Lecture 1: Introduction

Three questions we already ask about an algorithm:

1. Is the algorithm correct?
2. How fast does the algorithm run?
3. Can we improve to have a better algorithm?

Integer addition problem

Given two integers x and y, we want to compute z = x + y. It may seem this is not needed since addition seems doable in one step. This is not true, especially for long integers. We now analyze a grade-school algorithm. Suppose both x and y are binary with \( n \) bits each. We let \( a[i] \) be the i-th bit of \( a \).

1: \( c \leftarrow 0 \)
2: \( \text{for } i = 1 \text{ to } n \text{ do} \)
3: \( t \leftarrow c + a[i] + b[i] \)
4: \( z[i] \leftarrow 1 \) if \( t \) is odd, and 0 otherwise.
5: \( c \leftarrow 1 \) if \( t \geq 2 \) and 0 otherwise.
6: \( \text{end for} \)
7: if \( c \geq 1 \), \( z[n+1] = 1 \).

To measure how fast the algorithm runs, we use the number of single bit addition operations. Note that there are \( n \) iterations of the loop, and in each loop we perform two single bit additions. Thus, the number of single bit additions is at most \( 2n \).

Integer multiplication problem

We are given two numbers \( x \) and \( y \), each with \( n \) bits. We want to compute \( z = x \times y \). We will compare two algorithms, using the number of single-bit additions as performance criteria.

1: \( z \leftarrow 0 \)
2: \( \text{for } i = 1 \text{ to } y \text{ do} \)
3: \( z \leftarrow z + x \).
4: \( \text{end for} \)

1: \( z \leftarrow 0 \)
2: \( \text{for } i = 1 \text{ to } n \text{ do} \)
3: \( t \leftarrow x \) if \( y[i] = 1 \), and 0 otherwise.
4: \( \text{Left-shift } t \text{ by } i - 1 \text{ bit.} \)
5: \( z \leftarrow z + t \).
6: \( \text{end for} \)

Let us first analyze the first algorithm, which looks simpler. There are \( y \) iterations. In each iteration, we perform one addition of two integers which are of at least length \( n \). Since \( y \) has \( n \) bits, so \( y \geq 2^{n-1} \). Thus, we will need at least \( 2^{n-1} \times 2n = n2^n \) single bit additions.

Now we turn to the second algorithm. We have \( n \) iterations of the loop. In each iteration, we are performing no more than \( 2n \) single bit summations for adding two integers. Note: since we left shift \( t \), when properly implemented, we will still only add no longer than \( n \) bits in the addition. Thus, the algorithm will
use \( n \times 2n = 2n^2 \) single bit additions. If you try a few values for \( n \) (say \( n=50 \)), you will be convinced that the second algorithm is much more efficient.

**Lecture 2: Algorithm Analysis**

A main topic discussed here is the analysis of the insertion sort algorithm, which is well documented in the textbook.

We then briefly discussed the concept the big-O notation. Formally, for two functions \( f(n) \) and \( g(n) \), we say \( f(n) = O(g(n)) \) if \( f(n) \leq Cg(n) \) for some constant \( C \) when \( n \) is large enough (i.e. \( n \geq n_0 \) for some fixed \( n_0 \)). For insertion sort, the run time \( T(n) = a_2x^2 + a_1x + a_0 \). We claim \( T(n) = O(n^2) \). This is because we can pick \( C = a_2 + a_1 + a_0 \). The main motivation is, the highest order term, \( n^2 \), dominates the run time when \( n \) is large. Thus, we want to focus on this term. We will come back to this subject in the next subject.

Now, we discussed two problems as example problems we will study later in the semester. The first problem is to find shortest path between two nodes \( x \) and \( y \) in a graph \( G \) with \( n \) nodes. We will present much more efficient algorithm later, but for now we start with some simple brute force algorithm. This algorithm simply tries all possible paths of the graph. To make the enumeration finite, we impose the constraint that there is no duplicate nodes in any single path. You should pause and think why adding this restriction does not miss any shortest path. Now, to enumerate paths, we simply try all possible permutations of nodes of \( G \) (which starts with \( x \) and ends with \( y \)). One should note that when \( G \) is fully connected (i.e. a complete graph), each such permutation of the nodes corresponds to a valid path. Now for each enumerated path \( p \), we exam whether this is a legal path (i.e. make sure there is an edge between two adjacent nodes given in the permutation), and compute the length of \( p \). This takes only \( O(n) \) steps. The run time is \( O(n(n^2)!) \) since there are up to \( (n-2)! \) paths.

The second problem is the graph 3-coloring: given a (undirected) graph \( G \) with \( n \) nodes, we want to color the vertex in three colors s.t. no two vertex colored with the same color is connected by an edge. We give a simple brute-force algorithm. We simply try all possible coloring of the nodes. Since for each node, there are 3 choices as its color, and there are \( n \) nodes, so the number of possible coloring is \( 3 \times 3 \ldots \times 3 = 3^n \). Then for each enumerated (fixed) coloring, we examine each edge of \( G \) to see whether its two end nodes are of the same color. If so, this is not a good coloring. If no edges with same coloring on its two nodes, we find a valid 3-coloring of \( G \). Since there are no more than \( n(n-1)/2 \) edges in \( G \) (note each pair of nodes can have only one edge), and it takes constant number of steps to check whether this edge violates the coloring constraints, this second step can be performed in \( O(n) \) steps. Thus, the whole algorithm runs in \( O(n3^n) \) steps. This is not very efficient. Unfortunately, as we will see later, we perhaps can not do much better than this.