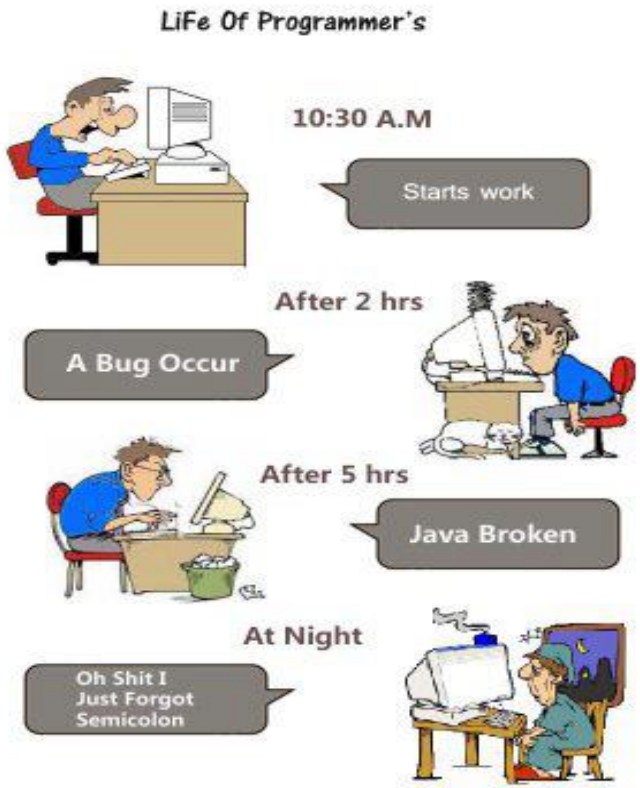
Algoritmi e strutture dati

MergeSort e *QuickSort*



Menú di questa lezione

In questa lezione vedremo gli algoritmi *MergeSort* e *QuickSort*, in diverse versioni, ne dimostreremo la correttezza e ne calcoleremo la complessità.

Algoritmi di ordinamento

Il problema dell'ordinamento di un array è un mattone fondamentale dell'algoritmica. Non solo appare come problema in innumerevoli applicazioni dirette (pensiamo a ordinare nomi anagrafici, valori, prioritá di processi, ecc.), ma appare anche come sotto-problema in una varietá di situazioni. Essendo, comunque, un problema facile da definire e facile da risolvere (almeno quando non facciamo particolare riferimento all'efficienza), è ideale per iniziare lo studio degli algoritmi complessi, e soprattutto le dimostrazioni di correttezza e complessità.

Algoritmi di ordinamento

Senza considerare la grande quantità di casi in cui l'ordinamento è un elemento fondamentale di qualche algoritmo piú complesso (e ne vedremo alcuni), gli usi diretti di un insieme ordinato includono la soluzione di problemi molto variegati, tra cui ricerca di un elemento (come sappiamo, molto piú rapida se effettuata su un insieme ordinato), trovare la coppia di elementi piú vicina (qual è la coppia di elementi a minima distanza in un insieme?), stabilire l'unicità di elementi (stabilire se un elemento è o no unico in un insieme), trovare una distribuzione di frequenze (stabilire, per ogni elemento, quante volte appare), costruire uno shuffling (disordinamento) casuale di un insieme, assieme a molti altri.

Algoritmi di ordinamento elementari

Chiameremo **elementari** quei metodi, basati sui confronti, che condividono la seguente caratteristica: ogni elemento viene spostato verso il/al posto giusto separatamente dagli altri, in maniera iterativa o ricorsiva. Questi metodi includono, tra gli altri, i ben noti *InsertionSort* (che abbiamo giá visto), *SelectionSort*, *BubbleSort*, e moltissimi altri.

Alternative a *InsertionSort*: *SelectionSort*

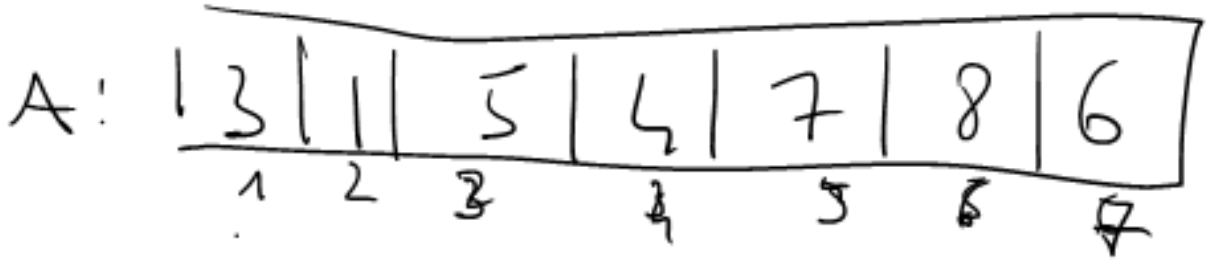
Per esempio l'algoritmo *SelectionSort* è una alternativa (elementare) a *InsertionSort* per l'ordinamento:

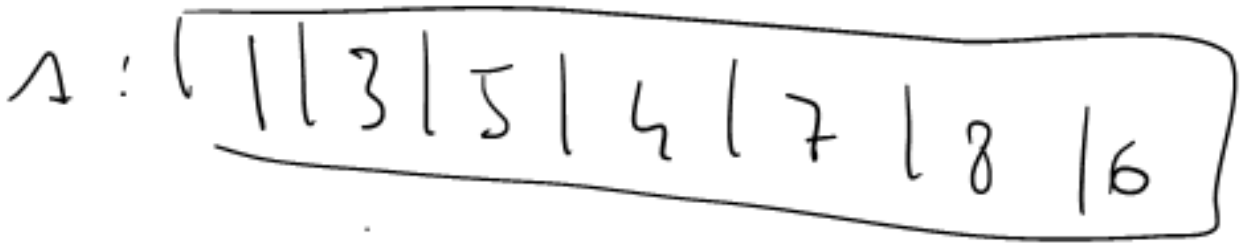
```
proc SelectionSort (A)
  for (j = 1 to A.length − 1)
    min = j
    for (i = j + 1 to A.length)
    if (A[i] < A[min])
      then min = i
  SwapValue(A, min, j)
```

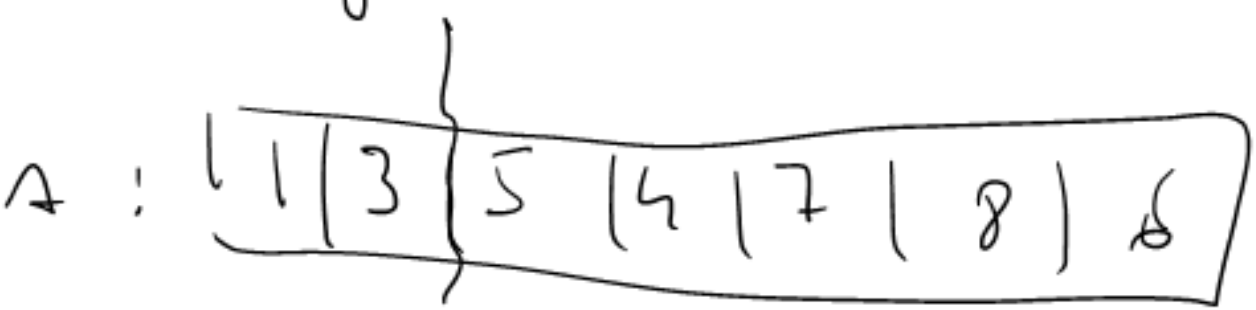
Trova l'indice del minimo tra i rimasti

Nel codice di *SelectionSort* abbiamo usato una procedura, *SwapValue*(), che, dati due indici dell'array da ordinare, ne scambia i contenuti. Questa procedura ha costo costante, e la useremo molto nel seguito. Non studiamo in dettaglio *SelectionSort* e gli altri algoritmi elementari di ordinamento. Dimostrarne la correttezza attraverso lo studio di una invariante, calcolarne la complessitá, e studiarne le proprietá, comunque, è un ottimo esercizio.

ESEMPIO DI ESECUZIONE SELECTION SORT







INV: $A[1..j-1]$ contiene i $j-1$ elementi più piccoli di 1 e ben ordinati.

Conclusioni:

- Non ci sono casi (peggiori, medi, migliori)

- $T(n) = n + (n-1) + (n-2) + (n-3) + \dots + 1$

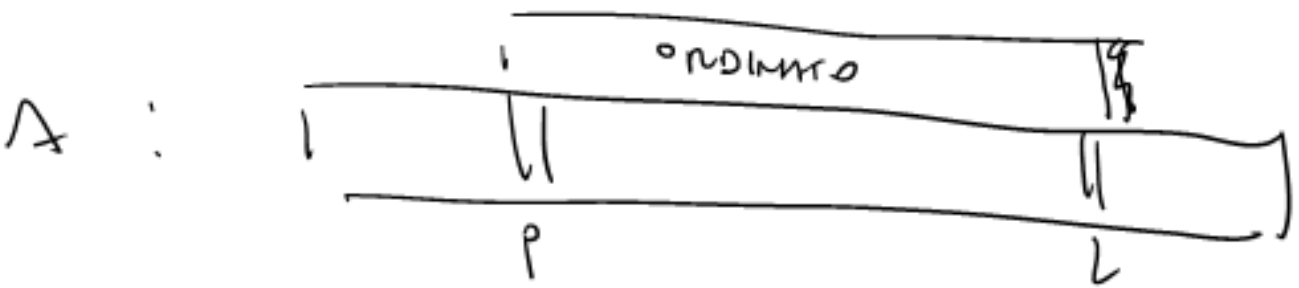
$$= \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Algoritmi di ordinamento non elementari

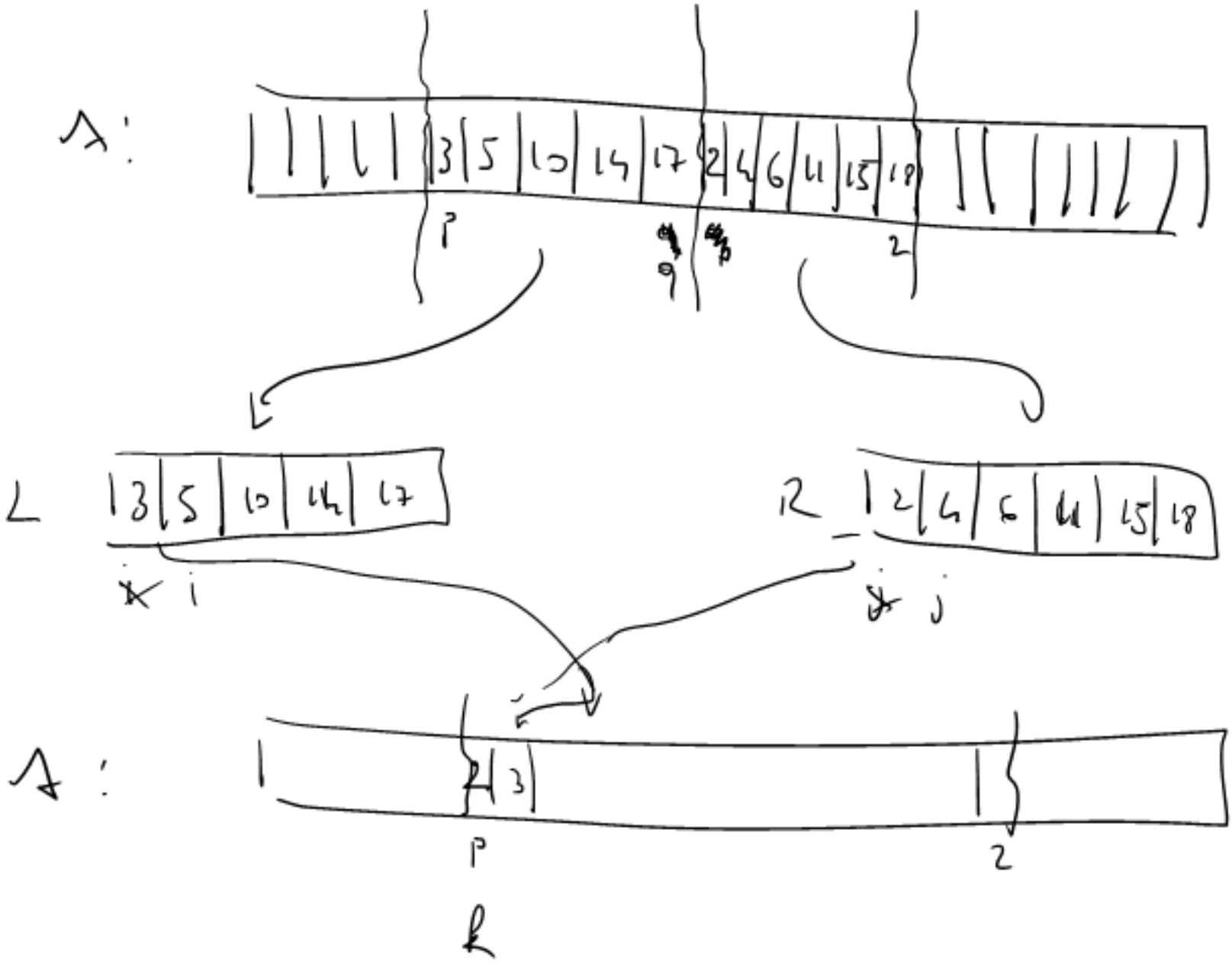
Tipicamente, gli algoritmi di ordinamento non elementari che non fanno uso di ipotesi aggiuntive sono ricorsivi (ma questa non è una regola). Nel nostro programma ne vediamo tre, che ci insegnano tre cose diverse, non solo sull'ordinamento, ma anche sullo studio della complessitá e sui paradigmi di approccio: *MergeSort*, *QuickSort*, e *HeapSort*. In questa lezione, ci concentriamo su *MergeSort*, per la cui introduzione viene accreditato il famoso (per ben altri contributi all'informatica) John Von Neumann, e poi su *QuickSort*.

MergeSort e *Merge*

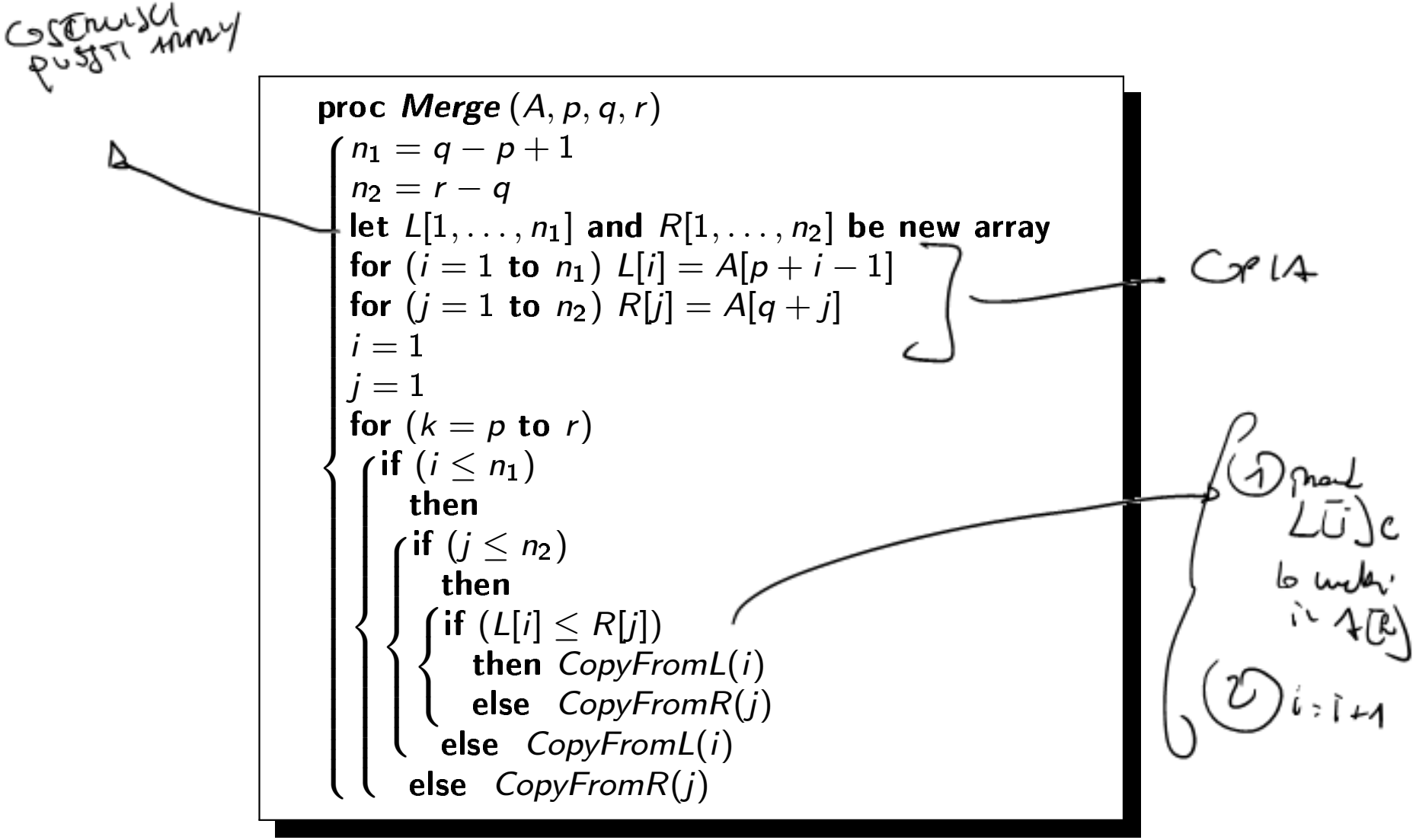
Per la definizione dell'algoritmo *MergeSort*, studiamo prima un algoritmo che risolve un problema più semplice, cioè quello di costruire una sequenza ordinata partendo da due sequenze ordinate. L'input è un array A di interi, e tre indici p, q, r su di esso, tali che $p \leq q < r$ e che entrambi $A[p, \ldots, q]$ e $A[q+1, \ldots, r]$ sono ordinati. Vogliamo, come risultato, una permutazione di A tale che $A[p, \ldots, r]$ è ordinato.



ESEMPIO NUMERICO



MergeSort e Merge



```
proc Merge (A, p, q, r)
  n1 = q − p + 1
  n2 = r − q
  let L[1, . . . , n1] and R[1, . . . , n2] be new array
  for (i = 1 to n1) L[i] = A[p + i − 1]
  for (j = 1 to n2) R[j] = A[q + j]
  i = 1
  j = 1
  for (k = p to r)
    if (i ≤ n1)
      then
      if (j ≤ n2)
        then
        if (L[i] ≤ R[j])
          then CopyFromL(i)
          else CopyFromR(j)
        else CopyFromL(i)
      else CopyFromR(j)
```

MergeSort e *Merge*

L'idea alla base di *Merge* è scegliere il minimo tra i due elementi in cima a L ed a R, e posizionarlo nel giusto posto in A; in seguito, a seconda della scelta, si avanza su L o su R. Utilizziamo due procedure elementari chiamate *CopyFromL* e *CopyFromR* che hanno come effetto quello di copiare in A un elemento (quello il cui indice è dato come parametro) di L (rispettivamente, R) e avanzare di una posizione la variabile indicata come parametro. La scelta di usare oggetti esterni L ed R è molto conveniente per poter usare A come array di arrivo, ma questo significa che, **nella nostra implementazione**, *Merge* non è in place: quanto piú grande è l'input, maggiore è la quantitá di spazio utilizzata da L ed R, che pertanto non è costante. Chiaramente l'algoritmo di ordinamento che andiamo a vedere (*MergeSort*) eredita questa caratteristica, e quindi non è in place.

Correttezza e complessitá di *Merge*

Come sempre, studiamo la **terminazione**. I primi due cicli **for** terminano quando $i = n_1$ e $j = n_2$, rispettivamente; il terzo ciclo **for** è sempre vincolato tra p e r, e per ipotesi $p < r$. Pertanto l'esecuzione di *Merge* sempre termina quando la chiamata rispetta le ipotesi. Per la **correttezza**, la nostra **invariante** è: nel terzo ciclo **for**, $A[p, \ldots, k-1]$ contiene i $k-p$ elementi piú piccoli di $L[1, \ldots, n_1]$ e di $R[1, \ldots, n_2]$, ordinati; inoltre $L[i]$ (se $i \leq n_1$) e $R[j]$ (se $j \leq n_2$) sono i piú piccoli elementi di L ed R non ancora ricopiati in A. Dimostriamo adesso per induzione su k questo fatto. Per assicurare che l'invariante va bene, dobbiamo mostrare che dalla stessa si ottiene la proprietá desiderata sull'output. Ma è proprio così: infatti, quando $k = r + 1$, abbiamo che $A[p, \ldots, k-1] = A[p, \ldots, r]$ contiene i $k - p = r - p + 1$ elementi piú piccoli ed ordinati. Quindi l'algoritmo è corretto.

Correttezza e complessitá di *Merge*

Adesso mostriamo che l'invariante vale.

- **Caso base**: $k = p$. In questo caso $A[p, \ldots, k-1]$ è vuoto. Si verifica quindi che contiene i $k - p = 0$ elementi piú piccoli di L e R, e poichè $i = j = 1$ si ha che $L[i]$ ed $R[j]$ sono gli elementi piú piccoli dei rispettivi oggetti non ancora ricopiati in A.
- **Caso induttivo**: si assume che la proprietá valga per k e si dimostra che vale per $k+1$. All'inizio della iterazione $k+1$, $L[i]$ ed $R[j]$ sono gli elementi piú piccoli dei rispettivi oggetti non ancora ricopiati in A (ipotesi induttiva). Se $L[i] \leq R[j]$ (e $i, j \leq n_1, n_2$), allora $L[i]$ è il piú piccolo elemento non ancora copiato in A. Poichè $A[p, \ldots, k-1]$ sono gli elementi piú piccoli e sono ordinati (ipotesi induttiva) e poichè $L[i]$ si copia in $A[k]$, dopo la $k+1$-esima iterazione $A[p, \ldots, k]$ contiene gli elementi piú piccoli, ordinati, e $L[i]$ è l'elemento piú piccolo di L (giacchè i si è incrementato). Il caso $L[i] > R[j]$, e i casi in cui i o j sono n_1 o n_2, rispettivamente, si dimostrano in maniera simile.

Correttezza e complessitá di *Merge*

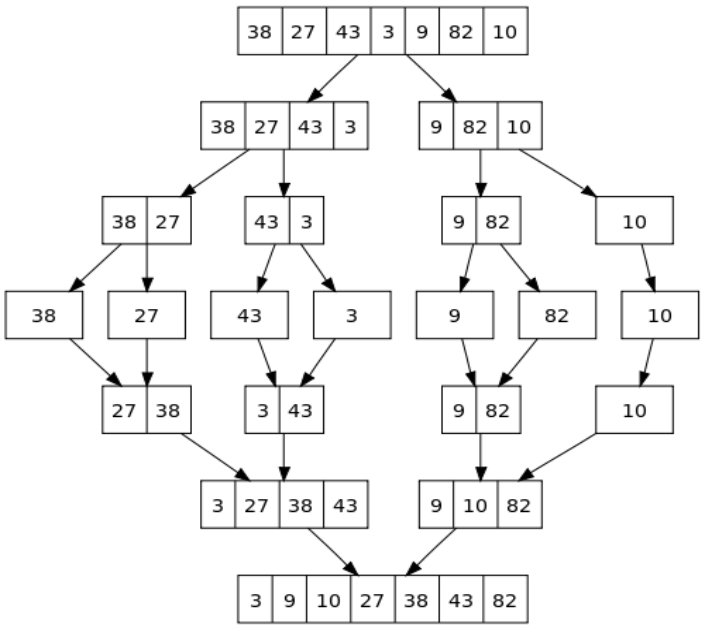
Per completare lo studio di *Merge*, manca ancora da vedere la **complessitá**. I primi due cicli **for** impiegano $\Theta(n_1)$ e $\Theta(n_2)$, ed entrambi sono equivalenti a $\Theta(n)$ (ricordiamo: $n = r - p + 1$). Il terzo ciclo **for** impiega $\Theta(n)$. Poichè $\Theta(n) + \Theta(n) = \Theta(n)$, questa ultima è la complessità di *Merge*. Adesso sappiamo che *Merge* funziona e quanto costa eseguirlo. Si osservi che non ha senso, nel caso di *Merge*, parlare di caso pessimo o caso medio: tutta la complessità dipende da cicli vincolati, e tutti i casi si comportano allo stesso modo.

Ordinamento con *MergeSort*

Usiamo l'algoritmo *Merge* come sotto-procedura per un nuovo algoritmo di ordinamento. L'algoritmo *MergeSort* presenta un codice molto semplice, peró è **ricorsivo**. Questo implica che il calcolo della complessitá porterà a una ricorrenza. La caratteristica di essere ricorsivo spiega anche che nell'input dell'algoritmo appaiano due parametri p, r che sembrano non avere senso, quando il problema dell'ordinamento è definito su tutto l'array di input. Infatti dobbiamo pensare che l'algoritmo **richiama se stesso** durante la computazione; quindi, da un punto di vista dell'utente, dobbiamo solo capire qual'è la giusta chiamata iniziale.

Ordinamento con *MergeSort*

proc ***MergeSort*** (A, p, r)
if $(p < r)$
then
$q = [(p + r)/2]$
$MergeSort(A, p, q)$
$MergeSort(A, q + 1, r)$
$Merge(A, p, q, r)$



La chiamata corretta a *MergeSort* per ordinare A è $MergeSort(A, 1, A.length)$.

Correttezza e complessitá di *MergeSort*

Terminazione: ad ogni chiamata ricorsiva, la distanza tra p e r diminuisce. Si effettua una nuova chiamata solo quando $p < r$, quindi, prima o poi, non si effettueranno più chiamate perchè varrà $p \geq r$, e l'algoritmo termina. Adesso vediamo la **correttezza**, che, come giá sappiamo, si basa su un'invariate ricorsiva (o induttiva). In questo caso, affermiamo che, dopo ogni chiamata ricorsiva $\mathit{MergeSort}(A, p, r)$, $A[p, \ldots, r]$ è ordinato.

Correttezza e complessitá di *MergeSort*

Dobbiamo fare vedere, per induzione, che l'invariante è corretta.

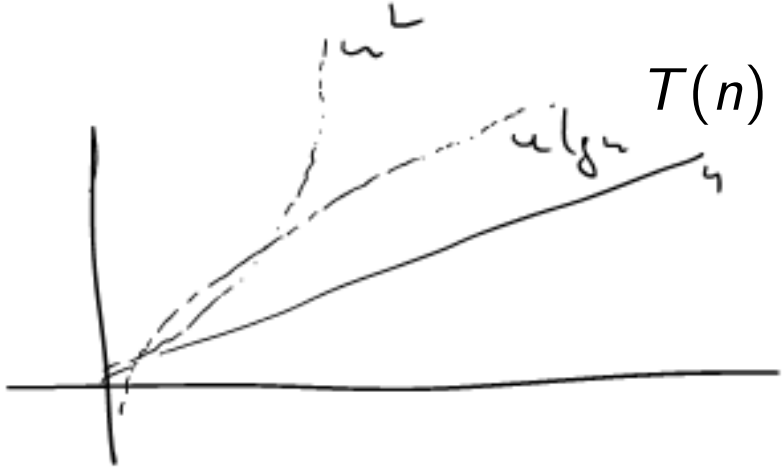
- **Caso base.** Quando $p = r$, non si effettuano chiamate; ma $A[p, \ldots, r]$ contiene una sola posizione, ed è quindi ordinato, come volevamo.
- **Caso induttivo**. Supponiamo quindi che $p < r$, e ragioniamo sulla chiamata $k + 1$-esima. Questa corrisponde a due chiamate a *MergeSort*, entrambe a livello k della ricorsione, una su $A[p, \ldots, \frac{p+r}{2}]$ ed una su $A[\frac{p+r}{2} + 1, \ldots, r]$. Per ipotesi induttiva, entrambe restituiscono un array ordinato. Questi due risultati vengono poi passati a *Merge*, il quale - abbiamo giá dimostrato - correttamente restituisce un array ordinato tra le posizioni p ed r.

Correttezza e complessitá di *MergeSort*

Stabiliamo adesso la **complessitá** di questo algoritmo. Come in *Merge*, non ha senso chiedersi se ci sono degli input che migliorano, o peggiorano le sue prestazioni. *MergeSort* dá luogo alla seguente ricorrenza:

$$T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$$

Seguendo il Master Theorem, questa si esplicita in

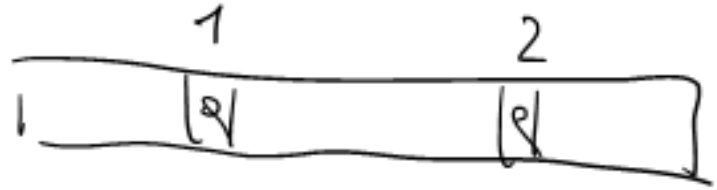


$$T(n) = \Theta(n \cdot log(n)).$$

Correttezza e complessitá di *MergeSort*

La funzione $n \cdot log(n)$ è chiamata **pseudo lineare**, e questo rende l'idea di quanto piú efficiente è rispetto a n^2. É istruttivo provare a concretizzare questa funzione con degli esempi numerici e confrontarla appunto con n^2, cosí come disegnare i rispettivi grafici, per osservare come effettivamente la crescita è molto diversa. Ricordiamo che lo studio delle complessitá ha come scopo la formalizzazione del comportamento degli algoritmi in maniera asintotica, cioè per input crescenti.

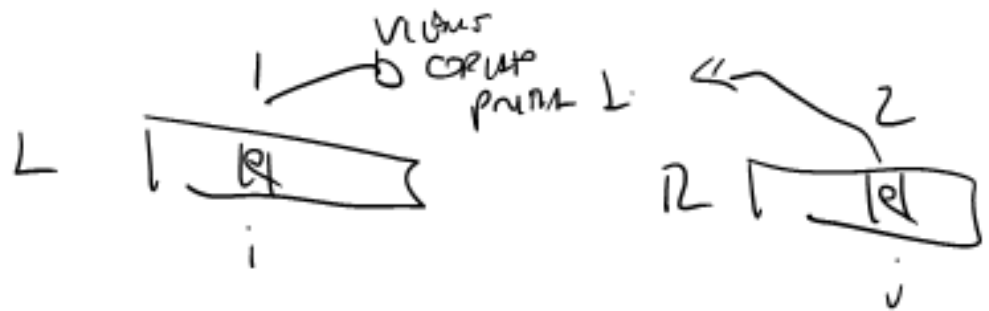
Esercizio 35

A: 

Un algoritmo L' ordinando i valori in

A 

Nel caso L' insertion sort, la ritorna quando



[C]

ESERCIZIO 37

10

$$\max_{[1..n]} A = \max\left\{\max A\left[1..\frac{n}{2}\right], \max A\left[\frac{n}{2}+1, n\right]\right\}$$

```
Proc RecursiveMax(A, i, j)
    if i = j return A[i]
    else
        k = (i+j)/2
        m1 = RecursiveMax(A, i, k)
        m2 = RecursiveMax(A, k+1, j)
        return max{m1, m2}
```

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

M.T.:

$a = 2$

$b = 2$

$f_b(n) = 1$

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

$T(n) = \Theta(n^{\log_b a}) = \Theta(n)$

QuickSort

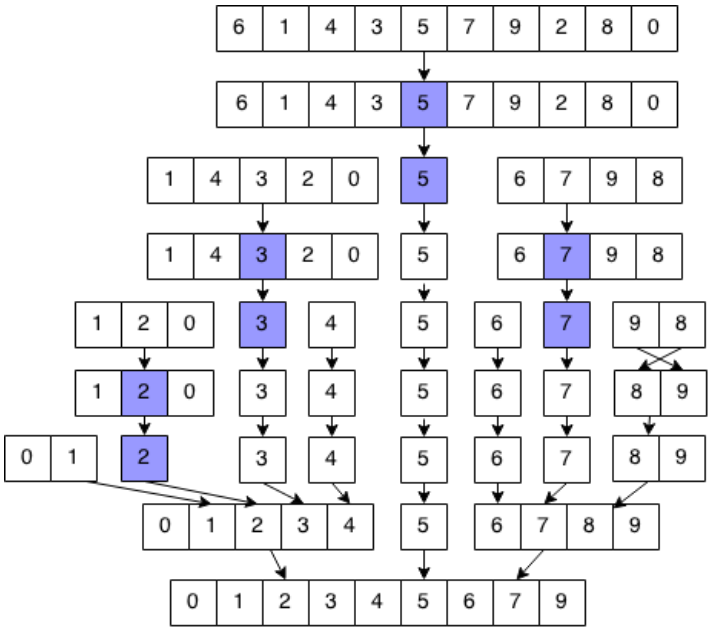
Adesso vediamo un altro algoritmo non elementare di ordinamento. Come *MergeSort*, si tratta di un algoritmo ricorsivo che usa una procedura ausiliare, chiamata *Partition*, che è meritevole di essere studiata per sè, e che permette di formulare diverse soluzioni a problemi che sono considerati **satellite** rispetto all'ordinamento. Anche lo studio della complessità di *QuickSort* è un caso nel quale vediamo dei concetti particolari e meritevoli di nota. *QuickSort*, un altro esempio di strategia divide and conquer, è stato ideato da Tony Hoare.

L'idea alla base di questo algoritmo di ordinamento è procedere ricorsivamente (di nuovo, divide and conquer) nella seguente maniera. Un array A, considerato dalla posizione p alla posizione r, viene prima separato in due sotto-strutture $A[p, \ldots, q-1]$ e $A[q, \ldots, r]$ in maniera che tutti gli elementi **prima** di q siano minori o uguali a $A[q]$, il quale, a sua volta, sia minore o uguale a tutti gli elementi dalla posizione $q+1$ in poi. Quindi, richiamiamo ricorsivamente l'algoritmo su $A[p, \ldots, q-1]$ e $A[q, \ldots, r]$: al ritorno dalla chiamata ricorsiva, entrambi sono ordinati e pertanto non c'è bisogno di alcuna operazione per combinarli. Abbiamo così ordinato $A[p, \ldots, r]$.

QuickSort

```
proc Partition (A, p, r)
  x = A[r]                    Attenzione!
  i = p − 1
  for (j = p to r − 1)
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc QuickSort (A, p, r)
  if (p < r)
    then
      q = Partition(A, p, r)
      QuickSort(A, p, q − 1)
      QuickSort(A, q + 1, r)
```



Correttezza di *QuickSort*

La correttezza di *QuickSort* dipende direttamente dalla correttezza di *Partition*. Per quest'ultimo, si ha che la sua **terminazione** dipende dal fatto che il ciclo principale è un **for**, che noi sappiamo terminare sempre. Per quanto riguarda la **correttezza** usiamo la seguente **invariante**: all'inizio di ogni iterazione, per ogni indice k: se $p \leq k \leq i$, allora $A[k] \leq x$; se $i+1 \leq k \leq j-1$, allora $A[k] > x$, e se $k = r$, allora $A[k] = x$. Come si vede, non possiamo dire nulla sul contenuto di A da j in poi, in quanto non siamo ancora arrivati a quel punto con il ciclo. Alla terminazione, succede che $j = r$; applicando l'invariante, dopo l'ultima istruzione si ottiene un array che, tra le posizioni p ed r, è tale che esiste un valore **pivot** $(i+1)$ e tutti i valori di A prima di $i+1$ sono minori o uguali al valore $A[i+1]$ il quale a sua volta è minore o uguale a tutti i valori dopo la posizione $i+1$.

Correttezza di *QuickSort*

Dimostriamo la validità dell'invariante. L'induzione è sulla variabile j.

- **Caso base**. Osserviamo cosa accade prima della prima esecuzione del ciclo **for**. Si ha che $i = p - 1$ e $j = p$, quindi la parte di A fino alla posizione $j - 1$ (cioè $p - 1$) è vuota, pertanto l'invariante predica trivialmente su un insieme vuoto ed è vera.
- **Caso induttivo**. Prima della j-esima esecuzione, l'elemento j-esimo non è stato ancora visto, e l'invariante predica su tutti gli elementi fino al $j - 1$-esimo compreso. Sul destino del j-esimo elemento decide la condizione $A[j] \leq x$. Supponiamo che sia vera: quindi, l'elemento $A[j]$ va spostato nel primo gruppo. Per fare questo, i avanza di uno e si scambiano le posizioni $i + 1$ e j. Quindi, a fine esecuzione di questo ciclo, il gruppo di sinistra ha un elemento in più (prima condizione dell'invariante), e il gruppo di destra ha lo stesso numero di elementi, ma j è avanzato di una posizione. Se si verificasse che $A[j] > x$, i non si sposterebbe (mantenendo lo stesso numero di elementi a sinistra), ma j si, aumentando il gruppo di destra di un elemento.

Correttezza di *QuickSort*

A questo punto, sappiamo che *Partition* è corretto. La correttezza di *QuickSort* può essere quindi dimostrata banalmente. La sua **terminazione** dipende dal fatto che, come in *MergeSort*, la distanza tra p ed r diminuisce sempre perchè q case sempre in mezzo ai due. Quindi, prima o poi, non ci saranno più chiamate ricorsive e il processo terminerà. Per la **correttezza** abbiamo la seguente **invariante ricorsiva** è: al termine di ogni chiamata ricorsiva con indici p, r, $A[p, \ldots, r]$ è ordinato.

Correttezza di *QuickSort*

Dimostramo la validità dell'invariante:

- Il **caso base** è triviale: quando $p = r$, $A[p, \ldots, r]$ è composto da una sola posizione, ed è quindi ordinato.
- Nel **caso induttivo**, si considerino certi indici $p < r$. La prima operazione consiste nel chiamare *Partition*, che abbiamo mostrato essere corretto, e che restituisce un indice q e una permutazione di $A[p, \ldots, r]$ tale che $A[p, \ldots, q-1]$ contiene solo elementi più piccoli o uguali a $A[q]$, il quale a sua volta è più piccolo di $A[q+1, \ldots, r]$. Dopo le due chiamate ricorsive, per ipotesi induttiva, $A[p, \ldots, q-1]$ e $A[q+1, \ldots, r]$ sono entrambi ordinati. Ma allora $A[p, \ldots, r]$ è ordinato, come volevamo dimostrare.

Complessitá di *QuickSort*

La **complessitá** di *Partition* è facile da calcolare: $\Theta(n)$. Il calcolo della complessitá di *QuickSort*, d'altra parte, presenta molti problemi, come vedremo. Per cominciare, andiamo ad analizzare il caso peggiore. Il primo problema che ci troviamo di fronte è proprio stabilire qual'è la situazione che ci porta al caso peggiore: lo denominiamo **partizione sbilanciata**. Una partizione sbilanciata si ottiene quando la scelta del pivot è tale che, ad ogni passo, genera un sottoproblema di dimensione 0 ed un sottoproblema di dimensione $n-1$. In questo caso la ricorrenza che si ottiene è:

$$T(n) = T(n-1) + \Theta(n),$$

la cui soluzione, vista come sviluppo di una serie aritmetica, è evidentemente $T(n) = \Theta(n^2)$. Quindi, stiamo considerando un algoritmo che, secondo l'analisi che abbiamo fatto finora, si comporta in maniera peggiore di *MergeSort*, e, inoltre, è tale che il suo caso peggiore si verifica precisamente quando l'array di partenza è giá ordinato in partenza: proprio una situazione nella quale addirittura *InsertionSort* si comporta meglio ($\Theta(n)$).

Complessitá di *QuickSort*

D'altra parte, il caso migliore si verifica quando la partizione è sempre bilanciata nel modo piú equo possibile. In questo caso la ricorrenza che si ottiene è:

$$T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n),$$

la cui soluzione, per il Master Theorem, è $\Theta(n \cdot log(n))$. Cosa accade quando la partizione è bilanciata in modo molto iniquo? Per esempio assumendo che si creino sempre due sottoproblemi di dimensione relativa 9 a 1?

Complessitá di *QuickSort*

Anche questo è un caso particolare. La ricorrenza risultante è

$$T(n) \leq T(\frac{9 \cdot n}{10}) + T(\frac{n}{10}) + \Theta(n),$$

la cui soluzione si puó mostrare essere $T(n) = O(n \cdot log(n))$. Questi esempi suggeriscono che quando la partizione è costante (e nessuno dei due sottoproblemi è di dimensione 0) allora non è mai sbilanciata. Rimane peró il fatto che non c'è garanzia alcuna che la partizione non sia sbilanciata.

Complessitá di *QuickSort*

La considerazione precedente si puó portare all'estremo, osservando che **per ogni** scelta del pivot (quindi, per ogni versione si *Partition*) **esiste** un input tale che porterá la corrispondente versione di *QuickSort* al caso peggiore. Possiamo pensare a questo problema in termini di gioco: per ogni mossa del giocatore **buono** (chi costruisce *Partition*) esiste una mossa del giocatore **cattivo** (chi costruisce l'input) in maniera da generare il caso peggiore. Ma per un dato input ed una data versione di *Partition*, qual è la probabilitá che questo succeda?

Complessitá di *QuickSort*

Mostreremo che la complessitá del caso medio di *QuickSort* è $O(n \cdot log(n))$, sotto l'ipotesi che tutti gli input siano equamente probabili. In particolare, peró, possiamo **garantire** che tutti gli input siano equamente probabili attraverso una versione **randomizzata** dell'algoritmo. Otteniamo due vantaggi: primo, non ci possono essere input **volutamente** pessimi (possono ancora esistere, ma con una bassa probabilitá), e, secondo, l'assunzione "tutti gli input sono equamente probabili" è rispettata, il che rende leggermente piú semplice il calcolo.

La variante *RandomizedQuickSort* e la sua complessitá

```
proc RandomizedPartition(A, p, r)
  s = p ≤ Random() ≤ r
  SwapValue(A, s, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
    if (A[j] ≤ x)
      then
        i = i + 1
        SwapValue(A, i, j)
  SwapValue(A, i + 1, r)
  return i + 1
```

```
proc RandomizedQuickSort(A, p, r)
  if (p < r)
    then
      q = RandomizedPartition(A, p, r)
      RandomizedQuickSort(A, p, q − 1)
      RandomizedQuickSort(A, q + 1, r)
```

La variante *RandomizedQuickSort* e la sua complessitá

La scelta casuale del pivot implica che ad ogni passo tutte le partizioni sono igualmente probabili. Questo significa che possiamo parlare della complessitá di *RandomizedQuickSort* nel caso medio in termini di probabilitá di una determinata partizione. Guardiamo cosa succede quando effettuiamo una partizione qualsiasi. Supponiamo che, dopo aver eseguito *Partition*, la situazione sia la seguente:

$$\underbrace{abc\ldots}_{k}\,P\,\underbrace{\ldots abcabcabc}_{(n-1)-k}$$

La posizione di P (il pivot) determina su quanti elementi *QuickSort* si richiamerá. Detta k la cardinalità del lato sinistro, la cardinalità del lato destro sará $n-1-k$, considerato che il pivot non viene piú toccato.

La variante *RandomizedQuickSort* e la sua complessitá

Se tutte le partizioni sono ugualmente probabili, la probabilità di una specifica partizione è proprio $\frac{1}{n}$. Formalizzando, si ottiene che

$$T(n) = \frac{1}{n} \cdot \Sigma_{k=0}^{n-1}(T(k) + T(n-k-1)) + \Theta(n),$$

e quindi:

$$T(n) = \frac{2}{n} \cdot \Sigma_{k=0}^{n-1} T(k) + \Theta(n)$$

Dimostreremo che $T(n) = O(n \cdot log(n))$ attraverso la tecnica della sostituzione. In particolare, mostreremo che $T(n) \leq a \cdot n \cdot log(n) + b$ per qualche costante a, b.

La variante *RandomizedQuickSort* e la sua complessitá

Procediamo. In prima istanza semplifichiamo la nostra espressione:

$$
\begin{aligned}
T(n) &= \tfrac{2}{n} \cdot \textstyle\sum_{k=0}^{n-1} T(k) + \Theta(n) && \text{ricorrenza} \\
&\leq \tfrac{2}{n} \cdot \textstyle\sum_{k=1}^{n-1} (a \cdot k \cdot log(k) + b) + \tfrac{2 \cdot b}{n} + \Theta(n) && \text{ip. induttiva} \\
&= \tfrac{2}{n} \cdot \textstyle\sum_{k=1}^{n-1} (a \cdot k \cdot log(k) + b) + \Theta(n) && k = 0 \text{ messo in } \Theta(n) \\
&= \tfrac{2}{n} \cdot \textstyle\sum_{k=1}^{n-1} a \cdot k \cdot log(k) + \tfrac{2}{n} \cdot \textstyle\sum_{k=1}^{n-1} b + \Theta(n) && \text{somma distribuita} \\
&= \tfrac{2 \cdot a}{n} \cdot \textstyle\sum_{k=1}^{n-1} k \cdot log(k) + \tfrac{2 \cdot b}{n} \cdot (n-1) + \Theta(n) && 2^a \text{ somma valutata} \\
&\leq \tfrac{2 \cdot a}{n} \cdot \textstyle\sum_{k=1}^{n-1} k \cdot log(k) + 2 \cdot b + \Theta(n) && \tfrac{n-1}{n} < 1
\end{aligned}
$$

La variante *RandomizedQuickSort* e la sua complessitá

Secondariamente, osserviamo che $\Sigma_{k=1}^{n} k \cdot log(k)$ puó essere limitata verso l'alto, come segue:

$$
\begin{aligned}
\Sigma_{k=1}^{n-1}(k \cdot log(k)) &= \Sigma_{k=1}^{\lceil \frac{n}{2} \rceil - 1}(k \cdot log(k)) + \Sigma_{k=\lceil \frac{n}{2} \rceil}^{n-1}(k \cdot log(k)) && \text{spezzo} \\
&\leq \Sigma_{k=1}^{\lceil \frac{n}{2} \rceil - 1}(k \cdot log(\tfrac{n}{2})) + \Sigma_{k=\lceil \frac{n}{2} \rceil}^{n-1}(k \cdot log(n)) && \text{maggioro} \\
&= (log(n) - 1) \cdot \Sigma_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + log(n) \cdot \Sigma_{k=\lceil \frac{n}{2} \rceil}^{n-1} k && \text{estraggo} \\
&= log(n) \cdot \Sigma_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k - \Sigma_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + log(n) \cdot \Sigma_{k=\lceil \frac{n}{2} \rceil}^{n-1} k && \text{distrib.} \\
&= log(n) \cdot \Sigma_{k=1}^{n-1} k - \Sigma_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k && \text{sommo} \\
&\leq \tfrac{1}{2} \cdot log(n) \cdot n \cdot (n-1) - \tfrac{1}{2} \cdot \tfrac{n}{2} \cdot (\tfrac{n}{2} - 1) && \text{valuto} \\
&= \tfrac{1}{2} \cdot (n^2 \cdot log(n) - n \cdot log(n)) - \tfrac{1}{8} \cdot n^2 + \tfrac{1}{4} \cdot n) && \text{valuto} \\
&\leq \tfrac{1}{2} \cdot n^2 \cdot log(n) - \tfrac{1}{8} \cdot n^2. && n \geq 2
\end{aligned}
$$

La variante *RandomizedQuickSort* e la sua complessitá

Finalmente, usiamo questo risultato per completare il caso induttivo:

$$\begin{aligned} T(n) &\leq \frac{2\cdot a}{n} \cdot \left(\frac{1}{2} \cdot \left(n^2 \cdot log(n) - \frac{1}{8} \cdot n^2\right) + 2 \cdot b + \Theta(n)\right) \quad \text{dalla precedente} \\ &= a \cdot n \cdot log(n) - \frac{a \cdot n}{4} + 2 \cdot b + \Theta(n) \\ &\leq a \cdot n \cdot log(n) + b, \end{aligned}$$

Quindi il tempo del caso medio di *RandomizedQuickSort* è precisamente $O(n \cdot log(n))$ e abbiamo escluso la possibilitá di poter **scegliere** un'istanza di input al fine di ottenere il caso peggiore!

Conclusione

MergeSort è un esempio di strategia **divide and conquer** ("dividi e vincerai"). Questa strategia si basa sull'idea di dividere il problema in due o piú sotto-problemi separati tra loro, e poi combinare i risultati di questi ultimi. Si contrappone a strategie piú semplici, come quelle iterative (esempio: *InsertionSort*), piú complesse, come la **programmazione dinamica**, di cui vedremo esempi piú avanti, o semplicemente diverse, come la strategia **greedy**, di cui, anche, vedremo degli esempi. Allo stesso modo, *QuickSort*, *Partition*, e *RandomizedQuickSort* racchiudono numerosi concetti che possono essere considerati archeotipici di molte idee algoritmiche. É dunque necessario conoscerli bene, ed implementarli per verificare queste conoscenze nella pratica.