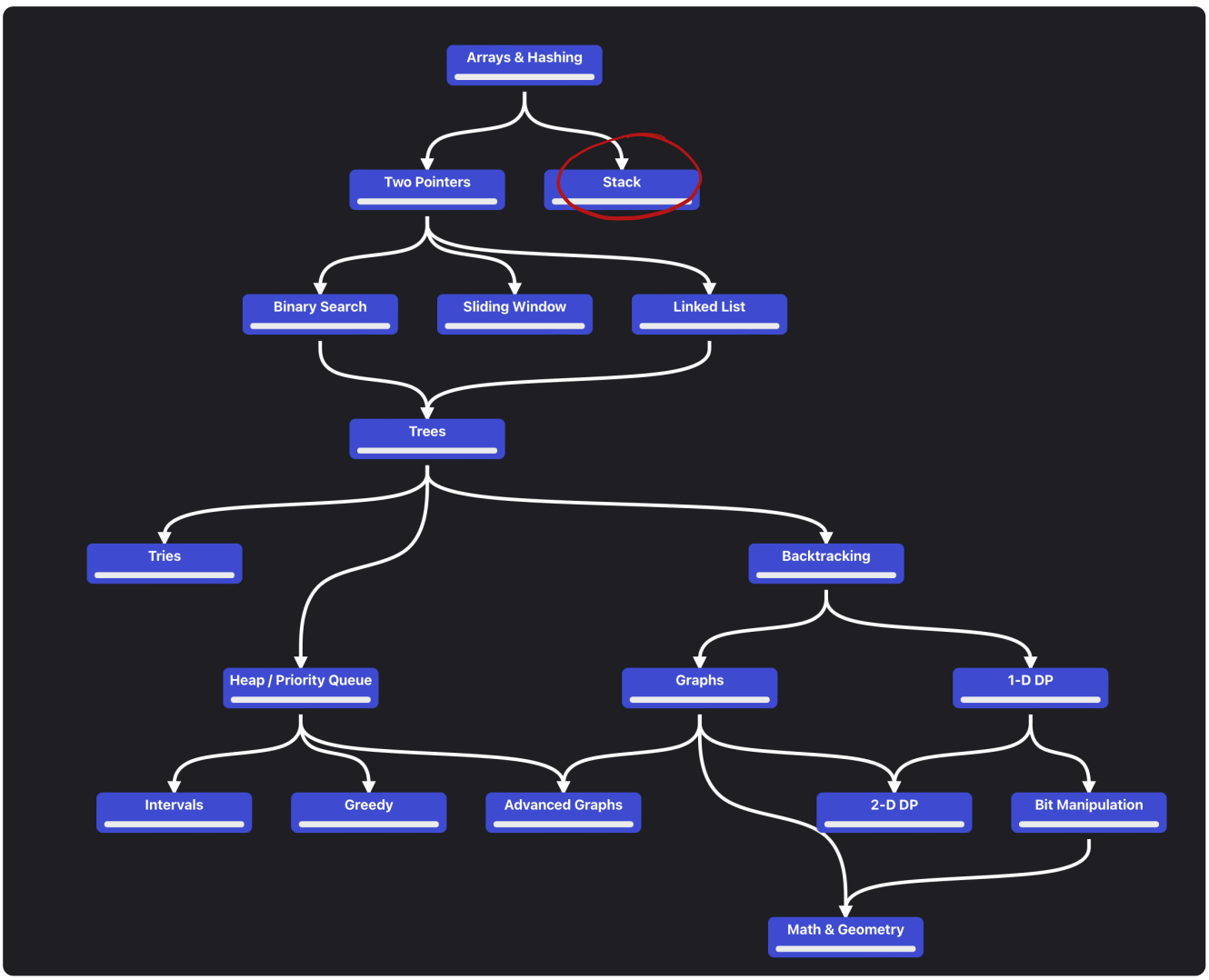
# Stack


Arrays & Hashing
Two Pointers
Stack
Binary Search
Sliding Window
Linked List
Trees
Tries
Backtracking
Heap / Priority Queue
Graphs
1-D DP
Intervals
Greedy
Advanced Graphs
2-D DP
Bit Manipulation
Math & Geometry

## 20. Valid Parentheses

Hint

Easy 23.1K 1.6K

Companies

Given a string `s` containing just the characters `'('`, `')'`, `'{'`, `'}'`, `'['` and `']'`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

**Example 1:**

> **Input:** s = "()"
> **Output:** true

**Example 2:**

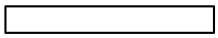
> **Input:** s = "()[]{}"
> **Output:** true

**Example 3:**

> **Input:** s = "(]"
> **Output:** false

# 20. Valid Parenthesis

( ) [ ] { }



```cpp
bool isValid(string s) {

    int size = s.size();

    stack<char> my_stack;

    for (int i = 0; i < size; i++) {

        char curr = s[i];
        if (curr == '(' || curr == '{' || curr == '[' ) {
            my_stack.push(curr);
        } else {   // curr is a closing symbol

            if (my_stack.size() == 0) {
                return false;
            }

            char top = my_stack.top();
            my_stack.pop();

            if (curr == ')' && top != '(') {  // if current is a closing symbol and the char just popped is not it's correspondent
                return false;                 // opening symbol -> INVALID
            } else if (curr == '}' && top != '{') {
                return false;
            } else if (curr == ']' && top != '[') {
                return false;
            }
        }
    }

    if (my_stack.size() != 0) {
        return false;
    }

    return true;
}
```

# 155. Min Stack

Medium 13.5K 822

Companies

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with `O(1)` time complexity for each function.

**Example 1:**

**Input**
```
["MinStack","push","push","push","getMin","pop"
,"top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]
```

**Output**
```
[null,null,null,null,-3,null,0,-2]
```

**Explanation**
```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

```cpp
class MinStack {
public:

    int size;
    Node* min;
    Node* head;

    MinStack() {
        head = nullptr;
        min = nullptr;
        size = 0;
    }

    void push(int val) {

            Node* new_node = new Node(val, head);
            head = new_node;

            if (size == 0 || min->get_data() >= val) {
                new_node->set_next_min(min);
                min = new_node;
            }

            size++;
        }

    void pop() {
        if (size == 0) {
            throw std::runtime_error("Stack is empty");
        }

        if (head == min) {
            min = head->get_next_min();
        }

        Node* temp = head->get_next();
        delete head;
        head = temp;

        size--;
    }


    int top() const {
        if (size == 0) {
            throw std::runtime_error("Stack is empty");
        }
        return head->get_data();
    }

    int getMin() const {
        if (size == 0) {
            throw std::runtime_error("Stack is empty");
        }
        return min->get_data();
    }
};
```

```cpp
class Node {
    int data;
    Node* next;
    Node* next_min;

public:
    Node(int data, Node* next = nullptr, Node* next_min = nullptr)
        : data(data), next(next), next_min(next_min) {}

    int get_data() const {
        return data;
    }

    Node* get_next() const {
        return next;
    }

    Node* get_next_min() const {
        return next_min;
    }

    void set_next_min(Node* next_min) {
        this->next_min = next_min;
    }
};
```

# 150. Evaluate Reverse Polish Notation

Medium 6.9K 1K

Companies

You are given an array of strings `tokens` that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return *an integer that represents the value of the expression*.

**Note** that:

- The valid operators are `'+'`, `'-'`, `'*'`, and `'/'`.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

**Example 1:**

> **Input:** tokens = ["2","1","+","3","*"]
> **Output:** 9
> **Explanation:** ((2 + 1) * 3) = 9

**Example 2:**

> **Input:** tokens = ["4","13","5","/","+"]
> **Output:** 6
> **Explanation:** (4 + (13 / 5)) = 6

**Example 3:**

> **Input:** tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
> **Output:** 22
> **Explanation:** ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
> = ((10 * (6 / (12 * -11))) + 17) + 5
> = ((10 * (6 / -132)) + 17) + 5
> = ((10 * 0) + 17) + 5
> = (0 + 17) + 5
> = 17 + 5
> = 22

# 150. Evaluate Reverse Polish Notation

```cpp
int evalRPN(vector<string>& tokens) {
    int size = tokens.size();

    stack<int> my_stack;

    for (int i = 0; i < size; i++) {

        string c = tokens[i];

        if (c == "+" || c == "-" || c == "*" || c == "/") {

            int op1 = my_stack.top();  // Pop operands
            my_stack.pop();
            int op2 = my_stack.top();
            my_stack.pop();

            // Determine operator string to perform actual operation
            if (c == "+") {
                my_stack.push(op2 + op1);
            }

            if (c == "-") {
                my_stack.push(op2 - op1);
            }

            if (c == "*") {
                my_stack.push(op2 * op1);
            }

            if (c == "/") {
                my_stack.push(op2 / op1);
            }

        } else {  // c is a number
            my_stack.push(stoi(tokens[i]));
        }

    }

    int result = my_stack.top();

    return result;
}
```

```cpp
int evalRPN(vector<string>& tokens) {
    std::stack<int> my_stack;

    // Use a map for operators to simplify the operation logic
    std::unordered_map<string, std::function<int(int, int)>> ops = {
        {"+", [](int a, int b) { return a + b; }},
        {"-", [](int a, int b) { return a - b; }},
        {"*", [](int a, int b) { return a * b; }},
        {"/", [](int a, int b) { return a / b; }}
    };

    for (const string& token : tokens) {
        if (ops.find(token) != ops.end()) {
            int op1 = my_stack.top();
            my_stack.pop();
            int op2 = my_stack.top();
            my_stack.pop();

            // Perform the operation and push the result back on the stack
            my_stack.push(ops[token](op2, op1));
        } else {
            // If it's a number, simply push it onto the stack
            my_stack.push(stoi(token));
        }
    }

    return my_stack.top();
}
```

# 22. Generate Parentheses

Medium 20.3K 855

Companies

Given `n` pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Example 1:**

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

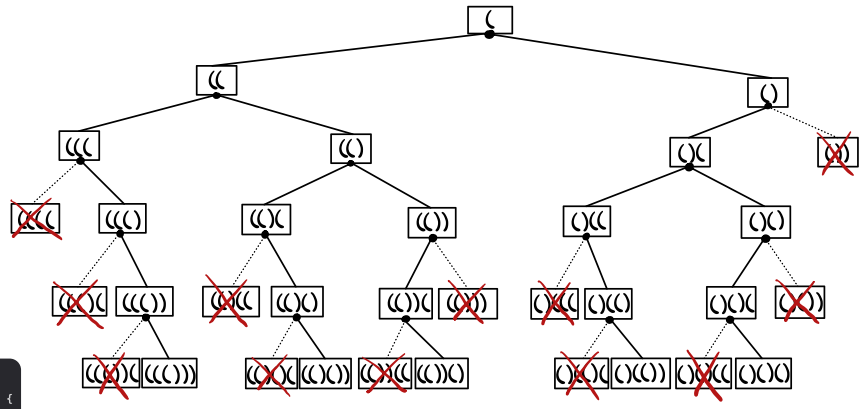
**Example 2:**

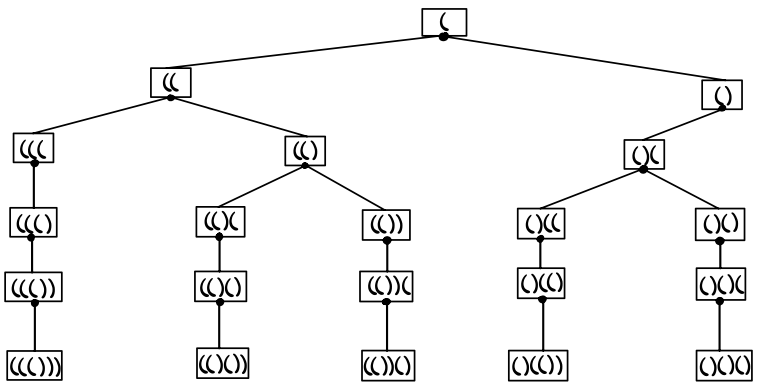
```
Input: n = 1
Output: ["()"]
```

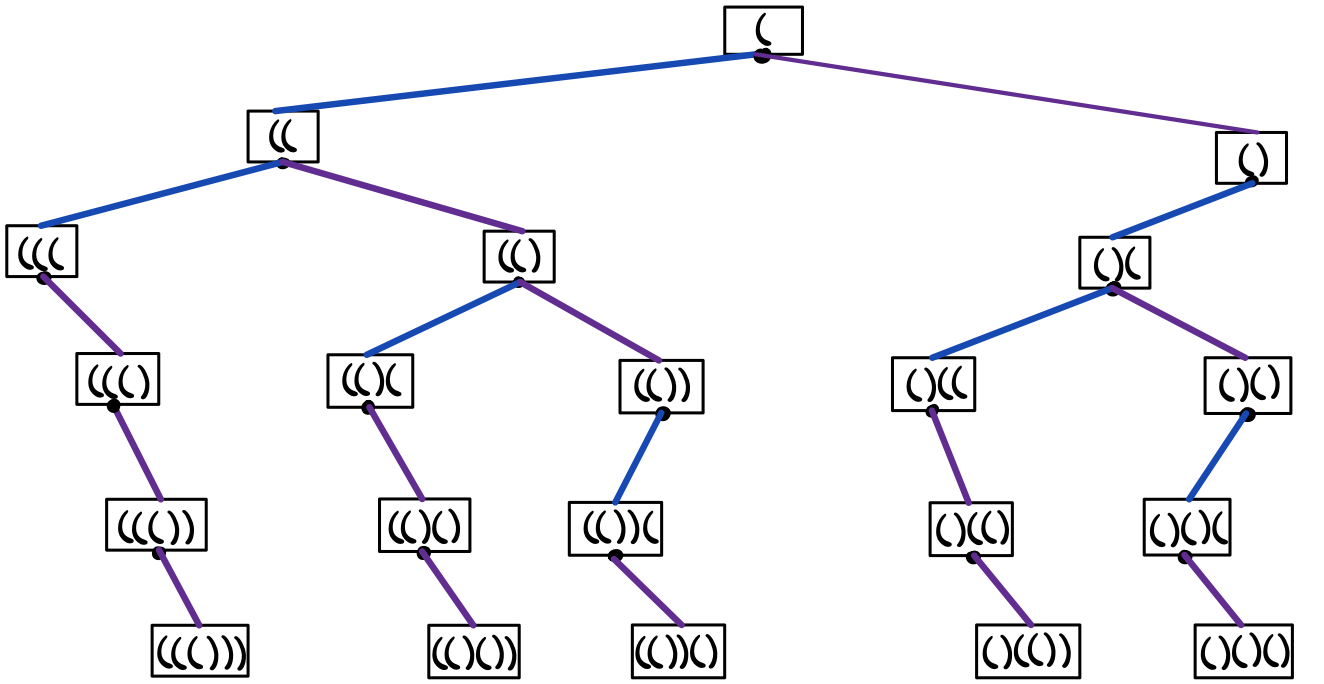
**Constraints:**

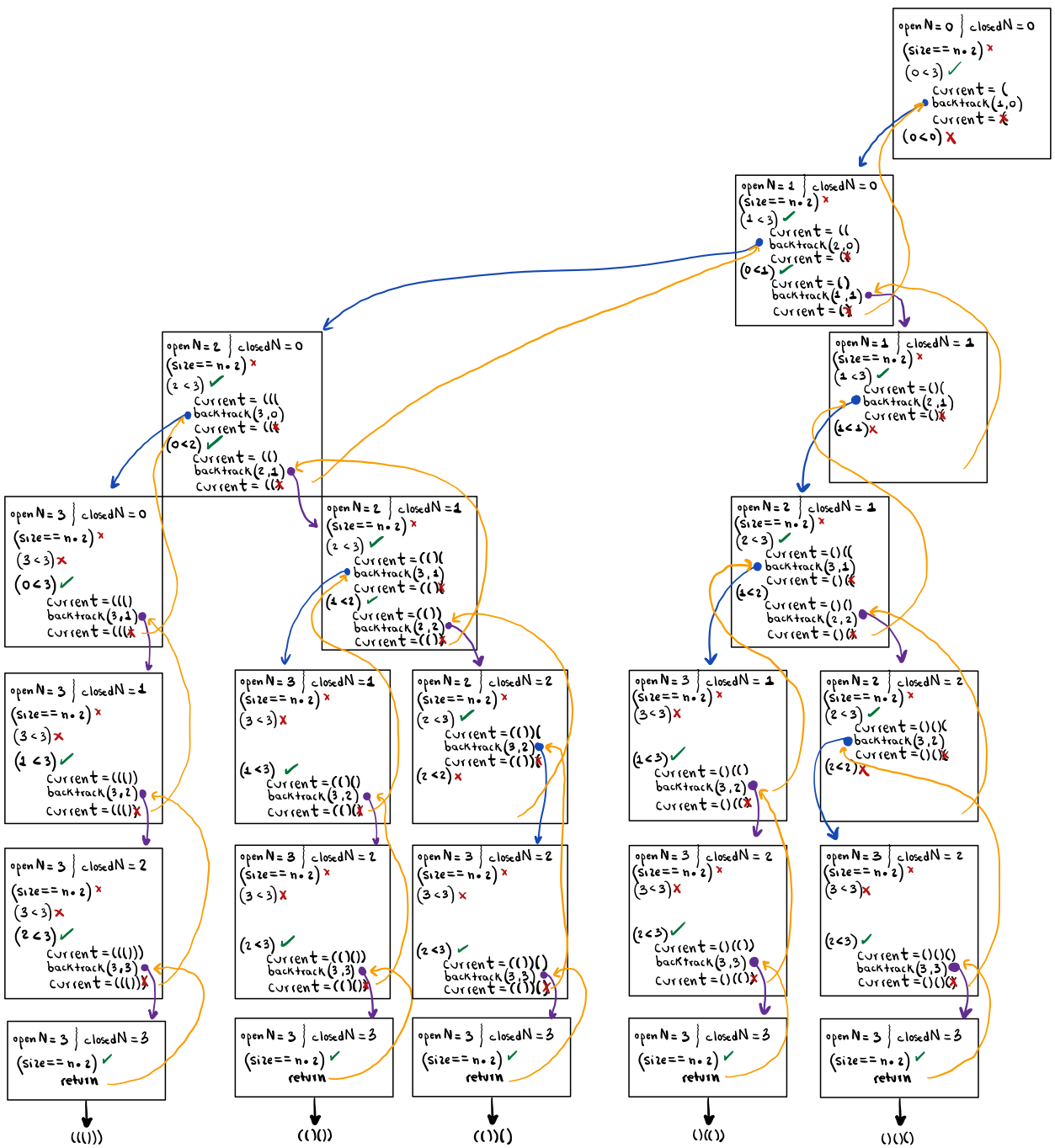
- `1 <= n <= 8`

```cpp
class Solution {
private:
    void backtrack(int openN, int closedN, int n, string &current, vector<string> &result) {
        // If the current string is a valid combination, add it to the result
        if (current.size() == n * 2) {
            result.push_back(current);
            return;
        }

        // If we can add an open parenthesis, do it and recurse
        if (openN < n) {
            current.push_back('(');
            backtrack(openN + 1, closedN, n, current, result);
            current.pop_back(); // Backtrack
        }

        // If we can add a close parenthesis, do it and recurse
        if (closedN < openN) {
            current.push_back(')');
            backtrack(openN, closedN + 1, n, current, result);
            current.pop_back(); // Backtrack
        }
    }

public:
    vector<string> generateParenthesis(int n) {
        vector<string> result;
        string current;
        backtrack(0, 0, n, current, result);
        return result;
    }
};
```






(
((
()
(((
(()
()(
((((
((()
((())
()((
()()
(((()))
((()())
((())()
()(())
()()()


openN = 0 | closedN = 0
(size == n * 2) ✗
(0 < 3) ✓
Current = (
backtrack(1,0)
Current = ✗
(0 < 0) ✗
openN = 1 | closedN = 0
(size == n * 2) ✗
(1 < 3) ✓
Current = ((
backtrack(2,0)
Current = (( ✗
(0 < 1) ✓
Current = ()
backtrack(1,1)
Current = ( ✗
openN = 2 | closedN = 0
(size == n * 2) ✗
(2 < 3) ✓
Current = (((
backtrack(3,0)
Current = ((( ✗
(0 < 2) ✓
Current = (()
backtrack(2,1)
Current = (( ✗
openN = 1 | closedN = 1
(size == n * 2) ✗
(1 < 3) ✓
Current = ()(
backtrack(2,1)
Current = ()( ✗
(1 < 1) ✗
openN = 3 | closedN = 0
(size == n * 2) ✗
(3 < 3) ✗
(0 < 3) ✓
Current = (((()
backtrack(3,1)
Current = ((( ✗
openN = 2 | closedN = 1
(size == n * 2) ✗
(2 < 3) ✓
Current = (()(
backtrack(3,1)
Current = (()( ✗
(1 < 2) ✓
Current = (())
backtrack(2,2)
Current = (()) ✗
openN = 2 | closedN = 1
(size == n * 2) ✗
(2 < 3) ✓
Current = ()((
backtrack(3,1)
Current = ()(( ✗
(1 < 2) ✓
Current = ()()
backtrack(2,2)
Current = ()() ✗
openN = 3 | closedN = 1
(size == n * 2) ✗
(3 < 3) ✗
(1 < 3) ✓
Current = ((()
backtrack(3,2)
Current = ((() ✗
openN = 3 | closedN = 1
(size == n * 2) ✗
(3 < 3) ✗
(1 < 3) ✓
Current = (()()
backtrack(3,2)
Current = (()() ✗
openN = 2 | closedN = 2
(size == n * 2) ✗
(2 < 3) ✓
Current = (())(
backtrack(3,2)
Current = (())( ✗
(2 < 2) ✗
openN = 3 | closedN = 1
(size == n * 2) ✗
(3 < 3) ✗
(1 < 3) ✓
Current = ()(()
backtrack(3,2)
Current = ()(() ✗
openN = 2 | closedN = 2
(size == n * 2) ✗
(2 < 3) ✓
Current = ()()(
backtrack(3,2)
Current = ()()( ✗
(2 < 2) ✗
openN = 3 | closedN = 2
(size == n * 2) ✗
(3 < 3) ✗
(2 < 3) ✓
Current = (((()))
backtrack(3,3)
Current = (((())) ✗
openN = 3 | closedN = 2
(size == n * 2) ✗
(3 < 3) ✗
(2 < 3) ✓
Current = (()())
backtrack(3,3)
Current = (()()) ✗
openN = 3 | closedN = 2
(size == n * 2) ✗
(3 < 3) ✗
(2 < 3) ✓
Current = (())()
backtrack(3,3)
Current = (())() ✗
openN = 3 | closedN = 2
(size == n * 2) ✗
(3 < 3) ✗
(2 < 3) ✓
Current = ()(())
backtrack(3,3)
Current = ()(()) ✗
openN = 3 | closedN = 2
(size == n * 2) ✗
(3 < 3) ✗
(2 < 3) ✓
Current = ()()()
backtrack(3,3)
Current = ()()() ✗
openN = 3 | closedN = 3
(size == n * 2) ✓
return
openN = 3 | closedN = 3
(size == n * 2) ✓
return
openN = 3 | closedN = 3
(size == n * 2) ✓
return
openN = 3 | closedN = 3
(size == n * 2) ✓
return
openN = 3 | closedN = 3
(size == n * 2) ✓
return
((()))
(()())
(())()
()(())
()()()

# 739. Daily Temperatures

**739. Daily Temperatures** Hint

Medium 12.1K 277

Companies

Given an array of integers `temperatures` represents the daily temperatures, return *an array* `answer` *such that* `answer[i]` *is the number of days you have to wait after the* `i^th` *day to get a warmer temperature*. If there is no future day for which this is possible, keep `answer[i] == 0` instead.

**Example 1:**

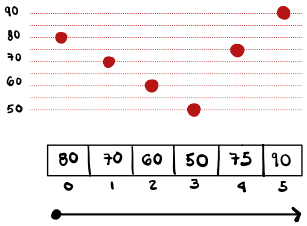
```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

**Example 2:**

```
Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]
```

**Example 3:**

```
Input: temperatures = [30,60,90]
Output: [1,1,0]
```



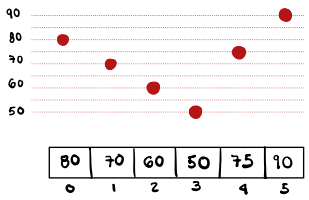

For each temperature:

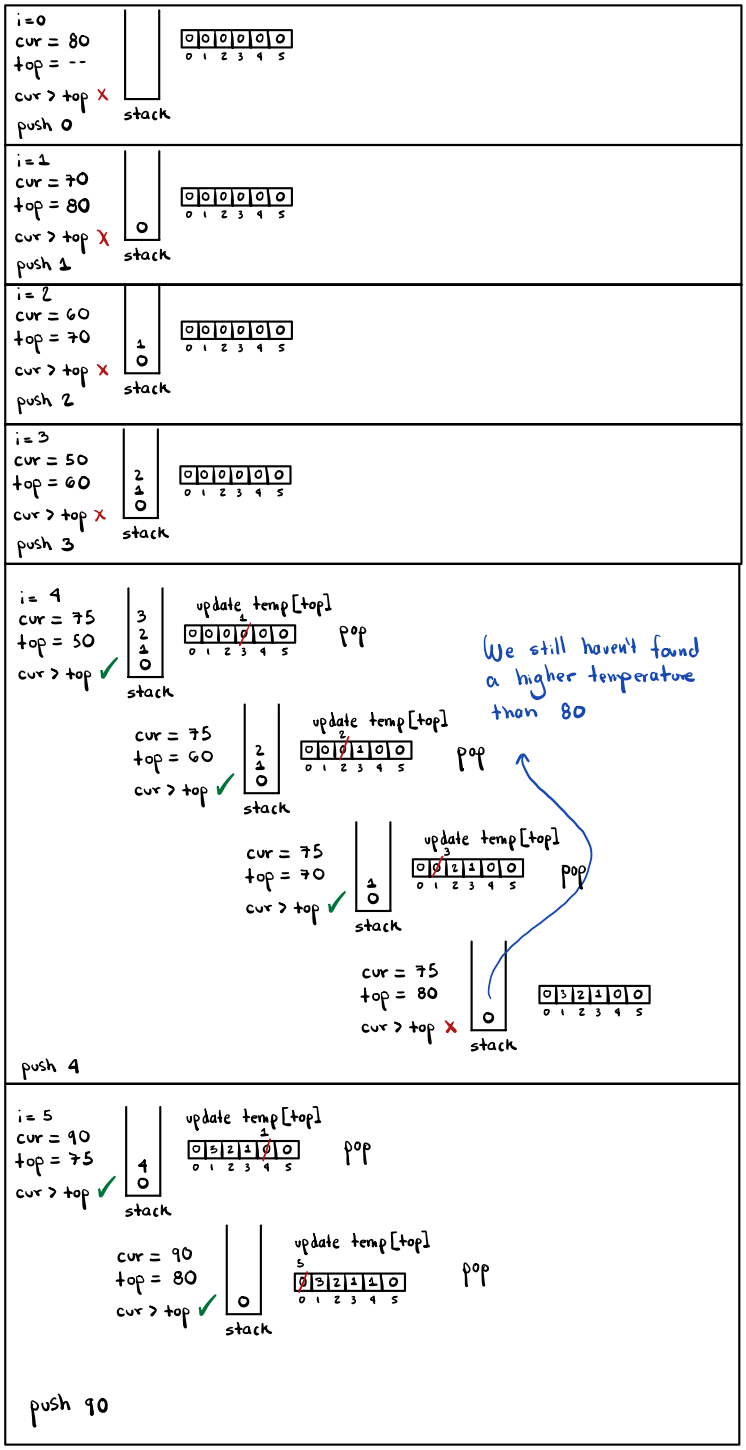
pop indices of all temperatures less than current temperature and at each pop, calculate time interval.

push index at the end of each iteration

Note: The stack keeps the index of the temperature for which we haven't found a higher temperature.

```cpp
vector<int> dailyTemperatures(vector<int>& temperatures) {

    int n = temperatures.size();

    vector<int> answer(n, 0); // Initialize all elements to zero.
    stack<int> days_stack; // Will keep track of the days [0,n] for which
                           // we haven't found a warmer temperature.

    for (int i = 0; i < n; i++) {

        // While the current temperature is greater than
        // the temperature of the DAY (index) on top of the stack
        while (!days_stack.empty() && temperatures[i] > temperatures[days_stack.top()]) {
            int n_of_days = i - days_stack.top(); // Calculate day difference
            answer[days_stack.top()] = n_of_days; // Update corresponding answer entry
            days_stack.pop();
        }
        days_stack.push(i);
    }
    // Small note: At the end, the stack contains the last temperature.
    return answer;
}
```


90
80
70
60
50
80 70 60 50 75 90
0 1 2 3 4 5


i=0
cur = 80
top = --
cur > top X
push 0
stack
i=1
cur = 70
top = 80
cur > top X
push 1
stack
i=2
cur = 60
top = 70
cur > top X
push 2
stack
i=3
cur = 50
top = 60
cur > top X
push 3
stack
i = 4
cur = 75
top = 50
cur > top ✓
stack
update temp[top]
pop
cur = 75
top = 60
cur > top ✓
stack
update temp[top]
pop
cur = 75
top = 70
cur > top ✓
stack
update temp[top]
pop
cur = 75
top = 80
cur > top X
stack
We still haven't found a higher temperature than 80
push 4
i = 5
cur = 90
top = 75
cur > top ✓
stack
update temp[top]
pop
cur = 90
top = 80
cur > top ✓
stack
update temp[top]
pop
push 90

# 853. Car Fleet

**853. Car Fleet**

Medium 3.2K 815

Companies

There are `n` cars going to the same destination along a one-lane road. The destination is `target` miles away.

You are given two integer array `position` and `speed`, both of length `n`, where `position[i]` is the position of the $i^{th}$ car and `speed[i]` is the speed of the $i^{th}$ car (in miles per hour).

A car can never pass another car ahead of it, but it can catch up to it and drive bumper to bumper **at the same speed**. The faster car will **slow down** to match the slower car's speed. The distance between these two cars is ignored (i.e., they are assumed to have the same position).

A **car fleet** is some non-empty set of cars driving at the same position and same speed. Note that a single car is also a car fleet.

If a car catches up to a car fleet right at the destination point, it will still be considered as one car fleet.

Return *the **number of car fleets** that will arrive at the destination*.

**Example 1:**

**Input:** target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]
**Output:** 3
**Explanation:**
The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at 12.
The car starting at 0 does not catch up to any other car, so it is a fleet by itself.
The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.
Note that no other cars meet these fleets before the destination, so the answer is 3.

**Example 2:**

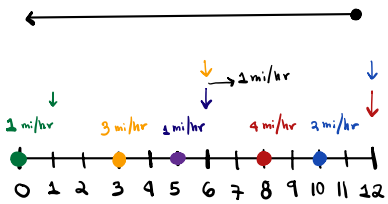
**Input:** target = 10, position = [3], speed = [3]
**Output:** 1
**Explanation:** There is only one car, hence there is only one fleet.

**Example 3:**

**Input:** target = 100, position = [0,2,4], speed = [4,2,1]
**Output:** 1
**Explanation:**
The cars starting at 0 (speed 4) and 2 (speed 2) become a fleet, meeting each other at 4. The fleet moves at speed 2.
Then, the fleet (speed 2) and the car starting at 4 (speed 1) become one fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

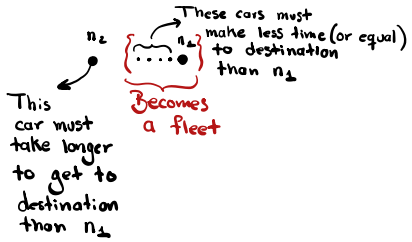
Problem List

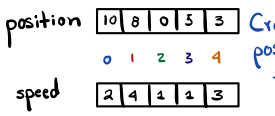
* Key ideas: - Cars cannot pass each other



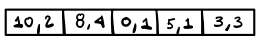

**General idea:**

- Traverse cars starting from the closest car from destination.
- For each car, calculate time to get to destination.
- Main observation:
  - * If cars to the left take less time (or equal) to get to the destination at some point they will reach a car (or fleet) that will make more time.
  - * If there is a car to the left the will make more time, it will belong to another fleet (as it will never reach anyone)

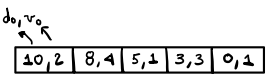


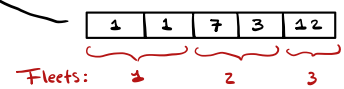

Create position, speed pairs:




Sort by position

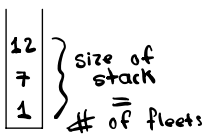



Calculate time to destination $t = \frac{d}{v} = \frac{target - d}{v}$

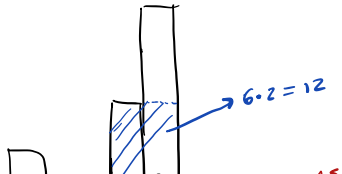



**Using a stack:**

- Push every time we encounter a larger number than the one on top.
- Do not push if number is less than or equal to the one on top.




```cpp
int carFleet(int target, vector<int>& position, vector<int>& speed) {

    vector<pair<int,int>> d_s_pairs;    // For (position, speed) pairs
    vector<float> time_to_target;       // Time for each car to get to target
    stack<double> stack;                // To count number of fleets

    int n = position.size();

    for (int i = 0; i < n; i++) {  // Create (position, speed) pairs
        pair<int,int> next_pair (position[i], speed[i]);
        d_s_pairs.push_back(next_pair);
    }

    // Sort in descending order
    sort(d_s_pairs.begin(), d_s_pairs.end()); // Sort in ascending order first
    reverse(d_s_pairs.begin(), d_s_pairs.end()); // Reverse

    for (auto &pair : d_s_pairs) {
        float next_time = float (target - pair.first) / pair.second; // (target - position) / speed
        time_to_target.push_back(next_time);
    }

    stack.push(time_to_target[0]); // push the first car's time (It's guaranteed it will arrive first)
                                   // as cars cannot pass each other.
    for (auto &next_time : time_to_target) {
        if (next_time > stack.top()) {
            stack.push(next_time);
        }
    }

    return stack.size();
}
```



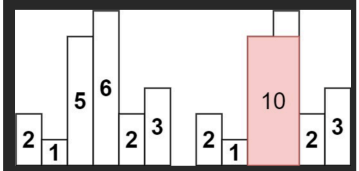
## 84. Largest Rectangle in Histogram

Hard 16.6K 242

Companies

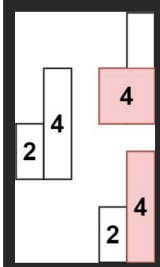
Given an array of integers `heights` representing the histogram's bar height where the width of each bar is `1`, return *the area of the largest rectangle in the histogram.*

**Example 1:**

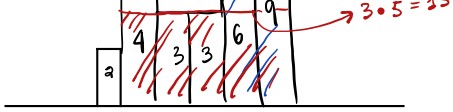


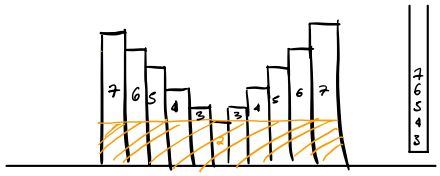

**Input:** heights = [2,1,5,6,2,3]
**Output:** 10
**Explanation:** The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.

**Example 2:**




**Input:** heights = [2,4]
**Output:** 4



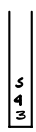




calculate area

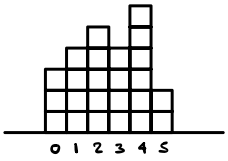
22

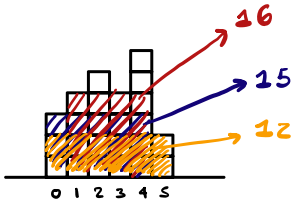
```cpp
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

int largestRectangleArea(vector<int>& heights) {
    stack<int> indices;
    int maxArea = 0;
    int n = heights.size();

    for (int i = 0; i <= n; i++) {
        // Handle the end of array by considering a height of 0
        int currentHeight = (i == n) ? 0 : heights[i];

        while (!indices.empty() && currentHeight < heights[indices.top()]) {
            int top = indices.top();
            indices.pop();
            int width = indices.empty() ? i : i - indices.top() - 1;
            maxArea = max(maxArea, heights[top] * width);
        }

        indices.push(i);
    }

    return maxArea;
}
```

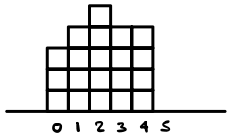
current height is shorter than the one on top




4<5

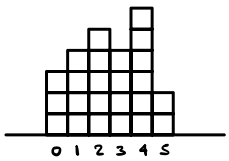


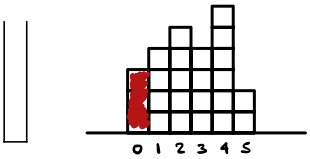




```cpp
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

int largestRectangleArea(vector<int>& heights) {
    stack<int> indices;
    int maxArea = 0;
    int n = heights.size();

    for (int i = 0; i <= n; i++) {
        // Handle the end of array by considering a height of 0
        int currentHeight = (i == n) ? 0 : heights[i];

        while (!indices.empty() && currentHeight < heights[indices.top()]) {
            int top = indices.top();
            indices.pop();
            int width = indices.empty() ? i : i - indices.top() - 1;
            maxArea = max(maxArea, heights[top] * width);
        }

        indices.push(i);
    }

    return maxArea;
}
```

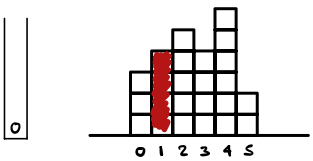
current height is shorter than the one on top




**i = 0**
CurHeight = 3
Non-empty Stack && CurHeight < heights[top] (Not executed; INVALID)
top =
width =
maxArea =
push(0)

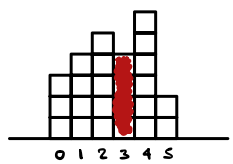



**i = 1**
CurHeight = 4 — 4 < 3 ✗
Non-empty Stack && CurHeight < heights[top]
top =
width =
maxArea =
push(1)



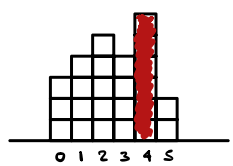

**i = 2**
CurHeight = 5 — 5 < 4 ✗
Non-empty Stack && CurHeight < heights[top]
top =
width =
maxArea =
push(2)


**i = 3**
CurHeight = 4 — 4 < 5 ✓
Non-empty Stack && CurHeight < heights[top]
top = 2
width = 3 − 1 − 1 = 1
maxArea = max(0, 5×1) = 5

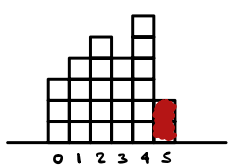



4 < 4 ✗
Non-empty Stack && CurHeight < heights[top]
top =
width =
maxArea =
push(3)

**i = 4**
CurHeight = 6 — 6 < 4 ✗
Non-empty Stack && CurHeight < heights[top]
top =
width =
maxArea =
push(4)




**i = 5**
CurHeight = 2 — 2 < 6 ✓
Non-empty Stack && CurHeight < heights[top]
top = 4
width = 5 − 3 − 1 = 1
maxArea = max(5, 6×1) = 6




2 < 4 ✓
Non-empty Stack && CurHeight < heights[top]
top = 3
width = 5 − 1 − 1 = 3 (? width=3)
maxArea = max(6, 4×3) = 12

2 < 4 ✓
Non-empty Stack && CurHeight < heights[top]
top = 1
width = 5 − 0 − 1 = 4
maxArea = (12, 4×4) = 16