Dispense di algoritmi e strutture dati

Leonardo De Faveri

A.A. 2021/2022

Indice

Capitolo Nr.1

Analisi di algoritmi

1.1 Studio degli algoritmi

Studiare gli algoritmi significa studiarne la *complessità* e, l'obiettivo di quest'analisi, è quello di realizzare nuove versioni di quegli stessi algoritmi che godano di una *complessità* minore e siano quindi più efficienti.

Definizione 1 - Complessità di un algoritmo.

Studiare la complessità di un algoritmo significa fare l'analisi delle risorse impiegate da un algoritmo per risolvere un problema, in funzione della dimensione e della tipologia di input.

A questo punto la *complessità* si misura con una funzione del tipo:

$$\text{Complessità} : \text{"Dimensione dell'input"} \rightarrow \text{"Tempo di esecuzione"}$$

Ma cosa si intende con *dimensione dell'input*?

Dimensione dell'input Ci sono due criteri per stimare la dimensione:

- *Criterio di costo uniforme*: la taglia dell'input è il numero di elementi di cui è costituito;
- *Criterio di costo logaritmico*: la taglia dell'input è il numero di bit necessari per rappresentarlo;

E con *tipologia di input*? Questo aspetto non impatta su tutti gli algoritmi allo stesso modo. Il caso in cui è più facile comprenderne le conseguenze è quello dell'ordinamento di array. L'algoritmo *insertion sort*, per esempio, si adatta più efficientemente a situazioni in cui gli elementi vengono forniti uno per volta e l'algoritmo *selection sort* invece, è preferibile quando si hanno già tutti gli elementi da ordinare[1].

Tempo di esecuzione Il *tempo di esecuzione* coincide con il numero di istruzioni elementari.

Definizione 2 - Istruzione elementare.

Un'istruzione si considera elementare se può essere eseguita in tempo "costante" dal processore.

[1] La complessità di questi algoritmi verrà discussa più nel dettaglio nel corso di questo capitolo

1.1.1 Modello di calcolo

Nello studio degli algoritmi, un parametro da tenere in considerazione è il *modello di calcolo* utilizzato in quanto modelli diversi potrebbero cambiare, in meglio o in peggio, la *complessità* e l'efficienza di un algoritmo.

Definizione 3 - Modello di calcolo.

Un modello di calcolo è una rappresentazione astratta di un calcolatore.

Analizziamo nel dettaglio questa definizione per capire quali caratteristiche debba avere un *modello di calcolo*:

- *Astrazione*: deve nascondere i dettagli tecnico-operativi;
- *Realismo*: deve riflettere la situazione reale del sistema di calcolo;
- *Potenza matematica*: deve consentire di trarre conclusioni *formali* sul costo;

Esistono molteplici *modelli di calcolo* e il modello della *Macchina di Turing* ne è un esempio. Tuttavia, nel corso della trattazione faremo riferimento al *Modello RAM*[2] caratterizzato da:

- *Memoria*: è costituita da infinite celle di dimensione finita
- *Processore*: ha un set di istruzioni simili a quelli reali:
  - Operazioni logico-aritmetiche;
  - Istruzioni di controllo;
- *Costo uniforme*: il costo delle istruzioni elementari è uniforme;

1.2 Studio di alcuni algoritmi

Proviamo a vedere la logica dietro l'analisi della *complessità* con degli esempi.

1.2.1 Minimo di un array

Di seguito viene riportato l'algoritmo per l'estrazione del minimo elemento di un array. La funzione prende in input un array di n elementi e la relativa dimensione n.

Frammento 1 - Implementazione funzione min per la ricerca del minimo.

	Costo	N. esecuzioni
`ITEM min(ITEM[] A, int n)`		
`  ITEM min = A[1]`	c_1	1
`  for (i = 2 to n) do`	c_2	n
`    if (A[i] < min) then`	c_3	$n-1$
`      min = A[i]`	c_4	$n-1$
`  return min`	$c5$	1

Analisi del costo Analizziamo il costo di questo algoritmo. Ogni operazione elementare ha un tempo costante di esecuzione che indicheremo con c_i e ogni istruzione viene eseguita un certo numero di volte.

[2] *RAM: Random Access Machine*

Indichiamo con $T(n)$ la *funzione di costo* dell'algoritmo, ovvero la funzione che ne traccia il tempo di esecuzione. Calcoliamo $T(n)$ sommando il tempo di esecuzione di ogni istruzione considerando anche il numero di volte che ogni istruzione viene eseguita.

$$\begin{aligned} T(n) &= c_1 + c_2 n + c_3(n-1) + c_4(n-1) + c_5 \\ &= (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) \\ &= an + b \end{aligned}$$

1.2.2 Ricerca binaria ricorsiva

Consideriamo l'algoritmo per la ricerca binaria.

Frammento 2 - Implementazione funzione binarySearch per la ricerca binaria.

```
int binarySearch(ITEM[] A, ITEM v, int i, int j)
    if (i > j) then
        return 0
    else
        int m = ⌊(i + j) / 2⌋
        if (A[m] = v) then
            return m
        else if (A[m] < v) then
            return binarySearch(A, v, m + 1, j)
        else
            return binarySearch(A, v, i, m - 1)
```

Costo	N. $(i > j)$	N. $(i \leq j)$
c_1	1	1
c_2	1	0
c_3	0	1
c_4	0	1
c_5	0	0
c_6	0	1
$c_7 + T(\lfloor (n-1)/2 \rfloor)$	0	0/1
$c_7 + T(\lfloor n/2 \rfloor)$	0	1/0

La funzione prende in input un array A ordinato in senso crescente, l'elemento v da cercare e due indici i e j che indicano rispettivamente l'estremo destro e sinistro della porzione di array all'interno della quale ricercare l'elemento[3].

Nell'algoritmo sono presenti delle selezioni, che "modificano" il costo in termini di tempo dell'algoritmo. Viene selezione un elemento "centrale" nell'array e, se non è l'elemento cercato, si applica ricorsivamente l'algoritmo alla parte di sinistra, se l'elemento cercato è minore di quello analizzato, oppure alla parte destra. Per questa caratteristica dell'algoritmo si ha che la parte di sinistra ha dimensione $(n-1)/2$, mentre quella di destra $n/2$.

Analizziamo il caso pessimo, cioè quello in cui l'elemento cercato non è presente e quindi viene controllato l'intero array. Scegliamo di agire in questo modo così da poter dare una stima del tempo massimo di esecuzione. Per lo stesso motivo, ipotizziamo che l'elemento cercato sia maggiore di ogni elemento presente nell'array e che quindi la chiamata ricorsiva venga effettuata sempre sulla parte di destra che ha una dimensione maggiore.

Inoltre, per semplicità, assumiamo che l'array contenga un numero 2^k di elementi e assegniamo ad ogni istruzione elementare un costo c_i.

Nella stima del costo dell'algoritmo abbiamo due casi:

- $i > j$ per $n = 0$ quindi $T(n) = c_1 + c_2 = c$;
- $i \leq j$ per $n > 0$ quindi $T(n) = T(n/2) + c_1 + \cdots + c_7 = T(n/2) + d$;

Possiamo vedere il tutto come una funzione che determina il costo dell'algoritmo:

$$T(n) = \begin{cases} T(n/2) & n \geq 1 \\ c & n = 0 \end{cases}$$

[3] All'inizio indicano rispettivamente l'indice del primo e dell'ultimo elemento

Poiché $T(n/2)$ rappresenta una chiamata ricorsiva, possiamo analizzare le varie chiamate:

$$\begin{aligned} T(n) = &= T(n/2) + d \\ &= T(n/4) + 2d \\ &= T(n/8) + 3d \\ &\dots \\ &= T(1) + kd \qquad n = 2^k \Rightarrow k = \log n \\ &= T(0) + (k+1)d \\ &= kd + (c+d) \\ &= d\log(n) + e \end{aligned}$$

1.2.3 Ordini di complessità

Dall'analisi dei due algoritmi abbiamo ottenuto due espressioni matematiche: $an+b$ e $d\log(n)+e$[4]. Viste quelle espressioni, dette *funzioni di complessità*, possiamo affermare che i due algoritmi hanno rispettivamente un costo *polinomiale* e *logaritmico*.

Queste due tipologie di costo sono anche dette *ordini* o *classi di complessità* e la tabella sottostante ne riassume i principali.

f(n)	$\mathbf{10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$	**Tipo**
$\log n$	3	6	9	13	Logaritmico
$\sqrt{n}$	3	10	31	100	Sublineare
n	10	100	1000	10000	Lineare
$n\log n$	30	664	9965	132877	Loglineare
n^2	10^2	10^4	10^6	10^8	Quadratico
n^3	10^3	10^6	10^9	10^{12}	Cubico
2^n	1024	10^{30}	10^{300}	10^{3000}	Esponenziale

1.3 Notazione asintotica

Abbiamo quindi introdotto il concetto di *ordine di complessità* come metro di misura del costo di un algoritmo. Ora andremo ad introdurre uno strumento, la *notazione asintotica*, che ci permetterà di descrivere in maniera più formale il costo di un algoritmo.

Quelle che andremo ad introdurre in realtà sono tre notazioni, indicate rispettivamente dalle lettere O, Ω, Θ.

Definizione 4 - Notazione O.

Sia $g(n)$ una funzione di costo. Si indica con $O(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists\, c > 0, \exists\, m \geq 0 : f(n) \leq c \cdot g(n) \quad \forall\, n \geq m$$

Le funzioni $f(n)$ che rispettano questa disuguaglianza sono dette essere *O-grandi* di $g(n)$ e si scrive in simboli $f(n) = O(g(n))$. $g(n)$ è un *limite asintotico superiore* di $f(n)$, ciò significa che $f(n)$ cresce al più come $g(n)$.

[4] Il logaritmo è sottinteso essere in base 2

Definizione 5 - Notazione Ω.

Sia $g(n)$ una funzione di costo. Si indica con $\Omega(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists\, c > 0, \exists\, m \geq 0 : f(n) \geq c \cdot g(n) \quad \forall\, n \geq m$$

Le funzioni $f(n)$ che rispettano questa disuguaglianza sono dette essere Ω-*grandi* di $g(n)$ e si scrive in simboli $f(n) = \Omega(g(n))$. $g(n)$ è un *limite asintotico inferiore* di $f(n)$, ciò significa che $f(n)$ cresce almeno come $g(n)$.

Definizione 6 - Notazione Θ.

Sia $g(n)$ una funzione di costo, Si indica con $\Theta(g(n))$ l'insieme delle funzioni $f(n)$ tali per cui:

$$\exists\, c_1 > 0, \exists\, c_2 > 0, \exists\, m \geq 0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall\, n \geq m$$

Le funzioni $f(n)$ che rispettano quella disuguaglianza sono dette essere Θ di $g(n)$ e si scrive in simboli $f(n) = \Theta(g(n))$. $f(n)$ è un Θ di $g(n)$, ovvero $f(n) = \Theta(g(n))$, se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$, da cui deriva che $f(n) = \Theta(g(n))$ significa che $f(n)$ cresce esattamente come $g(n)$.

Graficamente, se $f(n) = \Theta(g(n))$, si osserva un comportamento di questo tipo:

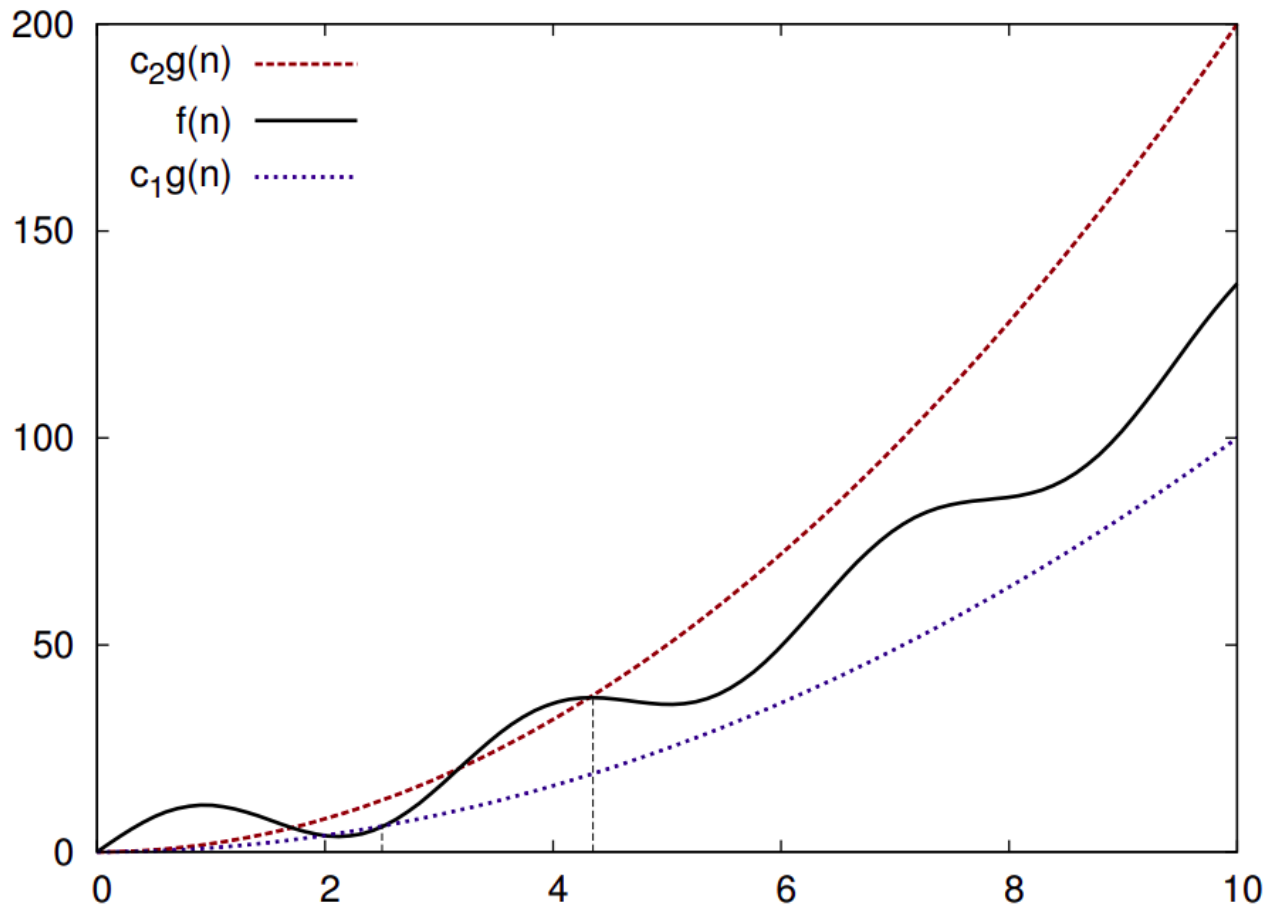


Fig. 1.1: Curva di crescita di $f(n) = \Theta(g(n))$

Si noti come all'inizio la curva di $f(n)$ sembri disattendere le stime, ma essendo *stime asintotiche* esse valgono per $n \to +\infty$ e infatti, superata una certa soglia, la curva di $f(n)$ rimane costantemente all'interno della porzione di grafico disegnata dalle due curve di $g(n)$.

Esempio 1 - Verifica delle proprietà di un *O*-grande.

Si dimostri che $f(n) = 10n^3 + 2n^2 + 7$ *è un* O*-grande di* n^3.

Riprendendo la definizione di O*-grande, dobbiamo dimostrare che:*

$$\exists\, c > 0, \exists\, m \geq 0 : f(n) \leq c \cdot n^3 \quad \forall\, n \geq m$$

Procediamo:

$$\begin{array}{rcll} f(n) & = & 10n^3 + 2n^2 + 7 & \\ & \leq & 10n^3 + 2n^3 + 7 & \forall\, n \geq 1 \\ & \leq & 10n^3 + 2n^3 + n^3 & \forall\, n \geq \sqrt[3]{7} \\ & = & 13n^3 & \end{array}$$

A questo punto ci chiediamo se esistono un valore $c > 0$ *e un valore* $m \geq 0$ *tali che:*

$$13n^3 \leq c \cdot n^3$$

La risposta è banale poiché basta prendere $c \geq 13$ *e* $m \geq 1 \geq \sqrt[3]{7}$*, ad esempio,* $m = 2$.

NB. In questo esempio abbiamo potuto moltiplicare $2n^2$ per n e sostituire il 7 con n^3, perché $n \geq 0$. In realtà, nelle *funzioni di costo* n sarà sempre *non negativo*, quindi lo potremo sempre fare.

Esempio 2 - Verifica delle proprietà di un Θ.

Si dimostri che $f(n) = 3n^2 + 7n$ *è un* Θ *di* n^2.

Procediamo verificando singolarmente i due estremi, inferiore e superiore. Iniziamo con lo studio del limite inferiore, cioè Ω*-grande. Ricordando la definizione di* Ω*-grande dobbiamo dimostrare che:*

$$\exists\, c_1 > 0, \exists\, m_1 \geq 0 : f(n) \geq c_1 \cdot n^2 \quad \forall\, n \geq m_1$$

Quindi:

$$\begin{array}{rcll} f(n) & = & 3n^2 + 7n & \\ & \geq & 3n^2 & \forall\, n \geq 0 \\ & \geq & c_1 \cdot n^2 & \forall\, c_1 \leq 3 \wedge \forall n \geq 0 \end{array}$$

Pongo quindi $m_1 = 0$.

Passiamo ora allo studio del limite superiore. Come nell'esempio precedente dobbiamo dimostrare che:

$$\exists\, c_2 > 0, \exists\, m_2 \geq 0 : f(n) \leq c_2 \cdot n^2 \quad \forall\, n \geq m_2$$

Quindi:

$$\begin{array}{rcll} f(n) & = & 3n^2 + 7n & \\ & \leq & 3n^2 + 7n^2 & \forall\, n \geq 1 \\ & = & 10n^2 & \\ & \leq & c_2 \cdot n^2 & \forall\, c_2 \geq 10 \wedge \forall n \geq 1 \end{array}$$

Pongo quindi $m_2 = 1$.

Torniamo ora al problema originario e riprendiamo la definizione di Θ*:*

$$\exists\, c_1 > 0, \exists\, c_2 > 0, \exists\, m \geq 0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall\, n \geq m$$

Chiaramente, possiamo prendere $c_1 = 3$ *e* $c_2 = 10$*, mentre, poiché l'ultima disequazione dev'essere vera* $\forall\, n \geq m$*, pongo* $m = \max\{m_1, m_2\} = \max\{0, 1\} = 1$.

Nel grafico seguente possiamo verificare la correttezza del nostri calcoli:

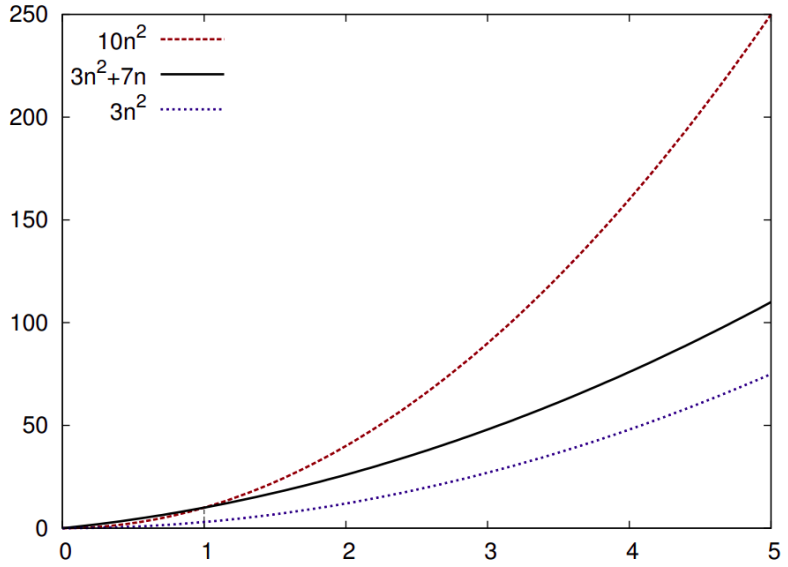


Fig. 1.2: Curva di $f(n) = 3n^2 + 7n$

1.4 Complessità degli algoritmi e dei problemi

Solitamente, per risolvere uno stesso problema esistono una caterva di algoritmi diversi e inevitabilmente alcuni sono più efficienti di altri. In che modo lo studio della *complessità* di questi *algoritmi* può aiutarci a capire la *complessità del problema*?

Per rispondere a questa domanda sfruttiamo di nuovo la *notazione asintotica*.

Definizione 7 - Notazione O.

Un problema ha complessità $O(f(n))$ se esiste almeno un algoritmo in grado di risolverlo che ha complessità $O(f(n))$.

Definizione 8 - Notazione Ω.

Un problema ha complessità $\Omega(f(n))$ se tutti i possibili algoritmi che lo risolvono hanno complessità $\Omega(f(n))$.

Notazione asintotica per gli algoritmi e per i problemi Nell'analisi della *complessità degli algoritmi*, possiamo riassumere come segue il significato delle *notazioni asintotiche*:

- $O(f(n))$: per tutti gli input n, l'algoritmo costa al più $f(n)$;
- $\Omega(f(n))$: per tutti gli input n, l'algoritmo costa almeno $f(n)$;
- $\Theta(f(n))$: per tutti gli input n, l'algoritmo costa $f(n)$;

Nell'analisi della *complessità dei problemi*, vale invece:

- $O(f(n))$: $O(f(n))$ è la complessità del miglior algoritmo in grado di risolvere il problema;
- $\Omega(f(n))$: vale se si riesce a dimostrare che nessun algoritmo può risolvere il problema in un tempo inferiore a $\Omega(f(n))$;

Se riusciamo a dimostrare che un problema ha complessità $O(f(n))$ e $\Omega(f(n))$, un algoritmo con *complessità* $\Theta(f(n))$ in grado di risolvere quel problema, è il miglior algoritmo possibile.

1.5 Ordinamento degli array in senso crescente[5]

Consideriamo ora tre diversi approcci al problema dell'ordinamento di array e analizziamone la *complessità*.

1.5.1 Algoritmo Selection sort

Questo algoritmo segue un approccio molto intuitivo: cerca il minimo tra tutti gli elementi, lo ordina e poi ripete il tutto per i restanti elementi.

Frammento 3 - Implementazione algoritmo Selection Sort.

```
SelectionSort(ITEM[] A, int n)
for (i = 1 to n - 1) do
   int min = min(A, i, n)
   A[i] ↔ A[min]              % Scambia il minimo attuale e l'elemento A[i]
```

Frammento 4 - Implementazione funzione min con indice di partenza arbitrario.

```
int min(ITEM[] A, int i, n)
   int min = i                                  % Posizione del minimo parziale
   for (j = i + 1 to n) do
      if (A[j] < A[min]) then
         min = j                                        % Nuovo minimo parziale
   return min
```

Quali sono le *complessità* nei casi pessimo, medio e migliore?

Il caso pessimo è quello in cui l'array è ordinato in senso decrescente, per cui ad ogni iterazione il minimo si trova nell'ultima posizione. Il caso migliore è invece quello in cui l'array è già ordinato in senso crescente. Nonostante questo, possiamo intuire che il costo dell'algoritmo non cambi, perché in ogni caso si dovrà ricercare il minimo $n-1$ volte.

Provando a definire la *funzione di costo* di questo algoritmo abbiamo:

$$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = {}^{6}\frac{n(n-1)}{2} = n^2 - \frac{n}{2} = O(n^2)$$

Questo vale per tutti i casi, quindi l'algoritmo *insertion sort* ha un costo $\Theta(n^2)$.

1.5.2 Algoritmo Insertion sort

Vediamo ora un approccio diverso al problema nel quale proviamo a prendere un elemento per volta e a metterlo nella posizione giusta.

In particolare, ad ogni iterazione viene preso un valore e confrontato progressivamente con i valori precedenti. Se viene trovato un valore minore di quello preso, quest'ultimo viene salvato nella penultima cella controllata.

[5] Il ragionamento è analogo per ordinamenti in senso decrescente

[6] *Formula di Gauss*

Frammento 5 - Implementazione algoritmo Insertion Sort.

```
insertionSort(ITEM[] A, int n)
   for (i = 2 to n) do
      ITEM temp = A[i]
      int j = i
      while (j > 1 and A[j - 1] > temp) do
         A[j] = A[j - 1]
         j = j - 1
      A[j] = temp
```

Cosa succede nei casi pessimo, medio e migliore?

Come prima, il caso pessimo è quello in cui l'array è ordinato in senso contrario. In una situazione del genere, per ogni i devono essere fatti $i-1$ confronti. Si ha quindi una funzione di questo tipo:

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

Nel caso migliore invece, il vettore è già ordinato nel senso corretto col risultato che ad ogni iterazione il controllo `A[j-1] > temp` è sempre falso e quindi la funzione di costo è:

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

1.5.3 Algoritmo Merge sort

Questo algoritmo è basato sulla tecnica del *divide-et-impera*, cioè procede suddividendo l'array in due metà che vengono poi ordinate separatamente e quindi ricombinate per ottenere l'array di partenza ordinato.

Come sarà facile intuire, questo algoritmo sfrutta la ricorsione per suddividere l'array in due componenti e una qualche funzione per l'unione dei due sottoarray ordinati.

Frammento 6 - Implementazione algoritmo Merge sort.

```
mergeSort(ITEM[] A, int first, int last)
   if (first < last) then
      int mid = ⌊(first + last) / 2⌋
      mergeSort(A, first, mid)                    % Ordina la parte sinistra
      mergeSort(A, mid + 1, last)                   % Ordina la parte destra
      merge(A, first, last, mid)            % Combina i due sottoarray ordinati
```

Nella funzione `merge` ipotizziamo di avere a disposizione un array di appoggio B nel quale andiamo ad inserire, in modo ordinato, i valori dei due sottoarray di A. Una volta riempito, tutti i valori di B vengono ricopiati di nuovo su A.

Frammento 7 - Implementazione funzione merge.

```
merge(ITEM[] A, int first, int last, int mid)
   int i, j, k, h
```

```
i = first
j = mid + 1
k = first                          % Indice delle posizioni nell'array di appoggio
while (i ≤ mid and j ≤ last) do                                % Popolamento di B
   if (A[i] ≤ A[j]) then
      B[k] = A[i]
      i = i + 1
   else
      B[k] = A[j]
      j = j + 1
   k = k + 1
j = last
for (h = mid downto i) do
   A[j] = A[h]
   j = j - 1
for (j = first to k - 1) do                                    % Copia di B su A
   A[j] = B[j]
```

Qual è la complessità del *merge sort*?

Studiamo per prima la sola funzione *merge*. Nel ciclo `while` e nel secondo ciclo `for`, vengono fatte n assegnazioni, dunque la funzione ha complessità $O(n)$.

Passando a considerare l'algoritmo nel suo insieme vediamo che l'array viene diviso in due metà e poi viene invocata *merge*. Ciò significa che se si tracciano i partizionamenti dell'array durante l'esecuzione dell'algoritmo, si ottiene un albero binario di altezza $k = \log n$[7] nel quale, ad ogni livello i, la funzione *merge* viene invocata 2^i volte.

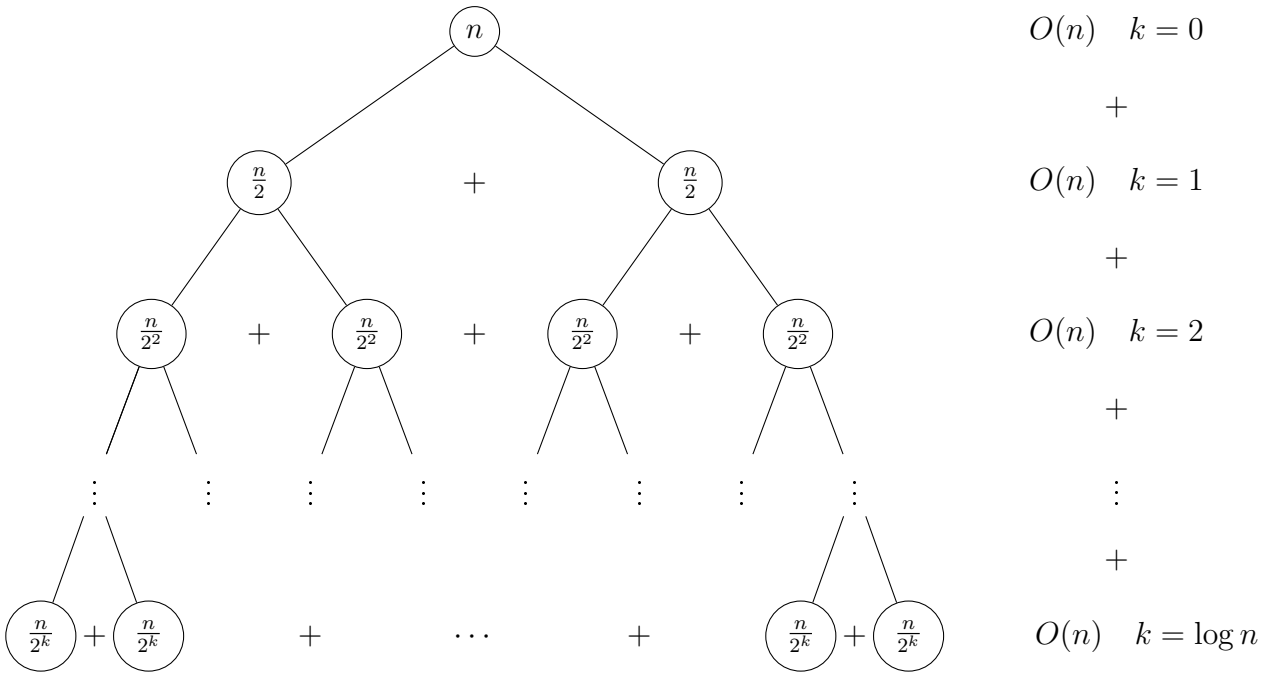


Fig. 1.3: Schema d'esecuzione del *Merge sort*

Ogni livello ha un costo $O(n)$, quindi, essendoci $k = \log n$ livelli, il costo dell'algoritmo è

[7] Per semplicità ipotizziamo che il numero di elementi dell'array sia una potenza di 2

$O(n \log n)$. Possiamo verificare il nostro risultato svolgendo direttamente i calcoli:

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) = O(n \log n)$$

Nel primo passaggio, l'argomento della sommatoria è quello perché per ogni livello sommiamo il costo necessario ad eseguire il merge su ogni sottoarray di dimensione $\frac{n}{2^i}$ (e.g. il sottoarray al livello 1 avrà dimensione $\frac{n}{2^1}$), ma consideriamo anche la numerosità dei sottoarray: 2^i per ogni livello i.

Ragionando anche sulla *funzione di costo* constatiamo che, poiché ad ogni livello vengono effettuate due invocazioni della `mergeSort` e un'invocazione della `merge`, che ha costo $O(n)$, vale la seguente:

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

Capitolo Nr.2

Notazione asintotica

Nel precedente capitolo abbiamo introdotto velocemente la *notazione asintotica*, ma ora la riprendiamo per approfondirne le caratteristiche e iniziamo con l'esplicitare una regola generale che finora abbiamo sfruttato senza pensarci troppo.

Definizione 9 - Regola generale per le espressioni polinomiali.

Per ogni espressione polinomiale di grado k del tipo

$$f(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0 \quad a_k > 0$$

vale:

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

Dimostrazione. La condizione necessaria per poter dire che $f(n) = \Theta(n^k)$ è:

$$f(n) = O(n^k) \quad \wedge \quad f(n) = \Omega(n^k)$$

Iniziamo quindi con la verifica del *limite superiore*, ovvero dimostriamo che vale la seguente:

$$\exists\, c > 0, \exists\, m \geq 0 : f(n) \leq c \cdot n^k \quad \forall\, n \geq m$$

Procediamo:

$$\begin{array}{rcll} f(n) & = & a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0 & \\ & \leq & a_k \cdot n^k + |a_{k-1}| \cdot n^{k-1} + \cdots + |a_1| \cdot n + |a_0| & \\ & \leq & a_k \cdot n^k + |a_{k-1}| \cdot n^k + \cdots + |a_1| \cdot n^k + |a_0| \cdot n^k & \forall\, n \geq 1 \\ & = & (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|) \cdot n^k & \\ & \leq & c \cdot k^n & \end{array}$$

L'ultima disequazione è vera per:

$$c \geq (a_k + |a_{k-1}| + \cdots + |a_1| + |a_0|) \quad \wedge \quad m = 1$$

Ora passiamo alla verifica del *limite inferiore* e quindi dimostriamo che vale anche la seguente:

$$\exists\, d > 0, \exists\, m \geq 0 : f(n) \geq d \cdot n^k \quad \forall\, n \geq m$$

Procediamo:

$$\begin{aligned} f(n) &= a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots + a_1 \cdot n + a_0 \\ &\geq a_k \cdot n^k - |a_{k-1}| \cdot n^{k-1} - \cdots - |a_1| \cdot n - |a_0| \\ &\geq a_k \cdot n^k - |a_{k-1}| \cdot n^{k-1} - \cdots - |a_1| \cdot n^{k-1} - |a_0| \cdot n^{k-1} \quad \forall\, n \geq 1 \\ &\geq d \cdot n^k \end{aligned}$$

L'ultima disequazione è vera per:

$$d \leq a_k - \frac{|a_{k-1|}}{n} - \frac{|a_{k-2}|}{n} - \cdots - \frac{|a_1|}{n} - \frac{|a_0|}{n}$$

Poiché $d > 0$, anche il termine destro della disequazione dev'essere > 0, e quindi deve valere:

$$a_k - \tfrac{|a_{k-1|}}{n} - \tfrac{|a_{k-2}|}{n} - \cdots - \tfrac{|a_1|}{n} - \tfrac{|a_0|}{n} > 0 \quad \Leftrightarrow \quad n \geq \tfrac{|a_{k-1}|+\cdots+|a_1|+|a_0|}{a_k} = m$$

□

2.1 Proprietà della notazione asintotica

Definizione 10 - Proprietà di dualità.

Per ogni coppia di funzioni di costo $f(n)$ e $g(n)$ vale:

$$f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$$

Dimostrazione.

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq c \cdot g(n) \quad \forall\, n \geq m \\ &\Leftrightarrow g(n) \geq \tfrac{1}{c} \cdot f(n) \quad \forall\, n \geq m \\ &\Leftrightarrow g(n) \geq c' \cdot f(n) \quad \forall\, n \geq m,\, c' = \tfrac{1}{c} \\ &\Leftrightarrow g(n) = \Omega(f(n)) \end{aligned}$$

□

Questa proprietà stabilisce che se $f(n)$ è un $O(g(n))$, allora $g(n)$ è un $\Omega(f(n))$.

Definizione 11 - Proprietà di eliminazione delle costanti.

Per ogni funzione di costo $f(n)$ vale:

$$f(n) = O(g(n)) \Leftrightarrow a \cdot f(n) = O(g(n)) \quad \forall\, a > 0$$

$$f(n) = \Omega(g(n)) \Leftrightarrow a \cdot f(n) = \Omega(g(n)) \quad \forall\, a > 0$$

Dimostrazione.

$$\begin{aligned} f(n) = O(g(n)) &\Leftrightarrow f(n) \leq c \cdot g(n) && \forall\, n \geq m \\ &\Leftrightarrow a \cdot f(n) \leq a \cdot c \cdot g(n) && \forall\, n \geq m,\, \forall\, a > 0 \\ &\Leftrightarrow a \cdot f(n) \leq c' \cdot g(n) && \forall n \geq m,\, c' = a \cdot c > 0 \\ &\Leftrightarrow a \cdot f(n) = O(g(n)) \end{aligned}$$

□

NB. La dimostrazione è analoga per $\Omega(g(n))$.

Questa proprietà permette di ignorare le costanti moltiplicative per le *funzioni di costo*.

Definizione 12 - Proprietà di massimo costo.

Per ogni coppia di funzioni di costo $f_1(n)$ e $f_2(n)$ vale:

$$f_1(n) = O(g_1(n)),\ f_2(n) = O(g_2(n)) \Rightarrow f_1(n) + f_2(n) = O(\max\{g_1(n),\ g_2(n)\})$$

$$f_1(n) = \Omega(g_1(n)),\ f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) + f_2(n) = \Omega(\max\{g_1(n),\ g_2(n)\})$$

Dimostrazione.

$$\begin{aligned} f_1(n) = O(g_1(n)) \land f_2(n) = O(g_2(n)) &\Rightarrow \\ f_1(n) \leq c_1 \cdot g_1(n) \land f_2(n) \leq c_2 \cdot g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leq c_1 \cdot g_1(n) + c_2 \cdot g_2(n) &\Rightarrow \\ f_1(n) + f_2(n) \leq \max\{c_1,\ c_2\} \cdot (2\max\{g_1(n),\ g_2(n)\}) &\Rightarrow \\ f_1(n) + f_2(n) = O(\max\{g_1(n),\ g_2(n)\}) & \end{aligned}$$

□

NB. La dimostrazione è analoga per $f_1(n) + f_2(n) = \Omega(\max\{g_1(n),\ g_2(n)\})$.

Questa proprietà stabilisce, che nel caso in cui si vada a sommare più *funzioni di costo* (e.g. sequenze di algoritmi), si può considerare solo la funzione di costo maggiore.

Definizione 13 - Proprietà del prodotto dei costi.

Per ogni coppia di funzioni di costo $f_1(n)$ e $f_2(n)$ vale:

$$f_1(n) = O(g_1(n)),\ f_2(n) = O(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

$$f_1(n) = \Omega(g_1(n)),\ f_2(n) = \Omega(g_2(n)) \Rightarrow f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$$

Dimostrazione.

$$\begin{aligned} f_1(n) = O(g_1(n)) \land f_2(n) = O(g_2(n)) &\Rightarrow \\ f_1(n) \leq c_1 \cdot g_1(n) \land f_2(n) \leq c_2 \cdot g_2(n) &\Rightarrow \\ f_1(n) \cdot f_2(n) \leq c_1 \cdot c_2 \cdot g_1(n) \cdot g_2(n) &\Rightarrow \\ f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)) & \end{aligned}$$

□

NB. La dimostrazione è analoga per $f_1(n) \cdot f_2(n) = \Omega(g_1(n) \cdot g_2(n))$.

Questa proprietà stabilisce che nel caso in cui si vada a moltiplicare tra loro più *funzioni di costo* (e.g. cicli annidati), il costo totale è proprio il prodotto dei costi delle singole funzioni.

Definizione 14 - Proprietà di simmetria.

Per ogni coppia di funzioni di costo $f(n)$ e $g(n)$ vale:

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

Dimostrazione. Grazie alla *proprietà di dualità* vale:

$$\begin{aligned} f(n) = \Theta(g(n)) &\Rightarrow f(n) = O(g(n)) \Rightarrow g(n) = \Omega(f(n)) \\ f(n) = \Theta(g(n)) &\Rightarrow f(n) = \Omega(g(n)) \Rightarrow g(n) = O(f(n)) \end{aligned}$$

□

Questa proprietà stabilisce che se $f(n)$ è un $\Theta(g(n))$ allora anche $g(n)$ è un $\Theta(f(n))$.

Definizione 15 - Proprietà di transitività.

Prese tre funzioni di costo $f(n)$, $g(n)$ e $h(n)$ tali che:

$$f(n) = O(g(n)) \quad \wedge \quad g(n) = O(h(n))$$

vale:

$$f(n) = O(h(n))$$

Dimostrazione.

$$\begin{aligned} f(n) = O(g(n)) \wedge g(n) = O(h(n)) \quad &\Rightarrow \\ f(n) \leq c_1 \cdot g(n) \wedge g(n) \leq c_2 \cdot h(n) \quad &\Rightarrow \\ f(n) \leq c_2 \cdot h(n) \quad &\Rightarrow \\ f(n) = O(h(n)) \end{aligned}$$

□

2.2 Altre notazioni

Definizione 16 - Notazione o.

Sia $g(n)$ una funzione di costo. Si indica con $o(g(n))$ l'insieme delle funzioni di costo tali per cui:

$$\forall\, c, \exists\, m : f(n) < c \cdot g(n) \quad \forall\, n \geq m$$

Le funzioni $f(n)$ che rispettano questa disuguaglianza sono dette essere *o-piccoli* di $g(n)$ e si scrive in simboli $f(n) = o(g(n))$.

Definizione 17 - Notazione ω.

Sia $g(n)$ una funzione di costo. Si indica con $\omega(g(n))$ l'insieme delle funzioni di costo tali per cui:

$$\forall\, c, \exists\, m : f(n) > c \cdot g(n) \quad \forall\, n \geq m$$

Le funzioni $f(n)$ che rispettano questa disuguaglianza sono dette essere *ω-piccoli* di $g(n)$ e si scrive in simboli $f(n) = \omega(g(n))$.

Osservazioni Date due *funzioni di costo* $f(n)$ e $g(n)$ possiamo fare le seguenti affermazioni:

$$\begin{aligned} \lim_{n\to+\infty} \tfrac{f(n)}{g(n)} = 0 \quad &\Rightarrow \quad f(n) = o(g(n)) \\ \lim_{n\to+\infty} \tfrac{f(n)}{g(n)} = c \neq 0 \quad &\Rightarrow \quad f(n) = \Theta(g(n)) \\ \lim_{n\to+\infty} \tfrac{f(n)}{g(n)} = +\infty \quad &\Rightarrow \quad f(n) = \omega(g(n)) \end{aligned}$$

Si noti anche che:

$$\begin{aligned} f(n) = o(g(n)) \quad &\Rightarrow \quad f(n) = O(g(n)) \\ f(n) = \omega(g(n)) \quad &\Rightarrow \quad f(n) = \Omega(g(n)) \end{aligned}$$

2.3 Classificazione delle funzioni di costo

Giunti a questo punto possiamo definire un ordinamento per le principali *funzioni di costo*:

Definizione 18 - Ordinamento delle funzioni di costo.

Per ogni $r < s$, $h < k$ e $a < b$ vale:

$$O(1) \subset O(\log^r n) \subset O(\log^s n) \subset O(n^h) \subset O(n^h \log^r n)$$

$$\subset O(n^h \log^s n) \subset O(n^k) \subset O(a^n) \subset O(b^n)$$

Capitolo Nr.3

Funzioni di ricorrenza

Quando si calcola la *complessità* di un algoritmo ricorsivo, questa viene espressa tramite un'*equazione* o una *funzione di ricorrenza*, ovvero una formula matematica definita in maniera ricorsiva.

Un esempio di *equazione di ricorrenza* è quella che descrive il costo dell'algoritmo *Merge sort*:

$$T(n)^{1} = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Il nostro obiettivo è ottenere, quando possibile, una *formula*, o *forma chiusa*, che rappresenti la *classe di complessità* della funzione. Nell'esempio del *Merge sort* la *formula chiusa* è:

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

In realtà, le *equazioni di ricorrenza* possono essere usate anche per risolvere problemi. Vediamone un esempio:

Problema 1 - Applicazione delle funzioni di ricorrenza nella risoluzione di problemi.

Un bambino scende una scala composta da n scalini. Ad ogni passo, può decidere di fare $1, 2, 3, 4$ *scalini alla volta. Determinare in quanti modi diversi può scendere le scale.*

Ad esempio, se $n = 7$*, alcuni dei modi possibili sono i seguenti:*

- *1,1,1,1,1,1,1;*
- *1,2,4;*
- *4,2,1;*
- *2,2,2,1;*
- *1,2,2,1,1;*

[1] Questa descrive il caso generale, mentre la precedente si basava sull'ipotesi di array di dimensione 2^i con $i \in \mathbb{N}$

Soluzione *Sia $M(n)$ il numero di modi in cui è possibile scegliere n scalini. $M(n)$ può essere espresso come segue:*

$$M(n) = \begin{cases} 0 & n < 0 \\ 1 & n = 0 \\ \sum_{k=1}^{4} M(n-k) & n > 0 \end{cases}$$

Questa *ricorrenza* può poi essere trasformata in un algoritmo[2] tramite *ricorsione* o *programmazione dinamica*.

3.1 Studio delle equazioni di ricorrenza

Vediamo di seguito alcuni modi per analizzare le *funzioni di ricorrenza* e ricavarne la *forma chiusa*.

3.1.1 Metodo dell'albero di ricorsione

Il *metodo dell'albero di ricorsione* (o per *livelli*) prevede che la *ricorrenza* venga "srotolata" in un albero i cui nodi rappresentano il costo ai vari livelli della ricorsione.

Esempio 3 - Esempio semplice di analisi con albero di ricorsione.

Sia $T(n)$ la seguente funzione di ricorrenza:

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

Ipotizziamo, per semplicità, che $n = 2^k$, quindi proviamo a risolvere $T(n) = T(2^k)$:

$$\begin{aligned} T(n) &= b + T(n/2) \\ &= b + b + T(n/4) \\ &= b + b + b + T(n/8) \\ &= \dots \\ &= \underbrace{b + \dots + b}_{\log n} + T(1) \\ &= b \cdot \log n + c \end{aligned}$$

Quindi, $T(n) = b \cdot \log k + c = \Theta(\log n)$.

Esempio 4 - Utilizzo delle serie matematiche nel processo di semplificazione.

Sia $T(n)$ la seguente funzione di ricorrenza:

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

[2] Il risultato sono i *numeri di Tetranacci*: $1, 1, 2, 4, 8, 15, 29, 56, 108, 208, 401, 773, 1490, 2872, 5536, \dots$.

Ipotizziamo sempre $n = 2^k$:

$$
\begin{aligned}
T(n) &= n + 4T(n/2) \\
&= n + 4(n/2) + 16T(n/4) \\
&= n + 2n + 16(n/4) + 64T(n/8) \\
&= \dots \\
&= n + 2n + 4n + 8n + \cdots + 2^{\log(n)-1}n + 4^{\log n}T(1) \\
&= \left(\sum_{i=0}^{\log(n)-1} 2^i\right) \cdot n + 4^{\log n}
\end{aligned}
$$

A questo punto, possiamo sfruttare la serie geometrica finita *e riscrivere la funzione come:*

$$
\begin{aligned}
T(n) &= \left(\sum_{i=0}^{\log(n)-1} 2^i\right) \cdot n + 4^{\log n} \\
&= \left(\frac{2^{\log n}-1}{2-1}\right) \cdot n + 4^{\log n} && \textit{Serie geometrica finita} \\
&= (2^{\log n} - 1) \cdot n + 4^{\log n} \\
&= (n^{\log 2} - 1) \cdot n + n^{\log 4} && \textit{Teorema di scambio base-argomento} \\
&= (n^1 - 1) \cdot n + n^2 \\
&= (n - 1) \cdot n + n^2 \\
&= n^2 - n + n^2 \\
&= 2n^2 - n
\end{aligned}
$$

Da qui è evidente che $T(n) = \Theta(n^2)$.

Esempio 5 - Utilizzo di una tabella dei costi.

Sia $T(n)$ la seguente funzione di ricorrenza:

$$
t(n) = \begin{cases} 4T(n/2) + n^3 & n > 1 \\ 1 & n \leq 1 \end{cases}
$$

Proviamo ad analizzare l'albero delle chiamate per i primi 3 livelli:

$$n^3$$

$$
\overbrace{
\left(\tfrac{n}{2}\right)^3 \quad \left(\tfrac{n}{2}\right)^3 \quad \left(\tfrac{n}{2}\right)^3 \quad \left(\tfrac{n}{2}\right)^3
}
$$

$$
\overbrace{\left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3}
\quad
\overbrace{\left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3}
\quad
\overbrace{\left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3}
\quad
\overbrace{\left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3 \left(\tfrac{n}{4}\right)^3}
$$

È chiaro che con una funzione di questo tipo, non è possibile procedere in questo modo. Una strategia migliore è quella di usare una tabella come la seguente.

Livello	Dim. input	Costo per chiamata	N. chiamate	Costo livello
0	n	n^3	1	n^3
1	$n/2$	$(n/2)^3$	4	$4(n/2)^3$
2	$n/4$	$(n/4)^3$	16	$16(n/4)^3$
...	...	...	...	...
i	$n/2^1$	$(n/2^i)^3$	4^i	$4^i(n/2^i)^3$
...	...	...	...	...
$\log(n)-1$	$n/2^{\log(n)-1}$	$\left(n/2^{\log(n)-1}\right)$	$4^{\log(n)-1}$	$4^{\log(n))-1}\left(n/2^{\log(n)-1}\right)^3$
$\log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Ora, il costo totale è dato dalla somma del costo di ogni livello. Vale quindi:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log(n)-1}\left(4^i\left(n/2^i\right)^3\right)+4^{\log n} \\
&= \left(\sum_{i=0}^{\log(n)-1}\frac{4^i}{2^{3i}}\right)\cdot n^3+4^{\log n} \\
&= \left(\sum_{i=0}^{\log(n)-1}\frac{2^{2i}}{2^{3i}}\right)\cdot n^3+4^{\log n} \\
&= \left(\sum_{i=0}^{\log(n)-1}\left(\frac{1}{2}\right)^i\right)\cdot n^3+4^{\log n} \\
&= \left(\sum_{i=0}^{\log(n)-1}\left(\frac{1}{2}\right)^i\right)\cdot n^3+n^{\log 4} && \textit{Teorema di scambio base-argomento} \\
&= \left(\sum_{i=0}^{\log(n)-1}\left(\frac{1}{2}\right)^i\right)\cdot n^3+n^2 \\
&\leq \left(\sum_{i=0}^{+\infty}\left(\frac{1}{2}\right)^i\right)\cdot n^3+n^2 && \textit{Estensione ad infinito della sommatoria}
\end{aligned}$$

Giunti a questo punto abbiamo:

$$T(n)\leq\left(\sum_{i=0}^{+\infty}\left(\frac{1}{2}\right)^i\right)\cdot n^3+n^2$$

Tuttavia, possiamo riconoscere nella sommatoria la serie geometrica infinita decrescente *e riscrivere $T(n)$ come:*

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

Abbiamo quindi dimostrato che $T(n)=O(n^3)$, tuttavia, poiché $T(n)\geq n^3$, possiamo anche affermare che $T(n)=\Omega(n^3)$, quindi $T(n)=\Theta(n^3)$.

NB. Abbiamo indicato n^3 come costo della prima chiamata perché di sicuro lo si dovrà pagare, ma in realtà il vero costo avrebbe dovuto essere $c\cdot n^3$ per qualche $c>0$. Abbiamo rimosso c per semplicità, ma grazie alla *Proprietà di eliminazione delle costanti* sappiamo che considerarlo non sarebbe comunque servito.

Esempio 6 - Ulteriore esempio di utilizzo di una tabella dei costi.

Sia $T(n)$ la seguente funzione di ricorrenza:

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

Procediamo utilizzando una tabella come quella di prima.

Livello	Dim. input	Costo per chiamata	N. chiamate	Costo livello
0	n	n^2	1	n^2
1	$n/2$	$(n/2)^2$	4	$4(n/2)^2$
2	$n/4$	$(n/4)^2$	16	$16(n/4)^2$
...	...	...	...	...
i	$n/2^i$	$(n/2^i)^2$	4^i	$4^i(n/2^i)^2$
...	...	...	...	...
$\log(n)-1$	$n/2^{\log(n)-1}$	$(n/2^{\log(n)-1})$	$4^{\log(n)-1}$	$4^{\log(n)-1}(n/2^{\log(n)-1})^2$
$\log n$	1	$T(1)$	$4^{\log n}$	$4^{\log n}$

Scriviamo come prima la funzione di ricorrenza sotto forma di sommatoria:

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log(n)-1}(4^i \cdot (n/2^i)^2) + 4^{\log n} \\ &= \left(\sum_{i=0}^{\log(n)-1} \frac{4^i}{2^2 i}\right) \cdot n^2 + 4^{\log n} \\ &= \left(\sum_{i=0}^{\log(n)-1} \frac{2^2 i}{2^2 i}\right) \cdot n^2 + 4^{\log n} \\ &= \left(\sum_{i=0}^{\log(n)-1} 1\right) \cdot n^2 + 4^{\log n} \\ &= \left(\sum_{i=0}^{\log(n)-1} 1\right) \cdot n^2 + n^{\log 4} \\ &= \left(\sum_{i=0}^{\log(n)-1} 1\right) \cdot n^2 + n^2 \\ &= n^2 \log n + n^2 \end{aligned}$$

Quindi $T(n) = \Theta(n^2 \log n)$.

3.1.2 Metodo della sostituzione

Con questo metodo si cerca di "indovinare" la *forma chiusa* di una *funzione di ricorrenza* e di dimostrarne la validità per induzione.

Esempio 7 - Esempio di intuizione corretta.

Sia $T(n)$ *la seguente funzione di ricorrenza:*

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Proviamo ad indovinarne la forma chiusa*:*

$$\begin{aligned} T(n) &= n + T(\lfloor n/2 \rfloor) \\ &= n + \tfrac{n}{2} + T(\lfloor n/4 \rfloor) \\ &= n + \tfrac{n}{2} + \tfrac{n}{4} + T(\lfloor n/8 \rfloor) \\ &= \dots \\ &= n + \tfrac{n}{2} + \tfrac{n}{4} + \tfrac{n}{8} + \dots + T(1) \\ &\leq 2n \end{aligned}$$

Vedendo questa catena di uguaglianze possiamo supporre $T(n) = O(n)$*. Proviamo quindi a dimostrarlo procedendo per induzione.*

Caso base *Dimostriamo la disequazione per* $T(1)$*:*

$$T(1) = 1 \leq 1 \cdot c \quad \forall\, c \geq 1$$

Il caso base è verificato.

Passo induttivo *Ipotizziamo che* $T(k) \leq c \cdot k \quad \forall\, k < n$ *e dimostriamo la disequazione per* $T(n)$*:*

$$\begin{array}{rcll} T(n) & = & T(\lfloor n/2 \rfloor) + n & \\ & \leq & c \cdot (\lfloor n/2 \rfloor) + n & \textit{Sostituzione} \\ & \leq & c \cdot (n/2) + n & \textit{Rimozione dell'intero inferiore} \\ & = & (c/2 + 1) \cdot n & \\ & \leq & c \cdot n & \end{array}$$

L'ultima disequazione è vera per:

$$\frac{c}{2} + 1 \leq c \Leftrightarrow c \geq 2$$

Anche il passo induttivo è verificato, dunque vale:

$$T(n) \leq c \cdot n \text{ per } \begin{cases} c \geq 1 & \textit{Caso base} \\ c \geq 2 & \textit{Passo induttivo} \end{cases}$$

Quelle condizioni sono verificate $\forall\, c \geq 2$ *e* $\forall\, n \geq 1 = m$*. Abbiamo quindi dimostrato la correttezza della nostra ipotesi, cioè* $T(n) = O(n)$*.*

Riusciamo a fare lo stesso sul limite inferiore? Al primo livello la funzione costa, di sicuro, almeno n*. È quindi plausibile aspettarsi che* $T(n) = \Omega(n)$*. Procediamo di nuovo per induzione.*

Caso base *Dimostriamo la disequazione per* $T(1)$*:*

$$T(1) = 1 \geq 1 \cdot d \quad \forall\, d \leq 1$$

Il caso base è verificato.

Passo induttivo *Ipotizziamo che* $T(k) \geq d \cdot k \quad \forall\, k < n$ *e dimostriamo la disequazione per* $T(n)$*:*

$$\begin{array}{rcll} T(n) & = & T(\lfloor n/2 \rfloor) + n & \\ & \geq & d \cdot (\lfloor n/2 \rfloor) + n & \textit{Sostituzione} \\ & \geq & d \cdot (n/2) - 1 + n & \textit{Rimozione dell'intero inferiore} \\ & = & (d/2 - 1/n + 1) \cdot n & \\ & \geq & d \cdot n & \end{array}$$

L'ultima disequazione è vera per:

$$\frac{d}{2} - \frac{1}{n} + 1 \geq d \Leftrightarrow d \leq 2 - \frac{2}{n}$$

Anche il passo induttivo è verificato, dunque vale:

$$T(n) \geq d \cdot n \text{ per } \begin{cases} d \leq 1 & \textit{Caso base} \\ d \leq 2 - \frac{2}{n} & \textit{Passo induttivo} \end{cases}$$

Quelle condizioni sono verificate per $d = 1$ *e* $\forall\, n \geq 2 = m$*. Abbiamo quindi verificato la correttezza della nostra ipotesi, cioè* $T(n) = \Omega(n)$*.*

Avendo dimostrato che $T(n)$ *è sia un* $O(n)$ *che un* $\Omega(n)$*, ne consegue che* $T(n)$ *è anche un* $\Theta(n)$*.*

NB. Nel precedente esempio abbiamo provato per induzione che

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases} = \Omega(n)$$

Tuttavia, è possibile giungere allo stesso risultato senza ricorrere alla ricorsione. Ricordiamo la definizione di Ω:

$$\exists\, d > 0,\, \exists\, m \geq 0 : f(n) \geq d \cdot g(n) \quad \forall\, n \geq m$$

Vale la seguente catena di disequazioni:

$$T(n) = T(\lfloor n/2 \rfloor) + n \geq n \geq d \cdot n \quad \forall\, d \leq 1$$

Questa condizione è la stessa del caso base della dimostrazione per induzione, dunque $T(n) = \Omega(n)$.

Esempio 8 - Esempio di intuizione errata.

Sia $T(n)$ la seguente funzione di ricorrenza:

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

Risolvendo questa funzione di ricorrenza per livelli vale:

$$\begin{aligned} T(n) &= n + T(n-1) \\ &= (n-1) + n + T(n-2) \\ &= (n-2) + (n-1) + n + T(n-3) \\ &= \dots \\ &= 1 + \dots + (n-2) + (n-1) + n \\ &= \textstyle\sum_{i=1}^{n} i = \frac{n(n-1)}{2} = \Theta(n^2) \end{aligned}$$

Supponiamo però di voler provare a dimostrare che $T(n) = O(n)$. Procediamo quindi per induzione.

Caso base *Dimostriamo la disequazione per $T(1)$:*

$$T(1) = 1 \leq 1 \cdot c \quad \forall\, c \geq 1$$

Il caso base è verificato.

Passo induttivo *Ipotizziamo che $T(k) \leq c \cdot k \quad \forall\, k < n$ e dimostriamo la disequazione per $T(n)$:*

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\leq c \cdot (n-1) + n && \textit{Sostituzione} \\ &= c \cdot n - c + n \\ &= (c+1)n - c \\ &\leq (c+1)n && \textit{Rimozione elemento negativo} \\ &\leq c \cdot n \end{aligned}$$

L'ultima disequazione è impossibile poiché per essere vera dovrebbe vale:

$$c + 1 \leq c$$

Dunque, $T(n) \neq O(n)$.

Esempio 9 - Limiti inferiori e superiori.

Sia $T(n)$ la seguente funzione di ricorrenza:

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Qual è il costo di questa funzione?

Iniziamo di nuovo procedendo per livelli.

Livello	Dim. input	Costo per chiamata	N. chiamate	Costo livello
0	n	1	1	1
1	$n/2$	2	2	2
2	$n/4$	4	4	4
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
i	$n/2^i$	2^i	2^i	2^i
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
$\log(n) - 1$	$n/2^{\log(n)-1}$	1	$2^{\log(n)-1}$	$2^{\log(n)-1}$
$\log n$	1	$T(1)$	$2^{\log n}$	$2^{\log n}$

Scrivendo ora la funzione di ricorrenza come sommatoria otteniamo:

$$\begin{aligned} T(n) &= \textstyle\sum_{i=0}^{\log n} 2^i \\ &= 2^0 + 2^1 + \cdots + 2^{\log(n)-1} + 2^{\log n} \\ &= 2^0 + 2^1 + \cdots + \tfrac{2^{\log n}}{2} + 2^{\log n} && \textit{Teorema del prodotto} \\ &= 2^0 + 2^i + \cdots + \tfrac{n^{\log 2}}{2} + n^{\log 2} && \textit{Teorema di scambio base-argomento} \\ &= 2^0 + 2^1 + \cdots + \tfrac{n}{2} + n = \Theta(n) \end{aligned}$$

Quindi, $T(n) = \Theta(n)$. Adesso però proviamo a procedere per sostituzione. *Dimostriamo quindi che $T(n) = O(n)$.*

Caso base *Dimostriamo la disequazione per $T(1)$:*

$$T(1) = 1 \leq 1 \cdot c \quad \forall c \geq 1$$

Il caso base è verificato.

Passo induttivo *Ipotizziamo che $T(k) \leq c \cdot k \quad \forall k < n$ e dimostriamo la disequazione per $T(n)$:*

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\leq c \cdot (\lfloor n/2 \rfloor) + c \cdot (\lceil n/2 \rceil) + 1 && \textit{Sostituzione} \\ &= c \cdot n + 1 \\ &\leq c \cdot n \end{aligned}$$

L'ultima disequazione è impossibile poiché si riduce a $1 \leq 0$ che è un'affermazione falsa.

Non siamo riusciti a dimostrare che $T(n) = O(n)$, tuttavia noi sappiamo che in realtà lo è. Cosa c'è di sbagliato? Abbiamo ottenuto a sinistra un termine troppo grande, quindi proviamo a partire da un'ipotesi più ristretta. Proviamo con $O(n - b)$.

Caso base *Dimostriamo la disequazione per* $T(1)$*:*

$$T(1) = 1 \leq 1 \cdot c - b \quad \forall\, c \geq b + 1$$

Il caso base è verificato.

Passo induttivo *Ipotizziamo che* $\exists\, b > 0 : T(k) \leq c \cdot k - b \quad \forall\, k < n$ *e dimostriamo la disequazione per* $T(n)$*:*

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\leq c \cdot (\lfloor n/2 \rfloor) - b + c \cdot (\lceil n/2 \rceil) - b + 1 \qquad \textit{Sostituzione} \\ &= c \cdot n - 2b + 1 \\ &\leq c \cdot n - b \end{aligned}$$

L'ultima disequazione è vera per:

$$-2b + 1 \leq -b \Leftrightarrow b \geq 1$$

Quindi, per verificare sia il caso base che il passo induttivo, è sufficiente porre $b = 1$ *e* $c = 2$*. Inoltre, questo vale per ogni* $n \geq 1$*, quindi poniamo anche* $m = 1$*. Abbiamo dunque provato che* $T(n) = O(n - b) = O(n)$*.*

A questo punto resta da dimostrare soltanto il limite inferiore, cioè che $T(n) = \Omega(n)$*.*

Caso base *Dimostriamo la disequazione per* $T(1)$*:*

$$T(1) = 1 \geq 1 \cdot d \quad \forall\, d \leq 1$$

Il caso base è verificato.

Passo induttivo *Ipotizziamo che* $T(k) \geq d \cdot k \quad \forall\, d \leq k$ *e dimostriamo la disequazione per* $T(n)$*:*

$$\begin{aligned} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ &\geq d \cdot (\lfloor n/2 \rfloor) + d \cdot (\lceil n/2 \rceil) + 1 \qquad \textit{Sostituzione} \\ &= d \cdot n + 1 \\ &\geq d \cdot n \end{aligned}$$

Chiaramente, l'ultima disequazione è sempre vera, dunque è dimostrato che $T(n) = \Omega(n)$ *e, di conseguenza, anche che* $T(n) = \Theta(n)$*.*

Esempio 10 - Problemi con i casi base.

Sia $T(n)$ *la seguente funzione di ricorrenza:*

$$T(n) = \begin{cases} 2T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Calcoliamo la forma chiusa procedendo per livelli.

$$\begin{aligned} T(n) &= n + 2T(\lfloor n/2 \rfloor) \\ &= n + 2(n/2) + 4T(\lfloor n/4 \rfloor) \\ &= n + 2(n/2) + 4(n/4) + 8T(\lfloor n/8 \rfloor) \\ &= \dots \\ &= n + 2(n/2) + 4(n/4) + \dots + 2^{\log(n)-1}(n/2^{\log(n)-1}) + 2^{\log(n)} \cdot T(1) \\ &= n \log n = \Theta(n \log n) \end{aligned}$$

Passiamo al procedimento per sostituzione e proviamo a dimostrare che $T(n) = O(n \log n)$*.*

Caso base *Dimostriamo la disequazione per* $T(1)$*:*

$$T(1) = 1 \leq c \cdot 1 \log 1 = 0 \Rightarrow 1 \leq 0$$

Come nell'esempio precedente, pur avendo fatto un tentativo corretto non siamo riusciti a dimostrarlo. Sta volta il problema è nel caso base, tuttavia, non dimentichiamo che stiamo lavorando con notazioni asintotiche, che quindi valgono a partire da un certo valore di n*.*

Proviamo quindi a partire da un valore di $n > 1$*:*

$$\begin{aligned} T(2) &= 2T(\lfloor 2/2 \rfloor) + 2 = 2T(1) + 2 = 4 \leq c \cdot 2 \log 2 \quad \Leftrightarrow \quad c \geq 2 \\ T(3) &= 2T(\lfloor 3/2 \rfloor) + 3 = 2T(1) + 3 = 5 \leq c \cdot 3 \log 3 \quad \Leftrightarrow \quad c \geq \tfrac{5}{3 \log 3} \\ T(4) &= 2T(\lfloor 4/2 \rfloor) + 2 = 2T(\lfloor 2 \rfloor) + 4 \end{aligned}$$

Non serve dimostrare l'ultima disequazione perché è basata su $T(2)$ *che abbiamo già verificato.*

Passo induttivo *Ipotizziamo che* $T(k) \leq c \cdot k \log k \quad \forall\, k < n$ *e dimostriamo la disequazione per* $T(n)$*:*

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2 \cdot c \cdot (\lfloor n/2 \rfloor) \log(\lfloor n/2 \rfloor) + n && \textit{Sostituzione} \\ &\leq 2 \cdot c \cdot (n/2) \log(n/2) + n && \textit{Rimozione intero inferiore} \\ &= c \cdot n(\log n - 1) + n && \textit{Teorema del rapporto} \\ &= c \cdot n \log n - c \cdot n + n \\ &\leq c \cdot n \log n \end{aligned}$$

L'ultima disequazione è vera per:

$$-c \cdot n + n \leq 0 \Leftrightarrow -c \cdot n \leq -n \Leftrightarrow c \geq 1$$

A questo punto vale che $T(n) \leq c \cdot n \log n$ *per:*

- Caso base*:* $\forall\, c \geq 2$, $\forall\, c \geq \frac{5}{3 \log 3}$*;*
- Passo induttivo*:* $\forall\, c \geq 1$

Siccome in tutti e tre i casi c *è in una relazione* $\geq$ *con il secondo termine, è sufficiente porre* $c = \max\left\{1, 2, \frac{5}{3 \log 3}\right\}$*. Inoltre, poiché abbiamo dimostrato il caso base per* $n = 2$ *e* $n = 3$*, prendiamo* $m = 2$*. Ecco provato che* $T(n) = O(n)$*.*

La dimostrazione del limite inferiore non è necessaria poiché alla prima invocazione di T *pagheremo certamente* n*, dunque* $T(n) = \Omega(n)$*.*

Ricapitolando, questo metodo risolutivo si basa sul tentare di intuire la *forma chiusa* delle *funzioni di ricorrenza* e sul dimostrare per induzione, andando a sostituire il tentativo all'interno della funzione, la correttezza dell'intuizione.

3.1.3 Metodo delle ricorrenze comuni

Molte *funzioni di ricorrenza* sono risolvibili velocemente mediante l'applicazione di un qualche teorema. Esistono diversi teoremi che sono specifici per particolari classi di *funzioni di ricorrenza.*

Ricorrenze lineari con partizione bilanciata Per questa classe di *funzioni di ricorrenza* esistono due versioni di uno stesso teorema. Vediamo prima la versione *ridotta*.

Definizione 19 - Teorema delle ricorrenze lineari con partizione bilanciata - Rid.

Siano a e b costanti intere tali che $a \geq 1$ e $b \geq 2$. Siano poi c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia $T(n)$ una funzione di ricorrenza della seguente forma:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ d & n \leq 1 \end{cases}$$

Allora, posto $\alpha = \frac{\log a}{\log b} = \log_b a$ vale:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \alpha > \beta \\ \Theta(n^\alpha \log n) & \alpha = \beta \\ \Theta(n^\beta) & \alpha < \beta \end{cases}$$

NB. Di seguito, nella dimostrazione, ipotizziamo che $n = b^k \Rightarrow k = \log n$ così da semplificare i calcoli.

Dimostrazione. Sia $T(n)$ la seguente *funzione di ricorrenza*:

$$T(n) = \begin{cases} aT(n/b) + cn^\beta & n > 1 \\ d & n \leq 1 \end{cases}$$

Calcoliamone la *forma chiusa* procedendo per *livelli*:

Livello	Dim. input	Costo per chiamata	N. chiamate	Costo livello
0	b^k	$cb^{k\beta}$	1	$cb^{k\beta}$
1	b^{k-1}	$cb^{(k-1)\beta}$	a	$acb^{k\beta}$
2	b^{k-2}	$cb^{(k-2)\beta}$	a^2	$a^2cb^{k\beta}$
...	...	...	...	...
i	b^{k-i}	$cb^{k\beta}$	a^i	$a^icb^{k\beta}$
...	...	...	...	...
$k-1$	b	cb^β	a^{k-1}	$a^{k-1}cb^{k\beta}$
k	1	d	a^k	a^kd

Sommando i costi di tutti i livelli si ottiene la seguente espressione:

$$T(n) = da^k + cb^{k\beta} \sum_{i=0}^{k-1} \frac{a^i}{b^{i\beta}} = da^k + cb^{k\beta} \sum_{i=0}^{k=1} \left(\frac{a}{b^\beta}\right)^i$$

Possiamo osservare che:

$$a^k = a^{\log_b n} = a^{\frac{\log n}{\log b}} = 2^{\log a \frac{\log n}{\log b}} = n^{\frac{\log a}{\log b}} = n^\alpha$$
$$\alpha = \frac{\log a}{\log b} \Rightarrow \alpha \log b = \log a \Rightarrow \log b^\alpha = \log a \Rightarrow a = b^\alpha$$

A questo punto, se poniamo $q = \frac{\alpha}{b^\beta} = \frac{b^\alpha}{b^\beta} = b^{\alpha-\beta}$ possiamo riscrivere $T(n)$ come:

$$T(n) = da^k + cb^{k\beta} \sum_{i=0}^{k-1} \left(\frac{\alpha}{b^\beta}\right)^i = dn^\alpha + cb^{k\beta} \sum_{i=0}^{k-1} q^i$$

Passiamo dunque alla dimostrazione, caso per caso, del teorema.

Caso $\alpha > \beta$ Se $\alpha > \beta$ segue che $q = b^{\alpha-\beta} > 1$, quindi:

$$\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\textstyle\sum_{i=0}^{k-1} q^i \\
&= dn^\alpha + cb^{k\beta}\frac{q^k-1}{q-1} && \text{Serie geometrica finita} \\
&\leq dn^\alpha + cb^{k\beta}\frac{q^k}{q-1} \\
&= dn^\alpha + \frac{cb^{k\beta}\alpha^k}{b^{k\beta}(q-1)} && \text{Sostituzione di } q \\
&= dn^\alpha + \frac{c\alpha^k}{q-1} \\
&= dn^\alpha + \frac{cn^\alpha}{q-1} && a^k = n^\alpha \\
&= n^\alpha\left(d + \frac{c}{q-1}\right)
\end{aligned}$$

Quindi, $T(n) = O(n^\alpha)$ e, per via della componente dn^α, $T(n) = \Omega(n^\alpha)$, dunque $T(n) = \Theta(n^2)$.

Caso $\alpha = \beta$ Se $\alpha = \beta \Rightarrow q = b^{\alpha-\beta} = 1$, quindi:

$$\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\textstyle\sum_{i=0}^{k-1} q^i \\
&= dn^\alpha + cb^{k\alpha}k && \alpha = \beta \wedge q^i = 1^i = 1 \\
&= dn^\alpha + cn^\alpha k && b^\alpha = a \wedge a^k = n^\alpha \\
&= n^\alpha(d + ck) \\
&= n^\alpha(d + c\tfrac{\log n}{\log b}) && k = \log_b n = \tfrac{\log n}{\log b}
\end{aligned}$$

Quindi, $T(n) = \Theta(n^\alpha \log n)$.

Caso $\alpha < \beta$ Se $\alpha < \beta \Rightarrow q = b^{\alpha-\beta} < 1$, quindi:

$$\begin{aligned}
T(n) &= dn^\alpha + cb^{k\beta}\textstyle\sum_{i=0}^{k-1} q^i \\
&= dn^\alpha + cn^{k\beta}\frac{q^k-1}{q-1} && \text{Serie geometrica finita} \\
&= dn^\alpha + cb^{k\beta}\frac{1-q^k}{1-q} && \text{Cambio di segno} \\
&\leq dn^\alpha + cb^{k\beta}\frac{1}{1-q} \\
&= dn^\alpha + \frac{cn^\beta}{1-q} && b^k = n
\end{aligned}$$

Quindi, $T(n) = O(n^\beta)$ e, per il termine non ricorsivo $cb^{k\beta} = cn^\beta$, $T(n) = \Omega(n^\beta)$, dunque $T(n) = \Theta(n^\beta)$. □

Definizione 20 - Teorema delle ricorrenze lineari con partizione bilanciata - Est.

Siano $a \geq 1$, $b > 1$ e $f(n)$ una funzione asintoticamente positiva. Sia poi $T(n)$ una funzione di ricorrenza della seguente forma:

$$T(n) = \begin{cases} aT(n/b) + f(n) & n > 1 \\ d & n \leq 1 \end{cases}$$

Valgono le seguenti casistiche:

1) *Se* $\exists\, \epsilon > 0 : f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ $\Rightarrow$ $T(n) = \Theta\left(n^{\log_b a}\right)$
2) *Se* $f(n) = \Theta\left(n^{\log_b a}\right)$ $\Rightarrow$ $T(n) = \Theta\left(f(n)\log n\right)$
3) *Se* $\exists\, \epsilon > 0 : f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right) \wedge \exists\, c : 0 < c < 1, \exists\, m \geq 0 : af(n/b) \leq cf(n) \quad \forall\, n \geq m$ $\Rightarrow$ $T(n) = \Theta(f(n))$

NB. Nonostante non siano presenti nelle definizioni, i teoremi appena visti valgono anche per *funzioni di ricorrenza* espresse usando gli operatori di *intero-inferiore* e *intero-superiore*.

Esempio 11 - Applicazione della forma estesa del teorema.

Sia $T(n)$ la seguente funzione di ricorrenza:

$$T(n) = \begin{cases} 9T(n/3) + n\log n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Trovarne la forma chiusa.

Proviamo ad applicare il teorema appena visto. Poiché il fattore non ricorsivo, cioè $n\log n$, non è una semplice potenza di n e, contemporaneamente, è una funzione crescente, quindi asintoticamente positiva, applichiamo la versione estesa del teorema.

Ricorrenza	a	b	$\log_b a$	**Caso**	**Funzione**
$9T(n/3) + n\log n$	9	3	2	(1)	$\Theta(n^2)$

In questo esempio si applica la prima casistica del teorema perché $f(n) = n\log n$ è certamente un $O(n^2)$, ma cresce comunque meno rispetto a n^2 e più di n, quindi se prendiamo $\epsilon < 1$, vale:

$$f(n) = n\log n = O(n^{\log_b(a)-\epsilon}) = O(n^{2-\epsilon})$$

E quindi, per il primo caso del Teorema delle ricorrenze lineari con partizione bilanciata esteso, $T(n) = \Theta(n^2)$.

Esempio 12 - Applicazione della forma ridotta del teorema.

Sia $T(n)$ la seguente funzione di ricorrenza:

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

Trovare la forma chiusa.

Questa volta possiamo applicare la versione ridotta del teorema.

Ricorrenza	a	b	$\log_b a$	β	**Caso**	**Funzione**
$T(2n/3) + 1$	1	3/2	0	0	(2)	$\Theta(\log n)$

In questo caso $\alpha = \log_b a = \log_{\frac{3}{2}} 1 = 0$ e poiché anche $\beta = 0$ si applica la seconda casistica, cioè $T(n) = \Theta(\log n)$.

In tutti i casi in cui è applicabile la versione ridotta del teorema, si può usare anche quella estesa. Con questa funzione, ad esempio, avremmo potuto notare che:

$$f(n) = 1 = \Theta(1) = \Theta(n^0) = \Theta(n^{\log_b a})$$

E quindi, per il secondo caso, avremmo ottenuto:

$$T(n) = \Theta(f(n)\log n) = \Theta(1\log n) = \Theta(\log n)$$

Esempio 13 - Applicazione della forma estesa del teorema.

Sia $T(n)$ la seguente funzione di ricorrenza:

$$T(n) = \begin{cases} 3T(n/4) + n\log n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Trovare la forma chiusa.

$f(n) = n\log n$, quindi applico il teorema delle ricorrenze lineari con partizione bilanciata nella sua forma estesa.

Ricorrenza	a	b	$\log_b a$	Caso	Funzione
$3T(n/4) + n\log n$	3	4	≈ 0.79	(3)	$\Theta(f(n))$

Si tratta del terzo caso perché $f(n) = n\log n = \Omega(n) = \Omega(n^{\log_b(a)-\epsilon})$ con $\epsilon < 1 - \log_b a = 1 - \log_4 3 \approx 0.208$.

Tuttavia, dobbiamo anche dimostrare che $\exists\, c : 0 < c < 1$ e $\exists\, m \geq 0$ tali che:

$$af(n/b) \leq cf(n) \quad \forall\, n \geq m$$

Vale quanto segue:

$$\begin{array}{rcll} af(n/b) & = & a(n/b)\log(n/b) & \textit{Sostituzione} \\ & = & 3(n/4\log(n/4)) & \\ & = & \frac{3}{4}n(\log(n) - \log(4)) & \textit{Teorema del rapporto} \\ & \leq & \frac{3}{4}n\log n & \\ & \leq & cn\log n & \end{array}$$

L'ultima disequazione è vera per $c = \frac{3}{4}$ e per qualsiasi m, quindi è dimostrato che $T(n) = \Theta(f(n)) = \Theta(n\log n)$.

Esempio 14 - Esempio di inapplicabilità del teorema.

Sia $T(n)$ la seguente funzione di ricorrenza:

$$T(n) = \begin{cases} 2T(n/2) + n\log n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

Trovare la forma chiusa.

In questo caso vale $\log_b a = \log_2 2 = 1$ e $f(n) = n\log n = \Omega(n) = \Theta(n\log n) = O(n^2)$, ma poiché:

- $n\log n \neq O(n^{1-\epsilon}) \quad \forall\, \epsilon > 0$
- $n\log n \neq \Theta(n^1)$
- $n\log n \neq \Omega(n^{1+\epsilon}) \quad \forall\, \epsilon > 0$

non è possibile applicare nessuna casistica del teorema, dunque è necessario usare altre tecniche.

Ricorrenze lineari di ordine costante

Definizione 21 - Teorema delle ricorrenze lineari di ordine costante.

Siano $a_1, a_2, \ldots, a_h$ costanti intere non negative con h costante e positiva. Siano poi c e β costanti reali tali che $c > 0$ e $\beta \geq 0$. Sia infine $T(n)$ definita dalla seguente funzione di ricorrenza:

$$T(n) = \begin{cases} \sum_{i=1}^{h} (a_i T(n-i)) + cn^{\beta} & n > m \\ \Theta(1) & n \leq m \leq h \end{cases}$$

Allora, posto $a = \sum_{i=1}^{h} a_i$ vale:

1) *Se* $a = 1 \Rightarrow T(n) = \Theta(n^{\beta+1})$
2) *Se* $a \geq 2 \Rightarrow T(n) = \Theta(a^n n^{\beta})$

Esempio 15 - Applicazione del teorema delle ricorrenze lineari di ordine costante.

Sia $T(n)$ la seguente funzione di ricorrenza:

$$T(n) = \begin{cases} T(n-10) + n^2 & n > 1 \\ \Theta(1) & n \leq 1 \end{cases}$$

Trovare la forma chiusa.

Proviamo ad applicare il teorema delle ricorrenze lineari di ordine costante.

Ricorrenza	$\boldsymbol{a}$	$\boldsymbol{\beta}$	**Caso**	**Funzione**
$T(n-10) + n^2$	1	2	(1)	$\Theta(n^{\beta+1})$

In questo esempio vale il caso 1 perché il coefficiente di $T(n-10)$ è 1, quindi, per il teorema, risulta $T(n) = \Theta(n^{\beta+1}) = \Theta(n^3)$.

Esempio 16 - Applicazione del teorema delle ricorrenze lineari di ordine costante.

Sia $T(n)$ la seguente funzione di ricorrenza:

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

Trovare la forma chiusa.

Applichiamo di nuovo il teorema appena enunciato.

Ricorrenza	$\boldsymbol{a}$	$\boldsymbol{\beta}$	**Caso**	**Funzione**
$T(n-2) + T(n-1) + 1$	2	0	(2)	$\Theta(a^n n^{\beta})$

Qui si applica il caso 2 del teorema e poiché $\beta = 0$ vale $T(n) = \Theta(a^n n^{\beta}) = \Theta(2^n)$, cioè questa funzione ha costo esponenziale.

Problema 2 - Funzioni di ricorrenza parametriche.

Siano $T(n)$ e $T'(n)$ le seguenti funzioni di ricorrenza:

$$T(n) = \begin{cases} 7T(n/2) + n^2 & n > 1 \\ 1 & n \leq 1 \end{cases} \quad \textit{Algoritmo } A^{3}$$

$$T'(n) = \begin{cases} aT'(n/4) + n^2 & n > 1 \\ 1 & n \leq 1 \end{cases} \quad \textit{Algoritmo } A'$$

Trovare il massimo valore intero di a che renda A' asintoticamente più veloce di A.

Iniziamo calcolando la complessità di A e, in particolare, applichiamo il Teorema delle ricorrenze lineari con partizione bilanciata - Rid.

Ricorrenza	$\boldsymbol{a}$	$\boldsymbol{b}$	$\boldsymbol{\log_b a}$	$\boldsymbol{\beta}$	**Caso**	**Funzione**
$7T(n/2) + n^2$	7	2	≈ 2.81	2	(1)	$\Theta(n^{\log_2 7})$

Ora, poiché in $T'(n)$ compare $T'(n/4)$, trasformo $\log_2 7$ in un qualche $\log_4$:

$$\begin{aligned} \log_2 7 &= \frac{\log_4 7}{\log_4 2} && \textit{Cambio di base} \\ &= \frac{\log_4 7}{1/2} \\ &= 2\log_4 7 \\ &= \log_4 7^2 && \textit{Teorema della potenza} \\ &= \log_4 49 \end{aligned}$$

A questo punto studio $T'(n)$ in base al variare di a:

$$\begin{array}{llllll} a < 16 & \Rightarrow & \alpha = \log_4 a < 2 & \Rightarrow & \alpha < \beta & \Rightarrow & T'(n) = \Theta(n^2) = O(T(n)) \\ a = 16 & \Rightarrow & \alpha = \log_4 a = 2 & \Rightarrow & \alpha = \beta & \Rightarrow & T'(n) = \Theta(n^2 \log n) = O(T(n)) \\ 16 < a \leq 49 & \Rightarrow & \alpha = \log_4 a > 2 & \Rightarrow & \alpha > \beta & \Rightarrow & T'(n) = \Theta(n^\alpha) = O(T(n)) \\ a > 49 & \Rightarrow & \alpha = \log_4 a > 2 & \Rightarrow & \alpha > \beta & \Rightarrow & T'(n) = \Theta(n^\alpha) = \Omega(T(n)) \end{array}$$

Nel terzo caso $2 < a \leq \log_4 49$ quindi $T'(n)$ cresce al più come $T(n)$, mentre per $a > 49$, $\alpha > \log_4 49$ e quindi $T'(n)$ cresce più di $T(n)$.

Quindi, il massimo valore intero di a che rende A' asintoticamente più veloce di A è 49.

[3] Funzione di ricorrenza dell'algoritmo di *Strassen*

Capitolo Nr.4

Strutture dati

4.1 Introduzione

Iniziamo con delle definizioni:

Definizione 22 - Dato.

In un linguaggio di programmazione, un dato è un valore che una variabile può assumere.

Definizione 23 - Tipo di dato astratto.

Un tipo di dato astratto è un modello matematica dato da una collezione di valori e un insieme di operazioni ammesse su questi valori.

Definizione 24 - Tipo di dato primitivo.

Un tipo di dato fornito direttamente da un linguaggio è detto essere primitivo.

Definizione 25 - Specifica e implementazione di un tipo di dato.

Per ogni tipo di dato astratto si definiscono due livelli:

- Specifica*: costituisce l'interfaccia di utilizzo del tipo di dato e ne nasconde i dettagli implementativi;*
- Implementazione*: è la realizzazione vera e propria del tipo di dato;*

I seguenti sono esempi di *specifica* e *implementazione* di un *tipo di dato*:

Specifica	**Implementazione**
Numeri reali	IEEE754
Pile	Pile basate su vettori Pile basate su puntatori
Code	Code basate su vettori circolari Code basate su puntatori

Definizione 26 - Struttura di dati.

Una struttura di dati è una collezione di dati caratterizzata dalla struttura della collezione, piuttosto che dal tipo dei dati contenuti.

Una *struttura di dati* è caratterizzata da un insieme di operatori che consentono di manipolarne la struttura e da un modo sistematico di organizzare i dati in essa contenuti.

Le *strutture di dati* possono essere categorizzate sulla base di tre parametri:

- *Lineari/Non lineari*: se è presente o meno una sequenza;
- *Dinamiche/Statiche*: se è possibile modificare o meno la dimensione della struttura dopo averla creata;
- *Omogenee/Disomogenee*: se i dati contenuti sono tutti dello stesso tipo o di tipi diversi;

4.2 Strutture di dati astratte

4.2.1 Sequenze

Definizione 27 - Sequenza.

Una sequenza è una struttura dati dinamica e lineare rappresentante una sequenza ordinata di valori che possono comparire anche più di una volta.

NB. L'ordine dei valori all'interno di una *sequenza* è importante!

Operazioni ammesse Su una *sequenza* sono ammesse le seguenti operazioni:

- Data una posizione è possibile aggiungere o togliere elementi, cioè se $s = s_1, s_2, \ldots, s_n$ è la *sequenza*, l'elemento s_i è in posizione pos_i, inoltre, esistono posizioni fittizie quali pos_0 e pos_{n+1} che rappresentano la posizione del primo elemento e dell'elemento successivo a quello in pos_n;
- Accesso diretto alla testa o alla coda;
- Accesso sequenziale alle altre posizioni;

Specifica

Frammento 8 - Sequenza.

```
% Restituisce true se la sequenza è vuota
boolean isEmpty()
% Restituisce true se p = pos_0 o se p = pos_{n+1}
boolean finished(POS p)
% Restituisce la posizione del primo elemento
POS head()
% Restituisce la posizione dell'ultimo elemento
POS tail()
```

```
% Restituisce la posizione dell'elemento che segue p
POS next(POS p)
% Restituisce la posizione dell'elemento che precede p
POS prev(POS p)
% Inserisce l'elemento v di tipo ITEM nella posizione p e restituisce
% la posizione del nuovo elemento, che diviene il predecessore di p
POS insert(POS p, ITEM v)
% Rimuove l'elemento contenuto nella posizione p e restituisce la posizione
% del successore di p, che diviene il successore del predecessore di p
POS remove(POS p)
% Legge l'elemento di tipo ITEM contenuto nella posizione p
ITEM read(POS p)
% Scrive l'elemento v di tipo ITEM nella posizione p
write(POS p, ITEM v)
```

4.2.2 Insiemi

Definizione 28 - Insieme.

Un insieme è una struttura dati dinamica e non lineare che memorizza una collezione non ordinata di valori non ripetuti.

NB. L'ordinamento fra elementi è dato dall'eventuale relazione d'ordine definita sul tipo degli elementi stessi.

Operazioni ammesse Su un *insieme* sono ammessi diversi tipi di operazioni:

- *Operazioni di base*:
  - Inserimento;
  - Rimozione;
  - Test di appartenenza;
- *Operazioni di ordinamento*: estrazione valore massimo/minimo;
- *Operazioni insiemistiche*:
  - Unione;
  - Intersezione;
  - Differenza;
- *Iterazione sugli elementi*: esecuzione di operazioni su tutti gli elementi dell'*insieme* (e.g. `foreach (x ∈ S) do ...`);

Specifica

Frammento 9 - Insieme.

```
% Restituisce la cardinalità dell'insieme
int size()
% Restituisce true se x è contenuto nell'insieme
boolean contains(ITEM x)
% Inserisce x nell'insieme, se non è giù presente
insert(ITEM x)
% Rimuove x dall'insieme, se è presente
remove(ITEM x)
% Restituisce un nuovo insieme che è l'unione di A e B
Set union(Set A, Set B)
% Restituisce un nuovo insieme che è l'intersezione di A e B
Set intersection(Set A, Set B)
% Restituisce un nuovo insieme che è la differenza di A e B
Set difference(Set A, Set B)
```

4.2.3 Dizionari

Definizione 29 - Dizionario.

Un dizionario è una struttura dati dinamica che rappresenta il concetto matematico di relazione univoca, o associazione chiave-valore, $R : D \to C$, dove D è un l'insieme dominio i cui elementi sono detti chiavi e C è un insieme codominio di elementi detti valori.

Operazioni ammesse Su un *dizionario* sono ammesse le seguenti operazioni:

- Data una chiave è possibile accedere al valore associato o a `nil` se la chiave non è associata a nulla;
- Creazione di una nuova associazione chiave-valore, eventualmente sostituendo il precedente valore associato a quella chiave;
- Eliminazione di un'associazione chiave-valore;

Specifica

Frammento 10 - Dizionario.

```
% Restituisce il valore associato alla chiave k se presente, nil altrimenti
ITEM lookup(ITEM k)
% Associa il valore v alla chiave k
insert(ITEM k, ITEM v)
% Rimuove l'associazione della chiave k
remove(ITEM k)
```

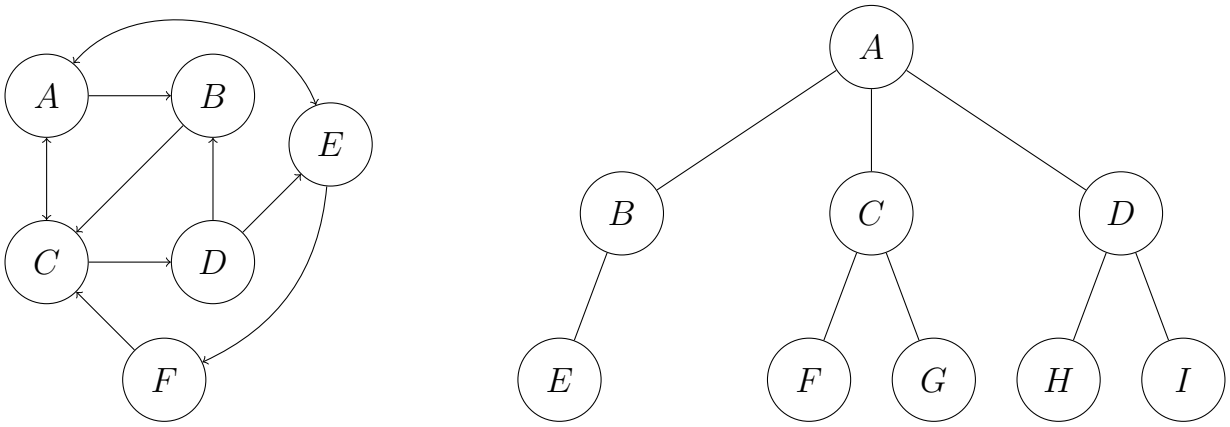
4.2.4 Grafi e alberi

Definizione 30 - Grafo.

Un grafo è una struttura composta da un insieme di elementi detti nodi o vertici e un insieme di coppie di nodi, ordinate o meno, dette archi.

Definizione 31 - Albero ordinato.

Un albero ordinato è una struttura dati dinamica composta da un insieme finito di elementi detti nodi. Uno di questi nodi è detto radice, tutti gli altri sono partizionati in insiemi ordinati e disgiunti che sono anch'essi alberi ordinati.



(a) *Grafo* con archi orientati (b) *Albero ordinato*

Fig. 4.1: *Grafo* VS *Albero ordinato*

Operazioni ammesse Oltre a inserimento e rimozione, le operazioni ammesse su *grafi* e *alberi* ruotano attorno alla possibilità di accedere a tutti gli elementi delle strutture secondo diverse *visite*[1].

4.2.5 Criticità nell'implementazione di strutture dati astratte

Quando si implementa una *struttura di dati astratta* può essere naturale prediligere una particolare *struttura di dati elementare* piuttosto che un'altra. Ad esempio, viene naturale implementare una *sequenza* usando una *lista*, o un *albero astratto* come *albero di puntatori*. Tuttavia, esistono possibilità meno scontate, come l'utilizzo di un vettore di booleani per l'implementazione di un *insieme*, o un *albero* implementato come *vettore dei padri*.

La *struttura di dati elementare* scelta per l'implementazione di una *struttura dati astratta* si ripercuote sull'efficienza delle singole operazioni. Per esempio, un *dizionario* implementato come *tabella hash* permette di avere *complessità* $O(1)$ nella funzione `lookup` e $O(n)$ nella ricerca del minimo. Invece, lo stesso *dizionario* implementato come *albero* porta la *complessità* della `lookup` a $O(\log n)$, ma riduce a $O(1)$ quella per la ricerca del minimo.

[1] Tutte le *visite* saranno discusse nel dettaglio nel prossimo capitolo

4.3 Strutture di dati elementari

Vediamo ora le possibili implementazioni delle più comuni *strutture dati elementari*: *liste*, *pile* e *code*.

4.3.1 Liste

Definizione 32 - Lista.

Una lista è una sequenza di nodi contenenti dati arbitrati e uno o due puntatori all'elemento successivo e/o precedente.

NB. È importante notare che nodi contigui nella *lista* non lo sono necessariamente anche in memoria.

NB. In una *lista* tutte le operazioni hanno costo $O(1)$.

Diverse implementazioni di una *lista* possono essere categorizzate sulla base di tre parametri:

- *Monodirezionali/Bidirezionali*: sono *bidirezionali* le implementazioni in cui ogni nodo contiene due puntatori: uno all'elemento precedente, l'altro al successivo;
- *Con sentinelle/Senza sentinella*: sono *con sentinella* tutte le implementazioni in cui una *lista* vuota ha un elemento;
- *Circolare/Non circolare*: sono *circolari* le implementazioni in cui l'ultimo elemento ha come successivo il primo, o viceversa;

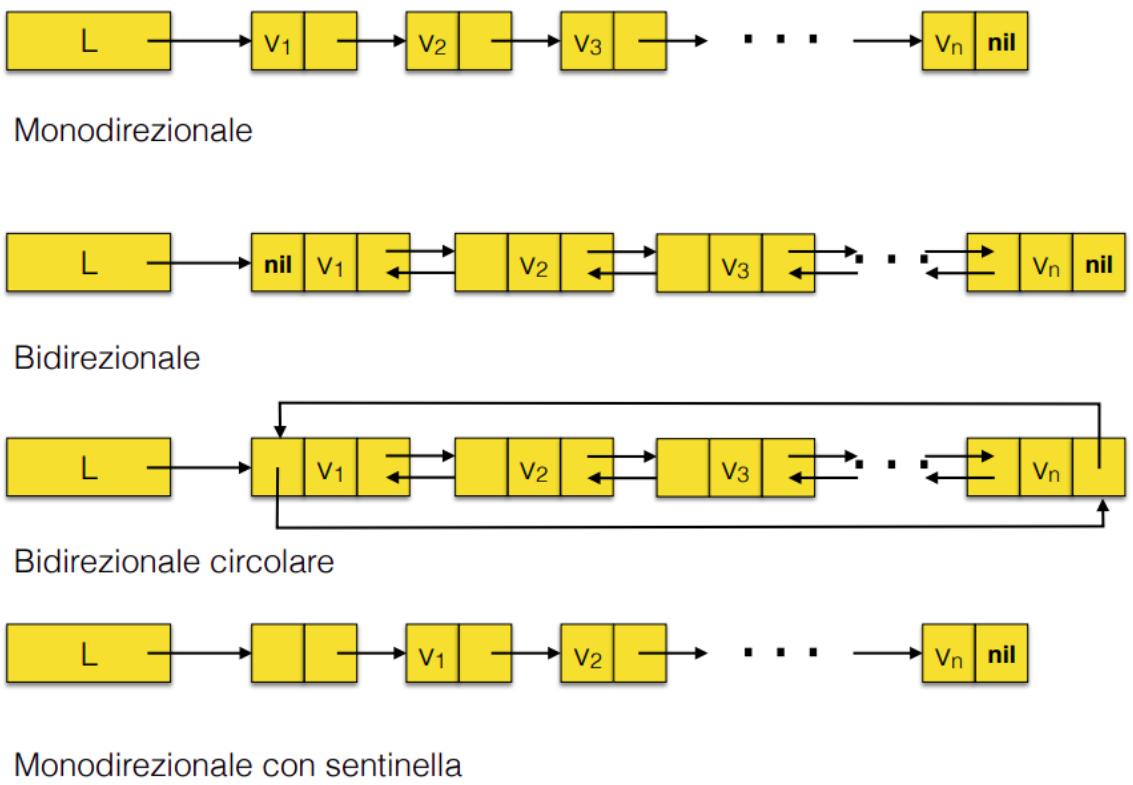


Fig. 4.2: Diverse implementazioni di una *lista*

Implementazione del tipo di dato POS

Frammento 11 - POS.

```
POS succ                                    % Elemento successivo
POS pred                                    % Elemento precedente
ITEM value                                  % Valore

POS Pos(ITEM v)
   POS p = new POS
   p.succ = nil
   p.pred = nil
   p.value = v
```

Implementazione di una lista bidirezionale con sentinella

Frammento 12 - Lista bidirezionale con sentinella.

```
LIST pred                % Predecessore
LIST succ                  % Successore
ITEM value                     % Valore

LIST List()
   LIST t = new LIST
   t.pred = t
   t.succ = t
   return t

boolean isEmpty()
   return (pred == succ == this)

POS head()
   return succ

POS tail()
   return pred

POS next(POS p)
   return p.succ

POS prev(POS p)
   return p.pred
```

```
% Restituisce true se p è la
% posizione sentinella
boolean finished(POS p)
   return (p == this)

ITEM read(POS p)
   return p.value

write(POS p, ITEM v)
   p.value = v

POS insert(POS p, ITEM v)
   POS t = Pos(v)
   t.pred = p.pred
   p.pred.succ = t
   t.succ = p
   p.pred = t
   return t

POS remove(POS p)
   p.pred.succ = p.succ
   p.succ.pred = p.pred
   POS t = p.succ
   delete p
   return t
```

NB. I tipi `POS` e `LIST` sono equivalenti.

NB. Alla posizione *sentinella* non è stato assegnato alcun valore in quanto serve soltanto a semplificare le operazioni di inserimento e rimozione.

4.3.2 Pile

Definizione 33 - Pila.

Una pila è una struttura dati dinamica e lineare, nella quale l'accesso agli elementi è definito in base all'ordine in cui sono stati inseriti. In particolare, è possibile accedere direttamente soltanto all'ultimo elemento inserito.

NB. Le *pile* sono basate sull'approccio *LIFO* (*Last In, First Out*).

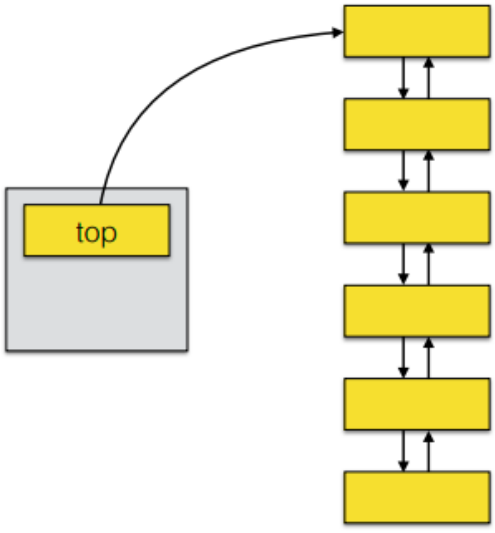
Specifica

Frammento 13 - Pila.

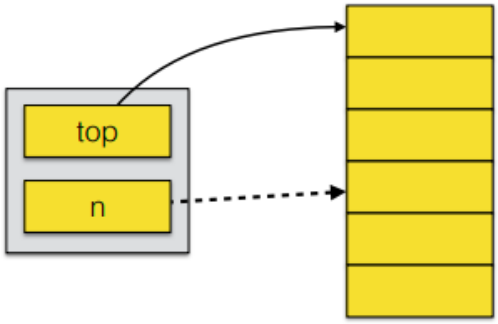
```
% Restituisce true se la pila è vuota
boolean isEmpty()
% Inserisce v in cima alla pila
push(ITEM v)
% Rimuove l'elemento in cima alla pila e lo restituisce
ITEM pop()
% Legge l'elemento in cima alla pila
ITEM top()
```

NB. Nel gergo delle *pile*, l'elemento "in cima" è l'ultimo elemento inserito, mentre quello "in fondo", il primo.

Una *pila* può essere implementata come *vettore*, quindi di dimensione limitata, o come *lista bidirezionale* con puntatore all'elemento in testa.



(a) *Pila* implementata come *lista*



(b) *Pila* implementata come *vettore*

Fig. 4.3: Diverse implementazioni di una *pila*

Implementazione di una pila basata su vettore

Frammento 14 - Pila basata su vettore.

```
ITEM[] V                    % Elementi
int cur                % Posizione cursore
int max_dim           % Dimensione massima

STACK Stack(int dim)
   STACK t = new STACK
   t.V = new int[1...dim]
   t.V = new int[1...dim]
   t.cur = 0
   t.max_dim = dim
   return t

ITEM top()
   precondition:(cur > 0)
   return V[cur]

boolean isEmpty()
   return (cur == 0)

ITEM pop()
   precondition:(cur > 0)
   ITEM t = V[cur]
   cur = cur - 1
   return t

push(ITEM v)
   precondition:(cur < max_dim)
   cur = cur + 1
   V[cur] = v
```

NB. In una *pila* implementata come *lista* non è necessario specificare una dimensione massima.

4.3.3 Code

Definizione 34 - Coda.

Una coda è una struttura dati dinamica e lineare, nella quale l'accesso agli elementi è definito in base all'ordine in cui sono stati inseriti. In particolare, è possibile accedere direttamente soltanto al primo elemento inserito.

NB. Le *code* sono basate sull'approccio *FIFO* (*First In, First Out*).

Specifica

Frammento 15 - Coda.

```
% Restituisce true se la coda è vuota
boolean isEmpty()
% Inserisce v in fondo alla coda
enqueue(ITEM v)
% Estrae l'elemento in testa alla coda e lo restituisce al chiamante
ITEM dequeue()
% Legge l'elemento in testa alla coda
ITEM top()
```

NB. Nel gergo delle *code* l'elemento "in testa" è il primo inserito, mentre quello "in coda", l'ultimo.

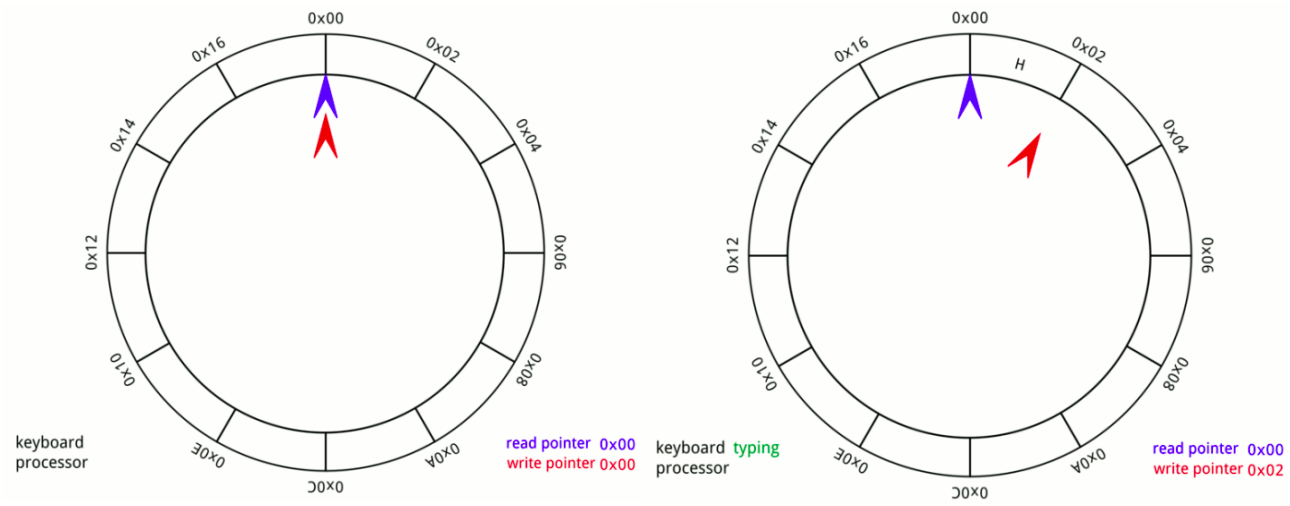
Una *coda* può essere implementata come *vettore circolare*, quindi di dimensione limitata, o come *lista monodirezionale* con puntatore *head* per l'estrazione e *tail* per l'inserimento.

Implementazione di una coda basata su vettore circolare In un *vettore circolare* gli indici possono essere gestiti efficientemente facendo uso dell'operazione di *modulo*. Bisogna tuttavia fare attenzione alla gestione dei problemi di *overflow*, provocati, non dall'accesso ad aree di memoria esterne alla struttura (usando correttamente il *modulo* ciò è impossibile), ma dalla scrittura su valori non ancora letti. Questo problema si propone quando il *vettore* è pieno.

Vediamo come questo problema può essere evitato.

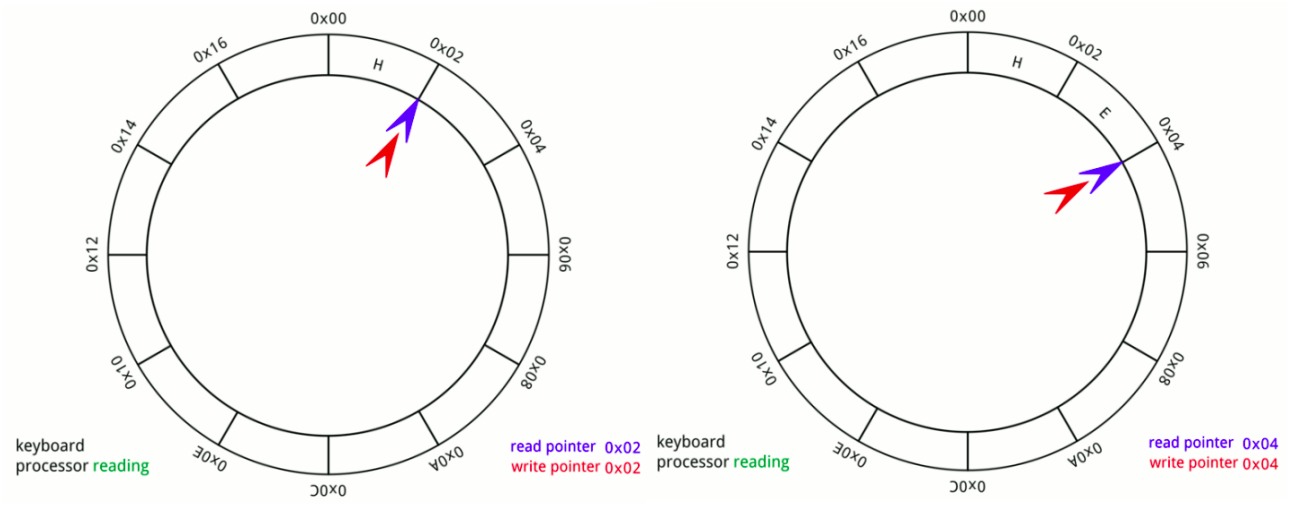
Utilizzando due puntatori, *read pointer* e *write pointer*, che sono, rispettivamente, il puntatore alla *testa* e alla *coda* della struttura, si può fare in modo che non sia possibile scrivere nuovi valori quando il *write pointer* punta alla cella immediatamente precedente a quella puntata dal *read pointer*.

Vediamo nel dettaglio graficamente:



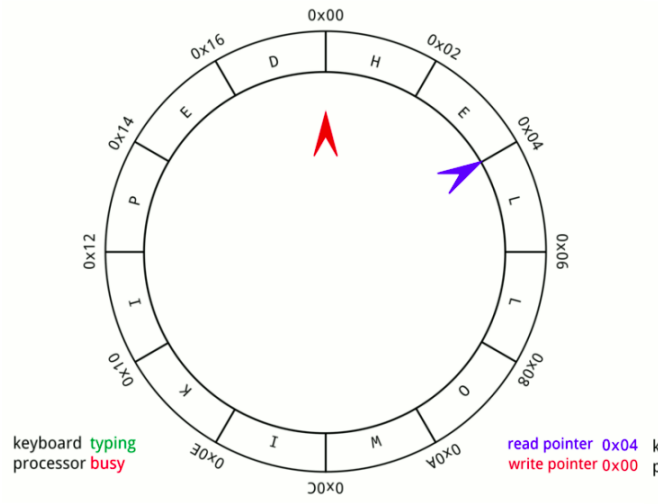
(a) Prima scrittura

(b) Seconda scrittura

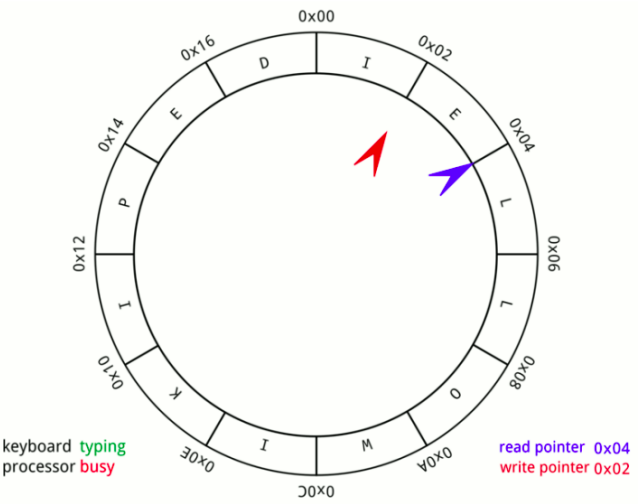


(c) Prima lettura

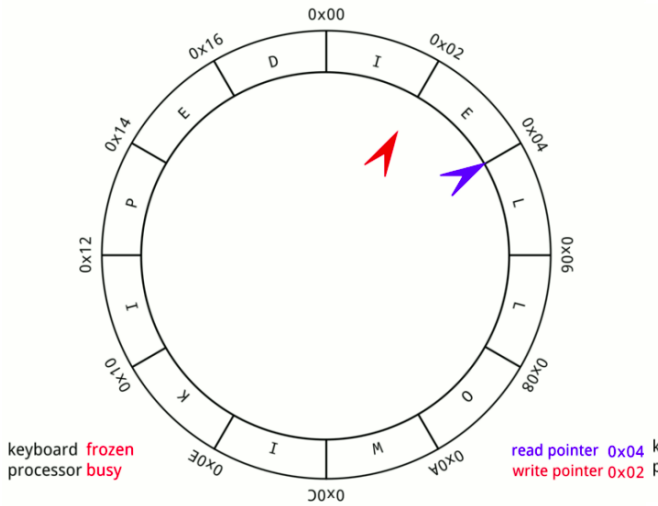
(d) Seconda lettura



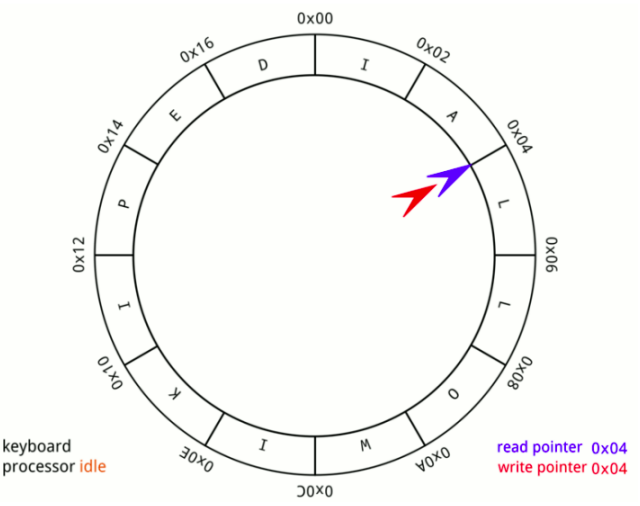
(e) Ultima scrittura su cella libera



(f) Prima scrittura su cella già letta



(g) Blocco delle scritture



(h) Lettura di tutte le celle

Fig. 4.3: Esempio di esecuzione di una *coda* implementata come *vettore circolare*

Nelle immagini di cui sopra si vede molto bene come, dopo ogni operazione di lettura e scrittura, i rispettivi puntatori avanzino fino alla cella successiva. Fino a quando ci sono celle libere i due puntatori avanzano in modo indipendente, ma quando queste finiscono, le successive scritture vanno a sovrascrivere i valori precedentemente inseriti e che sono stati già letti.

Una volta che il *write pointer* si trova sulla cella che precede quella indicata dal *read pointer*, la *coda* è piena e quindi non è possibile inserire nuovi elementi fino a quando non ne vengono letti alcuni. Quando invece il *read pointer* va a sovrapporsi al *write pointer* significa che la *coda* è stata svuotata e di conseguenza vengono bloccate le letture.

Frammento 16 - Coda basata su vettore circolare.

```
ITEM[] V                        % Elementi
int cur_dim          % Dimensione attuale
int head               % Testa della coda
int max_dim          % Dimensione massima

QUEUE Queue(int dim)
   QUEUE t = new QUEUE
   t.V = new int[dim]
   t.max_dim = dim
   t.head = 0
   t.cur_dim = 0
   return t

ITEM top()
   precondition:(cur_dim > 0)
   return V[head]

boolean isEmpty()
   return (cur_dim == 0)

ITEM dequeue()
   precondition:(cur_dim > 0)
   ITEM t = V[head]
   head = (head + 1) mod max_dim
   cur_dim = cur_dim - 1
   return t

enqueue(ITEM v)
   precondition:(cur_dim < max_dim)
   V[(head + cur_dim) mod max_dim] =
   v
   cur_dim = cur_dim + 1
```

NB. In una *coda* implementata come *lista* non è necessario specificare una dimensione massima.

Capitolo Nr.5

Alberi binari e alberi generici

5.1 Introduzione

Iniziamo con una definizione:

Definizione 35 - Albero radicato.

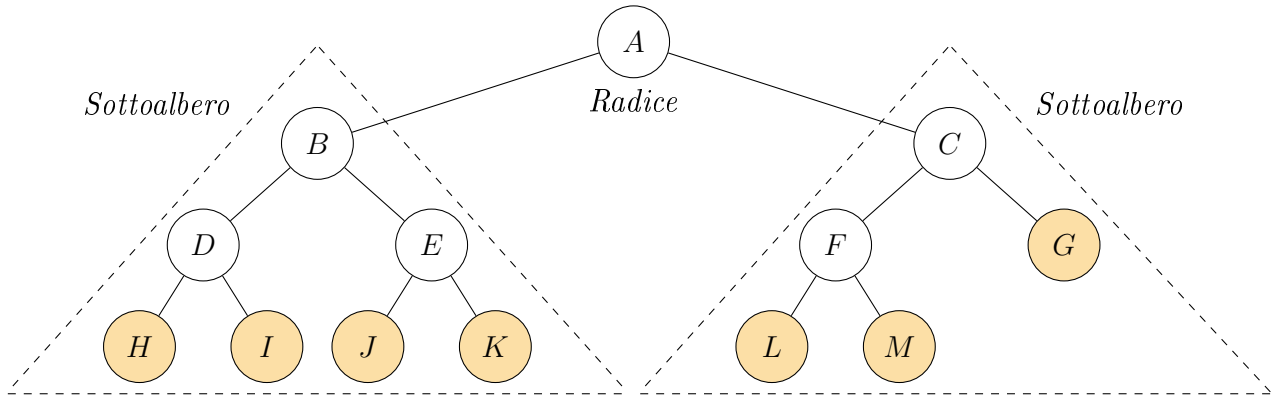
Un albero radicato consiste di un insieme di nodi e di archi orientati che connettono coppie di nodi con le seguenti proprietà:

- *Un nodo dell'albero è designato come nodo radice;*
- *Ogni nodo n, a parte la radice, ha esattamente un arco entrante;*
- *Per ogni nodo esiste un unico cammino che parte dalla radice e raggiunge quel nodo;*
- *L'albero è connesso;*

Definizione 36 - Albero radicato (definizione ricorsiva).

Un albero radicato è definito come un insieme vuoto, oppure un nodo radice e zero o più sottoalberi, ognuno dei quali è un albero radice. La radice è connessa alla radice di ogni sottoalbero con un arco orientato.

5.1.1 Terminologia



Partendo dallo schema di cui sopra possiamo definire la seguente terminologia:

- A è la *radice*;
- B e C sono *radici* dei *sottoalberi*;
- D ed E sono *fratelli*;
- D ed E sono *figli* di B;
- B è il *padre* di D ed E;
- I *nodi* gialli sono *foglie*;
- Gli altri *nodi* sono *nodi interni*;

Per ogni *albero* possiamo poi definire i seguenti parametri:

Definizione 37 - Profondità di un nodo (depth).

È definita profondità di un nodo la lunghezza del cammino semplice dalla radice al nodo. La lunghezza è misurata in numero di archi attraversati.

Definizione 38 - Livello (level).

È definito livello l'insieme dei nodi alla stessa profondità

Definizione 39 - Altezza di un albero (height).

È definita altezza di un albero la profondità massima della sue foglie.

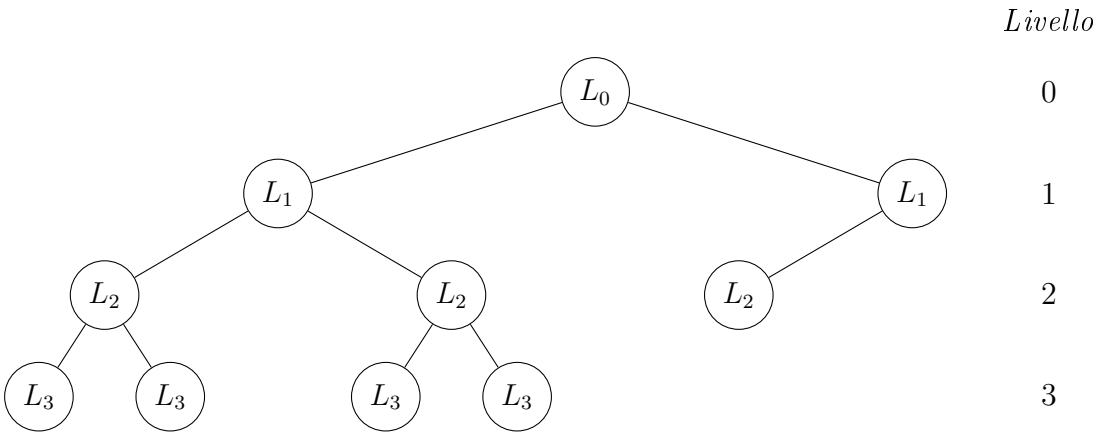


Fig. 5.1: *Albero* di *altezza* 3

5.2 Alberi binari

Definizione 40 - Albero binario.

Un albero binario è un albero radicato in cui ogni nodo ha al più due figli, identificati come figlio sinistro e figlio destro.

NB. Due *alberi* T e U che hanno gli stessi *nodi*, gli stessi *figli* per ogni *nodo* e la stessa *radice*, sono distinti qualora un *nodo* u sia designato come *figlio sinistro* di v in T e come *figlio destro* di v in U.

5.2.1 Specifica

Frammento 17 - Albero binario.

```
% Costituisce un nuovo nodo, contenente v, senza figli o genitori
Tree(ITEM v)
% Legge il valore memorizzato nel nodo
ITEM read()
% Modifica il valore memorizzato nel nodo
write(ITEM v)
% Restituisce il padre, oppure nil se questo è il nodo radice
TREE parent()
% Restituisce il figlio sinistro di questo nodo, oppure nil se è assente
TREE left()
% Restituisce il figlio destro di questo nodo, oppure nil se è assente
TREE right()
% Inserisce il sottoalbero radicato t come figlio sinistro di questo nodo
insertLeft(TREE t)
% Inserisce il sottoalbero radicato t come figlio destro di questo nodo
insertRight(TREE t)
% Distrugge ricorsivamente il figlio sinistro di questo nodo
deleteLeft()
% Distrugge ricorsivamente il figlio destro di questo nodo
deleteRight()
```

5.2.2 Memorizzazione di un albero binario

In memoria, per ogni *nodo*, memorizziamo un puntatore al *nodo padre* (`P`), che sarà `nil` nel caso della *radice*, e altri due puntatori, contenenti, rispettivamente, il riferimento al *figlio sinistro* (`L`) e *destro* (`R`) di quel *nodo*. Nel caso di *nodi foglia*, quei puntatori saranno entrambi `nil`.

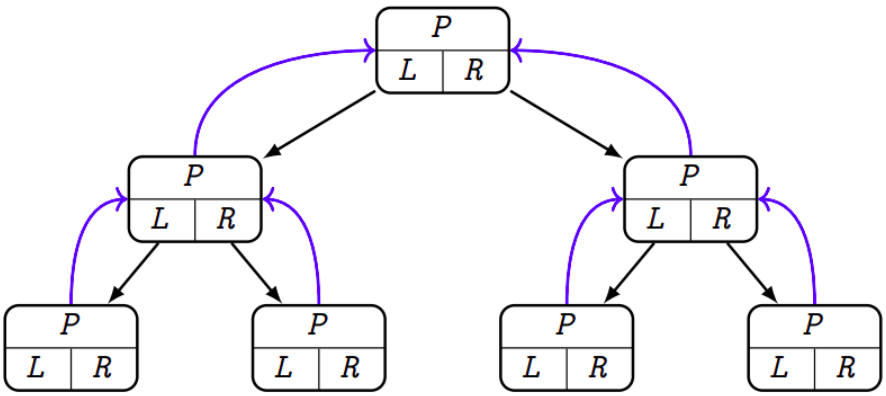


Fig. 5.2: Schema di memorizzazione di un *albero binario*

5.2.3 Implementazione di un albero binario

Frammento 18 - Implementazione di un albero binario.

```
Tree(ITEM v)
   TREE t = new TREE
   t.parent = nil
   t.left = nil
   t.right = nil
   t.value = v
   return t

ITEM read()
   return value

write(ITEM v)
   value = v

TREE parent()
   return parent

TREE left()
   return left

TREE right()
   return right
```

```
% Ipotizziamo che sia possibile
% inserire un figlio solo se non
% ne esiste già uno
insertLeft(TREE t)
   if (left == nil) then
      t.parent = this
      left = t

insertRight(TREE t)
   if (right == nil) then
      t.parent = this
      right = t

deleteLeft()
   if (left ≠ nil) then
      left.deleteLeft()
      left.deleteRight()
      delete left
      left = nil

deleteRight()
   if (right ≠ nil) then
      right.deleteLeft()
      right.deleteRight()
      delete right
      right = nil
```

5.2.4 Visite di un albero binario

Definizione 41 - Visita di un albero.

Una visita è una strategia per visitare tutti i nodi dell'albero.

Definizione 42 - Visita in profondità (depth first search).

Una visita in profondità è un tipo di visita nella quale vengono visitati ricorsivamente tutti i sottoalberi.

NB. Questo tipo di *visita* richiede l'utilizzo di una *pila*.

Per la *visita in profondità* esistono tre varianti che si differenziano per il momento in cui viene utilizzato il valore di un *nodo*:

- *Visita in Pre-order*: il valore del *nodo* viene usato prima di visitare i *sottoalberi*;

- *Visita in In-order*: il valore del *nodo* viene usato dopo aver visitato il *sottoalbero di sinistra* e prima di visitare quello di *destra*;
- *Visita in Post-order*: il valore del *nodo* viene usato dopo aver visitato i *sottoalberi*;

Definizione 43 - Visita in ampiezza (breadth first search).

Una visita in ampiezza è un tipo di visita nella quale l'albero viene visitato un livello alla volta partendo dalla radice.

NB. Questo tipo di *visita* richiede l'utilizzo di una *coda*.

Implementazione delle visite

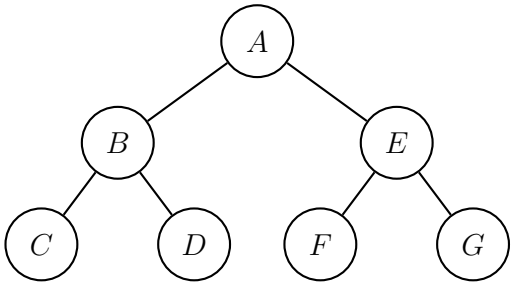
Frammento 19 - Implementazione delle visite.

```
% Depth-First-Search in Pre-order
dfs_pre_order(TREE t)
   if (t ≠ nil) then
      print t.value
      dfs_pre_order(t.left)
      dfs_pre_order(t.right)

% Depth-First-Search in In-order
dfs_in_order(TREE t)
   if (t ≠ nil) then
      dfs_in_order(t.left)
      print t.value
      dfs_in_order(t.right)

% Depth-First-Search in Post-order
dfs_post_order(TREE t)
   if (t ≠ nil) then
      dfs_post_order(t.left)
      dfs_post_order(t.right)
      print t.value
```

```
% Breadth-First-Search
bfs(TREE t)
   QUEUE q = Queue()
   q.enqueue(t)
   while (not q.isEmpty()) do
      TREE u = q.dequeue()
      print u.value
      % Accoda entrambi i figli
      % del nodo corrente
      u = u.left()
      if (u ≠ nil) then
         q.enqueue(u)
      u = u.parent().right()
      if (u ≠ nil) then
         q.enqueue(u)
```

Esempi di visite

Vediamo un esempio di utilizzo delle *visite*:



- *DFS Pre-order*: `A B C D E F G`;
- *DFS In-order*: `C B D A F E G`;
- *DFS Post-order*: `C D B F G E A`;
- *BFS*: `A B E C D F G`;

NB. Il *costo computazionale* di una *visita* di un *albero* contenente n *nodi* è $\Theta(n)$ poiché ogni *nodo* viene visitato una sola volta.

5.3 Alberi generici

Negli *alberi generici*, ovviamente, non possiamo parlare di *figlio sinistro* e *figlio destro*, ma possiamo comunque distinguere i *figli* di un *nodo* in base alla loro posizione. Parleremo infatti, di *figlio più a sinistra*, o *primo figlio*, e *fratello destro*, o *prossimo fratello*.

5.3.1 Specifica

Frammento 20 - Albero generico.

```
% Costituisce un nuovo nodo, contenente v, senza figli o genitori
Tree(ITEM v)
% Legge il valore memorizzato nel nodo
ITEM read()
% Modifica il valore memorizzato nel nodo
write(ITEM v)
% Restituisce il padre, oppure nil se questo è il nodo radice
TREE parent()
% Restituisce il primo figlio da sinistra, oppure nil se questo nodo
% è una foglia
TREE leftmostChild()
% Restituisce il primo fratello sulla destra, oppure nil se è assente
TREE rightSibling()
% Inserisce il sottoalbero t come primo figlio di questo nodo
insertChild(TREE t)
% Inserisce il sottoalbero t come prossimo fratello di questo nodo
insertSibling(TREE t)
% Distrugge l'albero radicato identificato dal primo figlio
deleteChild()
% Distrugge l'albero radicato identificato dal prossimo fratello
deleteSibling()
% Distrugge l'albero radicato identificato dal nodo
delete(TREE t)
```

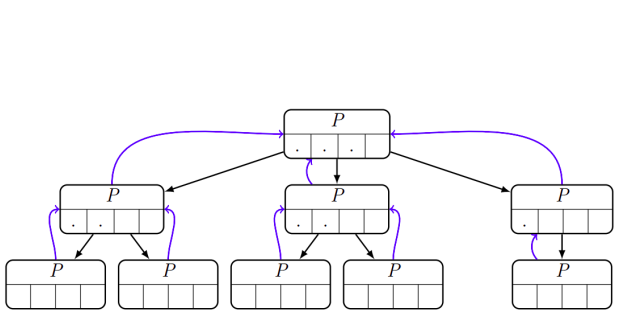
5.3.2 Memorizzazione di un albero generico

A differenza degli *alberi binari*, per gli *alberi generici* esistono diversi modi per rappresentarli in memoria. I modi sono tre:

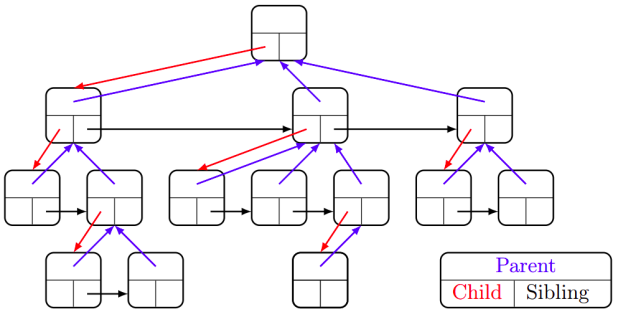
- *Vettore dei figli*: ogni *nodo* contiene un riferimento al *padre* e un vettore contenente i puntatori ai *figli*;
- *Primo figlio, prossimo fratello*: ogni *nodo* contiene un riferimento al *padre* e un riferimento al prossimo *fratello*;
- *Vettore dei padri*: l'*albero* è rappresentato da un vettore di coppie nelle quali il primo valore è il valore associato al *nodo* e il secondo è l'indice della posizione del *padre* nel vettore;

NB. L'approccio con *vettore dei figli* può causare uno spreco di memoria se molte celle di quei vettori puntano a `nil`.

Tecniche di memorizzazione a confronto

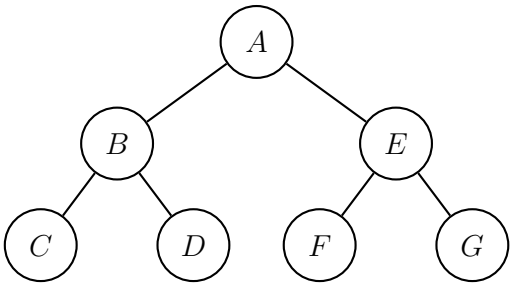


(a) *Vettore dei figli*



(b) *Primo figlio, prossimo fratello*

1	A	0
2	B	1
3	E	1
4	C	2
5	D	2
6	F	3
7	G	3



(c) *Vettore dei padri*

Fig. 5.3: Memorizzazione di *alberi generici*

5.3.3 Implementazione di un albero generico

Di seguito, è riportata l'implementazione di un *albero generico* realizzato con la tecnica di memorizzazione *primo figlio, prossimo fratello*.

Frammento 21 - Implementazione di un albero generico.

```
TREE parent  % Riferimento al padre
TREE child         % Riferimento al
                     % primo figlio
TREE sibling       % Riferimento al
                 % prossimo fratello
ITEM value                  % Valore

TREE Tree(ITEM v)
   TREE t = new TREE
   t.value = v
   t.parent = nil
   t.child = nil
   t.sibling = nil
   return t
```

```
% Inserisce t prima dell'attuale
% figlio
insertChild(TREE t)
   t.parent = this
   t.sibling = child
   child = t

% Inserisce t prima dell'attuale
% prossimo fratello
insertSibling(TREE t)
   t.parent = parent
   t.sibling = sibling
   sibling = t
```

```
ITEM read()
   return value

write(ITEM v)
   value = v

TREE parent()
   return parent

TREE leftmostChild()
   return child

TREE rightSibling()
   return sibling
```

```
deleteChild()
   TREE c = child.rightSibling()
   delete(child)
   child = c

deleteSibling()
   TREE s = sibling.rightSibling()
   delete(sibling)
   sibling = s

delete(TREE t)
   TREE u = t.leftmostChild()
   while (u ≠ nil) do
      TREE next = u.rightSibling()
      delete(u)
      u = next
   delete t
```

5.3.4 Visite di un albero generico

Anche per gli *alberi generici* valgono le stesse *visite* viste per gli *alberi binari* ad eccezione della *visita In-order* perché in questo caso non è ben definibile una situazione intermedia.

Implementazione delle visite

Frammento 22 - Implementazione delle visite.

```
% Depth-First-Search in Pre-order
dfs_pre_order(TREE t)
   if (t ≠ nil) then
      print t.value
      TREE u = t.leftmostChild()
      while (u ≠ nil) do
         dfs_pre_order(u)
         u = u.rightSibling()

% Depth-First-Search in Post-order
dfs_post_order(TREE t)
   if (t ≠ nil) then
      TREE u = t.leftmostChild()
      while (u ≠ nil) do
         dfs_post_order(u)
         u = u.rightSibling()
      print t.value
```

```
% Breadth-First-Search
bfs(TREE t)
   QUEUE q = Queue()
   q.enqueue(t)
   while (not q.isEmpty()) do
      TREE u = q.dequeue()
      print u.value
      % Accoda tutti i figli
      % del nodo corrente
      u = u.leftmostChild()
      while (u ≠ nil) do
         q.enqueue(u)
         u = u.rightSibling()
```

Capitolo Nr.6

Alberi binari di ricerca e alberi bilanciati

6.1 Alberi binari di ricerca

Precedentemente abbiamo accennato al concetto di *Dizionario*, ovvero una *struttura dati* costituita da coppie chiave-valore. Un *dizionario* può essere implementato in molteplici modi e come abbiamo visto, implementazioni diverse possono comportare *complessità* diverse per le singole operazioni che agiscono su quella *struttura*.

Ad, esempio, la seguente tabella riporta le complessità delle operazioni per tre possibili implementazioni:

Struttura dati	**lookup**	**insert**	**remove**
Vettore ordinato	$O(\log n)$	$O(1)$	$O(n)$
Vettore non ordinato	$O(n)$	$O(1)$	$O(1)$
Lista non ordinata	$O(n)$	$O(1)$	$O(1)$

NB. La *complessità* delle operazioni di `insert` e `remove` per le implementazioni con *vettore e lista non ordinati* è $O(1)$ solo se ipotizza che, nella `insert` non si debba verificare se l'elemento è già presente o meno, e nella `remove`, che si conosca già la posizione dell'elemento da rimuovere.

Se non si fanno queste ipotesi, la *complessità* diventa $O(n)$.

A questo punto ci chiediamo se non sia possibile implementare un *dizionario* sfruttando gli *alberi binari*. Supponiamo che ogni *nodo* contenga una coppia formata da una chiave e un valore. E ipotizziamo anche che le chiavi appartengano ad un insieme totalmente ordinato, cioè che sia sempre possibile stabile se una chiave precede o meno un'altra.

L'implementazione di un *nodo* dell'*albero* potrebbe quindi essere:

Frammento 23 - Nodo albero.

```
TREE parent
TREE left
TREE right
ITEM key, value
```

Poiché le chiavi sono ordinabili, possiamo ipotizzare che per ogni *nodo* u valgano le seguenti due proprietà:

1. Le chiavi contenute nel *sottoalbero sinistro* di u sono tutte minori di $u.key$;
2. Le chiavi contenute nel *sottoalbero destro* di u sono tutte maggiori di $u.key$;

NB. Le proprietà appena definite permettono di realizzare un algoritmo di ricerca dicotomica. Giunti a questo punto possiamo definire una regola generale:

Definizione 44 - Albero binario di ricerca (ABR).

Un albero binario di ricerca è un albero binario nel quale per ogni nodo u vale:

1. *Ogni nodo del sottoalbero sinistro ha un valore inferiore di quello del nodo u;*
2. *Ogni nodo del sottoalbero destro ha un valore maggiore di quello del nodo u;*

6.1.1 Specifica

Per un *dizionario* implementato come *albero binario di ricerca* possiamo definire la seguente *specifica*.

Frammento 24 - Dizionario come albero binario di ricerca.

```
% Getter
TREE parent()
TREE left()
TREE right()
ITEM key()
ITEM value()

% Operazioni per il dizionario
ITEM lookup(ITEM k)
insert(ITEM k, ITEM v)
remove(ITEM k)

% Operazioni definibili su un albero di ricerca binaria

% Restituisce il nodo successore di t, cioè il più piccolo
% tra i nodi più grandi di t
TREE successorNode(TREE t)
% Restituisce il nodo predecessore di t, cioè il più grande
% tra i nodi più piccoli di t
TREE predecessorNode(TREE t)
% Restituisce il nodo con valore minore
TREE min(TREE t)
% Restituisce il nodo con valore maggiore
TREE max(TREE t)

% Funzioni interne ausiliarie

% Restituisce il nodo dell'albero t contenente la chiave k,
% se è presente, altrimenti nil
TREE lookupNode(TREE t, ITEM k)
% Inserisce l'associazione chiave-valore (k,v) nell'albero t.
% Se la chiave è già presente sostituisce il valore associato, altrimenti
```

```
% viene inserita una nuova associazione.
% Se t è nil restituisce il nodo creato, altrimenti t
TREE insertNode(TREE t, ITEM k, ITEM v)
% Rimuove il nodo associato alla chiave k e restituisce la radice
% dell'albero (potrebbe essere stata cambiata)
TREE removeNode(TREE t, ITEM k)
```

Implementazione dizionario Operazioni a parte, l'implementazione di *dizionario* mediante *albero binario di ricerca* non è altro che:

Frammento 25 - Dizionario come ABR.

```
TREE tree

Dictionary()
   tree = nil
```

6.1.2 Ricerca

L'implementazione della funzione `lookup` è banale:

Frammento 26 - Dizionario come ABR - lookup.

```
ITEM lookup(ITEM k)
   TREE t = lookupNode(tree, k)
   if (t ≠ nil) then
      return t.value()
   else
      return nil
```

La `lookupNode` può essere implementata sia in modo ricorsivo che iterativo:

Frammento 27 - Implementazione ricorsiva e iterativa di lookupNode.

```
% Versione ricorsiva
TREE lookupNode(TREE t, ITEM k)
   if (t == nil or t.key == k) then
      return t
   else
      if (k < t.key) then
         return lookupNode(t.left, k)
      else
         return lookupNode(t.right, k)
```

```
% Versione iterativa
TREE lookupNode(TREE t, ITEM k)
   TREE u = t
   while (t ≠ nil and t.key ≠ k) do
      if (k < u.key) then
         % Sottoalbero di sinistra
         u = u.left
      else
         % Sottoalbero di destra
         u = u.right
   return u
```

6.1.3 Massimo e minimo

Grazie alle proprietà degli *alberi binari di ricerca* la ricerca del minimo e del massimo si riduce alla ricerca del *nodo* più a sinistra e più a destra dell'*albero*.

Frammento 28 - Implementazione minimo e massimo.

```
% Ricerca del minimo
TREE min(TREE t)
   TREE u = t
   if (u == nil) do
      return u
   while (u.left ≠ nil) do
      u = u.left
   return u
```

```
% Ricerca del massimo
TREE max(TREE t)
   TREE u = t
   if (u == nil) do
      return u
   while (u.right ≠ nil) do
      u = u.right
   return u
```

6.1.4 Successore e predecessore

Definizione 45 - Successore.

Il successore di un nodo u è il più piccolo nodo maggiore di u.

Definizione 46 - Predecessore.

Il predecessore di un nodo u è il più grande nodo minore di u.

Ricerca del successore Poiché il *successore* di un *nodo* è il più piccolo tra i *nodi* maggiori di esso, la ricerca del *successore* di un *nodo* u non può che essere fatta sui *nodi* maggiori di u. A questo punto quindi, si configurano due casi:

1. *u ha un figlio destro*: il *successore* di u è il *minimo* tra i *nodi* del suo *sottoalbero destro*;
2. *u non ha un figlio destro*: il *successore*, se esiste, è uno degli *avi* di u, ovvero uno dei *nodi* superiori. Quindi, risalendo l'*albero*, il *successore* è il primo *nodo* maggiore di u, cioè il primo *nodo* per il quale u sta nel suo *sottoalbero sinistro*;

Frammento 29 - Implementazione successore.

```
TREE successorNode(TREE t)
   if (t == nil) then return t
   if (t.right ≠ nil)                          % Caso 1: t ha un figlio destro
      return min(t.right)
   else                                        % Caso 2: t non ha un figlio destro
      TREE p = t.parent
      while (p ≠ nil and t == p.right) do
         t = p
         p = p.parent
      return p
```

Ricerca del predecessore Per il *predecessore* valgono le stesse considerazioni fatte per la ricerca del *successore*, ad eccezione del fatto che, se esiste, il *predecessore* di un *nodo* u sarà il massimo del suo *sottoalbero sinistro* oppure il primo *avo* che ha u come elemento del *sottoalbero destro*.

Frammento 30 - Implementazione predecessore.

```
TREE successorNode(TREE t)
   if (t == nil) then return t
   if (t.left ≠ nil)                          % Caso 1: t ha un figlio sinistro
      return max(t.left)
   else                                     % Caso 2: t non ha un figlio sinistro
      TREE p = t.parent
      while (p ≠ nil and t == p.left) do
         t = p
         p = p.parent
      return p
```

6.1.5 Inserimento

Per eseguire l'inserimento di una nuova associazione nel *dizionario* è sufficiente fare uso della funzione `insertNode`, che va ad inserire un *nodo*, o a modificarne uno esistente, in un *albero binario di ricerca*.

Frammento 31 - Dizionario come ABR - insert.

```
insert(ITEM k, ITEM v)
   tree = insertNode(tree, k, v)
```

Nemmeno l'implementazione della `insertNode` crea molti problemi, in quanto è sufficiente applicare ricorsivamente le *proprietà degli alberi binari di ricerca*. Andremo quindi ad invocare ricorsivamente l'`insertNode`, sul *sottoalbero sinistro* o *destro* a seconda della relazione d'ordine tra il valore del nodo (i.e il valore della chiave associata al *nodo*) e il valore della chiave da inserire, fino al raggiungimento di una *foglia*, alla quale aggiungeremo un nuovo *figlio destro* o *sinistro*, o di un *nodo*, associato ad una chiave con lo stesso valore di quella da inserire, nel qual caso sostituiremo il valore salvato nel *nodo* con quello nuovo.

Prima però di passare all'implementazione vera e propria, definiamo una funzione ausiliaria `link` che presi due *nodi* p, u e una chiave k, inserisce il *nodo* u come *figlio sinistro* di p se $p.key > k$, altrimenti lo inserisce come *figlio destro*.

Frammento 32 - Implementazione link.

```
link(TREE p, TREE u, ITEM k)
   if (u ≠ nil) then
      u.parent = p                          % Inserisce il riferimento al padre
   if (p ≠ nil) then
      if (p.key > k) then
         p.left = u                          % Inserisce u come figlio sinistro
      else
         p.right = u                           % Inserisce u come figlio destro
```

Frammento 33 - Implementazione ricorsiva insertNode.

```
TREE insertNode(TREE t, ITEM k, ITEM v)
   % L'albero è vuoto, quindi creo un nuovo nodo e lo restituisco
   if (t == nil) then
      return Tree(k, v)
   if (t.key == k) then
      t.value = v      % L'associazione esiste già, per cui aggiorno il valore
      return t
   if (t.key > k) then
      if (t.left == nil) then        % Inserisce il nodo come figlio sinistro
         link(t, Tree(k, v), k)
         return t
      else
         return insertNode(t.left, k, v)      % Continua a discendere l'albero
   else
      if (t.right == nil) then         % Inserisce il nodo come figlio destro
         link(t, Tree(k, v), k)
         return t
      else
         return insertNode(t.right, k, v)     % Continua a discendere l'albero
```

È anche possibile realizzare una versione iterativa della `insertNode`

Frammento 34 - Implementazione iterativa insertNode.

```
TREE insertNode(TREE t, ITEM k, ITEM v)          % Riferimento al padre
   TREE p = nil
   TREE u = t
   while (u ≠ nil and u.key ≠ k) do       % Cerca la posizione di inserimento
      p = u
      u = iif(k < u.key, u.left, u.right)
   if (u ≠ nil and u.key == k) then                % La chiave è già presente
      u.value = v
   else
      TREE newTree = Tree(k, v)                          % Crea un nuovo nodo
      link(p, newTree, k)                          % Inserisce il nuovo nodo
      if (p == nil) then              % Il nodo creato è il primo dell'albero
         t = newTree
   return t                  % Restituisce il nuovo albero o quello modificato
```

6.1.6 Cancellazione

Come per la ricerca e l'inserimento, anche l'implementazione della funzione `remove` è basata su un'altra funzione, la `removeNode`.

Frammento 35 - Dizionario come ABR - remove.

```
remove(ITEM k)
   tree = removeNode(tree, k)
```

L'implementazione della `removeNode` non è per nulla banale, in quanto, se u è il *nodo* da eliminare, vanno considerate diverse casistiche:

1. *u non ha figli*: si elimina il *nodo*;
2. *u ha un figlio f*: si elimina u e si rende f il *figlio* dell'*ex-padre* di u in sostituzione di u (*short-cut*);
3. *u ha due figli*: si individua il *successore* s di u. Poiché il *successore*, per definizione, non ha *figlio sinistro*, si stacca s dal proprio *padre* e si attacca il suo *figlio destro*, se esiste, come *figlio sinistro* del *padre* di s. Infine, si copia il valore di s su u e si elimina s;

Frammento 36 - Implementazione removeNode.

```
TREE removeNode(TREE t, ITEM k)
   TREE u = lookupNode(t, k)
   if (u ≠ nil) then
      if (u.left == nil and u.right == nil) then                    % Caso 1
         link(u.parent, nil, k)                              % Rimuove il figlio
         delete u
      else if (u.left ≠ nil and u.right ≠ nil) then                 % Caso 3
         TREE s = successorNode(u)
         link(s.parent, s.right, s.key) % Attacca il figlio di s al padre di s
         u.key = s.key                                  % Copia su u la chiave di s
         u.value = s.value                              % Copia su u il valore di s
         delete s
      else if (u.left ≠ nil and u.right == nil)  % Caso 2 con figlio sinistro
         link(u.parent, u.left, k)
         if (u.parent == nil) then
            t = u.left
      else                                              % Caso 2 con figlio destro
         link(u.parent, u.right, k)
         if (u.parent == nil) then
            t = u.right
   return t
```

Vediamo la dimostrazione della correttezza di questa implementazione:

Dimostrazione. Procediamo caso per caso. Sia u il *nodo* che si intende rimuovere dall'*albero*:

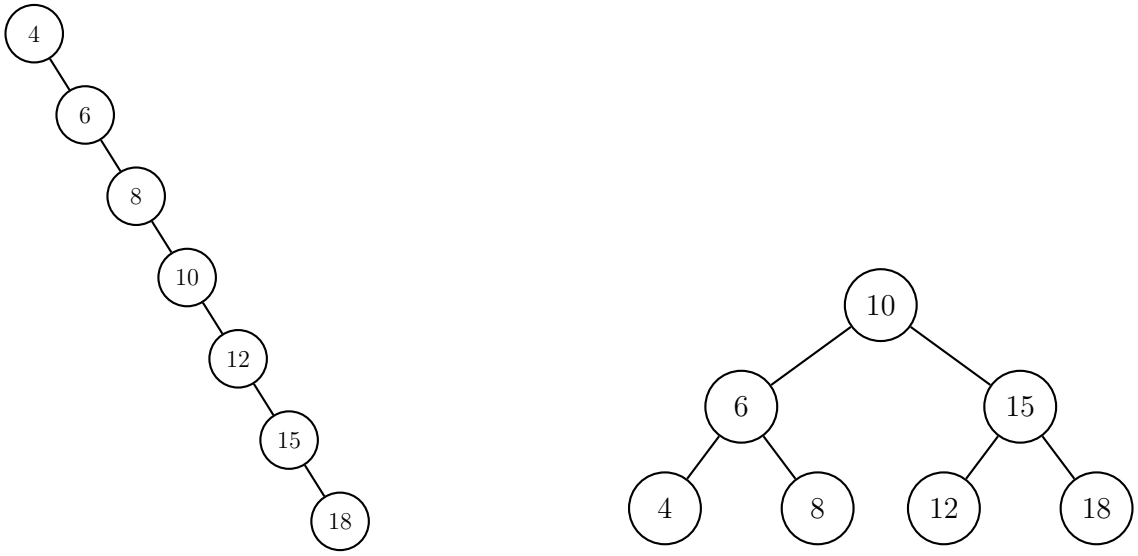
1. *Nessun figlio*: eliminare *foglie* non inficia sull'ordine degli altri nodi;
2. *Un solo figlio*:
   (a) *Figlio destro*: se u è il *figlio destro* di p tutti i valori nell'*albero* che ha come *radice* u sono maggiori di p;
   (b) *Figlio sinistro*: se u è il *figlio sinistro* di p tutti i valori nell'*albero* che ha come *radice* u sono minori di p;
3. *Due figli*: il *successore* s del *nodo* è maggiore di tutti i nodi del *sottoalbero di sinistra* di u, ma allo stesso tempo, è anche minore di tutti i *nodi* nel *sottoalbero destro* di u, di conseguenza può essere sostituito a u;

□

6.1.7 Costo computazionale delle operazioni

Tutte le operazioni viste per gli *alberi binari di ricerca* sono confinate ai *nodi* posizionati lungo un cammino semplice dalla *radice* a una *foglia*. Di conseguenza, se h è l'*altezza* di un *albero*, tutte le operazioni sono $O(h)$.

A questo punto vale la pena chiedersi quali siano il caso peggiore e quello migliore.



(a) Caso peggiore $h = O(n)$

(b) Caso migliore $h = O(\log n)$

Fig. 6.1: *Caso peggiore* e *caso migliore*

Da queste immagini risulta evidente che sia meglio evitare di lavorare con *alberi* troppo "*sbilanciati*".

6.2 Alberi binari di ricerca bilanciati

Abbiamo terminato l'ultima sezione discutendo brevemente il modo in cui l'*altezza* di un *albero* determini il costo delle operazioni. In particolare, abbiamo accennato al fatto nel caso pessimo tutti i *nodi* sono posizionati come in una *lista*, mentre nel caso migliore l'*albero* gode una struttura più simmetrica rispetto alla *radice*.

Ovviamente, nella maggior parte delle situazioni non si avrà a che fare né con l'uno che con l'altro ed è quindi ragionevole attendersi, generalmente, un'altezza inferiore a quella del caso pessimo. Addirittura, è dimostrato che se gli elementi vengono inseriti in ordine casuale, l'*altezza media* è $O(\log n)$.

Nella realtà però, difficilmente ci si affida alla casualità e quindi, per lavorare con *alberi* di *altezza* ottimale, si deve ricorrere a tecniche che garantiscano un buon livello di *bilanciamento*.

Definizione 47 - Fattore di bilanciamento.

È definito fattore di bilanciamento di un nodo v la massima differenza di l'altezza tra i suoi sottoalberi ed è indicato in simboli come $\beta(v)$.

Esistono diversi algoritmi di *bilanciamento*, quali ad esempio:

- *Alberi AVL* (Adelson-Veskii e Landis): bilanciano l'*albero* facendo uso di rotazioni e garantiscono che per ogni *nodo* v $\beta(v) \leq 1$;

- *B-alberi* (Bayer e McCreight): ogni *nodo* v ha $\beta(v) = 0$ e sono specializzati per *strutture* salvate in memoria secondaria;
- *Alberi 2-3* (Hopcroft): ogni *nodo* v ha $\beta(v) = 0$ e sono basati su una *struttura* dinamica che attraverso operazioni di merge e split, distribuisce i *nodi* su *alberi* con due o tre *figli*;

Noi, in questa trattazione, analizzeremo un quarto tipo di *alberi*, gli *Alberi Red-Black*, ideati da Guibas e Sedgewick nel 1978.

6.3 Alberi Red-Black

Definizione 48 - Alberi Red-Black.

Gli Alberi Red-Black sono alberi binari di ricerca nei quali ogni nodo è colorato di rosso oppure di nero. Le chiavi vengono mantenute soltanto nei nodi interni e le foglie sono costituite da nodi speciali con valore Nil.

Inoltre, vengono rispettati i seguenti vincoli:

1. *La radice è nera;*
2. *Tutte le foglie sono nere;*
3. *Entrambi i figli di un nodo rosso sono neri;*
4. *Ogni cammino semplice da un nodo ad una delle sue foglie ha sempre lo stesso numero di nodi neri;*

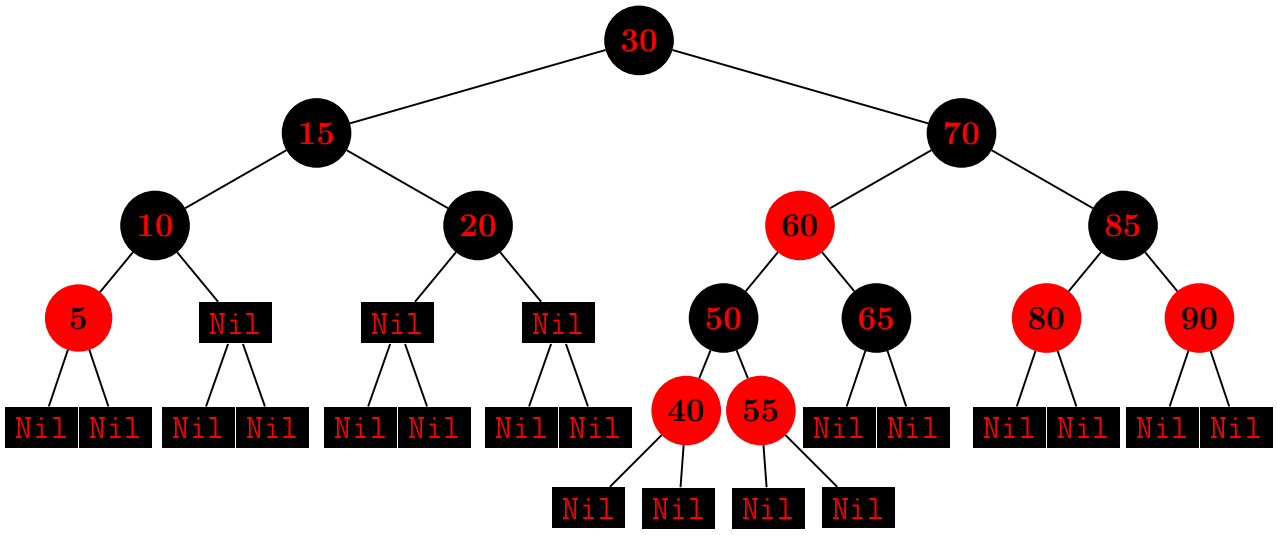


Fig. 6.2: *Albero Red-Black*

Diamo inoltre le seguenti definizioni:

Definizione 49 - Altezza nera di un nodo.

Dato un nodo v, l'altezza nera di quel nodo, indicata come $bh(v)$, è il numero di nodi neri lungo ogni cammino da v (escluso) ad ogni sua foglia (inclusa).

Definizione 50 - Altezza nera di un Albero Red-Black.

L'altezza nera di un Albero Red-Black corrisponde all'altezza nera della sua radice.

L'*albero* della figura 6.2 ha *altezza nera* pari a 3.

NB. Per ogni *Albero Red-Black* sono possibili più colorazioni che possono anche portare ad *altezze nere* diverse.

Foglie come nodi Nil Come detto, le *foglie* sono rappresentate da speciali *nodi* con valore `Nil`. Questo significa, che invece di un puntatore `nil`, c'è un riferimento ad un *nodo* `Nil` di colore nero. E, poiché tutti i *nodi* `Nil` sono uguali, ne esiste uno solo al quale fanno riferimento tutti i *nodi* che hanno delle *foglie*.

In realtà, i *nodi* che hanno come *figli* questo tipo di *nodi*, sono essi stessi le vere foglie dell'*albero*, in quanto, i *nodi* `Nil` non sono altro che *sentinelle* create per permettere di accedere al colore dei *figli* senza dover gestire il caso in cui questi siano `nil`.

Definizione di un nodo in un Albero Red-Black In generale, la definizione dei *nodi* per *Alberi Red-Black* non è diversa da quella dei *nodi* per *alberi binari di ricerca* classici. L'unica differenza sta nella necessità di tenere traccia del colore assegnato ad ogni *nodo*.

Frammento 37 - Implementazione di un nodo in un Albero Red-Black.

```
TREE parent
TREE left
TREE right
int color                    % Per efficienza si sarebbe potuto usare un boolean
ITEM key, value
```

6.3.1 Rotazioni

Ogni volta che si va ad inserire un elemento in un *Albero Red-Black* è possibile che le *condizioni di bilanciamento* vengano violate.

Quando ciò accade è possibile agire applicando delle operazioni di *rotazione*, oppure andando direttamente a modificare i colori dei *nodi* presenti nella zona in cui i vincoli risultano violati.

Una *rotazione* prevede che la *radice* di un *albero* venga sostituita con uno dei suoi *figli*. Un'operazione di questo tipo, se opportunamente sfruttata, permette di bilanciare un "*albero sbilanciato*".

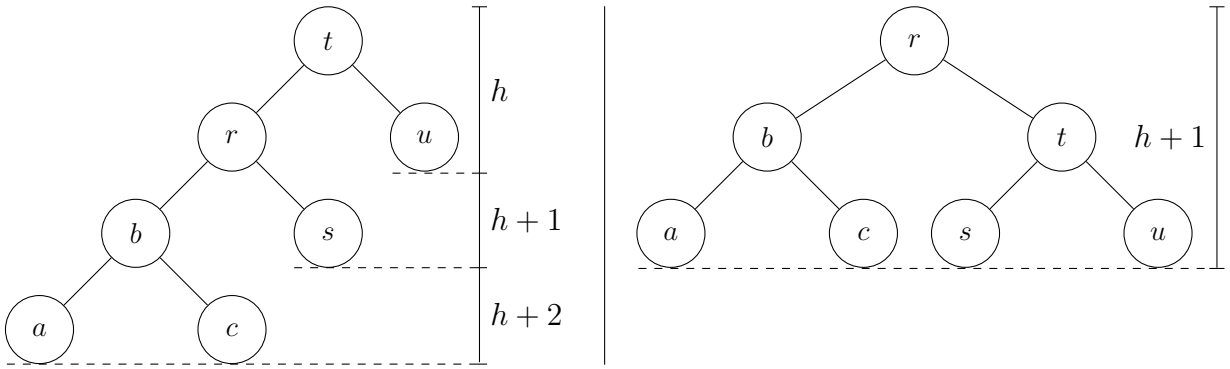


Fig. 6.3: Esempio di *rotazione verso destra* del *nodo t*

Frammento 38 - Implementazione funzioni di rotazione.

```
% Rotazione verso sinistra
TREE rotateLeft(TREE r)
   TREE s = r.right
   TREE t = r.parent

   % Il sottoalbero sinistro di s
   % diventa il figlio destro di r
   r.right = s.left
   if (s.left ≠ nil) then
      s.left.parent = r

   % r diventa il figlio sinistro
   % di s
   s.left = r
   r.parent = s

   % s diventa il figlio di t
   s.parent = t
   if (t ≠ nil) then
      if (t.left == r) then
         t.left = s
      else
         t.right = s
   return s
```

```
% Rotazione verso destra
TREE rotateRight(TREE r)
   TREE b = r.left
   TREE t = r.parent

   % Il sottoalbero destro di b
   % diventa il figlio sinistro di r
   r.left = b.right
   if (b.right ≠ nil) then
      b.right.parent = r

   % r diventa il figlio destro
   % di b
   b.right = r
   r.parent = b

   % b diventa il figlio di t
   b.parent = t
   if (t ≠ nil) then
      if (t.left == r) then
         t.left = b
      else
         t.right = b
   return b
```

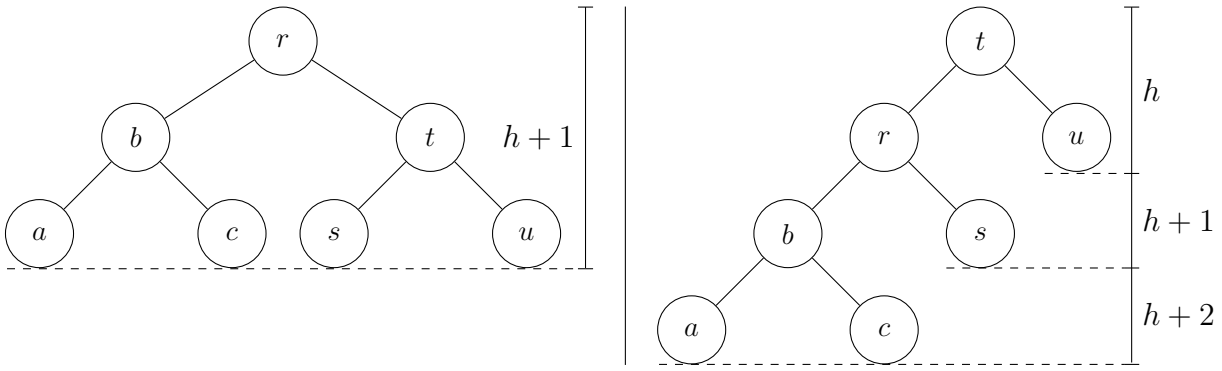


Fig. 6.4: Esempio di *rotazione verso sinistra* del *nodo t*

Da questo esempio si vede anche che applicando una rotazione su un *nodo* e applicando poi la rotazione inversa sullo stesso *nodo*, si riottiene l'*albero* di partenza della prima rotazione.

6.3.2 Inserimento

A questo punto, possiamo affrontare l'argomento *inserimento*, che nei principi generali non si discosta molto da quanto visto per gli *alberi binari di ricerca*. In particolare, la posizione di inserimento viene ricercata con le stesse regole, ma in più il nuovo *nodo* viene "colorato" di rosso.

Ovviamente, come già accennato, l'inserimento di un *nodo* può comportare la violazione dei *vincoli di bilanciamento*. È quindi opportuno che ad ogni inserimento, se necessario, l'*albero* venga ribilanciato.

Frammento 39 - Implementazione insertNode per Alberi Red-Black.

```
TREE insertNode(TREE t, ITEM k, ITEM v)
   TREE p = nil                                          % Riferimento al padre
   TREE u = t
   while (u ≠ nil and u.key ≠ k) do          % Cerca la posizione di inserimento
      p = u
      u = iif(k < u.key, u.left, u.right)
   if (u ≠ nil and u.key == k) then                        % Chiave già presente
      u.value = v
   else
      TREE newTree = Tree(k, v)                             % Crea un nuovo nodo
      link(p, newTree, k)                                % Inserisce il nuovo nodo
      balanceInsert(newTree)           % Garantisce il bilanciamento dell'albero
      if (p == nil) then                % Il nodo creato è il primo dell'albero
         t = newTree
   return t
```

La funzione `balanceInsert` si occupa di garantire il rispetto di tutti i *vincoli di bilanciamento* modificando la colorazione dei *nodi* o applicando delle rotazioni.

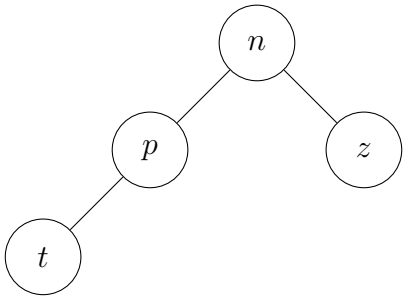
Questa funzione procede con un approccio bottom-up a partire dal *nodo* inserito. A mano a mano che risale, i *figli* di ogni *nodo* rosso vengono colorati di nero, garantendo così il rispetto del *vincolo 3*. Tutte le violazioni ancora irrisolte vengono in questo modo spostate verso l'alto. Quando infine viene raggiunta la *radice*, questa viene colorata di nero come richiesto dal *vincolo 1*, ma ciò si rende necessario solo se il *nodo* inserito era proprio la *radice* o se risalendo l'*albero* era stata colorata di rosso.

NB. Le operazioni di ripristino sono necessarie solo quando due *nodi* consecutivi sono rossi.

NB. Ogni *nodo* inserito viene in automatico colorato di rosso perché in questo modo c'è la possibilità che non sia necessario ribilanciare l'*albero*. Al contrario, l'inserimento di un *nodo* nero comporta sempre un ribilanciamento in quanto viene modificata l'*altezza nera* dell'*albero*.

Quando si inserisce un *nodo*, per capire se i *vincoli di bilanciamento* sono stati violati, è sufficiente analizzare quattro *nodi*:

- Il *nodo* t appena inserito;
- Il *padre* p di t;
- Il *padre* n di p o, equivalentemente, il *nonno* di t;
- Il secondo *figlio* z di n, ovvero il *fratello* di p o lo *zio* di t;



A seconda del colore di questi quattro *nodi* si configurano 7 possibili casi.

Caso 1 - Il nuovo nodo non ha padre Questo caso si configura quando viene inserita la *radice* dell'*albero* o quando risalendo si è arrivati alla *radice*. A questo punto l'unico vincolo che potrebbe risultare violato è il numero 1, quindi è sufficiente colorare il *nodo* di nero.

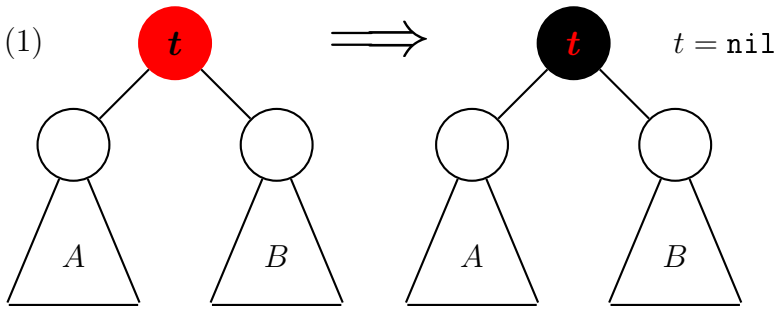


Fig. 6.5: Risoluzione del caso 1

Caso 2 - Il padre di *t* è nero In questo caso non serve fare nulla, poiché nessun vincolo può risultare violato.

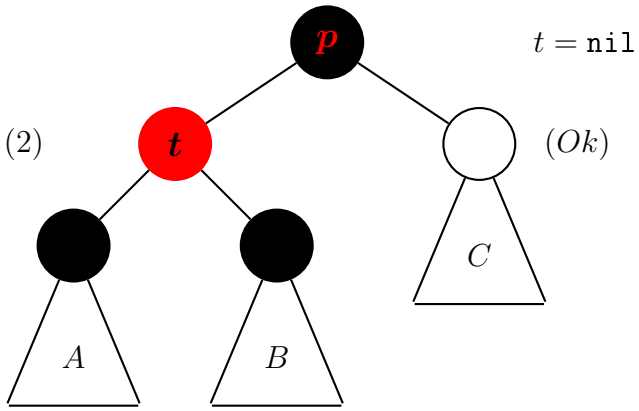


Fig. 6.6: Risoluzione del caso 2

Caso 3 - Sia *t* che il padre e lo zio sono rossi Poiché z è rosso, è possibile colorare di nero p e z e poi colorare di rosso n. A questo punto, poiché tutti i cammini che passano per p e z passano anche per n, l'*altezza nera* certamente non avrà subito variazioni.

A questo punto, è ancora possibile che n violi i *vincoli 1* e *3*. Di conseguenza, risaliamo l'*albero* ponendo $t = n$ e ripetiamo il ciclo.

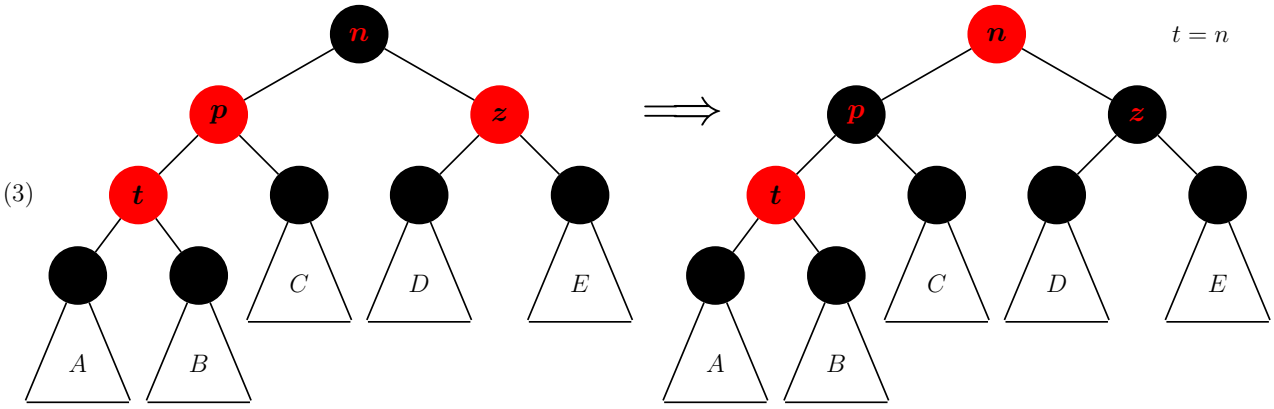


Fig. 6.7: Risoluzione del caso 3

Caso 4 - Sia t che il padre sono rossi e lo zio è nero

a. *t è il figlio destro di p e p è il figlio sinistro di n*: viene eseguita una rotazione verso sinistra partendo dal *nodo p* in modo da rendere p il *figlio sinistro* di t e potersi così ricondurre al *caso 5a*. In questo caso vengono coinvolti soltanto i *nodi* p e t che essendo entrambi rossi, non influenzano l'*altezza nera*;

b. *t è il figlio sinistro di p e p è il figlio destro di n*: viene eseguita una rotazione verso destra partendo dal *nodo p* in modo da rendere p il *figlio destro* di t e potersi così ricondurre al *caso 5b*. In questo caso vengono coinvolti soltanto i *nodi* p e t che essendo entrambi rossi, non influenzano l'*altezza nera*;

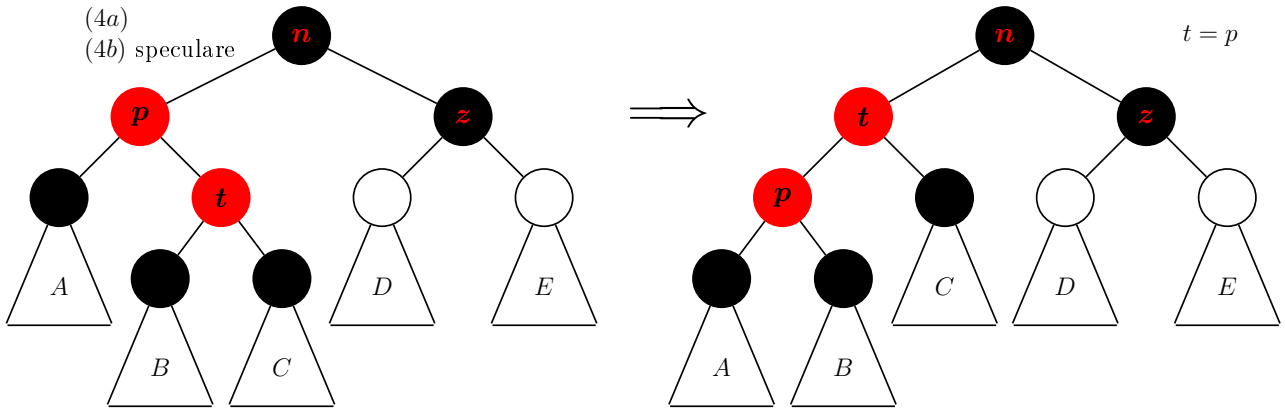


Fig. 6.8: Risoluzione del caso 4

Caso 5 - Sia t che il padre sono rossi e lo zio è nero

a. *t è il figlio sinistro di p e p è il figlio sinistro di n*: viene eseguita una rotazione verso destra partendo dal *nodo n* in modo da rendere t ed n *figli* di p. Quindi, colorando di rosso n e di nero p ci si porta in una situazione nella quale tutti i *vincoli di bilanciamento* sono rispettati e l'*altezza nera* è uguale alla situazione iniziale;

b. *t è il figlio destro di p e p è il figlio destro di n*: viene eseguita una rotazione verso sinistra partendo dal *nodo n* in modo da rendere t ed n *figli* di p. Quindi, colorando di rosso n e di nero p ci si porta in una situazione nella quale tutti i *vincoli di bilanciamento* sono rispettati e l'*altezza nera* è uguale alla situazione iniziale;

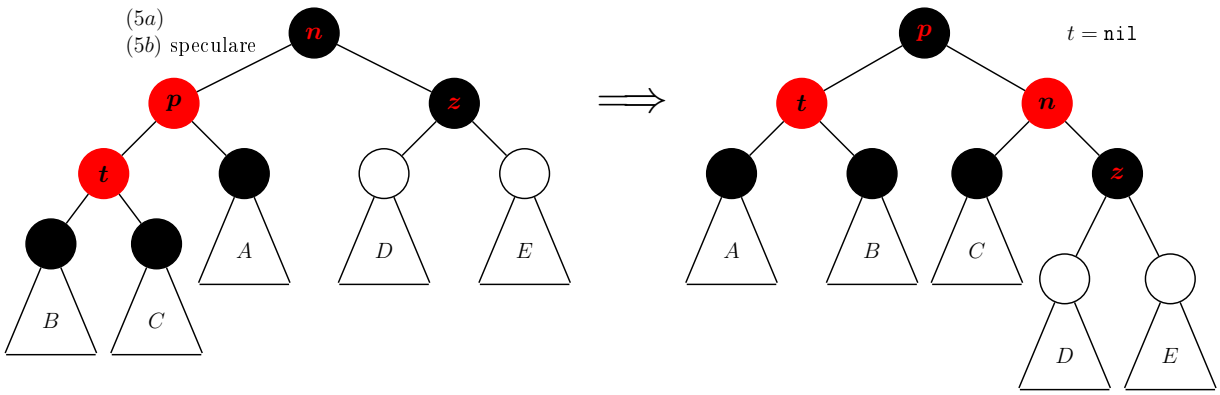


Fig. 6.9: Risoluzione del caso 5

Frammento 40 - Implementazione balanceInsert.

```
balanceInsert(TREE t)
   t.color = RED       % Il nodo inserito viene inizialmente colorato di rosso
   while (t ≠ nil) do
      TREE p = t.parent                                              % Padre
      TREE n = iif(p ≠ nil, p.parent, nil)                           % Nonno
      TREE z = iif(n ≠ nil, iif(n.left == p, n.right, n.left), nil)   % Zio
      if (p == nil) then                                             % Caso 1
         t.color = BLACK
         t = nil
      else if (p.color == BLACK) then                                % Caso 2
         t = nil
      else if (z.color == RED) then                                  % Caso 3
         p.color = BLACK
         z.color = BLACK
         n.color = RED
         t = n
      else
         if (t == p.right and p == n.left) then                      % Caso 4a
            rotateLeft(p)
            t = p
         else if (t == p.left and p == n.right) then                 % Caso 4b
            rotateRight(p)
            t = p
         else
            if (t == p.left and p == n.left) then                    % Caso 5a
               rotateRight(n)
            else if (t == p.right and p == n.right) then             % Caso 5b
               rotateLeft(n)

            p.color = BLACK
            n.color = RED
            t = nil
```

Complessità della balanceInsert Non è difficile rendersi conto che tutte le istruzione interne al ciclo `while` hanno *complessità* $O(1)$. Di conseguenza, la *complessità* totale della funzione dipende dal numero di volte che viene eseguito il ciclo. Poiché t quando viene inserito è una *foglia*,[1] il caso peggiore è quello in cui il ribilanciamento coinvolge tutti i *livelli* e quindi termina solo dopo aver raggiunto la *radice*.

In quel caso la *complessità* dipende dall'*altezza* dell'*albero*. Vedremo che negli *Alberi Red-Black* esiste una relazione ben precisa tra la loro *altezza* il numero di noi *nodi* che contengono.

[1] I *nodi* di un *albero* vengono sempre inseriti al *livello* delle *foglie*

6.3.3 Altezza degli Alberi Red-Black

Definizione 51 - Numero massimo di nodi interni.

In un Albero Red-Black, un sottoalbero di radice u contiene un numero $n \geq 2^{bh(u)} - 1$ di nodi interni.

Dimostrazione. Procediamo per induzione sull'altezza del nodo u.

Caso base Dimostro la disequazione per $h = 0$:

se $h = 0$, u è una *foglia* `Nil` e quindi il *sottoalbero* con *radice* u non può contenere *nodi interni*, quindi n deve per forza valere 0. Poiché:

$$n \geq 2^{bh(u)} - 1 = 2^0 - 1 = 1 - 1 = 0$$

ciò è confermato e quindi il caso base risulta verificato.

Passo induttivo Ipotizzo che $n \geq 2^{bh(u)} - 1$ per ogni *albero* con *radice* u e *altezza* $h \leq 1$. Dimostro la disequazione per $h > 1$:

se $h > 1$, u è certamente un *nodo interno* e quindi ha due *figli*. Ogni *figlio* di u ha un *altezza nera* che è pari a $bh(u)$ se è rosso e $bh(u) - 1$ se è nero. Ogni *sottoalbero* con *radice* uno dei *figli* di u ha *altezza* $h - 1$ e quindi, per ipotesi induttiva, so che ognuno di quei sottoalberi ha almeno $2^{bh(u)-1} - 1$ *nodi interni*.

Allora, per il *sottoalbero* con *radice* u vale:

$$n \geq 2 \cdot \left(2^{bh(u)-1} - 1\right) + 1$$

cioè, il numero di *nodi interni* del *sottoalbero* con *radice* u è pari, almeno, alla somma dei limiti inferiori al numero di *nodi interni* dei *sottoalberi* che hanno come *radice* un *figlio* di u, più 1, perché bisogna contare anche il *nodo* u stesso.

Poiché vale seguente catena di uguaglianze:

$$n \geq 2 \cdot \left(2^{bh(u)-1} - 1\right) + 1 = 2 \cdot \left(\frac{2^{bh(u)}}{2} - 1\right) + 1 = 2^{bh(u)} - 2 + 1 = 2^{bh(u)} - 1$$

il passo induttivo risulta verificato. □

Definizione 52 - Quota nera minima.

In un Albero Red-Black, almeno la metà dei nodi dalla radice a una foglia deve essere nera.

Dimostrazione. Per il secondo *vincolo di bilanciamento*, i *figli* di un *nodo* rosso devono essere neri e, poiché la situazione in cui sono presenti il minor numero di *nodi* neri è quando *nodi* rossi e neri sono alternati, almeno la metà dei *nodi* devono essere neri. □

Definizione 53 - Lunghezza massima dei cammini semplici.

In un Albero Red-Black*, dati due* cammini semplici*, dalla* radice *a due* foglie*, non è possibile che uno sia lungo più del doppio dell'altro.*

Dimostrazione. Per il quarto *vincolo di bilanciamento*, tutti i *cammini* da un *nodo* ad una *foglia* hanno lo stesso numero di *nodi* neri e, per la definizione precedente, almeno la metà di quei *nodi* deve essere nera. Di conseguenza, il caso più sbilanciato è quello in cui un *cammino* è costituito da soli *nodi* neri e l'altro da *nodi* neri e rossi alternati e quindi si ha che il secondo *cammino* è lungo il doppio del primo. □

Definizione 54 - Altezza massima di un Albero Red-Black.

*L'*altezza *massima di un* Albero Red-Black *con* n nodi interni *è al più* $2\log(n+1)$.

Dimostrazione. Vale quanto segue:

$$\begin{aligned} n \geq 2^{bh(u)} - 1 &\Leftrightarrow n \geq 2^{h/2} - 1 && \textit{Quota nera minima} \\ &\Leftrightarrow n + 1 \geq 2^{h/2} \\ &\Leftrightarrow \log(n+1) \geq h^2 && \textit{Applicazione logaritmo ai termini} \\ &\Leftrightarrow h \leq 2\log(n+1) \end{aligned}$$

□

Complessità funzione insertNode Fatte queste considerazioni sull'*altezza* massima degli *Alberi Red-Black*, possiamo affermare che la *complessità* della funzione `insertNode` è $O(\log n)$. Questo è vero in quanto costa $O(\log n)$ discendere l'*albero* dalla *radice* al punto di inserimento, costa $O(1)$ inserire il *nodo* e infine, costa $O(\log n)$ ribilanciare l'*albero* nel caso peggiore, il caso 3.

6.3.4 Cancellazione

Similmente a quanto visto per l'inserimento, la funzione `removeNode` per gli *Alberi Red-Black* è costruita a partire dalla controparte per *alberi binari di ricerca* ed è costituita da un insieme di casistiche che possono comportare o meno, la necessità di ribilanciare l'*albero*.

Le operazioni di ribilanciamento si rendono necessarie soltanto quando viene rimosso un *nodo* nero. Ciò è vero in quanto l'eliminazione di un *nodo* nero comporta sicuramente una variazione dell'*altezza nera* e potrebbe anche portare ad avere due *nodi* rossi in posizioni consecutive.

Più in generale, quando viene rimosso un *nodo* nero possono essere violati i *vincoli* 1, 3 o 4. In particolare:

- *Violato il vincolo 1*: viene eliminata la *radice* e la nuova *radice* è rossa;
- *Violato il vincolo 3*: viene eliminato un *nodo* che aveva il *padre* e almeno uno dei *figli* rossi;
- *Violato il vincolo 4*: ogni volta che viene eliminato un *nodo* nero, l'*altezza* nera cambia;

Frammento 41 - Implementazione removeNode per Alberi Red-Black.

```
TREE removeNode(TREE t, ITEM k)
   TREE u = lookupNode(t, k)
   if (u ≠ nil) then
      if (u.left == nil and u.right == nil) then                    % Caso 1
         link(u.parent, nil, k)                                % Rimuove il figlio
         delete u
      else if (u.left ≠ nil and u.right ≠ nil) then                 % Caso 3
         TREE s = successorNode(u)
         link(s.parent, s.right, s.key) % Attacca il figlio di s al padre di s
         u.key = s.key                                  % Copia su u la chiave di s
         u.value = s.value                              % Copia su u il valore di s
         delete s
```

```
      else                                          % Caso 2 con figlio destro
         link(u.parent, u.right, k)
         if (u.parent == nil) then
            t = u.right

      if (u.color == BLACK) then
         balanceRemove(t, u)          % Garantisce il bilanciamento dell'albero
   return t
```

Frammento 42 - Implementazione balanceRemove.

```
balanceRemove(TREE r, TREE t)
   while (t ≠ r and t.color == BLACK) do   % Il ciclo termina quando t diventa
                                           % la radice o un nodo rosso
      TREE p = t.parent                                          % Padre
      if (t == p.left) then                    % È sicuramente diverso da nil
         TREE f = p.right                                        % Fratello
         TREE ns = f.left                                  % Nipote sinistro
         TREE nd = f.right                                   % Nipote destro
         (f.color == RED) then                                      % Caso 1
            p.color = RED
            f.color = BLACK
            rotateLeft(p)
         else
            if (ns.color == BLACK and nd.color == BLACK) then       % Caso 2
               f.color = RED
               t = p
            else if (ns.color == RED and nd.color == BLACK) then    % Caso 3
               ns.color = BLACK
               f.color = RED
               rotateRight(f)
            else if (nd.color == RED) then                          % Caso 4
               f.color = p.color
               p.color = BLACK
               nd.color = BLACK
               rotateLeft(p)
               t = r                                  % Fa terminare il ciclo
      else                        % I casi si ripetono simmetrici ai precedenti
         ...
```

Complessità della funzione removeNode Come per la `insertNode` anche in questo caso la *complessità* totale è $O(\log n)$ in quanto dipende dall'*altezza* dell'*albero*.

Capitolo Nr.7

Grafi

Moltissimi problemi possono essere visti come problemi sui *grafi* e sebbene questi abbiano forma astratta, le loro soluzioni trovano applicazioni negli ambiti più disparati.

Abbiamo già dato una prima definizione generica di *grafo*, ma adesso andremo ad estenderla introducendo una serie di concetti che saranno necessari per affrontare proficuamente questo capitolo.

7.1 Introduzione

Diamo le seguenti definizioni.

Definizione 55 - Grafo orientato.

Un grafo orientato è una coppia $G = (V, E)$ dove V è l'insieme dei vertici, o dei nodi, del grafo ed E è un insieme di coppie ordinate di nodi (u, v) per $u, v \in V$ dette archi, o lati.

$$
\begin{aligned}
V &= \{\, a, b, c, d, e, f \,\} \\
E &= \{\, (a,b), (a,d), (d,a), \\
&\quad\ (b,c), (d,c), (d,e), \\
&\quad\ (e,c) \,\}
\end{aligned}
$$

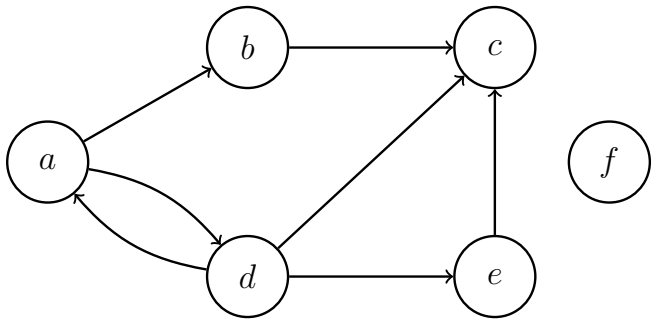


Fig. 7.1: *Grafo orientato*

Definizione 56 - Grafo non orientato.

Un grafo non orientato è una coppia $G = (V, E)$ dove V è l'insieme dei vertici, o dei nodi, del grafo ed E è un insieme di coppie non ordinate di nodi (u, v) per $u, v \in V$ dette archi, o lati.

$$
\begin{aligned}
V &= \{\, a, b, c, d, e, f \,\} \\
E &= \{\, (a,b), (a,d), (b,c), \\
&\quad\ (d,c), (d,e), (e,c) \,\}
\end{aligned}
$$

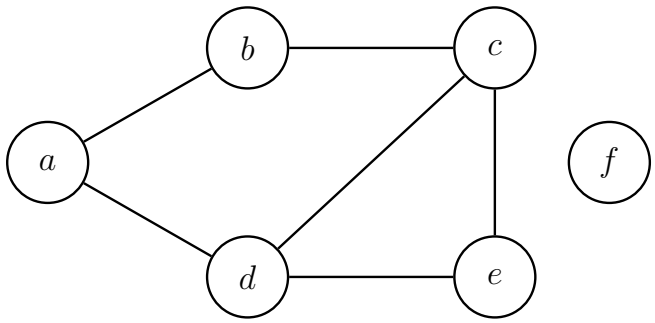
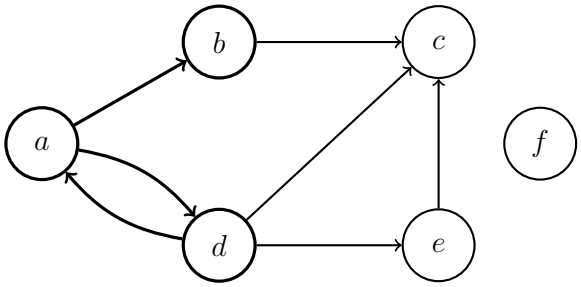


Fig. 7.2: *Grafo non orientato*

NB. I *grafi orientati* e *non orientati* sono anche detti, rispettivamente, *directed* e *undirected*. In un *grafo orientato* un *vertice* v è detto *adiacente* a u se esiste un *arco* (u, v) e, quell'arco, è detto *incidente* da u in v. Ovviamente, in un *grafo non orientato* la nozione di *adiacenza* è simmetrica.

Esempio 17 - Adiacenza e incidenza.

Nel seguente grafo orientato, per la parte evidenziata, valgono le seguenti relazioni di adiacenza e incidenza.



- (a, b) *è incidente da* a *a* b*;*
- (a, d) *è incidente da* a *a* d*;*
- (d, a) *è incidente da* d *ad* a*;*
- b *è adiacente ad* a*;*
- d *è adiacente ad* a*;*
- a *è adiacente a* d*;*

7.1.1 Dimensioni dei grafi

Definizione 57 - Numero di nodi e di archi.

Dato un grafo $G = (V, E)$ *il numero di nodi e il numero di archi corrispondono, rispettivamente, alla cardinalità degli insiemi* V *ed* E*.*

NB. Di seguito, salvo diverse indicazioni, indicheremo con n il numero di *nodi* e con m il numero di *archi*, cioè $n = |V|$ ed $m = |E|$.

Definizione 58 - Numero massimo di archi in un grafo orientato.

Dato un grafo orientato vale la seguente relazione sul numero di archi:

$$m \leq n(n-1)$$

Definizione 59 - Numero massimo di archi in un grafo non orientato.

Dato un grafo non orientato vale la seguente relazione sul numero di archi:

$$m \leq \frac{n(n-1)}{2}$$

NB. In entrambi i casi possiamo affermare che il numero di *archi* è $O(n^2)$.

NB. Per gli algoritmi che operano sui *grafi* la *complessità* è espressa sia in termine di n che m e quindi considereremo *lineare* un algoritmo con *complessità* $O(n+m)$.

7.1.2 Casi speciali

Definizione 60 - Grafo completo.

Un grafo con un arco tra tutte le coppie di nodi è detto completo.

Definizione 61 - Grafi sparsi e densi.

Un grafo con "pochi archi" è detto essere sparso, mentre uno con "tanti archi", denso.

NB. La definizione di *grafi sparsi* e *densi* non è formale, ma tendenzialmente un *grafo* con $m = O(n)$ o $O(n \log n)$ è considerato *sparso*. Sono invece considerati *densi* i *grafi* con $m = \Omega(n^2)$.

Possiamo definire gli *alberi* in funzione dei *grafi*.

Definizione 62 - Albero libero.

Un grafo con $m = n - 1$ è detto albero libero.

Definizione 63 - Albero radicato (visto come grafo).

Un grafo connesso con $m = n - 1$ archi nel quale uno dei nodi è stato designato come radice è detto albero radicato.

Definizione 64 - Foresta.

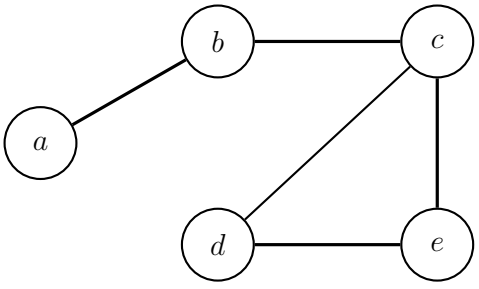
Un insieme di alberi è detto foresta.

7.1.3 Cammini e gradi dei nodi

Definizione 65 - Cammino.

In un grafo $G = (V, E)$, una sequenza di $k+1$ nodi u_0, u_1, ..., u_k è definita essere un cammino di lunghezza k, se è tale per cui l'arco $(u_i, u_{i+1}) \in E \; \forall 0 \leq i \leq k-1$.

Esempio 18 - Cammino.



La sequenza di nodi a, b, c, e, d è un cammino di lunghezza 4.

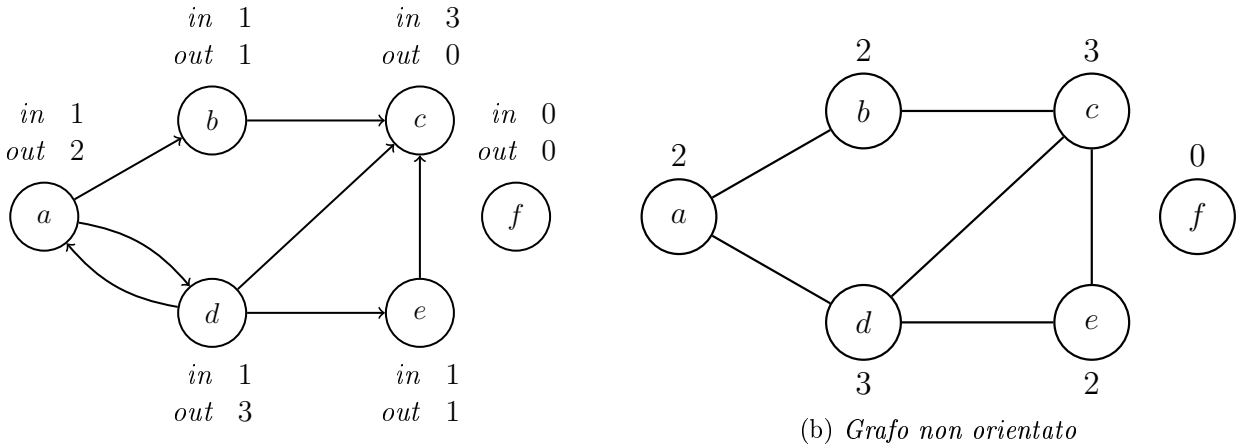
NB *In questo caso il cammino è anche detto semplice perché tutti i nodi che lo compongono sono distinti.*

Definizione 66 - Grado di un nodo in un grafo orientato.

In un grafo orientato si definisce grado entrante di un nodo il numero di archi incidenti su di esso, mentre il grado uscente è definito come il numero di archi incidenti da esso.

Definizione 67 - Grado di un nodo in un grafo non orientato.

In un grafo non orientato il grado di un nodo è definito come il numero di archi incidenti su di esso.



(a) *Grafo orientato*

(b) *Grafo non orientato*

Fig. 7.3: *Grado* dei *nodi* nei *grafi orientati* e *non orientati*

7.1.4 Specifica

Frammento 43 - Grafo.

```
% Crea un nuovo grafo
GRAPH Graph()
% Restituisce l'insieme di tutti i vertici
SET V()
% Restituisce il numero di nodi
int size()
% Restituisce l'insieme dei nodi adiacenti ad u
SET adj(NODE u)
% Aggiunge un nodo u al grafo
insertNode(NODE u)
% Aggiunge l'arco (u,v) al grafo
insertEdge(NODE u, NODE v)
% Rimuove il nodo u dal grafo
removeNode(NODE u)
% Rimuove l'arco (u,v) dal grafo
removeEdge(NODE u, NODE v)
```

7.1.5 Memorizzare un grafo

Per la memorizzazione dei *grafi* esistono due approcci: *matrici di adiacenza* e *liste di adiacenza*.

Memorizzazione di un grafo orientato L'approccio con *matrice di adiacenza* prevede che venga realizzata una matrice $n \times n$ tale che:

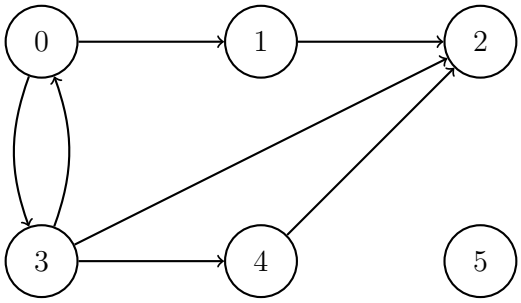
$$m_{u,v} = \begin{cases} 1 & \text{se } (u,v) \in E \\ 0 & \text{se } (u,v) \notin E \end{cases}$$

Quindi, il *grafo* viene rappresentato tramite una matrice di valori binari nella quale la cella che si trova nell'u-esima riga e v-esima colonna è 1 se e solo se esiste un *arco* da u a v.

D'altra parte, le *liste di adiacenza* sfruttano il concetto di *adiacenza* espresso nelle specifiche dalla funzione `adj`. Vale infatti, la seguente definizione matematica di `adj` e quindi di *adiacenza*:

$$G.adj(u) = \{v \mid (u,v) \in E\}$$

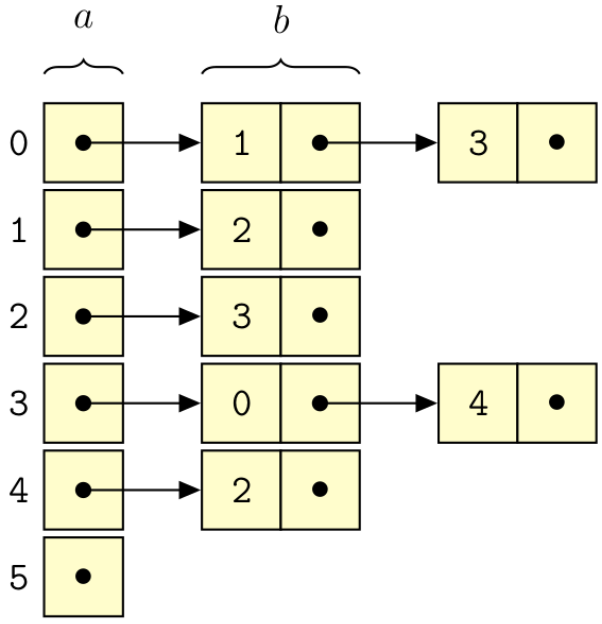
Quindi, nell'approccio basato su *liste*, viene realizzata una *lista* contenente tutti i *vertici* del *grafo*. Ogni *nodo* della *lista* poi, a cascata, contiene un riferimento ad un uno dei *vertici* che nel *grafo* sono *adiacenti* al *vertice* associato a quel *nodo* della *lista*.



(a) *Grafo orientato*

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

(b) *Matrice di adiacenza*



(c) *Lista di adiacenza*

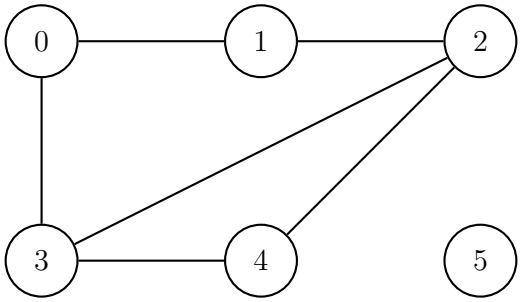
Fig. 7.4: Memorizzazione di un *grafo orientato*

Dall'immagine si può derivare facilmente il *costo di memorizzazione* nei due casi. Per le *matrici di adiacenza* lo spazio necessario dipende unicamente dal quadrato del numero di *nodi* e quindi

è $O(n^2)$, mentre per memorizzare una *lista di adiacenza*, lo spazio usato dipende sia dal numero di *nodi* che di *lati* e quindi è $O(a \cdot n + b \cdot m) = O(n + m)$.

Memorizzazione di un grafo non orientato Per i *grafi non orientati* non cambia molto, ma nel caso delle *matrici di adiacenza* possiamo dimezzare lo spazio grazie alla simmetria dell'*adiacenza*. Di conseguenza lo spazio necessario diventa $O(\frac{n(n-1)}{2})$, che però è comunque un $O(n^2)$.

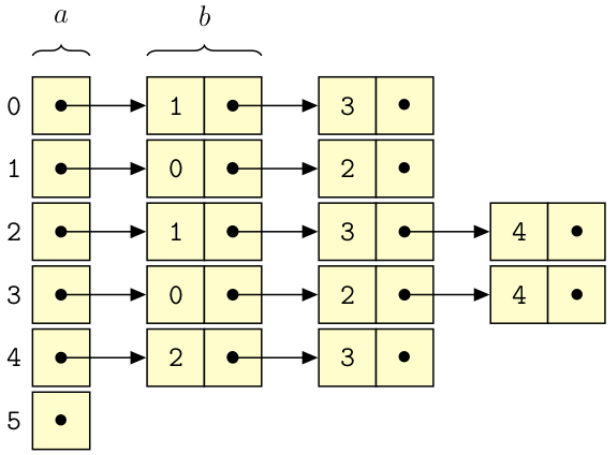
Per le *liste di adiacenza* invece, un *arco* (u, v) deve essere memorizzato sia tra i *nodi adiacenti* a u che a v. Di conseguenza, il numero di *archi* da memorizzare raddoppia e, conseguentemente, lo spazio necessario diventa $O(a \cdot n + 2b \cdot m)$ che anche sta volte è comunque $O(n + m)$.



(a) *Grafo non orientato*

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

(b) *Matrice di adiacenza*



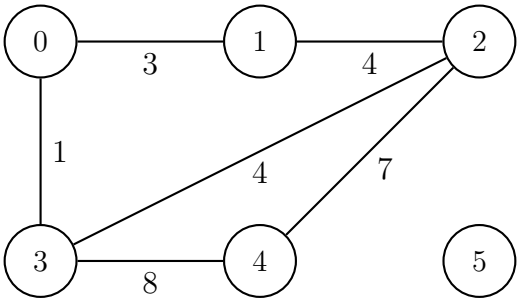
(c) *Lista di adiacenza*

Fig. 7.5: Memorizzazione di un *grafo non orientato*

Giunti a questo punto possiamo dire che, a parità di metodo di memorizzazione scelto, lo spazio necessario per memorizzare un *grafo orientato* non è diverso da quello che servirebbe se il *grafo* fosse *non orientato*.

Memorizzazione di un grafo pesato Un caso interessante è quello in cui si debba memorizzare un *grafo pesato*, cioè un *grafo* nel quale ad ogni *arco* è associato un costo (o un profitto). Matematicamente possiamo pensare il peso di un *arco* come l'applicazione di una funzione di peso del tipo: $w : V \times V \to \mathbb{R}$. L'unico caso particolare è quello in cui non esista un *arco* tra due *vertici* e quindi, a seconda del problema che si sta affrontando, si può decidere di assegnare un peso nullo o infinito.

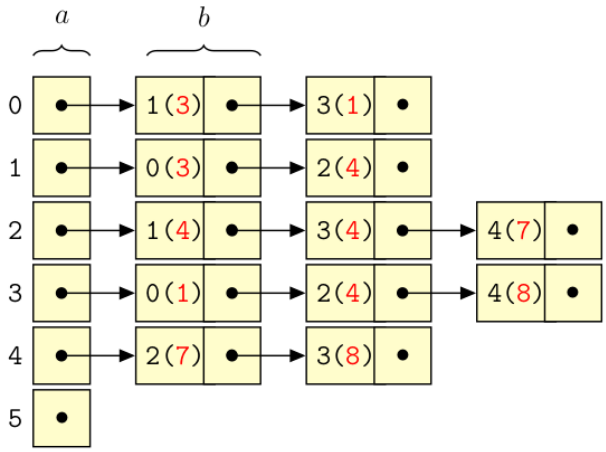
Ciò che cambia nella memorizzazione è che nella *matrice di adiacenza*, invece di indicare un valore binario, si indica il peso dell'*arco*, mentre nella *lista di adiacenza* si aggiunge il valore del peso all'interno di ogni *nodo*.



(a) *Grafo pesato non orientato*

	0	1	2	3	4	5
0		3	0	1	0	0
1			4	0	0	0
2				4	7	0
3					8	0
4						0
5						

(b) *Matrice di adiacenza*



(c) *Lista di adiacenza*

Fig. 7.6: Memorizzazione di un *grafo pesato non orientato*

Dettagli sull'implementazione Se non diversamente specificato, di seguito assumeremo che i *grafi* siano memorizzati come *vettori di adiacenza*, ovvero *liste* implementate tramite *vettori* (*dinamici* o *statici*). Inoltre, consideriamo $O(1)$ il costo per l'accesso alle informazioni di un oggetto di classe `NODE` e per l'inserimento e la rimozione di *nodi* e *archi* dai *grafi*. Ipotizzeremo inoltre, che dopo l'inizializzazione i *grafi* siano *statici* e che quindi non sia più possibile modificarli.

Fatte queste premesse, possiamo concludere questa sezione discutendo del *costo* complessivo di alcune operazioni basilari sui *grafi*.

Frammento 44 - Iterazione su *nodi* e *archi*.

```
% Iterazione su tutti i nodi
foreach (u ∈ G.V()) do
   { Operazioni sul nodo u }
```

```
% Iterazione su tutti i nodi e tutti gli archi
foreach (u ∈ G.V()) do
   { Operazioni sul nodo u }
   foreach (v ∈ G.adj(u)) do
      { Operazioni sull'arco (u, v) }
```

Matrici di adiacenza

- Lo spazio richiesto per la memorizzazione è $O(n^2)$;
- Il tempo richiesto per verificare se un *nodo* u è *adiacente* a un altro *nodo* v è $O(1)$;
- Il tempo necessario per iterare su tutti gli *archi* è $O(n^2)$

Liste di adiacenza

- Lo spazio richiesto per la memorizzazione è $O(n+m)$;
- Il tempo richiesto per verificare se un *nodo* u è *adiacente* a un altro *nodo* v è $O(n)$;
- Il tempo necessario per iterare su tutti gli *archi* è $O(n+m)$

In generale, le *matrici di adiacenza* sono adatte per memorizzare *grafi densi* e, al contrario, per *grafi sparsi* risulta più convenienti l'utilizzo delle *liste*.

7.2 Visite di un grafo

Definizione 68 - Visita di un grafo.

Dati, un grafo $G = (V, E)$ e un vertice $r \in V$ detto radice o sorgente, visitare un grafo significa visitare una e una sola volta tutti i nodi del grafo che possono essere raggiunti da r.

Come per gli *alberi*, esistono *visite in profondità* e in *ampiezza*. Un primo approccio alla *visita* di un *grafo* potrebbe essere il doppio ciclo visto in precedenza:

Frammento 45 - Visita scriteriata di un grafo.

```
% Iterazione su tutti i nodi e tutti gli archi
foreach (u ∈ G.V()) do
   { Operazioni sul nodo u }
   foreach (v ∈ G.adj(u)) do
      { Operazioni sull'arco (u,v) }
```

Questo approccio tuttavia non va bene in quanto non viene tenuto conto della struttura del *grafo* e, cosa più importante, non segue alcun criterio nell'ordine di visita dei *nodi* e degli *archi*. A dirla tutta, questo approccio viola la definizione stessa di *visita* in quanto non offre alcuna garanzia sul fatto che *nodi* e *archi* verranno visitati una sola volta.

7.2.1 Visita in ampiezza

Nella *visita in ampiezza* andiamo a visitare i *nodi* in ordine crescente di distanza dalla *radice*. Ciò può essere utile per calcolare il *cammino* più breve da r a tutti gli altri *nodi* e quindi riuscire a generare un *albero breadth-first*.

Definizione 69 - Albero breadth-first.

Un albero breadth-first è un albero contenente tutti i nodi raggiungibili da r e tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u nel grafo.

Dunque, per l'implementazione della *visita in ampiezza* possiamo partire da quanto fatto per gli *alberi generici* ipotizzando di trattare i *nodi adiacenti* come *figli*. Con le opportune modifiche si ottiene una funzione di questo tipo:

Frammento 46 - Implementazione errata visita in ampiezza di un grafo.

```
bfsTraversal(GRAPH G, NODE r)
   QUEUE q = Queue()
   q.enqueue(r)
   while (not q.isEmpty()) do
      NODE u = q.dequeue()
      { Visita il nodo u }
      foreach (v ∈ G.adj(u)) do
         q.enqueue(v)                          % Accoda tutti i nodi adiacenti a u
```

Questa implementazione, come la precedente, è errata e il motivo è che non solo lo stesso *nodo* può essere visitato più volte, ma la funzione non termina mai l'esecuzione. Infatti, se u e v sono *adiacenti*, quando viene visitato u viene messo in coda v e quando viene visitato v viene messo in coda u. È quindi necessario utilizzare un sistema che permetta di tenere traccia dei *nodi* già visitati.

Frammento 47 - Implementazione generica visita in ampiezza di un grafo.

```
graphTraversal(GRAPH G, NODE r)
   SET s = Set()                                          % Insieme generico
   s.insert(r)                                              % Da specificare
   { Marca il nodo r }
   while (s.size() > 0) do
      NODE u = s.remove()                                   % Da Specificare
      foreach (v ∈ G.adj(u)) do
         { Visita l'arco (u,v) }
         if (v non è stato ancora marcato) then
            { Marca il nodo v }
            s.insert(v)                                     % Da specificare
```

Frammento 48 - Implementazione visita in ampiezza di un grafo.

```
bfs(GRAPH G, NODE r)
   QUEUE q = Queue()
   q.enqueue(r)
   boolean[] visited = new boolean[G.size()]              % Vettore marcatore
   foreach (u ∈ G.V() - {r}) do                % Estrae tutti i nodi tranne r
      visited[u] = false
   visited[r] = true                                                % Marca r
   while (not q.isEmpty()) do
      NODE u = q.dequeue()
      { Visita il nodo u }
      foreach (v ∈ G.adj(u)) do
```

```
        { Visita l'arco (u, v) }
        if (not visited[v]) then  % Controlla se il nodo è già stato visitato
           visited[v] = true                                          % Marca v
           q.enqueue(v)
```

7.2.2 Visita in profondità

La *visita in profondità* viene usata per visitare tutti i *nodi* e non solo quelli raggiungibili da una singola sorgente come avviene con la *visita in ampiezza*. Proprio per questo, la ricerca restituisce una *foresta* di *alberi depth-first* e non un solo *albero*.

La funzione `dfs` per la *visita in profondità* può essere realizzata in modo ricorsivo o iterativo. In entrambi i casi si fa uso di una *pila*, ma nel caso della ricorsione la *pila* ha una dimensione massima definita dal sistema operativo, quindi per grafi di grandi dimensioni è bene considerare attentamente il tipo di implementazione più opportuno.

Frammento 49 - Implementazione ricorsiva visita in profondità di un grafo.

```
dfs(GRAPH G, NODE u, boolean[] visited)
   visited[u] = true
   { Visita il nodo u }                                          % Pre-order
   foreach (v ∈ G.adj(u)) do
      if (not visited[v]) then
         { Visita l'arco (u, v) }
         dfs(G, v, visited)
   { Visita il nodo u }                                         % Post-order
```

Frammento 50 - Implementazione iterativa visita in profondità di un grafo.

```
dfs(GRAPH G, NODE u)
   STACK s = Stack()
   s.push(r)
   boolean[] visited = new boolean[G.size()]
   foreach (u ∈ G.V()) do
      visited[u] = false
   while (not s.isEmpty()) do
      NODE u = s.pop()
      if (not visited[u]) then
         { Visita il nodo u }                                    % Pre-order
         visited[u] = true
         foreach (v ∈ G.adj(u)) do
            { Visita l'arco (u, v) }
            s.push(v)
```

È bene esplicitare alcuni particolari dell'implementazione iterativa. In particolare, un *nodo* può essere inserito più volte nella *pila*, in quanto il controllo sul marcatore viene effettuato al momento dell'estrazione e non dell'inserimento.

La *visita* in *post-order* prevede che alla scoperta di un *nodo*, questo venga inserito nella *pila* impostando un tag `discovery`. Alla successiva estrazione di quel *nodo*, il tag `discovery` viene cambiato in `finish` e il *nodo* viene inserito nuovamente nella *pila*. Alla terza estrazione viene effettuata la *visita*. L'utilizzo dei tag `discovery` e `finish` garantisce che un *nodo* con tag `finish` venga estratto soltanto dopo che tutti i suoi vicini sono già stati inseriti.

7.2.3 Complessità delle visite

Tutte e tre le implementazioni viste per i due tipi di *visite* hanno *complessità* $O(n+m)$. Nello specifico, l'implementazione iterativa della *dfs* costa $O(n+m)$ perché effettua $O(m)$ *visite* degli *archi*, $O(m)$ inserimenti ed estrazioni e $O(n)$ *visite* dei *nodi*.

7.3 Problemi sui grafi risolubili con visite in ampiezza

Le *visite in ampiezza* e in *profondità* sono solitamente sfruttate all'interno di altri algoritmi per risolvere problemi più complessi.

L'utilizzo più importante delle *visite in ampiezza* è nella ricerca del *cammino più breve* tra due *nodi*. Tuttavia, prima di arrivare a discutere quel problema, facciamo una piccola digressione e andiamo a vedere come calcolare la distanza tra un *nodo* e tutti gli altri.

7.3.1 Calcolo della distanza tra nodi

Per distanza tra *nodi* si intende il numero di *archi* che collegano un *nodo* a tutti gli altri. Il *nodo* di partenza ha distanza 0 da se stesso, i suoi vicini, ovvero i *nodi* ad esso *adiacenti*, hanno distanza 1 e così via. In generale, se un *nodo* u è *adiacente* ad un *nodo* che è a distanza k e, contemporaneamente, non è *adiacente* a nessun *nodo* che ha distanza inferiore a k dal *nodo* di partenza, allora u è a distanza $k+1$.

NB. Il concetto di distanze appena espresso deriva dai cosiddetti *Numeri di Erdös*. Erdös fu un matematico estremamente prolifico che scrisse più di 1500 articoli con più di 500 coautori. I *numeri di Erdös* descrivono la distanza di una persona da Erdös. Quindi, Erdös ha valore $erdos = 0$, i suoi coautori hanno $erdos = 1$ e, in generale, una persona coautore di qualcuno con $erdos = k$ e che non è coautore con nessuno che abbia $erdos < k$, ha $erdos = k+1$.

Quindi, l'algoritmo per il calcolo della distanza di un *nodo* dagli altri, non fa altro che calcolare il *numero di Erdös* di ogni *nodo*, e lo fa, sfruttando il meccanismo delle *visite in ampiezza*.

Frammento 51 - Implementazione distance per il calcolo dei numeri di Erdös.

```
distance(GRAPH G, NODE r, int[] distances)
   QUEUE q = Queue()
   q.enqueue(r)
   foreach (u ∈ G.V() - {r}) do
      distances[u] = ∞                  % Tutti i nodi non visitati sono a ∞
   distances[r] = 0
   while (not q.isEmpty()) do
      NODE u = q.dequeue()
      foreach (v ∈ G.adj(u)) do
         if (distances[v] == ∞) then      % Un nodo mai visitato ha erdos = ∞
            distances[v] = distance[u] + 1      % Se u ha erdos = k, v ha k+1
            q.enqueue(v)
```

7.3.2 Ricerca del cammino più breve fra due nodi

Partendo dalla funzione per il calcolo dei *numeri di Erdös* è possibile implementare una funzione che, dato un *nodo* r, costruisca l'*albero di copertura* con *radice* r. Quell'*albero* è realizzato tramite un *vettore dei padri* e può essere utilizzato per determinare, non solo la distanza di r da uno dei *nodi*, ma anche il *cammino semplice* più breve.

Frammento 52 - Implementazione distance per la ricerca dei cammini brevi.

```
distance(GRAPH G, NODE r, int[] distances, NODE[] parents)
   QUEUE q = Queue()
   q.enqueue(r)
   foreach (u ∈ G.V() - {r}) do
      distances[u] = ∞                    % Tutti i nodi non visitati sono a ∞
      parents[u] = nil          % Tutti i nodi non visitati non avranno un padre
   distances[r] = 0
   parents[r] = nil                     % r è la radice dell'albero di copertura
   while (not q.isEmpty()) do
      NODE u = q.dequeue()
      foreach (v ∈ G.adj(u)) do
         if (distances[v] == ∞) then       % Un nodo mai visitato ha erdos = ∞
            distances[v] = distance[u] + 1       % Se u ha erdos = k, v ha k + 1
            parents[v] = u                        % Popola il vettore dei padri
            q.enqueue(v)
```

Il *cammino semplice* tra due *nodi* può poi essere stampato con la seguente funzione.

Frammento 53 - Implementazione printPath per la stampa del cammino tra due nodi.

```
printPath(NODE r, NODE s, NODE[] parents)
   if (r == s) then                              % Cammino tra un nodo e se stesso
      print s
   else if (parents[s] == nil) then % Se il padre è nil, non esiste un cammino
      print "error"
   else
      printPath(r, parents[s], parents)
      print s
```

Complessità Entrambe le funzioni hanno *complessità lineare* $O(n + m)$ perché si basano sull'algoritmo per la *visita in ampiezza*.

7.4 Problemi sui grafi risolubili con visite in profondità

7.4.1 Ricerca delle componenti connesse

Definizione 70 - Raggiungibilità.

Un nodo v è raggiungibile da un nodo u se esiste almeno un cammino da u a v.

NB. Nei *grafi non orientati* il concetto di *raggiungibilità* è simmetrico.

Definizione 71 - Grafo connesso.

Un grafo non orientato $G = (V, E)$ è connesso se e solo se ogni suo nodo è raggiungibile da ogni altro nodo.

Definizione 72 - Sottografo.

$G' = (V', E')$ è un sottografo di $G = (V, E)$, ovvero $G' \subseteq G$ se e solo se $V' \subseteq V$ e $E' \subseteq E$.

Definizione 73 - Grafo massimale.

G' è massimale se e solo se non esiste un altro sottografo G'' di G che sia connesso e più grande di G', ovvero se non esiste un grafo G'' che soddisfi la seguente relazione:

$$G' \subseteq G'' \subseteq G$$

Definizione 74 - Componente connessa.

Un grafo $G' = (V', E')$ è una componente connessa di $G = (V, E)$ se e solo se è un sottografo connesso e massimale di G.

Fatte salve queste definizioni, possiamo dire che una *visita in profondità* può sia permetterci di capire se un grafo è *connesso* o meno, che di conoscere le sue *componenti connesse*. Per fare ciò, è sufficiente usare un *vettore* contenente gli identificatori delle *componenti connesse*. In particolare, il valore in posizione u del *vettore* è l'identificatore della *componente connessa* alla quale appartiene il *nodo* u.

Frammento 54 - Implementazione cc per la ricerca delle componenti connesse.

```
int[] cc(GRAPH G)
   int[] id = new int[G.size()]                    % Vettore degli identificatori
   foreach (u ∈ G.V()) do
      id[u] = 0
   int counter = 0                                 % Contatore Componenti Connesse
   foreach (u ∈ G.V()) do
      if (id[u] == 0) then
         counter = counter + 1
         ccdfs(G, counter, u, id)
   return id
% Funzione ausiliaria che marca tutti i nodi tra loro raggiungibili
ccdfs(GRAPH G, int counter, NODE u, int[] id)
   id[u] = counter
   foreach (v ∈ G.adj(u)) do
      if (id[v] == 0) then
         ccdfs(G, counter, v, id)
```

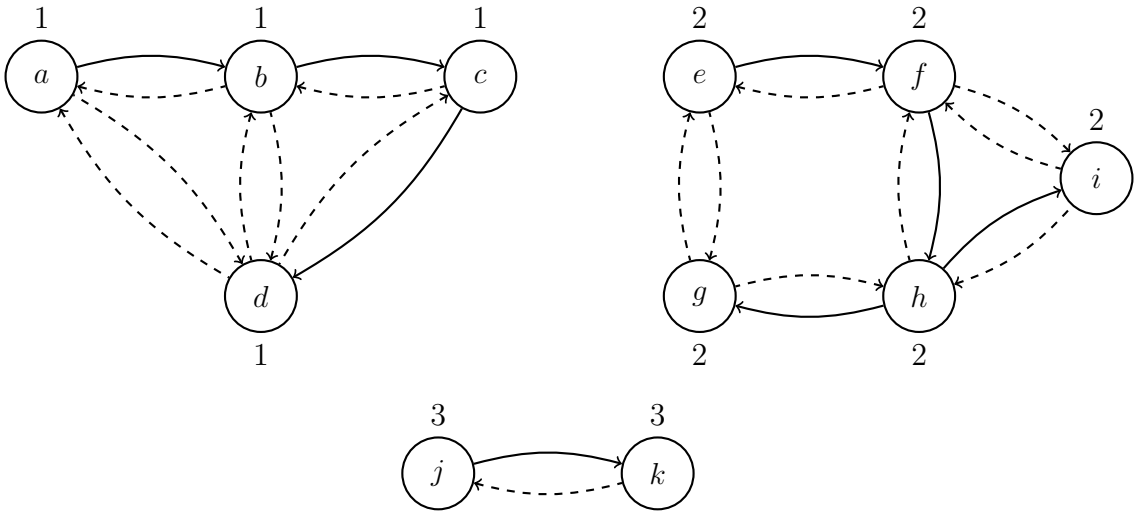


Fig. 7.7: *Grafo* con tre *componenti connesse*

NB. Nella figura di cui sopra è rappresentato un *grafo orientato*, ma poiché per ogni coppia di *nodi* esiste un *arco* in entrambe le direzioni, il *grafo* può anche essere visto come *non orientato*.

7.4.2 Verifica di esistenza di cicli nei grafi non orientati

Definizione 75 - Ciclo in un grafo non orientato.

In grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1 \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E \; \forall 0 \leq i \leq k-1$ e $u_0 = u_k$.

NB. Il vincolo $k > 2$ serve per escludere i *cicli* banali composti da coppie di *archi* (u, v) e (v, u) che sono onnipresenti nei *grafi non orientati*.

Definizione 76 - Grafo aciclico.

Un grafo non orientato che non contiene cicli è detto aciclico.

Definizione 77 - Grafo ciclico.

Un grafo che contiene almeno un ciclo è detto essere ciclico.

Riuscire a capire se un *grafo non orientato* contiene cicli, non è difficile. È sufficiente modificare l'algoritmo per la *visita in profondità* aggiungendo un parametro che ad ogni invocazione contenga un riferimento all'ultimo *nodo* visitato. Quindi, si esegue la normale *visita in profondità*, evitando però di visitare il *nodo* indicato dal parametro, e, se si arriva a un altro *nodo* già visitato, allora si è in presenza di un *ciclo*.

Frammento 55 - Implementazione hasCycle per la verifica di esistenza di cicli in grafi non orientati.

```
boolean hasCycle(GRAPH G)
    boolean[] visited = new boolean[G.size()]
    foreach (u ∈ G.V()) do
```

```
      visited[u] = false
   foreach (u ∈ G.V()) do
      if (not visited[u]) then
         if (hasCycleRec(G, u, nil, visited)) then
            return true
   return false

% Funzione ausiliaria
boolean hasCycleRec(GRAPH G, NODE u, NODE p, boolean[] visited)
   visited[u] = true
   foreach (v ∈ G.adj(u) - {p}) do    % Visita tutti i nodi adiacenti tranne p
      if (visited[v]) then
         return true
      else if (hasCycleRec(G, v, u, visited)) then
         return true
   return false
```

7.4.3 Verifica di esistenza di cicli nei grafi orientati

Definizione 78 - Ciclo in un grafo orientato.

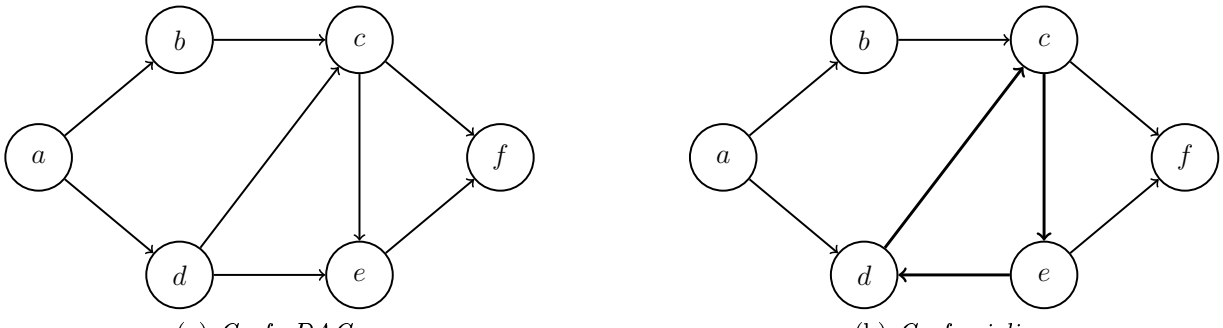
In un grafo orientato $G = (V, E)$, un ciclo C di lunghezza $k \geq 2$ è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E \; \forall 0 \leq i \leq k-1$ e $u_0 = u_k$.

NB. Nei *grafi orientati* i *cicli* banali formati da coppie di *archi* (u, v) e (v, u) sono accettati.

NB. I *cicli* in cui tutti i *nodi* ad eccezione del primo e l'ultimo sono distinti sono detti *cicli semplici*.

Definizione 79 - Grafi orientati aciclici - DAG.

Un grafo orientato che non contiene cicli è detto essere un DAG (Directed Acyclic Graph).



(a) *Grafo DAG* (b) *Grafo ciclico*

Fig. 7.8: *Grafi DAG* VS *grafi ciclici*

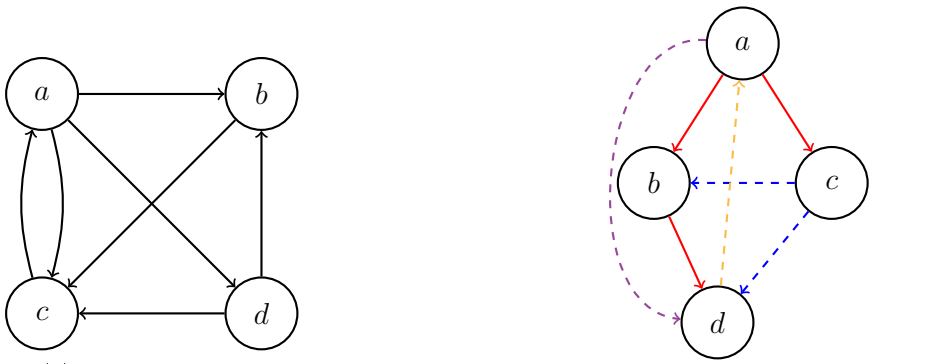
Verificare l'esistenza di *cicli* all'interno di *grafi orientati* non è semplice come nei *grafi non orientati*, infatti non è possibile usare l'algoritmo visto in precedenza, ma ne serve uno ad-hoc.

Definizione 80 - Archi degli alberi di copertura.

Ogni volta che si esamina un arco da un nodo marcato a uno non marcato, tale arco viene detto arco dell'albero.

Quando si effettua una *visita*, l'*albero di copertura* risultante non contiene tutti gli *archi* del *grafo*. Gli *archi* (u, v) non compresi nell'*albero di copertura* T possono essere classificati in tre modi:

- Se u è un antenato di v in T, (u, v) è detto *arco in avanti*;
- Se u è un discendente di v in T, (u, v) è detto *arco all'indietro*;
- Se (u, v) non è né un *arco in avanti*, né un *arco all'indietro*, è un *arco di attraversamento*;



(a) *Grafo orientato* (b) *Albero di copertura con radice a*

Fig. 7.9: *Classificazione degli archi*

Nella figura si vede l'*albero di copertura* ottenuto dopo aver effettuato una *visita in profondità* partendo dal *nodo a*. Gli *archi* rossi sono gli *archi dell'albero di copertura*, mentre gli *archi* (a, c) e (c, a) sono, rispettivamente, un *arco in avanti* e un *arco all'indietro*. Infine, gli *archi* uscenti da d sono *archi di attraversamento*.

Frammento 56 - Implementazione dfs_schema per la classificazione degli archi.

```
dfs_schema(GRAPG G, NODE u, int &time, int[] dt, int[] ft)
   time = time + 1                              % Aggiorna il contatore del tempo
   dt[u] = time                          % Imposta il tempo di scoperta del nodo u
   foreach (v ∈ G.adj(u)) do
      if (dt[v] == 0) then                                    % Arco dell'albero
         { Visita l'arco (u, v) }
         dfs_schema(G, v, time, dt, ft)
      else if (dt[u] > dt[v] and ft[v] == 0) then          % Arco all'indietro
         { Visita l'arco (u, v) }
      else if (dt[u] < dt[v] and ft[v] ≠ 0) then             % Arco in avanti
         { Visita l'arco (u, v) }
      else                                                % Arco di attraversamento
         { Visita l'arco (u, v) }
   time = time + 1                              % Aggiorna il contatore del tempo
   ft[u] = time                          % Imposta il tempo di fine visita del nodo u
```

In questa funzione, il *tempo di scoperta* viene impostato quando per la prima volta si visita un *nodo*, mentre il *tempo di fine visita* quando sono già stati visitati tutti i *nodi adiacenti*.

Definizione 81 - Teorema di caratterizzazione delle coppie di nodi.

Data una visita in profondità di un grafo $G = (V, E)$, per ogni coppia di nodi $u, v \in V$, vale una delle seguenti condizioni:

- *Gli intervalli $[dt[u], ft[u]]$ e $[dt[v], ft[v]]$ non si intersecano, né sovrappongono in alcun modo dunque, u e v, nella foresta depth-first, non sono discendenti l'uno dell'altro;*
- *L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$ dunque, in uno degli alberi depth-first, u è un discendente di v;*
- *L'intervallo $[dt[u], ft[u]]$ contiene $[dt[v], ft[v]]$ dunque, in uno degli alberi depth-first, u è un antenato di v;*

Grazie a questa definizione possiamo dire che, dati due *nodi* u e v, se vale la seguente:

$$[dt[u], ft[u]] \subset [dt[v], ft[v]]$$

valgono le seguenti affermazioni:

- Il *nodo* v è un *antenato* di u;
- Il *nodo* u è un *discendente* di v;
- L'*arco* (v, u), se non è un *arco dell'albero*, è un *arco in avanti*;
- L'*arco* (u, v), se non è un *arco dell'albero*, è un *arco all'indietro*;

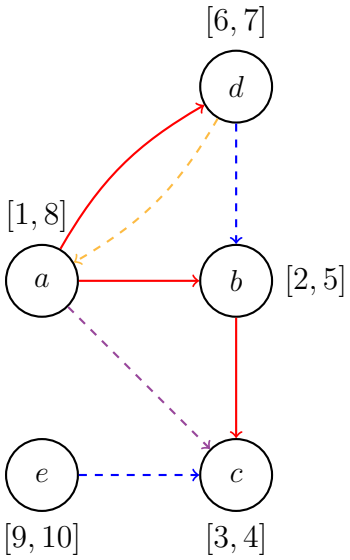


Fig. 7.10: Classificazione degli *archi* e intervalli di tempo

Nella figura di cui sopra, si è effettuata una *visita in profondità* a partire dal *nodo* a e si sono visitati, nell'ordine, i *nodi* a, b, c, d. Gli *archi* (a, b), (b, c) e (a, d) sono *archi dell'albero* poiché quando b, c e d sono stati scoperti avevano certamente $dt = 0$.

L'intervallo associato al *nodo* d è contenuto in quello associato al *nodo* a, quindi l'*arco* (d, a), poiché non è un *arco dell'albero*, è un *arco all'indietro*. In realtà, quando l'algoritmo

arriva a visitare il *nodo* d, $ft[a]$ non è ancora stato impostato, ma dato che la condizione `dt[u] > dt[v] and ft[v] == 0` con $u = d$ e $v = a$ risulta verificata, l'*arco* (d, a) è effettivamente un *arco all'indietro*.
L'*arco* (a, c), invece, è un *arco in avanti* perché verifica la relazione sugli intervalli, ovvero $[dt[c], ft[c]] \subset [dt[a], ft[a]]$, ma anche perché durante l'esecuzione dell'algoritmo la condizione `dt[u] < dt[v] and ft[v]` $\neq$ `0` con $u = a$ e $v = c$ viene verificata.

Infine, il *nodo* e non appartiene alla *componente connessa* di a, quindi non fa nemmeno parte dello stesso *albero di copertura*. L'*arco* (e, c) non soddisfa nessuna condizione quindi è un *arco di attraversamento*.

Fatta questa digressione, possiamo ritornare al problema originale: verificare l'esistenza di *cicli* in un *grafo orientato*. Enunciamo il seguente teorema.

Definizione 82 - Teorema di esistenza dei cicli.

Un grafo orientato è aciclico se e solo se non esistono archi all'indietro.

Dimostrazione.

- ($\Rightarrow$): si supponga esista un *ciclo*. Sia (v, u) un *arco* del *ciclo* e sia u il primo *nodo* ad essere visitato. Prima o poi verrà visitato il *cammino* che connette u a v e da v verrà scoperto l'*arco all'indietro* (v, u);
- ($\Leftarrow$): se esiste un *arco all'indietro* (u, v), dove v è un antenato di u, allora esistono un *cammino* da v a u e un *arco* da u a v, ovvero un *ciclo*;

□

Quindi, per verificare l'esistenza di un *ciclo* è sufficiente cercare un *arco all'indietro*. Se ne esiste almeno uno, esisterà sicuramente un anche un *ciclo*.

Frammento 57 - Implementazione hasCycle per la verifica di esistenza di cicli in un grafo orientato.

```
boolean hasCycle(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
   time = time + 1
   foreach (v ∈ G.adj(u)) do
      if (dt[v] == 0) then
         return hasCycle(G, v, time, dt, ft)
      else if (dt[u] > dt[v] and ft[v] ≠ 0) then
         return true
   time = time + 1
   ft[u] = time
   return false
```

7.4.4 Realizzare un ordinamento topologico di un grafo orientato

---

Definizione 83 - Ordinamento topologico.

Dato un grafo orientato aciclico (DAG) G, un ordinamento topologico di G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.

NB. Se il *grafo* è *aciclico* esistono più ordinamenti possibili, ma se esiste anche solo un *ciclo* non ne esiste alcuno.

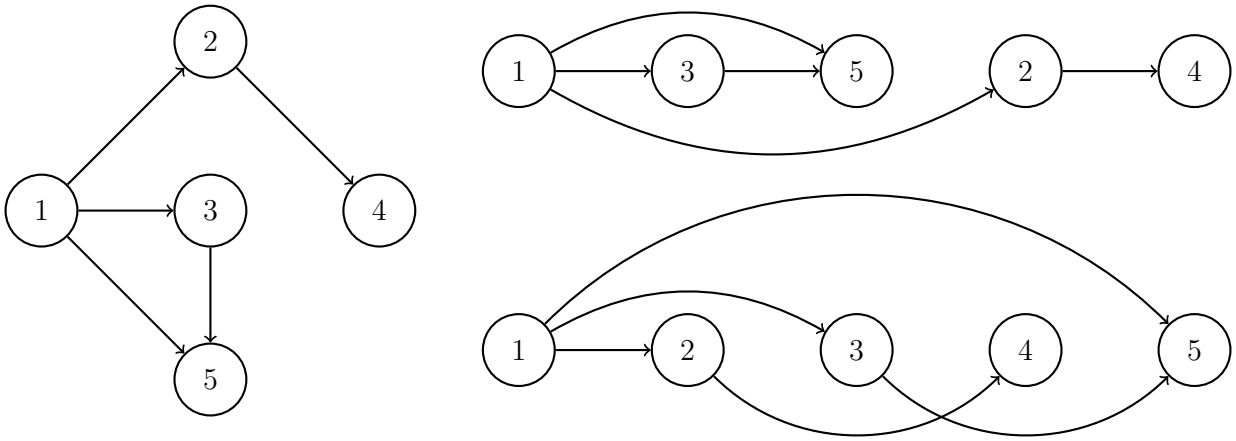
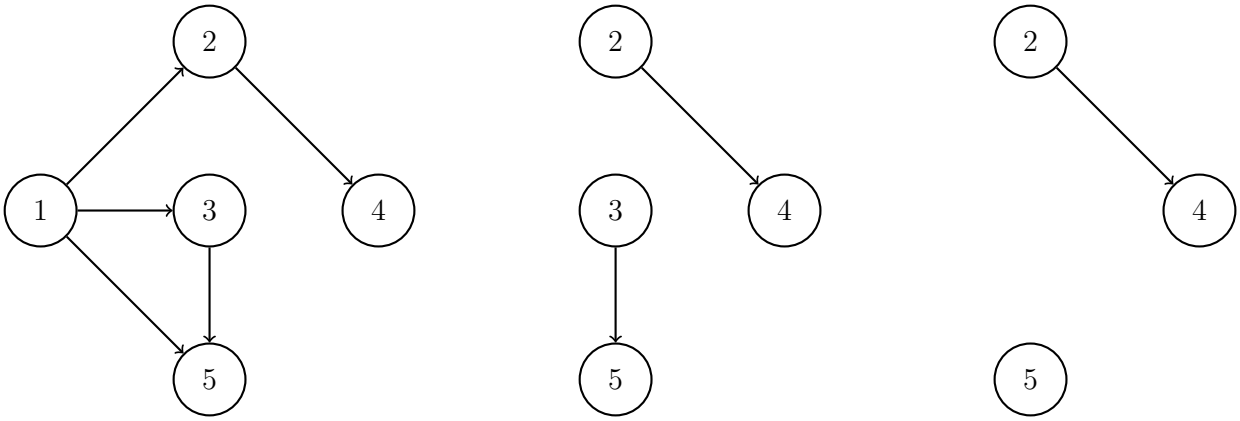


Fig. 7.11: Possibili *ordinamenti topologici* di un *grafo*

Approccio naive Un primo approccio a questo problema potrebbe essere quello in cui, scelto un *nodo* privo di *archi entranti*, lo si aggiunge all'ordinamento e si rimuovono tutti i suoi *archi*. Quindi, si ripete l'operazione per tutti gli altri *nodi*.



(a) Output: (b) Output: 1 (c) Output: 1 3

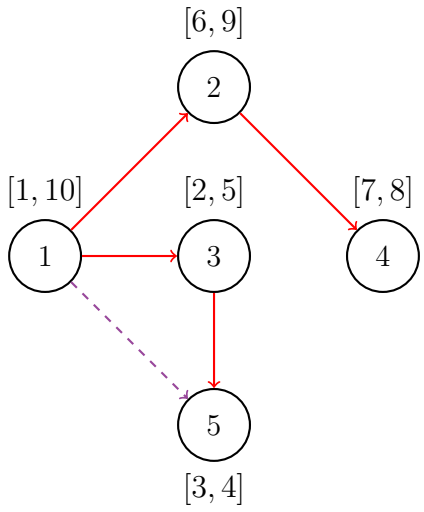
(d) Output: 1 3 5 (e) Output: 1 3 5 2 (f) Output: 1 3 5 2 4

Fig. 7.11: Esecuzione dell'algoritmo "naive" per l'*ordinamento topologico*

Approccio efficiente Un approccio migliore prevede di effettuare una *visita in Post-order* inserendo tutti i *nodi* in una *pila* a mano a mano che li si visita. L'utilizzo della *visita in Post-order* fa si che un *nodo* venga inserito solo quando tutti i suoi *discendenti* sono già stati scoperti ed aggiunti alla *pila*, e, l'utilizzo della *pila* permette di estrarre i *nodi* in ordine inverso a quello di *inserimento*. Ciò, permette di ottenere un corretto *ordinamento topologico*.

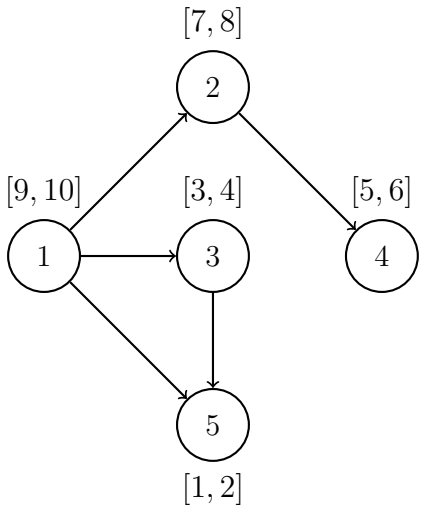
Frammento 58 - Implementazione topSort per l'ordinamento topologico di un grafo DAG.

```
STACK topSort(GRAPH G)
   STACK s = Stack()
   boolean[] visited = new boolean[G.size()]
   foreach (u ∈ G.V()) do
      visited[u] = false
   foreach (u ∈ G.V()) do
      if (not visited[u]) then
         ts-dfs(G, u, visited, s)          % Effettua la visita in Post-order
   return s

% Funzione ausiliaria
ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK s)
   visited[u] = true
   foreach (v ∈ G.adj(u)) do
      if (not visited[v]) then
         ts-dfs(G, v, visited, s)
   s.push(u)
```



`Stack = {1, 2, 4, 3, 5}`

(a) *Ordinamento topologico* partendo da 1



`Stack = {1, 2, 4, 3, 5}`

(b) *Ordinamento topologico* partendo da 5

Fig. 7.12: Esempi di *ordinamenti topologici*

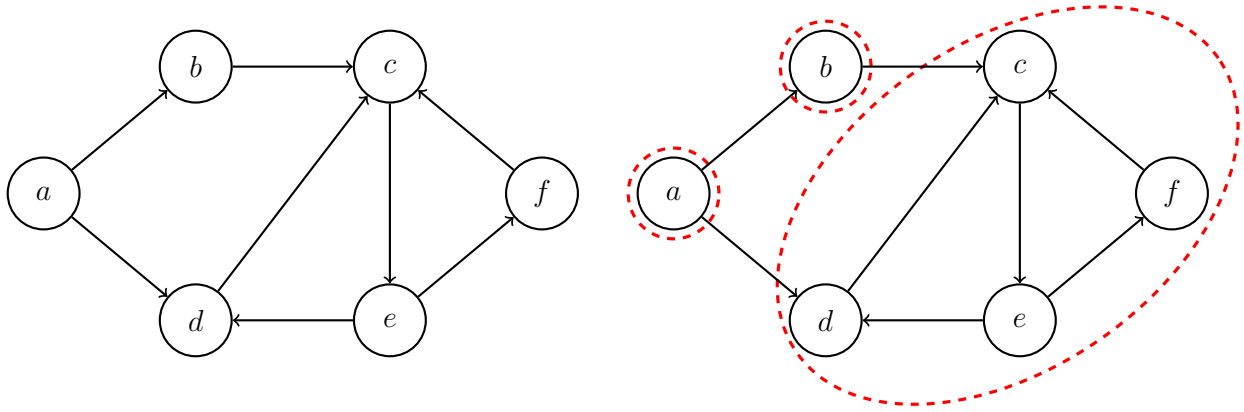
7.4.5 Ricerca delle componenti fortemente connesse

Definizione 84 - Grafo fortemente connesso.

Un grafo orientato $G = (V, E)$ è fortemente connesso se e solo se ogni suo nodo è raggiungibile da ogni altro nodo.

Definizione 85 - Componente fortemente connessa.

Un grafo $G' = (V', E')$ è una componente fortemente connessa di $G = (V, E)$ se e solo se G' è un sottografo fortemente connesso e massimale di G.



(a) *Grafo orientato* (b) *Componenti fortemente connesse*

Fig. 7.13: *Componenti fortemente connesse* di un *grafo*

Una soluzione ingenua potrebbe essere quella di applicare `cc` al *grafo*, ma purtroppo il risultato dipenderebbe dal *nodo* di partenza. Ad esempio, nella figura di cui sopra, se si applicasse `cc` partendo da a si otterrebbe un'unica componente connessa, mentre applicandola da b se ne otterrebbero due. La soluzione corretta è quella fornita dall'*Algoritmo di Kosaraju*.

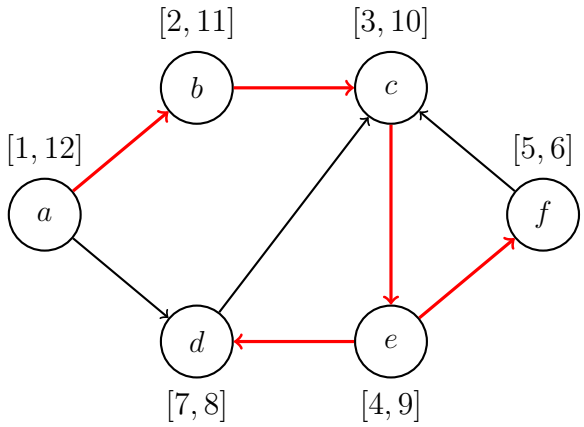
Frammento 59 - Implementazione Algoritmo di Kosaraju.

```
int[] scc(GRAPH G)
    STACK S = topsort(G)                          % Prima visita
    GRAPH G^T = traspose(G)                % Calcolo del grafo trasposto
    return cc(G^T, S)                             % Seconda visita
```

Utilizzo della topsort La prima cosa che dovrebbe farci storcere il naso è l'utilizzo della funzione `topsort` per *grafi* non *aciclici*. Quando abbiamo parlato di *ordinamento topologico* avevamo infatti posto come condizione l'*aciclicità* dei *grafi*, vincolo necessario per poter definire un ordine di visita dei *nodi*. In questo caso però, una *componente connessa* non banale, che quindi include più di un *nodo*, è sicuramente un *ciclo* e l'ordine di visita dei *nodi* in un *ciclo* non ha importanza.

Di conseguenza, applicando la `topsort` su un *grafo* generico siamo sicuri che:

- Se l'*arco* (u, v) non appartiene a un *ciclo*, il *nodo* u compare prima di v nell'ordinamento;
- Se l'*arco* (u, v) appartiene a un *ciclo*, i *nodi* compaiono in un qualche ordine ininfluente;




```
Stack = {a, b, c, e, d, f}
```

Fig. 7.14: *Ordinamento topologico* di un *grafo* generico

NB. L'utilizzo della `topsort` permette di ottenere i *nodi* di un *grafo* in ordine decrescente di tempo di fine.

Utilizzo della traspose

Definizione 86 - Grafo trasposto.

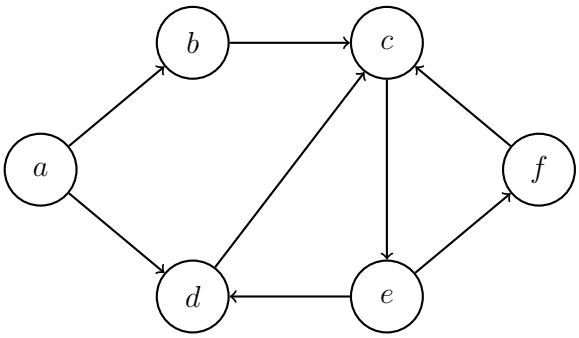
Dato un grafo orientato $G = (V, E)$, il grafo trasposto $G_T = (V, E_T)$ ha gli stessi nodi e archi di G, ma gli archi sono orientati in senso opposto. Cioè:

$$E_T = \{(u, v) \,|\, (v, u) \in E\}$$

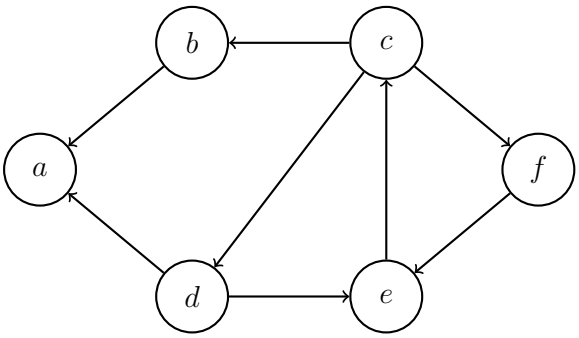
Frammento 60 - Implementazione traspose per la generazione di un grafo trasposto.

```
GRAPH traspose(GRAPH G)
   GRAPH G^T = Graph()                              % Crea un grafo vuoto
   foreach (u ∈ G.V()) do
      G^T.insertNode(u)                          % Aggiunge a G^T tutti i nodi di G
   foreach (u ∈ G.V()) do
      foreach (v ∈ G.adj(u)) do
         G^T.insertEdge(v, u)        % Aggiunge a G^T gli archi di G invertendoli
```

Il costo di questa funzione è *lineare*, cioè $O(n+m)$, perché vengono aggiunti tutti i *nodi* e tutti gli *archi*.



(a) *Grafo* originale



(b) *Grafo trasposto*

Fig. 7.15: *Grafo orientato* e relativo *grafo trasposto*

Utilizzo della cc Usiamo un'implementazione della `cc` diversa rispetto a quella vista precedentemente. In particolare, l'ordine di visita dei *nodi* viene stabilito in base all'ordine in cui questi sono stati inseriti nella *pila* ricevuta come parametro.

Frammento 61 - Implementazione alternativa di cc.

```
int[] cc(GRAPH G, STACK S)
   int[] id = new int[G.size()]
   foreach (u ∈ G.V()) do
      id[u] = 0
   int counter = 0
   while (not S.isEmpty()) do
      NODE u = S.pop()
      if (id[u] == 0) then
         counter = counter + 1
   ccdfs(G, counter, u, id)
   return id

% L'implementazione della funzione ausiliaria non cambia
ccdfs(GRAPH G, int counter, NODE u, int[] id)
   id[u] = counter
   foreach (v ∈ G.adj(u)) do
      if (id[v] == 0) then
         ccdfs(G, counter, v, id)
```

Quindi, per ricercare le *componenti fortemente connesse*, applichiamo l'algoritmo di ricerca delle *componenti connesse* al *grafo trasposto* e procediamo a visitare i *nodi* nell'ordine stabilito dall'*ordinamento topologico* del *grafo* originale.

Questo significa che, nell'esempio che stiamo considerando, la `cc` inizia esaminando il *nodo* a. Poiché nel *grafo trasposto* a non ha alcun *arco uscente*, quel singolo *nodo* viene marcato come appartenente alla *componente connessa* 1. Alla seconda iterazione del ciclo `while` viene estratto il *nodo* b. Nel *grafo trasposto* b è adiacente soltanto ad a che è già stato marcato, quindi b viene assegnato alla *componente connessa* banale con identificativo 2.

Infine viene estratto c che fa parte di due *cicli*. Poiché da c possono essere raggiunti tutti i *nodi* che fanno parte di quei *cicli*, i *nodi* c, e, f, d, vengono assegnati a un'unica *componente connessa*.

Al termine dell'esecuzione di `cc` siamo riusciti ad'identificare con successo tutte e tre le *componenti fortemente connesse* del *grafo*.

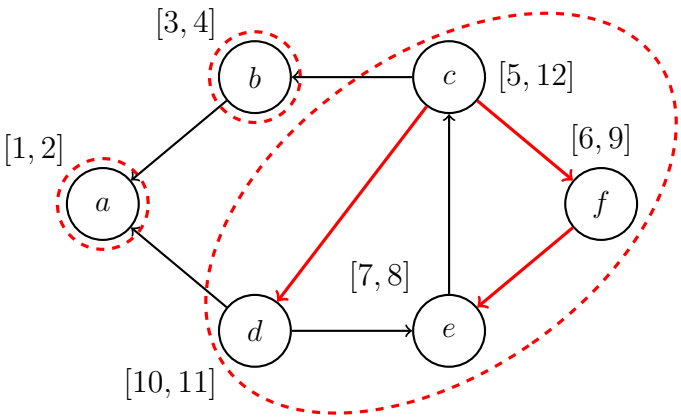


Fig. 7.16: *Componenti fortemente connesse* del *grafo*

Costo computazionale Poiché tutte e tre le funzioni che vengono richiamate dalla `scc` hanno costo $O(n+m)$ anche la `scc` ha la stessa *complessità*.

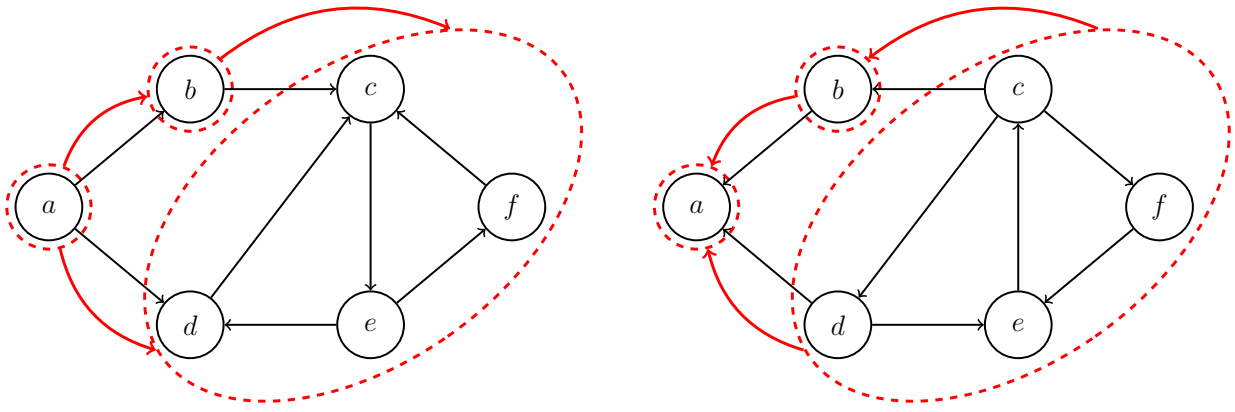
Dimostrazione di correttezza

Definizione 87 - Grafo delle componenti.

Dato un grafo $G = (V, E)$, il suo grafo delle componenti $C(G) = (V_C, E_C)$ è un grafo tale che:

- $V_C = \{C_1, C_2, \ldots, C_k\}$, *dove C_i è la i-esima componente fortemente connessa di G;*
- $E_C = \{(C_i, C_j) \mid \exists\, (u_i, u_j) \in E : u_i \in C_i \wedge u_j \in C_j\}$

NB. Dati un *grafo* G e il suo *trasposto* G^T, vale $C\left(G^T\right) = [C\,(G)]^T$.



(a) *Grafo delle componenti* del *grafo* originale (b) *Grafo delle componenti* del *grafo trasposto*

Fig. 7.17: *Grafi delle componenti* del *grafo* d'esempio e del suo *trasposto*

NB. I *grafi delle componenti* sono sempre *aciclici*.

Definizione 88 - Relazione tra discovery e finish time nei grafi delle componenti.

Dato un nodo C di un grafo delle componenti, valgono le seguenti equivalenze:

$$dt(C) = \min\{dt(u) \mid u \in C\}$$

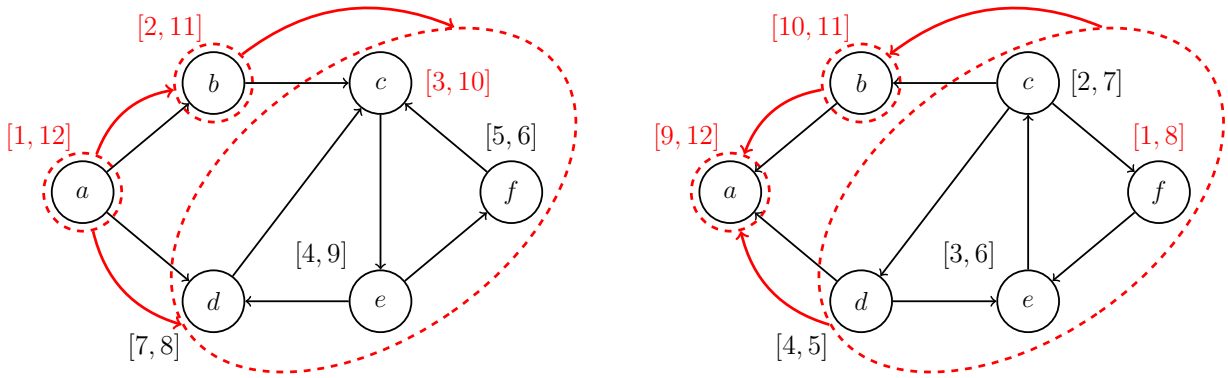
$$ft(C) = \max\{ft(u) \mid u \in C\}$$

NB. Il *discovery* e *finish time* di una *componente fortemente connessa* corrispondono al *discovery* e *finish time* del primo *nodo* visitato in quella *componente*.

Definizione 89 - Teorema di ordinamento delle componenti fortemente connesse.

Siano C e C' due distinte componenti fortemente connesse del grafo orientato $G = (V, E)$. Se esiste un arco $(C, C') \in E_C$, allora $ft(C) > ft(C')$.

NB. Ovviamente nel *grafo trasposto* G_T, le relazioni d'ordine sui *finish time* sono invertite.



(a) *Componenti* nel *grafo* originale

(b) *Componenti* nel *grafo trasposto*

Fig. 7.18: *Discovery* e *finish time* delle *componenti*

NB. Quanti si considera un *arco* tra due *componenti fortemente connesse*, in realtà, bisogna considerare l'*arco* che collega i due *nodi* appartenenti ad una e all'altra *componente*.

Nelle immagini sopra si vede bene come le *componenti* banali a e b siano collegate dall'*arco* (a, b) nel *grafo* originale e da (b, a) nel *grafo trasposto*. Quindi, nel primo caso C e C' sono rispettivamente le *componenti* a e b e infatti vale $ft(a) > ft(b)$. Nel *trasposto* invece, C e C' sono invertite, ovvero C' contiene a e C contiene b. Di conseguenza, la relazione d'ordine sui *finish time* è invertita: $ft(C) < ft(C')$.

A questo punto abbiamo tutti gli strumenti per dimostrare la correttezza dell'*Algoritmo di Kosaraju*.

Dimostrazione. Se le *componenti* C_x e C_y sono connesse mediante un *arco* $(x, y) \in E_T$, sicuramente $ft(C_x) < ft(C_y)$. Di conseguenza, la visita di C_y inizierà prima di quella di C_x. Inoltre, poiché non esistono *cammini* da C_y a C_x, perché altrimenti il *grafo* sarebbe *ciclico*, la visita di C_y non raggiungerà mai *nodi* appartenenti a C_x.

In altre parole, la funzione `cc` assegnerà correttamente tutti gli identificatori delle *componenti*. □

Capitolo Nr.8

Hashing

Nei capitoli precedenti abbiamo affrontato il concetto di Dizionario e abbiamo visto come la scelta della *struttura dati* si ripercuota sulla *complessità* delle operazioni.

In particolare, abbiamo osservato le seguenti *complessità*:

Operazione	**Array non ordinato**	**Array ordinato**	**Lista**	**Alberi RB**
`insert()`	$O(1)$, $O(n)$	$O(n)$	$O(1)$, $O(n)$	$O(\log n)$
`lookup()`	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
`remove()`	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
`foreach`	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Esiste una *struttura dati* che ci permetta di fare meglio di così? La risposta è sì e si chiama *hash table* (o *tabella hash*) e ci consente di ottenere *complessità* $O(1)$ in tutte le operazioni (tranne ovviamente nel `foreach`).

Vediamo quindi che cos'è una *tabella hash*.

Definizione 90 - Tabella hash.

Una tabella hash è una struttura dati dinamica che permette di memorizzare associazioni chiave-valore. L'insieme delle possibili chiavi è rappresentato da un insieme universo $\mathcal{U}$ di dimensione u e le chiavi sono memorizzati in un vettore `T[0..m-1]` di dimensione m detto tabella hash.

Ogni elemento di $\mathcal{U}$ è associato ad una sola posizione all'interno della tabella e questa associazione è definita da una *funzione hash*.

Definizione 91 - Funzione hash.

Una funzione hash è una funzione definita come $h : \mathcal{U} \to \{0, \ldots, m-1\}$.

Le *funzioni hash* sono tali per cui la coppia chiave-valore $\langle k, v \rangle$ è memorizzata nella tabella alla posizione $h(k)$.

NB. L'insieme universo $\mathcal{U}$ ha potenzialmente una dimensione infinità, mentre la *tabella hash* ha una dimensione limitata, quindi è sicuro che più elementi saranno associati alla stessa posizione.

Definizione 92 - Collisione.

Quando due o più chiavi nel dizionario hanno lo stesso valore hash diciamo che è avvenuta una collisione.

Idealmente, vorremmo realizzare *funzioni hash* che non generino collisioni.

Tabelle ad accesso diretto In alcuni casi in cui l'insieme delle chiavi è noto a priori ed è un sottoinsieme "piccolo" di $\mathbb{Z}^+$ (e.g insieme dei giorni dell'anno) è possibile definire la *funzione hash* come la funzione identità, ovvero:

$$h(k) = k \quad \forall\, k \in \mathcal{U} \subset \mathbb{Z}^+$$

e definire una *tabella hash* di dimensione $m = u$.

In situazioni di questo tipo, ovviamente, non si genereranno mai *collisioni*, ma purtroppo non sono quasi mai praticabili perché se u è molto grande è richiesto uno spazio eccessivo, mentre se vengono usate poche delle possibili chiavi si va a sprecare memoria.

8.1 Caratteristiche e implementazioni di funzioni hash

Andiamo a vedere come possono essere definite le *funzioni hash* e quali sono le loro caratteristiche.

Definizione 93 - Funzione hash perfetta.

Una funzione hash h si dice perfetta se è iniettiva, ovvero se vale:

$$\forall\, k_1,\, k_2 \in \mathcal{U} : k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$$

NB. Le funzioni iniettive non danno mai origine a *collisioni*.

Per le stesse ragioni espresse per le *tabelle ad accesso diretto*, è molto difficile poter ottenere una *funzione hash perfetta* e inoltre, lo spazio delle chiavi è spesso grande, sconosciuto e sparso.

Quindi, se non possiamo evitare che si generino *collisioni* cerchiamo almeno di minimizzarne il numero. Per farlo cerchiamo *funzioni hash* che distribuiscano in modo uniforme le chiavi all'interno della *tabella hash*. Ma cosa significa "in modo uniforme"?

Definizione 94 - Uniformità semplice.

Siano $P(k)$ la probabilità che una chiave k sia inserita nella tabella e $Q(i)$ la probabilità che una chiave venga inserita nella posizione i, cioè:

$$Q(i) = \sum_{k \in \mathcal{U} : h(k) = i} P(k)$$

Una funzione hash gode di uniformità semplice se vale:

$$Q(i) = \frac{1}{m} \quad \forall\, i \in [0, \ldots, m-1]$$

Esempio 19 - Funzione hash con uniformità semplice.

Se $\mathcal{U}$ è l'insieme dei numeri reali $[0, 1[$ e ogni chiave ha la stessa probabilità di essere scelta, la funzione hash $h(k) = \lfloor km \rfloor$ gode di uniformità semplice.

Notiamo che poter ottenere una *funzione hash* con *uniformità semplice* dobbiamo conoscere la distribuzione delle probabilità P e quindi si ripropongono gli stessi problemi di prima: insieme delle chiavi sconosciuto in principio. Per questo motivo, nella realtà si usano tecniche "euristiche".

8.1.1 Realizzare una funzione hash

Assumiamo che ogni chiave sia traducibile in valori numeri interpretando la propria rappresentazione in memoria come un valore intero.

Ad esempio, ipotizziamo di avere a disposizione le seguenti funzioni per trasformare una stringa in un intero:

Frammento 62 - Funzioni per la manipolazioni di chiavi stringhe.

```
% Restituisce il valore ordinale binario di c in una qualche codifica[1]
bin bin(char c)
% Restituisce la rappresentazione binaria di una chiave k, concatenando
% i valori binari dei caratteri che la compongono
bin bin(string k)
% Restituisce il valore numerico associato al numero binario b
int int(bin b)
% Restituisce la traduzione in valore intero di una chiave k
int int(string k)
    return int(bin(k))
```

Esempio 20 - Tradurre una chiave stringa in un intero.

Calcolare il valore intero corrispondente alla chiave $k =$*"DOG" di tipo* **string**.

Applichiamo la funzione $bin(\text{"}DOG\text{"})$ *e calcoliamo i valori binari dei caratteri della stringa "DOG":*

$$ord('D') = 01000100 \quad ord('O') = 01001111 \quad ord('G') = 01000111$$

Quindi, restituiamo la concatenazione dei tre valori binari:

$$01000100\ 01001111\ 01000111$$

Ora, usando la funzione $int(\text{"}DOG\text{"})$ *traduciamo in intero la stringa:*

$$int(01000100) = 68 \cdot 256^2 \quad int(01001111) = 79 \cdot 256^1 \quad int(01000111) = 71 \cdot 256^0$$

Terminiamo restituendo la somma dei tre valori:

$$68 \cdot 256^2 + 79 \cdot 256^1 + 71 \cdot 256^0 = 4476743$$

Come facciamo a trasformare quel numero in un valore compreso tra 0 e $m - 1$?

8.1.2 Metodo dell'estrazione

Il primo metodo sarebbe quello di scegliere $m = 2^p$ e definire la *funzione hash* come la traduzione in intero di p bit scelti tra i bit di $bin(k)$, ovvero:

$$h(k) = bin(b) \quad \text{dove} \quad b \subset bin(k) : \#b = p$$

Il problema di questa tecnica è che ha un alta probabilità di generare *collisioni*.

[1] D'ora in avanti useremo la codifica binaria `ASCII` a 8 bit

Esempio 21 - Estrazione dei bit meno significativi.

Sia $m = 2^p = 2^{16}$ e si supponga che $bin(k)$ restituisca i 16 bit meno significativi della rappresentazione.

$$\begin{array}{rcl} bin(\text{“}Alberto\text{”}) & = & 01110100\ 01101111 \\ bin(\text{“}Roberto\text{”}) & = & 01110100\ 01101111 \end{array} \quad \Rightarrow \quad \begin{array}{rcl} h(\text{“}Alberto\text{”}) & = & 29\,807 \\ h(\text{“}Roberto\text{”}) & = & 29\,807 \end{array}$$

Esempio 22 - Estrazione di un gruppo arbitrario di bit.

Sia $m = 2^p = 2^{16}$ e si supponga che $bin(k)$ restituisca i 16 bit che vanno dal quinto al ventesimo (estremi inclusi).

$$\begin{array}{rcl} bin(\text{“}Alberto\text{”}) & = & 0001\ 01101100\ 0110 \\ bin(\text{“}Alessio\text{”}) & = & 0001\ 01101100\ 0110 \end{array} \quad \Rightarrow \quad \begin{array}{rcl} h(\text{“}Alberto\text{”}) & = & 5\,830 \\ h(\text{“}Alessio\text{”}) & = & 5\,830 \end{array}$$

In entrambi i casi abbiamo rilevato delle *collisioni*.

8.1.3 Metodo dello XOR

Prendiamo sempre $m = 2^p$, ma questa volta la *funzione hash* restituisce la traduzione in intero di un valore b ottenuto effettuando la somma modulo 2, bit-a-bit, di sottoinsiemi di dimensione p di $bin(k)$. Questo tipo di somma altro non è se non la funzione `XOR`.

Tuttavia, neanche questo metodo va bene perché permutazioni della stessa stringa possono generare lo stesso valore.

Esempio 23 - Utilizzo della funzione XOR.

Sia $m = 2^p = 2^{16}$.

$$\begin{array}{rrrl} bin(\text{“}Montreson\text{”}) = & & & \\ & 01101101 & 01101111 & \oplus \\ & 01101110 & 01110100 & \oplus \\ & 01110010 & 01100101 & \oplus \\ & 01110011 & 01101111 & \oplus \\ & 01110010 & 00000000 & \oplus \\ = & 01110000 & 00010001 & \end{array} \qquad \begin{array}{rrrl} bin(\text{“}Sontremor\text{”}) = & & & \\ & 01110011 & 01101111 & \oplus \\ & 01101110 & 01110100 & \oplus \\ & 01110010 & 01100101 & \oplus \\ & 01101101 & 01101111 & \oplus \\ & 01110010 & 00000000 & \oplus \\ = & 01110000 & 00010001 & \end{array}$$

Le due codifiche sono uguali quindi $h(\text{“}Montresor\text{”}) = h(\text{“}Sontremor\text{”}) = 28\,689$.

8.1.4 Metodo della divisione

In questo caso scegliamo per m un valore dispari e possibilmente primo. La *funzione hash* è definita come $h(k) = int(k) \bmod m$.

Esempio 24 - Utilizzo del metodo della divisione.

Sia $m = 383$.

$$\begin{array}{rcl} h(\text{“}Alberto\text{”}) & = & 18\,415\,043\,350\,787\,183 \bmod 383 \\ h(\text{“}Alessio\text{”}) & = & 18\,415\,056\,470\,632\,815 \bmod 383 \\ h(\text{“}Cristin\text{”}) & = & 4\,860\,062\,892\,481\,405\,294 \bmod 383 \end{array} \quad \begin{array}{rcl} & = & 221 \\ & = & 77 \\ & = & 130 \end{array}$$

Siamo finalmente riusciti a non ottenere alcuna *collisione*.

Abbiamo deciso di assegnare ad m un numero primo perché se avessimo scelto una potenza di 2, l'operatore di modulo avrebbe finito per farci considerare solo i p bit meno significativi facendoci così ritornare alla situazione del *metodo dell'estrazione*. Inoltre, anche un valore del tipo $2^p - 1$ avrebbe creato problemi in quanto si può dimostrare che permutazioni di stringhe con un set di caratteri di dimensione 2^p hanno sempre lo stesso *hash*.

In definitiva, questo metodo funziona se ad m viene assegnato un valore primo che sia "distante" da potenze di 2 o di 10.

8.1.5 Metodo della moltiplicazione

Per questo metodo scegliamo un m qualsiasi, ma è meglio se è una potenza di due. Definiamo poi una costante reale C compresa tra 0 e 1 (estremi esclusi). A questo punto, se $i = bin(k)$, la *funzione hash* è definita come $h(k) = \lfloor m(C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$.

Notiamo che $C \cdot i - \lfloor C \cdot i \rfloor$ estrae la componente decimale di $C \cdot i$.

Esempio 25 - Utilizzo del metodo della moltiplicazione.

Siano $m = 2^{16}$ *e* $C = \frac{\sqrt{5}-1}{2}$[2].

$$\begin{array}{rcccl} h(\text{“}Alessio\text{”}) & = & 65\,536 \cdot 0.51516739168 & = & 33\,762 \\ h(\text{“}Alberto\text{”}) & = & 65\,536 \cdot 0.78732161432 & = & 51\,598 \\ h(\text{“}Cristian\text{”}) & = & 65\,536 \cdot 0.72143641000 & = & 47\,280 \end{array}$$

Per questo metodo è consigliato scegliere un valore di m che sia una potenza di 2 perché consente di rendere più efficiente l'implementazione dell'algoritmo.

Infatti, se $m = 2^p$ e w è la dimensione in bit di una parola in memoria, sia per $i = bit(k)$ che per m, vale i, $m \leq 2^w$. Se poi $s = \lfloor C \cdot 2^w \rfloor$, $i \cdot s$ può essere espresso come $r_1 \cdot 2^w + r_0$ dove r_1 e r_0 contengono rispettivamente la parte intera e frazionaria di $i \cdot C$. Infine, il valore $h(k)$ di ogni chiave corrisponde ai p bit più significativi di r_0.

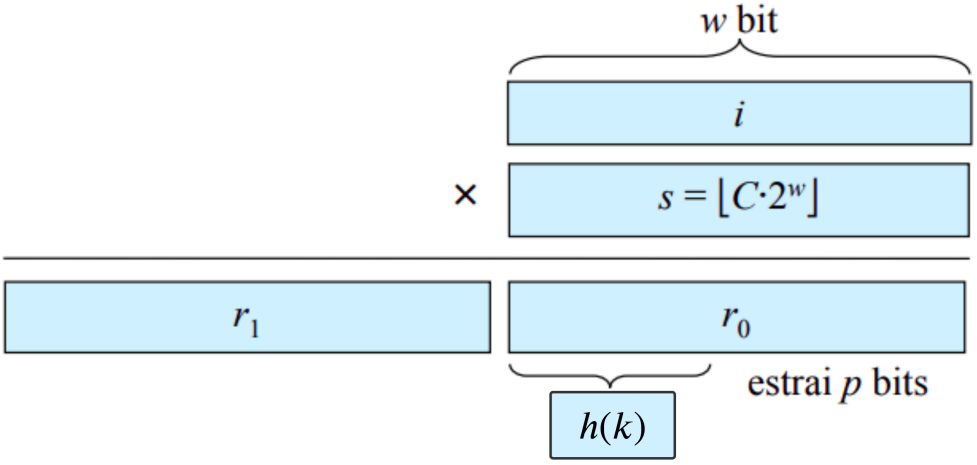


Fig. 8.1: Implementazione efficiente del *metodo della moltiplicazione*

In definitiva, il *metodo della moltiplicazione* sembra offrire una soluzione sufficientemente efficace anche se in realtà, la *funzione hash* ottenuta non gode di *uniformità semplice* e non viene mai usata in applicazioni reali.

[2] $C = \frac{\sqrt{5}-1}{2}$ è il valore suggerito da Knuth, l'ideatore dell'algoritmo

8.2 Gestione delle collisioni

Se abbiamo stabilito che non possiamo garantire l'assenza di *collisioni* dobbiamo definire una metodologia per la loro gestione. In particolare, se, in fase di inserimento, la posizione in cui dovremmo inserire una chiave è già occupata, dovremo trovarne una alternativa e, allo stesso modo, se in fase di ricerca la chiave cercata non è nella posizione attesa, dovremo cercarla altrove. Questa ricerca dovrebbe costare $O(1)$ nel caso medio e $O(n)$ nel caso pessimo.

Le tecniche possibili sono due: *liste di trabocco* e *indirizzamento aperto*. Queste due sono anche chiamate rispettivamente tecniche a *memorizzazione esterna* e *interna*.

8.2.1 Liste di trabocco

Questa metodologia prevede che tutte le chiavi con lo stesso *hash* vengano memorizzate in una *lista monodirezionale* o in un *vettore dinamico*. Ogni cella della *tabella hash* poi conterrà un puntatore alla testa della *lista* o al primo elemento del *vettore* associato a quella posizione.

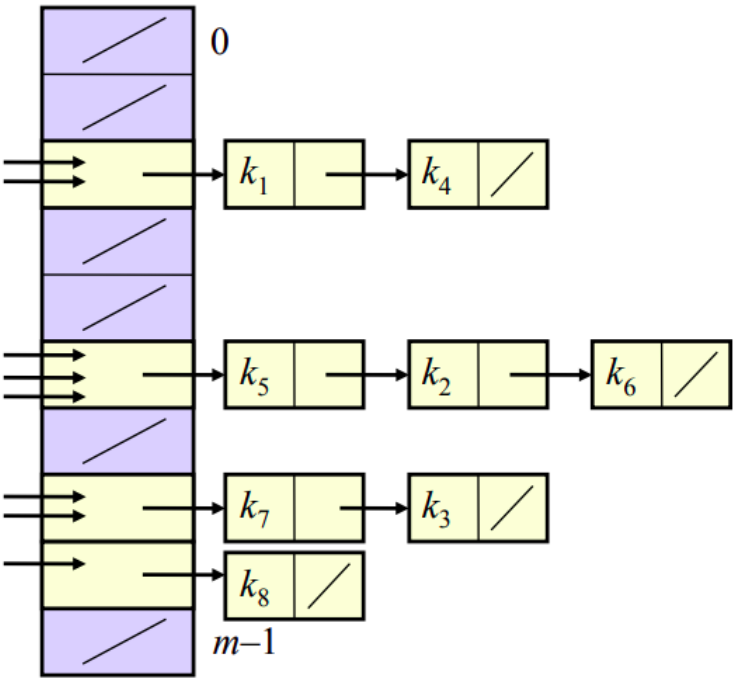


Fig. 8.2: Struttura di una *tabella hash* con *liste di trabocco*

La funzione `insert()` va ad aggiungere in testa la chiave, mentre la `lookup()` e la `remove()` scansionano la *lista* fino a trovare la chiave cercata.

Complessità Per studiare la *complessità* di questa soluzione abbiamo bisogno di definire alcuni parametri:

- n: numero di chiavi memorizzate nella *tabella hash*;
- m: dimensione della *tabella hash*;
- $\alpha = \frac{n}{m}$: *fattore di carico* della *tabella hash*;
- $I(\alpha)$: numero medio di accessi per la ricerca di una chiave che non è presente nella *tabella hash* (ricerca con insuccesso);

- $S(\alpha)$: numero medio di accessi per la ricerca di una chiave che è presente nella *tabella hash* (ricerca con successo);

Ovviamente il caso pessimo è quello in cui $m = 1$ e quindi tutte le chiavi sono inserite in un'unica lista e quindi la *complessità* è pari a quella che si avrebbe in un *dizionario* implementato come *vettore non ordinato* o come *lista*.

Studiare il caso medio invece, ci richiede di fare delle assunzioni. In particolare, ipotizziamo che la *funzione hash* costi $\Theta(1)$ e che consenta un *hashing uniforme*. Fatte queste ipotesi, ci aspettiamo che ogni *lista/vettore* abbia una lunghezza pari ad $\alpha = \frac{n}{m}$.

In una ricerca con insuccesso devono ovviamente essere toccati tutti i valori di una lista, mentre nelle ricerche con successo, in media, ne vengono toccati la metà. Questo, unito al fatto che calcolare la posizione di ricerca costa $\Theta(1)$, ci porta ad avere, rispettivamente, un costo $\Theta(1) + \alpha$ e $\Theta(1) + \frac{\alpha}{2}$ per le ricerche con insuccesso e con successo.

Ma qual è il significato del *fattore di carico*?

Il parametro influenza il costo delle operazioni sulla *tabella hash*. Per esempio, se $n = O(m)$, $\alpha = O(1)$ e quindi tutte le operazioni costano $O(1)$.

8.2.2 Indirizzamento aperto

Il problema delle *liste di trabocco* è che ci costringono a realizzare una *struttura dati* complessa. L'idea alla base delle implementazioni a *indirizzamento aperto* invece, è di memorizzare tutte le chiavi nella *tabella stessa* in modo tale che ogni posizione contenga una chiave, oppure `nil`.

Quindi, se in fase di inserimento la posizione calcolata è già occupata, se ne sceglie un'altra. Per la ricerca invece, si parte dalla posizione prescelta e si visitano tutte le posizioni alternative fino a quando non si trova la chiave cercata. Per affrontare nel dettaglia questa tecnica implementativa diamo alcune definizioni:

Definizione 95 - Ispezione.

Un'ispezione è l'esame di una posizione durante la ricerca.

Ovviamente, il numero massimo di ispezioni coincide con la dimensione m della *tabella hash*. Modifichiamo la definizione di *funzione hash* in modo da includere il concetto di *ispezione*.

Definizione 96 - Funzione hash per tabelle ad indirizzamento aperto.

In una tabella hash implementata con la tecnica dell'indirizzamento aperto, la funzione hash H è definita come

$$H : \mathcal{U} \times \overbrace{\{0, \ldots, m-1\}}^{\textit{Numero di ispezione}} \to \overbrace{\{0, \ldots, m-1\}}^{\textit{Posizione nella tabella}}$$

Definizione 97 - Sequenza di ispezione.

Una sequenza di ispezione $[H(k,0), H(k,1), \ldots, H(k,m-1)]$ è una permutazione degli indici $[0, \ldots, m-1]$ corrispondente all'ordine in cui vengono visitate le posizioni della tabella.

Generalmente non dovrebbe servire visitare tutte le posizioni, ma in ogni caso, vogliamo evitare di visitarne più volte la stessa.

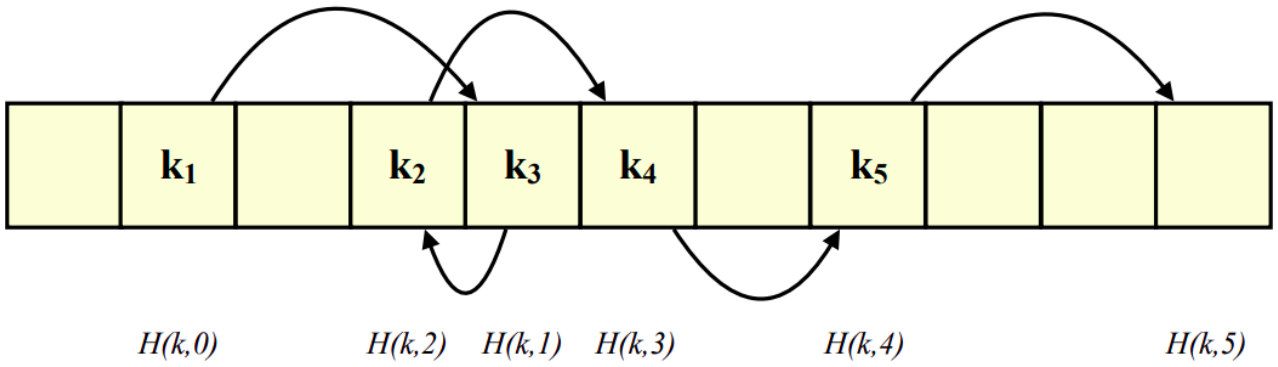


Fig. 8.3: Esempio di *sequenza di ispezione*

Ma cosa succede al *fattore di carico*?

Siccome $0 \leq n \leq m$, anche α è compreso tra 0 e 1 (estremi inclusi). Questo comporta che la *tabella hash* potrebbe andare in overflow, ovvero occupare tutte le posizioni.

Generalizzando il concetto di *uniformità semplice*, diamo la definizione di *hashing uniforme*.

Definizione 98 - Hashing uniforme.

Si parla di hashing uniforme quando una chiave ha la stessa probabilità di avere come sequenza di ispezione una qualsiasi delle $m!$ *permutazioni di* $[0, \ldots, m-1]$.

Nella realtà è difficile raggiungere l'*hashing uniforme*, per cui ci si accontenta di ottenere alcune delle possibili permutazioni.

Le *sequenze di ispezione* dipendono dalla tecnica di ispezione che si usa e le più diffuse sono: *ispezione lineare*, *quadratica* e *doppio hashing*.

Ispezione lineare Questa tecnica definisce la *funzione hash* come:

$$H(k,i) = (H_1(k) + h \cdot i) \bmod m$$

In questa definizione, h è il passo della sequenza e di conseguenza la *sequenza di ispezione* per una chiave k diventa: $[H_1(k),\ H_1(k)+h,\ H_1(k)+2h,\ \ldots]$.

Questa tecnica ci permette di avere al massimo m possibili *sequenze* che sono tutte determinabili dalla posizione della prima *ispezione*. Inoltre, c'è il problema della cosiddetta *aggregazione primaria* (o *primary clustering*) che comporta la formazione di sotto sequenze sempre più lunghe di posizioni occupate, col risultato che una posizione libera preceduta da i posizione occupate viene riempita con probabilità $\frac{i+1}{m}$ e i tempi di inserimento e cancellazione crescono sempre di più.

Ispezione quadratica L'*ispezione quadratica* segue lo stesso principio di quella *lineare*, ma le *ispezioni* hanno un offset che dipende da una funzione quadratica del numero di *ispezione*. La *funzione hash* è infatti definita come:

$$H(k,i) = (H_1(k) + h \cdot i^2) \bmod m$$

Anche in questo caso sono possibili m *sequenze di ispezione*, ma nessuna di esse è una permutazione e si propone il problema dell'*aggregazione secondaria* (o *secondary clustering*) ovvero, chiavi con la stessa posizione iniziale hanno anche *sequenze di ispezione* identiche.

Doppio hashing In questo caso H è:

$$H(k,i) = (H_1(k) + i \cdot H_2(k)) \bmod m$$

dove $H_1(k)$ fornisce la posizione iniziale e $H_2(k)$ l'offset per le successive *ispezioni*. Sono possibili m^2 *sequenze*, ma per garantire una permutazione completa $H_2(k)$ deve essere coprimo con $H_1(k)$. Cioè, se $m = 2^p$, $H_2(k)$ deve essere un numero dispari, mentre se m è primo, $H_2(k)$ deve essere un valore minore di m.

Frammento 63 - Implementazione tabella hash con hashing doppio.

```
ITEM[] K                                          % Tabella delle chiavi
ITEM[] V                                          % Tabella dei valori
int m                                             % Dimensione della tabella hash

HASH Hash(int dim)
   HASH t = new HASH
   t.m = dim
   t.K = new ITEM[dim]
   t.V = new ITEM[dim]
   for (i = 0 to dim - 1) do
      t.K[i] = nil
   return t
% Cerca la posizione associata ad una chiave
int scan(ITEM k, boolean insert)
   int delpos = m                                 % Prima posizione deleted
   i = 0                                          % Numero di ispezione
   j = H(k)                                       % Posizione di partenza
   while (K[i] ≠ k and K[j] ≠ nil and i < m) do
      if (K[j] == deleted and delpos == m) then
         delpos = j
      j = (j + H'(k)) mod m
      i = i + 1
   if (insert and K[j] ≠ k and delpos < m) then
      return delpos
   return j
% Cerca e restituisce il valore associato a una chiave oppure nil
ITEM lookup(ITEM k)
   int i = scan(k, false)
   if (k[i] == k) then
      return V[i]
   else
      return nil
```

```
% Inserisce un'associazione chiave-valore o ne modifica il valore associato
insert(ITEM k, ITEM v)
   int i = scan(k, true)
   if (K[i] == nil or K[i] == deleted or K[i] == k) then
      k[i] = k                                          % È inutile se K[i] == k
      V[i] = v
   else
      { Errore: la tabella è piena }
% Rimuove un'associazione chiave-valore
remove(ITEM k)
   int i = scan(k, false)
   if (K[i] == k) then
      K[i] = deleted
      V[i] = nil
```

Esaminiamo più nel dettaglio la funzione `scan()`. Può infatti risultare strano il fatto che abbiamo introdotto il valore `deleted` che usiamo al posto di `nil` per marcare una posizione come libera dopo una cancellazione. Questo serve a evitare situazioni come quella in figura, in cui la ricerca della chiave k_5 si interrompe all'*ispezione* della posizione `nil` facendo erroneamente credere che k_5 non sia presente nella *tabella hash*.

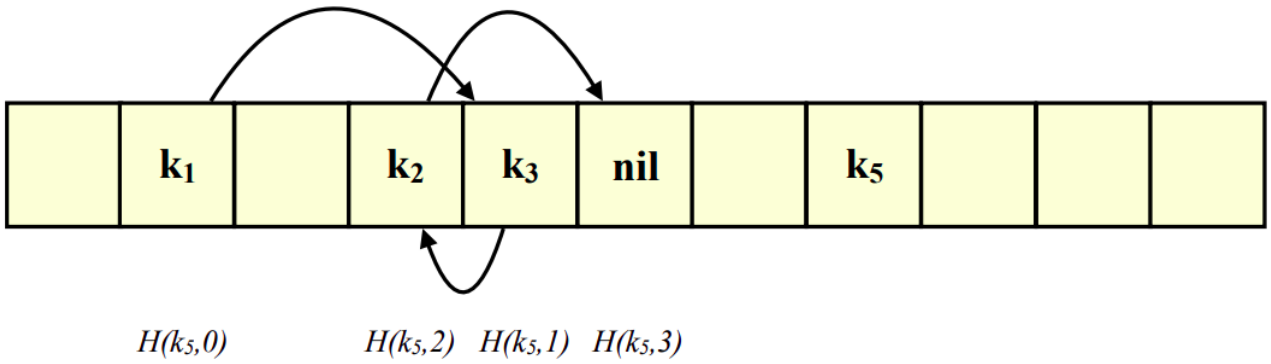


Fig. 8.4: *Sequenza di inspezione* errata

L'utilizzo del valore `deleted` al posto di `nil` ci consente di marcare le posizioni nelle quali in precedenza c'era un valore che poi è stato eliminato. In particolare, le posizioni `deleted` sono considerate come piene in fase di ricerca e vuote in fase di inserimento.

Questa soluzione risolve il problema della ricerca, ma rende il tempo di ricerca non più dipendente dal *fattore di carico* α e fa sì che il concatenamento sia più comune nelle implementazioni che ammettono la rimozione.

8.2.3 Complessità delle diverse implementazioni

Mettiamo a confronto le *complessità* di alcune delle implementazioni viste:

Tecnica	$\boldsymbol{\alpha}$	$\boldsymbol{I(\alpha)}$	$\boldsymbol{S(\alpha)}$
Lineare	$0 \leq \alpha < 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\frac{\alpha}{2}}{1-\alpha}$
Hashing doppio	$0 \leq \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$
Liste di trabocco	$\alpha \geq 0$	$1+\alpha$	$1+\frac{\alpha}{2}$

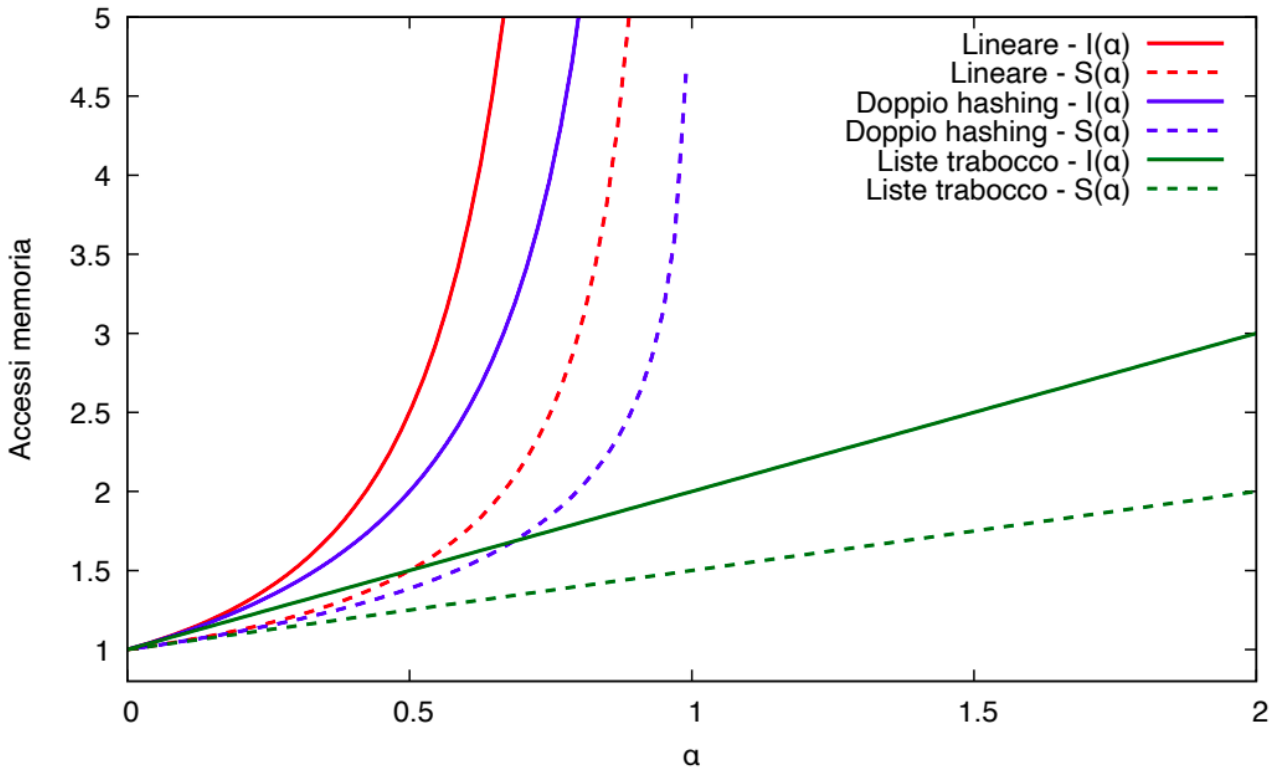


Fig. 8.5: *Complessità* delle *tabelle hash* a confronto

Conclusioni Per concludere possiamo dire che le *tabelle hash* permettono di implementare in modo molto efficiente dei *dizionari*, ma hanno una scarsa "locality of reference" che genera molte *cache miss* e inoltre, non permettono di ordinare le chiavi.

Capitolo Nr.9

Analisi ammortizzata

Dopo aver parlato a lungo di *strutture dati*, vediamo ora una tecnica che ci permette di effettuare una stima dall'alto del costo che è necessario pagare per eseguire delle operazioni su quelle strutture.

Definizione 99 - Analisi ammortizzata.

L'analisi ammortizzata è una tecnica per l'analisi di complessità che valuta il tempo richiesto per eseguire, nel caso pessimo, una sequenza di operazioni su una struttura dati.

In generale, esistono operazioni più o meno costose, ma se le operazioni più costose sono poco frequenti, il loro costo può essere ammortizzato da quelle meno costose.

È importante esplicitare la profonda differenza tra l'analisi del caso medio e l'*analisi ammortizzata*, effettuata sul caso pessimo. Nella prima, l'analisi è di tipo probabilistico ed è effettuata sulle singole operazioni, mentre, la seconda, è un'analisi deterministica effettuata su una sequenza di operazioni, e in particolare, sulla sequenza più costosa possibile.

9.1 Metodi per l'analisi ammortizzata

Esistono diverse tecniche per realizzare un'*analisi ammortizzata*.

Definizione 100 - Metodo dell'aggregazione.

Col metodo dell'aggregazione si calcola la complessità $T(n)$ necessaria per eseguire n operazioni in sequenza nel caso pessimo, quindi, si calcola il costo ammortizzato come $T(n)/n$ cioè come media su n operazioni.

Definiamo meglio i termini presenti nella definizione:

- *Sequenza*: è una sequenza di operazioni che permettono alla *struttura dati* di evolvere;
- *Caso pessimo*: è la *sequenza* con *complessità* più alta tra tutte quelle possibili;
- *Aggregazione*: la *complessità* $T(n)$ è ottenuta mediante una sommatoria dei costi delle singole operazioni;

Definizione 101 - Metodo degli accantonamenti.

Col metodo degli accantonamenti si assegna ad ogni operazione un costo ammortizzato che può anche essere diverso dal costo effettivo. In particolare, le operazioni meno costose vengono caricate di un costo aggiuntivo detto credito, che viene poi usato per pagare le operazioni più costose.

NB. Potenzialmente, ad ogni operazione potrebbe essere assegnato un costo diverso.

In generale, utilizzando il *metodo degli accantonamenti*, il *costo ammortizzato* assegnato alle operazioni meno costose è definito come:

$$\textit{Costo ammortizzato} = \textit{Costo effettivo} + \textit{Credito prodotto}$$

Viceversa, il costo per le operazioni più costose è:

$$\textit{Costo ammortizzato} = \textit{Costo effettivo} - \textit{Credito consumato}$$

Quindi, l'obiettivo, quando si utilizza questo metodo, è dimostrare che la somma dei *costi ammortizzati* a_i è un limite superiore ai costi effettivi c_i, ovvero che vale:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$$

Ovviamente, la dimostrazione deve valere per tutte le sequenze, ma se vale per il caso pessimo vale anche per tutti gli altri. Il *credito* dopo la t-esima operazione è sempre positivo ed è espresso dalla seguente formula:

$$\sum_{i=1}^{t} a_i - \sum_{i=1}^{t} c_i \geq 0$$

Definizione 102 - Metodo del potenziale.

Col metodo del potenziale si associa una funzione di potenziale Φ ad uno stato S della struttura. La funzione Φ definisce la "quantità di lavoro" $\Phi(S)$ che è stato contabilizzato nell'analisi ammortizzata, ma non ancora spesa.

NB. $\Phi(S)$ rappresenta la quantità di energia potenziale che è stata "immagazzinata" in quello stato e che può essere spesa per eseguire le operazioni più costose.

In pratica, $\Phi(S)$ può essere vista come la cumulazione dei *crediti prodotti* dalle operazioni fino al raggiungimento dello stato S.

In generale, il *costo ammortizzato* di un'operazione è pari a:

$$\begin{aligned} \textit{Costo ammortizzato} &= \textit{Costo effettivo} + \textit{Variazione di potenziale} \\ a_i &= c_i + (\Phi(S_i) - \Phi(S_{i-1})) \end{aligned}$$

dove S_i è lo stato associato all'i-esima operazione.

Se proviamo a calcolare il *costo ammortizzato* di una sequenza di n operazioni, vale quanto segue:

$$\begin{aligned} A &= \textstyle\sum_{i=1}^{n} a_i \\ &= \textstyle\sum_{i=1}^{n} (c_i + \Phi(S_i) - \Phi(S_{i-1})) \\ &= \textstyle\sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(S_i) - \Phi(S_{i-1})) \\ &= C + (\Phi(S_1) - \Phi(S_0)) + (\Phi(S_2) - \Phi(S_1)) + \cdots + (\Phi(S_n) - \Phi(S_{n-1})) \\ &= C + \Phi(S_n) - \Phi(S_0) \end{aligned}$$

Allora, se $\Phi(S_n) - \Phi(S_0) \geq 0$, A è un limite superiore al costo effettivo.

9.2 Esempio di analisi ammortizzata

Esempio 26 - Contatore binario.

Si consideri un contatore binario a k bit implementato come un vettore A di booleani, nel quale $A[0]$ è il bit meno significativo e $A[k-1]$ il più significativo. Il valore del contatore corrisponde al risultato della seguente sommatoria:

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

Alla struttura del contatore è associata soltanto la funzione `increment` *per l'incremento del valore.*

Frammento 64 - Implementazione funzione increment per contatori binari.

```
increment(int[] A, int k)
   int i = 0
   while (i < k and A[i] == 1) do    % Imposta a 0 la prima sequenza di bit a 1
      A[i] = 0
      i = i + 1
   if (i < k) then                                  % Se i < k non c'è overflow
      A[i] = 1
```

Se $k = 8$, si ottiene un contatore binario a 8 bit e la seguente tabella ne mostra lo stato dopo il sedicesimo incremento.

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Proviamo ora a realizzare una stima dall'alto del costo della funzione `increment` *usando i metodi per l'analisi ammortizzata.*

Metodo dell'aggregazione *Utilizzando questo metodo ci chiediamo quale sia il costo $T(n)$ da pagare per eseguire n operazioni in sequenza.*

Inizialmente potremmo notare che per rappresentare n in binario servono $k = \lceil \log(n+1) \rceil$ bit. Un'invocazione di `increment` *nel caso pessimo richiede un tempo $O(k)$, quindi n invocazioni costano $T(n) = O(nxk)$. A questo punto, il costo ammortizzato di ogni operazione è $T(n)/n = O(nk)/n = O(k) = O(\log n)$.*

Se però notiamo che il costo per eseguire l'intera sequenza è proporzionale al numero di bit che vengono modificati, possiamo provare a realizzare una stima più ristretta. Proviamo quindi a contare per ogni riga e per ogni colonna il numero di bit che sono stati modificati.

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	#*bit*
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5
#bit	0	0	0	1	2	4	8	16	

Guardando l'ultima riga possiamo notare che il bit in posizione $A[0]$ viene modificato ad ogni incremento, quello in $A[1]$ ogni due, quello in $A[2]$ ogni 4 e così via. Di conseguenza, il costo $T(n)$ è una funzione di questo tipo:

$$T(n) = \sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i = 2n$$

e quindi il costo ammortizzato vale:

$$\frac{T(n)}{n} \leq \frac{2n}{n} = 2 = O(1)$$

Metodo degli accantonamenti *Supponiamo che il costo effettivo della* `increment` *sia d, dove d è il numero di bit che vengono modificati ad ogni incremento. Proviamo però ad assegnare alla* `increment` *un costo di 2 che include 1 per il costo effettivo che si paga per cambiare un bit da 0 a 1 e 1 per il futuro cambio dello stesso bit da 1 a 0.*

Di conseguenza, in ogni istante, il credito è pari al numero di bit a 1 presenti e quindi il costo totale vale $O(n)$ e il costo ammortizzato $O(1)$.

Metodo del potenziale *Scegliamo come funzione di potenziale Φ il numero di bit a 1 presenti nel contatore. Ad ogni utilizzo della funzione* `increment`*, se t è il numero di bit a 1 incontrati prima del primo 0, il costo effettivo è $1 + t$ perché cambiamo lo stato di tutti i primi t bit a 1 e del primo bit a 0.*

La variazione di potenziale, invece, vale $1 - t$ in quanto tutti i bit a 1 incontrati diventano 0 e il primo bit a 0 diventa 1. Quindi, se allo stato S_i $\Phi(S_i) = t$, allo stato S_{i+1} $\Phi(S_{i+1}) = 1$ e, conseguentemente, la variazione di potenziale tra lo stato S_{i+1} e lo stato S_i vale:

$$\Phi(S_{i+1}) - \Phi(S_i) = 1 - t$$

Il costo ammortizzato di un'invocazione della `insert`*, allora, vale:*

$$\begin{aligned} \text{Costo ammortizzato} &= \text{Costo effettivo} + \text{Variazione di potenziale} \\ &= (1 + t) + (1 - t) \\ &= 2 = O(1) \end{aligned}$$

Inoltre, poiché allo stato S_0, $\Phi(S_0) = 0$ perché non ci sono bit impostati a 1, e allo stato S_n, $\Phi(S_n) \geq 0$, risulta verificata anche la condizione $\Phi(S_n) - \Phi(S_0) \geq 0$.

9.3 Vettori dinamici

Possiamo utilizzare le tecniche di *analisi ammortizzata* per stimare la *complessità* delle operazioni di inserimento e cancellazione nei vettori con ridimensionamento dinamico della dimensione.

Prima però vediamo brevemente cosa significa "ridimensionamento dinamico della dimensione". Quando si implementa una *Sequenza* utilizzando un vettore, se ne specifica una dimensione iniziale detta *capacità*. Ovviamente, non è sempre nota a priori la quantità di elementi che dovranno essere inseriti nella struttura e quindi la *capacità* iniziale può rivelarsi insufficiente. I *vettori dinamici* risolvono questo problema allocando un nuovo vettore con una capacità maggiore e ricopiando in esso tutti i valori presenti nel vettore originale, che poi può essere eliminato.

9.3.1 Inserimento

Quando in fase di inserimento si rende necessario aumentare la dimensione del vettore, esistono due principali metodologie di approccio: *incremento fisso* e *incremento variabile*.

Il primo approccio prevede che la *capacità* venga incrementata di un fattore fisso (e.g. $+1$, $+2$, $+10$, ...), mentre il secondo è dipendente dalla dimensione attuale della struttura (e.g. $\cdot 2$, $\cdot 1.5$, ...).

Ma qual è il metodo migliore? Proviamo a utilizzare l'*analisi ammortizzata* per realizzare una stima.

Incremento fisso Il costo effettivo di un inserimento con ridimensionamento fisso è descritto dalla seguente espressione:

$$c_i = \begin{cases} i & \text{se } (i \bmod d) = 1 \text{[1]} \\ 1 & \text{altrimenti} \end{cases}$$

dove d è sia la dimensione iniziale che il valore di cui viene incrementata.

[1] L'inserimento ha un costo i quando $(i \bmod d) = 1$ perché se è stata raggiunta la dimensione massima quell'operazione di modulo vale 1

Supponiamo $d = 4$ e vediamo il costo delle prime 17 operazioni di inserimento:

n	Costo
1	1
2	1
3	1
4	1
5	$1 + d = 5$
6	1
7	1
8	1
9	$1 + 2d = 9$
10	1
11	1
12	1
13	$1 + 3d = 13$
14	1
15	1
16	1
17	$1 + 4d = 17$

A questo punto, calcoliamo il costo effettivo di n operazioni:

$$\begin{aligned} T(n) &= \textstyle\sum_{i=1}^{n} c_i \\ &= n + \textstyle\sum_{j=1}^{\lfloor n/d \rfloor} d \cdot j \\ &= n + d \textstyle\sum_{j=1}^{\lfloor n/d \rfloor} j \\ &= n + d\frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\ &\leq n + \frac{(n/d+1)n}{2} && \textit{Rimozione dell'intero inferiore} \\ &= n + \frac{n^2/d+n}{2} = O(n^2) \end{aligned}$$

Il *costo ammortizzato* vale:

$$\frac{T(n)}{n} = \frac{O(n^2)}{n} = O(n)$$

Incremento dinamico Supponendo che ad ogni incremento la dimensione del vettore venga raddoppiata, il costo effettivo vale:

$$c_i = \begin{cases} i & \text{se } \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

Se la dimensione iniziale è 1, il costo delle prime 17 operazioni è il seguente:

n	Costo
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Il costo effettivo di n operazioni vale:

$$\begin{aligned} T(n) &= \textstyle\sum_{i=1}^{n} c_i \\ &= n + \textstyle\sum_{j=0}^{\lfloor \log n \rfloor} 2^j \\ &= n + 2^{\lfloor \log n \rfloor + 1} - 1 \\ &\leq 2^{\log(n)+1} - 1 && \textit{Rimozione dell'intero inferiore} \\ &= n + 2n - 1 = O(n) \end{aligned}$$

e di conseguenza quello *ammortizzato* è:

$$\frac{T(n)}{n} = \frac{O(n)}{n} = O(1)$$

Giunti a questo punto, siamo riusciti a dimostrare che l'incremento dinamico ha un costo inferiore all'incremento fisso.

9.3.2 Cancellazione

Se il vettore non è ordinato, rimuovere un elemento significa spostare l'ultimo elemento della *sequenza* nella posizione dell'elemento da rimuovere.

Per ridurre lo spreco di memoria, è opportuno ridurre la dimensione del vettore quando il *fattore di carico* $\alpha = \frac{Dim}{Capacità}$[2] scende sotto una certa soglia.

L'operazione di riduzione della *capacità* è detta *contrazione* e, similmente a quanto avviene con l'estensione, prevede che venga allocato un nuovo vettore di dimensione inferiore a quella

[2]Il valore *Dim* rappresenta il numero di elementi che in dato momento sono presenti nella struttura

attuale, che vi vengano copiati i valori del vettore originale e che quindi quest'ultimo venga eliminato.

Ma qual è la soglia di *contrazione* ottimale?

Una prima strategia potrebbe essere quella di dimezzare la *capacità* quando *Dim* raggiunge la metà della *capacità*, cioè quando $\alpha = \frac{1}{2}$. Il problema di questa scelta è che dopo la *contrazione* nella struttura non rimangono posizioni libere e quindi un successivo inserimento richiederebbe di fare subito un'*espansione*.

Una strategia migliore è quella invece di scegliere $\alpha = \frac{1}{4}$. In questo modo, se quando viene raggiunta la *soglia di contrazione*, si va a dimezzare la *capacità*, il *fattore di carico* α, invece di aumentare fino a 1 si fermerà a $\frac{1}{2}$ concedendoci di aggiungere al vettore tanti elementi quanti sono quelli già presenti prima di dover effettuare un'*espansione*.

Analisi del costo Proviamo a stimare il costo di questa seconda soluzione utilizzando il *metodo del potenziale*. Scegliamo una funzione di potenziale Φ che vale 0 all'inizio e immediatamente dopo un'*espansione* o una *contrazione* e il cui valore cresca fino a raggiungere il numero di elementi presenti nella struttura quando $\alpha = 1$ e diminuisca fino a un quarto quando α si riduce alla *soglia di contrazione*.

Quindi, la definizione di Φ è la seguente:

$$\Phi = \begin{cases} 2 \cdot Dim - Capacità & \text{se } \alpha \geq \frac{1}{2} \\ \frac{Capacità}{2} - Dim & \text{se } \alpha < \frac{1}{2} \end{cases}$$

Vediamone il valore in alcuni casi particolari:

- $\alpha = \frac{1}{2}$: è stata appena effettuata un'*espansione* o una *contrazione* e quindi $\Phi = 0$;
- $\alpha = 1$ la struttura è piena, ovvero $Dim = Capacità$ e quindi $\Phi = Dim = Capacità$;
- $\alpha = \frac{1}{4}$: il *fattore di carico* ha raggiunto la *soglia di contrazione*, ovvero $Dim = \frac{Capacità}{4}$ e quindi $\Phi = Dim$;

Una funzione di potenziale di questo tipo ci garantisce che il potenziale presente nel momento in cui si effettua un'*espansione* o una *contrazione* sia sufficiente per "pagare" quelle stesse operazioni.

Proviamo quindi a calcolare il *costo ammortizzato* in base ai diversi valori di α:

- $\alpha \geq \frac{1}{2}$:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2Dim_i - Capacità_i) - (2Dim_{i-1} - Capacità_{i-1}) \\ &= 1 + 2(Dim_{i-1} + 1) - Capacità_i - 2Dim_{i-1} + Capacità_{i-1} \\ &= 1 + 2(Dim_{i-1} + 1) - Capacità_{i-1} - 2Dim_{i-1} + Capacità_{i-1} \\ &= 1 + 2Dim_{i-1} + 2 - Capacità_{i-1} - 2Dim_{i-1} + Capacità_{i-1} \\ &= 3 \end{aligned}$$

- $\alpha = 1$:

$$\begin{aligned} a_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + Dim_{i-i} + (2Dim_i + Capacità_i) - (2Dim_{i-1} - Capacità_{i-1}) \\ &= 1 + Dim_{i-i} + 2(Dim_{i-1} + 1) - 2Dim_{i-1} - 2Dim_{i-1} + Dim_{i-1} \\ &= 3 \end{aligned}$$

In entrambi i casi il costo è $O(1)$.

NB. Poiché non è conveniente che il *fattore di carico* α cresca troppo, regole di ridimensionamento simili vengono usate anche nelle *tabelle hash*. Solitamente, quando α raggiunge una soglia di 0.5 o 0.75 la *tabella hash* viene estesa raddoppiandone la *capacità*. Quest'operazione, oltre che dimezzare il *fattore di carico*, rimuove anche tutti gli elementi `deleted`. Il costo nel caso peggiore è $O(m)$, ma quello *ammortizzato* è $O(1)$.

9.4 Discussione sugli insiemi

Giunti a questo punto della trattazione, abbiamo introdotto una vasta gamma di *strutture dati* più o meno complesse e caratterizzata da costi più o meno convenienti. Abbiamo anche visto che *strutture dati astratte* possono essere implementate utilizzando diverse *strutture dati concrete* e che questo influenza il costo delle operazioni. Per chiudere il cerchio, mettiamo ora a confronto diverse implementazioni di un *insieme*.

9.4.1 Insieme come vettore di booleani

Quando si a che fare con m valori ordinabili e consecutivi è possibile implementare un *insieme* per quei valori come un vettore di booleani di dimensione m. Ogni posizione del vettore vale `true` l'elemento associato appartiene all'insieme, altrimenti vale `false`.

Vediamone l'implementazione:

Frammento 65 - Insieme come vettore di booleani.

```
boolean[] V      % Vettore di booleani
int size % Numero di elementi presenti
int capacity      % Dimensione massima

SET Set(int m)
   SET t = new SET
   t.size = 0
   t.capacity = m
   t.V = new int[1...m] = false
   return t

boolean contains(int x)
   if (1 ≤ x ≤ capacity) then
      return V[x]
   else
      return false

SET union(SET A, SET B)
   int newSize = max(A.capacity, B.capacity)
   SET C = Set(newSize)
   for (i = 1 to A.capacity) do
      if (A.contains(i)) then
         C.insert(i)
   for (i = 1 to B.capacity) do
      if (B.contains(i)) then
         C.insert(i)
   return C
```

```
int size()
   return size

insert(int x)
   if (1 ≤ x ≤ capacity) then
      if (not V[x]) then
         size = size + 1
         V[x] = true

remove(int x)
   if (1 ≤ x ≤ capacity) then
      if (V[x]) then
         size = size - 1
         V[x] = false
```

```
SET difference(SET A, SET B)
   SET C = Set(A.capacity)
   for (i = 1 to A.capacity) do
      if (A.contains(i) and not B.contains(i)) then
         C.insert(i)
   return C

SET intersection(SET A, SET B)
   int newSize = min(A.capacity, B.capacity)
   SET C = Set(newSize)
   for (i = 1 to newSize) do
      if (A.contains(i) and B.contains(i)) do
         C.insert(i)
   return C
```

I vantaggi di questa implementazione sono certamente la semplicità e l'efficienza delle operazioni di inserimento, rimozione e verifica di appartenenza. Tuttavia, la memoria occupata è indipendente dal numero di elementi effettivamente contenuti, così come lo sono le operazioni che visitano tutti gli elementi. Ad esempio, le operazioni di `union`, `difference` e `intersection` hanno costo $O(m)$.

9.4.2 Insieme come lista non ordinata

In un *insieme* implementato come *lista* non ordinata le operazioni di inserimento, ricerca e rimozione hanno un costo $O(n)$ dove n è il numero di elementi presenti. Per le operazioni di unione, intersezione e differenza non va meglio, infatti dati due *insiemi* A e B di dimensione n_A e n_B, la *complessità* è $O(n_A n_B)$.

NB. Se supponiamo di sapere che un elemento non appartiene all'*insieme*, l'inserimento può avere costo $O(1)$.

Frammento 66 - Insieme come lista non ordinata.

```
SET union(SET A, SET B)
   SET C = Set()
   foreach (s ∈ A) do                                  % Costa O(n_A)
      C.insert(s)                                      % Costa O(n_C)[3]
   foreach (s ∈ B) do                                  % Costa O(n_B)
      C.insert(s)                                      % Costa O(n_C)
   return C                          % Abbiamo pagato O(n_A n_C + n_B n_C) = O(n_C m)

SET difference(SET A, SET B)
   SET C = Set()
   foreach (s ∈ A) do                                  % Costa O(n_A)
      if (not B.contains(s))                           % Costa O(n_B)
         C.insert(s)                        % Possiamo supporre ci costi O(1)
   return C                             % Abbiamo pagato O(n_A n_B 1) = O(n_A n_B)
```

[3] Avremmo anche potuto supporre di pagare $O(1)$ in quanto sicuramente nessuno degli elementi di A apparteneva a C

```
SET intersection(SET A, SET B)
   SET C = Set()
   foreach (s ∈ A) do                                          % Costa O(n_A)
      if (B.contains(s))                                       % Costa O(n_B)
         C.insert(s)                                % Possiamo supporre ci costi O(1)
   return C                                  % Abbiamo pagato O(n_A n_B 1) = O(n_A n_B)
```

NB. Il costo dell'implementazione come vettore non ordinato è in tutto e per tutto equivalente all'implementazione come *lista* non ordinata.

9.4.3 Insieme come lista ordinata

Con le *liste* ordinate i costi rimangono identici ad eccezione di unione, intersezione e differenza il cui costo si riduce a $O(n)$.

Frammento 67 - Insieme come lista ordinata.

```
SET union(SET A, SET B)
   SET C = Set()
   POS pos_A = A.head()
   POS pos_B = B.head()
   while (not A.finished(pos_A) and not B.finished(pos_B)) do    % O(max(n_A, n_B))
      C.insert(C.tail(), A.read(pos_A))       % L'inserimento in coda costa O(1)
      if (A.read(pos_A) == B.read(pos_B)) then
         pos_A = A.next(pos_A)
         pos_B = B.next(pos_B)
      else if (A.read(pos_A) < B.read(pos_B)) then
         pos_A = A.next(pos_A)
      else
         pos_B = B.next(pos_B)
   return C

SET difference(SET A, SET B)
   SET C = Set()
   POS pos_A = A.head();
   POS pos_B = B.head();
   while (not A.finished(pos_A) and not B.finished(pos_B)) do    % O(max(n_A, n_B))
      if (A.read(pos_A) ≠ B.read(pos_B)) then
         C.insert(C.tail(), A.read(pos_A))    % L'inserimento in coda costa O(1)
         if (A.read(pos_A) > B.read(pos_B)) then
            pos_B = B.next(pos_B)
         pos_A = A.next(pos_A)
   return C
```

```
SET intersection(SET A, SET B)
   SET C = Set()
   POS pos_A = A.head();
   POS pos_B = B.head();
   while (not A.finished(pos_A) and not B.finished(pos_B)) do    % O(max(n_A, n_B))
      if (A.read(pos_A) == B.read(pos_B)) then
         C.insert(C.tail(), A.read(pos_A))    % L'inserimento in coda costa O(1)
         pos_A = A.next(pos_A)
         pos_B = B.next(pos_B)
      else if (A.read(pos_A) < B.read(pos_B)) then
         pos_A = A.next(pos_A)
      else
         pos_B = B.next(pos_B)
   return C
```

NB. Il costo dell'implementazione come vettore ordinato è equivalente all'implementazione come *lista* ordinata, tranne la funzione di `contains` che ha costo $O(\log n)$ in quanto su un vettore ordinato è possibile usare un algoritmo di ricerca dicotomica.

9.4.4 Confronto generale

In generale, se n è il numero di elementi presenti in un *insieme* ed m è la capacità di quell'*insieme*, la *complessità* delle operazioni a seconda dell'implementazione usata è riassunta dalla tabella sottostante.

Operazione / **Struttura**	`contains` `lookup`	`insert`	`remove`	`min`	`foreach`	**Ordinabile**
Vettore booleano	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	Sì
Lista non ordinata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	No
Lista ordinata	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sì
Vettore ordinato	$O(\log n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	Sì
Albero bilanciato	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	Sì
Tabella Hash Mem. interna	$O(1)$	$O(1)$	$O(1)$	$O(m)$	$O(m)$	No
Tabella Hash Mem. esterna	$O(1)$	$O(1)$	$O(1)$	$O(m+n)$	$O(m+n)$	No

Capitolo Nr.10

Divide-et-impera

10.1 Introduzione

Giunti a questo punto della trattazione, è arrivato il momento di parlare delle tecniche per la risoluzione di problemi, ovvero di quelle tecniche che permettono di arrivare alla definizione di un algoritmo per la risoluzione di un particolare problema.

Esiste un'ampia di gamma di categorie di problemi, tra le quali troviamo:

- *Problemi decisionali*: l'obiettivo è riuscire a stabilire se il dato in ingresso soddisfa o meno una proprietà;
- *Problemi di ricerca*: l'obiettivo è ricercare all'interno dell'insieme dei dati di input un sottoinsieme di dati che soddisfano una certa proprietà;
- *Problemi di ottimizzazione*: in un insieme di soluzioni alle quali è associata una funzione di costo, l'obiettivo è trovare la soluzione di costo minimo;

Le tecniche per la ricerca di soluzioni sono molteplici e nel corso della trattazione ne approfondiremo diverse. In questo capitolo, iniziamo a vedere la tecnica del *Divide-et-impera*.

Definizione 103 - Tecnica del Divide-et-impera.

La tecnica del Divide-et-impera prevede che il problema da risolvere venga suddiviso in sotto-problemi indipendenti e che le soluzioni ai sotto-problemi vengano combinate per ottenere la soluzione al problema di partenza.

La definizione lascia intendere molto chiaramente la simbiosi che esiste tra algoritmi *Divide-et-impera* e ricorsione. Infatti, la suddivisione in sotto-problemi viene realizzata applicando ricorsivamente l'algoritmo ad un sottoinsieme dei dati di input.

L'approccio *Divide-et-impera* si compone di tre fasi:

- *Divide*: il problema viene suddiviso in sotto-problemi indipendenti;
- *Impera*: vengono risolti i sotto-problemi;
- *Combina*: le soluzioni dei sotto-problemi vengono combinate per ottenere la soluzione al problema di partenza;

NB. La tecnica del *Divide-et-impera* trova applicazione soprattutto negli ambiti dei *problemi decisionali* e *di ricerca*.

Vediamo un esempio tipico di applicazione della tecnica *Divide-et-impera*.

Problema 3 - Problema della Torre di Hanoi.

Quello della Torre di Hanoi è un problema matematico molto famoso. Esistono tre pioli e n dischi di dimensione diversa. Inizialmente i dischi sono impilati in ordine decrescente sul piolo di sinistra. Lo scopo del gioco è riuscire a impilare quegli stessi dischi in ordine decrescente sul piolo di destra.

Ad ogni mossa è possibile spostare un disco ed è possibile usare il piolo centrale come appoggio. L'importante è non posizionare mai un disco sopra uno più piccolo.

Frammento 68 - Implementazione algoritmo per risoluzione della Torre di Hanoi.

```
hanoi(int n, int src, int dest, int aux)
   if (n == 1) then
      print src → dest
   else
      hanoi(n-1, src, aux, dest)     % Sposta n − 1 dischi sul piolo ausiliario
      print src → dest                          % Sposta il disco più grande
      hanoi(n-1, aux, src, dest)        % Sposta n − 1 dischi sul piolo finale
```

In questo codice è evidente la suddivisione in sotto-problemi in quanto l'invocazione `hanoi(n-1, src, aux, dest)` *sposta tutti i dischi tranne l'ultimo sul piolo ausiliario usando il piolo che sarebbe di destinazione come piolo d'appoggio. Quindi, sposta il disco rimasto, il più grande, sul piolo di destinazione. L'invocazione* `hanoi(n-1, aux, dest, src)` *sposta di nuovo gli* $n-1$ *dischi di prima sul piolo di destinazione.*

È interessante provare a studiare la complessità di questa soluzione. La funzione di ricorrenza è la seguente:

$$T(n) = 2T(n-1) + 1$$

e per il Teorema delle ricorrenze lineari di ordine costante il costo di questo algoritmo è $\Theta(2^n)$*, che nonostante sia un costo esponenziale è comunque il migliore possibile in quanto è dimostrabile l'ottimalità della soluzione proposta.*

Convenienza degli algoritmi Divide-et-impera L'utilizzo di una qualsiasi tecnica per la risoluzione di problemi non sempre permette di arrivare ad una soluzione ottima o anche solo conveniente. Ad esempio, la seguente implementazione ricorsiva della funzione di ricerca del minimo non è migliore della sua controparte iterativa:

Frammento 69 - Implementazione minrec per la ricerca ricorsiva del minimo.

```
int minrec(int[] A, int i, int j)
   if (i == j) then
      return A[i]
   else
      m = ⌊(i + j) / 2⌋
      return min(minrec(A, i, m), minrec(A, m + 1, j))
```

La *funzione di ricorrenza* di questa soluzione è:

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

e per il *Teorema delle ricorrenze lineari con partizione bilanciata - Rid* vale $T(n) = \Theta(n)$ che è la stessa *complessità* della versione iterativa dell'algoritmo. Di conseguenza, in questo caso non conviene usare la tecnica del *Divide-et-impera* perché l'algoritmo ottenuto è più complicato di quello che già conoscevamo.

10.2 Algoritmo Quicksort

All'inizio della trattazione abbiamo già esaminato un algoritmo di ordinamento basato sul *Divide-et-impera*: l'algoritmo *merge sort*.

10.2.1 Principi di funzionamento

Il *Quicksort* è un altro algoritmo di ordinamento, basato anch'esso su *Divide-et-impera*, che riceve in input un vettore $A[1 \dots n]$ e due indici lo e hi tali che $1 \leq lo \leq hi \leq n$.

Nella fase di *divide* viene scelto un valore $p \in A[lo \dots hi]$ detto *perno* o *pivot*. Quindi, tutti gli elementi del vettore vengono permutati in modo da portare tutti i valori più piccoli del *pivot* alla sua sinistra e gli altri alla sua destra.

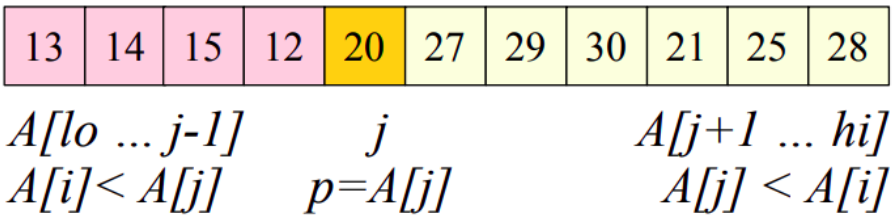


Fig. 10.1: Posizionamento dei valori dopo la scelta del *pivot*

Nella fase di *Impera* vengono ordinati i due sottoarray $A[lo \dots j-1]$ e $A[j+1 \dots hi]$ e infine nella fare di *Combina* non serve fare nulla in quanto il vettore risulta già ordinato.

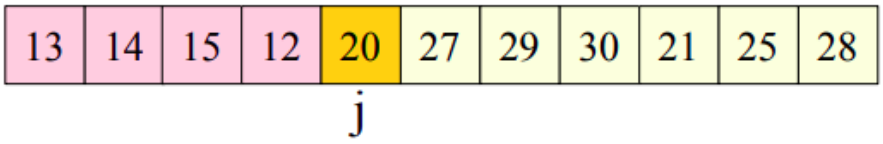


Fig. 10.2: Vettore ordinato al termine della fase di *Impera*

10.2.2 Implementazione

Frammento 70 - Implementazione dell'algoritmo Quicksort.

```
quicksort(ITEM[] A, int lo, int hi)
   if (lo < hi) then
      int j = pivot(A, lo, hi)                          % Calcola il pivot
      quicksort(A, lo, j - 1)                % Ordina il sottoarray sinistro
      quicksort(A, j + 1, hi)                  % Ordina il sottoarray destro

% Funzione per la ricerca del pivot
int pivot(ITEM[] A, int lo, int hi)
   ITEM pivot = A[lo]                          % Scelta del valore del pivot
   int j = lo                                                % Indice del pivot
   for (i = lo + 1 to hi) do
      if (A[i] < pivot) then
         j = j + 1                              % Aggiorna l'indice del pivot
         swap(A, i, j)    % Scambia di posizione gli elementi agli indici i e j
   A[lo] = A[j]
   A[j] = pivot                              % Mette in posizione j il pivot
   return j

% Funzione ausiliaria per lo scambio di posizione di due elementi
swap(ITEM[] A, int i, int j)
   int tmp = A[i]
   A[i] = A[j]
   A[j] = tmp
```

10.2.3 Costo computazionale

Per studiare il costo della funzione principale `quicksort` analizziamo le funzioni ausiliarie. La `swap` è banale e ha un costo $\Theta(1)$, mentre la `pivot` costa $\Theta(n)$ perché va a confrontare il *pivot* con ogni elemento dell'array. Per il costo della `quicksort` consideriamo separatamente il caso pessimo il caso migliore.

Il caso pessimo è quello in cui la scelta del *pivot* porta sempre ad avere due sottoarray di dimensione 0 e $n-1$ e conseguentemente induce una *funzione di ricorrenza* di questo tipo:

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

Nel caso migliore invece, il vettore viene sempre diviso in due sottoarray di dimensione $n/2$ e quindi il costo è descritto dalla seguente funzione:

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

E il caso medio? Fortunatamente, i partizionamenti nel caso medio sono molto più simili al caso migliore che al peggiore. Ad esempio, con un partizionamento $9 - a - 1$ vale:

$$T(n) = T(n/10) + T(9n/10) + cn = \Theta(n \log n)$$

Lo stesso accade anche un partizionamento $99 - a - 1$:

$$T(n) = T(n/100) + T(99n/100) + cn = \Theta(n \log n)$$

NB. In questi esempi non stiamo considerando i fattori moltiplicativi, ma in certi contesti potrebbero essere importanti.

Riassumendo, il costo dell'algoritmo *Quicksort* è $\Theta(n \log n)$ nei casi ottimo e medio, mentre è $\Theta(n^2)$ nel caso pessimo. Il *Merge sort* aveva un costo $\Theta(n \log n)$ in tutti i casi, quindi a prima vista sembrerebbe essere più conveniente. In verità, poiché il *Quicksort* non usa memoria addizionale, gode di fattori moltiplicativi inferiori rispetto al *Merge sort*, è parallelizzabile ed esistono tecniche Euristiche che consentono di evitare il caso pessimo, è spesso preferito al *Merge sort*.

Funzione euristica per la selezione del pivot Vediamo di seguito un esempio di una funzione per la selezione del *pivot* in maniera euristica.

Frammento 71 - Implementazione euristica della funzione pivot.

```
int pivot(ITEM[] A, int lo, int hi)
   int m = ⌊(lo + hi) / 2⌋
   if (A[lo] > A[hi]) then
      swap(A, lo, hi)                    % Sposta il massimo in ultima posizione
   if (A[m] > A[hi]) then
      swap(A, m, hi)                     % Sposta il massimo in ultima posizione
   if (A[m] > A[lo]) then
      swap(A, m, lo)                      % Sposta il mediano in prima posizione
   ITEM pivot = A[lo]
   int j = lo                                                  % Indice del pivot
   for (i = lo + 1 to hi) do
      if (A[i] < pivot) then
         j = j + 1                                  % Aggiorna l'indice del pivot
         swap(A, i, j)    % Scambia di posizione gli elementi agli indici i e j
   A[lo] = A[j]
   A[j] = pivot                                  % Mette in posizione j il pivot
   return j
```

10.3 Esercizio di applicazione del Divide-et-impera

Problema 4 - Ricerca di un gap in un vettore.

Dato un vettore V contenente $n \geq 2$ interi, un gap è un indice i con $1 < i \leq n$ tale che $V[i-1] < V[i]$.

- *Dimostrare che se $n \geq 2$ e $V[1] < V[n]$, allora V contiene sicuramente almeno un gap;*
- *Progettare un algoritmo che, dato un vettore V contenente $n \geq 2$ valori e tale che $V[1] < V[n]$, restituisce la posizione di un gap nel vettore;*

Iniziamo dimostrando il primo punto.

Dimostrazione. Supponiamo per assurdo che non esista alcun gap all'interno di V. Di conseguenza, deve per forza valere la seguente catena di disuguaglianze:

$$V[1] \geq V[2] \geq \cdots \geq V[n]$$

Questo però è impossibile in quanto per ipotesi $V[1] < V[n]$ e quindi deve per forza esistere almeno un gap all'interno di V. □

Per il secondo punto proviamo a fare un ragionamento per induzione. Siano i e j due indici tali che $1 \leq i < j \leq n$ e supponiamo che $V[i] < V[j]$. In base a questa definizione di i e j, nel sottoarray $V[i \dots j]$ ci sono almeno due elementi e il primo, $V[i]$, è minore dell'ultimo, $V[j]$.

Proviamo a dimostrare per induzione sulla dimensione del sottoarray $V[i \dots j]$ che esiste almeno un gap.

Caso base *$n = 2$ e quindi $j - i + 1 = 2$, ovvero $i = j - 1$. Da questo segue che $V[i] < V[j] \Leftrightarrow V[j-1] < V[j]$ e quindi alla posizione j esiste un gap.*

Passo induttivo *Ipotizziamo che dato un qualunque sottoarray $V[h \dots k]$, di dimensione $n' < n$ e tale che $V[h] < V[k]$, contenga un gap. A questo punto, se consideriamo un qualunque indice $m \in]i, j[$, si verificherà sicuramente almeno uno dei seguenti casi:*

- *$V[m] < V[j]$: per ipotesi induttiva, esiste sicuramente un gap all'interno di $V[m \dots j]$;*
- *$V[i] < V[m]$: per ipotesi induttiva, esiste sicuramente un gap all'interno di $V[i \dots m]$;*

Fatte queste considerazioni, possiamo usare la tecnica del Divide-et-impera per definire un algoritmo per la ricerca di gap:

Frammento 72 - Implementazione algoritmo Divide-et-impera per la ricerca di gap.

```
int gap(int[] V, int n)
   return gaprec(V, 1, n)

int gaprec(int[] V, int i, int j)
   if (j == i + 1) then                                    % Caso base
      return j
   int m = ⌊(i + j) / 2⌋                          % Scelta dell'indice m ∈ ]i, j[
   if (V[m] < V[j]) then
      return gaprec(V, m, j)
   else
      return gaprec(V, i, m)
```

Capitolo Nr.11

Strutture dati specializzate

Finora abbiamo esaminato una serie di *strutture dati* e per ciascuna ne abbiamo anche analizzato il costo delle operazioni. È possibile però definire *strutture dati speciali*, o per meglio dire "*specializzate*", nelle quali vengono implementate soltanto alcune delle operazioni e per questo motivo quelle implementazioni possono essere realizzare in modo più efficiente.

In questo capitolo vedremo due esempi di *strutture specializzate*: le *code a priorità* e gli *insiemi disgiunti*.

11.1 Code a priorità

Definizione 104 - Coda a priorità.

Una coda a priorità è una struttura dati astratta, simile a una coda, in cui ogni elemento possiede un valore che ne indica la "priorità" e che viene usato per stabilire l'ordine di estrazione degli elementi dalla struttura.

Esistono due tipi di *code a priorità* (*priority queue*):

- *Min-priority queue*: l'estrazione avviene per valori crescenti di priorità;
- *Max-priority queue*: l'estrazione avviene per valori decrescenti di priorità;

Specifica

Frammento 73 - MINPRIORITYQUEUE.

```
% Crea una coda a priorità con capacità n
PRIORITYQUEUE PriorityQueue(int n)
% Restituisce true se la coda a priorità è vuota
boolean isEmpty()
% Restituisce l'elemento minimo di una coda a priorità non vuota
ITEM min()
% Rimuove e restituisce l'elemento minimo di una coda a priorità non vuota
deleteMin()
% Inserisce l'elemento x con priorità p nella coda a priorità e restituisce
% un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(ITEM x, int p)
```

```
% Diminuisce la priorità dell'oggetto identificato da y portandola a p
decrease(PRIORITYITEM y, int p)
```

NB. La specifica di una *max-priority queue* è uguale, ma invece delle operazioni `min`, `deleteMin` e `decrease` ha `max`, `deleteMax` e `increase`.

Utilizzando le *strutture dati* viste finora possiamo calcolare i seguenti costi:

Operazione	Lista Vettore non ordinato	Lista ordinata	Vettore ordinato	Albero Red-Black
min	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$
deleteMin	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
insert	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
decrease	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$

È possibile fare meglio di così?

La risposta è sì, utilizzando uno *heap*, una *struttura dati speciale* che associa i vantaggi di un *albero*, cioè la *complessità* $O(\log n)$, e la memorizzazione efficiente ottenibile con i normali vettori.

11.1.1 Heap

La *struttura dati* dello *heap* fu inventata da J. Williams nel 1964 con l'obiettivo di realizzare l'algoritmo di ordinamento *HeapSort*. Vediamo come si è arrivati all'ideazione dello *heap*.

Consideriamo un *albero binario perfetto* come il seguente:

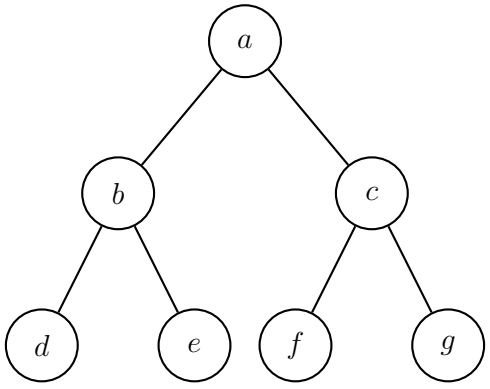


Fig. 11.1: *Albero binario perfetto*

Tutte le *foglie* sono alla stessa *profondità* h e tutti i *nodi* interni hanno *grado uscente* pari a 2. Se n è il numero di *nodi*, l'*altezza* vale $h = \lfloor \log n \rfloor$ e, data l'*altezza* h, il numero di *nodi* è $n = 2^{h+1} - 1$.

Cosa accade se si aggiunge un *nodo*?

Ovviamente l'*albero* non può più essere definito *perfetto*. Supponiamo però di "accatastare" tutti i nuovi *nodi* a partire da sinistra. In questo modo otteniamo un *albero binario completo* nel quale tutte le *foglie* hanno *profondità* h o $h - 1$, i *nodi* al *livello* h sono "accatastati" a sinistra e tutti *nodi* interni hanno *grado uscente* pari a 2 eccetto al più uno. Come prima poi, per l'*altezza* h vale $h = \lfloor \log n \rfloor$.

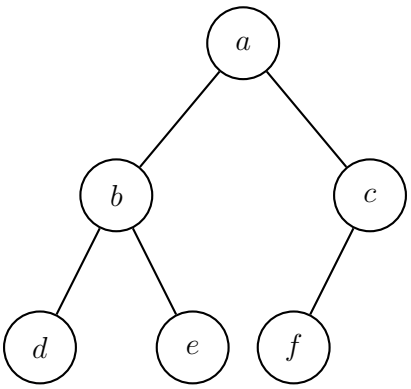


Fig. 11.2: *Albero completo*

Definizione 105 - Albero min-heap.

Un albero min-heap è un albero binario completo tale per cui il valore memorizzato in ogni nodo è minore dei valori memorizzati nei suoi figli.

Definizione 106 - Albero max-heap.

Un albero max-heap è un albero binario completo tale per cui il valore memorizzato in ogni nodo è maggiore dei valori memorizzati nei suoi figli.

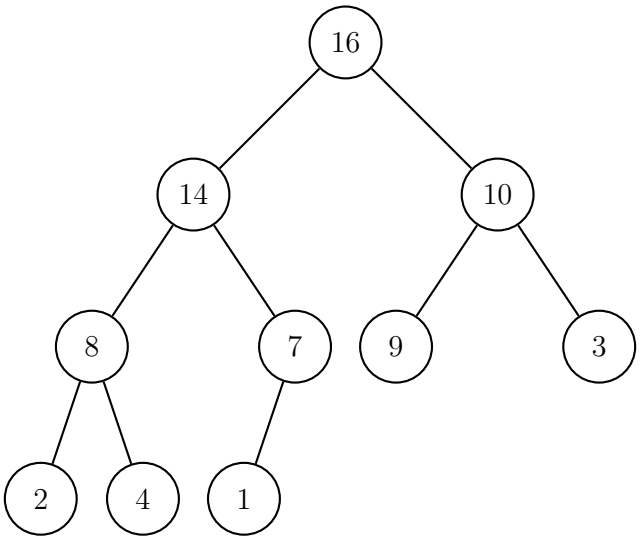


Fig. 11.3: *Albero max-heap*

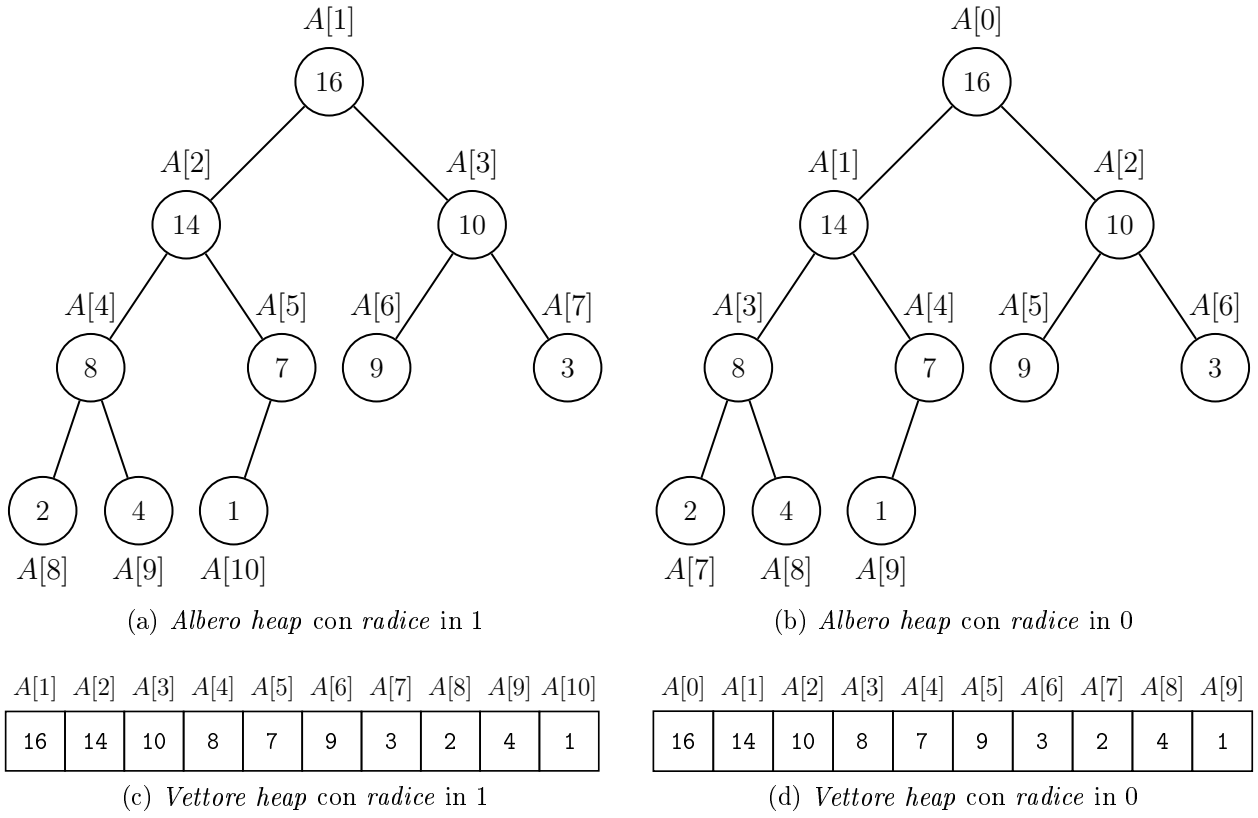
Un *albero heap* non impone un ordinamento totale tra i *figli* di un *nodo*, bensì definisce un *ordinamento parziale* e quindi soddisfa le seguenti tre proprietà:

- *Riflessività*: ogni *nodo* è maggiore o uguale a se stesso;
- *Antisimmetria*: se $n \geq m$ e $m \geq n$, allora $m = n$;
- *Transitività*: se $n \geq m$ e $m \geq r$, allora $n \geq r$;

NB. Un ordinamento parziale è una nozione più debole, ma più facile da realizzare.

Memorizzazione Un *albero heap* può essere rappresentato mediante un *vettore heap*, cioè un vettore nel quale, se la *radice* è in posizione 1, il *padre* e i *figli sinistro* e *destro* di un *nodo* in posizione i si trovano rispettivamente alle posizioni $p(i) = \lfloor i/2 \rfloor$, $l(i) = 2i$ e $r(i) = 2i + i$.

Se invece la *radice* è in posizione 0, gli indici del *padre* e dei *figli* diventano rispettivamente $p(i) = \lfloor (i-1)/2 \rfloor$, $l(i) = 2i + 1$ e $r(i) = 2i + 2$.



(a) *Albero heap* con *radice* in 1

(b) *Albero heap* con *radice* in 0

(c) *Vettore heap* con *radice* in 1

(d) *Vettore heap* con *radice* in 0

Fig. 11.4: Memorizzazione *alberi heap*

Dalle definizioni di *alberi min-heap* e *max-heap* possiamo dedurre delle proprietà sui relativi *vettori*:

- *Vettori min-heap*: $A[i] \leq A[l(i)]$ e $A[i] \leq A[r(i)]$;
- *Vettori max-heap*: $A[i] \geq A[l(i)]$ e $A[i] \geq A[r(i)]$;

11.1.2 Algoritmo HeapSort

Vediamo quindi l'algoritmo *HeapSort* per l'ordinamento in senso crescente di un vettore.

L'algoritmo ordina un vettore "in-place", costruendo prima su di esso un *albero max-heap* e poi spostando progressivamente in fondo gli elemento massimi. Ad ogni spostamento vengono ripristinate le proprietà degli *alberi max-heap*.

Le funzioni utilizzate sono due:

- `heapBuild`: costruisce un *albero max-heap* a partire da un vettore non ordinato;
- `maxHeapRestore`: ripristina le proprietà degli *alberi max-heap*;

NB. Tutte le operazioni vengono effettuate "in-place", cioè agiscono sul vettore stesso senza usare *strutture ausiliarie*. Quindi, l'*albero max-heap* non viene realmente creato, ma grazie alle sue proprietà di memorizzazione, il vettore di input viene semplicemente interpretato come fosse un *albero max-heap*.

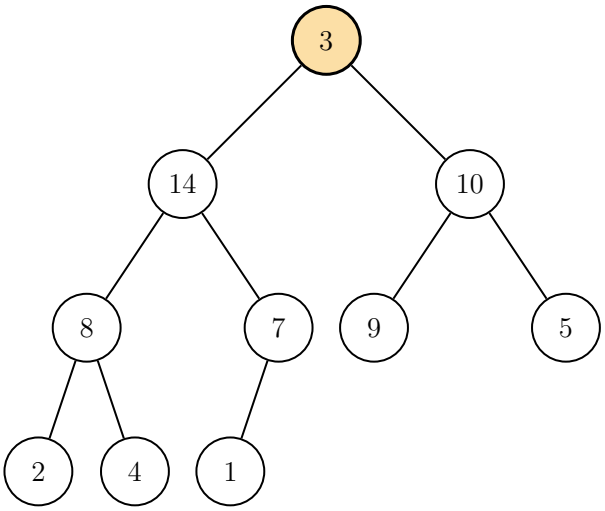
Per questo motivo, la funzione *maxHeapRestore* riceve in input un vettore A e un indice i e si occupa di garantire che al termine dell'esecuzione, l'*albero binario* con radice i sia un *albero max-heap*.

Esempio 27 - Esempio di esecuzione della maxHeapRestore.

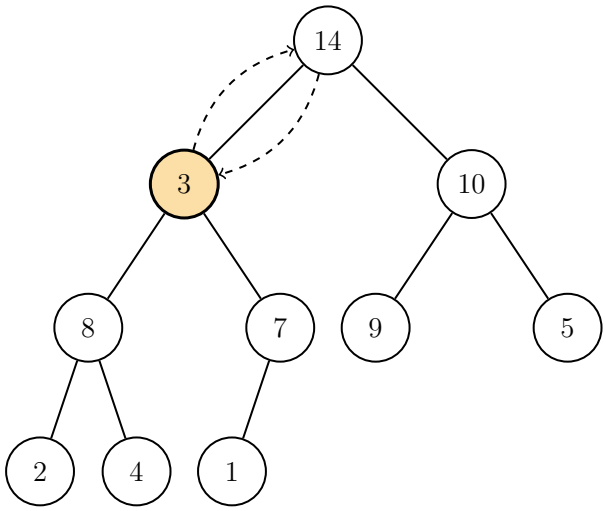
Consideriamo il seguente vettore:

$$(3, 14, 10, 8, 7, 9, 5, 2, 4, 1)$$

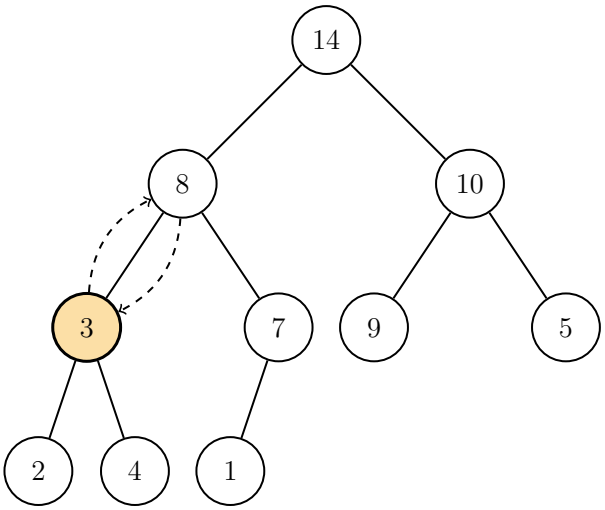
e ipotizziamo di invocare `maxHeapRestore` *su di esso usando come radice la posizione* 1.



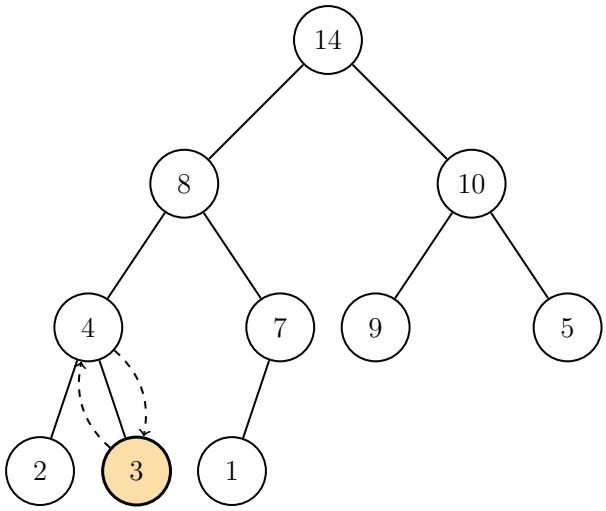
(a) Situazione iniziale



(b) Il 3 viene scambiato con il 14



(c) Il 3 viene scambiato con l'8



(d) Il 3 viene scambiato con il 4

Al termine dell'esecuzione abbiamo ottenuto un albero max-heap corretto.

Implementazione

Frammento 74 - Implementazione maxHeapRestore.

```
maxHeapRestore(ITEM[] A, int i, int dim)
   int max = i                                              % Sceglie la radice
   if (l(i) ≤ dim A[l(i)] > A[max]) then
      max = l(i)
   if (r(i) ≤ dim A[r(i)] > A[max]) then
      max = r(i)
   if (i ≠ max) then                              % Se i == max l'albero è apposto
      swap(A, i, max)       % Scambia la radice e il maggiore tra i suoi figli
      maxHeapRestore(A, max, dim)   % Controlla il sottoalbero con radice max
```

Complessità Ad ogni chiamata vengono eseguiti $O(1)$ confronti e se il *nodo* i non è massimo, si invoca ricorsivamente `maxHeapRestore` su uno dei *figli*. L'esecuzione nel caso peggiore termina al raggiungimento di una *foglia*, quindi il costo è limitato dall'*altezza* dell'*albero*, ovvero $T(n) = O(\log n)$.

Dimostrazione di correttezza

Dimostrazione. Dobbiamo dimostrare che al termine dell'esecuzione l'*albero* con *radice* in $A[i]$ rispetta le proprietà degli *alberi max-heap*. Procediamo per induzione sull'*altezza* h dell'*albero*.

Caso base $h = 0$. L'*albero* è composto da un solo *nodo* e quindi rispetta per forza le proprietà.

Passo induttivo Ipotizziamo che l'algoritmo funzioni correttamente su tutti gli *alberi* di *altezza* minore di h. A questo punto si configurano tre casi:

- *Caso 1*: $A[i] \geq A[l(i)]$ e $A[i] \geq A[r(i)]$, ovvero il *nodo* $A[i]$ è più grande dei propri *figli* e quindi l'*albero radicato* in $A[i]$ è un *albero max-heap* e l'algoritmo termina;
- *Caso 2*: $A[i] < A[l(i)]$ e $A[r(i)] < A[l(i)]$, ovvero il *figlio sinistro* è più grande sia del *padre* che del *fratello*. Quindi, vengono scambiati di posizione il *figlio sinistro* e il *padre*. A questo punto valgono $A[i] \geq A[l(i)]$ e $A[i] \geq A[r(i)]$, ma il *sottoalbero* con *radice* in $A[l(i)]$ potrebbe non rispettare più le proprietà per gli *alberi max-heap*. Di conseguenza, l'algoritmo continua applicando ricorsivamente `maxHeapRestore` sull'*albero* con *radice* in $A[l(i)]$, ma siccome quell'*albero* ha *altezza* minore di h, per ipotesi induttiva, l'algoritmo riesce correttamente a garantire il rispetto di tutte le proprietà;
- *Caso 3*: $A[i] < A[r(i)]$ e $A[l(i)] < A[r(i)]$, ovvero il *figlio destro* è più grande sia del *padre* che del *fratello*. Quindi, vengono scambiati di posizione il *figlio destro* e il *padre*. A questo punto valgono $A[i] \geq A[l(i)]$ e $A[i] \geq A[r(i)]$, ma il *sottoalbero* con *radice* in $A[r(i)]$ potrebbe non rispettare più le proprietà per gli *alberi max-heap*. Di conseguenza, l'algoritmo continua applicando ricorsivamente `maxHeapRestore` sull'*albero* con *radice* in $A[r(i)]$, ma siccome quell'*albero* ha *altezza* minore di h, per ipotesi induttiva, l'algoritmo riesce correttamente a garantire il rispetto di tutte le proprietà;

□

Consideriamo ora la funzione `heapBuild`. La funzione riceve in input un vettore da ordinare e, per le proprietà di memorizzazione degli *alberi heap*, sappiamo che tutti i *nodi* con indici compresi tra $\lfloor n/2 \rfloor + 1$ e n sono *foglie*, ovvero *alberi heap* contenenti un solo elemento.

La funzione `heapBuild` quindi, non fa altro che applicare `maxHeapRestore` a tutti i *nodi*, partendo dal primo che non è una *foglia*, cioè da $A[\lfloor n/2 \rfloor]$, e risalendo fino alla *radice*.

Implementazione

Frammento 75 - Implementazione heapBuild.

```
heapBuild(ITEM[] A, int n)
   for (i = ⌊n/2⌋ downto 1) do
      maxHeapRestore(A, i, n)
```

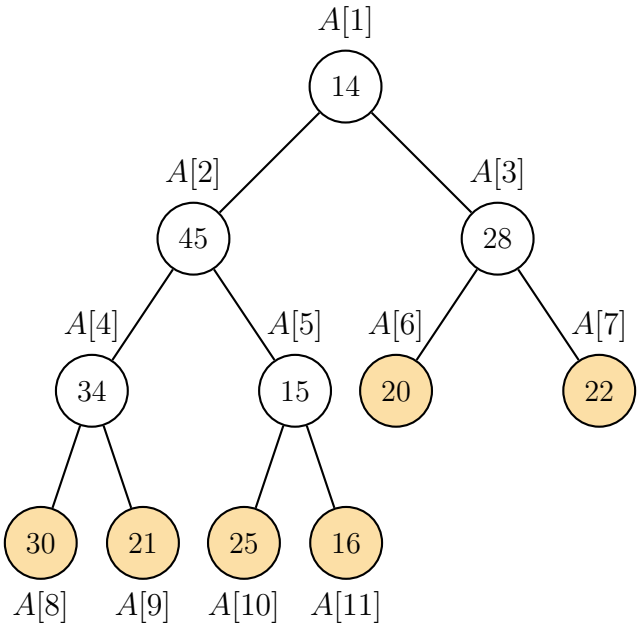


Fig. 11.5: `maxHeapRestore` viene applicata solo ai *nodi* interni

Dimostrazione di correttezza

Dimostrazione. Dimostriamo la seguente *invariante di ciclo*:
All'inizio di ogni iterazione del ciclo `for`*, i nodi* $[i+1 \dots n]$ *sono radici di alberi heap.*

Inizializzazione All'inizio $i = \lfloor n/2 \rfloor$. Supponiamo che $\lfloor n/2 \rfloor + 1$ non sia una *foglia* e che quindi abbia almeno il *figlio sinistro*. Se così fosse, il *figlio* dovrebbe trovarsi alla posizione $2(\lfloor n/2 \rfloor + 1) = 2\lfloor n/2 \rfloor + 2$, ma ciò sarebbe assurdo perché le posizioni $n+1$ e $n+2$ eccedono la dimensione massima che n. La stessa dimostrazione vale anche per tutti gli indici successivi.

Conservazione È possibile applicare `maxHeapRestore` a ogni *nodo* $i \in [\lfloor n/2 \rfloor + 1 \dots n]$ perché entrambi i *nodi* alle posizioni $2i < 2i + 1 \leq n$ sono *radici* di *alberi heap*. Al termine di ogni iterazioni, i *nodi* $[i \dots n]$ sono *radici* di *alberi heap*.

Conclusione Al termine, $i = 0$ e quindi il *nodo* 1 è *radice* di un *albero heap*. □

Complessità Sicuramente possiamo dire che `heapBuild` è un $O(n \log n)$ in quanto il costo di `maxHeapRestore` è $O(\log n)$ e viene invocata $\lfloor n/2 \rfloor = O(n)$ volte. Ma è veramente questo il costo?

In realtà, le operazioni `maxHeapRestore` vengono eseguite un numero decrescente di volte in un *albero* di *altezza* crescente. E in particolare, vale la seguente tabella:

Altezza	**# esecuzioni**
0	$\lfloor n/2^1 \rfloor$
1	$\lfloor n/2^2 \rfloor$
2	$\lfloor n/2^3 \rfloor$
...	...
h	$\lfloor n/2^{h+1} \rfloor$

Se questo è vero, possiamo scrivere la *funzione di ricorrenza* come:

$$
\begin{aligned}
T(n) &\leq \sum_{h=1}^{\lfloor n/2 \rfloor} \frac{n}{2^{h+1}} h \\
&= n \sum_{h=1}^{\lfloor n/2 \rfloor} \left(\frac{1}{2}\right)^{h+1} h \\
&= \frac{n}{2} \sum_{h=1}^{\lfloor n/2 \rfloor} \left(\frac{1}{2}\right)^{h} h \\
&\leq \frac{n}{2} \sum_{h=1}^{+\infty} \left(\frac{1}{2}\right)^{h} h && \textit{Estensione ad infinito della somamtoria} \\
&= \frac{n}{2} \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} && \textit{Serie geometrica infinita decrescente} \\
&= \frac{n}{2} \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} \\
&= \frac{n}{2} 2 \\
&= n = O(n)
\end{aligned}
$$

Implementazione

Frammento 76 - Implementazione dell'algoritmo HeapSort.

```
heapsort(ITEM[] A, int n)
   heapBuild(A, n)
   for (i = n downto 2) do
      swap(A, 1, i)                  % L'elemento massimo viene spostato in fondo
      maxHeapRestore(A, 1, n - 1)                    % Ripristina le proprietà
```

In pratica, l'*HeapSort* inizia costruendo un *albero max-heap* sul vettore da ordinare. Ad ogni ciclo il primo elemento sarà sicuramente il massimo e quindi viene scambiato di posto con l'elemento in posizione i. Dopo ogni scambio viene invocata `maxHeapRestore` per ripristinare le proprietà degli *alberi max-heap*. Il tutto viene ripetuto fino a quando i non diventa 1.

Complessità La funzione `heapBuild` costa $\Theta(n)$, mentre le `maxHeapRestore` ha un costo $\Theta(\log i)$ per i che varia da n a 2. La *funzione di ricorrenza* è la seguente:

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

Dimostrazione di correttezza

Dimostrazione. Dimostriamo la seguente *invariante di ciclo*:
All'inizio di ogni passo i il sottovettore $A[i+1 \dots n]$ è ordinato e ogni elemento in $A[1 \dots i]$ è minore o al più uguale a ogni elemento in $A[i+1 \dots n]$. Inoltre, l'elemento $A[1]$ è la radice di un albero heap di dimensione i.

Inizializzazione Dopo la `heapBuild` il primo elemento è la *radice* di un *albero max-heap* e quindi è maggiore dei propri *figli* e, per la proprietà transitiva, anche dei *figli* dei *figli* fino alle *foglie*. Alla prima iterazione quindi, $A[1]$ è il massimo e viene scambiato di posto con $A[i] = A[n]$. A quel punto, $A[n]$ è l'elemento massimo e il sottovettore $A[1 \dots n-1]$ viene riorganizzato secondo le regole della `maxHeapRestore` portandone il massimo in $A[1]$.

Conservazione Per ogni $i \in [2 \dots n]$, il sottovettore $A[i+1 \dots n]$ è ordinato in senso crescente e in posizione $A[1]$ si trova l'elemento massimo del sottovettore $A[1 \dots i]$. Al termine dell'iterazione, $A[i \dots n]$ è ordinato e $A[1]$ è il massimo in $A[1 \dots i-1]$.

Conclusione Al termine, $i = 1$ e quindi il sottovettore $A[2 \dots n]$ è ordinato. Poiché vale sicuramente $A[1] \leq A[2]$, possiamo dire che tutto il vettore $A[1 \dots n]$ è ordinato in senso crescente. □

11.1.3 Implementazione di code a priorità

Prima di vedere l'implementazione di una *coda a priorità*, vediamo la definizione di un **PRIORITYITEM** e l'implementazione della funzione `swap`.

Frammento 77 - PRIORITYITEM.

```
int priority                                  % Valore di priorità
ITEM value                                    % Valore
int pos                                       % Posizione nel vettore
```

Frammento 78 - Implementazione funzione swap.

```
swap(PRIORITYITEM H, int i, int j)
   PRIORITYITEM temp = H[i]
   H[i] = H[j]
   H[j] = temp
   H[i].pos = i
   H[j].pos = j
```

Di seguito quindi, vediamo l'implementazione di una *min-priority queue* come vettore di coppie $\langle valore, priorità \rangle$. Ovviamente, tutto ciò che vedremo vale simmetrico per le *max-priority queue*.

Frammento 79 - Min-priority queue.

```
int capacity                                  % Dimensione massima della coda
int dim                    % Numero di elementi attualmente presenti nella coda
PRIORITYQUEUE[] H                                          % Vettore heap
```

```
PRIORITYQUEUE PriorityQueue(int n)
   PRIORITYQUEUE t = new PRIORITYQUEUE
   t.capacity = n
   t.dim = 0
   t.H = new PRIORITYITEM[1...n]
   return t

PRIORITYITEM insert(ITEM x, int p)
   precondition:  dim < capacity
   dim = dim + 1
   H[dim] = new PRIORITYITEM                % Aggiunge l'elemento in fondo
   H[dim].value = x
   H[dim].priority = p
   H[dim].pos = dim
   int i = dim
      while (i > 1 and H[i].priority < H[p(i)].priority) do
         swap(H, i, p(i))     % Sposta il nuovo elemento nella posizione giusta
         i = p(i)
      return H[i]

minHeapRestore(PRIORITYITEM[] A, int i, int n)
   int min = i
   if (l(i) ≤ dim and A[l(i)].priority < A[min].priority) then
      min = l(i)
   if (r(i) ≤ dim and A[r(i)].priority < A[min].priority) then
      min = r(i)
   if (i ≠ min) then
      swap(A, i, min)
      minHeapRestore(A, min, dim)

ITEM deleteMin()
   precondition:  dim > 0
   swap(H, 1, dim)                      % Scambia il primo e l'ultimo elemento
   dim = dim - 1
   minHeapRestore(H, 1, dim)                          % Ripristina le proprietà
   return H[dim + 1].value       % Restituisce il valore dell'elemento rimosso

ITEM min()
   precondition:  dim > 0
   return H[1].value

decrease(PRIORITYITEM x, int p)
   precondition:  p < x.priority
   x.priority = p
   int i = x.pos
   while (i > 1 and H[i].priority < H[p(i)].priority) do
      swap(H, i, p(i))    % Sposta l'elemento modificato nella posizione giusta
      i = p(i)
```

Complessità Tutte le operazioni che modificano gli *heap* ne ripristinano anche le proprietà. In particolare, questo viene fatto nel *cammino radice-foglia* dalla `deleteMin` e in quello *nodo-radice* dalla `insert` e dalla `decrease`. Poiché l'*altezza* è $\lfloor \log n \rfloor$, il costo di tali operazioni è $O(\log n)$.

La *complessità* di tutte le operazioni è riassunta dalla seguente tabella:

Operazione	Costo
`insert`	$O(\log n)$
`deleteMin`	$O(\log n)$
`min`	$\Theta(1)$
`decrease`	$O(\log n)$

11.2 Insiemi disgiunti

In alcune applicazioni (e.g. ricerca delle *componenti connesse* di un *grafo*) siamo interessati a gestire una collezione $S = \{S_1, \ldots, S_n\}$ di *insiemi dinamici disgiunti*, ovvero tali per cui:

$$S_i \cap S_j = \emptyset \quad \forall i, j : i \neq j$$

$$\bigcup_{1=1}^{n} S_i = S \quad \text{con } n = |S|$$

Definiamo quindi la struttura dati *merge-find set* con le seguenti primitive:

- Creazione di n *insiemi disgiunti* composti da un elemento ciascuno;
- `find`: identificazione dell'*insieme* a cui appartiene un elemento;
- `merge`: unione di due *insiemi*;

Ogni insieme è identificato da un proprio elemento che viene designato come *rappresentante*. Di conseguenza, la primitiva `find` deve restituire il *rappresentante* dell'*insieme* in cui si trova elemento cercato. Definito l'elemento *rappresentante*, questo può essere cambiato solo in caso di unione con un altro *insieme*.

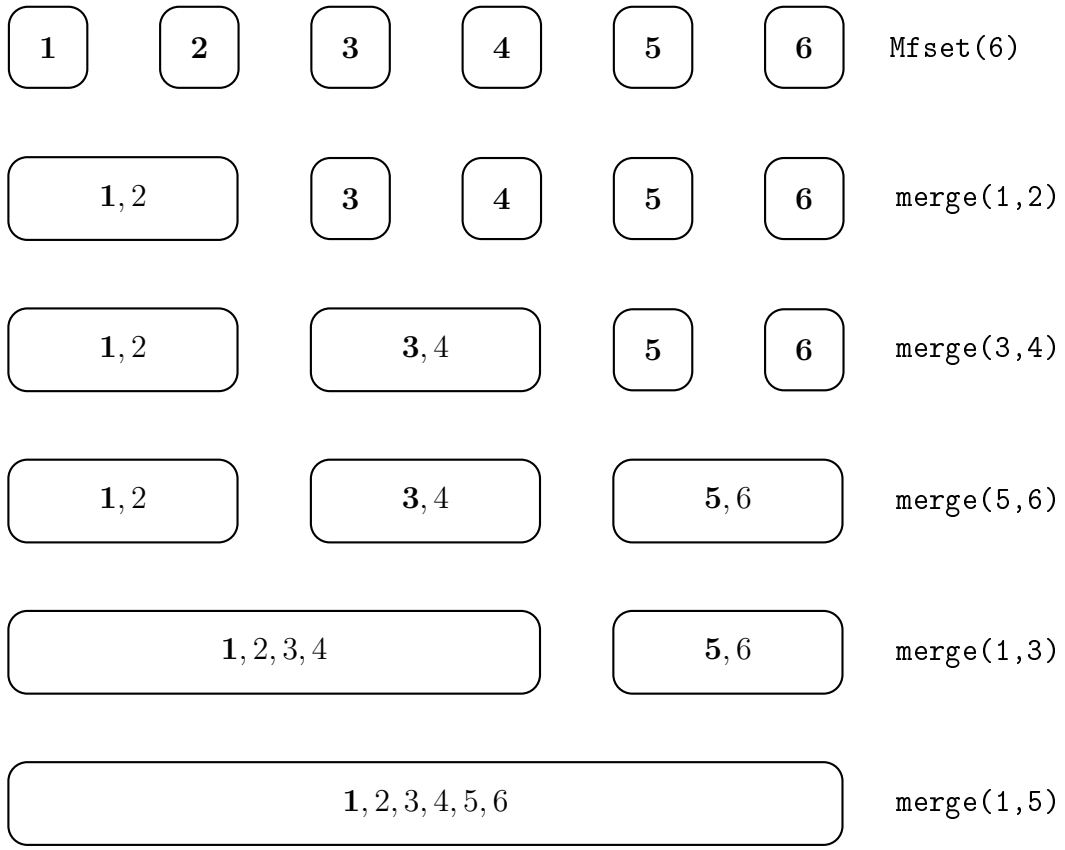
NB. Per semplicità di esposizione, ipotizziamo che gli elementi siano identificati da un valore intero $1 \ldots n$ e che l'associazione con il valore vero e proprio sia memorizzata esternamente.

Specifica

Frammento 80 - Merge-find set.

```
% Crea n insiemi {1},...,{n}
MFSET Mfset(int n)
% Restituisce il rappresentante dell'insieme contenente x
int find(int x)
% Unisce gli insiemi che contengono x e y
merge(int x, int y)
```

Esempio 28 - Esempio di utilizzo di un merge-find set.



Utilizzando la struttura dati così definita, possiamo semplificare molto l'algoritmo per la ricerca delle *componenti connesse* di un *grafo* $G = (V, E)$. In particolare, è sufficiente inizializzare la struttura vedendo ogni *nodo* come un *insieme*, quindi procedere ad unire `merge(u,v)` per ogni *arco* $(u, v) \in E$. Al termine, ogni *insieme disgiunto* rappresenterà una *componente connesse*.

Frammento 81 - Ricerca delle componenti connesse con merge-find set.

```
MFSET cc(GRAPH G)
   MFSET M = Mfset(G.size())
   foreach (u ∈ G.V()) do
      foreach (v ∈ G.adj(u)) do
         M.merge(u,v)
   return M
```

La *complessità* di questa versione è $T(n) = O(n + m)$ dove m è il numero di invocazioni di `merge`.

NB. Questa versione dell'algoritmo è particolarmente indicata nel caso di *grafi dinamici* ovvero *grafi* nei quali vengono spesso aggiunti nuovi *archi*.

Ma quanto costa eseguire una `merge` e come è implementata?

Ci sono diverse implementazioni possibili tra le quali implementazioni basate su *insiemi di liste* e su *insiemi di alberi* (i.e. *foreste*).

11.2.1 Implementazione basata su insiemi di liste

Ogni *insieme* viene rappresentato da una *lista concatenata* in cui ciascun elemento, oltre al proprio valore, contiene un puntatore all'elemento successivo e un puntatore al *rappresentante*. Il *rappresentate* è il primo elemento della lista.

La primitiva `find` è banale, perché dato un *elemento* è sufficiente restituire il valore del puntatore al *rappresentante*, quindi richiede un tempo $O(1)$. Per la `merge`, si prende il *rappresentate* di un elemento, lo si "appende" alla *lista* del primo e si aggiornano tutti i *rappresentati* della *lista* "appesa". Nel caso pessimo, eseguire n operazioni `merge` costa $O(n^2)$, quindi il *costo ammortizzato* è $O(n)$.

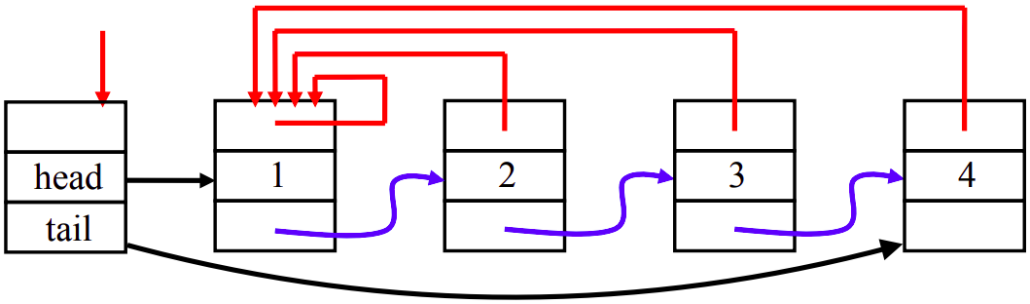


Fig. 11.6: *Merge-find set* implementati come *insiemi di liste*

11.2.2 Implementazione basata su insiemi di alberi

Ogni *insieme* viene rappresentato da un *albero* in cui ogni *nodo* contiene il proprio valore e un puntatore al *padre*. La *radice* è il *rappresentante* dell'*insieme* e il puntatore al *padre* punta a se stessa.

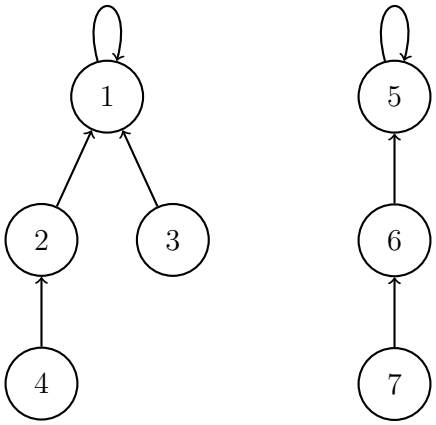


Fig. 11.7: *Merge-find set* implementati come *insiemi di alberi*

In questo caso, nel `find` si risale l'intero *albero* fino alla *radice* e, nel caso pessimo, questo costa $O(n)$. Per fare la `merge` invece, è sufficiente cambiare il puntatore al *padre* della *radice* in uno dei due *alberi* in modo che punti all'altra *radice*. Il costo, se non consideriamo quello per la ricerca dei *rappresentanti*, è $O(1)$.

11.2.3 Implementazioni con tecniche euristiche

Possiamo migliorare le implementazioni viste sfruttando qualche piccolo accorgimento?

Nel caso dei *merge-find set* basati su *liste* potremmo provare ad eseguire la `merge` modificando i puntatori della *lista* più corta. Per quanto riguarda gli *alberi* invece, potremmo ridurre il costo della `find` minimizzando l'*altezza* degli *alberi*. Tecniche di questo tipo sono chiamate *euristiche*, e gli algoritmi che le usano sono detti *euristici*.

Definizione 107 - Algoritmo euristico.

È detto euristico un algoritmo progettato per risolvere un problema più velocemente, qualora i metodi tradizionali non siano sufficienti, oppure per ricavare una soluzione approssimata, qualora non sia possibile ricavarne una esatta.

Euristica sul peso Approfondendo l'intuizione sulla lunghezza delle *liste*, possiamo memorizzare in ogni *lista* l'informazione sulla propria lunghezza e implementare la `merge` in modo che modifichi i puntatori dei *nodi* che stanno nella *lista* più corta. La lunghezza può essere mantenuta in $O(1)$ ed è possibile dimostrare che in questo tipo di implementazione il *costo ammortizzato* della `merge` è $O(\log n)$.

Euristica sul rango Abbiamo detto che per gli *alberi* conviene cercare di minimizzare le *altezze*. Quindi, associamo ad ogni *nodo* k il proprio *rango*, $rank[k]$, definito come segue:

Definizione 108 - Rango di un nodo.

È definito rango di un nodo, il numero di archi del più lungo cammino tra quel nodo e una delle proprie foglie.

NB. Possiamo anche definire il *rango* di un *nodo* come l'*altezza* del *sottoalbero* in esso radicato.

A questo punto, per ridurre il costo della `find` è sufficiente modificare la `merge` in modo che nel caso di *alberi* di con *ranghi* diversi, sia l'*albero* con *rango* minore ad essere "agganciato" all'altro. In questa situazione, l'*altezza* dell'*albero* con *rango* maggiore non cambia, ma se invece i due *alberi* avessero pari *rango*, l'*altezza* finale sarebbe incrementata di 1.

Definizione 109 - Legame tra il rango di un albero e il proprio numero di nodi.

Un albero MFSET con radice r e ottenuto tramite euristica sul rango ha almeno $2^{rank[r]}$ nodi.

Dimostrazione. Procediamo per induzione sul *rango* di r.

Caso base: $\boldsymbol{rank[r] = 0}$ Dopo l'inizializzazione della struttura ogni *albero* ha $2^{rank[r]} = 1$ *nodi*;

Passo induttivo: $\boldsymbol{rank[r] > 0}$ Facendo la `merge` di due *alberi* x e y di *rango* $rank[x]$ e $rank[y]$ distinguiamo due casi:

1. $\boldsymbol{rank[x] > rank[y]}$: il *rango* dell'*albero* r ottenuto è $rank[r] = rank[x]$. Per induzione, il numero di *nodi* di r è almeno pari a $2rank[x] + 2^{rank[y]}$ e quindi è maggiore di $2^{rank[x]}$;
2. $\boldsymbol{rank[x] = rank[y]}$: il *rango* dell'*albero* r ottenuto è $rank[r] = rank[x] + 1$. Per induzione, il numero di *nodi* di r è almeno pari a $2^{rank[x]} + 2^{rank[y]} = 2^{rank[x]} + 2^{rank[x]} = 2^{rank[x]+1}$;

□

Dal teorema appena dimostrato possiamo ricavare il seguente corollario:

Definizione 110 - Corollario.

Un albero MFSET con radice r e n nodi ha un altezza inferiore a $\log n$.

Dimostrazione. Vale la seguente relazione:

$$n \geq 2^{rank[r]} \Leftrightarrow rank[r] \leq \log n$$

□

Detto questo, la *complessità* della primitiva `find` è $O(\log n)$.

Euristica di compressione dei cammini Poiché il costo della `find` è legato all'*altezza* degli *alberi*, se tutti avessero *altezza* pari a 1, cioè se il *padre* di ogni *nodo* fosse la *radice*, riusciremmo a ridurre il costo di ogni invocazione a $O(1)$. Per raggiungere questo obiettivo, possiamo modificare l'implementazione della `find` in modo che quando risaliamo l'*albero*, ogni *nodo* modifichi il proprio puntatore al *padre* indicando il *padre* del proprio *padre*. In questo modo, la prima invocazione di `find` costa $O(\log n)$, mentre ricerche successivo su *nodi* già visitati, costeranno $O(1)$.

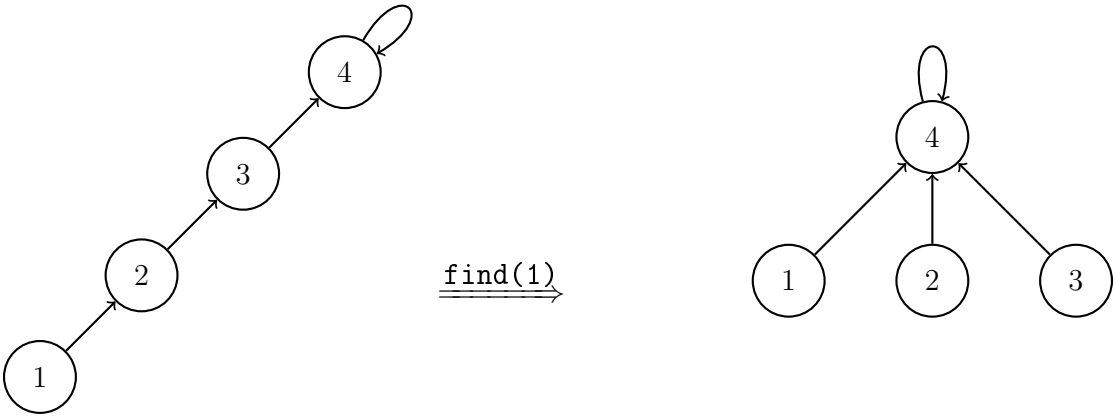


Fig. 11.8: Esempio di *compressione dei cammini*

NB. Applicando entrambe le *euristiche*, il *rango* non indica più l'*altezza* del *nodo*, bensì un limite superiore. Questo perché mantenere il valore corretto diventa troppo difficile (i.e. complesso) e comunque non è necessario.

Complessità Utilizzando entrambe le *tecniche euristiche* viste, il *costo ammortizzato* di m operazioni `merge-find` in un *insieme* di n elementi è $O(m \cdot \alpha(n))$, dove $\alpha(n)$ è la *funzione inversa di Ackermann* che ha una crescita estremamente lenta[1]. Di conseguenza, il *costo ammortizzato* di una singola operazione è $O(1)$.

[1] Ad esempio, per $n \leq 2^{65536}$, $\alpha(n) \leq 5$

Frammento 82 - Merge-find set basati su insiemi di alberi.

```
int[] parent
int[] rank

MFSET Mfset(int n)
   MFSET M = new MFSET
   t.parent = new int[1...n]
   t.rank = new int[1...n]
   for (i = 1 to n) do
      t.parent[i] = i
      t.rank[i] = 1
   return t

int find(int x)
   if (parent[x] ≠ x) then
      parent[x] = find(parent[x])
   return parent[x]
```

```
merge(int x, int y)
   int r_x = find(x)
   int r_y = find(y)
   if (r_x ≠ r_y) then
      if (rank[r_x] > rank[r_y]) then
         parent[r_y] = r_x
      else if (rank[r_y] > rank[r_x])
         then
         parent[r_x] = r_y
      else
         parent[r_x] = r_y
         rank[r_y] = rank[r_y] + 1
```

11.2.4 Complessità

Riassumendo, le *complessità* delle possibili implementazioni dei *merge-find set* sono le seguenti[2]:

Algoritmo	`find`	`merge`
Liste	$O(1)$	$O(n)$
Alberi	$O(n)$	$O(1)^+$
Liste con euristica sul peso	$O(1)$	$O(\log n)^*$
Albero con euristica sul rango	$O(\log n)$	$O(1)^+$
Albero con euristica sul rango e compressione dei cammini	$O(1)^*$	$O(1)$

[2] $*$: *complessità ammortizzata*;

$+$: Consideriamo solo il costo della `merge`, senza la ricerca dei *rappresentanti* con la `find`;

Capitolo Nr.12

Programmazione dinamica

12.1 Introduzione

Finora, l'unica tecnica di risoluzione di problemi, o di ricerca di algoritmi, che abbiamo visto è il Divide-et-impera. La *programmazione dinamica* è un'altra tecnica basata su un approccio molto simile.

Entrambe prevedono di spezzare il problema di partenza in sotto-problemi più semplici e di ricostruire la soluzione del problema originale a partire dalle soluzioni dei sotto-problemi. La differenza fondamentale sta nel fatto che la *programmazione dinamica* risolve ogni sotto-problema una sola volta, mentre il *Divide-et-impera* non pone questo vincolo.

Per realizzare ciò, le soluzioni di tutti i sotto-problemi vengono salvate in una tabella che viene consultata ogni volta che è necessario risolvere uno dei sotto-problemi. In particolare, se la tabella non contiene la soluzione al sotto-problema considerato, questa viene calcolata e aggiunta alla tabella. In caso contrario, viene sfruttata la soluzione già nota.

12.1.1 Approccio generale

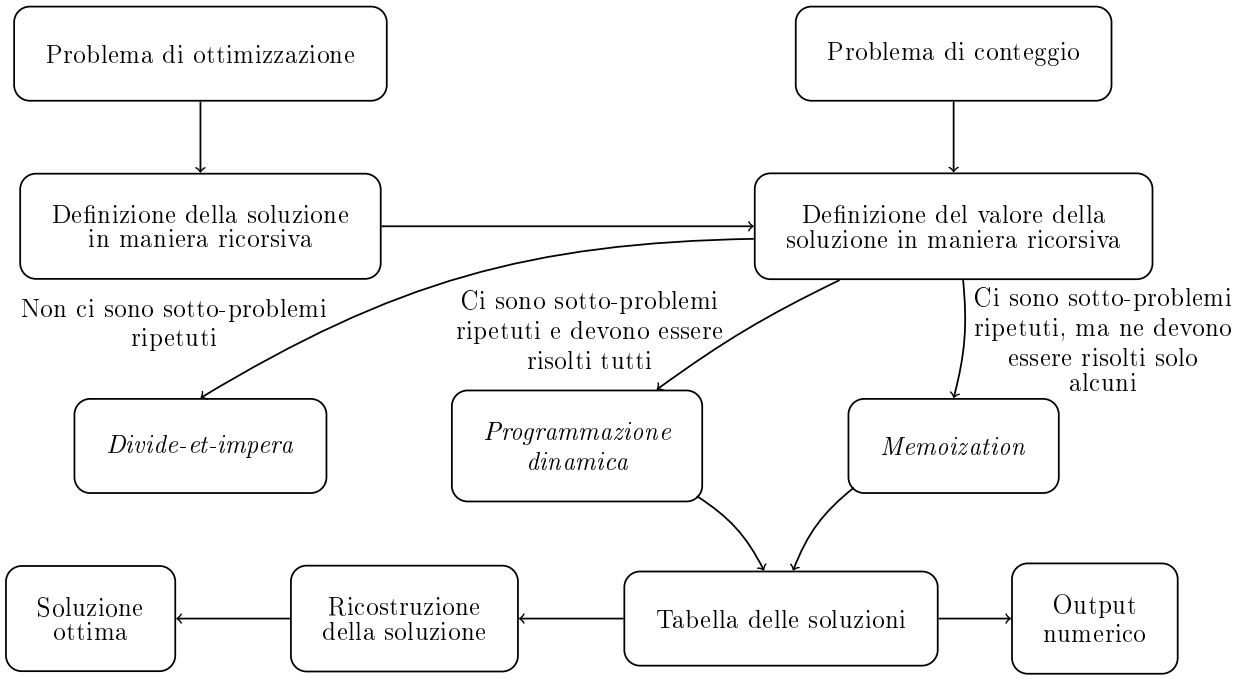


Fig. 12.1: Schema generale di approccio

Lo schema di cui sopra mostra un approccio generale alla scelta della tecnica risolutiva più adatta al tipo di problema da risolvere. In particolare, distinguiamo la soluzione del problema dal proprio valore, ad esempio, nella ricerca del *cammino breve* tra due *nodi*, la soluzione è il *cammino*, mentre il valore potrebbe esserne la lunghezza.

Una volta definiti questi parametri, se la suddivisione del problema non porta a dover risolvere più volte uno stesso sotto-problema, possiamo procedere ad implementare la soluzione sfruttando l'approccio *Divide-et-impera* classico. Caso contrario, ricadiamo nei casi d'uso della *programmazione dinamica*, ma dobbiamo ancora fare una distinzione: la risoluzione di alcuni dei sotto-problemi potrebbe non essere necessaria per arrivare alla soluzione del problema di partenza. Se il caso è questo, procediamo con la tecnica della *memoization* che non è altro se non un approccio top-down alla *programmazione dinamica* "classica".

In ogni caso, utilizzando la *programmazione dinamica* si arriva alla definizione di una *tabella delle soluzioni* e, se il problema è un *problema di conteggio*, da cui possiamo ricavare direttamente la soluzione. Se invece stiamo considerando un *problema di ottimizzazione*, la soluzione dovrà essere ricavata attraverso un processo di "ricostruzione".

12.2 Gioco del domino

Vediamo adesso un primo esempio di utilizzo della *programmazione dinamica* e di come la soluzione si differenzi, soprattutto in termini di efficienza, da una classica soluzione *divide-et-impera*.

Problema 5 - Domino lineare.

Il gioco del domino è basato su tessere di dimensione 2×1. *Scrivere un algoritmo che prenda in input un valore intero positivo* n *e restituisca il numero di modi in cui è possibile disporre le tessere del gioco in modo che creino un rettangolo* $2 \times n$.

Ad esempio, per n pari a 0, 1, 2, 3, 4 l'algoritmo dovrà restituire 1, 1, 2, 3, 5.

NB. Per $n = 0$, restituiamo 1 perché per ottenere un rettangolo di dimensione 2×0 c'è un solo modo: non disporre nessuna tessera.

Basandoci sullo schema di cui sopra possiamo identificare questo problema con un *problema di conteggio* e quindi la prima cosa da fare è la "definizione del valore della soluzione in maniera ricorsiva".

12.2.1 Approccio basato su divide-et-impera

Chiamiamo $DP[n]$ la funzione che per ogni valore $n \in \mathbb{N}$ restituisce il numero di disposizioni possibili delle tessere. La funzione DP è ricorsiva, quindi, prima di tutto identifichiamo il caso base: poiché per $n = 0$ e $n = 1$ $DP[n] = 1$, possiamo decidere che $DP[n] = 1$ per $n \leq 1$.

A questo punto, consideriamo il caso più complesso in cui dobbiamo esaminare un rettangolo $2 \times n$ arbitrariamente grande. Poiché le tessere del domino hanno dimensione 2×1 e non posso essere posizionate in modo sfalsato, i possibili posizionamenti di ogni tessera sono due: in verticale o in orizzontale. Se ipotizzassimo di mettere una tessera verticale a destra o a sinistra del rettangolo, il problema si ridurrebbe all'analisi di un rettangolo $2 \times (n - 1)$. Se invece posizionassimo la tessera in orizzontale otterremmo una figura composta da un rettangolo $2 \times (n - 2)$ e da un altro rettangolo più piccolo di dimensione 1×2. Chiaramente, il rettangolo più piccolo corrisponde ad una tessera orizzontale posta sopra o sotto quella che abbiamo già messo e, di conseguenza, arriviamo alla conclusione che qual'ora si disponga una tessera in orizzontale se ne debba sempre aggiungere anche una seconda sopra o sotto la prima.

Fatta questa riflessione possiamo terminare dicendo che, dato un generico rettangolo $2 \times n$, il numero di possibili disposizioni di tessere è pari a $DP[n-1] + DP[n-2]$ e che quindi la definizione della funzione ricorsiva è:

$$DP[n] = \begin{cases} 1 & n \leq 1 \\ DP[n-1] + DP[n-2] & n > 1 \end{cases}$$

NB. Le soluzioni per $n-1$ e $n-2$ si sommano perché per ogni rettangolo $2 \times n$ ho la possibilità di mettere una tessera verticale o due tessere orizzontali.

Provando a valutare la funzione per valori crescenti di n otteniamo la seguente sequenza:

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

che corrisponde esattamente alla *sequenza di Fibonacci*. L'implementazione dell'algoritmo corrisponde perciò all'implementazione della funzione per il calcolo di tale sequenza.

Frammento 83 - Prima implementazione della soluzione.

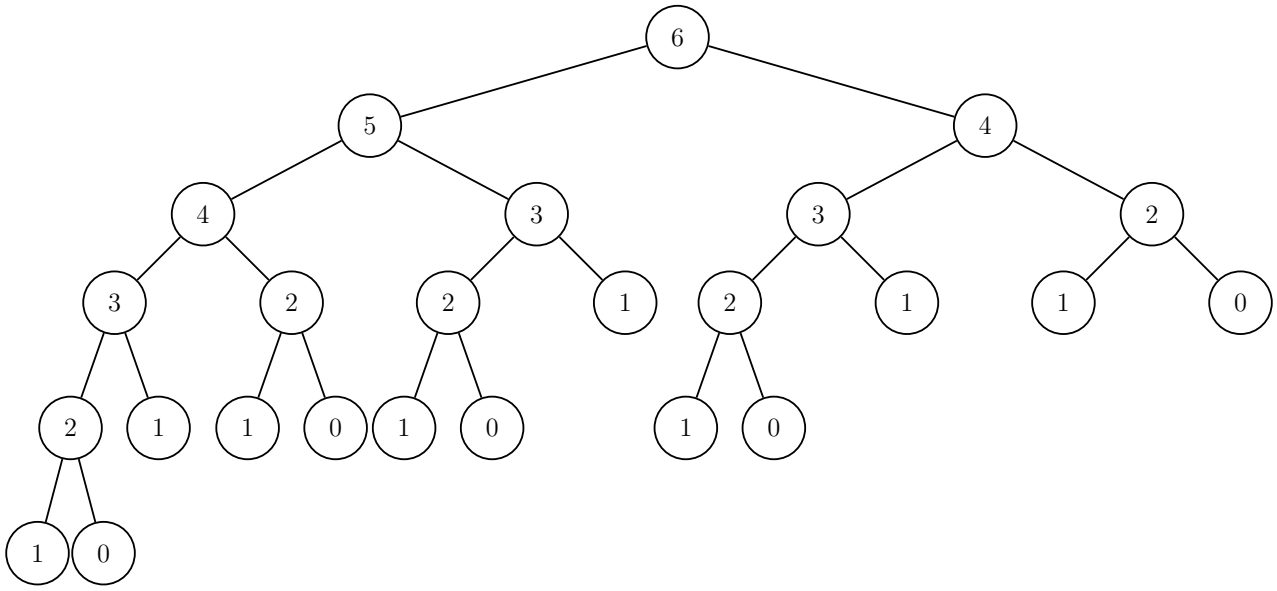
```
int domino1(int n)
   if (n ≤ 1) then
      return 1
   else
      return domino1(n-1) + domino1(n-2)
```

La *funzione di ricorrenza* associata a questa funzione è la seguente:

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

Per il *Teorema delle ricorrenze lineari di ordine costante* la *forma chiusa* di tale funzione è $T(n) = \Theta(2^n)$. Un costo esponenziale è tutt'altro che efficiente, non possiamo fare meglio di così?

Proviamo ad esaminare l'*albero delle invocazioni* per $n = 6$:



Se ogni *nodo* rappresenta un sotto-problema, è evidente come molti sotto-problemi vengano risolti più di una volta. Questa è sicuramente un'inefficienza, quindi, proviamo ad evitare di eseguire invocazioni ridondanti.

12.2.2 Approccio basato su programmazione dinamica

La presenza di sotto-problemi ripetuti ci costringe quindi ad evitare una soluzione realizzata con un classico *divide-et-impera*, e ad utilizzare invece la *programmazione dinamica*, con conseguente costruzione di una *tabella delle soluzioni*. In particolare, tale tabella conterrà un elemento per ogni sotto-problema da risolvere.

Poiché, per ogni $n \in \mathbb{N}$ il calcolo della soluzione si basa su i due valori precedenti di n, ci troviamo in una situazione in cui tutti i sotto-problemi devono essere risolti e quindi procediamo con la *programmazione dinamica* classica o bottom-up. Ciò significa che per costruire la *tabella delle soluzioni* partiamo dai casi base e risaliamo fino ad n.

Frammento 84 - Seconda implementazione della soluzione.

```
int domino2(int n)
    int[] DP = new int[0...n]                    % Tabella delle soluzioni
    DP[0] = 1
    DP[1] = 1
    for (i = 2 to n) do
        DP[i] = DP[i - 1] + DP[i - 2]
    return DP[n]
```

NB. Sebbene finora avessimo parlato soltanto di soluzioni ricorsive, nulla ci vieta di realizzarne di iterative se queste sono estensionalmente equivalenti.

L'aumento in efficienza è evidente in quanto la *complessità* si è ridotta a $\Theta(n)$, tuttavia, l'utilizzo di una *tabella delle soluzioni* ci porta a considerare anche la *complessità* dal punto di vista spaziale, cioè quanto spazio di memoria è necessario per la sua memorizzazione. In questo caso memorizziamo i valori in un vettore di dimensione $n+1$, quindi la *complessità* è $S(n) = \Theta(n)$.

Possiamo fare meglio di così?

Da un punto di vista temporale no, ma da quello spaziale si. Se consideriamo bene la funzione ci rendiamo facilmente conto del fatto che, ad ogni iterazione, utilizziamo soltanto i valori delle due posizioni precedenti. Di conseguenza, potremmo eliminare il vettore e utilizzare delle semplici variabili.

Frammento 85 - Terza implementazione della soluzione.

```
int domino3(int n)
    int DP_0 = 1
    int DP_1 = 1
    int DP_2 = 1
    for (i = 2 to n) do
        DP_0 = DP_1
        DP_1 = DP_2
        DP_2 = DP_1 + DP_0
    return DP_2
```

Ora, la *complessità spaziale* si è ridotta a $\Theta(1)$.

Criterio di costo logaritmico Le *complessità* calcolate finora sono tali fino a quando ci basiamo su un *criterio di costo uniforme*, nel quale consideriamo soltanto il numero di elementi. Ora però, proviamo a chiederci come cambierebbero le *complessità* se utilizzassimo un *criterio di costo logaritmico*.

Iniziamo ragionando sul modo in cui crescono i numeri della serie di Fibonacci. Se $F(n)$ è l'n-esimo numero della serie, vale la *formula di Binet*:

$$F(n) = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}} \quad \text{con} \quad \phi = \frac{1+\sqrt{5}}{2} = 1.6180339887\ldots$$ [1]

NB. Vale la pena far notare che $F(n) = DP[n-1]$ perché DP vale per $n \geq 0$, mentre $F(n)$ per $n \geq 1$.

Il primo termine della formula è un'esponenziale con base un numero maggiore di 1, quindi cresce esponenzialmente, mentre il secondo tende a 0. Questo ci dice che i bit necessari a memorizzare l'n-esimo valore della serie crescono linearmente. Ovvero, per memorizzare l'n-esimo termine servono $\Theta(n)$ bit e sommare due valori consecutivi costa $\Theta(n)$.

Detto questo, possiamo ricalcolare tutte le *complessità* delle tre soluzioni sotto il *criterio di costo logaritmico*.

Soluzione	Complessità temporale	Complessità spaziale
`domino1`	$O(n2^n)$	$O(n^2)$
`domino2`	$O(n^2)$	$O(n^2)$
`domino3`	$O(n^2)$	$O(n)$

NB. In pratica, abbiamo semplicemente moltiplicato per n tutti i valori.

12.3 Problema di Hateville

Problema 6 - Hateville.

Hateville è un villaggio particolare composto da n case numerate e disposte linearmente lungo una singola strada. Ad Hateville ognuno odia i propri vicini: l'abitante i odia i suoi vicini $i-1$ e $i+1$ (se esistono).

Si vuole realizzare una sagra per la quale è necessario raccogliere dei fondi. Ogni abitante i è disposto a donare una quantità $D[i]$, ma solo se nessuno dei suoi vicini partecipa.

Scrivere un algoritmo che restituisca la quantità massima di fondi che può essere raccolta e un secondo algoritmo che restituisca il sottoinsieme di indici $S \subseteq \{1,\ldots,n\}$ tale per cui la donazione totale $T = \sum_{i \in S} D[i]$ sia massimale.

Ad esempio, se il vettore delle donazioni è $D = [10, 5, 5, 10]$ la massima quantità di fondi che può essere raccolta è 20 e l'insieme S è $\{1, 4\}$.

12.3.1 Approccio basato su divide-et-impera

Iniziamo provando a ridefinire il problema. Se $HV(i)$ è uno dei possibili insiemi di indici che consente di ottenere una raccolta massimale dalle prime i case, la soluzione al problema è $HV(n)$.

Consideriamo ora il vicino i. Possiamo accettare o meno la sua donazione. Se non la accettiamo possiamo accettare quella del vicino $i-1$ e quindi $HV(i) = HV(i-1)$. Se invece accettiamo la donazione di i, dobbiamo rifiutare quella di $i-1$, perciò $HV(i) = HV(i-2) \cup \{i\}$.

[1] ϕ rappresenta il valore della *sezione aurea* ed è anche definito come $\lim_{n\to+\infty} \frac{F(n+1)}{F(n)}$, ovvero come il rapporto tra numeri consecutivi della serie di Fibonacci

Per decidere se accettare o meno la donazione di i è sufficiente vedere quale scelta permette di raccogliere più fondi. Se *highest* è una funzione che, dati due insiemi, seleziona quello che garantisce una donazione più alta, $HV(i)$ diventa:

$$HV(i) = highest\left(HV(i-1), HV(i-2) \cup \{i\}\right)$$

Ciò che abbiamo detto finora è vero secondo quanto affermato dal seguente teorema:

Definizione 111 - Sottostruttura ottima.

Siano P_i il problema dato dalle prime i case e S_i una sua soluzione ottima. Valgono le seguenti:

1. *Se $i \notin S_i$, allora $S_i = S_{i-1}$;*
2. *Se $i \in S_i$, allora $S_i = S_{i-2} \cup \{i\}$;*

Dimostrazione. Procediamo dimostrando separatamente i due punti.

Caso 1: $\boldsymbol{i \notin S_i}$ Secondo il teorema, $S_i = S_{i-1}$, cioè S_i è la soluzione ottima sia di P_i che di P_{i-1}. Se supponessimo, per assurdo, che non fosse così, dovrebbe esistere una soluzione S'_{i-1} per il problema P_{i-1} tale che $S'_{i-1} = highest\left(S'_{i-1}, S_i\right)$. Cioè, la soluzione S'_{i-1} dovrebbe garantire una donazione maggiore di quella di S_i. Ma, se fosse così, S'_{i-1} dovrebbe essere anche la soluzione di P_i, generando un assurdo.

Caso 2: $\boldsymbol{i \in S_i}$ Se $i \in S_i$, $i - 1 \notin S_i$, altrimenti non sarebbe una soluzione ammissibile. Quindi, $S_i - \{i\}$ deve essere una soluzione ottima per P_{i-2}. Se supponessimo, per assurdo, che non fosse così, dovrebbe esistere una soluzione S'_{i-2} per il problema P_{i-2} tale che $S'_{i-2} = highest\left(S'_{i-2}, S_i - \{i\}\right)$. Ma, se fosse così, $S'_{i-2} \cup \{i\}$ dovrebbe essere una soluzione di P_i migliore di S_i, generando un assurdo. □

Dimostrate le nostre ipotesi, possiamo procedere con la definizione ricorsiva della soluzione. I casi base sono due:

- Se $i = 0$, $HV(0) = \emptyset$;
- Se $i = 1$, $HV(1) = \{1\}$;

La funzione che otteniamo è la seguente:

$$HV(i) = \begin{cases} \emptyset & i = 0 \\ \{1\} & i = 1 \\ highest\left(HV(i-1), HV(i-2) \cup \{i\}\right) & i \geq 2 \end{cases}$$

Osservando la definizione possiamo notare la similitudine con la funzione per il calcolo della *sequenza di Fibonacci*. Essendoci sotto-problemi ripetuti, possiamo scartare subito qualunque soluzione basata su *divide-et-impera*.

12.3.2 Approccio basato su programmazione dinamica

La *programmazione dinamica* ci richiede di costruire una *tabella delle soluzioni*. Proviamo a costruire tale tabella partendo da un esempio: sia il vettore delle donazioni D definito come segue:

$$D = [10, 5, 5, 8, 4, 7, 12]$$

La *tabella delle soluzioni* è la seguente:

i	0	1	2	3	4	5	6	7
HV	$\emptyset$	$\{1\}$	$\{1\}$	$\{1,3\}$	$\{1,4\}$	$\{1,3,5\}$	$\{1,4,6\}$	$\{1,3,5,7\}$

Ci sono un paio di problemi. Prima di tutto dobbiamo definire la funzione *highest*, ma il problema più grosso è la difficoltà di memorizzazione degli insiemi in una tabella.

Per risolvere entrambi i problemi in una volta potremmo definire il valore delle soluzioni come la quantità di fondi che vengono raccolti. Definiamo quindi una funzione DP tale che $DP[i]$ è la massima quantità di fondi che si possono ottenere dalle prime i case di Hateville. Partendo da HV possiamo definire DP come segue:

$$DP[i] = \begin{cases} 0 & i = 0 \\ D[1] & i = 1 \\ \max\left(DP[i-1], DP[i-2] + D[i]\right) & i \geq 2 \end{cases}$$

Adesso implementare la soluzione è una banalità.

Frammento 86 - Implementazione della soluzione al primo problema.

```
int hateville(int[] D, int n)
   int[] DP = new int[0...n]                    % Tabella delle soluzioni
   DP[0] = 0
   DP[1] = D[1]
   for (i = 2 to n) do
      DP[i] = max(DP[i - 1], DP[i - 2] + D[i])
   return DP[n]
```

La *complessità temporale* e *spaziale* della soluzione sono entrambe $\Theta(n)$. Possiamo notare che, come nel problema del domino, per ogni i la soluzione è ottenuta a partire dalle soluzioni dei due valori precedenti di i, quindi, potremmo eliminare l'array e introdurre delle variabili semplici riducendo a $\Theta(1)$ la *complessità spaziale*.

Frammento 87 - Implementazione migliorata della soluzione al primo problema.

```
int hateville(int[] D, int n)
   int DP_0 = 0
   int DP_1 = 1
   int DP_2 = 1
   for (i = 2 to n) do
      DP_2 = max(DP_1, DP_0 + D[i])
   return DP_2
```

Ricostruire la soluzione A questo punto abbiamo risolto la prima richiesta del problema. Per quanto riguarda la seconda, dobbiamo riuscire a ricostruire l'insieme di indici a partire dalla *tabella delle soluzioni*. Grazie al teorema sulla *Algoritmo euristico* possiamo fare le seguenti assunzioni:

- Se la casa i non è stata selezionata, $i \notin S_i$ e quindi $S_i = S_{i-1}$. Questo, riflesso sulla *tabella delle soluzioni*, garantisce che se $DP[i] = DP[i-1]$, i non appartiene all'insieme S di indici;

- Se la casa i è stata selezionata, $i \in S_i$ e quindi $S_i = S_{i-2} \cup \{i\}$. Questo, riflesso sulla *tabella delle soluzioni*, garantisce che se $DP[i] = DP[i-2] + D[i]$, i appartiene all'insieme S di indici;

NB. È possibile che risultino verificati entrambi i casi e, se succede, significa che esistono due soluzioni possibili.

Chiariti questi punti, possiamo ricostruire la soluzione fino a i in modo ricorsivo. Nello specifico, se $DP[i] = DP[i-1]$, si prende la soluzione valida per $i-1$, altrimenti si prende quella valida per $i-2$ e le si aggiunge i.

Frammento 88 - Implementazione della soluzione al secondo problema.

```
SET hateville(int[] D, int n)
   int[] DP = new int[0...n]                          % Tabella delle soluzioni
   DP[0] = 0
   DP[1] = D[1]
   for (i = 2 to n) do
      DP[i] = max(DP[i - 1], DP[i - 2] + D[i])
   return solution(DP, n)

% Funzione ausiliaria per la ricostruzione della soluzione
SET solution(int[] DP, int i)
   if (i == 0) then
      return ∅
   if (i == 1) then
      return {1}
   if (DP[i] == DP[i - 1]) then
      return solution(DP, i - 1)
   else
      SET sol = solution(DP, i - 2)
      sol.insert(i)
      return sol
```

La *complessità temporale* di `solution` è $\Theta(n)$.

NB. È importante notare che non avremmo potuto ricostruire la soluzione senza l'intera *tabella delle soluzioni* e quindi non possiamo utilizzare la versione migliorata di `hateville`.

12.4 Problema dello zaino

Problema 7 - Problema dello zaino[2].

Dato un insieme di oggetti, ognuno caratterizzato da un peso e da un profitto, e uno "zaino" di capacità finita, individuare un sottoinsieme di oggetti tali per cui il loro peso totale sia al più uguale alla capacità dello zaino e il loro profitto totale sia massimale.

Questo problema ci chiede di realizzare un algoritmo che prenda in input un vettore w dei pesi, un vettore p dei profitti e la capacità C dello zaino e che restituisca un insieme $S \subseteq \{1, \ldots, n\}$ tale che:

[2]Questo è uno dei problemi classici dell'informatica ed è noto con il nome di *Knapsack problem*

- il peso totale non superi la capacità, cioè $w(S) = \sum_{i \in S} w[i] \leq C$;
- il profitto totale sia massimale, cioè $argmax_S\ p[S] = \sum_{i \in S} p[i]$;

Ad esempio, se $C = 12$, $w = [10, 4, 8]$ e $p = [20, 6, 12]$, l'algoritmo deve restituire l'insieme $S = \{1\}$.

Proviamo a dare un valore alla soluzione. Definiamo una funzione $DP[i][c]$ come il massimo profitto che può essere ottenuto dai primi $0 \leq i \leq n$ oggetti in uno zaino di capacità $c \leq C$. Il massimo profitto ottenibile nel problema originale è allora $DP[n][C]$.

Ora, consideriamo un generico oggetto i. Se non lo prendiamo, $DP[i][c] = DP[i-1][c]$ perché non cambiano né il profitto né la capacità. Se invece lo prendiamo, $DP[i][c] = DP[i-1][c - w[i]] + p[i]$ in quanto, il peso dell'oggetto i viene sottratto alla capacità rimanente e il profitto associato viene aggiunto a quello ottenuto con i precedenti $i - 1$ oggetti.

La scelta migliore tra le due è quella che massimizza il profitto e la possiamo sintetizzare con la seguente:

$$DP[i][c] = \max\left(\overbrace{DP[i-1][c-w[i]] + p[i]}^{\text{Preso}}, \overbrace{DP[i-1][c]}^{\text{Non preso}}\right)$$

Per quanto riguarda i casi base, è facile dedurre che per $i = 0$ o $c = 0$, il profitto massimo possibile sia 0. C'è però un terzo caso: quando proviamo a prendere un oggetto, lo facciamo senza verificare che il peso non superi la capacità, quindi, potremmo ritrovarci ad avere un valore negativo per quest'ultima. Se questo succede, è ovvio che non avremmo potuto raccogliere quell'oggetto e, di conseguenza, decidiamo che in quel caso il profitto valga $-\infty$ in modo da assicurare che non venga selezionato dalla funzione max.

Detto questo, possiamo scrivere la formula nella sua forma completa:

$$DP[i][c] = \begin{cases} 0 & i = 0 \vee c = 0 \\ -\infty & c < 0 \\ \max\left(DP[i-1][c-w[i]] + p[i],\ DP[i-1][c]\right) & \text{altrimenti} \end{cases}$$

Equivalentemente, possiamo rimuovere il $-\infty$ e riscrivere la formula come segue:

$$DP[i][c] = \begin{cases} 0 & i = 0 \vee c = 0 \\ DP[i-1][c] & w[i] > c \\ \max\left(DP[i-1][c-w[i]] + p[i],\ DP[i-1][c]\right) & w[i] \leq c \end{cases}$$

12.4.1 Approccio basato su programmazione dinamica

Frammento 89 - Implementazione iterativa della soluzione.

```
int knapsack(int[] w, int[] p, int n, int C)
   int[][] DP = new int[0...n][0...C]                % Tabella delle soluzioni
   for (i = 0 to n) do
      DP[i][0] = 0
   for (c = 0 to C) do
      DP[0][c] = 0
   for (i = 1 to n) do
      for (c = 1 to C) do
         if (w[i] ≤ c) then
            DP[i][c] = max(DP[i - 1][c - w[i]] + p[i], DP[i - 1][c])
```

```
        else
            DP[i][c] = DP[i - 1][c]
    return DP[n][C]
```

Esempio 29 - Esempio di esecuzione.

Dato uno zaino di capacità $C = 9$ e i seguenti vettori dei pesi e dei profitti:

$$w = [4, 2, 3, 4]$$

$$p = [10, 7, 8, 6]$$

al termine dell'esecuzione dell'algoritmo di cui sopra, la matrice DP corrisponde alla seguente tabella:

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10
2	0	0	7	7	10	10	17	17	17	17
3	0	0	7	8	10	15	17	18	18	25
4	0	0	7	8	10	15	17	18	18	25

Complessità Siccome l'algoritmo costruisce e popola una tabella di dimensione $n \cdot C$, la *complessità* è $T(n) = \Theta(nC)$. Ovviamente lo stesso vale anche per la *complessità spaziale*. Quindi, il costo dell'algoritmo dipende da due variabili, ma poiché C non rappresenta la dimensione dell'input e viene rappresentato mediante $k = \lceil \log C \rceil$ bit, la *complessità temporale* può essere riscritta come:

$$T(n) = \Theta(n2^k)$$

Ciò, rende questa implementazione dell'algoritmo *pseudo-polinomiale*[3].

È possibile fare meglio di così?

Possiamo provare a realizzare una versione ricorsiva dell'algoritmo.

Frammento 90 - Implementazione ricorsiva della soluzione.

```
int knapsack(int[] w, int[] p, int n, C)
    return knapsackRec(w, p, n, C)

int knapsackRec(int[] w, int[] p, int n, c)
    if (c ≤ 0) then
        return -∞
    if (i == 0 or c == 0) then
        return 0
    int notTaken = knapsackRec(w, p, i - 1, c)
    int taken = knapsackRec(w, p, i - 1, c - w[i]) + p[i]
    return max(notTaken, taken)
```

[3] Parleremo meglio di cosa significhi più avanti nella trattazione

In questa implementazione abbiamo semplicemente tradotto in codice la prima forma della formula per DP e l'*equazione di ricorrenza* è:

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

Per il *Teorema delle ricorrenze lineari di ordine costante* la *forma chiusa* associata è $T(n) = \Theta(2^n)$ e quindi la *complessità* è addirittura più alta di quella della soluzione iterativa.

Purtroppo infatti, secondo l'opinione della maggior parte degli informatici, non esistono soluzioni meno complesse di quelle che abbiamo esaminato. Ciò che possiamo comunque provare a fare, è realizzare delle implementazioni che, a parità di *complessità*, migliorino quelle già esistenti.

12.4.2 Approccio basato su memoization

Come già accennato, la *memoization* è una tecnica risolutiva che unisce l'utilizzo di una *tabella delle soluzioni* all'approccio top-down tipico del *divide-et-impera* classico.

In particolare, la *tabella delle soluzioni* viene inizializzata ad un valore arbitrario che usiamo per indicare che un certo sotto-problema non è ancora stato risolto. Quindi, quando è necessario risolvere un sotto-problema, se la tabella contiene già la soluzione, la si usa direttamente, altrimenti la si calcola. In questo modo, ogni sotto-problema viene risolto una sola volta.

Nel problema che stiamo affrontando, usiamo -1 come valore speciale e modifichiamo la soluzione ricorsiva in modo da riutilizzare le soluzioni.

Frammento 91 - Implementazione basata su memoization.

```
int knapsack(int[] w, int[] p, int n, int C)
   int[][] DP = new int[1...n][1...C]                % Tabella delle soluzioni
   for (i = 1 to n) do
      for (c = 1 to n) do
         DP[i][c] = -1
   return knapsackRec(w, p, n, C, DP)

int knapsackRec(int[] w, int[] p, int i, int c, int[][] DP)
   if (c < 0 ) then
      return -∞
   if (i == 0 or c == 0) then return 0
   if (DP[i][c] < 0) then
      int notTaken = knapsackRec(w, p, i - 1, c, DP)
      int taken = knapsackRec(w, p, i - 1, c - w[i], DP) + p[i]
      DP[i][c] = max(notTaken, taken)
   return DP[i][c]
```

Complessità La *complessità* rimane $T(n) = \Theta(nC)$ perché è il costo che paghiamo per inizializzare DP. Tuttavia, siamo comunque riusciti a ridurre le chiamate ricorsive che vengono generate. Addirittura, sostituendo la tabella con un *dizionario* implementato come *hash table* ci liberiamo anche della necessità fare l'inizializzazione e quindi il costo diventa $T(n) = \min(2^n, nC)$.

A conferma di quanto affermato circa la riduzione delle chiamate ricorsive, osserviamo un esempio di esecuzione sugli stessi dati dell'esempio precedente.

Esempio 30 - Esempio di esecuzione.

Dato uno zaino di capacità $C = 9$ e i seguenti vettori dei pesi e dei profitti:

$$w = [4, 2, 3, 4]$$

$$p = [10, 7, 8, 6]$$

al termine dell'esecuzione dell'algoritmo di cui sopra, la matrice DP corrisponde alla seguente tabella:

	c									
i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0										
1		-1	0	0	10	10	10	10	-1	10
2		-1	7	-1	-1	10	17	-1	-1	17
3		-1	-1	-1	-1	15	-1	-1	-1	25
4		-1	-1	-1	-1	-1	-1	-1	-1	25

Le chiamate ricorsive sono state soltanto 14, invece delle 36 che avevamo nella prima soluzione.

Sebbene il problema originale non lo richieda, può essere interessante provare a implementare l'algoritmo per la ricostruzione della soluzione a partire da DP.

Il principio è molto semplice. Considerando $DP[i][c]$, se $DP[i][c] \neq DP[i-1][c]$, l'elemento i è stato raccolto, quindi l'algoritmo continua ricorsivamente da $DP[i-1][c-w[i]]$. Se invece l'elemento i non è stato raccolto, l'algoritmo continua da $DP[i-1][c]$.

Frammento 92 - Implementazione algoritmo per la ricostruzione della soluzione.

```
SET solution(int[] w, int i, int c, int[][] DP)
   if (i == 0 or c == 0) then                    % Non si può raccogliere nulla
      return ∅
   if (i == 1 and DP[i][c] > 0) then
      return {i}
   if (DP[i][c] == DP[i-1][c]) then          % L'elemento i non è stato raccolto
      return solution(w, i - 1, c, DP)
   SET S = solution(w, i - 1, c - w[i], DP)
   S.insert(i)
   return S
```

NB. Il controllo `i == 1` è necessario solo se per popolare DP è stato usato l'algoritmo realizzato mediante *memoization* perché, in quel caso, $DP[0][c]$ non esiste.

Per ricostruire l'intera soluzione è sufficiente invocare `solution(w, n, C, DP)` dove DP è stato precedentemente popolato.

12.5 Problema dello zaino senza limiti

Vediamo ora una variante al problema appena affrontato per studiarne possibili semplificazioni.

Problema 8 - Problema dello zaino senza limiti.

Dato un insieme di oggetti, ognuno caratterizzato da un peso e da un profitto, e uno "zaino" di capacità finita, individuare un sottoinsieme di oggetti tali per cui il loro peso totale sia al più uguale alla capacità dello zaino e il loro profitto totale sia massimale, ma senza porre limiti al numero di volte che un oggetto può essere selezionato.

La funzione che descrive il valore della soluzione può essere ottenuta modificando quella usata per il problema originale. In particolare, quando un oggetto viene selezionato, non è più necessario decrementare il valore dell'indice i, perché lo stesso potrebbe essere raccolto di nuovo. Applicando la modifica, otteniamo la seguente:

$$DP[i][c] = \begin{cases} 0 & i = 0 \vee c = 0 \\ -\infty & c < 0 \\ \max\left(DP[i][c - w[i]] + p[i],\ DP[i-1][c]\right) & \text{altrimenti} \end{cases}$$

Possiamo semplificare di più?

La risposta è si, infatti, se un oggetto può essere raccolto più volte, non è più necessario mantenere il valore i nella *tabella delle soluzioni*. Questo ci consente di ridefinire il valore della soluzione.

Nello specifico, dato uno zaino senza limiti di scelta, di capacità C e n oggetti caratterizzati da un peso w e un profitto p, definiamo $DP[c]$ il massimo profitto che può essere ottenuto da tali oggetti in uno zaino di capacità $c \leq C$.

Questa nuova descrizione ci porta a scrivere quanto segue:

$$DP[c] = \begin{cases} 0 & c = 0 \\ \max_{w[i] \leq c}\{DP[c - w[i]] + p[i]\} & c > 0 \end{cases}$$

Per la traduzione in codice utilizziamo la tecnica della *memoization*.

Frammento 93 - Implementazione della soluzione.

```
int knapsack(int[] w, int[] p, int n, int C)
   int[] DP = new int[0...C]                          % Tabella delle soluzioni
   for (c = 0 to C) do
      DP[c] = -1
   return knapsackRec(w, p, n, C, DP)

int knapsackRec(int[] w, int[] p, int n, int c, int[] DP)
   if (c == 0) then
      return 0
   if (DP[c] < 0) then        % Se è falso, la soluzione è già stata calcolata
      int maxSoFar = 0
      for (i = 1 to n) do
         if (w[i] ≤ c) then
            int value = knapsackRec(w, p, n, c - w[i], DP) + p[i]
            maxSoFar = max(maxSoFar, value)
      DP[c] = maxSoFar
   return DP[c]
```

Complessità Nel caso pessimo vengono riempite tutte le celle della tabella DP e, poiché per riempire una cella devono essere provati tutti gli n elementi, per ognuna si paga $\Theta(n)$. La *complessità temporale* totale è $T(n) = O(nC)$, quella *spaziale* è invece $S(n) = \Theta(C)$.

Rispetto al problema precedente abbiamo ridotto lo spazio utilizzato, ma la ricostruzione della soluzione a partire da DP è più difficile perché per capire da quale elemento deriva il profitto massimo dovremmo ispezionarli tutti. Per tanto, conviene memorizzare tale informazione nel momento stesso in cui costruiamo DP.

Frammento 94 - Implementazione con ricostruzione della soluzione.

```
LIST knapsack(int[] w, int[] p, int n, int C)
   int[] DP = new int[0...C]                          % Tabella delle soluzioni
   int[] pos = new int[0...C]   % Elemento raccolto per ogni valore di capacità
   for (c = 0 to C) do
      DP[c] = -1
      pos[c] = -1
   knapsackRec(w, p, n, C, DP, pos)
   return solution(w, C, pos)

int knapsackRec(int[] w, int[] p, int n, int c, int[] DP, int[] pos)
   if (c == 0) then
      return 0
   if (DP[c] < 0) then        % Se è falso, la soluzione è già stata calcolata
      DP[c] = 0
      for (i = 1 to n) do
         if (w[i] ≤ c) then
            int value = knapsackRec(w, p, n, c - w[i], DP, pos) + p[i]
            if (value > DP[c]) then
               DP[c] = value
               pos[c] = i
   return DP[c]

LIST solution(int[] w, int c, int[] pos)
   if (c == 0 or pos[c] < 0) then
      return List()
   LIST L = solution(w, c - w[pos[c]], pos)  % Lista degli indici selezionati
   L.insert(L.tail(), pos[c])
   return L
```

12.6 Ricerca della sottosequenza comune massimale

Definizione 112 - Sottosequenza.

Una sequenza P è una sottosequenza di T se P è ottenuto da T rimuovendo uno o più dei suoi elementi.

Alternativamente, possiamo definire P anche come l'insieme degli indici $\{1, \ldots, n\}$ degli elementi di T che compaiono anche in P. I rimanenti elementi sono poi elencati nello stesso ordine, senza essere necessariamente contigui.

Ad'esempio, la stringa `P="AAATA"` è *sottosequenza* di `T="AAAATATGA"` perché in T compaiono almeno tre `A`, seguite da almeno una `T`, seguita a sua volta da almeno un'altra `A`.

NB. La sequenza vuota $P = \emptyset$ è *sottosequenza* di qualunque altra stringa.

Definizione 113 - Sottosequenza comune.

Una sequenza X è sottosequenza comune (Common Subsequence) di due sequenze T ed U se è una sottosequenza di entrambe e scriviamo in simboli:

$$X \in \mathcal{CS}(T, U)$$

Definizione 114 - Sottosequenza comune massimale.

Una sequenza $X \in \mathcal{CS}(T, U)$ è una sottosequenza comune massimale (Longest Common Subsequence) di due sequenze T ed U se non esiste una una sequenza comune $Y \in \mathcal{CS}(T, U)$ che sia più lunga di X ovvero tale per cui $|Y| > |X|$. Se è così, scriviamo in simboli:

$$X \in \mathcal{LCS}(T, U)$$

Date queste definizioni preliminari possiamo venire alla consegna del problema in questione:

Problema 9 - Ricerca della sottosequenza comune massimale.

Date due sequenze T ed U, trovare la più lunga sottosequenza comune di T ed U.

Ad esempio, se `T="AAAATTGA"` e `U="TAACGATA"`, l'algoritmo deve restituire `AAATA`.

Ovviamente la prima soluzione che potrebbe venirci in mente sarebbe quella di ricercare tutte le *sottosequenze* e sceglierne una tra le più lunghe. Questo approccio ci porterebbe però ad una funzione di *complessità* $T(n) = \Theta(2^n(n+m))$ perché ricercare tutte le *sottosequenze* di una stringa T lunga n costa $\Theta(2^n)$ e verificare se una sequenza è *sottosequenza* di un'altra costa $O(m+n)$.

12.6.1 Approccio basato su programmazione dinamica

Per fare meglio di così possiamo provare a sfruttare la *programmazione dinamica* e il meccanismo della ricorsione. In particolare, definiamo una funzione che ci restituisca il *prefisso* di una sequenza:

Definizione 115 - Prefisso.

Data una sequenza T composta da n caratteri $t_1, \ldots, t_n$, chiamiamo $T(i)$ la funzione che denota il prefisso di T dato dai primi i caratteri, ovvero tale che:

$$T(i) = t_1, \ldots, t_i$$

NB. Ovviamente vale l'identità $T(0) = \emptyset$.

A questo punto, date le due sequenze T ed U lunghe rispettivamente n ed m caratteri, vogliamo scrivere la definizione di una formula ricorsiva $\mathcal{LCS}(T(i), U(j))$ che restituisca la *sottosequenza comune massimale* dei prefissi $T(i)$ e $U(j)$.

Il caso base è banale, perché se $i = 0$ o $j = 0$, l'unica *sottosequenza comune massimale* è la *sottosequenza* vuota $\emptyset$.

Per quanto riguarda i casi ricorsivi possiamo distinguerne due:

Caso 1: $t_i = u_j$ In questo caso, gli ultimi caratteri dei *prefissi* $T(i)$, $U(j)$ coincidono, questo significa che il carattere $t_i = u_j$ è comune ad entrambe le sequenze e quindi farà parte della *sottosequenza comune massimale*. La ricerca di tale *sottosequenza* può quindi continuare dai *prefissi* $T(i-1)$, $U(j-1)$.

Caso 2: $t_i \neq u_j$ In questo caso, gli ultimi caratteri non coincidono e quindi la soluzione è provare a cercare la *sottosequenza comune massimale* sia nella coppia di *prefissi* $T(i-1)$, $U(j)$ che nella coppia $T(i)$, $U(j-1)$.

Tutto ciò che abbiamo detto si traduce nella seguente formulazione:

$$\mathcal{LCS}(T(i), U(j)) = \begin{cases} \emptyset & i = 0 \vee j = 0 \\ \mathcal{LCS}(T(i-1), U(j-1)) \oplus t_i & i > 0 \wedge j > 0 \wedge t_i = u_j \\ longest \begin{pmatrix} \mathcal{LCS}(T(i-1), U(j)), \\ \mathcal{LCS}(T(i), U(j-1)) \end{pmatrix} & i > 0 \wedge j > 0 \wedge t_i \neq u_j \end{cases}$$

NB. Il simbolo $\oplus$ è l'operatore di concatenazione.

La dimostrazione della correttezza della formula si basa sul *Teorema della sottostruttura ottima.*

Definizione 116 - Sottostruttura ottima.

Date due sequenze $T = (t_1, \ldots, t_n)$ e $U = (u_1, \ldots, u_m)$, se $X \in \mathcal{LCS}(T(n), U(m))$ valgono i seguenti tre casi:

1. *Se $t_n = u_m$, allora $x_k = t_n = u_m$ e $X(k-1) \in \mathcal{LCS}(T(n-1), (U(m-1)))$;*
2. *Se $t_n \neq u_m$ e $x_k \neq t_n$, allora $X \in \mathcal{LCS}(T(n-1), U(m))$;*
3. *Se $t_n \neq u_m$ e $x_k \neq u_m$, allora $X \in \mathcal{LCS}(T(n), U(m-1))$;*

Dimostrazione. Procediamo dimostrando separatamente i tre punti.

Caso 1: $t_n = u_m$ Se per assurdo supponessimo che $x_k \neq t_n = u_m$, allora esisterebbe una *sottosequenza* $Y = X \oplus t_n$. Se così fosse però, varrebbero $Y \in \mathcal{CS}(T(n), U(m))$ e $|Y| > |X|$ generando un assurdo.

Venendo alla seconda parte, se supponessimo per assurdo che $X(k-1) \notin \mathcal{LCS}(T(n-1), U(m-1))$, allora esisterebbe un $Y \in \mathcal{LCS}(T(n-1), U(m-1))$ tale che $|Y| > |X(k-1)|$. Quindi, $Y \oplus t_n \in \mathcal{CS}(T(n), U(m))$ e $|Y| > |X(k-1) \oplus t_n| = X$ generando un assurdo.

Caso 2: $t_n \neq u_m$ e $x_k \neq t_n$ Se ipotizzassimo per assurdo che $X \notin \mathcal{LCS}(T(n-1), U(m))$, allora esisterebbe un $Y \in \mathcal{LCS}(T(n-1), U(m))$ tale che $|Y| > |X|$. Conseguentemente, sarebbe anche vero affermare che $Y \in \mathcal{LCS}(T(n), U(m))$ e quindi anche che $X \notin \mathcal{LCS}(T(n), U(m))$, generando un assurdo.

Caso 3: $t_n \neq u_m$ e $x_k \neq u_m$ Se ipotizzassimo per assurdo che $X \notin \mathcal{LCS}(T(n), U(m-1))$, allora esisterebbe un $Y \in \mathcal{LCS}(T(n), U(m-1))$ tale che $|Y| > |X|$. Conseguentemente, sarebbe anche vero affermare che $Y \in \mathcal{LCS}(T(n), U(m))$ e quindi anche che $X \notin \mathcal{LCS}(T(n), U(m))$, generando un assurdo. □

Conclusa in questo modo la dimostrazione, possiamo passare alla scrittura della ricorrenza per il valore della soluzione. Trattandosi di sequenze di caratteri, il loro valore è dato dalla loro lunghezza. Vale quindi la seguente:

$$DP[i][j] = \begin{cases} 0 & i = 0 \vee j = 0 \\ DP[i-1][j-1] + 1 & i > j \wedge j > 0 \wedge t_i = u_j \\ \max(DP[i-1][j], DP[i][j-1]) & i > 0 \wedge j > 0 \wedge t_i \neq u_j \end{cases}$$

Chiaramente la soluzione al problema originale è $DP[n][m]$.

Frammento 95 - Implementazione della soluzione.

```
int lcs(ITEM[] T, ITEM[] U, int n, int m)
   int[][] DP = new int[0...n][0...m]                % Tabella delle soluzioni
   for (i = 0 to n) do
      DP[i][0] = 0
   for (j = 0 to m) do
      DP[0][j] = 0
   for (i = 1 to n) do
      for (j = 1 to m) do
         if (T[i] == U[j]) then
            DP[i][j] = DP[i - 1][j - 1] + 1
         else
            DP[i][j] = max(DP[i - 1][j], DP[i][j - 1])
   return DP[n][m]
```

Il problema non lo richiede, ma se volessimo ricostruire la soluzione, l'algoritmo sarebbe il seguente.

Frammento 96 - Ricostruzione della soluzione.

```
LIST solution(ITEM[] T, ITEM[] U, int i, int j, int[][] DP)
   if (i == 0 or j == 0) then
      return List()
   if (T[i] = U[i]) then
      LIST S = subsequence(T, U, i - 1, j - 1, DP)
      S.insert(S.tail(), T[i])
      return S
   if (DP[i - 1][j] > DP[i][j - 1]) then
      return subsequence(T, U, i - 1, j, DP)
   else
      return subsequence(T, U, i, j - 1, DP)
```

Complessità Per quanto riguarda la `subsequence`, siccome DP è una matrice $n \times m$ e per ricostruire la soluzione analizziamo la matrice partendo dalla posizione (n, m) e "salendo" verso $(0, 0)$, i possibili movimenti sono tre:

1. Da $DP[i][j]$ a $DP[i-1][j-1]$ se $T[i] = U[j]$;
2. Da $DP[i][j]$ a $DP[i-1][j]$ se $DP[i-1][j] > DP[i][j-1]$;

3. Da $DP[i][j]$ a $DP[i][j-1]$ se $DP[i-1][j] < DP[i][j-1]$;

Di conseguenza, nella situazione peggiore concludiamo l'esecuzione nella posizione $(0,0)$ arrivandoci non per la diagonale. Il costo in quella situazione è $O(n+m)$ perché vengono inseriti nella lista $n+m$ caratteri.

La *complessità* della `lcs` è $O(nm)$ perché dobbiamo riempire tutta la matrice DP che ha dimensione $n \times m$.

Se non siamo interessati a ricostruire la soluzione possiamo migliorare la *complessità spaziale* della `lcs` mantenendo in memoria soltanto due righe. Questo perché, quando calcoliamo il valore di $DP[i][j]$ utilizziamo $DP[i-1][j-1]$ oppure $DP[i-1][j]$ e $DP[i][j-1]$. Questi tre valori risiedono sempre su due sole righe, quindi le precedenti non sono necessarie.

Fatta questa riflessione, l'implementazione che ne risulta è la seguente:

Frammento 97 - Implementazione migliorata della soluzione.

```
int lcs(ITEM[] T, ITEM[] U, int n, int m)
   int[] DP = new int[0...m]                                  % Riga corrente
   int[] DP' = new int[0...m]                                 % Riga precedente
   for (j = 0 to m) do
      DP[j] = 0
   for (i = 1 to n) do
      DP ⇔ DP'                                                % Swap delle righe
      DP[0] = 0
      for (j = 1 to m) do
         if (T[i] == U[j]) then
            DP[j] = DP'[j - 1] + 1                  % Pari a DP[i-1][j-1] + 1
         else
            DP[j] = max(DP'[j], DP[j - 1]) % Pari a max(DP[i-1][j], DP[i][j-1])
   return DP[m]
```

In questo modo la *complessità spaziale* si riduce a $O(\min(n,m))$ perché la dimensione di DP e DP' può essere scelta in base al minore tra n e m.

NB. Questo tipo di algoritmi sono molto usati nello studio del DNA e nella ricerca delle similitudini tra diverse sequenze di DNA.

12.7 Problema dello string matching approssimato

Definizione 117 - Occerrenza *k*-approssimata.

Siano:

- $P = p_1, \ldots, p_m$ *una stringa detta pattern;*
- $T = t_1, \ldots, t_n$ *una stringa detta testo;*

Un'occorrenza k-approssimata di P in T è una copia di P in T in cui sono ammessi k "errori" tra P e T del seguente tipo:

1. Sostituzione*: I corrispondenti caratteri in P e T sono diversi;*
2. Inserimento*: un carattere di P non è incluso in T;*

3. Cancellazione*: un carattere di T non è incluso in P;*

Ad esempio se `T="questoèunoscempio"` e `P="unesempio"`, è possibile trovare un'*occorrenza approssimata* di P in T "correggendo due errori", ovvero cancellando la `c` di `scempio` e sostituendo la `o` di `uno` con una `e`.

Detto questo, possiamo introdurre il problema di questa sezione.

Problema 10 - String matching approssimato.

Date due stringhe P e T, trovare un'occorrenza $k-$approssimata di P in T tale che $0 \leq k \leq m$ sia minimo.

Questo problema ci chiede quindi di trovare il minimo valore di k per cui è possibile trovare un'*occorrenza approssimata* di P in T e, estendendolo, potremmo anche essere interessati a sapere dove si trovino gli errori e quali siano.

Come al solito, iniziamo definendo il valore della soluzione. Sia $DP[0 \dots m][0 \dots n]$ una tabella in cui $DP[i][j]$ contiene il valore minimo di k per cui esiste un'*occorrenza k-approssimata* di $P(i)$ in $T(j)$ che termina nella posizione j.

NB. La semantica delle funzioni $P(i)$ e $T(i)$ è la stessa vista nel problema della *sottosequenza comune massima*.

A questo punto, per ogni coppia di valori i e j diversi da zero, per $DP[i][j]$ valgono 4 possibili casistiche:

1. Se $P[i] = T[j]$ non c'è nessun errore e quindi $DP[i][j] = DP[i-1][j-1]$;
2. Se $P[i] \neq T[j]$ c'è un errore e le possibili correzioni sono tre:
   (a) *Sostituzione*: $DP[i][j] = DP[i-1][j-1] + 1$;
   (b) *Inserimento*: $DP[i][j] = DP[i-1][j] + 1$;
   (c) *Cancellazione*: $DP[i][j] = DP[i][j-1] + 1$;

Quindi, per ogni coppia (i, j), viene scelta la possibilità che restituisce il valore minore. Ciò è diretta conseguenza del fatto che anche se $P[i] = T[j]$ potrebbe essere più conveniente cercare il pattern nei caratteri precedenti del testo. La formulazione completa è dunque la seguente:

$$DP[i][j] = \begin{cases} 0 & i = 0 \\ i & j = 0 \\ \min \begin{pmatrix} DP[i-1][j-1] + \delta \\ DP[i-1][j] \\ DP[i][j-1] \end{pmatrix} & i > 0 \wedge j > 0 \wedge \delta = \begin{cases} 0 & P[i] = T[j] \\ 1 & P[i] \neq T[j] \end{cases} \end{cases}$$

Diversamente dagli esempi precedenti, la soluzione al problema non si trova necessariamente in $DP[m][n]$, in quanto, in quella posizione si trova il numero minimo di correzioni che devono essere apportate per ottenere un'*occorrenza k-approssimata* del pattern che termina nella posizione n del testo. Come abbiamo detto, il pattern potrebbe trovarsi più convenientemente cercandolo in altre porzioni del testo, quindi la soluzione è il minimo tra i valori $DP[m][j]$ con $0 \leq j \leq n$.

Frammento 98 - Implementazione della soluzione.

```
int stringMatching(ITEM[] P, ITEM[] T, int n, int m)
   int[][] DP =new int[0...m][0...n]                    % Tabella delle soluzioni
   for (j = 0 to n) do
      DP[0][j] = 0                                                    % Caso i = 0
   for (i = 0 to m) do
      DP[i][0] = i                                                    % Caso j = 0
   for (i = 1 to m) do
      for (j = 1 to n) do
         DP[i][j] = min(
            DP[i - 1][j - 1] + iif(P[i] == T[j], 0, 1),
            DP[i - 1][j] + 1,
            DP[i][j - 1] + 1
         )
   int pos = 0                                % Indice per cui DP[m][pos] è minimo
   for (j = 1 to n) do
      if (DP[m][j] < DP[m][pos]) then
         pos = j
   return DP[m][pos]
```

12.8 Problema del prodotto a catena di matrici

Problema 11 - Prodotto a catena di matrici.

Data una sequenza di matrici $A_1, \ldots, A_n$ a due a due compatibili per il prodotto matriciale, calcolarne il prodotto minimizzando il numero di prodotti scalari necessari.

NB. Il prodotto matriciale è associativo, ma non commutativo.

Ad esempio, date tre matrici A, B, C di dimensioni che sono rispettivamente 100×1, 1×100 e 100×1, il prodotto $A \cdot B \cdot C$ può essere calcolato come $(A \cdot B) \cdot C$ o $A \cdot (B \cdot C)$. Nel primo caso vengono calcolati $100 \cdot 1 \cdot 100 + 100 \cdot 100 \cdot 1 = 10000 + 10000 = 20000$ prodotto scalari. Nel secondo caso invece, i prodotti sono $1 \cdot 100 \cdot 1 + 100 \cdot 1 \cdot 1 = 100 + 100 = 200$.

Siamo riusciti a ridurre di un fattore 100 il numero di operazioni semplicemente cambiando la posizione delle parentesi, cioè modificando la *parentesizzazione*.

Definizione 118 - Parentesizzazione.

Una parentesizzazione $P_{i,j}$ del prodotto $A_i \cdot \ldots \cdot A_j$ consiste nella matrice A_i se $i = j$ e nel prodotto di due parentesizzazioni $(P_{i,k} \cdot P_{i+1,k})$ altrimenti.

Definizione 119 - Parentesizzazione ottima.

La parentesizzazione che minimizza il numero di prodotti scalari è detta essere ottima.

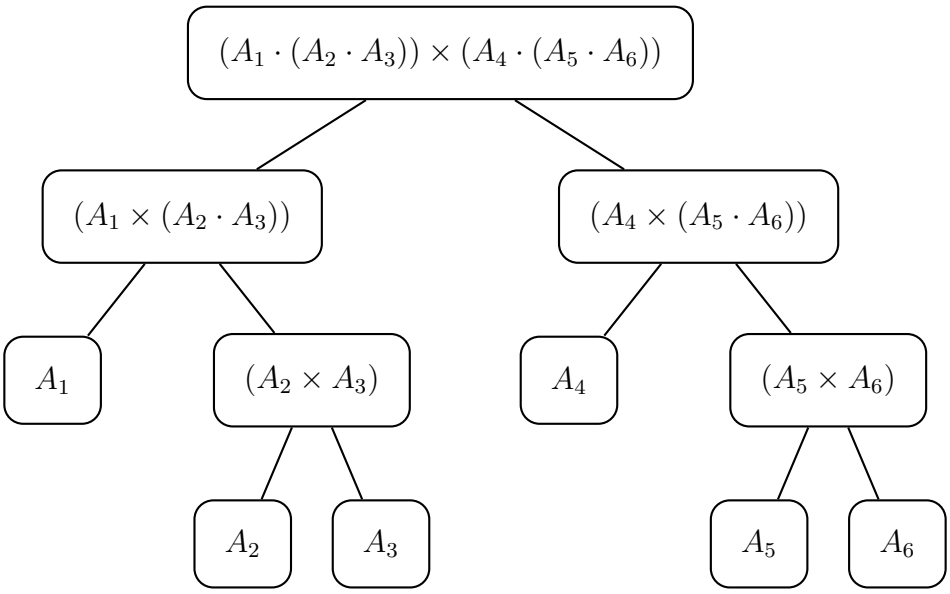
Esempio 31 - Possibile parentesizzazione.

Data la catena di prodotti:

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4 \cdot A_5 \cdot A_6$$

una possibile parentesizzazione per $k = 3$ è la seguente:

$$(A_1 \cdot (A_2 \cdot A_3)) \times (A_4 \cdot (A_5 \cdot A_6))$$



NB. Il simbolo $\times$ identifica il *prodotto finale* o *ultimo prodotto*.

Quante sono le *parentesizzazioni* possibili per una generica catena di n matrici?

Se $P(n)$ indica il numero di *parentesizzazioni* possibili per n matrici $A_1 \cdot \ldots \cdot A_n$, l'*ultimo prodotto* può occorrere in $n-1$ posizioni. Quindi, fissato l'indice k dell'*ultimo prodotto*, otteniamo $P(k)$ *parentesizzazioni* per $A_1 \cdot \ldots \cdot A_k$ e altre $P(n-k)$ *parentesizzazioni* per $A_{k+1} \cdot \ldots \cdot A_n$.

Ciò si traduce nella seguente formula:

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k) \cdot P(n-k) & n > 1 \end{cases}$$

Come esempio, per valori di n fino a 10, $P(n)$ vale:

$\boldsymbol{n}$	1	2	3	4	5	6	7	8	9	10
$\boldsymbol{P(n)}$	1	1	2	5	14	42	132	429	1430	4862

NB. I valori di $P(n)$ crescono seguendo la sequenza dei numeri di Catalan, che possono essere espressi anche con la seguente formulazione:

$$P(n) = C(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} = \Theta\left(\frac{4^n}{n\sqrt{n}}\right)$$

È possibile dimostrare che tale sequenza cresce come $\Omega(2^n)$ e quindi ne deduciamo che algoritmi a forza bruta non possono essere utilizzati.

Prima di procedere oltre, fissiamo la notazione utilizzata.

Notazione	Significato
$A_1 \cdot \ldots \cdot A_n$	Il prodotto di n matrici da ottimizzare
c_{i-1}	Il numero di righe della matrice A_i
c_i	Il numero di colonne della matrice A_i
$A[i \ldots j]$	Il sottoprodotto di $A_i \cdot \ldots \cdot A_j$
$P[i \ldots j]$	Una *parentesizzazione* per $A[i \ldots j]$

NB. Il numero di righe di una matrice coincide sempre con il numero di colonne della matrice che la precede nella catena e questo è dovuto alle regole di compatibilità del prodotto matriciale.

NB. $P[i \ldots j]$ non indica necessariamente la *parentesizzazione* migliore.

Possiamo osservare che data una sottosequenza $A[i \ldots j]$, se ne consideriamo una *parentesizzazione ottima*, esiste un *ultimo prodotto*, ovvero esiste un indice k tale che:

$$P[i \ldots j] = P[i \ldots k] \cdot P[k+1 \ldots j]$$

Quali sono le caratteristiche delle sue *sotto-parentesizzazioni*?

Definizione 120 - Teorema di sottostruttura ottima.

Se $P[i \ldots j] = P[i \ldots k] \cdot P[k+1 \ldots j]$ è una parentesizzazione ottima del prodotto $A[i \ldots j]$, allora $P[i \ldots k]$ e $P[k+1 \ldots j]$ sono rispettivamente le parentesizzazioni ottime dei prodotti $A[i \ldots k]$ e $A[k+1 \ldots j]$.

Dimostrazione. Procediamo con una dimostrazione per assurdo considerando separatamente le due *sotto-parentesizzazioni*:

- Se supponessimo esistesse una *parentesizzazione ottima* $P'[i \ldots k]$ di $A[i \ldots k]$ con costo inferiore a $P[i \ldots k]$, allora $P'[i \ldots k] \cdot P[k+1 \ldots j]$ sarebbe una *parentesizzazione ottima* di $A[i \ldots j]$ con costo inferiore a $P[i \ldots j]$, generando un assurdo;
- Se supponessimo esistesse una *parentesizzazione ottima* $P'[k+1 \ldots j]$ di $A[k+1 \ldots k]$ con costo inferiore a $P[k+1 \ldots j]$, allora $P[i \ldots k] \cdot P'[k+1 \ldots j]$ sarebbe una *parentesizzazione ottima* di $A[i \ldots j]$ con costo inferiore a $P[i \ldots j]$, generando un assurdo;

□

A questo punto, possiamo passare a ragionare sul valore della soluzione ottima. Definiamo una matrice DP in cui $DP[i][j]$ indica il numero minimo di prodotti scalari necessari per calcolare il prodotto $A[i \ldots j]$.

Sicuramente, se $i = j$, non è necessario eseguire alcun prodotto e quindi $DP[i][j] = 0$. Altrimenti, esiste una *parentesizzazione ottima* $P[i \ldots j] = P[i \ldots k] \cdot P[k+1 \ldots j]$ e, sfruttando la ricorsione, otteniamo che $DP[i][j] = DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j$. Il fattore $c_{i-1} \cdot c_k \cdot c_j$ è il costo per moltiplicare la matrice ottenuta dal prodotto $A_i \cdot \ldots \cdot A_k$ che ha c_{i-1} righe e c_k colonne e la matrice ottenuta dal prodotto $A_{k+1} \cdot \ldots \cdot A_j$ che ha c_k righe e c_j colonne.

Ma qual è il valore di k?

Non lo sappiamo, ma poiché k può assumere valori compresi tra i e $j-1$, possiamo provarli tutti e scegliere quello che comporta un numero minore di prodotti. La formula finale è quindi la seguente:

$$DP[i][j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{DP[i][k] + DP[k+1][j] + c_{i-1} \cdot c_k \cdot c_j\} & i < j \end{cases}$$

12.8.1 Approccio basato su divide-et-impera

Traducendo quella formula in codice otteniamo la seguente funzione:

Frammento 99 - Implementazione ricorsiva della soluzione.

```
int recPar(int[] c, int i, int j)
   if (i == j) then
      return 0
   int minSoFar = +∞
      for (k = i to j - 1) do
         int val = recPar(c, i, k) + recPar(c, k + 1, j)
         val = val + c[i - 1] · c[k] · c[j]
         if (val < minSoFar) then
            minSoFar = val
   return minSoFar
```

Il parametro c è un vettore con indici da 0 a n contenente le dimensioni di tutte le matrici. In particolare, $c[0]$ contiene il numero di righe della prima matrice, $c[i-1]$ il numero di righe della matrice $A[i]$ e $c[i]$ il numero di colonne della stessa.

Complessità Ad ogni livello vengono eseguite due chiamate, quindi la *complessità* è $T(n) = \Theta(2^n)$ che non è migliore di un algoritmo a forza bruta. Il problema è che molti sotto-problemi vengono risolti più di una volta e il loro numero è $\frac{n(n+1)}{2}$.

12.8.2 Approccio basato su programmazione dinamica

Introduciamo due matrici $n \times n$: DP e $last$. $DP[i][j]$ contiene il numero minimo di prodotti scalari necessari per moltiplicare le matrici $A[i \dots j]$, mentre $last[i][j]$ contiene il valore k dell'ultimo prodotto che minimizza il costo del sotto-problema.

Frammento 100 - Implementazione iterativa della soluzione.

```
int computePar(int[] c, int n)
   int[][] DP = new int[1...n][1...n]
   int[][] last = new int[1...n][1...n]
   for (i = 1 to n) do
      DP[i][i] = 0                          % Pone a 0 la diagonale principale
   for (h = 2 to n) do                          % h:  indice della diagonale
      for (i = 1 to n - h + 1) do                                  % i: riga
         int j = i + h + 1                                        % j: colonna
         DP[i][j] = +∞
         for (k = 1 to j - 1) do                  % k: indice ultimo prodotto
            int temp = DP[i][k] + DP[k + 1][j]
            temp = temp + c[i - 1] · c[k] · c[j]
            if (temp < DP[i][j]) then
               DP[i][j] = temp
               last[i][j] = k
   return DP[1][n]
```

Esempio 32 - Esempio di esecuzione.

Consideriamo il seguente vettore c:

$$c = [7, 8, 4, 2, 3, 5, 6]$$

Abbiamo quindi sei matrici di dimensioni:

$$A_1 = 7 \times 8, A_2 = 8 \times 4, A_3 = 4 \times 2, A_4 = 2 \times 3, A_5 = 3 \times 5, A_6 = 5 \times 6$$

Le tabelle DP e last vengono popolate procedendo diagonalmente a partire dalla diagonale principale e muovendo verso destra. Al termine dell'esecuzione valgono quindi:

DP	**1**	**2**	**3**	**4**	**5**	**6**
1	0	224	176	218	276	350
2		0	64	112	174	250
3			0	24	70	138
4				0	30	90
5					0	90
6						0

last	**1**	**2**	**3**	**4**	**5**	**6**
1	0	1	1	3	3	3
2		0	2	3	3	3
3			0	3	3	3
4				0	4	5
5					0	5
6						0

Il valore restituito dall'algoritmo è 350: il numero minimo di prodotti scalari che devono essere eseguiti per calcolare $A[1 \dots 6]$.

Complessità Il costo computazione di questa soluzione è $T(n) = \Theta(n^3)$ perché la matrice ha dimensione n^2 e ogni cella richiede $\Theta(n)$ per essere popolata.

12.9 Ricerca dell'insieme indipendente di peso massimo

Problema 12 - Ricerca dell'insieme indipendente di peso massimo.

Siano dati n intervalli distinti $[a_1, b_1[, \dots [a_n, b_n[$ della retta reale, aperti a destra e associati a un costo w_i per $1 \leq i \leq n$. Trovare un insieme indipendente di peso massimo, ovvero un sottoinsieme di intervalli, disgiunti tra loro, tale che la somma dei loro pesi sia massima.

Definizione 121 - Intervalli disgiunti.

Due intervalli i e j sono disgiunti se e solo se $b_j \leq a_i$ o $b_i \leq a_j$.

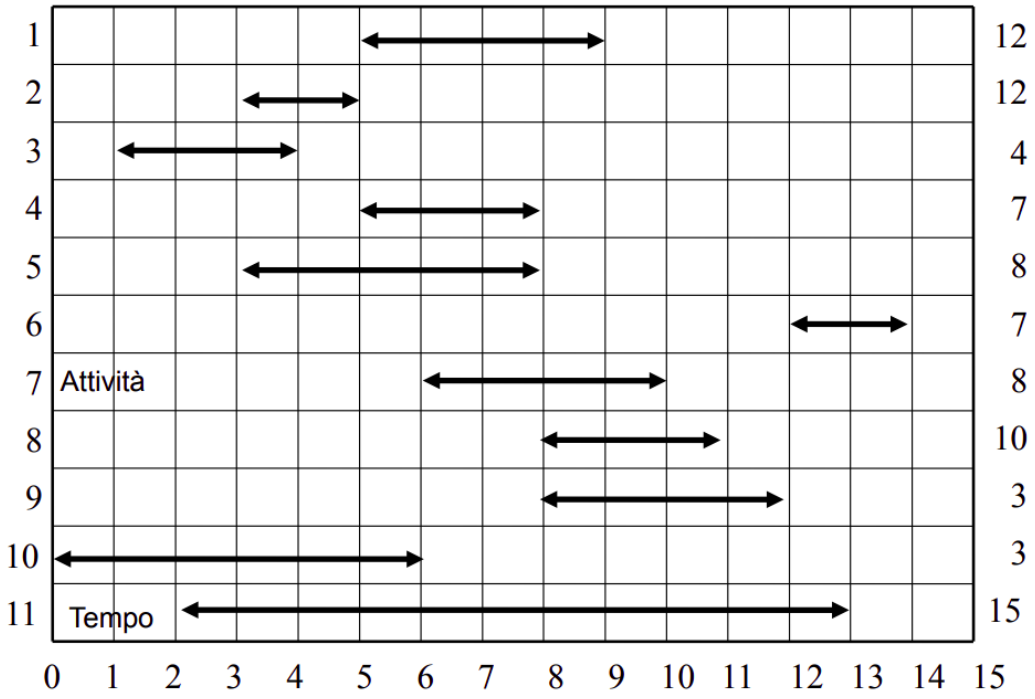
NB. Due intervalli sono *disgiunti* se non si intersecano.

Vediamo un esempio che possa aiutare a rendere più chiaro il problema.

Esempio 33 - Affitto di una sala conferenze.

Consideriamo un hotel che deve decidere a quali clienti concedere in affitto la propria sala conferenze. Ogni cliente richiede la sala per un certo numero di ore ed è disposto a pagare una determinata cifra. L'hotel vuole scegliere l'insieme di clienti che gli permetta di ottenere il guadagno maggiore.

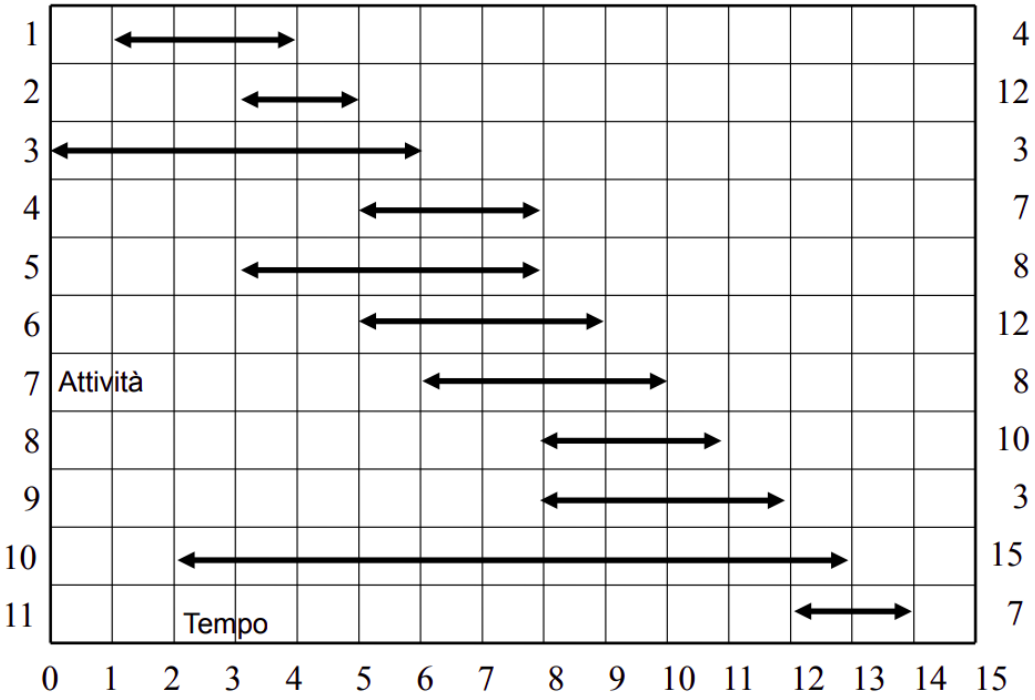
L'immagine di seguito schematizza l'insieme delle richieste ricevute dall'hotel e sul lato sinistro sono indicate le somme di denaro che i clienti hanno offerto.



Siccome non possono esserci sovrapposizioni, se l'hotel decidesse di concedere la sala al cliente 6, non potrebbe concederla al cliente 11 perché i due intervalli di tempo si intersecherebbero.

Il problema con questa disposizione è che preso un intervallo, per capire quali altri intervalli possono ancora essere considerati, siamo costretti ad analizzarli tutti.

Tuttavia, in una situazione come questa può essere conveniente eseguire una pre-elaborazione dei dati. In questo caso, potremmo ordinare gli intervalli per estremi di fine non decrescenti.



Ora per decidere se tenere o meno l'ultimo intervallo, possiamo precedere come abbiamo fatto negli esempi precedenti, cioè considerando una qualche tabella DP associata all'intervallo di tempo che si conclude nell'istante in cui inizia l'intervallo considerato.

Comunque, per rispondere al quesito, in questo caso l'hotel può massimizzare i profitti, concedendo la sala ai clienti: 2, 4, 8 e 11.

Chiarita la necessità di effettuare una pre-elaborazione, dobbiamo decidere in che modo trattare i dati. L'idea già citata è quella di ordinare gli intervalli per estremi di fine in modo non decrescente, cioè ordinarli in modo che $b_1 \leq \cdots \leq b_n$. A questo punto, definiamo una tabella DP tale che $DP[i]$ contenga il massimo profitto realizzabile con i primi i intervalli:

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max\left(DP[i-1], \max\left\{DP[j] + w_[i] : j < i \wedge b_j \leq a_i\right\}\right) & i > 0 \end{cases}$$

Con questa soluzione, cercare l'indice j costa $O(n)$ perché partendo dall'intervallo i è necessario scandire tutti gli altri. Poiché quest'operazione dev'essere ripetuto per ogni intervallo, il costo finale è $O(n^2)$.

È possibile trovare una pre-elaborazione migliore?

Potremmo provare a pre-calcolare il predecessore $pred_i = j$ di i tale che j sia il massimo valore minore di i per cui $b_j \leq a_i$. Se non esiste un tale valore, consideriamo $pred_i = 0$.

Noto il predecessore, la definizione di DP diventa:

$$DP[i] = \begin{cases} 0 & i = 0 \\ \max\left(DP[i-1], DP[pred_i] + w[i]\right) & i > 0 \end{cases}$$

NB. È possibile escludere un intervallo i, se sceglierne un altro j con stesso tempo di fine, ma tempo di inizio precedente, ha un valore $DP[j] > DP[i]$.

Un modo per calcolare i predecessori è il seguente:

Frammento 101 - Funzione per il calcolo dei predecessori.

```
int[] computePredecessors(int[] a, int[] b, int n)
   int[] pred = new int[0...n]
   pred[0] = 0
   for (i = 1 to n) do
      j = i - 1
      while (j > 0 and b[j] > a[i]) do
         j = j - 1
      pred[i] = j
   return pred
```

Il costo di questa funzione rimane $O(n^2)$, ma esistono delle implementazioni di *complessità* $O(n \log n)$.

Frammento 102 - Implementazione della soluzione.

```
SET maxInterval(int[] a, int[] b, int[] w, int n)
   % { Ordina gli intervalli per estremi di fine non decrescenti }
   int[] pred = computePredecessors(a, b, n)
   int[] DP = new int[0...n]
   DP[0] = 0
   for (i = 1 to n) do                    % Riempimento tabelle delle soluzioni
      DP[i] = max(DP[i - 1], DP[pred[i]] + w[i])
   int i = n
   SET S = Set()
```

```
   while (i > 0) do                                % Ricostruzione della soluzione
      if (DP[i - 1] > DP[pred[i]] + w[i]) do
         i = i - 1
      else
         S.insert(i)
         i = pred[i]
return S
```

Complessità La *complessità* dell'intero algoritmo è $O(n \log n)$ in quanto paghiamo $O(n \log n)$ per ordinare gli intervalli e calcolare i predecessori, mentre per popolare la tabella e ricostruire la soluzione paghiamo $\Theta(n)$.

Capitolo Nr.13

Scelta della struttura dati

Abbiamo già discusso più volte su come la scelta di una *struttura dati* influisca sulla *complessità* degli algoritmi, tuttavia, ora riprendiamo l'argomento perché input diversi allo stesso algoritmo potrebbero far risultare più efficiente l'uso di una *struttura dati* piuttosto che un'altra.

Quanto detto emerge chiaramente nel caso degli algoritmi per la ricerca dei *cammini minimi* all'interno di *grafi*. È un problema che abbiamo già affrontato in precedenza, ma qui ne vediamo due versioni più complesse: la ricerca dei *cammini minimi* verso tutti i *nodi*, da sorgente singola e da sorgente multipla.

13.1 Ricerca dei cammini minimi da sorgente singola

Problema 13 - Ricerca dei cammini minimi da sorgente singola.

Dati un grafo orientato $G = (V, E)$, una funzione di peso $w = E \to R$ e un nodo sorgente s, trovare un cammino da s a u, per ogni $u \in V$, il cui costo sia minimo, ovvero minore o uguale al costo di ogni altro cammino da s a u.

Definizione 122 - Costo di un cammino.

Dato un cammino $p = (v_1, \ldots, v_k)$ con $k > 1$, il costo del cammino è dato dalla seguente sommatoria:

$$w(p) = \sum_{i=2}^{k} w(v_{i-1}, w_i)$$

Per quanto riguarda i pesi, possono essere sia numeri interi che reali, ma anche positivi e negativi e alcuni algoritmi potrebbero non funzionare per alcune tipologie di pesi.

Consideriamo i due seguenti *grafi*:

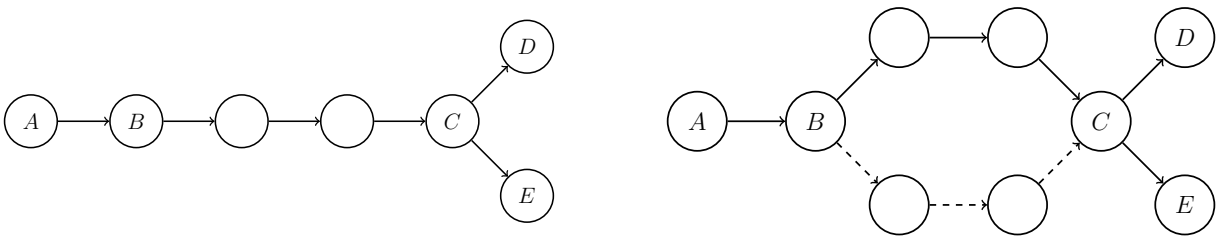


Fig. 13.1: Esempi di *cammini minimi*

Notiamo subito che *cammini minimi* verso *nodi* diversi potrebbero percorrere un tratto in comune, ma non potrebbero convergere su un *nodo* comune dopo aver percorso tratti diversi. Da ciò consegue che l'insieme dei *cammini minimi* da un *nodo* a tutti gli altri può essere rappresentato come un *albero di copertura*.

Definizione 123 - Albero di copertura.

Dato un grafo $G = (V, E)$ non orientato e connesso, un albero di copertura di G è un sottografo $T = (V, E_T)$ tale per cui T è un albero che contiene tutti i nodi di G ed $E_T \subseteq E$.

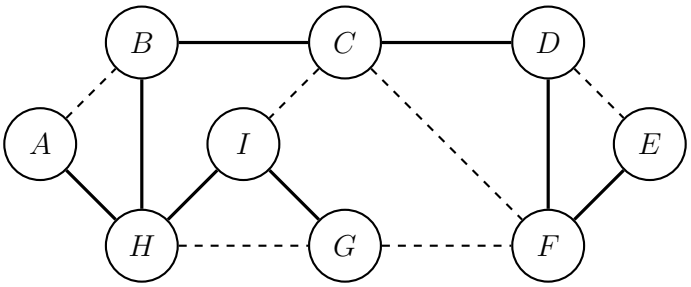
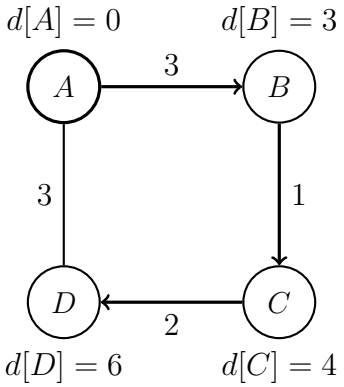


Fig. 13.2: Esempio di *albero di copertura*

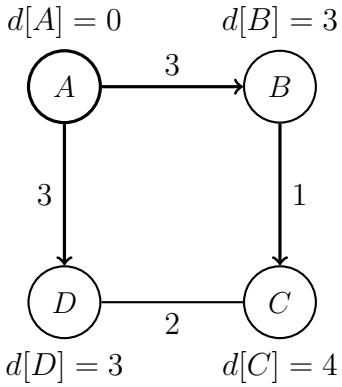
Detto questo, possiamo dire che tutte le soluzioni che non generano un *albero di copertura* non sono ammissibili come soluzioni al problema.

Definizione 124 - Soluzione ammissibile.

Una soluzione ammissibile può essere descritta da un albero di copertura T radicato in s e da un vettore delle distanze d, i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T.



(a) Soluzione non ammissibile



(b) Soluzione ammissibile

Fig. 13.3: Possibili *cammini* con sorgente nel *nodo A*

Nel primo caso la soluzione non è ammissibile perché la distanza del *nodo D* da *A* è 6, quando potrebbe essere 3.

Per rappresentare l'*albero di copertura* possiamo usare una versione modificata della `printPath` che avevamo usato per stampare il *cammino* tra due *nodi*.

Frammento 103 - printPath per la stampa dell'albero di copertura.

```
printPath(NODE s, NODE d, NODE[] T)
   if (s == d) then
      print s
   else if (T[d] == nil) then
      print "error"
   else
      printPath(s, T[d], T)
      print d
```

Siccome tra tutte le soluzioni possibili siamo interessati a quelle che includono solo *cammini minimi*, restringiamo il campo di ricerca alle *soluzioni ottime*, quelle cioè che rispettano il seguente teorema.

Definizione 125 - Teorema di Belman.

Dato un grafo $G = (V, E)$ e una soluzione $T = (V, E_T)$ in esso ammissibile, T è anche una soluzione ottima se e solo se:

$$\begin{aligned} d[v] &= d[u] + w(u, v) \quad \forall (u, v) \in E_T \\ d[v] &\leq d[u] + w(u, v) \quad \forall (u, v) \in E \end{aligned}$$

Nell'esempio di prima, la soluzione della prima figura non è *ottima* perché $d[D] > d[A] + w(A, D)$.

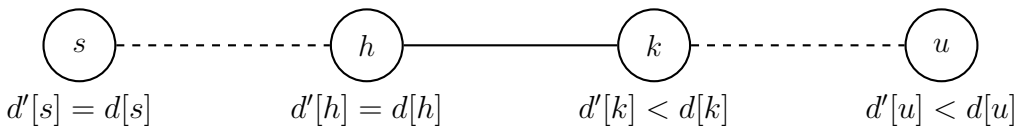
Dimostrazione. Dimostriamo separatamente le due parti del teorema.

Parte 1 Sia T una *soluzione ottima* e sia $(u, v) \in E$:

- Se $(u, v) \in E_T$, allora $d[v] = d[u] + w(u, v)$;
- Se $(u, v) \notin E_T$, allora $d[v] \leq d[u] + w(u, v)$, perché altrimenti esisterebbe nel *grafo* G un *cammino* da s a v più corto di quello in T, generando un assurdo.

Parte 2 Supponiamo per assurdo che T non sia una *soluzione ottima*. Se così fosse, esisterebbe un *cammino non ottimo* C da s a un altro *nodo* u in T. Quindi, esisterebbe anche un *albero di copertura* T', in cui il *cammino* C' da s a u abbia distanza $d'[u] < d[u]$ dove d' è il vettore delle distanze associato a T'.

Poiché, $d'[s] = d[s] = 0$, ma $d'[u] < d[u]$, esiste un *arco* (h, k) in C' tale che $d'[h] = d[h]$ e $d'[k] < d[k]$. La situazione sarebbe dunque la seguente:



Per costruzione, $d'[h] = d[h]$ e $d'[k] = d'[h] + w(h, k)$, mentre, per ipotesi, vale $d[k] \leq d[h] + w(h, k)$. Combinando queste due relazioni si ottiene:

$$d'[k] = d'[h] + w(h, k) = d[h] + w(h, k) \geq d[k]$$

Da ciò seguirebbe $d'[k] \geq d[k]$ che contraddice la relazione $d'[k] < d[k]$ trovata in precedenza. □

13.1.1 Prototipo di algoritmo

Vediamo quale potrebbe essere la struttura base di un algoritmo per la ricerca dei *cammini minimi*.

Frammento 104 - Prototipo di algoritmo.

```
⟨int[], int[]⟩ minPathPrototype(GRAPH G, NODE s)
   % Inizializza T a una foresta di copertura composta da nodi isolati
   % Inizializza d con sovrastima della distanza, cioè d[s] = 0, d[x] = +∞
   while (∃⟨u, v⟩ : d[u] + G.w(u, v) < d[v]) do
      d[v] = d[u] + G.w(u, v)
      % Sostituisci il padre di v in T con u
   return ⟨T, d⟩
```

Frammento 105 - Algoritmo generico.

```
⟨int[], int[]⟩ shortestPath(GRAPH G, NODE s)
   int[] T = new int[1...G.size()]          % T[u] è il padre di u nell'albero T
   int[] d = new int[1...G.size()]                % d[u] è la distanza di u da s
   boolean[] b = new boolean[1...G.size()]              % b[u] è true se u ∈ S
   foreach (u ∈ G.V() - {s}) do
      T[u] = nil
      d[u] = +∞
      b[u] = false
   T[s] = nil
   d[s] = 0
   b[s] = true
   DATASTRUCTURE S = DataStructure()
   S.add(s)
   while (not S.isEmpty()) do
      int u = S.extract()
      b[u] = false
      foreach (v ∈ G.adj(u)) do
         if (d[u] + G.w(u, v) < d[v]) then
            if (not b[v]) then
               S.add(v)
               b[v] = true
            else
               % Azione da svolgere nel caso v ∈ S
            T[v] = u
            d[v] = d[u] + G.w(u, v)
   return ⟨T, d⟩
```

13.1.2 Algoritmo di Dijkstra

La prima implementazione vera e propria che vediamo è quella proposta da Dijkstra nel 1959. Si basa su *code a priorità* e funziona bene solo se i pesi sono positivi.

NB. Siccome gli *heap* furono introdotti solo nel 1964, la prima versione dell'algoritmo utilizzava *code a priorità* implementate usando un vettore.

Frammento 106 - Algoritmo di Dijkstra.

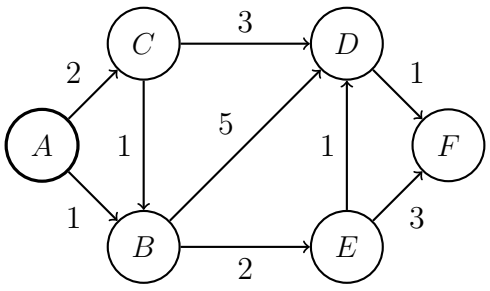
```
⟨int[], int[]⟩ shortestPath(GRAPH G, NODE s)
   int[] T = new int[1...G.size()]          % T[u] è il padre di u nell'albero T
   int[] d = new int[1...G.size()]                % d[u] è la distanza di u da s
   boolean[] b = new boolean[1...G.size()]              % b[u] è true se u ∈ S
   foreach (u ∈ G.V() - {s}) do
      T[u] = nil
      d[u] = +∞
      b[u] = false
   T[s] = nil
   d[s] = 0
   b[s] = true
   PRIORITYQUEUE Q = PrioriryQueue()
   Q.insert(s, 0)                                          % La priorità di s è 0
   while (not Q.isEmpty()) do
      int u = Q.deleteMin()   % A ogni ciclo viene estratto il nodo più vicino
      b[u] = false
      foreach (v ∈ G.adj(u)) do
         if (d[u] + G.w(u, v) < d[v]) then
            if (not b[v]) then
               Q.insert(v, d[u] + G.w(u, v))          % Aggiungo v alla coda
               b[v] = true
            else
               Q.decrease(v, d[u] + G.w(u, v))  % Riduco la priorità di v da s
         T[v] = u
         d[v] = d[u] + G.w(u, v)
   return ⟨T, d⟩
```

L'idea alla base di questa soluzione è quella di usare la distanza di un nodo da s come valore per la sua *priorità*, quindi ad ogni iterazione, estrarre il *nodo* con la *priorità* minore significa estrarre il *nodo* il cui cammino da s è di peso minore.

Quando viene estratto un *nodo* e l'esecuzione dell'algoritmo ricade nel ramo `else` del controllo sul valore $b[v]$, significa che dal *nodo* u è possibile raggiungere v con costo minore a quello che stiamo pagando attualmente. Di conseguenza, aggiorniamo il valore di popolarità di v indicando il costo del nuovo *cammino*.

Esempio 34 - Esempio d'esecuzione.

Consideriamo il seguente grafo:



Eseguendo l'algoritmo partendo da A otteniamo la seguente tabella:

		A	***B***	***C***	***E***	***D***	***F***
A	0	**0**	∅	∅	∅	∅	∅
B	$+\infty$	1	**1**	~~1~~	~~1~~	~~1~~	~~1~~
C	$+\infty$	2	2	**2**	~~2~~	~~2~~	~~2~~
D	$+\infty$	$+\infty$	6	5	4	**4**	~~4~~
E	$+\infty$	$+\infty$	3	3	**3**	~~3~~	~~3~~
F	$+\infty$	$+\infty$	$+\infty$	$+\infty$	6	5	**5**

Nella tabella, ogni colonna contiene lo stato del vettore d all'inizio di ogni iterazione del ciclo `while (not Q.isEmpty())`*, mentre ogni riga traccia l'evoluzione dello stato del rispettivo nodo. I nodi barrati sono quelli che non sono presenti nella coda.*

Prima dell'ingresso nel ciclo, la sorgente, ovvero il nodo A è a distanza 0 da sé stesso. Tutti gli altri sono a $+\infty$. Alla prima iterazione viene estratto A e vengono inseriti i nodi B e C ad esso adiacenti, indicando anche il peso del relativo arco.

Alla seconda iterazione viene estratto B perché il costo per raggiungerlo è minimo. Come prima, vengono inseriti nella coda i nodi D ed E che sono adiacenti a B. L'algoritmo continua fino all'estrazione di F.

Dimostrazione di correttezza per pesi positivi

Dimostrazione. La correttezza dell'algoritmo per pesi positivi si basa su due assunti:

1. Ogni *nodo* viene estratto una sola volta;
2. Al momento dell'estrazione il peso dal *cammino* dalla sorgente s è minimo;

Per la dimostrazione procediamo per induzione sul numero k di *nodi* estratti.

Caso base: $k = 0$ Il caso è verificato in quanto $d[s] = 0$ e non ci sono pesi negativi.

Passo induttivo: $k > 0$ Per ipotesi induttiva supponiamo che gli assunti siano corretti per i primi $k - 1$ elementi. Quando viene estratto il k-esimo elemento u, il peso $d[u]$ dipende esclusivamente dai *nodi* già estratti, quindi la sua distanza da s è minima. Siccome non ci sono pesi negativi e tutti gli altri *nodi* hanno una distanza da s almeno pari a quella di u, u non verrà mai più inserito nella *coda*. □

Complessità Il costo per inizializzare i vettori T, d e b è $\Theta(n)$, la ricerca del minimo all'interno di una *coda a priorità* implementata come vettore costa $\Theta(n)$, mentre l'inserimento e la modifica della *priorità* di un elemento costano $\Theta(1)$. Complessivamente, dunque, l'algoritmo ha *complessità* $T(n) = \Theta(n^2)$ perché vengono inseriti e rimossi dalla *coda* tutti i *nodi*.

Esistono altre versioni dello stesso algoritmo che però fanno uso di implementazioni più efficienti delle *code a priorità*.

La versione di *Johnson* del 1977 si basa su *heap binari* e il costo delle operazioni sulla *coda* diventa $O(\log n)$, riducendo la *complessità* totale a $T(n) = O(m \log n)$. Una terza versione, quella proposta da *Fredman e Tarjan* nel 1987, usa gli *heap di Fibonacci*, per cui le operazioni di inserimento e rimozione costano $O(\log n)$, mentre la modifica di *priorità* ha un *costo ammortizzato* $O(1)$. La *complessità* complessiva è $O(m + n \log n)$.

13.1.3 Algoritmo di Bellman-Ford-Moore

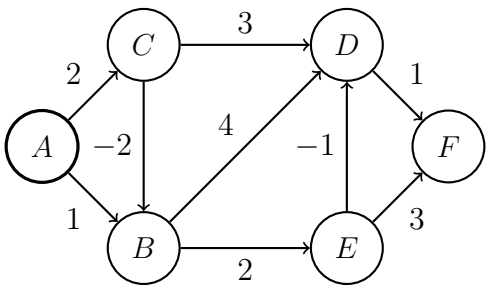
Questo algoritmo ha una struttura simile a quello appena visto, ma al posto di una *coda a priorità* viene usata una *coda* "classica". Questa soluzione permette di trattare anche *grafi* con *archi* di peso negativo.

Frammento 107 - Algoritmo di Bellman-Ford-Moore.

```
⟨int[], int[]⟩ shortestPath(GRAPH G, NODE s)
   int[] T = new int[1...G.size()]          % T[u] è il padre di u nell'albero T
   int[] d = new int[1...G.size()]                % d[u] è la distanza di u da s
   boolean[] b = new boolean[1...G.size()]              % b[u] è true se u ∈ S
   foreach (u ∈ G.V() - {s}) do
      T[u] = nil
      d[u] = +∞
      b[u] = false
   T[s] = nil
   d[s] = 0
   b[s] = true
   QUEUE Q = Queue()
   Q.enqueue(s)
   while (not Q.isEmpty()) do
      int u = Q.dequeue()
      b[u] = false
      foreach (v ∈ G.adj(u)) do
         if (d[u] + G.w(u, v) < d[v]) then
            if (not b[v]) then
               Q.enqueue(v)
               b[v] = true
         T[v] = u
         d[v] = d[u] + G.w(u, v)
   return ⟨T, d⟩
```

Esempio 35 - Esempio d'esecuzione.

Consideriamo il seguente grafo:



Eseguendo l'algoritmo partendo da A otteniamo la seguente tabella:

		Passate										
Nodo	**Inizio**	**0**	**1**		**2**			**3**			**4**	**5**
		A	***B***	***C***	***D***	***E***	***B***	***F***	***D***	***E***	***D***	***F***
A	0	0	0	0	0	0	0	0	0	0	0	0
B	$+\infty$	1	1	0	0	0	0	0	0	0	0	0
C	$+\infty$	2	2	2	2	2	2	2	2	2	2	2
D	$+\infty$	$+\infty$	5	5	5	3	3	3	3	2	2	2
E	$+\infty$	$+\infty$	4	4	4	4	3	3	3	3	3	3
F	$+\infty$	$+\infty$	$+\infty$	$+\infty$	6	6	6	6	4	4	3	3
Coda	*A*	*BC*	*CDE*	*DEB*	*EBF*	*BFD*	*FDE*	*DE*	*E*	*D*	*F*	

Inizialmente, il peso dei cammini da A a tutti gli altri nodi è $+\infty$. Nella passata 0 vengono considerati tutti i nodi che possono essere raggiunti da A attraversando 0 archi. Ovviamente, viene considerato soltanto A che è anche l'unico nodo presente nella coda.

Quindi, A viene estratto dalla coda e ne vengono considerati i vicini: B e C. Siccome il peso dei cammini da A verso i suoi vicini è minore del valore attualmente noto (i.e. $+\infty$), B e C vengono aggiunti alla coda e i pesi dei cammini vengono aggiornati.

Nella passata 1 vengono considerati tutti i nodi che possono essere raggiunti da A attraversando un arco, e di nuovo ciò coincide con i nodi presenti nella coda. Da B è possibile raggiungere D ed E in modo conveniente, quindi questi vengono accodati e il costo per raggiungerli viene aggiornato. Da C invece, si possono raggiungere D e B. Per quanto riguarda D, il costo del cammino passante per C è $2 + 3 = 5$ che è uguale a quello del cammino passante per B, quindi D non viene aggiunto nuovamente alla coda. D'altra parte, il cammino che raggiunge B passando per C ha peso totale pari a 0, che è minore del peso del cammino diretto da A, quindi B viene riaggiunto in coda.

L'algoritmo continua in questo modo fino a quando non viene svuotata la coda.

Possiamo definire in modo più formale ciò che abbiamo chiamato *passata*. Sfruttando la ricorsione possiamo dire che per $k = 0$, la 0-*esima passata* consiste nell'estrazione dalla *coda* del solo *nodo* sorgente. Per $k > 0$ invece, la k-*esima passata* consiste nell'estrazione dalla *coda* di tutti i *nodi* presenti al termine della $k - 1$-*esima passata*.

Questo ci permette anche di dire che al termine della k-*esima passata*, i vettori T e d descrivono i *cammini minimi* di lunghezza k. Conseguentemente, dopo la $n - 1$-*esima passata*, dove n è il numero di *nodi*, sono noti tutti i *cammini minimi* del *grafo*. Ogni *nodo* può infatti essere inserito nella *coda* al massimo $n - 1$ volte.

Complessità L'inizializzazione dei vettori costa sempre $\Theta(n)$, mentre tutte le operazioni sulla *coda* hanno *complessità* $\Theta(1)$. Poiché ogni *nodo* può essere reinserito nella *coda* fino a $n - 1$ volte, la `dequeue` viene eseguita $O(n^2)$ volte. L'`enqueue` viene invece invocata tante volte quante sono i vicini di ogni *nodo*, ovvero $O(nm)$, dove m è il numero di *archi* del *grafo*. La *complessità* totale dell'algoritmo è dunque $O(nm)$.

13.1.4 Ricerca dei cammini minimi sui grafi DAG

I *grafi DAG* sono un caso particolare in quanto, non essendoci *cicli*, i *cammini* sono sempre definiti anche in presenza di pesi negativi. Di conseguenza, possiamo ricercare i *cammini minimi* semplicemente applicando l'algoritmo per l'*ordinamento topologico*.

Frammento 108 - Ricerca dei cammini minimi su grafi DAG.

```
⟨int[], int[]⟩ shortestPath(GRAPH G, NODE s)
   int[] T = new int[1...G.size()]
   int[] d = new int[1...G.size()]
   foreach (u ∈ G.V() - {s}) do
      T[u] = nil
      d[u] = +∞
   T[s] = nil
   d[s] = 0
   STACK S = topsort(G)
   while (not S.isEmpty()) do
      NODE u = S.pop()
      foreach (v ∈ G.adj(u)) do
         if (d[u] + G.w(u, v) < d[v]) then
            T[v] = u
            d[v] = d[u] + G.w(u, v)
   return ⟨T, d⟩
```

13.1.5 Casi d'uso delle soluzioni

Algoritmo	Complessità	Caso d'uso
Dijkstra	$\Theta(n^2)$	Pesi positivi, grafi densi
Johnson	$O(m \log n)$	Pesi positivi, grafi sparsi
Fredman-Tarjan	$O(m + n \log n)$	Pesi positivi, grafi densi, dimensioni molto grandi
Bellman-Ford-Moore	$O(nm)$	Pesi negativi
topsort	$O(n + m)$	Grafi DAG
bfs	$O(n + m)$	Grafi non pesati

13.2 Ricerca dei cammini minimi da sorgente multipla

Un possibile approccio al problema è quello di invocare gli algoritmi appena visti una volta per ogni *nodo sorgente*. Il costo che si andrebbe a pagare sarebbe n volte il costo delle singole esecuzioni.

13.2.1 Algoritmo di Floyd-Warshall

L'intuizione alla base dell'algoritmo è quella dei *cammini k-vincolati*.

Definizione 126 - Cammini k-vincolati.

Sia $k \in \{0, \ldots, n\}$. Diciamo che un cammino p^k_{xy} è un cammino minimo k-vincolato fra x e y se esso ha il costo minimo tra tutti i cammini minimi da x a y che non passano per nessun vertice in $\{v_{k+1}, \ldots, v_n\}$.

NB. Stiamo assumendo, come in realtà abbiamo sempre fatto, che esista un ordinamento fra i *nodi* $v_1, \ldots, v_n$ del *grafo*.

NB. Seguendo la definizione data, p^0_{xy} e p^n_{xy} corrispondono rispettivamente all'*arco* da x a y, se esiste, e al *cammino minimo* da x a y in tutto il *grafo*, cioè senza vincoli.

Se esistono i *cammini k-vincolati* possiamo definire anche la *distanza k-vincolata*.

Definizione 127 - Distanza *k*-vincolata.

Denotiamo con $d^k[x][y]$ il costo del cammino k-vincolato da x a y:

$$d^k[x][y] = \begin{cases} w(p^k_{xy}) & \textit{Se esiste} \\ +\infty & \textit{altrimenti} \end{cases}$$

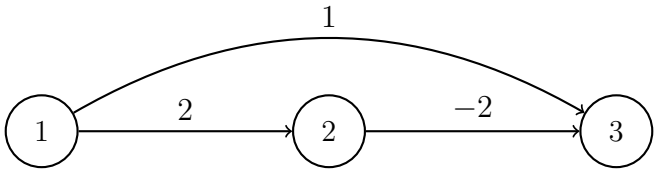
NB. Parallelamente a prima, $d^0[x][y]$ e $d^n[x][y]$ corrispondono rispettivamente al costo dell'*arco* tra x e y, se esiste, e al costo del *cammino minimo* da x a y in tutto il *grafo*.

I *cammini k-vincolati* ci permettono di ricercare i *cammini minimi* in modo ricorsivo partendo da $k = 0$ e incrementandolo, e tenendo di volta in volta il *cammino* di costo minore. Possiamo quindi esprimere la *distanza k-vincolata* con la seguente:

$$d^k[x][y] = \begin{cases} w(x,y) & k = 0 \\ \min\left(d^{k-1}[x][y], d^{k-1}[x][k] + d^{k-1}[k][y]\right) & k > 0 \end{cases}$$

Esempio 36 - Esempio del calcolo delle distanze *k*-vincolate.

Consideriamo il seguente grafo:



Le distanze k-vincolate tra 1 e 3 per valori crescenti di k sono:

$$\begin{aligned} d^0[1][3] &= 1 \\ d^1[1][3] &= 1 \\ d^2[1][3] &= \min\left(d^1[1][3], d^1[1][2] + d^1[2][3]\right) \\ &= \min(1, 0) = 0 \end{aligned}$$

Oltre alla matrice d dei costi, definiamo anche una matrice T dei *padri* dove $T[x][y]$ rappresenta il *predecessore* di y in nel *cammino minimo* tra x e y. Nell'esempio di prima, avremmo avuto $T[1][2] = 1$, $T[2][3] = 3$ e $T[1][3] = 2$.

Frammento 109 - Algoritmo di Floyd-Warshall.

```
⟨int[][], int[][]⟩ floydWarshall(GRAPH G)
   int[][] d = new int[1...G.size()][1...G.size()]
   int[][] T = new int[1...G.size()][1...G.size()]
```

```
foreach (u ∈ G.V()) do
   foreach (v ∈ G.V()) do
      d[u][v] = +∞
      T[u][v] = nil
foreach (u ∈ G.V()) do
   foreach (v ∈ G.adj(u)) do
      d[u][v] = G.w(u, v)
      T[u][v] = u
for (k=1 to n) do
   foreach (u ∈ G.V()) do
      foreach (v in G.V()) do
         if (d[u][k] + d[k][v] < d[u][v]) then
            d[u][v] = d[u][k] + d[k][v]
            T[u][v] = T[k][v]
return ⟨d, T⟩
```

Complessità I tre cicli annidati fanno sì che la *complessità* dell'algoritmo risulti $\Theta(n^3)$.

Capitolo Nr.14

Programmazione greedy

Gli algoritmi realizzati per risolvere *problemi di ottimizzazione* eseguono una sequenza di decisioni che nel caso della *programmazione dinamica* sono prese in maniera bottom-up e vengono valutate tutte le possibilità, evitando però di ripetere decisioni già esaminate. Con la *programmazione greedy* invece, tra tutte le possibili decisioni ne viene scelta solo una, quella che sembra ottima, o meglio localmente ottima. Tuttavia, è necessario dimostrare che quella particolare scelta permette di ottenere un risultato ottimo anche a livello globale.

Proprio per questo motivo, la *programmazione greedy* è utilizzabile sono per quei problemi che presentano una *sottostruttura ottima* e per i quali è possibile dimostrare che esiste una scelta "ingorda", cioè una scelta che porta ad una soluzione ottima.

14.1 Problema del resto

Problema 14 - Problema del resto.

Dati un insieme di "tagli" di monete, espressi come un vettore $t[1 \dots n]$ di interi positivi, e un valore R rappresentate un resto da dover restituire, trovare il più piccolo numero intero di pezzi necessari per dare un resto R di centesimi utilizzando i tagli disponibili, assumendo di avere un numero illimitato di monete.

Formalmente, è necessario trovare un vettore x di interi non negativi tale che:

$$R = \sum_{i=1}^{n} x[i] \cdot t[i] \quad e \quad m = \sum_{i=1}^{n} x[i] \quad ha\ valore\ minimo$$

Questo problema possiede una *sottostruttura ottima*:

Definizione 128 - Sottostruttura ottima.

Siano $S(i)$ il problema di dare un resto pari ad i e $A(i)$ una soluzione ottima del problema $S(i)$ rappresentata da un multi-insieme. Sia poi $j \in A(i)$. Allora, $S(i - t[j])$ è un sotto-problema di $S(i)$, la cui soluzione è data da $A(i) - \{j\}$.

14.1.1 Approccio basato su programmazione dinamica

Prima di vedere come potrebbe essere realizzata una soluzione *greedy*, utilizziamo le proprietà della *sottostruttura ottima* per implementare una soluzione basata su *programmazione dinamica*.

Se $DP[i]$ rappresenta il numero minimo di monete necessario a risolvere il problema $S(i)$, vale la seguente funzione ricorsiva:

$$DP[i] = \begin{cases} 0 & i = 0 \\ \min_{1 \leq j \leq n}\{DP[i - t[j]] \mid t[j] \leq i\} + 1 & i > 0 \end{cases}$$

Frammento 110 - Soluzione basata su programmazione dinamica.

```
int[] moneyChange(int[] t, int n, int R)
   int[] DP = new int[0...R]                          % Tabella delle soluzioni
   int[] coins = new int[0...R]       % Moneta da usare per uno specifico resto
   DP[0] = 0
   for (i = 1 to R) do
      DP[i] = +∞
      for (j = 1 to n) do
         if (i > t[j] and DP[i - t[j]] + 1 < DP[i]) then
            DP[i] = DP[i - t[j]] + 1
            coins[i] = j
   % Ricostruzione della soluzione
   int[] x = new int[1...n] = {0}               % Contatore delle monete usate
   while (R > 0) do
      x[coins[R]] = x[coins[R]] + 1
      R = R - t[coins[R]]
   return x
```

Complessità La *complessità* di questa soluzione è $\Theta(nR)$ per la presenza dei due cicli annidati.

14.1.2 Approccio basato su programmazione greedy

Non è difficile immaginare che per minimizzare il numero di monete utilizzate sia sufficiente usare sempre il taglio più grande possibile. Ovvero, selezionare la moneta j più grande, tale per cui $t[j] < R$, e risolvere il sotto-problema $S(R - t[j])$.

Frammento 111 - Soluzione basata su programmazione greedy.

```
int[] moneyChange(int[] t, int n, int R)
   int[] x = new int[1...n]
   { Ordina le monete in modo decrescente }
   for (i = 1 to n) do
      x[i] = ⌊R / t[i]⌋
      R = R - x[i] · t[i]
   return x
```

Complessità Questa soluzione abbassa la *complessità* a $O(n \log n)$ se l'insieme di monete non è ordinato, $O(n)$ altrimenti.

Ciò che dobbiamo fare adesso è dimostrare che la scelta fatta sia corretta.

Dimostrazione. Supponiamo $t = [50, 10, 5, 1]$. Sia x una qualunque soluzione ottima, ovvero tale per cui:

$$R = \sum_{i=1}^{n} x[i] \cdot t[i] \quad \text{e} \quad m = \sum_{i=1}^{n} x[i] \quad \text{ha valore minimo}$$

Sappiamo che $t[k] \cdot x[k] < t[k-1]$ altrimenti basterebbe sostituire un certo numero di monete di taglia $t[k]$ con quelle del taglio $t[k-1]$. In questo caso, i limiti al numero di monete per ogni taglio sono:

$$\begin{array}{rcrcrcrcl} t[2] \cdot x[2] & = & 10 \cdot x[2] & < & t[1] & = & 50 & \Rightarrow & x[2] < 5 \\ t[3] \cdot x[3] & = & 5 \cdot x[3] & < & t[2] & = & 10 & \Rightarrow & x[3] < 2 \\ t[4] \cdot x[4] & = & 1 \cdot x[4] & < & t[3] & = & 5 & \Rightarrow & x[4] < 5 \end{array}$$

Sia ora m_k la somma delle monete di taglio inferiore a $t[k]$:

$$m_k = \sum_{i=k+1}^{4} x[i] \cdot t[i]$$

Se riusciamo a dimostrare che $m_k < t[k]\ \forall k$, allora la soluzione proposta dall'algoritmo è proprio quella ottima. Per ogni valore di k otteniamo:

$$\begin{array}{rcrcrcrcrcl} m_4 & = & 0 & & & & < & 1 & & & = t[4] \\ m_3 & = & x[4] \cdot 1 + m_4 & \leq & 4 \cdot 1 + m_4 & < & 4 + 1 & = & 5 & & = t[3] \\ m_2 & = & x[3] \cdot 1 + m_3 & \leq & 1 \cdot 5 + m_3 & < & 5 + 5 & = & 10 & & = t[2] \\ m_1 & = & x[2] \cdot 1 + m_2 & \leq & 4 \cdot 10 + m_2 & < & 40 + 10 & = & 50 & & = t[1] \end{array}$$

□

NB. Non tutti i tagli di monete permettono di utilizzare la *programmazione greedy*. Ad esempio, per $t[10, 8, 1]$ l'algoritmo *greedy* utilizzerebbe una moneta da 10 e 7 da 1, quando la scelta migliore sarebbe quella di prenderne 2 da 8 e una da 1.

NB. In ogni caso, l'insieme delle monete deve includere una moneta di taglio unitario.

14.2 Insieme indipendente massimale di intervalli

Vediamo un caso particolare del problema sulla Ricerca dell'insieme indipendente di peso massimo in cui il peso di tutti gli intervalli è 1. Cioè vogliamo cercare l'insieme contenente il maggior numero di intervalli.

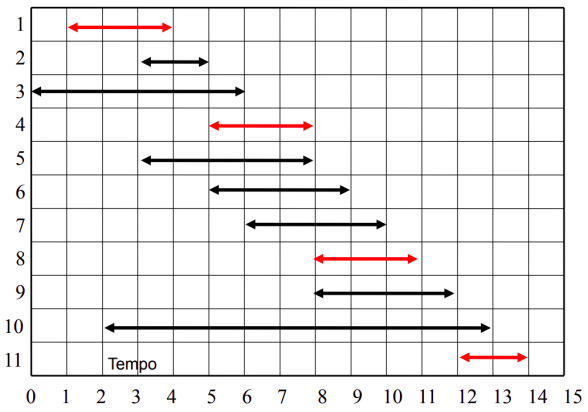


Fig. 14.1: Possibile soluzione

14.2.1 Approccio basato su programmazione dinamica

Iniziamo ricercando una *sottostruttura ottima*. Assumiamo che gli intervalli siano ordinati per tempo di fine:

$$b_1 \leq \cdots \leq b_n$$

Definiamo il sotto-problema $S[i \dots j]$ come l'insieme di intervalli che iniziano dopo la fine di i e finiscono prima dell'inizio di j:

$$S[i \dots j] = \{k \mid b_i \leq a_k < b_k \leq a_j\}$$

Aggiungiamo due intervalli fittizi:

- *Intervallo* 0: $b_0 = -\infty$;
- *Intervallo* $n+1$: $a_{n+1} = +\infty$;

Il problema generale corrisponde quindi al problema $S[0 \dots n+1]$.

Definizione 129 - Sottostruttura ottima.

Supponiamo che $A[i \dots j]$ sia una soluzione ottimale di $S[i \dots j]$ e sia k un intervallo che appartiene ad $A[i \dots j]$. Allora:

- *Il problema $S[i \dots j]$ può essere suddiviso in due intervalli:*
  1. *$S[i \dots k]$ contenente gli intervalli di $S[i \dots j]$ che finiscono prima di k;*
  2. *$S[k \dots j]$ contenente gli intervalli di $S[i \dots j]$ che iniziano dopo k;*
- *$A[i \dots j]$ contiene le soluzioni ottimali di $S[i \dots k]$ e $S[k \dots j]$:*
  1. *$A[i \dots j] \cap S[i \dots k]$ è la soluzione ottimale di $A[i \dots k]$;*
  2. *$A[i \dots j] \cap S[k \dots j]$ è la soluzione ottimale di $A[k \dots j]$;*

Quanto detto ci permette di definire $A[i \dots j]$ come:

$$A[i \dots j] = A[i \dots k] \cup \{k\} \cup A[k \dots j]$$

Per determinare k proviamo tutte le possibilità. A questo punto, sia $DP[i][j]$ la dimensione del più grande sottoinsieme $A[i \dots j] \subseteq S[i \dots j]$ di intervalli indipendenti. Possiamo definire $DP[i][j]$ con la seguente funzione ricorsiva:

$$DP[i][j] = \begin{cases} 0 & S[i \dots j] = \emptyset \\ \max_{k \in S[i \dots j]} \{DP[i][k] + DP[k][j] + 1\} & \text{altrimenti} \end{cases}$$

Questa definizione ci permette di realizzare un algoritmo basato su *programmazione dinamica*, o su *memoization*, con *complessità* pari a $O(n^3)$. Questo perché bisogna risolvere tutti i problemi con $i < j$ e, nel caso peggiore, paghiamo $O(n)$ per ogni sotto-problema.

Possiamo fare meglio di così?

Potremmo semplicemente usare la soluzione al problema con pesi generici, che ha *complessità* $O(n \log n)$, ponendo a 1 tutti i pesi. Tuttavia, possiamo provare a chiederci se sia davvero necessario analizzare tutti i possibili valori di k. Definiamo il seguente teorema:

Definizione 130 - Scelta ingorda.

Siano $S[i \dots j]$ un sotto-problema non vuoto e m l'intervallo di $S[i \dots j]$ con il minor tempo di fine. Allora, valgono i due seguenti punti:

1. *Il sotto-problema $S[i \dots m]$ è vuoto;*
2. *m è compreso in una qualche soluzione ottima di $S[i \dots j]$, ovvero $m \in A[i \dots j]$;*

Dimostrazione. Dimostriamo separatamente i due punti.

Punto 1 - $S[i \dots j] = \emptyset$ Per la definizione di intervallo sappiamo che $a_m < b_m$. Inoltre, poiché m è l'intervallo con il minor tempo di fine $\forall k \in S[i \dots j]$ $b_m \leq b_k$ e, quindi, ne consegue che $\forall k \in S[i \dots j]$ $a_m < b_k$. Se nessun intervallo in $S[i \dots j]$ termina prima di a_m, allora $S[i \dots m] = \emptyset$.

Punto 2 - $m \in A[i \dots i]$ Siano $A'[i \dots j]$ una soluzione ottima di $S[i \dots j]$ e m' l'intervallo con minor tempo di fine in $A'[i \dots j]$. Sia ora $A[i \dots j] = A'[i \dots j] - \{m'\} \cup \{m\}$ una nuova soluzione ottima ottenuta togliendo m' e aggiungendo m ad $A'[i \dots j]$. $A[i \dots j]$ è una soluzione ottima che contiene m in quanto ha la stessa cardinalità di $A'[i \dots j]$ e gli intervalli sono indipendenti. □

Questa dimostrazione ci permette di selezionare sempre gli intervalli con tempo di fine minore e ignorare tutti quelli che si intersecano con esso, riducendo il problema alla ricerca dell'*insieme indipendente massimale* in $S[m \dots j]$.

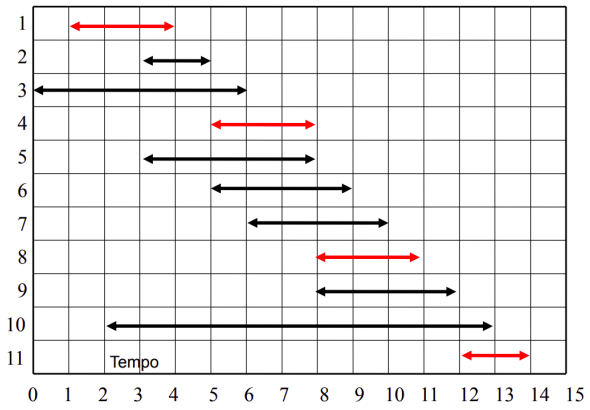


Fig. 14.2: Soluzione con *scelte ingorde*

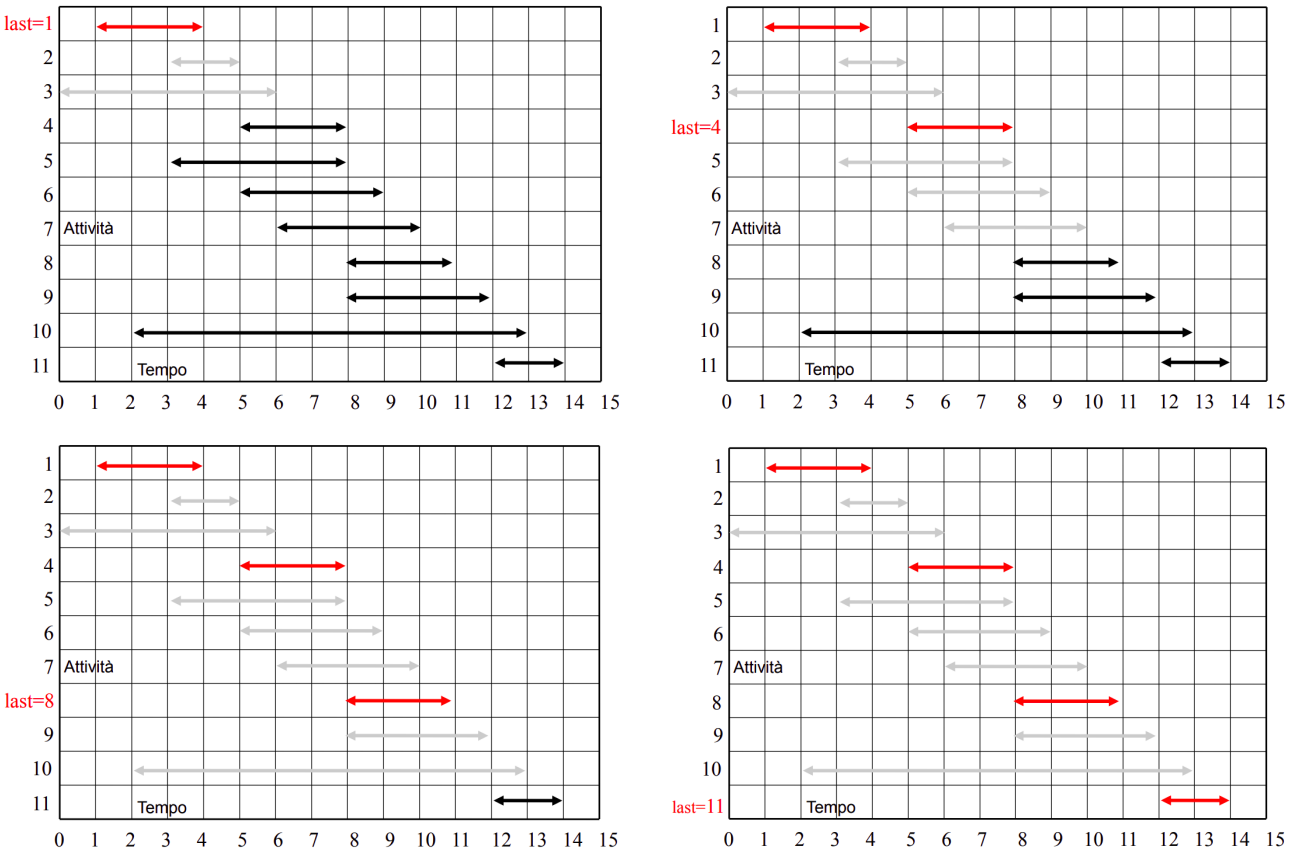
Frammento 112 - Soluzione basata su programmazione greedy.

```
SET indipendentSet(int[] a, int[] b)
   { Ordina i vettori a e b in modo che b_1 ≤ ··· ≤ b_n }
   SET S = Set()
   S.insert(1)
   int last = 1
   for (i = 2 to n) do
      if (a[i] ≥ b[last]) then
         S.insert(i)
         last = i
   return S
```

Complessità Questa soluzione abbassa la *complessità* a $O(n)$ se i vettori di input sono già ordinati, $O(n \log n)$ altrimenti.

Esempio 37 - Esempio d'esecuzione.

Il comportamento del nuovo algoritmo con gli stessi intervalli degli esempi precedenti è il seguente:



14.3 Problema dello scheduling

Problema 15 - Scheduling dei processi.

Dati un processore e n processi da eseguire su di esso. Se ogni processo i è caratterizzato da un tempo d'esecuzione $t[i]$ noto a priori, trovare una sequenza d'esecuzione che minimizzi il tempo di completamento medio.

Definizione 131 - Tempo di completamento.

Dato un vettore $A[1 \dots n]$ contenente una permutazione di $\{1 \dots n\}$, il tempo di completamento dell'h-esimo processo nella permutazione è calcolato come:

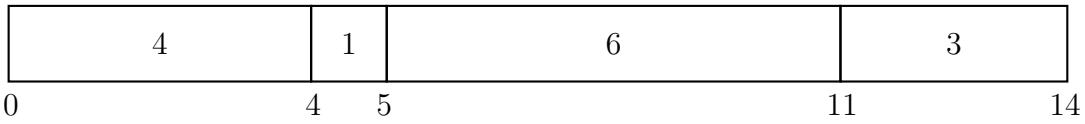
$$T_A(h) = \sum_{i=1}^{h} t[A[i]]$$

Esempio 38 - Possibili permutazioni.

Consideriamo i seguenti processi con relativo tempo d'esecuzione:

Processo	1	2	3	4
$t[i]$	4	1	6	3

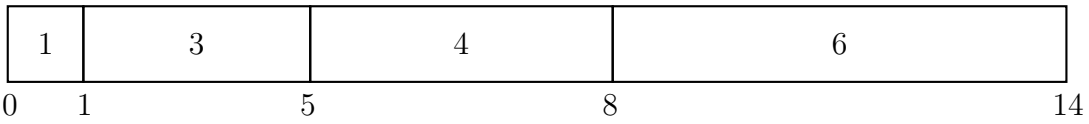
Se eseguissimo i processi nell'ordine $A[1,2,3,4]$ la situazione sarebbe la seguente:



Il tempo di completamento medio sarebbe:

$$\frac{\sum_{i=1}^{4} T_A(i)}{4} = \frac{4+5+11+14}{4} = \frac{34}{4} = 8.5$$

Se invece ordinassimo i processi per tempo d'esecuzione crescente, ponendo $A = [2,4,1,3]$, la situazione diventerebbe:



Il tempo di completamento medio adesso è:

$$\frac{\sum_{i=1}^{4} T_A(i)}{4} = \frac{1+5+8+14}{4} = \frac{27}{4} = 6.75$$

NB. Dall'esempio intuiamo che la scelta ottima potrebbe essere quella di prendere sempre il processo con minor *tempo d'esecuzione*.

Definizione 132 - Scelta ingorda.

Esiste una soluzione ottima A in cui il processo con minor tempo d'esecuzione si trova in prima posizione.

Definizione 133 - Sottostruttura ottima.

Sia A una soluzione ottima di un problema con n processi, in cui il processo con minor tempo d'esecuzione m si trova in prima posizione. La permutazione dei seguenti $n-1$ processi è una soluzione al sotto-problema in cui m non viene considerato.

Dimostrazione. Consideriamo una *permutazione ottima* A:

$$A = [A[1], \dots A[m], \dots, A[n]]$$

Siano m l'indice del processo con minor *tempo d'esecuzione* e A' una permutazione in cui i processi in posizione 1 e m vengono scambiati, ovvero:

$$A = [A[m], \dots, A[1], \dots, A[n]]$$

Il *tempo di completamento medio* di A' è minore o uguale a quello di A. Questo è vero perché i processi nelle posizioni $1, \ldots, m-1$ in A' hanno un *tempo di completamento* minore o uguale a quello dei processi in posizione $1, \ldots, m-1$ in A. D'altra parte, i processi nelle posizioni $m, \ldots, n$ hanno lo stesso *tempo di completamento* sia in A che in A'.

Ora, poiché avevamo ipotizzato che A fosse una *permutazione ottima*, il suo *tempo di completamento medio* non può essere superiore a quello di A', quindi anche A' è una *permutazione ottima*. □

14.4 Problema dello zaino frazionario

Consideriamo una variante del *problema dello zaino* in cui è possibile prendere frazioni di oggetti.

Esempio 39 - Possibili ordinamenti degli oggetti.

Consideriamo uno zaino con capacità 70 e di avere i tre oggetti seguenti:

i	p_i	w_i
1	60	10
2	200	40
3	120	30

Per prendere gli oggetti possiamo provare a ordinarli in qualche modo così poter operare una scelta ponderata sul tipo di ordinamento applicato.

Ordinamento per profitto decrescente *In questo modo andiamo a scegliere gli oggetti* 2 *e* 3 *ottenendo un guadagno totale di* 320.

Ordinamento per peso crescente *Questo approccio ci porta a scegliere la totalità degli oggetti* 1 *e* 3, *per un peso di* 40. *Per usare la rimanente capacità prendiamo i* $\frac{3}{4}$ *dell'oggetto* 2. *Il profitto che otteniamo sale quindi a* 330.

Ordinamento per profitto specifico decrescente *Definiamo il profitto specifico come il rapporto* $\frac{p_i}{w_i}$. *Per gli oggetti* 1, 2, 3 *il profitto specifico vale* 6, 5 *e* 4 *rispettivamente. Di conseguenza, prendiamo la totalità degli oggetti* 1 *e* 2 *e i* $\frac{2}{3}$ *dell'ultimo oggetto. Questa scelta ci assicura un guadagno di* 340.

Frammento 113 - Implementazione della soluzione.

```
float[] zaino(float[] p, float[] w, float C, int n)
   float[] x = new float[1...n]                    % Vettore delle quantità raccolte
   { Ordina p e w in modo che p[1]/w[1] ≥ ··· ≥ p[n]/w[n] }
   for (i = 1 to n) do
      x[i] = min(C / w[i], 1)
      C = C - x[i] · w[i]
   return x
```

Complessità La *complessità* è $O(n)$ se i vettori di input sono già ordinati, $O(n \log n)$ altrimenti.

Abbiamo definito l'algoritmo, ma in realtà non abbiamo ancora dimostrato che quel tipo di scelta sia corretta.

Dimostrazione. Assumiamo che gli oggetti siano ordinati per *profitto specifico decrescente*. Sia x una soluzione ottima e supponiamo che $x[1] < \min\left(\frac{C}{w[i]}, 1\right) < 1$. Allora, possiamo costruire una nuova soluzione in cui $x'[1] = \min\left(\frac{C}{w[i]}, 1\right)$ e la porzione di uno o più oggetti è ridotta di conseguenza. La soluzione x' è sicuramente di profitto uguale o maggiore rispetto ad x perché il *profitto specifico* del primo oggetto è massimo. □

14.5 Problema della compressione

Quello della compressione dei dati è una problema classico dell'informatica. Una delle possibili tecniche per la compressione dei caratteri[1] è tramite una *funzione di codifica* del tipo $f : f(c) = x$ in cui c un carattere preso da un alfabeto Σ e x è la sua rappresentazione binaria.

NB. Σ può essere definito come l'insieme dei caratteri usati all'interno di un file da comprimere.

Esempio 40 - Possibili compressioni.

Supponiamo di avere un file di n caratteri, che $\Sigma = \{a, b, c, d, e, f\}$ e di conoscere la frequenza di ogni carattere.

Caratteri	a	b	c	d	e	f	**Dimensione**
Frequenza	45%	13%	12%	16%	9%	5%	
ASCII	01100001	01100010	01100011	01100100	01100101	01100110	8n
Codifica 1	000	001	010	011	100	101	3n
Codifica 2	0	101	100	111	1101	1100	2.24n

Nella codifica 1 abbiamo codificato i caratteri usando il numero minimo di bit necessari per rappresentare 6 valori. Per quanto riguarda la codifica 2 invece, abbiamo calcolato la dimensione totale come:

$$(0.45 \cdot 1 + 0.13 \cdot 3 + 0.12 \cdot 3 + 0.16 \cdot 3 + 0.09 \cdot 4 + 0.05 \cdot 4)\, n = 2.24n$$

Definizione 134 - Codice a prefisso.

In un codice a prefisso, nessun codice è prefisso di un altro.

La caratteristica dei *codici a prefisso* è necessaria per consentire la decodifica.

Esempio 41 - Codici a prefisso e non.

Consideriamo un codice del tipo:

$$a \to 0,\ b \to 10,\ c \to 11$$

[1] Per le immagini esistono tecniche migliori

La codifica per la stringa `babaca` *è:*

$$10\ 0\ 10\ 0\ 11\ 0$$

che può essere decodificata senza ambiguità percorrendola da sinistra, in quanto, appena viene riscontrata la corrispondenza con un carattere, si può sostituire quel carattere alla sua codifica sapendo che i bit successivi costituiranno la rappresentazione di un altro carattere.

Nel caso di un codice di questo tipo invece:

$$a \to 0,\ b \to 1,\ c \to 11$$

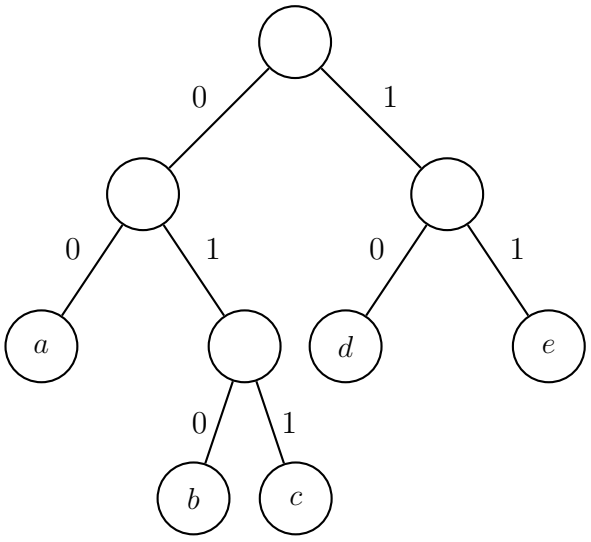
dovendo decodificare la sequenza di bit:

$$111111$$

si crea un'ambiguità, in quanto un singolo bit 1 *rappresenta la* b*, mentre due bit rappresentano la* c*.*

14.5.1 Algoritmo di Huffman

L'*Algoritmo di Huffman* utilizza rappresentazioni ad albero per definire il codice di codifica a partire dal testo e per ritornare al testo originale a partire dalla codifica. In particolare, viene usato un *albero binario* in cui al ramo sinistro di ciascun *nodo* è associato il bit 0, viceversa al ramo destro è sempre associato il bit 1. I caratteri si trovano nelle *foglie* e il *cammino* che si percorre per andare dalla *radice* a ciascuna *foglia* corrisponde alla codifica del carattere associato.



Carattere	a	b	c	d	e
Codifica	00	010	011	10	11

Fig. 14.3: Esempio di *albero binario di decodifica*

Questa rappresentazione ci permette di legare il numero di bit necessari per codificare un carattere alla *profondità* della *foglia* associata all'interno dell'*albero*.

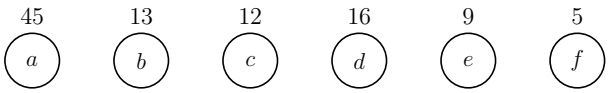
Numero di bit necessari a codificare un file In particolare, se Σ è l'alfabeto di un file F e T è un albero che rappresenta la codifica, per ogni carattere $c \in \Sigma$, $d_T(c)$ rappresenta la *profondità* della *foglia* che rappresenta c in T. Dunque, la codifica di c richiede $d_T(c)$ bit e, se $f[c]$ è il numero di occorrenze di c in F, la dimensione della codifica per l'intero file è:

$$C(F,T) = \sum_{c \in \Sigma} f[c] \cdot d_T(c)$$

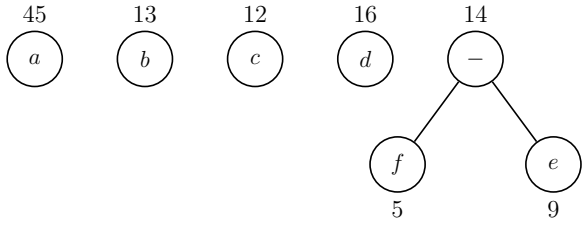
Principi dell'Algoritmo di Huffman L'idea alla base dell'*Algoritmo di Huffman* è quella di minimizzare la lunghezza delle codifiche per i caratteri che compaiono più frequentemente e, al contrario, assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'*albero*.

NB. Ogni codifica è associata a un solo file.

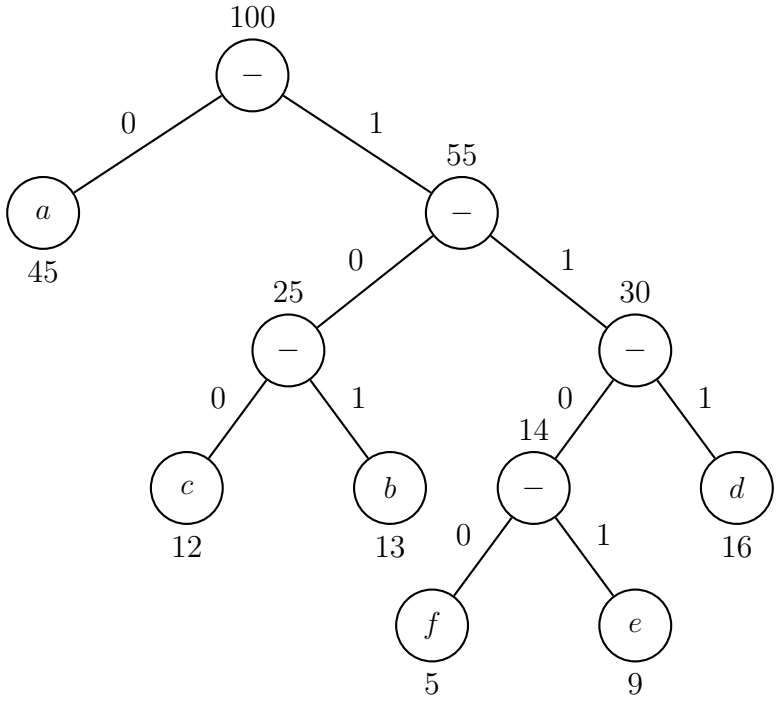
Ad ogni *nodo* dell'*albero* viene associato un carattere con la relativa frequenza; la costruzione dell'*albero* procede per aggregazione delle coppie di *nodi* con frequenza minima. Ad ogni iterazione, i due *nodi* x, y con frequenza f_x e f_y minori vengono resi *figli* di un nuovo *nodo* fittizio con frequenza $f_x + f_y$. Questo *nodo* viene quindi aggiunto all'insieme iniziale di caratteri. L'algoritmo continua in questo modo fino a quando non rimane un solo *nodo* e termina dopo aver etichettato gli *archi*.



(a) *Nodi* iniziali



(b) Prima aggregazione



(c) *Albero* al termine della costruzione

La codifica ottenuta è quindi:

Carattere	a	b	c	d	e	f
Codifica	0	101	100	111	1101	1100

Frammento 114 - Algoritmo di Huffman.

```
TREE huffman(int[] c, int[] f, int n)
   PRIORITYQUEUE Q = MinPriorityQueue()
   for (i = 1 to n) do
      % Inserisco nella coda il nodo c[i] con priorità f[i]
      Q.insert(f[i], Tree(f[i], c[i]))
   for (i = 1 to n - 1) do
      z_1 = Q.deleteMin()
      z_2 = Q.deleteMin()
      z = Tree(z_1.f, z_2.f, nil)
      z.insertLeft(z_1)
      z.insertRight(z_2)
   return Q.deleteMin()
```

NB. In questo frammento ipotizziamo che ad ogni *nodo* sia possibile associare una frequenza oltre che un valore. La firma della funzione `Tree` è quindi `Tree(int frequence, ITEM value)`. Oltre a ciò, ipotizziamo che sia possibile accedere alla frequenza di un *nodo* mediante un attributo `f`.

Complessità La costruzione dell'*albero* attraverso l'uso di *code a priorità* costa $O(n \log n)$, quindi la *complessità* finale è $T(n) = O(n \log n)$.

Come al solito dobbiamo dimostrare la correttezza dell'approccio scelto. In particolare, dobbiamo dimostrare che costruire un *albero* scegliendo sempre i due caratteri con frequenza minore conduce sempre ad una soluzione ottimale. Inoltre, dobbiamo dimostrare che questa soluzione gode di una *sottostruttura ottima*, per la quale, dato un problema sull'alfabeto Σ, è possibile costruire un sotto-problema con un alfabeto più piccolo aggregando i due caratteri con frequenza minore.

Quanto detto è riassumibile nel seguente teorema:

Definizione 135 - Teorema per la correttezza dell'Algoritmo di Huffman.

Dato un alfabeto Σ e un vettore delle frequenze f, se x e y sono i due caratteri con frequenza minore, esiste un codice ottimo per Σ in cui x e y hanno la stessa profondità massima e i loro codici differiscono solo per l'ultimo bit.

Dimostrazione. Supponiamo che esista una codice ottimo T in cui i due caratteri a e b con *profondità massima* siano diversi da x e y. Possiamo assumere senza perdere di generalità che:

$$f[x] \leq f[y] \wedge f[a] \leq f[b]$$

Poiché le frequenze di x e y sono minime, vale:

$$f[x] \leq f[a] \wedge f[y] \leq f[b]$$

Se scambiamo x ed a otteniamo una soluzione T' e se poi scambiamo anche y e b otteniamo una terza soluzione T''. A questo punto, dimostriamo che $C(f, T'') \leq C(f, T') \leq C(f, T)$:

$$\begin{aligned} C(f,T) - C(f,T') &= \textstyle\sum_{c \in \Sigma} f[c] d_T(c) - \sum_{c \in \Sigma} f[c] d_{T'}(c) \\ &= (f[x] d_T(x) - f[a] d_T(a)) - (f[x] d_{T'}(x) - f[a] d_{T'}(a)) \\ &= (f[x] d_T(x) - f[a] d_T(a)) - (f[x] d_T(a) - f[a] d_T(x)) \\ &= (f[a] - f[x]) \cdot (d_T(a) - d_T(x)) \\ &\leq 0 \end{aligned}$$

Allo stesso modo possiamo dimostrar che $C(f, T') - C(f, T'') \geq 0$, ma poiché T è ottimo deve valere anche $C(f, T) \leq C(f, T'')$ e quindi anche T'' è una soluzione ottima. □

Capitolo Nr.A

Appendice A

A.1 Proprietà dei logaritmi

1. $\log_a a = 1$ — *Proprietà fondamentale*
2. $\log_a 1 = 0$ — *Proprietà fondamentale*
3. $\log_a(b \cdot c) = \log_a(b) + \log_a(c)$ — *Teorema del prodotto*
4. $\log_a\left(\frac{b}{c}\right) = \log_a(b) - \log_a(c)$ — *Teorema del rapporto*
5. $\log_a(b^c) = c \cdot \log_a(b)$ — *Teorema della potenza*
6. $\log_{a^n}(b^m) = \frac{m \cdot \log_a(b)}{n}$ — *Potenza alla base e all'argomento*
7. $\log_{\frac{1}{a}}(b) = -\log_a(b)$ — *Base frazionaria*
8. $\log_a\left(\frac{1}{b}\right) = -\log_b(a)$ — *Argomento frazionario*
9. $\log_{\frac{1}{a}}\left(\frac{1}{b}\right) = \log_a(b)$ — *Base e argomento frazionario*
10. $\log_a(b) = \frac{1}{\log_b(a)}$ — *Commutazione base e argomento*
11. $\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$ — *Cambio di base*
12. $a^{\log_b(c)} = c^{\log_b(a)}$ — *Scambio base-argomento*

A.2 Serie matematiche convergenti

1. $\sum_{k=0}^{+\infty} k = \frac{k(k+1)}{2}$ — *Formula di Gauss*
2. $\sum_{k=0}^{n} q^k = \frac{1-q^{n+1}}{1-q} = \frac{q^{n+1}-1}{q-1} \quad \forall\, q : |q| \geq 1$ — *Serie geometrica finita*
3. $\sum_{k=0}^{n} q^k = \frac{1}{1-q} \quad \forall\, q : |q| < 1$ — *Serie geometrica finita*
4. $\sum_{k=0}^{+\infty} q^k = \frac{1}{1-q} \quad \forall\, |q| < 1$ — *Serie geometrica infinita decrescente*
5. $\sum_{k=0}^{+\infty} kq^k = \frac{q}{(1-q)^2} \quad \forall\, |q| < 1$ — *Serie geometrica infinita decrescente*
6. $\sum_{k=1}^{+\infty} \frac{1}{k(k+1)} = \sum_{k=1}^{+\infty} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1$ — *Serie di Mengoli*