

Informatica per Statistica

Riassunto della lezione dell'11/10/2013

Igor Melatti

Analisi della correttezza dell'Insertion Sort

- Si riconsideri l'algoritmo *Insertion Sort* (Figura 1)

```
1 Insertion-Sort(A)
2   for  $j \leftarrow 2$  to length(A) do
3     key  $\leftarrow A[j]$ 
4     /* Inserimento di  $A[j]$  nella sequenza ordinata  $A[1..j-1]$  */
5      $i \leftarrow j - 1$ 
6     while  $i > 0$  and  $A[i] > key$  do
7        $A[i + 1] \leftarrow A[i]$ 
8        $i \leftarrow i - 1$ 
9      $A[i + 1] \leftarrow key$ 
```

Figure 1: Algoritmo *Insertion Sort*

- Volendolo descrivere matematicamente a più alto livello, ovvero volendo ragionare in modo da riscriverlo da capo con cognizione di causa, lo si potrebbe riassumere così
 - sia n il numero di elementi della sequenza A da ordinare (quindi $n = \text{length}(A)$)
 - si supponga che, per un certo $1 \leq j \leq n$, gli elementi da $A[1]$ a $A[j-1]$ siano già ordinati
 - * per $j = 1$ quanto appena detto è banalmente vero, perché la sequenza da $A[1]$ a $A[0]$ è vuota, e tutte le sequenze vuote sono già ordinate
 - * anche per $j = 2$ quanto appena detto è banalmente vero, perché la sequenza da $A[1]$ a $A[1]$ contiene un solo elemento ($A[1]$, per l'appunto), e tutte le sequenze di un solo elemento sono già ordinate

- * quindi si prende come punto di partenza $j = 2$, e le assunzioni di base sono valide
- se si riesce ad inserire $A[j]$ all'interno della sequenza *ordinata* $A[1 \dots j-1]$ in modo tale da ottenere una sequenza ordinata $A[1 \dots j]$, si potrà ripetere il procedimento con il j successivo, dato che le assunzioni di partenza saranno ancora valide
 - * ovvero, si vorrà inserire $A[j + 1]$ nella sequenza già ordinata $A[1 \dots j]$
 - * fare tutto ciò equivale a definire un ciclo in cui c'è una proprietà che vale all'inizio di ogni esecuzione (*iterazione*) del ciclo stesso
 - * tale proprietà viene chiamata *invariante*
 - * in questo caso, si avrà che il ciclo `for` di riga 2 ha come invariante il fatto che, per l'attuale valore di $j \in \{1, \dots, n\}$, gli elementi da $A[1]$ a $A[j - 1]$ sono già ordinati
 - * infatti tale `for` parte da $j = 2$, che come detto sopra rende banalmente vera l'invariante
 - * il corpo del (cioè il blocco di istruzioni all'interno del) `for` è fatto in modo da inserire correttamente (come si vedrà nel seguito) $A[j]$ all'interno della sequenza ordinata $A[1 \dots j - 1]$
 - * quindi all'iterazione successiva l'invariante sarà ancora valida
 - * per induzione (sul numero di iterazioni del `for`) l'algoritmo alla fine sarà corretto
- per inserire correttamente $A[j]$ all'interno della sequenza ordinata $A[1 \dots j - 1]$ si procede come segue
- si scorre la sequenza $A[1 \dots j - 1]$ fino a trovare il posto i dove inserire $A[j]$
- ovvero si trova il più piccolo indice i t.c. $A[i] \geq A[j]$
- ricordando che nella sequenza ordinata $A[1 \dots j - 1]$ vale che ogni elemento è minore uguale del successivo (in formule: $\forall k \in [1, j - 2]$ si ha che $A[k] \leq A[k + 1]$), il che implica che un qualsiasi elemento è maggiore uguale di tutti quelli che lo precedono (in formule: $\forall h, k \in [1, j - 1]$ si ha che $h < k$ implica $A[h] \leq A[k]$)...
- ...si può concludere che i è il posto giusto perché da una parte $\forall k \in [1, i - 1]$ si ha che $A[k] < A[j]$ (perché i è il più piccolo indice...), e dall'altra $\forall k \in [i, j - 1]$ si ha che $A[k] \geq A[j]$ (in quanto $A[k] \geq A[i] \geq A[j]$)
- quindi se si spostano di una posizione a destra tutti gli elementi da i a $j - 1$ (che quindi si verranno a trovare nelle posizioni da $i + 1$ a j), e poi si mette $A[j]$ al posto di $A[i]$, ci si è correttamente ricondotti al caso base

- * ovvero, la sequenza $A[1..j]$ è ordinata e si può passare all'iterazione successiva, per inserire l'elemento $A[j + 1]$
 - * detta così può sembrare complicato (perché si usano termini matematici...), ma è come quando si ordinano in mano le carte da gioco
 - * di volta in volta si hanno le prime $j - 1$ a sinistra già ordinate, e si mette la j al suo posto *inserendola* tra quelle già presenti
 - * questo inserimento è facile se fatto a mano; algoritmicamente occorre badare ad alcuni dettagli spiegati qui di seguito
- nel realizzare tutto ciò occorre stare attenti a due dettagli importanti, che potrebbero pregiudicare la correttezza o l'efficienza dell'algoritmo
1. fare attenzione alla fase di spostamento (*shift*) a destra degli elementi della sequenza e di inserimento di $A[j]$: fatto in modo ingenuo cancellerebbe $A[j]$ o $A[i]$
 - * se come prima cosa si fa lo spostamento, allora $A[j - 1]$ sovrascrive $A[j]$, il cui valore viene quindi perso
 - * quindi, quando poi si mette $A[j]$ al posto di $A[i]$, in realtà ci si sta mettendo quello che prima dello spostamento era $A[j - 1]$, che ora sarà quindi presente 2 volte nella sequenza (oltretutto potenzialmente fuori ordine)...
 - * ad esempio, se la sequenza è $\langle 1\ 4\ 7\ 9\ 6 \rangle$, e si vuole inserire 6 (che si trova all'indice $j = 5$) all'indice 3, allora con lo spostamento si ottiene $\langle 1\ 4\ 7\ 7\ 9 \rangle$, dopodiché, con l'assegnamento di $A[j] = A[5] = 9$ ad $A[3]$, si ha $\langle 1\ 4\ 9\ 7\ 9 \rangle$
 - tale sequenza non solo non più ordinata, ma rispetto a quella in input è sparito il numero 6, e il 9 compare 2 volte
 - * allora si potrebbe pensare di fare prima l'assegnamento e poi lo spostamento, ma ancora non funziona, perché l'assegnamento cancella in modo irrecuperabile $A[i]$
 - * nell'esempio di sopra, si avrebbe $\langle 1\ 4\ 6\ 9\ 6 \rangle$ e poi $\langle 1\ 4\ 6\ 6\ 9 \rangle$, che è ordinato, ma ci si è perso il 7, ovvero $A[i]$
 - * la cosa giusta da fare è salvarsi uno dei due valori in una variabile a parte, per poi ripristinarne il valore
 - * si potrebbe salvare $A[i]$ in una variabile *key*, fare l'assegnamento $A[i] \leftarrow A[j]$, fare lo spostamento, e poi assegnare $A[i + 1] \leftarrow key$
 - **Esercizio:** dopo l'assegnamento $A[i] \leftarrow A[j]$ e lo spostamento dei valori dell'array, ma *prima* di assegnare $A[i + 1] \leftarrow key$, quante volte è presente nella sequenza il numero $A[j]$, e in quali posizioni?

- * oppure si potrebbe salvare $A[j]$ in una variabile key , fare lo spostamento, e poi assegnare $A[i] \leftarrow key$
 - * già così si vede che quest'ultima è la soluzione migliore (si fanno meno assegnamenti); un ulteriore motivo è dato dal punto successivo
2. come si ricerca l'elemento i detto sopra (se fatto male, l'algoritmo è meno efficiente)
- * si potrebbe pensare di farlo come segue
 - * per i che va da 1 a $j - 1$, si vede se $A[i] \geq A[j]$ oppure no
 - * se è falso, ovvero se $A[i] < A[j]$, si procede con l' i successivo
 - **Esercizio:** se alla riga 6 di Figura 2 si fosse scritto key al posto di $A[j]$, l'algoritmo sarebbe stato ancora corretto?
 - **Esercizio:** la riga 3 di Figura 2 potrebbe essere spostata più avanti: fino a prima di quale istruzione, al massimo?
 - * se è vero, allora l'indice i con le caratteristiche cercate è stato trovato (per costruzione, è il più piccolo, altrimenti questo piccolo ciclo su i sarebbe finito prima)
 - * il problema è che ora occorre fare un altro ciclo che effettua lo spostamento a destra (Figura 2)

```

1 Insertion-Sort-Ter(A)
2   for  $j \leftarrow 2$  to length(A) do
3      $key \leftarrow A[j]$ 
4     /* ricerca di  $i$  */
5      $i \leftarrow 1$ 
6     while  $i \leq j - 1$  and  $A[i] < A[j]$  do
7        $i \leftarrow i + 1$ 
8     /* Inserimento di  $A[j]$  nella sequenza ordinata  $A[1 \dots j - 1]$  */
9      $k \leftarrow i$ 
10    while  $k \leq j - 1$  do
11       $A[k + 1] \leftarrow A[k]$ 
12     $A[i] \leftarrow key$ 

```

Figure 2: Algoritmo *Insertion Sort Ter*

- * invece, il tutto può essere organizzato di modo tale che la ricerca di i e lo spostamento avvengano in un unico ciclo
- * basta far sì che i vada a ritroso da $j - 1$ a 1 (nota: scrivere $i > 0$ e scrivere $i \geq 1$ è lo stesso, essendo i intero)
- * in questo modo, contemporaneamente si cerca i e si spostano gli elementi; una volta che i è stato trovato, basta uscire dal ciclo e copiare key nel posto giusto

- * anche qui occorre fare un po' di attenzione: la condizione di ricerca per i va invertita e questo ha qualche conseguenza...
 - * è evidente che *Insertion Sort* (Figura 1) è scritto meglio di *Insertion Sort Ter* (Figura 2)
 - * la complessità *asintotica* dei due algoritmi è in realtà la stessa, ma nello scrivere un algoritmo si cerca sempre di farlo il più corto possibile e di usare meno risorse possibili (in *Insertion Sort* non serve l'ulteriore variabile k)
- Da notare che l'*Insertion Sort* effettua un ordinamento *in loco*, ovvero non ha bisogno di un altro array (di appoggio) oltre ad A
 - ovvero, l'ordinamento avviene spostando elementi all'interno di A
 - ad ogni istante, solo un numero *costante* di elementi è memorizzato fuori da A
 - per essere ancora più precisi, questo numero costante è 1: si tratta della variabile *key*

Notazioni per la complessità degli algoritmi: definizioni

- La complessità di un algoritmo viene solitamente espressa come funzione della dimensione dell'input
 - la dimensione dell'input va individuata sulla base del problema che si sta risolvendo
 - tipicamente, se si tratta di effettuare computazioni su una sequenza, la dimensione dell'input è la lunghezza della sequenza stessa
 - si è interessati al *comportamento asintotico* dell'algoritmo: come si comporta al crescere della dimensione dell'input
 - ovverosia, quanto tempo occorre aspettare per una data dimensione dell'input (*complessità temporale*)
- La dimensione dell'input è sempre un intero non-negativo, mentre il tempo di attesa risultante è un reale positivo
- Si arriva così a considerare la complessità di un algoritmo come una funzione che ha come dominio \mathbb{N} , e come codominio \mathbb{R}^+
 - verrà indicata come $T : \mathbb{N} \rightarrow \mathbb{R}^+$, dove $\mathbb{R}^+ = \{x \mid x \in \mathbb{R} \text{ e } x > 0\}$
 - $T(n)$ è la complessità per input di dimensioni n
 - nel seguito si sottintende che l'argomento delle funzioni è n

- Detta così non ha molto senso, dato che può accadere che lo stesso algoritmo, per input aventi la stessa dimensione, si comporti in modo molto diverso
 - per esempio, l'*Insertion Sort* è molto più veloce quando il vettore è già ordinato
 - quindi lo stesso algoritmo (*Insertion Sort*) si comporta diversamente per input aventi la stessa dimensione (ad esempio, $\langle 1, 3, 5, 6, 10 \rangle$ e $\langle 15, 10, 25, 30, 1 \rangle$)
 - **Esercizio:** nel caso dei due input dati sopra dire, per ogni iterazione del ciclo `for` di *Insertion Sort*, quante istruzioni volte viene eseguito il controllo della condizione del `while`
 - insomma, a voler essere precisi, la complessità di un algoritmo non dipende solo da quanto è lungo il suo input, ma anche da qual è in effetti l'istanza in input
 - però considerare tutti le possibili istanze non è possibile, e invece si vuole poter ragionare solo sulla dimensione dell'input
- Occorre infatti specificare se si sta facendo riferimento al *caso peggiore*, al *caso migliore*, o al *caso medio*
 - caso peggiore: su nessun altro input della stessa dimensione l'algoritmo può impiegarci di più (al massimo lo stesso tempo)
 - caso migliore: su nessun altro input della stessa dimensione l'algoritmo può impiegarci di meno (come minimo lo stesso tempo)
 - caso medio: statisticamente è l'input che si presenta più di frequente
- Dato che tipicamente si vuole una *garanzia* che l'algoritmo dia una risposta entro un certo periodo di tempo, si sceglie il caso peggiore
- Quindi $T(n)$ viene definito come il tempo necessario all'algoritmo per completare la computazione su un input di dimensione n nel caso peggiore
- Anche messa così è difficile dare la formula (esplicita) precisa per $T(n)$, e in realtà, essendo l'interesse focalizzato sul comportamento asintotico, non è neanche interessante
- È sufficiente dare una sorta di ordine di grandezza
- Per questo ci sono le notazioni O , Ω , Θ
- $O(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0. f(n) \leq cg(n)\}$
 - quindi $O(g(n))$ contiene tutte le funzioni che sono asintoticamente (ovvero, per n che tende ad infinito) limitate superiormente da $g(n)$ stessa (ovvero, $g(n)$ le “supera” sull'asse delle ordinate)

- a parziale correzione di quanto appena detto, si permette che $g(n)$ sia moltiplicata per una costante positiva reale $c > 0$
 - si permette anche che $cg(n)$ venga “superata” da qualche funzione dentro $O(g(n))$, ma solo per un numero *finito* di valori di n
 - da un certo punto (n_0) in poi (quindi per ogni $n \geq n_0$), è sempre più grande $cg(n)$
- $\Omega(g(n)) = \{f(n) \mid g(n) \in O(f(n))\}$
 - ovvero $\Omega(g(n)) = \{f(n) \mid \exists c \in R^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0. cg(n) \leq f(n)\}$
 - vale il discorso detto prima al contrario: stavolta nell’insieme ci sono tutte le funzioni che “superano” $g(n)$, anche se si moltiplica g per una costante $c < 0$
 - la relazione appena data può anche non valere, ma solo per un numero finito di valori di n
 - da un certo punto (n_0) in poi (quindi per ogni $n \geq n_0$), è sempre più piccola $cg(n)$
 - $\Theta(g(n)) = \{f(n) \mid g(n) \in O(f(n)) \text{ e } f(n) \in O(g(n))\}$
 - ovvero $\Theta(g(n)) = \{f(n) \mid g(n) \in O(f(n)) \text{ e } g(n) \in \Omega(f(n))\}$
 - ovvero $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 \in R^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0. c_1g(n) \leq f(n) \leq c_2g(n)\}$
 - $\Theta(g(n))$ contiene tutte le funzioni che all’infinito si comportano come $g(n)$, tranne al più delle costanti
 - È comune abusare di queste notazioni in svariati modi; in particolare, anziché scrivere $f(n) \in O(g(n))$, si scrive $f(n) = O(g(n))$
 - da intendersi: $f(n)$ è una *qualunque* funzione limitata superiormente da $g(n)$ (con le restrizioni dette sopra, quindi limite asintotico e a meno di una costante)
 - Proprietà importanti, ma ovvie dalle definizioni di sopra:
 - se $f(n) = \Theta(g(n))$ e $g(n) = \Theta(h(n))$, allora $f(n) = \Theta(h(n))$ (proprietà *transitiva*)
 - lo stesso vale con O e Ω al posto di Θ
 - $f(n) = \Theta(f(n))$ (proprietà *riflessiva*)
 - lo stesso vale con O e Ω al posto di Θ
 - se $f(n) = \Theta(g(n))$ allora $g(n) = \Theta(f(n))$ (proprietà *simmetrica*)
 - lo stesso *non* vale con O e Ω al posto di Θ ; per O e Ω valgono invece le seguenti

- se $f(n) = \Omega(g(n))$ allora $g(n) = O(f(n))$ e viceversa (proprietà *simmetrica trasposta*)
- in effetti, c'è una semplice analogia con le relazioni d'ordine sui reali, derivate direttamente dalle definizioni: O è come il \leq , Ω è come il \geq , Θ è come l'= $$
- ovvero, dire $f(n) = \Omega(g(n))$ è all'incirca come dire $f \geq g$, dire $f(n) = O(g(n))$ è all'incirca come dire $f \leq g$, dire $f(n) = \Theta(g(n))$ è all'incirca come dire $f = g$

- Esempi:

- $n^k = O(n^m)$ se $0 \leq k \leq m$
- $\sum_{i=0}^{k-1} a_i n^i = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 = O(n^k)$
- $O(1) = O(k)$ se k è costante rispetto alla dimensione dell'input; quindi per indicare tempi di esecuzione costanti si dirà $O(1)$ (sarebbe lo stesso dire $\Theta(1)$...)
- $n^k = O(a^n)$ per $a > 1$; $n^k = \Omega(a^n)$ per $a \leq 1$ (per $a < 1$, a^n tende a 0 anziché all'infinito...)
- $\log_2 n = O(n)$
- $n = O(n \log_2 n)$
- $a^n = O(n!)$ qualsiasi sia a (ma per $a \leq 1$ è ovvio)
- $n! = O(n^n)$

- Dire che un algoritmo ha una complessità nel caso peggiore $T(n) = O(g(n))$ è come dire che, per nessun input di dimensione n , quell'algoritmo richiederà più di $g(n)$ (sempre a meno di costanti e per n sufficientemente grande)
- Quindi c'è una sorta di garanzia sul tempo di esecuzione
- Dire che un algoritmo ha una complessità nel caso migliore $T(n) = \Omega(g(n))$ è come dire che, per nessun input di dimensione n , quell'algoritmo richiederà meno di $g(n)$ (sempre a meno di costanti e per n sufficientemente grande)
- Quindi un algoritmo avrà tipicamente una complessità compresa tra $\Omega(f(n))$ (caso migliore) e $O(g(n))$ (caso peggiore)
 - non meno dell'uno e non più dell'altro, qualsiasi sia l'input

| | | | |
|---|--|--------------------|--------------------------|
| 1 | <code>Insertion-Sort(A)</code> | <code>costo</code> | <code>num. volte</code> |
| 2 | <code>for j ← 2 to length(A) do</code> | c_1 | n |
| 3 | <code>key ← A[j]</code> | c_2 | $n - 1$ |
| 4 | <code>i ← j - 1</code> | c_3 | $n - 1$ |
| 5 | <code>while i > 0 and A[i] > key do</code> | c_4 | $\sum_{j=2}^n t_j$ |
| 6 | <code>A[i + 1] ← A[i]</code> | c_5 | $\sum_{j=2}^n (t_j - 1)$ |
| 7 | <code>i ← i - 1</code> | c_6 | $\sum_{j=2}^n (t_j - 1)$ |
| 8 | <code>A[i + 1] ← key</code> | c_7 | $n - 1$ |

Figure 3: Analisi della complessità dell'*Insertion Sort*

Analisi della complessità dell'*Insertion Sort*

- Per l'*Insertion Sort*, si ha $f(n) = n$ (caso migliore) e $g(n) = n^2$ (caso peggiore)
- Per provare che le complessità migliore e peggiore sono quelle date, si può procedere come in Figura 3 (con $n = \text{length}(A)$)
 - si suppone che ogni istruzione richieda un tempo costante
 - * valido nel modello di calcolo che viene qui considerato (architettura di Von Neumann)
 - si assume che ogni diversa istruzione richieda un tempo di esecuzione (costante) potenzialmente diverso
 - * nel caso dell'*Insertion Sort*, ci sono 7 passi, quindi ci saranno sette diverse costanti da c_1 a c_7
 - si considera poi il numero di volte che una data riga viene eseguita
 - in generale, le intestazioni dei cicli sono eseguite un passo in più delle istruzioni al loro interno
 - * infatti, il `for` di riga 2 viene eseguito n volte, mentre le righe 3, 4 e 8, che sono all'interno del `for`, sono eseguite $n - 1$ volte
 - * questo è dovuto al fatto che il `for` è in realtà eseguito nel seguente modo:
 - prima si compie la fase di inizializzazione (a j assegna 2); tale fase viene eseguita una volta sola, quindi non viene considerata (impiega un tempo costante, quindi trascurabile rispetto al ciclo stesso che costante non è)
 - poi si esegue la fase di controllo (j è minore o uguale di $\text{length}(A)$, cioè di n ?)
 - poi si eseguono le istruzioni all'interno del `for`

- e si ritorna all'inizio del ciclo, per effettuare, nell'ordine, prima l'incremento della variabile (j in questo caso) e poi il controllo ($j \leq n$)
 - quando la fase di controllo fallisce, il `for` è finito
 - esattamente come nel C, come si vedrà
 - quindi, la fase di incremento-controllo (che si trova concettualmente nell'intestazione del `for`) viene sempre eseguita una volta di più delle istruzioni al suo interno: per ogni esecuzione delle istruzioni all'interno del ciclo, c'è una corrispondente fase di controllo con esito positivo, dopodiché c'è un'ulteriore fase di controllo che fallisce
 - ad esempio, se un ciclo ha una sola istruzione al suo interno, e questa istruzione viene eseguita 1 volta, allora la fase di incremento-controllo viene eseguita 2 volte (una dove il controllo ha successo prima dell'istruzione, un'altra dove fallisce dopo)
- ci possono essere dei *cicli annidati*, ovvero un ciclo (*interno* o *annidato*) tra le istruzioni di un altro ciclo (*esterno*)
 - è quello che succede qui: il `while` di riga 5 è annidato (ovvero, scritto dentro) il `for` (esterno) di riga 2
 - ovviamente, tali cicli vanno eseguiti un certo numero di volte, moltiplicato per il numero di volte del ciclo più esterno
 - * per esempio, se si potesse assumere che il `while` di riga 5 sia eseguito esattamente m volte, allora il numero di volte dell'esecuzione dell'intestazione del `while` (riga 5) sarebbe $m(n-1)$, mentre quello per le istruzioni interne al `while` (righe 6-7) sarebbe $(m-1)(n-1)$
 - qui tuttavia non si è in grado di dire quante volte il `while` sarà eseguito (dipende, come si vedrà, da com'è fatta la sequenza in input)
 - dato quindi che, per ogni valore j della variabile j , ci potrebbe essere un numero diverso di volte che il `while` viene eseguito, si definisce t_j proprio come il numero delle volte che il `while` viene eseguito quando la variabile j vale j
 - * il “trucco” sta nel fatto che, pur non sapendo quanto vale di preciso t_j , è possibile ragionarci sopra attraverso altre vie
 - quindi, data la definizione di t_j e quanto detto sopra, se ne ricava che il `while` di riga 5 viene eseguito $\sum_{j=2}^n t_j$, mentre le due istruzioni al suo interno sono eseguite $\sum_{j=2}^n (t_j - 1)$ volte ciascuna
 - infine, la funzione $T(n)$ che dà la complessità temporale per input di dimensione n generici si ottiene moltiplicando per ciascuna istruzione il numero di volte con il costo, e poi sommando tutte le istruzioni

– in questo caso:

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + \\
&\quad + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 (\sum_{j=2}^n t_j - \sum_{j=2}^n 1) + \\
&\quad + c_6 (\sum_{j=2}^n t_j - \sum_{j=2}^n 1) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 (\sum_{j=2}^n t_j - (n-1)) + \\
&\quad + c_6 (\sum_{j=2}^n t_j - (n-1)) + c_7(n-1) \\
&= (c_1 + c_2 + c_3 - c_5 - c_6 + c_7)n + (-c_2 - c_3 + c_5 + c_6 - c_7) + \\
&\quad + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n t_j
\end{aligned}$$

- messa così, non è molto utile a causa di quell'incognito t_j
- tuttavia, è possibile dire qualcosa su t_j
- da un lato, deve essere almeno 1 (lower bound), perché la condizione del **while** va testata almeno una volta, quindi $\forall j = 2, \dots, n$ vale che $t_j \geq 1$
- dall'altro lato (upper bound), deve essere al più j (cioè, il valore della variabile j in quel momento), quindi $\forall j = 2, \dots, n$ vale che $t_j \leq j$
 - * è meno evidente del fatto precedente, ma non meno vero
 - * lo si può provare come segue: nel caso del lower bound, affinché $t_j = 1$, è necessario che sia falsa sin da subito la condizione del **while**
 - * tale condizione è formata da due parti: una controlla che i , inizializzata al passo precedente a $j - 1$, sia strettamente maggiore di zero; dato che $j \geq 2$ (per via della riga 2...), si ha che $i = j - 1 \geq 1$ e quindi $i > 0$ vale sempre
 - * resta quindi la seconda parte, ovvero $A[i] > key$, che può anche essere falsa da subito
 - * nel caso dell'upper bound, si ha praticamente il contrario
 - * si cerca in quale caso $i > 0$ **and** $A[i] > key$ vale il più a lungo possibile
 - * ma dato che i parte da $j - 1$ e all'interno del **while** i viene sempre decrementato di 1 (riga 7), arriverà a 0 (facendo quindi fallire la prima parte del test $i > 0$ **and** $A[i] > key$, e quindi tutto il test, dato che è un **and**) in al più j passi (da $j - 1$ a 0 è lo stesso che da 0 a $j - 1$...)
- il caso migliore, ovviamente, sarà quello per cui $\forall j = 2, \dots, n$ vale che $t_j = 1$
- il caso peggiore, altrettanto ovviamente, sarà quello per cui $\forall j = 2, \dots, n$ vale che $t_j = j$
- se si assume il caso migliore, si ha quanto segue:

$$\begin{aligned}
T_{migl}(n) &= (c_1 + c_2 + c_3 - c_5 - c_6 + c_7)n + (-c_2 - c_3 + c_5 + c_6 - c_7) + \\
&\quad + c_4 \sum_{j=2}^n 1 + c_5 \sum_{j=2}^n 1 + c_6 \sum_{j=2}^n 1 \\
&= (c_1 + c_2 + c_3 - c_5 - c_6 + c_7)n + (-c_2 - c_3 + c_5 + c_6 - c_7) + \\
&\quad + c_4(n-1) + c_5(n-1) + c_6(n-1) \\
&= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7) \\
&= \Omega(n)
\end{aligned}$$

– se si assume il caso peggiore, si ha quanto segue:

$$\begin{aligned}
T_{pegg}(n) &= (c_1 + c_2 + c_3 - c_5 - c_6 + c_7)n + (-c_2 - c_3 + c_5 + c_6 - c_7) + \\
&\quad + c_4 \sum_{j=2}^n j + c_5 \sum_{j=2}^n j + c_6 \sum_{j=2}^n j \\
&= (c_1 + c_2 + c_3 - c_5 - c_6 + c_7)n + (-c_2 - c_3 + c_5 + c_6 - c_7) + \\
&\quad + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) + c_6 \left(\frac{n(n+1)}{2} - 1 \right) \\
&= \frac{1}{2}(c_4 + c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n + \\
&\quad + (-c_2 - c_3 - c_4 - c_7) \\
&= O(n^2)
\end{aligned}$$

* nel primo passaggio si è sfruttata la nota formula di Gauss:

$$\sum_{j=1}^n j = \frac{n(n+1)}{2}, \text{ dalla quale segue che } \sum_{j=2}^n j = \left(\sum_{j=1}^n j \right) - 1 = \frac{n(n+1)}{2} - 1$$

– volendo, l'ultimo passaggio per le due complessità poteva anche essere scritto come: $T_{migl} = \Theta(n)$ e $T_{pegg} = \Theta(n^2)$, sarebbe stato ugualmente corretto

– scritto così (con Ω e O), mette in risalto che la complessità è sempre tra n ed n^2