COMPSCI 330: Design and Analysis of Algorithms 1/14/2016

Introduction

Lecturer: Debmalya Panigrahi *Scribe: Tianqi Song, Tianyu Wang*

1 Overview

Welcome to Computer Science 330. This lecture is an introduction to algorithms. It covers asymptotic analysis and recurrence relations. We demonstrate these principles with simple algorithms such as Fibonacci numbers, searching, and sorting.[1]

2 Calculating Fibonacci Numbers

Fibonacci numbers can be defined inductively. The nth Fibonacci number, $F(n)$ is

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise.} \end{cases} \tag{1}$$

It is easy to write an algorithm based on this definition, however as we will see, this algorithm is very slow.

Algorithm 1 Calculate the nth Fibonacci number

```
1: function FIB(n)
2:     if n = 0 then
3:         return 0
4:     if n = 1 then
5:         return 1
6:     else
7:         return FIB(n − 1) + FIB(n − 2)
```

What is the recurrence for the running time of Algorithm 1? It is

$$T(n) = T(n-1) + T(n-2) + O(1). \tag{2}$$

Unfortunately, the solution to Equation 2 is exponential. The reason is that $T(n) = T(n-1)+T(n-2) \geq 2T(n-2) \geq 2^2 T(n-4) \geq \cdots \geq 2^{\frac{n}{2}} T(0) = 2^{\frac{n}{2}}$. We can also see its computation tree in Figure 1 and detailed analysis in chapter 27.1 of CLRS, 3rd Edition. We often say that exponential running time is bad, but why is this the case? According to Moore's Law, the speed of computers roughly doubles each year. That means that if finding the kth Fibonacci number takes 1 hour this year, it will take 1 hour to find the $k+1$th Fibonacci number next year. If we want to be able to calculate the $k+100$th Fibonacci number in 1 hour, we would need to wait 100 years! This means that exponential running times are bad, even though computers get much faster each year.

[1] Most of the material in this note is from a previous note by Samuel Haney for this class in Fall 2014.

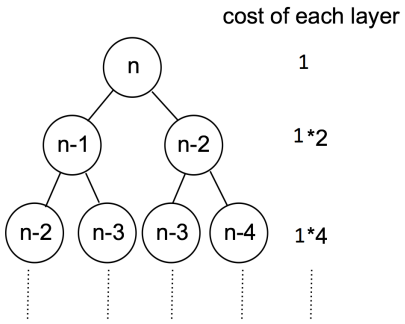

Figure 1: Computation tree of Algorithm 1 which is a recursive way to compute Fibonacci number. This algorithm is not efficient because some subproblems are unnecessarily computed repetitively.

Let's see if we can improve the running time of calculating the nth Fibonacci number by using an iterative algorithm instead of recursion.

Algorithm 2 Calculate the nth Fibonacci number iteratively

```
1: function FIB-ITER(n)
2:     F[0] ← 0
3:     F[1] ← 1
4:     for i ← 2 to n do
5:         F[i] ← F[i − 2] + F[i − 1]
6:     return F[n]
```

The running time of Algorithm 2 appears to be much better than Algorithm 1 (see its computation tree in Figure 2). Naively, we might think the recurrence relation is

$$T(n) \approx n. \tag{3}$$

However, this assumes the we can add any two numbers in one unit of time. This is a bad assumption! In fact, the time it takes to add n-bit numbers depends on n. The size of the nth Fibonacci number is on the order of n bits, so these additions take a very long time. If we assume that adding two n bit numbers takes n time, we get a new recurrence:

$$T(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1. \tag{4}$$

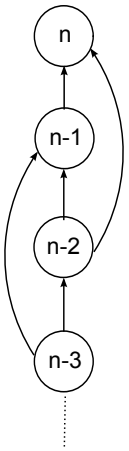

Figure 2: Computation tree of Algorithm 2 which is a iterative way to compute Fibonacci number. This algorithm is efficient because each subproblem is only computed once.

While Equation 3 is linear, Equation 4 is quadratic since the n^2 term dominates. This running time is still much better than Algorithm 1. This brings us to our next topic: what does it mean to say that the n^2 term dominates?

3 Asymptotic Analysis

Definition 1. *For functions f and g both defined on the natural numbers, we say that f is $O(g)$ ($f = O(g)$) if $\exists N, c \in \mathbb{N}$, $c > 1$ such that $\forall n > N$, $f(n) \leq c \cdot g(n)$.*

Example 1. Let $f(n) = 2n^2 + 2n + 1$ and $g(n) = n^2$. We would like to show that $f = O(g)$. For this, we set $c = 3$. In order to satisfy Definition 1, we must have

$$\begin{aligned} 2n^2 + 2n + 1 &< 3n^2 \\ 2n + 1 &< n^2. \end{aligned} \tag{5}$$

Equation 5 is satisfied for $n > 3$, so we set $N = 3$.

Example 2. Let $f(n) = \frac{n}{1000000}$ and $g(n) = \log n$. For small values of n, $f(n) \ll g(n)$. However, as n grows to very large numbers, f will dominate g. Therefore, it is not true that $f = O(g)$, even though it appears that way for small numbers.

Definition 2. *For functions f and g both defined on the natural numbers, we say that f is $\Omega(g)$ ($f = \Omega(g)$) if $\exists N, c \in \mathbb{N}$, $c > 1$ such that $\forall n > N$, $c \cdot f(n) \geq g(n)$.*

In Example 2, $f = \Omega(g)$.

Definition 3. *For functions f and g both defined on the natural numbers, we say that f is $\Theta(g)$ ($f = \Theta(g)$) if $f = O(g)$ and $f = \Omega(g)$.*

Example . Let $f(n) = n + 1000000$ and $g(n) = 2n$. Then f is $\Theta(g)$.

Definition 4. *For functions f and g both defined on the natural numbers, we say that f is $o(g)$ ($f = o(g)$) if $f = O(g)$ and $f \neq \Theta(g)$.*

In Example 2, g is $o(f)$.

Definition 5. *For functions f and g both defined on the natural numbers, we say that f is $\omega(g)$ ($f = \omega(g)$) if $f = \Omega(g)$ and $f \neq \Theta(g)$.*

4 Euclid's Algorithm

In this section, we give one of the oldest algorithms that is still used today: Euclid's Algorithm, for finding the gcd of a pair of numbers.

Algorithm 3 Euclid

```
1: function GCD(a, b)
2:     if b = 0 then                ▷ Assume WLOG that a > b. Otherwise the names can be reversed
3:         return a
4:     else
5:         return GCD(b, a mod b)
```

Proof: Let $a = bp + q$. If α is a common divisor of a and b, it is also a divisor of $a - bp$ which is q, so α is a common divisor of b and q. If β is a common divisor of b and q, it is also a divisor of $bp + q$ which is a, so β is a common divisor of a and b. Therefore, a number is a common divisor of a and b iff it is a common divisor of b and q, and then $GCD(a,b) = GCD(b, a \mod b)$.

4.1 Worst case analysis of Euclid's algorithm

Notice that the algorithm stops when the second input becomes 1. Denote by (a_i, b_i) pairs of a and b at the i-th last iteration. Note that $b_i = a_{i+1} \mod b_{i+1}$ or $a_{i+1} = kb_{i+1} + b_i$ with $k \geq 1$ and $b_i < b_{i+1}$. Note that the worst case runtime happens when $k = 1$. Otherwise, the values of a and b decrease faster and the algorithm terminates faster. This gives us at least (in terms of term-wise magnitude) the Fibonacci sequence. In other words, the worst case happens when the two inputs are no less than two consecutive Fibonacci numbers, or simply $b_i \geq F_{i-1}$ where F_m is the m-th Fibonacci number. On the other hand, recall from Section 2 that the Fibonacci sequence increases exponentially in magnitude. Therefore, Euclid algorithm takes $O(\log b)$ time to terminate. So the worst case time complexity is $O(\log b)$ if we assume that the $\mod$ operation takes constant time.

5 Searching

In this section, we analyze a two searching algorithms, linear search and binary search. Throughout this section, we use k as the key we are searching for.

5.1 Linear Search

The algorithm for linear search is given below.

Algorithm 4 Linear search

```
1: function LIN-SEARCH(A, k)
2:     for i ← 1 to n do
3:         if A[i] = k then
4:             return true
5:     return false
```

Algorithm 5 Linear search with recursion

```
1: function LIN-SEARCH-RECURSIVE(A[1...n], k)
2:     if A[1] = k then
3:         return true
4:     if A is empty then
5:         return false
6:     else
7:         return LIN-SEARCH-RECURSIVE(A[2...n], k)        ▷ Note that A[2...n] is empty if n = 1
```

We would like to write a recursion relation for the running time, and this is easier if we rewrite the algorithm recursively.

We can now write a recurrence relation for the running time.

$$T(n) = T(n-1) + 2 = T(n-1) + O(1) \tag{6}$$

If we expand out Equation 6, we get

$$\begin{aligned} T(n) &= T(n-1) + O(1) \\ &= T(n-2) + O(1) + O(1) \\ &\quad \vdots \\ &= O(1) + \cdots + O(1) \text{ (n times)} \\ &= O(n). \end{aligned}$$

We now give a more rigorous proof that $T(n) = O(n)$.

Proof. Proof is by induction. The base case $T(1) = O(1)$ holds. Our inductive hypothesis is

$$T(n-1) \leq c \cdot (n-1),$$

and we must show

$$T(n) \leq c \cdot n.$$

We choose $c = 2$ which gives

$$T(n) \leq c(n-1) + 2 = 2(n-1) + 2 = 2n = cn.$$

□

Algorithm 6 Binary search

```
1: function BIN-SEARCH(A[1...n], k)
2:     if A is empty then
3:         return false
4:     if k = A[n/2] then
5:         return true
6:     if k > A[n/2] then
7:         return BIN-SEARCH(A[n/2+1, ..., n], k)
8:     if k < A[n/2] then
9:         return BIN-SEARCH(A[1, ..., n/2-1], k)
```

5.2 Binary Search

It is possible to search a sorted list much faster than linear time. We present an algorithm that accomplishes this.

The recurrence for Algorithm 6 is

$$T(n) = T(n/2) + O(1).$$

We give an inductive proof that the solution is $T(n) = O(\log n)$.

Proof. Proof is by induction. The base case $T(1) = O(1)$ holds. Our inductive hypothesis is

$$T(n/2) \leq c \log_2(n/2),$$

and we must show

$$T(n) \leq c \log_2(n).$$

From our recurrence and inductive hypothesis we have

$$\begin{aligned} T(n) &\leq c \log_2(n/2) + c \\ &= c(\log_2(n/2) + 1) \\ &= c(\log_2 n - \log_2 2 + 1) \\ &= c \log_2 n \\ &= O(\log n) \end{aligned}$$

□

Therefore, Algorithm 6 takes $O(\log n)$ time. However, in order to give the proof, we first needed to guess that the correct answer was $\log n$. How did we do this? We can think about the size of the array in each of our recursive calls. Before the first call, the size is n. After the first call the size is $n/2$, then $n/4$, then $n/8$, and so on. In the kth level of recursion, the size of the array is $\frac{n}{2^k}$. Our algorithm terminates when the array has only one element remaining. Solving for k in $\frac{n}{2^k} = 1$ gives $k = \log_2 n$, which is therefore the number of levels of recursion.