EDA 软件设计 I

Lecture 22

对于适用算法的判断

积累、练习

见的多了，思考多了，是可以快速判断出适用的算法的

“捷径都是最远的路”

动态规划

Dynamic Programming (DP)

一种解决问题的原理/思想

理解思路：递归暴力解法 → 带备忘录的递归解法 → 非递归的动态规划解法

通过一个简单的例子理解DP

简单的例子有利于：

- 将注意力集中在算法背后的普遍思想和技巧
- 不困惑于复杂的细节

Fibonacci数列

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ (对于$n \geq 2$)

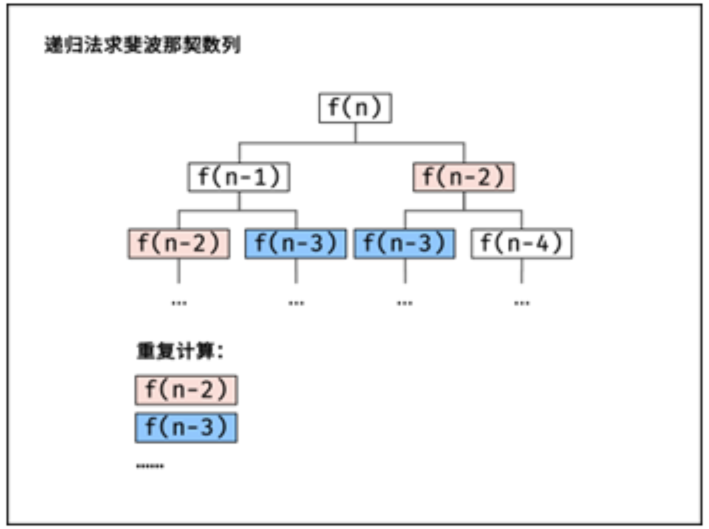
- **最基础的解法：递归算法**

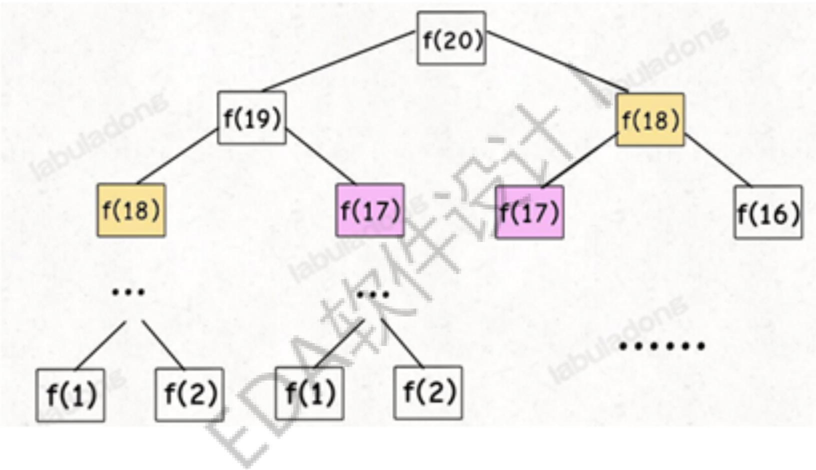
```python
def fibonacci(n):
    if n == 1:
        return 1
    elif n == 0:
        return 0
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

- **暴力递归：简洁易懂，却十分低效**

暴力递归的低效原因

- **递归树：可视化递归算法执行过程的工具，分析算法复杂度时特别有用**
  - 节点：代表一次函数调用（子问题），节点值通常是该调用的输入参数
  - 边：函数调用（子问题）之间的关系
- **递归算法的时间复杂度 = 子问题个数 × 解决一个子问题需要的时间**
  - 斐波那契递归树为二叉树
  - 二叉树节点为指数级别：O(2^n)
  - 解决一个子问题: 只有一个加法操作，时间为 O(1)
  - Fibonacci数列问题暴力递归的时间复杂度：**O(2^n)**
- **暴力递归低效的原因：存在大量重量（冗余）计算——「重叠子问题」**





递归暴力解法 → 带“备忘录”的递归解法

解法解决「重叠子问题」

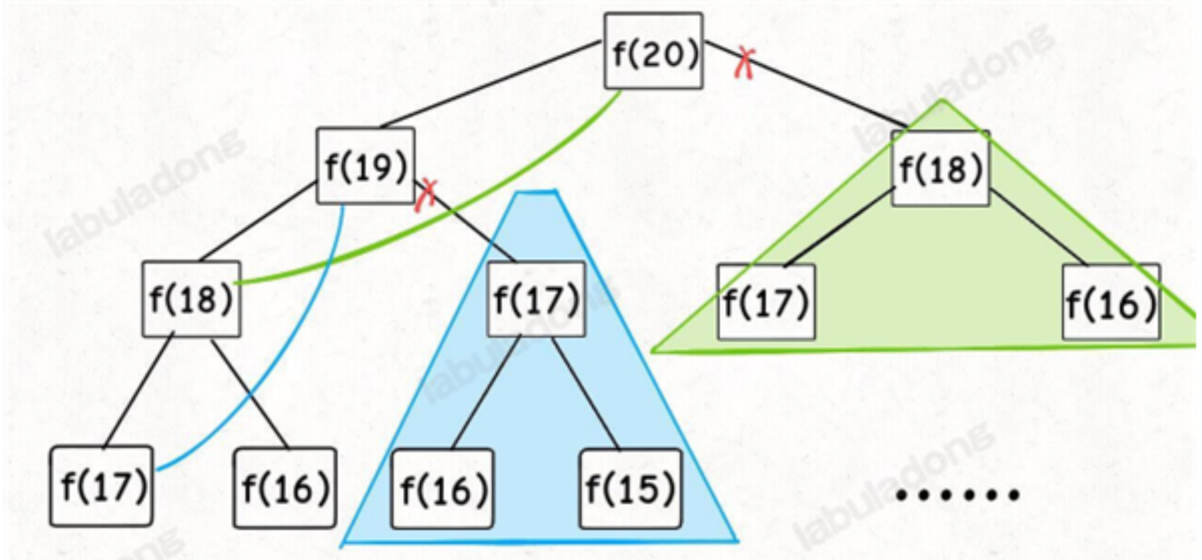
解决「重叠子问题」：记忆化递归

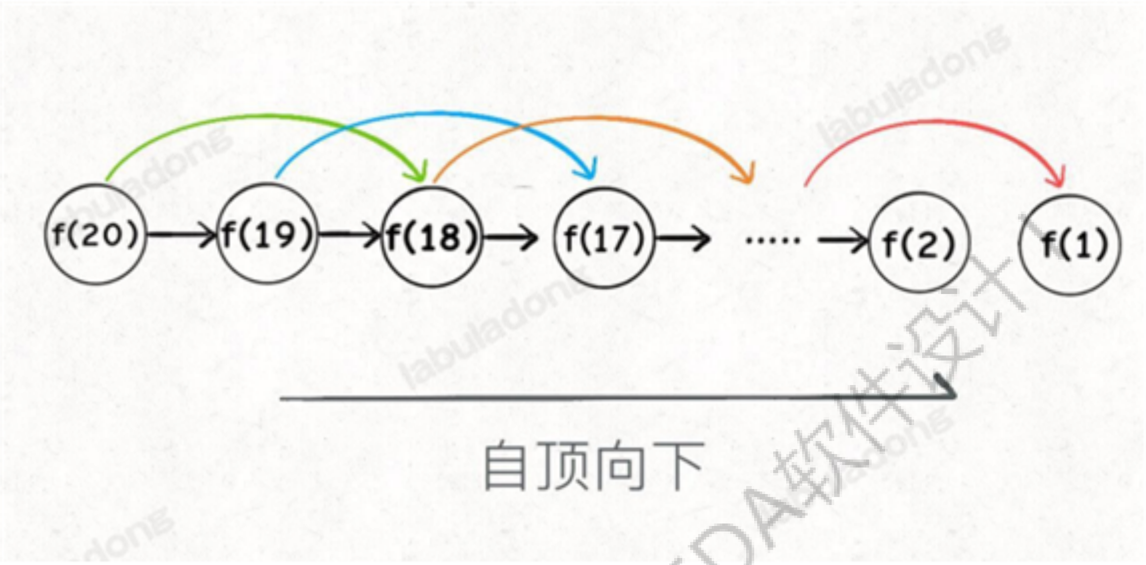
- **带"备忘录"的递归解法（记忆化递归）**
  1. 建立"备忘录"(memorization)：算出某个子问题的答案，存档在"备忘录"里面
  2. 遇到子问题时，先去"备忘录"里面查一下是否已有答案（无需再耗时计算）
- **一般可使用「数组」或者「哈希表」来做这个"备忘录（memo）"**
- **注意：为什么使用helper函数**

```python
def fibonacci_withMemo(N):
    memo = [0] * (N + 1) # 备忘录里每个状态都初始化为0
    # 进行带备忘录的递归
    return helper(memo, N)

# 带着备忘录进行递归
def helper(memo, n):
    # base case
    if n == 0 or n == 1: return n
    # 已经计算过了，不用再计算，直接返回备忘录里面的值
    if  memo[n] != 0: return memo[n]
    # 否则，递归计算
    memo[n] = helper(memo, n - 1) + helper(memo, n - 2)
    return memo[n]
```

记忆化递归

- **带“备忘录”的递归解法（记忆化递归）—— 一种动态规划解法**
  - 在递归树上剪枝（Pruning）：把存在巨量冗余的递归树，剪成一颗不存在冗余的递归图
  - 此时的时间复杂度：
    - 子问题个数：$O(n)$
    - 处理子问题的时间：$O(1)$
    - 时间复杂度：$\mathbf{O(n)}$





带备忘录的递归解法 → 非递归（迭代）的动态规划解法

递归 → 迭代

自顶向下 → 自底向上

动态规划

- **记忆化递归可被视作一种动态规划解法：**
  - 它「自顶向下」地进行「递归」（recursion）求解
  - 「自顶向下」：递归树从上往下延伸，是从一个较大规模的问题，向下逐渐分解规模，直到达到base case，然后逐层返回答案
- **(非递归的）动态规划解法：**
  - 它「自底向上」进行「递推（迭代）」（iteration）求解
  - 「自底向上」：直接从最底下、最简单、问题规模最小的base case开始往上推，直到推到我们想要的答案

动态规划

- **动态规划key point：**
  - 类似于“备忘录”，它利用一张表（dp table），在这张表上完成「自底向上」的推算
- **思考：是否可以去掉第2、3行?**

```python
1  def fibonacci_dp(N):
2      if N == 0:
3          return 0
4
5      # 创建dp table
6      dp_table = [0] * (N+1)
7
8      # base case
9      dp_table[0] = 0
10     dp_table[1] = 1
11
12     # 状态转移 （iteration）
13     for i in range(2, N+1):
14         dp_table[i] = dp_table[i-1] + dp_table[i-2]
15
16     return dp_table[N]
```

动态规划迭代过程

```python
# base case
dp_table[0] = 0
dp_table[1] = 1

# 状态转移 （iteration）
for i in range(2, N+1):
    dp_table[i] = dp_table[i-1] + dp_table[i-2]
```

	f(0)	**f(1)**
Index:	0	1
dp_table:	**0**	**1**

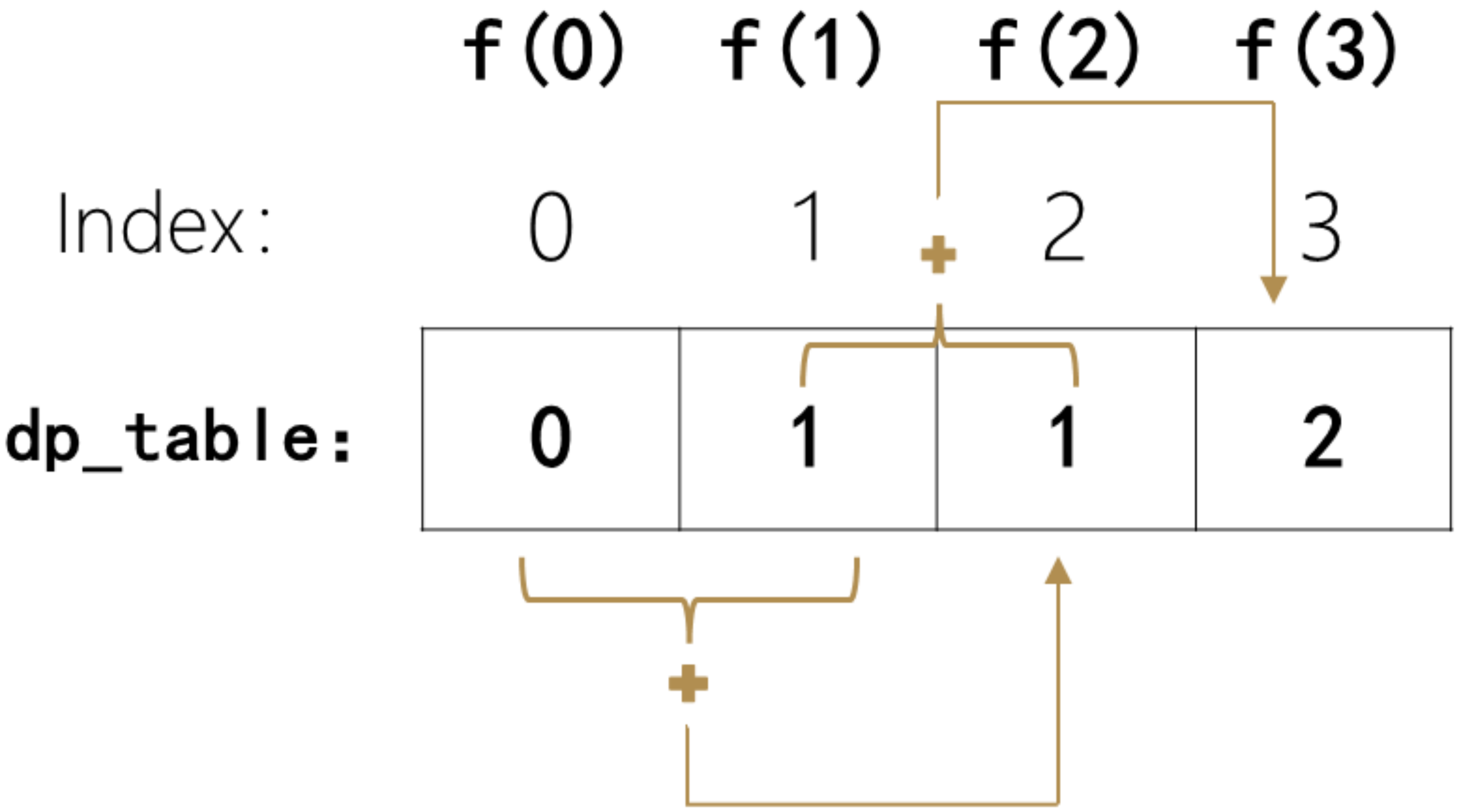
动态规划迭代过程

```python
    # base case
    dp_table[0] = 0
    dp_table[1] = 1

    # 状态转移 （iteration）
    for i in range(2, N+1):
        dp_table[i] = dp_table[i-1] + dp_table[i-2]
```

	f(0)	f(1)	f(2)
Index:	0	1	2
dp_table:	**0**	**1**	**1**

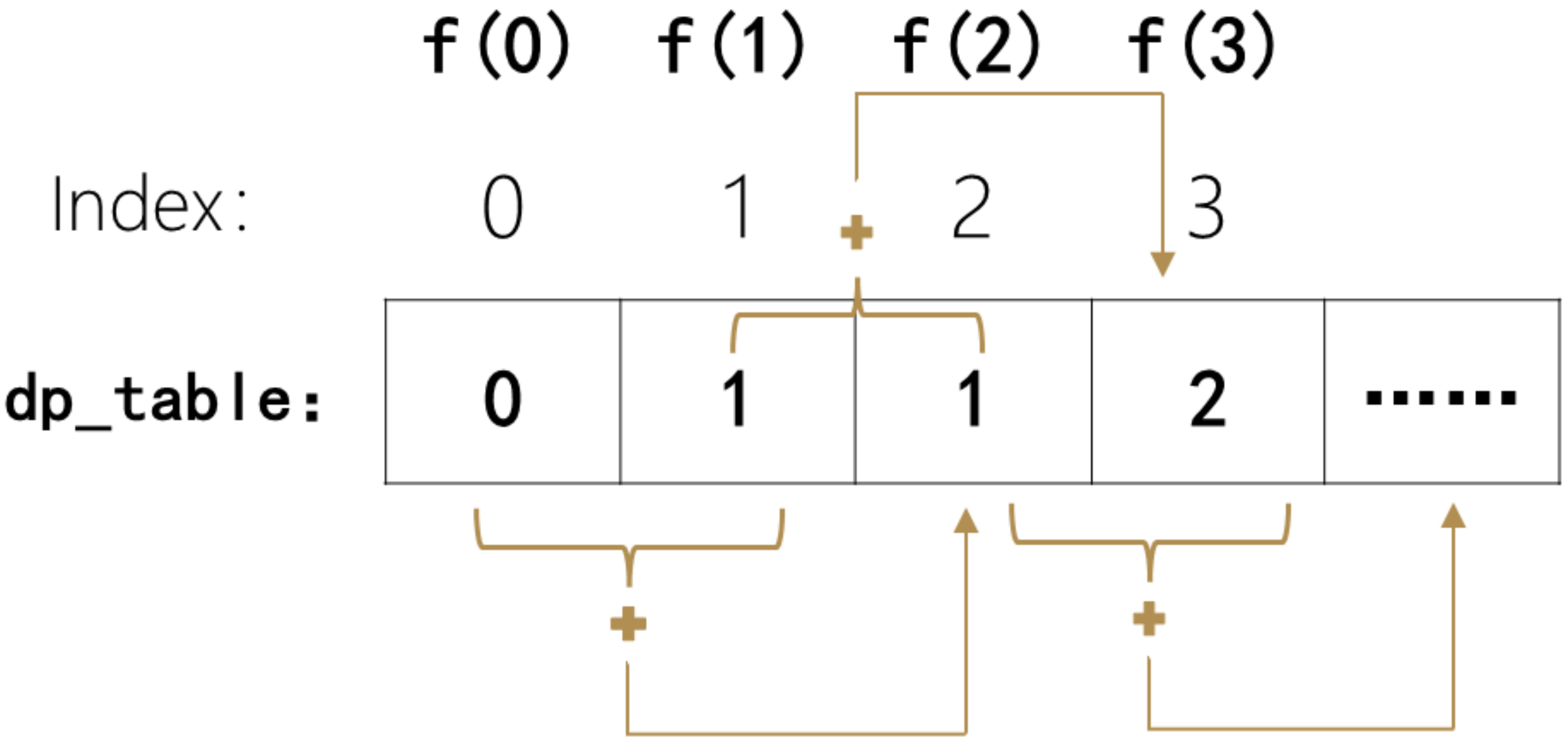
动态规划迭代过程

```python
# base case
dp_table[0] = 0
dp_table[1] = 1

# 状态转移 （iteration）
for i in range(2, N+1):
    dp_table[i] = dp_table[i-1] + dp_table[i-2]
```



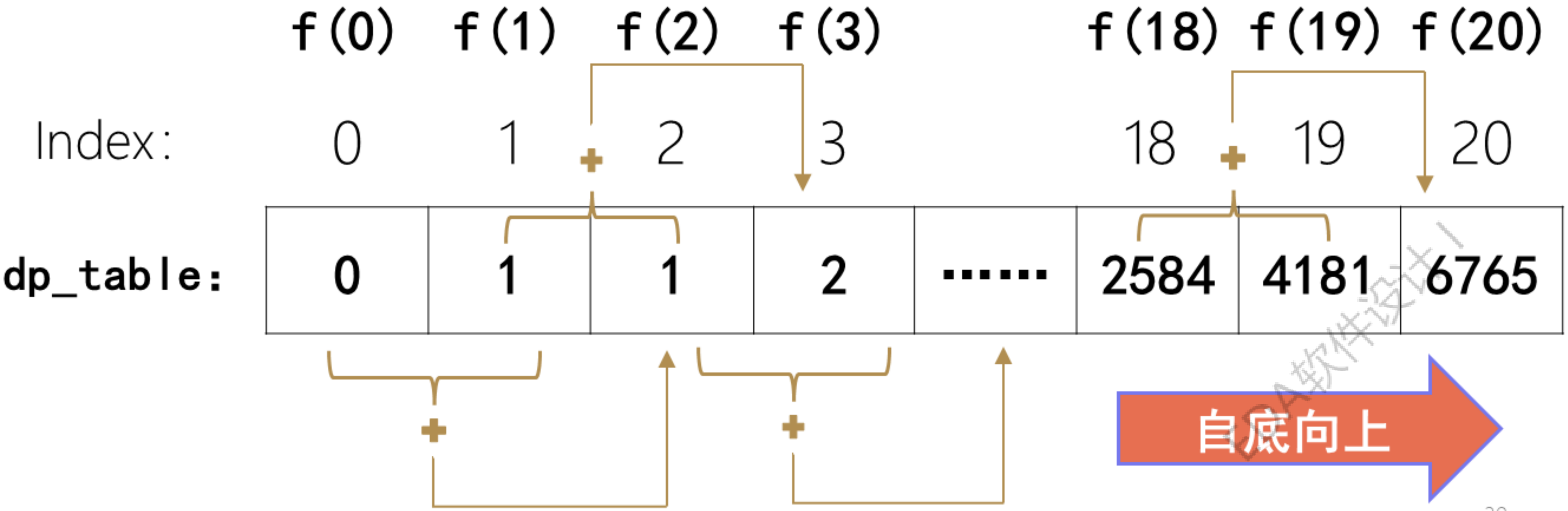
动态规划迭代过程

```python
# base case
dp_table[0] = 0
dp_table[1] = 1

# 状态转移 (iteration)
for i in range(2, N+1):
    dp_table[i] = dp_table[i-1] + dp_table[i-2]
```



动态规划迭代过程

```python
# base case
dp_table[0] = 0
dp_table[1] = 1

# 状态转移 （iteration）
for i in range(2, N+1):
    dp_table[i] = dp_table[i-1] + dp_table[i-2]
```



动态规划核心要素

💡 状态转移方程：描述问题结构的数学形式

- Particularly，在Fibonacci数列问题中，状态转移方程可写成：

$$dp[i] = dp[i-1] + dp[i-2], i \geq 2$$

  - 理解：你把状态参数 i 想做一个状态，这个状态 i 是由状态 $i-1$ 和状态 $i-2$ 转移（相加）而来，这就叫状态转移
- Generally
  - 状态定义：$dp[i]$ 代表什么含义
  - 状态转移方程：$dp[i]$ 和 $dp[i-1]$等之间的关系
  - 初始状态：最基础的状态值（base case）

动态规划核心要素

1. 核心思想

- 把大问题分解成小问题（子问题）
- 通过解决子问题来解决大问题
- 储存子问题的解以避免重复计算

2. 设计步骤

① 写出暴力解

② 优化：

- 使用“备忘录”，自顶向下的记忆化搜索，or
- 使用DP table，自底向上的迭代算法

3. 使用条件?

经典优化问题——背包问题

Knapsack Problem

背包问题

A. 0-1背包问题：

- 给定一组物品，每个物品都有一个特定的重量和价值。目标是选择一些物品放入背包中，使得背包的总重量不超过给定的容量，同时总价值最大。每个物品只能选择一次，即“0-1”表示每个物品要么被选中（1），要么不被选中（0）

B. 分数背包问题：

- 在分数背包问题中，物品可以被分割，允许将物品的一部分放入背包。目标仍然是最大化背包中的总价值，且背包的总重量不超过给定的容量

- **可以用贪心策略or动态规划求解?**

DP：斐波那契数列细节优化

```python
def fib(n):
    dp = [0] * (n + 1)
    dp[0], dp[1] = 0, 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]
```

- **时间复杂度：O(n)**
- **空间复杂度：O(n)**
- **可否优化空间复杂度?**

- **当前状态 n 只和之前的 n-1, n-2 两个状态有关，并不需要那么长的DP table存储所有状态，只要想办法存储之前的两个状态就行了**
- **把空间复杂度降为 O(1)**

```python
def fib(n):
    if n <= 1:
        return n
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b
```