Measuring Problem Complexity

When we began the semester, we focused on polynomial time complexity as a working definition of algorithm efficiency. We then explored a small number of algorithm design paradigms that allowed us to construct efficient algorithms for many important problems, including:

- graph traversal: BFS, DFS, PFS
- Computing (strongly) connected components
- Topological Sorting of DAGs
- Many resource scheduling problems
- Shortest paths in graphs
- Minimum-cost spanning tree construction
- Optimal RNA folding
- Sequence alignment
- Optimal (minimizing scalar multiplications) multiplication of a sequence of matrices
- Optimal flows in networks
- Maximum matchings in bipartite graphs
- Disjoint path computation

Along the way we discovered that some of our algorithms were not as efficient as we might like

- Subset sum
- Knapsack problems
- Text Formatting with Slack² Minimization

These algorithms might take exponential time in some cases.

These latter problems are just a few examples of a vast set of disparate problems that share many common features

- There are no known algorithms that can efficiently solve all instances of the problem: Every known algorithm requires, in some cases, exponential time
- There are no known proofs that no efficient (polynomial time) algorithms exist to solve these problems
- These problems are equally difficult: That is, an efficient algorithm for any one of them would yield efficient algorithms for all of them.
- When these problems are decision problems (e.g., produce output yes or no), and when the answer is yes, there is a short "proof" of the correctness of the answer that can be verified efficiently!
We’ll be spending most of the rest of the semester exploring these claims, and looking for loopholes: methods that might let us:

- Efficiently solve a range of instances of the problem
- Efficiently produce approximate solutions for all instances of the problem
- Efficiently solve instances with high probability

But first we need to carefully define the notion of one problem being ”no harder than” another....

**Polynomial Reducibility**

**Definition 1. Turing Reduction**

A problem $X$ Turing reduces or polynomially reduces to a problem $Y$ ($X \leq_p Y$) if, for any instance $I$ of $X$, $I$ can be solved by an algorithm that

- Executes at most a polynomial number of standard computational steps
- Makes at most a polynomial number of calls to a method that solves problem $Y$

The ”method” that solves instances of $Y$ is called an oracle for $Y$; a call to the oracle for $Y$ is counted as a single computational step.

This definition is pretty useless for comparing problems $X$ and $Y$ both of which have polynomial time solutions, since $X \leq_p Y$ trivially (don’t call the oracle for $Y$ at all) and similarly $Y \leq_p X$.

However, if one (or both) of $X$ and $Y$ are of unknown time complexity, then this definition gives a measure of how much harder $Y$ might be than $X$.

Example: Subset Sum and Knapsack

**SubsetSum:** Given $S = \{w_1, \ldots, w_n\}$ and $W$ all positive integers, find $R \subseteq S$ with maximum sum subject to $\sum_{w \in R} w \leq W$.

**Knapsack:** Given $S = \{w_1, \ldots, w_n\}, W$, and a non-negative value function $v : S \rightarrow \mathbb{N}$, find a subset $R \subseteq S$ that maximizes $\sum_{w \in R} v(w)$ subject to $\sum_{w \in R} w \leq W$.

Then $\text{SubsetSum} \leq_p \text{Knapsack}$: The algorithm for Subset sum first defines $v(w) = w$ for each $w \in S$, then calls an oracle for Knapsack with $(S, v, W)$.

Note that we don’t use the ”full power” of the oracle to solve arbitrary Knapsack instances, but only a very specific kind of Knapsack instances (i.e., $v$ is the identity function on $S$).

We can easily establish the following

- If $X \leq_p Y$ and $Y$ can be solved in polynomial time, then $X$ can be solved in polynomial time
- If $X \leq_p Y$ and $X$ cannot be solved in polynomial time, then $Y$ cannot be solved in polynomial time
- If $X \leq_p Y$ and $Y \leq_p Z$ then $X \leq_p Z$ (transitivity)

This motivates the following definition

**Definition 2. Polynomial Equivalence**

Two problems $X$ and $Y$ are polynomially equivalent ($X =_p Y$) if $X \leq_p Y$ and $Y \leq_p X$
Note that polynomial equivalence is (not surprisingly) an equivalence relation, so it groups *problems* that have time complexity within a polynomial factor of one another.

Now let’s look at some further examples....