Comparative Analysis of QuickSelect, HeapSelect and MedianOfMediansSelect

A project by:

Name Surname	Student-Id	E-Mail
Francesco Verrengia	157847	157847@spes.uniud.it
Martina Ammirati	161831	161831@spes.uniud.it
Riccardo Gottardi	162077	162077@spes.uniud.it

University of Udine Department of Mathematical, Computer and Physical Sciences.

26 May 2024

Purpose of the project

The project focuses on the implementation of three different algorithms for selecting the $k-th$ smallest element contained in an array, and on the comparative analysis of the behaviour of these algorithms in the average case. It also focuses on the implementation of a $benchmark$ that is able to accurately measure the performance of the three algorithms and to generate a logarithmic and linear scale graph that effectively represents their performance as the size of the arrays used and the choice of parameter k change.

My role in this project (Francesco Verrengia ~ Verryx_02)

I mainly handled the organizational aspects of the work, but also wrote the $benchmark$, implemented the graph drawing function, added comments to each procedure, generated the graphs, and wrote this report. Each phase of the project was carefully reviewed by all members of the group.

Introduction: selection algorithms in general

Selection algorithms, such as $QuickSelect$, $HeapSelect$ and $MedianOfMediansSelect$, are used to find the $k-th$ smallest (or largest) element in a data set without having to sort the entire set. This type of operation is crucial in various areas of computer science and applied mathematics.

In statistics, for example, finding the median or other percentiles of a data set is critical. In fact, the median is often used as a central measure resistant to outliers, while percentiles describe the distribution of the data. Selection algorithms allow these measures to be computed efficiently.

In machine learning, during data preprocessing, selecting the median or other ordered elements can help normalize the data, making it easier to analyze. A common method of normalization is normalization with median, in which selection algorithms play a key role.

These algorithms are also widely used in computer graphics and in optimizing sorting algorithms, such as $QuickSort$.

Languages and libraries used

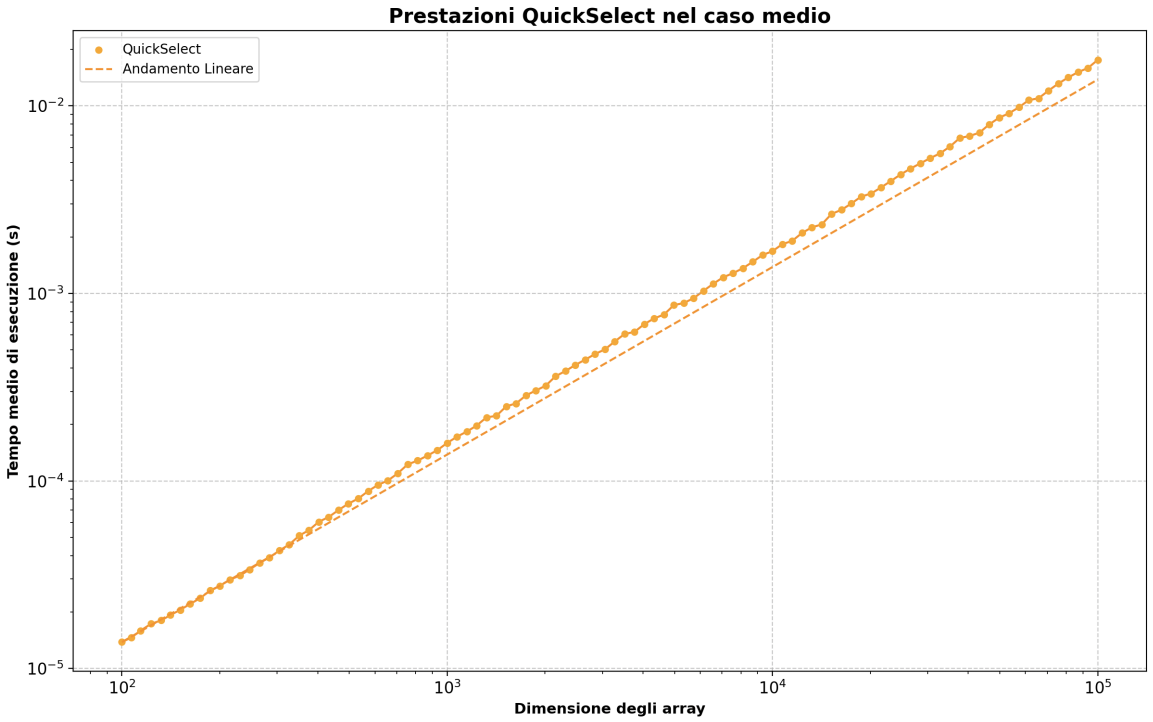
We chose to structure the entire project in $Python$ because, although it is slower than languages such as C and $C++$, it offers a very simple syntax. This choice favors readability of the code, which is our main prerogative for this project.
Again for the sake of readability, all the code is included in the dedicated “Appendix” sections and will not be included in the sections on the individual algorithms.
For the selection algorithms, no external libraries were used. For the $benchmark$, however, the following libraries were used: $random$, $time$, os, $matplotlib$.
All graphs are made first on a logarithmic scale and then on a linear scale on both axes, except for the graph 'Prestazione algoritmi di selezione al variare di k' which was made only on a linear scale.

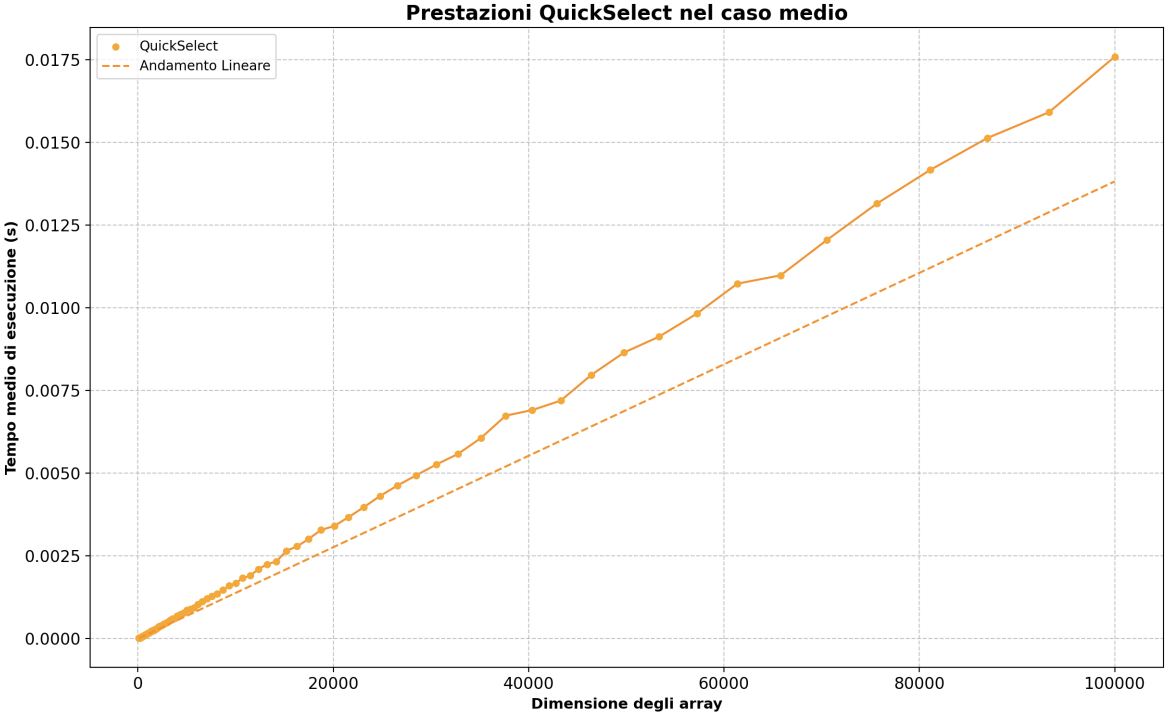
Tool used

The $benchmark$ was run in a controlled environment, on a standard 2020 $Macbook$ Air $M1$.
No other programs were running during the execution of the benchmark.
CPU : $AppleM1$
Ram : $8GB$
SSD : $250GB$
$Operating$ $System$: $macOS$ $Sonoma$ 14.5
Note that the $benchmark$ was also tested on a laptop running $Ubuntu$ $22.04.4LTS$ and was not intended to run on $Windows$.
In particular, the execution times, on $Windows$ turn out to be extremely high, probably due to the use of the $time.monotonic()$ function, moreover, in the case of execution on $Windows$, the graph will not be generated.

QuickSelect

In the implemented (iterative) version of $QuickSelect$, during the search phase for the $k-th$ element the call to $partition$ is iterated over subsets of the array of smaller and smaller size until either the pivot on which the partitioning is performed coincides with the $k-th$ element or the subarray considered is of unit size.
In fact, the complexity of the algorithm is due almost entirely to $partition$, whose job is to partition the array with respect to a pivot: moving all the smaller elements of it to its left, and all the larger ones to its right, ending with the pivot at the position it would be in if the array were sorted.
Subsequent to the call to partition: If k is less than the pivot, the call is repeated on the array partition containing the minor elements of the pivot. If k is greater than the pivot, it is reiterated on the partition containing the greater elements.
Otherwise, the pivot coincides with the $k-th$ element, so the algorithm terminates by returning the pivot.
In the best case, partition is called $\Theta(1)$ times. In that case $QuickSelect$ thus has a complexity of $\Theta(n)$.
The worst case for $QuickSelect$ occurs when the array is sorted ascending and is required to find an element whose position k is near the beginning of the array. In that case partition will be executed on an array partition that decreases by exactly one element each time. It will therefore perform about n2/2 operations, so QuickSelect will therefore have a complexity of Θ(n2). In general, we can say that QuickSelect has time complexity O(n2). However, in the average case, QuickSelect seems to approach a linear complexity more closely. Below are graphs (logarithmic and linear) of the performance of QuickSelect in the average case. On the abscissas the array size; on the ordinates the execution time. The dotted line represents the linear performance.


Prestazioni QuickSelect nel caso medio
QuickSelect
Andamento Lineare
Tempo medio di esecuzione (s)
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{2}
10^{3}
10^{4}
10^{5}
Dimensione degli array

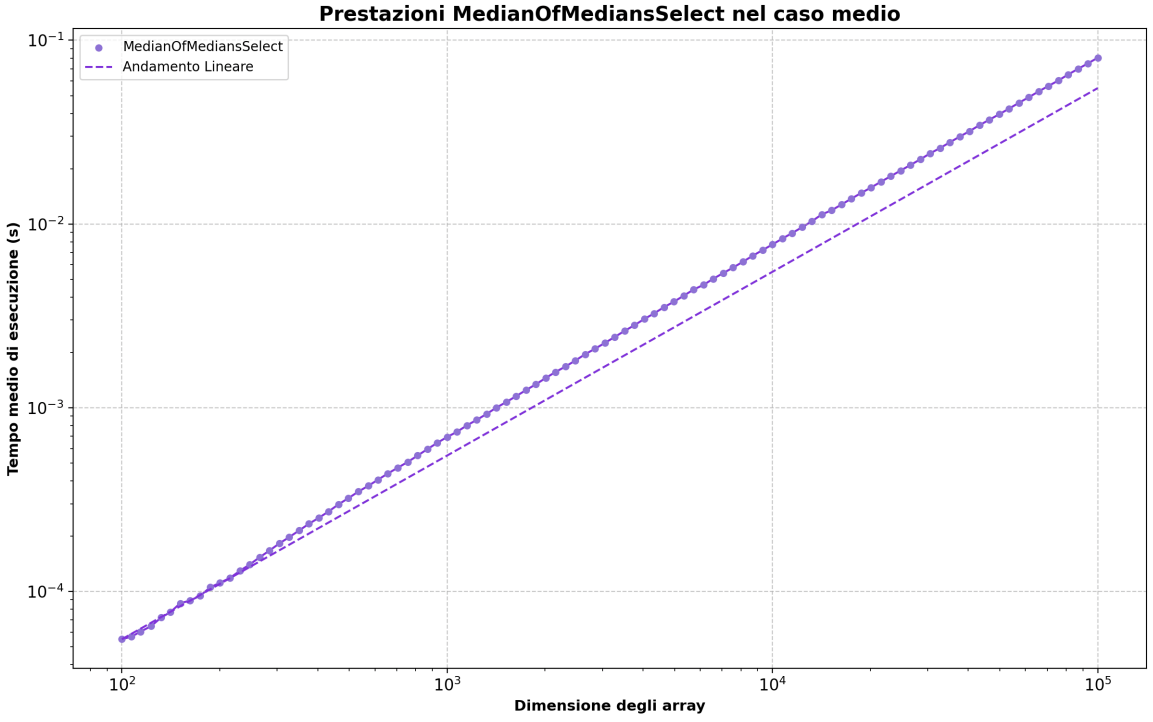

Prestazioni QuickSelect nel caso medio
QuickSelect
Andamento Lineare
Tempo medio di esecuzione (s)
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
20000
40000
60000
80000
100000
Dimensione degli array

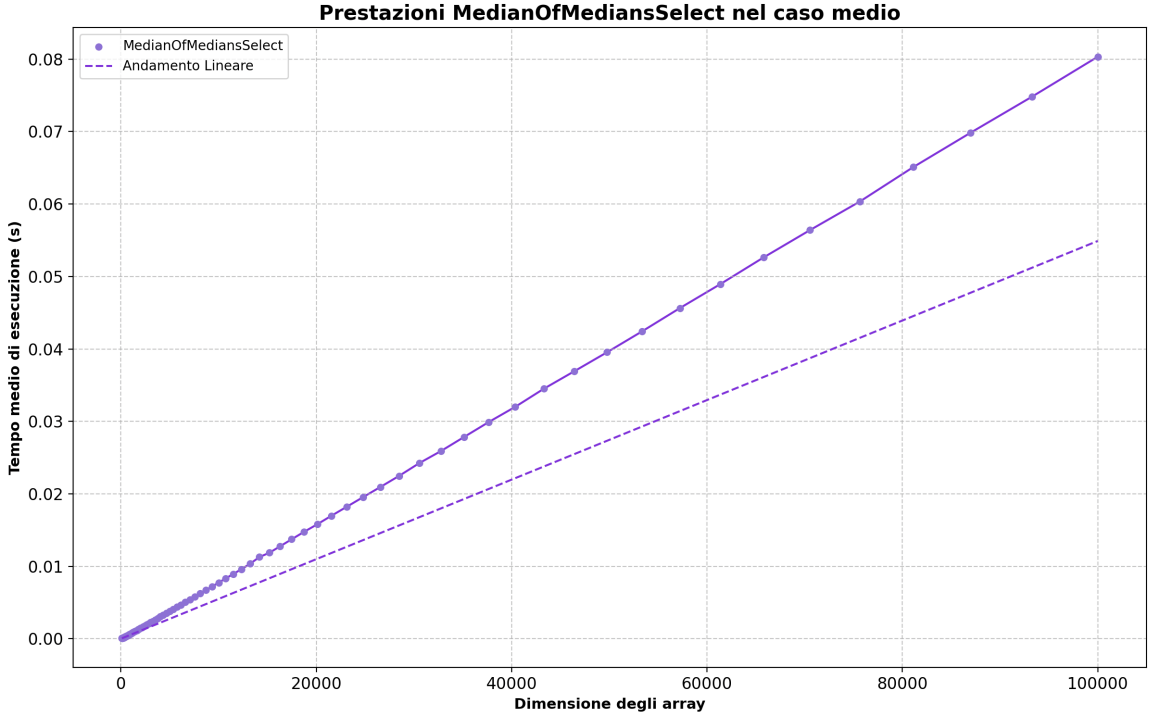
MedianOfMediansSelect

To improve $QuickSelect$, $Partition$, at each iteration, should take as pivot the median of the array (element that would be at position $k = \frac{len(A)}{2}$, if the array were ordered).
The idea of $MedianOfMediansSelect$ is based on this very concept.
This algorithm is implemented in an 'almost in place' version in that it does not use auxiliary structures, but makes use of recursive calls.
The median is found by the procedure $RecMedianOfMediansSelect$, which performs the following operations:

1. Divides the array into blocks of 5 elements and sorts them with $InsertionSort$.
2. Moves the median of each block to the top of the array.
3. Recursively calls itself in the head portion of the array, containing the medians.
4. Moves the median of the medians to the bottom of the array when it comes to consider a head sub-array of 5 elements (base case).
5. Call Partition and recursively call RecMedianOfMedians on the half of the array containing the element sought until the median of the medians is the $k-th$ element sought.

The $MedianOfMediansSelect$ algorithm has a worst-case complexity of $\Theta(n)$.
In general, we can say that it is the least variable of the three and maintains the same complexity in the medium case.
Below are graphs (logarithmic and linear) of the performance of $MedianOfMediansSelect$ in the average case. On the abscissas the array size; on the ordinates the execution time.
The dotted line represents the linear performance.



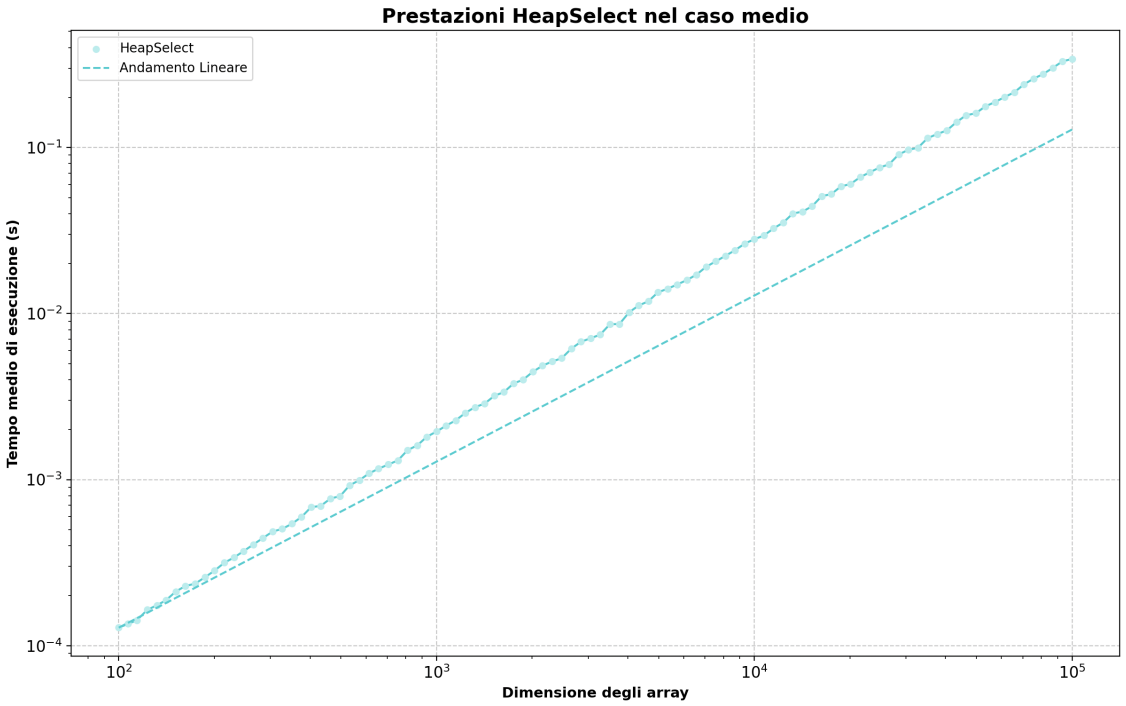


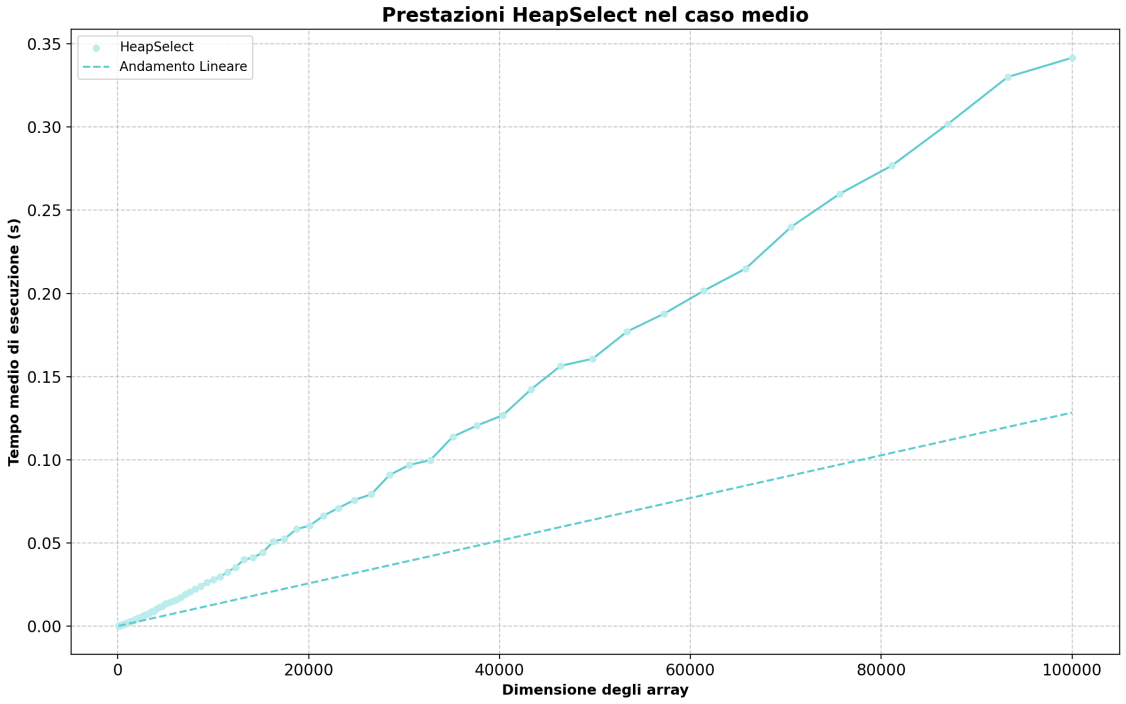
HeapSelect

To find the smallest element in a minHeap, simply return its root.
Similarly, to find the largest element in a maxHeap it is sufficient to return its root.
The algorithm exploits this property to find the $k-th$ smallest/largest element by extracting $k-1$ times the root.

Specifically, after validating the index k, if it is less than half the length of the heap, the function constructs a min-heap from $H1$ and initialises $H2$ as a min-heap containing a tuple $[element, position]$ where $element$ corresponds to the root of $H1$ and $position$ corresponds to the position of that element.
Otherwise, it constructs a max-heap from $H1$ and initialises $H2$ as a max-heap of tuples. In the latter case, the problem is reformulated to find the $k-th$ largest element, then k is recalculated as $len(H1)-k+1$.
After that, the function iterates the following process:

1. Extracts the root of $H2$.
2. Finds the left and right children of the extracted position in the heap $H1$.
3. Inserts the children found in $H2$ (if any).
   After iterating $k-1$ times, the function returns the root of $H2$ which corresponds to the $k-th$ smallest element.$HeapSelect$ has complexity $O(n+k*log(k))$ in both the worst and average cases. It should therefore be preferable to $QuickSelect$ for very small k.
   Below are graphs (logarithmic and linear) of the performance of $HeapSelect$ in the average case. On the abscissas the array size; on the ordinates the execution time.
   The dotted line represents the linear performance.





Benchmark

The $benchmark$ measures the average execution time of each algorithm in the $algorithmOfSuccession$ array.
It generates 100 ($stepSuccession$) arrays of increasing size following a geometric series. The size of the arrays ranges from 100 to 100000.
At each step, for each of the algorithms:

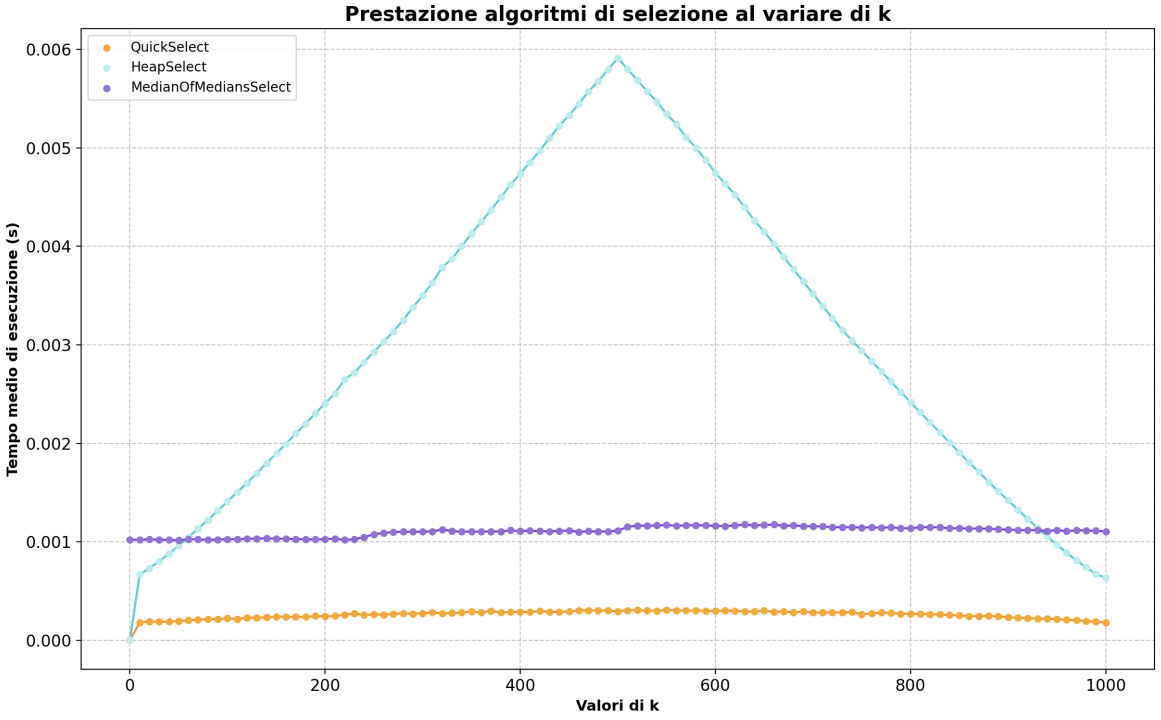
1. 500 ($testPerEachN$) tests are run with k random, to ensure that the execution times of the algorithms are averaged.
2. in a matrix ($timesAverage$), the following are stored:
   1) size of the array considered
   2) average execution times
   At each test:
3. the array is filled with random values in a range $[-1000, 1000]$.
4. the execution time is measured (for each of the algorithms) ensuring a relative error of less than 1%.
5. average execution times are saved in a matrix ($averagetimes$).
   At the end of the 100 steps, the average times data are saved in separate files: one for each algorithm.
   Finally, the function is called that draws the graph by taking the data from the generated files.

Experiments and results

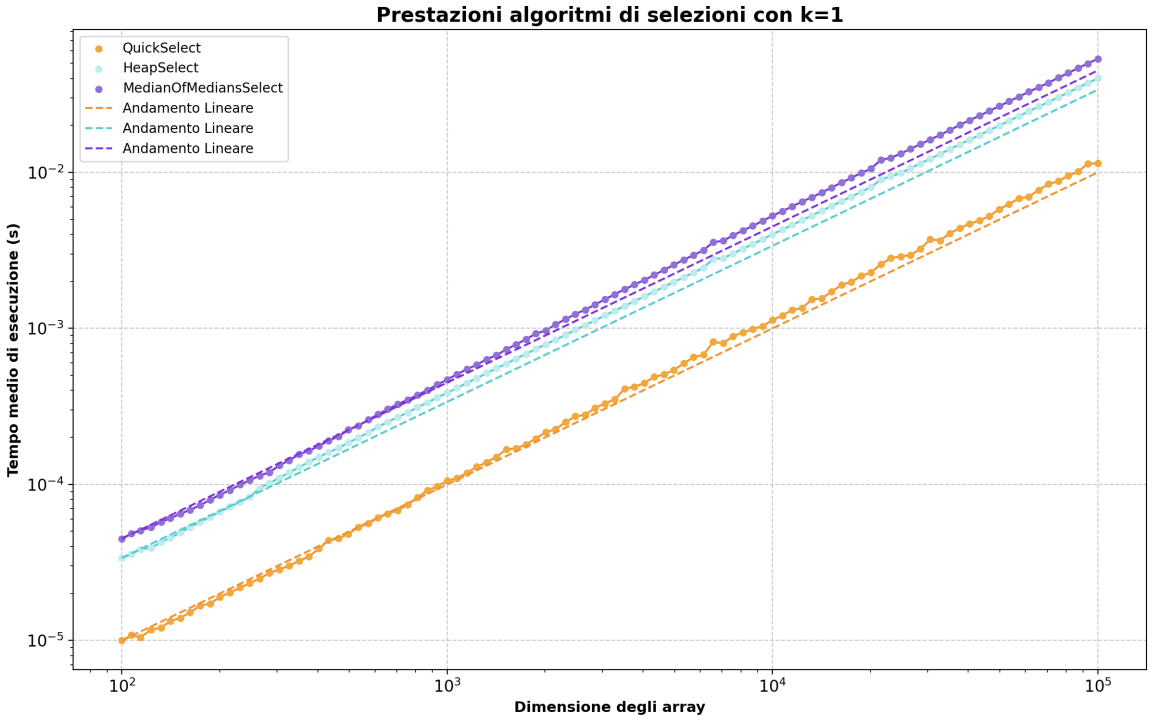
To evaluate the dependence on the chosen parameter k, we performed 4 further tests.

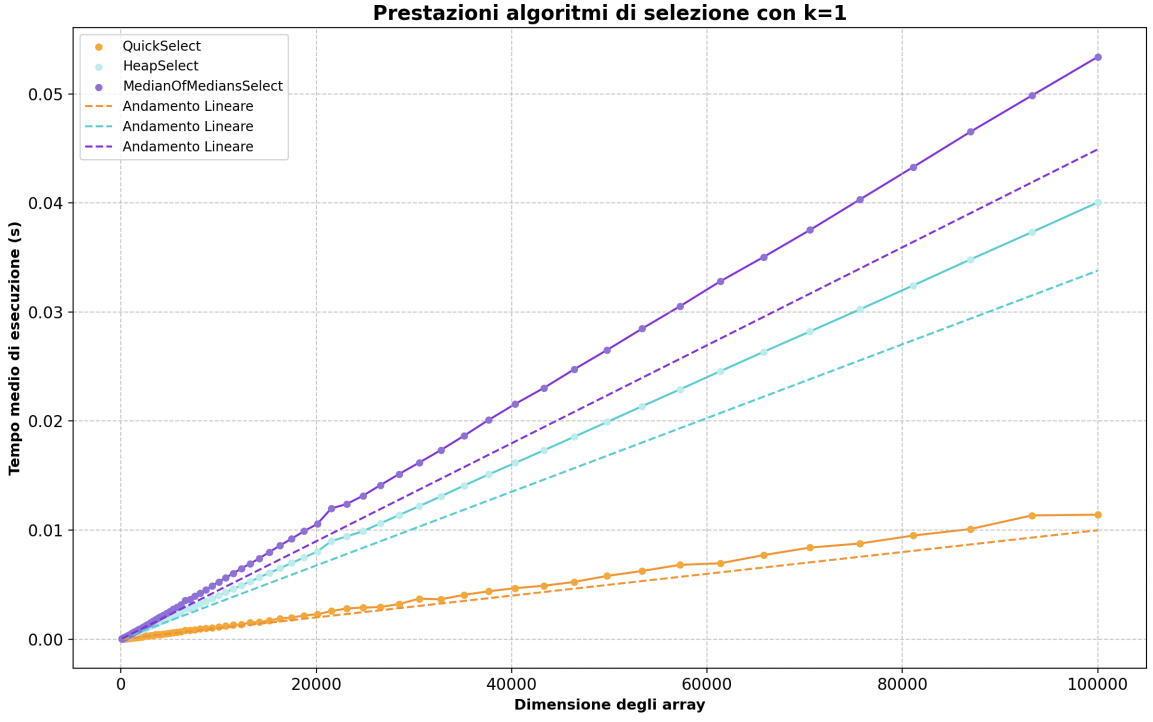
1. With the array size fixed at 1000 and $k = 0, 10, 20, \ldots, 1000$
2. By fixing $k = 1$
3. Setting $k = lenArray//2$
4. Fixing $k = lenArray$

In the first test, we obtained the following graph from which the k dependence of $HeapSelect$ is particularly evident:
On the abscissas the k value; on the ordinates the execution time.

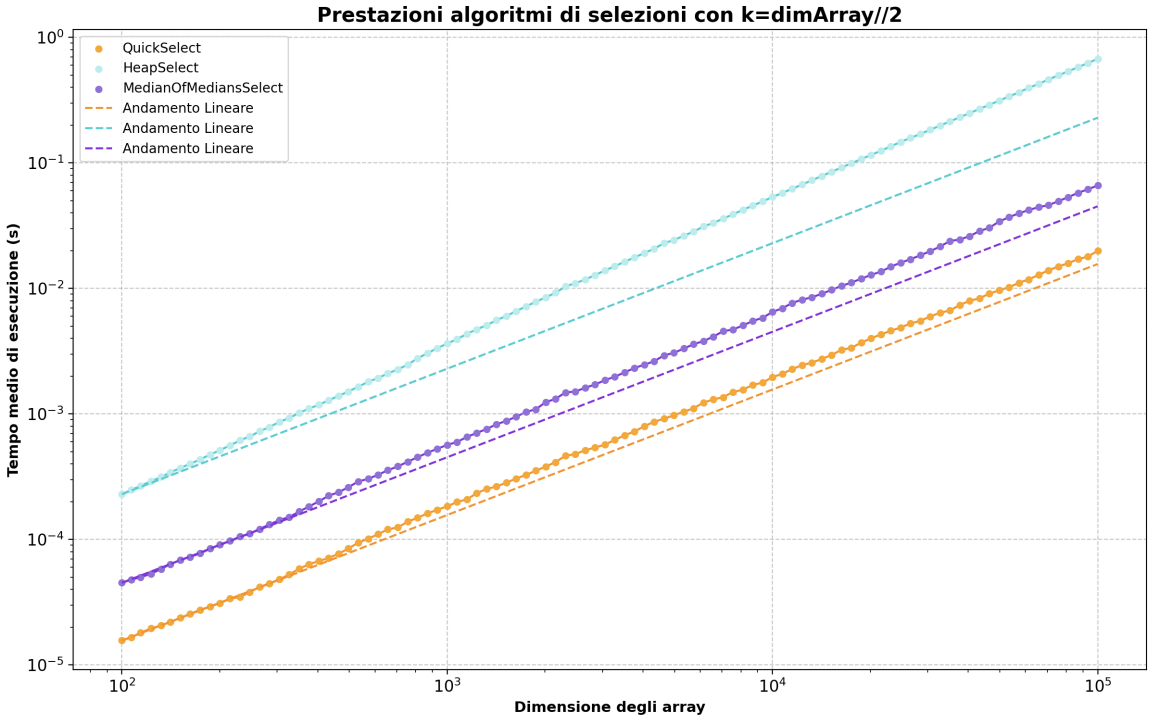


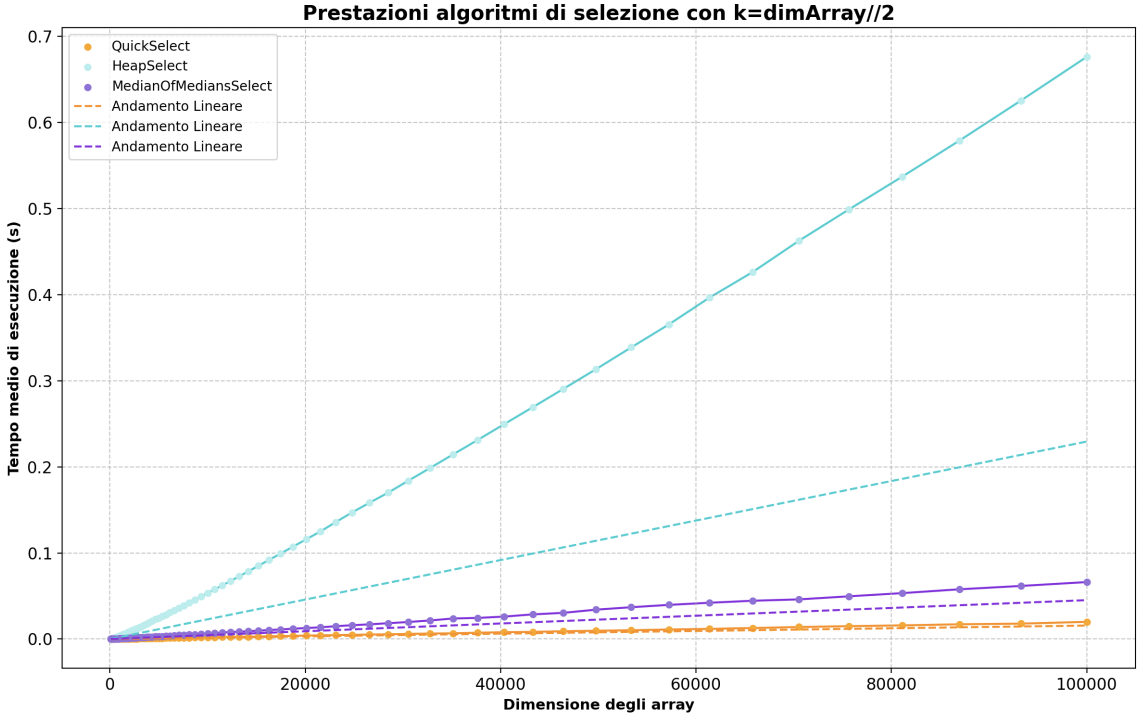
For $k = 1$ we fall into the best possible case for $HeapSelect$, which is super linear and less variable than the average case.
On the abscissas the array size, on the ordinates the execution time.



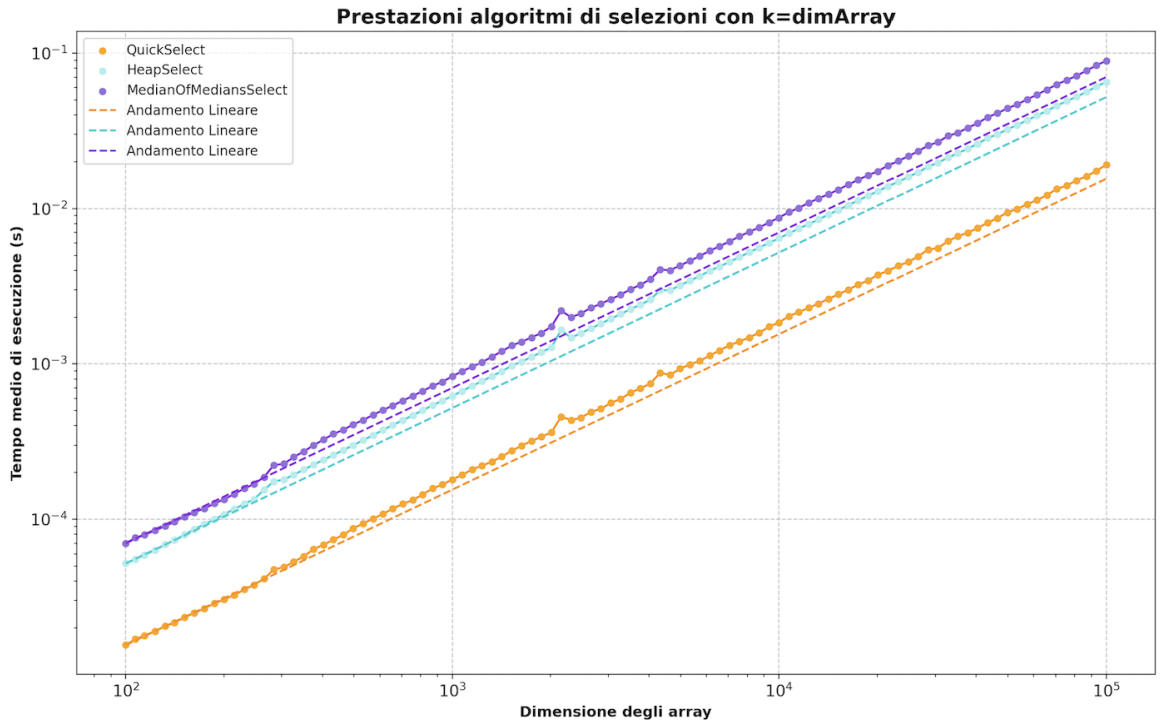


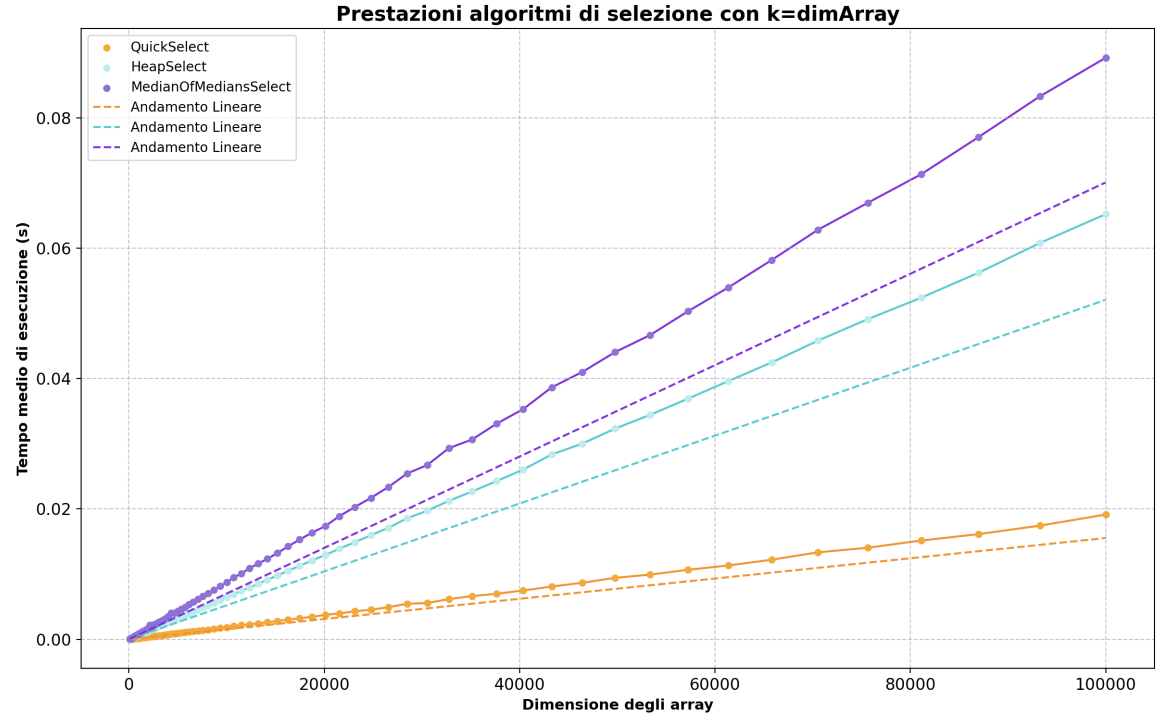
For $k = len(array)//2$ we fall into the worst case for $HeapSelect$. In fact, the execution times are higher, the trend remains very similar to that of the average case.
$QuickSelect$ and $MedianOfMediansSelect$ on the other hand turn out to be unchanged.
On the abscissas the array size, on the ordinates the execution time.



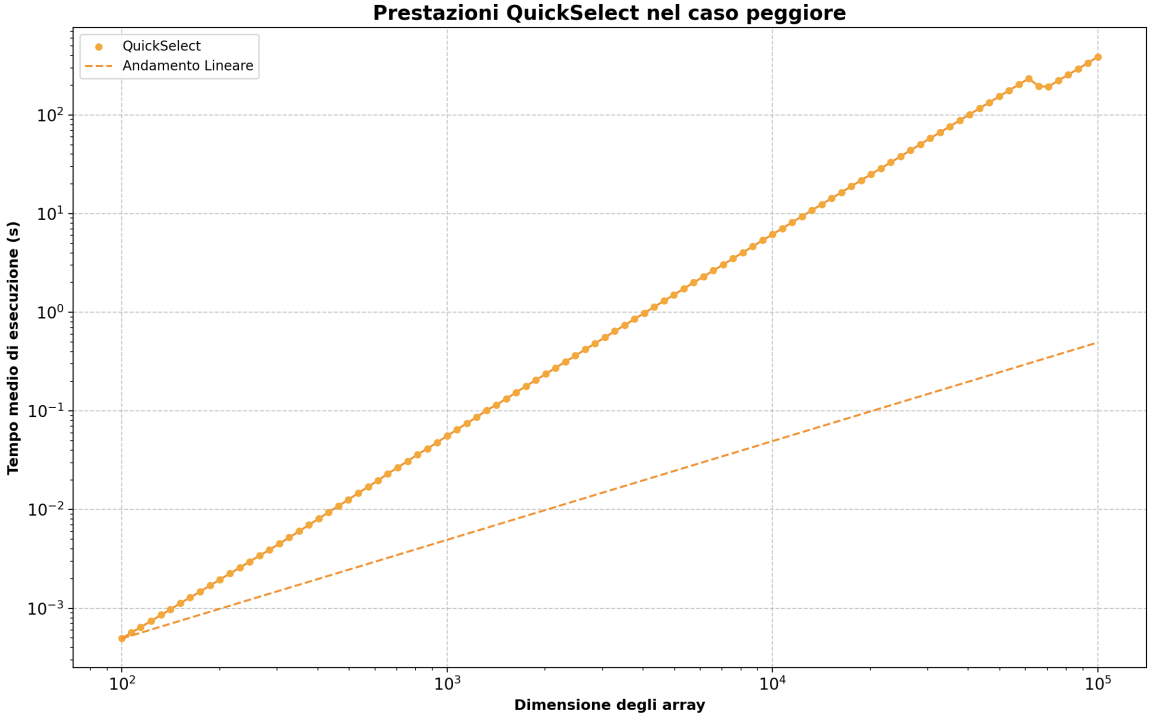


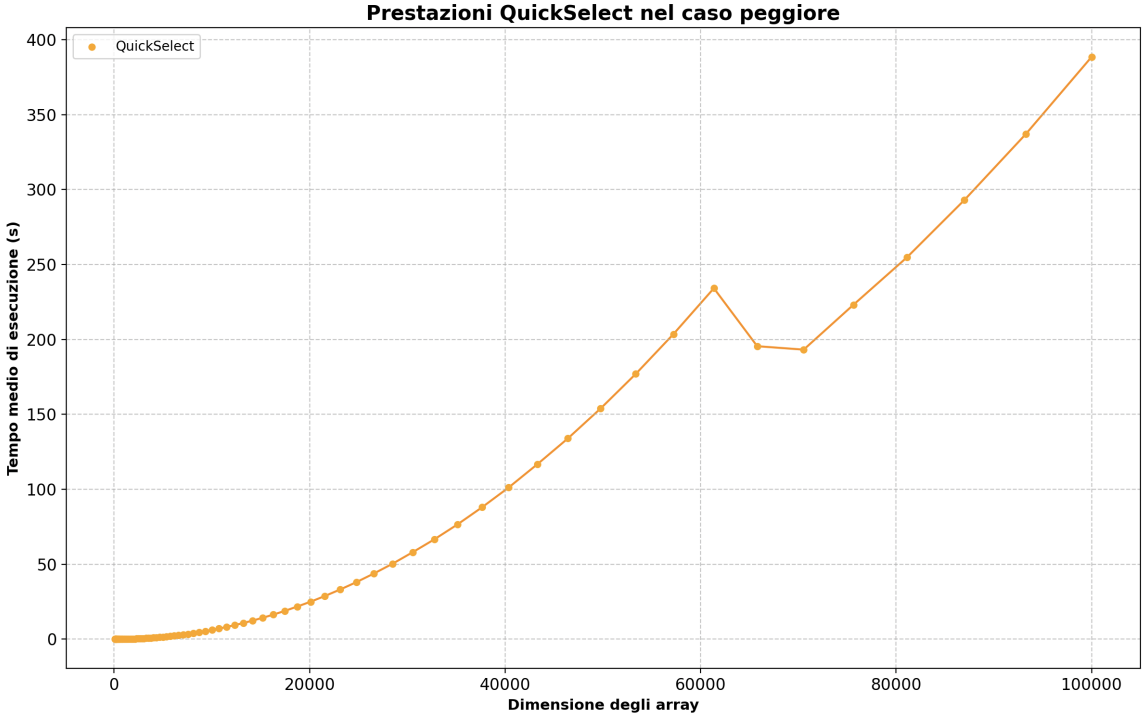
For $k = lenArray$ we obtain an analogous case to $k = 1$, best case for $HeapSelect$.
On the abscissas the array size, on the ordinates the execution time.



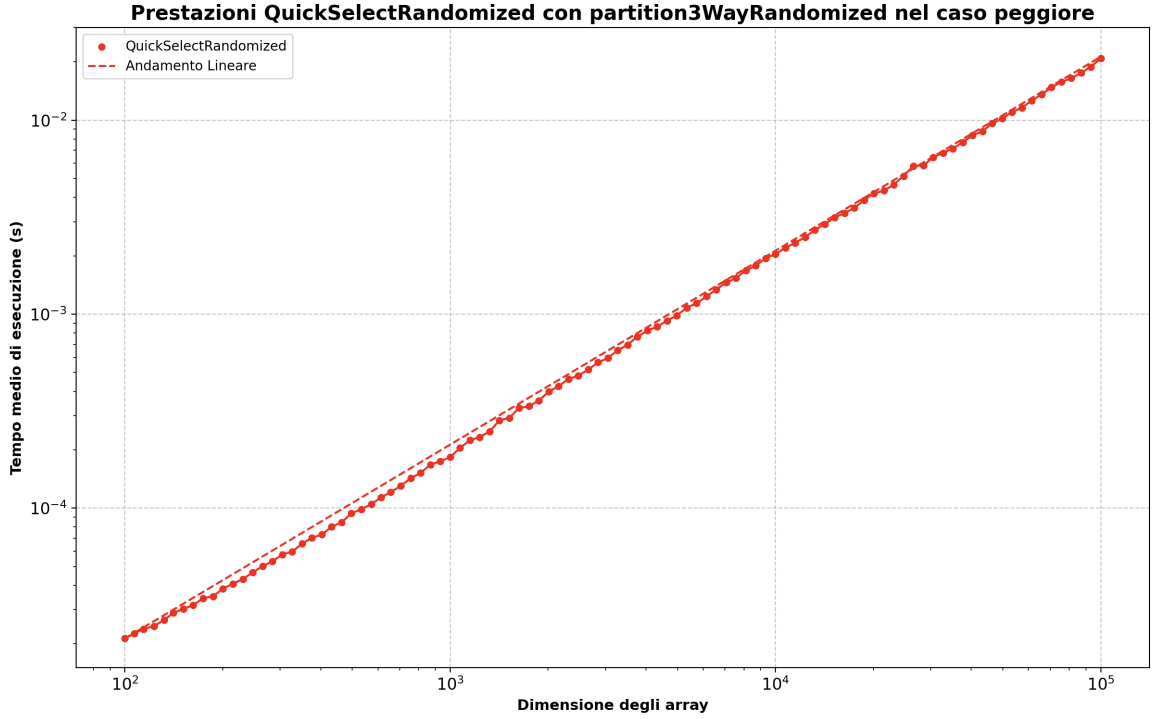


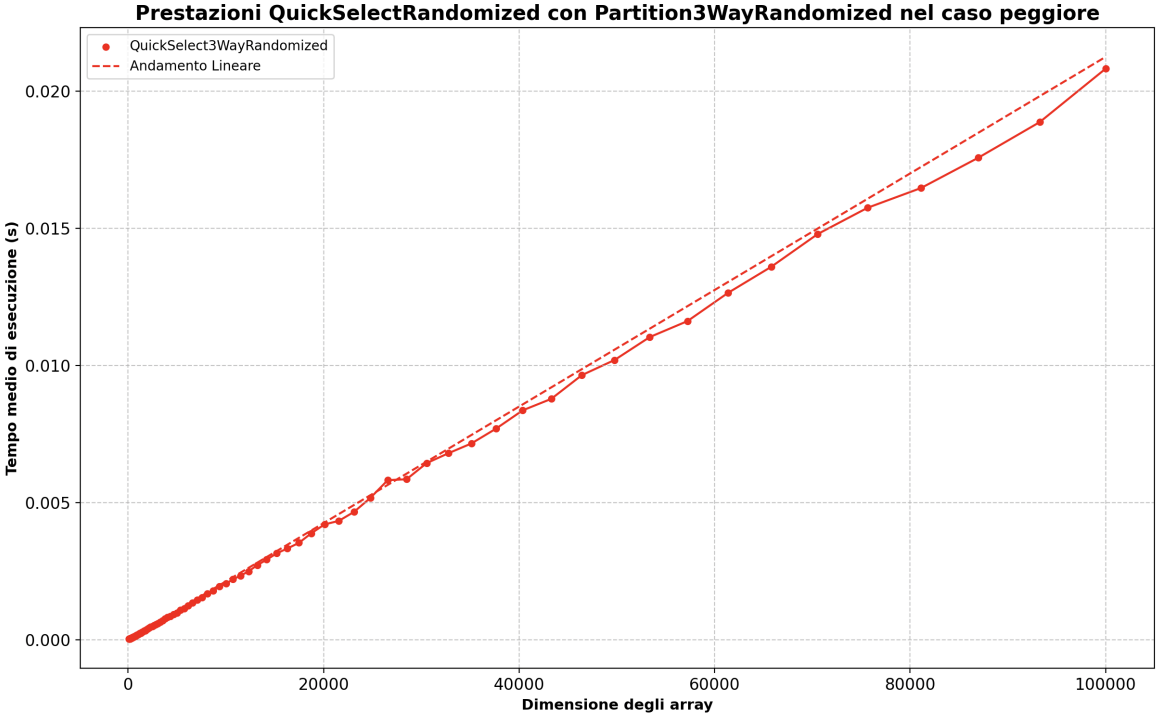
We have seen that $QuickSelect$ is not particularly bound to k. We then proceeded with the analysis of execution time in its worst case: with arrays sorted in ascending order, $k = 1$ and $testPerEveryN = 20$ (instead of 500 as the benchmark execution time in that case would have been ridiculously higher). Furthermore, having fixed k, and using always ordered arrays, the variability of performance was reduced, so there was no need to run so many $testPerEveryN$.
Fun fact: for the array of 100000 elements, the average execution time was 388.4449569665987 seconds.
On the abscissas the array size, on the ordinates the execution time.





We then wanted to try to improve the complexity of $QuickSelect$ by using $partition3WayRandomized$ and simulating the same worst-case conditions for $QuickSelect$(with $partition$ normal), i.e. using arrays sorted in ascending direction and $k = 1$, to show that in that case, this version is much more efficient.
This test, like all the others, with the exception of the worst-case $QuickSelect$ test, was performed with $500\ testPerOneN$.
It is important to emphasise that the proposed conditions do not represent the worst case for $QuickSelectRandomised$. In fact, its worst case is impossible to determine since the pivots of $partition3WayRandomized$ are chosen randomly.

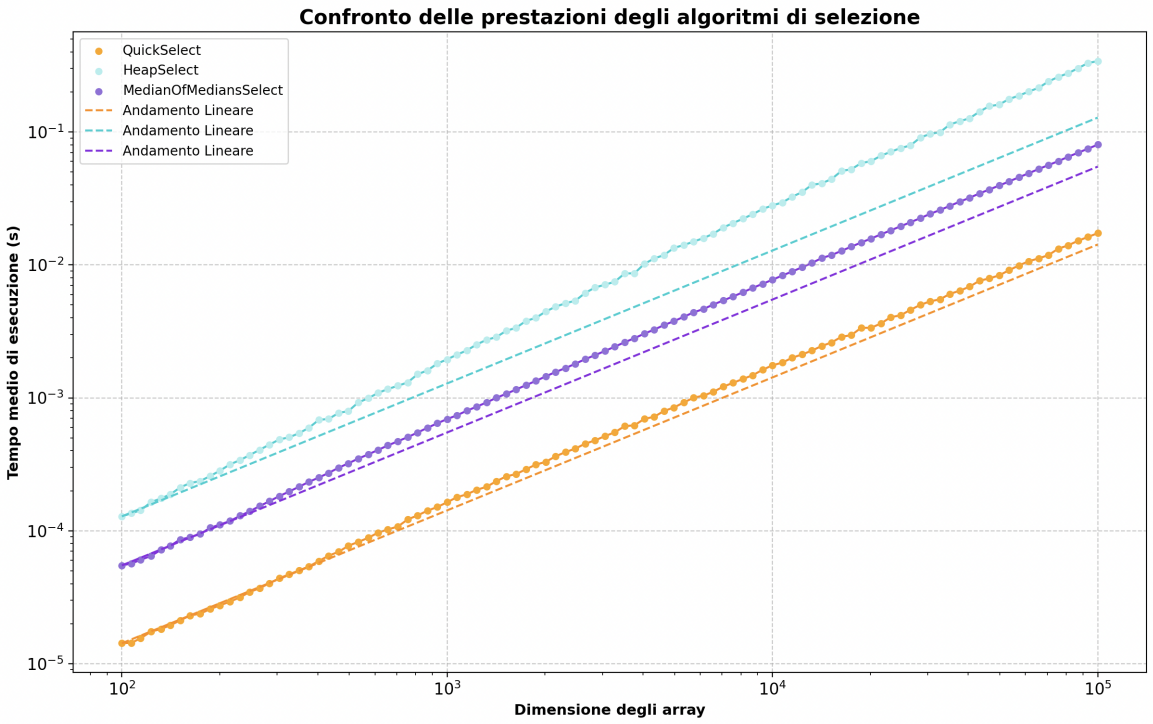


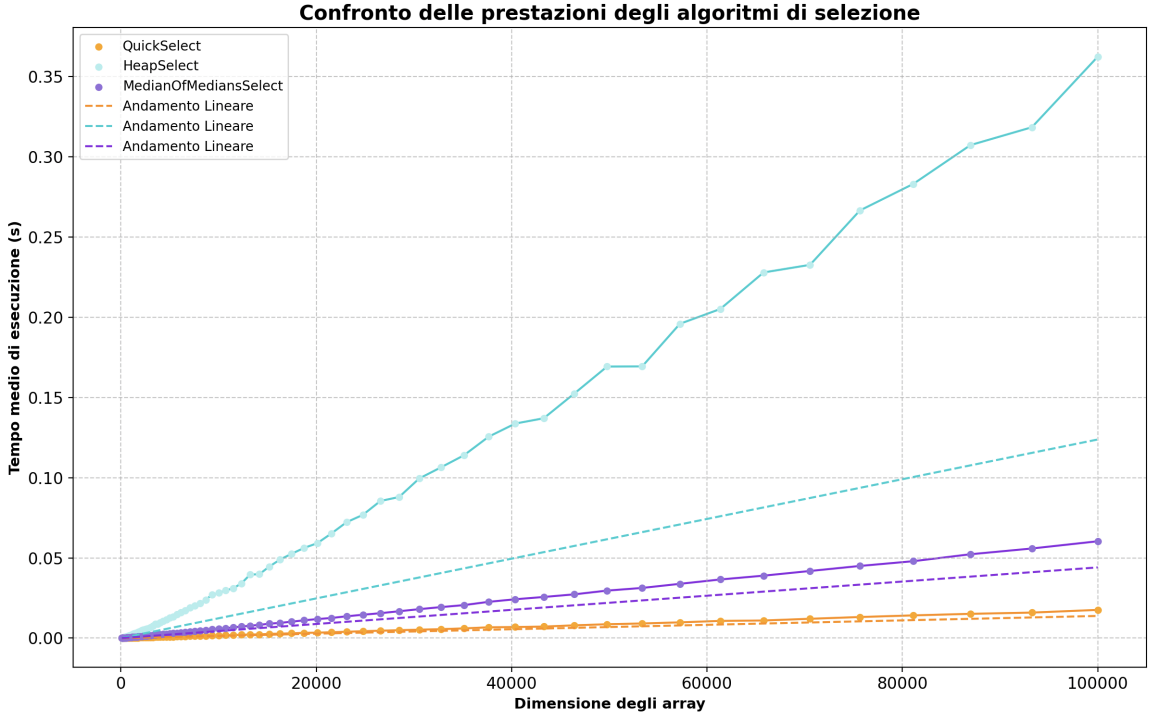


Conclusions and final graph

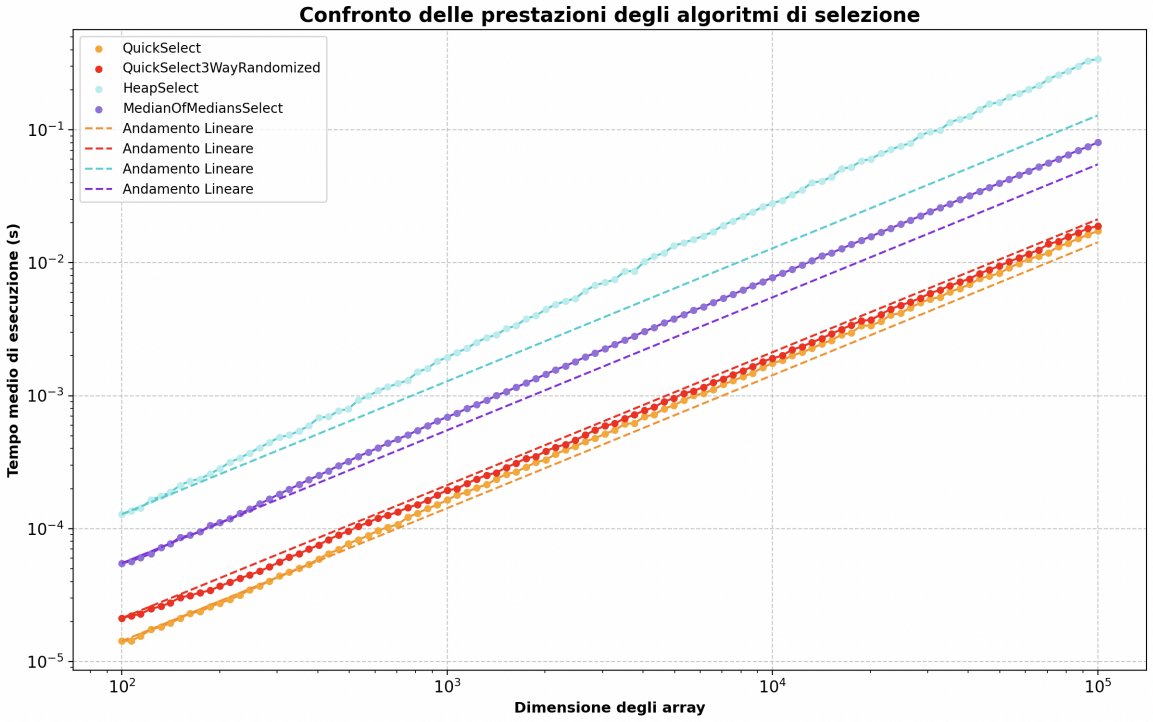
Looking only at the complexities of the three proposed algorithms, one might think that $QuickSelect$ is the worst of the three, in fact in the worst case it has a complexity of $\Theta(n^2)$.
However, it is curious to note from the graph, that for random arrays, in the average case, the most efficient algorithm is indeed $QuickSelect$, followed by $MedianOfMediansSelect$, $HeapSelect$.

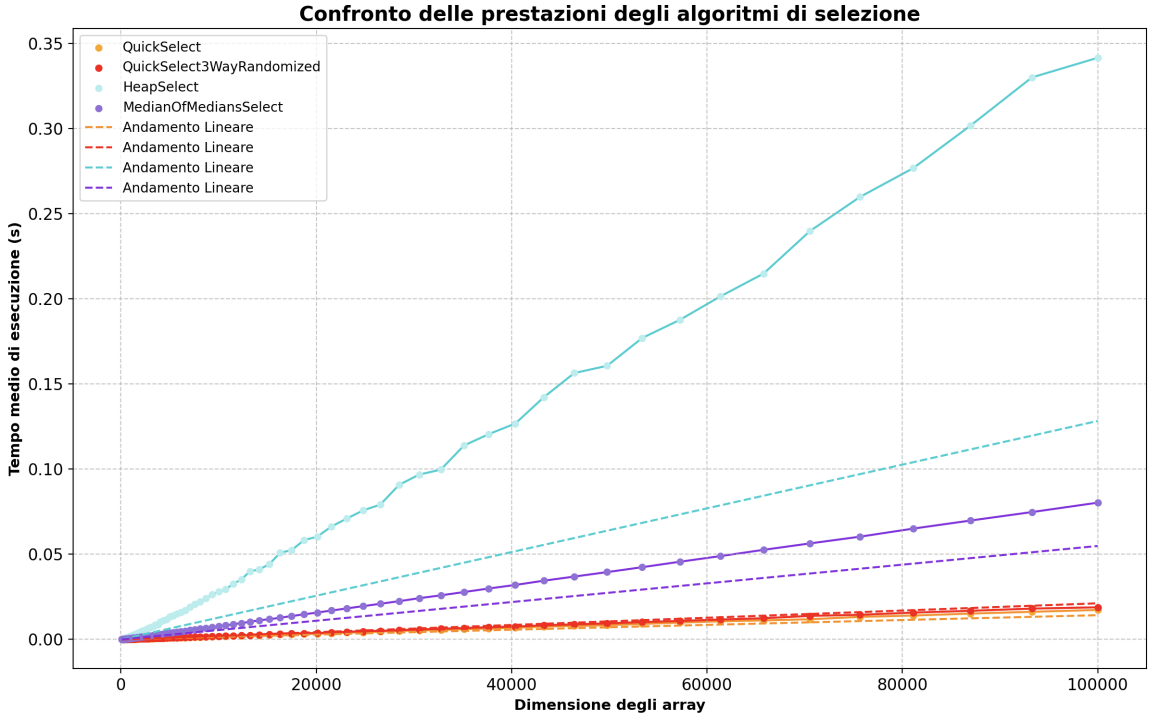
The $benchmark$ to generate this graph ran 500 tests for each array size, ensuring a good average case for each algorithm.





Using $QuickSelectRandomized$ with $partition3WayRandomized$ in the average case we obtained the following graphs:





Problems encountered and solutions found

1. When testing $QuickSelect$ in its worst case, with ordered arrays of more than 997 elements and $k = 1$, the execution ended with a 'recursion depth' error because python could not compute all those recursions. We therefore transformed the recursive programme into an iterative one. All the graphs in this report were generated by the iterative version.
2. Initially, an integer key was used for $HeapSelect$ instead of a tuple $[element, position]$ to represent a node.
   However, this caused 2 main problems:
   1. In the case of repeated elements, a linear scan of the array was used to obtain the position of that key, which returned the position of the first occurrence of the specified element, thus only ever obtaining the children of the node that was needed first. This was solved by keeping track of the number of times the extracted keys appeared, with an occurrence map.
   2. Having only the element keys available, we were forced to scroll through the array whenever we needed to find the position of one of them (to access its children).
      By using a tuple $[element, position]$, the position of the children can be calculated immediately upon occurrence.
3. Initially for the $benchmark$ the idea was to save the data generated in an array to be passed to the function drawing the graph. However, we decided to save the data generated by the $benchmark$ for each algorithm in separate files, just in case we wanted to:
   1. Generate aesthetically different graphs.
   2. Create comparative graphs for the different algorithms.
      In addition, very few tests were generated for each array at the beginning. This caused a very bad average case, especially for $QuickSelect$, whose variability is rather high as it is very dependent on both the choice of parameter k and the arrangement of the elements in the array. For this reason, we chose to run the $benchmark$ with 500 tests for each generated array size.
4. The CPU of the macbook Air on which we ran the benchmark reached rather high temperatures during the tests, so processor frequencies were also reduced and algorithm execution times increased more than expected with increasing array size until they stabilised from a certain point onwards. This, however, made the graphs inaccurate. This was a minor but easily avoidable distortion. Ridiculous as it sounds, we solved the problem by running the benchmark while holding the laptop over a running air conditioner in a university lecture hall. By combining the active dissipation provided by the air conditioner and the passive dissipation provided by the aluminium case, we greatly reduced the temperature and ensured more accurate results.

QuickSelect Appendix

```
'''
quickSelect:
    Given an array and an index k:
    1) Returns the k-th smallest element in the array.
    2) Returns None if k is not an acceptable value
'''
def quickSelect(A,k):
    #validate index k
    if k<1 or k>len(A):
        return None #If k is not acceptable print None
    else:

        #Initialisation of variables
        start = 0
        end = len(A) - 1
        k -= 1

        #search for the k-th element
        while start < end:
            pivotPos = partition(A, start, end)
            if k >= start and k <= pivotPos-1: #if I have to search before pivot
                end = pivotPos - 1
            elif k >= pivotPos+1 and k <= end: #if I need to search after
                start = pivotPos + 1
            else: #if p = k-1
                break

        #return the result
        return A[k]

'''
partition:
    Given an array and a range (p, q):
    1) Move to the left of the pivot all the minor elements of the pivot.
    2) Moves all major elements of the pivot to the right of the pivot.
    3) Returns the position of the pivot.
'''
def partition(A,p,q):
    valuePivot=A[q]
    i=p-1
    for j in range(p,q+1):
        if A[j]<=valuePivot:
            i+=1
            swap(A,i,j)
    return i

'''
swap:
    Given an array and two positional indices:
    1) Swap the element at position i with the one at position j
'''
def swap(A,i,j):
    temp=A[i]
    A[i]=A[j]
    A[j]=temp

def partition3WayRandomized(A, start, end):
    pos_a = partition(A, start, end, random.randint(start, end))
    if pos_a < (start + end)//2:
        if pos_a != end:
            pos_b = partition(A, pos_a+1, end, random.randint(pos_a+1, end))
        else:
            pos_b = end
    else:
        if pos_a != start:
            pos_b = partition(A, start, pos_a-1, random.randint(start, pos_a-1))
        else:
            pos_b = start

    if pos_a < pos_b:
        return (pos_a, pos_b)
    else:
        return (pos_b, pos_a)


def quickSelectRandomized(A, k):
    # Validation of the index k
    if k < 1 or k > len(A):
        return None  # If k is not valid, return None
    else:
        # Initialization of variables
        start = 0
        end = len(A) - 1
        k -= 1
```

```
        # Search for the k-th element
        while start < end:
            pos_a, pos_b = partition3WayRandomized(A, start, end)
            if k == pos_a or k == pos_b:
                break
            elif k < pos_a:
                end = pos_a
            elif k < pos_b:
                start = pos_a
                end = pos_b
            else:
                start = pos_b

        # Return the result
        return A[k]
```

MedianOfMediansSelect Appendix

```
'''
partition:
    Given an array and a range (p, q):
    1) Place the element at position q in the central position of the array.
    2) Place all elements smaller than the pivot to its left.
    3) Place all elements greater than the pivot to its right.
'''
def partition(A, p, q):
    pivot = A[q]
    i = p-1
    for j in range(p, q+1):
        if A[j] < pivot:
            i = i+1
            swap(A, i, j)
    i+=1
    swap(A, i, q)
    return i

'''
swap:
    Given an array and two positional indices:
    1) Swap the element at position i with the one at position j.
'''
def swap(A, i, j):
    t = A[i]
    A[i] = A[j]
    A[j] = t

'''
insertionSort:
    Given an array and a range (start, end):
    1) Sort the section of the array defined by the range.
    2) Does not return the array. It operates on the array passed as a parameter.
'''
def insertionSort(A, start, end):
    for i in range(start, end+1):
        j = i
        while j > start and A[j] < A[j-1]:
            swap(A, j, j-1)
            j-=1

'''
recMedianOfMediansSelect:
    Given an array, a range, and a positional index k:
    1) Divide the array into blocks of 5 elements (except the last block, which may have fewer) and sort them using InsertionSort.
    2) Move the median of each block to the beginning of the array.
    3) Recursively call itself on the head portion of the array containing the medians.
    4) Move the median of medians to the end of the array when considering a head sub-array of 5 elements (base case).
    5) Call Partition and recursively invoke RecMedianOfMedians on the half of the array containing the desired element until the median of medians is the k-th desired element.
'''
def recMedianOfMediansSelect(A, start, end, k):
    if k<0 or k>len(A):
        return -1
    if end-start+1<=5:
        insertionSort(A, start, end)
        return(A[k-1])
    else:
        posMediano=start
        for x in range(start+4, end, 5):
            insertionSort(A, x-4, x)
            swap(A, posMediano, x-2)
            posMediano+=1
            if end-x<=5:
                insertionSort(A, x+1, end)
                swap(A, posMediano, (end+x)//2)
        recMedianOfMediansSelect(A, start, posMediano, (posMediano+start+1)//2)
        swap(A, end, (posMediano+start+1)//2)

    pivotPos = partition(A, start, end)
    if pivotPos == k-1:
        return A[k-1]
    elif pivotPos < k-1:
        start = pivotPos + 1
    else:
        end = pivotPos - 1
    return recMedianOfMediansSelect(A, start, end, k)

'''
medianOfMediansSelect:
    Helper function used only to initialize the parameters to start the recursion tree.
'''
def medianOfMediansSelect(A, k):
    start = 0
    end = len(A)-1
    return recMedianOfMediansSelect(A, start, end, k)
```

HeapSelect Appendix

```
from Heap import maxHeapInt, maxHeapTuple, minHeapInt, minHeapTuple
```

```
def heapSelect(H1, k):
    if len(H1) > 0 and k > 0 and k <= len(H1):
        if k < len(H1) // 2:

            minHeapInt.buildHeap(H1)
            H2 = minHeapTuple([])
        else:

            maxHeapInt.buildHeap(H1)
            H2 = maxHeapTuple([])
            k = len(H1) - k + 1
        # Node of H2 are tuple that have sa 1° element the a key of H1
        # and as 2° the position of that key in H1
        H2.push((H1[0], 0))


        for i in range(1, k):
            node = H2.pop()
            l = maxHeapInt.getLeft(H1, node[1])
            r = maxHeapInt.getRight(H1, node[1])
            if l[0] != None : H2.push(l)
            if r[0] != None : H2.push(r)


        return H2.pop()[0]
    else:
        return -1
```

```
"""
Set of function that modifyes the given array
- Organize the array satisfing MaxHeap invariant
- Perform Heap operation
"""
class maxHeapInt:
    def getLeft(H, i):
        lPos = 2 * i + 1
        if i >= 0 and lPos < len(H):
            return (H[lPos], lPos)
        else:
            return (None, -1)

    def getRight(H, i):
        rPos = 2 * i + 2
        if i >= 0 and rPos < len(H):
            return (H[rPos], rPos)
        else:
            return (None, -1)

    def getParent(H, i):
        pPos = (i - 1) // 2
        if i > 0 and i < len(H):
            return (H[pPos], pPos)
        else:
            return (None, -1)

    def buildHeap(H):
        for i in range((len(H) // 2) - 1, -1, -1):
            maxHeapInt.__heapify(H, i)

    def __heapify(H, i):
        l = maxHeapInt.getLeft(H, i)
        r = maxHeapInt.getRight(H, i)
        if l[0] != None and l[0] > H[i]:
            m = l[1]
        else:
            m = i
        if r[0] != None and r[0] > H[m]:
            m = r[1]
        if m != i:
            maxHeapInt.__swap(H, i, m)
            maxHeapInt.__heapify(H, m)

    def __swap(H, i, j):
        tmp = H[i]
        H[i] = H[j]
        H[j] = tmp


class minHeapInt:
    def getLeft(H, i):
        lPos = 2 * i + 1
        if i >= 0 and lPos < len(H):
            return (H[lPos], lPos)
        else:
            return (None, -1)

    def getRight(H, i):
        rPos = 2 * i + 2
        if i >= 0 and rPos < len(H):
            return (H[rPos], rPos)
        else:
            return (None, -1)

    def getParent(H, i):
        pPos = (i - 1) // 2
        if i > 0 and i < len(H):
            return (H[pPos], pPos)
        else:
            return (None, -1)

    def buildHeap(H):
        for i in range(len(H) // 2, -1, -1):
            minHeapInt.__heapify(H, i)

    def __heapify(H, i):
        l = minHeapInt.getLeft(H, i)
        r = minHeapInt.getRight(H, i)
        if l[0] != None and l[0] < H[i]:
            m = l[1]
        else:
```

```
            m = i
        if r[0] != None and r[0] < H[m]:
            m = r[1]
        if m != i:
            minHeapInt.__swap(H, i, m)
            minHeapInt.__heapify(H, m)

    def __swap(H, i, j):
        tmp = H[i]
        H[i] = H[j]
        H[j] = tmp


"""
Classes that realize Heap with node that can store indexed element
the 1° element is considered for the Heap organization (to preserve the invarint)
"""
class maxHeapTuple:
    def __init__(self, A):
        self.H = A
        self.__buildHeap()

    def getHeapSize(self):
        return len(self.H)

    def getLeftPosition(self, pos):
        l = 2 * pos + 1
        if pos >= 0 and l < self.getHeapSize():
            return l
        else:
            return -1

    def getRightPosition(self, pos):
        r = 2 * pos + 2
        if pos >= 0 and r < self.getHeapSize():
            return r
        else:
            return -1

    def getParentPosition(self, pos):
        if pos > 0 and pos < self.getHeapSize():
            return (pos - 1) // 2
        else:
            return -1

    def getNode(self, pos):
        if pos >= 0 and pos < self.getHeapSize():
            return self.H[pos]
        else:
            return None

    def getLeft(self, pos):
        l = self.getLeftPosition(pos)
        if l != -1:
            return self.getNode(l)
        else:
            return None

    def getRight(self, pos):
        r = self.getRightPosition(pos)
        if r != -1:
            return self.getNode(r)
        else:
            return None

    def getParent(self, pos):
        p = self.getParentPosition(pos)
        if p != -1:
            return self.getNode(p)
        else:
            return None

    def __buildHeap(self):
        for pos in range(self.getHeapSize()//2, -1, -1):
            self.__heapify(pos)

    def __heapify(self, pos):
        l = self.getLeftPosition(pos)
        r = self.getRightPosition(pos)
        if l != -1 and self.getNode(l)[0] > self.getNode(pos)[0]:
            m = l
        else:
            m = pos
        if r != -1 and self.getNode(r)[0] > self.getNode(m)[0]:
            m = r
        if m != pos:
            self.__swap(m, pos)
            self.__heapify(m)

    def __swap(self, i, j):
        tmp = self.getNode(i)
        self.H[i] = self.getNode(j)
        self.H[j] = tmp

    def push(self, node):
        self.H.append(node)
        pos = self.getHeapSize() - 1
        while pos > 0 and self.getNode(pos)[0] > self.getParent(pos)[0]:
            self.__swap(pos, self.getParentPosition(pos))
            pos = self.getParentPosition(pos)

    def pop(self):
        if self.getHeapSize() > 0:
            node = self.getNode(0)
            self.__swap(0, self.getHeapSize() - 1)
            self.H.pop(self.getHeapSize() - 1)
            self.__heapify(0)
            return node
        else:
            return None
```

```
    def __str__(self):
        str = "[ "
        for x in self.H:
            str += "({}, {}) ".format(x[0], x[1])
        str += "]"
        return str


class minHeapTuple:
    def __init__(self, A):
        self.H = A
        self.__buildHeap()

    def getHeapSize(self):
        return len(self.H)

    def getLeftPosition(self, pos):
        l = 2 * pos + 1
        if pos >= 0 and l < self.getHeapSize():
            return l
        else:
            return -1

    def getRightPosition(self, pos):
        r = 2 * pos + 2
        if pos >= 0 and r < self.getHeapSize():
            return r
        else:
            return -1

    def getParentPosition(self, pos):
        if pos > 0 and pos < self.getHeapSize():
            return (pos - 1) // 2
        else:
            return -1

    def getNode(self, pos):
        if pos >= 0 and pos < self.getHeapSize():
            return self.H[pos]
        else:
            return None

    def getLeft(self, pos):
        l = self.getLeftPosition(pos)
        if l != -1:
            return self.getNode(l)
        else:
            return None

    def getRight(self, pos):
        r = self.getRightPosition(pos)
        if r != -1:
            return self.getNode(r)
        else:
            return None

    def getParent(self, pos):
        p = self.getParentPosition(pos)
        if p != -1:
            return self.getNode(p)
        else:
            return None

    def __buildHeap(self):
        for pos in range(self.getHeapSize()//2, -1, -1):
            self.__heapify(pos)

    def __heapify(self, pos):
        l = self.getLeftPosition(pos)
        r = self.getRightPosition(pos)
        if l != -1 and self.getNode(l)[0] < self.getNode(pos)[0]:
            m = l
        else:
            m = pos
        if r != -1 and self.getNode(r)[0] < self.getNode(m)[0]:
            m = r
        if m != pos:
            self.__swap(m, pos)
            self.__heapify(m)

    def __swap(self, i, j):
        tmp = self.getNode(i)
        self.H[i] = self.getNode(j)
        self.H[j] = tmp

    def push(self, node):
        self.H.append(node)
        pos = self.getHeapSize() - 1
        while pos > 0 and self.getNode(pos)[0] < self.getParent(pos)[0]:
            self.__swap(pos, self.getParentPosition(pos))
            pos = self.getParentPosition(pos)

    def pop(self):
        if self.getHeapSize() > 0:
            node = self.getNode(0)
            self.__swap(0, self.getHeapSize() - 1)
            self.H.pop(self.getHeapSize() - 1)
            self.__heapify(0)
            return node
        else:
            return None

    def __str__(self):
        str = "[ "
        for x in self.H:
            str += "({}, {}) ".format(x[0], x[1])
        str += "]"
        return str
```

Benchmark Appendix

```python
from HeapSelect import heapSelect
from QuickSelect import quickSelect
from partition3WayRandomized import quickSelectRandomized
from MedianOfMediansSelect import medianOfMediansSelect
from DisegnaGrafico import disegnaGrafico

import random
import time
import os
################################## ! DISCLAIMER ! ####################################################################

# THE FOLLOWING BENCHMARK IS NOT DESIGNED TO RUN ON WINDOWS.
# IT HAS BEEN TESTED ON MACOSX AND UBUNTU, SO WE DO NOT RECOMMEND EXECUTION ON DIFFERENT OPERATING SYSTEMS.
# IN PARTICULAR, THE BENCHMARK EXECUTION TIMES ON WINDOWS MAY BE EXTREMELY HIGHER.
# ALSO, THE GRAPH WILL NOT BE GENERATED.



'''
minimumMeasurableTime:
    Returns the minimum measurable time by the time.monotonic() counter.
'''
def minimumMeasurableTime():
    startMeasurement = time.monotonic()
    while time.monotonic() == startMeasurement:
        pass
    endMeasurement = time.monotonic()
    return endMeasurement - startMeasurement


'''
measureTimes:
    1) Accurately measures the execution time of each algorithm in the algoritmiDiSelezione list.
    2) Ensures a relative error less than 1%.
    3) Adds the average measured time to the average time counter of each executed algorithm.
'''
tMinMeasurable = minimumMeasurableTime() * (1 + (1 / 0.01))  # max_rel_error = 1% = 0.01

def measureTimes(array, k, algoritmiDiSelezione, averageTimes):
    for i, func in enumerate(algoritmiDiSelezione):
        count = 0
        startTime = time.monotonic()
        while time.monotonic() - startTime < tMinMeasurable:
            func(array.copy(), k)
            count += 1
        duration = time.monotonic() - startTime
        averageTimes[i] += (duration / count)


'''
benchmark:
    1) Measures the average execution time of each algorithm in the algoritmiDiSelezione list.
    2) Generates passiSuccessione(100) arrays of increasing size following a geometric series.
        The size of the arrays ranges from 100 to 100000.
    3) At each step, testPerOgniN(500) tests are performed with random k to ensure that the execution times of the algorithms are average.
    4) In each test, the array of predetermined size is filled with random values in a range [-1000,1000].
    5) In each test, the execution time is measured.
    6) At the end of the tests, the average times for that step of the series are saved in a list of lists averageTimes.
    7) At each step, after performing testPerOgniN(500), the size of the array used for that step is saved.
    8) At the end of the 100 steps, the average time data is saved in separate files. One for each algorithm.
    9) Finally, the function that draws the graph using the data from the generated files is called.
'''
# List of selection algorithms
algoritmiDiSelezione = [quickSelect, quickSelectRandomized, heapSelect, medianOfMediansSelect]
nomi_algoritmiDiSelezione = ["QuickSelect", "QuickSelectRandomizedPTW", "HeapSelect", "MedianOfMediansSelect"]


# Initialize the lists for average times
averageTimes = [[] for _ in algoritmiDiSelezione]     # list of as many lists as there are selection algorithms
generatedArraySizes = []

def benchmark():
    A = 100  # Start of the geometric series
    B = (100000 / 100) ** (1 / 99)  # Calculate B to get final n of 100000 (99th root)
    passiSuccessione = 100
    testPerOgniN = 1


    for i in range(passiSuccessione):
        arraySize = int(A * (B ** i))
        print("Executing step {} / {} of the series \t len(A) : {}".format(i+1, passiSuccessione, arraySize))

        # Initialize average times for this array size
        stepAverageTimes = [0] * len(algoritmiDiSelezione)

        for _ in range(testPerOgniN):
            array = [random.randint(-1000, 1000) for _ in range(arraySize)]
            k = random.randint(1, arraySize)
            measureTimes(array, k, algoritmiDiSelezione, stepAverageTimes)

        for j in range(len(algoritmiDiSelezione)):
            averageTimes[j].append(stepAverageTimes[j] / testPerOgniN)

        generatedArraySizes.append(arraySize)

    # Set the working directory relative to the location of this file
    directoryPath = os.path.join(os.path.dirname(__file__), 'RisultatiBenchmark')

    # Check if the folder exists, otherwise create it
    if not os.path.exists(directoryPath):
        os.makedirs(directoryPath)

    # Build the file paths relative to this directory
    arraySizesPath = os.path.join(directoryPath, 'dimensioniArray.txt')

    # Write the size of each generated array to a file.
    with open(arraySizesPath, 'w') as f:
        for n in generatedArraySizes:
            f.write(f"{n}\n")
```

```
    # Write the execution time results to separate text files. One for each algorithm.
    for i, name in enumerate(nomi_algoritmiDiSelezione):
        filePath = os.path.join(directoryPath, f'tempi{name}.txt')
        with open(filePath, 'w') as f:
            for time in averageTimes[i]:
                f.write(f"{time}\n")

print("Benchmark started")
benchmark()
print("Benchmark completed. Creating the graph.")
disegnaGrafico()
```

DrawGraph Appendix

```
import matplotlib.pyplot as plt
import os

'''
readFromFile:
    Reads from a file and saves the read values in an array.
'''
def readFromFile(filePath):
    values = []
    with open(filePath, 'r') as file:
        for line in file:
            value = float(line.strip())
            values.append(value)
    return values


'''
drawGraph:
    1) Creates a logarithmic scale graph of the performance of quickSelect, heapSelect, medianOfMediansSelect.
    2) On the x-axis are the sizes of the arrays on which the execution time measurements were made.
    3) On the y-axis are the execution times of the indicated selection algorithms.
    4) The data used to plot the graph must be saved in the same path where this program is located.
    5) The generated graph consists of lines and points. The plot and scatter functions are used, respectively.
    6) The generated graph is NOT saved automatically.
'''

def drawGraph():
    # Set the working directory relative to the position of this file
    directoryPath = os.path.join(os.path.dirname(__file__), 'BenchmarkResults')

    fileSizePath = os.path.join(directoryPath, 'arraySizes.txt')
    size = readFromFile(fileSizePath)

    # Quick Random
    fileExecutionTimeQuickRandom = os.path.join(directoryPath, 'executionTimesQuickSelectRandomizedPTW.txt')
    executionTimeQuickSelectRandom = readFromFile(fileExecutionTimeQuickRandom)

    fileExecutionTimeQuick = os.path.join(directoryPath, 'executionTimesQuickSelect.txt')
    executionTimeQuickSelect = readFromFile(fileExecutionTimeQuick)

    # Heap
    fileExecutionTimeHeapSelect = os.path.join(directoryPath, 'executionTimesHeapSelect.txt')
    executionTimeHeapSelect = readFromFile(fileExecutionTimeHeapSelect)

    # Median
    fileExecutionTimeMedianOfMedians = os.path.join(directoryPath, 'executionTimesMedianOfMediansSelect.txt')
    executionTimeMedianOfMedians = readFromFile(fileExecutionTimeMedianOfMedians)

    # Subplot is used to combine the two created graphs (points and lines)
    fig, ax = plt.subplots(figsize=(10, 6)) # The graph will be 10x6 inches

    # Title of the graph
    plt.title('QuickSelect Performance in the Average Case', fontsize = 15, fontweight = 'bold')
    ax.grid(True, linestyle='--', alpha=0.7)
    ax.tick_params(labelsize=12)

    # Draw the points. Priority is 1 so the points are drawn above the lines. 's' is the point size
    plt.scatter(size , executionTimeQuickSelect, color='orange', label = 'QuickSelect', zorder=2, s=20) # Quick
    plt.scatter(size , executionTimeQuickSelectRandom, color='red', label = 'QuickSelect3WayRandomized', zorder=2, s=20) # QuickRandom
    plt.scatter(size, executionTimeHeapSelect,  color='paleturquoise', label = 'HeapSelect', zorder=2, s=20) # Heap
    plt.scatter(size, executionTimeMedianOfMedians, color='mediumpurple', label = 'MedianOfMediansSelect', zorder=2, s=20) # Median

    # Draw the lines. Priority is 2 so the lines are drawn below the points.
    ax.plot(size, executionTimeQuickSelect, '-', color='darkorange', zorder=1) # Quick
    ax.plot(size, executionTimeQuickSelectRandom, '-', color='red', zorder=1) # QuickRandom
    ax.plot(size, executionTimeHeapSelect, '-', color='darkturquoise', zorder=1) # Heap
    ax.plot(size, executionTimeMedianOfMedians, '-', color='blueviolet', zorder=1) # Median

    # Draw the linear reference function for QuickSelect
    k = executionTimeQuickSelect[0] / size[0]
    linearTime = [(k * x) for x in size]
    ax.plot(size, linearTime, '--', color='darkorange', label = 'Linear Trend', zorder=3)

    # Draw the linear reference function for QuickSelectRandomized
    k = executionTimeQuickSelectRandom[0] / size[0]
    linearTime = [(k * x) for x in size]
    ax.plot(size, linearTime, '--', color='red', label = 'Linear Trend', zorder=3)


    # Draw the linear reference function for HeapSelect
    k = executionTimeHeapSelect[0] / size[0]
    linearTime = [(k * x) for x in size]
    ax.plot(size, linearTime, '--', color='darkturquoise', label = 'Linear Trend', zorder=3)

    # Draw the linear reference function for Medians
    k = executionTimeMedianOfMedians[0] / size[0]
    linearTime = [(k * x) for x in size]
    ax.plot(size, linearTime, '--', color='blueviolet', label = 'Linear Trend', zorder=3)


    # Set the logarithmic scale on both axes
    plt.xscale('log')
    plt.yscale('log')

    # Set the legend
    ax.legend(loc='upper left', fontsize=10)
```

```
# Set the labels on the x and y axes
ax.set_xlabel('Array Size', fontsize=11, fontweight = 'bold')
ax.set_ylabel('Average Execution Time (s)', fontsize=11, fontweight = 'bold')

# Show the graph
plt.show()
```