Università degli Studi di Udine

Scienze Informatiche

Laboratorio di Algoritmi e Strutture Dati

Anno: 2019/2020

Riccardo Ferrarese

mat. 140491

ferrarese.riccardo@spes.uniud.it

Contents

1 Introduzione

Il seguente elaborato propone tre diverse soluzioni per il problema del calcolo del k-esimo elemento più piccolo in un vettore non ordinato di interi.

Le implementazioni sono state realizzate in linguaggio *C++*. Grafici, varianza e tempi medi totali sono stati calcolati con l'ausilio di codice scritto in *R*.
Nelle seguenti sezioni verranno presentate ed analizzate le soluzioni.
L'analisi dei tempi cerca di ridurre al minimo le variazioni nei tempi calcolati facendo eseguire più volte l'algoritmo su un input fissato per raggiungere una determinata precisione nella misurazione.

2 Quick Select

Algoritmo

Si tratta di una variante dell'algoritmo di ordinamento *quickSort*, in cui ogni chiamata ricorsiva su un intervallo [i,j] del vettore fornito in input termina in tempo costante ogniqualvolta il parametro k non sia contenuto nell'intervallo [i,j].
Dato in input un vettore di numeri *numbers* che va da [1..*dim*] e la sua dimensione, la procedura **quick_select** utilizza **partition** (con pivot casuale) per partizionare il vettore attorno al pivot.
Così facendo se il pivot si trova ad un indice più grande di *k*, si continua la ricerca nella partizione di destra.
Viceversa se *k* è più grande dell'indice del pivot si continua la ricerca nella partizione di sinistra.

Pseudocodice

Algorithm 1 Quick Select - first solution

```
function QUICK_SELECT(numbers, start, end, k)
    if k ≤ start and k ≥ end then
        pivot_index ← partition(numbers, dim)
        if pivot_index == k then
            return numbers[k]
        else if pivot_index > k then
            return quick_select(numbers, start, pivot_index-1, k)
        else return quick_select(numbers, start, pivot_index-1, k)
        end if
    end if
    return null
end function
```

Di seguito lo pseudocodice della procedura **Partition**, il quale preso un

vettore di numeri interi e un pivot randomico (in questo caso l'ultimo elemento del vettore) partiziona gli elementi più piccoli del pivot a sinistra e quelli più grandi a destra.
Come dimostrato a lezione *Partition* ha complessità $\mathcal{O}(n)$.

Algorithm 2 Partition_random

function $\textsc{Partition}(numbers, start, end)$
  $pivot \leftarrow numbers[end]$
  for $j \leftarrow start,\ j \leq end,\ j++$ **do**
    if $numbers[j] \leq pivot$ **then**
      $i \leftarrow i + 1$
      swap$(numbers[i], numbers[j])$
    end if
  end for
  swap$(numbers[i+1], numbers[end])$
  return $i + 1$ ▷ posizione del pivot
end function

Complessità e analisi dei tempi

Definiamo l'equazione di complessità dell'algoritmo **quick_select**.
Sia $m \in [1..n-1]$ il numero di numeri su cui svolgo la ricorsione, sia $[i, j]$ la porzione di array in cui mi sto muovendo:

$$T(n) = \begin{cases} \mathcal{O}(1) & \text{se } k \notin [i, j], \\ T(m) + \mathcal{O}(n) & \text{con } m = j - i \text{ e } k \in [i, j] \end{cases}$$

La complessità dipende dalla dimensione di m, che nel caso peggiore corrisponde a $n-1$, cioè quando il pivot cade come primo o ultimo elemento ed effettuo la ricorsione sul resto dell'array.
Se ad ogni chiamata ricorsiva ci ritroviamo nel caso descritto precedentemente, al massimo verranno effettuate n chiamate su array di dimensione $n-1$.
Quindi l'algoritmo nel caso peggiore risulta avere complessità pari a:

$$\begin{aligned} T(n) &= T(n-1) + \mathcal{O}(n) \\ &= \sum i = 1^n \mathcal{O}(i) \\ &= \mathcal{O}(n^2) \end{aligned} \tag{1}$$

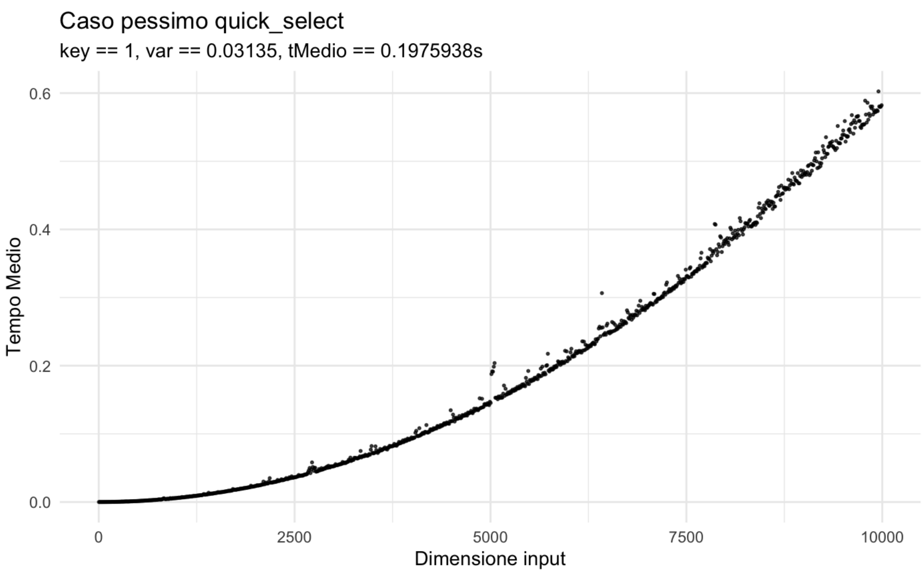
Nel caso medio, il pivot ha valore tale da dividere quasi equamente l'array in parti uguali. L'equazione di complessità diventa quindi:

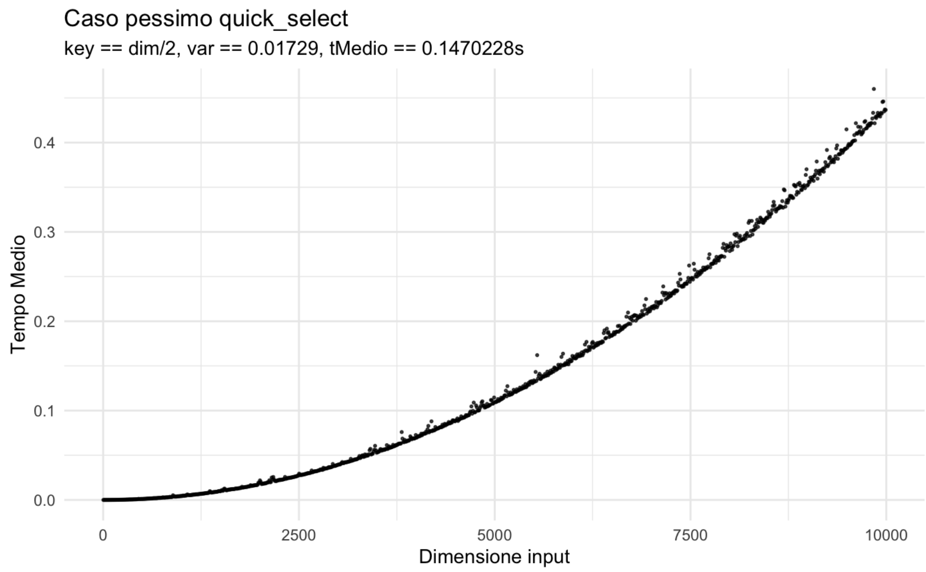
$$
\begin{aligned}
T(n) &= T(\frac{n}{2}) + \theta(n) \qquad\qquad \text{sia } n = 2^k \\
T(2^k) &= T(2^(k-1)) + \theta(2^k) \\
G(k) &= G(k-1) + \theta(2^k) \\
\sum_{i=0}^{\log(n)} \mathcal{O}(2^i) &= \mathcal{O}(\sum_{i=0}^{\log(n)} 2^i) = \theta(2^i)
\end{aligned} \tag{2}
$$

$$
T(2^k) = \theta(2^i) \iff T(n) = \theta(n) \tag{3}
$$

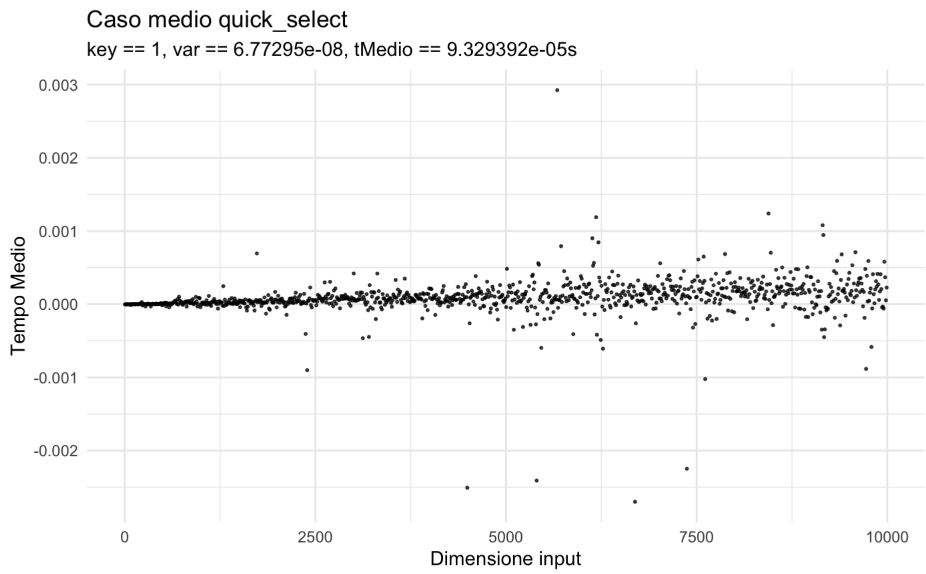
Grafici

Si mostra l'andamento dei tempi al variare dell'input, su dimensione crescente, e al variare di k.
Per il caso pessimo si è scelto di generare delle stringhe di input in cui tutti i valori sono uguali, in modo da partizionare l'array sempre in maniera sbilanciata ed effetturare la ricorsione su $n-1$ elementi.
Si danno due esempi per k diverso.

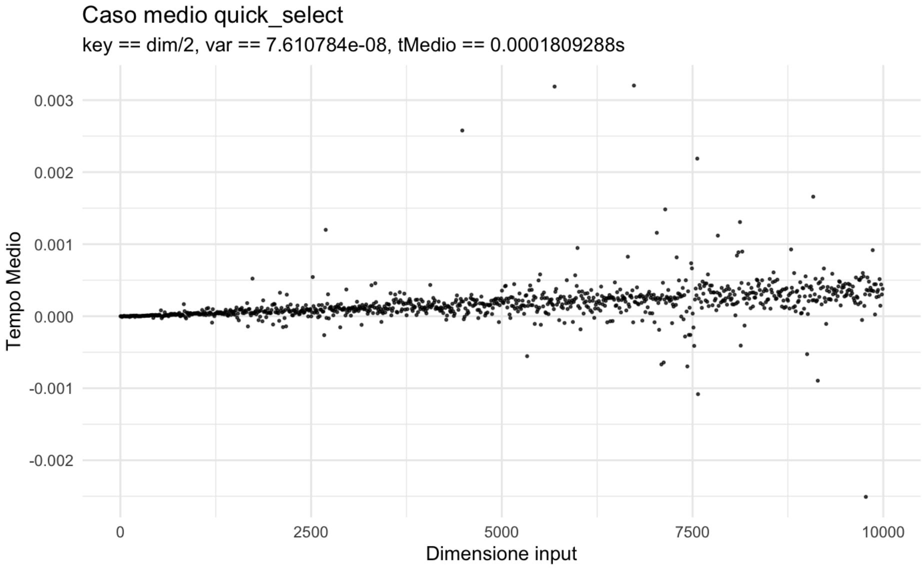


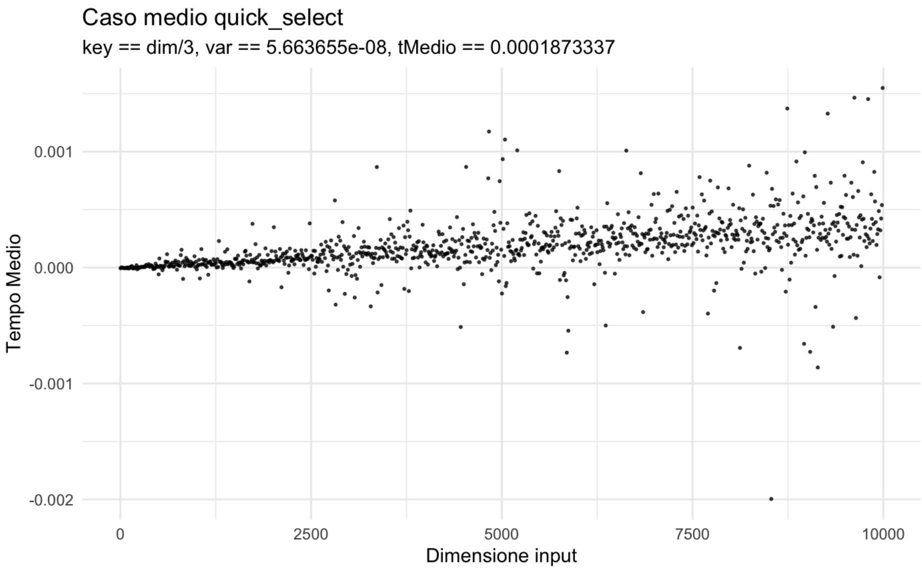


Come da ipotesi il tempo medio ha crescita quadratica ripestto alla dimensione dell'input.
Di seguito si mostrano dei casi medi, in cui i valori dell'input sono scelti randomicamente. Nel misurare i tempi si riscontrano dei valori negativi, si ipotizza a causa della scorporazione dal tempo di esecuzione della preparazione dell'input, che potrebbe impiegare un tempo maggiore rispetto al tempo totale nel caso in cui il valore da trovare viene trovato subito.



Già da questo grafico, oltre a vedere che il tempo medio ha crescita più o meno lineare rispetto la dimensione dell'input, si può notare come i casi non siano uniformi, questo a causa della scelta randomica del pivot per la partiozione e della distribuzione dei valori nel vettore. Si danno altri esempi al variare di k.


Caso medio quick_select
key == dim/2, var == 7.610784e-08, tMedio == 0.0001809288s
Tempo Medio
0.003
0.002
0.001
0.000
-0.001
-0.002
0
2500
5000
7500
10000
Dimensione input


Caso medio quick_select
key == dim/3, var == 5.663655e-08, tMedio == 0.0001873337
Tempo Medio
0.001
0.000
-0.001
-0.002
0
2500
5000
7500
10000
Dimensione input

3 Heap Select

Algoritmo

Questo algoritmo di selezione risolve il problema del *k-esimo* elemento più piccolo di una lista non ordinata di numeri utilizzando la struttura dati Heap. Se il valore di k è minore della metà, scegliamo di utilizzare la min-heap, viceversa, se è più grande della metà, possiamo sfruttare la max-heap per riportarci al problema duale, cioè cercando il $(dim - k + 1)$-esimo elemento più grande.

La prima heap H1 é costruita a partire dal vettore fornito in input in tempo lineare e non viene modificata. La seconda heap H2 contiene solamente indici di valori corrispondenti alla prima heap. Inizialmente H2 contiene un solo nodo, corrispondente alla radice di H1. All'i-esima iterazione, per i che va da 1 a k-1 (con k oppportuno se scelgo di utilizzare la max-heap) l'algoritmo estrae la radice di H2, corrispondente a un nodo x_i in H1 e reinserisce in H2 gli indici del figlio destro e sinistro di x_i (se esistono) nella heap H1. Dopo k-1 iterazioni, la radice di H2 corrisponderà all'indice del *k-esimo* elemento più piccolo del vettore fornito in input.

Pseudocodice

Algorithm 3 Heap Select - second solution

function HEAP_SELECT($numbers, start, end, k$)
  if $k \geq \frac{dim}{2}$ **then**
    return $max\_heap\_select(numbers, start, end, k)$
  else
    return $min\_heap\_select(numbers, start, end, k)$
  end if
  return 1
end function

Di seguito lo pseudocodice della procedura **max_heap_select**, analoga sarà la procedura **min_heap_select** con valori di iterazione $[1..k-1]$ fatta con il k preso in input.

Per implementare la precedente procedura si è scelto di utilizzare una struttura per la **minHeap** e una per la **maxHeap**, inoltre vengono implementate due strutture di heap ausiliare (max e min) in cui i nodi hanno come valore gli indici che corrispondono ai valori dei nodi nella heap iniziale.
Il metodo *successoriDi(x)* inserisce nella seconda heap gli indici del figlio sinistro e destro del nodo avente valore x.

Algorithm 4 Heap Select - second solution

```
function MAX_HEAP_SELECT(numbers, dim, k)
    MaxHeap_1 ← numbers
    MaxHeap_2 ← MaxHeap1.getMax()
    i ← 0
    new_k = HeapSize(MaxHeap1) − k + 1
    while i ≤ new_k − 1 do
        x ← MaxHeap_2.extractMax()
        MaxHeap_2 ← successoriDi(x)        ▷ Aggiungo a 2 i successori di x
        i + +
    end while
     return MaxHeap_1[MaxHeap_2.getMax()]
end function
```

Complessità e analisi dei tempi

Definiamo l'equazione di complessità dell'algoritmo **heap_select_min**, analogamente si può procedere per la soluzione con le *maxHeap* .
Sappiamo che la complessità di *Build_Heap* è pari a $\mathcal{O}(n)$. Il ciclo while viene eseguito k volte e il corpo ha complessità pari alla complessità di *extractMin* sulla heap ausiliaria. Quest'ultima conterrà al massimo k elementi essendo che ad ogni iterazione viene rimosso un elemento e successivamente ne vengono inseriti due.
Facciamo un esempio per $k = 4$, con $k - 2$ iterazioni del corpo del while:

- $i = 0$ $\rightarrow$ un elemento inziale, due finali.
- $i = 1$ $\rightarrow$ due elemento inziale, tre finali.
- $i = 2$ $\rightarrow$ tre elemento inziale, quattro finali.

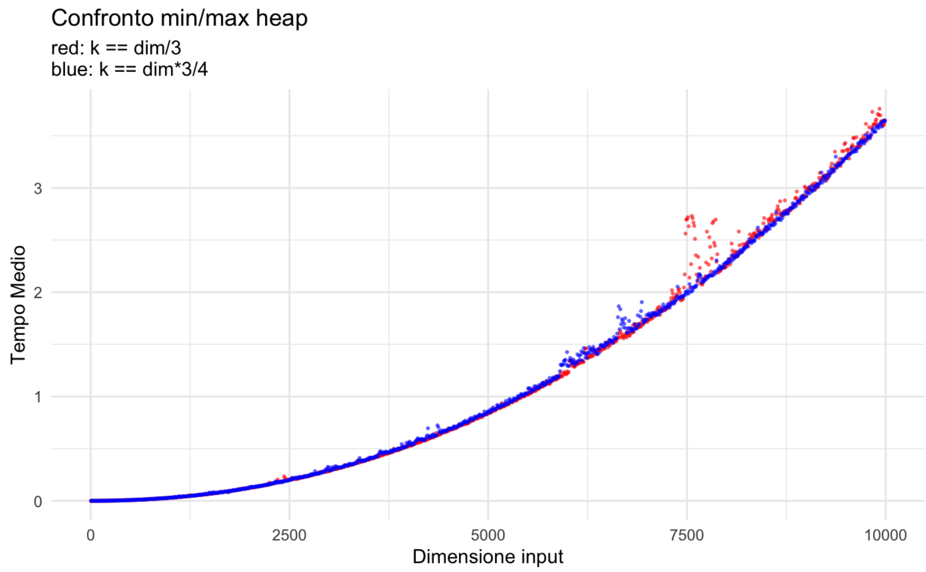
L'equazione di complessità risulta quindi:

$$T(n) = \mathcal{O}(n) + O(klog(k))$$

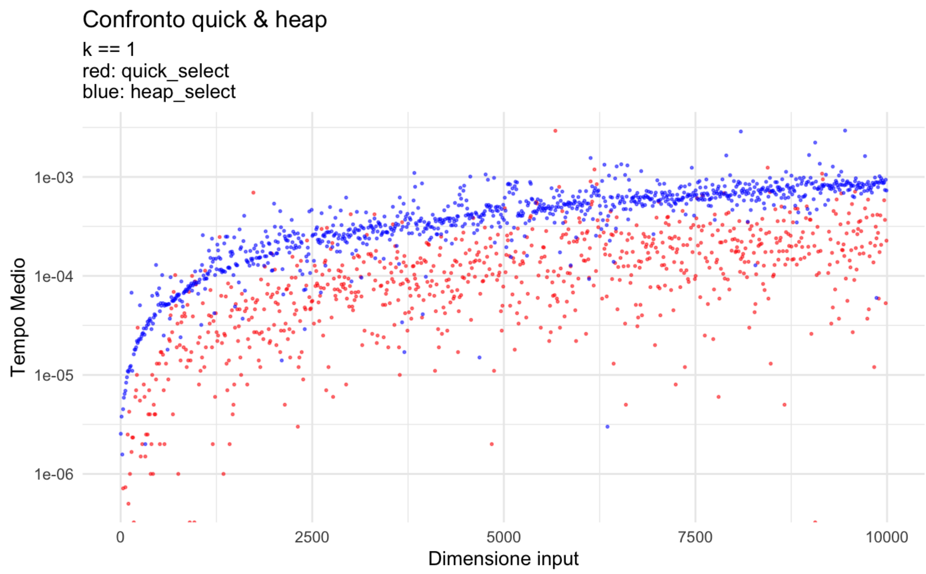
= O(n + k log(k))

Grafici

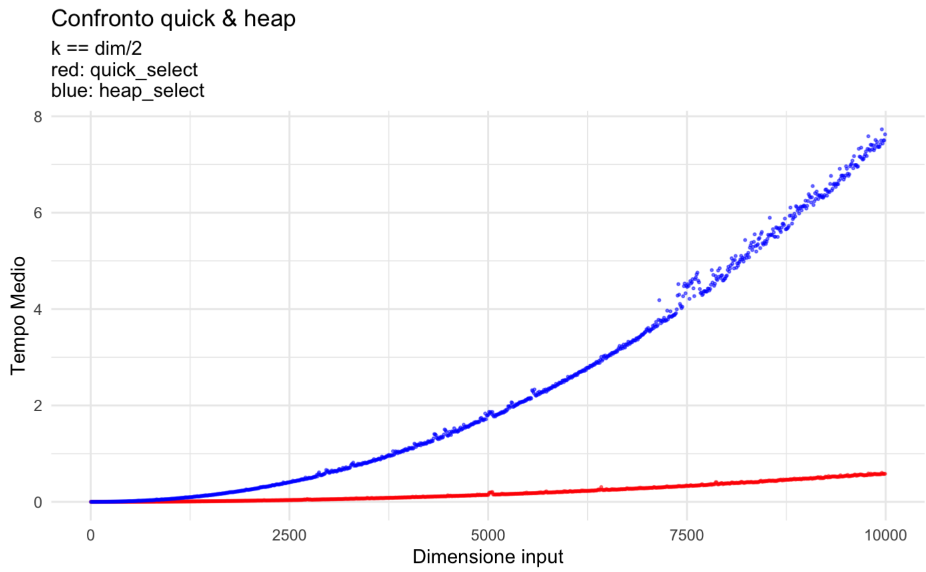
Di seguito viene proposto un grafico comparativo per k rispettivamente nella prima e seconda metà del vettore. Si mostra che gli andamenti dei tempi di esecuzione corrispondono, avendo utilizzato nel primo caso una *min-heap* e nel secondo una *max-heap*.



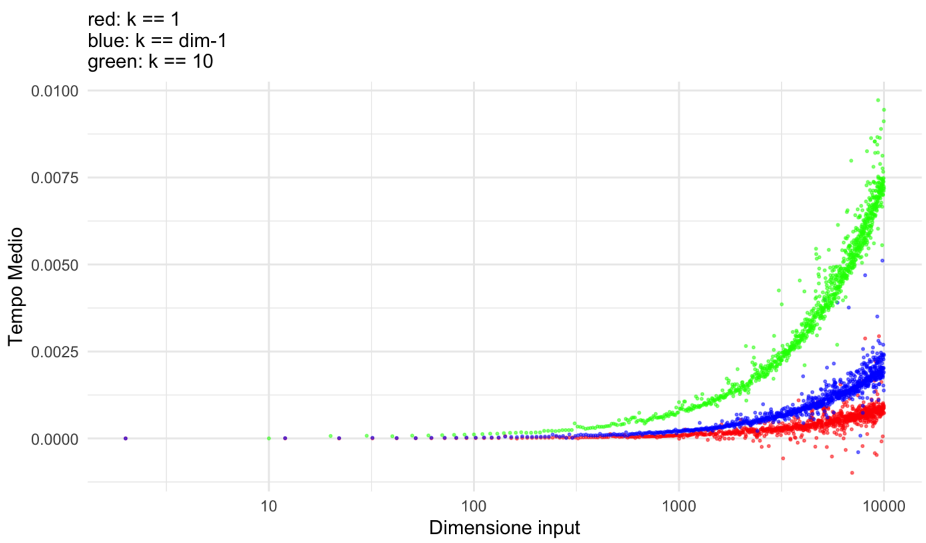
Il seguente grafico mostra come in caso di k molto piccolo (o rispettivamente molto grande) è preferibile l'algoritmo *heap_select* rispetto a *quick_select* avendo tempo medio più uniforme, non dipendendo dalla disposizione degli elementi nel vettore. L'asse y è in scala logaritmica per visualizzare meglio la dispersione dei dati rispetto il tempo.

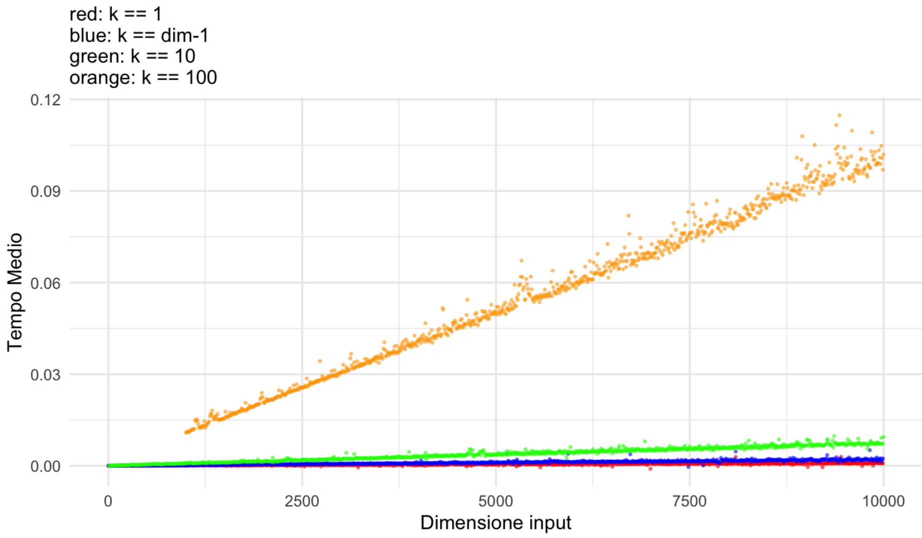


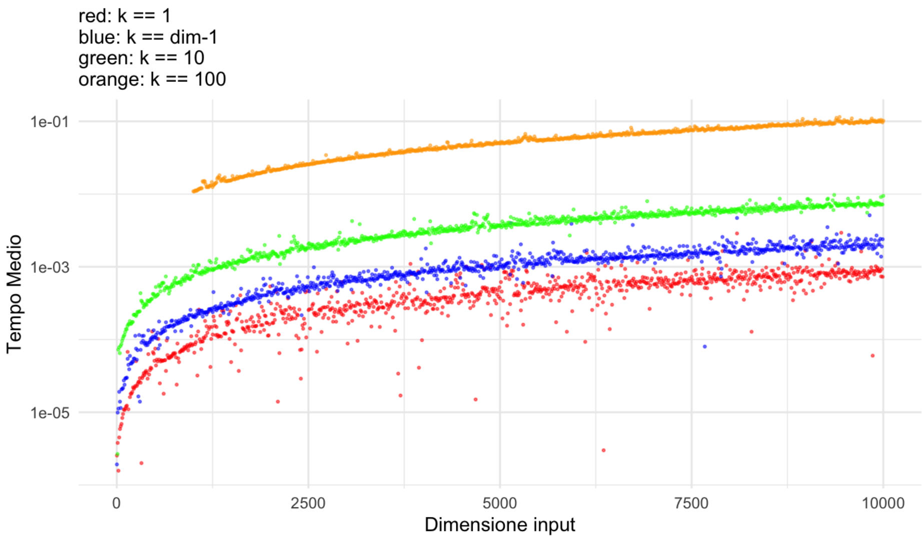
Considerando invece il caso di un k centrale, l'algoritmo *heap_select* risulta pessimo rispetto il worst case di *quick_select*.



Il seguente grafico, con asse delle x in scala logaritmica, mostra come per vettori di dimensione relativamente ridotta non si ha grande differenza di tempo rispetto alla scelta di k. Per vettori di dimensione abbastanza grande la differenza tra $k = 10$ e $k = 1$ risulta importante per il tempo di esecuzione.




red: k == 1
blue: k == dim-1
green: k == 10
orange: k == 100
0.12
0.09
0.06
0.03
0.00
Tempo Medio
0
2500
5000
7500
10000
Dimensione input


red: k == 1
blue: k == dim-1
green: k == 10
orange: k == 100
1e-01
1e-03
1e-05
Tempo Medio
0
2500
5000
7500
10000
Dimensione input

4 Median of Median Select

Algoritmo

Questo algoritmo ci permette di trovare la mediana dei mediani ed utilizzarla come pivot nella partizione degli elementi, in modo da avere il pivot compreso tra $\frac{1}{4}$ e $\frac{3}{4}$ dell'array. Più precisamente, l'algoritmo esegue le seguenti operazioni:

- divisione dell'array in blocchi di 5 elementi, escluso eventualmente l'ultimo blocco che potrà contenere meno di 5 elementi;
- ordinamento e calcolo della mediana di ciascun blocco;
- calcolo della mediana M delle mediane dei blocchi, attraverso chiamata ricorsiva allo stesso algoritmo;
- partizionamento dell'intero array attorno alla mediana M, attraverso una variante della procedura *Partition* dell'algoritmo *quick select*;
- chiamata ricorsiva nella parte di array che sta a sinistra o a destra della mediana M, in funzione del valore k fornito in input.

L'algoritmo è in place muovendosi nell'array attraverso gli indici, *InsertionSort* copia la porzione da ordinare per non modificare l'array su cui successivamente fare partition. Supponendo n numero di elementi nel vettore, il numero di gruppetti che si verranno a formare è uguale a $\frac{n+4}{5}$, aggiungiamo 4 per far si che il numero minimo di gruppeti sia 1 e non 0.

Pseudocodice

La funzione ha come argomenti l'array di numeri interi, il valore k, *start* e *end* che sono gli indici di inizio e fine della porzione di array su cui si esegue la chiamata ricorsiva.

La funzione *find_median* copia la porzione di array su cui si esegue la chiamata di funzione (per non modificare l'array iniziale) e la ordina per trovare l'elemento mediano.
In corrispondenza della linea 11 dell'algoritmo 5, se la dimensione dell'array dei mediani è minore o uguale a 5, si proceda direttamente a trovare l'elemento mediano, altrimenti si faccia una chiamata ricorsiva alla procedura *Median of Median Select* con argomenti l'array dei mediani e k con $dim/2$.

Algorithm 5 Median of Median Select - third solution

```
function MEDIAN_OF_MEDIAN_SELECT(numbers, start, end, k)
    dim = end − start + 1
    if  k > 0 and k <= dim then
        mediani[(dim+4)/5]
        for i = 0 to dim/5 do
            mediani[i] = findMedian(numbers, start + i ∗ 5, left + i ∗ 5 + 4)
        end for
        if  (i ∗ 5 < dim)  then            ▷ Per gli elementi fuori dai gruppetti
            mediani[i] = findMedian(numbers, start + i ∗ 5, dim%5)
        end if
        if (i ≤ 5)  then
            mediano = findMedian(numbers, 0, i − 1)
        else
            mediano = median_of_median_select(median, 0, i − 1, i/2)
        end if
        pos = partition(numbers, start, end, medianOfmedian)
        if  (pos − left == k − 1)  then
            return numbers[pos]
        else if  (pos − left > k − 1)  then
            return median_of_median_select(numbers, start, pos − 1, k)
        else
            new_k ← k − pos + left − 1
            return median_of_median_select(numbers, pos + 1, end, new_k)
        end if
    end if
end function
```

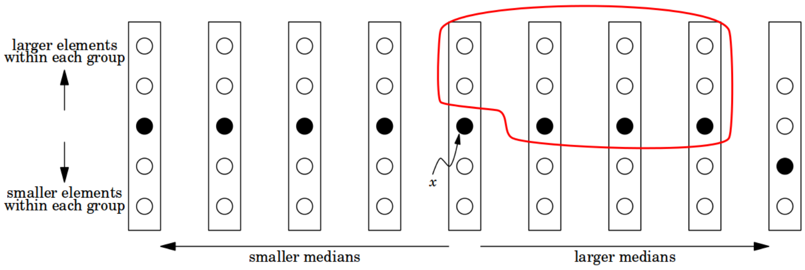
Algorithm 6 Partition

```
function PARTITION(numbers, start, end, medianOfmedian)
    i ← start
    while numbers[i] != medianOfmedian  do         ▷ cerco indice del pivot
        i + +
    end while
    swap(numbers[i], numbers[end])                 ▷ sposto il pivot alla fine
    for  j ← start to end do
        if numbers[j] ≤ pivot then
            i ← i + 1
            swap(numbers[i], numbers[j])
        end if
    end for
    swap(numbers[i + 1], numbers[end])
    return i + 1                                   ▷ posizione del pivot
end function
```

Complessità e analisi dei tempi

Definiamo l'equazione di complessità dell'algoritmo **Median of Median Select**. Di seguito si elecano le complessità dei passi svolti durante l'esecuzione:

- $\mathcal{O}(n)$ divisione degli elementi in gruppi di 5.
- $\mathcal{O}(n)$ ordinamento e selzione dell'elemento mediano di ogni gruppo.
- $T(\frac{n}{5})$ esecuzione della chiamata ricorsiva sulla lista degli elementi medi ottenuti al passo precedente.
- $\mathcal{O}(n)$ costo di partition su tutto l'array in input utilizzando come pivot l'elemento ottenuto nel passo precedente.
- $T(\frac{3}{4}n)$ ricorsione su uno dei sottoarray in base al valore di k.



suddivisione in gruppi per i mediani

Per analizzare il tempo di esecuzione dell'algoritmo dobbiamo prima determinare quanti elementi sono più grandi del pivot. Chiamiamo p il pivot trovato. Almeno la metà delle mediane trovate sono maggiori della mediana delle mediane e quindi almeno metà degli $\lfloor \frac{n}{5} \rfloor$ gruppi contribuiscono con 3 elementi che sono più grandi di x, tranne il gruppo di x stesso che ha meno elementi e quindi si hanno almeno:

$$3(\lceil \tfrac{1}{2} \lceil \tfrac{n}{5} \rceil \rceil - 2) \geq \tfrac{3}{10}n - 6$$

elementi più grandi di x.
Un ragionamento speculare può esser fatto per il numero di elementi minori di x che quindi sono $\geq \frac{3}{10}n - 6$.
Nel caso peggiore quindi la chiamata verrà effettuata su di un array di $\frac{7}{10}n + 6$ elementi. L'equazione di complessità risulta quindi:

$$T(n) = \begin{cases} \theta(1) & \text{con } n \leq c, \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7}{10}n + 6) + \theta(n) \end{cases}$$

Dimostriamo che per qualche costante $cT(n) \leq cn$

$$
\begin{aligned}
T(n) &\leq c\left\lceil \frac{n}{5} \right\rceil + c\left\lceil \frac{7}{10} + 6 \right\rceil + \mathcal{O}(n) \\
&\leq c\frac{n}{5} + c + \frac{7}{10}cn + 6c + \mathcal{O}(n) \\
&\leq \frac{9}{10} + 7c + \mathcal{O}(n) \\
&\leq cn
\end{aligned}
$$

Quindi abbiamo:

$$c\frac{9}{10}n + 7c + kn \leq cn \iff kn \leq c(\frac{1}{10}n - 7) \iff k \leq c(\frac{1}{10} - \frac{7}{n}) \iff c \geq \frac{k}{(\frac{1}{10} - \frac{7}{n})}$$

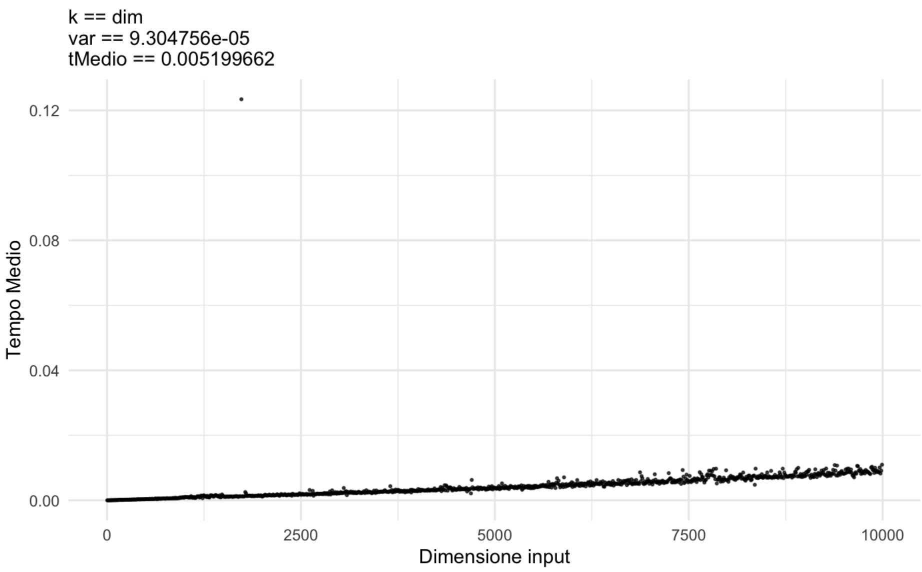
Se prendiamo $n \geq 80$ $c = \frac{k}{\frac{1}{10} - \frac{7}{80}}$ allora c è sicuramente più grande.

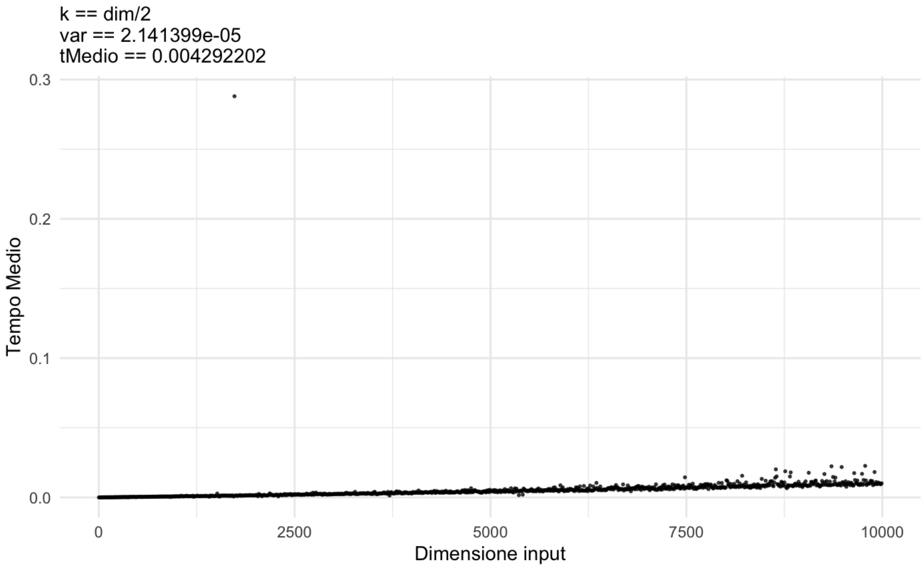
Concludiamo che la procedura **Median of median Select** sia nel caso pessimo che in quello medio ha complessità $\mathcal{O}(n)$.

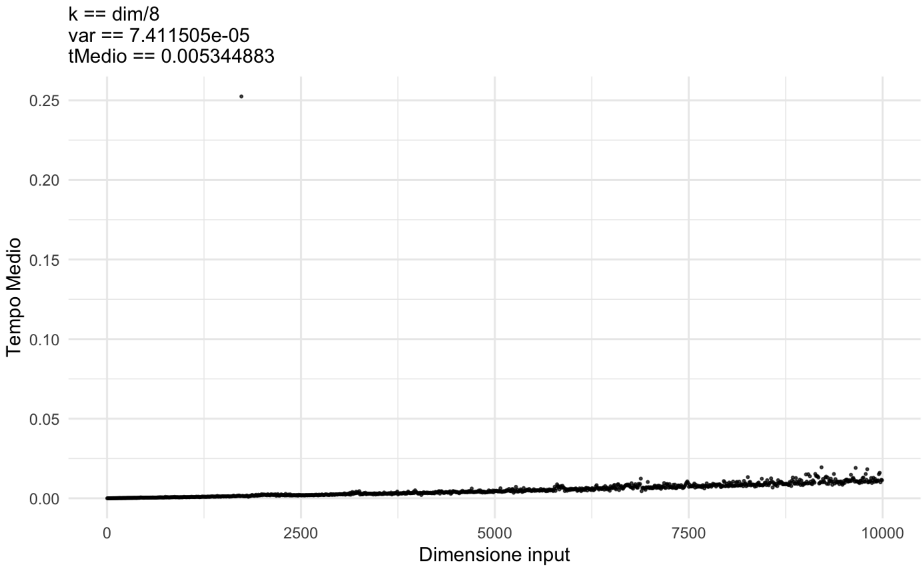
Il grafico dei tempi risultante conferma l'analisi teorica fatta, risultando la soluzione più efficiente per il problema proposto.

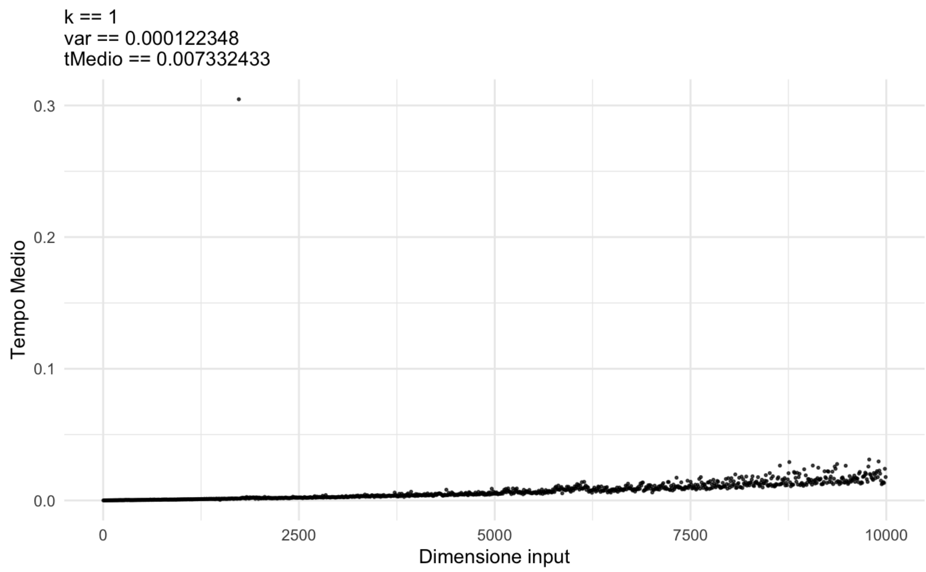
Grafici

Si mostra al variare di k i grafici ottenuti dall'analisi del tempo di esecuzione. Si può notare come per qualsiasi k scelto, i tempi al variare della dimensione crescono in maniera lineare e uniforme rispetto a input con valori diversi, rispecchiando l'analisi teorica con di complessità $\mathcal{O}(n)$ in tutti i casi.

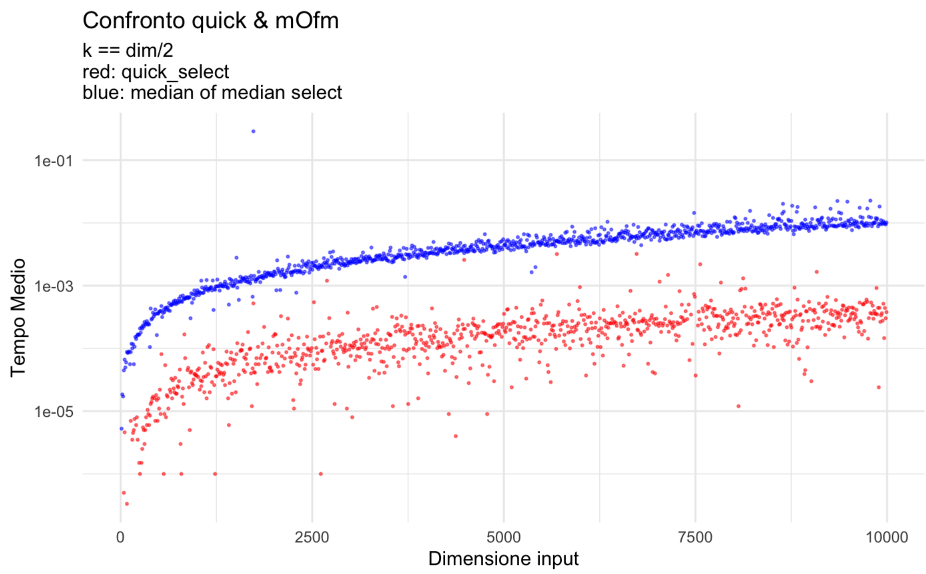



k == dim/2
var == 2.141399e-05
tMedio == 0.004292202
0.3
0.2
0.1
0.0
Tempo Medio
0
2500
5000
7500
10000
Dimensione input

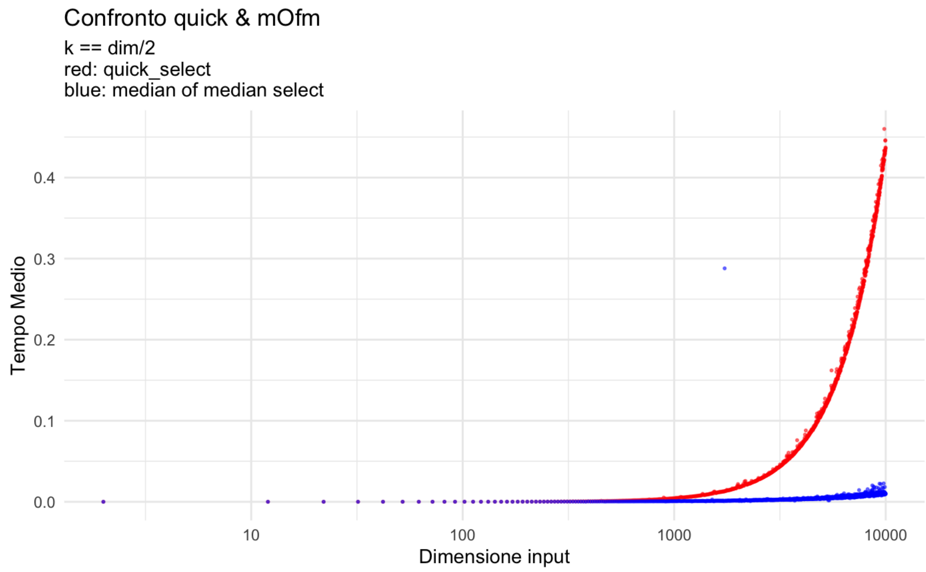

k == dim/8
var == 7.411505e-05
tMedio == 0.005344883
0.25
0.20
0.15
0.10
0.05
0.00
Tempo Medio
0
2500
5000
7500
10000
Dimensione input



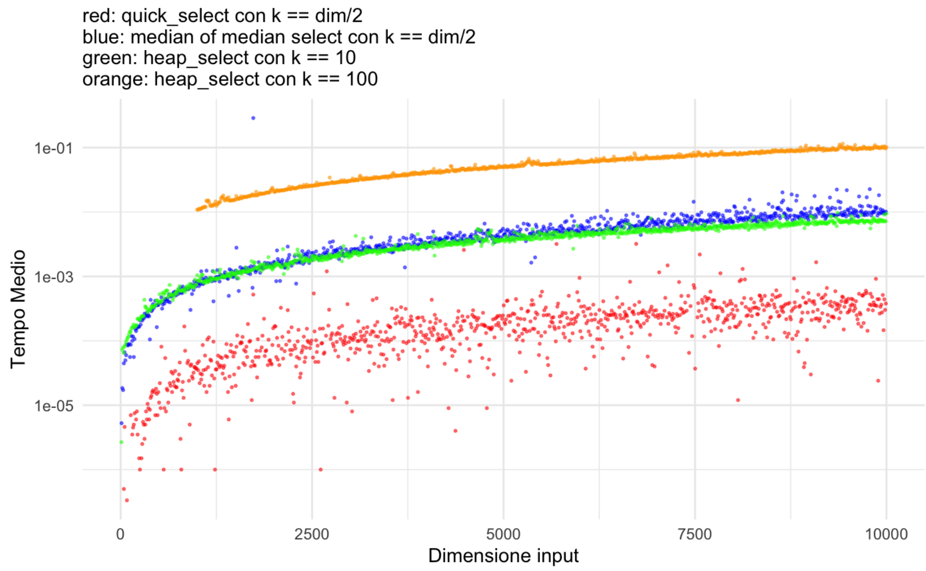
Solamente guardano il valore del tempo medio totale, ma a confermare l'ipotesi anche il grafico sottostante, sembrerebbe che l'algoritmo *quick_select* sia migliore. Ciò non è del tutto vero, essendo che è molto più facile ricadere nel caso pessimo rispetto a quello medio scegliendo un pivot casuale, a differenza di *mOfm_select* che mantiene il tempo medio più o meno costante qualsiasi sia la distribuzione dell'input e la k scelta.



Conviene comunque l'utilizzo di *mOfm_select* essendo che, nel caso in cui si ricada nel caso pessimo di *quick_select*, su input di dimensioni rilevati, viene risparmiato un tempo di esecuzione importante.



Ugualmente in confronto a *heap_select*, avendo quest'ultimo relativamente pochi best case su input di dimensioni rilevanti, rispetto alla costanza di *mOfm_select*. Di seguito un grafico comparativo fra tutti gli algoritmi presentati.



Si può notare come l'uso di *heap_select* dipenda molto dalla k scelta, l'uso di *quick_select* dalla distribuzione dell'input. In casi con k che varia molto, o senza conoscere la distribuzione dell'input, è preferibile l'utilizzo dell'algoritmo *median of median select*.