Sorting algorithm performance comparison

March 31, 2024

Author : Tasmir Hossain Zihad (2107071)

1 Sorting Algorithm Performace Analysis

1.1 Sorting Algorithms

The attached files are contain extensive comments, the snippets added here do not

1.1.1 Bubble Sort

Given an array, the largest element is moved to the end. * The inner loop swap two adjacent elements, if left one is larger than right one. thus moving the largest one to right. * The outer loop skips the right sorted side. * If an array is already sorted it avoids resorting

[ ]:
```cpp
template <class T>
void bubble_sort(vector<T>& array) {
  for (int i = array.size(); i >= 0; i--) {
    bool swapped = false; // sorted array check
    for (int j = 1; j < i; j++)
      if (array[j - 1] > array[j]) {
        swap(array[j - 1], array[j]);
        swapped = true;
      }
    if (!swapped) break;
  }
}
```

Complexity

Case	Complexity	Swaps	Reason
Best	$ O(n) $	$ O(1) $	In case of sorted array, the swapped flag remains false, so the outer loop breaks
Average	$ O(n^2) $	$ O(n^2) $	Two inner loops
Worst	$ O(n^2) $	$ O(n^2) $	Reversed array, two inner loops

1.1.2 Insertion Sort

Given an array, constructs the result by moving element to a already sorted subarray * The first element is considered sorted. * A loop is then run for later elements, containing another loop which finds the proper place for this next element. During iteration of the inner loop the previous element is moved to next till finding proper position for current element, then the saved current element is placed in its proper position * This algorithm runs O(n) on sorted arrays as the inner loop never runs if the previous element is smaller

[ ]:
```
template <class T>
void insertion_sort(vector<T>& array) {
  for (int i = 1; i < array.size(); i++) {
    T temp = array[i]; // saving in a temporary variable as the
    //previous one would be shifted it current location
    for (int j = i - 1; j >= 0; j--) {
      if (temp > array[j]) {
        array[j + 1] = temp;
        break;
      }
      array[j + 1] = array[j];
    }
  }
}
```

Complexity

Case	Complexity	Swaps	Reason
Best	$O(n)$	$O(1)$	The inner loop does not run, if data is sorted
Average	$O(n^2)$	$O(n^2)$	Two inner loops
Worst	$O(n^2)$	$O(n^2)$	Reversed array, two inner loops

1.1.3 Selection Sort

Given an array, per iteration the smallest form a subsection is acquired and placed at the start of the subsection * This algorithm is not adaptive * Number of swapping required is smaller in this algorithm

[ ]:
```
template <class T>
void selection_sort(vector<T>& array) {
  for (int i = 0; i < array.size(); i++) {
    T minimum = array[i];
    int min_location = i;
    for (int j = i; j < array.size(); j++) {
      if (minimum > array[j]) minimum = array[j], min_location = j;
    }
    swap(array[i], array[min_location]);
  }
}
```

Complexity

Case	Complexity	Swaps	Reason
Best	$O(n^2)$	$O(n^2)$	Sorted array, two inner loops
Average	$O(n^2)$	$O(n^2)$	Two inner loops
Worst	$O(n^2)$	$O(n^2)$	Reversed array, two inner loops

1.1.4 Quick Sort

- A pivot is selected usually the last element.
- Then the array is transformed as such that, the middle element(or elements, having the same value), reaches the correct position and elements lower than pivot stays at left and larger are placed at right
- Then for the left and right sides, quicksort is recursively called(in the below example a stack is maintained to perform recursion)

[ ]:
```cpp
#include <bits/stdc++.h>
using namespace std;

template <class T>
void quick_sort(vector<T>& array) {
  stack<pair<int, int>> st;
  st.push({0, array.size() - 1});
  while (!st.empty()) {
    pair<int, int> p = st.top();
    st.pop();
    if (p.first >= p.second) continue;
    if (p.second - p.first == 1) {
      if (array[p.first] > array[p.second])
        swap(array[p.first], array[p.second]);
      continue;
    }
    T pivot = array[p.second];
    int last_smallest = p.first;
    int last_largest = p.second;
    for (int i = p.first; i <= last_largest;) { // 3 way partitioning , Lumoto
      if (array[i] < pivot)
        swap(array[i], array[last_smallest++]), i++;
      else if (array[i] > pivot)
        swap(array[i], array[last_largest--]);
      else
        i++;
    }
    pivot = last_smallest - 1;
    st.push({last_largest, p.second});
    st.push({p.first, last_smallest - 1});
  }
}
```

Complexity

Case	Complexity	Swaps	Reason
Best	$O(nlog(n))$	$O(1)$	Sorted array, no change of position for pivots
Average	$O(nlog(n))$	$O(nlog(n))$	Two inner loops
Worst	$O(n^2)$	$O(n^2)$	Reversed array, n level recursion, ~n swaps per case

1.1.5 Merge Sort

- Generally ,the array is split is two halves until each subarray has only 1 element
- After the split the subarray of 1 element is considered sorted, and two broken subarrays are joined together in a special process called merging
- Merging - a process of combining two sorted arrays in linear time, in each iteration the smaller head element from one of the array is taken and pushed to the result array. This operation requires extra space (for running in linear time with easier implementation).

However, in this implementation, * The array was sorted in bottom up manner , the merging was done by taking 2^i elements each time where

$$i \in [2, \lceil \log(size) \rceil]$$

[ ]:
```
template <class T>
void merge_sort(vector<T>& array) {
  int iterations = pow(2, ceil(log2(array.size())));
  for (int i = 2; i <= iterations; i *= 2) {
    for (int j = 0; j < array.size(); j += i) {
      int l = j;
      int r = j + i / 2;
      vector<T> vv;
      while (l < j + i / 2 && r < j + i && l < array.size() &&
             r < array.size()) {
        if (array[l] > array[r])
          vv.push_back(array[r++]);
        else
          vv.push_back(array[l++]);
      }
      while (l < j + i / 2 && l < array.size()) vv.push_back(array[l++]);
      while (r < j + i && r < array.size()) vv.push_back(array[r++]);
      for (int k = j, p = 0; k < j + i && k < array.size(); k++, p++)
        array[k] = vv[p];
    }
  }
}
```

Complexity

Case	Complexity	Swaps	Reason
Best	$ O(n log(n)) $	$ O(n log(n)) $	n swaps/ placements in the last iteration, total iteration $\lceil log(size) \rceil$ times
Average	$ O(n log(n)) $	$ O(n log(n)) $	n swaps/ placements in the last iteration, total iteration $\lceil log(size) \rceil$ times
Worst	$ O(nlog(n)) $	$O(nlog(n))$	n swaps/ placements in the last iteration, total iteration $\lceil log(size) \rceil$ times

1.1.6 Heap sort

- The array is converted to a heap structure.
- Heap stucture is an array representation of a complete binary tree, where for ith node: $ 2 * i +1 $ is its left and $ 2 * i +2 $ is its right child. The parent and child nodes are related by by order.
- This structure is achieved from `heapify` and `Heapify` function, where `heapify` makes a heap from a leaf element, and `Heapify` constructs from a parent element.
- In a max heap, the heap order is - parent must be larger than its children. By extracting the top element and moving it to the end of the array repeatedly, the array can be sorted in ascending order

[ ]:
```
template <class T>
void heapify(vector<T>& array, int n) {  // on insertion or from bottom to top

  if (!n) return;
  if (array[n] > array[n / 2]) swap(array[n], array[n / 2]);
  heapify(array, n / 2);
}
template <class T>
void Heapify(vector<T>& array, int N, int n) {  // from top to bottom
  int i = 2 * n + 1;
  int j = 2 * n + 2;
  int largest = n;
  if (i < N && array[largest] < array[i]) largest = i;
  if (j < N && array[largest] < array[j]) largest = j;
  if (largest != n) {
    swap(array[largest], array[n]);
    Heapify(array, N, largest);
  }
}

template <class T>
void heap_sort(vector<T>& array) {
  for (int i = array.size() / 2; i < array.size(); i++) {
    heapify(array, i);
  }
  for (int i = array.size() - 1; i >= 0; i--) {
```

```
    swap(array[0], array[i]);
    Heapify(array, i, 0);
  }
}
```

Complexity

Case	Complexity
Best	$O(nlog(n))$
Average	$O(nlog(n))$
Worst	$O(nlog(n))$

1.1.7 Counting sort

- It's a non comparison sorting algorithm that works for only integers
- It uses an auxiliary array of size , maximum element of that previous array
- The counts of occurrences of an element in the main array is stored in the auxiliary array, then prefix sum is applied to the auxiliary array , this step makes the sorting stable
- Then the resulting array is constructed from the prefix sum array by iterating the main array in reverse order (the same elements are placed at last, ensuring stability).

[ ]:
```
template <class T>
void count_sort(vector<T>& array) {
  int maximum = -1;
  for (auto i : array) maximum = max(maximum, i);
  vector<T> v(maximum + 1, 0), result(array.size());
  for (auto i : array) v[i]++;
  for (int i = 1; i < v.size(); i++) v[i] = v[i - 1] + v[i];
  for (int i = array.size() - 1; i >= 0; i--) result[--v[array[i]]] = array[i];
  array = result;
}
```

Complexity : $ O(n+k) $

k = Maximum element

1.1.8 Radix Sort

- It is a improvised version of counting sort : saves space and becomes faster on non-uniform cases
- Instead of the whole number counting sort is applied to digits iteratively from LSB to MSB
- The stable nature of counting sort, maintains LSB's order : making sorting with radix sort possible

[ ]:
```
template <class T>
void radix_sort(vector<T>& array) {
  int maximum = -1;
  for (auto i : array) maximum = max((int)ceil(log10(i)), maximum);
```

```
  /*
   * exploits the fact that this approach of counting sort is stable
   * So it always remains previous lower digit order
   * thus the whole array gets sorted
   */
  for (int j = 0; j < maximum; j++) {
    vector<T> v(10, 0), result(array.size());
    for (auto i : array) v[(i / (int)pow(10, j)) % 10]++;
    for (int i = 1; i < v.size(); i++) v[i] = v[i - 1] + v[i];
    for (int i = array.size() - 1; i >= 0; i--)
      result[--v[(array[i] / (int)pow(10, j)) % 10]] = array[i];
    // without taking from reverse each time order will be changed and stability
    // would be lost scanning from end ensures last one is placed in the last
    // space, as count decreases gradually
    array = result;
  }
}
```

Complexity : $ O(d*n)) $

$$d = \lceil \log_{10}(maximum) \rceil$$

1.2 Theoretical comparison

Property	Bubble Sort	Insertion Sort	Selection Sort	Heap Sort	Merge Sort	Quick Sort	Counting Sort	Radix Sort
Best Case Complexity	O(n) (already sorted)	O(n) (already sorted)	O(n^2)	O (n log n)	O(n log n)	O(n log n)	O(n + k) (k is small)	O(dk) (k is small)
Average Case Complexity	O(n^2)	O(n^2)	O(n^2)	O(n log n)	O(n log n)	O(n log n)	O(n + k) (k ~=n)	O(dn) (k is small)
Worst Case Complexity	O(n^2)	O(n^2)	O(n^2)	O(n log n)	O(n log n)	O(n^2) (worst case, reverse)	O(n + k) (k is large)	O(dn) d is large
Recursion	No	No	No	Yes	Yes /No	Yes (pivot)	No	No

Property	Bubble Sort	Insertion Sort	Selection Sort	Heap Sort	Merge Sort	Quick Sort	Counting Sort	Radix Sort
Memory Usage	O(1)	O(1)	O(1)	O(1)	O(n) (merging)	O(log n) (stack)	O(n + k) (uses extra space for counting)	O(n + k) (uses extra space for counting)
Adaptability	Partial	Partial	No	No	No	No	No	No
Stability	Yes	Yes	No	No	Yes	No	Yes	Yes
Online/Offline	Offline	Offline	Offline	Offline	Offline	Offline	Online	Online
Serial/Parallel	Serial	Serial	Serial	Serial	Can be parallelized	Can be parallelized	Serial	Serial
In Place	Yes	Yes	Yes	Yes	Can be	Yes	No	No

1.3 Empirical Analysis

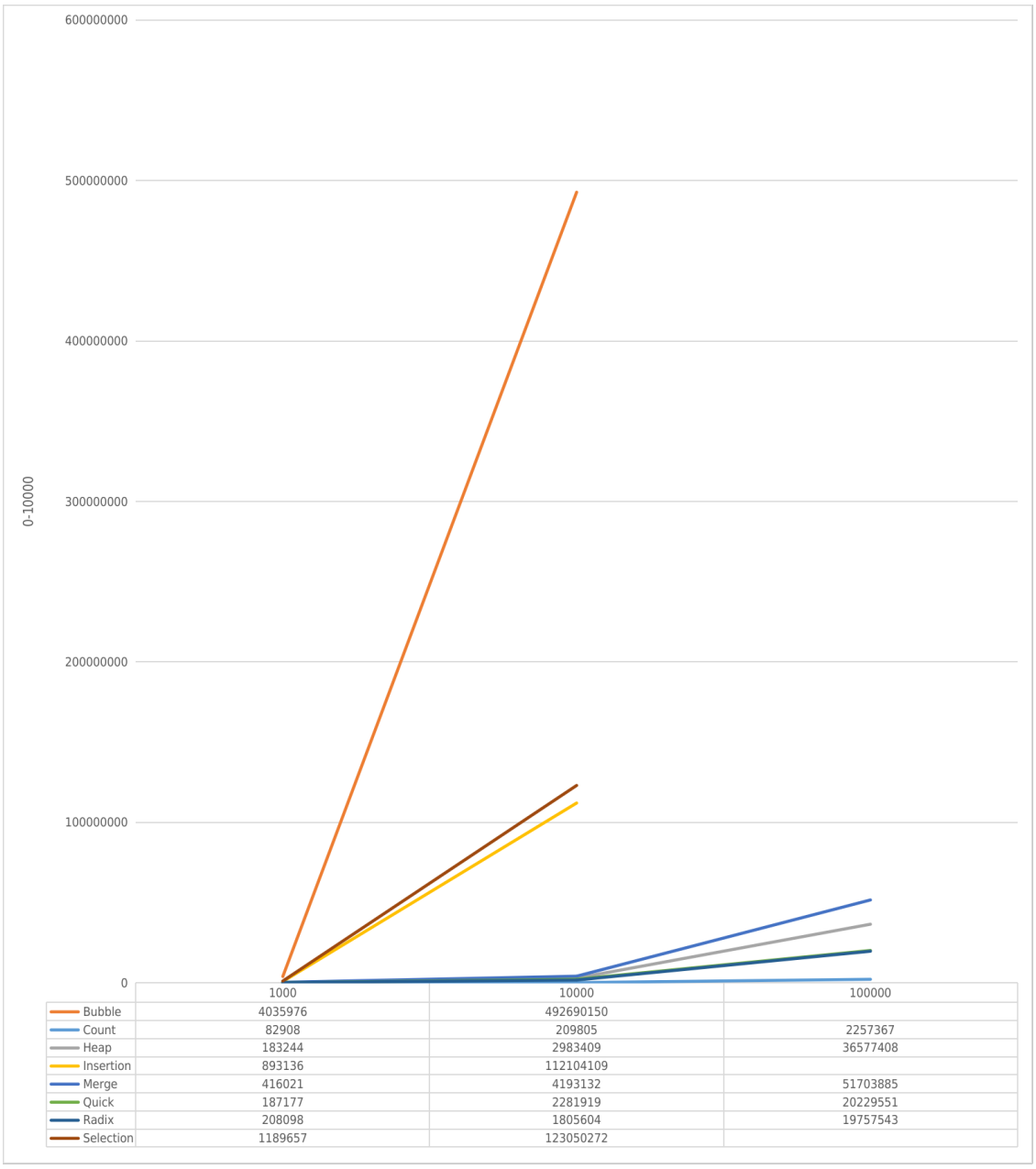
1.3.1 Data Tables

Number of Elements	Maximum	Algorithm	CPU Clocks taken
1000	10000	Bubble Sort	4035976
1000	10000	Count Sort	82908
1000	10000	Heap Sort	183244
1000	10000	Insertion Sort	893136
1000	10000	Merge Sort	416021
1000	10000	Quick Sort	187177
1000	10000	Radix Sort	208098
1000	10000	Selection Sort	1189657
1000	100000	Bubble Sort	3831858
1000	100000	Count Sort	604092
1000	100000	Heap Sort	164445
1000	100000	Insertion Sort	898520
1000	100000	Merge Sort	356513
1000	100000	Quick Sort	174396
1000	100000	Radix Sort	229309
1000	100000	Selection Sort	1181087
10000	10000	Bubble Sort	492690150
10000	10000	Count Sort	209805
10000	10000	Heap Sort	2983409
10000	10000	Insertion Sort	112104109
10000	10000	Merge Sort	4193132
10000	10000	Quick Sort	2281919
10000	10000	Radix Sort	1805604
10000	10000	Selection Sort	123050272

Number of Elements	Maximum	Algorithm	CPU Clocks taken
10000	100000	Bubble Sort	422810013
10000	100000	Count Sort	847383
10000	100000	Heap Sort	2560357
10000	100000	Insertion Sort	101990521
10000	100000	Merge Sort	4625901
10000	100000	Quick Sort	2852161
10000	100000	Radix Sort	2574060
10000	100000	Selection Sort	123707568
100000	10000	Count Sort	2257367
100000	10000	Heap Sort	36577408
100000	10000	Merge Sort	51703885
100000	10000	Quick Sort	20229551
100000	10000	Radix Sort	19757543
100000	100000	Count Sort	2545094
100000	100000	Heap Sort	37654346
100000	100000	Merge Sort	50033447
100000	100000	Quick Sort	36635401
100000	100000	Radix Sort	26099629
1000000	2147483647	Heap Sort	423495886
1000000	2147483647	Merge Sort	578784423
1000000	2147483647	Quick Sort	347657708
1000000	2147483647	Radix Sort	534466236
10000000	2147483647	Heap Sort	5712155289
10000000	2147483647	Merge Sort	6730631927
10000000	2147483647	Quick Sort	4138637182
10000000	2147483647	Radix Sort	4840681680

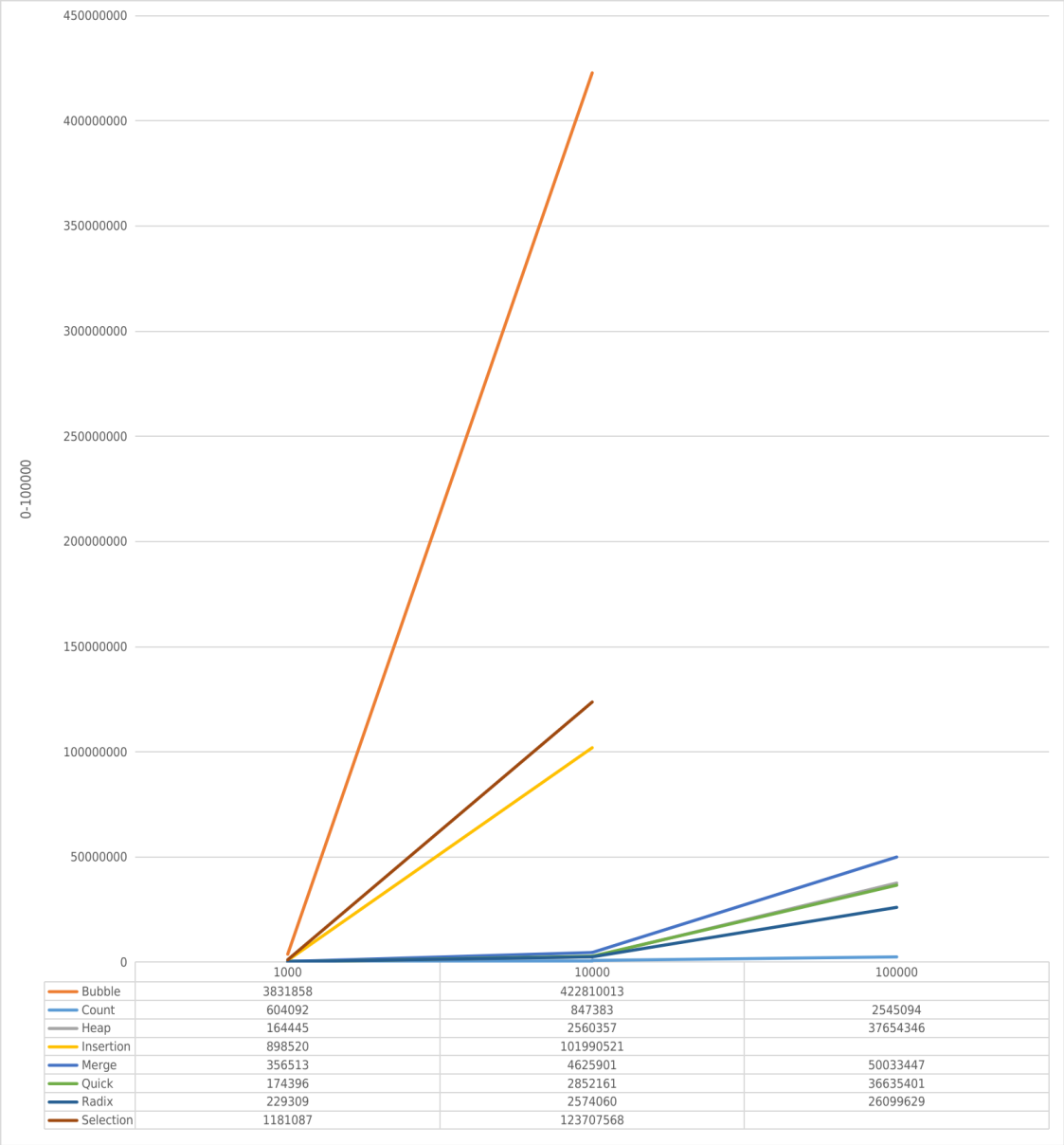
1.3.2 Plots

vertical line - No. of clocks, horizontal line - Number of data the lower the better



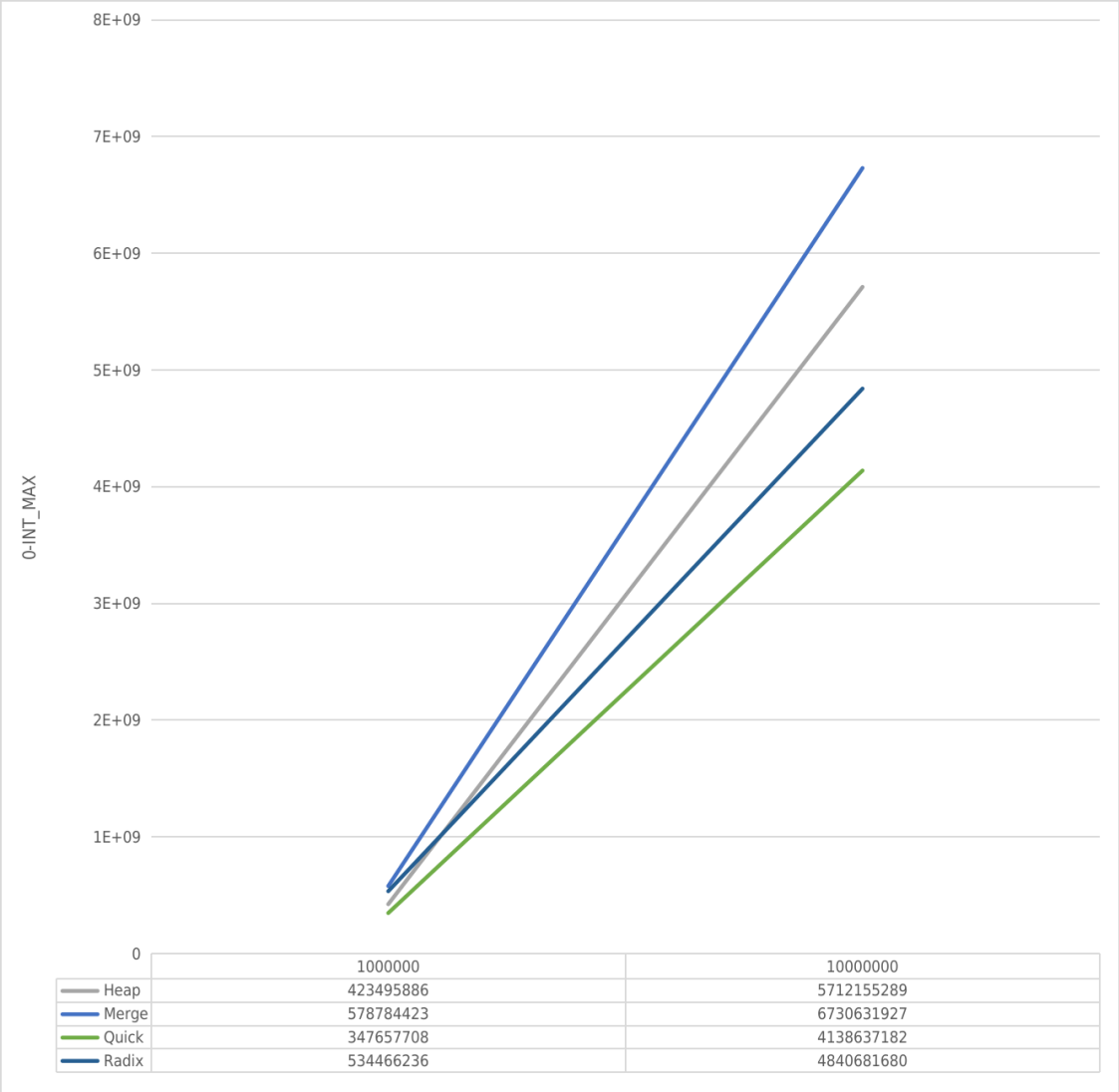
	1000	10000	100000
Bubble	4035976	492690150	
Count	82908	209805	2257367
Heap	183244	2983409	36577408
Insertion	893136	112104109	
Merge	416021	4193132	51703885
Quick	187177	2281919	20229551
Radix	208098	1805604	19757543
Selection	1189657	123050272	

Range [0,10000]



	1000	10000	100000
Bubble	3831858	422810013	
Count	604092	847383	2545094
Heap	164445	2560357	37654346
Insertion	898520	101990521	
Merge	356513	4625901	50033447
Quick	174396	2852161	36635401
Radix	229309	2574060	26099629
Selection	1181087	123707568	

Range [0,100000]



	1000000	10000000
Heap	423495886	5712155289
Merge	578784423	6730631927
Quick	347657708	4138637182
Radix	534466236	4840681680

Range [0,INT_MAX]

1.4 Discussion

An interesting observation in the plot, *Merge sort* performed worse than all other $O(n \log_2(n))$ sorting algorithm. More surprisingly, *Quick sort* performed the best. As the provided data were uniform and random, for each algorithm ran in it's average case complexity. For Merge sort merging and copying are costly and is constant as merge sort is not adaptive. For quick sort, levels can vary but the operations done in per level are not time consuming. For *Heap sort* the preprocessing takes $O(n \log_2(n)))$ time and extraction of all elements takes $O(n \log_2(n))$ time, and heapification is recursive

Among the second order nested loop algorithms, *Insertion sort* and *Selection sort* performed similar, the slight advantage that insertion sort has over selection may come due to the fact - for any sorted portions, insertion sort doesn't need to do any operations.In average cases, the number of comparisons needed is lower in insertion sort. In case of Selection sort it performs way better than *Bubble sort* as number of swapping is constant $O(n)$.

Counting sort performed the best in it's applicable data range, however outside it's data range and it's supported data type - it cannot perform. It's improvised version *Radix Sort* performed better for smaller data range however as the data range became larger $ d > \log (n) $ so it fell behind quick sort.

However The upper section is only a comparison for speed, for real world problems constraints may , and sorting algorithm should be chosen considering it's all properties