**MARMARA UNIVERSITY**

**FACULTY OF ENGINEERING**

CSE



**CSE 2046**

**Analysis of Algorithms**

**HOMEWORK 2**

NAME STUDENT NUMBER

Yasin Enes POLAT 150117015

Fatih AKGÜNDÜZ 150117032

In all parts, we use 2 different input list. Firstly, we choose sizes as 100, 200, 300 till 1000 and at each size we sent 100 different input and illustrate the ***Graph 1***. Secondly, we choose sizes randomly between 100 and 1000. We sent 1000 different sizes and illustrate the ***Graph 2***.

**\*\*\*We use 3 different input types which has;**

* **100, 200, …, 1000 size and 100 of each,**
* **fully random size, between 10-1000 and 1 of each,**
* **almost sorted numbers and size of 100, 110, 120, …, 1000.**

**\*\*\*We use MATLAB to plot dot graphs. The x-axis represents size and y-axis represents count.**

**Task Distrubition:**

We both;

* Coded
* Generated inputs

Fatih, checked algorithms from book, plotted graphs with using outputs of input files.

Enes generated outputs, illustrated tables.

**1-) Insertion Sort**

Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

**Theoritical Time Complexity ->** **O(n2)**

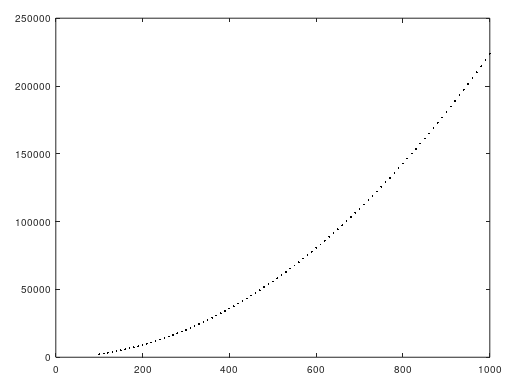
**Following results are from our code;**

**input size | total number of basic operations**

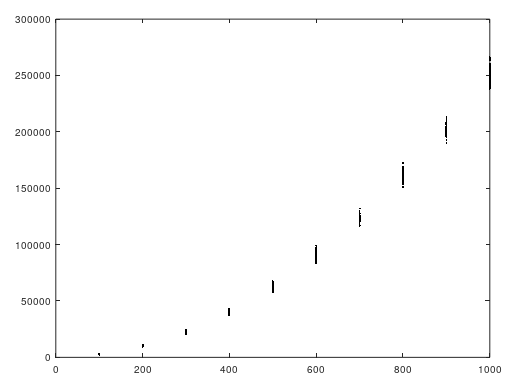
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **137 -> 4948** | **268 -> 18271** | **444 -> 48403** | **640 -> 101338** | **818 -> 173190** |

|  |  |  |
| --- | --- | --- |
| **Ratio** | **Expected** | **Actual** |
| 268 / 137 = 1.96 | 1.962 = 3.84 | 18271 / 4948 = 3.69 |
| 444 / 137 = 3.24 | 3.242 = 10.5 | 48403 / 4948 = 9.78 |
| 640 / 137 = 4.67 | 4.672 = 21.81 | 101338 / 4948 = 20.48 |
| 818 / 137 = 5.97 | 5.972 = 35.65 | 173190 / 4948 = 35.00 |

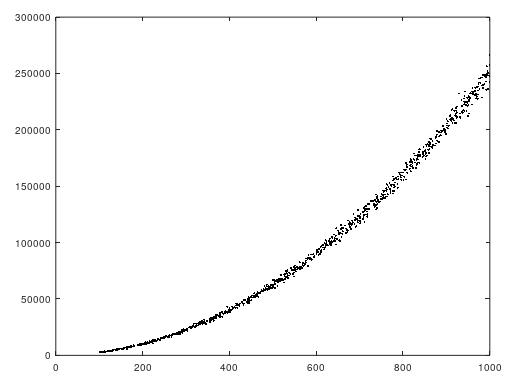
*Graph of Almost Sorted Arrays’ Results*



*Graph 1*

****

*Graph 2*

****

**2-) Merge Sort**

*This algorithm is very consistent for almost all different number of inputs, there is just 1 outlier in our results, flexibility of time complexity is very limited, almost none.*

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

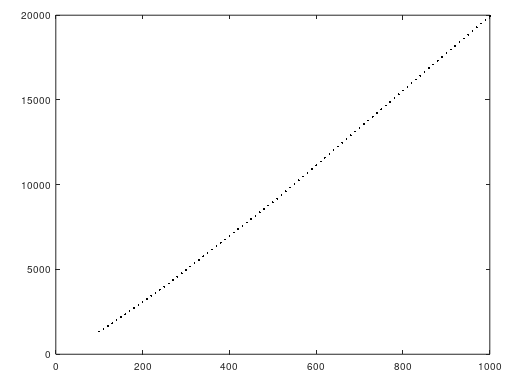
**Theoritical Time Complexity ->** **O(nlogn)**

**Following results are from our code;**

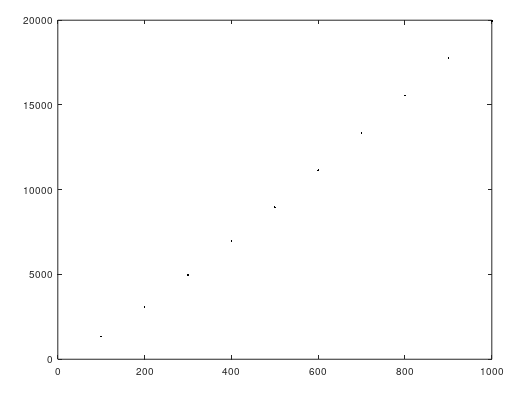
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **137 -> 1954** | **268 -> 4336** | **444 -> 7856** | **640 -> 12032** | **818 -> 15948** |

|  |  |  |
| --- | --- | --- |
| **Ratio** | **Expected** | **Actual** |
| 268 / 137 = 1.96 | 1.96\*log1.96 = 1.32 | 4336 / 1954 = 2.22 |
| 444 / 137 = 3.24 | 3.24\*log3.24 = 3.81 | 7856 / 1954 = 4.02 |
| 640 / 137 = 4.67 | 4.67\*log4.67 = 7.20 | 12032 / 1954 = 6.15 |
| 818 / 137 = 5.97 | 5.97\*log5.97 = 10.67 | 15948 / 1954 = 8.16 |

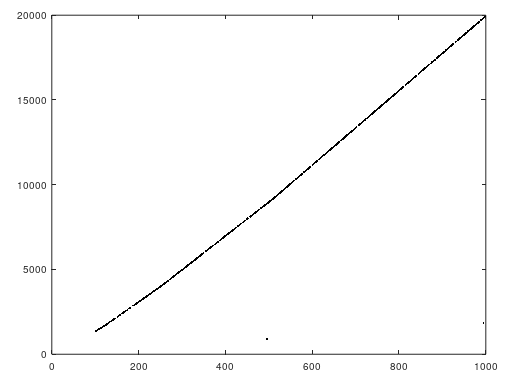
*Graph of Almost Sorted Arrays’ Results*



*Graph 1*



*Graph 2*



**3-) Max Heap**

*This algorithm is also very consistent and has very limited flexibility on time complexity. It takes O(1) to get max element from heap and it reconstructs the heap again to put max element on root.*

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

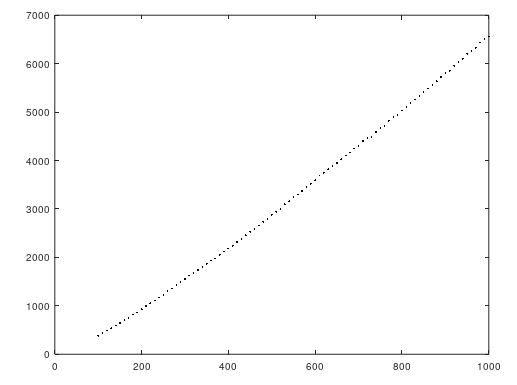
**Theoritical Time Complexity ->** **O(nlogn)**

**Following results are from our code;**

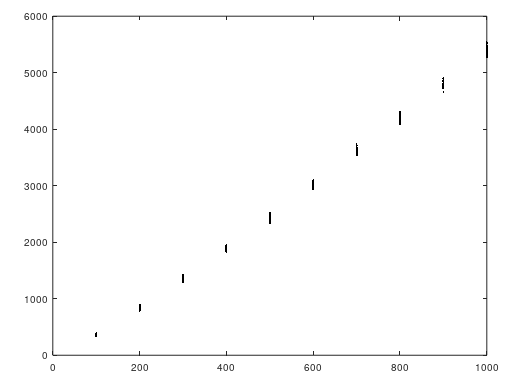
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **137 -> 521** | **268 -> 1156** | **444 -> 2141** | **640 -> 3283** | **818 -> 4272** |

|  |  |  |
| --- | --- | --- |
| **Ratio** | **Expected** | **Actual** |
| 268 / 137 = 1.96 | 1.96\*log1.96 = 1.32 | 1156 / 521 = 2.22 |
| 444 / 137 = 3.24 | 3.24\*log3.24 = 3.81 | 2141 / 521 = 4.10 |
| 640 / 137 = 4.67 | 4.67\*log4.67 = 7.20 | 3283 / 521 = 6.30 |
| 818 / 137 = 5.97 | 5.97\*log5.97 = 10.67 | 4272 / 521 = 8.20 |

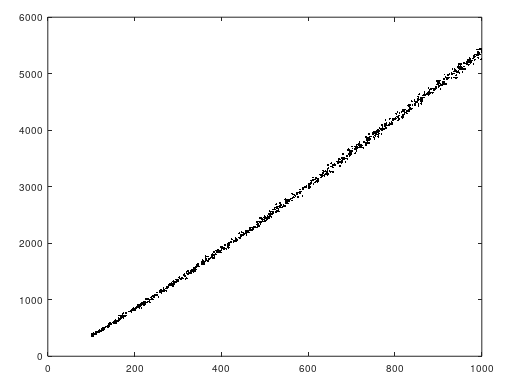
*Graph of Almost Sorted Arrays’ Results*



*Graph 1*

****

*Graph 2*

****

**4-) Quick Select**

*We can use quick select algorithms with small number of inputs because it is very inconsistent with large number of inputs as you see on graphs. Probability of worst case scenario O(n2) is much higher than best case.*

**a. First elements as pivot**

*We can use quick select algorithms with small number of inputs because it is very inconsistent with large number of inputs as you see on graphs.*

Quickselect is a selection algorithm to find the k-th smallest element in an unordered list. If index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part.

On each iteration we select the first element of remaining array as pivot.

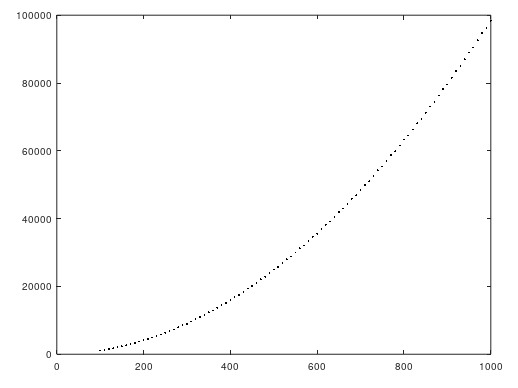
**Theoritical Time Complexity ->** **O(n)**

**Following results are from our code;**

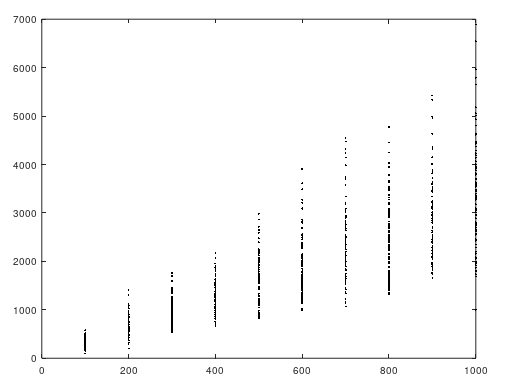
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **168 -> 739** | **255 -> 726** | **488 -> 2501** | **690 -> 2431** | **936 -> 3347** |

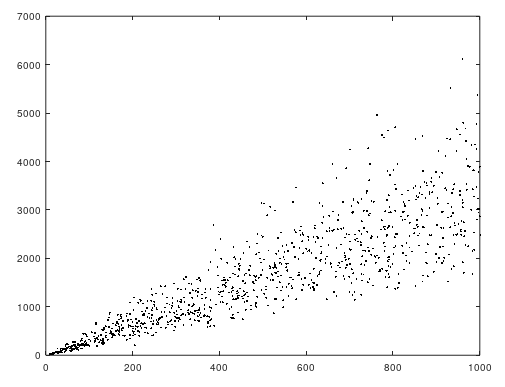
|  |  |  |
| --- | --- | --- |
| **Ratio** | **Expected** | **Actual** |
| 255 / 168 = 1.52 | 1.52 | 726 / 739 = 0.98 |
| 488 / 168 = 2.90 | 2.90 | 2501 / 739 = 3.38 |
| 690 / 168 = 4.11 | 4.11 | 2431 / 739 = 3.29 |
| 936 / 168 = 5.57 | 5.57 | 3347 / 739 = 4.53 |

*Graph of Almost Sorted Arrays’ Results*



*Graph 1*



*Graph 2*

**b. Median-of-three pivot selection**

Quickselect is a selection algorithm to find the k-th smallest element in an unordered list. If index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part.

On each iteration we select the median of first, last and middle element of remaining array as pivot.

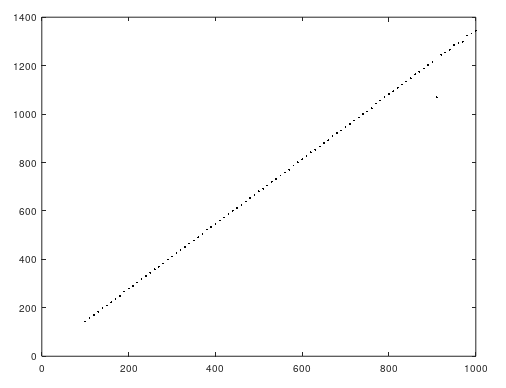
**Theoritical Time Complexity ->** **O(n)**

**Following results are from our code;**

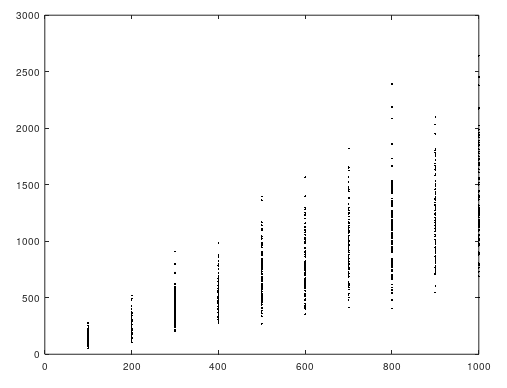
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **168 -> 316** | **255 -> 278** | **488 -> 514** | **690 -> 462** | **936 -> 939** |

|  |  |  |
| --- | --- | --- |
| **Ratio** | **Expected** | **Actual** |
| 168 / 255 = 0.66 | 0.66 | 316 / 278 = 1.14 |
| 488 / 255 = 1.91 | 1.91 | 514 / 278 = 1.85 |
| 690 / 255 = 2.71 | 2.71 | 462 / 278 = 1.66 |
| 936 / 255 = 3.67 | 3.67 | 939 / 278 = 3.38 |

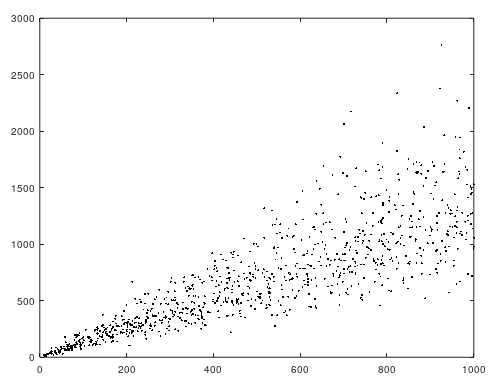
*Graph of Almost Sorted Arrays’ Results*



*Graph 1*



*Graph 2*



**c. Median-of-medians pivot selection**

Quickselect is a selection algorithm to find the k-th smallest element in an unordered list. If index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part.

Divides arr[] in groups of size 5,  calculates median of every group, and stores it in median[] array.

**Theoritical Time Complexity ->** **O(n)**

**Following results are from our code;**

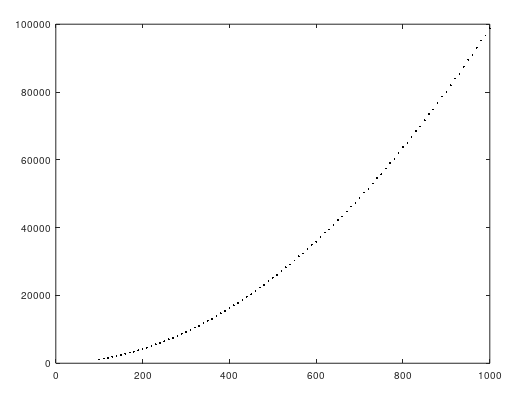
This algorithm finds [n/2]th smallest element on given array.

**Theoritical Time Complexity ->** **O(n)**

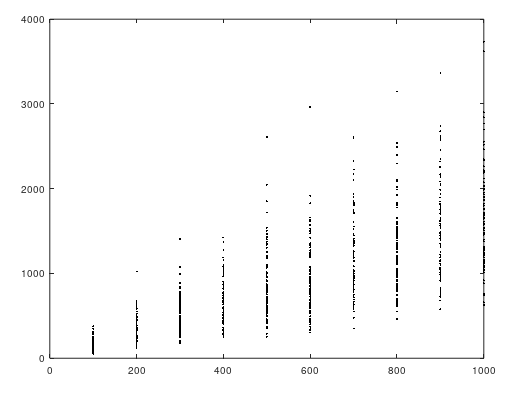
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **137 -> 127** | **268 -> 720** | **444 -> 675** | **640 -> 1103** | **818 -> 1232** |

|  |  |  |
| --- | --- | --- |
| **Ratio** | **Expected** | **Actual** |
| 268 / 137 = 1.96 | 1.96 | 720 / 127 = 5.67 |
| 444 / 137 = 3.24 | 3.24 | 675 / 127 = 5.31 |
| 640 / 137 = 4.67 | 4.67 | 1103 / 127 = 8.68 |
| 818 / 137 = 5.97 | 5.97 | 1232 / 127 = 9.70 |

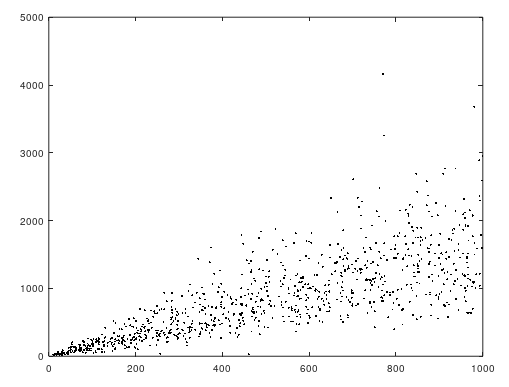
*Graph of Almost Sorted Arrays’ Results*



*Graph 1*

****

*Graph 2*

****