Esercizio 1

Giovanni Stefanini - 6182949

Aprile 2021

1 Introduzione

In questo esercizio è stato prima implementato una funzione che generasse un vettore casuale, e in seguito sono stati implementati gli algoritmi di InsertionSort() e QuickSort(). Il tutto è stato implementato nel file *Es1ASD.py*.

In particolare nel file *Test1ASD.py* sono stati poi implementati delle funzioni che testassero gli algoritmi applicandoli ad array di dimensioni crescenti con valori casuali, e prendendo come elemento di riferimento per la qualità dell'algoritmo il **tempo di esecuzione**.

Quindi per poter effettuare l'analisi sono stati prodotti dei grafici e delle tabelle che permettessero di confrontare i due algoritmi di ordinamento.

2 Insertion Sort

L'**insertion sort** è un algoritmo di ordinamento iterativo adatto ad ordinare un array di relativamente pochi elementi; dato che non necessita una copia del vettore per ordinarlo (risparmiando così memoria), ricade nella categoria degli algoritmi *in place*.

Il suo funzionamento è simile all'ordinamento di un mazzo di carte: inizialmente si ha una mano vuota, e le carte da ordinare sul tavolo; ad ogni iterazione viene presa una carta e inserita nella posizione corretta, trovata confrontandola con ogni altra carta in mano. In particolare, in ogni momento le carte nella mano sono ordinate.

Ha tempi di esecuzione:
Caso migliore: $\Theta(n)$ (array già ordinato)
Caso medio: $\Theta(n^2)$
Caso peggiore: $\Theta(n^2)$ (array ordinato al contrario)

Algoritmo Insertion Sort:

```
def insertionSort(A):
    for j in range(1, len(A)):
        key = A[j]
        i = j - 1
        while i >= 0 and A[i] > key:
            A[i + 1] = A[i]
            i = i - 1
        A[i + 1] = key
    return A
```

3 Quick Sort

Quick Sort usa due funzioni, *Partition*, che sceglie un pivot $x = A[q]$ e suddivide A in due sottoarray $A[p..q-1]$ e $A[q+1..r]$ tali che $A[p..q-1] <= A[q] <= A[q+1..r]$ (per ogni loro elemento), e *Quicksort*, il quale riordina A suddividendolo ricorsivamente in sottoarray sempre più piccoli.

Asintoticamente ottimo, in quanto ha tempi di esecuzione:

Caso migliore: $\Theta(n \lg n)$ (sottoarray di uguale dimensione)
Caso medio: $\Theta(n \lg n)$
Caso peggiore: $\Theta(n^2)$ (sottoarray con 1 ed $n-1$ elementi)

Algoritmo Quick Sort

```
def quickSort(A, low, high):
    if len(A) == 1:
        return A
    if low < high:
        partitionIndex = partition(A, low, high)
        quickSort(A, low, partitionIndex-1)
        quickSort(A, partitionIndex+1, high)
    return A
```

Partition

```
def partition(A, low, high):
    i = (low-1)
    pivot = A[high]
    for j in range(low, high):
        if A[j] <= pivot:
            i = i + 1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[high] = A[high], A[i+1]
    return (i+1)
```

4 Specifiche tecniche

Ai fini dell'esperimento, i due algoritmi sono stati implementati nel linguaggio Python nella loro versione più semplice. Le due funzioni sono state utilizzate in dei programmi di test presenti in *Test1ASD.py*.
I programmi di test sono composti da due cicli *for*. Il primo è utile per aumentare il numero di valori (da 1 fino a un valore prefissato a 1000) dell'array che viene ordinato dai due algoritmi, mentre il secondo è utile per fare una media di 10/15 esecuzioni per ottenere un risultato più attendibile. Inoltre tali programmi sono serviti per ottenere il tempo di esecuzione nel caso migliore, peggiore e medio per insertion-sort, e per ottenere il tempo di esecuzione nel caso peggiore e medio di quick-sort.

Le **specifiche hardware e software** della macchina utilizzata per eseguire i test sono:

Scheda madre: X580VD Scheda di base ASUSTeK COMPUTER INC.
CPU: Intel Core i7-7700HQ CPU - 2.80GHz, 2808 Mhz, 4 core, 8 processori logici
RAM: 16 GB
SSD: SanDisck SD8SN8U128G1002 - 120 GB
HDD: TOSHIBA MQ04ABF100 - 1 TB
SO: Microsoft Windows 10 Home
IDE: JupyterLab 2.2.6

5 Simulazione e Risultati

5.1 Dati causali: Si mostra di seguito il tempo di esecuzione dei due algoritmi su un vettore di 10 numeri causali.

Array da ordinare:	182	566	270	265	970	502	393	633	70	39

$A.length = 10$

Risultati:

InsertionSort	QuickSort
5,66 x 10^{-4} s	8.97 x 10^{-5} s

Array ordinato:	39	70	182	265	270	393	502	566	633	970

$A.length = 10$

Osservazioni:
Si può notare che nel caso si debba ordinare lo stesso array casuale di 10 elementi l'insertion-sort risulta leggermente più lento del quick-sort.

5.2 Caso Medio InsertionSort e QuickSort: Si mostra, di seguito, il tempo di esecuzione dei due algoritmi su un array con numero di valori crescente da 1 a 1000 dati casuali, nel caso MEDIO. Per avere un risultato più stabile è stata fatta una media su 10 esecuzioni sullo stesso numero di valori. Per costruire la tabella sono stati presi i tempi di esecuzione dopo che il numero di valori dell'array è aumentato di 100 in 100.

Risultati per InsertionSort e QuickSort con array casuale (Caso Medio):

Numero di valori	Tempo InsertionSort caso Medio (ms)	Tempo QuickSort caso Medio (ms)
0	0.00082	0.00061
100	0.37865	0.12095
200	1.41676	0.25874
300	3.27591	0.43572
400	6.20018	0.67145
500	9.47265	0.80505
600	14.02249	1.06474
700	18.96982	1.17452
800	26.29462	1.5214
900	31.50008	1.59348
999	39.603	1.76817

Figura 1: Tabella del caso medio di InsertionSort e QuickSort

Osservazioni:
Si può notare che QuickSort risulta più veloce dell'InsertionSort.
In particolare possiamo notare che il blocco, che più aumenta il numero di valori più aumenta il distacco tra i tempi dei due algoritmi, diventando sempre più evidente quanto il quicksort sia migliore dell'insertionsort per una array con valori casuali.

Se mettiamo a confronto i due grafici che otteniamo dall'esecuzione di InsertionSort e Quick Sort risulta evidente l'andamento quadratico per l'InsertionSort e come $nlg(n)$ per il QuickSort, infatti otteniamo i seguenti grafici:

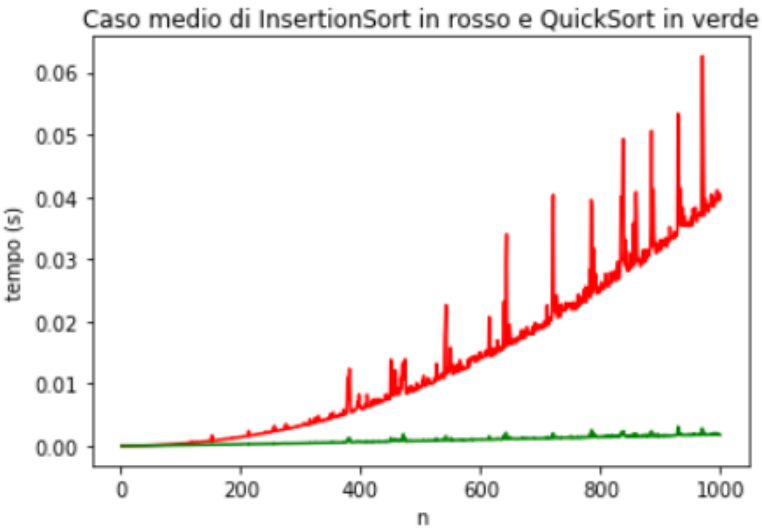

Figura 2: Grafico del caso medio di InsertionSort e QuickSort

5.3 Caso Peggiore InsertionSort e QuickSort: Per testare invece il caso peggiore di InsertionSort sono stati usati array, sempre di grandezza crescente da 1 a 1000, ma ordinati in modo decrescente. Infine è stato testato il caso PEGGIORE per QuickSort cioè quando i dati sono ordinati o meglio il caso peggiore di quicksort lo ottengo quando scelgo come pivot l'elemento più grande o più piccolo dell'array, ottenendo così 2 sottoarray con 1 ed $n-1$ elementi. Anche in questo caso per avere un risultato più stabile è stata fatta una media su 10 esecuzioni sullo stesso numero di valori. Per costruire la tabella sono stati presi i tempi di esecuzione dopo che il numero di valori dell'array è aumentato di 100 in 100.

Risultati InsertionSort e QuickSort caso Peggiore:

Numero di valori	Tempo InsertionSort caso Peggiore (ms)	Tempo QuickSort caso Peggiore (ms)
0	0.00093	0.00069
100	0.85475	0.59712
200	3.41297	2.15101
300	6.7231	4.65495
400	12.06219	8.42028
500	25.61159	19.54194
600	28.28808	21.59703
700	39.19119	26.88968
800	50.51099	35.42325
900	63.82183	44.92032
999	79.529	55.55736

Figura 3: Tabella del caso peggiore di Insertionsort e Quicksort

Osservazioni:

Confrontando i grafici dei due algoritmi risulta evidente l'andamento quadratico sia per l'InsertionSort, sia per il QuickSort. DI seguito i due grafici:

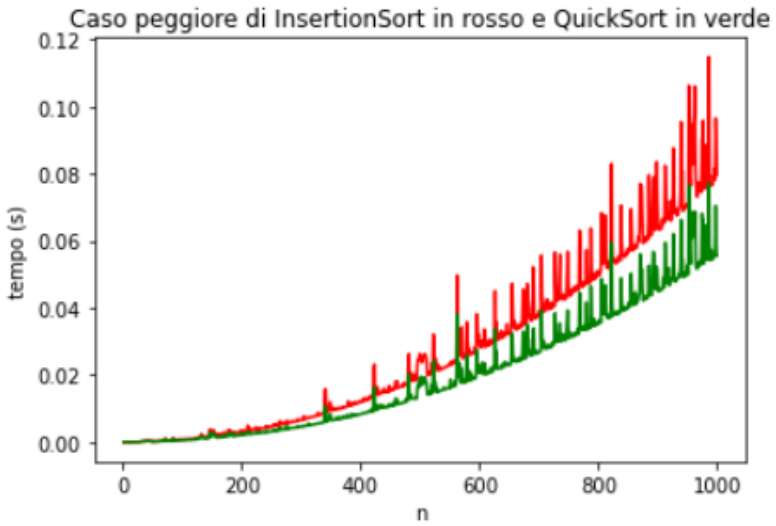

Figura 4: Grafico del caso peggiore di Insertionsort e Quicksort

5.4 Caso Migliore InsertionSort: Di seguito viene testato il caso MIGLIORE per InsertionSort cioè quando i dati sono ordinati in ordine crescente. A tal fine è stato utilizzato lo stesso metodo del precedente test, cioè sono stati usati array con numero di elementi crescenti da 1 a 1000 ma sempre ordinati. In questo caso però è stata fatta una media su 15 esecuzioni. Ciò è stato possibile grazie alla velocità di insertionsort nel suo caso migliore, permettendo così di avere un risultato più uniforme.

Risultati InsertionSort caso Migliore:

Numero di valori	Tempo InsertionSort caso Migliore (ms)
0	0.00066
100	0.02206
200	0.03886
300	0.05781
400	0.07905
500	0.09879
600	0.11952
700	0.14137
800	0.16081
900	0.18129
999	0.20251

Figura 5: Tabella del caso migliore di Insertionsort

Osservazioni:
Notare che InsertionSort risulta molto veloce nel suo caso migliore. É talmente tanto veloce che riesce ad ordinare l'array più grande in un tempo inferiore rispetto al caso medio di quicksort (0,2 ms di insertionsort contro i 1,7 ms di quicksort).

Il grafico che otteniamo dal caso migliore di InsertionSort è il seguente:

Figura 6: Grafico del caso migliore di InsertionSort

5.5 Caso Migliore QuickSort: Il caso migliore di quicksort svolgendo ricerche, è risultato che tale situazione accade quando l'array ha dimensioni pari a potenze del 2 e costituisce la sequenza di attraversamento in ordine di un albero binario bilanciato, una serie non facilmente generabile tramite semplici formule. Notando che tale soluzione è risultata discutibile e poco affidabile, motivo per cui, assieme all'effettiva mancanza di un'accurata analisi di questo caso nell'ambito del corso di Algoritmi e Strutture Dati, il caso migliore è stato escluso dalla relazione finale.

6 Conclusioni

Tranne che nel caso migliore di quicksort, che non è stato possibile implementare correttamente, tutte le performance studiate nella teoria dei diversi casi di insertion sort e quicksort sono state confermate dall'esperimento, indicando come l'analisi matematica degli algoritmi sia effettivamente corretta e visualizzabile anche nella pratica.

Nello specifico, il caso migliore di insertion sort, ottenuto chiamandolo su array già ordinati, conferma ampiamente l'analisi effettutata matematicamente di un tempo atteso $\Theta(n)$ in quanto, come osservabile dai risultati ottenuti, i tempi di esecuzione hanno andamento come una funzione lineare.

Similmente viene confermata anche l'ipotesi di costo quadratico $\Theta(n^2)$ nel caso medio (array random) e peggiore (array ordinati al contrario), in cui l'andamento si identifica con una parabola.

Il caso medio di quicksort, invece, implementato chiamandolo su array randomizzati, mostra la correttezza dell'ipotesi di costo $\Theta(n \lg n)$. Confrontandolo con il caso migliore dell'insertion sort per numero di valori alto notiamo che è decisamente inferiore al caso medio del qucik sort.

Infine, in modo simile ai casi medio e peggiore di insertion sort, risulta confermato anche il costo quadratico del caso peggiore di quicksort (array ordinato al contrario o già ordinato), in quanto l'andamento è come una parabola.