**“COMPLEXITY ANALYSIS OF SORTING ALGORITHMS”**

***A***

***Project Report***

*submitted in partial fulfillment of the*

*requirements for the award of the degree of*

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE & ENGINEERING**

**by**

|  |  |
| --- | --- |
| **Name** | **Roll No.** |
| **Rakshit Garg** | **R103215057** |
| **Jayant Arora** | **R103215030** |
| **Riteyu S.Anand** | **R103213066** |
|  |  |

***under the guidance of***

**Mr.Vishal Kaushik**

****

**Department of Analytics**

**School of Computer Science and Engineering**

**University of Petroleum & Energy Studies**

**Bidholi, Via Prem Nagar, Dehradun, UK**

**2017**

****

**CANDIDATE’S DECLARATION**

We hereby certify that the project work entitled **“ Complexity Analysis of Sorting Algorithms”** in partial fulfilment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING with specialization in Business Analytics and Optimization and submitted to the Department of Computer Science & Engineering at Center for Information Technology, University of Petroleum & Energy Studies, Dehradun, is an authentic record of our work carried out during a period from **August**, **2017** to **December**, **2107** under the supervision of **Mr. Vishal Kaushik , Assistant Professor and College proctor.**

The matter presented in this project has not been submitted by me/ us for the award of any other degree of this or any other University.

**Rakshit garg Jayant Arora Riteyu S.Anand**

**Roll No.57 Roll No.30 Roll No.66**

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: 9th November 2017 **(Mr. Vishal Kaushik)**

Project Guide

**Dr. T.P Singh**

Program Head – Department of Analytics

Center for Information Technology

University of Petroleum & Energy Studies

Dehradun – 248 001 (Uttarakhand)

**ACKNOWLEDGEMENT**

We wish to express our deep gratitude to our guide **Mr. Vishal Kaushik** , for all advice, encouragement and constant support he has given us through out our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thank to our respected Program Head of the Department,

**Dr. T.P Singh** , for his great support in doing our project in **Area (like Network, Big data etc.)** at **SoCSE**.

We are also grateful to **Dr. S.K Banerjee, Associate Dean** and

**Dr. Kamal Bansal, Dean** **CoES** , UPES for giving us the necessary facilities to carry out our project work successfully.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Rakshit Garg** | **Jayant Arora** | **Riteyu S.Anand** |  |
| **Roll No.** | **R103215057** | **R103215030** | **R103213066** |  |

**ABSTRACT**

The Project Title is Complexity Analysis of sorting algorithms. An essential aspect to data structures is algorithms. Data structures are implemented using algorithms. An algorithm is a procedure that you can write as a C function or program, or any other language. It states explicitly how the data will be manipulated. Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency. The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process. For this project we will do the time complexity analysis for selection sort, insertion sort, merge sort, bubble sort, quick sort ,heap sort , radix sort , counting sort , shell sort ,bucket sort .

**TABLE OF CONTENTS**

**S.No. Contents Page No**

1. **Introduction 1**
   1. History 2
   2. Literature Review 2
   3. Classification 3
   4. Why algorithms 5
   5. Why C? 5
   6. Parameters 5
   7. Time Complexity 6
   8. Big-O notation 6
2. **Algorithms** 
   1. Insertion Sort 8

2.2.Quick Sort 8

2.3.Bubble Sort 9

2.4.Bucket Sort 10

2.5.Selection Sort 11

2.6.Shell Sort 12

2 .7.Counting Sort 12

2.8.Heap Sort 13

2.9.Radix Sort 13

2.10.Merge Sort 14

**3.USE CASE Diagram 15**

**4.CLASS Diagram 16**

**5.Implementation 17**

**1.INTRODUCTION**

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the use of other algorithms (such as search and merge algorithms) which require input data to be in sorted lists; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

The output is in non decreasing order (each element is no smaller than the previous element according to the desired total order);

The output is a permutation (reordering but with all of the original elements) of the input.

Further, the data is often taken to be in an array, which allows random access, rather than a list, which only allows sequential access, though often algorithms can be applied with suitable modification to either type of data.

It is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, sometimes called a list, and outputs a sorted array. In other words, a sorted array is an array that is in a particular order. For example, (a ,b ,c ,d) is sorted alphabetically, (1,2,3,4,5) is a list of integers sorted in increasing order, and (5,4,3,2,1) is a list of integers sorted in decreasing order.

A sorting algorithm takes an array as input and outputs a permutation of that array that is sorted. All sorting algorithms share the goal of outputting a sorted list, but the way that each algorithm goes about this task can vary. When working with any kind of algorithm, it is important to know how fast it runs and in how much space it operates — in other words, its time complexity and space complexity. The running time describes how many operations an algorithm must carry out before it completes. The space complexity describes how much space must be allocated to run a particular algorithm. For example, if an algorithm takes in a list of size n , and for some reason makes a new list of size n for each element in n, the algorithm needs n^2 space.

* 1. **HISTORY**

From the beginning of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. Among the authors of early sorting algorithms around 1951 was Betty Holberton (née Snyder), who worked on ENIAC and UNIVAC. Bubble sort was analyzed as early as 1956. Comparison sorting algorithms have a fundamental requirement of Ω(n log n) comparisons (some input sequences will require a multiple of n log n comparisons); algorithms not based on comparisons, such as counting sort, can have better performance. Although many consider sorting a solved problem—asymptotically optimal algorithms have been known since the mid-20th century—useful new algorithms are still being invented, with the now widely used Timsort dating to 2002, and the library sort being first published in 2006.

Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures such as heaps and binary trees, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and upper and lower bounds.

* 1. **LITERATURE REVIEW**

In computer science, sorting is an essential work for many applications towards searching and locating a prominent number of data. General description of sorting believed to be the process of rearranging the data into a particular order. The orders used are either in numerical order or lexicographical order. Sorting arranges the integer data into increasing or decreasing order and an array of strings into alphabetical order. It may also be called as ordering the data. Sorting is considered as one of the most fundamental tasks in many computer applications for the reason that searching a sorted array or list takes less time when compared to an unordered or unsorted list. There have been many attempts made to analyze the complexity of sorting algorithms and many interesting and good sorting algorithms have been proposed. There are more advantages in the study of sorting algorithms in addition to understanding the sorting methods. These studies have gained a significant amount of power to solve many other problems. Even though sorting is one of the extremely studied problems in computer science, it remains the most general integrative algorithm problem in practice. Moreover, selecting a good sorting algorithm depending upon several factors such as the size of the input data, available main memory, disk size, the extent to which the list is already sorted and the distribution of values. To measure the performance of different sorting algorithm we need to consider the following facts such as the number of operations performed, the execution time and the space required for the algorithm . Since sorting algorithms are common in computer science, some of its context

contributes to a variety of core algorithm concepts such as divide-and-conquer algorithms, data structures, randomized algorithms, etc. The majority of an algorithm in use have an algorithmic efficiency of either O(n^2) or O(n log n).

* 1. **CLASSIFICATION**

Sorting algorithms are often classified by:

1. Computational complexity (worst, average and best behavior) in terms of the size of the list (n). For typical serial sorting algorithms good behavior is O(n log n), with parallel sort in O(log2 n), and bad behavior is O(n2). (See Big O notation.) Ideal behavior for a serial sort is O(n), but this is not possible in the average case. Optimal parallel sorting is O(log n). Comparison-based sorting algorithms need at least Ω(n log n) comparisons for most inputs.
2. Computational complexity of swaps (for "in-place" algorithms).

Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in-place". Strictly, an in-place sort needs only O(1) memory beyond the items being sorted; sometimes O(log(n)) additional memory is considered "in-place".

1. Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
2. Stability: stable sorting algorithms maintain the relative order of records with equal keys (i.e., values).
3. Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
4. General method: insertion, exchange, selection, merging, etc. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort. Also whether the algorithm is serial or parallel. The remainder of this discussion almost exclusively concentrates upon serial algorithms and assumes serial operation.
5. Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be adaptive.

**1.4 Why sorting algorithms?**

Most languages already have a sort method implemented in standard libraries, so why learn sorting algorithms?

Learning sorting algorithms teaches you algorithm design. You will learn how to use paradigms like divide and conquer, how to measure algorithm complexity and when to use different sorting algorithms to maximize efficiency.

**1.5 Why C?**

C is a low-level high-level language.It's close enough to the metal that we still need to consider memory management. Being able to see how we manage memory in the algorithm makes it easier to calculate the running time.At the same time, it's a high-level language, so the syntax is human readable.

Finally, a lot of languages are based on C, so the syntax will be familiar to many developers.

**1.6 PARAMETERS**

The parameter names used throughout the algorithms

arr - Array to be sorted.

Note: Arrays in C are passed by reference

n - Number of elements in arr.

**1.7 TIME COMPLEXITY**

Time complexity is a way of measuring how long an algorithm takes to run.

It's important to consider how long an algorithm takes to run. A poorly written algorithm could take tens of thousands of times longer to finish than a well written algorithm.

But measuring the running time of an algorithm can be tricky.

We can't measure an algorithm absolutely (i.e. in milliseconds), because the amount of time it takes will vary between hardware and between runs. An algorithm could take 1200ms to complete on my machine, and 4000ms on a slow computer.

Instead, we measure the relative running time.

To do this, we count the number of operations performed in the algorithm.

The most common way to express time complexity is with big O notation.

**1.8 BIG – O NOTATION**

Big O notation measures the relative time an algorithm takes to run given the worst case.

Big O only cares about the big operations.

Generally we only care about the rough running time, so constant actions aren't included in the calculation.

Here's a table to use as a reference when looking through the algorithms in this project:

| **Big-O** | **Name** | **Description** |
| --- | --- | --- |
| **O(1)** | constant | **This is the best.** The algorithm always takes the same amount of time,  regardless of how much data there is.  Example: looking up an element of an array by its index. |
| **O(log n)** | logarithmic | **Pretty great.** These kinds of algorithms halve the amount of data  with each iteration. If you have 100 items,  it takes about 7 steps to find the answer. With 1,000 items,  it takes 10 steps. And 1,000,000 items only take 20 steps.  This is super fast even for large amounts of data.  Example: binary search. |
| **O(n)** | linear | **Good performance.** If you have 100 items, this does 100 units of work. Doubling the number of items makes the algorithm take exactly twice as  long (200 units of work). Example: sequential search. |
| **O(n log n)** | "linearithmic" | **Decent performance.** This is slightly worse than linear but not too  bad. Example: the fastest general-purpose sorting algorithms. |
| **O(n^2)** | quadratic | **Kinda slow.** If you have 100 items, this does 100^2 = 10,000 units of  work. Doubling the number of items makes it four times slower  (because 2 squared equals 4).  Example: algorithms using nested loops, such as insertion sort. |
| **O(n^3)** | cubic | **Poor performance.** If you have 100 items,  this does 100^3 = 1,000,000 units of work.  Doubling the input size makes it eight times slower.  Example: matrix multiplication. |
| **O(2^n)** | exponential | **Very poor performance.** You want to avoid these kinds of algorithms,  but sometimes you have no choice. Adding just one bit to the  input doubles the running time.  Example: traveling salesperson problem. |
| **O(n!)** | factorial | **Intolerably slow.** It literally takes a million years to do anything. |

**2 ALGORITHMS**

Bubble Sort

Insertion Sort

Quick Sort

Radix Sort

Bucket Sort

Heap Sort

Counting Sort

Selection Sort

Merge Sort

Shell Sort

* 1. **INSERTION SORT**

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shellsort is a variant of insertion sort that is more efficient for larger lists.

ALGORITHM –

// Sort an arr[] of size n

insertionSort(arr, n)

Loop from i = 1 to n-1.

……a) Pick element arr[i] and insert it into sorted sequence arr[0…i-1]

TIME COMPLEXITY –

Best Case- n

Average Case – n^2

Worst Case – n^2

ADVANTAGE –Relatively Simple and easy to implement, twice faster than bubble sort

DISADVANTAGE- inefficient for large lists

* 1. **QUICK SORT**

Quicksort is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected . All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. This yields average time complexity of O(n log n), with low overhead, and thus this is a popular algorithm. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries.

The important caveat about quicksort is that its worst-case performance is O(n2); while this is rare, in naive implementations (choosing the first or last element as pivot) this occurs for sorted data, which is a common case. The most complex issue in quicksort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower O(n2) performance, but good choice of pivots yields O(n log n) performance, which is asymptotically optimal. For example, if at each step the median is chosen as the pivot then the algorithm works in O(n log n). Finding the median, such as by the median of medians selection algorithm is however an O(n) operation on unsorted lists and therefore exacts significant overhead with sorting. In practice choosing a random pivot almost certainly yields O(n log n) performance.

ALGORITHM –

/\* low --> Starting index, high --> Ending index \*/

quickSort(arr[], low, high)

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); // Before pi

quickSort(arr, pi + 1, high); // After pi

}

}

TIME COMPLEXITY –

Best Case – n log n

Average Case – n log n

Worst Case – n^2

ADVANTAGE – Fast and efficient

DISADVANTAGE- Show horrible result if the list is already sorted

* 1. **BUBBLE SORT**

Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average time and worst-case performance is O(n^2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2n time**.**

ALGORITHM –

Example:

First Pass:

( 5 1 4 2 8 ) –> ( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.

( 1 5 4 2 8 ) –> ( 1 4 5 2 8 ), Swap since 5 > 4

( 1 4 5 2 8 ) –> ( 1 4 2 5 8 ), Swap since 5 > 2

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

( 1 4 2 5 8 ) –> ( 1 4 2 5 8 )

( 1 4 2 5 8 ) –> ( 1 2 4 5 8 ), Swap since 4 > 2

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

( 1 2 4 5 8 ) –> ( 1 2 4 5 8 )

TIME COMPLEXITY –

Best Case – n

Average Case – n^2

Worst Case – n^2

ADVANTAGE – Simplicity and ease of implementation

DISADVANTAGE- inefficient code

* 1. SHELL SORT

Shellsort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. The concept behind Shellsort is that both of these algorithms perform in O(kn) time, where k is the greatest distance between two out-of-place elements. This means that generally, they perform in O(n2), but for data that is mostly sorted, with only a few elements out of place, they perform faster. So, by first sorting elements far away, and progressively shrinking the gap between the elements to sort, the final sort computes much faster. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

The worst-case time complexity of Shell sort largely depends on the gap sequence used, and can range from O(n2) to O(n log2 n). Also, unlike efficient sorting algorithms, Shellsort does not require use of the call stack, making it useful in embedded systems where memory is at a premium.

TIME COMPLEXITY –

Best Case- n

Average Case – n^2

Worst Case – n^2

ADVANTAGE – Efficient for medium size lists

DISADVANTAGE- Complex algorithm , not that efficient

* 1. **MERGE SORT**

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(n) space complexity, and involves a large number of copies in simple implementations.

Merge sort has seen a relatively recent surge in popularity for practical implementations, due to its use in the sophisticated algorithm Timsort, which is used for the standard sort routine in the programming languages Python and Java (as of JDK7). Merge sort itself is the standard routine in Perl, among others, and has been used in Java at least since 2000 in JDK1.3.

ALGORITHM –

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide the array into two halves:

middle m = (l+r)/2

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

TIME COMPLEXITY –

Best Case- n

Average Case – n^2

Worst Case – n^2

ADVANTAGE – well suited for large data sets

DISADVANTAGE- atleast twice the memory requirements than other sorts

* 1. **HEAP SORT**

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O(n log n) time, and this is also the worst case complexity.

ALGORITHM –

Input data: 4, 10, 3, 5, 1

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

The numbers in bracket represent the indices in the array

representation of data.

Applying heapify procedure to index 1:

4(0)

/ \

10(1) 3(2)

/ \

5(3) 1(4)

Applying heapify procedure to index 0:

10(0)

/ \

5(1) 3(2)

/ \

4(3) 1(4)

The heapify procedure calls itself recursively to build heap

in top down manner.

TIME COMPLEXITY –

Best Case- nlogn

Average Case – nlogn

Worst Case – nlogn

ADVANTAGE – Well suited for large data sets

DISADVANTAGE- Less efficient than quick sort and merge sort

* 1. **SELECTION SORT**

Selection sort is an in-place comparison sort. It has O(n2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

ALGORITHM –

arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]

// and place it at beginning

11 25 12 22 64

// Find the minimum element in arr[1...4]

// and place it at beginning of arr[1...4]

11 12 25 22 64

// Find the minimum element in arr[2...4]

// and place it at beginning of arr[2...4]

11 12 22 25 64

// Find the minimum element in arr[3...4]

// and place it at beginning of arr[3...4]

11 12 22 25 64

TIME COMPLEXITY –

Best Case- n^2

Average Case – n^2

Worst Case – n^2

ADVANTAGE – Easy to implement

DISADVANTAGE- inefficient for large lists

* 1. **COUNTING SORT**

Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in O(|S| + n) time and O(|S|) memory where n is the length of the input. It works by creating an integer array of size |S| and using the ith bin to count the occurrences of the ith member of S in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm often cannot be used because S needs to be reasonably small for the algorithm to be efficient, but it is extremely fast and demonstrates great asymptotic behavior as n increases. It also can be modified to provide stable behavior.

ALGORITHM –

For simplicity, consider the data in the range 0 to 9.

Input data: 1, 4, 1, 2, 7, 5, 2

1) Take a count array to store the count of each unique object.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 2 0 1 1 0 1 0 0

2) Modify the count array such that each element at each index

stores the sum of previous counts.

Index: 0 1 2 3 4 5 6 7 8 9

Count: 0 2 4 4 5 6 6 7 7 7

The modified count array indicates the position of each object in

the output sequence.

3) Output each object from the input sequence followed by

decreasing its count by 1.

Process the input data: 1, 4, 1, 2, 7, 5, 2. Position of 1 is 2.

Put data 1 at index 2 in output. Decrease count by 1 to place

next data 1 at an index 1 smaller than this index.

TIME COMPLEXITY –

Best Case- n+r

Average Case – n+r

Worst Case – n+r

ADVANTAGE – It is the best algorithm for sorting nfor 1 to 1000 numbers

DISADVANTAGE- The length of the counting array showul be equal to the number of sorted numbers

* 1. **BUCKET SORT**

Bucket sort is a divide and conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

A bucket sort works best when the elements of the data set are evenly distributed across all buckets.

ALGORITHM –

bucketSort(arr[], n)

1) Create n empty buckets (Or lists).

2) Do following for every array element arr[i].

.......a) Insert arr[i] into bucket[n\*array[i]]

3) Sort individual buckets using insertion sort.

4) Concatenate all sorted buckets.

TIME COMPLEXITY –

Best Case- n+r

Average Case – n+r

Worst Case – n+r

ADVANTAGE – Runs in linear time in the average case

DISADVANTAGE- Complexity depends on other sorting algorithms

**2.10 RADIX SORT**

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in O(n · k) time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort. A hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly.

ALGORITHM –

The Radix Sort Algorithm-

1) Do following for each digit i where i varies from least significant digit to the most significant digit.

………….a) Sort input array using counting sort (or any stable sort) according to the i’th digit.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [\*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [\*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

TIME COMPLEXITY –

Best Case- n.k/d

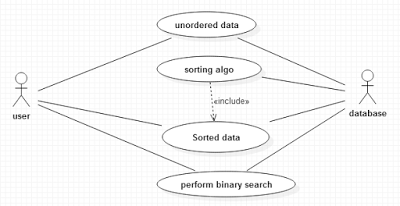
Average Case – n.k/d

Worst Case – n+2^d

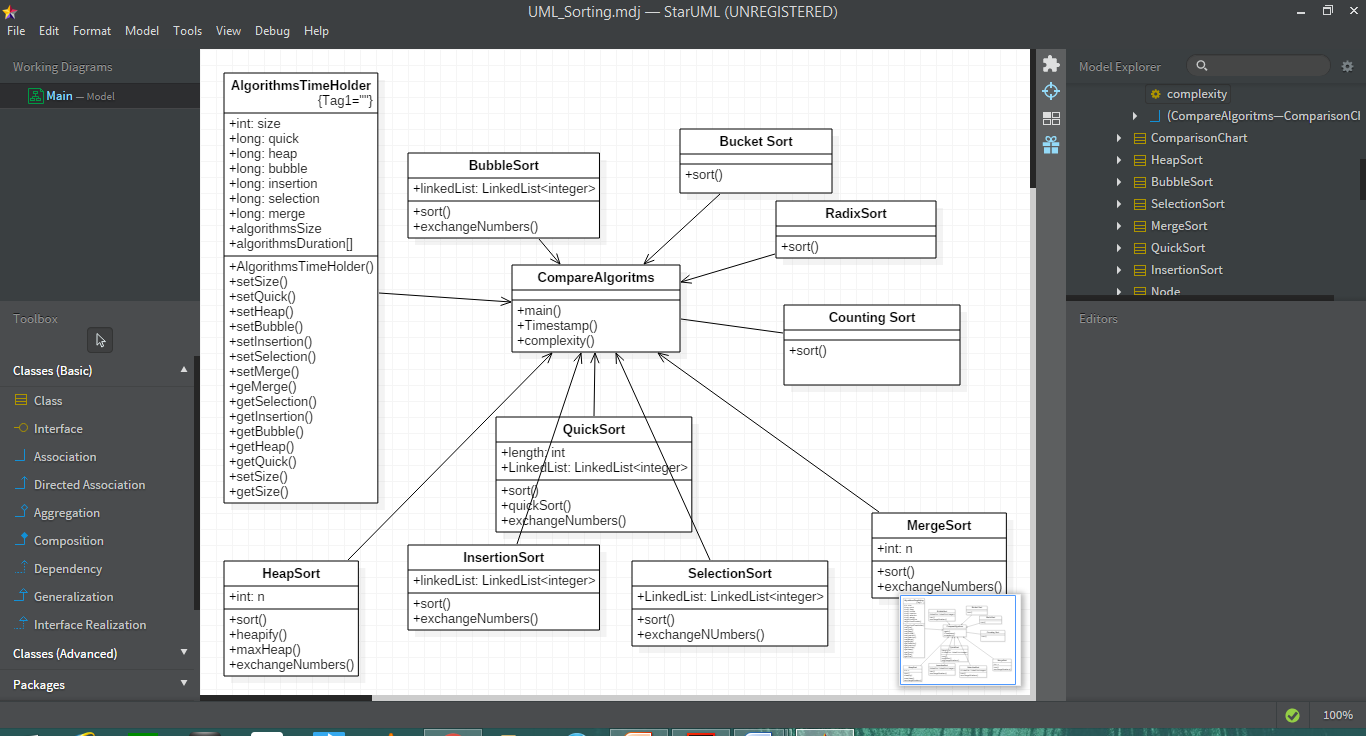
ADVANTAGE – Fast and complexity doesn’t depend on amount of data

DISADVANTAGE- Less flexible , takes more time

**3 USE CASE DIAGRAM**



1. **CLASS DIAGRAM**

****

**5 IMPLEMENTAION**

**5.1 QUICK SORT**

/\* C implementation QuickSort \*/

#include<stdio.h>

// A utility function to swap two elements

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

/\* This function takes last element as pivot, places

   the pivot element at its correct position in sorted

    array, and places all smaller (smaller than pivot)

   to left of pivot and all greater elements to right

   of pivot \*/

int partition (int arr[], int low, int high)

{

    int pivot = arr[high];    // pivot

    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)

    {

        // If current element is smaller than or

        // equal to pivot

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

/\* The main function that implements QuickSort

 arr[] --> Array to be sorted,

  low  --> Starting index,

  high  --> Ending index \*/

void quickSort(int arr[], int low, int high)

{

    if (low < high)

    {

        /\* pi is partitioning index, arr[p] is now

           at right place \*/

        int pi = partition(arr, low, high);

        // Separately sort elements before

        // partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", arr[i]);

    printf("n");

}

// Driver program to test above functions

int main()

{

    int arr[] = {10, 7, 8, 9, 1, 5};

    int n = sizeof(arr)/sizeof(arr[0]);

    quickSort(arr, 0, n-1);

    printf("Sorted array: n");

    printArray(arr, n);

    return 0;

}

**5.2 INSERTION SORT**

// C program for insertion sort

#include <stdio.h>

#include <math.h>

/\* Function to sort an array using insertion sort\*/

void insertionSort(int arr[], int n)

{

   int i, key, j;

   for (i = 1; i < n; i++)

   {

       key = arr[i];

       j = i-1;

       /\* Move elements of arr[0..i-1], that are

          greater than key, to one position ahead

          of their current position \*/

       while (j >= 0 && arr[j] > key)

       {

           arr[j+1] = arr[j];

           j = j-1;

       }

       arr[j+1] = key;

   }

}

// A utility function ot print an array of size n

void printArray(int arr[], int n)

{

   int i;

   for (i=0; i < n; i++)

       printf("%d ", arr[i]);

   printf("\n");

}

/\* Driver program to test insertion sort \*/

int main()

{

    int arr[] = {12, 11, 13, 5, 6};

    int n = sizeof(arr)/sizeof(arr[0]);

    insertionSort(arr, n);

    printArray(arr, n);

    return 0;

}

**5.3 SELECTION SORT**

// C program for implementation of selection sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

void selectionSort(int arr[], int n)

{

    int i, j, min\_idx;

    // One by one move boundary of unsorted subarray

    for (i = 0; i < n-1; i++)

    {

        // Find the minimum element in unsorted array

        min\_idx = i;

        for (j = i+1; j < n; j++)

          if (arr[j] < arr[min\_idx])

            min\_idx = j;

        // Swap the found minimum element with the first element

        swap(&arr[min\_idx], &arr[i]);

    }

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", arr[i]);

    printf("\n");

}

// Driver program to test above functions

int main()

{

    int arr[] = {64, 25, 12, 22, 11};

    int n = sizeof(arr)/sizeof(arr[0]);

    selectionSort(arr, n);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

**5.4 HEAP SORT**

// C implementation of Heap Sort

#include <stdio.h>

#include <stdlib.h>

// A heap has current size and array of elements

struct MaxHeap

{

int size;

int\* array;

};

// A utility function to swap to integers

void swap(int\* a, int\* b) { int t = \*a; \*a = \*b; \*b = t; }

// The main function to heapify a Max Heap. The function

// assumes that everything under given root (element at

// index idx) is already heapified

void maxHeapify(struct MaxHeap\* maxHeap, int idx)

{

int largest = idx; // Initialize largest as root

int left = (idx << 1) + 1; // left = 2\*idx + 1

int right = (idx + 1) << 1; // right = 2\*idx + 2

// See if left child of root exists and is greater than

// root

if (left < maxHeap->size &&

maxHeap->array[left] > maxHeap->array[largest])

largest = left;

// See if right child of root exists and is greater than

// the largest so far

if (right < maxHeap->size &&

maxHeap->array[right] > maxHeap->array[largest])

largest = right;

// Change root, if needed

if (largest != idx)

{

swap(&maxHeap->array[largest], &maxHeap->array[idx]);

maxHeapify(maxHeap, largest);

}

}

// A utility function to create a max heap of given capacity

struct MaxHeap\* createAndBuildHeap(int \*array, int size)

{

int i;

struct MaxHeap\* maxHeap =

(struct MaxHeap\*) malloc(sizeof(struct MaxHeap));

maxHeap->size = size; // initialize size of heap

maxHeap->array = array; // Assign address of first element of array

// Start from bottommost and rightmost internal mode and heapify all

// internal modes in bottom up way

for (i = (maxHeap->size - 2) / 2; i >= 0; --i)

maxHeapify(maxHeap, i);

return maxHeap;

}

// The main function to sort an array of given size

void heapSort(int\* array, int size)

{

// Build a heap from the input data.

struct MaxHeap\* maxHeap = createAndBuildHeap(array, size);

// Repeat following steps while heap size is greater than 1.

// The last element in max heap will be the minimum element

while (maxHeap->size > 1)

{

// The largest item in Heap is stored at the root. Replace

// it with the last item of the heap followed by reducing the

// size of heap by 1.

swap(&maxHeap->array[0], &maxHeap->array[maxHeap->size - 1]);

--maxHeap->size; // Reduce heap size

// Finally, heapify the root of tree.

maxHeapify(maxHeap, 0);

}

}

// A utility function to print a given array of given size

void printArray(int\* arr, int size)

{

int i;

for (i = 0; i < size; ++i)

printf("%d ", arr[i]);

}

/\* Driver program to test above functions \*/

int main()

{

int arr[] = {12, 11, 13, 5, 6, 7};

int size = sizeof(arr)/sizeof(arr[0]);

heapSort(arr, size);

printf("\nSorted array is \n");

printArray(arr, size);

return 0;

}

**5.5 BUBBLE SORT**

/ C program for implementation of Bubble sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

    int temp = \*xp;

    \*xp = \*yp;

    \*yp = temp;

}

// A function to implement bubble sort

void bubbleSort(int arr[], int n)

{

   int i, j;

   for (i = 0; i < n-1; i++)

       // Last i elements are already in place

       for (j = 0; j < n-i-1; j++)

           if (arr[j] > arr[j+1])

              swap(&arr[j], &arr[j+1]);

}

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

    int i;

    for (i=0; i < size; i++)

        printf("%d ", arr[i]);

    printf("n");

}

// Driver program to test above functions

int main()

{

    int arr[] = {64, 34, 25, 12, 22, 11, 90};

    int n = sizeof(arr)/sizeof(arr[0]);

    bubbleSort(arr, n);

    printf("Sorted array: \n");

    printArray(arr, n);

    return 0;

}

**5.6 BUCKET SORT**

/\*

\* C Program to Sort Array using Bucket Sort

\*/

#include <stdio.h>

/\* Function for bucket sort \*/

void Bucket\_Sort(int array[], int n)

{

int i, j;

int count[n];

for (i = 0; i < n; i++)

count[i] = 0;

for (i = 0; i < n; i++)

(count[array[i]])++;

for (i = 0, j = 0; i < n; i++)

for(; count[i] > 0; (count[i])--)

array[j++] = i;

}

/\* End of Bucket\_Sort() \*/

/\* The main() begins \*/

int main()

{

int array[100], i, num;

printf("Enter the size of array : ");

scanf("%d", &num);

printf("Enter the %d elements to be sorted:\n",num);

for (i = 0; i < num; i++)

scanf("%d", &array[i]);

printf("\nThe array of elements before sorting : \n");

for (i = 0; i < num; i++)

printf("%d ", array[i]);

printf("\nThe array of elements after sorting : \n");

Bucket\_Sort(array, num);

for (i = 0; i < num; i++)

printf("%d ", array[i]);

printf("\n");

return 0;

}

**5.7 RADIX SORT**

#include<stdio.h>

int getMax(int arr[], int n) {

int mx = arr[0];

int i;

for (i = 1; i < n; i++)

if (arr[i] > mx)

mx = arr[i];

return mx;

}

void countSort(int arr[], int n, int exp) {

int output[n]; // output array

int i, count[10] = { 0 };

// Store count of occurrences in count[]

for (i = 0; i < n; i++)

count[(arr[i] / exp) % 10]++;

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

// Build the output array

for (i = n - 1; i >= 0; i--) {

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

for (i = 0; i < n; i++)

arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using Radix Sort

void radixsort(int arr[], int n) {

int m = getMax(arr, n);

int exp;

for (exp = 1; m / exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

void print(int arr[], int n) {

int i;

for (i = 0; i < n; i++)

printf("%d ", arr[i]);

}

int main() {

int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };

int n = sizeof(arr) / sizeof(arr[0]);

radixsort(arr, n);

print(arr, n);

return 0;

}

**5.8 COUNTING SORT**

// C Program for counting sort

#include <stdio.h>

#include <string.h>

#define RANGE 255

// The main function that sort the given string arr[] in

// alphabatical order

void countSort(char arr[])

{

    // The output character array that will have sorted arr

    char output[strlen(arr)];

    // Create a count array to store count of inidividul

    // characters and initialize count array as 0

    int count[RANGE + 1], i;

    memset(count, 0, sizeof(count));

    // Store count of each character

    for(i = 0; arr[i]; ++i)

        ++count[arr[i]];

    // Change count[i] so that count[i] now contains actual

    // position of this character in output array

    for (i = 1; i <= RANGE; ++i)

        count[i] += count[i-1];

    // Build the output character array

    for (i = 0; arr[i]; ++i)

    {

        output[count[arr[i]]-1] = arr[i];

        --count[arr[i]];

    }

    // Copy the output array to arr, so that arr now

    // contains sorted characters

    for (i = 0; arr[i]; ++i)

        arr[i] = output[i];

}

// Driver program to test above function

int main()

{

    char arr[] = "geeksforgeeks";//"applepp";

    countSort(arr);

    printf("Sorted character array is %sn", arr);

    return 0;

}

**5.9 SHELL SORT**

// C++ implementation of Shell Sort

#include  <iostream>

using namespace std;

/\* function to sort arr using shellSort \*/

int shellSort(int arr[], int n)

{

    // Start with a big gap, then reduce the gap

    for (int gap = n/2; gap > 0; gap /= 2)

    {

        // Do a gapped insertion sort for this gap size.

        // The first gap elements a[0..gap-1] are already in gapped order

        // keep adding one more element until the entire array is

        // gap sorted

        for (int i = gap; i < n; i += 1)

        {

            // add a[i] to the elements that have been gap sorted

            // save a[i] in temp and make a hole at position i

            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct

            // location for a[i] is found

            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

                arr[j] = arr[j - gap];

            //  put temp (the original a[i]) in its correct location

            arr[j] = temp;

        }

    }

    return 0;

}

void printArray(int arr[], int n)

{

    for (int i=0; i<n; i++)

        cout << arr[i] << " ";

}

int main()

{

    int arr[] = {12, 34, 54, 2, 3}, i;

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "Array before sorting: \n";

    printArray(arr, n);

    shellSort(arr, n);

    cout << "\nArray after sorting: \n";

    printArray(arr, n);

    return 0;

}

**5.10 MERGE SORT**

/\* C program for Merge Sort \*/

#include<stdlib.h>

#include<stdio.h>

// Merges two subarrays of arr[].

// First subarray is arr[l..m]

// Second subarray is arr[m+1..r]

void merge(int arr[], int l, int m, int r)

{

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

/\* create temp arrays \*/

int L[n1], R[n2];

/\* Copy data to temp arrays L[] and R[] \*/

for (i = 0; i < n1; i++)

L[i] = arr[l + i];

for (j = 0; j < n2; j++)

R[j] = arr[m + 1+ j];

/\* Merge the temp arrays back into arr[l..r]\*/

i = 0; // Initial index of first subarray

j = 0; // Initial index of second subarray

k = l; // Initial index of merged subarray

while (i < n1 && j < n2)

{

if (L[i] <= R[j])

{

arr[k] = L[i];

i++;

}

else

{

arr[k] = R[j];

j++;

}

k++;

}

/\* Copy the remaining elements of L[], if there

are any \*/

while (i < n1)

{

arr[k] = L[i];

i++;

k++;

}

/\* Copy the remaining elements of R[], if there

are any \*/

while (j < n2)

{

arr[k] = R[j];

j++;

k++;

}

}

/\* l is for left index and r is right index of the

sub-array of arr to be sorted \*/

void mergeSort(int arr[], int l, int r)

{

if (l < r)

{

// Same as (l+r)/2, but avoids overflow for

// large l and h

int m = l+(r-l)/2;

// Sort first and second halves

mergeSort(arr, l, m);

mergeSort(arr, m+1, r);

merge(arr, l, m, r);

}

}

/\* UTILITY FUNCTIONS \*/

/\* Function to print an array \*/

void printArray(int A[], int size)

{

int i;

for (i=0; i < size; i++)

printf("%d ", A[i]);

printf("\n");

}

/\* Driver program to test above functions \*/

int main()

{

int arr[] = {12, 11, 13, 5, 6, 7};

int arr\_size = sizeof(arr)/sizeof(arr[0]);

printf("Given array is \n");

printArray(arr, arr\_size);

mergeSort(arr, 0, arr\_size - 1);

printf("\nSorted array is \n");

printArray(arr, arr\_size);

return 0;

}