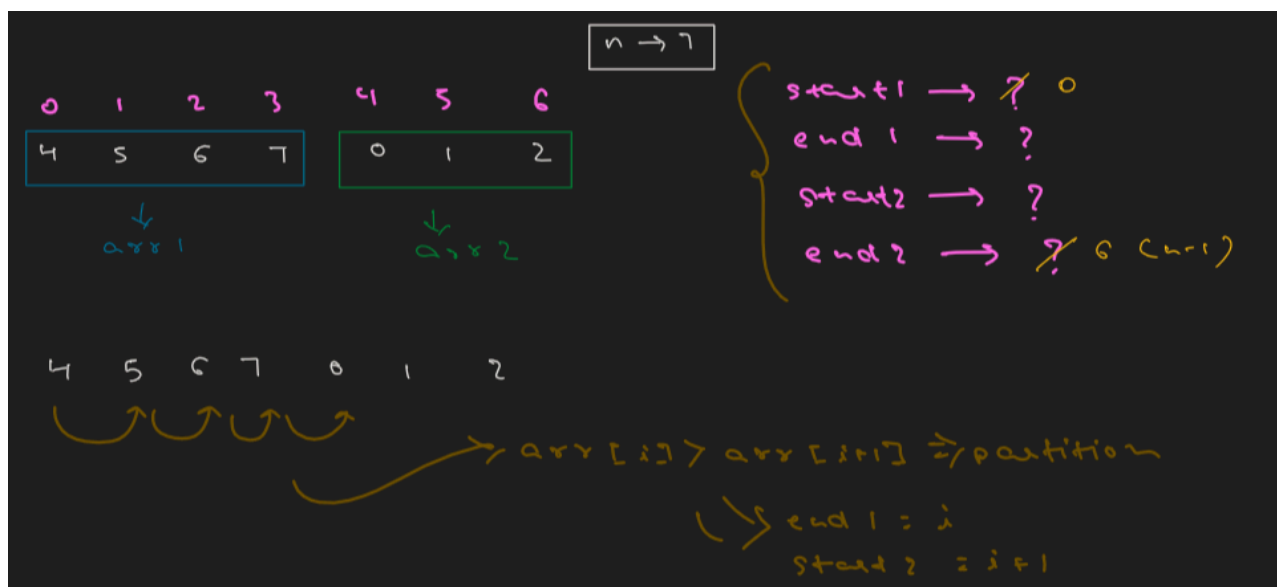


Binary Search on part of an array

```
public class Main {  
  
    public static void main(String[] args) {  
        int[] arr = { 2, 3, 4, 5, 6, 7, 8, 11, 13, 16, 17, 18 };  
        int item = 13;  
        int n = arr.length;  
        int start = 5;  
        int end = 9;  
        System.out.println(binarySearch(arr, start, end, item));  
    }  
  
    public static int binarySearch(int[] arr, int low, int high, int item) {  
        while (low <= high) {  
            int mid = (low + high) / 2;  
            if (arr[mid] == item) {  
                return mid;  
            } else if (arr[mid] > item) {  
                high = mid - 1;  
            } else {  
                low = mid + 1;  
            }  
        }  
        return -1;  
    }  
}
```

Search in Rotated Sorted Array

<https://leetcode.com/problems/search-in-rotated-sorted-array/>



The main logic lies here: if you look at the array, **it is sorted in two parts**. The first part of the array is sorted, and the second part is also sorted. We can clearly see that if we are able to find the index at which these two parts are separated, we can extract two individually sorted arrays — array one and array two. Then, we can apply binary search to both parts individually since each is sorted, and get the index of the desired element.

Now, how do we detect the point at which they are separated?

If you look at the array, **the rotation point is the moment where the next element becomes smaller than the previous one**. For example, in an array like `[4, 5, 6, 7, 0, 1, 2]`, up until `7`, each next element is greater than the previous. The moment this order breaks — when `0` comes after `7` — is the rotation point. So, the previous element becomes greater than the next element. This is how we can detect the rotation point.

```
class Solution {
    public int search(int[] nums, int target) {
        int n = nums.length;

        int start1 = 0;
        int end1 = 0;
        int start2 = 0;
        int end2 = n-1;
        for(int i=0; i<n-1; i++){
            if(nums[i]>nums[i+1]){
                end1 = i;
                start2 = i+1;
            }
        }

        System.out.println(start1 + " " + end1 + " " + start2 + " " + end2);

        int ans = binarySearch(nums, start1, end1, target);
        if(ans == -1){
            ans = binarySearch(nums, start2, end2, target);
        }

        return ans;
    }

    public static int binarySearch(int[] arr, int low, int high, int item) {
        while (low <= high) {
            int mid = low + (high - low)/2;
            if (arr[mid] == item) {
                return mid;
            } else if (arr[mid] > item) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
    }
}
```

```
        return -1;

    }

}
```



Binary Search for Ascending & Descending Sorted Arrays

```
public class Main {

    public static void main(String[] args) {
        int[] arr1 = { 2, 3, 4, 5, 6, 7, 8, 11, 13, 16, 17, 18 };
        int item = 13;
        System.out.println(binarySearchAscending(arr1, item));

        int[] arr2 = {18, 17, 16, 15, 13, 12, 7, 6, 1};
        System.out.println(binarySearchDescending(arr2, item));
    }

    public static int binarySearchAscending(int[] arr, int item) {
        int n = arr.length;
        int low = 0;
        int high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == item) {
                return mid;
            } else if (arr[mid] > item) {
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }
        return -1;
    }

    public static int binarySearchDescending(int[] arr, int item) {
        int n = arr.length;
        int low = 0;
        int high = n - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == item) {
                return mid;
            } else if (arr[mid] > item) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }
    }
}
```

```

    }
  }
  return -1;
}
}

```

🎯 Majority Element

<https://leetcode.com/problems/majority-element/>

🎯 Maximum Subarray

<https://leetcode.com/problems/maximum-subarray/>

🧩 Subarray

A subarray is simply a contiguous part of an array. Think of it like slicing a continuous portion of a long list — you don't skip elements, and you keep them in order.

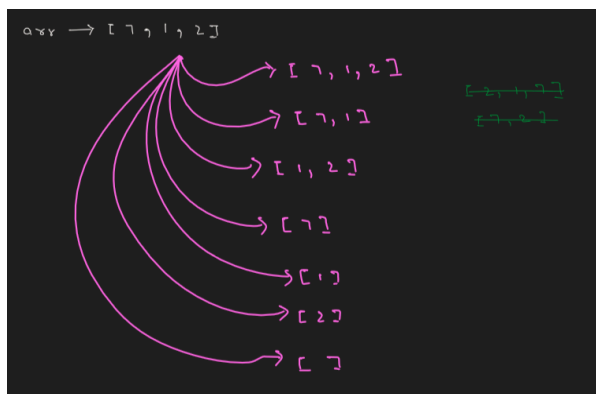
🛒 Subarray as a "Sabzi Mandi Shopping Spree"

Imagine you're at a bustling Indian sabzi mandi (vegetable market) with a long row of stalls selling veggies. Each stall represents an element in an array — say, potatoes, onions, tomatoes, bhindi, lauki, etc.

Now, your mom gives you a quirky challenge: "Beta, pick any continuous stretch of stalls and bring me all the veggies from that stretch — no skipping!"

- If you pick stalls 2 to 4 (onions, tomatoes, bhindi), that's a subarray.
- You can't pick stall 1, skip stall 2, and then pick stall 3 — that's not allowed. No skipping!
- You can pick just one stall (a subarray of length 1), or the entire row (the full array), or anything in between — as long as it's continuous.

So, a subarray is like your sabzi mandi haul — a continuous scoop of veggies from the lineup, no cheating!



Example:

```
arr = [7, 1, 2]
subarrays = [7], [1], [2], [7, 1], [1, 2], [7, 1, 2], [7, 2], [2, 1, 7]
```

The Problem: Maximum Subarray as a Buffet Challenge

Imagine you're at a lavish Indian wedding. The buffet line is long and dramatic — like a Bollywood saga. Each dish has a **taste score**:

- Delicious dishes = positive scores (Paneer Tikka: +5)
- Disastrous ones = negative scores (Ghevar: -6)
- Neutral ones = zero (Plain Rice: 0)

Now, you walk along the buffet and want to **fill your plate with a continuous stretch of dishes** that gives you the **maximum joy** — i.e., the highest total taste score.

But here's the twist:

- You can't skip dishes in between. If you start with Paneer, you must take whatever comes next — even if it's Karela — until you decide to stop.
- You're allowed to start anywhere and end anywhere — as long as the dishes are **contiguous**.

Your Goal

Find the **contiguous sequence of dishes** that gives you the **maximum total taste score**.

Example Buffet Line

Let's say the buffet is laid out like this:

[Paneer Tikka (+5), Karela (-3), Gulab Jamun (+4), Ghevar (-6), Chole (+7), Plain Rice (0), Ice cream (7), Malai Kofta (2), Water (-1)]

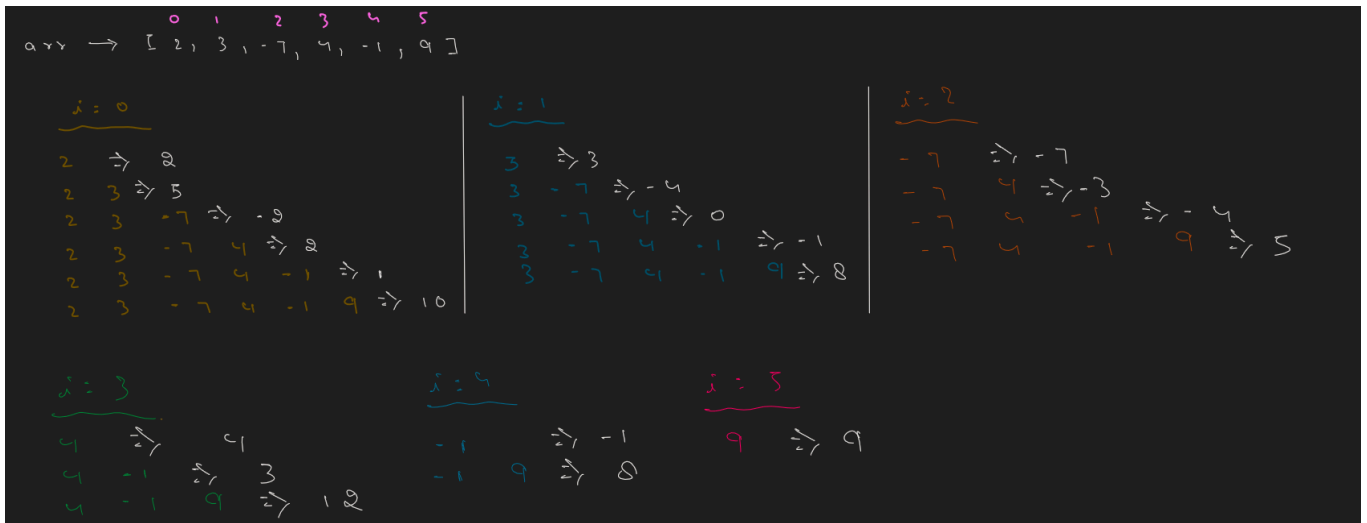
Your job is to look at this lineup and say:

“Hmm... if I start from Ice cream and go till Malai Kofta, I get +9. That's the best stretch!”

What this challenge is:

- Pick a **contiguous subarray** (a stretch of dishes).
- Add up their scores.
- Find the one with the **maximum total**.

Solution



The image shows a handwritten solution for the Maximum Subarray problem using a brute force approach. It starts with an array `arr → [2, 3, -7, 4, -1, 9]`. The solution then iterates through all possible starting indices i from 0 to 5. For each i , it calculates the sum of all contiguous subarrays starting at i and ending at j (where j ranges from i to 5). The maximum sum found is 12, which corresponds to the subarray `[4, -1, 9]` starting at index 3 and ending at index 5.

i	$j=i$	$j=i+1$	$j=i+2$	$j=i+3$	$j=i+4$	$j=i+5$
0	2	5	-5	1	-1	8
1	3	-4	0	-1	8	
2	-7	-3	-4	5		
3	4	3	12			
4	-1	0				
5	9					

A simple observation we've made is that, to find the sum of the next subarray, I just added the new element to the previous subarray sum, and that gave me the sum of the new subarray.

```
public class Main {

    public static void main(String[] args) {
        int[] arr = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
        System.out.println(maximumSum(arr));
    }

    public static int maximumSum(int[] arr) {
        int ans = Integer.MIN_VALUE; // -2^31
        for (int i = 0; i < arr.length; i++) {
            int sum = 0;
            for (int j = i; j < arr.length; j++) {
                sum += arr[j];
                ans = Math.max(ans, sum);
            }
        }
        return ans;
    }
}
```

```
class Solution {
    public int maxSubArray(int[] nums) {
        int ans = Integer.MIN_VALUE; // Initialize with smallest possible int val

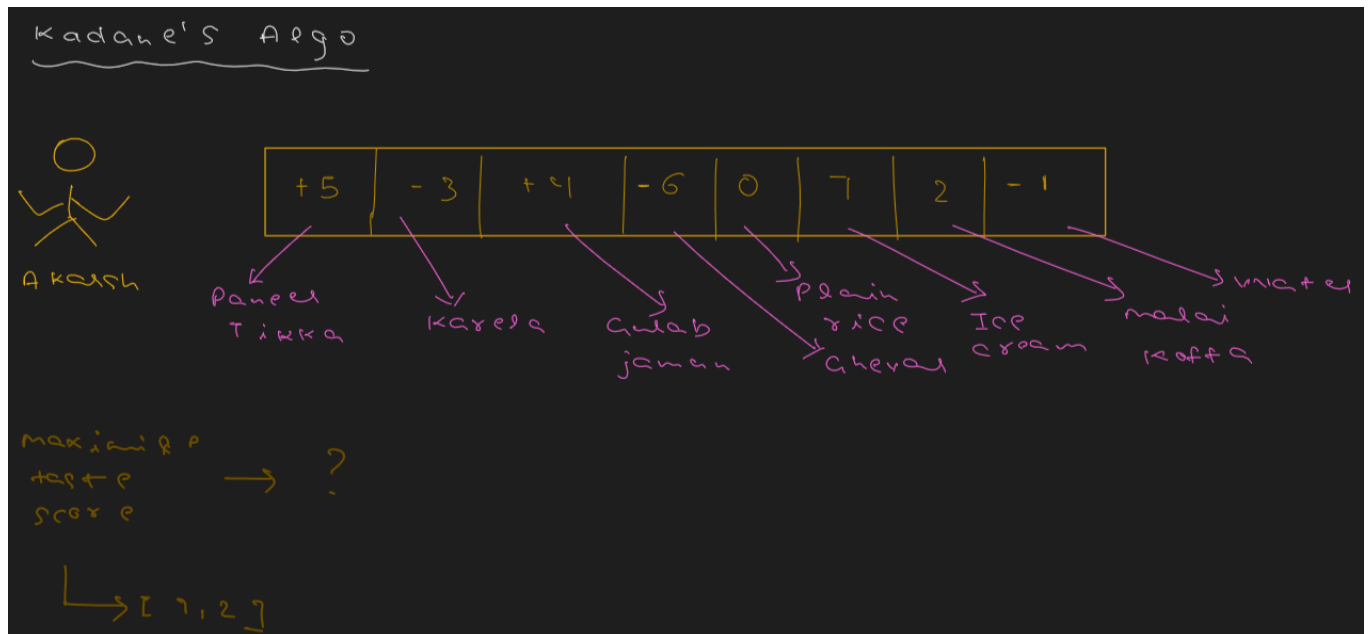
        // Outer loop to fix the starting point of subarrays
        for (int i = 0; i < nums.length; i++) {
            int sum = 0; // Initialize sum for the current subarray

            // Inner loop to extend the subarray to the right
            for (int j = i; j < nums.length; j++) {
                sum += nums[j]; // Add current element to the sum
                ans = Math.max(ans, sum); // Update maximum subarray sum if needed
            }
        }

        return ans; // Return the maximum subarray sum
    }
}
```

Kadane's algorithm

The Great Indian Wedding Buffet



Imagine you're at a big fat Indian wedding. The buffet is laid out in a long line of dishes — some are heavenly (paneer tikka, gulab jamun), some are disasters (karela, ghevar), and some are just meh (plain rice).

Each dish has a “taste score”:

- Paneer Tikka: +5
- Karela: -3
- Gulab Jamun: +4
- Ghevar: -6
- Plain Rice: 0
- Ice cream: 7
- Malai Kofta: 2
- Water: -1

Now, your goal is to **walk along the buffet line and pick a continuous stretch of dishes** that gives you the **maximum joy** — i.e., the highest total taste score.

But here's the twist: you can't skip dishes in between. If you start with Paneer, you have to take whatever comes next — even if it's Karela — unless you decide to start fresh from Gulab Jamun.

So what do you do?

You start tasting:

- First dish: Paneer (+5) → Yum!
- Next: Karela (-3) → Meh, still net +2
- Next: Gulab Jamun (+4) → Now we're at +6
- Next: Ghevar (-6) → Oof, total drops to 0
- You say, “Forget this stretch!” and start fresh from the next dish.

This is exactly what Kadane's Algorithm does:

- It keeps adding scores (current sum).
- If the sum drops below zero, it resets — just like you dropping the plate and starting fresh from the next tasty dish.

📌 Sample Buffet (Array): [5, -3, 4, -6, 0, 7, 2, -1]

- Best stretch: [7, 2] → Total taste score = 9
- That's your **maximum subarray**!

Try to find the maximum sum among all the subarrays starting from index zero.

For example, like Trump's bankruptcy, you can restart your life even if everything seems to be going wrong.

No issue with the negative element; the issue is with the sum being negative. A few losses are okay in business, but the entire business shouldn't be in loss.

```
public class Main {

    public static void main(String[] args) {
        int[] arr = { -2, 1, -3, 4, -1, 2, 1, -5, 4 };
        System.out.println(maximumSum(arr));
    }

    public static int maximumSum(int[] arr) {
        int ans = Integer.MIN_VALUE;
        int sum = 0;
        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
            ans = Math.max(ans, sum);
            if (sum < 0) {
                sum = 0;
            }
        }
        return ans;
    }
}
```

```
class Solution {
    public int maxSubArray(int[] nums) {
        int ans = Integer.MIN_VALUE; // Stores maximum subarray sum found so far
        int sum = 0; // Running sum of the current subarray

        for (int i = 0; i < nums.length; i++) {
            sum += nums[i]; // Add current element to the running sum
            ans = Math.max(ans, sum); // Update max if current sum is greater
            if (sum < 0) {
                sum = 0; // Reset sum if it becomes negative
            }
        }
    }
}
```



```

    return ans; // Return the maximum subarray sum
}
}

```

🎯 Trapping Rain Water

<https://leetcode.com/problems/trapping-rain-water/>

💧 Trapping Rain Water

Imagine a narrow *gali* in Old Delhi during monsoon. Buildings of random heights line the street—some tall like Sharma ji's 3-storey house, some short like the tea stall. After a heavy rain:

- Water collects between tall buildings where short ones sit in between.
- Kids shout: "Bhaiya, idhar to swimming pool ban gaya!" 💧

🏠 Example: Array = [4, 2, 0, 3, 2, 5] Think of it as building heights. Water gets trapped over the short ones like 0 and 2, between tall ones like 4 and 5.

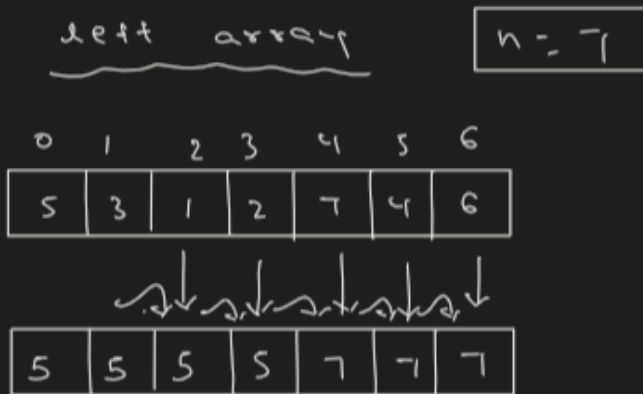
🔪 Formula: Water at position = $\min(\text{max left}, \text{max right}) - \text{height}[i]$



Computing left array

Suppose there are 100 students in a class, and we already know that the topper is Anjali. Now, imagine a new student named Raj joins the class. To determine who the new topper is, Raj doesn't need to compete with all 100 students individually. Instead, he can directly compete with Anjali—the current topper—and if he beats her, he becomes the new topper.

That's exactly how world championships work. If you've seen the movie *Apne*, you'll remember how Sunny Deol fights the reigning world champion at the end. By defeating him, Sunny becomes the new world champion. That's the same logic!



```
left[0] = arr[0];  
left[1] = max(left[0], arr[1]);  
left[2] = max(left[1], arr[2]);  
left[3] = max(left[2], arr[3]);  
left[4] = max(left[3], arr[4]);  
left[5] = max(left[4], arr[5]);  
left[6] = max(left[5], arr[6]);
```

```
for(int i=1; i<n; i++) {  
    left[i] = max(left[i-1], arr[i]);  
}
```

Computing right array

right array

$n = 7$

0	1	2	3	4	5	6
5	3	1	2	7	4	6

7	7	7	7	7	6	6
---	---	---	---	---	---	---

$right[n-1] = arr[n-1];$

$right[6] = arr[6];$

$right[5] = \max(right[6], arr[5]);$

$right[4] = \max(right[5], arr[4]);$

$right[3] = \max(right[4], arr[3]);$

$right[2] = \max(right[3], arr[2]);$

$right[1] = \max(right[2], arr[1]);$

$right[0] = \max(right[1], arr[0]);$

```

for (int i = n-2; i >= 0; i--) {
    right[i] = max(right[i+1],
                    arr[i]);
}
    
```

```

public class Main{

    public static void main(String[] args) {
        int[] arr = { 0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1 };
        System.out.println(trapping(arr));
    }

    public static int trapping(int[] arr) {
        int n = arr.length;
        int[] left = new int[n];
        int[] right = new int[n];

        // Fill left[] array with max height to the left of each index
        left[0] = arr[0];
        for (int i = 1; i < n; i++) {
            left[i] = Math.max(left[i - 1], arr[i]);
        }
    }
}
    
```

```

    }

    // Fill right[] array with max height to the right of each index
    right[n - 1] = arr[n - 1];
    for (int i = n - 2; i >= 0; i--) {
        right[i] = Math.max(right[i + 1], arr[i]);
    }

    // Calculate total water trapped at each index
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += Math.min(left[i], right[i]) - arr[i];
    }

    return sum;
}
}

```

```

class Solution {
    public int trap(int[] height) {

        int n = height.length;
        int[] left = new int[n];
        int[] right = new int[n];

        // Fill left[] array with max height to the left of each index
        left[0] = height[0];
        for (int i = 1; i < n; i++) {
            left[i] = Math.max(left[i - 1], height[i]);
        }

        // Fill right[] array with max height to the right of each index
        right[n - 1] = height[n - 1];
        for (int i = n - 2; i >= 0; i--) {
            right[i] = Math.max(right[i + 1], height[i]);
        }

        // Calculate total water trapped at each index
        int sum = 0;
        for (int i = 0; i < right.length; i++) {
            sum = sum + Math.min(left[i], right[i]) - height[i];
        }
        return sum;
    }
}

```

◆ Prefix-Suffix Concept

In the above question, the rainwater trapping problem uses the prefix-suffix concept. For example, if I write "Mr Akarsh", "Mr" is a prefix. Similarly, if I write "Mr Akash Jaiswal", then "Jaiswal" is the suffix — basically, the suffix is the ending and the prefix is the beginning. In this question, to calculate the value at the *i*th index, I used the prefix-suffix idea. I calculated all the maximum values *before* the *i*th index and stored them in the *left* array (which you can think of as the prefix). I then calculated all the maximum values *after* the *i*th index and stored them in the *right* array (which acts like the suffix). So, the *left* array is the prefix, and the *right* array is the suffix.

🎯 Product of Array Except Self

<https://leetcode.com/problems/product-of-array-except-self/>

🧳 Family Packing for a Trip 🚗

👨👩 The Situation:

A family of 4 — **Papa, Mummy, Sonu, and Monu** — is packing bags for a road trip. Each person is responsible for packing a **certain number of essential items**:

bags = [2, 3, 5, 4]

- Papa packed 2 items
- Mummy packed 3
- Sonu packed 5
- Monu packed 4

🎯 The Mission:

Mummy wants to know:

"If I remove **each** person's contribution, how much effort did **everyone else** make *together*?"

In other words: **product of all other contributions**, for **each** family member — but without using division!

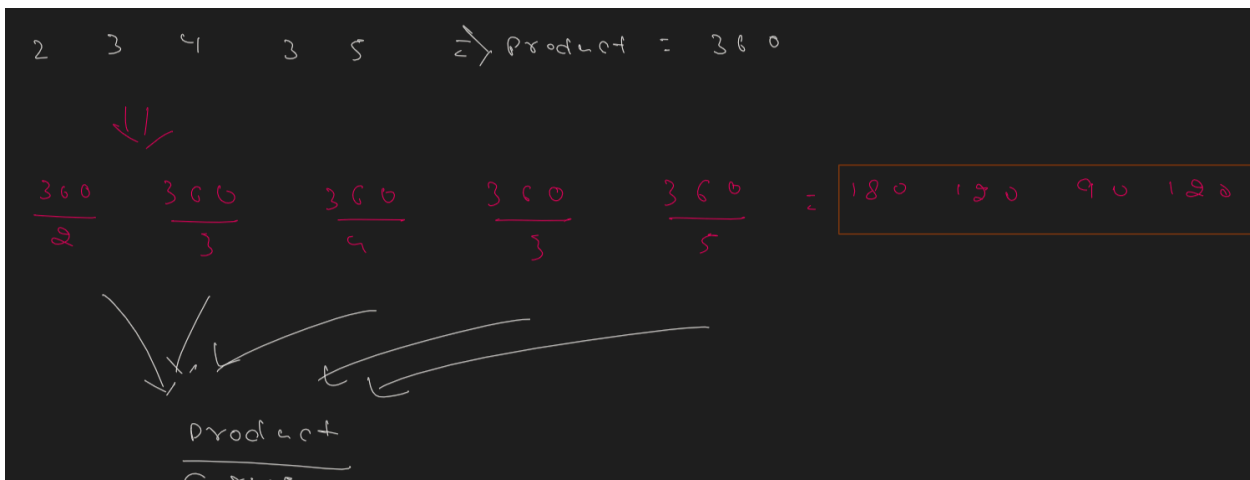
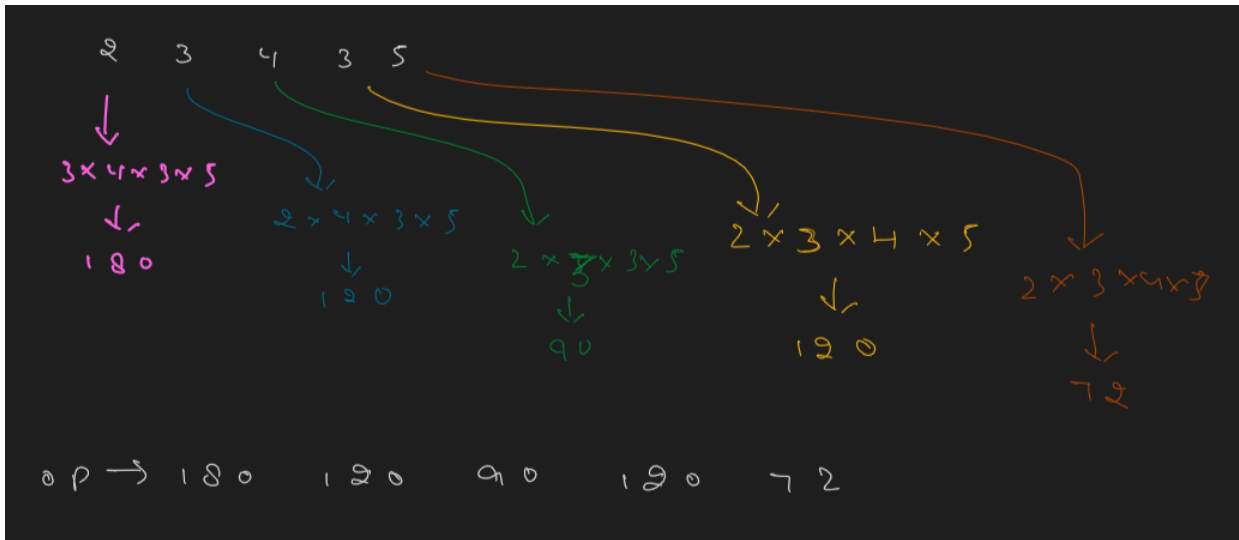
(Because Indian moms don't divide — they **delegate smartly** 😎)

✅ Desired Output:

- Papa's contribution removed: $3 \times 5 \times 4 = 60$
- Mummy's removed: $2 \times 5 \times 4 = 40$
- Sonu's removed: $2 \times 3 \times 4 = 24$
- Monu's removed: $2 \times 3 \times 5 = 30$

Final result: [60, 40, 24, 30]

✗ Brute Force Solution: Using Division Operator



```
public class Main {  
  
    public static void main(String[] args) {  
        int[] arr = {2, 3, 4, 3, 5};  
        // System.out.println(product(arr));  
  
        int[] nums = productOfArrayExceptSelf(arr);  
  
        for(int i=0; i<nums.length; i++){  
            System.out.print(nums[i]+" ");  
        }  
  
    }  
  
    public static int product(int[] arr){  
        int prod = 1;  
        int n = arr.length;  
        for(int i=0; i<n; i++){  
            prod = prod*arr[i];  
        }  
    }  
}
```

```

    }
    return prod;
}

public static int[] productOfArrayExceptSelf(int[] arr){
    int prod = product(arr);
    int n = arr.length;
    int[] op = new int[n];

    for(int i=0;i<n;i++){
        op[i] = prod/arr[i];
    }

    return op;
}
}

```

🧠 Real-Life Jugaad: How Mummy Thinks

She doesn't divide. She uses "left and right contribution tracking":

🌱 Step 1: Left-side Contribution (before each person)

left = [1, 2, 2×3=6, 6×5=30] → [1, 2, 6, 30]

🌱 Step 2: Right-side Contribution (after each person)

right = [3×5×4=60, 5×4=20, 4, 1] → [60, 20, 4, 1]

💥 Final Output:

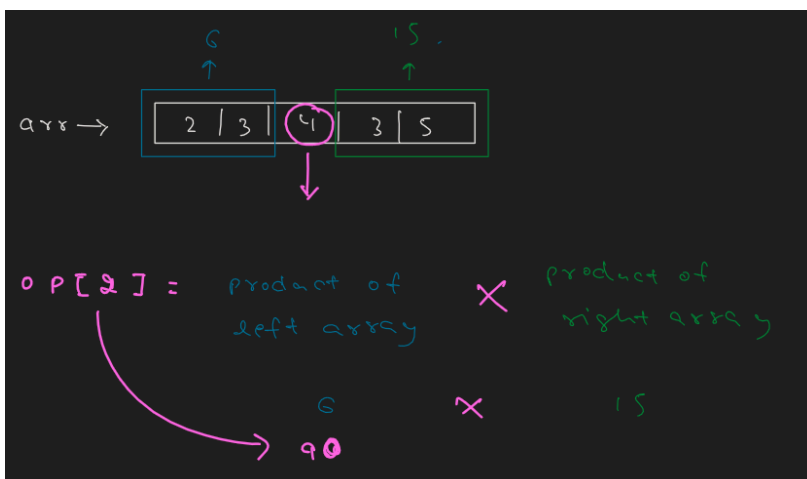
result = [1×60, 2×20, 6×4, 30×1] → [60, 40, 24, 30]

😊 Mummy's Wisdom:

"Now I know who's slacking and who's actually pulling their weight! Next trip, I'm giving Papa fewer tasks!" 😂

✅ Why This Analogy Works:

- Each person's role matters, but we want to measure their absence's effect.
- In real life, we often want to know: "If this person didn't help, how much would be left for others?"



To solve this problem, I need to find the **prefix and suffix product arrays**. Suppose I'm calculating the left (prefix) array — **for each element, I will leave out that specific element and calculate the product of all the elements to its left**. I'll do the same for the right (suffix) array — leaving the current element and finding the product of all the elements to its right. The final result for each position will be the product of the corresponding values from the left and right arrays.



left[0] = 1 ✗

```

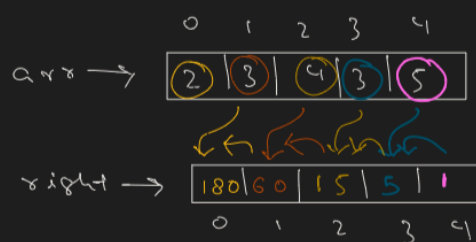
left[1] = left[0] * arr[0]
left[2] = left[1] * arr[1]
left[3] = left[2] * arr[2]
left[4] = left[3] * arr[3]

```

```

for(int i = 1; i < n; i++) {
    left[i] = left[i-1] * arr[i-1]
}

```



right[n-1] = 1

```

right[3] = right[4] * arr[4]
right[2] = right[3] * arr[3]
right[1] = right[2] * arr[2]
right[0] = right[1] * arr[1]

```

```

for(int i = n-2; i >= 0; i--) {
    right[i] = right[i+1] * arr[i+1]
}

```



```

public class Main {

    public static void main(String[] args) {

        int[] arr = { 1, 2, 3, 4 };
        int[] a = productOfArray(arr);
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }

    }

    public static int[] productOfArray(int[] arr) {

        int n = arr.length;
        int[] left = new int[n];
        int[] right = new int[n];

        left[0] = 1;
        for (int i = 1; i < n; i++) {
            left[i] = left[i - 1] * arr[i - 1];
        }

        right[n - 1] = 1;
        for (int i = n - 2; i >= 0; i--) {
            right[i] = right[i + 1] * arr[i + 1];
        }

        for (int i = 0; i < n; i++) {
            left[i] = left[i] * right[i];
        }

        return left;
    }

}

```

```

class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;

        // Arrays to store left and right products
        int[] left = new int[n];
        int[] right = new int[n];

        // left[i] contains product of all elements to the left of index i
        left[0] = 1;
        for (int i = 1; i < n; i++) {
            left[i] = left[i - 1] * nums[i - 1];
        }
    }
}

```

```
    }

    // right[i] contains product of all elements to the right of index i
    right[n - 1] = 1;
    for (int i = n - 2; i >= 0; i--) {
        right[i] = right[i + 1] * nums[i + 1];
    }

    // Multiply left and right products for the final result
    for (int i = 0; i < n; i++) {
        left[i] = left[i] * right[i];
    }

    return left; // left now contains the result
}

}
```