorgente s, la visita in ampiezza scopre tutti i vertici raggiungibili da s in ordine di distanza crescente, e calcola la distanza da s a ciascun vertice raggiungibile. Se il grafo è connesso la visita scopre tutti i vertici, altrimenti solo la componente connessa che contiene la sorgente. Si utilizza una struttura dati di appoggio che è una coda Q con schema FIFO per tenere traccia del lavoro svolto. A differenza della visita sugli alberi, sui grafi vi è il problema dei cicli. Per evitare questo problema si usa la *tecnica della colorazione*: per ogni vertice del grafo si manterrà un campo $v.color$ che potrà avere uno dei seguenti valori:

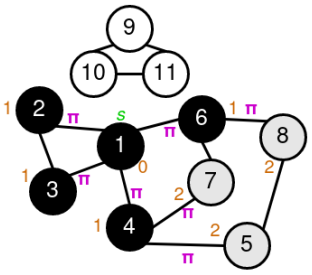
- *Bianco*, prima di essere scoperto, cioè non è stato ancora visitato
- *Grigio*, quando viene visitato per la prima volta
- *Nero*, quando tutti gli archi uscenti dal vertice sono stati visitati, cioè l'esame di $Adj[v]$ è terminato

si manterrà anche un campo intero $v.d$ in cui si memorizzerà la distanza dalla sorgente e infine un campo $v.\pi$ che conterrà il nome del vertice da cui v è stato scoperto che lo si chiamerà *predecessore*, ovvero u è il predecessore di v se quest'ultimo è stato scoperto esplorando l'arco (u, v). Il predecessore è importante perché permetterà di costruire una volta che la visita è stata completata un **albero BF** che è un albero di *cammini minimi* che contiene tutti i vertici raggiungibili da s. All'inizio tutti i vertici avranno il campo $color$ uguale a bianco, il campo predecessore uguale a NIL e il campo distanza sarà inizializzato per tutti i vertici a ∞. La visita procede in questo modo:

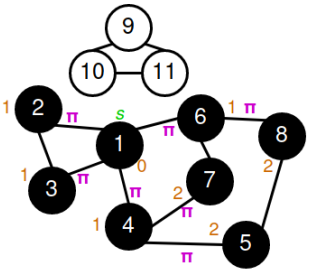
1. si parte dalla sorgente, quindi si colora di grigio perché si è scoperto il vertice e si mette nella coda
2. si estrae dalla coda, e si esamina la lista di adiacenza della sorgente. Se nella lista ci sono dei vertici che sono ancora bianchi, si colorano di grigio e si inseriscono nella coda. Il primo vertice che si trova è 2, si colora di grigio e si inserisce in coda. Siccome il vertice 2 si è scoperto esaminando la lista di 1, il predecessore di 2 diventa 1. Oltre al predecessore si calcola anche la distanza. Stesso procedimento per gli altri vertici presenti nella lista di 1 cioè 3, 4 e 6
3. quando finisce l'esame della lista di adiacenza della sorgente, si colora di nero la sorgente e lo si estrae dalla coda.



Q: ~~1~~, 2, 3, 4, 6

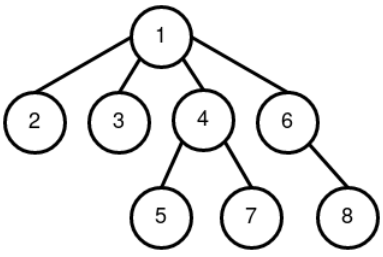


Q: ~~1~~, ~~2~~, ~~3~~, ~~4~~, 6, 5, 7, 8



Q: ~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~6~~, ~~5~~, ~~7~~, 8

Si estrae il vertice 2, e si esamina la sua lista di adiacenza: si trovano 1 e 3, dove uno è colorato di nero, mentre l'altro è colorato di grigio quindi non si aggiunge nessun vertice in coda perché non si è scoperto niente di nuovo dal grafo. Si colora il vertice 2 di nero. Stessa sorte per il vertice 3. Si estrae il vertice 4, si trovano 7 e 5, si colorano di grigio si settano i predecessori e si calcolano le distanze cioè prendono il valore della distanza del predecessore incrementato di uno. Si estrae dalla coda 6, si trova il vertice 8 quindi lo si colora di grigio, si setta il predecessore e si calcola la distanza. Infine si estrae dalla coda 5 ma i suoi sono già stati scoperti e quindi la visita terminerà. Stesso procedimento per i vertici 7 e 8. La coda è terminata e quindi la visita finisce. Gli altri tre nodi che facevano parte dell'altra componente connessa non vengono raggiunti perché sono composti da nodi non raggiungibili dalla sorgente. In parallelo all'esecuzione della visita implicitamente si costruisce l'albero BF che riporta tutti i vertici che sono stati raggiunti dalla sorgente e soltanto gli archi etichettai con π:



Pseudocodice:

Algorithm 44: costo $O(|V|+|E|)$

```
1  BFS(G, s)
2  for ogni vertice v ∈ V − {s} do
3      v.color ← BIANCO;
4      u.d ← ∞;
5      u.π ← NIL;
6  s.color ← GRIGIO ;                                   // setta la sorgente
7  s.d = 0;
8  s.π = NIL;
9  Q ← ∅;
10 ENQUEUE(Q, s) ;                                      // inserimento nella coda
11 while Q ≠ ∅ do
12     u ← DEQUEUE(Q) ;                                 // estrai la testa dalla coda
13     for ogni v ∈ G.Adj[u] do                         // esamina vertici lista di u
14         if v.color = BIANCO then
15             v.color ← GRIGIO;
16             v.d ← u.d + 1;
17             v.π ← U;
18             ENQUEUE(Q, v)
19     u.color ← NERO ;                                 // al termine del controllo colora di nero u
```

Le operazioni di inserimento e cancellazione dalla coda richiedono un tempo costante, quindi il tempo totale dedicato alle operazioni con la coda è $O(|V|)$. La fase preliminare (righe 2-8) costa $\Theta(|V|)$ perché si devono settare i valori ad ogni vertice e poi lo si fa anche sul nodo sorgente. La lista di adiacenza di un nodo viene esaminata al più una volta, quando il vertice è estratto dalla coda. Il costo del while è pari alla somma della lunghezza della lista di adiacenza di ogni vertice v estratto dalla coda Q, cioè è pari alla somma dei gradi di ogni vertice v estratto dalla coda, che equivale a $O(|E|)$. Il tempo di esecuzione totale di BFS è $O(|V|+|E|)$, diventa $\Theta(|V|+|E|)$ se il grafo è connesso. L'occupazione in spazio per svolgere la visita è pari al numero dei vertici $\Theta(|V|)$. L'albero BF è un sottografo $BF = (V_\pi, E_\pi)$ dove V_π è l'insieme di tutti vertici che si è scoperto, cioè $v.\pi \neq NIL + s$ (la sorgente ha $v.pi = NIL$ ma comunque è all'interno dell'insieme V_π). Invece E_π è l'insieme degli archi composti da un vertice e dal suo predecessore (v_π, v) per ogni $v \in V_\pi - s$. L'albero BF è sicuramente un albero perché è connesso e perché si conoscono esattamente il suo numero di archi che sono tanti quanti gli elementi di V_π meno uno.

11.3.1 Stampa cammini minimi

La visita BFS si applica a tutti i problemi che richiedono di trovare un cammino minimo, perché una volta fatta, l'albero BF permette, per esempio, di stampare un cammino minimo

Algorithm 45: costo $O(n)$

```
1  PRINT-PATH(G, s, v)
2  if s = v then
3      stampa s
4  else if v.π = NIL then
5      stampa "non ci sono cammini da s a v"
6  else
7      PRINTH − PATH(G, s, v.π);
8      stampa v
```

Se si vuole andare dalla sorgente alla sorgente, quindi si rimane dove si è, si passa da un solo vertice che è s e quindi il cammino finisce. Se il vertice v non è la sorgente e non ha un predecessore significa che non è raggiungibile dalla sorgente, dunque non esistono cammini da s a v. Se v è raggiungibile da s allora si costruisce il cammino ricorsivamente. La procedura costa lineare sulla lunghezza del cammino minimo da s a v, quindi al massimo sarà ordine di n

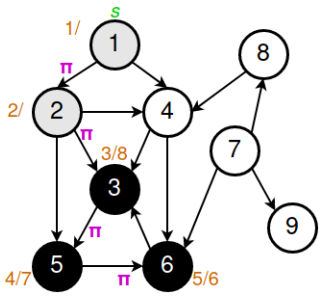
11.4 Depth-First Search

La strategia adottata da **DFS**, o visita in **profondità**, consiste nel visitare il grafo sempre più in profondità se possibile. Nella visita in profondità, gli archi vengono ispezionati a partire dall'ultimo vertice scoperto v che ha ancora archi non ispezionati che escono da esso. Quando tutti gli archi di v sono stati ispezionati, la visita "fa marcia indietro" per ispezionare gli archi che escono dal vertice dal quale v era stato scoperto. Questo processo continua finché non saranno stati scoperti tutti i vertici che sono raggiungibili dal vertice sorgente originale. Se restano dei vertici non scoperti, allora uno di essi viene selezionato come nuovo vertice sorgente e la visita riparte da questa sorgente. L'intero processo viene ripetuto finché non saranno scoperti tutti i vertici del grafo. La DFS ha una struttura ricorsiva, e la si può ottenere con una BFS sostituendo la coda con una pila perché la visita ogni volta riparte dall'ultimo vertice scoperto. Come nella visita in ampiezza, quando un vertice v viene scoperto durante un'ispezione della lista di adiacenza di un vertice u già scoperto, la visita in profondità registra questo evento assegnando u all'attributo $v.\pi$ di v (il predecessore). Il sottografo dei predecessori prodotto da una visita in profondità può essere formato da più alberi, perché la visita può essere ripetuta da più sorgenti, cioè forma una **foresta DF** composta da vari alberi alberi DF. Come nella visita in ampiezza, i vertici vengono colorati per indicare il loro stato. Inizialmente tutti i vertici sono bianchi. Un vertice diventa grigio quando viene scoperto durante la visita, e diventa nero quando viene completato, ovvero quando la sua lista di adiacenza è stata completamente ispezionata. Questa tecnica garantisce che ogni vertice vada a finire in un solo albero DF, in modo che questi alberi siano disgiunti. Oltre a creare una foresta DF, la visita in profondità associa anche a ciascun vertice delle informazioni temporali. Ogni vertice v ha due informazioni temporali:

1. $v.d$ registra il momento in cui il vertice v viene scoperto (e colorato di grigio)
2. $v.f$ registra il momento in cui la visita completa l'ispezione della lista di adiacenza del vertice v (che diventa nero)

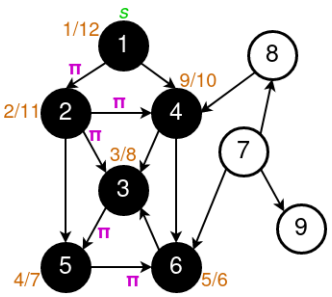
Queste informazioni temporali sono numeri interi compresi tra 1 e $2|V|$, perché ciascuno dei $|V|$ vertici può essere scoperto una sola volta e la sua visita può essere completata una sola volta. Per ogni vertice u si ha $u.d < u.f$. La visita procede in questo modo:

1. si parte dalla sorgente, il tempo $v.d$ è pari a uno, si colora di grigio, e si scorrono i suoi archi uscenti presenti nella lista di adiacenza dove si troverà 2 e 4. Siccome 2 è il primo ed è bianco lo si scopre, $v.d$ è uguale a due e 1 diventa il suo predecessore. Si lascia in sospeso l'esame della lista di adiacenza di 1 e si riparte con la visita da 2

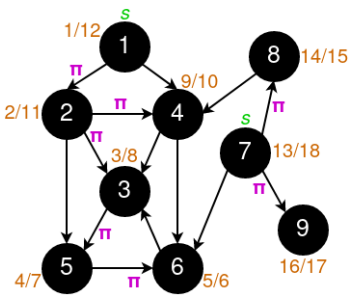


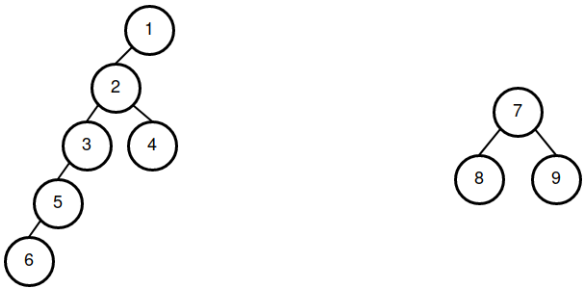
2. Per primo nella lista di 2 si vede 3, si setta quindi il predecessore e si calcola il tempo. Stesso procedimento nella lista di 3 dove al tempo quattro si è scoperto il vertice 5, e al passo cinque si scopre 6 dalla lista di adiacenza di 5. Dal vertice 6 esiste un arco uscente che porta verso il vertice 3, ma quest'ultimo è già stato scoperto.

3. A questo punto al passo sei è terminata la visita del vertice 6, quindi $v.f = 6$ e si colora il vertice di nero. Ora si è su un vertice che non si ha più archi uscenti da esplorare, allora si torna sul predecessore cioè 5, ma nemmeno questa volta si trovano archi uscenti da esplorare, quindi al passo sette si completa la visita del vertice 5. Si torna su 3 e si completa al passo otto.



Il predecessore di 3 è 2, che ha un arco uscente verso 4 che è ancora colorato di bianco. Si setta il predecessore di 4 che è 2, si imposta $v.d = 9$. Si controllano gli archi uscenti di 4 che sono verso i vertici scoperti 3 e 6, allora si termina il vertice al passo dieci. Si torna al vertice 2 che però non ha più archi uscenti e quindi termina al passo undici. Al tempo dodici si ha la stessa fine per il vertice 1. A questo punto la visita DFS ha scoperto tutti i vertici che sono raggiungibili dal vertice sorgente, ma dato che rimangono dei vertici bianchi, la visita prosegue dal minore tra i vertici rimasti, cioè il vertice 7. Si sceglie 7 come nuova sorgente all'istante tredici, e si ricomincia: la sua lista di adiacenza contiene il 6 che però è già stato scoperto, allora si scopre il vertice 8 al passo quattordici che però termina subito al passo quindici perché non ha archi uscenti; dopo essere tornati su 7, si va al vertice 9 al tempo sedici che come prima non ha archi uscenti allora termine nell'istante diciassette; dopo essere tornati al nodo sorgente 7, si termina con $v.f = 18$. Implicitamente si costruisce la foresta DF che in questo caso è composto da due diversi alberi DF:





La foresta DF non è unica ma dipende da come sono ordinate le liste. Pseudocodice:

Algorithm 46: costo $\Theta(|V| + |E|)$

```
1 DFS(G)
2 for v ∈ V do
3     v.color ← BIANCO;
4     u.π ← NIL;
5 time ← 0 ;                    // variabile globale
6 for v ∈ V do
7     if v.color = BIANCO then
8         DFS − VISIT(G, v)
```

Algorithm 47: costo $\Theta(|E|)$

```
1  DFS-VISIT(G, u)
2  time ← time + 1;
3  u.d ← time;
4  u.color ← GRIGIO;
5  for v ∈ G.Adj[u] do        // ispeziona arco (u, v)
6      if v.color = BIANCO then
7          v.π ← u;
8          DFS − VISIT(G, u);
9  u.color ← NERO ;           // visita completata
10 time ← time + 1;
11 u.f ← time
```

La variabile $time$ è una variabile globale che si utilizza per registrare le informazioni temporali. La procedura DFS opera nel seguente modo: le righe 2-4 colorano di bianco tutti i vertici e inizializzano i loro attributi π a NIL; la riga 5 azzera il contatore globale del tempo; le righe 6-8 controllano, uno alla volta, tutti i vertici in V e, quando trovano un vertice bianco, lo visitano utilizzando la procedura DFS-VISIT. Ogni volta che viene chiamata la procedura $DFS-VISIT(u)$ nella riga 7, il vertice u diventa la radice di un nuovo albero della foresta DF. Quando la procedura DFS termina, a ogni vertice u è stato assegnato un tempo di scoperta $u.d$ e un tempo di completamento $u.f$. In ogni chiamata di $DFS - VISIT(u)$, il vertice u è inizialmente bianco. La procedura DFS-VISIT svolge i seguenti passi: la riga 2 incrementa la variabile globale $time$; la riga 3 registra il nuovo valore di $time$ come il tempo di scoperta $u.d$ e la riga 3 colora di grigio u; le righe 5-8 ispezionano ogni vertice v adiacente a u e visitano in modo ricorsivo il vertice v, se è bianco; Infine, dopo che tutti i nodi che escono da u sono stati ispezionati, le righe 9-11 colorano di nero u, incrementano $time$ e registrano in $u.f$ il tempo di completamento della visita.

11.4.1 Analisi DFS

I cicli nelle righe 2-4 e 6-8 della procedura DFS impiegano un tempo $\Theta(|V|)$, escluso il tempo per eseguire le chiamate di DFS-VISIT. La procedura DFS-VISIT è chiamata esattamente una volta per ogni vertice $v \in V$, perché DFS-VISIT viene invocata soltanto se un vertice è bianco e la prima cosa che fa è colorare di grigio il vertice. Durante un'esecuzione di DFS-VISIT, il ciclo 5-8 viene eseguito un numero di volte pari alla somma delle lunghezze delle liste di adiacenza di ogni vertice, ovvero $\Theta(|E|)$. Il costo totale per eseguire le righe 5-8 di DFS-VISIT è $\Theta(|E|)$. Il tempo di esecuzione di DFS-VISIT è dunque $\Theta(|V|+|E|)$. L'occupazione in spazio è pari allo spazio richiesto per le chiamate ricorsive (o della pila nel caso si volesse scrivere la procedura iterativamente) e lo spazio dei campi aggiuntivi come la colorazione, inizio visita, fine visita, predecessore quindi il costo totale è pari a $\Theta(|V|)$

11.4.2 Proprietà e teoremi DFS

La più importante proprietà della visita in profondità è che il sottografo dei predecessori G_π forma una foresta di alberi, in quanto la struttura degli alberi DF rispecchia esattamente la struttura delle chiamate ricorsive di DFS-VISIT. La foresta contiene un albero per ogni vertice selezionato come sorgente. Altre proprietà sono che u è il padre di v nella foresta DF se e solo se v è stato scoperto esaminando la lista di adiacenza di u, e, v è un discendentedi v nella foresta DF se e solo se v è stato scoperto quando u era grigio. **Teorema delle parantesi**: per ogni coppia di vertici $u, v \in V$ si indica $I_u = [u.d, u.f]$ e $I_v = [v.d, v.f]$ allora è soddisfatta una sola delle seguenti condizioni:

- I due intervalli sono disgiunti $I_u \cap I_v = \emptyset$ e, u e v non sono discendenti l'uno dell'altro nella foresta DF
- L'intervallo I_v è contenuto in I_u, $I_v \subset I_v$ e, v è un discendente di u nella foresta DF

- L'intervallo I_u è contenuto in I_v, $I_u \subset I_v$ e, u è un discendente di v nella foresta DF

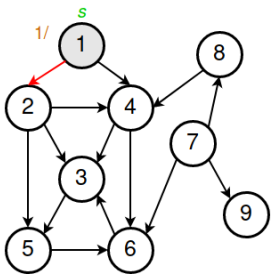
Per conseguenza del teorema si ha il corollario dell'**annidamento degli intervalli dei discendenti**: il vertice v è un discendente del vertice u nella foresta DF per un grafo G se e soltanto se $I_v \subset I_u$, con $v \neq u$. **Teorema del cammino bianco**: in una foresta DF di un grafo G, il vertice v è un discendente del vertice u se e soltanto se al tempo $u.d$, in cui viene scoperto u, il vertice v può essere raggiunto da u lungo un cammino che è formato esclusivamente da vertici bianchi. La visita DFS non aiuta a scoprire le distanze dei cammini minimi, ma la visita in profondità può essere utilizzata per la **classificazione degli archi** del grafo. Si suppone che G sia un grafo orientato, durante la visita DFS si possono definire quattro tipi di archi:

- **Archi d'albero**: sono gli archi nella foresta DF G_π. L'arco (u, v) è un arco d'albero se v viene scoperto la prima volta durante l'esplorazione di (u, v)
- **Archi all'indietro**: sono quei archi (u, v) che collegano un vertice u a un antenato v in un albero DF. Un arco (u, v) si può classificare come arco all'indietro quando durante l'ispezione si trova v grigio
- **Archi in avanti**: sono gli archi (u, v), diversi dagli archi d'albero, che collegano un vertice u a un discendente v in un albero DF. Un arco (u, v) si può classificare come arco in avanti quando durante l'ispezione si trova v nero e $u.d < v.d$, cioè v è stato scoperto dopo u
- **Archi trasversali**: sono quei archi (u, v) dove u e v non sono discendenti uno dell'altro. Un arco (u, v) si può classificare come arco trasversale quando durante l'ispezione si trova v nero e $v.d < u.d$, cioè v è stato scoperto prima di u

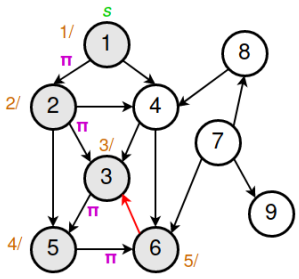
Il grafo G contiene un ciclo se e solo se G contiene almeno un arco all'indietro. Se durante l'esecuzione della visita si disegna la foresta DF è semplice vedere le varie relazioni. Nei grafi non orientati gli archi sono soltanto archi d'albero o archi all'indietro. Nel caso il grafo sia orientato si può modificare la procedura DFS-VISIT per far si che classifichi anche gli archi:

Algorithm 48: costo $\Theta(|E|)$

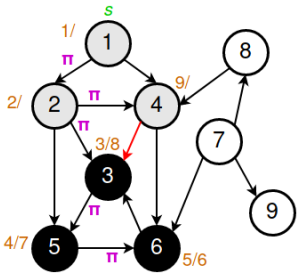
```
1  DFS-VISIT(G, u)
2  time ← time + 1;
3  u.d ← time;
4  u.color ← GRIGIO;
5  for v ∈ G.Adj[u] do                                   // ispeziona arco (u, v)
6      if v.color = BIANCO then
7          v.π ← u;
8          print "(u, v) arco d'albero";
9          DFS − VISIT(G, u);
10     else if v.color = GRIGIO then
11         print "(u, v) arco all'indietro"
12     else                                               // allora v è nero
13         if v.d < v.d then
14             print "(u, v) arco in avanti"
15         else
16             print "(u, v) arco trasversale"
17 u.color ← NERO ;                                       // visita completata
18 time ← time + 1;
19 u.f ← time
```



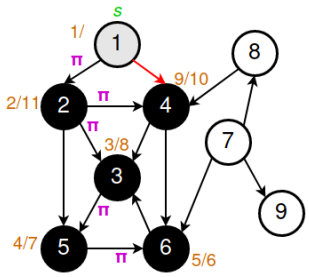
$(1, 2)$ è un arco d'albero perché porta verso un vertice bianco



$(6, 3)$ è un arco all'indietro perché porta a un vertice grigio



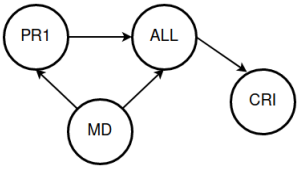
$(4, 3)$ è un arco trasversale perché 4 e 3 non hanno rapporti di discendenza diretti

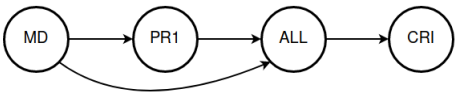


$(1, 4)$ è un arco in avanti perché 4 è nero ed è un discendente di 1

11.4.3 Ordinamento topologico

La visita in profondità può essere utilizzata per eseguire l'ordine topologico di un grafo **orientato aciclico** o semplicemente **dag** (*directed acyclic graph*). Un ordinamento topologico di un dag $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici tale che, se G contiene un arco (u, v), allora u appare prima di v nell'ordinamento. Un ordinamento topologico di un grafo può essere visto come un ordinamento dei suoi vertici lungo una linea orizzontale in modo che tutti gli archi orientati siano diretti da sinistra a destra, cioè permette di capire in che ordine poter eseguire determinati eventi rispettando tutti i vincoli, quindi l'ordinamento topologico è diverso dal problema usuale del *sorting*. Per esempio si illustra il caso di dover sostenere 4 esami con queste regole: *programmazione 1* deve essere sostenuto prima di *algoritmica*, *matematica discreta* blocca entrambe le due materie precedenti e infine per sostenere l'esame di *crittografia* si deve prima passare l'esame di algoritmica, quindi il grafo risultato e il rispettivo ordinamento topologico è il seguente:





L'ordinamento topologico è un ordinamento parziale ovvero non è unico perché possono esistere dei vertici che non sono tra di loro confrontabili. Per esempio se nell'esempio precedente si aggiunge l'esame di *ricerca operativa* con la regola che deve essere sostenuto prima di poter svolgere l'esame di *crittografia*, non esistono vincoli con gli altri esami allora l'ordinamento topologico non è unico. L'intuizione dell'ordinamento topologico è che esistono dei vertici che sono privi di archi uscenti, ovvero non vincola nessun altro vertice (come *crittografia*) e quindi si può mettere in fondo alla linea orizzontale. Gli altri vertici si posizioneranno nell'ordine inverso dei tempi di fine visita, questo perché quando si visita in profondità un vertice e quest'ultimo non ha più archi uscenti ancora da esplorare significa che si sono soddisfatti tutti i vincoli, e allora si può inserire il vertice nell'ordinamento.

Algorithm 49: costo $\Theta(|V| + |E|)$

```
1 TOPOLOGICAL-SORT(G)
2 chiama DFS(G) per calcolare i tempi di completamento v.f
3 una volta completata l'ispezione di un vertice,
4         si inserisce il vertice in testa a una lista
5 return lista dei vertici
```

È possibile eseguire un ordinamento topologico in $\Theta(|V| + |E|)$ perché la visita in profondità impiega un tempo $\Theta(|V|+|E|)$ e occorre un tempo $O(1)$ per inserire ciascuno dei $|V|$ vertici in testa alla lista. Per provare l'ordinamento topologico si deve dimostrare che per ogni $(u, v) \in E$, u precede v nella lista risultante. Siccome i nodi vengono inseriti in testa alla lista, u deve essere stato messo nella lista dopo v, ovvero $u.f > v.f$. Sia un arco generico $(u, v) \in E$, quando si ispeziona (u, v) esistono due casi:

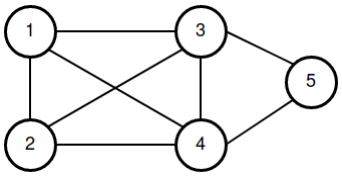
1. v è bianco allora (u, v) è un arco d'albero dunque v è discendente di u (è il figlio nella foresta DF), per costruzione la visita di v termina prima della visita di u, cioè $v.f < u.f$
2. v è nero, ciò significa che la visita di v è terminata mentre la visita di u è ancora in corso, allora per costruzione $v.f < u.f$

v non può essere grigio perché significherebbe avere un arco all'indietro e dunque avere un ciclo.

11.5 Esempi di problemi su grafi

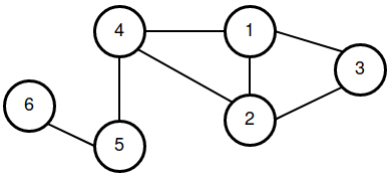
Tre dei più famosi problemi su grafi sono:

- **Clique**, dato in input un grafo $G = (V, E)$ e un intero $k > 0$ stabilisce se G contiene un sottografo completo di k nodi. La variante si chiama **max-clique** in cui dato un grafo si trova il sottografo completo più grande contenuto nel grafo. Questo problema non ha algoritmi efficienti di soluzione. La soluzione più efficiente conosciuta è il metodo della **forza bruta** dove si considerano tutti i possibili sottoinsiemi di k vertici e si controlla se si forma una clique oppure no. Non esistono nemmeno limiti inferiori che indicano qual è la soluzione migliore. Si dice che il problema della clique è un problema presumibilmente intrattabile ovvero probabilmente non esistono algoritmi efficienti per questo problema ma non si è riuscito a dimostrarlo.



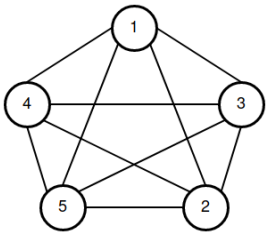
con $k = 3$ OK
con $k = 4$ OK
con $k = 5$ NO

- **Cammino hamiltoniano**, dato un grafo (si suppone non orientato) verificare se contiene un cammino semplice (o ciclo) che passa da tutti i vertici una sola volta. Anche per questo problema non si conoscono algoritmi più efficienti del metodo di forza bruta, cioè è un problema presumibilmente intrattabile

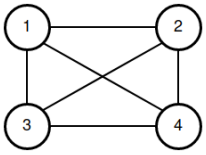


$<$6, 5, 4, 1, 3, 2 $>$ è un cammino hamiltoniano. Non esiste un ciclo hamiltoniano, cioè non si riesce a tornare al punto di partenza

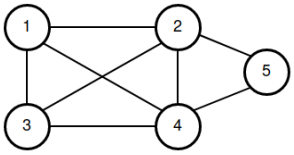
- **Ciclo euleriano**, trovare, se esiste, un percorso nel grafo che attraversa tutti gli archi una ed una sola volta, e torna al punto di partenza. Il problema è molto simile al cammino hamiltoniano, ma dal punto di vista computazionale è estremamente diverso perché questa volta la soluzione polinomiale (lineare) esiste. Eulero ha dimostrato che un grafo possiede un ciclo euleriano se tutti i vertici sono di grado pari e il grafo è connesso



presente ciclo euleriano



non è presente ciclo euleriano



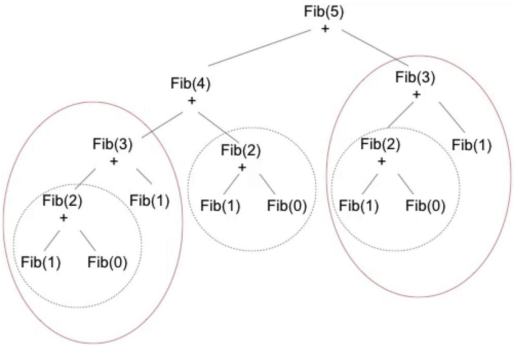
presente ciclo euleriano

Chapter 12

Programmazione dinamica

12.1 Paradigma della programmazione dinamica

La programmazione dinamica, come il metodo divide et impera, risolve i problemi combinando le soluzioni dei sotto-problemi. La programmazione dinamica può essere applicata quando i sottoproblemi non sono indipendenti, ovvero quando i sottoproblemi hanno in comune dei sottosottoproblemi. Un algoritmo di programmazione dinamica risolve ciascun sottosottoproblema una sola volta e salva la sua soluzione in una **tabella di programmazione dinamica**, evitando così il lavoro di ricalcolare la soluzione ogni volta che si presenta il sottosottoproblema. In questo contesto, un algoritmo divide et impera svolge molto più lavoro del necessario, risolvendo ripetutamente i sottoproblemi comuni. Per esempio i numeri di fibonacci (0, 1, 1, 2, 3, 5, 8, 13, ...)



Algorithm 50: costo $O(2^n)$

```
1 FIB(n)
2 if n ≤ 1 then
3     return n
4 else
5     return FIB(n − 1) + FIB(n − 2)
```

Alcune chiamate sono presenti più volte e questo rende l'algoritmo estremamente lento perché esso è chiamato a ripetere operazioni già eseguite sugli stessi dati. I sottoproblemi non sono indipendenti. Una soluzione più efficiente è quella iterativa dove si costruisce la successione dei numeri di Fibonacci fino al punto n, dove ciascun elemento è direttamente calcolato come la somma dei due precedenti. Poiché ogni elemento della successione si calcola in tempo costante mediante un'addizione, il tempo complessivo è di ordine $\Theta(n)$

Algorithm 51: costo $\Theta(n)$

```
1 FIB(n)
2 F : declare array n + 1 integers;
3 F[0] ← 0;
4 F[1] ← 1;
5 for i ← 2 to n do
6     F[i] ← F[i − 1] + F[i − 2]
7 return F[n]
```

L'array F è la tabella di programmazione dinamica. La programmazione dinamica tipicamente viene utilizzata in problemi di ottimizzazione, ovvero problemi per i quali possono esistere molte soluzioni che soddisfano i requisiti ma per i quali è richiesta la soluzione ottima, come per esempio il massimo o il minimo di una certa funzione. Si applica la programmazione dinamica quando la soluzione ottima si può ricavare dalle soluzioni ottime dei sotto-problemi. Un algoritmo di programmazione dinamica viene definito in quattro passi:

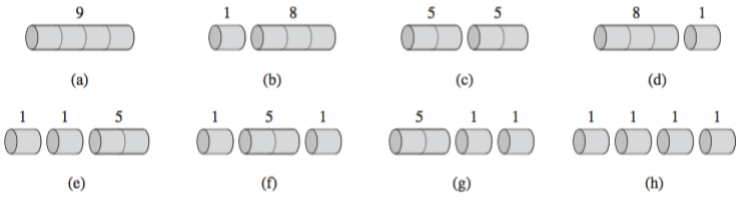
1. problema generale e sotto-problemi si definiscono nello stesso modo, ovvero devono avere la stessa struttura
2. definire i sottoproblemi elementari, cioè la soluzione diretta dei "casi base"
3. definire regola ricorsiva per il calcolo dei sottoproblemi a partire dai valori già calcolati

4. memorizzazione in tabella dei sottoproblemi

La programmazione dinamica di solito riduce il tempo di esecuzione ma aumenta l'utilizzo dello spazio.

12.1.1 Taglio della corda

Data una corda di lunghezza n centimetri e una tabella di prezzi p_i si vuole trovare il massimo guadagno r_n ottenibile tagliando la corda e vendendo i singoli pezzi. Le lunghezze delle corde sono sempre un numero intero. Nel caso in cui $n = 4$ i modi in cui può essere tagliata la corda sono:

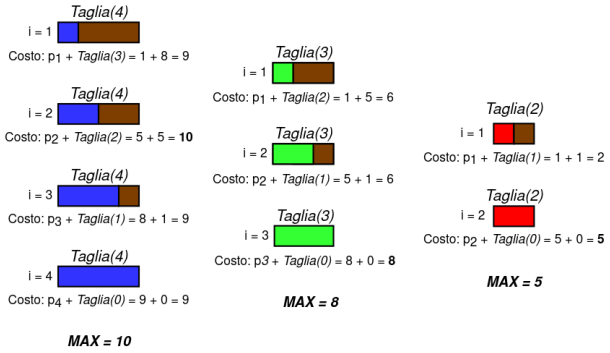


length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Per una corda di lunghezza 4, il ricavo ottimo è quello di tagliare la corda in due pezzi: $p_2 + p_2 = 10$. In generale, ci sono 2^{n-1} modi per tagliare una corda di lunghezza n. Se la soluzione ottima taglia la corda in k pezzi allora la decomposizione ottima $n = i_1 + i_2 + ... + i_k$ con guadagno $r_n = p_{i1} + p_{i2} + ... + p_{ik}$. Quando si taglia per la prima volta si hanno due pezzi lunghi i e $n - i$ quindi il guadagno derivato è r_i e r_{n-1}. Si devono considerare tutti i possibili valori di i ma sapendo che vi è anche il caso in cui la corda è venduta senza tagli. In generale $r_n = MAX(p_n, r_1+r_{n-1}, r_2+r_{n-2}, ..., r_{n-1}+r_1)$. Allora l'idea è di decomporre un primo pezzo a sinistra di lunghezza i, il secondo pezzo che resta a destra è di lunghezza $n - 1$. Si prova a dividere ulteriormente solo il pezzo a destra questo per ogni i, quindi $r_n = MAX\{p_i + r_{n-1}, 1 \leq i \leq n\}$, ovvero si cerca la soluzione solo a un sotto-problema. La seguente procedura implementa il calcolo in modo ricorsivo **top-down**, cioè parte dal problema grande fino ad arrivare al problema piccolo

Algorithm 52: costo $O(2^n)$

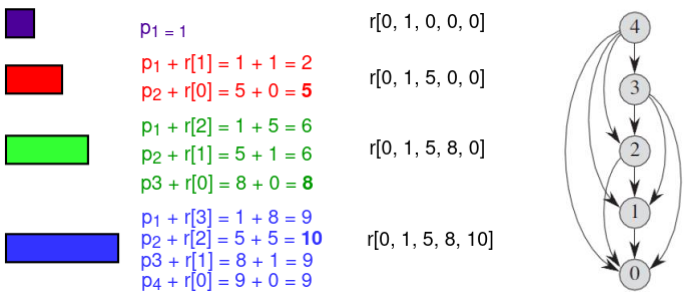
```
1  CUT-ROD(p, n)
2  if n = 0 then
3      return 0
4  q ← −∞;
5  for i ← 1 to n do
6      q ← MAX(q, p[i]+CUT−ROD(p, n−1))
7  return q
```



La procedura CUT-ROD chiama più e più volte sé stessa in modo ricorsivo con gli stessi valori dei parametri, ovvero risolve ripetutamente gli stessi sottoproblemi. L'idea, come fatto per Fibonacci, è partire dai valori che già si conoscono, cioè quelli più piccoli, e si memorizzano nella tabella di programmazione dinamica, poi si calcola via via quelli più grandi fino a trovare la soluzione del problema. Se si ha bisogno di nuovo della soluzione di un sotto-problema, si potrà riaverla immediatamente senza bisogno di ricalcolarla. Questo approccio è il **metodo bottom-up**

Algorithm 53: costo $O(n^2)$

```
1  BOTTOM-UP-CUT-ROD(p, n)
2  r : declare array n integers;
3  r[0] = 0;
4  for j ← 1 to n do
5      q ← −∞;
6      for i ← 1 to j do
7          if q < p[i] + r[j − 1] then
8              q ← p[i] + r[j − 1];
9      r[j] ← q
10 return r[n]
```



La riga 2 crea un array $r[0, ..., n]$ in cui salvare i risultati dei sotto-problemi. La riga 3 inizializza $r[0]$ a 0 in quanto una corda di lunghezza 0 non genera alcun ricavo. Le righe 4-7 risolvono ciascun sotto-problema di dimensione j per $j = 1, 2, ..., n$, nell'ordine delle dimensioni crescenti. L'approccio utilizzato per risolvere un problema di una particolare dimensione j è uguale a quello utilizzato da CUT-ROD. ad eccezione della riga 7, che adesso fa riferimento direttamente all'elemento $r[j-1]$ dell'array, anziché fare una chiamata ricorsiva per risolvere il sotto-problema di dimensione $j-1$. La riga 8 salva in $r[j]$ la soluzione del sotto-problema di dimensione j. Infine, la riga 8 restituisce $r[n]$, che è uguale al valore ottimo r_n. Il tempo esponenziale della soluzione ricorsiva diventa così polinomiale. BOTTOM-UP-CUT-ROAD restituisce il valore ottimo ma non indica dove tagliare la corda. Per sapere il punto dove tagliare la corda occorre utilizzare un secondo array $s[0, ..., n]$ per ricordarsi i valori di i che restituiscono il massimo. Per recuperare la soluzione ottima si stampa la lista completa delle dimensioni dei pezzi per una decomposizione ottima di un'asta di lunghezza n.

Algorithm 54: costo $O(n^2)$

```
1  EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  r : declare array n integers;
3  s : declare array n integers;
4  r[0] = 0;
5  for j ← 1 to n do
6      q ← −∞;
7      for i ← 1 to j do
8          if q < p[i] + r[j − 1] then
9              q ← p[i] + r[j − 1];
10             s[j] ← i;
11     r[j] ← q
12 return r and s
```

Algorithm 55: costo $O(n^2)$

```
1  PRINT-CUT-ROD-SOLUTION(p, n)
2  (r, s) ←
     EXTENDED − BOTTOM − UP − CUT − ROD(p, n);
3  while n > 0 do
4      print s[n];
5      n ← n − s[n]
```

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

12.1.2 Longest common subsequence

Una sequenza P è una **sottosequenza** di T se P è ottenuta da T rimuovendo uno o più dei suoi elementi. Esempio: *P = "AAATA", T = "**AAA**AT**T**G**A***. La sequenza vuota è sottosequenza di ogni altra sequenza. Una sequenza X è una **sottosequenza comune** di due sequenze T, U, se è sottosequenza sia di T che di U. Una sequenza X è una **sottosequenza comune massimale** di due sequenze T, U, se non esiste un'altra sottosequenza comune Y tale che Y sia più lunga di X. Esempio:

T	=	"AAAATTGA"	"A"	*sottosequenza comune*
U	=	"TTACGATA"	"AAGA"	*sottosequenza comune*
			"AAATA"	*sottosequenza comune massimale*

La soluzione per *forza bruta* è controllare per ogni sotto-sequenza di T se è sotto-sequenza di U. Data una sequenza T lunga n, le sottosequenze sono 2^n, per verificare se una sequenza è sottosequenza di un'altra costa $O(m+n)$, quindi la complessità computazione di LSC è $\Theta(2^n(m+n))$. Si utilizza approccio ricorsivo con questa intuizione: data una sequenza T composta dai caratteri $t_1 t_2 ... t_n$, $T(i)$ denota il prefisso $i - esimo$ dei primi caratteri, ovvero $T(i) = t_1 t_2 ... t_i$. Per esempio se $T = "ABDCCABD"$ allora $"T(0) = ""$ e $T(3) = "ABD"$. Il caso base dell'algoritmo ricorsivo sarà quando la stringa è vuota allora la sottosequenza comune di entrambe le stringhe è la stringa vuota. Per i casi ricorsivi si considerano due prefissi $T(i)$ e $U(j)$ tali per cui:

- l'ultimo loro carattere coincide $t_i = u_j$, per esempio $T(i) = "ALBERTO"$ e $U(j) = "PIERO"$, in questo caso di sicuro la sottosequenza comune massimale conterrà l'ultimo carattere, quindi $LSC(T(i), U(j)) = LCS(T(i-1), U(j-1)) + t_i$, cioè $LSC("ALBERTO", "PIERO") = LCS("ALBERT", "PIER") + "O"$
- l'ultimo loro carattere è differente $t_i \neq u_j$, per esempio $T(i) = "ALBERT"$ e $U(j) = "PIER"$, in questo caso si prova togliendo l'ultimo carattere di T oppure di U, perché sicuramente una delle due non serve e si prende la più lunga sottosequenza, quindi $LSC(T(i), U(j)) = longest(LCS(T(i-1), U(j)), LCS(T(i), U(j-1)))$, cioè $LSC("ALBERT", "PIER") = longest(LCS("ALBER", "PIER"), LCS("ALBERT", PIE))$

Mettendo tutto insieme: date due sequenze T e U di lunghezza n e m, la formula ricorsiva $LCS(T(i), U(j))$ restituisce la LCS dei prefissi $T(i)$ e $U(j)$

$$LCS(T(i), U(j)) = \begin{cases} \emptyset & i = 0 \textbf{ or } j = 0 \\ LCS(T(i-1), U(j-1)) + t_i & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i = u_j \\ longest(LCS(T(i-1), U(j)), LCS(T(i), U(j-1)) & i > 0 \textbf{ and } j > 0 \textbf{ and } t_i \neq u_j \end{cases}$$

Utilizzando l'equazione precedente si potrebbe scrivere facilmente un algoritmo ricorsivo con tempo esponenziale per calcolare la lunghezza di una LCS di due sequenze. Tuttavia poiché ci sono soltanto $\Theta(mn)$ sottoproblemi distinti, si può utilizzare la programmazione dinamica per calcolare le soluzioni con un metodo bottom-up. La procedura LCS-LENGHT riceve come input due sequenze $T = t_1 t_2 ... t_m$ e $U = u_1 u_2 ... u_n$ e memorizza i valori $L[i, j]$ in una tabella $L[0, ...m, 0, ..., n]$ dove vengono inseriti i valori della prima riga di L da sinistra a destra, poi vengono inseriti i valori nella seconda riga e così via. La procedura utilizza anche una tabella $indice[1, ...m, 1, ...n]$ per semplificare la costruzione di una soluzione ottima. Intuitivamente $indice[i, j]$ punta alla posizione della tabella che corrisponde alla soluzione ottima del sottoproblema che è stata scelta per calcolare $L[i, j]$. La procedura restituisce le tabelle L e $indice$. La posizione $L[m, n]$ contiene la lunghezza di una LCS di T e U:

Algorithm 56: costo $\Theta(mn)$

```
1  LCS-LENGHTH(T, U)
2  for i ← 1 to m do
3      L[i][0] = 0
4  for j ← 0 to n do
5      L[0][j] = 0
6  for i ← 1 to m do
7      for j ← 1 to n do
8          if t_i = u_j then
9              L[i][j] ← L[i − 1][j − 1] + 1;
10             indice[i, j] ← "" ↖ ""
11         else if L[i − 1][j] > L[i][j − 1] then
12             L[i][j] ← L[i − 1][j];
13             indice[i, j] ← "" ↑ ""
14         else
15             L[i][j] ← L[i][j − 1];
16             indice[i, j] ← "" ← ""
17 return L and indice
```

		0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

La tabella L della procedura LCS-LENGHT può essere utilizzata per costruire rapidamente una LCS: si inizia da $indice[m, n]$ e si attraversa la tabella seguendo le frecce. Quando si incontra un ↖ in $indice[i, j]$ significa che $t_i = u_j$ è un elemento della LCS trovata da $LCS - LENGHT$. In questo modo gli elementi della LCS si incontrano in ordine inverso. La seguente procedura ricorsiva stampa una LCS nell'ordine corretto:

Algorithm 57: costo $O(m + n)$

```
1  PRINT-LCS(indice, T, i, j)
2  if i = 0 or j = 0 then
3      return
4  if L[i, j] = "" ↖ "" then
5      PRINT − LCS(indice, T, i − 1, j − 1);
6      print x_i
7  else if L[i, k] = "" ↑ "" then
8      PRINT − LCS(indice, T, i − 1, j)
9  else
10     PRINT − LCS(indice, T, i, j − 1)
```

Il codice può essere migliorato eliminando completamente la tabella $indice$ perché ogni posizione $L[i, j]$ dipende solo da $L[i - 1, j - 1], L[i - 1, j]$ e $L[i, j - 1]$. Dato il valore $L[i, j]$ si può determinare in tempo costante quale dei tre valori è stato utilizzato per calcolare $L[i, j]$ senza ispezionare la tabella $indice$, quindi si risparmia ma in ogni caso non si è ridotta asintoticamente lo spazio poiché si ha sempre bisogno della tabella L

12.1.3 Edit distance

L'**edit distance** è il numero minimo di operazioni di editing che trasformano x in y, dove le operazioni di editing sono sostituzione, cancellazione e inserzione di un carattere. Si vuole trovare l'allineamento ottimo cioè quello a distanza minima che totalizza il minor numero di errori:

- **mismatch**: il confronto fra due caratteri è errato

- **inserzione**: un carattere di y viene confrontato con un carattere vuoto di x
- **cancellazione**: un carattere di x viene confrontato con un carattere vuoto di y

Tutti e tre gli errori hanno peso uno. Esempio:

x	=	A	L	B	E	R	O
y	=	L	A	B	B	R	O
		1	1	=	1	=	=

La distanza, ovvero la somma del numero degli errori, è 3. Si può tentare con un diverso allineamento:

x	=	-	A	L	B	E	R	O
y	=	L	A	B	B	-	R	O
		1	=	1	=	1	=	=

anche in questo caso la distanza è 3. Indicate le sequenze come $x = x_1x_2...x_n$ e $y = y_1y_2...y_m$, il calcolo impiega una matrice M di dimensioni $(n+1) \times (m+1)$ dove $M[i, j]$ indica la edit distance tra il prefisso $x = x_1x_2...x_i$ di X e il prefisso $y = y_1y_2...y_j$ di Y. Dunque i caratteri di x corrispondo alle righe di M, i caratteri di y alle colonne, e l'ultimo valore $M[n, m]$ indica la edit distance tra le due sequenze. I sottoproblemi elementari sono: $E(0, 0) = 0$ perché vi sono due stringhe vuote; $E(i, 0)$ si confronta x con la stringa vuota di y quindi si collezionano i errori; $E(0, j)$ stesso procedimento cioè si collezionano j errori; ora si conosce come riempire la prima riga e la prima colonna di M che si producono rispettivamente in $\Theta(n)$ e $\Theta(m)$. Si conoscono i valori delle stringhe fino a $1...i - 1$ e $1...j - 1$, per riempire la casella generica $M[i, j]$ partendo dai valore già calcolati esistono 3 modi possibili in cui l'allineamento può terminare:

- **Termina con una sostituzione**

x_1	x_2	...	x_{i-1}	$\boldsymbol{x_i}$
y_1	y_2	...	y_{j-1}	$\boldsymbol{y_j}$

  si definisce una funzione $p[x_i, y_j]$ che vale 0 se $x_i = y_i$ o 1 se $x_i \neq y_i$, allora il costo equivale a $M[i, j] = p[x_i, y_j] + M[i - 1, j - 1]$

- **Termina con una cancellazione**

x_1	x_2	...	x_{i-1}	$\boldsymbol{x_i}$
y_1	y_2	...	y_j	$-$

  costo: $M[i, j] = 1 + M[i - 1, j]$

- **Termina con un inserimento**

x_1	x_2	...	x_i	$-$
y_1	y_2	...	y_{j-1}	$\boldsymbol{y_j}$

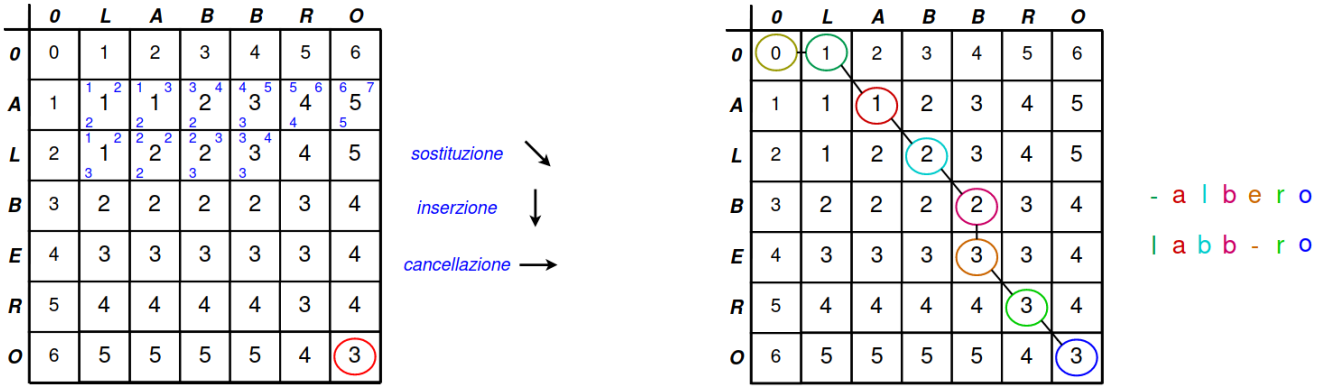
  costo: $M[i, j] = 1 + M[i, j - 1]$

Per riempire tutte le caselle si impiega $\Theta(n^2)$. Ora si può calcolare la formula ricorsiva $M[i, j] = min\{p[x_i, y_j] + M[i - 1, j - 1], 1 + M[i - 1, j], 1 + M[i, j - 1]\}$. La formulazione dell'algoritmo fa uso di due vettore x e y che contengono le sequenze da confrontare:

Algorithm 58: costo $\Theta(nm)$

```
1  ED(X, Y)
2  for i ← 0 to n do                                   // inizializza la colonna 0
3      M[i, 0] = i
4  for j ← 0 to m do                                   // inizializza la riga 0
5      M[0, j] = j
6  for i ← 1 to n do
7      for j ← 1 to m do
8          if x[i] = y[j] then
9              p ← 0
10         else
11             p ← 1
12         M[i, j] = min(p[x_i, y_j] + M[i − 1, j − 1], 1 + M[i − 1, j], 1 + M[i, j − 1])
13 return M[n, m]
```

Esempio:



Gli allineamenti ottimi che danno luogo all'edit distance si ricostruiscono risalendo all'indietro dalla casella soluzione $M[n,m]$ fino alla casella $M[0,0]$. Possono esistere più percorsi all'indietro, ovvero avere diversi allineamenti ottimi. La lunghezza dell'allineamento l è $max\{n,m\} \leq l \leq n+m$. Lo pseudocodice dell'allineamento ottimo tra due sequenze contenute nei vettori x e y di dimensioni n e m, il valore di h restituito dall'algoritmo indica da che punto interpretare il contenuto dei vettori ALX, ALY

Algorithm 59: costo $O(n+m)$

```
1  ALLINEA(x, y, M)
2  k ← n + m ;                                   // numero di caselle massime che si devono riempire
3  h ← k ;                                                                  // la k-esima casella
4  i ← n, j ← m;
5  while i > 0 or j > 0 do
6      if ((i > 0 and j > 0) and (M[i, j] = M[i − 1, j − 1]) and (x[i] = y[j])) or
7      ((M[i, j] = M[i − 1, j − 1] + 1) and (x[i] ≠ y[j])) then
8          ALX[h] ← x[i], ALY[h] ← y[j] ;                                        // diagonale ↖
9          i ← i − 1, j ← j − 1
10     else
11         if j > 0 and M[i, j] ← M[i, j − 1] + 1 then
12             ALX[h] ← " − ", ALY[h] ← y[j] ;                                      // sopra ↑
13             j ← j − 1
14         else
15             ALX[h] ← x[i], ALY ← " − " ;                                     // sinistra ←
16             i ← i − 1
17     h ← h − 1
18 return h
```

12.1.4 Problema dello zaino

Il problema dello zaino, è un problema di ottimizzazione, che prende in input un massimo valore trasportabile W e n oggetti $A = \{a_1, a_2, ..., a_n\}$ ognuno con un valore v_i e un peso w_i. Il problema restituisce in output il sottoinsieme $S \subseteq A$ che rappresenta il carico più prezioso che si può trasportare nello zaino, ovvero il carico di valore massimo purché la somma dei loro pesi sia inferiore a quella massima trasportabile. In formula $\sum_{i=1}^{n} w_i \leq W_i$ **and** $\sum_{i=1}^{n} max\ v_i$. Una possibile strategia è la **l'algoritmo greedy** si ordinano gli oggetti in ordine decrescente di rendimento v_i/p_i, e si selezionano finché entrano nello zaino

W = 8

	w	v	v/w
a_1	5	10	2
a_2	4	6	1.5
a_3	4	5	1.25

si prende $a_1 \Rightarrow S = \{a_1\}, W = 8 - 5 = 3$

Non si può prendere nient'altro,
S non è la soluzione ottima

La complessità è $\Theta(nlogn)$ per ordinare i v_i e un tempo costante per selezionare il valore. Si studia una variante il **problema dello zaino frazionario**, dove anziché inserire nello zaino solo parti intere, si può prendere anche frazioni di ogni a_i. Questa variante garantisce l'ottimo, e al massimo un solo oggetto è frazionato (l'ultimo inserito). Si risolve il problema dello zaino con la programmazione dinamica: dato uno zaino di capacità W e n oggetti caratterizzati

da peso w e profitto p, si definisce una matrice $C[i][j]$ come il massimo profitto che può essere ottenuto dai primi $i \leq n$ oggetti contenuti in uno zaino di capacità $j < W$. Il massimo profitto ottenibile è rappresentato da $C[n][W]$. I sottoproblemi elementari sono quelli in cui $j = 0$ e $i = 0$, cioè si ha un peso nullo trasportabile e quando non si hanno oggetti, in entrambi i casi ottiene 0 come valore. Per la composizione ricorsiva si calcola il generico $C[i][j]$ e si considera l'ultimo elemento:

- Se $w_i > j$ non lo si prende $C[i][j] = C[i-1][j]$, la capacità non cambia, non c'è profitto e si ha un oggetto in meno
- Se $w_i \leq j$ lo si prende $C[i][j] = C[i-1][j-w_i] + v_i$, si sottrae il peso dalla capacità e si aggiunge il valore relativo

La soluzione migliore la si sceglie come $C[i][j] = max\{C[i-1][j], C[i-1][j-w_i] + v_i\}$.

Algorithm 60: costo $O(n \times W) = O(n \times 2^k)$

```
1  ZAINO(n, v[], w[], W)
2  for j ← 0 to W do                                   // no oggetti (1 riga matrice)
3      C[0, j] ← 0
4  for i ← 1 to n do                                   // no zaino (1 colonna matrice)
5      C[i, 0] ← 0
6  for i ← 1 to n do
7      for j ← 1 to W do
8          if w[i] ≤ c then
9              C[i][j] = max(C[i − 1][j], C[i − 1][j − w[i]] + v[i])
10         else
11             C[i][j] = C[i − 1][j]
12 return C[n, W]
```

	w	v
a_1	5	10
a_2	4	6
a_3	4	5

	C								
	i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
	0	0	0	0	0	0	0	0	0	0
a_1	*1*	0	0	0	0	0	10	10	10	10
a_2	*2*	0	0	0	0	6	10	10	10	10
a_3	*3*	0	0	0	0	6	10	10	10	11

S = {a_2, a_3}
valore 11

max(C[2,8], C[2, 8-4] +5)
max(10, 6+5) = 11

L'algoritmo è **pseudo-polinomiale** perché sono necessari $k = logW$ bit per rappresentare W e quindi la complessità è $O(n \times 2^k)$

Chapter 13

Teoria della calcolabilità e complessità

13.1 Classificazione problemi

La **teoria della calcolabilità** è una branca dell'informatica teorica che si occupa delle questione fondamentali sulla **potenza** e i **limiti** dei sistemi di calcolo. I problemi oggetto di studio sono i **problemi computazionali**, ovvero, problemi formulati matematicamente di cui si cerca una soluzione algoritmica. La calcolabilità classifica i problemi computazionali in base al fatto che possono essere risolti o meno da un algoritmo:

- **problemi non decidibili**
- **problemi decidibili**

La **teoria della complessità** si occupa di soli problemi computazionali decidibili e li classifica in:

- **problemi trattabili**, possono essere risolti da algoritmi efficienti di costo polinomiale
- **problemi intrattabili**, possono essere risolti solo da algoritmi di costo esponenziale

La dimostrazione dell'esistenza di problemi che non ammettono un algoritmo di risoluzione viene dal fatto che ci sono molti più problemi che algoritmi. Un algoritmo è una sequenza finita di operazioni, che indipendentemente dal modello che si sceglie per la formulazione, si rappresenta con sequenze finite di caratteri di un alfabeto finito, ovvero gli algoritmi sono possibilmente infiniti, ma numerabili. I problemi computazionali non sono numerabili, e quindi esistono problemi privi di un corrispondente algoritmo di calcolo. La teoria della calcolabilità dipende dal modello di calcolo? I linguaggi di programmazione esistenti sono tutti equivalenti? È possibile che problemi oggi irrisolvibili possano essere risolti in futuro con altri linguaggi o calcolatori? Queste sono tutte domande per le quali non si conosce una risposta certa o una risposta sotto forma di teorema, ma si conosce la **tesi di Church-Turing**: tutti i ragionevoli modelli di calcolo risolvono esattamente la stessa classe di problemi e dunque si equivalgono nella possibilità di risolvere problemi, pur operando con diversa efficienza. Questa tesi porta a dire che la decidibilità è una proprietà del problema, e quindi non dipende dal modello di calcolo, incrementi qualitativi alla struttura di una macchina servono solo a abbassare il tempo d'esecuzione e/o rendere più agevole la programmazione

13.2 Il problema dell'arresto

Il problema dell'arresto è un problema posto in forma decisionale (ovvero problemi la cui risposta può essere solo *True/False*) ed è formulato in questo modo: *preso un algoritmo arbitrario A e un insieme di dati di input D, decidere in **tempo finito** se la computazione di A su D termina o no* (supponendo di non avere limiti di tempo e memoria). Il problema dell'arresto è un problema che indaga sulle proprietà di un altro algoritmo trattato come dato in input, ed è legittimo, perché si può usare lo stesso alfabeto per codificare algoritmi e i loro dati in ingresso. Turing ha dimostrato che riuscire a dimostrare se un programma arbitrario si arresta e termina in tempo finito la sua esecuzione è in generale impossibile allora il problema dell'arresto è un problema indecidibile. *Dimostrazione:* se il problema dell'arresto fosse decidibile, allora esisterebbe un algoritmo *arresto* che, presi A e D come dati in input, risponde in tempo finito $ARRESTO(A, D) = 1$ se $A(D)$ termina, altrimenti, $ARRESTO(A, D) = 0$ se $A(D)$ non termina. Si può scegliere $D = A$, cioè considerare la computazione $A(A)$. Se esistesse l'algoritmo ARRESTO, esisterebbe l'algoritmo $PARADOSSO(A)$ che prende in input un algoritmo generico A:

```
PARADOSSO(A)
WHILE(ARRESTO(A,A)){ ; }
```

consiste in un ciclo la cui guardia è la chiamata dell'algoritmo ARRESTO e il corpo del while non ha nessuna operazione. Se la guardia è vera, l'algoritmo non termina, ed essa è vera quando $ARRESTO(A, A) = 1$. Se la guardia è falsa, PARADOSSO termina immediatamente. Se si calcola $PARADOSSO(PARADOSSO)$ questo termina se e solo se $ARRESTO(PARADOSSO, PARADOSSO) = 0$ e ciò è equivalente a dire che $PARADOSSO(PARADOSSO)$ non termina, quindi si ha una contraddizione. L'unico modo di risolvere la contraddizione è che l'algoritmo PARADOSSO non possa esistere, dunque non può esistere nemmeno l'algoritmo ARRESTO, e questo dimostra che il problema dell'arresto è indecidibile.

13.3 Problemi intrattabili

Il problema intrattabile è un problema per il quale non esiste un algoritmo con complessità polinomiale in grado di risolverlo. La **Torre di Hanoi** è un quesito matematico composto da tre paletti e un certo numero di dischi di grandezza decrescente, che possono essere infilati in uno qualsiasi dei paletti. Lo scopo è portare tutti i dischi su un paletto diverso, potendo spostare solo un disco alla volta e potendo mettere un disco solo su un altro disco più grande, mai su uno più piccolo. La soluzione base si formula in modo ricorsivo. Siano i paletti etichettati con A, B, C, e i dischi numerati da 1 (il più piccolo) a n (il più grande). L'algoritmo si esprime come segue:

1. Sposta i primi $n-1$ dischi da A a B (questo lascia il disco n da solo sul paletto A)
2. Sposta il disco n da A a C
3. Sposta $n-1$ dischi da B a C

Algorithm 61: costo $O(2^n)$

```
1 HANOI(n, A, B, C)
2 if n = 1 then
3     print A → C
4 else
5     HANOI(n − 1, A, B, C);
6     print A → C;
7     HANOI(n − 1, B, C, A);
```

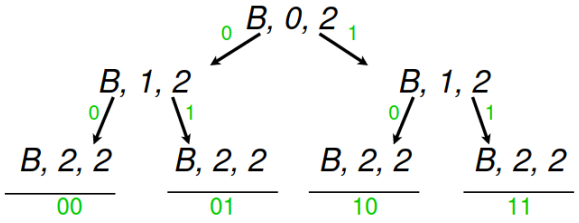
L'equazione di ricorrenza corrisponde a $T(n) = \begin{cases} 1 & se\ n = 1 \\ 2T(n-1)+1 & se\ n > 1 \end{cases}$

Con il metodo della sostituzione si arriva a $T(n) = 2^n - 1$, cioè questo algoritmo svolge un numero esponenziale di mosse. **Generazione delle sequenze binarie**, cioè tutte le configurazioni binarie di n bit, questo è un problema importante perché le sequenze binarie permettono di rappresentare sottoinsiemi. Si immagina di avere un insieme di n oggetti $A = \{a_0, a_1, ..., a_{n-1}\}$ e un sottoinsieme $S \subseteq A$, si può descrivere S con un array $B_S(i) = \begin{cases} 0 & se\ a_i \notin S \\ 1 & se\ a_i \in S \end{cases}$.

Durante la generazione delle 2^n sequenze binarie, si memorizza ciascuna sequenza binaria B, e si utilizza la procedura ELABORA per stampare B o per elaborare il corrispondente sottoinsieme. B viene riutilizzato ogni volta sovrascrivendone il contenuto ricorsivamente. Il seguente codice ricorsivo, con n dimensione di B e i la posizione di B in cui scrivere, permette di ottenere tutte le 2^n sequenze binarie di lunghezza n.

Algorithm 62: costo $\Theta(2^n \times costo(ELABORA))$

```
1 GENERABINARIE(B, i, n)
2 if i = n then
3     ELABORA(B, n)
4 else
5     B[i] = 0;
6     GENERABINARIE(B, i + 1, n);
7     B[i] = 1;
8     GENERABINARIE(B, i + 1, n)
```



Si simula l'algoritmo invocando la procedura con $GENERABINARIE(B, 0, 2)$, cioè si vogliono generare le sequenze di 2 bit. Il **problema dello zaino** con l'algoritmo di programmazione dinamica ha una soluzione esponenziale quindi chiaramente inefficiente. Si studia un algoritmo alternativo che è sempre esponenziale ma in alcuni casi, a seconda dei valori di n e di W può essere una variante più efficiente all'algoritmo di programmazione dinamica

Algorithm 63: costo $\Theta(2^n \times n)$

```
1 ZAINO(v, w, W)
2 vmax ← 0 ;                                          // conterrà valore carico più prezioso
3 sol : declare array n integers ;      // conterrà sequenza binaria che descrive carico più prezioso
4 B : declare array n integers;
5 GENERABINARIE(B, 0, n, vmax, sol, W, v, w) ;          // vmax viene passato per riferimento
6 return vmax and sol
```

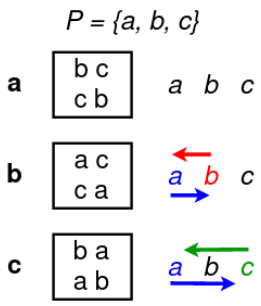
Dopo l'esecuzione di GENERABINARIE, $vmax$ e sol contengono la soluzione del problema. Nella procedura GENERABINARIE cambia solo la procedura elabora:

Algorithm 64: costo $\Theta(n)$

```
1  ELABORA(B, n, vmax, sol, v, w, W)
2  peso ← 0;
3  valore ← 0;
4  for i ← 0 to n do
5      if B[i] = 1 then
6          peso ← peso + w[i];;
7          valore ← valore + v[i]
8  if peso ≤ W  and  valore > vmax then
9      vmax ← valore;
10     for j ← 0 to n do
11         sol[j] ← B[j]
```

Generazione delle permutazioni, dato un array P di n elementi, si costruisce un algoritmo ricorsivo che genera in P le $n!$ permutazioni. L'algoritmo si basa sull'idea di creare ricorsivamente:

- $(n-1)!$ permutazioni che iniziano con $P[0]$
- $(n-1)!$ permutazioni che iniziano con $P[1]$
- ...
- $(n-1)!$ permutazioni che iniziano con $P[n-1]$



Algorithm 65: costo $\Theta(n! \times costo(ELABORA)))$

```
1 GENERAPERMUTAZIONI(P, i, n)
2 if i = n − 1 then
3     ELABORA(P, n)
4 else
5     for j ← i to n do
6         swap P[i] with P[j];
7         GENERAPERMUTAZIONI(P, i + 1, n);
8         swap P[i] with P[j] ;                          // ripristina l'ordine iniziale
```

13.4 Le classi P e NP

La maggior parte dei problemi incontrati fin'ora (*ordinamento, ricerca di chiavi, alberi, grafi...*) possono essere risolti con algoritmi polinomiali ed essi vengono chiamati *problemi trattabili*. Esistono altri problemi (*clique, cammino hamiltoniano, zaino, ...*) il cui stato computazionale non è chiaro, ovvero si hanno a disposizione solo algoritmi di costo esponenziale e non si hanno limiti inferiori per dimostrare che non possono esistere algoritmi di costo polinomiale. Questi problemi si chiamano **problemi presumibilmente intrattabili**. Dato un problema Π si definisce I l'insieme delle istanze in ingresso e S l'insieme delle soluzioni. Per i *problemi decisionali* si indica con *istanze positive*, le istanze di input per cui la risposta del problema è *true*, e *istanze negative* quando la risposta è *false*. Per i *problemi di ricerca*, data un'istanza x si restituisce una soluzione s. Per i *problemi di ottimizzazione*, data un'istanza x, si vuole trovare la *migliore soluzione s* tra tutte le soluzioni possibili. La teoria della complessità è definita principalmente in termini di problemi di decisione, questo perché:

- essendo la risposta binaria, tutto il tempo è speso esclusivamente per il calcolo e non per la restituzione del risultato
- la difficoltà di un problema è già presente nella sua versione decisionale. Qualsiasi problema di ottimizzazione lo si può formulare come un problema decisionale chiedendo l'esistenza di una soluzione che soddisfi una certa proprietà. La versione decisionale non può essere più difficile della versione di ottimizzazione, perché si sa risolvere il problema di ottimizzazione allora si sa risolvere immediatamente anche il problema decisionale. Se si trova il costo computazionale per il problema decisionale, questo costo rappresenterà un limite inferiore per il problema di ottimizzazione

Dato un problema decisionale Π e un algoritmo A, si dice che A risolve Π se, data un'istanza di input x allora $A(x) = 1 \Leftrightarrow \Pi(x) = 1$. L'algoritmo A risolve Π in tempo $t(n)$ e spazio $s(n)$ se il tempo di esecuzione e l'occupazione di memoria di A sono rispettivamente $t(n)$ e $s(n)$. Data una qualunque funzione $f(n)$ si indica con la classe **Time(f(n))** l'insieme dei problemi decisionali che possono essere risolti in tempo $O(f(n))$, in modo analogo si definisce la classe **Space(f(n))**. Si definisce:

- **Classe P**, la classe dei problemi risolvibili in tempo polinomiale nella dimensione n dell'istanza di ingresso
- **Classe PSPACE**, la classe dei problemi risolvibili in spazio polinomiale nella dimensione n dell'istanza di ingresso
- **Classe ExpTime** la classe dei problemi risolvibili in tempo esponenziale nella dimensione n dell'istanza di ingresso

$P \subseteq PSPACE$, $PSPACE \subseteq ExpTime$, non è noto ad oggi se le inclusioni siano proprie. I problemi dello zaino, della clique, e del cammino hamiltoniano sono tutti problemi per cui si conosce solo una soluzione esponenziale, e quindi si potrebbero mettere nella *classe ExpTime* ma non si sa se possono rientrare all'interno di *P*. In un problema decisionale si è interessati a verificare se una istanza del problema soddisfa una certa proprietà. Per alcuni problemi, le istanze che possiedono la proprietà possono fornire dei **certificati**, cioè delle informazioni aggiuntive, che permettono di verificare se l'istanza ha la proprietà cercata in tempo polinomiale. In poche parole è come se si avesse già la soluzione che soddisfa certe proprietà di un determinato problema e la si debba solo controllare, questo procedimento lo si può fare in tempo polinomiale. I certificati si definiscono solo per le istanze accettabili ed in generale non è facile costruire certificati di non esistenza. Chi non ha un certificato, può individuarla in tempo esponenziale considerando tutti i casi possibili con il metodo della forza bruta. Un certificato per il problema clique è un sottoinsieme di k vertici che forma la clique. L'idea è utilizzare il costo della verifica di un certificato per un'istanza accettabile per caratterizzare la complessità del problema stesso. Un problema Π è **verificabile in tempo polinomiale** se

- ogni istanza accettabile x di Π di lunghezza n ammette un certificato y di lunghezza polinomiale in n
- esiste un algoritmo di verifica polinomiale in n e applicabile a ogni coppia $< x, y >$, che permette di attestare che x è accettabile

Si definisce la **classe NP** come la classe dei problemi decisionali che possono essere verificati in tempo polinomiale, dove la N sta per tempo polinomiale *non deterministico*. Naturalmente la classe P è incluso in NP perché ogni problema in P ammette un certificato verificabile in tempo polinomiale: si esegue l'algoritmo che risolve il problema per costruire il certificato. Le due classi P e NP sono uguali o diverse? In altre parole risolvere e verificare sono la stessa cosa? Si pensa che sono diverse ma non lo si è dimostrato. Con la congettura $P \neq NP$, si cerca di individuare i problemi più difficili all'interno della classe NP che prendono il nome di **problemi NP-completi**. Se esistesse un algoritmo polinomiale per risolvere uno solo di questi problemi, allora tutti i problemi in NP potrebbero essere risolti in tempo polinomiale, e dunque $P = NP$. Per definire i problemi NP-completi si ha bisogno della *riduzione polinomiale*, ovvero si converte un problema in un altro: dati due problemi decisionali Π_1 e Π_2, e siano I_1 e I_2 insiemi delle istanze di input di Π_1 e Π_2, si dice che Π_1 si riduce in tempo polinomiale a Π_2 (notazione $\Pi_1 \leq_p \Pi_2$) se esiste una funzione $f : I_1 \to I_2$ calcolabile in tempo polinomiale tale che, per ogni istanza x di Π_1, x è un'istanza accettabile di Π_1 se e solo se $f(x)$ è una istanza accettabile di Π_2. In altre parole in tempo polinomiale si vuole trasformare il problema Π_1 nel problema Π_2 in modo tale che x sia un'istanza accettabile per Π_1 se e solo se la sua traduzione $f(x)$ in istanza per il problema Π_2 è accettabile per Π_2. Il concetto di riduzione polinomiale ha la conseguenza se esistesse un algoritmo per risolvere Π_2 lo si potrebbe utilizzare per risolvere Π_1. Un problema Π si dice **NP-hard** se per ogni $\Pi' \in NP$ lo si può ridurre in tempo polinomiale a Π. Dimostrare che un problema Π è NP-hard o NP-completo non è facile perché si dovrebbe dimostrare che tutti i problemi in NP si riducono polinomialmente a Π. In realtà la prima dimostrazione di NP-completezza "aggira" il problema: **SAT** siano V un insieme di variabili booleane, si riferisce ad una variabile o alla sua negazione come ad un *letterale*. Se si fa l'OR di un gruppo di letterali, si

crea una clausola. Un espressione booleana su V si dice **forma normale congiuntiva (FNC)** se è espressa come congiunzione di clausole, ovvero come un AND di clausole ognuna formata da un OR di letterali. Data una espressione in forma normale congiuntiva, il problema SAT chiede di verificare se esiste un qualche assegnamento di valore *True* e *False* tali da rendere l'intera espressione vera. Per risolvere il problema si considerano tutti i 2^n assegnamenti di valori alle n variabili, e per ciascuno si verifica se la formula è vera. Il **teorema di cook** dimostra che SAT è NP completo: dati un qualunque problema Π in NP ed una qualunque istanza x per Π, si può costruire una espressione booleana in forma normale congiuntiva che descrive il calcolo di un algoritmo per risolvere Π su x, l'espressione è vera se e solo se l'algoritmo restituisce 1. A questo punto per dimostrare che un problema decisionale Π è NP-completo, $\Pi \in NP$ e SAT (o qualunque altro problema NP-completo) si riduce al problema Π. Quando si trova un problema decisionale NP-completo o con un problema di ottimizzazione NP-hard la cui soluzione è troppo difficile da ottenere, una soluzione quasi ottima ottenibile facilmente forse è buona abbastanza perché a volte, avere una soluzione esatta non è strettamente necessario quindi una soluzione che non si discosta troppo da quella ottima o che si possa calcolare in tempo polinomiale, può andare bene.