COMPLEXITY CLASSES

JERRY SUN

Abstract. Complexity Classes are a way to describe how hard a problem is. It is important to Cryptography because many well known cryptosystems work because they are based off of hard problems. El Gamal and RSA are both based off of problems that don't have effiecient solutions.

1. Definitions

Complexity Class: The set of problems that can be solved by an abstract machine M using big $\mathcal{O}(f(n))$ of resource R, where n is the size of the input.
The time complexity of an algorithm is usually used when describing the number of steps it needs to take to solve a problem, but it can also be used to describe how long it takes verify an answer (more on this in the NP section). There are many ways of finding time complexity. One could figure out time complexity by determining how many times a particular line of code executes in a program, or by figuring out how many steps a Turing machine takes when solving the problem. Knowing the time complexity of an algorithm helps programmers and computer scientists know if an algorithm is efficient or not. It helps us to know how long an algorithm will take to give an answer.

1.1. **Formal Definition.** $f(n) = \mathcal{O}(g(n))$ means there are positive constants c and k, such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n. Also known as O, asymptotic upper bound
The space complexity of an algorithm describes how much memory the algorithm needs in order to operate. In terms of Turing machines, the space needed to solve a problem relates to the number of spaces on the Turing machine's tape it needs to do the problem.
Probabilistic Turing Machine: In computability theory, a probabilistic Turing machine is a non-deterministic Turing machine which chooses between the available transitions at each point according to some probability distribution.

2. Types of Complexity Classes

2.1. **BQP.** In computational complexity theory, bounded-error quantum polynomial time (BQP) is the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of at most 1/3 for all instances.[1] It is the quantum analogue of the complexity class BPP.

A decision problem is a member of BQP if there exists a quantum algorithm (an algorithm that runs on a quantum computer) that solves the decision problem with high probability and is guaranteed to run in polynomial time. A run of the algorithm will correctly solve the decision problem with a probability of at least 2/3.

Date: December 8, 2019.

2.2. **BPP.** Informally, a problem is in BPP if there is an algorithm for it that has the following properties:

It is allowed to flip coins and make random decisions It is guaranteed to run in polynomial time On any given run of the algorithm, it has a probability of at most 1/3 of giving the wrong answer, whether the answer is YES or NO

2.3. **ExpTime.** ExpTime is the set of all problems that can be solved in 2 to a polynomial of the input

2.4. **Polynomial Time.** P is the set of all problems that can be solved in a polynomial of the input.

2.5. **Pspace.** P is the set of all problems that take a polynomial amount of time based on the input.

2.6. **NP.** the set of all decision problems whose solutions can be verified in polynomial time; NP may be equivalently defined as the set of decision problems that can be solved in polynomial time on a non-deterministic Turing machine.

2.7. **NP-Complete.** NP-complete problems a subset of NP. These are problems that are equivalent to each other, meaning if you can solve one of them you can solve all of them.

2.8. **NP-hard.** a problem H is NP-hard when every problem L in NP can be reduced in polynomial time to H; that is, assuming a solution for H takes 1 unit time, H's solution can be used to solve L in polynomial time. As a consequence, finding a polynomial algorithm to solve any NP-hard problem would give polynomial algorithms for all the problems in NP, which is unlikely as many of them are considered difficult.

3. Boolean Satisfiability problem (SAT)

In computer science, the Boolean satisfiability problem is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. In other words, it asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE. If this is the case, the formula is called satisfiable. On the other hand, if no such assignment exists, the function expressed by the formula is FALSE for all possible variable assignments and the formula is unsatisfiable.

4. Hamiltonian path Problem

he decision problem form of the knapsack problem (Can a value of at least V be achieved without exceeding the weight W?) is NP-complete, thus there is no known algorithm both correct and fast (polynomial-time) in all cases.

5. subset sum problem

In computer science, the subset sum problem is an important decision problem in complexity theory and cryptography. There are several equivalent formulations of the problem. One of them is: given a set (or multiset) of integers, is there a non-empty subset whose sum is zero?
For example, given the set {-7,-3,-2,5,8}, the answer is yes because the subset {-3,-2,5} sums to zero. The problem is NP-complete, meaning roughly that while it is easy to confirm whether a proposed solution is valid, it may inherently be prohibitively difficult to determine in the first place whether any solution exists.

6. Knapsack problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The decision problem form of the knapsack problem (Can a value of at least V be achieved without exceeding the weight W?) is NP-complete, thus there is no known polynomial-time algorithm that works in general.

7. Subgraph isomorphism problem

The subgraph isomorphism problem is a computational task in which two graphs G and H are given as input, and one must determine whether G contains a subgraph that is isomorphic to H. Subgraph isomorphism is a generalization of both the maximum clique problem and the problem of testing whether a graph contains a Hamiltonian cycle, and is therefore NP-complete.d However certain other cases of subgraph isomorphism may be solved in polynomial time.

8. Travelling salesman problem (decision version)

The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.

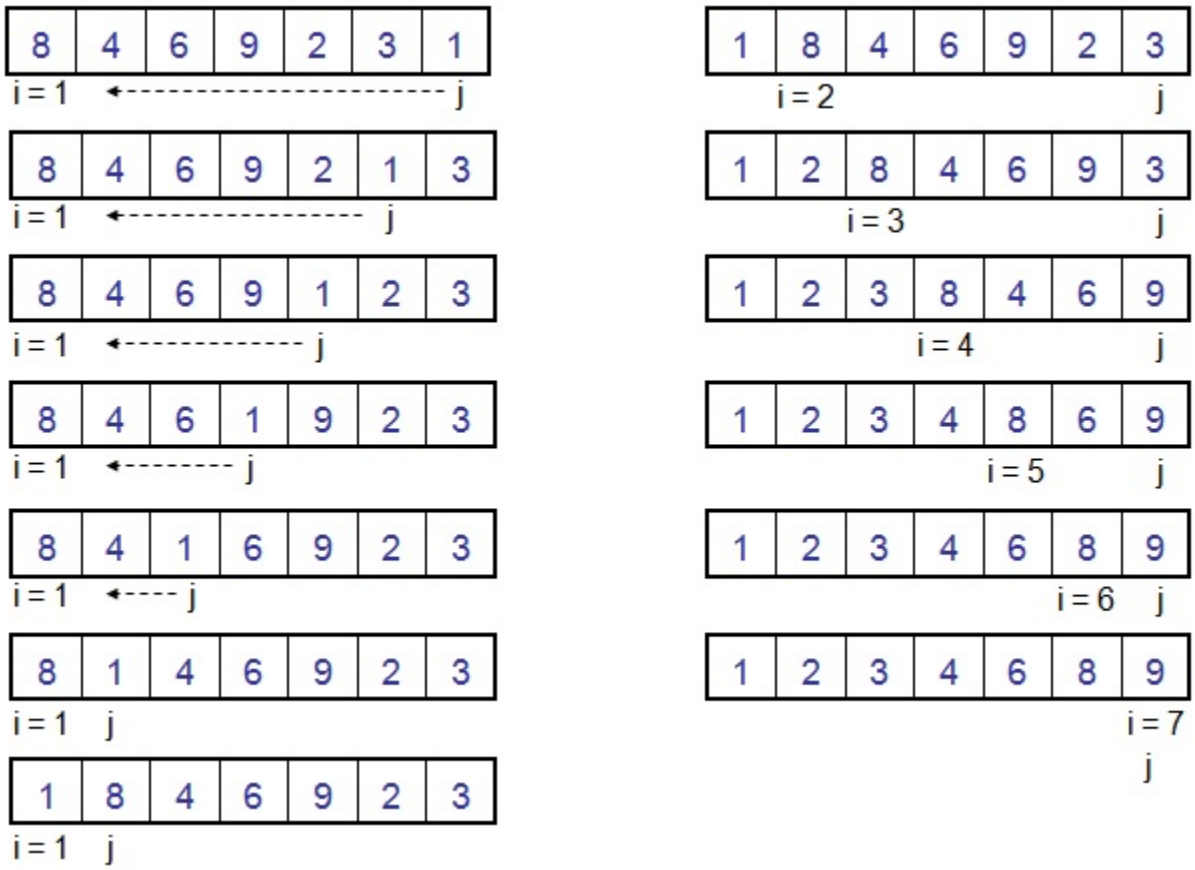
9. P=NP Conjecture

The P versus NP problem is a major unsolved problem in computer science. It asks whether every problem whose solution can be quickly verified (this means that it should be solved in polynomial time) can also be solved quickly (again, in polynomial time).

10. Calculating O(n)

Now we will practice calculating the big o of some famous sorting algorithms and graph theory problems.

10.1. **Mergesort.** Conceptually, a merge sort works as follows: Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted). Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list. In sorting n objects, merge sort has an average and worst-case performance of $\mathcal{O}(n \log n)$. If the running time of merge sort for a list of length n is T(n), then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem for divide-and-conquer recurrences.



10.2. **Bubble Sort.** Bubble sort, is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. It first makes n comparison, which brings the largest number to the the correct position, then it makes another n passes which bring the second largest number to the pass. Overall it makes n passes of length n so it has time complexity $o(n^2)$. The above picture shows an example of bubble sort.

11. Conclusion

Complexity classes are extremely useful in cryptography because, good cryptosystems are based on hard problems and so we need to find how long it takes to solve these hard problems.

References

[1] Wikipedia contributors. (2019, September 22). NP-completeness. In Wikipedia, The Free Encyclopedia. Retrieved 05:47, December 8, 2019, from https://en.wikipedia.org/w/index.php?title=NP-completenessoldid=917231396

[2] Wikipedia contributors. (2019, May 23). Complexity class. In Wikipedia, The Free Encyclopedia. Retrieved 05:52, December 8, 2019, from https://en.wikipedia.org/w/index.php?title=Complexityclassoldid=898430776

[3] Weisstein, Eric W. "Graph Isomorphism Complete." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GraphIsomorphismComplete.html

[4] Wikipedia contributors. (2019, February 10). List of complexity classes. In Wikipedia, The Free Encyclopedia. Retrieved 05:51, December 8, 2019, from https://en.wikipedia.org/w/index.php?title=Listofcomplexity$_c lassesoldid$ = 882711619