Recursion I

Introduction

`Recursion` is an important concept in computer science. It is the foundation to many other algorithms and data structures. However, it can be tricky to grasp for many beginners.

Before getting started with this card, we strongly recommend that you complete the binary tree and stack Explore cards first.

In this Explore card, we answer the following questions:

1. What is `recursion`? How does it work?
2. How to solve a problem recursively?
3. How to analyze the `time` and `space complexity` of a recursive algorithm?
4. How can we apply `recursion` in a better way?

After completing this card, you will feel more confident in solving problems recursively and analyzing the complexity on your own.

Before you start, bear in mind that should you have any questions or comments, you can always post them in the Discussion forum that is located at the end of this card. We'll do our best to respond to you as soon as we can.

Principle of Recursion

- [ ] Principle of Recursion
- [x] Reverse String
- [ ] Solution - Reverse String
- [ ] Recursion Function
- [ ] Swap Nodes in Pairs

Recurrence Relation

- [ ] Recurrence Relation
- [ ] Pascal's Triangle
- [ ] Pascal's Triangle II
- [ ] Reverse Linked List
- [ ] Reverse Linked List

Memoization

- [ ] Duplicate Calculation in Recursion
- [ ] Fibonacci Number
- [ ] Climbing Stairs
- [ ] Climbing Stairs

Complexity Analysis

Time Complexity - Recursion	Space Complexity - Recursion
Tail Recursion	Maximum Depth of Binary Tree
Maximum Depth of Binary Tree	Pow(x, n)
Pow(x, n)	

Conclusion

Conclusion - Recursion I	Merge Two Sorted Lists
Merged Two Sorted Lists	K-th Symbol in Grammar
K-th Symbol in Grammar	Unique Binary Search Trees II
Unique Binary Search Trees II	

Principle of Recursion

In this chapter, we will:

- Explain the basic concept of recursion;
- Demonstrate how to apply the recursion to solve certain problems;
- Finally provide some exercises for you to practice recursion.

Recurrence Relation

In the previous chapter, we learned the basic concept of recursion.

There are two important things that you need to figure out before implementing a recursion function: `base case` and `recurrence relation`.

In this chapter, we will:

- Go through a detailed example on how to define the base case and recurrence relation;
- Then, we will have some exercises for you to practice with.

Memoization

In the previous chapter, we talked about the `duplicate calculation` problem in a recursion algorithm. In the best case, duplicate calculations would increase the time complexity of the algorithm, and in the worst case, it would lead to an infinite loop.

Therefore, in this chapter, we will:

- Start with an example and show you how duplicate calculations can occur;
- Show you how to avoid duplicate calculations using a technique called `memoization`.

Complexity Analysis

In this chapter, we will talk about how to estimate the `time` and `space` complexity of recursion algorithms.

In particular, we will present you a useful technique called `Tail Recursion`, which can be applied to optimize the space complexity of some recursion problems, and more importantly to avoid the problem of `stack overflow`.

Conclusion

In the previous chapters, we went through the concept and the principles of recursion.

As a reminder, here is the general workflow to solve a recursion problem:

1. Define the `recursion function`;
2. Write down the `recurrence relation` and `base case`;
3. Use `memoization` to eliminate the `duplicate calculation` problem, if it exists.
4. Whenever possible, implement the function as `tail recursion`, to optimize the space complexity.

In this chapter, we conclude on the recursion algorithms and provide you with more tips on how to solve some problems with recursion.

A Principle of Recursion

> Recursion is an approach to solving problems using a function that calls itself as a subroutine.

You might wonder how we can implement a function that calls itself. The trick is that each time a recursive function calls itself, it reduces the given problem into subproblems. The recursion call continues until it reaches a point where the subproblem can be solved without further recursion.

A recursive function should have the following properties so that it does not result in an infinite loop:

1. A simple `base case` (or cases) — a terminating scenario that does not use recursion to produce an answer.
2. A set of rules, also known as `recurrence relation` that reduces all other cases towards the base case.

Note that there could be multiple places where the function may call itself.

Example

Let's start with a simple programming problem:

> Print a string in reverse order.

You can easily solve this problem iteratively, *i.e.* looping through the string starting from its last character. But how about solving it recursively?

You can easily solve this problem iteratively, *i.e.* looping through the string starting from its last character. But how about solving it recursively?

First, we can define the desired function as `printReverse(str[0...n-1])`, where `str[0]` represents the first character in the string. Then we can accomplish the given task in two steps:

1. `printReverse(str[1...n-1])`: print the substring `str[1...n-1]` in reverse order.
2. `print(str[0])`: print the first character in the string.

Notice that we call the function itself in the first step, which by definition makes the function recursive.

Here is the code snippet:

C++ | Java

Copy

```java
private static void printReverse(char [] str) {
  helper(0, str);
}

private static void helper(int index, char [] str) {
  if (str == null || index >= str.length) {
    return;
  }
  helper(index + 1, str);
  System.out.print(str[index]);
}
```

Next, you will find an exercise that is slightly different from the above example. You should try to solve it using recursion.

Note: For this exercise, we also provide a detailed solution in this Explore chapter.

Reverse String

Go to Discuss

Write a function that reverses a string. The input string is given as an array of characters `char[]`.

Do not allocate extra space for another array, you must do this by **modifying the input array** in-place with O(1) extra memory.

You may assume all the characters consist of printable ascii characters.

Example 1:

```
Input: ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]
```

Example 2:

```
Input: ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]
```

Python

```python
class Solution(object):
    def reverseString(self, s):
        """
        :type s: List[str]
        :rtype: None Do not return anything, modify s in-place instead.
        """
        for i in range(len(s)/2):
            temp = s[i]
            s[i] = s[-i-1]
            s[-i-1] = temp

class Solution(object):
    def reverseString(self, s):
        l = len(s)
        if l < 2:
            return s
        return self.reverseString(s[l/2:]) + self.reverseString(s[:l/2])


class SolutionPythonic(object):
    def reverseString(self, s):
        return s[::-1]
```

A Solution - Reverse String

In this article, we present a sample solution for the problem of Reverse String.

> The problem is not difficult, yet the trick part is that we have an additional **constraint** for the problem, *i.e.* one must modify the string with $\mathcal{O}(1)$ extra space.

Let's define the problem as the function `reverseString(str[0...n-1])`, where `str[0...n-1]` is a list of characters with the first character denoted as `str[0]`.

Below, we will discuss how we can solve this problem with recursion.

First Attempt

If we follow the idea of the problem of printing a string in reversed order, as we presented in the first article of this card, we might come up with the following algorithm:

1. take the leading character `str[0]` from the input string.
2. call the function itself on the remaining substring, *i.e.* `reverseString(str[1...n-1])`.
3. then append the leading character to the result returned in the step (2).

The above algorithm could work, except that it does not meet the constraint imposed on the problem. This is because one would need to keep the intermediate result in step **(2)** which is proportional to the input string (*i.e.* with at least $\mathcal{O}(N)$ space complexity), which in no case could satisfy the constraint (use $\mathcal{O}(1)$ space to modify the string)

Another Divide-and-Conquer Solution

Looking closer at the constraint imposed by the problem, if we put it into the context of recursion, we could interpret it as not having additional space consumption between two consecutive recursive calls, *i.e.* we should divide the problem into independent subproblems.

So one of the ideas about how to divide the problem would be reducing the input string at each step into two components: *1).* the leading and trailing characters. *2).* the remaining substring without the leading and trailing characters. We then can solve the two components independently from each other.

Following the above idea, we could come up the algorithm as follows:

1. Take the leading and trailing characters from the input string, *i.e.* `str[0]` and `str[n-1].`
2. Swap the leading and trailing characters in place.
3. Call the function recursively to reverse the remaining substring, *i.e.* `reverseString(str[1...n-2])`.

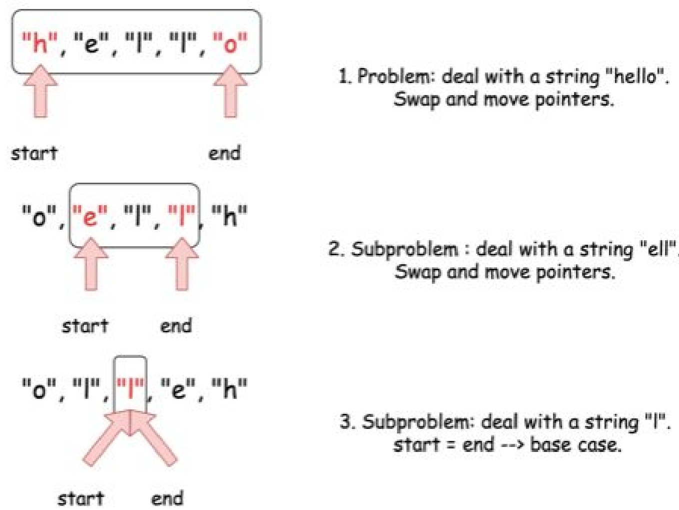
Note that you can actually swap the order of steps *(2)* and *(3)*, since they are independent tasks. Yet, it is better to keep them in this order, since this way we can use the optimization called tail recursion. We'll shed more light on tail recursion in later chapters.

Here is an implementation of the above algorithm.

Java Python3 Copy

```python
class Solution:
    def reverseString(self, s):
        """
        :type s: List[str]
        :rtype: void Do not return anything, modify s in-place instead.
        """
        def helper(start, end, ls):
            if start >= end:
                return

            # swap the first and last element
            ls[start], ls[end] = ls[end], ls[start]

            return helper(start+1, end-1, ls)

        helper(0, len(s)-1, s)
```

Given the input string `["h", "e", "l", "l", "o"]`, we illustrate how it can be divided and solved:



As one can see, we only need a constant memory in each recursive call in order to swap the leading and trailing characters. As a result, it meets the constraint of the problem.

A Recursion Function

For a problem, if there exists a recursive solution, we can follow the guidelines below to implement it.

For instance, we define the problem as the function $F(X)$ to implement, where X is the input of the function which also defines the scope of the problem.

Then, in the function $F(X)$, we will:

1. Break the problem down into smaller scopes, such as $x_0 \in X, x_1 \in X, ..., x_n \in X$;
2. Call function $F(x_0), F(x_1), ..., F(x_n)$ ***recursively*** to solve the subproblems of X;
3. Finally, process the results from the recursive function calls to solve the problem corresponding to X.

Example

To showcase the above guidelines, we give another example on how to solve a problem recursively.

> Given a linked list, swap every two adjacent nodes and return its head.
>
> *e.g.* for a list 1-> 2 -> 3 -> 4, one should return the head of list as 2 -> 1 -> 4 -> 3.

We define the function to implement as `swap(head)`, where the input parameter `head` refers to the head of a linked list. The function should return the `head` of the new linked list that has any adjacent nodes swapped.

Following the guidelines we lay out above, we can implement the function as follows:

1. First, we swap the first two nodes in the list, *i.e.* `head` and `head.next`;
2. Then, we call the function self as `swap(head.next.next)` to swap the rest of the list following the first two nodes.
3. Finally, we attach the returned head of the sub-list in step (2) with the two nodes swapped in step (1) to form a new linked list.

As an exercise, you can try to implement the solution using the steps we provided above. Click on "Swap Nodes in Pairs" to try to implement the solution yourself.

Swap Nodes in Pairs

Go to Discuss

Given a linked list, swap every two adjacent nodes and return its head.

You may **not** modify the values in the list's nodes, only nodes itself may be changed.

Example:

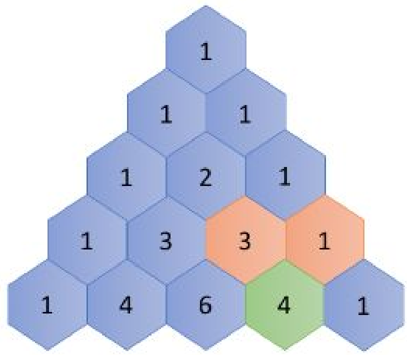
```
Given 1->2->3->4, you should return the list as 2->1->4->3.
```

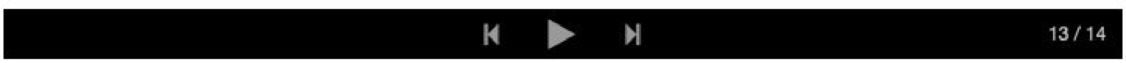
```python
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def swapPairs(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        # Iteratively
        dummy = p = ListNode(0)
        dummy.next = head
        while head and head.next:
            tmp = head.next
            head.next = tmp.next
            tmp.next = head
            p.next = tmp
            head = head.next
            p = tmp.next
        return dummy.next

    # Recursively
    def swapPairs(self, head):
        if head and head.next:
            tmp = head.next
            head.next = self.swapPairs(tmp.next)
            tmp.next = head
            return tmp
        return head
```

Custom Testcase (Contribute)

Run Code Submit Solution

Here's the illustration of the Pascal's Triangle with 5 rows:





Recurrence Relation

Let's start with the recurrence relation within the Pascal's Triangle.

First of all, we define a function $f(i, j)$ which returns the number in the Pascal's Triangle in the `i-th` row and `j-th` column.

We then can represent the recurrence relation with the following formula:

$$f(i, j) = f(i-1, j-1) + f(i-1, j)$$

Base Case

As one can see, the leftmost and rightmost numbers of each row are the `base cases` in this problem, which are always equal to 1.

As a result, we can define the base case as follows:

$$f(i, j) = 1 \quad where \quad j = 1 \ or \ j = i$$

Demo

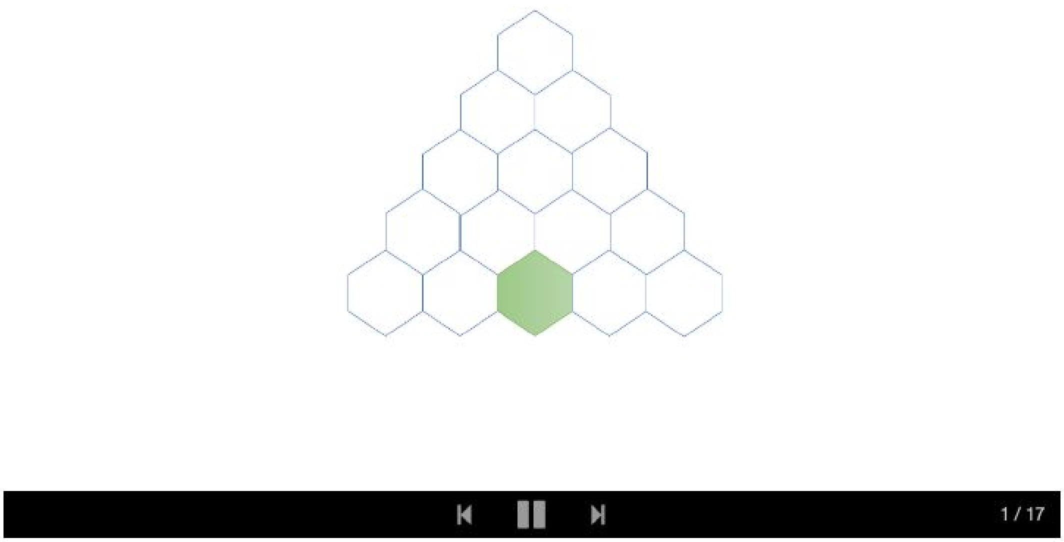
As one can see, once we define the `recurrence relation` and the `base case`, it becomes much more intuitive to implement the recursive function, especially when we formulate these two elements in terms of mathematical formulas.

Here is an example of how we can apply the formula to recursively calculate $f(5, 3)$, *i.e.* the `3rd` number in the `5th` row of the Pascal Triangle:

Demo

As one can see, once we define the `recurrence relation` and the `base case`, it becomes much more intuitive to implement the recursive function, especially when we formulate these two elements in terms of mathematical formulas.

Here is an example of how we can apply the formula to recursively calculate $f(5, 3)$, *i.e.* the `3rd` number in the `5th` row of the Pascal Triangle:



Starting from $f(5,3)$, we can break it down as $f(5,3) = f(4,2) + f(4,3)$, we then call $f(4,2)$ and $f(4,3)$ recursively:

- For the call of $f(4,2)$, we could extend it further until we reach the base cases, as follows:

$$f(4,2) = f(3,1) + f(3,2) = f(3,1) + (f(2,1) + f(2,2)) = 1 + (1 + 1) = 3$$

- For the call of $f(4,3)$, similarly we break it down as:

$$f(4,3) = f(3,2) + f(3,3) = (f(2,1) + f(2,2)) + f(3,3) = (1 + 1) + 1 = 3$$

- Finally we combine the results of the above subproblems:

$$f(5,3) = f(4,2) + f(4,3) = 3 + 3 = 6$$

Next

In the above example, you might have noticed that the recursive solution can incur some duplicate calculations, *i.e.* we compute the same intermediate numbers repeatedly in order to obtain numbers in the last row. For instance, in order to obtain the result for the number $f(5,3)$, we calculate the number $f(3,2)$ twice both in the calls of $f(4,2)$ and $f(4,3)$.

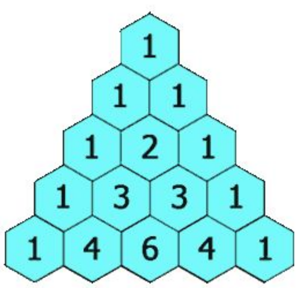
We will discuss how to avoid these `duplicate calculations` in the next chapter of this Explore card.

Following this article, you will find exercises for problems related to Pascal's Triangle.

Pascal's Triangle

Go to Discuss

Given a non-negative integer *numRows*, generate the first *numRows* of Pascal's triangle.



In Pascal's triangle, each number is the sum of the two numbers directly above it.

Example:

```
Input: 5
Output:
[
     [1],
    [1,1],
   [1,2,1],
  [1,3,3,1],
 [1,4,6,4,1]
]
```

Reverse Linked List

Go to Discuss

Reverse a singly linked list.

Example:

```
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL
```

Follow up:

A linked list can be reversed either iteratively or recursively. Could you implement both?

Python

```python
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def reverseList(self, head):
        prev = None
        curr = head

        while curr:
            next = curr.next
            curr.next = prev
            prev = curr
            curr = next

        return prev
```

Solution

Approach #1 (Iterative) [Accepted]

Assume that we have linked list `1 → 2 → 3 → Ø`, we would like to change it to `Ø ← 1 ← 2 ← 3`.

While you are traversing the list, change the current node's next pointer to point to its previous element. Since a node does not have reference to its previous node, you must store its previous element beforehand. You also need another pointer to store the next node before changing the reference. Do not forget to return the new head reference at the end!

```
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode curr = head;
    while (curr != null) {
        ListNode nextTemp = curr.next;
        curr.next = prev;
        prev = curr;
        curr = nextTemp;
    }
    return prev;
}
```

Complexity analysis

- Time complexity : $O(n)$. Assume that n is the list's length, the time complexity is $O(n)$.
- Space complexity : $O(1)$.

Approach #2 (Recursive) [Accepted]

The recursive version is slightly trickier and the key is to work backwards. Assume that the rest of the list had already been reversed, now how do I reverse the front part? Let's assume the list is: $n_1 \to \dots \to n_{k-1} \to n_k \to n_{k+1} \to \dots \to n_m \to Ø$

Assume from node n_{k+1} to n_m had been reversed and you are at node n_k.

$n_1 \to \dots \to n_{k-1} \to \mathbf{n_k} \to n_{k+1} \leftarrow \dots \leftarrow n_m$

We want n_{k+1}'s next node to point to n_k.

So,

n_k.next.next = n_k;

Be very careful that n_1's next must point to Ø. If you forget about this, your linked list has a cycle in it. This bug could be caught if you test your code with a linked list of size 2.

```
public ListNode reverseList(ListNode head) {
    if (head == null || head.next == null) return head;
    ListNode p = reverseList(head.next);
    head.next.next = head;
    head.next = null;
    return p;
}
```

Complexity analysis

- Time complexity : $O(n)$. Assume that n is the list's length, the time complexity is $O(n)$.
- Space complexity : $O(n)$. The extra space comes from implicit stack space due to recursion. The recursion could go up to n levels deep.

A Duplicate Calculation in Recursion

Recursion is often an intuitive and powerful way to implement an algorithm. However, it might bring some undesired penalty to the performance, *e.g.* duplicate calculations, if we do not use it wisely. For instance, at the end of the previous chapter, we have encountered the duplicate calculations problem in Pascal's Triangle, where some intermediate results are calculated multiple times.

In this article we will look closer into the duplicate calculations problem that could happen with recursion. We will then propose a common technique called `memoization` that can be used to avoid this problem.

To demonstrate another problem with duplicate calculations, let's look at an example that most people might be familiar with, the Fibonacci number. If we define the function `F(n)` to represent the Fibonacci number at the index of `n`, then you can derive the following recurrence relation:

`F(n) = F(n - 1) + F(n - 2)`

with the base cases:

`F(0) = 0, F(1) = 1`

Given the definition of a Fibonacci number, one can implement the function as follows:

Java Python Copy

```python
def fibonacci(n):
    """
    :type n: int
    :rtype: int
    """
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

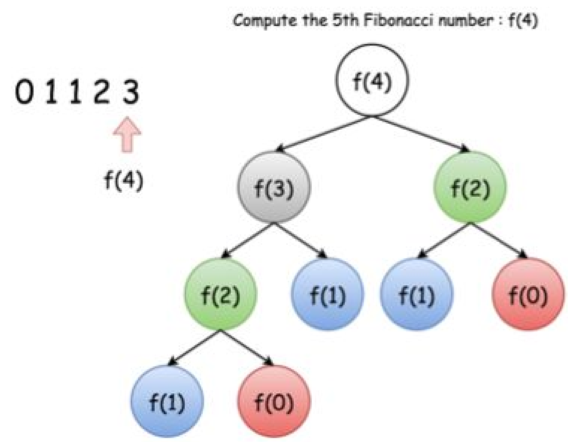
Now, if you would like to know the number of `F(4)`, you can apply and extend the above formulas as follows:

Now, if you would like to know the number of `F(4)`, you can apply and extend the above formulas as follows:

`F(4) = F(3) + F(2) = (F(2) + F(1)) + F(2)`

As you can see, in order to obtain the result for `F(4)`, we would need to calculate the number `F(2)` twice following the above deduction: the first time in the first extension of `F(4)` and the second time for the intermediate result `F(3)`.

Here is the tree that shows all the duplicate calculations (grouped by colors) that occur during the calculation of `F(4)`.



Memoization

To eliminate the duplicate calculation in the above case, as many of you would have figured out, one of the ideas would be to **store** the intermediate results in the cache so that we could reuse them later without re-calculation.

This idea is also known as *memoization*, which is a technique that is frequently used together with recursion.

> Memoization is an optimization technique used primarily to **speed up** computer programs by **storing** the results of expensive function calls and returning the cached result when the same inputs occur again. (Source: wikipedia)

Back to our Fibonacci function `F(n)`. We could use a hash table to keep track of the result of each `F(n)` with `n` as the key. The hash table serves as a cache that saves us from duplicate calculations. The memoization technique is a good example that demonstrates how one can reduce compute time in exchange for some additional space.

For the sake of comparison, we provide the implementation of Fibonacci number solution with memoization below.

As an exercise, you could try to make memoization more general and non-intrusive, *i.e.* applying memoization without changing the original function. (*Hint*: one can refer to a design pattern called **decorator**).

Java | Python | Copy

```python
def fib(self, N):
    """
    :type N: int
    :rtype: int
    """
    cache = {}
    def recur_fib(N):
        if N in cache:
            return cache[N]

        if N < 2:
            result = N
        else:
            result = recur_fib(N-1) + recur_fib(N-2)

        # put result in cache for later reference.
        cache[N] = result
        return result

    return recur_fib(N)
```

Following this article, we provide the Fibonacci number problem and another classic problem called climbing stairs, which could be really fun and challenging to solve.

In the next chapter, we will dive a bit into the complexity analysis of recursion algorithms.

A Climbing Stairs

Summary

You are climbing a stair case. It takes n steps to reach to the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

A Time Complexity - Recursion

In this article, we will focus on how to calculate the time complexity for recursion algorithms.

> Given a recursion algorithm, its time complexity $\mathcal{O}(T)$ is typically the product of **the number of recursion *invocations*** (denoted as R) and **the time complexity of calculation** (denoted as $\mathcal{O}(s)$) that incurs along with each recursion call:
>
> $$\mathcal{O}(T) = R * \mathcal{O}(s)$$

Let's take a look at some examples below.

Example

As you might recall, in the problem of printReverse, we are asked to print the string in the reverse order. A recurrence relation to solve the problem can be expressed as follows:

`printReverse(str) = printReverse(str[1...n]) + print(str[0])`

where `str[1...n]` is the substring of the input string `str`, without the leading character `str[0]`.

As you can see, the function would be recursively invoked `n` times, where `n` is the size of the input string. At the end of each recursion, we simply print the leading character, therefore the time complexity of this particular operation is constant, *i.e.* $\mathcal{O}(1)$.

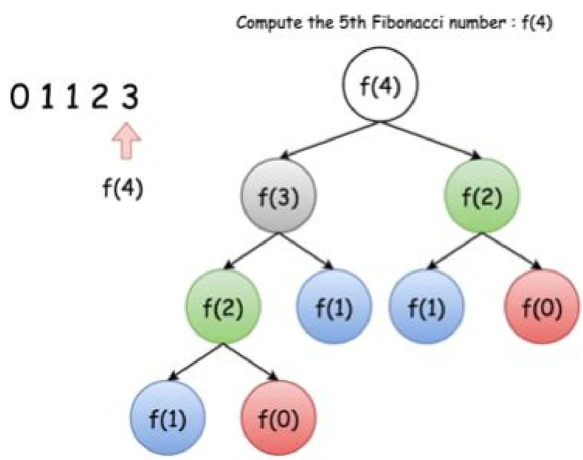
To sum up, the overall time complexity of our recursive function `printReverse(str)` would be $\mathcal{O}(printReverse) = n * \mathcal{O}(1) = \mathcal{O}(n)$.

Execution Tree

For recursive functions, it is rarely the case that the number of recursion calls happens to be linear to the size of input. For example, one might recall the example of Fibonacci number that we discussed in the previous chapter, whose recurrence relation is defined as `f(n) = f(n-1) + f(n-2)`. At first glance, it does not seem straightforward to calculate the number of recursion invocations during the execution of the Fibonacci function.

> In this case, it is better resort to the ***`execution tree`***, which is a tree that is used to denote the execution flow of a recursive function in particular. Each node in the tree represents an invocation of the recursive function. Therefore, the total number of nodes in the tree corresponds to the number of recursion calls during the execution.

The execution tree of a recursive function would form an `n-ary tree`, with `n` as the number of times recursion appears in the recurrence relation. For instance, the execution of the Fibonacci function would form a ***binary tree***, as one can see from the following graph which shows the execution tree for the calculation of Fibonacci number `f(4)`.



In a full binary tree with `n` levels, the total number of nodes would be $2^n - 1$. Therefore, the upper bound (though not tight) for the number of recursion in `f(n)` would be $2^n - 1$, as well. As a result, we can estimate that the time complexity for `f(n)` would be $\mathcal{O}(2^n)$.

Memoization

In the previous chapter, we discussed the technique of memoization that is often applied to optimize the time complexity of recursion algorithms. By caching and reusing the intermediate results, memoization can greatly reduce the number of recursion calls, *i.e.* reducing the number of branches in the execution tree. One should take this reduction into account when analyzing the time complexity of recursion algorithms with memoization.

Let's get back to our example of Fibonacci number. With memoization, we save the result of Fibonacci number for each index `n`. We are assured that the calculation for each Fibonacci number would occur only once. And we know, from the recurrence relation, the Fibonacci number `f(n)` would depend on all `n-1` precedent Fibonacci numbers. As a result, the recursion to calculate `f(n)` would be invoked `n-1` times to calculate all the precedent numbers that it depends on.

Now, we can simply apply the formula we introduced in the beginning of this chapter to calculate the time complexity, which is $\mathcal{O}(1) * n = \mathcal{O}(n)$. Memoization not only optimizes the time complexity of algorithm, but also simplifies the calculation of time complexity.

In the next article, we will talk about how to evaluate the space complexity of recursion algorithms.

A Space Complexity - Recursion

In this article, we will talk about how to analyze the space complexity of a recursion algorithm.

> There are mainly two parts of the space consumption that one should bear in mind when calculating the space complexity of a recursion algorithm: `recursion related` and `non-recursion related space.`

Recursion Related Space

The recursion related space refers to the memory cost that is incurred directly by the recursion, *i.e.* the stack to keep track of recursive function calls. In order to complete a typical function call, the system should allocate some space in the stack to hold three important pieces of information:

1. the returning address of the function call. Once the function call is completed, the program should know where to return to, *i.e.* the point before the function call;
2. the parameters that are passed to the function call;
3. the local variables within the function call.

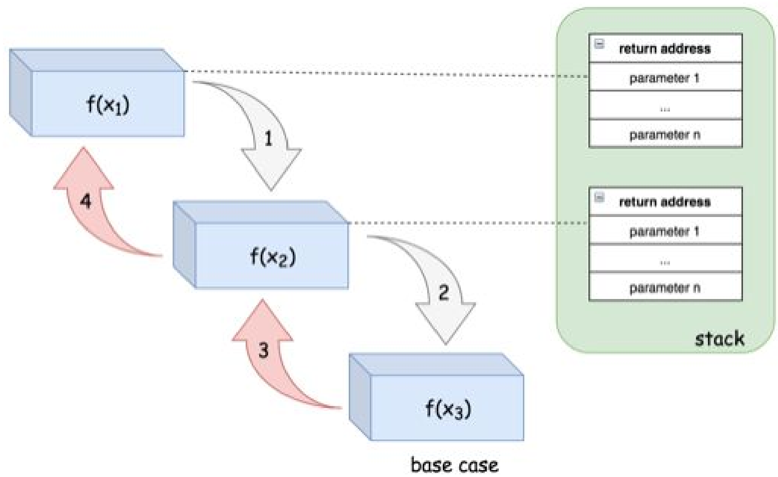
This space in the stack is the minimal cost that is incurred during a function call. However, once the function call is done, this space would be freed.

For recursion algorithms, the function calls would chain up successively until they reach a `base case` (*a.k.a.* bottom case). This implies that the space that is used for each function call would also accumulate.

> For a recursion algorithm, if there is no other memory consumption, then this recursion incurred space would be the space upper-bound of the algorithm.

For example, in the exercise of printReverse, we don't have extra memory usage outside the recursion call, since we simply print a character. For each recursion call, let's assume it could take certain space up to a constant value. And the recursion calls would chain up to `n` times, where `n` is the size of the input string. So the space complexity of this recursion algorithm would be $\mathcal{O}(n)$.

To illustrate this, for a sequence of recursion calls $f(x_1) \rightarrow f(x_2) \rightarrow f(x_3)$, we show the sequence of execution steps along with the layout of stack:



A space in the stack would be allocated for $f(x_1)$ in order to call $f(x_2)$. Similarly in $f(x_2)$, the system would allocate another space for the call to $f(x_3)$. Finally in $f(x_3)$, we reach the base case, therefore there is no further recursion call within $f(x_3)$.

Tail Recursion

In the previous article, we talked about the implicit extra space incurred on the system stack due to recursion calls. However, you should learn to identify a special case of recursion called tail recursion, which is **exempted** from this space overhead.

> **Tail recursion** is a recursion where the recursive call is the final instruction in the recursion function. And there should be only **one** recursive call in the function.

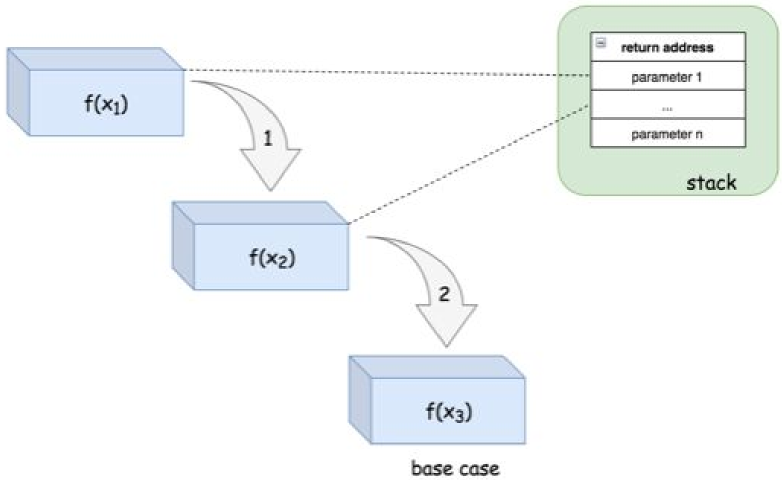
We have already seen an example of tail recursion in the solution of Reverse String. Here is another example that shows the difference between non-tail-recursion and tail-recursion. Notice that in the non-tail-recursion example there is an extra computation after the very last recursive call.

Java Python Copy

```python
def sum_non_tail_recursion(ls):
    """
    :type ls: List[int]
    :rtype: int, the sum of the input list.
    """
    if len(ls) == 0:
        return 0

    # not a tail recursion because it does some computation after the recursive call returned.
    return ls[0] + sum_non_tail_recursion(ls[1:])


def sum_tail_recursion(ls):
    """
    :type ls: List[int]
    :rtype: int, the sum of the input list.
    """
    def helper(ls, acc):
        if len(ls) == 0:
            return acc
        # this is a tail recursion because the final instruction is a recursive call.
        return helper(ls[1:], ls[0] + acc)

    return helper(ls, 0)
```

The benefit of having tail recursion is that it could avoid the accumulation of stack overheads during the recursive calls, since the system could reuse a fixed amount space in the stack for each recursive call.

For example, for the sequence of recursion calls `f(x1) -> f(x2) -> f(x3)`, if the function `f(x)` is implemented as tail recursion, then here is the sequence of execution steps along with the layout of the stack:



Note that in tail recursion, we know that as soon as we return from the recursive call we are going to immediately return as well, so we can skip the entire chain of recursive calls returning and return straight to the original caller. That means we don't need a call stack at all for all of the recursive calls, which saves us space.

For example, in step (1), a space in the stack would be allocated for `f(x1)` in order to call `f(x2)`. Then in step (2), the function `f(x2)` would recursively call `f(x3)`. However, instead of allocating new space on the stack, the system could simply reuse the space allocated earlier for this second recursion call. Finally, in the function `f(x3)`, we reach the base case, and the function could simply return the result to the original caller without going back to the previous function calls.

A tail recursion function can be executed as non-tail-recursion functions, *i.e.* with piles of call stacks, without impact on the result. Often, the compiler recognizes tail recursion pattern, and optimizes its execution. However, not all programming languages support this optimization. For instance, C, C++ support the optimization of tail recursion functions. On the other hand, Java and Python do not support tail recursion optimization.

Maximum Depth of Binary Tree

Go to Discuss

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: A leaf is a node with no children.

Example:

Given binary tree `[3,9,20,null,null,15,7]`,

```
    3
   / \
  9  20
    /  \
   15   7
```

return its depth = 3.

Maximum Depth of Binary Tree

Solution

Solution

Tree definition

First of all, here is the definition of the `TreeNode` which we would use.

C++ | Java | Python

Copy

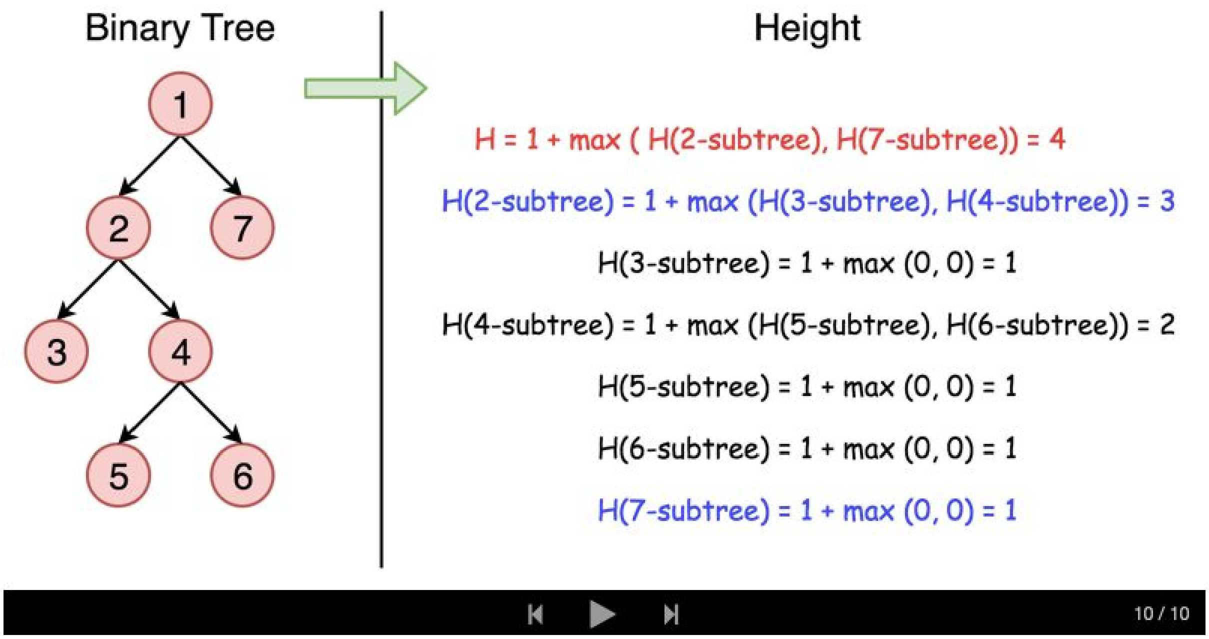
```python
class TreeNode(object):
    """ Definition of a binary tree node."""
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None
```

Approach 1: Recursion

Intuition By definition, the maximum depth of a binary tree is the maximum number of steps to reach a leaf node from the root node.

> From the definition, an intuitive idea would be to traverse the tree and record the maximum depth during the traversal.

Algorithm



One could traverse the tree either by Depth-First Search (DFS) strategy or Breadth-First Search (BFS) strategy. For this problem, either way would do. Here we demonstrate a solution that is implemented with the **DFS** strategy and **recursion**.

C++ | Java | Python | Copy

```python
class Solution:
    def maxDepth(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """
        if root is None:
            return 0
        else:
            left_height = self.maxDepth(root.left)
            right_height = self.maxDepth(root.right)
            return max(left_height, right_height) + 1
```

Complexity analysis

- Time complexity : we visit each node exactly once, thus the time complexity is $\mathcal{O}(N)$, where N is the number of nodes.
- Space complexity : in the worst case, the tree is completely unbalanced, *e.g.* each node has only left child node, the recursion call would occur N times (the height of the tree), therefore the storage to keep the call stack would be $\mathcal{O}(N)$. But in the best case (the tree is completely balanced), the height of the tree would be $\log(N)$. Therefore, the space complexity in this case would be $\mathcal{O}(\log(N))$.

Approach 3: Iteration

Intuition

We could also convert the above recursion into iteration, with the help of the *stack* data structure. Similar with the behaviors of the function call stack, the stack data structure follows the pattern of FILO (First-In-Last-Out), *i.e.* the last element that is added to a stack would come out first.

With the help of the *stack* data structure, one could mimic the behaviors of function call stack that is involved in the recursion, to convert a recursive function to a function with iteration.

Algorithm

> The idea is to keep the next nodes to visit in a stack. Due to the FILO behavior of stack, one would get the order of visit same as the one in recursion.

We start from a stack which contains the root node and the corresponding depth which is `1`. Then we proceed to the iterations: pop the current node out of the stack and push the child nodes. The depth is updated at each step.

A Conclusion - Recursion I

Now, you might be convinced that recursion is indeed a powerful technique that allows us to solve many problems in an elegant and efficient way. But still, it is no silver bullet. Not every problem can be solved with recursion, due to the time or space constraints. And recursion itself might come with some undesired side effects such as stack overflow.

In this chapter we would like to share a few more tips on how to better apply recursion to solve problems in the real world.

> When in doubt, write down the **recurrence relationship**.

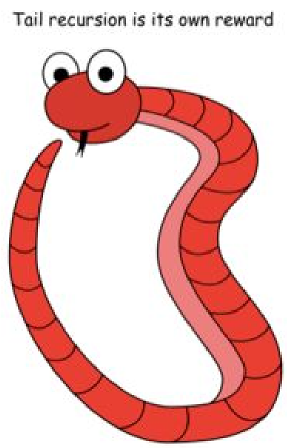
Sometimes, at a first glance it is not evident that a recursion algorithm can be applied to solve a problem. However, it is always helpful to deduct some relationships with the help of mathematical formulas, since the recurrence nature in recursion is quite close to the mathematics that we are familiar with. Often, they can clarify the ideas and uncover the hidden `recurrence relationship`. Within this chapter, you can find a fun example named Unique Binary Search Trees II, which can be solved by recursion, with the help of mathematical formulas.

> Whenever possible, apply **memoization**.

When drafting a recursion algorithm, one could start with the most naive strategy. Sometimes, one might end up with the situation where there might be `duplicate calculation` during the recursion, *e.g.* Fibonacci numbers. In this case, you can try to apply the memoization technique, which stores the intermediate results in cache for later reuse. Memoization could greatly improve the time complexity with a bit of trade on space complexity, since it could avoid the expensive duplicate calculation.

> When stack overflows, **tail recursion** might come to help.

There are often several ways to implement an algorithm with recursion. Tail recursion is a specific form of recursion that we could implement. Different from the memoization technique, tail recursion could optimize the *space* complexity of the algorithm, by eliminating the stack overhead incurred by recursion. More importantly, with tail recursion, one could avoid the problem of `stack overflow` that comes often with recursion. Another advantage about tail recursion is that often times it is easier to read and understand, compared to non-tail-recursion. Because there is no post-call dependency in tail recursion (*i.e.* the recursive call is the final action in the function), unlike non-tail-recursion. Therefore, whenever possible, one should strive to apply the tail recursion.



Next

Now, with everything that you've learned so far about recursion, you can carry on to solve many more problems on LeetCode! We provide a few more classic exercises in this chapter for you to practise your newly acquired hammer — `recursion` !

Enjoy the exercises! If you have any questions, you can always go back to review previous chapters, or simply make a post in the Discuss forum at the end of this Explore card.

Merge Two Sorted Lists

Go to Discuss

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Example:

```
Input: 1->2->4, 1->3->4
Output: 1->1->2->3->4->4
```

It is due to these recursion related space consumption that sometimes one might run into a situation called stack overflow, where the stack allocated for a program reaches its maximum space limit and the program ends up with failure. Therefore, when designing a recursion algorithm, one should carefully evaluate if there is a possibility of stack overflow when the input scales up.

Non-Recursion Related Space

As suggested by the name, the non-recursion related space refers to the memory space that is not directly related to recursion, which typically includes the space (normally in heap) that is allocated for the global variables.

Recursion or not, you might need to store the input of the problem as global variables, before any subsequent function calls. And you might need to save the intermediate results from the recursion calls as well. The latter is also known as ***memoization*** as we saw from previous chapters. For example, in the recursion algorithm with memoization to solve the Fibonacci number problem, we used a map to keep track of all intermediate Fibonacci numbers that occurred during the recursion calls. Therefore, in the space complexity analysis, we should take the space cost incurred by the memoization into consideration.