Lezione_171225_DeA_bozza

8.1.4.2. Partition

Algoritmo `Partition(S,a,b)`
Input: Sequenza S di n chiavi, indici $0 \leq a < b < n$.
Output: Sequenza S riorganizzata tra a e b, e l'indice $l \in [a,b]$ tale che $S[i] \leq S[l] \leq [j]$ per ogni i,j con $a \leq i \leq l \leq j \leq b$

```
p<-S[b];
l<-a;
r<-b - 1;
while(l <= r) do{
    while((l <= r) AND (S[l] <= p)) do l<-l +1;
    while((l <= r) AND (S[r >= p)) do r<-r - 1;
    if(l < r) then swap(S[l],S[r]);
}
swap(S[l],S[b]);
return l;
```

8.1.4.2.1. Esempio

$\ell = a$						r	b
85	24	63	45	17	31	96	50
ℓ					r		
85	24	63	45	17	31	96	50
ℓ					r		
31	24	63	45	17	85	96	50
		ℓ		r			
31	24	63	45	17	85	96	50
		ℓ		r			
31	24	17	45	63	85	96	50
			r	ℓ			
31	24	17	45	63	85	96	50
			r	ℓ			
31	24	17	45	50	85	96	63

8.1.4.3. Correttezza QuickSortInPlace

La correttezza di `QuickSortInPlace` è immediata conseguenza della correttezza di `Partition`.

8.1.4.3.1. Correttezza Partition

La correttezza di `Partition` è basata sul seguente invariante, che vale alla fine di ogni iterazione del while esterno
[disegno]

- $p = S[b]$;
- $S[j] \leq p \forall a \leq j < l$;
- $S[j] \geq p \forall r < j \leq b$;
- $l \leq r + 1$.
  È facile dimostrare che l'invariante è mantenuto alla fine di ogni iterazione del while esterno.

Quando il while esterno termina la sua esecuzione, avremo che $l = r + 1$ e la situazione di S è la seguente:
[disegno]
Tutto questo grazie all'invariante.

Finito il while esterno, l'istruzione `swap(S[b],S[l])` renderà l'output di `Partition` (l'indice l) corretto in base alla specifica input/output.
□

8.1.4.4. Complessità Partition

> La complessità di `Partition(S,a,b)` è $\Theta(b - a + 1)$, ovvero lineare nel numero di chiavi presenti tra gli indici a e b.

La proposizione discende dalle seguenti osservazioni:

- Ogni due iterazioni consecutive del while esterno:
  - L'indice l cresce e l'indice r decresce;
  - Il numero di operazioni eseguito è proporzionale al numero di incrementi/decrementi dei due indici l e r.
- Il numero totale di incrementi/decrementi dei due indici l e r è proporzionale alla lunghezza della sottosequenza $b - a + 1$.

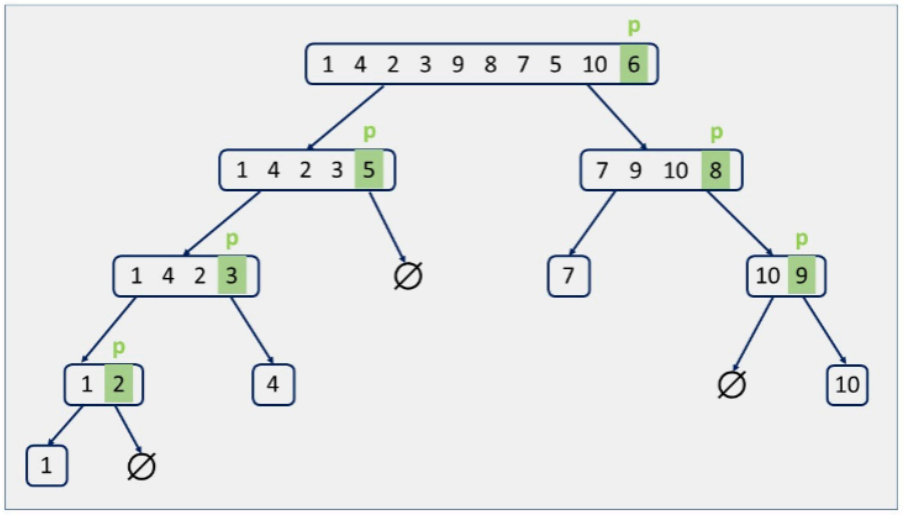
8.1.4.5. Complessità QuickSortInPlace

> La complessità di `QuickSortInPlace(S,0,n-1)` è $\Theta(n^2)$.

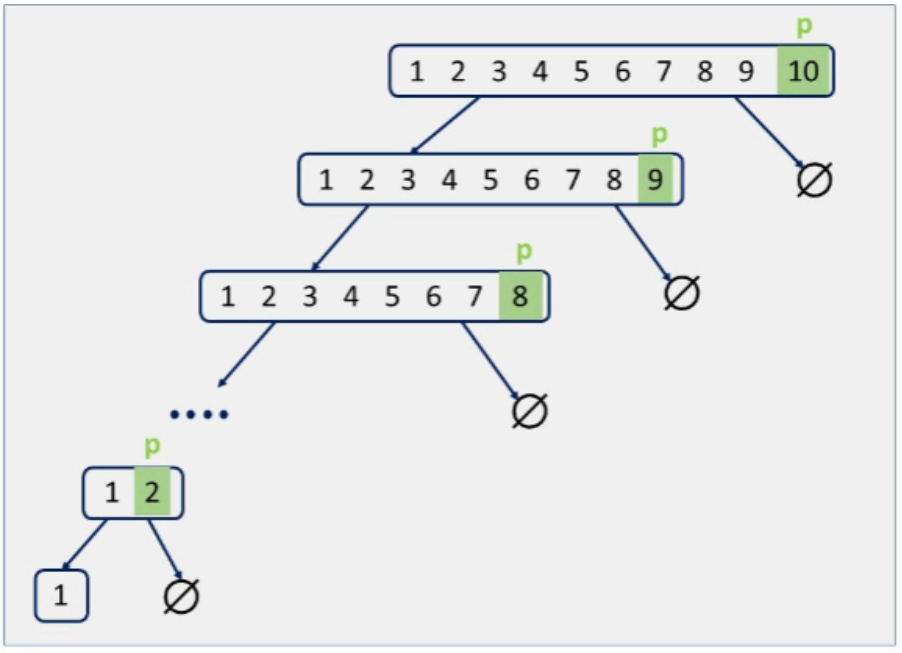
Per l'analisi di complessità dobbiamo considerare l'albero della ricorsione.

Prima di procedere con l'analisi facciamo due esempi:

Vediamo l'albero della ricorsione per $S = 1, 4, 2, 3, 9, 8, 7, 5, 10, 6$



Vediamo ora l'albero della ricorsione per $S = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$



Consideriamo l'albero della ricorsione per una istanza arbitraria di taglia n:

- Il numero di livelli dell'albero della ricorsione è al massimo n, perché da un livello al successivo almeno un pivot è sistemato definitivamente;
- Per il punto precedente, la taglia aggregata delle istanze del livello i è al massimo $n - i$;
- Il costo contribuito da un nodo dell'albero della ricorsione è lineare nella taglia della istanza a essa associata, in quanto dominato dalla complessità di `Partition`

- $\implies$ Deduco che:
  - Ci sono al più n livelli;
  - Il costo contribuito dal livello i è $O(n-i)$
    $\implies$ Complessità: $O\left(\sum_{i=0}^{n-1}(n-i)\right) = O(n^2)$.

Per dimostrare che la complessità è anche $\Omega(n^2)$ (e quindi alla fine $\Theta(n^2)$), consideriamo come istanza S la sequenza di n chiavi ordinate. Generalizzando l'esempio si vede facilmente che per questa istanza l'albero della ricorsione ha *esattamente* n livelli e il costo contribuito dal livello i è $\Theta(n-1) \implies$ il costo toale è $\Theta_8 \sum_{i=0}^{n_1}(n-i) = \Theta(n^2)$.
Questa è l'istanza "cattiva" che dimostra che la complessità è $\Omega(n^2)$.

> **Osservazione**
>
> `QuickSortInPlace` è in place rispetto alla memoria Heap che mantiene le chiavi sempre nella sequenza S di input, ma non è in place in quanto le variabili a, b, l sono generate in più copie, una per ogni chiamata ricorsiva, e richiedono uno spazio proporzionale all'altezza dell'albero della ricorsione (che può essere proporzionale a n).

8.1.4.6. Randomized QuickSort

`Randomized QuickSort` è una variante di `QuickSortInPlace` che *evita il caso pessimo probabilisticamente*.
Dentro `Partition` sostituisce l'istruzione `p<-S[b]` con le seguenti tre istruzioni:

```
i<- inero random in [a,b];
swap(S[i],S[b]);
p<-S[b];
```

> **Osservazione**
>
> Per ogni istanza S esistono molte esecuzioni possibili e quindi molti diversi tempi di esecuzione, funzione delle scelte probabilistiche (scelta del pivot) fatte dall'algoritmo.

8.1.4.6.1. Analisi di Randomized QuickSort

`Randomized QuickSort` è un algoritmo probabilistico in quanto esistono diverse possibili esecuzioni per ogni istanza, che sono determinate dalle scelte del pivot.

Sia $t_{i,RQS}$ la variabile aleatoria che rappresenta il numero di operazioni eseguite da `Randomized QuickSort`d per risolvere l'istanza i.

> Per ogni istanza i di taglia n si ha che
>
> $$\mathbb{E}[t_{i,RQS}] \in \Theta(n \log n)$$
>
> (la *complessità in media*) e
>
> $$\Pr(t_{i,RQS}(n \log n)) \geq 1 - \frac{1}{n}$$
>
> (*complessità in alta probabilità*).

Questa proposizione rende quantitativamente evidente il fatto che, in pratica, la performance di `Randomized QuickSort` *per una qualsiasi istanza di input* è paragonabile a quella di `MergeSort` e dei miglio algoritmi basati su confronti.

8.1.5. InsertionSort

Algoritmo `InsertionSort(S)`
Input: Sequenza S di n chiavi.
Output: Sequenza S ordinata in senso crescente.

```
for i<-1 to n-1 do{
    curr<-S[i];
    j<-i-i;
    while((j >= 0) AND (S[j] > curr)) so{
        S[j+1]<-S[j];
        j<-j-1;
    }
    S[j+1]<-curr;
}
```

> **Osservazione**
>
> `InsertionSort` è un algoritmo *in-place*.

8.1.5.1. Complessità di InsertionSort

8.1.5.1.1. Analisi standard

- Complessità $O(n^2)$.
  Nell'iterazione i del for, il ciclo while esegue *al più* i iterazioni di costo $\Theta(1)$ ciascuna. La complessità totale risulta quindi essere $O\left(\sum_{i=1}^{n-1} i\right) = O(n^2)$
- Complessità $\Omega(n^2)$.
  Per l'istanza costituita da chiavi distinte inizialmente ordinate in senso decrescente si ha che nell'iterazione i del for, il ciclo while esegue esattamente i iterazioni di costo $\Theta(1)$ ciascuna.

8.1.5.1.2. Analisi più fine

> Data una sequenza di n chiavi S, un'*inversione*, rispetto all'ordinamento crescente, è una coppia di indici (i, j) con $0 \leq i \neq j < n$ tale che $j < i$ e $S[j] > S[i]$.

Osservazione

Sia K il numero di inversioni in una sequenza S di n chiavi. È immediato vedere che

$$0 \leq K \leq \binom{n}{2} = \frac{n(n-1)}{2}$$

- Se S è ordinata in senso crescente $\implies K = 0$;
- Se S è ordinata in senso decrescente $\implies K = \binom{n}{2}$.

L'obiettivo è l'analisi della complessità di `InsertionSort` in funzione della taglia e del numero di inversioni della sequenza S.

> La complessità di `InsertionSort(S)` è $\Theta(n + K)$, dove n è il numero di chiavi e K il numero di inversioni in S.

Osservazione

- L'analisi non contraddice quella standard, dato che si può avere $K = \Theta(n^2)$, ma evidenzia il fatto che `InsertionSort` è molto efficiente quando la sequenza di input è quasi ordinata (ovvero ha poche inversioni).

- L'introduzione di K per una partizione più fine delle istanze e quindi una più precisa analisi della complessità.

Sia S una sequenza di n chiavi con K inversioni. Le operazioni eseguite da `InsertionSort` sono raggruppabili in due termini:

- A: operazioni eseguite nei cicli while;
- B: tutte le operazioni eseguite fuori dal while.
  È immediato vedere che $B \in \Theta(n)$.

Per ogni $1 \leq i \leq n$ definiamo K_i il numero di inversioni (i, j) con $j < i$ e $S[j] > S[i]$ $\implies$ il costo del while eseguito nella'iterazione i del for è $\Theta(K_i)$.
$\implies A \in O\left(\sum_{i=1}^{n} K_i\right) = O(K)$ dato che orni inversione è contata una sola volta in un K_i.

$\implies$ Complessità di `InsertionSort` è $\Theta(A + B) = \Theta(K + n)$.