# 🎈

# 312. Burst Balloons

| | |
|---|---|
| DSA 👻 | |
| | 52 |
| 👾 Algo | Imp |
| topic 🦦 | |
| 🦧 Desc | Key identification is neighbor dependencies.<br><br>If the cost of an operation depends on its neighbors, and the array shrinks, try solving it by choosing the<br>*last* operation in each subproblem. |

## A classic DP (Dynamic Programming) problem known as "Burst Balloons"

### 🔍 Let's explore your confusion:

You asked:

> Why not just remove a number (like 1 from [3,5,1,8]), and then the remaining subproblem [3,5,8] has no dependencies?

At first glance, it **seems** like removing any balloon independently shouldn't affect others — but **that's not the case**.

---

### 🎯 Key Insight:

The formula for coins when bursting `nums[i]` is:

```
coins = nums[i-1] * nums[i] * nums[i+1]
```

So **when you burst a balloon, the reward depends on its left and right neighbors** — meaning **the remaining balloons are not independent anymore** because **the neighbor relationship changes**.

That's why we **flip the problem**:

🔁 Instead of asking "Which balloon to burst now?",

we ask: "Which balloon should be the **last** to burst in this subarray?"

---

## ✅ Why bursting the last balloon in a range creates independent subproblems

Let's say you have balloons between index `left` and `right`.

You decide that `k` will be the **last** balloon to burst in that range.

Then:

- The coins gained by bursting `k` last will be:
  `nums[left-1] * nums[k] * nums[right+1]`
- Everything left of `k` (i.e., `left to k-1`) and right of `k` (i.e., `k+1 to right`) can be solved independently.

This ensures **no dependency leakage** because `left-1` and `right+1` are fixed during the subproblem.

That's why we pad the array with `1` on both ends → `[1,3,5,1,8,1]`.

---

## 🔁 So why not go forward?

If you remove a balloon in the middle early on, the neighbors change and thus **you lose control over the surrounding context**.

This violates the principle of **optimal substructure**, which is needed for DP.

---

### 🔨 Final Takeaway:

- **Forward approach fails** because **neighbor context keeps shifting**, breaking subproblem independence.
- **Reverse (burst last) approach works** because:
  - We fix the neighbors: `left-1`, `right+1`.
  - The two sides of the chosen last balloon can be solved **independently**.
  - This allows clean memoization.

---

# Why would removing a balloon and creating a copy not work?

### ✅ You're Right:

> If modifying the array (removing elements) causes issues due to neighbor dependencies, then why not:

- Copy the array at each recursive step,
- Burst a balloon,
- Pass the new array forward.

That way the **original array stays intact**.

---

### 🧠 But Here's the Catch:

Yes, this **does work logically**, but...

> ❗ It's incredibly inefficient.

Because every recursive step:

- Creates a new array (copy),
- Does O(n) work to remove a balloon,

- And continues this for `n` balloons.

That makes the time complexity **O(n!)**, just like full brute-force.

So even with copying, **you don't get dynamic programming benefits** like **memoization** because:

- You're changing the structure (the array itself),
- So **subproblems are hard to identify or reuse**.

---

## 🧠 But When We Use "Burst Last" Approach:

Instead of copying arrays, we do this:

- Work on **fixed intervals** `[left, right]`
- For each interval, try all `k` in `(left, right)` as the **last balloon**
- Memoize `dp[left][right]` so we **never recompute** that interval

This makes time complexity:

```
O(n^3)
```

Which is **exponentially faster** than `O(n!)`.

---

## 🔁 Summary

| Idea | Is it Correct? | Is it Efficient? |
|---|---|---|
| Copy array & simulate bursting | ✅ Logically correct | ❌ Very slow (O(n!)) |
| DP with "burst last in interval" | ✅ Correct | ✅ Efficient ($O(n^3)$) |

---

## 👇 Visualization (Burst Last Approach):

Suppose `nums = [3,1,5,8]`, padded: `[1,3,1,5,8,1]`

Let `dp[i][j] = max coins you can get by bursting all balloons between i and j (exclusive)`

We do:

```
for length in range(2, n):
    for left in range(0, n - length):
        right = left + length
        for k in range(left+1, right):
            dp[left][right] = max(
                dp[left][right],
                dp[left][k] + dp[k][right] + nums[left]*nums[k]*nums[right]
            )
```

This solves the problem bottom-up, without modifying arrays.

---

## ✅ Key Observations & Thought Process in an Interview

### 1. Non-local dependencies (left/right neighbors involved)

You get coins based on **left and right neighbors**:

```
ini
CopyEdit
coins = nums[i - 1] * nums[i] * nums[i + 1]
```

💡 **Red Flag**:

> Bursting nums[i] affects the rewards of bursting other balloons due to changing neighbors.

🧠 Trigger:

> This breaks greedy and forward DP. Time to fix the neighbors instead of the balloon!

---

## 2. Order of operation affects rewards

> Bursting the same balloon at different times gives different coins due to shifting neighbors.

🔁 Instead of thinking "Which balloon do I burst now?", flip to:

> "Which balloon should be the last one in this subproblem?"

That fixes neighbors and enables **independent subproblems**.

---

## 3. Subproblem Structure Exists → DP Time!

If you realize:

- The total problem can be broken into left/right subproblems.
- Subproblems overlap (same `[left, right]` range may occur again).

🚀 Boom: **DP with memoization or bottom-up table**.

---

## 4. Optimal Substructure

The total coins for bursting `[left+1 ... right-1]` balloons is:

```
sql
CopyEdit
dp[left][right] = max over k in (left+1 to right-1) of:
   dp[left][k] + dp[k][right] + nums[left] * nums[k] * nums[right]
```

This clean recurrence is a dead giveaway that **DP on intervals** is the best approach.

---

## 5. Avoid modifying the input

If you find yourself **removing elements** or **copying arrays** during recursion:

⚠️ That's a hint you might need to switch to a DP-based indexing approach.

---

## 🧭 Interview Strategy Summary (Mental Checklist)

| Question Characteristic | Think |
|---|---|
| Bursting/removal gives score based on neighbors | Fix neighbors → burst last |
| Changing the order gives different scores | DP, not greedy |
| Problem has overlapping subproblems | Memoization |
| Problem asks for **max** score / coins | Optimization + DP |
| Removing elements breaks independence | Use index ranges instead |

---

Would you like a **one-liner decision rule** to help you in interviews? Here it is:

> 🧠 If the cost of an operation depends on its neighbors, and the array shrinks, try solving it by choosing the last operation in each subproblem.

This applies to problems like:

- Burst Balloons
- Matrix Chain Multiplication
- Optimal BST

---