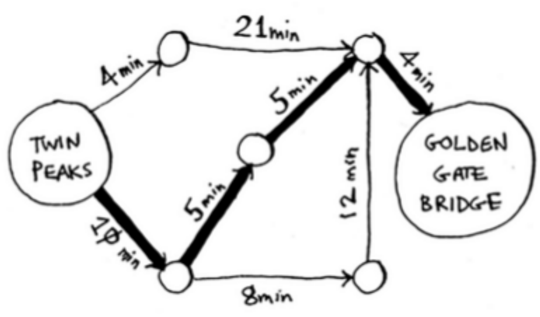
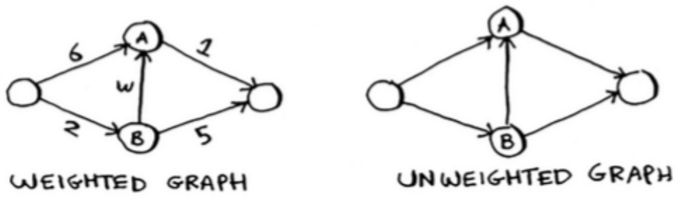
Dijkstras Algorithm

- Dijkstra's algorithm, lets you answer "What's the shortest path to X?" for weighted graphs.

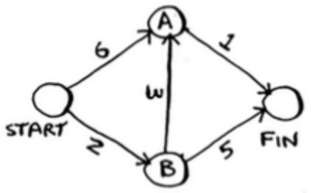


- Breadth-first search will find you the path with the fewest steps. What if you want the fastest path instead? You can do that fastest with a different algorithm called Dijkstra's algorithm.
- Dijkstra's algorithm has four steps:
  1. Find the cheapest node. This is the node you can get to in the least amount of time.
  2. Check whether there's a cheaper path to the neighbors of this node. If so, update their costs.
  3. Repeat until you've done this for every node in the graph.
  4. Calculate the final path. (Coming up in the next section!)
- Dijkstra's algorithm, each edge in the graph has a number associated with it. These are called weights. A graph with weights is called a weighted graph. A graph without weights is called an unweighted graph.

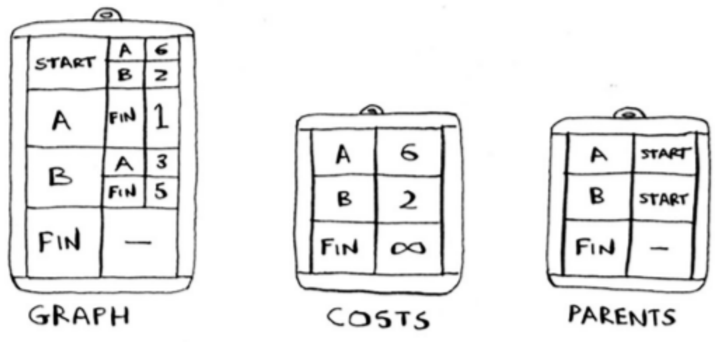


- Dijkstra's algorithm only works with directed acyclic graphs, called DAGs for short. In other words, Dijkstra's algorithm can not work with recurcive cycle graphs.
- You can't use Dijkstra's algorithm if you have negative-weight edges. Negative-weight edges break the algorithm.

Implementation



To code this example, you'll need three hash tables.



GRAPH

START	A	6
	B	2
A	FIN	1
B	A	3
	FIN	5
FIN	—	

COSTS

A	6
B	2
FIN	∞

PARENTS

A	START
B	START
FIN	—

- You'll update the costs and parents hash tables as the algorithm progresses. First, you need to implement the graph:

```
graph = {}
```

- This time, we need to store the neighbors and the cost for getting to that neighbor. For example, Start has two neighbors, A and B. How do you represent the weights of those edges? Why not just use another hash table?

```
graph["start]["a"] = 6
graph["start]["b"] = 2
graph["a"]["fin"] = 1
graph["b"]["a"] = 3
graph["b"]["fin"] = 5
graph["fin"] = {}
```

- So graph["start"] is a hash table. You can get all the neighbors for Start like this:

```
>>> print graph["start"].keys()
["a", "b"]
```

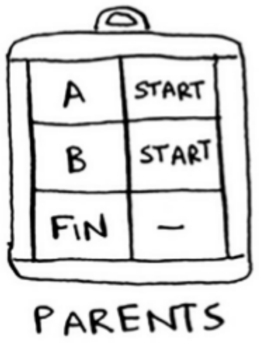
- Next you need a hash table to store the costs for each node. The cost of a node is how long it takes to get to that node from the start. If you don't know the cost yet, you put down infinity. Can you represent infinity in Python? Turns out, you can:

```
infinity = float("inf")
```

- Here's the code to make the costs table:

```
infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

- You also need another hash table for the parents:



A	START
B	START
FIN	—

PARENTS

- Here's the code to make the hash table for the parents:

```
parents = {}
parents["a"] = "start"
parents["b"] = "start"
parents["fin"] = None
```

- Finally, you need an array to keep track of all the nodes you've already processed, because you don't need to process a node more than once:

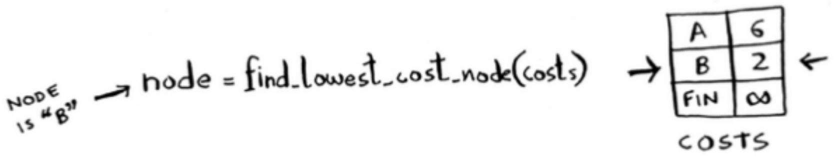
```
processed = []
```

- I'll show you the code first and then walk through it. Here's the code:

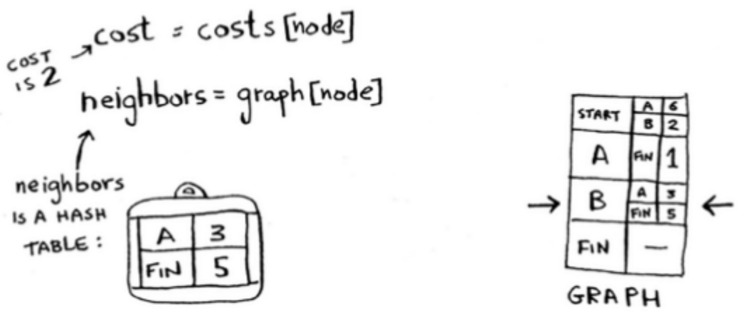
```
                                                      Find the lowest-cost node
node = find_lowest_cost_node(costs) ◄··········· that you haven't processed yet.
while node is not None: ◄··········· If you've processed all the nodes, this while loop is done.
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys(): ◄··········· Go through all the neighbors of this node.
        new_cost = cost + neighbors[n]          If it's cheaper to get to this neighbor
        if costs[n] > new_cost:◄··············· by going through this node ...
            costs[n] = new_cost  ◄········· ... update the cost for this node.
            parents[n] = node  ◄··········· This node becomes the new parent for this neighbor.
    processed.append(node) ◄··············· Mark the node as processed.
    node = find_lowest_cost_node(costs) ◄······ Find the next node to process, and loop.
```

- First, let's see this `find_lowest_cost_node` algorithm code in action:

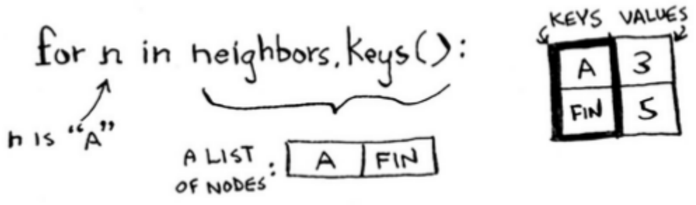
Find the node with the lowest cost.



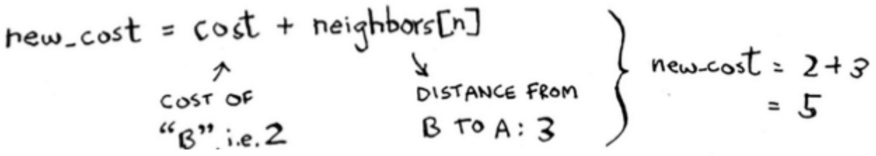
Get the cost and neighbors of that node.



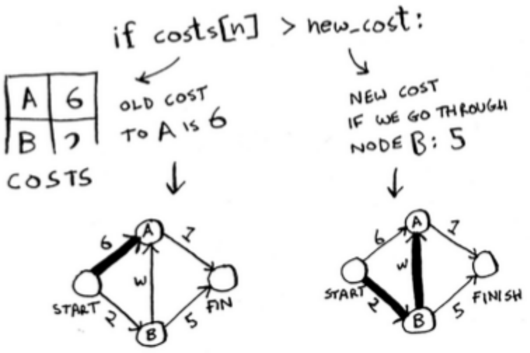
Loop through the neighbors.



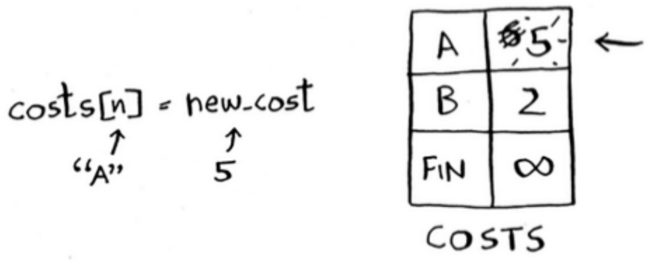
Each node has a cost. The cost is how long it takes to get to that node from the start. Here, you're calculating how long it would take to get to node A if you went Start > node B > node A, instead of Start > node A.



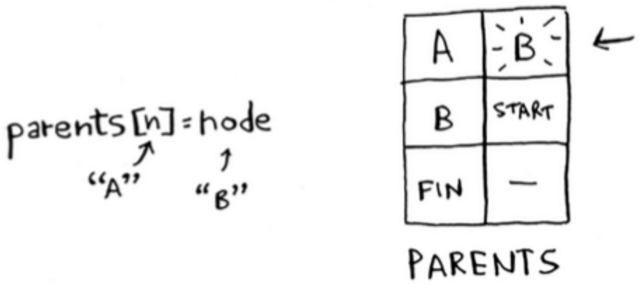
Let's compare those costs.



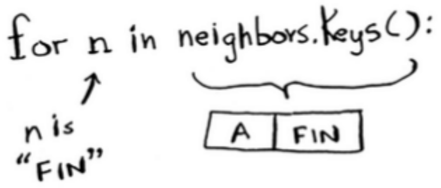
You found a shorter path to node A! Update the cost.



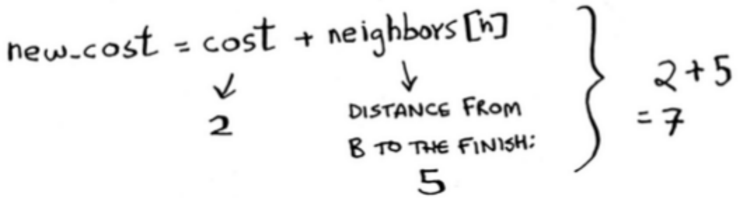
The new path goes through node B, so set B as the new parent.



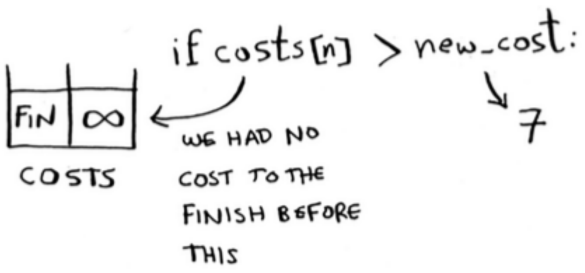
Ok, you're back at the top of the loop. The next neighbor `for` is the Finish node.



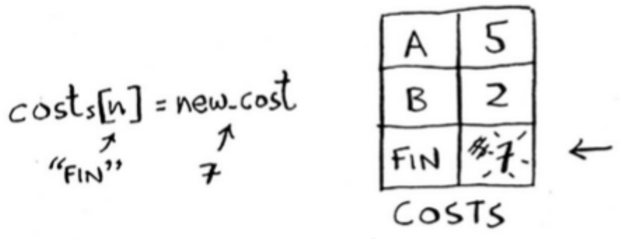
How long does it take to get to the finish if you go through node B?

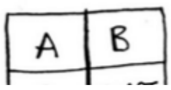

new_cost = cost + neighbors[n]
2
DISTANCE FROM B TO THE FINISH: 5
2+5 = 7

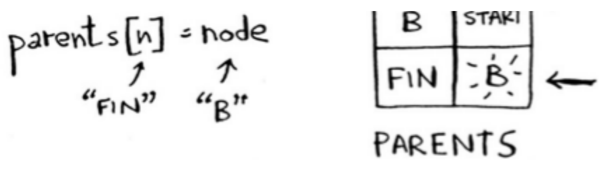
It takes 7 minutes. The previous cost was infinity minutes, and 7 minutes is less than that.



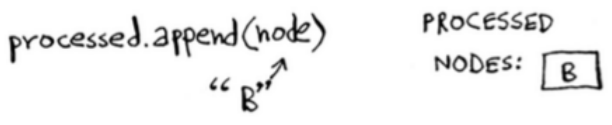
Set the new cost and the new parent for the Finish node.



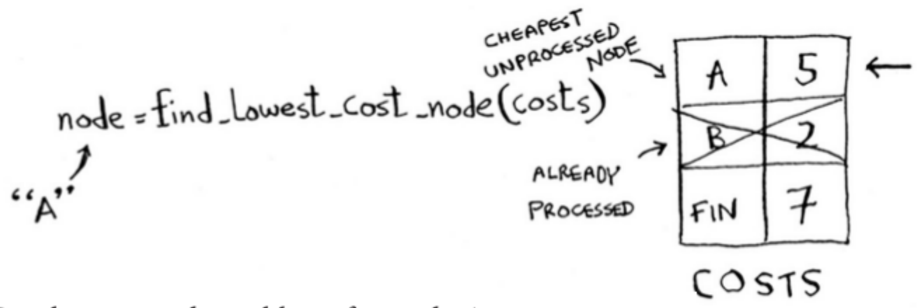




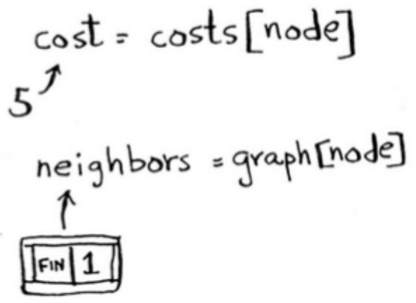
Ok, you updated the costs for all the neighbors of node B. Mark it as processed.



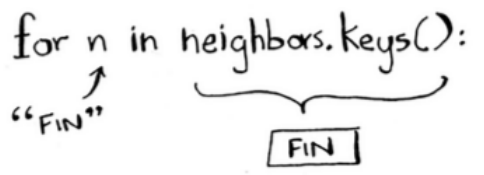
Find the next node to process.


node = find_lowest_cost_node(costs)
"A"
CHEAPEST UNPROCESSED NODE
ALREADY PROCESSED
A | 5
~~B | 2~~
FIN | 7
COSTS

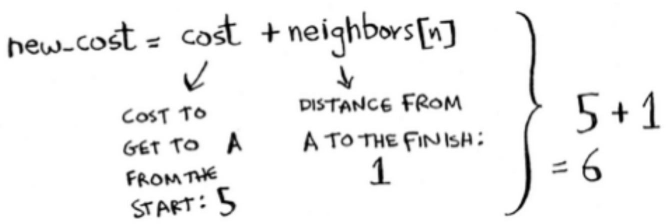
Get the cost and neighbors for node A.

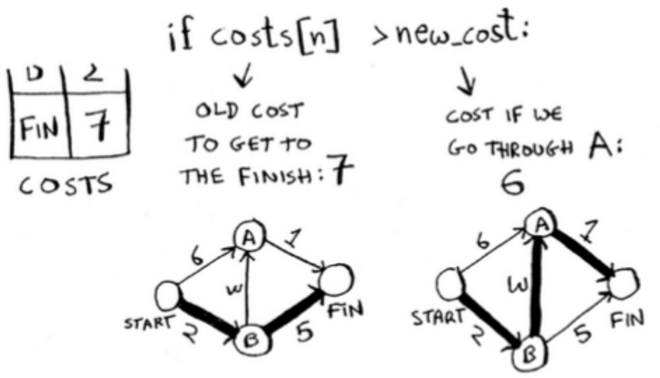


Node A only has one neighbor: the Finish node.

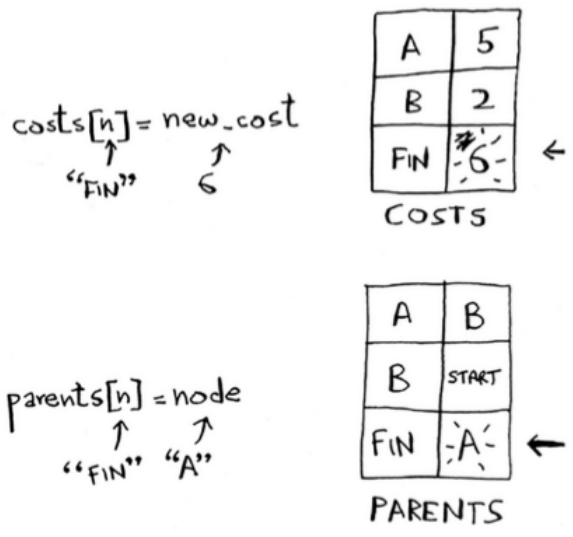


Currently it takes 7 minutes to get to the Finish node. How long would it take to get there if you went through node A?

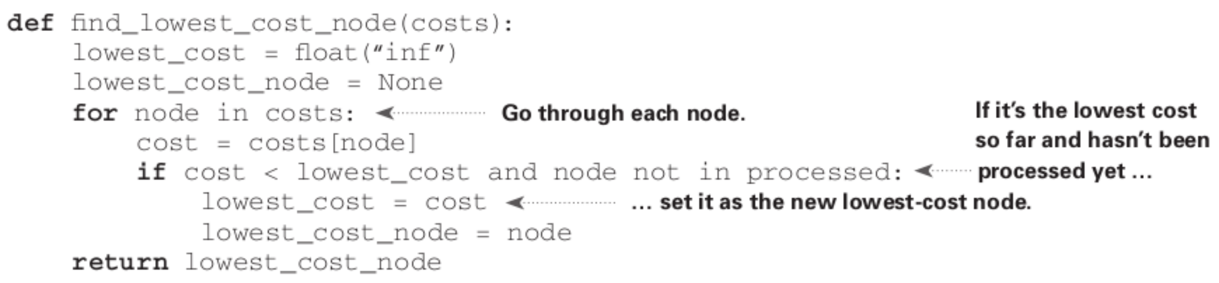

new_cost = cost + neighbors[n]
COST TO GET TO A FROM THE START: 5
DISTANCE FROM A TO THE FINISH: 1
5 + 1
= 6



It’s faster to get to Finish from node A! Let’s update the cost and parent.



- Once you've processed all the nodes, the algorithm is over. I hope the walkthrough helped you understand the algorithm a little better. Finding the lowest-cost node is pretty easy with the `find_lowest_cost_node` function. Here it is in code:



```
def find_lowest_cost_node(costs):
    lowest_cost = float(“inf”)
    lowest_cost_node = None
    for node in costs:
        cost = costs[node]
        if cost < lowest_cost and node not in processed:
            lowest_cost = cost
            lowest_cost_node = node
    return lowest_cost_node
```

Go through each node.

If it’s the lowest cost so far and hasn’t been processed yet …

… set it as the new lowest-cost node.

Recap

- Breadth-first search is used to calculate the shortest path for an unweighted graph.
- Dijkstra’s algorithm is used to calculate the shortest path for a weighted graph.
- Dijkstra’s algorithm works when all the weights are positive.
- If you have negative weights, use the Bellman-Ford algorithm.