

Greedy Algorithms

A greedy algorithm is one where you take the step that seems the best at the time while executing the algorithm. Some of the graph algorithms we've already seen, such as Dijkstra's and Kruskal's are considered greedy because they don't consider all possibilities, but still obtain the optimal answer, because of the nature of the problem and algorithm. (Kruskal's works because it's always safe to put an edge in an MST that's the minimal edge that connects two separate forests. Dijkstra's works because the no negative edge weight assumption infers that it's impossible to go to a vertex outside of S and return back to S with a negative sum of edges.)

Here we'll look at a few problems that have greedy solutions. The hardest part of these problems is proving that some greedy algorithm correctly solves the problem. There are many problems for which no greedy algorithm correctly solves it. There are other problems that do have greedy solutions, but a majority of greedy algorithms one could devise are incorrect. One should always be a bit skeptical when evaluating a potential greedy algorithm. Before committing to one, one should prove the algorithm's correctness.

Example #1: 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.)

Example #2: 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 exist 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 m th event scheduled by S ends AFTER the m th 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.

Example #3: Multiple Room Scheduling

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 k th 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.

Example #4: 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, \dots, 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.

Example #5: 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.