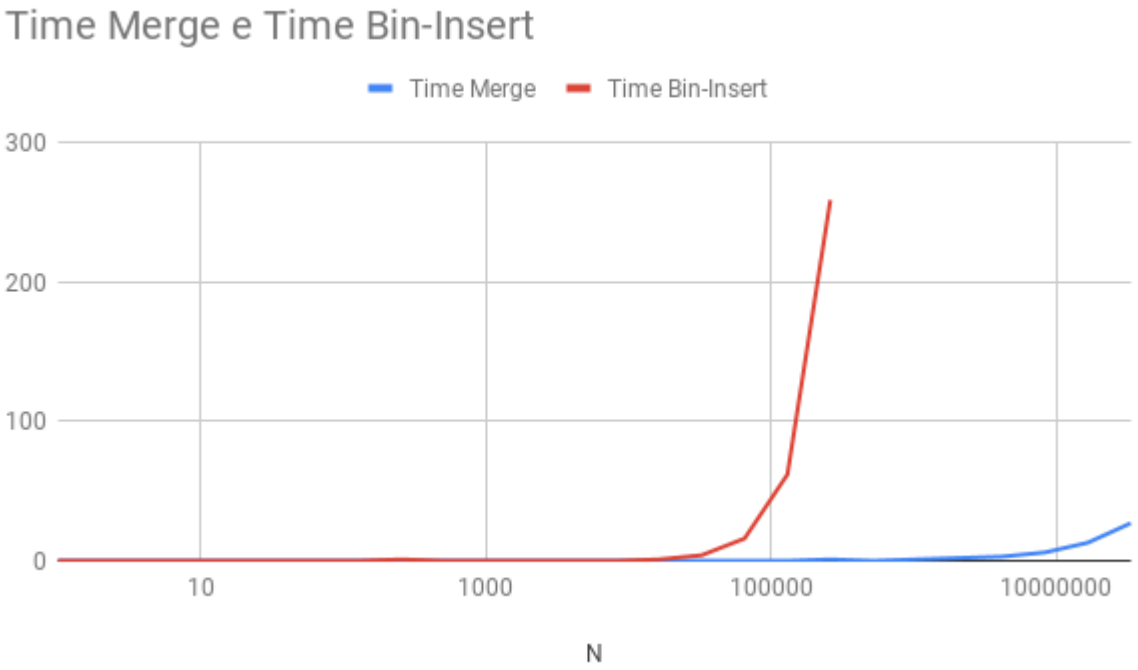
In questo esercizio è stato implementato il “Merge-BinaryInsertion Sort”, un tentativo di fusione tra il Binary Search, l’Insertion Sort ed il Merge Sort. La parte chiamata BinaryInsertion Sort non è altro che l’insertion sort, con però l’aggiunta che la ricerca nel sottoarray ordinato di dove mettere il prossimo elemento dell’array intero viene fatta usando il Binary Search. Infine, questo BinaryInsertion Sort viene usato nel caso base del Merge Sort, variando anche quando questo caso base viene raggiunto. Infatti, differenza del Merge Sort classico, dove il caso base viene raggiunto quando si ha un sottoarray di dimensione 1, in questo caso la dimensione del sottoarray del caso base, caso in cui verrà chiamato il BinaryInsertion Sort, è controllata da un parametro K. L’obiettivo dell’esercizio è variare K, tentando di trovare un valore “ideale”, che fonda al meglio i due algoritmi di sorting. La speranza è che, per array sufficientemente piccoli, il BinaryInsertion Sort risulti più veloce del Merge Sort, e l’obiettivo sarebbe impostare K con il valore dell’array più grande per il quale il BinaryInsertion Sort è più veloce del Merge Sort.

La premessa di tutto ciò è il fatto che per array sufficientemente grandi, il BinaryInsertion Sort sia sempre (molto) più lento del Merge Sort. Ciò è vero perché la relazione tra il tempo di esecuzione di BinaryInsertion Sort e il numero N di elementi dell’array da ordinare è O(N^2) mentre quello del Merge Sort è noto essere O(Nlog(N)).

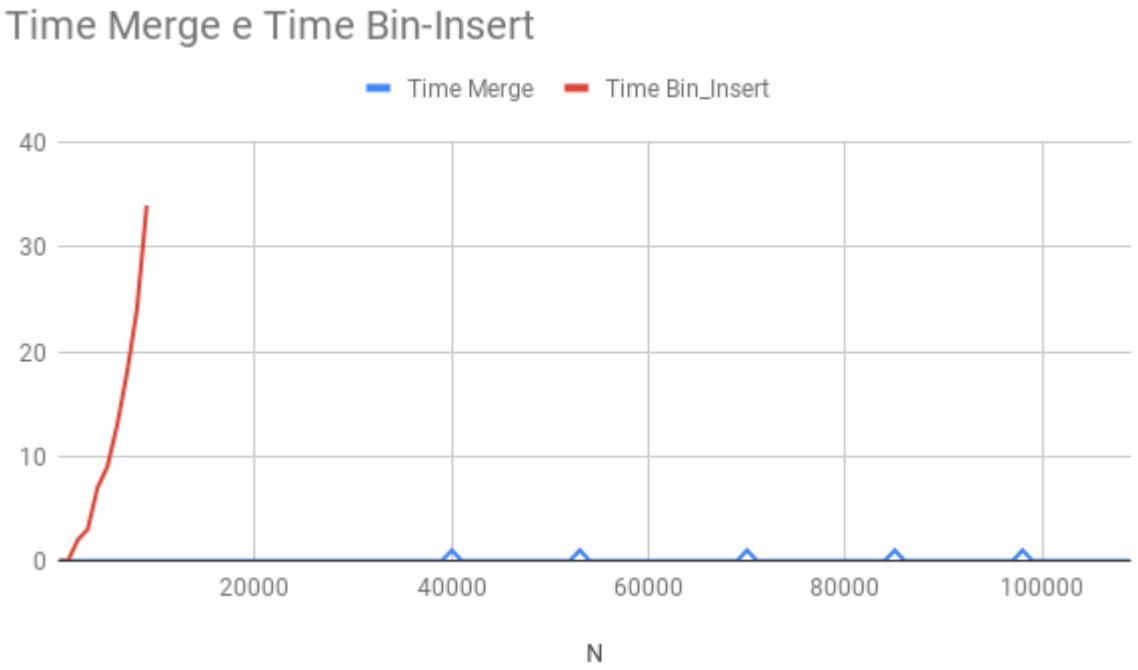
Un’intuizione del perché BinaryInsertion Sort sia O(N^2) è che l’algoritmo scorre N-1 elementi, e per ognuno di questi sposta, in media, N/2 elementi per inserire ognuno di questi nel sottoarray ordinato, arrivando quindi ad avere O(N-1)*O(N/2)=O(N)*O(N)=O(N^2).

Per testare che anche in pratica, per array sufficientemente grandi il BinaryInsertion Sort fosse di molto peggiore al Merge sort, è stata creata la funzione random_arr, che prende in input la lunghezza di un array e da come output un array di double random lungo quanto la lunghezza presa in input. Usando tale funzione, è stato realizzato un ciclo infinito che chiamava la funzione random_arr con successive potenze di 2, in modo da avere delle differenze notevoli tra i due algoritmi di sorting in pochi passi, ed su tali array sono stati, appunto, invocati i due algoritmi di sort (in maniera indipendente l’uno dall’altro), e sono stati scritti su file i tempi di esecuzione. Infine è stato creato un grafico basato sul file di output, usando una scala logaritmica sull’asse contenente N (numero di elementi dell’array), in modo da controbilanciare la crescita esponenziale di N, e vedere come sarebbero le tendenze dei due algoritmi con N lineare:



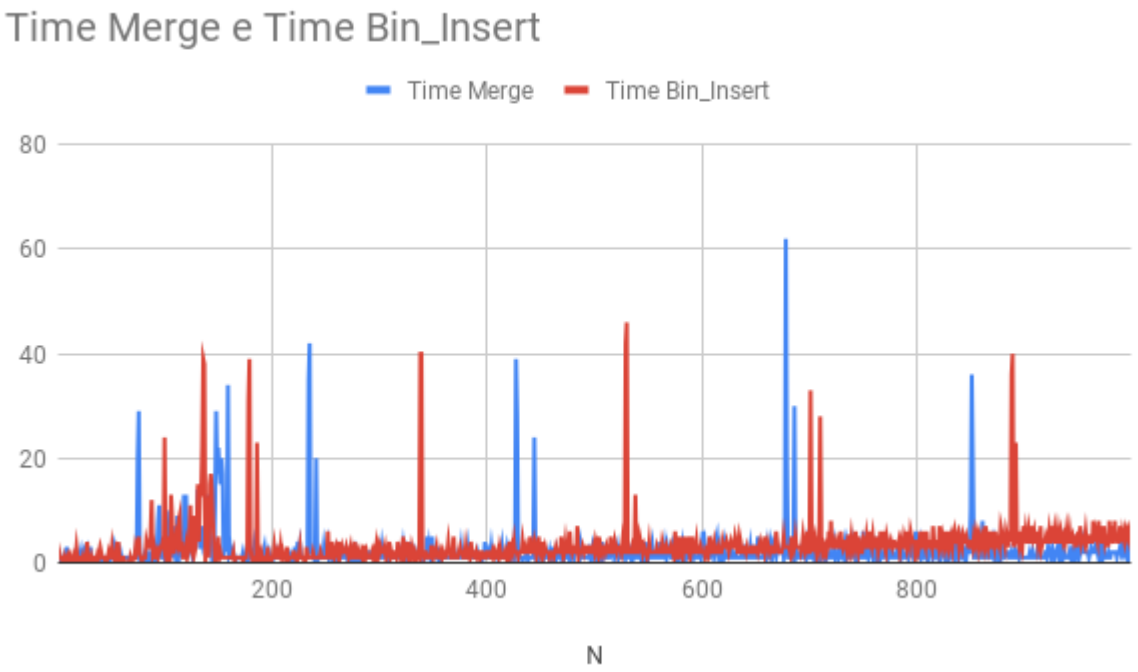
(il tempo è misurato in secondi)

Da questo grafico si vede anche che, però, nel range da 0 a 50 000 i due algoritmi sono praticamente sovrapposti, per cui può darsi che, aumentando la “granularità” della scala, aumentando ad ogni passo N di 1000, si potrebbe notare che il BinaryInsertion è effettivamente migliore del Merge, anche se, magari, di poco. Tentando di fare ciò, però, misurando il tempo in secondi porta al seguente, non troppo inaspettato risultato:

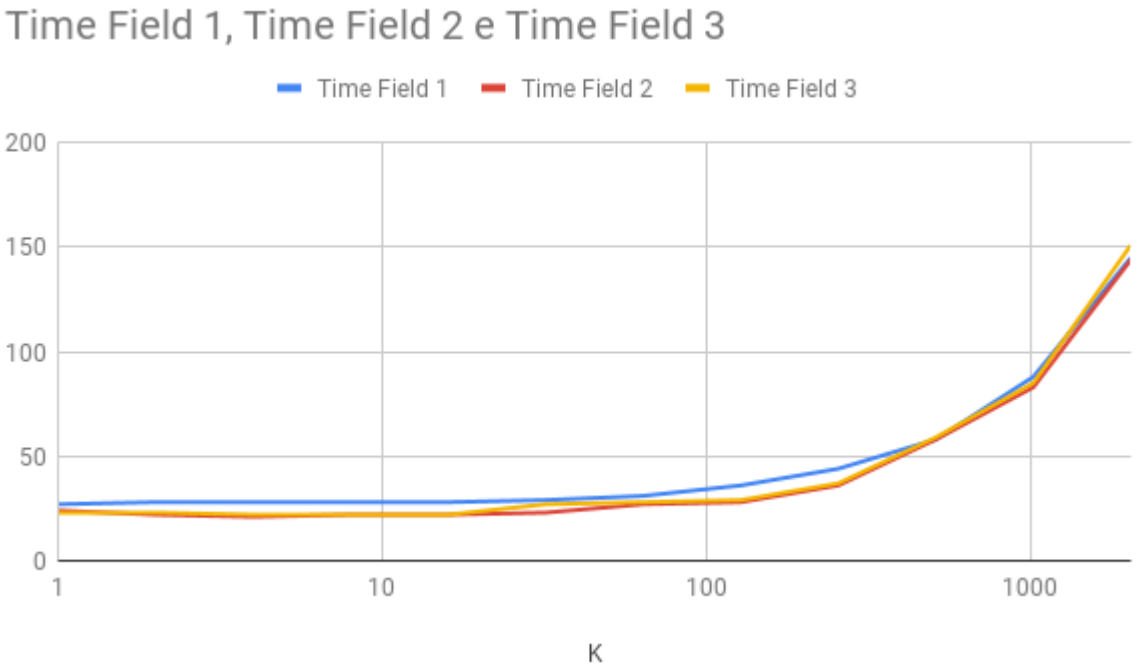


Il Merge Sort è praticamente piatto, fermo su 0 secondi, mentre il BinaryInsertion Sort cresce al tasso parabolico atteso. Il punto in cui i due sembrano di nuovo sovrapporsi a

questo punto sembra stare nel range da 0 a 1000, ma il problema è che il tempo per entrambi, in quel range, è di 0 secondi. Volendo dare una risposta più soddisfacente al quesito è stato deciso di usare la funzione clock() di time.h, invece che time(), ed i risultati su N da 1 a 1000 sono i seguenti:



Tralasciando i vari spike casuali, dovuti probabilmente all'interferenza di altri processi concorrenti, si può notare come persino a questa risoluzione temporale i due algoritmi sono al massimo identici, ma in nessun punto il BinaryInsertion Sort è più veloce del Merge Sort. A conferma finale di questo fatto, usando il file di input, che ci è stato fornito per questo esercizio, e variando K, da 1 (algoritmo che si riduce al Merge Sort) in poi, si nota come per un po' i tempi di ordinamento dei tre campi rimane pressoché immutato, per poi iniziare, prevedibilmente a salire:



(il tempo è misurato in secondi)

Per motivi di convenienza, anche in questo caso è stato scelto di incrementare K in maniera esponenziale, e di poi rappresentarlo su grafico in scala logaritmica. L'impennata parabolica, che si può notare, risulta dal BinaryInsertion Sort che diventa sempre più dominante. Si può infine evincere che il valore ideale di K sia qualunque valore tra 1 e 100, da questo e anche dalle considerazioni fatte in precedenza, per cui si può anche constatare, in conclusione, che l'algoritmo più efficiente sarebbe il Merge Sort puro, eliminando l'overhead della chiamata del BinaryInsertion Sort.