U.C. Berkeley — CS170: Intro to CS Theory
Lecturer: Umesh Vazirani

Handout N1
January 20, 2004

Notes 1 for CS170

1 Topics to be covered

1. Data structures and graph algorithms—Quick review of prerequisites.

2. *Divide-and-conquer algorithms* work by dividing problem into two smaller parts and combining their solutions. They are natural to implement by recursion, or a stack (e.g. binary search; Fast Fourier Transform in signal processing, speech recognition)

3. *Graph algorithms*: depth-first search (DFS), breadth-first search (BFS) on graphs, with properties and applications (e.g. finding shortest paths, as in trip planning)

4. *Randomization*: how to use probability theory to come up with fast and elegant algorithms and data structures.

5. *Data Compression*: the theory and the algorithms.

6. *Dynamic programming*: works by solving subproblems, like in divide-and-conquer, but differently.

7. *Linear Programming* (LP) solves systems of linear equations and inequalities. We will concentrate on using LP to solve other problems, (e.g. production scheduling, computing capacities of networks)

8. *NP-completeness*: many algorithms we will study are quite efficient, costing $O(n)$ or $O(n \log n)$ or $O(n^2)$ on input size n. But some problems are much harder, with algorithms costing at least $\Omega(2^n)$ (or some other *exponential*, rather than *polynomial* function of n). It is believed that we will *never* be able to solve such problem completely for large n, e.g. TSP. So we are forced to approximate them. We will learn to recognize when a problem is so hard (NP-complete), and how to devise algorithms that provide approximate if not necessarily optimal solutions.

9. The Fast Fourier Transform (or FFT) is widely used in signal processing (including speech recognition and image compression) as well as in scientific and engineering applications.

2 On Algorithms

An algorithm is a recipe or a well-defined procedure for transforming some input into a desired output. Perhaps the most familiar algorithms are those those for adding and multiplying integers. Here is a multiplication algorithm that is different the algorithm you learned in school: write the multiplier and multiplicand side by side, say 13x15=195. Repeat the following operations—divide the first number by 2 (throw out any fractions) and multiply the second by 2, until the first number is 1. This results in two columns of numbers. Now cross out all rows in which the first entry is even, and add all entries of the second column that haven't been crossed out. The result is the product of the two numbers.

In this course we will ask a number of basic questions about algorithms:

- Does it halt?

  The answer for the algorithm given above is clearly yes, *provided* we are multiplying positive integers. The reason is that for any integer greater than 1, when we divide it by 2 and throw out the fractional part, we always get a smaller integer which is greater than or equal to 1.

- Is it correct?

  To see that the algorithm correctly computes the product of the integers, observe that if we write a 0 for each crossed out row, and 1 for each row that is not crossed out, then reading from bottom to top just gives us the first number in binary. Therefore, the algorithm is just doing the standard multiplication in binary, and is in fact essentially the algorithm used in computers, which represent numbers in binary.

- How much time does it take?

  It turns out that the algorithm is as fast as the standard algorithm. (How do we implement the division step, which seems harder than multiplication, in a computer?) In particular, if the input numbers are n digits long, the algorithm takes $O(n^2)$ operations. Later in the course, we will study a faster algorithm for multiplying integers.

- How much memory does it use? (When we study cache-aware algorithms, we will ask more details about what kinds of memory are used, eg cache, main memory, etc.)

The history of algorithms for simple arithmetic is quite fascinating. Although we take them for granted, their widespread use is surprisingly recent. The key to good algorithms for arithmetic was the positional number system (such as the decimal system). Roman numerals (I, II, III, IV, V, VI, etc) were just the wrong data structure for performing arithmetic efficiently. The positional number system was first invented by the Mayan Indians in Central America about 2000 years ago. They used a base 20 system, and it is unknown whether they had invented algorithms for performing arithmetic, since the Spanish conquerors destroyed most of the Mayan books on science and astronomy.

The decimal system that we use today was invented in India in roughly 600 AD. This positional number system, together with algorithms for performing arithmetic were transmitted to Persia around 750 AD, when several important Indian works were translated into arabic. In particular, it was around this time that the Persian mathematician Al-Khwarizmi wrote his arabic textbook on the subject. The word "algorithm" comes from Al-Khwarizmi's name. Al-Khwarizmi's work was translated into Latin around 1200 AD, and the positional number system was propagated throughout Europe from 1200 to 1600 AD.

The decimal point was not invented until the 10^{th} century AD, by a Syrian mathematician al-Uqlidisi from Damascus. His work was soon forgotten, and five centuries passed before decimal fractions were re-invented by the Persian mathematician al-Kashi.

With the invention of computers in this century, the field of algorithms has seen explosive growth. There are a number of major successes in this field: not all of which we can discuss in CS 170:

- Parsing algorithms—these form the basis of the field of programming languages (CS 164)
- Fast Fourier transform—the field of digital signal processing is built upon this algorithm. (CS 170, EE)
- Linear programming—this algorithm is extensively used in resource scheduling. (CS 170, IEOR)
- Sorting algorithms - until recently, sorting used up the bulk of computer cycles. (CS 61B, CS 170)
- String matching algorithms—these are extensively used in computational biology. (CS 170)
- Number theoretic algorithms—these algorithms make it possible to implement cryptosystems such as the RSA public key cryptosystem. (CS 276)
- Numerical algorithms, for evaluating models of physical systems, replacing the need for expensive or impossible physical experiments (e.g. climate modeling, earthquake modeling, building the Boeing 777) (Ma 128ab, and many engineering and science courses)
- Geometric algorithms, for computing and answering questions about physical shapes that appear in computer graphics and models of physical systems.

The algorithms we discuss will be in "pseudo-code", so we will not worry about certain details of implementation. So we might say "choose the i-th element of a list" without worrying about exactly how the list is represented. But the efficiency of an algorithm can depend critically on the right choice of data structure (e.g. $O(1)$ for an array versus $O(i)$ for a linked list). So we will have a few programming assignments where success will depend critically on the correct choice of data structure.

3 Computing the nth Fibonacci Number

Remember the famous sequence of numbers invented in the 15th century by the Italian mathematician Leonardo Fibonacci? The sequence is represented as $F_0, F_1, F_2, \ldots$, where $F_0 = F_1 = 1$, and for all $n \geq 2$ F_n is defined as $F_{n-1} + F_{n-2}$. The first few numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, $\ldots$. F_{30} is greater than a million! The Fibonacci numbers grow exponentially. Not quite as fast as 2^n, but close: F_n is about $2^{.694\ldots n}$.

Suppose we want to compute the number F_n, for some given large integer n. Our first algorithm is the one that slavishly implements the definition:

```
function $F(n$ : integer): integer
```

```
if $n \leq 1$ then return 1
else return $F(n-1) + F(n-2)$
```

It is obviously correct, and always halts. The problem is, it is too slow. Since it is a recursive algorithm, we can express its running time on input n, $T(n)$, by a *recurrence equation*, in terms of smaller values of T (we shall talk a lot about such recurrences later in this class). So, what is $T(n)$? We shall be interested in the *order of growth* of $T(n)$, ignoring constants. If $n \leq 1$, then we do constant amount of work to obtain F_n; since we are suppressing constants, we can write $T(n) = 1$. Otherwise, if $n \geq 2$, we have:

$$T(n) = T(n-1) + T(n-2) + 1,$$

because in this case the running time of F on input n is just the running time of F on input $n-1$—which is $T(n-1)$—plus $T(n-2)$, plus the time for the addition. This is nearly the Fibonacci equation. We can change it to be exactly the Fibonacci equation by noting that the new variable $T'(n) \equiv T(n) + 1$ satisfies $T'(n) = T'(n-1) + T'(n-2)$, $T'(1) = T(1) + 1 = 2$ and $T'(0) = T(0) + 1 = 2$. Comparing this recurrence for $T'(n)$ to the Fibonacci recurrence, we see $T'(n) = 2F_n$ and so $T(n) = 2F_n - 1$. In other words, the running time of our Fibonacci program *grows as the Fibonacci numbers*—that is to say, way too fast.

Can we do better? This is the question we shall always ask of our algorithms. The trouble with the naive algorithm is wasteful use of recursion: The function F is called with the same argument over an over again, exponentially many times (try to see how many times $F(1)$ is called in the computation of $F(5)$). A simple trick for improving this performance is to *memoize* the recursive algorithm, by remembering the results from previous calls. Specifically, we can maintain an array $A[0 \ldots n]$, initially all zero, except that $A[0] = A[1] = 1$. This array is updated before the return step of the F function, which now becomes $A[n] := A[n-1] + A[n-2]$; return $A[n]$. More importantly, *it is consulted in the beginning* of the F function, and, if its value is found to be non-zero—that is to say, defined— it is immediately returned. But then of course, there would be little point in keeping the recursive structure of the algorithm, we could just write:

```
function $F(n : \text{integer})$: integer
array $A[1 \ldots n]$ of integers, initially all 0
$A[0] := A[1] := 1$
for $i = 2$ to $n$ do:
$A[n] := A[n-1] + A[n-2]$
return $A[n]$
```

This algorithm is correct, because it is just another way of implementing the definition of Fibonacci numbers. The point is that its running time is now much better. We have a single "for" loop, executed $n-1$ times. And the body of the loop takes only constant time (one addition, one assignment). Hence, the algorithm takes only $O(n)$ operations.

This is usually an important moment in the study of a problem: We have come from the naive but prohibitively exponential algorithm to a polynomial one.

Now, let us be more precise in our accounting of the time requirements for all these methods, we have made a grave and common error: We have been too liberal about *what constitutes an elementary step of the algorithm.* In general, in analyzing our algorithms we shall assume that each arithmetic step takes unit time, because the numbers involved will be typically small enough—about n, the size of the problem— that we can reasonably expect them to fit within a computer's word. But in the present case, we are doing arithmetic on huge numbers, with about n bits—remember, a Fibonacci number has about $.694\ldots n$ bits—and of course we are interested in the case n is truly large. When dealing with such huge numbers, and if exact computation is required, we have to use sophisticated *long integer packages.* Such algorithms take $O(n)$ time to add two n-bit numbers—hence the complexity of the two methods was not really $O(F_n)$ and $O(n)$, but instead $O(nF_n)$ and $O(n^2)$, respectively; notice that the latter is still exponentially faster.

4 Algorithm Design Paradigms

Every algorithmic situation (every computational problem) is different, and there are no hard and fast algorithm design rules that work all the time. But the vast majority of algorithms can be categorized with respect to the *concept of progress* they utilize.

In an algorithm you want to make sure that, after each execution of a loop, or after each recursive call, some kind of *progress* has been made. In the absence of such assurance, you cannot be sure that your algorithm will even terminate. For example, in depth-first search, after each recursive call you know that another node has been visited. And in Dijkstra's algorithm you know that, after each execution of the main loop, the correct distance to another node has been found. And so on.

Divide-and-Conquer Algorithms: These algorithms have the following outline: To solve a problem, divide it into subproblems. Recursively solve the subproblems. Finally glue the resulting solutions together to obtain the solution to the original problem. Progress here is measured by how much smaller the subproblems are compared to the original problem.

You have already seen an example of a divide and conquer algorithm in cs61B: mergesort. The idea behind mergesort is to take a list, *divide* it into two smaller sublists, *conquer* each sublist by sorting it, and then *combine* the two solutions for the subproblems into a single solution. These three basic steps—divide, conquer, and combine—lie behind most divide and conquer algorithms.

With mergesort, we kept dividing the list into halves until there was just one element left. In general, we may divide the problem into smaller problems in any convenient fashion. Also, in practice it may not be best to keep dividing until the instances are completely trivial. Instead, it may be wise to divide until the instances are reasonably small, and then apply an algorithm that is faster on small instances. For example, with mergesort, it might be best to divide lists until there are only four elements, and then sort these small lists quickly by other means. We will consider these issues in some of the applications we consider.

5 Maximum/minimum

Suppose we wish to find the minimum *and* maximum items in a list of numbers. How many comparisons does it take? The obvious loop takes $2n - 2$:

```
m = L[1]; M = L[1];
for i=2, length(L)
```

```
     m = min(m, L[i])
     M = max(M, L[i])
```

Can we do it in fewer? A natural approach is to try a divide and conquer algorithm. Split the list into two sublists of equal size. (Assume that the initial list size is a power of two.) Find the maxima and minimum of the sublists. Two more comparisons then suffice to find the maximum and minimum of the list.

```
function [m,M] = MinMax(L)
  if (length(L) < 3)
     if (L[1] < L[length(L))
        return [ L[1], L[length(L)] ]
     else
        return [ L[length(L)], L[1] ]
  else
     [ m1, M1 ] = MinMax( L[ 1 : n/2 ] )     ... find min, max of first half of L
     [ m2, M2 ] = MinMax( L[ n/2+1 : n ] )  ... find min, max of last  half of L
     return [ min(m1,m2) , max(M1,M2) ]
```

Hence, if $T(n)$ is the number of comparisons, then $T(n) = 2T(n/2) + 2$ and $T(2) = 1$.

How do we solve such a recursion? Suppose that we “guess” that the solution is of the form $T(n) = an + b$. Then a and b have to satisfy the system of equations

$$\begin{cases} 2a + b = 1 \\ an + b = 2\left(a\frac{n}{2} + b\right) + 2 \end{cases}$$

which solves to $b = -2$ and $a = 3/2$, so that that (for n a power of 2), $T(n) = 3n/2 - 2$, or 75% of the obvious algorithm. One can in fact show that $\lceil 3n/2 \rceil - 2$ comparisons are *necessary* to find both the minimum and maximum (CLR Problem 10.1-2).

A similar idea can be applied to a sequential algorithm. Suppose for simplicity that the array has an odd number of entries, then the algorithm is as follows:

```
function [m,M] = MinMax(L)
  curr_min = min(L[1],L[2])
  curr_max = max(L[1],L[2])
  for (i=3,i< length(L),i=i+2)
     m = min(L[i],L[i+1])
     M = max(L[i],L[i+1])
     if m < curr_min
        curr_min = m
     if M > curr_max
        curr_max = M
  return [ curr_min , curr_max ]
```

Each iteration of the `for` loop requires three comparisons, and the initialization of the variables requires one comparison. Since the loop is repeated $(n-2)/2$ times, we have a total of $3n/2 - 2$ comparisons, for even n.