**CHAPTER 0-5**

**complexity**

**Complexity of algorithms**

* How do we measure the complexity of an algorithm?
* Intellectual complexity.
  + Intellectual complexity has to do with the understandability of a program.
  + It deals with the amount of decision making and branching involved in an algorithm.
  + In this light, a complex algorithm would be one that involves a twisted, entwined set of directions which makes the algorithm difficult to follow and understand.
  + Intellectual complexity might be compatible with the point of view of a software engineer who is interested in issues relating to algorithm discovery and representation, but it does not capture the concept of complexity from a machine’s point of view.
* Time complexity.
  + Time complexity reflects the complexity of an algorithm from a machine’s point of view because it measures the number of steps a machine must perform when executing an algorithm.
  + Note that this is not the same as the number of instructions appearing in the written program.
  + Ultimately, time complexity is concerned with the time it takes a machine to execute a solution and not with the size of the program representing the solution.
  + We consider a problem to be complex if all its solutions require a lot of time.
  + The Big O notation is used to describe the time complexity of an algorithm.
* Space complexity.
  + Space complexity has to do with the storage space requirements of a program.
  + The space complexity of an algorithm is determined by the amount of storage space required to run the algorithm.
  + The Big O notation is also used to describe the space complexity of an algorithm.

**Complexity of problems**

* How do we measure the complexity of a problem?
  + In computer science, the problems of interest are those that are solvable by machines.
  + The solutions to these problems are formulated as algorithms.
  + The complexity of a problem is determined by the properties of the algorithms that solve that problem. In particular, the time complexity of a problem is defined to be the time complexity of its best solution.
* Unfortunately, finding the best solution to a problem and knowing that it is the best is often a difficult problem in itself.
  + Thus, to say that a problem belongs to O(f(n)) often means that it has a solution whose complexity is in O(f(n)) but it could possibly have a better solution.

**Polynomials**

* Given a non-negative integer n, a polynomial function of 1 variable, say x, of degree n can always be written as:

f(x)=anxn + an-1xn-1 + an-2xn-2 + … + a2x2 + a1x1 + a0 (the ai's are real coefficients, an≠0)

* A polynomial function in x of degree 0 is called a constant function.
* A polynomial function in x of degree 1 is called a linear function.
* A polynomial function in x of degree 2 is called a quadratic function.
* A polynomial function in x of degree 3 is called a cubic function.

**Polynomial versus Nonpolynomial Problems**

* Suppose f(n) and g(n) are mathematical expressions. To say that g(n) is bounded by f(n) means that as we apply these expressions to larger and larger values of n, the value of f(n) will ultimately become greater than that of g(n) and remain greater than g(n) for all larger values of n.
* We say that a problem is a polynomial problem if the problem is in O(f(n)), where the expression f(n) is either a polynomial itself or bounded by a polynomial.
* The collection of all polynomial problems is represented by P.
* Identifying the problems that belong to P is of major importance in computer science because it is closely related to questions regarding whether problems have practical solutions.
  + Indeed, problems that are outside the class P are characterized as having extremely long execution times, even for inputs of moderate size.
* Consider, for example, a problem whose solution requires 2n steps. The exponential expression 2n is not bounded by any polynomial.
* An algorithm whose complexity is identified by an exponential expression is said to require exponential time.
* As a particular example, consider the problem of listing all possible subcommittees that can be formed from a group of n people.
  + Because there are 2n - 1 such subcommittees, any algorithm that solves this problem must have at least 2n - 1 steps and thus a complexity at least that large.
  + So this subcommittees problem is a nonpolynomial problem. It cannot be in P because it cannot have a polynomial solution.
* The fact that the problems that are theoretically solvable but are not in P have such enormous time complexities leads us to conclude that these problems are essentially unsolvable from a practical point of view.
  + Computer scientists call these problems intractable.

**The mysterious NP problems**

* Let us now consider the traveling salesperson problem (TSP), which involves a traveling salesperson who must visit each of his clients in different cities without exceeding his travel budget.
  + His problem, then, is to find a path starting from his home, connecting the cities involved, and returning to his home whose total length does not exceed his allowed mileage.
* The traditional solution to this problem is to consider the potential paths in a systematic manner, comparing the length of each path to the mileage limit until either an acceptable path is found or all possibilities have been considered.
  + Since there is an edge connecting any 2 cities, the number of possible tours is (n-1)!. Any algorithm that solves the TSP problem must have at least (n-1)! steps in the worst case and thus a complexity at least that large.
  + No one has found an algorithm to solve the TSP with polynomial runtime efficiency.
  + On the other hand, no one has proven the TSP to be a nonpolynomial problem.
* Let us consider the following algorithm:

Pick one of the possible paths, and compute its total distance.

if (this distance is not greater than the allowable mileage):

Declare a success.

else:

Declare nothing.

Observe that if a satisfactory path exists and we happen to select it first, our present algorithm terminates quickly. However, this set of instructions is not an algorithm in the technical sense. Its first instruction is ambiguous in that it does not specify which path is to be selected. Instead, it relies on the creativity of the mechanism executing the program to make the decision on its own. We say that such instructions are nondeterministic, and we call an “algorithm” containing such statements a nondeterministic algorithm.

* Note that the time required to execute the nondeterministic algorithm is bounded by a polynomial.
  + Thus it is possible to solve the traveling salesperson problem by a nondeterministic algorithm in polynomial time.
* Of course, our nondeterministic solution is not totally satisfactory.
  + It relies on a lucky guess.
* Whether there is a deterministic solution to the traveling salesperson problem that runs in polynomial time remains an open question.
  + In fact, the traveling salesperson problem is one of many problems that are known to have nondeterministic polynomial solutions but for which no deterministic polynomial solution has yet been found.
* A problem that can be solved in polynomial time by a nondeterministic algorithm is called a nondeterministic polynomial problem, or an NP problem for short.

**Deterministic versus Nondeterministic**

* P is defined as the set of **all problems** for which a polynomial algorithm exists.
  + Or equivalently, P is defined as the set of **all problems** for which an algorithm exists which can be carried out by a deterministic Turing machine in polynomial time.
* NP is defined as the set of **all problems** for which a nondeterministic polynomial algorithm exists.
  + Or equivalently, NP is defined as the set of all problems for which an algorithm exists which can be carried out by a nondeterministic Turing machine in polynomial time.
* A nondeterministic TM is like a conventional TM, but at any given step during computation, it can create multiple instances of itself and therefore can calculate a vast number of possibilities simultaneously.
* Consider the nondeterministic algorithm for the TSP again:

Pick one of the possible paths, and compute its total distance.

if (this distance is not greater than the allowable mileage):

Declare a success.

else:

Declare nothing.

When a nondeterministic TM executes the first step of this algorithm, it simply creates (n-1)! TMs, with each TM handling a different path. We can easily see that each TM will either declare success or failure in polynomial time.

* There is no computer in the world which qualifies as a true nondeterministic TM. Nondeterministic TMs are a purely abstract concept and do not exist in reality.

**P=NP?**

* Whether P=NP is a major unsolved problem in computer science.
* Clearly, all of the P problems are also in NP.
  + Any nondeterministic TM can masquerade as a deterministic TM by simply not splitting at any step.
  + So, any problem solvable by a deterministic TM in polynomial time is also solvable by a nondeterministic TM in polynomial time. Thus, P ⊆ NP.
* Whether all of the NP problems are also in P, however, is an open question, as demonstrated by the traveling salesperson problem.
* Efforts to resolve the question of whether NP = P have led to the discovery of a class of problems within the class NP known as the NP-complete problems.
* NP-complete problems have the property that a polynomial time solution for any one of them would provide a polynomial time solution for all the other problems in NP as well.
  + The traveling salesperson problem is an example of an NP-complete problem.
* This issue whether P=NP remains open because no one has found a polynomial running time solution for any NP-complete problem nor has anybody been able to prove that such an algorithm cannot exist.

**Summary**

* Problems can be classified as either solvable (having an algorithmic solution) or unsolvable (not having an algorithmic solution).
* Moreover, within the class of solvable problems are 2 subclasses.
  + One is the collection of polynomial problems that contains those problems with practical solutions.
  + The second is the collection of nonpolynomial problems whose solutions are practical for only relatively small inputs.
* Note that the mysterious NP problems are in the class of solvable problems, but we do not know if they are polynomial problems or nonpolynomial problems

**HW**

1. Suppose a problem can be solved by an algorithm in O(n2) as well as another algorithm in O(2n). Will one algorithm always outperform the other?
2. Give an example of a polynomial problem. Give an example of a nonpolynomial problem. Give an example of an NP problem that as yet has not been shown to be a polynomial problem.
3. If the time complexity of algorithm X is greater than that of algorithm Y, is algorithm X necessarily harder to understand than algorithm Y? Explain your answer.