# 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 | 1 1 1 1 1 | 1 1 1 1 1 |
| 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 | 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 ith scheduled event. Also, let $t'_i$ be the time at which S' completes its ith 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 $I_1$, $I_2$, ... $I_n$. 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) Tale 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.

*Containers Problem*

Imagine a sequence of containers coming to a dock. When the containers come to the dock, they must immediately be placed on top of a stack. Each container has a letter indicating its size, A being the smallest, Z being the largest. When you place a container on a stack, you can only place it on a container of the same size or larger. Thus, the container with size D can be placed on top of a container with size D or E, but not one of size C. Alternatively, a container can be placed on the ground to create a new stack.

Given a sequence of containers that come to dock, write a program to calculate the minimum number of different stacks that have to be created.

Here is an example input:

ABBEFEDBEDFBA

We have to make 4 stacks for the first 5:

```
      B
A     B     E     F
```

Now the key decision is, "Where do we put the next item, E? Should we put it on the stack with the top E or top F?"

The key to the correct algorithm is realizing that it's better to put the E on the E and not the F. If we put it on the F we get:

```
      B           E
A     B     E     F
```

then the problem is, if we get an F in the future, then we would be forced to make a new stack. But, if we put it on E, we could handle an F:

```
      B     E
A     B     E     F
```

Thus, the algorithm is as follows:

Whenever you get a new container, place it on the stack whose top is the least, of the stacks whose value is greater than or equal to the new container value. Then, just count up how many stacks you have at the end!

*Last Alphabetic Subsequence (Top Coder SRM 518 D1 250)*

Problem: Given an alphabetic string, find the subsequence that comes last alphabetically in the string.

Any subsequence that starts at the "latest possible letter" is better than other competing subsequences. Thus, our first letter should be the alphabetically last letter that occurs in the string. As we are iterating through the letters, replace our "current answer" completely, if we see a letter that comes after the first letter in our current answer.

Adding a letter to any string makes it alphabetically after its prefix, so by default, we would always want to add a new letter to a non-empty string.

If we have some string "geb", for example, if the next letter is "f", then we see that "gf" is better than "gef" or "gebf".

Once we make this observation, then our algorithm is streamlined as follows:

1) Keep a stack of letters – initialize it as empty.

Repeat the following steps as you read through each letter in the input string:
2) When reading in a new letter, pop all letters off the stack that come strictly before the current letter, alphabetically.
3) Push the current letter on the stack.

When we finish, the subsequence we desire can be read off the stack from the bottom to the top. (In code, if we were using a stack, we'd pop each item off sequentially, storing them from back to front.)

The key in developing this algorithm is to note that given any previous subsequence that is the 'best", we develop our new best by "building off" the old best.