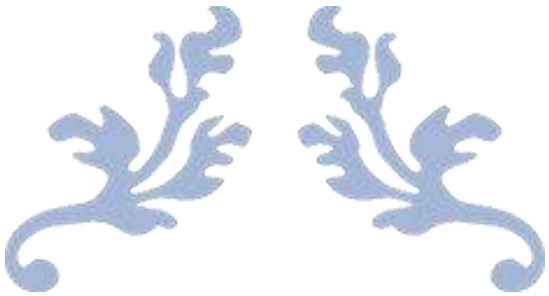
***Due Date: 05.05.2019***



# *CSE 2046.1 ANALYSIS OF ALGORITHM*

## *HOMEWORK #2*


***COURSE INSTRUCTOR**: Assistant Prof. Ömer Korçak*

***COURSE T.A** : Muhammed Nur Avcil*

***GROUP MEMBERS***

*150116065 – Deniz Arda Gürzihin*
*150117509 -- Mustafa Abdullah Hakkoz*


## *1.0) What is an algorithm?*

An algorithm is a detailed series of instructions for carrying out an operation or solving a problem. In a non-technical approach, we use algorithms in everyday tasks, such as a recipe to bake a cake or a do-it-yourself handbook.

Technically, computers use algorithms to list the detailed instructions for carrying out an operation. For example, to sort an array, the computer uses an algorithm. To accomplish this task, appropriate data must be entered into the system. In terms of efficiency, various algorithms are able to accomplish operations or problem solving easily and quickly.

Lets continoue on the example; to sort an array we can use lots of different algorithms such as

1. Selection Sort
2. Bubble Sort
3. Recursive Bubble Sort
4. Insertion Sort
5. Recursive Insertion Sort
6. Merge Sort
7. Iterative Merge Sort
8. Quick Sort
9. Iterative Quick Sort
10. Heap Sort
11. Counting Sort
12. Radix Sort
13. Bucket Sort
14. ShellSort
15. TimSort
16. Comb Sort
17. Pigeonhole Sort
18. Cycle Sort
19. Cocktail Sort
20. Strand Sort

.
.
.

## *Homework 2*

But in this homework we will just make experiment and implement

1. Insertion-sort,
2. Merge-sort,
3. Quick-sort,
4. Heap-sort,
5. Counting-sort

Firstly, lets start with how this algorithms works and their average, best and worst case.

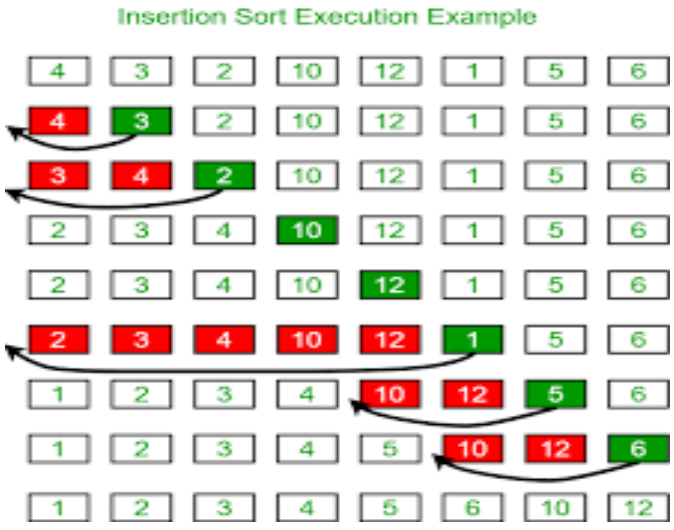
## *1.1) Insertion Sort*

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

Pseudocode of the complete algorithm of the insertion sort

```
i ← 1
while i < length(A)
    x ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > x
        A[j+1] ← A[j]
        j ← j - 1
    end while
    A[j+1] ← x
    i ← i + 1
end while
```



**Best Case Of Insertion Sort:**

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(*n*)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

**Worst Case Of Insertion Sort:**

The simplest worst case input is an array sorted in reverse 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., $O(n^2)$).

**Average Case Of Insertion Sort**

Suppose that the array starts out in a random order. Then, on average, we'd expect that each element is less than half the elements to its left. In this case, on average, a call to insert on a subarray of k elements would slide k/2 of them. The running time would be half of the worst-case running time. But in asymptotic notation, where constant coefficients don't matter, the running time in the average case would still be $\Theta(n^2)$, just like the worst case.

## *1.2) Merge Sort*

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

**Idea:**

- ✓ Divide the unsorted list into N sublists, each containing 1 element.
- ✓ Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. N will now convert into N/2 lists of size 2.
- ✓ Repeat the process till a single sorted list of obtained.

While comparing two sublists for merging, the first element of both lists is taken into consideration. While sorting in ascending order, the element that is of a lesser value becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the new combined sublist comprises all the elements of both the sublists.

***Pseudocode For Merge Sort***

```
procedure mergesort( var a as array )
  if ( n == 1 ) return a

  var l1 as array = a[0] ... a[n/2]
  var l2 as array = a[n/2+1] ... a[n]
```

```
  l1 = mergesort( l1 )
  l2 = mergesort( l2 )

  return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

  var c as array
  while ( a and b have elements )
    if ( a[0] > b[0] )
      add b[0] to the end of c
      remove b[0] from b
    else
      add a[0] to the end of c
      remove a[0] from a
    end if
  end while

  while ( a has elements )
    add a[0] to the end of c
    remove a[0] from a
  end while

  while ( b has elements )
    add b[0] to the end of c
    remove b[0] from b
  end while

  return c

end procedure
```

## *Time Complexity Of Merge Sort*

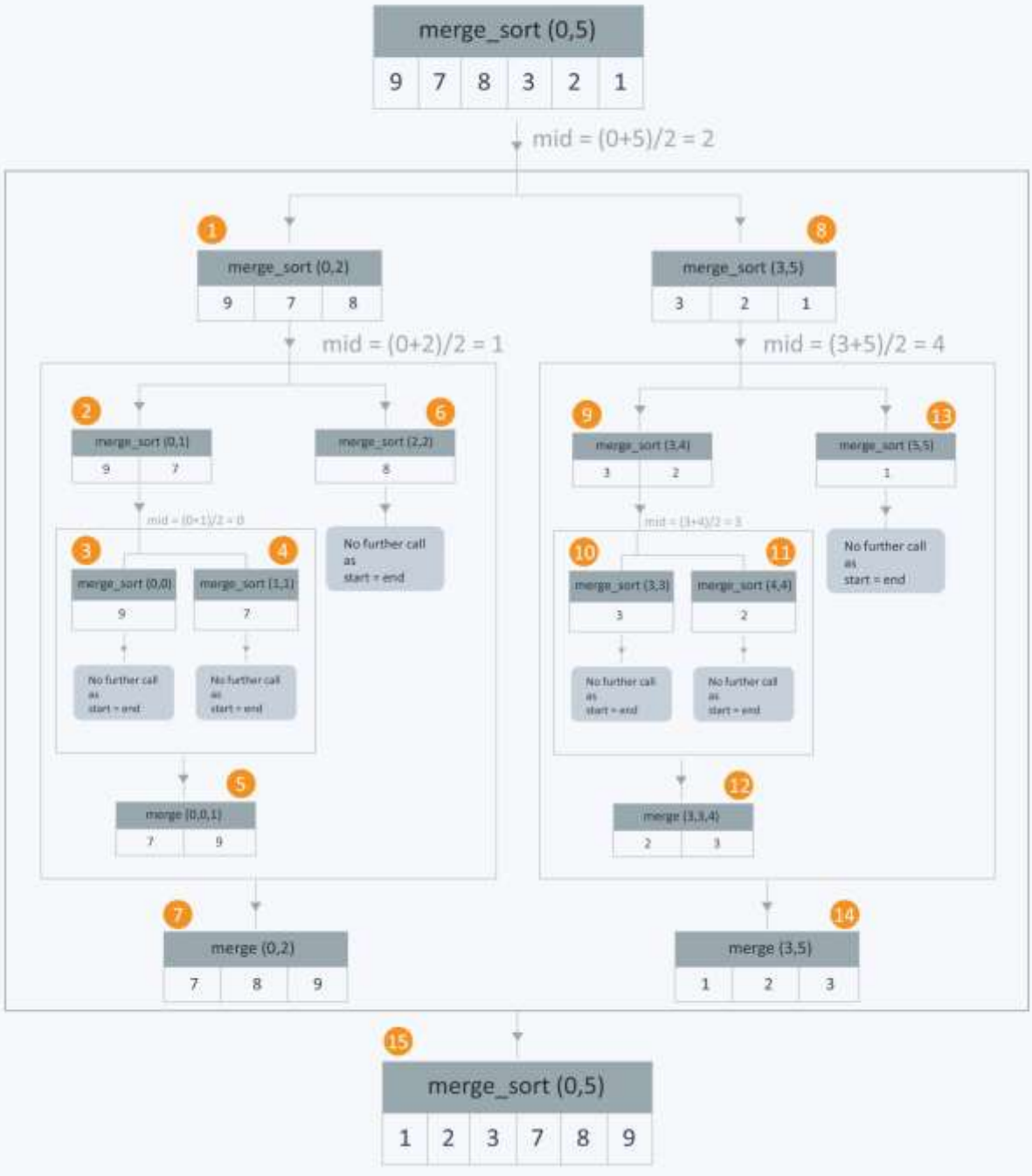
Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.

$$T(n) = 2T(n/2) + \Theta(n)$$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is ***Θ(nlog(n))***. Since the algorithm is same for the input size, it doesnt mather its worst, best or average case and it is

# ***Θ(nlog(n))***.

# Merge Sort


merge_sort (0,5)
9 7 8 3 2 1
mid = (0+5)/2 = 2
1
merge_sort (0,2)
9 7 8
mid = (0+2)/2 = 1
8
merge_sort (3,5)
3 2 1
mid = (3+5)/2 = 4
2
merge_sort (0,1)
9 7
6
merge_sort (2,2)
8
No further call as start = end
9
merge_sort (3,4)
3 2
13
merge_sort (5,5)
1
No further call as start = end
3
merge_sort (0,0)
9
4
merge_sort (1,1)
7
No further call as start = end
No further call as start = end
10
merge_sort (3,3)
3
11
merge_sort (4,4)
2
No further call as start = end
No further call as start = end
5
merge (0,0,1)
7 9
12
merge (3,3,4)
2 3
7
merge (0,2)
7 8 9
14
merge (3,5)
1 2 3
15
merge_sort (0,5)
1 2 3 7 8 9

## *1.3) Quick Sort*

Quick sort is a divide and conquer algorithm. Quick sort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quick sort can then recursively sort the sub-arrays. The steps are:

1. ***Pick*** an element, called a *pivot*, from the array.
2. ***Partitioning***: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. ***Recursively*** apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The base case of the recursion is arrays of size zero or one, which are in order by definition, so they never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance. You can select pivot like these;

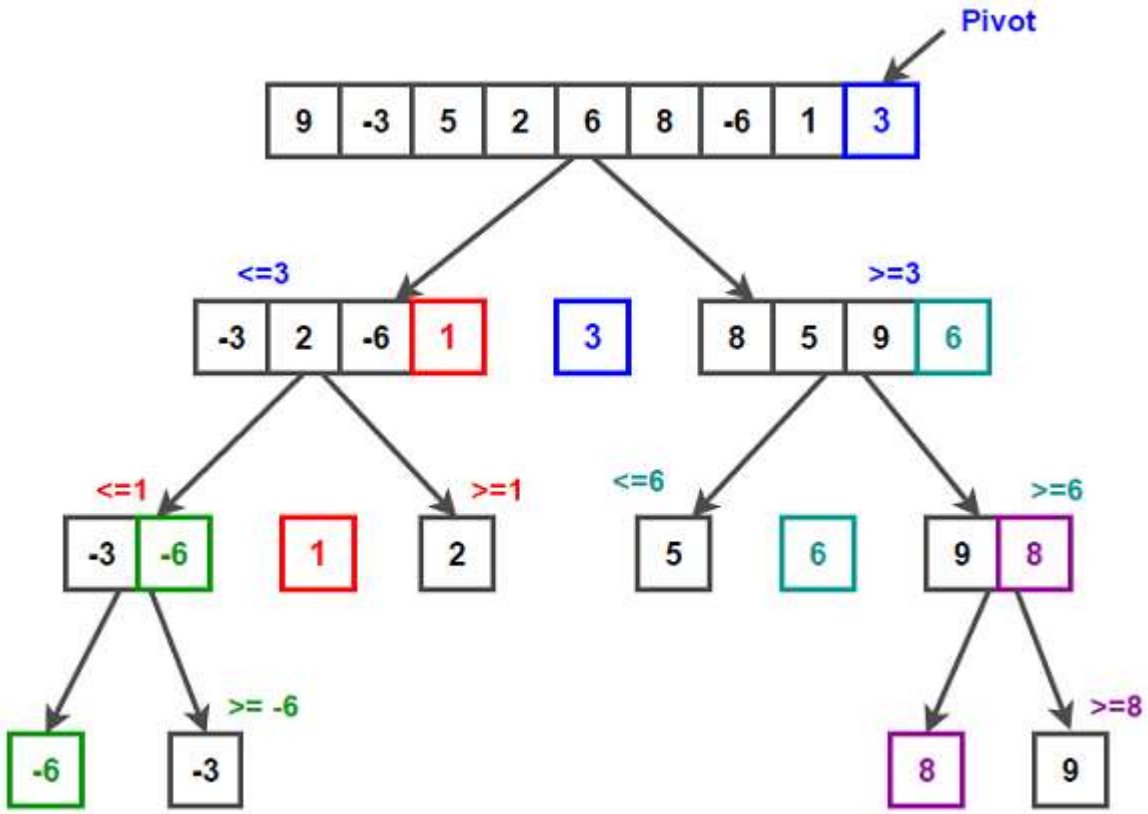
- Always pick first element as pivot
- Always pick last element as pivot
- Pick a random element as pivot
- Pick median as pivot

…

***Pseudocode Of Quick Sort***

```
 /* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}

/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element
```

```
    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than or
        // equal to pivot
        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)
}
```




**Best Case Of Quick Sort:**

The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$$T(n) = 2T(n/2) + \Theta(n)$$

The solution of above recurrence is $\Theta(n\log(n))$. It can be solved using case 2 of Master Theorem.

**Worst Case Of Quick Sort:**

The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

$$T(n) = T(0) + T(n-1) + \Theta(n)$$
which is equivalent to
$$T(n) = T(n-1) + \Theta(n)$$

The solution of above recurrence is $\Theta(n^2)$.

**Average Case Of Quick Sort:**

Let $C_{avg}(n)$ be the average number of key comparisons made by quicksort on a randomly ordered array of size n. A partition can happen in any position $s(0 \le s \le n-1)$ after n+1 comparisons are made to achieve the partition.

After the partition, the left and right subarrays will have s and n-1-s elements, respectively. Assuming that the partition split can happen in each position s with the same probability 1/n, we get the following recurrence relation:

$$C_{avg}(n) = 1/n \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \qquad \text{for } n > 1;$$

$$C_{avg}(0)=0, \quad C_{avg}(1)=0.$$

Its solution, which is much trickier than the worst- and best-case analyses, turns out to be

$$C_{avg}(n) \approx 2n\ln(n) \approx 1.39\log_2(n)$$

## *1.4) Heap Sort*

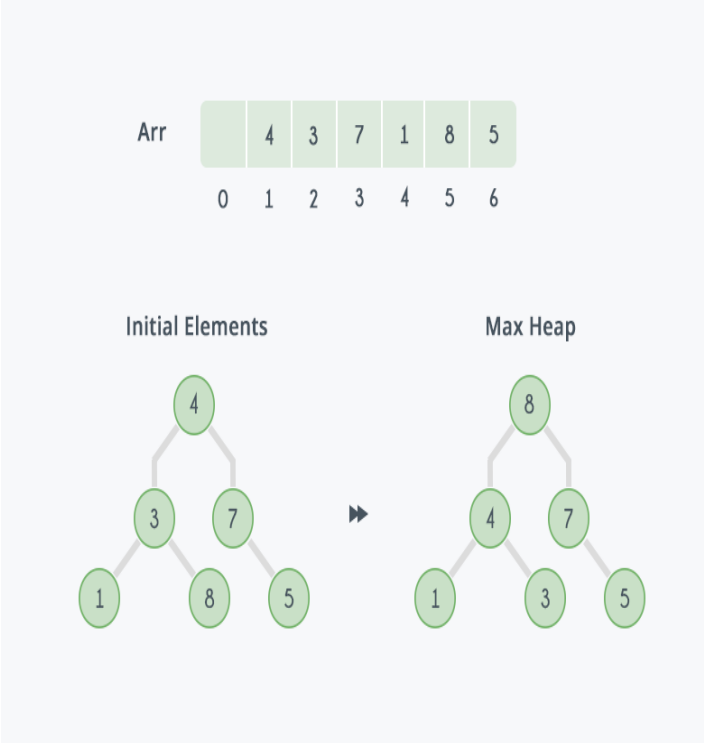
The Heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

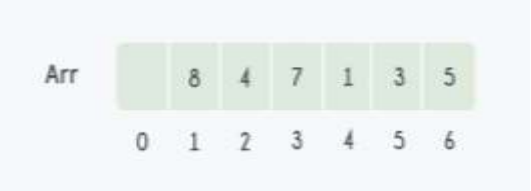
The steps are:

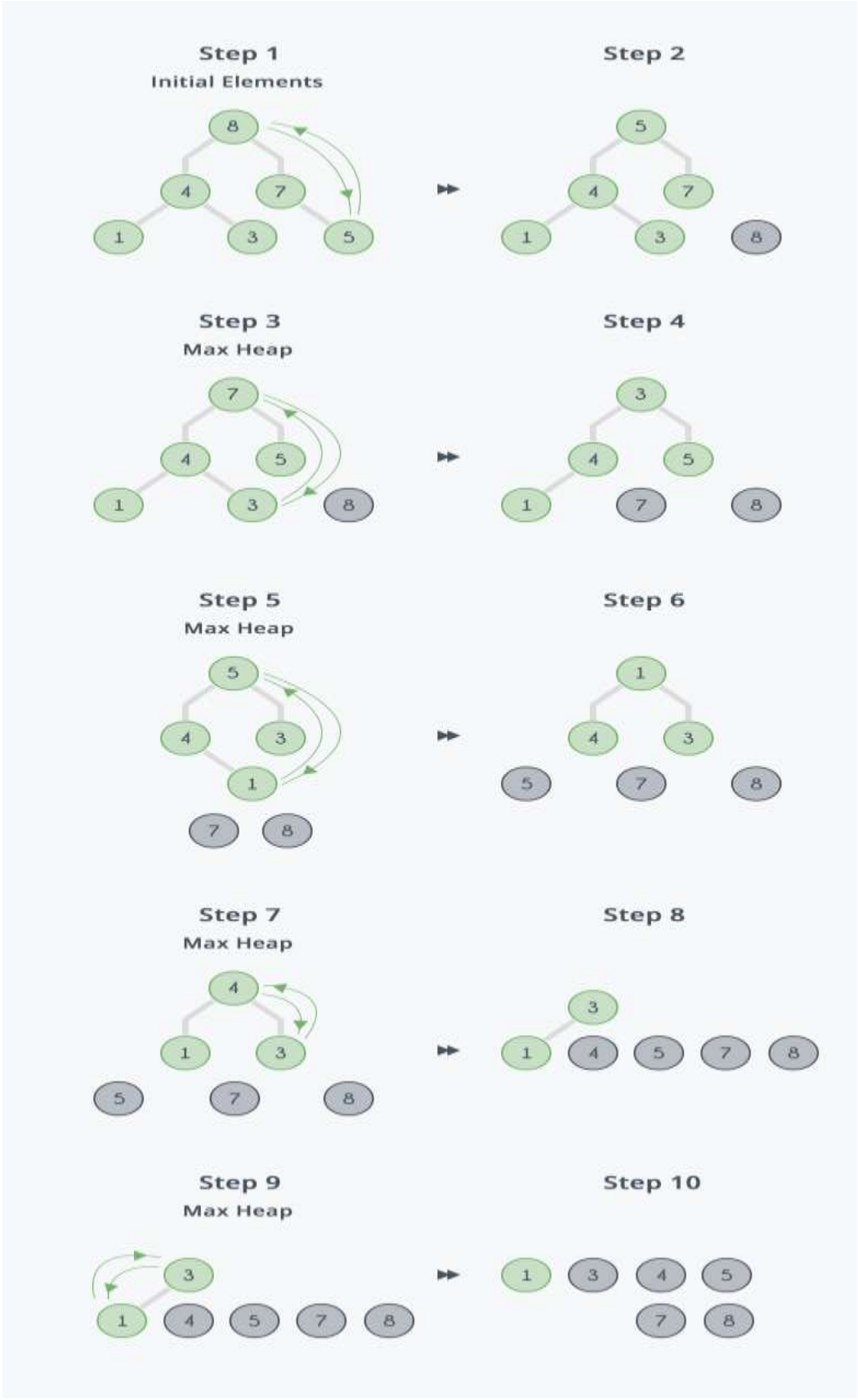
1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the siftDown() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

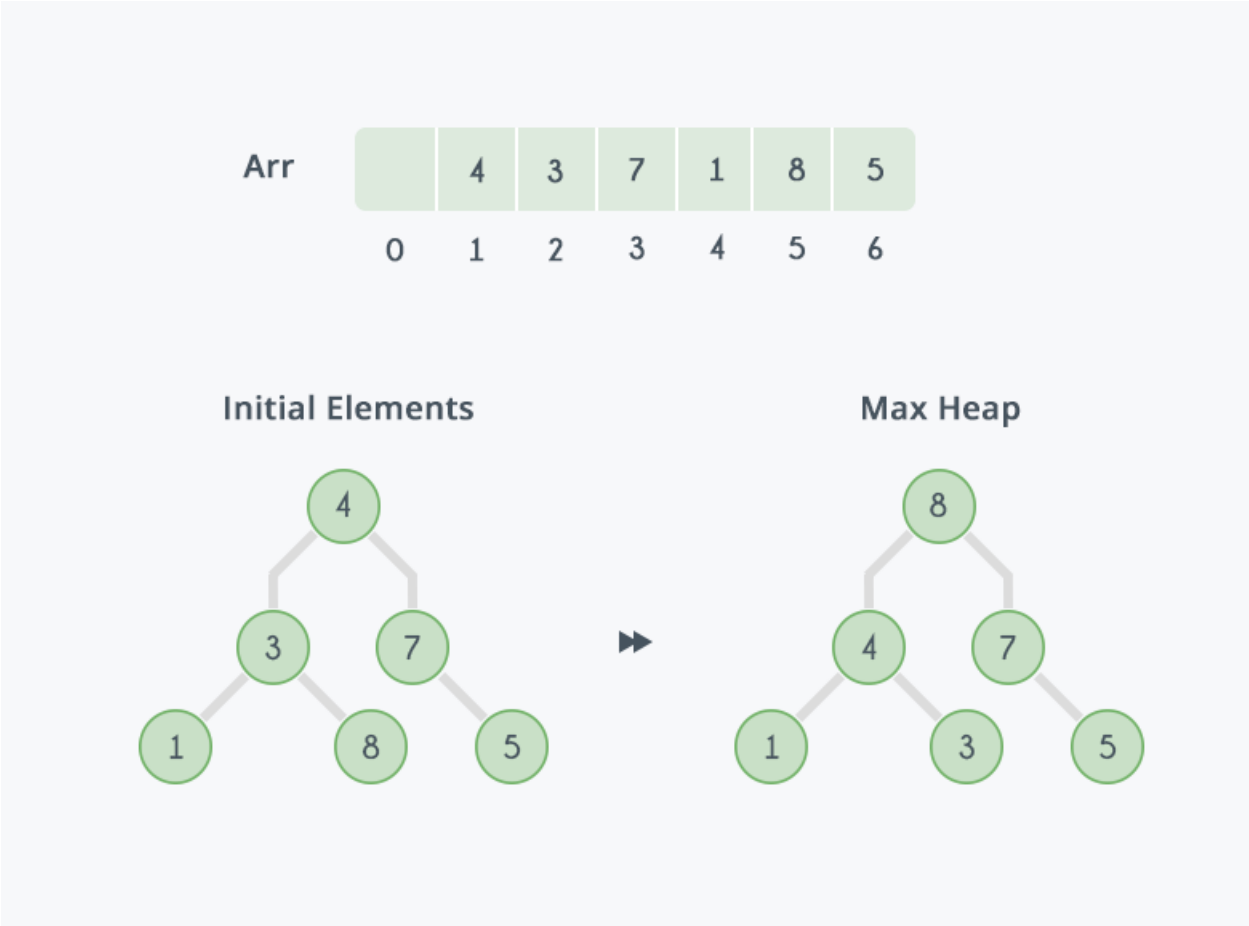
***Pseudocode For Heap Sort***

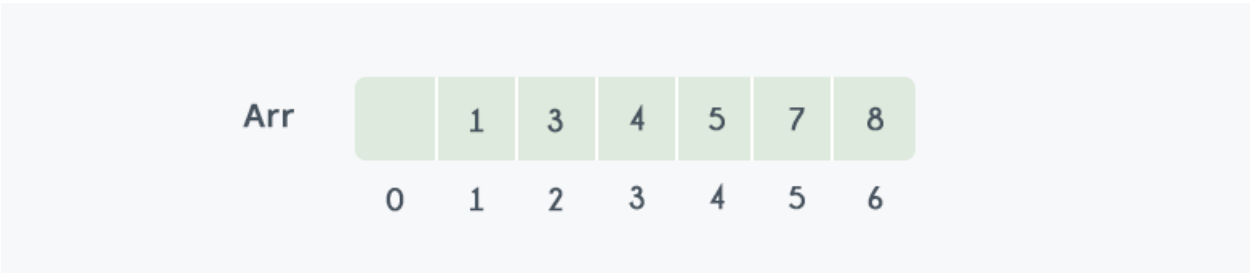
```
Heapsort(A as array)

    BuildHeap(A)
    for i = n to 1
        swap(A[1], A[i])
        n = n - 1
        Heapify(A, 1)

BuildHeap(A as array)
    n = elements_in(A)
    for i = floor(n/2) to 1
        Heapify(A,i,n)

Heapify(A as array, i as int, n as int)
    left = 2i
    right = 2i+1

    if (left <= n) and (A[left] > A[i])
        max = left
    else
        max = i

    if (right<=n) and (A[right] > A[max])
        max = right

    if (max != i)
        swap(A[i], A[max])
        Heapify(A, max)
```




Step 1
Initial Elements
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10




***Time Complexity Of Heap Sort***

The buildMaxHeap() operation is run once, and is O(*n*) in performance. The siftDown() function is O(log *n*), and is called *n* times. Therefore, the performance of this algorithm is O(*n* + *n* log *n*) = ***O(n log n).***

## *1.5) Counting Sort*

Counting sort is a stable sorting technique, which is used to sort objects according to the keys that are small numbers. It counts the number of keys whose key values are same. This sorting technique is effective when the difference between different keys are not so big, otherwise, it can increase the space complexity.

***Pseudocode For Counting Sort***

```
CountingSort(A)
  //A[]-- Initial Array to Sort
  //Complexity: O(k)
  for i = 0 to k do
  c[i] = 0

  //Storing Count of each element
  //Complexity: O(n)
  for j = 0 to n do
  c[A[j]] = c[A[j]] + 1

  // Change C[i] such that it contains actual
  //position of these elements in output array
  ////Complexity: O(k)
  for i = 1 to k do
  c[i] = c[i] + c[i-1]

  //Build Output array from C[i]
  //Complexity: O(n)
  for j = n-1 downto 0 do
  B[ c[A[j]]-1 ] = A[j]
  c[A[j]] = c[A[j]] - 1
        end func
```

***Time Complexity Of Counting Sort***

In this algorithm, the initialization of the count array and the loop which performs a prefix sum on the count array takes O(k) time. And other two loops for initialization of the output array takes O(n) time. Therefore, the total time complexity for the algorithm is : **O(k)+ O(n)+ O(k)+ O(n)= O(n+k).**

Worst Case Time complexity: O (n+k)
Average Case Time complexity: O(n+k)
Best Case Time complexity: O(n+k)
Space Complexity: O(k)

**Input Data**

| 0 | 4 | 2 | 2 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 2 | 4 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Count Array**

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 5 | 3 | 4 | 0 | 2 |

**Sorted Data**

| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 4 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

### *1.6) All Cases Of Sort Algorithms*

| Cases/Algorithms | Insertion Sort | Merge Sort | Quick Sort | Heap Sort | Counting Sort |
|---|---|---|---|---|---|
| Worst Case | $\Theta(n^2)$ | $\Theta(n\log(n))$ | $\Theta(n^2)$ | $\Theta(n\log(n))$ | O(n+k) |
| Average Case | $\Theta(n^2)$ | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | O(n+k) |
| Best Case | $\Theta$ (n) | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | $\Theta(n\log(n))$ | O(n+k) |

## *2.0) Code Implementation*

Secondly, Lets take a look how we implement all this sort algorithms.

### *2.1) Insertion Sort*

```
1.  void insertion_sort(int *a, int n) {
2.      for(size_t i = 1; i < n; ++i) {
3.          int tmp = a[i];
4.          size_t j = i;
5.          while(j > 0 && tmp < a[j - 1]) {
6.              a[j] = a[j - 1];
7.              --j;
8.          }
9.          a[j] = tmp;
10.     }
11. }
```

Implementation of insertion sort consists only one pice of code, a void function (`void insertion_sort`) which takes an integer array (`int *a`) and size of it (`int n`) as parameters. Then apllies the insertion sort algorithm which is defined at **section 1.1**.

### 2.2) *Merge Sort*

```
1.  void merge (int *a, int n, int m) {
2.      int i, j, k;
3.      int *x = malloc(n * sizeof (int));
4.      for (i = 0, j = m, k = 0; k < n; k++) {
5.          x[k] = j == n      ? a[i++]
6.               : i == m      ? a[j++]
7.               : a[j] < a[i] ? a[j++]
8.               :               a[i++];
9.      }
10.     for (i = 0; i < n; i++) {
11.         a[i] = x[i];
12.     }
13.     free(x);
14. }
15.
16. void merge_sort (int *a, int n) {
17.     if (n < 2)
18.         return;
19.     int m = n / 2;
20.     merge_sort(a, m);
21.     merge_sort(a + m, n - m);
22.     merge(a, n, m);
23. }
```

Implementation of merge sort consists two pieces of code, a void function `merge` and another void function `merge_sort`. First function takes an integer array (`int *a`), size of it (`int n`) and middle index of array (`int m`) as parameters. This function is used in second one, to sort the array while merging two parts of it.

Second function (`merge_sort`) takes an integer array (`int *a`) and size of it (`int n`) as parameters. Calculates m value which is necessary for first function then calls it and repeats this part recursively to split array to two parts.

### 2.3) *Quick Sort*

```
1.  void quick_sort(int *A, int len) {
2.    if (len < 2) return;
3.
4.    int pivot = A[len / 2];
5.
6.    int i, j;
7.    for (i = 0, j = len - 1; ; i++, j--) {
8.      while (A[i] < pivot) i++;
9.      while (A[j] > pivot) j--;
10.
11.     if (i >= j) break;
12.
13.     int temp = A[i];
14.     A[i]     = A[j];
15.     A[j]     = temp;
16.   }
17.
18.   quick_sort(A, i);
19.   quick_sort(A + i, len - i);
20. }
```

Implementation of quick sort consists only one pice of code, a void function (`void quick_sort`) which takes an integer array (`int *a`) and size of it (`int len`) as parameters. Chooses middle element as pivot and applies the quick sort algorithm which is defined at **section 1.2**.

### *2.4) Heap Sort*

```
1.  int max (int *a, int n, int i, int j, int k) {
2.      int m = i;
3.      if (j < n && a[j] > a[m]) {
4.          m = j;
5.      }
6.      if (k < n && a[k] > a[m]) {
7.          m = k;
8.      }
9.      return m;
10. }
11.
12. void downheap (int *a, int n, int i) {
13.     while (1) {
14.         int j = max(a, n, i, 2 * i + 1, 2 * i + 2);
15.         if (j == i) {
16.             break;
17.         }
18.         int t = a[i];
19.         a[i] = a[j];
20.         a[j] = t;
21.         i = j;
22.     }
23. }
24.
25. void heap_sort (int *a, int n) {
26.     int i;
27.     for (i = (n - 2) / 2; i >= 0; i--) {
28.         downheap(a, n, i);
29.     }
30.     for (i = 0; i < n; i++) {
31.         int t = a[n - i - 1];
32.         a[n - i - 1] = a[0];
33.         a[0] = t;
34.         downheap(a, n - i - 1, 0);
35.     }
36. }
```

Implementation of heap sort consists three pieces of code, an integer function `max` and two void functions `downheap` and `heap_sort`. First function takes an integer array (`int *a`), size of it (`int n`) and three integers (`int i, int j, int k`) as parameters. This function is used in second one, to find out max element among three indexes i, j and k.

Second function (`downheap`) takes an integer array (`int *a`), size of it (`int n`) and integer (`int i`) which is calculated in third function, as parameters. Calculates max element and swaps it with *i*th element.

Third function (`heap_sort`) takes an integer array (`int *a`) and size of it (`int n`) as parameters. Calculates integer i, and use it as input while calling function `downheap`. Then swaps last child with root, removes last child and calls `downheap` again to rearrange the tree to assure heap conditions.

### *2.5) Counting Sort*

```
1.  void counting_sort(int *array, int n)
2.  {
3.    int i, j, z, min, max;
4.
5.    min = max = array[0];
6.    for(i=1; i < n; i++) {
7.      if ( array[i] < min ) {
8.        min = array[i];
9.      } else if ( array[i] > max ) {
10.       max = array[i];
11.     }
12.   }
13.
14.   int range = max - min + 1;
15.   int *count = malloc(range * sizeof(*array));
16.
17.   for(i = 0; i < range; i++) count[i] = 0;
18.   for(i = 0; i < n; i++) count[ array[i] - min ]++;
19.
20.   for(i = min, z = 0; i <= max; i++) {
21.     for(j = 0; j < count[i - min]; j++) {
22.       array[z++] = i;
23.     }
24.   }
25.
26.   free(count);
27. }
```

Implementation of quick sort consists only one pice of code, a void function (`void counting _sort`) which takes an integer array (`int *a`) and size of it (`int len`) as parameters. Calculates min - max elements of array, then build a new array with range of [max-min] and fills it with zeros. Stores count of each element in the new array and in final step, builds the output array.

Then we have tested this algorithms with following several kinds of input that we have created.

## *3.0) Input Selection*

We have selected 8 different kinds of input to test this algorithms. These are;

- An array with full of same constant number

- An array with reverse sorted integer
- An array with fully random integer
- An array with all numbers are sorted
- An array with at most %15 percent of unsorted
- An array with last 256 elements are unsorted
- An array with numbers are repating itself rapidly
- A sorted array with numbers are repating itself rapidly
- An array with just number is repating oftenly

Reverse ordered array and sorted array had been chosed to show best case and worst case of the insertion sort algorithm, average case can be determined by random array.

Since merge sort has Θ(nlog(n)) time complexity, there is no certain worst, best or average case to show in input, the expecting result is to show the same result for all inputs.

Quick sort could have worst case if it has implemented such a way that the pivot selected as first or last element. But if median has implemented as pivot then there is no worst case for quicksort. And we have made the implementation with median so O(n(log(n)) time complexity has been waiting for all case of quick sort.

Since the heap sort has heapConstrucion() function with O(n) time complexity and downheap() function with O(log(n)) time complexity, the final will be O(nlog(n)) time complexity. Thus, there is no specified input to see different time complexity as result.

In counting sort, it is also the same, there is no specified input that makes algorithm to pass the O(n+k) time complexity. Because counting sort count the value of the same keys which takes O(n) time and O(k) times to initialize frequencies for all cases.

Then we have decided on how many elements do we need to store on these arrays to see the how, time complexity is changes during the experiment. We have decided these input sizes;

- ✓ 100,
- ✓ 500,
- ✓ 1000,
- ✓ 1024,
- ✓ 4000,
- ✓ 4096,
- ✓ 8000,
- ✓ 8192,
- ✓ 30000,
- ✓ 32768,
- ✓ 60000,
- ✓ 65536,
- ✓ 130000,
- ✓ 131072
- ✓

Powers of 2 and theirs nearest rounded values are selected to observe if there will be any major difference between them.

## *4.0) Input Implementation*

Then we have implemented this inputs on C language. Our input has been created during the life time of the program so we have not created a input file. These are the algorithms that we have used to create inputs.

### *4.1) Constant Number Array*

C Implementation Of This Array;

```
1.  static int fill_constant;
2.
3.  void setfillconst(int c)
4.  {
5.    fill_constant = c;
6.  }
7.
8.  void fillwithconst(int *v, int n)
9.  {
10.   while( --n >= 0 ) v[n] = fill_constant;
11. }
```

All array has full filled with one constant number. This number can be set with setfillconst() function.

### *4.2) Sorted Array*

C Implementation Of This Array;

```
1.  //required function to use built-in qsort()
2.  int intcompare( const void *a, const void *b )
3.  {
4.    if ( *(const int *)a < *(const int *)b ) return -1;
5.    else return *(const int *)a > *(const int *)b;
6.  }
7.
8.  //All elements are sorted
9.  void sortedarray(int *v, int n){
10.     int length; length=n; int max_number = 1001;
11.     int a=0;
12.     for(a=0; a<length; a++){
13.         v[a] = rand() % max_number;
14.     }
15.
16.     qsort(v, n, sizeof(int), intcompare);
17. }
```

First we start with assign random numbers to the array in the mod of 1001 cause in this homework input elements must be in 0 to 1000 range. Then we sort array with qsort function which is in C library.

### 4.3) *Reverse Order Array*

C Implementation Of This Array;

```
1.  //rerverse sorted
2.  void reversesorted(int *v, int n)
3.  {
4.    int c, d;
5.    int *b = malloc(n * sizeof(int));
6.    sortedarray(v, n);
7.    for (c = n - 1, d = 0; c >= 0; c--, d++)
8.      b[d] = v[c];
9.    for (c = 0; c < n; c++)
10.     v[c] = b[c];
11. }
```

In reverse sort, firstly we have called the function sortedarray to create an array and sort it. Then, with a temporary array, we have stored the corresponding reversed index value, then assign back this values to real array.

### *4.4) Array With Random Numbers*

C Implementation Of This Array;

```
5.0)    //shuffled array
6.0)    void shuffledarray(int *v, int n){
7.0)        int length; length=n; int max_number = 1001;
8.0)        int a=0;
9.0)        for(a=0; a<length; a++){
10.0)           v[a] = rand() % max_number;
11.0)       }
12.0)   }
```

In this input, first we have assign 1001 values to an integer variable, Then, we have used a for loop to reach all elements of array v and assigned them to random number in the range of 0 to 1000.

### *4.5) At Most %15 Unsorted Array*

C Implementation Of This Array;

```
1.  //%15 percent of array not sorted
2.  void almostsortedarray(int *v, int n){
3.      int length; length=n; int max_number = 1001;
4.      int a=0;
5.      for(a=0; a<length; a++){
6.          v[a] = rand() % max_number;
7.      }
8.
```

```
9.      qsort(v, n, sizeof(int), intcompare);
10.
11.     //change almost %15 of the array with random numbers
12.     long almost_element = (length*15)/100;
13.     for(a=0; a<almost_element; a++){
14.         int random_index = rand() % length;
15.         v[random_index] = rand() % max_number;
16.     }
17.
18. }
```

Firstly, an array with random number has been created then sorted by qsort function like in sorted array. After this, at most %15 of array element has been replaced by random numbers in the range of 0 to 1000. At most %15 of array because we implement such a way that while in for loop, the same index can be change multiple times.

### *4.6) Last 256 Elements Is Unsorted*

C Implementation Of This Array;

```
1.  //Last 256 element unsorted
2.  void last256notsortedarray(int *v, int n){
3.      int length; length = n; int max_number = 1001;
4.      int a=0;
5.      for(a=0; a<length; a++){
6.          v[a] = rand() % max_number;
7.      }
8.
9.      qsort(v, n, sizeof(int), intcompare);
10.
11.     //make random last 256 element
12.     if(n>256){
13.         int last;
14.         for(last=n-256; last<length; last++){
15.             v[last] = rand() % max_number;
16.         }
17.     }
18.
19. }
```

Firstly, like ***4.5) At most %15 Unsorted Array*** elements are randomly created then sorted. After this operation, if array has more than 256 elements, then we are replacing this last 256 numbers with new random numbers in the range of 0 to 1000.

### *4.7) Array With High Repetition For All*

C Implementation Of This Array;

```
1.  //All array highRepetition
```

```
void highRepetitionArray(int *v, int n){
    int length; length = n; int max_number = 11;
    int a=0;
    for(a=0; a<length; a++){
        v[a] = rand()%max_number;
    }
}
```

In this input type, we have implement such a way that all elements occuring rapidly. In order to do this, maximum number that can participate in array decrease 10 which means the range of the array decrease to 10. So, array will have same values a lot. For example, in 100 elements array, there is going to be one element that will repeat itself at least 10 times.

### *4.8) Sorted Array With High Repetition For All*

C Implementation Of This Array;

```
//All array highRepetition Sorted
void SortedHighRepetitionArray(int *v, int n){
    int length; length = n; int max_number = 11;
    int a=0;
    for(a=0; a<length; a++){
        v[a] = rand()%max_number;
    }

    qsort(v, n, sizeof(int), intcompare);
}
```

The same idea in 4.7 but with a one difference to see what can be change if this array were sorted. Sort operation had been done by the qsort().

### *4.9) Array With One Element Is Repeating*

C Implementation Of This Array;

```
// Just one element is repeating too much more than 1000 to length
void OneHighRepetitionArray(int *v, int n){
    int length; length = n; int max_number = 1001;
    int a=0;
    for(a=0; a<length; a++){
        v[a] = rand()%max_number;
    }

    if (n>1001){
        //en az 1000 en Ã§ok n-1
        int repetition_count = (rand()%(n-1000))+1000;
        int random_number = rand()%1001;
        int i=0;
        for(i=0; i<repetition_count; i++){
            int random_index = rand()%n;
            while(v[random_index] == random_number){
                random_index = rand()%n;
```

```
            }
            v[random_index] = random_number;
        }
    }

}
```

First of all, array with random numbers has been created. Then we have checked that does it have more than 1001element if it has, the number that is going to repeat and which number is going to be this number randomly choosen. Then, we have changed the elements with randomly chosen number for repeat number times. We did this with a for loop which start from 0 to repeat number and we have changed the random index values with random chosen number. So that, we have created an array which has one element such way that it at least repeat itself 1000 times to "length of array-1" times.

After the implementation of input has finished, we have tested all 5 sort algorithms with these 9 inputs to see which algorithm is better on which condition.

## *5.0) The Testing Invorenment*

In order to test the speed of the different sorting algorithms we made a C program which runs each algorithm several times for randomly-generated arrays.

The test was run in a Intel T9300 2.5GHz running Ubuntu 16.04, and the programs were compiled using gcc 3.3.5. Compiling steps are a bit complicated since there was two exe files with different compilation instructions. **writetimings.exe** for calculating running times then printing them to **.dat** files and **fitdata.exe** for calculating linear graph's coefficients which are necessary to draw polynomial fitting then printing them to **.fd** files. Further explanations for compilation can be found in **README.txt** in code's main folder.

The **rand()** function of **glibc** was used for random number generation and built-in **qsort()** function was used to prepare input arrays. In addition to that, **gsl/gsl_multifit** library was used for polinomial fitting. And **gnuplot** was used for drawing graphs. Drawing steps also can be found in **README.txt**.

## *6.0) Results*

We have tested our algorithms speed with 10 different input of the same category and we have found the mean value of 10 output and we have written this mean value to txt file. Additionally, the result had been found in terms of milisecond which means we have used milisecond metric system for our test result.

### *6.1) Tables*

By the result in the txt files, the tables given below has been created and same abbreviation has been used. These are;

- 4.1) for Constant Number Array
- 4.2) for Sorted Array
- 4.3) for Reverse Sorted Array
- 4.4) for Randomized Array
- 4.5) for At most %15Unsorted Array
- 4.6) for Last 256 is Unsorted Array
- 4.7) for High Repetition Array For All Elements
- 4.8) for High Repetition Array For All Elements Sorted
- 4.9) for One Element Repetition Array

#### *6.1.1) Insertion Sort*

| ***Size/Input*** | ***4.1)*** | ***4.2)*** | ***4.3)*** | ***4.4)*** | ***4.5)*** | ***4.6)*** | ***4.7)*** | ***4.8*** | ***4.9)*** |
|---|---|---|---|---|---|---|---|---|---|
| ***100*** | 0.00357 | 0.00179 | 0.00488 | 0.00359 | 0.00197 | 0.00199 | 0.00406 | 0.00199 | 0.00363 |
| ***500*** | 0.01001 | 0.00574 | 0.08494 | 0.04191 | 0.01120 | 0.02316 | 0.03999 | 0.00525 | 0.04270 |
| ***1000*** | 0.01795 | 0.00939 | 0.29257 | 0.14154 | 0.03290 | 0.05462 | 0.13101 | 0.00887 | 0.14867 |
| ***1024*** | 0.01833 | 0.01089 | 0.30224 | 0.15039 | 0.03352 | 0.06530 | 0.14288 | 0.00924 | 0.01047 |
| ***4000*** | 0.06660 | 0.03324 | 4.43066 | 2.12501 | 0.42207 | 0.26922 | 1.98183 | 0.03197 | 1.03203 |
| ***4096*** | 0.07068 | 0.03274 | 4.64255 | 2.25749 | 0.47358 | 0.29626 | 2.08945 | 0.03268 | 0.24924 |
| ***8000*** | 0.13109 | 0.05822 | 16.61986 | 8.60593 | 1.68448 | 0.55048 | 7.84895 | 0.06323 | 7.92381 |
| ***8192*** | 0.13399 | 0.05950 | 18.01901 | 8.81084 | 1.63485 | 0.60784 | 8.18498 | 0.06563 | 6.79289 |
| ***30000*** | 0.25776 | 0.24067 | 245.6943 | 114.48028 | 21.87954 | 2.46604 | 110.1865 | 0.23597 | 94.23529 |
| ***32768*** | 0.25481 | 0.24894 | 290.8995 | 139.82929 | 26.29214 | 2.58545 | 124.7249 | 0.25361 | 57.11689 |
| ***60000*** | 0.46666 | 0.43103 | 972.3152 | 466.63805 | 87.88203 | 4.74555 | 423.4565 | 0.46431 | 131.62234 |
| ***65536*** | 0.50719 | 0.49330 | 1116.251 | 556.01589 | 102.66261 | 5.02173 | 514.8088 | 0.51066 | 253.89020 |
| ***130000*** | 1.03203 | 0.98352 | 4450.522 | 2149.89990 | 408.52902 | 9.71029 | 2021.946 | 1.02020 | 906.45577 |
| ***131072*** | 1.00913 | 1.00781 | 4505.177 | 2213.26758 | 406.51001 | 9.87648 | 2309.823 | 1.02186 | 788.27724 |

By these table we can understand that Insertion Sort shows better performance for sorted array as explained in ***1.1 Best Case Of Insertion Sort*** also, as explained in ***1.1 Worst Case Of Insertion Sort,*** we have waited to observe that reverse ordered array must have taken the longest time. Now lets have a look to table to see for all input size the reversed ordered array took the longest time. For average case, we can certainly say, there is huge difference between sorted and unsorted array that comes from the knowledge that we have learned in ***1.1 Average Case Of Insertion Sort***. This difference comes from the time complexity difference where sorted array has O(n) and average cases unsorted array has $O(n^2)$.

***Compatibility Of Input Array Type for Insertion Sort***

4.1=4.2=4.8>4.6>4.5>4.9>4.4>4.7>4.3

### *6.1.2) Merge Sort*

| ***Size/Input*** | ***4.1)*** | ***4.2)*** | ***4.3)*** | ***4.4)*** | ***4.5)*** | ***4.6)*** | ***4.7)*** | ***4.8*** | ***4.9)*** |
|---|---|---|---|---|---|---|---|---|---|
| ***100*** | 0.02195 | 0.02018 | 0.02022 | 0.01935 | 0.01900 | 0.02054 | 0.01894 | 0.02099 | 0.01963 |
| ***500*** | 0.10574 | 0.10260 | 0.09708 | 0.10379 | 0.09771 | 0.10305 | 0.09873 | 0.11412 | 0.10113 |
| ***1000*** | 0.21406 | 0.21629 | 0.22388 | 0.21737 | 0.21065 | 0.20928 | 0.21118 | 0.21485 | 0.21287 |
| ***1024*** | 0.21658 | 0.21668 | 0.21199 | 0.22178 | 0.21409 | 0.21361 | 0.21709 | 0.21381 | 0.21338 |
| ***4000*** | 1.13565 | 0.95397 | 0.95467 | 0.98032 | 0.94781 | 0.94188 | 1.00641 | 0.94330 | 0.97628 |
| ***4096*** | 0.97467 | 0.97284 | 0.96070 | 1.00007 | 0.96759 | 0.96072 | 0.97610 | 0.96232 | 0.97696 |
| ***8000*** | 1.94974 | 1.94782 | 1.94543 | 2.09865 | 1.95759 | 1.93971 | 1.99632 | 1.93896 | 1.96949 |
| ***8192*** | 2.18981 | 1.97684 | 2.02583 | 2.05216 | 1.99749 | 1.97644 | 2.02177 | 1.97626 | 2.01951 |
| ***30000*** | 7.83694 | 7.73424 | 7.91525 | 7.97677 | 7.79242 | 7.70589 | 7.85170 | 7.70503 | 7.88108 |
| ***32768*** | 8.46770 | 8.49305 | 8.62631 | 8.74885 | 8.49749 | 8.40218 | 8.60858 | 8.42083 | 8.48593 |
| ***60000*** | 16.0575 | 16.08997 | 16.41482 | 16.56195 | 16.1626 | 15.9603 | 16.3936 | 15.9928 | 16.0627 |
| ***65536*** | 17.5835 | 17.63806 | 18.20529 | 18.11516 | 17.6139 | 17.4270 | 18.1570 | 17.4496 | 17.9501 |
| ***130000*** | 36.2751 | 36.43904 | 36.80316 | 37.69681 | 36.4052 | 36.1303 | 37.2412 | 36.0166 | 36.8912 |
| ***131072*** | 36.5164 | 36.73075 | 36.52090 | 38.35095 | 36.6299 | 37.0685 | 37.2622 | 36.2762 | 37.4379 |

From the knowledge of ***1.2 Time Complexity Of Merge Sort,*** we can tell that merge sort has O(nlog(n)) time complexity and we want to observe the same result for 9 different input type where the size the same. From the table, we can tell that this is true. Test results meet with the theory of merge sort.

***Compatibility Of Input Array Type for Merge Sort***

4.1=4.2=4.3=4.4=4.5=4.6=4.7=4.8=4.9

### *6.1.3) Quick Sort*

| ***Size/Input*** | ***4.1)*** | ***4.2)*** | ***4.3)*** | ***4.4)*** | ***4.5)*** | ***4.6)*** | ***4.7)*** | ***4.8*** | ***4.9)*** |
|---|---|---|---|---|---|---|---|---|---|
| ***100*** | 0.00794 | 0.00509 | 0.00501 | 0.00539 | 0.00537 | 0.00498 | 0.00819 | 0.00816 | 0.00657 |
| ***500*** | 0.03854 | 0.02991 | 0.02881 | 0.03232 | 0.03024 | 0.03462 | 0.04124 | 0.04305 | 0.03391 |
| ***1000*** | 0.08400 | 0.07094 | 0.06746 | 0.07461 | 0.07479 | 0.07940 | 0.07950 | 0.08381 | 0.07948 |
| ***1024*** | 0.08655 | 0.06969 | 0.07192 | 0.07672 | 0.07503 | 0.08255 | 0.08259 | 0.08442 | 0.08927 |
| ***4000*** | 0.36387 | 0.34714 | 0.34781 | 0.37066 | 0.35551 | 0.35808 | 0.33608 | 0.33056 | 0.36427 |
| ***4096*** | 0.36793 | 0.35667 | 0.35779 | 0.38137 | 0.36471 | 0.36472 | 0.35115 | 0.34341 | 0.35756 |
| ***8000*** | 0.77327 | 0.68775 | 0.69354 | 0.74073 | 0.70608 | 0.70278 | 0.71542 | 0.69669 | 0.77063 |
| ***8192*** | 0.79433 | 0.70381 | 0.70493 | 0.75396 | 0.72363 | 0.72172 | 0.73417 | 0.71658 | 0.78631 |
| ***30000*** | 3.17262 | 2.82537 | 2.82075 | 3.00504 | 2.86832 | 2.84789 | 2.91147 | 2.85382 | 2.99509 |
| ***32768*** | 3.45362 | 3.09075 | 3.10705 | 3.28569 | 3.13538 | 3.19752 | 3.24551 | 3.14629 | 3.46328 |
| ***60000*** | 6.75769 | 5.71263 | 5.74050 | 6.09378 | 5.85666 | 5.72695 | 6.13436 | 5.97489 | 6.11747 |
| ***65536*** | 7.57917 | 6.20532 | 6.24546 | 6.62625 | 6.33639 | 6.20808 | 6.82255 | 6.64433 | 6.75424 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ***130000*** | 15.09911 | 12.64825 | 12.69123 | 13.4377 | 12.8147 | 11.8732 | 14.1112 | 13.7867 | 13.4731 |
| ***131072*** | 15.00160 | 12.69777 | 12.76389 | 13.5412 | 12.9257 | 12.7361 | 14.3306 | 13.9289 | 15.1470 |

From the knowledge of ***1.3 Quick Sort Time Complexity*** Part, best case and average case has O(nlog(n)) time complexity and worst case could be $O(n^2)$ depends on the implementation, we have implemented quick sort with median pivot so we have fixed the worst case as O(nlog(n)) so we are waiting observe the almost same values for all results where time complexity is O(nlog(n)). From the table, we can certainly say that our theory has met with the emprical result.

***Compatibility Of Input Array Type for Quick Sort***

4.1=4.2=4.3=4.4=4.5=4.6=4.7=4.8=4.9

### *6.1.4) Heap Sort*

| ***Size/Input*** | ***4.1)*** | ***4.2)*** | ***4.3)*** | ***4.4)*** | ***4.5)*** | ***4.6)*** | ***4.7)*** | ***4.8)*** | ***4.9)*** |
|---|---|---|---|---|---|---|---|---|---|
| ***100*** | 0.00415 | 0.01861 | 0.01971 | 0.01980 | 0.01864 | 0.01975 | 0.01904 | 0.01737 | 0.02060 |
| ***500*** | 0.01700 | 0.11548 | 0.12147 | 0.12159 | 0.11588 | 0.12178 | 0.10942 | 0.10130 | 0.11403 |
| ***1000*** | 0.03330 | 0.25609 | 0.26477 | 0.26700 | 0.24977 | 0.26429 | 0.23472 | 0.21590 | 0.25146 |
| ***1024*** | 0.03417 | 0.26135 | 0.25824 | 0.26340 | 0.25775 | 0.26388 | 0.22764 | 0.22277 | 0.03497 |
| ***4000*** | 0.13233 | 1.19070 | 1.16465 | 1.18437 | 1.18238 | 1.17432 | 1.00786 | 1.00903 | 1.12286 |
| ***4096*** | 0.13477 | 1.20633 | 1.21577 | 1.23317 | 1.22009 | 1.21112 | 1.05571 | 1.02003 | 0.99666 |
| ***8000*** | 0.25126 | 2.52747 | 2.47892 | 2.52418 | 2.50293 | 2.50302 | 2.15433 | 2.14662 | 2.51082 |
| ***8192*** | 0.25808 | 2.58834 | 2.55921 | 2.61185 | 2.59454 | 2.58650 | 2.21658 | 2.19920 | 2.52050 |
| ***30000*** | 0.93993 | 10.14439 | 10.12184 | 10.3020 | 10.2175 | 9.65366 | 9.08544 | 9.03028 | 10.3518 |
| ***32768*** | 1.02545 | 11.10355 | 11.04802 | 10.9466 | 11.1364 | 10.5707 | 10.0238 | 9.96403 | 10.9120 |
| ***60000*** | 1.87623 | 21.02472 | 21.01433 | 20.3264 | 21.1938 | 19.8841 | 19.1654 | 19.19949 | 12.1111 |
| ***65536*** | 2.04886 | 22.99796 | 22.98540 | 21.9262 | 23.2181 | 23.1920 | 21.2380 | 21.18103 | 10.5243 |
| ***130000*** | 4.06640 | 47.83708 | 47.65854 | 48.6810 | 47.5626 | 48.0332 | 44.6291 | 44.33128 | 43.0792 |
| ***131072*** | 4.09752 | 48.00265 | 48.13041 | 49.0740 | 45.0843 | 48.4145 | 44.9595 | 43.11431 | 50.1660 |

Lets recall the knowledge we have learned in ***1.4 Time Complexity of Heap Sort,*** Heap Sort has O(nlog(n)) time complexity no mather what the input is. But from the implementation we have made we have succeed to change the time complexity for one input and this array is the constant number array. This come from the int max() function of Heap Sort where mentioned in ***2.4 Heap Sort.*** But all other cases heap sort has time complexity of O(n(log(n)) and this has met the empricial result that we have found.

***Compatibility Of Input Array Type for Heap Sort***

4.1>4.2=4.3=4.4=4.5=4.6=4.7=4.8=4.9

### *6.1.5) Counting Sort*

| *Size/Input* | *4.1)* | *4.2)* | *4.3)* | *4.4)* | *4.5)* | *4.6)* | *4.7)* | *4.8* | *4.9)* |
|---|---|---|---|---|---|---|---|---|---|
| ***100*** | 0.00299 | 0.01283 | 0.01043 | 0.01034 | 0.01023 | 0.01024 | 0.00289 | 0.00285 | 0.01164 |
| ***500*** | 0.00766 | 0.02304 | 0.02323 | 0.02287 | 0.02304 | 0.02284 | 0.00795 | 0.00797 | 0.02293 |
| ***1000*** | 0.01431 | 0.03563 | 0.03806 | 0.03519 | 0.03535 | 0.03536 | 0.01420 | 0.01420 | 0.03589 |
| ***1024*** | 0.01533 | 0.03631 | 0.03694 | 0.03622 | 0.03608 | 0.03520 | 0.01450 | 0.01452 | 0.01981 |
| ***4000*** | 0.05428 | 0.08635 | 0.08655 | 0.08472 | 0.08610 | 0.08655 | 0.05257 | 0.05263 | 0.06376 |
| ***4096*** | 0.05904 | 0.08794 | 0.08884 | 0.08747 | 0.08715 | 0.08603 | 0.05376 | 0.05386 | 0.08280 |
| ***8000*** | 0.10801 | 0.13954 | 0.13819 | 0.13787 | 0.13586 | 0.13703 | 0.10553 | 0.10435 | 0.12894 |
| ***8192*** | 0.11085 | 0.14050 | 0.14585 | 0.13797 | 0.16837 | 0.13973 | 0.10674 | 0.10805 | 0.11497 |
| ***30000*** | 0.40367 | 0.43343 | 0.42972 | 0.43177 | 0.58470 | 0.42517 | 0.39243 | 0.39672 | 0.43439 |
| ***32768*** | 0.47278 | 0.47331 | 0.46080 | 0.47324 | 0.48762 | 0.47752 | 0.42954 | 0.45573 | 0.46316 |
| ***60000*** | 0.78635 | 0.82045 | 0.84951 | 0.82145 | 0.82493 | 0.81519 | 0.78355 | 0.78758 | 0.81599 |
| ***65536*** | 0.88394 | 0.88706 | 0.89369 | 0.88635 | 0.88804 | 0.88327 | 0.88112 | 0.88018 | 0.81599 |
| ***130000*** | 1.70565 | 1.72044 | 1.74348 | 1.75569 | 1.72373 | 1.72245 | 1.69682 | 1.71912 | 1.72228 |
| ***131072*** | 1.74373 | 1.74659 | 1.75014 | 1.75645 | 1.73339 | 1.72844 | 1.73487 | 1.72238 | 1.76850 |

From the definition of Counting Sort where we made at ***1.5 Counting Sort,*** array is going once from start to end and while doing this, counts the same values that algorithm crossed and initialize them so we are waiting to observe the O(n+k) time complexity for all results for 9 different input type where their size are the same. If we look up the table, we can tell that our theorical calculation has met with the emprical results.

***Compatibility Of Input Array Type for Counting Sort***

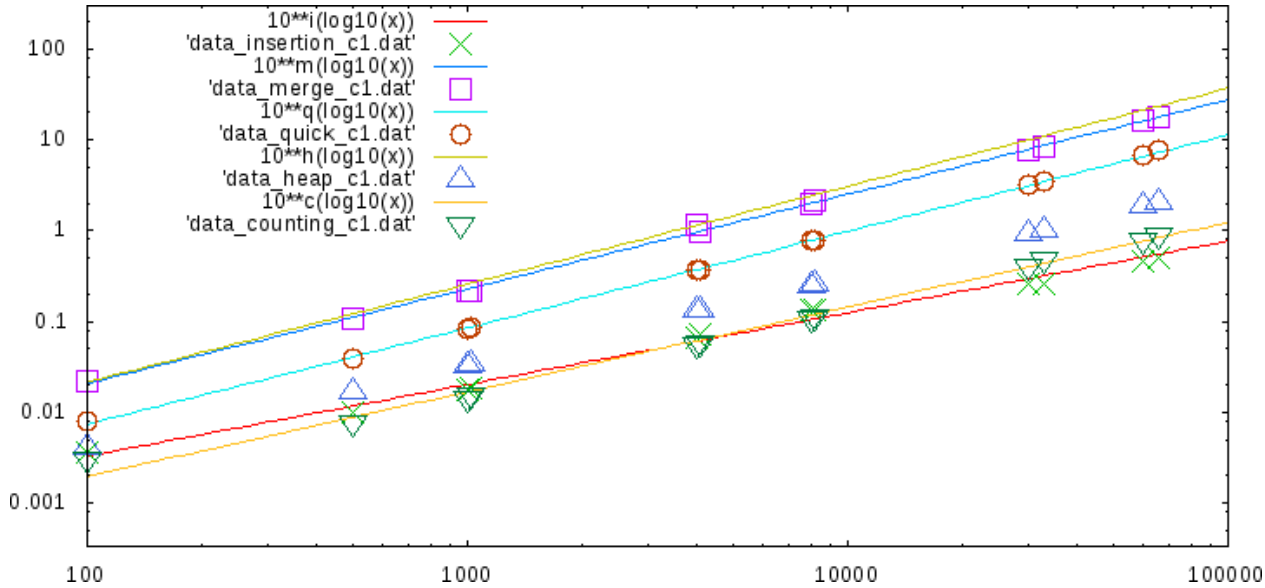
4.1=4.2=4.3=4.4=4.5=4.6=4.7=4.8=4.9

## *6.2) Graphs*

In this part of test we examined sorting algorithms' performance for each input cases to determine which algorithm are best and which are worst, depending on performance graphs.

Horisontal axis of charts are data sizes, with range of 100-100000. Vertical axis is execution times for sorting algorithms, in milliseconds, with range of 0,001ms – 100 ms. Each dat in the charts expresses the time taken by the sorting algorithm, in average of 10 repetition, for relevant input size. The dots are changing to different shapes and colors for each sorting algorithm. The lines are representing **nlogn** polinomial fitting of dots to show theorotical expectations. Thus lower lines are better. They also will be useful to compare theory(the line) to test outputs (the dots).

### *6.2.1) An Array Filled With Constant Number (_c1.dat)*



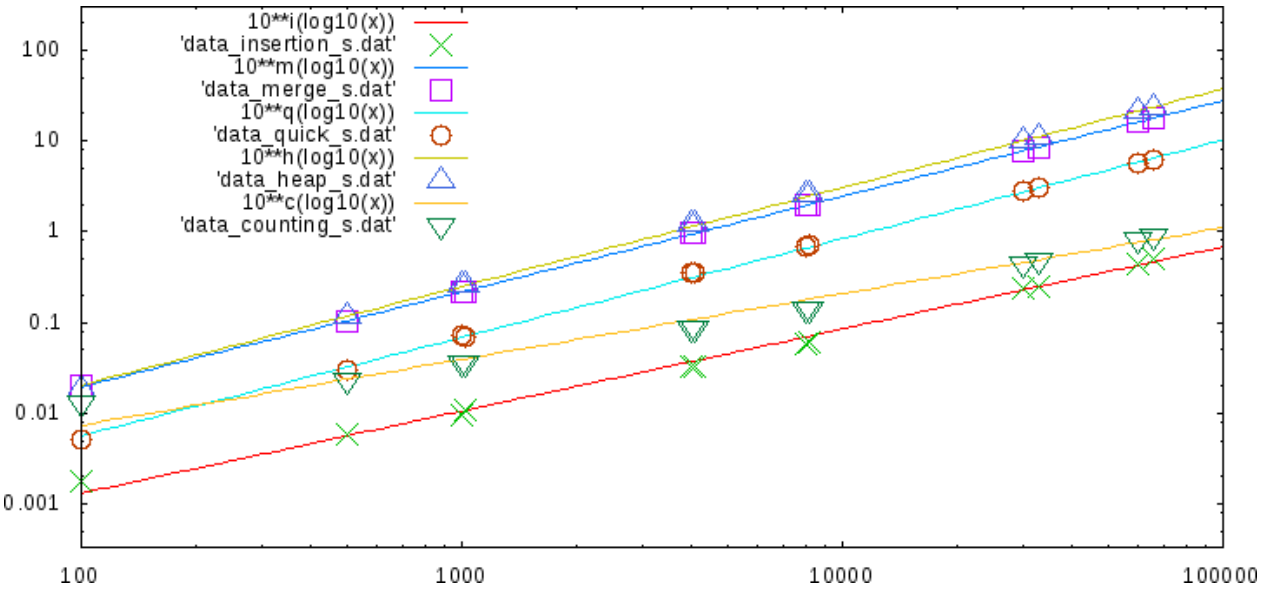

As seen in the chart, when the array is already sorted, insertion sort is tremendously faster than anything else because it can iterate faster over array with all same number despite its poor time compexity $O(n^2)$. Also counting sort can compete with instertion sort for this case since there's only 1 type of input element which is integer "1".

Slowest one is merge sort altough it has complexity of O(nlogn), because it tries to divide and merge the array anyway, without looking its content .

***Fastest to Slowest Algoritm For Constant Number Array:***

Insertion Sort > Counting Sort > Heap Sort > Quick Sort > Merge Sort

### *6.2.2) Sorted Array (_s.dat)*

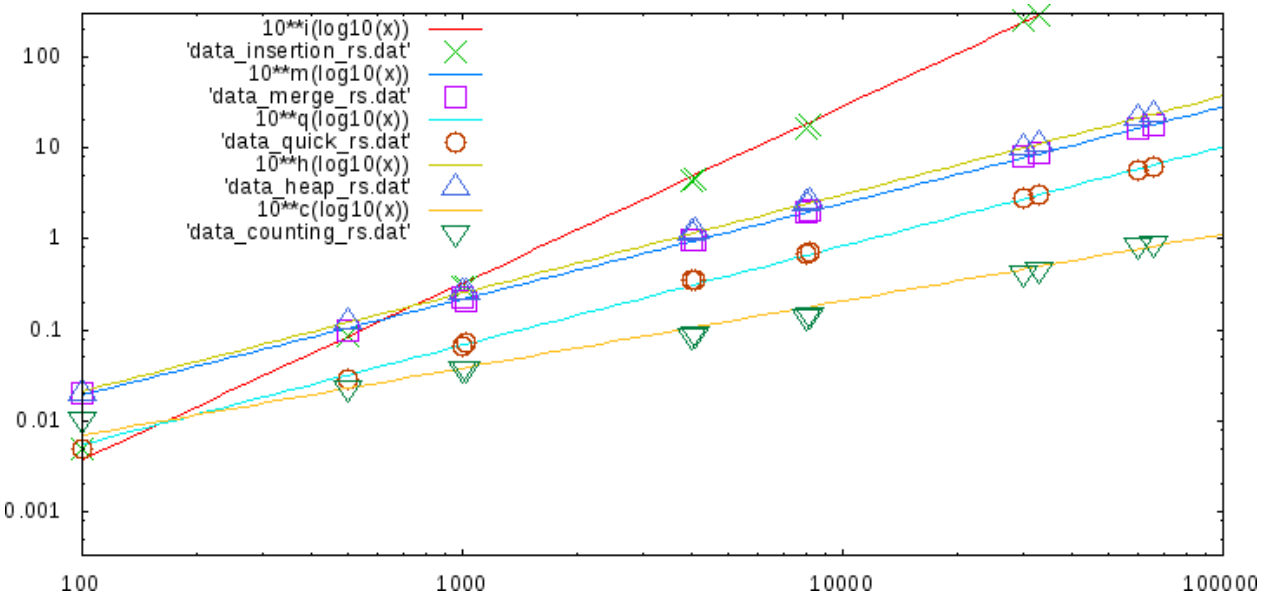


As we said before, when the array is already sorted, insertion sort is fastest algorithm. Also counting sort can catch it when size rises to >100.000. Slowest ones are merge and heap, since they are trying to implement their algorithm to sort an already sorted array.

***Fastest to Slowest Algoritm For Sorted Array:***

Insertion Sort > Counting Sort > Quick Sort > Merge Sort > Heap Sort

### *6.2.3) Reverse Sorted Array (_rs.dat)*

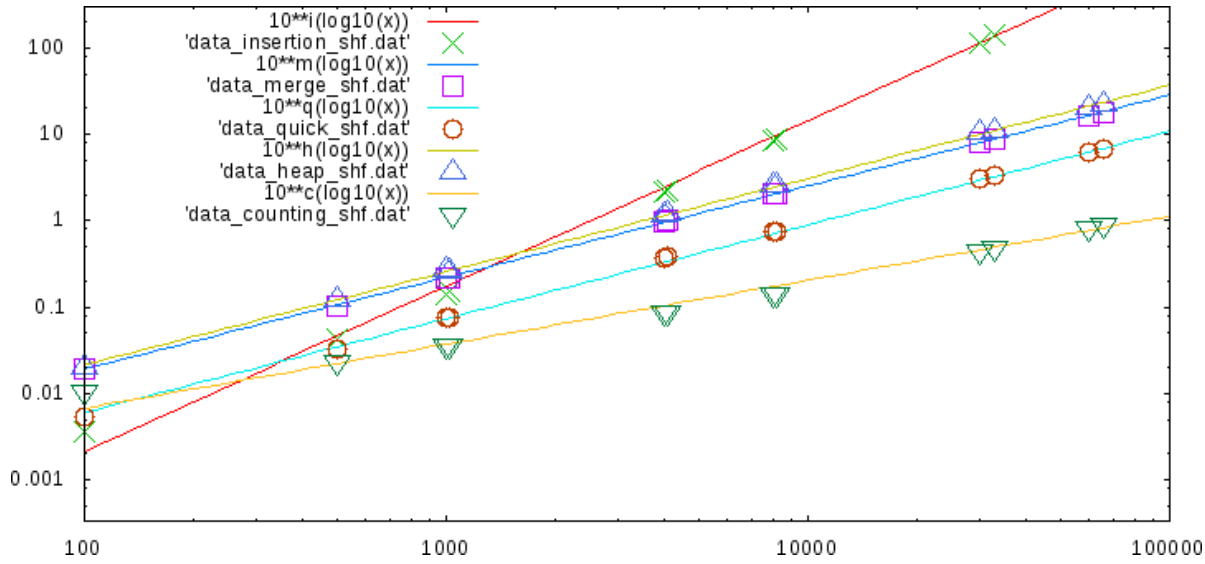



Insertion sort take a lot of time to sort large numbers so this sorting should be avoided. Counting sort shines among five sorting algorithms with its time complexity of O(n+k). When it comes to three logarithmic algorithms(merge, heap and quick), quick sort does the best work here, despite reverse sorting is accepted as it's worst case scenerio. Because we used median as pivot element in our implemention of quick sort, instead of last or first element.

***Fastest to Slowest Algoritm For Reverse Sorted Array:***

Counting Sort > Quick Sort > Merge Sort > Heap Sort > Insertion Sort

### *6.2.4) Unsorted Array (_shf.dat)*

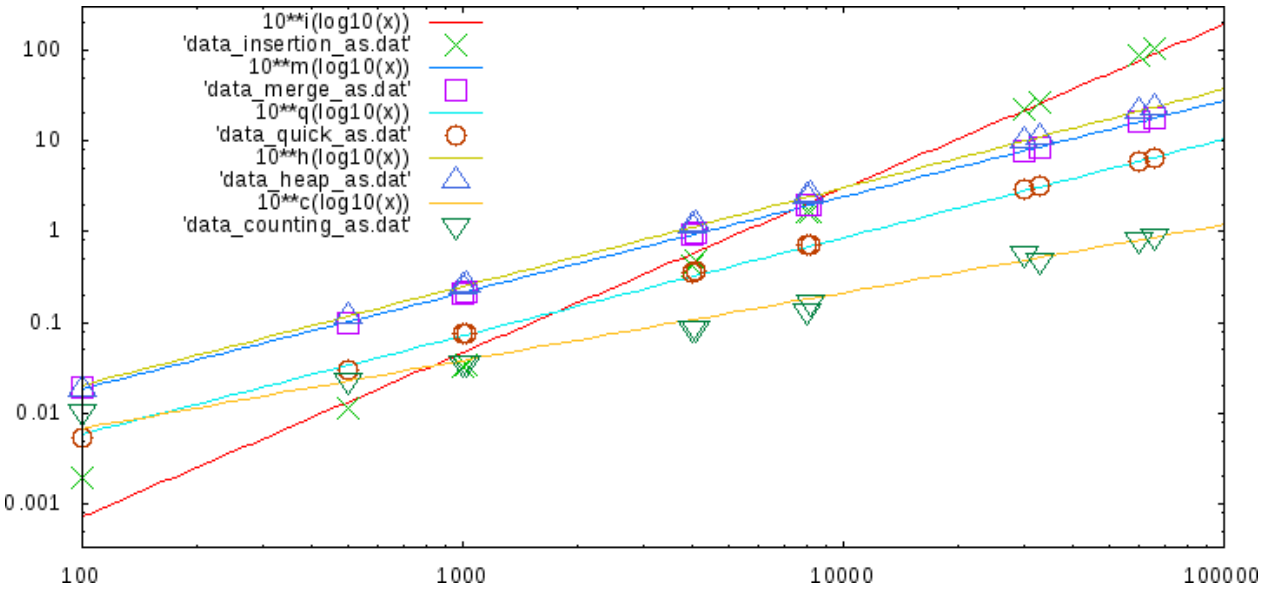



Here, we can do main comparison since this case is most common situation. For lower sizes (<500) we can see that insortion sort does best work, and when size increases counting sort takes the throne. Amongst other three algorithm quick sort comes with second position and merge and heap shares third position with almost same performance as expected from theory.

***Fastest to Slowest Algoritm For Unsorted Array:***

Counting Sort > Quick Sort > Merge Sort > Heap Sort > Insertion Sort

### *6.2.5) Almost Sorted Array With At Most %15 Unsorted Part (_as.dat)*



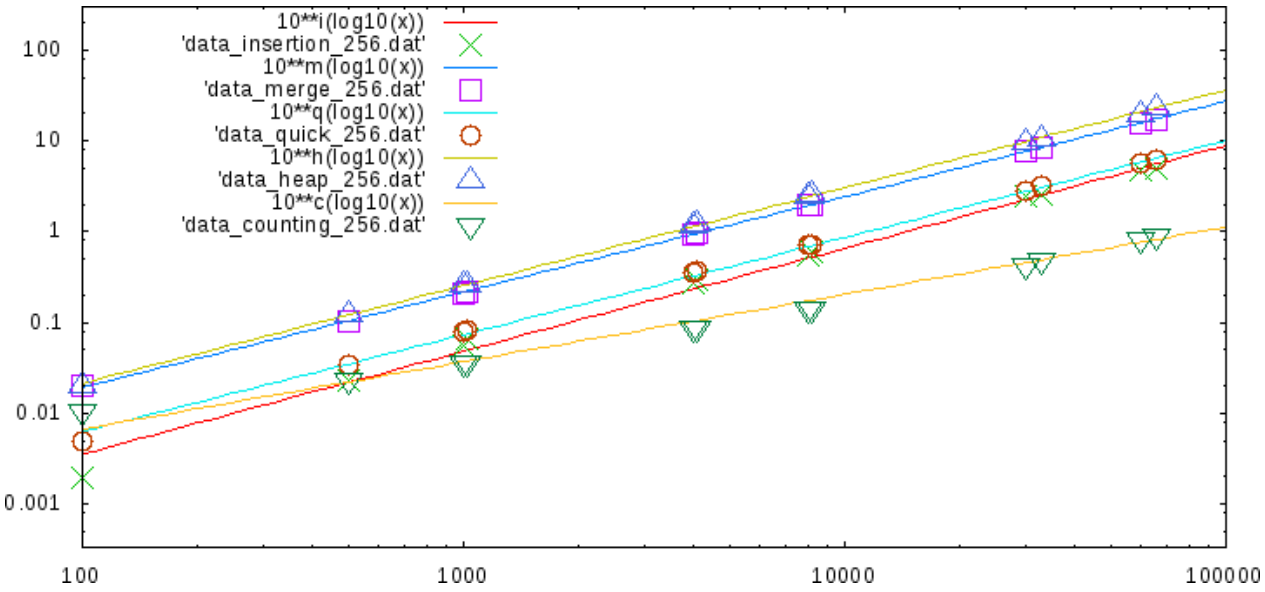
In almost sorted case, all algorithms, except insertion sort, perform same with unsorted array case. For insertion sort we can say that it keeps the gap short until size of 10.000 since input array is %85 sorted, then falls off.

We inspected this case, if there'd be any pathalogical circumstance for especially for quick sort but it performs as well as the previous case (unsorted array).

***Fastest to Slowest Algoritm For Almot Sorted Array:***

Counting Sort > Quick Sort > Merge Sort > Heap Sort > Insertion Sort

### *6.2.6) Sorted Array With Last 256 Elements Are Unsorted (_256.dat)*




In this case we encounter unexpected results for insertion sort. Altough this case has almost logic as the previous case, since they both are sorted arrays with little unsorted part on their end, insertion sort behaves differently. It can compete with quick even for bigger sizes!

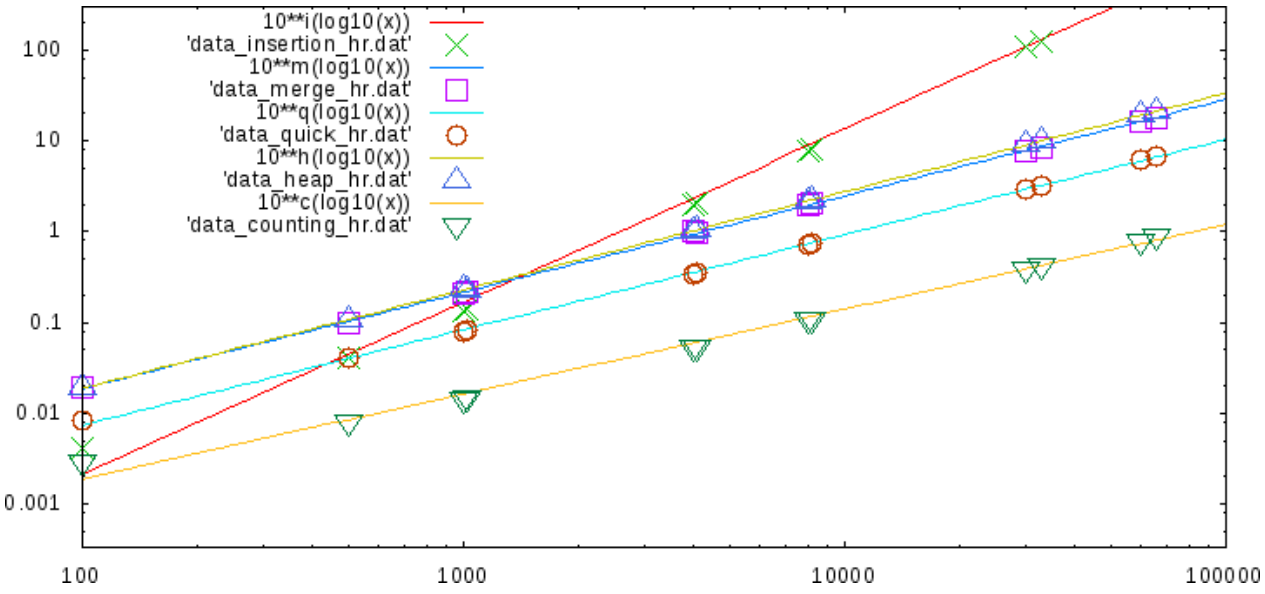
Actually we expected that insertion sort performs better than this, because 256 elements represent only tiny part of bigger sizes(>10.000) but apperantly, it still effects insertion sort's performance.

We designed this case to see if merge sort behaves different than case of sorted array or not but it performs same.

***Fastest to Slowest Algoritm For Last 256 Is Unsorted Array:***

Counting Sort > Insertion Sort > Quick Sort > Merge Sort > Heap Sort

### *6.2.7) Unsorted Array With High Repetition of Numbers (_hr.dat)*

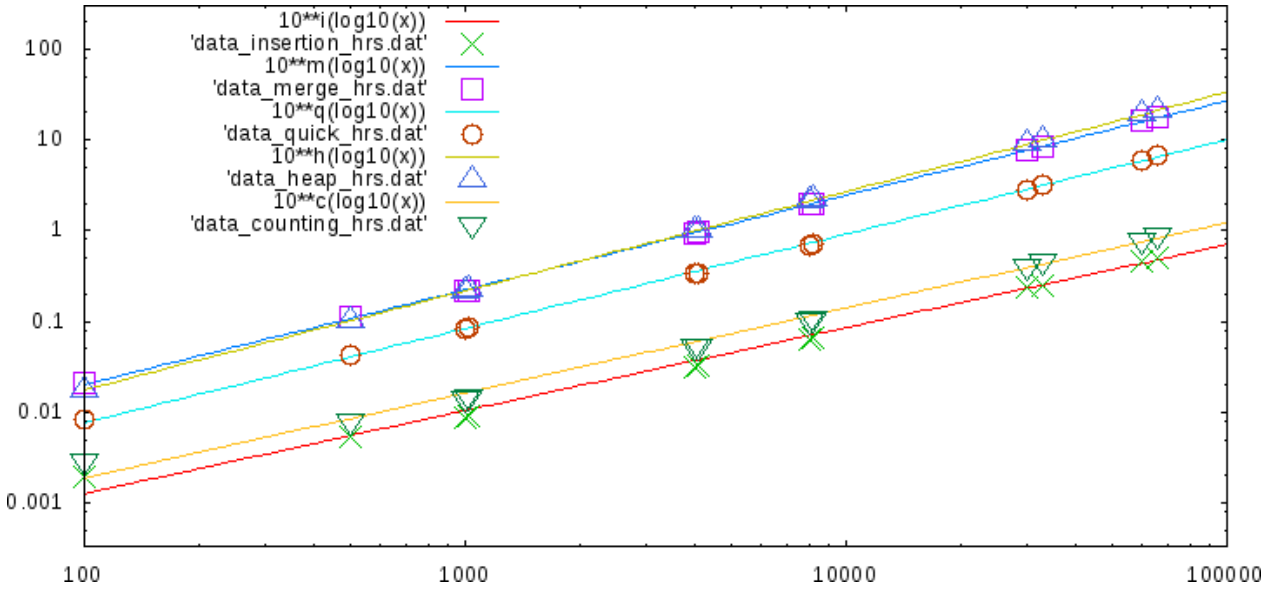



In this case, if we examine low size arrays (<500) we can easily see that counting sort does its best work here. Normally we expect that insertion sort is best algorithm around for lower sizes but counting sort take its crown here even for input size of 100. Because the input arrays consist of highly repetitive numbers (only ten different integers for our case).

***Fastest to Slowest Algoritm For High Repetitin Array:***

Counting Sort > Quick Sort > Merge Sort > Heap Sort > Insertion Sort

### *6.2.8) Sorted Array With High Repetition of Numbers (_hrs.dat)*

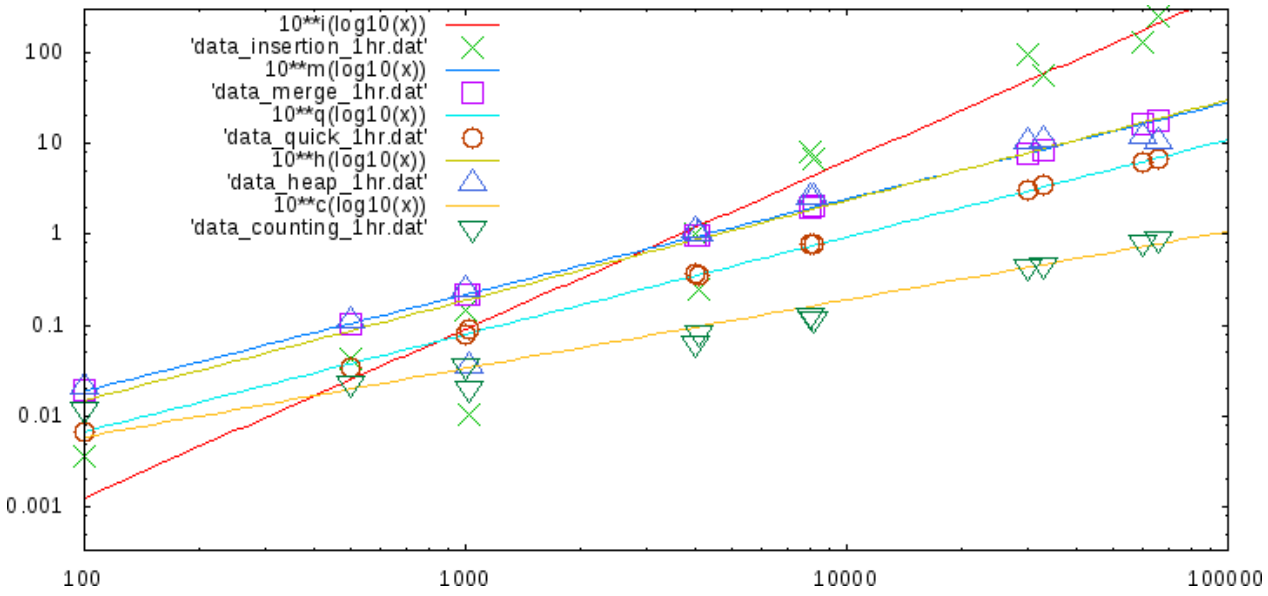


Comparing to previous case, we can see that insertion sort takes its throne back since we are using sorted arrays this time.

***Fastest to Slowest Algoritm For High Repetitin Array Sorted:***

Insertion Sort > Counting Sort > Quick Sort > Merge Sort > Heap Sort

### *6.2.9) Unsorted Array With Only One High Repetative Number*




We desinged this case to show if any algorith would behave anormally and we can see that insertion sort does it. It shows wavy and inconsistent performance while others are same with case of unsorted array.

***Fastest to Slowest Algoritm For One Element Repetitin Array:***

Counting Sort > Quick Sort > Merge Sort > Heap Sort > Insertion Sort

# *CONCLUSION*

In this experiment we have made six thousand and three hundred calculation to find the best empiricial result to compare with the theory of the algorithms. In the end, empricial results show us, if we have extra memory to use counting sort; the best sort algorithm except sorted arrays is **Counting Sort** with time complexity O(n+k). If we have sorted array and we want to sort again the best is **Insertion Sort** with time complexity O(n) otherwise **Insertion Sort** is the worst. The fastest second algorithm has become **Quick Sort** which has O(n(log(n)) time comlexity with the small change, we have made on pivot selection. Because it is faster to compare with pivot than merging. The third place has gone to **Merge Sort** with O(n(log(n)) time complexity because merge operation is faster than heapify operation of heap sort. So, the fourth one become **Heap Sort** with time complexity O(n(log(n)).

Finally, we can also tell, **Quick Sort, Merge Sort** and **Heap Sort** has O(n(log(n)) but **Quick Sort** has a less coeffient then **Merge Sort** and **Merge Sort** has less coeffient than **Heap Sort**. Thus, **Quick Sort** is faster than **Merge Sort** while **Merge Sort** is faster than **Heap Sort**.