CS 161, Lecture 1 Introduction
Scribes: Virginia V. Williams and Jessica Su (2015) **Date:** March 25, 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`. There is also a Piazza site which you should sign up for- the link is on the website. The class will be based on

- Homework (50% of grade):
  - **6 homeworks** (+ Homework 0)
  - All homework comes out on Fridays and are due on the following **Friday, at 12 pm.**
  - **All homework should be uploaded to Gradescope- the sign-up link is on the website.** This does not mean you have to type your homework, but you do have to upload it to the Internet.
  - You can collaborate and use the Internet for your assignments, but when you write down your solutions, you need to **write down who and where you got help from.** Write down who you collaborated with and which sources you used.
  - **No late homework will be accepted**, but we will drop the lowest grade. You should still attempt all the assignments.
  - Homework 0 comes out this Friday, and should be easy for people who have the background required for this class.
- Midterm (20% of grade): May 3, most likely at 7pm
- Final (30% of grade): Saturday, June 4, at 7pm

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 at the core 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 Two applications of algorithms

We will cover two applications today: Internet routing and computational biology.

3.1 Internet routing

Suppose we are writing an email and want to send it to a friend. The email has to get there somehow. So it comes in a packet, and it is routed from machine to machine until it reaches its destination. Which machines should we route the packets through so 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 simplify our model. You can think of a computer network as a graph, where the vertices (or "nodes") are machines on the Internet. There will be 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 link or via wireless.

Now we no longer need to think in terms of machines, 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 this 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, or Dijkstra's algorithm if the graph were 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. In other words, the algorithms sort all vertices in the graph by their distance from the source. 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. This not only consumes large amounts of memory, but also, intuitively, does not makes 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 paths 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– the 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!

Even when the exact algorithms are not used, the basic design principles are used, and this is the power of algorithmic research.

In this application we talked about one particular Internet graph. There are other Internet-associated graphs. For instance, there is a graph associated with the Web: the vertices are webpages and the edges are hyperlinks. There are social network graphs, e.g. for Facebook and Twitter. There the graphs are the users and the edges correspond to friendships. There are many algorithms that come up in trying to solve problems on such graphs, e.g. PageRank.

3.2 Sequence alignment (computational biology and elsewhere)

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 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, maybe the genes do similar things.

DNA sequences can be represented by strings of the letters A, C, G, and T. For example, GGGCAAT or AGGCGAA. 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 GGGCAAT is a gene associated with disease X. Now, in the human genome, they discovered AGGCGAA and want to figure out how similar this gene is to GGGCAAT. 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 can ask **"what is the minimum number of changes you have to make to convert string A into string B?"** In biology, this could be the number of mutations.

To change the string, you can add or delete symbols or you can change a symbol into another one. We can model by adding "gaps" possibly in both the first and 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 positions. Now we put the two sequences (with the possible gaps) one on top of the other so 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:

```
_GGGCAAT
AGGCGAA_
```

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: 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 and last base pair in the alignment get assigned a cost of 1. The G-C and C-G base pairs each get assigned a cost of 2 so the total cost is 6.

3.2.2 How to compute the best alignment?

Needleman and Wunsch were both biologists. We want to point out that they would not have defined sequence similarity this way if it were not computable!

The problem can indeed be solved. There is an easy to describe algorithm, **Brute Force**: There are a finite number of possible alignments, 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 use an algorithm design concept called **dynamic programming**, which is a very clever concept that we will learn about in this class. By the end of this class, designing an algorithm for sequence alignment would be a routine task for you!

3.3 Multiplying numbers

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

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.

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 doing four subproblems, we only need to do three!

Sure, we need to do more additions. But as we will see in a few weeks, we can formally show that additions are cheap and this algorithm is better.