Relazione sul Primo Esercizio

Tutti e quattro gli algoritmi ordinano correttamente e con risultati prevedibili. Il Quick Sort, sebbene di pochissimo, è più veloce del Merge Sort che è comunque ottimo. L' Heap ha tempi discreti e in ultimo c'è l'Insertion che essendo quadratico in "n" ha tempi talmente diluiti da rendere impossibile un test diretto sul campione totale. Per testarne la correttezza, oltre agli unit test, ho creato un sottocampione dal file csv mediante il comando:

"head -20000 univer.csv > minicampione.csv"

Per quanto riguarda i tempi di esecuzione ho notato che dipendono fortemente dalla macchina su cui gira il programma. Era ovvio una certa influenza, ma non immaginavo fosse cosi discriminante. Essendo il tempo cosi dipendente dalla macchina ho scelto di registrare i tempi sia dal mio pc, e sia su una macchina comune a tutti (quella del laboratorio). In questo modo i tempi sono paragonabili e si rende esplicita la profonda differenza di cui sopra.

Tempo di lettura dal file:

(+) Loading data from univer.csv: Completed in 8.54128 seconds

Tempi di esecuzione del Quick Sort:

String Field = 31.18523 seconds **VS** 8.66 seconds
Numb Field = 24.08124 seconds **VS** 10.42 seconds
Float Field = 25.90864 seconds **VS** 11.79 seconds

Tempi di esecuzione del MergeSort:

String Field = 23.20302 seconds **VS** 8.90 seconds
Numb Field = 25.76723 seconds **VS** 12.42 seconds
Float Field = 27.02772 seconds **VS** 13.56 seconds

Tempi di esecuzione del Heap Sort:

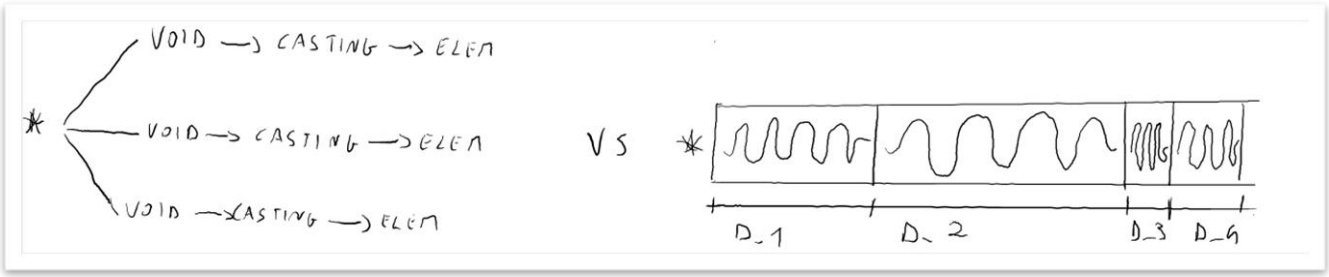
String Field = 57.68011 seconds **VS** 22.06 seconds
Numb Field = 69.62781 seconds **VS** 30.05 seconds
Float Field = 70.34341 seconds **VS** 31.40 seconds

Tempi di esecuzione del Insertion Sort:

String Field = ho interrotto dopo
Numb Field = un periodo di attesa
Float Field = maggiore di 15 minuti

Ottimizzazione algoritmo Quicksort:
Inizialmente avevo scelto di implementare la versione del quick sort vista a lezione e presentata sulle slide ma, sebbene molto chiara, essa è più orientata a far capire il concetto teorico che ad fornire una versione ottima dell'algoritmo. In seguito mi sono accorto che la versione presentata sul Cormen era più efficiente in quanto rispetto a quella vista a lezione faceva si più chiamate a "partition" ma ognuna di esse è in media più veloce, quindi nel complesso era molto più efficente l'ordinamento, soprattutto con le stringhe (perché ci sono molte stringhe uguali e quindi il controllo si prolunga per tutta la lunghezza della stringa).

Nota: Ho scelto di implementare un array di puntatori a void per rendere più agevole lo svolgimento e il mantenimento del progetto (nonché svincolarmi dalle limitazioni fisiche della macchina tramite astrazione). Con l'array di puntatori void una volta fatto un defernziamento ci troviamo di fronte ad un puntatore all' elemento, il quale può essere comparato mediante funzione apposita. Con un array di void diretti invece, una volta deferenziato avremmo direttamente il dato e non sapendo quanto esso sia grande l'esecutore non saprebbe accedervi se non in seguito ad una specifica indicazione del programmatore, mediante aritmetica dei puntatori.



*array[i] *(array+dim_elemento)

Consumi in termini di spazio (Ram occupata):
0,0 GB -> stato iniziale
2,0 GB -> durante il riempimento delle struct (picco massimo)
1,3 GB -> durante il sorting

Il picco massimo è dovuto alla presenza comune dei 700mb del file in ram e al tempo stesso dei medesimi dati ripetuti nelle varie struct. Subito dopo il lavoro del parser questi dati vengono liberati e il programma lavora con il minor carico possibile.

La versione iniziale del progetto aveva costi in termini di RAM pari circa a 4 Giga, che in seguito ad analisi e test sono stati dimezzati. Qui di seguito spiego nel dettaglio come e perché.

La struttura originale che avrebbe dovuto salvare i dati era di soli puntatori, e non statica come adesso. Questa è la rappresentazione del vecchio tipo di struct:

```
struct Records
        int* id;
        char* string_field;
        int* numb_field;
        float* float_field;
```

Questo comporta un enorme spreco di spazio in quanto:

1) Ogni intero occupava sempre 64 bit invece che il numero minimo di bit necessari per la sua rappresentazione binaria.

Perché avveniva questo? Propongo un piccolo esempio grafico. Questa qui di seguito è una fittizia struct e il suo puntatore ha valore 1800 (in un caso reale avrebbe un valore diverso e scritto su 64 bit). Quando alloco un puntatore a int come primo elemento della struct il compilatore dovrà allocare tutto lo spazio necessario e quindi si sposterà in avanti di 8 byte, cioè dello spazio richiesto per un puntatore (ovvero sempre 64 bit) arrivando all' indirizzo 1807. Invece allocando direttamente un intero il compilatore si sarebbe spostato solo di 32 bit, ovvero di 4 byte (ogni int occupa 4 byte, e per questo che il compilatore ha bisogno di sapere il tipo del dato (casting) per accedervi) arrivando all' indirizzo 1803.

```
 1800      1801       1802        1803      1804       1805       1806        1807
| 0000|   | 1111 |   | 0000 |    | 1111 |  | 1010 |   | 0000 |   | 1001 |    | 0000 |
|1111 |   | 0000 |    | 1111|    | 1111 |  | 0110 |    | 1111|    | 0101|     | 1111 |
  1          2           3           4          5          6          7           8
```

Per fare un esempio diretto: il valore 8 occupa 3 bit, il puntatore al valore 8 ne occupa 64. (Nella struct non sapendo a prescindere il numero che si inserirà si alloca lo spazio massimo potenzialmente occupabile, che comunque è di 32 bit e non 64).

2) Per le stringhe ho notato che una intera riga media è scritta su 30 caratteri, quindi un singolo sotto elemento della riga non arriverà mai a più di 30. Il motivo per cui ho stabilito a priori il limite massimo è che lo spazio allocato per una stringa è calcolato in base alla prima potenza di due superiore al valore richiesto. Per esempio una stringa di 17 caratteri richiede uno spazio di 32.

A seguito di tali considerazioni ho deciso di re implementare la struttura cambiando i vari deferenziamenti e accessi alla stessa:

Esempi di deferenziamenti ai vecchi puntatori a int:

```
"int p1  =  *(*((struct Records*)ptr1)).numb_field;
```