Lecture 2: Algorithm Analysis

The first subject is analyzing the brute-force algorithm for solving the stable matching problem. We can enumerate all possible perfect matching for \( n \) men and \( n \) women, and then check whether a matching is stable. However, as we discussed in class, there are \( n! \) possible perfect matching, which just grows too fast as \( n \) increases. This counting can be shown by induction: we pick one man and there are \( n \) choices to match this man, and then the rest reduces to matching \( n - 1 \) men and women.

Now, we discussed three problems as example problems we will study later in the semester. The first problem is the classic sorting problem: given \( n \) numbers, find a permutation of these numbers in the sorted order. Note: a brute-force algorithm would enumerate all possible permutations and then checking whether a permutation is sorted. While the second step is relatively fast to check (how?), enumeration again is not feasible: there are \( n! \) possible permutations.

The next 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 (and such path is called simple paths). 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 \( p \). So this step is not that bad, but again, enumerating all possible paths is hard: there are up to \( (n-2)! \) such (simple) paths between two nodes.

The third 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 relatively fast, but the overall algorithm will still be slow due to the last number of possible coloring. This is not very efficient. Unfortunately, as we will see later, we perhaps can not do much better than this.

Then, we discussed 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 class.