Hacettepe University

Computer Engineering Department

BBM204 Software Practicum II - 2024 Spring

Programming Assignment 1

March 22, 2024

Student name:
Salih Eren Yüzbaşıoğlu

Student Number:
b2220356040

1 Problem Definition

In computer science, sorting and searching algorithms are very useful and intertwined. These 2 algorithms serve as the backbone for countless applications across the discipline, from databases to the way internet works to many other parts of the discipline.

As computers got faster and bigger we used them for harder and bigger tasks. Unfortunately for tasks with higher complexity than linear we can't use this speed increase efficiently. Since as speed increases 10 times complexity of algorithm increases more than 10 times. So getting them as close as possible to linear time complexity has been the holy grail of computer science and topic of research for many decades.

In this analysis report as many did before us, we will look at some widely used sort and search algorithms and compare them.

2 Sorting Algorithms

In order to search fast, data needs to be sorted. So we will start with sort algorithms. In this report we will analyse 3 sorting algorithms namely insertion sort, merge sort and counting sort.

For the sake not using any extra memory I implemented all sort methods as void. So they are not returning a new array but instead changing the initial one.

2.1 Insertion Sort

Code	Unit Cost	Times
`public static void doInsertionSort(int[] arr) {`		
`  for (int i = 1; i < arr.length; i++) {`	c_1	n
`    int key = arr[i];`	c_2	$n-1$
`    int j = i - 1;`	c_3	$n-1$
`    while (j >= 0 && arr[j] > key) {`	c_4	$\sum_{i=1}^{n-1} t_i$
`      arr[j + 1] = arr[j];`	c_5	$\sum_{i=1}^{n-1} (t_i - 1)$
`      j--;`	c_6	$\sum_{i=1}^{n-1} (t_i - 1)$
`    }`		
`    arr[j + 1] = key;`	c_7	$n-1$
`  }`		
`}`		

Table 1: Unit cost and times for insertion sort algorithm.

Looking at the Table we realize there are variables that are not based on n but instead t_i. These values depend on whether array is sorted, reverse sorted or half sorted. We will analyse both cases.

For sorted data t_i will always be 1 so total cost of the algorithm is

$$c_1 \cdot n + (c_2 + c_3 + c_7 + c_4) \cdot (n-1)$$

since above expression is proportional to n, complexity is denoted as $\mathcal{O}$(n).

And for reverse data t_i will be equal to i meaning the expression $\sum_{i=1}^{n-1}(t_i - 1)$ will be equivalent to $\sum_{i=1}^{n-1}(i-1)$ hence if we calculate the total time we get

$$c_1 \cdot n + (c_2 + c_3 + c_7) \cdot (n-1) + c_4 \cdot \sum_{i=1}^{n-1}(i) + (c_5 + c_6) \cdot \sum_{i=1}^{n-1}(i-1)$$

if we simplify $\sum$ expressions we get

$$c_1 \cdot n + (c_2 + c_3 + c_7) \cdot (n-1) + \frac{c_4 \cdot (n-1) \cdot n}{2} + \frac{c_5 \cdot c_6 \cdot (n-2) \cdot (n-1)}{2}$$

since above expression is proportional to n^2, complexity is denoted as $\mathcal{O}(n^2)$.

For sorted data we found complexity to be $\mathcal{O}(n)$ and for reverse sorted we found it $\mathcal{O}(n^2)$. For random data even though we can't exactly calculate complexity due to t_i, it would be rational think t_i would be proportional to i hence giving us a expression which is multiple of n^2. Even though n^2 complexity scales really badly. For data that is mostly sorted it performs linear. This algorithms is also in-place meaning it does not use any extra memory.

Example usage of this algorithm is introsort which does quicksort and heapsort until number of elements are less than 16 then does insertion sort. Introsort is used in standard library of C++.

2.2 Merge Sort

Below code might be little different than the given pseudo code but in terms of theoretical time and space complexity it is basically same.

```
1   private static void doMerge(int[] arr, int l, int m, int r) {
2       int n1 = m - l + 1, n2 = r - m;;
3       int L[] = new int[n1];
4       int R[] = new int[n2];
5       for (int i = 0; i < n1; ++i) {
6           L[i] = arr[l + i];
7       }
8       for (int j = 0; j < n2; ++j) {
9           R[j] = arr[m + 1 + j];
10      }
11      int i = 0, j = 0, k = l;
12      while (i < n1 && j < n2) {
13          if (L[i] <= R[j]) {
14              arr[k++] = L[i++];
15              continue;
16          }
17          arr[k++] = R[j++];
18      }
19      while (i < n1) { arr[k++] = L[i++]; }
20      while (j < n2) { arr[k++] = R[j++]; }
21  }
22  public static void doMergeSort(int[] arr, int l, int r) {
23      if (l < r) {
24          int m = (l + r) / 2;
25          doMergeSort(arr, l, m);
26          doMergeSort(arr, m + 1, r);
27          doMerge(arr, l, m, r);
28      }
29  }
```

Since merge sort involves recursion it is hard to analyse it line by line similar to what we did in insertion sort. Instead we will analyse it in higher level. Starting with helper function doMerge.

The doMerge() function has two for loops that each run n1 and n2 times, respectively, where n1 = m - l + 1 and n2 = r - m. Both n1 and n2 are roughly n/2 if n is the number of elements in the segment of the array being merged (since m is roughly the midpoint). So, each loop has a complexity of $\mathcal{O}$(n/2) which simplifies to $\mathcal{O}$(n). Then, there's a while-loop that runs at most n1 + n2 times, which is essentially n times. Inside this loop, the operations are constant time, so this part is also $\mathcal{O}$(n). The remaining two while-loops each run at most n1 or n2 times, adding up to $\mathcal{O}$(n). So, the time complexity for doMerge() is $\mathcal{O}$(n), where n is the number of elements in the segment of the array it's working on.

The doMergeSort() function is a recursive function that splits the array into halves until it has segments of length 1. This splitting results in a tree-like structure where the depth of the

tree is $\mathcal{O}$(log n). At each level of recursion, doMerge() is called which we've established is $\mathcal{O}$(n). Combining these, the overall time complexity of comes at $\mathcal{O}(n \cdot \log n)$ Merge Sort is known for it's stability as it performs at $\mathcal{O}$(n$\cdot$ logn) in every case not depending on the way input data was given. Worst part of merge is that it used memory proportional to n. Making at bad when it comes to sorting large amount of data.

2.3 Counting Sort

Code	Unit Cost	Times
`public static void doCountingSort(int[] arr) {`		
`  int k = 0;`	c_1	1
`  for (int i = 0; i < arr.length; i++) {`	c_2	$n+1$
`    k = Math.max(k, arr[i]);`	c_3	n
`  }`		
`  int[] count = new int[k + 1];`	c_4	1
`  int n = arr.length;`	c_5	1
`  for (int i = 0; i <= k; ++i) {`	c_6	$k+2$
`    count[i] = 0;`	c_7	$k+1$
`  }`		
`  for (int i = 0; i < n; ++i) {`	c_8	$n+1$
`    count[arr[i]]++;`	c_9	n
`  }`		
`  for (int i = 1; i <= k; ++i) {`	c_{11}	$k+1$
`    count[i] += count[i - 1];`	c_{12}	k
`  }`		
`  int[] output = new int[n];`	c_{13}	1
`  for (int i = n - 1; i >= 0; i--) {`	c_{14}	$n+1$
`    output[count[arr[i]] - 1] = arr[i];`	c_{15}	n
`    count[arr[i]]--;`	c_{16}	n
`  }`		
`  for (int i = 0; i < n; ++i) {`	c_{17}	$n+1$
`    arr[i] = output[i];`	c_{18}	n
`  }`		
`}`		

Table 2: Unit cost and times for counting sort algorithm.

Total cost of the algorithm is

$$c_1 + c_2 \cdot (n+1) + c_3 \cdot n + c_4 + c_5 + c_6 \cdot (k+2) + c_7 \cdot (k+1) +$$
$$c_8 \cdot (n+1) + c_9 \cdot n + c_{11} \cdot (k+1) + c_{12} \cdot k + c_{13} + c_{14} \cdot (n+1) +$$
$$c_{15} \cdot n + c_{16} \cdot n + c_{17} \cdot (n+1) + c_{18} \cdot n$$

If we simplify the equation we get

$$(c_2 + c_3 + c_8 + c_9 + c_{14} + c_{15} + c_{16} + c_{17} + c_{18}) \cdot n + (c_6 + c_7 + c_{11} + c_{12}) \cdot k$$

$$(c_1 + c_4 + c_5 + c_{13}) +$$

Since this is a multiple of both n and k time complexity is $\mathcal{O}$(n+k).

Counting sort can perform with $\mathcal{O}$(n) if maximum of the array is smaller than n, which makes it ideal if input data involves lots of duplicate values. Bad side of this sorts is that it only works with discrete values and not with continuous values also it uses memory proportional to both n and k.

3 Searching Algorithms

Unlike sorting algorithms the world of searching algorithms is not wide. Since there is no magic to searching on random data other than linearly searching every data and for sorted data bottleneck of algorithm is not search but rather sorting. Also while we were trying to reach as close as to $\mathcal{O}$(n) time complexity with sorting, most basic and known sorted search algorithm binary search already has time complexity of $\mathcal{O}(\log n)$ which basically means for any data we can fit into a computer finding a value from it can happen instantaneously.

So in this part we will look at 2 search algorithms namely linear search and binary search.

3.1 Linear Search

Code	Unit Cost	Times
`public static int doLinearSearch(int[] arr, int x) {`		
`  for (int i = 0; i < arr.length; i++) {`	c_1	$n+1$
`    if (arr[i] == x) {`	c_2	n
`      return i;`	c_3	n
`    }`		
`  }`		
`  return -1;`	c_4	1
`}`		

Table 3: Unit cost and times for linear search algorithm.

Finding the total cost

$$c_1 \cdot (n+1) + c_2 \cdot n + c_3 \cdot (n) + c_4$$

Looking at above expression it is clear that linear search works in $\mathcal{O}(n)$ time complexity and does not use any extra memory. Linear search can be useful if there is no way to sort the data or if the search algorithms will be used only few times.

3.2 Binary Search

```
30      public static int doBinarySearch(int[] arr,int x){
31          int l = 0, r = arr.length - 1;
32          while (l <= r) {
33              int m = l + (r - l) / 2;
34              if (arr[m] == x) {
35                  return m;
36              }
37              if (arr[m] < x) {
38                  l = m + 1;
39              } else {
40                  r = m - 1;
41              }
42          }
43          return -1;
44      }
```

Since binary search involves not so clear loops it would be easier to analyse it not using unit costs but higher level analysis.

The main loop runs as long as $l <= r$. At each step, the range $[l, r]$ is approximately halved, leading to a total of $\mathcal{O}(\log n)$ iterations, where n is the number of elements in the array. This is because, with each iteration, the search interval is divided by 2.

Calculating the middle index m and comparing arr[m] with x are constant-time operations within the loop. Updating the variables l or r based on the comparison results are also constant-time operations. Since each iteration of the loop involves only constant time operations and the loop runs $\mathcal{O}(\log n)$ times, the overall time complexity is $\mathcal{O}(\log n))$.

Search also does not use any extra memory.

4 Results, Analysis, Discussion

We have shown the code of every sort and search algorithm and we also calculated their theoretical time and space complexity. Now it is time to see whether or not these values also comply with real world data.

We have downloaded a data with over 250 thousand values and compared these algorithms. Important to note that below values are rounded meaning executions with 0 milliseconds did not happen instantaneously but rather was under 1 millisecond.

Table 4: Results of the running time tests performed for varying input sizes (in ms).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
	Random Input Data Timing Results in ms									
Insertion sort	0	0	0	1	5	19	73	292	1202	4858
Merge sort	0	0	0	1	1	2	3	6	13	31
Counting sort	239	139	144	139	138	148	140	144	164	143
	Sorted Input Data Timing Results in ms									
Insertion sort	0	0	0	0	0	0	0	0	0	0
Merge sort	0	0	0	0	0	1	2	3	6	17
Counting sort	0	0	0	0	0	0	0	0	1	148
	Reversely Sorted Input Data Timing Results in ms									
Insertion sort	0	0	1	3	10	35	145	624	2334	8465
Merge sort	0	0	0	0	0	1	2	4	12	16
Counting sort	146	140	158	136	138	135	154	142	150	168

Table 5: Results of the running time tests of search algorithms of varying sizes (in ns).

	Input Size n									
Algorithm	**500**	**1000**	**2000**	**4000**	**8000**	**16000**	**32000**	**64000**	**128000**	**250000**
Linear search (random data)	628	1396	259	415	760	1238	2989	4639	12553	26756
Linear search (sorted data)	142	278	258	440	573	1008	2925	6723	16864	42285
Binary search (sorted data)	398	243	243	219	194	349	254	210	267	680

Table 6: Computational complexity comparison of the given algorithms.

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Insertion sort	$\Omega(n)$	$\Theta(n^2)$	$\mathcal{O}(n^2)$
Merge sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$\mathcal{O}(n \log n)$
Counting Sort	$\Omega(k+n)$	$\Theta(k+n)$	$\mathcal{O}(k+n)$
Linear Search	$\Omega(1)$	$\Theta(n)$	$\mathcal{O}(n)$
Binary Search	$\Omega(1)$	$\Theta(\log n)$	$\mathcal{O}(\log n)$

Table 7: Auxiliary space complexity of the given algorithms.

Algorithm	**Auxiliary Space Complexity**
Insertion sort	$\mathcal{O}(1)$
Merge sort	$\mathcal{O}(n)$
Counting sort	$\mathcal{O}(n+k)$
Linear Search	$\mathcal{O}(1)$
Binary Search	$\mathcal{O}(1)$

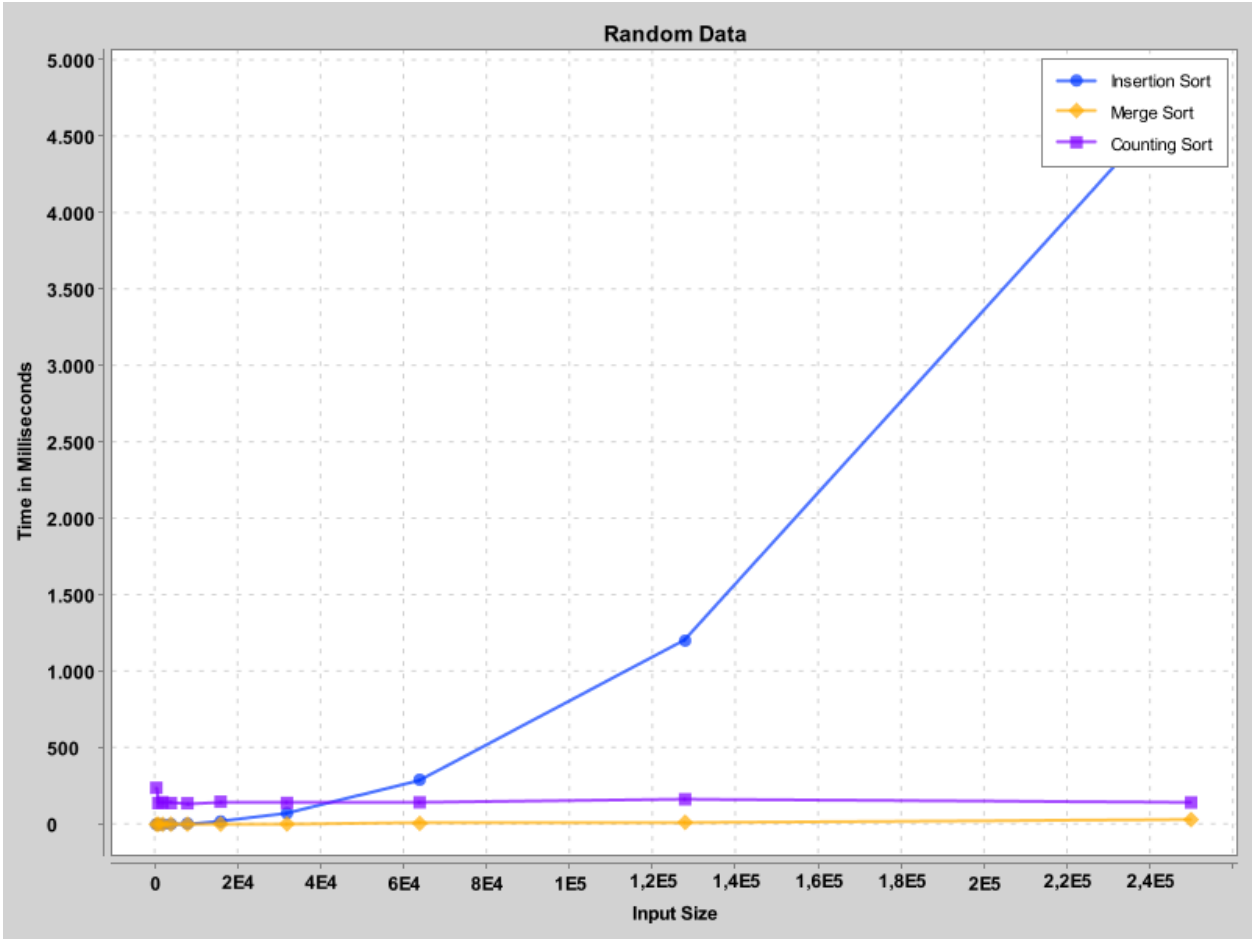

Figure 1: Plot of the sort algorithms on random data.

It is no surprise that merge sort executed with such speed. It might be confusing that counting sort is much slower than merge sort but since counting sort is also dependant on maximum value of array it makes a big difference. In this case maximum of the whole array is 119999780 which

is lot bigger than 250000. The ınsertion sort might look too slow compared to others but if you calculate the difference between n^2 and $n \cot \log n$ when n is equal to 250000. You get about 40 thousand increase in complexity which explains the time difference.

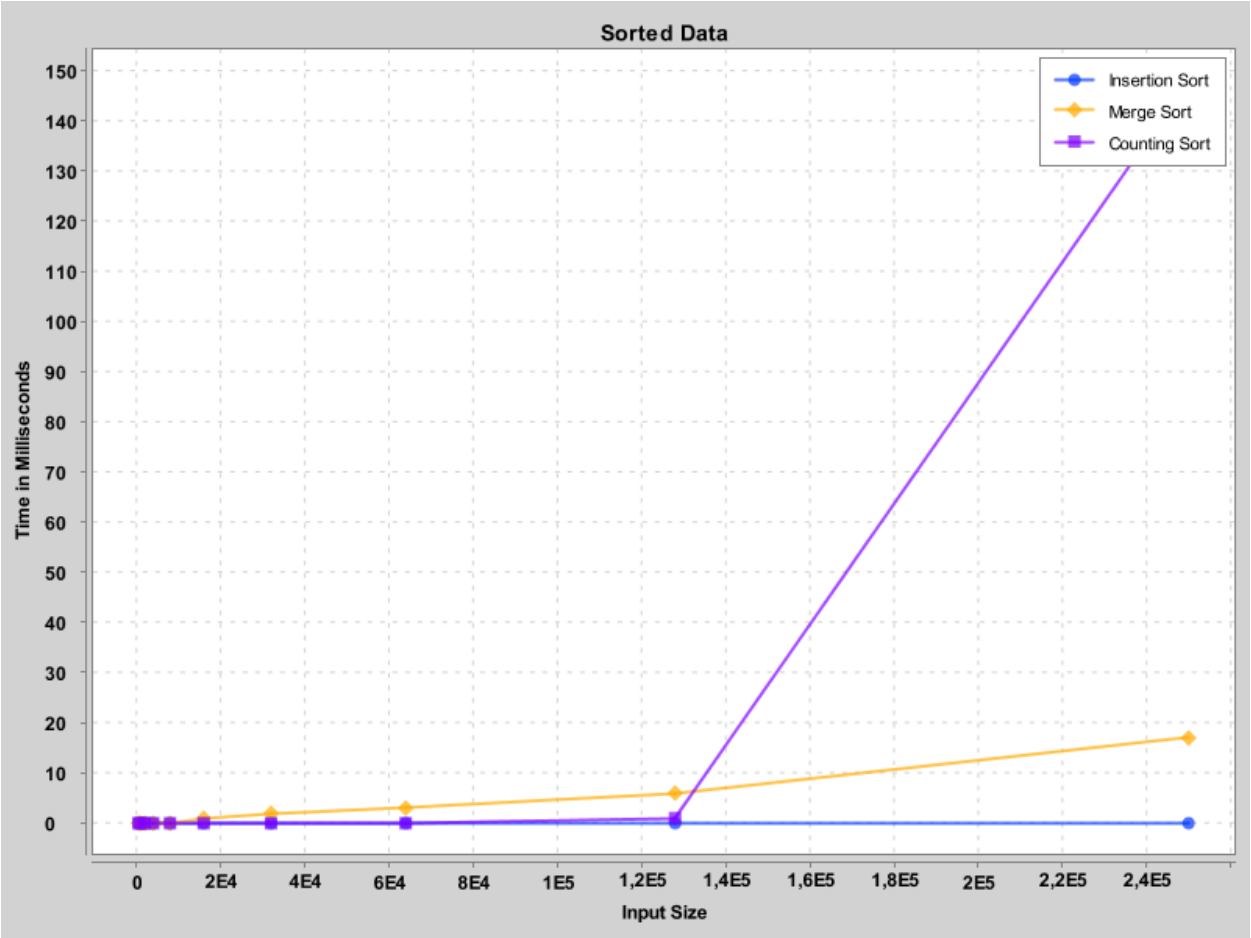

Figure 2: Plot of the sort algorithms on sorted data.

Since in sorted data insertion sort works in $\mathcal{O}(n)$ it works the fastest just behind it we see merge sort with a line whose slope is increasing due to $\mathcal{O}(n \log n)$. It might be confusing that counting sort performed similar to O(n) until the half than suddenly increased, but that is due to maximum of the array increasing in sudden fashion between 128k and 250k. Since maximum of array when n is 128k is 241248 but when it is 250k it is 119821739 explaining the sudden increase.

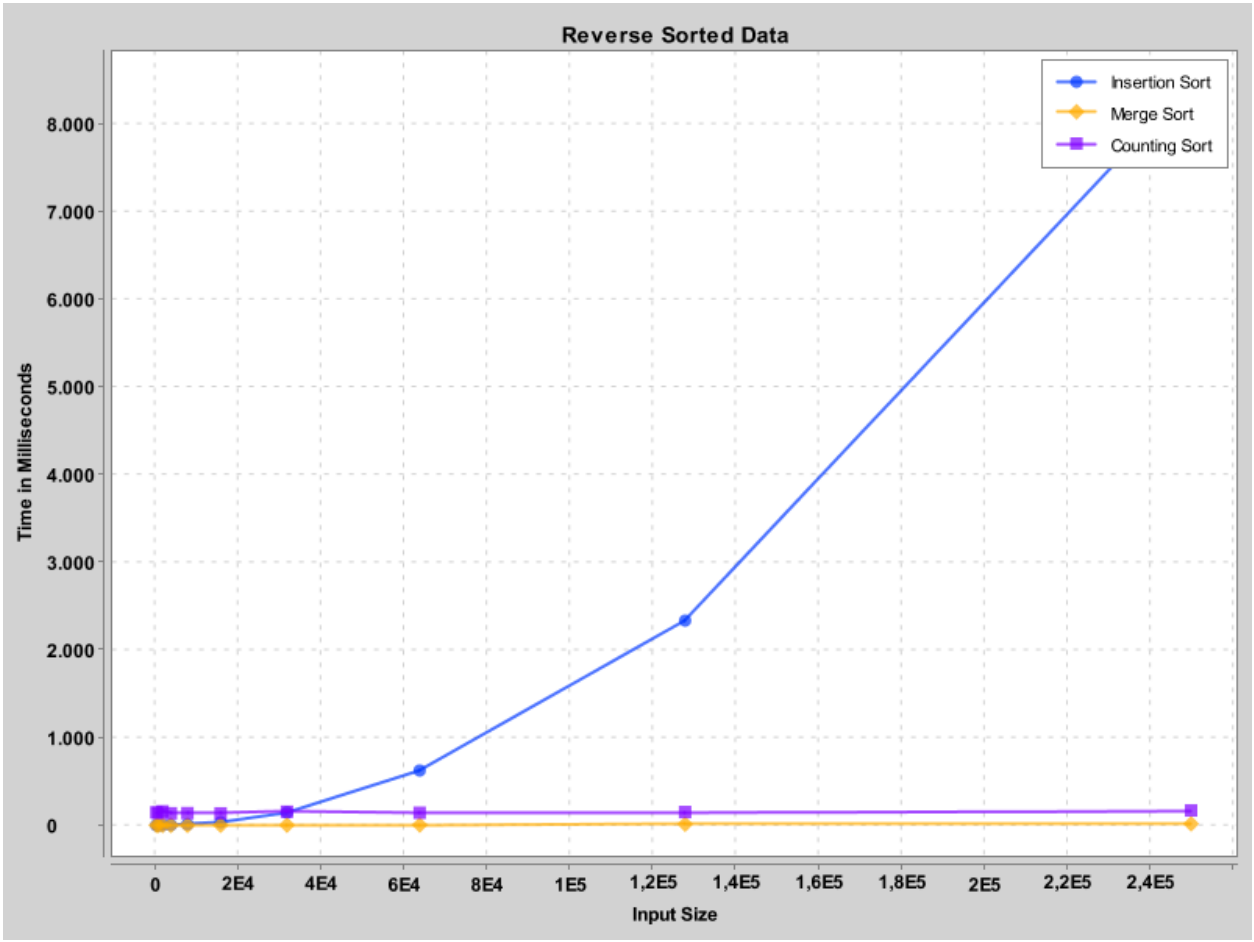

Figure 3: Plot of the sort algorithms on reverse sorted data.

In above figure we see a very similar plot compared to one on random data expect that insertion sort is little bit slower. Which is due to coefficient of complexity between reverse sorted data and random data.

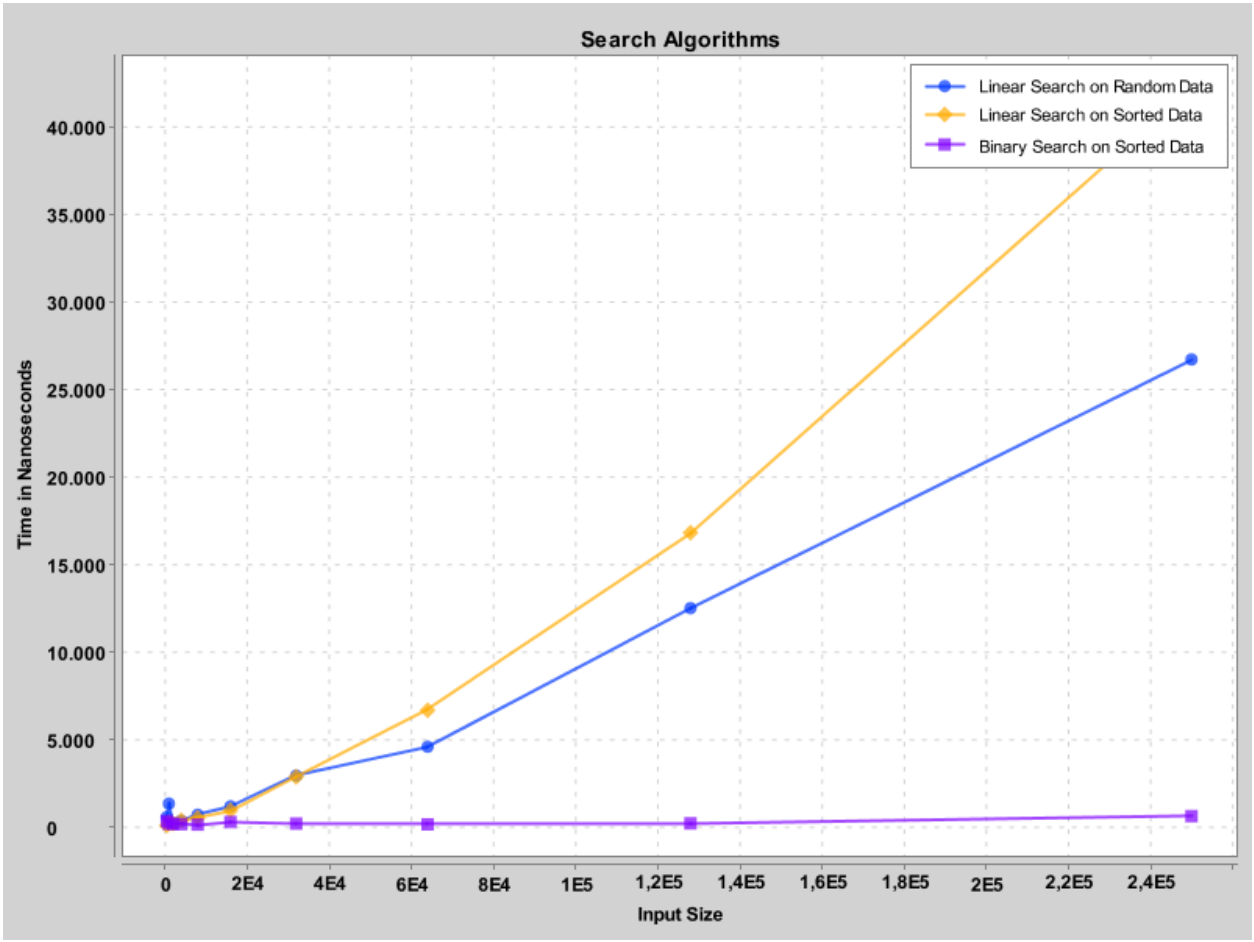

Figure 4: Plot of the search algorithms.

We see in above figure binary search is fastest as it works in $\mathcal{O}(\log n)$. For the linear search it increases in linear fashion as expected. It might be confusing that linear search on sorted data is slower but that is due to duplicate values. Since in sorted data duplicate values are in clusters it does not make it easier to search for them but for random data they are spread making it shorter to find one of them.

5 Notes

Based on everything that we did on this analysis here are insights to every algorithms we looked at.

Insertion Sort: This sorting method works well if the data is almost sorted or if there are not many items to sort. It's fast in these cases but becomes slow when the data is in reverse order or completely random.

Merge Sort: This method sorts any list of items at a consistent speed, no matter how they're arranged to start with. It uses extra memory, which might not be ideal if there's not much memory available.

Counting Sort: This sort is good for sorting numbers when you know the range of values you have. It's quick if there are many repeated numbers but can use a lot of memory if the range of numbers is large.

Linear Search vs. Binary Search: Linear search goes through each item one by one, which is okay for small lists but not for big ones. Binary search is much faster but only works if the list is already sorted.

References

- Local large language model Mixtral-7b
- Wikipedia.com
- https://devdocs.io/cpp/