Intuition

We must determine if an array has **132 pattern** or not.

First solution that would pop into mind is to use **Brute Force**. 💪
Use **three for loop** to get what you want but unfortunately, it will throw TLE error. 😨
even if we lower them to **two for loops** it will also give us TLE error. 😨

Seems challenging problem. 😠
But we can solve it. 💪 🚀

let's revise the **Brute Force** Solutions
For the **three loops** solution, We can loop on all the array elements to get the subsequence that satisfies the **132 pattern**.

```
for (size_t i = 0; i < nums.size() - 2; i++) {
        for (size_t j = i + 1; j < nums.size() - 1; j++) {
           for (size_t k = j + 1; k < nums.size(); k++) {
              if (nums[k] > nums[i] and nums[j] > nums[k]) {
                 return true;
              }
           }
        }
     }
```

For the **two loops** solution, We can maintain the **minimum** number before our **current** number and simply add **second** loop to search for a number that is between the current number and minimum number.

```
for (size_t j = 0; j < nums.size() - 1; j++) {
        min_i = min(min_i, nums[j]);
        for (size_t k = j + 1; k < nums.size(); k++) {
           if (nums[k] < nums[j] and min_i < nums[k]) {
              return true;
           }
        }
     }
```

Till now, everything looks great but how can we make the solution only **one** for loop? 🤔
We can realize here that our **problem** in the two for loop solution is the **third** number which is the **middle-value** in **132 pattern** and we need a way to find it without for loop. 🤩

Why not **reverse traversal** the array ? 🤔
Since this number is always the **third** then simply **reverse traversal** and **cache** every number that you encounter in some way.

what would be that way ? 🤔
Soooooooo, in the **132 pattern** we need to find three numbers 1-valued call it a, 3-valued call it b and 2-valued call it c.

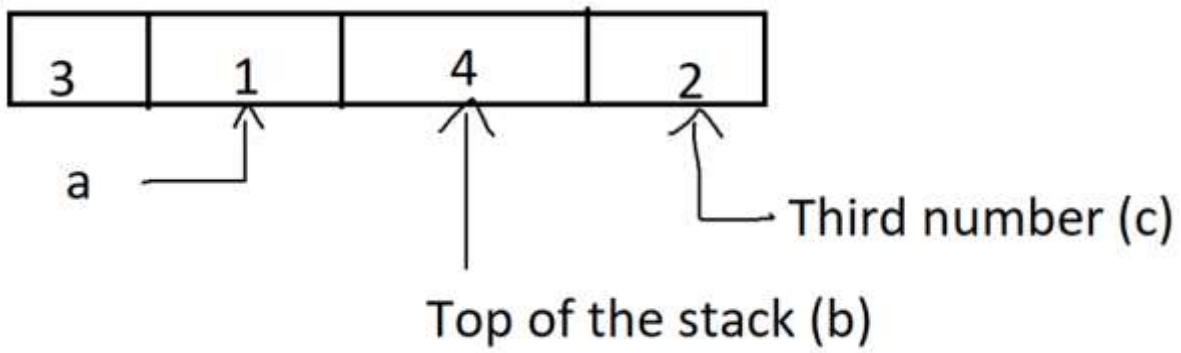
we can **cache** c element easy in some variable.
what about b element ?
Here comes the hero of the day **THE STACK** 🦸 .

- We can use the stack for to things
    - Obviously to **store** all numbers from the end of the array
    - When you encouter number **bigger** than the **top** of the stack simply **pop** the top of the stack and put this number in the variable for c
    - push that **bigger** number into the stack

Why did we do the last two steps ? 🤔
if the number we encoutered is **bigger** than the top of the stack then pop any number from the stack, **consider** the popped number as c the **encountered** number as b.

- And we are sure from two values:
    - b which is in the top of the stack.
    - c which is in the cached variable.

the only remaining step is to find a and that what we will maintaining by continue the looping.



Approach

1. Create a stack decreasingStack to keep track of decreasing elements.
2. Initialize maxThirdElement to the minimum possible value.
3. **Traverse** the Array from Right to Left and for each element in the array:
    - If the **current** element is **less** than maxThirdElement, return **true** (found a 132 pattern).
    - While the **stack** is not empty and the top element of the stack is **less** than the current element:
        - **Update** maxThirdElement to the top element of the stack.
        - **Pop** the top element from the stack.
    - **Push** the Current Element onto the Stack.
4. If no 132 pattern is found after traversing the array, return **false**.

Complexity

- **Time complexity:** O(N)
  Since we are iterating the array from right to left then it is **linear** time and all the operations inside the loop are O(1) then final complexity is O(N).
- **Space complexity:** O(N)
  Since we are maintaining a stack that at any point can have elements equal the elements of the array so complexity is O(N).