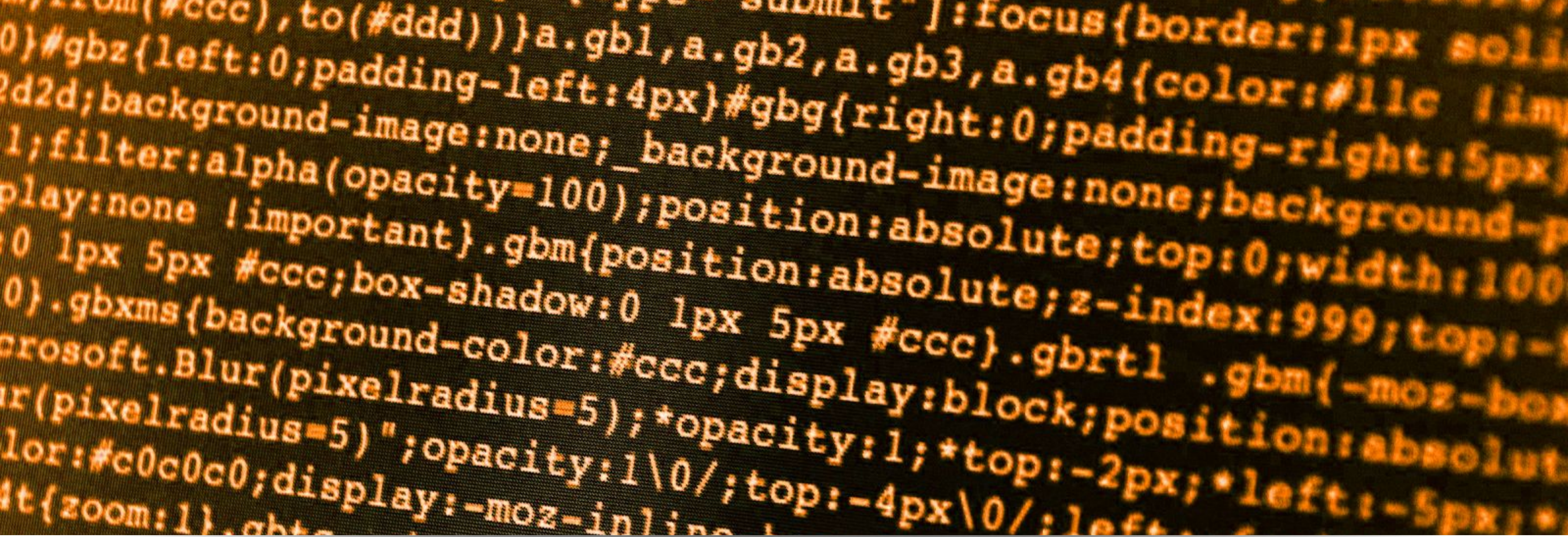


10

November 19th-25th, 2024

Algorithm Families and Dynamic Programming

Announcements

- Lab 9 Quiz (No AG) Due **Monday, November 25th 11:59pm**
- Lab 10 Handwritten Due **Monday, November 25th IN LAB**
- Project 4 Due **Monday, December 9th 11:59pm**
- Lab 10 Quiz & AG Due **Monday, December 2nd 11:59pm**
- Final Feedback Survey: Due **Monday, December 2nd 11:59pm**
  - https://forms.gle/5tbmDmviYQpgVPPq9
- **Final Exam Friday December 13th @ 8:00-10:00 AM EDT**
  - For alternate exam requests, fill out the form on canvas
    - also here: Alternate exam link
    - Any other concerns, email eecs281admin@umich.edu immediately.

Agenda

- Lab 9 Handwritten Review
- Dijkstra's Algorithm
- Generating Permutations
- Algorithm Families
- Dynamic Programming
- Handwritten Problem

Handwritten Problem Review

Handwritten Problem

Given an **undirected** connected graph, check if the graph contains a cycle.

```
bool is_graph_cyclic(vector<vector<int>> const& adj_list);
```

Feel free to use a helper function!

You can assume that if u and v are adjacent, then `adj_list[u]` will contain v and `adj_list[v]` will contain u. Also, u will not appear in `adj_list[u]`.

Complexity: O(V + E) time

Attempt 1: DFS

```
bool is_graph_cyclic_util(vector<vector<int>> const& adj_list,
                          vector<bool>& visited, int u, int parent) {
    visited[u] = true;
    for (int const neighbor : adj_list[u]) {
        if (neighbor != parent) {
            if (visited[neighbor]) {
                return true;
            }
        }
        if (is_graph_cyclic_util(adj_list, visited, neighbor, u)) {
            return true;
        }
    }
    return false;
}

bool is_graph_cyclic(vector<vector<int>> const& adj_list) {
    vector<bool> visited(adj_list.size(), false);
    return is_graph_cyclic_util(adj_list, visited, 0, -1);
}
```

There are two bugs in this code which cause segfaults. What are the bugs?

Solution 1: DFS

```
bool is_graph_cyclic_util(vector<vector<int>> const& adj_list,
                          vector<bool>& visited, int u, int parent) {
    visited[u] = true;
    for (int const neighbor : adj_list[u]) {
        if (neighbor != parent) {
            if (visited[neighbor]) {
                return true;
            }
            if (is_graph_cyclic_util(adj_list, visited, neighbor, u)) {
                return true;
            }
        }
    }
    return false;
}

bool is_graph_cyclic(const vector<vector<int>> &adj_list) {
   vector<bool> visited(adj_list.size(), false);
   return not adj_list.empty() and is_graph_cyclic_util(adj_list, visited, 0, -1);
}
```

Only perform recursive call if neighbor != parent.

Empty graphs don't have cycles.

Solution 2: BFS

```
bool is_graph_cyclic(vector<vector<int>> const& adj_list) {
    if (adj_list.empty()) return false;
    vector<int> parent(adj_list.size(), -1);
    queue<int> frontier;
    parent[0] = -2;
    frontier.push(0);
    while (not frontier.empty()) {
        int const current = frontier.front();
        frontier.pop();
        for (int neighbor : adj_list[current]) {
            if (neighbor != parent[current]) {
                if (parent[neighbor] != -1) {
                    return true;
                }
                parent[neighbor] = current;
                frontier.push(neighbor);
            }
        }
    }
    return false;
}
```

Solution 3: Counting Edges

```
bool is_graph_cyclic(vector<vector<int>> const& adj_list) {
    if (adj_list.empty()) {
        return false;
    }
    // if the number of edges = the number of vertices - 1 it’s acyclic
    int const num_vertices = adj_list.size();
    int num_edges = 0;
    for (auto const& vertex : adj_list) {
        num_edges += vertex.size();
    }
    num_edges /= 2;
    return num_edges != num_vertices - 1;
}
```

A connected undirected graph is
acyclic exactly when it’s a tree.

Dijkstra's Algorithm

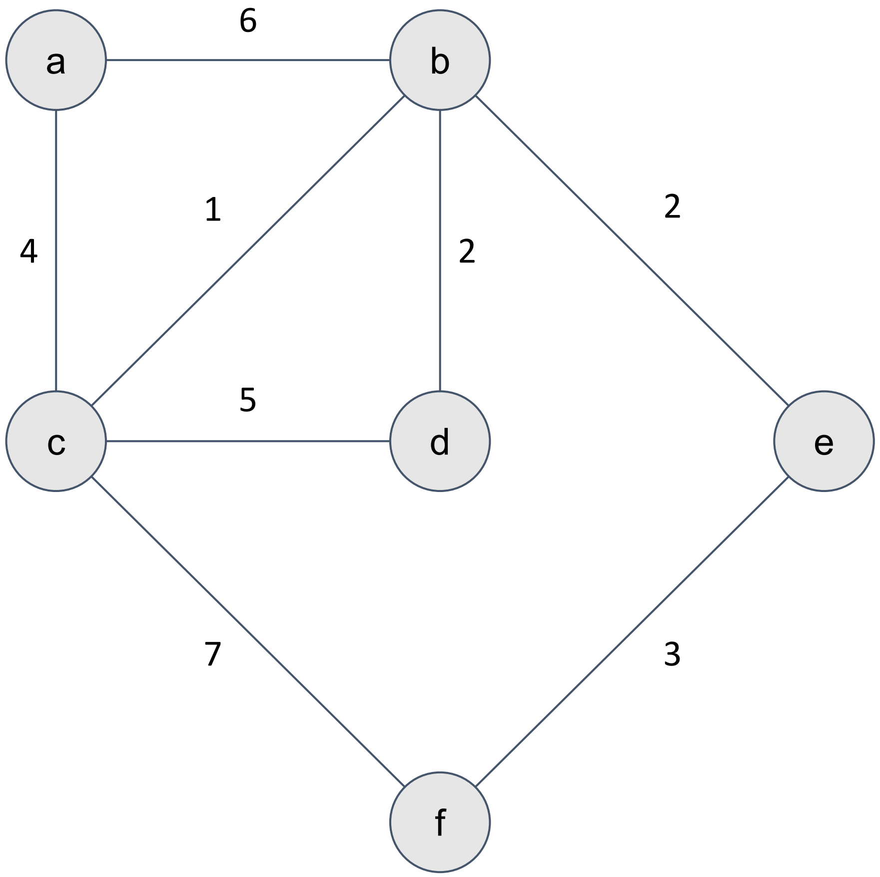
Dijkstra’s Algorithm Overview

- Shortest Path algorithm between a “source” node and all other nodes in a graph
- Considered a “greedy” algorithm
- Uses the **weight** of an edge to minimize the total distance
- Main differences from Prim’s algorithm:
  - finds shortest path, not MST
  - can work on BOTH directed and undirected graphs

Dijkstra's algorithm does not necessarily yield the correct solution in graphs containing negative edge weights, while Prim's algorithm can handle this.

Dijkstra's Algorithm

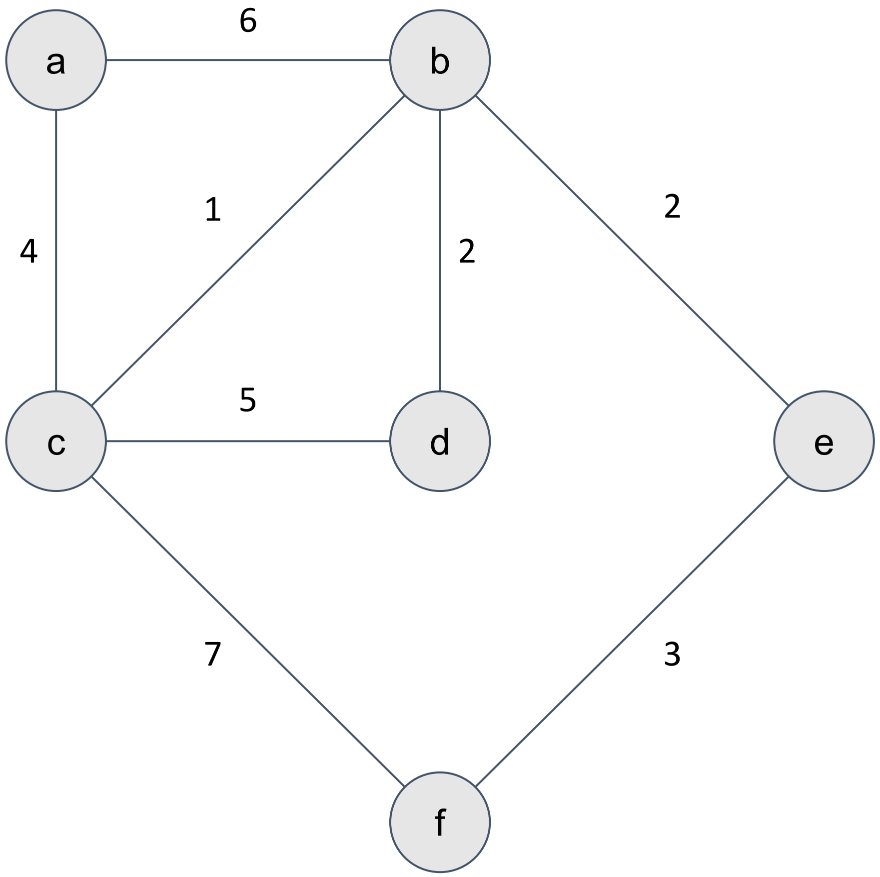
Problem: Given a graph with non-negative edge weights, find the shortest path from a starting vertex *s* to every other vertex *v*.



Dijkstra's Algorithm

Method: Dijkstra's Algorithm

- Greedy algorithm
- Similar to Prim's: use a table that tracks for each v:
  - Boolean k_v for if the shortest path from s to v is known
    - Initially false
  - Current shortest distance d_v from s to v
    - Initially infinity
  - Previous vertex p_v of v
    - Initially unknown

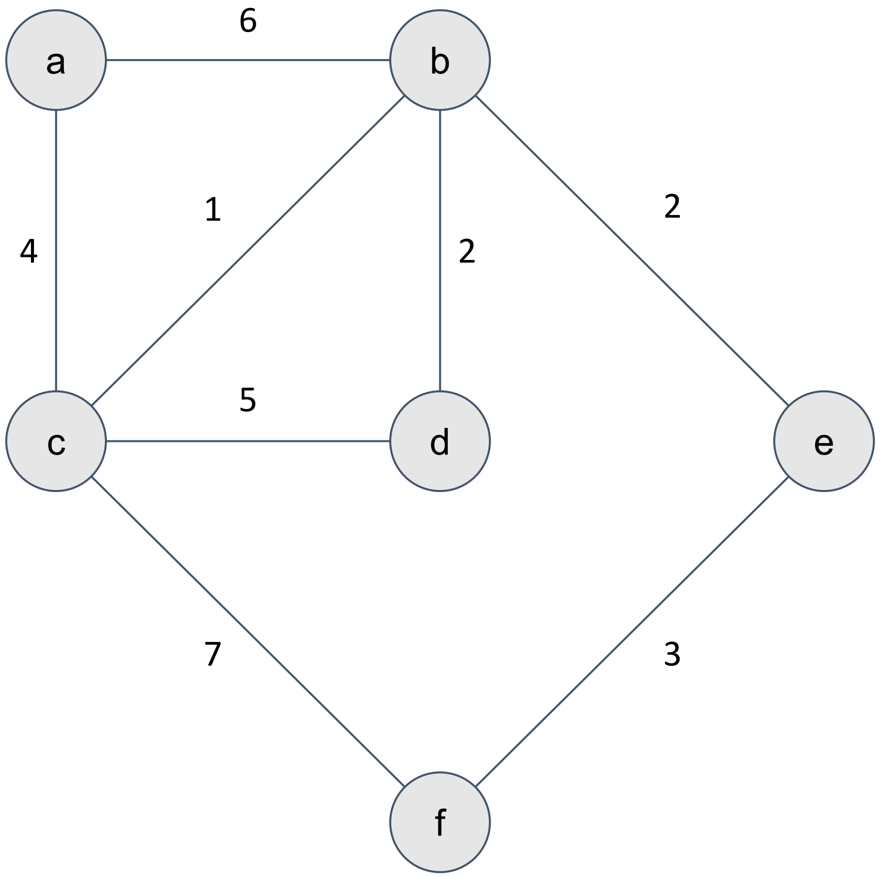


Dijkstra’s Algorithm

Method: Dijkstra’s Algorithm

- Difference from Prim’s:

  Distance update also uses distance from *s* to *v* instead of just the distance between adjacent vertices



Add the vertex nearest to the source, not the vertex nearest to the tree

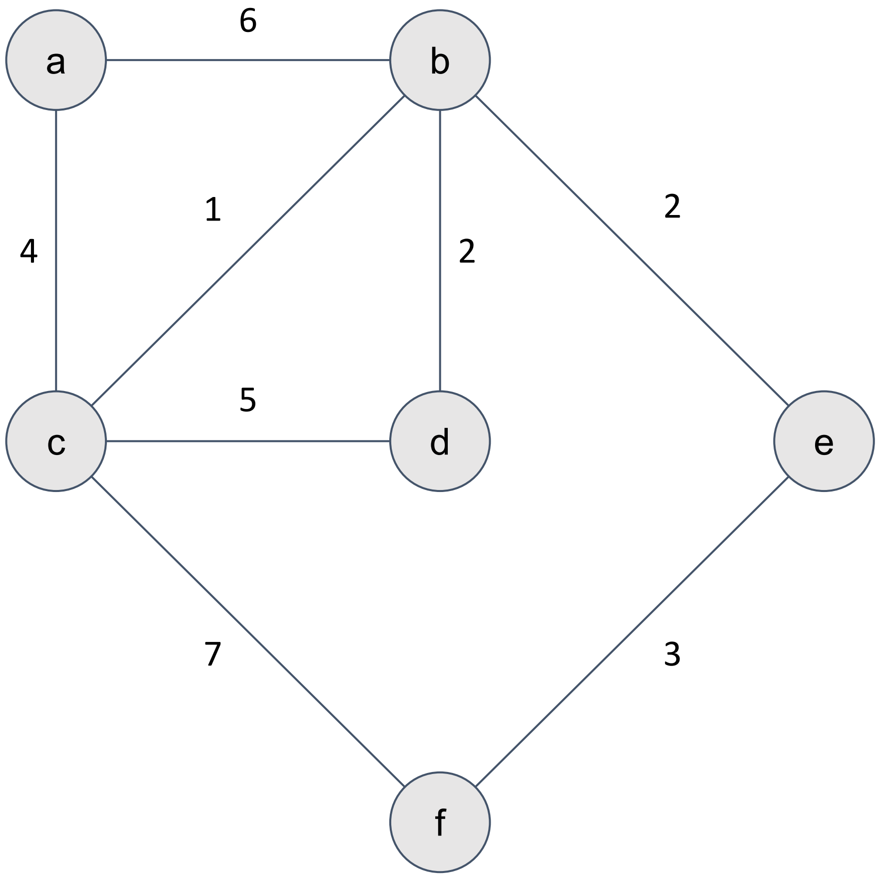
Dijkstra's Algorithm

Algorithm:

Set the distance for *s* to be 0

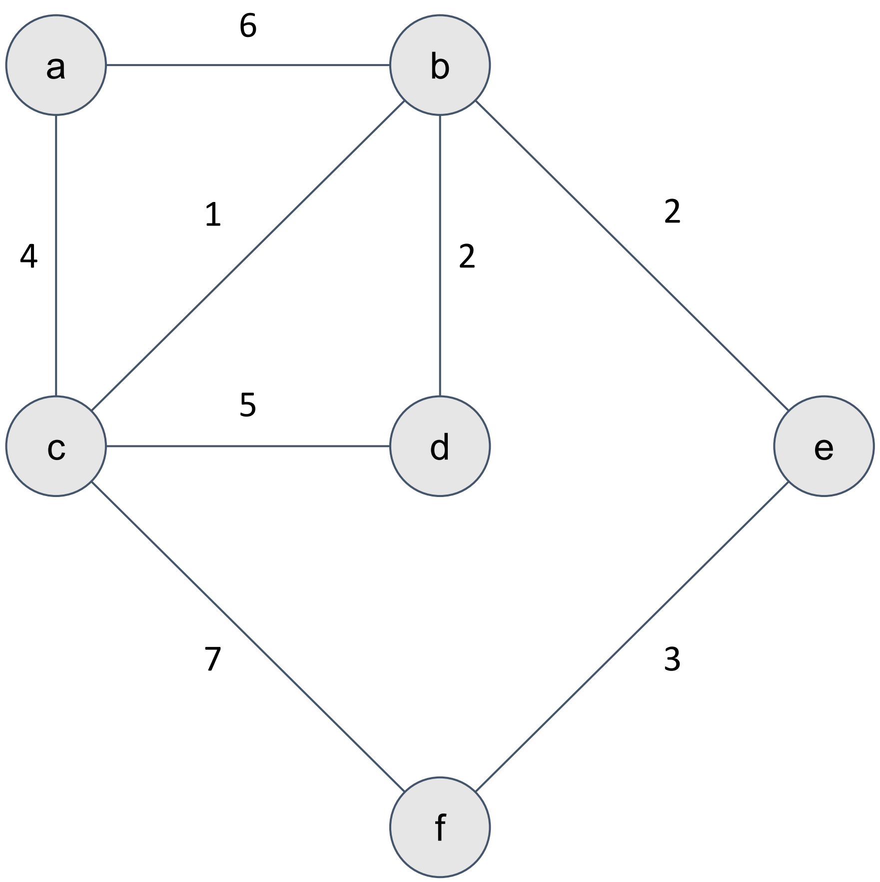
Loop |V| times:

1. Find unvisited vertex *v* with the shortest distance
2. Mark k_v as visited/true
3. For each adjacent unvisited vertex *u* do:
   a. If d_v + weight(v , u) < d_u then
      i. Update d_u to be that sum and p_u to be v



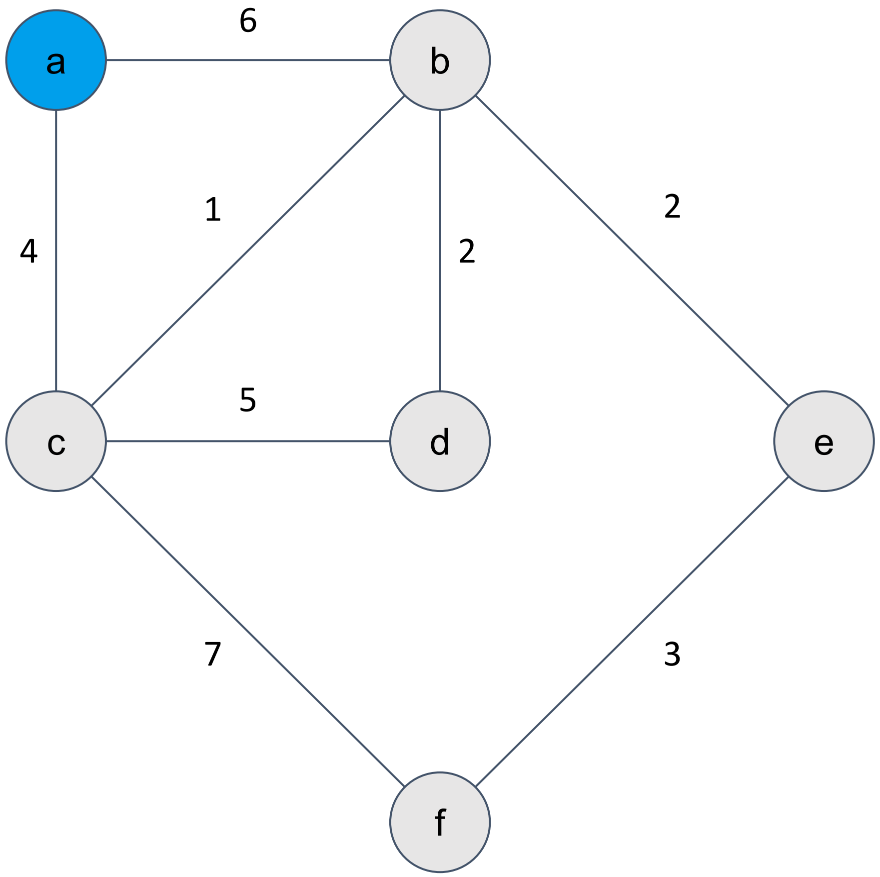
Dijkstra's Algorithm

Vertex	k_v	d_v	p_v
a			
b			
c			
d			
e			
f			



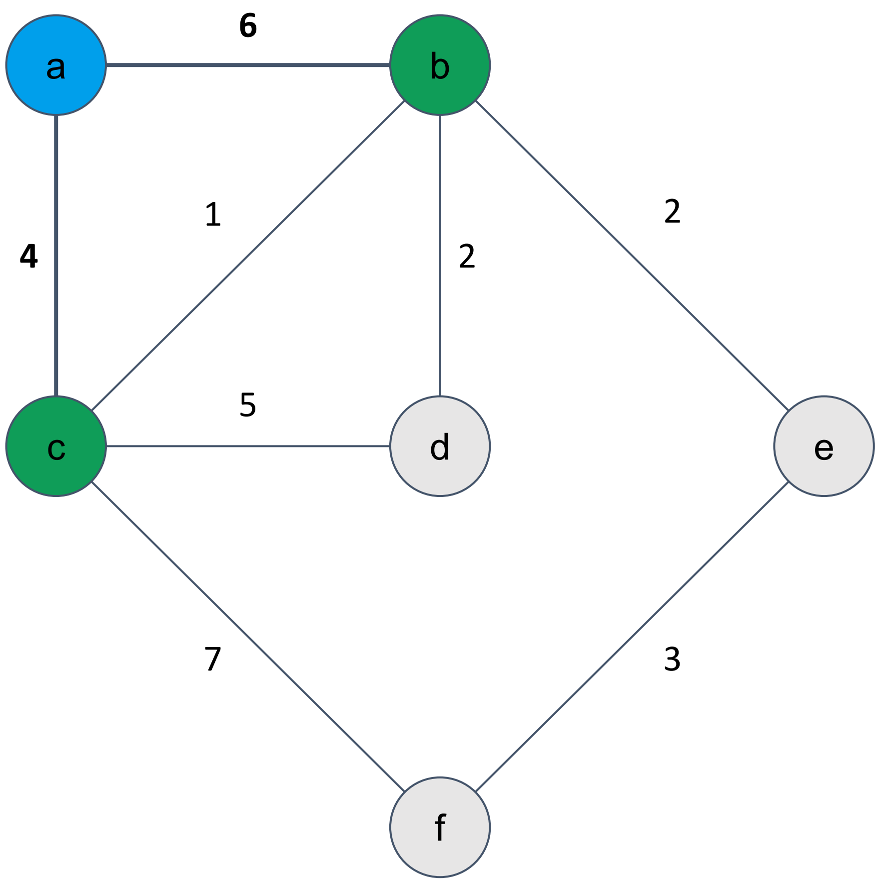
Dijkstra’s Algorithm

Vertex	k_v	d_v	p_v
a	F	0	
b	F	∞	
c	F	∞	
d	F	∞	
e	F	∞	
f	F	∞	



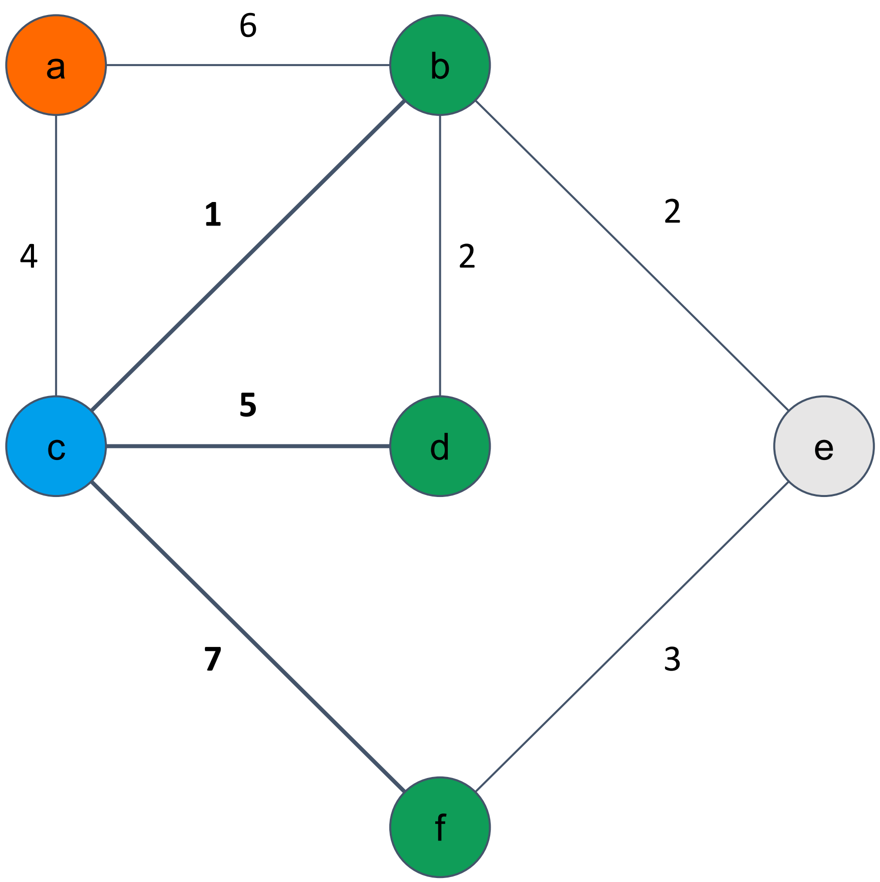
Dijkstra's Algorithm

Vertex	k_v	d_v	p_v
a	**T**	0	---
b	F	**6**	**a**
c	F	**4**	**a**
d	F	∞	
e	F	∞	
f	F	∞	



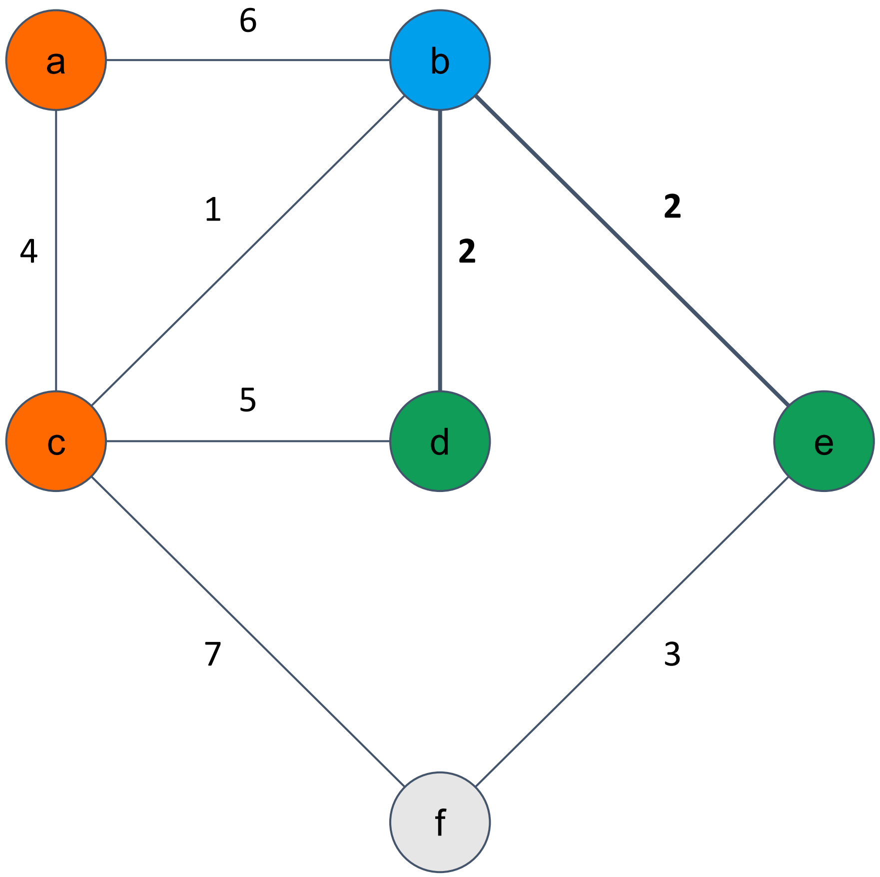
Dijkstra’s Algorithm

Vertex	k_v	d_v	p_v
a	T	0	---
b	F	**5**	**c**
c	**T**	4	a
d	F	**9**	**c**
e	F	∞	
f	F	**11**	**c**



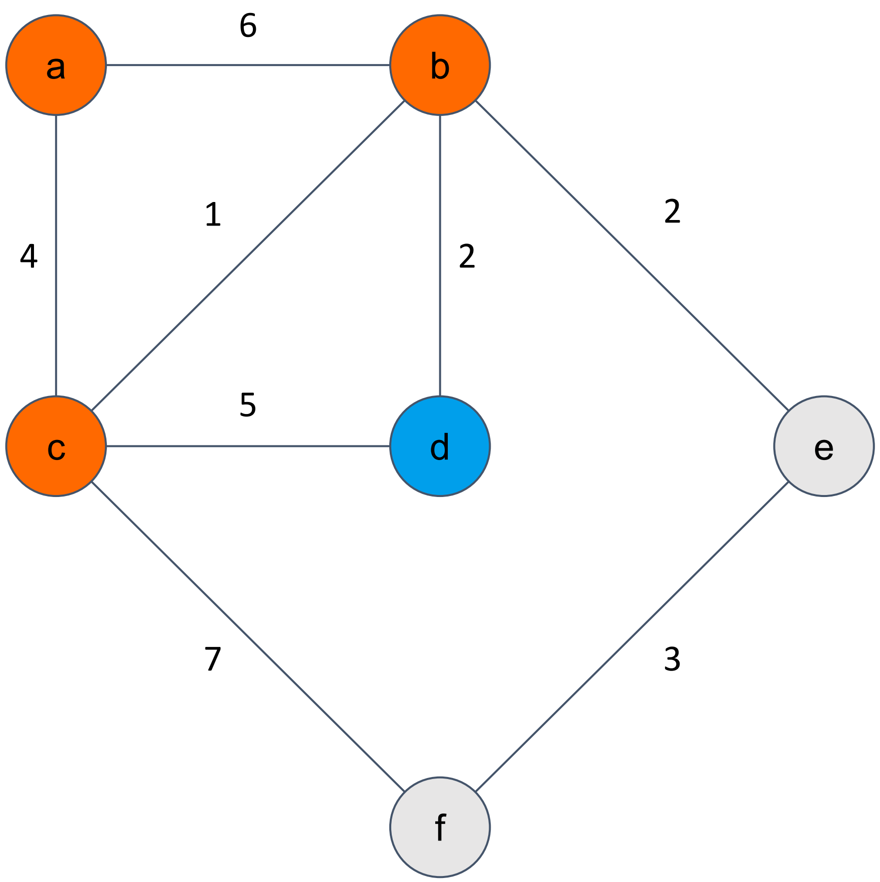
Dijkstra’s Algorithm

Vertex	k_v	d_v	p_v
a	T	0	---
b	**T**	5	c
c	T	4	a
d	F	**7**	**b**
e	F	**7**	**b**
f	F	11	c



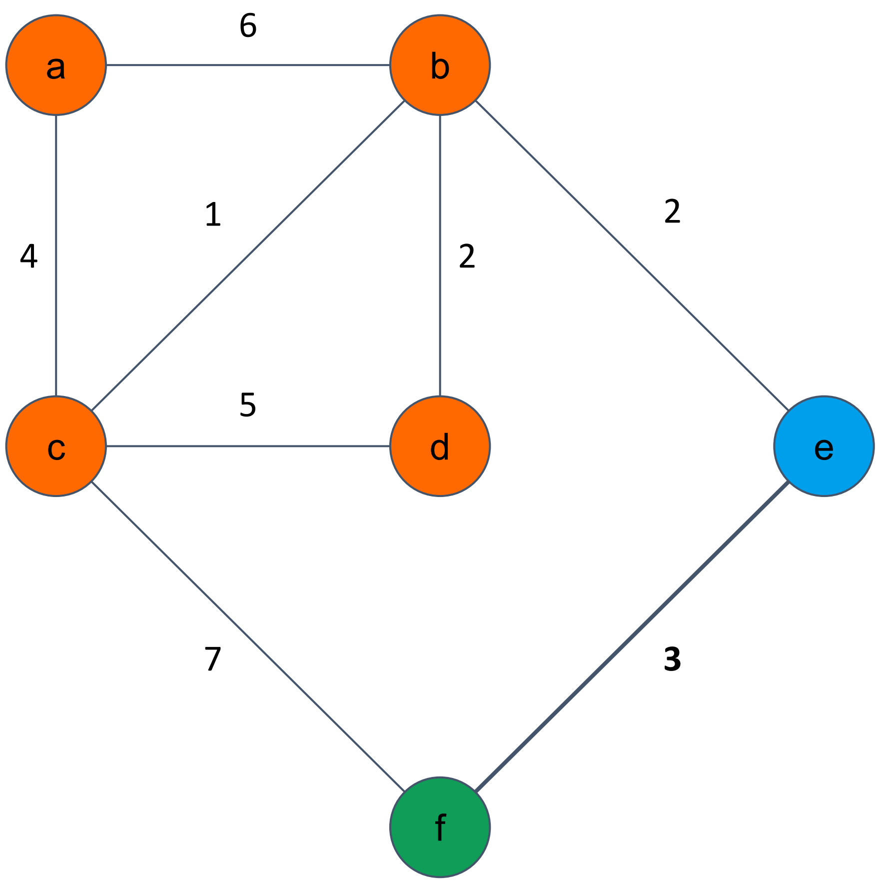
Dijkstra's Algorithm

Vertex	k_v	d_v	p_v
a	T	0	---
b	T	5	c
c	T	4	a
d	**T**	7	b
e	F	7	b
f	F	11	c



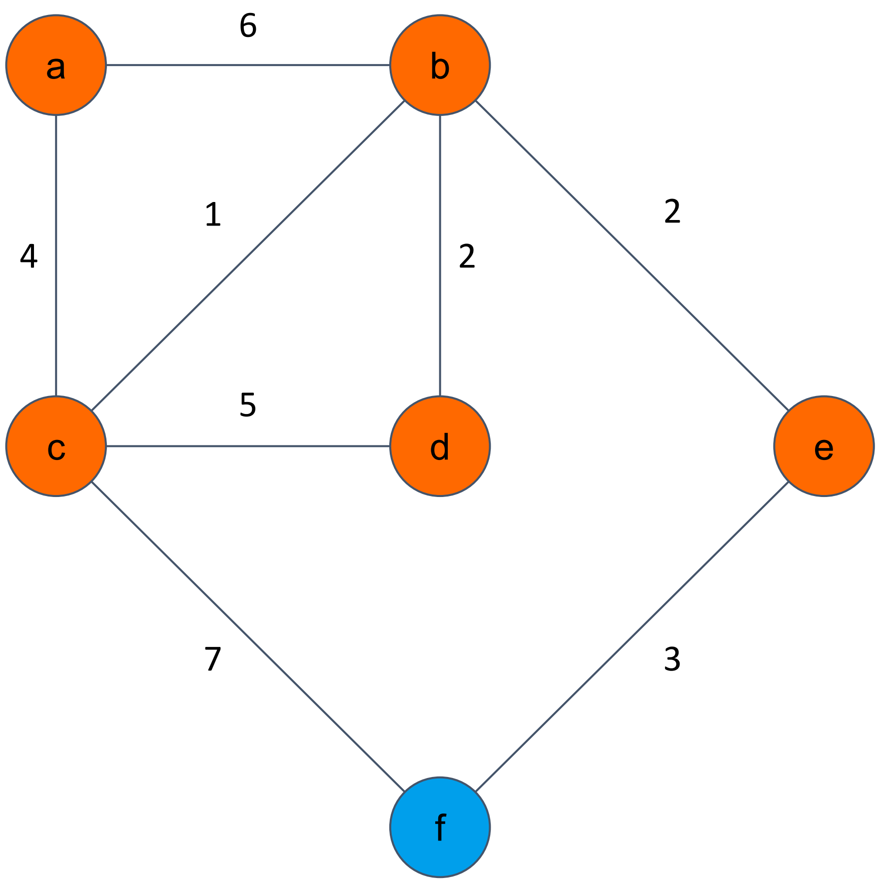
Dijkstra's Algorithm

Vertex	k_v	d_v	p_v
a	T	0	---
b	T	5	c
c	T	4	a
d	T	7	b
e	**T**	7	b
f	F	**10**	**e**



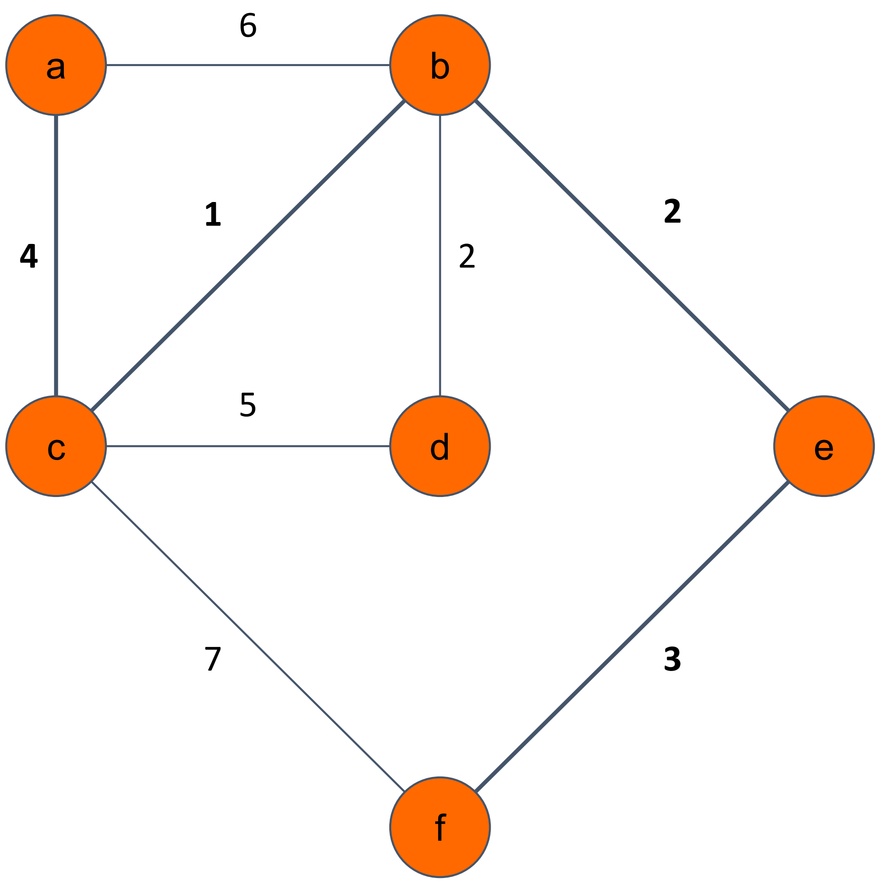
Dijkstra's Algorithm

Vertex	k_v	d_v	p_v
a	T	0	---
b	T	5	c
c	T	4	a
d	T	7	b
e	T	7	b
f	**T**	10	e



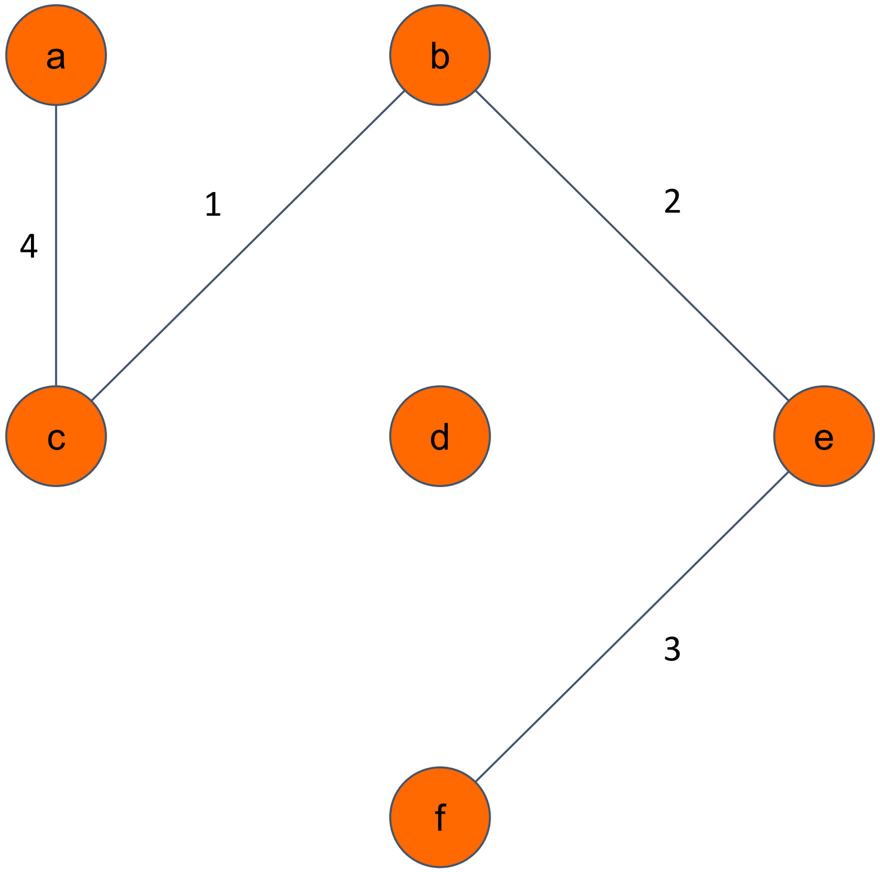
Dijkstra's Algorithm: Find Shortest Path from a to f

Vertex	k_v	d_v	p_v
a	T	0	---
b	T	5	c
c	T	4	a
d	T	7	b
e	T	7	b
f	T	10	e



Dijkstra's Algorithm: Find Shortest Path from a to f

Vertex	k_v	d_v	p_v
a	T	0	---
b	T	5	c
c	T	4	a
d	T	7	b
e	T	7	b
f	T	10	e



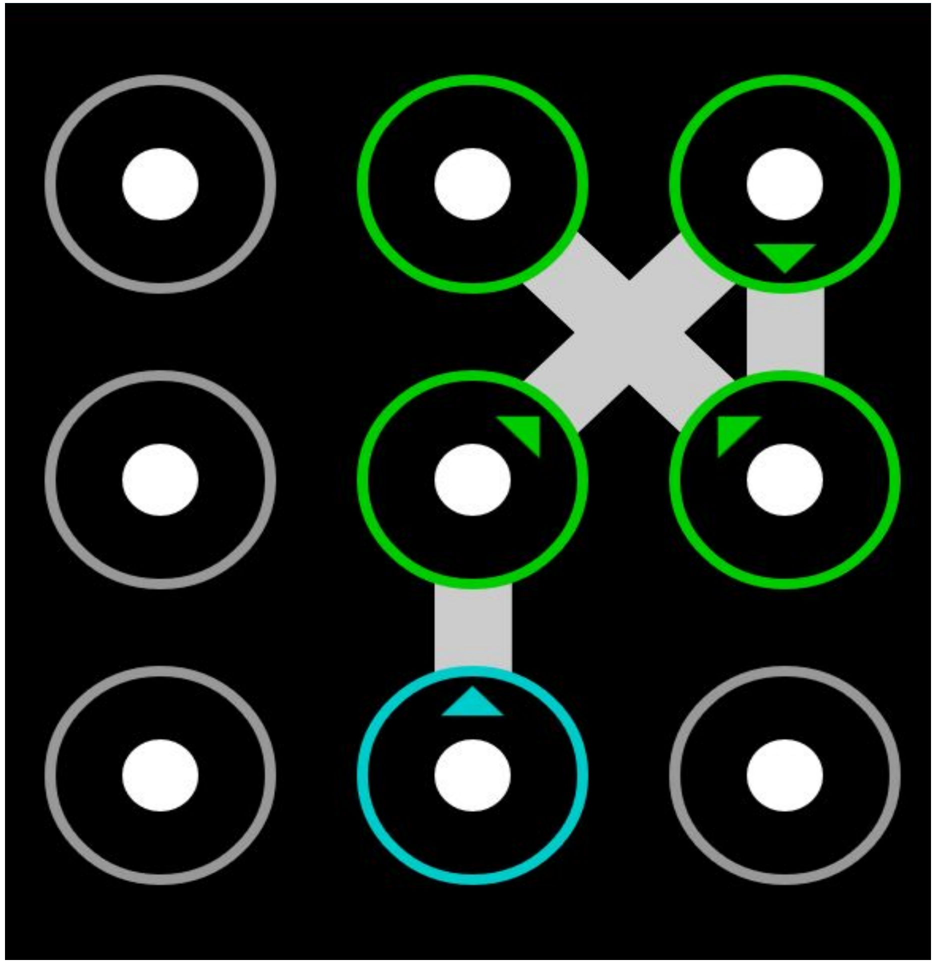
Generating Permutations

Generating Permutations

Problem: Forgot phone password.

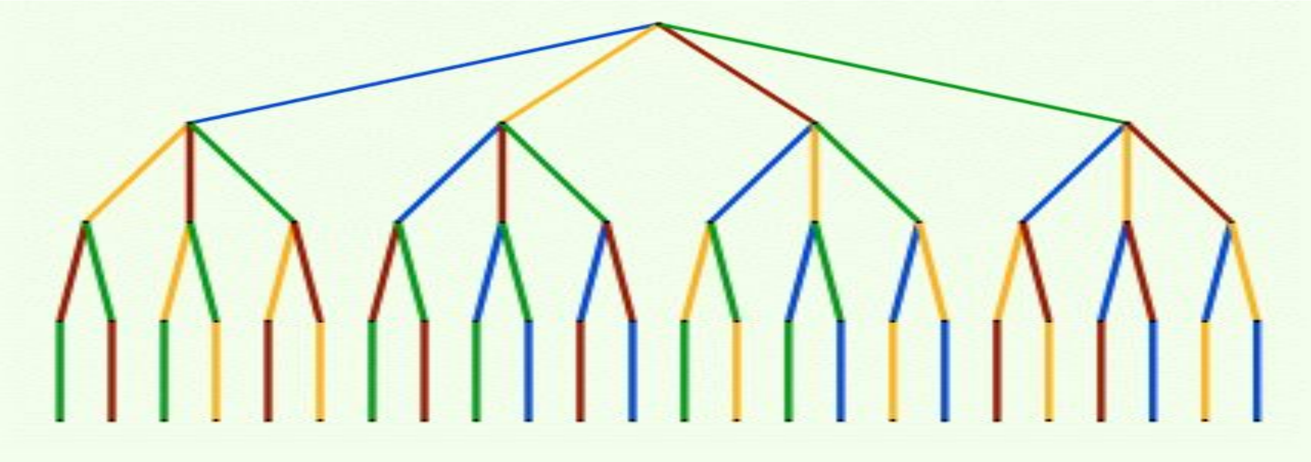
Solution: Try all n! combinations.

n! is too much to store, we need to enumerate without storing all combinations at once.



Solution Spaces

Think of permutations as being a part of a tree. For example, the following tree illustrates all color combinations out of possible four colors:



We enumerate all permutations by doing a **depth first search** of this tree and guarantee we won't repeat ourselves

Permutations: Why Depth-First?

Why not breadth-first search?

Depth-First (Stack):

Maximum size of stack?

Breadth First (Queue)

Maximum size of queue?

Permutations: Why Depth-First?

Why not breadth-first search?

Depth-First (Stack):

Maximum size of stack?

n

Breadth First (Queue)

Maximum size of queue?

n!

Generating Permutations

```
void gen_perms(vector<int>& items, int nfixed) {
  if (nfixed == items.size()) {
    // Base case, entire permutation is fixed. Process items.
    return;
  }
  // Permute values in [nfixed, items.size()).
  for (int i = nfixed; i < items.size(); ++i) {
    // Fix items[nfixed] as items[i].
    swap(items[nfixed], items[i]);
    // Permute values in [nfixed + 1, items.size()).
    gen_perms(items, nfixed + 1);
    // Revert.
    swap(items[nfixed], items[i]);
  }
}
```

Generating Permutations

gen_perms is a procedure that will enumerate permutations of an input list.

[1, 2, 3, 4] has permutations:

1234	2134	3214	4231
1243	2143	3241	4213
1324	2314	3124	4321
1342	2341	3142	4312
1432	2431	3412	4132
1423	2413	3421	4123

Generating Permutations

gen_perms is a procedure that will enumerate permutations of an input list.
There are 7! or 5040 permutations of [A, B, C, D, E, F, G].
Which of them **start with ACE**?

ACEBFDG	ACEFDBG	ACEDGBF
ACEBFGD	ACEFDGB	ACEDGFB
ACEBDFG	ACEFGDB	ACEGFDB
ACEBDGF	ACEFGBD	ACEGFBD
ACEBGDF	ACEDFBG	ACEGDFB
ACEBGFD	ACEDFGB	ACEGDBF
ACEFBDG	ACEDBFG	ACEGBDF
ACEFBGD	ACEDBGF	ACEGBFD

Generating Permutations

gen_perms is a procedure that will enumerate permutations of an input list.
There are 7! or 5040 permutations of [A, B, C, D, E, F, G].
Which of them **start with ACE**? **24**

- There are 4! or 24 ways to order the remaining letters (B, D, F, G)!

ACEBFDG	**ACEFDBG**	**ACEDGBF**
ACEBFGD	**ACEFDGB**	**ACEDGFB**
ACEBDFG	**ACEFGDB**	**ACEGFDB**
ACEBDGF	**ACEFGBD**	**ACEGFBD**
ACEBGDF	**ACEDFBG**	**ACEGDFB**
ACEBGFD	**ACEDFGB**	**ACEGDBF**
ACEFBDG	**ACEDBFG**	**ACEGBDF**
ACEFBGD	**ACEDBGF**	**ACEGBFD**

Generating Permutations

gen_perms(items, nfixed)
produces every permutation of items where the first **nfixed** items are fixed.
(gen_perms(ACEBFGD, 3) would give the permutations on the previous slide)

items is passed by reference for efficiency, so gen_perms can operate in-place.
gen_perms modifies **items**, but it **reverts any changes** before it returns.

Generating Permutations

```
void gen_perms(vector<int>& items, int nfixed) {
  if (nfixed == items.size()) {
    // Base case, entire permutation is fixed. Process items.
    return;
  }
  // Permute values in [nfixed, items.size()).
  for (int i = nfixed; i < items.size(); ++i) {
    // Fix items[nfixed] as items[i].
    swap(items[nfixed], items[i]);
    // Permute values in [nfixed + 1, items.size()).
    gen_perms(items, nfixed + 1);
    // Revert.
    swap(items[nfixed], items[i]);
  }
}
```

Algorithm Families

One Table to Rule Them All

Algorithm Family	Notes
Brute Force	Guarantees optimality. Slow.
Greedy	Pick the next best option - not guaranteed to be optimal
Divide and Conquer	Combining solutions to subproblems (e.g. mergesort)
Backtracking	Constraint satisfaction problems. Note - satisfying constraints is not necessarily optimal.
Branch and Bound	Optimization problems
Dynamic Programming	Save answers to overlapping subproblems to optimize time complexity

Branch and Bound

Branch and Bound

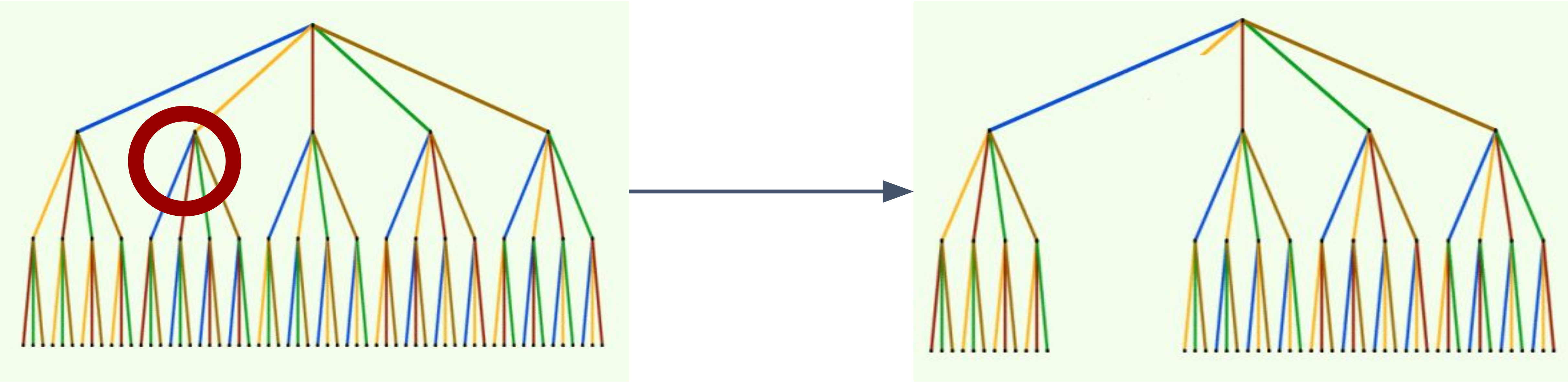
Branch and Bound Pseudocode:

```
checknode (node v, best):
    if !is_promising(v):
        return
    if is_solution(v):
        update(best)
        return
    for each child u of v:
        checknode(u, best)
```

Can we use branch and bound optimization with our gen_perms code?

Pruning

What happens when we prune a branch?



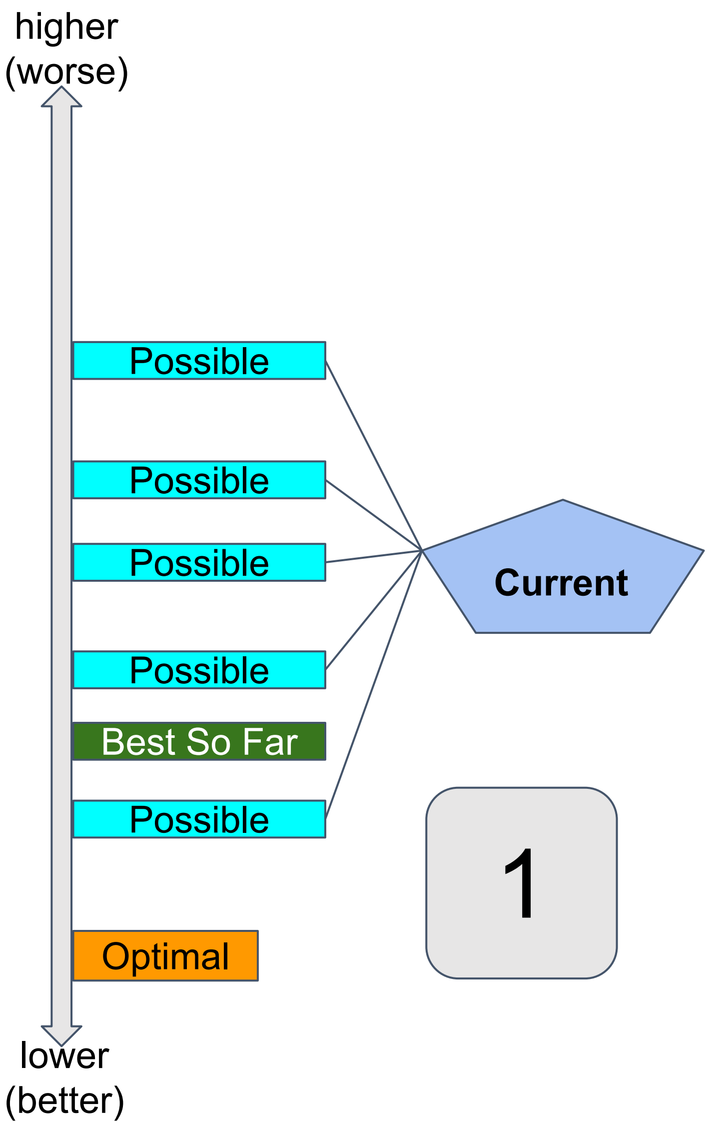
Bounds

With optimization problems, we can determine when to prune by using bounds. A bound is an estimate of the potential of a *solution space*.

Upper bound: Best solution we've found so far.
Lower bound: Optimistic estimate for remaining solutions in this branch.

Lower bound + current is the best case scenario for continuing on this branch. **So, if lower bound + current is less optimal than upper bound, prune this branch.**

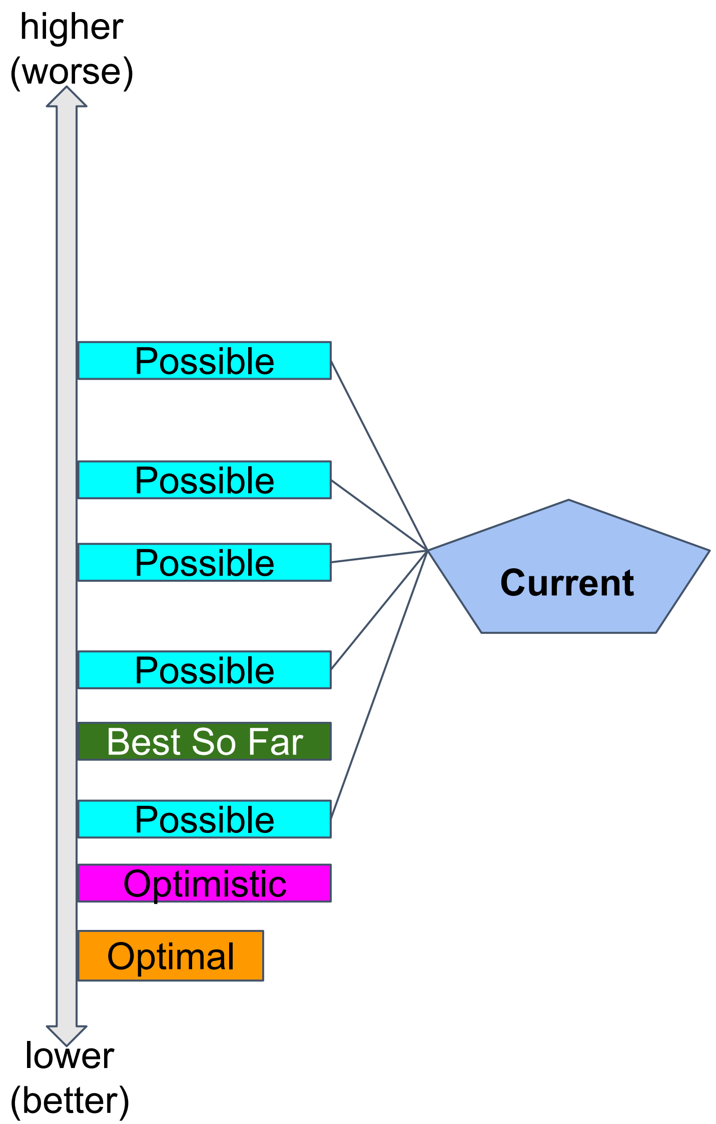
Bounds (Minimization Problems)



The optimal exists, but we don't know what it is.

We do know "best so far."

Bounds (Minimization Problems)



The optimal exists, but we don’t know what it is.

We do know “best so far” (our **upper** bound)

We generate a **lower** bound (aka optimistic):

Must be ***at least as good as*** all possible outcomes.

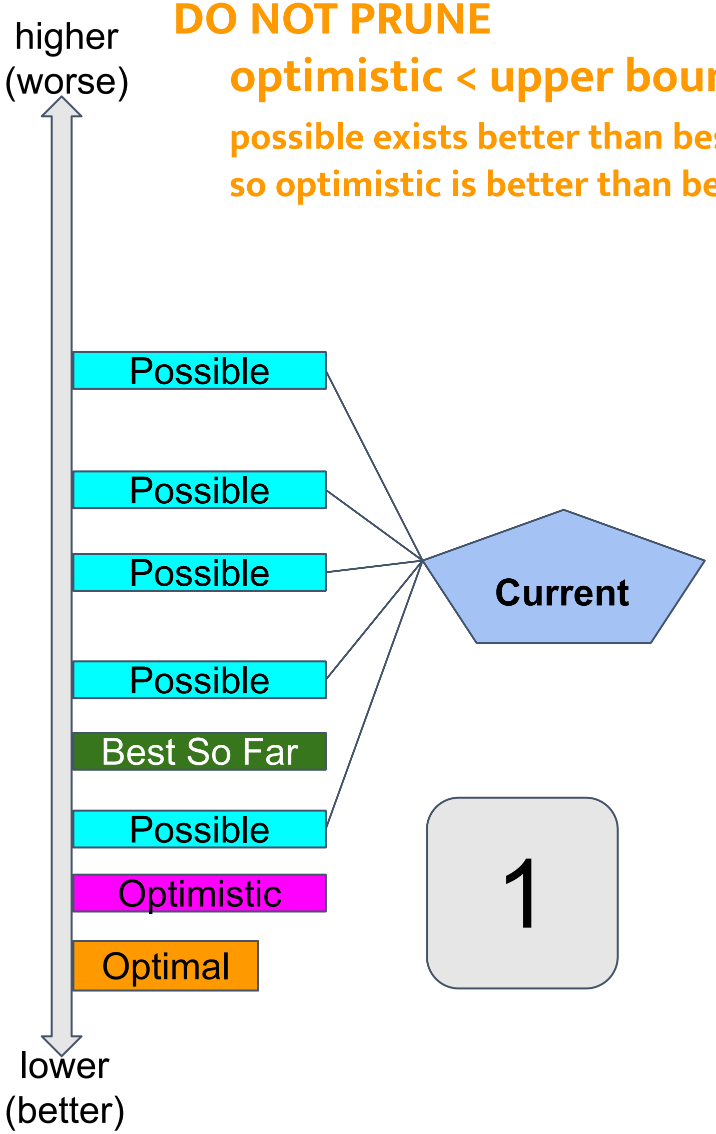
Note that here, “optimistic” = lower-bound + current

Bounds (Minimization Problems)

DO NOT PRUNE

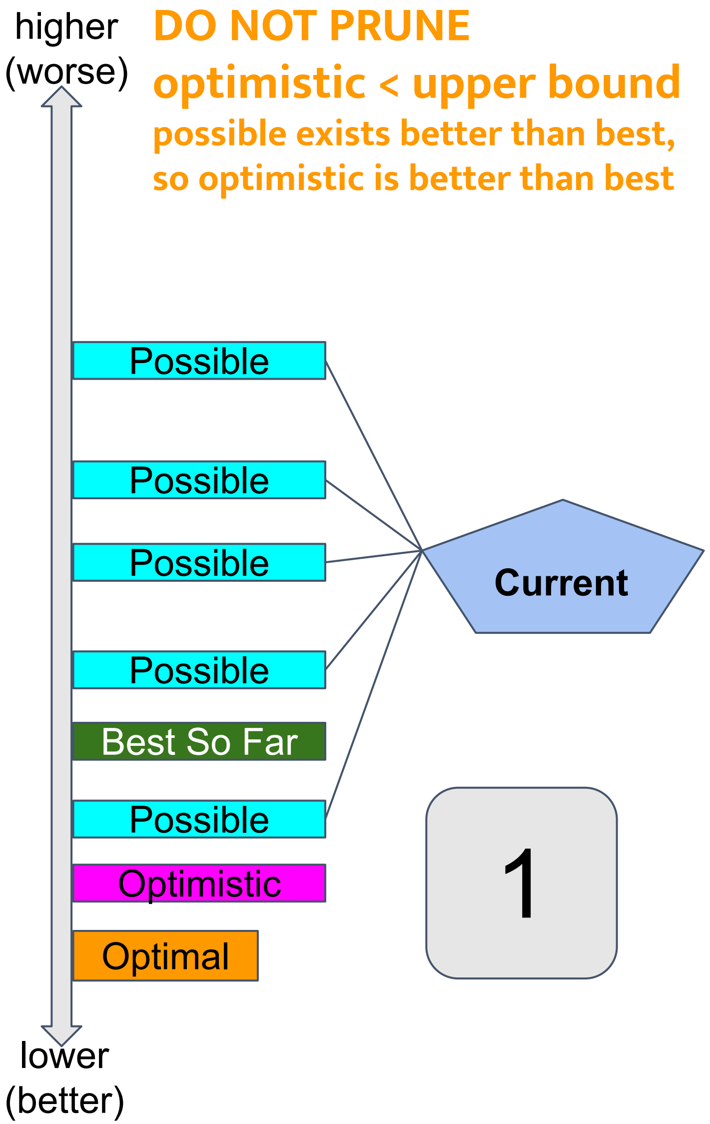
optimistic < upper bound

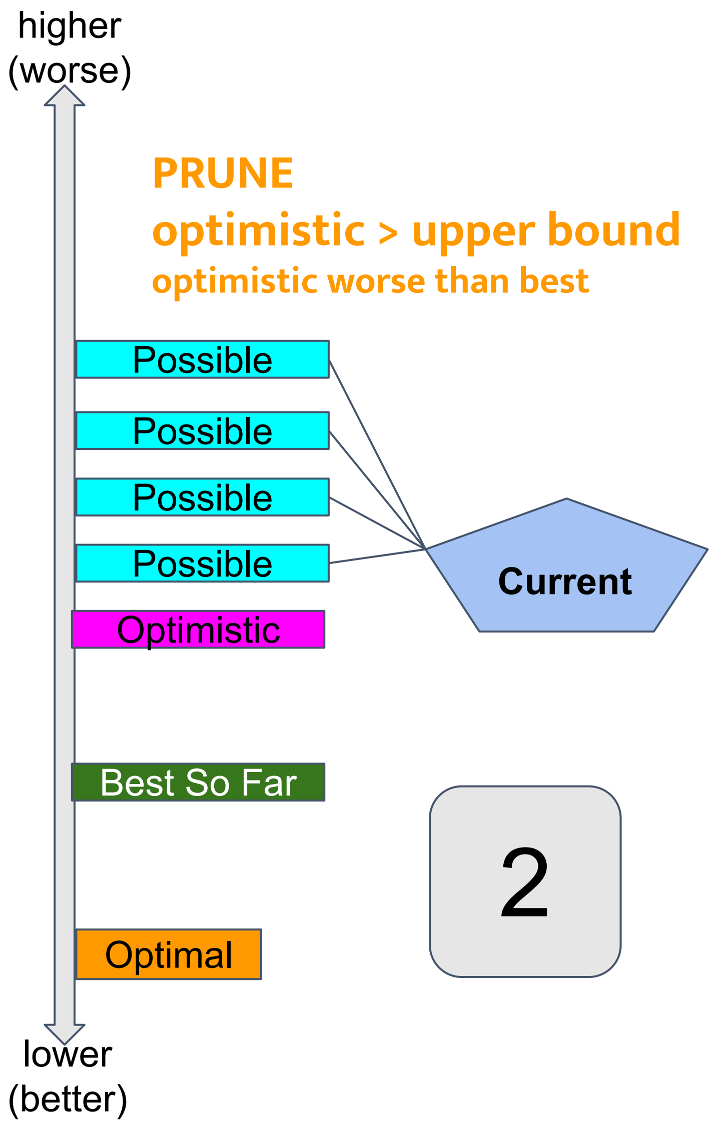
possible exists better than best,
so optimistic is better than best



Note that here, "optimistic" = lower-bound + current

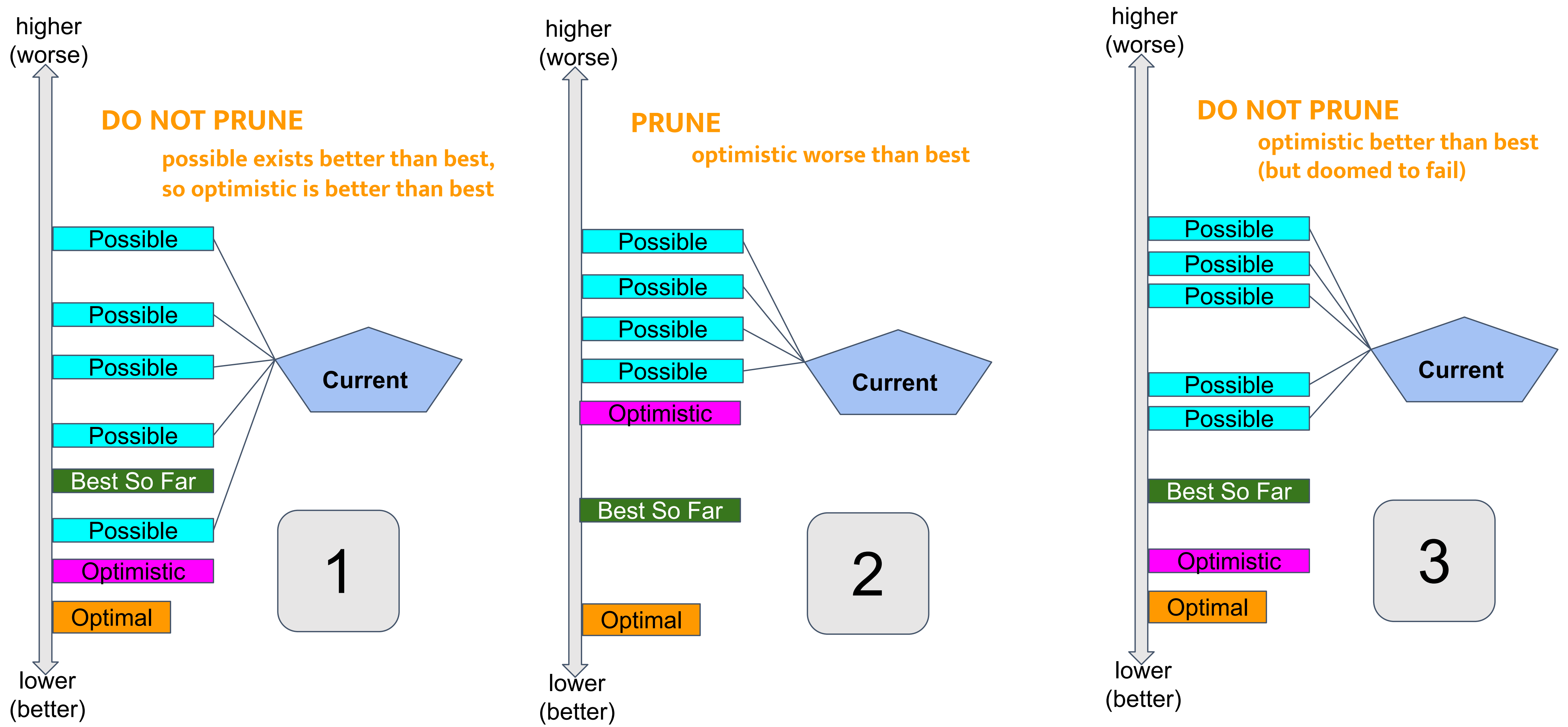
Bounds (Minimization Problems)



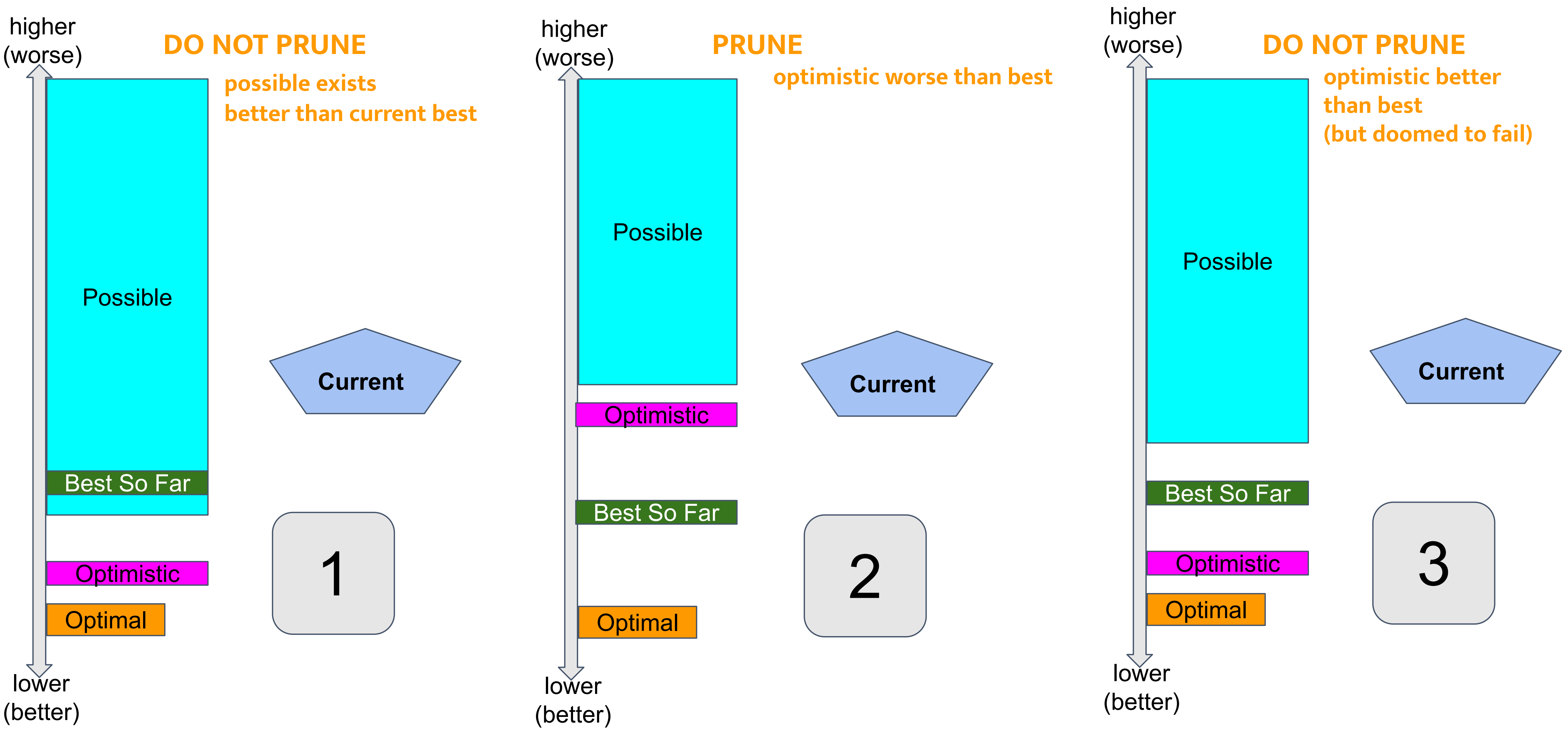


Note that here, “optimistic” = lower-bound + current

Bounds (Minimization Problems)



Bounds (Minimization Problems)



Over-pruning

Invariants:

- upper bound (best solution we've found so far) >= optimal solution
- lower bound (optimistic estimation) <= optimal solution

If this relationship is violated, the bounds are **invalid.**

- If our upper bound is invalid, we will never find a solution.
- If our lower bound is invalid, we may accidently prune the true optimal solution and our answer will be wrong.

[TODO spend some time thinking this one over]

Bounds Tuning

Tradeoff between tightness (accuracy) and performance

- Tighter bounds
  - Allow for more pruning
  - May take up more time
- If the cost of computing our bounds uses more time than saved from pruning, **the bounds are useless.** Experiment to find out which bounds are most effective.

Generating and Pruning Permutations

```
void gen_perms(vector<int>& items, int nfixed) {
  if (nfixed == items.size()) {
    // Base case, entire permutation is fixed. Process items.
    return;
  }
  if (!is_promising(items, nfixed)) {
    return;
  }
  // Permute values in [nfixed, items.size()).
  for (int i = nfixed; i < items.size(); ++i) {
    // Fix items[nfixed] as items[i].
    swap(items[nfixed], items[i]);
    // Permute values in [nfixed + 1, items.size()).
    gen_perms(items, nfixed + 1);
    // Revert.
    swap(items[nfixed], items[i]);
  }
}
```

Dynamic Programming

Dynamic Programming

- ***Remember the answer to every subproblem*** for overlapping subproblems
  - Avoids duplicated calculation of a subproblem by using extra memory
  - e.g. If I call f() 1000 times, but it always gives the same answer, I only need to run it once.

- Dynamic programming trades time for memory (i.e. improves time at the cost of more memory usage).

Naive Fibonacci

```
int fib(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fib(n - 1) + fib(n - 2)
}
```

Running time?

~$O(1.6^n)$. Exponential. Sad!

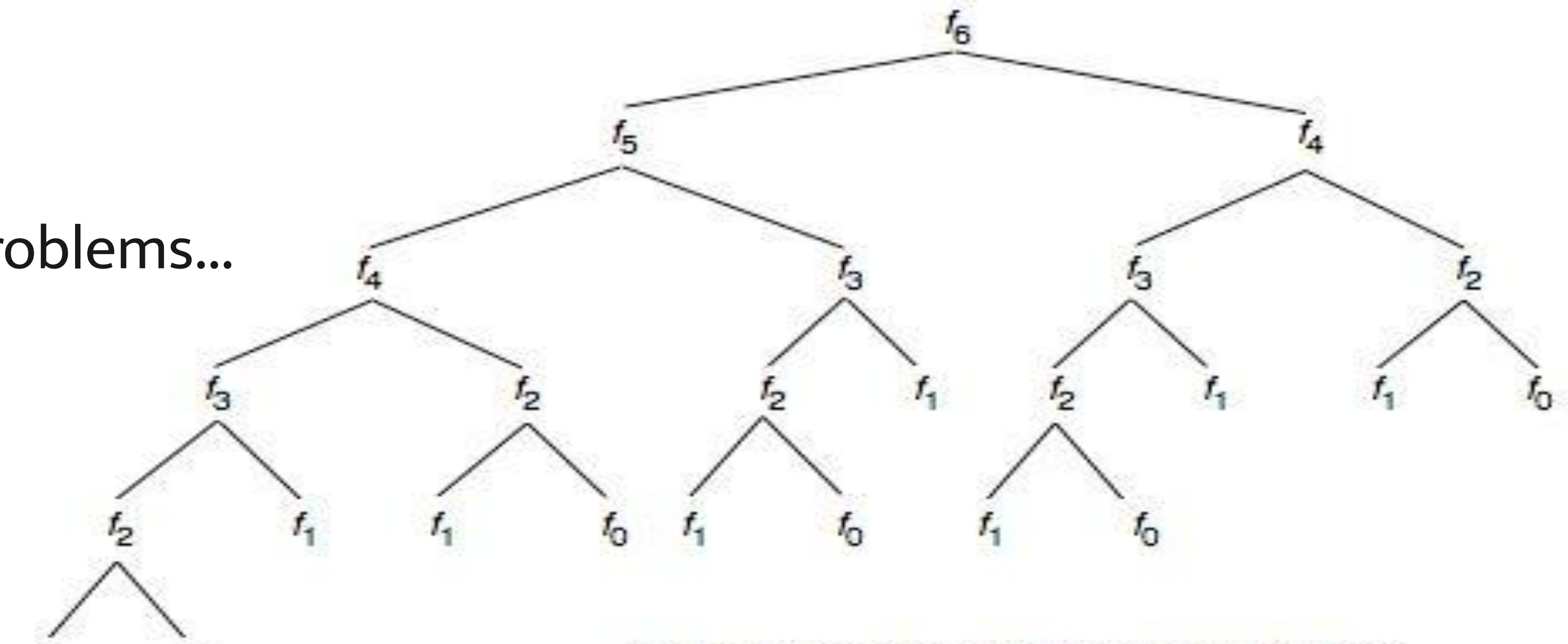
Naive Fibonacci

```
int fib(int n) {
   if (n == 0) return 0;
   if (n == 1) return 1;
   return fib(n - 1) + fib(n - 2)
}
```

Running time?

~O(1.6^n). Exponential. Sad!

But lots of overlapping subproblems...



Recursive call tree for Fibonacci number f_6

Dynamic Programming: Bottom Up

Iterative Approach

1. Start with the base case
2. Build up array/map from base case
3. Stop when you get to the value you are trying to compute

Bottom Up Fibonacci

Compute **fib(i)** at most once.
We do this by storing each result in an array.

```
int fib(int n) {
    static vector<int> fibs(n + 1);
    fibs[0] = 0;
    fibs[1] = 1;
    for (int i = 2; i <=n; i++)
        fibs[i] = fibs[i-1] + fibs[i-2]
    return fibs[n];
}
```

Runtime: O(n)

Dynamic Programming: Top Down

Recursive Approach

1. Check if we already have value needed
2. If true, return it
3. Else calculate, store result, then return result

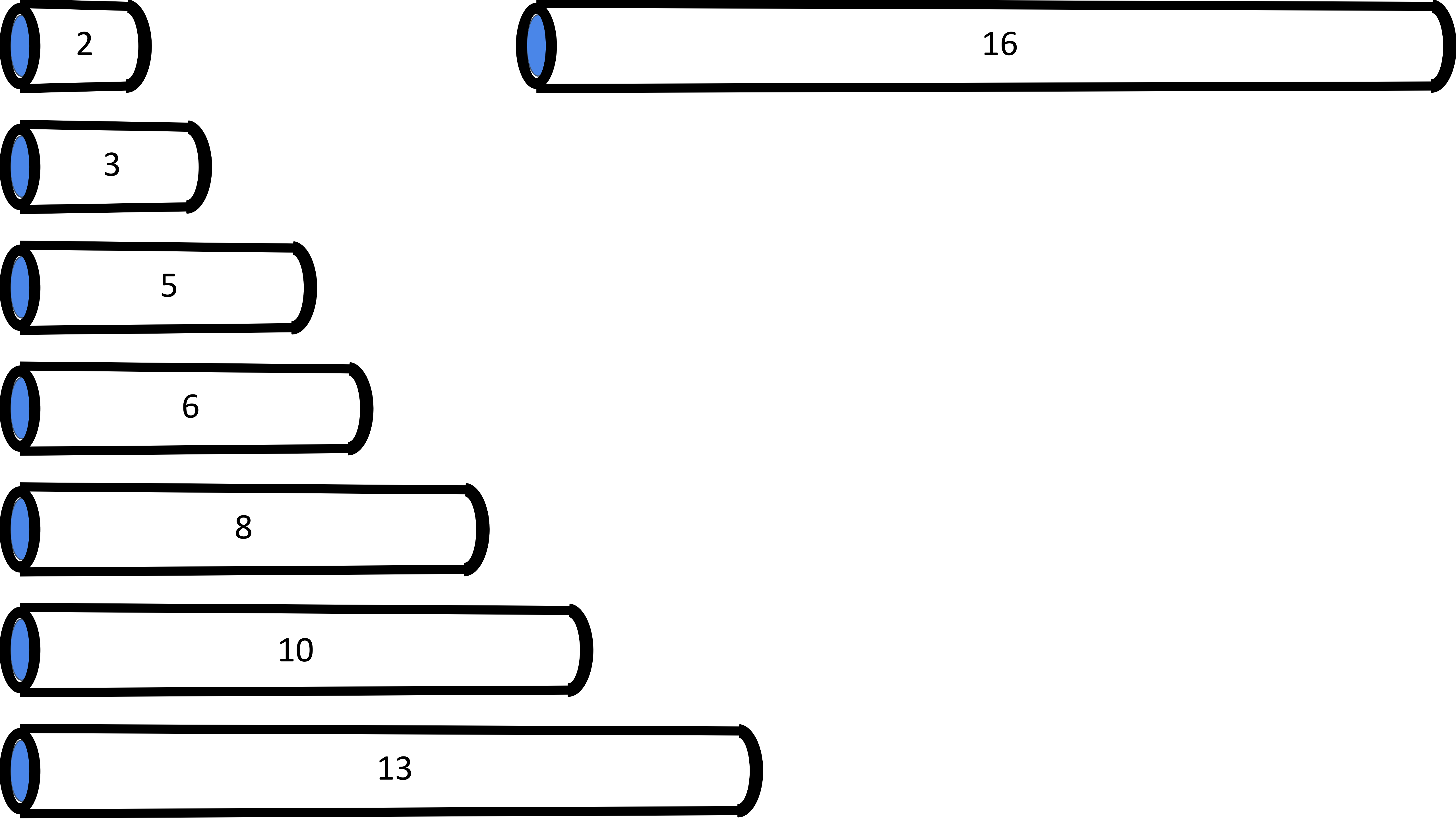
Top Down Fibonacci

Compute **fib(i)** at most once.
We do this by storing each result in an array.

```
static vector<int> remembered;
int fib(int n) {
   if (n >= remembered.size())
      remembered.resize(n+1, -1);
   if (remembered[n] != -1)   // check if we have the value stored
      return remembered[n];
   int result = fib(n-1) + fib(n-2);   // calculate new result
   remembered[n] = result;             // remember result
   return result;
}
```

Runtime: O(n)

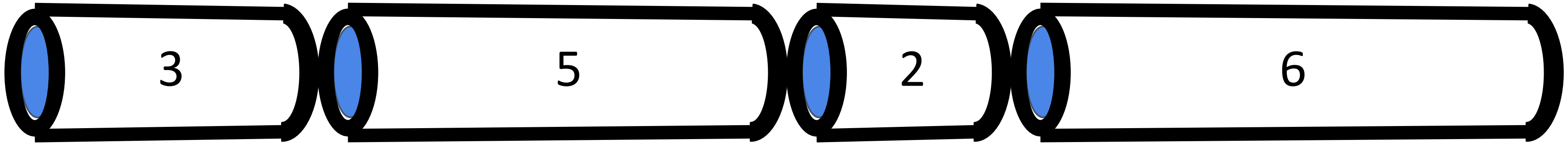
DP: Pipe Welding


2
16
3
5
6
8
10
13

Pipe Welding - Problem Statement

You are trying to weld together a series of pipes of weights $\mathtt{w_1, \ w_2, \ldots, \ w_n}$. The cost to weld together two pipes of weights $\mathtt{w_i}$ and $\mathtt{w_j}$ is $\mathtt{max(w_i, w_j)}$, and welding these pipes creates a new pipe of weight $\mathtt{w_i + w_j}$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided (e.g. pipes 1 and 2 can be welded together, but 1 and 3 cannot be)?

Example: (3, 5, 2, 6)

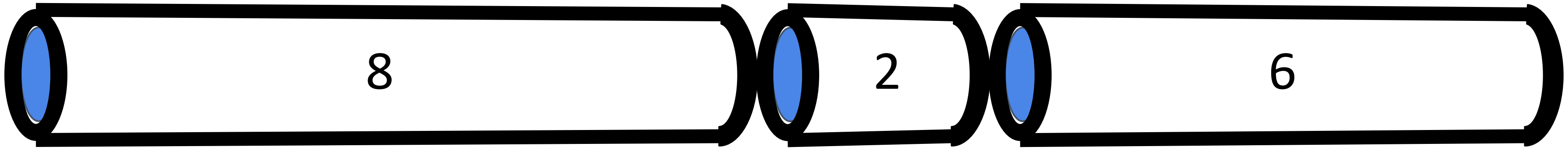


Possible solution: First weld 3 and 5 together.
Running cost: max(3, 5) = 5

Pipe Welding - Example

You are trying to weld together a series of pipes of weights $\mathbf{w_1}, \mathbf{w_2}, \ldots, \mathbf{w_n}$. The cost to weld together two pipes of weights $\mathbf{w_i}$ and $\mathbf{w_j}$ is $\mathbf{max(w_i, w_j)}$, and welding these pipes creates a new pipe of weight $\mathbf{w_i + w_j}$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided (e.g. pipes 1 and 2 can be welded together, but 1 and 3 cannot be)?

Example: (3, 5, 2, 6)

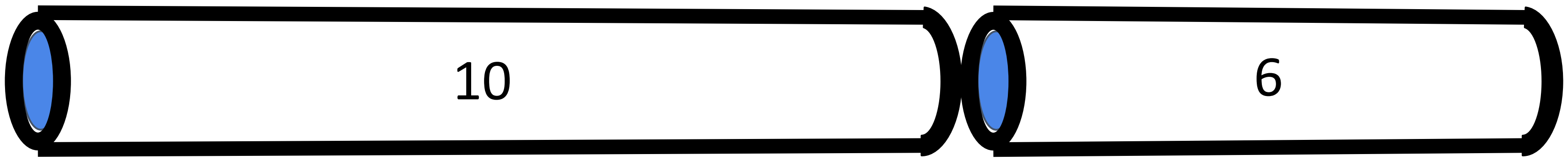


Next weld 8 and 2 together.
Running cost: 5 + max(8, 2) = 13

Pipe Welding - Example

You are trying to weld together a series of pipes of weights $\mathbf{w_1}, \mathbf{w_2}, \dots, \mathbf{w_n}$. The cost to weld together two pipes of weights $\mathbf{w_i}$ and $\mathbf{w_j}$ is $\mathbf{max}(\mathbf{w_i}, \mathbf{w_j})$, and welding these pipes creates a new pipe of weight $\mathbf{w_i} + \mathbf{w_j}$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided (e.g. pipes 1 and 2 can be welded together, but 1 and 3 cannot be)?

Example: (3, 5, 2, 6)

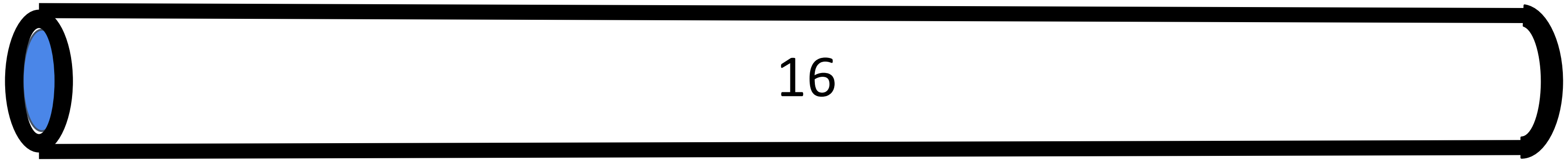


Lastly weld 10 and 6 together.
Running cost: 13 + max(10, 6) = 23

Pipe Welding - Example

You are trying to weld together a series of pipes of weights $\mathtt{w_1}, \mathtt{w_2}, \ldots, \mathtt{w_n}$. The cost to weld together two pipes of weights $\mathtt{w_i}$ and $\mathtt{w_j}$ is $\mathtt{max(w_i, w_j)}$, and welding these pipes creates a new pipe of weight $\mathtt{w_i + w_j}$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided (e.g. pipes 1 and 2 can be welded together, but 1 and 3 cannot be)?

Example: (3, 5, 2, 6)



Total cost: 23

Can we do better?

Pipe Welding - Determining Solution

You are trying to weld together a series of pipes of weights $\mathbf{w_1, w_2, \ldots, w_n}$. The cost to weld together two pipes of weights $\mathbf{w_i}$ and $\mathbf{w_j}$ is $\mathbf{max(w_i, w_j)}$, and welding these pipes creates a new pipe of weight $\mathbf{w_i + w_j}$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided?

Bad solution: Try all possibilities

- Exponential runtime!

Better: Use dynamic programming

Steps:

1. Determine recurrence
2. Determine memo

Pipe Welding - Base Case

You are trying to weld together a series of pipes of weights $w_1, w_2, \ldots, w_n$. The cost to weld together two pipes of weights w_i and w_j is $\max(w_i, w_j)$, and welding these pipes creates a new pipe of weight $w_i + w_j$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided?

Base cases:

- Use base case of welding a pipe **with itself**, rather than 2 pipes - it makes it much simpler!
- Since welding with itself does not actually require welding, the cost is 0 and the weight is the weight of that pipe

Pipe Welding - Determining Recurrence

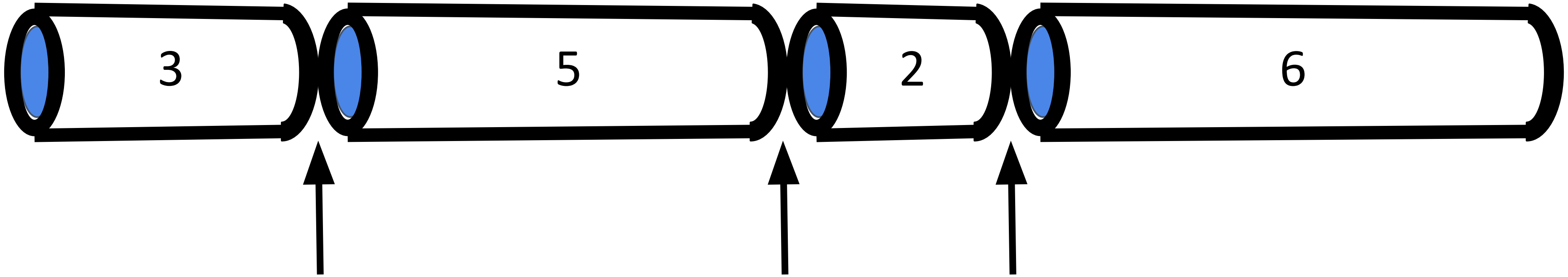
You are trying to weld together a series of pipes of weights $w_1, w_2, \ldots, w_n$. The cost to weld together two pipes of weights w_i and w_j is $\max(w_i, w_j)$, and welding these pipes creates a new pipe of weight $w_i + w_j$. What is the minimum cost to weld together all the pipes, given the constraint that they must be welded in the order that they were provided?

Idea for recurrence:

- Find the location of the **last** weld in an optimal solution
- For each possible location of the last weld, determine the cost of that last weld assuming that these last two pipes have each been welded together in an optimal way.
- Cost is given by the cost of the last weld and the cost of creating the last two pipes

Pipe Welding - Determining Recurrence

- Find the location of the **last** weld in an optimal solution
- For each possible location of the last weld, determine the cost of that last weld assuming that these last two pipes have each been welded together in an optimal way.
- Cost is given by the cost of the last weld and the cost of creating the last two pipes



3 possible locations for last weld

Pipe Welding - Last Weld After 1st Pipe

- Find the location of the **last** weld in an optimal solution
- For each possible location of the last weld, determine the cost of that last weld assuming that these last two pipes have each been welded together in an optimal way.
- Cost is given by the cost of the last weld and the cost of creating the last two pipes

Minimal cost of getting 3: 0

Minimal cost of getting 13: 5 + 7 = 12

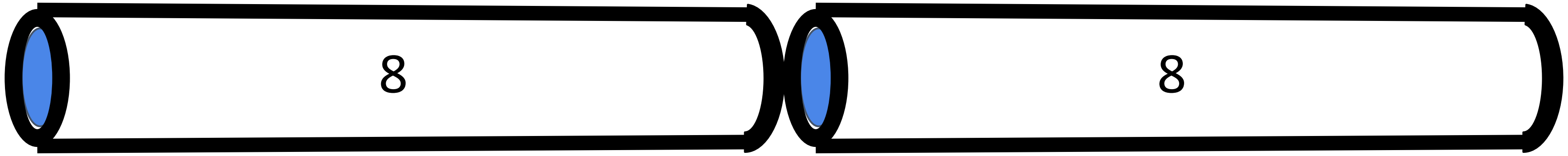
- We leave it as an exercise to verify that

Cost of final weld: max(3, 13) = 13

Total cost: 0 + 12 + 13 = 25

Pipe Welding - Last Weld After 2nd Pipe

- Find the location of the **last** weld in an optimal solution
- For each possible location of the last weld, determine the cost of that last weld assuming that these last two pipes have each been welded together in an optimal way.
- Cost is given by the cost of the last weld and the cost of creating the last two pipes



Minimal cost of getting 8: 5
Minimal cost of getting 8: 6
Cost of final weld: max(8, 8) = 8
Total cost: 5 + 6 + 8 = 19

Pipe Welding - Last Weld After 3rd Pipe

- Find the location of the **last** weld in an optimal solution
- For each possible location of the last weld, determine the cost of that last weld assuming that these last two pipes have each been welded together in an optimal way.
- Cost is given by the cost of the last weld and the cost of creating the last two pipes

Minimal cost of getting 10: 5 + 7 = 12

- We leave it as an exercise to verify that

Minimal cost of getting 6: 0

Cost of final weld: max(10, 6) = 10

Total cost: 12 + 0 + 10 = 22

Pipe Welding - Optimal Solution

Minimal cost: min(25, 19, 22) = 19

Optimal solution:

- Weld 3 and 5 with a cost of 5 to get 8.
- Weld 2 and 6 with a cost of 6 to get 8.
- Weld 8 and 8 with a cost of 8 to get 16.

Total cost is 5 + 6 + 8 = 19.

Pipe Welding - Memoization

- Need memos for both costs of welding and weights of new pipes created
- Memos will be 2D vectors where the value at [row][col] gives the minimal cost and weight of combining pipes[row] through pipes[col], both endpoints inclusive
- Base cases are found along the diagonal, where row == col, meaning we don't need to do any welding.
  - Bottom up: Need to fill the memo starting at the diagonal and moving toward the top right corner. Move left to right across columns, bottom to top from diagonal across rows
  - Top down: Start at the top right corner and recurse down to the diagonal
- At each cell [row][col], try all locations of final weld by looking at all [row][i] and [i + 1][col] where row <= i < col
  - The final weld is after the ith pipe
  - [row][i] gives the cost and weight of welding pipes[row] through pipes[i], while [i+1][col] gives the cost and weight of welding pipes[i+1] through pipes[col]

Pipe Welding - Memoization Example

pipes = {3, 5, 2}

Costs memo:

Weights memo:

Pipe Welding - Memoization Example

pipes = {3, 5, 2}
[0][0] - base case
Costs memo:

0		

Weights memo:

3		

Pipe Welding - Memoization Example

pipes = {3, 5, 2}

[1][1] - base case

Costs memo:

0		
	0	

Weights memo:

3		
	5	

Pipe Welding - Memoization Example

pipes = {3, 5, 2}
[0][1] - only possible to weld between pipes[0] and pipes[1]
cost[0][1] = cost(making pipe 0) + cost(making pipe 1) + cost(weld pipe 0 and pipe 1)
= cost[0][0] + cost[1][1] + max(3, 5)

Costs memo:

0	0 + 0 + max(3, 5) = 5	
	0	

Weights memo:

3	3 + 5 = 8	
	5	

Pipe Welding - Memoization Example

pipes = {3, 5, 2}

[2][2] - base case

Costs memo:

0	5	
	0	
		0

Weights memo:

3	8	
	5	
		2

Pipe Welding - Memoization Example

pipes = {3, 5, 2}
[1][2] - only possible to weld between pipes[1] and pipes[2]
cost[1][2] = cost(making pipe 1) + cost(making pipe 2) + cost(weld pipe 1 and pipe 2)
= cost[1][1] + cost[2][2] + max(5, 2)

Costs memo:

0	5	
	0	**0 + 0 + max(5, 2) = 5**
		0

Weights memo:

3	8	
	5	**5 + 2 = 7**
		2

Pipe Welding - Memoization Example

pipes = {3, 5, 2}
[0][2]
Costs memo:

0	5	min(0 + 5 + max(3, 7), 5 + 0 + max(8, 2)) = min(12, 13) = 12
	0	5
		0

Weights memo:

3	8	3 + 7 = 10
	5	7
		2

2 possibilities:

1. Weld 3 and 7
   cost = cost[0][0] + cost[1][2] + max(3, 7)
   = 0 + 5 + 7
   = 12
1. Weld 8 and 2
   cost = cost[0][1] + cost[2][2] + max(8, 2)
   = 5 + 0 + 8
   = 13

Thus the minimum cost is 12.
Weight is the same regardless:
3 + 7 = 2 + 8 = 10

Pipe Welding - Bottom Up Solution

```
int weld(vector<int> &pipes) {
  vector<vector<int>> weights (pipes.size(),vector<int>(pipes.size())); //memo for weight of welded pipes
  vector<vector<int>> costs (pipes.size(),vector<int>(pipes.size()));  // memo for weld cost
  for (int col = 0; col < pipes.size(); ++col){  //iterate through columns left to right
    for (int row = col; row >= 0; --row) { //iterate through rows bottom to top, starting at diagonal
      if (row == col) { // base cases
        costs[row][col] = 0;
        weights[row][col] = pipes[row];
        continue;
      }
      int min_cost = std::numeric_limits<int>::max();
      for (int i = row; i < col; ++i){  // iterate through all possible final weld places
        cost_i = costs[row][i] + costs[i+1][col] + max(weights[row][i], weights[i+1][col]); //from recurrence
        min_cost = min(min_cost, cost_i);
      } // for i
      weights[row][col] = weights[row][row] + weights[row + 1][col]; //weight will be the same regardless of
      how we got there, so arbitrarily choose these two cells
      costs[row][col] = min_cost;
    } // for row
  } // for col
  return costs[0][pipes.size() - 1]; // minimum cost of welding from pipe 0 to the last pipe
} // weld
```

Runtime: $O(n^3)$, Memory $O(n^2)$ where n is the number of pipes

Pipe Welding - Top Down Solution

```
int weld(const vector<int> &pipes, int begin, int end){
  vector<vector<int>> weights (pipes.size(),vector<int>(pipes.size()));    // memo for weight of welded pipes
  vector<vector<int>> costs (pipes.size(),vector<int>(pipes.size(), -1));  // memo for weld cost
  return weld_helper(pipes, costs, weights, 0, pipes.size() - 1);
}
// weld_helper finds cost and weight of welding pipes[begin] through pipes[end], inclusive
int weld_helper(const vector<int> &pipes, const vector<vector<int>> &costs, const vector<vector<int>> &weights,
                int begin, int end){
  if (costs[begin][end] != -1) return costs[begin][end];  // no need to redo work
  if (begin == end){  // base cases
    costs[begin][end] = 0;
    weights[begin][end] = pipes[begin];
    return 0;
  }
  int min_cost = std::numeric_limits<int>::max();
  for (int i = begin; i < end; ++i) {  // iterate through all possible final weld places
    int cost_i = weld_helper(pipes, costs, weights, begin, i) + weld_helper(pipes, costs, weights, i+1, end);
    //cost of creating last 2 pipes
    cost_i += max(weights[begin][i], weights[i+1][end]);  //cost of final weld
    min_cost = min(min_cost, cost_i);
  }
  costs[begin][end] = min_cost;
  weights[begin][end] = weights[begin][begin] + weights[begin + 1][end]; //weight will be the same regardless of
  how we got there, so arbitrarily choose these two cells
  return costs[begin][end];
}
```

Runtime: $O(n^3)$, Memory: $O(n^2)$ where n is the number of pipes

Dynamic Programming - Types of Problems

Type	Definition	Approach	Example
Count Ways	Count the number of ways to achieve a goal	Sum all possible ways to reach the current state.	Knight Moves
Find Min/Max Path	Find the best path to a goal, given a cost for each step.	Choose best (min/max) path of all possible states from which we reach the current state, and add cost of the current state.	Coin Change* Min Cost Climbing Stairs*
Merge	Combine all elements with the optimal cost.	Try all possible combinations of subintervals + cost of merging and return the best one (min/max).	Pipe Welding
Decision	Choose a subset of items to include given some constraints.	If we use the current element, look at previous states where that element was not used. Else, look at previous states where that element was used.	Knapsack
String/List	Typically deals with modifying, adding, or removing characters from strings, or elements from lists.	Two common approaches: - One index per string/list: used when multiple strings/lists given as input. - Two indices on one string/list: used when one string/list given as input.	Longest Increasing Subsequence* Edit Distance*

*Not covered in lecture or lab slides, but it will either come up in practice problems or it is a common DP problem that can be googled

Handwritten Problem

Handwritten Problem

To practice DP, we would like you to try to come up with the DP solution to the 0-1 Knapsack Problem, as described in lecture. Don't just copy it from the slides, try to build it up yourself!

You have a bag of weight capacity (**`cap`**) and a selection of N items, which each have a value and a weight. Values and weights are passed in as vectors, where the i^{th} item has value of **`value[i]`** and weight of **`weight[i]`**.

Choose which items to put in your bag so that you maximize the combined value of all the items in your bag without going over the weight capacity. Return this maximum value.

```
int knapsack(int cap, vector<int> value, vector<int> weight);
```

DP: Positive Subset Sum

Positive Subset Sum

You're given an array of numbers, like the following:

{6, 9, 10, 15, 36, 44, 68}

Does some subset of this list sum up to exactly **104**?

This problem will not be covered this semester due to time, but you can watch previous semester videos for a walkthrough of this problem!

- **Spring 2020: https://youtu.be/RqZ1X9b39oQ?t=2635**
- **Winter 2020: https://youtu.be/IwpEKUmNTog?t=2381**

Positive Subset Sum

You're given an array of numbers, like the following:

{6, 9, 10, 15, 36, 44, 68}

Does some subset of this list sum up to exactly **104**?

Positive Subset Sum

You're given an array of numbers, like the following:

{6, 9, 10, 15, 36, 44, 68}

Does some subset of this list sum up to exactly **104**?

Yes: 9+15+36+44 = 104

Positive Subset Sum

You're given an array of numbers, like the following:

{6, 9, 10, 15, 36, 44, 68}

Does some subset of this list sum up to exactly **104**?

Yes: 9+15+36+44 = 104

Does some subset of this list sum up to exactly **106**?

Positive Subset Sum

You're given an array of numbers, like the following:

{6, 9, 10, 15, 36, 44, 68}

Does some subset of this list sum up to exactly **104**?

Yes: 9+15+36+44 = 104

Does some subset of this list sum up to exactly **106**?

No

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

How can we solve this?

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

How can we solve this?

- Generate all subsets, add them up, check if it matches
  - Time: $O(n2^n)$

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

How can we solve this?

- ~~Generate all subsets, add them up, check if it matches~~
- ???

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

How can we solve this?

- ~~Generate all subsets, add them up, check if it matches~~
- Use dynamic programming
  - If the array contains only small integers, then we can do this using DP!

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

Let *f(k, x)* be *true* if *some* subset of $\{a_0, a_1, ..., a_{k-1}\}$ adds up to *x* (where $k \leq n$)

"Can we make x with just the first *k* of the numbers?"

Suppose we can. Then there are two cases:

- we can do it without using a_{k-1}
- we can do it by using a_{k-1}
- (also both)

Positive Subset Sum

Let $f(k, x)$ be *true* if any subset of $\{a_0, a_1, ..., a_{k-1}\}$ adds up to x (where $k \leq n$)
"Can we do it with just the first k of the numbers?"

Suppose we can. Then there are two cases:

- we can do it without using a_{k-1} — *Then the same set is a subset of $\{a_0, ..., a_{k-2}\}$ adding to x*
- we can do it by using a_{k-1} — *Then if we remove a_{k-1}, it adds up to $x - a_{k-1}$*

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

The Subproblem

Let *f(k, x)* be *true* if a subset of $\{a_0, a_1, ..., a_{k-1}\}$ adds up to *x* (where $k \leq n$)

$f(0, 0)$	=	true	*empty has sum 0*
$f(0, x \neq 0)$	=	false	*empty can't sum to anything else*
$f(k, x)$	=	???	

Positive Subset Sum

You're given an array of numbers, like the following:

$\{a_0, a_1, a_2, ..., a_{n-1}\}$

Does some subset of this list sum up to exactly X?

The Subproblem

Let *f(k, x)* be *true* if a subset of $\{a_0, a_1, ..., a_{k-1}\}$ adds up to *x* (where $k \leq n$)

$f(0, 0)$	=	true
$f(0, x \neq 0)$	=	false
$f(k, x)$	=	$f(k-1, x - a_{k-1})$ *(use a_{k-1})* **or** $f(k-1, x)$ *(don't use a_{k-1})*

Positive Subset Sum Example

Given the array: [1,2,5,3]

SubsetSum(9, 4) == can we make 9 with first 4 elements?

Positive Subset Sum Example

Given the array: [1,2,5,3]

SubsetSum(9, 4) == can we make 9 with first 4 elements?

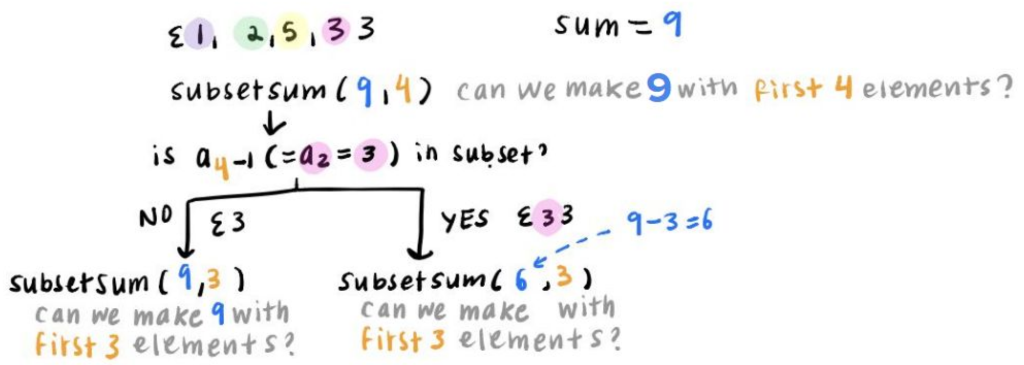
Yes

Positive Subset Sum Example (Top Down)

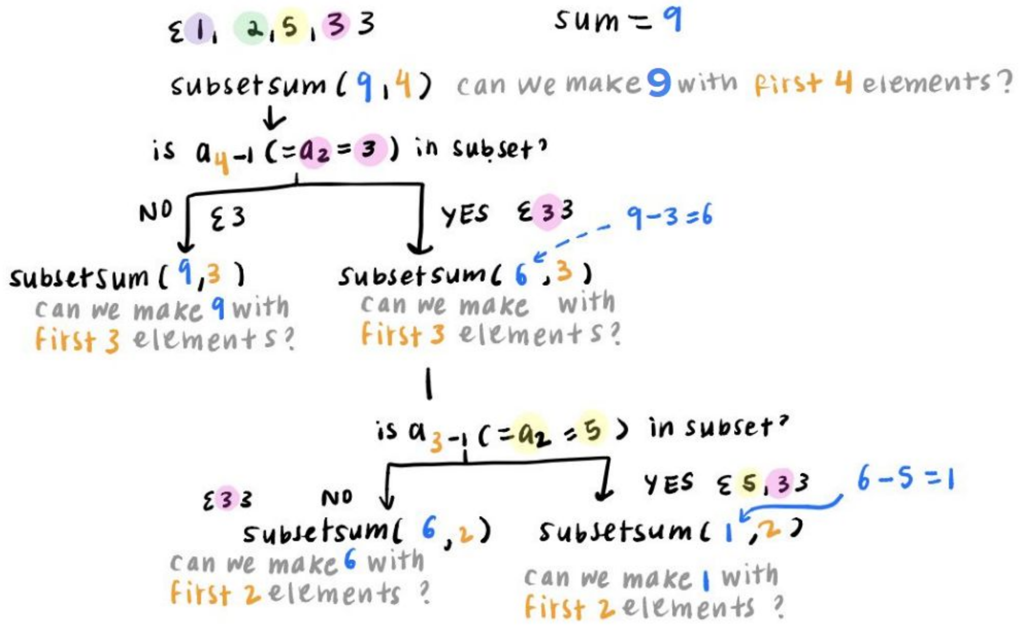
{1, 2, 5, 3} sum = 9

subsetsum(9, 4) can we make 9 with first 4 elements?

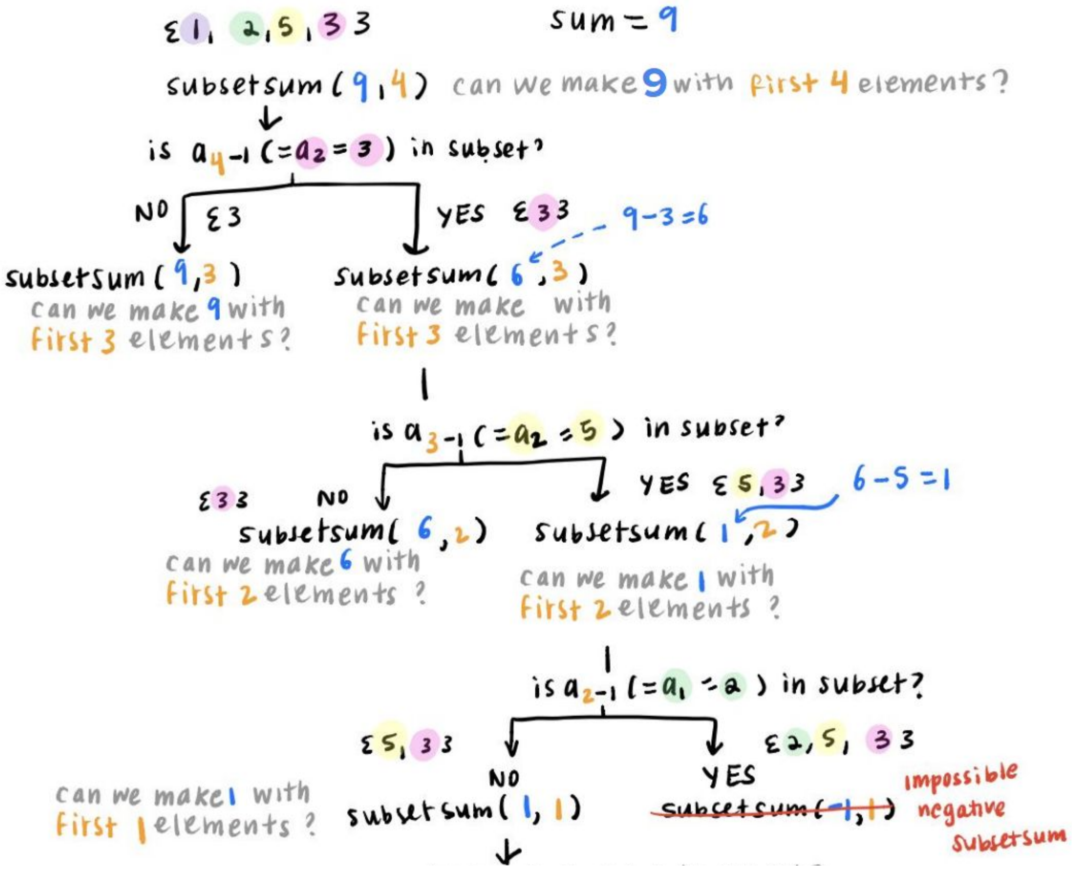
Positive Subset Sum Example (Top Down)



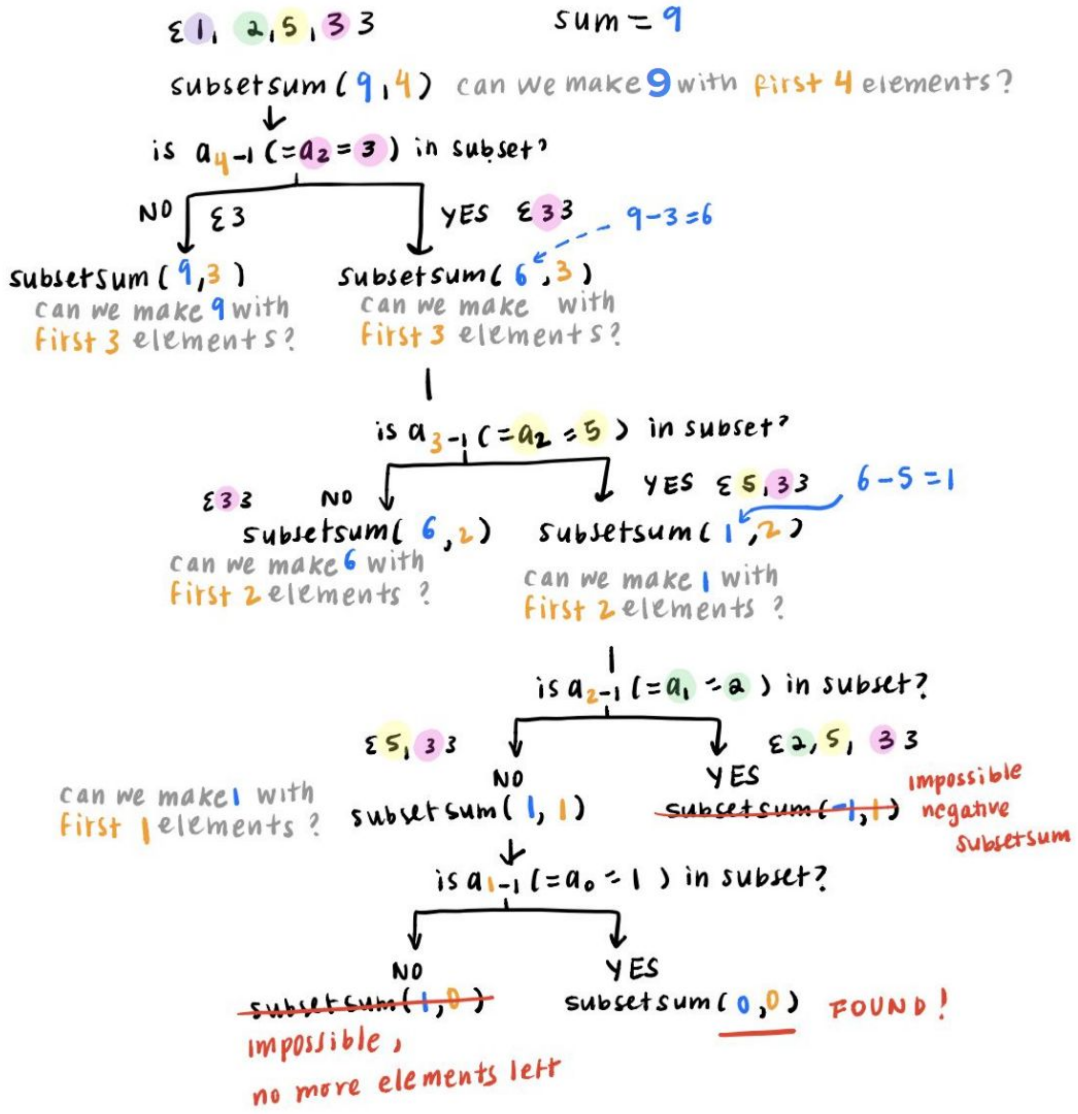
Positive Subset Sum Example (Top Down)



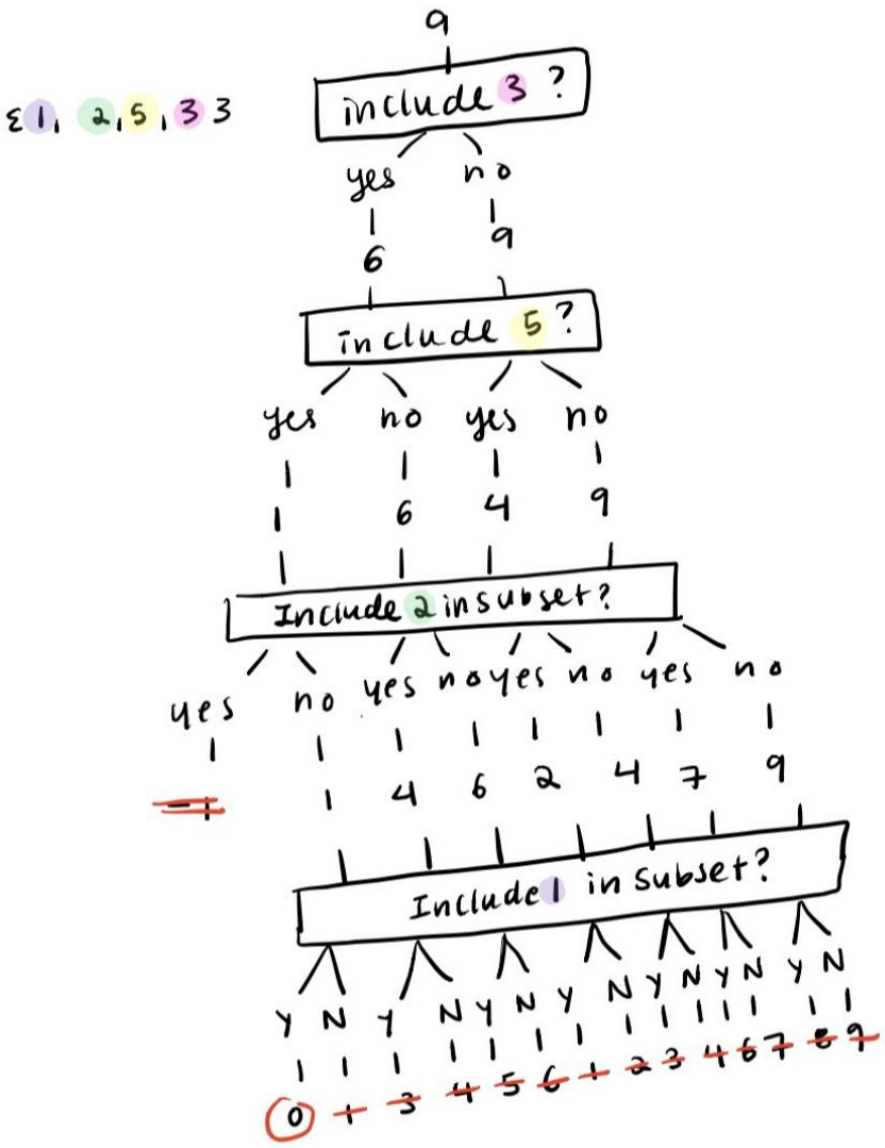
Positive Subset Sum Example (Top Down)



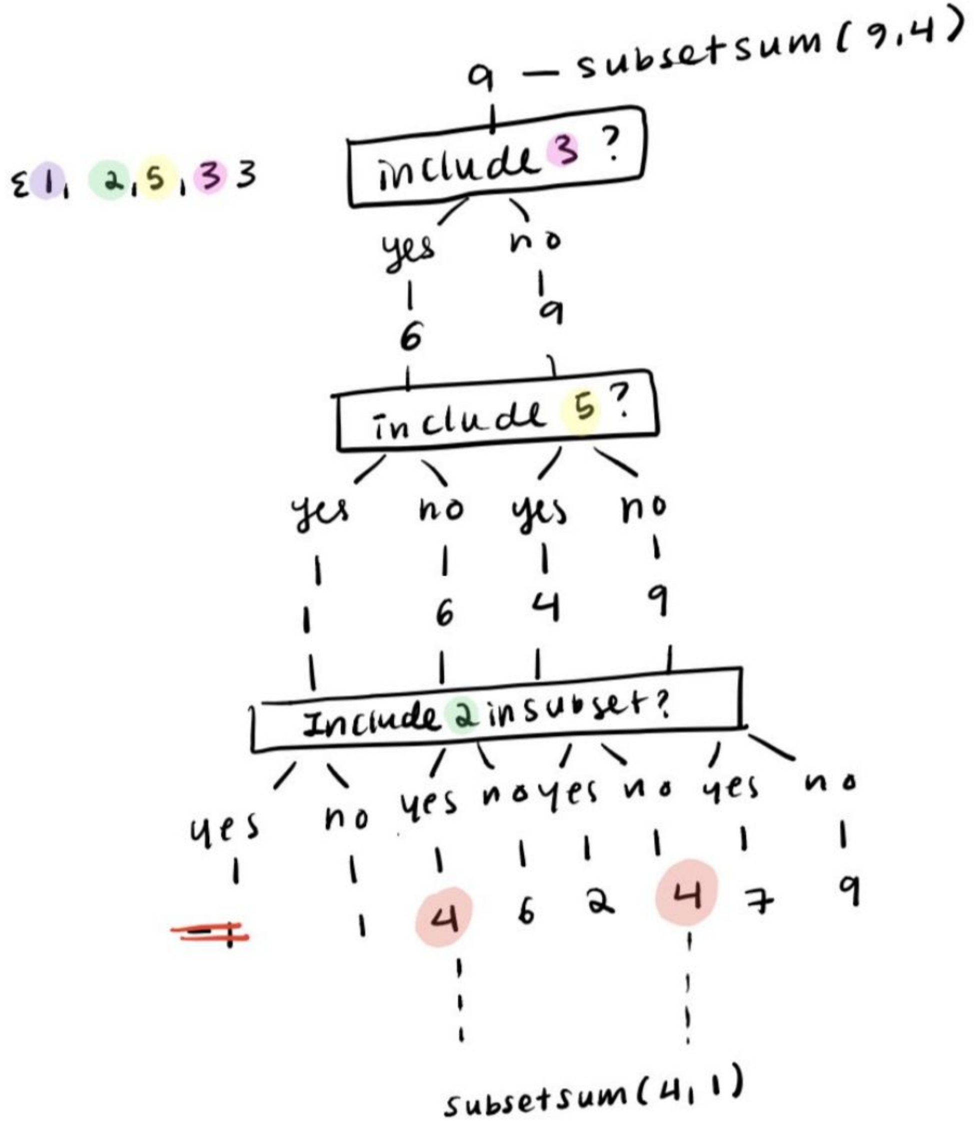
Positive Subset Sum Example (Top Down)



Positive Subset Sum Example (Top Down)



Positive Subset Sum Example (Top Down)



Positive Subset Sum Code (Naive)

Just Recursive (Naive, No DP):

```
bool can_sum_to( const vector<int>& nums, int k, int target ) {

    if (k == 0)
        return target == 0;     // base case
    if (target < 0)
        return false;           // hopeless

    bool take  = can_sum_to(nums, k-1, target - nums[k-1]);   // subset with a_{k-1}
    bool leave = can_sum_to(nums, k-1, target);               // subset without a_{k-1}

    bool ans = take || leave;

    return ans;
}

// the above is the recursive helper
bool subset_sum_to(const vector<int>& nums, int target) {
    return can_sum_to(nums, nums.size(), target);
}
```

Positive Subset Sum Code (Top Down)

Top-Down DP (Memoize):

```
bool can_sum_to( const vector<int>& nums, int k, int target, vector<unordered_map<int, bool>>& memo) {
    if (k == 0)
        return target == 0;      // base case
    if (target < 0)
        return false;            // hopeless
    if (memo[k].count(target))
        return memo[k][target];      // if value exists in map, return it

    bool take  = can_sum_to(nums, k-1, target - nums[k-1], memo);    // subset with a_{k-1}
    bool leave = can_sum_to(nums, k-1, target, memo);                 // subset without a_{k-1}

    bool ans = take || leave;    // calculate value
    memo[k][target] = ans;       // store result
    return ans;
}

bool subset_sum_to(const vector<int>& nums, int target) {
    vector<unordered_map<int, bool>> memo( nums.size() + 1 );
    return can_sum_to(nums, nums.size(), target, memo);
}
```

Runtime: O(target * num.size())

Positive Subset Sum Example (Bottom Up)

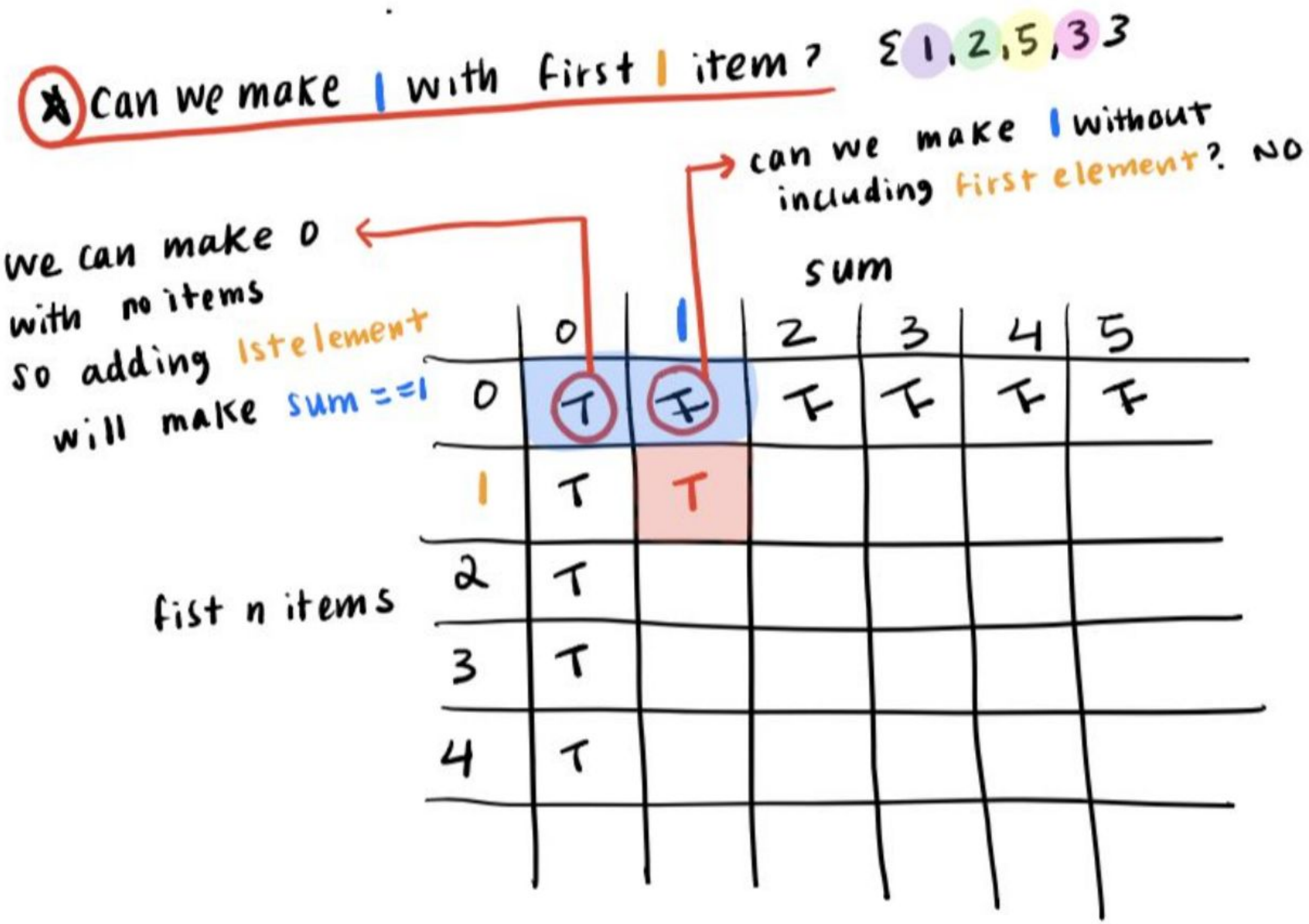
sum

sum>0 but no items available

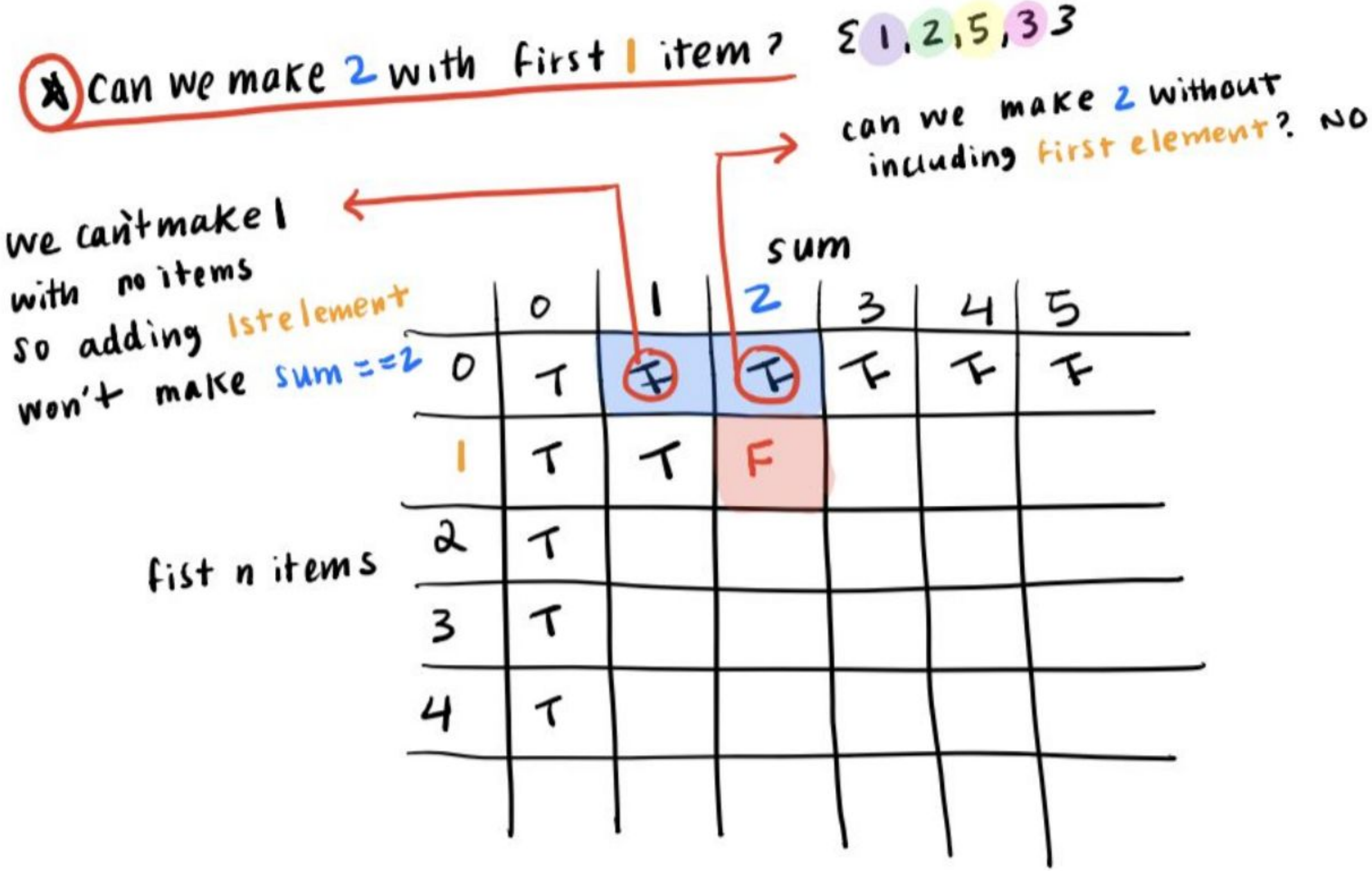
		0	1	2	3	4	5
none	0	T	F	F	F	F	F
fist n items	1	T					
	2	T					
	3	T					
	4	T					

sum==0 then True

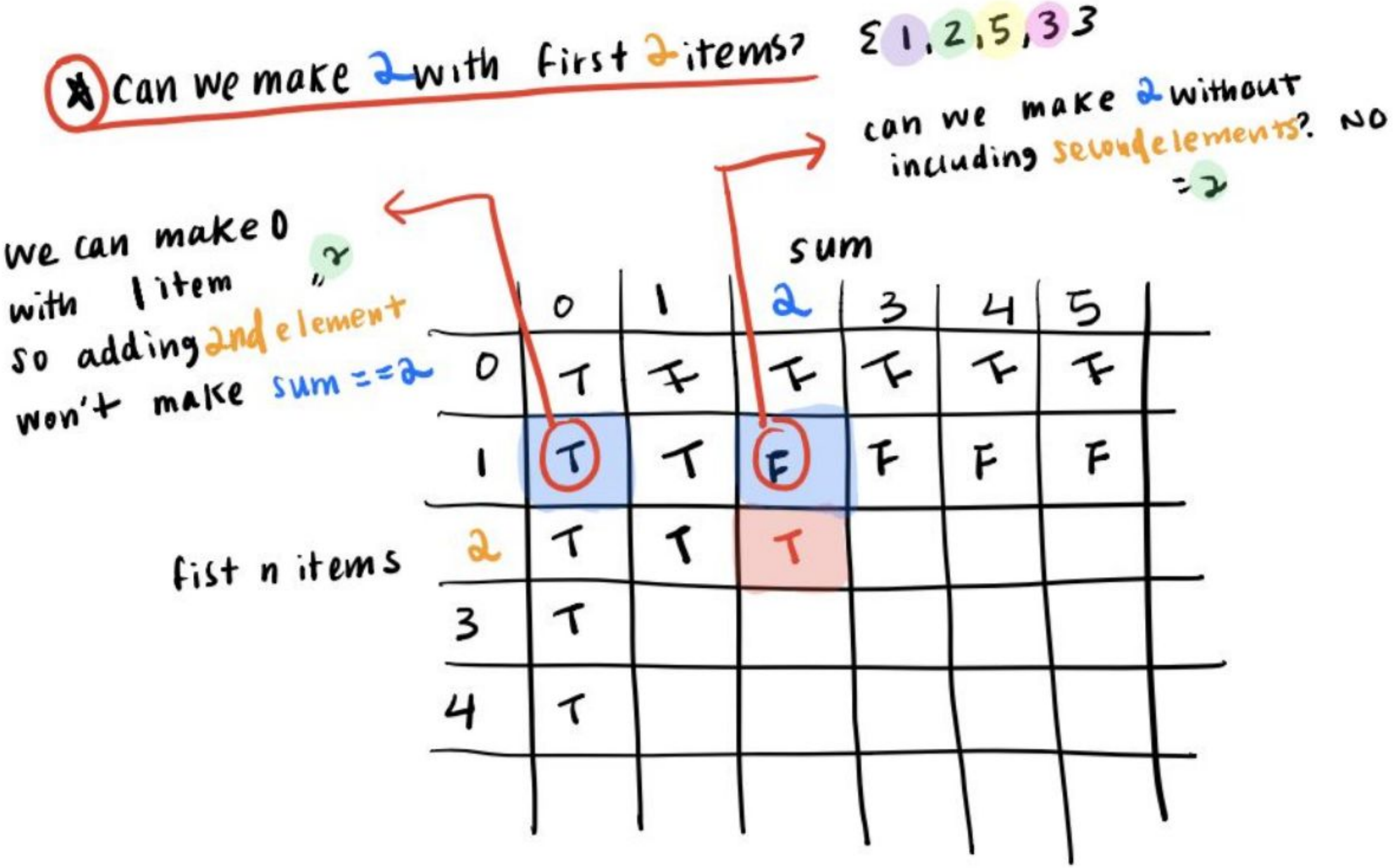
Positive Subset Sum Example (Bottom Up)



Positive Subset Sum Example (Bottom Up)



Positive Subset Sum Example (Bottom Up)



Positive Subset Sum Example (Bottom Up)

Question!

Can we make 5 with first 4 items?

{1, 2, 5, 3}

can we make 5 without including 4th elements? =3

can we make 2 with 4 items ,,3 so adding 4th element won't make sum ==5

fist n items

fist n items \ sum	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	T	F	F	F	F
2	T	T	T	T	F	F
3	T	T	T	T	F	T
4	T	T	T	T	F	

Positive Subset Sum Example (Bottom Up)

Can we make 5 with first 4 items? {1, 2, 5, 3}

{2, 3} + {3}

can we make 5 without including 4th elements? =3 {5}

Answer!

can we make 2 with 4 items ,3 so adding 4th element won't make sum == 5

fist n items

first n items \ sum	0	1	2	3	4	5
0	T	F	F	F	F	F
1	T	T	F	F	F	F
2	T	T	T	T	F	F
3	T	T	T	T	F	T
4	T	T	T	T	F	T

Positive Subset Sum Code (Bottom Up)

Bottom-Up DP:

```
bool subset_sum(vector<int> nums, int target){
    vector<vector<bool>> table(target + 1, vector<bool>(nums.size()+1));

    for(int t = 0; t < target; ++t) {
        for(int k = 0; k <= nums.size(); ++k) {
            if (k == 0)
                table[t][k] = (t == 0);    // base case
            else
                table[t][k] = table[t][k-1] || table[t - nums[k-1]][k-1];// build table
        }
    }

    return table[target][nums.size()];   // return needed value
}
```

Runtime: O(target * num.size())