Algorithms: Nice and Lovely

Samir Khuller *

1 Preface

First of all Algorithms is not Logarithms spelt incorrectly. This is to correct the impression that some of my relatives have about what is it that I do. Most people, upon hearing that I am a "Computer Science Professor" immediately conclude that I am immensely qualified to fix their PC. Nothing could be further from the truth! In fact, I had never even used a PC in my life until 1997, when I was asked to fix one for a neighbor. I was too embarassed to tell her that I will not be able to fix the PC. In the end I was able to fix it, but only by a stroke of luck. The printer seemed to reset its head when the PC was booted, but the PC seemed to think that no printer was attached when it was booted. I suggested that the cable was faulty, lacking any other useful suggestion. She changed the cable the next day and called me to tell me that the printer now worked. Now she thinks I am a whiz at fixing PC's. In fact I was so scared of the next call I was going to get from her, when there was a serious problem with her PC, that we moved out of the neighborhood.

Another incident that prompted the writing of this monograph was a discussion with a cousin who was taking a Master's course in computer applications. I explained to her about my work, by using a simple example of sorting since she seemed to be familiar with that problem. Given a set of numbers such as $S = \{63, 24, 1, 49, 3\}$ how do we put them in sorted order? The sorted order (in increasing value) is $1, 3, 24, 49, 63$. I started to explain that there were many algorithms that could be used to obtain this desired output from the given input. Different algorithms can take a different number of steps based on what method they use. Some algorithms use more steps, and some use few steps. After this short lecture on sorting algorithms she said "But all sorting algorithms take the same amount of time. You just hit return and the answer comes back in sorted order." Sigh.

Finally, I decided that I must write a monograph that I can hand to my relatives or friends, or their high school children, so that they can get an understanding of my field of research if they do so care. People are knowledgeable about elementary Chemistry, Physics, Mathematics, Biology, and other subjects that they learn about in middle and high school. Sadly, when people talk about teaching computer science at the high school level, its mostly about programming in one language, or programming in another language, or both. I learnt about the field of Algorithms somewhat late in life unfortunately. To me, it seems like the most fascinating field in the entire subject of Computer Science. It is simply to convey some of the depth, beauty and elegance of this field did I decide to write this monograph. I see no reason as to why students interested in Mathematics, or abstract thinking cannot learn this field at earlier stages of their schooling. In addition, some of these algorithms play a role in the design of widely used software. We all use

*Research supported by NSF Award CCR-0113192. Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail : samir@cs.umd.edu.

these software packages to do things for us, without realizing how these algorithms are used, or are even being aware of their existence.

I will introduce the reader to many problems, and discuss the solutions to several of these problems explaining how the algorithms work. The field of algorithms is a little like mathematics. Often the problems are easy to describe, the solutions are sometimes very ingenious, and sometimes even easy to verify by a reader who knows little about the problem! Often it was easy to understand new problems, as well as understand their solutions. However, coming up with the solutions can be quite challenging; and being an expert sometimes is of no help at all. The most creative solutions could come from smart graduate students, who have just started research in the area (often they are the ones with wild ideas, and have the energy to show that some of these wild ideas actually work). In the process of explaining the problems and their solutions, we will see that the reader can learn some useful tools which can then be used to attack other problems.

I do not want to assume a high level of mathematical maturity, so the "proofs" will be informal, usually by simple explanation or examples. The reader is urged to read the technical papers or books referenced at the end of each section to identify further readings.

2 Algorithms Background

Algorithms are simply methods or recipes for solving problems. An algorithm is a step by step procedure that takes an input and produces the desired output. Take for example the simple problem of "given an integer n, is it prime?". Recall that a prime number is one that is only divisible by 1 or itself, and no other number. For example, 2 and 3 are prime numbers, but 4 is not prime as 2 divides 4. Again, 5 and 7 are prime, but 6, 8 and 9 are not since 2 divides 6 and 8, and 3 divides 9. For small numbers, it is easy to obtain a listing of all prime numbers by checking for divisors that are smaller than the number itself. However, given a large number such as 1019, is it prime? One trivial algorithm would be to check if some number i divides it, but we will have to perform this test for each value of $i < 1019$. Is there no better way?

Suppose we have a street map of DC. We look up the location of the airport, and decide to drive there. How do we choose a route? People have different algorithms for doing this. Some rely on local knowledge of the area, and knowledge of traffic patterns etc. So clearly if we have been living in an area for a long time, we can do a good job doing route planning. Suppose we are just visiting a city and renting a car and we have no idea about the traffic patterns? Suppose we just want to simply take the shortest route? Or a route that will get us there in the least amount of time. Often we look at a map and do "visual navigation" to compute a route. One algorithm would be "go from your starting point via some short route to the nearest highway, and then use the highway to approximately where we wish to go, and then get off at the appropriately marked exit, and then look up the map for a short route to the destination". Have you ever thought about how a computer would solve this problem? There are services such as mapquest[**?**] that gives directions and routes between chosen locations. Ofcourse there is an issue of the "representation" of the map in the computer. One can scan a map in, and this fine for simply doing a visual display of the map, or for zooming and viewing parts of the map. How about actually doing route planning? For this a different representation is required.

How do we know that the algorithm is good or bad? Typically most algorithms are evaluated by the number of elementary steps that they take to solve a given problem (also referred to as running time). Often, the actual number of steps taken by the algorithm depend on several factors. The size of the input to the algorithm is itself one of the biggest factors determining

the running time of the algorithm. Moreover, the running time of the algorithm may vary when given two different inputs of the same size. We usually paramterize the size of the input to the algorithm by a parameter n, and measure the *maximum* running time of the algorithm for any input of size n. This is referred to as *worst* case analysis of the algorithm, and sometimes is too pessimistic.

3 History

Algorithms is a very old field. One of the earliest recorded algorithms is the one given by Euclid to compute the greatest common divisor of two numbers (this particular algorithm will be discussed shortly). This algorithm dates back to 300 BC (or earlier). In addition, algorithms were developed in the 9th Century (AD) by Al-Khowarizmi, a Persian Scholar, who developed algorithms for adding, multiplying and dividing decimal numbers. In fact, the word "Algorithm" was derived from his name.

In addition, algorithmic problems have been studied within the field of mathematics. Often proofs of various theorems are constructive and easily yield algorithms to solve problems.

However, rapid advances in this field were only made in the last 50 years with advances in computing. Now there are many journals and conferences devoted to the study of this subject.

4 Choice of Problems

One interesting characteristic of the field of Algorithms, is that often the most innovative ideas come not from the "experts" in the field, but from young graduate students whose minds are not as yet trained to think in a particular way. Often they have their own ways of "viewing" problems, and these can often lead to new insights for old problems. The problems are often easy to describe, and frequently does not require a huge amount of knowledge to start working on a problem.

There are many many sub-areas in the field of algorithms - this can be seen by the proliferation of conferences and workshops in thie field such as online algorithms, approximation algorithms, dynamic algorithms, efficient algorithms, geometric algorithms, graph algorithms, string matching algorithms, algebraic algorithms, biological algorithms etc.

To illustrate the beauty of algorithms I have chosen topics that are more readily understandable to a non-specialist. Sometimes, the background needed to explain a particular result requires more mathematical knowledge than I wish to assume, or simply requires knowledge about other results. I have tried to pick examples that are related to real-life problems, or fundamental problems that are useful in developing methods that are really useful in practice, and at the same time easy to explain.

5 Binary Search

Binary search is a simple algorithm that is widely used.

6 Stable Marriage Problem

This problem finds applications in a variety of situations where we are trying to find an "optimal" pairing of candidates. One situation arises in the assignment of medical interns to hospitals, where an algorithm based on this approach is used, once the interns and the hospitals submit their "preference" lists.

In the most basic version of the problem we are given a set of n boys and n girls. Each girl

provides a preference list of all the boys such that the first element in the list is the boy that she likes the most and the last element in the list is the least preferred boy. She is required to rank order all the boys. In a similar manner, each boy rank orders all the girls. The preference lists of the girls and the boys contain all the individuals of the opposite sex. A pairing of each boy with a distinct girl is called a marriage, and the matched pairs are called couples. So for example, let us assume that there are three boys and three girls. Suppose that the boys are called John, Sanjeev and Pierre and the girls are called Janet, Saira and Marie. Each boy submits a rank ordered preference list of the girls - so for example, the list submitted by the boys could be:

John: Saira, Marie, Janet.
Sanjeeev: Marie, Saira, Janet.
Pierre: Marie, Janet, Saira.

The lists submitted by the girls could be:

Janet: Sanjeev, Pierre, John.
Saira: Pierre, Sanjeev, John.
Marie: John, Pierre, Sanjeev.

A marriage M is a pairing of each boy with a distinct girl. A marriage is called *unstable* if the following is true: There is a pair of individuals, who are not married to each other, but both prefer each other to their current mates. So for example, the marriage $(John, Saira), (Sanjeev, Janet), (Pierre, Marie)$ is unstable since Sanjeev clearly prefers Saira to his current partner Janet. At the same time, Saira prefers Sanjeev to her current mate John. This is an unstable marriage. The marriage is called unstable, regardless of how the jilted partners John and Janet like each other.

The most interesting fact is that a stable marriage always exists! Moreover, there is a simple algorithm that can find it. (If only Computer Scientists were allowed to run society, and people's preference lists did not change!)

The algorithm proceeds in parallel rounds, with the boys making proposals. Initally everyone is unengaged.

Algorithm for finding a stable marriage

ALGORITHM 6.1.

Each currently unengaged boy proposes to the most preferred girl that he has not proposed to until now.

Every girl who receives a proposal checks her preference list. If she is unengaged, she accepts the best proposal, turning down the other proposals that she receives. If she is currently engaged, she breaks the engagement if she receives a better proposal (naturally(!)) and accepts the new proposal, and gets engaged to the most preferred boy among the ones who proposed to her.

In the next round, the boys who lost their fiancés as well as the ones who are unengaged, will propose. If ever we reach a situation where everyone is engaged then the algorithm halts and outputs this solution as a stable marriage.

We will illustrate this algorithm by an running it on the above example to make it clear. In the first round, all three boys propose to their top choice. Thus John proposes to Saira, Sanjeev to Marie and Pierre to Marie. Note that Saira receives one proposal and accepts it,

getting engaged to John. At the same time Marie receives two proposals, and prefers Pierre to Sanjeev. She gets engaged to Pierre and turns down Sanjeev's proposal. In the next round, Sanjeev proposes to the next girl on his list, namely Saira. Saira breaks her engagement with John, since she receives a preferred proposal and gets engaged to Sanjeev. In the next round, John proposes to Marie, the next girl on his list. Marie prefers the new proposal from John, and breaks her engagement with Pierre and gets engaged to John. In the next round, Pierre proposes to Janet. This is the first proposal that Janet receives and she accepts it. All parties are now engaged and we perform the wedding ceremonies.

The above algorithm will always terminate: because, when a boy proposes to the last girl in his list all the other girls are engaged (to the other $n-1$ boys), so the remaining girl (who is not engaged) has to accept his proposal.

PROPOSITION 6.1. The marriage found by the above algorithm is *stable*.

Proof. We can informally argue this as follows. Assume that there is a boy B1 who prefers G2 to his partner G1. At the same time G2 prefers B1 to her partner B2. Since B1 prefers G2, B1 must have proposed to G2 before proposing to G1 and was turned down (either immediately, or was engaged for a while before G2 broke the engagement). In either case, G2 gets a partner who is preferable to B1. Hence the partner that G2 ends up with, must be a preferred choice over B1. This is a contradiction to the assumption that G2 prefers B1 to her partner B2.

To see this point, observe that in the stable marriage produced by the algorithm there is no unstable pair. For example, note that Sanjeev prefers Marie to Saira, his wife. However, he did propose to Marie, who turned him down for Pierre, and ended up with an even better partner John (from her point of view).

Interestingly, the algorithm can also be run with the girls proposing to the boys. This may produce a *different* stable marriage. In fact, the solution is far from unique. There are situations when there is a large number of possible stable marriages. In fact, the sex that proposes has a distinct advantage in this approach. With the way in which we have described the algorithm, the boys end up with the best choices they possibly can in any stable solution. The girls end up with the worst possible solution from their point of view! (The reader may want to run the algorithm with the girls making the proposals to see that a solution can be derived rather easily, which is clearly better from the girls point of view.)

7 Primality

Prime numbers are easy to define. A number is said to be *prime* if it is divisible only by 1 and itself. For example the numbers 2 and 3 are both prime. The number 4 is not prime as 2 divides 4. The numbers 5 and 7 are prime, but 6 is not prime as both 2 and 3 divide 6. Given an arbitrary number n, how can we tell whether or not it is prime. While this task is easy for small numbers, simply check to see if a number $j < n$ divides it or not; it is actually quite computationally intensive for large numbers. For example, how can we check if 109990034572439 is prime or not? We could check for each number $109990034572439 > j > 1$ whether or not j divides it. However, this could turn out to be an extremly slow algorithm. Is there no better way to tell whether or not this number is prime?

Sieve Method: We first study an old algorithm due to Eratosthenes[1] called the Sieve Method.

[1] Eratosthenes was a famous scholar and the director of the famous library of Alexandria. One of his many achievements was to accurately estimate the diameter of the earth.

The algorithm works as follows. Suppose we want to know whether or not p is prime. First we put all the integers $1, 2, \ldots, p$ in a list. Let q be the first prime number 2 (initially). The algorithm will modify the value of q later. We go down the list and remove all multiples of the prime q. When $q = 2$ we remove all the even numbers from the list, namely 4, 6, 8 etc. These number are not prime since they are multiples of q (in this case multiples of 2).

8 Greatest Common Divisor Algorithm

Euclid's algorithm for computing the Greatest Common Divisor (GCD) of two numbers a and b is one of the oldest known algorithms. The GCD of a pair of numbers is a number that divides both numbers perfectly, and at the same time is the largest number with this property. For example, the greatest common divisor of 30 and 36 is 6. The common divisors of these two numbers are 2,3 and 6 with 6 being the largest common divisor.

Assume that $a > b \geq 0$. In the algorithm, we first check if b divides a. If so, we are done. b is the GCD. Otherwise, we remove b from a to obtain the pair b and $a - b$. We apply the same algorithm to this pair of (smaller) numbers.

For example, take the numbers 12 and 3. Since 3 divides 12, the GCD is 3. Now consider 12 and 9. 9 does not divide 12. So we now consider the pair 9 and 12-9=3. We apply the algorithm on this pair of numbers. Since 3 divides 9, the algorithm terminates with 3 as the answer.

The key idea is to show that either b divides a or $GCD(a, b) = GCD(b, a - b)$. Let us apply this algorithm to the numbers 99 and 81. $GCD(99, 81) = GCD(81, 18) = GCD(18, 63) = GCD(18, 45) = GCD(18, 27) = GCD(18, 9)$. The algorithm now terminates with 9 as the answer. Note that at the stage when the algorithm has the pair 81, 18 we can actually jump directly to 18,9 after subtracting 18 from 81 four times. This can substantially reduce the number of steps that the algorithm takes to compute the GCD. In fact, we argue next that this is indeed a very fast algorithm. Another (equivalent) way to describe the algorithm us by using the mod operation. $a \bmod b = r$ where r is the remainder when we repeatedly subtract b from a as long as we are left with a non-negative number. Hence $5 \bmod 3 = 2$. We subtract 3 from 5 once, and cannot subtract 3 again, otherwise we will get a negative number. Similarly $11 \bmod 5 = 1$. We subtract 5 from 11 twice and the remainder is 1.

Hence $GCD(a, b) = a$ if $b = 0$ otherwise $GCD(a, b) = GCD(b, a \bmod b)$.

Number of steps taken by Euclid's Algorithm: There is a rather interesting connection between the running time of Euclid's algorithm, and Fibonacci numbers [2].

If Euclid's algorithm takes k steps of re-writing the pair (a, b) then we can prove that $a \geq f_{k+2}$ and $b \geq f_{k+1}$. From this it follows that if $b < f_{k+1}$ for some value k, then the algorithm takes at most k steps. Each "step" involves computing $a \bmod b$ which is a non-trivial operation for large numbers. However, we can assume that this can be performed in a polynomial number of steps in the size of the numbers.

For example, if we have a pair of numbers (a, b) for which we are trying to compute the GCD, then suppose $b < f_{k'+1}$ for some value k'. If $b = 12$ then $k' = 6$ since $f_7 = 13$. In this case, there are at most 6 computations of the mod function.

[2] Fibonacci numbers are easy to define. Let $f_0 = 0, f_1 = 1$. Define $f_i = f_{i-1} + f_{i-2}$. Thus $f_2 = 1 + 0 = 1$. Similarly, $f_3 = f_2 + f_1 = 1 + 1 = 2$. $f_4 = f_3 + f_2 = 2 + 1 = 3$, and $f_5 = f_4 + f_3 = 3 + 2 = 5$. Note that $f_6 = f_5 + f_4 = 5 + 3 = 8$. $f_7 = f_6 + f_5 = 8 + 5 = 13$. These number actually increase very rapidly, and have what is called "exponential growth". These numbers were originally studied in the context of rabbit populations. If we start with a pair of rabbits, and each female rabbit delivers a pair of offsprings, every month starting at the age of 2 months, then the number of pairs of rabbits after n months is f_n. This assumes that the rabbits do not die (if they have a life span of a few years, then this assumption is true for the few years that we decide to breed rabbits). This also assumes that half the rabbits born are female.

The proof of the claim is by a method called induction. We first prove that the claim is true for $k = 1$. (If we are claiming that the claim is true for all integers $k > 0$ then it must be true for $k = 1$.) If we perform one mod computation then we know that $b > 0$. Since $a > b > 0$ we have that $b \geq 1$ and $a \geq 2$. Thus $a \geq f_3 = 2$ and $b \geq f_2 = 1$. We now show that if the claim is true for $k = m - 1$ (induction hypothesis) then it must also be true for $k = m$. Thus the claim is true for $k = 2$, since it is true for $k = 1$. If its is true for $k = 2$ then it must be true for $k = 3$ etc. Suppose we have m computations of the mod function. In the first mod computation we replace $GCD(a, b)$ by $GCD(b, a \bmod b)$. After this if we only have $m-1$ mod computations then we can claim that $b \geq f_{m-1+2}$ and $a \bmod b \geq f_{m-1+1}$. This must be true by the induction hypothesis. Note that b is the first parameter and $a \bmod b$ is the second parameter. Since $b \geq f_{m+1}$ the claim about b follows. Note that $a \geq b + a \bmod b$. This is because $a > b$. By the induction hypothesis the right hand side (RHS) is at least $f_{m+1} + f_m = f_{m+2}$. Thus $a \geq f_{m+2}$. This completes the proof.

9 Finding the k^{th} smallest

Given a collection of n numbers, how should we find the smallest number? The general problem can be posed as follows: given a collection of n numbers, how can we find the k^{th} smallest number? For example, when $k = 1$ this corresponds to the above problem of finding the smallest number. When $k = 2$ this corresponds to the problem of finding the second smallest number etc. One simple algorithm can be used to find the smallest element – we simply scan all the numbers, keeping track of the minimum element "seen so far". For example, if the collection of numbers was 34, 51, 21, 11, 13, 17. We would consider the first element 34. This would be the minimum element "seen so far". We would then compare this to the second element 51. Since $34 < 51$ we would not change the minimum element seen so far. When we consider 21, since $21 < 34$ we would update the minimum element seen so far to 21. When we scan the fourth element 11, we would update the minimum element seen so far once more to 11, since $11 < 21$. After that, all the elements we encounter will be greater than 11, so we will not make any changes. The algorithm will then terminate with 11 as the answer. Note that the running time of this algorithm is $O(n)$. The number of updates to the minimum element may be a lot fewer.

How can we find the second smallest element? One method would involve doing a second scan of the set, to find the smallest element after temporarily removing the smallest element. What if we wanted to find the 3^{rd} smallest element, and so on? Would we have to scan the set three times? If we wanted to find the k^{th} smallest element, this technique would involve k scans of the set. It turns out that there are much better ways that can be used to solve the problem, especially if k is large.

One approach can be described as follows. We first pick a partitioning element, say "17" from the above collection. We then compare all the elements to the chosen element to create two sets: one set is called SMALL and the other is called BIG. For the above example, SMALL would contain elements 11 and 13. BIG would containt the elements 34,51, and 21. SMALL denotes the elements that are smaller than the chosen element, and BIG denotes the elements that are larger than the chosen elements. Now we compare k with the size of the set SMALL. If $k \leq |SMALL|$ then we run the algorithm again on the set SMALL, looking for the k^{th} smallest element. If $k = |SMALL| + 1$ then the chosen element is the element we are looking for. If $k > |SMALL| + 1$ then we run the algorithm on the set BIG, except that now we are looking for the $(k - (|SMALL + 1))^{th}$ element in the set BIG. We keep running this algorithm until the set of elements becomes very small and then we can solve the problem easily. Let us illustrate

this by an example.

If $k = 2$ and we were looking for the second smallest element. For the previous choice of chosen element, $|SMALL| = 2$. Note that since $k \leq 2$ we can only consider the set SMALL since it must contain the second smallest element. If $k = |SMALL| + 1$ then we know that the chosen element, namely 17 is the 3^{rd} smallest element. Suppose $k = 4$; now we know that if look in the set BIG, we are actually looking for the smallest element since we have eliminated 3 elements from consideration. Namely, elements 17 and the set SMALL, containing 11 and 13. Since the 4^{th} smallest element is 21, it is actually the smallest element in the set BIG. We now continue looking for the smallest element in the set BIG. This time we choose an element from the set BIG, say "34". With this choice of element, the sets SMALL and BIG are 21 and 51 respectively. Since $k = 1$ now, we continue looking for the smallest element in SMALL. Since SMALL only has a single element, the process terminates and we stop and we have found the answer, namely 21 which is the 4^{th} smallest element of the original set.

For this algorithm we can actually prove that the total (expected) running time is $O(n)$ for any value of k provided we choose the partitioning element randomly. In other words, we could pick any element from the given set. The choice is made in an impartial manner, and the choice of each element is equally likely, like in a lottery. This kind of choice is actually extremely useful and often yields simple and natural algorithms.