**16/05/2021**

**CSE 2046**

**ANALYSIS OF ALGORITHMS**

**Homework 1**



**Yiğit Göksel 150119053  
 Boran Kanat 150119051**

**PURPOSE**

The aim of this homework is to analyze different types of sorting algorithms. And finding which algorithm works best in which cases.   
After analyzing them one by one then we compared these algorithms with each other to find the optimal sorting technique for various different cases.

**EXECUTION OF THE EXPERIMENT**

**Deciding on reasonable inputs**

\*We tried these algorithms with different sizes of arrays. The sizes were 50,100,200,500,1000.

\*We tried these algorithms for different cases. These cases were:

-**Average case**: Array is filled with random unsorted integers.  
-**Sorted case**: This is a sorted array with ascending order

-**Reverse sorted case**: This is a sorted array with descending order

-**Duplicated case**: This array is filled with all 1’s.

We decided on these arrays because these cover the important cases for sorting techniques.(Best case,worst case,average case)

**Deciding on reasonable metrics**

We compared the execution times by counting the number of basic operations of these sorting techniques. Because there were cases of errors when we tried to measure the execution time in java. We figured the execution times in seconds were too inconsistent.

**Analysis Of QuickSort Algorithm**

Quicksort is another sorting algorithm that uses Divide and Conquer for its implementation. Firstly Quicksort chooses a pivot and divides the elements to be sorted into two sections one with small elements and one with large elements. Elements that are less than or equal to pivot will move towards the left, while the elements that are greater than or equal to pivot will move towards the right.

Then it swaps these elements with each other until the right and left search positions have met or passed each other.

The basic operation of quicksort is comprasion. Algorithm compares elements with pivot and rearranges position of these elements.

When the algorithm divides elements into sections. It determines which elements are small and which are large with the assistance of pivot element. A pivot element can be any element in the array but we will analyse 2 kinds of pivot selection in our algorithms.

**First Element Pivot Algorithm**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| QuickSort |  | 50 | 100 | 200 | 500 | 1000 |
| Average Case | | 252 | 647 | 1566 | 4904 | 12804 |
| Sorted |  | 507 | 1916 | 8518 | 54345 | 121999 |
| Reverse Sorted | | 294 | 1240 | 2108 | 7733 | 25375 |
| Duplicated | | 1225 | 4950 | 19900 | 124750 | 499500 |

The table above us shows that there is how many basic operations there for different input sizes and types.

The worst case complexity of the QuickSort algorithm is but it's a rare occurrence. In average-case complexity of the QuickSort algorithm is **O(n\*logn)** so it makes this algorithm quite good for general purpose.

**Theoretical and Empirical Results**

**Worst Case Time Complexity Of Quick-sort:** The worst case time complexity of the Quick-Sort algorithm happens either pivot is the smallest or biggest element of the array.

   When this occurs the array would not be divided into two equally sized partitions, but one of length 0 and one of length n-1.

Because there would be no element that will be smaller or bigger than the pivot element to rearrange.

**Theroetically:**

**= 1/4 = 0.25**

**Empirical Results For Worst Case :**

**Sorted Array:**  **=** 0.224

**Reverse Sorted Array:** **=** 0.588

**Duplicate Array:**=0.248

We can say the duplicate array is the most representative array for worst-case thanks to our data.

**Average Case Time Complexity Of Quick-sort:**

In average-case complexity of the QuickSort algorithm is **O(n\*logn)**

**Theroetically:**

**=** 0.424

**Empirically:**

**Average Case Array=**  **=** 0.389

**Best Case Time Complexity Of QuickSort**

Quicksort performs best performance when it divide the arrays and subarrays into equal size.

In best-case complexity of the QuickSort algorithm is **O(n\*logn)**

**Median Of Three Pivot Algorithm**

In the median of the three-technique, the median of the first, last, and middle element is chosen as the pivot. This helps in avoiding the worst-case time complexity of **.**

But it doesn't affect the average case and best case. **O(n\*logn)** stays the same.

All remaining process are the same as first element pivot Quicksort.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| MedianQuickSort | | 50 | 100 | 200 | 500 | 1000 |
| Average Case | | 255 | 629 | 1565 | 5654 | 13149 |
| Sorted |  | 1227 | 3754 | 10423 | 31179 | 67897 |
| Reverse Sorted | | 2323 | 7549 | 22572 | 73037 | 163945 |
| Duplicated | | 3548 | 12499 | 42472 | 197787 | 663445 |

The table above us shows that there is how many basic operations there for different input sizes and types.

**Theoretical and Empirical Results**

**Average Case Time Complexity Of Median Of Three Quick-sort**

It’s same as Quicksort = **O(n\*logn)**

**Theroetically:**

**=** 0.424

**Empirical Results:**

**Average Case Array=**  **=** **0.405**

**Sorted Array =** =**0.326**

**Reverse Sorted=**= **0.307**

**Duplicated=**= **0.028**

**Comparing Pivot Selection**

Every different pivot selection in Quicksort has its own advantages and disadvantages. Selecting the first element as a pivot simplifies the partition progress by removing comparison and exchange operations. Because we know pivot is always at the left section.

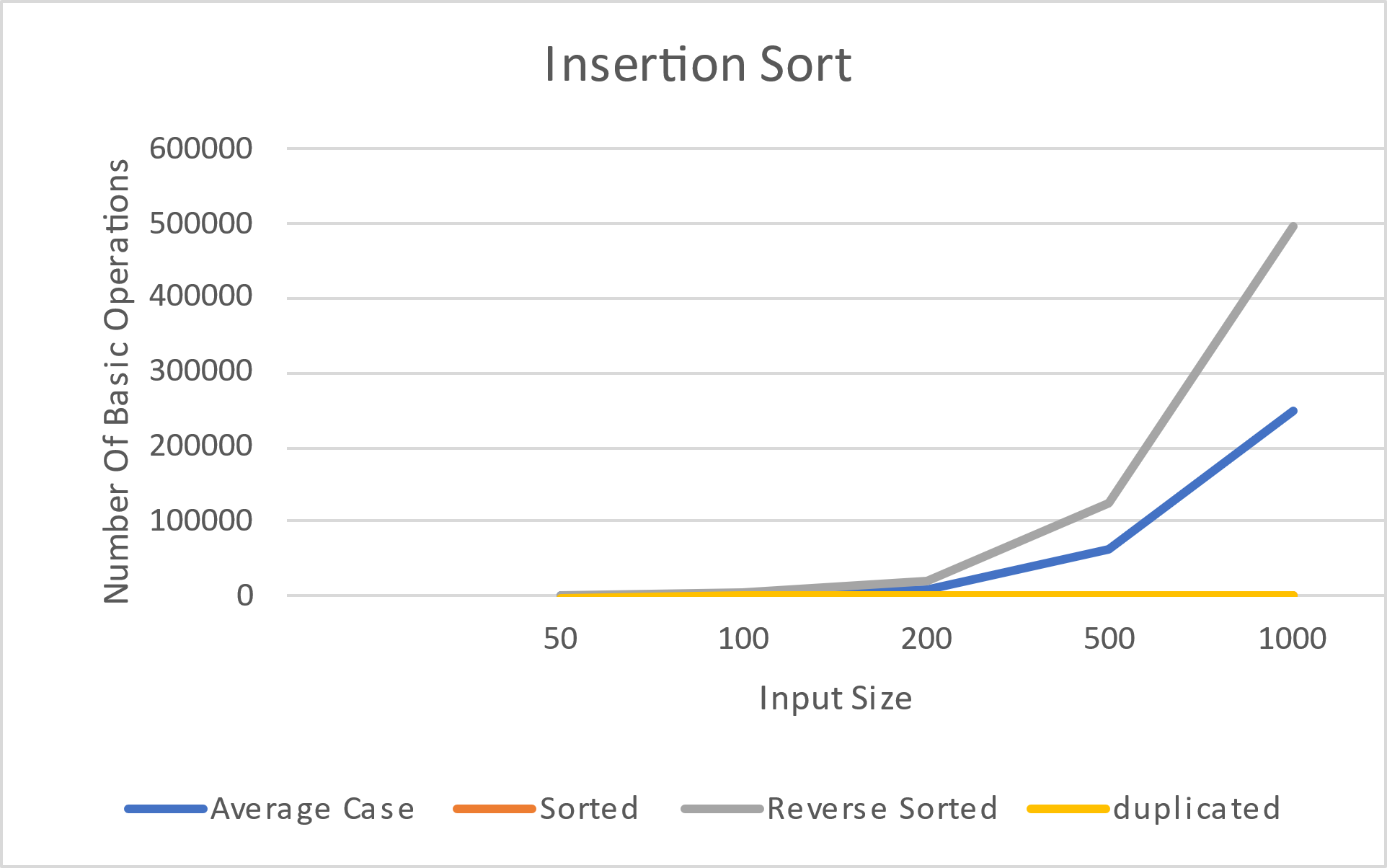
But when the algorithm works in Sorted data, this pivot selection gives us an error and makes its worst case O().

Overall In sorted array case median of three QuickSort is more efficient than normal quicksort.

**INSERTION SORT**

Insertion sort is done by taking the elements one by one. Starting from the second element this algorithm compares the elements to the elements to the left of it and places them in the correct position one by one.

Insertion sort is efficient for small data sets. It is less efficient on large lists than more advanced algorithms such as quicksort, heapsort or mergesort.



There is a graphic above for insertion sort. It shows how many basic operations are made for different types and sizes of arrays.   
The basic operation of insertion sort is comparison. And we counted number of comparisons.



Time complexity of insertion sort is O().

The best case scenario for this algorithm is if the given array is already sorted. And in that case the time complexity of the insertion sort is O(n).

**Theoretical and Empirical Results**

We tried this algorithm with different input sized arrays.

For example if we compare 50 and 100 sized arrays.

**For the best case** (sorted array) : Time complexity is O(n) so:

= 0.5 🡪 Theoretical result

= 0.495 🡪 Empirical result

**For the average case:** Time complexity is O() so:

= 0.25 🡪 Theoretical result

= 0.2 🡪 Empirical result

**For the reverse sorted case:** Time complexity is O() so:

= 0.25 🡪 Theoretical result

= 0.253 🡪 Empirical result

**For the duplicated case:** Time complexity is O(n) so:

= 0.5 🡪 Theoretical result

= 0.495 🡪 Empirical result

We can see that they are very similar.

**BINARY INSERTION SORT**

Binary Insertion Sort is similar to Insertion Sort. But to insert a new element into a sorted subarray, we use binary search algorithm to find the correct position for that element.

Binary Insertion Sort will reduce the number of comparisons in Insertion Sort, but the number of exchange elements does not still change.

In normal insertion sort, it takes O(n) comparisons in the worst case. We can reduce it to O(log n) by using binary search.

There is a graphic above for binary insertion sort. It shows how many basic operations are made for different types and sizes of arrays.



The complexity of the binary search is O(nlogn).

And the complexity of the swap operation is O(n) on the best case and O(n²) on the worst case. So:

In the best case of the Binary Insertion Sort, its total number of comparisons is O(nlogn) (because nlogn of the binary search grows faster than n of the swap).

And in the worst case, its total number of comparisons is O(n²) (because n² of the swap grows faster than nlogn of the binary search).

**Theoretical and Empirical Results**

If we compare 50 and 100 sized arrays.

**For the best case**: In this case we can use an already sorted array or an array that all its elements are the same. In this example we used a sorted array for best case.

Time complexity is O(nlogn) so:

🡪 = 0.425 🡪 Theoretical result

= 0.414 🡪 Empirical result

**For the average case:** Time complexity is O() so:

= 0.25 🡪 Theoretical result

= 0.19 🡪 Empirical result

**For the reverse sorted case:** Time complexity is O() so:

= 0.25 🡪 Theoretical result

= 0.248 🡪 Empirical result

**For the duplicated case:** Time complexity is O(nlogn) so:

🡪 = 0.425 🡪 Theoretical result

= 0.414 🡪 Empirical result

We can see that they are very similar.

**HEAP SORT**

Heapsort is a comparison-based sorting algorithm. This algorithm first creats a heap of the unsorted array. Then creates a sorted array by removing the minimum element from the heap and placing it into the beginning of the array. Then it does the same operation repeatedly.

There is a graphic above for heap sort. It shows how many basic operations are made for different types and sizes of arrays.

The basic operation of heap sort is comparison.



Heap sort’s best and worst cases are both O(nlogn).

**Theoretical and Empirical Results**

If we compare 50 and 100 sized arrays.

All of the cases have time complexity of O(nlogn) so :

**For the sorted case**:

🡪 = 0.425 🡪 Theoretical result

= 0.423 🡪 Empirical result

**For the average case:** Time complexity is O(nlogn) so:

🡪 = 0.425 🡪 Theoretical result

= 0.435 🡪 Empirical result

**For the reverse sorted case:** Time complexity is O(nlogn) so:

🡪 = 0.425 🡪 Theoretical result

= 0.408 🡪 Empirical result

**For the duplicated case:** Time complexity is O(nlogn) so:

🡪 = 0.425 🡪 Theoretical result = 0.496 🡪 Empirical result

We can see that they are very similar.

**MERGE SORT ANALYSATION**

Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist has an only single element and merging those sublists that creates a sorted list.

Merge Sort is very fast, and has a time complexity of O(nlogn) for every cases.(Best,average,worst). We declare basic operation is comparison. So we count comparisons for our graphs and data.

There is a graphic above for merge sort. It shows how many basic operations are made for different types and sizes of arrays.

We can see in randomized(average) array merge sort works very cumbersomely.

It works similarly for sorted-reverse sorted-duplicate arrays but in duplicate array, merge sort process fastest.

It doesn't show the sorted because it is very similar numbers as the same as reverse sorted. There is a table that clarifies our input sizes and basic operation numbers.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Input Size |  | 50 | 100 | 200 | 500 | 1000 |
| Average Case | | 220 | 532 | 1261 | 3869 | 8680 |
| Sorted |  | 153 | 356 | 812 | 2272 | 5044 |
| Reverse Sorted | | 141 | 347 | 832 | 2506 | 5629 |
| Duplicated | | 153 | 356 | 812 | 2272 | 5044 |

Merge Sort has a time complexity of O(nlogn) for every case(Best,worst,average). Because as the table above us shows, it has similar basic operations for different array types.

**Theoretical and Empirical Results**

The merge process does not contain any nested loops, so it is executed with linear complexity.

We take N as 50. **Theoretically** it must be:

 ==  ==0.424

**Empirically:**

**For Average Case**: =0.413

**For Sorted Case=** **=** 0.429

**For Reverse Sorted Case:** =0.406

**For Duplicated Case:** **=** 0.429

Empirical results are very similar for Theoretical expectations. So it proves that we have done counting basic operations correctly.

**Analysis Of Counting Sort**

Unlike other sorting algorithms, counting sort is a non-comparison sort. This means it sorts without comparing values of the elements. Comparison sorts have a speed limit at Ω(nlogn). But counting sort doesn't have this limit so it can beat other sorts in terms of speed.

Counting sort doesn't have Ω(nlogn) limit but it has a range restriction.

For example, if the value of range is 50 all the inputs must be between this range.(0….50)

 static void countSort(int arr[], int size, int countarray[]) {

        for (int i = 0; i < size; i++) {

            countarray[arr[i]]++;

        }

        int j = 0;

        for (int i = 0; i < size; i++) {

            if (j < 10001) {

                while (countarray[j] == 0) {

                    j++;

                }

                if (j > 10001) {

                    arr[i] = j;

                    j++;

                }

            }

        }

    }

In the first iteration, we iterate 0 to input size for making a duplicate array so its running time is Θ(n). In the second iteration we iterate 0 to input size because we know the range in our project is 10000. So time complexity of this sort will be O(n)

When we don't know the range already. it has to be O(k+n) to O(n) but we know the range so we don't make a loop for range

This graph shows counting sort is running in linear time. So counting sort is the best option if you know the range of the list you want to sort.

This shows that there is no time complexity difference between types of the array when counting sort does sorting. The algorithm goes to every element of the array no matter what type of the array is.

**CONCLUSION**

After analyzing every algorithm in different input sizes and types. We decided in our experiment most efficient sorting algorithm is the counting sort algorithm. Because we got range size of 10000. So it works on the time complexity of O(n) for every case and size. But if we don't have a specified range this algorithm wouldn't be most efficient.