CS 3510 — Algorithms, Fall 2022

Lecture 2: Multiplication & D&C

January 13, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

Homework 1 will be released either today (Thursday) or tomorrow (Friday) and due a week afterwards (either Friday or Saturday of next week).

1 Addition and Multiplication

This class will generally focus on these concepts from an algorithmic perspective. In particular, we generally analyze these topics using binary. We now wonder what the runtime (big-O) is for addition.

$$\begin{array}{cccccccc} & & 1 & 1 & 1 & 0 & 0 \\ + & & 1 & 0 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

The algorithm for addition is as follows:

1. For each column, we either add 2 or 3 (including carry bit) digits which is $O(1)$ (a small lookup table)

Hence, the runtime for this algorithm is $O(n)$ where n is the number of bits that are being added. We can't do better as this is a linear algorithm and at the very least, we have to traverse each bit regardless.

For multiplication of two binary numbers a and b, we perform an algorithm in two phases (note: n is the length of a or b)

				1	0	1	0	
			x	1	1	0	1	
				1	0	1	0	n numbers
+			0	0	0	0		
+		1	0	1	0			
+	1	0	1	0				
1	0	0	0	0	0	1	0	

1. Multiply digit by digit which will create n numbers with total length $\leq 2n$ and $n \cdot 2n = O(n^2)$

2. Add these numbers together such that we get $n - 1$ additions of numbers of length $\leq 2n$ creating a total runtime of $2n \cdot (n - 1) = O(n^2)$

1.1 Challenge Question / Discussion

What is the time complexity of multiplying m bit by n bit number?

We still have two phases in this multiplication:

1. Multiply digit by digit which will create n numbers that are of a different size! Now, each number has a maximum size of m meaning this step takes a total time of $O(nm)$

2. In a similar fashion, we now have to add together $\approx n - 1$ numbers of numbers that have length m with a runtime of $O(nm)$

Hence, the runtime for multiplying two numbers with length n and m bits gives a runtime of $O(nm)$. $\square$

1.2 Divide and Conquer

Can we make a faster algorithm to multiply numbers? We want to find an algorithm faster than $O(n^2)$ for two n-bit numbers.

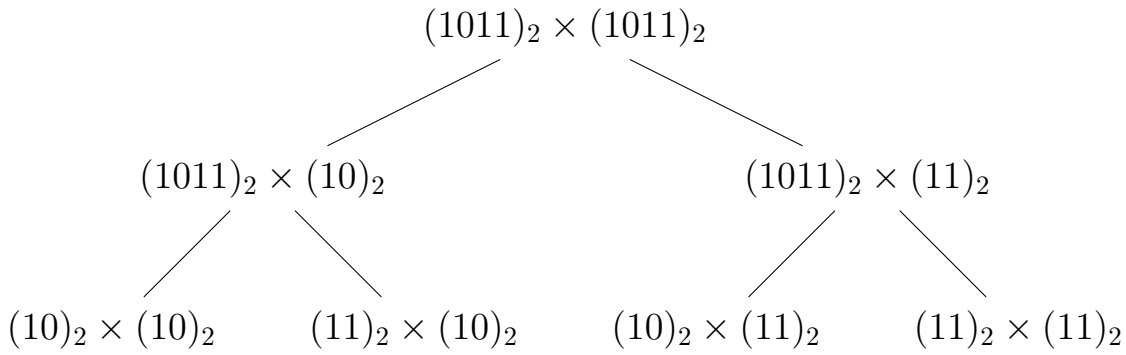
Idea: Split the multiplication into smaller subproblems and recursively solve. Finally, aggregate all the subsolutions to get a final solution.

$$\begin{array}{rcccc} & 1 & 0 & 1 & 1 \\ \text{x} & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

1. In the above example, split the top $(1011)_2$ into $(10)_2$ shifted by 2 and $(11)_2$ to create two seperate multiplication subproblems where a 4 bit number is multiplied by a 2 bit number.

2. In the 2 subproblems, split the bottom $(1011)_2$ into $(10)_2$ shifted by 2 and $(11)_2$ to create a total of four seperate multiplication subproblems.

3. Continue this splitting until we only have to multiply 1 bit numbers together (base case, $1 \times 0 = 0 \times 1 = 0 \times 0 = 0$ and $1 \times 1 = 1$).



For brevity, the rest of the tree is not drawn. Further, the above tree is more verbose than reality. Splitting from $1 \to 4$ multiplications happens in one step by splitting both a and b into halves on each step. The next two steps in the tree would split into a total of 16 multiplications each of which contain 1 bit to multiply.

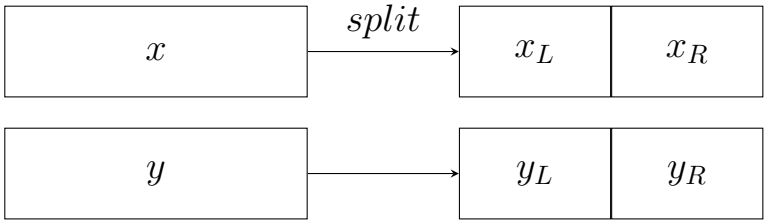
In order to reconstruct the final solution, we add together children on the same level and shift the left sibling (as the left sibling solves for the left half of the final output). This algorithm is formalized below.

1.3 Formalized D&C Multiplication

Problem: Multiply two numbers x and y together

Algorithm:

1. Split x into x_L and x_R and y into y_L and y_R being the left and right halves



2. With these splits, we can still represent x and y using $x = 2^{n/2}x_L + x_R$ (same applies for y)

3. This form gives us $xy = 2^n(x_Ly_L) + 2^{n/2}(x_Ry_L + x_Ly_R) + x_Ry_R$

In a sense, we can compute $x \times y$ by using the halves of the numbers. In the above equation, how do we compute x_Ly_L? This is where recursion and divide and conquer comes in! x_Ly_L is computed with the same equation as above with *its own halves*. Here is pseudocode that makes the recursion more obvious.

```
1  Function Mult(x, y):
       /* split x, y into left and right halves (bitwise)                */
2      x_L ← x[0 : n/2]
3      x_R ← x[n/2 : n]
4      y_L ← y[0 : n/2]
5      y_R ← y[n/2 : n]
       /* recursion to multiply smaller numbers, all of size n / 2       */
6      A ← Mult(x_L, y_L)
7      B ← Mult(x_L, y_R)
8      C ← Mult(x_R, y_L)
9      D ← Mult(x_R, y_R)
       /* put everything together, see equation from above               */
10     return 2^n A + 2^{n/2}(B + C) + D ;      // combining takes O(n) time
```

Definition 1.1. $T(n)$ is *number* of steps an algorithm takes for input size n

In the above algorithm, it takes $O(n)$ time to get halves and combine the answer for the final return. On top of that, we make 4 total calls to multiply numbers of size $n/2$, hence $T(n)$ is:

$$T(n) = O(n) + 4T\left(\frac{n}{2}\right)$$

As of now, we can't solve this for a runtime – but it is $O(n^2)$. You will be able to solve this later in the course. This runtime is the same as the naive grade school multiplication, *but...* Notice the following:

$$(x_L + x_R)(y_L + y_R) = x_L y_L + x_R y_R + x_L y_R + x_R y_L$$
$$(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R = x_L y_R + x_R y_L$$

From above, recall $A = x_L y_L$, $D = x_R y_R$ so the above is equivelant to

$$\underbrace{(x_L + x_R)(y_L + y_R)}_{\text{single multiplication}} - A - B = \underbrace{B + C}_{\text{two multiplications}}$$

Hence, define a new value $Q = (x_L + x_R)(y_L + y_R)$ which is still a multiplication of smaller numbers as $x_L + x_R \neq 2^{n/2}x_L + x_R$. This gives a new algorithm:

```
1 Function Mult(x, y):
2     x_L ← x[0 : n/2]
3     x_R ← x[n/2 : n]
4     y_L ← y[0 : n/2]
5     y_R ← y[n/2 : n]
6     A ← Mult(x_L, y_L)
7     B ← Mult(x_L, y_R)
      // new multiplication, but no more B and C
8     Q ← Mult(x_L + x_R, y_L + y_R)
9     return 2^n A + 2^{n/2}(Q − A − D) + D ;        // combining still O(n)
```

In this algorithm, we still take $O(n)$ time to perform additions and subtractions but only make 3 recursive calls, giving a recurrence of:

$$T(n) = O(n) + 3T\left(\frac{n}{2}\right)$$

This is $O\left(n^{\log_2 3}\right) \approx O\left(n^{1.59}\right)$ as you will be able to solve next week.