Università degli Studi di Padova

Appunti di Algoritmi e Strutture Dati

a.a. 2018/2019

Autori:
Timoty Granziero
Marco Siragna

Repository:
https://github.com/Marco305/ASD-Notes

24 giugno 2019

Indice

1 Introduzione

1.1 Problem Solving

1. Formalizzazione del problema;
2. Sviluppo dell'**algoritmo** (focus del corso);
3. Implementazione in un programma (codice).

Algoritmo Sequenza di passi elementari che risolve il problema.

$$\text{Input} \rightarrow \textbf{Algoritmo} \rightarrow \text{Output}$$

Dato un problema, ci sono tanti algoritmi per risolverlo.

e.g.[1] Ordinamento dei numeri di una `Rubrica`. L'idea è quella di trovare tutte le permutazioni di ogni numero.

$$\text{30 numeri: } \textit{complessità } 30! \cong 2 \times 10^{32} ns \Rightarrow$$
$$3^{19} \text{anni (con } ns = \text{nanosecondi)}$$

std::vector È un esempio nel `C++` delle ragioni per cui si studia questa materia. Nella documentazione della `STL`, sono riportati i seguenti:

- `Random access`: complessità $O(1)$;
- `Insert`: complessità $O(1)$ ammortizzato.

Il `random access` è l'accesso a un elemento casuale del `vector`. $O(1)$ implica che l'accesso avviene in tempo costante (pari a 1).

Per `insert` si intende l'inserimento di un nuovo elemento in coda. Avviene in tempo $O(1)$ ammortizzato: questo perchè ogni N inserimenti, è necessario un resize del vector e una copia di tutti gli elementi nel nuovo vettore (questa procedura è nascosta al programmatore).

[1] For the sake of example.

1.2 Analisi

- Tempo di esecuzione;
- Spazio (memoria);

◦ Correttezza;

◦ Manutenibilità.

Approfondimento sul tempo di esecuzione T(n)

◦ *P Problems*: complessità polinomiale. L'algoritmo è trattabile

◦ *NP Complete*: problemi NP completi. **e.g**: Applicazione sugli algoritmi di sicurezza. Si basano sull'assunzione che per essere risolti debbano essere considerate tutte le soluzioni possibili.

◦ *NP Problems*: problemi con complessità (ad esempio) esponenziale/ fattoriale. Assolutamente non trattabili.

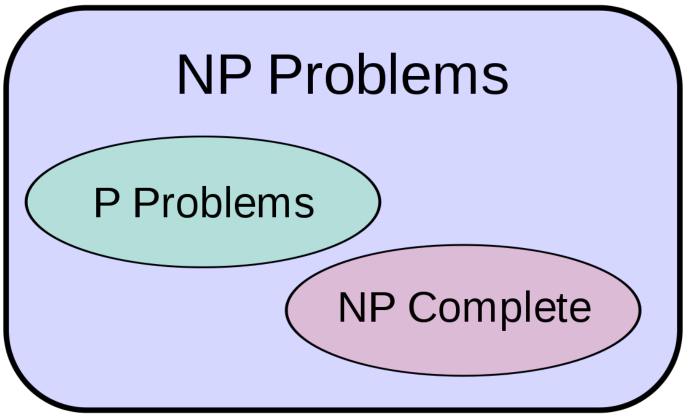


Figura 1: Complessità T(n).

2 Ordinamento

2.1 Problema dell'Ordinamento (Sorting)

`Input`: sequenza di numeri

$$a_0 a_1 \ldots a_n;$$

`Output`: permutazione

$$a'_0 a'_1 \ldots a'_n$$

tale che

$$a'_0 \leq a'_1 \leq \cdots \leq a'_n$$

Vedremo due algoritmi:

- `InsertionSort`;
- `MergeSort`.

2.2 Insertion Sort

Insertion Sort un algoritmo di **sorting incrementale**. Viene applicato naturalmente ad esempio quando si vogliono ordinare le carte nella propria mano in una partita a scala 40: si prende ogni carta a partire da sinistra, e la si posiziona in ordine crescente.

Astrazione Prendiamo ad esempio il seguente array:

5	2	8	4	7

Partiamo dal primo elemento: 5. È già ordinato con se stesso, quindi procediamo con il secondo elemento.
Confronto il numero 2 con l'elemento alla sua sinistra:
$2 \geq 5$? No, quindi lo inverto con l'elemento alla sua sinistra, come segue

2	5	8	4	7

Key: 8

La key analizzata è 8.
$8 \geq 5$? Sì, quindi è ordinato in modo corretto.

2	5	8	4	7

Key: 4

La key analizzata è 4.
  $4 \geq 8$? No, quindi lo sposto a sinistra invertendolo con 8.
  $4 \geq 5$? No, lo sposto a sinistra invertendolo con 5.
  $4 \geq 2$? Sì, quindi è nella posizione corretta.

2	4	5	8	7

Key: 7

Key analizzata 7.
  $7 \geq 8$? No, lo sposto a sinistra invertendolo con 8.
  $7 \geq 5$? Sì, è nella posizione corretta.

Ottengo l'array ordinato:

2	4	5	7	8

Algorimo Passiamo ora all'implementazione dell'algoritmo, con uno pseudocodice similare a `Python`[1]

`Input`: $A[1, .., n]$, $A.length$.

$$\text{È noto che: } A[i] \leq key < A[i+1]$$

Pseudocodice Segue lo pseudocodice dell'`InsertionSort`.

```
INSERTION-SORT(A)
n = A.length
for j = 2 to n // il primo elemento è già ordinato
    key = A[j] // A[1..j-1] ordinato
    i = j - 1
    while (i > 0) and (A[i] > key)
        A[i+1] = A[i]
        i = i - 1
    A[i+1] = key
```

Quando il `while` termina, ci sono due casi:

- `i = 0`: tutti gli elementi prima di `j` sono maggiori di `key`; `key` va al primo posto (1);
- `(i > 0) and (A[i]` $\leq$ `key)`: `A[i+1] = key`.

2.2.1 Correttezza di Insertion Sort

for `A[1..j-1]` è ordinato e contiene gli elementi in `(1,j-1)` iniziali.

while `A[1..i]A[i+2..j]` ordinato e `A[i+2..j] > key`.

In uscita abbiamo:

- `j = n+1`;
- `A[1..n]` ordinato, come da invariante: vale `A[1..j-1]` ordinato, e `j` vale `n+1`.

[1] **ATTENZIONE**: verranno usati array con indici che partono da 1.

2.2.2 Complessità di Insertion Sort

Assunzione Tutte le istruzioni richiedono un tempo <u>costante</u>.
Rivediamo l'algoritmo:

```
INSERTION-SORT(A)
1  n = A.length
2  for j = 2 to n // il primo elemento è già ordinato
3        key = A[j] // A[1..j − 1] ordinato
4        i = j − 1
5        while (i > 0) and (A[i] > key)
6              A[i + 1] = A[i]
7              i = i − 1
8        A[i + 1] = key
```

Diamo il nome c_0 alla chiamata della procedura, `InsertionSort(A)`; A ogni riga numerata, diamo il nome $c_1, c_2, .., c_8$ [1].
Vediamo il **costo** di ogni istruzione:

$\boldsymbol{c_0} \to 1$

$\boldsymbol{c_1} \to 1$

$\boldsymbol{c_2} \to n$

$\boldsymbol{c_3} \to (n-1)$

$\boldsymbol{c_4} \to (n-1)$

$$\boldsymbol{c_5} \to \sum_{j=2}^{n} t_j + 1$$

$$\boldsymbol{c_6}, \boldsymbol{c_7} \to \sum_{j=2}^{n} t_j$$

$\boldsymbol{c_8} \to (n-1)$

$$T^{IS}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n}(t_j + 1) + (c_6 + c_7) \sum_{j=2}^{n} t_j$$

[1] (c_1 corrisponde alla riga 1, c_2 alla riga 2 e così via).

$\boldsymbol{t_j}$ dipende, oltre che da n, dall'istanza dell'array che stiamo considerando. È chiaro che questo calcolo non da indicazioni precise sull'effettiva complessità dell'algoritmo.

Andiamo ad analizzare i 3 possibili casi:

a) Caso migliore (2.2.2)

b) Caso peggiore (2.2.2)

c) Caso medio (2.2.2)

Caso migliore $\rightarrow A$ ordinato $\Rightarrow t_j = 0 \ \forall \ j$

La **complessità** diventa:

$$T^{IS}_{min}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_5 + c_8)(n-1) = an + b \approx n$$

Ossia, si comporta come n. Il **caso migliore non** è interessante, visto che è improbabile si presenti.

Caso peggiore $\rightarrow A$ ordinato in senso inverso $\Rightarrow \forall \ j \ t_j = j - 1$

La **complessità** diventa:

$$T^{IS}_{max}(n) = c_0 + c_1 + c_2 n + (c_3 + c_4 + c_8)(n-1) + c_5 \sum_{j=2}^{n} j + (c_6 + c_7) \sum_{j=2}^{n} (j-1)$$

Per valutare il costo di $\sum_{j=2}^{n} j$ e di $\sum_{j=2}^{n}(j-1)$, usiamo la **somma di Gauss**:

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

Otteniamo:

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

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

Per finire, ricalcoliamo $T^{IS}_{max}(n)$

$$T^{IS}_{max}(n) = a'n^2 + b'n + c' \approx n^2$$

Caso medio Il caso medio è **difficile da calcolare**, e in una considerevole parte dei casi, coincide con il caso peggiore.

Comunque, l'idea è la seguente:

$$\frac{\displaystyle\sum_{\text{perm. di input}} T^{IS}(p)}{n!} \approx n^2 \qquad \text{posso pensare che } t_j \cong \frac{j-1}{2}$$

2.3 Divide et Impera

Un algoritmo di sorting **divide et impera** si può suddividere in 3 fasi:

divide divide il problema dato in sottoproblemi più piccoli;

impera risolve i sottoproblemi:

- ricorsivamente;
- la soluzione è nota (e.g. array con un elemento);

combina compone le soluzioni dei sottoproblemi in una soluzione del problema originale.

2.4 Merge Sort

Merge Sort[1] è un esempio di algoritmo **divide et impera**. Andiamo ad analizzarlo.

[1] Si consiglia di dare uno sguardo all'algoritmo anche da altre fonti, poichè presentarlo graficamente in LaTeX, come è stato visto a lezione, non è facile.

Astrazione Consideriamo il seguente array A.

5	2	4	7	1	2	3	6

Lo divido a metà, ottenendo due parti separate.

5	2	4	7

1	2	3	6

Consideriamo il primo, ossia `A[1..4]` (A originale). Divido anche questo a metà.

5	2

4	7

Divido nuovamente a metà, ottenendo:

5

2

5 e 2 sono due blocchi già ordinati. Scelgo il minore tra i due e lo metto in prima posizione, mentre l'altro in seconda posizione, ottenendo un blocco composto da 2 e 5.

Riprendo con il blocco composto da 4 e 7. Lo divido in due blocchi da un elemento. Faccio lo stesso procedimento fatto per 2 e 5: metto in prima posizione 4 e in seconda posizione 7. La situazione è la seguente:

2	5

4	7

So che i blocchi ottenuti contengono elementi ordinati. Con questa assunzione, posso ragionare nel seguente modo: considero il primo elemento dei due blocchi (2 e 4 in questo caso) e metto in prima posizione il minore tra i due. Ora considero il successivo elemento del blocco che è stato scelto e lo stesso elemento dell'altro blocco, e inserisco nell'array l'elemento minore. Continuo fino ad ottenere un blocco ordinato.

2	4	5	7

Faccio lo stesso procedimento con la parte di array originale `A[5..8]`, ottenendo

2	4	5	7

1	2	3	6

A questo punto, i blocchi da 4 contengono elementi tra loro ordinati. Faccio lo stesso ragionamento usato per comporli, per ottenere l'array originale ordinato. Considero[1]:

[1]Questo procedimento è stato applicato anche ai passaggi precedenti; qui è spiegato più rigorosamente.

- `L[1..4]` = `A[1..4]`: indice $i = 1$ per scorrerlo;
- `R[1..4]` = `A[5..8]`: indice $j = 1$ per scorrerlo;

Valuto `L[i]` e `R[j]`.

- Se `L[i]` $\leq$ `R[j]`, inserisco `L[i]` e incremento `i`.
- Altrimenti, inserisco `R[j]` e incremento `j`.
- Itero finchè entrambi gli indici non sono out of bounds.

Pseudocodice Segue lo pseudocodice del `MergeSort`.

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋ // arrotondato per difetto
    MERGE-SORT(A, p, q) // ordina A[p..q]
    MERGE-SORT(A, q + 1, r) // ordina A[q+1..r]
    MERGE(A, p, q, r) // "Merge" dei due sotto-array
```

```
MERGE(A, p, q, r)
n1 = q − p + 1 // gli indici partono da 1
n2 = r − q
// L sotto-array sx, R sotto-array dx
for i = 1 to n1
    L[i] = A[p + i − 1]
for j = 1 to n2
    R[j] = A[q + j]
L[n1 + 1] = R[n2 + 1] = ∞
i = j = 1
for k = p to r
    if L[i] ≤ R[j]
        A[k] = L[i]
        i = i + 1
    else // L[i] > R[j]
        A[k] = R[j]
        j = j + 1
```

Invarianti e Correttezza **L** e **R** contengono rispettivamente `A[p..q]` e `A[q+1..r]`. L'indice `k` scorre `A`. Il sotto-array `A[p..k-1]` è ordinato, e contiene `L[1..i-1]` e `R[1..j-1]`.

$$A[p\,..\,k-1] \leq L[i\,..\,n1], R[j\,..\,n2]$$
$$\Downarrow$$
$$A[p\,..\,k-1] = A[p\,..\,r+1-1] \implies A[p\,..\,r] \text{ ordinato}$$

Dimostrazione per induzione su r-p

⇒ Se $r-p == 0$ (oppure -1) abbiamo al più un elemento $\implies$ array già ordinato.

⇒ Se $r-p > 0$, vale

$$\#\text{elem}(A[p\,..\,q]), \#\text{elem}(A[q+1\,..\,r]) < \#\text{elem}(A[p\,..\,r])$$

Per ipotesi induttiva:

- `MergeSort(A,p,q)` ordina `A[p..q]`;
- `MergeSort(A,q+1,r)` ordina `A[q+1..r]`;
  Per correttezza di `Merge()`, dopo la sua chiamata ottengo `A[p..r]` ordinato.

2.4.1 Approfondimento sull'Induzione

Induzione ordinaria Proprietà $P(n)$, e.g. $P(n) =$ "Se n è pari, $n+1$ è dispari" oppure "tutti i grafi con n nodi …".

Per dimostrare che $P(n)$ vale per ogni n

- $P(0)$: **caso base**;
- assumo vera $P(n) \to$ dimostro $P(n+1)$, allora $P(n)$ è vera per ogni n.

Induzione completa

- $\big[P(0)\big]$ (non necessaria, è un'istanza del passo successivo);
- dimostro $P(m)\ \forall\ m < n \to$ vale $P(n)\ \forall\ n$.

2.4.2 Complessità di Merge Sort

$n =$ # elementi da ordinare[1]

Merge(A, p, q, r)

inizializzazione: $a'n + b'$;

ciclo: $a'n + b'$;

Sommandoli, ottengo una complessità all'incirca di:

$$T^{merge}(n) = an + b$$

Nel dettaglio:

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(n_1) + T^{MS}(n_2) + T^{merge}(n) & \text{altrimenti} \end{cases}$$

$$\Downarrow$$

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(n_1) + T^{MS}(n_2) + an + b & \text{altrimenti} \end{cases}$$

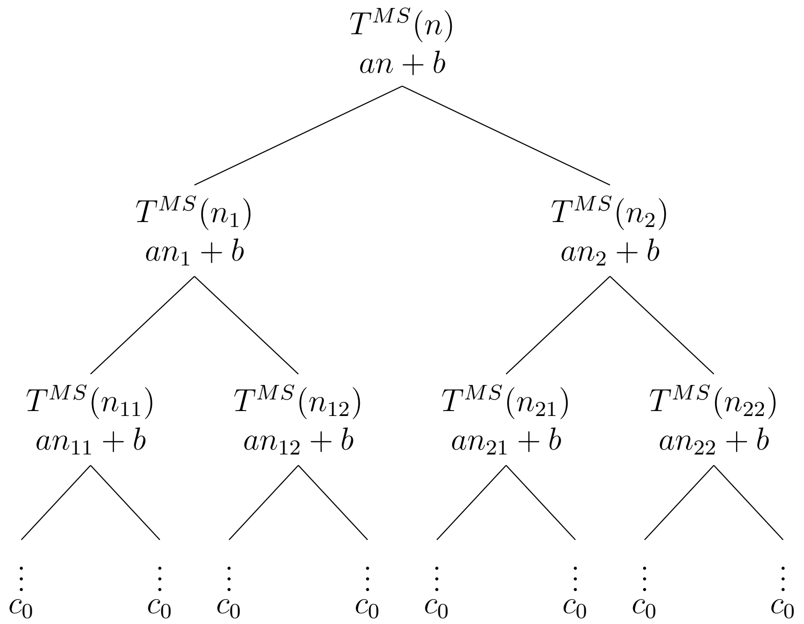
con

[1] Il simbolo # verrà usato per indicare la cardinalità di un insieme.

$n_1 = \lfloor \frac{n}{2} \rfloor$

$n_2 = \lceil \frac{n}{2} \rceil$

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(\lfloor \frac{n}{2} \rfloor) + T^{MS}(\lceil \frac{n}{2} \rceil) + an + b & \text{altrimenti} \end{cases}$$



Otteniamo c_0 ripetuto n volte all'ultimo livello dell'albero. L'altezza dell'albero è circa $\log_2 n$. Vediamo nel dettaglio la complessità nelle varie iterazioni.

$\boldsymbol{i = 0}$ $\quad an + b$

$\boldsymbol{i = 1}$ $\quad a(n_1 + n_2) + 2b \approx an + 2b$

$\boldsymbol{i = 2}$ $\quad a(n_{11} + n_{12} + n_{21} + n_{22}) + 4b \approx an + 4b$

$\dots$

$\boldsymbol{i = h}$ $\quad c_0 n$

Poniamo $n = 2^h$. Abbiamo

$$
\begin{aligned}
T^{MS}(n) &= \sum_{i=0}^{h-1}(an + 2^i b) + c_0 n \\
&= anh + b\sum_{i=0}^{h-1} 2^i && (h = \log_2 n) \\
&= an\log_2 n + b2^h - b + c_0 n && (2^h = n) \\
&= an\log_2 n + (b + c_0)n - b
\end{aligned}
$$

$$T^{MS}(n) = an\log_2 n + b''n + c'' \approx n\log_2 n$$

2.5 Confronto tra Insertion Sort e Merge Sort

$$T^{IS}(n) = a'n^2 + b'n + c'$$
$$T^{MS}(n) = a''n\log_2 n + b''n + c''$$

Posso calcolare il limite del rapporto:

$$\lim_{n\to+\infty}\frac{T^{MS}(n)}{T^{IS}(n)} = \lim_{n\to+\infty}\frac{a''n\log_2 n + b''n + c''}{a'n^2 + b'n + c'} = 0$$

Per definizione

$$\forall\ \varepsilon > 0\ \exists\ n_0 : \forall\ n \geq n_0 \quad \frac{T^{MS}(n)}{T^{IS}(n)} < \varepsilon$$
$$\Downarrow$$
$$T^{MS}(n) < \varepsilon T^{IS}(n) = \frac{T^{IS}}{m} \qquad \text{(Ponendo, ad esempio, } \varepsilon = \frac{1}{m}\text{)}$$

Detto a parole, c'è un certo n oltre il quale, ad esempio, `MergeSort` su un **Commodore 64** esegue più velocemente di un `InsertionSort` su una macchina moderna. Possiamo vedere una comparazione tra i due algoritmi nella seguente tabella.

n	$T^{IS}(n) = n^2$	$T^{MS}(n) = n\log n$
10	$0.1ns$	$0.033ns$
1000	$1ms$	$10\mu s$
10^6	17 minuti	$20ms$
10^9	70 anni	$30s$

3 Complessità Asintotica

3.1 Notazione Asintotica

Il **tempo di esecuzione** è difficile da calcolare, come visto nella sezione 2.2.2. Il modo in cui è stato calcolato è pieno di dettagli "inutili".

Rivediamo le complessità di `InsertionSort` e `MergeSort`:

$$T^{IS} = an^2 + bn + c$$
$$T^{MS} = an \log_2 n + bn + c$$

A noi interessa calcolare $T(n)$ per n "grande". Non consideriamo le costanti moltiplicative, che sono non fondamentali. Ecco una lista di possibili complessità ordinate in senso decrescente (le prime due categorie appartengono alla classe degli **NP problems**, ossia non trattabili):

- 3^n
- 2^n

- n^k
- n^2
- $n \log n$
- n
- $\log n$
- 1

Prendiamo in esame due funzioni: $f(n)$, $g(n)$:

$$f, g : \mathbb{R}^+ \to \mathbb{R}^+$$

- $f(n)$ è la funzione in esame della complessità del nostro problema P;
- $g(n)$ è la funzione che, moltiplicata per un'opportuna costante c_i, dopo un certo n, fa da limite superiore o inferiore per ogni punto di $f(n)$.

3.1.1 Limite Asintotico Superiore

Data $g(n)$, indichiamo con $O\big(g(n)\big)$ il **limite asintotico superiore**, definito come segue:

$$O\big(g(n)\big) = \{f(n) : \exists\ c > 0 \quad \exists\ n_0 \in \mathbb{N} : \forall\ n \geq n_0 \quad (0 \leq) f(n) \leq c \cdot g(n)\}$$

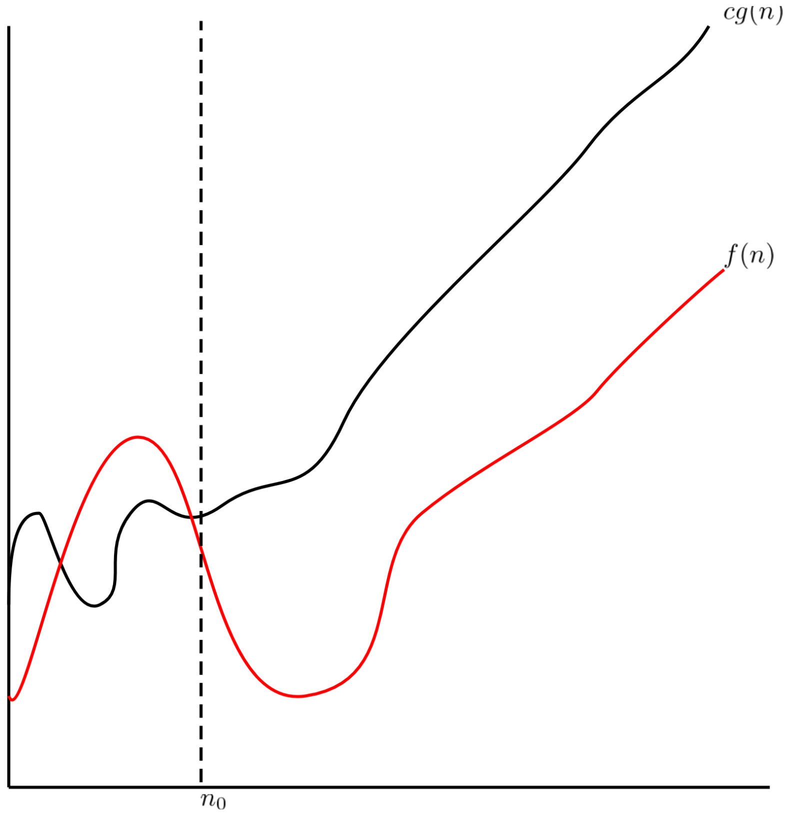


Figura 2: Rappresentazione del limite asintotico superiore per $f(n)$

Esempi

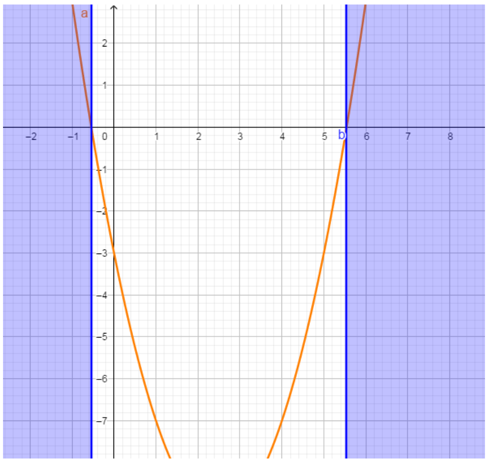
- $f_1(n) = 2n^2 + 5n + 3 = O\big(g(n^2)\big)$? Sì.
  Deve valere $f_1(n) < cn^2 \qquad \exists\ c > 0,\ n \geq n_0$

Ipotizziamo $c = 3$

$$2n^2 + 5n + 3 \leq 3n^2$$
$$n^2 - 5n - 3 \geq 0$$

$$\frac{5 \pm \sqrt{2 \cdot 5 + 12}}{2} = \frac{5 \pm \sqrt{37}}{2} \cong 5.54$$

(Non considero la soluzione negativa, poiché siamo in $\mathbb{R}^+$)



Prendo $c = 3$ e $n_0 = 6$. Vale dunque:

$$f_1(n) \leq cn^2 \quad \forall\, n \geq n_0$$

- $f_1(n) = O\big(g(n^3)\big)$? Sì.
  $c = 3$
  $n_0 = 6 \quad \forall\, n \geq n_0$
  $f_1(n) \leq cn^2 \leq cn^3$
- $f_2(n) = 2 + \sin(n) = O(1)$? Sì.
  $-1 \leq \sin(n) \leq 1$
  $1 \leq f_2(n) \leq 3$
  Vale la seguente

$$\exists\, c > 0 \quad \exists\, n_0 : \forall\, n \geq n_0 \quad f_2(n) \leq c \cdot 1$$
$$\text{ok per } c = 3,\; n_0 = 0$$

3.1.2 Limite Asintotico Inferiore

Data $g(n)$, indichiamo con $\Omega\big(g(n)\big)$ il **limite asintotico inferiore**, definito come segue:

$$\Omega\big(g(n)\big) = \{f(n) : \exists\ c > 0 \quad \exists\ n_0 \in \mathbb{N} : \forall\ n \geq n_0 \quad c \cdot g(n) \leq f(n)\}$$

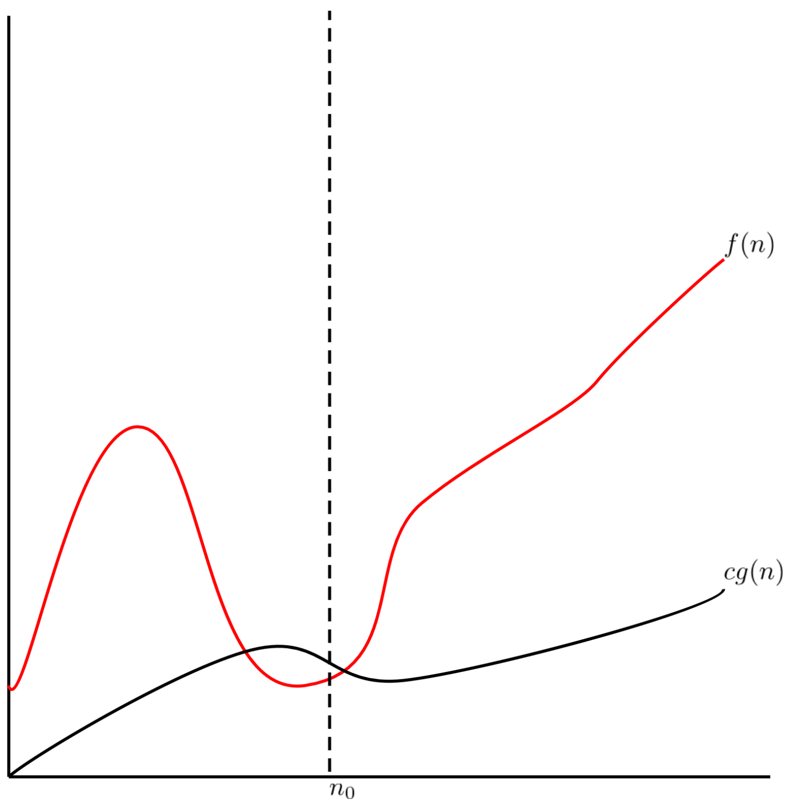


Figura 3: Rappresentazione del limite asintotico inferiore per $f(n)$

Esempi

- $f_1(n) = 2n^2 + 5n + 3 = \Omega\big(g(n^2)\big)$? Sì.

  Deve valere:

  $$\exists\ c > 0 \quad \exists\ n_0 : \forall\ n \geq n_0 \quad cn^2 \leq 2n + 5n + 3$$

  Basta porre $c = 1$, $n_0 = 0$.

- $f_2(n) = 2 + \sin(n) = \Omega(1)$? Sì.

  $$1 \leq f_2(n) \leq 3 \quad c = 1,\ n_0 = 0$$

3.1.3 Limite Asintotico Stretto

Data $g(n)$, indichiamo con $\Theta\big(g(n)\big)$ il **limite asintotico stretto**, definito come segue:

$$\Theta\big(g(n)\big) = \{f(n) : \exists\ c_1, c_2 > 0 \quad \exists\ n_0 \in \mathbb{N} : \forall\ n \geq n_0 \\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

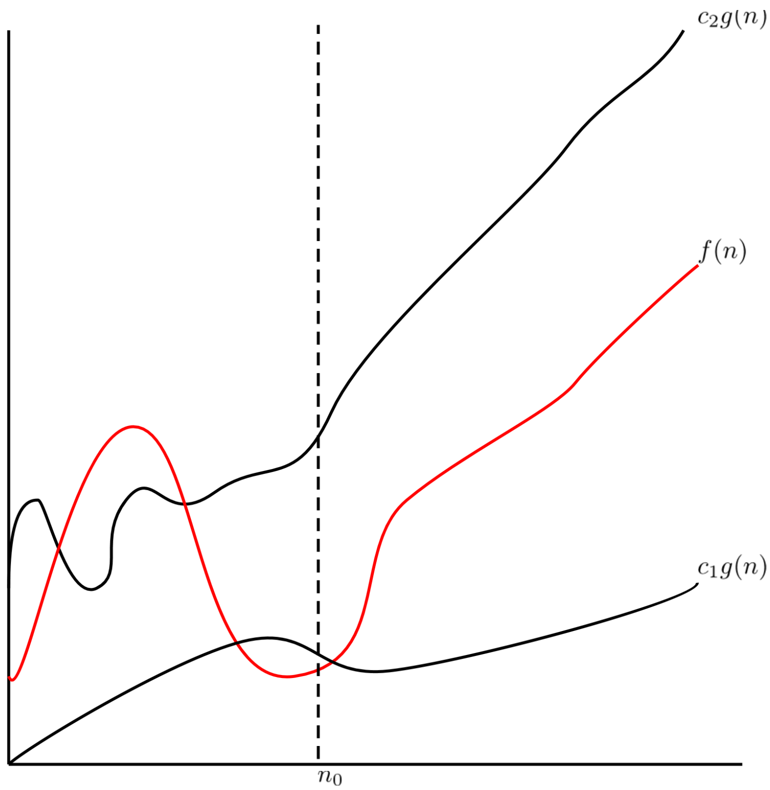


Figura 4: Rappresentazione del limite asintotico stretto per $f(n)$

Esempi

$$\begin{aligned} f_1(n) =& 2n^2 + 5n + 3 = \Theta(n^2) \\ & c_1 = 1 \quad c_2 = 3 \quad n_0 = 6 \\ f_2(n) =& 2 + \sin(n) = \Theta(1) \\ & c_1 = 1 \quad c_2 = 3 \quad n_0 = 0 \end{aligned}$$

$$\begin{gathered} f_1(n) \neq \Theta(n^3) \\ f_1(n) = O(n^3) \\ f_1(n) \neq \Omega(n^3) \\ \Downarrow \\ \frac{f_1(n)}{n_3} \to 0 \end{gathered}$$

3.2 Metodo del Limite

Siano $f(n),\ g(n) > 0 \quad \forall\ n$

Se $\exists\ \lim_{n\to+\infty} \frac{f(n)}{g(n)}$, allora:

1. Se $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = k > 0$ allora $f(n) = \Theta\big(g(n)\big)$.
2. Se $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$ allora $f(n) = O\big(g(n)\big)$ e $f(n) \neq \Omega\big(g(n)\big)$.
3. Se $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = \infty$ allora $f(n) = \Omega\big(g(n)\big)$ e $f(n) \neq O\big(g(n)\big)$.

Dimostrazione

1. Sia $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = k > 0$

$$\begin{aligned}
&\Rightarrow \forall\ \varepsilon > 0 \quad \exists\ n_0 : \forall\ n \geq n_0 \quad \left|\frac{f(n)}{g(n)} - k\right| \leq \varepsilon \\
&\Rightarrow -\varepsilon \leq \frac{f(n)}{g(n)} - k \leq \varepsilon \\
&\Rightarrow k - \varepsilon \leq \frac{f(n)}{g(n)} \leq k + \varepsilon \\
&\Rightarrow (k-\varepsilon)g(n) \leq f(n) \leq (k+\varepsilon)g(n) \qquad \text{per } 0 < \varepsilon < k \\
&\Rightarrow f(n) = \Theta\big(g(n)\big)
\end{aligned}$$

2. Sia $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$

$$\begin{aligned}
&\Rightarrow \forall\ \varepsilon > 0 \quad \exists\ n_0 : \forall\ n \geq n_0 \quad \frac{f(n)}{g(n)} \leq \varepsilon \\
&\Rightarrow f(n) \leq \varepsilon g(n) \\
&\Rightarrow f(n) = O\big(g(n)\big)
\end{aligned}$$

Sia $f(n) = \Omega\big(g(n)\big)$

$$\begin{aligned}
&\Rightarrow \exists\ c > 0 \quad \exists\ n_0 \in \mathbb{N} : \forall\ n \geq n_0 \quad cg(n) \leq f(n) \\
&\Rightarrow \text{Impossibile, infatti sia } \varepsilon = \frac{c}{2} > 0 \\
&\Rightarrow \exists\ n_1 \in \mathbb{N} : \forall\ n \geq n_1 \quad f(n) \leq \frac{c}{2} g(n) < cg(n) \\
&\Rightarrow \text{Contraddizione!}
\end{aligned}$$

3. Si dimostra in modo analogo al punto (2)

3.3 Proprietà Generali

- $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0 = \Theta(n^k)$
- $h \neq k \quad \Theta(n^h) \neq \Theta(n^k)$
- $a \neq b \quad \Theta(a^k) \neq \Theta(b^n)$
- $h \neq k \quad \Theta(a^{n+h}) = \Theta(a^{n+k})$
- $a \neq b \quad \Theta(\log_a n) = \Theta(\log_b n)$

In generale

$$O(1) \subseteq O(\log n) \subseteq O(n) \subseteq O(n \log n) \subseteq O(n^2) \subseteq \ldots$$

3.4 Complessità di un Problema

Dato un problema P {INPUT → OUTPUT}, la **complessità** di P è la complessità dell'algoritmo più efficiente che risolve P.

Limite superiore per complessità di P Se A è un algoritmo per P con complessità $O\big(f(n)\big)$, allora P è $O\big(f(n)\big)$.

Limite inferiore per complessità di P Se ogni algoritmo che risolve P ha complessità $\Omega\big(f(n)\big)$, allora P ha complessità $\Omega\big(f(n)\big)$

$\Longrightarrow$ se P è $O\big(f(n)\big)$ e $\Omega\big(f(n)\big) \Rightarrow$ P è $\Theta\big(f(n)\big)$

3.4.1 Esempio: limite inferiore per ordinamento basato su scambi di elementi contigui

Def (inversione) Dato `A[1..n]`, una **inversione** è una coppia (i, j) con $i, j \in [1, n]$ con $i < j$ e `A[i]` $>$ `A[j]`.

Operazione disponibile: `A[k]` $\leftrightarrow$ `A[k+1]` (scambio tra gli elementi in posizione k e $k + 1$).

$$\begin{aligned} \#inv(A) &= \text{numero di inversioni di } A \\ &= \Big|\ \{(i, j) : 1 \leq i \leq j \leq n,\ A[i] > A[j]\}\ \Big| \end{aligned}$$

1. A è ordinato sse $\#inv(A) = 0$;
2. A è ordinato in senso inverso sse
$$\sum_{j=2}^{n} j - 1 = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$
Ossia, $\#inv(A)$ è massimizzato.

Vediamo cosa succede alle coppie (i, j) e a $\#inv(A)$ nel caso avvenga uno scambio `A[k]` $\leftrightarrow$ `A[k+1]`.

- $i, j \neq k$ e $i, j \neq k + 1 \implies (i, j)$ è inversione prima sse è inversione dopo;
- $i = k,\ j = k + 1$
$$\Longrightarrow \begin{cases} A[i] < A[j] & +1 \text{ inversione} \\ A[i] = A[j] & \#inv(A) \text{ non cambia} \\ A[i] > A[j] & -1 \text{ inversione} \end{cases}$$

- $i = k$ oppure $i = k+1$, $j > k+1 \implies (k, j)$ è inversione prima sse $(k+1, j)$ è inversione dopo;
- $j = k$ oppure $j = k+1$, $i < k$, analogo al caso precedente.

Per concludere, possiamo dire che l'operazione `A[k]` $\leftrightarrow$ `A[k+1]` riduce $\#inv(A)$ al massimo di 1.

$\implies$ qualunque algoritmo di ordinamento basato su scambi è $\Omega\left(\frac{n(n-1)}{2}\right) = \Omega(n^2)$

3.5 Soluzione di Ricorrenze

Abbiamo visto per `MergeSort` la complessità nel modo seguente:

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p+r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r) // complessità an + b
```

$$T^{MS}(n) = \begin{cases} c_0 & \text{se } n \leq 1 \\ T^{MS}(\lfloor \frac{n}{2} \rfloor) + T^{MS}(\lceil \frac{n}{2} \rceil) + an + b & \text{se } n > 1 \end{cases}$$

È stato tuttavia un approccio non molto preciso. Ci sono due metodi per risolvere precisamente i problemi di ricorrenza:

- **Metodo di sostituzione** (3.5.1);
- **Master Theorem** (3.5.2).

3.5.1 Metodo di Sostituzione

Dato una ricorrenza, si può provare a "indovinare" la soluzione e dimostrare che è corretta, oppure si può sviluppare l'**albero delle ricorrenze**:

- **radice**: chiamata di cui vogliamo la complessità;
- per ogni nodo:
  - → costo della parte non ricorsiva;
  - → un figlio per ogni chiamata.

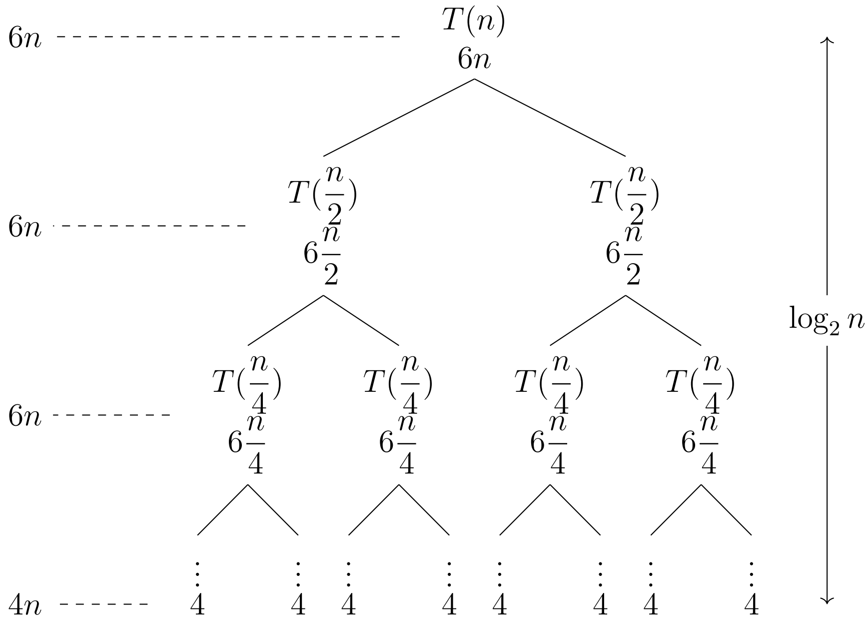
Esempio

$$T(n) = \begin{cases} 4 & \text{se } n = 1 \\ 2T(\frac{n}{2}) + 6n & \text{se } n > 1 \end{cases}$$

In generale, si può benissimo trascurare il caso base per poter ottenere espressioni meno verbose, in questo caso otterremmo:

$$T(n) = 2T(\frac{n}{2}) + 6n$$

Costruendo l'albero delle ricorrenze si intuisce già la soluzione:



Per essere sicuri della soluzione, facciamo il procedimento per intero. Proviamo a "indovinare" la soluzione. Assomiglia a `MergeSort`, quindi ipotizziamo abbia una complessità con un andamento simile

$$T(n) = an\log n + bn + c$$

Facciamo la prova induttiva.

$$\begin{aligned} (n = 1) \quad T(1) &= 4 \\ &= a \cdot 1 \cdot \log 1 + b \cdot 1 + c && (\log 1 = 0) \\ &= b + c && \text{ok se } b + c = 4 \\ (n > 1) \quad T(n) &= 2T\Big(\frac{n}{2}\Big) + 6n \end{aligned}$$

Per ipotesi induttiva

$$T\left(\frac{n}{2}\right) = a\frac{n}{2} \cdot \log\frac{n}{2} + b\frac{n}{2} + c$$

Calcolo ora $T(n)$

$$\begin{aligned} T(n) &= an\log_2\frac{n}{2} + bn + 2c + 6n = \\ &= an\log_2 n - an\log_2 2 + bn + 6n + 2c = \quad (\log_2 2 = 1) \\ &= an\log_2 n + n(b + 6 - a) + 2c = \\ &= an\log_2 n + bn + c \\ &\quad\Downarrow \end{aligned}$$

$$\begin{aligned} b + 6 - a = b &\Rightarrow a = 6 \\ 2c = c &\Rightarrow c = 0 \end{aligned}$$

$$\begin{aligned} b + c = 4 &\Rightarrow b = 4 \\ T(n) &= an\log n + bn + c \\ &= 6n\log n + 4n \end{aligned}$$

Esercizio (importante)

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + 6n \\ &= 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n) \\ \text{vale } \exists\ c > 0\ \exists\ n_0 &: \forall\ n \geq n_0 \quad \Theta(n) \leq cn \end{aligned}$$

Voglio dimostrare che

1. $T(n) = O(n \log n)$
2. $T(n) = \Omega(n \log n)$

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

$$\text{significa che } \exists\ d > 0\ \exists\ n_1 \in \mathbb{N} : T(n) \leq dn \log n \quad \forall\ n \geq n_1$$

Dimostro per induzione $T(n) \leq dn \log n \quad \forall\ n \geq n_1$.

Ometto il caso base, poiché non è molto interessante (mi basterebbe aumentare ulteriormente d per avere un valore accettabile).

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn && \text{ip. induttiva } T\left(\frac{n}{2}\right) = d\frac{n}{2}\log\frac{n}{2} \\ &\leq 2 \cdot \frac{n}{2} d \log \frac{n}{2} + cn && \left(\log\frac{n}{2} = \log n - \log 2\right) \\ &= dn \log n - dn \log 2 + cn \\ &= dn \log n - n(d \log 2 - c) \leq dn \log n \\ &\Rightarrow -n(d \log 2 - c) \leq 0 \\ &n(d \log 2 - c) \geq 0 \\ &d \log 2 - c \geq 0 \\ &\quad d \geq \frac{c}{\log 2} \end{aligned}$$

2. $T(n) = \Omega(n \log n)$ è analoga.

$$\exists\ \delta > 0 : \forall\ n > n_0 \Rightarrow T(n) \geq \delta n \log n$$

Ho l'ipotesi induttiva $T(\frac{n}{2}) \geq \delta\frac{n}{2}\log\frac{n}{2}$

$$\begin{aligned}
T(n) &\geq 2\delta\frac{n}{2}\log\frac{n}{2} + cn = \\
&= \delta n\log n - \delta n\log 2 + cn = \\
&= \delta n\log n + n(c - \delta\log 2) \geq \delta n\log n \\
&\quad \text{Deve valere } c - \delta\log 2 \geq 0 \\
&\quad \Rightarrow 0 < \delta \leq \frac{c}{\log 2}
\end{aligned}$$

Esercizio $T(n) = T\big(\frac{n}{3}\big) + T\big(\frac{2n}{3}\big) + \Theta(n) \quad (\Theta(n) \leq c \cdot n)$

Ipotizzo un andamento simile a `MergeSort`: $\Theta(n\log n)$. Dimostro:

1. $T(n) = O(n\log n)$
2. $T(n) = \Omega(n\log n)$

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

$$\exists\ d > 0 : \forall\ n > n_0 \Rightarrow T(n) \leq dn\log n$$

Ometto il caso base. L'ipotesi induttiva è la seguente:

$$T(n) \leq d\frac{n}{3}\log\frac{n}{3} + d\frac{2n}{3}\log\frac{2n}{3} + cn$$

Procedo con i calcoli ...

$$\begin{aligned}
T(n) &\leq T\Big(\frac{n}{3}\Big) + T\Big(\frac{2n}{3}\Big) + cn \\
&\leq d\frac{n}{3}\log\frac{n}{3} + d\frac{2n}{3}\log\frac{2n}{3} + cn = \\
&= d\frac{n}{3}\Big(\log n - \log 3\Big) + d\frac{2n}{3}\Big(\log n - \log\frac{2}{3}\Big) + cn = \\
&= dn\log n - \frac{dn}{3}\Big(\log 3 - 2\log\frac{2}{3}\Big) + cn = \\
&= dn\log n - \frac{dn}{3}\Big(\log 3 - \log\frac{4}{9}\Big) + cn = \\
&= dn\log n - n\Big(\frac{d}{3}\log\frac{27}{4} - c\Big) \leq dn\log n \\
&\ \frac{d}{3}\log\frac{27}{4} - c \geq 0 \\
&\Rightarrow d \geq \frac{3c}{\log\frac{27}{4}} \qquad (\log\frac{27}{4} > 1 \text{ poiché } arg > 1)
\end{aligned}$$

2. $T(n) = \Omega(n \log n)$ è analoga

$$\exists\ \delta > 0 : \forall\ n > n_0 \Rightarrow T(n) \geq \delta n \log n$$

L'ipotesi induttiva è la seguente:

$$T(n) \geq \delta \frac{n}{3} \log \frac{n}{3} + \delta \frac{2n}{3} \log \frac{2n}{3} + cn$$

Calcoli ...

$$\begin{aligned}
T(n) &\geq T\Big(\frac{n}{3}\Big) + T\Big(\frac{2n}{3}\Big) + cn \\
&\geq \delta \frac{n}{3} \log \frac{n}{3} + \delta \frac{2n}{3} \log \frac{2n}{3} + cn = \\
&= \delta \frac{n}{3}\Big(\log n - \log 3\Big) + \delta \frac{2n}{3}\Big(\log n - \log \frac{2}{3}\Big) + cn = \\
&= \delta n \log n + \frac{\delta n}{3}\Big(-\log 3 + 2 \log \frac{2}{3}\Big) + cn = \\
&= \delta n \log n + \frac{\delta n}{3}\Big(-\log 3 + \log \frac{4}{9}\Big) + cn = \\
&= \delta n \log n + n\Big(-\frac{\delta}{3} \log \frac{27}{4} + c\Big) \geq \delta n \log n \\
&\quad -\frac{\delta}{3} \log \frac{27}{4} + c \geq 0 \\
&\quad \Rightarrow 0 < \delta \leq \frac{3c}{\log \frac{27}{4}}
\end{aligned}$$

3.5.2 Master Theorem

Dato un problema con `size` n, vogliamo dividerlo in a sottoproblemi con `size` $\frac{n}{b}$. Otteniamo la seguente ricorrenza (ricordiamo che il caso base è omesso per semplicità):

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

con $a \geq 1,\ b > 1$, allora possiamo confrontare

- $f(n)$;
- $n^{\log_b a}$.

Tre possibili casi:

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

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

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

$$T(n) = \Theta\big(n^{\log_b a} \cdot \log n\big)$$

3. Se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche $\varepsilon > 0$, e vale la **regolarità**

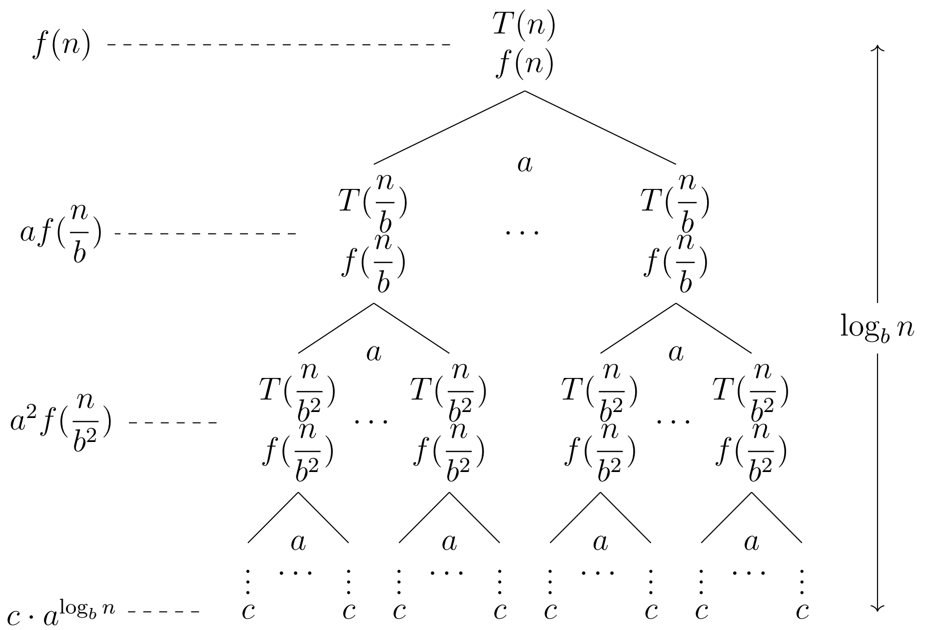
$$\exists\ 0 < k < 1\ :\ a \cdot f(\frac{n}{b}) \leq k \cdot f(n)$$

allora

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

Intuizione sul perchè $n^{\log_b a}$

$$T(n) = f(n) + af\left(\frac{n}{b}\right) + a^2 f\left(\frac{n}{b^2}\right) + \cdots + a^{\log_b n} f\left(\frac{n}{b^{\log_b n}}\right) + c \cdot a^{\log_b n}$$
$$a^{\log_b n} = \left(b^{\log_b a}\right)^{\log_b n} = \left(b^{\log_b n}\right)^{\log_b a} = n^{\log_b a}$$
$$\text{Nota bene: } af\left(\frac{n}{b}\right) \leq k \cdot f(n) \text{ con } k < 1$$



Vediamo ora i casi in cui sarà possibile finire, e le conclusioni legate ad essi.

A)

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = l(>0) \neq \infty$$

$$\textbf{Caso 2} \Rightarrow T(n) = \Theta\big(n^{\log_b a} \cdot \log n\big)$$

B)

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = 0$$

$$\text{Potrei essere nel } \textbf{Caso 1} \Rightarrow \text{se } \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a-\varepsilon}} = l(\geq 0) \neq \infty \ (\varepsilon > 0)$$
$$\Rightarrow T(n) = \Theta\big(n^{\log_b a}\big)$$

C)

$$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = \infty \quad \& \ \exists\, \varepsilon > 0 : \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a+\varepsilon}} = \infty$$
$$\& \ \textbf{Regolarità} \Rightarrow \textbf{Caso 3:} \quad T(n) = \Theta(f(n))$$

Esercizi

- $T^{MS} = 2T\big(\frac{n}{2}\big) + a'n + b'$

  Abbiamo (rispetto alla forma $T(n) = a \cdot T\big(\frac{n}{b}\big) + f(n)$)

  $$a = 2,\ b = 2$$
  $$f(n) = a'n + b' \qquad n^{\log_2 2} = n$$

  È chiaro che le due funzioni hanno lo stesso andamento (di ordine $\Theta(n)$):

  $$a'n + b' = \Theta(n)$$
  $$\textbf{Caso 2} \Rightarrow T(n) = \Theta\Big(n^{\log_2 2} \log n\Big) = \Theta(n \log n)$$

- $T(n) = 5T\big(\frac{n}{2}\big) + 2n^2 + n \log n$

  Abbiamo (rispetto alla forma $T(n) = a \cdot T\big(\frac{n}{b}\big) + f(n)$)

  $$a = 5,\ b = 2$$
  $$f(n) = n^2 + n \log n \qquad n^{\log_2 5} \quad (\log_2 5 > 2)$$
  $$0 < \varepsilon < \log_2 5 - 2 \Rightarrow \lim_{n \to \infty} \frac{2n^2 + n \log n}{n^{\log_2 5 - \varepsilon}} = 0 \Rightarrow f(n) = O(n^{\log_2 5})$$
  $$\textbf{Caso 1} \Rightarrow T(n) = \Theta\big(n^{\log_2 5}\big)$$

- $T(n) = 5T(\frac{n}{2}) + n^3$ per esercizio.

- $T(n) = 5T(\frac{n}{2}) + n^3 \log n$

  Abbiamo

  $$a = 5,\ b = 2$$
  $$f(n) = n^3 \log n \qquad n^{\log_2 5} \quad (\log_2 5 < 3)$$
  $$0 < \varepsilon < 3 - \log_2 5 \Rightarrow \lim_{n \to \infty} \frac{n^3 \log n}{n^{\log_2 5 + \varepsilon}} = \infty$$

  Possibile **caso 3**. **Regolarità**?

  $$af\Big(\frac{n}{b}\Big) \leq kf(n) \quad \text{per } 0 < k < 1 \text{ opportuno}$$
  $$5\Big(\frac{n}{2}\Big)^3 \log \frac{n}{2} = \frac{5}{8} n^3 \log \frac{n}{2} \leq \frac{5}{8} n^3 \log n \leq kn^3 \log n \quad \text{per } 0 < k \leq \frac{5}{8} < 1$$
  $$\Downarrow$$
  $$\textbf{Caso 3}: T(n) = \Theta\big(f(n)\big) = \Theta(n^3 \log n)$$

- $T(n) = 27T(\frac{n}{3}) + n^3 \log n$

$$f(n) = n^3 \log n \qquad n^{\log_3 27} \quad (\log_3 27 = 3)$$

$$\lim_{n\to\infty} \frac{n^3 \log n}{n^{3+\varepsilon}} = +\infty \quad \forall\, \varepsilon > 0, \text{ non possiamo dimostrare 3}$$

$$\Rightarrow \text{Non siamo in } \textbf{nessun} \text{ caso del Master Theorem.}$$

Anche valutando la **regolarità**, ricadiamo in un assurdo. Dobbiamo dimostrare che $af\left(\frac{n}{b}\right) < kf(n)$ per qualche $k > 0$

$$27\left(\frac{n}{3}\right)^3 \log \frac{n}{3} = n^3(\log n - \log 3) \not< kn^3 \log n \;\text{ per nessun } k > 0$$

$$\text{Infatti } \frac{(\log n - \log 3)n^3}{n^3 \log n} \to 1$$

(Posso usare il Metodo di Sostituzione)

$$T(n) = 27T\left(\frac{n}{3}\right) + n^3 \log n$$

Costruiamo l'albero delle ricorrenze:

- **radice**: costo $n^3 \log n$;
- ogni nodo ha 27 figli.
  - i 27 figli del primo livello hanno costo $(\frac{n}{3})^3 \log \frac{n}{3}$;
  - i 27^2 figli del secondo livello hanno costo $(\frac{n}{9})^3 \log \frac{n}{9}$;
  - ...
  - le 27^n foglie terminali hanno costo $O(1)$.

$$T(n) = \sum_{j=0}^{\log_3 n} n^3 \log \frac{n}{3^j} = n^3 \sum_{j=0}^{\log_3 n} (\log n - j \log 3) + cn =$$

$$= n^3(\log n)^2 - n^3 \log 3 \sum_{j=0}^{\log_3 n} j + cn \qquad \left(\sum_{j=0}^{\log_3 n} j \cong (\log_3 n)^2\right)$$

$$T(n) = 27T\left(\frac{n}{3}\right) + n^3 \log n$$

$$T(n) = \Theta\big(n^3(\log n)^2\big) \qquad \text{ipotesi ricavata}$$

Devo dimostrare che valgano le seguenti condizioni:

1. $T(n) = O\big(n^3(\log n)^2\big)$
2. $T(n) = \Omega\big(n^3(\log n)^2\big)$

1. $T(n) = O\big(n^3(\log n)^2\big)$

$$T(n) \leq c \cdot n^3\big(n^3(\log n)^2\big) \qquad c > 0$$
$$T(n) = 27T\left(\frac{n}{3}\right) + n^3 \log n$$
$$\left(\text{ipotesi induttiva } T\left(\frac{n}{3}\right) \leq c \cdot \left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2\right)$$
$$\leq 27c\left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 + n^3 \log n =$$
$$= \frac{\cancel{27}cn^3}{\cancel{27}}(\log n - \log 3)^2 + n^3 \log n =$$
$$= cn^3\Big((\log n)^2 - 2\log 3 \log n + (\log 3)^2\Big) + n^3 \log n =$$
$$= cn^3(\log n)^2 - n^3\Big(\log n(2c\log 3 - 1) - c(\log 3)^2\Big)$$
$$\leq cn^3(\log n)^2$$

Per un n abbastanza grande, vale la disuguaglianza con un opportuno valore di c:

$$c > \frac{1}{2\log 3}$$

2. $T(n) = \Omega\big(n^3(\log n)^2\big)$

$$\exists\ d > 0 : T(n) \geq dn^3(\log n)^2$$
$$\geq 27\left(\frac{n}{3}\right)^3 \left(\log \frac{n}{3}\right)^2 + n^3 \log n$$
$$= \cdots = dn^3(\log n)^2 - n^3\Big(\log n(2d\log 3 - 1) - d(\log 3)^2\Big)$$
$$\geq dn^3(\log n)^2$$

Per un n abbastanza grande, vale la disuguaglianza con un opportuno valore di d:

$$2d\log 3 - 1 < 0 \qquad \text{ok per } 0 < d < \frac{1}{2\log 3}$$

4 Ordinamento (cont.)

Ordinamento Finora abbiamo visto due algoritmi di ordinamento, in cui avevamo le seguenti premesse:

`IN`: $a_1 \dots a_n$;

`OUT`: permutazione $a'_1 \dots a'_n$ ordinata.

In particolare, abbiamo concluso che:

- `InsertionSort`: $O(n^2)$, basato su scambi;
- `MergeSort`: $\Theta(n \log n)$, ma con un costo in termini di **memoria**.

Memoria

- `InsertionSort`:

$$\textit{input} \ + \ 1 \text{ variabile} \Rightarrow \text{spazio } \textbf{costante } \Theta(1) \text{ (detto ``in loco'')}$$

- `MergeSort`: spazio con costo lineare.

$$\begin{aligned} S_{MS}(n) &= max\left\{S\left(\left\lfloor\frac{n}{2}\right\rfloor\right),\ S\left(\left\lceil\frac{n}{2}\right\rceil\right),\ \Theta(n)\right\} \\ &= \Theta(n) \end{aligned}$$

4.1 Heapsort

L'Heapsort[1] è un algoritmo di ordinamento basato su una struttura chiamata heap, che prende le caratteristiche positive di `InsertionSort` e `MergeSort`:

- in "loco" (spazio $\Theta(1)$);
- complessità $\Theta(n \log n)$.

Cos'è un heap? Un `heap` è una struttura dati basata sugli alberi che soddisfa la "proprietà di heap": se A è un genitore di B, allora la chiave di A è ordinata rispetto alla chiave di B conformemente alla relazione d'ordine applicata all'intero heap.

Seguono alcune definizioni.

[1] Anche qui, si consiglia di dare un occhio ad altre fonti. In classe, sono stati viste molte rappresentazioni grafiche degli heap, e, come già detto, in LaTeX non è per me facile rappresentarli.

Altezza: è la distanza dalla radice alla foglia più distante;

Albero completo: è un albero di altezza h con $\sum_{i=0}^{h} 2^i - 1$ nodi;

Albero quasi completo: è un albero completo a tutti i livelli eccetto l'ultimo, in cui possono mancare delle foglie e le foglie presenti sono addossate a sinistra.

Gli heap verranno rappresentati in array monodimensionali, nel modo descritto di seguito:

$$\forall\ i > 0$$

- `A[i]` è il nodo genitore;
- `A[2i]` è il figlio sx del nodo `A[i]`;
- `A[2i+1]` è il figlio dx.

Inoltre, ogni array `A` sarà dinamico, e avrà:

- $A.length$ potenziale spazio, capacità massima dell'array;
- $A.heapsize$ celle effettive dell'array.

Vediamo alcune funzioni di utilità che verranno usate.

```
LEFT(i)
    // restituisce il figlio sx del nodo i
    return 2 * i
```

```
RIGHT(i)
    // restituisce il figlio dx del nodo i
    return 2 * i + 1
```

```
PARENT(i)
    // restituisce il genitore del nodo i
    return ⌊i/2⌋
```

4.1.1 Max Heap

Max Heap è uno heap che soddisfa la seguente proprietà:

$$\begin{gathered}\forall \text{ nodo } A[i],\\ A[i] \geq \text{discendenti}\\ \Downarrow\\ A[i] \geq A[\mathbf{Left(i)}],\ A[\mathbf{Right(i)}]\end{gathered}$$

Equivalentemente

$$\begin{gathered}\forall \text{ nodo } A[i],\\ A[i] \leq \text{antenati}\\ \Downarrow\\ A[i] \leq A[\mathbf{Parent(i)}]\end{gathered}$$

Osservazioni

- Uno heap con un solo elemento è un **Max Heap**.
- Dati due Max Heap T_1 e T_2 e un nodo N, possiamo "combinarli" in uno heap con N come radice, T_1 come **left** e T_2 come **right**.

Ecco ora una procedura che, dato un nodo i, trasforma in un Max Heap il sotto-albero eradicato in esso (con radice i).

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if (l ≤ A.heapsize) and (A[l] > A[i])
    max = l
else
    max = i
if (r ≤ A.heapsize) and (A[r] > A[max])
    max = r
if (max ≠ i)
    A[i] ↔ A[max]
    MAX-HEAPIFY(A, max)
```

Correttezza di MaxHeapify

- **Casi base**: $max = i$, distunguo due casi:
  - i è foglia ($l, r > A.heapsize$);
  - $A[i] \geq A[l], A[r]$;
- **Induzione**: $max \neq i$, distinguo due casi:
  - $max = l,\ A[l] \geq A[i], A[r]$;
  - $max = r,\ A[r] \geq A[i], A[l]$.

Complessità $O(h)$, con h altezza del sotto-albero radicato in i

$$\begin{aligned} n &\geq (2^{(h-1)+1}) + 1 = 2^h \\ h &\leq \log_2 n \\ &\Rightarrow O(h) \cong O(\log n) \end{aligned}$$

Ora vogliamo scrivere una procedura che costruisce un **Max Heap** da un array qualunque.

Quali sono i nodi foglia?

- Se $i \geq \lfloor \frac{n}{2} \rfloor + 1$

$$\begin{aligned} 2i = 2\left(\frac{n}{2} + 1\right) &\geq n + 2 - 1 = n + 1 \\ &\Rightarrow i \text{ foglia} \end{aligned}$$

- Se $i \leq \lfloor \frac{n}{2} \rfloor$

$$\begin{aligned} 2i = 2\lfloor \frac{n}{2} \rfloor &\leq n \\ &\Rightarrow i \text{ non foglia} \end{aligned}$$

```
Build-Max-Heap(A)
A.heapsize = A.length
for i = ⌊A.length/2⌋ down to 1
    Max-Heapify(A, i)
```

L'algoritmo esegue $\frac{n}{2}$ volte `MaxHeapify` (che ha complessità $O(\log n)$), ottenendo una complessità finale $O(n \log n)$, tuttavia questa stima è molto pessimistica.

Definiamo:

- h_T altezza del cammino più lungo dello heap;
- $h_T - 1$ di conseguenza è l'altezza dell'albero meno l'ultimo livello, che è generalmente incompleto.

$$\begin{aligned} n &= \left(2^{(h_T-1)+1} - 1\right) + 1 \\ &= 2^{h_T} \\ h_T &\leq \log n \\ n &\geq 2^{h_T} \end{aligned}$$

$$\begin{aligned} T(n) &= \sum_{h=1}^{\lfloor \log n \rfloor} 2^{h_T-h} \cdot O(h) \\ &\qquad (2^{h_T-h} = \# \text{ chiamate a } \texttt{MaxHeapify} \text{ al livello } h) \\ &= \sum_{h=1}^{\lfloor \log n \rfloor} \frac{2^{h_T}}{2^h} O(h) \qquad (2^{h_T} = n) \\ &= O\Big(\big(\sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}\big)n\Big) = O(n) \qquad \Big(\sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \sum_{h=1}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2\Big) \end{aligned}$$

Passiamo ora all'algoritmo di ordinamento `Heapsort`. La radice di un **Max Heap** contiene il valore massimo. Quindi, la prima operazione, e quella su cui si basa `Heapsort`, consiste nel mettere la radice in ultima posizione.

Es. `A`: 9 8 7 5 7 4 0 4 3 6 1 2	è un max heap.
⇒8 7 5 7 4 0 4 3 6 1 2 9	ignoro l'ultimo elemento, chiamo `MaxHeapify` sulla radice e itero.

Poi chiama `MaxHeapify` sul resto dell'array per renderlo un **Max Heap**, e itera il procedimento sul nuovo array.

```
Heapsort(A)
Build-Max-Heap(A) // O(n)
for i = A.length down to 2
    A[1] ↔ A[i]
    A.heapsize = A.heapsize − 1
    Max-Heapify(A, 1) // O(log n)
```

Complessità $O(n \log n)$.

4.1.2 Code con Priorità

S insieme dinamico di oggetti.

x è l'indice, $x.key$ è il corrispondente valore relativo a quell'indice. Voglio poter eseguire le seguenti operazioni:

- `Insert(S,x)`
- `Max(S)`
- `ExtractMax(S)`
- `IncreaseKey(S,x,`δ`)`
- `ChangeKey(S,x,`δ`)`
- `Delete(S,x)`

Idea Uso un **Max Heap** (A).

MAX(A)
if $A.heapsize = 0$
    error
else return $A[1]$

La procedura `Max(A)` ha complessità costante $\Theta(1)$.

EXTRACT-MAX(A)
$max = A[1]$
$A[1] = A[A.heapsize]$
$A.heapsize = A.heapsize - 1$
MAX-HEAPIFY($A, 1$) // ripristina le proprietà di **MaxHeap**
return max

La procedura `ExtractMax(A)` ha la stessa complessità di `MaxHeapify`: $O(\log n)$.

Per `Insert`, le cose diventano più delicate. L'idea è quella di inserire in coda ad A: in questo modo, l'unico elemento che potrebbe compromettere la proprietà di **Max Heap** è la cella di indice i (nel nostro caso, l'ultima). Deve valere la proprietà:

$$\text{Per ogni } j \neq i$$
$$A[j] \leq \text{antenati}$$

Non possiamo dire nulla su i. Va ristabilita la proprietà di **Max Heap**: per fare ciò usiamo la procedura `MaxHeapifyUp`.

```
Max-Heapify-Up(A, i)
if (i > 1) and (A[i] > A[Parent(i)])
    A[i] ↔ A[Parent(i)]
    Max-Heapify-Up(A, Parent(i))
```

Correttezza di MaxHeapifyUp

Casi base

`(i = 1)` ok, non faccio nulla;

`(A[i]` $\leq$ `A[Parent(i)])` ok, la proprietà di **Max Heap** è mantenuta.

Induzione

`(A[i] > A[Parent(i)])` scambio le due celle. I discendenti (sottoalberi) della nuova cella `A[i]` mantengono la proprietà di **Max Heap**.

Complessità $O(\log i)$, nel caso peggiore $O(\log n)$.

Ecco ora lo pseudocodice della funzione `Insert`.

```
Insert(A, x)
A.heapsize = A.heapsize + 1
A[A.heapsize] = x
Max-Heapify-Up(A, A.heapsize)
```

`Insert` ha complessità $O(\log n)$, la stessa di `MaxHeapifyUp`.

```
Increase-Key(A, i, δ)
    // Precondizione: δ ≥ 0
A[i] = A[i] + δ
Max-Heapify-Up(A, i)
```

`IncreaseKey` ha complessità $O(\log n)$.

```
Change-Key(A, i, δ)
A[i] = A[i] + δ
if δ > 0
    Max-Heapify-Up(A, i)
else // δ ≤ 0
    Max-Heapify(A, i)
```

`ChangeKey` è come `IncreaseKey`, ma può utilizzare valori di δ qualsiasi, ed è corretto per la seguente proprietà:

$$\text{Se per ogni } j \neq i \quad A[j] \geq \text{discendenti}$$
$$\Rightarrow \text{dopo } \texttt{MaxHeapify} \text{ ho un } \textbf{MaxHeap}$$

```
DELETE(A, i)
    old = A[i]
    A[i] = A[A.heapsize]
    A.heapsize = A.heapsize − 1
    if old ≤ A[i]
        MAX-HEAPIFY-UP(A, i)
    else
        MAX-HEAPIFY(A, i)
```

`Delete` ha complessità $O(\log n)$.

4.2 Quicksort

Il Quicksort è probabilmente l'algoritmo di ordinamento più utilizzato e nella pratica efficiente, nonostante abbia un caso pessimo di $O(n^2)$.

- Caso pessimo $O(n^2)$;
- Caso medio e migliore $O(n \log n)$;
- costanti basse.

Si basa sul paradigma del *divide et impera*:

- *Divide*
  - → Secglie un *pivot* x in `A[p, r]`;
  - → partiziona in `A[p, q-1]` $\leq x$ e `A[q+1, r]` $\geq x$;
- *Impera*

  Ricorre su `A[p, q-1]` e `A[q+1, r]`;
- *Combina*

  (Non fa nulla).

Pseudocodice Segue lo pseudocodice del `Quicksort`.

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
x = A[r] // pivot A[r]
i = p − 1
// A[p, i] ≤ x
// A[i+1, j-1] > x
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        A[i] ↔ A[j]
A[i + 1] ↔ A[r]
return i + 1
```

4.2.1 Correttezza di Quicksort

Caso base array già ordinato, 0 o 1 elemento.

Induzione Abbiamo, dopo `Partition`

$\leq$ `A[q]`	`A[q]`	$\geq$ `A[q]`

`Quicksort(A, p, q)`	$\leq$ `A[q]`, ord	`A[q]`	$>$ `A[q]`
`Quicksort(A, q+1, r)`	$\leq$ `A[q]`, ord	`A[q]`	$>$ `A[q]`, ord

Esempio Dato l'array A, scelgo come **pivot** x l'ultimo elemento.

9	6	0	8	4	2

pivot: $\boxed{2}$

`i` punta alla cella 0 (ossia nessuna cella)
`j` punta alla cella 1: $\boxed{9}$
$9 > 2$? Sì $\Rightarrow$ `j++`
$6 > 2$? Sì $\Rightarrow$ `j++`
$0 > 2$? No $\Rightarrow$ `i++`, `A[i]` $\leftrightarrow$ `A[j]`, `j++`

0	6	9	8	4	2

pivot: $\boxed{2}$

`i` punta alla cella 1: $\boxed{0}$
`j` punta alla cella 4: $\boxed{8}$
$8 > 2$? Sì $\Rightarrow$ `j++`
$4 > 2$? Sì $\Rightarrow$ `j++`
Scambio `A[i+1]` con `x`, ottenendo

0	2	9	8	4	6

I primi due $(i+1)$ elementi sono ordinati:

0	2

Chiamo ricorsivamente `Quicksort` con $q = i + 1$.

9	8	4	6

pivot: $\boxed{6}$

`i` punta alla cella 0 (ossia nessuna cella)
`j` punta alla cella 1: $\boxed{9}$
$9 > 6$? Sì $\Rightarrow$ `j++`
$8 > 6$? Sì $\Rightarrow$ `j++`
$4 > 6$? No $\Rightarrow$ `i++`, `A[i]` $\leftrightarrow$ `A[j]`, `j++`

$\boxed{4}\boxed{8}\boxed{9}\boxed{6}$ **pivot:** $\boxed{2}$

`i` punta alla cella 1: $\boxed{4}$
`j` punta alla cella 4: $\boxed{6}$, quindi ho finito.
Scambio `A[i+1]` con `x`, ottenendo

$\boxed{4}\boxed{6}\boxed{9}\boxed{8}$

I primi due $(i + 1)$ elementi sono ordinati:

$\boxed{4}\boxed{6}$

Chiamo ricorsivamente `QuicsSort` con $q = i + 1$.

$\boxed{9}\boxed{8}$ **pivot:** $\boxed{8}$

`i` punta alla cella 0 (ossia nessuna cella)
`j` punta alla cella 1: $\boxed{9}$
$9 > 8$? Sì $\Rightarrow$ `j++`
Ho finito, scambio `A[i+1]` con `x`, ottenendo

$\boxed{8}\boxed{9}$

Guardando l'array completo ottengo il risultato atteso:

$\boxed{0}\boxed{2}\boxed{4}\boxed{6}\boxed{8}\boxed{9}$

4.2.2 Complessità di Quicksort

`Partition` costa $\Theta(n)$

$$T^{QS} = \Theta(n) + T^{QS}(q - p) + T^{QS}(n - (q - p) - 1)$$
$$q - p < n$$

Caso peggiore

$$\begin{aligned} T^{QS} = &\Theta(n) + T^{QS}(n-1) = \Theta(n^2) \qquad (\Theta(n) = cn) \\ &T(n) \\ &cn \\ &cn - 1 \\ &cn - 2 \\ &\dots \\ &d \end{aligned}$$

$$\begin{aligned}\sum_{j=1}^{n-1} c(n-j) + d &= \sum_{k=1}^{n} ck + d = \\ &= c\sum_{k=1}^{n} k + d \qquad \left(\frac{c(n+1)n}{2} + d = \Theta(n^2)\right)\end{aligned}$$

$$T(n) = \Theta(n^2) \Rightarrow \begin{cases} = O(n^2) \\ = \Omega(n^2) \end{cases}$$

- $T(n) = O(n^2)$

$$\begin{aligned}T(n) &= O(n^2) \Rightarrow T(n) \leq cn^2 \qquad \forall\ n \geq n_0,\ c > 0 \\ &= T(n-1) + \Theta(n) \leq dn \\ &\leq c(n-1) + dn \\ &= cn^2 - 2cn + c + dn \leq cn^2 \\ &\quad 2cn - dn - c \geq 0 \\ &\quad n(2c-d) - c \geq 0 \qquad \text{ok, } c > \frac{d}{2} \\ T(n) &= O(n^2)\end{aligned}$$

- $T(n) = \Omega(n^2)$ analogo.

Caso migliore

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

Caso medio Qualunque partizionamento proporzionale da complessità $\Theta(n \log n)$, come ad esempio

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) = \Theta(n \log n)$$

Solo il caso in cui una delle due partizioni è costante, si ricade nel caso pessimo. Per ovviare al problema, si può utilizzare una versione di `Partition` che rende impossibile il partizionamento costante.

Randomized-Partition(A, p, r)

```
q = Random(p, r)
A[q] ↔ A[r]
return Partition(A, p, r)
```

4.3 Quicksort a Tre Partizioni

`Quicksort` con `RandomizedPartition` funziona bene ed evita, quasi in ogni circostanza, di imbattersi nel caso pessimo, ad eccezione di un caso particolare: se in **input** viene dato un array con tutti gli elementi uguali, si ottiene il temuto caso pessimo $O(n^2)$.

Per ovviare al problema, è sufficiente partizionare `Quicksort` in tre partizioni invece di due. Dato un **pivot** x, partizioniamo A nel seguente modo:

$< x$	$= x$	$> x$

Durante l'algoritmo, la disposizione sarà questa:

$< x$	$= x$		$> x$

(La cella vuota è la regione ancora da esplorare).

```
TRIPARTITION(A, p, r)
x = A[r]
i = p − 1
k = p
j = r
while k < j
    if A[k] < x
        i = i + 1
        A[i] ↔ A[k]
        k = k + 1
    else if A[k] > x
        j = j − 1
        A[j] ⇔ A[k]
    else
        k = k + 1
// k = j
A[j] ⇔ A[r]
return (i + 1, j) // restituisce una coppia di valori
```

```
QUICKSORT(A, p, r)
if p < r
    q1, q2 = TRIPARTITION(A, p, r)
    QUICKSORT(A, p, q1 − 1)
    QUICKSORT(A, q2 + 1, r)
```

4.4 Limite Inferiore

`Input`: $a_1 \ldots a_n$

`Output`: permutazione $a'_1 \ldots a'_n$ tale che

$$a'_1 \leq a'_2 \leq \cdots \leq a'_n$$

Confronti e assegnamenti Osservazioni:

→ Se "conto" solo alcune operazioni il limite inferiore vale in generale. Consideriamo solo l'operatore di confronto;

→ Elementi tutti distinti ($a_i \neq a_j$ se $i \neq j$), l'operatore di confronto == restituisce sempre False.

4.5 Albero di Decisione

È una rappresentazione "astratta" delle possibili esecuzioni di un algoritmo di ordinamento su un input di dimensione fissata $A[1 \ldots n]$.

→ nodi interni:

$$i \ : \ j \Rightarrow \text{confronta } A[i] \leq A[j]$$

→ **foglie** (ogni foglia è una possibile permutazione)

Ecco un esempio di **Albero di Decisione** per l'array $A[a_1, a_2, a_3]$ con

$$a_1 = 1, \ a_2 = 2, \ a_3 = 3$$

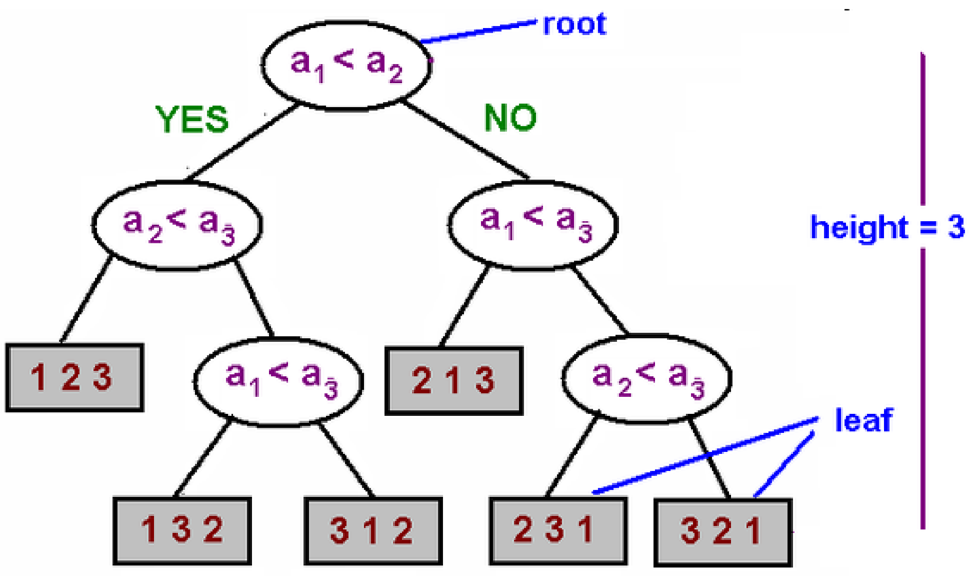


Figura 5: Albero di decisione per l'array $A[1, 2, 3]$

Osservazione

- Altezza dell'albero di decisione = limite inferiore per caso pessimo

$$\begin{array}{ll} \text{per IS} & n^2 \\ \text{per MS} & n \log n \end{array}$$

- Ogni foglia ha una sola permutazione. Ogni permutazione compare (almeno) in una foglia.

In generale, le foglie contengono <u>tutte</u> le permutazioni.

$$\# \text{ foglie} \geq \# \text{ permutazioni} = n! \qquad (\# \text{ foglie} \leq 2^h)$$

$$\begin{aligned} h &\geq \log_2 n! \\ &\geq \log_2 \left(n(n-1)(n-2) \dots \frac{n}{2} \right) \\ &\geq \log_2 \left(\frac{n}{2} \left(\frac{n}{2} - 1 \right) \left(\frac{n}{2} - 2 \right) \dots \frac{n}{2} \right) \\ &\geq \log_2 \left(\frac{n}{2} \right)^{\left(\frac{n}{2}\right)} = \frac{n}{2} (\log_2 n - \log_2 2) = \frac{n}{2} (\log_2 n - 1) = \Theta(n \log n) \end{aligned}$$

Inoltre,

- ◦ # operazioni $\geq h = \Omega(n \log n)$
- ◦ `Heapsort, MergeSort` $O(n \log n))$

$\Rightarrow$ ordinamento (bastato su confronti) $\Theta(n \log n)$

4.6 Counting Sort

Esistono degli algoritmi di ordinamento che, in certe condizioni e per certi input, permettono di ordinare in tempo lineare $\Omega(n)$

Assumo

- interi;
- in $[0, k]$

`Input`: $A[1\,..\,n]$ con $A[j] \in [0, k] \quad \forall\, 1 \leq j \leq n$;

`Output`: $B[1\,..\,n]$ permutazione ordinata di A;

`Supporto`: $C[0\,..\,k]$.

Counting-Sort(A, B, k)

$C[0\,..\,k] \leftarrow 0$
for $j = 1$ **to** $A.\mathit{length}$
    // $C[x] =$ # elementi in A con valore x
    $C[A[j]] = C[A[j]] + 1$
for $i = 1$ **to** k
    // $C[x] =$ # elementi in A con valore $\leq x$
    $C[i] = C[i-1] + C[i]$
for $j = A.\mathit{length}$ **down to** 1
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$

Complessità

$C[0, k] \leftarrow 0$	$\Theta(k)$
`for j=1`...	$\Theta(n)$
`for i=1`...	$\Theta(k)$
`for j=A.length`...	$\Theta(n)$

Somma $\Theta(n + k)$ con $k = \Theta(1) \Rightarrow \Theta(n)$

Problema di memoria Il problema di `CountingSort` è la memoria. Infatti, al crescere di k, la memoria richiesta per allocare `C` cresce esponenzialmente.

Dimensione k	Memoria occupata da `C[]`
1 Byte = 8 bit	2^8Bytes = 256Bytes
2 Bytes = 16 bit	2^{16}Byte $\cdot$ 2Bytes = 256Megabytes
8 Bytes = 64 bit	2^{64}Byte $\cdot$ 8Bytes = 512Terabytes

4.6.1 Proprietà di Stabilità

Dato $A[1..n]$ in input, se $A[i] \leq A[j]$ con $i \leq j$, allora nell'output $A[i]$ e $A[j]$ sono nello stesso ordine relativo.

Algoritmi stabili:

- `MergeSort`
- `InsertionSort`

Algoritmi non stabili:

- `CountingSort`
- `Quicksort`
- `Heapsort`

4.7 Radix Sort

Il Radix Sort è un algoritmo di ordinamento in tempo lineare $O(n)$, come `CountingSort`, che risolve i problemi di memoria di quest'ultimo.

L'idea è quella di ordinare cifra per cifra, dalla cifra meno significativa alla più significativa con un algoritmo **stabile**.

(iniziale)	(terza cifra)	(seconda cifra)	(prima cifra)
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

`Input`: `A[1..n]` con `A[i]` di d cifre e base b, `A[i]`$= a_d a_{d-1} \dots a_1$.

Radix-Sort(A, d)

1 **for** $j = 1$ **to** d
2 ordina A rispetto alla cifra j // $A^j[i] = a_j a_{j-1} \dots a_1$
con Counting-Sort // A^{j-1}*ordinato*

Correttezza di Radix Sort

- **Inizializzazione**: ok;
- **Mantenimento**: se A^{j-1} è ordinato e ordino rispetto alla j-esima cifra con un algoritmo stabile, allora A^j è ordinato.

$$i < i' \Rightarrow A^j[i] \leq A^j[i']$$

$$\begin{aligned} \text{Siano } A^j[i] &= a_j a_{j-1} \dots a_1 \\ A^j[i'] &= a'_j a'_{j-1} \dots a_1 \end{aligned}$$

Posso distinguere due casi:

1.

$$\begin{aligned} a_j \neq a'_j &\Rightarrow a_j < a'_j \\ &\Rightarrow A^j[i] < A^j[i'] \end{aligned}$$

2.

$$\begin{aligned} a_j = a'_j &\Rightarrow A^j[i] \le A^j[i'] \qquad \text{(stabilità)} \\ &\Rightarrow A^j[i] = a_j A^j[i] \le \\ &\le A^j[i'] = a'_j A^{j-1}[i'] \end{aligned}$$

Complessità

$$\begin{gathered} d \text{ volte } \texttt{CountingSort} \;\; \Theta(n+b) \Rightarrow \Theta\big(d(n+b)\big) = \Theta(n) \\ \text{con } d \text{ cifre } = \Theta(1), \;\text{base } b = \Theta(n) \end{gathered}$$

m bit, r bit per cifra, $\frac{m}{r}$ cifre, base 2^r

$$\begin{aligned} \Theta(\frac{m}{r}(m+2^r)) &= & r = \log_2 n \\ &= \Theta(\frac{m}{\log n}(n + 2^{\log n})) \\ &= \Theta(\frac{m}{\log n} n) & m = O(\log n) \\ &= \Theta(n) \end{aligned}$$

5 Tabelle Hash

$$U \text{ universo delle chiavi}$$
$$U = \{0, 1, \ldots, |U| - 1\}$$
$$T[0\,..\,|U| - 1] \text{ tabella hash}$$

$$T[k] \text{ contiene} \quad \begin{cases} \text{elemento } x \text{ con } x.key = k & \text{se c'è} \\ \textsc{nil} & \text{altrimenti} \end{cases}$$

Insert(T, x)
$T[x.key] = x$ // $\Theta(1)$

Delete(T, x)
$T[x.key] = $ nil // $\Theta(1)$

Search(k)
return $T[k]$ // $\Theta(1)$

Problema e.g. consideriamo che la **key** sia di 8 caratteri (e 8 bit per rappresentare un carattere). Risulta molto costosa in termini di memoria la tabella hash.

$$2^8 \ldots 2^8$$
$$(2^8)^8 = 2^{64} \cong 10^{19}$$

Idea

$$U = \{0, 1, \ldots, |U| - 1\}$$
$$T[0 \ldots m - 1] \qquad m << |U|$$

La "traduzione" per ottenere $x.key$ da x cosa comporta?

$$h\colon U \to \{0, 1, \ldots, m - 1\} \text{ funzione di } \textbf{hashing}$$
$$n = \# \text{ elementi memorizzati nella tabella } T$$
$$m = \# \text{ celle}$$

Se $n > m$, esisteranno $x_1, x_2 : h(x_1.key) = h(x_2.key) \Rightarrow$ conflitto

Abbiamo **due soluzioni**:

1. **Chaining** (5.1);
2. **Open Addressing** (5.2).

5.1 Chaining

Il **Chaining** propone come soluzione quella di mettere sulla tabella liste dinamiche di elementi, invece che singoli elementi, in modo che in caso si incorra in una cella già occupata dopo un **hashing**, l'elemento venga inserito in coda (o in testa) alla lista.

Idea `T[i]` = lista elementi x tali che $h(x.key) = i$

Insert(T, x)
1 Inserisci x nella lista $T[h(x.key)]$ // $O(1)$

Delete(T, x)
1 Elimina x da $T[h(x.key)]$ // $O(1)$

Search(T, k)
1 Cerca in $T[h(k)]$ un elemento x tale che $x.key = k$ // $O(n)$

`Search` ha una complessità di $O(n)$, e questo è inaccettabile.

$$n = \#\text{ elementi inseriti}$$
$$m = \text{dimensione di } T$$
$$\alpha = \frac{n}{m} \quad \text{fattore di carico}$$
$$\alpha \text{ può essere } <, = \text{ oppure } > \text{ di } 1$$

5.1.1 Hashing Uniforme Semplice

Ogni elemento di **input** è "mandato" da h con la stessa probabilità $\left(\frac{1}{m}\right)$ in una delle m celle.

Caso medio $\Theta(1 + \alpha)$, 1 è l'accesso alla tabella.

Consideriamo $n_1, n_2, \ldots, n_{m-1}$ la lunghezza delle m liste. La lunghezza attesa di una lista è:

$$E[n_j] = \sum_{i=1}^{n} \frac{1}{m} \cdot 1 = \frac{n}{m} = \alpha$$

Ricerca di una chiave La chiave può essere:

- **Assente**. `Search(k)`, `k` non c'è.
  - Calcolo `h(k)` $\rightarrow (\Theta(1))$;
  - Accedo a `T[h(k)] = j` $\rightarrow (\Theta(1))$;
  - Scorro n_j elementi $(n_j = \alpha) \rightarrow (\Theta(\alpha))$.

  Nel complesso, ho $\Theta(1+\alpha)$
- **Presente**. `Search(k)`, `k` presente.
  - `h(k)` e `T[h(k)]`

  Se $x_1, x_2, \ldots x_n$ sono gli elementi inseriti

  Costo della ricerca di x_i:

$$\begin{aligned} &1 + \#\text{ elementi} \quad x_j : j > 1,\ h(x_i.key) = h(x_j.key) \\ &= 1 + \sum_{j=i+1}^{n} (prob\ h(x_i.key) = h(x_j.key)) \\ &= 1 + \sum_{j=i+1}^{n} \frac{1}{m} = 1 + \frac{n-i}{m} \end{aligned}$$

$$\begin{aligned} &\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) \\ &= \frac{1}{n}\left(n + \sum_{i=1}^{n}\frac{n-i}{m}\right) = \frac{1}{n}\left(n + \frac{1}{m}\sum_{z=0}^{n-1} z\right) \\ &= 1 + \frac{1}{m\cdot n}\cdot\frac{n(n-1)}{2} = 1 + \frac{n}{2m} - \frac{1}{2m}\cdot\left(\frac{n}{n}\right) \qquad \left(\alpha = \frac{n}{m}\right) \\ &= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1+\frac{\alpha}{2}) = \Theta(1+\alpha) \end{aligned}$$

$$\begin{gathered} \alpha \text{ costante} \\ n = O(m) \\ n \leq k\cdot m \\ \alpha = \frac{n}{m} \leq k \\ \Rightarrow \Theta(1+\alpha) = \Theta(1) \end{gathered}$$

- $h \colon U \rightarrow \{0, 1, \ldots, m-1\} \Rightarrow h(x) = 0$

5.1.2 Funzioni di Hash

Una **funzione di hash** deve soddisfare la proprietà di **hashing uniforme**, ossia

> "Ogni chiave ha la stesso probabilità $\frac{1}{m}$ di essere mandata in una qualsiasi delle m celle, indipendentemente dalle chiavi inserite precedentemente."

Consideriamo:

- $k \in [0, 1)$ $(0 \leq k < 1)$, k chiave, estratta in modo indipendente dalla distribuzione uniforme (non realistica).
- Allora $h(k) = \lfloor mk \rfloor$ soddisfa la proprietà di **hashing uniforme**.

L'ipotesi di hash uniforme semplice dipende dalle probabilità con cui vengono estratti gli elementi da inserire; probabilità che in genere non sono note. Le funzioni hash che descriveremo assumono che le chiavi siano degli interi non negativi.

Metodo della Divisione

$$U = \{0, 1, \ldots, |\cup| - 1\}$$
$$h(k) \in \{0, 1, \ldots, m - 1\}$$
$$h(k) = k \bmod m$$

- $m = 2^p$ caso pessimo;
- $m = 2^p - 1$ caso non buono. 2^p cifre base.

La soluzione migliore è quella di scegliere chiavi lontane dalle potenze di 2, meglio ancora se numeri primi.

Metodo della Moltiplicazione

$$k \in U$$

$$0 < A < 1 \text{ fissato}$$

$$h(k) = m(kA \bmod 1) \qquad \text{Miglior } A : \frac{\sqrt{5} - 1}{2}$$

$$
\begin{aligned}
m &= 2^p \quad w = \#\text{ bit parola} \\
A &= \frac{q}{2^w} \quad 0 < q < 2 \\
& m(kA \bmod 1) \\
& = m\left(k\frac{q}{2^w} \bmod 1\right) \qquad (\textbf{shift} \text{ di } w \text{ bit, prendo la parte decimale} \\
& \qquad\qquad\qquad\qquad\qquad ka \bmod 1 \text{ e la moltiplico per } m = 2^p)
\end{aligned}
$$

5.1.3 Hashing Universale

Per avere una distribuzione più uniforme delle chiavi nelle liste e non dipendente dall'input, possiamo usare la **randomizzazione**.

Insieme H di funzioni di hash. Scelgo randomicamente $h \in H$.

Sotto certe ipotesi ottengo per `Search`:

$$\Theta(1+\alpha)$$

Def (Hashing universale) $\forall\ k_1, k_2 \in U,\ k_1 \neq k_2$

$$
\begin{aligned}
|\{h \in H : h(k_1) = h(k_2)\}| &\leq \frac{|H|}{m} \\
prob(h(k_1) = h(k_2)) = \frac{|\{h \in H : h(k_1) = h(k_2)\}|}{|H|} &\leq \frac{1}{m}
\end{aligned}
$$

Teorema Con il **chaining**, H è universale $\forall\ k \in U,\ j = h(k)$

$$\text{Costo medio } \Theta(1+\alpha) \begin{cases} k \text{ non è in } T \to E[n_j] \leq \alpha \\ k \text{ è in } T \to E[n_j] \leq 1+\alpha \end{cases}$$

5.2 Open Addressing

$h(k, i)$: k è la chiave, i è il tentativo.

Provo con $h(k, 0)$: se capito in una cella occupata, provo con $h(k, 1)$, poi $h(k, 2)$ e così via, fino a che non trovo una cella libera.

Per esplorare tutta la tabella:

$$h(k, 0), h(k, 1), \dots, h(k, m-1) \quad \forall\ k \in U$$

che è una permutazione di

$$\{0, 1, \dots, m-1\}$$

```
INSERT(T, x)
i = 0
repeat
    j = h(x.key, i)
    if (T[j] = NIL) or (T[j] = DELETED) // posizione libera
        T[j] = x
        return j
    i = i + 1
until i = m
error
```

```
SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j].key = k
        return j
    i = i + 1
until (i = m) or (T[j] = NIL)
return NOT FOUND
```

```
DELETE(T, j)
T[j] = DELETED
```

L'**Open Addressing** risulta una soluzione inefficiente in caso avvengano molte cancellazioni.

5.2.1 Hashing Uniforme

Per ogni elemento di input, tutte ($m!$) le sequenze di ispezione sono equiprobabili.

5.2.2 Funzioni di Hash

1. **Ispezione lineare**. Sia $h'(k)$ funzione di hash "ordinaria". Se ricado in una cella occupata, mi sposto su quella immediatamente successiva.
$$h(k,i) = (h'(k) + i) \bmod m$$
Caratteristiche:
   - è semplice;
   - poche permutazioni (m dipende solo da $h'(k)$);
   - causa addensamenti di celle occupate (**addensamento primario**).
2. **Ispezione quadratica**. Fisso $h'(k)$.
$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m, \qquad c_2 > 0$$
3. **Doppio Hash**. Fisso $h_1(k)$, $h_2(k)$
$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$
Osservazioni:
   - I salti sono di dimensione $h_2(k)$ all'incrementare di i;
   - Ci sono m^2 sequenze di ispezione;
   - $h_2(k)$ e m primi tra loro $\quad (MCD = 1)$;
   - $h(k,0), h(k,1), \dots, h(k,m-1)$ è permutazione di $\{0,1,\dots,m-1\}$;
   - $i, i' < m \quad h(k,i) = h(k,i') \Rightarrow i = i' \quad$ (**iniettività**)
$$h(k,i)\colon \{0,\dots,m-1\} \to \{0,\dots,m-1\}$$
iniettiva $\Rightarrow$ biiettiva
$$\begin{gathered} h(k,i) = h(k,i') \\ (h_1(k) + ih_2(k)) \bmod m = (h_1(k) + i'h_2(k)) \bmod m \\ ((i-i')h_2(k)) \bmod m = (ih_2(k) - i'h_2(k)) \bmod m = 0 \\ (i-i') \bmod m = 0 \\ i \geq i' \quad i - i' < m-1 \\ \Rightarrow i - i' = 0 \\ \Rightarrow i = i' \end{gathered}$$
Scelgo $m = 2^p$, $h_2(k) = 1 + 2h'(k)$, $h'(k)$ qualunque.
es. $h_2(k) = 1 + h'(k \bmod m')$ $\quad$ con $m' < m, m$ primo

Costo Il costo della `Search` con **hashing uniforme** si può riassumere come segue.

$$0 \leq \alpha = \frac{n}{m} \leq 1$$

Ricerca di una chiave non presente

(a) $\frac{1}{1-\alpha}$ se $\alpha < 1$

(b) m se $\alpha = 1$

Probabilità di ispezionare la i-esima cella

cella	**probabilità**
$i = 0$	1
$i = 1$	prob. cella 0 occupata: $\frac{n}{m}$
$i = 2$	prob. cella 1 occupata: $\frac{n}{m} \cdot \frac{n-1}{m-1}$
...	
i	$\frac{n}{m} \cdot \frac{n-1}{m-1} \ldots \frac{n-i+1}{m-i+1} \leq \alpha \cdot \alpha \ldots \alpha = \alpha^i$

Valore atteso per # tentativi

$$1 + \alpha + \alpha^2 + \cdots + \alpha^{i-1} + \cdots + \alpha^{m-1}$$

(a) $\alpha < 1 \Rightarrow \frac{1-\alpha^m}{1-\alpha} \leq \frac{1}{1-\alpha}$

(b) m

Ricerca di una chiave presente

(a) $\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$ $\alpha < 1$

(b) $1 + \log m$ $\alpha = 1$

Costo atteso per k_i:

$$\frac{1}{1-\alpha_{i-1}} = \frac{1}{1-\frac{i-1}{m}} \qquad \left(\alpha_{i-1} = \frac{i-1}{m} < 1\right)$$
$$= \frac{m}{m-i+1}$$

Il costo atteso è la media per $i = 1, \ldots, n$

$$= \frac{1}{n}\sum_{i=1}^{n} \frac{m}{m-i+1} = \frac{m}{n}\sum_{i=1}^{n} \frac{1}{m-i+1} \qquad \left(\frac{m}{n} = \frac{1}{\alpha}\right)$$
$$= \frac{1}{\alpha}\sum_{j=m-n+1}^{m} \frac{1}{j}$$

- Se $\alpha < 1$

$$\begin{aligned}
&\le \frac{1}{\alpha}\int_{n-m}^{m}\frac{1}{x}\mathrm{d}x \\
&= \frac{1}{\alpha}\left(\log m - \log(m-n)\right) = \frac{1}{\alpha}\left(\log\frac{m}{m-n}\right) \\
&= \frac{1}{\alpha}\log\frac{1}{\frac{m-n}{m}} \\
&= \frac{1}{\alpha}\log\left(\frac{1}{1-\left(\frac{n}{m}\right)}\right) = \frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)
\end{aligned}$$

- Se $\alpha = 1$

$$\begin{aligned}
\sum_{l=1}^{m}\frac{1}{l} = 1 + \sum_{l=2}^{m}\frac{1}{l} &\le \int_{1}^{m}\frac{1}{x}\mathrm{d}x \\
&= 1 + (\log m - \log 1) = 1 + \log m
\end{aligned}$$

Confrontiamo le complessità dei due casi.

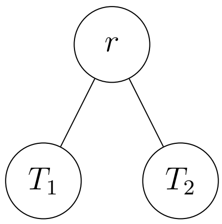
α	$\frac{l}{1-\alpha}$	$\frac{1}{\alpha}\log\left(\frac{1}{1-\alpha}\right)$
$\alpha = 0.3$	1.43	1.19
$\alpha = 0.5$	2.00	1.39
$\alpha = 0.7$	3.33	1.72
$\alpha = 0.9$	10	2.56
$\alpha = 0.99$	100	4.65

6 Alberi

6.1 Alberi Binari di Ricerca

Definizione induttiva

- $\varnothing$ è un albero;
- Se r è un nodo, T_1 e T_2 alberi $\Rightarrow r(T_1, T_2)$ è un albero.



Ogni nodo x ha i seguenti campi:

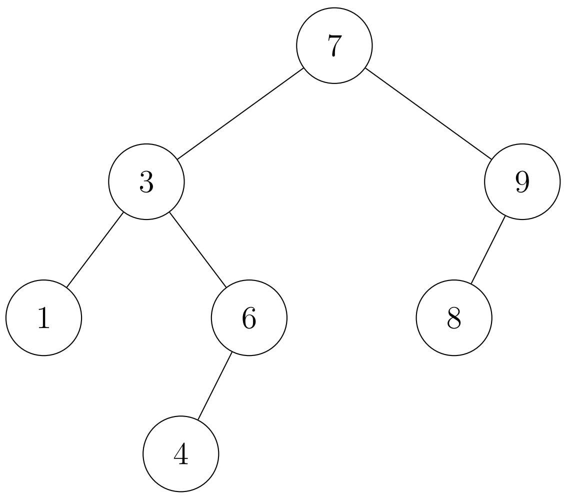
- $x.p$
- $x.key$
- $x.left$
- $x.right$

Proprietà $\forall\, r$

→ Per ogni nodo y in T_1 $\quad y.key \leq x.key$;

→ Per ogni nodo y in T_2 $\quad y.key \geq x.key$.

Esempio Ecco un **albero binario di ricerca** d'esempio:



6.1.1 Visita Simmetrica

La visita simmetrica (ordine infisso) visita i nodi in ordine crescente.

```
IN-ORDER(x)
if x ≠ NIL
    IN-ORDER(x.left)
    PRINT(x) // Θ(1)
    IN-ORDER(x.right)
```

Complessità

$$T(n) = \begin{cases} c & n = 0 \\ T(k) + T(n-k-1) + d & n > 0,\ k < n \end{cases}$$

Stima di complessità: $T(n) = (c+d)n + c$.
Vediamo la dimostrazione (per induzione).

$(n = 0)$ $T(n) = c = (c+d) \cdot 0 + c$

$(n \to n+1)$ $T(n) = T(k) + T(n-k) + d$. Non basta l'induzione ordinaria, usiamo l'**induzione completa**.

$(n > 0)$ Proprietà vera per $n' < n$

$$\begin{aligned} T(n) &= T(k) + T(n-k-1) + d \\ &\qquad \text{con } T(k) = (c+d)k + c \text{ e } T(n-k-1) = (c+d)(c-k-1) + c \\ &= (c+d)(\cancel{k} + n - \cancel{k} - 1) + 2c + d \\ &= n(c+d) - c - d + 2c + d \\ &= n(c+d) + c - \cancel{d} + \cancel{d} \\ &\cong \Theta(n) \end{aligned}$$

6.1.2 Ricerca

Ricerca di una chiave k in un albero radicato nel nodo x.

- Se x è NIL $\Rightarrow$ restituisce NIL;
- Altrimenti se $x.key = k \Rightarrow$ restituisce x;
- Altrimenti, ricorre sul prossimo nodo.

```
SEARCH(x, k)
if (x = NIL) or (x.key = k)
    return x
else if k < x.key
    return SEARCH(x.left, k)
else
    return SEARCH(x.right, k)
```

Complessità La complessità è l'altezza h dell'albero, ovvero $O(h)$.

Vediamo una versione iterativa di `Search`.

```
SEARCH-IT(x, k)
while (x ≠ NIL) or (x.key ≠ k)
    if k < x.key
        x = x.left
    else
        x = x.right
return x
```

Procedura che restituisce il **minimo** di un albero:

```
MIN(T)
x = T.root
if x = NIL
    return NIL
else
    while x.left ≠ NIL
        x = x.left
return x
```

Procedura che restituisce il **massimo** di un albero.

```
MAX(T)
x = T.root
if x = NIL
    return NIL
else
    while x.right ≠ NIL
        x = x.right
return x
```

Complessità $O(h)$

6.1.3 Successore

Si intende il nodo elencato dopo un nodo x passato come parametro in una visita simmetrica.

Se le chiavi fossero tutte distinte, allora il **successore** di x è il minimo tra i "nodi più grandi di x".

- Se x ha un figlio destro, il **successore** è Min($x.right$);
- Altrimenti, il successore è l'antenato più vicino di cui x è nel sottoalbero sinistro.

```
Successor(x)
if x.right ≠ NIL
    return Min(x.right)
else
    y = x.p
    while (y ≠ NIL) and (x = y.right)
        x = y
        y = y.p
    return y
```

Complessità $O(h)$

6.1.4 Inserimento

```
INSERT(T, z)
x = T.root
y = NIL
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else
        x = x.right
z.p = y
if y = NIL
    T.root = z
else
    if z.key < y.key
        y.left = z
    else
        y.right = z
```

Complessità $O(h)$

6.1.5 Eliminazione

Distingueremo 2 casi nell'algoritmo:

(1) z ha al più un figlio;

(2) z ha due figli.

Per fare ciò, usiamo una funzione ausiliaria `Transplant`, con complessità $O(1)$.

```
TRANSPLANT(T, u, v)
if u.p = NIL
    T.root = v
else
    if u = u.p.left
        u.p.left = v
    else
        u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
DELETE(T, z)
if z.left = NIL
    TRANSPLANT(T, z, z.right)
else if z.right = NIL
    TRANSPLANT(T, z, z.left)
else
    y = MIN(z.right)
    if y.p ≠ z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    y.left = z.left
    y.left.p = y
    TRANSPLANT(T, z, y)
```

Complessità $O(h)$

6.2 Red-Black Trees

I **Red-Black Trees** sono ABR i cui nodi hanno un campo **colore** $x.color$, che può essere:

- RED per il rosso;
- BLACK per il nero.

Accorgimento NIL sarà in realtà un nodo, $T.nil$, con $T.nil.color =$ BLACK.

Caratteristiche **RB-Tree** è in realtà un ABR tale che:

(1) Ogni nodo x ha $x.color \in \{\text{RED}, \text{BLACK}\}$;

(2) La radice *root* è BLACK;

(3) Le foglie ($T.nil$) sono BLACK;

(4) Se x è RED, i figli sono BLACK;

(5) Per ogni nodo x, ogni cammino da x a una qualsiasi delle foglie ha lo stesso numero di nodi BLACK (calcolato con $bh(x)$).

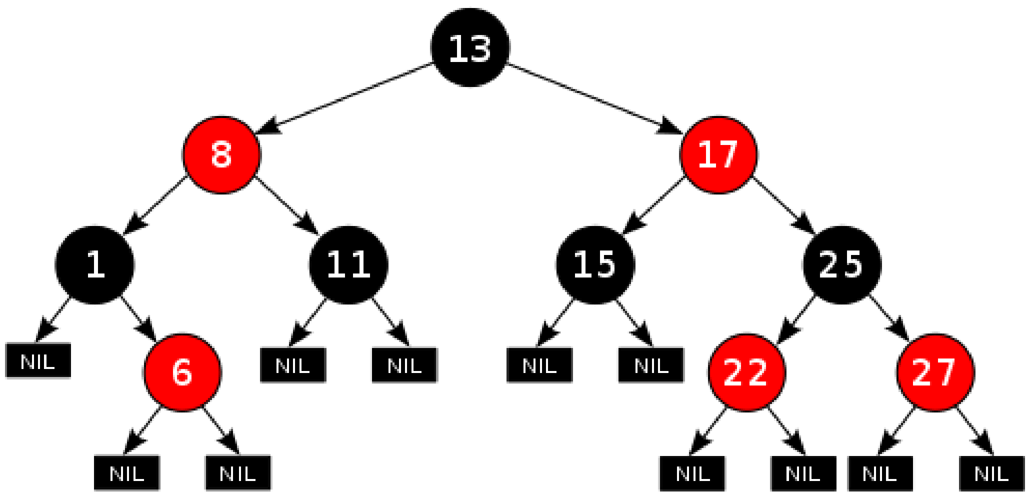


Figura 6: Esempio di un RB-tree.

È possibile notare che:

- In caso non ci fossero nodi rossi, avremo un albero perfettamente bilanciato;
- In ogni cammino, il # di nodi BLACK è almeno la metà del # dei nodi RED

Osservazione Se T è un **RB-Tree** con n nodi interni ($\neq$ NIL) e h altezza, allora vale

$$h \leq 2\log(n+1)$$

Dimostrazione Consideriamo

$$n_x \geq 2^{bh(x)} - 1$$

La dimostrazione è per induzione su h_x (altezza del sotto-albero radicato in x).

($h_x = 1$) Allora ho solo $T.nil \Rightarrow n_x = 0 = 2^0 - 1 \qquad (2^0 \text{ con } 0 = bh(x))$

($h_x > 1$) Consideriamo x radice. x ha due figli, x_1 e x_2.

Sicuramente vale $h_1, h_2 < h$. Per ipotesi induttiva, valgono:

$$n_{x_1} \geq 2^{bh(x_1)} - 1$$
$$n_{x_2} \geq 2^{bh(x_2)} - 1$$

$$\begin{aligned} n_x &= n_{x_1} + n_{x_2} + 1 \\ &\geq 2^{bh(x_1)} + 2^{bh(x_2)} - 1 \\ &\geq 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1 \\ &\qquad (\text{valgono } bh(x_1) \geq bh(x) - 1,\ bh(x_2) \geq bh(x) - 1) \end{aligned}$$

Complessivamente

$$n = n_{root} \geq 2^{bh(root)} - 1$$

Essendo $bh(root) \geq \frac{h}{2}$, posso ottenere

$$\begin{aligned} n_{root} &\geq 2^{bh(root)} - 1 \\ & 2^{\frac{h}{2}} - 1 \\ &\Rightarrow 2^{\frac{h}{2}} \leq n+1 \\ &\frac{h}{2} \leq \log_2(n+1) \Rightarrow h \leq 2\log_2(n+1) \end{aligned}$$

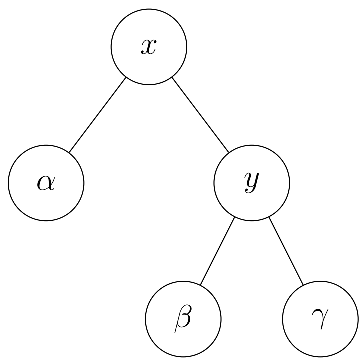
6.2.1 Complessità Algoritmi RB-Trees

`Search, Succ, Min, Pred, Max` hanno complessità $O(h) = O(\log n)$

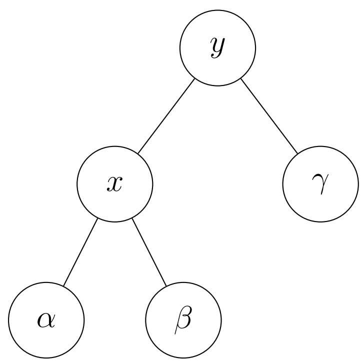
6.2.2 RB-Insert e RB-Delete

A differenza di quelle citate precedentemente, che risultano semplici sia come complessità asintotica che come implementazione, bisogna porre particolare attenzione a queste due procedure: `RB-Insert` e `RB-Delete`.

Per ovviare a ciò, posso utilizzare le **rotazioni**. Consideriamo il seguente albero, in cui x e y sono nodi normali, mentre α, β e γ sono sotto-alberi (il colore dei nodi non ha importanza ai fini della procedura che andremo a vedere)[1]:



Applichiamo `Left(T,x)`, ottenendo:



Osservazione La **visita simmetrica** è identica per i due alberi:

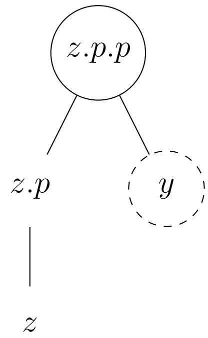
$$\alpha \to x \to \beta \to y \to \gamma$$

[1]Di conseguenza, applicandola a un **RB-Tree**, gli assiomi di validità potrebbero venire violati.

RB-Insert(T, z) Voglio inserire z nell'albero T. L'idea è quella di porre $z.color$ = RED poichè meno insidioso[1].

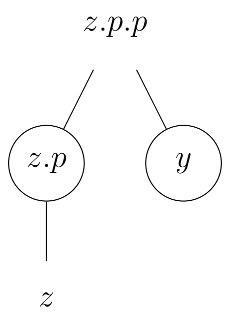
- Se violo (2) $\Rightarrow$ $z.color$ = BLACK;
- Se violo (4):
  - Risolvo localmente;
  - Sposto verso l'alto il problema.

Abbiamo due **macrocasi**. $z.p$ è figlio sinistro, oppure destro. Noi analizzeremo solo il primo: **z.p figlio sinistro**[2].



Abbiamo due possibilità per y:

1. $y.color$ = RED. Inverto il colore di $z.p.p$ con quello dei figli, ottenendo:



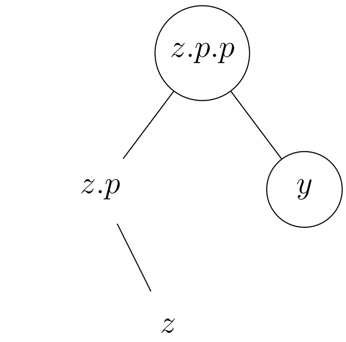
In questo modo, risolviamo localmente e rimandiamo il problema in alto.

[1]Andando a modificare il numero di nodi neri, cambia l'altezza nera, e la cosa è difficile da sistemare.

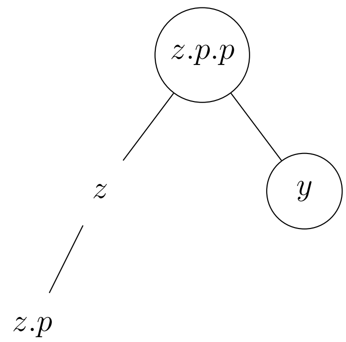
[2]I nodi non cerchiati sono RED, quelli cerchiati sono BLACK, quelli tratteggiati e puntinati possono essere sia RED che BLACK.

2. $y.color$ = BLACK. Possiamo distinguere due sottocasi:

(2.1) z figlio destro.

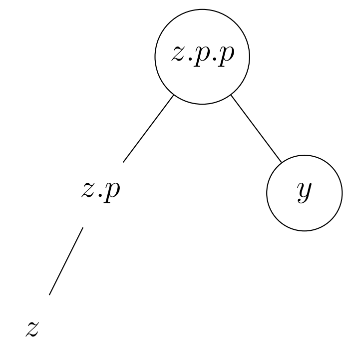


Applico `Left(T,z.p)`, ottenendo:

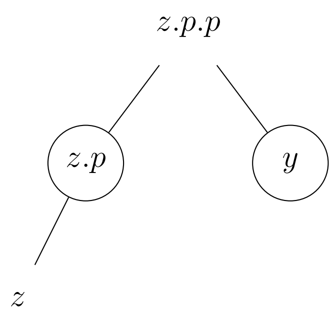


Mi riconduco al caso (2.2).

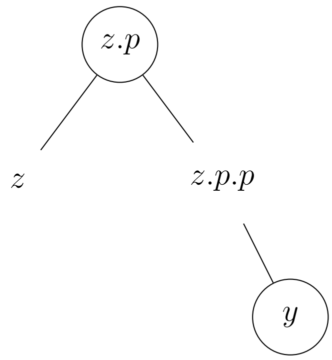
(2.2) z figlio sinistro.



Scambio i colori di $z.p.p$ con $z.p$, ottenendo:



Applico `Right(T,z.p.p)`[1], ottenendo:



RB-INSERT(T, z)

```
Insert(T, z)
z.color = RED
RB-Insert-Fixup(T, z)
```

[1] Analoga di `Left`.

RB-INSERT-FIXUP(T, z)

```
while z.p.color = RED
    if z.p = z.p.p.left // Macrocaso z.p figlio sinistro
        y = z.p.p.right
        if y.color = RED // Caso 1
            z.p.p.color = RED
            z.p.color = BLACK
            y.color = BLACK
            z = z.p.p
        else // Caso 2
            if z = z.p.right // Caso (2.1)
                LEFT(T, z.p)
                z = z.left
            // Caso (2.2)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT(T, z.p.p)
    else
        ... // Macrocaso z.p figlio destro
T.root.color = BLACK
```

Complessità $O(\log n) + \max 2$ rotazioni

RB-Delete(T, z) Distinguiamo 2 casi:

(1) z ha un figlio;

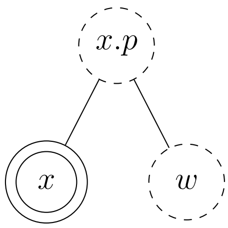
(2) z ha due figli.

Ci comportiamo allo stesso modo della `Delete(T,z)` per un ABR, facendo però un'ulteriore accorgimento:

- se $z.color$ = RED non ho problemi
- se $z.color$ = BLACK violo (5):
    - Risolvo localmente;
    - Sposto verso l'alto il problema.

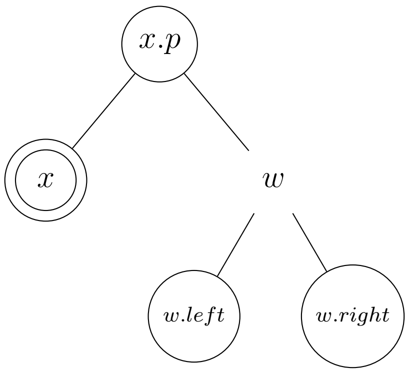
Dunque, in seguito alla `Delete(T,z)`, il nodo x che ha preso il posto di z ne "assorbirà" il colore diventando doppiamente BLACK.

Abbiamo due **macrocasi**. x è figlio sinistro, oppure destro. Noi analizzeremo solo il primo: **x figlio sinistro**.

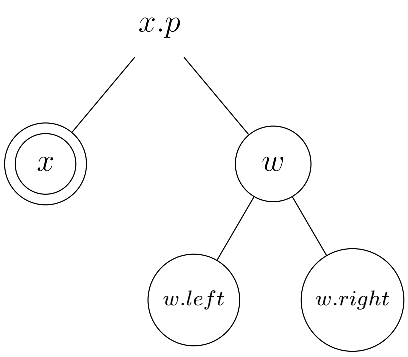


Abbiamo due possibilità per w:

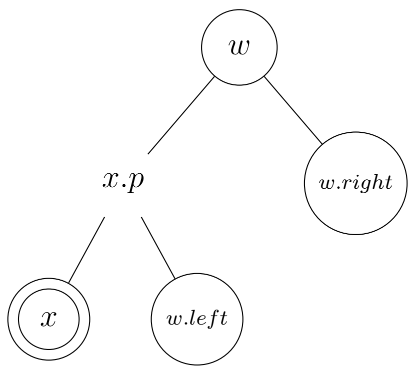
1. $w.color$ = RED

   Scambio i colori di w con $x.p$, ottenendo:

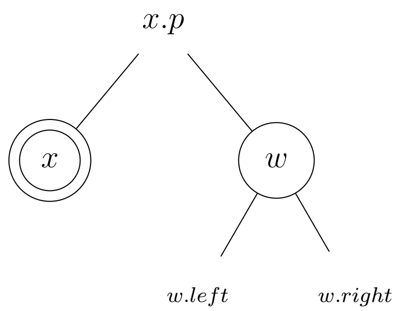
   

   Applico `Left(T,x.p)`, ottenendo:



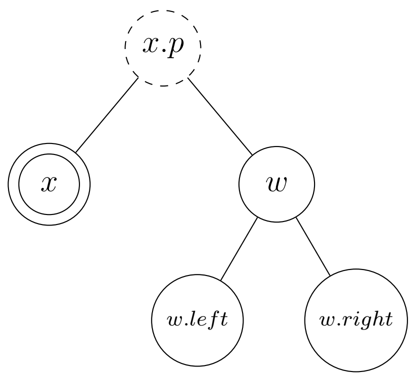
Mi riconduco al caso 2

2. $w.color$ = BLACK

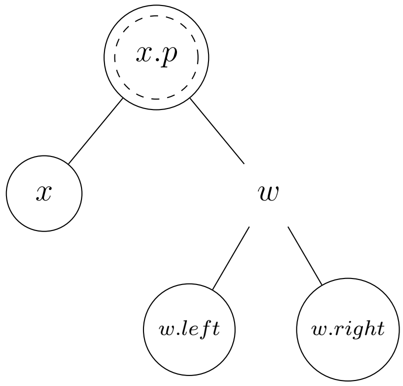


Possiamo distinguere tre sottocasi:

(2.1) $w.left.color$ = BLACK e $w.right.color$ = BLACK

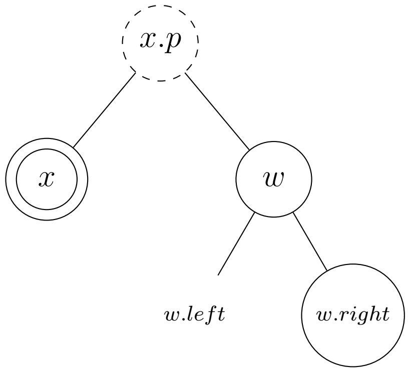


x cede un suo BLACK a $x.p$ e w diventa per forza RED, ottenendo:

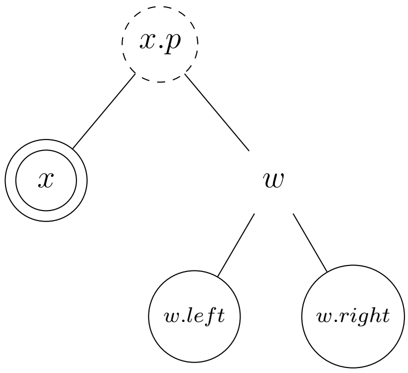


In questo modo, risolviamo localmente e rimandiamo il problema in alto.

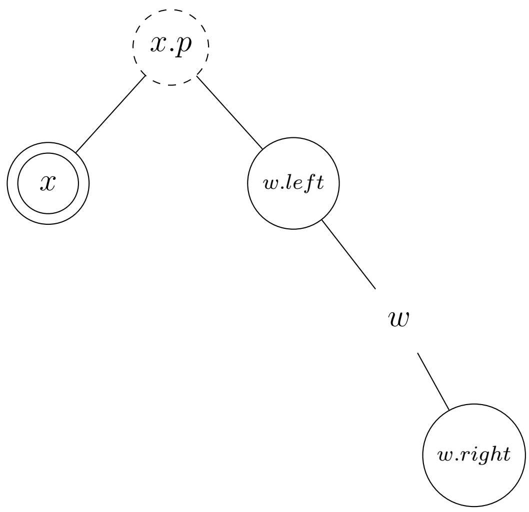
(2.2) $w.right.color =$ BLACK



Scambio i colori di w con $w.left$, ottenendo:

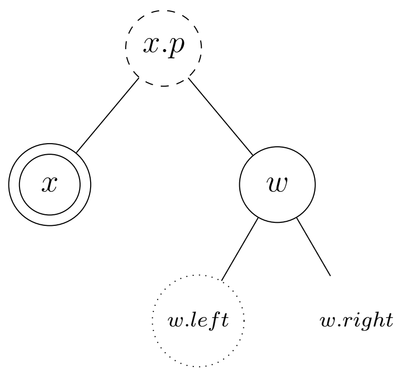


Applico `Right(T,w)`, ottenendo:

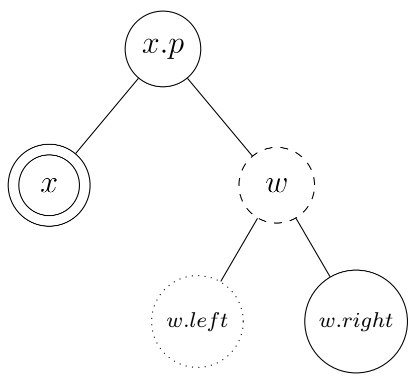


Mi riconduco al caso (2.3)

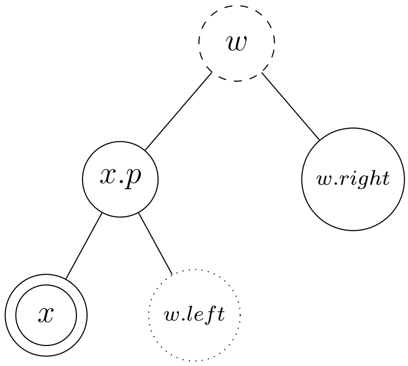
(2.3) $w.right.color =$ RED



Scambio i colori di $x.p$ con w e $w.right$ diventa BLACK, ottenendo:



Applico `Left(T,x.p)`, ottenendo:



Ho risolto.

Complessità $O(\log n)$ + max 3 rotazioni

6.3 Arricchimento di Strutture Dati

Vedremo due esempi, uno per gli RB-Trees, e un altro per gli ABR.

- **Statistiche d'ordine** (6.3.1)
- **Interval Trees** (6.3.3)

6.3.1 Statistiche d'Ordine

Struttura che parte da un RB-Tree. Aggiungo:

- `Select(T,i)` $\equiv$ nodo x che occuperebbe la posizione i nei nodi ordinati per chiave (in una **visita simmetrica**);
- `Rank(T,x)` $\equiv$ posizione i (in una **visita simmetrica**) che occupa il nodo x.

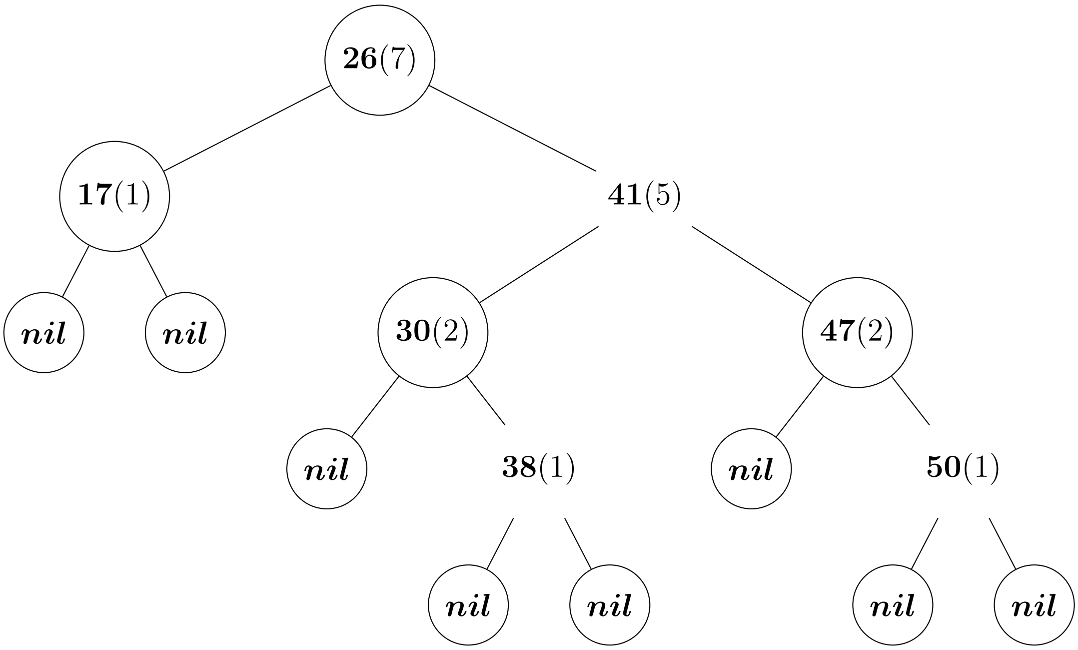
Per implementare queste due procedure, ho bisogno di un nuovo campo dati. Aggiungo il campo

$$x.size = \#\text{nodi radicati nel sottoalbero } T_x$$

Valgono

$$T.nil.size = 0$$
$$x.size = x.left.size + x.right.size + 1$$

Esempio In ogni nodo, tra le parentesi è riportato la *size* di quel nodo. Ricordiamo che i nodi *nil* ($T.nil$) hanno $size = 0$.



Vediamo un'implementazione non efficiente della procedura `Size`.

SIZE(x)
if $x = T.nil$
  $x.size = 0$
else
  $l = $ SIZE($x.left$)
  $r = $ SIZE($x.right$)
  $x.size = l + r + 1$
return $x.size$

Complessità La complessità è $O(n)$, che come preannunciato, non è efficiente. Questo perchè le procedure `Insert/Delete` di un RB-Tree sono nel peggiore dei casi $O(h)$.

Questa procedura, `Select`, restituisce il nodo di posizione i in T_x in una visita simmetrica.

```
Select(x, i)
    // Pre: x ≠ T.nil,  i : 1 ≤ i ≤ x.size
    r = x.left.size + 1
    if i = r
        return x
    else if i < r
        return Select(x.left, i)
    else
        return Select(x.right, i − r)
```

Complessità $O(h) = O(\log n)$

`Rank` restituisce la posizione i che occupa il nodo x.

```
Rank(x)
    r = x.left.size + 1
    y = x
    while y ≠ T.root // idea: r contiene la posizione di x in T_y
        if y.p.right = y
            r = r + y.p.left.size + 1
        y = y.p
    return r
```

Complessità $O(h) = O(\log n)$

Versione aggiornata di `RB-Insert` :

```
RB-Insert(T, z)
  // (1) versione aggiornata di Insert
  z.size = 1 // modifica 1
  x = T.root
  y = T.nil
  while x ≠ T.nil
      x.size = x.size + 1 // modifica 2
      y = x
      if z.key < x.key
          x = x.left
      else
          x = x.right
  z.p = y
  if y = NIL
      T.root = z
  else if z.key < y.key
          y.left = z
      else
          y.right = z
  // (2) RB-Insert-Fixup
  z.color = RED
  RB-Insert-Fixup(T, z)
```

Versione aggiornata di `Left`:

```
Left(T, x)
  x.right = y.left
  x.right.p = x
  y.left = x
  x.p = y
  Transplant(T, x, y)
  y.size = x.size // modifica 1
  x.size = x.left.size + x.right.size + 1 // modifica 2
```

Versione aggiornata di `Delete`:

DELETE(T, z)
    // Identica a quella per gli RB-Trees
    ⋮
    $w = y.p$
    while $w \neq$ NIL
        $w.size = w.size - 1$
        $w = w.p$

Versione aggiornata di `RB-Delete`:

RB-DELETE(T, z)
    DELETE(T,z) // $O(\log n)$
    RB-DELETE-FIXUP(T,z) // $O(\log n)$

6.3.2 Teorema dell'Aumento degli RB-Trees

Def. Sia $x.field$ un campo che si calcola in tempo costante usando $x, x.left, x.right$ ($x.field = F(x, x.left, x.right)$). Allora è possibile modificare `RB-Insert` e `RB-Delete` in modo che mantengano aggiornato il campo $x.field$ con complessità asintotica $O(\log n)$.

6.3.3 Interval Trees

Gli **Interval Trees** sono alberi binari di ricerca con un campo $x.int$, che a sua volta presenta due campi:

- $int.low$, che è anche la chiave;
- $int.high$.

E anche di un campo $x.max$ = max estremo di intervallo per i nodi in T_x, ossia

$$x.max = max \begin{Bmatrix} x.left.max \\ x.right.max \\ x.int.high \end{Bmatrix}$$

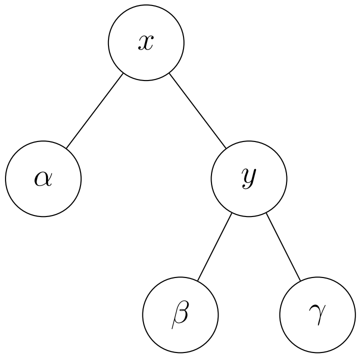
L'idea è quella in cui ogni nodo rappresenti un intervallo.

Vogliamo implementare le seguenti procedure:

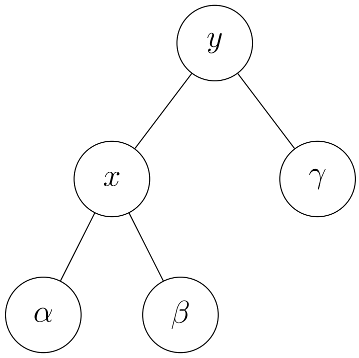
- `Insert(T,x)`
- `Delete(T,x)`
- `ISearch(T,i)` con $i = [low, high]$:

- x tale che $x.int \cap i \neq \varnothing$;
- $T.nil$ se un tale x non c'è.

Rotazioni Prendiamo il seguente albero di esempio.



Applico `Left(T,x)`, ottenendo



Sistemo i massimi. `Left` costa ancora $O(1)$

- $y.max = x.max$
- $x.max = \max\{x.int.high, x.left.max, x.right.max\}$

Vediamo `ISearch`.

ISEARCH(x, i)

if $(x = T.nil)$ **or** $(x.int \cap i \neq \varnothing)$
  return x
else if $(x.left \neq T.nil)$ **and** $(x.left.max \geq i.low)$
  return ISEARCH$(x.left, i)$
else
  return ISEARCH$(x.right, i)$

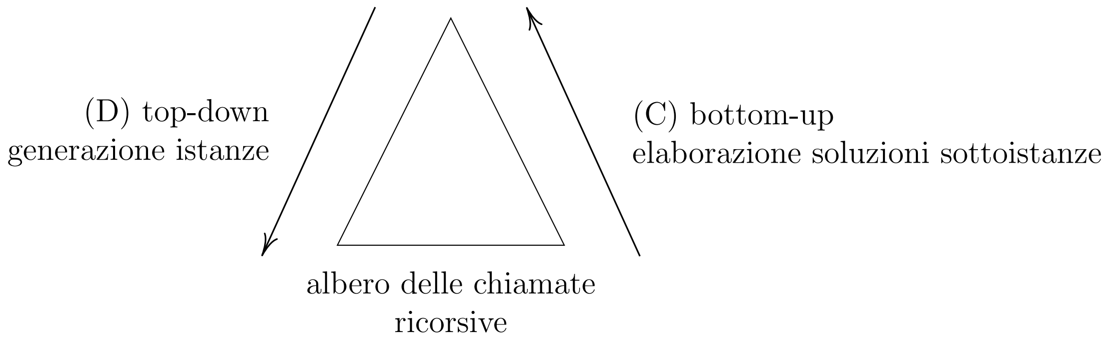
Correttezza

- **Else if**. Consideriamo in $x.left$ un intervallo i'. Abbiamo 2 possibilità.
  (1) $i \cap i' \neq \varnothing$
  (2) $i \cap i' = \varnothing$, ovvero vale $i.high < i'.low$. Questo varrà per ogni nodo dei sotto-alberi, quindi è inutile ispezionare gli antenati di quel sotto-albero.
- **Else**. $\forall$ i' in $x.left \Rightarrow i' \cap i \neq \varnothing$.

Complessità $O(h) = O(\log n)$

7 Programmazione Dinamica

7.1 Critica al Divide & Conquer (D&C)



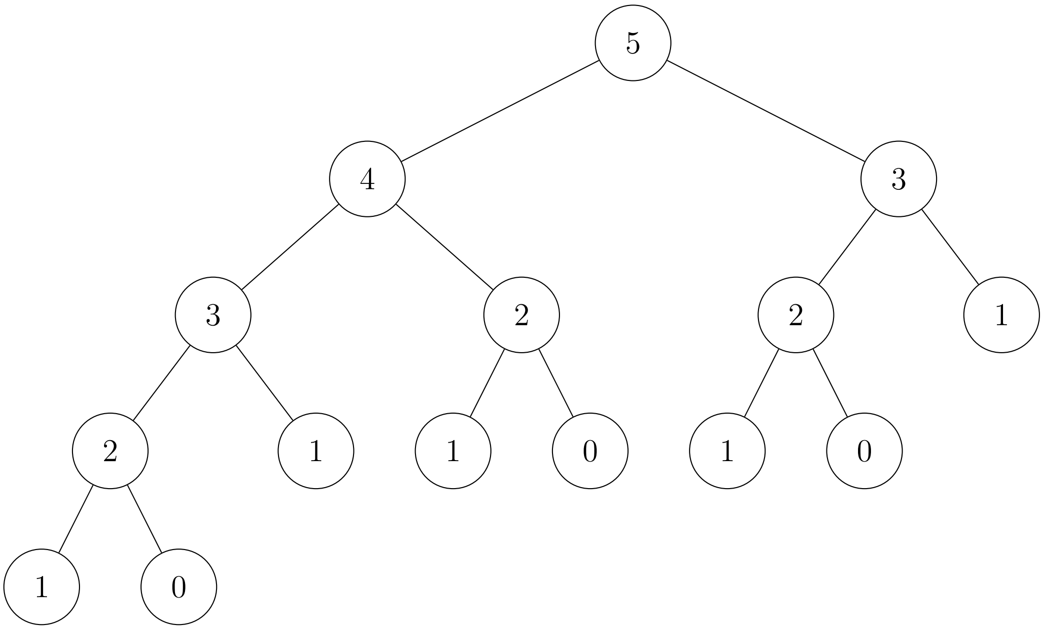
Il processo di soluzione non ha memoria, quindi le soluzioni di sottoistanze vanno ricalcolate.

Esempio Vediamo uno "spreco" usando D&C: la sequenza di Fibonacci.

$$F(n) = \begin{cases} 1 & \text{se } n = 0, 1 \\ F(n-1) + F(n-2) & \text{se } n \geq 2 \end{cases}$$

```
Rec-Fib(n)
if (n = 0) or (n = 1)
    return 1
return Rec-Fib(n − 1) + Rec-Fib(n − 2)
```

Ad esempio, con $n = 5$



Vengono ricalcolate $F(3)$ e $F(2)$.

Complessità

$$T(n) = \begin{cases} 0 & \text{se } n = 0, 1 \qquad \text{(il "return" costa 0)} \\ T(n-1) + T(n-2) + 1 & \text{se } n \geq 2 \qquad \text{(il "+" costa 1)} \end{cases}$$

$$\begin{aligned} T(n) &\geq T(n-1) + T(n-2) + 1 \\ &\geq 2T(n-2) + 1 \\ &\geq 2(2T(n-2-2) + 1) + 1 \\ &= 2^2 T(n-2-2) + 2 + 1 \\ &\geq 2^i T(n-2 \cdot i) + \sum_{j=0}^{i-1} 2^j \end{aligned}$$

$$\begin{aligned} i_0 \rightarrow i &= \left\lfloor \frac{n}{2} \right\rfloor \\ \text{se n è pari:} \quad & 2^{\frac{n}{2}} T(n - 2\frac{n}{2}) = 2^{\frac{n}{2}} T(0) \\ \text{se n è dispari:} \quad & \left\lfloor \frac{n}{2} \right\rfloor = \frac{n-1}{2} \\ & 2^{\frac{n-1}{2}} T(n - 2\frac{n-1}{2}) = 2^{\frac{n-1}{2}} T(1) \end{aligned}$$

Otteniamo

$$T(n) \geq \sum_{j=0}^{\lfloor \frac{n}{2} \rfloor - 1} 2^j = \Theta(2^{\frac{n}{2}})$$

In verità,

$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

Vediamo ora una versione iterativa:

```
It-Fib(n)
if (n = 0) or (n = 1)
    return 1
F[0] = 1
F[1] = 1
for i = 2 to n
    F[i] = F[i − 1] + F[i − 2]
return F[n]
```

Complessità $\Theta(n)$

La programmazione dinamica salta la fase top-down.

7.2 Memoizzazione

È un ibrido tra il D&C e la programmazione dinamica che vuole mantenere la fase top-down pur cercando di ricordare le soluzioni ai sottoproblemi.

Def Un algoritmo **memoizzato** è costituito da due subroutine distinte:

1) **routine di inizializzazione**: risolve direttamente i casi base e inizializza una struttura dati che contiene le soluzioni ai casi base e gli elementi per tutte le sottoistanze da calcolare, inizializzate ad un valore di default

2) **routine ricorsiva**: esegue il codice D&C preceduto da un test sulla struttura dati per verificare se la soluzione è già stata calcolata e memorizzata. Se sì, si ritorna, altrimenti la si calcola ricorsivamente e la si memorizza nella struttura.

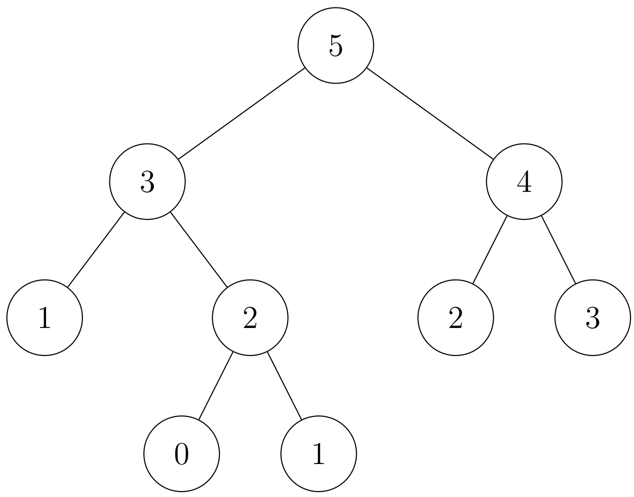
Esempio Riprendiamo l'esempio di prima sulla sequenza di Fibonacci.

```
INIT-FIB(n)
if (n = 0) or (n = 1)
    return 1
F[0] = 1
F[1] = 1
for i = 2 to n
    F[i] = 0
return REC-FIB(n)
```

Complessità $\Theta(n)$

```
REC-FIB(i)
if F[i] = 0
    F[i] = REC-FIB(i − 2) + REC-FIB(i − 1)
return F[i]
```

Riprendiamo l'esempio di prima, con $n = 5$



Questa volta, $F(3)$ e $F(2)$ non vengono ricalcolate.
Abbiamo n foglie e $n - 1$ nodi interni (n parte da 0).

7.3 Problemi di Ottimizzazione

I = insieme delle istanze
S = insieme delle soluzioni
$\Pi \subseteq I \times S$
$\forall\ i \in I,\ S(i) = \{s \in S : (i, s) \in \Pi\}$ = insieme delle soluzioni ammissibili
funzione di costo $c\colon S \to \mathbb{R}$
Determinare, data $i \in I,\ s^* \in S(i) : c(s^*) = \min(/\max)\{c(s) : s \in S(i)\}$

Problema della raggiungibilità su un grafo orientato

$I = \{\langle G = (V, E),\ u,\ v\rangle\ :\ V \subseteq \mathbb{N},\ V \text{ finito},\ E \subseteq V \times V,\ u, v \in V\}$
$S = \{\langle v_1, v_2, \ldots, v_k\rangle\ :\ k \geq 1,\ v_i \in \mathbb{N}\quad \forall\ 1 \leq i \leq k\} \cup \{\varepsilon\}\qquad (\varepsilon = \text{cammino vuoto})$

$$\Big(i = \langle G = (V, E),\ u,\ v\rangle,\ s\Big) \in \Pi \Longleftrightarrow \begin{cases} S = \varepsilon, \exists \text{ un cammino tra } u \text{ e } v \text{ in } G \\ S = \langle v_1, v_2, \ldots, v_k\rangle,\ v_1 = u,\ v_k = v, \\ \qquad (v_i, v_{i+1}) \in E \quad \forall\ 1 \leq i \leq k \end{cases}$$

$c(\langle v_1, v_2, \ldots, v_k\rangle) = k - 1$
$c(\varepsilon) = +\infty$

Caratteristiche Un problema di ottimizzazione, per essere risolto con la programmazione dinamica, deve avere le seguenti caratteristiche:

- struttura ricorsiva;
- esistenza di sottoistanze ripetute;
- spazio di sottoproblemi "piccolo".

Paradigma Generale

1. Caratterizza la struttura di una soluzione ottima s^* in funzione di soluzione ottime $s_1^*, s_2^*, \ldots, s_k^*$ di sottoistanze di taglia inferiore.

2. Determina una relazione di ricorrenza del tipo $c(s^*) = f(c(s_1^*), \ldots, c(s_k^*))$.

3. Calcola $c(s^*)$ impostando il calcolo in maniera bottom-up (oppure memoizzando).

4. Mantiene informazioni strutturali aggiuntive che permettono di ricostruire s^*.

7.4 Problemi su Stringhe

Def Dato un alfabeto finito Σ, una **stringa**

$$X = \langle x_1, x_2, \dots, x_m \rangle, \quad x_i \in \Sigma \quad \forall\, 1 \le i \le m$$

è una concetazione finita di simboli in Σ.

$m = |X| =$ lunghezza di X
$\Sigma^* =$ insieme di tutte le stringhe di lunghezza finita costruibili su Σ
$\varepsilon =$ stringa vuota

Data una stringa X, il **prefisso** di X è

$$X_i = \langle x_1, x_2, \dots, x_i \rangle, \quad 1 \le i \le m$$

Data una stringa X, il **suffisso** di X è

$$X^i = \langle x_i, x_{i+1}, \dots, x_m \rangle, \quad 1 \le i \le m$$

Per convenzione $X_0 = X^{m+1} = \varepsilon$

Def Data una stringa X, la **sottostringa** di X è

$$X_{i \dots j} = \langle x_i, x_{i+1}, \dots, x_j \rangle, \quad 1 \le i \le j \le m$$

Per convenzione $X_{i \dots j} = \varepsilon \quad$ se $i > j$

possibili sottostringhe di una stringa con m caratteri:

$$\underset{\substack{\uparrow \\ i \neq j}}{\binom{m}{2}} + \underset{\substack{\uparrow \\ i = j}}{m} + \underset{\substack{\uparrow \\ \varepsilon}}{1} = \frac{m(m+1)}{2} = \Theta(m^2)$$

Lo spazio delle sottostringhe "non è troppo grande".

Def Data una stringa

$$X = \langle x_1, x_2, \dots, x_m \rangle \in \Sigma^*$$

e

$$Z = \langle z_1, z_2, \dots, z_k \rangle \in \Sigma^*$$

si dice che Z è **sottosequenza** di X se $\exists$ una successione crescente di indici

$$1 \le i_1 \le i_2 \le \dots \le i_k \le m \ : \ z_j = x_{ij} \quad \forall\, 1 \le j \le k$$

Esempio

$X = \langle a, b, c, b, b, d \rangle$
$Z_1 = \langle a, b, c \rangle = X_{1\ldots3}$
$Z_2 = \langle a, c, b \rangle \qquad i_1 = 1, \quad i_2 = 3, \quad i_3 = 4 \text{ o } 5$
$Z_3 = \langle b, b \rangle = X_{4\ldots5} \qquad i_1 = 2, \quad i_2 = 5$

\# possibili sottosequenze di una stringa con m caratteri:

$$\sum_{k=0}^{m} \qquad \underset{\substack{\uparrow \\ \text{stringhe lunghe } k \\ \text{prese da un insieme} \\ \text{di } m \text{ elementi}}}{\binom{m}{k}} \qquad = 2^m$$

7.5 Longest Common Subsequence (LCS)

7.5.1 Problema di Ottimizzazione

Date due stringhe X, Y determina Z tale che:

1) Z è sottosequenza di X e Y;

2) Z è la più lunga tra tutte le sottosequenze comuni.

Esempio

$X = \langle a, b, c, b, b, d \rangle$
$Y = \langle a, d, c, c, b, d \rangle$
$Z = \langle a, c, b, d \rangle$ è una LCS (in questo caso è l'unica)
$\quad i_1 = 1, \quad i_2 = 3, \quad i_3 = 4 \text{ o } 5, \quad i_4 = 6$
$\quad j_1 = 1, \quad j_2 = 3 \text{ o } 4, \quad j_3 = 5, \quad j_4 = 6$

Risolvo il problema:

$|X| = m$
$|Y| = n$

L'approccio "brute force" ha complessità $\Omega(2^m \cdot 2^n)$.

Devo cercare di individuare una proprietà di sottostruttura, cioè la LCS deve "nascondere" al suo interno LCS di qualche stringa più piccola di X e Y.

$X = \langle b, c, f, a \rangle$
$Y = \langle c, f, d, a \rangle$
$Z = LCS(X, Y) = \langle Z', a \rangle \qquad$ con $Z' = LCS(X_3, Y_3)$

$X = \langle X', a \rangle$
$Y = \langle Y', b \rangle$
Z o non termina con a, o non termina con b
$Z = LCS(X', Y)$ o $LCS(X, Y')$
$S = \{LCS(X_i, Y_j) : 0 \leq i \leq m,\ 0 \leq j \leq n\}, \quad |S| = (m\underset{\substack{\uparrow\\ \varepsilon}}{+1})(n\underset{\substack{\uparrow\\ \varepsilon}}{+1})$

7.5.2 Proprietà di Sottostruttura Ottima

Dati i prefissi

$X_i = \langle x_1, x_2, \dots, x_i \rangle$
$Y_i = \langle y_1, y_2, \dots, y_j \rangle$
Sia $Z = \langle z_1, z_2, \dots, z_k \rangle = LCS(X_i, Yj)$

0. caso base: o $i = 0$ o $j = 0$
   $\Rightarrow Z = \varepsilon$

1. $i, k > 0$
   se $x_i = y_j$ allora

   (a) $z_k = x_i (= y_j)$
   (b) $Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$

2. $i, j > 0$
   se $x_i \neq y_j$ allora
   Z è la stringa di lunghezza massima tra $LCS(X_i, Y_{j-1})$ e $LCS(X_{i-1}, Y_j)$

Dimostrazione

0. banale

1. $x_i = y_j$

   $Z = LCS(X_i, Y_j) = \langle z_1, z_2, \dots, z_k \rangle = \langle x_{i_1}, x_{i_2}, \dots, x_{i_k} \rangle = \langle y_{j_1}, y_{j_2}, \dots, y_{j_k} \rangle$
   $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq i, \qquad 1 \leq j_1 \leq j_2 \leq \dots \leq j_k \leq j$

   (a) Ragioniamo per assurdo

   $z_k = x_{i_k} = y_{j_k}$
   $z_k \neq (x_i = y_j)$
   $\Rightarrow i_k < i, \quad j_k < j$
   $Z' = \langle Z, x_i \rangle$
   $1 \leq i_1 \leq i_2 \leq \dots \leq i_k \leq i_{k+1} = i, \qquad 1 \leq j_1 \leq j_2 \leq \dots \leq j_k \leq j_{k+1} = j$

   (b) Devo dimostrare che

   $Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$
   $Z_{k-1} = \langle x_{i_1}, x_{i_2}, \dots, x_{i_{k-1}} \rangle = \langle y_{j_1}, y_{j_2}, \dots, y_{j_{k.1}} \rangle$
   $i_{k-1} \leq i - 1 < i$
   $Z_{k-1} = CS(X_{i-1}, Y_{j-1})$

Ora dimostro che

$Z_{k-1} = LCS(X_{i-1}, Y_{j-1})$

Suppongo per assurdo che

$Z_{k-1} \neq LCS(X_{i-1}, Y_{j-1})$
$\Rightarrow \exists\ Z'$ con $|Z'| \geq k$
$\Rightarrow$ creo $Z'' = \langle \underset{\geq k}{\underset{\uparrow}{Z'}} , \underset{1}{\underset{\uparrow}{x_i}}(= y_j)\rangle \Rightarrow \geq k+1$

2. (come esercizio)

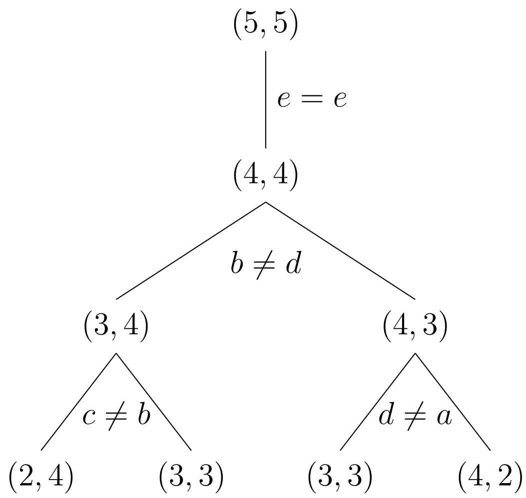
Il D&C ”non funziona” perchè ci sono molti sottoproblemi ripetuti.

Esempio

$X = \langle a, b, c, d, e \rangle$
$Y = \langle b, c, a, b, e \rangle$

Trova $LCS(X, Y)$

Albero delle chiamate:



L'istanza $(3, 3)$ è ripetuta.

Complessità Strategia Ricorsiva Modello di costo: confronto tra caratteri.

$$T(n, m) = \begin{cases} 0 & \text{se } n = 0 \text{ o } m = 0 \\ T(n-1, m) + T(n, m-1) + 1 & \text{se } n, m > 0 \end{cases}$$

Si dimostra che

$$T(n, m) = \Theta\left(\binom{m}{n}\right)$$
$$\binom{m}{2} \geq \binom{m}{2}^n$$

Caso $m = 2n$

$$\binom{m}{2}^n = 2^n$$

7.5.3 Ricorrenza sui Costi

La scrittura della ricorrenza sui costi è il secondo passo per costruire un algoritmo di programmazione dinamica.

Definisco

$l(i,j) = |LCS(X_i, Y_j)|$

$$l(i,j) = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 & \text{(caso 0)} \\ l(i-1, j-1) + 1 & \text{se } i,j > 0 \text{ e } x_i = x_j & \text{(caso 1)} \\ \max\{l(i, j-1), l(i-1, j)\} & \text{se } i,j > 0 \text{ e } x_i \neq x_j & \text{(caso 2)} \end{cases}$$

Alla fine ci interessa calcolare $l(m,n)$.

Per calcolare $l(i,j)$ mi possono servire tre valori:

$$L = \begin{bmatrix} & & \\ (i{-}1, j{-}1) & & (i{-}1, j) \\ & \nwarrow & \uparrow \\ (i, j{-}1) & \leftarrow & (i,j) \\ & & \end{bmatrix}$$

Scansione "row-major": riempio la tabella per righe, da sinistra a destra.

Informazione addizionale per costruire la sequenza (vera e propria):

$$b(i,j) = \begin{cases} '\nwarrow' & \text{se } x_i = y_j \\ '\leftarrow' & \text{se } x_i \neq x_j \text{ e } max = LCS(i, j-1) \\ '\uparrow' & \text{se } x_i \neq y_j \text{ e } max = LCS(i-1, j) \end{cases}$$

Pseudocodice

```
LCS(X, Y)
m = X.[length]
n = Y.length
for i = 0 to m
      L[i, 0] = 0
for j = 0 to n
      L[0, j] = 0
for i = 1 to m
      for j = 1 to n
            if x_i = y_j
                  L[i, j] = L[i − 1, j − 1] + 1
                  B[i, j] = '↖'
            else if L[i − 1, j] ≥ L[i, j − 1]
                  L[i, j] = L[i − 1, j]
                  B[i, j] = '↑'
            else
                  L[i, j] = L[i, j − 1]
                  B[i, j] = '←'
return (L[m, n], B)
```

Complessità

$T(m, n) = \Theta(m \cdot n)$
Caso $m = n \Rightarrow T(m, n) = \Theta(n^2)$

Procedura per stampare la LCS:

```
PRINT-LCS(B, X, i, j)
if i = 0 or j = 0
      return ε
if B[i, j] = '↖'
      PRINT-LCS(B, X, i − 1, j − 1)
      PRINT(x_i)
else if B[i, j] = '←'
      PRINT-LCS(B, X, i, j − 1)
else // B[i, j] = '↑'
      PRINT-LCS(B, X, i − 1, j)
```

Complessità $\Theta(m) = \Theta(|LCS|)$

Esercizio

$X = \langle b, d, c, d \rangle$
$Y = \langle a, b, c, b, d \rangle$

Restituisci $LCS(X, Y)$ e $|LCS(X, Y)|$

$$L = \begin{array}{c} \\ \\ b \\ d \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccccc} & a & b & c & b & d \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 2 \\ 0 & 0 & 1 & \mathbf{2} & 2 & 2 \\ 0 & 0 & 1 & 2 & 2 & \mathbf{3} \end{bmatrix} \end{array} \qquad B = \begin{array}{c} \\ \\ b \\ d \\ c \\ d \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & b & d \end{array} \\ \begin{bmatrix} & & & & \\ \uparrow & \nwarrow & \leftarrow & \nwarrow & \leftarrow \\ \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow \\ \uparrow & \uparrow & \nwarrow & \leftarrow & \uparrow \\ \uparrow & \uparrow & \uparrow & \uparrow & \nwarrow \end{bmatrix} \end{array}$$

$LCS(X, Y) = \langle b, c, d \rangle \qquad |LCS(X, Y)| = 3$

Pseudocodice Memoizzato

```
INIT-LCS(X, Y)
m = X.length
n = Y.length
if (m = 0) or (n = 0)
     return 0
for i = 0 to m
     L[i, 0] = 0
for j = 0 to n
     L[0, j] = 0
for i = 1 to m
     for i = 1 to n
          L[i, j] = −1
return R-LCS(X, Y, m, n)
```

```
R-LCS(X, Y, i, j)
if L[i, j] = −1
     if x_i = y_j
          L[i, j] = R-LCS(X, Y, i − 1, j − 1)
     else if R-LCS(X, Y, i − 1, j) ≥ R-LCS(X, Y, i, j − 1)
          L[i, j] = L[i − 1, j]
     else
          L[i, j] = L[i, j − 1]
return L[i, j]
```

Complessità $O(m \cdot n)$

Osservazione Se $x_i = y_j$ sempre, invoco `R-LCS(X,Y,i-1,j-1)` ma non invoco mai `R-LCS(X,Y,i-1,j)` o `R-LCS(X,Y,i,j-1)`.

Ad esempio

$X = \langle a, a, b, b, c \rangle$
$Y = \langle b, b, c \rangle$

Albero delle chiamate:

$$
\begin{array}{l}
(5,3) \\
\quad \big| \; c = c \\
(4,2) \\
\quad \big| \; b = b \\
(3,1) \\
\quad \big| \; b = b \\
(2,0)
\end{array}
$$

In generale, se Y è suffisso di $n \leq m$ caratteri di X, la complessità di `R-LCS` nel caso migliore è:

$$T_{R-LCS}(m,n) = n$$

Inoltre,

$$\Omega_{LCS}(m+n) \approxeq \Omega(n)$$
$$O_{LCS,R-LCS}(m \cdot n) \approxeq O(n^2)$$

Spazio

$$S_{LCS}(m,n) = \Theta(m,n)$$

Tuttavia, posso migliorarlo a

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

poichè mi bastano due righe della tabella in memoria ad ogni istante, quindi due vettori lunghi n.

Inoltre, se $m \ll n$, posso fare un'ulteriore ottimizzazione utilizzando la tecnica "column-major", cioè scansione per colonne, con due vettori lunghi m.

7.6 Longest Increasing Subsequence (LIS)

Def Dato un alfabeto Σ totalmente ordinato ($\forall\ a, b \in \Sigma \quad a < b$ o $a = b$ o $a > b$) e dato $X = \langle x_1, x_2, \ldots, x_n \rangle$, si dice che $Z = \langle z_1, z_2, \ldots, z_k \rangle$ è **sottosequenza crescente** di X ($Z = IS(X)$).

7.6.1 Problema di Ottimizzazione

Determinare la **più lunga sottosequenza crescente** di X ($Z = LIS(X)$)

Esempio

$X = \langle 5, 2, 4, 3, 7, 8 \rangle$
$Z = LIS(X) = \langle 2, 3, 7, 8 \rangle$
$Z' = LIS(X) = \langle 2, 4, 7, 8 \rangle$

Tentativo Data X, calcolo $LIS(X_i)\ \forall\ 0 \le i \le n$
$Z = \langle z_1, z_2, \ldots, z_k \rangle = LIS(X_i)$

- caso fortunato: $z_k < X_{i+1}$
- caso sfortunato: $z_k \ge X_{i+1}$

Def $Z = \overline{LIS}(X_i)$ è la più lunga tra le $IS(X_i)$ con $Z = \langle z_1, z_2, \ldots, z_k \rangle = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$ con $i_k = i$.

Esempio

$X = \langle 8, 2, 5, 1, 3 \rangle$
$LIS(X_4) = \langle 2, 5 \rangle$
$\overline{LIS}(X_4) = \langle 1 \rangle$

In generale, LIS e $\overline{LIS}$ sono problemi molto diversi.

Osservazione $|LIS(X)| = \max\limits_{1 \le i \le n} \{ |\overline{LIS}(X_i)| \}$

Dimostrazione

- $|LIS(X)| \leq \max_{1 \leq i \leq n}\{|\overline{LIS}(X_i)|\}$

  $Z = LIS(X) = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$
  $Z = \overline{LIS}(X_{i_k})$
  $\Rightarrow |Z| \leq \max_{1 \leq i \leq n}\{|\overline{LIS}(X_i)|\}$

- $|LIS(X)| \geq \max_{1 \leq i \leq n}\{|\overline{LIS}(X_i)|\}$
  Si dimostra analogamente al punto precedente.

7.6.2 Proprietà di Sottostruttura Ottima

1. caso base: $\overline{LIS}(X_1) = \langle x_1 \rangle$ $(= LIS(X_1))$
2. $i > 1$
   (a) $\forall\ j : 1 \leq j \leq i \quad x_j \geq x_i$

   $\overline{LIS}(X_i) = \langle x_i \rangle$ $(= LIS(X_i))$

   (b) $\exists\ \overline{j} : 1 \leq n\overline{j} \leq i, \quad x_{\overline{j}} < x_i$

   $|\overline{LIS}(X_i)| \geq 2$
   $\overline{LIS}(X_i) = \langle z_1, z_2, \ldots, z_k \rangle = \langle Z_{k-1}, x_i \rangle$ con $Z_{k-1} : |Z_{k-1}| = \max_{1 \leq j < i}\{\overline{LIS}(X_j) : x_j < x_i\}$

Dimostrazione

1. banale
2. $i > 1$
   (a) $\forall\ j < i \quad x_j < x_i$

   $\langle x_i \rangle = IS(X_i)$ e chiaramente non può essere $|Z| > 1$

   Suppongo per assurdo che

   $|Z| = |\overline{LIS}(X_i)| > 1$
   $\Rightarrow Z = \langle z_1, z_2, \ldots, z_k \rangle, \quad k > 1$
   $Z = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_k} \rangle$
   $i_k = i$
   $\Rightarrow i_{k-1} < i_k = i$

Assurdo perchè allora avrei

$x_{i_{k-1}} \geq x_{i_k}$

che contraddice l'ipotesi che Z è una sequenza crescente!

(b) Si dimostra analogamente al sottocaso precedente.

7.6.3 Ricorrenza sui Costi

Definisco

$$l(i) = \left|\overline{LIS}(X_i)\right|$$

$$l(i) = \begin{cases} 1 & \text{se } i = 1 \\ 1 + \max\limits_{1 \leq j < i}\{l(j) : x_j < x_i\} & \text{se } i > 1 \end{cases}$$

Per costruire la sottosequenza

$$\overline{LIS}(X_i) = \begin{cases} \langle x_i \rangle & & (1) \\ \langle \overline{LIS}(X_{\overline{j}}), x_i \rangle & \text{con } 1 \leq \overline{j} < i & (2) \end{cases}$$

l'informazione addizionale è:

- $prev(i) = \begin{cases} 0 & \text{nel caso (1)} \\ \overline{j} & \text{nel caso (2)} \end{cases}$
- $len = \max\limits_{1 \leq i \leq n}\{l(i)\}$
- $end = i$ se $\overline{LIS}(X_i) = LIS(X)$
  (mantiene l'indice dell'ultimo carattere della LIS)

Pseudocodice bottom-up

```
LIS(X)
L[1] = 1
len = 1
end = 1
prev[1] = 0
for i = 2 to n
    L[i] = 1
    prev[i] = 0
    for j = 1 to i − 1
        if x_j < x_i
            if L[i] < 1 + L[j]
                L[i] = 1 + L[j]
                prev[i] = j
    if len < L[i]
        len = L[i]
        end = i
return (len, prev, end)
```

Procedura per stampare la LIS:

```
R-Print(X, prev, i)
if prev[i] ≠ 0
    R-Print(X, prev, prev[i])
Print(x_i)
```

Complessità $\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2} = \Theta(n^2)$

7.7 Completamento a Palindromo (CP)

Def Una stringa $Z = \langle z_1, z_2, \ldots, z_m \rangle$ è **palindroma** se $z_{1+h} = z_{m-h} \quad \forall\, 0 \leq h \leq m-1$.

Problema Data $X = \langle x_1, x_2, \ldots, x_n \rangle$, un **complemento palindromo** di X è una stringa $Z = CP(X) = \langle z_1, z_2, \ldots, z_m \rangle$ con $m \geq n$ tale che:

1) Z è palindroma;

2) X è sottosequenza di Z

(cioè, Z è un palindromo che contiene X come sottosequenza).

Esempio

$X = \langle a, c, b \rangle$
$Z = \langle a, c, b, b, c, a \rangle$
$Z' = \langle a, b, c, c, b, a \rangle$

Osservazione $|X| = n \leq |Z| \leq 2n = 2|X|$

7.7.1 Problema di Ottimizzazione

Determinare $Z = CP(X)$ di lunghezza minima.
Spazio dei sottoproblemi: $X_{i\ldots j}$ (cioè lo spazio delle sottostringhe di X).
Determinare un algoritmo bottom-up quadratico nella lunghezza di X.
$\forall\, i, j : 1 \leq i \leq j \leq n$, determinare il minimo $Z = CP(X_{i\ldots j})$.

7.7.2 Proprietà di Sottostruttura Ottima

Casi base:

1. $i = j$ (1 carattere)

   $X = \langle x_i \rangle$
   $CP(X_{i\ldots j}) = X_{i\ldots j}$
   $|CP(X_{i\ldots j})| = |X_{i\ldots j}|$

2. $j = i + 1$ (2 caratteri)

   $X_{i\ldots j} = \langle x_i, x_{i+1} \rangle$

(a) $x_i = X_{i+1}$

$CP(X_{i\ldots i+1}) = X_{i\ldots i+1}$

$|CP(X_{i\ldots i+1})| = 2$

(b) $x_i \neq x_{i+1}$

Un possibile $CP(X_{i\ldots i+1})$ è $\langle x_i, x_{i+1}, x_i\rangle$

$|CP(X_{i\ldots i+1})| = 3$

Caso generale:

$X_{i\ldots j} = \langle x_i, x_{i+1}, \ldots, x_j\rangle$

(a) $x_i = x_j$

$Z = CP(X_{i\ldots j})$

$z_1 = z_k = x_i \ (= x_j)$

$Z_{2\ldots k-1} = CP(X_{i+1\ldots j-1})$

(b) $x_i \neq x_j$ Ci sono due possibili soluzioni:

i. $z_1 = z_k = x_i$ e $Z_{2\ldots k-1} = CP(X_{i+1\ldots j})$

ii. $z_1 = z_k = x_j$ e $Z_{2\ldots k-1} = CP(X_{i\ldots j-1})$

Dimostrazione

(b).i. Suppongo per assurdo che

$z_1 = z_k \neq x_i$

$\Rightarrow X_{i\ldots j}$ è sottosequenza di

$Z_{2\ldots k-1}$ che è palindroma e più corta di 2 rispetto a Z

Assurdo perchè Z è la più corta!

(b).ii. Si dimostra analogamente al sottocaso precedente.

7.7.3 Ricorrenza sulle Lunghezze

Definisco

$l(i,j) = |CP(X_{i\ldots j})|$

$$l(i,j) = \begin{cases} 1 & \text{se } i = j \\ 2 & \text{se } j = i+1 \text{ e } x_i = x_j \\ 3 & \text{se } j = i+1 \text{ e } x_i \neq x_j \\ 2 + l(i+1, j-1) & \text{se } j > i+1 \text{ e } x_i = x_j \\ 2 + \min\{l(i+1,j), l(i,j-1)\} & \text{se } j > i+1 \text{ e } x_i \neq x_j \end{cases}$$

Per calcolare $l(i,j)$ mi possono servire tre valori:

$$L = \left[\begin{array}{ccc} & (i, j{-}1) & \leftarrow \; (i,j) \\ & \swarrow & \downarrow \\ (i{+}1, j{-}1) & & (i{+}1, j) \end{array}\right]$$

Riempio la tabella per diagonali, dall'alto verso il basso e da sinistra verso destra.

```
SPC(X)
n = X.length
for i = 1 to n − 1
    L[i, j] = 1
    if x_i = x_{i+1}
        L[i, i + 1] = 2
    else
        L[i, i + 1] = 3
L[n, n] = 1
for l = 3 to n // scansione diagonale con l indice della diagonale
    for i = 1 to n − l + 1
        j = i + l − 1
        if x_i = x_j
            L[i, j] = 2 + L[i + 1, j − 1]
        else
            L[i, j] = 2 + min{L[i + 1, j], L[i, j − 1]}
return L[1, n]
```

Complessità

$$\sum_{i=1}^{n-1} 1 + \sum_{l=3}^{n} \sum_{i=1}^{n-l+1} 1 = \sum_{i=1}^{n-1} 1 + \sum_{l=3}^{n} (n-l+1) = \sum_{i=1}^{n-1} 1 + \sum_{i=2}^{n-1} (n-i) =$$

$$= \sum_{i=1}^{n-1} (n-i) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2)$$

8 Algoritmi Greedy

Un **algoritmo greedy** ("incauto", "irruento") è un tipo di algoritmo per problemi di ottimizzazione che cerca di risolvere il problema in modo diretto, scegliendo la soluzione al sottoproblema più piccolo che al momento sembra la migliore.

Caratteristiche:

- semplice, con complessità di solito linerare;
- difficoltà di analisi;
- limitato campo di applicazione.

8.1 Critica alla Programmazione Dinamica

La programmazione dinamica è "prudente", nel senso che risolve tutti i sottoproblemi di una certa taglia.

8.2 Problema di Selezione di Attività Compatibili

Abbiamo:

- risorsa condivisa (e.g. aula);
- insieme di attività $S = \{a_i : 1 \leq i \leq n\}$

  $a_i = [s_i, f_i), \quad 0 \leq s_i \leq f_i \quad (s_i = \text{tempo di inizio}, f_i = \text{tempo di fine})$

Def Diciamo che a_i e a_j sono **compatibili** sse

$$[s_i, f_i) \cap [s_j, f_j) = \emptyset$$

Equivalentemente

$$f_i \leq s_j \text{ oppure } f_j \leq s_i$$

Esempio

Aula Lum250

$a_1 = [10.30, 12.00)$

$a_2 = [11.30, 13.00)$

$a_3 = [12.30, 14.00)$

a_1 e a_2 non sono compatibili

a_1 e a_3 sono compatibili

a_2 e a_3 non sono compatibili

8.2.1 Problema di Ottimizzazione

Determinare un sottoinsieme di massima cardinalità di attività mutuamente compatibili (cioè compatibili a coppie).

Ulteriore assunzione:

$$0 < f_1 \leq f_2 \leq \cdots \leq f_n$$

Voglio applicare la programmazione dinamica:

determinare i sottoproblemi

$$S_{ij} = \{a_k : f_i \leq s_k < f_k \leq s_j\}$$

Osservazione

1. $i > j \Rightarrow S_{ij} = \emptyset$

   perchè $f_i \leq s_k < f_k \leq s_j$
   ma $f_i \geq f_j$

2. S_{ij} non contiene tutte le attività di indice k con $i < k < j$

3. $|S_{ij}| \leq j - 1 - i, \quad 1 \leq i \leq n$

   Definisco due attività fittizie:

   $a_0 \to f_0 = 0$
   $a_{n+1} \to s_{n+1} = +\infty$
   $S = S_{0\,n+1}$

8.2.2 Proprietà di Sottostruttura Ottima

Sia A^*_{ij} un sottoinsieme di attività compatibili di S_{ij} di cardinalità massima.

Caso base: $S_{ij} = \emptyset \Rightarrow A_{ij} = \emptyset$

Caso generale: $Sij \neq \emptyset$

1. $A^*_{ij} \neq \emptyset$
2. se $a_k \in A^*_{ij}$ allora $A^*_{ij} = A^*_{ik} \cup A^*_{kj}$
   dove A^*_{ik} e A^*_{kj} sono soluzioni ottime per S_{ik} e S_{kj}

Dimostrazione

Caso base: banale

Caso generale: $S_{ij} \neq \emptyset$

1. Un'attività è sempre compatibile con sè stessa $\Rightarrow \left|A^*_{ij}\right| \geq 1$
2. Dimostreremo che una qualsiasi attività $\in A^*_{ij}$ e $\neq a_k$ si trova o in S_{ik} o in S_{kj}

   Ho a_k, una qualsiasi attività $\in A^*_{ij}$ deve essere compatibile con a_k

   Prendiamo $a_l \in A^*_{ij},$ con $l \neq k$
   Deve valere
   $f_i \leq s_l < f_l \leq s_k \Rightarrow a_l \in S_{ik}$
   oppure
   $s_k < f_k \leq s_l < f_l \leq s_j \Rightarrow a_l \in S_{kj}$
   Quindi $A^*_{ij} = \overline{A}_{ik} \cup \{a_k\} \cup \overline{A}_{kj}$
   dove $\overline{A}_{ij}$ = tutte le attività compatibili in S_{ij}

   Ora dimostriamo che gli $\overline{A}$ sono anche A^*

   Per assurdo suppongo che $\overline{A}_{ik}$ non sia l'unico insieme di attività compatibili massimale in S_{ik}

   $$A^*_{ij} = \underset{\substack{\uparrow \\ A^*_{ik}}}{\cancel{\overline{A}_{ik}}} \cup \{a_k\} \cup \overline{A} \ \rightarrow \ \text{soluzione con cardinalità } > \left|A^*_{ij}\right|$$

8.2.3 Ricorrenza sui Costi

Definisco

$$c(i,j) = \left|A^*_{ij}\right|$$

$$c(i,j) = \begin{cases} 0 & \text{se } S_{ij} = \emptyset \\ 1 + \max\limits_{a_k \in S_{ij}} \{c(i,k) + c(k,j)\} & \text{se } S_{ij} \neq \emptyset \end{cases}$$

L'algoritmo bottom-up ha complessità $\Theta(n^3)$

Abbiamo visto che $A^*_{ij} = A^*_{ik} \cup \{a_k\} \cup A^*_{kj}$ ma non so chi è k.
Se conoscessi k potrei scrivere $1 + c(i,k) + c(k,j)$.

Teorema Sia a_m l'attività tale che

$$f_m = \min\{f_k : a_k \in S_{ij}\}$$

Se $S_{ij} \neq \emptyset$ allora

1. $\exists\ A^*_{ij}$ soluzione ottima tale che $a_m \in A^*_{ij}$
2. il sottoproblema $S_{im} = \emptyset$

Dimostrazione

1. Si consideri una soluzione ottima $\overline{A}_{ij}$ a S_{ij}

   Se $a_k = a_m \Rightarrow$ ho finito

   Altrimenti, $a_k \neq a_m \Rightarrow f_m \leq f_k \Rightarrow$ posso togliere a_k da $\overline{A}_{ij}$
2. banale

Strategia

1) Scegliere a_m
2) Risolvere S_{mj} (perchè $A^*_{ij} = \{a_m\} \cup A^*_{mj}$)

A noi interessa risolvere $S_{0\,n+1}$

Osservazione Devo risolvere solo problemi del tipo $S_{m\,n+1}$
$\Rightarrow$ lo spazio dei sottoproblemi è linerare (perchè il 2° indice dei sottoproblemi è fisso).

Codice ad alto livello

OPT($S_{i\,n+1}$)
if $S_{i\,n+1} \neq \emptyset$
    $a_m = f_m = \min\{f_k : a_k \in S_{i\,n+1}\}$ // molto grande
else
    return $\emptyset$

Pseudocodice

```
REC-SEL(S, f, i)
n = S.length
m = i + 1
while (m ≤ n) and (s_m < f_i)
    m = m + 1
if m ≤ n
    return {a_m} ∪ REC-SEL(S, f, m)
else
    return ∅
```

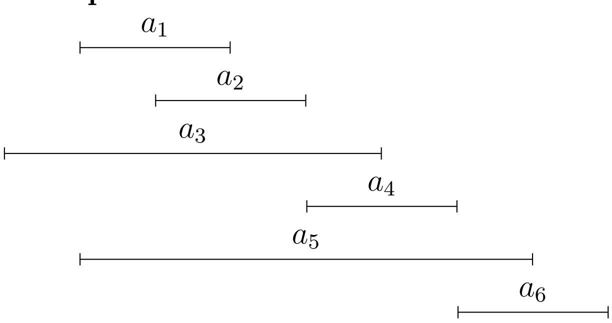
Complessità Modello di costo: confronti tra elementi s e f.

La complessità è $\Theta(n)$ perchè ogni attività viene coinvolta in un confronto una sola volta.

Versione iterativa:

```
GREEDY-SEL(S, f)
n = S.length
A = {a_1}
last = 1 // indice dell'ultima attività selezionata
for m = 2 to n
    if s_m ≥ f_last
        A = A ∪ {a_m}
        last = m
return A
```

Esempio



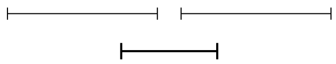
$A = a_1$	$A = a_1, a_4$	$A = a_1, a_4, a_6$
$last = 1$	$last = 4$	$last = 6$

8.2.4 Scelte Greedy Alternative

Oltre alla scelta greedy vista precedentemente, ne esistono altre:

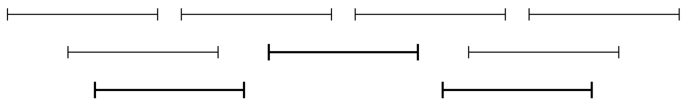
- Scegli l'attività di durata inferiore → non è ottima.

  Controesempio:

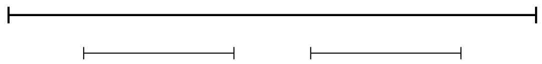
- Scegli l'attività col minor numero di sovrapposizioni → non è ottima.

  Controesempio:

- Scegli l'attività che inizia per prima → non è ottima.

  Controesempio:

- Scegli l'attività che inizia per ultima → è ottima.

8.3 Compressione dei Dati: Codici di Huffman

La compressione può essere di due tipi:

- lossy, cioè con perdita di informazione (e.g. immagine, video);
- lossless, cioè senza perdita di informazione (e.g. testo).

Esempio File di caratteri con frequenze associate:

	a	b	c	d	e	f
frequenza (%)	45	13	12	16	9	5

La codifica ASCII usa 8 bit per carattere.
File con 100K caratteri → 800 Kbit

Definisco una codifica che usa 3 bit per carattere:

	a	b	c	d	e	f
codeword	000	001	010	011	100	101

File con 100K caratteri → 300 Kbit (risparmio più del 50%)
+ spazio per una tabella di conversione/decodifica

0	'a'
1	'b'
2	'c'
3	'd'
4	'e'
5	'f'

Nella posizione i si trova il carattere la cui codifica è i stesso (in binario).

Def

C = alfabeto dei simboli presenti nel file
funzione di encoding $e\colon C \to 0, 1^*$

Proprietà che deve avere e:

- invertibile → iniettiva: se $a, b \in C, a \neq b \Rightarrow e(a) \neq e(b)$;
- ammettere algoritmi efficienti: e, e^{-1}.

Una funzione di encoding che associa ad ogni carattere di C una stringa di 0 e 1 della stessa lunghezza si chiama **fixed length encoding**.

Finora ho ignorato la frequenza dei caratteri.

Idea Associare a caratteri più frequenti codeword più corte e, di conseguenza, a caratteri meno frequenti codeword più lunghe.

	a	b	c	d	e	f
frequenza (%)	45	13	12	16	9	5
codeword	0	00	01	1	100	101

Problema $e(aa) = e(b)$
Non sono come decodificare in modo univoco perchè esistono delle codeword che sono prefissi di altre codeword.

La codifica che sto cercando deve:

- avere lunghezza variabile della codeword;
- affinchè si possa avere l'invertibilità, il codice deve essere libero da prefissi (**codice prefisso**[1]), cioè $\nexists\ a, b \in C : e(a)$ è un prefisso di $e(b)$.

Soluzione

	a	b	c	d	e	f
frequenza (%)	45	13	12	16	9	5
codeword	0	101	100	111	1101	1100

Questo è un codice a lunghezza variabile libero da prefissi ed è ottimo tra tutti i codici che associano una codeword ad ogni singolo carattere.

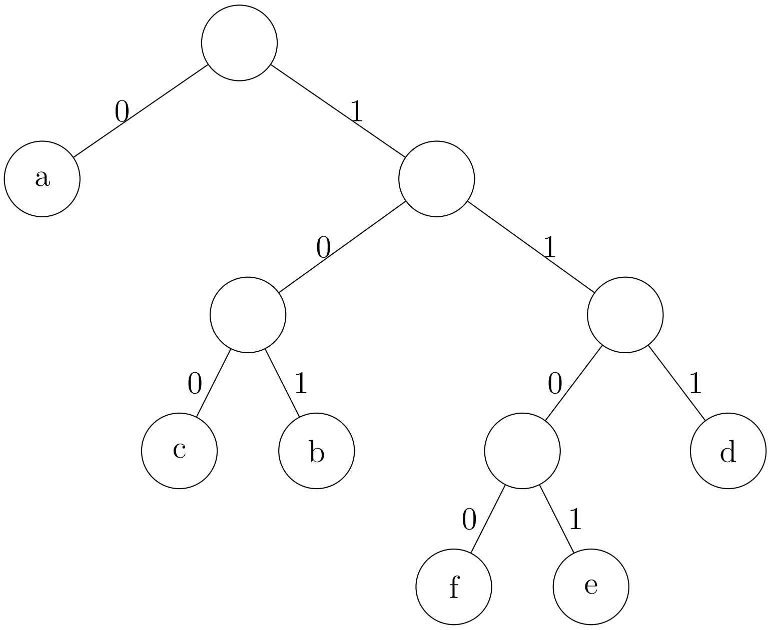
Spazio occupato da questo encoding:

$$100\text{K}\left(\frac{45}{100}\cdot 1+\frac{13}{100}\cdot 3+\frac{12}{100}\cdot 3+\frac{16}{100}\cdot 3+\frac{9}{100}\cdot 4+\frac{5}{100}\cdot 4\right) = 224 \text{ Kbit}$$

300 Kbit $\rightarrow$ risparmio $\approx 25\%$

Serve memorizzare dell'informazione aggiuntiva per la decodifica:

[1] Attenzione: con il termine "codice prefisso" si intende un codice senza prefissi.



Stringa codificata: 110001001101
Stringa decodificata: face

Un qualsiasi codice binario può essere rappresentato in modo compatto con un albero binario.

Un codice prefisso è associato a un **albero di codifica** T in cui i caratteri da codificare appaiono tutti alle foglie.

$\forall\ c \in C$ ho $f(c)$ frequenza

Definisco

$$d_T(c) = \text{profondità di c in T}$$

$d_T(\text{'a'}) = 1$

$d_T(\text{'b'}) = d_T(\text{'c'}) = d_T(\text{'d'}) = 3$

$d_T(\text{'e'}) = d_T(\text{'f'}) = 4$

La profondità corrisponde alla lunghezza della codeword associata.

Funzione di costo:

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

La dimensione del file compresso è

$$|F_c| = \frac{B(T) \cdot |F|}{100}$$

dove

$$|F| = \text{taglia del file compresso (in \# di caratteri)}$$

$B(T)$, a meno di una costante, è il fattore di compressione $\min B(T)$.

Proprietà Un albero ottimo è sempre pieno (cioè i nodi interni hanno due figli).

Spazio di problemi: spazio di alberi pieni con n foglie ($n = |C|$)

Idea Prendo due nodi con frequenza minore, gli unisco in un nodo che avrà come frequenza la somma delle due, ripeto $n-1$ volte.
In questo modo, i nodi con la frequenza maggiore verranno aggiunti per ultimi alla cima dell'albero, quindi avranno una profondità bassa, ovvero la lunghezza dalla codeword corta, proprio come volevo.

Q: coda di priorità (Heap) di nodi con chiave $f(z)$

Attributi di un nodo:

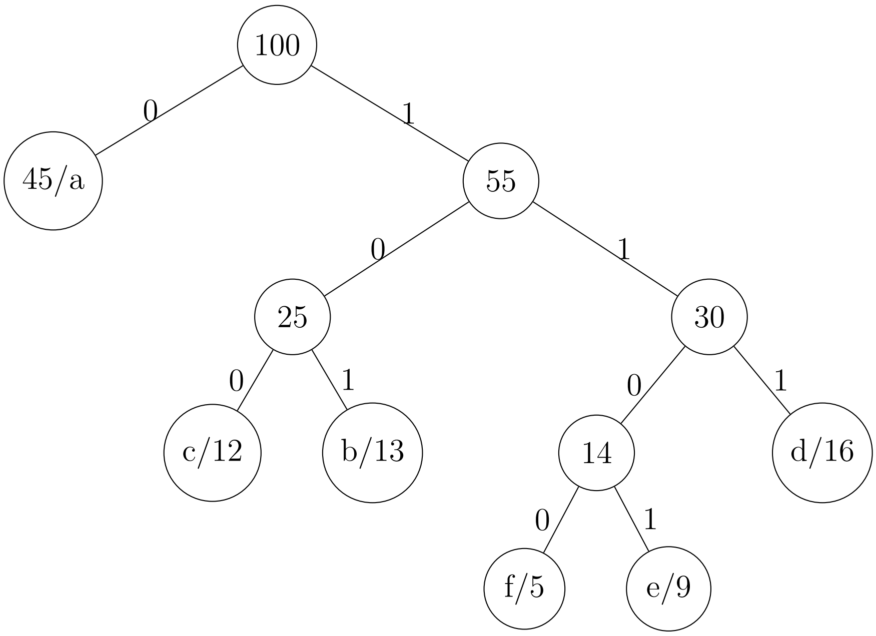
- $z.left$
- $z.right$
- $z.f$

Pseudocodice

```
HUFFMAN(C, f)
n = |C|
Q = ∅
for each c ∈ C // inizializzazione
    z = NEW-NODE() // crea un nuovo nodo
    z.f = f(c)
    z.left = NIL
    z.right = NIL
    INSERT(Q, z) // Θ(log n)
for i = 1 to n − 1
    x = EXTRACT-MIN(Q)
    y = EXTRACT-MIN(Q)
    z = NEW-NODE()
    z.left = x
    z.right = y
    z.f = x.f + y.f
    INSERT(Q, z)
```

Complessità $\Theta\left(\sum_{i=1}^{n-1} \log(n-i)\right) = \Theta(n \log n)$

Esempio Riprendiamo l'esempio iniziale e applichiamo l'algoritmo.



	a	b	c	d	e	f
codeword	0	101	100	111	1101	1100

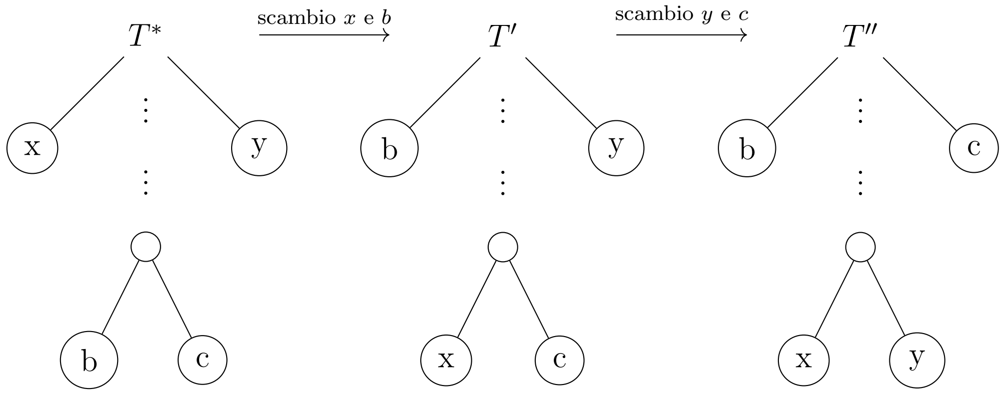
$$B(T) = \left(\frac{45}{100} \cdot 1 + \frac{13}{100} \cdot 3 + \frac{12}{100} \cdot 3 + \frac{16}{100} \cdot 3 + \frac{9}{100} \cdot 4 + \frac{5}{100} \cdot 4\right) = 224$$

8.3.1 Proprietà di Scelta Greedy

Sia C un alfabeto e siano x e y i caratteri in C di frequenza minore.
Allora esiste un codice prefisso ottimo T in cui x e y sono foglie attaccate alo stesso padre (sorelle).

Dimostrazione
Sia T^* una soluzione ottima arbitraria $\Rightarrow B(T^*)$ è minima.
Non so dove siano x e y in T^*
Siano b e c le foglie sorelle di profondità massima in T^*

1. $d_{T^*}(b) = d_{T^*}(c) \geq d_{T^*}(x), d_{T^*}(y)$
2. $f(x) \leq f(b), \quad f(y) \leq f(c)$



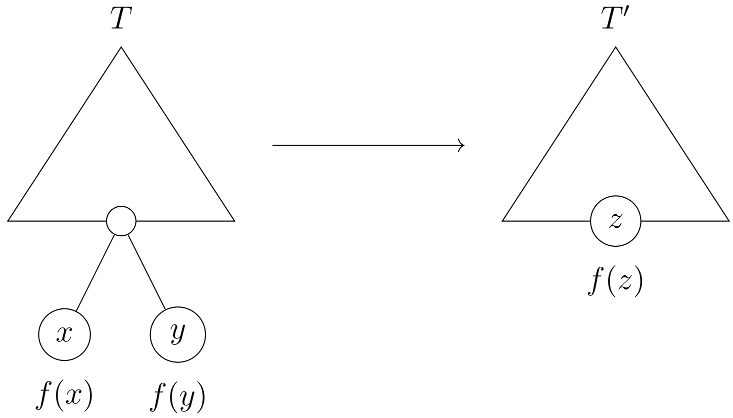
$T^* \to T'$
$B(T^*) \geq B(T')$
$B(T^*) - B(T') \geq 0$

$$
\begin{aligned}
B(T^*) - B(T') &= \sum_{c \in C} f(c)\, d_{T^*}(c) - \sum_{c \in C} f(c)\, d_{T'}(c) \\
&= f(b)\, d_{T^*}(b) + f(x)\, d_{T^*}(x) - f(b)\, \underbrace{d_{T'}(b)}_{d_{T^*}(b)} - f(x)\, \underbrace{d_{T'}(x)}_{d_{T^*}(x)} \\
&= f(b)\, d_{T^*}(b) + f(x)\, d_{T^*}(x) - f(b)\, d_{T^*}(b) - f(x)\, d_{T^*}(x) \\
&= f(b)\big(d_{T^*}(b) - d_{T^*}(x)\big) - f(x)\big(d_{T^*}(b) - d_{T^*}(x)\big) \\
&= \Big(\underbrace{f(b) - f(x)}_{\geq 0 \text{ perchè } f(x) \leq f(b)} \Big)\Big(\underbrace{d_{T^*}(b) - d_{T^*}(x)}_{\geq 0 \text{ per il punto 1}} \Big) \geq 0
\end{aligned}
$$

$T' \to T''$ si dimostra analogamente.

8.3.2 Proprietà di Sottostruttura Ottima

Sia T un codice prefisso ottimo che contiene la scelta greedy (sui caratteri x e y).
Sia z un nuovo carattere (cioè $\notin C$) con $f(z) = f(x) + f(y)$.
Allora il codice prefisso $T' = T \setminus \{x, y\}$ è ottimo per $C \setminus \{x, y\} \cup \{z\}$



T ha n foglie.
T' ha $n-1$ foglie.
Cerco una relazione tra $B(T)$ e $B(T')$.

Osservazione

1. $\forall\ c \in C \setminus \{x,y\},\ c \neq z : d_{T'}(c) = d_T(c)$
2. $d_{T'}(z) = d_T(x) - 1$

Dimostrazione

$$\begin{aligned} B(T) &= \sum_{c\in C} f(c)\, d_T(c) && \big(d_T(x) = d_T(y)\big) \\ &= \sum_{c\in C\setminus\{x,y\}} f(c)\, d_T(c) + \big(f(x)+f(y)\big)d_T(x) \\ &= \sum_{c\in C\setminus\{x,y\}} f(c)\, d_T(c) + \big(\underbrace{f(x)+f(y)}_{f(z)}\big)\big(\underbrace{d_T(x)-1}_{d_{T'}(z)}\big) + \big(f(x)+f(y)\big) \\ &= \sum_{c\in C\setminus\{x,y\}\cup\{z\}} f(c)\, d_{T'}(c) + f(x) + f(y) \\ &= B(T') + f(x) + f(y) \end{aligned}$$

Suppongo per assurdo che T non sia il codice prefisso ottimo per il sottoproblema $C \setminus \{x,y\} \cup \{z\}$.
Allora $\exists\ T''$ di costo strettamente inferiore.

$$B(T) = \underbrace{B(T'')}_{<B(T')} + f(x) + f(y)$$

Assurdo!