Algoritmi

Università di Verona
Imbriani Paolo -VR500437
Professor Roberto Segala

January 14, 2025

Contents

1 Introduzione agli algoritmi

Ci sono diverse definizioni di 'algoritmo' ma quella più semplice per definirlo è una sequenza di istruzioni volta a risolvere un problema. In maniera più semplice, possiamo definirlo come una ricetta. Le ricette sono delle istruzioni per creare dei piatti: queste istruzioni sono precise tuttavia possono avere anche delle varianti. Il problema non è trovare la ricetta, ma capire quale sia quella gisuta da utilizzare in base al problema posto.

In questo corso impareremo come *decidere*. Studiare i diversi approcci e il metodo migliore per affrontare un problema. Capiremo come confrontare gli algoritmi fra di loro.

1.1 Complessità

Classifichiamo gli algoritmi in base alla loro complessità ovvero quanto tempo ci mettono per essere completati. Il nostro obiettivo è quello non di misurare il tempo in sè (poiché può dipendere dal tipo di macchina che utilizziamo) che impiega un programma a finire ma piuttosto capire il numero di istruzioni elementari che vengono impiegate per risolvere il problema.

La complessità non è un numero bensì una funzione, poiché mappa la dimensione del problema in numero di istruzioni da eseguire. Per esempio, quando si parla di dimensione di una matrice ci si viene più comodo vedere il numero di colonne e righe piuttosto che contare il numero di elementi all'interno della matrice. La dimensione del problema è un insieme di oggetti, tipicamente un insieme di numeri che ci permette di avere un idea chiara di capire quanto sia grande il problema. Se riusciamo a misurarlo bene, potremmo facilitarci la vita nel risolvere il problema.

1.2 Complessità dei costrutti e ordini di grandezza

È doveroso stare attenti a cosa moltiplichiamo: se moltiplichiamo due vettori, il numero di operazioni da eseguire è esattamente n. Mentre se fossero orgnanizzate in matrici quadrate con lunghezza del lato $\sqrt{n}$ la sua complessità andrebbe ad aumentare a $O(n\sqrt{n})$. Come si rappresentano le istruzioni di un programma? Se sono in serie:

$$\begin{array}{ll} I_1 & c_1(n) \\ I_2 & c_2(n) \\ \vdots & \\ I_l & c_l(n) \end{array}$$

Dove la complessità totale allora sarà:

$$\sum_{i=1}^{l} c_i(n)$$

oppure all'interno di un costrutto **if-else:**

$$\begin{array}{cc} \text{if cond} & c_{\text{cond}}(n) \\ I_2 & c_1(n) \\ \text{else} & \\ I_l & c_2(n) \end{array}$$

In questo caso, come faccio a sapere la complessità totale di questi istruzioni? Per capirlo, studiamo il caso nel *worst case scenario* ovvero nel caso peggiore. A volte vengono mostrati anche i casi migliore ma di solito si calcola il tempo peggiore. Quindi ci interessa specialmente il *limite superiore* dell'algoritmo, quindi piuttosto che dire che la complessità è esattamente **uguale** questo numero, invece noi diremo che è **minore o uguale** del numero calcolato.

$$C(n) = c_{\text{cond}}(n) + max(c_1(n), c_2(n))$$

Mentre per un while loop?

$$\begin{array}{cc} \text{while cond} & c_{cond}(n) \\ I & c_0(n) \end{array}$$

Sia m limite superiore numero di iterazioni che esegue l'algoritmo. La complessità di questo algoritmo quindi sarà:

$$C(n) = c_{cond}(n) + m(c_{cond}(n) + c_0(n))$$

Proviamo ora a calcolare la molteplicazioni tra matrici. Siano due matrici A e B rispettivamente con dimensione n x m e m x l.

```
For i <- 1 to n
    For j <- 1 to l
        C[i][j] <- 0
        For k <- 1 to m
            C[i][j] += A[i][j] * B[k][j]
```

Avere un risultato del tipo $5m + 1$ (riguardo il for più interno) è inutile perché non ci da informazioni realmente utili sulla complessità dell'algoritmo. Ad ogni modo contando tutti i for, avremo un risultato del tipo:

$$n(5ml + 4l + 2) + n + 1 = 5nml + 4nl + 3n + 1$$

Ci sono alcuni elementi in questa operazioni che non influiscono realmente sul risultato finale. Indi per cui, possiamo anche ometterlo all'interno del calcolo della complessità di un algoritmo. In realtà ciò che ci interessa realmente nel risultato finale è:

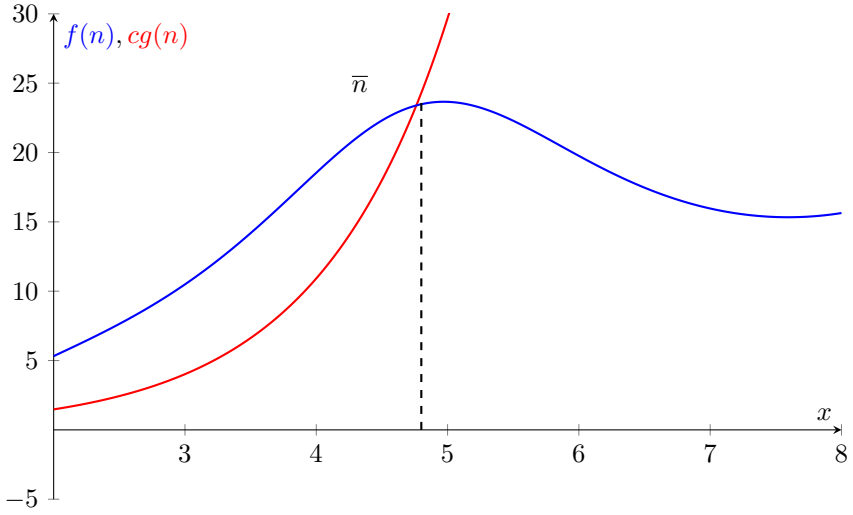
$$5\,\mathbf{nml} + 4nl + 3n + 1$$

Infatti nml è capace di dirci tutto sulla complessità dell'algoritmo e da cosa dipende. Quando si parla di ordine di grandezza si parla in realtà del comportamento asintotico della funzione calcolata.

Definition 1.1.

$$f \in O(g) \iff \exists c > 0,\ \exists \overline{n},\ \forall n \geq \overline{n} \,|\, f(n) \leq cg(n)$$

Figure 1: $f \in O(g)$

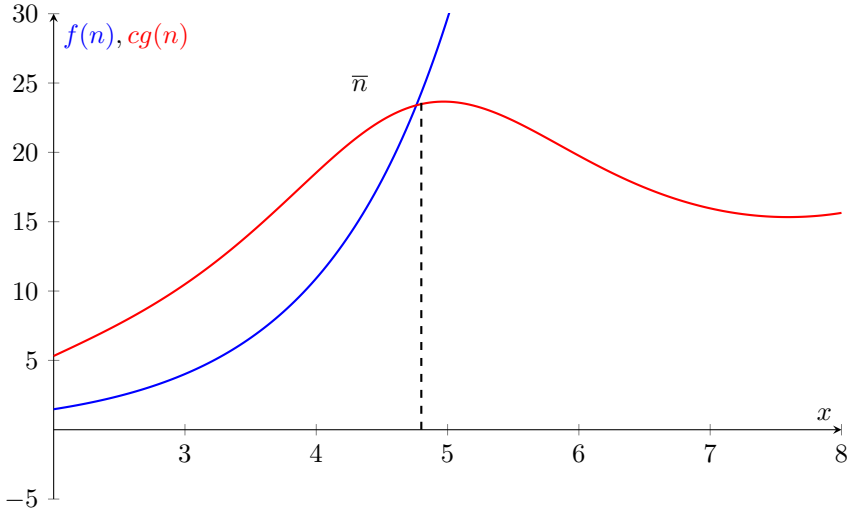


Definition 1.2.

$$f \in \Omega(g) \iff \exists c > 0,\ \exists \overline{n},\ \forall n \geq \overline{n} \,|\, f(n) \geq cg(n)$$

Che rispettivamente è l'inverso della funzione O grande.

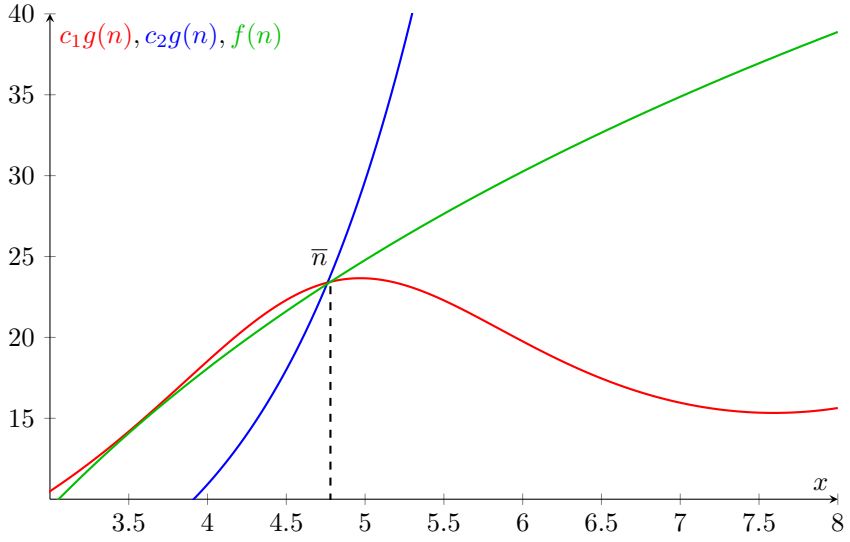
Figure 2: $f \in \Omega(g)$



Definition 1.3.

$$f \in \Theta(g) \iff f \in O(g) \ \wedge \ f \in \Omega(g)$$

Figure 3: $f \in \Theta(g)$



Esempio 1

Questa affermazione vale?

$$n \in O(2n)$$

Sì, perché esiste un c che soddisfa $n \leq c2n$.

Quest'altra affermazione vale?

$$2n \in O(n)$$

Sì perché esiste un c che soddisfa $2n \leq 2n$.

Esempio 2

Questa affermazione vale?

$$f \in O(g) \iff g \in \Omega(g)$$

Proviamo a dimostrare. Sappiamo che esiste un c e $\overline{n}$ tale che $\forall n > \overline{n}$ $f(n) \leq cg(n)$. Isoliamo la g:

$$g(n) \geq \frac{1}{c} f(n)$$

Esiste quindi un c' che soddisfa la disuguaglianza.

$$c' = \frac{1}{c}$$

Esempio 3

$$f_1 \in O(g), f_2 \in O(g) \Longrightarrow f_1 + f_2 \in O(g)$$

Dobbiamo dimostrare che esiste c_1 e c_2, $\overline{n}_1$ e $\overline{n}_2$ tali che:

$$\forall n > \overline{n}_1 \mid f_1(n) \leq c_1 g(n)$$

$$\forall n > \overline{n}_2 \mid f_2(n) \leq c_2 g(n)$$

Quindi

$$\overline{n} = max(\overline{n}_1, \overline{n}_2)$$

$$f_1(n) + f_2(n) \leq (c_1 + c_2) g(n)$$

Esempio 4

$$f_1 \in O(g_1), f_2 \in O(g_2) \mid f_1 f_2 \in O(g_1 g_2)$$

Quindi esiste c_1 e c_2, $\overline{n}_1$ e $\overline{n}_2$ tali che:

$$\forall n > \overline{n}_1 \mid f_1(n) \leq c_1 g(n)$$

$$\forall n > \overline{n}_2 \mid f_2(n) \leq c_2 g(n)$$

$$f_1(n) f_2(n) \leq c_1 g_1(n) c_2 g_2(n)$$

$$f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$$

Quindi:

$$c = c_1 c_2$$

$$\overline{n} = max(\overline{n}_1, \overline{n}_2)$$

1.3 Ordini di grandezza per le funzioni

L'algoritmo di ricerca A termina entro n. Immaginiamo che l'algoritmo A sia il seguente:

```
For i <- 0 to length(a) - 1
    if a[i] = x
        ret i
ret -1
```

Capiamo che la sua complessità è uguale a:

$$A \in O(n)$$

Per appurarci che la stima di complessità sia accurata dobbiamo controllare che $A \in \Omega(g)$ vuol dire che esiste uno schema di input tale per cui se $g(n)$ è il numero di passi necessari per risolvere l'istanza n, allora $g \in \Omega(f)$. Se riusciamo a fare questo allora $g \in \Theta(f)$ e quindi possiamo dire che la stima è accurata.

$P \in O(f)$ vuol dire che il problema P riesco a risolvere in tempo f. Supponiamo per assurdo che esista un algoritmo riesca a capire se l'elemento si

trova nell'array o no. Possiamo capire velocemente che per contraddizione, essendo che esiste almeno un elemento dove l'algoritmo non ha guardato, siamo certi del malfunzionamento dell'algoritmo e quindi non è vero che esiste una stima di complessità più bassa di f.

Ora andiamo ad affrontare i tipi di algoritmi che si definiscono di "ordinamento".

Definition 1.4. Input: Sequenza $(a_1, ..., a_n)$ di oggetti su cui è definita una relazione di ordinamento (in questo caso ordinamento per confronti).

Output: Permutazione $(a'_1, ..., a'_n)$ di $(a_1, ..., a_n)$ t.c. $\forall\, i > j,\ \ a'_i \leq a'_j$.

Andiamo ora a vedere i diversi tipi di algoritmi di ordinamento.

1.4 insertion_sort (A)

In questo caso la j sarà chiamata variabile "invariante" ovvero che mantiene una proprietà che continua a valere nel run-time dell'algoritmo. In questo caso, la j è invariante perchè tutti gli oggetti "a sinistra" di essa saranno considerati già ordinati. (Parte da 2 perché abbiamo deciso che la posizione all'interno dell'array parte da 1).

```
for j <- 2 to length[A]
    key <- A[j]
    i <- j - 1
    while i > 0 and A[i] > key
        A[i+1] <- A[i]
        i--
    a[i+1] <- key
```

Ora dobbiamo capire quale sia la complessità di questo algoritmo. Se le cose vanno bene, l'algoritmo potrebbe terminare in $O(n)$. Tuttavia, seppur giusto, non è preciso. Infatti, nel peggiore dei casi, l'algoritmo termina in $O(n^2)$. Infatti il caso peggiore di input che mi può arrivare è un array ordinato al contrario con complessità uguale a:

$$1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Questo algoritmo, quanto spazio di memoria usa in più rispetto allo spazio occupato dai dati?

Lo spazio di memoria di questo algoritmo rimane costante a prescindere dalla dimensione del problema. Un algoritmo del genere si dice che ***ordina in loco*** se la quantità di memoria extra è costante. Si parla di ***stabilità*** di un algoritmo di ordinamento quando l'ordine relativo di elementi uguali non viene scambiato dall'algoritmo. Quindi:

$$\text{Se } a_i = a_j \, i < j, \text{mantiene l'ordinamento}$$

2 Concetto di "Divide et impera"

2.1 Fattoriale e funzioni ricorsive

```
Fatt(n)
    if n = 0
        ret 1
    else
        ret n * fatt(n - 1)
```

L'argomento della funzione ci fa capire la complessità dell'algoritmo:

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

Con problemi ricorsivi si avrà una complessità con funzioni definite ricorsivamente. Questo si risolve induttivamente:

$$\begin{aligned} T(n) &= 1 + T(n-1) \\ &= 1 + 1 + T(n-2) \\ &= 1 + 1 + 1 + T(n-3) \\ &= \underbrace{1 + 1 + \ldots + 1}_{i} + T(n-i) \end{aligned}$$

La condizione di uscita è: $n - i = 0 \quad n = i$

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

Questo si chiama passaggio iterativo.

Esempio 1

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

Questa funzione si può riscrivere come:

$$T(n) = \begin{cases} \text{Costante} & \text{se } n < a \\ 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n & \text{se } n \geq a \end{cases}$$

Se la complessità fosse già data bisognerebbe soltanto verificare se è corretta. Usando il metodo di sostituzione:

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

sostituiamo nella funzione di partenza:

$$\begin{aligned} T(n) &= 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \\ &\leq 2c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \log \left\lfloor \frac{n}{2} \right\rfloor + n \\ &\leq \not{2} c \frac{n}{\not{2}} \log \frac{n}{2} + n \\ &= cn \log n - cn \log 2 + n \\ &\overset{?}{\leq} cn \log n \quad \text{se } n - cn \log 2 \leq 0 \end{aligned}$$

$$c \geq \frac{n}{n \log 2} = \frac{1}{\log 2}$$

Il metodo di sostituzione dice che quando si arriva ad avere una disequazione corrispondente all'ipotesi, allora la soluzione è corretta se soddisfa una certa ipotesi.

Esempio 2

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \quad \in O(n)$$

$$T(n) \leq cn$$

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + c\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &= c\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &= cn + 1 \overset{?}{\leq} cn \end{aligned}$$

Il metodo utilizzato non funziona perchè rimane l'1 e non si può togliere in alcun modo. Per risolvere questo problema bisogna risolverne uno più forte:

$$T(n) \leq cn - b$$

$$\begin{aligned} T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 1 \\ &\leq c\left(\left\lfloor \frac{n}{2} \right\rfloor\right) - b + c\left(\left\lceil \frac{n}{2} \right\rceil\right) - b + 1 \\ &= c\left(\left\lfloor \frac{n}{2} \right\rfloor + \left\lceil \frac{n}{2} \right\rceil\right) - 2b + 1 \\ &= cn - 2b + 1 \overset{?}{\leq} cn - b \\ &= \underbrace{cn - b}_{} + \underbrace{1 - b}_{\leq 0} \leq cn - b \quad \text{se } b \geq 1 \end{aligned}$$

Se la proprietà vale per questo problema allora vale anche per il problema iniziale perchè è meno forte.

Esempio 3

$$\begin{aligned}
T(n) &= 3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right) + n \\
&= n + 3T\left(\left\lfloor\frac{n}{4}\right\rfloor\right) \\
&= n + 3\left(\left\lfloor\frac{n}{4}\right\rfloor + 3T\left(\left\lfloor\frac{\left\lfloor\frac{n}{4}\right\rfloor}{4}\right\rfloor\right)\right) \\
&= n + 3\left\lfloor\frac{n}{4}\right\rfloor + 3^2T\left(\left\lfloor\frac{n}{4^2}\right\rfloor\right) \\
&\leq n + 3\left\lfloor\frac{n}{4}\right\rfloor + 3^2\left(\left\lfloor\frac{n}{4^2}\right\rfloor + 3T\left(\left\lfloor\frac{\left\lfloor\frac{n}{4^2}\right\rfloor}{4}\right\rfloor\right)\right) \\
&= n + 3\left\lfloor\frac{n}{4}\right\rfloor + 3^2\left\lfloor\frac{n}{4^2}\right\rfloor + 3^3T\left(\left\lfloor\frac{n}{4^3}\right\rfloor\right) \\
&= n + 3\left\lfloor\frac{n}{4}\right\rfloor + \ldots + 3^{i-1}\left\lfloor\frac{n}{4^{i-1}}\right\rfloor + 3^iT\left(\left\lfloor\frac{n}{4^i}\right\rfloor\right)
\end{aligned}$$

Per trovare il caso base poniamo l'argomento di T molto piccolo:

$$\begin{aligned}
\frac{n}{4^i} &< 1 \\
4^i &> n \\
i &> \log_4 n
\end{aligned}$$

L'equazione diventa:

$$\leq n + 3\left\lfloor\frac{n}{4}\right\rfloor + \ldots + 3^{\log_4 n - 1}\left\lfloor\frac{n}{4^{\log_4 n - 1}}\right\rfloor + 3^{\log_4 n}c$$

Si può togliere l'approssimazione per difetto per ottenere un maggiorante:

$$\begin{aligned}
&\leq n\left(1 + \frac{3}{4} + \left(\frac{3}{4}\right)^2 + \ldots + \left(\frac{3}{4}\right)^{\log_4 n - 1}\right) + 3^{\log_4 n}c \\
&\leq n\left(\sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i\right) + c3^{\log_4 n}
\end{aligned}$$

Per capire l'ordine di grandezza di $3^{\log_4 n}$ si può scrivere come:

$$3^{\log_4 n} = n^{\left(\log_n 3^{\log_4 n}\right)} = n^{\log_4 n \cdot \log_n 3} = n^{\log_4 3}$$

Quindi la complessità è:

$$= O(n) + O(n^{\log_4 3})$$

Si ha che una funzione è uguale al termine noto della funzione originale e l'altra che è uguale al logaritmo dei termini noti. Se usassimo delle variabili uscirebbe:

$$\begin{aligned}
T(n) &= aT\left(\left\lfloor\frac{n}{b}\right\rfloor\right) + f(n) \\
&= O(f(n)) + O(n^{\log_b a})
\end{aligned}$$

2.2 Master Theorem o Teorema dell'esperto

Data una relazione di occorrenza di questa forma:

$$T(n) = aT\left(\left\lfloor \frac{n}{b} \right\rfloor\right) + f(n)$$

Distinguiamo tre casi:

1. $$f(n) \in O(n^{\log_b a - \epsilon}) \Longrightarrow T(n) \in \Theta(n^{\log_b a})$$

2. $$f(n) \in \Theta(n^{\log_b a}) \Longrightarrow T(n) \in \Theta(f(n) \log n)$$

3. $$f(n) \in \Omega(n^{\log_b a + \epsilon}) \Longrightarrow T(n) \in \Theta(f(n))$$

Esempio 1

$$T(n) = 9T\left(\frac{n}{3}\right) + n$$

$$a = 3, b = 3, f(n) = n$$

Basta che trovo un ϵ che mi dia n.

$$n^{\log_b a} = n^{\log_3 9} = n^2 * n^{-\frac{1}{2}}$$

In questo caso $\epsilon = n^{-\frac{1}{2}}$ e ci troviamo nel **PRIMO CASO** e la soluzione è $T(n) \in \Theta(n^2)$.

Esempio 2

$$T(n) = T\left(\frac{2n}{3}\right) + 1$$

$$a = 1, b = \frac{3}{2}, f(n) = n^0$$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0$$

Ci troviamo nel **SECONDO CASO** e la soluzione è $T(n) \in \Theta(\log n))$

Esempio 3

$$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$$

$$a = 3, b = 4, f(n) = n \log n$$

Ci troviamo nel **TERZO CASO** quindi basta qualsiasi valore di ϵ basta che sia contenuto tra $\log_3 4 \leq \epsilon \leq 1$. La soluzione è $T(n) \in \Theta(n \log n)$.

Esempio 4

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

$$a = 2, b = 2, f(n) = n\log n$$

$$n\log n \in \Omega(n^{1+\overset{?}{\epsilon}})$$

$$\log n \in \Omega(n^{\epsilon}) \text{ NON VALE}$$

Poiché un logaritmo è sempre più piccolo di un polinomio. Questo è un caso dove il teorema *non* si applica

2.3 Merge Sort (A, n)

Questo algoritmo di ordinamento *ricorsivo* utilizza il concetto di *divide et impera*.

Concettualmente, un merge sort funziona come segue:

1. **Dividi** l'array non ordinato in n sottoarray, ognuno contenente un elemento (un array di un elemento è considerato ordinato).
2. **Unisci** ripetutamente i sottoarray per produrre nuovi sottoarray ordinati finché non ne rimane solo uno. Questo sarà l'array ordinato.

La sua complessità considerando il merge con complessità lineare risulta:

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

Utilizzando il **Master Theorem** e cadendo del *secondo caso* possiamo confermare che il risultato è:

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

```
// A e' l'array mentre p ed r sono rispettivamente l'indice di
    partenza e di arrivo
mergeSort(A, p, r) // O(n log n)
    if (p < r)
        q <- floor((p+r)/2)
        mergeSort(A, p, q)
        mergeSort(A, q+1, r)
        merge(A, p, q, r)
```

```
merge(A, p, q, r)
    i <- 1
    j <- p
    k <- q+1
    // Ordina gli elementi di A in B
    // O il lato sinistro ha finito
    while(j <= q or k <= r) // O(n)
        if j <= q and (k > r or A[j] <= A[k])
            B[i] <- A[j]
            j++
        else
            B[i] <- A[k]
            k++
        i++
```

```
    // Copia gli elementi di B in A
    for i <- 1 to r-p+1  // O(n)
        A[p+i-1] <- B[i]
```

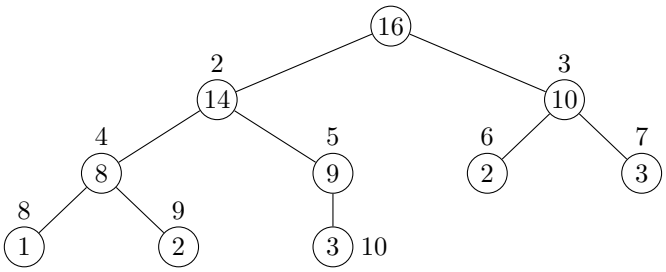
L'algoritmo è **stabile** poiché non vengono scambiati gli elementi uguali. Tuttavia non ordina **in loco** poiché utilizza uno spazio di memoria aggiuntivo.

2.4 Heap

L'Heap è un albero semicompleto (ogni nodo ha 2 figli ad ogni livello tranne l'ultimo che è completo solo fino ad un certo unto) in cui i nodi contengono oggetti con relazione di ordinamento.

Proprietà Heap:

$$\forall \text{ nodo, il contenuto è } \geq \text{ del contenuto dei figli}$$



La complessità dell'algoritmo è in base al numero di livelli dell'albero.

$\longrightarrow$ albero con n livelli:

$$\#\text{Nodi} = 2^0 + 2^1 + 2^2 + ... + 2^{n-1} = \frac{1-2^n}{1-2} = 2^n - 1$$

$\longrightarrow$ albero con n nodi:

$$\#\text{Livelli} = \log_2 n$$
$$\#\text{Foglie} = \frac{n}{2}$$

Le foglie di un albero sono la metà dei nodi dell'albero.

```
extractMax(H) // O(log n)
    h[1] <- H[H.heap_size()]
    H.heap_size()--
    heapify(H, 1)
```

```
heapify(H, 1) // O(log n)
    l <- left[i] //
    r <- right[i]
    if(l < h.heap_size() AND H[l] > H[i]
        largest <- l
    else
        largest <- i
    if m <= h.heap_size() and H[r] > H[largest]
        largest <- r
    if largest != i
        swap(H[i], H[largest])
        heapify(H, largest)
```

Creare uno heap da un array:

```
buildHeap(A)
    A.heap_size() <- length[A]
        for i <- length[A]/2 down to 1
            heapify(A, i)
```

Immagino tutte le foglie come heap con un solo nodo. L'indice del primo nodo che non è heap corrisponde a $\frac{length(A)}{2}$ su questo allora chiamo `heapify`.

2.5 Heapsort

```
HeapSort(A) // O(n log n)
    buildHeap(A)
    for i <- length(A) to 1
        scambia(A[1], A[i])
        heapsize(A)--
        heapify(a, i)
```

L'Heap Sort è un algoritmo che lavora **in loco** tuttavia non è **stabile**. Tuttavia riusciamo a fare una stima migliore e più corretta?

$$n \log i = \sum_{i=1}^{n} \log i = \log \prod_{i=1}^{n} i = \log n! = \Theta(\log n^n) = \Theta(n \log n)$$

In questo caso, essere accurati non aiuta, ma abbiamo avuto la certezza che non esiste una stima migliore.

2.6 Quicksort

Come funziona l'algoritmo?

1. Dividi prima l'array in due parti. (Partizione)
2. Devi essere sicuro che tutti gli elementi di sinistra siano $\leq$ di quelli di destra. Ricorsivamente ordina la parte sinistra e la parte destra.
3. A questo punto l'array è ordinato.

```
quickSort(A, p, r)
    if (p < r)
        q <- partition(a, p, r)
        quickSort(A, p, q)
        quickSort(A, q+1, r)
```

```
partition(A, p, r) // O(n)
    x <- A[p] // Elemento Perno
    i <- p-1
    k <- r+1
    while true
        repeat j-- until a[j] <= x
        repeat i++ until a[i] >= x
        if i < j
            scambia(a[i], a[j])
        else
            ret j
```

Scegliamo un elemento a caso in base a quello comparato rispetto all'elemento perno tale che:

$$sx \leq perno \leq dx$$

Questo algoritmo non è **stabile** ma lavora **in loco**. La sua complessità?

$$T(n) = T(\text{partition}) + T(q) + T(n - q)$$

Se il quicksort è perfettamente diviso in due, allora la sua complessità è $O(n \log n)$. Se invece l'array è già ordinato la sua equazione di ricorrenza sarà:

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

Tuttavia non ci aspettiamo che questo caso sia frequente e quindin nella stra grande maggioranza dei casi allora:

$$T(n) = n + T(cn) + T((1-c)n)$$

Un equazione di questo tipo sappiamo che ha come complessità $\Theta(n \log n)$.

```
rand_Partition(A, p, r)
    i <- rand(p .. r) // Ora l'elemento perno e' un elemento a caso
    scambio(A[p], A[i]
    ret partition(A, p, r)
```

$$\begin{aligned} T(n) = n + \frac{1}{n}(T(1) + T(n-1) + \frac{1}{n}(T(2) + (T-2)) + \ \ldots \ + \\ \frac{1}{n}(T(n-2) + T(2)) + \frac{1}{n}(T(n-1) + T(1) = \\ = n + \frac{1}{n}\sum_i (T(i) + T(n-i)) \\ = n + \frac{2}{n}\sum_i T(i) \in O(n \log n) \end{aligned}$$

Qualsiasi algoritmo che *lavora per confronti* deve fare almeno $O(n \log n)$.

3 Algoritmi di ordinamento in tempo lineare

3.1 Algoritmi non basati su confronti

3.1.1 Counting Sort

Tuttavia possiamo trovare algoritmi che come tempo di esecuzione hanno tempo lineare. Come? Non lavorando a **confronti**.

Come ordinare n numeri con valori da 1 a k?

```
countingSort(A, k)
    for i <- 1 to k
        C[i] <- 0
    for j <- 1 to len(A)
        C[A[j]]++
    for i <- 2 to k
        C[i] <- C[i-1]+C[i]
    for j <- len(A) down to 1
        B[C[A[j]]] <- a[j]
        C[A[J]]--
```

La complessità di questo algoritmo è $O(n+k)$ dove n è la lunghezza dell'array e k e il range di valori.

3.1.2 Radix Sort

Il radix sort è un algoritmo di ordinamento che ordina gli elementi confrontando i singoli bit. Quello che fa è ordinare per la cifra meno significativa, poi per la seconda cifra meno significativa e così via.

```
radixSort(A, d) // O(d(n+k))
  for i <- 1 to d
      countingSort(A, n)
```

La complessità di questo algoritmo è

$$\Theta(d(n+k))$$

dove d è il numero di cifre e k è il range di valori. Se si vuole invece ordinare n valori da 1 a n^2-1, le costanti nascoste all'interno del Θ sono molto alte e quindi non è un algoritmo efficiente. Tuttavia si posson rappresentare i numeri in base n e quindi ottenere un algoritmo lineare.

3.1.3 Bucket Sort

Il Bucket Sort è un algoritmo di ordinamento che funziona bene quando i dati sono distribuiti uniformemente. Quindi su un array di n elementi **distribuiti uniformemente** su $[0,1)$, si può dividere l'intervallo in n sottointervalli con probabilità $\frac{1}{n}$ e poi ordinare i singoli sottointervalli chiamate anche *"Bucket"*. Infatti se i dati sono distribuiti uniformemente allora la complessità dell'algoritmo è lineare:

$$\Theta(n)$$

Questo perché in ogni bucket ci si aspetta un valore costante e quindi indipendente dal valore di n.

Il caso pessimo però è quando tutti gli elementi ricadono nello stesso bucket. La probabilità che questo accada è molto bassa infatti è:

$$\underbrace{\frac{1}{n} * \frac{1}{n} * \cdots * \frac{1}{n}}_{n-1} = \frac{1}{n^{n-1}}$$

e la sua complessità diventa:

$$O(n^2)$$

Sia X_{ij} la variabile aleatoria che vale:

$$\begin{cases} 1 & \text{Se l'elemento } i \text{ va nel bucket } j \\ 0 & \text{altrimenti} \end{cases}$$

Per esprimere il numero di elementi nel bucket j si ha:

$$N_j = \sum_i X_{ij}$$

La complessità di questo algoritmo quindi può essere espressa come:

$$C = \sum_j {N_j}^2$$

Dove il valore atteso è:

$$E[C] = E\left[\sum_j {N_j}^2\right] = \sum_j E[{N_j}^2] = \sum_j \left(Var(N_j) - E[N_j]^2\right)$$

Dove $E[N_j]$ è:

$$E[N_j] = \sum E[X_{ij}] = \sum_{j=1}^{n} \frac{1}{n} = 1$$

$$Var[N_j] = \sum Var(X_{ij}) = \sum \frac{1}{n} * \left(1 - \frac{1}{n}\right) = 1 - \frac{1}{n}$$

E quindi possiamo svolgere il calcolo precedente dove:

$$\begin{aligned}\sum_j \left(Var(N_j) - E[N_j]^2\right) &= \sum_j \left(\left(1 - \frac{1}{n}\right) + 1\right) \\ &= \sum_j 2 - \frac{1}{n} \\ &= 2n - 1\end{aligned}$$

Le distribuzioni possono essere arbitrarie ma basta che tutti i bucket abbiano la stessa probabilità. Prendiamo:

$$n_1, ..., n_2, ..., n_l$$

$$\frac{1}{n_1}, \ldots, \frac{1}{n_2}, \ldots, \frac{1}{n_l}$$

La *turing-riduzione* è un algoritmo che riduce un problema ad un altro problema.

4 Algoritmi di selezione

Dato in input un array A di oggetti su cui è definita una relazione di ordinamento e un indice i compreso tra 1 e n (n è il numero di oggetti nell'array), l'output dell'algoritmo è l'oggetto che si trova in posizione i nell'array ordinato.

```
selezione(A, i)
  ordina(A) // O(n log n)
  return A[i]
```

Quindi la complessità di questo algoritmo nel caso peggiore è $O(n \log n)$ (limite superiore). È possibile selezionare un elemento in tempo lineare? Analizziamo un caso particolare dell'algoritmo di selezione, ovvero la ricerca del minimo (o del massimo).

4.1 Ricerca del minimo o del massimo

In tempo lineare si può trovare il minimo e il massimo di un array:

```
minimo(A)
  min <- A[1]
  for i <- 2 to length[A]
    if A[i] < min
      min <- A[i]
  return min
```

trovare il minimo equivale a trovare `selezione(A, 1)` e trovare il massimo equivale a trovare `selezione(A, n)`. Si può però andare sotto la complessità lineare?

Per trovare il massimo (o il minimo) elemento n di un array bisogna fare **almeno** $n-1$ confronti perchè bisogna confrontare ogni elemento con l'elemento massimo (o minimo) trovato per poter dire se è il massimo (o minimo). Di conseguenza, non è possibile avere un algoritmo per la ricerca del massimo (o minimo) in cui c'è un elemento che non "perde" mai ai confronti (cioè risulta sempre il più grande) e non viene dichiarato essere il più grande (o più piccolo).

Dimostrazione: Per dimostrarlo si può prendere un array in cui l'elemento a non perde mai ai confronti, ma l'algoritmo dichiara che il massimo è l'elemento b. Allora si rilancia l'algoritmo sostituendo l'elemento a con $a =$ `max(b+1,a)` e si ripete l'algoritmo con questo secondo array in cui a è l'elemento più grande. Si ha quindi che i confronti in cui a non è coinvolto rimangono gli stessi e i confronti in cui a è coinvolto non cambiano perchè anche prima a non perdeva mai ai confronti, di conseguenza l'algoritmo dichiarerà che il massimo è b e quindi l'algoritmo non è corretto, dimostrando che non esiste un algoritmo che trova il massimo in meno di $n-1$ confronti.

Abbiamo quindi trovato che la complessità del massimo (o minimo) nel caso migliore è $\Omega(n)$ (limite inferiore) e nel caso peggiore è $O(n)$ (limite superiore). Di conseguenza la complessità è $\Theta(n)$.

4.1.1 Ricerca del minimo e del massimo contemporaneamente

Si potrebbe implementare unendo i 2 algoritmi precedenti:

```
min_max(A)
  min <- A[1]
  max <- A[1]
  for i <- 2 to length[A]
    if A[i] < min
      min <- A[i]
    if A[i] > max
      max <- A[i]
  return (min, max)
```

Questo algoritmo esegue $n-1+n-1 = 2n-2$ confronti.

- **Limite inferiore**: Potenzialmente ogni oggetto potrebbe essere il minimo o il massimo. Sia m il numero di oggetti potenzialmente minimi e M il numero di oggetti potenzialmente massimi. Sia n il numero di oggetti nell'array.
  - All'inizio $m+M = 2n$ perchè ogni oggetto può essere sia minimo che massimo.

– Alla fine $m + M = 2$ perchè alla fine ci sarà un solo minimo e un solo massimo.

Quando viene fatto un confronto $m + M$ può diminuire.

– Se si confrontano due oggetti che sono potenzialmente sia minimi che massimi, allora $m + M$ diminuisce di 2 perchè:

$$a < b$$

b non può essere il minimo e a non può essere il massimo e si perdono 2 potenzialità.

– Se si confrontano due potenziali minimi (o massimi), allora $m + M$ diminuisce di 1 perchè:

$$a < b$$

b non può essere il minimo e si perde 1 potenzialità.

Un buon algoritmo dovrebbe scegliere di confrontare sempre 2 oggetti che sono entrambi potenziali minimi o potenziali massimi.

Due oggetti che sono potenzialmente sia minimi che massimi esistono se $m+M > n+1$ perchè se bisogna distribuire n potenzialità ne avanzano due che devono essere assegnate a due oggetti che hanno già una potenzialità. Quindi fino a quando $m + M$ continua ad essere almeno $n + 2$ si riesce a far diminuire $m + M$ di 2 ad ogni confronto.

Questa diminuzione si può fare $\lfloor \frac{n}{2} \rfloor$ volte, successivamente $m + M$ potrà calare solo di 1 ad ogni confronto.

Successivamente il numero di oggetti rimane:

$$\begin{cases} n+1 & \text{se } n \text{ è dispari} \\ n & \text{se } n \text{ è pari} \end{cases}$$

– n dispari:

$$\begin{aligned} & n + 1 - 2 + \left\lfloor \frac{n}{2} \right\rfloor \\ &= n - 1 + \left\lfloor \frac{n}{2} \right\rfloor \\ &= \left\lfloor \frac{3}{2} n \right\rfloor - 1 \\ &= \left\lceil \frac{3}{2} n \right\rceil - 2 \end{aligned}$$

– n pari:

$$\begin{aligned} & n - 2 + \left\lfloor \frac{n}{2} \right\rfloor \\ &= n - 2 + \frac{n}{2} \\ &= \frac{3}{2} n - 2 \\ &= \left\lceil \frac{3}{2} n \right\rceil - 2 \end{aligned}$$

Quindi la complessità è $\Omega(\lceil \frac{3}{2}n \rceil - 2) = \Omega(n)$ (limite inferiore). Meglio di così non si può fare, ma non è detto che esista un algoritmo che raggiunga questo limite inferiore.

Un algoritmo che raggiunge il limite inferiore è il seguente:

1. Dividi gli oggetti in 2 gruppi:

$$\underbrace{\underbrace{\begin{matrix} a_1 \\ a_2 \\ \vdots \\ a_{\lfloor \frac{n}{2} \rfloor} \end{matrix}}_{\text{Potenziali minimi}} \quad \underbrace{\begin{matrix} b_1 \\ b_2 \\ \vdots \\ b_{\lceil \frac{n}{2} \rceil} \end{matrix}}_{\text{Potenziali massimi}}}_{\text{Potenziali sia minimi che massimi}}$$

2. Confronta a_i con b_i, supponendo $a_i < b_i$ (mette a sinistra i più piccoli e a destra i più grandi). Una volta aver fatto il confronto possiamo swappare gli elementi nella loro apposita sezione.
3. Cerca il minimo degli a_i e cerca il massimo dei b_i:
4. Sistema l'eventuale elemento in più (se l'array è dispari)

4.2 Randomized select

Si può implementare un algoritmo che divide l'array in 2 parti allo stesso modo in cui viene effettuata la `partition` di quick sort:

```
// A: Array
// p: Indice di partenza
// r: Indice di arrivo
// i: Indice che stiamo cercando (compreso tra 1 e r-p+1)
randomized_select(A, p, r, i)
  if p = r
    return A[p]
  q <- randomized_partition(A, p, r)
  k <- q - p + 1 // Numero di elementi a sinistra
  // Controlla se l'elemento cercato e' a sinistra o a destra
  if i <= k
    return randomized_select(A, p, q, i) // Cerca a sinistra
  else
    return randomized_select(A, q+1, r, i-k) // Cerca a destra
```

- Se dividessimo sempre a metà si avrebbe:

$$T(n) = n + T\left(\frac{n}{2}\right) = \Theta(n) \text{ (terzo caso del teorema dell'esperto)}$$

- Mediamente:

$$\begin{aligned} T(n) &= n + \frac{1}{n}T\left(max(1, n-1)\right) + \frac{1}{n}T\left(max(2, n-2)\right) + \dots \\ &= n + \frac{2}{n}\sum_{i=\frac{n}{2}}^{n-1} T\left(i\right) \end{aligned}$$

La complessita media è lineare.

Si esegue un solo ramo, che nel caso pessimo è quello con più elementi. La risoluzione è la stessa del quick sort.

Esiste un algoritmo che esegue la ricerca in tempo lineare anche nel caso peggiore?

Si potrebbe cercare un elemento perno più ottimale, cioè che divida l'array in **parti proporzionali**:

1. Dividi gli oggetti in $\left\lfloor \frac{n}{5} \right\rfloor$ gruppi di 5 elementi più un eventuale gruppo con meno di 5 elementi.
2. Calcola il mediano di ogni gruppo di 5 elementi (si ordina e si prende l'elemento centrale). $\Theta(n)$
3. Calcola ricorsivamente il mediano x dei mediani
$$T\left(\left\lceil \frac{n}{5} \right\rceil\right)$$
4. Partiziona con perno x e calcola k (numero di elementi a sinistra). $\Theta(n)$
5. Se $i < k$ cerca a sinistra l'elemento i, altrimenti cerca a destra l'elemento $i - k$. La chiamata ricorsiva va fatta su un numero di elementi sufficientemente piccolo, e deve risultare un proporzione di n, quindi ad esempio dividere in gruppi da 3 elementi non funzionerebbe.

$$T(?)$$

$$\begin{array}{ccccccccccccc} m_1 & \to & m_2 & \to & m_3 & \to & \underset{x}{m_4} & \to & m_5 & \to & m_6 & \to & m_7 \\ & & & & & & & & \downarrow & & \downarrow & & \downarrow \\ & & & & & & & & m_{5,4} & & m_{6,4} & & m_{7,4} \\ & & & & & & & & \downarrow & & \downarrow & & \\ & & & & & & & & m_{5,5} & & m_{6,5} & & \end{array}$$

Gli elementi verdi sono maggiori dell'elemento x e ogni elemento verde avrà 2 elementi maggiori di esso (tranne nel caso del gruppo con meno di 5 elementi rappresentato in blu).

$$3 \cdot \left(\underbrace{\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil}_{\text{verdi + blu + rosso}} - \underbrace{2}_{\text{rosso + blu}} \right) = \frac{3}{10}n - 6$$

Da ogni parte si hanno almeno $\frac{3}{10}n - 6$ elementi, quindi al massimo si hanno $n - \left(\frac{3}{10}n - 6\right) = \frac{7}{10}n + 6$ elementi.

Quindi abbiamo trovato $T(?)$:

$$T(n) = \Theta(n) + T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 6\right)$$

Applichiamo il metodo di sostituzione $T(n) \leq cn$:

$$T(n) \leq n + c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7}{10}n + 6\right)$$

Tuttavia non sappiamo se $\left(\frac{7}{10}n + 6\right)$ sia $< n$. Scopriamo che la disuguaglianza è vera solo se $n > 20$. Quindi per risolvere il problema ci basta scegliere un $\overline{n} > 20$. Le costanti quindi sono abbastanza alte.

$$\begin{aligned}
&\leq n + c + n\frac{c}{5} + \frac{7}{10}cn + 6c \\
&= \frac{9}{10}cn + 7c + n \\
&\overset{?}{\leq} cn \\
&= cn + \left(-\frac{1}{10}cn + 7c + n\right) \leq cn \\
&\overset{\text{sse}}{\iff} \left(n + 7c - \frac{1}{10}cn\right) \leq 0
\end{aligned}$$

Dove l'equazione di ricorrenza diventa:

$$T(n) \leq \Theta(n) + c\left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7}{10} + 6\right)$$

Quindi $T(n) \leq cn$ e quindi $T(n) \in \Theta(n)$. Quindi è un algoritmo ottimo. Il problema è che le costanti sono così alte che nella pratica è meglio il `randomized_select`.

Esistono modi per strutturare meglio le informazioni nel calcolatore per trovare l'elemento cercato in tempo $O(\log n)$? Dobbiamo trovare delle rappresentazione che ci permettono di rispondere a delle domande in tempo logaritmico.

5 Strutture dati

Una struttura dati è un modo per organizzare i dati in modo da poterli manipolare in modo efficiente.

5.1 Stack o Pila

Definition 5.1. Uno stack è una struttura dati che permette di inserire e rimuovere elementi in modo LIFO (Last In First Out).

- `push(x)`: Inserisce l'elemento x nello stack. (Produce un oggetto che corrisponde allo stack originale a cui è stato aggiunto l'elemento).
- `pop(S)`: Rimuove l'elemento in cima allo stack. (Produce un oggetto che corrisponde all'elemento originale a cui è stato rimosso l'elemento).
- `top(S)`: Restituisce l'elemento in cima allo stack.
- `new()`: Crea uno stack vuoto.
- `isEmpty(S)`: Restituisce `true` se lo stack è vuoto, `false` altrimenti.

Da queste operazioni si possono definire certe proprietà come:

- top(push(S,x)) = x
- pop(push(S,x)) = S

Quindi un modo per definire uno stack tramite le operazioni che andiamo a fare sull'oggetto stesso:

$$\texttt{Push(Push(Push(Empty(),}\ x_1\texttt{)}\ x_2\texttt{)}\ x_3\texttt{)}\ \ldots$$

5.2 Queue o Coda

Definition 5.2. Una coda è una struttura dati che permette di inserire e rimuovere elementi in modo FIFO (First In First Out).

- `enqueue(x)`: Inserisce l'elemento x in coda alla coda. $\longrightarrow$ O(1) se ci tiene salvata il puntatore alla fine dell'array
- `dequeue(Q)`: Rimuove l'elemento in testa alla coda. $\longrightarrow$ O(1)
- `head(Q)`: Restituisce l'elemento in testa alla coda.
- `new()`: Crea una coda vuota.
- `isEmpty(Q)`: Restituisce `true` se la coda è vuota, `false` altrimenti.

5.3 Binary Tree o Albero binario

Definition 5.3. Un albero binario è una struttura dati che permette di organizzare i dati in modo gerarchico. Ogni nodo ha al massimo 2 figli.

- `new()`: Crea un albero vuoto.
- `root(T)`: Restituisce la radice dell'albero.
- `left(T)`: Restituisce il sottoalbero sinistro.
- `right(T)`: Restituisce il sottoalbero destro.
- `key(T)`: Restituisce la chiave del nodo.
- `isEmpty(T)`: Restituisce `true` se l'albero è vuoto, `false` altrimenti.

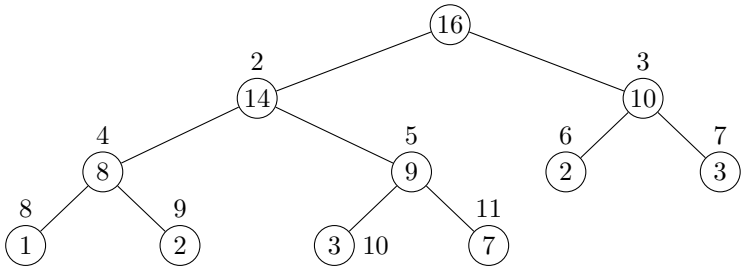


Figure 4: Esempio di albero binario

La profondità di un albero binario è il logaritmo in base 2 del numero di nodi.

$$P(n) = 1 + P\left(\left\lceil \frac{n-1}{2} \right\rceil\right) = \Theta(\log_2 n)$$

5.4 Binary Search Tree o Albero binario di ricerca

Definition 5.4. Un albero binario di ricerca è un albero binario in cui per ogni nodo x valgono le seguenti proprietà:

- Tutti i nodi nel sottoalbero sinistro di x hanno chiavi minori di x.
- Tutti i nodi nel sottoalbero destro di x hanno chiavi maggiori di x.

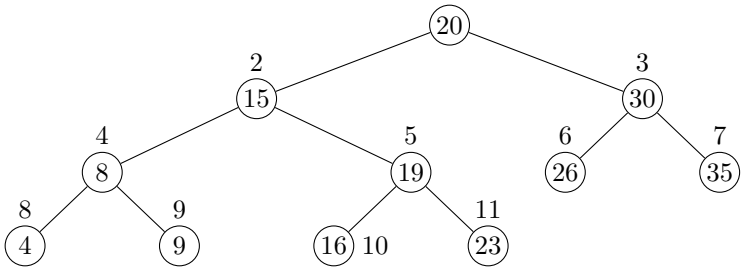


Figure 5: Esempio di albero binario di ricerca

Per cercare un elemento all'interno dell'albero binario di ricerca si può fare in *tempo logaritmico*. Infatti la sua complessità è $\Theta(\log n)$. Tuttavia nel caso peggiore devo scorrere tutto l'albero e quindi l'algoritmo finisce ad aveere $O(n)$. Le funzioni che potremmo implementare per questa struttura potrebbe essere:

- `search(T, k)`: Cerca la chiave k nell'albero T.
- `insert(T, k)`: Inserisce la chiave k nell'albero T. $\implies \Theta(\log n)$
- `extract(T, k)`: Estrae la chiave k dall'albero T. $\implies \Theta(\log n)$

Potremmo essere capaci di inserire un elemento nella giusta posizione in tempo logaritmico se l'albero è sbilanciato? No, infatti se l'albero è sbilanciato la complessità diventa lineare. Tuttavia ci sono dei workaround ma serviranno degli alberi particolari.

Una volta che un nodo viene rimosso o aggiunto non puoi essere certo che l'albero sia ancora ribilanciato. Quindi quello che bisogna fare è ribilanciare l'albero ogni volta che si rimuove o si inserisce un nodo.

5.5 Liste doppiamente concatenate

Come faccio ad eliminare un nodo all'interno della lista evitando di scrivere una lunga sintassi per riassegnare i puntatori dell'elemento prima e dell'elemento dopo? Utilizzo le sentinelle.

5.5.1 Sentinelle

Inserisco due nodi speciali all'inizio e alla fine della lista. Questi nodi speciali non contengono dati e sono chiamati sentinelle. Questi nodi speciali mi permettono di eliminare un nodo in modo più semplice. A volte l'efficienza non è la cosa più importante, infatti ci sono alcuni pezzi di codice che vogliamo ottimizzare nel migliore dei modi. Per tutti gli altri siamo disposti a perdere un po' di memoria. Proprio come le sentinelle.

5.6 RB-Tree o Albero rosso-nero

Definition 5.5. Perché si chiamano red black? Questi sono normali alberi di ricerca, devono rispettare alcune proprietà:

- Ogni nodo è rosso o nero.
- Ogni foglia è necessariamente nera.
- Figli di un rosso sono necessariamente neri.
- Ogni cammino radice-fopglia ha lo stesso numero di nodi neri.

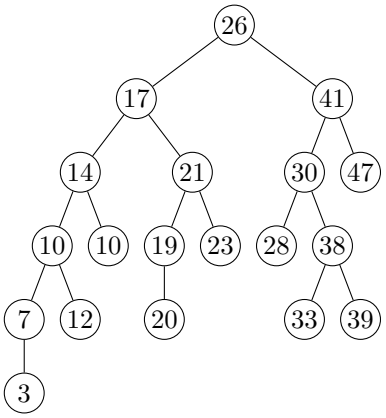


Figure 6: Esempio di albero binario

La proprietà 1 e 3 sono rispettate mentre la proprietà 2. Per sistemare questo problema potremmo bilanciare l'albero aggiungendo alle foglie rossi dei nodi "NIL" che saranno neri. In questo modo la proprietà 2 è rispettata.

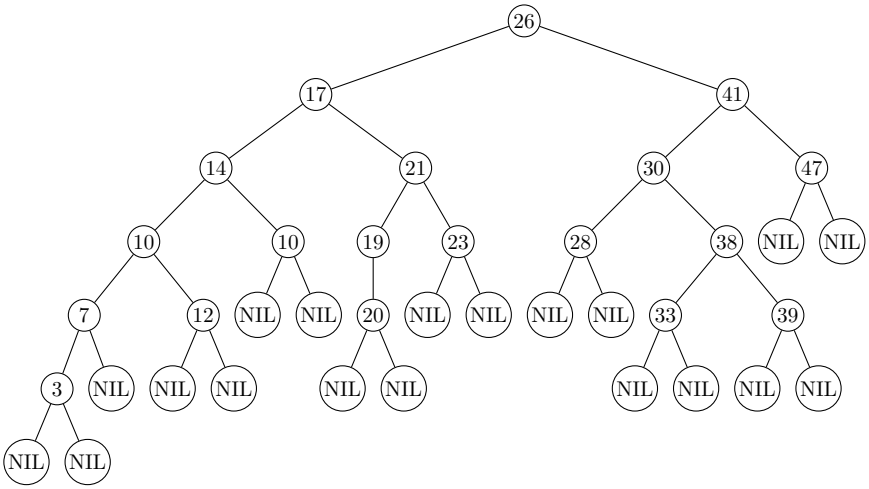


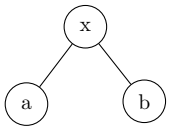
Figure 7: RB Albero bilanciato con i NIL

Il concetto principlae di questo tipi di alberi è quello della **Black Height** (bh(X) dove X è una foglia) cioè il numero di nodi neri che si incontrano lungo un cammino dalla radice ad una foglia.

Lemma 5.6. *Per ogni nodo X il sottoalbero radicato in X ha almeno $2^{bh(X)} - 1$ nodi.*

Proviamo a dimsotrare il lemma:

Proof. Dimostriamo il lemma per induzione. Per $bh(X) = 0$ il lemma è vero. Supponiamo che il lemma sia vero per tutti i nodi. Allora i figli di X hanno $bh(X) = k - 1$. Per ipotesi induttiva:



dove $bh(a) \geq bh(X) - 1$ e $bh(b) \geq bh(x) - 1$

$$\begin{aligned} \#nodi &\geq 2^{bh(a)} - 1 + 2^{bh(b)} - 1 + 1 \\ &\geq 2^{bh(X)-1} - 1 + 2^{bh(X)-1} \cancel{-1+1} \\ &\geq 2 \cdot 2^{bh(X)-1} - 1 \\ &= 2^{bh(X)} - 1 \quad \square \end{aligned}$$

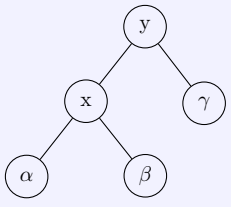
Prendiamo un albero RB di altezza h, possiamo dire che l'altezza di una foglia sia uguale a 0. Quindi l'altezza di un nodo che non sia una foglia sia uguale al numero di tutti i nodi che incontro lungo il cammino meno uno. $bh(X) \geq \frac{h}{2}$ L'altezza nera è almeno la metà dell'altezza dell'albero. Quindi l'altezza dell'albero è al massimo il doppio dell'altezza nera.

$$\#\text{nodi interni} \geq 2^{\frac{h}{2}} - 1$$

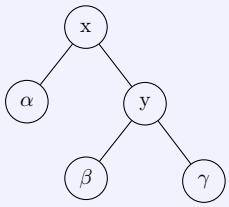
$$\begin{aligned} 2^{\frac{h}{2}} &\leq n + 1 \\ \frac{h}{2} &\leq \log_2(n+1) \\ h &\leq 2\log_2(n+1) \end{aligned}$$

Come faccio a decidere se il nodo inserito deve essere rosso o nero? In questo caso, decidiamo che di base il nodo inserito sarà rosso. Se il padre del nodo inserito è rosso per la regola numero 2, allora quella parte di albero non sarà un RB-Albero creando un anomalia. L'oggetto RB-Albero in questo momento ha un puntatore che punta ad un nodo X che potrebbe essere un nodo rosso figlio di un nodo rosso. Se l'albero presenta un'anomalia, dobbiamo eliminarla. Posso decidere se propagare l'anomalia verso l'alto in modo che risolvo l'anomalia per tutta la profondità dell'albero. In questo caso ci viene in aiuto l'algoritmo di rotazione destra o sinistra che ribalincierà l'albero:

Esempio di rotazione dx e sx



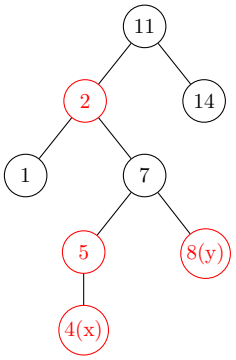
Ruota-dx →:



Pseudocodice dell'algoritmo della rotazione:

```
//Gli passiamo il nodo che ha l'anomalia insieme all'intero albero
// la funziona p[x] prende il parent del nodo x
while != root && color(P[x]) = red
  if p[x] = left[p[p[x]]]
    y <- right[p[p[x]]]
    if color[y] = red
      color(p[p[x]]) <- red
      color[y] <- black
      color[p(x)] <- black
      x <- p[p[x]]
    else
      if x = right[p[x]]
        x <- p[x]
        rotation-sx(x)

      color[p[x]] <- black
      color[p[p[x]]] <- red
      rotation-dx(p[p[x]])
      x <- root
```



L'idea è quella di spostare l'anomalia in alto. Se il padre del nodo inserito è rosso, allora il nonno del nodo inserito sarà nero. Quindi posso cambiare colore al nonno e dare al nonno l'anomalia e così via, fino ad arrivare alla radice.

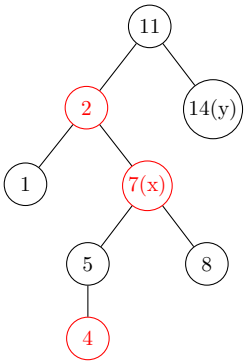


Figure 8: Il padre e lo zio del nodo anomalo diventano nero e il nonno rosso

Ruotiamo a sinistra

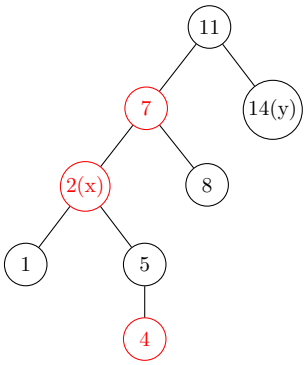


Figure 9: Rotazione a sinistra e l'ordine dei neri non è cambiato

Si cambiano i colori e si effettua una rotazione a destra:

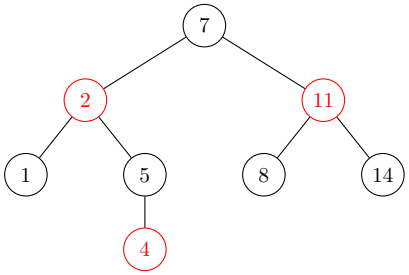
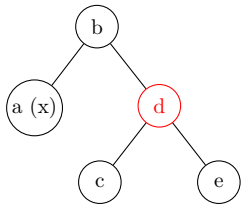


Figure 10: Rotazione a sinistra e l'ordine dei neri non è cambiato

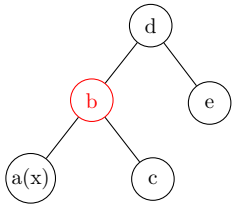
Come faccio *per l'estrazione di un elemento* ora? Se voglio eliminare un nodo rosso, non è un problema perché non c'è nessuna anomalia. Tuttavia se voglio eliminare un nodo nero, allora devo segnarmi l'anomalia perché mi servirebbe un nero in più per fare in modo che ogni cammino radice-foglia abbia lo stesso numero di nodi neri. L'anomalia si crea e un nodo diventa "doppiamente nero" poiché devo contenere per due neri se voglio che rimanga un RB-Albero

L'algoritmo che ribilancia l'albero anch'esso cerca di spostare l'anomalia verso l'alto ed è diviso in diversi casi:

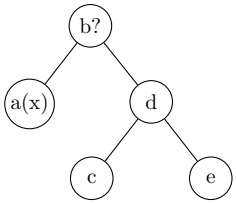
1. Se il fratello del nodo è rosso. Sappiamo che DEVONO esistere il padre, lo zio e i nipoti grazie alle regole dell'RB-Tree.



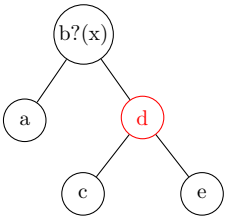
   allora il padre del nodo diventa rosso e il fratello diventa nero. Si effettua una rotazione a sinistra su B



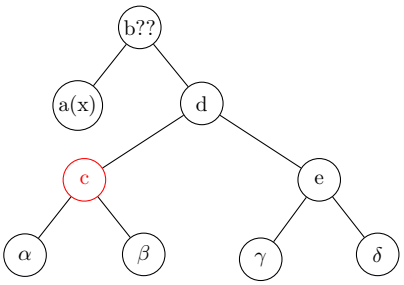
2. Se il fratello del nodo è nero andiamo a controllare i nipoti e sia C che E sono neri. (Il simbolo "?" davanti al nome del nodo vuol dire che può essere di qualsiasi colore)



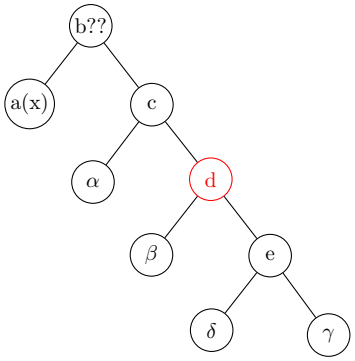
   Togliamo 1 nero ad A e D e aggiungilo a B



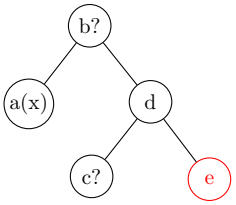
3. Non sappiamo che colore abbia B perché D è nero e assumiamo che uno dei nipoti (C) sia rosso ed E è nero.



Si scambiano colore D e C e fai una rotazione a destra su D



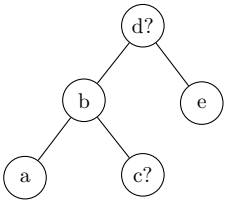
4. Se il nipote destro è rosso e il nipote sinistro è di qualsiasi colore:



Proviamo ad implementare tramite codice e applicarlo sull'albero:

```
color[d] <- color[b]
color[b] <- black
color[e] <- black
rotation-sx(b)

x <- root
```



Abbiamo presentato quattro scenari diversi e per ciascuno di questi scenari abbiamo fatto una trasformazione e ci siamo assicurati ceh quest'ultima mantenga le proprietà dell'RB-Tree. Dobbiamo anche assicurarci che non ci siano nodi rossi figli di rossi.
Riassiumiamo in breve questi casi prendendo come riferimento i nipoti del nodo anomalo:

- C nero, E nero (caso 2)
- C rosso, E nero (caso 3)
- C nero, E rosso (caso 4)
- C rosso, E rosso (caso 4)

Tutti i casi sono esaustivi poiché abbiamo coperto ogni caso. Il caso 1 è il caso in cui D è rosso.

- Il caso 3 e il caso 4 sono casi terminali perché portano al termine dell'algoritmo.
- Il caso 2 non termina ma porta l'anomalia in alto.
- Il caso 1 porta l'anomalia in **basso** e questo è un gran problema. Dopo il caso 1 non può verificarsi nuovamente il caso 1. Se è seguito da un caso 3 e 4 allora l'algoritmo termina. Se invece è seguito dal caso 2 nonostante il caso 2 non sia un algoritmo che termina, **l'algoritmo termina** perché l'anomalia è stata spostata in alto e B era rosso. Quindi anche questo algoritmo termina in tempo $O(\log n)$.

5.6.1 Estrazione in un RB-Albero

Se nell'albero binario posso andare a sinistra o a destra per ogni passaggio, ora sicuramente so che la complessità del mio algoritmo in un RB-Albero è $O(\log n)$.

Per sapere la posizione effettiva della mia radice, posso aggiungere alla mia struttura del nodo una proprietà *size* che mi permette di sapere quanti nodi ci sono sotto al nodo x. Per sapere la posizione effettiva di ogni nodo, posso utilizzare la seguente formula:

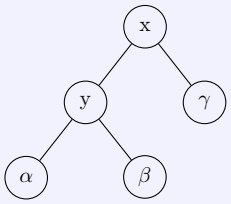
$$size(left(x)) + 1$$

Lo pseudocode per l'estrazione di un nodo in un RB-Albero è il seguente:

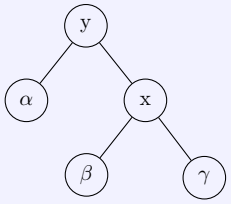
```
// X e' la lista
// i e' la posizione del nodo da estrarre
Select(x, i)
  r <- size(leftx) + 1
  if r = i
    return x
  else if(r > i)
    ret select[left[x], i]
  else
    ret select[right[x], i - r]
```

Tuttavia una volta aver estratto il valore, devo aggiornare la proprietà *size* di tutti i nodi che sono stati toccati.

Esempio



Proviamo a ruotare e poi cambiare la proprietà *size* in questa maniera:



$$size(x) \leftarrow size(left[x]) + size(right[x]) + 1$$

La size di y dopo la rotazione è esattamente la size di x prima della rotazione. Gli unici che cambiano la size quindi, sono x e y e la complessità di questo algoritmo è O(1) e quindi è costante.

5.6.2 Campi aggiuntivi

Si possono aggiornare dei campi aggiuntivi all'interno della struttura dell'RB-Albero per velocizzare alcuni algoritmi (come abbiamo visto per size) ma dobbiamo prestare attenzione a mantenere la complessità delle operazioni inalterate.

Theorem 5.7. *Sia F un campo aggiuntivo, se*

$$\exists f.\ \forall x F[x] = f(key[x], f[left[x]], f[right[x]], key[left[x]], key[right[x]])$$

allora F mantenibile in tempo logaritmico.
Cioè se il campio aggiuntivo è calcolabile utilizzando le operazioni definite in f allora la complessità di F rimane inalterata.

Osservazione. Se siamo in un Albero AVL possiamo aggiungere un campo aggiuntivo tuttavia il teorema appena enunciato non varrà poiché l'inserimento e la rimozione impiega tempo $O((\log n)^2)$. Perché deve svolgere le rotazioni (che impiegano tempo logaritmico) per ogni nodo all'interno del cammino radice-foglia.

Esempio 1

Se volessi aggiungere il campo "Rank(x)" che ritorna la posizione che occupa quel nodo in un ordinamento crescente all'interno dell'albero.

```
// x: Nodo da cui si parte per la selezione
// y: Nodo dove bisogna trovare la
Rank(x, y)
  count = size[left[x]] + 1
  // finche' x non e' la radice
  while(x != root)
    // Se sei figlio destro allora aggiungo la size del
    sottoalbero sinistro e il padre
    if (x == right[parent[x]])
      return count + size[left[parent[x]]] + 1
    // altrimenti x diventa il suo parent e vai sopra
    x = parent[x]

  return count
```

Esempio 2

Se si volessero inserire gli elementi di un RB-Albero in un array ordinato, si potrebbe usare un algoritmo di visita in-order dell'albero. Esistono diversi modi per visitare un albero: Lo pseudocodice è il seguente:

```
// x: Nodo da cui si parte la visita
// i: indice dell'array in cui inserire x
// prima si scrive la radice, poi il sottoalbero sinistro e
    poi il sottoalbero destro
pre_visit(x, i)
  if x != null
    visit(x)
    pre_visit(left[x], i)
    pre_visit(right[x], i)

// la radice si trova al centro e a sinistra si trovo il
    sottoalbero sinistro e a destra il sottoalbero destro
in_visit(x, i)
  if x != null
    invisit(left[x], i)
    visit(x)
    invisit(right[x], i)

post_visit(x, i)
  if x != null
    post_visit(left[x], i)
    post_visit(right[x], i)
    visit(x)

```

Possiamo dimostrare che la visita ha complessità $O(n)$. Per dimostrarlo possiamo usare l'equazione di ricorrenza:

$$\begin{aligned} T(n) &= T(n_{left}) + T(n_{right}) + 1 \\ &= n_{left} + n_{right} + 1 = n \end{aligned}$$

Supponiamo per induzione che $T(n_{left})$ e $T(n_{right})$ sono rispettivamente n_{left} e n_{right}. Quindi possiamo visitare i nodi degli RB-Alberi in **tempo**

lineare.

Se volessi inserire gli elementi di un RB-Albero in un array ordinato, posso usare la visita in-order dell'albero. Tuttavia la sua complessità è $O(n \log n)$ poiché facendo una sequenza di inserimento:

$$1 + \log 1 + \log 2 + \cdots + \log n = \log n! = \Theta(n \log n)$$

Ci chiediamo se possiamo costruire un RB-Albero partendo da un array in un tempo più basso di $O(n \log n)$. Se è così, allora otteniamo un array ordinato e questo vuol dire che riesco a scrivere un algoritmo di ordinamento con complessità inferiore di $O(n \log n)$.

```
sort(A)
  T <- build_tree(A)
  B <- in_visit(T)
```

Tuttavia avevamo già dimostrato che non è possibile fare un ordinamento in tempo inferiore a $O(n \log n)$. Riusciamo a scrivere una funzione "buildtree" che dato un array ordinato mi costruisce un RB-Albero

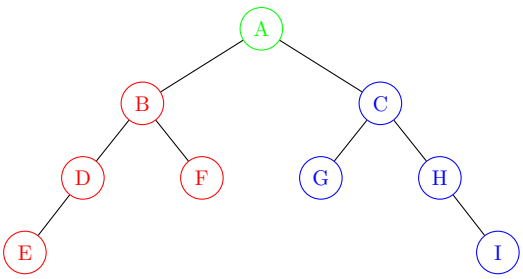
```
build_tree(A)
  if A = []
    return null
  else
    m <- A.length / 2
    x <- new Node(A[m])
    left[x] <- build_tree(A[0, m-1])
    right[x] <- build_tree(A[m+1, A.length])
    return x
```

Proviamo ora a scrivere un algoritmo che costruisce un nodo e prende in input due stringhe prese dalla previsita e dalla invisita.

```
buildtree(p, i)
  if(length(i) == 0) {
    return null
  } else {
    x <- add[p[0]]
    newP <- i.indexOf(p[0])
    left(x) <- buildtree(P[1 .. newP], I[0 .. newP-1])
    right(x) <- buildtree(P[newP+1 .. p.length], I[newP+1 .. I.
    length])
  }
```

Prendiamo come esempio le due stringhe:

$$PREvisit = [\text{A, B, D, E, C, C, G, H, I}]$$
$$INvisit = [\text{E, D, B, F, A, G, C, H, I}]$$



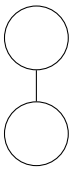
Non è possibile unire due RB-Alberi in tempo logaritmico in modo deterministico perché:

- L'operazione richiede spesso una manipolazione globale della struttura.
- Il bilanciamento degli RB-Alberi impone vincoli che possono richiedere tempo lineare rispetto alla dimensione degli alberi coinvolti.

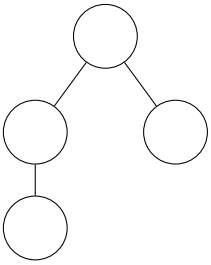
5.7 Heap Binomiale

Un heap binomiale è una collezione di alberi binomiali. Un albero binomiale di dimensione 0 è un singolo nodo.

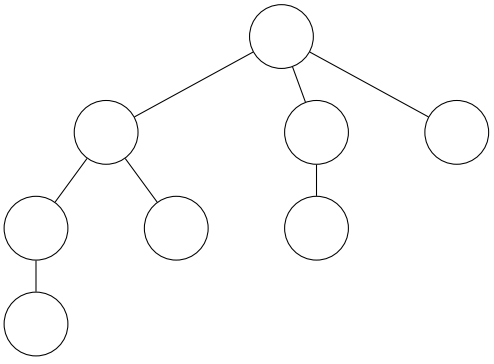
Un albero di dimensione 1 è un albero binario con un solo figlio.



Un albero di dimensione 2 diventa:



Un albero di dimensione 3 diventa:



Dato un albero di dimensione i posso costruire un albero di dimensione $i+1$ in tempo costante. Questi alberi hanno una serie di proprietà:

Lemma 5.8. *In un albero binomiale di dimensione k:*

- *ci sono 2^k nodi.*
- *L'altezza è k.*
- *A profondità i ci sono $\binom{k}{i}$ nodi.*
- *I figli della radice da sx a dx sono radici di alberi binomiali di dimensione $k-1, k-2, \ldots, 0$.*

Cerchiamo di dimostrare le proprietà che abbiamo appena elencato:

Proof. Dimostrazione per induzione: Ci sono 2^k nodi: La proprietà è valida per gli alberi di dimensione 0 (B_0) perché ha $2^0 = 1$ nodo. supponiamo per induzione che in un albero di dimensione k (B_k) ci siano 2^k nodi. Dimostriamo che nell'albero di dimensione $k+1$ (B_{k+1}) formato da due alberi B_k sono dunque presenti:

$$2^k + 2^k = 2 \cdot 2^k = 2^{k+1}$$

Proof. L'altezza è $k+1$: Supponiamo per induzione che in albero di dimensione k ci siano 2^k nodi. Come è fatto un albero di dimensione $k+1$? L'albero di dimensione $k+1$ è formato da un albero di dimensione k e un albero di dimensione k. Quindi ci sono $2 \cdot 2^k = 2^{k+1}$ nodi. L'altezza è $k+1$. A profondità i ci sono $\binom{k+1}{i}$ nodi. I figli della radice da sx a dx sono radici di alberi binomiali di dimensione $k, k-1, \ldots, 0$.

Proof. I figli dellla radice da sx a dx sono radici di alberi binomiali di dimensione $k-1, k-2, \ldots, 0$: Per $k = 0$ è vero. Dato che l'altezza è $k+1$, il figlio sinistro della radice è un albero binomiale di dimensione k e il figlio destro è un albero binomiale di dimensione $k-1$.

Proof. A profondità i ci sono $\binom{k}{i}$ nodi: Supponiamo per induzione che in albero di dimensione k ci siano $\binom{k}{i}$ nodi. Come è fatto un albero di dimensione $k+1$? L'albero di dimensione $k+1$ è formato da un albero di dimensione k e un albero di dimensione k. Quindi ci sono $\binom{k}{i} + \binom{k}{i-1}$ nodi. Quindi, tramite l'identità di Pascal, otteniamo $\binom{k+1}{i}$ nodi. L'identità di Pascal dice che

$$\binom{k}{i} + \binom{k}{i-1} = \binom{k+1}{i}$$

A profondità i ci sono $\binom{k+1}{i}$ nodi. I figli della radice da sx a dx sono radici di alberi binomiali di dimensione $k, k-1, \ldots, 0$.

Quindi riassumendo: *La Heap Binomiale è una lista di alberi binomiali* dove

- I contenuti dei nodi sono oggetti su cui è definita una relazione di ordinamento
- Per ogni di dimensione c'è al più un albero binomiale
- I vari nodi soddisfano la proprietà di heap (chiave[x] $\leq$ chiave[children[x]])

Esempio 1

La complessità di trovare il minimo all'interno di questo Heap Binomiale si basa sul numero di radici. Dove n è il numero di nodi:

$$k > \log_2 n$$

$$2^k > 2^{\log_2 n} = n$$

Quindi la complessità è $O(\log n)$, anzi $\Omega(\log n)$ perché devo scorrere tutti gli alberi.

Esempio 2

Se volessi costruire uno Heap Binomiale con n nodi fissato, è possibile perché essendo ogni albero rappresentabile come 2^k posso scomporre il valore n come somma di potenze di due.

$$n = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_m}$$

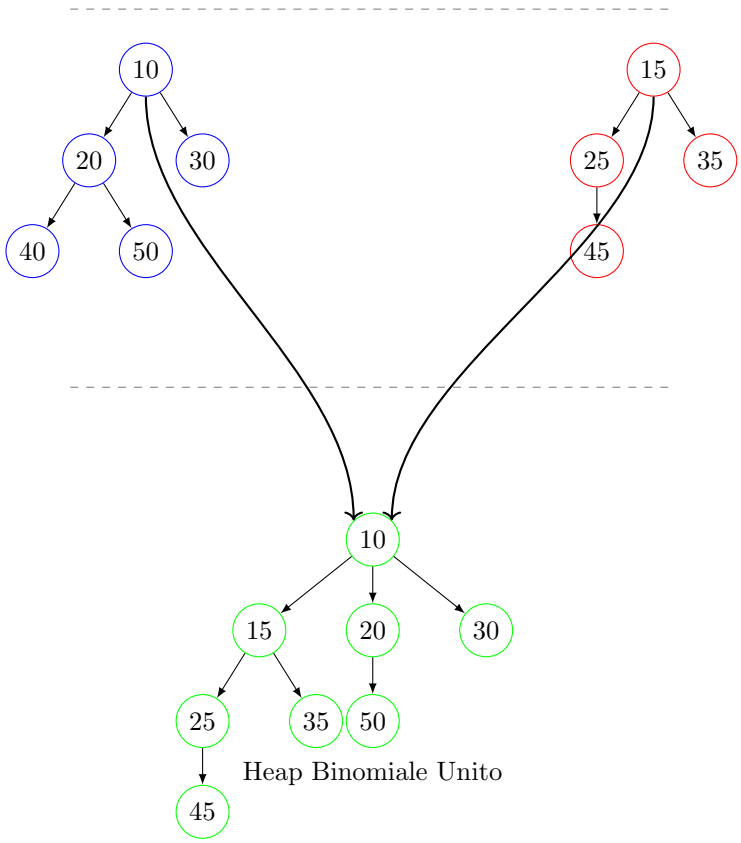
Per esempio:

$$37 = 2^5 + 2^2 + 2^0$$

Quindi posso costruire uno Heap Binomiale con qualsiasi n nodi.

5.7.1 Unione di Heap Binomiali

Supponiamo di voler fare l'unione di due heap binomiali H_1 e H_2.



Massimo di iterazione massimo $O(\log n)$ e quindi la complessità è $O(\log n)$. L'inserimento di un nodo non è altro che l'unione di un nodo di dimensione 0 con l'heap binomiale a cui vogliamo aggiungere il nodo.

5.7.2 Rimozione di un nodo

La rimozione di un nodo in uno heap binomiale è un'operazione che può essere eseguita in tempo $O(logn)$. Supponendo di voler rimuovere il nodo con la chiave più piccola (che corrisponde sempre alla radice dell'heap), il processo si articola nei seguenti passi:

- Identificazione della radice con chiave minima: La radice con chiave minima può essere trovata scorrendo l'elenco delle radici degli alberi binomiali. Questo richiede tempo

  $O(logn)$, dato che il numero massimo di alberi binomiali in uno heap è proporzionale a logn.

- Rimozione della radice: Una volta identificata la radice minima, essa viene rimossa. A seguito della rimozione, i sottoalberi figli della radice vengono "scompattati". Ciascuno di questi sottoalberi è un albero binomiale separato, che può essere gestito individualmente.

- Ricostruzione dello heap: Per ripristinare la struttura dello heap binomiale, i sottoalberi risultanti vengono uniti tra loro e con il resto dello heap. L'operazione di unione avviene combinando alberi di gradi uguali, procedendo come nell'algoritmo di fusione. Anche questa operazione richiede $O(logn)$.

In sintesi, l'operazione di rimozione implica un'iterazione sugli alberi binomiali (per identificare la radice minima), seguita dalla fusione dei sottoalberi generati. Entrambe le fasi hanno complessità $O(logn)$, rendendo l'intero processo efficiente.

5.7.3 Diminuzione di una chiave e rimozione di un elemento arbitrario

Per diminuire la chiave di un nodo in uno heap binomiale, è necessario prestare attenzione a non violare la proprietà di heap. Se la chiave diminuita è minore della chiave del padre, occorre scambiare il nodo con il padre e ripetere questa operazione (nota come *risalita*) fino a quando la proprietà di heap non è più violata. La complessità di questa operazione è $O(\log n)$, poiché la profondità massima di un albero binomiale è $O(\log n)$.
La diminuzione della chiave è utile in molte applicazioni, come negli algoritmi di ottimizzazione. Tuttavia, ci sono operazioni più potenti che potrebbero comportare una complessità maggiore.
Per **rimuovere un nodo arbitrario**, possiamo sfruttare l'operazione di diminuzione della chiave. Riducendo la chiave del nodo a $-\infty$, il nodo verrà spinto alla radice, rendendo la sua rimozione equivalente a quella della radice, che richiede $O(\log n)$ tempo.
Quindi, con uno heap binomiale, è possibile eseguire le seguenti operazioni in tempo $O(\log n)$:

- Inserire un nodo
- Trovare il minimo
- Unire due heap binomiali
- Rimuovere la radice
- Diminuire una chiave
- Rimuovere un nodo arbitrario

Tuttavia, alcune operazioni non possono essere eseguite in tempo $O(\log n)$:

- Trovare il massimo
- Cercare un elemento arbitrario

5.7.4 Trovare il mediano

Poiché l'array di partenza non è ordinato, trovare il mediano richiede generalmente un'operazione di tempo lineare, ovvero $O(n)$. Tuttavia, è possibile ottimizzare questo processo trasformando l'array in un heap binomiale. L'heap

binomiale, sebbene non permetta l'accesso diretto al mediano, offre una struttura che può facilitare il processo.
Per trovare il mediano utilizzando un heap binomiale, è necessario seguire alcuni passaggi aggiuntivi. In particolare, dopo aver costruito l'heap binomiale, si può procedere ad estrarre gli elementi in ordine crescente (o decrescente), fino a raggiungere l'elemento centrale. Sebbene l'heap binomiale non fornisca un accesso diretto alla posizione del mediano, l'ordinamento parziale che essa garantisce può ridurre il numero di operazioni necessarie rispetto a un ordinamento completo dell'array.
In alternativa, un approccio come l'algoritmo di selezione del k-esimo elemento potrebbe essere utilizzato in combinazione con l'heap binomiale per trovare il mediano in modo più efficiente.

5.7.5 RB-Alberi e Search Closest

Se in un RB-Albero non dobbiamo cercare la chiave precisa ma la chiave più vicina dobbiamo cercare di utilizzare l'RB Albero in modo tale da poter usare la potenza del suo ordinamento. Come chiave di ordinamento uso la chiave x-k ovvera la differenza tra le due misure.

```
// Primo candidato
// k valore a cui voglio avvicinarmi
// BestCandidate e' un ulteriore candidato
searchClosest(x,k)
  if x == null
      return null
  else if(key[x] > k)
      bestCandidate = searchClosest(right[x], k)
  else if (key[x] < k)
      bestCandidate = searchClosest(left[x], k)

  if bestCandidate == null ||  (key[x] - k) < (bestCandidate - k)
    return x
  else
    return bestCandidate
```

Questo algoritmo ha complessità $O(\log n)$ poiché dobbiamo comunque passare per il cammino radice foglia.

Esempio

Ora immaginiamo questo problema: Siamo dei venditori di macchine e conteniamo i dati delle macchine all'interno di un record dove sono contenuti informazioni come: prezzo di vendita, data di vendita, ecc. Vogliamo trovare la somma dei prezzi delle macchine che abbiamo venduti fino ad una certa data. Per costruire questo tipo di albero possiamo servirci della possibilità di aggiungere campi aggiuntivi mantenibili in tempo logaritmico.

```
SearchDate(x, k)
  if(x == null)
    return null
  else if (date[x] <= k)
    bestDate = SearchDate(right[x], k)
    return best ?? x //equivalente a scrivere -> best != null
    ? best : x
```

```
7   else if (date[x] > k)
8     bestDate = SearchDate(left[x], k)
9     return best ?? x
```

Quindi ora abbiamo trovato il candidato migliore della data a cui possiamo applicare il rango per trovare la somma dei prezzi delle macchine vendute fino a quella data.

5.8 Strutture dati facilmente partizionabili

Vogliamo una struttura dati che sia formata da insiemi disgiunti di oggetti e possega le seguenti operazioni:

- `MakeSet(x)`: Crea un insieme contenente l'oggetto x.
- `Union(x, y)`: Unisce gli insiemi contenenti x e y.
- `FindSet(x)`: Restituisce l'insieme contenente x. Cerca quindi il **rappresentante** dell'insieme a cui appartiene x. Quindi quando voglio capire se due oggetti appartengono allo stesso insieme posso controllare se il risultato di `FindSet(x)` e `FindSet(y)` sono uguali.

posso organizzare gli oggetti nelle liste concatenate in cui il rappresentante è il primo oggetto della lista.

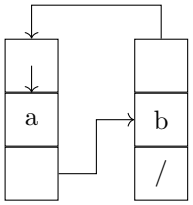


Figure 11: Esempio di una lista concatenata

in questo modo `MakeSet(x)` e `FindSet(x)` hanno complessità costante, ma la `Union(x,y)` avrebbe complessità lineare perché devo capire se i due oggetti hanno lo stesso rappresentante, poi scorrere uno dei due insiemi fino alla fine e poi cambiare il puntatore dell'insieme aggiunto al nuovo rappresentante per risolvere il primo problema che causa linearità potrei aggiungere al rappresentante il puntatore all'ultimo elemento.

5.8.1 Ottimizzazione della complessità con alberi e unione per rango

Se vengono effettuate m operazioni di cui n sono `make_set` (cioè il numero di oggetti, di conseguenza il numero di union possibili), la complessità nel caso pessimo è $O(m * n)$. Di tutte queste operazioni, le `union` saranno al massimo $n - 1$ perchè non si possono unire più di n insiemi. Tutte le rimanenti saranno operazioni di tempo costante, di conseguenza un altro limite superiore corretto sarebbe $O(m + n^2)$. Un limite superiore migliore si può ottenere calcolando la complessità media di tutte le operazioni e quindi la complessità finale diventa: $O(\frac{m+n^2}{m}) = O(1 + \frac{n^2}{m})$.

Per capire se questa complessità è accurata bisogna trovare un insieme di operazioni da fare che portino ad avere quella complessità.

1. Si fanno n `make_set` creando n insiemi
2. Si uniscono le coppie di insiemi. Per la prima coppia il costo è 1, per la seconda 2 e così via. Il costo totale è:

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

Per fare meglio di $\Theta(n^2)$ bisogna ottimizzare la union, cioè l'aggiornamento di tutti i puntatori. Si potrebbe unire l'insieme più piccolo a quello più grande, al posto di unire quello più grande a quello più piccolo e a questo punto bisogna cambiare soltanto un puntatore, nel caso in cui l'elemento più piccolo abbia un elemento. Questo metodo è chiamato **unione per rango**. Il numero massimo di volte che si può modificare il puntatore al rappresentante dopo questa tecnica diventa $\log_2 n$ perchè:

1. Se ad un insieme di un elemento viene cambiato il puntatore al rappresentante allora sappiamo che l'insieme risultante avrà minimo 2 elementi:

$$\geq 2$$

2. Se ad un insieme di due elementi viene cambiato il puntatore al rappresentante allora sappiamo che l'insieme risultante avrà al minimo 4 elementi (perchè sappiamo che bisogna unire l'insieme più piccolo ad un altro insieme, di conseguenza se sappiamo che l'insieme di due elementi è il più piccolo, l'altro insieme avrà al massimo 2 elementi):

$$\geq 4 = 2^2$$

3. ...
4. Se ad un insieme di i elementi viene cambiato il puntatore al rappresentante allora sappiamo che l'insieme risultante avrà al minimo 2^i elementi:

$$\geq 2^i$$

Questa costruzione si chiama **iterative squaring**. La complessità con l'unione per rango diventa:

$$\frac{m + n \log n}{m} = 1 + \frac{n \log n}{m} \leq 1 + \log n$$

Per fare meglio si può rappresentare in un modo alternativo gli insiemi, quindi con degli alberi.

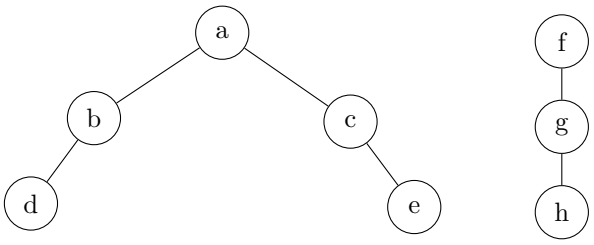


Figure 12: Esempio di alberi come insiemi

- `make_set(x)`: si crea un albero con un solo nodo

$$O(1)$$

- `find_set(x)`: si risale l'albero fino alla radice

$$O(n)$$

- `union(x,y)`: si fa diventare la radice di un albero il figlio dell'altra

$$O(n)$$

Come prima si può usare l'unione per rango per migliorare la complessità. Ogni volta che un nodo aumenta di profondità (grazie all'unione ad un altro albero), il numero di nodi raddoppia. Facendo l'unione per rango sappiamo che la complessità della ricerca del rappresentante è $O(\log n)$ e la complessità dell'unione è $O(\log n)$.

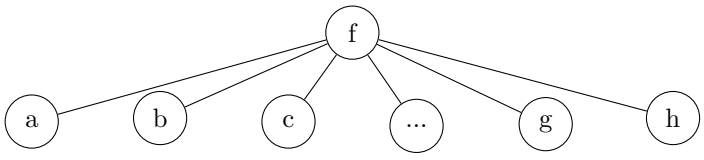
Siccome la `find_set` è $O(\log n)$ siamo comunque peggio di prima, quindi si può far puntare ogni nodo direttamente al rappresentante, ma questo viene fatto soltanto quando viene richiamata la `find_set`, im modo da non farlo più alle chiamate successive. L'algoritmo è il seguente:

```
find_set(x):
  if parent(x) == x
    return x
  else
    parent(x) = find_set(parent(x))
    return parent(x)
```

oppure più compatto:

```
find_set(x):
  if parent(x) != x
    x <- find_set(parent(x))
  return parent(x)
```

Man mano che vengono eseguite le `find_set` la struttura si comprime (**tecnica di compressione dei cammini**):



Di conseguenza la complessità della `find_set` diventa costante.

La nuova complessità sapendo di avere m operazioni di cui n `make_set` diventa:

$$O(m\alpha(m,n))$$

dove α è l'inversa della funzione di Ackermann:

$$\alpha(m,n) = min\left\{i \geq 1, \left|A\left(i, \left\lfloor \frac{m}{n} \right\rfloor > \log n\right)\right.\right\}$$

$$A(i,1) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad \text{(i volte)}$$

Siccome la funzione di Ackermann cresce molto velocemente, la sua inversa cresce molto lentamente. Di conseguenza non si riuscirà mai a superare una certa costante, quindi la complessità diventa costante.

5.9 Il concetto di funzione calcolabile

Le funzioni descritte in maniera **costruttiva** non sono la totalità delle funzioni esistenti perché esse sono maggiori della cardinalità stessa dei naturali (Infinito non numerabile). I fratelli maggiori Robison partendo da assiomi di Peano:

- 0 è un numero naturale
- $\forall x \in \mathbb{N},\ x = x$
- $x + 0 = x$
- $x + (y + 1) = (x + y) + 1$

hanno dimostrato che l'insieme delle funzioni costruttive sono ricorsive primitive. Ackermann aveva capito che la seguente struttura confutava ciò che i Robinson avevano dimostrato:

$$x * 1 = x$$
$$x * (y + 1) = x + x * y$$

oppure

$$x^1 = x$$
$$x^{y+1} = x \cdot x^y$$

Questo dimostrava come ci fossero funzioni che non erano ricorsive primitive.

```
// Funzione di Ackermann in Pseudocodice
A(x, y)
  if x == 0
    return y + 1
  else if y == 0
    return A(x-1, 1)
  else
    return A(x-1, A(x, y-1))
```

5.9.1 Esempio di mantenimento di un campo in un RB-Albero

Esempio

Immaginiamo di gestire un agenda in cui poter inserire e togliere appuntamenti con inizio e fine, poi anche una funzione che preso un intervallo di tempo mi restituisce un appuntamento in cui andrebbe in collisione. Usiamo gli RB-Alberi in cui i nodi rappresentano gli appuntamenti. In ogni nodo x teniamo il campo $max[x]$ che mantiene il massimo del tempo finale tra tutti i nodi radicati in x. Questo campo è mantenibile in tempo logaritmico.

```
search(x,i)
  if x == nil || i in x
    return x
  else if left[x] != nil && max[left[x]] > min[i]
    return search(left[x], i)
  else
    return search(right[x], i)

```

6 Tecniche di programmazione

Fin'ora abbiamo utilizzato la tecnica del **divide et impera**, cioè dividere il problema in parti più piccole e risolverle con lo stesso algoritmo per poi unirle per ottenere il risultato. Questa non è l'unica tecnica di programmazione esistente, ce ne sono molte altre, ad esempio la **programmazione greedy**, cioè prendere le decisioni il prima possibile e sperare che siano le migliori, oppure la **programmazione dinamica**, cioè creare un'infrastruttura prima di poter prendere una decisione.

6.1 Programmazione dinamica

Prendiamo ad esempio il problema della **moltiplicazione di matrici**. Sappiamo che se abbiamo due matrici A di dimensione $n \times m$ e B di dimensione $m \times l$, il prodotto tra le due matrici avrà complessità $\Theta(nml)$.

Se volessimo moltiplicare 3 matrici: $A_1 \cdot A_2 \cdot A_3$ di dimensione:

$$A_1 : 10 \times 100$$
$$A_2 : 100 \times 5$$
$$A_3 : 5 \times 50$$

possiamo sfruttare la proprietà associativa:

$$(A_1 \cdot A_2) \cdot A_3 = A_1 \cdot (A_2 \cdot A_3)$$

Solo che il numero di operazioni nei due casi è diverso, quindi per rendere minimo il numero di operazioni è più conveniente fare: $(A_1 \cdot A_2) \cdot A_3$ perchè:

$$(A_1 \cdot A_2) \cdot A_3 \quad 10 \times 100 \times 5 + 10 \times 5 \times 50 = 5000 + 2500 = 7500$$
$$A_1 \cdot (A_2 \cdot A_3) \quad 100 \times 5 \times 50 + 10 \times 100 \times 50 = 25000 + 50000 = 75000$$

Supponiamo di avere un insieme di matrici $A_1, A_2, \ldots, A_n$, vogliamo trovare la **parentesizzazione** ottimale per minimizzare il numero di operazioni.

Un modo per affrontare il problema è quello di provare tutte le combinazioni di parentesi e vedere quale è la migliore.

C'è per forza una moltiplicazione che dovrà essere eseguita per ultima, quindi distinguiamo k come il punto in cui verrà fatta l'ultima moltiplicazione.

$$(A_1 \cdots A_k) \cdot (A_{k+1} \cdots A_n)$$

Il numero modi per moltiplicare le matrici è dato dal numero di modi per moltiplicare le matrici $A_1 \dots A_k$ e le matrici $A_{k+1} \dots A_n$.

$$P(n) = \sum_{k=1}^{n-1} P(k) + P(n-k) \quad \in \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

La complessità è troppo alta, quindi provare tutte le combinazioni possibili non è efficace.

Supponiamo che qualcuno ci dica il valore ottimo di k, allora è sicuro che il modo in cui vengono moltiplicate le matrici $(A_1 \cdots A_k)$ e $(A_{k+1} \cdots A_n)$ è ottimale:

$$\underbrace{(A_1 \cdots A_k)}_{\text{Ottimo}} \cdot \underbrace{(A_{k+1} \cdots A_n)}_{\text{Ottimo}}$$

Questa tecnica si chiama **Sottostruttura ottimale**, un problema si può dividere in problemi della stessa natura, ma più piccoli.

Il numero di moltiplicazioni effettuate è dato da:

$$N\left(A_1 \cdots A_n\right) =$$

$$\begin{aligned} &= N\left(A_1 \cdots A_k\right) + N\left(A_{k+1} \cdots A_n\right) + \text{Costo ultima moltiplcazione} \\ &= N\left(A_1 \cdots A_k\right) + N\left(A_{k+1} \cdots A_n\right) + rows(A_1) \cdot cols(A_k) \cdot cols(A_n) \end{aligned}$$

Se k è ottimo, allora $N\left(A_1 \cdots A_k\right)$ e $N\left(A_{k+1} \cdots A_n\right)$ sono ottimi perchè il risultato viene sommato e se non fossero ottimi il risultato finale non sarebbe ottimo. Il problema è che non sappiamo quale sia il valore di k ottimo.

Un possibile algoritmo per trovare k è il seguente:

```
// P e' un vettore che descrive le matrici come:
//   - P[0] = rows(A1)
//   - P[i] = cols(Ai)
// Quindi Ai ha dimensione P[i-1] x P[i]
// i e' l'indice di partenza
// j e' l'indice di fine
matrix_chain_order(P,i,j)
  if i == j
    return 0

  m <- +inf
  for k <- i to j-1
    m <- min(m,
      matrix_chain_order(P,i,k) +
      matrix_chain_order(P,k+1,j) +
      P[i-1]*P[k]*P[j])
  return m
```

Questo algoritmo ha una complessità esponenziale.

Cerchiamo di migliorare l'algoritmo, implementando l'**albero di ricorrenza**, cioè un alberi i cui nodi rappresentano le chiamate ricorsive dell'algoritmo, identificate dai parametri i e j.

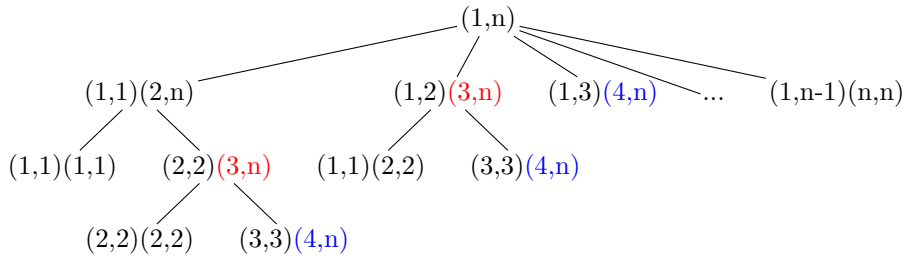


Figure 13: Albero di ricorrenza

Notiamo che ci sono molte chiamate ricorsive ripetute, ad esempio $(4, n)$, e questo aumenta la complessità dell'algoritmo inserendo istanze del problema che abbiamo già affrontato. Al massimo ci saranno n^2 istanze di un problema.

Anzichè risolvere il problema ogni volta che lo incontriamo, possiamo risolvere ogni singolo problema una sola volta e salvare il risultato:

```
matrix_chain_order(P)
  n <- length(p) - 1
  for i <- 1 to n // n
    // La moltiplicazione di una sola matrice costa 0 perche' non c
    'e' nulla da moltiplicare
    M[i,i] <- 0 // Matrice che salva tutte le soluzioni (i,j)

  for l <- 2 to n //           n --* Problemi composti da n matrici
    for i <- 1 to n - l + 1 // n --*
      j <- i + l - 1 //            | n^3
      M[i,j] <- +inf //            |
      for k <- i to j - 1 //   n --*
        M[i,j] <- min(M[i,j],
          M[i,k] + M[k+1,j] + P[i-1]*P[k]*P[j])

  return m[1,n]
```

Questo ci dice quante moltiplicazioni sono necessarie per moltiplicare tutte le matrici, però per sapere la posizione dell'ultima moltiplicazione da fare invece si può riscrivere l'algoritmo precedente come:

```
matrix_chain_order(P)
  ...
      for k <- i to j - 1
        m <- M[i,k] + M[k+1,j] + P[i-1]*P[k]*P[j]
        if m < M[i,j]
          M[i,j] <- m // Numero moltiplicazioni
          S[i,j] <- k // Indice dell'ultima moltiplicazione
  ...
```

La complessità di questo algoritmo è $O(n^3)$.

L'algoritmo precedente scritto in modo iterativo si potrebbe anche scrivere ricorsivamente e l'idea è che prima di calcolare il risultato si controlla se è già stato calcolato e si utilizza quello, altrimenti si calcola e si salva il risultato.

```
matrix_chain_order(P)
  for i <- 1 to n
    for j <- 1 to n
      M[i,j] <- +inf

  matrix_chain_order_aux(P,1,n)


matrix_chain_order(P,i,j)
  if M[i,j] != +inf
    return M[i,j]
  else
    if i == j
      M[i,j] <- 0
    else
      m <- +inf
      for k <- i to j-1
        m <- min(m,
          matrix_chain_order(P,i,k) +
          matrix_chain_order(P,k+1,j) +
          P[i-1]*P[k]*P[j])
      M[i,j] <- m

    return M[i,j]
```

Per ogni istanza del problema si calcola una sola volta il risultato, quindi la complessità è $O(n^3)$.

Definition 6.1. La tecnica di memorizzare i risultati già calcolati si chiama **memoizzazione**.

L'algoritmo che data la matrice S, che contiene gli indici delle parentesizzazioni ottimal e la matrice A, che contiene le matrici da moltiplicare, restituisce la moltiplicazione ottimale è il seguente:

```
optimal_multiply(S, A, i, j)
  if i == j
    return A[i]

  return optimal_multiply(S, A, i, S[i,j]) *
         optimal_multiply(S, A, S[i,j] + 1, j)
```