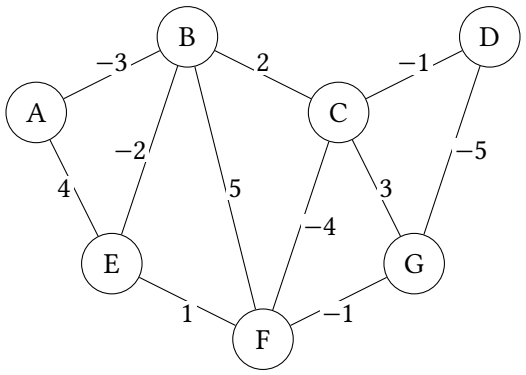
# CSE 421 Winter 2026: Section 3

January 22, 2026

**Instructions:** This section worksheet is designed to assist you in working through some example problems and developing your basics. You are encouraged to collaborate on the problems on this section as well as use the course's generative AI.

## 1 MST algorithm practice

Practice computing minimum spanning trees for this graph by implementing Prim's (starting from E) and Kruskal's algorithms by hand (with ties broken alphabetically). Compute the tree using both algorithms. What is the order in which the vertices/edges are processed?




## 2 Step-by-step algorithmic walkthrough

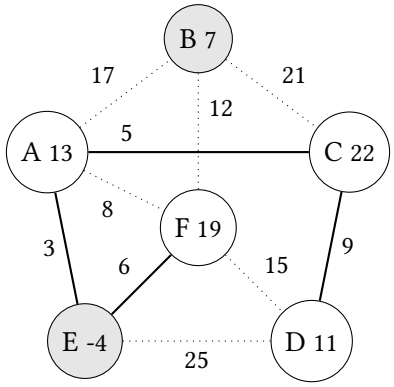
We model the problem of building a power grid as a graph problem. A power grid powers a city $v$ if either the city $v$ has a generator or there is a path along power lines that connects the city to a city $u$ that has a generator.

The cost to build a generator at city $v$ is given by array $g(v) \in \mathbb{R}$ (note, it could be negative). The set of power lines that can be built is described by an undirected weighted graph $G = (V, E)$ where $V$ is the set of cities. The cost to build a power line is given by $w(u, v)$ for $w : E \to \mathbb{R}_{\geq 0}$.

For this problem, design an algorithm that finds the optimal placement of generators and power lines to power all the cities in $V$. Assume that initially, no generators or power lines have been built, and that

"optimal" means minimal cost.

In the following example, selecting vertices $B$ and $E$ for generators and laying the solid power lines would cost $7 + (-4)$ in generators and $5 + 3 + 9 + 6$ for a total of 26.




Hint: Construct a new graph on $n + 1$ vertices for which you can use a known graph algorithm. Then come up with a 1-to-1 correspondence between solutions to the $n + 1$ vertex graph problem and the original problem at hand.

**Instructions:** You are free to solve this problem however you like — the following "step-by-step" procedure is our general technique for thinking about algorithms from last week's section.

1. **What is the problem asking?** It is hard to solve a problem if you don't know what you are supposed to do. Reread the problem slowly, highlighting key parameters, requirements, and the format of the "return type" that the answer needs to be provided in. Ask yourself the following questions:

   - Are they are technical terms in the problem that you don't understand?
   - What is the input type? (ex., Array, Graph (directed/undirected/weighted), Integer, something else)?
   - What is the output type? Do I need to keep track of decisions I make (ex. assignments, weights, values, etc.) as I run the algorithm?
   - For a word problem, can you extract the core algorithmic question?

2. **What are some good examples?** Come up with some examples in an effort to ensure that you truly understand the problem. Generate a few sample instances and calculate the optimal solution.

   Did you notice any patterns while coming up with the solution? If so, what were they, and can those "patterns" be expressed algorithmically? If so, what properties might we need to prove? At this stage, the goal is to just come up with a few (2-3) examples, but it is important to keep an eye on the

goal.

In terms of examples, our suggestion is to **not** focus on base or edge cases. Rather, we want to understand the general behavior of the algorithm. Come up with examples that are decently large ($n \approx 5 - 10$) and are sufficiently different. For example, if the problem is described by a graph, make sure that at least one example is *non-planar*.

3. **What is a baseline?** In a time-constrained setting (ex., interview or exam), it is good to come up with a "baseline" or "brute-force" algorithm. It may not be optimal, but it can help give you a baseline to compare any future improvement against.

4. **What problems does this remind me of?** A great starting point is identifying all the problems you have seen that are similar. Try thinking through the following questions:

   - Does this remind me of any prior problems? What techniques did we use there?
   - Does the problem look like a *standard* algorithm when turned on its head? For example, do I need to create a graph or adjust the data and *then* run a standard algorithm on the new graph/data?
   - Is there some structure to the input? For example, is it 1-dimensional (like this problem!)?
   - Is there some structure to the output? If we need to make multiple selections or decisions, can we make them *greedily*?

5. **Write an algorithm.** Try running your algorithms against your examples. Did something click? If one of your algorithms seems to be working, construct an example problem that bucks your algorithm – meaning make sure every path through your example is explored. If you have an "if" argument in your algorithm, make sure there exist examples that apply to both sides of the "if" statement. If you cannot come up with such an example, perhaps the "if" statement wasn't needed.

6. **Prove that your algorithm is correct.** As we explore more algorithms, we will also come up with more techniques for proving correctness. For this problem, there are a couple of equivalent proofs of correctness. Any will do; come up with the one that feels most natural for you to write.

# 3 Additional practice problems

## 3.1 Greedy algorithms can be deceiving: Traveling Salesman

One of the most famous optimization problems in algorithms is the *Traveling Salesman Problem (TSP)*:

> Given a set of cities and the distances between every pair, find the shortest *tour* that visits every city exactly once and returns to the start.

TSP is a great test case for algorithm design because it is easy to state, but NP-hard to solve optimally. A very natural idea is the following *greedy heuristic*:

> *Nearest-Neighbor Heuristic:* From your current city, always travel next to the closest city you have not visited yet.

This sounds sensible, but it can perform surprisingly poorly. To see this, go play the following Sporcle game:

`https://www.sporcle.com/games/Tr4pD00r/nearest-us-capital-roadtrip`

The game enforces the "visit the nearest next city" strategy on U.S. state capitals starting from D.C. As you play the game, observe and try to think of some ideas as to why this greedy algorithm is failing to produce the optimal strategy. Of particular interest to me is what capitals it has to visit near the end of the tour.

(Bonus): Construct an example of $n$ points on a 2D plane such that the optimal path (i.e., start at the left-most point and go right) is $\Omega(\log n)$ times shorter than the greedy path created by the nearest-neighbor heuristic.

### 3.2 Knapsack

**Part 1: The (0/1) Knapsack Problem.** You are given a weight capacity $W \in \mathbb{Z}_{>0}$ and $n$ items. Each item $i$ has an integer weight $w_i > 0$, and an integer value $v_i > 0$. You may either take an item in its entirety or not take it at all. The goal is to choose a subset of items whose total weight is at most $W$ and whose total value is maximized.
Consider the following greedy strategies:

1. Selecting items in decreasing order of value $v_i$.
2. Selecting items in increasing order of weight $w_i$.
3. Selecting items in decreasing order of density $v_i/w_i$.

For each strategy, construct a concrete example (with a small number of items) showing that the greedy strategy can be suboptimal for the $0/1$ knapsack problem.

**Part 2: The Fractional Knapsack Problem.** In the fractional knapsack problem, the setup is the same as above, except that you are allowed to take any fraction of an item. That is, for each item $i$, you may take any amount between 0 and 1 of the item, receiving proportional weight and value.

Design a greedy algorithm that always produces an optimal solution to the fractional knapsack problem. Describe the algorithm clearly in technical English and explain why the greedy choice is natural in this setting.

### 3.3 Coin Change

You are given a set of coin denominations $C = \{c_1, c_2, \ldots, c_k\}$, where each $c_i$ is a positive integer and $c_1 = 1$. You are also given a target value $V$. A *coin representation* of $V$ is a multiset of coins from $C$ whose total value is exactly $V$. The goal is to represent $V$ using as few coins as possible.

1. *The greedy algorithm.* Construct a greedy algorithm for computing the coin representation of $V$ using the fewest possible coins. Prove that your algorithm is correct when the coin denominations form a geometric series $c_i = a^{i-1}$ for $a \geq 2$.

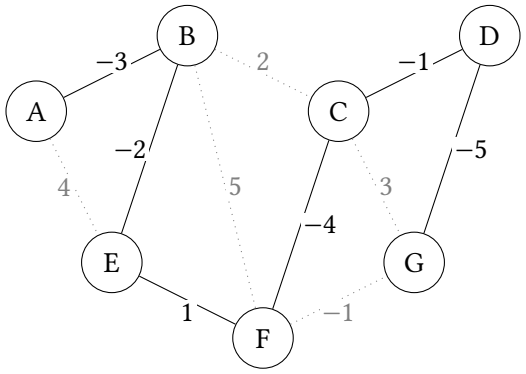
2. *Failure of greedy in general.* Construct an explicit set of coin denominations (including 1) for which the greedy algorithm does *not* produce an optimal solution for some target value $V$. Explain why the greedy choice fails in your example.

3. *Dynamic programming alternative.* Later in this course, we will learn of an algorithmic technique called dynamic programming, which can be used to solve the coin-change problem optimally for arbitrary coin denominations in time $O(Vk)$. Remember this problem for then, and you can return to this section worksheet to see how this algorithm works.

## 4 Solutions

### 4.1 MST algorithm practice

If you run Prim's starting from $E$, the visit order should be $E, B, A, F, C, D, G$. The order in which edges are added for Kruskal's algorithm is $D-G, C-F, A-B, B-E, C-D, E-F$. Both algorithms produce the same MST in this case, which is




### 4.2 Step-by-step algorithmic walkthrough

1. **What is the problem asking?**

We are given an undirected weighted graph $G = (V, E)$ of cities and potential power lines, where each edge $(u, v) \in E$ has nonnegative construction cost $w(u, v) \geq 0$, and each city $v \in V$ has a (possibly negative) generator cost $g(v) \in \mathbb{R}$. We must choose:

- a set $S \subseteq V$ of cities at which we build generators, and
- a set $F \subseteq E$ of power lines to build,

so that *every* city is powered, meaning: for each $x \in V$, either $x \in S$ or there is a path in the built graph $(V, F)$ from $x$ to some generator city $s \in S$. The objective is to minimize total cost

$$\sum_{v \in S} g(v) \; + \; \sum_{e \in F} w(e).$$

Some observations we hope students make during this step are:

- *any feasible solution must pick at least one generator.* Indeed, if $S = \emptyset$, then no city has a generator and no city can be connected to a generator by definition, so no city is powered. Thus $S \neq \emptyset$ is necessary.
- *we need to pick a subset of both edges and vertices.* We have seen algorithms in class for picking one or the other. Often, great techniques for such problems involve building a new graph on more vertices so that choices on the new graph can correspond to *either* edges or vertices in the original graph.

The output type is a pair $(S, F)$: the set of generator locations and the set of power lines built.

2. **What are some good examples?**

   Aside from the example provided,

   - *Single city.* If $V = \{v\}$ and $E = \emptyset$, then the only feasible solution is to build a generator at $v$ (so $S = \{v\}$) and build no lines. This reinforces the necessity that $S \neq \emptyset$.
   - *Two cities with one line.* Suppose $V = \{u, v\}$, one edge $(u, v)$ of cost $w(u, v) = 3$, and generator costs $g(u) = 5$, $g(v) = 1$. The cheapest feasible plan is to place a generator at $v$ and build the line $(u, v)$, total cost $1 + 3 = 4$, rather than building two generators (cost 6). This shows lines can substitute for additional generators.
   - *A negative generator.* If some city $x$ has $g(x) = -10$, then (all else equal) it is beneficial to include $x$ as a generator, and then connect other cities to $x$ using cheap lines if possible. This highlights that generator costs behave like "edges to a source" and may be negative.

3. **What is a baseline?**

   A brute-force baseline is: try every subset $S \subseteq V$ as the set of generator cities (discard $S = \emptyset$ immediately since it is infeasible). For each $S$, compute the minimum cost set of lines $F$ such that

every vertex is connected in $(V, F)$ to at least one vertex in $S$; then add $\sum_{v \in S} g(v)$ and take the best. This approach is exponential in $|V|$ because there are $2^{|V|} - 1$ nonempty subsets, so it is not efficient, but it clarifies that the decision of "where to put generators" in concurrence with "where to put the power lines" is the heart of the problem and motivates transforming the problem into a single standard graph optimization.

4. **What problems does this remind me of?**

   This problem resembles the *Minimum Spanning Tree (MST)* problem: we want to connect vertices with minimum total edge weight, and cycles are never helpful in an optimal network. The hint suggests creating a graph on $n + 1$ vertices so that a known algorithm applies. A natural way to enforce "at least one generator" and to account for generator costs is to add a new super-vertex (a virtual power source) and represent building a generator at $v$ as connecting $v$ to this super-vertex with an edge of cost $g(v)$. Then, powering all cities becomes the same as ensuring all cities are connected (in some way) to this super-vertex. This **is** the minimum spanning tree problem on the augmented graph; crucially, this is not the shortest path problem from the newly added source vertex.

5. **Write an algorithm (in technical English).**

   **Construction of an augmented graph.** Create a new graph $G' = (V', E')$ as follows:

   - Add one new vertex $r$ (think "root" or "super-source"). Let $V' = V \cup \{r\}$.
   - For each original edge $(u, v) \in E$, include it in $E'$ with the same weight $w(u, v)$.
   - For each city $v \in V$, add a new edge $(r, v)$ to $E'$ with weight $g(v)$.

   (These new edges may have negative weights; MST algorithms still apply.)

   **Compute an MST of $G'$.** Run a standard minimum spanning tree algorithm on $G'$: using **Kruskal's algorithm** or **Prim's algorithm** runs in time $O(|E'| \log |V'|) = O((|E| + |V|) \log(|V|))$ as $|E'| = |E| + |V|$. To extract a solution from the MST subroutine, let $T$ be the MST of $G'$. Define:

   - Generator set $S = \{ v \in V : (r, v) \in T \}$.
   - Power-line set $F = \{ (u, v) \in E : (u, v) \in T \}$ (i.e., all MST edges not incident to $r$).

   Answer with $(S, F)$.

   *Feasibility check (why at least one generator is chosen).* Since $T$ is a spanning tree on the vertex set $V' = V \cup \{r\}$, the vertex $r$ has degree at least 1 in $T$. Therefore there exists at least one $v \in V$ with $(r, v) \in T$, so $S \neq \emptyset$. Hence, the extracted plan always chooses at least one generator, as required for feasibility.

6. **Prove that your algorithm is correct (via a clean 1-to-1 correspondence).** We prove correctness by establishing a direct correspondence between feasible solutions to the original power-grid

problem on $G$ and spanning trees of the augmented graph $G'$, and showing that costs are preserved under this correspondence.

*From solutions in $G$ to spanning trees in $G'$.* Take any feasible solution $(S, F)$ for the original problem, where $S$ is the (nonempty) set of generator cities and $F$ is the set of built power lines. Form a subgraph of $G'$ by including:

- all edges in $F$, and
- for each city $v \in S$, the edge $(r, v)$ connecting it to the new vertex $r$.

Because every city is powered in $(S, F)$, each city has a path in $F$ to some generator in $S$, and each generator in $S$ is directly connected to $r$. Therefore, every vertex in $G'$ has a path to $r$, so this subgraph is connected and spans all vertices of $G'$. Removing any cycles yields a spanning tree of $G'$. The total weight of this tree is the total cost of $(S, F)$.

*From spanning trees in $G'$ to solutions in $G$.* Now, take any spanning tree $T$ of $G'$. Define a solution $(S, F)$ as follows:

- A city $v$ is in $S$ if and only if the edge $(r, v)$ appears in $T$.
- An original edge $(u, v)$ is in $F$ if and only if it appears in $T$.

Since $T$ spans all vertices of $G'$, there is a unique path in $T$ from any city to $r$. This path must include exactly one edge of the form $(r, v)$, which identifies a generator city $v$. The remaining edges on the path are original edges, giving a path from the city to a generator using edges in $F$. Thus, every city is powered, so $(S, F)$ is feasible. Moreover, because $T$ is a tree, $r$ has at least one incident edge, guaranteeing that $S$ is nonempty.

*Cost preservation and optimality.* Under this correspondence, costs are the same. Consequently, minimum-cost feasible solutions to the original problem correspond exactly to minimum spanning trees of $G'$. Since the algorithm computes an MST of $G'$ and then extracts $(S, F)$ using the rule above, the resulting solution is feasible and has the minimum possible total cost.

## 4.3 Additional practice problems

### 4.3.2 Knapsack

**Part 1** We give counterexamples for each proposed greedy strategy (there are many).

*Greedy by highest value.* Let the knapsack capacity be $W = 10$. Consider two items:

| Item | Weight | Value |
|---|---|---|
| 1 | 10 | 60 |
| 2 | 9 | 59 |
| 3 | 1 | 2 |

The greedy strategy selects item 1 (value 60), filling the knapsack completely. The total value is 60. But selecting items 2 and 3 gives total weight 10 and total value 64, which is better. Thus selecting by highest value is suboptimal.

*Greedy by smallest weight.* Let $W = 10$ and consider:

| Item | Weight | Value |
|---|---|---|
| 1 | 1 | 1 |
| 2 | 10 | 100 |

The greedy strategy selects item 1 first (smallest weight) and cannot select anymore. This is suboptimal as the selection of item 2 is better.

*Greedy by highest density.* Let $W = 10$ and consider:

| Item | Weight | Value | Density |
|---|---|---|---|
| 1 | 6 | 60 | 10 |
| 2 | 6 | 60 | 10 |
| 3 | 10 | 99 | 9.9 |

The greedy strategy will select one of items 1 and 2, achieving value 60. However, selecting item 3 alone gives value 99, which is better. Therefore, even density-based greedy fails for the 0/1 knapsack problem.

**Solution to Part 2: Fractional Knapsack** *Greedy Algorithm.* The optimal greedy strategy for the fractional knapsack problem is:

- Sort all items in decreasing order of density $v_i/w_i$.
- Initialize remaining capacity to $W$.
- Iterate through the sorted list:
  - If the current item fits entirely in the remaining capacity, take the whole item.
  - Otherwise, take exactly the fraction of the item that fills the remaining capacity and stop.

**Correctness** Since we can select fractional items, we can divide each item of weight $w_i$ and value $v_i$ into $w_i$ individual item-chunks, each of weight 1 and value/density $v_i/w_i$. Now all item-chunks have equal weight. Therefore, an exchange argument tells us that the selection of item-chunks by decreasing density is optimal. Since the item-chunks derived from the same item have the same density, it follows that an iterative algorithm selecting item-chunks will continuously select as many chunks as possible from the same item. Observe that this is the greedy algorithm presented.

### 4.3.3 Coin change

1. *The greedy algorithm (geometric denominations).*

   Assume the coin denominations are $1, a, a^2, \ldots, a^{k-1}$ for some integer $a \geq 2$. The greedy algorithm repeatedly selects the largest coin whose value does not exceed the remaining amount and subtracts it, until the remaining amount is zero.

   To prove correctness, consider any remaining value $R$ and let $a^j$ be the largest coin not exceeding $R$. Any way of forming $a^j$ using smaller coins requires at least $a$ coins, so replacing $a^j$ by smaller denominations cannot reduce the number of coins used. Therefore, there exists an optimal solution that includes a coin of value $a^j$. The greedy algorithm makes exactly this choice. Removing this coin reduces the problem to a smaller instance of the same form. Repeating this argument shows that the greedy algorithm is optimal.

2. *Failure of greedy in general.*

   Consider the coin denominations $C = \{1, 3, 4\}$ and target value $V = 6$. The greedy algorithm chooses 4, then 1, then 1, using three coins total. However, the optimal solution uses two coins, namely $3 + 3$.

   The greedy choice fails because selecting the largest coin early prevents a better combination of smaller coins. Unlike geometric coin systems, larger coins do not necessarily dominate combinations of smaller coins in terms of coin count.