### NP Completeness

- **P** is the set of problems you can solve in polynomial time, e.g., minimum spanning tree, matchings, flows, shortest path.
- **NP** is the set of problems you can verify in the polynomial time:
  - If the answer should be yes then there’s some extra input (a “witness” or “certificate” or “hint”) that you can be given that makes it easy (i.e., in poly time) to check answer is yes
  - If the answer should be “no” then there doesn’t such an input.
- **Y** is **NP**-Complete if **Y** ∈ **NP** and **X** ≤p **Y** ∀ **X** ∈ **NP**.
- Useful Properties: Suppose **X** ≤p **Y**. Then
  - If **Y** ∈ **P** then **X** ∈ **P**.
  - If **Y** ∈ **NP** and **X** is **NP**-complete then **Y** is also **NP** complete
  - If **Y** ∈ **P** and **X** is **NP**-complete then **P** = **NP**.
- **NP**-Complete problems are in some sense the hardest problems in **NP**. If you can solve one of them in polynomial time then you prove **P** = **NP**. But very few people believe this is possible.

### Polynomial Time Reductions

- We focus on decision problems, e.g., for input (G, k), the question is does there exist a vertex cover with at most k nodes.
- Given two decision problems **X** and **Y**, **X** ≤p **Y** means that it’s possible to transform an input I of **X** into an input f(I) of **Y** in polynomial time such that
  
  I is a yes instance of **X** iff f(I) is a yes instance of **Y**

  The transformation is a reduction from **X** to **Y**.
- Useful property: If **X** ≤p **Y** and **Y** ≤p **Z** then **X** ≤p **Z**.

### Approximation Algorithms

- Even if a problem **NP**-complete, it is often possible to approximate the optimization version of the problem (e.g., find the smallest value of k such that there is a vertex cover of size k).
- An algorithm for a minimization problem is an α-approximation if for all input

  \[
  \frac{\text{value returned by the algorithm}}{\text{value of optimum solution}} \leq \alpha
  \]

  for some value of α ≥ 1. Ideally α would be close to 1.
- We saw a 2-approximation for vertex cover and k-center clustering, a 1.5-approximation for load balancing, and a ln n approximation for set cover.
- An algorithm for a maximization problem is an α-approximation if for all input

  \[
  \frac{\text{value of optimum solution}}{\text{value returned by the algorithm}} \leq \alpha
  \]

### Randomized Algorithms

- For some problems, the fastest known algorithm is randomized rather than deterministic, i.e., it uses some random bits to determine some of the steps it takes.
- We saw examples such as Median Finding, QuickSelect, MinCut, Resource Contention.
- Two main types of randomized algorithms:
  - Algorithms that give the wrong answer with some probability. We can normally make this probability arbitrarily close to 0 by repeating the algorithm multiple times.
  - Algorithms that always return the correct answer but whose running time is random. We want the expected running time to be polynomial.

### Network Flows

- Flow network
  - Directed graph
  - Source node s and target node t
  - Edge capacities c(e) ≥ 0
- Flow
  - Capacity Constraints: 0 ≤ f(e) ≤ c(e) on each edge
  - Conservation Constraints:
    \[
    f^{\text{in}}(v) = 0, f^{\text{out}}(t) = 0, \forall v \in V \setminus \{s, t\} f^{\text{in}}(v) = f^{\text{out}}(v)
    \]
    where \( f^{\text{in}}(v) = \sum_{e \in \text{ in to } v} f(e) \) and \( f^{\text{out}}(v) = \sum_{e \in \text{ out of } v} f(e) \)
  - Max flow problem: find a flow of maximum value \( v(f) = f^{\text{out}}(s) \)
  - Residual network encodes how you can change the current flow without violating the capacity constraints.
  - Ford Fulkerson Algorithm: Repeatedly increases the flow by finding augmenting paths in the residual network.
Dynamic Programming

Whenever you are tempted to write a recursive algorithm, ask yourself whether it is better to write a dynamic program:

- Consider all the sub-problems that you might need to solve if you wrote the algorithm recursively. Solve them all in some systematic way that usually starts by solving the simplest ones first.

- For example suppose we want to want to pick the a subset of the positive values \( \{x_1, x_2, \ldots, x_n\} \) that sums up to as close to \( W \) as possible but don’t exceed \( W \).

- Let \( D[i, w] \) be the closest you can get to \( w \) when restricted to picking from \( \{x_1, \ldots, x_i\} \).

- Then \( D[i, w] = \max(D[i-1, w], x_i + D[i-1, w-x_i]) \)

- Subproblems: Find \( D[i, w] \) for \( 1 \leq i \leq n \) and \( 1 \leq w \leq W \)

- Can find \( D[n, W] \) by solving all subproblems.
Divide and Conquer Algorithms Recap

- Given a problem of an input of size $n$,
- We generate (multiple) smaller instances of the problem
- We solve each of these smaller instances
- We use the solutions of the small instances to solve the original problem.
- Suppose that the first and third steps can be performed in $O(n^\alpha)$ time. If there are $a$ smaller instances generated, each of size $n/2$, then the running time $T(n)$ of the algorithm satisfies the recurrence.

$$T(n) \leq aT(n/2) + O(n^\alpha)$$

Greedy Algorithms

- Greedy algorithms are “short sighted” algorithms that take each step based on what looks good in the short term.
  - Example: Kruskal’s Algorithm adds lightest edge that doesn’t complete a cycle when building an MST.
  - Example: When maximizing the number of non-overlapping TV shows we always added the show that finished earliest out of the remaining shows.
- Things to note: 
  - If a greedy algorithm requires first sorting the input, remember to include the running time of sorting in your overall analysis.
  - Usually greedy algorithms can easily be shown to be poly-time but often extra work is required to get the most efficient implementation, e.g., union-find data structure.
  - Correctness proofs can be tricky: saw proofs by contradiction and induction, rearrangement arguments, some graph theory.
  - Another example: minimizing number of coins when giving someone change.

Asymptotic Analysis

Given two positive functions $f(n)$ and $g(n)$:

- $f(n) = O(g(n))$ iff $f(n)/g(n)$ tends to some constant $c \geq 0$ as $n \to \infty$
- $f(n) = \Omega((g(n))$ iff $g(n)/f(n)$ tends to some constant $c \geq 0$ as $n \to \infty$
- $f(n) = \Theta((g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega((g(n))$.

Divide and Conquer: Recurrences

- Suppose $T(n) \leq aT(n/2) + n^\alpha$ and $T(1) \leq 1$. Then:

$$T(n) =
\begin{cases} 
O(n^\alpha) & \text{if } \alpha > \log_2 a \\
O(n^{\log_2 a}) & \text{if } \alpha < \log_2 a \\
O(n^\alpha \log n) & \text{if } \alpha = \log_2 a
\end{cases}$$

- If you forget this formula just apply the “unrolling method”:

$$T(n) \leq aT(n/2) + n^\alpha \\
\leq a(aT(n/4) + (n/2)^\alpha) + n^\alpha \\
\leq a(a(aT(n/8) + (n/4)^\alpha) + (n/2)^\alpha) + n^\alpha \\
\leq \cdots$$

- Some example recurrence: $T(n) \leq T(n/2) + 1$ and $T(n) \leq 4T(n/2) + n$
- Another a divide and conquer example: counting inversions.

Graph Algorithms: BFS and DFS Trees

- BFS trees with root $r$:
  - Partitions the nodes into levels $L_0 = \{r\}, L_1, L_2, L_3, \ldots$ where $L_i$ consists of all neighbors of nodes in $L_{i-1}$ that aren’t already in $L_0 \cup L_1 \cup \ldots \cup L_{i-1}$.
  - If $v \in L_i$ the length of the shortest path in the original graph between $r$ and $v$ is $i$.
  - For any edge $(u, v)$ in the original graph, $u$ and $v$ are in the same level or adjacent levels.
- DFS trees with root $r$:
  - For any edge $(u, v)$ in the original graph, $u$ is an ancestor of $v$ in the tree or vice versa.
  - Can be used to find the connected components of a graph and test whether the graph is bipartite.
  - A directed graph is acyclic is there is no directed cycle. There is no directed cycle iff there is a topological ordering. Can find a topological order using the fact that a DAG has a node with no incoming edges.