

ALGORITMO: Procedura computazionale ben definita
che produce uno o più valori (output)
in funzione di uno o più altri valori (input)
per risolvere problemi computazionali

Paradossalmente non esiste una definizione formale di
Algoritmo in quanto il suo concetto è **elusivo**, essendo
qualcosa di strettamente fisico, su cui è impossibile non
trascurare punti importanti.

Un algoritmo può quindi esserne visto come una **relazione**
tra input e output.

PROBLEMA COMPUTAZIONALE: Relazione tra input e output.
es. **Algoritmo di sorting**

INPUT: $(a_1, a_2, a_3, \dots, a_m)$ sequenza di m numeri

OUTPUT: permutazione $(a_{i_1}, a_{i_2}, a_{i_3}, \dots, a_{i_m})$ di m numeri
tali che $a_{i_1} \leq a_{i_2} \leq a_{i_3} \dots \leq a_{i_m}$

Ogni specifico input si dirà **istanza del problema**

- I problemi di **ordinamento** sono detti **problemI fondamentali**
in quanto essi possono essere spesso **sottoproblemi**
di problemi più grandi.
- Non esistono criteri per stabilire se un problema è
fondamentale. Un algoritmo che risolve un problema
fondamentale è detto **Algoritmo fondamentale**

Gli algoritmi esistevano prima dei calcolatori (es. 1600),
infatti escono con la necessità di risolvere un problema
computazionale per il quale esiste una **soluzione esauriente**

VELOCITA' DI UN ALGORITMO

Dato un problema, si cerca di trovare l'algoritmo migliore anche se non è detto che esista.

- Si considerano due algoritmi a_1 ed a_2 per un medesimo problema computazionale di ordinamento (es. a_1 = insertion sort, a_2 = merge sort)

- Supponiamo che:

$$T_{a_1}(m) = c_1 m^2$$

$$T_{a_2}(m) = c_2 m \log m$$

Dove con T andiamo a considerare il **caso pessimo** e con c_1 e c_2 due costanti più o meno grandi, dipendenti dall'**implementazione**.

- Per valori sufficientemente grandi di m avremo:

■ $T_{a_2}(m) < T_{a_1}(m)$, perché

$$\lim_{m \rightarrow \infty} \frac{c_2 m \log m}{c_1 m^2} = 0 \quad \stackrel{<}{\searrow}$$

per le regole degli infiniti.

Quindi l'algoritmo mergesort è migliore su grandi input ma ciò non è detto per tutti i possibili valori di m .

- Supponiamo ora di avere 2 computer, A veloce (10^9 op/sec) e B lento (10^7 op/sec).

Supponiamo di avere delle implementazioni di a_1 su A e di a_2 su B, tali che:

$$T_{a_1}(m) = \frac{2}{c_1} m^2 \quad \text{e} \quad T_{a_2}(m) = \frac{50}{c_2} m \log m \quad \text{per implementazione}$$

Confrontiamo i tempi di esecuzione con un input di $m = 10^6$

$$A = \frac{2 \cdot (10^6) \text{ ist}}{10^9 \text{ ist/sec}} = 2000 \text{ sec} \quad \text{tempo di esecuzione} \quad \frac{T(m)}{\text{velocità}}$$

$$B = \frac{50 \cdot 10^6 \cdot \log 10^6 \text{ ist}}{10^7 \text{ ist/sec}} \approx 100 \text{ sec}$$

Quindi nonostante i vari vantaggi offerti ad α , α risulta comunque più efficiente, per questo gli algoritmi possono essere considerati come una **Tecnologia**.

INSERTION SORT CORRETTEZZA E COMPLESSITÀ

INPUT: $A[1..n]$ di int

OUTPUT: una permutazione di A in ordine monotonamente decrescente

INSERTION SORT (A)

for ($j=2$ to $A.length$)

do {

 Key = $A[j]$

$i = j - 1$

 while ($i > 0 \text{ e } A[i] > key$)

 do {

$A[i+1] = A[i]$

$i = i - 1$

} f

$A[i+1] = key$

}

Questo algoritmo è **stabile** poiché elementi aventi la stessa chiave mantengono l'ordine relativo (il più a sinistra è il più a dx a dx).

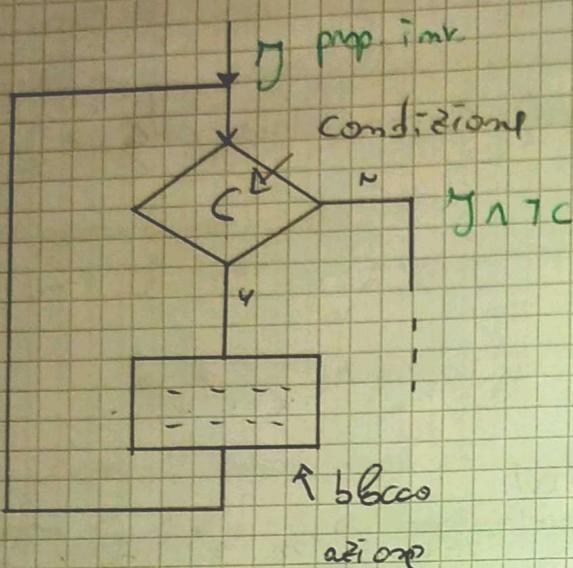
Questa proprietà è importante quando la Key ha altre informazioni collegate.

CORRETTEZZA INSERTION SORT

Si utilizza la tecnica delle proprietà invarianti di ciclo. Dove si va ad analizzare ogni blocco con un ciclo. Questa tecnica è molto dispendiosa perché deve considerare tutti i possibili cicli.

- Per analizzare il ciclo dobbiamo trovare un'opportuna **proprietà invariante** (I). Una proprietà si dice invarianta se, ad ogni ripetizione del ciclo, questa proprietà è vera (potrebbe controllare delle variabili), ovviamente non tutte le proprietà sono utili.
La proprietà invarianta attiva è la formalizzazione del concetto su cui è basato l'algoritmo.

Per dimostrare che una proprietà invarianta è vera, dobbiamo dimostrare che la proprietà sia vera al primo transito e che nei successivi la proprietà rimanga vera, questa modalità ricorda l'**induzione**



2. Se y è vera e passiamo C e il blocco, y continuerà ad essere vera

1. Inizializzazione base

y è vera prima della prima iterazione

2. Mantenimento induttivo

Se y è vera prima dell'esecuzione di un'iterazione del ciclo, rimane vera prima della successiva iterazione

3. Conclusione

quando il ciclo termina vale $y \wedge C$

continua

All'uscita del ciclo varia la proprietà y e la condizione sono falsa (la condizione C potrà essere usata per l'ampliarsi di altri blocchi).

• Questa verifica manipola grandi numeri

CORRETTEZZA DEL CICLO FOR tramite proprietà invarianta di ciclo

• Supponiamo che il ciclo while sia corretto, cioè inserisce l'elemento $A[S]$ nel sottoarray ordinato $A[1 \dots S-1]$

For ($S = z \rightarrow \text{length}(A)$)

Key = $A[S]$

$i = S - 1$

while ($i > 0 \ \& \ A[i] > \text{Key}$) {
 $A[i+1] = A[i]$

{
 $i = i - 1;$

$A[i+1] = \text{Key}$

$y =$ il sottoarray $A[1 \dots S-1]$ è formato dagli elementi ordinati in $A^{(0)}[1 \dots S-1]$ e $1 \leq S-1 \leq \text{length}[A]$

$C = S \leq \text{length}[A]$

NOTA: $A^{(0)}$ è A nelle sue condizioni iniziali.

INIZIAZIONE ($S=2$)

\Rightarrow il sottoarray $A[1..i]$ è formato dagli elementi ordinati in $A^{(o)}[1..i]$
- Banalmente vero

MANTENIMENTO ($S \leq \text{length}[A]$)

Se il sottoarray $\cdot A[1..S-1]$ è formato dagli elementi ordinati di $\cdot A^{(o)}[1..S-1]$, dopo l'esecuzione del ciclo for, $A[S]$ è inserito correttamente in $\cdot A[1..S-1]$ e dunque $A[1..S]$ è formato da elementi ordinati di $\cdot A^{(o)}[1..S]$

CONCLUSIONE ($S < \text{length}[A]$)

A conclusione dell'esecuzione del ciclo for, volgendo a 1. Dunque $1 \leq S-1 \leq \text{length}[A]$ e $S > \text{length}[A] \Rightarrow S = \text{length}[A] + 1$ pertanto: il sottoarray $A[1..(\text{length}[A]+1)-1] = A$ è formato dagli elementi ordinati in $A^{(o)}$, quindi il ciclo for è corretto.

ANALISI COMPLESSITÀ DEGLI ALGORITMI

è la misura delle risorse richieste dall'esecuzione di un algoritmo, quali: tempo di elaborazione, memoria. Si fa riferimento ad un modello di calcolo RAM in cui le istruzioni sono eseguite una per volta.

- Per un dato algoritmo, si cerca una relazione tra la dimensione dell'input e il tempo di elaborazione

DIMENSIONE DELL'INPUT

- può essere:
- numero degli elementi
 - numero di bit per rappresentare l'input
 - coppia di numeri

TEMPO DI ELABORAZIONE

- numero di operazioni primitive eseguite
- Ogni istruzione coinvolge un certo numero di operazioni primitive, ha quindi costo costante

ESEMPIO SULL'INSERTION SORT

1. For $S=2$ to $\text{length}[A]$
2. do $\text{Key} = A[S]$
3. { inserisce $A[S]$ in $A[1..S-1]$ }
4. $i = S-1$
5. while $i > 0$ & $A[i] > \text{Key}$
6. do $A[i+1] = A[i]$
7. $i = i-1$
8. $A[i+1] = \text{Key}$

COSTO

c_1

m

c_2

$m-1$

0

$m-1$

c_4

$m-1$

c_5

$\sum_{S=2}^m T_S$

c_6

$\sum_{S=2}^m (T_S - 1)$

c_7

$\sum_{S=2}^{m-1} (T_S - 1)$

c_8

$T_S = \# \text{ di esecuzioni del test del ciclo while } (S \rightarrow) \text{ per il robes}$

$$T(m) = c_1 m + (c_2 + c_4 + c_8)(m-1) + c_5 \sum_{S=2}^m T_S + (c_6 + c_7) \sum_{S=2}^{m-1} T_S - 1$$

Proprietà invarianti
di ciclo

- Applicazione su insertion sort.

Come già abbiamo che
a ogni iterazione del
for $A[0..S-1]$ è ordinato

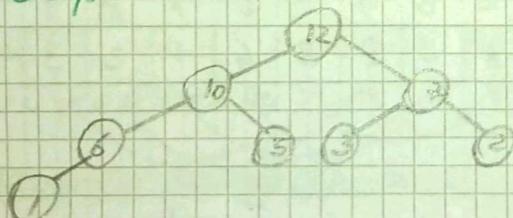
~~il resto non lo è~~

Un **heap binario** è una struttura dati simile ad un albero binario quasi completo (a meno dell'ultimo livello), dove i nodi vanno inseriti da sinistra verso destra, inoltre gode della proprietà dell'**ordinamento parziale**. Per semplicità, l'heap viene rappresentato fisicamente non come un albero ma come un **array** che parte da 1, questa particolarità permette di abbassare notevolmente la sua complessità.

- Un albero binario si dice **completo** quando ogni suo livello è completo, ovvero, quando ogni nodo, escluso le foglie, ha 2 figli, quindi il massimo.
- Possiamo distinguere l'heap in due varianti: simmetriche:
 - **min-heap**, ($x.\text{key}() \geq x.\text{parent}.\text{key}()$)
 - **max-heap**, ($x.\text{key}() \leq x.\text{parent}.\text{key}()$)

Le proprietà di ordinamento tra padre e figli, sono dette proprietà dell'**ordinamento parziale**, in quanto si sta definendo un ordinamento solo tra padre e figli e non per tutto l'albero come per i BST.

EX: max-heap



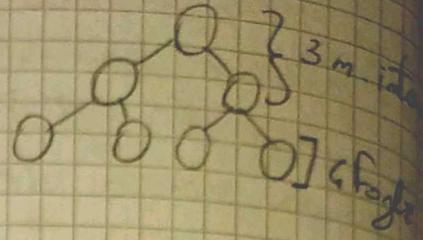
Nel max-heap, la chiave più grande si trova sempre in testa mentre il valore più piccolo sono sempre nelle foglie, per i min-heap vale l'esatto contrario.

In un heap binario le chiavi vengono distribuite sempre da sinistra verso destra, livello per livello.

- Nel max-heap, seguendo qualsiasi path dalla root alle foglie avremo una sequenza decrescente.

PROPRIETA' DEGLI HEAP BINARI

- Dato un heap completo di altezza h :
 - Le foglie saranno 2^h
 - i nodi interni $2^h - 1$



- Dato un heap completo con m modi:
 - avranno $m/2$ foglie
 - e $\lfloor m/2 \rfloor$ modi interni (Floor: arrotondato a int più piccolo)

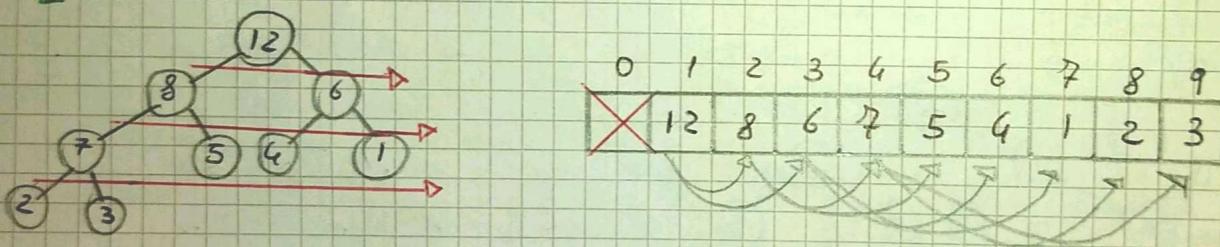
Possiamo quindi stabilire che

- per trovare la Key $> (o <)$ il tempo sono 1 , poiché sarà la root.
- il tempo per cercare la Key $< (o >)$ sono $m/2$, quindi $O(m)$, poiché dovranno scendere alle foglie.

RAPPRESENTAZIONE DI UN HEAP

L'heap è una struttura dati più rigorosa rispetto ad un normale albero e, per facilitare l'implementazione e migliorarne la complessità si utilizza l'**array**, il cui indice parte da 1 per questioni di operazioni sulla struttura.

EX:



Disponiamo i numeri dell'array da sinistra verso destra completando ogni livello.

È possibile tracciare le parentele dei nodi anche sull'array:

- dato elemento $i = \text{padre}$:

- $\text{left}(i) = 2i$;
- $\text{right}(i) = 2i + 1$;

- per calcolare il padre, dato i figlio:

- $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$ (Floor di $i/2$ per arrotondare dx disponibile)

Ex: dato l'array precedente

$$i=7, \text{figlio}, i=3: \text{padre} \Rightarrow \lfloor \frac{7}{2} \rfloor = \lfloor 3,5 \rfloor = 3.$$

Per garantire la minima complessità computazionale, è possibile le sfruttare le operazioni di shiftimg su indici espressi in sistema binario, usando così un solo ciclo di clock per operazione:

- CALCOLO LEFT(i)

$$\text{left}(i) = i \ll 1 \quad // \text{shift 1 a sx} = \text{moltiplicazione}$$

- CALCOLO RIGHT(i)

$$\text{right}(i) = i \ll 1$$

$$\text{right}(i) = i \& 100001 \quad // \text{OR con maschera}$$

- CALCOLO PARENT(i)

$$\text{parent}(i) = (i \gg 2) \quad // \text{Floor di shift a destra}$$

CODE CON PRIORITÀ

La coda è una struttura elementare basata sulla politica FIFO, a volte però è possibile assegnare una priorità agli elementi di una coda, un esempio può essere il sistema dei codici al pronto soccorso.

Per gestire le priorità, possiamo sia assegnare un "codice" al modo che ordinare per grandezza di key, a parità di key entra in gioco la politica FIFO.

Ad ogni coda, possiamo associare un max-heap, notando che la root, sarà sempre il primo elemento e con priorità più elevata.

OPERAZIONI SU CODE DI PRIORITÀ ASSOCIATE AD HEAP

MAX-HEAP

- $O(1)$ $\lceil \text{maximum}()$
- $\lceil \text{extract_max}()$

- $O(\log m)$ $\lceil \text{insert}()$
- $\lceil \text{increase_key}()$

MIN-HEAP

- $\lceil \text{minimum}()$
- $\lceil \text{extract_min}()$
- $\lceil \text{insert}()$
- $\lceil \text{decrease_key}()$

- MAXIMUM & MINIMUM: ritornano la root dell'heap che corrisponde all'elemento con priorità maggiore

- INCREASE-DECREASE: aumentano la priorità dell'elemento

- INSERT: modifica la struttura dell'albero.

ESEMPIO DI CODA A PRIORITA' TRAMITE PESO DEL PROBLEMA

1	<table border="1"><tr><td>6</td><td>3</td><td>1</td></tr></table>	6	3	1		
6	3	1				
2	<table border="1"><tr><td>6</td><td>3</td><td>1</td></tr></table> ← 7	6	3	1		
6	3	1				
3	<table border="1"><tr><td>7</td><td>6</td><td>3</td><td>1</td></tr></table>	7	6	3	1	
7	6	3	1			
4	<table border="1"><tr><td>7</td><td>6</td><td>3</td><td>1</td></tr></table> ← 7	7	6	3	1	
7	6	3	1			
5	<table border="1"><tr><td>7</td><td>7</td><td>6</td><td>3</td><td>1</td></tr></table>	7	7	6	3	1
7	7	6	3	1		

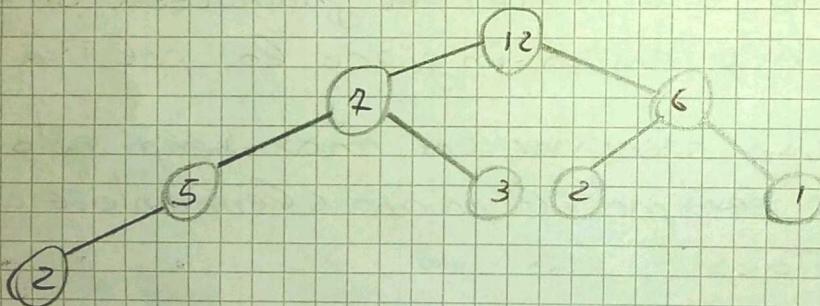
// a pari priorita' => FIFO.

ESERCIZIO

Verificare se il seguente array è un max-heap

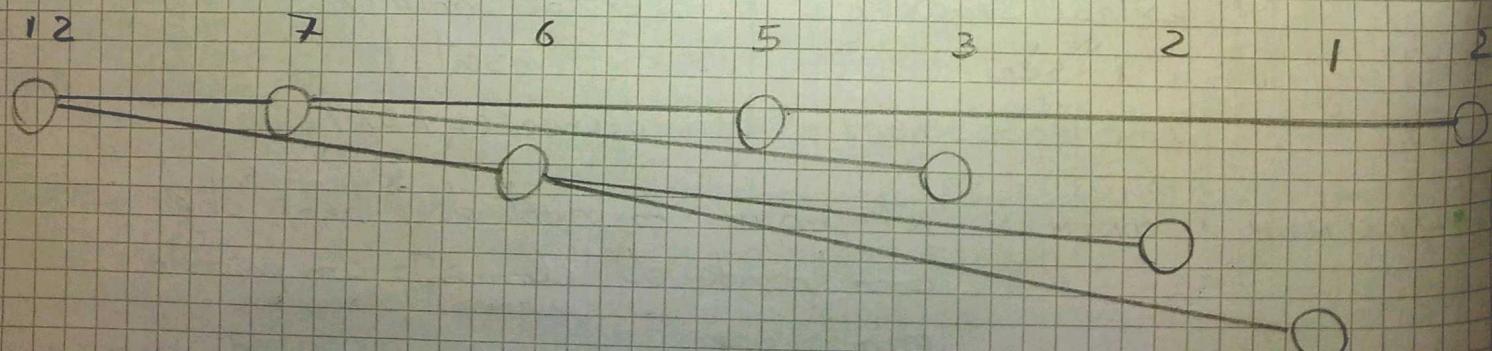
12	7	6	5	3	2	1	2
----	---	---	---	---	---	---	---

Basta riempire un heap da dx verso sx e verificare la proprietà dell'ordinamento parziale



L'array è un max-heap poiché :

$\forall x$ modo generico $\Rightarrow x.\text{key}() \leq x.\text{parent}.\text{key}()$



COMPLESSITÀ INSERTION SORT

```

1. For  $s=2$  to  $\text{length}[A]$ 
2.   do  $\text{key} = A[s]$ 
3.     { inserisce  $A[s]$  in  $A[i..s-1]$ }
4.      $i = s - 1$ 
5.     while  $i > 0$  and  $A[i] > \text{key}$ 
6.       do  $A[i+1] = A[i]$ .
7.        $i = i - 1$ 
8.    $A[i+1] = \text{key}$ 

```

$1 \leq s \leq 5$	COSTO	# ESECUZIONI
	c_1	m
	c_2	$m-1$
	0	$m-1$
	c_4	$m-1$
	c_5	$\sum_{s=2}^5 t_s$
	c_6	$\sum_{s=2}^m (t_s - 1)$
	c_7	$\sum_{s=2}^m (t_s - 1)$
	c_8	$m-1$

- Nella prima istruzione il for viene eseguito in volta e nell' m -esima passata non possiede il controllo quindi, verrà interrotto, le altre istruzioni all'interno del for verranno eseguite $m-1$ volte.
- Le istruzioni 5, 6, 7 costituiscono il corpo del while e verranno eseguite un numero t_s di volte con $1 \leq s \leq 5$. Il while sarà eseguito precisamente t_s volte, contando anche l'ultimo ciclo che non possiede il controllo, mentre le operazioni all'interno verranno eseguite $t_s - 1$ volte.
- Sommando i costi di ogni istruzione moltiplicati per il numero di volte che un'istruzione viene eseguita, avremo il costo totale dell'algoritmo $T(m)$.

N.B. Anche per input della stessa dimensione m , i valori t_s vogliono per il particolare input.

CASO MIGLIORE (BEST CASE ANALYSIS)

Si ha quando l'array di input è già ordinato in senso non decrescente.

In questo caso, $t_s = 1$ per $s = 1, 2, \dots, m$, avremo quindi che la complessità totale si ridurrà a

$$T(m) = (c_1 + c_2 + c_4 + c_5 + c_8)m - (c_2 + c_4 + c_5 + c_8)$$

Cioè: $T(m) = Am + B$, con A e B costanti.

Dunque, in questo caso, $T(m)$ è lineare.

CASO PEGGIORE

(WORST CASE ANALYSIS)

Si ha quando l'input è ordinato in modo decrescente.
In questo caso $t_S = S$ per $S = 2, 3, \dots, m$, e quindi, facendo uso dell'identità:

$$\sum_{S=1}^m S = \frac{m(m+1)}{2}$$

avremo:

$$T(m) = \frac{1}{2} (c_5 + c_6 + c_7) m^2 + (c_1 + c_2 + c_4 + \frac{c_3}{2} + \frac{c_5}{2} + \frac{c_7}{2} + \frac{c_8}{2}) m - (c_2 + c_4 + c_5 + c_6)$$

Cioè:

$$T(m) = Am^2 + Bm + C, \text{ con } A, B, C \text{ costanti.}$$

In questo caso $T(m)$ è quadratico.

- Nel caso peggiore il while scatta sempre e l'algoritmo lavora al massimo

In generale ci limiteremo ad analizzare il caso peggiore in quanto

- rappresenta il limite superiore del tempo di esecuzione per qualsiasi input.

- in molti casi si verifica spesso il caso peggiore,
- il caso medio è spesso cattivo quanto il peggiore, ad esempio ($t_S = S/2$ in media, sempre m^2).
- Inoltre l'analisi del caso medio richiede tecniche di **analisi probabilistica**

In generale siamo interessati all'ordine di crescita della funzione **tempo di esecuzione**, quindi possiamo ignorare le costanti e i termini di ordine inferiore.

Possiamo dunque dire che la complessità dell'insertion sort è:

- $\Theta(m)$, nel caso migliore
- $\Theta(m^2)$, nel caso peggiore

APPROCCIO INCREMENTALE

L'insertion sort è basato sull' **approccio incrementale**, in quanto il problema viene risolto passo dopo passo, infatti, l'array **A** viene ordinato in modo incrementale:

$A[1..2]$, $A[1..3]$, $A[1..m-1]$, $A[1..m]$

DIVIDE ET IMPERA

Il paradigma **divide et impera** prevede **3 passi** ad ogni livello di ricorsione:

DIVIDE: il problema viene suddiviso in un certo numero di sotto-problemi (tutti della stessa natura)

IMPERA: ciascun problema è risolto in maniera ricorsiva (o in maniera diretta, se sufficientemente piccolo).

COMBINA: le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale.

Se anche solo un passo non è applicabile, questa strategia non può essere usata per un dato problema.

MERGE SORT

DIVIDE: la sequenza degli elementi da ordinare è divisa in **2 sottosequenze** di **$\frac{m}{2}$** (circa) elementi.

IMPERA: ciascuna sottosequenza è ordinata ricorsivamente

COMBINA: le 2 sottosequenze ordinate sono fuse in un'unica sequenza ordinata

L'algoritmo si divide in 2 parti, una si occupa di scomporre in sotto problemi e l'altra di ricomporli in maniera ordinata.

CODIFICA

MERGE-SORT(A, p, r){

 IF(p < r)

 then $q = \lfloor (p+r)/2 \rfloor$

 merge-sort(A, p, q)

 merge-sort(A, q+1, r)

 merge(A, p, q, r)

}

MERGE(A, p, q, r){

$m_1 = q - p + 1$

$m_2 = r - q$

CREA $L[1..m_1+1]$ E $R[1..m_2+1]$ //array di supporto

For $i = 1$ to m_1

do $L[i] = A[p+i-1]$ //copia $A[p..q]$ in $L[1..m_1]$

For $j = 1$ to m_2

do $R[j] = A[q+j]$ //copia $A[q+1..r]$ in $R[1..m_2]$

$L[m_1+1] = +\infty$

{ //sentinella

$R[m_2+1] = +\infty$

$i = 1$

$S = 1$

For $K = p$ to r

do ; $F(L[i] \leq R[S])$

then $A[K] = L[i]$

$i = i + 1$

else $A[K] = R[S]$

$S = S + 1$

COMPISSITÀ: $\Theta(m)$, $m = r - p + 1$

}

Se $p = r$, l'array ha un solo elemento, è quindi ordinato.

- Esistono codifiche del mergesort in-place con stessa complessità.

• Si intuisce che l'algoritmo da la possib. d'ta' di essere eseguito su più processori contemporaneamente.

Supponendo di avere m/p processori, la complessità sono lineare.

AMALI SI DEGLI ALGORITMI DIVIDE ET IMPERA

- $t(m)$ = tempo di esecuzione su input di dimensione m
- a = numero di sotto problemi
- m/b = dimensione di ciascun sotto problema
- s = soglia al di sotto della quale non c'è ricorsione
- $\delta(m)$ = tempo per **dividere** il problema in sotto problemi
- $C(m)$ = tempo per **combinare** le soluzioni dei sotto problemi nella soluzione del problema originale

Si ottiene la seguente **riconvenza**

$$t(m) = \begin{cases} \Theta(1) & \text{se } m \leq s \\ a + \left(\frac{m}{b}\right) + \delta(m) + C(m) & \text{se } m > s \end{cases}$$

NEL CASO DEL **MERGESORT** SI HA:

$$\alpha = 2$$

$$b = 2$$

$$\delta(m) = \Theta(m)$$

$$C_m = \Theta(m) \quad (\text{merge})$$

Quindi $t(m)$ soddisfa la seguente riconvenza:

$$t(m) = \begin{cases} \Theta(1) & \text{se } m = 1 \\ 2t\left(\frac{m}{2}\right) + \Theta(m) & \text{se } m > 1 \end{cases}$$

che ha soluzione:

$$t(m) = \Theta(m \log m)$$

METODO DI SOSTITUZIONE

Riscriviamo la ricchezza come:

$$T(m) = \begin{cases} c & \text{se } m=1 \\ 2T\left(\frac{m}{2}\right) + cm & \text{se } m \geq 1 \end{cases}$$

REGOLE DI RISCRITTURA

$$T(k) = ck + 2T\left(\frac{k}{2}\right), k \geq 2$$

$$T(1) = c$$

$$\begin{aligned} T(m) &= cm + 2T\left(\frac{m}{2}\right) = \\ &= cm + 2\left(cm_2 + 2T\left(\frac{m}{2^2}\right)\right) = \frac{m}{2} = \frac{m}{2^k} \\ &= 2cm + 2^2\left(cm_{2^2} + 2T\left(\frac{m}{2^3}\right)\right) = \\ &= 2cm + 2^2\left(cm_{2^2} + 2T\left(\frac{m}{2^3}\right)\right) = \\ &= 3cm + 2^3\left(cm_{2^3} + 2T\left(\frac{m}{2^4}\right)\right) = \\ &\vdots \\ &= km + 2^k\left(cm_{2^k} + 2T\left(\frac{m}{2^{k+1}}\right)\right) = \\ &= cm \log_2 m + mT(1) = \\ &= cm \log_2 m + cm \end{aligned}$$

$$\boxed{\text{supp. } m = 2^k \\ k = \log_2 m}$$

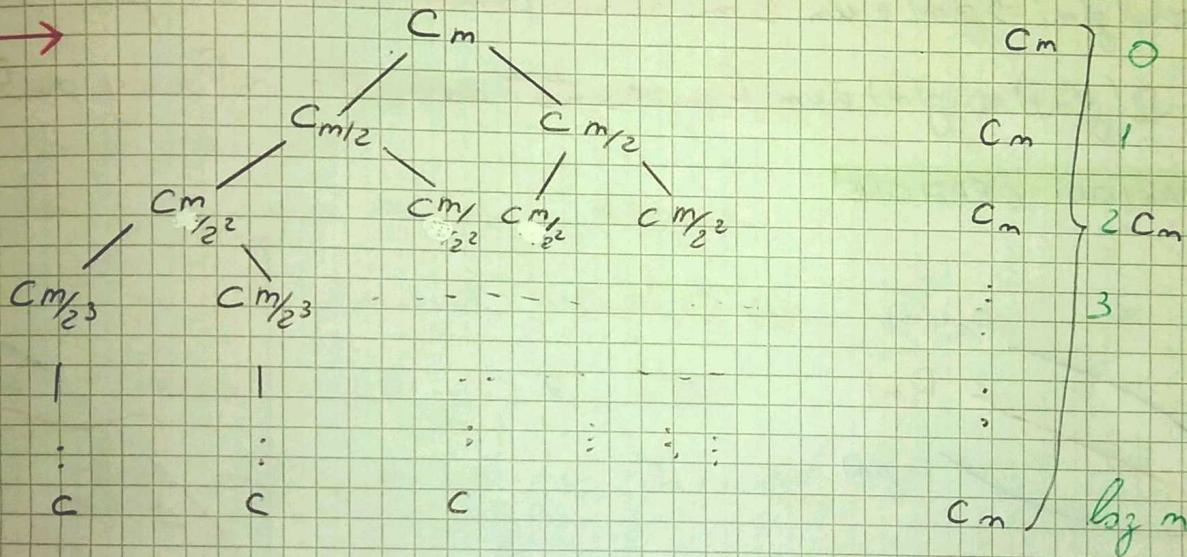
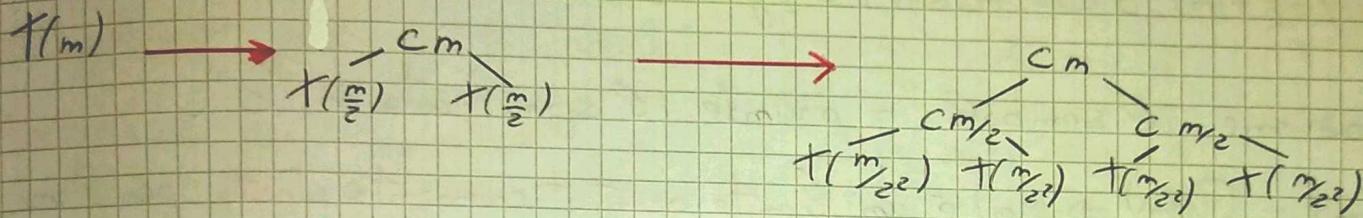
METODO DI SOSTITUZIONE SU ALBERO DI RICORSIONE

Il metodo di sostituzione è più potente se applicato sull'albero di ricorsione.

REGOLE DI RISCRITURA

$$\frac{T(i)}{c} = \frac{T(k)}{ck}$$

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



Supp. $m = 2^k$, $k = \log_2 m$

- # livelli = $k + 1 = \log_2 m + 1$
- $T(m) = cm(\log_2 m + 1) = cm \log_2 m + cm$

L'altezza dell'albero è proporzionale al numero di modi, con m modi l'altezza sarà $\log m$.

- La somma dei valori dell'albero è sempre $T(m)$, andremo avanti fino al caso base sostituendo con c .
- La complessità sarà data dalla somma di tutti i livelli ($\log m$) più il livello 0 $\Rightarrow \log m + 1$.

NOTAZIONI ASINTOTICHE

Vengono usate per caratterizzazione il tasso di crescita del tempo di esecuzione di un dato algoritmo

- Sia $g: \mathbb{N} \rightarrow \mathbb{N}$:

$$\Theta(g(m)) = \{F(m) : \exists c_1, c_2 > 0, m_0 \in \mathbb{N} \mid 0 \leq c_1 g(m) \leq F(m) \leq c_2 g(m) \quad \forall m \geq m_0\}$$

$$O(g(m)) = \{F(m) : \exists c > 0, m_0 \in \mathbb{N} \mid 0 \leq F(m) \leq c g(m) \quad \forall m \geq m_0\}$$

$$\Omega(g(m)) = \{F(m) : \exists c > 0, m_0 \in \mathbb{N} \mid 0 \leq g(m) \leq F(m) \quad \forall m \geq m_0\}$$

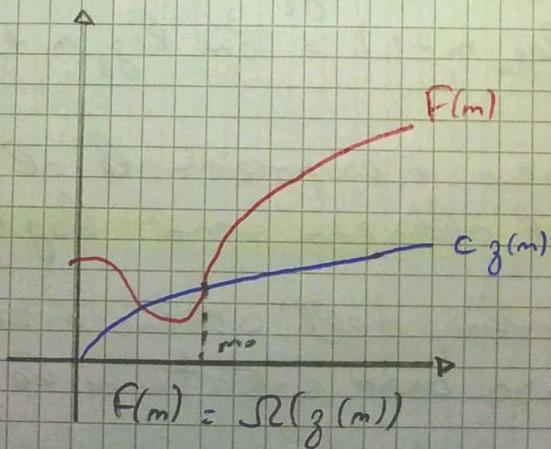
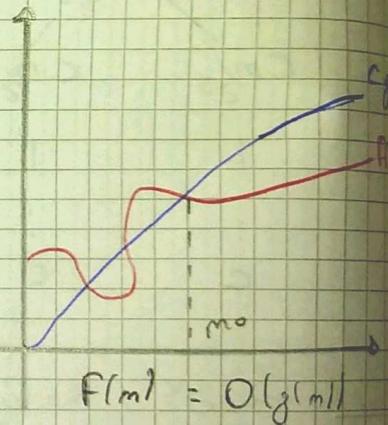
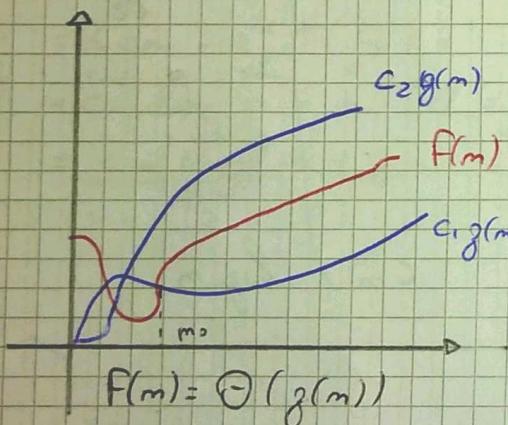
Per notazione, il simbolo " \sim " = "prende il significato di " ϵ ".

- Se $F(m) = \Theta(g(m)) \Rightarrow g(m)$ è un limite asintoticamente stretto per $F(m)$.

- Se $F(m) = O(g(m)) \Rightarrow g(m)$ è un lim. asintoticamente superiore per $F(m)$.

- Se $F(m) = \Omega(g(m)) \Rightarrow g(m)$ è un lim. asintoticamente inferiore per $F(m)$.

RAPPRESENTAZIONI GRAFICHE



ESEMPI

$$\frac{1}{2}m^2 - 3m = \Theta(m^2)$$

Occorre determinare che $\exists c_1, c_2 > 0$ e $m_0 \in \mathbb{N}$:

$$c_1 m^2 \leq \frac{1}{2}m^2 - 3m \leq c_2 m^2 \quad \forall m \geq m_0$$

Dividiamo per m^2

$$c_1 \leq \frac{1}{2} + \frac{3}{m} \leq c_2$$

$$\text{scogliendo } c_2 \geq \frac{1}{2} \Rightarrow \frac{1}{2} + \frac{3}{m} \leq c_2 \text{ vol } m \geq 1$$

$$\text{scogliendo } 0 \leq c_1 \leq \frac{1}{16} \Rightarrow c_1 \leq \frac{1}{2} + \frac{3}{m} \text{ vol } m \geq 7$$

$$\text{DUNQUE} \Rightarrow \frac{1}{16}m^2 \leq \frac{1}{2}m^2 - 3m \leq \frac{1}{2}m^2, \quad \forall m \geq 7$$

EX2:

$$6m^3 \neq \Theta(m^2)$$

se fosse vero, esisterebbero $c_2 > 0$ e $m_0 \in \mathbb{N}$ | $6m^3 \leq c_2 m^2$, dividendo per m^2

$$\Rightarrow 6m \leq c_2 \text{ per } m \geq m_0, \text{ ASSURDO}$$

$$\frac{\geq}{\leq} = \infty, \frac{\leq}{\geq} = 0$$

LEMMA \lim per es. met. asintotica

Se si sente i calcoli

Siamo $F: \mathbb{N} \rightarrow \mathbb{N}$ e $g: \mathbb{N} \rightarrow \mathbb{N}^+$

$$(a). \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = \alpha > 0 \Rightarrow f(m) = \Theta(g(m))$$

$$(b). \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = 0 \Rightarrow f(m) = O(g(m)) \wedge f(m) \neq \Omega(g(m))$$

$$(c) \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = +\infty \Rightarrow f(m) = \Omega(g(m)) \wedge f(m) \neq O(g(m))$$

Se il \lim -te non esiste dobbiamo usare l'altra tecnica

COROLARIO

Siamo $f: \mathbb{N} \rightarrow \mathbb{N}$ e $g: \mathbb{N} \rightarrow \mathbb{N}^+$

$$(a) \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = \alpha > 0 \Rightarrow f(m) = \Theta(g(m))$$

$$(b) \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = \alpha \geq 0 \Rightarrow f(m) = O(g(m))$$

$$(c) \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} > 0 \Rightarrow f(m) = \Omega(g(m))$$

EX: Sia $P(m) = \sum_{i=0}^d \alpha_i m^i$ un polinomio di grado d con $\alpha_d > 0$

AUORA:

$$P(m) = \Theta(m^d)$$

$$P(m) = O(m^\alpha), \forall \alpha \geq d$$

$$P(m) = \Omega(m^\beta), \forall 0 \leq \beta \leq d$$

Per il corollario precedente basta osservare che:

$$\bullet \lim_{m \rightarrow +\infty} \frac{P(m)}{m^d} = \alpha_d > 0$$

$$\bullet \lim_{m \rightarrow +\infty} \frac{P(m)}{m^\alpha} = \alpha > 0, \forall \alpha \geq d$$

$$\bullet \lim_{m \rightarrow +\infty} \frac{P(m)}{m^\beta} > 0, \forall 0 \leq \beta \leq d$$

Con un leggero abuso di notazione scrivremo che:

$$\Theta(1) \text{ al posto di } \Theta(m^0)$$

Per ogni costante $c > 0 \Rightarrow c = \Theta(1)$

PROPRIETA'

$$\bullet \Theta(g(m)) \subseteq O(g(m))$$

$$\bullet \Theta(g(m)) \subseteq \Omega(g(m))$$

$$\bullet \Theta(g(m)) = O(g(m)) \cap \Omega(g(m))$$

$f(x) =$ mostra f.
 $g(x) =$ comparsa per

OPERAZIONI SU HEAP BINARIO

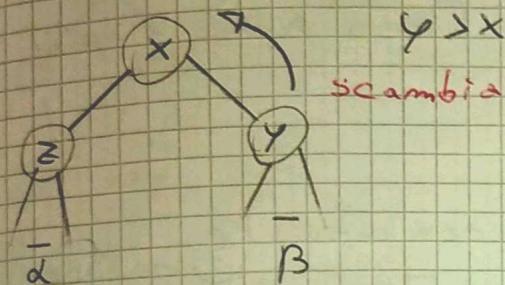
LEZ 4 FARO

INSERT

Ogni volta che inseriamo un nodo, dobbiamo sia
sia la struttura che l'ordinamento parziale.

Il nuovo nodo va posizionato alla fine come foglio
e poi si fa risalire fino a quando non troviamo
la sua posizione corretta.

Il modo inserito salina fino a quando sopra
non avrò un nodo con una Key più grande



L'operazione di scambio è safe perché essendo
già la struttura un heap ed essendo $y > x$ non provocherà
problemi sopra.

Non sarà neanche un problema per z in quanto
 $z < x < y \Rightarrow z < y$, la proprietà rimane quindi inviolata

CONIFICA

INSERT (A, k) {

heapsize++; //La root parte da pos A[1]

i = heapsize; //A[heapsize] = k mette k come ultimo elemento

while (i > 1 & A[parent(i)] < A[i]) { //Fino a quando sono dentro

swap (A, i, parent(i)); //L'elenco d il valore del padre

i = parent(i); //è < di quello che sta sotto (Figlio)

return A;

B

C

//Scambia i e suo padre

//Aumenta l'indice

A è l'array che contiene fisicamente l'heap, la struttura
ad albero è virtuale

heapsize è il numero di elementi in dell'heap.

A[heapsize] = ultima posizione dell'heap padre parte da 1

ESEMPIO

1	2	3	4	5	6	7	8	9
16	10	9	7	6	5	3	2	14

// 1-8 heap, 9 nuovo elemento

$$p(i) = \lfloor i/2 \rfloor$$

16	10	9	14	6	5	3	2	7
----	----	---	----	---	---	---	---	---

$$i$$

$$p(i) = \lfloor i/2 \rfloor$$

16	16	9	10	6	5	3	2	7
----	----	---	----	---	---	---	---	---

// continuo a scambiare fino a quando $i > p(i)$

$p(i) > i \Rightarrow$ l'heap rispetta le proprietà

- La complessità sarà al massimo $\Theta(m)$ (altezza dell'albero)
e nel caso migliore, quando il nuovo elemento è già posizionato.

MAXIMUM

MAXIMUM(A) {

 return A[0]; // ritorna la root ovvero l'elemento più grande

}

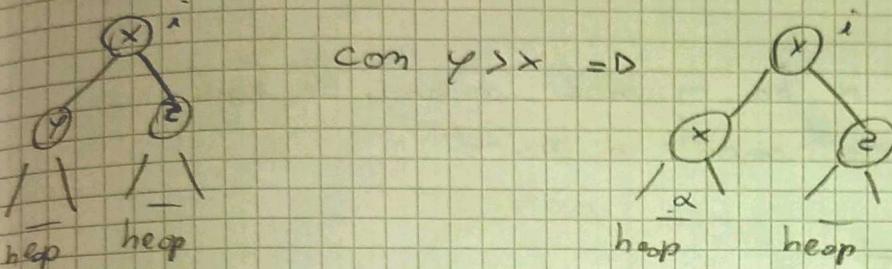
La complessità sarà $O(1)$

HEAPIFY

La procedura heapify rende un heap una struttura sot. Heapify puo' essere chiamata solo su nodi i cui figli sono già heap.

La procedura consiste nel prendere il maggiore tra lo key di un modo e de' suoi 2 figl. e mettere come root, il modo con la Key piu' grande

Ex:



Problema:

Avendo inserito x al posto di y potrei aver causato delle violazioni su α , perche':

$\cdot y > \alpha$.

$\cdot y > x$

$\cdot x > \alpha$??.

Per risolvere il problema chiamiamo heapify conservamente su x , alla fine sarei sicuro di aver creato un heap

CODIFICA

HEAPIFY(A, i) {

$l = \text{left}(i);$

$r = \text{right}(i);$ //assegna i valori

$\max = i;$

$\text{if } l \leq \text{heapsize} \text{ dd } A[l] > A[\max] \text{ } //\text{se esiste (e dentro l'olbero) e}$
 $\max = l$ il suo val > i, lo assegna

$\text{if } r \leq \text{heapsize} \text{ dd } A[r] > A[\max]$

$\max = r;$

$\text{if } (\max \neq i) \quad //\text{se effettivamente esistono vali piu' grandi.}$
 $\text{swap}(A, i, \max) \quad //\text{lo scambia}$

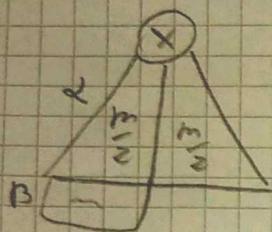
HEAPIFY(A, \max) //chama heapify sul figlio di i con
il quale e' avvenuto lo swap

EQUAZIONE DI RICORRENZA DI HEAPIFY

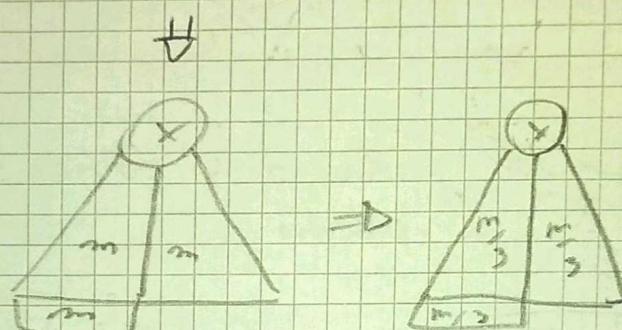
Indicando con m il numero di elementi dell'albero e con $T(m)$ il tempo di esecuzione della procedura

CASO PEGGIORE

Il caso peggiore si verifica quando chiamiamo la procedura sul sottoalbero di sinistra, poiché, nel sottoalbero di sinistra vanno a posizionarsi i nodi del livello incompleto, in quanto quest'ultimo si riempie da sinistra verso destra.



Sappiamo che se α ha m modi interni, le foglie di β saranno esattamente m , quindi:



L'heap sarà diviso in 3 parti uguali e chiamando la procedura sul sottoalbero sinistro, prenderemo $\frac{2}{3}m$ modi.

L'equazione di ricorrenza nel caso peggiore sarà quindi:

$$T(m) = O(1) + T\left(\frac{2}{3}m\right) \Rightarrow T(m) = O(\log m)$$

costo $\xrightarrow{\text{F}}$ chiamata ricorsiva
su sx

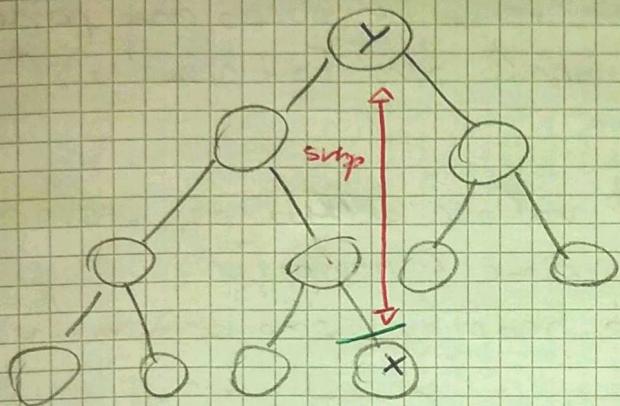
heapiify serve a riorganizzazione l'heap, ripristinando le proprietà:

- Albero quasi completo da sx a dx
- prop. ordinamento parziale
- Se ha m modi $\Rightarrow O(\log m)$ } perché un heap con altezza $\lceil \log m \rceil$ ha $2^{\lceil \log m \rceil}$ modi
- se è alto $m \Rightarrow O(m)$

ESTRAZIONE DEL MASSIMO

Quando estraggo il massimo, non faccio altro che rimuovere la root dall'heap, ma essa dovrà essere rimpiazzata.

Dopo aver rimosso la root, insisto l'ultimo modo inserito (foglia) al posto della root [lo avrei spostato a prescindere perché avrei dovuto soddisfare la proprietà di completezza con un modo immenso], infine chiamo heapify per ristabilire la proprietà di ordinamento parziale.



Scambio x testa con y ultimo modo, rimuovo la foglia che contiene x e chiamo heapify sulla root per ristabilire le proprietà:

CODIFICA

EXTRACT-MAX(A)

```

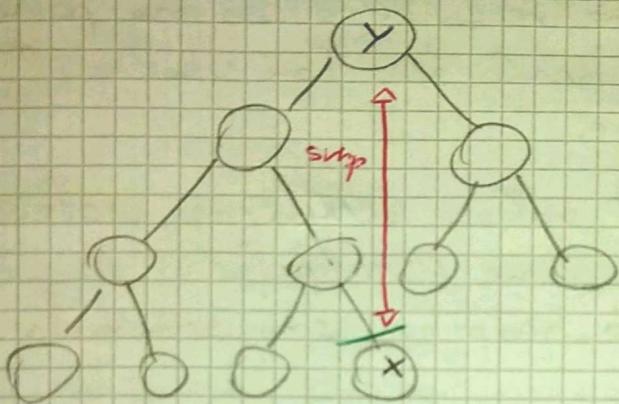
01) swap(A, i, heapsize) //scambia la root con l'ultimo el.
    { heapsize--;
      cancella l'ultimo elemento
    }
02) { heapify(A, i) //chiamo heapify sulla root
      returnm (A[heapsize + 1]) //torna l'elemento estratto che
                                //sarebbe una posizione fuori
                                //il bound
    }
  
```

Ad ogni scambio tra la root e l'ultimo elemento, la root sarà posizionata come ultimo elemento

ESTRAZIONE DEL MASSIMO

Quando estraggo il massimo, non faccio altro che rimuovere la root dall'heap, ma essa dovrà essere rimpiaccata.

Dopo aver rimosso la root, inserisco l'ultimo modo inserito (foglia) al posto della root [lo avrei spostato a prescindere perché avrei dovuto soddisfare la proprietà di completezza con un modo immemo], infine chiamo heapify per ristabilire la proprietà di ordinamento parziale.



Scambio x testo con y ultimo modo, rimuovo Ra Foglia che contiene x e chiamo heapify sulla root per ristabilire le proprietà.

CODIFICA

EXTRACT-MAX(A)

```

a) { swap(A, 1, heapsize) //scambia la root con l'ultimo el.
    { heapsize --, //cancella l'ultimo elemento
      b) { heapify(A, 1) //chiamo heapify sulla root
           return A[heapsize + 1] //ritorno l'elemento estratto che
           sara' una posizione fuori
           il bound
    }
  }

```

Ad ogni scambio tra la root e l'ultimo elemento, la root sarà posizionata come ultimo elemento

INCREASE KEY

CODIFICA

INCREASE_KEY(A, k)

$$A[i] = k$$

while ($i > 1$ $\&$ $A[\text{parent}(i)] < A[i]$) {

 swap($A, i, \text{parent}(i)$)

} $i = \text{parent}(i)$

Con $A[i] < k$ prima della procedura

- Se $k > A[i]$ non ho violazione verso il basso ma potrei averne verso l'alto, per sistemarlo devo far uscire il modo fino alla sua posizione.
- Se volessi decrementare il valore di $A[i]$ avrei violazione verso il basso, dovrò quindi usare l'heapify.
Le operazioni di risolta e heapify sono simmetriche
 - Quando inserisco un valore più piccolo, potrei avere problemi con il basso (heapify)
 - Quando inserisco un valore più grande potrei avere problemi con l'alto (risolta del modo).

COMPLESSITA' DELLE STRUTTURE

	INSERT $O(n)$	MAX $O(m)$	EXTRACT $O(m)$	INCREASE VAL $O(1)$
ARRAY non ordinato	$O(n)$	$O(m)$	$O(1)$	$O(n)$
ARRAY ordinato	$O(n)$	$O(1)$	$O(1)$	$O(n)$
BST degenere	$O(n)$	$O(m)$	$O(m)$	$O(n)$
BST bilanciato	$\textcircled{H} O(\log m)$	$O(\log m)$	$O(\log m)$	$O(\log n)$
HEAP binario	$O(\log m)$	$O(1)$	$O(\log m)$	$O(\log n)$

Dalla motivazione l'heap è il miglior poiché a differenza del BST che ha come caso medio $\log n$, l'heap ha $\log n$ come caso pessimo, ovvero in genere la complessità è minore.

EX:

SELECTION SORT

Generalmente il selection sort ordina dal minimo, potrebbe però ordinare gli elementi posizionando prima il massimo, poiché cercare il massimo in un array disordinato costa $O(m)$, per m elementi: $m + (m - 1) + (m - 2) + \dots + 3 + 2 + 1$, ovvero

$$= \sum_{i=1}^m i = \frac{m(m+1)}{2} = O(m^2)$$

Guardando lo stesso problema su una coda a priorità, dove semplicemente estrane ogni volta l'elemento in testa oppure su un heap, dove vira all'algoritmo **heapsort**.

HEAPSORT

Se usassi un heap come struttura per il select, omsort, mi basterebbe estrarre il max, mettelo in coda all'array e ne effettuare l'estrazione m volte su array di lunghezza sempre più piccola.

CODIFICA

HEAPSORT(A, m)

BUILD-HEAP(A, m) // crea l'heap sull'array

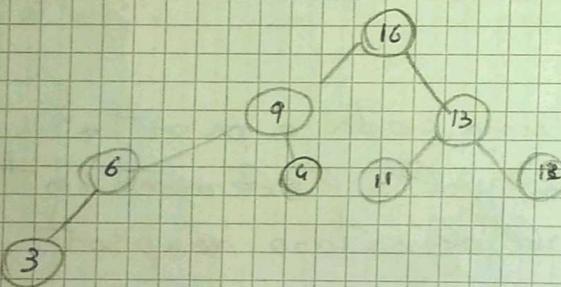
For $i = 1$ to $m-1$ // per tutti gli elementi dell'heap ($m-1$ volte)

EXTRACT-MAX(A) // estrae il max, lo mette in coda e fa heapify

- L'estrazione del massimo mette già in coda all'array l'elemento estraotto che sarà poi ignorato diminuendo l'heapsize. Alla stessa tempo l'elemento estraotto sarà già ordinato.
- Alla fine dell'esecuzione, l'heap si scatterà e l'array su cui giaceva sarà trasformato in un array ordinato.

ES:

10	9	13	6	4	11	2	3
----	---	----	---	---	----	---	---



Quindi:

1. estraie la testa e la mette a heapsize
2. diminuiamo heapsize di 1 e chiamiamo heapify
3. riapplica EXTRACT-MAX fino a quando l'heap sarà vuoto

Eseguendo m volte un'operazione che costa $\log m$ avrà una complessità di $O(m \log m)$

Dato che gli algoritmi basati su confronti hanno complessità $\Omega(m \log m)$, l'heapsort è ottimale.

A differenza del mergesort, l'heapsort è in-place, l'unico problema è che esso non è stabile.

Un algoritmo si dice stabile se mantiene l'ordinamento parziale fra chiavi con lo stesso valore.

PROPRIETA' NOTAZIONE ASINTOTICA

Per valere $\mathcal{O}(f(m))$, deve valere che $\mathcal{R}(f(m)) = \mathcal{O}(f(m))$

NOTAZIONE O - PICCOLO

$$\mathcal{O}(g(m)) = \left\{ f(m) : \forall c > 0 \exists m_0 > 0 : 0 \leq f(m) \leq c g(m), \forall m \geq m_0 \right\}$$

Si osservi che $\mathcal{O}(m^2) \neq m^2$

PROPRIETA'

$$0 \leq f(m) \leq c g(m) \Leftrightarrow 0 \leq \frac{f(m)}{g(m)} \leq c.$$

Quindi:

Per $f: \mathbb{N} \rightarrow \mathbb{N}$, $g: \mathbb{N} \rightarrow \mathbb{N}^+$ si ha

$$\cdot f(m) = \mathcal{O}(g(m)) \Leftrightarrow \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = 0 \quad F \text{ più lento di } g$$

NOTAZIONE \omega - PICCOLO

$$\omega(g(m)) = \left\{ f(m) : \forall c > 0 \exists m_0 > 0 : 0 \leq c g(m) \leq f(m), \forall m \geq m_0 \right\}$$

Si osservi che $m^2 \neq \omega(m^2)$

PROPRIETA'

$$0 \leq c g(m) \leq f(m) \Leftrightarrow 0 \leq \frac{f(m)}{g(m)} \leq c$$

Quindi:

per $f: \mathbb{N} \rightarrow \mathbb{N}$, $g: \mathbb{N} \rightarrow \mathbb{N}^+$ si ha

$$\cdot f(m) = \omega(g(m)) \Leftrightarrow \lim_{m \rightarrow +\infty} \frac{f(m)}{g(m)} = +\infty \quad F \text{ più veloce di } g$$

ALTRO USO DELLE NOTAZIONI ASINTOTICHE

$$\cdot h(m) = k(m) + \Theta(g(m))$$

$\Theta(g(m))$ insieme di funzioni

Sigifica:

$$\exists f(m) \in \Theta(g(m)) : h(m) = k(m) + f(m)$$

$$\cdot h(m) + \Theta(g(m)) = \Theta(k(m))$$

Sigifica:

$$\forall f(m) \in \Theta(g(m)), h(m) + f(m) = \Theta(k(m))$$

RELAZIONI TRA NOTAZIONI

TRANSITIVITA'

$$f(m) = \Theta(g(m)) \wedge g(m) = \Theta(h(m)) \Rightarrow f(m) = \Theta(h(m))$$

vol per ($\mathcal{R}, O, o, \omega$)

RIFLESSIVITA'

$$f(m) = \Theta(f(m))$$

$$f(m) = O(f(m))$$

$$f(m) = \Omega(f(m))$$

ANTIRIFLESSIVITA'

$$f(m) \neq o(f(m))$$

$$f(m) \neq \omega(f(m))$$

SIMMETRIA

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

SIMMETRIA TRASPOSTA

$$f(m) = \Theta(g(m)) \Leftrightarrow g(m) = \Omega(f(m))$$

$$f(m) = o(g(m)) \Leftrightarrow g(m) = \omega(f(m))$$

ANALOGIA TRA CONFRONTI

Siamo f, g 2 funzioni e $a, b \in \mathbb{R}$

$$f(m) = O(g(m)) \quad \approx \quad a \leq b$$

$$f(m) = \Omega(g(m)) \quad \approx \quad a \geq b$$

$$f(m) = \Theta(g(m)) \quad \approx \quad a = b$$

$$f(m) = o(g(m)) \quad \approx \quad a < b$$

$$f(m) = \omega(g(m)) \quad \approx \quad a > b$$

Tuttavia la proprietà di **tricotomia**, non è sempre valida per il confronto asintotico, ovvero, 2 funzioni non sono sempre confrontabili.

EX:

- $f(m) = m$

- $g(m) = m^{1+\sin m} \Rightarrow [0 \leq \sin m \leq 1] \Rightarrow 0 \leq m^{1+\sin m} \leq 2$

f e g non sono asintoticamente confrontabili



NOTAZIONI STANDARD E FUNZIONI COMUNI

$\lfloor x \rfloor = (\text{massimo intero} \leq x)$ floor

$\lceil x \rceil = (\text{minimo intero} \geq x)$ ceiling

$$\cdot \forall x \in \mathbb{R} \Rightarrow x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil \leq x + 1$$

$$\cdot \forall m \in \mathbb{N} \Rightarrow \lceil m_2 \rceil + \lfloor m_2 \rfloor = m$$

$$\cdot \forall m \in \mathbb{N}, \forall a, b \in \mathbb{N}^+ \Rightarrow \lceil \lceil m_a \rceil / b \rceil = \lceil m_{ab} \rceil$$

$$\lfloor \lceil m_a \rceil / b \rfloor = \lfloor m_{ab} \rfloor$$

$f(m)$ si dice polimomiale limitata se $f(m) = \Theta(m^k)$, per qualche $k \geq 0$

ESPOENZIALI

Poiché: $\lim_{m \rightarrow +\infty} \frac{m^b}{a^m} = 0$, $\forall b, \forall k > 1 \Rightarrow m^b = o(a^m)$

$$\cdot e^x = 1 + x + \Theta(x^2)$$

LOGARITMI

$f(m)$ si dice pol-logaritmicamente limitata se $f(m) = \Theta(\log_k^k m)$ per qualche k

$$\cdot \log_b a = \frac{\log_c a}{\log_c b}$$

$$\cdot \log_b a = \frac{1}{\log_a b}$$

$$\cdot a^{\log_b c} = c^{\log_b a}$$

FATTORIALI

$$m! = \Theta(m^n)$$

$$m! = \Theta(z^m)$$

$$\log(m!) = \Theta(m \log m)$$

SOMMATORIE

$$\sum_{i=1}^m i = \frac{m(m+1)}{2} = \Theta(m^2)$$

$$\sum_{i=1}^m i^2 = \frac{m(m+1)(2m+1)}{6} = \Theta(m^3)$$

$$\sum_{i=1}^m i^3 = \frac{m^2(m+1)^2}{4} = \Theta(m^4)$$

$$\sum_{i=0}^m x^i = \frac{x^{m+1} - 1}{x - 1}$$

$$H_m = \sum_{i=1}^m \frac{1}{i} = \ln m + O(1)$$

RICORRENZE

Sono delle equazioni o diseguazioni che descrivono il valore di una funzione in termini del suo valore con input più piccoli.

Ex: MERGESORT

$$T(m) = \begin{cases} \Theta(1) & \text{se } m=1 \\ 2T(\frac{m}{2}) + \Theta(m) & \text{se } m>1 \end{cases} \Rightarrow T(m) = \Theta(m \log m)$$

Considereremo i seguenti 3 metodi:

- METODO DI SOSTITUZIONE
- METODO ITERATIVO O DELL'ALBERO DI RICORSIONE
- METODO MASTER PER RICORRENZE DELLA FORMA

$$T(m) = aT(\frac{m}{b}) + f(m) \quad a \geq 1, b > 1$$

METODO DI SOSTITUZIONE ①

Non è un metodo risolutivo ma di verifica, la soluzione deve essere quindi già fornita o intuita, esso si compone di due parti:

1. individuare una possibile soluzione
2. verificare la soluzione per induzione

Ex: Determinare un limite superiore per $T(m)$, ove $T(m) = 2T(\lfloor \frac{m}{2} \rfloor) + m$, cioè

$$T(m) \leq cm \log m \text{ per qualche } c > 0 \text{ e } m \text{ sufficientemente grande}$$

Supponiamo che:

$$T(\lfloor \frac{m}{2} \rfloor) \leq c \lfloor \frac{m}{2} \rfloor \log \lfloor \frac{m}{2} \rfloor$$

- In questo momento non stiamo applicando una vera induzione, stiamo effettuando un tentativo che ci dà delle condizioni su c . Se queste condizioni sono invertibili, potremmo dire che $\exists c > 0$ tali che la presupposizione da verificare risulta vera.
- Se invece questo approccio ci porta ad una conclusione in cui non esistono valori per quella condizione, vorrà dire che la nostra presupposizione era sbagliata.

Il nostro obiettivo è quindi dimostrare che esistono c e m_0 , inoltre questo approccio ci dà l'ambito di variabili, ovvero, che, basta prendere determinati valori di c ed m_0 per dimostrare la condizione.

Per dimostrare quindi che $t(m) \leq cm \log m$ sia vera, dimostriamola per induzione dando per vera $L^{\lceil m/2 \rceil}$.

Quindi:

Supponiamo:

$$t(L^{\lceil m/2 \rceil}) = c L^{\lceil m/2 \rceil} \log L^{\lceil m/2 \rceil} \quad \text{vera}$$

Speriamo di dimostrarlo per delle condizioni su c e m_0 , se queste condizioni sono non contraddittorie avremo una stima delle condizioni per soddisfare la ricchezza.

Ricordiamo di avere la ricchezza

$$t(m) = 2t(L^{\lceil m/2 \rceil}) + m$$

Al posto di $t(L^{\lceil m/2 \rceil})$, sfruttando l'ipotesi induttiva, andremo a sostituire *, ovvero, $c L^{\lceil m/2 \rceil} \log L^{\lceil m/2 \rceil}$.

Otteneremo:

$$t(m) \leq 2c L^{\lceil m/2 \rceil} \log(L^{\lceil m/2 \rceil}) + m$$

Maggiorando eliminiamo i floor, poiché $\lceil m/2 \rceil \leq m/2$:

$$\leq cm \log\left(\frac{m}{2}\right) + m$$

$$= cm \log m - cm \log 2 + m \quad // \log \frac{m}{2} = \log m - \log 2 \text{ e distribuisce}$$

$$= cm \log m - cm + m$$

$$// \text{poiché } \log 2 = 1$$

$$\leq cm \log m$$

// trascuriamo $-cm$

Alla fine abbiamo trovato $t(m) \leq cm \log m$.

Non abbiamo però finito poiché serve anche il **CASO BASE**

Il nostro obiettivo è quindi dimostrare che esistono c e m , inoltre questo approccio ci dà l'ambito di variabili; ovvero, che, basta prendere determinati valori di c ed m per dimostrare la condizione.

- Per dimostrare quindi che $t(m) \leq cm \log m$ sia vera, dimostriamola per induzione dando per vera $L^{\lceil m/2 \rceil}$.
Quindi:

Supponiamo:

$$t(\lfloor m/2 \rfloor) = c \lfloor m/2 \rfloor \log \lfloor m/2 \rfloor \quad \text{vera}$$

Speriamo di dimostrarlo per delle condizioni su c e m , se queste condizioni sono non contraddittorie avremo una stima delle condizioni per soddisfare la ricchezza.

Ricordiamo di avere la ricchezza

$$t(m) = 2t(\lfloor m/2 \rfloor) + m$$

Al posto di $t(\lfloor m/2 \rfloor)$, sfruttando l'ipotesi induttiva, andremo a sostituire *, ovvero, $c \lfloor m/2 \rfloor \log \lfloor m/2 \rfloor$.

Otteneremo:

$$t(m) \leq 2c \lfloor m/2 \rfloor \log(\lfloor m/2 \rfloor) + m$$

Maggiorando eliminiamo i floor, poiché $\lfloor m/2 \rfloor \leq m/2$:

$$\leq cm \log\left(\frac{m}{2}\right) + m$$

maggiorante

$$= cm \log m - cm \log 2 + m \quad // \log \frac{m}{2} = \log m - \log 2 \text{ e distribuz.}$$

$$= cm \log m - cm + m \quad // \text{poiché } \log 2 = 1$$

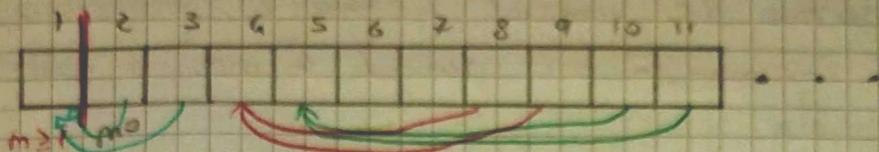
$$\leq cm \log m$$

// trascuriamo $-cm$

Alla fine abbiamo trovato $t(m) \leq cm \log m$.

Non abbiamo però finito poiché serve anche il **CASO BASE**

CASO BASE



Ponendendo in considerazione i vari casi (in questo caso 11), possiamo vedere l'applicazione di $T(L^{\lfloor m/2 \rfloor})$

- il caso 11, sfruttano $T(L^{\lfloor 11/2 \rfloor}) = 5$, stessa cosa il 10, il 9 e cos' via.

• Proviamo a vedere se $T(1)$ ha senso:

$$T(1) \leq C \cdot \underbrace{1 \log 1}_0 = 0 \quad \text{Non ha senso}$$

Quindi questa relazione non vale per $m=1$, questo non è però un problema poiché stiamo ragionando per m sufficientemente grande, quindi, come prima condizione, $m \geq 1$.

- Esaminando il 3, esso si sarebbe basato su $T(L^{3/2})$, ovvero 1, così anche il 2, $T(L^{2/2}) = 1$.

Sappiamo quindi che il 3 e il 2 non possono sfruttare la proprietà poiché l'ipotesi induktiva non vale per 1.

Dobbiamo partire quindi da un numero maggiore di 1, e i primi valori utili sono 2 e 3.

Quindi 2 e 3 costituiranno i casi base.

Dobbiamo quindi considerare la nostra relazione $T(m)$ per 2 e 3 casi base:

$$\begin{aligned} \bullet T(2) &= C \cdot 2 \log 2 = 2C \Rightarrow C \geq \frac{T(2)}{2} \\ \bullet T(3) &= C \cdot 3 \log 3 = C \geq \frac{T(3)}{3 \log 3} \end{aligned}$$

$$T(2) \leq 2C$$

Inoltre abbiamo verificato che il primo valore utile per applicare l'induzione è 2, quindi, $m_0 = 2$

• Abbiamo trovato dei vincoli compatibili.



Ex:
Se avessimo trovato dei vincoli come $C > S$ e $C < 4$, essi sarebbero stati incompatibili, quindi la nostra ipotesi sarebbe stata anata.

Per concludere:

Abbiamo dimostrato la nostra ipotesi induttiva con le seguenti condizioni:

- $m \geq 2$
- $C \geq \max\left(\frac{T(2)}{2}; \frac{T(3)}{3}\right)$

Fine RAFFORZAMENTO DELL'IPOTESI INDUTTIVA

In certi casi quest'approccio di sostituzione non funziona

Ex: Si dimostri che la \sim è comune a

$$T(m) = T(\lfloor L^{m_2} \rfloor) + T(\lceil R^{m_2} \rceil) + 1$$

ha soluzione

$$T(m) = O(m)$$

O come verificare quindi che:

$$T(m) \leq cm \quad \text{per qualche } c > 0 \quad \text{e } \forall m > m_0$$

Supponiamo induttivamente che

$$T(\lfloor L^{m_2} \rfloor) \leq c \lfloor L^{m_2} \rfloor, \quad T(\lceil R^{m_2} \rceil) \leq c \lceil R^{m_2} \rceil$$

Allora

$$T(m) \leq c \overbrace{\lfloor L^{m_2} \rfloor}^{cm} + c \lceil R^{m_2} \rceil + 1 = cm + 1 \quad \cancel{\Rightarrow} \quad T(m) \leq cm$$

Non ci siamo riusciti.

Questo non vuol dire che la nostra ipotesi sia sbagliata, perché la relazione $cm = O(m)$ non è un se e solo se, possiamo quindi rafforzare l'ipotesi aggiungendo quella che abbiamo trascurato all'inizio, in questo caso una semplice costante.

Rafforziamo l'ipotesi induktiva con $T(m) \leq cm - b$, $b \geq 0$

Avremo

$$\begin{aligned} T(m) &\leq [c\lfloor \frac{m}{2} \rfloor - b] + c(\lceil \frac{m}{2} \rceil - b) + 1 = \\ &= cm - cb + 1 \leq cm - b \quad // Lo abbiamo imposto \\ &\quad \boxed{\lfloor \frac{m}{2} \rfloor \rightarrow b \geq 1} \end{aligned}$$

Per $b \geq 1$, $c \geq T(1) + b$, $m \geq 1$

per c. B m=1

$$T(1) \leq c - b \Rightarrow c \geq T(1) + b$$

N.B. Abbiamo trovato delle condizioni necessarie e sufficienti poiché ora possiamo rileggere la dimostrazione al contrario arrivando a condizioni esatte.

SITUAZIONE DI POSSIBILE ERRORE

Data: $T(m) = 2T(\lfloor \frac{m}{2} \rfloor) + m$, cerchiamo di dimostrare che:

$$T(m) \leq O(m), \text{ ovvero } T(m) \leq cm \text{ per qualche } c > 0 \text{ e } \forall m \geq m_0$$

- Supponiamo per induzione che:

$$T(\lfloor \frac{m}{2} \rfloor) \leq c \lfloor \frac{m}{2} \rfloor$$

Allora

$$T(m) \leq 2c \lfloor \frac{m}{2} \rfloor + m \leq cm + m = (c+1)m \underset{\substack{\downarrow \\ \text{ERRORE}}}{=} O(m)$$

È un errore poiché, dobbiamo dimostrare che $T(m) \leq cm$, con la medesima costante c fissata

- Imponendo quell'uguaglianza, stiamo dicendo che la costante si è adottata, il che è un assurdo.

- In questo caso non si può rafforzare l'ipotesi poiché sappiamo che $T(m) = \Theta(m \log m) \neq O(m)$.

METODO ITERATIVO

Consiste nell'applicare iterativamente l'equazione di ricorrenza sino ad esprimere la funzione in termini di m e delle condizioni iniziali.

Ese:

$$T(m) = 3T(\lfloor \frac{m}{4} \rfloor) + m$$

Lo scopo è sostituire l'eq fino ad ottenere $f(x) = m$

$$\begin{aligned} T(m) &= m + 3T(\lfloor \frac{m}{4} \rfloor) \quad // \text{invertendo i termini per comodità} \\ &= m + 3\left(\lfloor \frac{m}{4} \rfloor + 3T(\lfloor \frac{m}{4^2} \rfloor)\right) \\ &= m + 3\left(\lfloor \frac{m}{4} \rfloor + 3\left(\lfloor \frac{m}{4^2} \rfloor + 3T(\lfloor \frac{m}{4^3} \rfloor)\right)\right) \\ &= m + 3\left\lfloor \frac{m}{4} \right\rfloor + 3^2\left\lfloor \frac{m}{4^2} \right\rfloor + 3^3T\left(\left\lfloor \frac{m}{4^3} \right\rfloor\right) \\ &\vdots \\ &= m + 3\left\lfloor \frac{m}{4} \right\rfloor + 3^2\left\lfloor \frac{m}{4^2} \right\rfloor + \dots + 3^{\lfloor \log_4 m \rfloor - 1} \cdot \left\lfloor \frac{m}{4^{\lfloor \log_4 m \rfloor - 1}} \right\rfloor \\ &\quad + 3^{\lfloor \log_4 m \rfloor} \cdot f\left(\left\lfloor \frac{m}{4^{\lfloor \log_4 m \rfloor}} \right\rfloor\right) \end{aligned}$$

Arrivato questo punto non possiamo più applicare l'equazione di ricorrenza, poiché $T\left(\left\lfloor \frac{m}{4^{\lfloor \log_4 m \rfloor}} \right\rfloor\right)$ è una quantità minore di a , quindi, questo numero uscirebbe dal range.

Questo numero sarà una costante.

Maggiorando i floor e scrivendo

$$\leq m + 3\frac{m}{4} + \left(\frac{3}{4}\right)^2 m + \dots + \left(\frac{3}{4}\right)^{\lfloor \log_4 m \rfloor - 1} m + 3^{\log_4 m}$$

continua

$$a m 2^{b+1} < m \cdot \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i + \Theta(m \log_4 3) \quad // 3^{\log_4 m} = m^{\log_4 3}$$

$$= 4m + \Theta(m \log_4 3)$$

$$= \Theta(m)$$

$$\log_4 3 < 1$$

$$\sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i = \frac{1}{1-\frac{3}{4}} = \frac{1}{\frac{1}{4}} = 4$$

Alla fine avremo $\Theta(m)$ poiché $4m$ domma su $m \log_4 3$.

ALBERI DI RICORSIONE

Sono particolarmente utili nel metodo iterativo.

Ex: $T(m) = 2T(\frac{m}{2}) + m^2$

$$T(m)$$

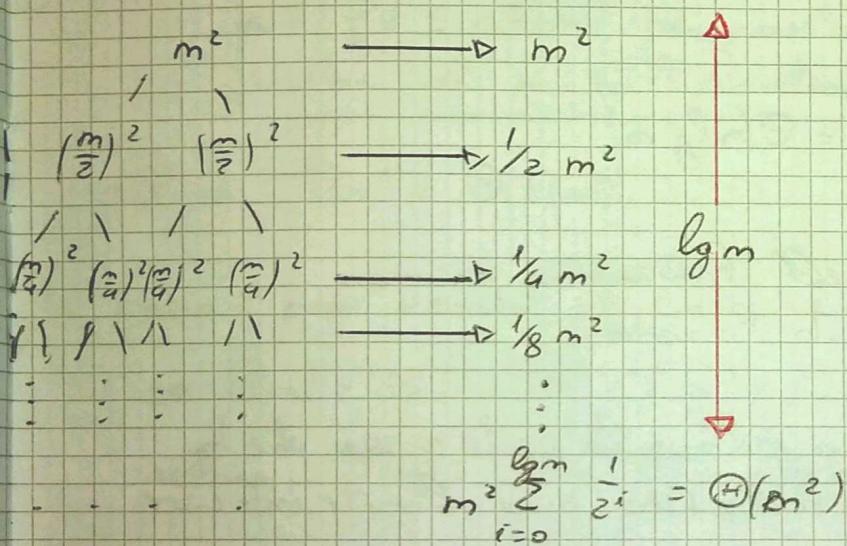
$$\begin{array}{c} m^2 \\ / \quad \backslash \\ T\left(\frac{m}{2}\right) \quad T\left(\frac{m}{2}\right) \end{array}$$

considerando

$$\frac{T(k)}{k^2}$$

$$\begin{array}{c} / \quad \backslash \\ T\left(\frac{k}{2}\right) \quad T\left(\frac{k}{2}\right) \end{array}$$

$$\begin{array}{c} m^2 \\ / \quad \backslash \\ \left(\frac{m}{2}\right)^2 \quad \left(\frac{m}{2}\right)^2 \\ / \quad \backslash \quad / \quad \backslash \\ T\left(\frac{m}{4}\right) \quad T\left(\frac{m}{4}\right) \quad T\left(\frac{m}{4}\right) \quad T\left(\frac{m}{4}\right) \end{array}$$



Sappiamo che la somma di ogni livello diminuisce di metà ad ogni livello, quindi, la somma di tutti i livelli

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

\downarrow
maggiora

ALBERI DI RICORSIONI SU TERMINI MISTI!

Ex:

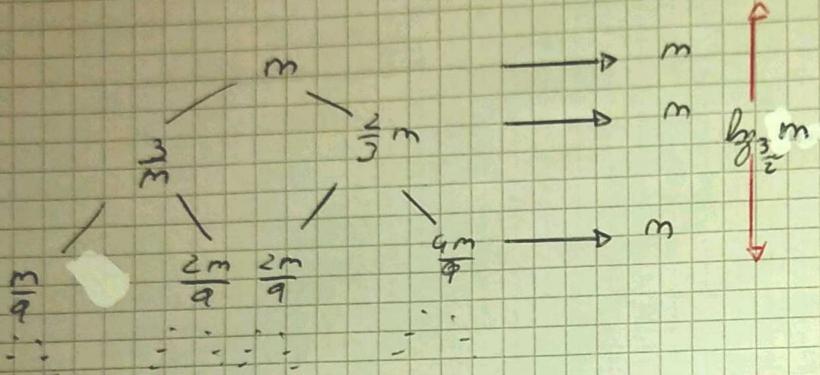
$$T(m) = T\left(\frac{m}{3}\right) + T\left(\frac{2m}{3}\right) + m$$

$$T(m)$$

$$\begin{array}{c} m \\ / \quad \backslash \\ T\left(\frac{m}{3}\right) \quad T\left(\frac{2m}{3}\right) \end{array}$$

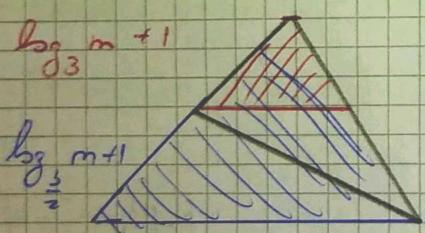
$$\begin{array}{c} m \\ / \quad \backslash \\ \frac{m}{3} \quad \frac{2m}{3} \\ / \quad \backslash \quad \backslash \\ T\left(\frac{m}{9}\right) \quad T\left(\frac{2m}{9}\right) \quad T\left(\frac{4m}{9}\right) \end{array}$$

$$T\left(\frac{m}{q}\right) \quad T\left(\frac{2m}{q}\right) \quad T\left(\frac{4m}{q}\right)$$



$$m \log_{\frac{3}{2}} m = \Theta(m \lg m)$$

Esaminando l'albero, da l'ult. fin. ramo prima,
l'albero avrà perciò questa forma



- misuriamo prima la parte rossa che ha $\log_3 m + 1$ livelli, essa, sarà limite inferiore
- È difficile contare solo una parte dell'albero, dunque maggioriamo e troviamo un limite superiore

Avevamo

$$m (\log_3 m) \leq T(m) \leq m (\log_{\frac{3}{2}} m)$$

"

"

0

+

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

TEOREMA MASTER

Per ricorrenze della forma $T(m) = aT(\frac{m}{b}) + f(m)$

Siamo $a \geq 1$, $b > 1$ costanti e sia $f(m)$ una funzione assegnata

Sia inoltre $\tilde{T}(m)$ tale che $\tilde{T}(m) = a\tilde{T}(\frac{m}{b}) + f(m)$

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

allora: $\tilde{T}(m) = \Theta(m^{\log_b a})$

2. Se $f(m) = \Theta(m^{\log_b a})$,

allora: $\tilde{T}(m) = \Theta(m^{\log_b a} \cdot \log m)$

3. Se $f(m) = \Omega(m^{\log_b a + \varepsilon})$ per qualche $\varepsilon > 0$

e se:

$a\tilde{T}(\frac{m}{b}) \leq c f(m)$ per qualche $c < 1$ e per valori di m sufficientemente grandi

CONDIZIONE
DI REGOLARITÀ

allora: $\tilde{T}(m) = \Theta(f(m))$

Il caso 2 può essere generalizzato

2': se $f(m) = \Theta(m^{\log_b a} \cdot \log^k m)$, con $k \geq 0$.

allora: $\tilde{T}(m) = \Theta(m^{\log_b a} \cdot \log^{k+1} m)$

EX:

$$0. \quad T(m) = 9T\left(\frac{m}{3}\right) + m^{\log_3 9} \quad a=9, b=3, m^{\log_b a} = m^2 \quad m \in O(m^{2-\varepsilon}) \quad 0 < \varepsilon \leq 1$$

$$-f(m) = m = O(m^{\log_3 9 - \varepsilon}) \quad \forall \varepsilon \leq 1 \Rightarrow T(m) = \Theta(m^2)$$

$$1. \quad T(m) = T\left(\frac{m}{3}\right) + 1 \quad a=1, b=\frac{3}{2}, m^{\log_{\frac{3}{2}} 1} = m^0 \quad 1 = \Theta(m^0)$$

$$-f(m) = 1 = \Theta(m^{\log_{\frac{3}{2}} 1}) \Rightarrow T(m) = \Theta(\log m)$$

$$2. \quad T(m) = 3T\left(\frac{m}{9}\right) + m \log m \quad a=3, b=9, m^{\log_9 3}$$

$$-f(m) = m \log m = \Omega(m^{\log_9 3 - \varepsilon}) \quad \forall \varepsilon \leq 1 \leq \log_9 3$$

$$\text{INDIETRE: } 3 \frac{3}{4} \log \frac{m}{4} \leq \frac{3}{4} m \log m \quad (c=3a) \\ \Rightarrow T(m) = \Theta(m \log m)$$

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

$$a=2, b=2 \quad m^{\log_2^2} = m'$$

$$f(m) = m \log m = \Theta(m \cdot \log m) \Rightarrow T(m) = \Theta(m \log^2 m)$$

• Quanto detto dal teorema, i m ciascuno dei 3 casi vengono confrontate tra di loro $f(m)$ e $m \log^k m$.

Informalmente il teorema master ci dice che vince il maggiore tra i 2.

- Se il maggiore è $m^{\log_b a}$ siamo nel **Caso 1**
- Se sono uguali siamo nel **Caso 2**
- Se è minore siamo nel **Caso 3**

COSTRUZIONE DELL'HEAPPOLL / HEAPSORT

LEZ 6
FARO

CODIFICA

BUILD-HEAP(A, m)

heap-size = 1

For $i = 2$ To m

 insert(A, A[i]) // inserisce l'elemento $A[i]$ che
 // poi risalendo per ordinarsi nell'heap

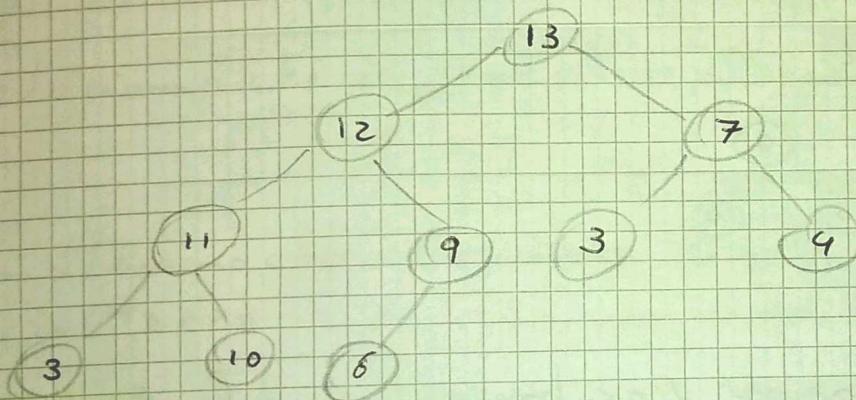
EX: Fase iniziale

	0	1	2	3	4	5	6	7	8	9	10
A	X	12	13	6	11	6	3	7	5	0	9

(12)

Passo per passo gli elementi verranno inseriti nell'heap
i dopo i, ordinandosi, fino a quando si sono creato l'heap

fase finale : Albero creato



COMPLESSITÀ

Il ciclo For viene eseguito m volte e ogni volta
effettua un insert che ha complessità $\log m$,
quindi la complessità totale sono $O(m \cdot \log m)$

COSTRUZIONE DELL'HEAP MIGLIORATA (FORD, 1969)

Nello stesso anno venne sviluppato un algoritmo per la costruzione di heap che sfrutta heapify.

CODIFICA

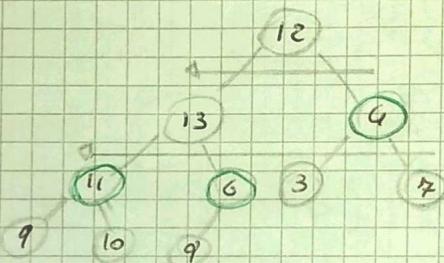
BUILD-HEAP(A, m)

heapsize = m

for $i = \lfloor \frac{m}{2} \rfloor$ down to 1 // $O(m)$
 heapify(A, i) // $O(m \log m)$

Sappiamo che heapify può essere chiamato soltanto su nodi i cui figli sono già heap. Per definizione, le foglie sono degli heap in quanto già ordinate poiché formate da un solo elemento.

Consideriamo il seguente albero di partenza



Dalle precedenti considerazioni, ~~nessuno~~ abbiamo come figli degli heap, posso quindi chiamare heapify sui loro. Una volta ordinati, posso considerarli heap e quindi chiamare heapify sul loro padre e così via a sorpresa fino a chiamarlo sulla root e ordinare tutto l'heap.

Sappiamo che nell'array, il nodo i , è in posizione $\lfloor \frac{m}{2} \rfloor$, avendo esso già come figli ϵ heap, poiché foglie, inizio ad applicare heapify da lì fino ad arrivare a posizione 1, e non

COMPLESSITÀ

Apparentemente questa versione del BUCO-HEAP non ha portato nessun vantaggio poiché $O(m \log m)$.

Questa complessità è corretta ma è sempre un limite superiore, potrebbe infatti esistere una funzione minore di $m \log m$ ma comunque corretta per indicare la complessità della procedura.

OSS Supponiamo di avere un heap di m modi, esso avrà:

- $h = \log m$, poiché completo
- $\lceil \frac{m}{2} \rceil$ foglie, (modi ad $h=0$)
- $\lceil \frac{m}{2^1} \rceil$ modi ad $h=1$ (padri delle foglie)
- $\lceil \frac{m}{2^2} \rceil$ modi ad $h=2$ (nonni delle foglie)
- ⋮
- $\lceil \frac{m}{2^{h-1}} \rceil = 1$ modi ad altezza $\log m$ (root)

Deduciamo che in generale, ad altezza h avremo al più

$$\lceil \frac{m}{2^{h+1}} \rceil \text{ modi}$$

Consideriamo che il costo di heapify sulla root di un albero di altezza h è $O(h)$ (con m nodi $\log m$)

Avremo:

$$T(m) = \sum_{h=0}^{\log m} \lceil \frac{m}{2^{h+1}} \rceil + O(h)$$

Maggiorando vediamo che $\lceil \frac{m}{2^{h+1}} \rceil \leq \frac{m}{2^h}$

Osservando la nostra equazione notiamo che essa è simile a una serie notevole la cui soluzione è nota

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{1-x^2}, \text{ per } |x| < 1 \text{ converge}$$

$$\Rightarrow O\left(m \sum_{h=0}^{\log m} \frac{1}{2^h}\right) \leq \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2} = 2.$$

La complessità è $m \log m \log m$ poiché heapify non è applicato a tutti i modi ma solo ad alcuni

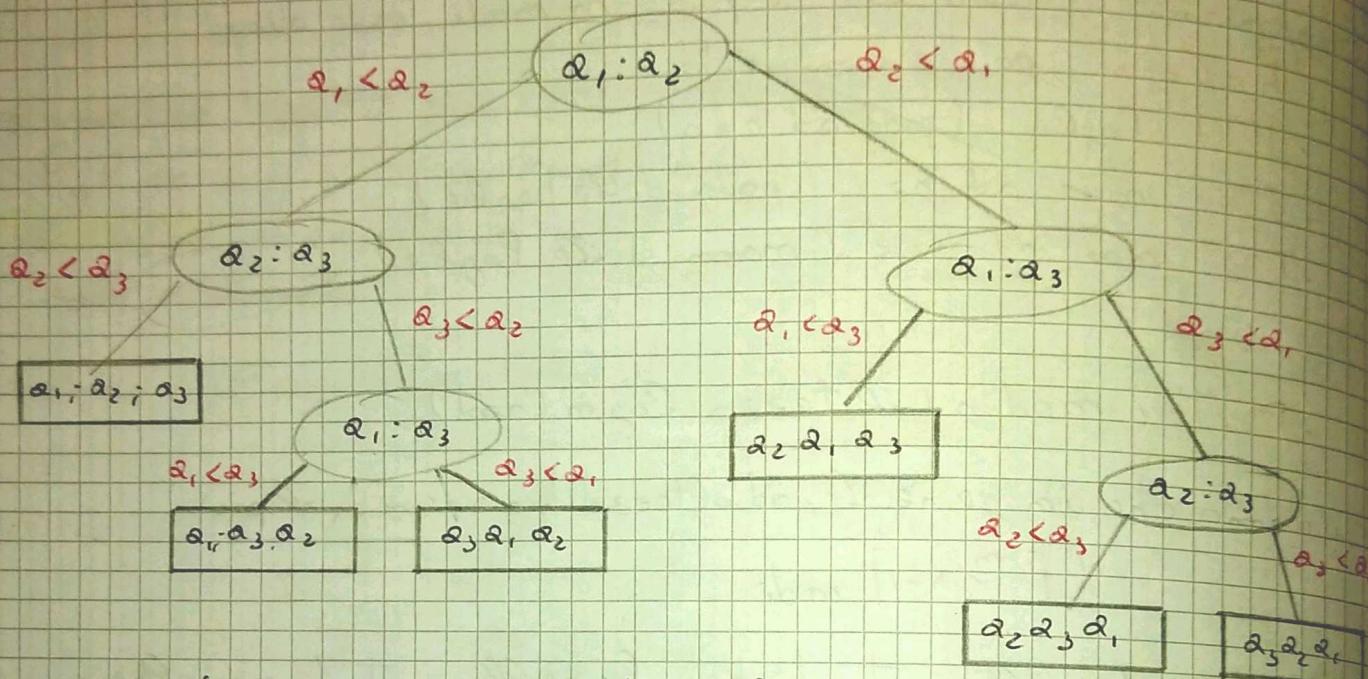
considerando m che era stato trascurato:

$$T(m) = O(zm) = O(m) \Rightarrow \text{la complessità è minore della 1^a versione}$$

ALBERO DI DECISIONE

Sappiamo che non è possibile ottenere algoritmi basati su confronti con complessità inferiore a $m \log m$, questo è dimostrabile grazie all'**albero decisionale**

Supponiamo di avere un array da ordinare $\langle a_1, a_2, a_3 \rangle$ e che l'algoritmo non conosca i valori.
Per ordinarli deve confrontarli tra di loro



N.B. I: indicamo l'operazione di confronto
i rettangoli gli ordinamenti trovati;

- Nell'albero creato ci sono esattamente $m!$ foglie
- e l'altezza massima è $m \log m$ (percorso più lungo)
- esistono però percorsi che hanno molti meno elementi.
- Algoritmi diversi prendono percorsi diversi in situazioni diverse.
- Esistono algoritmi non basati su confronti che hanno complessità inferiore a $m \log m$

COUNTING SORT

Il counting sort è un algoritmo di ordinamento basato sulla tecnica del conteggio, sviluppato da Sedgwick nel 1954.

SPIEGAZIONE

- Dato un array A da ordinare, ricerciamoci in esso il numero maggiore K .
- Una volta ottenuto K , alloco un array C di dimensione $K+1$, i cui indici andranno da 0 a K .
- Scomendo da sinistra verso destra A, esamino ogni valore e scrivo in $C[A[i]]$ il numero di volte che un valore compare nell'array A. Quindi, ad esempio, se in A c'è presente n volte il valore x , scriviamo che alla posizione $C[x]$ è memorizzato il numero n . (conta quindi le occorrenze).
- Dopo la fine di questo passaggio, noteremo che:
 $|A| = m$ e che $\sum_{i=0}^K C[i] = m$, quindi che la somma dei valori in ogni cella di C è esattamente m .
- Infine, una volta riempito C, alloco un array B di dimensione m quindi quanto A.
 Dentro B inseriamo in modo distribuito tante volte quante riportate in $C[i]$, l'indice i , nell'array B.
 Se ad esempio per $i = x$ e $C[x] = m$, inseriamo in B m volte il valore x .
- Una volta riempito B, avremo ottenuto l'array ordinato B.

Ex:

A	5 2 3 6 1 2 2 4 3 3 1	$K = 6$	$NO(m)$
C	0 1 2 3 4 5 6	$NO(K)$	
B	1 1 2 2 2 3 3 3 4 5 6	$NO(m)$	// 0 volte, 1 volta, 2 volte,

COMPLESSITÀ

L'algoritmo ha complessità $O(m + K)$, dove $O(m)$ è il costo per scandire A e $O(K)$ il costo per scandire C. Tutto dipende da K , se K è costante o è minore di m , la complessità sarà $O(m)$, se invece K non è definito, la complessità sarà dominata da K . Inoltre l'algoritmo non è implice.

CODIFICA

COUNTING-SORT(A, m)

K = max(A)

// ricerca il max

C = new array (K+1)

// alloca l'array C

For (i=0 to k)

{ // inizializza C

C[i] = 0

For (i=0 to m-1)

// scava A

C[A[i]] = C[A[i]] + 1 // incrementa il contenuto di C[A[i]]

B = new array (m)

// alloca B

J = 0

// alloca indice per B

For (i=0 to K+1)

// per ogni elemento di C

For (h=1 to C[i])

// ripete C[i] volte

B[J] = i

// riempie B

J++

1° OTTIMIZZAZIONE DEL COUNTING SORT

Possiamo ragionare sul fatto che, pur avendo tantissimi elementi essi non siano uniformemente distribuiti, ad esempio potrebbero essere distribuiti tutti sulla parte finale.

Ex:

A [52 51 50 52 53 51 50 69]

Considerando l'implementazione classica, venrebbe allocato un array C con indici da 0 a 56, allocando tantissimo spazio inutilmente.

Potremmo allocare un array C che va da $\min(A)$ a $\max(A)$ per ridurre lo spazio utilizzato e aumentare la performance dell'algoritmo.

N.B. Quando inserisco gli elementi da A in C, per trovare il giusto indice in C mi basterà fare $A[i] - m$

CODIFICA

COUNTING-SORT(A, m)

$$K = \max(A)$$

$$m = \min(A)$$

C = new array ($K - m + 1$)

For ($i = 0$ to $(K - m)$)

$$C[i] = 0$$

For ($i = 0$ to $m - 1$)

$$C[A[i] - m] = C[A[i] - m] ++$$

B = new array(m)

$$j = 0$$

For ($i = 0$ to $K - m$)

For ($h = 1$ to $C[i]$)

$B[j] = i + m$ // siccome ho diminuito l'indice, devo
aumentarlo per portarlo al
valore originale

2° OTTIMIZZAZIONE DEL COUNTING SORT

Esiste però un altro problema, non riusciamo a determinare la stabilità del counting sort in quanto, i valori ordinati in B, non sono quelli originali presenti in A ma solo delle copie.

Questo dà vita ad un altro problema, ovvero, il counting sort non permette di ordinare elementi che hanno stessa informazione satellite, in quanto non tenendo copiate andrebbero perse.

STRATEGIA RISOLUTIVA

A	8	9	6	10	7	8	8	10	10	12	10
---	---	---	---	----	---	---	---	----	----	----	----

$$K = 12 \quad m = 6$$

6-6	7-6	8-6	9-6	10-6		
0	1	2	3	4	5	5

C	1	1	3	1	6	0	1
---	---	---	---	---	---	---	---

$$|C| = K - m + 1 = 7$$

$C[i]$ = numero di elementi in A uguali ad i

Vorrei ottenere:

$C'[i]$ = numero di elementi in A \leq ad i

C'	1	2	3	4	5	6
	6-6					

in A ci sono "el minori di 12" ($12-6$)

in A ci sono gli el ≤ 6 , in 1, 2, 3, ..., poiché devo fare $A[i] - m$

OSS

Da C posso ottenere C' in una passata poiché

$$C'[i] = C[i-1] + C[i]$$

EX:

	0	1	2
C	3	2	1
	5	16	2
C'	3	5	6
	↑	↑	↑
	(0+3)	(3+2)	(5+1)

Una volta costruito C' posso procedere

EX:

Guardando A e C' precedenti, scelgo l'elemento 8 in A.

Vado a guardare in $C'[8-6]$, quanti elementi sono presenti prima di 8 e $C'[2]=5$.

Visto che ci sono 5 elementi minori uguali a 8, nell'array finale B, 8 sarà in posizione $C[2]-1$, quindi 4.

- Quindi una volta costruito C', posso anche inserire gli elementi in B in ordine sparso.

NOTA: Quando ho più elementi uguali, devo ovviamente accortarmi di diminuire il numero di elementi minori uguali di $C'[i]$. così da poter inserire anche gli altri elementi uguali.

Inoltre, partendo a inserire gli elementi da destra verso sinistra, questo algoritmo diventa stabile (elementi uguali da destra verso sinistra)

Quindi (Dati A e C' della pagina precedente)

B	0	1	2	3	4	5	6	7	8	9	10
	6	7	8	8	8	9	10	10	10	10	10