CS41 Algorithms Study Guide

Contents

1 Stable Matching

- **Def:** An **instability** is a pair (d, h) in matching S such that:
  1. $(d, h') \in S$
  2. $(d', h) \in S$
  3. d prefers $h > h'$ and h prefers $d > d'$
- **Def:** A **stable matching** is a perfect matching with no instability, i.e. everyone gets matched

1.1 Gale-Shapley Algorithm

Stable Matching Problem
- Input: a list of doctors and hospitals and their complete preference list
- Output: doctor-hospital matchings S

Facts:
1. S is a stable matching.
2. The algorithm favors the proposing side (i.e. hospitals)
3. For doctors (receiving side), once you're matched, you're never unmatched. (So although you might be unhappy with your choice, at least you're employed.)
4. **Runtime:**
   while loop runs $O(n^2)$, each while loop does $O(n)$ work, so the algorithm is $O(n^3)$.

2 Algorithm Analysis

- search space: the set of all possible solutions
- brute-force search: iterates through the entire search space
- efficient: polynomial-time
- tractable: if an algorithm is efficient, it's tractable

2.1 Big-O, Big-Omega, Big-Theta

1. **Big-O (asymptotic upper bound)**
   Def: $f = O(g)$ if there exists constant $c, n_0 > 0$ such that $f(n) \leq cg(n)$ for all $n > n_0$

Fact: Big-O is transitive.

2. **Big-Ω (asymptotic lower bound)**
   Def: $f = \Omega(g)$ if there exists constant $c, n_0 > 0$ such that $f(n) \geq cg(n)$ for all $n > n_0$

3. **Big-Θ (asymptotic tight bound)**
   Def: If $f = O(g)$ and $f = \Omega(g)$, then $f = \Theta(g)$

2.2 Runtime facts

- for any constant $b > a > 1$, $\log_a(n)$ is $\Theta(\log_b(n))$
- for any constant $\varepsilon > 0, \log(n)$ is $O(n^\varepsilon)$ (i.e. logorithm is always Big-O of polynomials)
- for all $r > 1$, $d > 0$, n^d is $O(r^n)$ (i.e. polynomial is always Big-O of exponentials)

3 Graphs

3.1 Adjacency Matrix vs. Adjacency List

- Adjacency matrix: $n \times n$ matrix, where n is the nodes of the graph and the entries into the matrix are the edges 1 = edge; 0 = no edge
  - **Good:** fast looking up an edge;
  - **Bad:** finding next neighbor is slow; space is n^2
- Adjacency List: n lists, where n are the nodes, and each list stores all the node's neighbors.
  - **Good:** fast finding next neighbor; space is bound in $n + m$
  - **Bad:** looking up an edge is slower

$$\boxed{\textbf{BFS and DFS should use adjacency list}}$$

3.2 General Graph Algorithms and Properties

3.2.1 BFS and DFS

Fact:

1. **DFS could be used to find reachables.**
2. **Runtime:** $O(m + n)$
3. BFS and DFS **only works** on **connected** graphs.

```
BFS/DFS
Get a stack
get a set to store visited nodes
pick a start and add to stack
while stack not empty
    pop node
    add to set of visited nodes
    for every outgoing neighbor
        add to stack
if visited node not equal to original set
    return false
```

Definition: A graph is **connected** if for every $u, v \in V$ there exists a path from u to v.
Definition: Vertices u and v are **strongly connected** if there is a path from $u \rightsquigarrow v$ and from $v \rightsquigarrow u$.

3.3 Testing Bipartiteness

Definition: A undirected graph $G = (V, E)$ is **bipartite** if its vertices can be divided into 2 subsets X and Y such that all edges $e \in E$ has one endpoint in X and the other in Y.

Problem
- Input: undirected $G : (V, E)$
- Output: Yes if G is bipartite; No if G is not bipartite

Algorithm Idea: The algorithm is basically just start coloring the nodes using BFS. If we can do it, return Yes, else return No. Since we're basically doing BFS, **runtime** is $O(m + n)$.

Fact:
1. If G is bipartite, then G contains no odd length cycles.
2. Graph G is bipartite if and only if it is 2-colorable.
3. A graph with max degree at most k is $k + 1$-colorable.

3.4 Directed Acyclic Graph (DAG)

Definition: A **topological ordering** of a DAG is an ordering of nodes such that all edges point forward. If a graph G has a topological ordering, then it's a **DAG**.

3.4.1 Topological Sort

Problem
- Input: a DAG
- Output: a topological ordering of V

Idea: Preprocess all in-degrees; There must be one node with in-degree 0, so add that to safe to pick queue; then, while queue not empty, dequeue, add to list, then for every outgoing neighbor, decrease in-degree by 1, and if in-degree becomes 0, add to safe to pick; at the end, return the list of nodes.

Runtime: $O(n^2)$

4 Greedy Algorithms

Definition: an algorithm is **greedy** if it builds a solution one step at a time, according to some strategy with *no backtracking.*

General Correctness Proof Strategies:
1. "Stay Ahead" Argument
2. The Exchange Method

4.1 Interval Scheduling

4.2 Minimum Spanning Tree

Definition: A **minimum spanning tree** (MST) is a subset of edges of an undirected weighted graph such that the subgraph is connected, and contains no cycles, and the weight is minimum.

4.2.1 Kruskal's Algorithm

Idea: Keep adding cheapest edge that wouldn't create a cycle

4.2.2 Prim's Algorithm

Idea: Keep adding cheapest edge with one vertex not in a stored set until the stored set equals the original vertex set.

Runtime: $O(m \log(n))$

4.2.3 The Cut Property

If you divide the nodes into subset A and B, then the cheapest edge connecting node $a \in A$ and node $b \in B$ must be in the spanning tree.

5 Divide and Conquer

6 Dynamic Programming

Definition: Dynamic Programming is a method of solving complex problems by breaking down the problem into smaller subproblems, solving these subproblems, and storing the solutions that will at the end be used to build up the solution to the problem.

Dynamic Programming Steps:
1. Characterize the structure of an optimal solution

2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution
4. If necessary, construct the optimal solution from computed information

Facts:

1. In general, you always need a **base case**.
2. **Runtime:** of dynamic programming usually depends on the "table" generated; Make sure at each entry of the table, the algorithm is doing something in constant time. If the amount of work done in each entry of the Dynamic programming table is not constant time, you have to multiply that by the total number of entries.
3. **Correctness:** usually proof by induction starting from the base case.

6.1 Tabulation vs. Memorization

1. **Tabulation:** Bottom-up approach; fill the entire table from small subproblems to big subproblems
2. **Memorization:** Top-down approach; stores the results of previous calls and return them if called again.

When using memorization, since function stacks are a finite resource, the machine might encounter stack overflow. Also, tabulation is **a constant factor** faster when all table entries are needed.

6.2 Dynamic Programming Problems:

6.2.1 Steel Rod Problem

Problem

- Input: n length rod; list of prices $P[i], i = 1, 2, ...n$ which is the revenue for selling rod of length i.
- Output: max revenue from cutting the rod + places to cut to get this revenue OR places to cut it

Strategy: Notice that:

$$OPT(n) = OPT(\text{left chunk}) + OPT(\text{right chunk})$$

and, let function $r(n)$ be the max revenue of rod of length n

$$r(n) = \max(r(j) + r(n-j)), \quad j = 0, 1, 2, ..., n-1$$

Memorization Solution:

```
CUTROD($n, P$)
1  if $n = 0$:
2        return 0
3  q = -1
4  for $i = 1, 2, ..., n$
5        q = max(q, $P[i]$+ CUTROD($n - i, P$))
6  return q
```

Tabulation Solution:

```
CUTROD($n, P$)
1 Initialize array r[$0, ..., n$]
2 r[0] = 0
3 for $j = 1, ..., n$ // loop to fill in the entire array r[$j$]
4     q = -1
5     for $i = 1, ..., j$ // loop to find the max revenue of length $j$
6         q = max(q, $P[i]$+ r[$j - i$])
7     r[$j$] = q
8 return r[$n$]
```

Runtime: $O(n^2)$

6.2.2 RNA problem

Problem
- Input: a string of RNA $B = b_1 b_2 ... b_n$, where each $b_i \in \{A, U, C, G\}$
- Output: size of the largest substructure S OR S itself

6.2.3 Subset Sum

Problem
- Input: integer $W > 0$; a list of n items $\{1, 2, ..., n\}$ each with weight w_i
- Output: a subset $S \subseteq \{1, 2, ..., n\}$ such that $\sum_{i \in S} w_i$ is as large as possible but $\leq W$; algorithm should be $O(nW)$.

Strategy:

$$OPT(\{1, ..., j\}, L) = \begin{cases} OPT(\{1, ..., j-1\}, L) & \text{if } j \text{ is not in the subset} \\ \{j\} \cup OPT(\{1, ..., j-1\}, L - w_j) & \text{if } j \text{ is in the subset} \end{cases}$$

This is only considering subsets (and used to build the actual subset if needed).
For weights:

$$OPTW(\{1, ..., j\}, L) = \begin{cases} OPTW(\{1, ..., j-1\}, L) & \text{if } j \text{ is not in the subset} \\ w_j + OPTW(\{1, ..., j-1\}, L - w_j) & \text{if } j \text{ is in the subset} \end{cases}$$

Base cases:

$$OPT(\{\}, L) = \{\} \quad OPTW(\{\}, L) = 0$$
$$OPT(\{1, ..., j\}, 0) = \{\} \quad OPTW(\{1, ..., j\}, L) = 0$$

6.2.4 Weighted Interval Scheduling (Hoagie Problem)

6.2.5 Longest Common Subsequence

Definition: A **subsequence** in a string is a sequence of letters that appears in the same order, but not necessarily contiguously. For example, "alg", "arhm", "goth", and "loihm" are all subsequences of "algorithms", but "xyz", "algog", "gla", and "rihtms" are not subsequences.

Definition: A **common subsequece** between two strings a, b is a string c that is a subsequence of both a and b. For many applications including in computational biology, we would like a way of finding the largest common subsequence between two strings.

Problem
- Input: 2 n-character strings
- Output: size of the largest subsequence between them

1. Define a function $OPT(s_1, s_2)$ such that it takes in two strings s_1 and s_2 and return the size of the largest common subsequence between them. Let $a[i:j]$ be the substring of a from $a[i]$ to $a[j]$, i.e. from the i^{th} letter to the j^{th} letter. Suppose we know that $a[x] = b[y], 1 < x < n, 1 < y < n$ is in the largest common subsequence between a and b (both with n characters). Then, the optimal solution is:

$$OPT(a,b) = OPT(a[1:x-1], b[1:y-1]) + 1 + OPT(a[x+1:n], b[y+1:n])$$

2. Now consider the last letter of a and b, $a[n]$ and $b[n]$. Then, there are two cases:

$$OPT(a,b) = \max \begin{cases} OPT(a[1:n-1], b[1:n-1]) + 1 & \text{if } a[n] = b[n] \text{ and } n-1 \neq 0 \\ \max(OPT(a[1:n-1], b), OPT(a, b[1:n-1])) & \text{if } a[n]! = b[n] \text{ and } n-1 \neq 0 \end{cases}$$

6.2.6 Longest Palidrome Subsequence

Definition: A **palindrome** is a string which reads the same backwards and forwards. Let s be a string of characters from an *alphabet* $\sum$ and let $c \in \sum$ be some character. The reversal of s is denoted as s^R. Then, the strings ss^R (s concatenated with s^R) and scs^R are both palindromes.

Problem
- Input: a n-character string s
- Output: size of the largest palindrome in s

Strategy: Let $s = s_1 s_2 ... s_n$ Define a function $L(1,n)$ that outputs the size of largest palindrome from index s_1 to s_n. We want to find $L(1,n)$.
Then:

$$L(1,n) = \max \begin{cases} L(2, n-1) + 2 & \text{if } s_1 = s_n \\ \max(L(1, n-1), L(2, n)) & \text{if } s_1 \neq s_n \end{cases}$$

Base cases:

$$L(i,i) = 1 \quad \text{for any } i \text{ index - 1-char substring}$$
$$L(i,j) = 2 \quad \text{if } s_i = s_j \text{ and substring is size 2}$$

This is basically filling in a $n \times n$ table (only upper diagonal is filled) where row is start index of substring and column is end index of substring and each entry stores the max length of largest palindrome. So runtime is $O(n^2)$

7 Network Flow

Network Flow Problem
- Input: G described below
- Output: A maximum flow

Question: Why can't you use greedy approach to solve network flow problems?
- Because you need backtracking.

Graph for Network Flow Problems
1. $G = (V, E)$ is a **directed** graph
2. every edge $e \in E$ has an **integer edge capacity** $C_e \geq 0$
3. 2 special nodes: **source** $s \in V$ (no incoming edges) and **sink** $t \in V$ (no outgoing edges)

Flow
Definition: A **flow** is a **function** $f : E \to \mathbb{R}^+$ that describes how much flow goes through each edge such that:
1. **Flow is conserved for all nodes except source and sink**
2. **Flow through an edge is limited by edge capacity**

Storing Flow: Use a dictionary where keys are edges and values are flows corresponding to that edge. Under the assumption that hash tables have $O(1)$ lookups, you have constant-time lookups for the flow on an edge.

Residual Graph
Definition: For a flow network G and a flow f on G, the **residual graph** G_f is a graph such that:
- It has the same vertices as G
- For the edges, if $e \in E$ and there's flow through it; there will be forward edge with capacity $C_e^f = C_e - f(e)$ and reverse edge with capacity $f(e)$; if not, then there will be a forward edge with the same capacity as $e \in E$.

7.1 Ford-Fulkerson Algorithm

Algorithm idea: Build a **residual graph**. Repeatedly find an **augmenting path** to send flow ($s \rightsquigarrow t$) through the residual graph. (This will be done by running **DFS** and return path p) Add this to our flow in G ($O(m)$ since we're updating edges), update the residual graph ($O(2m)$ since we are updating both forward and reverse edges), and repeat until you can't anymore.

*(For this class, use FF-algorithm as subroutine rather than messing with the algorithm.)

Definition: A path $s \rightsquigarrow t$ in the residual graph which can carry some non-zero amount of flow is an **augmenting path**.

Runtime: $O(C(m+n)) = O(Cm)$, where C is the max capacity (i.e. max flow out of the source) and m is the number of edges and n is the number of vertices. ($O(Cm)$ since $m \geq n/2$ so we can substitute n to simplify runtime.)

Analysis:
Definition: An **s-t cut** is a partition of vertices such that s is in one set and t is in the other.

For every flow network, the maximum flow is qual to the minimum capacity of s-t cut.

Fact:
1. Ford-Fulkerson returns max flow of the input graph G from s to t.
2. Ford-Fulkerson works for all **directed** graphs with no restrictions on connectedness, whether it's cylic or non-cyclic.
3. **Max flow:** $z = \sum_{e=(s,*)} C_e$ (sum of all edge capacities coming out of source)
4. Technically not polynomial (it's called **pseudo-polynomial**) because runtime includes max capacity C which is just a number and C is not polynomial in its input size.
5. If all capacities in the flow network are integers, then there is a maximum flow f for which every flow value $f(e)$ is an integer.
6. When we consider **real valued capacities**, Ford-Fulkerson can **run forever!**
   - Flow increment each time can become arbitrarily small, i.e. while loop might never end.

   $\Rightarrow$ **Algorithm does NOT halt!**

7.2 Flow Variant Problems

7.2.1 Network Flow with Vertex Capacity

Idea: Reduce problem into normal network flow problem: for each vertex, **create an identical vertex and add an edge** between them with edge capacity as the vertex capacity. Redirect outgoing edges to new vertex. Run Ford-Fulkerson Algorithm on new graph with s and new $t*$.

7.2.2 Network Flow with Multiple Sources and Sinks

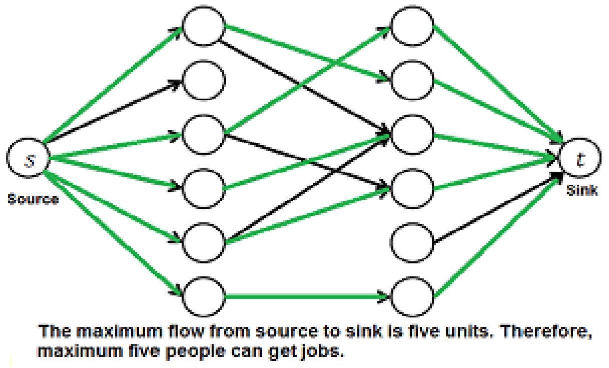
Idea: Reduce problem into normal network flow problem: **create a central source and central sink** that's the sum of the edge capacities of (outgoing edge capacities of source/incoming edge capacities of sink).

7.3 Maximum Bipartite Matching

Problem
- Input: Given bipartite graph $G = (V = (X, Y), E)$
- Output: Find **maximum matching**

Definition: A **matching** is a subset of edges $M \subseteq E$ such that each vertex appears in at most one edge of M

Algorithm Idea: Build a flow network and run Ford-Fulkerson.


s
Source
t
Sink
The maximum flow from source to sink is five units. Therefore, maximum five people can get jobs.

Remember:

- All edge capacities are 1
- After finding max flow using Ford-Fulkerson, you need to trace flow to return the corresponding matchings.
- **Runtime:** $O(n \cdot m)$, n = number of vertices, m = number of edges.

8 NP-Completeness

Definition: An **Optimization Problem** is one where the objective is to find the best solution among all feasible solutions.

Definition: An **Decision Problem** is one where the output is YES/NO or TRUE/FALSE.

List of NP-COMPLETE problems: (**Technically they can all reduce to each other.**)

1. SAT, 3SAT, one-in-three SAT
2. 3-coloring, k-coloring
3. Subset-sum
4. Vertex Cover, Independent Set, Set Cover, Clique
5. Travel Salesman Problem (TSP)
6. Multiple-Interval Scheduling
7. Hamiltonian Cycle
8. Intersection-Inference (ONE-IN-THREE SAT $\leq_P$ INTERSECTION-INFERENCE)

Note: Subset-Sum only becomes hard with truly large integers; when the magnitude of the input numbers are bounded by a polynomial function of n, the problem is solvable in polynomial time by dynamic programming.

To Show NP-Complete:

1. Show that there's a polynomial-time verifier
2. Pick a NP-COMPLETE (i.e. NP-HARD) problem and reduce it to the current problem

8.0.1 Cook-Levin Theorem

$$\boxed{\text{SAT is NP-COMPLETE}}$$

8.1 Polynomial Reduction (for Decision Problems)

Definition: Let A and B be 2 decision problems. A is **polynomial-reducible** to B ($A \leq_{\mathrm{P}} B$) if arbitrary instances of A can be solved using a polynomial amount of time plus a polynomial number of calls to a blackbox for B

$$\boxed{\text{Polynomial Reduction is } \textbf{transitive}.}$$

Analysis - need to check:

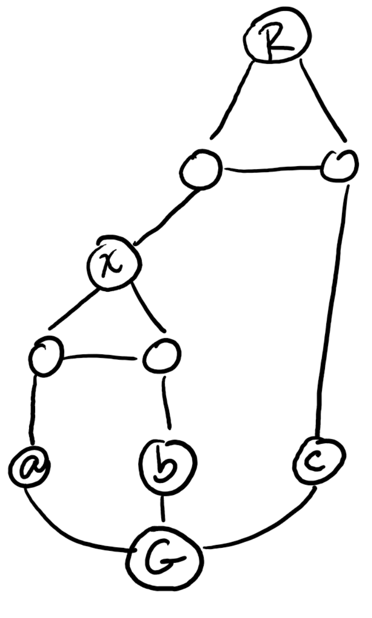
1. Reduction algorithm is polynomial-time and halts
2. **Correctness:** assume $A \leq_{\mathrm{P}} B$
   If x is a YES input for A, x' is a YES input for B.
   If x is a NO input for A, x' is a NO input for B. **OR** If x' is a YES input for B, then x is a YES input for A.

Reductions to understand:

1. Sat $\leq_P$ 3-Sat
   Consider cases where a clause in Sat has:
   - 1 literal: add 2 variables, and transform it to 4 clauses
   - 2 literals: add 1 variable, and transform it to 2 clasuses
   - 3 literals: nothing changes
   - k ($k > 3$) literals: add $k-3$ variables, and transform it to $k-3$ clauses
2. 3-Sat $\leq_P$ One-In-Three Sat
   Idea: For each clause in 3-Sat, add 4 new variables and 2 more clauses where:

$$(x \vee y \vee z) = \begin{cases} (\bar{x} \vee a \vee b) \\ (b \vee y \vee c) \\ (c \vee d \vee \bar{z}) \end{cases}$$

3. Vertex-Cover $\leq_P$ Independent-Set **and** Independent-Set $\leq_P$ Vertex-Cover
   Note: $VC(G,k) = IS(G, |V| - k)$
4. 3-Sat $\leq_P$ Independent-Set
   Idea: For each clause in 3-Sat, build a triangle. Between triangles, if variance u and v are negations of each other, add an edge $\{u, v\}$ between the triangles. Set $k = m$
5. 3-Sat $\leq_P$ Three-Coloring
   Idea: Forbid all nodes to be one of the colors (e.g. green), set one color to True (red) and the other to False (blue). Create 2 or gates.



6. One-In-Three Sat $\leq_P$ Intersection-Inference
   Idea: create a subset for each clause, with $c_i = 1$. For each variable x_i, add a subset $\{x_i, \bar{x_i}\}$ with $c = 1$.
7. Three-Coloring $\leq_P$ k-coloring
   Idea: For 4-coloring, add a node, connect to all the existing nodes. For k-coloring, add $k-3$ nodes, and connect to all existing nodes and to each other.

8. VERTEX-COVER $\leq_P$ SET-COVER
   Idea: Set the universe U to E the edges. For each vertex v_i, create a set s_i that contains all of edges with endpoint v_i. k remains the same.

8.1.1 Vectex Cover, Independent Set, Clique, Set Cover

Let $G = (V, E)$ be an undirected graph.
Definition: A **vertex cover** is a set of nodes $T \subseteq V$ such that every edge $\{u, v\} \in E$ has at least one endpoint in T.
Definition: An **independent set** is a set of nodes $S \subseteq V$ such that no pair $u, v \in S$ has $\{u, v\} \in E$.
Definition: A **clique** is a set of nodes $C \subseteq V$ such that every pair $u, v \in C$ has edge $\{u, v\} inE$

- INDEPENDENT-SET(G, k) returns YES iff there is an independent set in G of size $\geq k$,
- MAX-INDEPENDENT-SET(G) returns the size of the largest independent set in G,
- VERTEX-COVER(G, k) returns YES iff there is a vertex cover of G of size at most k,
- MIN-VERTEX-COVER(G) returns the size of the smallest vertex cover of G,
- CLIQUE(G, k) returns YES iff there is a clique in G of size k, and
- MAX-CLIQUE(G) returns the size of the largest clique.

Definition: Let U be the universe and $S = \{s_1, ..., s_m\}$ be the m subsets of U. A **set cover** is a set $T \subseteq S$ such that T covers the entire universe, i.e. $\bigcup_{s_i \in T} s_i = U$.

Fact:
1. If S is an independent set, then $V \setminus S$ is a vertex cover.
2. A clique and an independent set are complements of each other.
   (If $S \subseteq V$ is an independent set in G, create a graph G' with same V where if there's an edge in G, then it doesn't exist in G', and if there's no edge in G, then it's an edge in G'. Then $S \subseteq V'$ in G' is a clique.)
3. MAX-CLIQUE tells us potentially how many color is needed to color a graph. If size of max clique is k, you need at least k color to color the graph.

8.2 Polynomial Verifier

Definition: V is a **polynomial-time verifier** for a decision problem L if:
1. V is a polynomial-time algorithm on 2 inputs x and w, where x is all the inputs to L and w is some extra info (witness/certificate/hint) that tells us the solution.
2. There's a polynomial function p where:
   $x \in L$ is a YES input $\Leftrightarrow$ there exists some w polynomial in its input size, and $V(x, w) =$ YES

Analysis - need to check:
1. V is polynomial-time and halts
2. **Correctness:**
   If x is YES input, verifier returns YES.
   If x is NO input, verifier returns NO.

Remember:
1. Always define your w **as specific as possible**.
2. **Always check format** of input w

9 Approximation Algorithms

Definition: An **approximation algorithm** is one that runs in polynomial time, and finds a solution that's guaranteed close to the optimal solution by an *approximation ratio*.

Definition: An algorithm A has an **approximation ratio** $\rho(n)$ if for any input of size n, the solution produced by A has cost C, which is within a multiplicative factor of ρ of OPT cost of a solution.

$$\max\left\{\frac{C}{OPT}, \frac{OPT}{C}\right\}$$

we call A an $\rho(n)$-approximation algorithm.

Analysis - need to check:
1. Algorithm is polynomial-time and halts
2. **Correctness:**
   The approximation algorithm returns a **valid** output.
   The algorithm **guarantees an approximation ratio** (whether it's given or proved on your own).

Remember:
1. Find an intermediate value to show approximation ratio.
2. Problems where: if there exists a certain k-approximation in polynomial time, then P = NP
3. Useful facts:
   - If $A + B > C$, then either $A \geq \frac{1}{2}C$ or $B \geq \frac{1}{2}C$
   - If $A + B + C = X$, then either $A \leq \frac{1}{3}X$ or $B \leq \frac{1}{3}X$ or $C \leq \frac{1}{3}X$

9.1 The Pricing Method

9.2 Linear Programming

Definition: A **linear program** is written with two parts:
1. **Objective function**: minimize $cx_1 + c_2x_2 + ... + c_nx_n$
2. Subject to the **constraints**:

$$A \cdot \vec{X} \geq \vec{b}$$

   where A is an $n \times n$ matrix of constants. $\vec{X}$ a vector of all $x_i \geq 0$ and $\vec{b}$ a vector of constants.

Steps for Using LP for Approximation:
1. Express the problem as an integer linear program (with constraints relative to the nature of the problem)
2. Solve for real-valued solution, by setting an approximate threshold (this is where approximation comes in) for what's considered in the solution.

10 Randomized Algorithms

Fact:

Let X be the number of rounds a randomized algorithm A runs before halting. If A terminates with probability p each round, then the expected number of rounds A would run, $E[X] = \dfrac{1}{p}$

Two Types of Randomized Algorithms

1. Las Vegas: Always give **correct** answer; Analyze for expected runtime
2. Monte Carlo: Makes an error with probability $\leq 1/3$; Analyze for worst case runtime.
   (**How to remember:** LV is always right.)

Remember: Linearity of Expectation!

10.1 Randomized Algorithm as Approximation Algorithm

Strategy: Do Something Stupid, then prove expected output is good, then keep doing it until it gets to the approximation ratio you want.