Relazione esercizio 1: libreria di ordinamento

Studente: Cagnazzo Christian Damiano – Matricola: 883100

TABELLA DEI TEMPI DI CALCOLO: ordinamento di un array di 20 milioni di elementi

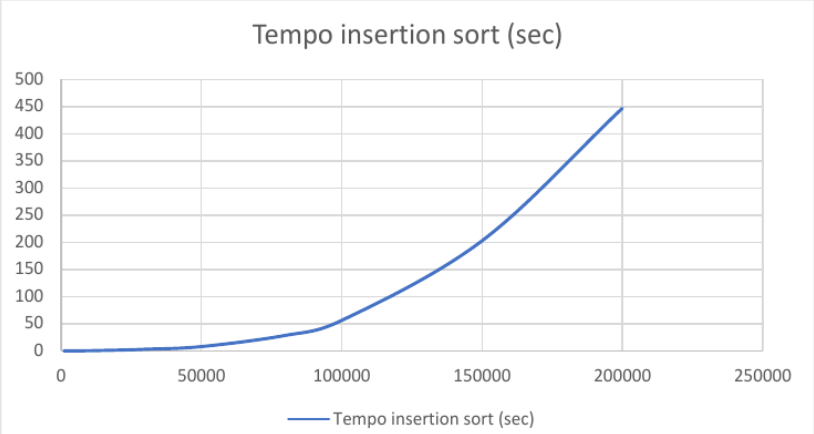
	INSERTION SORT	**QUICK SORT**
Primo campo (stringhe)	Oltre 10 minuti	Oltre 10 minuti
Secondo campo (interi)	Oltre 10 minuti	Circa 23 secondi
Terzo campo (float)	Oltre 10 minuti	Circa 25 secondi

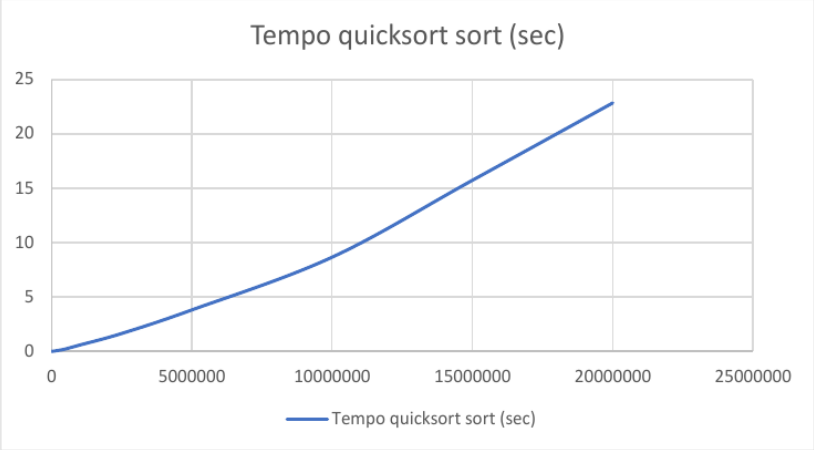
Osservazioni:

Per quanto riguarda l'algoritmo dell'*insertion sort* i risultati erano abbastanza prevedibili: il motivo è che il tempo richiesto da esso dipende dai dati in ingresso e inoltre può variare a seconda di come gli elementi sono posizionati anche in array di uguali dimensioni. Nel caso medio (simile a quello peggiore in cui l'array è ordinato in ordine decrescente), infatti, il tempo di calcolo di *insertion sort* risulta essere una funzione quadratica dell'input. Al contrario, il tempo di calcolo del *quick sort* (nonostante anch'esso nel caso peggiore sia una funzione quadratica) spesso è la soluzione pratica migliore per effettuare un ordinamento, poiché mediamente molto efficiente: il suo tempo di esecuzione medio, infatti, è $\Theta(n \lg n)$. A dimostrazione di ciò ecco alcuni test effettuati con i due algoritmi che ordinano il secondo campo, ovvero un intero:

Numero elementi	Tempo insertion sort (sec)
1000	0,0034
5000	0,1447
10000	0,2855
15000	0,8495
20000	1,3892
30000	2,9932
50000	7,8863
80000	28,5884
100000	56,1026
150000	202,6862
200000	446,8787

Numero elementi	Tempo quicksort sort (sec)
1000	0,0001
10000	0,0016
50000	0,0106
100000	0,0263
300000	0,1102
500000	0,2194
1000000	0,5757
2500000	1,6432
5000000	3,8095
10000000	8,6649
15000000	15,7394
20000000	22,8543





Come possiamo notare, per quanto riguarda l'*insertion sort,* con 200 mila elementi il suo tempo di esecuzione è di circa 450 secondi, ovvero circa 8 minuti; al contrario il *quick sort* impiega circa 2 secondi per un po' più della stessa quantità. Per ordinare l'intero array di 20 milioni di elementi il *quick sort* impiega circa 23 secondi.

Ciò che potrebbe però sembrare strano è che il *quick sort* impieghi tanto tempo per quanto riguarda il terzo campo, ovvero quello delle stringhe. Il motivo è da ritrovarsi nel fatto che nel nostro array di stringhe sono presenti tanti duplicati e il *quicksort* presenta scarse prestazioni con ingressi che contengono molti elementi ripetuti. Il motivo si riesce a capire facilmente prendendo un array con tutti gli elementi uguali: ad ogni ricorsione, (per come è implementata la procedura partition) la partizione destra rimane vuota (nessun valore è minore del perno) e la partizione sinistra è diminuita di un solo elemento (il perno viene rimosso). Di conseguenza, l'algoritmo richiederà tempo quadratico per ordinare l'array.

```
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  2  2  2  2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  2  2  2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  2  2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  2  -
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: 2  -
```

Una possibile soluzione prevede che, durante il processo di partizionamento, gli elementi uguali al perno vengano spostati immediatamente a fianco di esso. In questo modo si avranno tre partizioni: una con gli elementi minori del pivot, una con gli elementi uguali (che verrà "rimossa" dalle chiamate ricorsive) e una con gli elementi maggiori del perno. Ecco come cambia la situazione:

```
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2
2     2     2     2     2     2     2     2     2     2     Chiamata ricorsiva sugli array: -
```

Il caso che prima era peggiore, ovvero quello con tutti gli elementi uguali, è diventato il caso migliore, con due sole chiamate ricorsive su array vuoti.

Vediamo come lavora il nuovo algoritmo:

```
6   2   5   3   5   6   8   5   2   2

2   6   5   3   5   6   8   5   2   2
2   5   6   3   5   6   8   5   2   2
2   5   3   6   5   6   8   5   2   2
2   5   3   5   6   6   8   5   2   2
2   5   3   5   6   6   8   5   2   2
2   5   3   5   6   6   2   5   2   8
2   5   3   5   2   6   6   5   2   8
2   5   3   5   2   5   6   6   2   8
2   5   3   5   2   5   2   6   6   8     Chiamata ricorsiva sugli array: 2  5  3  5  2  5  2  -  8

2   2   3   5   2   5   5   6   6   8
2   2   3   5   2   5   5   6   6   8
2   2   5   5   2   3   5   6   6   8
2   2   2   5   5   3   5   6   6   8
2   2   2   5   5   3   5   6   6   8
2   2   2   5   5   3   5   6   6   8     Chiamata ricorsiva sugli array: -  5  5  3  5

2   2   2   5   5   3   5   6   6   8
2   2   2   3   5   5   5   6   6   8
2   2   2   3   5   5   5   6   6   8     Chiamata ricorsiva sugli array: 3  -
```

L'array da ordinare è composta da {6,2,5,3,5,6,8,5,2,2} e come abbiamo notato più elementi ripetuti ci sono e meno chiamate ricorsive effettuerà l'algoritmo.

Nel primo ciclo l'algoritmo si occupa di spostare il perno 6 nella sua corretta posizione e con lui anche tutti i suoi duplicati, per poi effettuare la chiamata ricorsiva sugli array 2-5-3-5-2-5-2 e 8 (che essendo di un solo elemento sarà già ordinato). Nel primo array, poi, l'algoritmo, posizionerà il perno 2, e tutti i duplicati, nella loro corretta posizione. La chiamata ricorsiva sarà poi effettuata sull'array 5-5-3-5 in cui viene posizionato il perno 5 correttamente e, infine, su 3 che essendo composto da un elemento è già ordinato. Abbiamo ottenuto il risultato che volevamo con un totale di 6 chiamate ricorsive, al contrario del *quick sort* originale che, come vediamo in figura, ne avrebbe impiegate il doppio.

```
6   2   5   3   5   6   8   5   2   2

2   2   5   3   5   6   2   5   6   8     Chiamata ricorsiva sugli array: 2  2  5  3  5  6  2  5  -  8

2   2   2   3   5   6   5   5   6   8     Chiamata ricorsiva sugli array: 2  2  -  3  5  6  5  5

2   2   2   3   5   6   5   5   6   8     Chiamata ricorsiva sugli array: 2  -

2   2   2   3   5   6   5   5   6   8     Chiamata ricorsiva sugli array: -  5  6  5  5

2   2   2   3   5   5   5   6   6   8     Chiamata ricorsiva sugli array: 5  5  -  6

2   2   2   3   5   5   5   6   6   8     Chiamata ricorsiva sugli array: 5  -
```

Eseguendo, infine, il *quicksort* utilizzando la nuova procedura di partizione anche per il terzo campo, ovvero quello delle stringhe in cui ci sono tanti elementi duplicati, si ottiene un tempo di esecuzione intorno ai 25 secondi.