Lecture 17 - Floyd-Warshall

Eric A. Autry

Last time: Bellman-Ford and Arbitrage

This time: Floyd-Warshall

Next time: Greed

HW 6 due today.

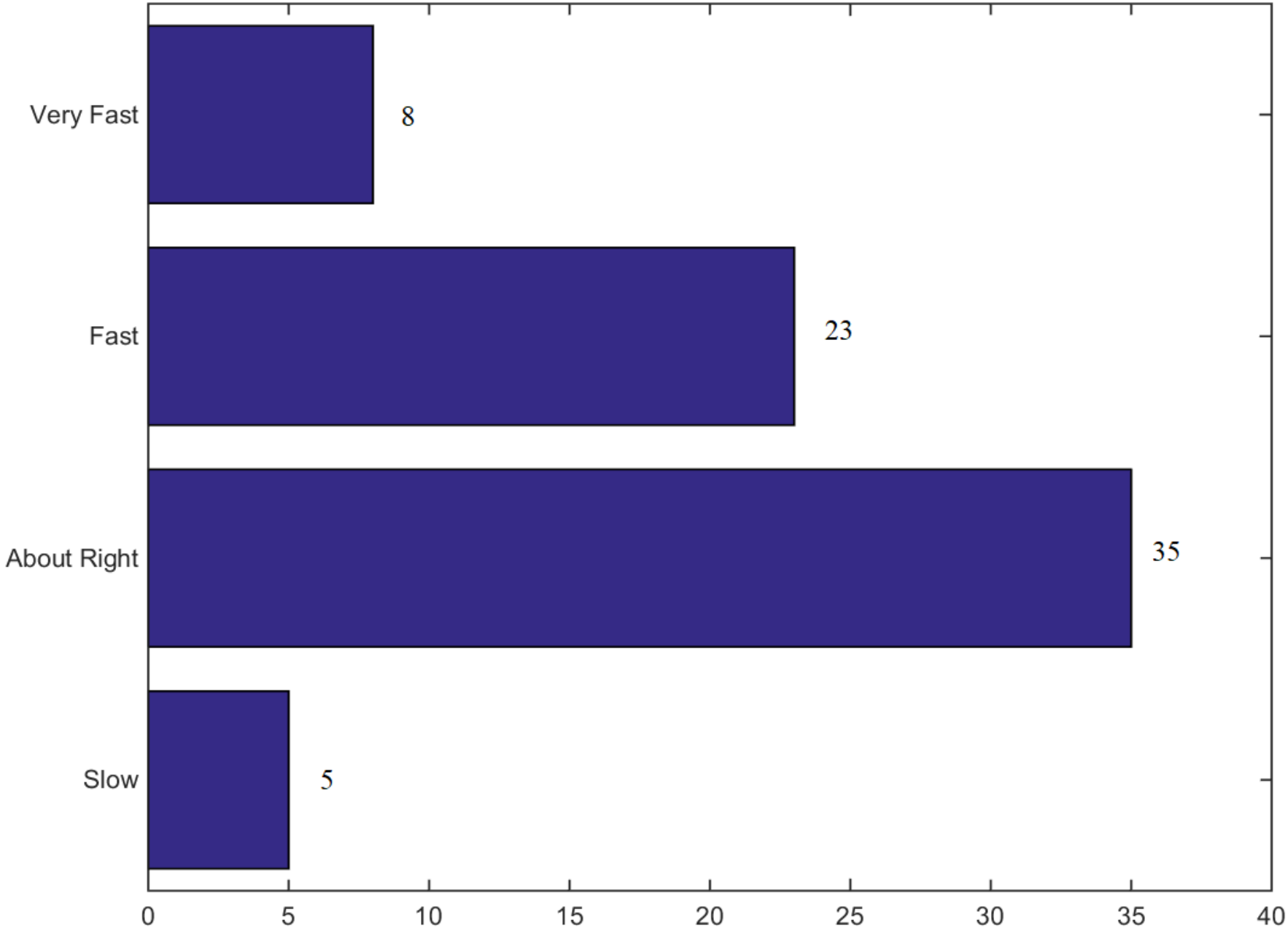
Project 2 due Friday the 16th.

Survey Results

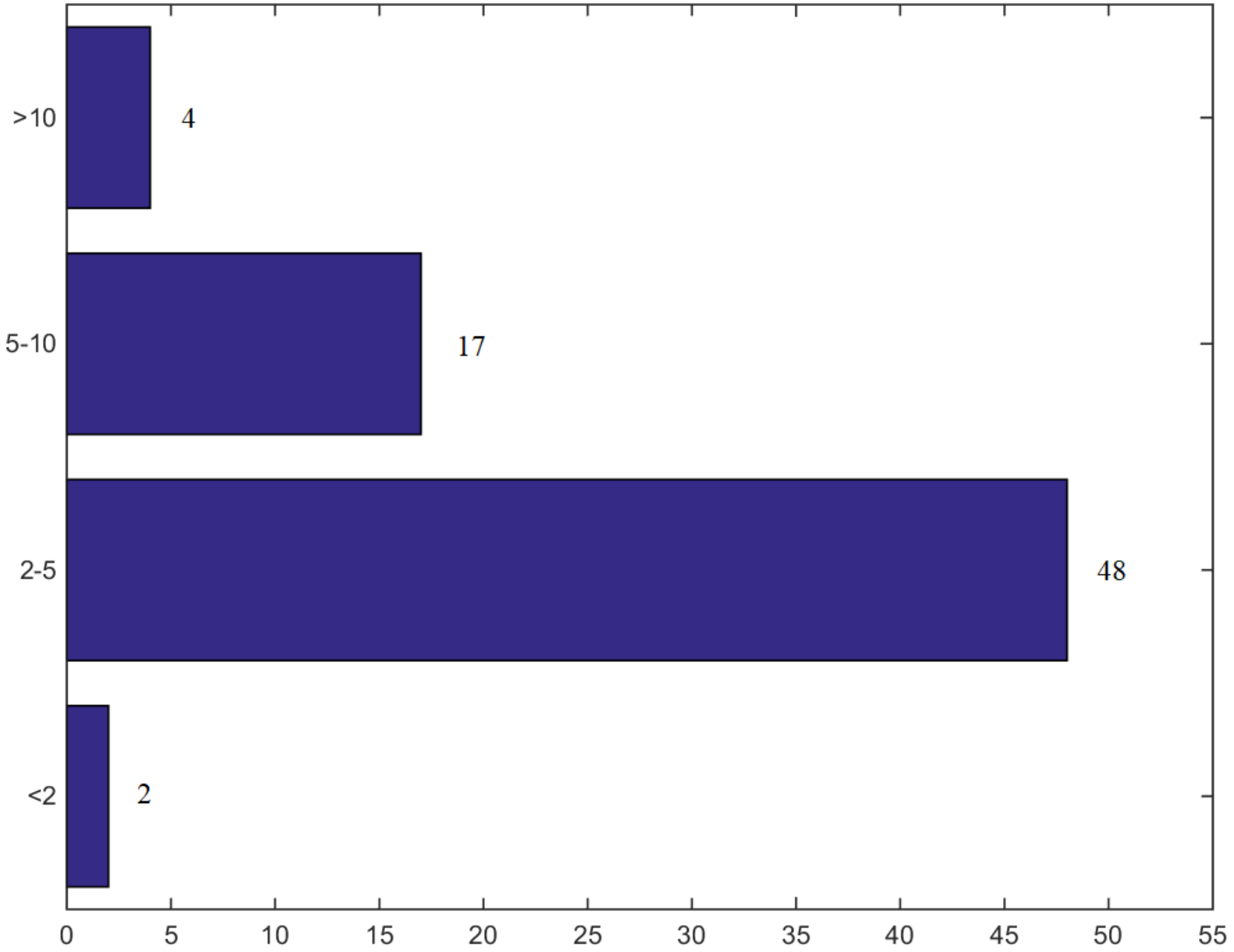
Thank you to everyone who completed the survey.

We had 71/114 students respond, a 62% response rate!

Survey Results - Pacing



Survey Results - Hours spent on HW



Survey Results - Main Comments

Top 4 Comments:

- Office Hours useful but too busy.
- Have trouble understanding concepts in class the 1st time.
- Have trouble understanding the HW the 1st time.
- Too much Automata Theory.

There was one other major piece of information on the survey: 4/71 students have not used a programming language before.

Survey Results - My Responses

- Next year, I will balance the Automata Theory part of the course better.
- The course is going to slow down somewhat so we can take a little more time working through concepts.
  - This does mean that we will not be able to cover the more advanced topics at the end of the course.
  - We will still cover the remaining core topics of algorithms: Greed, MST, Amortized Analysis, and NP-Completeness.
  - It is more important to understand these core concepts than to reach the advanced topics.
- The projects 3 and 4 have already been altered to accommodate this slower schedule.

Survey Results - My Responses

To help with office hours and understanding concepts:

- First, it is important to remember that it is completely fine if you don't understand the first time. These are complicated and new concepts that take time to understand.
- The homework is intended to help with understanding: by making you work through a problem, the homework requires you to understand that concept.
- In addition to slowing down the pacing, I will try to include more worked examples in class (like with Bellman-Ford).
- We will also try Think-Pair-Share in a few places, where I will give you a (short) problem to discuss in groups. I will talk about this more when it comes up next week.

Bellman-Ford Implementation

```
def bellmanFord(graph, start):
    # Set initial dist and prev.
    for vertex in graph:
        vertex.dist = ∞
        vertex.prev = null
    start.dist = 0

    # Iterate |V| − 1 times.
    for iter in range(0, |V| − 1):
        # Look at each vertex.
        for u in graph:
            # Check each neighbor of u.
            # Update predictions and previous vertex.
            for neigh of u:
                # Only update if the new value is better!
                if neigh.dist > u.dist + length(u, neigh):
                    neigh.dist = u.dist + length(u, neigh)
                    neigh.prev = u
```

How to find `length(u, neigh)`: if A is the adjacency matrix, then

```
length(u, neigh) = A[u.rank][neigh.rank]
```

Negative Cost Cycles

Shortest path algorithms are invalid when negative cost cycles exist because there cannot be shortest paths!

Bellman-Ford can actually detect these negative cost cycles:

- Run for 1 extra iteration, if any values change, there is a negative cost cycle.
- Why?
  - At the end of Bellman-Ford, all vertices have the correct shortest path distances.
  - After $|V| - 1$ iterations, even the longest possible path has been accounted for.
  - At this point, a distance could only change on a new update if a cycle is involved (i.e., the path had more than $|V|$ vertices).

Negative Cost Cycles

Application: finance.

Let's consider currency exchange rates.

Say that:

- 1 USD buys 0.82 Euro,
- 1 Euro buys 129.7 Yen,
- 1 Yen buys 12 Turkish Lira,
- 1 Turkish Lira buys 0.0008 USD

Then 1 USD buys:

$$0.82 \cdot 129.7 \cdot 12 \cdot 0.0008 \approx 1.02 USD$$

We made a 2% profit!

This is called arbitrage.

Shortest Paths between All Vertices

What do I do if I want to find the shortest paths between all pairs of vertices?

I could run Bellman-Ford $|V|$ times, taking each vertex as the starting vertex.

This would have a runtime of $O(|V|^2 \cdot |E|)$, which could be $O(n^4)$ worst case.

It turns out we can do better.

When we re-run Bellman-Ford, we are throwing away a lot of information. It seems like there should be a way to store that info...

Idea: consider a use-it-or-lose-it approach and use dynamic programming!

This will give us Floyd-Warshall.

Use-it-or-Lose-it Approach

Idea: consider the shortest path from u to v:

$$u \to w_1 \to w_2 \to \cdots \to w_k \to v$$

We can see that there might be intermediate vertices on this path: $w_1, w_2, \ldots, w_k$

We will build a use-it-or-lose-it approach by considering these intermediate vertices.

Use-it-or-Lose-it Approach

Base Case: what if there are no intermediate vertices?

- If there is an edge from u to v, then `dist(u,v) = length(u,v)`.
- If there is no edge from u to v, then `dist(u,v) =` ∞

Now the use-it-or-lose-it: consider the highest ranked vertex k.

Is it on the path between u and v?

- Use-it: k is on the path, so need the `dist` from u to k plus the `dist` from k to v. Since k has been used, remove it from the list of vertices to consider.
- Lose-it: k is not on the path, so remove k from the list of vertices to consider.

Use-it-or-Lose-it Approach

Note: this problem is 3 dimensional.

We need to keep track of:

- The start vertex.
- The end vertex.
- The list of possible intermediate vertices.

We will store the results in an $n \times n \times n$ DP table:

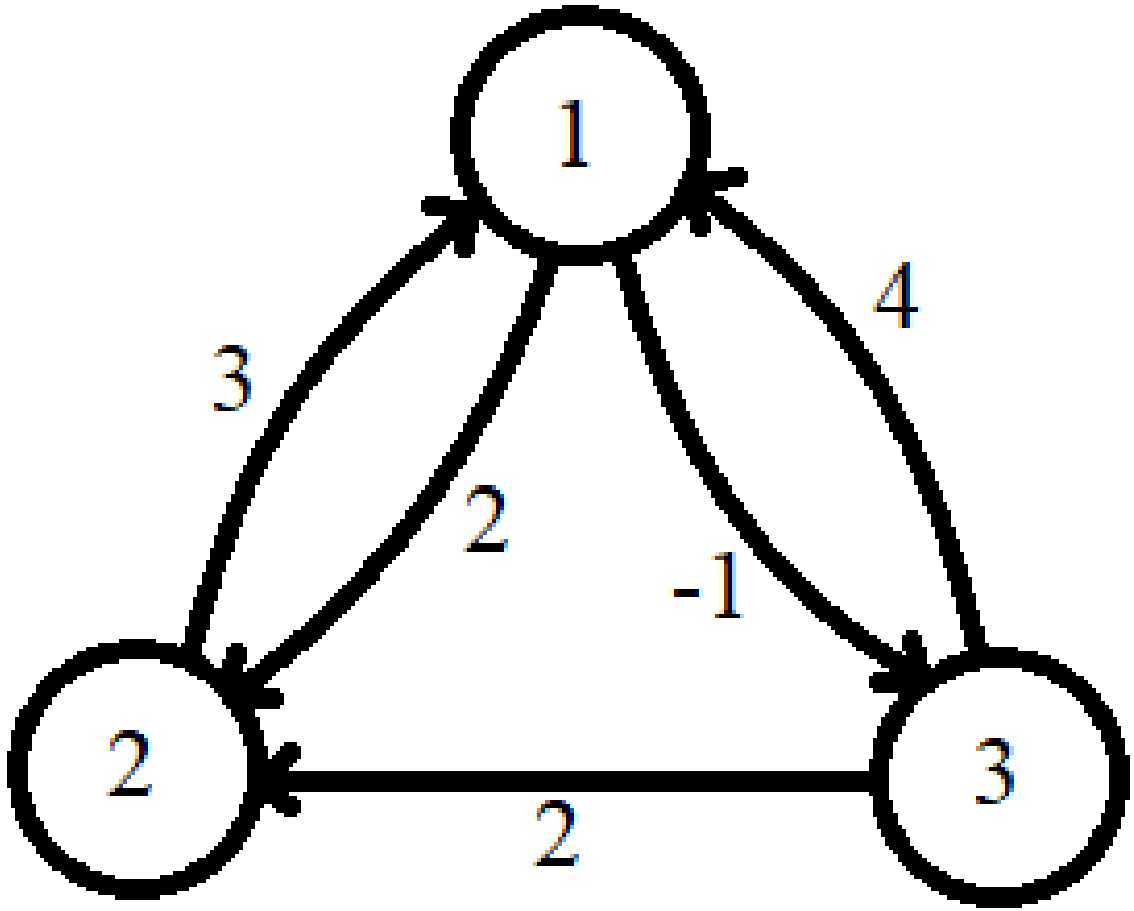
`dist[i,j,k]`

- i is the start.
- j is the end.
- The possible intermediate vertices are ranked $1, 2, \ldots, k$.

Floyd-Warshall

```
def Floyd-Warshall(length):
# Input edge lengths as the adjacency matrix.
# The length must be ∞ if no edge is present.

    # Base Case: O(n^2)
    # No intermediate vertices, just use single edges (k=0).
    for i = 1 to n:
        for j = 1 to n:
            dist[i][j][0] = length[i][j]

    # Now fill in the rest of the 3D table: O(n^3).
    for k = 1 to n:
        for i = 1 to n:
            for j = 1 to n:
                # Use-it: k is on the path.
                # So dist is (i -> k) + (k -> j).
                useit = dist[i][k][k-1] + dist[k][j][k-1]

                # Lose-it: k is not on the path, skip it.
                loseit = dist[i][j][k-1]

                dist[i][j][k] = min(useit, loseit)
    return dist
```

Floyd-Warshall - Worked Example



Greedy Algorithms

New algorithmic tactic: greed.

Idea: when making successive choices, always take the greediest option.

- Usually the greedy approach is easy to develop.
- Usually it seems reasonable and intuitive.
- Very often, however, it is NOT correct.
- So it is important to prove correctness, i.e., prove that greed gives an **optimal** solution.

Greedy Coins

Recall the change problem: What is the fewest number of coins required to make change for amount A?

We now add a special rule: denominations of coins are powers of some fixed amount b:

$$1, b, b^2, b^3, \ldots, b^k.$$

Idea: start with the largest coin and use as many of that coin as possible, then repeat with the next largest coin, etc.

Where was the greed?

- 'largest coin'
- 'use as many as possible'

How do we prove that this solution is optimal?

Greed is Optimal

Observation #1:

It is not optimal to use more than $b-1$ of any coin.

- If we use b coins of value b^i, then we have made change for $b \cdot b^i = b^{i+1}$, which is the next largest coin.

Observation #2:

Using at most $b-1$ of each coin, how much can we make?

$$(b-1)\cdot 1 + (b-1)\cdot b + (b-1)\cdot b^2 + \ldots (b-1)\cdot b^k$$

$$= (b-1)\frac{b^{k+1}-1}{b-1} = b^{k+1} - 1,$$

which is one less than the next largest coin.

Greed is Optimal

Idea: compare our greedy solution to some 'optimal' solution, then show that our solution was better (or at least not worse).

Say that the greedy algorithm gives us:

- g_k number of b^k-coins,
- g_{k-1} number of b^{k-1}-coins,
- $\ldots$
- g_1 number of b-coins,
- g_0 number of 1-coins.

Say that the 'optimal' solution gives us:

- p_k number of b^k-coins,
- p_{k-1} number of b^{k-1}-coins,
- $\ldots$
- p_1 number of b-coins,
- p_0 number of 1-coins.

Greed is Optimal

We will assume that this 'optimal' solution is different than the greedy solution.

But that means that at some index i, we have $p_i \neq g_i$.

Let's consider the largest denomination for which this is true, i.e., both solutions have the same number of larger coins than b^i, but a different number of coins of size b^i and smaller.

$$A = g_k b^k + g_{k-1} b^{k-1} + \cdots + g_i b^i + \cdots + g_1 b + g_0$$

$$A = p_k b^k + p_{k-1} b^{k-1} + \cdots + p_i b^i + \cdots + p_1 b + p_0$$

What do we know about the relative sizes of p_i and g_i?

The greedy algorithm always chose to give as many of the largest coins as possible, so it must be that

$$g_i > p_i$$

Greed is Optimal

So we know that:

$$g_i > p_i$$

This means that the 'optimal' solution would have to make up for an extra b^i amount of money using only smaller coins!

That would require more than $b - 1$ number of each smaller coin (from observation #2).

It **cannot** be optimal to choose more than $b - 1$ number of any coin (from observation #1).

So this 'optimal' solution could not have been different than our greedy solution while still being optimal!

The only way it could have been optimal was to have picked the same choices as our greedy algorithm.

Greedy Coins

Ex: say we set $b = 5$, so we have coins worth 1, 5, and 25.

How many coins will it take to give 42 cents in change?

- 1 coin worth 25 cents (17 cents left)
- 3 coins worth 5 cents (2 cents left)
- 2 coins worth 1 cent (0 cents left)

So 6 coins make 42 cents change.