Algorithm for the Challenge

Problem: Push `[ "1", "2", "3" ]`, pop 2, then push `"4"`. Which item is at the top?

Algorithm (step-by-step in sequence)

- **Start** with an empty stack.
- **Push "1"** onto the stack.
- **Push "2"** onto the stack.
- **Push "3"** onto the stack.
- **Pop** the top item (removes `"3"`).
- **Pop** the top item again (removes `"2"`).
- **Push "4"** onto the stack.
- **Return/View** the top element of the stack.

Stop.

Python Code Implementation

```
# Initialize empty stack
stack = []
# Step 2–4: Push ″1″, ″2″, ″3″
stack.append(″1″)
stack.append(″2″)
stack.append(″3″)
# Step 5–6: Pop two items
stack.pop()   # removes ″3″
stack.pop()   # removes ″2″
# Step 7: Push ″4″
stack.append(″4″)
# Step 8: View the topprint(″Final stack:″, stack)print(″Top of the
stack:″, stack[-1])
```

Explanation

After pushing **1, 2, 3**, the stack is: `["1", "2", "3"]`

After **popping twice**, the stack is: `["1"]`

After pushing **"4"**, the stack is: `["1", "4"]`

Top of the stack = "4"

Reflection: Why Stack Best Models Backtracking

Backtracking is a systematic way of exploring possible solutions to a problem by making a series of choices and undoing them when a dead end is reached. The **stack data structure** is the most natural tool for modeling backtracking because it strictly follows the **Last-In-First-Out (LIFO)** principle. The last action performed is always the first one that should be undone when an error or a wrong path is detected.

In the given examples, this behavior is clear. For instance, in the **UR quizzes**, we push `"QuizA"`, `"QuizB"`, and `"QuizC"` onto the stack. If the student wants to undo the last attempted quiz, the stack allows us to pop `"QuizC"`, since it was the most recent one added. This directly reflects the idea of backtracking: undoing the last step taken before continuing with earlier ones. Similarly, in the **Irembo steps**, when `"StepA"`, `"StepB"`, and `"StepC"` are added, the system can only backtrack by removing `"StepC"` first, then `"StepB"`, until it finally reaches `"StepA"`. This mirrors real-world processes like online form completion, where the last step filled must be the first corrected.

The **Challenges stack** strengthens this reflection. After pushing `"1"`, `"2"`, and `"3"`, two pops undo `"3"` and `"2"`, leaving `"1"` as the base state. By then pushing `"4"`, the stack reflects a new branch in the exploration path. This is exactly how algorithms such as depth-first search (DFS), maze solving, or the N-Queens problem operate: they push new possibilities onto the stack and pop them when they fail, returning to the previous choice point.

The reason a stack is superior for modeling backtracking is its **faithfulness to human reasoning and problem-solving patterns**. When people retrace their steps, they naturally undo the last action first, not the first one they ever did. A queue, for example, would undo the earliest step first, which breaks the logical sequence of corrections. The LIFO nature of stacks ensures that the computational model aligns with the cognitive model of “step forward, step back.”

In summary, stacks are the best model for backtracking because they:

- Maintain the exact order of operations,
- Enable efficient undoing of recent actions, and
- Support recursive and branching explorations in problem solving.

Whether in quizzes, online forms, or algorithmic challenges, stacks provide the structure needed for controlled trial-and-error, ensuring that backtracking is both efficient and logically consistent.

Reflection & Solution: Queue vs Stack for Distributing IDs

Introduction
In programming, data structures like queues and stacks are used to organize and process information. When distributing IDs, the choice of data structure determines the order in which recipients receive their IDs. A queue follows First-In-First-Out (FIFO), while a stack follows Last-In-First-Out (LIFO). This difference influences fairness and efficiency in real-world scenarios such as government institutions, banks, or ATM queues.

Algorithm for Queue Distribution (FIFO)
Step 1: Initialize an empty queue.
Code: ID_queue = deque()

Step 2: Insert all IDs into the queue, one by one.
Code:
```
for i in range(1, num_IDs + 1):
    ID_queue.append(f"ID{i}")
```

Step 3: Remove IDs from the front until the queue is empty.
Code:
```
while ID_queue:
    distributed_IDs.append(ID_queue.popleft())
```

Step 4: Output the distributed IDs in the order they were added.
Code: return distributed_IDs

Result: IDs are distributed in the same order they arrived (ID1, ID2, ID3…). This ensures fairness—first registered, first served.

Algorithm for Stack Distribution (LIFO)
Step 1: Initialize an empty stack.
Code: ID_stack = []

Step 2: Insert all IDs into the stack.
Code:
```
for i in range(1, num_IDs + 1):
    ID_stack.append(f"ID{i}")
```

Step 3: Remove IDs from the top until the stack is empty.
Code:
```
while ID_stack:
    distributed_IDs.append(ID_stack.pop())
```

Step 4: Output the distributed IDs in reverse order.
Code: return distributed_IDs

Result: The last ID added is given out first (ID5, ID4, ID3…). This benefits latecomers but is unfair to those who arrived early.

Comparison: Queue vs Stack

- Order: Queue = First-in, First-out | Stack = Last-in, First-out
- Fairness: Queue = fair (early arrivals served first) | Stack = unfair (late arrivals served first)
- Use Case: Queue = ID distribution, tickets, customer lines | Stack = undo operations, backtracking
- Example Output: Queue → [ID1, ID2, ID3, ID4, ID5] | Stack → [ID5, ID4, ID3, ID2, ID1]

Reflection:
The queue is the most suitable model for distributing IDs because it respects the order of arrival and guarantees fairness. In real-world contexts such as RRA citizens or BK ATM clients, a queue ensures that people who arrive first are served first, preventing conflict or bias. On the other hand, a stack is best for tasks like backtracking or undo operations, where we need to revisit the last step made.

Thus, while both data structures are powerful, their choice depends on the problem context. For ID distribution, the queue (FIFO) provides an efficient and just solution.

Reflection: Why FIFO Supports Fairness in Government Services

In public institutions like the Rwanda Revenue Authority (RRA) or banks such as BK, the way services are given to people must be fair and orderly. The **queue system (FIFO – First In, First Out)** is the best way to make sure of this. With FIFO, the first person to arrive is also the first one to be helped, and those who come later wait for their turn. This makes the process transparent, so no one feels ignored or treated unfairly.

For example, in the case of RRA, when six citizens arrive, they line up in the order they came. In the program, the `append()` function puts each new citizen at the end of the line. Later, the `popleft()` function serves them starting with Citizen1 and Citizen2, leaving the rest in the same order. This is exactly how it works at service counters—people are helped one by one in the order they came, which prevents complaints and frustration.

The same is seen at the BK ATM. Clients arrive and wait in line until it's their turn. In the program, they are served one after another until only the last client is left. If a stack (LIFO – Last In, First Out) was used instead, the last client to arrive would be the first one to get served, which would allow latecomers to jump the line. This would feel unfair to those who waited patiently, and it could even create conflicts or reduce people's trust in the system.

Even when distributing IDs, the difference between FIFO and LIFO is clear. In a queue, IDs are given in order—ID1, ID2, ID3, and so on. In a stack, however, the last ID put in is the first one given out, which means someone who registered late would get served before the person who registered early. In government services, this would be seen as injustice, because it rewards latecomers and disadvantages those who followed the rules and came on time.

In short, FIFO supports fairness because it treats everyone equally and respects the time they spent waiting. People can clearly see that the process is transparent, and this builds trust in institutions like RRA or BK. By following the principle of “first come, first served,” governments ensure justice, equality, and order in service delivery.