ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Môn: Cấu trúc dữ liệu và giải thuật. Học kỳ 3, năm học 2021 – 2022.

Giảng viên hướng dẫn: Th.S Bùi Huy Thông.

Đề tài: Các thuật toán sort: thử nghiệm và so sánh.

Nhóm: 10

Thành viên:

- Bành Minh Phương 21127398
- Nguyễn Thế Thiện 21127170
- Nguyễn Trần An Hòa 21127047
- Ngô Quốc Quý 21127679

Thông tin cấu hình máy được sử dụng:

- Processor: Intel(R) Core(TM) i5-3230M CPU 2.60 GHz.
- RAM: 4.00 GB.
- System type: 64-bit operating system, x64-based processor.

ALGORITHM INTRODUCTION

1) Selection Sort:

Idea:

- The selection sort algorithm is an array by repeatedly finding the minimum element (considering ascending order) from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.
- In every iteration of the selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

Algorithm:

- For the first position in the sorted array, the whole array is traversed from index 0 to n – 1 sequentially. The first position where "first element" is stored presently, after traversing the whole array, the "smallest element" will be found and replaced.
- For the second position to the final, the array is divided into 2 parts. The subarray which is sorted, and the one which is unsorted. Program continues finding the next-smallest element in the unsorted subarray, and gives it into the sorted subarray (to the right of the previous element).

Time complexity: O(n^2) for all cases.

Space complexity: O(1) for all cases.

2) Insertion Sort:

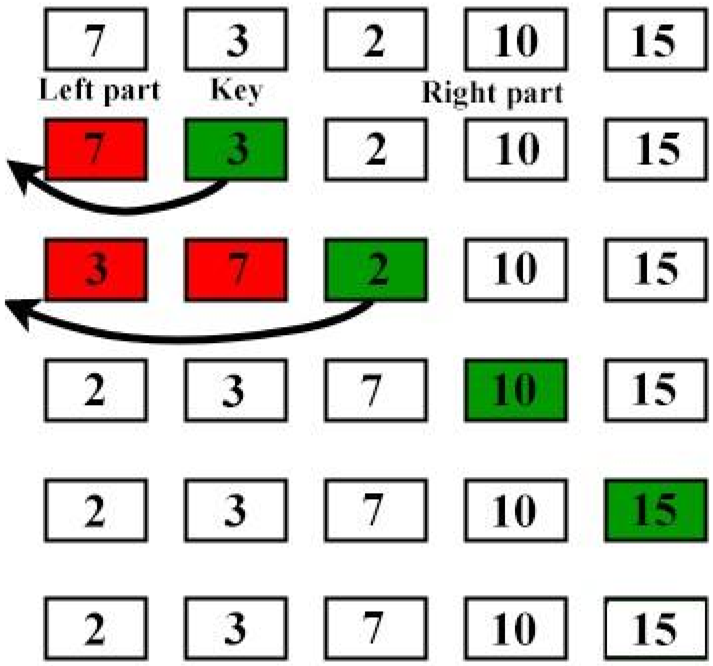
Idea:

- The array is divided into 2 parts:
  - The always sorted array (the left part).
  - The unsorted array (the right part).
- While the right part is not empty, the leftmost element of this part is picked up and compared to the elements on the right part, it will be inserted back into the left part such that this sorted subarray is one element larger.

Algorithm:

- Step 1: The left part only has only one element (the first element of the given array).
- Step 2: Let the leftmost element of the right part be 'key'.
- Step 3: Compare it with the left part.

- If elements in the left part are greater than the key, they will be moved to the right one position. (This means that it'd move the bigger elements one position up to make space for the swapped element.)
- If the key is greater or equal to an element in the left part, or the end of the left part is reached, the key will be inserted to the right position.



Example of insertion sort

Time complexity:

- Best case: O(n) -> the array is already sorted
- Average case: O(n^2)
- Worst case: O(n^2) -> the array is in the reverse order.

Space complexity: O(1) for all cases.

Improved insertion sort:

- Binary insertion sort uses binary search instead of linear search like the above version, which helps to reduce the comparative value of inserting a single element from O(N) to O(log N).
- This improved sort works faster, more stable, especially when the array is heavily sorted.

3) **Bubble Sort:**

Idea: from the name "bubble", while traversing the input array, we compare and swap adjacent pairs so that the biggest value goes to the very end of the array while having the smaller values go towards the beginning (like bubbles going to the surface).

Step-by-step descriptions:

- Step 1: Compare and swap an adjacent pair. For example, for the values at the indexes X and X+1, if arr[X+1] < arr[X], we swap the two values so the bigger is one step closer to the array's end, else the pair is unchanged.
- Step 2: Do step 1 from the beginning to the end of the array, this ensures that at the end of the iteration is the biggest value, since the value is always swapped towards the end of the array for every pair since its appearance.
- Step 3: Do step 2 again until the size of the unsorted region is 1, where the array is successfully sorted.

Space complexity: O(1) for all cases.

Time complexity: O(n^2) for all cases.

Improvements:

- Use a swap checker for each iteration of the sort: if there's no swap in an iteration, that means the array is already sorted, so we stop instead of continuing the next iterations. The complexity stays almost the same except time complexity for best case is O(n) for an already sorted array.

4) **Shaker Sort:**

Idea: basically, almost the same as Bubble Sort, where in each iteration, the check and swap adjacent pairs go from the start of the array to the end and back, leaving the biggest at the end and the smallest at the beginning after each iteration.

Step-by-step descriptions:

- Step 1: Compare and swap an adjacent pair that the smaller of the pair is towards the beginning and vice versa.
- Step 2: In each iteration, swap adjacent pairs from the beginning to the end, then from the end to the beginning.
- Step 3: Redo the step 2 except lowering the upper bound and raising the lower bound by 1 after each iteration.

→ For example: in the 1st iteration, swap from the pair of indexes 0 & 1 to the pair n-2 & n-1 (assume n is the size of the array), then swap from the pair n-2 & n-3 to pair 0 & 1. After each iteration, it's possible to decrease the number of pairs we must check.

Space complexity: O(1) for all cases.

Time complexity: O(n^2) for all cases.

Improvements:

- Use a swap checker, so that if there's an iteration where either traversing from the beginning to the end or the other way around resulting in no swap done: stop the algorithm as the array is already sorted. The complexity stays almost the same except time complexity for best case is O(n) for an already sorted array.

5) **Shell Sort:**

   Idea:

- To implement the Shell sort algorithm, the program will divide the array of elements that need to be sorted by concatenating the elements that are far apart in the array of elements, and then use Insertion sort to sort the sub-arrays that have just been created.
- After completing the arrangement of the above elements, we continue to concatenate new elements but will reduce the distance between the elements that we combine and repeat the Insertion Sort process.
- The process will be performed sequentially until the distance of the elements that we concatenate is 1 , this is also the time when the element array has finished sorting by the "Shell Sort" method.

Algorithm:

- **Step 1** : the variable that we would pass in the shellsort() function: array *a[]* containing all the elements that need to be sorted and *n* is the amount of the elements.
- **Step 2** : Declare indexes *i,j, interval* to represent for the distance of the elements we going to concatenate, first *interval* will have value = *n/2* and deal to 1
- **Step 3** : Create a loop with index *i* assign = *interval* and divided by two each time until it equals 0.
- **Step 4** : Dividing the *a[]* into an interval sub-array with the elements is the elements of *a[]* with the distance = interval.

- **Step 5** : Using Insertion Sort to sort this sub-array and back to step 3 until the loop ends.

Time Complexity:

- Worst case: O(n^2)
- Best case: O(n*log n)
- Average case: O(n*log n)

Space Complexity: O(1) for all cases.

6) **Heap Sort:**

Heap sort is a comparison-based sorting technique based on Binary Heap data structure

Binary Heap data structure is a structure with n elements with some conditions :

- *a[n]* (array a has n elements) and index *i* , for every *i* $(0 \leq i \leq n/2-1)$
- *a[2i+2]<a[i] & a[2i+1]<a[i]*

This above definition structure is called max heap (It also has min heap with opposite conditions).

Idea: Use a max heap to get the biggest number of the unsorted part of the array (the heap) and move it to the sorted part, like how Selection Sort works.

How to build a heap structure.

- **Step 1**: Create a loop with index *i = n2-1 —> 0* . Inside loop we launch function BuildHeap and pass in it (a[], n elements, , index i)
- **Step 2**: Input variable is (a[], n elements, i)
- Define *largest = i* , *l = 2i+1* , *r = 2i+2*
- **Step 3**: Compare *a[largest]* with *a[l]* if (*a[i] > a[largest]*) -> swap value of *i* with largest then back to step to with the *a[r]* .
- **Step 4**: Compare *i* with largest *if i ≠ largest* swap value of *a[i]* with *a[largest]* , and back to *step 1* with the input (a[], n elements, a[largest])

Heap sort algorithm

- **Step 5:** After we get heap structure, create a loop starting from the last element of *a[]* to the first. Inside loop swap variable of *a[i]* with *a[0]* (It's mean bring the highest value element into the *i* position which is the last position of array)
- **Step 6 :** Then build a heap structure again. (back to Step 1) until the loop ends.

Time Complexity: O(n logn) for all cases.

- Time complexity of heapifying is O(Logn).
- Time complexity of BuildHeap() is O(n).

Space complexity: O(1) for all cases.

7) **Merge Sort:**

Idea: It divides the input array into two halves, calls itself for two halves, and then it merges the two sorted halves.

Algorithm:

- Declare left variable to 0 and right variable to n-1
- Find mid by medium formula. mid = (left+right)/2
- Call merge sort on (left,mid)
- Call merge sort on (mid+1,right)
- Continue till left is less than right
- Then call the merge function to perform merge sort.

Time complexity: O(nlog n) for all cases.

Space complexity: O(n) for all cases.

8) **Quick Sort:**

Idea:

Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort. The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Algorithm:

- **Step 1** : Declare quicksort() function and pass in arr[] that needs to be sorted , and l,r as the first and last position of the arr[].
  Declare p as 'pivot' with value of a[l+r]/2.
- **Step 2** : Initialize a loop with index i = l and index j = r and the condition to exit loop is i > j. Inside this loop, initialize two other loops to find the a[i] that > p from the left of the array and find a[i] that < p from the right of the array. Every time the program finds a[i] and a[j] to satisfy the above conditions, the program would swap those values.
- **Step 3:** When i is still < r (it means i run from 0 to n-1) program would call itself which it passes in i instead of l.

When j is still > l (it means j runs from n-1 to 0) the program would call itself which it passes in j instead of r.

Time complexity:

- Best case: O(nlog n).
- Average case: O(nlog n).
- Worst case: O(n^2).

Space complexity: O(log n) for all cases.

9) Counting Sort:

Idea:

- Counting sort makes assumptions about data, for example, values are going to be in the range of 0 to 99, …
- Counting sort works by iterating through the input, counting the number of times each item occurs, and using those counts to compute an item's index in the final, sorted array.

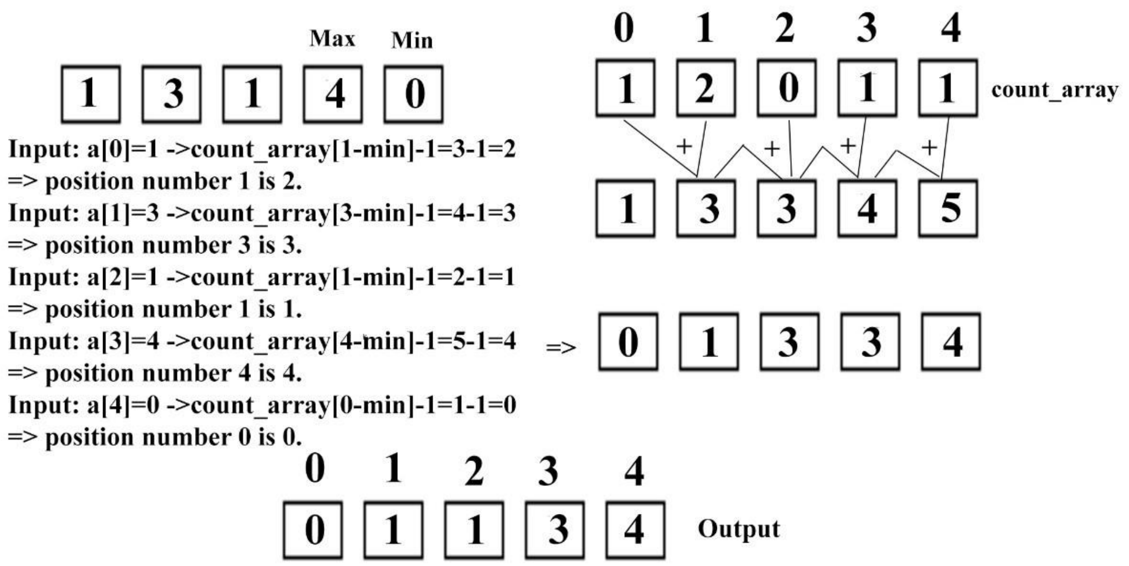
Algorithm:

- Step 1: Create another array to store the count of each individual input value.
- Step 2: Store the count of each unique element in the new array (if the element repeats, create its count).
- Step 3: Modify the array such that each element at each index stores the sum of previous counts.
    - Array now contains the actual position of input array values.
- Step 4: Having the sorted array by considering the input array and the new array, collating the input value and the index in the created array (the value of the index is the position of the input value in the sorted array) then minus 1 to have a new position for the next same number.

Space complexity: O(k)

Time complexity: depends on k.

- Best case: O(N+1) ->all items are in the same range.
- Average case: O(N+k)
- Worst case: occurs when the range k of the counting sort is large.



Input: a[0]=1 ->count_array[1-min]-1=3-1=2
=> position number 1 is 2.
Input: a[1]=3 ->count_array[3-min]-1=4-1=3
=> position number 3 is 3.
Input: a[2]=1 ->count_array[1-min]-1=2-1=1
=> position number 1 is 1.
Input: a[3]=4 ->count_array[4-min]-1=5-1=4
=> position number 4 is 4.
Input: a[4]=0 ->count_array[0-min]-1=1-1=0
=> position number 0 is 0.

Example of couting sort

10) Radix Sort:

Idea: Digit-by-digit sort starting from least significant digit to most significant digit. Radix sort uses Count Sort as a subroutine to sort.

Algorithm:

- Firstly, the array will be divided into groups based on the value of the unit digits.
- After dividing, we will have the array with the groups having unit digits from least to most.
- Continue to divide the array based on the value of tens digits. The same as step 1, the program will divide the array into the groups having tens digits from least to most.
- Do the same as the previous step with the remaining row digits.
- The algorithms will stop when all of the final row digits equal to 0.

Time complexity: O(kn) for all cases. (k is the maximum length of all numbers in the array)

Space complexity: O(k+n) for all cases.

11) Flash Sort:

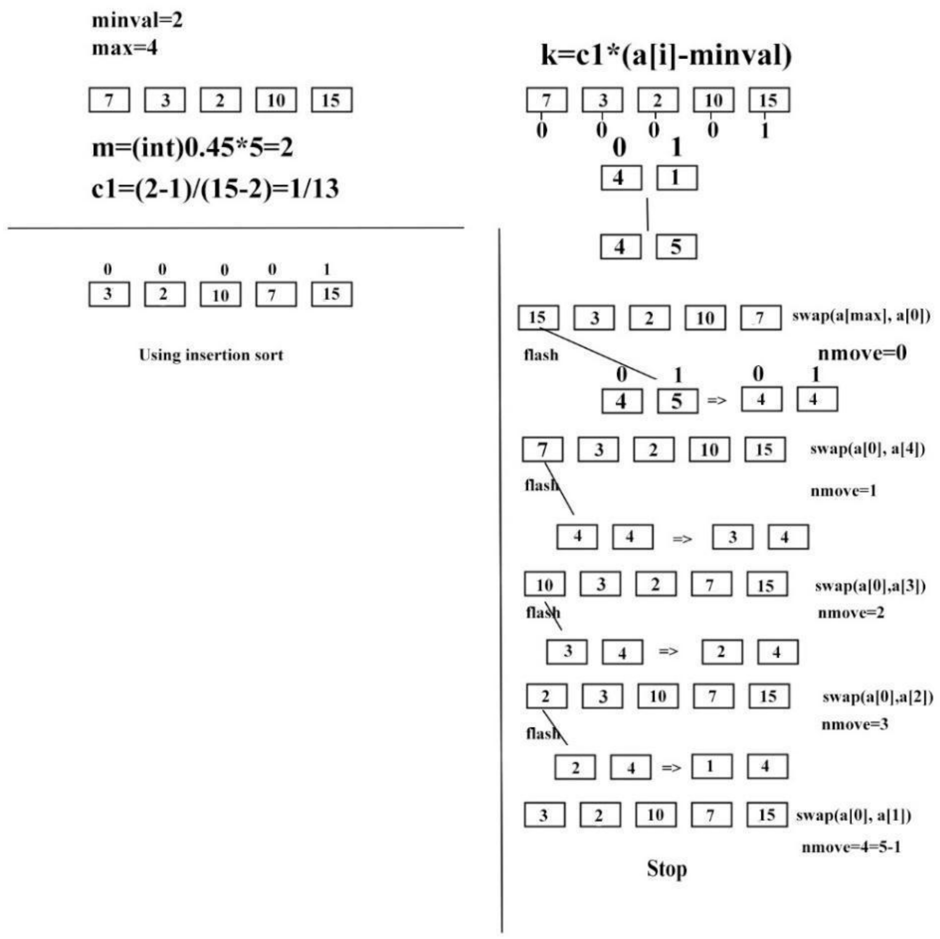
Idea: Divide elements of the array into certain classes (calculated by a special formula), and arrange elements to suitable classes. Then gathering elements in the same class, then using another sort to put it in the right position. (In this case, we use insertion sort).

Including 3 main steps:

- Data classification
- Global permutation
- Local arranging

Algorithm:

- Step 1: based on Bucket sort to classify the array, we find the smallest value of the array and the index of the maximum value.
- Step2: To know how many classes we need to divide elements into, multiply the number of arrays with 0.45 and round down the result to have an integer value.
- Step 3: calculating a special value (a factor helping classify): c1= (number of elements – 1)/ (maximum value – minimum value) (float is accepted)
- Step 4: To know which class is each of element in, k= c1*(current value – minimum value). K is the index of the 'Bucket' array. Elements of the 'Bucket' array go up 1 in accordance with k.
- Step 5: Swap maximum and the first element in the array. And initializing k=m-1 (m: the number of buckets), which is equivalent to the class of the maximum value.
- Step 6: As the sorting needs only n-1 swap (the maximum moves of swap), n elements will move to their right class. This is the condition to stop the loop while.
- Step 7: Initializing j=0, if j>L(k)-1 (L(k) is the element in the 'bucket' array), this means a[j] is in the right class, so we skip it and move to the next element j+1 to estimate the other elements.
- Step 8: When moving one element to the right class, the classes' last position minus 1 and increase the number of swaps. This process continues until L[k]=j, which means class k is full.
- Step 8: After global permuting, elements in the array are divided into the right class (but they are not sorted yet), so the insertion sort is used to move them to the right position.



Time complexity:

- Best case: O(N)
- Average case: O(N+r)
- Worst case: O(N^2)

EXPERIMENTAL RESULTS: BOARDS OF DATA

Data order: Random												
Data size	10000		30000		50000		100000		300000		500000	
Resul-ting statics	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons
Heap	7	637987	19	2E+06	33	4E+06	85	8E+06	288	3E+07	589	5E+07
Selection	132	1E+08	1025	9E+08	3849	3E+09	15605	1E+09	117979	9E+10	335037	3E+11
Insertion	67	5E+07	678	5E+08	1965	1E+09	7962	6E+08	62045	4E+10	202747	1E+11
Counting	0	69988	0	209989	1	332758	4	632754	9	2E+06	12	3E+06
Radix	1	140056	5	510070	10	850070	20	2E+06	142	5E+06	99	9E+06
Flash	0	100510	1	296869	2	499792	7	936551	31	262406	42	4E+06
Merge	13	583690	36	2E+06	63	3E+06	123	7E+06	426	2E+07	660	4E+07
Shell	2	6E+06	6	2E+06	15	4E+06	33	1E+07	91	3E+07	258	7E+07
Quick	2	299566	4	964505	9	2E+06	18	4E+06	47	1E+07	193	2E+07
Bubble	792	2E+08	7469	2E+09	23686	5E+09	85376	2E+10	886226	2E+11	2E+06	5E+11
Shaker	902	9E+07	5384	8E+08	15226	2E+09	60935	9E+09	589369	8E+10	2E+06	2E+11

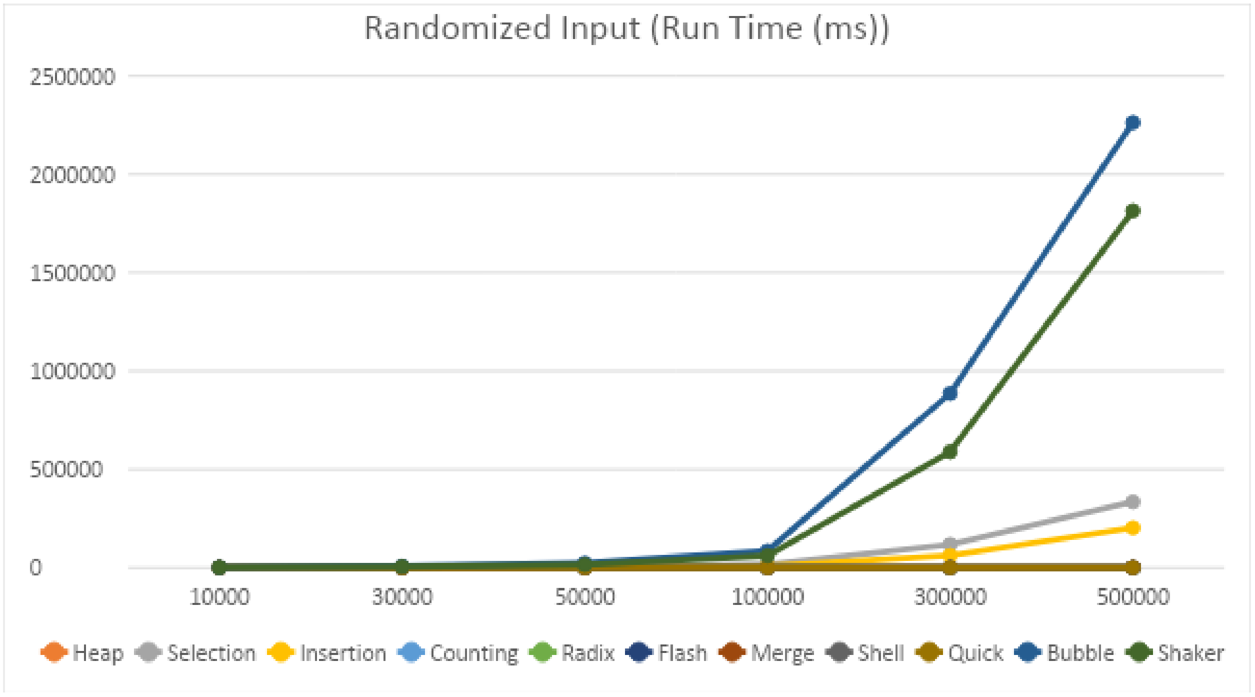
Data order: Sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resul-ting statics	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons	Running time	Compa-risons
Heap	8	670329	18	2E+06	32	4E+06	87	8E+06	262	3E+07	449	5E+07
Selection	135	1E+08	1183	9E+08	3245	3E+09	15618	14721	118564	9E+10	336232	3E+11
Insertion	0	29998	0	89998	0	149998	0	299998	1	899998	2	1E+06
Counting	0	60002	1	180002	2	300002	3	600002	6	2E+06	11	3E+06
Radix	2	140056	7	510070	11	850070	22	2E+06	81	6E+06	115	1E+07
Flash	1	127992	2	383992	3	639992	5	1E+06	13	4E+06	27	6E+06
Merge	10	475300	29	2E+06	60	3E+06	116	6E+06	273	2E+07	594	3E+07
Shell	0	360042	2	1E+06	6	2E+06	9	5E+06	26	2E+07	45	3E+07
Quick	0	154992	1	501950	3	980354	7	2E+06	14	8E+06	36	1E+07
Bubble	0	19998	0	59998	1	99998	0	199998	1	599998	2	999998
Shaker	0	20000	0	60000	0	100000	0	200000	1	600000	2	1E+06

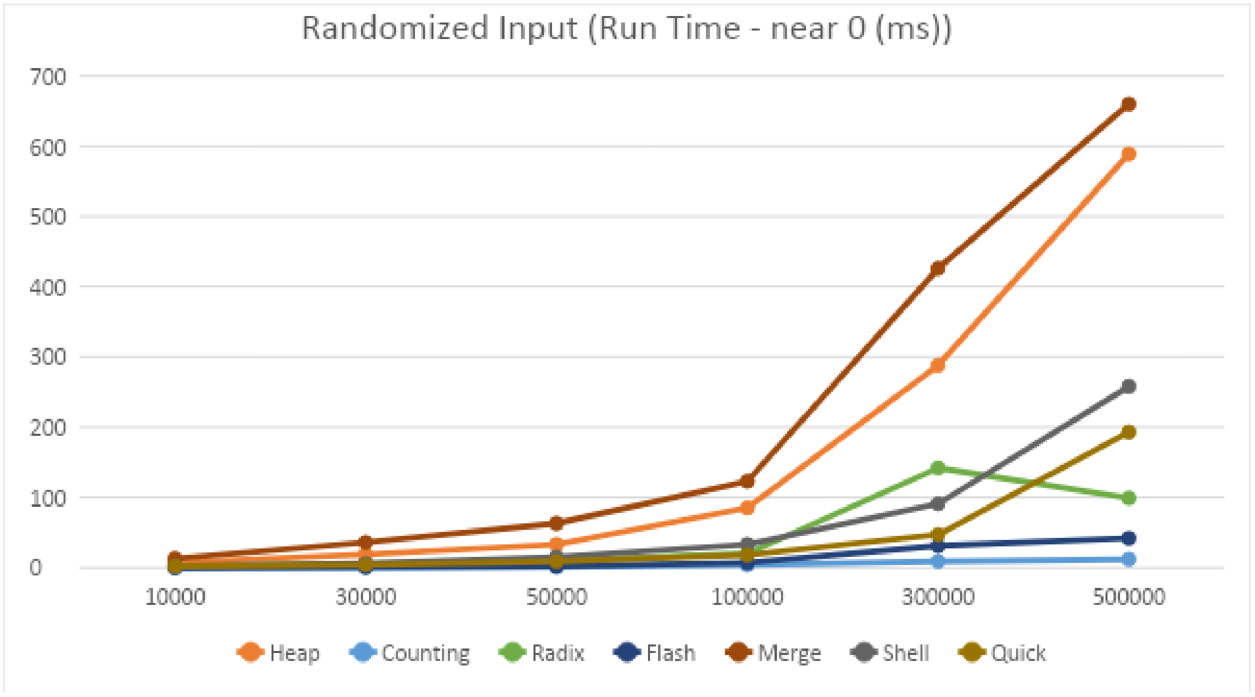
Data order: Nearly sorted												
Data size	10000		30000		50000		100000		300000		500000	
Resul-tin g statics	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons
Heap	6	606771	19	2E+06	28	4E+06	71	8E+06	257	3E+07	423	4E+07
Selection	126	1E+08	1114	9E+08	3053	3E+09		1E+09	125331	9E+10	318602	3E+11
Insertion	130	1E+08	1199	9E+08	3914	3E+09	15601	1E+09	140996	9E+10	392536	3E+11
Counting	0	70001	1	210001	2	350001	4	700001	8	2E+06	19	4E+06
Radix	2	140056	6	510070	12	850070	19	2E+06	105	6E+06	116	1E+07
Flash	0	110501	1	331501	2	552501	4	1E+06	16	3E+06	32	6E+06
Merge	10	496450	30	2E+06	65	3E+06	112	6E+06	281	2E+07	751	3E+07
Shell	1	475175	2	2E+06	8	3E+06	11	6E+06	31	2E+08	62	3E+07
Quick	1	164994	2	531960	4	963883	6	2E+06	16	6E+06	60	1E+07
Bubble	1030	2E+08	9784	2E+09	26873	5E+09	114505	2E+10	1E+06	2E+11	3E+06	5E+11
Shaker	1039	2E+08	9023	2E+09	25434	4E+09	102964	2E+10	1E+06	2E+11	3E+06	4E+11

Data order: Reversed												
Data size	10000		30000		50000		100000		300000		500000	
Resul-tin g statics	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons	Running time	Compa-r isons
Heap	8	669855	19	2E+06	32	4E+06	77	8E+06	278	3E+07	455	5E+07
Selection	133	1E+08	1189	9E+08	3284	3E+09	15624	1E+09	140590	9E+10	390213	3E+11
Insertion	1	173950	1	543914	1	664354	1	731246	2	1E+06	4	2E+06
Counting	0	67777	0	209671	1	330083	3	628123	6	2E+06	13	3E+06
Radix	0	140056	5	510070	12	850070	22	2E+06	77	6E+06	151	1E+07
Flash	0	127968	2	383962	2	639968	6	1E+06	14	4E+06	26	6E+06
Merge	12	501677	32	2E+06	60	3E+06	123	6E+06	268	2E+07	850	3E+07
Shell	1	402735	4	1E+06	6	2E+06	9	5E+06	23	2E+07	110	3E+07
Quick	0	155018	1	501990	4	913920	7	2E+06	14	6E+06	36	1E+07
Bubble	179	1E+08	2059	2E+09	3934	2E+09	5347	4E+09	20800	1E+10	24487	2E+10
Shaker	2	179976	4	1E+06	5	1E+06	7	3E+06	20	9E+06	31	1E+07

EXPERIMENTAL RESULTS: GRAPHS (RUN TIME)

Randomized data:





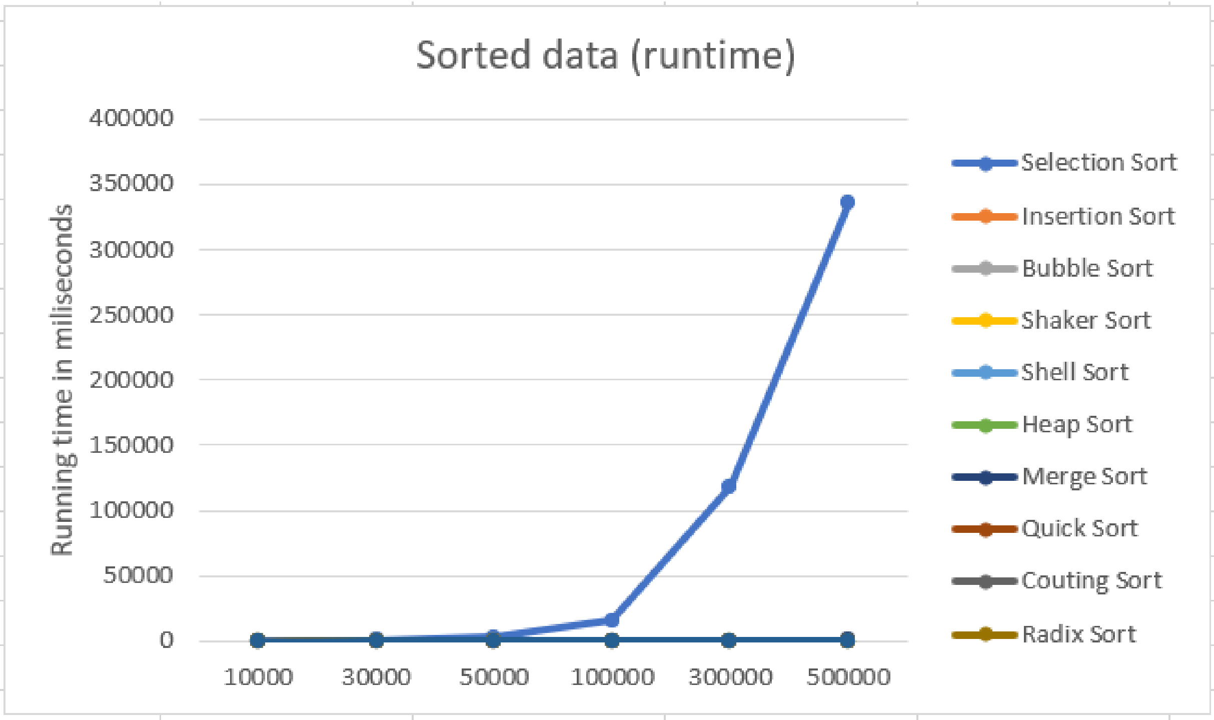
Comment:

The slowest of the most is the Bubble Sort (a close second is Shaker Sort), while the fastest is the Counting Sort.

Even though the Bubble Sort in this case has been improved by checking to stop the sorting if there's no swap in any iteration, there are still so many comparisons and swaps since the complexity is O(n^2), meanwhile the Counting Sort has O(n+k) as k is the range of the elements in the array which is much faster.

From the size of 100000 and up, the number of comparisons and the time it takes to complete Shaker Sort and Bubble Sort rocket, while the 5: Quick, Shell, Flash, Counting & Radix go up slowly in time and comparisons, and still be almost instant. Do note that it is unusual to see Radix Sort of size 500000 taking quite less time than itself of size 300000.

Sorted Data:

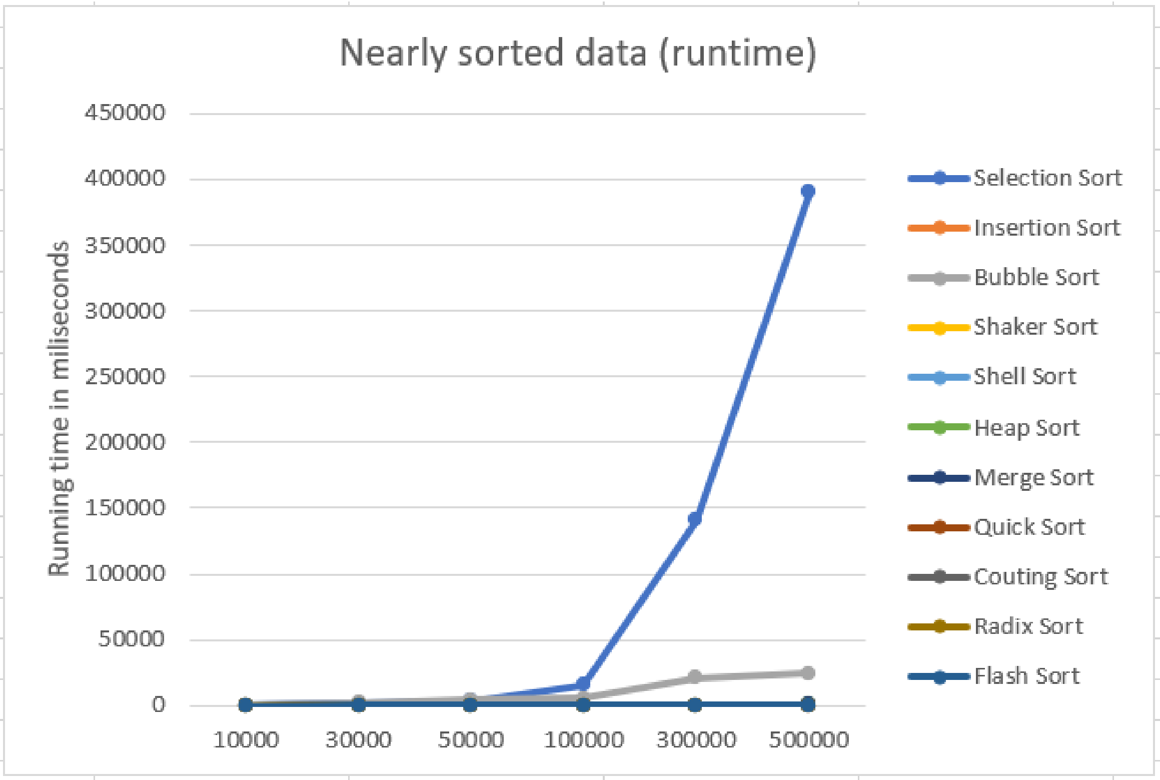


Comments:

- The most effective algorithm: Shaker sort and Insertion sort. These sorting algorithms have complexity O(n2), but with sorted input, it's stored in the memory with complexity O(1), and it’s stable and in this data type, both of them sort data in the best case (O(n)). So that, when sorting a sorted input data, it doesn’t waste too much time.

- The worse effective algorithm: Selection sort. It has a complexity O(n2) in every case, and it isn’t stable. When data size is bigger, the program also executes slower.

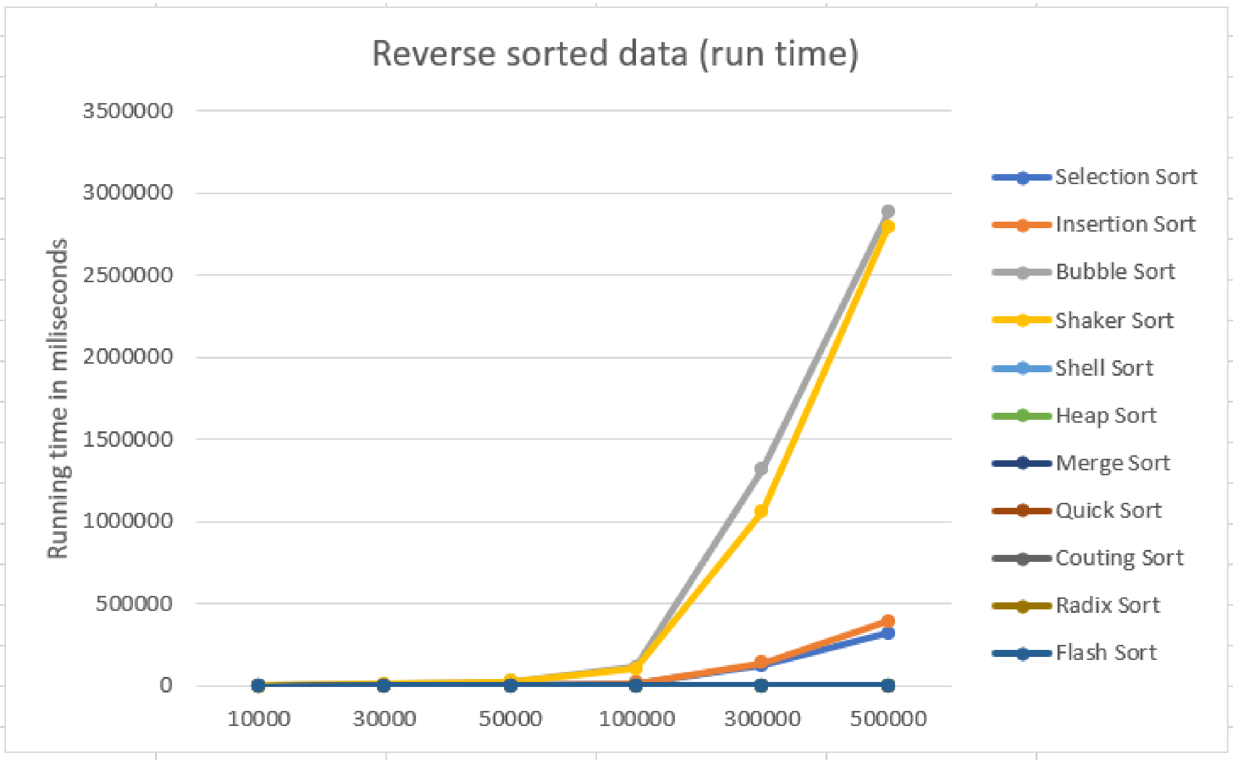
Nearly Sorted Data:



Comments:

- The most effective algorithm: Counting sort. The counting sort has a complexity O(n + k) for every case (best, worst, average). Inside of that, k = (max – min) + 1, in every time k changes, it’s always smaller than n, so that it doesn’t waste too much time to sort input with nearly sorted data.
- The worse effective algorithm: selection sort. The selection sort has a complexity O(n2) for every case. So, it’s easy to see that when the input data are given larger, the time program wastes more (n increase, time increase with squared of n).

Reverse Sorted Data:

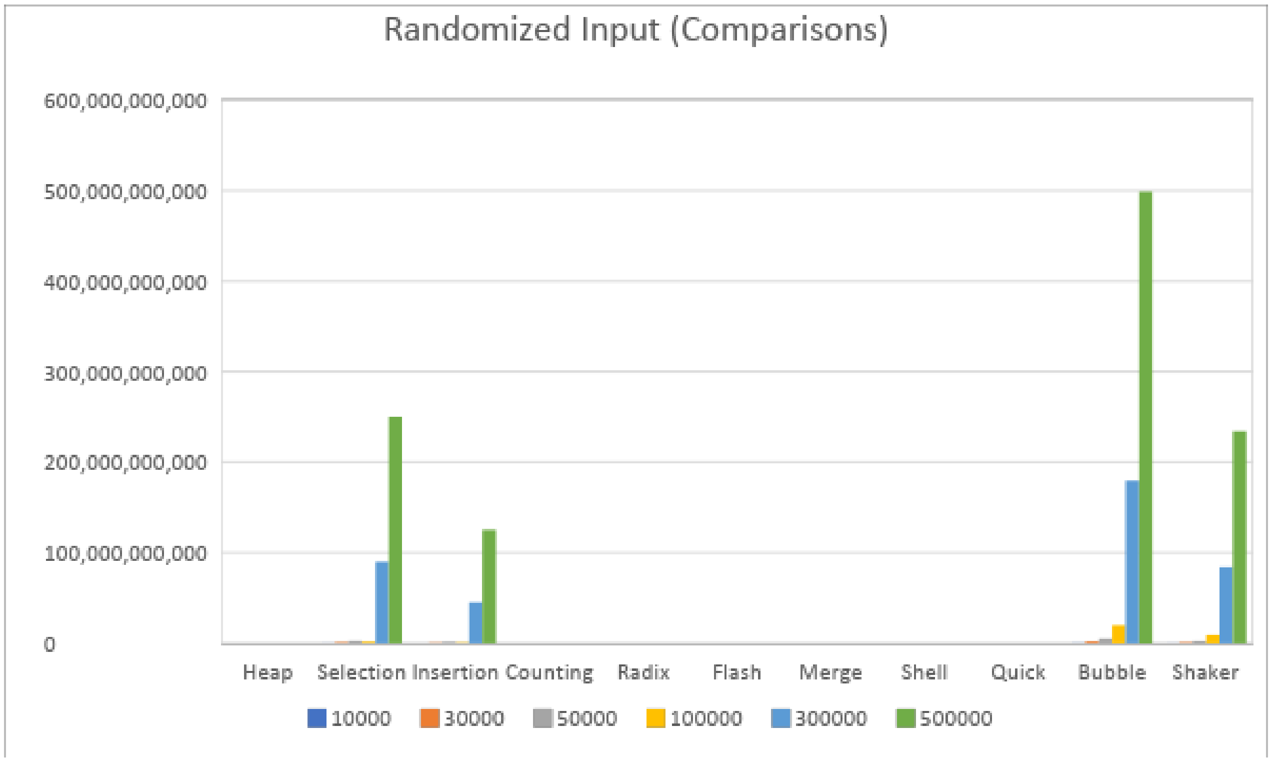


Comments:

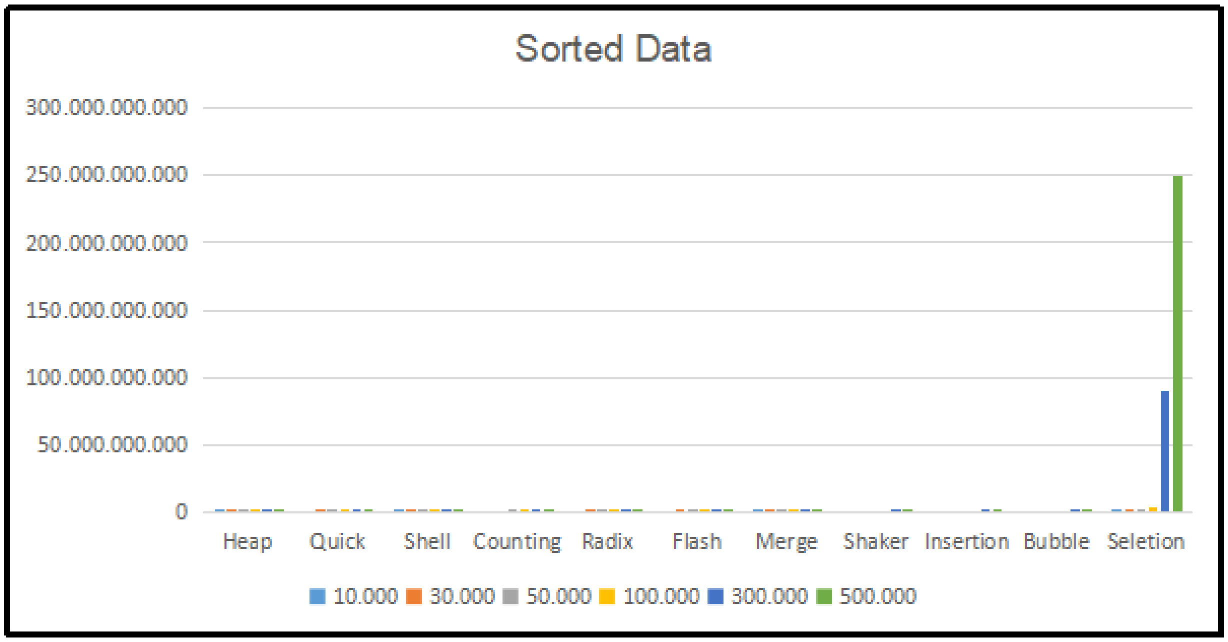
- The most effective algorithm: Counting sort, with this input data, we can approximately time sort by using counting sort which is equal to using flash sort. Counting sort has an unchangeable complexity in every case (O(n+k)). Flash sort has a complexity in the average case is O (n + r). And maybe, in this test, r is equal to k, although when we use data size 500.000, the time when using flash sort is larger, it isn't trivial.
- The worse effective algorithm: Bubble sort. The bubble sort has a complexity O(n2) in average and worst case. So, the larger input data is, the slower the time gets.

EXPERIMENTAL RESULTS: COMPARISONS

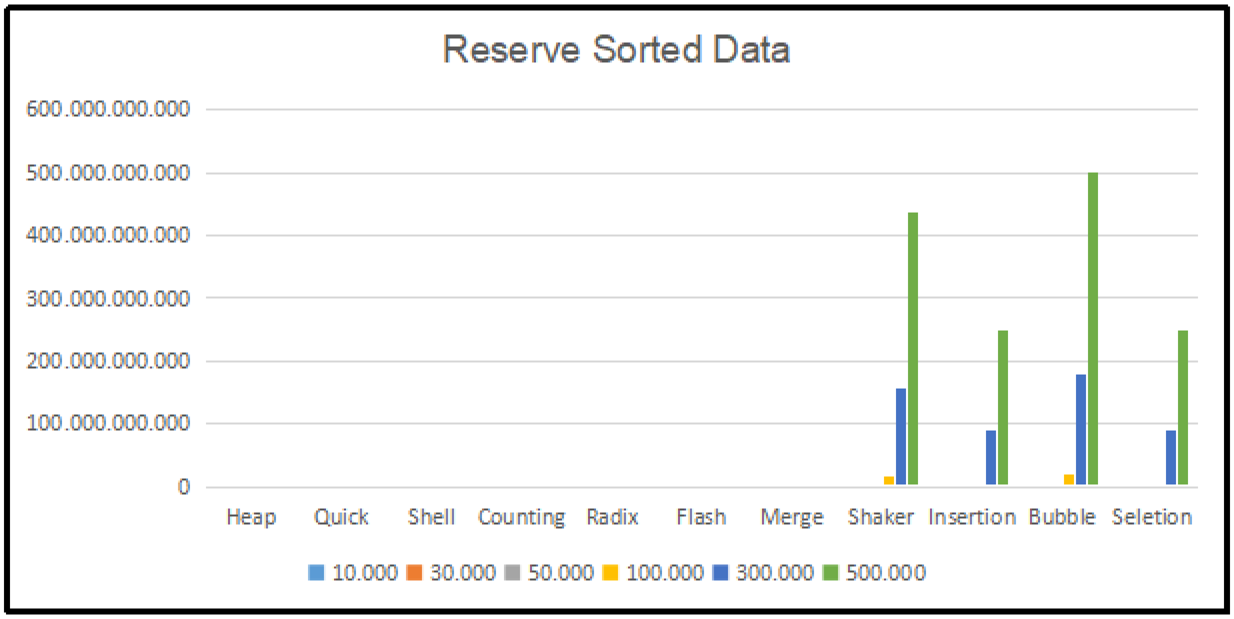
Randomized data:



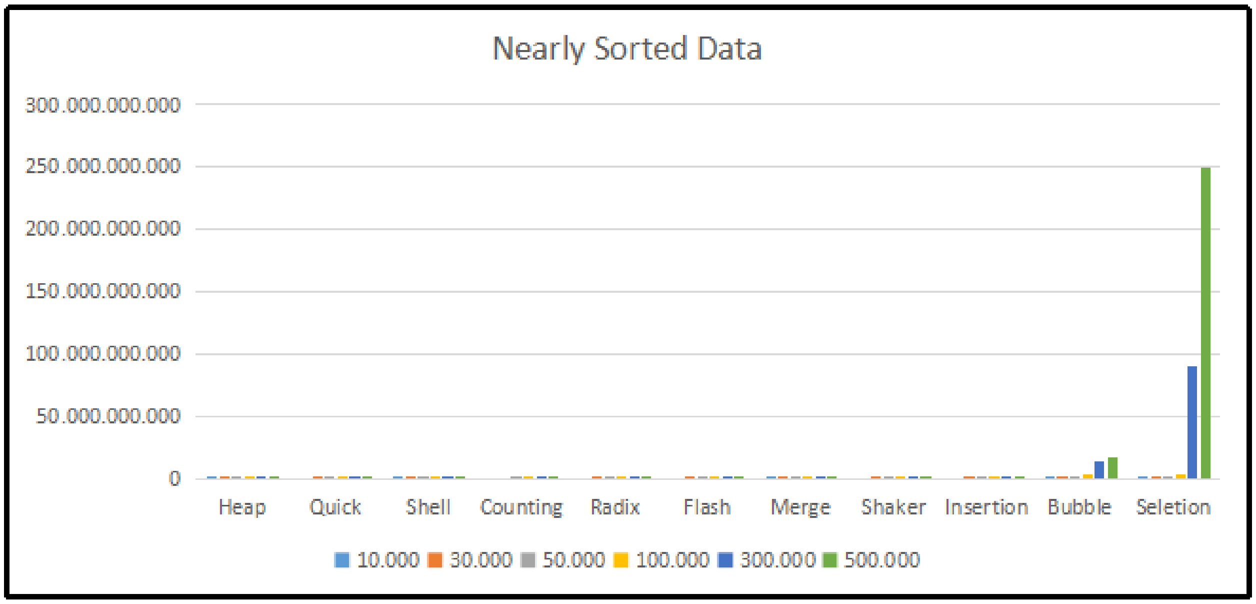
Sorted data:



Reversed data:



Nearly sorted data:



Comment

- **In the sorted data input case**, bubble sort and shaker (shaker sort is an improvement algorithm of bubble sort) are the most effective algorithms in this case. This effect is achieved thanks to the mechanism of action of bubble sort which does not need to compare the element which needs to be sorted with every single unsorted element.
  Opposite of bubble sort, selection sort needs to compare the element which needs to be sorted with every unsorted one to sort it.
- **In the reserved sorted data input case**, simple sorting algorithms show defects and instability of their mechanism. Nesting two loops inside each other leads to consequences that the program need to use n^n or even $2n^n$ comparisons (n is amount of elements need to be sorted) to complete the algorithm
- **In the nearly sorted data input case**, selection sort is still an algorithm needing most comparisons with the same reason as the sorted data input case.
- **In every case**, counting - quick - radix and flash sort are always show outstanding performances, they all have a step to divide elements into sub-group and sort this thing lead to the elements do not need to compare with all other elements and decrease total of comparisons need to use.
- **In addition**, the stable ones always preserve the relative order of equal elements. While unstable sorting algorithms don't. In other words, stable sorting maintains the position of two equal elements relative to one another.

EXPERIMENTAL RESULTS: OVERALL.

Data:

- With the size of data is smaller than 50000, the time difference is negligible.
- With input sorted data, the time we get to sort when using Bubble sort and Shaker sort is the fastest.
- Selection sort is worst in most cases, because of its complexity.
- Counting sort and Radix sort which are the algorithms have a low time when executed, but it wastes quite a lot of memory.
- Flash sort is one of the most effective algorithms.

Stability:

- The stable algorithms: Counting Sort, Quick Sort, Heap Sort, Radix Sort, Merge Sort, Shell Sort, Flash Sort.
- The unstable algorithms: Selection Sort, Insertion Sort, Shaker Sort, Bubble Short

PROGRAMMING NOTE

1) Files:
    a) main.cpp: This file contains the main functions for inputting, getting sorting results and giving outputs based on the commands.
    b) sortAlgorithms.h: This file contains the 11 sorting algorithms including 2 versions: count & run time. Do note there are some sorting algorithms that have recursive structure or multiple different functions.
    c) DataGenerator.h: This file contains the functions to generate random input based on the input size and commands required.
    d) support.h: This file contains all the library used in this project, and supporting functions such as swap to support the main code files above, make them shorter, simpler and easier to fix and debug.
2) Functions:
    a) int main(): The very core function of the program, takes the inputs then based on the inputs to decide which command the user wants the program to do.
    b) The command functions (commandline1, …) : Those take the inputs passed from the main(), do the sorting algorithms and display output on the terminal or pass output into files depending on the command used.
    c) sortRunTime() and sortComparison(): The 2 functions getting the sorting results (either running time in mili-seconds, or the number of comparisons done) from the inputs including the array (content and size), sorting algorithm,…
    d) GenerateData(): Generates the content of the array input based on its size and the array type required.
3) Library:
    a) <iostream>: Input and output stream, to get inputs from and display outputs onto the terminal.
    b) <time.h>: Provides the data structures and functions for the purpose of examining the running time of a block of codes.
    c) <string> & <cstring>: String type data and their related functions.
    d) <fstream>: Reading inputs from and writing outputs to files.

REFERENCE

Source code:

- *Quick Sort*
- *Shell Sort*
- *Heap Sort*
- *Flash Sort*
- *Counting sort*

Explanations on the algorithms: *Subject Data Structures and Algorithms. Lecture 3. Lecturer : Ngô Minh Nhựt*

Algorithms Descriptions

- Shell Sort
  - *Youtube*
  - *Wikipedia*
- Heap Sort : *Wikipedia*
- Quick Sort : *Wikipedia*
- *Flash sort:*
  - *Time complexity*
  - *Idea and Algorithm description : Teacher: Bui Huy Thong.*
- *Counting sort:*
  - *Time complexity*
  - *Space complexity*
  - *Idea and Algorithm description*

Determining complexity of algorithms : ***Algorithms Complexity***