UNIVERSITÀ
DEGLI STUDI
DI TORINO

Relazione Algoritmi e Strutture Dati

Eduard Antonovic Occhipinti, Iman Solaih, Marco Molica

June 13, 2022

Contents

Esercizio 1

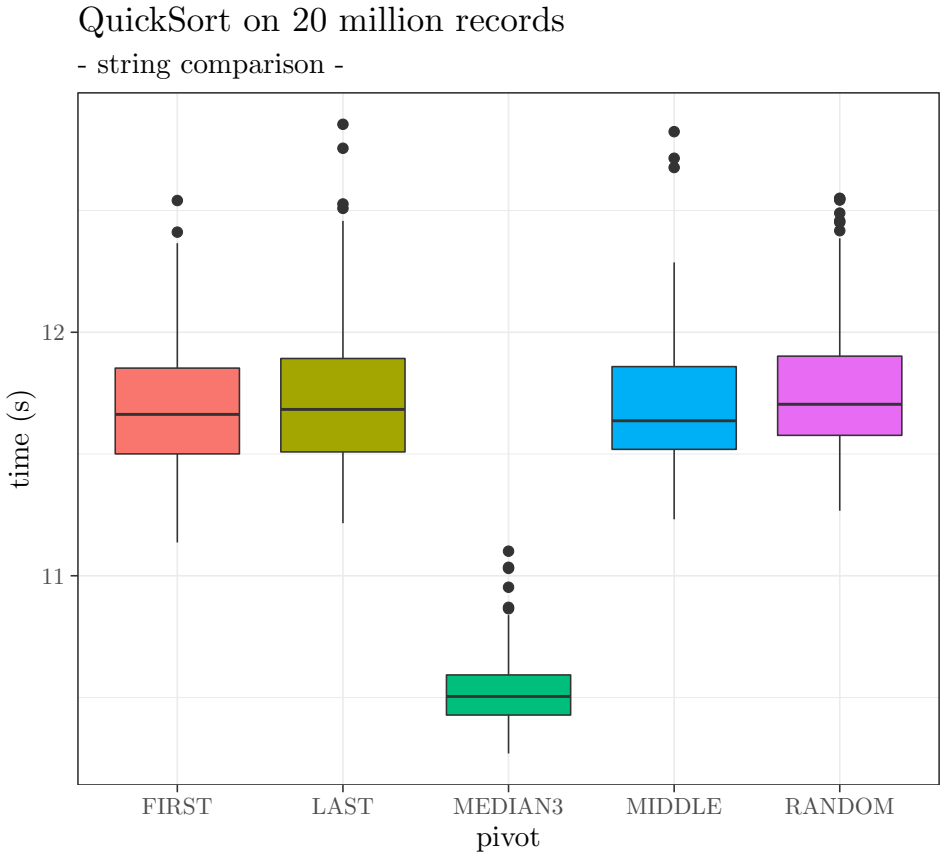
1 Quick Sort

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.

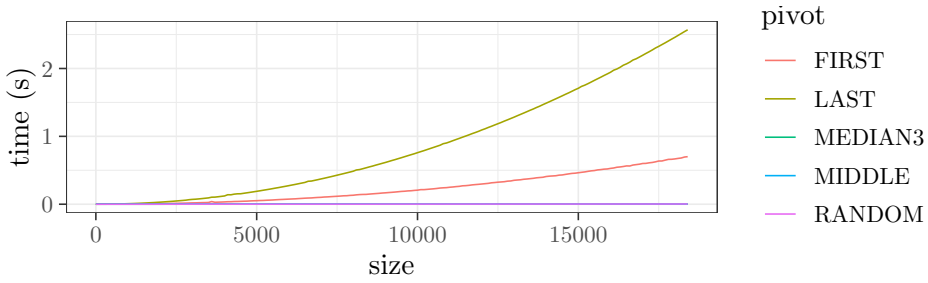
1.1 Impatto della scelta del pivot nel quick sort

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

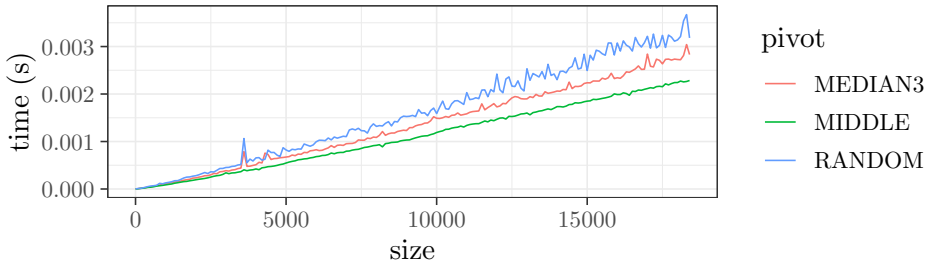


3000 samples, 1000 for each field prioritized, 200 for every pivot.
The records were randomly shuffled at every run.
Test conducted on an intel i5-11400F CPU, 16GB RAM, Ubuntu 22.04.

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 $O(n^2)$ e sia `LAST` che `FIRST` generano un grafico esponenziale ma con costanti diverse: fosse stato l'array ordinato in ordine inverso ci saremmo aspettati 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 performance 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 scegliere lo stesso pivot di `MIDDLE` e quindi il tempo aggiuntivo è interamente introdotto dal 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.

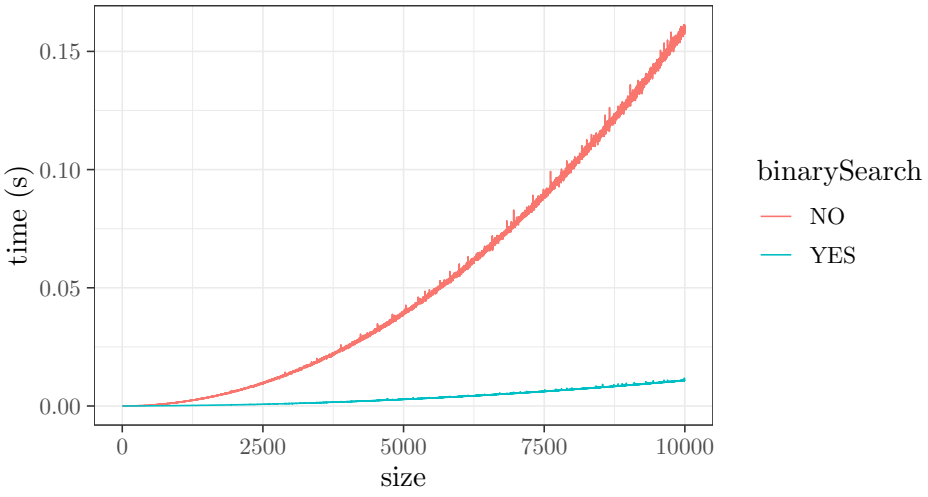
```
template <typename T>
int partition_lomuto(T array[], int left, int right)
{
    T pivot = array[right];
    int i = left - 1;
    for (int j = left; j < right; j++){
        if (array[j] <= pivot) {
            i++;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1],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

Nel caso però si lavorasse su un dataset con una quantità importante di elementi duplicati, la partition di Hoare inizierebbe 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 l’ordinamento dei 20 milioni di records finisca in tempi sensati: facendo due calcoli sui nostri computer dovrebbe impiegarci approssimativamente 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

3.1 Algoritmo per scegliere il numero di livelli

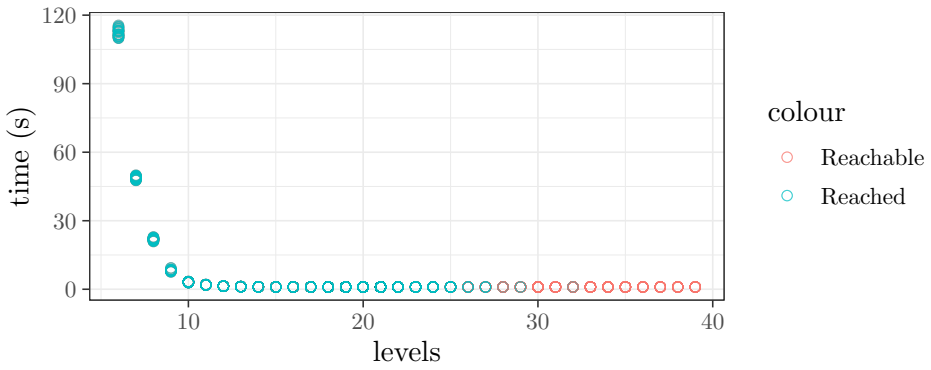
Dalla funzione utilizzata per scegliere il massimo numero di livelli per un dato nodo possiamo notare che la probabilità di scegliere un livello k è $1/2^{k-1}$, con il primo livello classificato come $k = 1$.

```
1    uint32_t random_level()
2    {
3        int lvl = 1;
4        while(rand() % 2 && lvl < MAX_HEIGHT) lvl++;
5        return lvl;
6    }
```

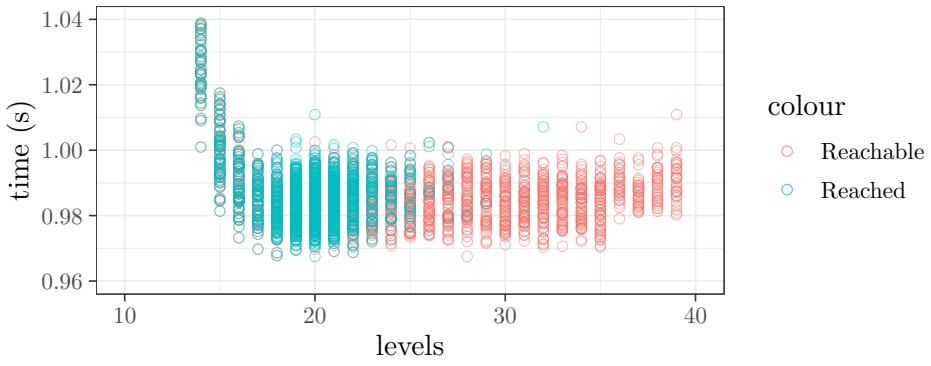
Ci aspettiamo quindi che il numero massimo di livelli, probabilisticamente, sia limitato da $O(log_2(n))$. Il vantaggio che possiamo notare nell'utilizzo di questo algoritmo deriva del fatto che la ricerca in una lista ordinata a due livelli, l'utilizzo di $\sqrt{n}$ livelli nel layer superiore al primo è ottimale se quest'ultimo è costituito da n nodi.
La time complexity in questo caso si riduce a $O(2\sqrt[2]{n}) = O(\sqrt{n})$.
Nella skiplist il numero di livelli si stabilizza su $O(log_2(n))$ quindi il costo diventa $log_2(n) \times \sqrt[log_2(n)]{n} = 2log_2(n)$, la complessità è quindi $O(log(n))$.

3.2 Analisi dei tempi di inserimento

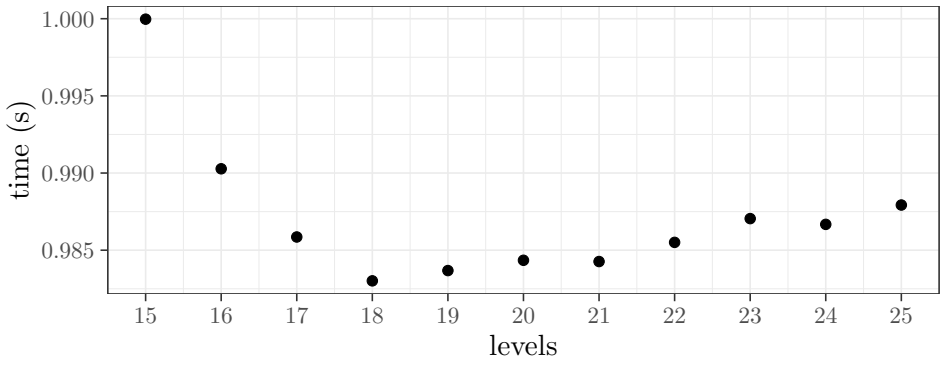
Dagli esperimenti effettuati i risultati dell'insertion mostrano come all' aumentare dei livelli il tempo di inserimento decresce in maniera logaritmica.



Ingrandiamo il grafico in maniera tale da meglio visualizzare la distribuzione dei tempi nei livelli più di nostro interesse.
Notiamo 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 $\frac{1}{2^n}$, e il livello 32, il massimo raggiunto, aveva probabilità 2.32×10^{-10}.

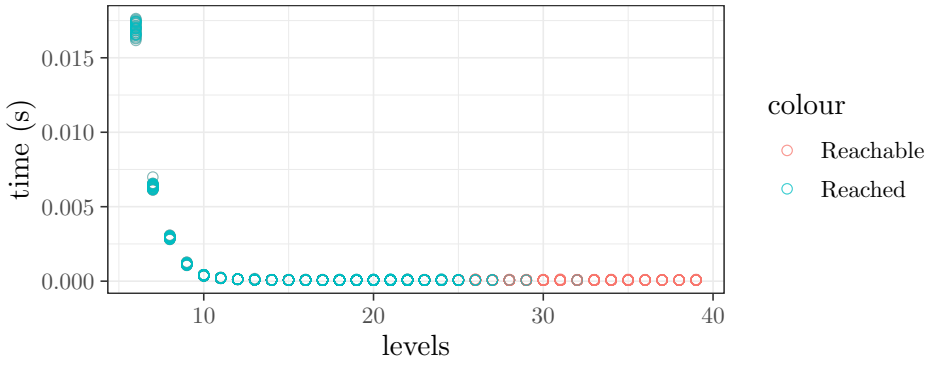


Facendo un grafico delle medie dei tempi di inserimento notiamo che 18 è il numero ottimale di livelli

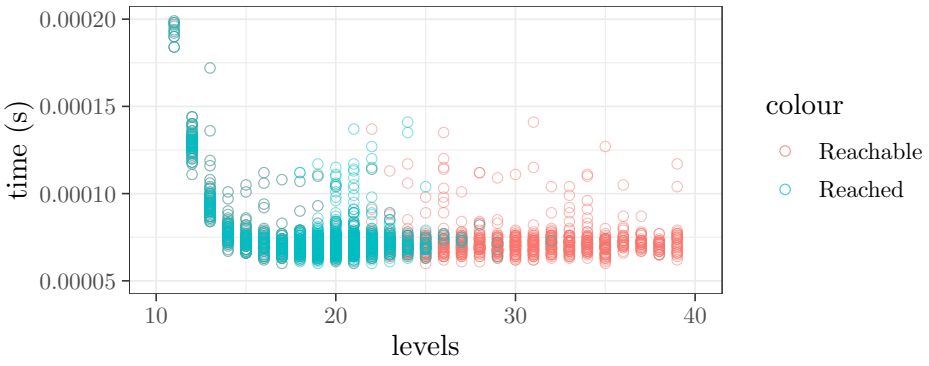


3.3 Analisi dei tempi di ricerca

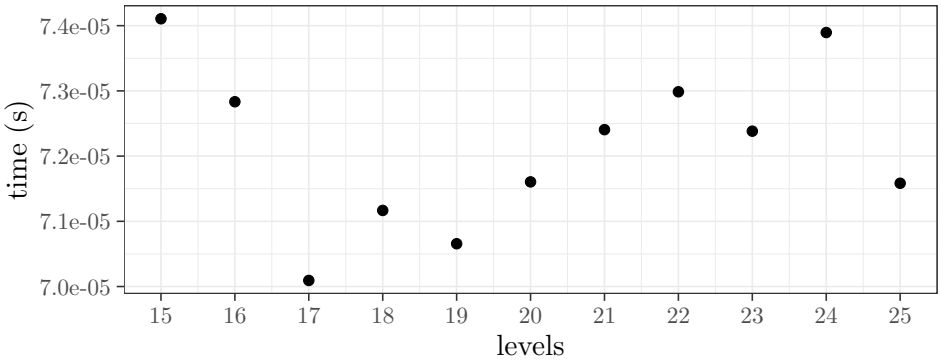
Analizzando i tempi di ricerca notiamo come all'aumentare dei livelli il search time decresca in maniera importante



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



Facendo un grafico delle medie dei tempi 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

Per garantire la complessità in $O(1)$ della restituzione del `left`, `right` e `parent` di un elemento, a partire dal valore dello stesso, abbiamo usato una struttura dati di supporto, una `HashMap<>`, che memorizza i valori degli elementi e li associa ai relativi indici nell' `ArrayList` che rappresenta il nostro heap.

Abbiamo deciso di creare anche un'interfaccia `PriorityQueue<>` che è la Abstract Data Structure sulla quale di basa il `MinHeap<>`

Esercizio 4

5 Graph

Abbiamo deciso di considerare il grafo diretto come la struttura dati base di un generico grafo, i grafi di tipo indiretto possono infatti essere visti come grafi diretti nei quali ad ogni arco viene associato anche un arco opposto. Abbiamo deciso quindi di creare una classe `UndirectGraph<>` che estende `DirectGraph<>`, con costruttori `protected`, ed una classe `Graph<>` che incapsula i due.

Vi sono diversi modi di rappresentare un grafo G(V, E) a livello software, i due metodi più intuitivi sono quelli della lista di adiacenza e della matrice di adiacenza. La nostra implementazione sfrutta invece una mappa di vertici associati a mappe di vertici associati al `weight` dell'arco.
Concettualmente questa `Map<V, Map<V, E>>` può essere vista come una lista di adiacenza ma offre in realtà tutti i vantaggi di una matrice di adiacenza.

Per aiutare nell'inizializzazione di un grafo, abbiamo deciso anche di creare una classe `GraphBuilder<>` che sfrutta il design pattern `Builder`.

6 Dijkstra

Abbiamo implementato l' algoritmo di Dijkstra nella classe `GraphHelper<>`, che contiene tutta una serie di metodi statici che possono essere di aiuto nell'utilizzo di un grafo.
Abbiamo implementato l'algoritmo di Dijkstra in maniera quasi completamente generica, assumendo però che l'etichette degli archi possano essere solamente di tipo numerico: il loro tipo infatti è limitato a `E extends Number`, principalmente per via dell'impossibilità in `Java` di effettuare l'override degli `operator`.
La funzione chiede che in input gli venga fornito, oltre all'oggetto grafo e l'elemento source, anche un `Comparator<? super E>` che permetta di effettuare la comparison tra i `weight` degli archi ed un "max" che ci permetta di capire qual'è il valore massimo di E (Ad esempio per `Integer` basta inserire `Integer.MAX_VALUE`). Questo valore è quello al quale inizializziamo i vertici del grafo.
La priority queue utilizzata per tenere traccia delle distanze tra `source` e i vari

vertici è il `MinHeap<>`, i cui elementi sono dei
`Node<vertex, distance from source>`. Per memorizzare i predecessori e le distanze abbiamo deciso di usare delle `HashMap<>`.
L'algoritmo restituisce un `Pair<>` che è una coppia di elementi nel quale il primo rappresenta il percorso minimo tra `source` e tutti i nodi del grafo, e il secondo contiene la rispettiva distanza per ogni altro nodo.