# 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

> 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.

---

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?

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:

```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.

# 1.Reverse String

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"]
```

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:

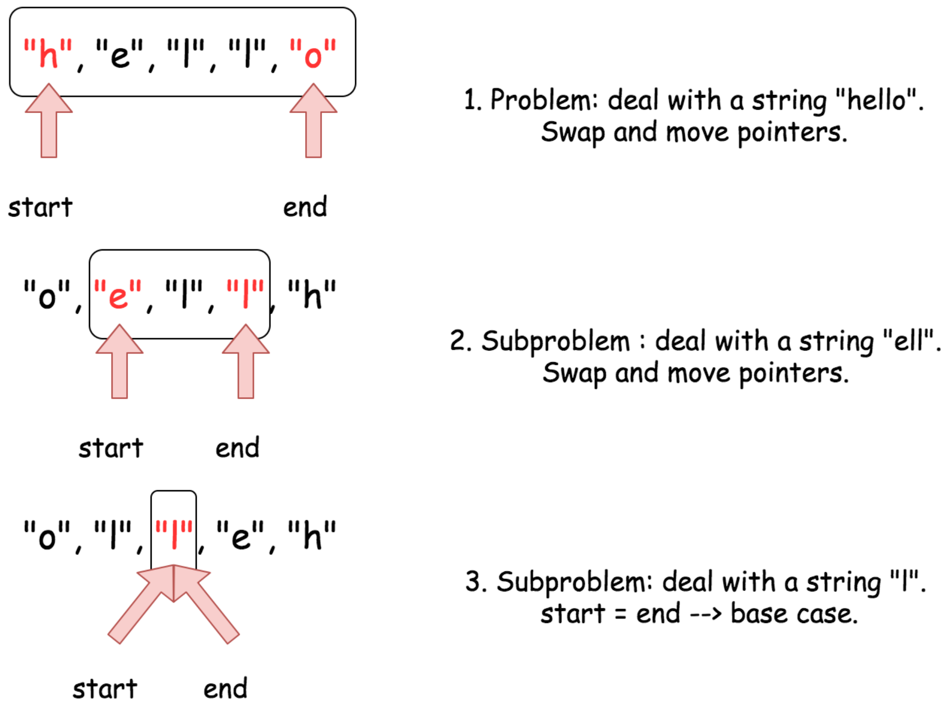
1. Take the leading and trailing characters from the input string, *i.e.* `str[0]` and `str[n-1].`
2. Swap the leading and trailing characters in place.
3. Call the function recursively to reverse the remaining substring, *i.e.* `reverseString(str[1...n-2])`.

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.

```
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:



"h", "e", "l", "l", "o"
start
end
1. Problem: deal with a string "hello". Swap and move pointers.
"o", "e", "l", "l", "h"
start
end
2. Subproblem : deal with a string "ell". Swap and move pointers.
"o", "l", "l", "e", "h"
start
end
3. Subproblem: deal with a string "l". start = end --> base case.

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.

## 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 $x_0 \in X, x_1 \in X, \ldots, x_n \in X$;

2. Call function $F(x_0), F(x_1), \ldots, F(x_n)$ ***recursively*** to solve the subproblems of $X$;
3. Finally, process the results from the recursive function calls to solve the problem corresponding to $X$.

---

o 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.

# 2.Swap Nodes in Pairs

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
class Solution(object):
    def swapPairs(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head or not head.next:
            return head
        tail = self.swapPairs(head.next.next)
        second = head.next
        head.next = tail
        second.next = head
        return second
```

# Recurrence Relation

There are two important things that one needs to figure out before implementing a recursive function:

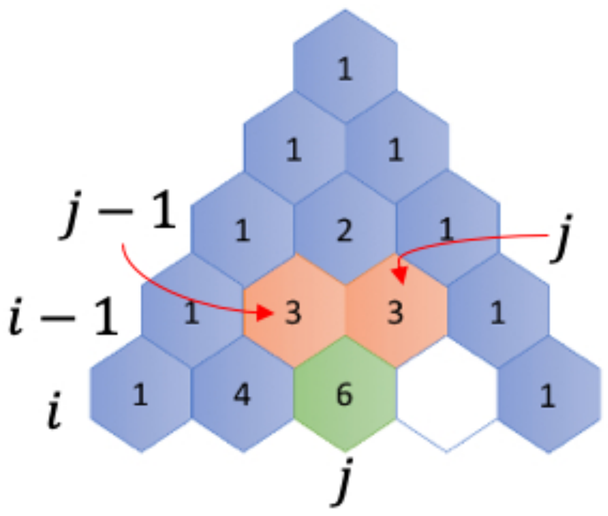
- `recurrence relation`: the relationship between the result of a problem and the result of its subproblems.
- `base case`: the case where one can compute the answer directly without any further recursion calls. Sometimes, the base cases are also called *bottom cases*, since they are often the cases where the problem has been reduced to the minimal scale, *i.e.* the bottom, if we consider that dividing the problem into subproblems is in a top-down manner.

> Once we figure out the above two elements, to implement a recursive function we simply call the function itself according to the `recurrence relation` until we reach the `base case`.

To explain the above points, let's look at a classic problem, `Pascal's Triangle`:

> Pascal's triangle are a series of numbers arranged in the shape of triangle. In Pascal's triangle, the leftmost and the rightmost numbers of each row are always 1. For the rest, each number is the sum of the two numbers directly above it in the previous row.

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



Given the above definition, one is asked to generate the Pascal's Triangle up to a certain number of 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.

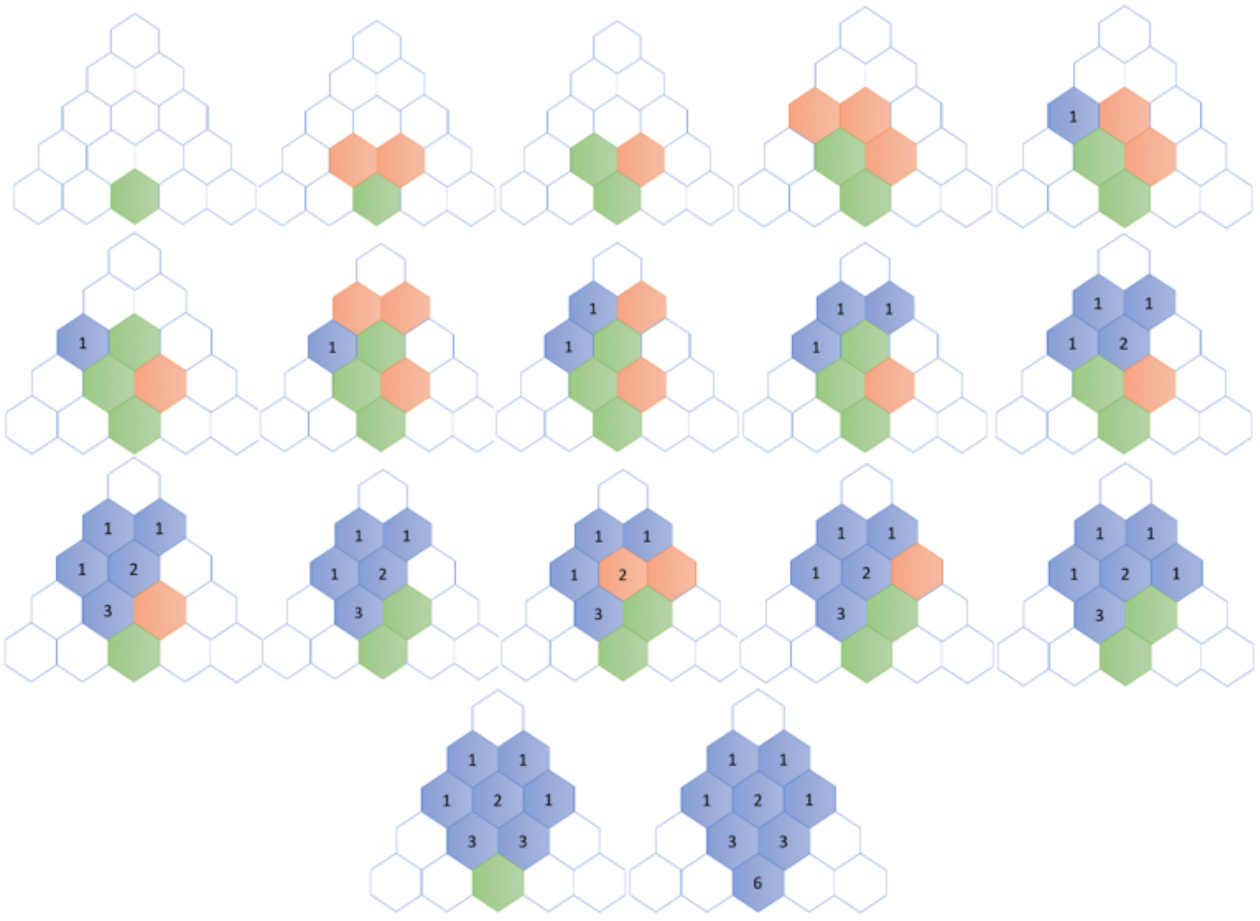
As a result, we can define the base case as follows:

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

## 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:



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.

# 3.Reverse Linked List(E 206)

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?

## Iterative Method

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!

java version：

```
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;
}
```

python version:

```
class Solution(object):
    def reverseList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head: return head
        nxt = head.next
        head.next = None
        while nxt:
            temp = nxt.next
            nxt.next = head
            head = nxt
            nxt = temp
        return head
```

or

```
class Solution(object):
    def reverseList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        curr, pre = head, None

        while curr:
            curr.next, pre, curr = pre, curr, curr.next
        return pre
```

**Complexity analysis**

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

## Recursive Method

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: $n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \rightarrow \ldots \rightarrow n_m \rightarrow \varnothing$

Assume from node $n_k + 1$ to nm had been reversed and you are at node $n_k$.

$$n_1 \rightarrow \ldots \rightarrow n_{k-1} \rightarrow n_k \rightarrow n_{k+1} \leftarrow \ldots \leftarrow n_m$$

We want $n_{k+1}$'s next node to point to $n_k$ . So, `nk.next.next = nk`;

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.

java version:

```java
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;
}
```

python version:

```python
class Solution(object):
    def reverseList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head or not head.next: return head
        node = self.reverseList(head.next)
        head.next.next = head
        head.next = None
        return node
```

**Complexity analysis**

- Time complexity : $O(n)$. Assume that $n$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.

# 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:

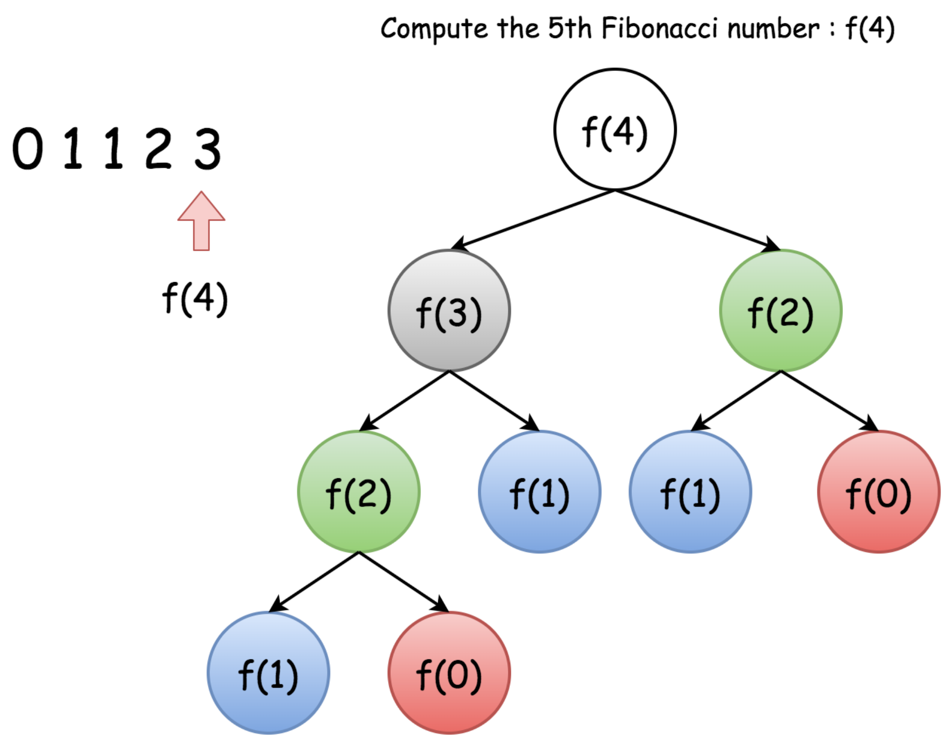
```python
def fibonacci(n):
    """
    :type n: int
    :rtype: int
    """
    if n < 2:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

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

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

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**).

```
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.

# 4.Climbing Stairs(E 70)

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?

**Note:** Given *n* will be a positive integer.

**Example 1:**

```
Input: 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

**Example 2:**

```
Input: 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

## 1: Brute Force

**Algorithm**

In this brute force approach we take all possible step combinations i.e. 1 and 2, at every step. At every step we are calling the function $climbStairs$ for step 11 and 22, and return the sum of returned values of both functions.

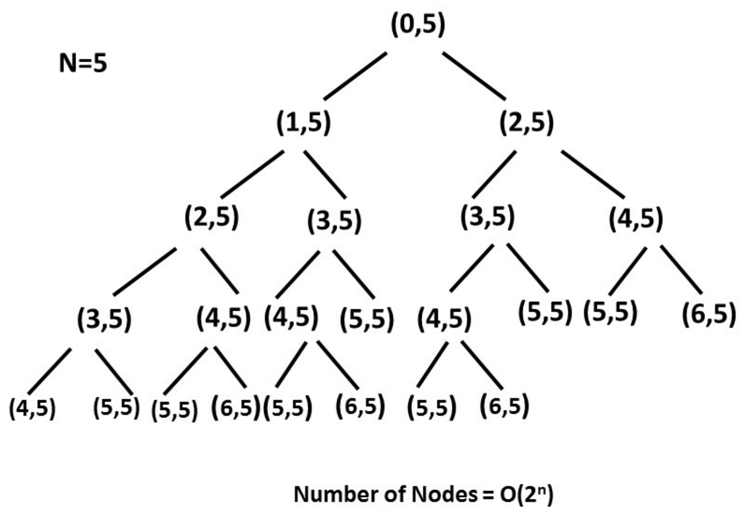
$$climbStairs(i,n) = (i+1,n) + climbStairs(i+2,n)$$

```java
public class Solution {
    public int climbStairs(int n) {
        climb_Stairs(0, n);
    }
    public int climb_Stairs(int i, int n) {
        if (i > n) {
            return 0;
        }
        if (i == n) {
            return 1;
        }
        return climb_Stairs(i + 1, n) + climb_Stairs(i + 2, n);
    }
}
```

where $i$ defines the current step and $n$ defines the destination step.

- Time complexity : $O(2^n)$ Size of recursion tree will be $2^n$.

  Recursion tree for n=5 would be like this:



- Space complexity : $O(n)$. The depth of the recursion tree can go upto $n$.

## 2: Recursion with memoization

**Algorithm**

In the previous approach we are redundantly calculating the result for every step. Instead, we can store the result at each step in $memo$ array and directly returning the result from the memo array whenever that function is called again.

In this way we are pruning recursion tree with the help of $memo$ array and reducing the size of recursion tree upto $n$.

```
public class Solution {
    public int climbStairs(int n) {
        int memo[] = new int[n + 1];
        return climb_Stairs(0, n, memo);
    }
    public int climb_Stairs(int i, int n, int memo[]) {
        if (i > n) {
            return 0;
        }
        if (i == n) {
            return 1;
        }
        if (memo[i] > 0) {
            return memo[i];
        }
        memo[i] = climb_Stairs(i + 1, n, memo) + climb_Stairs(i + 2, n, memo);
        return memo[i];
    }
}
```

**Complexity Analysis**

- Time complexity : $O(n)$. Size of recursion tree can go upto $n$.
- Space complexity : $O(n)$. The depth of recursion tree can go upto $n$.

## 3: Dynamic Programming

**Algorithm**

As we can see this problem can be broken into subproblems, and it contains the optimal substructure property i.e. its optimal solution can be constructed efficiently from optimal solutions of its subproblems, we can use dynamic programming to solve this problem.

One can reach $i^{th}$ step in one of the two ways:

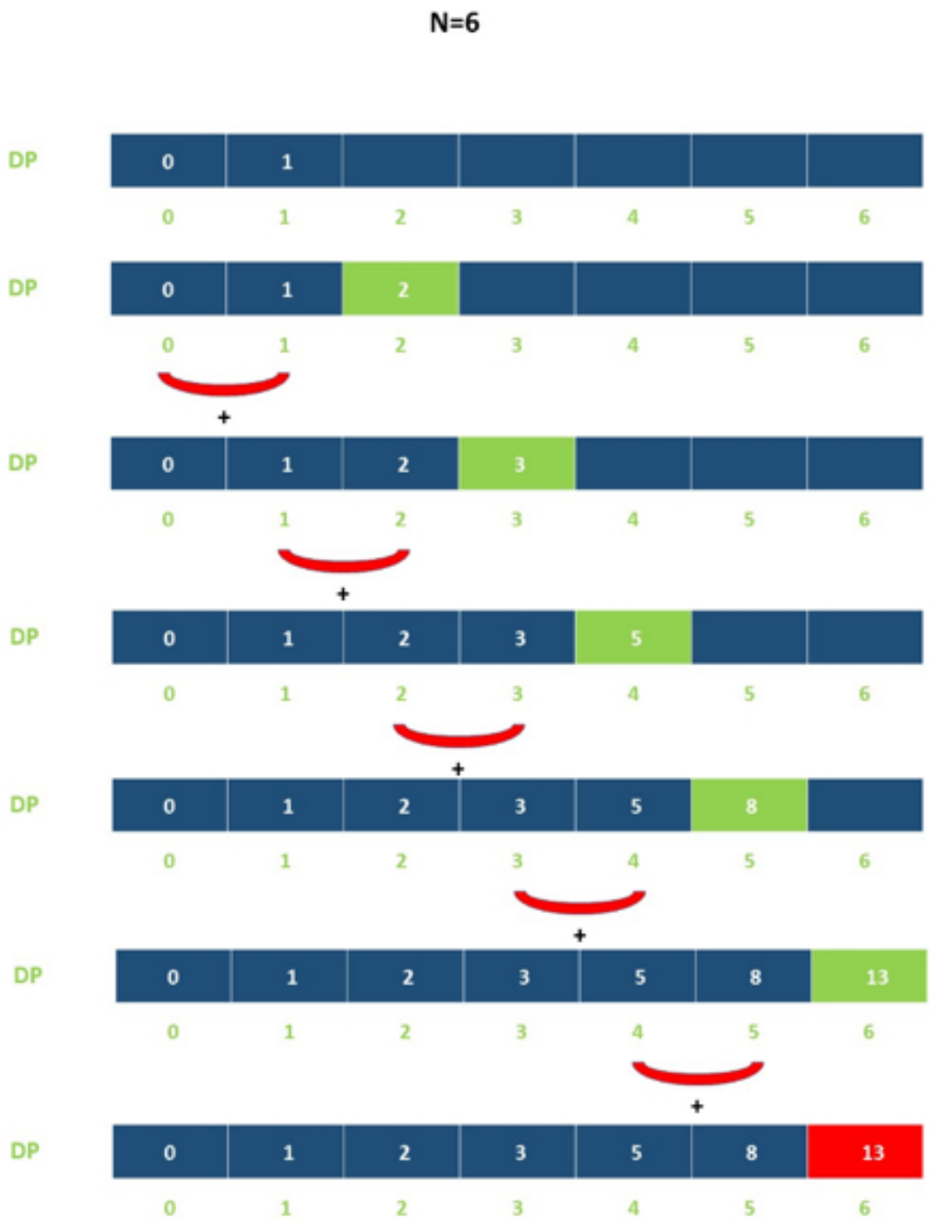
1. Taking a single step from $(i-1)^{th}$ step.
2. Taking a step of 22 from $(i-2)^{th}$ step.

So, the total number of ways to reach $i^{th}$ is equal to sum of ways of reaching $(i-1)^{th}$ step and ways of reaching $(i-2)^{th}$ step.

Let $dp[i]$ denotes the number of ways to reach on $i^{th}$ step:

$$dp[i] = dp[i-1] + dp[i-2]$$

Example:



```java
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int[] dp = new int[n + 1];
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
}
```

**Complexity Analysis**

- Time complexity : O(n). Single loop upto n$n$.
- Space complexity : O(n). $dp$ array of size $n$ is used.

## 4: Fibonacci Number

### Algorithm

In the above approach we have used $dp$ array where $dp[i] = dp[i-1] + dp[i-2]$. It can be easily analysed that $dp[i]$ is nothing but $i^{th}$ fibonacci number.

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Now we just have to find $n^{th}$ number of the fibonacci series having $1$ and $2$ their first and second term respectively, i.e. $Fib(1) = 1$ and $Fib(2) = 2$.

```
public class Solution {
    public int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }
        int first = 1;
        int second = 2;
        for (int i = 3; i <= n; i++) {
            int third = first + second;
            first = second;
            second = third;
        }
        return second;
    }
}
```

### Complexity Analysis

- Time complexity : $O(n)$. Single loop upto n$n$ is required to calculate $n^{th}$ fibonacci number.
- Space complexity : $O(1)$. Constant space is used.

## 5: Binets Method

This is an interesting solution which uses matrix multiplication to obtain the n^{th}*nth* Fibonacci Number. The matrix takes the following form:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Let's say $Q = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$. As per the method, the n^{th}*nth* Fibonacci Number is given by $Q^{n-1}[0,0]$.

Let's look at the proof of this method.

We can prove this method using Mathematical Induction. We know, this matrix gives the correct result for the $3^{rd}$ term(base case). Since $Q^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$. This proves that the base case holds.

Assume that this method holds for finding the n^{th}$nth$ Fibonacci Number, i.e. $F_n = Q^{n-1}[0,0]$, where

$$Q^{n-1} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix}$$

Now, we need to prove that with the above two conditions holding true, the method is valid for finding the $(n+1)^{th}$ Fibonacci Number, i.e. $F_{n+1} = Q^n[0,0]$.

Proof:

$$Q^n = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$Q^n = \begin{bmatrix} F_n + F_{n-1} & F_n \\ F_{n-1} + F_{n-2} & F_{n-1} \end{bmatrix}$$
$$Q^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Thus, $F_{n+1} = Q^n[0,0]$. This completes the proof of this method.

The only variation we need to do for our problem is that we need to modify the initial terms to 2 and 1 instead of 1 and 0 in the Fibonacci series. Or, another way is to use the same initial $Q$ matrix and use $result = Q^n[0,0]$ to get the final result. This happens because the initial terms we have to use are the 2nd and 3rd terms of the otherwise normal Fibonacci Series.

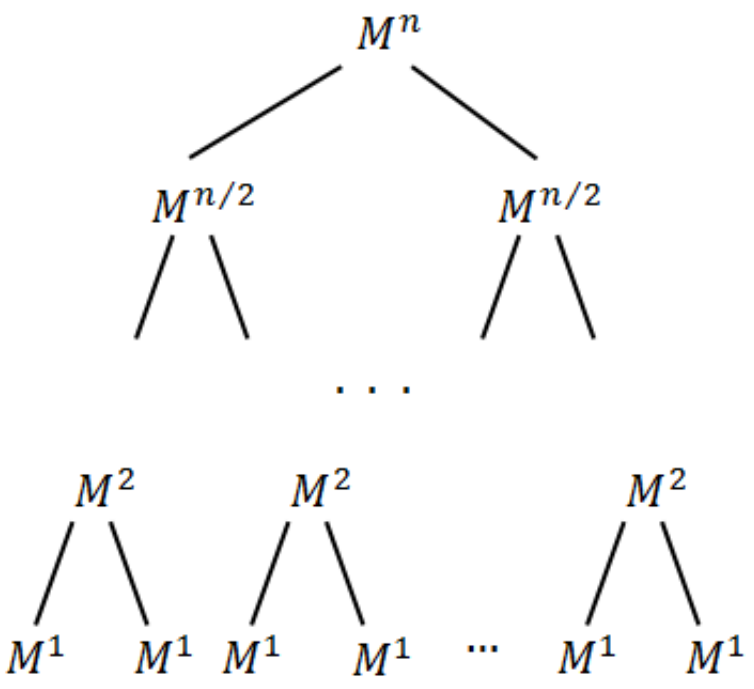
```java
public class Solution {
    public int climbStairs(int n) {
        int[][] q = {{1, 1}, {1, 0}};
        int[][] res = pow(q, n);
        return res[0][0];
    }
    public int[][] pow(int[][] a, int n) {
        int[][] ret = {{1, 0}, {0, 1}};
        while (n > 0) {
            if ((n & 1) == 1) {
                ret = multiply(ret, a);
            }
            n >>= 1;
            a = multiply(a, a);
        }
        return ret;
    }
    public int[][] multiply(int[][] a, int[][] b) {
        int[][] c = new int[2][2];
        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                c[i][j] = a[i][0] * b[0][j] + a[i][1] * b[1][j];
            }
        }
        return c;
    }
}
```

**Complexity Analysis**

- Time complexity : $O(log(n))$. Traversing on $log(n)$ bits.
- Space complexity : $O(1)$. Constant space is used.

Proof of Time Complexity:

Let's say there is a matrix $M$ to be raised to power $n$. Suppose, $n$ is the power of 2. Thus, $n = 2^i$, $i \in \mathbb{N}$, where $\mathbb{N}$ represents the set of natural numbers(including 0). We can represent in the form of a tree:



Meaning that: $M^n = M^{n/2} \cdot M^{n/2} = .... = \prod_1^n M^1$

So, to calculate $M^n$ matrix, we should calculate $M^{n/2}$ matrix and multiply it by itself. To calculate $M^{n/2}$ we would have to do the same with $M^{n/4}$ and so on.

Obviously, the tree height is $log_2 n$.

Let's estimate $M^n$ calculation time. M*M* matrix is of the same size in any power . Therefore, we can perform the multiplication of two matrices in any power in $O(1)$. We should perform $log_2 n$ of such multiplications. So, $M^n$ calculation complexity is $O(log_2 n)$.

In case, the number nn is not a power of two, we can break it in terms of powers of 2 using its binary representation:

$$n = \sum_{p \in P} 2^p, \text{where } P \subset \mathbb{N}$$

Thus, we can obtain the final result using:

$$M^n = \prod_{p \in P} M^{2^p}$$

This is the method we've used in our implementation. Again, the complexity remains $O(log_2 n)$ as we have limited the number of multiplications to $O(log_2 n)$.

## 6: Fibonacci Formula

**Algorithm**

We can find $n^{th}$ fibonacci number using this formula:

$$F_n = 1/\sqrt{5}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

For the given problem, the Fibonacci sequence is defined by $F_0 = 1 * F * 0 = 1, F_1 = 1 * F * 1 = 1, F_1 = 2 * F * 1 = 2, F_{n+2} = F_{n+1} + F_n$. *A standard method of trying to solve such recursion formulas is assume* $F_n$ *of the form* $F_n = a^n$. *Then, of course,* $F_{n+1} = a^{n+1}$ *and* $F_{n+2} = a^{n+2}$ *so the equation becomes* $a^{n+2} = a^{n+1} + a^n$. *If we divide the entire equation by an we arrive at* $a^2 = a + 1$ *or the quadratic equation* $a^2 - a - 1 = 0$.

Solving this by the quadratic formula, we get:

$$a = 1/\sqrt{5}\left(\left(\frac{1 \pm \sqrt{5}}{2}\right)\right)$$

The general solution, thus takes the form:

$$F_n = A\left(\frac{1+\sqrt{5}}{2}\right)^n + B\left(\frac{1-\sqrt{5}}{2}\right)^n$$

For $n = 0$, we get $A + B = 1$

For $n = 1$, we get $A\left(\frac{1+\sqrt{5}}{2}\right) + B\left(\frac{1-\sqrt{5}}{2}\right) = 1$

Solving the above equations, we get:

$$A = \left(\frac{1+\sqrt{5}}{2\sqrt{5}}\right), B = \left(\frac{1-\sqrt{5}}{2\sqrt{5}}\right)$$

Putting these values of $A$ and $B$ in the above general solution equation, we get:

$$F_n = 1/\sqrt{5}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$$

```
public class Solution {
    public int climbStairs(int n) {
        double sqrt5=Math.sqrt(5);
        double fibn=Math.pow((1+sqrt5)/2,n+1)-Math.pow((1-sqrt5)/2,n+1);
        return (int)(fibn/sqrt5);
    }
}
```

**Complexity Analysis**

- Time complexity : $O(log(n))$. $pow$ method takes $log(n)$ time.
- Space complexity : $O(1)$. Constant space is used.

# 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.* $\mathcal{O}(1)$.

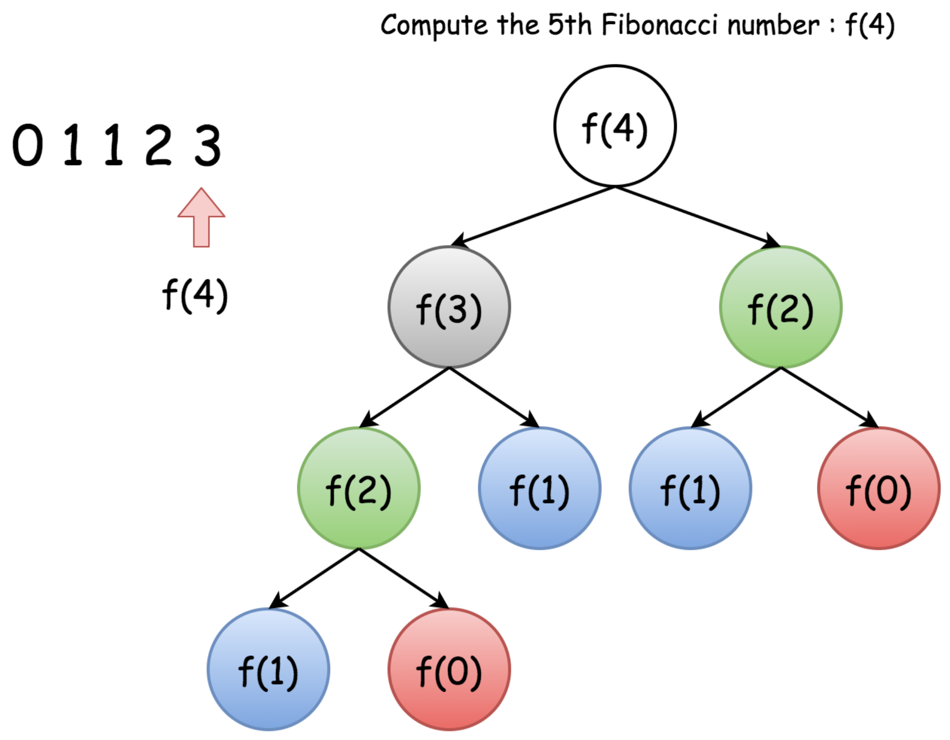
To sum up, the overall time complexity of our recursive function `printReverse(str)` would be $\mathcal{O}(printReverse) = n * \mathcal{O}(1) = \mathcal{O}(n)$.

## 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.

# 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.

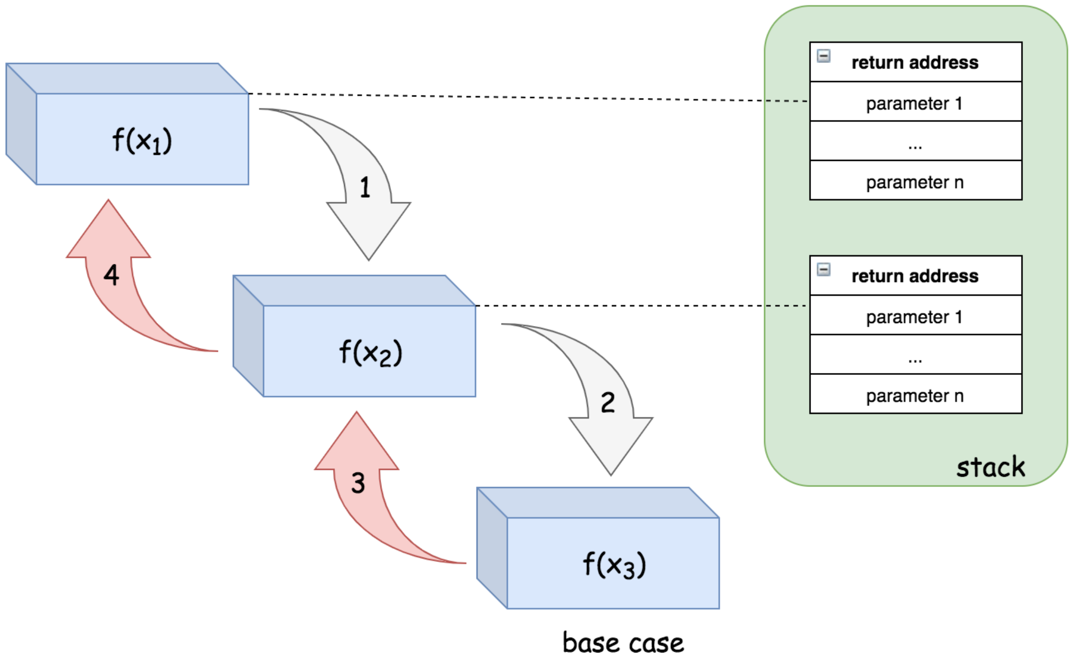
This space in the stack is the minimal cost that is incurred during a function call. However, once the function call is done, this space would be freed.

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(x1) -> f(x2) -> f(x3)`, we show the sequence of execution steps along with the layout of stack:



A space in the stack would be allocated for `f(x1)` in order to call `f(x2)`. Similarly in `f(x2)`, the system would allocate another space for the call to `f(x3)`. Finally in `f(x3)`, we reach the base case, therefore there is no further recursion call within `f(x3)`.

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.

# 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.

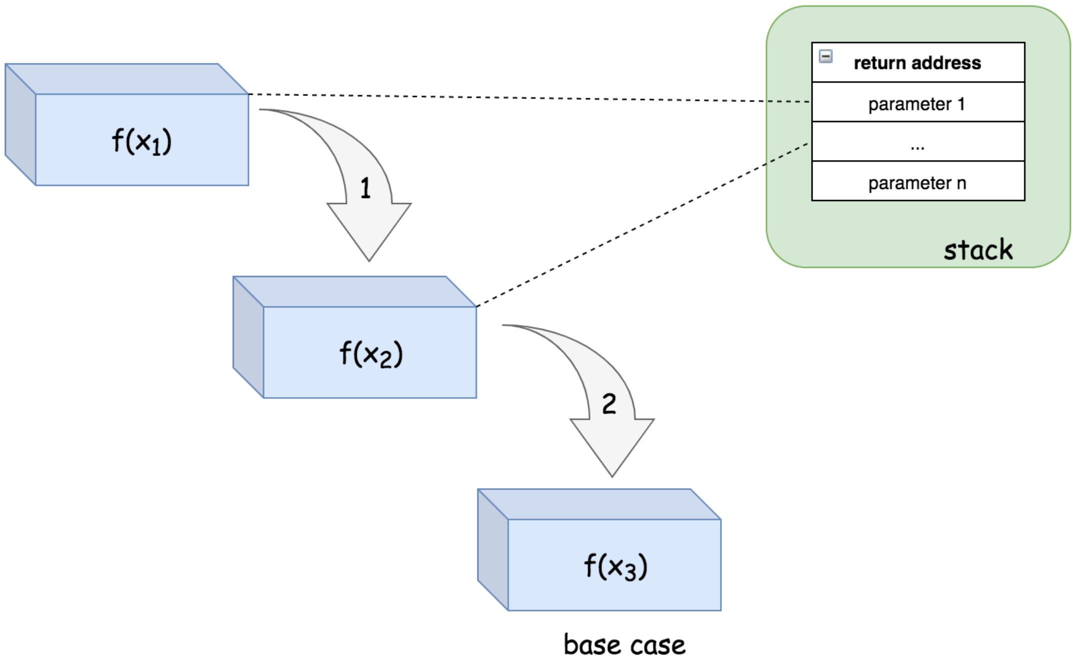
> **Tail recursion** is a recursion where the recursive call is the final instruction in the recursion function. And there should be only **one** recursive call in the function.

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.

```
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.

# 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.