Concetti di base di complessità degli algoritmi
+
Algoritmi di ordinamento

Problemi, algoritmi, programmi

- Problema: il compito da svolgere
  - quali *output* vogliamo ottenere a fronte di certi *input*
  - cioè quale *funzione* vogliamo realizzare
- Algoritmo: i passi (il processo) da seguire per risolvere un problema
  - un algoritmo prende gli *input* in ingresso ad un problema e li trasforma in opportuni *output*
- Come al solito, un problema può essere risolto da tanti algoritmi

- Un algoritmo è una sequenza di **operazioni concrete**
  - deve essere *eseguibile* da una "macchina"
- Un algoritmo deve essere **corretto**
  - deve calcolare la funzione giusta
  - sappiamo che determinare la correttezza di un algoritmo è un problema indecidibile...
  - ... questo però non vuole dire che non si possa fare niente per cercare di capire se un algoritmo è corretto o no

- Un algoritmo può essere descritto in diversi linguaggi
  - se usiamo un linguaggio di programmazione (C, C++, Java, C#, ecc.) abbiamo un *programma*

- Come linguaggio noi usiamo lo **pseudocodice**
  - non è un vero linguaggio di programmazione, ma ci assomiglia molto
  - facile da tradurre in codice di un linguaggio di programmazione quale C, Java, o Python
  - il particolare linguaggio di programmazione con cui un algoritmo è implementato è, dal punto di vista della complessità, un po' come l'hardware: cambia solo le costanti moltiplicative

Primo esempio di problema/algoritmo

- Problema: **ordinamento**
  - *Input*: una sequenza A di n numeri $[a_1, a_2, ... a_n]$
  - *Output*: una permutazione $[b_1, b_2, ... b_n]$ della sequenza di input tale che $b_1 \leq b_2 \leq ... \leq b_n$
- Algoritmo: *insertion sort*

```
INSERTION-SORT(A)
for j := 2 to A.length
  key := A[j]
  //Inserisce A[j] nella sequenza ordinata A[1..j−1]
  i := j − 1
  while i > 0 and A[i] > key
    A[i + 1] := A[i]
    i := i − 1
  A[i + 1] := key
```

pseudocodice

- assegnamento: `i := j`
  - assegnamento multiplo: `i := j := e`
    - applicato da destra a sinistra
    - cioè è la stessa cosa che scrivere `j := e; i := j`

- **`while`**, **`for`**, **`if-then-else`** come in C

- `//` inizia un commento, che termina alla fine della riga

- la struttura a blocchi è data dalla indentazione

```
while i > 0 and A[i] > key
  A[i + 1] := A[i]
  i := i − 1
A[i + 1] := key
```

=

```
while (i > 0 and A[i] > key)
{
  A[i + 1] := A[i]
  i := i − 1
}
A[i + 1] := key
```

- Le variabili sono locali alla procedura

- Agli elementi degli array si accede come in C
  - `A[j]` è l'elemento di indice *j* dell'array *A*
  - il primo elemento può avere un indice diverso da 0

- Sottoarray:
  - `A[i..j]` è il sottoarray che inizia dall'elemento *i*-esimo e termina all'elemento *j*-esimo
    - e.g. `A[1..5]` è il sottoarray con i primi 5 elementi dell'array *A*

- Dati composti sono organizzati in *oggetti*
- Gli oggetti hanno degli *attributi* (detti anche *campi*)
  - per indicare il valore di un attributo *attr* di un oggetto *x*, scriviamo `x.attr`
  - gli array rappresentano dati composti, quindi sono oggetti
  - ogni array ha un attributo *length*, che contiene la lunghezza dell'array

- Una variabile che corrisponde ad un oggetto (es. un array) è un *puntatore* all'oggetto
  - molto simile ai puntatori in C e, sopratutto, al concetto di *reference* in Java
  - per esempio, se abbiamo due variabili *x* and *y*, e *x* punta ad un oggetto con un attributo *f*, dopo le seguenti istruzioni

    ```
    y := x
    x.f := 3
    ```

    si ha che $x.f = y.f = 3$, in quanto, grazie all'assegnamento `y := x`, *x* e *y* puntano allo stesso oggetto

- Un puntatore che non fa riferimento ad alcun oggetto ha valore *NIL*

- I parametri sono passati *per valore*
  - la procedura invocata riceve una copia dei parametri passati
  - se una procedura PROC ha un parametro x e dentro a PROC il parametro x riceve il valore di y (`x := y`), la modifica non è visibile al di fuori della procedura (per esempio al chiamante)

- Quando un oggetto è passato come parametro, ciò che viene passato è il *puntatore* all'oggetto
  - degli attributi non viene fatta una copia, e modifiche a questi sono visibili al chiamante
  - se x è un parametro che è un oggetto con attributo f, gli effetti dell'assegnamento `x.f := 3` sono visibili al di fuori della procedura
  - questo è il funzionamento di Java.

Modello di computazione

- Quale è la "macchina" sulla quale vengono eseguiti gli algoritmi scritti in pseudocodice?
- La macchina RAM!

- Assunzione di base: ogni istruzione semplice di pseudocodice è tradotta in un numero finito di istruzioni RAM
  - per esempio `x := y` diventa, se a_x e a_y sono gli l'indirizzi in memoria delle variabili x e y (a_x e a_y sono delle costanti):

    LOAD a_y

    STORE a_x

- Da ora in poi adottiamo il *criterio di costo costante*
  - adatto per gli algoritmi che scriveremo, che non manipoleranno mai numeri né richiederanno quantità di memoria molto più grandi della dimensione dei dati in ingresso
- In conseguenza di ciò abbiamo che ogni istruzione i di pseudocodice viene eseguita in un tempo costante c_i
- Grazie a questa assunzione, da adesso in poi possiamo "dimenticarci" che il modello computazionale dello pseudocodice è la macchina RAM

- Inoltre, da ora in poi ci concentriamo sulla *complessità temporale*, più che su quella spaziale

Costo di esecuzione per `INSERTION-SORT`

`INSERTION-SORT(A)`	*costo*	*numero di volte*
for *j* := 2 **to** *A.length*	c_1	n
key := *A*[*j*]	c_2	n - 1
//Inserisce *A*[*j*] nella sequenza *A*[1..*j*−1]	0	n - 1
i := *j* − 1	c_4	n - 1
while *i* > 0 and *A*[*i*] > *key*	c_5	$\sum_{j=2}^{n} t_j$
A[*i* + 1] := *A*[*i*]	c_6	$\sum_{j=2}^{n} (t_j - 1)$
i := *i* − 1	c_7	$\sum_{j=2}^{n} (t_j - 1)$
A[*i* + 1] := *key*	c_8	n - 1

- Note:
  - n = *A.length* = dimensione dei dati in ingresso
  - $t_2, t_3 \ldots t_n$ = numero di volte che la condizione del ciclo **while** viene eseguita quando j = 2, 3, ... n

- Tempo di esecuzione di `INSERTION-SORT`:

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

- Se l'array *A* è già ordinato, $t_2 = \ldots = t_n = 1$
  - $T(n) = an+b$, cioè $T(n) = \Theta(n)$
    - questo è il *caso ottimo*
- Se *A* è ordinato, ma in ordine decrescente, $t_2=2$, $t_3=3$, ... $t_n=n$
  - $T(n) = an^2+bn+c$, cioè $T(n) = \Theta(n^2)$
    - questo è il *caso pessimo*

Un classico problema: l'ordinamento

- L'ordinamento degli elementi di una sequenza è un esempio classico di problema risolto mediante algoritmi
- C'è un gran numero di algoritmi di ordinamento disponibili: insertion sort, bubblesort, quicksort, merge sort, counting sort, ...
- Ne abbiamo appena visto uno di essi: insertion sort

- Abbiamo visto che *nel caso pessimo* $T_{\texttt{INSERTION-SORT}}(n)$ è $\Theta(n^2)$
  - possiamo anche scrivere che $T_{\texttt{INSERTION-SORT}}(n) = O(n^2)$, in quanto il limite superiore (che è raggiunto nel caso pessimo) è una funzione in $\Theta(n^2)$
  - è anche $T_{\texttt{INSERTION-SORT}}(n)=\Omega(n)$, in quanto il limite inferiore (raggiunto nel caso ottimo) è $\Theta(n)$
- Possiamo fare di meglio?
  - possiamo cioè scrivere un algoritmo con un limite superiore migliore?

Merge sort

- Idea dell'algoritmo:
  - se l'array da ordinare ha *meno di* 2 elementi, è già ordinato
  - altrimenti:
    - si divide l'array in 2 sottoarray, ognuno con la metà degli elementi di quello originario
    - si ordinano i 2 sottoarray ri-applicando l'algoritmo
    - si fondono (merge) i 2 sottoarray (che ora sono ordinati)

- `MERGE-SORT` è un algoritmo ricorsivo

pseudocodice di MERGE-SORT

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

- Per ordinare un array $A = [A[1], A[2], \ldots A[n]]$ usiamo `MERGE-SORT(A, 1, A.length)`

- `MERGE-SORT` adotta una tecnica algoritmica classica: **divide et ìmpera**
- Se il problema da risolvere è grosso:
  - *dividilo* in problemi più piccoli della stessa natura
  - *risolvi* (domina) i problemi più piccoli
  - *combina* le soluzioni

- Dopo un po' che dividiamo il problema in altri più piccoli, ad un certo punto arriviamo ad ottenere problemi “piccoli a sufficienza” per poterli risolvere senza dividerli ulteriormente
  - è una tecnica naturalmente ricorsiva in quanto, per risolvere i “problemi più piccoli”, applichiamo lo stesso algoritmo del problema più grosso

- Per completare l'algoritmo dobbiamo definire un sotto-algoritmo `MERGE` che "combina" le soluzioni dei problemi più piccoli

Fusione (merge) di sottoarray *ordinati*

- Definizione del problema (input/output)
  - *Input*: 2 array *ordinati* $A[p..q]$ e $A[q+1..r]$ di un array A
  - *Output*: l'array ordinato $A[p..r]$ ottenuto dalla fusione degli elementi dei 2 array iniziali

- Idea dell'algoritmo:
  1. si va all'inizio dei 2 sottoarray
  2. si prende il minimo dei 2 elementi correnti
  3. si inserisce tale minimo alla fine dell'array da restituire
  4. si avanza di uno nell'array da cui si è preso il minimo
  5. si ripete dal passo 2

pseudocodice

```
MERGE (A, p, q, r)
n1 := q – p + 1
n2 := r – q
crea (alloca) 2 nuovi array L[1..n1+1] e R[1..n2+1]
for i := 1 to n1
  L[i] := A[p + i - 1]
for j := 1 to n2
  R[j] := A[q + j]
L[n1 + 1] := ∞
R[n2 + 1] := ∞
i := 1
j := 1
for k := p to r
  if L[i] ≤ R[j]
    A[k] := L[i]
    i := i + 1
  else A[k] := R[j]
    j := j + 1
```

Esempio di funzionamento di Merge-Sort:

42 16 28 36 26 78 84 8

$\boxed{16 \quad 42}$ $\boxed{28 \quad 36}$ $\boxed{26 \quad 78}$ $\boxed{8 \quad 84}$

$\boxed{16 \quad 28 \quad 36 \quad 42}$ $\boxed{8 \quad 26 \quad 78 \quad 84}$

$\boxed{8 \quad 16 \quad 26 \quad 28 \quad 36 \quad 42 \quad 78 \quad 84}$

Analisi dell'algoritmo `MERGE`

- Nell'algoritmo `MERGE` prima si copiano gli elementi dei 2 sottoarray $A[p..q]$ e $A[q+1..r]$ in 2 array temporanei L e R, quindi si fondono L e R in $A[p..r]$

- Dimensione dei dati in input: $n = r - p + 1$
- L'algoritmo è fatto di 3 cicli **for**:
  - 2 cicli di inizializzazione (l. 4-7), per assegnare i valori a L e R:
  - il primo è eseguito n_1 volte, il secondo n_2 volte, con $n_1 + n_2 = n$, quindi $\Theta(n_1 + n_2) = \Theta(n)$
- Il ciclo principale (l. 12-17) è eseguito n volte, e ogni linea ha costo costante

- In totale $T_{\mathtt{MERGE}}(n) = \Theta(n)$

Complessità di un algoritmo *divide et impera*

- In generale, un algoritmo *divide et impera* ha le caratteristiche seguenti:
  - si divide il problema in *a* sottoproblemi, ognuno di dimensione *1/b* di quello originale
  - se il sottoproblema ha dimensione *n* piccola a sufficienza ($n<c$, con *c* una costante caratteristica del problema), esso può essere risolto in tempo costante (cioè $\Theta(1)$)
  - indichiamo con *D(n)* il costo di dividere il problema, e *C(n)* il costo di ricombinare i sottoproblemi
  - *T(n)* è il costo per risolvere il problema totale

- Possiamo esprimere il costo T(n) tramite la seguente *equazione di ricorrenza* (o *ricorrenza*):

$$\mathrm{T}(n)=\begin{cases}\Theta(1) & \text{se } n<c\\ \mathrm{D}(n)+a\,\mathrm{T}(n/b)+\mathrm{C}(n) & \text{altrimenti}\end{cases}$$

- Ricorrenza per l'algoritmo `MERGE-SORT`:
  $a = b = c = 2$, $D(n) = \Theta(1)$, $C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < 2 \\ 2T(n/2) + \Theta(n) & \text{altrimenti} \end{cases}$$

  - in realtà dovrebbe essere $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$ invece di $2T(n/2)$, ma l'approssimazione non influisce sul comportamento asintotico della funzione $T(n)$

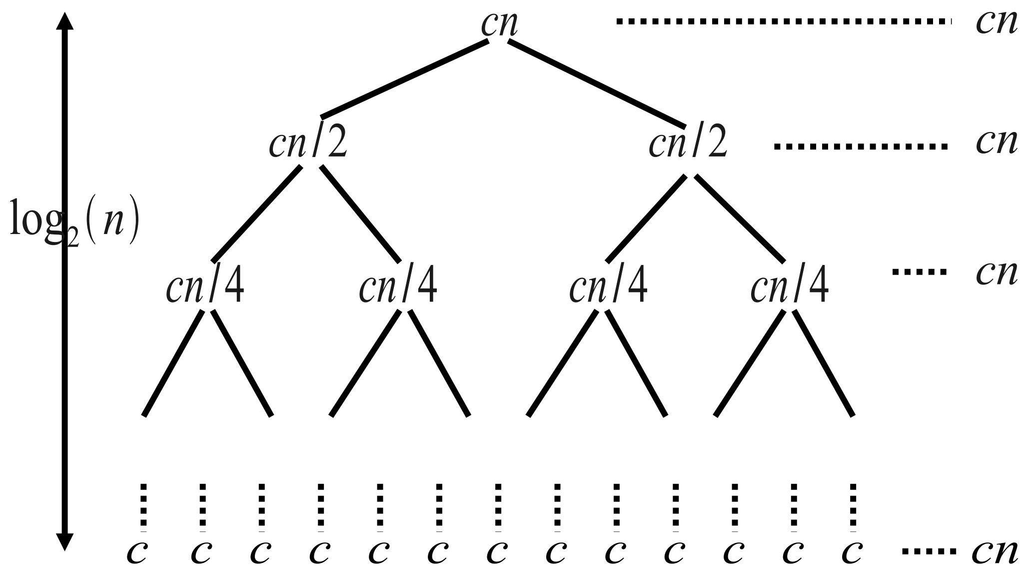
- Come risolviamo le ricorrenze? Vedremo tra poco, per ora:

Complessità di MERGE-SORT

- Riscriviamo la ricorrenza di MERGE-SORT:

$$\mathrm{T}(n) = \begin{cases} c & \text{se } n < 2 \\ 2\,\mathrm{T}(n/2) + cn & \text{altrimenti} \end{cases}$$

- Possiamo disegnare l'*albero di ricorsione* (consideriamo per semplicità il caso in cui la lunghezza n dell'array è una potenza di 2)



Totale: $cn\log(n) + cn$

- Sommando i costi dei vari livelli otteniamo
$\mathrm{T}(n) = cn \log(n) + cn$, cioè $\mathrm{T}_{\texttt{MERGE-SORT}}(n) = \Theta(n \log(n))$

Risoluzione di ricorrenze

- Tre tecniche principali:
  - sostituzione
  - albero di ricorsione
  - teorema dell'esperto (master theorem)

- Metodo della sostituzione:
  - formulare un'ipotesi di soluzione
  - sostituire la soluzione nella ricorrenza, e dimostrare (per induzione) che è in effetti una soluzione

- Esempio, cerchiamo un limite superiore per la seguente $T(n)$:
  $T(n) = 2T(\lfloor n/2 \rfloor) + n$
  - supponiamo $T(n) = O(n \log_2(n))$
  - dobbiamo mostrare che $T(n) \leq cn \log_2(n)$ per una opportuna costante $c > 0$ (def. O)
  - (Ip. induttiva) supponiamo che valga per $T(\lfloor n/2 \rfloor)$, cioè $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor)$
  - sostituendo in $T(n)$ abbiamo $T(n) \leq 2c\lfloor n/2 \rfloor \log_2(\lfloor n/2 \rfloor) + n \leq cn \log_2(n/2) + n =$
    $= cn \log_2(n) - cn \log_2(2) + n = cn \log_2(n) -cn + n \leq cn \log_2(n)$
    - basta che $c \geq 1$
  - (Condizione al contorno) dobbiamo inoltre mostrare che la disuguaglianza vale per $n = 1$; supponiamo che sia $T(1) = 1$, allora $T(1) = 1 \leq c1 \log_2(1) = 0$? No!
  - però $T(n) \leq cn \log_2(n)$ deve valere solo da un certo n_0 in poi, che possiamo scegliere arbitrariamente; prendiamo $n_0 = 2$, e notiamo che, se $T(1) = 1$, allora, dalla ricorrenza, $T(2) = 4$ e $T(3) = 5$
    - inoltre, per $n > 3$ la ricorrenza non dipende più dal problematico $T(1)$
  - ci basta determinare una costante c tale che $T(2) = 4 \leq c2 \log_2(2)$ e $T(3) = 5 \leq c3 \log_2(3)$
  - per ciò basta prendere $c \geq 2$

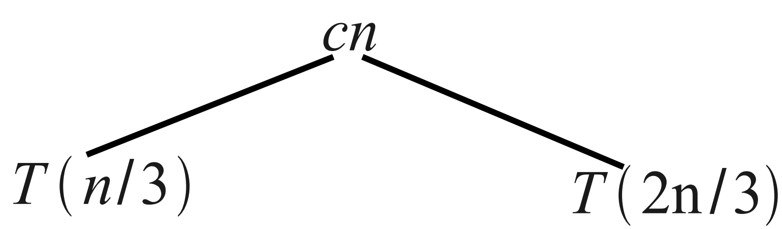
Osservazioni sul metodo di sostituzione

- Consideriamo il seguente caso:
  $T(n) = T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+1$
- Proviamo a vedere se $T(n) = O(n)$:
  - $T(n) \leq c\lfloor n/2 \rfloor+c\lceil n/2 \rceil+1 = cn+1$
  - basta prendere $c = 1$ e siamo a posto?
  - No, perché non abbiamo dimostrato la forma esatta della disuguaglianza!
    - dobbiamo derivare che il tutto è $\leq cn$, ma $cn+1$ non è $\leq cn$
- Potremmo prendere un limite più alto, e dimostrare che $T(n)$ è $O(n^2)$ (cosa che è vera), ma in effetti si può anche dimostrare che $T(n) = O(n)$, dobbiamo solo fare un piccolo aggiustamento.
- Mostriamo che $T(n) \leq cn-b$, con b un'opportuna costante
  - se fosse così, allora $T(n) = O(n)$
  - $T(n) \leq c\lfloor n/2 \rfloor-b+c\lceil n/2 \rceil-b+1 = cn-2b+1 \leq cn - b$
    - basta prendere $b \geq 1$

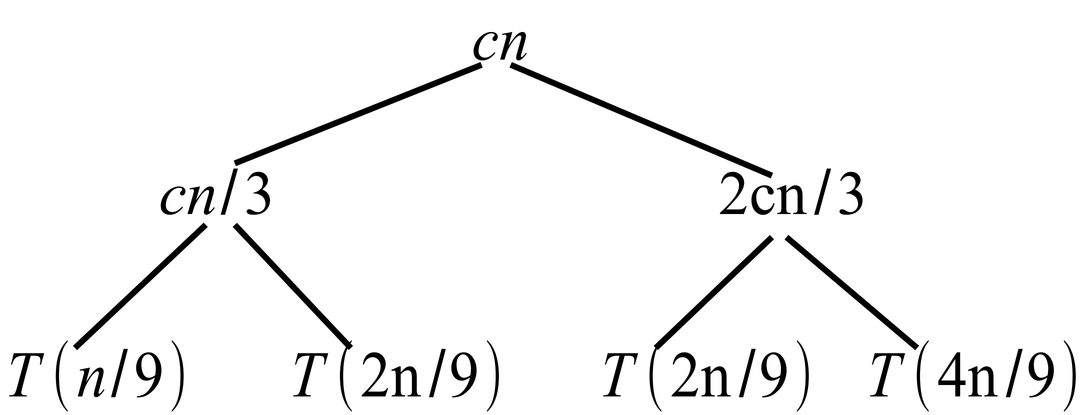
- Altro esempio:
  $T(n) = 2T(\lfloor\sqrt{n}\rfloor) + \log_2(n)$
  - poniamo $m = \log_2(n)$, quindi $n = 2^m$, otteniamo
  - $T(2^m) = 2T(2^{m/2})+m$
  - Ponendo $S(m) = T(2^m)$ abbiamo $S(m) = 2S(m/2) + m$ quindi $S(m) = O(m \log_2(m))$
  - Quindi, sostituendo all'indietro: $T(n) = O(\log_2(n) \log_2\log_2(n))$

Metodo dell'albero di ricorsione

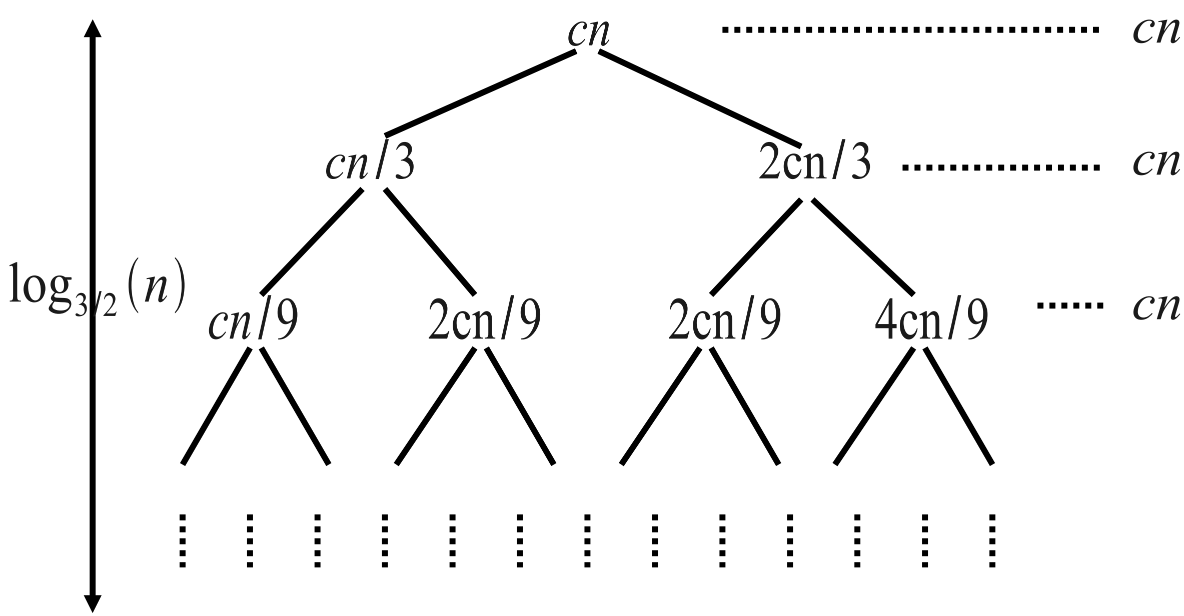
- Un metodo non molto preciso, ma utile per fare una congettura da verificare poi con il metodo di sostituzione
- Idea: a partire dalla ricorrenza, sviluppiamo l'albero delle chiamate, indicando per ogni chiamata la sua complessità
- Esempio: $T(n) = T(\lfloor n/3 \rfloor) + T(\lfloor 2n/3 \rfloor) + O(n)$
  - Prima chiamata:



  - Espandiamo:



– fino in fondo:



– Se l'albero fosse completo, sommando i costi livello per livello, a ogni livello avremmo un costo cn, ed il numero di livelli k sarebbe tale che $n(2/3)^k=1$, cioè $k = \log_{3/2} n$.

Albero di ricorsione (2)

- Però l'albero non è completo
  - il ramo più a destra è sì tale che alla fine $n(2/3)^k=1$, ma quello più a sinistra è tale che $n(1/3)^{k'}=1$, cioè $k' = \log_3 n$
- Però possiamo prendere l'altezza dell'albero minore, cioè k', e la maggiore, k.
- Sicuramente $T(n) = O(n\ k)$ e $T(n) = \Omega(n\ k')$, cioè $T(n) = O(n \log_{3/2} n)$ e $T(n) = \Omega(n \log_3 n)$.
- Ma il cambio di logaritmo ha un impatto solo sulla costante moltiplicativa, quindi posso direttamente dire $T(n) = \Theta(n \log_2 n)$

Teorema dell'esperto (Master Theorem)

- Data la ricorrenza:
  $T(n) = aT(n/b) + f(n)$
  (in cui $a \geq 1$, $b > 1$, e n/b è o $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$)
  1. se $f(n) = O(n^{\log_b a - \varepsilon})$ per qualche $\varepsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$
  2. se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log(n))$
  3. se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per qualche $\varepsilon > 0$, e $af(n/b) \leq cf(n)$ per qualche $c < 1$ e per tutti gli n grandi a sufficienza, allora $T(n) = \Theta(f(n))$

Osservazioni

- Nota $\log_b a = (\log a)/(\log b)$
  cioè (log numero sottoproblemi)/(log dimensioni sottoproblemi)
- La soluzione è data dal più grande tra $n^{\log_b a}$ e $f(n)$
  - se $n^{\log_b a}$ è il più grande, $T(n)$ è $\Theta(n^{\log_b a})$
  - se $f(n)$ è il più grande, $T(n)$ è $\Theta(f(n))$
  - se sono nella stessa classe secondo la relazione Θ, $T(n)$ è $\Theta(f(n)\log(n))$
- "Più grande" o "più piccolo" in effetti è "*polinomialmente* più grande" e "*polinomialmente* più piccolo"
  - n è polinomialmente più piccolo di n^2
  - $n \log(n)$ è polinomialmente più grande di $n^{½}$
- Il teorema dell'esperto non copre tutti i casi!
  - se una delle due funzioni è più grande, ma non polinomialmente più grande...
  - $n \log(n)$ è più grande di n, ma non *polinomialmente* più grande

- Esempio: applichiamo il teorema dell'esperto a `MERGE-SORT`:
  - $T(n) = 2T(n/2) + \Theta(n)$
    - $a = b = 2$
    - $f(n) = n$
    - $n^{\log_b a} = n^1 = n$
  - siamo nel caso 2: $T_{\text{MERGE-SORT}}(n) = \Theta(n \log(n))$

Un caso particolare

- Notiamo che l'enunciato del teorema dell'esperto si semplifica un po' se f(n) è una funzione $\Theta(n^k)$, con k una qualche costante:
  1. se $k < \log_b a$, allora $T(n) = \Theta(n^{\log_b a})$
  2. se $k = \log_b a$, allora $T(n) = \Theta(n^k \log(n))$
  3. se $k > \log_b a$, allora $T(n) = \Theta(n^k)$
  - nel caso 3 la condizione aggiuntiva è automaticamente verificata

Un ulteriore risultato

- Data la ricorrenza (in cui i coefficienti a_i sono interi ≥ 0)

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

- in cui poniamo $a = \sum_{1 \leq i \leq h} a_i$
- allora abbiamo che:
  - se a = 1, allora $T(n)=O(n^{k+1})$
  - se $a \geq 2$, allora $T(n)=O(a^n n^k)$

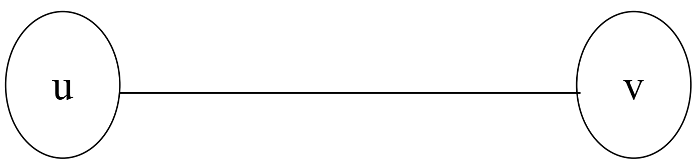
- Per esempio, data la ricorrenza $T(n) = T(n-1) + \Theta(n)$, otteniamo $T(n) = O(n^2)$
  - questa è la ricorrenza che otterremmo con una versione ricorsiva di `INSERTION-SORT`

Grafi (richiamo)

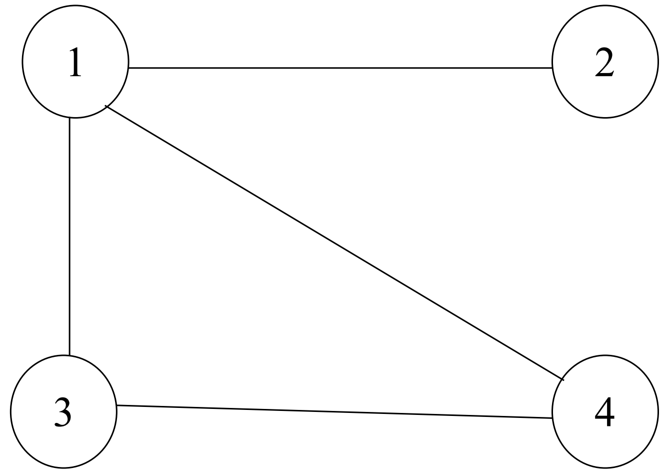
- Un *grafo* è una coppia (V, E) in cui V è un insieme finito di *nodi* (detti anche *vertici*), e $E \subseteq V \times V$ è una relazione binaria su V che rappresenta gli *archi* del grafo
  - se u e v sono nodi del grafo, la coppia (u,v) è un arco, ed è rappresentata graficamente come:

    in questo caso l'arco è *orientato*, in quanto c'è un ordine tra i nodi, prima u, poi v

  - se non c'è un ordine tra i nodi (che quindi sono solo un insieme, $\{u,v\}$) allora diciamo che l'arco è *non orientato*:



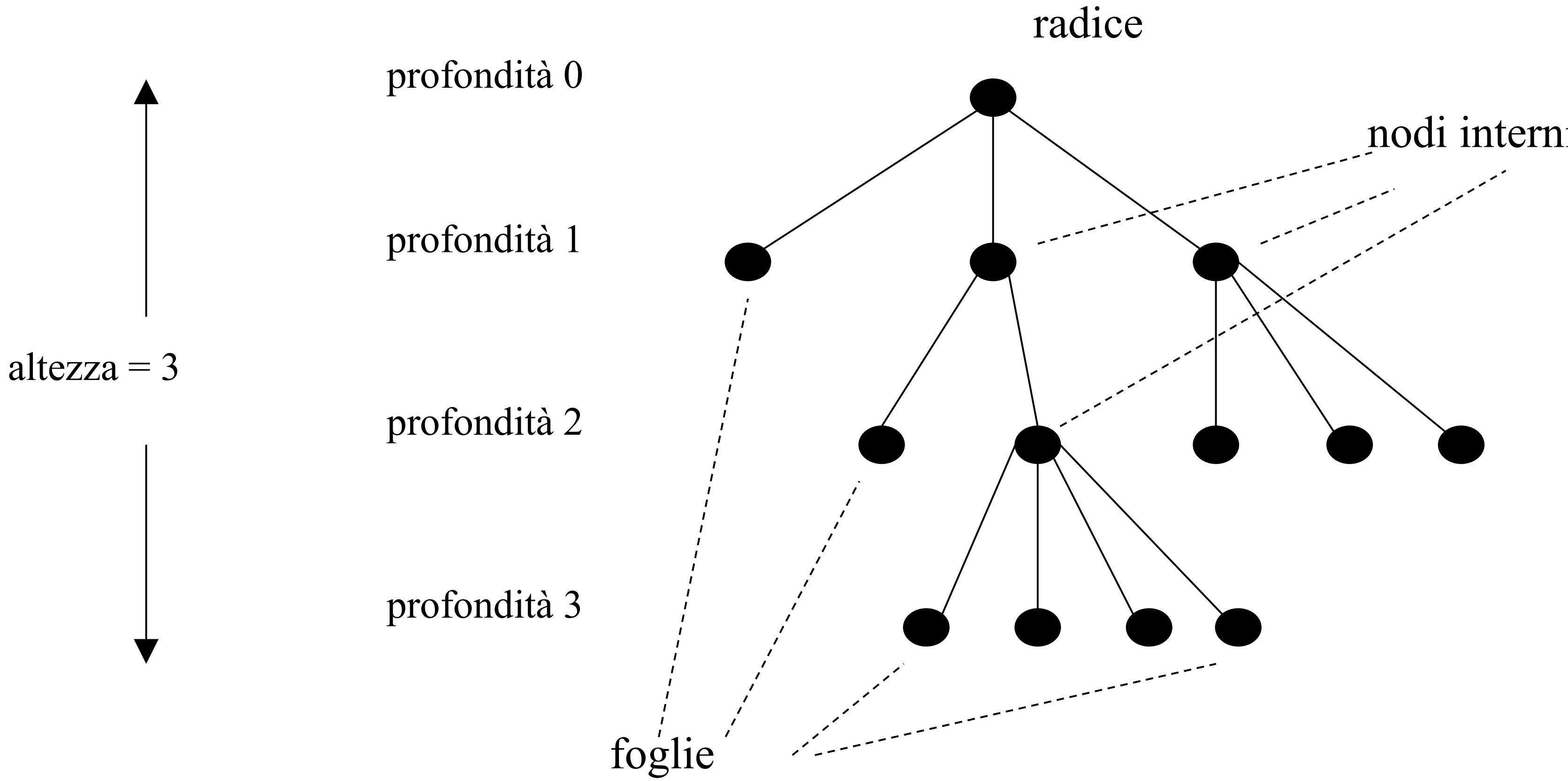
- Un grafo è *orientato* se i suoi archi lo sono, *non orientato* altrimenti
  - esempio di grafo non orientato:



- Un *cammino* è una sequenza di nodi $v_0, v_1, v_2, \ldots, v_n$ tali che tra ogni coppia di nodi della sequenza (v_i, v_{i+1}) c'è un arco
  - i nodi $v_0, \ldots v_n$ appartengono al cammino
  - la lunghezza del cammino è data da n (numero di vertici -1)
- In un grafo non orientato, il cammino forma un ciclo se $v_0 = v_n$
  - Un grafo che non ha cicli è *aciclico*
- Un grafo non orientato è *connesso* se tra ogni coppia di nodi esiste un cammino

Alberi (richiamo)

- Un *albero* è un grafo connesso, aciclico, non orientato
  - un albero è *radicato* se un nodo viene indicato come la *radice*



- Ogni nodo dell'albero è raggiungibile dalla radice tramite un cammino (che è unico, in quanto il grafo è aciclico).
- Chiamiamo *foglie* gli ultimi nodi dei cammini dalla radice.

- Ogni nodo ha un *padre* (a parte la radice) e uno o più *figli* (a parte le foglie).

- Chiamiamo:
  - *nodi interni*: tutti i nodi dei cammini tra la radice e le foglie
  - *profondità* (di un nodo *N*): la distanza di *N* dalla radice
  - *altezza* (dell'albero): la distanza massima tra la radice e una foglia
  - *antenato* (di un nodo *N*): ogni nodo che precede *N* sul cammino dalla radice a *N*
  - *padre* (di un nodo *N*): il nodo che immediatamente precede *N* lungo il cammino dalla radice a *N*
  - *figlio* (di un nodo *N*): ogni nodo di cui *N* è padre
  - *fratelli* (di un nodo *N*): i nodi che hanno lo stesso padre di *N*

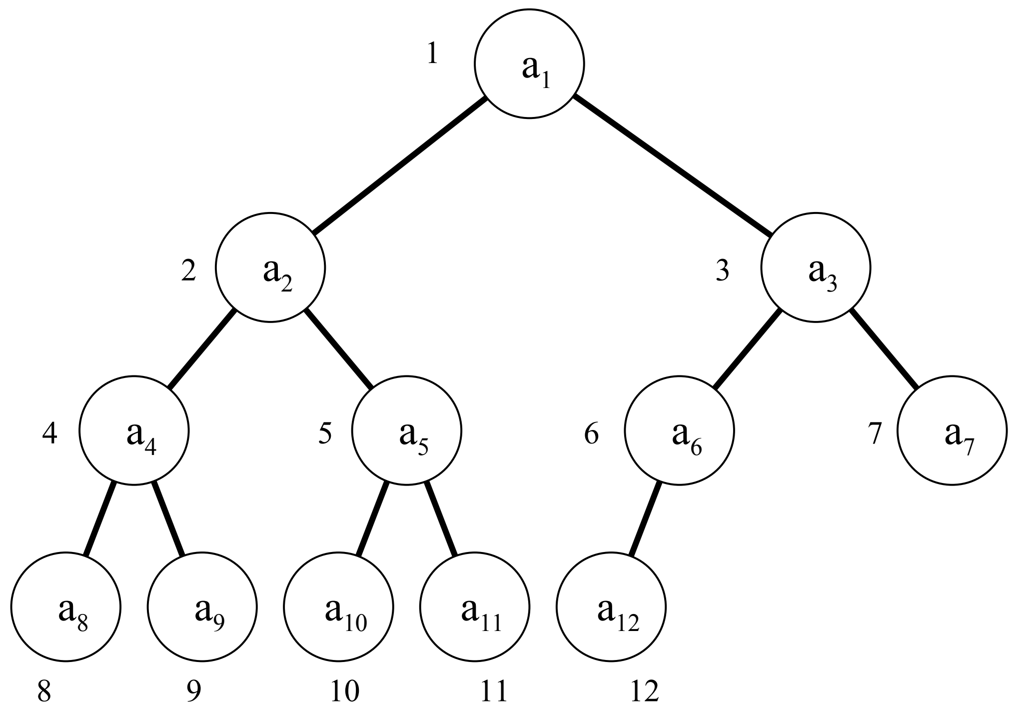
- Un albero è *binario* se ogni nodo ha *al più* 2 figli

HEAPSORT

- `MERGE-SORT` è efficiente dal punto di vista del tempo di esecuzione, ma non è ottimale dal punto di vista dell'uso della memoria
  - ogni `MERGE` richiede di allocare 2 array, di lunghezza $\Theta(n)$
  - usa una quantità di memoria aggiuntiva rispetto all'array da ordinare che non è costante, cioè non ordina *sul posto*
- `HEAPSORT`, invece, non solo è efficiente (ordina in tempo $\Theta(n \log(n))$), ma ordina sul posto
- L'idea alla base di `HEAPSORT` è che un array può essere visto come un albero binario:
  - $A[1]$ è la radice
  - per ogni elemento $A[i]$, $A[2i]$ e $A[2i+1]$ sono i suoi figli, e $A[\lfloor i/2 \rfloor]$ è il padre

- Esempio:

1	2	3	4	5	6	7	8	9	10	11	12
a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}



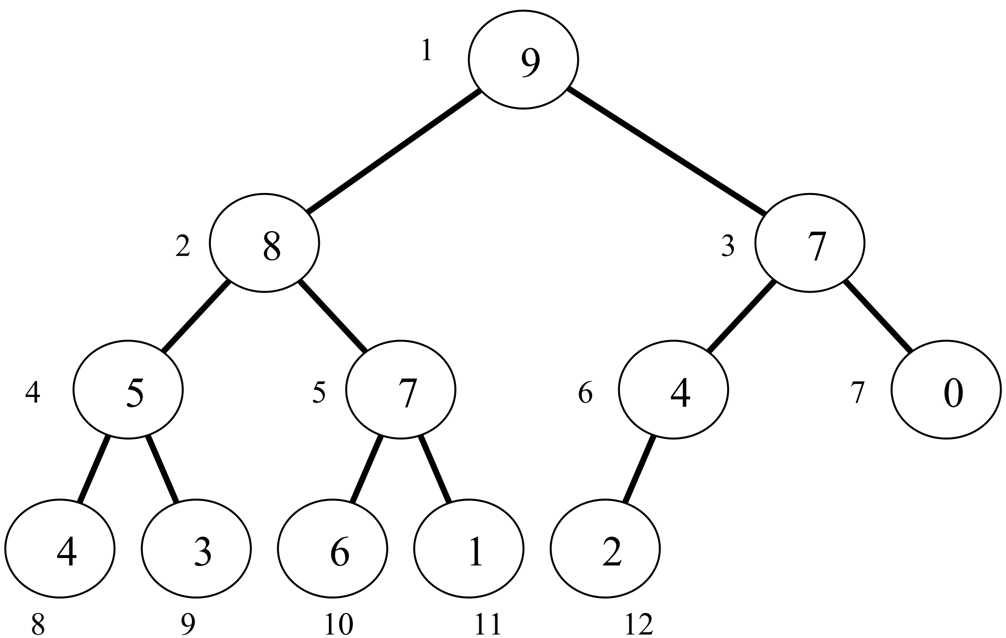
Gli heap (mucchi)

- Uno **heap binario** è un albero binario *quasi completo*
  - quasi completo = tutti i livelli sono completi, tranne al più l'ultimo, che potrebbe essere completo solo fino a un certo punto da sinistra
  - l'albero binario che deriva dall'interpretazione di un array come albero è quasi completo
- Un **max-heap** è uno heap tale che, per ogni nodo x dell'albero, il valore contenuto nel padre di x è $\geq$ del contenuto di x
  - usando la corrispondenza albero-heap, questo vuole dire che $A[\lfloor i/2 \rfloor] \geq A[i]$
- Il concetto di min-heap è perfettamente duale

- Gli heap sono una struttura dati utilizzata per varie cose, a parte l'ordinamento, ad esempio sono comode ed efficienti per gestire *code a priorità*

- Esempio:

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



- Si noti che in un max-heap l'elemento massimo è nella radice
  – dove è il minimo?

Alcune operazioni sugli heap

- Operazioni di base:

```
PARENT(i)
1 return ⌊i/2⌋
```

```
LEFT(i)
1 return 2*i
```

```
RIGHT(i)
1 return 2*i + 1
```

- Quindi, in un max-heap abbiamo che $A[\mathtt{PARENT}(i)] \geq A[i]$
  - esistono anche i min-heap, per le quali $A[\mathtt{PARENT}(i)] \leq A[i]$
- Per realizzare l'ordinamento usiamo i max-heap
- Ogni array A che rappresenta uno heap ha 2 attributi:
  - *A.length*, che rappresenta il numero totale di elementi dell'array
  - *A.heap-size*, che rappresenta il numero di elementi dello heap
  - *A.heap-size* ≤ *A.length*, e solo gli elementi fino a *A.heap-size* hanno la proprietà dello heap
  - l'array potrebbe contenere elementi dopo l'indice *A.heap-size*, se *A.heap-size* < *A.length*

Algoritmi di supporto

- Un algoritmo che, dato un elemento di un array tale che i suoi figli sinistro e destro sono dei max-heap, ma in cui $A[i]$ (la radice del sottoalbero) potrebbe essere < dei suoi figli, modifica l'array in modo che tutto l'albero di radice $A[i]$ sia un max-heap

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

- $T_{\texttt{MAX-HEAPIFY}} = O(h)$, dove h è l'altezza dell'albero, che è $O(\log(n))$, poiché l'albero è quasi completo
  - quindi, $T_{\texttt{MAX-HEAPIFY}} = O(\log(n))$
- Questo si sarebbe anche potuto mostrare usando il teorema dell'esperto per la seguente ricorrenza, che rappresenta il tempo di esecuzione di `MAX-HEAPIFY` nel caso pessimo: $T(n) = T(2n/3) + \Theta(1)$
  - nel caso pessimo l'ultimo livello dell'albero è esattamente pieno a metà, e l'algoritmo viene applicato ricorsivamente sul sottoalbero sinistro, che risulta quindi il più grande possibile.

Da array a heap

- Algoritmo per costruire un max-heap a partire da un array
  - idea: costruiamo il max-heap bottom-up, dalle foglie, fino ad arrivare alla radice
    - osservazione fondamentale: tutti gli elementi dall'indice `A.length/2` in poi sono delle foglie, quelli prima sono dei nodi interni
    - i sottoalberi fatti di solo foglie, presi singolarmente, sono già dei max-heap, in quanto composti da un unico elemento

```
BUILD-MAX-HEAP(A)
A.heap-size := A.length
for i := A.length/2 downto 1
  MAX-HEAPIFY(A, i)
```

- Costo di `BUILD-MAX-HEAP`?
  - ad occhio, ogni chiamata a `MAX-HEAPIFY` costa O(log(n)), e vengono fatte *n* chiamate (con *n* che è `A.length`), quindi il costo è O(n log(n))
  - ma in realtà questo limite non è stretto...

- Osserviamo che:
  - l'altezza di un albero quasi completo di *n* nodi è $\lfloor \log_2(n) \rfloor$
  - se definiamo come "altezza di un nodo di uno heap" la lunghezza del cammino più lungo che porta ad una foglia, il costo di `MAX-HEAPIFY` invocato su un nodo di altezza *h* è O(h)
  - il numero massimo di nodi di altezza *h* di uno heap è $\lceil n/2^{h+1} \rceil$
- Quindi `MAX-HEAPIFY` viene invocato $\lceil n/2^{h+1} \rceil$ volte ad ogni altezza h, quindi il costo di `BUILD-MAX-HEAP` è

$$\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

- cioè O(n), in quanto è noto che $\displaystyle\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

HEAPSORT

- Possiamo a questo punto scrivere l'algoritmo di `HEAPSORT`:

```
HEAPSORT(A)
BUILD-MAX-HEAP(A)
for i := A.length downto 2
  swap A[1] ↔ A[i]
  A.heap-size := A.heap-size − 1
  MAX-HEAPIFY(A,1)
```

  - idea: a ogni ciclo piazziamo l'elemento più grande (che è il primo dell'array, in quanto questo è un max-heap) in fondo alla parte di array ancora da ordinare (che è quella corrispondente allo heap)

- La complessità di `HEAPSORT` è O(n log(n)), in quanto
  - `BUILD-MAX-HEAP` ha costo O(n)
  - `MAX-HEAPIFY` è invocato n volte, e ogni sua chiamata ha costo O(log(n))

esempio funzionamento

A = [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]

heap A ad ogni passo dopo MAX-HEAPIFY:

```
[16, 14, 10, 8, 7, 9, 3,  2,  4,  1] max-heap
[14, 8,  10, 4, 7, 9, 3,  2,  1, 16]
[10, 8,  9,  4, 7, 1, 3,  2, 14, 16]
[9,  8,  3,  4, 7, 1, 2, 10, 14, 16]
[8,  7,  3,  4, 2, 1, 9, 10, 14, 16]
[7,  4,  3,  1, 2, 8, 9, 10, 14, 16]
[4,  2,  3,  1, 7, 8, 9, 10, 14, 16]
[3,  2,  1,  4, 7, 8, 9, 10, 14, 16]
[2,  1,  3,  4, 7, 8, 9, 10, 14, 16]
[1,  2,  3,  4, 7, 8, 9, 10, 14, 16]
```

QUICKSORT

- `QUICKSORT` è un algoritmo in stile divide-et-impera
  - ordina *sul posto*
- Nel caso pessimo (vedremo) ha complessità $\Theta(n^2)$
- Però in media funziona molto bene (in media ha complessità $\Theta(n \log(n))$)
  - inoltre ha ottime costanti

- Idea di base del `QUICKSORT`: dato un sottoarray $A[p..r]$ da ordinare:
  - (dividi) riorganizza in $A[p..r]$ 2 sottoarray $A[p..q\text{-}1]$ e $A[q+1..r]$ tali che tutti gli elementi di $A[p..q\text{-}1]$ sono $\leq A[q]$ e tutti gli elementi di $A[q+1..r]$ sono $\geq A[q]$
  - NB: A[q] è già al suo posto, quindi non verrà più spostato
  - (impera) ordina i sottoarray $A[p..q\text{-}1]$ e $A[q+1..r]$ riutilizzando `QUICKSORT`
  - (combina) nulla! L'array $A[p..r]$ è già ordinato

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

- Per ordinare un array A: QUICKSORT(A, 1, $A.length$)

- La parte più difficile di `QUICKSORT` è il partizionamento:

```
PARTITION(A, p, r)
x := A[r]
i := p − 1
for j := p to r − 1
  if A[j] ≤ x
    i := i + 1
    swap A[i] ↔ A[j]
swap A[i+1] ↔ A[r]
return i + 1
```

  - l'elemento x (cioè $A[r]$ in questa implementazione) è il *pivot* (o perno)
  - da p a i (inclusi): partizione con elementi ≤ x
  - da $i+1$ a $j-1$: partizione con elementi > x

```
[[p  (≤x)   i][      (>x)    ]j        r]
```

- Complessità di `PARTITION`: Θ(n), con $n = r - p + 1$

esempio funzionamento

Quicksort su 99, 4, 88, 7, 5, -3, 1, 34, 11

```
 1   2   3    4   5    6    7    8    9
[99,4, 88, 7, 5, -3,  1, 34, 11]   p: 1, r: 9
 p                               r
[4, 7, 5, -3, 1, 11, 88, 34, 99]   p: 1, q: 6, r: 9
 p                 q             r
[-3, 1, 5, 4, 7, 11, 88, 34, 99]   p: 1, q: 2, r: 5
 p   q           r
[-3, 1, 5, 4, 7, 11, 88, 34, 99]   p: 3, q: 5, r: 5
       p        qr
[-3, 1, 4, 5, 7, 11, 88, 34, 99]   p: 3, q: 3, r: 4
      pq   r
[-3, 1, 4, 5, 7, 11, 88, 34, 99]   p: 7, q: 9, r: 9
                        p       qr
[-3, 1, 4, 5, 7, 11, 34, 88, 99]   p: 7, q: 7, r: 8
                       pq   r
```

Complessità di `QUICKSORT`

- Il tempo di esecuzione di `QUICKSORT` dipende da come viene partizionato l'array
- Se ogni volta uno dei 2 sottoarray è vuoto e l'altro contiene n-1 elementi si ha il caso pessimo
  - la ricorrenza in questo caso è:
    $T(n) = T(n-1) + \Theta(n)$
    - abbiamo visto che la soluzione di questa ricorrenza è $O(n^2)$
    - si può anche dimostrare (per esempio per sostituzione) che è anche $\Theta(n^2)$
  - un caso in cui si ha sempre questa situazione completamente sbilanciata è quando l'array è già ordinato

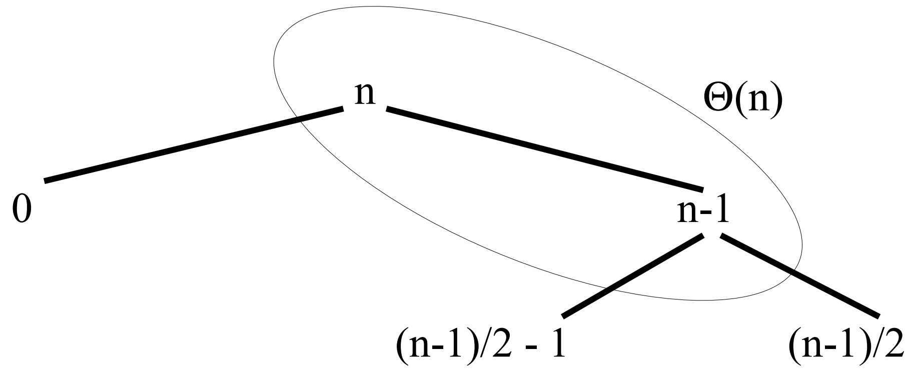
Complessità di `QUICKSORT`

- Nel caso ottimo, invece, i 2 array in cui il problema viene suddiviso hanno esattamente la stessa dimensione n/2
  - la ricorrenza in questo caso è:
    $T(n) = 2T(n/2) + \Theta(n)$
  - è la stessa ricorrenza di `MERGE-SORT`, ed ha quindi la stessa soluzione $\Theta(n \log(n))$
- Notiamo che se la proporzione di divisione, invece che essere n/2 ed n/2, fosse n/10 e 9n/10, comunque la complessità sarebbe $\Theta(n \log(n))$
  - solo, la costante “nascosta” dalla notazione Θ sarebbe più grande
  - abbiamo già visto qualcosa di molto simile per la suddivisione n/3 e 2n/3

`QUICKSORT` nel caso medio (solo intuizione)

- In media ci va un po' bene ed un po' male
  - bene = partizione ben bilanciata
  - male = partizione molto sbilanciata
- Qualche semplificazione:
  - ci va una volta bene ed una volta male
  - quando va bene: ottimo
    - n/2 e n/2
  - quando va male: pessimo
    - n-1 e 0

- Albero di ricorsione in questo caso (ogni divisione costa n):



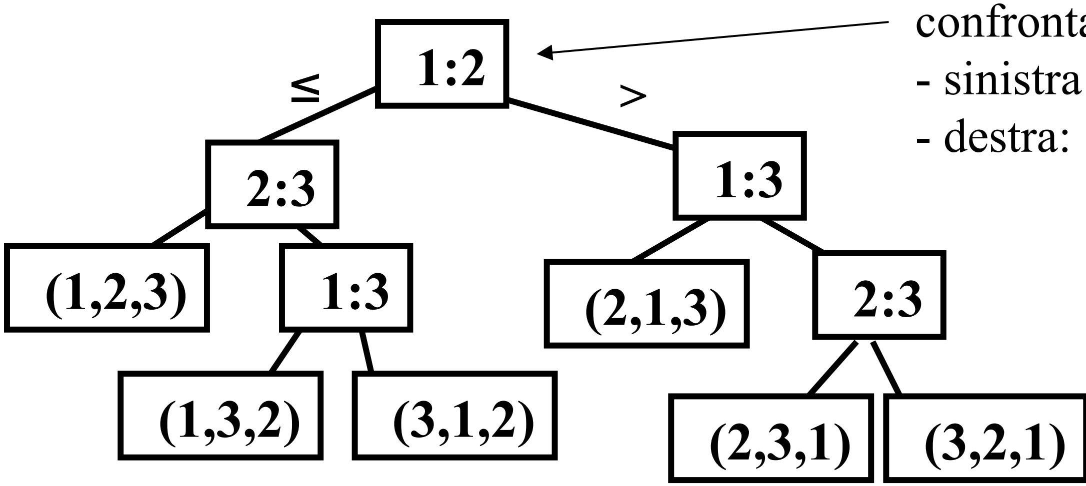
  – costo di una divisione “cattiva” + una divisione “buona” = $\Theta(n)$
    - è lo stesso costo di una singola divisione “buona”
  – dopo una coppia “divisione cattiva” – “divisione buona” il risultato è una divisione “buona”
  – quindi alla fine il costo di una coppia “cattiva – buona” è lo stesso di una divisione “buona”, ed il costo di una catena di tali divisioni è la stessa.
  – rispetto al caso ottimo, l'albero praticamente raddoppia come altezza
  – quindi: $\Theta(n \log(n))$
    - le costanti moltiplicative peggiorano, ma il comportamento asintotico rimane lo stesso del caso ottimo

Limite inferiore per ordinamento

- Quanto veloce può andare un algoritmo di ordinamento? Possiamo far meglio di *n log n*?
- Partiamo col concetto di *ordinamento per confronto*: supponiamo di poter ordinare esclusivamente confrontando il valore di coppie di elementi (quello che abbiamo sempre fatto)
- Limiti inferiori: $\Omega(n)$ – per forza dobbiamo leggere gli elementi!
- però per ora tutti gli ord. che abbiamo visto sono $\Omega(n \log n)$
- proviamo ad astrarre il problema dell'ordinamento per confronto:

contiamo esclusivamente i confronti:

- otteniamo un albero di decisione (esempio InsertSort):



confronta A[1] con A[2]:
- sinistra: ok,
- destra: scambio i due elementi

arrivati alle foglie, A è ordinato

- quante foglie ci sono? Tutte le permutazioni sono n!, però la stessa perm. potrebbe apparire più volte... ergo: > n!

- Posso costruire, dato un n, un albero simile per qualsiasi ordinamento per confronto (si confrontano x e y: solo due possibili risultati: $x \leq y$ (siamo già a posto) oppure $x > y$)
- qual'è la lunghezza massima dalla radice a una foglia? Dipende dall'algoritmo di ordinamento: es. InsertionSort: $\Theta(n^2)$,

  MergeSort: $\Theta(n \log n)$...

- Sfrutto il seguente Lemma:

L1: *Ogni albero binario di altezza h ha un numero di foglie al più* 2^h

- (dim banale - per induzione)
- A questo punto:

Teorema:

Ogni albero di decisione di ordinamento di n elementi ha altezza $\Omega(n \log n)$.

Dim:

Sia f il numero di foglie. Abbiamo visto prima che $f \geq n!$

Per L1: $n! \leq f \leq 2^h$, cioè $2^h \geq n!$

Questo vuol dire: $h \geq \log(n!)$

Sfruttiamo l'approssimazione di Stirling: $n! > (n/e)^n$

Ne segue:

$h \geq \log((n/e)^n) = n \log(n/e) = n \log n - n \log e = \Omega(n \log n)$

COUNTING-SORT

- Ipotesi fondamentale: i valori da ordinare sono tutti numeri naturali compresi tra 0 e una certa costante *k*
- Idea di base: se nell'array ci sono m_e valori più piccoli di un certo elemento *e* (il cui valore è v_e) nell'array ordinato l'elemento *e* sarà in posizione m_e+1
  - quindi, basta contare quante "copie" dello stesso valore v_e sono contenute nell'array
  - usiamo questa informazione per determinare, per ogni elemento *e* (con valore v_e tale che $0 \leq v_e \leq k$), quanti elementi ci sono più piccoli di *e*
  - dobbiamo anche tenere conto del fatto che nell'array ci possono essere elementi ripetuti, es. [2, 7, 2, 5, 1, 1, 9]

pseudocodice

- parametri: *A* è l'array di input (disordinato), *B* conterrà gli elementi ordinati (cioè è l'output), e *k* è il massimo tra i valori di *A*
    - *A* e *B* *devono* essere della stessa lunghezza *n*

```
COUNTING-SORT (A, B, k)
for i := 0 to k
  C[i] := 0
for j := 1 to A.length
  C[A[j]] := C[A[j]] + 1
//C[i] ora contiene il numero di elementi uguali a i
for i := 1 to k
  C[i] := C[i] + C[i - 1]
//C[i] ora contiene il numero di elementi ≤ i
for j := A.length downto 1
  B[C[A[j]]] := A[j]
  C[A[j]] := C[A[j]] - 1
```

esempio di `COUNTING-SORT`

- Se A = [2,5,3,0,2,3,0,3]
  - *A.length* = 8
  - *B* deve avere lunghezza 8

- Se eseguiamo `COUNTING-SORT`(A, B, 5)
  - prima di eseguire la linea 5 (cioè alla fine del loop 3-4)
    C = [2,0,2,3,0,1]

  - prima di eseguire la linea 8 C = [2,2,4,7,7,8]

  - dopo le prime 3 iterazioni del ciclo 9-11 abbiamo
    1. B = [_,_,_,_,_,_,3,_], C = [2,2,4,6,7,8]
    2. B = [_,0,_,_,_,_,3,_], C = [1,2,4,6,7,8]
    3. B = [_,0,_,_,_,3,3,_], C = [1,2,4,5,7,8]

  - alla fine dell'algoritmo
    B = [0,0,2,2,3,3,3,5], C = [0,2,2,4,7,7]

complessità di `COUNTING-SORT`

- La complessità di `COUNTING-SORT` è data dai 4 cicli **for**:
  - il ciclo **for** delle linee 1-2 ha complessità $\Theta(k)$
  - il ciclo **for** delle linee 3-4 ha complessità(n)
  - il ciclo **for** delle linee 6-7 ha complessità(k)
  - il ciclo **for** delle linee 9-11 ha complessità(n)
- La complessità globale è $\Theta(n + k)$
- Se k è O(n), allora il tempo di esecuzione è O(n)
  - lineare!

- `COUNTING-SORT` è "più veloce" (cioè ha complessità inferiore) di `MERGE-SORT` e `HEAPSORT` (*se* k è O(n)) perché fa delle *assunzioni* sulla distribuzione dei valore da ordinare (assume che siano tutti $\leq k$)
  - sfrutta l'assunzione: è veloce se k è O(n), altrimenti ha complessità maggiore (anche di molto) di `MERGE-SORT` e `HEAPSORT`

Nota bene:

- si può ottenere una versione più semplice dell'algoritmo senza usare l'array B (come?)
- la versione che abbiamo appena visto è però *stabile*
- questo vuol dire che, se nell'array da ordinare ci sono più elementi con lo stesso valore, questi appariranno nell'array ordinato mantenendo il loro ordine relativo iniziale
- es: supponiamo che in A ci siano due 35, il primo lo chiamiamo 35_a e il secondo 35_b. Dopo l'ordinamento 35_a apparirà sicuramente prima di 35_b.
- Questa proprietà non è particolarmente interessante se ci limitiamo ad ordinare numeri. Lo diviene se stiamo ordinando dei *dati complessi* (oggetti), utilizzando un valore per es. dei loro attributi come *chiave*.