Relazione Algoritmi e Strutture Dati

Eduard Antonovic Occhipinti, Iman Solaih, Marco Molica

May 12, 2022

Contents

Esercizio 1

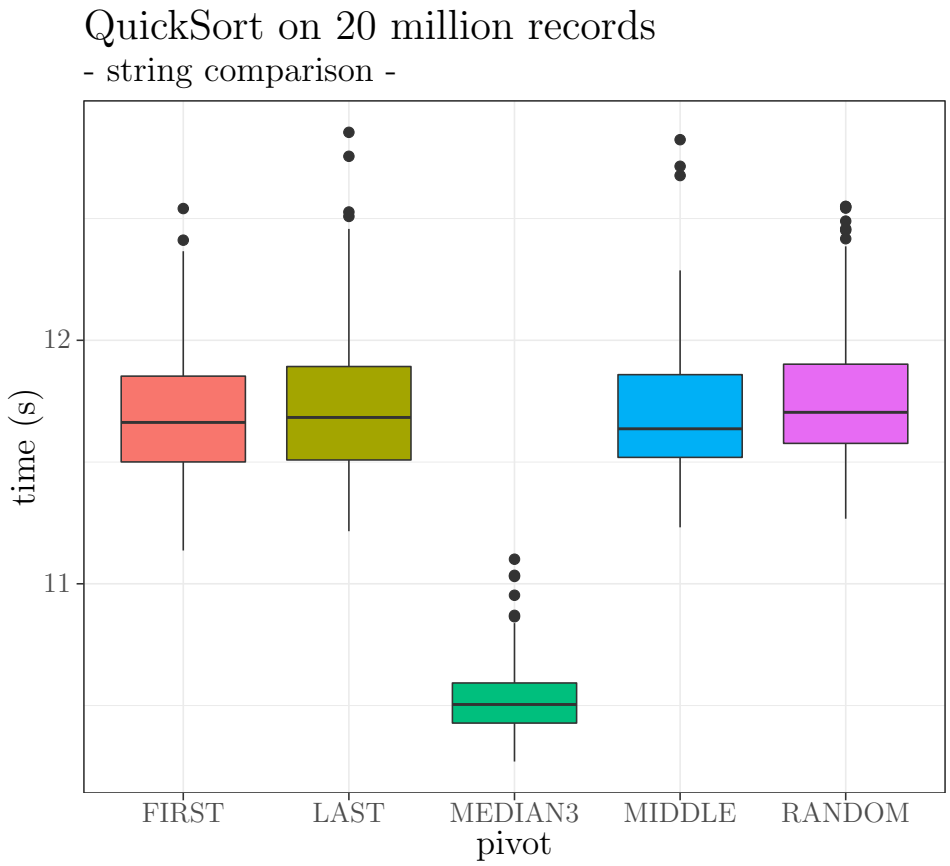
1 Quick Sort

Il `quick_sort()` è un algoritmo che ordina una collezione partendo da un pivot, questo può essere scelto in vari modi, e in base a quale viene scelto il tempo di sorting varia. Il `quick_sort()` utilizza `_part()` per scegliere il pivot prima di chiamare `partition()` per dividere gli elementi del range selezionato in un sottoinsieme di elementi maggiori e uno di elementi minori del pivot la cui posizione finale viene restituita dal metodo.

Premessa: nella seguente relazione analizzeremo solo i dati raccolti su records favorendo il primo `field` nell'ordinamento, i dati per i restanti due field sono equivalenti ma con costanti minori.

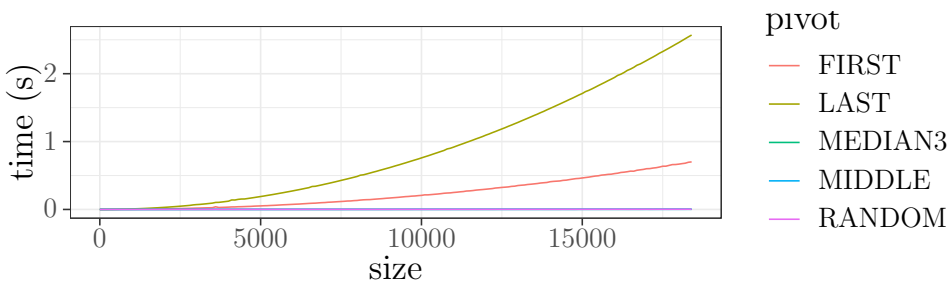
1.1 Impatto della scelta del pivot nel quick sort

La tabella sottostante riporta il tempo impiegato ad ordinare un array di 20 milioni elementi di tipo `struct Record`

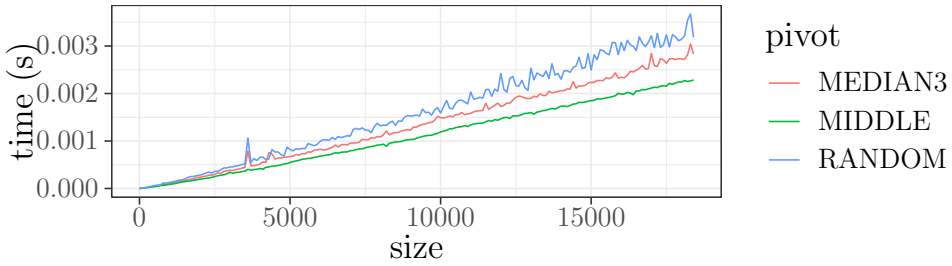


3000 samples, 1000 for each field prioritized, 200 for every pivot.
The records were randomly shuffled at every run.
Test conducted on an intel i5-11400F CPU, 16GB RAM, Ubuntu 22.04.

La scelta del pivot diventa importante quando l'array in input risulta già parzialmente o totalmente ordinato. Il grafico sottostante riporta il tempo impiegato da `quick_sort()` per scorrere un array già ordinato. Come ci aspettiamo, l'algoritmo degenera ad $O(n^2)$, sia `LAST` che `FIRST` generano un grafico esponenziale ma con costanti diverse, fosse l'array ordinato in ordine inverso ci aspettiamo il comportamento opposto tra questi due.



Concentrandoci in particolare sui pivot `median of 3`, `random` e `middle`, possiamo notare che per questi il tempo cresce in maniera costante.



In particolare `MIDDLE` è chiaramente il pivot con perfomance migliori, il risultato è quello aspettato considerando che in questo contesto qui, `partition()` non deve praticamente effettuare `SWAP`. Possiamo comunque notare che il pivot `RANDOM` si comporta discretamente, con una variabilità maggiore rispetto agli altri. `MEDIAN3` finirà per sceglire lo stesso pivot di `MIDDLE` e quindi il tempo aggiuntivo è interamente introdotto dall'overhead causato dal confronto dell'elemento centrale con il first e last dell'array.

1.2 Fallback a Insertion Sort

Quando il `quick_sort()` lavora su un range sufficientemente piccolo, è più efficiente utilizzare il `insert_sort()`. Il range di cutoff è stato impostato a 8 elementi.

1.3 Scelta del partition

Nel nostro dataset ogni `record` è virtualmente univoco, la partition di Lomuto si comporta quindi molto bene ed anzi, secondo i nostri test, anche meglio di quella di Hoare, nonostante quest'ultima infatti effettua meno `SWAP`, è più complessa a livello di codice e causa alla CPU una probabilità più alta di branch misprediction.

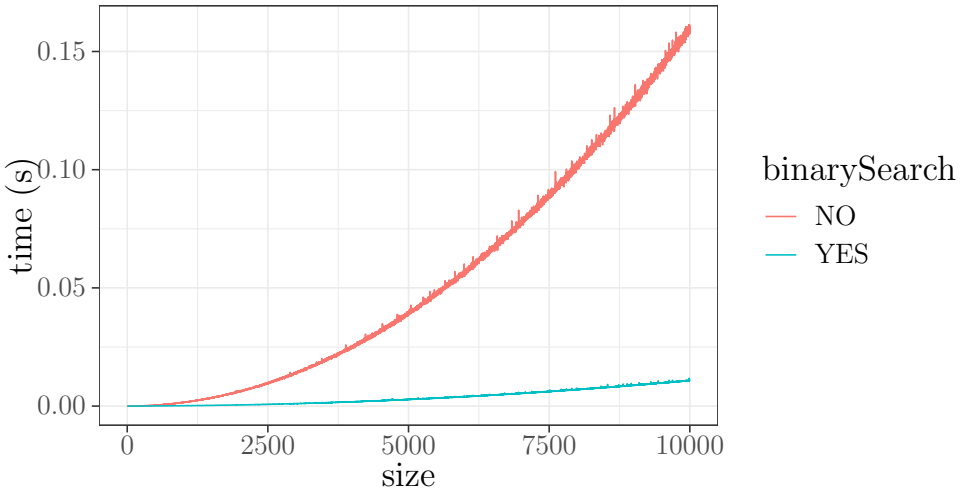
```
template <typename T>
int partition_lomuto(T array[], int left, int right)
{
    T pivot = array[right];
    int i = left - 1;
    for (int j = left; j < right; j++){
        if (array[j] <= pivot) {
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[right]);
    return i + 1;
}
```

Nel caso però si lavorasse su un dataset con una quantità importante di elementi duplicati, la partition di Hoare inizia subito ad avere perfomance molto migliori, una buona alternativa è anche una partition di Lomuto modificata in maniera tale da restituire due indici, dividendo quindi il subarray in tre parti: elementi minori, uguali e maggiori del pivot.

```
template <typename T>
int partition_hoare(T array[], int left, int right)
{
    T pivot = array[(left + right) / 2];
    int i = left - 1;
    int j = right + 1;
    while (1) {
        do {
            i++;
        } while (array[i] < pivot);
        do {
            j--;
        } while (array[j] > pivot);
        if (i >= j) {
            return j;
        }
        swap(&array[i], &array[j]);
    }
}
```

2 Binary Insertion Sort

‘ Essendo l’algoritmo di complessità $O(n^2)$, non ci aspettiamo che finisca in tempi sensati l’ordinamento dei 20 milioni di records, facendo due calcoli sui nostri computer dovrebbe metterci approssimativmaente 2 anni. Nel seguente schema possiamo però notare come la ricerca binaria del punto di inserimento migliori notevolmente la costante di tempo.

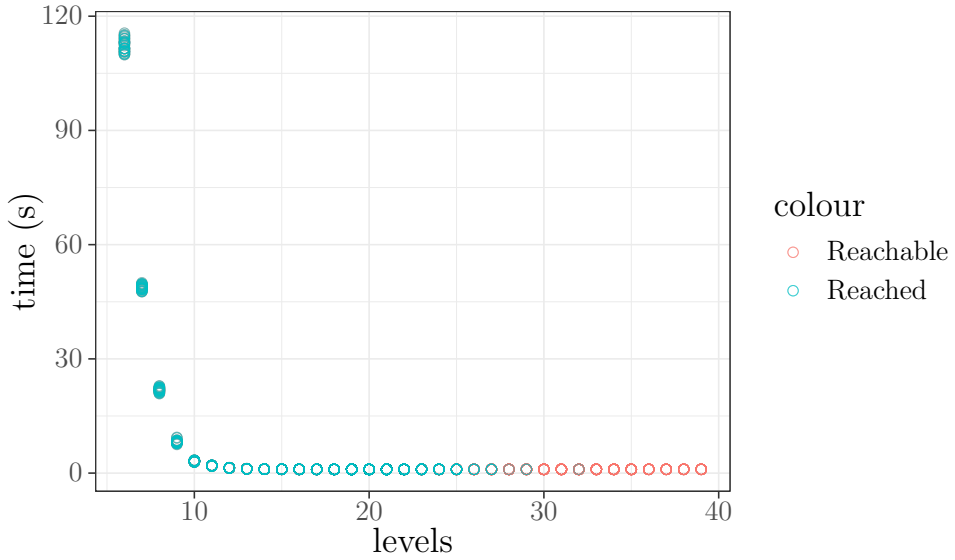


30000 samples for each algorithm, 10000 for each field prioritized, wiht increments of 1

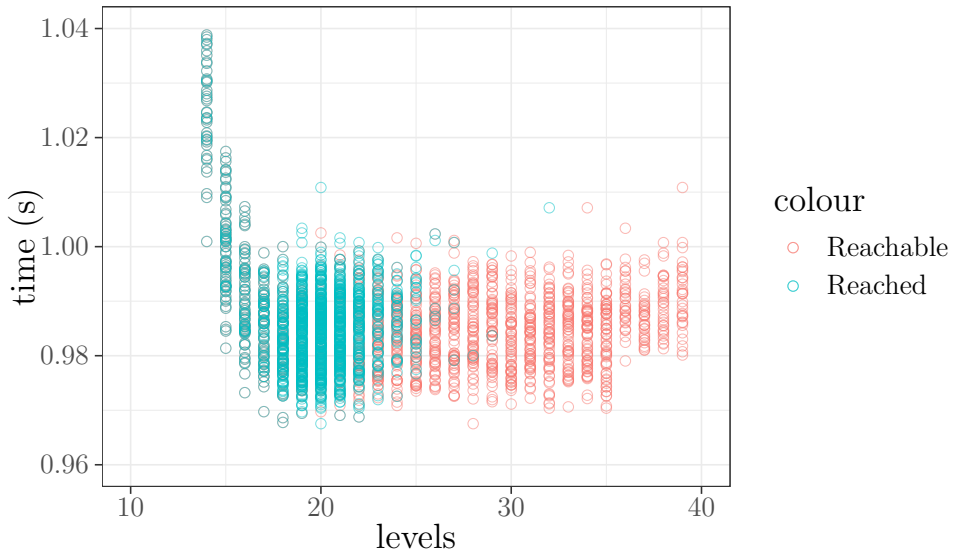
Esercizio 2

3 Skip List

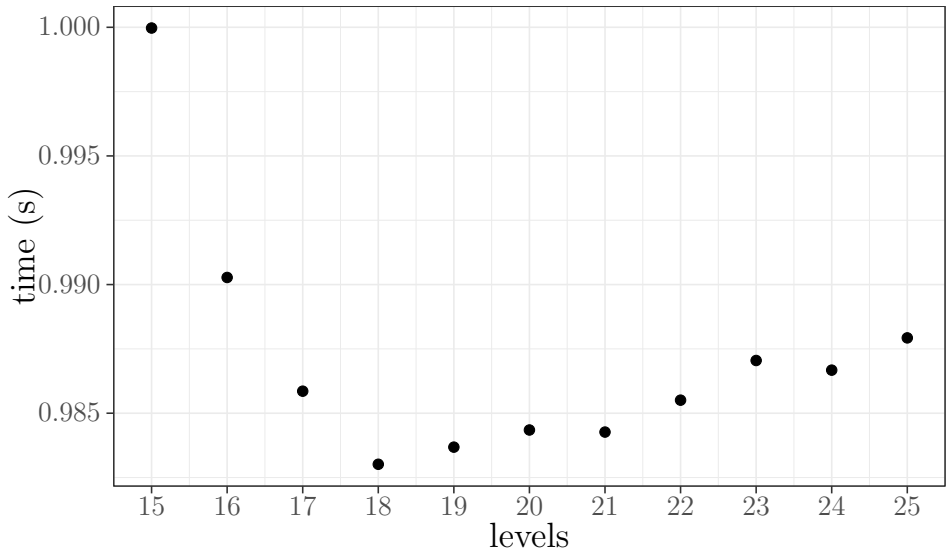
Dagli esperimenti effettuati i risultati dell'insertion mostrano come all'aumentare dei livelli il tempo di inserimento decresce in maniera esponenziale.



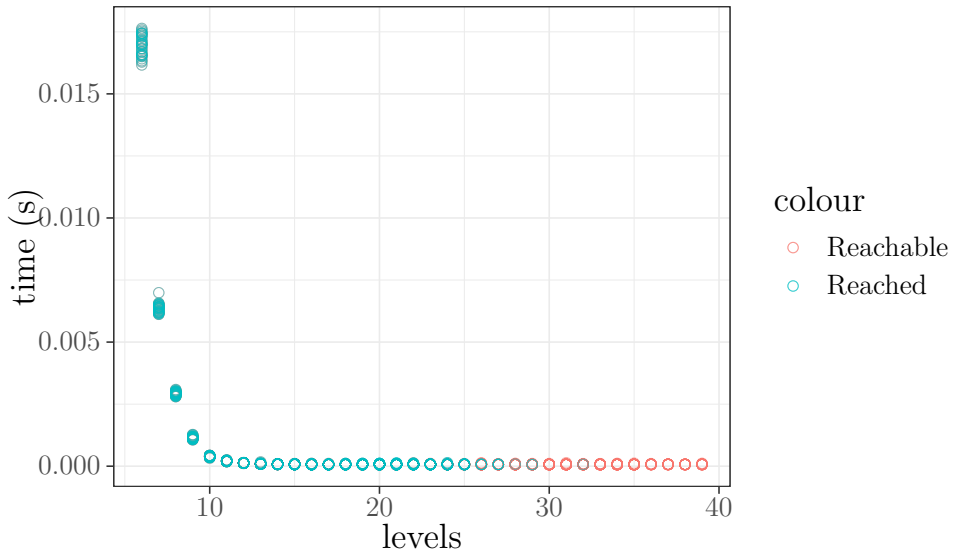
Dal grafico si nota come la distribuzione dei livelli raggiunti è concentrata attorno a 20, inoltre dal livello 30 in poi i livelli non vengono quasi mai raggiunti, difatti la probabilità di raggiungere ogni livello è $\frac{1}{2^n}$, il livello 32, il massimo raggiunto, aveva probabilità 2.32×10^{-10}.



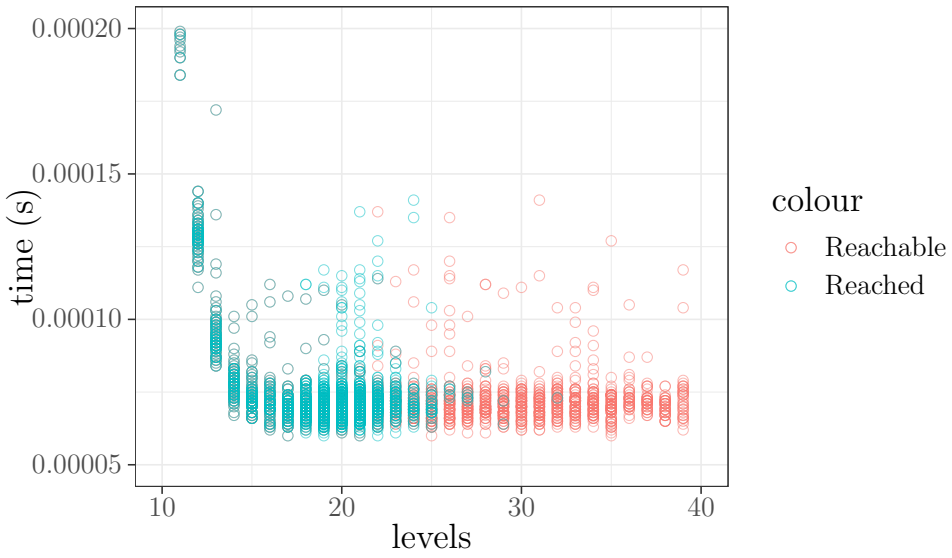
Bla bla bla facendo un grafico delle medie dei tempi di inserimento notiamo che 18 è il numero ottimale di livelli



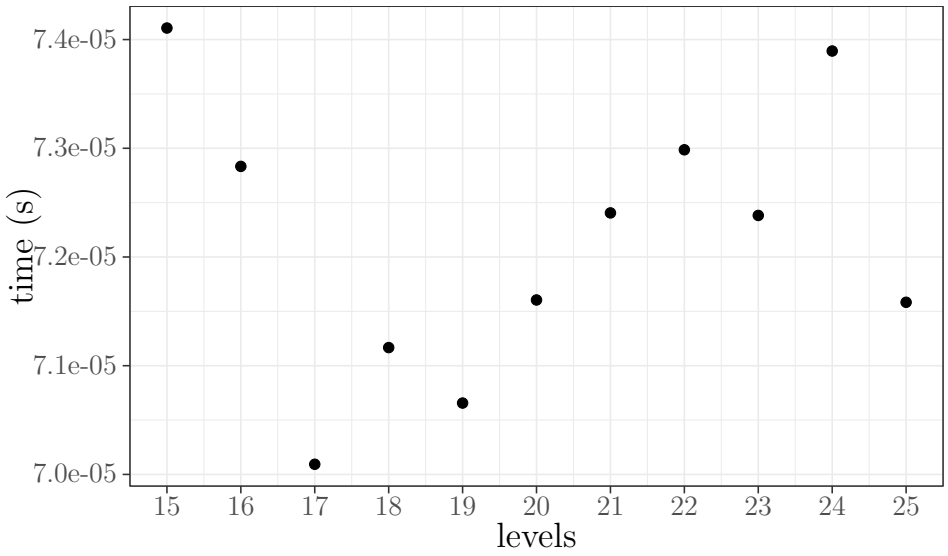
Bla bla bla search time decresce in maniera imporante



Bla bla bla in particolare zoommando sui livelli più di interesse ci rendiamo conto che la distribuzione è concentrata attorno a 19



Bla bla bla facendo un grafico delle medie dei tempi di inserimento notiamo che 17 è il numero ottimale di livelli



Sorprendentemente il numero ottimale di livelli non coincide esattamente con $ln(n)$

Esercizio 3

4 Minimum Heap

Esercizio 4

5 Graph