COT 5407:Introduction to Algorithms

Author and Copyright: Giri Narasimhan
Florida International University
Lecture 1: August 28, 2007.

1 Introduction

The field of *algorithms* is the bedrock on which all of computer science rests. Would you jump into a business project without understanding what is in store for you, without knowing what business strategies are needed, without understanding the nature of the market, and without evaluating the competition and the availability of skilled available workforce? In the same way, you should not undertake writing a program without thinking out a strategy (algorithm), without theoretically evaluating its performance (algorithm analysis), and without knowing what resources you will need and you have available.

While there are broad principles of algorithm design, one of the the best ways to learn how to be an expert at designing good algorithms is to do an extensive survey of "case studies". It provides you with a storehouse of strategies that have been useful for solving other problems. When posed with a new problem, the first step is to "model" your problem appropriately and cast it as a problem (or a variant) that has been previously studied or that can be easily solved. Often the problem is rather complex. In such cases, it is necessary to use general problem-solving techniques that one usually employs in modular programming. This involves breaking down the problem into smaller and easier subproblems. For each subproblem, it helps to start with a skeleton solution which is then refined and elaborated upon in a stepwise manner.

Once a strategy or algorithm has been designed, it is important to think about several issues: why is it correct? does is solve all instances of the problem? is it the best possible strategy given the resource limitations and constraints? if not, what are the limits or bounds on the amount of resources used? are improved solutions possible?

2 History of Algorithms

It is important for you to know the giants of the field, and the shoulders on which we all stand in order to see far. Besides laying down the foundations of geometry, the Greek mathematician, Euclid (ca. 300 BC) is best known for his **gcd** algorithm, He showed that the gcd of two natural numbers a and b ($a \geq b$) is the same as the gcd of the numbers (a mod b) (i.e., the remainder when you divide a by b) and b. This observation immediately leads to a simple algorithm for computing the gcd of natural numbers. An Indian mathematician, Bhaskara (ca. 6th century AD), is credited with inventing the decimal system of counting[1]. Al Khwarizmi (ca. 9th century), from Baghdad, wrote a book detailing how numbers can be added, subtracted, multiplied, and divided. He also explained how to compute square

[1]It is said that the most important contribution of the ancient Indian thinkers was a "zero".

roots and to compute the value of π. The nineteenth century English inventor, Charles Babbage, is considered the father of the computer. His achievement was the construction of mechanical, programmable computers, which he called the *difference engine* and the *analytical engine*. Alan Turing, was a twentieth century English mathematician, who proposed the *Turing model of computation*, a theoretical model of an algorithm that solves a problem. His formalization paved the way for the field of theory of computation and computational complexity theory. John von Neumann, a Hungarian-born genius, wrote influential papers in the areas of physics, mathematics, logic, game theory, and computer science. The *von Neumann architecture* including the *stored program concept* is largely a result of his work. In the field of algorithms, he is credited with inventing the Merge Sort and for initiating the study of pseudo-random number generators. In the last 50 years, the field of algorithms has seen many influential researchers including Edsger Dijkstra, Donald Knuth, Michael Rabin, Robert Floyd, Richard Karp, John Hopcroft, Robert Tarjan, Andrew Yao, and many more.

3 Fun "warm-up" problems

The following are a few problems that require very little prior knowledge of algorithms. The hope is that these problems will get you excited about the field and also get you thinking about the issues that are important for algorithm design.

3.1 Searching in bounded and unbounded sets

Consider the following problem. I have thought of an integer x that is known to lie in the range of integers a through b. Your task is to guess this number with the fewest number of guesses. (Translation: input is a and b, output is the correct value of x.) You are allowed to ask questions of the form:"Is the number x equal to P?" or "Is the number x greater than P?"; these will elicit a yes/no answer.

I recommend that for every algorithm, the first thing to do is to design a naive algorithm so that you know immediately what you need to "beat". Usually, the naive algorithm is simplistic and unsophisticated and involves finding the size of the search space.

Clearly, in this case, the size of the search space is $n = b - a + 1$. For each value of P in the range of a through b, asking a question of the form "Is the number x equal to P?" will find the correct solution. The naive search strategy employed above is the *sequential search* strategy. The number of questions will be n in the worst case. The standard question you have to train yourself to ask is: can we do better?

By now, if you have had a course on data structures or advanced programming, you would have guessed that *Binary Search* will do the trick. The idea behind binary search is simple. By choosing $P = \lceil (b-a)/2 \rceil$ and asking the question "Is the number x greater than P?", regardless of whether the answer is yes or no, the size of the search space is reduced by a half. Consequently, you will require $\lceil \log_2 n \rceil$ questions in the worst case. (Why?) Thus the number of questions is exponentially smaller (what does that mean?) than the number of questions required by the naive scheme we discussed earlier.

Once again, can we do better? It turns out that you cannot. However, the proof is more involved and will be delayed until our discussion of lower bounds.

If you are letting out a yawn already, it is time to wake you up with a harder question. Here is a modified problem. I have thought of a **positive** integer x. No upper bounds are provided for the value of x. Your task is to guess this number with the fewest number of guesses. The type of questions you are allowed to ask remains the same.

The problem has changed. But exactly how? What is the size of the search space? It seems that it is *infinite*, i.e., the size of the set of positive integers. Is there even a naive algorithm? Once again, a naive solution is to perform sequential search. For each value of P starting with $P = 1$ and incrementing it by 1 after every failed attempt, ask the question "Is the number x equal to P?". This will eventually stop and find the correct solution. Is this an "algorithm" for the problem at hand? (You need to check the definition of an algorithms.) The search strategy employed is sequential search. The number of questions will be x in the worst case. So, can we do better? What strategy should we employ?

If you replied *binary search* again, then you are wrong! (Why?) This has been coined the *gambler's strategy*. We let P take the following sequence of values: $1, 2, 4, \ldots, 2^m$, while asking the question "Is the number x greater than P?". Eventually $2^m \geq x$ and the answer will be yes. Such a search is called the *doubling search* strategy. We now know that the number x lies in the range $2^{m-1} + 1$ through 2^m. So, binary search can be employed to zone in on the value x.

How many questions were required. For the first phase of the search, we asked questions to find the upper bound on x. This requires $m = \lfloor \log x \rfloor + 1$ in the worst case. The second phase involves a straightforward binary search and requires $m-1$ questions. Thus the search requires a total of $2m-1 = 2\lfloor \log x \rfloor + 1$ questions in the worst case. The number of questions is only twice as much as the case when we know the bound.

Can we do better? Oh yes! Our algorithm used doubling search. What happens if we do a "tripling" search, or a search where the jumps are even larger? But then the second phase of the search becomes more expensive. Another way to view doubling search is to treat it as a sequential search for the smallest value m such that $2^m \geq x$. So why couldn't we do doubling search for m? How far can we take this argument? Try looking for a solution that runs in roughly $\log x + \log \log x$ number of questions.

3.2 Evaluating polynomials

Back in your College Algebra course, you must have learnt about *polynomials*. A polynomial of degree n can be written down as follows:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0,$$

where $a_n, a_{n-1}, \ldots, a_1, a_0$ are real-valued coefficients of the terms of the polynomial.

Consider the problem of evaluating a polynomial. Thus the input is a sequence of real coefficients along with an integer value representing the degree of the polynomial, and a real value X at which the polynomial needs to be evaluated. The output is the real value of the polynomial at $x = X$.

The naive solution involves computing each of the $n+1$ terms separately and then adding them up. Computing the i-th term of the evaluation involves computing $a_i x^i$, which involves making i multiplications. Thus the evaluation of the polynomial would end up requiring $n + (n-1) + \ldots + 1$ multiplications and n additions. In other words, this naive solution requires n additions and $n(n+1)/2$ multiplications. If you have taken a class in Computer Architecture or if you know additions and multiplications are performed by the hardware, you would know that multiplications are much more expensive than additions. Applications that require a large number of polynomial evaluations may see tremendous performance improvements if the number of multiplications performed during polynomial evaluation can be reduced.

Improving on the naive scheme is easy if you observe that computing x^n in the straightforward way involves computing $x^2, x^3, \ldots, x^{n-1}$ as intermediate results. One suggestion is to store these intermediate values in an array, thus preventing recomputation in later steps. Such an evaluation scheme will end up with $n + 1 + 1 + \ldots + 1 = 2n - 1$ multiplications. The first term is n because x^n needs to be computed and then multiplied to a_n. This revised scheme also uses n additions. However, it uses an extra array of size n to store the intermediate results.

A further improvement is possible if you know about *Horner's Rule*. Using this rule, the polynomial can be rewritten as follows:

$$p(x) = ((\cdots((a_n x + a_{n-1})x + a_{n-2})\cdots + a_1)x + a_0).$$

Using a standard evaluation procedure will start with the innermost parentheses, thus first evaluating $(a_n x + a_{n-1})$, which involves 1 multiplication and 1 addition. This intermediate results is then stored, multiplied by x and added to a_{n-2}, which requires an additional multiplication and addition operation. It is clear that the entire evaluation only requires n additions and n multiplications, and requires storage for only 1 intermediate value at any given time. Applications that require a large number of polynomial evaluations will certainly show enormous performance improvement using this improved scheme.

3.3 The paparazzi and the celebrity

Say that you are a photographer and a budding paparazzo[2]. You are heading to a party where, gossip has it that a "celebrity" may make an appearance. The problem is that you don't know what this celebrity looks like. The definition of a celebrity in the paparazzi circles is "one who knows nobody (at the party) *and* one that everyone (at the party) knows". The only way to find the celebrity (if one is present) is to go around asking a lot of questions. You wish to accomplish the task with the fewest number of questions. The only type of questions you are allowed to ask at the party is to approach person A, point to another person B, and ask if (s)he knows B. The only allowable answers are YES and NO, and nobody at the party

[2]Paparazzo was the last name of a photographer in the film *La Dolce Vita*, by Federico Fellini, the great Italian director. You are probably familiar with the plural form of this word, namely "paparazzi".

has a reason to lie to you. What is the strategy to find the celebrity in the fewest number of questions.

Assume that there are n people attending the party (other than yourself). We have been told that the celebrity may or may not make an appearance. So it is possible there is one celebrity at the party or there are none present. It is easy to see that there cannot be two celebrities at the party. (Why?) The obvious "naive" scheme is to approach every person at the party and ask this person a question about every other person at the party. This generates a total of $n(n-1)$ (i.e., $O(n^2)$) questions and is guaranteed to find the celebrity at the end of the process. Clearly, the aim of this scheme is to obtain complete information by asking questions before figuring out the identity of the celebrity.

You have probably heard of the algorithmic strategy called *Divide-and-Conquer*. If you were to use this strategy, then you would partition the party into two equal-sized groups (say the "men" and the "women"). You would solve it separately on the two partitions. This would result in two potential candidates, one from each partition. You would then be left with verifying if either of these two candidates is a celebrity. This would end up generating $O(n \log n)$ questions, as will be discussed later.

Once again, we ask whether we can do better. To see the improved solution, here's a hint: how many questions do you need to spot a "non-celebrity"? Even if you are able to answer that question, how does it help you? How many questions does your best strategy require? It should be $O(n)$. You should also try to figure out the constants in your expression for the number of questions asked. Can you prove that the algorithm is correct. In other words, if you end up figuring out that person X is a celebrity, does he satisfy the definition of a celebrity?

4 Summary

The "fun" problems discussed above were meant to be challenging, but not difficult. The pattern should be obvious. For many problems, it is usually easy to design a naive solution, a trivial, brute-force solution that is guaranteed to give you the correct answer. The naive solution helps to set the stage; it gives you the sense of the solution space that one needs to search. But it provides no clues to the structure of the solution. None of the special properties of the problem or the input are exploited. The improved solutions often require *lateral* thinking. It often requires you to stare at the problem for a while and to try to understand it deeply. It often stems from understanding the hurdles to designing more efficient algorithms. In the search problem, the improvements came from knowing that the upper bound is critical for applying binary search and that one can *search for the upper bound* in a manner similar to the search for the value x. In the polynomial evaluation problem, the improvements came from the realization that computing x^n involves many intermediate computations that can be reused. In the celebrity problem, the breakthrough came from the observation that it is easy to spot non-celebrities, and recognizing non-celebrities effectively reduces the size of the solution space and is hence useful.