CS 31: Algorithms (Spring 2019): Lecture 1

Date: 26th March, 2019 Topic: Algorithms with Numbers

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please email errors to deeparnab@dartmouth.edu.

---

1 What are Algorithms?

Algorithms solve *computational problems*. These problems have an *input* which define an *instance* of the problem. An algorithm is an object which maps inputs/instances of the problem to *solutions* or *outputs* for this problem. *Good* algorithms are correct and efficient. It takes creativity to *design* good algorithms and like all creative forms mastery in algorithms comes with practice, practice, and practice. Having said that, we know ***very little*** about algorithms, and understanding algorithms is one of the most exciting research areas in science. In this course, we will see some success stories which form the the tip-of-the algorithmic iceberg.

Correctness. An algorithm is correct if it has the desired input/output behavior. It is not trivial to figure out whether an algorithm is correct or not. Running the algorithm on a few instances and checking doesn't assure one that it will be correct always. One requires a *proof*. Indeed often times (in this course) the design of the algorithm keeps correctness in mind – the proof of correctness will follow from design.

Efficiency. Every instance of a problem is associated with a notion of *size*, a parameter which defines the magnitude of the input. Every algorithm is associated with a notion of *running time* which indicates the *efficiency* of the algorithm. The notion of time is often the *number of elementary operations* that needs to be performed to solve the problem. What is elementary is problem specific, but often clear from context.

Formally, let's use Π to denote the computational problem, and let $\mathcal{I}$ be an instance of Π and let $|\mathcal{I}|$ denote the size of the instance. Given an algorithm $\mathcal{A}$ for Π, let $T_{\mathcal{A}}(\mathcal{I})$ denote the time taken by $\mathcal{A}$ to solve $\mathcal{I}$. Throughout, we will be using the *worst case running time* notion. Given a natural number $n > 0$, we define

$$T_{\mathcal{A}}(n) := \max_{\mathcal{I} \in \Pi : |\mathcal{I}| \leq n} T_{\mathcal{A}}(\mathcal{I})$$

In plain English, any instance of Π whose size is $\leq n$ can be solved by $\mathcal{A}$ in time $T_{\mathcal{A}}(n)$.

Enough formalism. Towards concrete problems.

2 Addition

All of us see an algorithm early in grade school. Addition. What? Yes. Addition.

Given two numbers, we add them by putting them one below the other and then adding digit-by-digit, and taking care of carries, etc. It is a step-by-step method (in fact a for-loop). But why does it work? That is, why is $17 + 13$ when written down using the above algorithm gives the same answer as counting the total number of sticks if I have 17 sticks in one hand and 13 sticks in the other? Have you wondered this?

Indeed, let us first formalize the computational problem. The first question is : how are these numbers represented? We recall the *binary representation*.

Remark: *An n-bit number a is represented by a bit-array $a[0:n-1]$ where each $a[i]$ is 0 or 1, and*

$$a = \sum_{i=0}^{n-1} a[i] \cdot 2^i$$

So, for example, the number 37, whose binary representation is 100101 is represented by the bit-array $[1,0,1,0,0,1]$ which is the reverse of the binary representation. Only when we talk about numbers will it be convenient to read arrays right to left.

Now that we have refreshed our memory about bits and the binary representation, we can define the addition problem.

ADDITION
Input: Two n-bit numbers a,b expressed as bit-arrays $a[0:n-1], b[0:n-1]$.
Output: The number $c = a + b$ expressed as a bit-array.
Size: The number of bits n.

Given the definition, we are now ready to actually spell out the algorithm we learned in grade school as a pseudocode.

```
1: procedure ADD(a[0 : n − 1], b[0 : n − 1]): ▷ The two numbers are a and b
2:     Initialize carry ← 0.
3:     Initialize c[0 : n] to all zeros ▷ c[0 : n] will finally contain the sum
4:     for i = 0 to n − 1 do:
5:         (carry, c[i]) ← BIT-ADD(a[i], b[i], carry)
6:     c[n] ← carry
7:     return c
```

What are the elementary operations for this problem? We shall say that adding any three single bits forms one operation (this is called BIT-ADD above) which takes *one unit of time*. At some level this choice is arbitrary but also natural. Very soon, we will use the "Big-Oh" notation very nicely to sweep these distractions cosily under the rug to focus

on the bigger picture. For today's class, however, let this be our definition. With this definition, we can easily see that the number of BIT-ADDs is at most n. We cast this in the following theorem.

Theorem 1. The algorithm ADD adds two n bit numbers with worst case running time $T_{\text{ADD}}(n) = n$.

✍

Exercise: *Can you think of a scenario where* ADD *takes significantly smaller time to add?*

✍

Exercise: *Can you modify the above pseudo-code to add an n-bit number with an m-bit number (where $n \geq m$)? If $T(n, m)$ is the worst-case running time, what is this as a function of n and m?*

I don't know about you, but it is not utterly trivial to me that the above algorithm, when input two numbers a and b, actually returns a bit-array c which contains the sum $a + b$. That is, the number obtained by incrementing a exactly b times. The proof of this is given in the supplement.

Note that the trivial algorithm that implements the definition of addition, has running time $T(n) = 2^n - 1$ as the number b can be as large as $2^n - 1$. Thus, the above ADD algorithm is a *remarkable* algorithm. Indeed, the "place-value-system" of numbers is one of the most remarkable inventions of human kind. If you have a doubt about this, think Roman numbers and how you would add them. Or perhaps consider multiplying them.

3 Multiplication

MULTIPLICATION
Input: n-bit number x, m-bit number y expressed as bit-arrays $x[0 : n-1], y[0 : m-1]$.
Output: The number $z = x \cdot y$ expressed as a bit-array.
Size: The number of bits $n + m$.

On to multiplication. Many grade-school method of multiplication "reduces" multiplying two n-bit/digit numbers into adding n different numbers ranging from $n+1$ to $2n+1$ bits. Today, we see a different *recursive* algorithm for multiplication. Recursion is one essence of algorithm design which you should try to get in your blood :

Break the problem into smaller subproblems and let recursion take care of the smaller subproblems.

Correctness of recursive algorithms often will follow from the design. Mathematical Induction is often involved; good place to brush it up! Let us illustrate with multiplication.

```
1:  procedure MULT(x, y):
2:      ▷ The two numbers are input as bit-arrays; x has n bits, y has m bits. n ≥ m.
3:      if y = 0 then:
4:          ▷ Base Case
5:          return 0 ▷ An all zero bit-array
6:      x' ← (2x); y' ← ⌊y/2⌋ ▷ How much time does this take? See remark below.
7:      z ←MULT(x', y')
8:      if y is even then:
9:          return z
10:     else:
11:         return ADD(z, x) ▷ Time taken is the total number of bits in z and x.
```

Remark: *"Hold on!," I hear you say, "Above, you seem to have multiplied by* 2 *and divided by* 2 *and taken floors. How do we do that?" Indeed. Note that when* x *is expressed as a bit-array,* $(2x)$ *is just a left-shift. Similarly,* $\lfloor y/2 \rfloor$ *is a right-shift. For simplicity, we assume this takes* 0 *time. These are "easy" operations and today we won't even count them in our running time. In decimal notation, this would correspond to multiplying and dividing by* 10*.*

✍

Exercise: *Prove that* MULT(x, y) *is correct. See* https://www.cs.dartmouth.edu/~deepc/Courses/W19/lecs/lec16.pdf *for a proof framework.*

Define $T(n, m)$ to be the maximum time (that is, BIT-ADDS) MULT takes to multiply x, y where x is an n-bit number and y is an m-bit number. Recall, we assume adding an n bit number with an m bit number takes $\leq n$ time. We next write a *recurrence inequality* for $T(n, m)$.

Base Case: When $y = 0$, let us say $m = 0$, and define $T(n, 0) = 0$.

We assume Line 6 doesn't cost anything. Indeed, it doesn't contain any elementary operations as remarked above. Line 7 costs at most $T(n + 1, m - 1)$ time since $x' = (2x)$ has $n + 1$ bits and $y' = \lfloor y/2 \rfloor$ has $m - 1$ bits. If y is even, this is the end of it. If y is odd, then we need to add z and x. Now we use the fact that $z \leq x \cdot y$ has at most $(n + m)$ bits.

Fact 1. If x has n bits and y are m bit integers, then $x \cdot y$ has at most $(n + m)$ bits.

Proof. $x \leq 2^n$, $y \leq 2^m$ implying $x \cdot y \leq 2^n \cdot 2^m = 2^{n+m}$. □

Therefore, ADD(z, x) takes at most $(n + m)$ time. Putting the above together, we get that the function $T(n, m)$ satisfies the following *recurrence inequality*:

$$\begin{aligned} T(n, 0) &= 0 \\ T(n, m) &\leq T(n + 1, m - 1) + (n + m) \end{aligned} \tag{1}$$

Recurrence Inequalities: The heart of analyzing running times of recursive algorithms. Equation (1) is called a *recurrence* inequality; it is expressing the running time $T(n, m)$ as a function $3n$ plus $T()$ of something "smaller". Why is $(n+1, m-1)$ smaller than (n, m)? Because the lesser of the two numbers is becoming strictly smaller.

Recurrence inequalities form the bedrock of analyzing the efficiency of recursive algorithms, and in this class we will see how to solve a general class of them. The general way I like to think of it is the following picture (shown in Figure 1), which I call the *kitty method*[1].

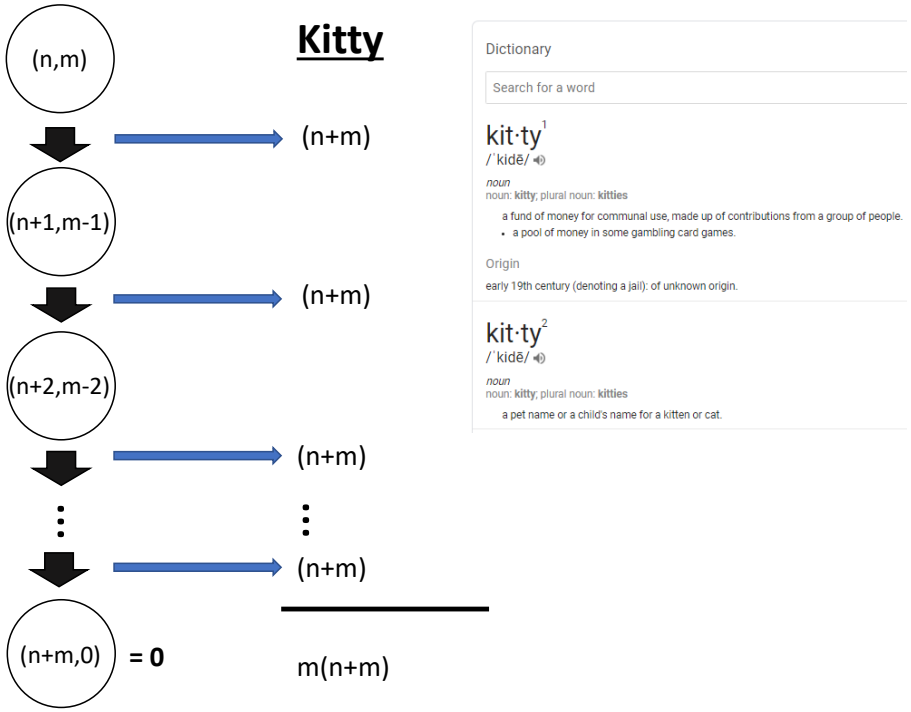

Figure 1: The circles contain the various sizes. As we go down the sizes become smaller and smaller till we reach a small enough size for which we can figure out the running time directly. In particular, when the size becomes 0, the running time becomes 0. However, breaking the problem is not free. To break every circle you need to pay some in the "kitty". This is given by the extra terms other than the $T(\cdot)$'s. In this case, each break "costs" $(n+m)$. In the end, we just add everything in the kitty to get the final answer.

Once we know the answer from the picture above, we can formally prove it as well. This is shown below.

[1]No one else (except students who have taken the class with me in the past) calls it by this name.

Theorem 2. MULT takes $T(n,m) \leq m(n+m)$ time (i.e. BIT-ADDs/elementary operations) to multiply an n-bit number with an m-bit number ($n \geq m$).

Proof. There are two ways to do this. One is to "open up" the brackets. That is,

$$\begin{aligned} T(n,m) &\leq T(n+1,m-1)+(n+m) \\ &\leq T(n+2,m-2)+(n+m)+(n+m) \\ &\leq T(n+3,m-3)+(n+m)+(n+m)+(n+m) \\ &\vdots \\ &\leq T(n+m,0)+m(n+m) \end{aligned}$$

Now, we use that $T(n+m,0) = 0$, the base case which we knew how to handle. This proves the theorem. □

4 Division

DIVISION
Input: n-bit number x, m-bit number y expressed as bit-arrays $x[0:n-1], y[0:m-1]$.
Output: The quotient-remainder pair (q,r) such that $x = qy + r$ where $r < y$.
Size: The number of bits $n+m$.

Our final course of the day is integer division. We want to take input two numbers x, y, and return the quotient and remainder obtained when x is divided by y. That is, we want to find non-negative integers (q,r) such that $x = qy + r$ and $r < y$.

Once again, we define a recursive algorithm to do the same. First we identify the base cases. If $x < y$, then we know that the quotient is 0 and remainder is x. If $x = y$, then the quotient is 1 and remainder is 0. Now suppose $x > y$ and $x = 2k$ is even. Then if (q', r') is what we obtain from dividing k by y, that is, $k = q'y + r'$, then $x = 2q' \cdot y + 2r'$. We should return $(2q', 2r')$ except $2r'$ may be bigger than y. In which case, we should return $(2q'+1, 2r'-y)$. This suffices since $r' < y'$ and so $2r' < 2y$ and so $2r' - y < y$. If $x = 2k+1$ and (q', r') is obtained by dividing k by y, again we get $x = 2k+1 = 2q'y + 2r' + 1$. Once again we repeat the same as above.

```
1:  procedure DIVIDE(x, y):
2:      ▷ The two numbers are input as bit-arrays; x has n bits, y has m bits. n ≥ m.
3:      ▷ Returns (q, r) where x = qy + r and 0 ≤ r < y.
4:      if x < y then:
5:          return (0, x)
6:      if x = y then:
7:          return (1, 0)
8:      x' ← ⌊x/2⌋ ▷ Obtained by right shifts
9:      (q', r') ←DIVIDE(x', y)
10:     q ← 2q'; r ← 2r' ▷ Obtained by left shifts
11:     if x is odd then:
12:         r ← r + 1 ▷ Obtained by ADD(r, 1).
13:     if r ≥ y then:
14:         q ← q + 1 ▷ Obtained by ADD(q, 1).
15:         r ← r − y ▷ Obtained by ADD(r, −y).
16:     return (q, r).
```

✍

Exercise: *Prove that* DIVIDE *is correct.*

Let $T(n, m)$ be the time taken to divide an n-bit number by an m-bit number. Line 15 is the "time-taking" step of *subtraction*. Just like addition, this takes $\leq m$ time since both y and r have $\leq m$ bits. How about Line 12 and Line 14. Note that we are adding 1 to an even number (see Line 10) in both case. Thus the addition actually takes 1 time in both cases (do you see this?).

Now we can argue about $T(n, m)$ as in MULT.

$$\begin{aligned} T(n,m) &= 0 \quad \text{if } n < m \\ T(n,m) &\leq T(n-1,m) + (m+2) \end{aligned} \tag{2}$$

Theorem 3. DIVIDE takes $T(n, m) \leq (m+2) \cdot (n-m+1)$ time (i.e. BIT-ADDs/elementary operations) to divide an n-bit number by an m-bit number where $n \geq m$.

Proof. Once again, the easiest way to prove this is "opening up the brackets".

$$\begin{aligned} T(n,m) &\leq T(n-1,m) + (m+2) \\ &\leq T(n-2,m) + (m+2) + (m+2) \\ &\vdots \\ &\leq T(m-1,m) + (m+2) \cdot (n-m+1) \end{aligned}$$

The proof completes by noting $T(m-1, m) = 0$. □

Corollary 1. If $n = m + c$, that is, x has only c more bits than y, then DIVIDE takes $\leq Cn$ time for some constant C dependent on c.