DEPARTMENT OF NETWORKING AND COMMUNICATIONS

PROJECT REPORT

Academic year: 2021-22

EVEN SEMESTER

Programme(UG/PG)	**: UG**
Semester	**: IV**
Course Code	**: 18CSC204J**
Course Title	**: DESIGN AND ANALYSIS OF ALGORITHMS**
Student Name	**: SYED SAYEED**
	SHASHWAT VERMA
	OBUL REDDY
Register Number	**: RA2111028010135**
	RA2111028010136
	RA2111028010137
	Branch With
Specialization	**: CSE-CLOUD COMPUTING**
Section	**: Q1**

SRM UNIVERSITY (Under section 3 of UGC Act 1956)

SCHOOL OF COMPUTING
FACULTY OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
SRM NAGAR,KATTANKULATHUR-603202

Project Report

DESIGN AND ANALYSIS OF ALGORITHMS

A COURSE PROJECT REPORT

By

Shashwat Verma [RA2111028010136]
Syed Sayeed [RA2111028010135]
Obulreddy
Dutthala[RA2111028010137]

Under the guidance of
DR. S Rajasoundaran

(Associate Professor)

In partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE ENGINEERING

Of

FACULTY OF ENGINEERING AND TECHNOLOGY

SRM UNIVERSITY
(Under section 3 of UGC Act 1956)

S.R.M. Nagar, Kattankulathur, Chengalpattu District

April, 2023

SRM UNIVERSITY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report titled “SOLVING PROBLEM

STATEMENT USING 3 DIFFERENT ALGORITHMS” is the bonafide work of “SHASHWAT VERMA [RA2111028010136] , SYED SAYEED [RA2111028010135], OBULREDDY DUTTHALA [RA2111028010137} who carried out the project work under our supervision. Certified further, that to the best of our knowledge the work reported herein does not form any other project report or dissertation based on which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

DR. S Rajasoundaran
Assistant Professor

TABLE OF CONTENTS

ABSTRACT

The study of building effective algorithms to address complicated issues and evaluating their effectiveness is known as design and analysis of algorithms. An algorithm is a collection of guidelines that spells out how to carry out a certain activity or resolve a particular issue. Making a set of instructions that can be followed to solve a problem as quickly and precisely as feasible is the aim of constructing an algorithm.

Understanding the issue, discovering various solutions, choosing the best one, and then fine-tuning it to increase performance are all phases in the process of building an algorithm. The method is assessed in terms of its time and space complexity, as well as its accuracy and applicability for the issue at hand, during the analysis step.

In computer science and related subjects, efficient algorithms are crucial because they are used to address a variety of issues such as data analysis, optimisation, machine learning, and cryptography. Research is continuing in the design and study of algorithms, with the goal of creating new algorithms and improving those that already exist in order to solve problems more effectively.

The Merge Sort sorting algorithm splits the input array into two equal halves, sorts each half independently, and then merges the sorted halves back together. It is an effective, all-purpose comparison-based sorting algorithm with an O(n log n) time complexity.

The input array is split into two halves initially, and then each half is sorted recursively using the algorithm. By comparing the initial element of each half and adding the smaller one into a new array, it then combines the two sorted halves back together. This procedure is repeated until the new array, which is now the sorted array of the original array, has all the items.

The relative order of identical elements in the input array is maintained using the stable sorting technique known as merge sort. It is also a divide-and-conquer algorithm, which divides a larger issue into smaller subproblems, resolves each subproblem independently, and then combines the results to get the desired outcome.

The relative order of identical elements in the input array is maintained using the stable sorting technique known as merge sort. It is also a divide-and-conquer

algorithm, which divides a larger issue into smaller subproblems, resolves each subproblem independently, and then combines the results to get the desired outcome.

Quick Sort is a well-liked sorting algorithm that sorts an array of elements using a divide-and-conquer strategy. The pivot element of the method is chosen from the array, the array is divided around the pivot element, and then the two resultant subarrays are sorted recursively. The first, last, or middle member of the array can be chosen as the pivot element, among other options. With an average time complexity of O (n log n) and a worst-case time complexity of O(n2), Quick Sort is renowned for its effectiveness. Due to its efficiency and ease of use, it is frequently used as the default sorting algorithm in many computer languages and libraries. The relative order of identical elements in the input array may not be maintained by Quick Sort since it is not a stable sorting method.

ACKNOWLEDGEMENT

We express our heartfelt thanks to our honourable **Vice Chancellor Dr. C. MUTHAMIZHCHELVAN**, for being the beacon in all our endeavours. We would like to express my warmth of gratitude to our **Registrar Dr. S. Ponnusamy,** for his encouragement. We express our profound gratitude to our **Dean (College of Engineering and Technology) Dr. T. V. Gopal,** for bringing out novelty in all executions. We would like to express my heartfelt thanks to Chairperson, School of Computing **Dr. Revathi Venkataraman,** for imparting confidence to complete my course project. We are highly thankful to my Course project Faculty **DR. S Rajasoundaran , Associate Professor,** for his/her assistance, timely suggestion, and guidance throughout the duration of this course project.

Finally, we thank our parents and friends near and dear ones who directly and indirectly contributed to the successful completion of our project. Above all, I thank the almighty for showering his blessings on me to complete my Course project.

INTRODUCTION

The study of building effective algorithms to address complicated issues and evaluating their effectiveness is known as design and analysis of algorithms. An algorithm is a collection of guidelines that spells out how to carry out a certain activity or resolve a particular issue. Making a set of instructions that can be followed to solve a problem as quickly and precisely as feasible is the aim of constructing an algorithm.

Understanding the issue, discovering various solutions, choosing the best one, and then fine-tuning it to increase performance are all phases in the process of building an algorithm. The algorithm is assessed in terms of its accuracy and applicability for the issue, as well as its time and space complexity, during the analysis step.

In computer science and related subjects, efficient algorithms are crucial because they are used to address a variety of issues such as data analysis, optimisation, machine learning, and cryptography. Research is continuing in the design and study of algorithms, with the goal of creating new algorithms and improving those that already exist in order to solve problems more effectively.

The Merge Sort sorting algorithm splits the input array into two equal halves, sorts each half independently, and then merges the sorted halves back together. It is an effective, all-purpose comparison-based sorting algorithm with an O(n log n) time complexity.

The input array is split into two halves initially, and then each half is sorted recursively using the algorithm. By comparing the initial element of each half and adding the smaller one into a new array, it then combines the two sorted halves back together. This procedure is repeated until the new array, which is now the sorted array of the original array, has all of the items.

The relative order of identical elements in the input array is maintained using the stable sorting technique known as merge sort. It is also a divide-and-conquer algorithm, which divides a larger issue into smaller subproblems, resolves each subproblem independently, and then combines the results to get the desired outcome.

A straightforward sorting algorithm called Bubble Sort continually runs over a list of entries, compares nearby components, and swaps out those that are out of order. This process is repeated until no more swaps are necessary, at which point the array is said to have sorted. Although Bubble Sort is renowned for being straightforward and simple to use, it is typically ineffective for big input arrays and has a worst-case time complexity of O(n2). For short lists and instructional reasons, it is still helpful. The relative order of identical elements in the input array is maintained using the stable sorting technique known as bubble sort.

Quick Sort is a well-liked sorting algorithm that sorts an array of elements using a divide-and-conquer strategy. The pivot element of the method is chosen from the array, the array is divided around the pivot element, and then the two resultant subarrays are sorted recursively. The first, last, or middle member of the array can be chosen as the pivot element, among other options. With an average time complexity of O (n log n) and a worst-case time complexity of O(n2), Quick Sort is renowned for its effectiveness. the input array's. Due to its efficiency and ease of use, it is frequently used as the default sorting algorithm in many computer languages and libraries. The relative order of identical elements in the input array may not be maintained by Quick Sort since it is not a stable sorting method.

PROBLEM STATEMENT

Let us say you are asked to arrange an array of N numbers in a non-decreasing order. The array, however, has a few odd characteristics:

• Both positive and negative numbers can appear in any order in the array.

• There are a lot of duplicate elements in the array.

• Because the array is so big, you need to sort it as quickly as you can.

Create an algorithm that sorts this array in a non-decreasing order using three distinct sorting techniques—Merge Sort, Bubble Sort, and Quick Sort—and evaluate the time complexity of each technique. Choose the sorting algorithm that performs the best for this task by comparing its effectiveness to that of the other two.

Input:

4, 12, 76, 10, 25, 84, 52

Output:

4, 10, 12, 25, 52, 76, 84

LITERATURE REVIEW

- Design and Analysis of Algorithm:

  An important area of computer science is the design and analysis of algorithms, which entails creating effective algorithms to address challenging issues and evaluating their effectiveness.

  Understanding the issue, discovering various solutions, choosing the best one, and then fine-tuning it to increase performance are all phases in the process of building an algorithm. The method is assessed in terms of its time and space complexity, as well as its accuracy and applicability for the issue at hand, during the analysis step.

  In computer science and related subjects, efficient algorithms are crucial because they are used to address a variety of issues such as data analysis, optimisation, machine learning, and cryptography. Making a set of instructions that can be followed to solve a problem as quickly and precisely as feasible is the aim of constructing an algorithm.

  Algorithm analysis entails assessing an algorithm's performance in terms of its time and space complexity. An algorithm's space complexity is how much memory it needs to execute, and its time complexity is how long it takes to finish. The Big-O notation, which reflects the upper bound on the algorithm's execution time or memory use as the input size approaches infinity, is commonly used to assess an algorithm's efficiency.

  Research is continuing in the design and study of algorithms, with the goal of creating new algorithms and improving those that already exist in order to solve problems more effectively. Complex issues that develop in a variety of fields, including finance, medical, engineering, and many others, require the use of this subject to effectively solve them. It is an incredibly interesting field of computer science that calls for mathematical rigour, inventiveness, and a thorough knowledge of algorithms and data structures.

- Sorting:

  Sorting is the process of placing a group of components in a specific order. Data processing, database administration, and information retrieval all make use of sorting, which is a fundamental activity in computer science.

  differing sorting algorithms have differing levels of effectiveness, complexity, and applicability to diverse kinds of data. Bubble sort, Insertion sort, Selection sort, Merge sort, Quick sort, and Heap sort are a few of the most used sorting algorithms.

  Although Bubble Sort, Insertion Sort, and Selection Sort are straightforward, understandable, and simple to use, huge data sets typically do not benefit from their use. The more complicated algorithms Merge Sort, Quick Sort, and Heap Sort perform better on bigger data sets. Using a divide-and-conquer strategy, these algorithms split the input into smaller subsets, sort them individually, and then combine them to get the final sorted output.

  The three main criteria used to assess sorting algorithms are time complexity, space complexity, and stability. A sorting algorithm's time complexity, which is often defined in terms of the number of comparisons or swaps necessary, is how long it takes to accomplish the sorting operation. The quantity of memory needed to carry out the sorting process is the space complexity. If a sorting method preserves the relative order of equal elements in the input array, it is stable.

  Numerous applications depend on sorting, which is frequently employed as a foundational step in other algorithms. The qualities of the input data, the intended output, and the performance requirements of the application all play a role in the selection of a sorting algorithm.

- Merge Sort:

  Merge Sort is a popular sorting algorithm that uses a divide-and-conquer approach to sort an array of elements. The algorithm works by dividing the input array into two halves, recursively sorting each half, and then merging the two sorted halves to form the final sorted array.

  The merge operation involves comparing and merging the elements of the two sorted halves to create a new sorted array. This process is repeated until the entire input array is sorted. Merge Sort is a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the input array.

Merge Sort is known for its efficiency and has a time complexity of O (n log n) in the worst-case, which makes it one of the most efficient sorting algorithms available. It also has a space complexity of O(n), which is relatively low compared to other sorting algorithms.

One of the key advantages of Merge Sort is that it can be easily parallelized, making it suitable for use in multi-core and distributed systems. Merge Sort is also widely used as a building block for other algorithms and data structures, such as merge-join operations in database management systems.

While Merge Sort is generally considered to be a more complex algorithm than other sorting algorithms such as Bubble Sort or Insertion Sort, it is still relatively easy to implement and understand. It is a popular sorting algorithm due to its efficiency, stability, and versatility.

- Bubble Sort:

  Bubble Sort is a simple and intuitive sorting algorithm that repeatedly steps through the input array, compares adjacent elements, and swaps them if they are in the wrong order. The algorithm gets its name from the way that smaller elements "bubble up" to the top of the array during each iteration.

  Bubble Sort has a time complexity of O(n^2) in the worst case, making it one of the slowest sorting algorithms for large data sets. However, it has a space complexity of O (1), which means that it requires very little memory to run.

  Bubble Sort is generally not used for large data sets or in applications where efficiency is critical. However, it is often used in educational contexts to teach basic sorting algorithms due to its simplicity and ease of implementation.

  Despite its limitations, Bubble Sort does have some advantages. It is a stable sorting algorithm, meaning that it maintains the relative order of equal elements in the input array. It is also easy to understand and modify, making it a useful tool for experimentation and testing.

  Overall, Bubble Sort is a useful algorithm for teaching basic sorting concepts, but it is not typically used in production applications where efficiency and scalability are critical.

- Quick Sort:

  Quick Sort is a popular sorting algorithm that uses a divide-and-conquer approach to sort an array of elements. The algorithm works by selecting a pivot element from the input array and partitioning the array into two sub-arrays, one containing element smaller than the pivot and the other containing elements larger than the pivot. The algorithm then recursively sorts the two sub-arrays until the entire input array is sorted.

  The partitioning step in Quick Sort is crucial to its efficiency. The pivot element can be selected in different ways, such as choosing the first, last, or middle element of the array. After selecting the pivot, the algorithm rearranges the elements of the array so that all elements smaller than the pivot are to the left of it, and all elements larger than the pivot are to the right of it.

  Quick Sort has a time complexity of O (n log n) in the average case and O(n^2) in the worst case when the pivot is poorly chosen. However, the worst-case scenario is rare in practice, and Quick Sort is generally considered to be one of the fastest sorting algorithms for large data sets. It also has a space complexity of O (log n), which is relatively low compared to other sorting algorithms.

  One of the key advantages of Quick Sort is its efficiency and versatility. It is used in a wide range of applications, from sorting large datasets in database management systems to implementing search algorithms in computer science. Quick Sort is also commonly used as a building block for other algorithms, such as quick select and median-of-medians algorithms.

  Overall, Quick Sort is a powerful and efficient sorting algorithm that is widely used in computer science and related fields. Its efficiency and versatility make it a popular choice for sorting large data sets, and its simplicity and elegance make it a valuable tool for teaching and experimentation.

ALGORITHM

- Merge Sort:

```
Step – 1: Start
Step – 2: Declare an array and left, right, mid variable
Step – 3: Perform merge function
          mergesort (array, left. Right)
          if left>right
          return
          mid=(left+right)/2
          mergesort (array, left, mid)
          mergesort (array, mid+1, right)
          merge (array, left, mid, right)
Step – 4: Stop
```

- Bubble Sort:

```
Step – 1:
Start
Step – 2: Repeat step 3 for 1 to N-1
Step – 3: if A[j]>a[j+1]
                Swap A[j] and A[J+1]
          [END OF LOOP]
Step – 4: Stop
```

- Quick Sort:

```
Step – 1: Start
Step – 2: quicksort(arr[], low, high){
              If(low<high){
                  pi=partition (arr, low, high);
                  quicksort (arr, low, pi-1);
                  quicksort (arr, pi+1, high);
              }
          }
Step – 3: partition (arr[], low, high)
          {
              Pivot = arr[high];
              i = (low-1)
              for (j=low;j<=high-1;j++){
                  if (arr[j] < pivot){
                      i++;
                      swap arr[i] and arr[j]
                  }
              }
              Swap arr[i+1] and arr[high])
              Return(i+1)
          }
```

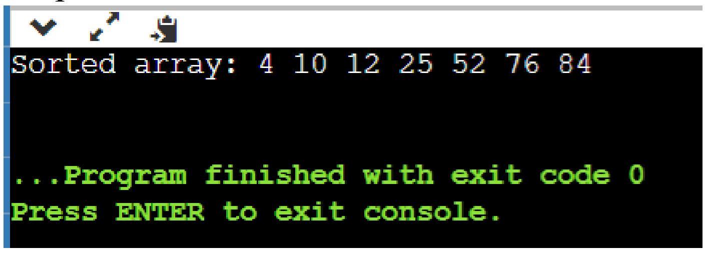
SOLVING PROBLEM USING MERGE SORT

Program:

```
#include <iostream>

using namespace std;

void merge(int arr[], int left, int mid, int right) {
   int i, j, k;
   int n1 = mid - left + 1;
   int n2 = right - mid;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[left + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[mid + 1 + j];

   i = 0;
   j = 0;
   k = left;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      }
      else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }

   while (i < n1) {
      arr[k] = L[i];
```

```
    i++;
        k++;
    }

    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        merge_sort(arr, left, mid);
        merge_sort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

int main() {
int arr[] = { 4, 12, 76, 10, 25, 84, 52};
    int n = sizeof(arr) / sizeof(arr[0]);

    merge_sort(arr, 0, n - 1);

    cout << "Sorted array: ";
     for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;


    return 0;
}
```

Output:



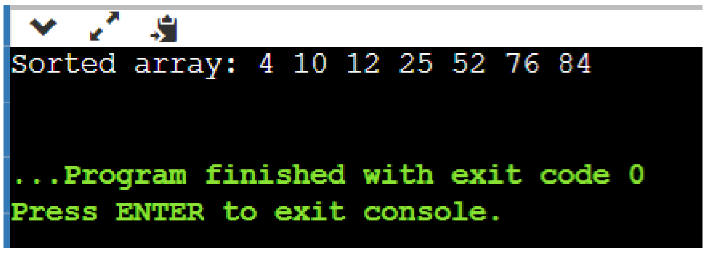
```
Sorted array: 4 10 12 25 52 76 84


...Program finished with exit code 0
Press ENTER to exit console.
```

SOLVING PROBLEM USING BUBBLE SORT

Program:

```
#include <iostream>

using namespace std;

void bubble_sort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {
      for (int j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            swap(arr[j], arr[j + 1]);
         }
      }
   }
}

int main() {
   int arr[] = { 4, 12, 76, 10, 25, 84, 52};
   int n = sizeof(arr) / sizeof(arr[0]);

   bubble_sort(arr, n);

   cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
      cout << arr[i] << " ";
   cout << endl;

   return 0;
}
```

Output:



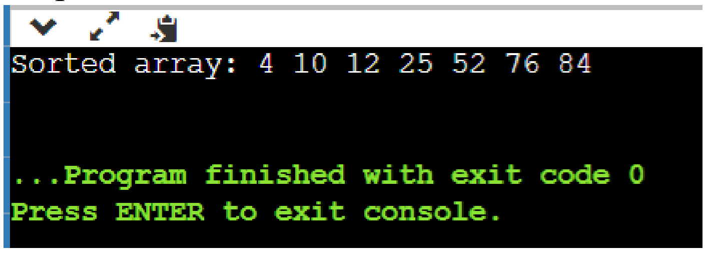
```
Sorted array: 4 10 12 25 52 76 84


...Program finished with exit code 0
Press ENTER to exit console.
```

SOLVING PROBLEM USING QUICK SORT

Program:

```
#include <iostream>

using namespace std;

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] <= pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    swap(arr[i + 1], arr[high]);
    return i + 1;
}

void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

int main() {
    int arr[] = { 4, 12, 76, 10, 25, 84, 52};
    int n = sizeof(arr) / sizeof(arr[0]);

    quick_sort(arr, 0, n - 1);

    cout << "Sorted array: ";
     for (int i = 0; i < n; i++)
       cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Output:



```
Sorted array: 4 10 12 25 52 76 84


...Program finished with exit code 0
Press ENTER to exit console.
```

ANALYSIS OF MERGE SORT

Let T (n) be the total time taken by the Merge Sort algorithm.

- Sorting two halves will take at the most $2T\frac{n}{2}$ time.
- When we merge the sorted lists, we come up with a total n-1 comparison because the last element which is left will need to be copied down in the combined list, and there will be no comparison.

Thus, the relational formula will be

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1$$

But we ignore '-1' because the element will take some time to be copied in merge lists.

So $T(n) = 2T\left(\frac{n}{2}\right) + n$...equation 1

Note stopping condition T (1) =0 because at last, there will be only 1 element that need to be copied, and there will be no comparison.

Putting $n=\frac{n}{2}$ in place of n in ....equation 1

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2^2}\right) + \frac{n}{2}$..................equation2

Put 2 equation in 1 equation

$$T(n) = 2\left[2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right] + n$$

$$= 2^2T\left(\frac{n}{2^2}\right) + \frac{2n}{2} + n$$

$T(n) = 2^2 T\left(\frac{n}{2^2}\right)+2n$......................equation 3

Putting $n=\frac{n}{2^2}$ in equation 1

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}$......................equation4

Putting 4 equation in 3 equation

T (n) = $2^2 \left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right] + 2n$

T (n) =$2^3\ T\left(\frac{n}{2^3}\right) + n + 2n$

T (n) = $2^3\ T\left(\frac{n}{2^3}\right) + 3n$............................equation5

From eq 1, eq3, eq 5.......we get

T (n) = $2^i\ T\left(\frac{n}{2^i}\right) + in$..............................equation6

From Stopping Condition:

$\frac{n}{2^i} = 1$ And $T\left(\frac{n}{2^i}\right)$=0

n = 2^i

Apply log both sides:

log n=$\log_2 i$

logn= i log2

$\frac{\log n}{\log 2}$ =i

$\log_2 n$=i

From 6 equation

T (n) = $2^i\ T\left(\frac{n}{2^i}\right) + in$

= 2^i x 0+ $\log_2 n \,.\, n$

= T (n) = n.log n

ANALYSIS OF BUBBLE SORT

In this unoptimised version the run time complexity is `Θ(N^2)`. This applies to all the cases including the worst, best and average cases because even if the array is already sorted the algorithm doesn't check that at any point and runs through all iterations. Although the number of swaps would differ in each case.

The number of times the inner loop runs

$$= \sum_{j=1}^{N-i-1} 1 \tag{2}$$

$$= (N-i) \tag{1}$$

The number of times the outer loop runs

$$= \sum_{i=1}^{N-1} \sum_{j=1}^{N-i-1} 1 = \sum_{i=1}^{N-1} N-i \tag{3}$$

$$= \frac{N*(N-1)}{2} \tag{4}$$

The number of swaps of two elements is equal to the number of comparisons in this case as every element is out of place.

$T(N) = C(N) = S(N) = \frac{N*(N-1)}{2}$, from equation 2 and 4

Therefore, in the worst case:

- Number of Comparisons: O(N^2) time
- Number of swaps: O(N^2) time

ANALYSIS OF QUICK SORT

- **B) Best Case :**
- In the best case the array can be divided into exactly two equal parts.
- So the time complexity is:

$$\begin{aligned}T(n) &= 2\,T(n/2) + n \\ &= 2\,[2T(n/4) + n/2] + n \\ &= 4\,T(n/4) + 2\,n \\ &= 4\,[2T(n/8) + n/4) + 2\,n \\ &= 8\,T(n/8) + 3\,n \\ &= 2^3\,T(n/2^3) + 3\,n \\ &\;\;. \\ &\;\;. \\ &\;\;. \\ &= 2^i\,T(n/2^i) + i\,n \\ &= 2^i\,T(1) + i\,n\end{aligned}$$

$n = 2^i$ so $i = \log_2 n$

$T(n) = a\,n + n\log_2 n$

So the Best Case time complexity is **$T(n) = \Omega(n \log_2 n)$**

Relational Formula for Worst Case:

T (n) =T (1) + T (n-1) + n...................... (1)

T (n-1) =T (1) +T (n-1-1) + (n-1)

By putting (n-1) in place of n in equation1

Put T (n-1) in equation1

T(n)=T(1)+T(1)+(T(n-2)+(n-1)+n.......................(ii)

T (n) =2T (1) +T (n-2) + (n-1) +n

T (n-2) =T (1) +T (n-3) + (n-2)

By putting (n-2) in place of n in equation1

Put T (n-2) in equation (ii)

T(n)=2T(1)+T(1)+T(n-3)+(n-2)+(n-1)+n

T(n)=3T(1)+T(n-3)+)+(n-2)+(n-1)+n

T (n-3) =T (1) +T (n-4) +n-3

By putting (n-3) in place of n in equation1

T(n)=3T(1)+T(1)+T(n-4)+(n-3)+(n-2)+(n-1)+n

=4T(1)+T(n-4))+(n-3)+(n-2)+(n-1)+n...........................(iii)

T(n)=(n-1) T(1) + T(n-(n-1))+(n-(n-2))+(n-(n-3))+(n-(n-4))+n

T (n) = (n-1) T (1) + T (1) + 2 + 3 + 4+............n

T (n) = (n-1) T (1) +T (1) +2+3+4+...........+n+1-1

[Adding 1 and subtracting 1 for making AP series]

T (n) = (n-1) T (1) +T (1) +1+2+3+4+........ + n-1

T (n) = (n-1) T (1) +T (1) + $\frac{n(n+1)}{2}$ -1

Stopping Condition: T (1) =0

Because at last there is only one element left and no comparison is required.

T (n) = (n-1) (0) +0+ $\frac{n(n+1)}{2}$ -1

T (n) =$\frac{n^2 +n-2}{2}$

Avoid all the terms expect higher terms n^2

T (n) =O (n^2)

Worst Case Complexity of Quick Sort is T (n) =O (n^2)

TIME COMPLEXITY

- Merge Sort:

  Best Case = O (n log n)
  Worst Case = O (n log n)
  Average Case = O (n log n)
- Bubble Sort:
  Best Case =
  O(n)
  Worst Case = O(n^2)
  Average Case = O(n^2)

- Quick Sort:

  Best Case = O (n log n)
  Worst Case = O(n^2)
  Average Case = O (n log n)

COMPARSION

Merge Sort and Quick Sort are generally considered to be more efficient than Bubble Sort, especially for larger data sets. Both Merge Sort and Quick Sort have a best-case, average-case, and worst-case time complexity of O (n log n), which is much faster than Bubble Sort's best-case time complexity of O(n) and worst-case time complexity of O(n^2).

Between Merge Sort and Quick Sort, both have the same time complexity in the best-case, average-case, and worst-case scenarios. However, Quick Sort is typically faster in practice due to its cache-friendly and in-place partitioning mechanism.

Therefore, Quick Sort is often the preferred choice for general-purpose sorting applications, especially when memory usage is a concern. Merge Sort is more suitable for situations where stability and predictability are essential, or when sorting large and complex data structures such as linked lists.

Bubble Sort, on the other hand, is a simple and easy-to-implement sorting algorithm, but its efficiency is poor compared to Merge Sort and Quick Sort, especially for larger data sets.

In summary, while each sorting algorithm has its own strengths and weaknesses, Merge Sort and Quick Sort are generally considered to be more efficient than Bubble Sort, and Quick Sort is often preferred due to its faster performance in practice.

CONCLUSION

In conclusion, the design and analysis of algorithms is a crucial aspect of computer science and plays a vital role in developing efficient and reliable software systems. Through this project, we have studied and implemented three popular sorting algorithms - Merge Sort, Bubble Sort, and Quick Sort - and analysed their time complexities and performance characteristics.

Merge Sort is a stable and efficient sorting algorithm with a time complexity of O (n log n) in the best-case, average-case, and worst-case scenarios. It is particularly useful for sorting linked lists and other complex data structures.

Bubble Sort, while simple to understand and implement, has a worst-case time complexity of O(n^2), making it inefficient for larger data sets.

Quick Sort is a widely used sorting algorithm with a time complexity of O (n log n) in the best-case and average-case scenarios, but it can degenerate to O(n^2) in the worst-case scenario. However, various optimization techniques can be used to mitigate this risk.

Overall, the choice of sorting algorithm depends on various factors such as the size and complexity of the data set, memory usage, and performance requirements. By understanding the strengths and weaknesses of each algorithm, we can make informed decisions and design efficient and effective software systems.

Merge Sort and Quick Sort are generally considered to be more efficient than Bubble Sort, especially for larger data sets. Both Merge Sort and Quick Sort have a best-case, average-case, and worst-case time complexity of O(n log n), which is much faster than Bubble Sort's best-case time complexity of O(n) and worst-case time complexity of O(n^2).

Between Merge Sort and Quick Sort, both have the same time complexity in the best-case, average-case, and worst-case scenarios. However, Quick Sort is typically faster in practice due to its cache-friendly and in-place partitioning mechanism.

Therefore, Quick Sort is often the preferred choice for general-purpose sorting applications, especially when memory usage is a concern. Merge Sort is more suitable for situations where stability and predictability are essential, or when sorting large and complex data structures such as linked lists.

Bubble Sort, on the other hand, is a simple and easy-to-implement sorting algorithm, but its efficiency is poor compared to Merge Sort and Quick Sort, especially for larger data sets.

In summary, while each sorting algorithm has its own strengths and weaknesses, Merge Sort and Quick Sort are generally considered to be more efficient than Bubble Sort, and Quick Sort is often preferred due to its faster performance in practice.

THANK YOU