COMP 285 (NC A&T, Spr '22) Lecture 1

Adapted From Virginia Williams' lecture notes. Additional credits: J. Su, W. Yang, Gregory Valiant, Mary Wootters, Aviad Rubinstein, Sami Alsheikh.

Karatsuba's Algorithm & Pseudocode Practice

1 Karatsuba Integer Multiplication Cont.

1.1 Divide and Conquer

The "Divide and Conquer" algorithm design paradigm is a very useful and widely applicable technique. We will see a variety of problems to which it can be fruitfully applied. The high-level idea is just to split a given problem up into smaller pieces, and the solve the smaller pieces, often recursively.

How can we apply Divide and Conquer to integer multiplication? Lets try splitting up the numbers. For example, if we were multiplying 1234×5678, we could express this as $((12 \cdot 100) + 34) \cdot ((56 \cdot 100) + 78)$. In general, if we are multiplying two n-digit numbers x and y, we can write $x = 10^{n/2} \cdot a + b$ and $y = 10^{n/2} \cdot c + d$. So

$$x \cdot y = (10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^n ac + 10^{n/2}(ad + bc) + bd.$$

Now we can split this problem into four subproblems, where each subproblem is similar to the original problem, but with half the digits. This gives rise to a recursive algorithm.

Interestingly enough, **this algorithm isn't actually better!** Intuitively this is because if we expand the recursion, we still have to multiply every pair of digits, just like we did before. But in order to prove this formally, we need to formally define the runtime of an algorithm, and prove that these algorithms are not very different in runtime.

1.2 Recurrence Relation

We can analyze the runtime of the algorithm as follows.

Let $T(n)$ be the runtime of the algorithm, given an input of size n (two n-digit numbers). Because we are breaking up the problem into four subproblems with half the digits, plus some addition with linear cost, we have the equation $T(n) = 4T(n/2) + O(n)$. (Don't worry if you haven't seen big-O notation before; we'll go over this in detail in the coming lectures.)

Although **in general you should pay attention to the $O(n)$ term**, today we will just ignore it because the term doesn't matter in this case.

By repeatedly breaking up the problem into subproblems, we find that

$$T(n) = 4T(n/2) = 16T(n/4) = \cdots = 2^{2t}T\left(\frac{n}{2^t}\right) = n^2T(1)$$

Since $T(1)$ is the time it takes to multiply two digits, we see that the above suggestion does not reduce the number of 1-digit operations.

Note: In the lecture slides, we'll consider a slightly different argument, which analyzes a recursion tree. It's a good exercise to understand both arguments! Again, we'll discuss both techniques more in coming lectures.

1.3 Divide and conquer (take 2)

Karatsuba found a better algorithm (in 1960, published in 1962) by noticing that we only need the sum of ad and bc, not their actual values. So he improved the algorithm by computing ac and bd as before, and computing $(a+b)\cdot(c+d)$. It turns out that if $t = (a+b)\cdot(c+d)$, then $ad + bc = t - ac - bd$. Now instead of solving four subproblems, we only need to solve three! This idea goes back to Gauss, who found a similar efficient way to multiply two complex numbers.

Sure, we need to do more additions, but again it turns out that additions are pretty cheap. To do a quick-and-dirty analysis of the number of operations required by Karatsuba multiplication, first assume that $n = 2^s$ for some integer s. (Note that we can always add 0's to the front of a number until the length is a power of two, so this assumption holds without loss of generality.) Letting $T(n)$ denote the number of multiplications of pairs of 1-digit numbers required to compute the product of two n-digit numbers, Karatsuba's algorithm gives $T(n) = 3T(n/2)$, since we've divided the problem into three recursive calls to multiplication of length $n/2$ numbers. [Note, we are cheating a bit here, since $(a+b)$ and $(c+d)$ might actually be $n/2+1$ digit numbers, but lets ignore this for now $\cdots$] Hence we have the following:

$$T(n) = 3T(n/2) = 3^2T(n/4) = \cdots = 3^sT(n/2^s)$$

Since we assumed that $n = 2^s$, we have that $T(n/2^s) = T(1) = 1$ since multiplying two 1-digit numbers counts as 1 basic operation. Hence $T(n) = 3s$, where $n = 2^s$. Solving for s yields $s = \log_2 n$, and hence we get

$$T(n) = 3^{\log_2 n} = 2^{(\log_2 3)(\log_2 n)} = n^{\log_w 3} \leq n^{1.6}$$

We were pretty sloppy with the above argumentation in a lot of ways. However, we'll see a much more principled way of analyzing the runtime of recursive algorithms in the coming classes, so we won't sweat about it too much now. The point is that (even if you do it correctly) the running time of this algorithm scales like $n^{1.6}$. Thus is **much** better than the n^2 algorithm that we learned in grade school!

1.4 Can we do better

Progress on efficient algorithms for multiplication of n-digit numbers compared continued beyond Karatsuba's algorithm. Although you don't need to know these algorithms for CS 161, it is interesting to review the history of progress on this problem. Toom and Cook (1963) developed an algorithm that ran in time $O(n^{1.465})$ by showing how a single n-sized problem could be broken up into five $n/3$-sized problems. Schönage and Strassen (1971) developed an algorithm that runs in time $O(n \log(n) \log\log(n))$. More than 35 years later, Fürer (2007) developed an algorithm that ran in time $n \log(n) 2 \log^*(n)$. In case you are wondering what that weird function $\log^*(n)$ (read "log star") is, it is the number of times you have to apply the logarithm function log() iteratively to n in order to get down to something ≤ 1. For all values of n less than the estimated number of atoms in the universe, the value of $\log^*(n)$ (with base 2) is less than 5. So $\log^*(n)$ is a really really really slowly growing function of n. Finally, Harvey and van der Hoeven (2019) gave an algorithm that runs in time $O(n \log n)$. This is conjectured to be optimal. It is quite amazing that the seemingly simple (and old) question of multiplying two numbers has proved to be so mysterious and has seen new research advances as recently as 2019. This is what makes the study of algorithms so exciting!

2 Practice with Pseudo-code

Let's now shift gears a bit from algorithms, and practice some C++. It's important for use to be able to take our ideas and convert them into code. Generally speaking, the following approach to solving problems is critical.

- Read the problem carefully and make sure you understand what you're trying to solve. This might require that you run through a few examples, by hand, just like we did in lecture.
- Try to figure out a *correct* solution to your problem. Usually, this comes down to figuring out how you would do it by hand, and then trying to convert this into a step-by-step process: an algorithm.
- Write down your proposed solution in pseudo-code. We will go through a few examples, but generally speaking, pseudo-code is where you can work through edge-cases, figure out what steps you're missing, and really just jot down everything you need.
- Convert your pseudo-code to code. This means you can convert your code to Python, C++, etc. Generally, you'll be converting your code to one of a few languages, but it could be any!

2.1 Finding the minimum element in a collection of elements

Let's start with a simple problem. We want to find the smallest element in an arbitrary collection. This is what the pseudo-code might look like for this problem:

```
algorithm findMinOfVector
```

```
  input: a vector of integers nums
  output: smallest integer within nums
  minSoFar = infinity
  for each element num in nums
    minSoFar = min(minSoFar, num)
  return minSoFar
```

This is a very simple algorithm, so we're pretty sure it's right. We'll jump straight to the last step above then, were we can to convert this function. This is what the corresponding pseudo-code looks like:

```cpp
#include <vector>
#include <limits>

int findMinOfVector(const std::vector<int>& nums) {
 int minSoFar = std::numeric_limits<int>::max();
 for(int i = 0; i < nums.size(); i++) {
   minSoFar = std::min(minSoFar, nums[i]);
 }
 return minSoFar;
}
```

You should be comfortable going back-and-forth between the options above. A few things to keep in-mind in the above code:

- auto
- range-based for-loops
- size_t (though not applicable with range-based for loop)

Also, check-out the course website C++ resources for more.

2.2 Practice Spotting Bugs in Pseudo-code

Another reason why pseudo-code is useful, is that it can be a low-commitment way to find bugs in your code. Before you actually get around to writing your code (which can take a lot of time, and requires you know the syntax for your language), it's very useful to read your pseudo-code and see if there are any bugs.

Let's try it out.

- What does the algorithm below say it does?
- What issues prevent the algorithm from being correct?

```
algorithm findSecondSmallest
  input: list of integers nums, where size of nums > 1
  output: the second smallest element in nums
  firstSmallest = -infinity
  secondSmallest = -infinity
  for each element num in nums
    if num < firstSmallest
      firstSmallest = num
      secondSmallest = firstSmallest
    else if num < secondSmallest
      secondSmallest = num
  return num
```

The first issue is with how we're setting the initial values for `firstSmallest` and `secondSmallest`. They should be set to a value which is trivially **larger** than all other values (eg, `infinity`).

The second issue has to do with the order in which we're updating the `firstSmallest` and `secondSmallest` when we find an element `num` that's smaller. We need to first set our `secondSmallest` variable to our `firstSmallest` (since we found a smaller number, our previous smallest is now the second smallest), and **then** set our `firstSmallest` to the number we just found.

The last issue with this code with the our return statement. First of all, `num` is likely our of scope at this point, and is not that we want to return. What we want to return is `secondSmallest`.

With the fixes discussed above, our pseudocode should look like this:

```
algorithm findSecondSmallest
  input: list of integers nums, where size of nums > 1
  output: the second smallest element in nums
  firstSmallest = infinity
  secondSmallest = infinity
  for each element num in nums
    if num < firstSmallest
      secondSmallest = firstSmallest
      firstSmallest = num
    else if num < secondSmallest
      secondSmallest = num
  return secondSmallest
```

2.3 Planning an Algorithm with Pseudocode

The next useful thing about pseudocode is that it helps us plan algorithms. We can plan and write out our pseudocode without ever having to think about syntax, or issues that we'll likely run into when actually trying to write the code. Let's work through an example.

Problem: Kanye West wants to make sure he's the only person named Kanye West in the world fast. Write a function that takes in both a vector of all names in the world sorted in alphabetical order along with a target name (e.g. "Kanye West"), and returns whether or not the target name appears no more than once in the vector.

- What are inputs and outputs?
- How would you approach this at a high-level?

You might write something like this:

```
algorithm findIfNameIsUnique
  input: sorted vector allNames, targetName
  output: true if targetName appears once, false otherwise

for each element name in allNames
  if name == targetName
    if next name == targetName
            return false
    else
            return true
return true
```

Once you have the initial version of the pseudo-code written, you might want to refine it further, by writing a bit more. Eventually, it looks almost like real code (see below), at which point, you can probably get started on actually writing out the code.

```
algorithm findIfNameIsUnique
  input: sorted vector allNames, targetName
  output: true if targetName appears once, false otherwise

start = 0
end = allNames.size() - 1
while start <= end
  midIndex = (start + end) / 2
  if allNames[midIndex] == targetName
    if neighboring name(s) are not targetName
return true
    else
return false
  else if allNames[midIndex] < targetName
    start = midIndex + 1
  else
    end = midIndex - 1
return true  // does not appear
```