Homework 2 Analysis of Algorithms

Malik Türkoğlu

topic:

Design an experiment to compare following five sorting algorithms:

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

We'll compare it to ourselves and we will compare to each other.

-we will compare according to specific criteria(doing emprical analysis,using physical unit of time) , also compare experimental values to theoretical complexity values.

sample inputs

Sorted : Sorted inputs

D-Sorted : Decreasing sorted

Random : Random

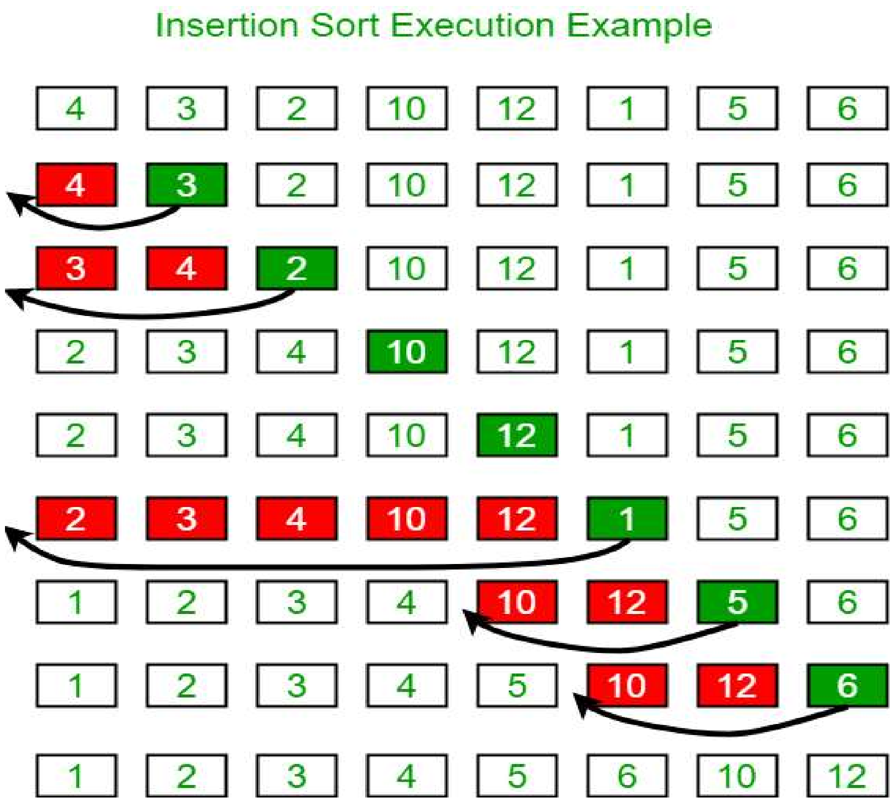
1.INSERTION SORT

explanation and theoretical part

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time .At the beginning of sorting, the first element is sorted already for first step.And then we look next element and compare the sorted array. If we operate this operation n*n times, we expect that the array is sorted at worst case.

Insertion sort performs two operations: it scans through the list, comparing each pair of elements, and it swaps elements if they are out of order. Each operation contributes to the running time of the algorithm.

As an example of insertion sort;



Insertion sort case :

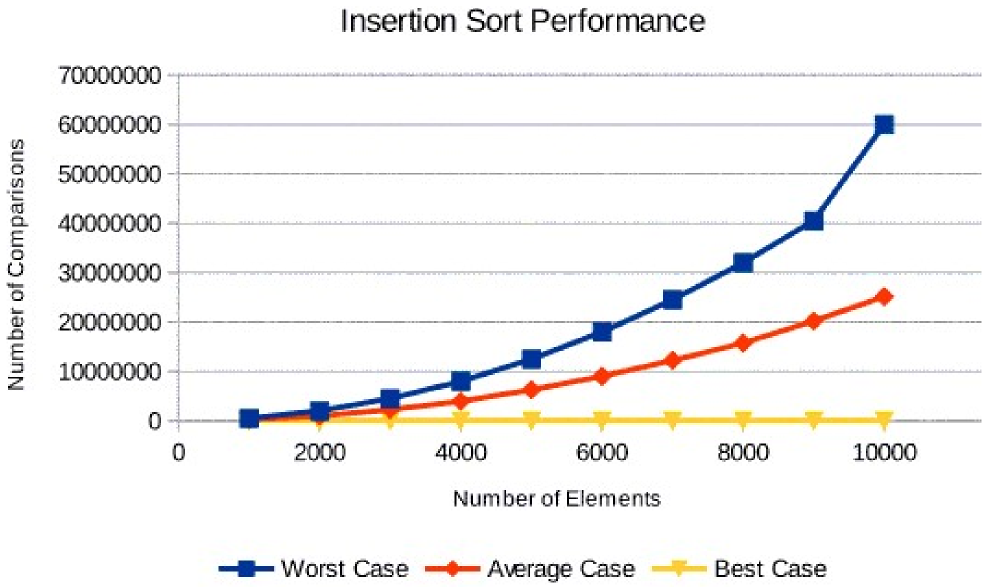
Best case : the input array is already in sorted order, insertion sort compares elements and performs no swaps. O(n)

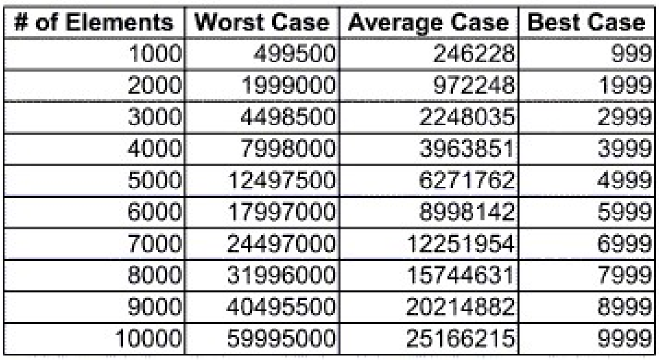
The worst case : for insertion sort will occur when the input list is in decreasing order. To insert the last element, we need at most (n-1) comparisons and at most (n-1) swaps.To insert the second to last element, we need at most (n-2) comparisons and at most (n-2) swaps, and so on.

we find 2(n-1)(n-1+1)/2 = n(n-1)

Use the master theorem to solve this recurrence for the running time. As expected, the algorithm's complexity is O(n^2)

The average case : average case have same complexity with worst case O(n^2)



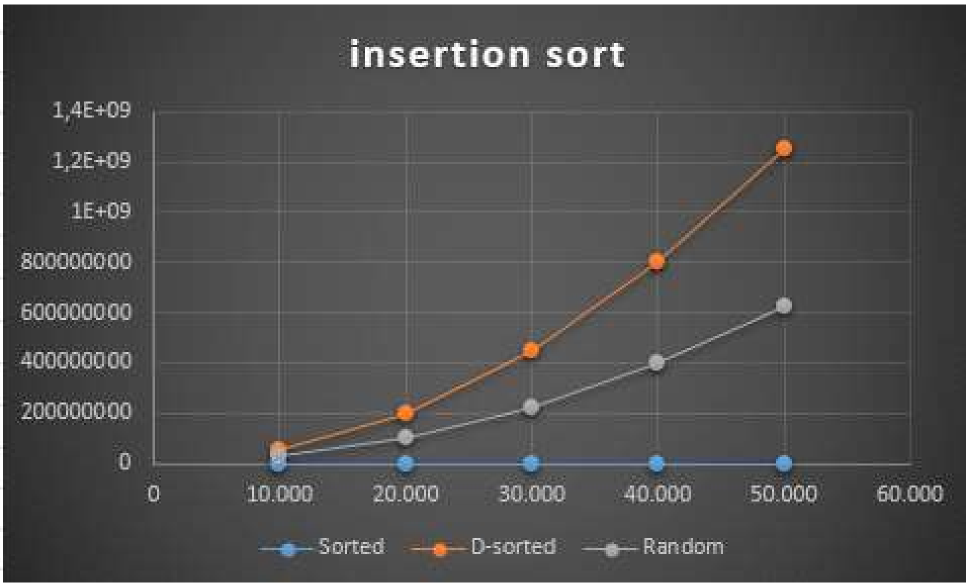


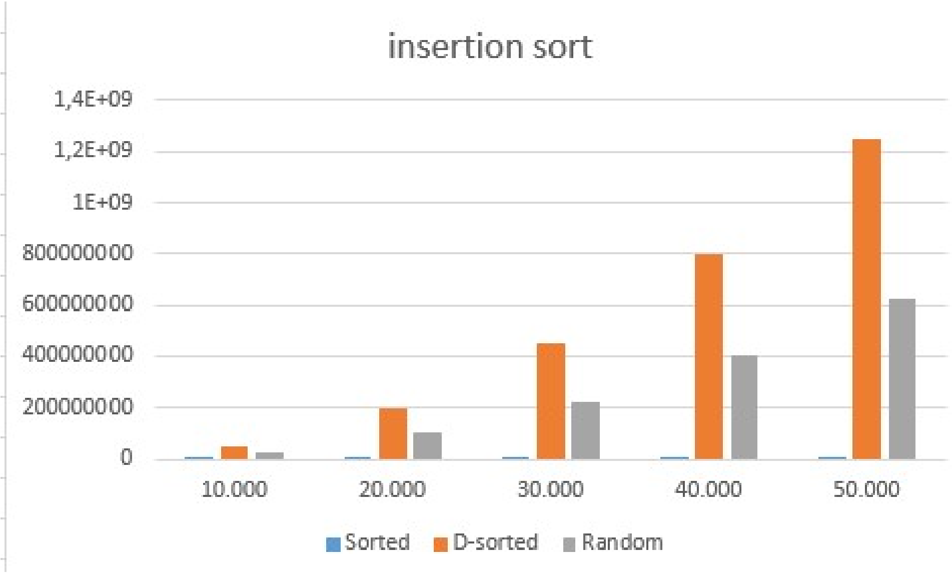
# of Elements	Worst Case	Average Case	Best Case
1000	499500	246228	999
2000	1999000	972248	1999
3000	4498500	2248035	2999
4000	7998000	3963851	3999
5000	12497500	6271762	4999
6000	17997000	8998142	5999
7000	24497000	12251954	6999
8000	31996000	15744631	7999
9000	40495500	20214882	8999
10000	59995000	25166215	9999

-experimental measurements of insertion-short :

INSERTION SORT TABLE

Input	Sorted	D-sorted	Random	Sorted(T)	D-sorted(T)	Random(T)
10.000	9999	49944979	24918771	0.74x10^-4	0.265000	0.157000
20.000	19999	199790083	100556438	1.49x10^-4	1.059000	0.546000
30.000	29999	449535120	225312630	2.37x10^-4	2.392000	1.212000
40.000	39999	799179461	400920755	3.02x10^-4	4.941000	2.147000
50.000	49999	1248725454	623392605	3.77x10^-4	7.069000	3.336000





2.MERGE-SORT

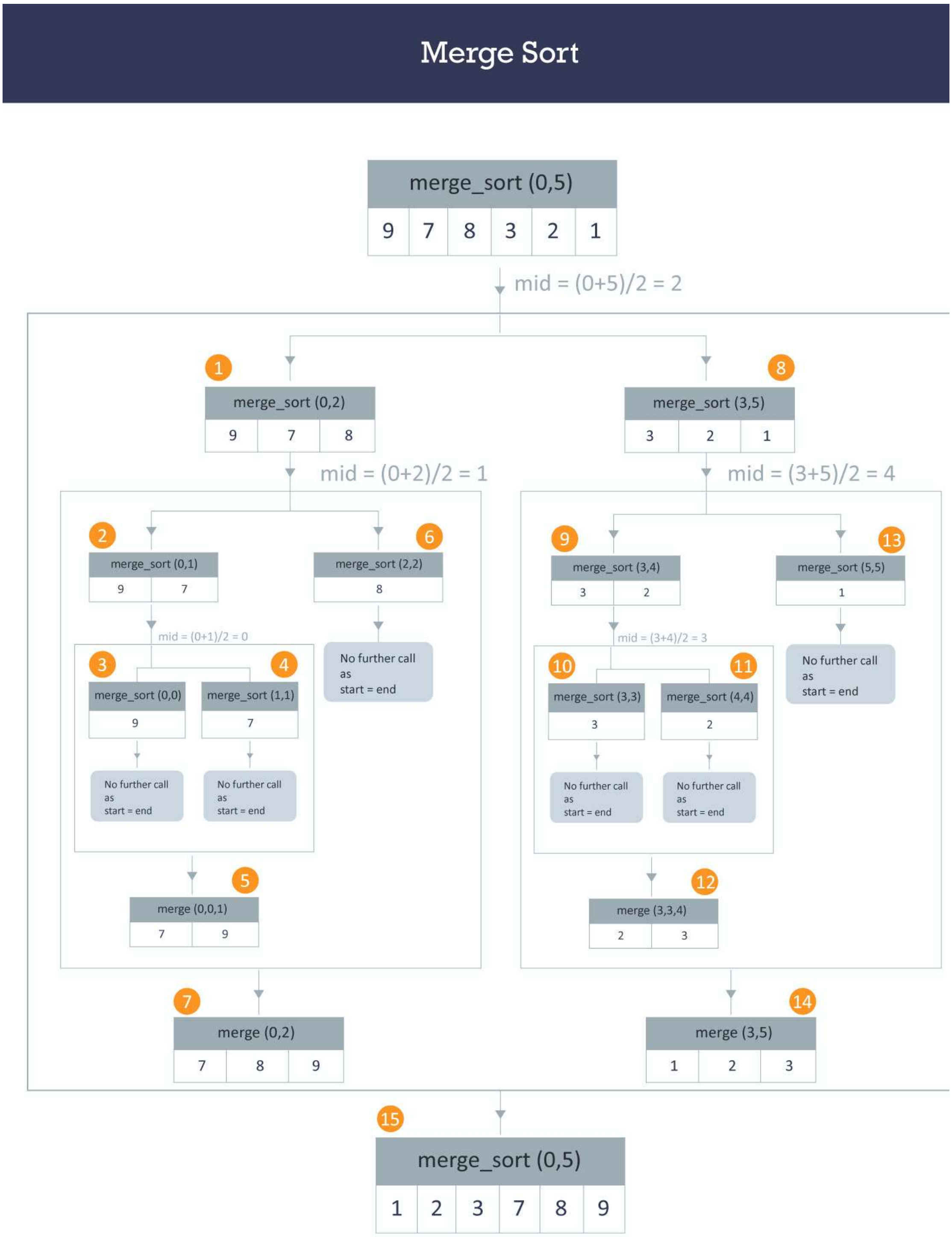
Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Divide the unsorted list into sublists, each containing element.

Take adjacent pairs of two singleton lists and merge them to form a list of 2 elements. will now convert into 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.



CASES :

best case : O(n logn)

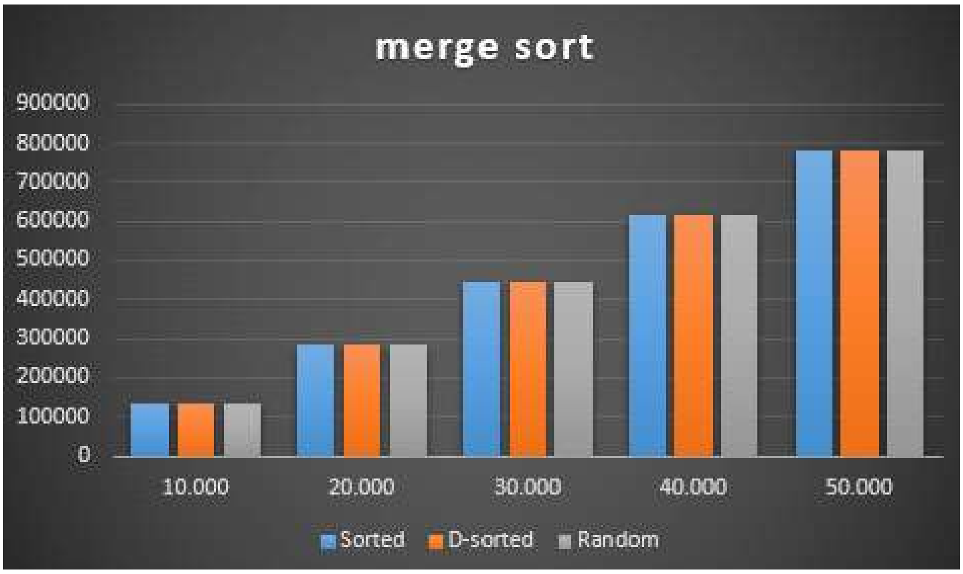
average case : O(n logn)

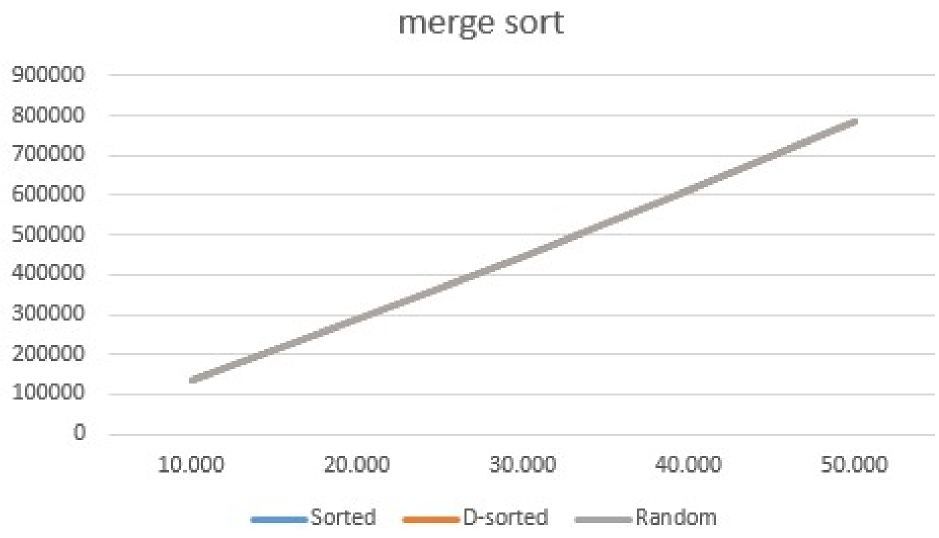
worst case : O(n logn)

-experimental measurements of merge-sort:

MERGE SORT

Input	Sorted	D-sorted	Random	Sorted(T)	D-sorted(T)	Random(T)
10.000	133616	133616	133616	0.01325	0.00160	0.00620
20.000	287232	287232	287232	0.01335	0.00190	0.00940
30.000	447232	447232	447232	0.01340	0.01200	0.01100
40.000	614464	614464	614464	0.01365	0.01240	0.01600
50.000	784464	784464	784464	0.01405	0.01440	0.01800





3.QUICK-SORT

explanation and theoretical part

Quicksort is a Divide and Conquer algorithm, It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quicksort that pick pivot in different ways.

we can take first element as pivot , we can take last element as pivot , we can take random element as pivot, we can take median as pivot.

Quicksort uses divide-and-conquer. As with merge sort, think of sorting a subarray array[p..r], where initially the subarray is array[0..n-1].

Divide by choosing any element in the subarray array[p..r]. Call this element the pivot. Rearrange the elements in array[p..r] so that all elements in array[p..r] that are less than or equal to the pivot are to its left and all elements that are greater than the pivot are to its right. We call this procedure partitioning. At this point, it doesn't matter what order the elements to the left of the pivot are in relation to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.

As a matter of practice, we'll always choose the rightmost element in the subarray, array[r], as the pivot. So, for example, if the subarray consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we choose 6 as the pivot. After partitioning, the subarray might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let q be the index of where the pivot ends up.

Conquer by recursively sorting the subarrays array[p..q-1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[q+ 1..r] (all elements to the right of the pivot, which must be greater than the pivot).

Combine by doing nothing. Once the conquer step recursively sorts, we are done.ll elements to the left of the pivot, in array[p..q-1], are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in array[q+1..r], are greater than the pivot and are sorted. The elements in array[p..r] can't help but be sorted!

Think about our example. After recursively sorting the subarrays to the left and right of the pivot, the subarray to the left of the pivot is [2, 3, 5], and the subarray to the right of the pivot is [7, 9, 10, 11, 12, 14]. So the subarray has [2, 3, 5], followed by 6, followed by [7, 9, 10, 11, 12, 14]. The subarray is sorted.

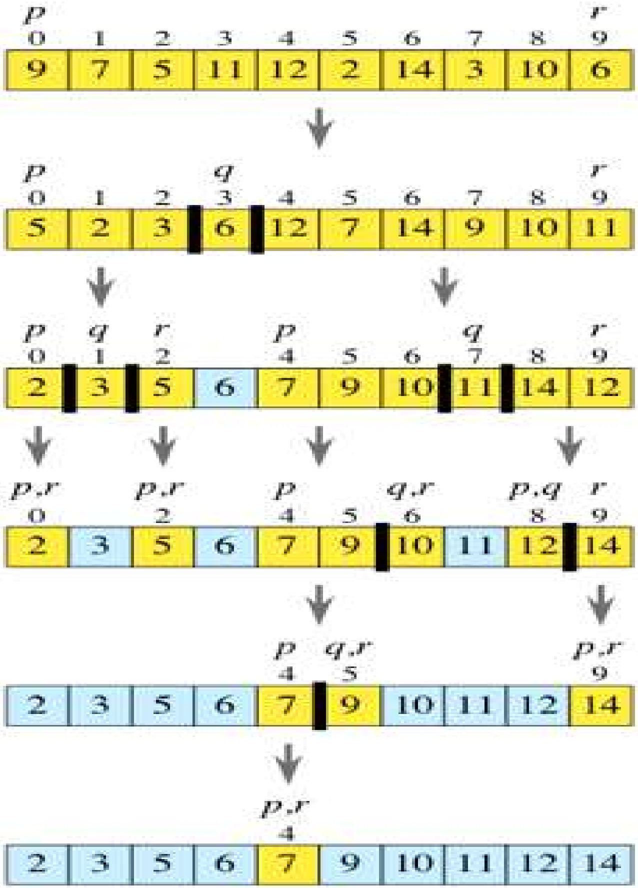
The base cases are subarrays of fewer than two elements, just as in merge sort. In merge sort, you never see a subarray with no elements, but you can in quicksort, if the other elements in the subarray are all less than the pivot or all greater than the pivot.

Let's go back to the conquer step and walk through the recursive sorting of the subarrays. After the first partition, we have subarrays of [5, 2, 3] and [12, 7, 14, 9, 10, 11], with 6 as the pivot.

To sort the subarray [5, 2, 3], we choose 3 as the pivot. After partitioning, we have [2, 3, 5]. The subarray [2], to the left of the pivot, is a base case when we recurse, as is the subarray [5], to the right of the pivot.

To sort the subarray [12, 7, 14, 9, 10, 11], we choose 11 as the pivot, resulting in [7, 9, 10] to the left of the pivot and [14, 12] to the right. After these subarrays are sorted, we have [7, 9, 10], followed by 11, followed by [12, 14].

Here is how the entire quicksort algorithm unfolds. Array locations in blue have been pivots in previous recursive calls, and so the values in these locations will not be examined or moved again:



CASES

Best case : If all the splits happen in the

middle of corresponding subarrays, we will have the best case.

Cbest(n) ∈ big theta(n*log2n)

Worst case : all the splits will be skewed to the extreme: one of the

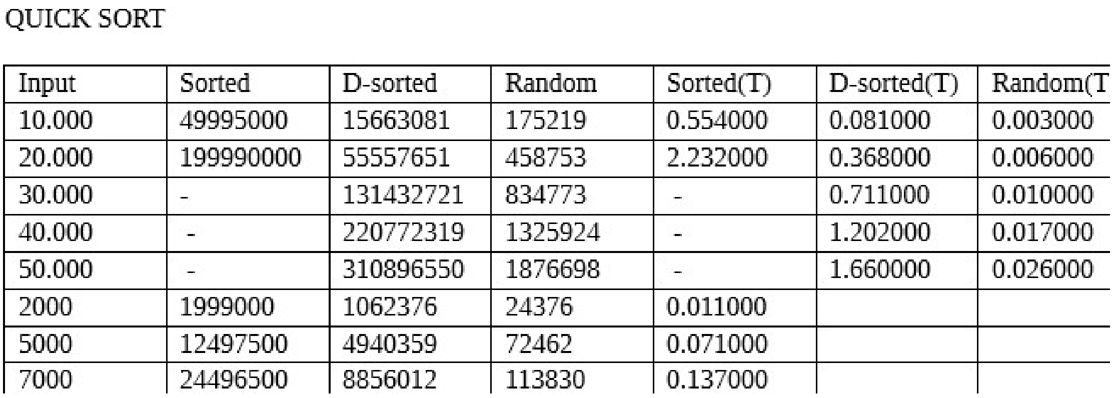
two subarrays will be empty, and the size of the other will be just 1 less than the size of the subarray being partitioned.

Cworst(n) ∈ big theta(n^2).

Average case : random inputs. about(9*n*log2n)

Cavg(n) ∈ big theta(n*log2n)

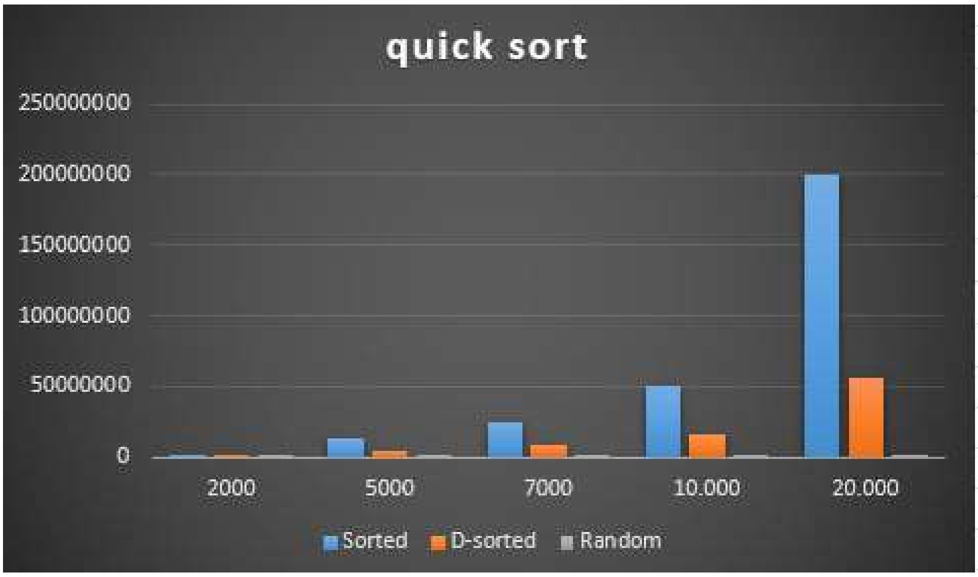
-experimental measurements of quick-short :

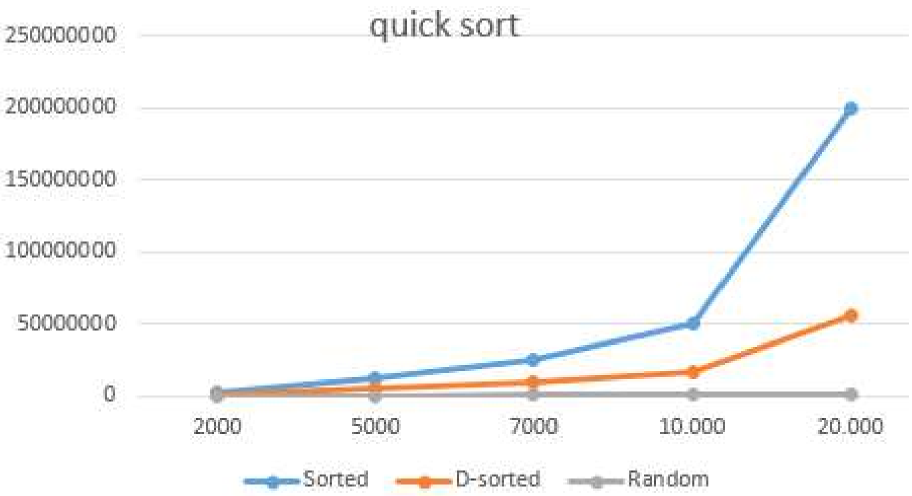


QUICK SORT

Input	Sorted	D-sorted	Random	Sorted(T)	D-sorted(T)	Random(T
10.000	49995000	15663081	175219	0.554000	0.081000	0.003000
20.000	199990000	55557651	458753	2.232000	0.368000	0.006000
30.000	-	131432721	834773	-	0.711000	0.010000
40.000	-	220772319	1325924	-	1.202000	0.017000
50.000	-	310896550	1876698	-	1.660000	0.026000
2000	1999000	1062376	24376	0.011000		
5000	12497500	4940359	72462	0.071000		
7000	24496500	8856012	113830	0.137000		

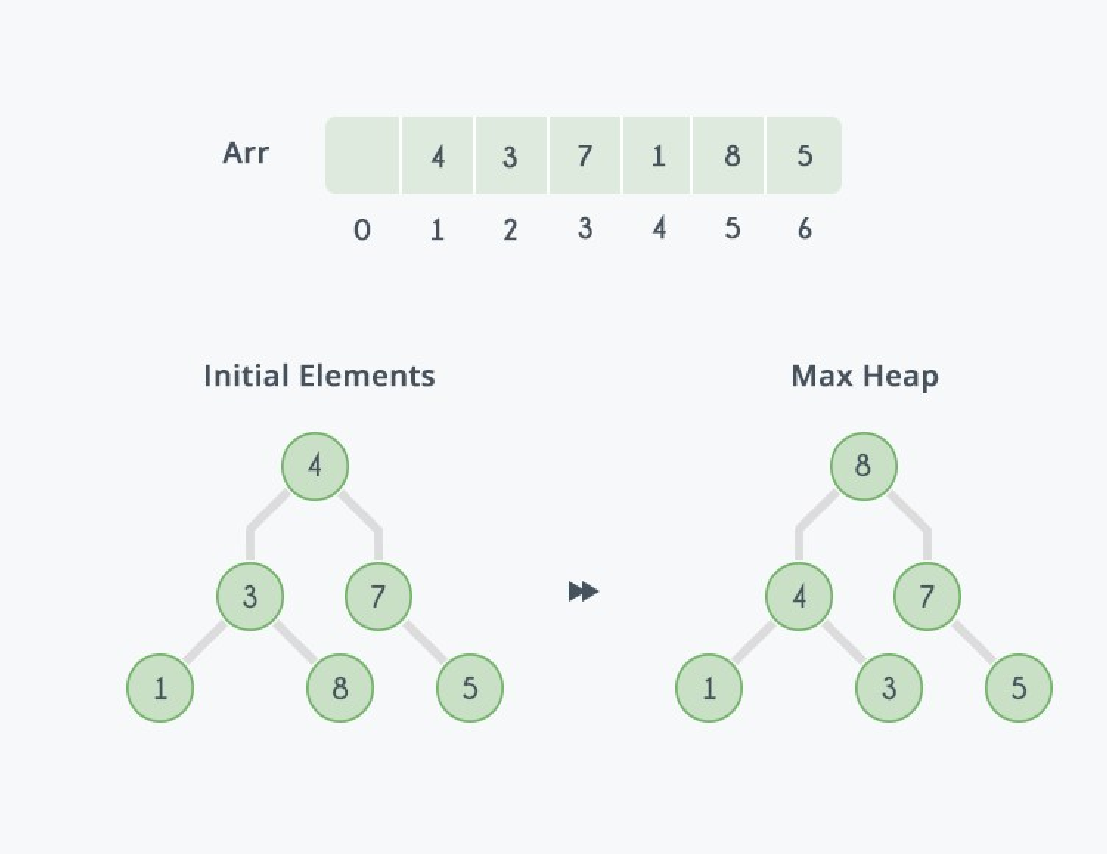
*C program could not count after 20.000 input, crushed. Therefore we counted 2000,5000,7000..





4.HEAP-SORT

In computer science, heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like that algorithm, it divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region. The improvement consists of the use of a heap data structure rather than a linear-time search to find the maximum.Although somewhat slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime. Heapsort is an in-place algorithm, but it is not a stable sort.



CASES

Worst Case Time Complexity: O(n*log n)

Best Case Time Complexity: O(n*log n)

Average Time Complexity: O(n*log n)

Space Complexity : O(1)

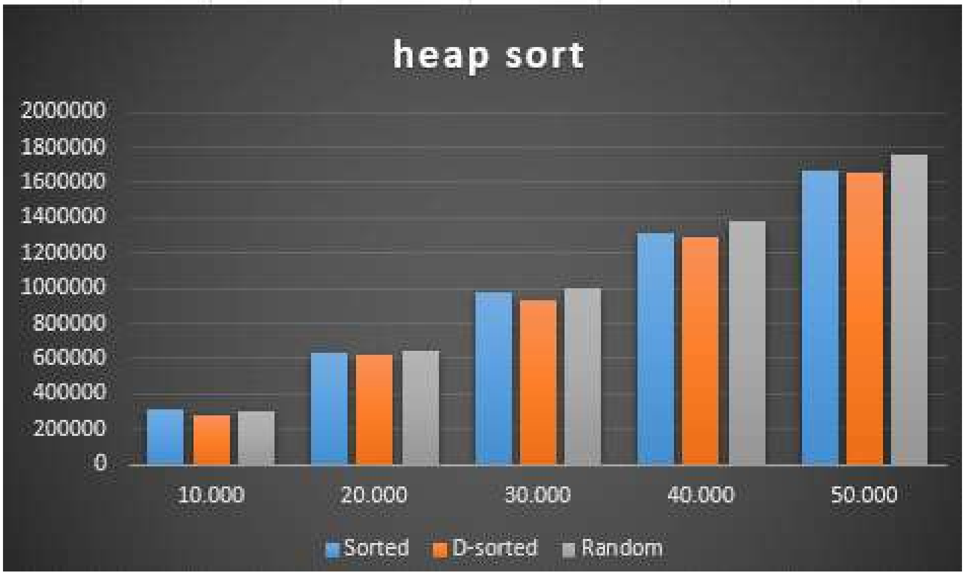
Heap sort is not a Stable sort, and requires a constant space for sorting a list.

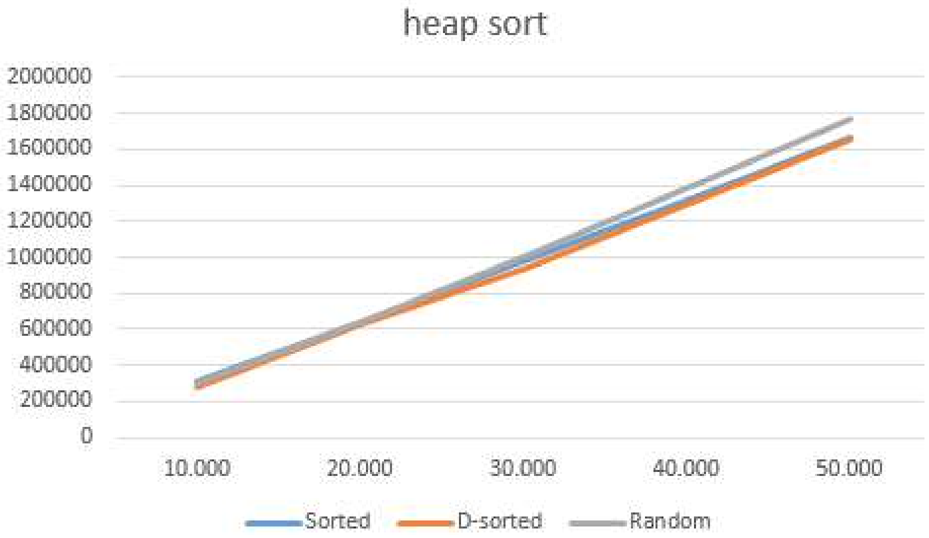
Heap Sort is very fast and is widely used for sorting.

-experimental measurements of heap-short:

HEAP SORT

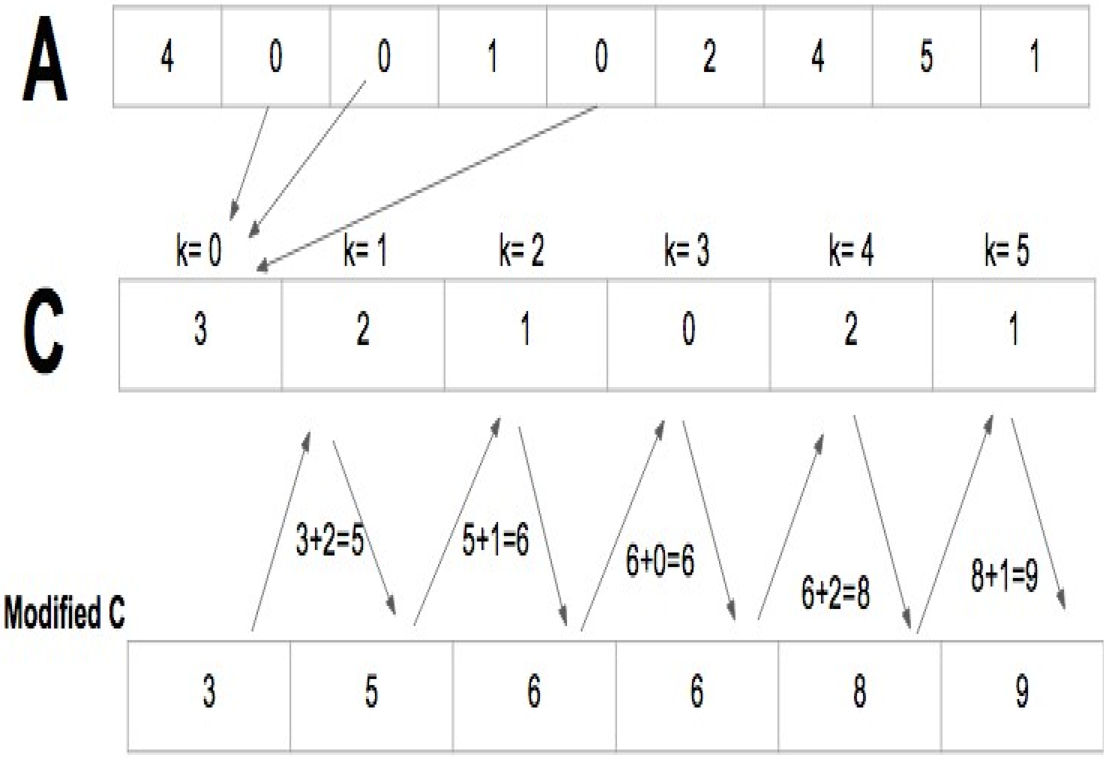
Input	Sorted	D-sorted	Random	Sorted(T)	D-sorted(T)	Random(T)
10.000	308179	275193	295575	0.00310	0.00310	0.00320
20.000	637612	626920	640748	0.00590	0.00600	0.00610
30.000	973232	935496	1004448	0.00890	0.00930	0.00940
40.000	1314926	1290776	1381050	0.01200	0.00120	0.01240
50.000	1666474	1652399	1766279	0.01500	0.00150	0.01700





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.



CASES

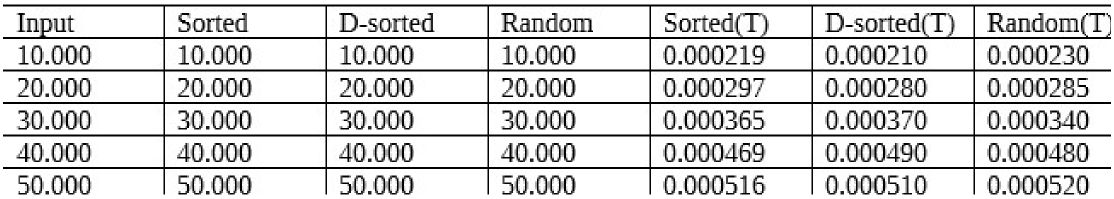
Linear time. Counting sort runs in O(n)O(n) time, making it asymptotically faster than comparison-based sorting algorithms like quicksort or merge sort.

Worst case time	O(n)
Best case time	O(n)

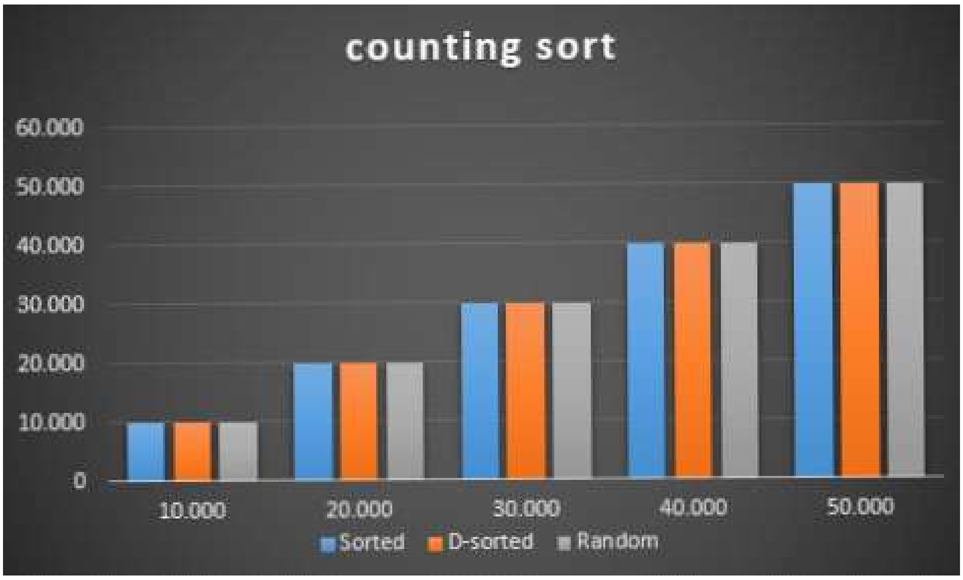
Average case time   O(n)

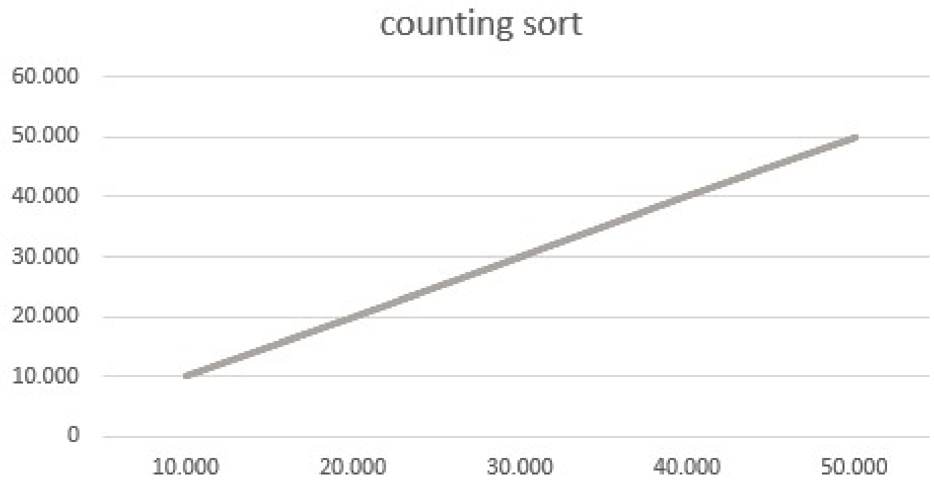
-experimental measurements of counting-short :

COUNTING SORT



Input	Sorted	D-sorted	Random	Sorted(T)	D-sorted(T)	Random(T)
10.000	10.000	10.000	10.000	0.000219	0.000210	0.000230
20.000	20.000	20.000	20.000	0.000297	0.000280	0.000285
30.000	30.000	30.000	30.000	0.000365	0.000370	0.000340
40.000	40.000	40.000	40.000	0.000469	0.000490	0.000480
50.000	50.000	50.000	50.000	0.000516	0.000510	0.000520





conclusion

To all appearances,counting sort is better than other, but we have to say that counting sort using more space, therefore counting sort seems best, but we have to know that, counting sort using more space to other.

Merge sort Treats every input equally, every same-size inputs have same comparison.

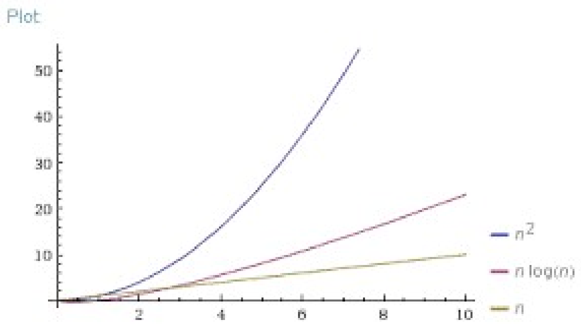
Insertion sort work very well sorted array, sorted array is best case for insertion sort, therefore we use sorted array with insertion sort. But other type inputs(random and D-sorted array) dont work very well, D-sorted inputs have big number of comparison.

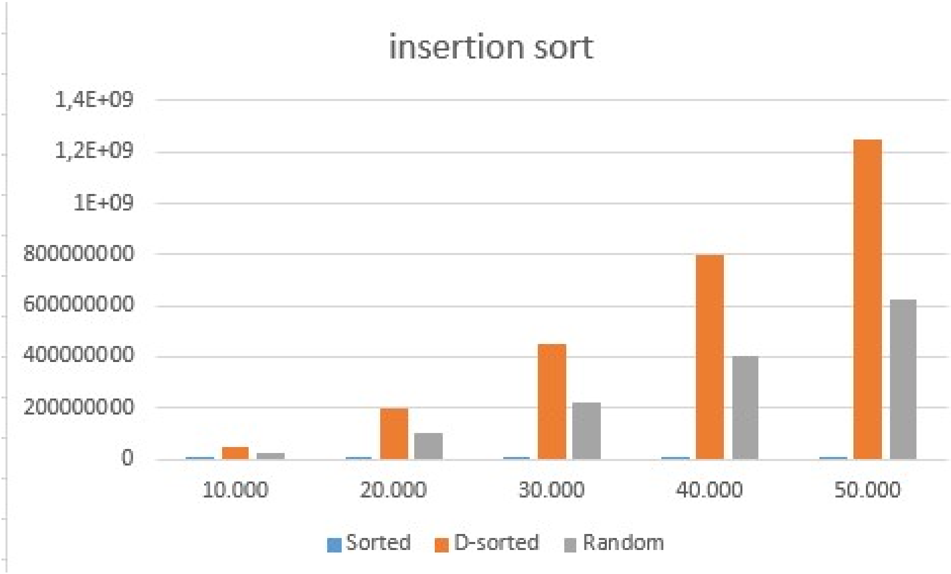
Heap sort algorithm treats every input equally, the result is very close each other.

In quicksort,we choose last element for pivot, therefore behave like worst case, average better than sorted array and D-sorted array.

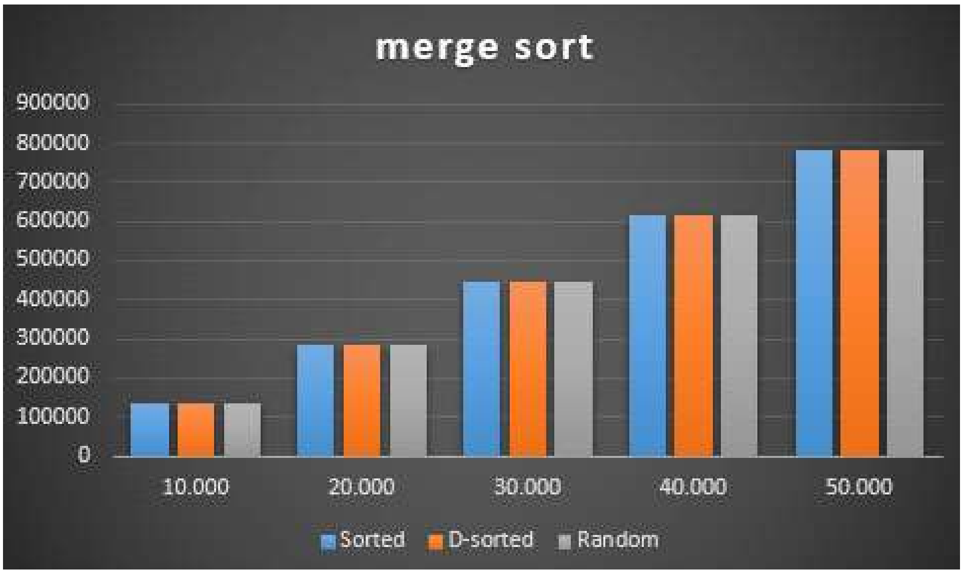
If we choose good pivot, quick sort algorithm have better than merge sort for little inputs.

Theoretical graph

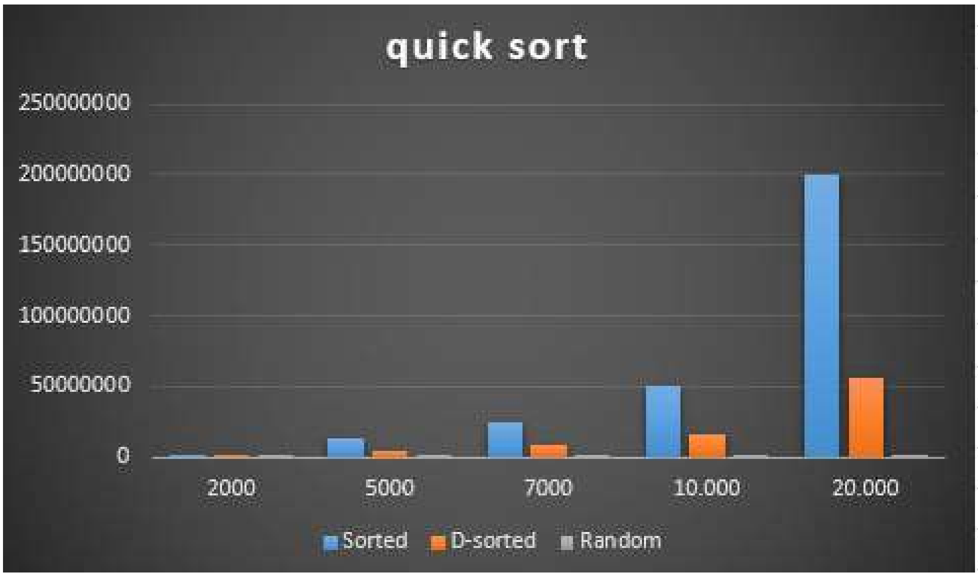




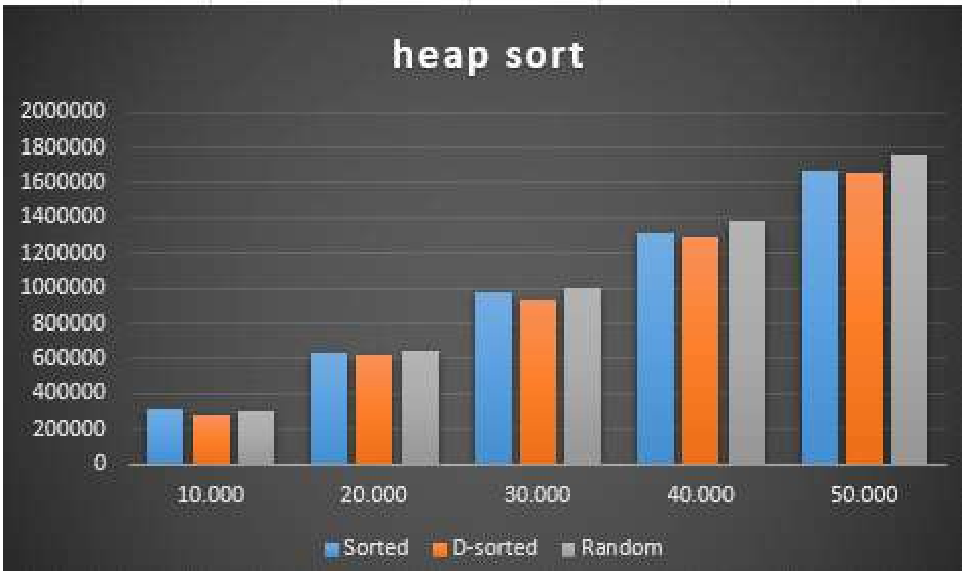
(This is best for Sorted inputs , work same as counting sort, but use less space. We can use for almost sorted array or sorted array, this is best for these.)



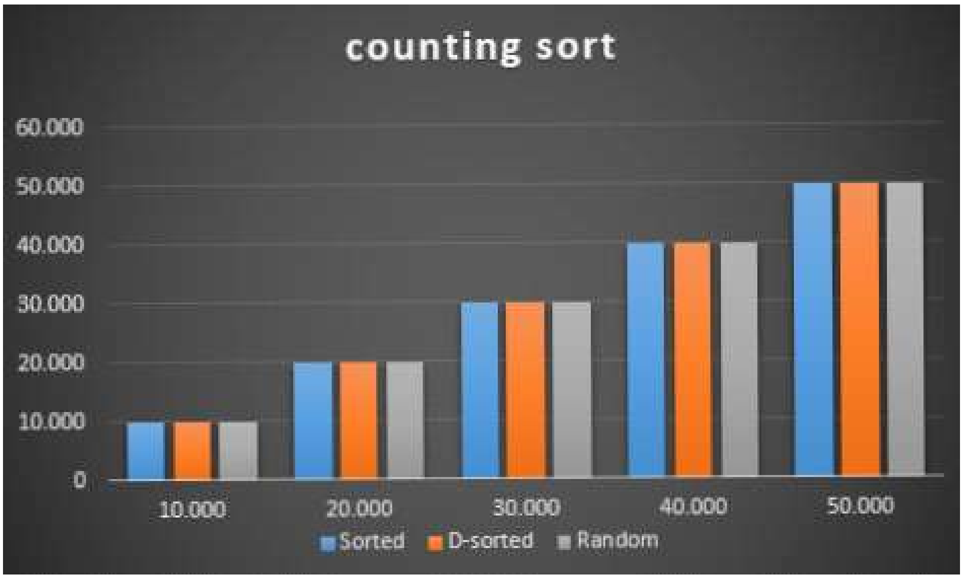
(This is best for if you dont know the inputs order and we have a few space(we compare counting sort for space). Also we can use big size data)



(Every time we choose last element for pivot, if we choose good pivot for quicksort(using method for pivot choosing),quick sort would be better than merge sort for little input.Also Mergesort uses extra space, quicksort requires little space.)



Heap sort algorithm can be implemented as an in-place(use less place)sorting algorithm. This means that its memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted. It needs no additional memory space to work.In contrast, the Merge sort algorithm requires more memory space. Similarly, the Quick sort algorithm requires more stack space.



(using most space,but result is better than others. But we can not use big size data for this algorithm because of space using.)

REFERENCES

www.geeksforgeeks.org

https://www.wikizero.com

https://stackoverflow.com/questions/31517645/counting-number-of-comparisons-in-quick-sort-using-middle-element-as-pivot-c

https://stackoverflow.com/questions/16275444/how-to-print-time-difference-in-accuracy-of-milliseconds-and-nanoseconds-from-c