Università degli studi di Udine
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
a.a. 2023/2024

IMPLEMENTAZIONE ED ANALISI DEI TEMPI DI ESECUZIONE E COMPLESSITÀ DEGLI ALGORITMI DI SELEZIONE

Relazione di laboratorio
Algoritmi Strutture Dati e Laboratorio
[MA0006]

Studente	**Email**	**Matricola**
Chelli Samir	159255@spes.uniud.it	159255
Crevatin Marco	152049@spes.uniud.it	152049
Antic Aleksandar	152107@spes.uniud.it	152107
Picano Alessandro	151982@spes.uniud.it	151982

Contents

1 Introduzione

1.1 Obiettivo

L'obiettivo di questo progetto è implementare e analizzare tre diversi algoritmi per risolvere lo stesso problema, valutando come la differente natura delle soluzioni influisca sui risultati in termini di tempi di esecuzione e complessità degli algoritmi stessi. Il problema affrontato consiste nel trovare il k-esimo elemento più piccolo all'interno di un vettore di lunghezza n. Gli algoritmi implementati sono: Quickselect, Heapselect e Median-of-Medians select. Successivamente, è stato condotto uno studio dettagliato dei dati, analizzando i tempi di esecuzione medi dei tre algoritmi. Infine, sono stati costruiti grafici illustrativi che mostrano la variazione del tempo di esecuzione in funzione della lunghezza del vettore n e del valore k fornito in input.

1.2 Accortezze

All'interno del progetto sono stati sfruttati porzioni di codice forniteci dal docente a cui abbiamo apposto alcune modifiche. Non lavorando all'interno di un sistema Unix-like (macOS, Linux) la funzione 'monotonic()'[1] è risultata problematica a causa di interferenze con il clock pertanto è stata sostituita con 'perf_counter()'[2]

2 Materiali e Metodi

2.1 Hardware

La Workstation adoperata per effettuare le misurazioni è un computer desktop composto dalle seguenti componenti:

- **Processore:** Intel(R) Core(TM) i5-9600K CPU @ 3.7GHz
- **RAM:** corsair vengeance DDR4, 16GB, 3200MHz
- **Scheda madre:** ASUSTeK COMPUTER INC. PRIME Z390-P

2.2 Software

I test sono stati effettuati all'interno del sistema operativo Windows 10 Pro (versione 22H2), il codice è stato realizzato ed eseguito all'interno dell'ambiente di sviluppo Visual Studio Code e le rappresentazioni grafiche sono state realizzate attraverso la piattaforma Draw.io

[1] Funzione presente all'interno del modulo 'Time' di Python 3.3 utilizzata per ottenere il valore di un orologio monotonico (che non può tornare indietro).

[2] Funzione presente all'interno del modulo 'Time' di Python 3.3 che fornisce un timer di alta precisione che fornisce un conteggio di performance del sistema con risoluzione molto alta.

2.3 Metodo di analisi

Per la misurazione dei tempi, sono stati generati un insieme di 100 tipi di array differenti con una dimensione compresa tra 100 e 100.000, sempre crescente mediante la seguente serie geometrica:

$$A = 100$$
$$B = 10^{3/99}$$
$$[\mathbf{math.floor}(A \cdot \mathbf{pow}(B, i)) \text{ for } i \text{ in } \mathbf{range}(100)]$$

In seguito, per ogni dimensione n dell'array, sono stati generati 100 array differenti, per un totale di 10.000 array. Infine, per ognuno di questi 10.000 array, sono state effettuate 50 ricerche adottando un k casuale presente nell'array, per un totale di $100 \cdot 100 \cdot 50 \cdot 3 = 1.500.000$ test. Dopo di che, è stata ricavata la media per ogni ricerca in ogni array di lunghezza n ed è stata riportata all'interno del file CSV Dati.py presente nella documentazione. Per ricavare i dati è stato eseguito il programma "misurazioneTempiEsec.py" per la diagnostica per un totale di 17180,528182 ovvero all'incirca 4 ore e tre quarti. La foto che segue rappresenta il formato con cui sono stati immagazzinati i dati.

```
1   lunghezza n,quickSelect,heapSelect,medianOfMedians
54  3764,0.0031244578003825152,0.0062669708066096065,0.010201459599891678
55  4037,0.0031536238002590833,0.007167978799931006,0.011170584200037411
56  4328,0.003432235799933551,0.007765413400309626,0.01167493639976601
57  4641,0.0037562948000559116,0.008163140006159666,0.012618040200235554
58  4977,0.0039853633998136505,0.009614007399912226,0.014149740199907683
59  5336,0.004563354999758303,0.009492732000304385,0.015215712000033818
60  5722,0.005197109400149202,0.01152624380015186,0.017355634599749464
61  6135,0.005579943400254706,0.01274115880066529,0.01759695340009057
62  6579,0.005904177600197727,0.013156057999964105,0.01819489479949698
63  7054,0.005783632799837506,0.01445530499983579,0.02047917280008551
64  7564,0.0062719065994606355,0.01357325280024088,0.022116584800096463
65  8111,0.006926009799848544,0.016429274399823043,0.023600845999375453
66  8697,0.008078459400130668,0.018177820799464825,0.024360140000208048
```

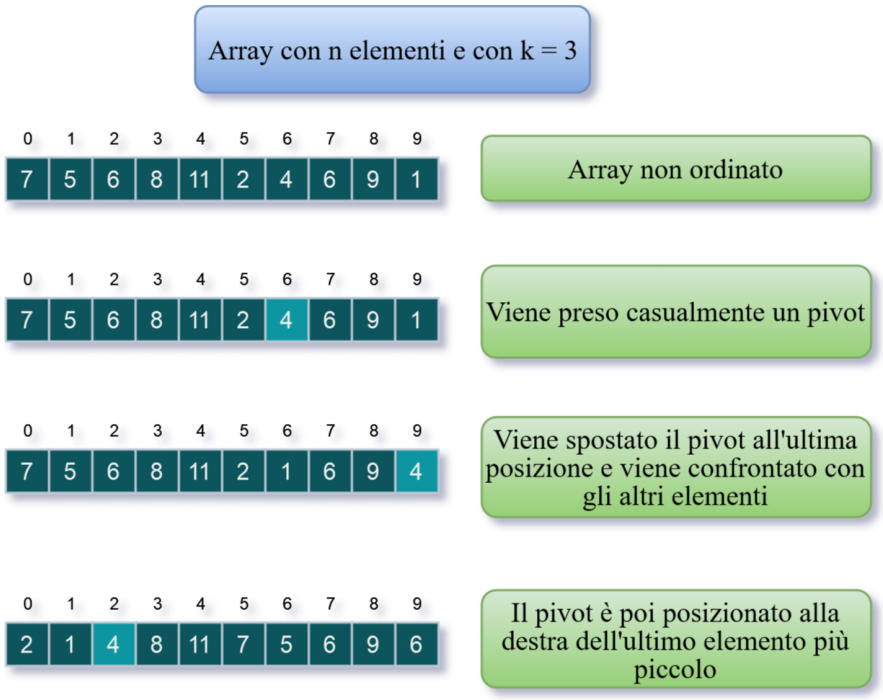
Figure 1: Esempio formattazione dati all'interno del file CSV

3 Quick Select

3.1 Teoria

Il QuickSelect è un algoritmo ricorsivo randomizzato progettato per individuare il k-esimo elemento in un array non ordinato di dimensione n. Derivato dall'algoritmo QuickSort, utilizza l'approccio prune and search [3] per selezionare un pivot in maniera randomica e partizionare l'array in base all'indice scelto, consentendo l'individuazione del k-esimo elemento richiesto. In situazioni in cui k supera la dimensione dell'array, l'algoritmo può terminare in tempo costante senza eseguire la procedura di partizionamento, ottimizzando così l'efficienza computazionale.

3.2 Rappresentazione grafica dell'esecuzione



[3] Metodo per risolvere problemi di ottimizzazione a procedura ricorsiva, ad ogni passo la taglia dell'input è ridotta di un fattore costante $0 < p < 1$.

3.3 Pseudocodice

```
Function Partition(arr, low, high):
    pivot ← arr[high];
    i ← low - 1;
    for j ← low to high do
        if arr[j] ≤ pivot then
            i ← i + 1;
            scambia arr[i] e arr[j] con arr[j] e arr[i];
        end
        scambia arr[i +1 ] e arr[high] con arr[high] con arr[i + 1];
    end
    return i + 1;

Function RandomizedPartition(arr, low, high):
    randIndex ← random.randint(low, high);
    scambia arr[high] e arr[randIndex] con arr[randIndex] e arr[high];
    return Partition(arr, low, high);

Function Quick(arr, low, high, k):
    if low ≤ high then
        pivotIndex ← RandomizedPartition(arr, low, high);
        if pivotIndex = k then
            return arr[pivotIndex];
        else if pivotIndex < k then
            return Quick(arr, pivotIndex + 1, high, k);
        else
            return Quick(arr, low, pivotIndex − 1, k);
        end
    return null;

Function QuickSelect(arr, k):
    if k > len(arr) or k ≤ 0 then
        return null;
    end
    return Quick(arr, 0, len(arr) − 1, k − 1);
```

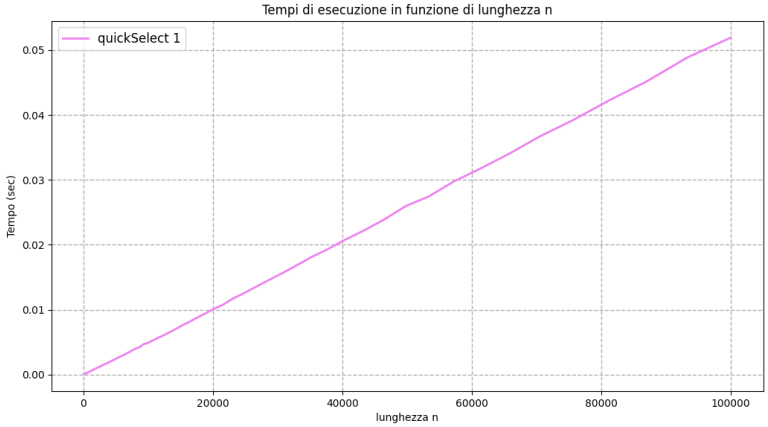
Algorithm 1: QuickSelect

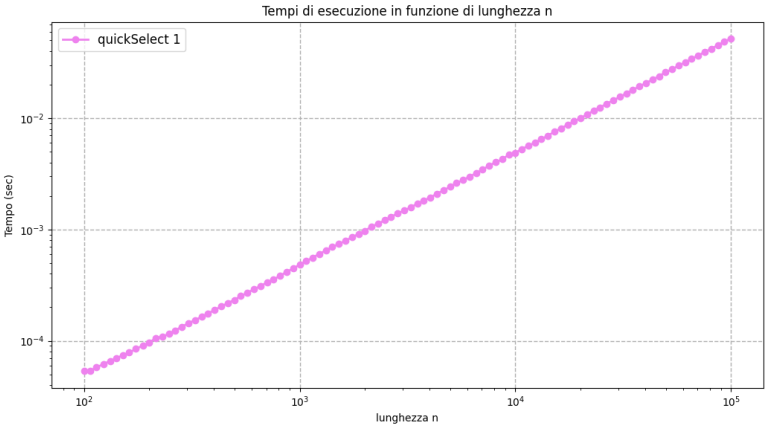
3.4 Complessità

Indicando con n la dimensione dell'input, $T(n)$ rappresenta la complessità temporale complessiva dell'algoritmo e $S(n)$ la complessità temporale dell'operazione di riduzione, allora la relazione di ricorrenza che descrive $T(n)$ sarà la seguente:

$$T(n) = S(n) + T(n(1-p))$$

La complessità dell'algoritmo nel caso medio è uguale a $O(n)$, mentre nel caso peggiore, ovvero quando il pivot scelto è ai margini dell'array, è di $\theta(n^2)$, utilizzando un pivot randomizzato minimizziamo la possibilità di avere il caso peggiore, seppur ancora possibile.



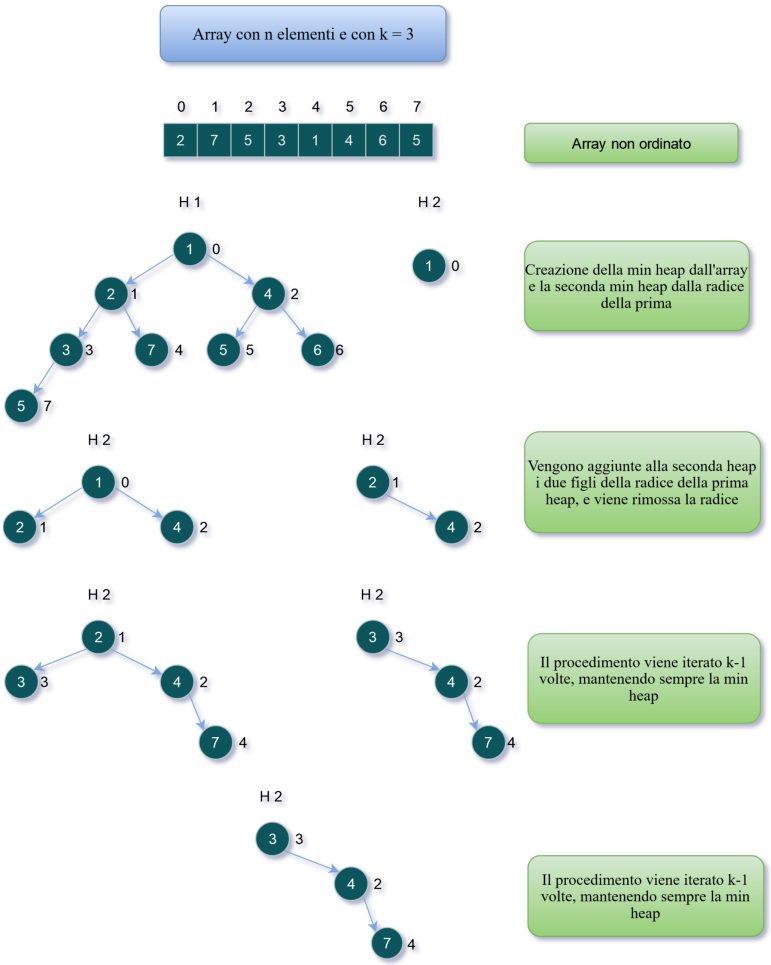


4 Heap Select

4.1 Teoria

L'algoritmo Heapselect è una procedura ricorsiva progettata per individuare il k-esimo elemento più piccolo in un array di dimensione n. Questo metodo impiega due strutture min-heap. La prima min-heap è costruita interamente dal vettore originale, mentre la seconda min-heap viene costruita in modo ricorsivo. Durante ogni ciclo, la radice della seconda heap viene rimossa e sostituita dai figli del nodo che sono presenti nella prima heap. Questo processo garantisce che, al termine del ciclo $(k-1)$ - esimo, la radice della seconda heap contenga il k-esimo valore più piccolo del vettore.

4.2 Rappresentazione grafica dell'esecuzione



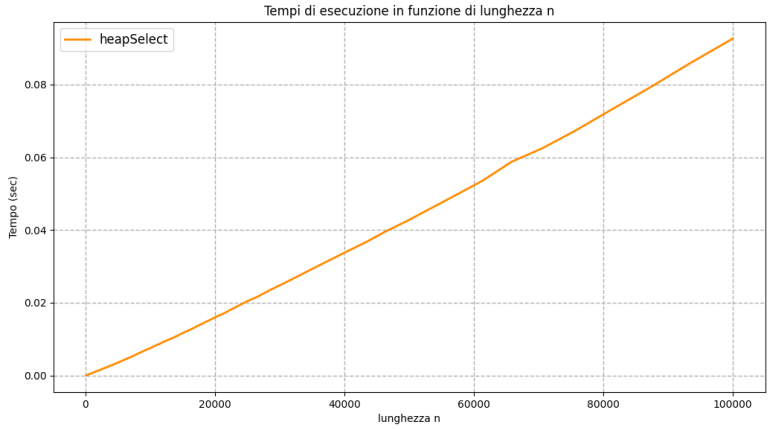
4.3 Pseudocodice

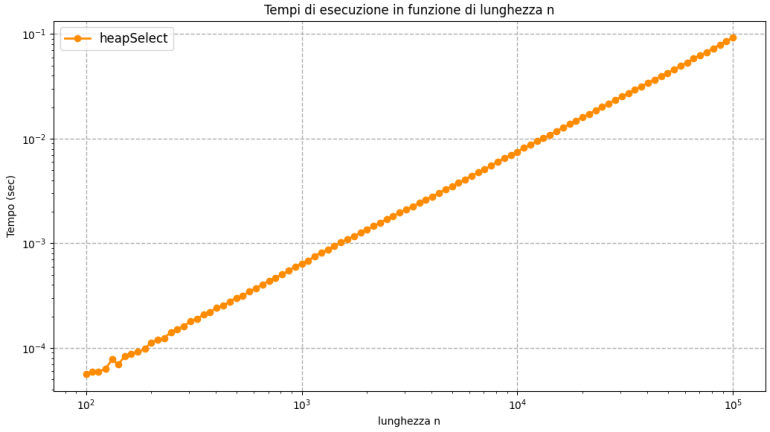
```
Function HeapSelect(arr, k):
    H1 ← arr;
    heapq.heapify(H1);
    H2 ← [(H1[0], 0)];
    for i ← 0 to k − 1 do
        node ← heapq.heappop(H2);
        rootIndex ← node[1];

        leftChildIndex ← 2 · rootIndex + 1;
        rightChildIndex ← 2 · rootIndex + 2;
        if leftChildIndex < len(H1) then
            heapq.heappush(H2, (H1[leftChildIndex], leftChildIndex));
        end
        if rightChildIndex < len(H1) then
            heapq.heappush( H2,
                            ( H1 [rightChildIndex], rightChildIndex ));
        end
    end
    return H2[0][0];
```

Algorithm 2: HeapSelect

4.4 Complessità

Considerando n come il numero di elementi dell'array e k il k-esimo elemento più piccolo da trovare, l'algoritmo HeapSelect ha la complessità temporale lineare uguale a $O(n+klogk)$, e si trova analizzando le fasi dell'algoritmo: viene richiamata la funzione heapify per la costruzione di H1, che ha complessità $O(n)$, la costruzione di H2, che ha complessità $O(klogk)$ e il ritorno del valore, che ha complessità $O(1)$, di conseguanze la complessità totale è $O(n+klogk)$.



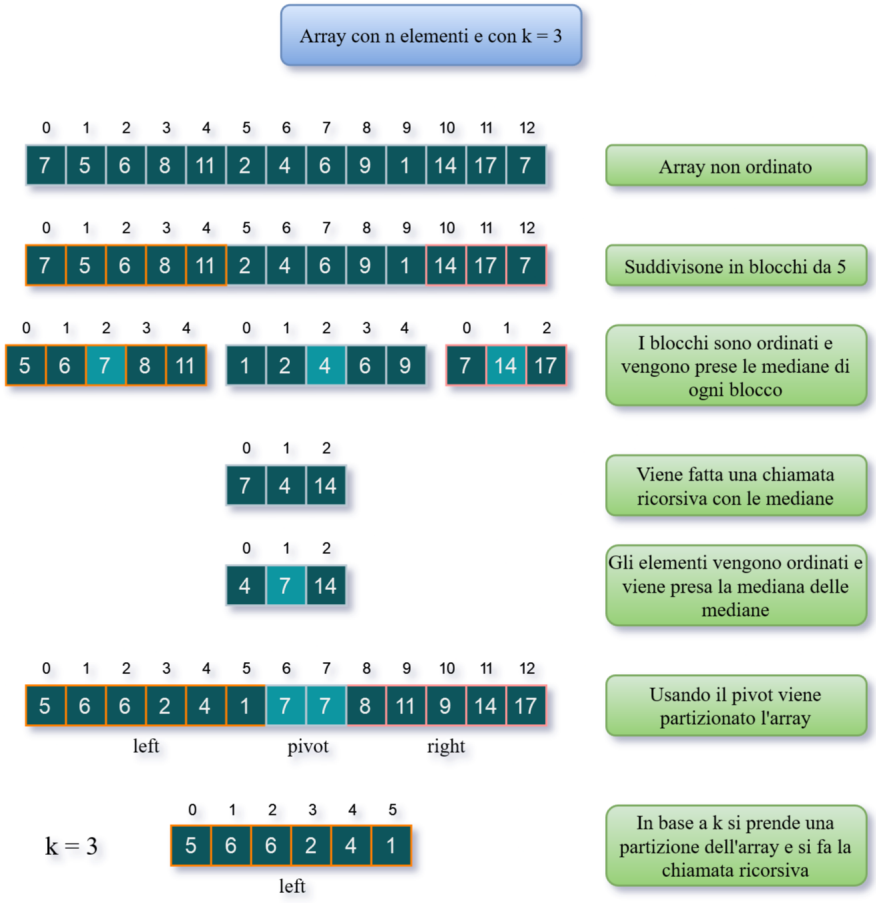


5 Median-of-Medians Select

5.1 Teoria

Median of Medians è un algoritmo ricorsivo concepito per individuare il k-esimo elemento più piccolo in un array di dimensione n. L'array viene suddiviso inizialmente in blocchi di 5 elementi ciascuno, ordinati singolarmente per estrarne il valore mediano. Successivamente, si effettua una chiamata ricorsiva per determinare la mediana delle mediane. Questa mediana viene utilizzata come pivot per partizionare l'array originale in tre segmenti: sinistra, destra e pivot. Infine, l'algoritmo richiama ricorsivamente la sezione appropriata dell'array in base al valore di k.

5.2 Rappresentazione grafica dell'esecuzione



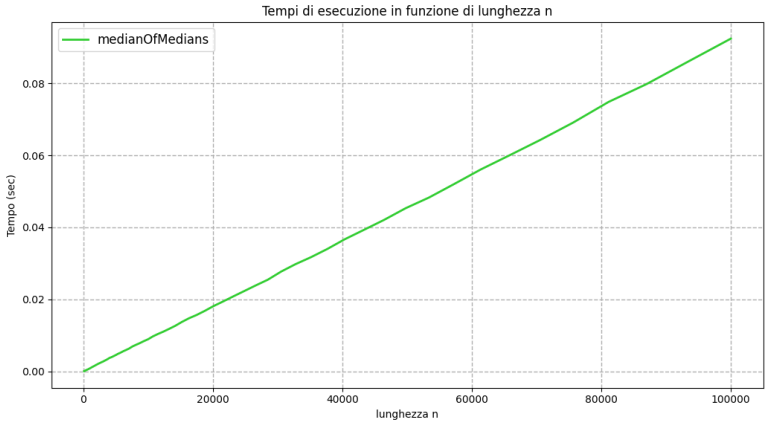
5.3 Pseudocodice

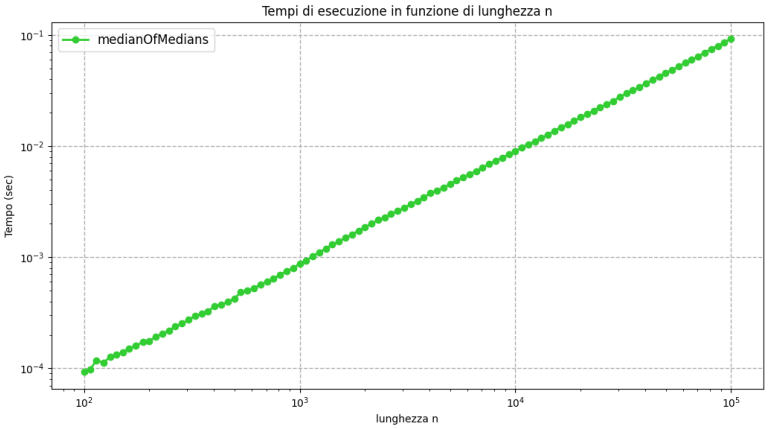
```
Function MedianOfMedians(arr, k):
    MedianOfMediansAux(arr, k − 1);

Function MedianOfMediansAux(arr, k):
    if len(arr) = 1 then
        return arr[0];
    end
    blocks ← [arr[i : i + 5] for i in range(0, len(arr), 5)];

    medians ← [sorted(block) [len(block)//2] for block in blocks ];

     pivot ← sAuxmedians, len(medians)//2;

    left ← [x for x in arr if x < pivot];
    right ← [x for x in arr if x > pivot];
    pivotCount ← arr.count(pivot);

    leftLen ← len(left);
    pivotLen ← len(arr) − leftLen − len(right);

    if k < leftLen then
        return MedianOfMediansAux(left, k);
    end
    else if k < leftLen + pivotLen then
        return pivot;
    end
    else
        return MedianOfMediansAux(right, k − leftLen − pivotCount);
    end
```

Algorithm 3: MedianOfMedians

5.4 Complessità

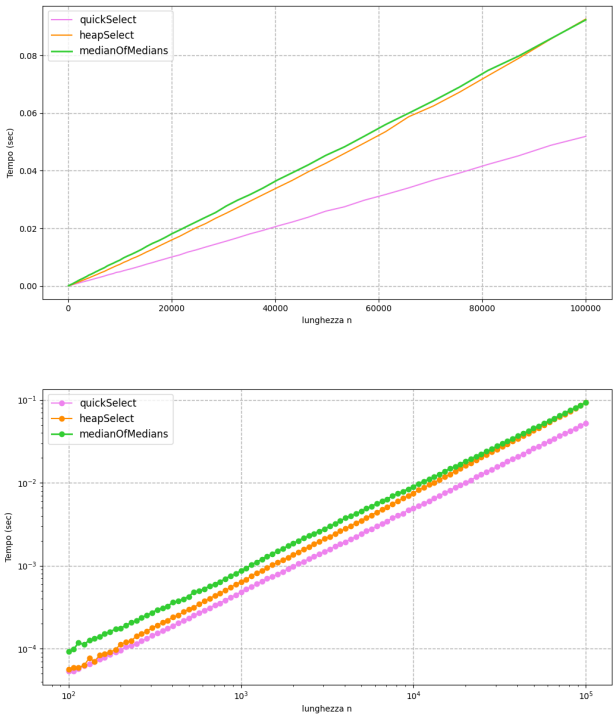
L'algoritmo Median Of Medians ha una complessità temporale lineare uguale a O(n), questo grazie a tre sostaziali passaggi: la prima suddivisione dell'array originale, dove ogni elemento viene analizzato uno alla volta (mantenendo la complessità a O(n)), la ricorsione per determinare la mediana delle mediane, che seppur il numero di livelli di ricorsione si a O(logn), mantiene la complessità totale a O(n), e l'individuazione del pivot e l'ordinamento dell'array, preservando la complessità lineare. Di conseguenza la complessità totale dell'algoritmo è O(n).





6 Conclusioni

Nel corso di questa indagine, abbiamo condotto un'analisi approfondita dei tempi di esecuzione e della complessità computazionale di tre algoritmi distinti, in seguito abbiamo confrontato i grafici generati per identificare l'algoritmo con la maggiore efficienza operativa tra quelli implementati. Se si considera esclusivamente la complessità teorica, il "Median of Medians" dovrebbe apparire come il candidato più promettente, poiché presenta una complessità costante sia nel caso medio che nel caso peggiore. Tuttavia, l'analisi empirica dei tempi di esecuzione rivela che il "Quick Select" prevale sugli altri algoritmi in termini di velocità, emergendo come la scelta più rapida ed efficiente nell'ambito pratico.



Come ultima considerazione, il gruppo di lavoro propone di ottimizzare i tempi di esecuzione dell'algoritmo di analisi adattandolo per il multithreading [4]. È stato osservato che, per impostazione predefinita, il sistema operativo allocava solo un singolo core, corrispondente a circa il 20% della capacità di calcolo totale. Implementando il multithreading, si prevede una riduzione significativa dei tempi di esecuzione.

[4] tecnica di programmazione che consente a un'applicazione di eseguire più sottoprocessi concorrentemente, permettendo un utilizzo più efficiente delle risorse hardware e migliorando le prestazioni