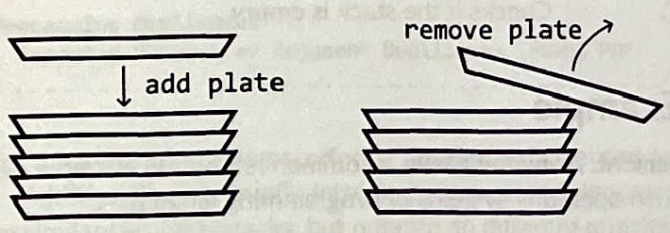
7

Stacks

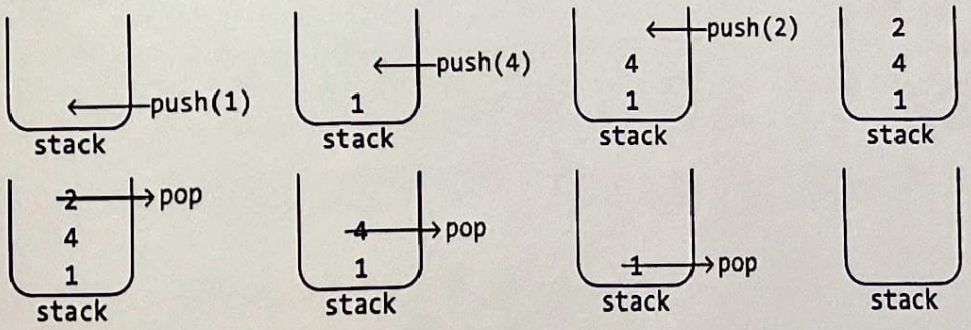
Introduction to Stacks

Imagine a stack of plates. You can only add a new plate to the top of the stack, and when you need a plate, you take the one from the top. It's not possible to take a plate from the bottom or middle without first removing all the plates above it.



This analogy encapsulates the essence of the stack data structure. Adding a plate to and taking a plate from the top of the stack, physically demonstrates the two main stack operations:

- **Push** (adds an element to the top of the stack).
- **Pop** (removes and returns the element at the top of the stack).



LIFO (Last-In-First-Out)

Stacks follow the LIFO principle, meaning the most recently added item is the first to be removed. This unique characteristic makes stacks particularly useful in various scenarios where the order of processing or removal is critical. Here are a few key applications:

- Handling nested structures: Stacks are a good option for parsing or validating nested structures such as nested parentheses in a string (e.g., "(())()"). They allow us to process the innermost nested structures first due to the LIFO principle.
- Reverse order: When elements are added (pushed) onto a stack and then removed (popped),

they come out in the reverse order of how they were added. This property is useful for reversing sequences.

- Substitute for recursion: Recursive algorithms use the recursive call stack to manage recursive calls. Ultimately, this recursive call stack is itself a stack. As such, we can often implement recursive functions iteratively using the stack data structure.
- Monotonic stacks: These special-purpose stacks maintain elements in a consistent, increasing or decreasing sorted order. Before adding a new element to the stack, any elements that break this order are removed from the top of the stack, ensuring the stack remains sorted.

Some examples of the above applications are explored in this chapter.

Below is a time complexity breakdown of common stack operations:

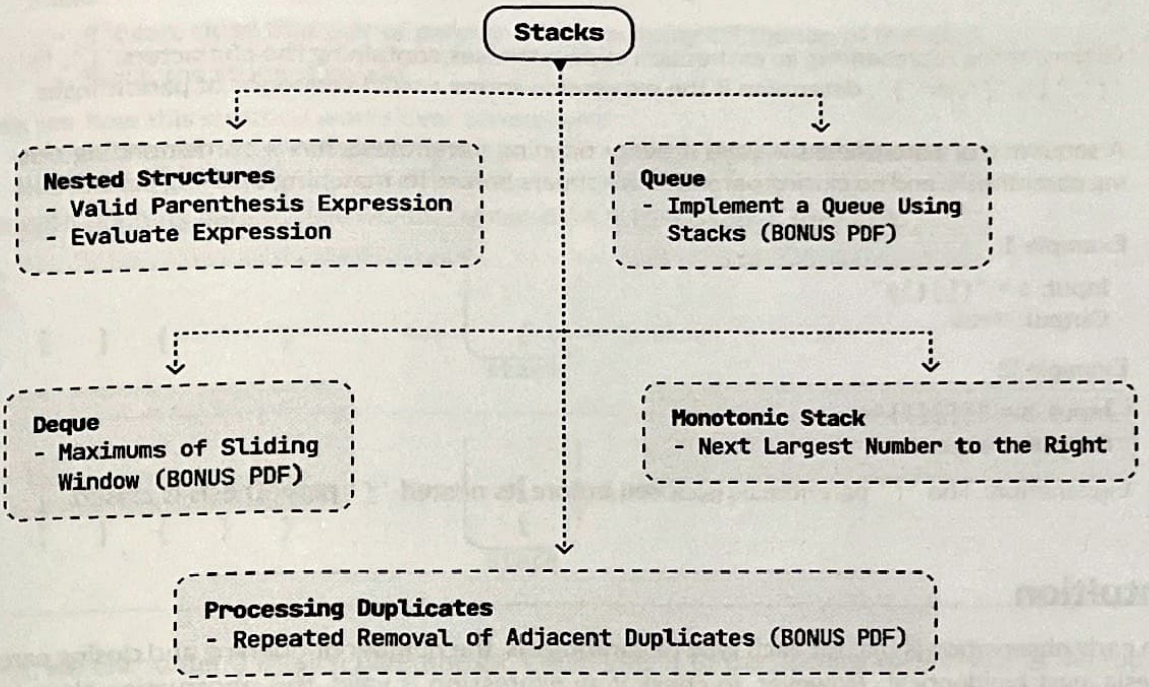
Operation	Worst case	Description
Push	$O(1)$	Adds an element to the top of the stack.
Pop	$O(1)$	Removes and returns the element at the top of the stack.
Peek	$O(1)$	Returns the element at the top of the stack without removing it.
IsEmpty	$O(1)$	Checks if the stack is empty.

Real-world Example

Function call management: As hinted above, a common real-world example of stacks is in function call management within operating systems or programming languages.

When a function is called, the program pushes the function's state (including its parameters, local variables, and the return address) onto the call stack. As functions call other functions, their states are also pushed onto the stack. When a function completes, its state is popped off the stack, and the program returns to the calling function. This stack-based approach ensures that functions return control in the correct order, managing nested or recursive function calls efficiently.

Chapter Outline



This chapter explores a variety of problems, offering detailed explanations for how to use stacks in problem solving. Additionally, we briefly introduce queues and deques, which are two data structures that share similarities with stacks, but operate on different principles.

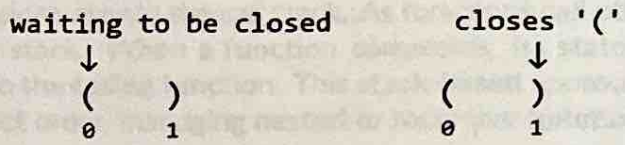
Valid Parenthesis Expression

> Given a string representing an expression of parentheses containing the characters `'('`, `')'`, `'['`, `']'`, `'{'`, or `'}'`, determine if the expression forms a valid sequence of parentheses.
>
> A sequence of parentheses is valid if every opening parenthesis has a corresponding closing parenthesis, and no closing parenthesis appears before its matching opening parenthesis.
>
> **Example 1:**
>
> Input: s = `"([]{})"`
> Output: `True`
>
> **Example 2:**
>
> Input: s = `"([]{)}"`
> Output: `False`
>
> Explanation: The `'('` parenthesis is closed before its nested `'{'` parenthesis is closed.

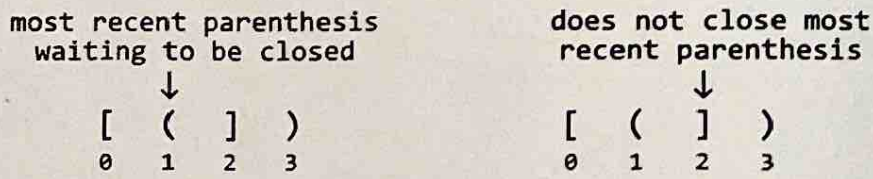
Intuition

An early observation is that for each type of parenthesis, the number of opening and closing parenthesis must be identical. However, to check if an expression is valid, this observation alone isn't enough. For example, the string `"())("` has the same number of opening and closing parentheses, but is still invalid. This means we need a way to account for the order of parentheses.

Consider the string `"()"`. The first parenthesis is opening, and we're waiting for it to be closing. Upon reaching the second parenthesis, the first parenthesis gets closed.



Now, consider the string `"[()]"`. When we reach index 1, we have two opening parentheses waiting to be closed. In particular, we expect `'('` to be closed before `'['`. The first closing parenthesis we encounter is `']'`, which does not close `'('`. Therefore, this string is invalid.



The key observation here is that **the most recent opening parenthesis we encounter should be the first parenthesis that gets closed**. So, opening parentheses are processed from most recent to least recent, which is indicative of a last-in-first-out (LIFO) dynamic. This leads to the idea that a **stack** can be used to solve this problem.

Stack

Here's a high-level strategy:

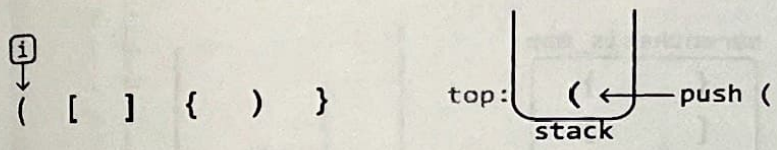
- Add each opening parenthesis we encounter to the stack. This way, the most recent parenthesis is always at the top of the stack.

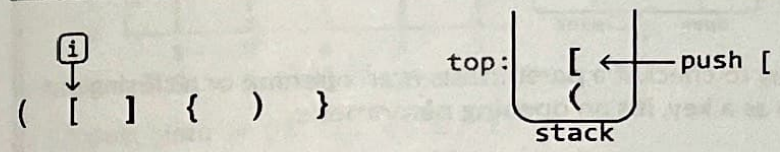
- When encountering a closing parenthesis, check if it can close the most recent opening parenthesis.
  - If it can, close that pair of parenthesis by popping off the top of the stack.
  - If not, the string is invalid.

Let's see how this strategy works over an example:

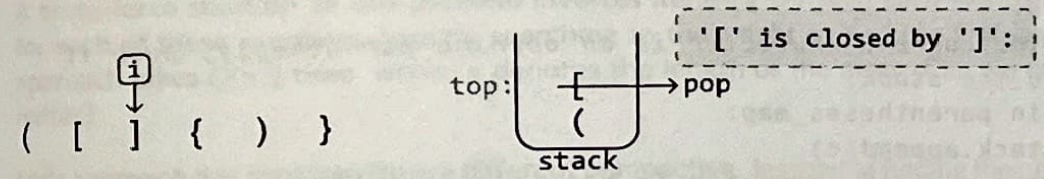
```
( [ ] { ) }
```

For each opening parenthesis we encounter, push it to the top of the stack:

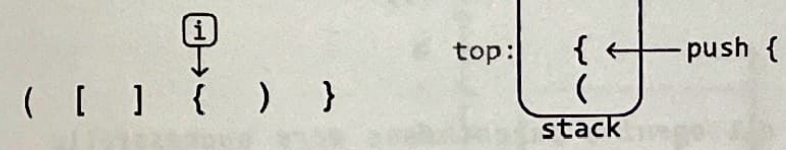




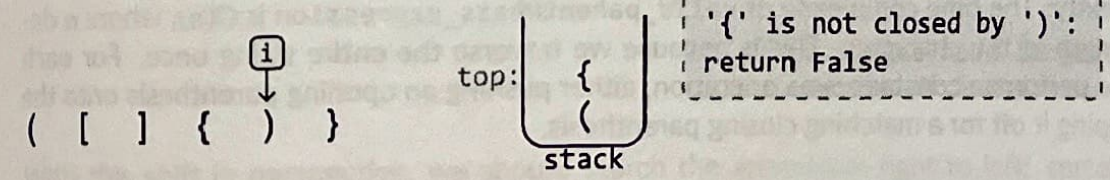
Next, we encounter a closing parenthesis. Comparing it to the opening parenthesis at the top of the stack, we see that it correctly closes that opening parenthesis. So, we can pop off the opening parenthesis at the top of the stack:



The next character is an opening parenthesis, which we just push to the top of the stack:



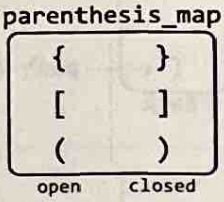
The next character is a closing parenthesis, ')', which does not close the opening parenthesis at the top of the stack, '{'. This means this parenthesis expression is invalid. As such, we return false.



If we've iterated over the entire string without returning false, that means we've accounted for all closing parentheses in the string.

Edge case: extra opening parentheses
We only check for invalidity at closing parenthesis, so we need to perform a final check to ensure there aren't any opening parentheses in the string left unclosed. This can be done by checking if the stack is empty after processing the whole input string, as a non-empty stack indicates opening parentheses remain in the stack.

Managing three types of parentheses
In our algorithm, we need a way to ensure we compare the correct types of opening and closing parentheses. We can use a **hash map** for this, which maps each type of opening parenthesis to its corresponding closing parenthesis:



This hash map can also be used as a way to check if a parenthesis is an opening or a closing one: if the parenthesis exists in this hash map as a key, it's an opening parenthesis.

Implementation

```
def valid_parenthesis_expression(s: str) -> bool:
    parentheses_map = {'(': ')', '{': '}', '[': ']'}
    stack = []
    for c in s:
        # If the current character is an opening parenthesis, push it
        # onto the stack.
        if c in parentheses_map:
            stack.append(c)
        # If the current character is a closing parenthesis, check if
        # it closes the opening parenthesis at the top of the stack.
        else:
            if stack and parentheses_map[stack[-1]] == c:
                stack.pop()
            else:
                return False
    # If the stack is empty, all opening parentheses were successfully
    # closed.
    return not stack
```

Complexity Analysis

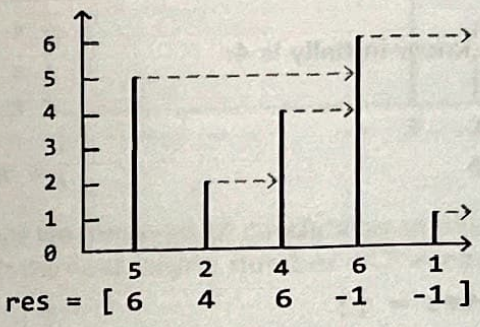
Time complexity: The time complexity of `valid_parenthesis_expression` is $O(n)$, where n denotes the length of the character. This is because we traverse the entire string once. For each character, we perform a constant-time operation, either pushing an opening parenthesis onto the stack or popping it off for a matching closing parenthesis.

Space complexity: The space complexity is $O(n)$ because the stack stores at most n characters, and the hash map takes up $O(1)$ space.

Next Largest Number to the Right

Given an integer array `nums`, return an output array `res` where, for each value `nums[i]`, `res[i]` is the first number to the right that's larger than `nums[i]`. If no larger number exists to the right of `nums[i]`, set `res[i]` to `-1`.

Example:

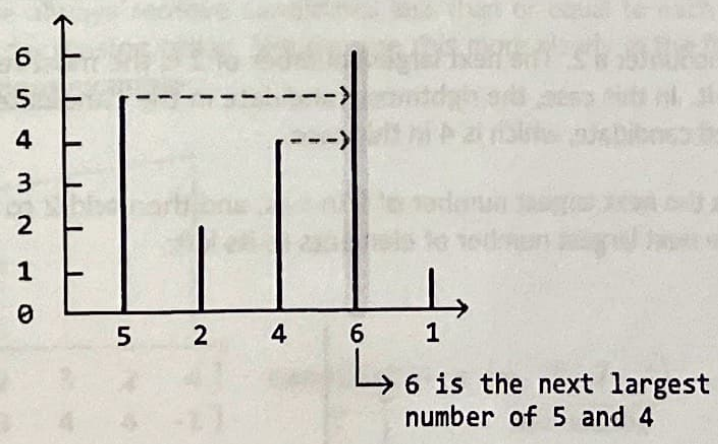


Input: `nums = [5, 2, 4, 6, 1]`
Output: `[6, 4, 6, -1, -1]`

Intuition

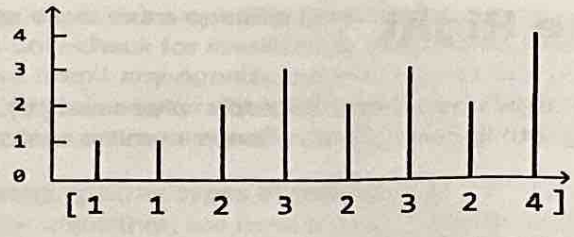
A brute-force solution to this problem involves iterating through each number in the array and, for each of these numbers, linearly searching to their right to find the first larger number. This approach takes $O(n^2)$ time, where n denotes the length of the array. Can we think of something better?

Let's approach this problem from a different perspective. Instead of finding the next largest number for each value, what if we **check whether the value itself is the next largest number for any value(s) to its left?** For example, can we figure out which values in the following example have 6 as their next largest number?:

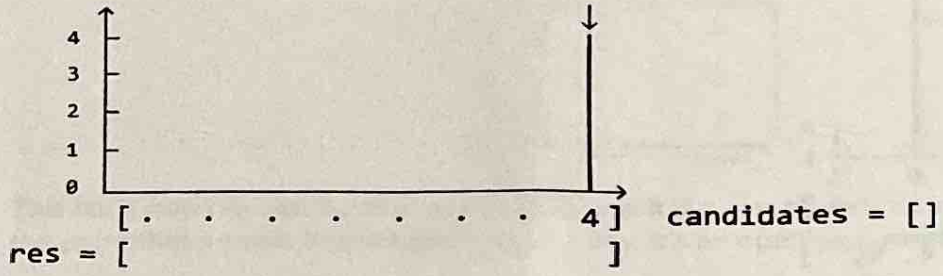


With this shift in perspective, we should search the array from right to left: certain values we encounter from the right could potentially be the next largest number of values to their left. Let's call these values "**candidates**." But how do we determine which numbers qualify as candidates?

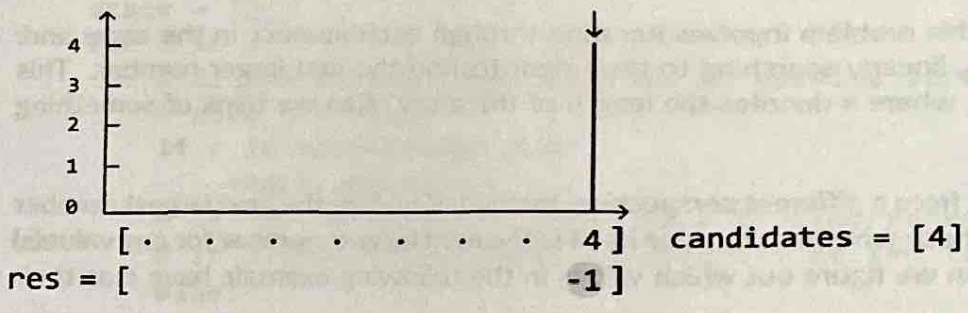
Consider the example below:



Let's start at the rightmost index, where the only value we know initially is 4:

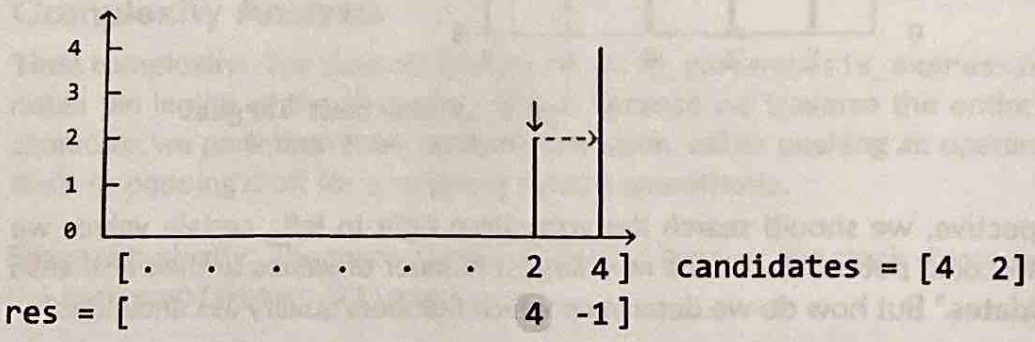


Right now, we can say 4 is a candidate as it might be the next largest number of values to its left. No candidates have been encountered before 4 because it's the rightmost element, so we should mark the result for 4 in `res` as -1:

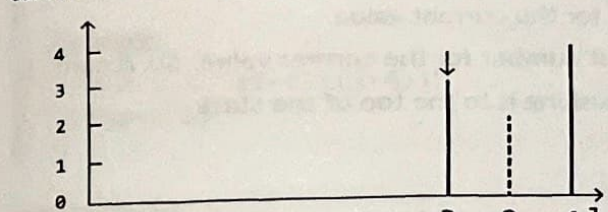


Next, we encounter a 2. The next largest number of 2 is the most recently added candidate that's larger than it. In this case, the rightmost candidate in the candidates list represents the most recently added candidate, which is 4 in this case.

Record 4 as the next largest number of 2 in `res`, and then add 2 to the candidates list because it could be the next largest number of elements to its left:

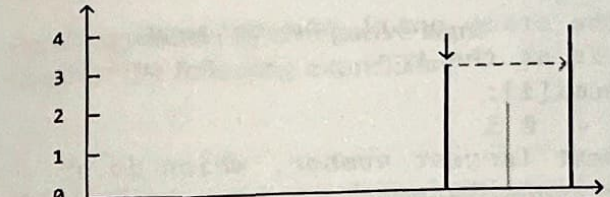


The next number is 3. Notice that with the introduction of 3, number 2 should no longer be considered a candidate. This is because it's now impossible for 2 to be the next largest number of any value to its left. Since 3 is both larger and further to the left in the array, it will always be prioritized over 2 as the next largest number. So, let's remove 2 from the candidates list, as well as any other candidate that's less than or equal to 3:




```
[ .  .  .  .  .  3  2  4 ]    remove all candidates ≤ 3:
res = [                       candidates = [4  ~~2~~]
                4  -1 ]
```

Since we removed all candidates smaller than 3, the rightmost candidate is now 4. So, let's record 4 as the next largest number of 3 in `res` and add 3 to the candidates list:

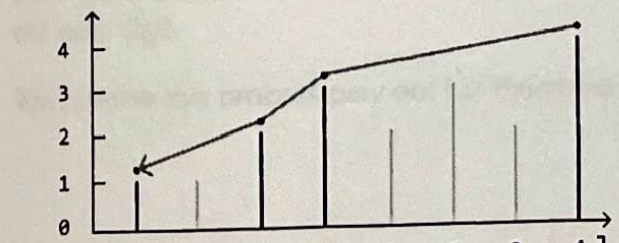



```
[ .  .  .  .  .  3  2  4 ]    candidates = [4  3]
res = [                 4  -1 ]
```

This provides a crucial insight:

> Whenever we move to a new number, **all candidates less than or equal to this number should be removed** from the candidates list.

Another key observation is that **the list of candidates always maintains a strictly decreasing order of values**. This is because we always remove candidates less than or equal to each new value, ensuring values are added in decreasing order. We can see this more clearly in the final state of the candidates list of the previous example:




```
[1  1  2  3  2  3  2  4 ]    candidates = [4  3  2  1]
res = [2  2  3  4  3  4  4  -1 ]                  decreasing →
```

This indicates a **stack** is the ideal data structure for storing the candidates list, since stacks can be used to efficiently maintain a **monotonic decreasing order** of values, as mentioned in the introduction.

The top of the stack represents the most recent candidate to the right of each new number encountered. Given this, here's how to use the stack to add and remove candidates at each value:

1. Pop off all candidates from the top of the stack less than or equal to the current value.
2. The top of the stack will then represent the next largest number of the current value.
   - Record the top of the stack as the answer for the current value.
   - If the stack is empty, there's no next largest number for the current value. So, record -1.
3. Add the current value as a new candidate by pushing it to the top of the stack.

Implementation

```
def next_largest_number_to_the_right(nums: List[int]) -> List[int]:
    res = [0] * len(nums)
    stack = []
    # Find the next largest number of each element, starting with the
    # rightmost element.
    for i in range(len(nums) - 1, -1, -1):
        # Pop values from the top of the stack until the current
        # value's next largest number is at the top.
        while stack and stack[-1] <= nums[i]:
            stack.pop()
        # Record the current value's next largest number, which is at
        # the top of the stack. If the stack is empty, record -1.
        res[i] = stack[-1] if stack else -1
        stack.append(nums[i])
    return res
```

Complexity Analysis

Time complexity: The time complexity of `next_largest_number_to_the_right` is $O(n)$. This is because each value of `nums` is pushed and popped from the stack at most once.

Space complexity: The space complexity is $O(n)$ because the stack can potentially store all n values.

Evaluate Expression

Given a string representing a mathematical expression containing integers, parentheses, addition, and subtraction operators, evaluate and return the result of the expression.

Example:

```
Input: s = "18-(7+(2-4))"
Output: 13
```

Intuition

At first, it might seem overwhelming to deal with expressions that include a variety of elements like negative numbers, nested expressions inside parentheses, and numbers with multiple digits. The key to managing this complexity is to break down the problem into smaller, more manageable parts. Let's first focus on evaluating simple expressions that contain no parentheses.

Handling positive and negative signs

Consider the following expression:

```
2 8 - 1 0 + 7
```

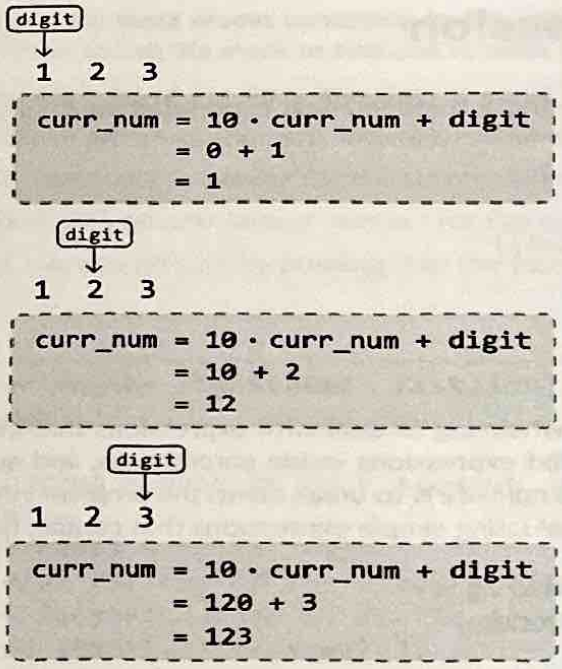
There's already some complexity in this expression with there being two signs to consider: plus and minus. An immediate simplification we can make is to **treat all expressions as ones of pure addition**. This is possible when we assign signs to each number (1 representing '+' and -1 representing '-'). This sign can be multiplied by the number to attain its correct value. This allows us to just focus on performing additions:

```
sign = 1   sign = -1   sign = 1
(+) 2 8    (-) 1 0     (+) 7
```

Processing numbers with multiple digits

Another complexity in this expression is that some numbers have multiple digits. We'll need a way to build numbers digit by digit until we reach the end of the number. We can build a number using the variable `curr_num`, which is initially set to 0. Every time we encounter a new digit, we multiply `curr_num` by 10 and add the new digit to it, effectively shifting all digits to the left and appending the new digit.

We can see this process play out for the string "123" in the following diagrams:



We can stop building this number once we encounter a non-digit character, indicating the end of the number.

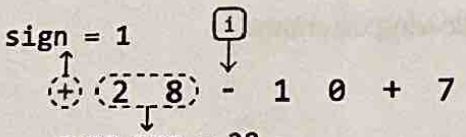
Evaluating an expression without parentheses

With the information from the section above, let's evaluate the following expression, which contains no parentheses. We'll start off with a sign of 1:

```
2 8 - 1 0 + 7,  sign = 1,  curr_num = 0
```

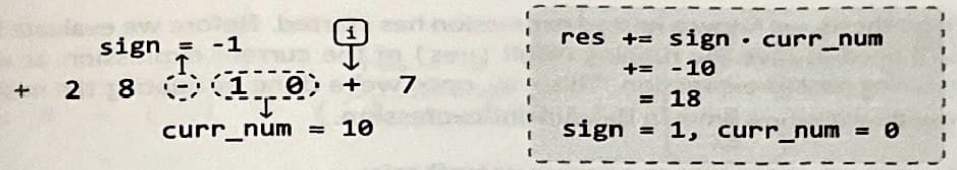
Upon reaching the '-' operator, we've reached the end of the first number (28). So, let's:

1. Multiply the current number (28) by its sign (1).
2. Add the resulting product (28) to the result.
3. Update the sign to -1 since the current operator is a minus sign.
4. Reset curr_num to 0 before building the next number.

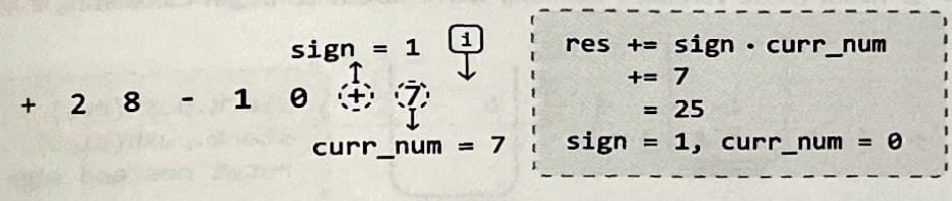



```
res += sign · curr_num
    += 28
     = 28
sign = -1, curr_num = 0
```

Once we reach the second operator, we multiply the current number (10) by its sign of -1 before adding the resulting product (-10) to the result. This effectively subtracts 10 from the result:



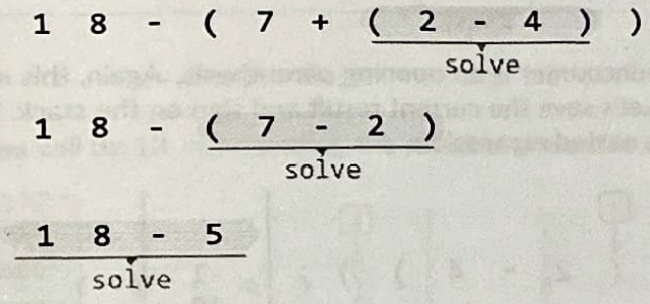
Finally, once we've reached the end of the string, we just add the final number (7) to the result after multiplying it by its sign of 1:



Evaluating expressions containing parentheses

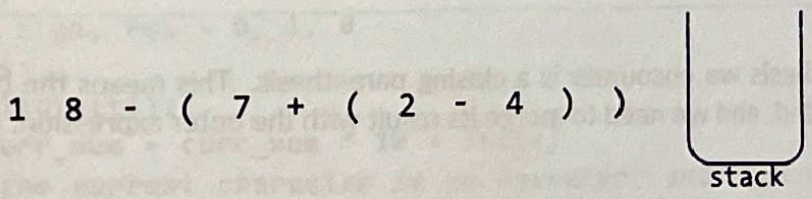
Now that we can solve simple expressions, it's time to bring parentheses into the discussion. Moving forward, we define a nested expression as one that's inside a pair of parentheses.

One challenge is that we need to evaluate the results of nested expressions before we can calculate the original expression. Once all nested expressions are evaluated, we can evaluate the original expression.



Consider another problem in this chapter that also contains parentheses: *Valid Parenthesis Expression*. In that problem, we used a **stack** to process nested parentheses in the right order. This suggests a stack might also help us evaluate nested expressions in the right order. Let's explore this idea further.

Similar to *Valid Parenthesis Expression*, an opening parenthesis '(' indicates the start of a new nested expression, whereas a closing parenthesis ')' indicates the end of one. Understanding this, let's try to use a stack to solve the following expression.

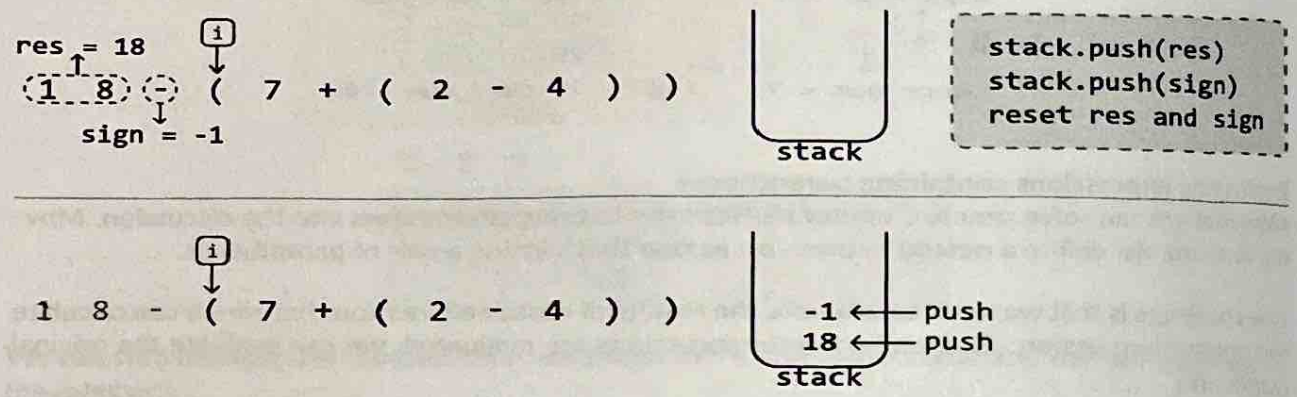


We already know how to evaluate expressions without parentheses, so let's just focus on what to do when we encounter a parenthesis.

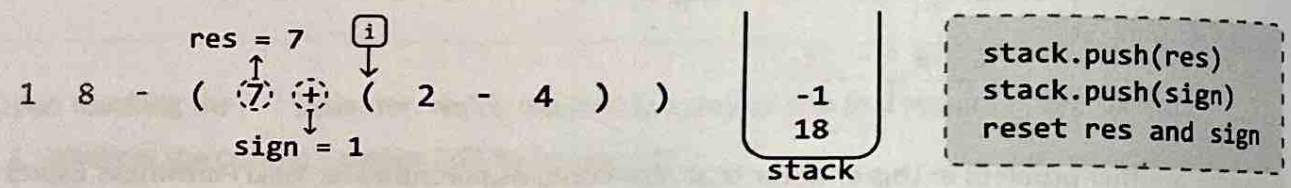
At the first opening parenthesis, we know a nested expression has started. Before we evaluate the nested expression, we'll need to save the running result (`res`) of the current expression, as well as the sign of this upcoming nested expression. This way, once we're done evaluating the nested expression, we can resume where we were in the current expression.

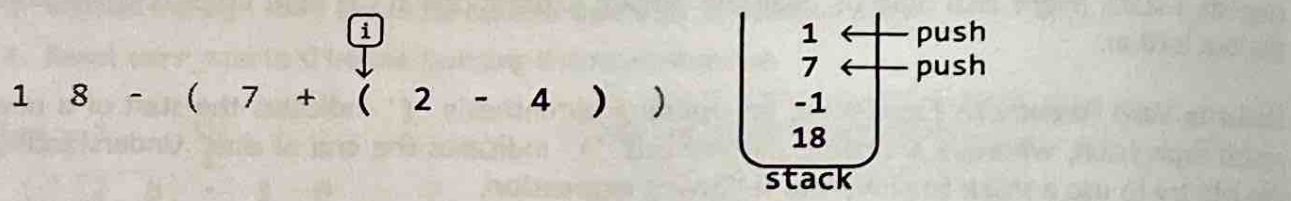
Here are the steps for when we encounter an opening parenthesis:

1. `stack.push(res)`: Save the running result on the stack.
2. `stack.push(sign)`: Save the sign of the upcoming nested expression on the stack.
3. `res = 0, sign = 1`: Reset these variables because we're about to begin calculating a new expression.



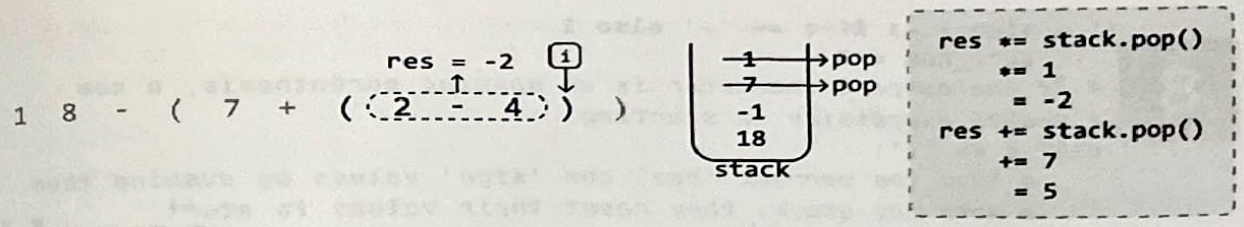
The next parenthesis we encounter is an opening parenthesis. Again, this indicates the start of a new nested expression. Let's save the current result and sign on the stack, before resetting them to evaluate the upcoming nested expression:



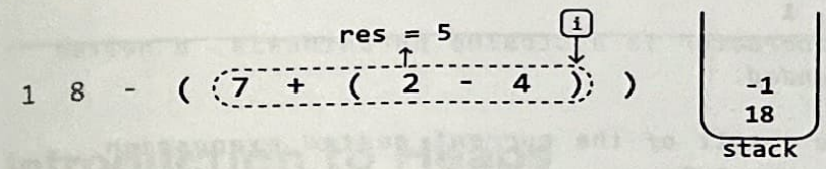


The next parenthesis we encounter is a closing parenthesis. This means the current nested expression just ended, and we need to merge its result with the outer expression. Here's how we do this:

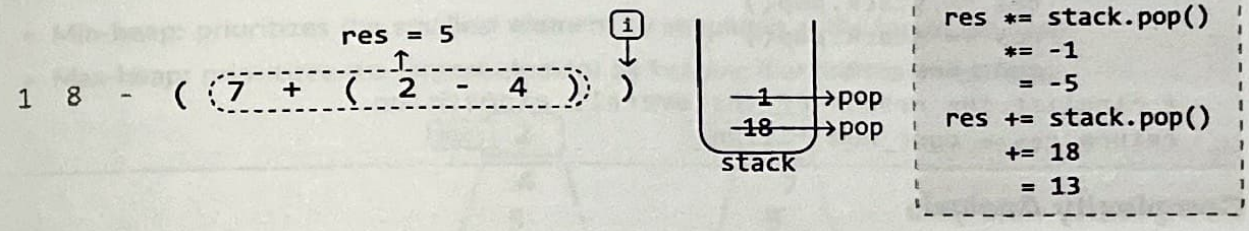
1. `res *= stack.pop()`: Apply the sign of the current nested expression to its result.
2. `res += stack.pop()`: Add the result of the outer expression to the result of the current nested expression.



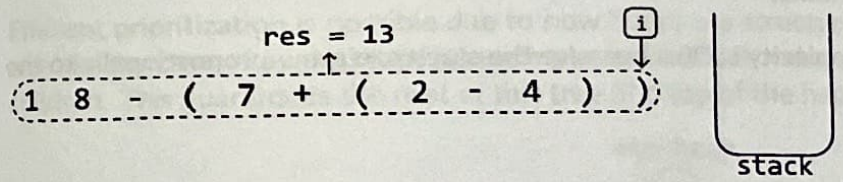
After applying those operations, the value of `res` will be 5, representing the result of the highlighted part of the expression below:



At the final closing parenthesis, we can apply the same steps:



Finally, the value of `res` will be 13, representing the result of the entire expression:



Now that we've reached the end of the string, we can return `res`.

Implementation

```
def evaluate_expression(s: str) -> int:
    stack = []
    curr_num, sign, res = 0, 1, 0
    for c in s:
        if c.isdigit():
            curr_num = curr_num * 10 + int(c)
        # If the current character is an operator, add 'curr_num' to
        # the result after multiplying it by its sign.
        elif c == '+' or c == '-':
            res += curr_num * sign
            # Update the sign and reset 'curr_num'.
```

```
            sign = -1 if c == '-' else 1
            curr_num = 0
        # If the current character is an opening parenthesis, a new
        # nested expression is starting.
        elif c == '(':
            # Save the current 'res' and 'sign' values by pushing them
            # onto the stack, then reset their values to start
            # calculating the new nested expression.
            stack.append(res)
            stack.append(sign)
            res, sign = 0, 1
        # If the current character is a closing parenthesis, a nested
        # expression has ended.
        elif c == ')':
            # Finalize the result of the current nested expression.
            res += sign * curr_num
            # Apply the sign of the current nested  expression's result
            # before adding this result to the result of the outer
            # expression.
            res *= stack.pop()
            res += stack.pop()
            curr_num = 0
    # Finalize the result of the overall expression.
    return res + curr_num * sign
```

Complexity Analysis

Time complexity: The time complexity of `evaluate_expression` is $O(n)$ because we traverse each character of the expression once, processing nested expressions using the stack, where each stack push or pop operation takes $O(1)$ time.

Space complexity: The space complexity is $O(n)$ because the stack can grow proportionally to the length of the expression.