CT Project 1819S2 JAVA

Written by Marco Men

Date: 09 – May -2019

Introduction

Concept of Sorting and Sorting Algorithms

Sorting can be defined in many ways.

Sorting is not restricted to technology, but also in day-to day (such as arranging and storing household items by relevance of items category -**grouping according to type or class**). When playing cards, the first thing we do when dealt a hand of cards is to sort them, first by deck and then ordering numbers. Both examples aid users in easy and quick access to items required… Plus in the second example, it speeds the analytical process.

In the study of algorithms one of the most common definition would be:

"…permutation(reordering) (a'1, a'2, a'3…,a'n)of the input sequence such that a'1<=a'2<=a'3<=…<=a'n.

Or, put into some well-defined order, 'arranging' systematically in groups. Also, known as "bucketing", "bucketizing", or "binning".

When sorting, the structure of data (array or linked list) and each of its elements must be considered. The collection of the data is called a **record.** The value to be sorted in the record is designated as a **key (or sort key).** All the other elements in the record (around the key) are known as **satellite data.** A few examples are given to best understand **record, key** and **satellite data:**

> In a phone book, the sort key would be the name, and the satellite data would be the address and phone number.
>
> In a search engine, the sort key would be the measure of relevance to the query and the satellite data would be the URL of the Web page, plus the data the search engine stores about the page.

Sorting involves permutating the keys and the satellite data.

In the case of a larger dataset (or record), the permutation is carried on an array of pointers to the records and not the records themselves. This is to avoid, or at least minimise data movements.

A sorting algorithm describes the method by which we determine the sorted order, either in sorting individual numbers or large records. The inputs to be sorted must be of the same category (numbers or letters in the case of lexicographical sorting or ordering-alphabetical), and not a mix.

Why is sorting important?

Well we already seen searching algorithms. There are significant differences between searching data in a sorted vs. unsorted data records, regarding complexity of time and space of the algorithms used. Sorting can help reduce both types of complexity.

Other usages of sorting would include efficiency improvements in an application that needs to sort information. Example, set-up and update customer (or users) records in let's say Facebook accounts. If sorting would not be used then perhaps new users would be assigned old accounts, passwords not recognised and the whole application would eventually run very slow and probably crash.

In computer graphics, objects are usually layered on top of each other. A program that renders objects on a screen might have to sort the objects according to an "above" relation so that it can draw these objects from bottom to top.

Sorting algorithms have been developed over the years to overcome many of the world's problems of its time. And they continue to be developed…

Let us assume a current world problem, such as global warming. Sorting could mean retrieve and analyse data, such as where are the highest levels of pollution. Place this information using some sort of ordering, from the cleanest (at the top) to the most polluted (at the bottom) locations in the world. By analysing both types of locations, world leaders could come up with some ideas to minimise the levels of environmental footprint.

Choosing the appropriate sorting algorithm for each specific situation requires a certain level of knowledge regarding keys and satellite data (data structure), caches and virtual memory (hardware-memory hierarchy) of the host computer and the software environment. I will briefly mention some of these in this project.

One of the objectives of this project is to benchmark 5 different sorting algorithms to see how they behave in time and complexity when sorting randomised/unsorted arrays with different number of inputs. My getRandomArray () method implemented with the algorithms seeks to (pseudo) generate integers with a range from 0 to 99, and some of my

input sizes will be higher than 100, some. In those instances, there will be duplication of integers. If the method generates the integers equally then, for an input size N=500 there should be 5 duplicates for each integer (N / range of numbers (which is a constant 100).

For input size(N) < 100, no duplicates generated.

Performance

The performance of sorting algorithms will depend if the algorithm uses comparisons or not.

Comparison sorting algorithms usually have their performance measured as O (n log n).

If the programmer knows something about the elements to be sorted in advance, there are faster ways for sorting. A hashing function can be used to partition a collection of elements into unique ordered buckets. A Bucket Sort can be then used for sorting, which is a linear performance algorithm (O(n)) for Best, Average and Worst

In – Place sorting

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list.

In-place sorting is the term to describe the

A sorting algorithm sorts in place if only a constant number of elements of the input array are ever stored outside the array. Selection and Insertion sorting are examples of in-place sorting algorithm.

Stable Sorting

When for example 2 elements in the original unordered collection are equal, maintaining their relative ordering in the sorted set may be important. Example, v1> v2, return v1, if v1 <= v2, return v2. 1<2, so in the event of both being equal return the element with the highest index. This attribute of sorting (maintain relative ordering by analysing the location of the elements in the original collection) is known as **stable sorting.**

Comparator Functions (cmp)

When talking about sorting algorithms, it is safe to assume that a comparator function can be provided. The main function of such comparator will be to compare 2 elements and return 0, if the elements are equal; a negative number, if element 1 < element 2; and a positive number, if element 1 > element 2.

In the case of complex records, the comparator might only compare the key for each element

Comparison – based Sorts

These can and are often viewed as decision trees (or a binary tree) where all the elements are paired with one another, compared and only one is returned to an upper level. Paired, compared and returned until there are not enough elements to pair with. Which will mean that there is only one desired element.

These algorithms can sort *n* numbers in O (n log n) time. Merge sort and heapsort achieve this in upper bound in the worst case. Quicksort in average bound.

To quote Thomas Cormen in his “Introduction to algorithms”:

“These algorithms share an interesting property: *the sorted order they determine is based only on comparisons between the input elements”*

Non – comparison-based Sorts

Bucket Sort, counting sort and radix sort are examples of sorting without comparisons. They tend to be faster than the comparison-based sorts as they can sort n elements in better than Θ (n log n) performance.

These use fast hashing to partition uniformly a collection of elements into distinct, ordered buckets. We must assume that each element is distinct, so comparisons like $n_a = n_b$ are unnecessary. Also, that all elements are in relative order, i.e. $n_a \leq n_b$

This will be shown on the Bucket Sort diagram depicted below.

Criteria For Sorting Algorithm Choice

To choose the appropriate sorting algorithm for a specific situation, the following questions should be asked:

- How big / small is the array/list to be sorted?
- Are the items almost sorted?
- Interested in worst – case scenarios or average cases?

- Are the items randomized and far apart from each other?
- How much code / time are we willing to write/spend?
- Is stable sort required?
- How and what will be the usage?

This list of questions is no way an exhaustive list, but it constitutes a good starting point. On the last question, I must point out the Netflix prize competition where several teams were tasked to write an algorithm that would increase the accuracy of the company's recommendation engine by 10 %. The winning team was awarded 1 million dollars (Belkor's Pragmatic Chaos) in 2009. But Netflix never used it, due to high engineering efforts and reduced deliverability.

Sorting Algorithms

Insertion Sort

This type of sorting uses an insert helper function that ensures M[0, i] is properly sorted. The insert starts in index position 1 compares the elements of M to the left. The swap occurs when required (when M[i] < M [0])

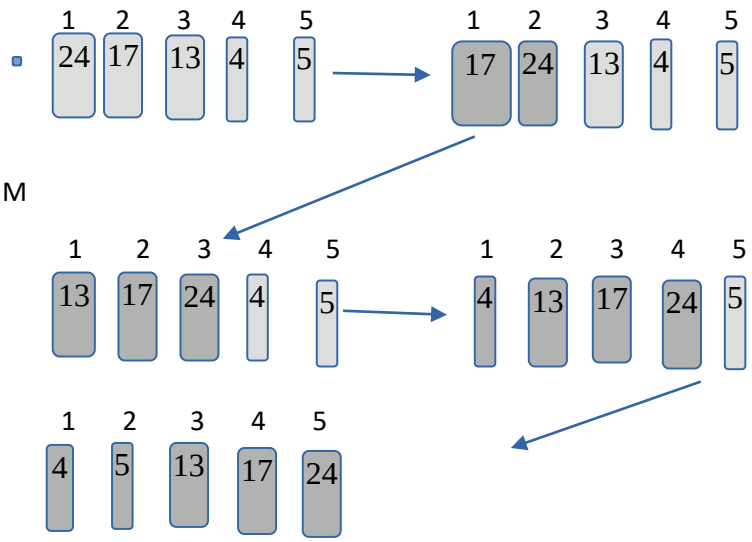
The insert reaches the last element of M when M is sorted

Insertion Sort is quite useful when sorting small sizes arrays or the elements of the array are nearly sorted. And in the best case occurs when the inner loop makes zero iterations each time. This occurs when M [i -1] ≤ M[i] each time the comparison executes. i.e. if the array M is already sorted and the outer loop iterates n – 1 times and takes constant amount of time and, so insertion sort takes $\Theta(n)$ time.

In the worst case, the inner loop makes the maximum possible number of iterations every time. i.e. The array M starts in reverse sorted order or *nonincreasing order*. In such scenario, each time the outer loop iterates, the inner loop iterates i – 1, because M[i-1] ≥ M[i], so the elements order must be reversed each time for all elements of the outer loop and it will take $\Theta(n^2)$ time.

In the diagram below, we can visualise how insertion sort works. The first 2 elements are compared, and the smallest shifted to the left. The shaded elements to the left are sorted to the leftmost position, moving one position to the right (outer loop) and comparing it to the sorted subarray to the left (inner loop).

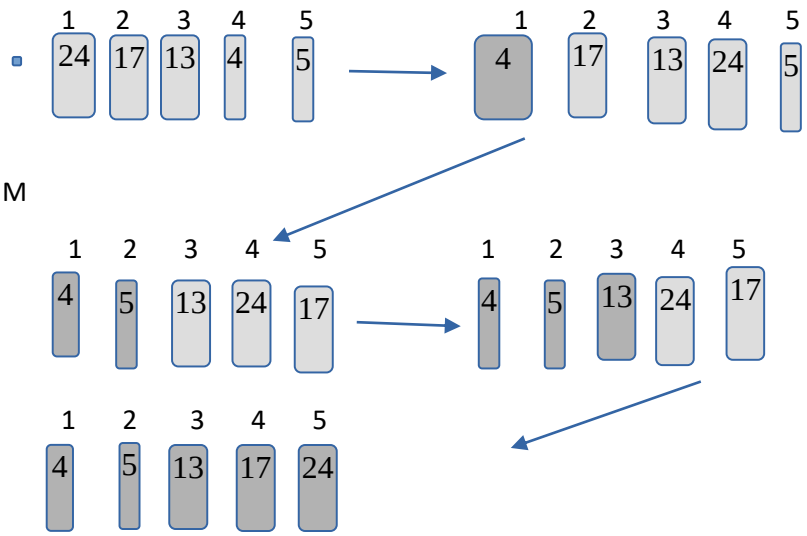
In terms of improved efficiency, this sorting algorithm performs best when there are duplicates in the array since there will be less swaps to perform. And is highly inefficient for value-based data because of the amount of memory that needs to be shifted to make room for a new value.



Selection Sort

This is not the fastest sorting algorithm, but it is quite useful when sorting elements with 2 parameters. Let us assume, books in a bookshelf that needs **sorting.** We can achieve this, by rearranging the books by author name first and then title. In such cases where books by the same author are present in the bookshelf, these would be sorted by title. A similar example with an array of numbers that repeat themselves. In such scenario, the selection sort would start from the highest value of the array *M* and swap its position with the rightmost M [*n – 1]* element. This process is then repeated until all the leftmost elements *n – 1* are sorted. Finding the smallest or highest value of *M* is a variant of linear search (seen in an earlier in this module). We must declare M[i] to be the smallest in the subarray (the darker shaded elements in the diagram below) so far, and then go through the rest of the subarray, updating the index of the smallest element every time a smallest element is found comparable to the current smallest until the last element of the array M is reached(n – 1). This process is done with nested loop iterations with inner loops being performed for each of the outer loop iterations (n^2).

This sorting algorithm is the slowest described in this project as it is quadratic in both, best and worse (and therefore average) times.



The running time of selection sort is Θ (n^2), and this is the same for O(n^2) and Ω(n^2) independently of the number of elements in the array.

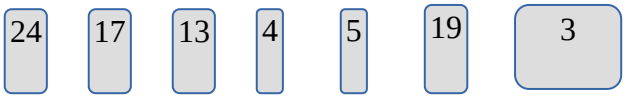
Both **Insertion** and **Selection** sorts are in-place sorting algorithms. These are also known as *transposition sorting.* Another sorting algorithm not discussed here, but worth mentioning that belongs to this group is **Bubble Sort**

Bucket Sort

Bucket Sort is a fast sorting algorithm as it assumes that the input is drawn from uniform distribution. This assumption is also present in counting sort (not mentioned in this project). Its average running time is O(n).

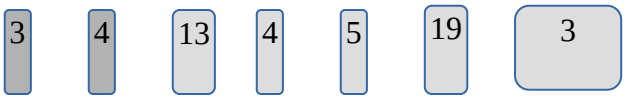
This type of sorting, divides into n smaller buckets of equal sized intervals and sorts each bucket in turn. The diagram below depicts,

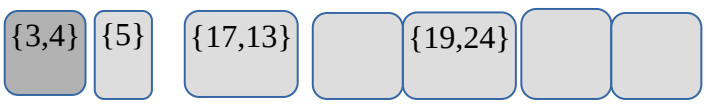
1)



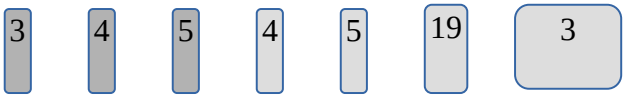
use hash(x) = x / 3

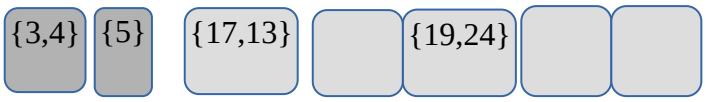
2)



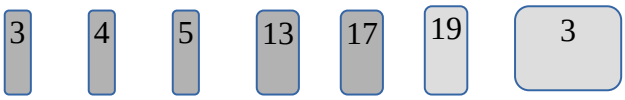


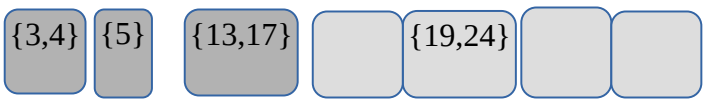
3)





4)



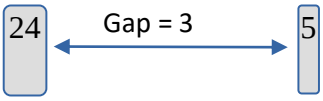


Shell Sort

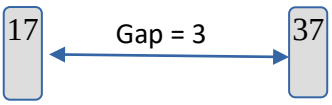
Shell sort was discovered in 1959, by Donald Shell as an improvement to insertion sort. It is a sub quadratic algorithm that works well in practice and easy to code.

The increased performance from insertion sort is due to avoidance of large data movement, by using comparisons of elements far apart, first. And then, elements less far apart. Shell sort uses an increment sequence like this $h_1, h_2, h_3, \ldots, h_t$, where $h_1 = 1$. After a phase using some increment h_k, we would have $M[i] \leq M[i + h_k]$ for every i where $i + h_k$ is a valid index. Every element spaced h_k are sorted. The array is said to be h_k sorted.

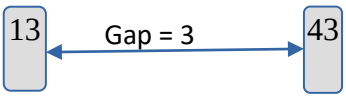
1)



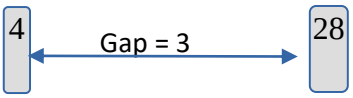
Swap positions, because 24 > 5.



17 and 37 are in right order. No swap

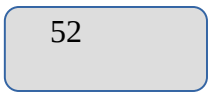


13 and 43 are in the right order. No swap

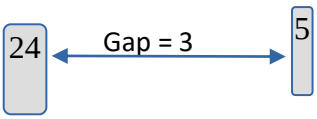


4 and 28 in the right order. No swap

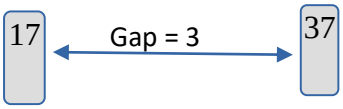
52 is on its own. Remains in position



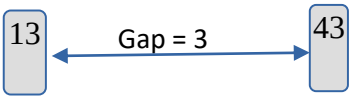
2)



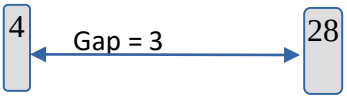
Swap positions, because 24 > 5.



17 and 37 are in the right order

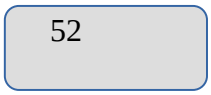


13 and 43 are in the right order. No swap



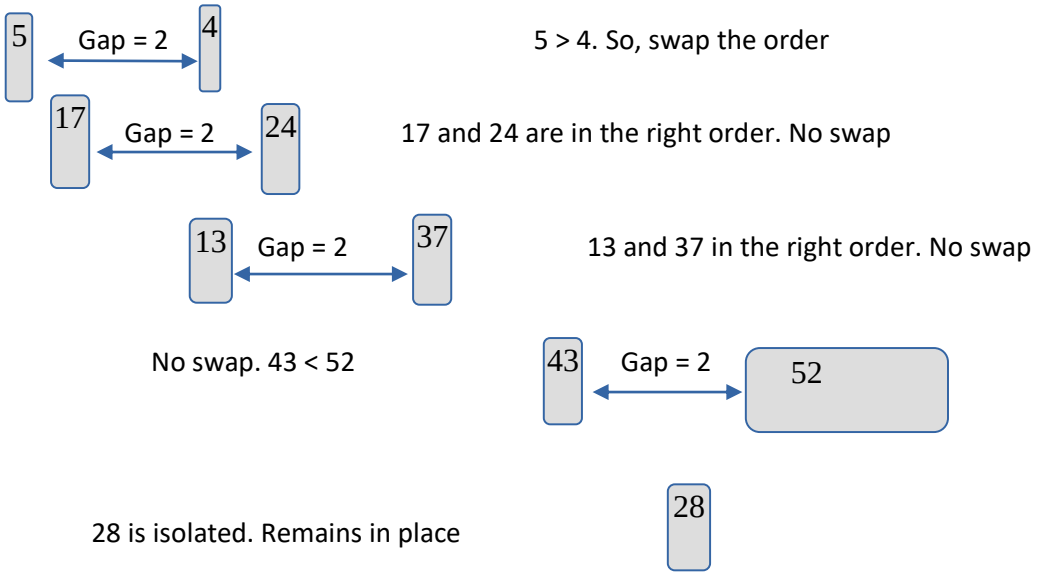
4 and 28 in the right order. No swap

52 is on its own. Remains in position



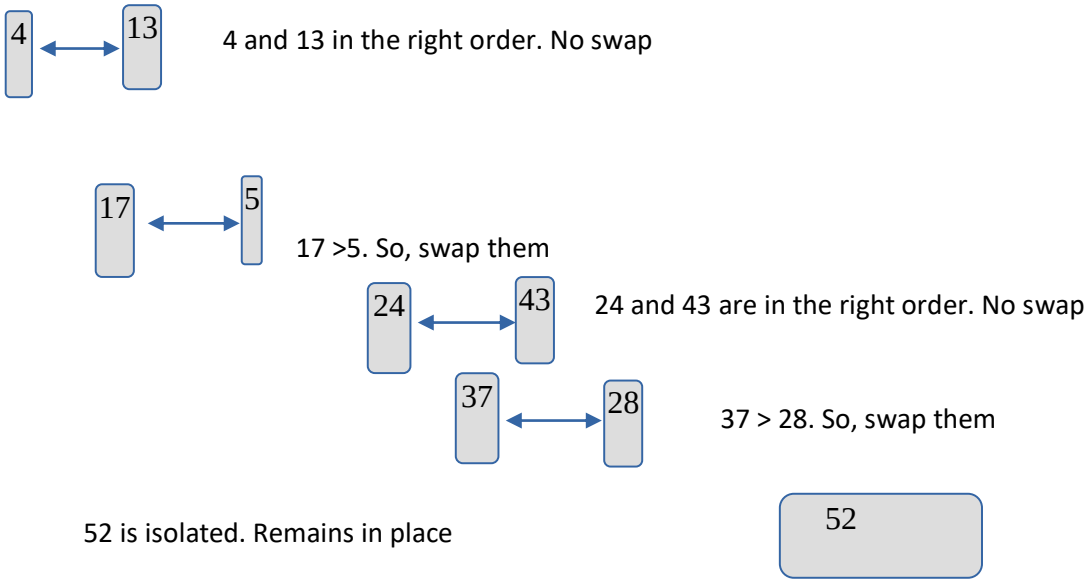
3)The array becomes

And now we reduce the gap to 2



4)The array becomes

And now we reduce the gap to 1



5)The array becomes

And now we perform an Insertion sort

6)The array becomes

7)The array becomes

8)The array becomes

9)The array becomes

Heap Sort

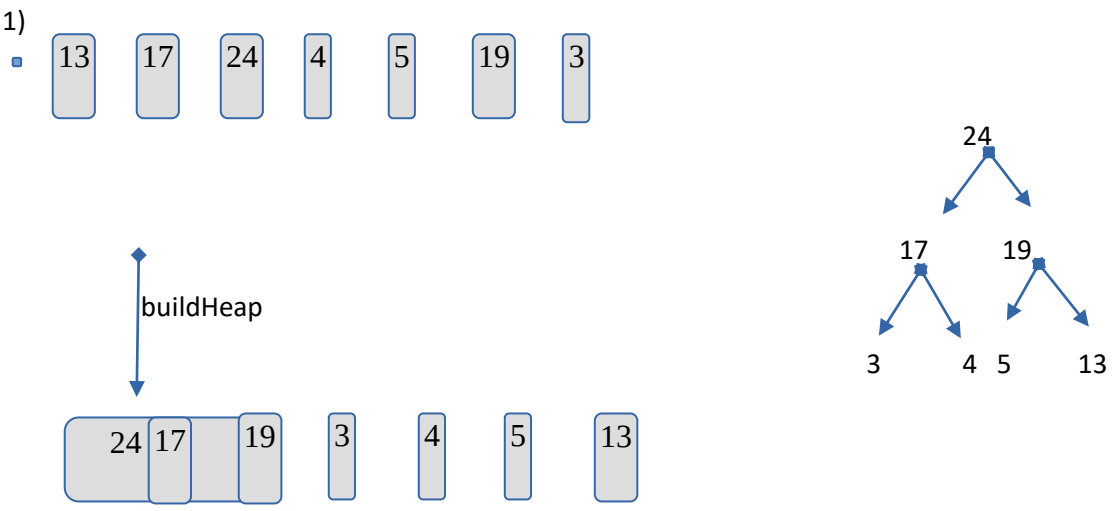
Heapify is a recursive operation that executes a fixed number of computations until the end of the heap is reached.

Heap sort is not a stable sort

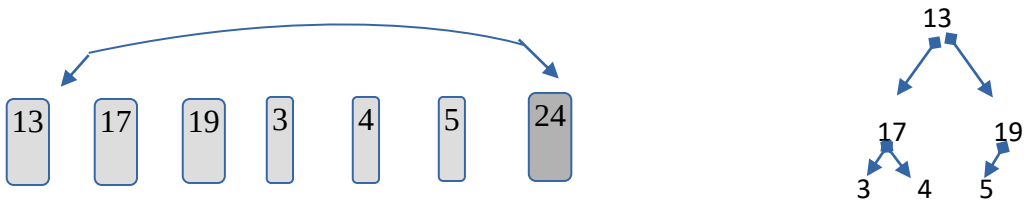
Heap sort sorts an array by first converting that array in place into a heap (by executing a buildHeap by heapifying the initial array from the midpoint ((n / 2) – 1)) down to index position 0. In the diagram below, Heap sort processes the array M of size n by treating it like two distinct subarrays split in the middle (M [0, m] and M[m,n]). Heap sort grows the sorted sub array M[i,n) downward by swapping the largest element in the heap (at position M[0]) with M[i]. After this swap, a reconstruction of M [0, i) is carried by executing heapify in order to make it a valid heap.

The resulting non-empty subarray M[i,n) will be sorted because the largest element in the heap represented in M[0,i), is guaranteed to be smaller than any element in the sorted subarray M[i,n). The diagram below best depicts the buildHeap and heapify processes until the array is sorted. With the rightmost index elements being sorted and locked and the pointer or key shifts one position to the left. The heap is built by swapping the highest value of the unsorted array to index position 0. M [0] is swapped with M[i], if M [0] > M[i]. Otherwise M [0] will remain in its position and M[i] locked and updated by shifting one position to the left. And this process will be repeated until the array is sorted

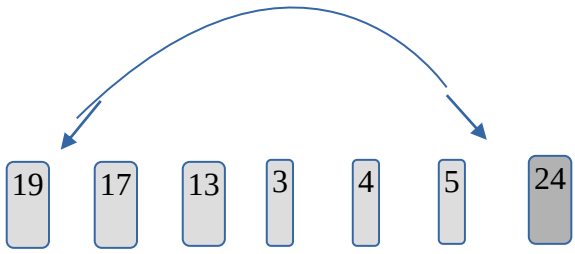
1)

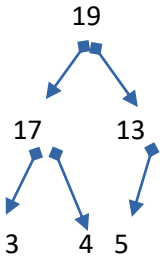


2) Not a Heap

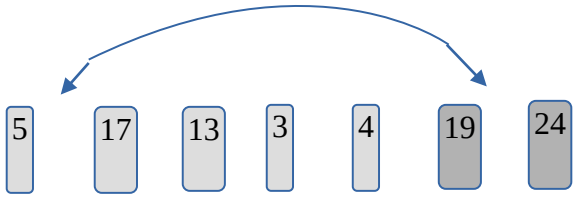


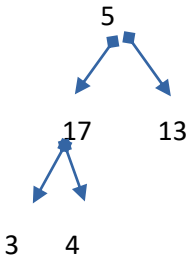
3)Heap Again



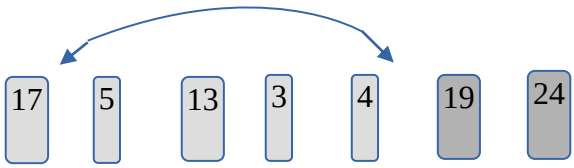


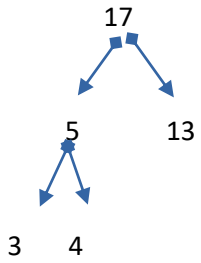
4)Not a Heap



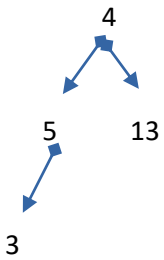


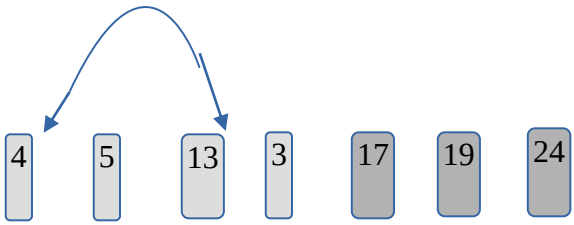
5)Heap Again



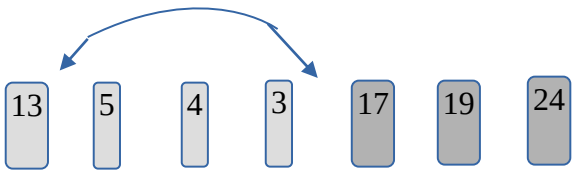


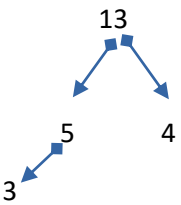
6)Not a Heap



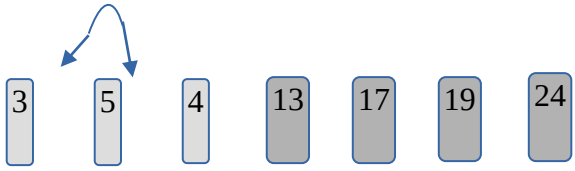


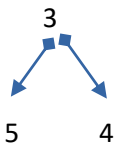
7)Heap Again



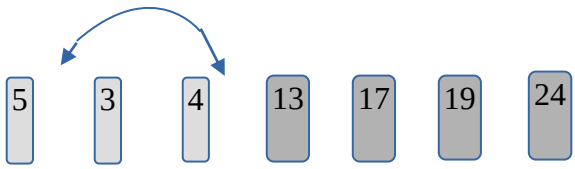


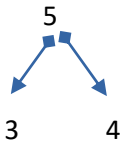
8)Not a Heap



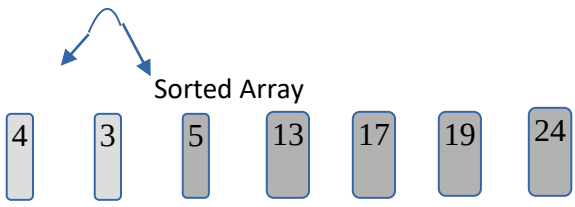


9)Heap Again





Heap Again



Sorted Array

Implementation & Benchmarking

Analysis Of The Results

Just by looking at the line plots, we can observe that Bucket sort, Shell sort and Heap sort present linear time complexity. Bucket sort outperforms all sorting algorithms as the input size N increases with the lowest runtimes and it will be the case if we append more results. The second fastest is Shell sort.

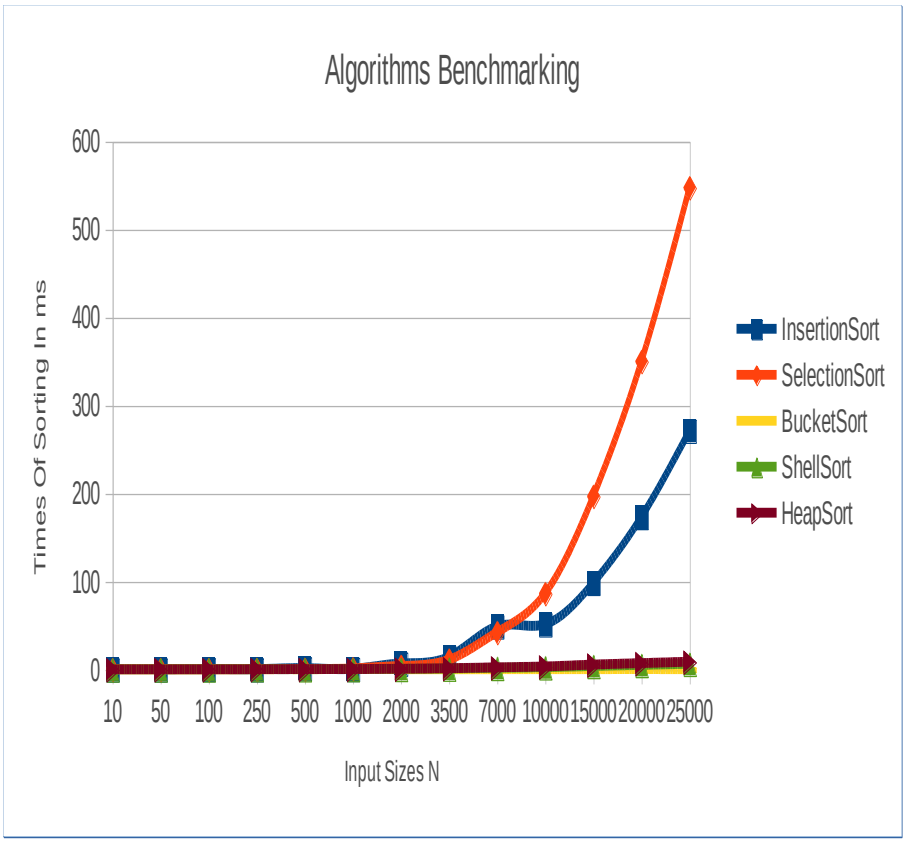
And the slowest is Selection sort which takes $O(n^2)$ in all cases independent of the input sizes and Insertion sort is linear in the best case with lower input sizes.

	Best Case	Average Case	Worst Case	Stability
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	YES
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	NO
Bucket Sort	O(n)	O(n)	O(n)	YES
Shell Sort	O(n)	O(n)	$O(n^2)$	NO
Heapsort	O (n log n)	O (n log n)	O (n log n)	NO

Current TimeStamp is: 2019-05-07 10:56:05.488

Average Times In Milliseconds

Size	10	50	100	250	500	1000	2000	3500	7000	10000	15000	20000	25000
InsertionSort	0.016	0.041	0.439	0.197	1.375	0.591	6.656	14.401	48.914	51.287	98.389	173.229	271.172
SelectionSort	0.008	0.08	0.09	0.295	0.4	1.069	3.821	10.956	42.215	86.388	196.918	350.186	547.563
BucketSort	0.007	0.011	0.015	0.028	0.049	0.447	0.158	0.051	0.103	0.153	0.192	0.224	0.158
ShellSort	0.006	0.035	0.079	0.217	0.19	0.342	0.6	0.937	1.717	2.464	3.784	5.008	6.325
HeapSort	0.013	0.03	0.053	0.122	0.243	0.326	0.688	1.278	2.267	3.285	5.383	7.061	8.708



Build of the Java Application

The java application called Benchmark class is the class that invokes all the other classes simultaneously and in a specific order. The order of each class and method being called is irrelevant, except for the Insertion Sort class. This needs to be first, because of the header rows, and current timestamp printout display. The reason why I did this was first, because of the way I started with each algorithm benchmark one by one and implementing functions and assigning variables on similar classes can lead to confusion. So, by keeping it neat, it would facilitate on testing and debugging if so required. Plus, on presentation and integration of the application I believe it is better to keep each class on its own and have one class that invokes all the classes and methods separated. All my classes do not print to the console. Instead they append the results on "Output.txt" and a message is displayed on the console when it is done. Also, a spreadsheet entitled "Benchmark.xls", which is the file I used to copy & paste my output strings and plot the multi-line plots is included in the submission folder.

A note on plagiarism if I missed any referencing accidently, all my sorting algorithms are from either "GeekForGeeks" or "w3resource" websites. I used information from the following websites

Literature Review

Algorithms in a nutshell (2nd Edition) – George T. Heineman, Gary Pollice & Stanley Selkow

Introduction to algorithms (3rd Edition) – Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein

Algorithms Unlocked – Thomas H. Cormen

Data structures and Problem solving in Java (Mark Allen Weiss)

Data Structure and Algorithm Analysis Using Java (4th Edition) - Mark Allen Weiss

Data Structures & Algorithms (6th Edition) - Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

Websites Used in Research:

https://openjdk.java.net/projects/code-tools/jmh/

https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/

https://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html

http://lessthanoptimal.github.io/Java-Matrix-Benchmark/manual/MethodologyRuntimeBenchmark/

https://www.youtube.com/watch?v=7af_QJiLWHI