CS 161, Lecture 1 Introduction
Scribes: Virginia V. Williams, Jessica Su (2015), Wilbur Yang **Date:** September 26, 2016

Optional reading for this lecture: Kleinberg-Tardos: p. 278-280 Sec 5.5, 6.9

1 Course information

- The class website is `http://cs161.stanford.edu`. Additionally, there is a Piazza site which you should sign up for — the link is on the class website. Final grades for the class will be based on
  - Homework (35%)
  - Midterm (25%): Monday, October 31, in class, 1:30 pm - 2:50 pm
  - Final (40%): Wednesday, December 14, 3:30 pm - 6:30 pm (Location TBA)
- We will use two textbooks for this course:
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms*, 3rd Edition, MIT Press
    This is the main textbook that we use. It is available online through the Stanford library.
  - Jon Kleinberg, Éva Tardos, *Algorithm Design*, Pearson/Addison-Wesley
    This is another textbook that we will usually occasionally.

2 Why are you here?

Many of you are here because the class is required. But why is it required?

1. **Algorithms is fundamental to CS:** Algorithms is the bread and butter of computer science. Wherever computer science reaches, an algorithm is there. Some CS classes that use algorithms include CS 140 (operating systems), CS 144 (networking), CS 143 (compilers), CS 229 (machine learning), CS 255 (cryptography), and CS 262 (computational biology). We'll discuss applications to all these subjects in this class.

   Algorithms is also important to other subjects, like biology and economics. These days we need to answer a lot of non-CS questions computationally, and this is where algorithms comes in.

2. **Algorithms is useful:** Algorithms is useful after graduation, and even before graduation. Many of you will do work that requires computer science, which will involve algorithms. Most programming interviews ask algorithms questions, because when you work, you will need techniques and tools from this class to solve the problems at hand.

   According to alumni surveys, CS 161 is one of the top two most useful CS classes at Stanford.

3. **Algorithms is fun!** It requires a lot of creativity and mathematical precision. It's fun for us, and hopefully it will be fun for you too.

3 Three applications of algorithms

Studying algorithms is important to solving problems in many different domains. Often, you might encounter a problem whose solution is not trivial. However, by studying classical algorithms, you will be able to identify the right algorithms to use to solve the problem. Additionally, by understanding algorithm design principles, you will be able to build upon classical algorithms or create novel algorithms to solve new problems.

Today, we will discuss at a high level three applications that are all important in their respective fields: Internet routing, genome sequencing, and multiplying large numbers.

3.1 Internet routing

Suppose we are writing an email at Stanford and want to send it to a friend at MIT. The email has to get there somehow. So it is sent in packets, and each packet is routed from machine to machine until it reaches its destination. Which machines should we route the packets through so that they reach their destination most efficiently? There are standard protocols that do this on the Internet today. But where did these protocols come from?

3.1.1 How to model the problem

In order to even approach this problem, we need to define our model of the Internet. A useful way to model the Internet is using graphs, because then we can apply preexisting techniques for analyzing graphs. We will see an example of this shortly in the form of the Internet routing problem.

There are many ways to think about the Internet in terms of a graph. Some examples are:

- *Internet routing graph.* The vertices will be routers and machines (endpoints), and we add edges that represent connections between them (either physical or wireless).

  We may also assign weights to the edges with more information about the network. For example, on each edge, we may add a weight representing the capacity (bandwidth) of the connection, allowing us to find routing paths that do not overload any connection.

- *Webgraph.* The vertices will be web pages, and we will add a directed edge for every hyperlink from one page to another. (This graph is used for the PageRank algorithm.)

- *Social network graph.* The vertices are people or profiles (e.g., on Facebook), and we add an undirected edge between two people who are friends.

  Using this graph, we can answer questions about social networks: we may infer how similar two entities are, or perform analyses similar to six degrees of separation on the social network.

Of the three graph models of the Internet that we've talked about, the one that seems most suitable to solving this problem is the Internet routing graph. Again, the vertices of this graph are routers or machines. There is a directed edge from one machine to another if the first machine can send a message directly to the second machine, either via a physical or wireless link.

Now we no longer need to think in terms of machines; we can abstract the problem away to just vertices on graphs. There are multiple routes you can take from your source vertex to your destination vertex. Question: which is the best route?

We can define the "best route" in several ways. Maybe the best route is the one with maximum bandwidth, or the shortest route. For now let's assume we want to find the route with the fewest number of hops (the shortest route).

3.1.2 How to solve it

The most well-known ways to solve a shortest route problem is to use breadth first search if the graph is unweighted, or Dijkstra's algorithm if the graph is weighted. (Don't worry if you have not heard of these algorithms. We'll cover them in detail later in class.)

These algorithms start at the source node, and look at their neighbors to see which nodes are distance 1 from the source node. Then from each neighbor, you look at their neighbors to find the machines you can reach within two hops and so on. If your destination is far away, these searches may need to traverse the whole graph.

What does this mean for our application? It means that in order to route the packet, the starting machine would need to know the whole Internet graph. In addition to being impractical by consuming large amounts of memory, it also, intuitively, does not make sense. Why should we need information about routes in Beijing to route from Stanford to MIT?

So Dijkstra's and BFS do not seem to be directly applicable to routing on the Internet. However, very early on, in 1956 and 1958, even before Dijkstra's algorithm, a completely different shortest path algorithm was designed independently by Bellman and Ford. Now called the Bellman-Ford algorithm, this algorithm only needs *local* information to compute the best route. That is, the starting node only needs to know information about the nodes that it can directly communicate with. The Bellman-Ford algorithm is not as efficient overall as Dijkstra's algorithm. However, its locality property allows for the design of "distributed" routing protocols, i.e. ones for which our routers only need to know local information about the network and can make quick decisions about where to route the packet next.

The Bellman-Ford algorithm was designed before the concept of the Internet was even conceived — ARPANET didn't exist until 1963. However, the development of this algorithm helped shape the Internet as it is today.

The algorithms we develop today will show up in the technologies of tomorrow!

It is worth mentioning that the Bellman-Ford algorithm is one of many applications of a very clever algorithmic design concept called **dynamic programming**, which we will cover in future lectures. Even when the exact algorithms are not used, the basic design principles are used, and this is the power of algorithmic research.

3.2 Gene sequencing (sequence alignment problem)

In the sequence alignment problem, you are given two sequences of numbers or letters, and you want to figure out how similar they are. For example, in biology, you might consider sequences of DNA base pairs. (This problem shows up in lectures 2 and 3 of CS262, if you're interested.)

This is important because we might want to figure out which genes cause certain traits, and one way to do that is by seeing which genes cause certain traits in animals, and comparing those genes to genes in the human genome. If the DNA sequences are similar, then maybe the genes do similar things. Biologists have been able to reconstruct the evolutionary tree from this data, predicting two species' most recent common ancestor from the similarity of their DNA. Sequence alignment is also used in gene sequencing, where it is efficient to break up DNA into multiple segments in many different ways, scan them, and then align them back together.

DNA sequences can be represented by strings of the letters A, C, G, and T. For example, CGAATG or GATTG. These sequences are not identical, but maybe we would want to find out how similar they are (you can have mutations in genes that still preserve gene function, so sequences that are similar but not identical may still map to the same trait).

The scenario you can think about is as follows: biologists worked on the mouse genome and discovered that CGAATG is a gene associated with disease X. Now, in the human genome, they discovered GATTG and want to figure out how similar this gene is to CGAATG. If the gene is similar, then maybe they can conclude that it is also associated with disease X but now in humans.

Sequence alignment algorithms are also useful in spell checker programs, where you'd want to find the word in the dictionary that's most similar to what the user typed. (E.g. what's the closest real word to the misspelled word "diffrense"?) They're also used in the Unix command "diff" which finds differences between two given texts.

To solve the sequence alignment problem, we first need to define a distance metric between two strings, i.e., a measure for how similar two strings are. Perhaps you can measure the distance between two DNA sequences by seeing how well they form Watson-Crick base pairs (A matches with T and C matches with G). However, for this problem we'll be defining the distance between string A and string B by asking, **"what is the minimum number of changes you have to make to convert string A into string B?"** In biology, this seems reasonable as it could represent the number of mutations.

To transform a string, you can insert symbols, delete symbols, or replace a symbol with another one. We can model the first two operations by adding "gaps" in the first or second string. A gap in the first string corresponds to a symbol insertion in that position, and a gap in the second corresponds to a symbol deletion in that position. Now we align the two sequences (with the gaps) one on top of the other so that every letter

is aligned with one from the other string. (The order is the same.) Some aligned letters can differ, in which case we say that there's a mismatch. This corresponds to mutations.

Here's an example of a sequence alignment:

```
CGAATG
_GATTG
```

Now that we have said what an alignment is, we still want to figure out what its cost is. Then we can define the similarity between two strings as the minimum cost of any alignment (i.e. the best alignment.)

3.2.1 How to model this

Needleman-Wunsch definition ('70): In this definition of alignment cost each (insertion or deletion) gap costs δ, which is a parameter that depends on your application. Each change in letter from a to b costs $\alpha(a, b)$, where if $a = b$, $\alpha(a, b) = 0$. (This takes into account the fact that it may be more difficult to change certain base pairs to other base pairs.)

To find the similarity between sequences, we should find the alignment of minimum total cost.

In our example above, suppose $\delta = 1$ and $\alpha(a, b) = 2$ for all pairs of letters (a, b). The first base pair in the alignment get assigned a cost of 1. The A-T base pair gets assigned a cost of 2, so the total cost is 3.

3.2.2 How to compute the best alignment

Needleman and Wunsch were both biologists. However, in order to find a practical definition of sequence similarity, the definition had to be computable.

This brings us to our first question: Is the number of alignments finite? Although it is true that there is an infinite number of possible alignments, we notice that we may consider only a finite portion of them: if the lengths of the strings are n and m, in the best alignment, we do not have to add more than $n + m$ gaps. This allows us to prune out many alignments that can't possibly be the best alignment.

The sequence alignment problem can indeed be solved. There is an easy to describe algorithm, **Brute Force**: there is a finite number of possible alignments to consider, so we can just go through all of them and pick the one with the lowest cost.

The question is: How many alignments would we have to go through?

Let's get a ballpark figure. Assume the first string has no mismatches, no insertions, and only deletions (this will produce a lower bound on the number of possible alignments; if we allow other changes we will have even more alignments). For each letter in the first string, we can either delete the letter, or keep it the same. So the number of possible choices is $2 \times 2 \times 2 \times \cdots \times 2$, where the number of 2's equals the length of the first string. That is, if the length of the strings is n, we get at least 2^n alignments.

How big is n in practice? For the biology application it can be on the order of millions. For the diff application, it can be smaller. Suppose $n = 300$. But 2^{300} is greater than the number of atoms in the universe! There is no way biologists (or anyone) can use the brute force approach.

In reality people would again use **dynamic programming**. By learning the principles behind dynamic programming, you will be able to derive the algorithm for sequence alignment by yourself.

3.3 Multiplying numbers

3.3.1 The problem

Suppose you have two large numbers, and you want to multiply them. We all know how to multiply numbers, but when the numbers get large, the grade-school algorithm becomes slow. Question: can we do better?

The quality of an algorithm can be measured by how long it takes. In the grade-school multiplication algorithm, we need to multiply each digit in the first number by each digit in the second number. So the number of one-digit multiplications we need to perform is $n \cdot n$, if we assume that both numbers have n

digits. The total amount of work you need to do to multiply these numbers also includes some additions, so it is more than n^2 1-digit operations.

To improve this algorithm, we could consider storing the products of all pairs of t-digit numbers. This does result in performance gains; however, it also leads to exponential storage costs.

3.3.2 Divide and conquer

Maybe we could do better by splitting up the numbers. For example, if we were multiplying 1234×1111, we could express this as $((12 \cdot 100) + 34) \cdot ((11 \cdot 100) + 11)$. 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.

3.3.3 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

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

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.

3.3.4 Divide and conquer (take 2)

Karatsuba found a better algorithm 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 as we will see in a few weeks, we can formally show that the additions are cheap and this algorithm is better.