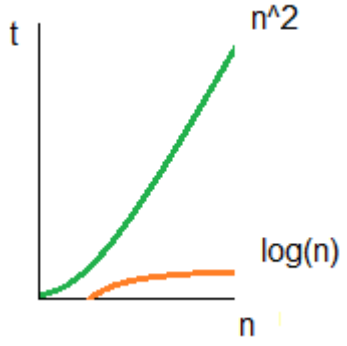
Riassunto di Algoritmi

Alberto Miola

NOTAZIONE ASINTOTICA

Quando studiamo un algoritmo ci interessa sapere principalmente quanto bene è stato fatto. In particolare è di interesse sapere quanto tempo di esecuzione impiega l'algoritmo quando i dati in input sono sempre maggiori (idealmente tendono ad infinito). Quindi un algoritmo è "fatto bene" oppure è "buono" se, al crescere degli elementi in input, il tempo di esecuzione è accettabile.



Quando n cresce (ci si sposta verso destra) il rispettivo valore su t cresce anche lui ma con velocità diverse. Considerando questa piccola tabella:

	n^2	$\log n$
1	1	0
2	4	0,3
3	9	0,5

Riformulando la prima frase; quando il numero di dati in input n cresce, il tempo di esecuzione cresce ma con velocità diverse.

In base alle considerazioni appena fatte possiamo dire che è molto meglio avere un algoritmo che ha andamento log(n) invece di uno che ha n^2 perché al crescere dei dati in input mi darà una risposta molto prima. Magari per valori di n piccoli vanno bene entrambi ma in caso di input da 500 dati la differenza di prestazioni è evidente.

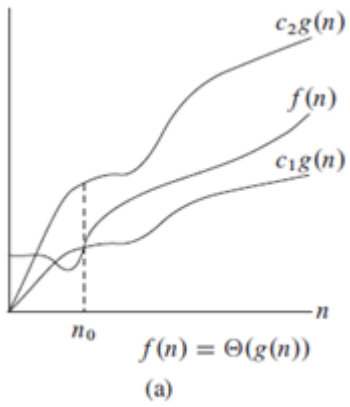
$$1 < \log n < n < n \log n < n^2 < n^3 < 2^n < 3^n < n!$$

Questa è la lista delle funzioni più importanti in ordine crescente: 1 indica la costante mentre da log(n) in poi ci sono tutte le funzioni che al crescere di n impiegano più tempo di esecuzione. Fino alla classe n^k la funzione è accettabile ma c^n (c > 1) e n! sono il peggio e il problema può considerarsi intrattabile.

Formalmente ci sono 3 tipi di notazioni che si possono usare per indicare i discorsi fatti fino ad adesso sul tempo di esecuzione in relazione all'input.

Notazione $\Theta(g(n))$

Theta di g(n)

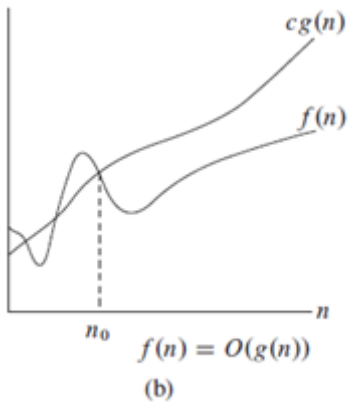


Una funzione f(n) appartiene a $\theta(g(n))$ se esistono due costanti c_1 e c_2 tali che f(n) possa essere racchiusa tra $c_1g(n)$ e $c_2g(n)$.

In altre parole vuol dire che se $f(n) = \Theta(\log n)$ la funzione avrà un andamento all'infinito che sarà circa quello del logaritmo e si distaccherà al massimo di c_1 oppure c_2. Queste due costanti indicano di quanto può variare l'andamento rispetto al logaritmo al massimo o al minimo.

Notazione $O(g(n))$

O grande di g(n)

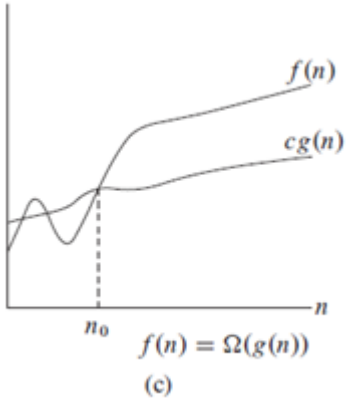


Una funzione f(n) appartiene a $O(g(n))$ se esiste una costante c_1 tale che f(n) sia dominata superiormente da $c_1g(n)$.

A differenza di prima qua mi sto limitando a dire che la funzione a +∞ è dominata superiormente da $c_1g(n)$ ma non so nulla sul limite inferiore. Questo mi serve a capire che all'infinito la f(n) più andare al massimo come $c_1g(n)$ ma non oltre. Questa notazione è la più usata.

Notazione $\Omega(g(n))$

Omega di g(n)



Una funzione f(n) appartiene a $\Omega(g(n))$ se esiste una costante c_1 tale che f(n) sia dominata inferiormente da $c_1g(n)$.

Questa è il contrario di quella prima perché ora ho solo il limite inferiore della funzione. Questo mi serve a capire che all'infinito la f(n) più andare almeno come $c_1g(n)$ ma non più sotto. Per quanto riguarda il limite superiore non so nulla.

Dato che è di interesse l'andamento di una funzione a +∞ (al crescere dei valori n in input) non ci interessa dei valori piccoli all'inizio. Basta sapere che da un certo punto n_0 in poi la funzione ha quell'andamento. La notazione più forte di tutti è $\Theta(g(n))$ e le altre due sono due suoi sottoinsiemi.

- Θ(g(n)) → entrambi i limiti entro i quali posso racchiudere la funzione
- O(g(n)) → solo il limite superiore
- Ω(g(n)) → solo il limite inferiore

Quindi se so di avere f(n) = Θ(g(n)) allora posso affermare anche che f(n) = O(g(n)) e f(n) = Ω(g(n)) perché unisco limite superiore (la O) con limite inferiore (l' Ω) e ottengo tutti e due i limiti (il Θ). Ovviamente NON vale l'inverso, cioè se f(n) = O(g(n)) allora NON è vero che f(n) = Θ(g(n)) perché la O da solo il limite superiore ma il theta entrambi.

Un'ultima cosa importante da ricordare: dato che si guarda sempre tutto a +∞ contano solo i termini di ordine maggiore. Quindi $O(n^7 + 5n^2)$ diventa $O(n^7)$ oppure $O(4^n + \log(n))$ diventa $O(4^n)$ perché i termini di ordine inferiore sono trascurabili. Per capire quali cancellare, basta guardare la catena data sopra con le funzioni.

ANALISI DELL'ALGORITMO

Ci possono essere due tipi di algoritmi da analizzare: quelli iterativi (che non usano la ricorsione) e quelli ricorsivi. Nel primo caso si fa un'analisi dei costi valutando i cicli e i vari componenti dell'algoritmo, nel secondo caso si usano equazioni di ricorrenza.

Caso iterativo.

In questo caso è molto semplice calcolare la complessità: basta guardare se ci sono cicli (for, while, repeat-until) e se sono annidati o meno. Alla fine si farà la somma di tutti i costi. Vediamo un esempio semplice:

```
CodiceDiProva(A[]) {
  a = 5, j = 1                  Assegnazione: O(1)
  for i = 1 to A.length do      Ciclo for: O(n)
    A[i] = a + 1                Assegnazioni: O(1)
  for i = 1 to A.length do      Ciclo for doppio: O(n * n)
   while (j < A.length) do       → il while è come il for
     A[i] = a + i*j; j++;       Assegnazione: O(1)
  a = -1*a;                     Assegnazione: O(1)
}
```

Le assegnazioni non contano perché si assume che siano veloci e che avvengano in tempo costante. Quando ho dei cicli for (il ragionamento è uguale per i while) devo analizzare quanti cicli fanno. Nel codice sopra le parti di rilievo sono solo i cicli.

- `CodiceDiProva` ha complessità $O(n^2)$

Trascurando gli O(1) sono di interesse l'O(n) del primo for e l'$O(n^2)$ del secondo. La complessità totale dell'algoritmo sarebbe $O(n^2 + n)$ perché si fa la sommatoria di tutti quanti i costi di ogni operazione. Tuttavia è più giusto scrivere $O(n^2)$ perché si considera solo l'esponente più grande dato che l'n è di ordine più piccolo rispetto all'n^2 e quindi è ignorabile.

Caso ricorsivo.

Un algoritmo ricorsivo ha sempre un caso base, utile a fermare le chiamate alla funzione, ed un caso ricorsivo che accumula un tot di chiamate alla funzione stessa. Le funzioni ricorsive sono associate naturalmente alle equazioni di ricorrenza perché ne descrivono in modo naturale la struttura.

$$T(n) = \begin{cases} \Theta(1), & n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), & n > 1 \end{cases}$$

La soluzione di questa equazione di ricorrenza è $T(n) = \Theta(n*\log(n))$

L'equazione sopra descrive l'andamento di Merge Sort, un algoritmo di ordinamento divide et impera che verrà analizzato più avanti. Questa equazione ci dice che:

1. Merge Sort è una funzione ricorsiva perché è descritta da questa ricorrenza; è di tipo divide et impera

2. Se l'input avesse dimensione 1 (caso base) allora l'algoritmo verrebbe eseguito in tempo costante $\Theta(1)$
3. Se l'input avesse dimensione maggiore di 1 (caso ricorsivo) allora l'equazione ci direbbe che l'algoritmo dividerebbe n volte l'input in due parti grandi tanto quanto la metà di n

In sostanza l'equazione di ricorrenza descrive come l'algoritmo esegue la ricorsione e la soluzione dell'equazione ci dice la complessità. In Merge sort Θ(n*log(n)) viene fuori perché è la soluzione dell'equazione di ricorrenza.

Metodo di sostituzione.

Questo metodo serve a risolvere un'equazione di ricorrenza ipotizzando di sapere la soluzione e dimostrando che abbiamo ragione. Quindi data ad esempio la ricorrenza T(n), si ipotizza (o ancora meglio viene già dato) che sia un O(qualcosa) e poi si cerca di trovare una costante c che soddisfi le seguenti regole.

Cosa devo dimostrare	Che calcoli devo fare
$T(n) = O(\ f(n)\ )$	$T(n) \leq d \cdot f(n)$
$T(n) = \Omega(\ f(n)\ )$	$T(n) \geq d \cdot f(n)$
$T(n) = \Theta(\ f(n)\ )$	$T(n) \leq \geq d \cdot f(n)$

Nell'ultimo caso il simbolo $\leq \geq$ indica che bisogna fare contemporaneamente le due disequazioni maggiore uguale e minore uguale. In pratica basta fare così:

1. Dentro alle parentesi di T() mettere la $f(n)$ calcolata in quello che stava dentro a T e sostituire la lettera T con una costante arbitraria d. (Vedere esempi)
2. Porre tutto quanto minore e/o maggiore uguale a $d \cdot f(n)$
3. Risolvere la disequazione ricavando un valore per d

Ovviamente la costante può essere anche diversa da d; nel caso $\Theta(\ f(n)\ )$ è meglio scegliere nomi diversi per le costanti. Facciamo alcuni esempi.

Esempio 1

Dimostrare che $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + n$ ha soluzione $T(n) = \Theta(n)$.

Soluzione

Siccome ho il Θ devo fare le due disequazioni e dimostrare che esistono due costanti che verificano la veridicità dell'ipotesi $\Theta(n)$.

Caso $T(n) \leq\ d \cdot f(n)$

1. $d\left(\frac{n}{2}\right) + d\left(\frac{n}{4}\right) + n$
2. $d\left(\frac{n}{2}\right) + d\left(\frac{n}{4}\right) + n\ \leq dn$
3. Faccio i conti e risolvo

Caso $T(n) \geq c \cdot f(n)$

1. $c\left(\frac{n}{2}\right) + c\left(\frac{n}{4}\right) + n$
2. $c\left(\frac{n}{2}\right) + c\left(\frac{n}{4}\right) + n\ \geq cn$
3. Faccio i conti e risolvo

$$n\left(\frac{d}{2}+\frac{d}{4}+1\right)\leq dn$$
$$\left(\frac{d}{2}+\frac{d}{4}+1\right)\leq d$$
$$\frac{d}{2}+\frac{d}{4}-d\leq -1$$
$$-\frac{d}{4}\leq -1\rightarrow d\ \geq 4$$

$$n\left(\frac{c}{2}+\frac{c}{4}+1\right)\geq cn$$
$$\left(\frac{c}{2}+\frac{c}{4}+1\right)\geq c$$
$$\frac{c}{2}+\frac{c}{4}-c\geq -1$$
$$-\frac{c}{4}\geq -1\rightarrow c\ \leq 4$$

Ho trovato un valore per c e d costanti quindi l'assunzione che quell'equazione abbia soluzione $\Theta(n)$ è vera. Dato che $f(n)=n$ mettendo la n dentro alla T il risultato non cambia, ovvero $T\left(\frac{n}{2}\right)$ resta $T\left(\frac{n}{2}\right)$. Se avessi avuto $f(n)=n^2$ allora $T\left(\frac{n}{2}\right)$ sarebbe diventato un $T\left(\frac{n^2}{4}\right)$. In altre parole, devo mettere dentro alla funzione $f(n)$ quello che sta dentro alla T.

Esempio 2

Dimostrare che $T(n)=2T\left(\frac{n}{2}\right)+6n$ ha soluzione $T(n)=O(n*logn)$.

Soluzione

Siccome ho O devo fare una sola disequazione e verificare se esiste una costante d che mostra la veridicità dell'ipotesi.

$$\text{Caso } T(n)\leq\ d\cdot f(n)$$

1. $2d\left(\frac{n}{2}*log\left(\frac{n}{2}\right)\right)+6n$

2. $2d\left(\frac{n}{2}*log\left(\frac{n}{2}\right)\right)+6n\ \leq d*n\,log(n)$

3. Faccio i conti e risolvo

$$2d\left(\frac{n}{2}*log\left(\frac{n}{2}\right)\right)+6n\ \leq d*n\,log(n)$$
$$d\left(n*log\left(\frac{n}{2}\right)\right)+6n\ \leq d*n\,log(n)$$
$$d(\,n*\log(n)\ -n*\log(2))+6n\ \leq d*n\,log(n)$$
$$d\,n*\log(n)\ -n\,(d*\log(2)-6)\ \leq d*n\,log(n)$$
$$-n\,(d*\log(2)-6)\ \leq 0$$
$$d*\log(2)-6\ \geq 0$$
$$d\geq\frac{6}{\log(2)}$$

Ho trovato un valore per d costante quindi l'assunzione che quell'equazione abbia soluzione $O(n*\log(n))$ è vera. Notare che siccome è richiesta la verifica

di un n*log(n) è stata inserita la funzione dentro all'equazione di ricorrenza. Quindi dato $2T\left(\frac{n}{2}\right) + 6n$ se avessi avuto da verificare altro:

Cosa verificare	Come diventa l'equazione
$O(n)$	$2d\left(\frac{n}{2}\right) + 6n$
$O(n * logn)$	$2d\left(\frac{n}{2} * log\left(\frac{n}{2}\right)\right) + 6n$
$O(logn)$	$2d\left(log\left(\frac{n}{2}\right)\right) + 6n$

Dovrebbe essere chiaro quanto detto nel punto 1 della lista che spiega cosa fare. Intanto si mette la costante al posto della T. Poi va scritto dentro alle parentesi la funzione che prende come argomento l'argomento di T.

Ho trovato una costante d per $O(n * logn)$ ma non è detto che tale costante possa esserci anche per altre funzioni tipo $O(n)$.

Master theorem.

Metodo alternativo a quello di sostituzione e non infallibile. Risolve un sottoinsieme di equazioni di ricorrenza che hanno forma:

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

Il teorema ci dice che il risultato finale sarà $f(n)$ o $n^{\log_b a}$ e per stabilire il vincitore si fa il limite del rapporto che tende ad infinito. Si distinguono tre casi:

Limite	Risultato	Soluzione
$\lim_{n\to\infty} \frac{f(n)}{n^{\log_b a}} = k$	$k = l$	$T(n) = \Theta\left(n^{\log_b a}\log(n)\right)$
	$k = 0$	$T(n) = \Theta\left(n^{\log_b a}\right)$ se $\exists k > 0 \mid \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a-\varepsilon}} = k$
	$k = \infty$	$T(n) = \Theta\left(f(n)\right)$ se $\exists k > 0 \mid \lim_{n\to\infty} \frac{f(n)}{n^{\log_b a-\varepsilon}} = k$ se $\exists k, 0 < k < 1 \mid a\,f\left(\frac{n}{b}\right) \leq kf(n)$

Nella tabella $k = l$ vuol dire che il risultato del limite è un numero l positivo maggiore di 0. Il master theorem non è impeccabile perché in alcuni casi fallisce e occorre fare uso del metodo di sostituzione. Ecco alcuni esempi.

Esempio 1

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

Soluzione

$$\lim_{n\to\infty} \frac{6n}{n^1} = 6$$

Si ha che $f(n)$ = 6n e $n^{\log_b a}$ diventa $n^{\log_2 2}$ dato che a = b = 1. Facendo il limite la n si semplifica ed il risultato è 6. Mi trovo nel primo caso dove $k = l$ quindi T(n) risulta essere un $\Theta\left(n^{\log_b a} \log(n)\right)$. Sostituendo:

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

Notare che lo stesso esercizio è stato risolto anche prima usando il metodo di sostituzione dimostrando che era un $n \log(n)$. Tuttavia è meglio usare il master theorem quando possibile perché più facile ed evita la difficoltà di dover intuire la soluzione per poi applicare la sostituzione (che non è sempre facile).

Esempio 2

Risolvere $T(n) = 5T\left(\frac{n}{2}\right) + 2n^2 + n * log(n)$

Soluzione

$$\lim_{n\to\infty} \frac{2n^2}{n^{\log_2 5}} = 0$$

Si ha che $f(n) = 2n^2$ poiché tengo conto solo dei termini con grado maggiore ed n^2 cresce più velocemente di $n * log(n)$ all'infinito; quindi ignoro la seconda parte. Poi ho a = 5, b = 2 ed ottengo $n^{\log_2 5}$ che è di ordine superiore rispetto ad n^2. Infatti $\log_2 5$ è più di due quindi il denominatore "batte" il numeratore ed il risultato è 0. Sono nel secondo caso:

$$T(n) = \Theta\left(n^{\log_2 5}\right)$$

Devo ora trovare, se esiste, un epsilon che soddisfi la condizione di esistenza (quella del *se* nella tabella). Il $\log_2 5$ è sicuramente più piccolo di 3 dato che $\log_2 8 = 3$ e $\log_2 5 < \log_2 8$. Possiamo quindi concludere che

$$0 < \varepsilon < 3 - \log_2 5$$

soddisfa le condizioni ed ε è sufficientemente piccolo e maggiore di zero. Basta quindi trovare un epsilon che solitamente sta compreso fra 0 e $x - \log_b a$ dove x è un numero (preferibilmente intero) che sicuramente è maggiore di $\log_b a$.

Esempio 3

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

Soluzione

$$\lim_{n\to\infty} \frac{n^3 \log(n)}{n^{\log_2 5}} = \infty$$

Il numeratore è di ordine 3 grazie ad n^3 mentre il denominatore è di ordine inferiore a 3 sicuramente dato che $\log_2 5 < 3$. Vince il numeratore e quindi il limite va ad infinito; sono nel terzo caso:

$$T(n) = \Theta(\, n^3 \log(n)\,)$$

Devo ora trovare, se esiste, un epsilon che soddisfi la condizione di esistenza (quella del *se* nella tabella). Il $\log_2 5$ è sicuramente più piccolo di 3 dato che $\log_2 8 = 3$ e $\log_2 5 < \log_2 8$. Possiamo quindi concludere che

$$0 < \varepsilon < 3 - \log_2 5$$

soddisfa le condizioni ed ε è sufficientemente piccolo e maggiore di zero. Come ultimo step devo verificare la regolarità risolvendo la disequazione e trovando un k compreso fra 0 ed 1.

$$5\left(\frac{n}{2}\right)^3 * \log\left(\frac{n}{2}\right) \le kn^3 * \log(n)$$

Trucco: siccome è una disequazione posso maggiorare le equazioni; sappiamo che $\log\left(\frac{n}{2}\right) \le \log(n)$ quindi posso scrivere:

$$5\left(\frac{n}{2}\right)^3 * \log\left(\frac{n}{2}\right)^3 \le 5\left(\frac{n}{2}\right)^3 * \log(n) \le kn^3 * \log(n)$$
$$5\left(\frac{n}{2}\right)^3 * \log(n) \le kn^3 * \log(n)$$

Risolvendola ottengo che $0 < k < 1$ perché il risultato sarebbe k < 5/8 quindi anche questa condizione è verificata. La soluzione è davvero $\Theta(\, n^3 \log(n)\,)$.

ALGORITMI DI ORDINAMENTO

Ci sono algoritmi di ordinamento che hanno complessità migliore di altri ma ciò non vuol dire che siano i migliori in ogni caso. A volte si possono combinare o preferire ad altri ma ognuno ha i suoi vantaggi.

Ad esempio: *Insertion Sort* è $O(n^2)$ e *Merge Sort* è O(n*log(n)) quindi chiaramente MS è il più performante dei due. Tuttavia per piccoli input (0 < n < 20 circa) IS potrebbe essere preferibile quindi si potrebbe adottare una strategia ibrida.

Insertion sort

Complessità $O(n^2)$. Si tratta di un algoritmo iterativo che prende in input un array A e modifica gli elementi al suo interno in modo da ordinarli. Vediamo come funziona:

- A = [12, 11, 13, 5, 6]
  Parto dal 12 (elemento A[1]) che è già ordinato di suo

- A = [12, 11, 13, 5, 6]
  Mi sposto a destra e l'11 diventa la chiave (elemento da spostare). Continuo a spostarlo a sinistra finché trovo un elemento che sia più grande di lui. Poi mi fermo e ottengo A = [11, 12, 13, 5, 6]

- A = [11, 12, 13, 5, 6]
  Mi sposto a destra e il 13 diventa la chiave (elemento da spostare). Continuo a spostarlo a sinistra finché trovo un elemento che sia più grande di lui. Il 13 è maggiore del 12 quindi è già al posto giusto e mi fermo subito.
  Ottengo A = [11, 12, 13, 5, 6]

- A = [11, 12, 13, 5, 6]
  Mi sposto a destra e il 5 diventa la chiave (elemento da spostare). Continuo a spostarlo a sinistra finché trovo un elemento che sia più grande di lui. Lo sposto fino a che arrivo all'inizio dell'array e, non essendoci più nulla, mi fermo.
  Ottengo A = [5, 11, 12, 13, 6]

- A = [5, 11, 12, 13, 6]
  Mi sposto a destra e il 6 diventa la chiave (elemento da spostare). Continuo a spostarlo a sinistra finché trovo un elemento che sia più grande di lui. Mi fermo prima del 5 ovviamente e poi, avendo analizzato tutti gli elementi, esco.
  Ottengo A = [5, 6, 11, 12, 13]

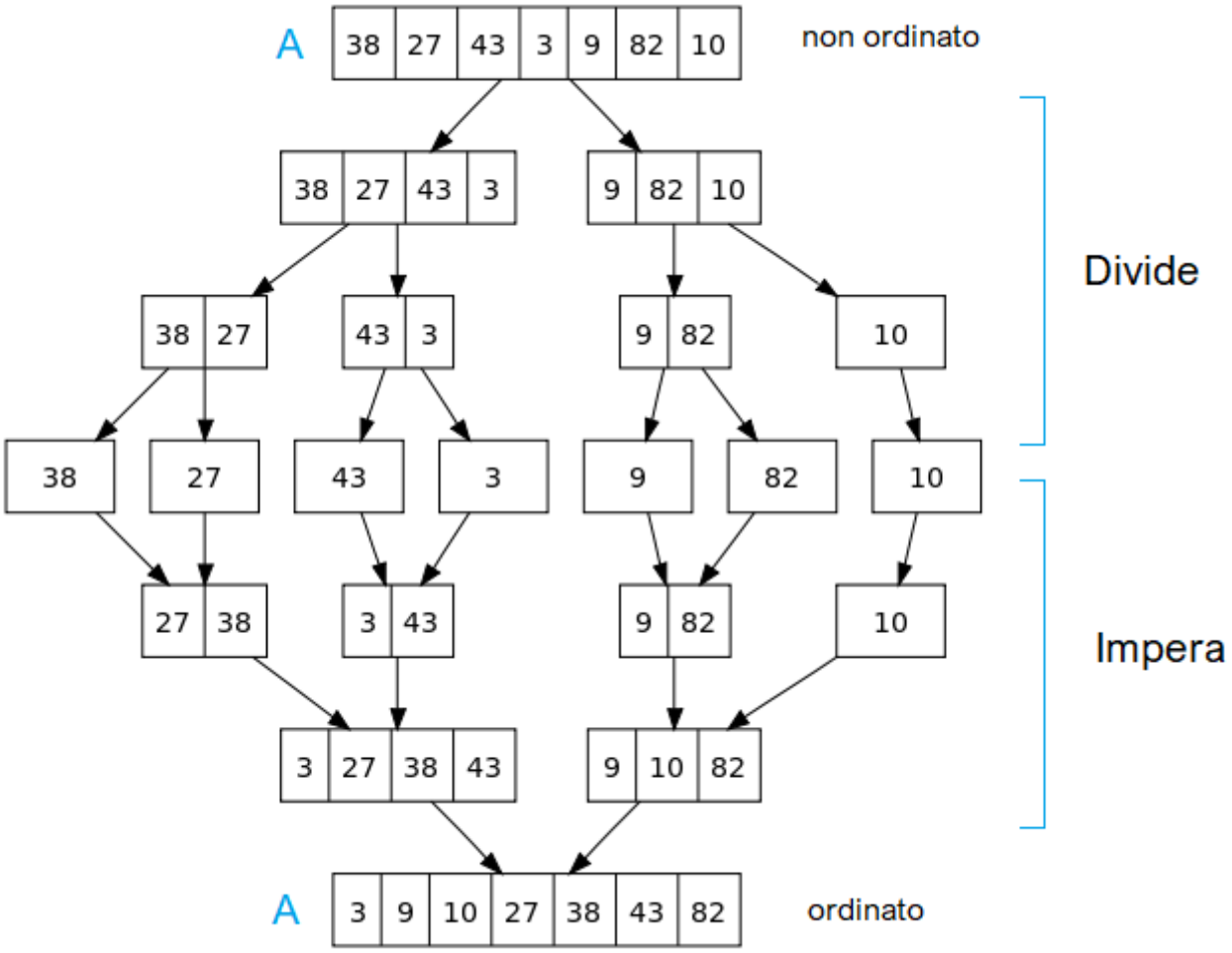
Questo algoritmo è molto semplice: si sposta da sinistra verso destra e guarda tutti gli elementi uno ad uno mettendo la chiave (l'elemento preso in considerazione) al posto giusto. Molti confronti, se l'array è ordinato decrescentemente ci mette tanto.

Codice	Descrizione
`InsertionSort(A) {`	
`for j := 2 to A.length do {`	→ Scorre l'array da sx a dx
`key := A[j];`	→ Elemento da spostare
`i := j-1;`	
`while ((i > 0) and (A[i] > key) {`	→ Sposta l'elemento a sinistra finché prima di lui non c'è nulla o c'è un numero più piccolo di lui
`A[i+1] := A[i];`	
`i--;`	
`}`	
`A[i+1] := key;`	→ Scrivi la chiave al posto giusto nell'array
`}`	
`}`	

L'array ordinato cresce da sinistra verso destra. Avendo un algoritmo iterativo che ha due cicli annidati che scorrono tutto l'array, come già visto, la complessità nel caso peggiore è $O(n^2)$.

Merge Sort

Complessità $\Theta(n * \log(n))$. Si tratta di un algoritmo ricorsivo divide et impera che prende in input un array A, l'inizio dell'array, la fine dell'array e lo ordina in modo crescente (come sempre). Vediamo come funziona:



Si parte dall'array A e si continua a dividerlo in due sotto array grandi tanto quanto la metà di A fino a che (a forza di dividere) si arriva ad avere tanti array di dimensione unitaria. Poi si procede a ritroso e si ricostruisce l'array nello stesso modo in cui era stato "tagliato" ma sta volta i tasselli vengono ordinati. Da notare:

- Se l'array avesse dimensione pari si avrebbero due parti simmetriche, cioè in caso di A.length = 8 si avrebbero due sotto array di dimensione 4

- Se l'array avesse dimensione dispari si otterrebbero due parti quasi simmetriche dove quella a sinistra ha un pezzo in più dell'altra. Come sopra, in caso per esempio di A.length = 7 si avrebbero due sotto array di dimensione rispettivamente 4 e 3.

- L'ordinamento avviene nella parte di ricostruzione, cioè al "ritorno" ricorsione.

Questo schema è alla base degli algoritmi divide et impera; c'è sempre una funzione che si chiama ricorsivamente spezzando l'array e poi alla fine si fanno le operazioni richieste. Ecco l'analisi dell'algoritmo.

Nota: p è l'indice di partenza dell'array, r è l'indice di fine dell'array, L ed R sono i due sotto array creati dalla divisione dell'array principale A.

Codice	Descrizione
`MergeSort(A, p, r) {`	
`  if (p < r) {`	→ Se c'è almeno un elemento
`  q = (p+r)/2;`	
`  MergeSort(A, p, q);`	→ Divido la prima metà
`  MergeSort(A, q+1, r);`	→ Divido la seconda metà
`  Merge(A, p, q, r);`	→ Unisco i vari sotto array creati dalla ricorsione e ordino gli elementi in A mettendoli al posto giusto.
`  }`	
`}`	

Codice	Descrizione
`Merge(A, p, q, r) {`	
`  n1 = q - p + 1;`	→ Dimens. primo sotto array
`  n2 = r - q;`	→ Dimens. secondo sotto array
`  for i := 1 to n1 do`	→ Riempio il primo sotto array con elementi di A
`  L[i] := A[p + i - 1];`	
`  for i := 1 to n2 do`	→ Riempio il secondo sotto array con elementi di A
`  R[i] := A[q + i];`	
`  L[n1 + 1] = x;`	→ Valore sentinella
`  L[n2 + 1] = x;`	
`  i = j = 1;`	
`  for k := p to r do {`	
`  if (L[i] <= R[j]) {`	→ Se l'elemento che sta nel sotto array L di sinistra è più piccolo dell'elemento che sta nel sotto array di destra vuol dire che è il minimo e deve andare in A[k]. Procedo così fino alla fine ed otterrò A che contiene la sequenza ordinata di elementi che stanno in L ed R.
`  A[k] := L[i];`	
`  i++;`	
`  } else {`	
`  A[k] := R[j];`	
`  j++;`	
`  }`	
`  }`	
`}`	

Notare che nell'ultimo ciclo quello che si fa è creare i due sotto array e poi usarli per confrontare ogni volta in due quale ci sia il minore. `A[k]` indica la posizione in cui va messo l'elemento più piccolo partendo da sinistra.

La complessità di merge è O(n) perché ho tre cicli ma nessuno è annidato quindi è tutto lineare. Tuttavia questa funzione viene chiamata all'interno di un'altra ricorsiva logaritmica e quindi moltiplicando le due funzioni si ottiene $O(n * \log(n))$.

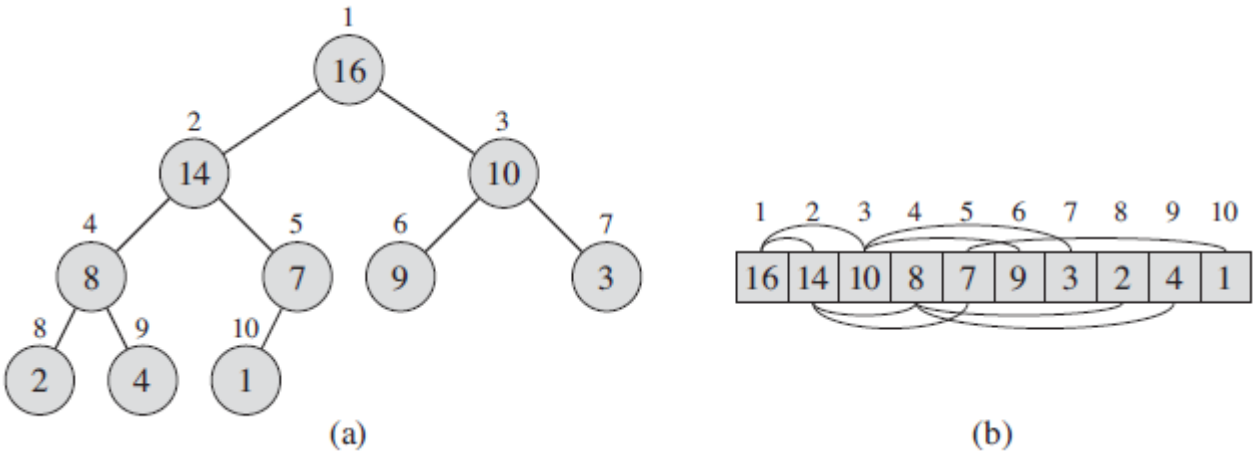
Dire $O(n * \log(n))$ è valido ma si può anche essere più precisi e dire che per il Merge sort la complessità è $\Theta(n * \log(n))$, ovvero sia nel caso peggiore che nel caso migliore l'andamento sarà sempre $n * \log(n)$.

Heap Sort

Complessità $\Theta(n * \log(n))$. Uno heap è una struttura dati che si può considerare come un albero binario quasi completo, ovvero un albero nel quale tutti i livelli sono riempiti completamente tranne l'ultimo. Le caratteristiche dello heap sono:

- L'ultimo livello incompleto si riempie partendo da sinistra
- L'albero viene rappresentato sotto forma di un array A che ha le proprietà *A.length* per la lunghezza di A e *A.heapsize* per la dimensione dello heap.
- Il primo elemento è A[1]

Notare che se ci sono 10 numeri nell'array ma *A.heapsize = 8* allora la struttura a heap sarà solo nei primi otto posti dell'array. A noi interessano i max heap ovvero degli heap con la caratteristica in più che un nodo ha l'attributo chiave maggiore o uguale dei suoi due figli.



Vediamo che *(a)* è uno heap perché soddisfa la descrizione nei 3 punti; in particolare è un max heap perché ogni nodo ha la chiave che è maggiore o uguale della chiave dei figli. Saranno utili queste tre funzioni da ricordare:

```
Parent(i)
  return i/2;
```

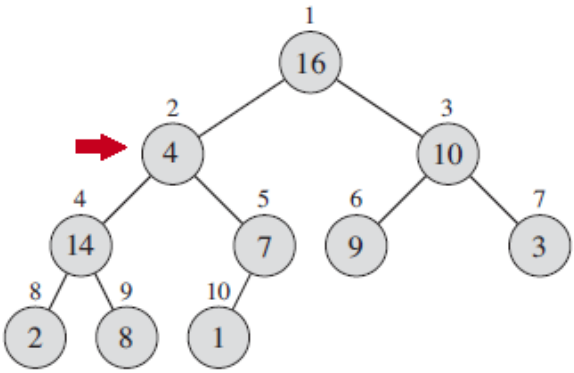
```
Left(i)
  return 2i;
```

```
Right(i)
  return 2i + 1;
```

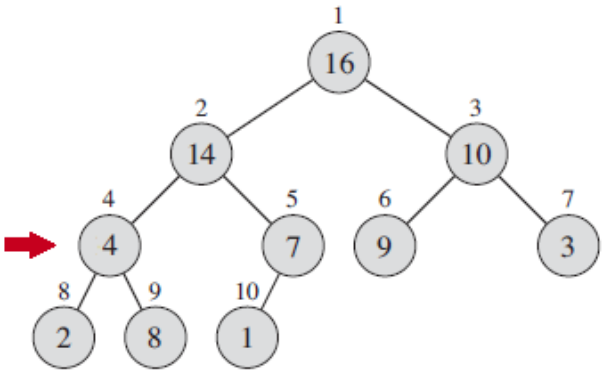
Supponendo che *i* sia un nodo allora la prima funzione ritorna il padre, le altre due ritornano i due nodi/foglie figli che stanno a sinistra e destra. Notare nel disegno che infatti *(b)* ha in posto pari i nodi più a sinistra mentre in posto dispari i nodi che stanno a destra.

Infine è importante notare che nell'array in *(b)* la lunghezza è 10 e nei primi 5 posti ci sono nodi, negli ultimi 5 le foglie. In altre parole, la caratteristica è che nella prima metà si trovano sempre nodi, nell'altra le foglie.

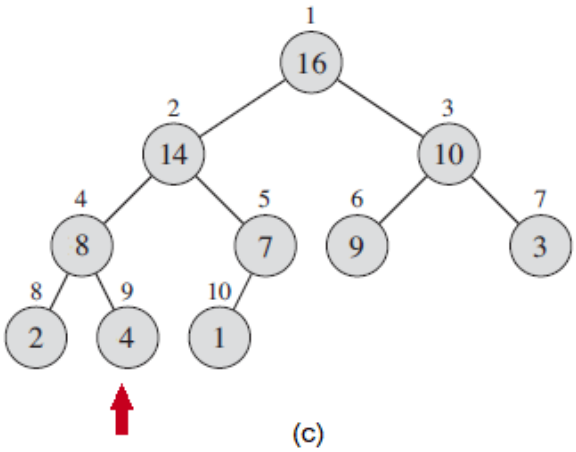
Vediamo ora la funzione `MaxHeapify(A, i)` che sistema un nodo al posto sbagliato, ovvero un nodo che NON è maggiore o uguale di entrambi i suoi figli. In pratica la funzione fa scendere questo nodo al posto giusto fino a che non soddisfa la proprietà di max heap.



(a)



(b)



(c)

- Il 4 è al posto sbagliato. Notare che i nodi sotto ad esso soddisfano la proprietà di max heap

- Il 4 (in posto i = 2) viene spostato in basso a sinistra ma non va ancora bene perchè c'è l'8 sotto che è >= 4

- Sposto il 4 in posto i = 9 ed ora ho finito perchè sotto non c'è altro. Ho aggiustato l'albero e ottenuto un max heap.

L'immagine descrive quello che fa `MaxHeapify(A,2)` ovvero sistemare il nodo in posto 2 in modo che l'albero torni ad essere un max heap. Ecco il codice commentato:

```
Max-Heapify(A,i) {                                  → i è la posizione
 l = Left(i)                                        del nodo da sistemare
 r = Right(i);
 if (l ≤ A.heapsize) and (A[l] ≥ A[i]) {            → vedi se la chiave
  max = l;                                          A[i] va spostata giù a
 } else { max = i }                                 sinistra
 if (r ≤ A.heapsize) and (A[r] ≥ A[max]) {          → vedi se la chiave
  max = r;                                          A[i] va spostata giù a
 }                                                  destra
 if (max != i) {                                    → se la chiave A[i]
  A[i] ←→ A[max]                                    non è maggiore dei due
  Max-heapify(A,max)                                figli a sx e dx,
 }                                                  spostala e richiama la
}                                                   funzione ancora.
```

Notare che il simbolo ←→ indica che si scambiano i valori che in A[i] e A[max]; viene richiamata ricorsivamente `Max-Heapify` per verificare che gli spostamenti appena fatti non abbiano creato altri problemi nello heap. Questa funzione è utile perché può creare un max heap dato un array qualsiasi.

```
BuildMaxHeap(A) {

 A.heapsize = A.length;                  → voglio che tutto l'array
                                         diventi un max heap
 for i := A.length/2 downto 1 {          → prendi tutti i nodi e mettili
  Max-Heapify(A, i);                     al posto giusto per creare uno
 }                                       heap con proprietà di max heap
}
```

Questa funzione permette, dato un array qualsiasi, di creare un max heap. In altre parole prende A, cambia di posto ai numeri dentro usando `Max-Heapify` e alla fine ottengo che A ha gli elementi disposti a max heap.

- `Max-Heapify`: dato un nodo, lo sposto al posto giusto
- `BuildMaxHeap`: prende tutti gli elementi dell'array e crea il max heap

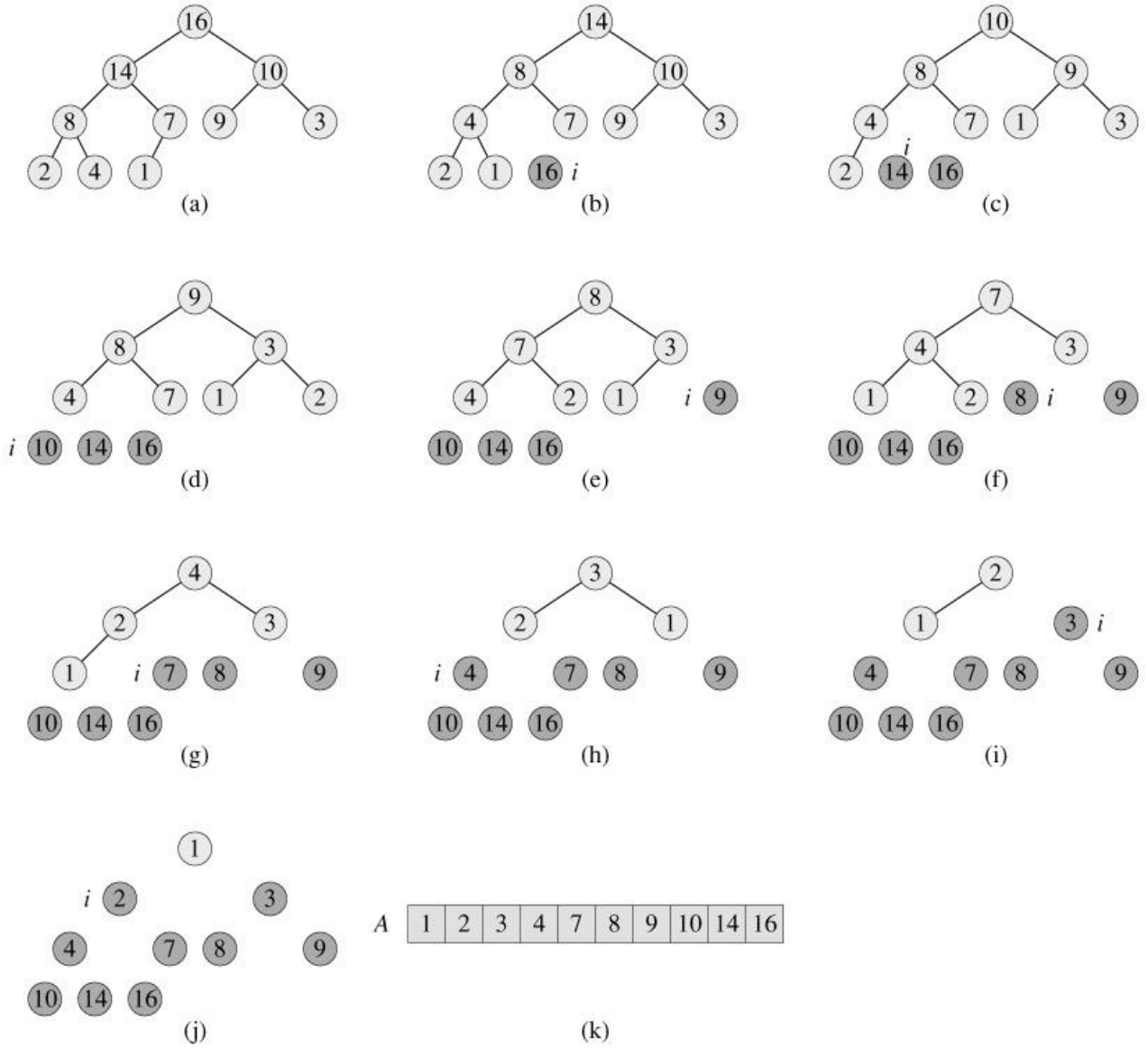
Importante notare che il ciclo parte dalla metà dell'array e lo scorre all'indietro fino all' 1. Prima abbiamo visto che in uno heap l'array ha da 1 a metà i nodi, da metà alla fine le foglie. Siccome `Max-Heapify` lavora sui nodi e NON sulle foglie inizio a chiamarlo da metà array, poi si arrangia lui a mettere al posto giusto i vari numeri. Le foglie sono già di per se un max heap (max heap con solo la radice) quindi non serve farci nulla. Finalmente, creiamo heap sort:

```
HeapSort(A) {
 BuildMaxHeap(A);                        → Costruisco un max heap perché
                                         devo lavorare su max heap
 for i := A.length downto 2 {            → sposto in fondo all'array il
  A[1] ←→ A[i];                          valore in posizione 1, riduco
  A.heapsize--;                          lo heap e sistemo i nodi grazie
  Max-Heapify(A, 1);                     a Max-Heapify.
 }
}
```

Dopo aver chiamato `BuildMaxHeap(A)` ci sarà in A[1] il massimo valore dell'array perché alla radice di un max heap c'è il massimo. Scambio l'elemento che stava in fondo con il massimo. Riduco di 1 la dimensione dello heap perché ho già sistemato un numero (quello massimo in fondo!) e chiamo `Max-Heapify` su A[1].

Avendo fatto lo scambio, ora in A[1] ho un numero a caso e devo sistemarlo al posto giusto del mio max heap, quindi chiamo `Max-Heapify`. Adesso mi trovo con un altro max heap che ha in A[1] l'elemento massimo e lo sposto in fondo prima di quello che avevo scambiato prima. Qui sotto un esempio che mostra come funziona heap sort (è già stato chiamato `BuildMaxHeap(A)`, qua mostra cosa si fa nel for).

Una buona implementazione di heap sort è efficace ma si preferisce merge sort; tuttavia dopo vedremo il Quicksort che può riverlarsi ancora meglio del merge. Si tratta sempre di algoritmi che hanno una crescita logaritmica o comunque col logaritmo (quindi non lineari $O(n)$).



Ogni volta che viene tolto un numero (in grigio scuro) si chiama `Max-Heapify` che ricostruisce lo heap il modo che poi il massimo possa essere ancora estratto e messo in coda dell'array.

Si possono utilizzare i max heap e sfruttare le loro caratteristiche per creare anche delle code con priorità, come quelle che utilizza lo *scheduler* dei processi nella CPU per stabilire quali mandare in esecuzione. Più grande è il numero di priorità e più importante sarà il processo da mandare in esecuzione; andranno in esecuzione prima i processi con valore maggiore.

- Restituisce l'elemento S con la chiave più grande

```
Maximum(A) {
 return A[1];
}
```

→ `La radice di un max heap è il valore più grande di tutta la struttura`

Viene eseguito in tempo $\Theta(1)$ ed è un'operazione veloce. Tuttavia a volte è necessario fare anche il *pop* ovvero trovare l'elemento con la chiave più grande, rimuoverlo e ritornare l'oggetto. `Maximum` si usa quando serve solo sapere la chiave più grande, `ExtractMax` si usa quando serve anche togliere dalla coda l'oggetto.

- Rimuove e restituisce l'elemento S con la chiave più grande

```
ExtractMax (A) {

 if (A.heapsize < 1) {
  error "underflow"
 }

 max = A[1];
 A[1] = A[A.heapsize;
 A.heapsize--;

 MaxHeapify(A, 1);
 return max;
}
```

→ La radice di un max heap è il valore più grande di tutta la struttura
→ controllo che lo heap contenga almeno un elemento, altrimenti sollevo una eccezione

→ Il massimo è la radice (come prima); avendola salvata in max posso toglierla dallo heap e diminuirne la dimensione

→ Mette A[1] al posto giusto nello heap

Viene eseguito in tempo $\Theta(\log n)$. Siccome va cancellato, scambio A[1] con l'ultimo elemento dello heap; riducendo di 1 lo heap scarto il massimo che era stato spostato in fondo. Ora mi ritrovo che in A[1] non c'è l'elemento con chiave più grande di tutti e quindi per spostarlo in giù al posto giusto chiamo il solito `MaxHeapify`.

- Aumenta il valore della chiave di x all'interno del max heap A

```
IncreaseKey (A, i, key) {

 if (key < A[i]) {
  error "key < A[i]"
 }

 A[i] = key;
 p = Parent(i);

 while (A[p] < A[i]) and (i > 1) {
  A[i] ←→ A[p];
  i = p;
 }
}
```

→ Array, elemento, valore da assegnare all'elemento
→ La chiave deve essere più grande del valore che aveva prima l'elemento i

→ Sposta in su nello heap l'elemento e fermati quando trovo che il padre è maggiore dell'elemento A[i]

Viene eseguito in tempo $\Theta(\log n)$. Dovendo aumentare la chiave ad un nodo del max heap potrebbe essere che questo nodo non si trovi più al posto giusto. NON posso usare `MaxHeapify` perché sposta i nodi verso il basso, in questo caso devo spostare il nodo verso l'alto perché la sua chiave aumenta (se la chiave diminuisse, allora userei si `MaxHeapify`). Il `while` sposta in su il nodo finché ha chiave maggiore del padre.

- Inserisce un nuovo oggetto di chiave `key` nel max heap

```
AddNewKey (A, key) {
 A.heapsize++;
 A[A.heapsize] = -∞;
 IincreaseKey(A, A.heapsize, key);
}
```

→ Aumento lo heap
→ Metto un valore temporaneo
→ Sposto in su la nuova chiave che voglio inserire

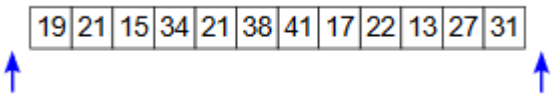
Viene eseguito in tempo $\Theta(log\ n)$. Aumento lo heap di 1 per fare spazio al nuovo elemento e ci inserisco un valore -∞ che è temporaneo. Infatti al posto suo ci andrà `key` che verrà inserito e messo al posto giusto da `IncreaseKey`.

Quick Sort

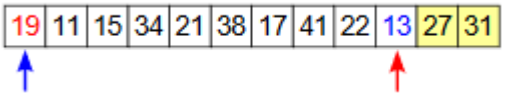
Complessità $\Theta(n * \log(n))$. Si tratta di un algoritmo di ordinamento divide et impera che ordina sul posto e utilizza dei sotto array di supporto. L'idea è: scegli un elemento dell'array (detto *pivot*) e partiziona in due sotto array mettendo a sinistra gli elementi minori uguali del pivot e a destra quelli maggiori. Poi ordina le due parti ricorsivamente. Il risultato sarà l'array A ordinato crescentemente.

```
QuickSort(A, p, r) {          → Array, indice di inizio array (1),
                              indice fine array (A.length)
 if (p < r)                   → se ci sono elementi da ordinare
  q = Partition(A, p, r);     → Partiziona l'array
  QuickSort(A, p, q-1);       → Ordina la parte a sinistra del pivot
  QuickSort(A, q+1, r);       → Ordina la parte a destra del pivot

}
```

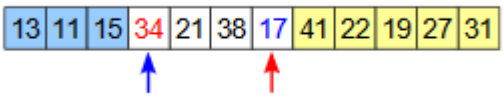
Il cuore di `QuickSort` è la funzione di partizione: dato il pivot *p* (un elemento dell'array) mette a sinistra di *p* tutti i numeri minori o uguali ad esso e alla destra di *p* tutti gli elementi maggiori a lui. Vediamo ora la funzione `Partition` in un esempio che mostra come funziona.



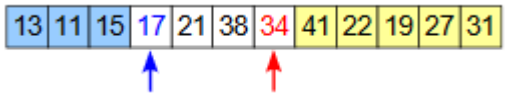
Supponiamo che questo sia A e che il pivot sia 17; ho due indici *i* e *j* rappresentati da due frecce blu. Quello a sinistra scorre verso destra fino a che non trova un elemento maggiore al pivot (= 17) mentre quello a destra scorre indietro fino a che non trova un elemento minore o uguale del pivot.



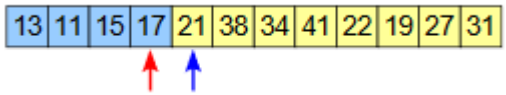
Gli indici scorrono finché possono. Il 19 è maggiore del pivot e mi fermo con *i*; invece *j* prosegue ma poi trova il 13 (che è minore di 17) quindi deve fermarsi. Ora che sono fermi tutti e due gli indici scambio i due elementi che bloccano, ovvero scambio il 13 col 19. Poi proseguo.



Come prima gli indici scorrono e poi si bloccano quando trovano a sinistra un numero maggiore del 17 e a destra uno minore o uguale. Li scambio e poi proseguo. Vado avanti fino a che i due indici non si "toccano", cioè fino a che non arrivano tutti e due all'elemento *pivot* 17.



Alla fine otterrò un array che ha a destra del 17 i numeri maggiori di esso, a sinistra quelli più piccoli. Notare che l'array non deve essere anche ordinato, la parte che ordina l'array viene dopo quando si fanno le chiamate ricorsive a `QuickSort`.



Vediamo il codice che sta dietro a `Partition` che restituisce un intero indicante la posizione nuova del pivot. Il codice assume che esso sia l'ultimo elemento dell'array. Guardando l'esempio dovrebbe essere facile capire il codice della funzione.

```
Partition(A, p, r) {
 i = p - 1;                    → indice di sinistra
 j = r + 1;                    → indice di destra
 x = A[p];                     → pivot è il primo elemento

 while (true) do {             → ciclo che continua all'infinito
   while (A[j] > x) { j--; }   → esce se A[j] <= x
   while (A[i] < x) { i++; }   → esce se A[i] >= x

   if (i < j) {                → fa lo scambio quando i due indici
    A[i] ←→ A[j]               non possono più proseguire E non si
   } else {                    sono ancora incontrati. Quando i e j
    break;                     si incontrano (i = j oppure i > j)
   }                           basta, ho finito.
 }

 return j;                     → ritorna la posizione del pivot.
}
```

Quindi chiamando ricorsivamente `QuickSort` la funzione `Partition` continua a modificare l'array di modo che il pivot stia nel "mezzo" fra quelli minori e maggiori di lui. Alla fine si otterrà un array ordinato.

Notare che la scelta del pivot come ultimo elemento o come primo non risulta essere sempre ottimale, un buon fix del problema è quello di scegliere il pivot *casualmente* in un range che sta fra *p* ed *r*.

Prima	Dopo
`x = A[p];`	`x = A[Random(p, r)];`

Counting Sort

Complessità $\Theta(n)$. Per applicare questo algoritmo bisogna sapere a priori il range di valori $0 \ldots k$ in cui sono compresi i numeri da ordinare (si assume anche che k sia maggiore di 0 altrimenti non avrebbe senso fare l'ordinamento). Supponiamo di avere come riferimento un array A che contiene valori compresi fra 0 e 5.

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

Utilizzando A da riferimento facciamo un esempio per capire come funziona counting sort. Si parte creando un array C di dimensione k (quindi 5, ecco a cosa serve sapere a priori il range di valori che contiene A) dove ogni cella dell'array memorizza quante volte si trova in A il numero di indice *i*.

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

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

In C troviamo che la cella 0 ha il numero 2 perché in A compare due volte lo zero; la cella 3 contiene 3 perché in A il numero tre compare tre volte. Quindi gli indici di C indicano i numeri che ci sono in A e il valore che sta nella cella di un certo indice dice quante volte quel numero compare dentro ad A.

Adesso modifico C in modo che nelle celle ci sia il valore che indica quanti numeri minori o uguali di *i* ci sono. Questo mi sarà fondamentale per capire dove andare a mettere i numeri in A per ordinarlo.

Prima

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

Dopo

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

Per calcolare il nuovo C basta sommare a due a due i valori del vecchio C con C[i] = C[i] + C[i-1]. C[3] = 7 vuol dire che ci saranno 7 numeri minori o uguali a tre, quindi il tre andrà messo in posto 7. Il risultato finale sarà:

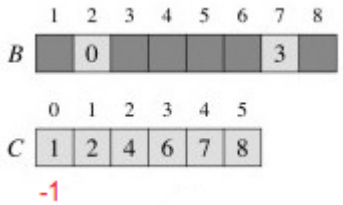
	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

L'array è stato ordinato basandosi su C perché in esso sono salvate le posizioni in cui andare a mettere i numeri. In particolare:

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8
				-1		

Prendiamo l'elemento che sta in C[3]; per costruzione (visto prima) la cella C[3] mi dice che ci sono 7 numeri in B minori o uguali del 3. Quindi idealmente B sarà un array tipo x x x x x x 3 y con x ≤ 3, y > 3. Il tre andrà il posizione 7 e poi decremento C[3] di 1 perché, avendolo messo giù in B, ho un elemento in meno da analizzare.



Analogamente il C[0] = 2 dice che ci sono due elementi minori o uguali a 2, quindi B sicuramente avrà x 2 y y y y y y con con x ≤ 2, y > 2. Proseguo così fino a che C non ha tutti zeri nelle celle così so di avere posizionato tutti gli elementi. Riassumendo:

1. Scrivo in C quante volte appare un numero dell'array basandomi sull'indice
2. Modifico C segnandomi quanti numeri ci sono prima dell'elemento *i*
3. Uso C che mi dice in quale cella di B mettere il numero di posto *i* in C

Notare che l'algoritmo prende in input l'array A da ordinare ed il range *k* di valori (in questo caso 0 . . 5) e ritorna il risultato in un array B. Ecco lo pseudocodice di Counting Sort con i relativi commenti:

```
CountingSort(A, B, k)                    → Array da ordinare, array dove
                                         salvare i risultati, range
 C = new array[0..k];
 for i = 0 to k do {                     → Preparo C riempiendolo di
  C[i] = 0;                              tutti zeri
 }
 for j = 1 to A.length do {              → Conto quanti elementi uguali
  C[A[j]]++;                             ci sono in A e lo salvo in C
 }
 for i = 1 to k do {                     → Conto quanti elementi minori o
  C[i] = C[i] + C[i - 1];                uguali di i ci sono in C
 }
 for j = A.length downto 1 do {          → Metto dentro all'array B i
  B[C[A[j]]] = A[j];                     numeri guardando gli indici ed i
  C[A[j]]--;                             valori che ho messo in C.
 }
}
```

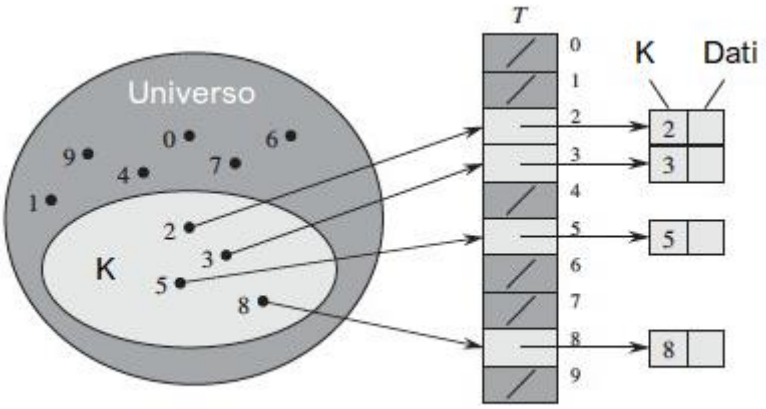
Questo algoritmo funziona perché si fa una assunzione sull'input, ovvero quella che si sanno a priori i numeri che si andranno ad ordinare. Sapendo il range di valori posso creare un array C di dimensione tot e poi fare tutti i conti. Per comodità è stato scelto di usare un array 0-indexed poiché è necessario sapere quante volte compare nell'array il numero 0.

TABELLE HASH

Una tavola hash è una generalizzazione della nozione di array alla C++, cioè di una sequenza di celle nelle quali posso salvare delle chiavi. Ho bisogno che siano veloci alcune operazioni come inserimento, ricerca e cancellazione. Queste saranno utili per implementare ed usare bene i dizionari, ovvero array che contengono coppie formate da (chiave, valore) che sono identificate dalla chiave.

Indirizzamento diretto

Supponiamo di avere un universo di chiavi (insieme di chiavi) piccolo che contiene valori da 0 a 9. Si può usare un array, o tavola ad indirizzamento diretto, dove ogni cella dell'array corrisponde ad una chiave. In questo modo le coppie (chiave, valore) possono essere indicizzate in base alla posizione che hanno nell'array.



In *Universo* ci sono tutte le chiavi, in *K* ci sono solo le chiavi utilizzate e T è un array che va da 0 a 9 perché deve contenere tutte le chiavi di *U*. L'array ha celle grigio scure per indicare chiavi non usate, celle grigio chiare per indicare celle utilizzate.

Questo metodo ha di vantaggioso che la chiave 2 si trova in posto 2, la chiave 3 in posto 3 e così via. Le 3 operazioni viste prima sono TUTTE eseguite in tempo O(1).

```
Search(T, k) {
 return T[k];
}
```

```
Insert(T, x) {
 T[x.key] := x;
}
```

```
Delete(T, k) {
 T[k] := nil;
}
```

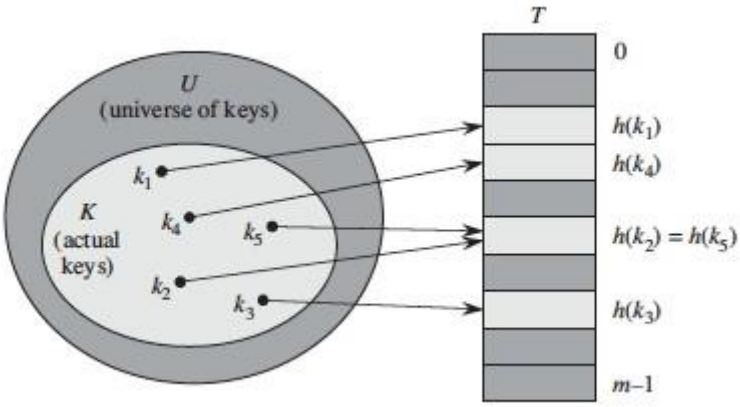
Ci si rende subito conto che se l'universo delle chiavi fosse troppo grande ci sarebbero delle difficoltà di memorizzazione (la RAM non ha dimensione infinita) ed anche la ricerca sarebbe molto dispendiosa. Inoltre ci sarebbe un grande spreco se su 10000 chiavi solo 6 verrebbero effettivamente usate.

Indirizzamento con tavole Hash

Quando l'insieme K di chiavi da memorizzare è molto più piccolo dell'universo U una buona strategia è quella di usare le tavole hash. Esse riducono di molto la dimensione dell'array da usare (grande tanto quanto K) e ricercano MEDIAMENTE in tempo O(1) un elemento. Notare la differenza:

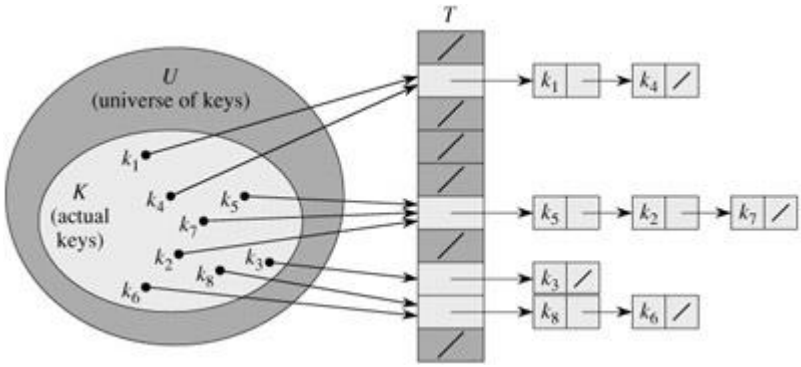
- Indirizzamento diretto: un elemento di chiave *k* è memorizzato in cella *k*
- Indirizzamento con hashing: un elemento di chiave k è memorizzato in cella *h(k)* dove *h* è una funzione.

In pratica adesso l'indice dell'array non è più dato dal numero ma dal risultato di una funzione hash. Se prima facevo T[3] ora devo "hashare" l'indice per sapere dove si trova l'elemento quindi devo fare T[*h(3)*].



Ora l'array non ha più dimensione |U| ma ha dimensione |m| cioè è grande tanto quanto il numero di chiavi che vengono realmente utilizzate. La funzione hash *h(x)* tuttavia può generare due output uguali e quindi può essere che due chiavi diverse vengano assegnate ad una stessa posizione. Si tratta di collisione e ci sono due modi per risolverle.

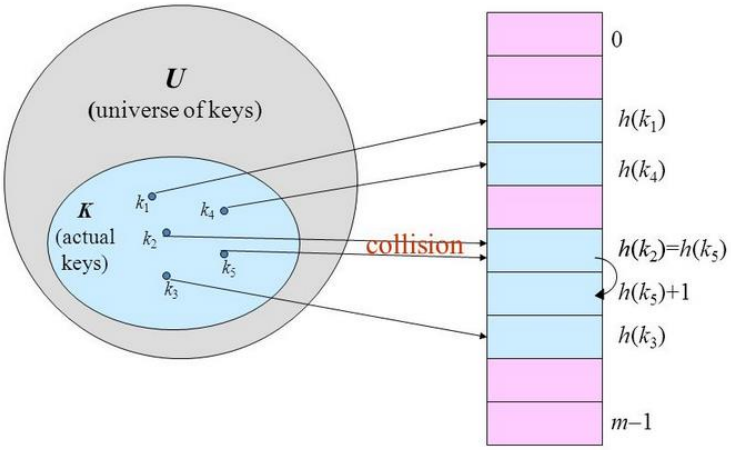
1. Concatenamento (chained hash)
   Il primo modo di risolvere le collisioni è semplice ed usa liste doppiamente concatenate; andrebbero bene anche quelle singolarmente concatenate ma sono più lente. Il difetto è che la ricerca di un elemento in una lista richiede tempo O(n) perché bisogna scorrerla tutta per cercare una determinata chiave.



   Tutti gli elementi che vengono associati da *h(x)* alla stessa cella sono messi in una lista doppiamente concatenata. In altre parole, quando si verifica una collisione (due chiavi diverse sono assegnate allo stesso slot) vengono messe in una lista. Nell'immagine sopra si vede che k_5, k_2 e k_7 sono chiavi con hash uguale e quindi sono messe in una lista concatenata.

2. Indirizzamento aperto (open addressing)
   Qui tutti gli elementi sono inseriti direttamente nella tavola hash (nell'array) quindi le celle o contengono l'elemento da salvare oppure nil che sta ad indicare il nulla. Potrebbe essere che la tavola si riempie in fretta e quindi verrà sollevata un'eccezione.

Open addressing



Per inserire un valore si usa la funzione di hash *h(x)* come al solito; se si verifica una collisione allora si ispezionano in successione le celle dopo quella che non va bene fino a che si arriva ad una libera. Nel disegno:

- Si inserisce k_2 in posto 5 (dato che $h(k_2) = 5$)
- Si tenta di inserire k_5 in posto 5 (dato che $h(k_5) = 5$) → collisione
- Si tenta di inserire k_5 in posto 6 (dato che $h(k_5)+1 = 6$) → ora va bene

Se anche il posto 6 fosse stato occupato allora l'algoritmo avrebbe continuato a fare +2, +3, +4 su $h(k_5)$ fino a trovare un posto libero. In pratica: quando c'è una collisione, si prosegue in sequenza fino a che non si trova un posto libero. La tecnica si chiama ispezione e si può fare in tre modi:

1. Ispezione lineare. Quella vista adesso, cioè si sposta sempre di una cella in basso alla ricerca di posto libero. La formula è $h(x) + i$.
2. Ispezione quadratica. Il concetto è uguale all'ispezione lineare ma al posto di spostarsi di 1 ci si sposta di c_1 e c_2 costanti arbitrarie. La formula adesso è $h(x) + c_1 i + c_2 i^2$.
3. Doppio hashing. Uno dei metodi migliori per l'ispezione che si basa su due hash con aritmetica modulare. La formula è $h(x, i) = (h_1(x) + i*h_2(x)) \bmod m$ dove h_1 e h_2 sono funzioni hash, m è la dimensione dell'array e mod è il resto della divisione. La *i* indica invece il "tentativo"; si parte sempre da i = 0 ma in caso di collisione (come visto nel disegno sopra) i incrementa di 1 fino a che non si trova uno spazio libero.

Una buona funzione hash soddisfa la condizione di *hashing uniforme semplice*: ogni chiave ha la stessa probabilità di essere inserita in una qualsiasi delle m celle della tavola. In pratica ogni chiave deve avere probabilità 1/m di finire in una delle varie celle ma non è così facile ottenere questo risultato. Le funzioni hash possono essere create secondo diversi metodi:

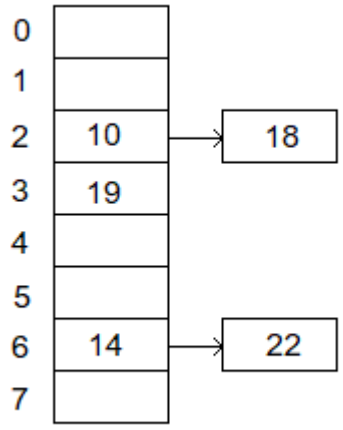
- Metodo del~~la moltiplicazione~~: $h(x) = k \bmod m$ Metodo di divisione questo qui!
- Metodo della moltiplicazione: $h(x) = m * (k*A \bmod 1)$ con $0 < A < 1$
- Hashing universale: viene scelta casualmente una *h(x)* fra un gruppo di $h(x)_n$

Ricordiamo che *m* è la dimensione della tavola hash e *mod* indica il resto della divisione. L'ultimo metodo è il più efficace: ogni volta viene scelta a caso una *h(x)* da un insieme di tante *h(x)* ben studiate per far si che ci siano meno collisioni possibili.

Esempio 1

Si consideri una tabella hash di dimensione $m = 8$ gestita con chaining e con una $h(x) = k \bmod m$. Si descriva l'inserimento delle chiavi: 14, 10, 22, 18, 19.

Soluzione



1. 14 mod 8 = 6
   Inserisco la chiave 14 in posto 6
2. 10 mod 8 = 2
   Inserisco la chiave 10 in posto 6
3. 22 mod 8 = 6 → collisione
   Inserisco la chiave 22 in posto 6 come secondo elemento della lista concatenata
4. 18 mod 8 = 2 → collisione
   Inserisco la chiave 18 in posto 2 come secondo elemento della lista concatenata
5. 19 mod 8 = 3

Ovviamente bisogna ricordare che *mod* è il resto della divisione quindi:
14 mod 8 = 6 perché 14/8 = 1 resto 6 (infatti 8/1 + 6 = 14)

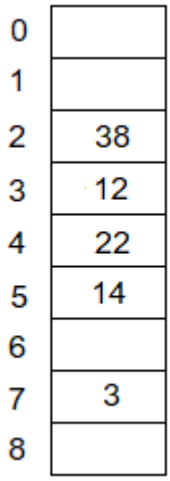
Esempio 2

Si consideri una tabella hash di dimensione $m = 9$ gestita con indirizzamento aperto e che si basa su $h_1(x) = k \bmod m$ e su $h_2(x) = 1 + k \bmod (m-2)$. Si descriva come avviene l'inserimento delle chiavi in sequenza: 12, 3, 22, 14, 38.

Soluzione

Le due funzioni hash sono $h_1(x) = k \bmod 9$ e su $h_2(x) = 1 + k \bmod 7$. Ricordiamo che $h(k, i) = (h_1(k) + i*h_2(k)) \bmod 9$. Dunque:

Attenzione qui:
Dopo una collisione, tutto il calcolo deve essere diviso per mod m oppure non viene giusto.
In questo caso (tutto) mod 9



1. h(12, 0) = (12 mod 9) + 0*(1 + 12 mod 7) = 3
   Inserisco la chiave 12 in posto 3
2. h(3, 0) = (3 mod 9) + 0*(1 + 3 mod 7) = 3 → collisione
   h(3, 1) = (3 mod 9) + 1*(1 + 3 mod 7) = 7
   Inserisco la chiave 3 in posto 7 perché prima ho avuto una collisione; ho incrementato i di 1
3. h(22, 0) = (22 mod 9) + 0*(1 + 12 mod 7) = 4
   Inserisco la chiave 22 in posto 4
4. h(14, 0) = (14 mod 9) + 0*(1 + 12 mod 7) = 5
   Inserisco la chiave 14 in posto 5
5. h(38, 0) = (38 mod 9) + 0*(1 + 12 mod 7) = 2

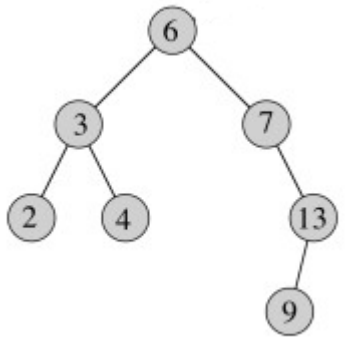
STRUTTURE AD ALBERO

Le due strutture principali che verranno analizzate sono gli alberi binari di ricerca, detti anche BST, e gli alberi rosso-neri, detti anche RBTree. Si tratta di strutture dati con specifiche differenti ma che comunque in generale supportano molte operazioni come ricerca, inserimento, cancellazione, massimi/minimi etc.

Alberi binari di ricerca (BST)

Si tratta di un albero con nodi *x* che hanno i seguenti campi: *x.key*, *x.left*, *x.right*, *x.p* che rappresentano rispettivamente la chiave, il figlio sinistro, il figlio destro e il padre del nodo. Le chiavi sono memorizzate in modo da rispettare la proprietà dei BST:

- Sia *x* un nodo di un albero binario di ricerca. Allora *x.left* punta ad un nodo *y* che ha campo *y.key* ≤ *x.key* e *x.right* punta ad un nodo z con campo *z.key* ≥ *x.key*.



In altre parole ogni nodo di un BST deve avere a sinistra un nodo con chiave minore rispetto a lui e a destra un nodo con chiave maggiore o uguale. Un nodo ovviamente può non avere qualcosa a sx o dx e in questo caso il puntatore è vuoto, nil.

→ Il nodo 6 ha a sinistra 3 (3 ≤ 6) e a destra 7 (7 ≥ 6)
→ Il nodo 13 ha a sinistra 9 (9 ≤ 13) e a destra non ha nessun figlio quindi x.right = nil.

I nodi dei BST verranno stampati in ordine infisso, cioè partendo da sinistra, salendo verso il nodo e poi tornando giù a destra. L'algoritmo è molto semplice:

```
Infisso(x) {
 if (x != nil) {                  → Se ci sono figli del nodo
  Infisso(x.left);                → Prima stampa a sinistra
  Stampa(x);
  Infisso(x.right);               → Poi stampa a destra
 }
}
```

Ricerca. Più interessanti sono le operazioni tipiche da fare su una struttura dati, quindi anche un BST non è da meno. Per quanto riguarda la ricerca di un nodo bisogna creare una funzione che prende in input la radice e la chiave da cercare.

```
Search(x, k) {
 if (x = nil) or (x = x.key)      → Se il nodo non c'è oppure se ho
  return x;                        trovato la chiave, mi fermo
 if (x.key < k)                   → Se la chiave del nodo è minore
  return Search(x.left, k);        di k vado a sinistra perché nei
 else                              BST a sinistra ci sono i nodi con
  return Search(x.right, k);       la chiave minore. Altrimenti vado
}                                  a sinistra.
```

Il codice mostra come viene sfruttata la caratteristica dei BST. Sappiamo che a sinistra trovo nodi con chiave minore o uguale e a destra quelli con chiave maggiore. Quindi al posto di scorrere tutti i nodi mi sposto a sx o dx in base al fatto che `x.key` sia maggiore uguale o minore di `k`.

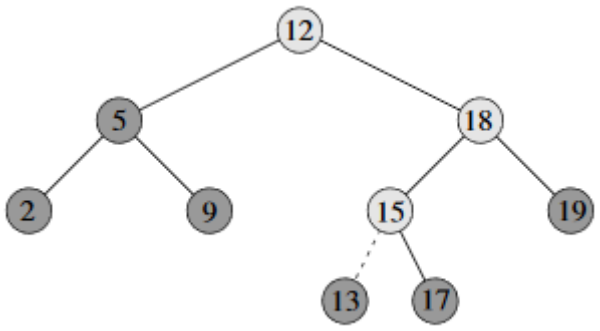
Massimo e Minimo. Si esplora l'albero dalla radice e si scende a sinistra o a destra in base alla chiave del nodo; ovviamente il parametro `x` è il nodo radice del BST:

```
Minimo(x) {
 while (x.left != nil) {
  x = x.left;
 }
 return x;
}
```

```
Massimo(x) {
 while (x.right != nil) {
  x = x.right;
 }
 return x;
}
```

Utilizziamo ancora la proprietà dei BST. Il minimo sarà per forza da qualche parte sulla sinistra, dove ho i nodi con chiave minore, e il massimo a destra, dove ho i nodi con chiave maggiore.

Inserimento. Per inserire un nodo usa una procedura simile a quella della ricerca. Si parte dalla radice e si scende in basso alla ricerca di un nodo che sia `NIL`; in questo modo so che posso inserire in modo sicuro il nodo.



In grigio chiaro è evidenziato il cammino che fa l'algoritmo per inserire il nuovo nodo (il 13, quello col tratteggio).

- Si parte dalla radice (12) e si guarda se la chiave del nodo da inserire è maggiore o minore delle chiavi che incontra e ci si sposta a destra o a sinistra
- Quando si trova un nodo che ha `nil` come puntatore al nodo si fa l'insert.

In pratica parto dalla radice e cerco un posto libero (= un nodo che abbia figlio `nil`) spostandomi a destra o a sinistra in base al fatto che la chiave del nodo da inserire sia maggiore o minore dei nodi che scorre (cosi rispetto la proprietà del BST). Si vede che il 13 è stato inserito dove c'è un posto libero e, siccome 13 < 15, sta alla sinistra del suo genitore.

Il codice `Tree-Insert` farà esattamente quello che è stato descritto nell'immagine. Si servirà in più di un nodo aggiuntivo y che servirà a ricordare il padre del nuovo nodo da inserire. Ci saranno infatti due nodi coinvolti:

- `x` scorre i nodi dell'albero fino a che non diventa `nil`
- `y` si tiene a mente il padre del nodo analizzato da

```
Tree-Insert(T, z) {
 y = nil;                          → nodo che ricorda il padre di x
 x = T.root;                       → nodo che scorre T e cerca posto
                                   libero (cerca un figlio nil)
 while (x != nil) {
  y = x;                           → salvo il padre di x
  if (z.key < x.key) {             → mi sposto a destra o a sinistra
   x = x.left;                     basandomi sul fatto che la chiave
  } else {                         del nuovo nodo z deve essere
   x = x.right;                    maggiore o minore del nodo x che
  }                                sto analizzando. Uscito dal
 }                                 while, x sarà nil e y sarà il
                                   nodo che farà da padre a z.
 z.p = y;                          → il padre di z è y

 if (y == nil) {                   → se y è nil vuol dire che non
  T.root = z;                      sono nemmeno entrato nel while,
 } else {                          quindi x = nil fin da subito e
  if (z.key < y.key) {             l'albero era vuoto. Quindi z sarà
   y.left = z;                     la radice. Se y non è nil allora
  } else {                         devo vedere se inserire z a
   y.right = z;                    destra o sinistra e, come sempre,
  }                                guardo la chiave se è > o <.
 }
```

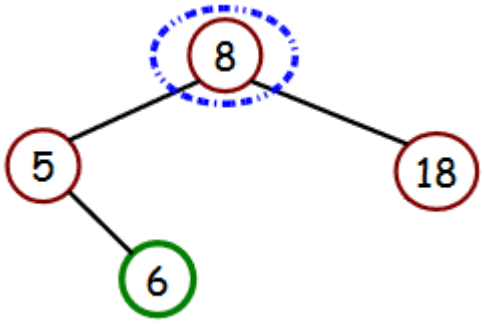
Quindi è importante capire che `x` fa l'esploratore in cerca del posto libero, `y` invece si ferma un nodo prima di `x`. Fermandosi 1 prima posso ricordare chi è il padre del nodo `nil`, ovvero sapere il padre del posto libero e di conseguenza sapere il padre del nodo `z` da inserire.

Cancellazione. Per eseguire questa procedura abbiamo bisogno di una funzione ausiliaria chiamata `Tree-Successor` che restituisce il successore di un nodo secondo l'ordine infisso. Supponiamo di usare la funzione `Infisso` e che il risultato sia:

Il successore di 3 è 4. Quindi `Tree-Successor` prende in input un nodo dell'albero (tipo il 3) e ritorna il successore di quel nodo secondo l'ordine infisso (ritorna il 4). Notare che `Infisso` stampa la chiave della radice nel mezzo tra la stampa dei valori nel sottoalbero sinistro e la stampa dei valori del sottoalbero destro. Ecco il codice:

```
Tree-Successor(x) {
 if (x.right != nil)               → se c'è un sottoalbero a destra,
  return Minimo(x.right);          il successore sarà il minimo.
 y = x.p;                          → altrimenti salvo il padre di x
 while ((y != nil)                 → risalgo l'albero partendo da x
        and (x == y.right)) {      finchè incontro un nodo che sia
  x = y; y = y.p;                  figlio sinistro di suo padre.
 }
 return y;
}
```

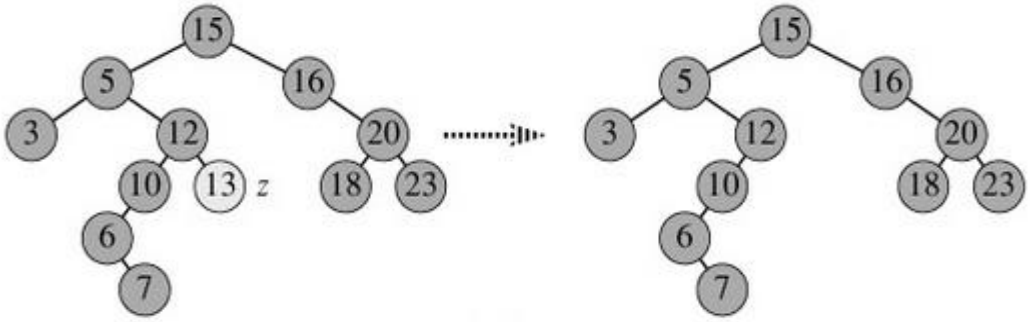
Dato un nodo, se esiste un sottoalbero a destra allora il successore sarà il minimo di quel sottoalbero (cioè il nodo più a sinistra del sottoalbero destro). Altrimenti se non c'è un sottoalbero destro, ma c'è un successore y, ritorno il primo antenato di x che trovo il cui figlio sinistro sia anche antenato di x



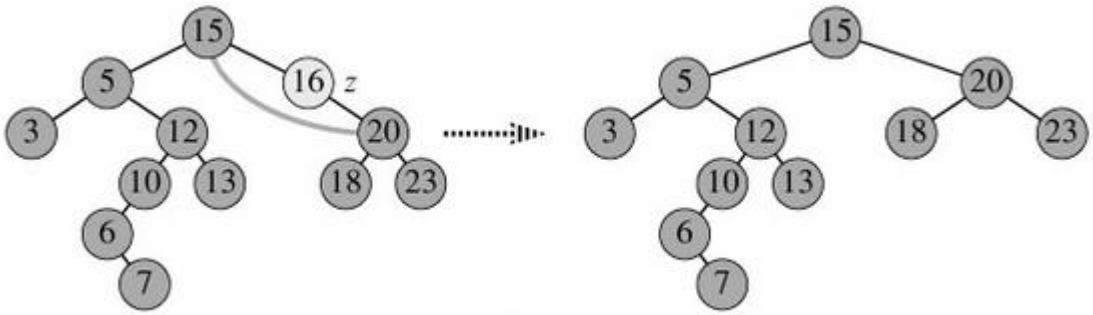
Cerchiamo il successore di 6 in questo BST con radice in 8. Dato che il nodo con 6 non ha figli a destra torno su.
Il risultato sarà 8 perché devo cercare un nodo che abbia un figlio sinistro che sia antenato di x (il 6). L'otto è antenato di 6, il 5 è antenato di 6 e il 5 è figlio sinistro di 8.

La cancellazione di un nodo `z` da un albero binario di ricerca considera tre casi base, dove uno di questi, il terzo, è un po' complicato.

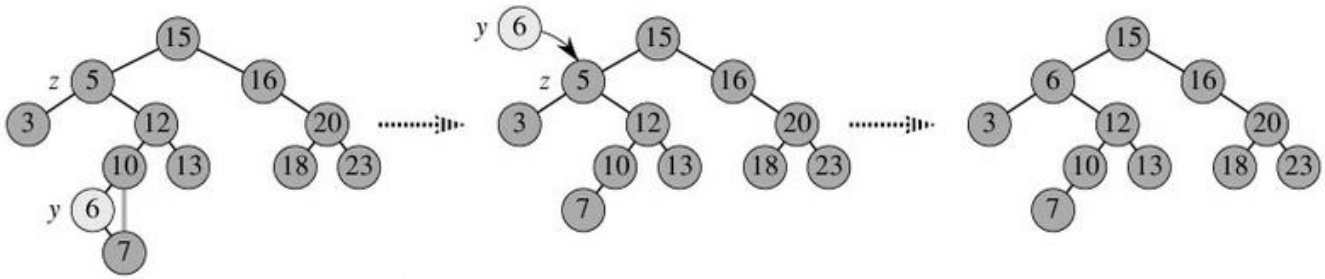
1. Se il nodo `z` da cancellare non ha figli, lo tolgo e modifico suo padre mettendo `nil` al posto di `z`. Tolgo il 13 e metto il `right` del nodo con 12 uguale a `nil`.



2. Se il nodo `z` da cancellare ha un solo figlio, cancello `z` e il figlio di `z` diventa figlio del padre del nodo che ho rimosso. Il nodo con 16 ha un solo figlio quindi il 20 diventa figlio 15 (il padre del nodo che ho rimosso, cioè il padre di 16.



3. Se il nodo `z` da cancellare ha due figli allora il codice è un po' un casino. Bisogna prendere il successore di `z`, che chiameremo `y`, e rimuoverlo dall'albero collegando il padre di `y` col figlio di `y`. Poi si sostituiscono la chiave ed i dati satellite di `z` con la chiave ed i dati satellite di `y`. Vedremo che sarà necessaria anche una funzione ausiliaria che sposta i sottoalberi.



Voglio cancellare il 5. Prendo il suo predecessore `y` (il 6) e lo tolgo attaccando il 7 al 10. Ora metto il 6 al posto del 5 e ho finito.

Il codice di `Tree-Delete` è complicato quindi viene proposto come funziona ma non è commentato nel dettaglio. Tuttavia una funzione utile che ritornerà anche dopo nella sezione degli alberi rosso-neri è la `Transplant`. Essa serve a spostare i sottoalberi in un BST; dato un nodo cambia il suo sottoalbero con un altro sottoalbero. In particolare la funzione sostituisce il sottoalbero con radice in `u` col sottoalbero radicato in `v`.

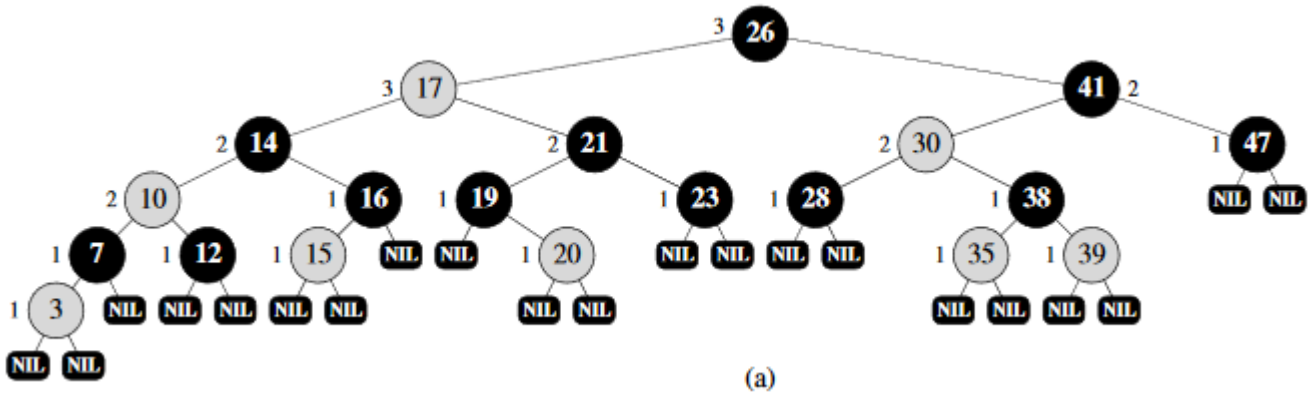
```
Transplant(T,u,v) {
 if (u.p == nil) {          → se u è radice (non ha padre) allora v
  T.root = v;               diventa la radice
 } else {
  if (u == u.p.left)        → se u è figlio destro o sinistro di un
   u.p.left = v;            nodo K, aggiorno i campi right o left di
  else                      K in modo da avere v come sottoalbero.
   u.p.right = v;           Quindi v diventa figlio destro o sinistro
 }                          del nodo K

 if (v != nil)              → ovviamente v può essere nil quindi nel
  v.p = u.p;                caso si aggiorna il puntantore al padre
}                           di v
```

Alberi rosso-neri (RBTree)

Si tratta di un BST con l'aggiunta che ogni nodo ha in più un bit per ricordare il colore: rosso o nero. Quindi un BRTree è formato nodi *x* che hanno i seguenti campi: *x.key*, *x.left*, *x.right*, *x.p*, *x.color* che rappresentano rispettivamente la chiave, il figlio sinistro, il figlio destro, il padre ed il colore. La struttura rispetta tali proprietà:

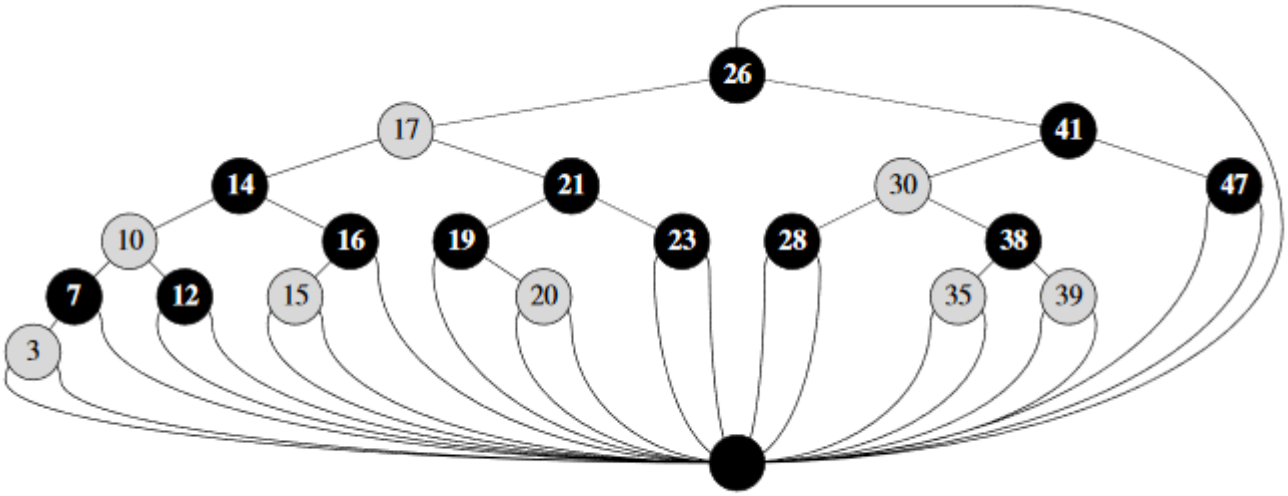
1. Ogni nodo è rosso o nero
2. La radice è nera
3. Le foglie (`nil`) sono nere
4. Se un nodo è rosso, i due figli devono essere entrambi neri
5. Tutti i cammini che vanno dal nodo alle foglie sue discendenti devono avere lo stesso numero di nodi neri.

Con foglie ovviamente si intende le foglie che sono `nil`; dal disegno proposto sotto l'idea dovrebbe essere chiara. Si chiama altezza nera di un nodo `x`, indicata da `bh(x)`, il numero di nodi neri lungo un cammino che inizia dal nodo x (ma non lo include) e finisce in una foglia.



(a)

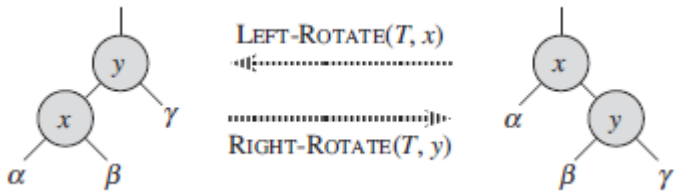
Nel disegno i numeri a fianco dei nodi indicano l'altezza nera; i nodi `nil` ovviamente hanno `bh(x) = 0`. Per comodità si utilizza anche un nodo sentinella chiamato T.nil che sostituisce le foglie nere; come esempio vediamo lo stesso albero visto sopra ma con la sentinella al posto delle foglie.



T.nil

Semplicemente si tratta di un nodo che rappresenta tutte le foglie nere ed è anche padre della radice. Prima di vedere come inserire in nodo in un albero rosso-nero dobbiamo capire come funzionano le rotazioni (fondamentali per l'insert).

Rotazioni. Le operazioni `Tree-Insert` e `Tree-Delete` modificano l'albero impiegando un tempo pari a $O(\log(n))$; tali modifiche possono però portate ad un risultato che non rispetti i 5 punti di prima. Bisogna modificare i colori di qualche nodo e anche la struttura dei puntatori utilizzando le *rotazioni*.



Una rotazione può avvenire a destra o a sinistra; si fa "perno" sul collegamento tra x e y scambiando quindi le radici e l'ordine dei puntatori ai figli dei nodi. Ecco il codice.

```
Left-Rotate(T,x,y) {
 x.right = y.left;        → applico gli spostamenti che mostra la
 x.right.p = x;           figura di sopra; cambio il figlio destro
 y.left = x;              ed il padre di x. Sistemo il figlio
 x.p = y;                 sinistro di y
 Transplant(T,x,y);       → sposta i sottoalberi (solito)
}
```

La procedura `Right-Rotate` è perfettamente simmetrica alla `Left-Rotate`, quindi chiamandole in successione (prima right e poi left o vice versa) ritornerei all'albero di partenza.

Le procedure `Tree-Insert` e `Tree-Delete` sono più complicate rispetto alle stesse usate per i BST, in particolare quella per la cancellazione, e non verranno analizzate. Tuttavia il *Cormen* nel capitolo 13 fornisce il codice completo delle funzioni con tutti i sotto casi di ciascuna funzione.

PROGRAMMAZIONE DINAMICA

La programmazione dinamica è molto simile alla tecnica divide et impera perché divide un grande problema in tanti sotto problemi che sono di facile risoluzione. Usa la ricorsione per cercare di dare una definizione del problema e tenta di ottimizzare la soluzione. Lo scopo quindi sarà quello di creare un algoritmo che mi consenta di risolvere un problema nel migliore dei modi. Il processo di sviluppo si divide in 4:

1. Capire come fare ad ottenere la forma ottima per un dato problema
2. Definire in modo ricorsivo il valore di una soluzione ottima
3. Calcolare il valore della soluzione ottima, solitamente usando uno schema di tipo bottom-up *(verrà spiegato dopo)*
4. Costruire una soluzione ottima delle informazioni calcolate

Per risolvere un problema di programmazione dinamica penso ad un algoritmo ricorsivo buono che mi dia la soluzione ottima, lo scrivo e poi traduco in codice. Vediamo ora come risolvere alcuni problemi tipici di ottimizzazione.

Taglio delle aste.

Una azienda compra lunghe aste di acciaio e le taglia liberamente in aste più corte che poi andranno vendute. Supponiamo di sapere per ogni lunghezza del taglio i il prezzo di vendita p_i. L'azienda vuole sapere il modo migliore di tagliare le aste per massimizzare il guadagno.

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

Data un'asta di lunghezza n ed una tabella di prezzi p_i vogliamo sapere il ricavo r_n massimo che si può ottenere tagliando l'asta. Facciamo l'esempio in cui $n = 4$, ovvero l'asta è lunga 4, e proviamo a vedere di massimizzare il ricavo.

Guardando la tabella ho lunghezza filo – prezzo in colonne e quindi cerco la combinazione che mi dia il miglior ricavo:

- Taglio l'asta in pezzi lunghi 1 e 3 → ricavo 1 + 8 = 9
- Taglio l'asta in un pezzo da 4 → ricavo 9 = 9
- Taglio l'asta in pezzi lunghi 2 e 2 → ricavo 5 + 5 = 10

Mi accorgo che tagliare in 2 pezzi da 2 è la soluzione migliore per un'asta lunga 4. La funzione che devo scrivere mi dirà i tagli ottimi da fare data una lunghezza.

- Se la soluzione ottima prevede il taglio dell'asta lunga *n* in *k* pezzi, per $1 \leq k \leq n$, allora una decomposizione ottima $d = i_1 + i_2 + \ldots + i_k$ dell'asta in i_k pezzi fornisce il massimo ricavo corrispondente $r_d = p_1 + p_2 + \ldots p_k$.

- Traduzione. Se la soluzione ottima prevede di tagliare un'asta lunga *n* in un certo numero di pezzi (almeno 1 e al massimo n) allora il miglior modo di fare questi tagli (di lunghezza i_1, i_2 …) mi dirà anche il massimo ricavo che posso ottenere.

Indicheremo con r_n il ricavo ottimo per un dato taglio in n parti. Quindi per esempio:

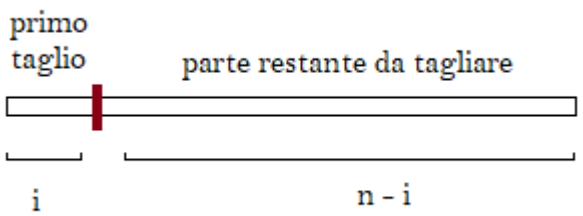
- $r_8 = 22$
  Tagliando in 8 pezzi il ricavo massimo è 22. Faccio un pezzo da 2 e uno da 6 che costano rispettivamente 5 e 17 (17 + 5 = 22)
- $r_5 = 13$
  Tagliando in 5 pezzi il ricavo massimo è 13. Faccio un pezzo da 2 e uno da 3 che costano rispettivamente 5 e 8 (5 + 8 = 13)

La parte fondamentale è quella di dare una definizione ricorsiva a questo problema del taglio ottimale delle aste. Possiamo esprimere i valori r_n in funzione dei ricavi ottimi delle aste più corte.

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

Analizziamo pezzo per pezzo quello che dice la funzione ricorsiva appena ricavata:

- r_n. Il massimo ricavo che posso ottenere tagliando in *n* pezzi
- $\max_{1 \leq i \leq n}$. Siccome il mio scopo è appunto calcolare il massimo ricavo, questa funzione ritorna il massimo valore di un insieme
- $p_i + r_{n-i}$. Considero un primo pezzo di lunghezza *i* (rappresentato da p_i) che è stato tagliato e so che sarà già un taglio con ricavo massimo. Da adesso dovrò preoccuparmi di massimizzare il ricavo nella parte restante dell'asta (r_{n-i}).

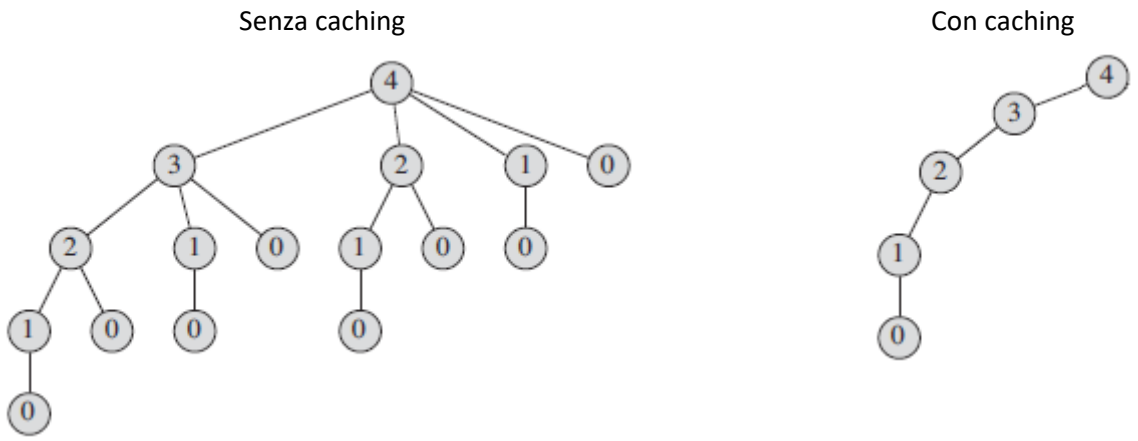


Quindi prima faccio un taglio iniziale p_i ed ottengo il massimo ricavo; non dovrò più interessarmi di questo. Mi dovrò interessare tuttavia della parte di asta rimanente lunga $n - i$ e lo faccio chiamando ricorsivamente r. In questo modo la chiamata ricorsiva sulla parte restante farà un altro primo taglio ottimo e così via fino ad arrivare a coprire tutta la lunghezza n.

Siccome non possiamo sapere a priori quale sia il valore di i ottimale per il primo taglio dobbiamo provarli tutti (da 1 ad n). Siamo sicuri che fra i tanti ci sia un taglio ottimo da fare quindi è per questo che usiamo $\max_{1 \le i \le n}$; ci ritornerà il giusto valore di i che possa massimizzare il taglio dell'asta. Vediamo il codice.

```
CutRod(p, n) {                          → p è l'array di prezzi, n è
 if (n == 0) {                          la lunghezza di p che può
  return 0;                             anche essere nulla
 } else {
  q = -1;                               → valore iniziale del max
  for i := 1 to n do {
   q = max(q, p[i]+CutRod(p, n-i));     → chiamata ricorsiva
  }
  return q;                             → ritorno il massimo
 }
}
```

Questo codice funziona ma fa schifo perché ha un tempo di esecuzione esponenziale in n, ovvero è un $O(2^n)$. Questo accade perché nel considerare tutte le posizioni di i la ricorsione risolve più volte uno stesso sotto problema.



A sinistra vediamo che per un input di dimensione 4 (la radice) il sotto problema con 2 viene risolto due volte, quello con 1 viene risolto 4 volte e quello con 0 è risolto 8 volte. La soluzione giusta è quella di creare una "memoria cache" che si ricordi i risultati dei sotto problemi in modo da affrontarli una volta sola. A destra si vede che usando il "caching" si risolve una sola volta lo stesso problema.

Abbiamo visto che una soluzione ricorsiva funziona ma è pessima, quindi usiamo un approccio iterativo. Facciamo uso di un array `r[0..n]` che salverà i risultati dei vari sotto problemi utilizzando prima il metodo top-down e poi il bottom-up (il secondo è più semplice e leggermente più efficiente, ma entrambi sono $O(n^2)$).

```
MemoizedCutRod(p, n) {
 r = new array[0..n];
 for i := 0 to n {
  r[i] := -1;
 }
 return MCutRodAux(p,n,r);
}
```

→ r è l'array che ricorda i risultati dei sotto problemi ed evita che vengano ricalcolati

→ la funzione CutRod

```
MCutRodAux (p, n, r) {
 if (r[n] >= 0)
  return r[n];
 //funzione CutRod - inizio
 if (n == 0) {
  q = 0;
 } else {
  q = -1;
  for i := 1 to n {
   q = max(q, p[i]+MCutRodAux(p,n-i,r))
  }
  r[n] = q;
 }
 // funzione CutRod - fine
 return q;
}
```

→ se ho già la soluzione non faccio nulla

→ eseguo il codice solito visto anche prima di CutRod. Se arrivo qui vuol dire che sto risolvendo un problema che prima non avevo già trovato e quindi mi ricordo di salvare la soluzione in r[n]

→ ritorno il massimo

È facile notare come in `MemoizedCutRod` venga creata la cache ovvero l'array r, poi la `MCutRodAux` applica la ricorsione. All'inizio il controllo con l'if è fondamentale perché se si accorge che la soluzione per un problema è già stata calcolata ed è nella cache (dentro ad r) non fa nulla perché già si sa il risultato. Vediamo l'approccio bottom-up.

```
BottomUpCutRod(p, n) {
 r = new array[0..n];
 r[0] = 0;
 for j := 1 to n {
  q := -1;
  for i := 1 to j {
   q := max(q, p[i]+r[j-i])
  }
 }
 return r[n];
}
```

→ array "cache" che salverà i dati

→ asta lunga 0 = ricavo 0 euro

→ solito metodo ma ora anziché fare una chiamata ricorsiva guardo direttamente dentro a r perché r ha i risultati delle soluzioni già calcolate per j = 1, 2, 3...

→ ritorno il valore ottimo per la lunghezza n (che è salvato nella cache, ovvero in r)

Questa versione è ancora più semplice e produce lo stesso risultato dell'altra e sono entrambe $O(n^2)$. Anche qui si vede che viene creato l'array e poi per ogni possibile sotto caso viene risolto il relativo sotto problema.

- `max(q, p[i]+MCutRodAux(p,n-i,r))`
- `max(q, p[i]+r[j-i])`

Le due notazioni sono equivalenti solo che nel primo caso faccio la ricorsione sulla parte restante basandomi sulla funzione, sul secondo faccio anche li la ricorsione ma basandomi sull'array di risultati salvati.

Moltiplicare una sequenza di matrici.

Fissiamo la notazione $A_{m,n}$ che sia una matrice A di *m* righe ed *n* colonne. Il prodotto di due matrici è possibile se e solo se il numero di colonne della prima è uguale al numero di righe della seconda. In tal caso il risultato sarà una matrice C con tante righe quante quelle della prima e tante colonne quante quelle della seconda.

- $A_{5,6} \times B_{6,2} = C_{5,2}$
  A ha 6 colonne, B ha 6 righe e quindi posso fare il prodotto delle due. La matrice C sarà il prodotto AB con tante righe quante A (ovvero 5) e tante colonne quante B (ovvero 2).
- $A_{5,8} \times B_{4,1} = ?$
  A ha 8 colonne, B ha 4 righe e quindi non posso fare il prodotto delle due. La formula del prodotto prevede la somma dei prodotti "incrociati" fra righe e colonne quindi avendo sballato A.colonne e B.righe l'algoritmo non funziona.

Supponiamo di avere tre matrici A_1, A_2 ed A_3 da moltiplicare e che rispettino la regola appena vista (quindi sono 3 matrici moltiplicabili). I prodotti possono essere fatti in vari modi in base a dove vengono messe le parentesi:

- $(A_1(A_2A_3))$
- $((A_1A_2)A_3)$
- $(A_1)(A_2)(A_3)$

Anche se non sembra, il modo di sistemare le parentesi influisce molto sul costo di calcolo della moltiplicazione. Supponiamo di avere per esempio $A_{10,100}$, $B_{100,5}$ e $C_{5,50}$. Applicando due schemi diversi di parentesizzazione otteniamo:

- ((AB)C) → eseguiamo 10 * 100 * 5 + 10 * 5 * 50 = 7500 prodotti scalari
- (A(BC)) → eseguiamo 100 * 5 * 50 + 10 * 100 * 50 = 50000 prodotti scalari

Il problema che dobbiamo risolvere è quello del posizionamento delle parentesi in modo ottimale che minimizzi il numero di prodotti scalari. In altre parole, bisogna dire qual è il giusto schema di parentesi da mettere (per fare le moltiplicazioni) in modo da fare meno conti possibile. Scriviamolo formalmente:

Problema. Data una sequenza di *n* matrici, dove una matrice A_i ha dimensioni $p_{i-1} * p_i$ per *i* che va da 1 ad *n*, determinare lo schema di parentesizzazione completa del prodotto che minimizza il numero di prodotti scalari.

Chiameremo $A_{i..j}$ la matrice che si ottiene calcolando il prodotto $A_i A_{i+1} A_{i+2} \dots A_j$. Subito possiamo distinguere i due macro casi che ci possono portare alla soluzione ricorsiva del problema di parentesizzazione.

1. $i = j$. In questo caso si ha che non si fa nessun prodotto scalare perché se i = j vuol dire che ho una sola matrice. In pratica mi trovo nella situazione di dovere parentesizzare A_i ma l'unico modo di farlo è (A_i) e non ho nessun prodotto di matrici da fare.
2. $i < j$. In questo caso si ha che si fa almeno un prodotto scalare perché se i < j vuol dire che posso avere almeno due matrici. Con i = 1 e j = 3 vuol dire che il

prodotto delle matrici $A_1A_2A_3$ ha dei prodotti scalari e ci sono delle parentesi da mettere (bisogna capire dove però).

Considerando il caso 2 ci deve essere per forza un valore *k* (tale che $i \le k \le j$) in grado di spezzare in due la matrice. In altre parole: per un qualsiasi valore *k* prima faccio il calcolo di $A_{i..k}$, poi $A_{k+1..j}$ e infine le moltiplico per ottenere $A_{i..j}$.

$$A_{ij} = \underbrace{(A_i A_{i+1} \ldots A_k)}_{\text{Parte 1}} \underbrace{(A_{k+1} \ldots A_j)}_{\text{Parte 2}}$$

$A_{i..j}$ posso spezzarla in due blocchi parentesizzati (Parte 1 e Parte 2) che vengono "separati" da un *k* messo in posizione opportuna. In particolare se *k* si trova nel posto giusto ottengo una parentesizzazione ottima e di conseguenza sono sicuro che anche la sotto sequenza dentro alla parentesi sarà una parentesizzazione ottima.

- Trovo il k adatto → ho una parentesizzazione ottima per $A_i \ldots A_k$ → sono sicuro che anche ciò che sta dentro ad $A_i \ldots A_k$ sia una parentesizzazione ottima perché se non lo fosse allora vorrebbe dire che avrei scelto all'inizio il k nel posto sbagliato!

Adesso abbiamo tutti i dati e per ricavare la ricorsione. Usiamo *m[i, j]* per definire il numero minimo di prodotti scalari richiesti per calcolare la matrice finale $A_{i..j}$. Allora riprendendo le due considerazioni fatte sopra:

1. $i = j$. Ho che m[i, i] = 0. Con $A_{i..i}$ non faccio nessun prodotto scalare (ho a che fare con una matrice sola).
2. $i < j$. Ho che $m[i, j] = m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$. Con $A_{i..j}$ abbiamo visto che si spezza la matrice in due parti ottime ad un certo indice *k*. Nel calcolo di m[i, j] faccio: "costo minimo da *i* a *k*" + "costo minimo da *k+1* a j" + "costo per fare la moltiplicazione di matrici".

Notare che $p_{i-1}p_kp_j$ salta fuori dal fatto che ogni matrice A_i ha dimensioni $p_{i-1} * p_i$ e siccome moltiplico $A_{i..k}$ e $A_{k+1..j}$ mi servono $p_{i-1}p_kp_j$ prodotti scalari. In altre parole:

- A_i → dimensione $p_{i-1} * p_i$
- $A_{i..j}$ → dimensione $p_{-1} * p_j$
- $A_{i..k} * A_{k+1..j}$ → dimensione $p_{i-1} * p_k * p_j$

Non conoscendo la posizione esatta di *k* che mi fornisce la parentesizzazione ottima dovrò provare tutte le possibili posizioni e tenere buona quella che mi restituisce il numero minimo. Ecco la definizione della ricorsione.

$$m[i,j] = \begin{cases} 0\,, & i = j \\ \min\limits_{i \le k \le j} \{\, m[i,k] + m[k+i,j] + p_{i-1}p_kp_j \,\}\,, & i < j \end{cases}$$

I valori *m[i, j]* sono i costi delle soluzioni ottime dei sotto problemi (= il numero minimo di prodotti scalari per calcolare la matrice $A_{i..j}$). Traduciamo ora in codice

utilizzando l'approccio top-down che è facile (poi il bottom-up) con la tecnica già vista di salvare in una "cache" i risultati dei sotto problemi. In questo modo evitiamo di calcolare più volte la soluzione dello stesso problema. Useremo una matrice `m` per salvare il numero minimo di prodotti scalari richiesti per calcolare $A_{i..j}$.

```
MemoizedLookupChain(p) {
 m = new matrix[1..n, 1..n];        → creo m
 for i := 1 to n                    → la riempio di valori
  for j := 1 to n                   sentinella che mi serviranno
   m[i,j] := ∞;                     dopo per vedere se ho già
                                    risolto quel problema o no
 return LookupChain(m,p,1,n);
}

LookupChain (m, p, i, j) {
 if (m[i,j] < ∞)                    → se ho già risolto il problema,
  return m[i,j];                    non faccio nulla ed esco

 if (i == j) {                      → caso base
  m[i,j] = 0;
 } else {
  for k := i to j-1 {               → applico la definizione
   q = LookupChain(m,p,i,k) +       ricorsiva e calcolo q; se sono
       LookupChain(m,p,k+1,j) +     arrivato qui vuol dire che non
       p_{i-1}p_k p_j;              avevo mai risolto il problema.
   if (q < m[i,j]) {                → provvedo a salvare q dentro ad
    m[i,j] = q;                     m[i,j] se è soluzione migliore
   }
  }
 }

 return m[i,j];                     → ritorno la soluzione migliore
}                                   per la coppia [i,j].
```

Vediamo ora l'approccio bottom-up che è più complesso. Utilizzeremo l'array *s[i, j]* che ci dirà (dati *i* e *j*) il valore di *k*, cioè *s[i, j]* mi dirà dove mettere il *k* per ottenere la parentesizzazione ottima. In particolare l'algoritmo userà:

- `m[1..n, 1..n]` = una tabella che salva i costi m[i, j] (salva il numero minimo di prodotti scalari per calcolare la matrice $A_{i..j}$)
- `s[1..n, 1..n]` = una tabella che salva l'indice *k* corrispondente al costo ottimo del calcolo di m[i, j]. Verrà usata `s` per costruire la soluzione ottima.

```
BottomUpMatrix(p) {                 → matrice in input
 n = p.length - 1;
 m = new matrix[1..n,1..n];         → matrici di supporto m ed s
 s = new matrix[1..n,1..n];

 for i := 1 to n {                  → riempio di zeri la matrice dei
  m[i,i] = 0;                       costi minimi in preparazione
 }
 for l := 2 to n {                  → applico la ricorrenza su m[]
  for i := 1 to n-l+1 {             (matrice che fa da "cache") per
```

```
    j = i+l-1;                  trovare i costi minimi per sequenze
    m[i,j] = ∞;                 di lunghezza l = 2 o superiore.
    for k := i to j-1 {
     c = pi-1pkpj
     q = m[i,k]+m[k+1,j]+c;     → applico la formula che ho ricavato

     if (q < m[i,j]){           → se mi accorgo di avere trovato un
      m[i,j] = q;               minimo "migliore" non ancora
      s[i,j] = k;               calcolato allora lo salvo dentro alle
     }                          matrici che salvano i costi ottimi e
    }                           le posizioni di k relative ai costi
   }                            ottimi
  }
  return [m, s];                → ritorno sia m che s.
}
```

Questo approccio è abbastanza complicato; l'approccio top-down è più semplice sia da scrivere che da capire guardando il codice.

- Nota. I due problemi affrontati fino ad ora hanno le stesse idee di fondo. Parto posizionando un indice fisso che è soluzione ottima (il primo taglio del problema 1 e la posizione del k nel problema 2) e poi ricorsivamente applico la legge ai sotto problemi. Anche il modo di scrivere il codice è simile perché si usa la "cache" e la stessa traccia di soluzione.

La più lunga sottosequenza comune (LCS)

Definiamo le stringhe formate dalle lettere nell'insieme D = {A, C, T, G} riferendoci a due esempi; la lunghezza di S_1 ed S_2 non ha restrizioni, basta che sia ≥ 1.

S_1 = A C C G G T C A G (lungh. = 9) S_2 = G T C G T T A C (lungh. = 8)

Lo scopo è quello di trovare una LCS, ovvero una stringa S_3 formata dalle lettere comuni ad entrambe le stringhe che si trovano nella stessa posizione ma non sono per forza contigue. In pratica partendo da sinistra guardo le lettere di S_1 ed S_2 e mi sposto a destra di 1; appena ne trovo due uguali le scrivo in S_3.

S_3 = -	S_1 = A C C G G T C A G S_2 = G T C G T T A C
S_3 = G	S_1 = A C C G G T C A G S_2 = G T C G T T A C
S_3 = G T	S_1 = G T C A G S_2 = T C G T T A C
S_3 = G T C	S_1 = C A G S_2 = C G T T A C
S_3 = G T C A	S_1 = A G S_2 = T T A C

La stringa finale S_3 è formata da una sequenza (G T C A) che è presente in entrambe le stringhe ed è la più lunga possibile fra le tante sottosequenze. In particolare S_3 è una sottosequenza comune di S_1 ed S_2 perché le lettere in S_3 compaiono nello stesso ordine anche in S_1 ed S_2 (non compaiono una dietro l'altra ma comunque sono in sequenza). Vediamo ora gli *indici* di S_3.

S_1 = A C C G G T C A G
S_3 = G T C A (4, 6, 7, 8)

S_1 = G T C G T T A C
S_3 = G T C A (1, 2, 3, 7)

Tra parentesi sono indicati gli indici della LCS (di S_3) che stanno ad indicare il posto in cui trovo la lettera dentro alla sequenza di origine. S_i = (4, 6, 7, 8) vuol dire che se guardo il S_1 trovo la G in posto 4, la T in posto 6, la C in posto 7 e la A in posto 8.

X = A B C B D A B

Y = B D C A B A (lungh. = 8)

Come ultimo esempio vediamo queste due stringhe. Notare che una sottosequenza prodotta dal calcolo sopra può essere Z = B C A che è valida ma non è la più lunga; serve avere una LCS (la sottosequenza più lunga). In questo caso ci sono due LCS ovvero Z = B C B A e Z = B D A B quindi capiamo che la soluzione può non essere unica e che fra le tante sottosequenze bisogna considerare le più lunghe.

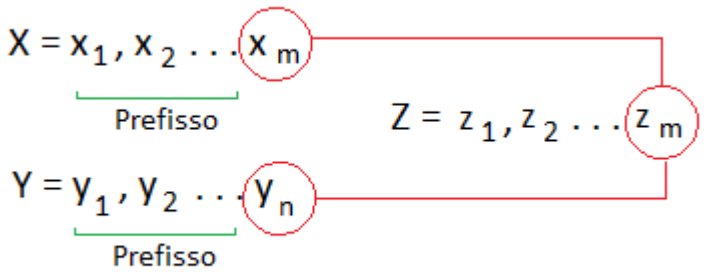
Problema. Date due sequenze $X = \{ x_1, x_2, x_3 \ldots x_m \}$ ed $Y = \{ y_1, y_2, y_3 \ldots y_n \}$ si vuole trovare una sottosequenza massima che sia comune ad X e Y.

Chiamiamo X_i il prefisso della stringa X di lunghezza *i*; vuol dire che se avessi una stringa X = A B C D E il prefisso 3 sarebbe X_3 = A B C. Il prefisso mi sta ad indicare quante lettere considerare della stringa (partendo da sinistra ovviamente). Siano:

- $X = \{ x_1, x_2, x_3 \ldots x_m \}$ una sequenza
- $Y = \{ y_1, y_2, y_3 \ldots y_n \}$ un'altra sequenza
- $Z = \{ z_1, z_2, z_3 \ldots z_k \}$ una LCS di X ed Y

Date queste 3 condizioni e sfruttando la cosa appena detta sui prefissi possiamo concludere che valgono questi 3 casi:

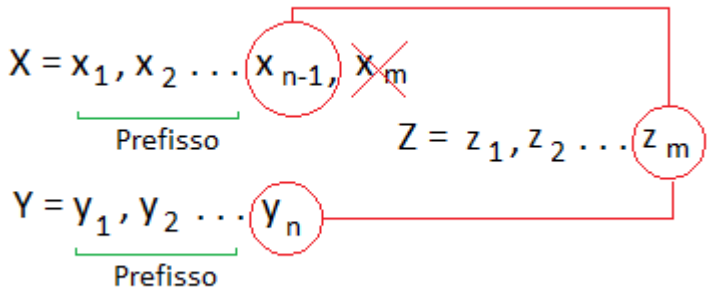
1. Se $x_m = y_n$ allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}



Nota. Se i due caratteri terminali sono uguali vuol dire che per forza faranno parte della LCS perché quest'ultima per definizione ha i caratteri che stanno in entrambe le stringhe. Allora tutti i caratteri prefissi di Z che non ho ancora

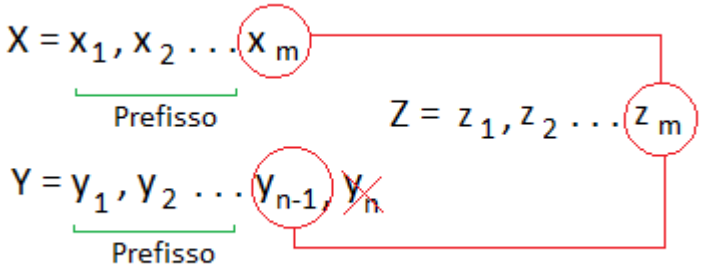
controllato (quelli prima di z_m cioè da z_{m-1} in giù) saranno LCS per X_{m-1} e Y_{n-1} perché tolgo 1 carattere ad X e Y che ho controllato.

2. Se $x_m \neq y_n$ allora $z_k \neq x_m$ implica che Z è una LCS di X_{m-1} e Y



Nota. Ragionamento simile a prima ma sta volta avendo che le lettere terminali sono diverse, ne tengo fissa una (la y_n) e scorro la x analizzando x_{n-1} (dato che x_n non va bene visto che è diversa da y_n)

3. Se $x_m \neq y_n$ allora $z_k \neq y_n$ implica che Z è una LCS di X e Y_{n-1}



Nota. Adesso nella ricerca di lettere uguali tengo ferma la x ma scorro sulle y in cerca di due caratteri uguali da mettere in Z.

Queste considerazioni fanno vedere come si procede: partendo dalla fine guardo se le lettere corrispondono: se si allora so che andranno messe in Z, altrimenti ne scorro una tenendo ferma l'altra e cerco un match.

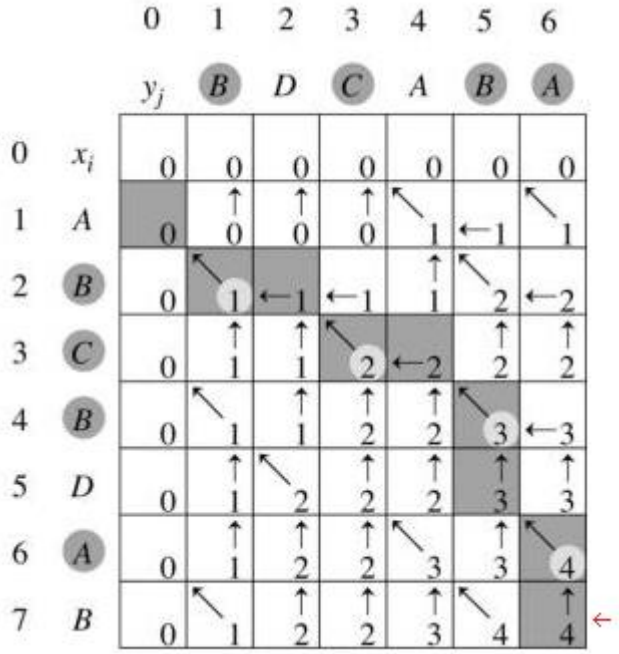
Se definissimo *c[i,j]* la lunghezza di una LCS delle sequenze X_i e Y_j allora otterremo la seguente formula ricorsiva:

$$c[i,j] = \begin{cases} 0, & i = 0, j = 0 \\ c[i-1, j-1] + 1, & i, j > 0; x_i = y_j \\ \max(c[i, j-1], c[i-1, j]), & i, j > 0; x_i \neq y_j \end{cases}$$

Nella prima riga gestisco il caso in cui ci siano le sequenze lunghe zero. Poi nella seconda riga gestisco il caso del punto 1 perché quando trova le due stringhe uguali $x_i = y_j$ aumenta di 1 la lunghezza della LCS (ho aggiunto il carattere a Z). L'ultima riga gestisce i punti 2 e 3 perché quando $x_i \neq y_j$ serve prendere la LCS di lunghezza

massima fra quelle ricavate tenendo fisso uno dei due indici e spostando l'altro. Prima di scrivere il codice vediamo questo esempio:

Rappresentazione grafica



X = A B C B D A B
Y = B D C A B A

- Sulla riga vediamo la stringa Y e nella colonna c'è X. Si parte da in basso a destra dove c'è la freccetta rossa, ovvero dal fondo delle stringhe.
- Notare che anche nei 3 punti visti sopra (quelli coi disegnetti) si partiva dal fondo della stringa. La tabella mostra esattamente quanto riportato nelle 3 condizioni.
- Le freccette indicano dove ci si sposta nella griglia.

Partendo da in basso a destra equivale a dire "parto dal fondo della stringa". Devo ricordare cosa avevo scritto prima, ovvero:

$$c[i,j] = \begin{cases} 0, & i = 0, j = 0 \\ c[i-1, j-1] + 1, & i, j > 0; x_i = y_j \\ \max(c[i, j-1], c[i-1, j]), & i, j > 0; x_i \neq y_j \end{cases}$$

Guardando questa equazione sono in grado di fare le mosse all'interno della griglia; basta guardare se le lettere x_i e y_j sono uguali (mi sposto in diagonale) oppure se sono diverse (mi sposto in su o a sinistra).

- Mi trovo nella cella di partenza (c[7,6] con la freccetta rossa) e ho $i, j > 0$ quindi confronto le due lettere. Siccome $x_i \neq y_j$ dato che B ≠ A ricado nel caso 3 e mi sposto in su di una cella e a sinistra di una cella.
- Guardo la cella sopra (la c[6,6]). Siccome $x_i = y_j$ dato che A = A ho trovato un match e ricado nel caso 2. Faccio un +1 (che equivale a spostarmi in diagonale) andando in c[5,5] e diminuisco il contatore di 1 (da 4 passa a 3). Ciò accade perché avendo avuto un match ho un carattere in meno da guardare.
- Guardo la cella a sinistra (la c[7,5]). Siccome $x_i = y_j$ dato che B = B ho trovato un match e ricado nel caso 2. Faccio un +1, mi sposto in diagonale e decremento.

Seguendo questi passi si completa la tabella con vari percorsi ma bisogna prendere il massimo (quello che equivale alla sottosequenza più lunga). L'equazione descrive le mosse da fare nella tabella ovvero descrive i valori da mettere dentro una matrice

che farà parte della funzione (scritta a breve). Notare anche che nei 3 punti con i disegnetti prima dell'equazione si partiva dal fondo della stringa e se si trovava un match dei caratteri si proseguiva, altrimenti ci si spostava di un passo. Infatti:

- Partire dal fondo della stringa → partire dal fondo della matrice
- Trovare un match→ decrementare il valore (spostarsi in diagonale)
- Non trovare il match → spostarsi a sinistra e in alto in cerca di un match

L'equazione di ricorrenza descrive i 3 punti analizzati nella considerazione quindi sono entrambi equivalenti. Scriviamo il codice della funzione `LCS-Length` che prende in input due stringhe X ed Y .

```
LCS-Length(X, Y) {                          → X e Y sono le stringhe
 m = X.length;
 n = Y.lenght;
 b = new matrix[1..m, 1..n];                → direzione degli spostamenti
 c = new matrix[0..m, 0..n];                → salva la LCS

 for i := 0 to m do {c[i,0] = 0;}           → riempio di 0 le righe e colonne
 for j := 0 to n do {c[0,j] = 0;}           iniziali della tabella

 for i := 1 to m do {                       → scorro tutta la tabella
  for j := 1 to n do {

   if (X[i] == Y[j]) {                      → se trovo due caratteri uguali
    c[i,j] = c[i-1,j-i] + 1;                aumento di 1 e mi sposto a in
    b[i,j] = "↖";                           diagonale (vedi b[i,j])
   } else {
    if (c[i-1,j] ≥ c[i,j-1]) {              → se trovo due caratteri che non
     c[i,j] = c[i-1,j];                     sono uguali decido se spostarmi
     b[i,j] = "↑";                          in alto o a sinistra. Salvo lo
    } else {                                spostamento che ho fatto in b
     c[i,j] = c[i,j-1];
     b[i,j] = "←";
    }
   }

  }
 }
 return b,m;
}                                           → ritorno LCS e spostamenti
```

Il costo complessivo di questa funzione è $\Theta(m * n)$.

ALGORITMI GOLOSI

Gli algoritmi golosi (*greedy algorithms*) fanno sempre la scelta che sembra ottima in un determinato momento, cioè fanno una scelta localmente ottima nella speranza che alla fine si giunga ad una soluzione che globalmente sia la migliore. A differenza della programmazione dinamica non analizzano tutti i casi possibili e per un gran numero di problemi sono molto efficienti. Vediamo alcuni esempi di problemi.

Selezione di attività.

Supponiamo di avere un insieme S = {a_1, a_2, a_3 . . . a_n} che contiene *n* attività *a* diverse che devono utilizzare una risorsa comune *r*; quest'ultima può essere usata da una sola attività per volta.

- Esempio. Supponiamo di avere un insieme S di lezioni universitarie (le attività a_1, a_2, a_3 . . . a_n) che devono tenersi in un'aula (la risorsa comune) ma ovviamente nell'aula si può svolgere una lezione alla volta (risorsa *r* usata da 1 attività sola per volta)

Ogni attività ha un tempo di inizio s_i ed un tempo di fine f_i che definiscono l'intervallo di esecuzione. Si dice che le attività sono compatibili se non si sovrappongono gli intervalli delle attività; date a_i ed a_j sono compatibili se $s_i \geq f_j$ o $s_j \geq f_i$.

- Esempio. Due lezioni sono compatibili se l'orario di inizio e fine di una non si accavalla con l'orario di inizio e fine dell'altra. In questo modo quando una lezione finisce, l'aula (= la risorsa) si libera e può iniziare l'altra lezione (= l'attività successiva).

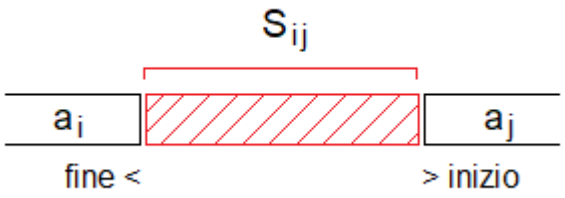
Vogliamo selezionare il sottoinsieme che contiene il maggior numero di attività compatibili, cioè scegliere quali attività fare (senza che si accavallino) in modo da poter fare più attività possibili.

- Esempio. Abbiamo un'aula e un insieme di lezioni con gli orari di inizio e fine. Lo scopo finale è quello di scegliere quali lezioni fare (= scegliere le attività) in modo che non si accavallino e in modo da farne il più possibile durante la giornata.

Vediamo un esempio col seguente insieme S di attività, ognuno col suo tempo di inizio e di fine.

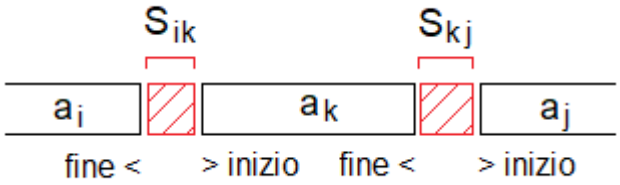
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Il sottoinsieme $A = \{a_3, a_9, a_{11}\}$ è compatibile perché gli intervalli selezionati non si sovrappongono, infatti a_9 inizia dopo che a_3 finisce per esempio. Tuttavia *A* non è il sottoinsieme di dimensione massima e quindi non ci interessa; il risultato ottimo di interesse invece è $A = \{a_1, a_4, a_8, a_{11}\}$ oppure $A = \{a_2, a_4, a_9, a_{11}\}$. Ci possono essere più soluzioni ottime al problema. Vediamo ora di ricavare l'equazione di ricorrenza.



Indichiamo con S_{ij} l'insieme delle attività che iniziano dopo la fine di una attività a_i e

che terminano prima dell'inizio di a_j. Supponiamo di volere un insieme massimo di attività compatibili che stia dentro a S_{ij}; supponiamo anche di avere A_{ij} un insieme massimo che contiene l'attività A_k. Includendolo nella soluzione ottima restano due sotto problemi da risolvere: quello S_{ik} e quello S_{kj}.



Si tratta di un ragionamento che abbiamo già visto. Supponendo che a_k sia un'attività ottima che va bene messa lì, mi restano gli altri due spazi da coprire, quindi altri due sottoproblemi. Indicando con $c[i,j]$ la dimensione di una soluzione ottima per S_{ij} si ricava la seguente equazione di ricorrenza.

$$c[i,j] = \begin{cases} 0 & , \quad S_{ij} = \emptyset \\ \max\{c[i,k] + c[k,j] + 1\}, & \quad S_{ij} \neq \emptyset \end{cases}$$

Ora potremo usare la programmazione dinamica per ottenere una funzione ricorsiva che analizza tutti i casi possibili ma invece useremo l'approccio *greedy*. L'intuito dice di scegliere l'attività in S che finisce per prima, così rimane più spazio disponibile per le prossime attività. Siccome i dati (vedi tabella sopra) sono dati in ordine crescente, la scelta golosa è l'attività a_1.

- Facendo la scelta golosa ci resta un solo sottoproblema da risolvere (e non due come si vedeva prima): trovare le attività che iniziano dopo la fine di a_1.

Possiamo quindi selezionare ripetutamente l'attività che finisce per prima a patto che sia compatibile con l'ultima esaminata (che non si accavallino le attività!). Vediamo il codice che riceve in input *s* ed *f* i due array che rappresentano i tempi di inizio e fine dell'attività k.

- Nota. Data un'attività k, `s[k]` ritorna il tempo di inizio di k ed `f[k]` la fine

Ritorna un insieme di attività compatibili. L'algoritmo inizia creando a_0 un'attività inventata che dura 0 secondi e si usa come punto di partenza.

```
Rec-AS(s,f,k,n) {
 m = k + 1;
 while ((m ≤ n) and (s[m] < f[k])){
  m++;
 }
 if (m ≤ n) then
  return {am} U Rec-AS(s,f,k,n);
 else
  return ∅;
}
```

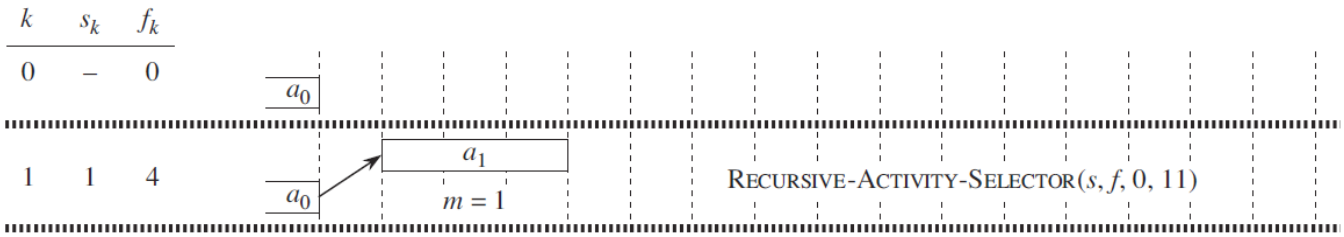
→ n dice quanto grande è la tabella di input
→ cerca la prima attività più corta che non si accavalli con la precedente
→ ritorna il sottoinsieme di attività compatibili ottimo con l'aggiunta dell'attività appena selezionata. Se è arrivato a fine ciclo, ritorna l'insieme vuoto.

Per capire meglio come funziona il codice supponiamo di avere come esempio la

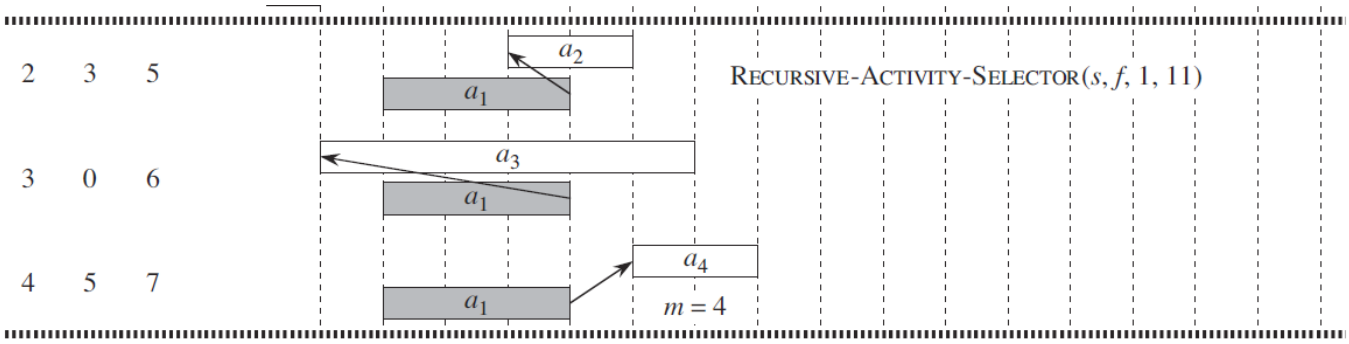
tabella di attività vista all'inizio del paragrafo:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

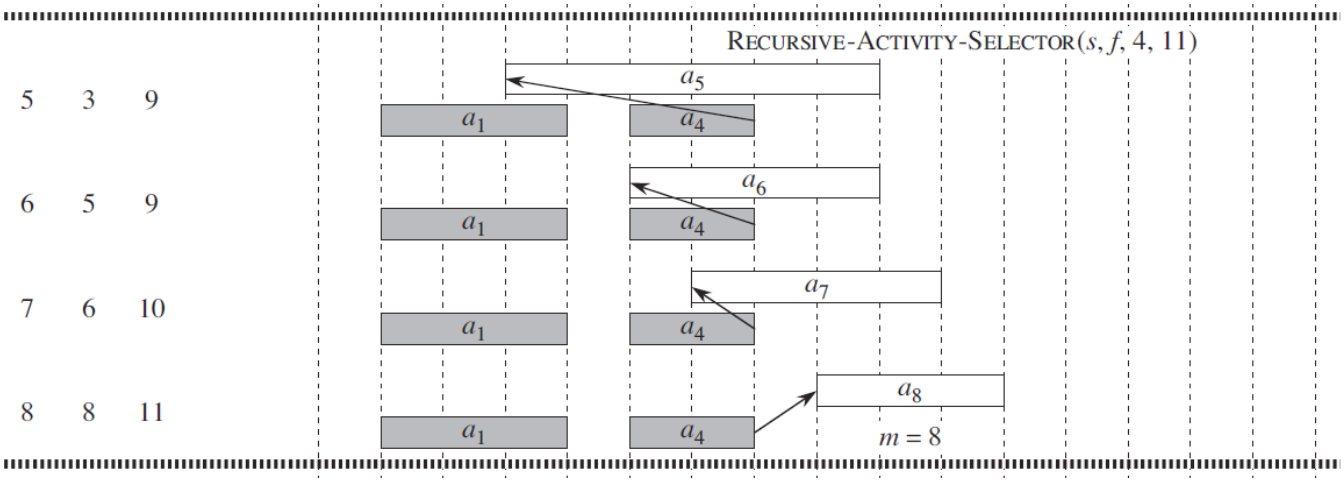
L'esecuzione della procedura `Rec-AS` su quella tabella produrrebbe i seguenti step dove k – s_k – f_k indicano l'attività k (della tabella sopra) con tempo di inizio e fine. Il `while` aumenta `m` in cerca della prima attività che inizia dopo la fine (`s[m] < f[k]`). La `m` dice il numero dell'attività da scegliere e salvare come buona.



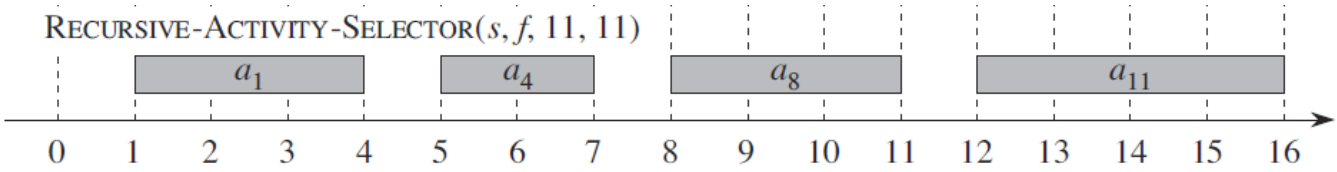
L'algoritmo prende subito la prima attività a_1 perché a_0 dura 0. Avendo m = 1 so che l'attività a_1 è compatibile e la salvo (la scrivo in grigio). Ora proseguo con la chiamata ricorsiva in cerca di altre attività.



L'attività 2 non va bene perché si accavalla con a_1 dato che ha intervallo [3-5] e nemmeno a_3 va bene. La a_4 va bene perché inizia dopo la fine di a1; m si ferma a 4 e quindi la prossima attività da salvare sarà a_4. Proseguo con la ricorsione.



Ragiono come prima e scarto tutte quelle col tempo di inizio inferiore al tempo di fine dell'ultima presa. Quando ne trovo una di compatibile (= col tempo di inizio maggiore del tempo di fine dell'ultima) la salvo. Alla fine otterrò:



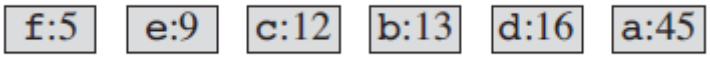
Si tratta del risultato che era già stato riportato in precedenza; mostra le attività scelte come compatibili. Il disegno dice che posso far stare al massimo 4 attività nell'arco di tempo da 1 a 16 e le posso disporre esattamente come segnato. Una versione iterativa equivalente a quella ricorsiva:

```
Iter-AS(s,f) {
 n = s.length;
 A = {a1};                    → la prima attività è scelta di default
 k = 1;                       → ricorda l'ultima attività inserita
 for m = 2 to n do {
  if (s[m] ≥ k[m])) then {    → trova l'attività che finisce per
   A = A U {am};              prima e che non si accavalli (inizia
   k = m;                     dopo la fine <-> s[m] ≥ k[m]. Salva
  }                           l'indice dell'ultima attività inserita.
 }
 return A;                    → ritorna la lista di attività
}                             compatibili salvate
```

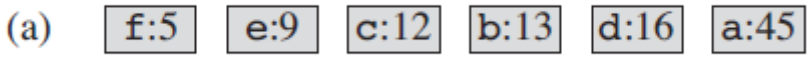
In questa versione risulta più facile vedere che questo algoritmo impiega $O(n)$; lo stesso vale per la funzione ricorsiva ovviamente.

Codici di Huffmann

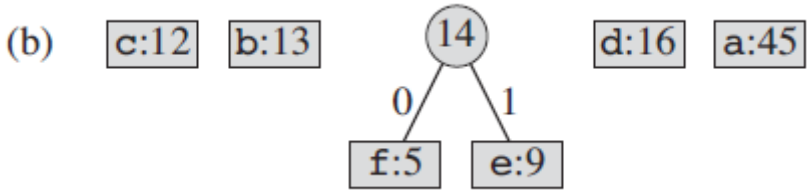
I codici di Huffmann sono una tecnica molto diffusa ed efficace per comprimere i dati; risparmi dal 20% al 90% sono tipici. I dati vengono considerati come una sequenza di caratteri e l'algoritmo goloso usa una tabella che contiene quante volte compare ogni carattere. Facciamo subito un esempio:



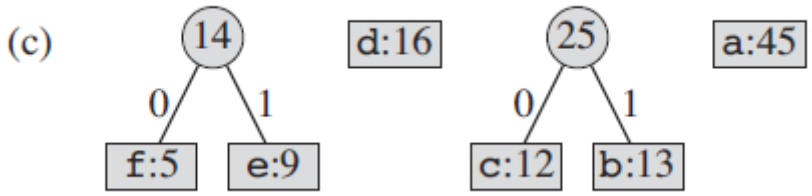
Ogni carattere rappresenta un dato e il numero dice quante volte compare quel dato, cioè indica la *frequenza*. Lo scopo è quello di costruire un albero guardando che la somma delle frequenze sia sempre la più bassa possibile. Vediamo un esempio.



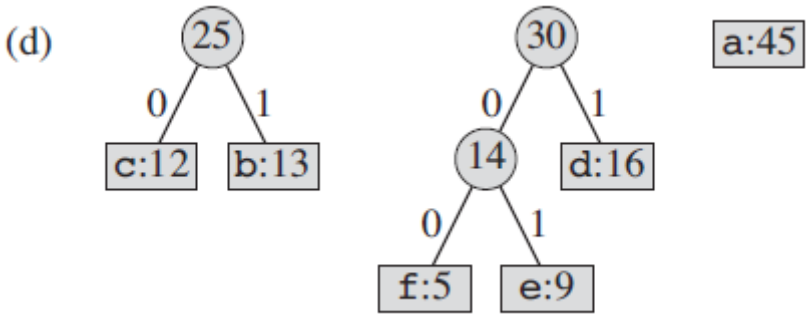
Si parte da qua e si vede che 5 e 9 sono i più piccoli quindi si fondono e si crea una struttura ad albero: 5 e 9 saranno foglie, 14 (5 + 9) sarà il padre.



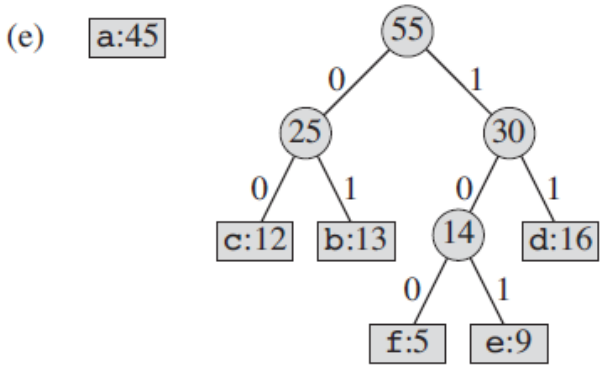
I due più piccoli ora sono 12 e 13, quindi applichiamo lo stesso procedimento per creare un nuovo albero.

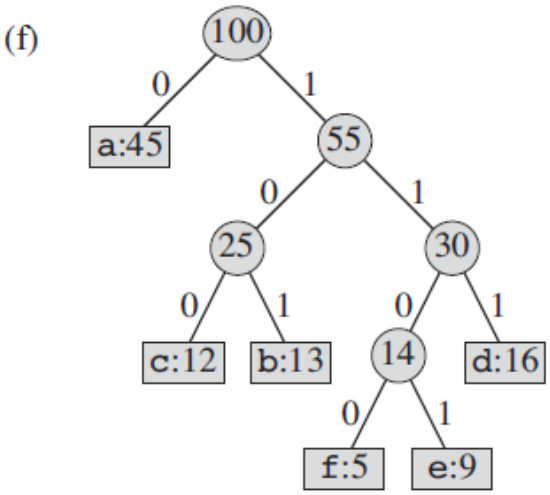


I numeri più piccoli adesso sono il 14 e il 16, quindi nuovamente creo un albero che avrà come radice la somma dei due numeri e come figli i due nodi/foglie.



I due numeri più piccoli ora sono 25 e 30; creo un nuovo nodo radice che conterrà il 55 (25 + 30) e i figli saranno due sottoalberi, ovvero gli alberi radicati in 25 e in 30.





L'ultimo passaggio è banale; l'albero avrà radice con frequenza 100 e i figli saranno una foglia contenente 45 ed l'albero rimanente.

Infine si numerano con 0 i collegamenti che vanno ai figli sinistri mentre 1 agli altri. L'algoritmo è molto semplice: si cercano sempre i due nodi con numero minore e si crea un padre che ha come campo la somma dei campi dei due nodi.