Greedy Algorithms

A greedy algorithm is one where you take the step that seems the best at the time while executing the algorithm.

Previous Examples: Huffman coding, Minimum Spanning Tree Algorithms

Coin Changing
The goal here is to give change with the minimal number of coins as possible for a certain number of cents using 1 cent, 5 cent, 10 cent, and 25 cent coins.

The greedy algorithm is to keep on giving as many coins of the largest denomination until you the value that remains to be given is less than the value of that denomination. Then you continue to the lower denomination and repeat until you've given out the correct change.

This is the algorithm a cashier typically uses when giving out change. The text proves that this algorithm is optimal for coins of 1, 5 and 10. They use strong induction using base cases of the number of cents being 1, 2, 3, 4, 5, and 10. Another way to prove this algorithm works is as follows: Consider all combinations of giving change, ordered from highest denomination to lowest. Thus, two ways of making change for 25 cents are 1) 10, 10, 1, 1, 1, 1, 1 and 2) 10, 5, 5, 5. The key is that each larger denomination is divisible by each smaller one. Because of this, for all listings, we can always make a mapping for each coin in one list to a coin or set of coins in the other list. For our example, we have:

10 10 11111 11111
10 5 5 5 5

Think about why the divisibility implies that we can make such a mapping.

Now, notice that the greedy algorithm leads to a combination that always maps one coin to one or more coins in other combinations and NEVER maps more than one coin to a single coin in another combination. Thus, the number of coins given by the greedy algorithm is minimal.

This argument doesn't work for any set of coins w/o the divisibility rule. As an example, consider 1, 6 and 10 as denominations. There is no way to match up these two ways of producing 30 cents:

10 10 10
6 6 6 6

In general, we'll run into this problem with matching any denominations where one doesn't divide into the other evenly.
In order to show that our system works with 25 cents, an inductive proof with more cases than the one in the text is necessary. (Basically, even though a 10 doesn't divide into 25, there are no values, multiples of 25, for which it's advantageous to give a set of dimes over a set of quarters.)

*Single Room Scheduling Problem*

Given a single room to schedule, and a list of requests, the goal of this problem is to maximize the total number of events scheduled. Each request simply consists of the group, a start time and an end time during the day.

Here's the greedy solution:

1) Sort the requests by finish time.
2) Go through the requests in order of finish time, scheduling them in the room if the room is unoccupied at its start time.

Now, we will prove that this algorithm does indeed maximize the number of events scheduled using proof by contradiction.

Let $S$ be the schedule determined by the algorithm above. Let $S$ schedule $k$ events. We will assume to the contrary, that there exists a schedule $S'$ that has at least $k+1$ events scheduled.

We know that $S$ finishes its first event at or before $S'$. (This is because $S$ always schedules the first event to finish. $S'$ can either schedule that one, or another one that ends later.) Thus, initially, $S$ is at or ahead of $S'$ since it has finished as many or more tasks than $S'$ at that particular moment. (Let this moment be $t_1$. In general, let $t_i$ be the time at which $S$ completes its $i$th scheduled event. Also, let $t_i$ be the time at which $S'$ completes its $i$th scheduled event.)

We know that

1) $t'_1 \geq t_1$
2) $t'_{k+1} < t_{k+1}$ since $S'$ ended up scheduling at least $k+1$ events.

Thus there must exists a minimal value $m$ for which

$t'_m < t_m$ and this value is greater than 1, and at most $k+1$.

(Essentially, $S'$ is at or behind $S$ from the beginning and will catch up and pass $S$ at some point...)

Since $m$ is minimal, we know that

$t'_{m-1} \geq t_{m-1}$. 
But, we know that the mth event schedule by S ends AFTER the mth event scheduled by S’. This contradicts the nature of the algorithm used to construct S. Since \( t_{m-1}' \geq t_{m-1} \), we know that S will pick the first event to finish that starts after time \( t_{m-1} \). BUT, S’ was forced to also pick some event that starts after \( t_{m-1} \). Since S picks the fastest finishing event, it's impossible for this choice to end AFTER S' choice, which is just as restricted. This contradicts our deduction that \( t_m' < t_m \). Thus, it must be the case that our initial assumption is wrong, proving S to be optimal.

**Multiple Room Scheduling (in text)**

Given a set of requests with start and end times, the goal here is to schedule all events using the minimal number of rooms. Once again, a greedy algorithm will suffice:

1) Sort all the requests by start time.
2) Schedule each event in any available empty room. If no room is available, schedule the event in a new room.

We can also prove that this is optimal as follows:

Let k be the number of rooms this algorithm uses for scheduling. When the kth room is scheduled, it MUST have been the case that all k-1 rooms before it were in use. At the exact point in time that the k room gets scheduled, we have k simultaneously running events. It's impossible for any schedule to handle this type of situation with less than k rooms. Thus, the given algorithm minimizes the total number of rooms used.

**Fractional Knapsack Problem**

Your goal is to maximize the value of a knapsack that can hold at most W units worth of goods from a list of items \( I_1, I_2, \ldots, I_n \). Each item has two attributes:

1) A value/unit; let this be \( v_i \) for item \( I_i \).
2) Weight available; let this be \( w_i \) for item \( I_i \).

The algorithm is as follows:

1) Sort the items by value/unit.
2) Take as much as you can of the most expensive item left, moving down the sorted list. You may end up taking a fractional portion of the "last" item you take.

Consider the following example:

There are 4 lbs. of \( I_1 \) available with a value of $50/lb.
There are 40 lbs. of \( I_2 \) available with a value of $30/lb.
There are 25 lbs. of \( I_3 \) available with a value of $40/lb.

The knapsack holds 50 lbs.

You will do the following:
1) Take 4 lbs of I$_1$.
2) Take 25 lbs. of I$_3$.
3) Take 21 lbs. of I$_2$.

Value of knapsack = 4*50 + 25*40 + 21*30 = $1830.$

Why is this maximal? Because if we were to exchange any good from the knapsack with what was left over, it is IMPOSSIBLE to make an exchange of equal weight such that the knapsack gains value. The reason for this is that all the items left have a value/lb. that is less than or equal to the value/lb. of ALL the material currently in the knapsack. At best, the trade would leave the value of the knapsack unchanged. Thus, this algorithm produces the maximal valued knapsack.

**Exact Change Problem**

Given a set of coin values, determine the minimum value n, for which there is no way to make change for n cents.

At first this problem looks like some harder version of the subset sum problem. But, the key observation is this: given some set of coins \{a_0, a_1, \ldots, a_{k-1}\} and their sum S, one of two possibilities occurs: (a) some value less than S is not obtainable, (b) all values up to S are obtainable and the answer to the query for the set is S+1.

Consider situation (b) and adding a new coin to this set so that S+1 can be obtained. Any coin will do, so long as its value is less than or equal to S+1.

In situation (a), let X < S be the smallest unobtainable value. It must be the case that some items of the set are less than X and other items of the set are greater than X. We posit that the sum of the items less than X is precisely X – 1. First of all, note that all of the values greater than X are of no help, so we can ignore these values. Since X is the FIRST value that can’t be obtained, some subset of these coins adds to every value from 1 to X – 1. Assume the opposite, that the sum of the items is greater than X – 1. If this is the case, then if we list all items, there should be some set of items we subtract out to obtain X – 1 exactly. But, in this case, consider one of the items subtracted out, Y. We know that Y < X. We also know that each sum up to X – 1 can be obtained exactly without using this coin, in particular, X – Y can be obtained from these coins. This means we can take the subset of coins that adds to X – Y and add the coin with value Y to it to obtain a sum of X, which contradicts our original premise that X was not obtainable. It follows that our assumption that the sum of all the coins less than X must equal X – 1.

With these observations, the algorithm becomes more clear: sort the coins in numerical order, from smallest to largest. Keep track of your running sum of the coins, setting this to 0, initially. Loop through the coins. The current coin must be less than or equal to 1 plus the running sum. If it is not, then the answer to the query is the running sum plus 1. If it is, then add this coin to the running sum and continue.
Minimum Spanning Trees
In this lecture we will explore the problem of finding a minimum spanning tree in an undirected weighted graph (if one exists). First let's define a tree, a spanning tree, and a minimum spanning tree:

tree: A connected graph without cycles. (A cycle is a path that starts and ends at the same vertex.)

spanning tree: a subtree of a graph that includes each vertex of the graph. A subtree of a given graph as a subset of the components of that given graph. (Naturally, these components must form a graph as well. Thus, if your subgraph can't just have vertices A and B, but contain an edge connecting vertices B and C.)

Minimum spanning tree: This is only defined for weighted graphs. This is the spanning tree of a given graph whose sum of edge weights is minimum, compared to all other spanning trees.

Crucial Fact about Minimum Spanning Trees
Let G be a graph with vertices in the set V partitioned into two sets V₁ and V₂. Then the minimum weight edge, e, that connects a vertex from V₁ to V₂ is part of a minimum spanning tree of G.

Proof: Consider a MST T of G that does NOT contain the minimum weight edge e. This MUST have at least one edge in between a vertex from V₁ to V₂. (Otherwise, no vertices between those two sets would be connected.) Let G contain edge f that connects V₁ to V₂. Now, add in edge e to T. This creates a cycle. In particular, there was already one path from every vertex in V₁ to V₂ and with the addition of e, there are two. Thus, we can form a cycle involving both e and f. Now, imagine removing f from this cycle. This new graph, T' is also a spanning tree, but it's total weight is less than or equal to T because we replaced e with f, and e was the minimum weight edge.

Each of the algorithms we will present works because of this theorem above.

Each of these algorithms is greedy as well, because we make the "greedy" choice in selecting an edge for our MST before considering all edges.

Kruskal's Algorithm

The algorithm is executed as follows:

Let V = ∅
For i=1 to n-1, (where there are n vertices in a graph)
    V = V ∪ e, where e is the edge with the minimum edge weight not already in V, and that does NOT form a cycle when added to V.

Return V
Basically, you build the MST of the graph by continually adding in the smallest weighted edge into the MST that doesn't form a cycle. When you are done, you'll have an MST. You HAVE to make sure you never add an edge the forms a cycle and that you always add the minimum of ALL the edges left that don't.

The reason this works is that each added edge is connecting between two sets of vertices, and since we select the edges in order by weight, we are always selecting the minimum edge weight that connects the two sets of vertices.

In order to do cycle detection here, one idea is to keep track of all the separate clusters of vertices. Initially, each vertex is in its own cluster. For each edge added, you are merging two clusters together. Indicate this by changing a variable that stores the cluster ID values of a vertex to be the same as every other vertex in the cluster. An edge can NOT be added in between two vertices within the same cluster.

*Prim's Algorithm*

This is quite similar to Kruskal's with one big difference:

The tree that you are "growing" ALWAYS stays connected. Whereas in Kruskal's you could add an edge to your growing tree that wasn't connected to the rest of it, here you can NOT do it.

Here is the algorithm:

1) Set \( S = \emptyset \).
2) Pick any vertex in the graph.
3) Add the minimum edge incident to that vertex to \( S \).
4) Continue to add edges into \( S \) (n-2 more times) using the following rule:

Add the minimum edge weight to \( S \) that is incident to \( S \) but that doesn't form a cycle when added to \( S \).

Once again, this works directly because of the theorem discussed before. In particular, the set you are growing is the partition of vertices and each edge you add is the smallest edge connecting that set to its complement.

For cycle detection, note that at each iteration, you must add exactly one vertex into the subgraph represented by the edges in \( S \). (You can think of "growing" the tree as successively adding vertices that are connected instead of adding edges.)