递归原理

> 递归是一种解决问题的有效方法，在递归过程中，函数将自身作为子例程调用

你可能想知道如何实现调用自身的函数。诀窍在于，每当递归函数调用自身时，它都会将给定的问题拆解为子问题。递归调用继续进行，直到到子问题无需进一步递归就可以解决的地步。

为了确保递归函数不会导致无限循环，它应具有以下属性：

1. 一个简单的`基本案例（basic case）`（或一些案例）—— 能够不使用递归来产生答案的终止方案。
2. 一组规则，也称作`递推关系（recurrence relation）`，可将所有其他情况拆分到基本案例。

注意，函数可能会有多个位置进行自我调用。

24.两两交换链表中的节点（中等）

1. 题目描述

   给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

   你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

   示例:

   ```
   给定 1->2->3->4，你应该返回 2->1->4->3.
   ```

2. 递归实现

   ```
   class Solution {
   public:
       ListNode* swapPairs(ListNode* head) {
           if(head && head->next){
               ListNode* newhead = head->next;
               head->next = newhead->next;
               newhead->next = head;
               head = newhead;
               head->next->next = swapPairs(head->next->next);
           }
           return head;
       }
   };
   ```

3. 非递归实现

   ```
   class Solution {
   public:
       ListNode* swapPairs(ListNode* head) {
           ListNode* root = new ListNode(-1);
           root->next = head;
   ```

```
        ListNode* done = root;
        while(done->next && done->next->next){
            ListNode* cur = done->next;
            done->next = cur->next;
            cur->next = done->next->next;
            done->next->next = cur;
            done = done->next->next;
        }
        return root->next;
    }
};
```

递推关系

在实现递归函数之前，有两件重要的事情需要弄清楚:

- `递推关系`： 一个问题的结果与其子问题的结果之间的关系。
- `基本情况`：不需要进一步的递归调用就可以直接计算答案的情况。 有时，基本案例也被称为 *bottom cases*，因为它们往往是问题被减少到最小规模的情况，*也就是*如果我们认为将问题划分为子问题是一种自上而下的方式的最下层。

> 一旦我们计算出以上两个元素，再想要实现一个递归函数，就只需要根据`递推关系`调用函数本身，直到其抵达`基本情况`。

为了解释以上几点，让我们来看一个经典的问题，`帕斯卡三角 (Pascal's Triangle)`：

> 帕斯卡三角形是排列成三角形的一系列数字。 在帕斯卡三角形中，每一行的最左边和最右边的数字总是 1。 对于其余的每个数字都是前一行中直接位于它上面的两个数字之和。

递推关系

让我们从帕斯卡三角形内的递推关系开始。

首先，我们定义一个函数 $f(i, j)$，它将会返回帕斯卡三角形`第 i 行`、`第 j 列`的数字。

我们可以用下面的公式来表示这一递推关系：

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

基本情况

可以看到，每行的最左边和最右边的数字是`基本情况`，在这个问题中，它总是等于 1。

因此，我们可以将基本情况定义如下：

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

演示

正如我们所看到的，一旦我们定义了`递推关系`和`基本情况`，递归函数的实现变得更加直观，特别是在我们用数学公式表示出这两个元素之后。

下面给出一个例子，展示我们如何用这个公式递归地计算 $f(5, 3)$，*也就是*帕斯卡三角形`第 5 行`中的`第 3 个`数。

Pascal_Triangle

我们可以将 $f(5, 3)$ 分解为 $f(5, 3) = f(4, 2) + f(4, 3)$，然后递归地调用 $f(4, 2)$ 和 $f(4, 3)$：

- 对于调用的 $f(4, 2)$，我们可以进一步展开它，直到到达基本情况，正如下面所描述的：

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

- 对于调用的 $f(4, 3)$，类似地，我们可以将其分解为：

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

- 最后，我们结合上述子问题的结果：

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

下一步

在上面的例子中，您可能已经注意到递归解决方案可能会导致一些重复的计算，例如，我们重复计算相同的中间数以获得最后一行中的数字。 举例说明，为了得到 $f(5,3)$ 的结果，我们在 $f(4,2)$ 和 $f(4,3)$ 的调用中计算了 $f(3,2)$ 两次。

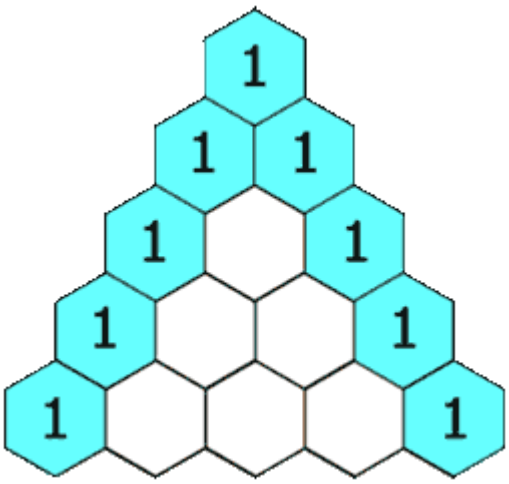
我们将在探索卡的下一章中讨论如何避免这些重复计算（duplicate calculations）。

在本文之后，你将会找到与帕斯卡三角相关的问题作为练习。

119 杨辉三角（简单）

1. 题目描述

   给定一个非负索引 *k*，其中 *k* ≤ 33，返回杨辉三角的第 *k* 行。



   在杨辉三角中，每个数是它左上方和右上方的数的和。

   示例:

```
输入: 3
输出: [1,3,3,1]
```

2. 递归实现

   大量重复求解会超时、、、

```
class Solution {
public:
     int getNum(int i, int j){
        if(j == 0 || i == j) return 1;
        else return getNum(i-1, j-1) + getNum(i-1, j);
    }
    vector<int> getRow(const int rowIndex) {
        vector<int> ans(rowIndex+1);
        for(int j = 0; j <= rowIndex; j++)
            ans[j] = getNum(rowIndex, j);
        return ans;
    }
};
```

206.反转链表（简单）

1. 题目描述

   反转一个单链表。

   示例:

   ```
   输入: 1->2->3->4->5->NULL
   输出: 5->4->3->2->1->NULL
   ```

   进阶: 你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

2. 递归实现

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(!head) return NULL;
        if(!head->next) return head;
        ListNode* newhead = reverseList(head->next);
        ListNode* cur = newhead;
        while(cur->next != NULL)
            cur  = cur->next;
        cur->next = head;
        head->next = NULL;
        return newhead;
    }
};
```

递归中的重复计算

通常情况下，递归是一种直观而有效的实现算法的方法。 但是，如果我们不明智地使用它，可能会给性能带来一些不希望的损失，*例如*重复计算。 在前一章的末尾，我们遇到了帕斯卡三角的重复计算问题，其中一些中间结果被多次计算。

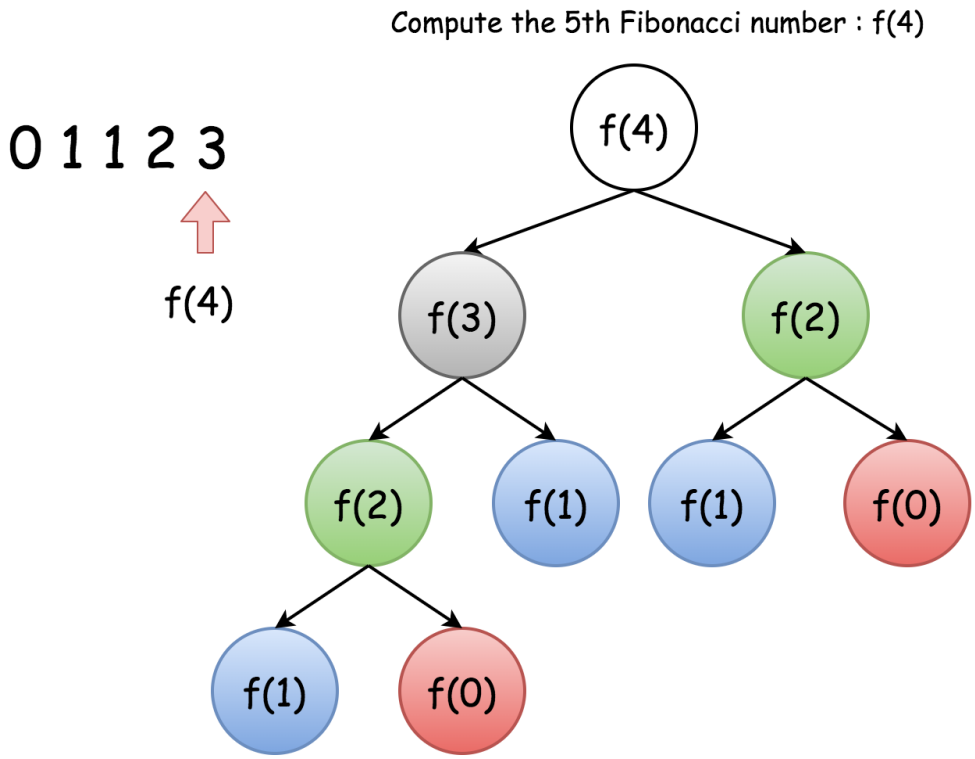
在本文中，我们将进一步研究递归可能出现的重复计算问题。 然后我们将提出一种常用的技术，称为`记忆化(memoization)`，可以用来避免这个问题。

为了演示重复计算的另一个问题，让我们看一个大多数人可能都很熟悉的例子，斐波那契数。 如果我们定义函数`F(n)`表示在索引`n`处的斐波那契数，那么你可以推导出如下的递推关系：`F(n) = F(n - 1) + F(n - 2)`

基本情况：`F(0) = 0, F(1) = 1`

现在，如果你想知道`F(4)`是多少，你可以应用上面的公式并进行展开：`F(4) = F(3) + F(2) = (F(2) + F(1)) + F(2)`

正如你所看到的，为了得到 f(4)的结果，我们需要在上述推导之后计算两次数`F(2)`：第一次在`F(4)`的第一次展开中，第二次在中间结果`F(3)`中。下面的树显示了在计算`F(4)`时发生的所有重复计算（按颜色分组）。



记忆化

为了消除上述情况中的重复计算，正如许多人已经指出的那样，其中一个想法是将中间结果**存储**在缓存中，以便我们以后可以重用它们，而不需要重新计算。

这个想法也被称为*记忆化*，这是一种经常与递归一起使用的技术。

> 记忆化 是一种优化技术，主要用于**加快**计算机程序的速度，方法是**存储**昂贵的函数调用的结果，并在相同的输入再次出现时返回缓存的结果。 (来源: 维基百科)

回到斐波那契函数`F(n)`。 我们可以使用哈希表来跟踪每个以`n`为键的`F(n)`的结果。 散列表作为一个缓存，可以避免重复计算。 记忆化技术是一个很好的例子，它演示了如何通过增加额外的空间以减少计算时间。为了便于比较，我们在下面提供了带有记忆化功能的斐波那契数列解决方案的实现。

```
import java.util.HashMap;
public class Main {
  HashMap<Integer, Integer> cache = new HashMap<Integer, Integer>();
```

```
  private int fib(int N) {
    if (cache.containsKey(N)) {
      return cache.get(N);
    }
    int result;
    if (N < 2) {
      result = N;
    } else {
      result = fib(N-1) + fib(N-2);
    }
    // keep the result in cache.
    cache.put(N, result);
    return result;
  }
}
```

509.斐波那契数（简单）

1. 题目描述

   斐波那契数，通常用 `F(n)` 表示，形成的序列称为**斐波那契数列**。该数列由 `0` 和 `1` 开始，后面的每一项数字都是前面两项数字的和。也就是：

   ```
   F(0) = 0,   F(1) = 1
   F(N) = F(N - 1) + F(N - 2), 其中 N > 1.
   ```

   给定 `N`，计算 `F(N)`。

   示例 1：

   ```
   输入：2
   输出：1
   解释：F(2) = F(1) + F(0) = 1 + 0 = 1.
   ```

   示例 2：

   ```
   输入：3
   输出：2
   解释：F(3) = F(2) + F(1) = 1 + 1 = 2.
   ```

   提示：

   - $0 \le$ `N` ≤ 30

2. 递归实现

```
class Solution {
public:
    unordered_map<int, int> F;
    int fib(int N) {
        if(N == 0) return 0;
        if(N == 1) return 1;
        if(F.count(N) > 0) return F[N];
        int ans = fib(N-1) + fib(N-2);
        F[N] = ans;
        return ans;
    }
};
```

70.爬楼梯（简单）

1. 题目描述

   假设你正在爬楼梯。需要 *n* 阶你才能到达楼顶。

   每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

   注意：给定 *n* 是一个正整数。

   示例 1：

```
输入：  2
输出：  2
解释：  有两种方法可以爬到楼顶。
1.  1 阶 + 1 阶
2.  2 阶
```

   示例 2：

```
输入：  3
输出：  3
解释：  有三种方法可以爬到楼顶。
1.  1 阶 + 1 阶 + 1 阶
2.  1 阶 + 2 阶
3.  2 阶 + 1 阶
```

2. 递归实现

```
class Solution {
public:
    unordered_map<int, int> F;
    int climbStairs(int n) {
        if(n==1)    return 1;
        if(n==2)    return 2;
        if(F.count(n) > 0) return F[n];
        int ans = climbStairs(n-1) + climbStairs(n-2);
        F[n] = ans;
        return ans;
    }
};
```

方法 5: Binets 方法

算法

这里有一种有趣的解法，它使用矩阵乘法来得到第 n 个斐波那契数。矩阵形式如下：

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

令 $Q = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$。按照此方法，第 n 个斐波那契数可以由 $Q^{n-1}[0,0]$ 给出。

让我们试着证明一下。

我们可以使用数学归纳法来证明这一方法。易知，该矩阵给出了第 3 项（基本情况）的正确结果。由于 $Q^2 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$。这证明基本情况是成立的。

假设此方法适用于查找第 n 个斐波那契数，即 $F_n = Q^{n-1}[0,0]$，那么：

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

现在，我们需要证明在上述两个条件为真的情况下，该方法可以有效找出第 $(n+1)$ 个斐波那契数，即，$F_{n+1} = Q^n[0,0]$。

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

从而，$F_{n+1} = Q^n[0,0]$。得证。

我们需要为我们的问题做的唯一改动就是将斐波那契数列的初始项修改为 2 和 1 来代替原来的 1 和 0 。或者，另一种方法是使用相同的初始矩阵 Q 并使用 $result = Q^n[0,0]$ 得出最后结果。发生这种情况的原因是我们必须使用原斐波那契数列的第 2 项和第 3 项作为初始项。

方法 6: 斐波那契公式

算法

我们可以使用这一公式来找出第 n 个斐波那契数：

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

对于给定的问题，斐波那契序列将会被定义为 $F_0 = 1$，$F_1 = 1$，$F_1 = 2$，$F_{n+2} = F_{n+1} + F_n$。尝试解决这一递归公式的标准方法是设出 F_n，其形式为 $F_n = a^n$。然后，自然有 $F_{n+1} = a^{n+1}$ 和 $F_{n+2} = a^{n+2}$，所以方程可以写作 $a^{n+2} = a^{n+1} + a^n$。如果我们对整个方程进行约分，可以得到 $a^2 = a + 1$ 或者写成二次方程形式 $a^2 - a - 1 = 0$。

对二次公式求解，我们得到：

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

一般解采用以下形式：

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

$n = 0$ 时，有 $A + B = 1$

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

解上述等式，我们得到：

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

将 A 和 B 的这些值带入上述的一般解方程中，可以得到：

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

复杂度分析

递归 —— 时间复杂度

在本文中，我们将重点介绍如何计算递归算法的时间复杂度。

> 给出一个递归算法，其时间复杂度 {\mathcal{O}(T)}O(T) 通常是**递归调用的数量**（记作 {R}R） 和计算的时间复杂度的乘积（表示为 {\mathcal{O}(s)}O(s)）的乘积：
>
> $\mathcal{O}(T) = R * \mathcal{O}(s)$

示例

也许你还记得，在反转字符串问题中，我们需要以相反的顺序打印字符串，解决问题的递归关系可以表示如下：

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

其中 `str[1...n]` 是输入字符串 `str` 的子串，仅不含前导字符 `str[0]`。

如您所见，该函数将被递归调用 n 次，其中 n 是输入字符串的大小。在每次递归结束时，我们只是打印前导字符，因此该特定操作的时间复杂度是恒定的，即 $\mathcal{O}(1)$。

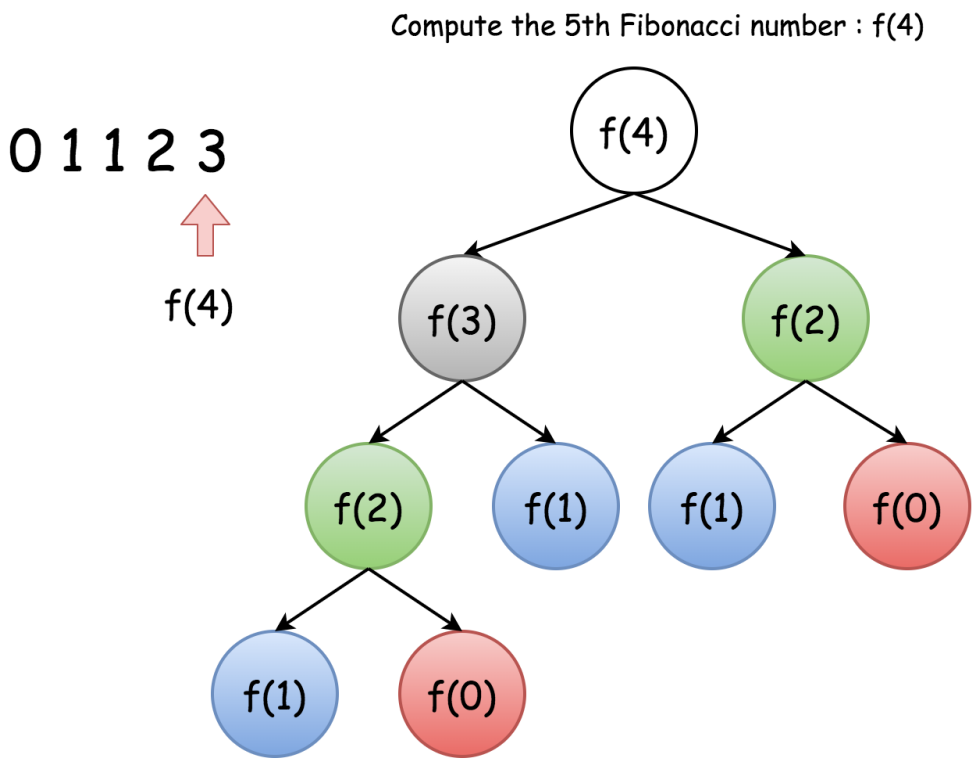
总而言之，我们的递归函数 `printReverse(str)` 的总体时间复杂度为 $\mathcal{O}(printReverse) = n * \mathcal{O}(1) = \mathcal{O}(n)$。

执行树

对于递归函数，递归调用的数量很少与输入的大小呈线性关系。如果你还记得我们在前一章中讨论过的斐波那契数问题，其递推关系被定义为 `f(n) = f(n-1) + f(n-2)`。乍一看，在执行斐波那契函数期间计算递归调用的数量似乎并不简单。

> 在这种情况下，最好采用**执行树**，这是一个用于表示递归函数的执行流程的树。树中的每个节点都表示递归函数的调用。因此，树中的节点总数对应于执行期间的递归调用的数量。

递归函数的执行树将形成 `n 叉树`，其中 `n` 作为递推关系中出现递归的次数。例如，斐波那契函数的执行将形成二叉树，下面的图示展现了用于计算斐波纳契数 `f(4)` 的执行树。



在 n 层的完全二叉树中，节点的总数为 $2^n - 1$。因此 `f(n)` 中递归数目的上限（尽管不严格）也是 $2^n - 1$。那么我们可以估计 `f(n)` 的时间复杂度为 $\mathcal{O}(2^n)$。

记忆化（Memoization）

在前一章中，我们讨论了通常用于优化递归算法的时间复杂度的记忆化技术。过高速缓存和重用中间结果，记忆化技术可以大大减少递归调用的数量，即减少执行树中的分支数量。在使用记忆化分析递归算法的时间复杂度时，也应该考虑到这种减少。

让我们回到斐波纳契数的例子。通过记忆化技术，我们保存每个索引 `n` 对应的的斐波那契数的结果。我们确信每个斐波那契数的计算只会发生一次。而从递推关系来看，斐波纳契数 `f(n)` 将取决于其所有 `n-1` 个先验斐波纳契数。结果，计算 `f(n)` 的递归将被调用 `n-1` 次以计算它所依赖的所有先验数字。

现在，我们可以简单地应用我们在本章开头介绍的公式来计算时间复杂度，即$\mathcal{O}(1) * n = \mathcal{O}(n)$。记忆化技术不仅可以优化算法的时间复杂度，还可以简化时间复杂度的计算。

递归 —— 空间复杂度

在本文中，我们将会讨论如何分析递归算法的空间复杂度。

> 在计算递归算法的空间复杂度时，应该考虑造成空间消耗的两个部分：递归相关空间（`recursion related space`）和非递归相关空间（`non-recursion related space`）。

递归相关空间

递归相关空间是指由递归直接引起的内存开销，即用于跟踪递归函数调用的堆栈。为了完成典型的函数调用，系统应该在栈中分配一些空间来保存三个重要信息：

1. 函数调用的返回地址。一旦函数调用完成，程序应该知道返回的位置，即函数调用之前的点
2. 传递给函数调用的参数
3. 函数调用中的局部变量

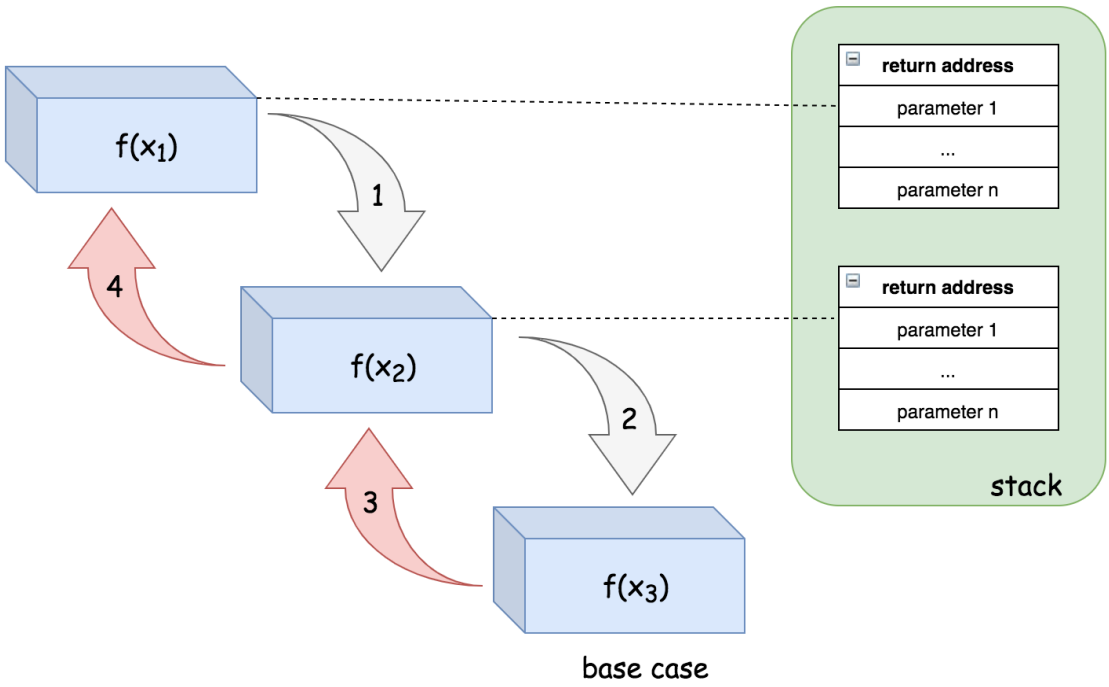
栈中的这个空间是函数调用期间产生的最小成本。然而，一旦完成函数调用，就会释放该空间。

对于递归算法，函数调用将连续链接直到它们到达基本情况（*也称为* 底层情况）。这意味着用于每个函数调用的空间也会累积。

> 对于递归算法，如果没有产生其他内存消耗，则此递归引起的空间将是算法的空间上限。

例如，在练习反转字符串中，我们没有使用额外的内存，因为我们仅仅是打印一个字符。对于每个递归调用，我们假设它可能需要一个最大为某一常量值的空间。并且递归调用最多可以链接 `n` 次，其中 `n` 是输入字符串的大小。因此，该递归算法的空间复杂度就是 $\mathcal{O}(n)$。

为了更好地说明这一点，接下来我们将会展示递归调用 `f(x1) -> f(x2) -> f(x3)` 的执行顺序以及栈空间的分配情况。



栈中的空间将会分配给 `f(x1)` 来调用 `f(x2)`。类似的情况也同样发生在 `f(x2)` 中，系统会为 `f(x3)` 的调用分配另一个空间，最后在 `f(x3)` 中，我们到达基本情况，因此在 `f(x3)` 中没有进行进一步的递归调用。

正是由于这些与递归相关的空间消耗，有时可能会遇到称为堆栈溢出的情况，其中为程序分配的堆栈达到其最大空间限制并导致程序最终失败。在设计递归算法时，应该仔细评估在输入规模扩大时是否存在堆栈溢出的可能性。

非递归相关空间

正如名称所示，非递归相关空间指的是与递归过程没有直接关系的内存空间，通常包括为全局变量分配的空间（通常在堆中）。

不管是否递归，你都可能需要在任何函数调用之前将问题的输入存储为全局变量。你可能还需要保存递归调用的中间结果。后者就是我们前面提到过的**记忆化技术**。例如，在使用带有记忆化技术的递归算法解决斐波那契数问题时，我们使用映射（map）来跟踪在递归调用期间产生的所有中间斐波那契数。因此，在分析空间复杂度时，我们应该考虑到因采用记忆化技术所导致的空间成本。

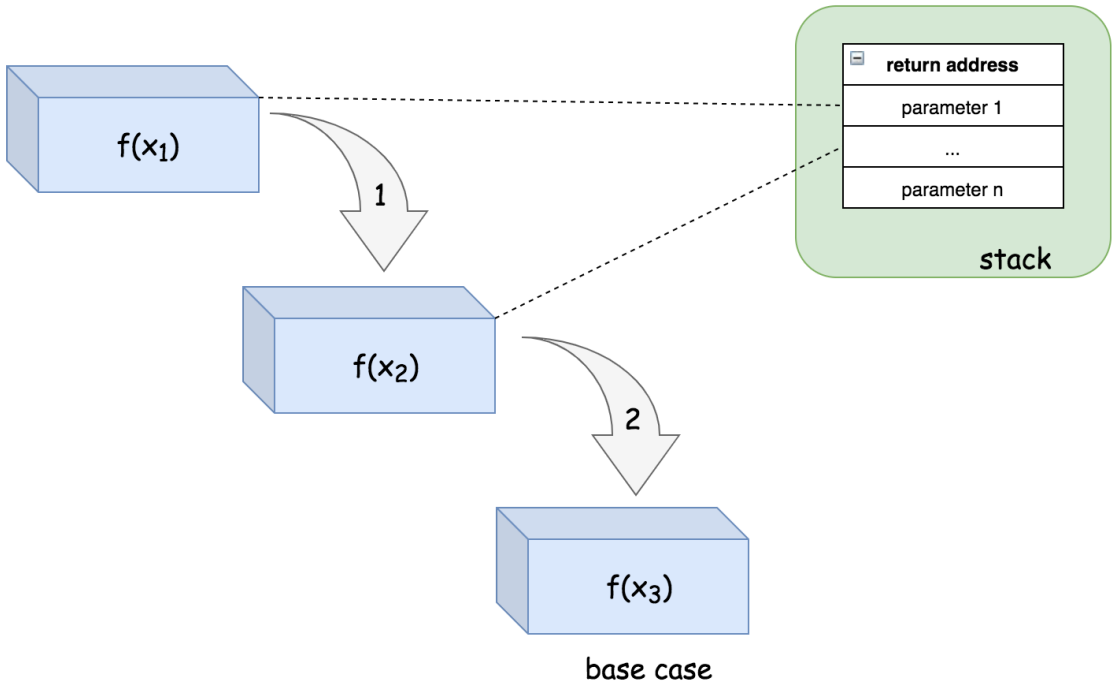
尾递归

在上一篇文章中，我们讨论了由于递归调用而在系统调用栈上产生的隐式额外空间。然而，你应该学习识别一种称为尾递归的特殊递归情况，它不受此空间开销的影响。

> 尾递归函数是递归函数的一种，其中递归调用是递归函数中的最后一条指令。并且在函数中应该只有一次递归调用。

尾递归的好处是，它可以避免递归调用期间栈空间开销的累积，因为系统可以为每个递归调用重用栈中的固定空间。

例如，对于递归调用序列 `f(x1) -> f(x2) -> f(x3)`，如果函数 `f(x)` 以尾递归的形式实现。那么其执行步骤的顺序和栈空间的分配如下所示：



请注意，在尾递归的情况下，一旦从递归调用返回，我们也会立即返回，因此我们可以跳过整个递归调用返回链，直接返回到原始调用方。这意味着我们根本不需要所有递归调用的调用栈，这为我们节省了空间。

例如，在步骤（1）中，栈中的一个空间将被分配给 `f(x1)`，以便调用 `f(x2)`。然后，在步骤（2）中，函数 `f(x2)` 能够递归地调用 `f(x3)`，但是，系统不需要在栈上分配新的空间，而是可以简单地重用先前分配给第二次递归调用的空间。最后，在函数 `f(x3)` 中，我们达到了基本情况，该函数可以简单地将结果返回给原始调用方，而不会返回到之前的函数调用中。

尾递归函数可以作为非尾递归函数来执行，也就是说，带有调用栈并不会对结果造成影响。**通常，编译器会识别尾递归模式，并优化其执行。然而，并不是所有的编程语言都支持这种优化，比如 C，C++ 支持尾递归函数的优化。另一方面，Java 和 Python 不支持尾递归优化。**

104.二叉树的最大深度（简单）

1. 题目描述

   给定一个二叉树，找出其最大深度。

   二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

   说明: 叶子节点是指没有子节点的节点。

   示例： 给定二叉树 `[3,9,20,null,null,15,7]`，

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

   返回它的最大深度 3 。

2. 递归实现

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root)
            return 0;
        else
            return 1+max(maxDepth(root->left),maxDepth(root->right));
    }
};
```

50.Pow(x, n)（中等）

1. 题目描述

   实现 pow(*x*, *n*)，即计算 x 的 n 次幂函数。

   示例 1:

   ```
   输入: 2.00000, 10
   输出: 1024.00000
   ```

   示例 2:

   ```
   输入: 2.10000, 3
   输出: 9.26100
   ```

   示例 3:

   ```
   输入: 2.00000, -2
   输出: 0.25000
   解释: 2-2 = 1/22 = 1/4 = 0.25
   ```

   说明:

   - -100.0 < *x* < 100.0
   - *n* 是 32 位有符号整数，其数值范围是 [−231, 231 − 1] 。

2. 递归实现

```
class Solution {
public:
    double fastPow(double x, long long n) {
        if (n == 0) {
            return 1.0;
        }
        double half = fastPow(x, n / 2);
        if (n % 2 == 0) {
            return half * half;
        } else {
            return half * half * x;
        }
```

```
    }
    double myPow(double x, int n) {
        long long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        return fastPow(x, N);
    }
};
```

总结 —— 递归 I

现在，您可能会相信递归确实是一种强大的技术，它使我们能够以一种优雅而有效的方式解决许多问题。但是，它仍然不是解决问题的灵丹妙药。由于时间或空间的限制，并不是所有的问题都可以用递归来解决。递归本身可能会带来一些不希望看到的副作用，如堆栈溢出。

在本章中，我们想分享更多关于如何更好地将递归应用于解决现实世界中的问题的提示。

> 当有疑问时，写下**重复出现的关系**。

有时，乍一看，我们并不清楚是否可以应用递归算法来解决问题。然而，由于递归的递推性质与我们所熟悉的数学非常接近，用数学公式来推导某些关系总是有帮助的。通常，利用数学公式，我们可以澄清思想，揭示隐藏的重复出现的关系。在本章中，你可以找到一个有趣的示例，名为不同的二叉搜索树 II，它可以借助数学公式通过递归来解决。

> 只要有可能，就应用**记忆化**。

在起草递归算法时，可以从最简单的策略开始。有时，在递归过程中，可能会出现重复计算的情况，例如斐波纳契数（Fibonacci）。在这种情况下，你可以尝试应用 Memoization 技术，它将中间结果存储在缓存中供以后重用，它可以在空间复杂性上稍加折中，从而极大地提高时间复杂性，因为它可以避免代价较高的重复计算。

> 当堆栈溢出时，尾递归可能会有所帮助。

使用递归实现算法通常有几种方法。尾递归是我们可以实现的递归的一种特殊形式。与记忆化技术不同的是，尾递归通过消除递归带来的堆栈开销，优化了算法的空间复杂度。更重要的是，有了尾递归，就可以避免经常伴随一般递归而来的堆栈溢出问题，而尾递归的另一个优点是，与非尾递归相比，尾部递归更容易阅读和理解。这是由于尾递归不存在调用后依赖（即递归调用是函数中的最后一个动作），这一点不同于非尾递归，因此，只要有可能，就应该尽量运用尾递归。

21.合并两个有序链表（简单）

1. 题目描述

   将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

   示例：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

2. 递归实现

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(!l1) return l2;
        if(!l2) return l1;
        if(l1->val <= l2->val){
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        }
        else{
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }
};
```

779.第K个语法符号（中等）

1. 题目描述

在第一行我们写上一个 `0`。接下来的每一行，将前一行中的 `0` 替换为 `01`，`1` 替换为 `10`。给定行数 `N` 和序数 `K`，返回第 `N` 行中第 `K` 个字符。（`K`从1开始）

例子:

```
输入: N = 1, K = 1
输出: 0

输入: N = 2, K = 1
输出: 0

输入: N = 2, K = 2
输出: 1

输入: N = 4, K = 5
输出: 1

解释:
第一行: 0
第二行: 01
第三行: 0110
第四行: 01101001
**注意: **
```

注意： 1. `N` 的范围 `[1, 30]` 2. `K` 的范围 `[1, 2^(N-1)]`

2. 递归实现

第N行第K个数只与第N-1行第K/2个数有关

```
class Solution {
public:
    int kthGrammar(int N, int K) {
        if(N == 1) return 0;
        int k = (K - 1) / 2 + 1;
        if(kthGrammar(N-1, k) == 0){
            if(K % 2 == 0) return 1;
            else return 0;
        }
        else{
            if(K %2 == 0) return 0;
            else return 1;
        }
    }
};
```

95不同的二叉搜索树 II（中等）

1. 题目描述

   给定一个整数 *n*，生成所有由 1 ... *n* 为节点所组成的**二叉搜索树。**

   示例:

```
输入: 3
输出:
[
  [1,null,3,2],
  [3,2,null,1],
  [3,1,null,null,2],
  [2,1,3],
  [1,null,2,null,3]
]
解释:
以上的输出对应以下 5 种不同结构的二叉搜索树：

   1         3     3      2      1
    \       /     /      / \      \
     3     2     1      1   3      2
    /     /       \                 \
   2     1         2                 3
```

2. 递归实现

   即找到所有中序遍历为1...n的二叉树，那我们可以依次以1...n中的某个值x为根，则其左子树又是一棵中序遍历为1...x-1的二叉树，右子树为一棵x+1...n的二叉树，以此构成循环

```
class Solution {
public:
    vector<TreeNode*> generate(int l, int r){//构建中序遍历为l...r的二叉树
        if(l > r) return {NULL};
        if(l == r) return {new TreeNode(l)};
```

```
        vector<TreeNode*> ans;
        for(int i = l; i <= r; i++){//依次以l...r为根建树
            vector<TreeNode*> left = generate(l, i-1);//左子树的所有可能
            vector<TreeNode*> right = generate(i+1, r);//右子树所有可能
            for(int j = 0; j < left.size(); j++){
                for(int k = 0; k < right.size(); k++){//组合所有可能
                    TreeNode* cur = new TreeNode(i);//每棵树根不同，必须初始化在循环内
                    cur->left = left[j];
                    cur->right = right[k];
                    ans.push_back(cur);
                }
            }
        }
        return ans;
    }
    vector<TreeNode*> generateTrees(int n) {
        if(n == 0) return vector<TreeNode*>();
        return generate(1, n);
    }
};
```