Relazione laboratorio: Esercizio 1

La libreria OrderedArray implementa una struttura dati “generica”, la cui genericità è garantita dall’utilizzo di puntatori a void invece che l’utilizzo di puntatori a tipi specifici, la ***struct \_OrderedArray*** è composta da 5 campi:

* Un ***void \*\*record*** utilizzato per contenere i singoli record del csv
* Un ***void \*\*cmp\_col*** utilizzato per contenere i singoli valori da ordinare
* Un ***unsigned long size*** per tenere conto di quanti elementi ci sono attualmente all’interno dell’array
* Un ***unsigned long capacity*** Che ne definisce una capacita’ temporanea dichiarata al momento della funzione per l’inizializzazione della struttura stessa
* Un ***int (\*precedes) (void\*, void\*)*** che contiene la funzione del tipo specifico con cui verranno confrontati i valori di ***cmp\_col***

Ogni campo viene inizializzato mediante una chiamata alla funzione ***OrderedArray \*ordered\_array\_init(unsigned long size)*** che prende come parametro la dimensione (***capacity***) iniziale che si vuole assegnare,

nel momento in cui ci fosse invece bisogno di aumentarla ulteriormente, possiamo ricorrere alla funzione  ***void ordered\_array\_realloc(OrderedArray \*ordered\_array, unsigned long int size)*** a cui viene passato un array già inizializzato e la dimensione corrente + l’eventuale incremento, di norma incrementeremo l’array con il doppio della dimensione attuale così da poterne favorire il tempo necessario per gli ulteriori inserimenti

in fine abbiamo ***void ordered\_array\_free(OrderedArray \*ordered\_array)*** che permette la corretta de allocazione di tutti i valori allocati all’interno della nostra strutta oltre ovviamente alla struttura stessa.

Il programma prende in input il percorso del file csv che si vuole analizzare e quindi riordinare, a questo punto possiamo utilizzare la funzione ***OrderedArray \*ordered\_array\_load\_csv(OrderedArray \*arr, FILE \* fp, int col)*** che ci permetterà di caricare l’intero csv all’interno della nostra struttura dati, il parametro ***int col*** definisce in quale colonna sono i valori su cui eseguire l’ordinamento così da poter scindere il record dal valore in esamine e rendere più immediata la comparazione senza dover ogni volta estrarre il valore da ***void \*\*record*** , la funzione utilizza il campo ***int (\*precedes) (void\*, void\*)*** del parametro ***arr*** per capire a quali valori castare la stringa proveniente dal csv su cui successivamente verrà applicata una delle tre funzioni *compare* per confrontare i valori.

Una volta terminata l’importazione del csv all’interno della struttura è possibile utilizzare separatamente le funzioni

* ***void binary\_insertion\_sort(****OrderedArray \*ordered\_array, long unsigned int low, long unsigned int high****)***
* ***ordered\_array\_merge\_sort(****OrderedArray \*ordered\_array, void \*\*temp, unsigned long low, unsigned long high****)***

La prima implementa la “fusione” tra una ***binary\_search*** e un ***insertion\_sort*** dalla ricerca binaria prendiamo la sua velocità logaritmica *O(log n)* di individuare un determinato elemento in un array ordinato con la variante di non cercare un elemento ma bensì la posizione corretta dove andare ad inserire l’elemento correntemente in esamine dalla parte di insertion\_sort della funzione, il problema principale di questa funzione e’ che la sua complessità rimane superiormente limitata dallo stesso insertion sort che pur sapendo dove andare ad inserire l’elemento corrente deve far fronte agli spostamenti necessari per posizionare l’elemento stesso arrivando ad avere una complessità massima O(n2),

La seconda funzione invece è un classico ***merge\_sort*** che quindi ha come complessità Θ(n log n)

In fine come richiesto dall’esercizio abbiamo la funzione

* ***merge\_binary\_insertion\_sort(OrderedArray \*arr, unsigned long subseq, int order, int save)***

Questa funzione cerca di trarre i benefici di entrambe le funzioni precedenti prendendo dalla prima l’ottima abilità per il riordinamento di array di ridotte dimensione e dalla seconda la parte ***merge*** responsabile della fusione delle sotto sequenze ordinate, come da ipotizzato le sotto sequenze piu piccole hanno una forte capacità di ammortizzare i tempi necessario al riordinamento a tal punto da rendere il mergesort classico peggiore con tutti i k < 19000

I dati presi in esamine sono riportati nel grafico sotto e sono stati calcolati mediante due ***timespec*** separate che andavano a catturare il tempo passato tra l’inizio della chiamata del ***ordered\_array\_merge\_sort*** (nel caso di un k <= 1) e la sua terminazione

o l’inizio del ***binary\_insertion\_sort*** (una volta suddiviso l’array attraverso le varie chiamate ricorsive della funzione ***merge\_binary\_insertion\_sort*** fino all’ottenimento di una sotto array <= k) e la sua terminazione questa volta andando però a calcolarne anche il tempo necessario alla funzione ***merge***

