Esercizio 1: algoritmo di Merge-BinaryInsertion Sort

Testo dell' Esercizio

Implementare una libreria che offre un algoritmo di ordinamento **Merge-BinaryInsertion Sort**.

Con **BinaryInsertion Sort** ci riferiamo a una versione dell'algoritmo *Insertion Sort* in cui la posizione, all'interno della sezione ordinata del vettore, in cui inserire l'elemento corrente è determinata tramite ricerca binaria.

Algoritmo Merge-BinaryInsertion

Il **Merge-BinaryInsertion Sort** è un algoritmo ibrido che combina *Merge Sort* e *BinaryInsertion Sort*. L'idea è di approfittare del fatto che il *BinaryInsertion Sort* può essere più veloce del *Merge Sort* quando la sottolista da ordinare è piccola. Ciò suggerisce di considerare una modifica del *Merge Sort* in cui le sottoliste di lunghezza k o inferiore sono ordinate usando il *BinaryInsertion Sort* e sono poi combinate usando il meccanismo tradizionale di fusione del *Merge Sort*.

Guida all'utilizzo

Qui di seguito, i comandi **make** per l'utilizzo del programma:

- **make all**: compila e crea l'eseguibile per il main e le unit tests
- **make tests**: compila, crea ed esegue il programma per le unit tests
- **make main**: compila, crea ed esegue il programma main

Decisioni di Sviluppo

Come detto in precedenza, il valore del parametro **k** assume un'importanza fondamentale nei **tempi di ordinamento** del file .csv dato come input.

In particolare, per **k=0** l'algoritmo ibrido *si comporterà esattamente come il Merge Sort classico*, mentre per **k>>0** *aumenta l'utilizzo del BinaryInsertion Sort*, proprio perché aumenta la dimensione delle sottoliste da ordinare.

Detto questo, qui sotto i passi che abbiamo seguito per trovare e raffinare un valore ottimo del parametro k!

k = sqrt(size)

Per trovare il valore di parametro ideale, siamo partiti con l'idea di utilizzare la **funzione di *radice quadrata*** *(sqrt(), dalla libreria standard <math.h>)* **sulla lunghezza dell'array da ordinare**. Un valore di partenza sufficientemente piccolo insomma, che ci ha permesso inoltre di avere un valore variabile e poter eseguire ***test su array di qualsiasi lunghezza***.

Tempo di ordinamento in media ottenuto: **8.5-14 secondi.**

k = ln(size)

Dopo diversi prove con il valore precedente abbiamo deciso di passare ad un'altra funzione per mantenere la flessibilità del parametro, ma **asintoticamente più piccola**. Parliamo della **funzione di logaritmo naturale** *(log(), dalla libreria standard <math.h>)* **sulla lunghezza dell'array da ordinare**. Essendo la lunghezza delle sottoliste da ordinare più piccola abbiamo notato un *leggero miglioramento*, ma quello che davvero ci ha fatto accorgere il nuovo valore e' stata l'importanza del **bilanciamento dei 2 algoritmi di sorting**: diminuendo di molto il valore di k, infatti, e' aumentato di conseguenza il lavoro svolto dalla suddivisione da parte del *Merge Sort*, in complemento al maggior lavoro svolto da parte del *BinaryInsertion Sort* nel caso della radice quadrata. In conclusione, **i tempi ottenuti sono molto simili al valore precedente**.

Tempo di ordinamento in media ottenuto: **7.5-13 secondi.**

k = costanti

Forti delle considerazioni fatte sopra, il nostro ultimo step è stato **raffinare il parametro con la precisione di valori costanti**, un valore di k ad-hoc specifico per il file records.csv datoci in consegna. In particolare un *valore compreso fra ln(size) e log(size)*, quindi un finestra di valori a noi già nota. Qui di seguito, 4 valori di k costanti con i rispettivi tempi medi.

(k = 3000) Tempo di ordinamento in media ottenuto: **9-14 secondi.**

(k = 300) Tempo di ordinamento in media ottenuto: **6-10 secondi.**

(k = 1500) Tempo di ordinamento in media ottenuto: **7-11 secondi.**

(k = 600) Tempo di ordinamento in media ottenuto: **6-10 secondi.**

All’interno del programma e’ stato deciso di tenere **k = 300**, per prestazioni lievemente migliori nei test effettuati.