Notes for Lecture 1

1 Overview of the Course

In the first lecture, we reviewed the draft syllabus of the course, which may change slightly as the course proceeds. The course will have, roughly speaking, two main parts: the first part will introduce techniques to develop and analyse algorithms for a variety of problems, with an emphasis on graph algorithms; the second part will focus on applications in which one is interested in proving the *non-existence* of algorithms, that is, computational complexity, computability, and cryptography.

In the study of algorithms, we will talk about;

- Divide-and-conquer algorithms, and the use of the Master Theorem to solve the recurrence relations that come up in the analysis of such algorithms.
- Basic notions of graph theory (paths, connectivity, strong connectivity, DAGs, trees)
- Properties of DFS
- The "greedy" algorithm design technique, illustrated by minimum spanning tree algorithms
- Dijkstra's algorithm for shortest paths
- "Iterative improvement" technique illustrated by the Ford-Fulkerson method for finding network flows. Max flow - min cut theorem.
- Karger's randomized algorithm for global min cut
- Finding matchings in graphs using network flows
- Dynamic programming, illustrated by all-pairs shortest path, edit distance, knapsack, and traveling salesman

In the second part of the course we will study:

- Computational complexity theory: the definition of P and NP and the P versus NP problem, reductions, NP-completeness and we will see several NP-completeness proofs, for a variety of problems.
- Computability theory: undecidability of the halting problem, problems that are not even recursively enumerable, sketch of how to prove Gödel's theorem using the undecidability of the halting problem
- Cryptography: rigorous definitions of security for encryption and authentication. Pseudorandom functions. MACs and CCA-secure encryption using pseudorandom functions.

2 Some Review

We began by reviewing some mathematical prerequisites. Do you remember how many subsets can be formed out of a set of n elements? How many leaves are there in a complete binary tree of depth k? How many internal nodes does such a tree have?

Recall also the big-Oh notation: if $f(n)$ and $g(n)$ are two functions, we say that $f(n) = O(g(n))$ if there is a constant c such that for all $n \geq 1$ we have $f(n) \leq c \cdot g(n)$. Note that the use of the "=" is quite misleading, in that $f(n) = O(g(n))$ is not an equation. An easy-to-remember way to see if a big-Oh relation hold is that if $\lim_{n\to\infty} f(n)/g(n)$ exists and is finite then $f(n) = O(g(n))$.

For example, $n^2 = O(n^3)$ and $10n^2 + n\log n = O(n^2)$, but it is not the case that $n \log n = O(n)$.

This notation is convenient to represent the asymptotic running time of algorithms, because it allows us to ignore multiplicative constants and small additive terms.

If $f(n) = O(g(n))$ then we also write $g(n) = \Omega(f(n))$. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we write $f(n) = \Theta(g(n))$.

For example, $10n^2 + n \log n = \Theta(n^2)$, but it is not the case that $n^2 = \Theta(n^3)$.

3 Mergesort

Our first example of designing and analysing an algorithm is mergesort. Mergesort is an algorithm that takes a sequence of elements on which a total order is defined (for example, a sequence of integers), outputs the sequence in non-decreasing order. See for example Dasgupta et al. Section 2.3 for the description of mergesort. If $T(n)$ is the running time of mergesort on inputs of length n, then we have that, if we normalize the time scale appropriately,

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T\left(\frac{n}{2}\right) + n \qquad \text{if } n \geq 2 \end{aligned}$$

Unfolding the second equation shows

$$\begin{aligned} T(n) =& n + 2T\left(\frac{n}{2}\right) \\ =& n + 2\frac{n}{2} + 4T\left(\frac{n}{4}\right) \\ & \cdots \\ =& n + 2\frac{n}{2} + \cdots + 2^{k-1}\frac{n}{2^{k-1}} + 2^k T\left(\frac{n}{2^k}\right) \\ =& n + 2\frac{n}{2} + 4\frac{n}{4} + 8\frac{n}{8} + \cdots + nT\left(\frac{n}{n}\right) \\ =& n \log n \end{aligned}$$