Complexity

Complexity Classes

- In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to groups problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.
- The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage. The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.
  The space complexity of an algorithm describes how much memory is required for the algorithm to operate.
- Complexity classes are useful in organizing similar types of problems.

Complexity Classes:

1.P Class

2.NP Class

3.CoNP Class

4.NP hard

5.NP complete

P Class

- The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.
- **Features:**

1.The solution to P problems is easy to find.

2.P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

- This class contains many natural problems like:
  1. **Calculating the greatest common divisor.**
  2. **Finding a maximum matching.**
  3. **Decision versions of linear programming.**

NP Class

- The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.
- **Features:**

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time.

NP Class

- Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons.
  This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a co-worker is satisfactory or not i.e. no pair taken from the co-worker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.
- It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.
- This class contains many problems that one would like to be able to solve effectively:

1. **Boolean Satisfiability Problem (SAT).**
2. **Hamiltonian Path Problem.**
3. **Graph coloring.**

Co-NP Class

- Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.
- **Features:**

1.If a problem X is in NP, then its complement X’ is also is in CoNP.

2.For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer “yes” or “no” in polynomial time for a problem to be in NP or CoNP.

- Some example problems for C0-NP are:
  1. **To check prime number.**
  2. **Integer Factorization.**

NP-hard class

- An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.
- **Features:**

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

- Some of the examples of problems in Np-hard are:
    1. **Halting problem.**
    2. **Qualified Boolean formulas.**
    3. **No Hamiltonian cycle.**

NP-complete class

- A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.
- **Features:**

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

- Some example problems include:
  1. **Decision version of 0/1 Knapsack.**
  2. **Hamiltonian Cycle.**
  3. **Satisfiability.**
  4. **Vertex cover.**

Characteristic features

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Algorithm Analysi

- **What is meant by Algorithm Analysis?**
- Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.
- **Why Analysis of Algorithms is important?**
- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

- **Bubble Sort** is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.
- **How does Bubble Sort Work?**
- ***Input:*** *arr[] = {5, 1, 4, 2, 8}*
- ***First Pass:***
- *Bubble sort starts with very first two elements, comparing them to check which one is greater.*
  - *(**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:***
- *Now, during second iteration it should look like this:*
  - *(**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**)*
- ***Third Pass:***
- *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.*
  - *(**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**)*
- ***Illustration:***

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i =1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

Follow the below steps to solve the problem:

- Run a nested for loop to traverse the input array using two variables **i** and **j**, such that $0 \leq i < n-1$ and $0 \leq j < n-i-1$
- If **arr[j]** is greater than **arr[j+1]** then swap these adjacent elements, else move on
- Print the sorted array

Time Complexity

- **Time Complexity:** $O(N^2)$
  Auxiliary Space: O(1)
- **Optimized Implementation of Bubble Sort:**
- The above function always runs **$O(N^2)$** time even if the array is sorted. It can be optimized by stopping the algorithm if the inner loop didn’t cause any swap.

Worst Case Analysis for Bubble Sort:

The **worst-case** condition for bubble sort occurs when elements of the array are arranged in decreasing order.
In the worst case, the total number of iterations or passes required to sort a given array is **(n-1).** where ‘n’ is a nu

At pass 1 : Number of comparisons = (n-1)
Number of swaps = (n-1)
At pass 2 : Number of comparisons = (n-2)
Number of swaps = (n-2)
At pass 3 : Number of comparisons = (n-3)
Number of swaps = (n-3)
.
.
.
At pass n-1 : Number of comparisons = 1
Number of swaps = 1

Now , calculating total number of comparison required to sort the array
= (n-1) + (n-2) + (n-3) + . . . 2 + 1
= (n-1)*(n-1+1)/2 *{ by using sum of N natural Number formula }*
= **n (n-1)/2**

- ***Total number of swaps = Total number of comparison***
  Total number of comparison (Worst case) = ***n(n-1)/2***
  Total number of swaps (Worst case) = ***n(n-1)/2***
- ***Worst and Average Case Time Complexity:*** *$O(N^2)$. The worst case occurs when an array is reverse sorted.*
  Best Case Time Complexity: *O(N). The best case occurs when an array is already sorted.*
  Auxiliary Space: *O(1)*

- **What is the Boundary Case for Bubble sort?**
- Bubble sort takes minimum time (Order of n) when elements are already sorted. Hence it is best to check if the array is already sorted or not beforehand, to avoid $O(N^2)$ time complexity.