Warm-up: Greedy or Not?

Sometimes it can be tricky to tell when a greedy algorithm applies. For each problem, say whether or not the greedy solution would work for the problem. If it wouldn't work, give a counter example.

1. You have unlimited objects of different sizes, and you want to completely fill a box with as few objects as possible. (Greedy: Keep putting the largest object possible in for the space you have left)
2. You have unlimited objects, all of which are size 3^k for some integer k, and you want to completely fill a box with as few objects as possible. (Greedy: same approach as the previous problem)
3. You want to print out a series of words (consisting of characters) on as few lines as possible while preserving their relative ordering. However, each line can only fit a fixed number of characters. (Greedy: Fit as many words as you can on a given line)
4. You want to get from hotel 1 to hotel n, and you can travel at most k distance between hotels before collapsing from exhaustion. Find the minimum cost of hotels. (Greedy: Go as far as you can before stopping at a hotel)

SOLUTION:

1. Greedy does not work! Consider a box of size 14 and objects of size 10, 7, and 1.
2. Greedy works! This is basically how you would write a number in base 3.
3. Greedy works!
4. Greedy does not work! Consider hotel costs [10, 20, 100, 10], where each hotel is 1 apart and k = 2.

Problem Solving Notes:

1. *Read and Interpret:* We're asking for if the greedy algorithm given can apply to each example. Does the algorithm work, or can you come up with a counterexample?
2. *Information Needed:* What are some different cases you can come up with for each example to test if the algorithm works or not?
3. *Solution Plan:* Test out some different cases to see if the greedy algorithm would apply!

Pareto Optimal

Given a set of 2d points P, a Pareto optimal point is a point (x, y) such that $\forall(x', y')$ we have either $x > x'$ or $y > y'$. Develop an algorithm to to find all Pareto optimal points.

SOLUTION: First sort the points by decreasing x coordinate and break ties using y. Clearly in this sorted list the first point is guaranteed to be pareto optimal (as it has the largest x coordinate and the largest y amongst points with that x)

Now, as we progress down our input list, each subsequent point has either equal or smaller x coordinates. Therefore, the only way that it could possibly be a Pareto point is if it has a larger y coordinate.

Therefore, we save the value of the largest coordinate we have yet encountered, and a Pareto point lower in the list must have a larger coordinate than our saved value.

```
Data: list of points (x_n, y_n)
sort input points (largest first) by x coordinate, then tiebreak by y
initialize pareto points array with first point from the sorted list
set Y to y coordinate of that point
for point in sorted input do
    if y coordinate of point > Y then
        add point to pareto points array
        update Y to the y coordinate of this point
return pareto points array
```

Problem Solving Notes:

1. *Read and Interpret:* What is this question asking? What makes a point Pareto optimal? For points that don't have the largest x-coordinate, what is the only way it could still be a Pareto optimal point?
2. *Information Needed:* What is a way we can sort the points such that it would be easier for us to identify Pareto optimal points?
3. *Solution Plan:* Sorting by decreasing x-coordinates first allows for easier comparison between the y-coordinates. The x's are already sorted among points, so we can just directly compare y-coordinates now to find Pareto optimal points.

Cutting Ropes

Suppose we are given n ropes of different lengths, and we want to tie these ropes into a single rope. The cost to connect two ropes is equal to sum of their lengths. We want to connect all the ropes with the minimum cost.

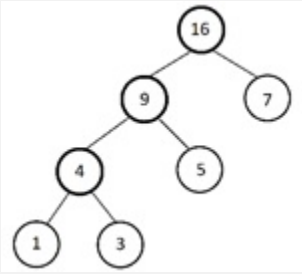
For example, suppose we have 4 ropes of lengths 7, 3, 5, and 1. One (not optimal!) solution would be to combine the 7 and 3 rope for a rope of size 10, then combine this new size 10 rope with the

size 5 rope for a rope of size 15, then combine the rope of size 15 with the rope of size 1 for a final rope of size 16. The total cost would be 10 + 15 + 16 = 41. (Note: the optimal cost for this problem is 29. How might you combine the ropes for that cost?)

Find a greedy algorithm for the minimum cost and prove the correctness of your algorithm.

SOLUTION: We will always combine the smallest ropes available to you until you have one single rope.

Justification:
We will formally prove the correctness of our algorithm by induction. To do this, note that we can write the strings as a graph, where the leaves are the original ropes and every node with two children is a sum of two ropes.



(from the example)

From here, we can use the Lemma from our Huffman analysis:

Lemma 1: *If x and y are ropes with the shortest length, there is an optimal tree where they are siblings.*
Lemma 2: *If we treat the nodes at a given level as leaves, we can still apply Lemma 1.*

Inductive hypothesis. By combining the two smallest ropes available to us at any given point, there is a minimal solution that extends the current solution.

Base case. When we haven't combined any of the ropes, there is clearly a minimal solution that extends the current (empty) solution.

Inductive Step. Suppose that we have combined ropes k times (meaning there are $n - k$ ropes remaining). Lemma 2 tells us that we can basically treat previously combined ropes the same as ropes that haven't been combined, and Lemma 1 tells us that there's an optimal solution where the shortest length ropes are 'siblings' to a parent node that's the sum of them – in other words, there's an optimal solution where the smallest ropes available are tied together.

Conclusion. By the nth step, we have not ruled out the optimal solution. Therefore, the solution we chose is optimal.

Problem Solving Notes:

1. *Read and Interpret:* The problem says the cost of connecting two ropes is the sum of their lengths. Therefore, the cost of combining X ropes is the sum of X-1 "sub-lengths". How can we keep this sum at a minimum?
2. *Information Needed:* What is the minimum cost when there's only one rope? When there are two? When there are three?
3. *Solution Plan:* Through induction, start off with the base case. How would we go about proving the inductive step?

Mice to Holes

There are n mice and n holes along a line. Each hole can accommodate only 1 mouse. A mouse can stay at his position, move one step right from x to $x+1$, or move one step left from x to $x-1$.

Any of these moves consumes 1 minute. Mice can move simultaneously. Assign mice to holes such that the time it takes for the last mouse to get to a hole is minimized, and return the amount of time it takes for that last mouse to get to its hole.

Example:
Mice positions: 4 -4 2
Hole positions: 4 0 5
Best case: the last mouse gets to its hole in 4 minutes $\{4 \to 4, -4 \to 0, 2 \to 5\}$ and $\{4 \to 5, -4 \to 0, 2 \to 4\}$ are both possible solutions

SOLUTION: Sort the mice locations and the hole locations. For $0 \leq i < n$, have the ith mouse go to the ith hole. The maximum distance will be the max distance between each mouse and its corresponding hole.

Justification: We'll first prove a lemma that will be useful in our induction proof later.

Lemma: Suppose $i_1 < i_2$ $j_1 < j_2$ and $dist(x, y) = |x - y|$. Then,

$$\max\{dist(i_1, j_1), dist(i_2, j_2)\} \leq \max\{dist(i_1, j_2), dist(i_2, j_1)\}$$

Without loss of generality, let's say $i_1 \leq j_1$. Our cases are then:

- $i_1 \leq i_2 \leq j_1 \leq j_2$,
- $i_1 \leq j_1 \leq i_2 \leq j_2$,
- $i_1 \leq j_1 \leq j_2 \leq i_2$

In any of these cases, the lemma holds. Now we proceed with induction.

Inductive hypothesis. By sending the i^{th} (sorted) mouse to the i^{th} (sorted) hole, there is a solution with minimal time that extends the current solution.

Base case. If we haven't sent any mice to any holes, we haven't eliminated the ideal solution.

Inductive Step. Suppose that we have sent the first $k-1$ sorted mice to the first $k-1$ sorted holes. Now suppose there is an optimal solution where the k^{th} mouse is sent to the p_0^{th} hole, where $k < p_0 < n$. In this case, the p_0^{th} mouse is sent to the p_1^{th} hole, and so on until the p_d^{th} (for some d) mouse is sent to the k^{th} hole.

We could then swap the p_0^{th} hole with the k^{th} hole; we know by our lemma that the result will not be worse than the optimal solution. Therefore, by sending the k^{th} mouse to the k^{th} hole, we have not eliminated an optimal solution.

Conclusion. By the n^{th} step, we have not ruled out the optimal solution. Therefore, the solution we chose is optimal.

Problem Solving Notes:

1. *Read and Interpret:* We're asking for the time it takes for all the mice to get to their corresponding holes. Since mice can move simultaneously, we just need to find the time it takes for the last mouse to get to its hole. How could we optimize the dataset that we're working with so that each mice goes to its optimal hole?
2. *Information Needed:* What is the maximum distance between each mouse and its corresponding hole? How do we make the inductive step?
3. *Solution Plan:* Once you have come up with how to sort the mice and holes and understood why the lemma applies, you just need to write up an inductive proof!

MST With Leaf Requirements

We are given an undirected weighted graph $G = (V, E)$ and a set $U \subset V$. Describe an algorithm to find a minimum spanning tree such that all nodes in U are leaf nodes. (The result may not be an MST of the original graph G.)

SOLUTION: Let $T = V \setminus U$ be the set of nodes we *don't* require to be leaves, and let $D \subset E$ be the edges between nodes in T. Create an MST on (T, D) using, say, Prim's algorithm. Then add nodes in U to this tree by taking the lightest edge from a node $u \in U$ to any node $t \in T$.

Lemma: *Any solution must have an MST on T as a sub-graph.*
Proof. Let S' be some optimal MST satisfying the leaf requirements. Since each leaf by definition has degree 1, removing the leaf nodes of U does not affect the connectedness of the remaining

nodes. Once we remove from S' all the leaf nodes of U, we are left with a spanning tree over the nodes of T. Now, suppose for the sake of contradiction that this spanning tree was not a MST over T. Then taking the MST over T in conjunction with the leaf edges we just removed would yield a lower cost spanning tree than S', which is a contradiction. Therefore, the MST on T must be contained in S'.

Therefore, our solution gives us a minimum-weight solution because otherwise there would be a lower-weight solution to the original problem.

Problem Solving Notes:

1. *Read and Interpret:* The problem is asking us to find a MST that has a subset of the nodes be the leaf nodes (end nodes).
2. *Information Needed:* What must be true in order for a node in U to be a leaf node? How can we build out the properties of a MST while maintaining leaf nodes? What algorithm could we use?
3. *Solution Plan:* Leaf nodes mean that the leaf cannot be connected to more than one element– a graph cannot be built with only leaf nodes. Therefore, we must make use of the set of nodes that we don't require to be leaves, or T, as the base for building the MST.

Roads and Airports

Given a set of n cities, we would like to build a transportation system such that there is some path from any city i to any other city j. There are two ways to travel: by driving or by flying. Initially all of the cities are disconnected. It costs r_{ij} to build a road between city i and city j. It costs a_i to build an airport in city i. For any two cities i and j, we can fly directly from i to j if there is an airport in both cities.

Give an efficient algorithm for determining which roads and airports to build to minimize the cost of connecting the cities.

SOLUTION: To find the roads and airports to build, we first note that there are two cases: either we do not build any airports or we build at least one airport.

To consider the case where we do not build any airports, we construct an undirected graph where the cities are the nodes and the roads are the edges with weights corresponding to the cost of building that road. We then construct the MST of this graph. This gives us the minimum construction cost using no airports. (If we constructed a non-tree connected graph, we could always remove a road to decrease cost without disconnecting the graph, so the optimal solution must be a tree.)

Then we consider the case where we choose to build at least one airport. To model this, we construct a slightly different graph. We start with the same graph from the previous case: an undirected graph where the cities are the nodes and the roads are the edges with weights corresponding to the cost of building that road. We then add another node to the graph, representing the air. Call this node a. We add an undirected edge between every city i and a with weight a_i. We then construct the MST of this graph.

To find the overall minimum cost set of roads and airports to build, we use either the MST from the first case or the MST from the second case, whichever has lower total cost. For every edge in the MST between two cities i and j, we build road between i and j, **or** for every edge between a city i and a the airport node, we build an airport in city i.

Problem Solving Notes:

1. *Read and Interpret:* The question is asking for an efficient algorithm to minimize the cost of connecting the cities. Since the cost of building a road and the cost of building an airport are variables, we should consider both sets of cases.
2. *Information Needed:* How can we find the minimum cost through only building roads? Through building at least one airport?
3. *Solution Plan:* Go with whichever MST is cheaper in cost!