

# EECS 281 Final Review Session

# Logistics

- Friday, June 25th, 2021
- Time slots: 9:00AM - 11:20AM, 4:00PM - 6:20PM (both EST)
- 24 MC, 2 Coding
- Upload your solutions to coding problems on Gradescope
- Try to read through most of the problems before starting
  - This is especially important for the handwritten coding problems
  - Reading through them before starting other parts of the exam lets your mind subconsciously work on it for the entire time!



# Agenda

- Hash Tables and Hash Collisions
- Binary Trees and Tree Algorithms
- AVL trees
- MST Algorithms
- Algorithm Families
- Backtracking + Branch & Bound
- Knapsack
- Dynamic Programming

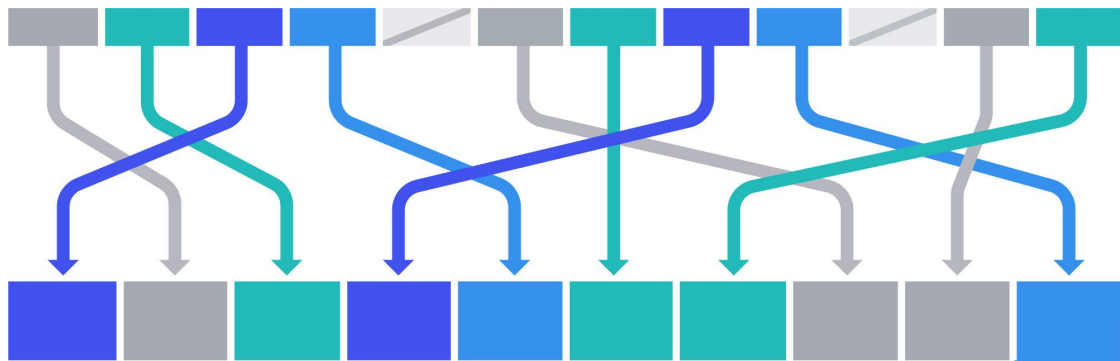




# Hash Tables and Hash Collisions

# Hash Tables - General Overview

- Aims for average case  $O(1)$  insert, lookup, and delete.
- Uses a **hash function** to map keys to an associated hash value (an integer)



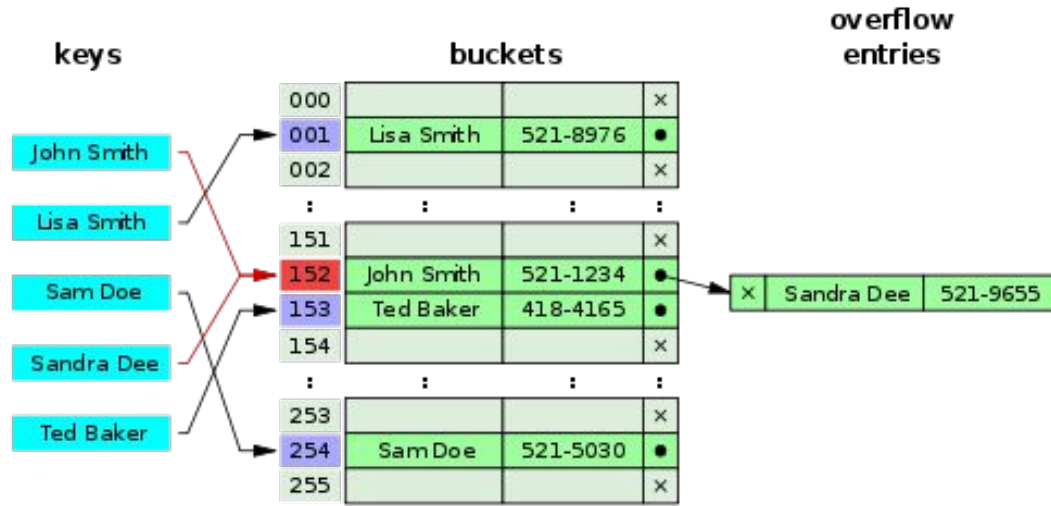
# Where is it good and where is it bad?

- Remember Project 3
- Hash tables are
  - **Good** at inserting, removing, and finding elements in a dataset with lots of distinct keys ( $O(1)$ )
  - **Bad** for  $>$ ,  $<$  queries on numbers
  - **Bad** when load factor is high ( $N/M$ )
    - $N$  are number of keys
    - $M$  is table size



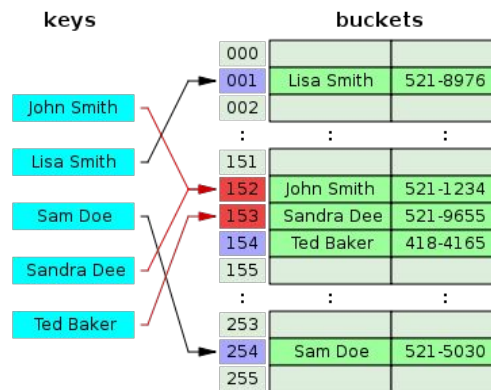
# Handling Collisions

- Separate Chaining (Can have load factor > 1)
  - Store a linked list in each bucket, append key value pair to the end



# Handling Collisions

- Open Addressing (Load factor must be  $< 1$ )
  - Linear Probing
    - $T(k) = (T(k) + i) \bmod M$  for  $i = 0, 1, 2, 3, 4 \dots$
  - Quadratic Probing
    - $T(k) = (T(k) + i^2) \bmod M$  for  $i = 0, 1, 2, 3, 4 \dots$
  - Double Hashing
    - $T(k) = (T(k) + i * T'(k)) \bmod M$  for  $i = 0, 1, 2, 3, 4 \dots$
    - $T'(k) = q - (T(k) \bmod q)$  for some prime  $q < M$





# Question - Resolution with Double Hashing

Consider a hash table storing keys that handles collision with **double hashing**:

- $M = 11$
- $T(k) = k \bmod M$
- $T'(k) = 7 - (k \bmod 7)$
- On collision:  $T(k) = (T(k) + i * T'(k)) \bmod M$  for  $i = 0, 1, 2, 3, 4...$

After inserting 35, 22, 44, 33 what would be the index of 33?



# Question - Resolution with Double Hashing

- On collision:  $T(k) = (T(k) + i * T'(k)) \bmod 11$  for  $i = 0, 1, 2, 3, 4...$
- $T(k) = k \bmod 11$
- $T'(k) = 7 - (k \bmod 7)$

0	1	2	3	4	5	6	7	8	9	10



# Question - Resolution with Double Hashing

Insert 35

- $T(k) = k \bmod 11$
- $T'(k) = 7 - (k \bmod 7)$
- $T(k) = (T(k) + i * T'(k)) \bmod 11$

0	1	2	3	4	5	6	7	8	9	10



# Question - Resolution with Double Hashing

Insert 35

$$T(35) = 35 \% 11 = 2$$

- $T(k) = k \bmod 11$
- $T'(k) = 7 - (k \bmod 7)$
- $T(k) = (T(k) + i * T'(k)) \bmod 11$

0	1	2	3	4	5	6	7	8	9	10



# Question - Resolution with Double Hashing

Insert 35

- $T(k) = k \bmod 11$
- $T'(k) = 7 - (k \bmod 7)$
- $T(k) = (T(k) + i * T'(k)) \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
		35								



# Question - Resolution with Double Hashing

Insert 22

$$T(22) = 22 \% 11 = 0$$

- $T(k) = k \bmod 11$
- $T'(k) = 7 - (k \bmod 7)$
- $T(k) = (T(k) + i * T'(k)) \bmod 11$

0	1	2	3	4	5	6	7	8	9	10
		35								



# Question - Resolution with Double Hashing

Insert 22

0	1	2	3	4	5	6	7	8	9	10
22		35								



# Question - Resolution with Double Hashing

Insert 44

$$T(44) = 44 \% 11 = 0$$

0	1	2	3	4	5	6	7	8	9	10
22		35								





# Question - Resolution with Double Hashing

Insert 44

0	1	2	3	4	5	6	7	8	9	10
22		35								



$$T'(k) = 7 - (44 \bmod 7) = 5$$

$$T(k) = (0 + 5 * 1) \bmod 11 = 5$$

# Question - Resolution with Double Hashing

Insert 44

0	1	2	3	4	5	6	7	8	9	10
22		35			44					



# Question - Resolution with Double Hashing

Insert 33

$$T(33) = 33 \% 11 = 0$$

0	1	2	3	4	5	6	7	8	9	10
22		35			44					



# Question - Resolution with Double Hashing

Insert 33

0	1	2	3	4	5	6	7	8	9	10
22		35			44					



$$T'(k) = 7 - (33 \bmod 7) = 2$$

$$T(k) = (0 + 2 * 1) \bmod 11 = 2$$

# Question - Resolution with Double Hashing

Insert 33

0	1	2	3	4	5	6	7	8	9	10
22		35			44					



$$T'(k) = 7 - (33 \bmod 7) = 2$$

$$T(k) = (0 + 2 * \mathbf{2}) \bmod 11 = 4$$

# Question - Resolution with Double Hashing

Insert 33 -> index 4

0	1	2	3	4	5	6	7	8	9	10
22		35		33	44					



# Question - Resolution with Double Hashing

Delete 33

0	1	2	3	4	5	6	7	8	9	10
22		35		33	44					



# Question - Resolution with Double Hashing

Delete 33

0	1	2	3	4	5	6	7	8	9	10
22		35		{ghost}	44					





# Question - Resolution with Double Hashing

Insert 59

$$T(44) = 59 \% 11 = 4$$

0	1	2	3	4	5	6	7	8	9	10
22		35		{ghost}	44					



# Question - Resolution with Double Hashing

Insert 59

$$T(59) = 59 \% 11 = 4$$

0	1	2	3	4	5	6	7	8	9	10
22		35		{ghost}	44					



$$T'(k) = 7 - (59 \bmod 7) = 4$$

$$T(k) = (4 + 1 * 4) \bmod 11 = 8$$

# Question - Resolution with Double Hashing

Insert 59

$$T(59) = 59 \% 11 = 4$$

0	1	2	3	4	5	6	7	8	9	10
22		35		59	44					



# Question - Invalid Hash Functions

Which of the following are **INVALID** hash functions for a hash table of size  $M$ ?

- I.  $H(str) = (str[0] * \text{rand}()) \% M$
- II.  $H(str) = (str[0] - 'a') * 503$
- III.  $H(str) = (\sum str[i] * i) \% M$



# Question - Invalid Hash Functions

Which of the following are **INVALID** hash functions for a hash table of size  $M$ ?

I.  $H(str) = (str[0] * \text{rand}()) \% M$

This is **invalid** because hashes must be consistent.

II.  $H(str) = (str[0] - 'a') * 503$

This is **invalid** because it might go out of bounds.

III.  $H(str) = (\sum str[i] * i) \% M$

This is **valid** because it fulfills all the properties of a hash.





# Trees and Tree Algorithms

# Tree Definitions

- Complete
  - All levels filled, except maybe the last, and all nodes in the last level are as far left as possible.
- Full/Proper
  - Every node has either 2 or 0 children



# Traversals

- Preorder
  - Visit the **parent**, **left** subtree, **right** subtree
- In- order
  - Visit **left** subtree, **parent**, **right** subtree
  - BST Squish -> Ordered list
- Post - order
  - Visit **left** subtree, **right** subtree, **parent**
- Level order
  - Breadth First Search
  - Use a queue
  - Think pairing heap destructor






- 1) The following code adds 100 numbers to a binary search tree and attempts to find a number.

```
BinarySearchTree<int> bst;  
for (int i = 0; i < 100; ++i) {  
    bst.insert(i);  
}
```

```
bst.find(280);  
bst.insert(281);
```

What are the worst-case time complexities of the find(280) and insert(281) operations?




- 1) The following code adds 100 numbers to a binary search tree and attempts to find a number.

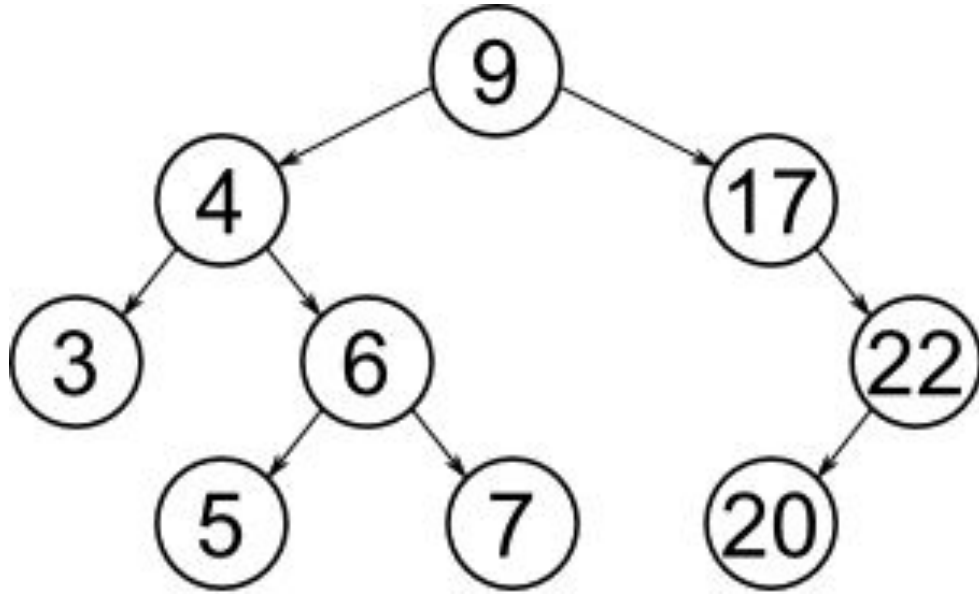
```
BinarySearchTree<int> bst;  
for (int i = 0; i < 100; ++i) {  
    bst.insert(i);  
}
```

```
bst.find(280);  
bst.insert(281);
```

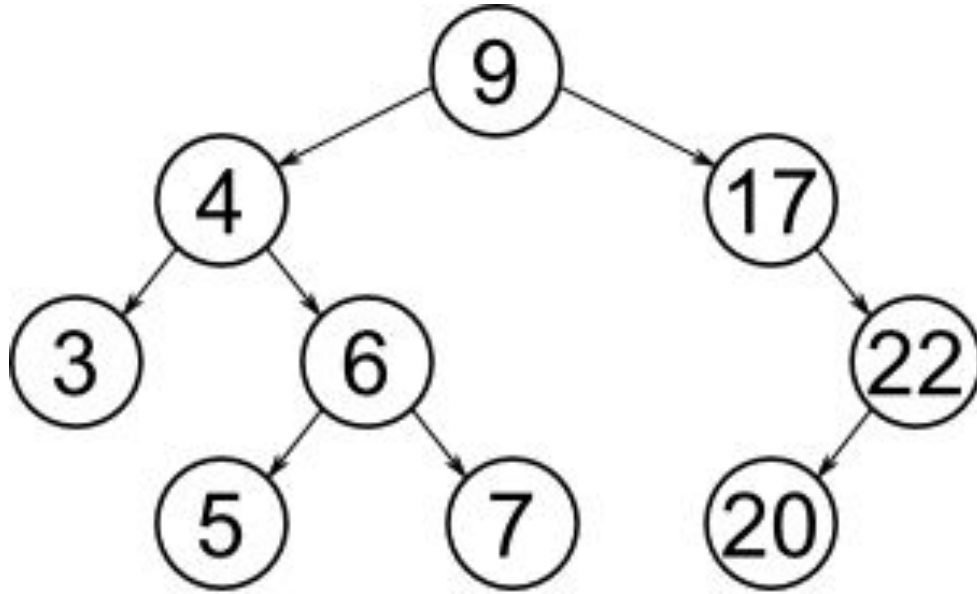
What are the worst-case time complexities of the find(280) and insert(281) operations?  $\Theta(n)$ ,  $\Theta(n)$



What is the pre-order traversal of this tree?

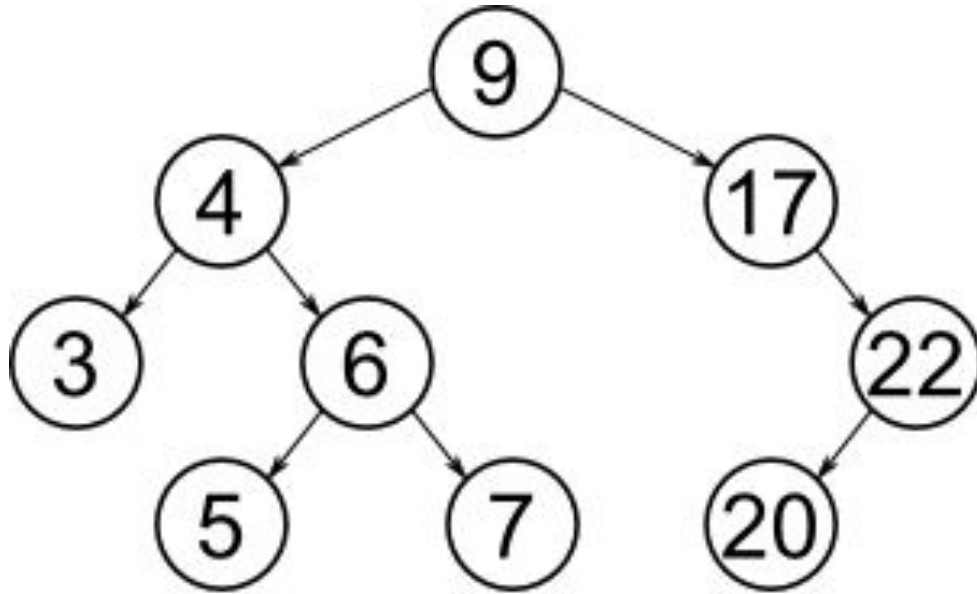


What is the pre-order traversal of this tree?

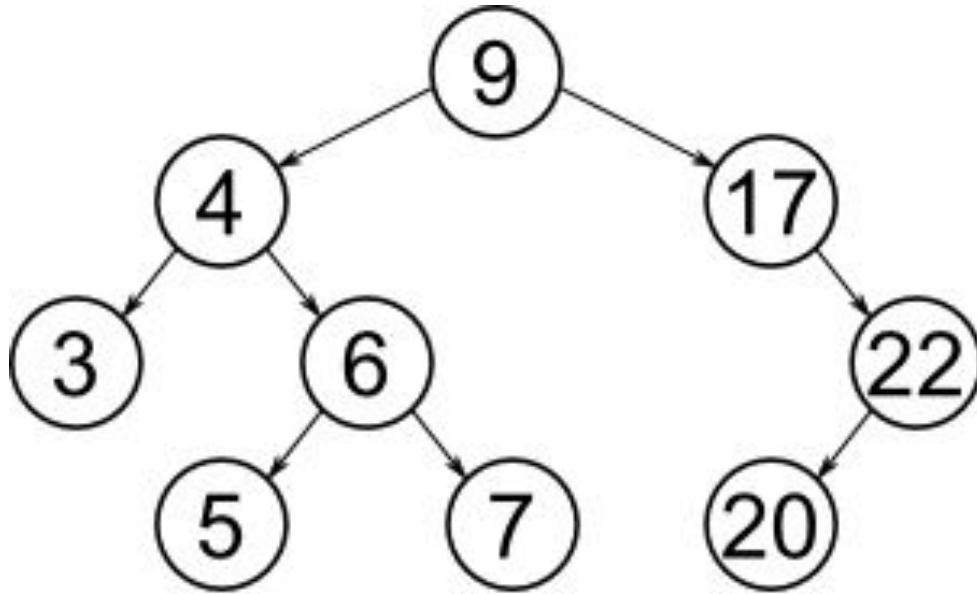


[9, 4, 3, 6, 5, 7, 17, 22, 20]

What is the in-order traversal of this tree?

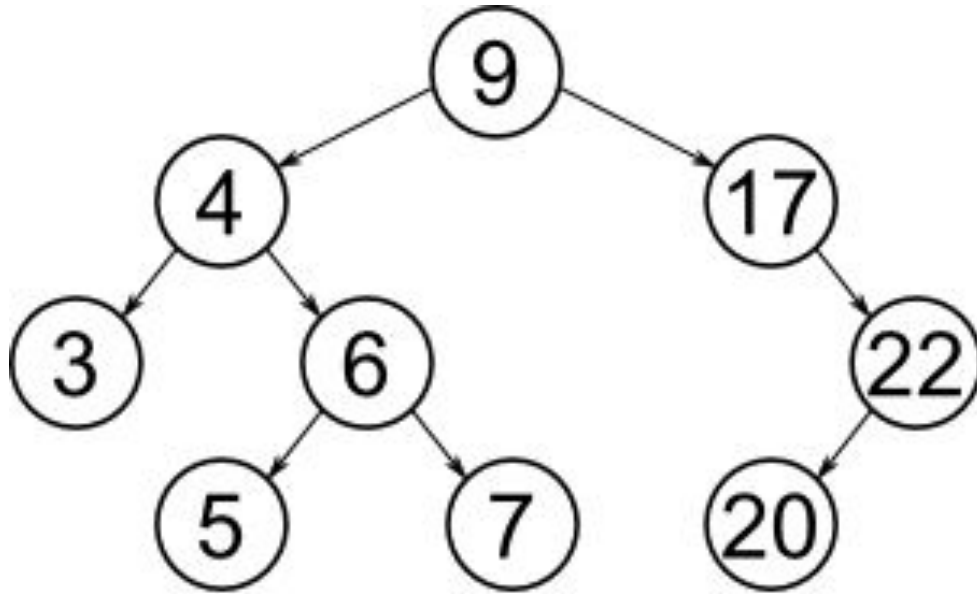


What is the in-order traversal of this tree?

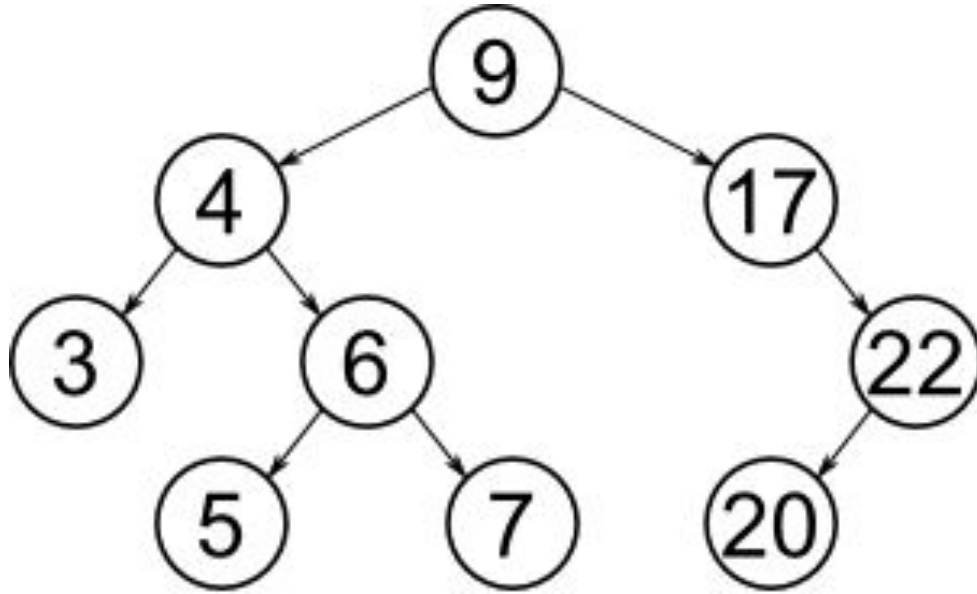


[3, 4, 5, 6, 7, 9, 17, 20, 22]

What is the post-order traversal of this tree?



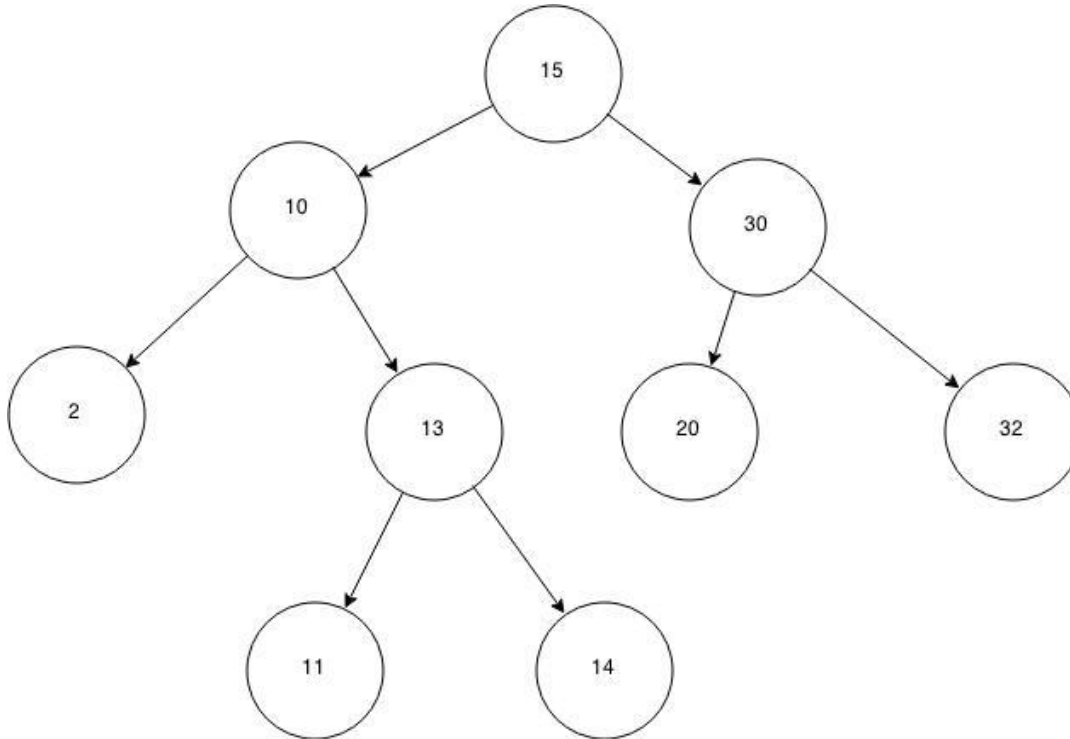
What is the post-order traversal of this tree?



[3, 5, 7, 6, 4, 20, 22, 17, 9]

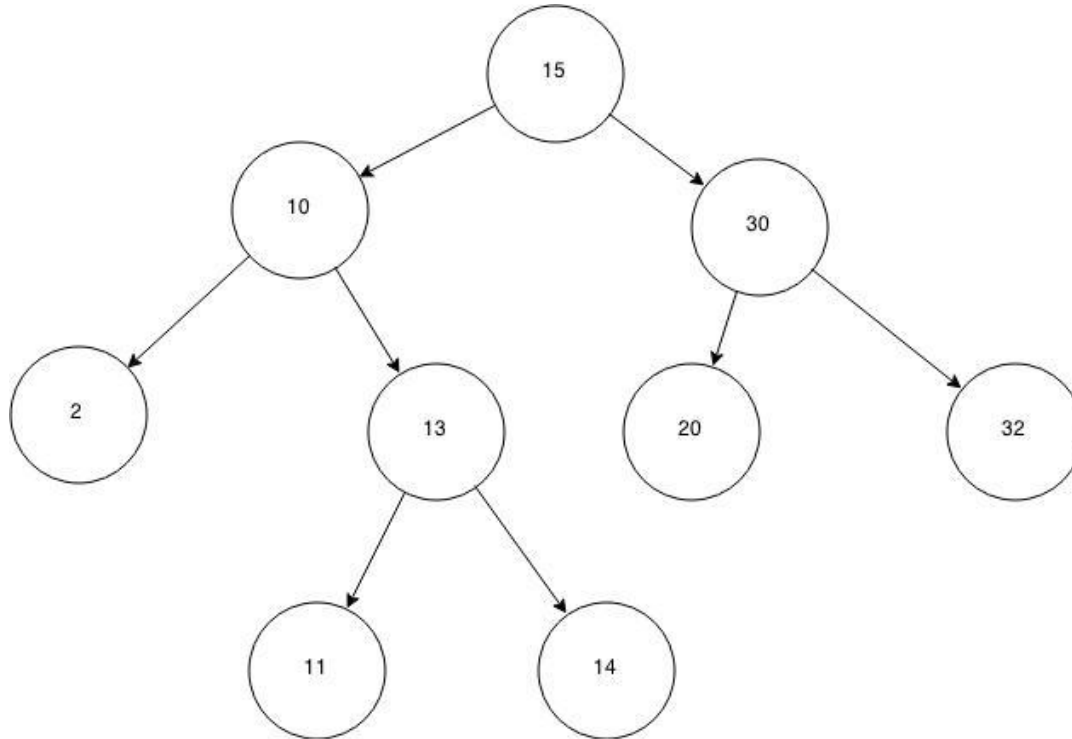


# Which of these correctly describe this tree?



- A. Complete, proper, balanced
- B. Not complete, proper, balanced
- C. Complete, not proper, not balanced
- D. Not complete, not proper, balanced
- E. Complete, not proper, balanced

# Which of these correctly describe this tree?



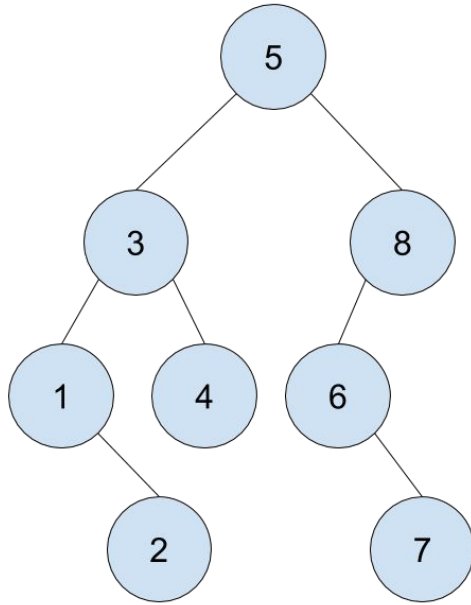
- A. Complete, proper, balanced
- B. Not complete, proper, balanced**
- C. Complete, not proper, not balanced
- D. Not complete, not proper, balanced
- E. Complete, not proper, balanced

Draw the tree resulting from the post-order and in-order traversals below.

Post-order: [2, 1, 4, 3, 7, 6, 8, 5]  
In-order: [1, 2, 3, 4, 5, 6, 7, 8]



Draw the tree resulting from the post-order and in-order traversals below.



Post-order: [2, 1, 4, 3, 7, 6, 8, 5]

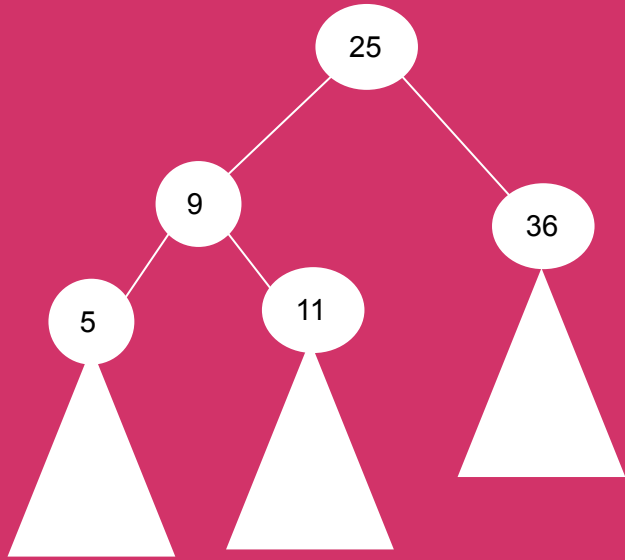
In-order: [1, 2, 3, 4, 5, 6, 7, 8]

# AVL Trees

# AVL Tree rules

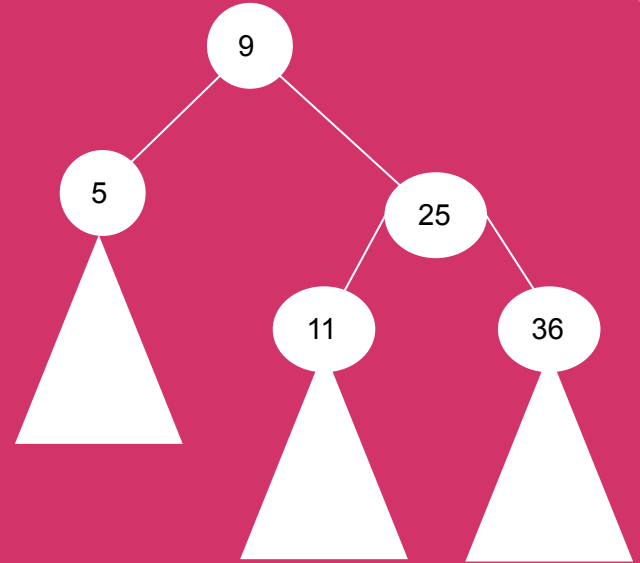
- Ensure that a tree stays balanced
  - "Balanced" means that  $|\text{Height}(\text{Left subtree}) - \text{Height}(\text{right subtree})| \leq 1$  for all nodes
- Four cases
  - Rotate Right
  - Rotate Left
  - Rotate Right then Rotate Left
  - Rotate Left then Rotate Right
- Insertions and deletions are same as BST, but
  - Require checking balance all parents of a new node up to the root for balance and rotating if necessary
  - See <https://visualgo.net/en/bst> to visualize how a BST works

# Rotation - Intuition



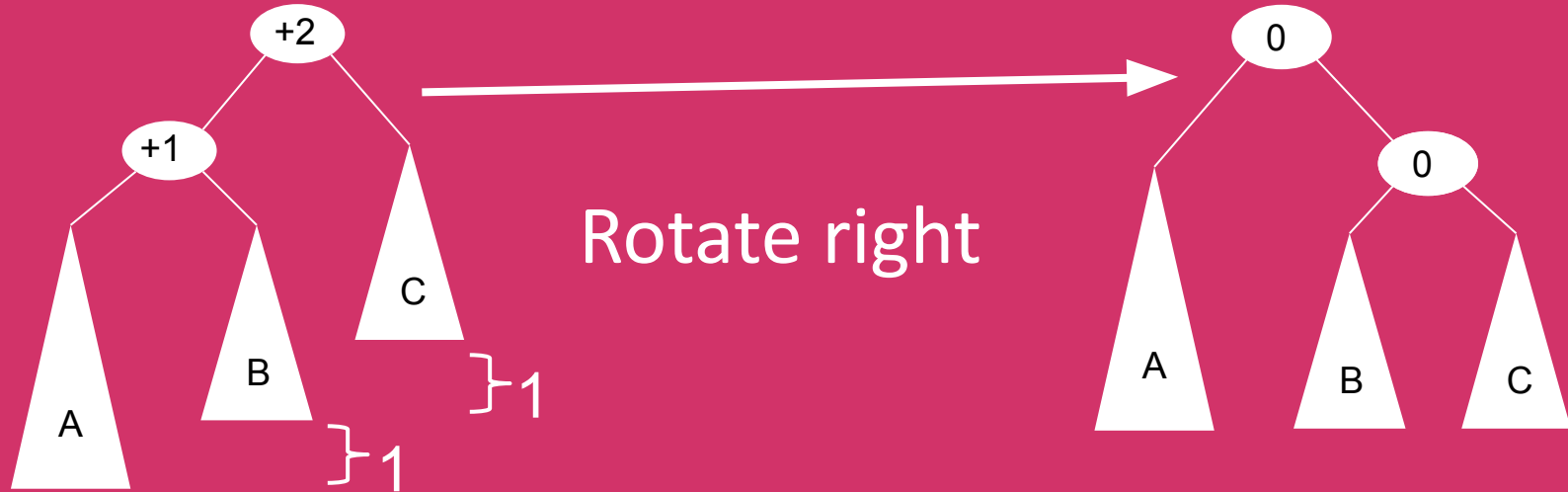
Rotate Right

Rotate Left



# AVL Rotation (Case 1)

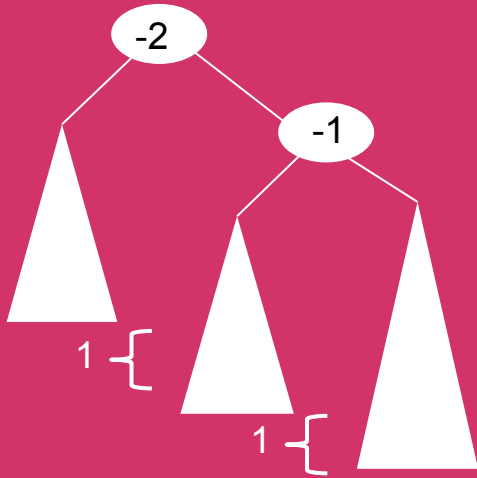
- **Left** sub-tree is too heavy, and the left subtree leans to the **left**



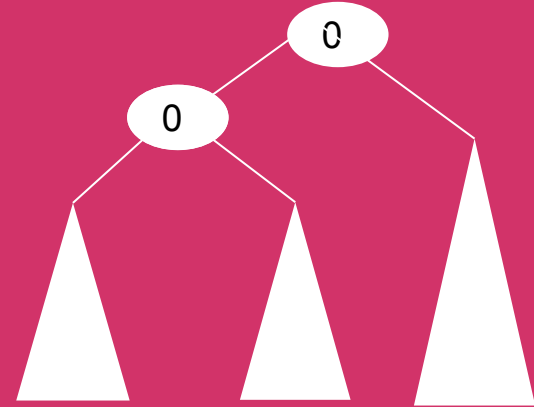


# AVL Rotation (Case 2)

- **Right** subtree is too heavy, and the right subtree leans to the right

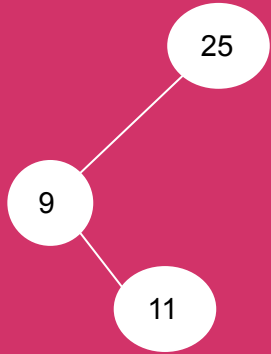


Rotate left



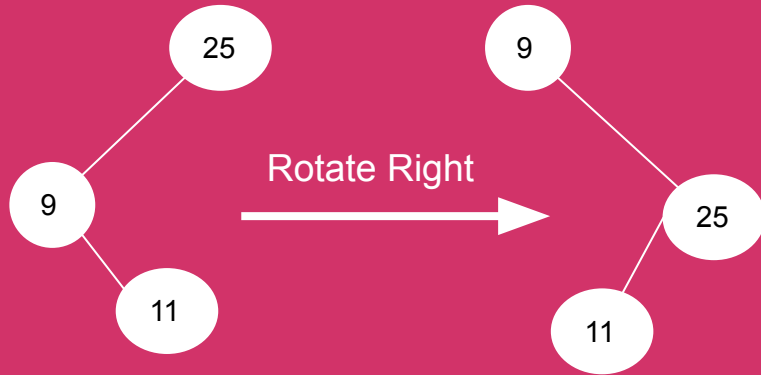
# Is that it?

Can this case be handled by the two cases we just saw?



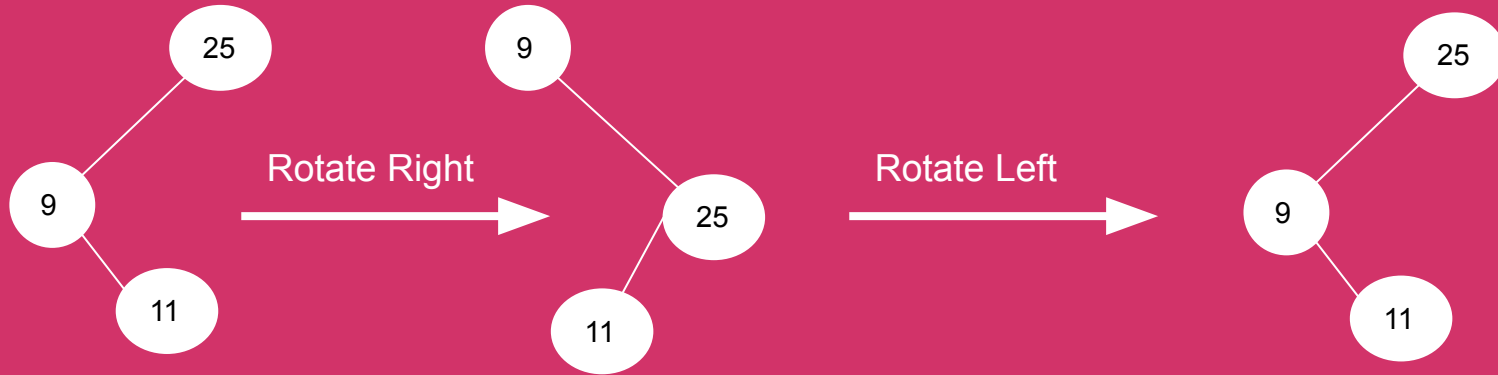
# Is that it?

Can this case be handled by the two cases we just saw?



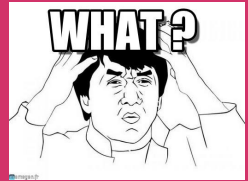
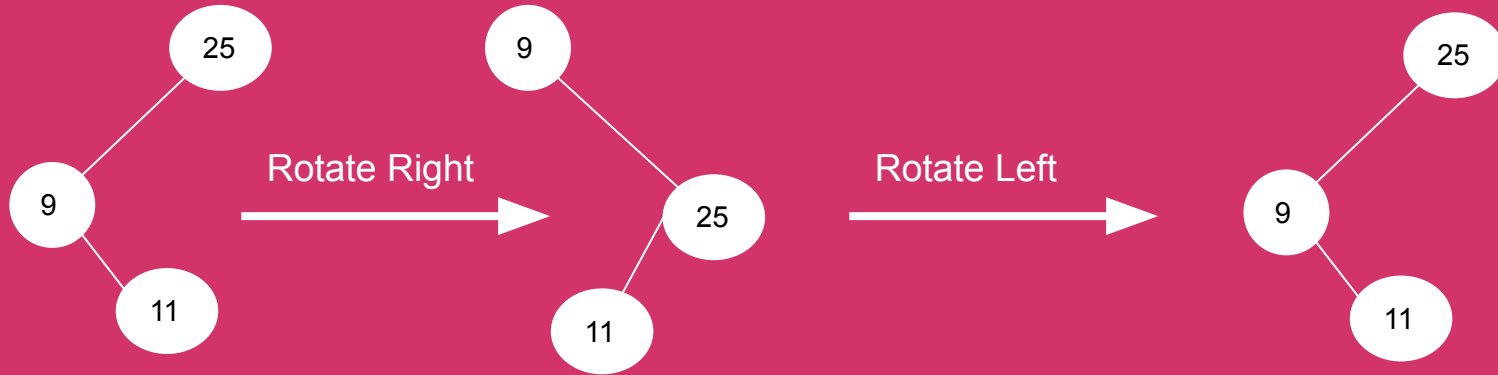
# Is that it?

Can this case be handled by the two cases we just saw?



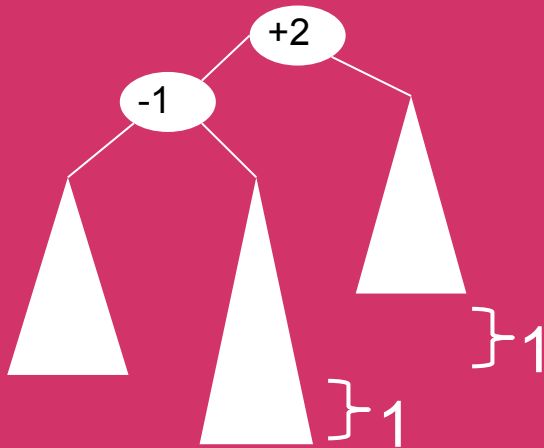
# Is that it?

Can this case be handled by the two cases we just saw?

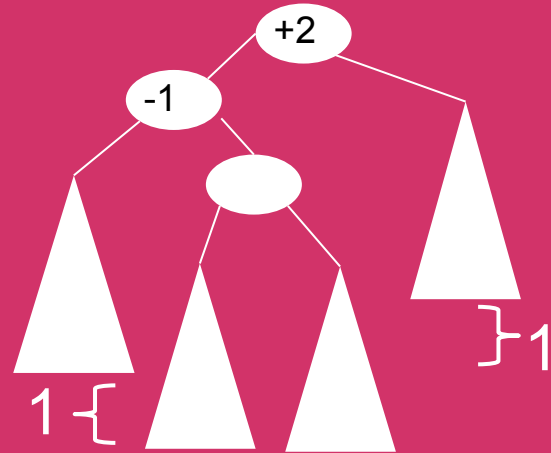


# AVL Rotation (Case 3)

- Left sub-tree is too heavy, and the left subtree leans to the right
  - Rotate the left subtree LEFT and then you have Case 1.

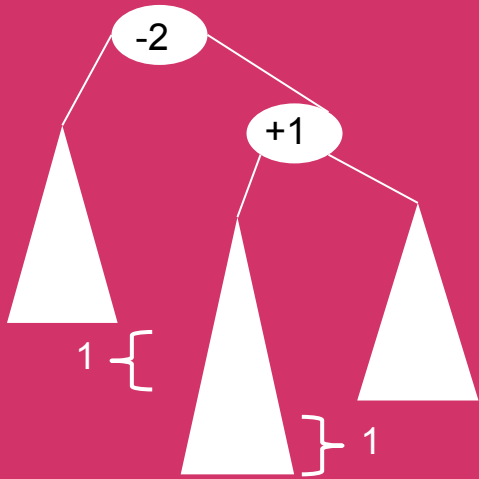


Is really

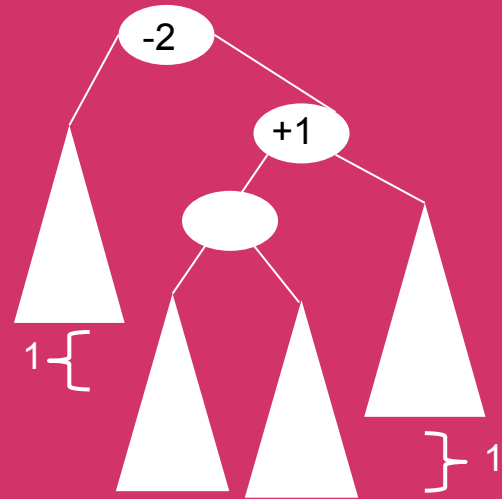


# AVL Rotation (Case 4)

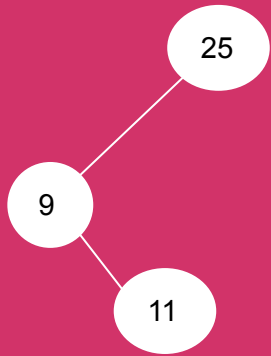
- Right sub-tree is too heavy, and the right subtree leans to the
  - left Rotate the right subtree RIGHT and then you have Case 2.



Is really

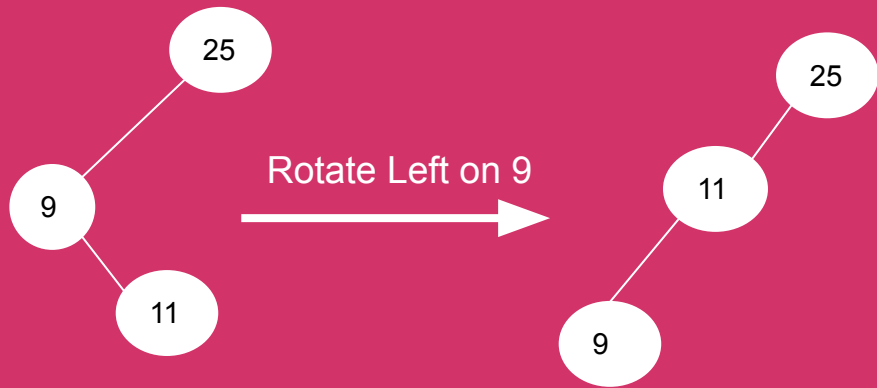


# Let's try again

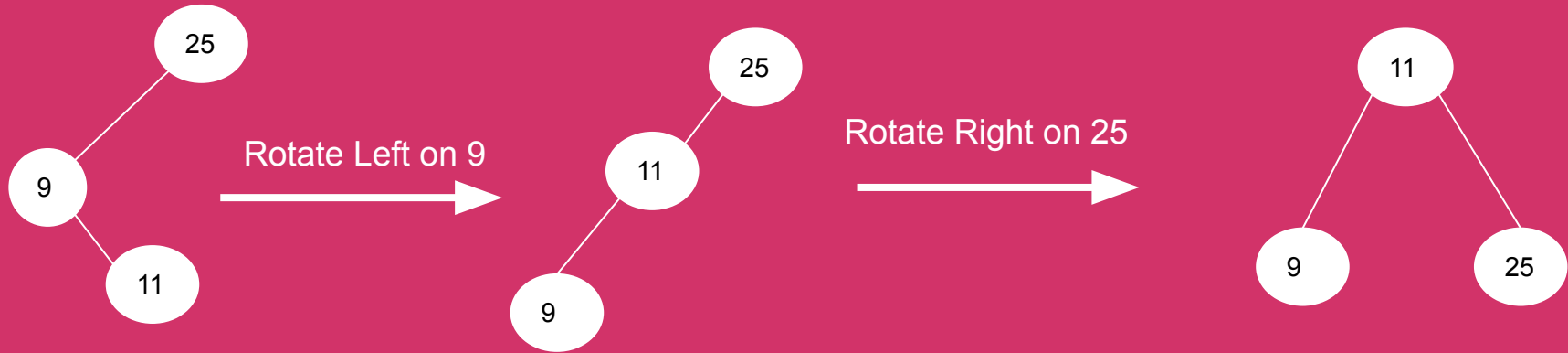




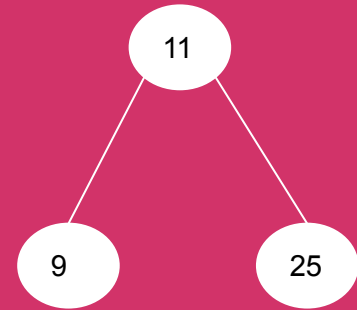
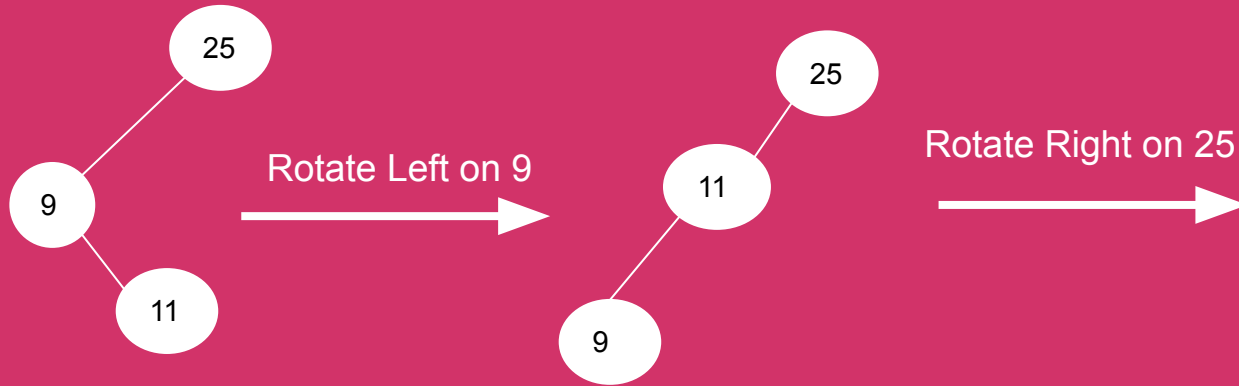
# Let's try again



# Let's try again



# Let's try again



# Balancing

```
Algorithm checkAndBal(Node *n)
    updateHeight(n)
    if bal(n) > +1
        if bal(n->left) < 0
            rotateL(n->left)
        rotateR(n)
    else if bal(n) < -1
        if bal(n->right) > 0
            rotateR(n->right)
        rotateL(n)
    else
        checkAndBal(n->parent)
```

What does the '+' in +1 convey?

=> Left subtree is bigger!

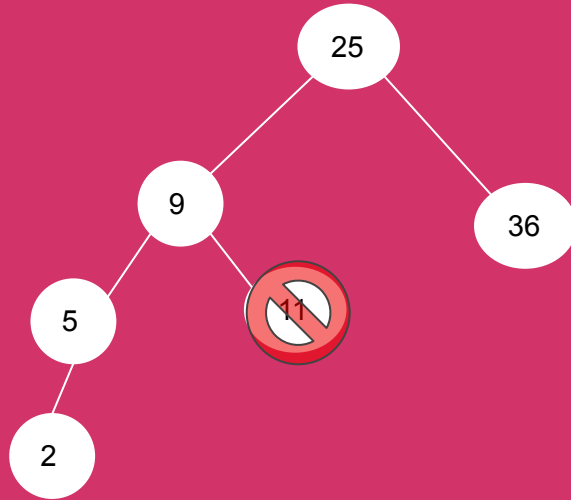
What if  $\text{bal}(n) > +1$ ?

=> Unbalanced! (since  $> 1$ )

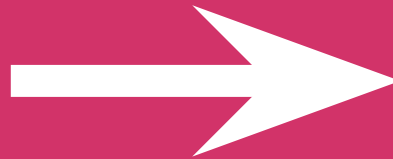
=> Left subtree is bigger

# Deletion: 0 Children

If the node you're deleting has no children, just remove it



Delete 11



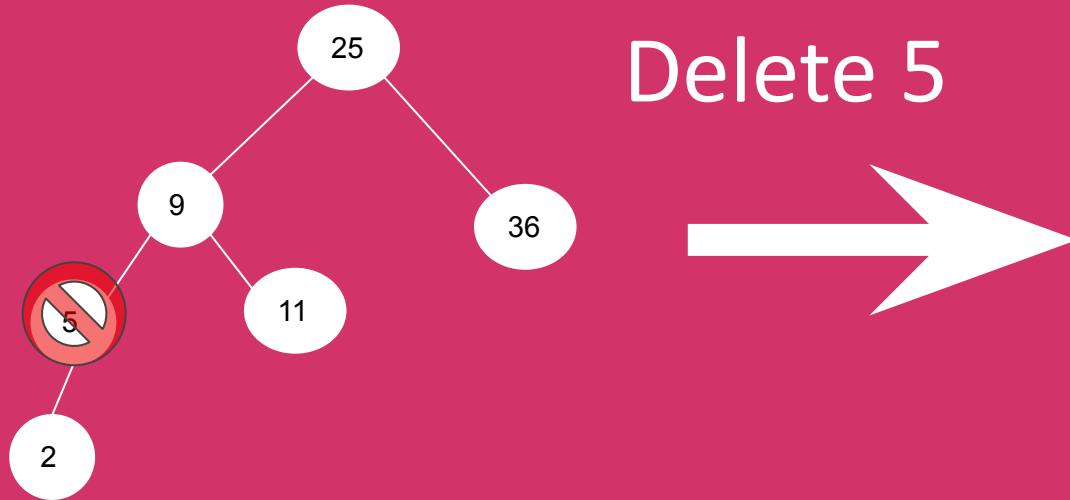
# Deletion: 0 Children

If the node you're deleting has no children, just remove it



# Deletion: 1 Child

If it only has one child, move that child up:



# Deletion: 1 Child

If it only has one child, move that child up:





# Deletion: 2 Children

Otherwise, it has two children. Swap it with its in-order successor or in-order predecessor, then delete it in its new spot. Rebalance from the place it was deleted, if needed.

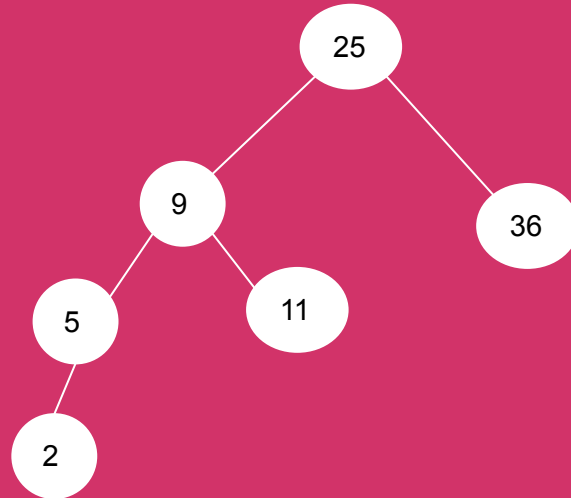
***Both choices always work*** (but we may tell you which to use)

*The successor/predecessor will always have 0 or 1 children (if the node itself has 2 children). It can be thought of as being the rightmost node in the left subtree or the leftmost node in the right subtree respectively.*

# Deletion: 2 Children

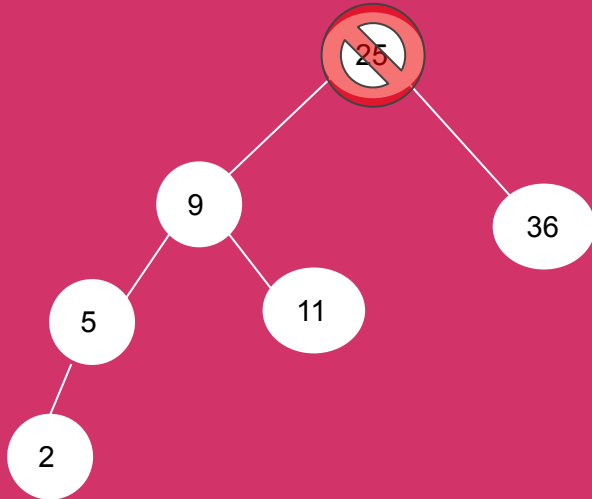
Replace with in-order successor or in-order predecessor, then delete in its new spot.

Delete 25



# Deletion

Let's replace 25 with inorder predecessor (11)



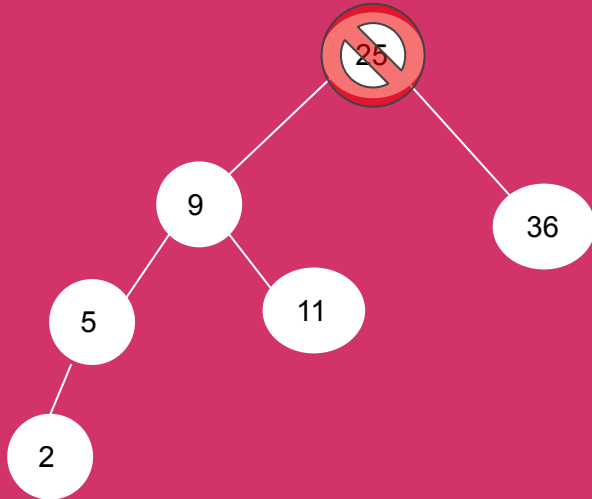
Delete 25  
(swap with  
predecessor)



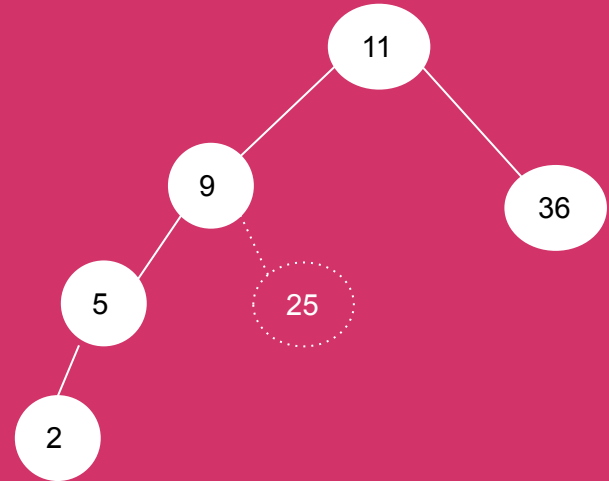
?

# Deletion

Let's replace 25 with inorder predecessor (11)

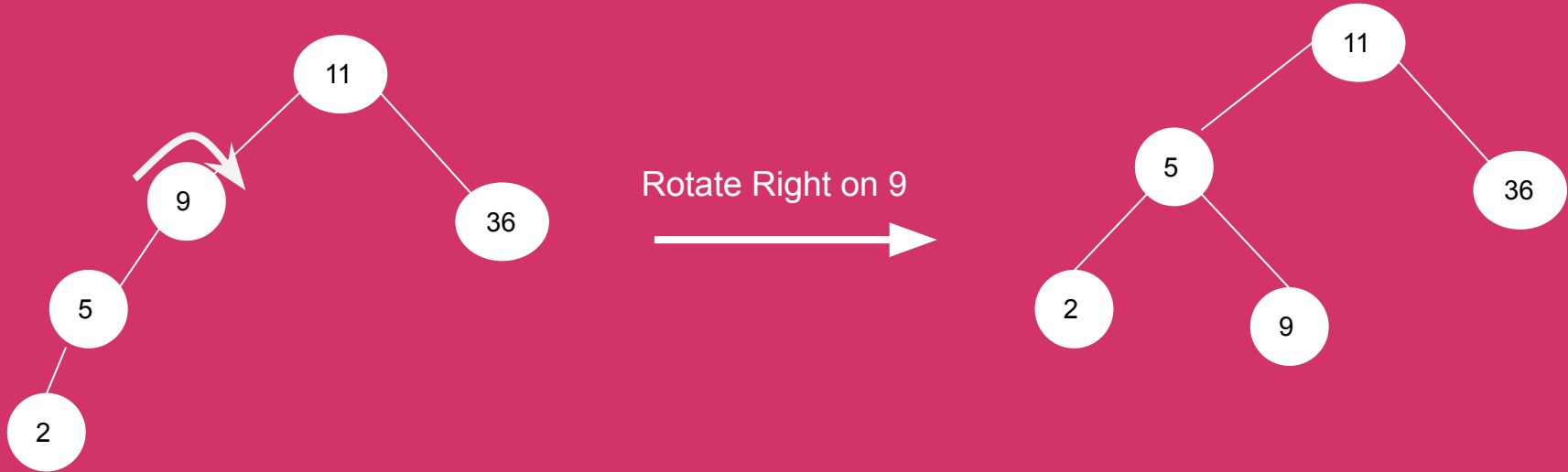


Delete 25  
(swap with  
predecessor)

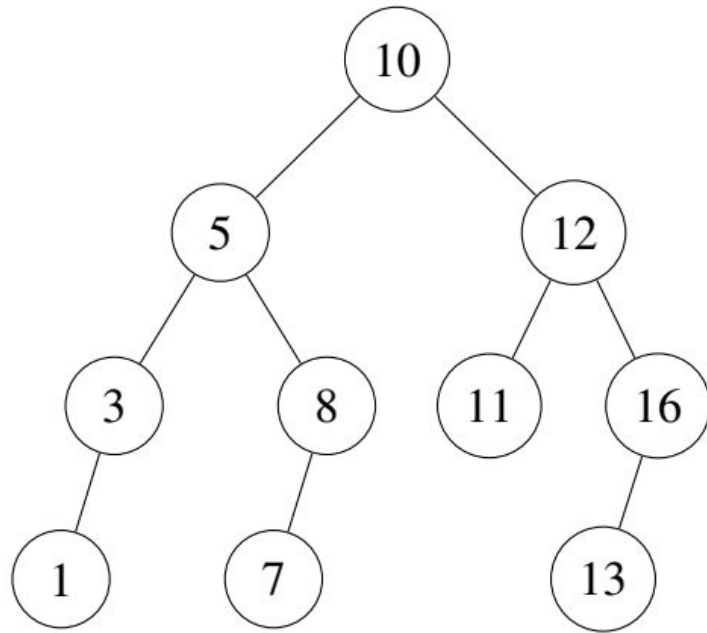


# Deletion

Replace with inorder successor or predecessor (In the exam, you will be told which one). Just remember **both work!**

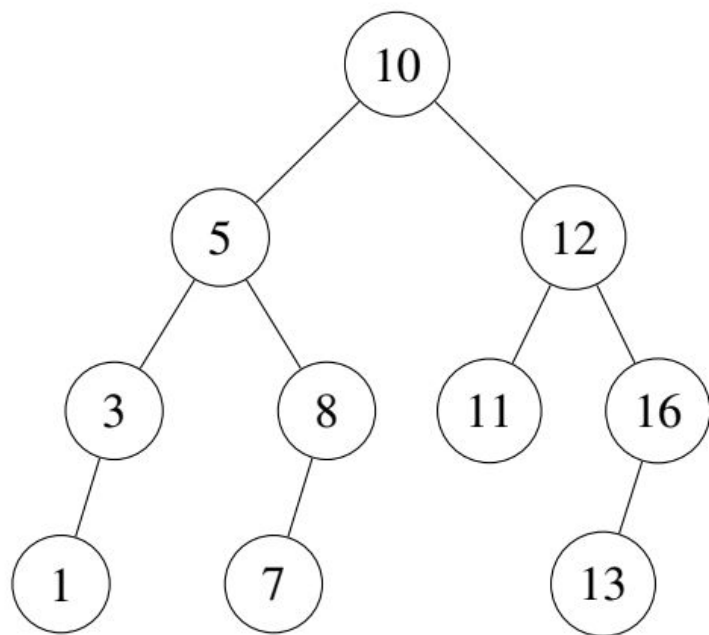


# Practice Question



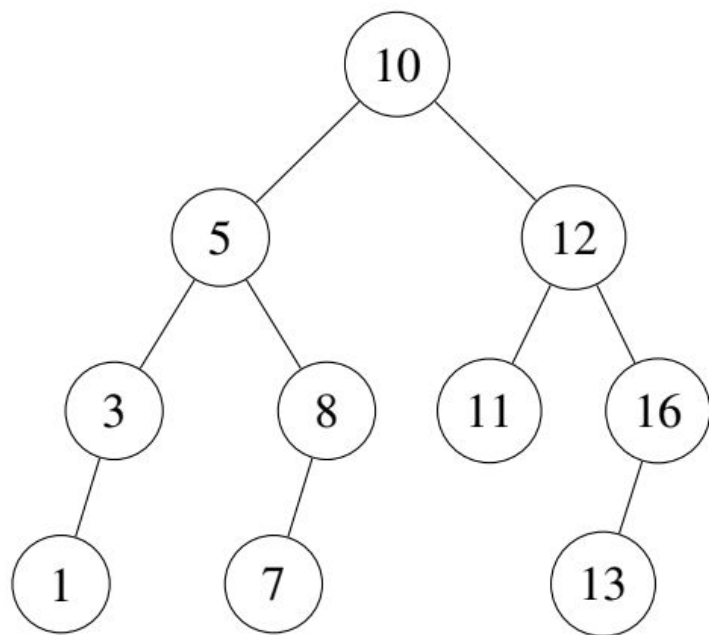
- After inserting 14 and performing any necessary rotations, what would be the produced pre-order traversal?

# Practice Question



- After inserting 14 and performing any necessary rotations, what would be the produced pre-order traversal?
- **A: 10, 5, 3, 1, 8, 7, 12, 11, 14, 13, 16**
- What if the node 10 was deleted, and we use the successor? (14 isn't inserted)

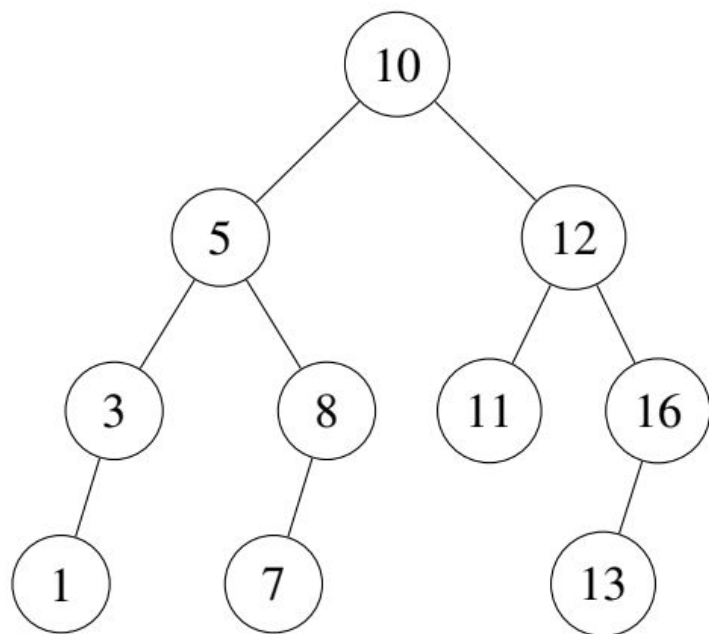
# Practice Question



- After inserting 14 and performing any necessary rotations, what would be the produced pre-order traversal?
- **A: 10, 5, 3, 1, 8, 7, 12, 11, 14, 13, 16**
- What if the node 10 was deleted, and we use the successor? (14 isn't inserted)
- **A: 11, 5, 3, 1, 8, 7, 13, 12, 16**
- What if nodes 12 and 13 are then deleted after node 10 is deleted? (14 isn't inserted)



# Practice Question



- After inserting 14 and performing any necessary rotations, what would be the produced pre-order traversal?
- **A: 10, 5, 3, 1, 8, 7, 12, 11, 14, 13, 16**
- What if the node 10 was deleted, and we use the successor? (14 isn't inserted)
- **A: 11, 5, 3, 1, 8, 7, 13, 12, 16**
- What if nodes 12 and 13 are then deleted after node 10 is deleted? (14 isn't inserted)
- **A: 5, 3, 1, 11, 8, 7, 16**

# Graph ADT

# Graph Representations

Adjacency List	Adjacency Matrix (2D Vector)
Better for sparse graphs ( $O(E)$ )	Better for dense graphs (always $O(V^2)$ )
$O(1 + E/V)$ to find if an edge exists	$O(1)$ look up to find an edge
Either 2D Vector or vector of linked lists	2D vector



# Graph

## 1. Trees and Graphs

Trees can technically be considered graphs that have the following properties:

- I.** Dense
- II.** Cyclic
- III.** Connected

- A)** I and III
- B)** II only
- C)** III only
- D)** I, II, and III



# Graph

## 1. Trees and Graphs

Trees can technically be considered graphs that have the following properties:

- I.** Dense
- II.** Cyclic
- III.** Connected

- A)** I and III
- B)** II only
- C)** III only
- D)** I, II, and III

C.



# Graph

Use the adjacency matrix given below. Starting from V1, determine a possible DFS sequence.

	V1	V2	V3	V4	V5	V6
V1	0	1	1	0	1	0
V2	1	0	0	1	0	0
V3	1	0	0	0	1	1
V4	0	1	0	0	1	0
V5	1	0	1	1	0	1
V6	0	0	1	0	1	0

**Note:** the order that adjacent vertices is visited is unspecified, you must determine which order is possible.

- A) V1, V2, V3, V4, V5, V6
- B) V1, V2, V4, V5, V6, V3
- C) V1, V3, V5, V2, V4, V6
- D) V1, V3, V4, V6, V5, V2

# Graph

Use the adjacency matrix given below. Starting from V1, determine a possible DFS sequence.

	V1	V2	V3	V4	V5	V6
V1	0	1	1	0	1	0
V2	1	0	0	1	0	0
V3	1	0	0	0	1	1
V4	0	1	0	0	1	0
V5	1	0	1	1	0	1
V6	0	0	1	0	1	0

**Note:** the order that adjacent vertices is visited is unspecified, you must determine which order is possible.

- A) V1, V2, V3, V4, V5, V6
- B) V1, V2, V4, V5, V6, V3
- C) V1, V3, V5, V2, V4, V6
- D) V1, V3, V4, V6, V5, V2

B.

# Greedy Algorithms: MSTs



# Minimum Spanning Tree

- Given an edge-weighted, undirected graph, find a spanning tree of minimum weight
- Spanning Tree
  - A subgraph containing all the vertices in the original (spanning)
  - Connected and acyclic (tree)



# Prim's

- Start with 2 sets of vertices: 'innies' & 'outies'
  - 'innies' are visited set
  - 'outies' are unvisited set
- Select first innie arbitrarily (usually lowest index)
- Iteratively choose the out-vertex which is closest to *any* in-vertex
- Best for dense graphs
- Complexity
  - $O(V^2)$  with linear search
  - $O(E \log V)$  with binary heaps



# MST

Starting from vertex 0, in which order would Prim's algorithm add vertices to the MST in the graph described by the below adjacency matrix?

	0	1	2	3	4
0	-	4	8	18	1
1	4	-	1	11	22
2	8	1	-	7	5
3	18	11	7	-	13
4	1	22	5	13	-



# MST

Starting from vertex 0, in which order would Prim's algorithm add vertices to the MST in the graph described by the below adjacency matrix?

	0	1	2	3	4
0	-	4	8	18	1
1	4	-	1	11	22
2	8	1	-	7	5
3	18	11	7	-	13
4	1	22	5	13	-

(0, 4)



# MST

Starting from vertex 0, in which order would Prim's algorithm add vertices to the MST in the graph described by the below adjacency matrix?

	0	1	2	3	4
0	-	4	8	18	1
1	4	-	1	11	22
2	8	1	-	7	5
3	18	11	7	-	13
4	1	22	5	13	-

(0, 4) (0, 1)



# MST

Starting from vertex 0, in which order would Prim's algorithm add vertices to the MST in the graph described by the below adjacency matrix?

	0	1	2	3	4
0	-	4	8	18	1
1	4	-	1	11	22
2	8	1	-	7	5
3	18	11	7	-	13
4	1	22	5	13	-

(0, 4) (0, 1) (1, 2)



# MST

Starting from vertex 0, in which order would Prim's algorithm add vertices to the MST in the graph described by the below adjacency matrix?

	0	1	2	3	4
0	-	4	8	18	1
1	4	-	1	11	22
2	8	1	-	7	5
3	18	11	7	-	13
4	1	22	5	13	-

(0, 4) (0, 1) (1, 2) (2, 3)



# Kruskal's

- Iteratively add lowest weight edges until all vertices connected
- Check for cycle each time
- Best for sparse graphs
- Complexity
  - Pre-sort all Edges  $O(E \log E)$
  - Since  $E < V^2$ 
    - $O(E \log E) = O(E \log V)$





# MST (True or False)

- If an MST of a graph does not contain some edge of the shortest length, then all edges of the graph must be the same length.
- If this graph is connected and has  $V$  vertices, an MST of this graph must contain exactly  $V - 1$  edges.
- When implementing Prim's algorithm for a dense graph, using a heap is better than using linear search.



# MST (True or False)

- If an MST of a graph does not contain some edge of the shortest length, then all edges of the graph must be the same length.  
F. There are some edges with the same length (the shortest).
- If this graph is connected and has  $V$  vertices, an MST of this graph must contain exactly  $V - 1$  edges.
- When implementing Prim's algorithm for a dense graph, using a heap is better than using linear search.



# MST (True or False)

- If an MST of a graph does not contain some edge of the shortest length, then all edges of the graph must be the same length.  
**F. There are some edges with the same length (the shortest).**
- If this graph is connected and has  $V$  vertices, an MST of this graph must contain exactly  $V - 1$  edges.  
**T. Have to connect every vertex exactly once.**
- When implementing Prim's algorithm for a dense graph, using a heap is better than using linear search.



# MST (True or False)

- If an MST of a graph does not contain some edge of the shortest length, then all edges of the graph must be the same length.

F. There are some edges with the same length (the shortest).

- If this graph is connected and has  $V$  vertices, an MST of this graph must contain exactly  $V - 1$  edges.

T. Have to connect every vertex exactly once.

- When implementing Prim's algorithm for a dense graph, using a heap is better than using linear search.

F. Heap:  $O(E \log V)$ , Linear Search:  $O(V^2)$

Dense:  $|E| \approx |V| * (|V| - 1) / 2$



# MST, Prim's & Kruskal's

Which of the following statements about Prim's, Kruskal's, and MSTs is **FALSE**?

- A) Prim's and Kruskal's algorithms will always return the same MST.
- B) Prim's algorithm works for creating an MST even when the weights of edges are negative.
- C) An MST does not need to contain the shortest path between any two vertices.
- D) Prim's algorithm can run slower than  $O(V^2)$ .



# MST, Prim's & Kruskal's

Which of the following statements about Prim's, Kruskal's, and MSTs is **FALSE**?

- A)** Prim's and Kruskal's algorithms will always return the same MST.
- B) Prim's algorithm works for creating an MST even when the weights of edges are negative.
- C) An MST does not need to contain the shortest path between any two vertices.
- D) Prim's algorithm can run slower than  $O(V^2)$ .

A: False, multiple potential MSTs

B: True, MST algorithms can generally deal with negative edges.

C: True, MST is for minimal connection, not shortest paths.

D: True, with binary heap  $\rightarrow O(E \log V)$ . When dense,  $O(E \log V) = O(V^2 \log V)$

# MST

A network of satellites needs to be able to establish communication between any two satellites in the network. However, they cannot all establish a connection directly to one another, so they must route signals through one another. The space agency wants to do this routing as quickly as possible, so at any given time they want to maintain a minimum spanning tree of the satellites. The communication times, in nanoseconds, between the connected satellites is shown in the following matrix:

	A	B	C	D	E	F	G
A	–	746	4571	1648	–	–	–
B	746	–	–	–	–	–	–
C	4571	–	–	–	–	–	–
D	1648	–	–	–	–	617	470
E	–	–	–	–	–	–	905
F	–	–	–	617	–	–	670
G	–	–	–	470	905	670	–

Given this information, could this be a valid solution for the space agency's current MST?

- A) No, because the graph is not fully connected
- B) No, because the spanning tree is not minimal
- C) No, because the graph does not contain a Hamiltonian cycle
- D) No, because the graph contains a cycle
- E) Yes, this is a potentially valid solution

# MST

A network of satellites needs to be able to establish communication between any two satellites in the network. However, they cannot all establish a connection directly to one another, so they must route signals through one another. The space agency wants to do this routing as quickly as possible, so at any given time they want to maintain a minimum spanning tree of the satellites. The communication times, in nanoseconds, between the connected satellites is shown in the following matrix:

	A	B	C	D	E	F	G
A	–	746	4571	1648	–	–	–
B	746	–	–	–	–	–	–
C	4571	–	–	–	–	–	–
D	1648	–	–	–	–	617	470
E	–	–	–	–	–	–	905
F	–	–	–	617	–	–	670
G	–	–	–	470	905	670	–

Given this information, could this be a valid solution for the space agency's current MST?

- A) No, because the graph is not fully connected
- B) No, because the spanning tree is not minimal
- C) No, because the graph does not contain a Hamiltonian cycle
- D) No, because the graph contains a cycle**
- E) Yes, this is a potentially valid solution

D.

DFG forms a cycle.



# Greedy Algorithms: Dijkstra's

# Dijkstra's Algorithm aka Single Source Shortest Path (SSSP)

- Greedy algorithm that finds the shortest path between any one vertex and all other vertices in the graph.
  - Greedily select the next vertex of the shortest distance from the Source
- Complexity
  - $V^2$  (linear probe to find the closest vertex)
  - $|E| \log |V|$  with a priority queue

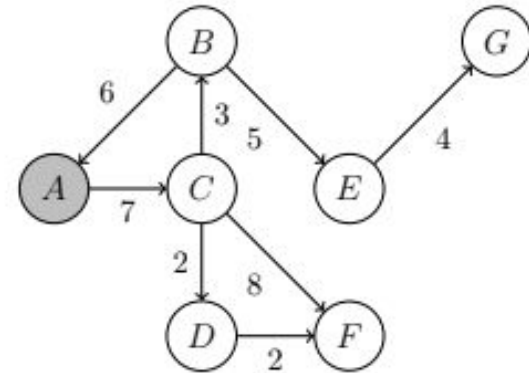


# Dijkstra's Algorithm

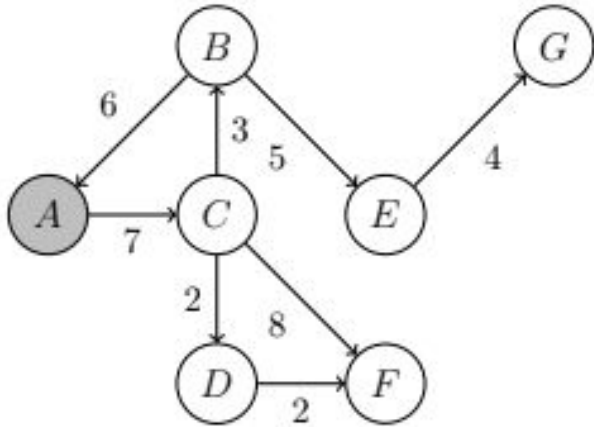
Starting at vertex A and trying to find the shortest path to G, which of the following progressions shows how the vertices are added to our set?

- A) A, C, D, B, F, E, G
- B) A, C, B, D, F, E, G
- C) A, C, D, B, E, F, G
- D) A, B, E, G
- E) A, C, B, E, G

A	0
B	-
C	7
D	-
E	-
F	-
G	-

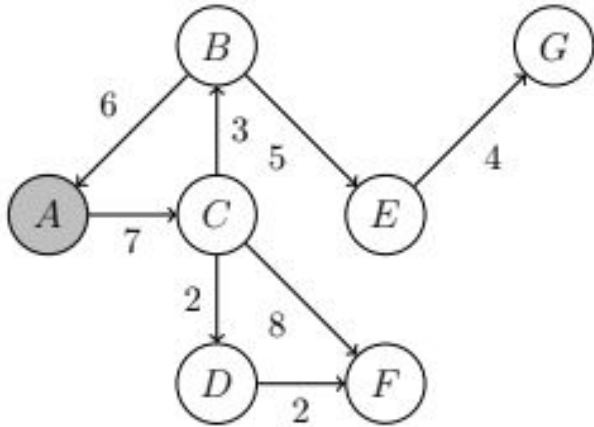


# Dijkstra's Algorithm



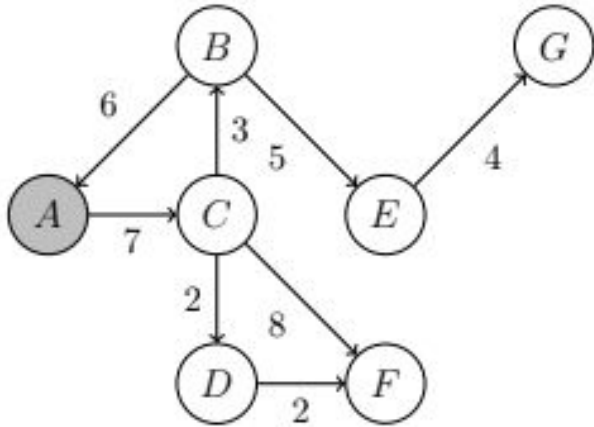
A	0
B	-
C	7
D	-
E	-
F	-
G	-

# Dijkstra's Algorithm



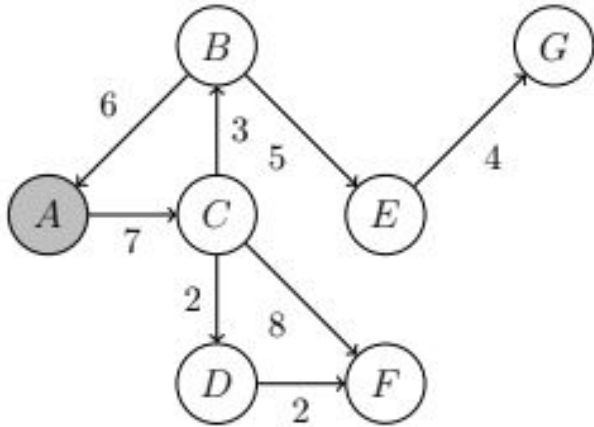
A	0
B	10
C	7
D	9
E	-
F	15
G	-

# Dijkstra's Algorithm



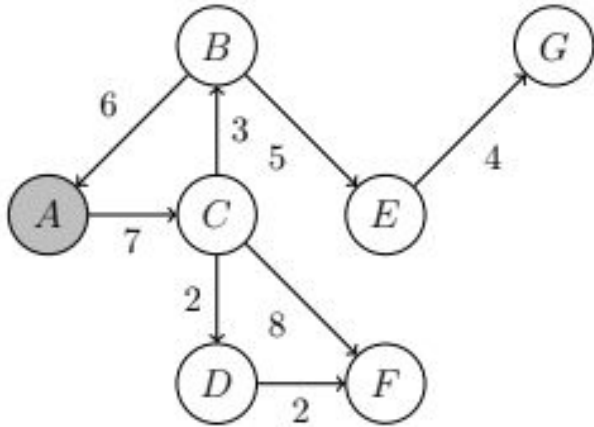
A	0
B	10
C	7
D	9
E	-
F	11
G	-

# Dijkstra's Algorithm



A	0
B	10
C	7
D	9
E	15
F	11
G	-

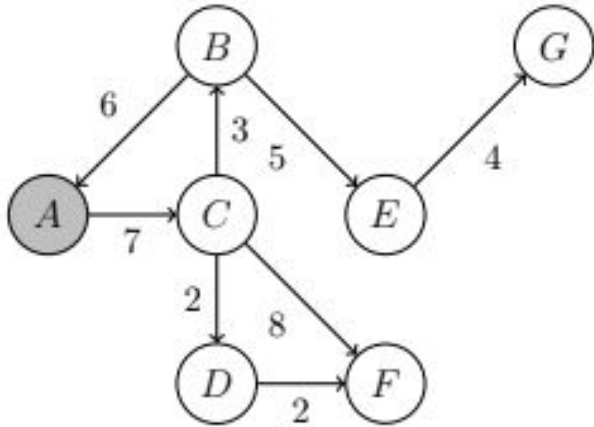
# Dijkstra's Algorithm



A	0
B	10
C	7
D	9
E	15
F	11
G	-

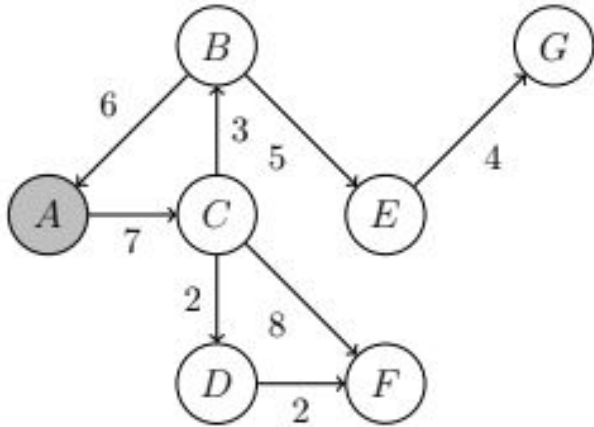


# Dijkstra's Algorithm



A	0
B	10
C	7
D	9
E	15
F	11
G	19

# Dijkstra's Algorithm

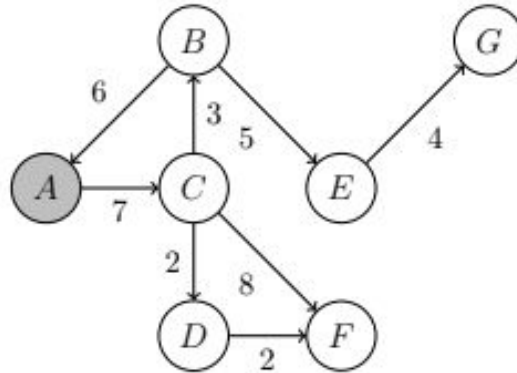


A	0
B	10
C	7
D	9
E	15
F	11
G	19

# Dijkstra's Algorithm

Starting at vertex A and trying to find the shortest path to G, which of the following progressions shows how the vertices are added to our set?

- A)** A, C, D, B, F, E, G
- B) A, C, B, D, F, E, G
- C) A, C, D, B, E, F, G
- D) A, B, E, G
- E) A, C, B, E, G



A	0
B	10
C	7
D	9
E	15
F	11
G	19

# Dijkstra's Algorithm

Which of the following is **FALSE** about Dijkstra's algorithm?

- A) Dijkstra's algorithm uses a Greedy approach to solving its problem.
- B) Dijkstra's algorithm cannot be used on an input graph with negative edge weights.
- C) Dijkstra's algorithm only finds the shortest path between two input vertices.
- D) Dijkstra's algorithm is capable of achieving a time complexity  $O(|E| \log |V|)$ .



# Dijkstra's Algorithm

Which of the following is **FALSE** about Dijkstra's algorithm?

- A) Dijkstra's algorithm uses a Greedy approach to solving its problem.
- B) Dijkstra's algorithm cannot be used on an input graph with negative edge weights.
- ☒ C) Dijkstra's algorithm only finds the shortest path between two input vertices.
- D) Dijkstra's algorithm is capable of achieving a time complexity  $O(|E| \log |V|)$ .

A: Dijkstra is greedy.

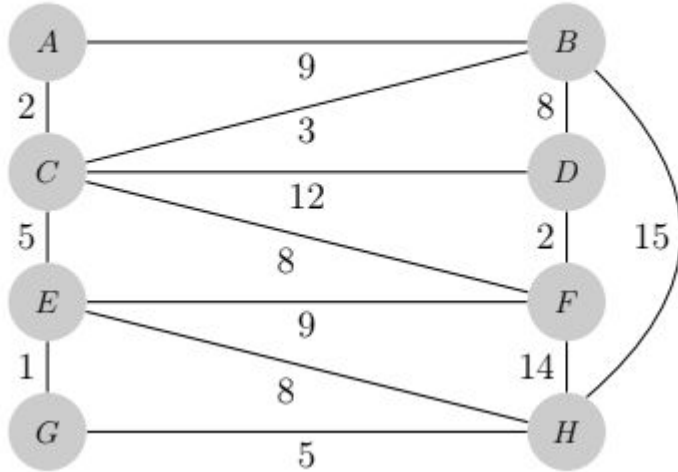
B: Dijkstra cannot handle negative edge weights.

C: Find the shortest path between the source and every other vertex.

D: With priority queue.

# Dijkstra's Algorithm

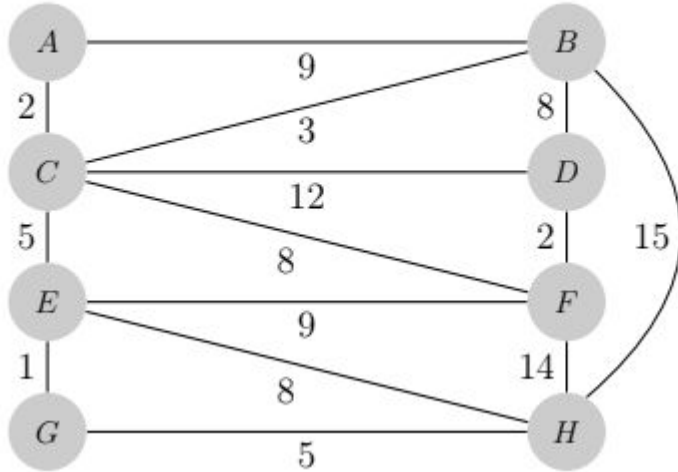
Apply Dijkstra's algorithm to the following graph, starting from vertex A. What is the order in which vertices are marked as known?



A	0
B	-
C	-
D	-
E	-
F	-
G	-
H	-

# Dijkstra's Algorithm

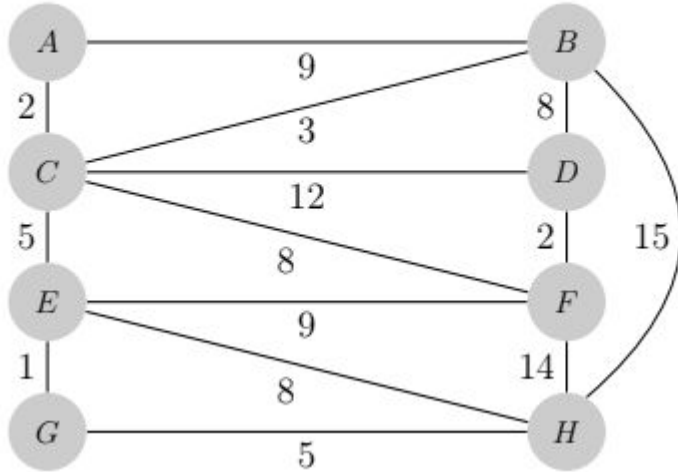
Apply Dijkstra's algorithm to the following graph, starting from vertex A. What is the order in which vertices are marked as known?



A	0
B	9
C	2
D	-
E	-
F	-
G	-
H	-

# Dijkstra's Algorithm

Apply Dijkstra's algorithm to the following graph, starting from vertex A. What is the order in which vertices are marked as known?

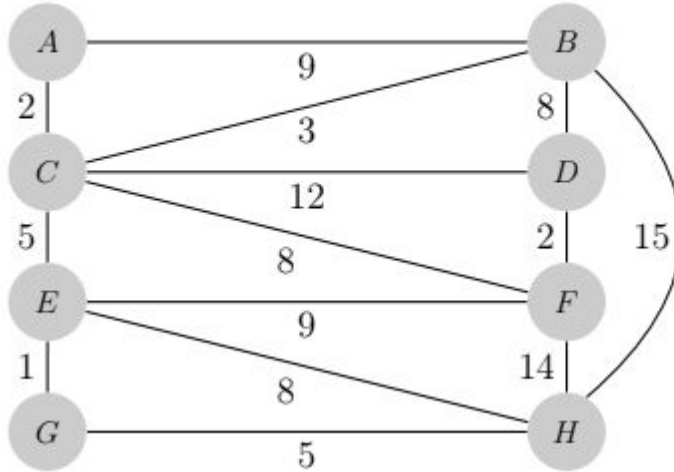


A	0
B	5
C	2
D	14
E	7
F	10
G	-
H	-



# Dijkstra's Algorithm

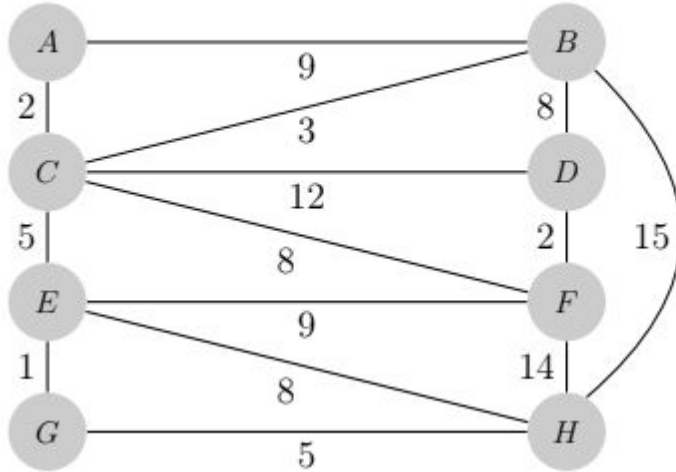
Apply Dijkstra's algorithm to the following graph, starting from vertex A. What is the order in which vertices are marked as known?



A	0
B	5
C	2
D	13
E	7
F	10
G	-
H	20

# Dijkstra's Algorithm

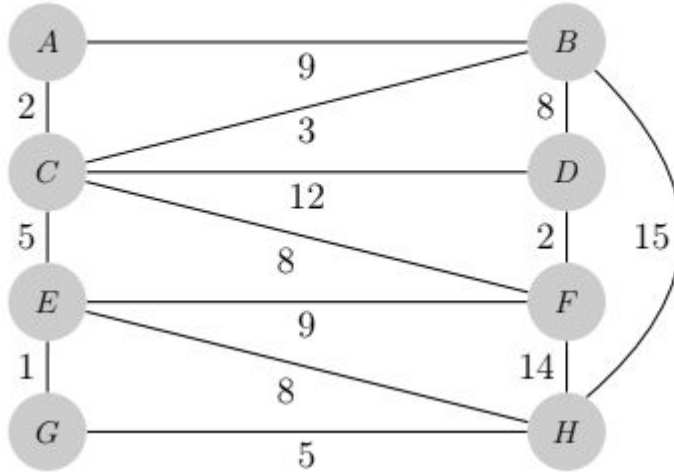
Apply Dijkstra's algorithm to the following graph, starting from vertex A. What is the order in which vertices are marked as known?



A	0
B	5
C	2
D	13
E	7
F	10
G	8
H	13

# Dijkstra's Algorithm

Apply Dijkstra's algorithm to the following graph, starting from vertex A. What is the order in which vertices are marked as known?



A	0
B	5
C	2
D	12
E	7
F	10
G	8
H	13

ACBEGFDH

# Algorithm Families

# Algorithm Families

- Brute-Force
- Greedy
- Divide and Conquer
- Dynamic Programming (also, later)
- Backtracking
- Branch and Bound



# Brute-Force

- Simple - checks all possible solutions, chooses best/satisfactory
- Guaranteed optimal (if it's an optimization problem)
- Usually inefficient
  - Usually the solution space is enormous



# Greedy

- Use heuristics to make next choice
  - **Locally** optimal choice
- If locally optimal solutions do not lead to globally optimal then it will not give the optimal
  - E.g., Knapsack
- Sometimes, locally optimal choices **do** lead to globally optimum
  - E.g., sorting (which algorithms?)
- (Typically) faster than brute-force



# Divide and Conquer

- Divide a problem solution into two or more smaller problems, pref of equal size, solve the smaller instances, and combine the solutions
  - Or, bottom-up (combine & conquer): start from smallest divisions and combine into larger solutions
- Often recursive
- Often involves  $\log n$  complexities (look at the master theorem for why)
- Quicksort, mergesort





# Dynamic Programming Algorithms

- Remember partial solutions (memoization)
- Typically fast (i.e. faster than non-DP solution)
- Requires additional of memory
- Often looks like D&C, but used when not distinct subproblems
  - We “remember” the solutions to subproblems because the subproblems overlap



# Backtracking + Branch & Bound

# Backtracking / Branch&Bound

- Two related classes of algorithms
  - The code looks similar
- They solve **different types of problems**.



# Backtracking / Branch&Bound

- Two related classes of algorithms
  - The code looks similar
- They solve **different types of problems**.
- **Bracktracking** *is used to answer this question:*
  - “What choices should I make to satisfy some constraints?”
- **Branch+Bound** *is used to answer this question:*
  - “What choices should I make to obtain the *best* solution?”



# Golden Rule

Looking for *any* solution?

**=> Backtracking (constraints only)**

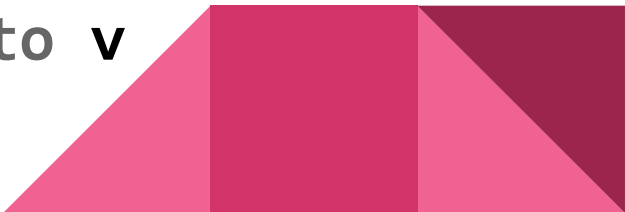
Looking for *best* solution?

**=> Branch + Bound (needs to calculate bound)**



# Backtracking pseudocode

```
algorithm checknode(node v)
  if satisfies_constraints(v)
    if is_solution(v)
      return solution
  else
    for each node u adjacent to v
      checknode(u)
```



# Branch and Bound pseudocode

```
algorithm find_best(node v)
```

```
    if is_solution(v)
```

```
        if score(v) > score(best)
```

```
            best = v
```

```
        return
```

```
    if best_possible_score(v) > score(best)
```

```
        for each child node u of v
```

```
            find_best(u)
```

(must initialize 'best' to some arbitrary solution)

(variable 'best' holds the best thing in the end)



# Knapsack



# Approaches

- Brute Force
- Greedy
- Dynamic Programming
- Backtracking
- Branch and Bound



# Approaches

- Brute Force
  - $n2^n$ , optimal
- Greedy
- Dynamic Programming
- Backtracking
- Branch and Bound



# Approaches

- Brute Force
  - $n \cdot 2^n$ , optimal
- Greedy
  - $O(n)$ , not optimal
- Dynamic Programming
- Backtracking
- Branch and Bound



# Approaches

- Brute Force
  - $n \cdot 2^n$ , optimal
- Greedy
  - $O(n)$ , not optimal
- Dynamic Programming
  - $O(nC)$ , optimal
  - $\text{Knap}(\text{int } n, \text{int } \text{Capacity}) = \max(\text{knap}(n-1, \text{Capacity}), \text{knap}(n-1, \text{Capacity} - \text{cost}[n]) + \text{value}[n])$
  - Requires integer weights, why?
- Backtracking
  - Constraint, doesn't work that well
- Branch and Bound
  - Initial estimate of best solution is lower bound on knapsack value
  - Need to estimate upper bound on remaining value of partial solution

# Dynamic Programming

# Dynamic Programming

- Use when you have smaller *shared/common* subproblems
- Basic Idea:
  - Save subproblem results in extra memory
  - Use extra memory to speed up



# Memoization

- “Top-down” DP
- Write a recursive function
  - At the start, check if we’ve already done this one
  - Before returning, store the result.




# Bottom Up

- Store the results in an array/map
- Build up the array/map starting from a base case
- Stop when you get to the input you're trying to compute



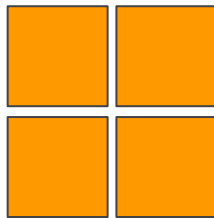


# How to solve them?

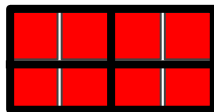
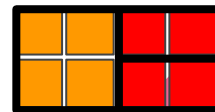
- Step 1: come up with a “subproblem”
    - Focus on how to **solve** a subproblem, not where it is used later
    - *I.e. “maximum number of ...” not “what you get if you ...”*
    - Which **other subproblems** can you use? What is the **final step** to combine the results of these subproblems?
  - Step 2: write a base case for that subproblem
  - Step 3: write a recurrence for that subproblem
  - Step 4: implement it bottom-up or top-down
- 

# Problem 2

2. How many ways are there to tile a  $2 \times n$  grid with the below tiles?



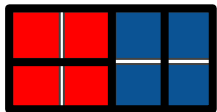
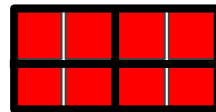
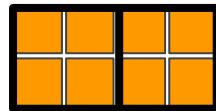
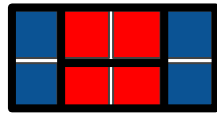
For  $n = 4$ , there are 11:



# Problem 2

## Step 1: Subproblem

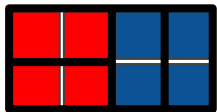
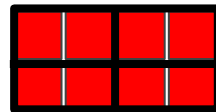
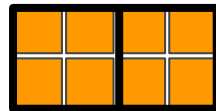
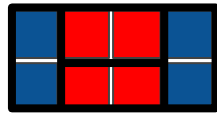
Let  $f(n)$  be the number of tilings on a  $2 \times n$  grid



# Problem 2

## Step 2: Base Cases

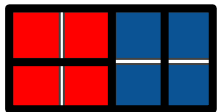
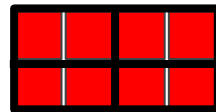
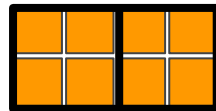
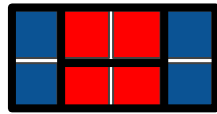
$$f(0) = 1; \quad f(1) = 1; \quad f(2) = 3;$$



# Problem 2

## Step 3: The Recurrence

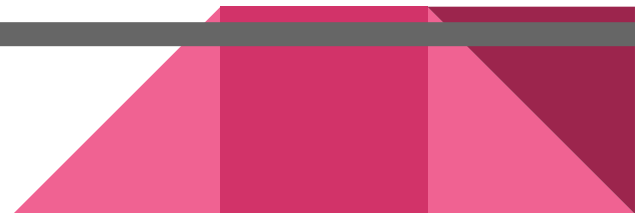
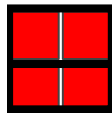
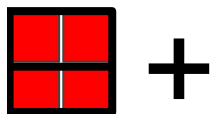
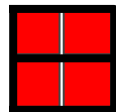
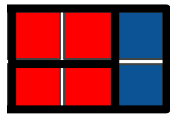
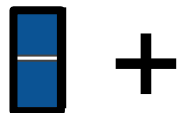
We can split tilings into three categories: leftmost is blue, orange, or red:



# Problem 2

## Step 3: The Recurrence

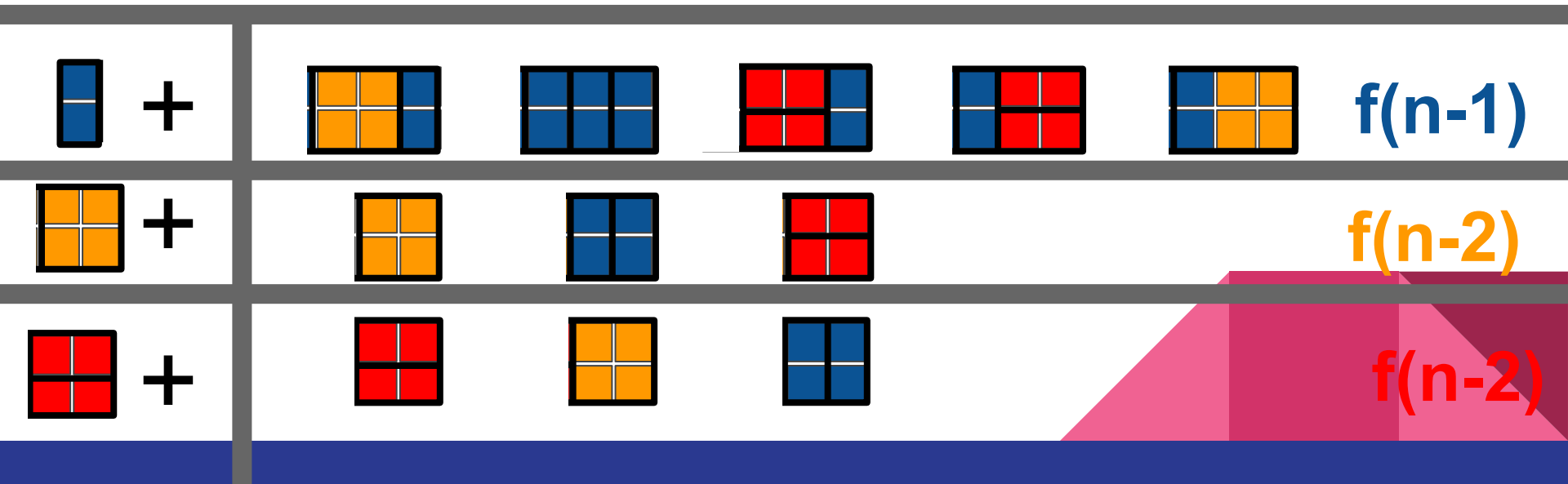
We can split tilings into three categories: leftmost is blue, orange, or red:



# Problem 2

## Step 3: The Recurrence

We can split tilings into three categories: leftmost is blue, orange, or red:

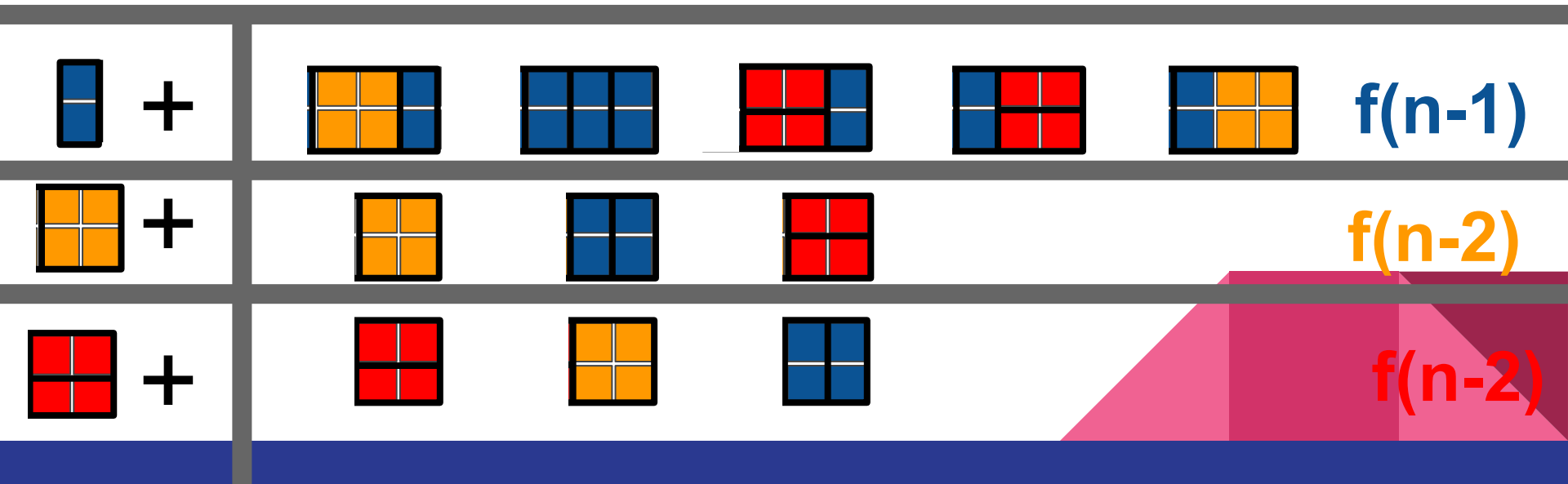


## Problem 2

$$f(n) = f(n-1) + f(n-2) + f(n-2)$$

### Step 3: The Recurrence

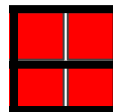
We can split tilings into three categories: leftmost is blue, orange, or red:





# Code

```
vector<int> answer(max(2, n+1)); // has size (n+1)
answer[0] = 1;
answer[1] = 1;
for(int i = 2; i <= n; i++){
    // either starts with a vertical bar (answer[i-1])
    // or two horizontal bars (answer[i-2])
    // or a 2x2 box (answer[i-2])
    answer[i] = answer[i-1] + answer[i-2] + answer[i-2];
}
return answer[n];
```




# Example 1

Consider the following function `biCoeff`, which computes the binomial coefficient ( $n$  choose  $m$ ).

```
int biCoeff(int n, int m) {  
    if (m == 0 || m == n) {  
        return 1;  
    }  
    else {  
        return biCoeff(n - 1, m - 1) + biCoeff(n - 1, m);  
    }  
} // biCoeff()
```

If we adapt `biCoeff` to use dynamic programming while continuing to use a top down approach, what does the memory complexity of `biCoeff` become?

- A.  $O(n)$
  - B.  $O(nm)$
  - C.  $O(n!)$
  - D.  $O(n + m)$
  - E.  $O(m)$
- 

# Example 1

Consider the following function `biCoeff`, which computes the binomial coefficient ( $n$  choose  $m$ ).

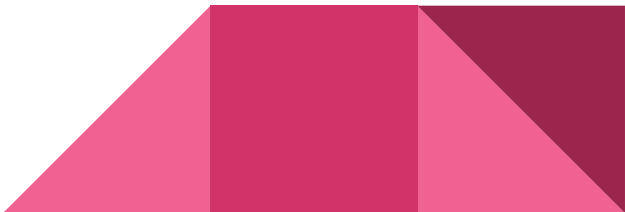
```
int biCoeff(int n, int m) {  
    if (m == 0 || m == n) {  
        return 1;  
    }  
    else {  
        return biCoeff(n - 1, m - 1) + biCoeff(n - 1, m);  
    }  
} // biCoeff()
```

If we adapt `biCoeff` to use dynamic programming while continuing to use a top down approach, what does the memory complexity of `biCoeff` become?

- A.  $O(n)$
- B.  $O(nm)$
- C.  $O(n!)$
- D.  $O(n + m)$
- E.  $O(m)$


## Example 2

Which of the following statements about dynamic programming is true?

- A. The top-down approach is always more efficient than the bottom-up approach
  - B. The bottom-up approach can be used anytime the top-down approach can be used
  - C. Dynamic programming is useful when a problem divides into independent sub-problems
  - D. When applying dynamic programming, the same subproblem is solved multiple times
  - E. None of the above
- 

## Example 2

Which of the following statements about dynamic programming is true?

- A. The top-down approach is always more efficient than the bottom-up approach
  - B. The bottom-up approach can be used anytime the top-down approach can be used
  - C. Dynamic programming is useful when a problem divides into independent sub-problems
  - D. When applying dynamic programming, the same subproblem is solved multiple times
  - E. None of the above
- 

# Protip

Follow these steps for an easy DP

1. What is the recursive solution to this problem?
2. How many possible inputs are there? (Size of memo)
3. Where can I use the saved information?



# Practice Questions

# Questions

- You are given maize blocks of length 1 and blue blocks of length 2. Find all the ways that you can arrange them in a line of length exactly  $N$ . Assume you have infinite maize and blue blocks.
- Recursive Formulation?
  - parameters?





# Questions

- You are given maize blocks of length 1 and blue blocks of length 2. Find all the ways that you can arrange them in a line of length exactly  $N$ . Assume you have infinite maize and blue blocks.
- Recursive Formulation
  - 1 parameter



# Questions

- You are given maize blocks of length 1 and blue blocks of length 2. Find all the ways that you can arrange them in a line of length exactly N. Assume you have infinite maize and blue blocks.
- Recursive Formulation
  - 1 parameter
- Possibilities(n) = ?



# Questions

- You are given maize blocks of length 1 and blue blocks of length 2. Find all the ways that you can arrange them in a line of length exactly  $N$ . Assume you have infinite maize and blue blocks.
- $\text{Possibilities}(n) = \text{Possibilities}(n-1) + \text{Possibilities}(n-2)$



Sequence of length  $n - 2$



Sequence of length  $n - 1$

# Extra Examples

# Time Traveling Trader

You've fixed up your time machine, and now you can bring back a whole lot more information. You've collected the prices for Stock X for ***n*** consecutive days:

$$\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$$

Back at day 0, you want to use this to make *as much money as possible*.

You want to avoid changing stock prices, so you can only make **one** trade per day.

In addition, you want to avoid attention, so you'll hold at most ***k*** units of stock.

You'll lose unsold stock when you go back to the future on day ***n***.

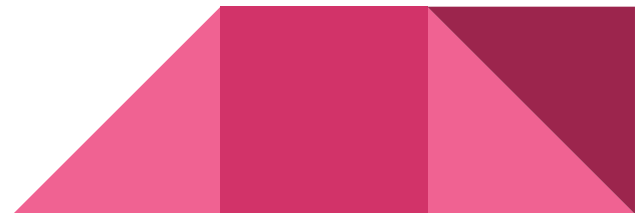
How can you maximize profit?



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most ***k*** units. Up to ***one*** trade per day.

***Step 1: The Sub-Problem***



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

## *Step 1: The Sub-Problem*

Let  $M(???)$  be the max profit you can make **when** ???



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

## ***Step 1: The Sub-Problem***

Let  $M(i, ???)$  be the max profit you can make starting on the morning of day  **$i$**  and also ???





# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

## ***Step 1: The Sub-Problem***

Let  $M(i, u)$  be the max profit you can make starting on the morning of day  **$i$**  while already owning  **$u$**  units of stock.



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

## ***Step 1: The Sub-Problem***

Let  $M(i, u)$  be the max profit you can make starting on the morning of day  **$i$**  while already owning  **$u$**  units of stock.

The answer to the overall problem is  $M(0, 0)$ . ***This is therefore not the base-case.***



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

## ***Step 1: The Sub-Problem***

Let  $M(i, u)$  be the max profit you can make starting on the morning of day  **$i$**  while already owning  **$u$**  units of stock.

The answer to the overall problem is  $M(0, 0)$ . ***This is therefore not the base-case.***

## ***Step 2: The Base Case***



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

## ***Step 1: The Sub-Problem***

Let  $M(i, u)$  be the max profit you can make starting on the morning of day  $i$  while already owning  $u$  units of stock.

The answer to the overall problem is  $M(0, 0)$ . ***This is therefore not the base-case.***

## ***Step 2: The Base Case***

On the morning of day  $n$ , you go home.



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

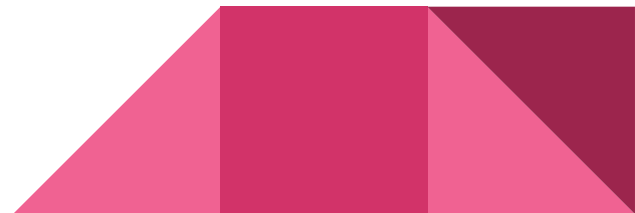
## ***Step 1: The Sub-Problem***

Let  $M(i, u)$  be the max profit you can make starting on the morning of day  $i$  while already owning  $u$  units of stock.

The answer to the overall problem is  $M(0, 0)$ . ***This is therefore not the base-case.***

## ***Step 2: The Base Case***

On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

**Step 1: The Sub-Problem**  $M(i, u)$  = max profit on morning  **$i$**  with  **$u$**  units of stock

**Step 2: The Base Case** On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .

**Step 3: The Recurrence** for  $M(i, u)$



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

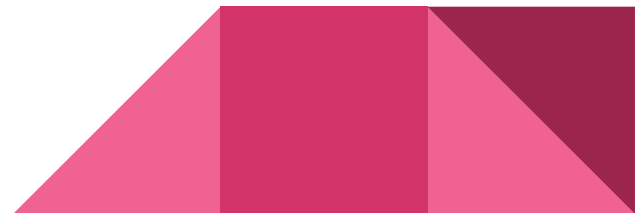
**Step 1: The Sub-Problem**  $M(i, u)$  = max profit on morning  **$i$**  with  **$u$**  units of stock

**Step 2: The Base Case** On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .

**Step 3: The Recurrence** for  $M(i, u)$

3 options:

- Nothing: only when ???
- Buy: only when ???
- Sell: only when ???



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

**Step 1: The Sub-Problem**  $M(i, u)$  = max profit on morning  **$i$**  with  **$u$**  units of stock

**Step 2: The Base Case** On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .

**Step 3: The Recurrence** for  $M(i, u)$

3 options:

- Nothing:  $M(i+1, u)$
- Buy: only when ???
- Sell: only when ???





# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

**Step 1: The Sub-Problem**  $M(i, u)$  = max profit on morning  **$i$**  with  **$u$**  units of stock

**Step 2: The Base Case** On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .

**Step 3: The Recurrence** for  $M(i, u)$

3 options:

- Nothing:  $M(i+1, u)$
- Buy: only when  $u < k$
- Sell: only when  $u > 0$



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

**Step 1: The Sub-Problem**  $M(i, u)$  = max profit on morning  **$i$**  with  **$u$**  units of stock

**Step 2: The Base Case** On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .

**Step 3: The Recurrence** for  $M(i, u)$

3 options:

- Nothing:  $M(i+1, u)$
- Buy:  $M(i+1, u+1) - p_i$  only when  $u < k$
- Sell: only when  $u > 0$



# Time Traveling Trader

Prices:  $\{p_0, p_1, p_2, \dots, p_{n-2}, p_{n-1}\}$ . Hold at most  **$k$**  units. Up to **one** trade per day.

**Step 1: The Sub-Problem**  $M(i, u)$  = max profit on morning  **$i$**  with  **$u$**  units of stock

**Step 2: The Base Case** On the morning of day  $n$ , you go home. So  $M(n, u) = 0$ .

**Step 3: The Recurrence** for  $M(i, u)$

3 options:

- Nothing:  $M(i+1, u)$
- Buy:  $M(i+1, u+1) - p_i$  only when  $u < k$
- Sell:  $M(i+1, u-1) + p_i$  only when  $u > 0$



# Time Traveling Trader

```
double best_profit(  
    const vector<double>& p, int n, int k, int i, int u  
) {  
    if (i >= n)  
        return 0;  
  
    double best = best_profit(p, n, k, i+1, u);  
  
    if (u < k)  
        best = max(best, best_profit(p, n, k, i+1, u+1) - p[i]);  
  
    if (u > 0)  
        best = max(best, best_profit(p, n, k, i+1, u-1) + p[i]);  
  
    return best;  
}
```

**Base Case:**  $M(n, u) = 0$ .

**Recurrence** for  $M(i, u)$ :

- Neither:  $M(i+1, u)$
- Buy:  $M(i+1, u+1) - p_i$   
when  $u < k$
- Sell:  $M(i+1, u-1) + p_i$   
when  $u > 0$

# Time Traveling Trader

**Base Case:**  $M(n, u) = 0$ .

**Recurrence** for  $M(i, u)$ :

- Neither:  $M(i+1, u)$
- Buy:  $M(i+1, u+1) - p_i$   
when  $u < k$
- Sell:  $M(i+1, u-1) + p_i$   
when  $u > 0$

Runtime:  
 $O(nk)$

```
double best_profit(  
    const vector<double>& p, int n, int k, int i, int u,  
    vector<vector<double>>& memo  
) {  
  
    if (i >= n)  
        return 0;  
    if (memo[i][u] != -INFINITY) return memo[i][u];  
  
    double best = best_profit(p, n, k, i+1, u);  
  
    if (u < k)  
        best = max(best, best_profit(p, n, k, i+1, u+1, memo) - p[i]);  
  
    if (u > 0)  
        best = max(best, best_profit(p, n, k, i+1, u-1, memo) + p[i]);  
    memo[i][u] = best;  
    return best;  
}
```

# Rod Cutting

You have a wood plank whose length is  $N$  inches.

You will cut the plank into integer lengths, and sell each of them.

A piece of length  $L$  will sell for  $P[L]$  dollars.

These prices might not be “uniform”:  $P[L] / L$  is not a constant!

You want to make as much money as possible.



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

## Step 1: The Subproblem



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

## Step 1: The Subproblem

Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.





# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

## Step 1: The Subproblem

Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

## Step 2: The Base Case



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

## Step 1: The Subproblem

Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

## Step 2: The Base Case

$$M(0) = ???$$



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

## Step 1: The Subproblem

Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

## Step 2: The Base Case

$$M(0) = 0$$



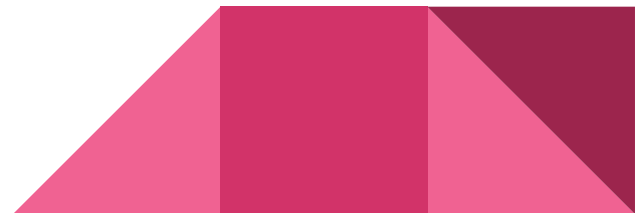
# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence**



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is ???

Cut at  $L$ : Profit is ???



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is  $P[N]$

Cut at  $L$ : Profit is ???



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is  $P[N]$

Cut at  $L$ : Profit is  $P[L] + ???$





# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is  $P[N]$

Cut at  $L$ : Profit is  $P[L] + M(N - L)$



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is  $P[N]$

But which  $L$  is best?

Cut at  $L$ : Profit is  $P[L] + M(N - L)$



# Rod Cutting: Solution

You have a wood plank whose length is  $N$  inches.

**Step 1: The Subproblem** Let  $M(N)$  be the maximum amount of money that you can sell an  $N$ -length plank, by first (possibly) cutting it into many pieces.

**Step 2: The Base Case**  $M(0) = 0$

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is  $P[N]$

Cut at  $L$ : Profit is  $P[L] + M(N - L)$

But which  $L$  is best?  
Dunno-  
let's try them all!



# Rod Cutting: Solution

**Step 3: Recurrence** Two cases: we sell the plank whole, or we cut it at least once.

No Cut: Profit is  $P[N]$


Cut at  $L$ : Profit is  $P[L] + M(N - L)$

So,

$$M(N) = \max\{ P[N], P[N-1] + M(1), P[N-2] + M(2), \dots, P[2] + M(N-2), P[1] + M(N-1) \}$$

# Write the DP - bottom up Step 4

```
int cutRod(vector<int> P, int N) {  
    assert(N < P.size());  
    vector<int> M(N+1, -1);  
    M[0] = 0;  
    for (int i = 1; i <= N; i++) {  
        int best = P[i];  
        for (int L = 1; L < i; L++)  
            best = max(best, P[L] + M[i - L]);  
  
        M[i] = best;  
    }  
    return M[N];  
}
```



# Write the DP - bottom up Step 4

```
int cutRod(vector<int> P, int N) {  
    assert(N < P.size());  
    vector<int> M(N+1, -1);  
    M[0] = 0;  
    for (int i = 1; i <= N; i++) {  
        int best = P[i];  
        for (int L = 1; L < i; L++)  
            best = max(best, P[L] + M[i - L]);  
  
        M[i] = best;  
    }  
    return M[N];  
}
```

$O(N^2)$



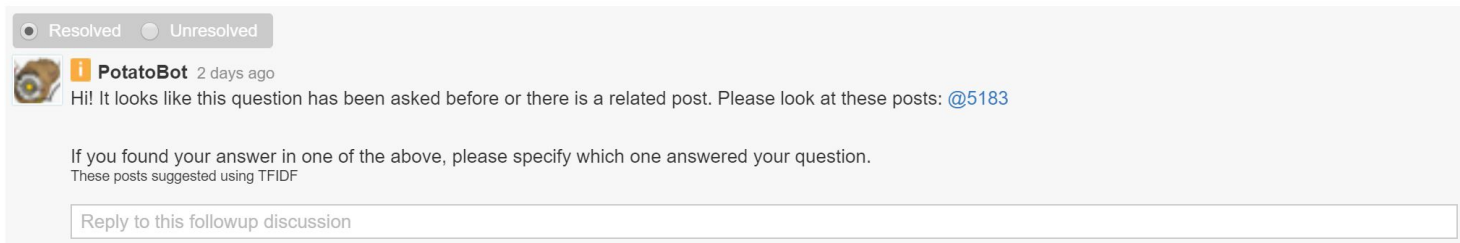
# Potatobot Planning



# Potatobot Planning

Potatobot reads all of the Piazza posts, and passes judgement upon them.

Some judgements take a while to compute:

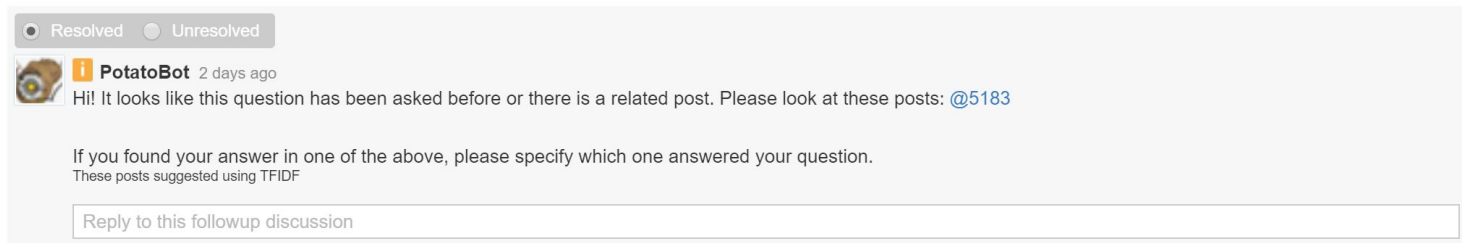




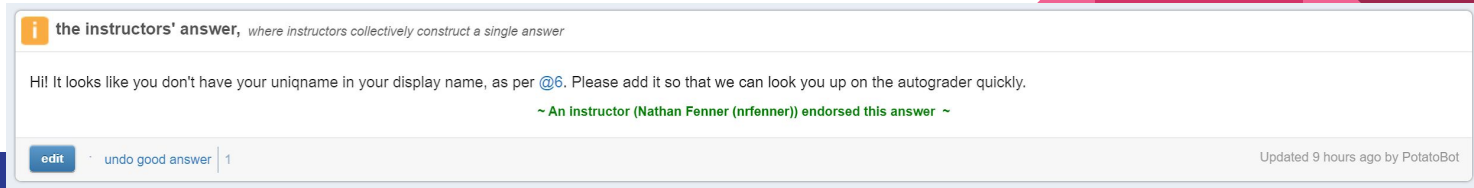
# Potatobot Planning

Potatobot reads all of the Piazza posts, and passes judgement upon them.

Some judgements take a while to compute:



Others are pretty quick:



# Potatobot Planning

Potatobot fell asleep, and when it woke up, there were  $n$  questions needing answers. Potatobot knows how long it will take to respond to each question:

$q_1^{\text{ms}}, q_2^{\text{ms}}, q_3^{\text{ms}}, q_4^{\text{ms}}, \dots, q_n^{\text{ms}}$



# Potatobot Planning

Potatobot fell asleep, and when it woke up, there were  $n$  questions needing answers. Potatobot knows how long it will take to respond to each question:

$q_1^{\text{ms}}, q_2^{\text{ms}}, q_3^{\text{ms}}, q_4^{\text{ms}}, \dots, q_n^{\text{ms}}$

It will take Potatobot  $(q_1 + q_2 + \dots + q_n)$  ms to answer them all.



# Potatobot Planning

Potatobot fell asleep, and when it woke up, there were  $n$  questions needing answers. Potatobot knows how long it will take to respond to each question:

$q_1^{\text{ms}}, q_2^{\text{ms}}, q_3^{\text{ms}}, q_4^{\text{ms}}, \dots, q_n^{\text{ms}}$

It will take Potatobot  $(q_1 + q_2 + \dots + q_n)^{\text{ms}}$  to answer them all.

Luckily, P.bot has a ***TURBO MODE*** for this kind of emergency!

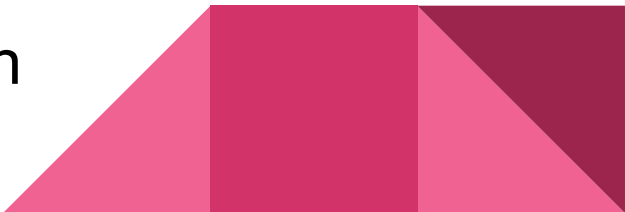


# Potatobot Planning

Potatobot is kept chilled to 0 °K for optimal performance, but ***TURBO MODE*** makes Potatobot heat up:

	Time	Temperature Change
<b><i>TURBO MODE</i></b>	1 ms	+q °C
Regular	q ms	-q °C

Running in regular mode lets P.bot cool down at the rate of 1°C / ms, so -q °C total.



# Potatobot Planning

If Potatobot reaches 300 K...



# Potatobot Planning

If Potatobot reaches 300 K...



# Potatobot Planning

If Potatobot reaches 300 K...



So we can't run ***TURBO MODE*** for too long.



# Potatobot Planning

Questions:

$q_1^{\text{ms}}, q_2^{\text{ms}}, q_3^{\text{ms}}, q_4^{\text{ms}}, \dots, q_n^{\text{ms}}$

$q^{\text{ms}}$ question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	$+q^{\circ}\text{C}$
Regular	$q^{\text{ms}}$	$-q^{\circ}\text{C}$

- Cannot reach  $300^{\circ}\text{K}$
- Start at  $0^{\circ}\text{K}$ 
  - “Safe” to go below  $0^{\circ}\text{K}$  but nothing happens

What's the minimum time to respond to all questions?



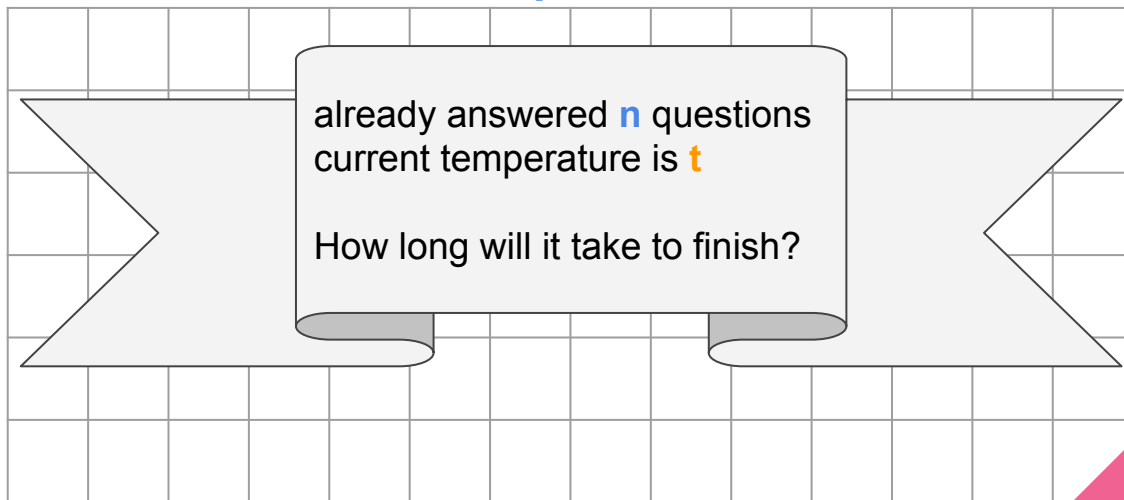
# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens

q ms question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	+q °C
Regular	q ms	-q °C

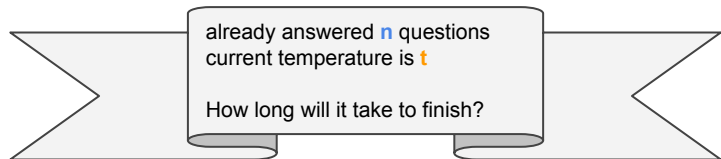
**n : questions**

**t: temperature**

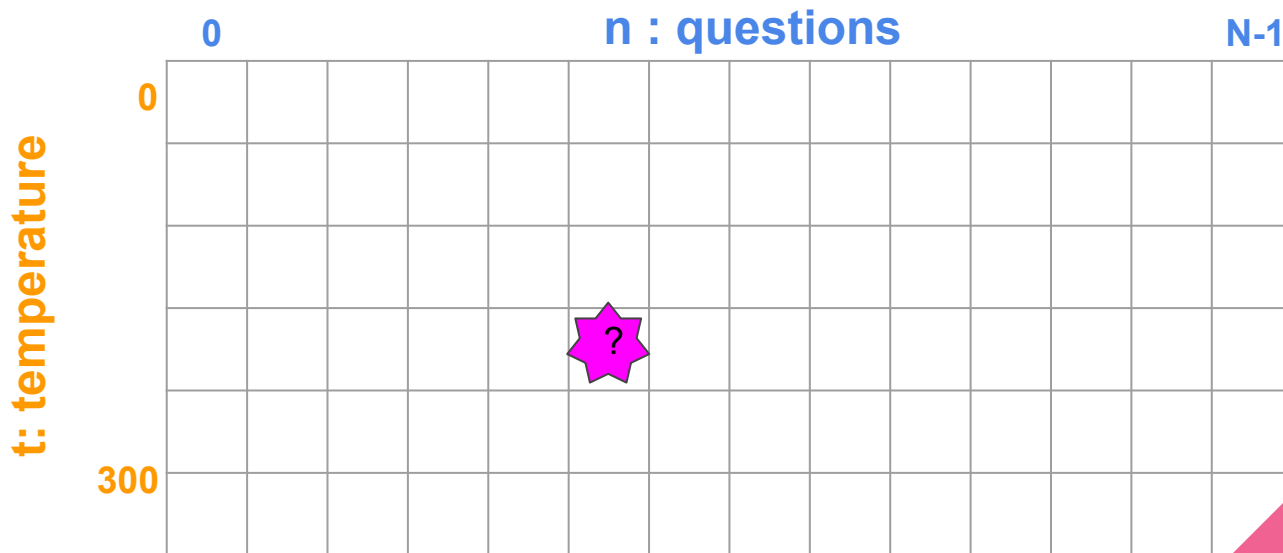


# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens

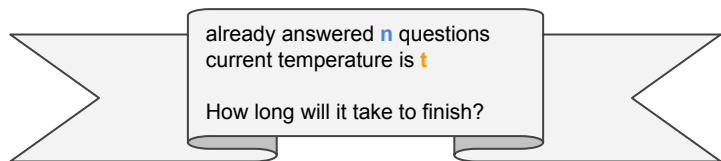


$q$ ms question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	$+q$ °C
Regular	$q$ ms	$-q$ °C

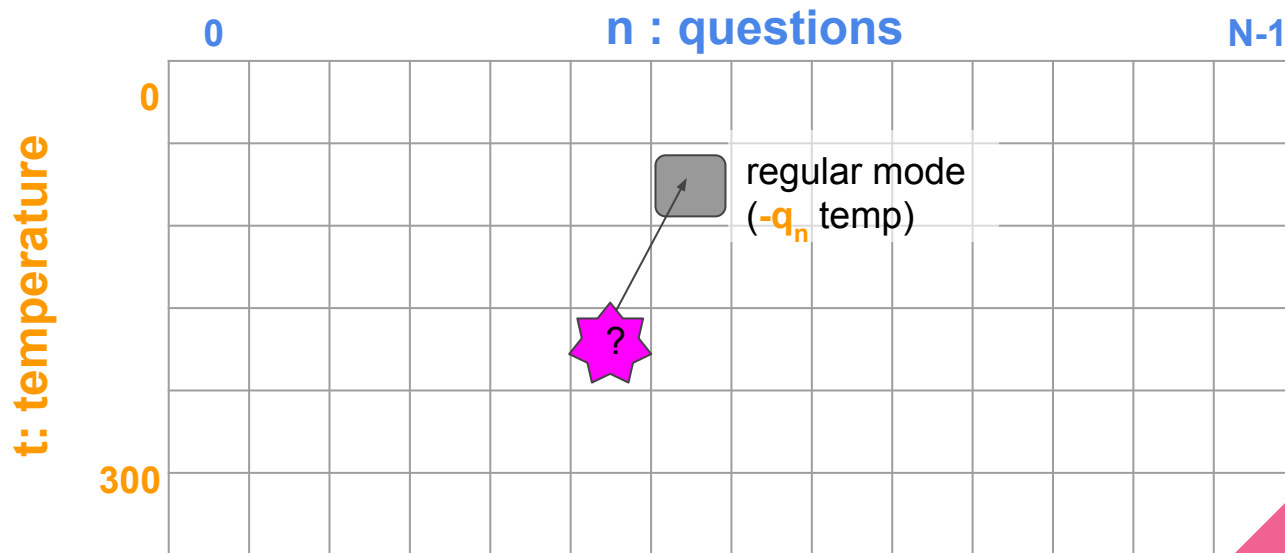


# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens

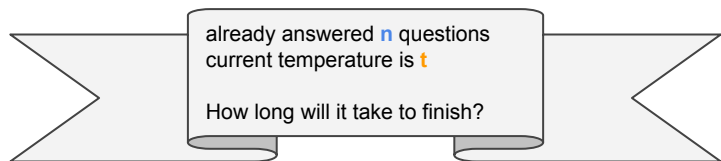


$q$ ms question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	$+q$ °C
Regular	$q$ ms	$-q$ °C

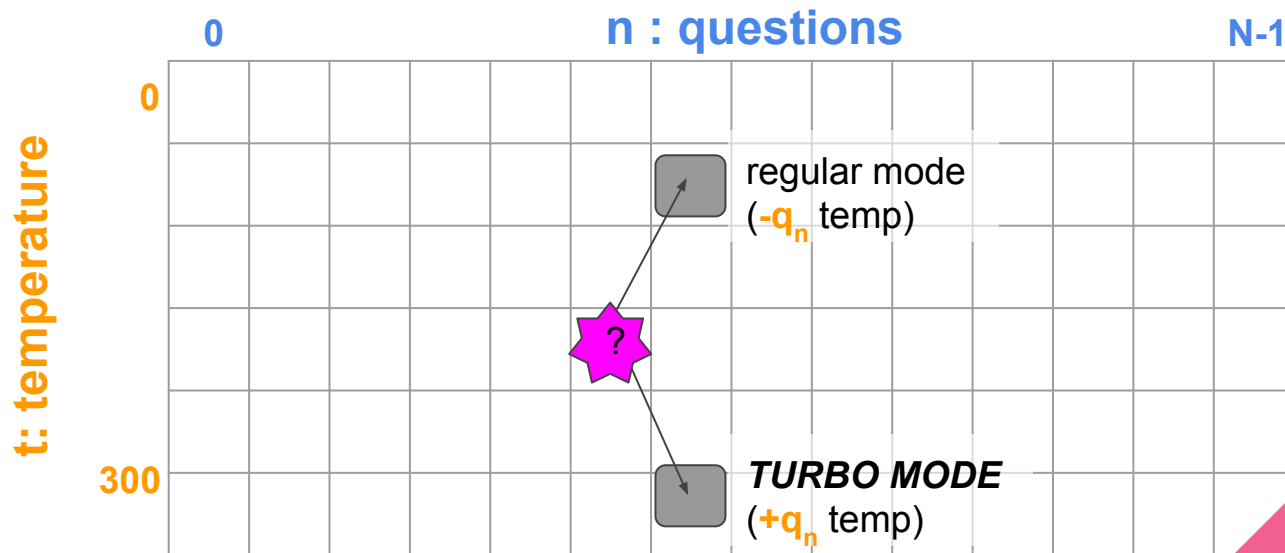


# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens

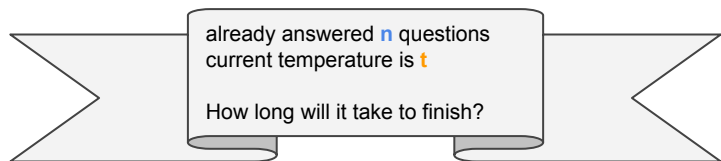


$q$ ms question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	$+q$ °C
Regular	$q$ ms	$-q$ °C

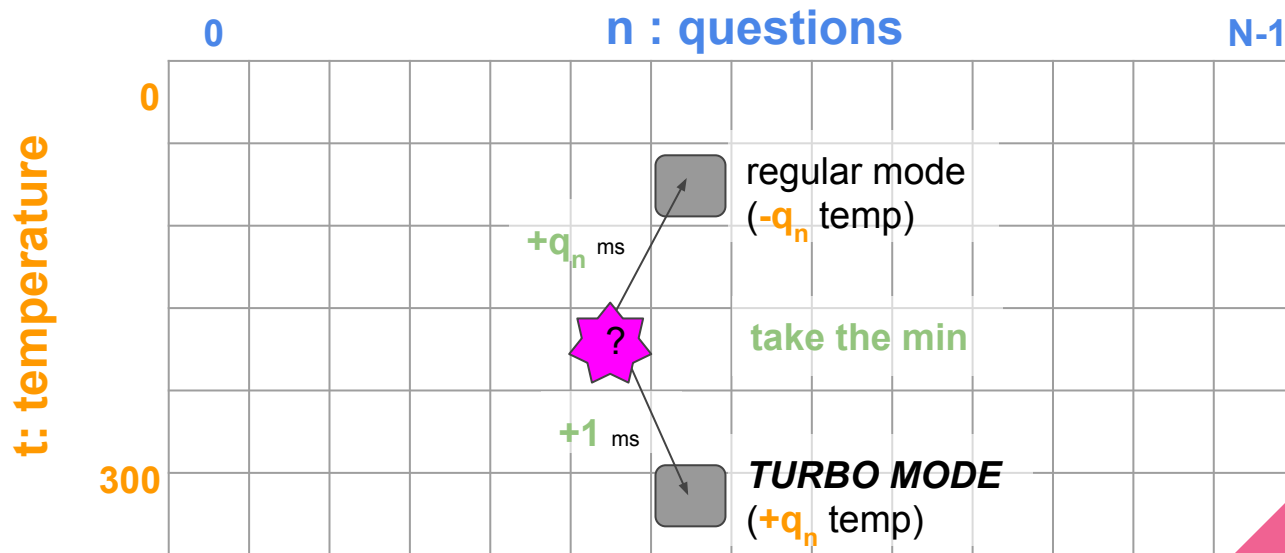


# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens

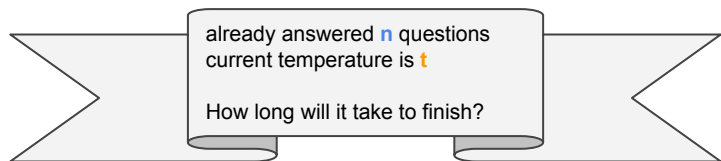


$q$ ms question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	$+q$ °C
Regular	$q$ ms	$-q$ °C

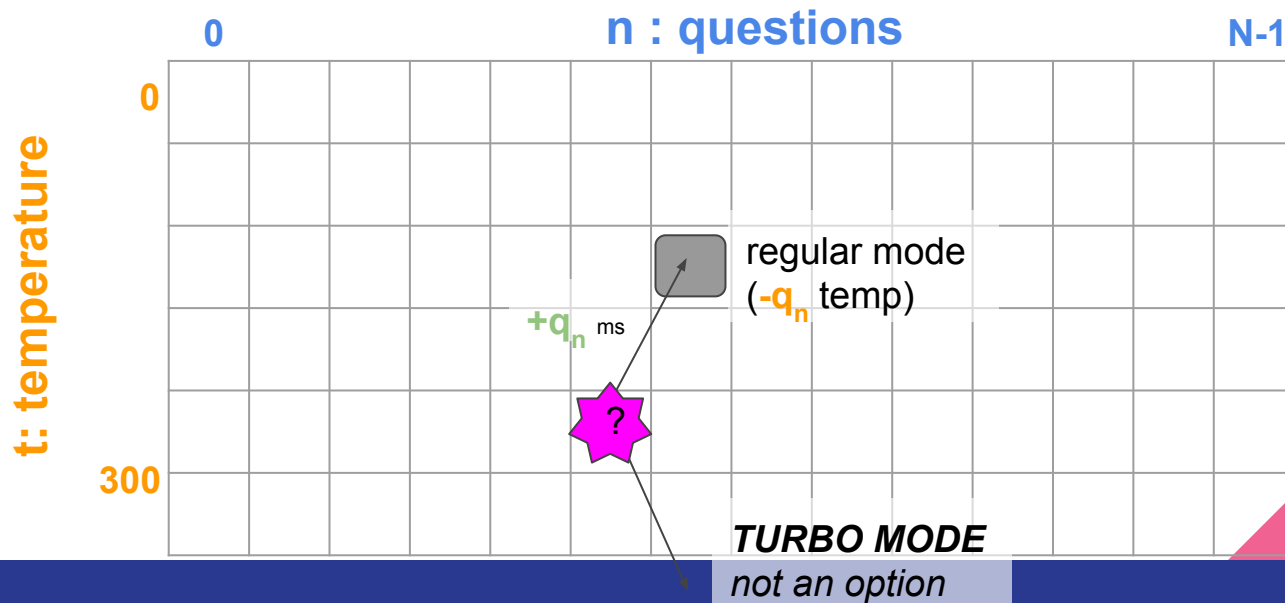


# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens



$q$ ms question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	$+q$ °C
Regular	$q$ ms	$-q$ °C



# Potatobot Planning

- Cannot reach 300 °K
- Start at 0 °K
  - “Safe” to go below 0°K but nothing happens

```
int total_time(vector<int> qs) {  
    vector<vector<int>> best(qs.size()+1, vector<int>(300, 0));  
    for (int i = qs.size()-1; i >= 0; i--) {  
        for (int t = 0; t < 300; t++) {  
            best[i][t] = best[i+1][max(0, t-qs[i])] + qs[i];  
            if (qs[i] + t < 300) {  
                int v = best[i+1][t+qs[i]] + 1;  
                if (v < best[i][t])  
                    best[i][t] = v;  
            }  
        }  
    }  
    return best[0][0];  
}
```

q <sub>ms</sub> question	Time	Temperature Change
<b>TURBO MODE</b>	1 ms	+q °C
Regular	q ms	-q °C