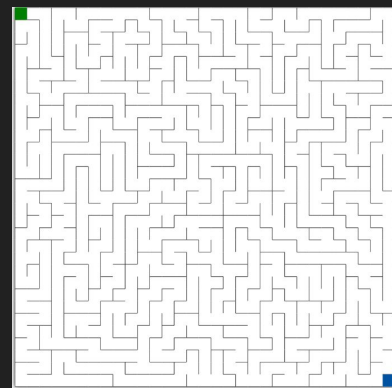
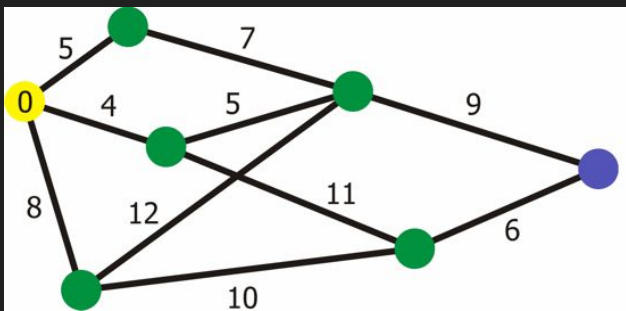


Please do not redistribute these slides
without prior written permission



CS 5008/5009

Data Structures, Algorithms, and Their Applications Within Computer Systems

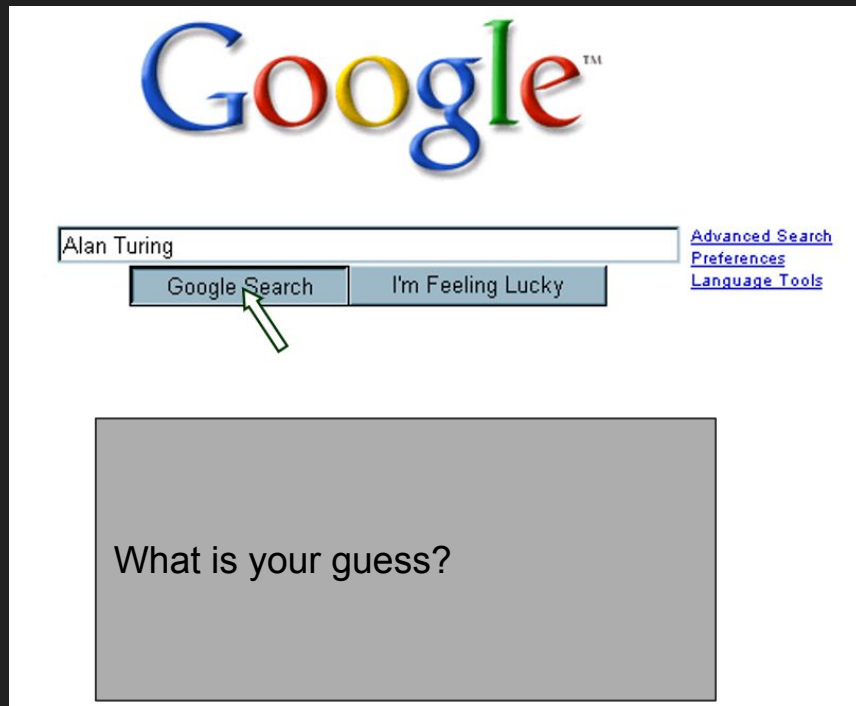


Dr. Mike Shah

	Wait
Sprinkle Cheese on top	Open the oven door
	Eat Pizza!
	Add pepperoni
	Roll out pizza dough
	Remove pizza
	Spread tomato sauce on top of dough
	Close oven door
	Put pizza in the oven


Pre-Class Warmup (1/2)

- Fun fact on how many processors it takes to query your Google search
 - source:
<http://www.cs.cmu.edu/~bryant/presentations/DISC-FCRC07.ppt>
- ^ Think about why this is important about choosing/developing efficient algorithms
 - (Perhaps also for energy conservation!)



Pre-Class Warmup (1/2)

- Fun fact on how many processors it takes to query your Google search
 - source:
<http://www.cs.cmu.edu/~bryant/presentations/DISC-FCRC07.ppt>
- ^ Think about why this is important about choosing/developing efficient algorithms
 - (Perhaps also for energy conservation!)

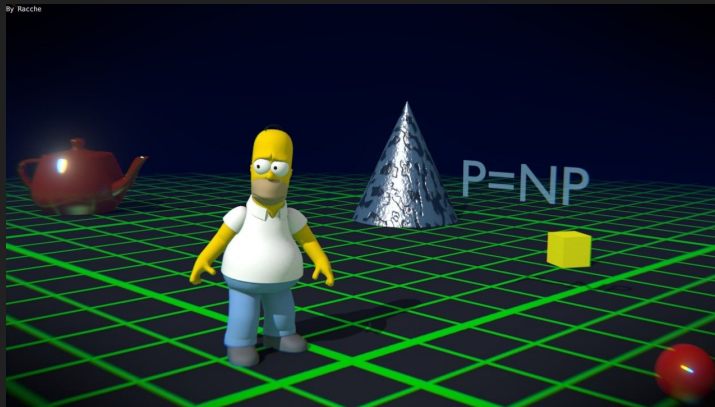
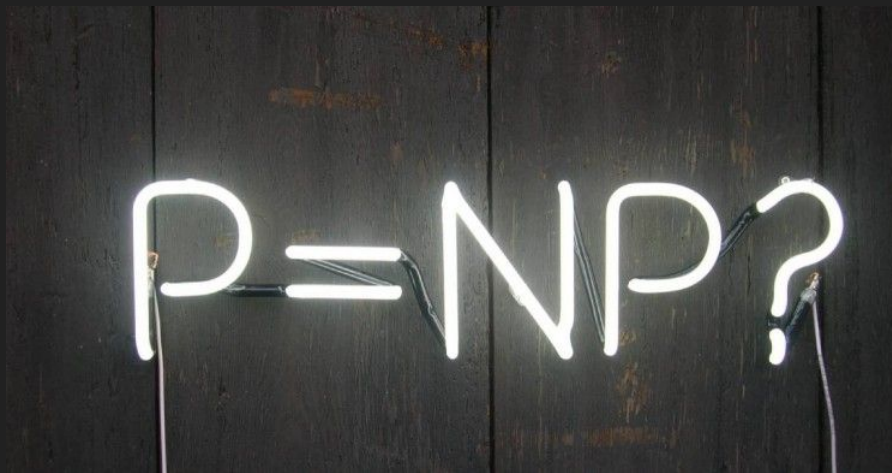


A screenshot of the Google homepage from the early 2000s. The Google logo is at the top in its signature multi-colored font. Below it is a search bar containing the text "Alan Turing". To the right of the search bar are links for "Advanced Search", "Preferences", and "Language Tools". Below the search bar are two buttons: "Google Search" and "I'm Feeling Lucky". A mouse cursor is pointing at the "Google Search" button.

- **200+ processors**
- **200+ terabyte database**
- **10^{10} total clock cycles**
- **0.1 second response time**
- **5¢ average advertising revenue**

Pre-Class Warmup (2/2)

- Does $P=NP$?
- It is one of the fundamental questions in Computer science.
- In short, can we both verify and solve a problem in polynomial time (if indeed $P=NP$)
 - This means we can solve some large problems!
 - (Check out this Simpsons episode after the semester to see the tribute to computer graphics and computer science!)



Famous 3D Halloween episode from
"The Simpsons"



Note to self: Start audio recording of lecture :)
(Someone remind me if I forget!)

Course Logistics

- HW11 extended
 - There is a 'part 2' coming to your repositories
- Our last lab will be out today
 - Make sure you are running your code on the servers whenever possible
 - ssh [username@login.khoury.neu.edu](ssh:username@login.khoury.neu.edu)
- Study Guide for final exam is on the main page for some topics for the exam.
 - Same format as the previous exam
 - It will be released sometime around April 19 and you'll have until the 26th to complete it.

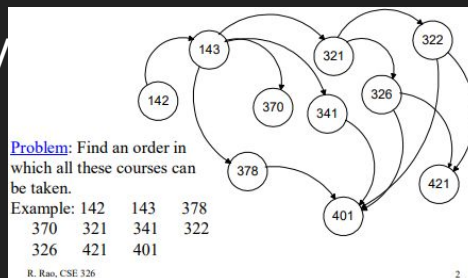
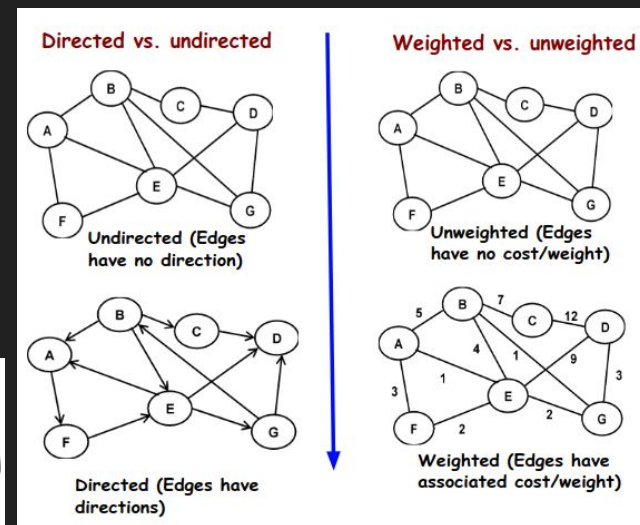
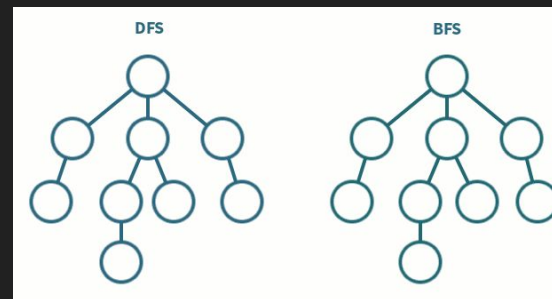
Last Time

- Trees again
 - Topological Sort
- Graphs vs Trees
- Graph Notation
- Representation of graphs
 - Adjacency Matrix
 - Adjacency List
- Breadth-First Search vs DFS
- Small introduction to greedy
- Lab:
 - Topological Sort
 - Adjacency Matrix

Vertices

	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	0	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	1	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Vertices

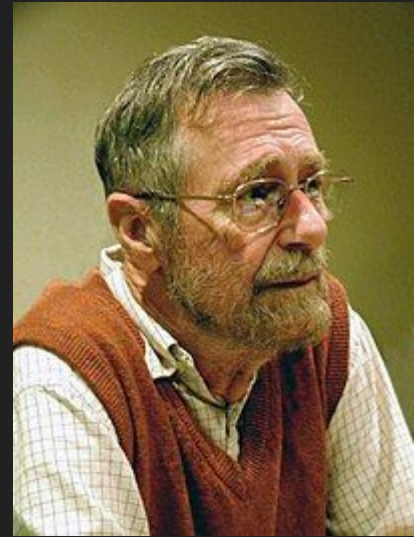


hw structure discussion

live drawing/coding/discussion

- Graphs part 1
 - You can write helper functions in DLL
 - Don't change the function signature in graphs
 - Advice
 - Implement remove node last
- Graphs part 2
 - Reachability
 - print path
 - (First check if it is reachable) then print the path
 - hasCycle
 - Helper functions can be used (e.g. DFS)
 - Can add a 'visited' attribute to the structs.

“Testing shows the
presence, not the absence
of bugs”

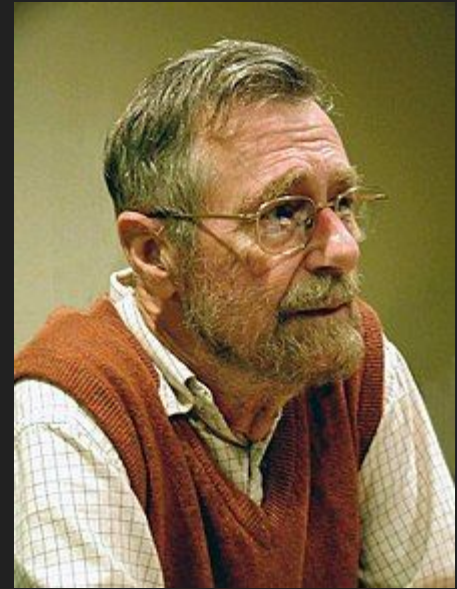


“Testing shows the presence, not the absence of bugs” Let’s write lots of tests then to help debug our programs!

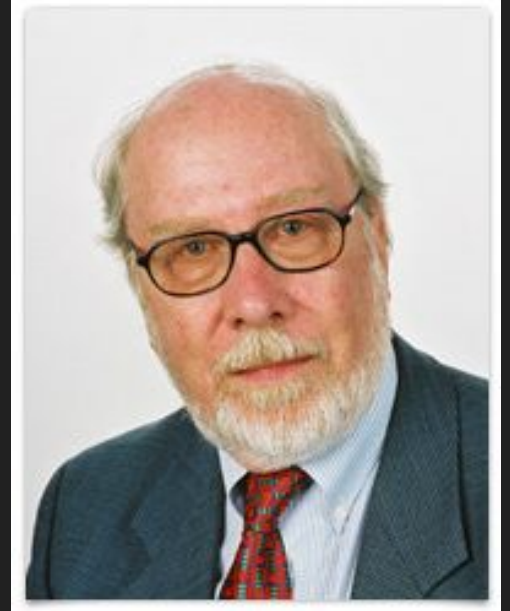


Edsger Dijkstra [[wiki](#)]

- Dutch Systems Scientist
- Known for many contributions to Algorithms and Data Structures
 - Dijkstra's Algorithm
 - Semaphores
 - etc
- Turing Award Winner
- Many famous quotes including: "Program testing can be used to show the presence of bugs, but never to show their absence!"
- We'll encounter Dijkstra's Algorithm soon!!



“A good designer must rely on experience, on precise, logic thinking; and on pedantic exactness. No magic will do.”



“A good designer must rely on experience, on precise, logic thinking; and on pedantic exactness. **No magic will do.**”

“Hey, that’s what I’m always saying”



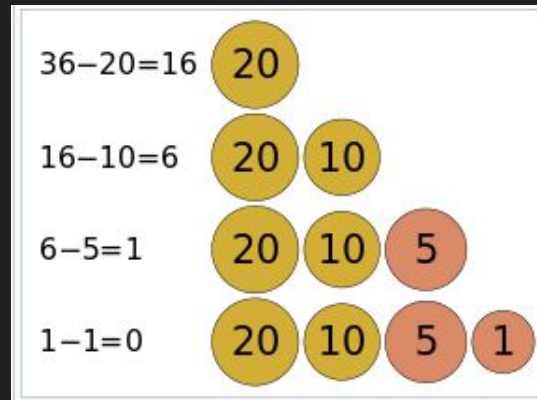
Niklaus Wirth [[wiki](#)] (pronounced 'virt')

- Swiss Computer Scientist
- Turing Award winner (1984) -- primarily for programming language contributions
 - In the 'Algo' family
 - (Yes, there are language that are not 'C' like)
- Wrote a book titled:
 - Algorithms + Data Structures = Programs
 - ^ This is what we are doing, and I thought it would be appropriate to review!



Today

- Greedy Algorithms (Algorithm Strategy)
 - Another search
 - Dijkstra's Shortest Path
- Dynamic Programming
 - Fibonacci

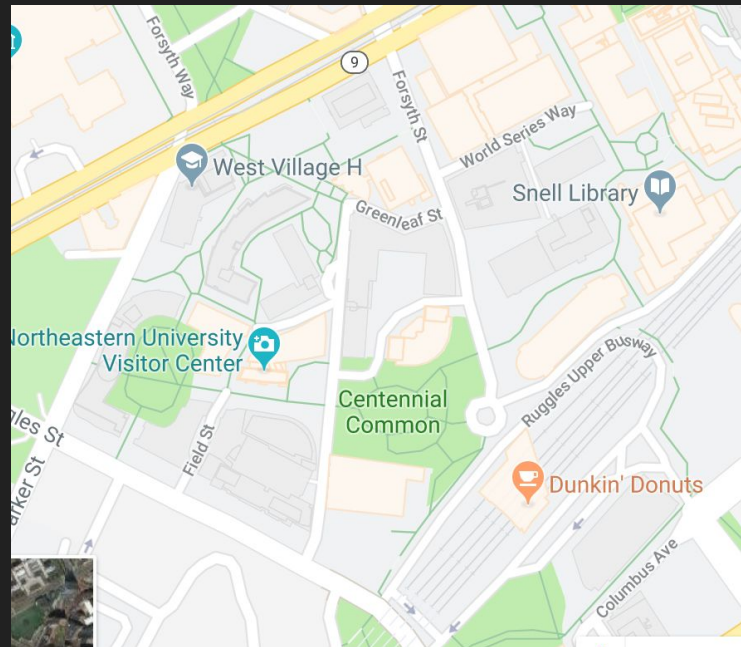


Greedy Algorithms

Dijkstra's Shortest Path

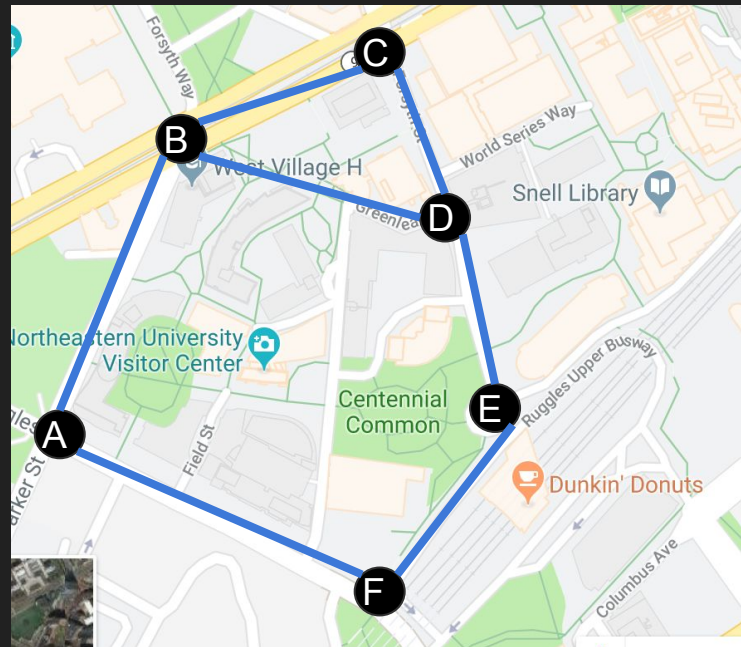
(Remember our Motivating Problem) (1/4)

- You're hired to work on the Google maps team
- They want your help searching graphs
 - (i.e. a map has roads as edges and locations as nodes)
- They want you to find routes that find:
 - the shortest amount of 'transfers' (Solved with BFS)
 - 'shortest path' to different destinations.



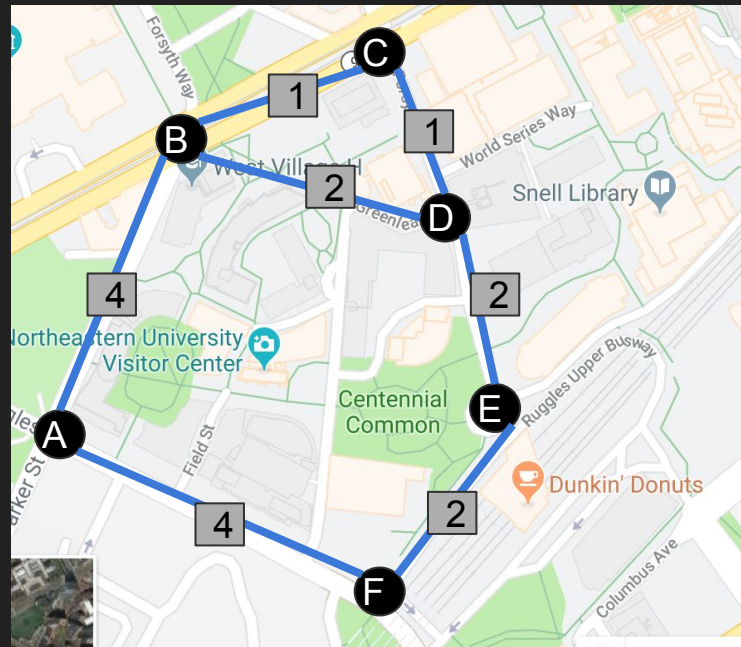
(Remember our Motivating Problem) (2/4)

- We helped solve part of the problem,
 - Given
 - a map(i.e. graph) with no edge weights
 - Problem:
 - Find the shortest amount of transfers from a starting destination (source) to all other nodes (destinations)
 - Solution
 - Use Breadth-First Search (BFS) to determine the shortest path (assuming every edge weighted equally) to any of the nodes from a starting point.



(Remember our Motivating Problem) (3/4)

- New Problem!
 - Given: A graph with positive edge weights
 - (i.e. weights measure distance in this case), we need to use a different algorithm.
 - Problem:
 - Find the shortest path from a source to a destination
 - Solution: ???



(Remember our Motivating Problem) (4/4)

- New Problem!

- Given: A graph with positive edge weights
 - (i.e. we need to find the shortest path)
- Problem:
 - Find the shortest path from source to destination
- Solution: ???

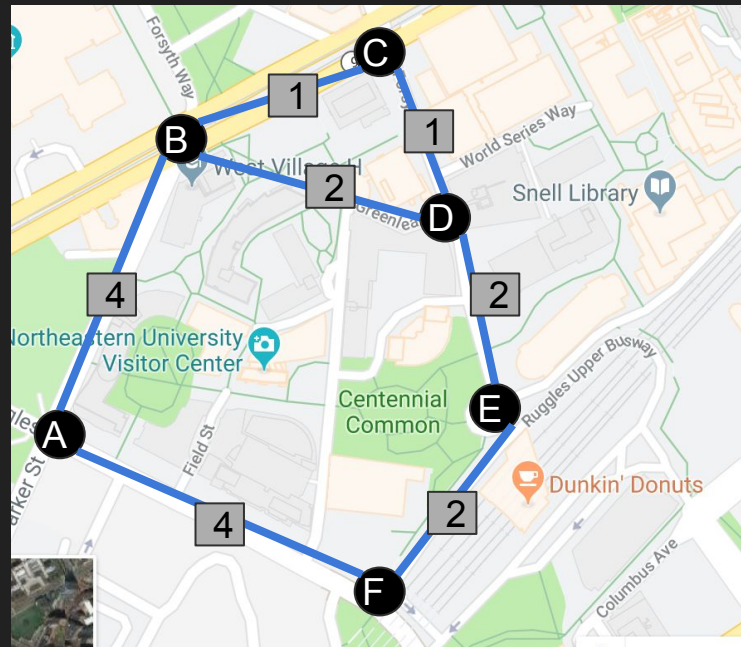
Question to the audience:

What strategies might you use to solve this problem?



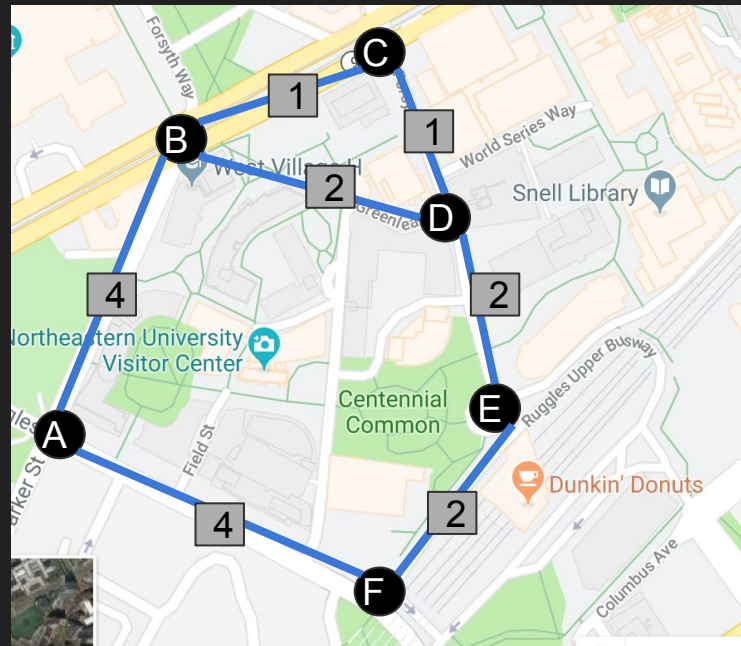
Solution # 1 of 2 - Brute Force

- We can always try the brute force approach
 - Strategy:
 - Permute every possible path and add up the weights
 - Then output the path with the lowest value
 - Complexity
 - Well, as we know, $O(n!)$ if we permute everything...
 - A
 - A, B
 - A, B, C
 - ...
 - A,B,C,D,E,F
 - A,B,C,D,F,E
 - A,B,C,E,F,D



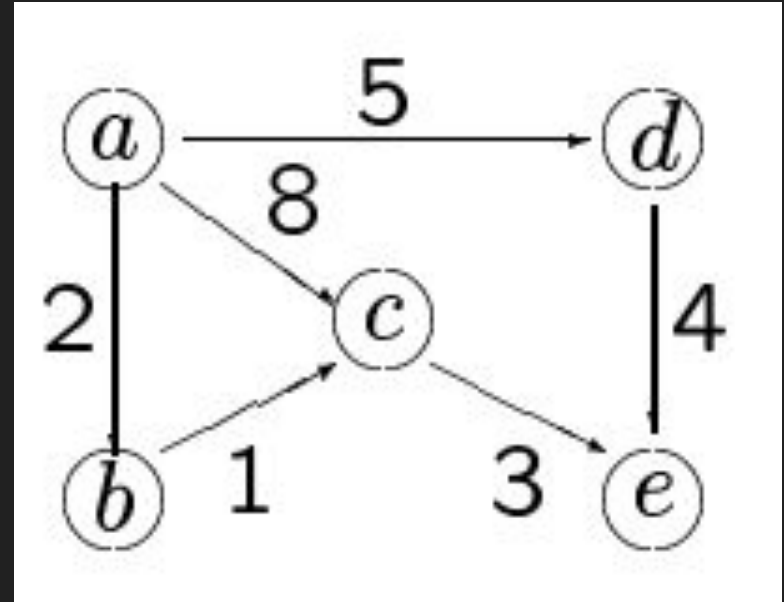
Solution # 2 of 2 - Greedy Strategy

- Greedy strategy
 - (Reminder: Greedy algorithms work by making the 'optimal decision' first)
 - Take the shortest path from our start, and continue to do that.
- Question to audience: Will this work?
 - Why or why not?
 - (Answer: next slide)



A Greedy Finding - Subpath of a shortest path is also a shortest path


- In the diagram on the right, the path:
- $\langle a, b, c, e \rangle$ is a shortest path to e
 - compare it to $\langle a, d, e \rangle$ for instance
- We can also note that $\langle a, b, c \rangle$ is also a shortest path
 - compare it to $\langle a, c \rangle$
- Thus taking the 'shortest sub path' of the shortest path must be along the shortest path.



Dijkstra's Shortest Path | Greedy Shortest Path Problems

- An example greedy algorithm is Dijkstra's Shortest Path
- It answers the question:
 - *From a single starting point, how long does it take to get to all destinations?*
 - i.e. What is the Single-Source Shortest Path (SSSP)
 - (i.e. finding a path is just another search problem)

Single-Source Single-Destination (1-1) <ul style="list-style-type: none">- No good solution that beats 1-M variant- Thus, this problem is mapped to the 1-M variant	Single-Source All-Destination (1-M) <ul style="list-style-type: none">- Need to be solved (several algorithms)- We will study this one
All-Sources Single-Destination (M-1) <ul style="list-style-type: none">- Reverse all edges in the graph- Thus, it is mapped to the (1-M) variant	All-Sources All-Destinations (M-M) <ul style="list-style-type: none">- Need to be solved (several algorithms)- We will study it (if time permits)



Dijkstra's Shortest Path

Pseudocode

Dijkstra's Algorithm

Pseudo-code [[wiki](#)] (1/2)

- First initialize our nodes to unknown distances
 - (i.e. something big like infinity)
 - i.e.
 - $\text{dist}[v] = \text{INFINITY}$ because we do not know how long it takes to get to that vertex from our source.
- Then visit neighbors (i.e. adjacent or connected nodes) from each node
 - We choose the minimum value to start
 - (using a priority queue which we will discuss)
 - (This is the greedy part--selecting the lowest or minimum value)
 - If the path to a neighbor is less than a known path (i.e the current alternative)
 - Then update the path
 - *This is known as the relaxing step*

```
1 function Dijkstra(Graph, source):
2     dist[source] ← 0
3
4     create vertex set Q
5
6     for each vertex v in Graph:
7         if v ≠ source
8             dist[v] ← INFINITY
9             prev[v] ← UNDEFINED
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:
15         u ← Q.extract_min()
16         for each neighbor v of u:
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

Dijkstra's Algorithm

Pseudo-code [[wiki](#)] (2/2)

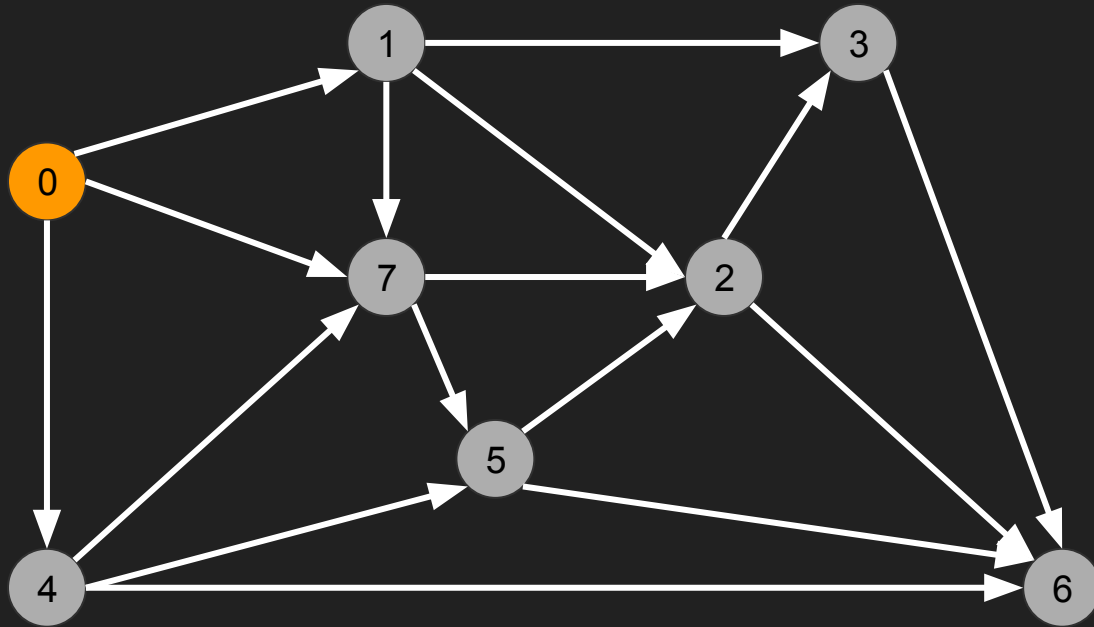
- First initialize our nodes to unknown distances
 - (i.e. something big like infinity)
 - i.e.
 - $\text{dist}[v] = \text{INFINITY}$ because we do not know how long it takes to get to that vertex from our source.
- Then visit neighbors (i.e. adjacent or connected nodes) from each node
 - We choose the minimum value to start
 - (using a priority queue which we will discuss)
 - (This is the greedy part--selecting the lowest or minimum value)
 - If the path to a neighbor is less than a known path (i.e the current alternative)
 - Then update the path
 - *This is known as the relaxing step*

```
1 function Dijkstra(Graph, source):
2     dist[source] ← 0
3
4     create vertex set Q
5
6     for each vertex v in Graph:
7         if v ≠ source
8             dist[v] ← INFINITY
9             prev[v] ← UNDEFINED
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:
15         u ← Q.extract_min()
16         for each neighbor v of u:
17             alt ← dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist, prev
```

Dijkstra's Algorithm Example

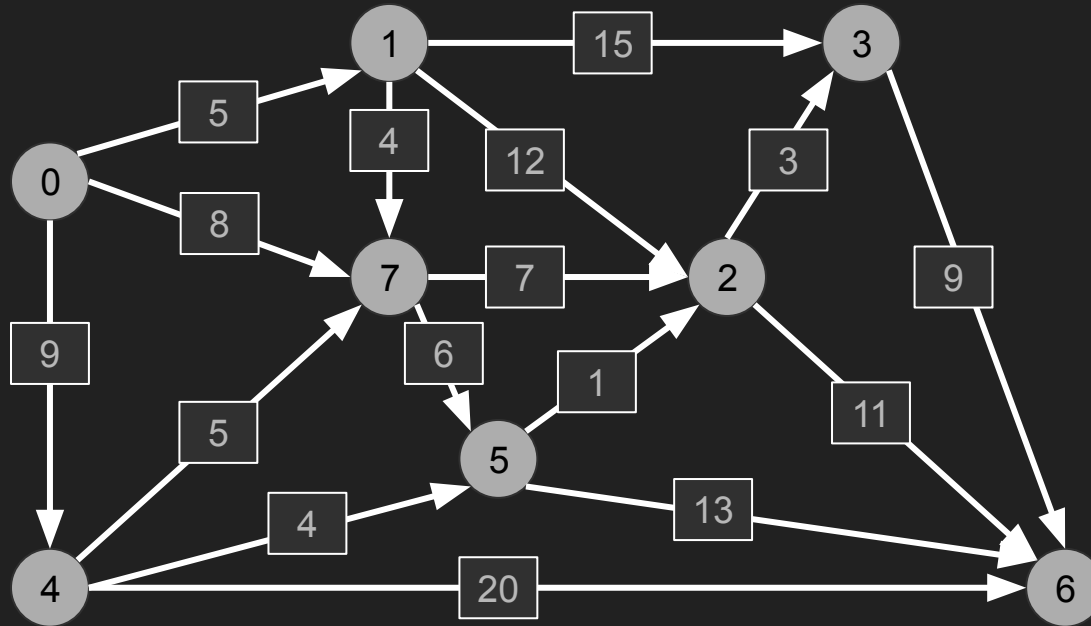
Graph from: <https://algs4.cs.princeton.edu/lectures/44DemoDijkstra.pdf> (see Module sources)

Given this graph, what is the shortest path from '0' to each node.



We have all of the edges here with their weights on the right

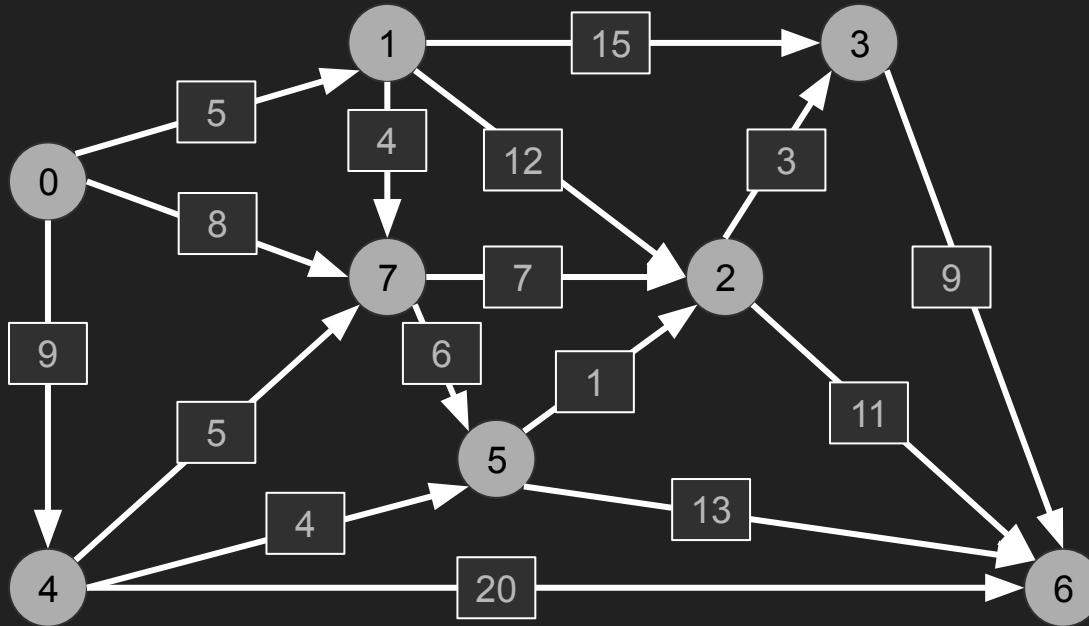
This is thus a directed graph (Dijkstra's works for undirected as well)



Edge List
and weights

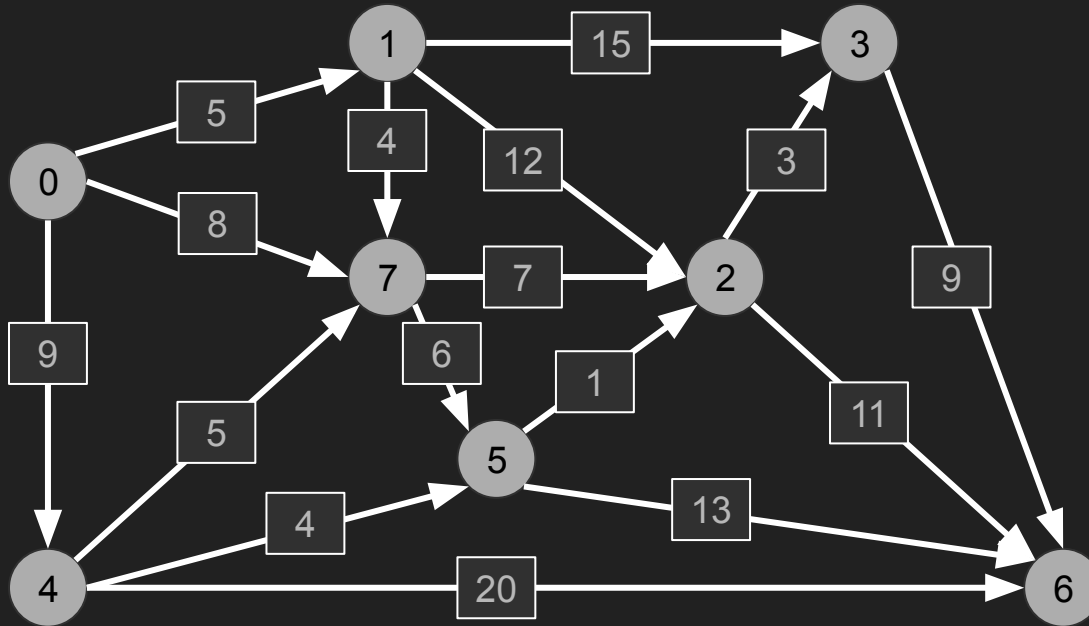
0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

Now we want to compute the 'shortest distance' from each node and we will keep track of this in a table.



v	distTo	EdgeTo
0		
1		
2		
3		
4		
5		
6		
7		

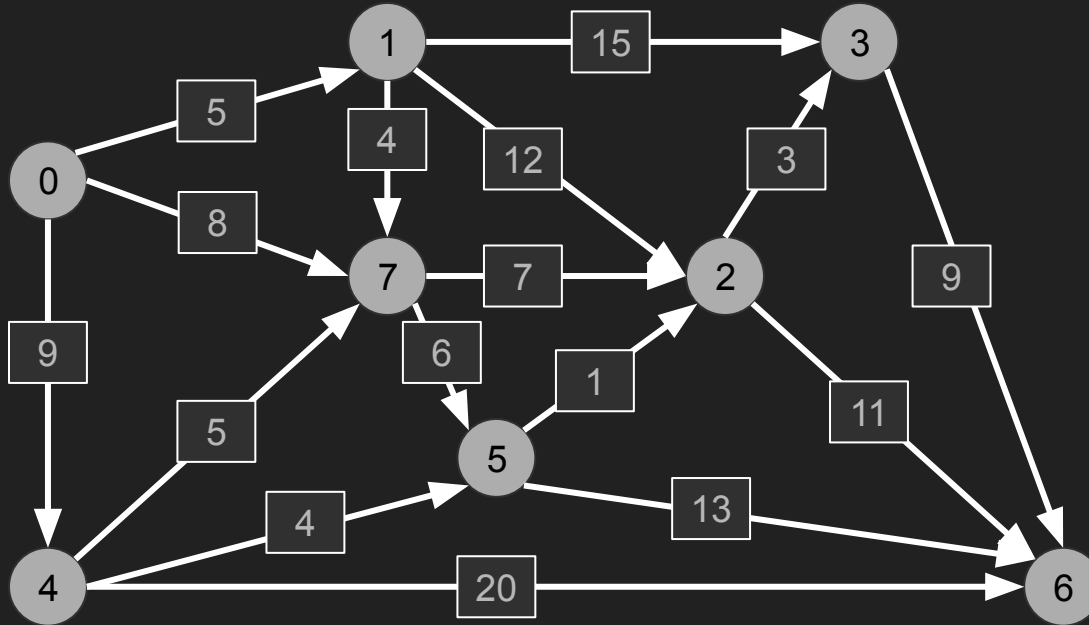
distTo - Is the current shortest path somewhere



v	<u>distTo</u>	EdgeTo
0	INF	
1	INF	
2	INF	
3	INF	
4	INF	
5	INF	
6	INF	
7	INF	

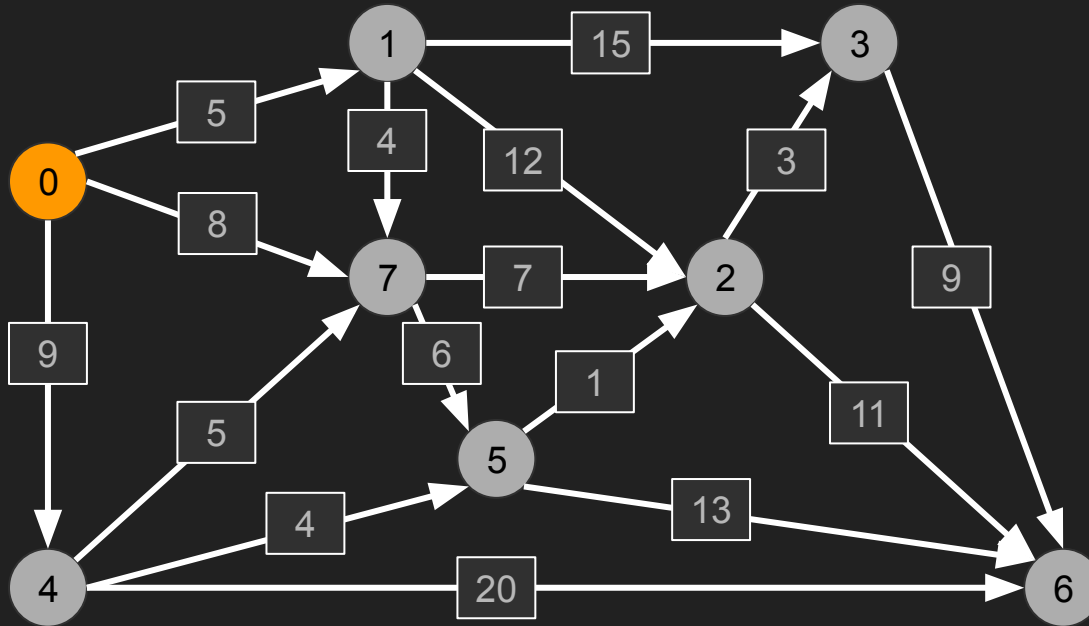
EdgeTo - Is previous edge that gets us to our shortest path

Note: (This is labeled as 'prev' in the previous pseudocode)



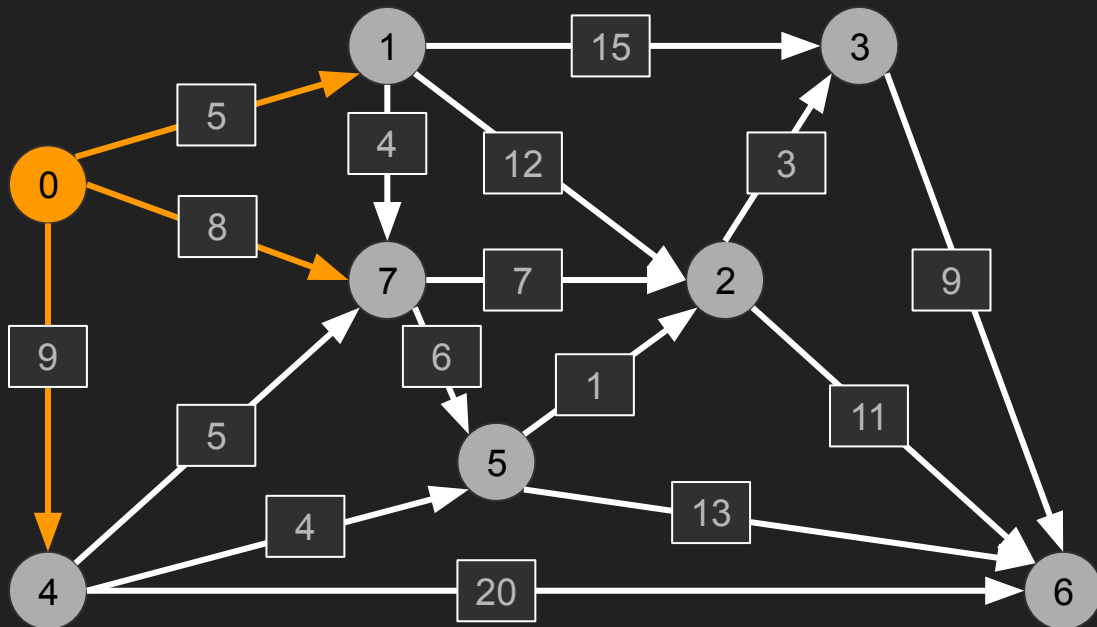
v	distTo	<u>EdgeTo</u>
0	INF	
1	INF	
2	INF	
3	INF	
4	INF	
5	INF	
6	INF	
7	INF	

Starting from node 0, we compute its distance to itself, which is '0'



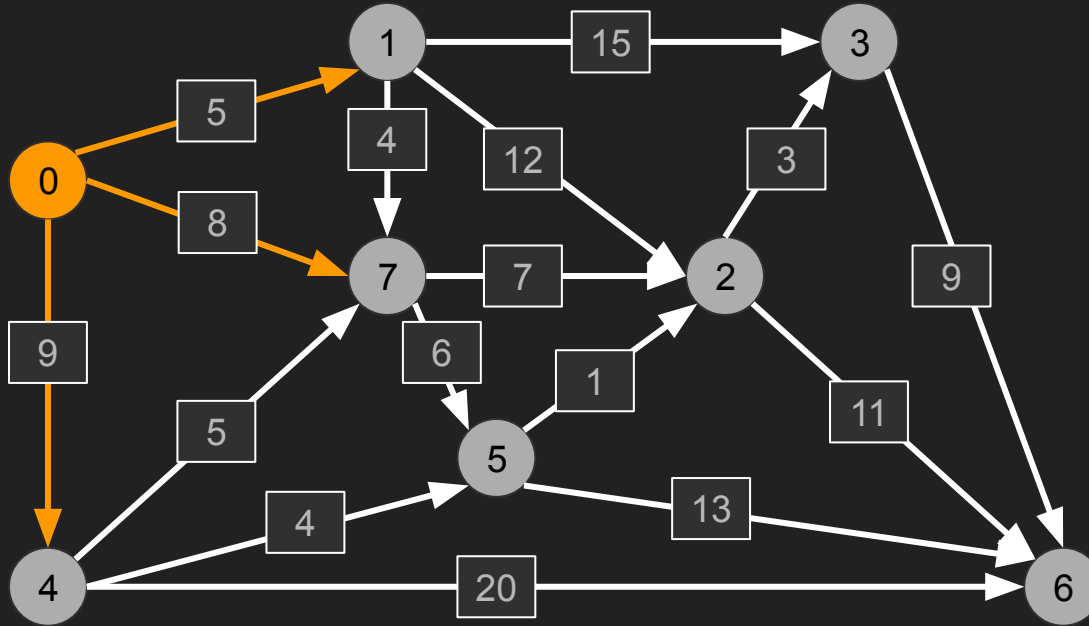
v	distTo	EdgeTo
0	INF	-
1	INF	
2	INF	
3	INF	
4	INF	
5	INF	
6	INF	
7	INF	

Now we look at all edges from 0



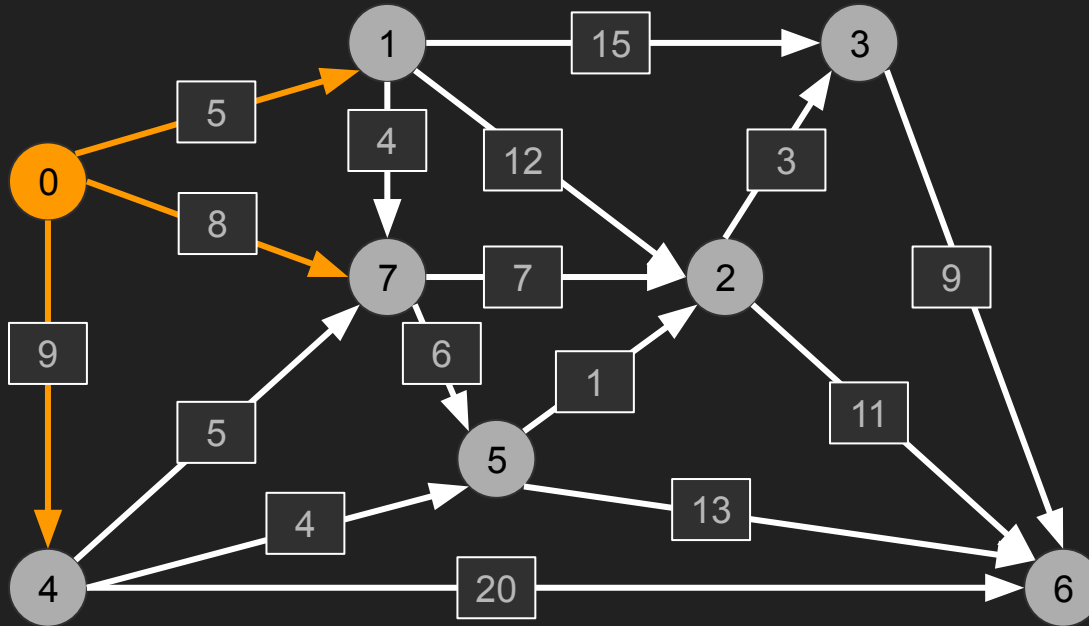
v	distTo	EdgeTo
0	0	-
1	INF	
2	INF	
3	INF	
4	INF	
5	INF	
6	INF	
7	INF	

We relax them (i.e. if the path is shorter to this path (which they are initially infinity) update the edgeTo value to a smaller value)



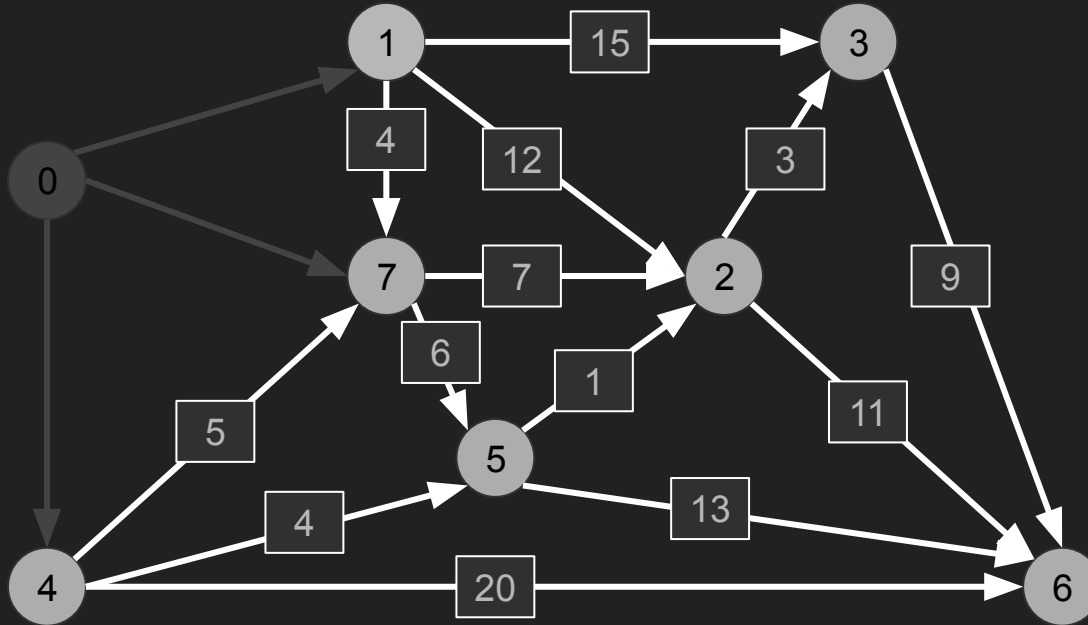
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	INF	
3	INF	
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

Note that we also mark from what edge we traveled to get there



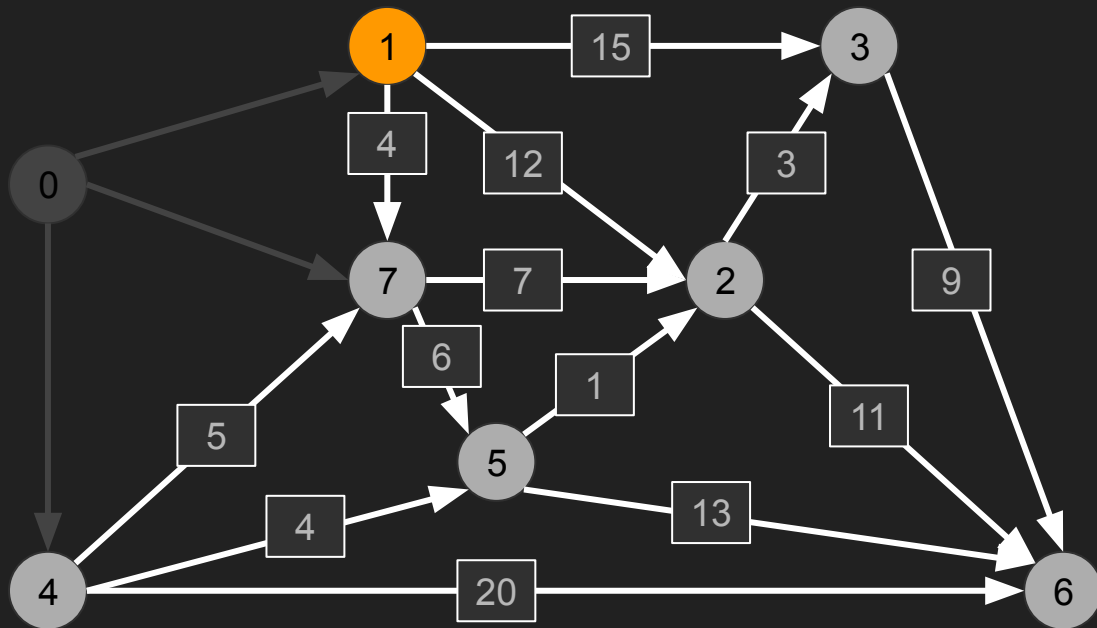
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	INF	
3	INF	
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

I now mark off '0' on my list



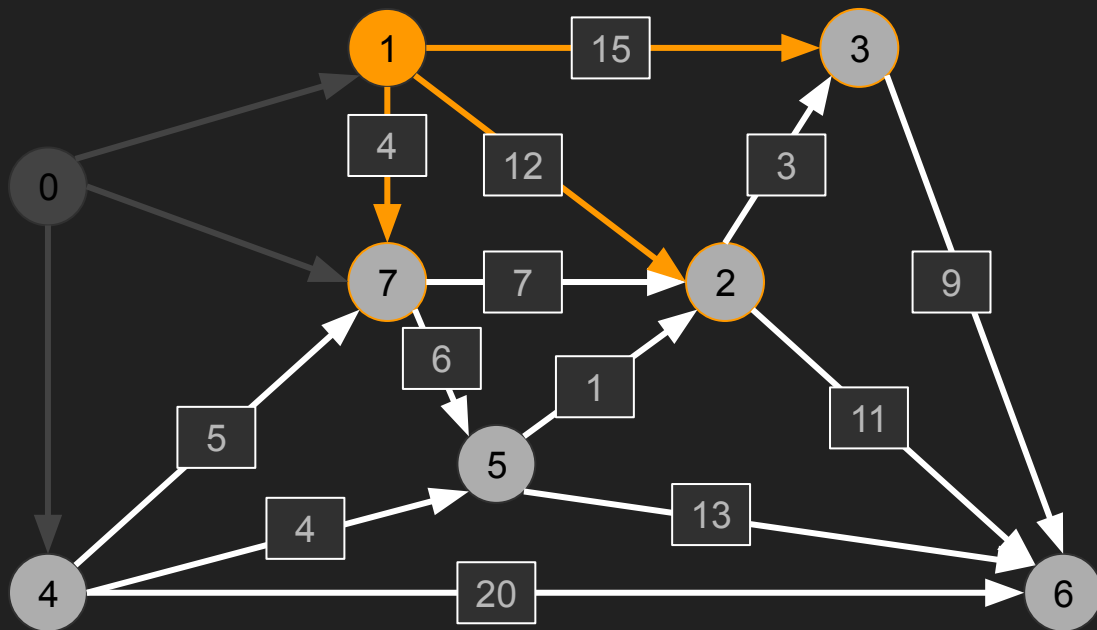
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	INF	
3	INF	
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

Now we consider the 'smallest' value next in distTo and repeat.
This means we start from vertex 1 ($5.0 < 8.0 < 9.0$)



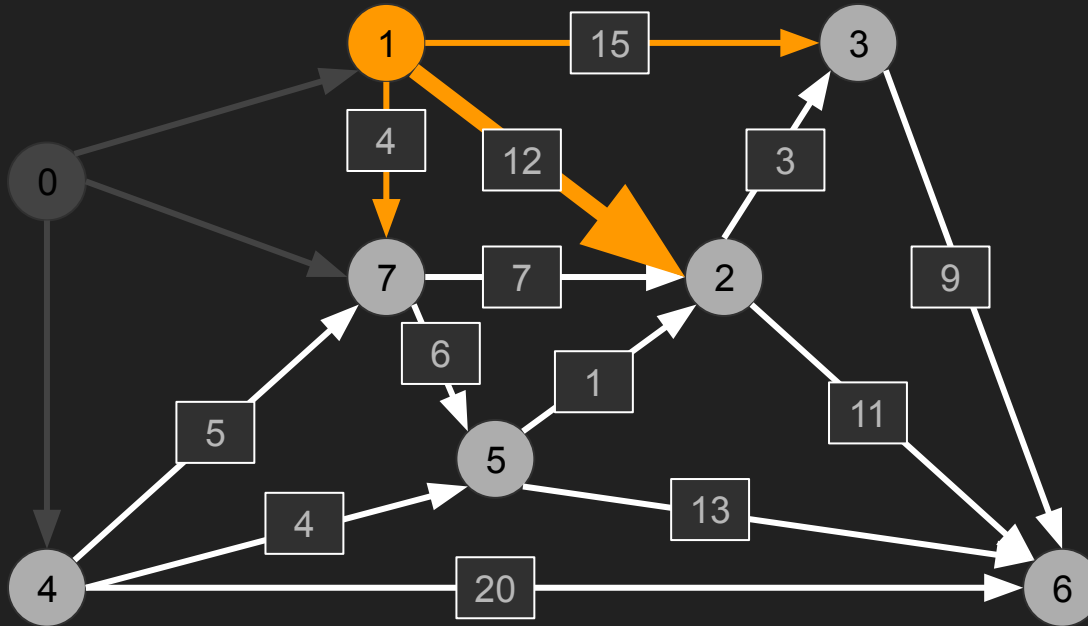
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	INF	
3	INF	
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

From node '1' I can reach nodes 2, 3, and 7



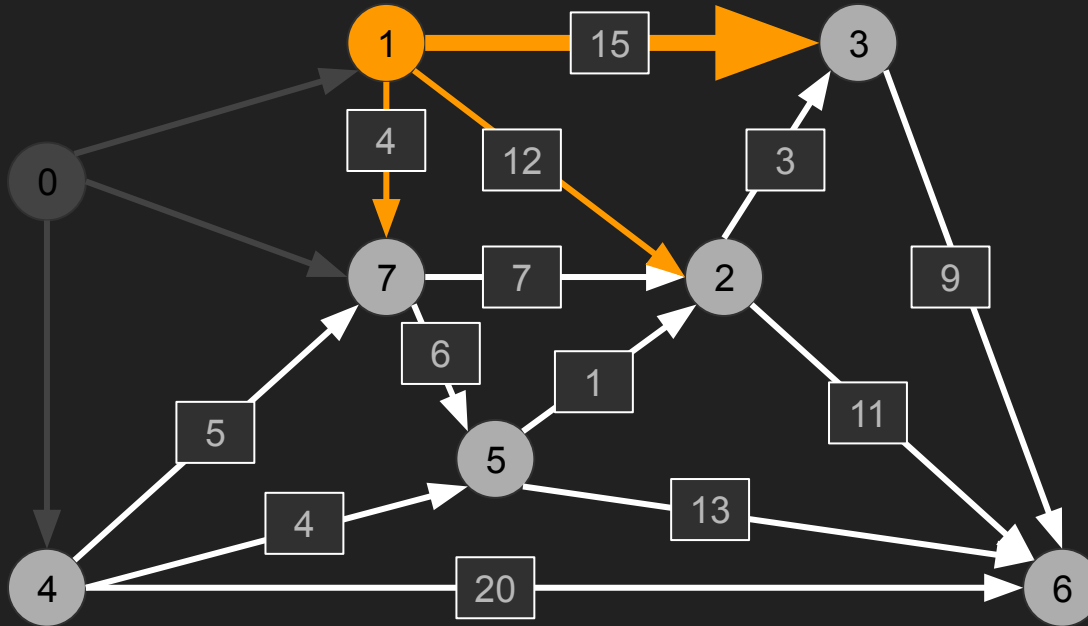
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	INF	
3	INF	
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

From node 1 to 2 (node 2 has not been visited), it costs me $5.0 + 12.0$



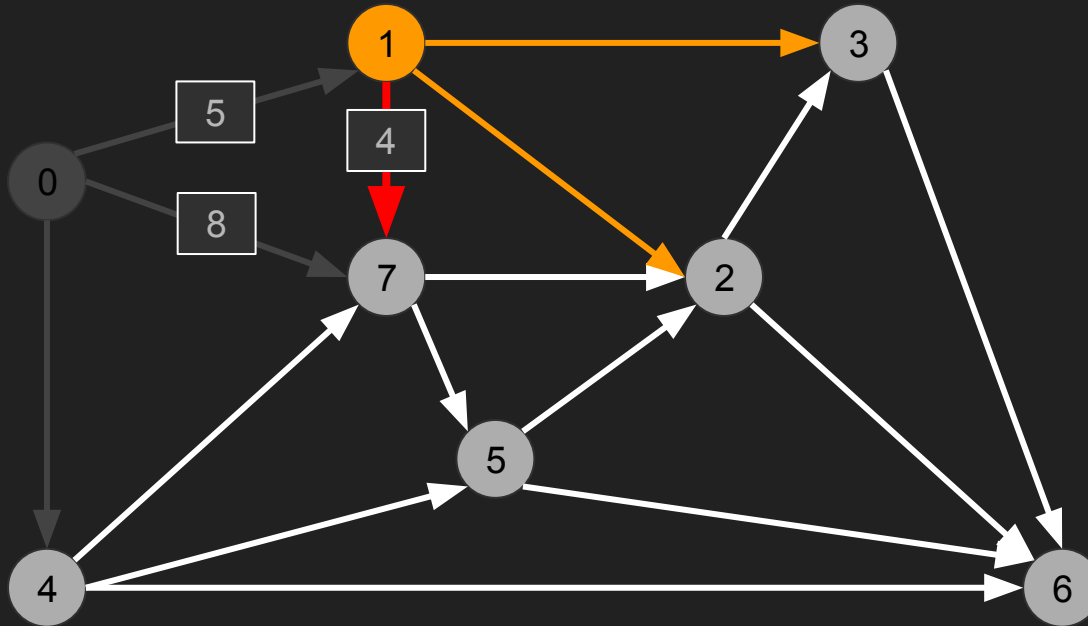
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	17.0	1->2
3	INF	
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

From node 1 to 3 (node 3 has not been visited), it costs me $5.0 + 15.0$



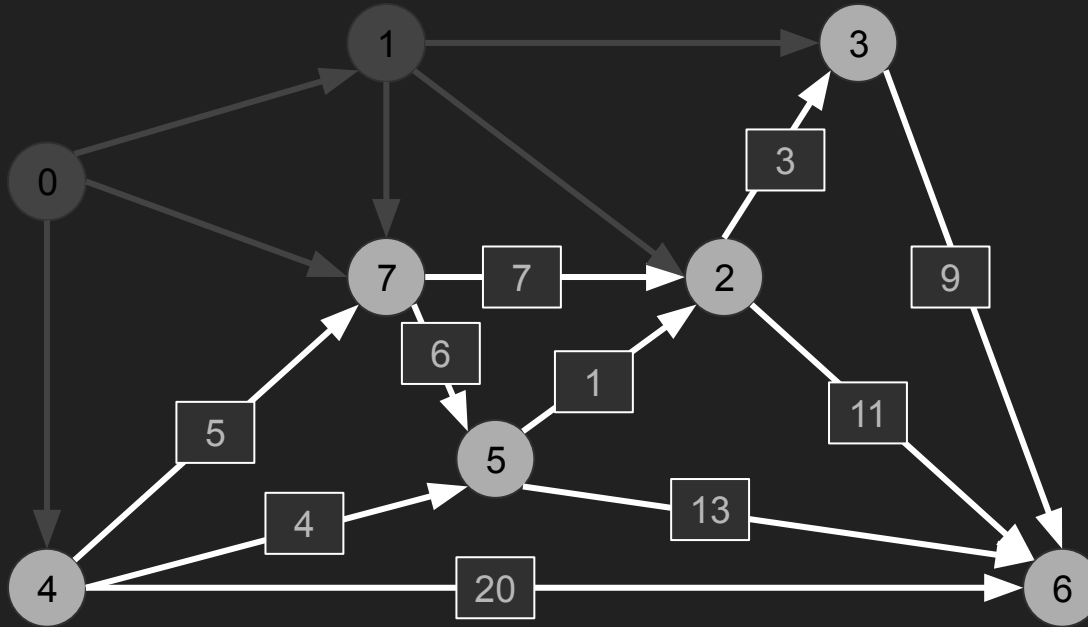
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	17.0	1->2
3	20.0	1->3
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

Of interest, is that we do not update the distance to node 7
Clearly we can see '8' (going 0->7) is less than '5+4' (0->1->7)



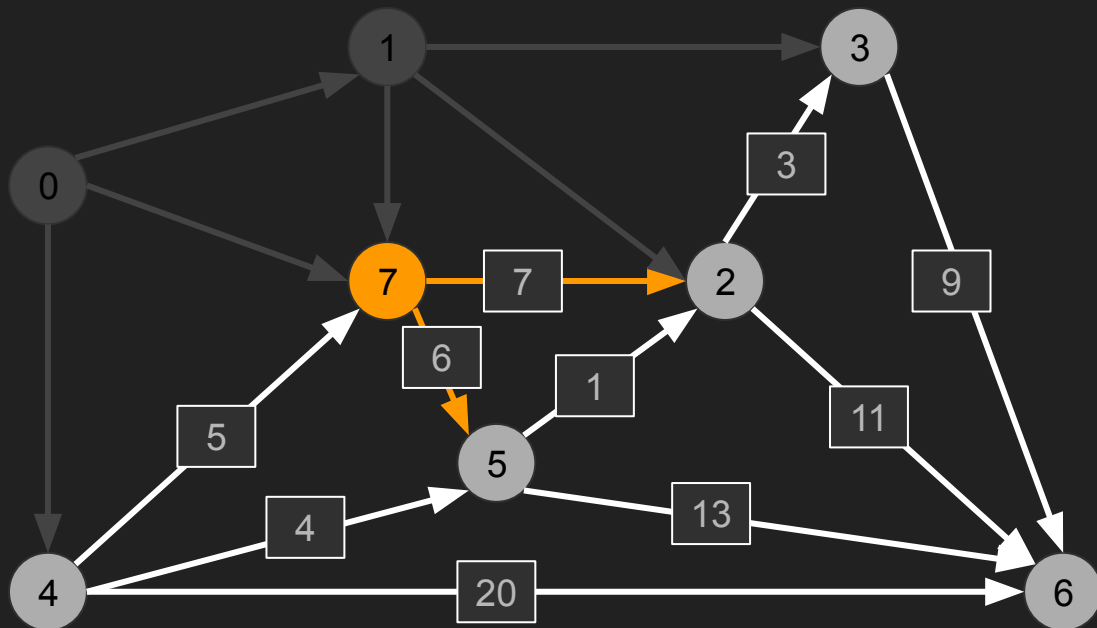
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	17.0	1->2
3	20.0	1->3
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

Okay, now we resume (Note: I checked off, '1' in our list, and we find the next smallest vertex '7' (I compared the values 8.0,9.0,17.0, and 20.0))



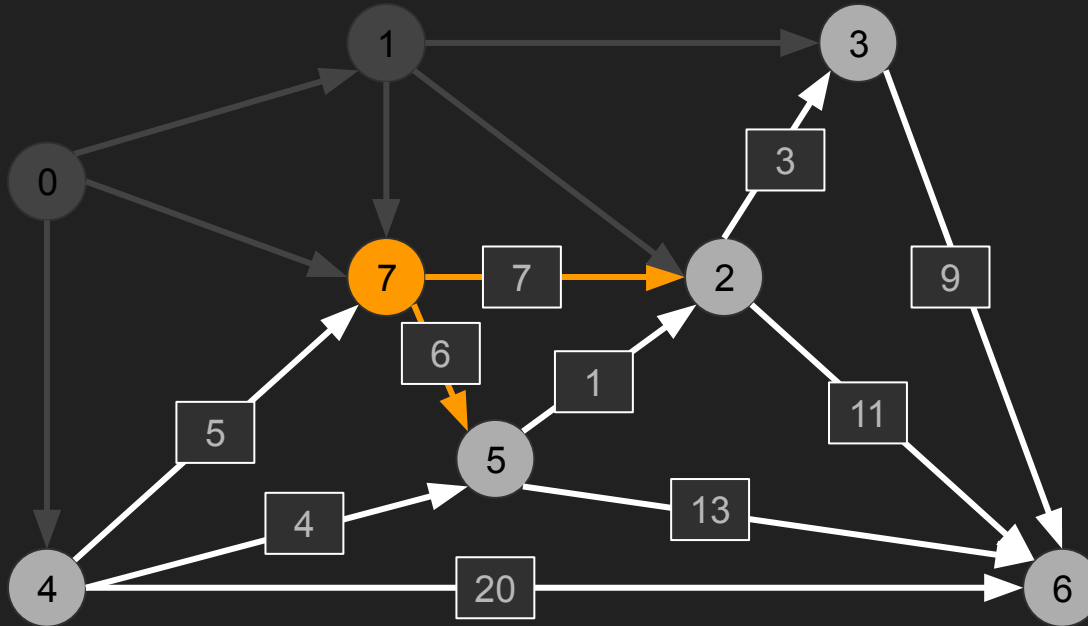
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	17.0	1->2
3	20.0	1->3
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

Find the edges



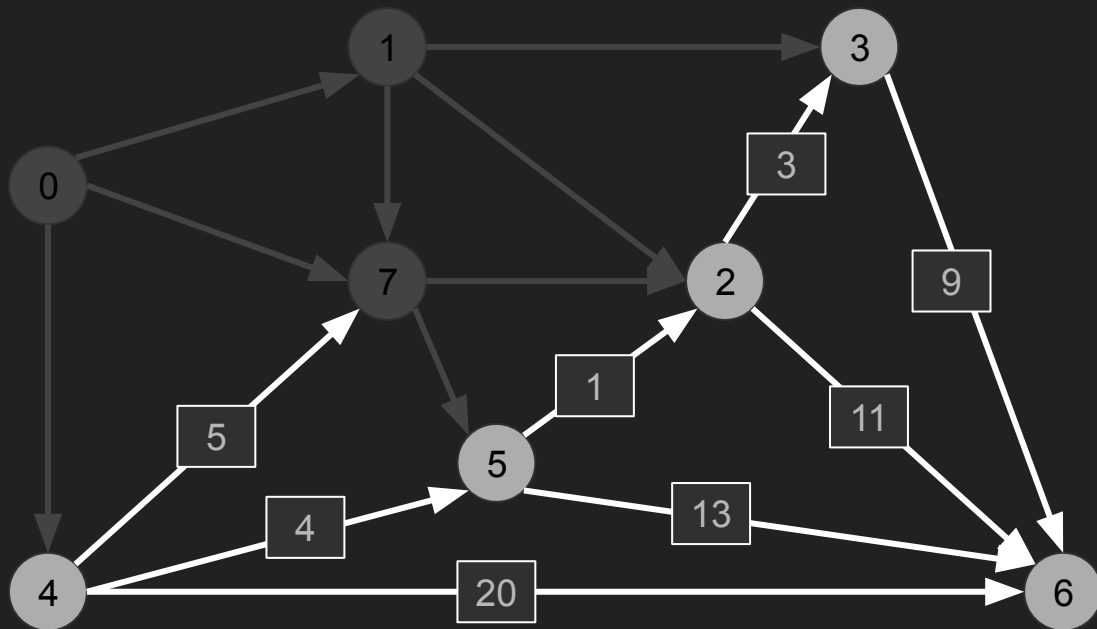
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	17.0	1->2
3	20.0	1->3
4	9.0	0->4
5	INF	
6	INF	
7	8.0	0->7

Update values



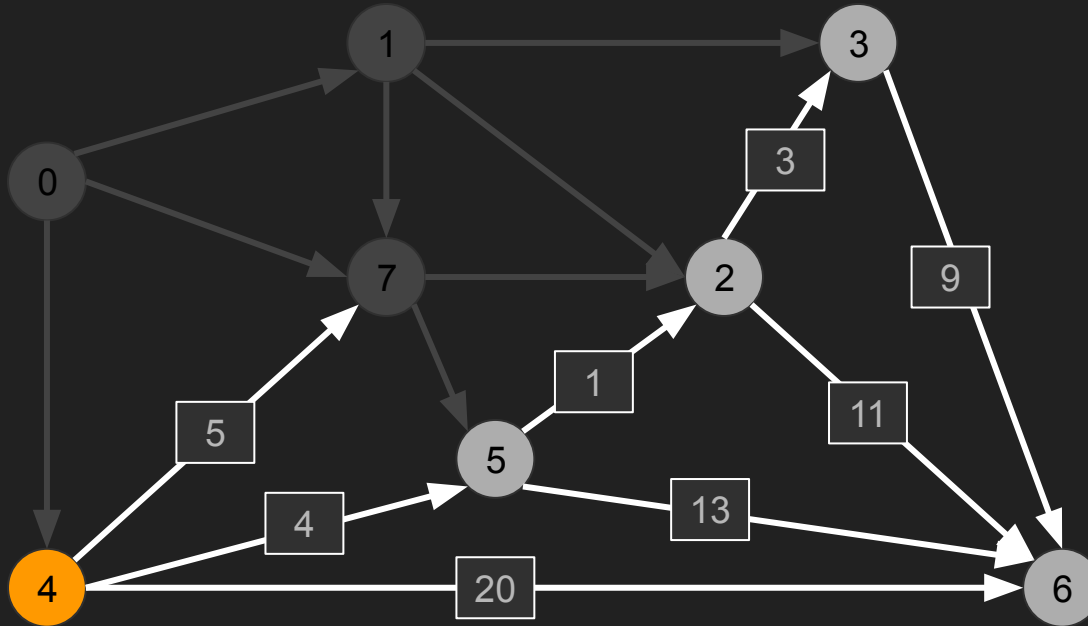
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	14.0	7->5
6	INF	
7	8.0	0->7

Done with 7



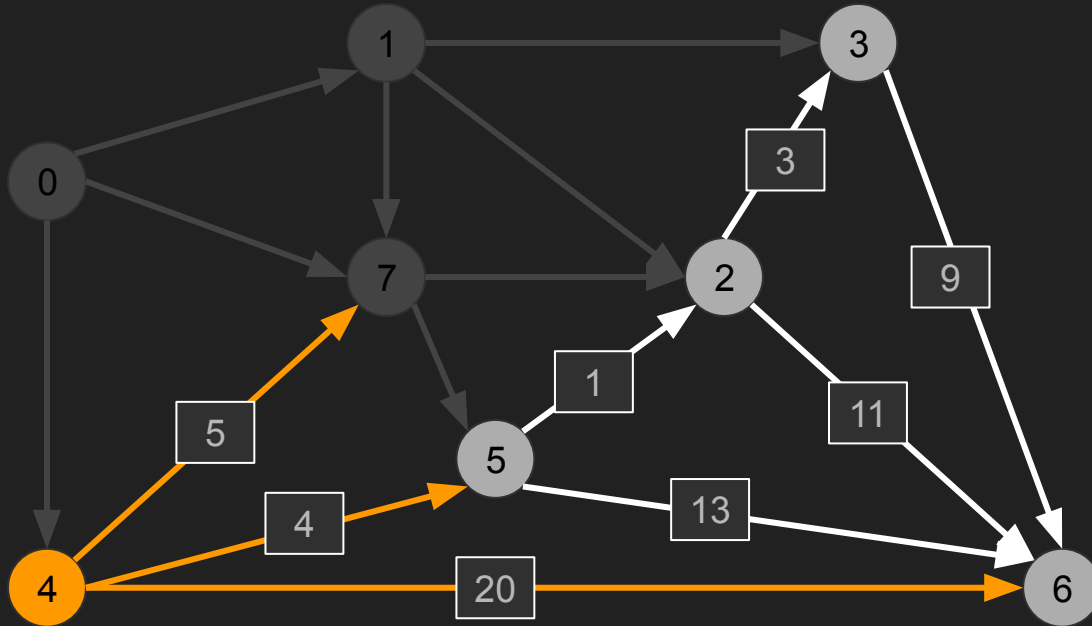
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	14.0	7->5
6	INF	
7	8.0	0->7

Next smallest value is 4



v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	14.0	7->5
6	INF	
7	8.0	0->7

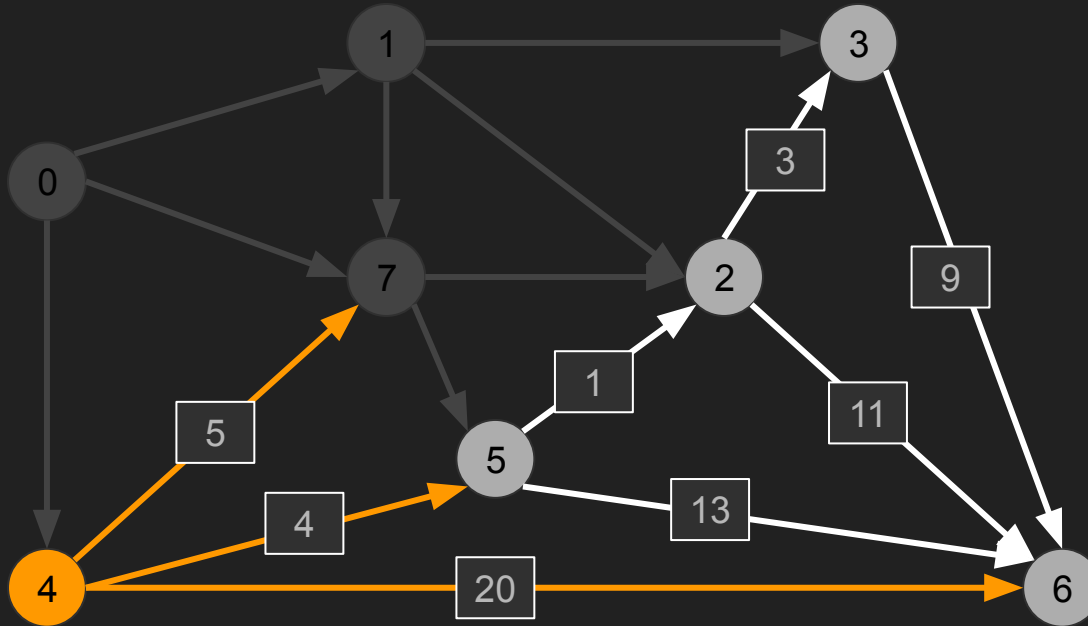
Relax paths



v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	14.0	7->5
6	INF	
7	8.0	0->7

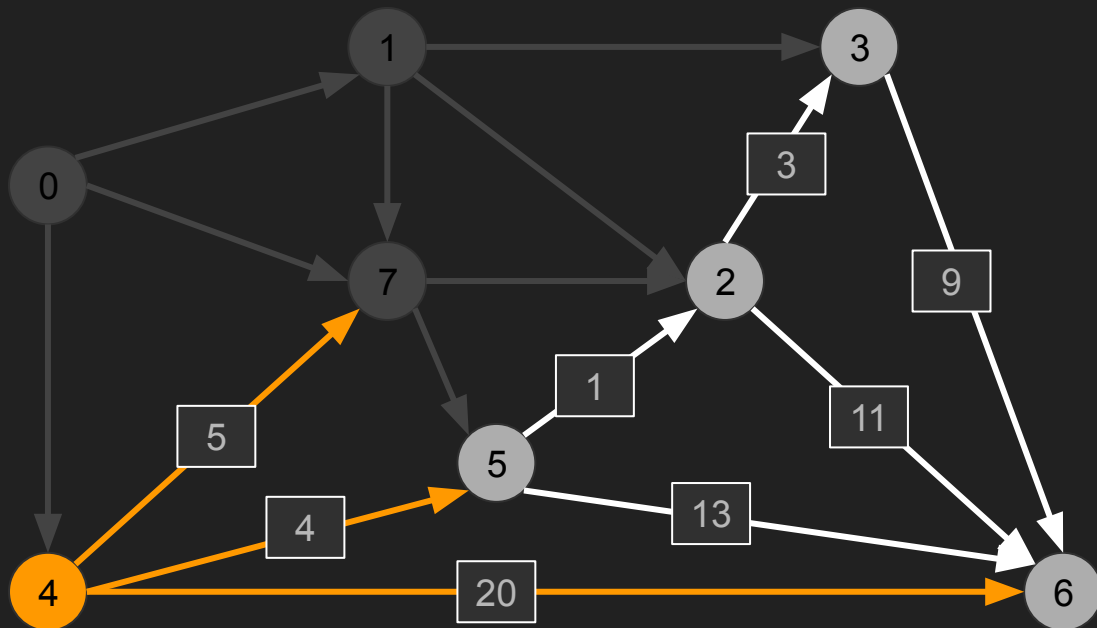
Make updates

I see I can get to node 5 in $(9.0 + 4.0)$ which is less than 14.0



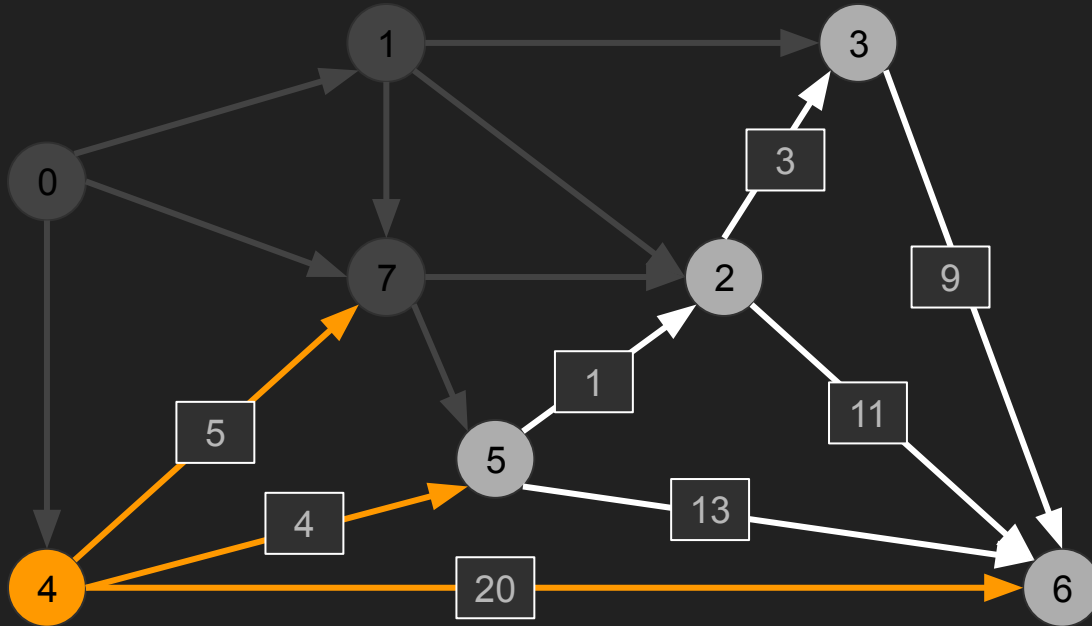
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	14.0	7->5
6	INF	
7	8.0	0->7

Update table



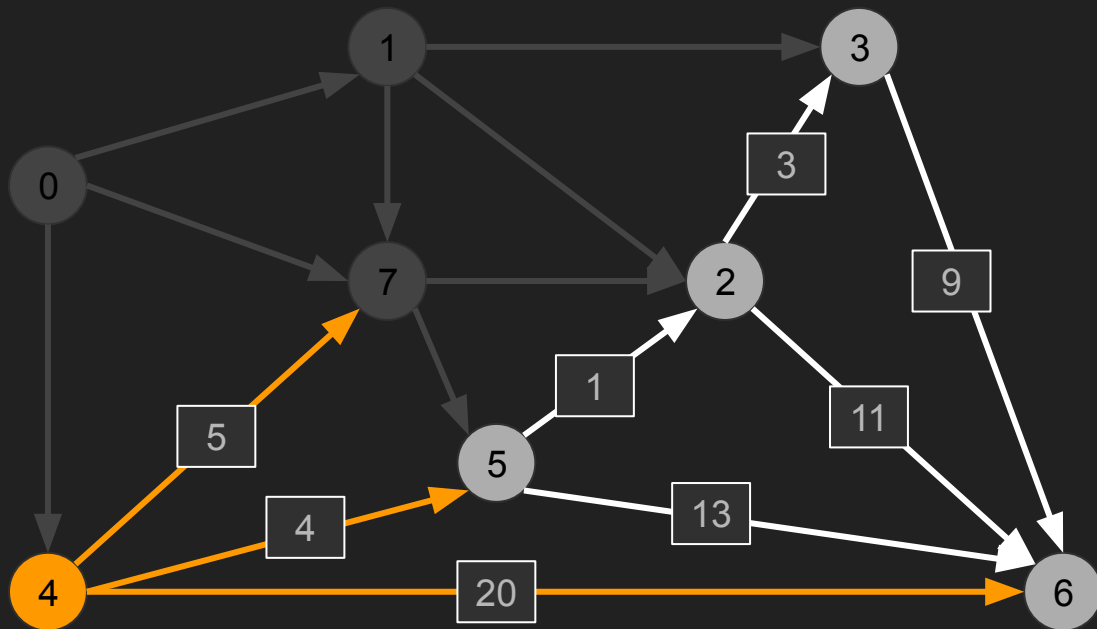
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	INF	
7	8.0	0->7

I can also get to node 6 now (so 9.0 it took me to get to 4, plus 20.0)



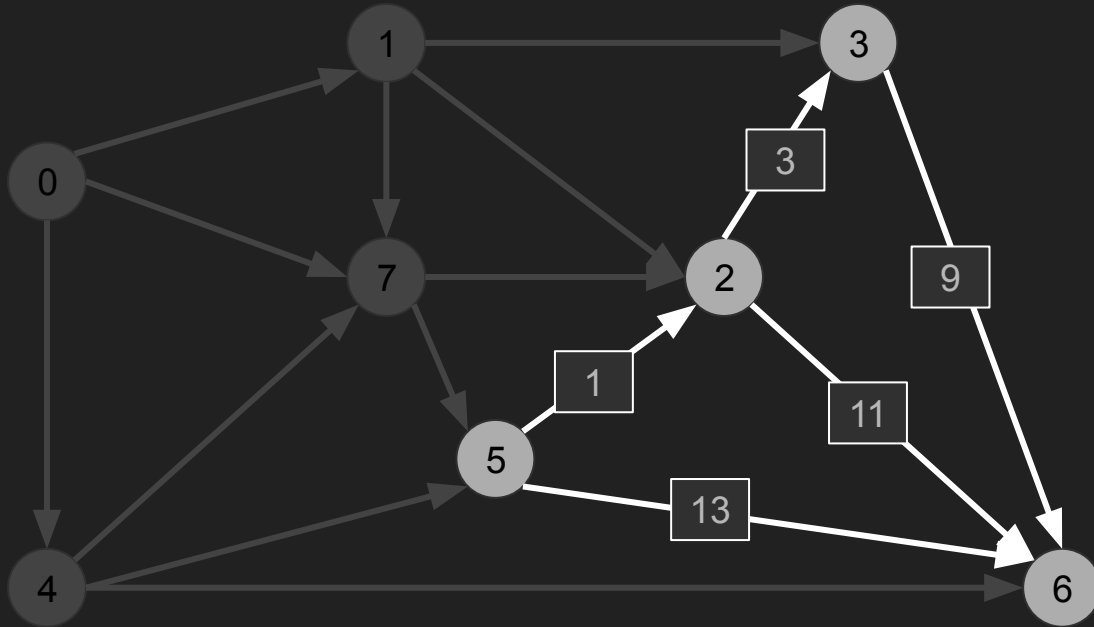
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	29.0	4->6
7	8.0	0->7

Getting to node 7 can be done in 8.0 (as opposed to $9.0 + 5.0$), so no updates made



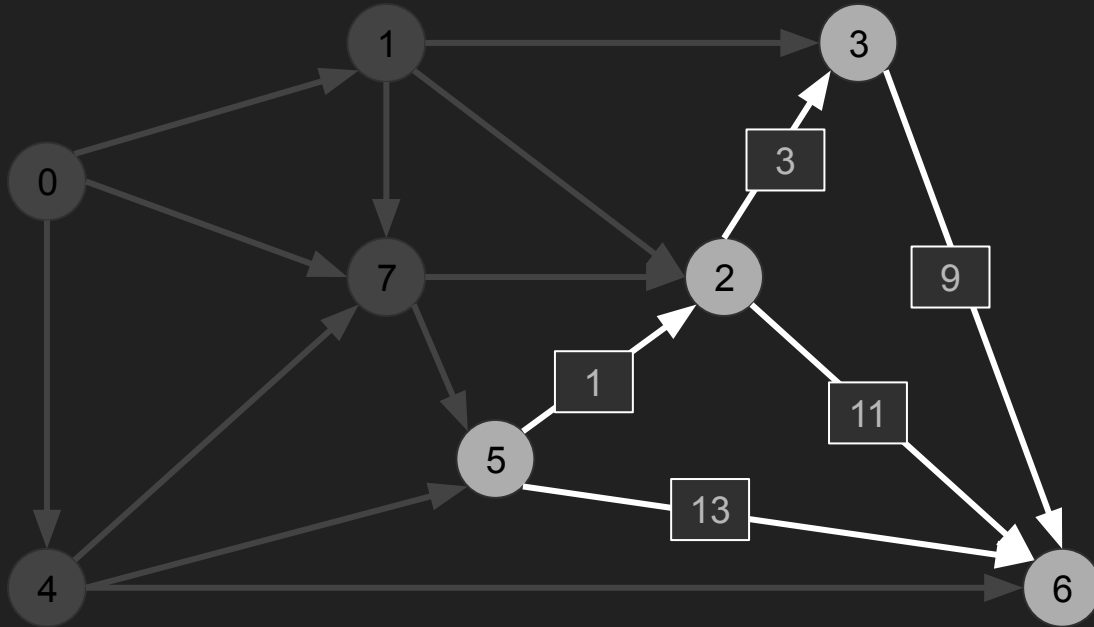
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	29.0	4->6
7	8.0	0->7

Done with 4



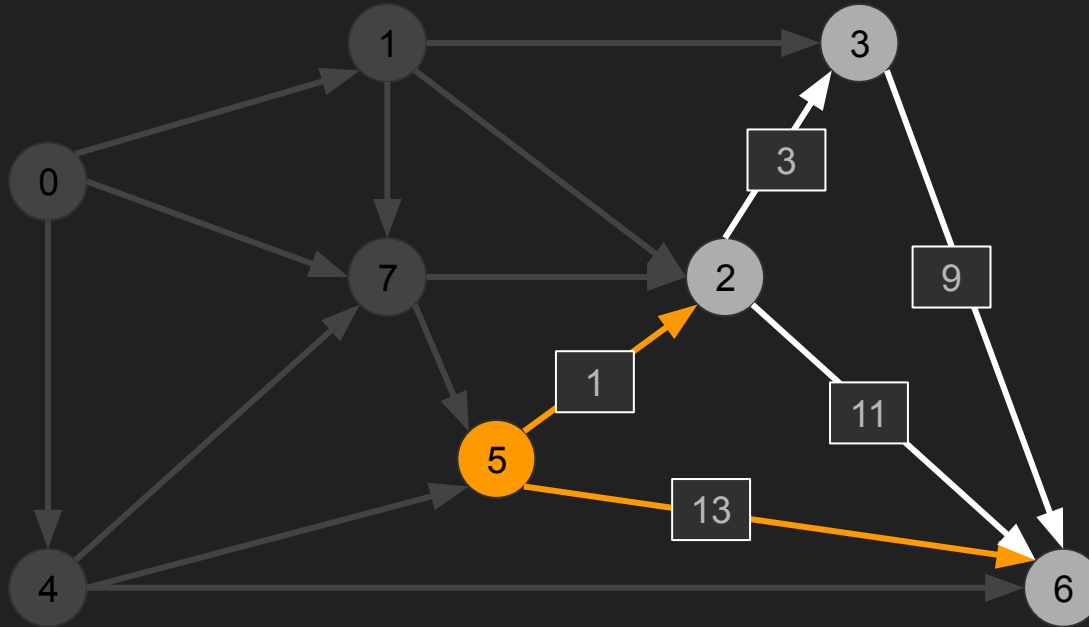
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	29.0	4->6
7	8.0	0->7

Now 5 is the lowest value for 'distTo'



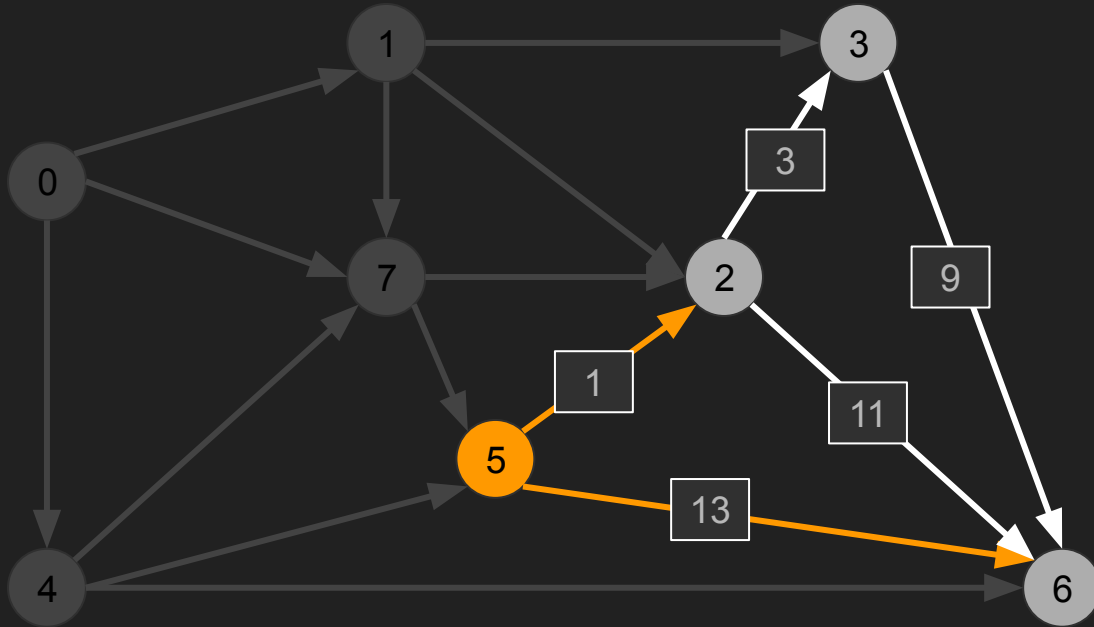
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	29.0	4->6
7	8.0	0->7

Now 5 is the lowest value for 'distTo'



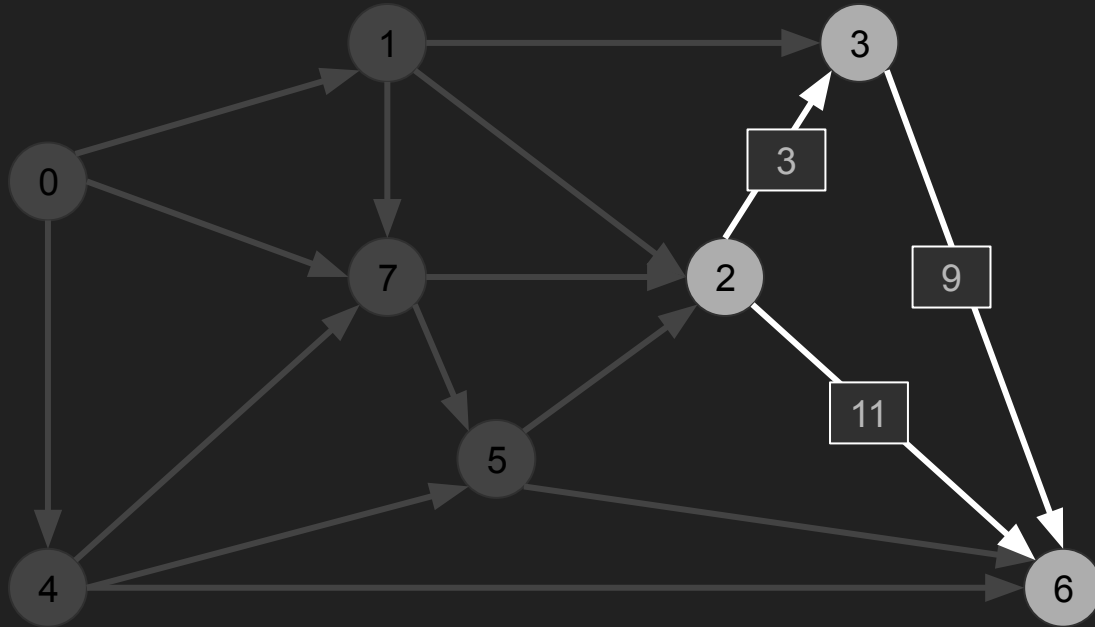
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	15.0	7->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	29.0	4->6
7	8.0	0->7

I update the table



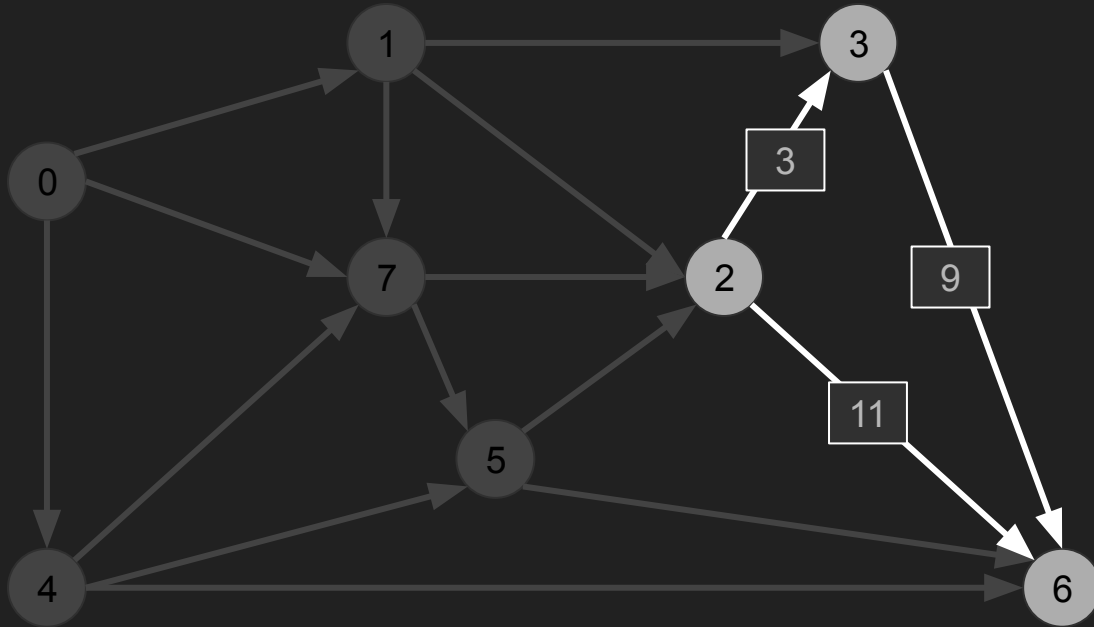
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	26.0	5->6
7	8.0	0->7

Done with 5



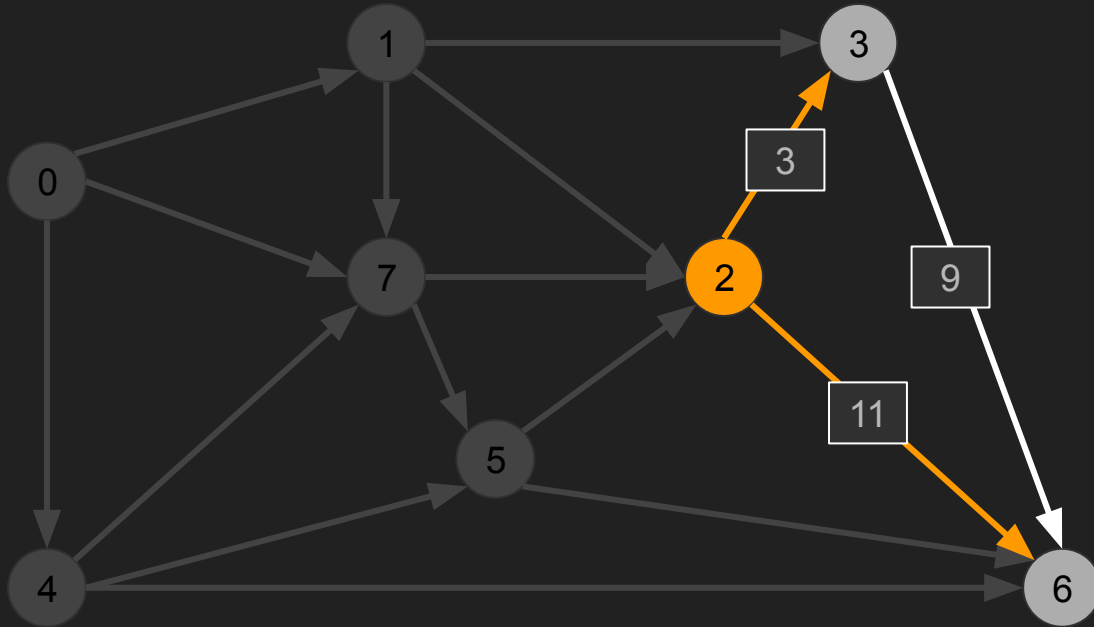
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	26.0	5->6
7	8.0	0->7

2 is the next smallest 'distTo' value



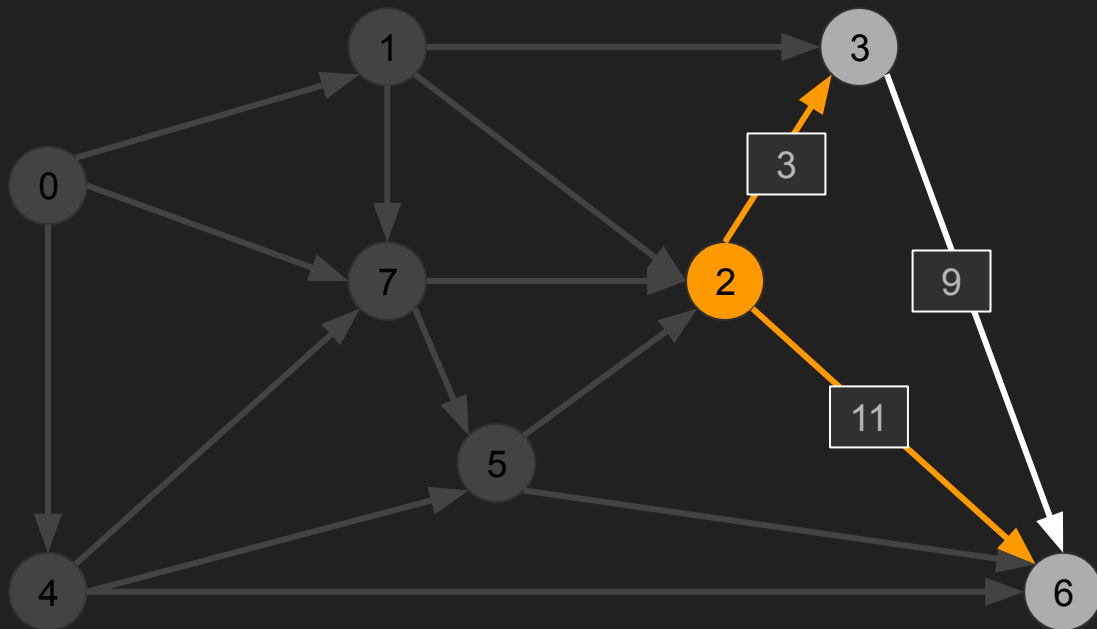
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	26.0	5->6
7	8.0	0->7

2 is the next smallest 'distTo' value



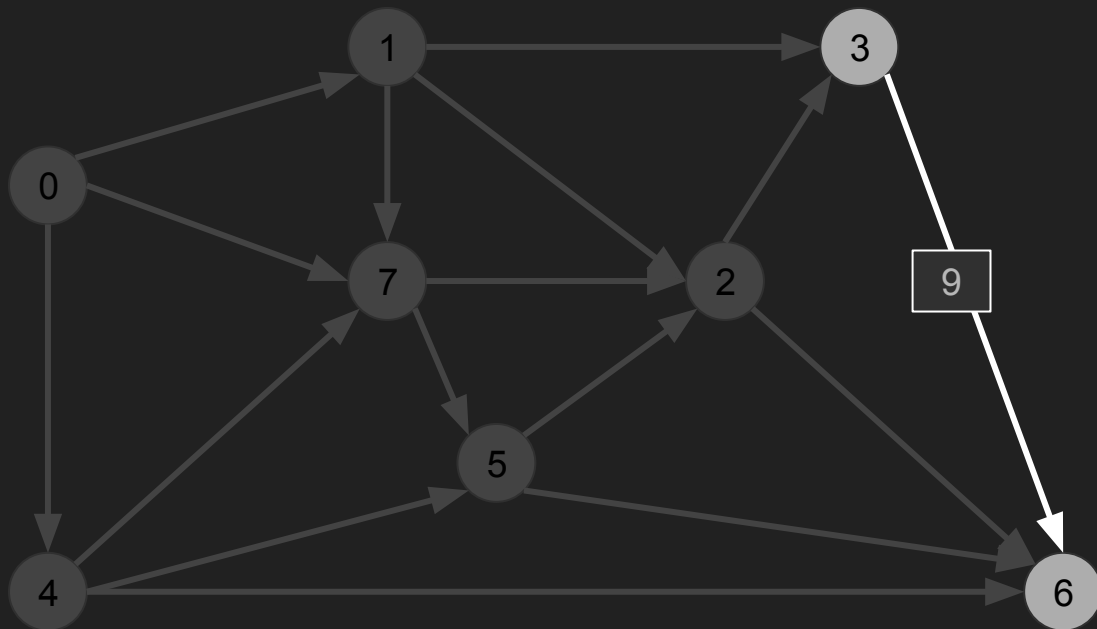
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	20.0	1->3
4	9.0	0->4
5	13.0	4->5
6	26.0	5->6
7	8.0	0->7

Update the table



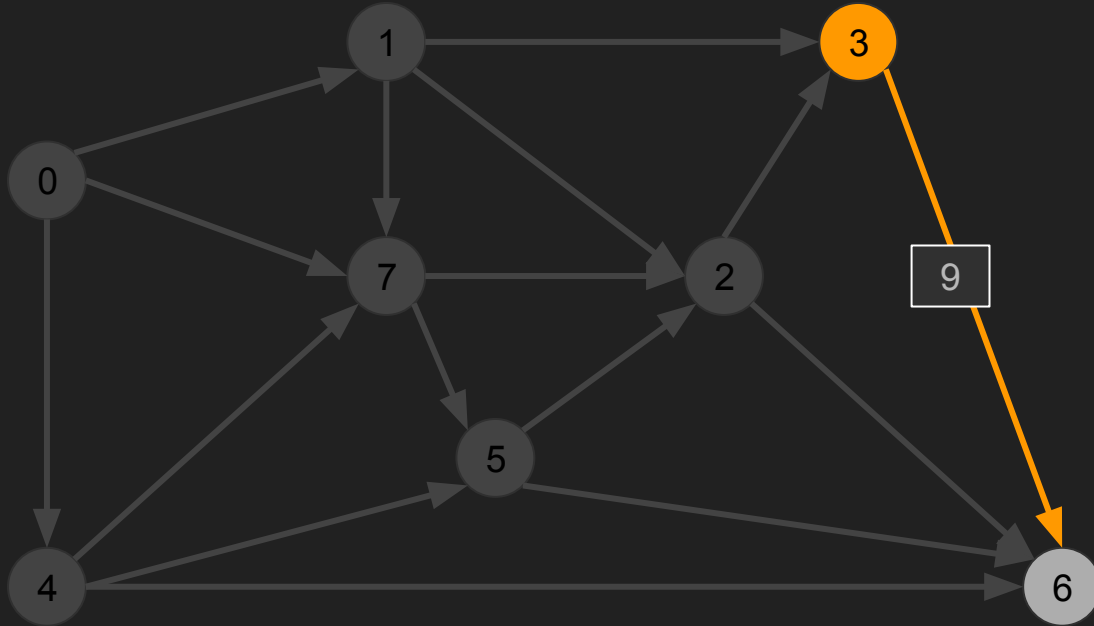
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

Done with 2



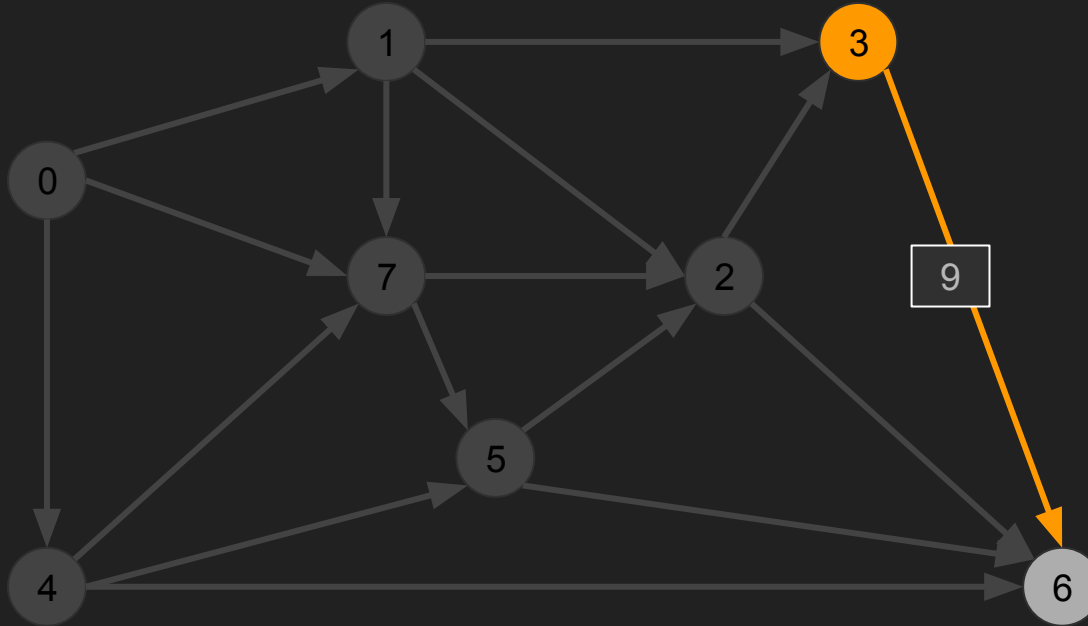
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

3 is the next minimum



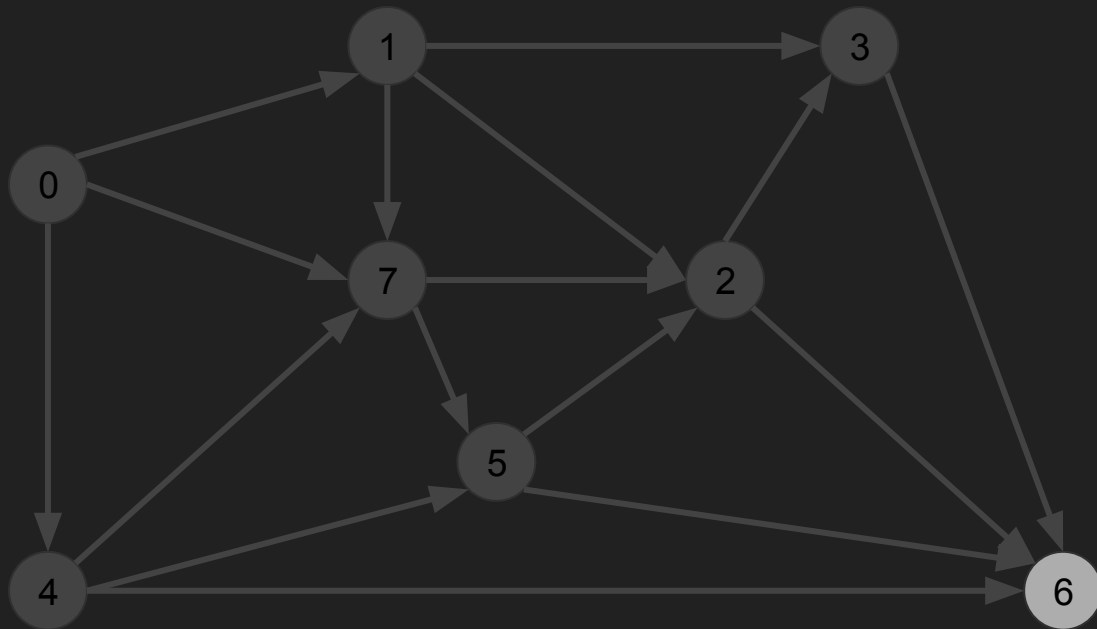
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

No updates to be made ($17+9$ would be greater than 25.0)



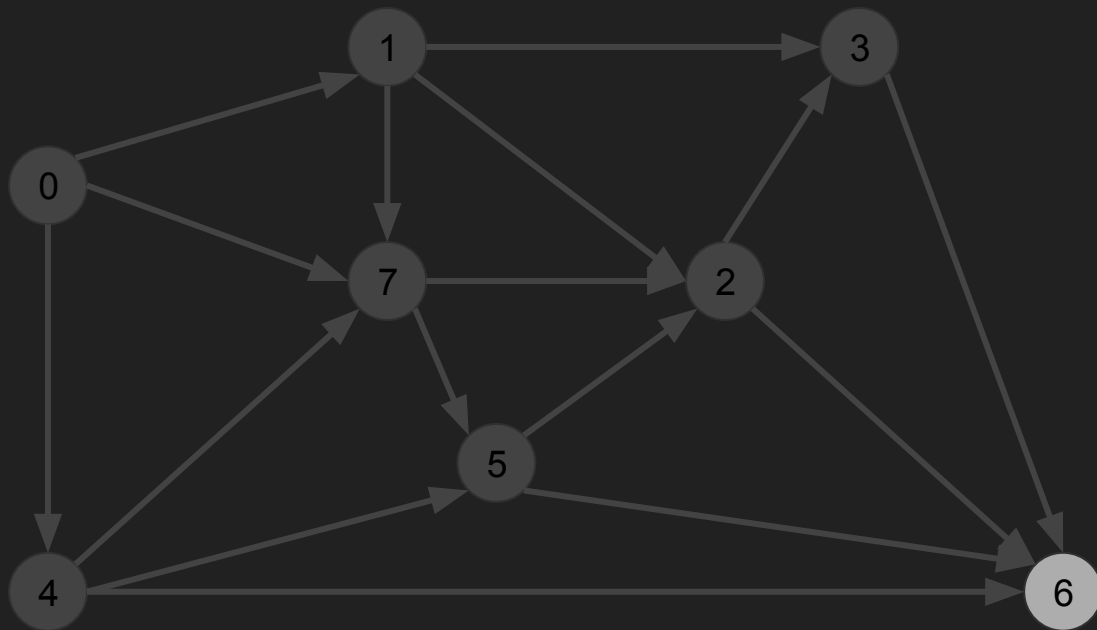
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

Done with 3



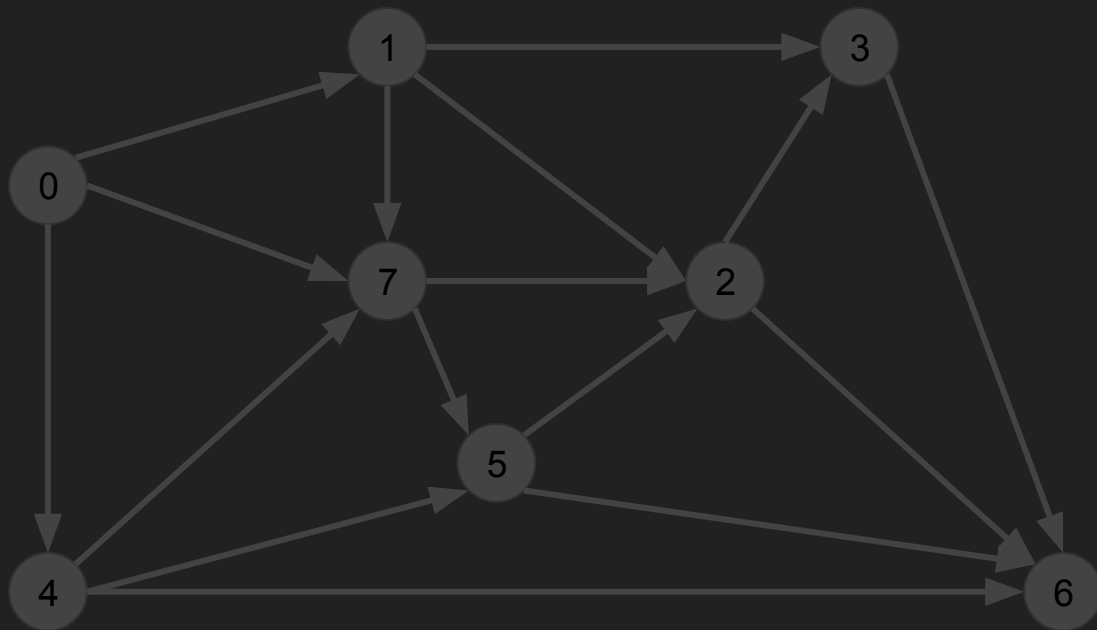
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

No edges pointing out of 6--so we're done!



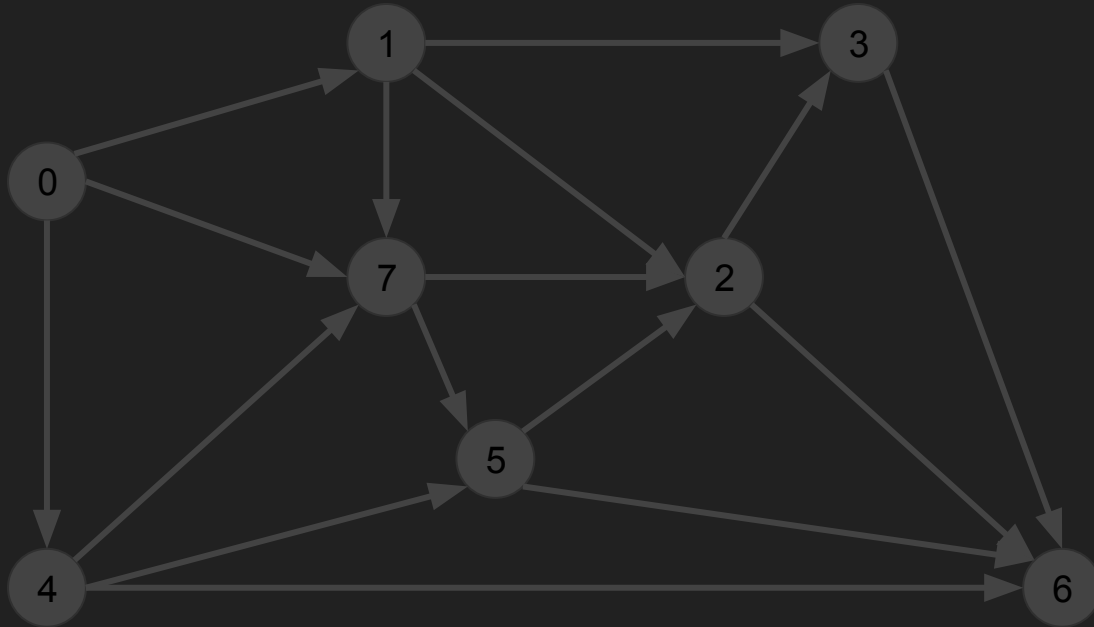
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

No edges pointing out of 6--so we're done!



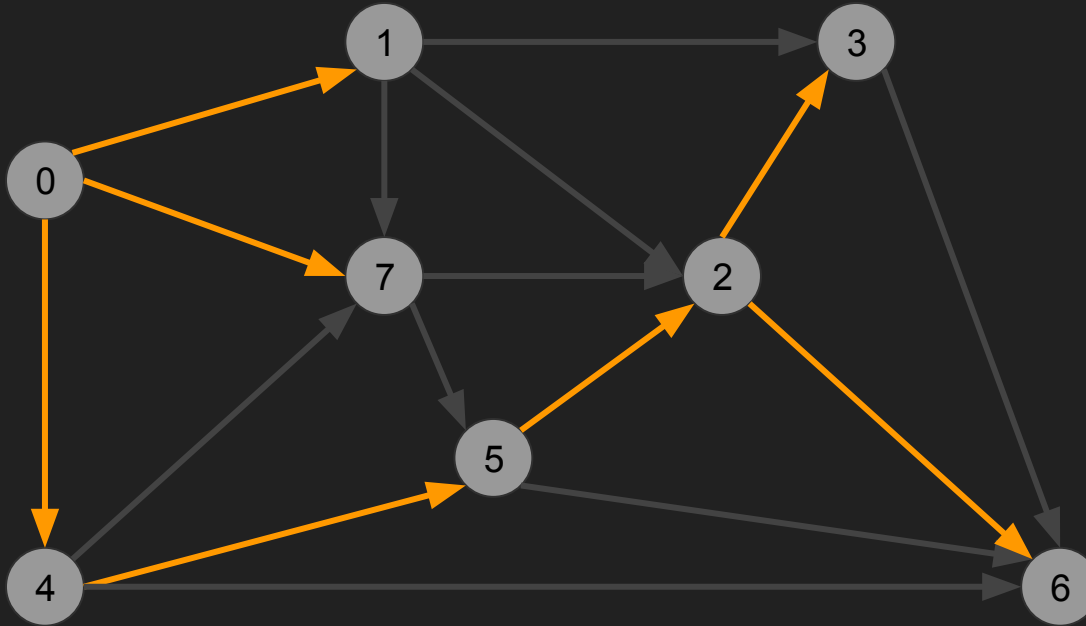
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

So if I highlight all of the 'EdgeTo' connections, I have the shortest paths



v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

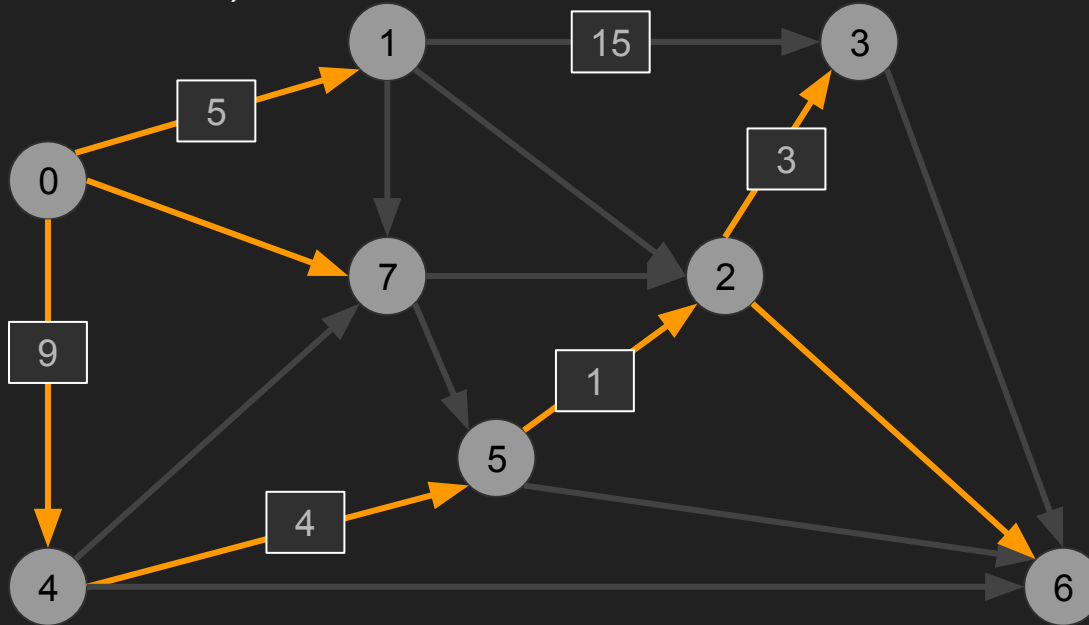
So if I highlight all of the 'EdgeTo' connections, I have the shortest paths to all other nodes from 0 (Single source shortest paths)



v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

As an example, the shortest path to '3' would be 0->4->5->2->3 (total of 11)

(We can see 0->1->3 would be a cost of 18.0, even though there are less connections)



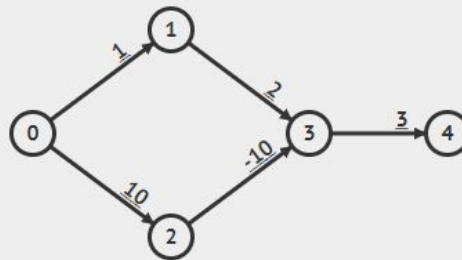
v	distTo	EdgeTo
0	0	-
1	5.0	0->1
2	14.0	5->2
3	17.0	2->3
4	9.0	0->4
5	13.0	4->5
6	25.0	2->6
7	8.0	0->7

Dijkstra's Complexity Analysis

- Intuitively, we saw that we explored every vertex, and each of its neighbors.
 - This implies $O(E+V)$ time, similar to BFS.
- The actual complexity depends on the data structure used.
 - What do I mean?
 - Well we need to do 'lookups' to find minimum cost vertices (i.e. the greedy part)
 - For V however, we are actually doing $V \cdot \log(V)$ searches if using a priority heap, so we get $O((E+V)\log(V))$ typically.
- This can be improved see [\[More info\]](#)

Dijkstra -- interactive demo

<https://visualgo.net/en/sssp?slide=1>



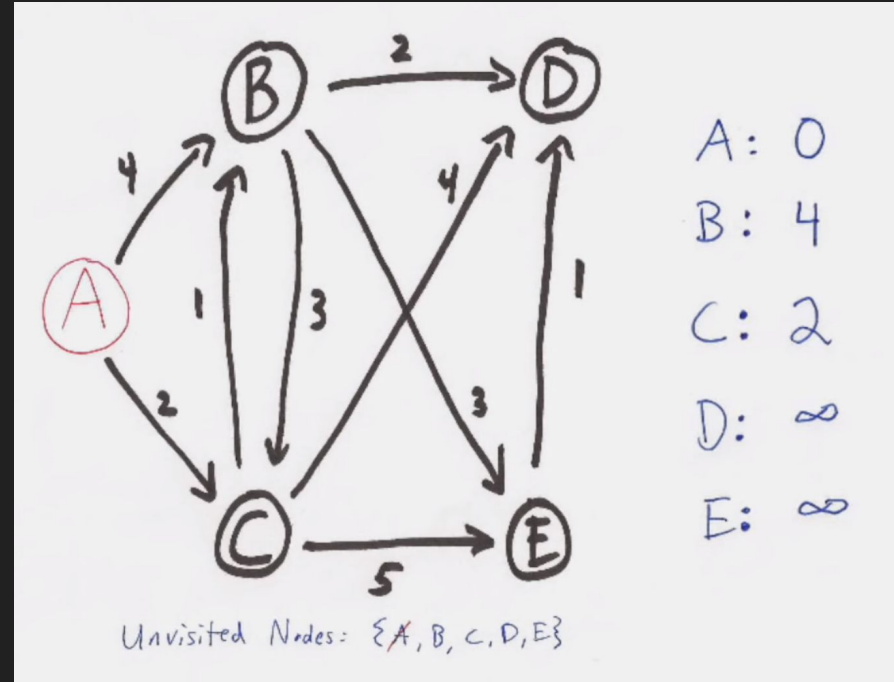
e-Lecture Example (auto play until done)
OriginalDijkstra(0)

#edge_processed = 7, $O((V+E) \log V) = 28$.
This is the SSSP spanning tree from source vertex 0.

```
show warning if the graph has -ve weight edge
initSSSP, pre-populate PQ
while !PQ.empty() // PQ is a Priority Queue
  for each neighbor v of u = PQ.front(), PQ.pop()
    relax(u, v, w(u, v)) + update PQ
// ch4_05_dijkstra.cpp/java, ch4, CP3
```

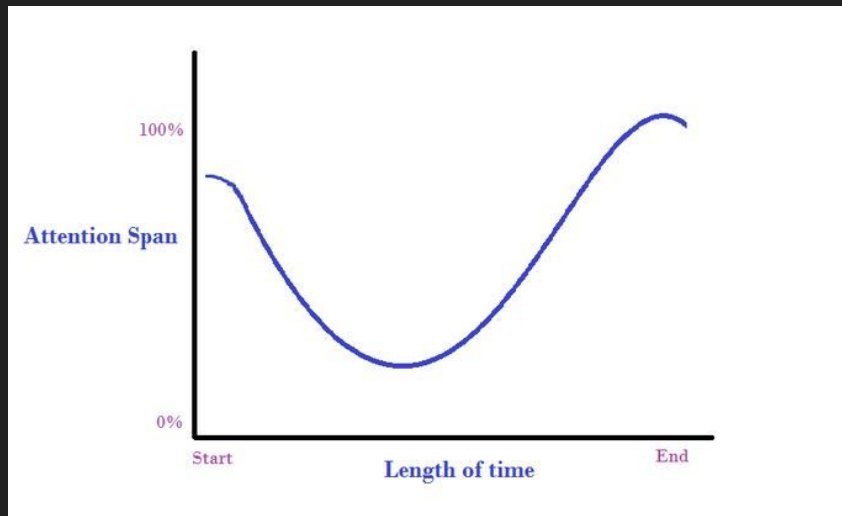
(FYI) Video Review of Dijkstra's Algorithm

- (Here as backup if you need to see it again)
- https://www.youtube.com/watch?v=_IHSawdgXpl



Short 5 minute break

- 3 hours is a long time.
- I will try to never lecture for more than half of that time without some sort of 'break' or transition to an in-class activity/lab.
- Use this time to stretch, check your phones, eat/drink something, etc.

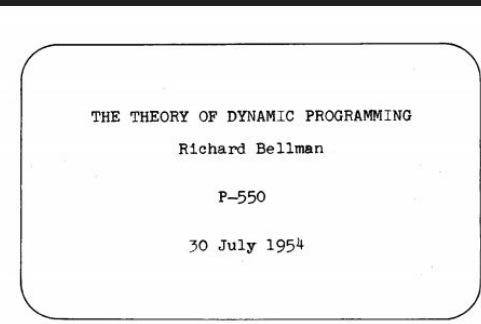


Dynamic Programming

Breaking problems into independent subproblems

What is Dynamic Programming?

- It *sounds* like something extra fancy, but it really is just another technique for solving problems.
- The idea was really pioneered by Richard Bellman [\[reference\]](#)
- Think of Dynamic programming as
 - 'planning over time'



Dynamic Programming - Defined

- It's a technique for taking complicated problems (i.e. those that take exponential or factorial time) and reducing them to polynomial time
- It involves breaking a problem down into smaller sub-problems.
 - Usually this can be easily done using extra storage

Dynamic Programming - Applications

- Almost everywhere--let's take a look at one problem to start!



Fibonacci -- our old friend



Fibonacci Sequence

0, 1, 1, 2, 3, 5, 8, 13, ...

- Computes the next number given the previous two results.
- Recursive formulation
 - Can be computed relatively easily, and the initial solution does not need any extra storage.

$$F_0 := 0; \quad F_1 := 1;$$

$$F_n = F_{n-1} + F_{n-2}, \text{ for all } n \geq 2.$$

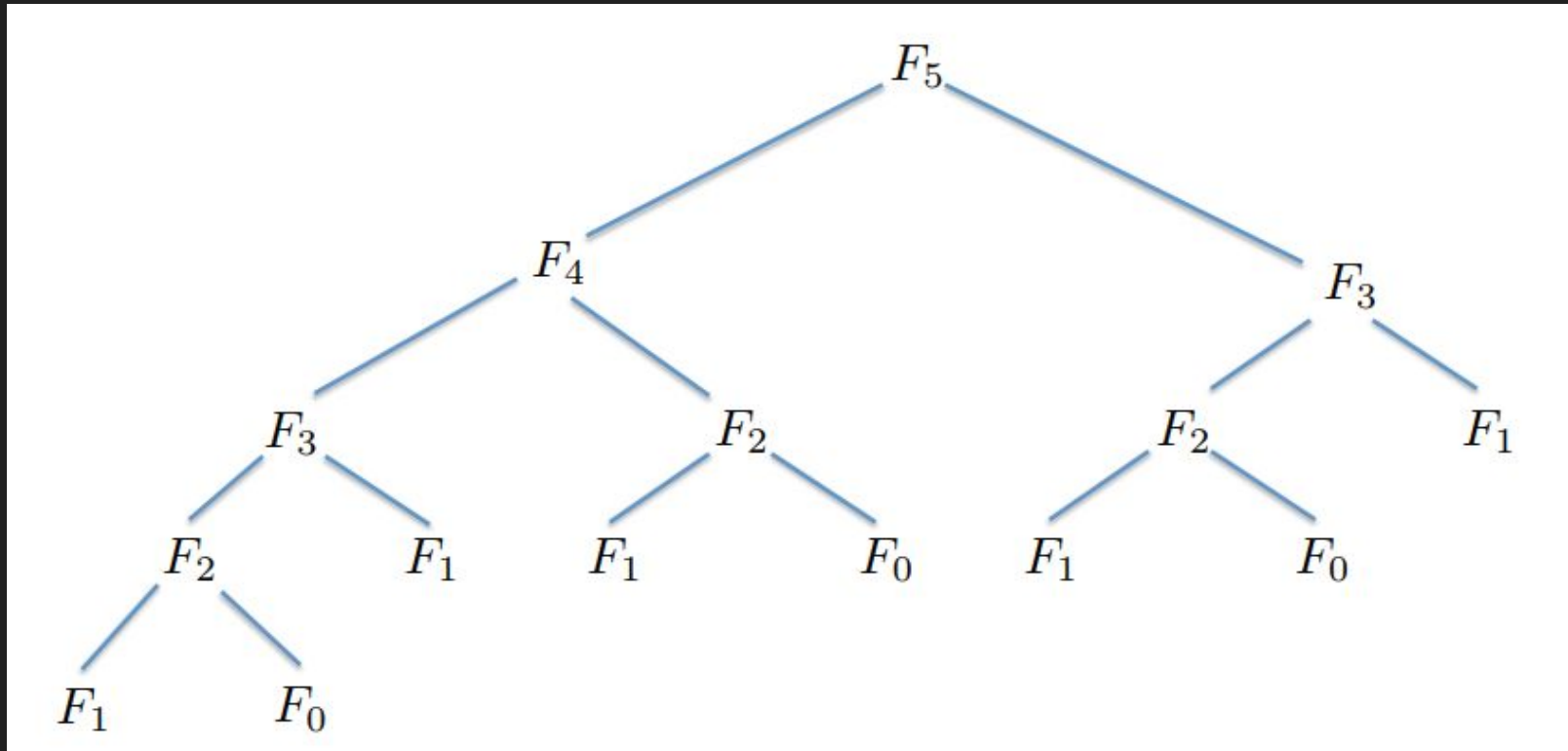
naive_fibo(n):

if $n = 0$: return 0

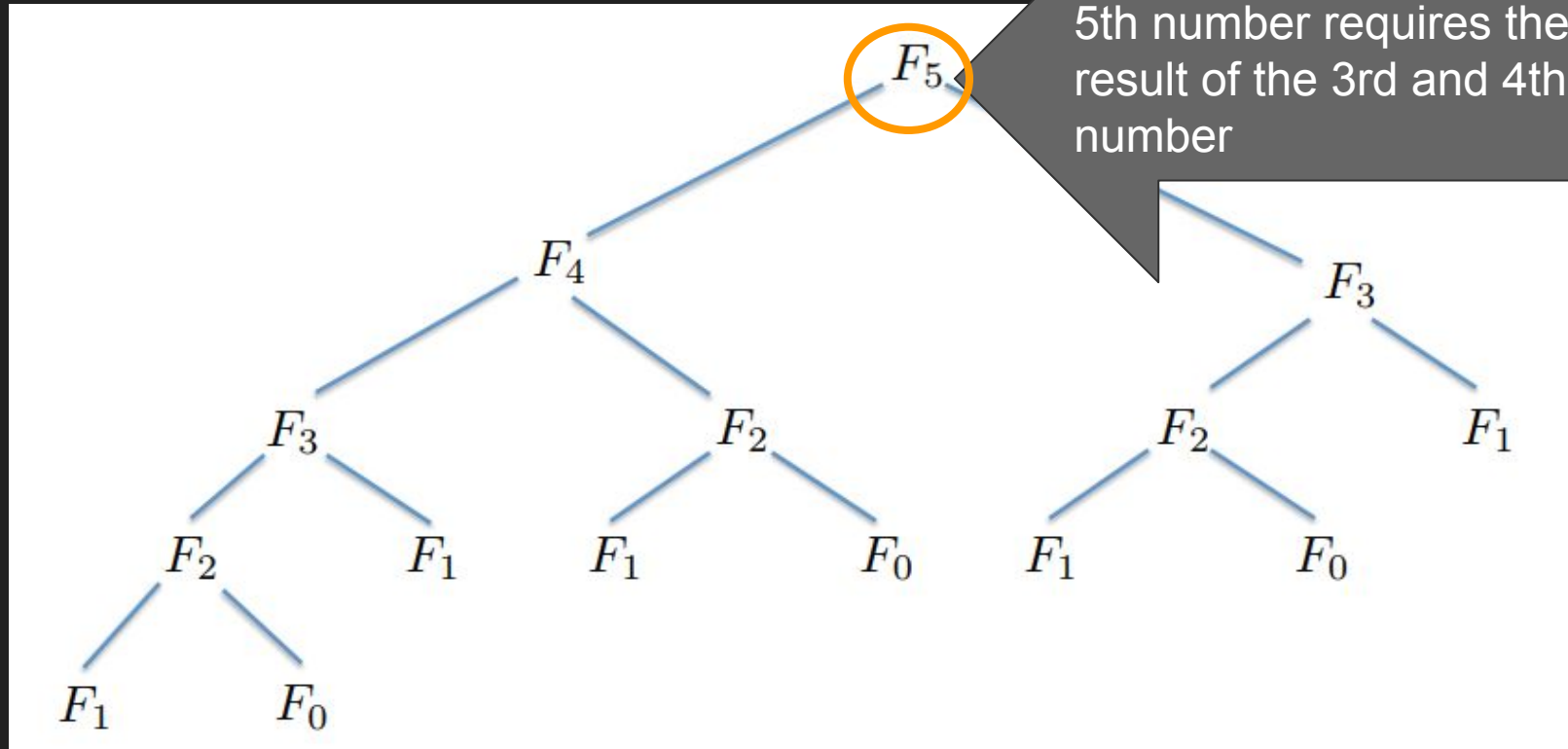
else if $n = 1$: return 1

else: return naive_fibo($n - 1$) + naive_fibo($n - 2$).

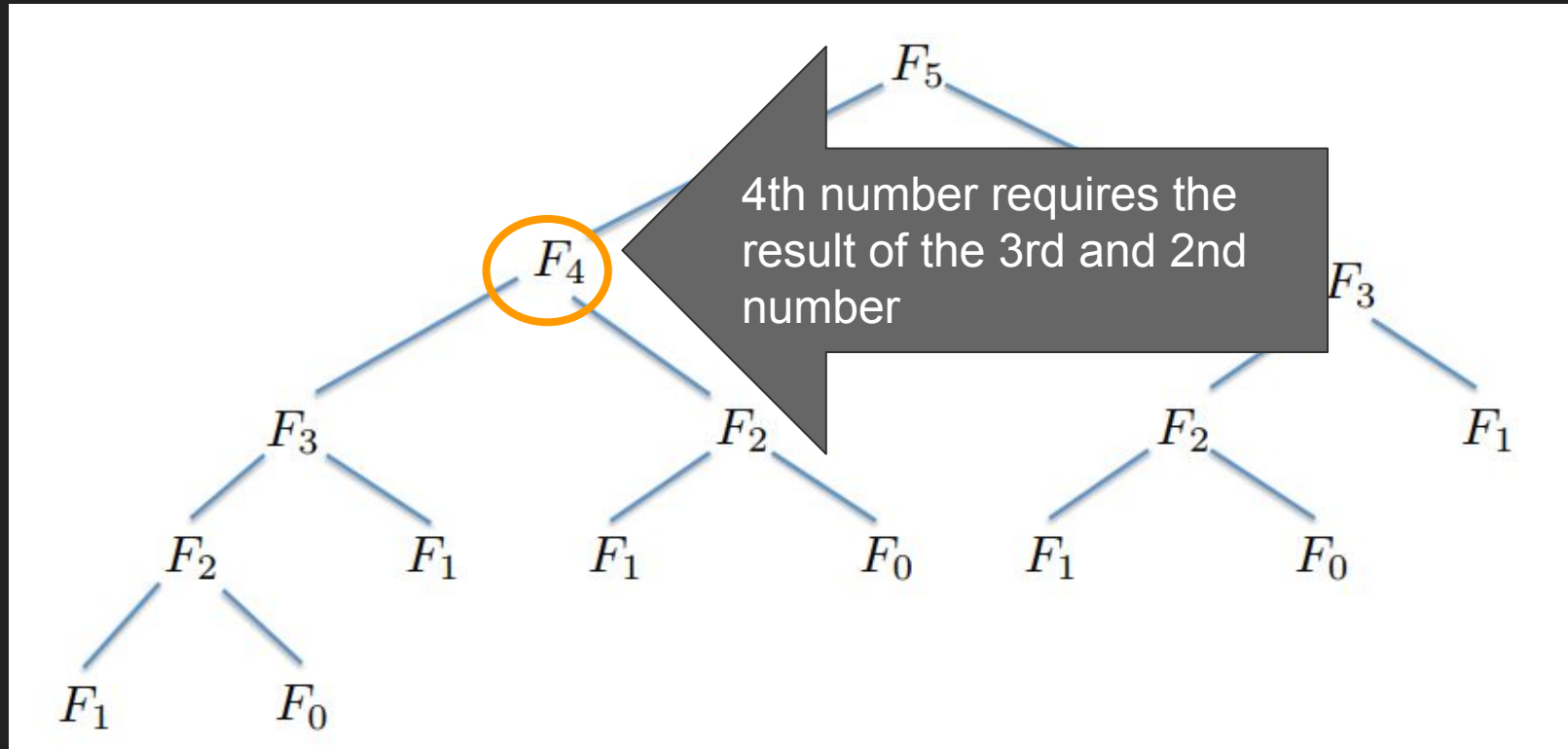
Computing the 5th fibonacci number



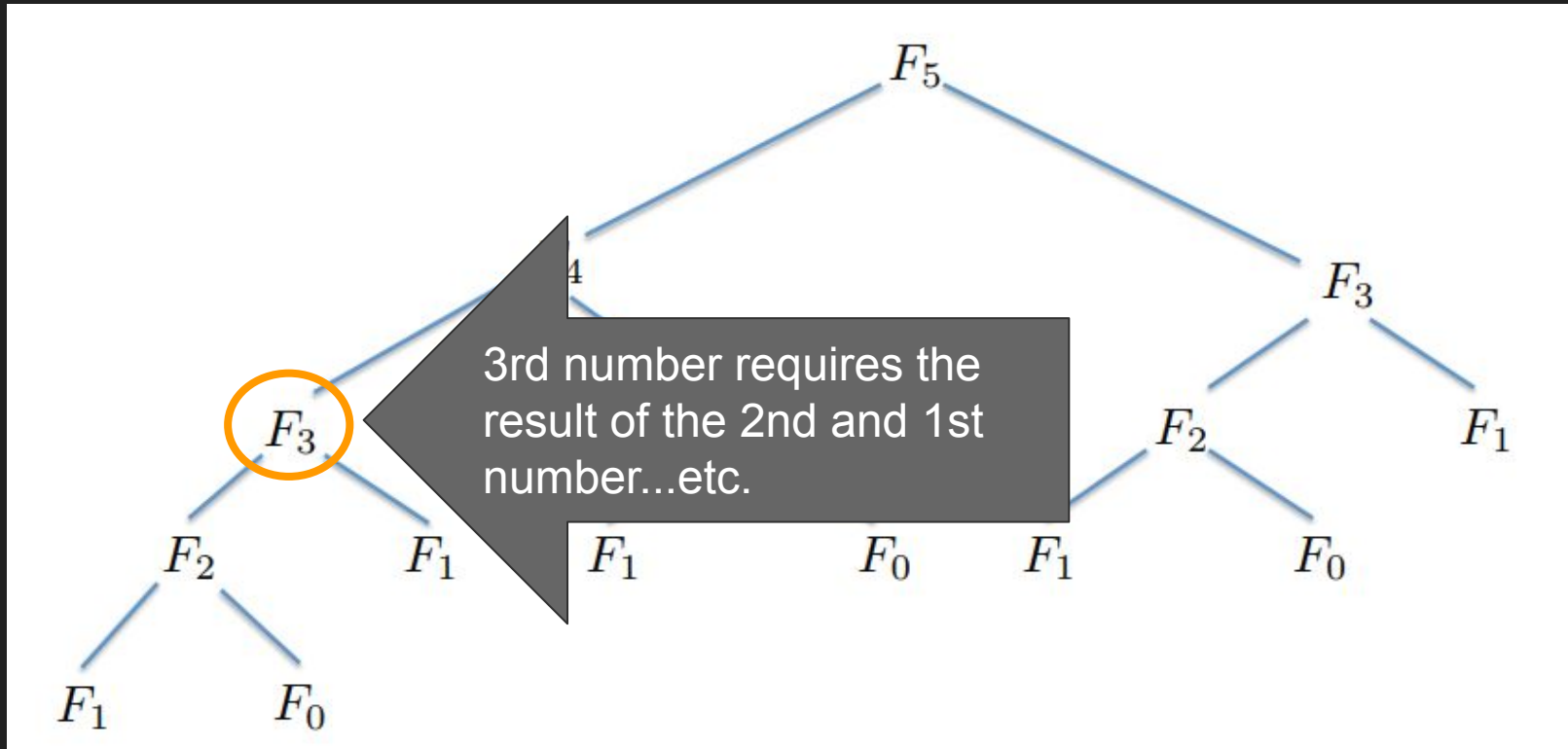
Computing the 5th fibonacci number



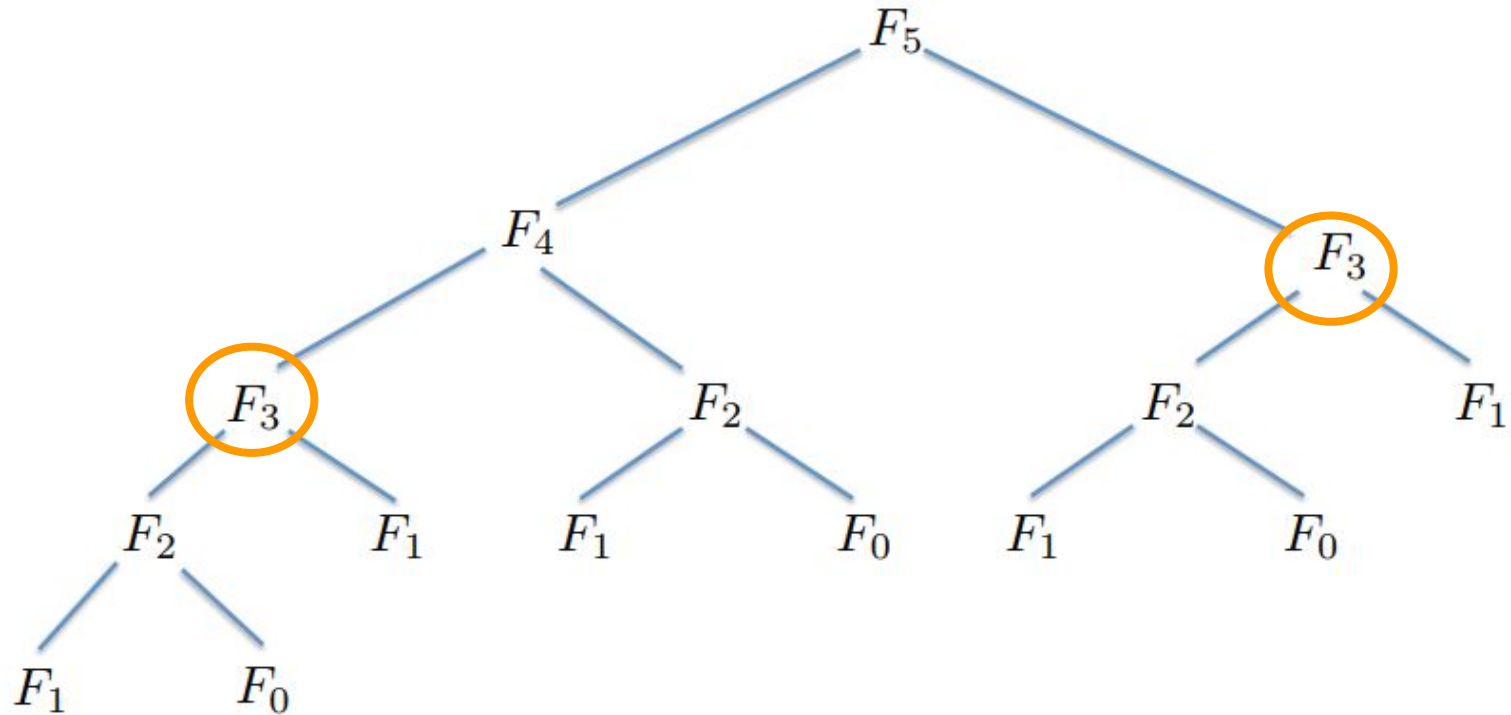
Computing the 5th fibonacci number



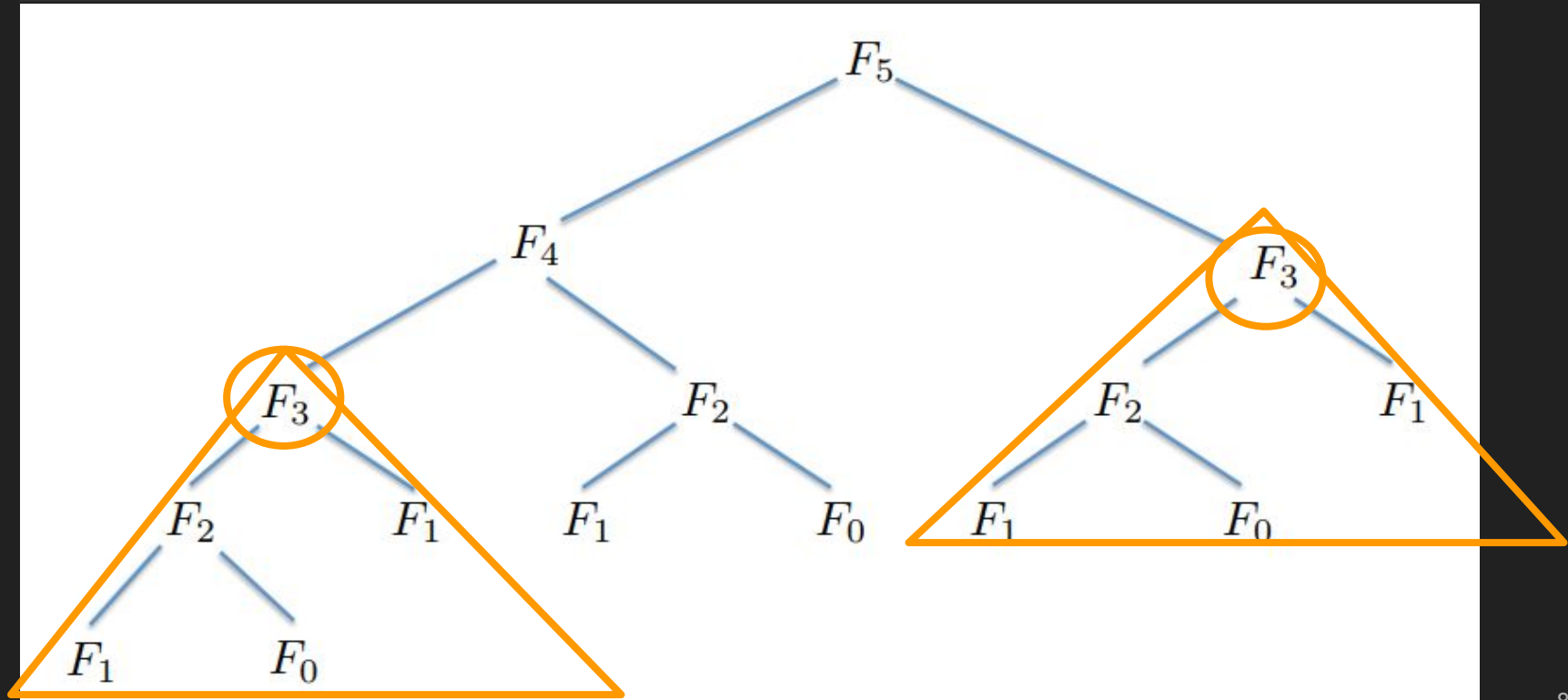
Computing the 5th fibonacci number



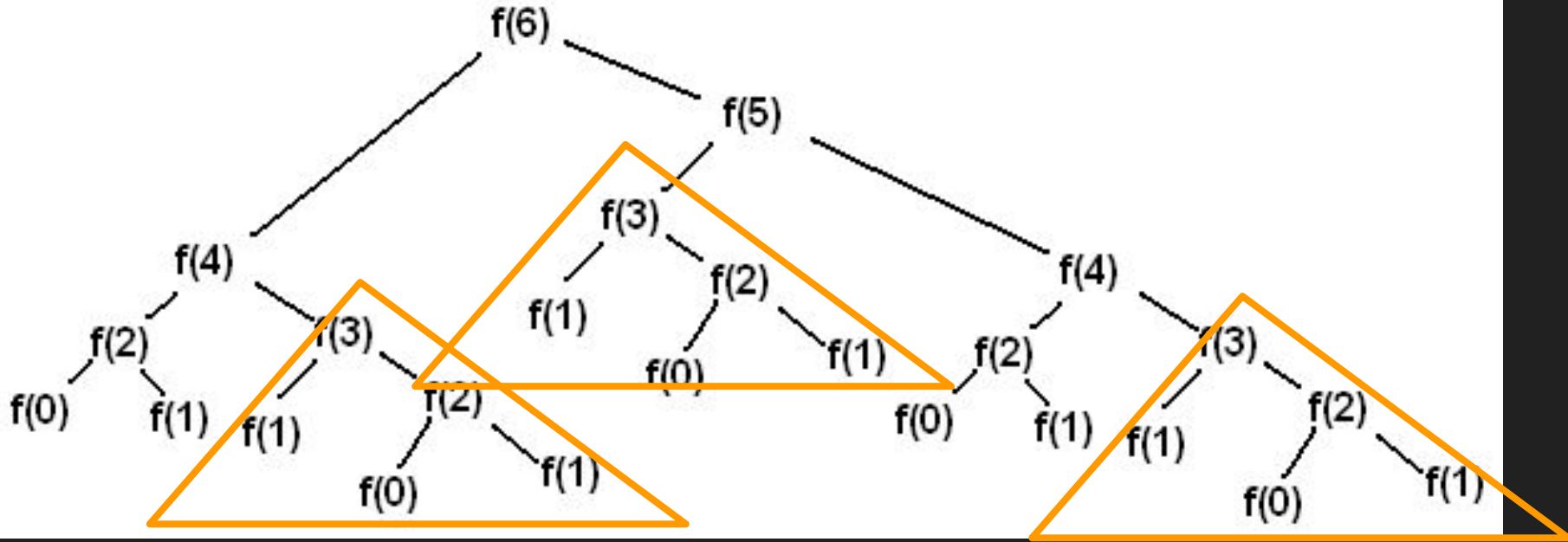
Code Here we recompute F_3 two times



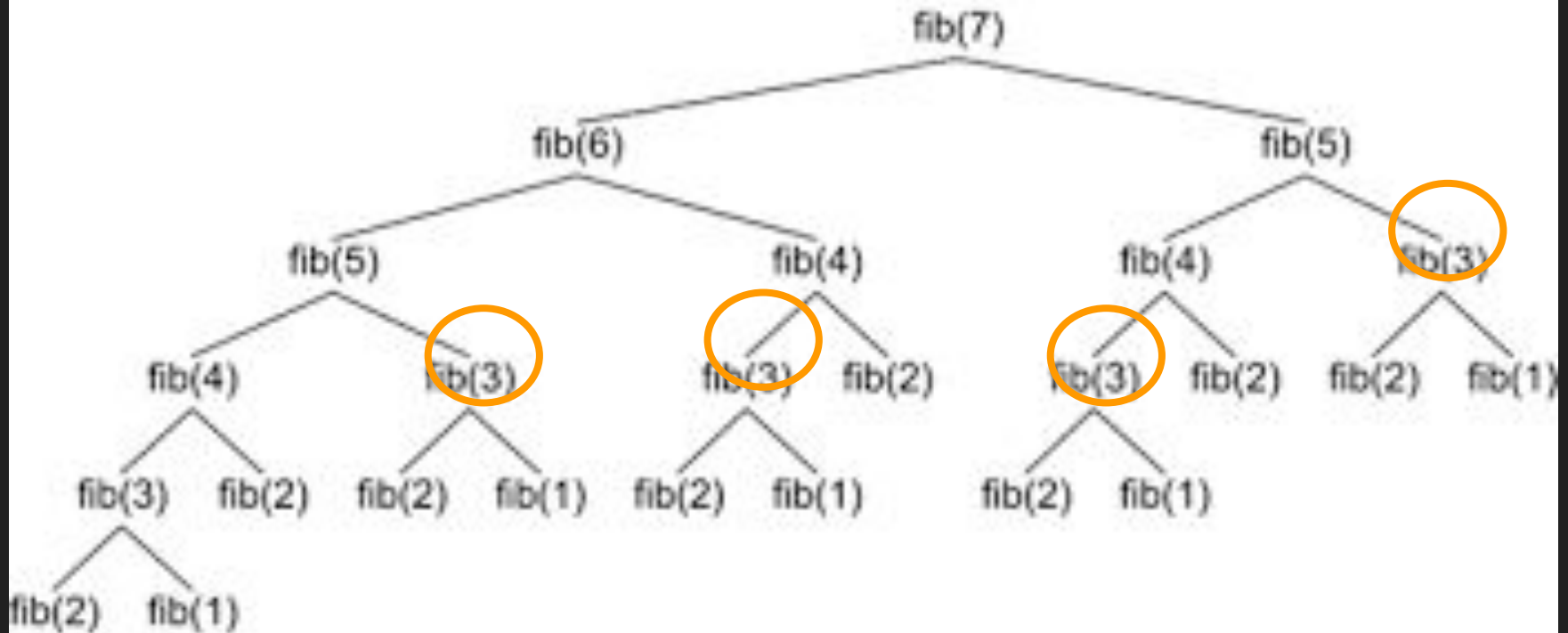
Co But we are also computing the subtree each time...



Bigger fibonacci numbers, mean bigger trees, and more repeats



You get the idea (and I am picking on a small fib of 3, we could choose 4, 5, etc.).



Fibonacci Runtime Analysis

```
naive_fibo(n):
```

```
    if  $n = 0$ : return 0
```

```
    else if  $n = 1$ : return 1
```

```
    else: return naive_fibo( $n - 1$ ) + naive_fibo( $n - 2$ ).
```

- Let's take a look at the recurrence
- We are trying to approach our base case ($n \leq 1$), so each call to fibonacci we approach this by $T(n-1)$ and $T(n-2)$.
- So we are *roughly* solving two sub-problems
 - It's fair to say $2T(n-2) + c$
 - (c is some constant factor for say adding numbers, checking if $n==1$ or $n==0$, etc.)

$$T(n) = T(n-1) + T(n-2) + c$$

$$2T(n-2) + c$$

Fibonacci Runtime Analysis

```
naive_fibo(n):
```

```
    if  $n = 0$ : return 0
```

```
    else if  $n = 1$ : return 1
```

```
    else: return naive_fibo( $n - 1$ ) + naive_fibo( $n - 2$ ).
```

$$2T(n - 2) + c$$

- So we are solving ‘two subproblems’ each time we recurse, and these start to add up
 - $F(n-2)$ twice
 - $F(n-3)$ three times
 - $F(n-4)$ four times...
 - $F(n-5)$ five times...
 - etc.
- Yikes, so the intuition here is that each time we solve a bigger fibonacci number we need to solve all sub-problems one more time.
 - And all sub-problems of those ($n-1$) again..
 - And then all sub-problems of those....

This indicates exponential growth $O(2^n)$
This is not good!

Fibonacci Runtime Analysis

```
naive_fibo(n):
```

```
    if  $n = 0$ : return 0
```

```
    else if  $n = 1$ : return 1
```

```
    else: return naive_fibo( $n - 1$ ) + naive_fibo( $n - 2$ ).
```

$$2T(n - 2) + c$$

How do we do better?

This indicates exponential growth $O(2^n)$
This is not good!

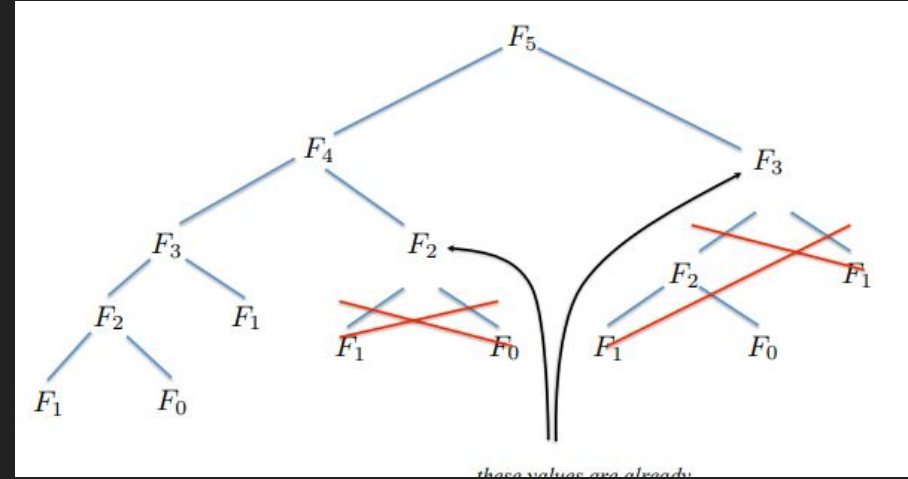
- So we are solving ‘two subproblems’ each time we recurse, and these start to add up
 - $F(n-2)$ twice
 - $F(n-3)$ three times
 - $F(n-4)$ four times
 - $F(n-5)$ five times.
 - etc.
- Yikes, so the intuition is that each time we solve a bigger fibonacci number we need to solve all sub-problems one more time.
 - And all sub-problems of those ($n-1$) again..
 - And then all sub-problems of those....

Memoization

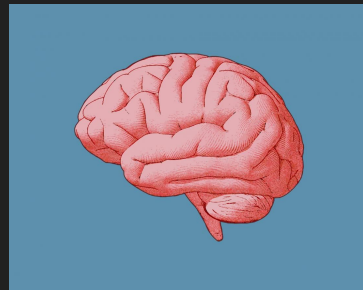
Storing previously computed results

The problem

- We want to avoid recomputing any values for which we have already computed solutions to
- The solution is known as 'memoization'

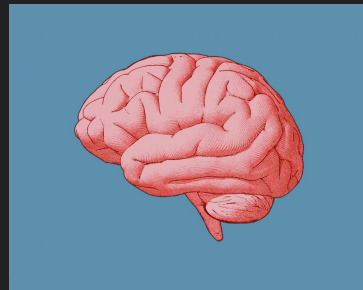


Memoization (1/2)



- Memoization is the art of committing something to memory
- So in computer science this means saving our results to some other data structure.
 - So we need to store our previously computed Fibonacci numbers
 - What data structure should we use? Our choices?
 - Array - if we can use an unsigned integer as a key
 - Linked List (if we need to expand)
 - Hash Map - Typically for any non-integer key
 - **Question to the audience:** For Fibonacci--what should we use?

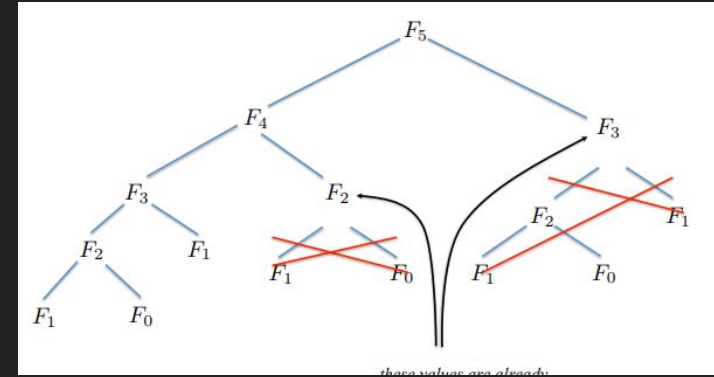
Memoization (2/2)



- Memoization is the art of committing something to memory
- So in computer science this means saving our results to some other data structure.
 - So we need to store our previously computed Fibonacci numbers
 - What data structure should we use? Our choices?
 - Array - if we can use an unsigned integer as a key
 - Linked List (if we need to expand)
 - Hash Map - Typically for any non-integer key
 - **Question to the audience:** For Fibonacci--what should we use?
 - Plain old, built-in C-style array will do!

Fibonacci - With Memoization

- This time we will use a little bit of storage to compute every fibonacci number from 1...N
- Then each time we compute a fibonacci number:
 - We will check if we have previously computed a value
 - If we have then we are done
 - If not, we still may be able to utilize some of the previous results
- This ends up being....faster (you will try in lab in moments!)
 - The time complexity is now $O(n)$ and the space complexity will be $O(n)$ (instead of $O(1)$)



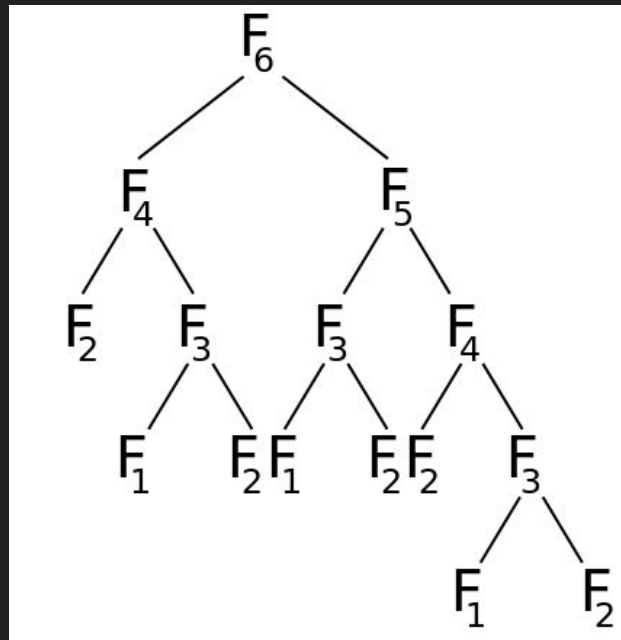
Fibonacci recursive versus Fibonacci Memoized

- Fibonacci recursive

- Time: $O(2^n)$
 - Why?
 - Each level in the tree grows *about twice as big*. (The tree is not perfectly balanced)
- Space: $O(n)$
 - Why?
 - Well the depth of the tree is proportional to how many elements we have, so we need at least that much stack space for our tree.

- Fibonacci memoized

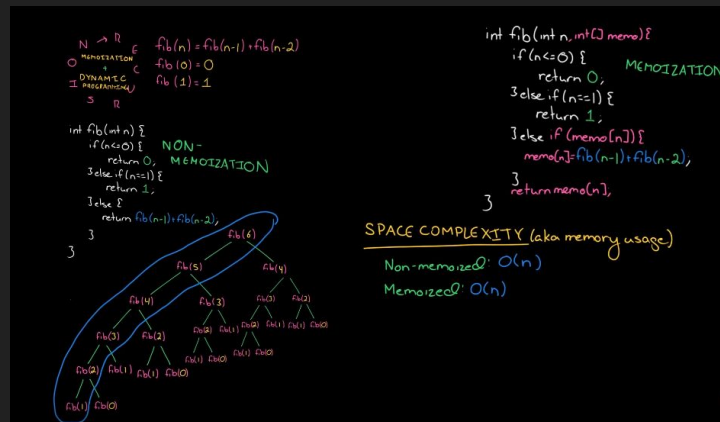
- Time: $O(n)$
- Space: $O(n)$ (for our table of pre-computed results)



Computer Systems Feed

YOUR DAILY FEED

- (An article/image/video/thought injected in each class!)
- Algorithms: Memoization and Dynamic Programming
 - <https://www.youtube.com/watch?v=P8Xa2BitN3I&t=1s>
 - HackerRank - 11 minute video to watch later
 - With author of *Cracking The Coding Interview* with Gayle Laakmann McDowell



This lecture in summary

- Greedy algorithms can sometimes be the optimal strategy.
 - We saw an example of Dijkstra's Shortest Path
- Memoization is a technique part of dynamic programming that can leveraged to improve performance greatly (trading space for time)

Algorithm, Data Structure, and Proof Toolboxes

For this course, I want you to be able to see how each data structure and algorithm is different.

- For data structures learn how each restriction on how we organize our data causes tradeoffs
- For algorithms, think about the higher level technique

Algorithm Toolbox: Searches and Sorts

Comparison Sorts

Bubble Sort - $O(n^2)$

6 5 3 1 8 7 2 4

Swap adjacent elements and 'bubble' up element

Selection Sort - $O(n^2)$

5 3 4 1 2

Selection Sort

Search for minimum element and place in ordered position amongst unordered elements

Insertion Sort - $O(n^2)$

6 5 3 1 8 7 2 4

Select each element and place in its sorted position amongst all elements that have been previously placed

Heap Sort-
 $O(n \log_2(n))$

Divide and Conquer Sorts

Merge Sort- $O(n \log_2(n))$

Randomized Algorithms

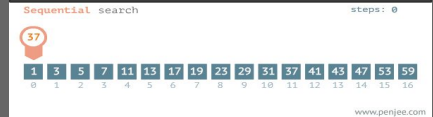
QuickSort- $\Theta(n \log_2(n))$
("theta")

$O(n^2)$ - in the worst case

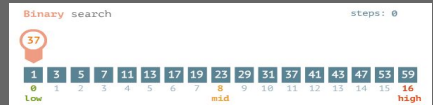
Searches

Linear Search - $O(n)$

Search and compare each element one at a time



Binary Search - $\log_2(n)$



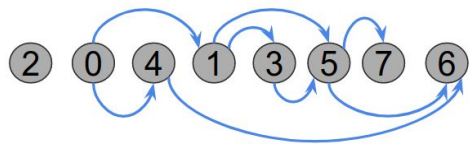
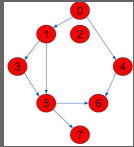
Search sorted data from midpoint, eliminate values less than or greater than 'element' you are search for each step until we match the mid

Algorithm Toolbox: Trees and Graphs

Tree Sorts

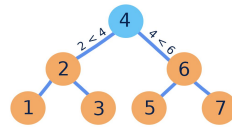
Topological Sort

Generates a possible linear ordering of nodes from a Tree by performing a DFS on each unvisited node. $O(|V|+|E|)$



Tree and/or Graph Searches

Binary Search Tree

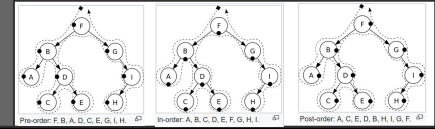


In Order Traversal: 1 2 3 4 5 6 7

Data structure containing a left and right child
 $O(\log_2(n))$ for search, insertion, and deletion

Depth-First Search (DFS)

Traverse graph (or tree) as far as possible in a direction, storing nodes in a stack. $O(|V|+|E|)$



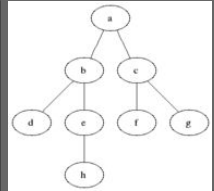
Shortest Path

Dijkstra's Shortest Path

Greedily take the shortest path from each node.

Breadth-first search (BFS)

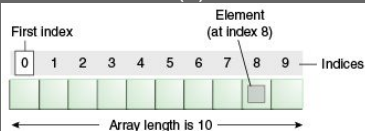
Traverse graph (or tree) as widely as possible (level-order traversal) storing each nodes neighbors in a queue. $O(|V|+|E|)$



Data Structure Toolbox: Fundamental Structures

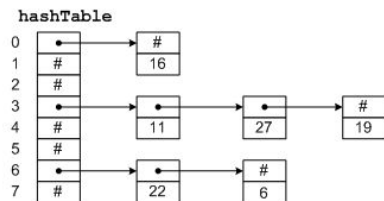
Associative Containers

Arrays - A contiguous block of memory, random access $O(1)$



HashMap (chained implementation) -

Associative Data Structure with key/value pairs and a 'hash function'



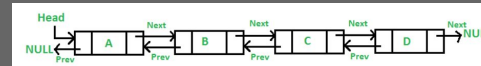
Sequence Containers

Linked Lists - A 'chain' of nodes, can traverse in one direction

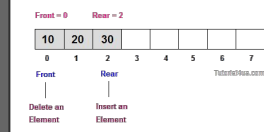


Doubly Linked Lists -

A 'chain' of nodes, can traverse in both directions



Queues - A First in, First out data structure (FIFO)

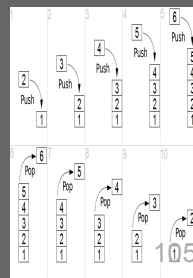


Priority Queues -

Uses a min-heap or max-heap and promotes min or max element to front of queue.

Stacks -

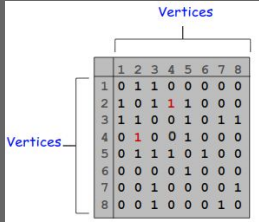
A Last in, last out data structure (LIFO)



Data Structure Toolbox: Tree and/or Graphs

Graph and Tree Data Structures

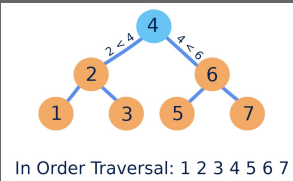
Adjacency Matrix -



2D array holding edge weights (or 0 if unconnected)

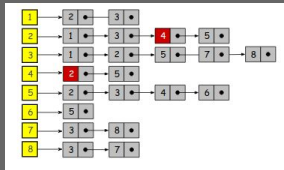
Space complexity of $O(|V|^2)$

Binary Tree



Data structure containing a left and right child
 $\Theta(\log_2(n))$ for search, insertion, and deletion

Adjacency List -



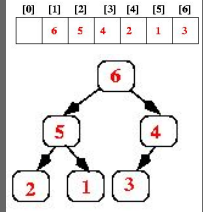
An array of lists or list of lists where each list indicates connectivity

Space complexity of $O(|V|+|E|)$

Heaps

Binary Heaps

Array-based structure to hold a complete binary tree



Proof

Toolbox

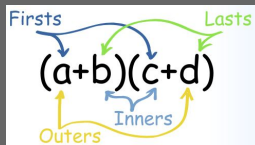
- Our tools so far!

Notation	Building Blocks
$\forall n$ - "for all" - "such that" $n \in \mathbb{Z}$ - "n is an element of the integers"	Definition - Something given, we can assume is true e.g. let $x = 7$
Proof Techniques	Proposition - A true or false statement e.g. $1+7 = 7$ FALSE $2+7 = 9$ TRUE
<ul style="list-style-type: none">• Proof by Case<ul style="list-style-type: none">◦ Enumerate or test all possible inputs• Proof by Induction<ul style="list-style-type: none">◦ Show that two cases hold• Proof by Invariant<ul style="list-style-type: none">◦ Step through 4 steps of algorithm• Big-O Analysis<ul style="list-style-type: none">◦ Prove run-time complexity	Predicate - A proposition whose truth depends on its input. It is a function that returns true or false. "P(n) ::= "n is a perfect square" P(4) thus is true, because 4 is a perfect square P(3) is false, because it is not a perfect square.
<ul style="list-style-type: none">• Recurrence<ul style="list-style-type: none">◦ Can be solved with Substitution Method• Recurrence Tree<ul style="list-style-type: none">◦ "A Visual Proof" (Somewhat informal)• Master Theorem<ul style="list-style-type: none">◦ Proven by definition• Substitution Method<ul style="list-style-type: none">◦ (Works for any recurrence)	

Math Toolbox

Mathematics

Multiplication



$$(a+b)(c+d) = ac + ad + bc + bd$$

Logs

Logs -

Usually we work in log base 2, i.e. $\log_2(n)$. The change of base formula is given below.

$$\log_a n = \frac{\log_b n}{\log_b a}$$

In this course we think about logs as 'halving' the number of our sub-problems (or search space).

Notation

Pi Production Notation

$$n! = \prod_{i=1}^n i.$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-2) \cdot (n-1) \cdot n$$

Big-O: $O(n)$ - "Worse Case Analysis or upper bound"

Big-Theta: Θ - "Average Case Analysis"

Big-Omega Ω - "Best Case or lower bound"

Factorial (!)

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Summation ("sigma")

$$\sum_{i=1}^n x_i = x_1 + x_2 + \dots + x_n$$

i = index of summation
 n = upper limit of summation

In-Class Activity

1. Complete the in-class activity from the schedule
 - a. (Do this during class, not before :))
2. Please take 2-5 minutes to do so
3. These make up a total of 5% of your grade
 - a. We will review the answers shortly



In-Class Activity or Lab (Enabled toward the end of lecture)

- [In-Class Activity link](#)
 - (This is graded)
 - This is an evaluation of what was learned in lecture.

For today's lab

- Implement Fibonacci recursively and using dynamic programming