T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE2046 - Analysis of Algorithms

Homework I - Report

Group Members

150116005 - Veysi Öz

150119802 - Nurefşan Yücel

150115061 - Abbas Göktuğ Yılmaz

Insertion Sort Algorithm

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part [1].

To sort an array of size n in ascending order:

- Iterate from arr[1] to arr[n] over the array.
- Compare the current element (key) to its predecessor.
- If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Pseudocode

```
Algorithm insertion_sort(A[0...n-1])
        for i = 1 to n-1
            key ← A[i]
             j←i-1
              while j >= 0 and A[j] > key
                        A[j+1] ← A[j]
                        j←j–1
              end while
              A[j+1] ← key
end for
```

Time Complexity of Insertion Sort

The Best Case

The best case of Insertion Sort Algorithm is that the given input sequence is already sorted. It gives the linear running time complexity O(n). During each iteration, only one comparison is done.

$$c_{op} = 1$$

$$C_{best}(n) = \sum_{i=1}^{n-1} = n - 1$$

$$T(n) = c_{op} \times C_{best}(n) = 1 \times (n-1)$$

$$T(n) = n - 1 \; \varepsilon \; O(n)$$

The Worst Case

The worst case of Insertion Sort Algorithm is that the given input array is sorted by descending order. The set of all worst-case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases, every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(n2)).

$$c_{op} = 1$$

$$C_{best}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} = \frac{n^2+n}{2}$$

$$T(n) = c_{op} \times C_{best}(n) = 1 \times \frac{n^2+n}{2}$$

$$T(n) = \frac{n^2+n}{2} \, \varepsilon \, O(n^2)$$

Merge Sort Algorithm

Merge sort is a method by which you break an unsorted array down into two smaller halves and so forth until you have a bunch of arrays of only two items. You sort each one, then merge each with another two-item array, then you merge the four-item arrays and so on all the way back up to a fully sorted array.[2]

Pseudocode

```
Algorithm mergesort(A[0..n-1])
// A is an array that will be sorted
// B is an array that is left-hand part of array A
// C is an array that is right-hand part of array A
      If n>1
            B ← A[0..n/2-1]
            C ← A[n/2..n-1]
            mergesort(B)
            mergesort(C)
            merge(B, C, A)
Algorithm merge(left[0..p-1], right[0..q-1], A[0..p+q-1])
// Merges two sorted array into one sorted array
      i ← 0
      j ← 0
      k ← 0
      while i < p and j < q
            If left[i] ≤ right[j]
                  A[k] ← left[i]
                  i ← i + 1
            Else
                  A[k] ← right[j]
                  j ← j + 1
            k ← k + 1
      if i = p
            A[k..p+q-1] ← C[j..q-1]
      else
            A[k..p+q-1] ← B[i..p-1]
```

Time Complexity

Best Case

Merge sort's best case is when the largest element of one sorted sub-list is smaller than the first element of its opposing sub-list, for every merge step that occurs. Only one element from the opposing list is compared, which reduces the number of comparisons in each merge step to N/2.

$$T(n) = 2T(n/2) + n/2$$
$$T(n) = 2[2T(n/4) + n/4] + n/2 = 4T(n/4) + n/2 + n/2$$
$$T(n) = 4[2(n/8) + n/8] + n/4 + n/2 = 8T(n/8) + n/2 + n/2 + n/2$$

$$Let\ n = 2^k and\ k = log_2 n$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^{k-1}$$

A single element array is already sorted.

$$T(1) = 0 \Rightarrow 2^{k-i} = 1 \Rightarrow k = i$$
$$T(2^k) = 2^k T(1) + k2^{k-1}$$
$$T(2^k) = 0 + k2^{k-1}$$
$$T(n) = (log_2 n)n/2 \in O(nlogn)$$

Worst Case

The worst case scenario for Merge Sort is when, during every merge step, exactly one value remains in the opposing list; in other words, no comparisons were skipped. This situation occurs when the two largest values in a merge step are contained in opposing lists. When this situation occurs, Merge Sort must continue comparing list elements in each of the opposing lists until the two largest values are compared.

$$T(n) = 2T(n/2) + n - 1$$
$$T(n) = 4T(n/4) + n - 2 + n - 1$$
$$T(n) = 8T(n/8) + n - 4 + n - 2 + n - 1$$

$$Let\ n = 2^k and\ k = log_2 n$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^k - \sum_{t=0}^{i-1} 2^t$$
$$T(2^k) = 2^i T(2^{k-i}) + i2^k - 2^i + 1$$

A single element array is already sorted.

$$T(1) = 0 \Rightarrow 2^{k-i} = 1 \Rightarrow k = i$$
$$T(2^k) = 2^k T(1) + k2^k + 2^k + 1$$
$$T(2^k) = 0 + k2^k + 2^k + 1$$
$$T(n) = log_2 n\,(n+1) + 1 \in O(nlogn)$$

Quick Sort Algorithm (Pivot is the first element)

Quicksort sorting technique is widely used in software applications. Quicksort uses a divide-and-conquer strategy like merge sort.In the quicksort algorithm, a special element called “pivot” is first selected and the array or list in question is partitioned into two subsets. The partitioned subsets may or may not be equal in size.The partitions are such that all the elements less than the pivot element are towards the left of the pivot and the elements greater than the pivot is at the right of the pivot. The Quicksort routine recursively sorts the two sub-lists. Quicksort works efficiently and also faster even for larger arrays or lists.[3]

Pseudocode

```
procedure quickSort(arr[], low, high)
   arr = list to be sorted
   low – first element of the array
   high – last element of array
begin
   if (low < high)
{
     // pivot – pivot element around which array will be
partitioned
       pivot = partition(arr, low, high);
       quickSort(arr, low, pivot - 1);  // call quicksort recursively to sort sub array
before pivot
        quickSort(arr, pivot + 1, high); // call quicksort recursively to sort sub array
after pivot
}
end procedure
//partition routine selects and places the pivot element into its proper position that
will partition the array.
//Here, the pivot selected is the last element of the array procedure partition (arr[],
low, high)
begin
  // pivot (Element to be placed at right position)
  pivot = arr[high];
   i = (low - 1)  // Index of smaller element
   for j = low to high
   {
        if (arr[j] <= pivot)
        {
             i++;    // increment index of smaller element
```

```
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
end procedure
```

Time Complexity

The worst case time complexity of a typical implementation of QuickSort is O(n2). The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when the input array is sorted or reverse-sorted and either the first or last element is picked as pivot.

Partial Selection Sort Algorithm

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array. The concept used in Selection Sort helps us to partially sort the array up to kth-smallest (or largest) element for finding the kth-smallest (or-largest) element in an array [4].

Pseudocode

```
function partialSelectionSort(arr[0..n], k) {
    for i in [0, k) {
        minIndex = i
        minValue = arr[i]
        for j in [i+1, n) {
            if (arr[j] < minValue)  then
                minIndex = j
                minValue = arr[j]
                swap(arr[i], arr[minIndex])
        }
    }
    return arr[k]
}
```

Time Complexity

partial selection sort yields a simple selection algorithm that takes O(k*n) time to sort the array. This is asymptotically inefficient but can be sufficiently efficient if k is small

Partial Heap Sort Algorithm

Heaps admit a simple single-pass partial sort when *k* is fixed: insert the first *k* elements of the input into a max-heap. Then make one pass over the remaining elements, add each to the heap in turn, and remove the largest element [5].

Pseudocode

```
def partial_heap_sort(array, k):
    n = len(array)
    for i in range(n//2 - 1, -1, -1):
        heapify(array, n, i)
    for i in range(n-1, k-1, -1):
        array[i], array[0] = array[0], array[i]
        heapify(array, i, 0)
    return array[0]
```

Time Complexity

Each insertion operation takes O (log k) time, resulting in O (n log k) time overall; this algorithm is practical for small values of k. Another option is to build a min-heap for all the values (the build takes O (n)) and take out the head of the heap K times; each remove operation takes O (log n). In that case, the algorithm takes O (n+klog n).

Quick Select Algorithm (Pivot is First Element)

Quickselect is a selection algorithm to find the k-th smallest element in an ordered list. The algorithm is similar to QuickSort. The difference is, instead of recurring for both sides (after finding pivot, which in our case is the first element), it recurs only for the part that contains the k-th smallest element. The logic is simple, if index of partitioned element is more than k, then we recur for left part. If index is the same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for the right part [6].

Pseudocode

```
quick_select(array, l, r, k, pivot):
    if (k > 0 and k <= r - l + 1):
        # Pivot is first or median-of-three according to pivot
 parameter
        if pivot == 0:
            index = first_element(array, l, r)
        if pivot == 1:
            index = median_of_three(array, l, r)
```

```
    if (index - l == k - 1):
        return array[index]
    if (index - l > k - 1):
        return quick_select(array, l, index - 1, k, pivot)

    return quick_select(array, index + 1, r, k - index + l
 - 1, pivot)
```

Time Complexity

If good pivots are chosen, meaning ones that consistently decrease the search set by a given fraction, then the search set decreases in size exponentially, and by induction, or summing the geometric series) one sees that performance is linear, as each step is linear and the overall time is a constant times this (depending on how quickly the search set reduces). However, if bad pivots are consistently chosen, such as decreasing by only a single element each time, then worst-case performance is quadratic: O. (n^2) This occurs for example, in searching for the maximum element of a set, using the first element as the pivot, and having sorted data. The probability of the worst-case occurring decreases exponentially with n, so quickselect has an almost certain O (n) time complexity.

Quick Select Algorithm (median-of-three pivot)

The median of three has you look at the first, middle, and last elements of the array, and choose the median of those three elements as the pivot. To get the "full effect" of the median of three, it's also important to sort those three items, not just use the median as the pivot this doesn't affect what's chosen as the pivot in the current iteration, but can/will affect what's used as the pivot in the next recursive call, which helps to limit the bad behavior for a few initial orderings one that turns out to be particularly bad in many cases is an array that's sorted, except for having the smallest element at the high end of the array (or largest element at the low end) [7].

Pseudocode

```
def median_of_three(array, low, high):

    middle = int((high-low)/2)

    median = statistics.median([array[low], array[middle], array[high]])

    if median == array[low]:

        return first_element(array, low, high)

    elif median == array[middle]:

        return middle_element(array, low, high)
```

```
    elif median == array[high]:

        return last_element(array, low, high)
```

Time Complexity

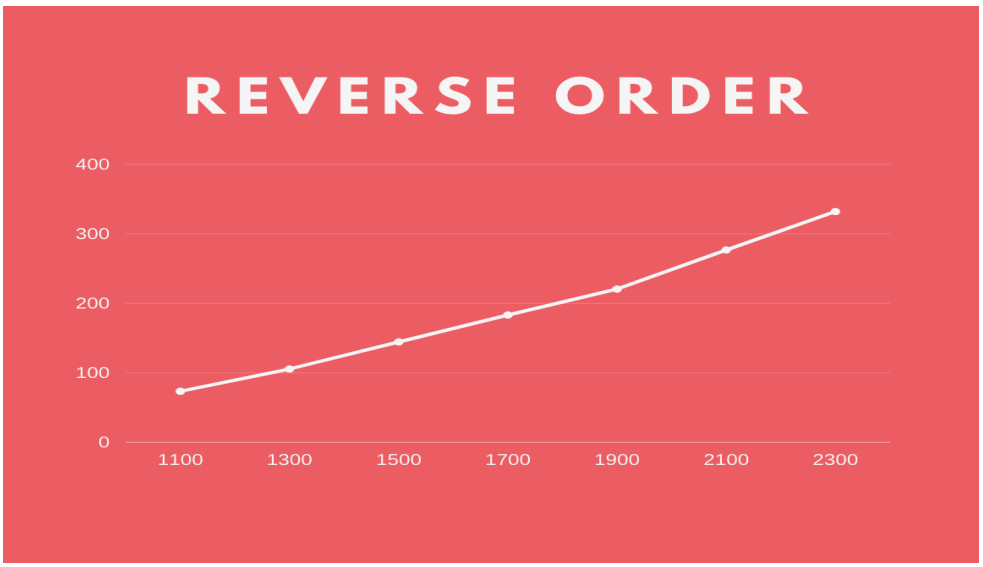
This algorithm runs in O(n) linear time complexity, we traverse the list once to find medians in sublists and another time to find the true median to be used as a pivot.

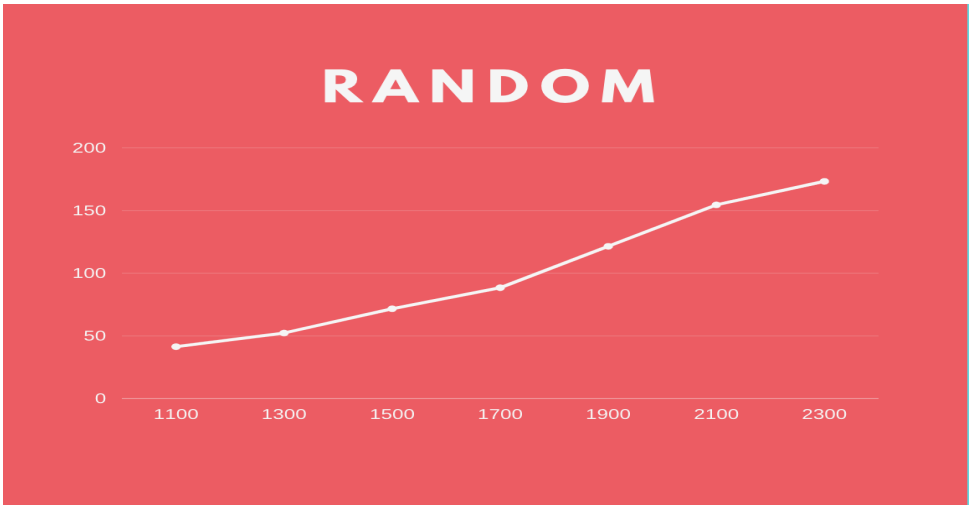
The space complexity is O(logn) , memory used will be proportional to the size of the lists.

Analysis of Outputs

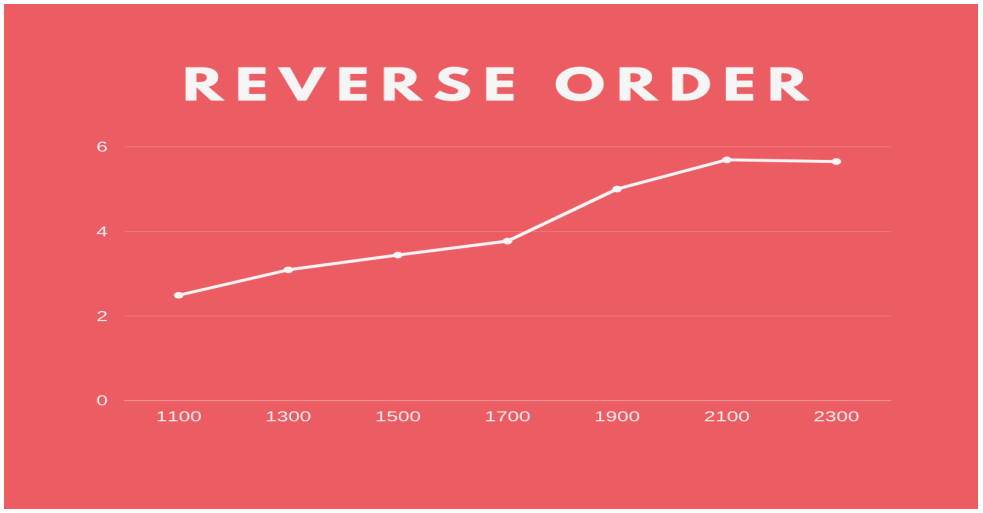
We compared the times to find the kth smallest element by giving each algorithm a list of different sizes of sequential, reverse-ordered, and random-ordered inputs. The x-axis of the graphics shows the various input sizes (1100,1300,1500,1700,1900,2100,2300), while the y-axis shows the time (milliseconds) for which the algorithm produces results.

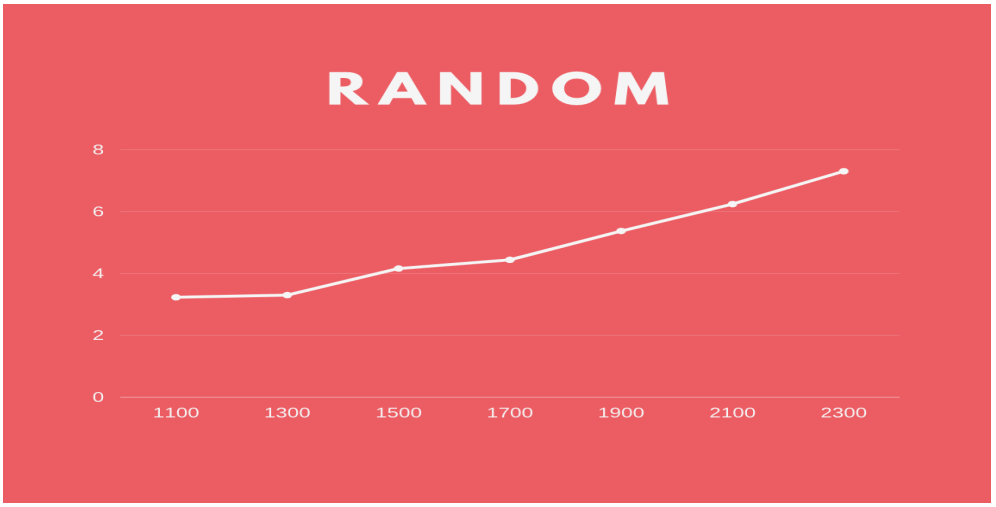
Insertion Sort Algorithm



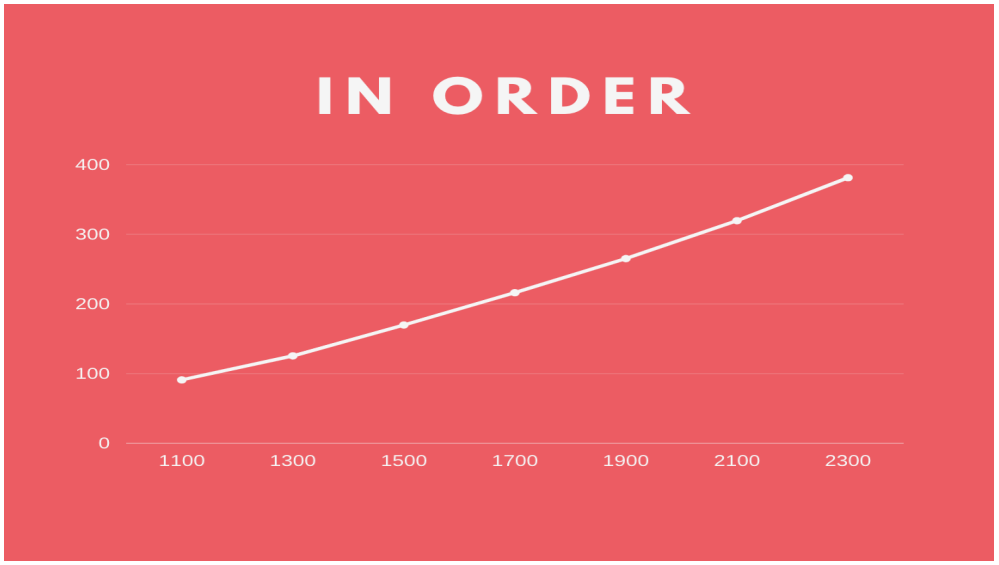


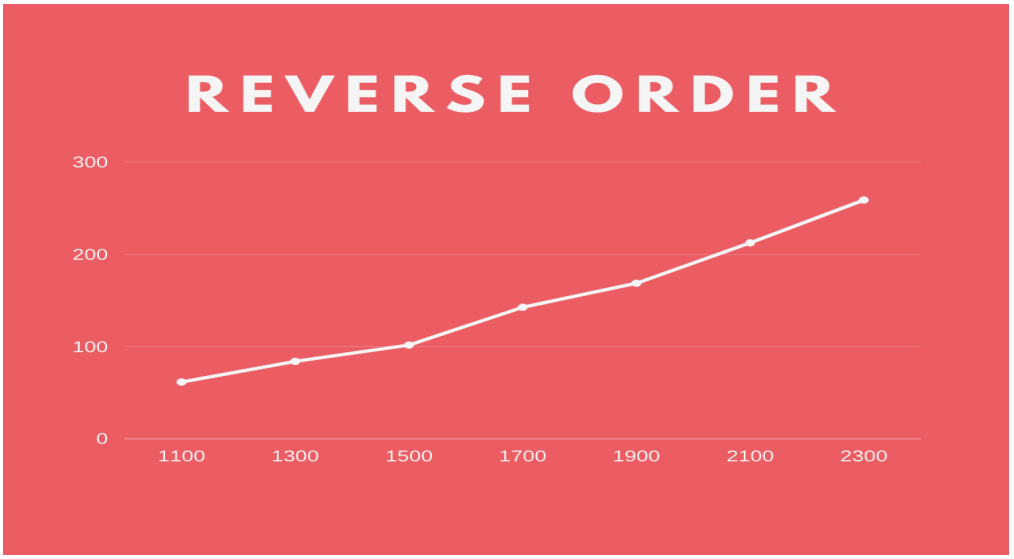
Merge Sort Algorithm

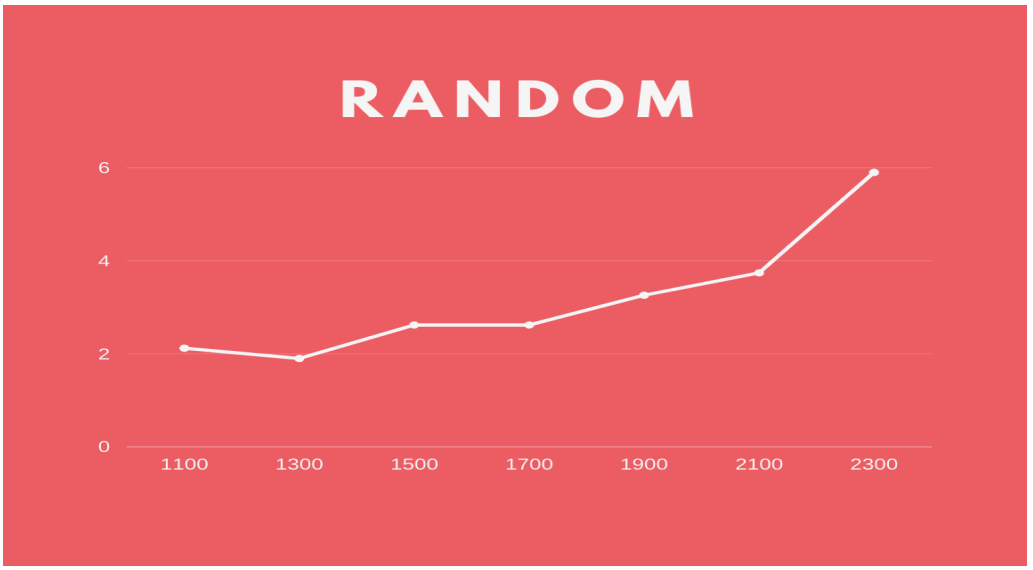




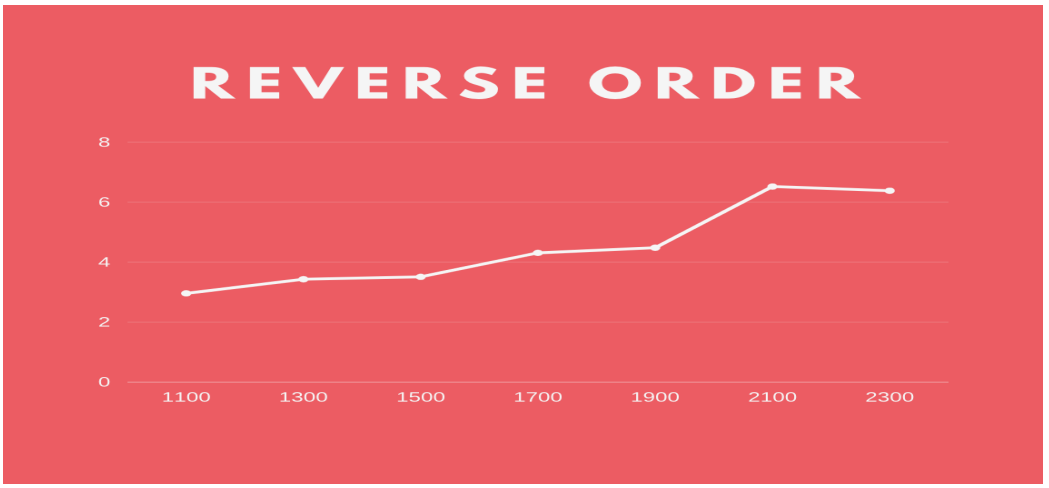
Quick Sort Algorithm (Pivot is the first element)

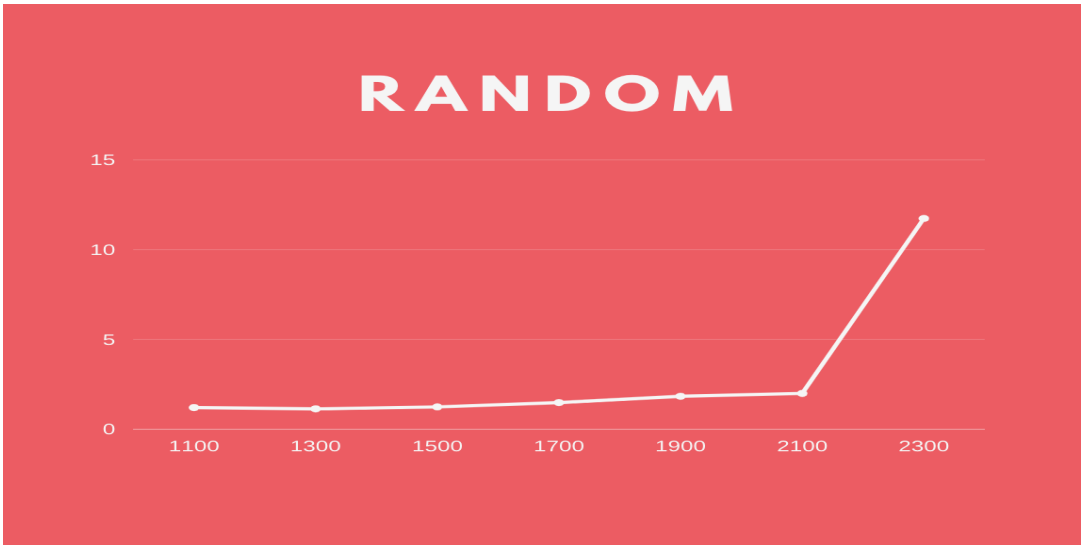



REVERSE ORDER
300
200
100
0
1100
1300
1500
1700
1900
2100
2300

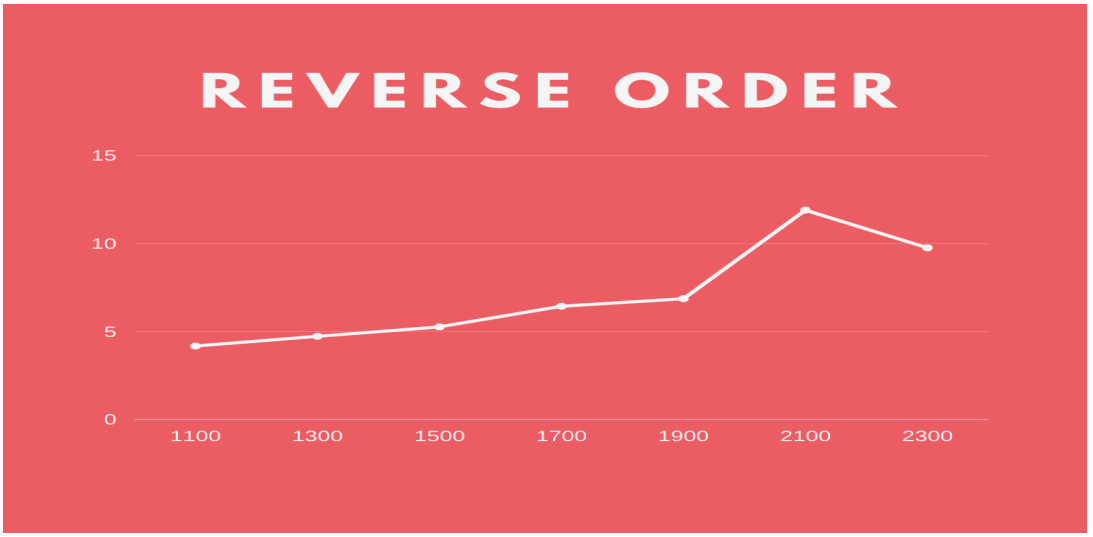

RANDOM
6
4
2
0
1100
1300
1500
1700
1900
2100
2300

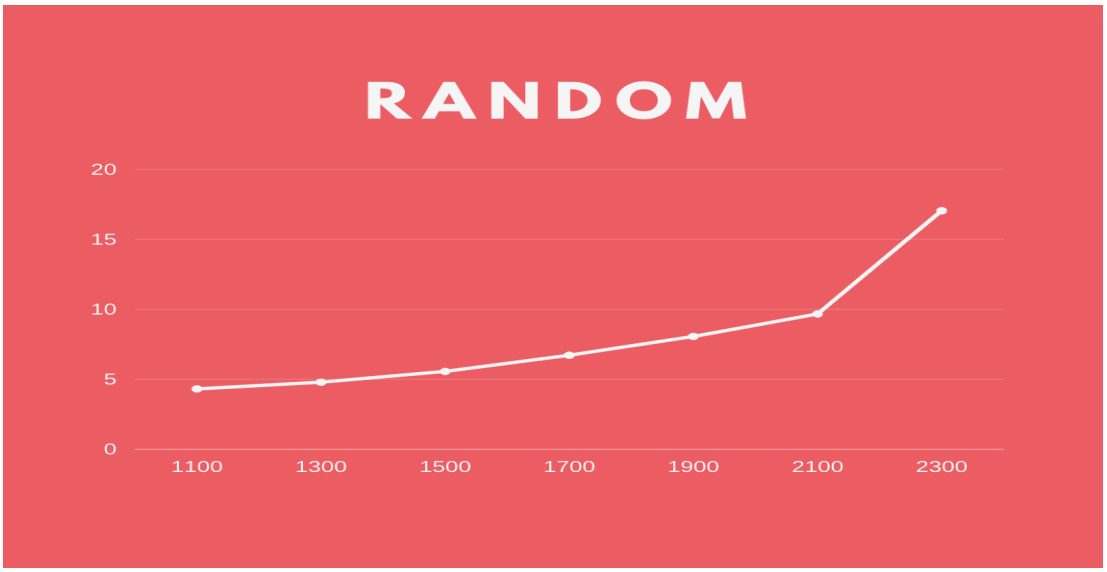
Partial Selection Sort Algorithm



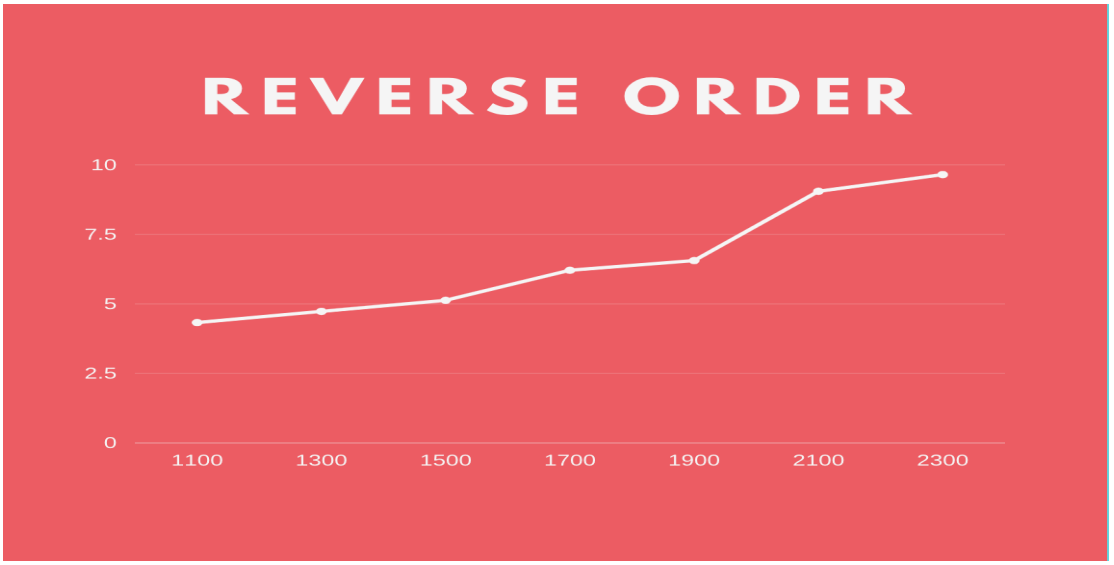


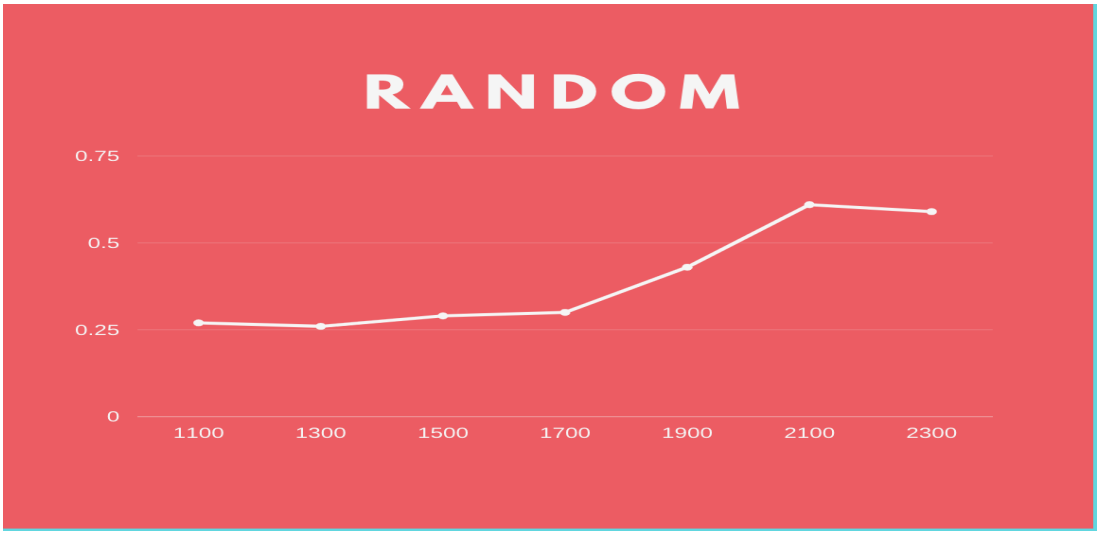
Partial Heap Sort Algorithm



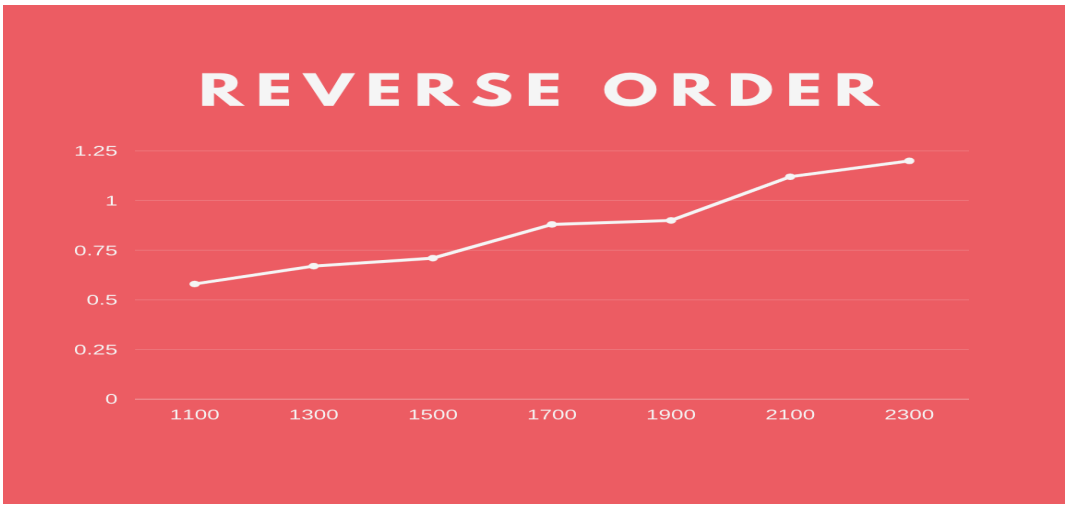


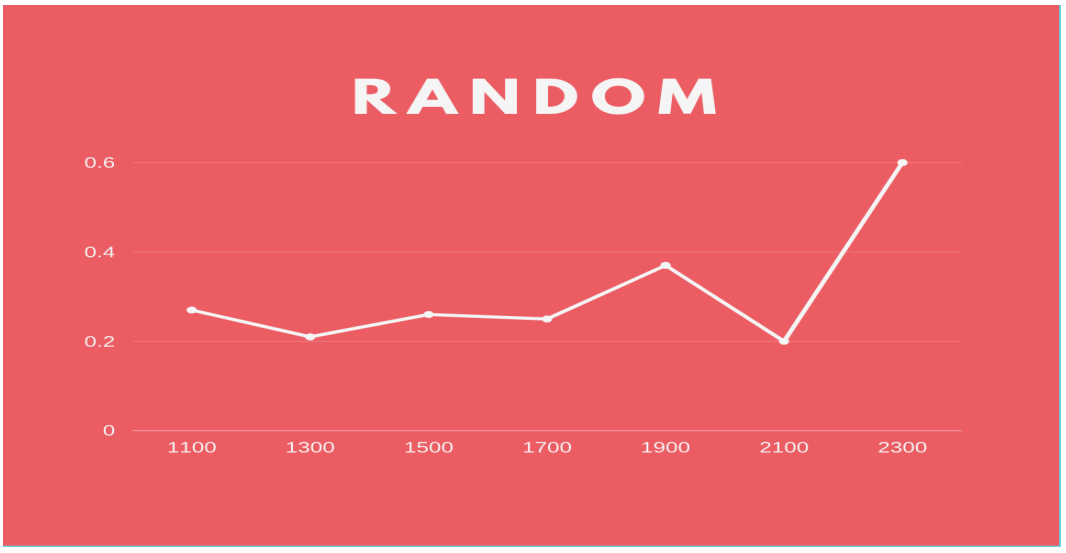
Quick Select Algorithm (Pivot is First Element)



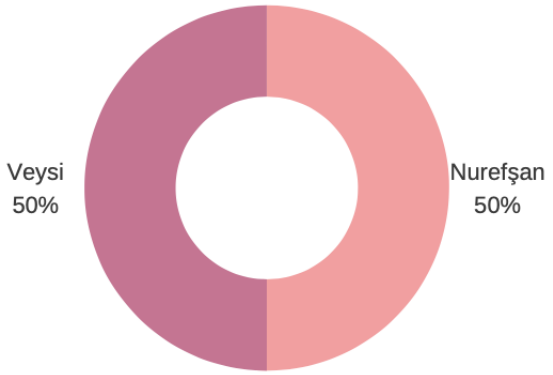


Quick Select Algorithm (median-of-three pivot)

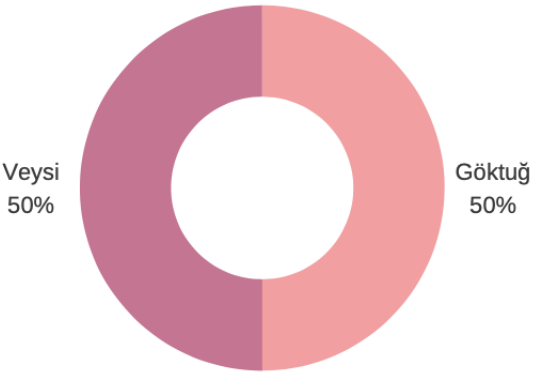




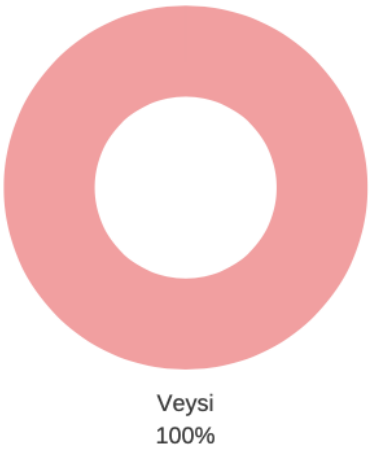
Work Load Charts



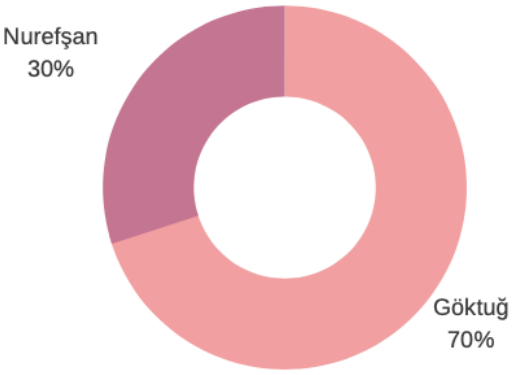
Insertion Sort Algorithm



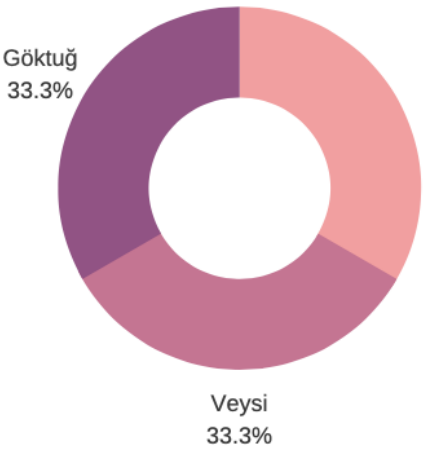
Merge Sort Algorithm



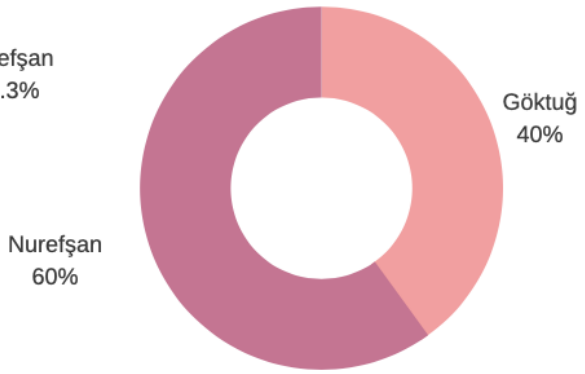
Quick Sort Algorithm (Pivot is the first element)



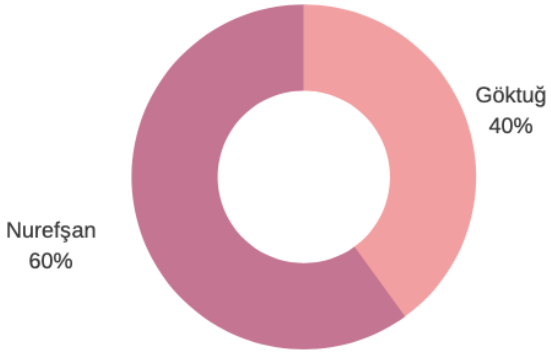
partial selection-sort



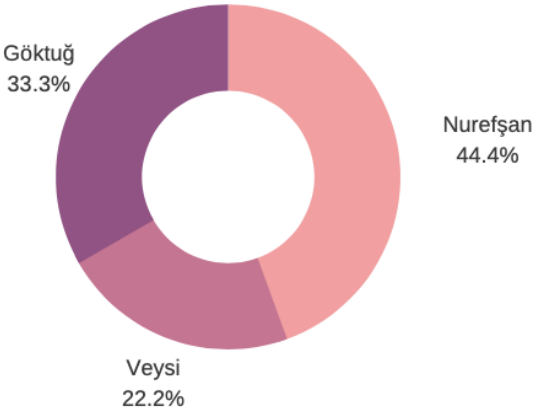
Partial Heap Sort Algorithm



Quick Select Algorithm



Quick Select (Median-of -three)



Repot

REFERENCES

[1] Insertion sort. GeeksforGeeks. (2021, July 8). Retrieved May 10, 2022, from https://www.geeksforgeeks.org/insertion-sort/

[2] Merge sort. GeeksforGeeks. (2022, April 22). Retrieved May 10, 2022, from https://www.geeksforgeeks.org/merge-sort/

[3] Quick sort .SoftwareTestinghelp.(2022, May 4) Retrieved May 10, 2022, from https://www.softwaretestinghelp.com/quicksort-in-java/

[4] Selection algorithms. GeeksforGeeks. (2020, March 18). Retrieved May 10, 2022, from https://www.geeksforgeeks.org/selection-algorithms/

[5] Wikimedia Foundation. (2022, April 7). Partial sorting. Wikipedia. Retrieved May 10, 2022, from https://en.wikipedia.org/wiki/Partial_sorting#Heap-based_solution

[6] Quickselect algorithm. GeeksforGeeks. (2021, August 10). Retrieved May 10, 2022, from https://www.geeksforgeeks.org/quickselect-algorithm/

[7] Quick select algorithm (median of three). Stackoverflow.(2015 , November 6)Retrieved May 10, 2022, from https://stackoverflow.com/questions/7559608/median-of-three-values-strategy