Lecture 1

An algorithm is a well defined computation procedure for processing a given input to produce a desired output.

This course covers how to design algorithms and how to analyze their performance.

Course structure

1. Introduction
    a. Example: Stable matching problem
    b. Math background
    c. Basic graph algorithms
2. Algorithm design techniques
    a. Greedy Algorithms
    b. Divide and conquer algorithms
    c. Dynamic programming
    d. Augmenting path methods
3. Intractability
    a. NP-completeness
    b. Approximation algorithms
4. Other models of computations
    a. Randomized algorithms
    b. Quantum algorithms

Specifying algorithms

To specify an algorithm, you can either use pseudocode or just a plain language description. Which one you should use is based on context and what your goal is. Want to just explain things in the clearest possible way, as these specifications will be read by other people, not by a computer.

Example:

```
int foo = 0
for (i = 0; i <= 100; i += 2)
  if x[i] != 0
    foo++
  endif
endfor
```

Instead, we could be a bit more abstract and language agnostic.

```
let foo be a counter initialized to 0
for every even value of i from 0 to 100
  if the ith entry of foo is nonzero, increment foo
```

The latter method of specifying algorithms is preferred for the class. It is less ambiguous, and what you are trying to do is more clearer. But you should take care to not oversimplify.

Example:

```
repeatedly find adjacent elements that are out of order and swap them
 until the list is sorted
```

This is very unclear and unambiguous, this should be avoided. Leaves a lot of the important specification details out.

Two main issues in algorithm analysis

- Correctness: the algorithm should produce correct output
- Efficiency: he algorithm should use as few resources as possible (ex. time, space, …)
    - We typically analyze this asa function of the input size, the number of bits it takes to specify the input.
    - Typically want bounds that hold in the worst case, but this might not be representative of how many resources are used in a typical execution.

Stable Matching (Marriage) Problem

Problem: Pair the members of two sets according to their preferences

Examples:

- Students and employers being matched for internships
- Marriage partners
- etc.

For simplicity, assume:

- the two sets have the same size
- we must match each member of the first set with exactly one member of the second set.

Definition: Given two sets S, T, a matching is a set of pairs (s, t) with $s \in S, t \in T$ such that each element of S appears in at most one pair, and each element of T appears in at most one pair.

Furthermore, a matching is perfect if every element of S and every element of T is in some pair.

Each member of the sets S and T ranks all members of the other set.

We will try to find a matching that is stable with respect to the preferences to avoid any defectors.

Definition: A matching is stable if there is no pair (s, t) that is not in the matching such that s prefers t over its assigned match partner, and t prefers s over its assigned partner

---

Example: Suppose we have two students A, B and two companies X, Y.

Preferences:

$A : X, Y$

$B : Y, X$

$X : A, B$

$Y : B, A$

An optimal matching in this case would be $A \leftrightarrow X$, $B \leftrightarrow Y$

Example:

$A : X, Y$

$B : X, Y$

$X : A, B$

$Y : A, B$

A stable matching in this case would again be $A \leftrightarrow X$, $B \leftrightarrow Y$. X would not be willing to defect for B, and A would not be willing to defect for Y.

It is a fact that no matter what the preferences are, there is always some possible stable matching between the two sets. We will later be able to show that this property is true.

Example:

$A : X, Y$

$B : Y, X$

$X : B, A$

$Y : A, B$

In this situation, both possible matchings are stable.