Algorithms for Integer Operations[1]

In this lecture, we are going to look at some algorithms involving numbers. There are two objectives. One, to show that when numbers are "huge", manipulating them takes time. This is probably the only lecture where we will encounter this; usually, we would take arithmetic manipulation to be $O(1)$-time operations. Second, is to introduce the powerful notion of recursion which we will use throughout the course (and life).

1 Addition

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

Given two numbers, we add them by putting them one below the other, add the overlaying digits, and take 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? One could use the decimal notation, where the number "17" is used to indicate the concept of seventeen. Computers use the binary representation, and we will use that way of representing for this lecture's notes. This is simply a choice we are making; all we say below (except for one or two exceptions) will make sense even if you have the decimal representation in your head. Let us start by recalling the *binary representation*.

Remark: *An n-bit number a is represented by a bit-array*[a] $a[0 : n-1]$ *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.*

[a] As you can see, I have indexed the bit-arrays above starting from 0 instead of 1 like I will usually do in this course. Again, this is just a convenience, and doesn't change what we want to understand.

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

<u>ADDITION</u>
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. However, we will need one "subroutine". We will need the ability to add three bits.

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 19th Mar, 2022*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

More precisely, we will assume that we have access to a subroutine BIT-ADD which takes inputs three bits (b_1, b_2, b_3) and returns (c, s) where the number (c, s) interpreted as a binary number (that is $2c+s$) is indeed $b_1 + b_2 + b_3$. See the supplement for a precise definition. Since BIT-ADD deals with a constant number of bits in all, the time to call BIT-ADD is $O(1)$.

procedure ADD($a[0 : n-1], b[0 : n-1]$): ▷ *The two numbers are a and b*
  Initialize carry $\leftarrow 0$.
  Initialize $c[0 : n]$ to all zeros ▷ *$c[0 : n]$ will finally contain the sum*
  for $i = 0$ to $n - 1$ **do**:
    $(\text{carry}, c[i]) \leftarrow$ BIT-ADD$(a[i], b[i], \text{carry})$
  $c[n] \leftarrow$ carry
  return c

Theorem 1. The algorithm ADD has worst case running time $T_{\text{ADD}}(n) = O(n)$.

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. Once again, this is taught to us in grade-school, but why it works does need a proof. Of course, the proof is not difficult (you can find it in the supplementary notes), but the point to appreciate is that a proof is *needed*.

Note that the "naive" 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 humanity. If you doubt this, think of Roman numerals and how you would add them. Or perhaps consider multiplying them.

2 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. Most grade-school methods 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. It is also a great philosophy and applicable to most things in life.

Mantra: *Break the problem into smaller subproblems and let recursion take care of the smaller subproblems. Remember to solve the smallest subproblem.*

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. The proof of correctness can be found in the supplement.

1: **procedure** MULT(x, y):
2: ▷ *The two numbers are input as bit-arrays; x has n bits, y has m bits. $n \geq m$.*
3: **if** $y = 0$ **then**: ▷ *Base Case*
4: **return** 0 ▷ *An all zero bit-array*
5: $x' \leftarrow (2x)$; $y' \leftarrow \lfloor y/2 \rfloor$ ▷ *How much time does this take? See remark below.*
6: $z \leftarrow$MULT(x', y')
7: **if** y is even **then**:
8: **return** z
9: **else**:
10: **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?" Good question! Note that when x is expressed as a bit-array, $(2x)$ is just a left-shift. That is, we take all of x and add a zero at the end. Similarly, $\lfloor y/2 \rfloor$ is a right-shift. That is, we just drop the last bit. We may assume this takes $O(1)$ 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.*

Define $T(n, m)$ to be the maximum time 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 $y = 0$ we get that $T(n, 0) = O(1)$.

Now, let us figure out how much time each step takes.

- As remarked above, Line 5 takes $O(1)$ time.
- How much does Line 6 cost? This is important. It is a recursive call on the input x' and y'. What is the size of this input? We see that $x' = 2x$ has $n+1$ bits and $y' = \lfloor y/2 \rfloor$ has $m-1$ bits. By the *definition* of the *worst case running time*, we can therefore conclude that this step takes *at most* $T(n+1, m-1)$ time. Observe that the pessimistic definition of worst-case-running-time really helped here. In sum, this step costs $T(n+1, m-1)$.
- Now, if y was even, this would be the end of the algorithm.
- However, we are in the worst-case land, and y could be odd. In this case we need to add z and x. Now we use the fact that $z \leq x \cdot y$ has at most $(n+m)$ bits (see Fact 1 below). Therefore, the operation ADD(z, x) takes at most $O(n+m)$ time by Theorem 1.

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

Proof. Since x is an n-bit integer, we get $x \le 2^n - 1$. Similarly, $y \le 2^m - 1$. Therefore, $x \cdot y \le (2^n - 1) \cdot (2^m - 1) < 2^{n+m} - 1$, where we have used $2^n + 2^m - 1 > 1$. Therefore, xy has at most $(n + m)$ bits. ☐

Putting all these together, we see that the running time $T(n, m)$ for MULT can be captured by the following ***recurrence inequality***.

$$
\begin{aligned}
T(n, 0) &= O(1), \quad \forall n \\
T(n, m) &\le T(n + 1, m - 1) + O(n + m) \quad \forall n, m > 0
\end{aligned}
\tag{1}
$$

Remark: *Please note the* **abuse** *of notation above; we are adding a function and a set. Please go to the supplementary notes on this abuse to see what this abuse really means.*

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 at mosts $(n + m)$ 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*[2].

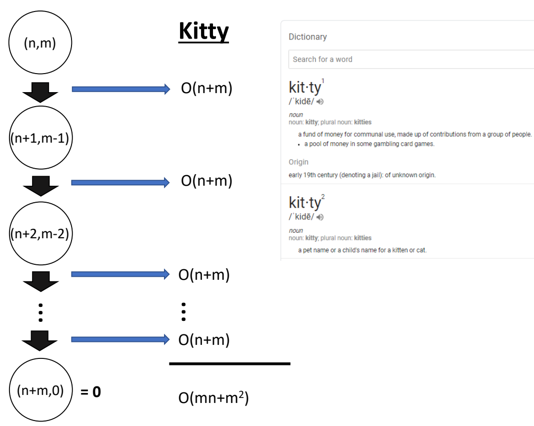

Figure 1: *The circles contain the various sizes. As we go down the sizes become smaller and smaller till we reach a small enough size for which we can figure out the running time directly. In particular, when the size becomes* 0*, the running time becomes* 0*. However, breaking the problem is not free. 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.

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

Theorem 2. MULT takes $T(n,m) = O(mn + m^2)$ time to multiply an n-bit number with an m-bit number ($n \geq m$).

Proof. See Figure 1 to see how to "open up" the brackets. Let us also be formal with the abuse of notation. Firstly, we can say (see the notes on Big-Oh) that there exists some constant $C > 0$ such that $T(n,m) \leq T(n+1, m-1) + C(n+m)$ for all $n, m \geq 1$. Thus, we get

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

Now, we use that $T(n+m, 0) = O(1)$. Together, we get $T(n,m) \leq Cm(n+m) = O(m^2 + mn)$. This proves the theorem. $\square$

3 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 $x > y$.

Case 1. $x = 2k$ is even. Then if (q', r') is what we obtain *recursively* when we divide the smaller number k by y, that is, $k = q'y + r'$, then $x = 2q' \cdot y + 2r'$. Therefore, 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 $2r' - y < y$. That is, $2r' - y$ is the correct remainder.

Case 2. $x = 2k + 1$ is odd. Again, suppose (q', r') is obtained *recursively* when we divide k by y. Then we get $x = 2k + 1 = 2q'y + 2r' + 1$. Now, since $r' < y$, that is, $r' \leq y - 1$, we get that $2r' \leq 2(y-1) = 2y' - 2$. Therefore, $2r' + 1 \leq 2y - 1 < 2y$. Which, in turn, implies $2r' - y < y$. That is, $2r' - y$ is the correct remainder.

1: **procedure** DIVIDE(x, y):
2: ▷ *The two numbers are input as bit-arrays; x has n bits, y has m bits. $n \geq m$.*
3: ▷ *Returns (q, r) where $x = qy + r$ and $0 \leq r < y$.*
4: **if** $x < y$ **then**:
5: **return** $(0, x)$
6: **if** $x = y$ **then**:
7: **return** $(1, 0)$
8: $x' \leftarrow \lfloor x/2 \rfloor$ ▷ *Obtained by right shifts*
9: $(q', r') \leftarrow$DIVIDE(x', y)
10: $q \leftarrow 2q'$; $r \leftarrow 2r'$ ▷ *Obtained by left shifts*
11: **if** x is odd **then**:
12: $r \leftarrow r + 1$ ▷ *Obtained by* ADD$(r, 1)$.
13: **if** $r \geq y$ **then**:
14: $q \leftarrow q + 1$ ▷ *Obtained by* ADD$(q, 1)$.
15: $r \leftarrow r - y$ ▷ *Subtraction is just addition with the "complement"*
16: **return** (q, r).

Once again, let us work to figure out the recurrence inequality for the running time. Let $T(n, m)$ be the time taken to divide an n-bit number by an m-bit number.

- Let's first understand the base cases. Line 5 and Line 7 take $O(1)$ time. Thus, we get $T(n, m) = O(1)$ if $n < m$. Note, we cannot say $n = m$ for x and y can both be m bits big and yet $x > y$.
- Line 8 and Line 10, as in the case of MULT, take $O(1)$ time as well.
- The recursive call in Line 9 takes at most $T(n-1, m)$ time. This is because x' has $n - 1$ bits, and the definition of worst-case runtime.
- Now consider Line 12 and Line 14. Note that in both cases we are adding 1 to an even number (see Line 10), that is, a number whose last bit is 0. Thus, one needs only one BIT-ADD to increment an even number by one. Thus, these steps cost $O(1)$ time.
- Line 15 is the "time-taking" step of *subtraction*. How does one subtract? As you can see in the supplement, subtraction is simply an addition[3] with "complementing", the time (or number of BIT-ADDs) is the same as to add. And thus, since both y and r have $\leq (m + 1)$ bits (note $r \leq 2y$), this step takes $O(m)$ time.

Therefore, we get the following recurrence for DIVIDE.

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

[3] If you have never seen this before, then I recommend going and reading this in the supplement.

Theorem 3. DIVIDE takes $T(n, m) = O(m \cdot (n - m + 1))$ 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, this can be solved by the kitty method as follows. Again, we may assume there is a large enough constant C such that $T(n, m) \leq T(n - 1, m) + Cm$ if $n \geq m$.

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

The proof completes by noting $T(m - 1, m) = O(1)$. $\square$

Corollary 1. If $n = m + c$, that is, x has only c more bits than y and c is some fixed constant (like 25), then DIVIDE takes $O(n)$ time.