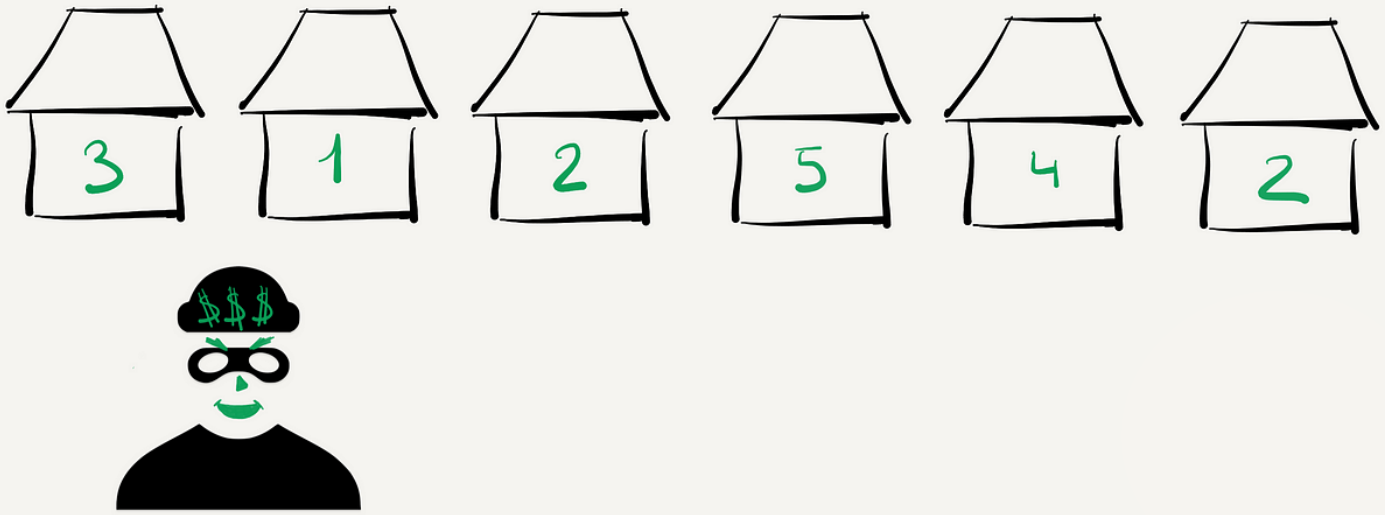
House Robber Problem



[House robber problem by Sergey Piterman]

Description | Editorial | Solutions (8.2K) | Submissions

198. House Robber

Medium | 17.4K | 329

Companies

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight **without alerting the police***.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house
5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.
```

Input: [3, 1, 2, 5, 4, 2]

Output: 10 (3 + 5 + 2)

What observations can we make?

Brute Force:

Try stealing from all valid combinations of houses

How?

[3, 1, 2, 5, 4, 2]

⇓ Try all valid subsets

{ [3], [1], [2], [5], [4], [2]

[3,2], [3,5], [3,4], [3,2],

[1,5], [1,4], [1,2]

[2,4], [2,2], [5,2]

[3,2,4], [3,2,2], [3,5,2], [1,5,2] }

"For each house in the neighborhood you are deciding between stealing from it and skipping the next house, or skipping the current house and making the decision again at the next house"

- Can be solved recursively with this relationship

maxGoldAt (houseIndex) =

Maximum of (houses[houseIndex] + maxGoldAt (houseIndex + 2)

and

maxGoldAt (houseIndex + 1))

Base Case :

maxGoldAt (houseIndex) =

if houseIndex ≥ houses.length

return 0

Time Complexity:

Exponential $< O(2^N)$

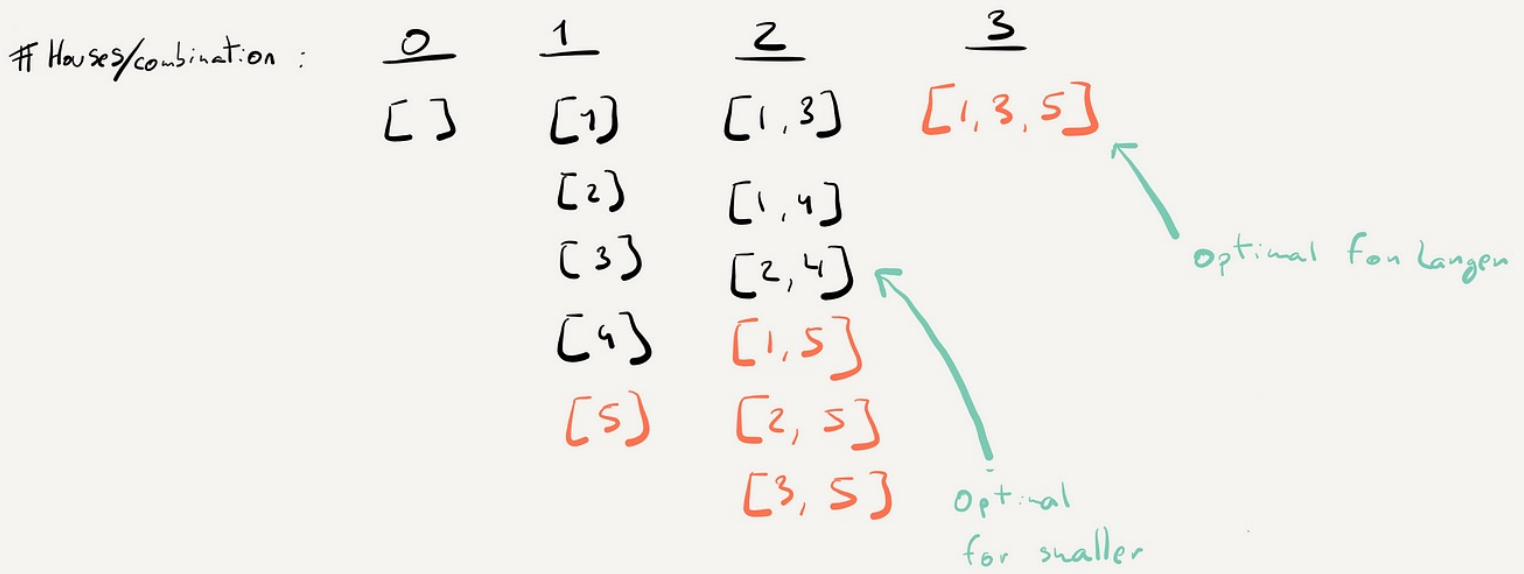
Adding 1 additional House doesn't quite double the amount of work, but it increases non-linearly

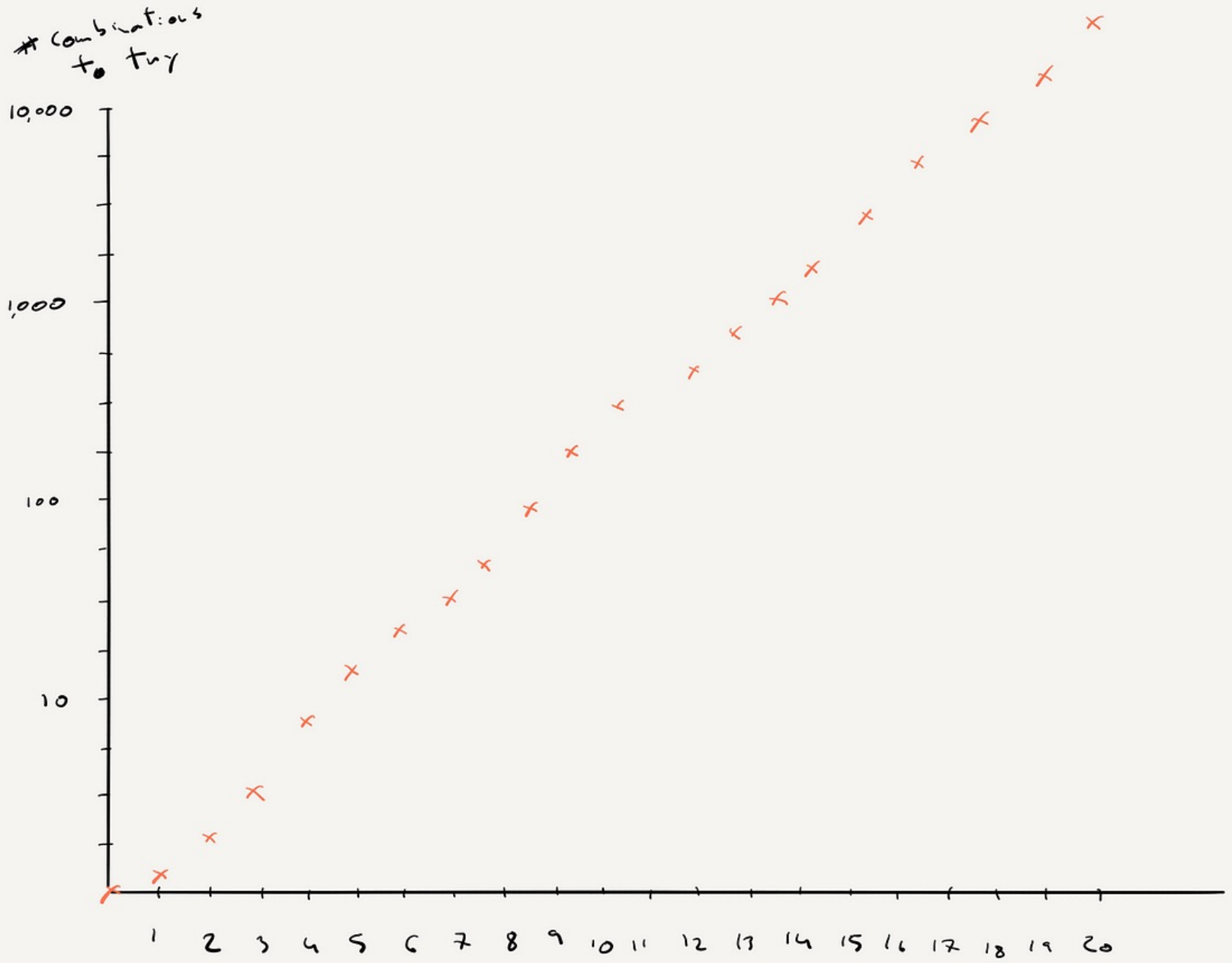
Smaller Input

[1, 2, 3, 4]

Larger Input

[1, 2, 3, 4, 5]




combinations to try
10,000
1,000
100
10
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Hours

Space complexity:

Auxiliary: $O(1)$

Call stack: $O(N)$

At most we'll have $N/2$ calls on the call stack when we just skip 1 house at a time.

Optimizations

1) Currently we are trying a lot of junk combinations:

why bother with trying only stealing from 1 house?*

* Key Assumption: All houses have positive gold
(and not poison... no negative #s)

Ask the interviewer!!

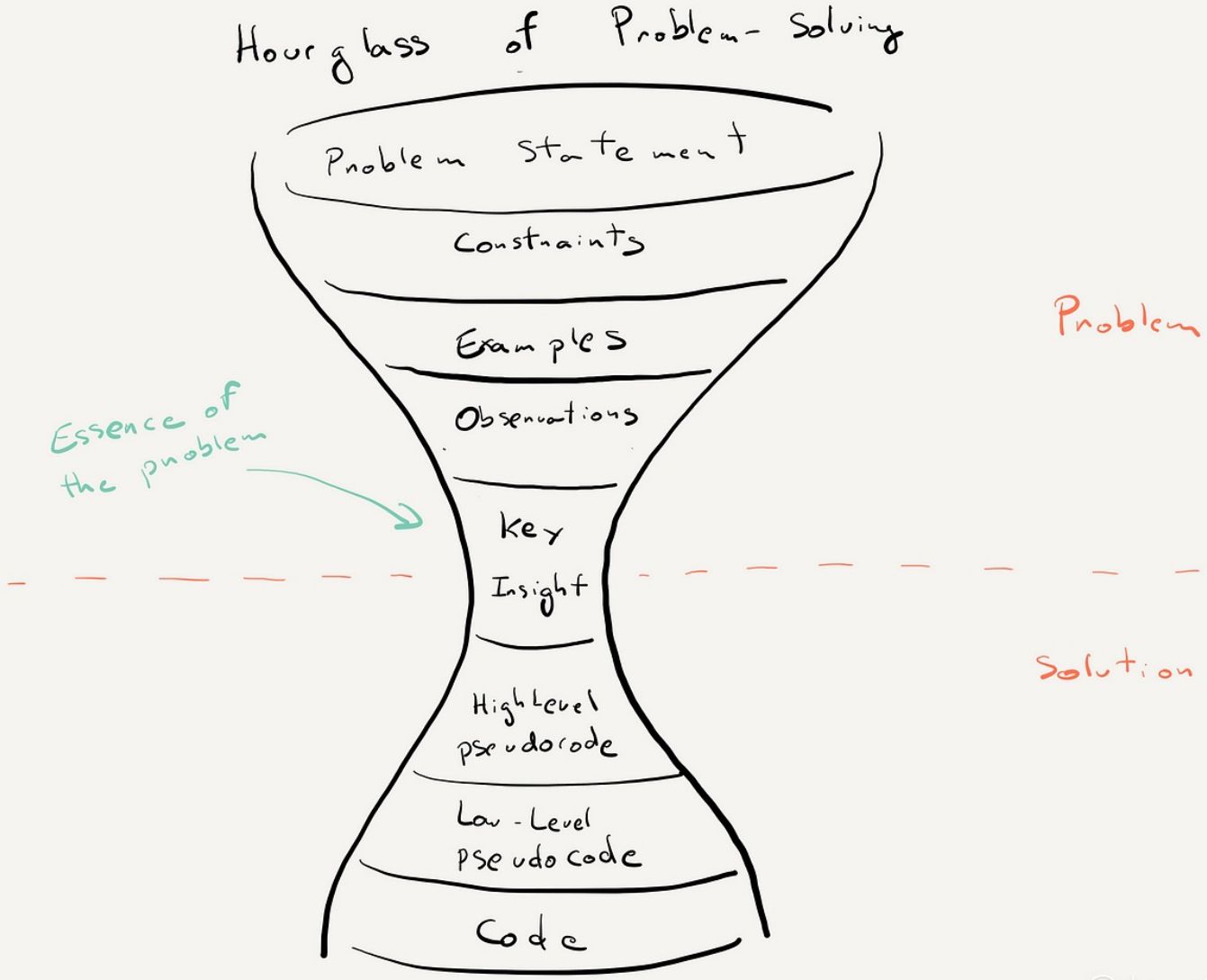
2) We are trying combinations contained in others:

E.g. [3, 1, 2, 5, 4, 2]

We are trying [3], [5], [2], [3,5], [3,2], [5,2]

all inside

[3, 5, 2]


Hourglass of Problem-Solving
Problem Statement
Constraints
Examples
Observations
Key Insight
High Level pseudocode
Low-Level pseudo code
Code
Essence of the problem
Problem
Solution

Key Insight:

Max distance between any two houses is 2

Example:

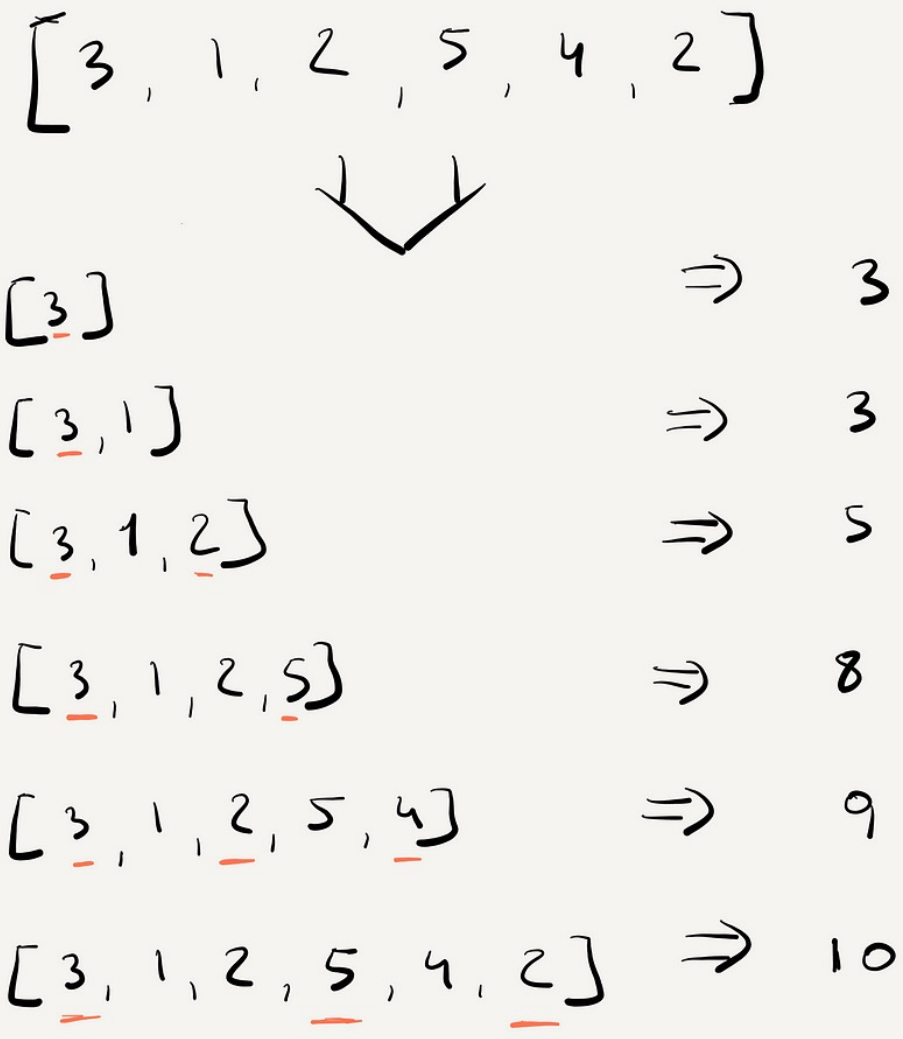
In:		Out:
[3, 1, 5]	⇒	8
[3, 1, 1, 5]	⇒	8
[3, 1, 1, 1, 5]	⇒	9

Reasoning: Any gap larger than 2 leaves gold we can steal

... So How do we use this insight?

Let's break our example down ...

See what happens to the optimal amount of total gold if you incrementall add houses to the neighborhood.


[3, 1, 2, 5, 4, 2]
[3] ⇒ 3
[3, 1] ⇒ 3
[3, 1, 2] ⇒ 5
[3, 1, 2, 5] ⇒ 8
[3, 1, 2, 5, 4] ⇒ 9
[3, 1, 2, 5, 4, 2] ⇒ 10

Notice!

#s always either stay the same or increase

2 more examples w/ obvious solutions

[4, 1, 5, 1, 8, 1, 6]

[3, 1, 1, 6, 1, 1, 7]

[4, 1, 5, 1, 8, 1, 6]

[4] ⇒ 4

[4, 1] ⇒ 4

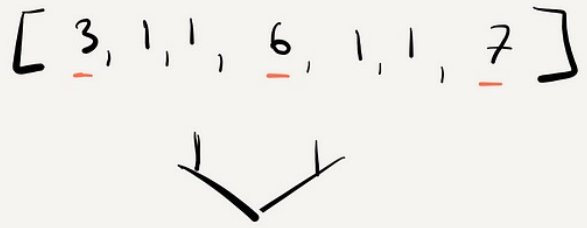
[4, 1, 5] ⇒ 9

[4, 1, 5, 1] ⇒ 9

[4, 1, 5, 1, 8] ⇒ 17

[4, 1, 5, 1, 8, 1] ⇒ 17

[4, 1, 5, 1, 8, 1, 6] ⇒ 23



[3] ⇒ 3

[3, 1] ⇒ 3

[3, 1, 1] ⇒ 4

[3, 1, 1, 6] ⇒ 9

[3, 1, 1, 6, 1] ⇒ 9

[3, 1, 1, 6, 1, 1] ⇒ 10

[3, 1, 1, 6, 1, 1, 7] ⇒ 16

Ascending pattern holds.

Let's rewrite those different outputs as tables:

Inputs:	[4, 1, 5, 1, 8, 1, 6]	[3, 1, 1, 6, 1, 1, 7]
Max Gold:	[4, 4, 9, 9, 17, 17, 23] → output	[3, 3, 4, 9, 9, 10, 17] → output

Tables represent the maximum gold you can have

by the time you reach the house at that index

Input : 3

Max Gold : 3

Input : 3 , 5

Max Gold : 3 , 5

Input : 3, 5, 3

Max Gold : 3, 5, 6

Input : 3, 5, 1

Max Gold : 3, 5, 5

Input: 3, 5, 1, 3

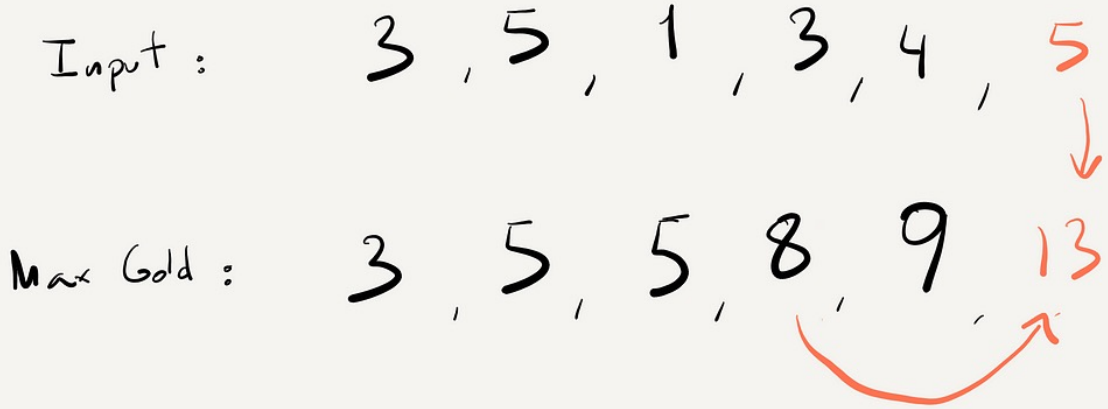
Max Gold: 3, 5, 5, 8

Input : 3, 5, 1, 3, 4

Max Gold : 3, 5, 5, 8, 9

The pseudocode would be :

> *“For every house in the list, either add its gold to the max gold 2 indices back or keep the max gold 1 index back.”*

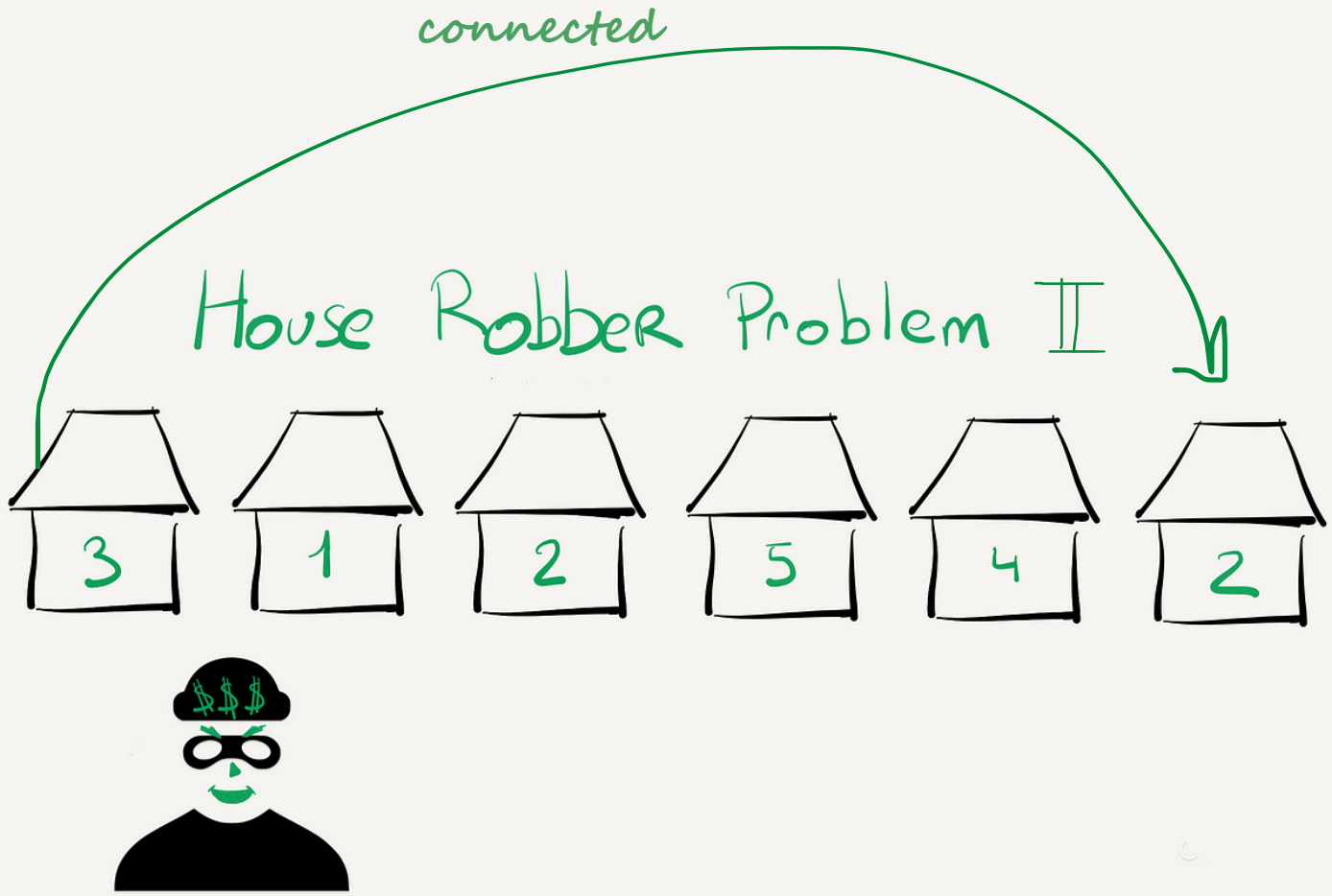

Input : 3, 5, 1, 3, 4, 5
Max Gold : 3, 5, 5, 8, 9, 13

Input : 3, 5, 1, 3, 4, 5, 1

Max Gold : 3, 5, 5, 8, 9, 13, 13

Question!

Why doesn't greedy approach work in this case?


connected
House Robber Problem II
3
1
2
5
4
2

213. House Robber II

Hint

Medium 8.1K 118

Companies

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

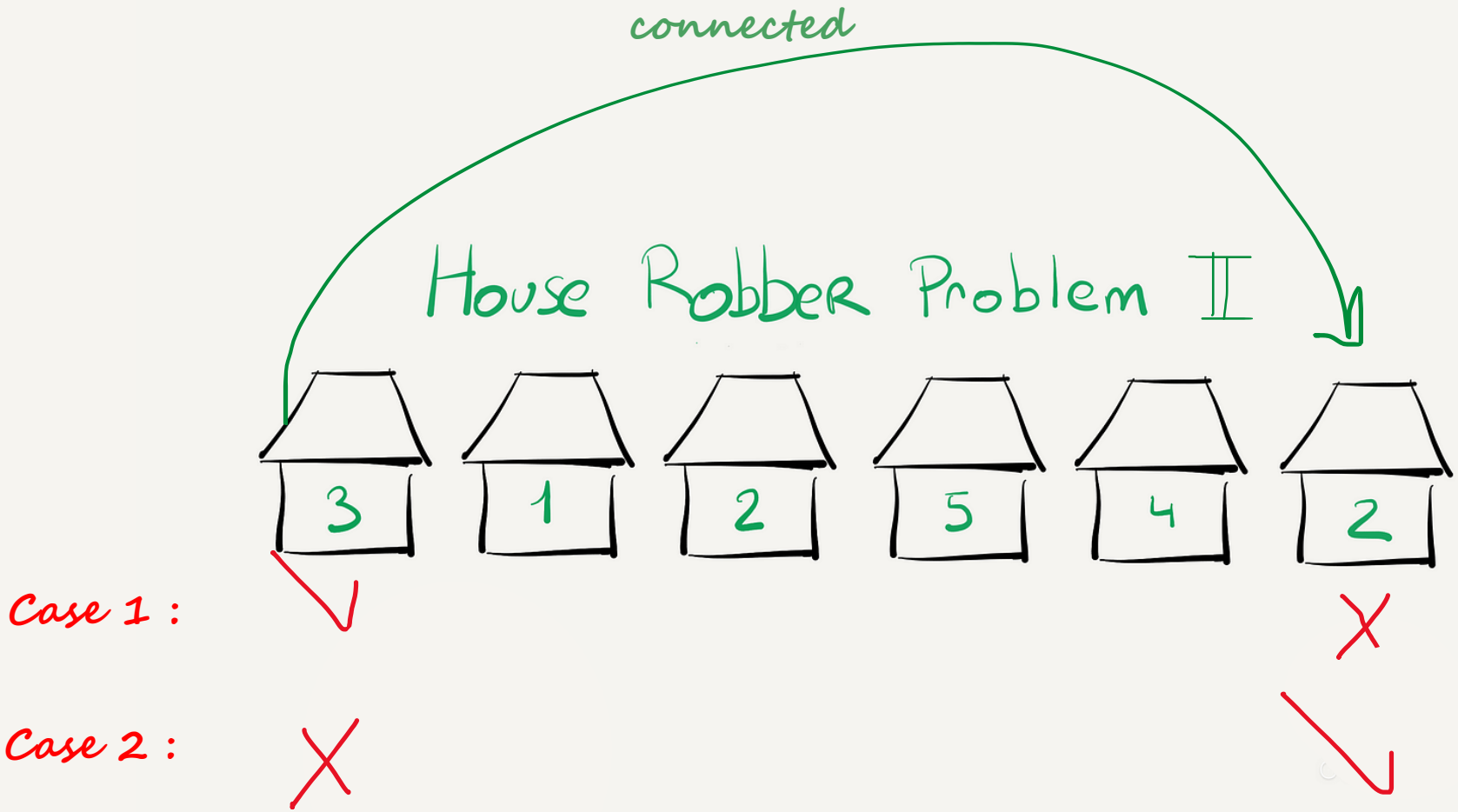
Given an integer array `nums` representing the amount of money of each house, return *the maximum amount of money you can rob tonight* ***without alerting the police***.

Example 1:

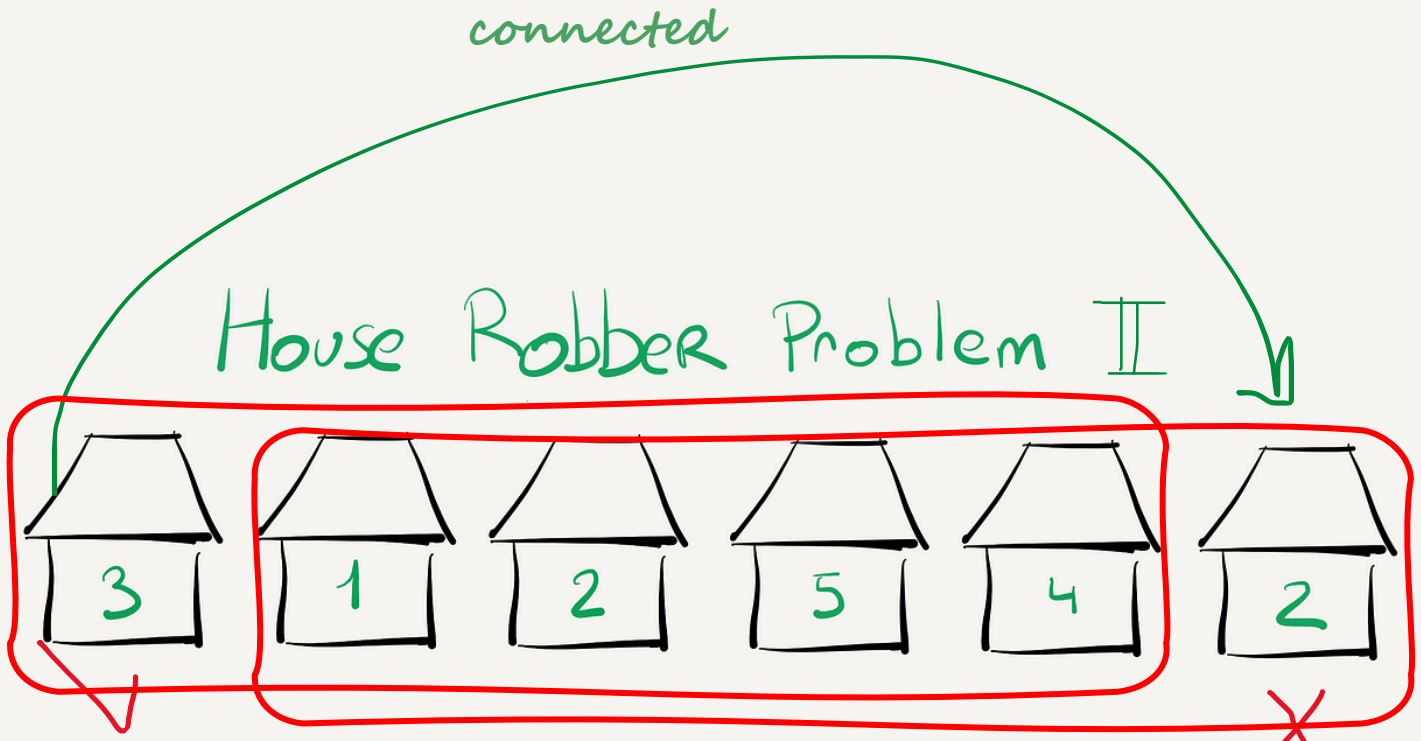
```
Input: nums = [2,3,2]
Output: 3
Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money
= 2), because they are adjacent houses.
```

Example 2:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
```

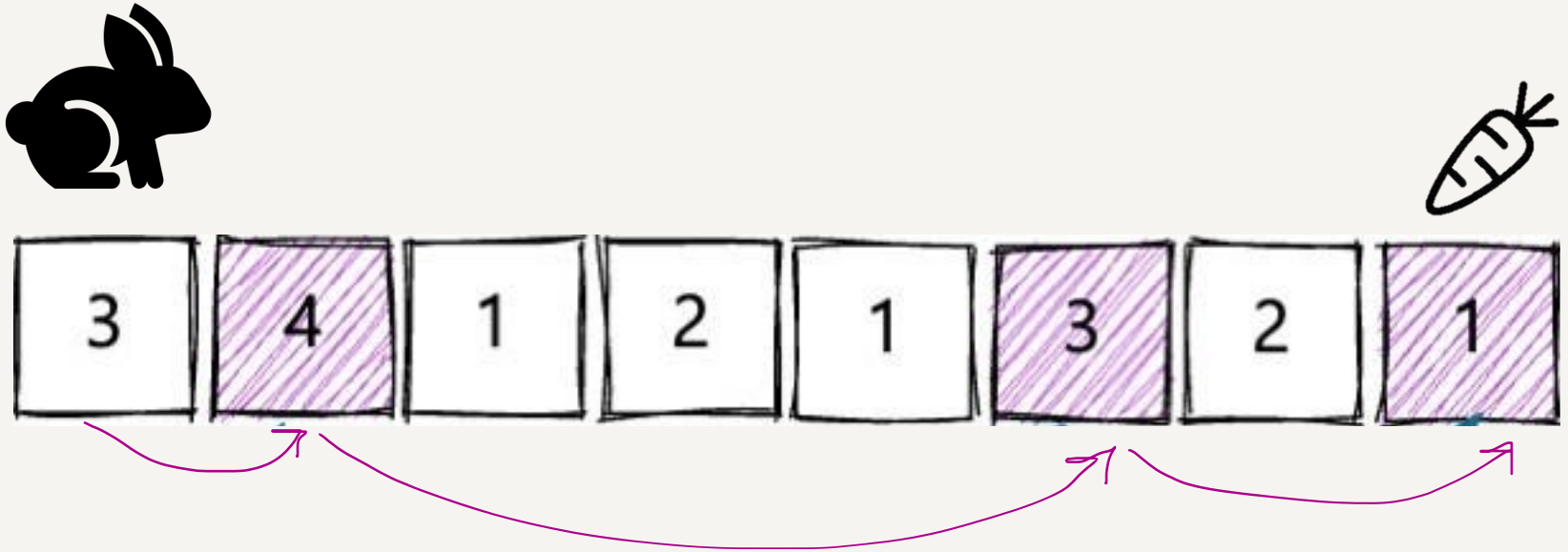


How to reuse the solution of house robber one?
Can I follow the same strategy?



Each 2 cases are same as house robber 1 problem :
Pick the max one !

Jump Game



55. Jump Game

Medium 15.9K 813

Companies

You are given an integer array `nums`. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return `true` *if you can reach the last index, or* `false` *otherwise*.

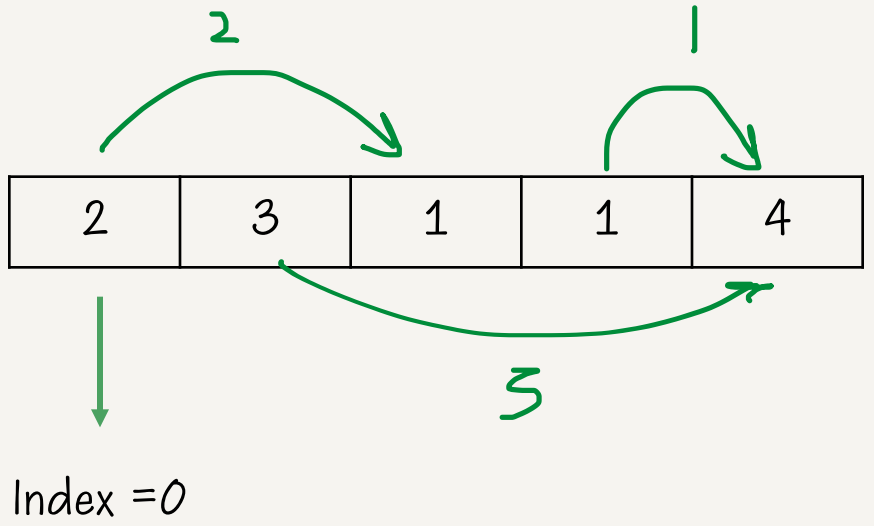
Example 1:

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.
```

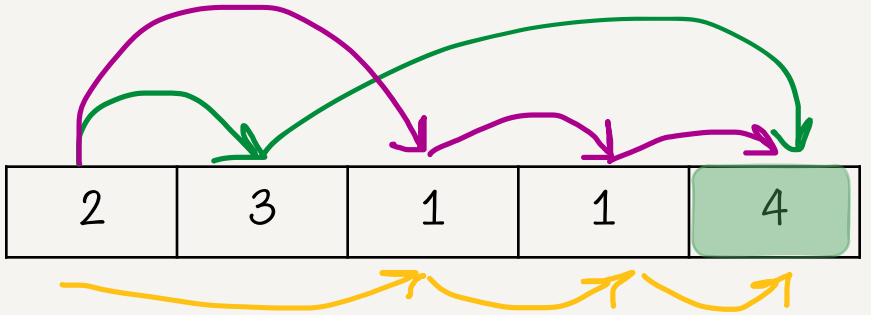
Each element ⟶ maximum possible jump



Can we reach the last index? (here is element 4, index=4)

Return : true/false

Each element ⟶ maximum possible jump



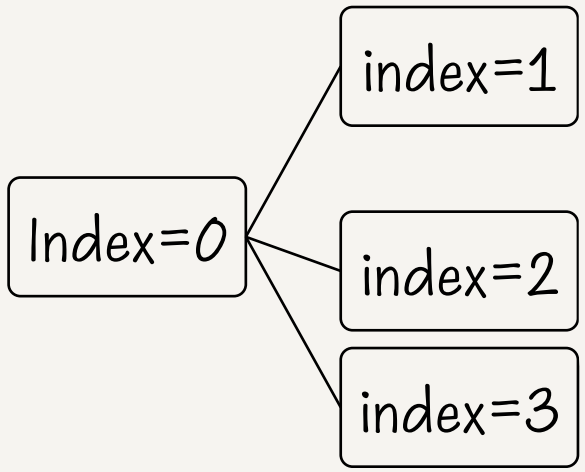
true

Can we reach the last index? (here is element 4, index=4)

Yes !

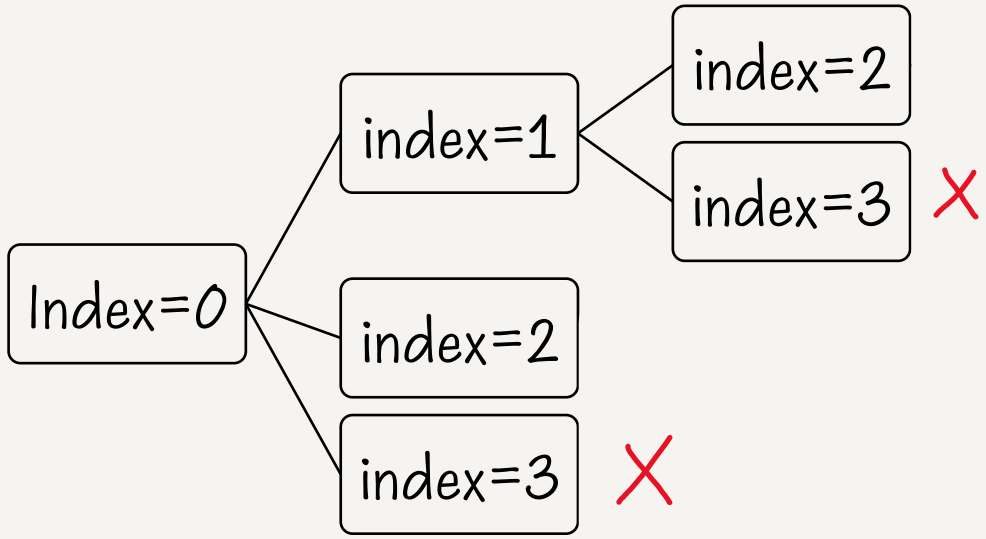
How about this example ? First approach : Brute-force

	3	2	1	0	4
Index=	0	1	2	3	4



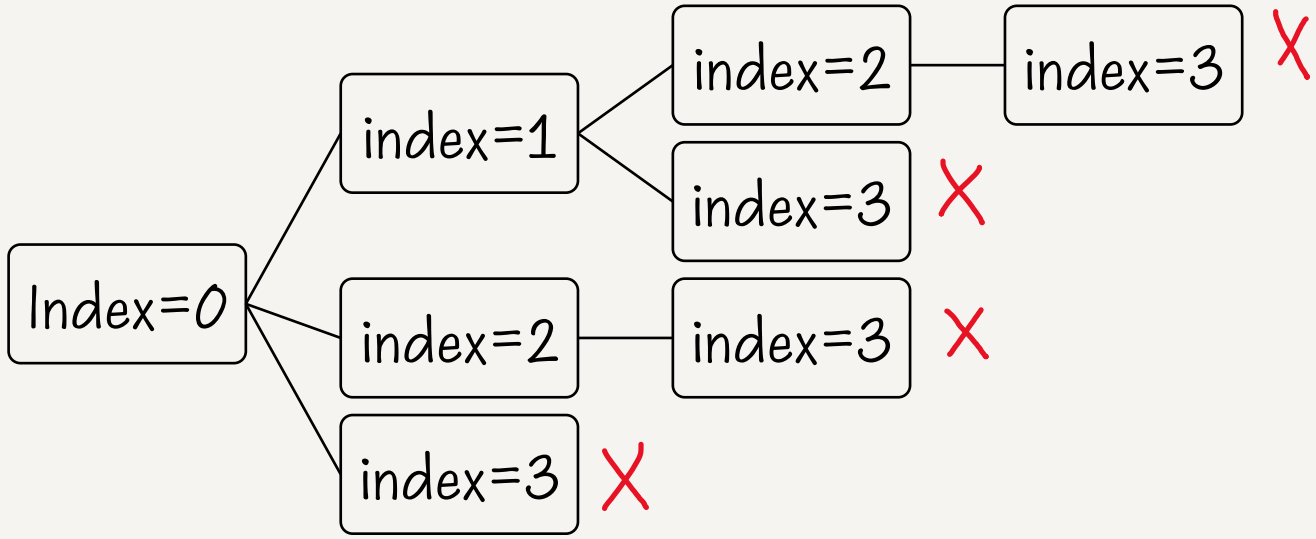
How about this example ? First approach : Brute-force

	3	2	1	0	4
Index=	0	1	2	3	4



How about this example ? First approach : Brute-force

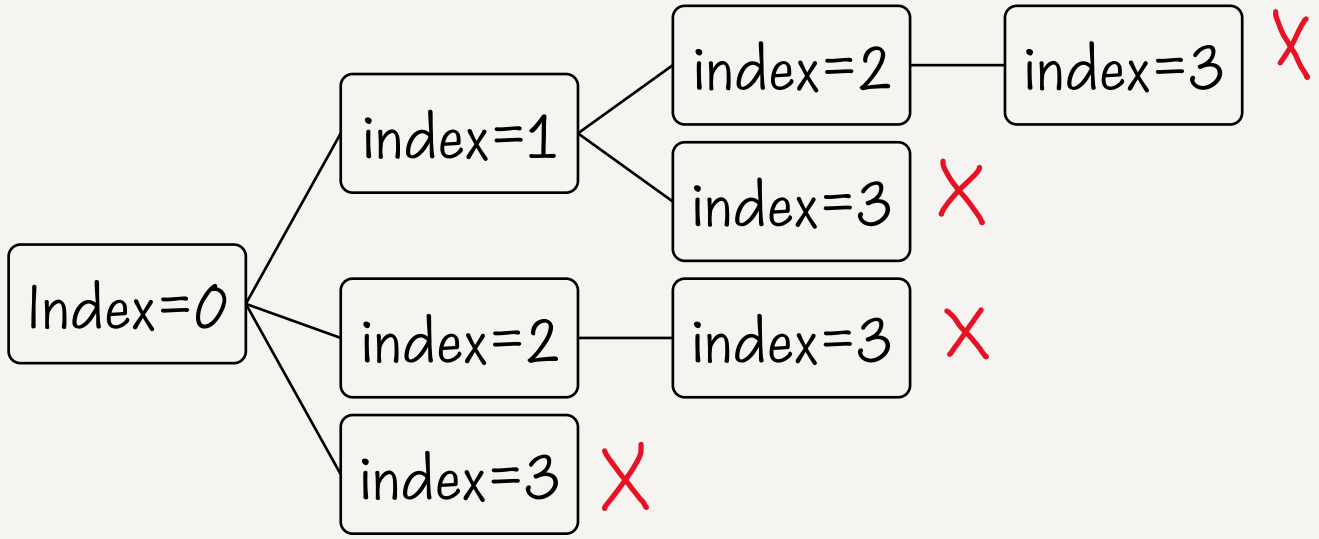
	3	2	1	0	4
Index=	0	1	2	3	4



First approach : Brute-force

Time complexity : exponential

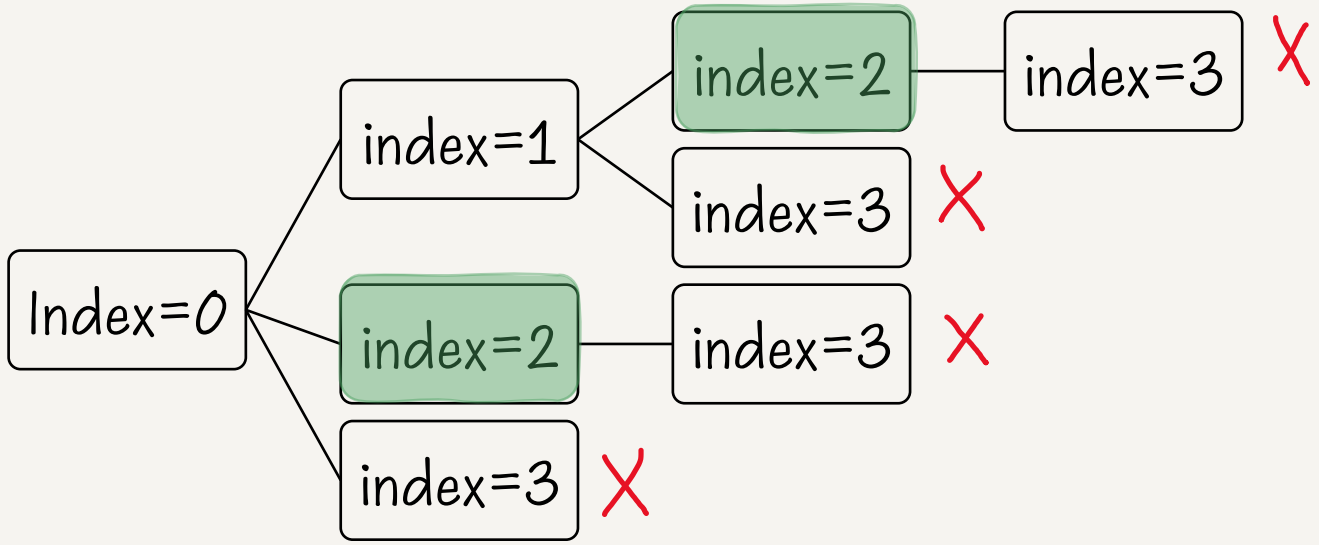
	3	2	1	0	4
Index=	0	1	2	3	4



First approach : Brute-force

Time complexity : exponential

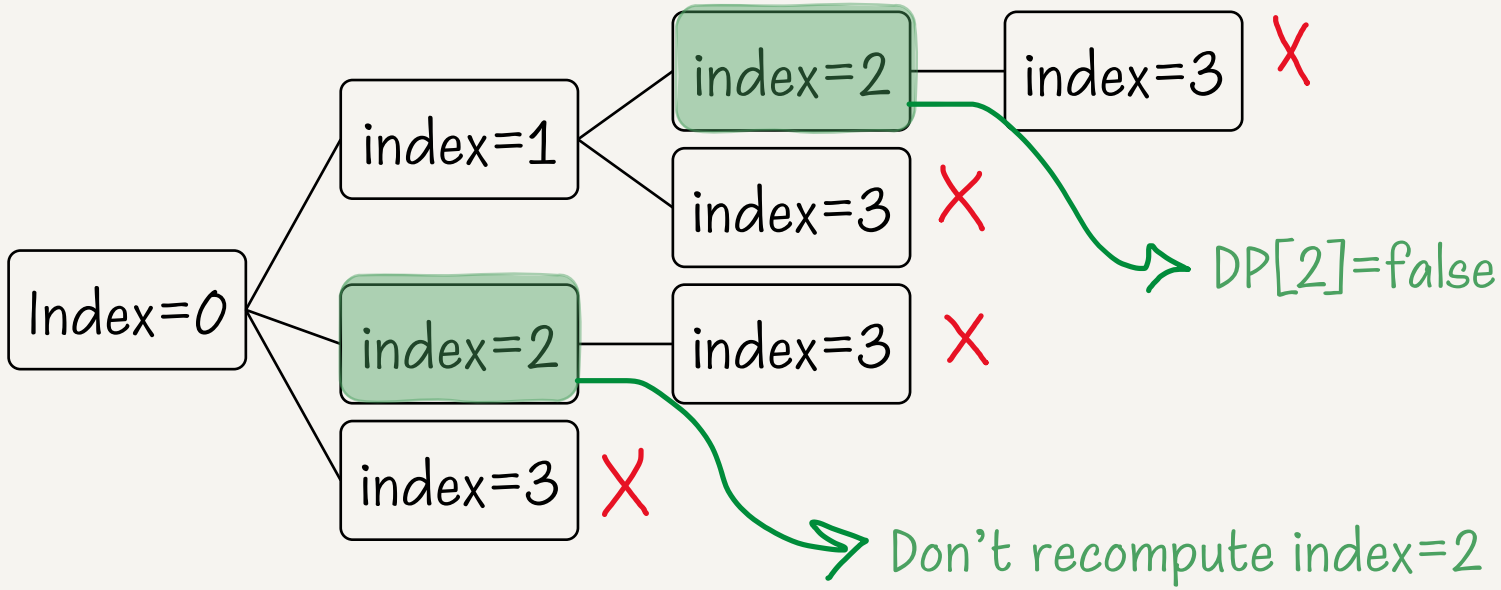
	3	2	1	0	4
Index=	0	1	2	3	4



Second approach : Brute-force → Careful Brute-force : Dynamic Programming

Time complexity : exponential

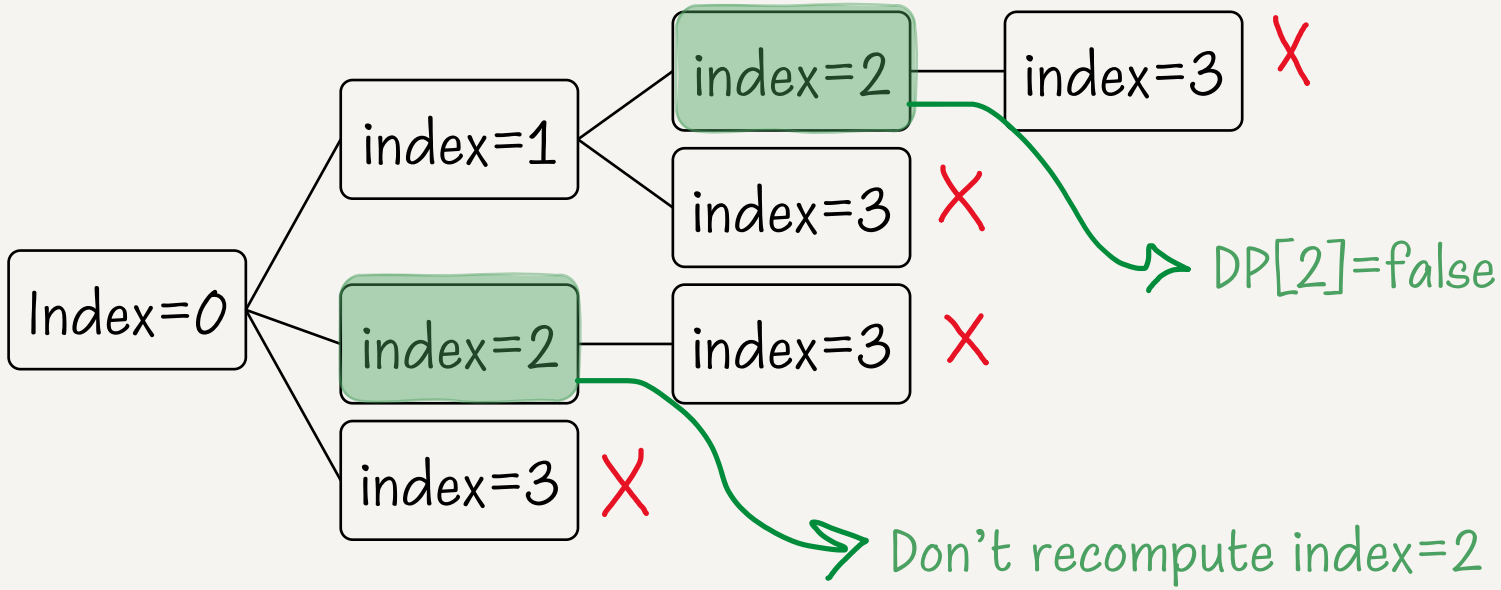
	3	2	1	0	4
Index=	0	1	2	3	4



Second approach : Brute-force → Careful Brute-force : Dynamic Programming

Time complexity : exponential → O (n^2)

	3	2	1	0	4
Index=	0	1	2	3	4



Using a cash to reduce time complexity :

DP[2]= false

DP[3]= false

DP[1]= false

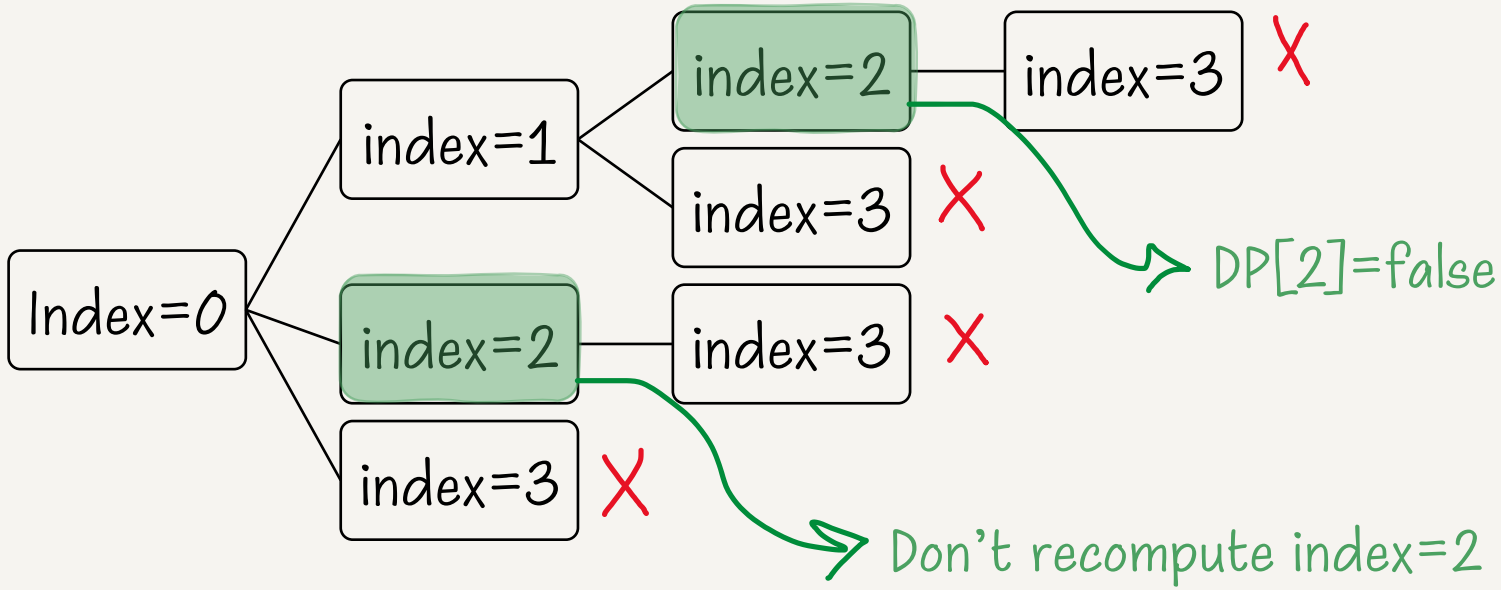
DP[0]= false

Second approach : DP

Any better idea?

Time complexity : exponential → O (n^2)

	3	2	1	0	4
Index=	0	1	2	3	4



Using a cash to reduce time complexity :

DP[2]= false

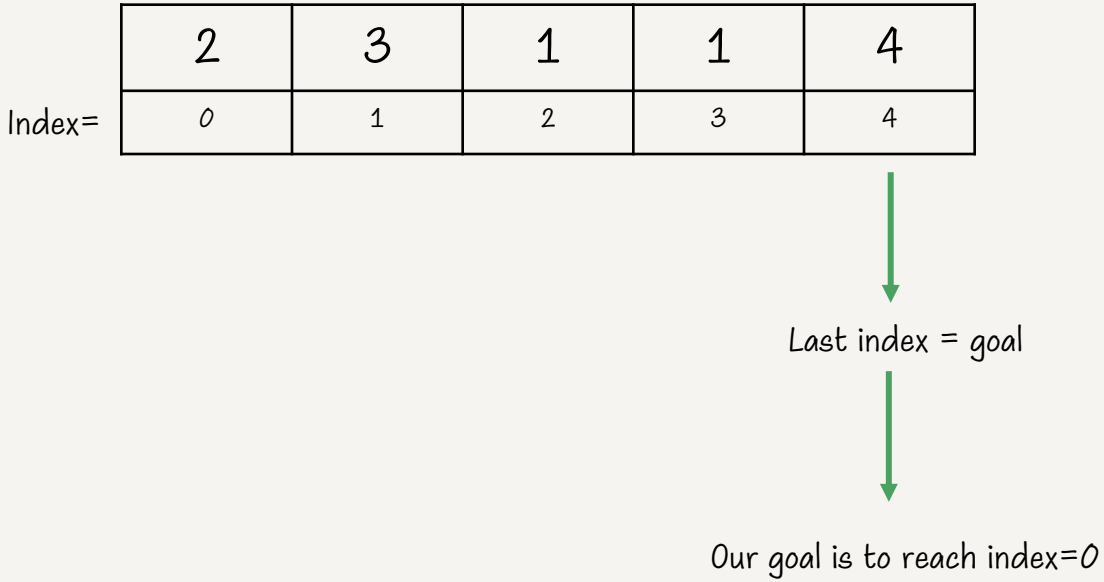
DP[3]= false

DP[1]= false

DP[0]= false

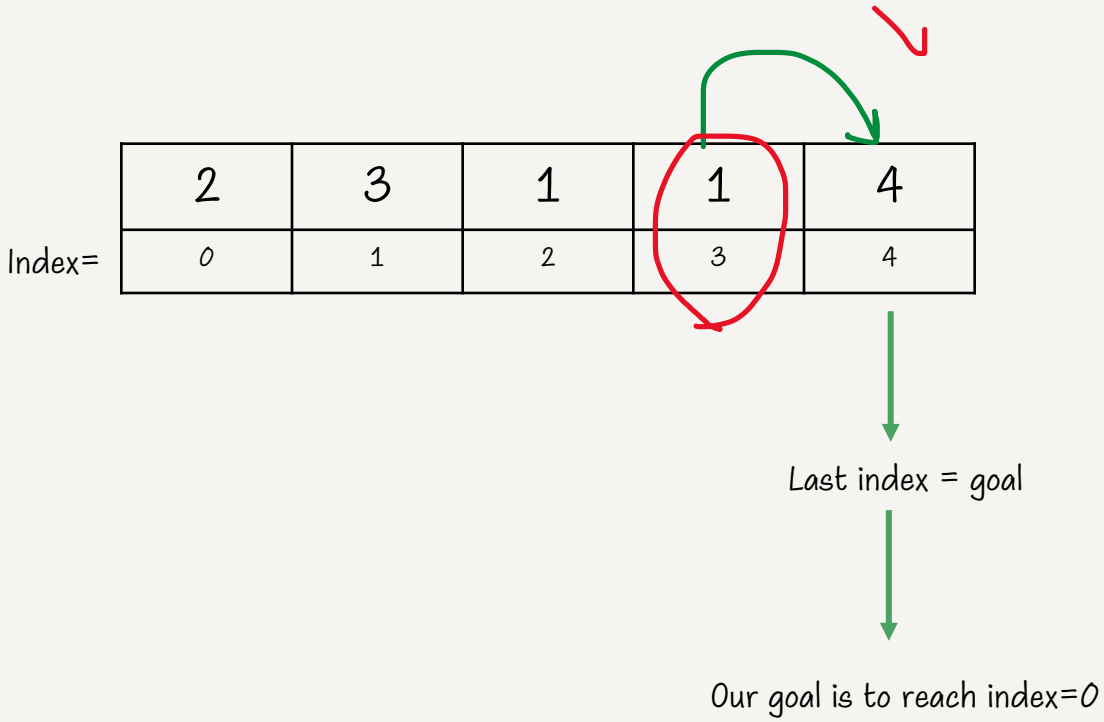
Last approach : Greedy Algorithms

let's reverse our logic; instead of starting from Index 0, why not start from the last index?



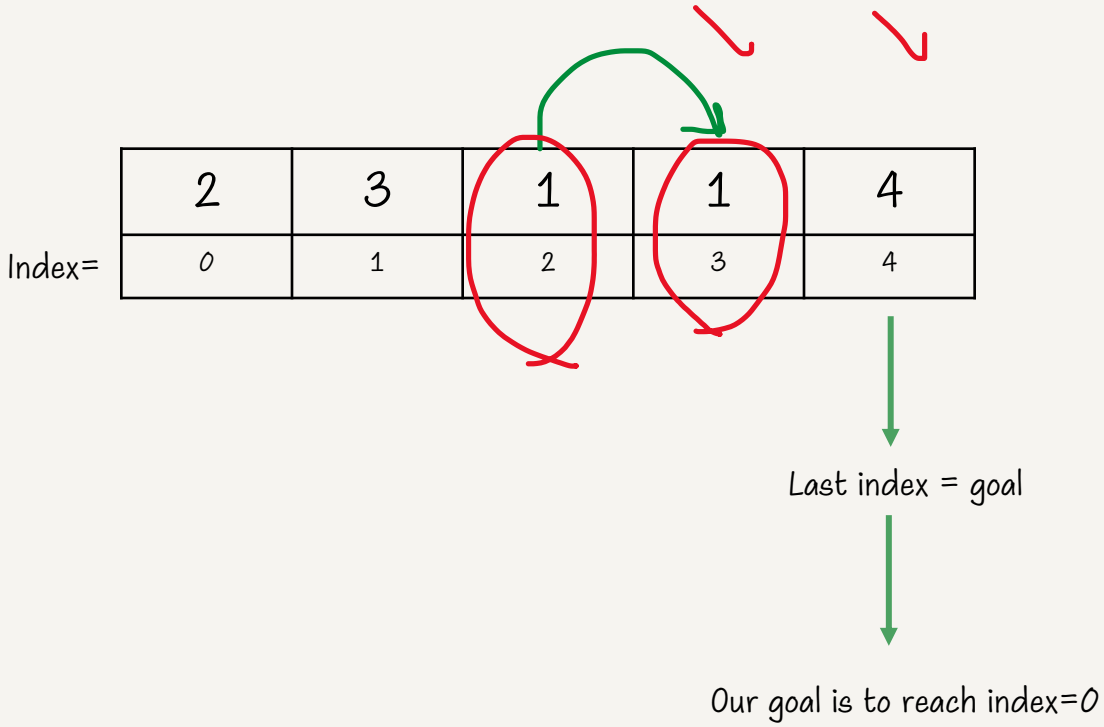
Last approach : Greedy Algorithms

let's reverse our logic; instead of starting from Index 0, why not start from the last index?



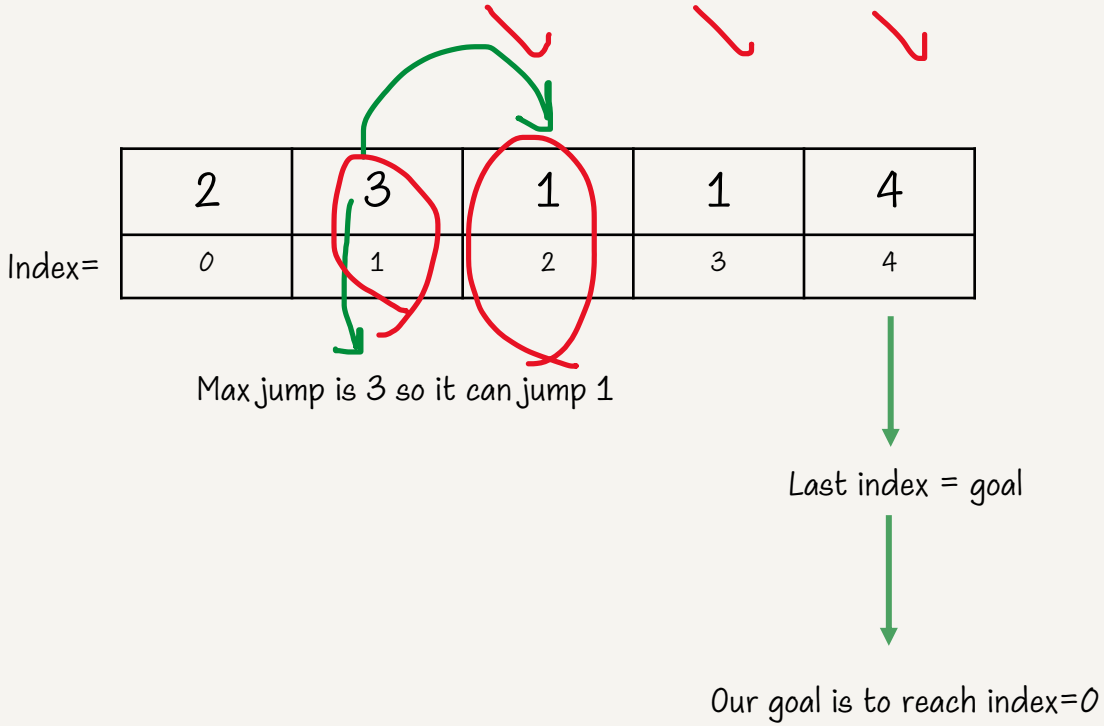
Last approach : Greedy Algorithms

let's reverse our logic; instead of starting from Index 0, why not start from the last index?



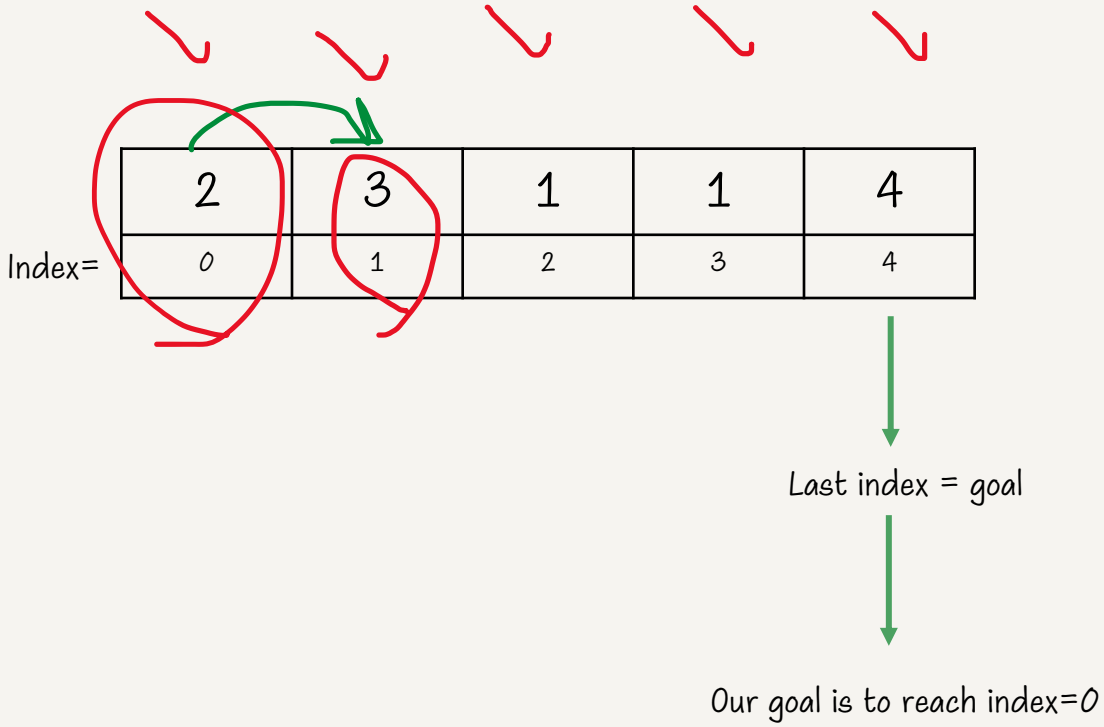
Last approach : Greedy Algorithms

let's reverse our logic; instead of starting from Index 0, why not start from the last index?



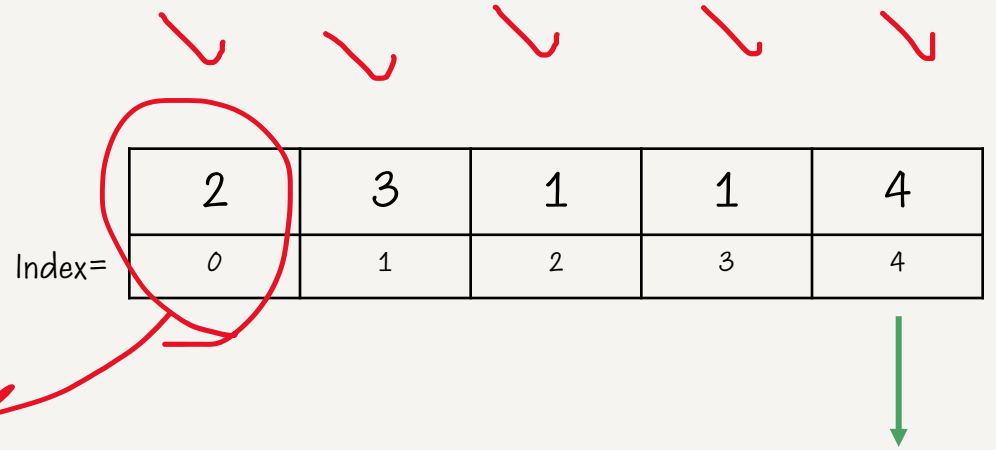
Last approach : Greedy Algorithms

let's reverse our logic; instead of starting from Index 0, why not start from the last index?



Last approach : Greedy Algorithms

let's reverse our logic; instead of starting from Index 0, why not start from the last index?



If we've reached the first index, we can jump to the last index, if the goal post is not, we cannot jump to it.

Last index = goal

Our goal is to reach index=0

Recap

Greedy Approach:

1. We're going to initialize a goal post, which is initially the last index.
2. We're going to setup a For Loop that iterates over the array backwards.
3. We ask each element 'Can you jump to or over the goal post from here?' if so, we set the goal post to that element.
4. If not, we continue looping.
5. We repeat this logic until we've been through the entire array.
6. Is the goal post now at the first index? If so we can jump to the last index, if not, we cannot jump to it.

Time complexity ?

Recap

Greedy Approach:

1.We're going to initialize a goal post, which is initially the last index.
2.We're going to setup a For Loop that iterates over the array backwards.
3.We ask each element 'Can you jump to or over the goal post from here?' if so, we set the goal post to that element.
4.If not, we continue looping.
5.We repeat this logic until we've been through the entire array.
6.Is the goal post now at the first index? If so we can jump to the last index, if not, we cannot jump to it.

Time complexity : O(n)

LeetCode

Description | Editorial | Solutions (4.5K) | Submissions

45. Jump Game II

Medium | 12.1K | 422

Companies

You are given a **0-indexed** array of integers `nums` of length `n`. You are initially positioned at `nums[0]`.

Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at `nums[i]`, you can jump to any `nums[i + j]` where:

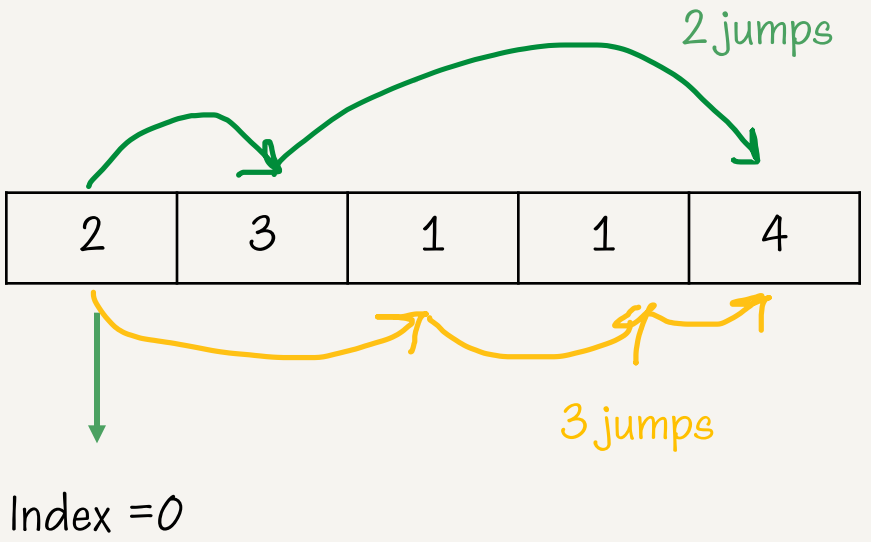
- `0 <= j <= nums[i]` and
- `i + j < n`

Return *the minimum number of jumps to reach* `nums[n - 1]`. The test cases are generated such that you can reach `nums[n - 1]`.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the
last index is 2. Jump 1 step from index 0 to 1, then 3
steps to the last index.
```

Each element → maximum possible jump

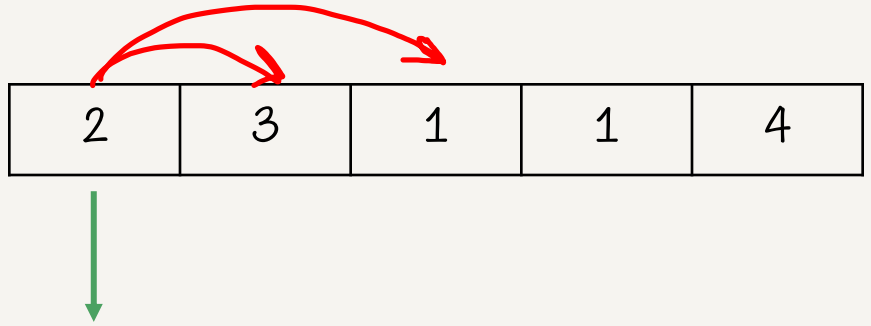


What is the minimum number of jumps to reach the last index? 2 jumps

Return : # of jumps

What can be the greedy choice? Let's figure out

What are the choices for jump form index 0 ?

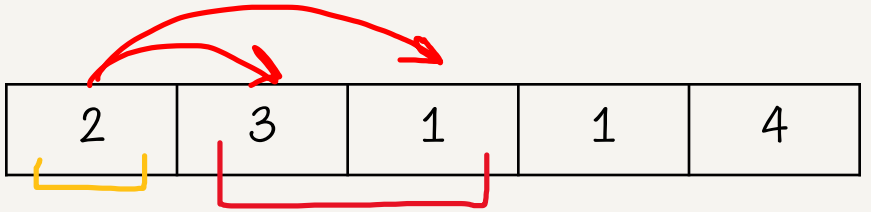


represents the maximum jump length and not the definite jump length

DP → $O(n^2)$
Greedy → $O(n)$

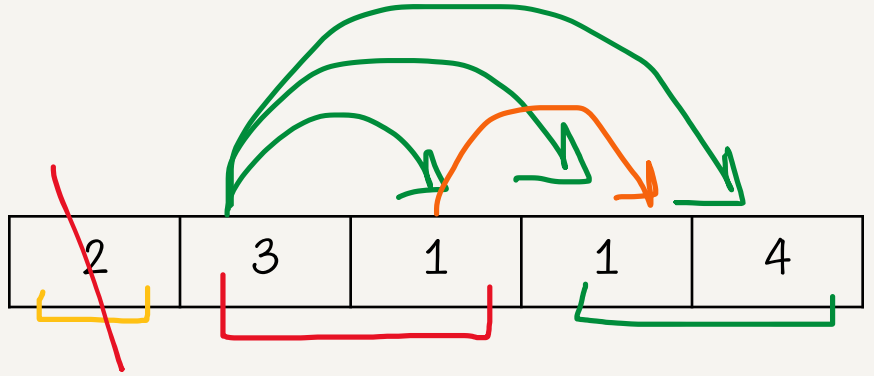
What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$



What can be the greedy choice? Let's figure out

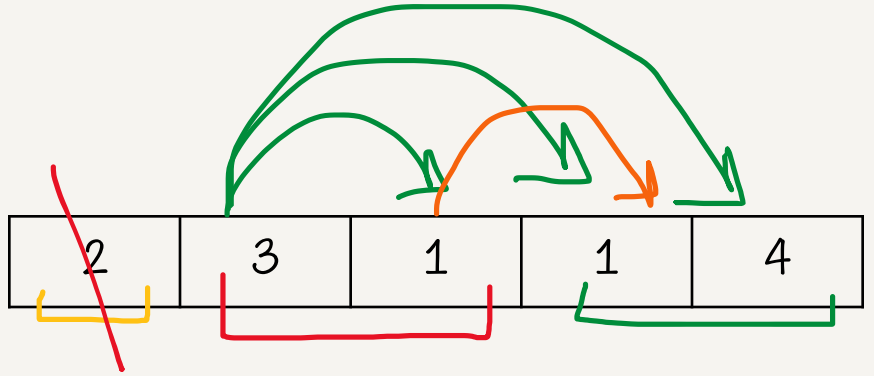
DP → $O(n^2)$
Greedy → $O(n)$



What are the next possible ones?

What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

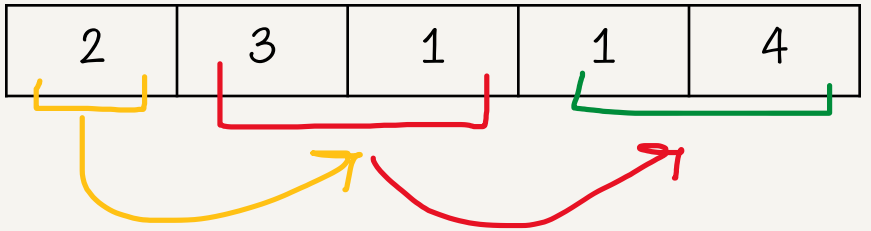


From red part
we can reach to
last index

What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

From each part part we can reach to the next part → color coding

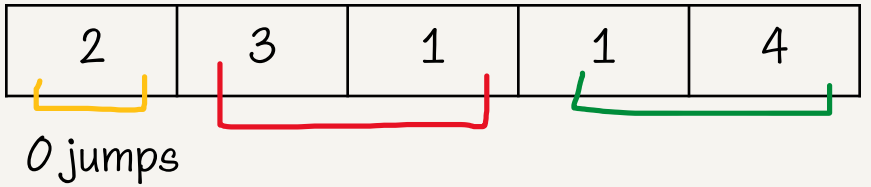


Each part → minimum # of jumps to reach that part

What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

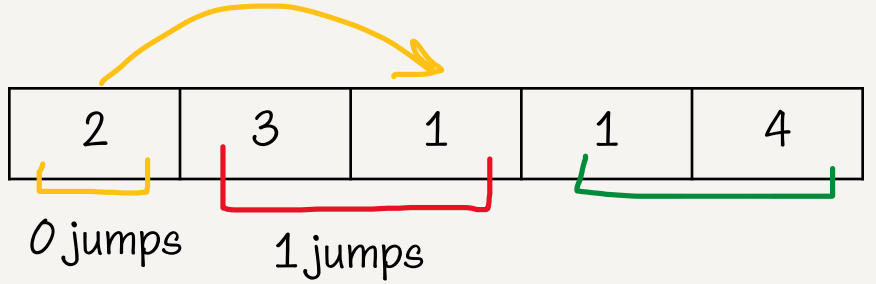
Each part → minimum # of jumps to reach that part



What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

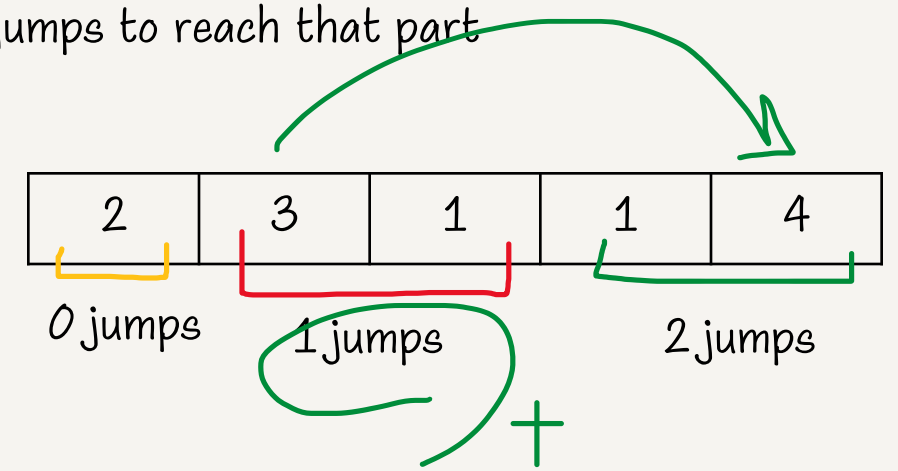
Each part → minimum # of jumps to reach that part



What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

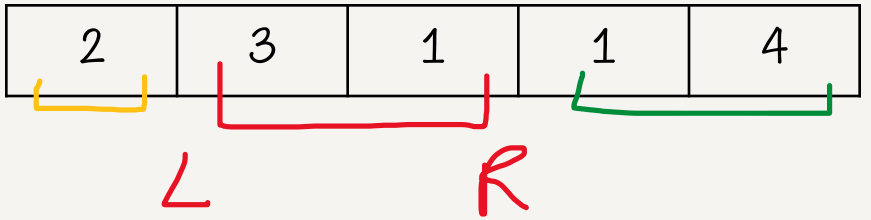
Each part → minimum # of jumps to reach that part



What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

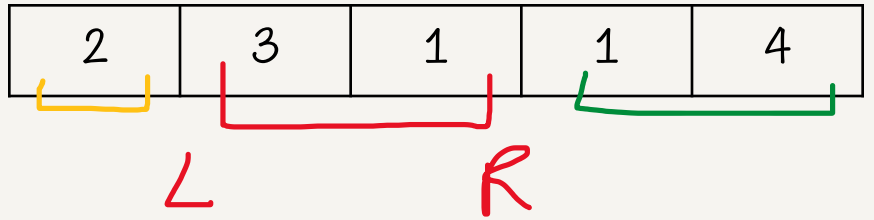
Define a window for each section : left & right ; initialize at 0 → [0,0]
What is our goal ? Reach to the most right position : last index



What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

Define a window for each section : left & right ; initialize at 0 → [0,0]
What is our goal ? Reach to the most right position : last index

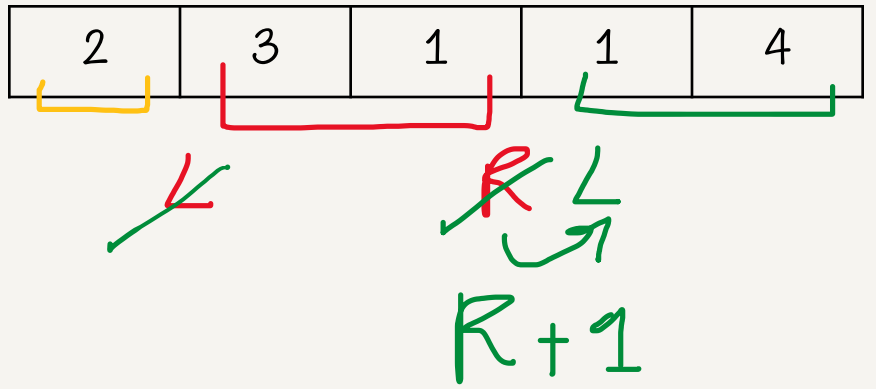


How to update the left & right for the next portion?

What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

Define a window for each section : left & right ; initialize at 0 → [0,0]
What is our goal ? Reach to the most right position : last index

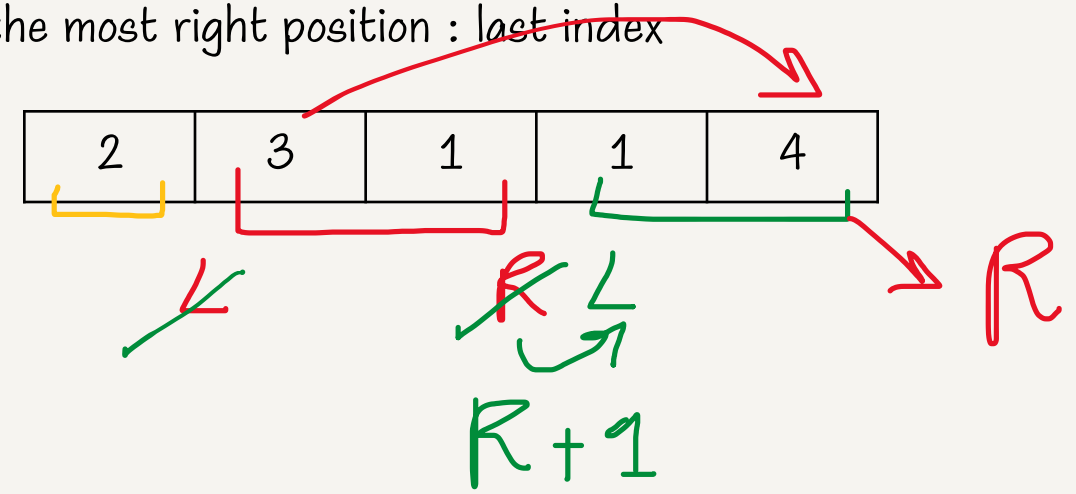


How to update the left & right for the next portion?
Left → adjacent to right → new left = right+1

What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

Define a window for each section : left & right ; initialize at 0 → [0,0]
What is our goal ? Reach to the most right position : ~~last index~~

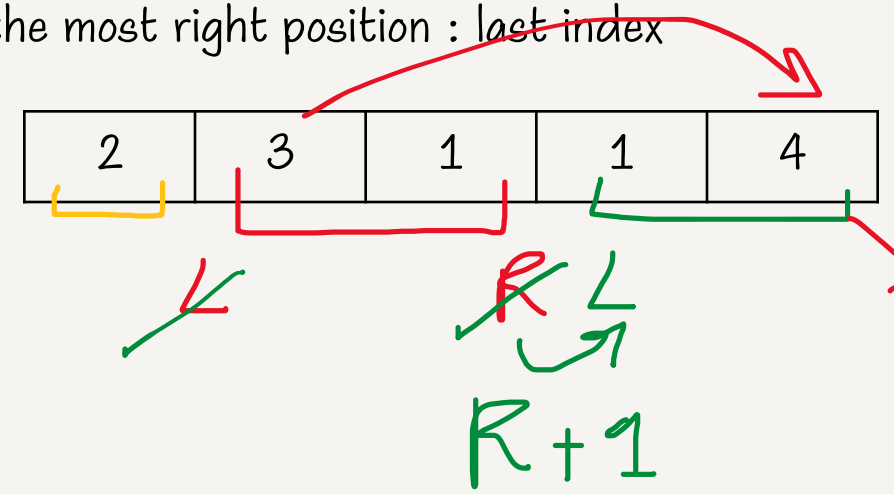


How to update the left & right for the next portion?
Left → adjacent to right → new left = right+1
How about new right ? Who can reach the furthest position?

What can be the greedy choice? Let's figure out

DP → $O(n^2)$
Greedy → $O(n)$

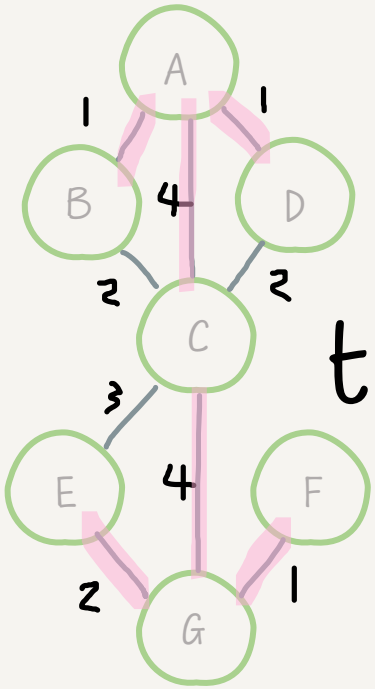
Define a window for each section : left & right ; initialize at 0 → [0,0]
What is our goal ? Reach to the most right position : ~~last index~~



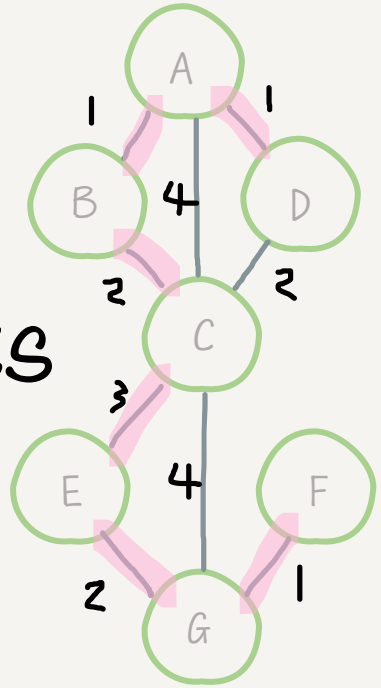
= furthest position we can jump from the previous part

How to update the left & right for the next portion?
Left → adjacent to right → new left = right+1
How about new right ? Who can reach the furthest position?

Min Cost to Connect All Points



X



✓

1584. Min Cost to Connect All Points

Hint

Medium 3.5K 84

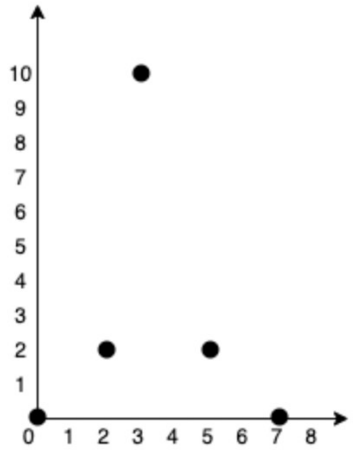
Companies

You are given an array `points` representing integer coordinates of some points on a 2D-plane, where `points[i] = [xi, yi]`.

The cost of connecting two points `[xi, yi]` and `[xj, yj]` is the **manhattan distance** between them: `|xi - xj| + |yi - yj|`, where `|val|` denotes the absolute value of `val`.

Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

Example 1:




```
Input: points = [[0,0],[2,2],[3,10],[5,2],[7,0]]
Output: 20
Explanation:
```

1584. Min Cost to Connect All Points

Hint

Medium 3.5K 84

Companies

You are given an array `points` representing integer coordinates of some points on a 2D-plane, where `points[i] = [x_i, y_i]`.

The cost of connecting two points `[x_i, y_i]` and `[x_j, y_j]` is the **manhattan distance** between them: `|x_i - x_j| + |y_i - y_j|`, where `|val|` denotes the absolute value of `val`.

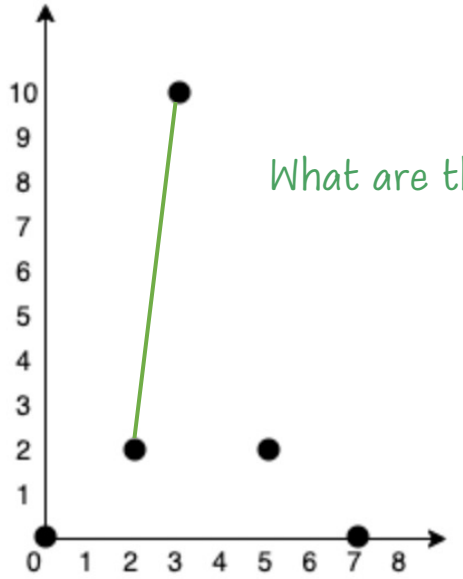
Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

Span all the points with minimum cost

No cycle → tree

Minimum spanning tree ?

Example 1:



What are the edges?

1584. Min Cost to Connect All Points

Hint

Medium 3.5K 84

Companies

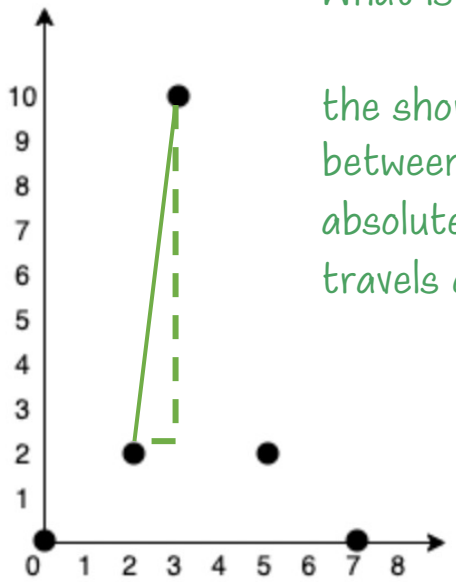
You are given an array `points` representing integer coordinates of some points on a 2D-plane, where `points[i] = [xi, yi]`.

The cost of connecting two points `[xi, yi]` and `[xj, yj]` is the **manhattan distance** between them: `|xi - xj| + |yi - yj|`, where `|val|` denotes the absolute value of `val`.

Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

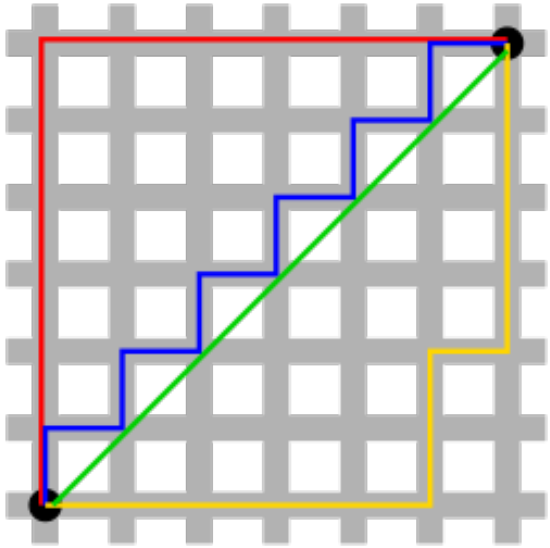
Minimum spanning tree

Example 1:



What is Manhattan distance?

the shortest path a taxi travels between two points is the sum of the absolute values of distances that it travels on avenues and on streets.



1584. Min Cost to Connect All Points

Hint

Medium 3.5K 84

Companies

You are given an array `points` representing integer coordinates of some points on a 2D-plane, where `points[i] = [xi, yi]`.

Define the edges

The cost of connecting two points `[xi, yi]` and `[xj, yj]` is the **manhattan distance** between them: `|xi - xj| + |yi - yj|`, where `|val|` denotes the absolute value of `val`.

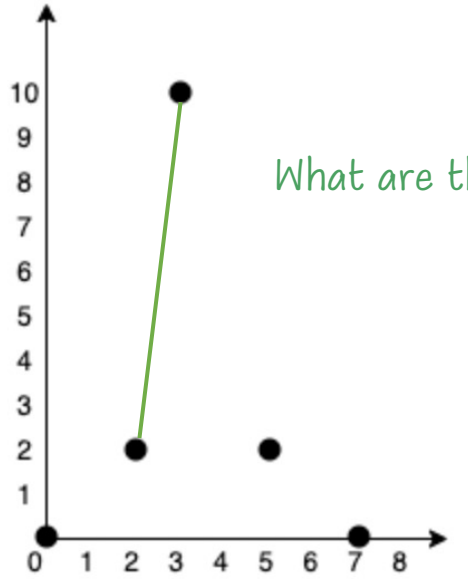
Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

Span all the points with minimum cost

No cycle → tree

Minimum spanning tree ?

Example 1:



What are the edges?

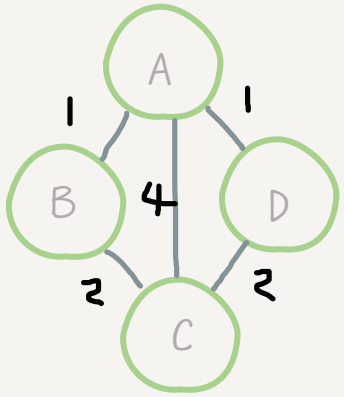
What are the steps ?
- Create edges
- Apply Prim's Algorithm

How do we actually create the edges?
Use Adjacency list

What is Adjacency list ?
“ For every vertex, its adjacent vertices are stored. In the case of a weighted graph, the edge weights are stored along with the vertices.”

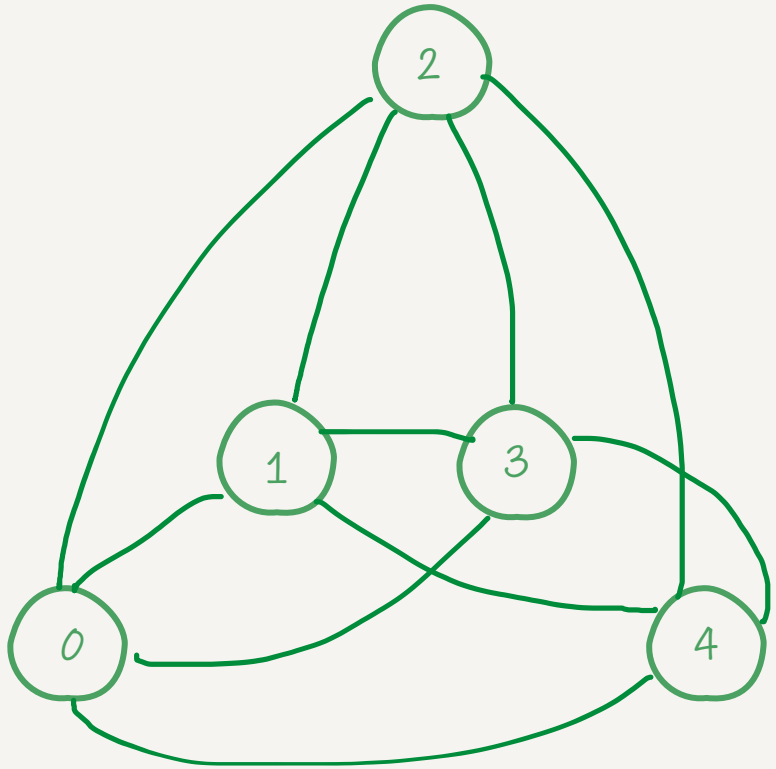
What is Adjacency list ?

" For every vertex, its adjacent vertices are stored. In the case of a weighted graph, the edge weights are stored along with the vertices."



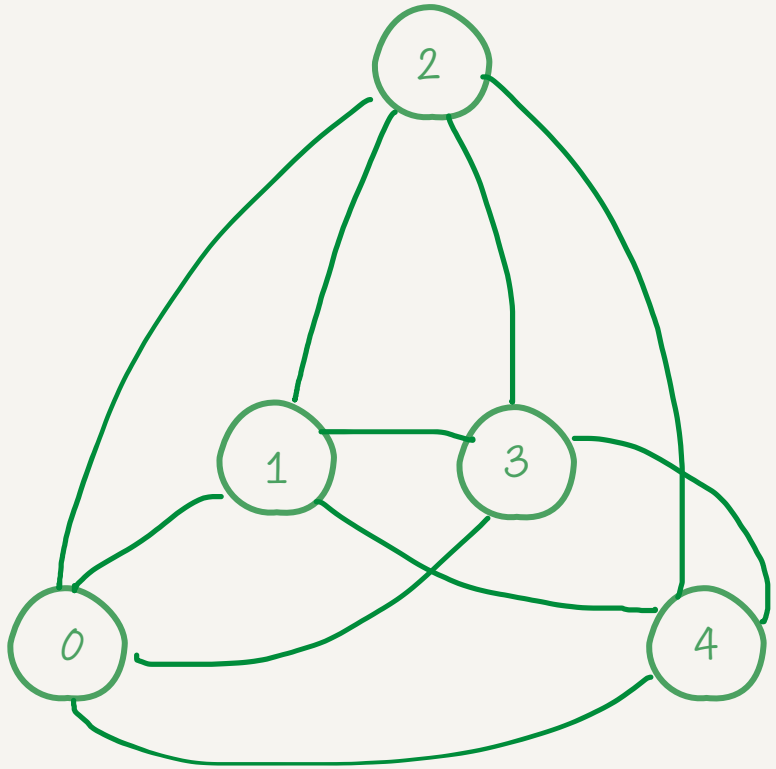
Adjacency list

```
{ 'A' :  [['B',1],[C,4],[D,1]] , 'B' : [['A',1],[C,2]] , 'C' : [['B',1],[A,4],[D,1]], , 'D' : [['A',1],[C,2]] }
```



Let's apply Prim's algorithm on a sample graph :

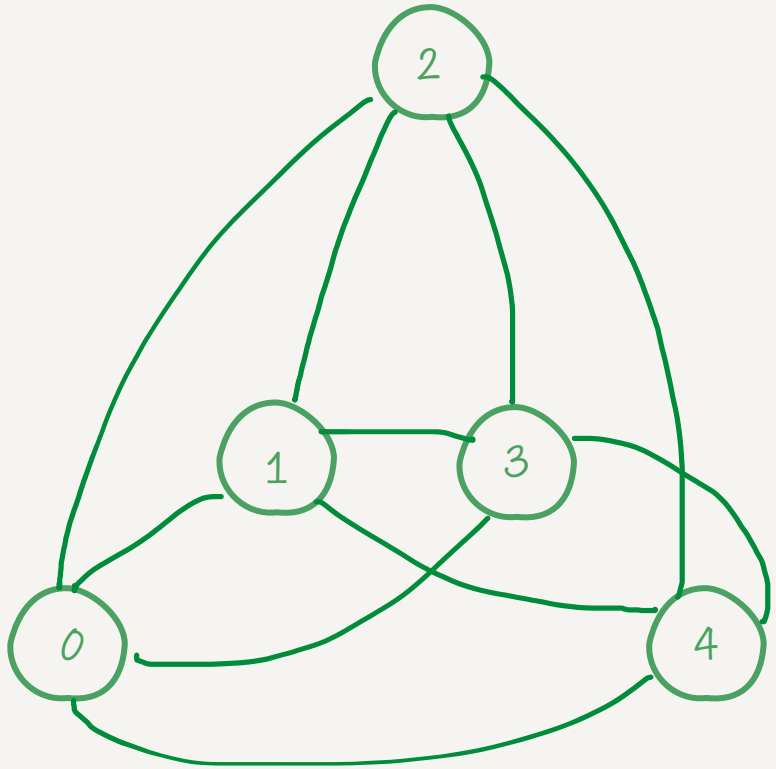
a graph with 5 nodes
All nodes are connected
How many edges should we have for spanning tree ?
|E| = |V| -1 → 5-1 = 4 edges; no cycle



Strat with any single node, perform BFS

For every node :

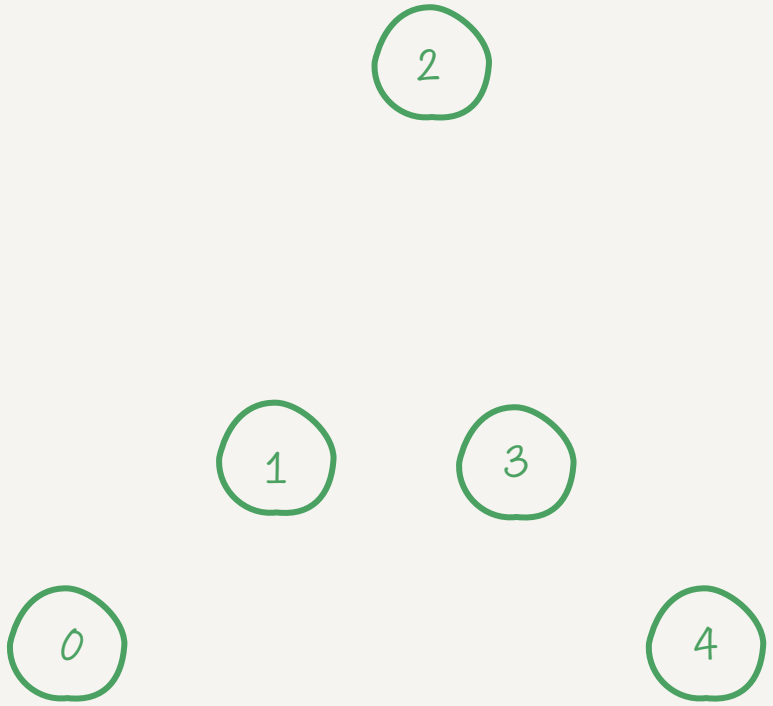
- visit (hash set data structure) : avoid cycle
- frontier (min-heap data structure : keep track of every possible nodes → pop the last



Strat with any single node, perform BFS

For every node :
- visit (hash set data structure) : avoid cycle
- frontier (min-heap data structure : store every possible edges → pop the last

When do we stop ?
Length of visit = # of initial nodes



Visit (hash set)

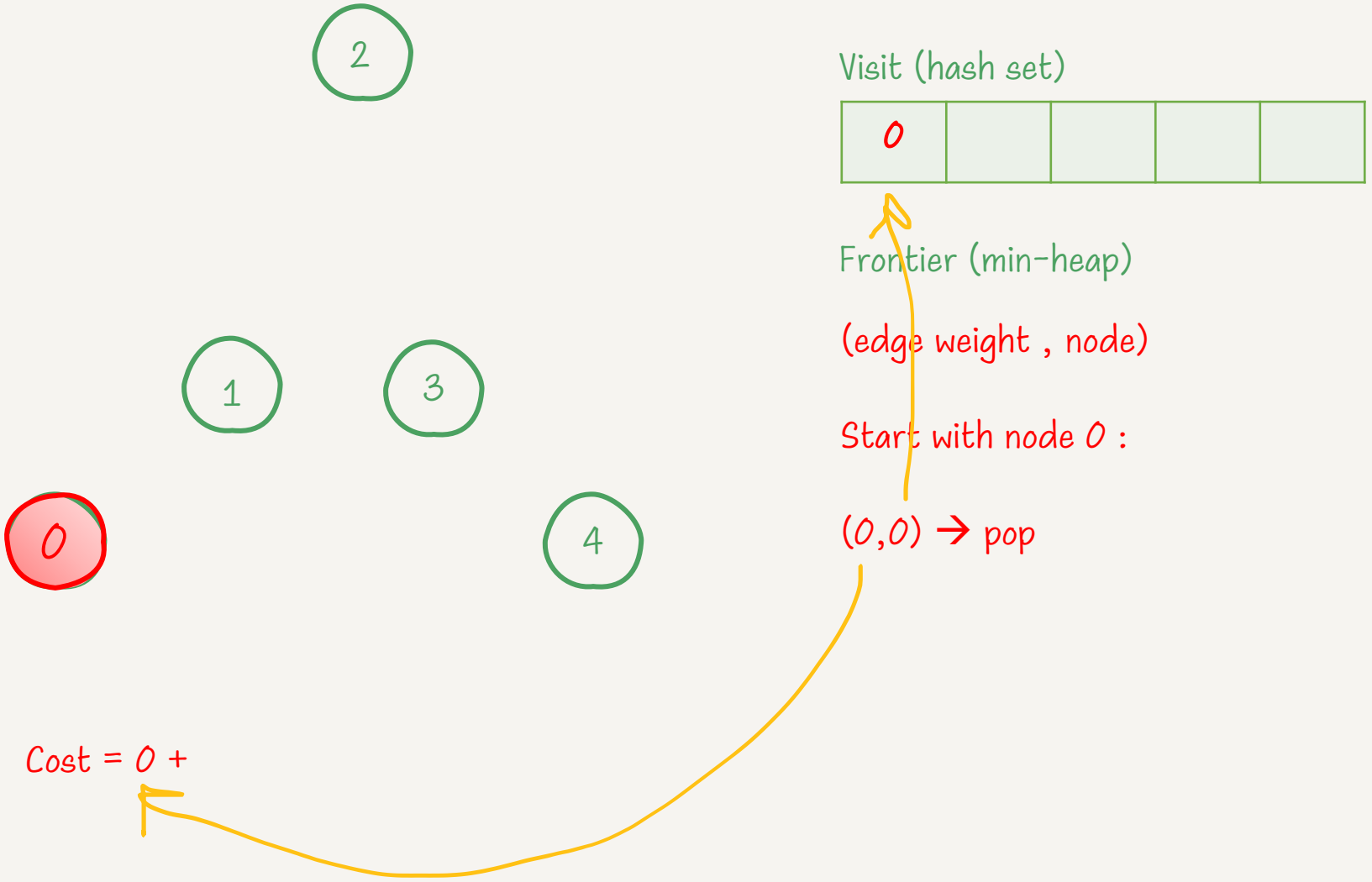
Frontier (min-heap)

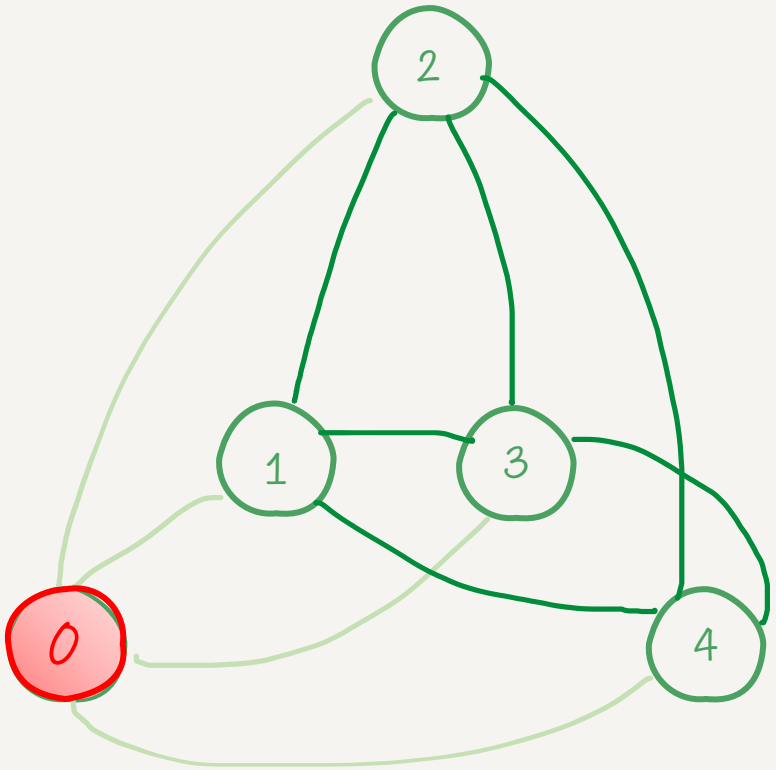
(edge weight , node)

Start with node 0 :

(0,0)

Cost =


2
1
3
0
4
Visit (hash set)
0
Frontier (min-heap)
(edge weight , node)
Start with node 0 :
(0,0) → pop
Cost = 0 +



Visit (hash set)

0				

Frontier (min-heap)

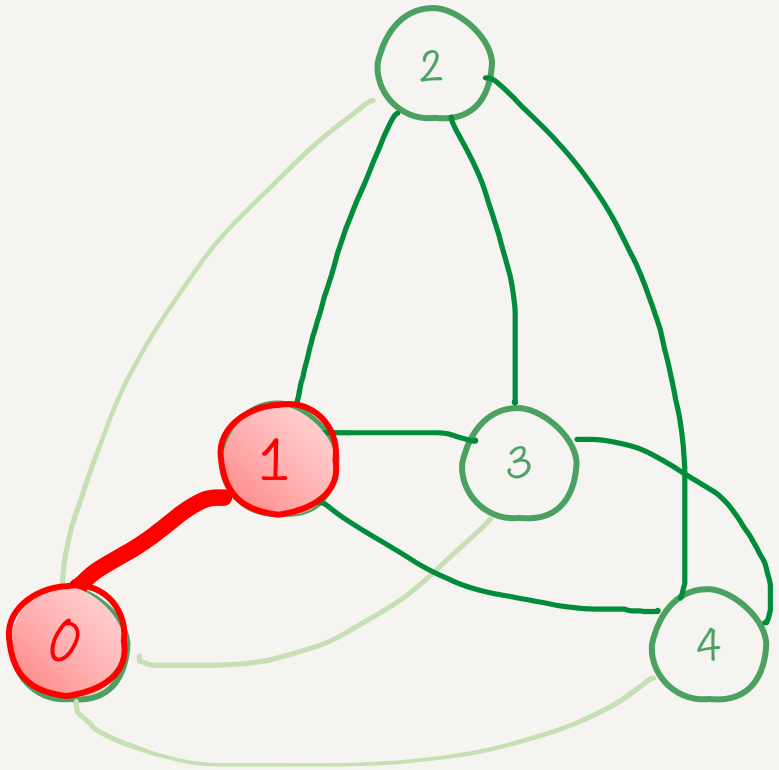
(edge weight , node)

Start with node 0 :

(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

Cost = 0 +



Visit (hash set)

0	1			

Frontier (min-heap)

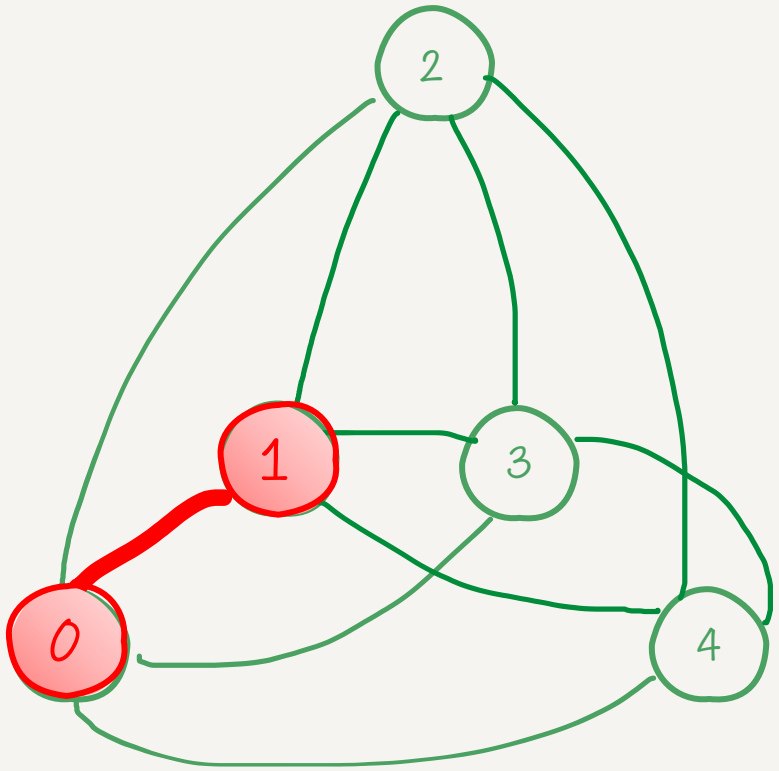
(edge weight , node)

Start with node 0 :

(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

Cost = 0 + 4



Cost = 0 + 4

Visit (hash set)

0	1			

Frontier (min-heap)

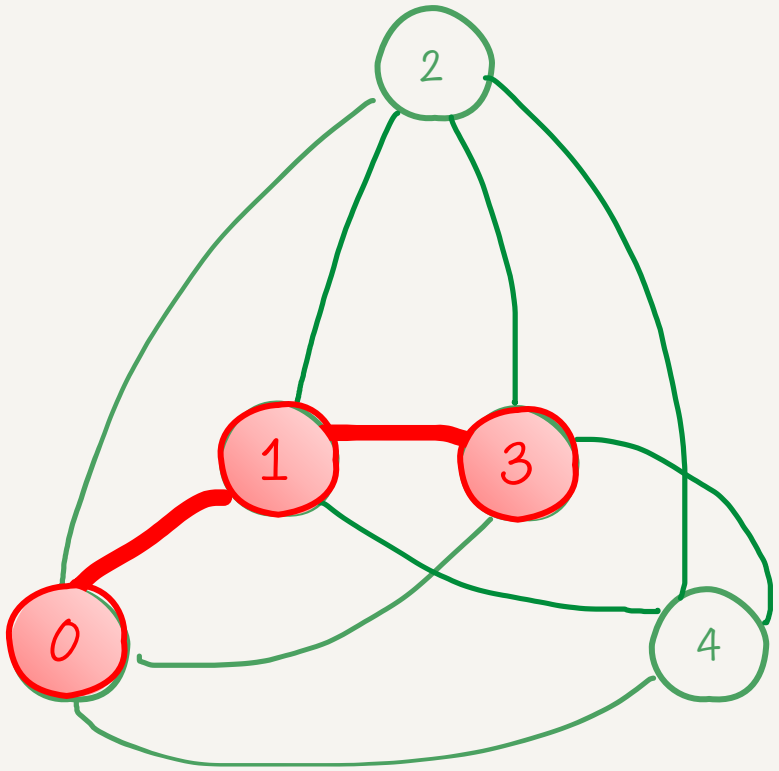
(edge weight , node)

Start with node 0 :

(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)



Visit (hash set)

0	1	3		

Frontier (min-heap)

(edge weight , node)

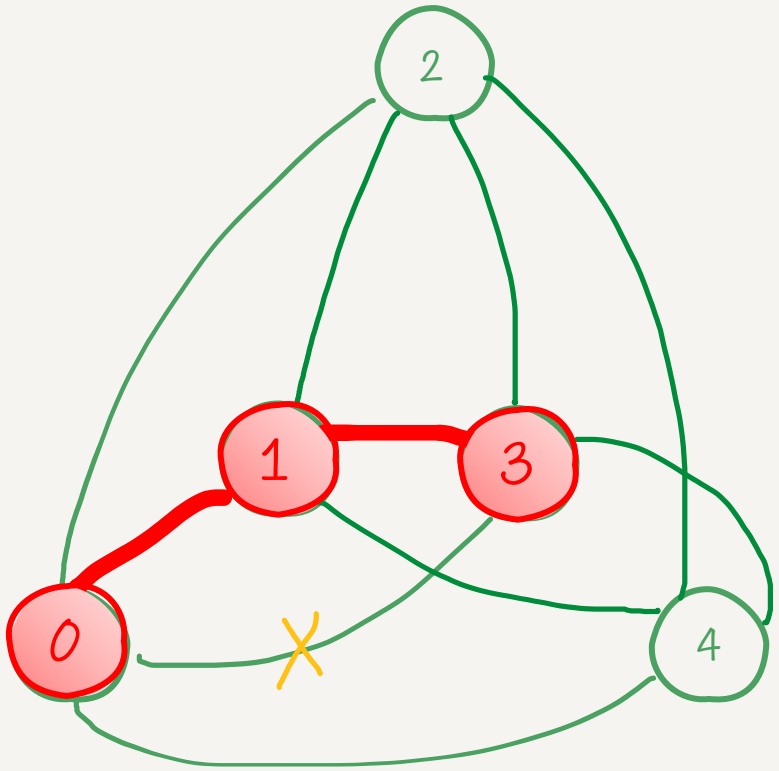
Start with node 0 :

(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)

Cost = 0 + 4 + 3



Cost = 0 + 4 + 3

Visit (hash set)

0	1	3		

Frontier (min-heap)

(edge weight , node)

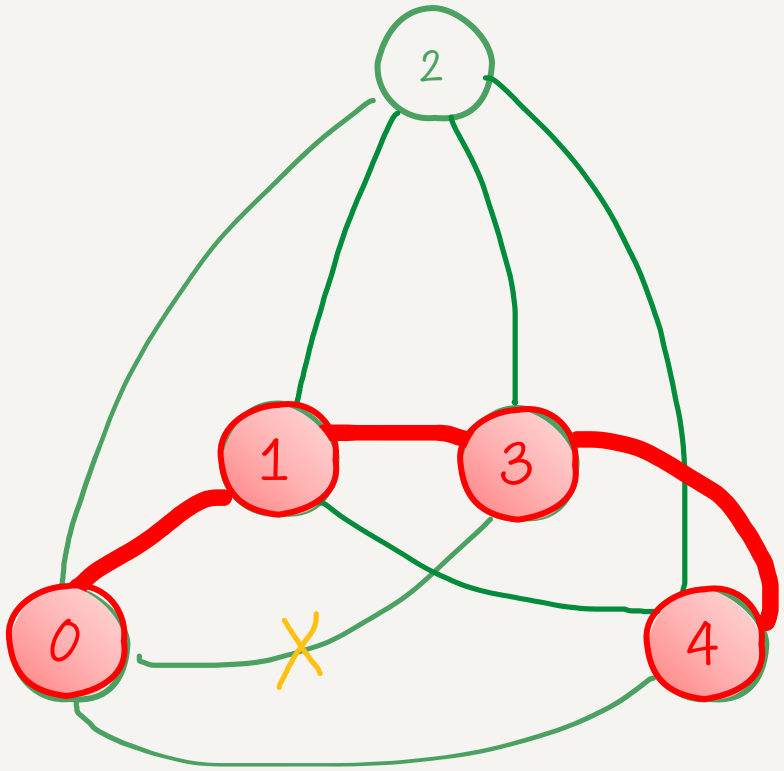
Start with node 0 :

(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)

(-,2), (-,4)



Visit (hash set)

0	1	3	4	

Frontier (min-heap)

(edge weight , node)

Start with node 0 :

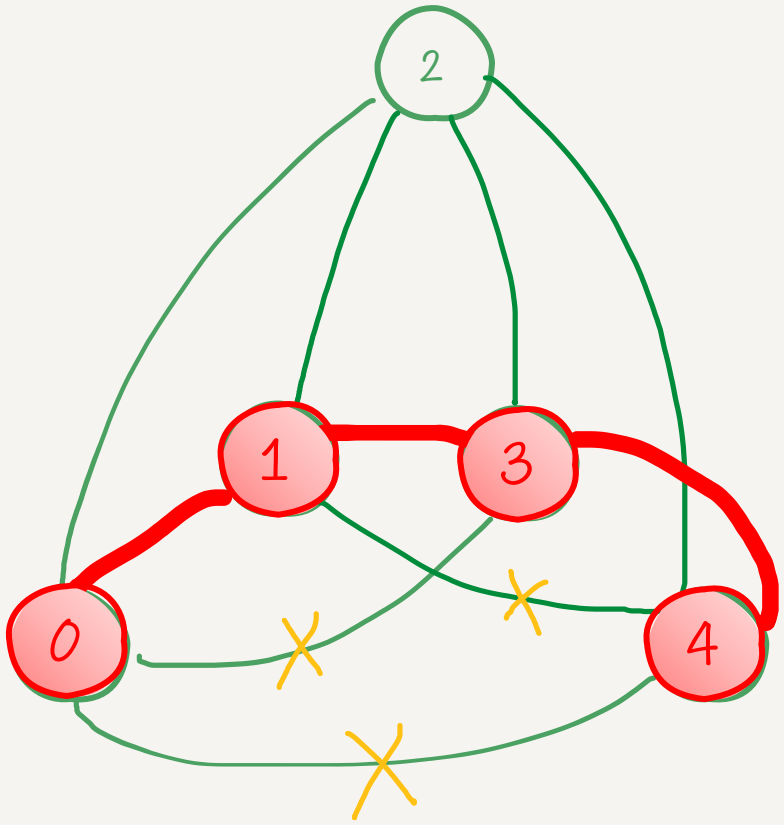
(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)

(-,2), (-,4)

Cost = 0 + 4 + 3 + 4



Cost = 0 + 4 + 3 + 4

Visit (hash set)

0	1	3	4	

Frontier (min-heap)

(edge weight , node)

Start with node 0 :

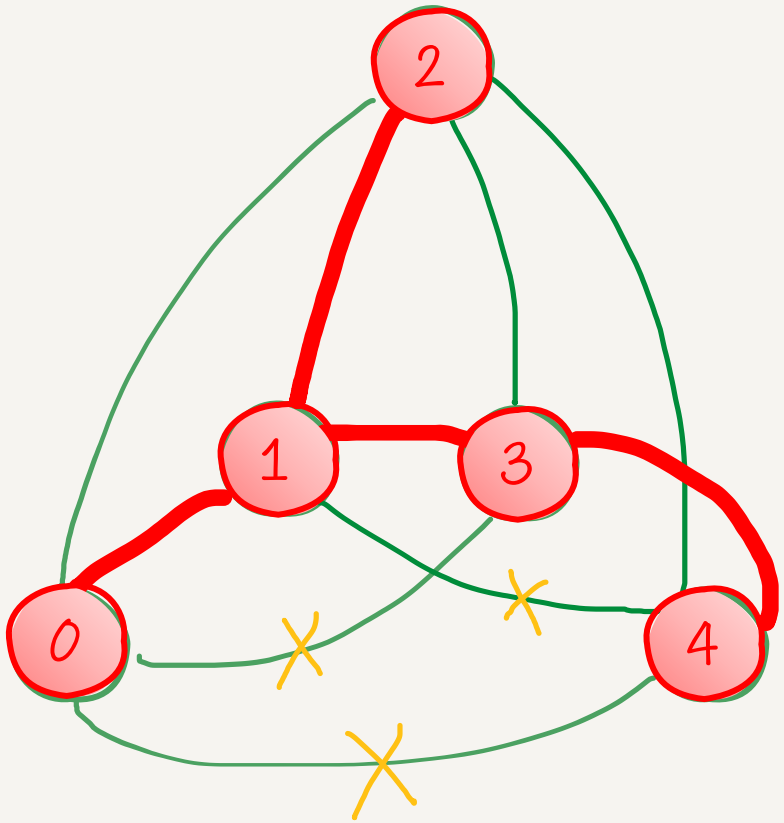
(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)

(-,2), (-,4)

(-,2)



Cost = 0 + 4 + 3 + 4 + 9

Visit (hash set)

0	1	3	4	2

Frontier (min-heap)

(edge weight , node)

Start with node 0 :

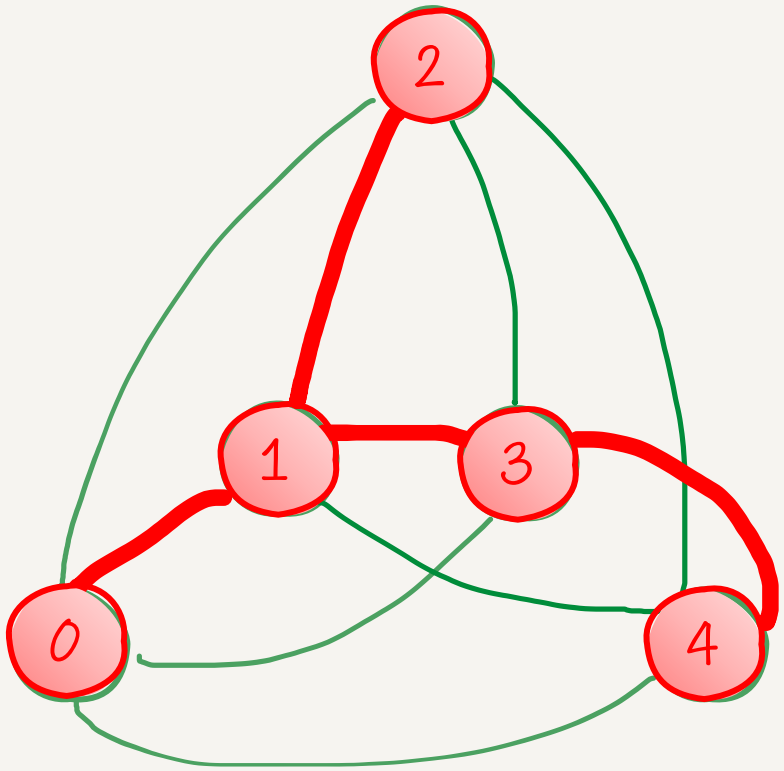
(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

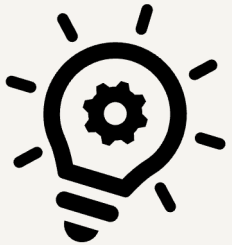
(-,2) , (-,3) , (-,4)

(-,2), (-,4)

(-,2)



Cost = 0 + 4 + 3 + 4 + 9



Visit (hash set)

0	1	3	4	2

When do we stop ?
Length of visit =
# of initial nodes

Frontier (min-heap)

(edge weight , node)

Start with node 0 :

(0,0) → pop

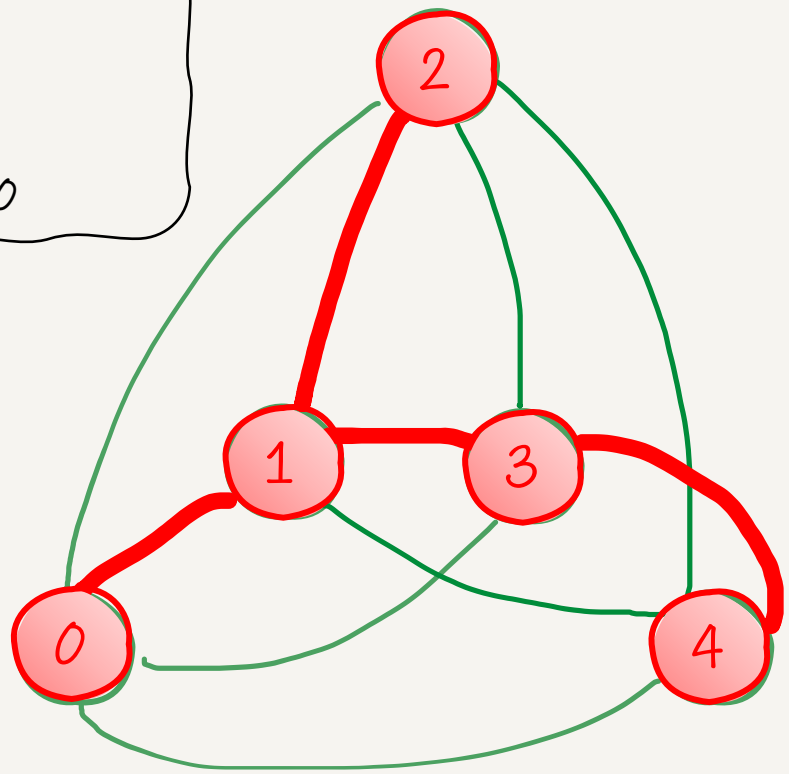
(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)

(-,2), (-,4)

(-,2)

Minimum spanning tree:
V = 5
E = 4
cost = 20



Cost = 0 + 4 + 3 + 4 + 9 = 20

Visit (hash set)

0	1	3	4	2

When do we stop ?
Length of visit =
# of initial nodes

Frontier (min-heap)

(edge weight , node)

Start with node 0 :

(0,0) → pop

(-,1) , (-,2) , (-,3) , (-,4)

(-,2) , (-,3) , (-,4)

(-,2), (-,4)

(-,2)