刷题笔记 - python - (44-68题)

数字序列中某一位的数字:

1. 将 $101112\cdots$ 中的每一位称为 **数位** ，记为 n ；
2. 将 $10, 11, 12, \cdots$ 称为 **数字** ，记为 num ；
3. 数字 10 是一个两位数，称此数字的 **位数** 为 2 ，记为 $digit$ ；
4. 每 $digit$ 位数的起始数字（即：$1, 10, 100, \cdots$），记为 $start$ 。

数字范围	位数	数字数量	数位数量
1~9	1	9	9
10~99	2	90	180
100~999	3	900	2700
...	...	...	...
start~*end*	*digit*	$9 \times start$	$9 \times start \times digit$

位数递推公式　$digit = digit + 1$

起始数字递推公式　$start = start \times 10$

数位数量计算公式　$count = 9 \times start \times digit$

观察上表，可推出各 $digit$ 下的数位数量 $count$ 的计算公式：$count = 9 \times start \times digit$

根据以上分析，可将求解分为三步：

1. 确定 n 所在 **数字** 的 **位数** ，记为 $digit$ ；
2. 确定 n 所在的 **数字** ，记为 num ；
3. 确定 n 是 num 中的哪一数位，并返回结果。

1. 确定所求数位的所在数字的位数

如下图所示，循环执行 n 减去 一位数、两位数、... 的数位数量 $count$ ，直至 $n \leq count$ 时跳出。

由于 n 已经减去了一位数、两位数、...、$(digit-1)$ 位数的 数位数量 $count$ ，因而此时的 n 是从起始数字 $start$ 开始计数的。

```
digit, start, count = 1, 1, 9
while n > count:
    n -= count
    start *= 10 # 1, 10, 100, ...
    digit += 1  # 1,  2,  3, ...
    count = 9 * start * digit # 9, 180, 2700, ...

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shu-zi-xu-lie-zhong-mou-yi-wei-de-shu-zi-lcof/solution/mian-shi-ti-44-shu-zi-xu-lie-zhong-mou-yi-wei-de-6/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

结论： 所求数位 ① 在某个 $digit$ 位数中； ② 为从数字 $start$ 开始的第 n 个数位。

1. 确定 n 所在数字的位数

数字范围	位数	数位数量	$n = n - count$
1~9	1	9	$n - 9$
10~99	2	180	$n - 9 - 180$
100~999	3	2700	$n - 9 - 180 - 2700$
$start$~end	$digit$	$9 \times start \times digit$	直至 $n \le count$

1. 所求数位 在某个 $digit$ 位数中；
2. 所求数位 为从数字 $start$ 开始的第 n 个数位。

2. 确定所求数位所在的数字

如下图所示，所求数位 在从数字 $start$ 开始的第 $[(n-1)/digit]$ 个 **数字** 中（ $start$ 为第 0 个数字）。

```
num = start + (n - 1) // digit
```

结论： 所求数位在数字 num 中。

2. 确定 n 所在的数字 num

$$digit = 2 \qquad start = 10$$

	1	0	1	1	1	2	1	3	...
n	1	2	3	4	5	6	7	8	...
$(n-1)$	0	1	2	3	4	5	6	7	...
$(n-1)/2$	**0**	**0**	**1**	**1**	**2**	**2**	**3**	**3**	...

所求数位 在数字 $num = start + (n-1)/digit$ 中

3. 确定所求数位在 num 的哪一数位

如下图所示，所求数位为数字 num 的第 $(n-1)\%digit$ 位（数字的首个数位为第 0 位）。

```
s = str(num) # 转化为 string
res = int(s[(n - 1) % digit]) # 获得 num 的 第 (n - 1) % digit 个数位，并转化为 int
```

结论： 所求数位是 res 。

3. 确定 n 在 num 的哪一数位

$$digit = 2 \qquad start = 10$$

	1	0	1	1	1	2	1	3	...
n	1	2	3	4	5	6	7	8	...
$(n-1)$	0	1	2	3	4	5	6	7	...
$(n-1)\ \%\ 2$	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	...

所求数位 在 num 的第 $(n-1)\ \%\ 2$ 位

```python
class Solution:
    def findNthDigit(self, n: int) -> int:
        digit, start, count = 1, 1, 9
        while n > count: # 1.
            n -= count
            start *= 10
            digit += 1
            count = 9 * start * digit
        num = start + (n - 1) // digit # 2.
        return int(str(num)[(n - 1) % digit]) # 3.

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shu-zi-xu-lie-zhong-mou-yi-wei-de-shu-zi-lcof/solution/mian-shi-ti-44-shu-zi-xu-lie-zhong-mou-yi-wei-de-6/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

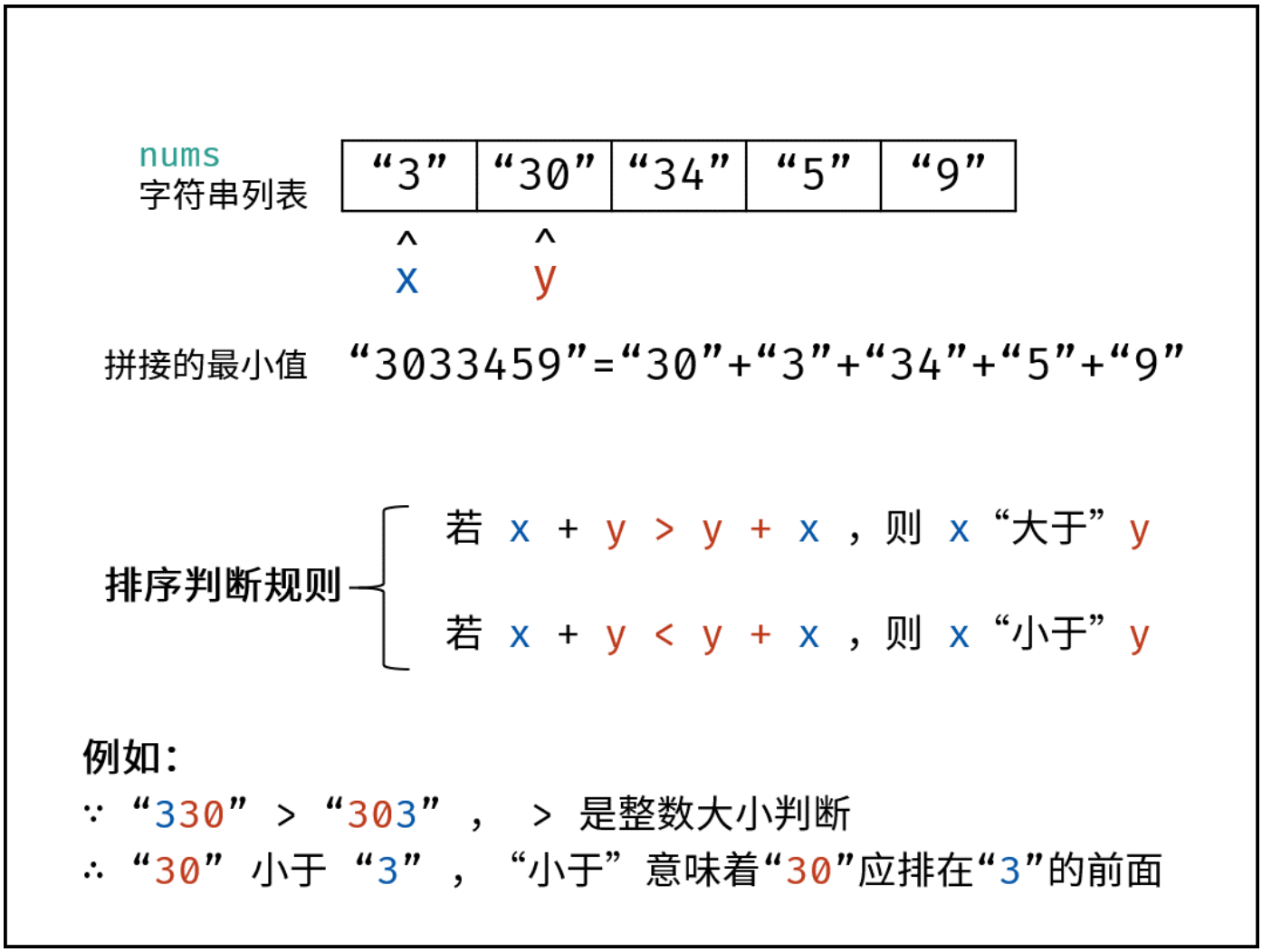
把数组排成最小的数:

此题求拼接起来的最小数字，本质上是一个排序问题。设数组 $nums$ 中任意两数字的字符串为 x 和 y ，则规定 **排序判断规则** 为：

- 若拼接字符串 $x + y > y + x$，则 x “大于” y ；
- 反之，若 $x + y < y + x$ ，则 x “小于” y ；

这里，x “小于” y 代表：排序完成后，数组中 x 应在 y 左边；“大于” 则反之。

根据以上规则，套用任何排序方法对 $nums$ 执行排序即可。



算法流程

1. **初始化**： 字符串列表 $strs$ ，保存各数字的字符串格式；
2. **列表排序**： 应用以上 “排序判断规则” ，对 $strs$ 执行排序；
3. **返回值**： 拼接 $strs$ 中的所有字符串，并返回。

我们使用**快速排序**方法。

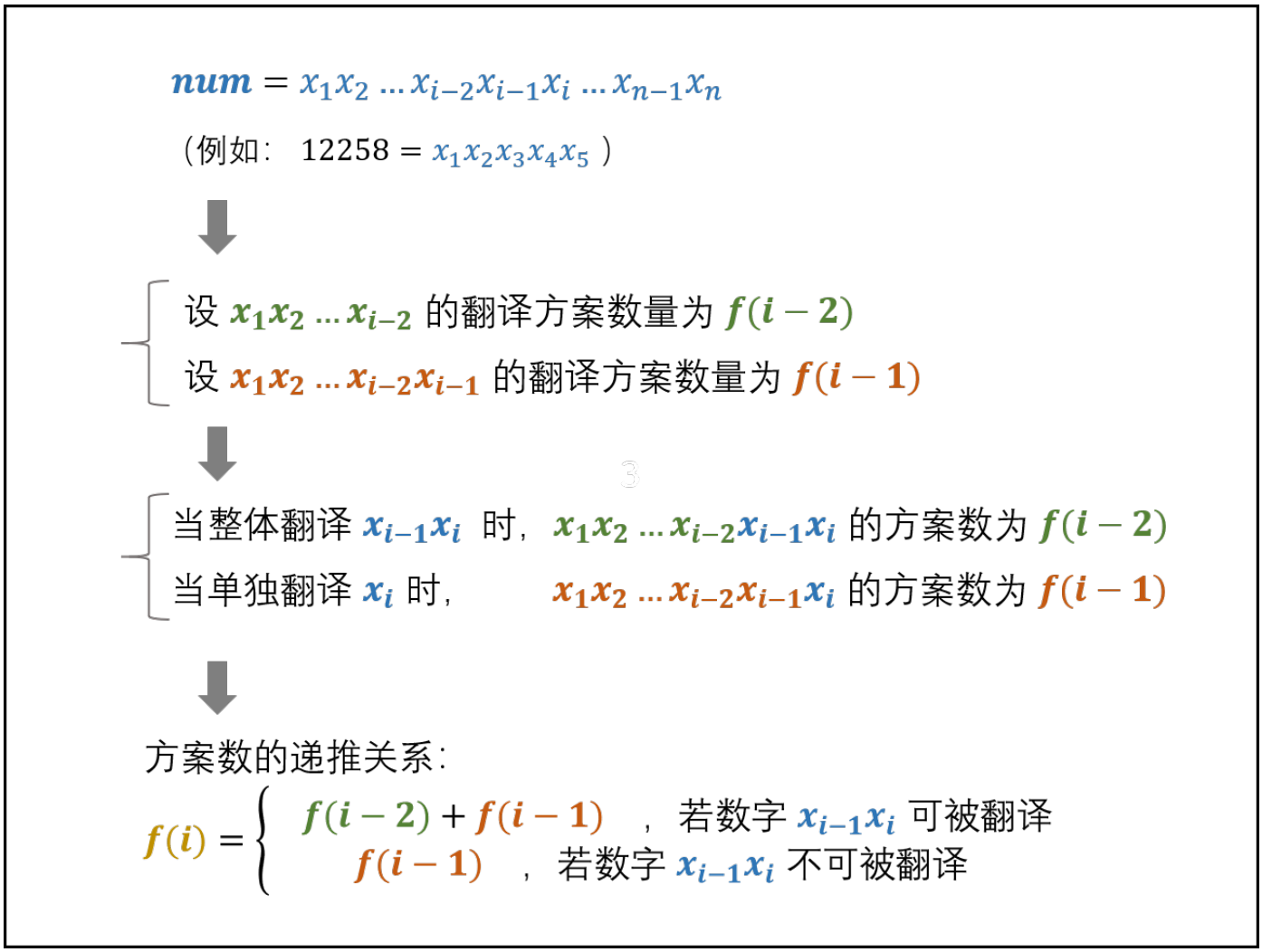
```
class Solution:
    def minNumber(self, nums: List[int]) -> str:
        def quick_sort(l , r):
            if l >= r: return
            i, j = l, r
            while i < j:
                while strs[j] + strs[l] >= strs[l] + strs[j] and i < j: j -= 1
                while strs[i] + strs[l] <= strs[l] + strs[i] and i < j: i += 1
                strs[i], strs[j] = strs[j], strs[i]
            strs[i], strs[l] = strs[l], strs[i]
            quick_sort(l, i - 1)
            quick_sort(i + 1, r)

        strs = [str(num) for num in nums]
```

```
        quick_sort(0, len(strs) - 1)
        return ''.join(strs)

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/ba-shu-zu-pai-cheng-zui-xiao-de-shu-lcof/solution/mian-shi-ti-45-ba-shu-zu-pai-cheng-zui-xiao-de-s-4/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

把数字翻译成字符串：

根据题意，可按照下图的思路，总结出 “递推公式” （即转移方程）。

因此，此题可用**动态规划**解决，以下按照流程解题。



动态规划解析：

记数字 num 第 i 位数字为 x_i ，数字 num 的位数为 n ；

例如： $num = 12258$ 的 $n = 5$, $x_1 = 1$ 。

- **状态定义**： 设动态规划列表 dp ， $dp[i]$ 代表以 x_i 为结尾的数字的翻译方案数量。
- **转移方程**： 若 x_i 和 x_{i-1} 组成的两位数字可以被翻译，则 $dp[i] = dp[i-1] + dp[i-2]$ ；否则 $dp[i] = dp[i-1]$ 。

- 可被翻译的两位数区间：当 $x_{i-1}=0$ 时，组成的两位数是无法被翻译的（例如 $00,01,02,\cdots$），因此区间为 $[10,25]$ 。

  $$dp[i]=\begin{cases} dp[i-1]+dp[i-2] & ,10x_{i-1}+x_i\in[10,25] \\ dp[i-1] & ,10x_{i-1}+x_i\in[0,10)\cup(25,99] \end{cases}$$

- **初始状态**：$dp[0]=dp[1]=1$，即“无数字”和“第 1 位数字”的翻译方法数量均为 1；
- **返回值**：$dp[n]$，即此数字的翻译方案数量。

> **Q**：无数字情况 $dp[0]=1$ 从何而来?
>
> **A**：当 num 第 $1,2$ 位的组成的数字 $\in[10,25]$ 时，显然应有 2 种翻译方法，即 $dp[2]=dp[1]+dp[0]=2$，而显然 $dp[1]=1$，因此推出 $dp[0]=1$ 。

方法一：字符串遍历

- 为方便获取数字的各位 x_i，考虑先将数字 num 转化为字符串 s，通过遍历 s 实现动态规划。
- 通过字符串切片 $s[i-2:i]$ 获取数字组合 $10x_{i-1}+x_i$，通过对比字符串 ASCII 码判断字符串对应的数字区间。
- **空间使用优化**：由于 $dp[i]$ 只与 $dp[i-1]$ 有关，因此可使用两个变量 a,b 分别记录 $dp[i],dp[i-1]$，两变量交替前进即可。此方法可省去 dp 列表使用的 $O(N)$ 的额外空间。

```
class Solution:
    def translateNum(self, num: int) -> int:
        s = str(num)
        a = b = 1
        for i in range(2, len(s) + 1):
            a, b = (a + b if "10" <= s[i - 2:i] <= "25" else a), a
        return a

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/solution/mian-shi-ti-46-ba-shu-zi-fan-yi-cheng-zi-fu-chua-6/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

方法二：数字求余

- 上述方法虽然已经节省了 dp 列表的空间占用，但字符串 s 仍使用了 $O(N)$ 大小的额外空间。

空间复杂度优化：

- 利用求余运算 $num\%10$ 和求整运算 $num//10$，可获取数字 num 的各位数字（获取顺序为个位、十位、百位...）。
- 因此，可通过 **求余** 和 **求整** 运算实现 **从右向左** 的遍历计算。而根据上述动态规划“对称性”，可知从右向左的计算是正确的。
- 自此，字符串 s 的空间占用也被省去，空间复杂度从 $O(N)$ 降至 $O(1)$ 。

```
class Solution:
    def translateNum(self, num: int) -> int:
        a = b = 1
        y = num % 10
```

```
        while num != 0:
            num //= 10
            x = num % 10
            a, b = (a + b if 10 <= 10 * x + y <= 25 else a), a
            y = x
        return a

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/ba-shu-zi-fan-yi-cheng-zi-fu-chuan-lcof/solution/mian-shi-ti-46-ba-shu-zi-fan-yi-cheng-zi-fu-chua-6/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

礼物的最大价值：动态规划

根据题目说明，易得某单元格只可能从上边单元格或左边单元格到达。(逆向思维)

这道理参考了K神的思想，但是是我自己写的代码。

重点在于转移方程的简化：

$$
dp(i,j) = \begin{cases} grid(i,j), & \text{if } i = 0, j = 0 \\ grid(i,j) + dp(i,j-1), & \text{if } i = 0, j \neq 0 \\ grid(i,j) + dp(i-1,j), & \text{if } i \neq 0, j = 0 \\ grid(i,j) + \max[dp(i-1,j), dp(i,j-1)], & \text{if } i \neq 0, j \neq 0 \end{cases}
$$

简化为dp(i, j)=grid(i, j) + max[bool(j) * dp(i, j - 1), bool(i) * dp(i - 1, j])

```
class Solution:
    def maxValue(self, grid: List[List[int]]) -> int:
        m, n = len(grid),len(grid[0])
        dp = [[ 0 for _ in range(n)] for _ in range(m)]
        for i in range(m):
            for j in range(n):
                dp[i][j]=grid[i][j]+max(bool(j)*dp[i][j-1],bool(i)*dp[i-1][j])
        return dp[m-1][n-1]
```

K神还有一个解法，当 grid 矩阵很大时， i=0 或 j=0 的情况仅占极少数，相当循环每轮都冗余了一次判断。因此，可先初始化矩阵第一行和第一列，再开始遍历递推。

```
class Solution:
    def maxValue(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        for j in range(1, n): # 初始化第一行
            grid[0][j] += grid[0][j - 1]
        for i in range(1, m): # 初始化第一列
            grid[i][0] += grid[i - 1][0]
        for i in range(1, m):
            for j in range(1, n):
                grid[i][j] += max(grid[i][j - 1], grid[i - 1][j])
```

```
        return grid[-1][-1]

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/li-wu-de-zui-da-jie-zhi-lcof/solution/mian-shi-ti-47-li-wu-de-zui-da-jie-zhi-dong-tai-gu/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

最长不含重复字符的子字符串:动态规划 / 双指针 + 哈希表

如果硬做，暴力解法的复杂度为$O(N^3)$，所以使用动态规划解决问题

状态定义： 设动态规划列表 dp ，$dp[j]$ 代表以字符 $s[j]$ 为结尾的 “最长不重复子字符串” 的长度。

转移方程： 固定右边界 j ，设字符 $s[j]$ 左边距离最近的相同字符为 $s[i]$ ，即 $s[i] = s[j]$ 。

1. 当 $i < 0$ ，即 $s[j]$ 左边无相同字符，则 $dp[j] = dp[j-1] + 1$;
2. 当 $dp[j-1] < j - i$，说明字符 $s[i]$ 在子字符串 $dp[j-1]$ 区间之外 ，则 $dp[j] = dp[j-1] + 1$;
3. 当 $dp[j-1] \geq j - i$，说明字符$s[i]$ 在子字符串 $dp[j-1]$ 区间之中 ，则 $dp[j]$ 的左边界由 $s[i]$ 决定，即 $dp[j] = j - i$;

当 $i < 0$ 时，由于 $dp[j-1] \leq j$ 恒成立，因而 $dp[j-1] < j - i$恒成立，因此分支 `1.` 和 `2.` 可被合并。

dp[j] = dp[j - 1] + 1, if dp[j - 1] < j - 1

j - i. , if dp[j - 1] >= j - 1

返回值： $\max(dp)$ ，即全局的 “最长不重复子字符串” 的长度。

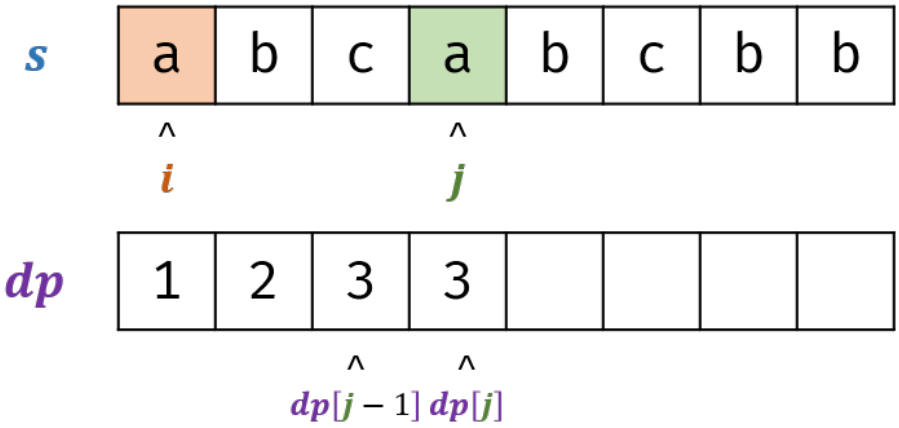
状态定义：

$dp[j]$ 代表以字符 $s[j]$ 为结尾的 “最长不重复子字符串” 的长度。

转移方程：

设字符 $s[j]$ 左边距离最近的相同字符为 $s[i]$ 。

$$dp[j] = \begin{cases} dp[j-1] + 1 & , dp[j-1] < j - i \\ j - i & , dp[j-1] \geq j - i \end{cases}$$



空间复杂度优化，我参考了**双指针 + 哈希表**

- 哈希表 dic 统计： 指针 j 遍历字符 s ，哈希表统计字符 $s[j]$ 最后一次出现的索引 。
- 更新左指针 i ： 根据上轮左指针 i 和 $dic[s[j]]$，每轮更新左边界 i ，保证区间 $[i+1, j]$ 内无重复字符且最大。

  i = max(dic[s[j]], i)
- **更新结果 res** ： 取上轮 res 和本轮双指针区间 $[i+1, j]$的宽度（即 $j - i$ ）中的最大值。

  res = max(res, j - i)

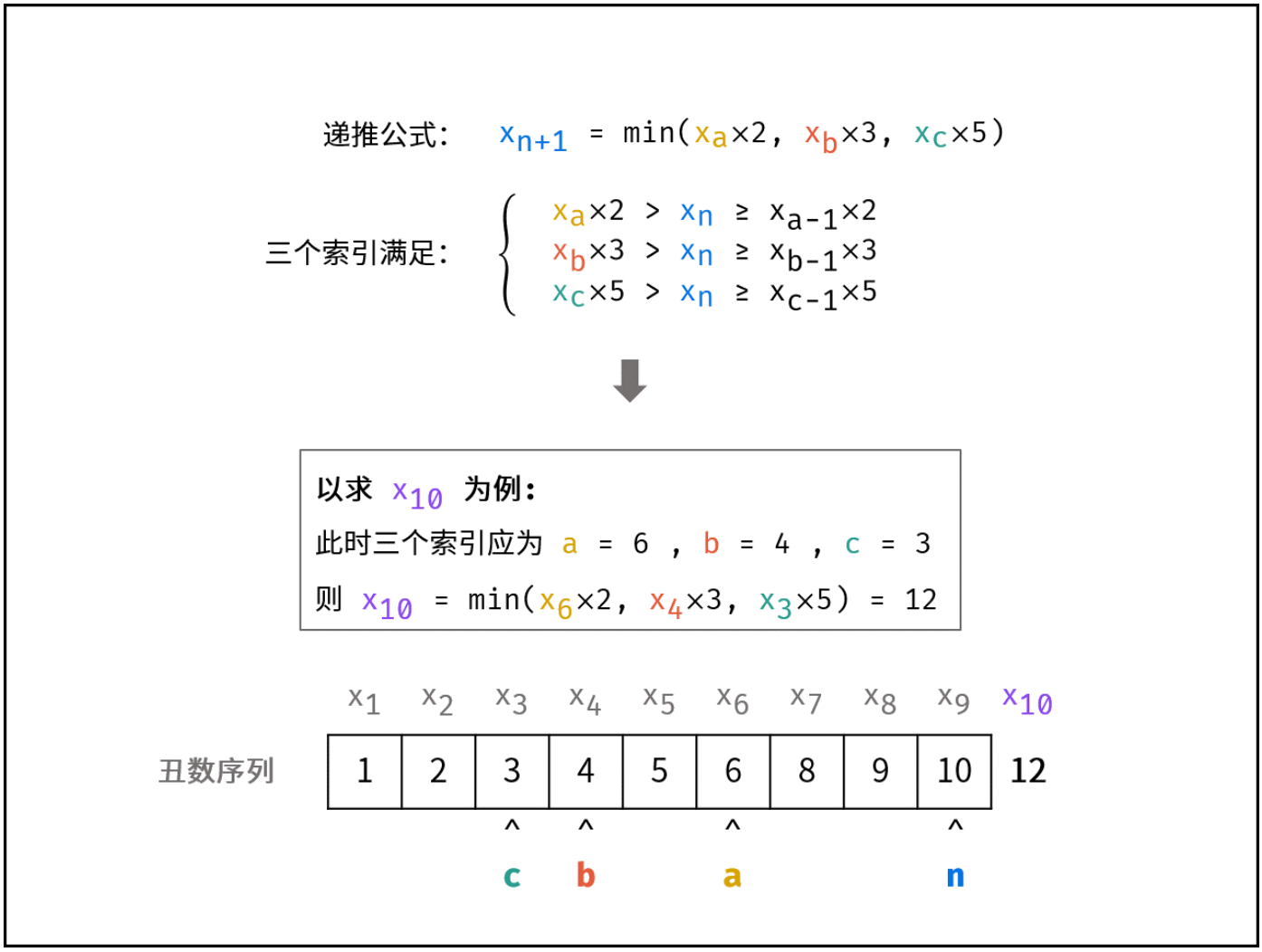
```
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        dic, res, i = {}, 0, -1
        for j in range(len(s)):
            if s[j] in dic:
                i = max(dic[s[j]], i) # 更新左指针 i
            dic[s[j]] = j # 哈希表记录
            res = max(res, j - i) # 更新结果
        return res

# 作者：jyd
```

```
# 链接：https://leetcode-cn.com/problems/zui-chang-bu-han-zhong-fu-zi-fu-de-zi-zi-fu-chuan-lcof/solution/mian-shi-ti-48-zui-chang-bu-han-zhong-fu-zi-fu-d-9/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

丑数：

这里有一个诀窍：**丑数一定是前面某一个丑数乘以2、3或者5得到的。**

如果要排序的话，下一个丑数一定是上面三种情况的最小值。



所以还是用动态规划。

- **状态定义**： 设动态规划列表 dp ， $dp[i]$ 代表第 $i+1$ 个丑数；
- **转移方程**：
  1. 当索引 a, b, c 满足以下条件时， $dp[i]$ 为三种情况的最小值；
  2. 每轮计算 $dp[i]$ 后，需要更新索引 a, b, c 的值，使其始终满足方程条件。实现方法：**分别独立判断** $dp[i]$ 和 $dp[a] \times 2$, $dp[b] \times 3$, $dp[c] \times 5$ 的大小关系，若相等则将对应索引 a , b , c 加 1 ；

$$\begin{cases} dp[a] \times 2 > dp[i-1] \geq dp[a-1] \times 2 \\ dp[b] \times 3 > dp[i-1] \geq dp[b-1] \times 3 \\ dp[c] \times 5 > dp[i-1] \geq dp[c-1] \times 5 \end{cases}$$

$$dp[i] = \min(dp[a] \times 2, dp[b] \times 3, dp[c] \times 5)$$

- **初始状态**：$dp[0] = 1$，即第一个丑数为 1；
- **返回值**：$dp[n-1]$，即返回第 n 个丑数；

```
class Solution:
    def nthUglyNumber(self, n: int) -> int:
        dp, a, b, c = [1] * n, 0, 0, 0
        for i in range(1, n):
            n2, n3, n5 = dp[a] * 2, dp[b] * 3, dp[c] * 5
            dp[i] = min(n2, n3, n5)
            if dp[i] == n2: a += 1
            if dp[i] == n3: b += 1
            if dp[i] == n5: c += 1
        return dp[-1]

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/chou-shu-lcof/solution/mian-shi-ti-49-chou-shu-dong-tai-gui-hua-qing-xi-t/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

第一个只出现一次的字符:

1. 遍历字符串 `s` ，使用哈希表统计 “各字符数量是否 > 1 ”。
2. 再遍历字符串 `s` ，在哈希表中找到首个 “数量为 1 的字符”，并返回。

算法流程：

- 新出现的字符，添加进`dic`，设置为`True`
- 再出现一次的字符，设置为`False`0
    - 最后返回`value`是`True`的`dic`第一个值，特殊情况是输入`''`返回`' '`

```
class Solution:
    def firstUniqChar(self, s: str) -> str:
        if s == '': return ' '
        dic = dict()
        for i in range(len(s)):
            if s[i] not in dic: dic[s[i]] = True
            else: dic[s[i]] = False
        for k in dic:
            if dic[k] == True: return k
        return ' '
```

数组中的逆序对：归并排序法

归并排序是分治思想的典型应用，它包含这样三个步骤：

- **分解**：待排序的区间为 $[l, r]$，令 $m = \lfloor \frac{l+r}{2} \rfloor$，我们把 $[l, r]$分成 $[l, m]$和$[m+1, r]$
- **解决**：使用归并排序递归地排序两个子序列

- **合并**：把两个已经排好序的子序列 $[l, m]$ 和 $[m+1, r]$合并起来

在待排序序列长度为 1 的时候，递归开始「回升」，因为我们默认长度为 1 的序列是排好序的。

那么求逆序对和归并排序又有什么关系呢？关键就在于「归并」当中「并」的过程。我们通过一个实例来看看。假设我们有两个已排序的序列等待合并，分别是$L = \{8, 12, 16, 22, 100\}$和 $R = \{9, 26, 55, 64, 91\}$。一开始我们用指针 `lPtr = 0` 指向 L 的首部，`rPtr = 0` 指向 R 的头部。记已经合并好的部分为 M。

```
L = [8, 12, 16, 22, 100]   R = [9, 26, 55, 64, 91]  M = []
     |                          |
   lPtr                       rPtr
```

我们发现 `lPtr` 指向的元素小于 `rPtr` 指向的元素，于是把 `lPtr` 指向的元素放入答案，并把 `lPtr` 后移一位。

```
L = [8, 12, 16, 22, 100]   R = [9, 26, 55, 64, 91]  M = [8]
        |                          |
      lPtr                       rPtr
```

这个时候我们把左边的 8 加入了答案，我们发现右边没有数比 8 小，所以 8 对逆序对总数的「贡献」为 0。

接着我们继续合并，把 9 加入了答案，此时 `lPtr` 指向 12，`rPtr` 指向 26。

```
L = [8, 12, 16, 22, 100]   R = [9, 26, 55, 64, 91]  M = [8, 9]
        |                             |
      lPtr                          rPtr
```

此时 `lPtr` 比 `rPtr` 小，把 `lPtr` 对应的数加入答案，并考虑它对逆序对总数的贡献为 `rPtr` 相对 R 首位置的偏移 1（即右边只有一个数比 12 小，所以只有它和 12 构成逆序对），以此类推。

我们发现用这种「算贡献」的思想在合并的过程中计算逆序对的数量的时候，只在 `lPtr` 右移的时候计算，是基于这样的事实：当前 `lPtr` 指向的数字比 `rPtr` 小，但是比 R 中 `[0 ... rPtr - 1]` 的其他数字大，`[0 ... rPtr - 1]` 的其他数字本应当排在 `lPtr` 对应数字的左边，但是它排在了右边，所以这里就贡献了 `rPtr` 个逆序对。

```python
class Solution:
    def mergeSort(self, nums, tmp, l, r):
        if l >= r:
            return 0

        mid = (l + r) // 2
        inv_count = self.mergeSort(nums, tmp, l, mid) + self.mergeSort(nums, tmp, mid + 1, r)
        i, j, pos = l, mid + 1, l
        while i <= mid and j <= r:
            if nums[i] <= nums[j]:
                tmp[pos] = nums[i]
                i += 1
                inv_count += (j - (mid + 1)) # 新增贡献值
```

```
            else:
                tmp[pos] = nums[j]
                j += 1
            pos += 1
        # 把(i, mid + 1)这部分数接到tmp
        for k in range(i, mid + 1):
            tmp[pos] = nums[k]
            inv_count += (j - (mid + 1)) # 新增贡献值
            pos += 1
        # 把(j, r + 1)这部分数接到tmp
        for k in range(j, r + 1):
            tmp[pos] = nums[k]
            pos += 1
        nums[l:r+1] = tmp[l:r+1]
        return inv_count # 统计贡献值

    def reversePairs(self, nums: List[int]) -> int:
        n = len(nums)
        tmp = [0] * n
        return self.mergeSort(nums, tmp, 0, n - 1)

# 作者：LeetCode-Solution
# 链接：https://leetcode-cn.com/problems/shu-zu-zhong-de-ni-xu-dui-lcof/solution/shu-zu-zhong-de-ni-xu-dui-by-leetcode-solution/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

两个链表的第一个公共节点:

一开始觉得使用栈即可，但是有一个例外是：**并不是第一个一样的节点就是公共节点**，这是一个坑。

```
class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        node1, node2 = headA, headB

        while node1 != node2:
            node1 = node1.next if node1 else headB
            node2 = node2.next if node2 else headA

        return node1

# 作者：z1m
# 链接：https://leetcode-cn.com/problems/liang-ge-lian-biao-de-di-yi-ge-gong-gong-jie-dian-lcof/solution/shuang-zhi-zhen-fa-lang-man-xiang-yu-by-ml-zimingm/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

其实很简单的原理，我当时思维被绕了一下。

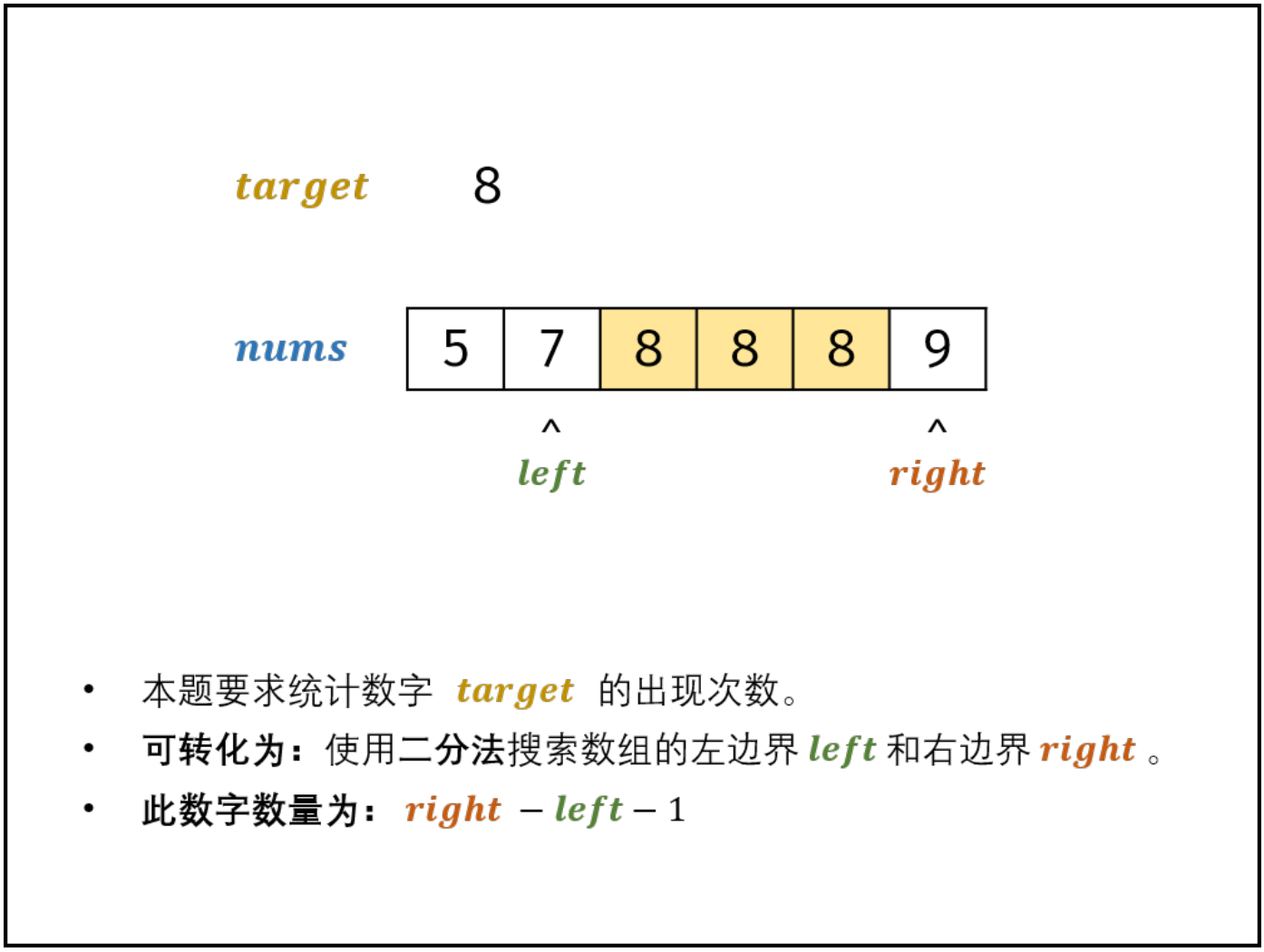
面试中的各种能力

在排序数组中查找数字-I：二分法降低复杂度

这里有一个隐含条件：**排序数组**，那相同数字必然是连续的。并且，排序数组中的搜索问题，首先想到 **二分法** 解决。

排序数组 $nums$ 中的所有数字$target$ 形成一个窗口，记窗口的 左 / 右边界 索引分别为 $left$ 和 $right$ ，分别对应窗口左边 / 右边的首个元素。

本题要求统计数字 $target$ 的出现次数，可转化为：使用二分法分别找到 左边界 $left$ 和 右边界 $right$ ，易得数字 $target$ 的数量为 $right - left - 1$ 。



1. **初始化**： 左边界 $i = 0$ ，右边界 $j = len(nums) - 1$ 。
2. **循环二分**： 当闭区间 $[i, j]$ 无元素时跳出；
    1. 计算中点 $m = (i + j)/2$ （向下取整）；
    2. 若 $nums[m] < target$ ，则 $target$ 在闭区间 $[m + 1, j]$ 中，因此执行 $i = m + 1$；
    3. 若 $nums[m] > target$ ，则 $target$ 在闭区间 $[i, m - 1]$ 中，因此执行 $j = m - 1$；
    4. 若 $nums[m] = target$ ，则右边界 $right$ 在闭区间 $[m + 1, j]$ 中；左边界 $left$ 在闭区间 $[i, m - 1]$ 中。因此分为以下两种情况：
        1. 若查找 **右边界** $right$ ，则执行 $i = m + 1$ ； （跳出时 i 指向右边界）
        2. 若查找 **左边界** $left$ ，则执行 $j = m - 1$ ； （跳出时 j 指向左边界）

3. **返回值**：应用两次二分，分别查找 $right$ 和 $left$，最终返回 $right - left - 1$ 即可。

这里有一个优化，基于：**查找完右边界 $right = i$ 后，则 $nums[j]$ 指向最右边的 $target$（若存在）。**

1. 查找完右边界后，可用 $nums[j] = j$ 判断数组中是否包含 $target$，若不包含则直接提前返回 0，无需后续查找左边界。
2. 查找完右边界后，左边界 $left$ 一定在闭区间 $[0, j]$ 中，因此直接从此区间开始二分查找即可。

```python
class Solution:
    def search(self, nums: [int], target: int) -> int:
        # 搜索右边界 right
        i, j = 0, len(nums) - 1
        while i <= j:
            m = (i + j) // 2
            if nums[m] <= target: i = m + 1
            else: j = m - 1
        right = i
        # 若数组中无 target ，则提前返回
        if j >= 0 and nums[j] != target: return 0
        # 搜索左边界 left
        i = 0
        while i <= j:
            m = (i + j) // 2
            if nums[m] < target: i = m + 1
            else: j = m - 1
        left = j
        return right - left - 1

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof/solution/mian-shi-ti-53-i-zai-pai-xu-shu-zu-zhong-cha-zha-5/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

或者将两次二分法查找函数封装一下：

```python
class Solution:
    def search(self, nums: [int], target: int) -> int:
        def helper(tar):
            i, j = 0, len(nums) - 1
            while i <= j:
                m = (i + j) // 2
                if nums[m] <= tar: i = m + 1
                else: j = m - 1
            return i
        return helper(target) - helper(target - 1)

# 作者：jyd
```

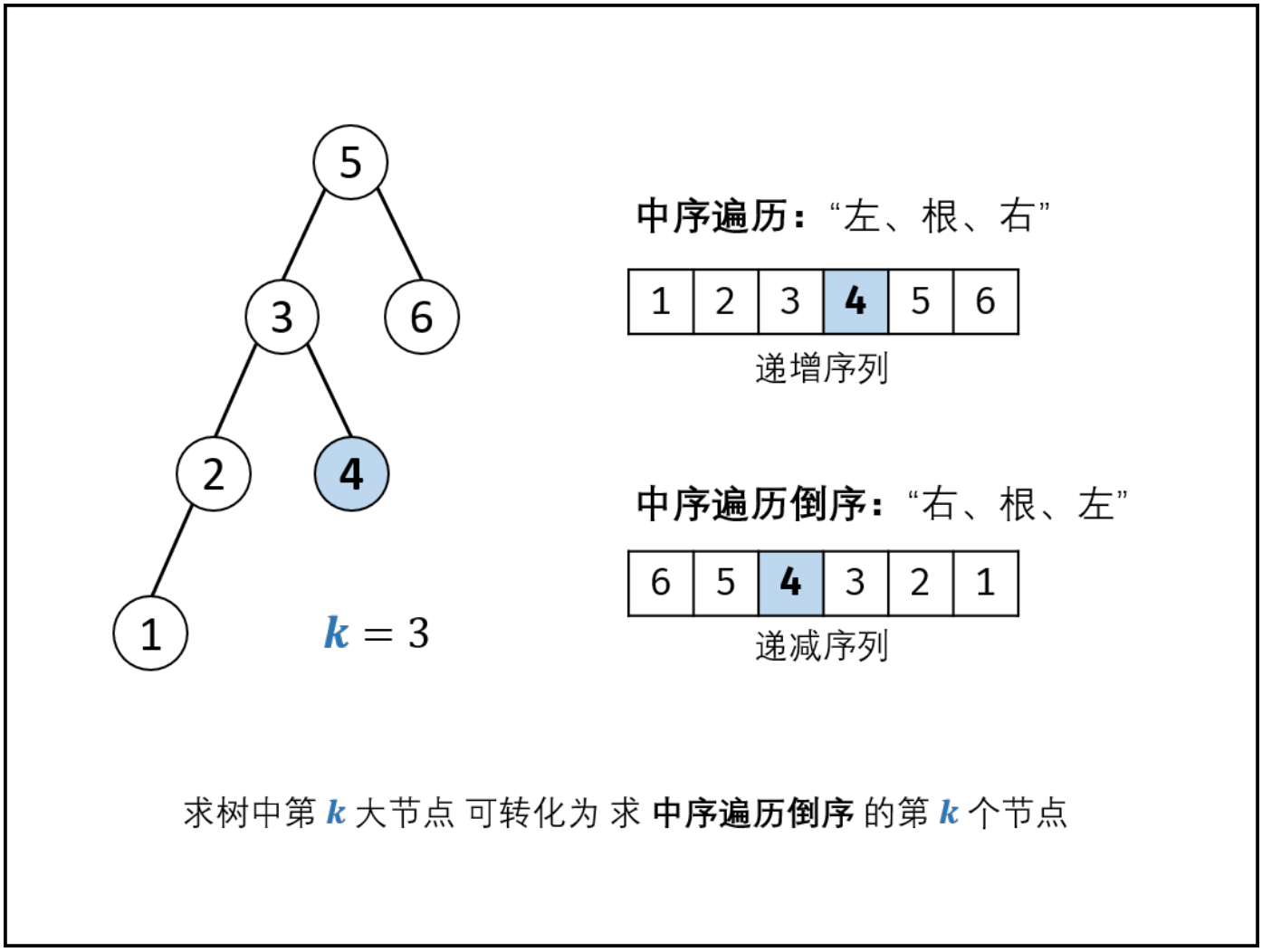
```
# 链接：https://leetcode-cn.com/problems/zai-pai-xu-shu-zu-zhong-cha-zhao-shu-zi-lcof/solution/mian-shi-ti-53-i-zai-pai-xu-shu-zu-zhong-cha-zha-5/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

0～n-1中缺失的数字：这个题限定很多：**递增排序数组，所有数字都唯一，都在范围0～n-1之内。在范围0～n-1内的n个数字中有且只有一个数字不在该数组中。**所以解答就很简单（还是用二分法，否则直接遍历会超时）：

```
def missingNumber(self, nums: List[int]) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            m = (l + r) // 2
            if nums[m] == m: l = m + 1
            else: r = m - 1
        return l
```

二叉搜索树的第k大节点：中序遍历+提前返回

二叉搜索树的 **中序遍历倒序** 为 **递减序列** ，因此，求第k大节点可以看成**中序遍历倒序**的第k个节点。



1. 递归遍历时计数，统计当前节点的序号；
2. 递归到第 k 个节点时，应记录结果 res ；
3. 记录结果后，后续的遍历即失去意义，应提前终止（即返回）。

```python
class Solution:
    def kthLargest(self, root: TreeNode, k: int) -> int:
        def dfs(root):
            if not root: return
            dfs(root.right)
            if self.k == 0: return
            self.k -= 1
            if self.k == 0: self.res = root.val
            dfs(root.left)

        self.k = k
        dfs(root)
        return self.res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-sou-suo-shu-de-di-kda-jie-dian-lcof/solution/mian-shi-ti-54-er-cha-sou-suo-shu-de-di-k-da-jie-d/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

二叉树的深度：递归

一棵树有一个节点：则深度为1，如只有其中一个子树，则深度就是这个子树深度+1，**若两个子树都有，则深度为两个子树深度较大值+1**

```python
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if not root: return 0
        nleft = self.maxDepth(root.left)
        nRight = self.maxDepth(root.right)

        return nleft + 1 if nleft > nRight else nRight + 1
```

平衡二叉树：（追求每个节点只遍历一次的解法）

后序遍历 + 剪枝 （从底至顶）：思路是对二叉树做后序遍历，从底至顶返回子树深度，若判定某子树不是平衡树则 “剪枝”，直接向上返回。

`recur(root)` 函数：

- 返回值：
  1. 当节点`root` 左 / 右子树的深度差 ≤ 1：则返回当前子树的深度，即节点 `root` 的左 / 右子树的深度最大值 $+1$ （`max(left, right) + 1`）；
  2. 当节点`root` 左 / 右子树的深度差 > 2：则返回 -1，代表 **此子树不是平衡树** 。
- 终止条件：
  1. 当 `root` 为空：说明越过叶节点，因此返回高度 0；
  2. 当左（右）子树深度为 -1：代表此树的 **左（右）子树** 不是平衡树，因此剪枝，直接返回 -1；

`isBalanced(root)` 函数：

- 返回值：若 `recur(root) != -1`，则说明此树平衡，返回 $true$；否则返回 $false$。

```
class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        def recur(root):
            if not root: return 0
            left = recur(root.left)
            if left == -1: return -1
            right = recur(root.right)
            if right == -1: return -1
            return max(left, right) + 1 if abs(left - right) <= 1 else -1

        return recur(root) != -1

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/ping-heng-er-cha-shu-lcof/solution/mian-shi-ti-55-ii-ping-heng-er-cha-shu-cong-di-zhi/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

数组中数字出现的次数 I：异或运算

一开始我用遍历统计哈希表，出现新的数放进，再出现再删除：

```
class Solution:
    def myFunction(self, nums: List[int]) -> List[int]:
        dic, i = dict(), 0
        while(i < len(nums)):
            if nums[i] not in dic.keys():
                dic[nums[i]] = True
            else:
                del dic[nums[i]]
            i += 1
        return list(dic.keys())
```

结果超空间了（题目要求空间复杂度是O(1)，哈希表统计法是不能用的）

异或运算有个重要的性质，两个相同数字异或为 0，即对于任意整数 a 有 $a \oplus a = 0$。因此，若将 $nums$ 中所有数字执行异或运算，留下的结果则为 出现一次的数字 x。

```
class Solution:
    def singleNumbers(self, nums: List[int]) -> List[int]:
        x, y, n, m = 0, 0, 0, 1
        for num in nums:          # 1. 遍历异或
            n ^= num
        while n & m == 0:         # 2. 循环左移，计算 m
            m <<= 1
        for num in nums:          # 3. 遍历 nums 分组
            if num & m: x ^= num # 4. 当 num & m != 0
```

```
            else: y ^= num        # 4. 当 num & m == 0
        return x, y               # 5. 返回出现一次的数字

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/shu-zu-zhong-shu-zi-chu-xian-de-ci-shu-lcof/solution/jian-zhi-offer-56-i-shu-zu-zhong-shu-zi-tykom/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

数组中数字出现的次数 II:

这时候不限制空间复杂度了，所以我照搬了之前的哈希表解法，改改，就可以了。（就是`return`不太优雅）

```
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        dic, i = dict(), 0
        while(i < len(nums)):
            if nums[i] not in dic.keys():
                dic[nums[i]] = 1
            elif dic[nums[i]] == 1:
                dic[nums[i]] = 2
            elif dic[nums[i]] == 2:
                dic[nums[i]] = 3
            i += 1
        return list(dic.keys())[list(dic.values()).index(1)]
```

和为 s 的两个数字:

一开始我利用排序树组，采用了双指针，将数组分成 $0\sim(\text{恰好大于}target//2\text{的指针})-1$，$(\text{恰好大于}target//2\text{的指针})\sim(\text{恰好大于}target\text{的指针})-1$，然后遍历一个子树组寻找差在不在另一个子树组。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        m = 0
        while nums[m] < (target // 2):
            m += 1
        print(m+1)
        n = m
        while nums[n] < target and n < len(nums) - 1:
            n += 1
        print(n+1)
        for i in range(m+1):
            print(nums[i], target - nums[i], nums[m:n+1])
            if (target - nums[i]) in nums[m:n+1]:
                return [nums[i], target - nums[i]]
        return None
```

然后超内存了。。

K神的思路是：用对撞双指针

1. 初始化： 双指针 i , j 分别指向数组 $nums$ 的左右两端 （俗称对撞双指针）。
2. 循环搜索： 当双指针相遇时跳出；
    1. 计算和 $s = nums[i] + nums[j]$ ；
    2. 若 $s > target$ ，则指针 j 向左移动，即执行 $j = j - 1$ ；
    3. 若 $s < target$ ，则指针 i 向右移动，即执行 $i = i + 1$ ；
    4. 若 $s = target$ ，立即返回数组 $[nums[i], nums[j]]$ ；
3. 返回空数组，代表无和为 $target$ 的数字组合。

可证明完备性

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        i, j = 0, len(nums) - 1
        while i < j:
            s = nums[i] + nums[j]
            if s > target: j -= 1
            elif s < target: i += 1
            else: return nums[i], nums[j]
        return []

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/he-wei-sde-liang-ge-shu-zi-lcof/solution/mian-shi-ti-57-he-wei-s-de-liang-ge-shu-zi-shuang-/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

我又给整复杂了。。

和为s的连续正数序列:滑动指针

```
class Solution:
    def findContinuousSequence(self, target: int) -> List[List[int]]:
        l, r, s, res = 1, 1, 0, []
        while l <= target // 2:
            if s < target:
                # 右边界向右移动
                s += r
                r += 1
            elif s > target:
                # 左边界向右移动
                s -= l
                l += 1
            else:
                # 记录结果
                res.append(list(range(l, r)))
                # 左边界向右移动
                s -= l
```

```
                l += 1
        return res
```

翻转单词顺序：（这个没用数据结构的特性，直接用的库函数）

```
class Solution:
    def reverseWords(self, s: str) -> str:
        lst = s.split(' ')
        while '' in lst:
            lst.remove('')
        lst_rev = lst[::-1]
        return ' '.join(lst_rev)
```

左旋转字符串：队列

```
class Solution:
    def reverseLeftWords(self, s: str, n: int) -> str:
        lst = collections.deque(s)
        for _ in range(n):
            tmp = lst[0]
            lst.popleft()
            lst.append(tmp)
        return ''.join(lst)
```

滑动窗口的最大值:

```
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        if not nums: return []
        i = 0
        res = []
        for _ in range(i, len(nums) - k + 1):
            res.append(max(nums[i:i+k]))
            i += 1
        return res
```

队列的最大值:

本算法基于问题的一个重要性质：当一个元素进入队列的时候，它前面所有比它小的元素就不会再对答案产生影响。

举个例子，如果我们向队列中插入数字序列 `1 1 1 1 2`，那么在第一个数字 2 被插入后，数字 2 前面的所有数字 1 将不会对结果产生影响。因为按照队列的取出顺序，数字 2 只能在所有的数字 1 被取出之后才能被取出，因此如果数字 1 如果在队列中，那么数字 2 必然也在队列中，使得数字 1 对结果没有影响。

按照上面的思路，我们可以设计这样的方法：从队列尾部插入元素时，我们可以提前取出队列中所有比这个元素小的元素，使得队列中只保留对结果有影响的数字。这样的方法等价于要求维持队列单调递减，即要保证每个元素的前面都没有比它小的元素。

那么如何高效实现一个始终递减的队列呢？我们只需要在插入每一个元素 `value` 时，从队列尾部依次取出比当前元素 `value` 小的元素，直到遇到一个比当前元素大的元素 `value'` 即可。

- 上面的过程保证了只要在元素 `value` 被插入之前队列递减，那么在 `value` 被插入之后队列依然递减。而队列的初始状态（空队列）符合单调递减的定义。
- 由数学归纳法可知队列将会始终保持单调递减。
- 上面的过程需要从队列尾部取出元素，因此需要使用双端队列来实现。另外我们也需要一个辅助队列来记录所有被插入的值，以确定 `pop_front` 函数的返回值。

保证了队列单调递减后，求最大值时只需要直接取双端队列中的第一项即可。

```
import queue
class MaxQueue:

    def __init__(self):
        self.deque = queue.deque()
        self.queue = queue.Queue()

    def max_value(self) -> int:
        return self.deque[0] if self.deque else -1


    def push_back(self, value: int) -> None:
        while self.deque and self.deque[-1] < value:
            self.deque.pop()
        self.deque.append(value)
        self.queue.put(value)

    def pop_front(self) -> int:
        if not self.deque:
            return -1
        ans = self.queue.get()
        if ans == self.deque[0]:
            self.deque.popleft()
        return ans

# 作者：LeetCode-Solution
# 链接：https://leetcode-cn.com/problems/dui-lie-de-zui-da-zhi-lcof/solution/mian-shi-ti-59-ii-dui-lie-de-zui-da-zhi-by-leetcod/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

n个骰子的点数：动态规划

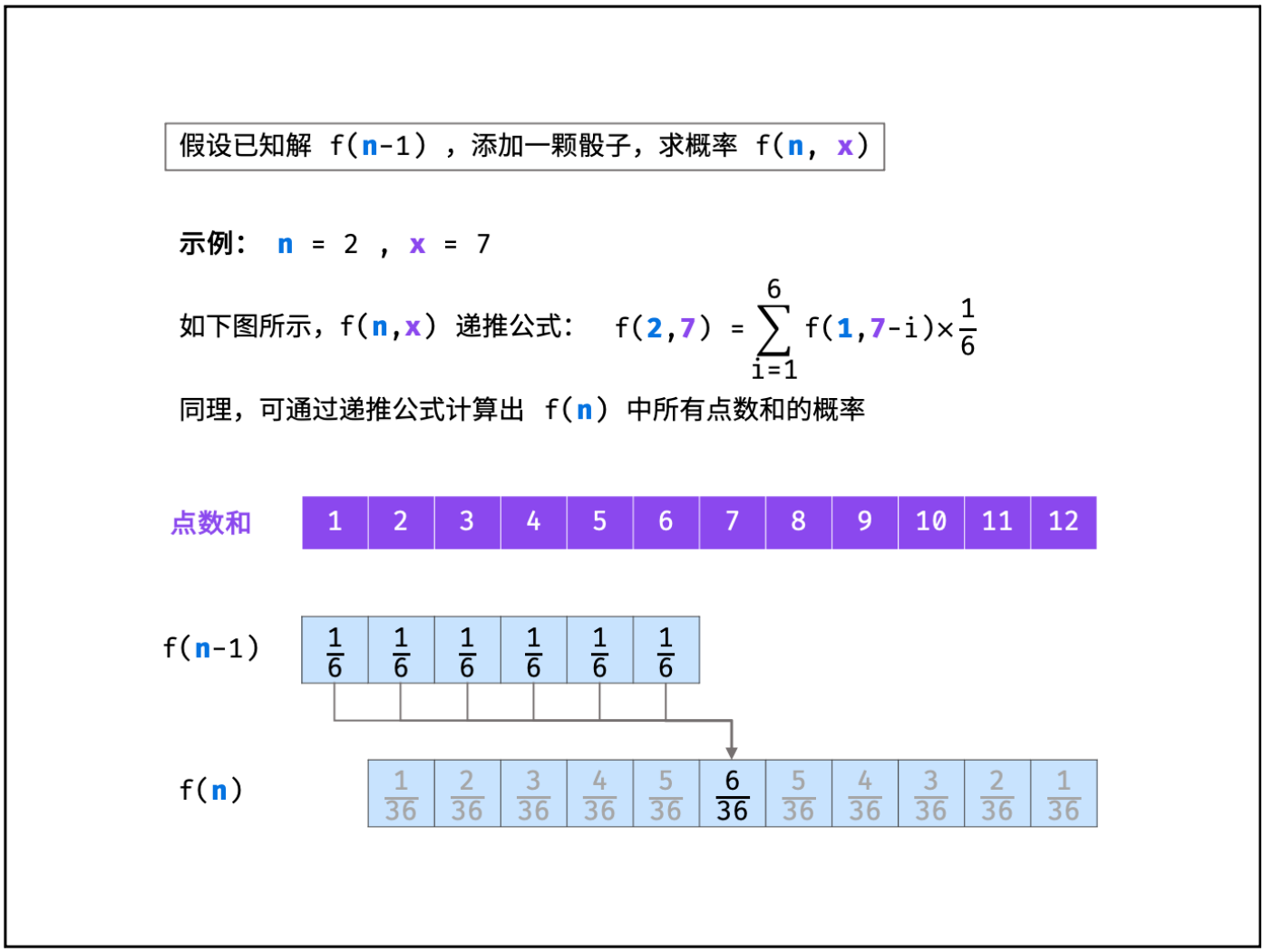
投掷 n 个骰子，所有点数出现的总次数是 6^n ，因为一共有 n 枚骰子，每枚骰子的点数都有 6 种可能出现的情况。

单纯使用递归搜索解空间的时间复杂度为 6^n ，会造成超时错误，因为存在重复子结构。

于是我们使用动态规划：

状态表示：$dp[i][j]$ ，表示投掷完 i 枚骰子后，点数 j 的出现次数。

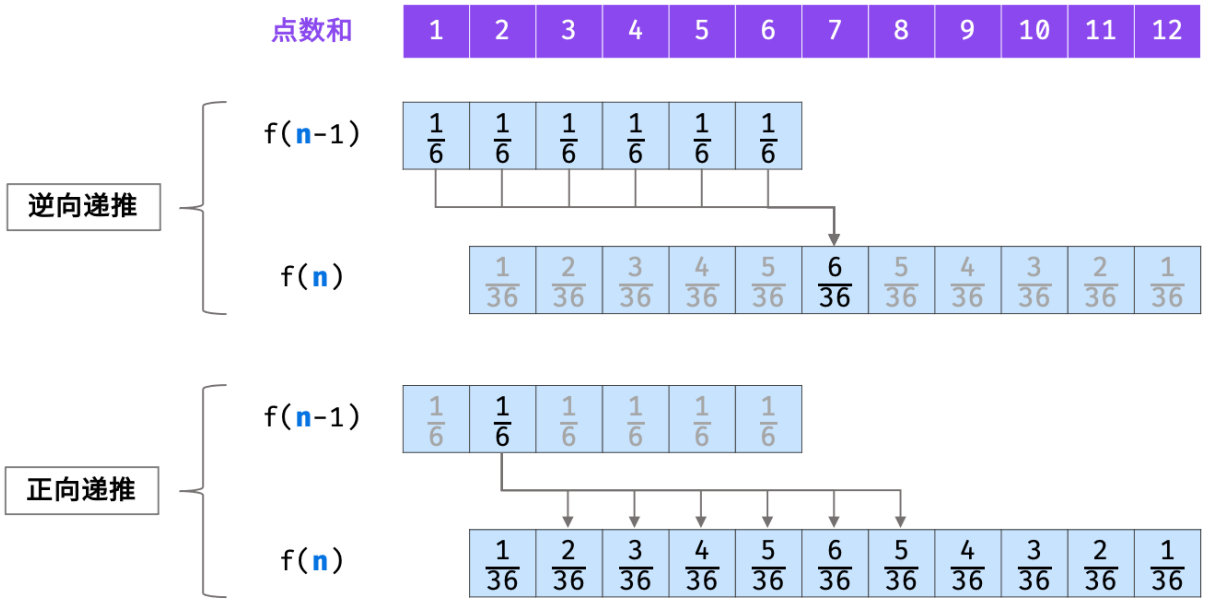
转移方程：$dp[n][x] = \sum_{i=1}^{6} dp[n-1][x-i] \times \frac{1}{6}$



边界处理：

以上递推公式虽然可行，但 $dp[n-1][x-i]$ 中的 $x-i$ 会有越界问题。例如，若希望递推计算 $dp[2][2]$，由于一个骰子的点数和范围为 $[1,6]$ ，因此只应求和 $dp[1][1]$ ，即 $dp[1][0]$, $dp[1][-1]$,...,$dp[1][-4]$,皆无意义。此越界问题导致代码编写的难度提升。

以上递推公式是 “逆向” 的，即为了计算 $dp[n][x]$，将所有与之有关的情况求和；而倘若改换为 “正向” 的递推公式，便可解决越界问题。



逆向递推（有越界问题）：为求概率 f(n,x)，将 f(n-1) 中所有与其有关的概率项求和

正向递推（无越界问题）：遍历 f(n-1)，统计每项 f(n-1,i) 对概率 f(n,i+1), f(n,i+2), …, f(n,i+6) 产生的贡献

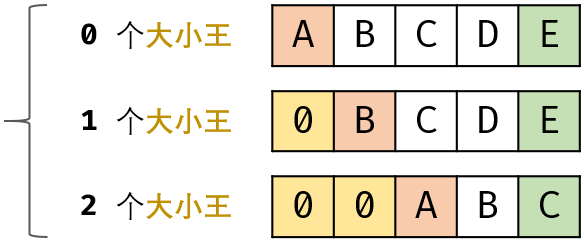
```python
class Solution:
    def dicesProbability(self, n: int) -> List[float]:
        dp = [1 / 6] * 6
        for i in range(2, n + 1):
            tmp = [0] * (5 * i + 1)
            for j in range(len(dp)):
                for k in range(6):
                    tmp[j + k] += dp[j] / 6
            dp = tmp
        return dp

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/nge-tou-zi-de-dian-shu-lcof/solution/jian-zhi-offer-60-n-ge-tou-zi-de-dian-sh-z36d/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

扑克牌中的顺子:

题中忘说了一个条件：**抽出来的牌是需要排好序的**。根据题意，此 5 张牌是顺子的 **充分条件** 如下：

1. 除大小王外，所有牌 **无重复** ；
2. 设此 55 张牌中最大的牌为 maxmax ，最小的牌为 minmin （大小王除外），则需满足：$max - min < 5$

易证**顺子**中**无重复的牌**，因此假设无重复的牌（大小王除外），
并将 5 张牌**排序**，则根据 大小王 的数量，可分为以下三种情况：



观察发现

由于 大小王 可以替代 **任何牌** ，因此三种情况的判断条件一致：

当 **最大牌** − **最小牌** < **5** ，则 **可** 构成顺子。
当 **最大牌** − **最小牌** ≥ **5** ，则 **不可** 构成顺子。

可推出充分条件

若同时满足条件：
1. 无 **重复的牌** *（大小王除外）*；
2. **最大牌** − **最小牌** < **5** *（大小王除外）*；
则此 5 张牌 可构成顺子。

1. 先对数组执行排序。
2. **判别重复**： 排序数组中的相同元素位置相邻，因此可通过遍历数组，判断 $nums[i] = nums[i+1]$ 是否成立来判重。
3. **获取最大 / 最小的牌**： 排序后，数组末位元素 $nums[4]$ 为最大牌；元素 $nums[joker]$ 为最小牌，其中 $joker$ 为大小王的数量。

```
class Solution:
    def isStraight(self, nums: List[int]) -> bool:
        joker = 0
        nums.sort() # 数组排序
        for i in range(4):
            if nums[i] == 0: joker += 1 # 统计大小王数量
            elif nums[i] == nums[i + 1]: return False # 若有重复，提前返回 false
        return nums[4] - nums[joker] < 5 # 最大牌 - 最小牌 < 5 则可构成顺子
```

圆圈中最后剩下的数字：用列表，注意索引要及时取余，不要越界。

```python
class Solution:
    def lastRemaining(self, n: int, m: int) -> int:
        circle_list = list(range(n))
        del_point = (m - 1) % len(circle_list)
        while len(circle_list) > 1:
            # print('del_point:', del_point)
            del circle_list[del_point]
            # print('circle_list:', circle_list)
            del_point = (del_point + (m - 1)) % len(circle_list)
        return circle_list[0]
```

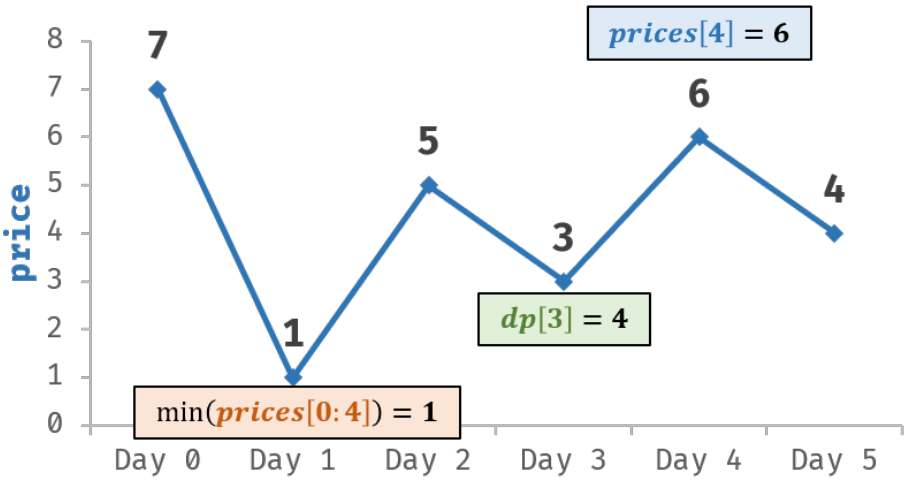
股票的最大利润：动态规划

- **状态定义**： 设动态规划列表 dp ， $dp[i]$ 代表以 $prices[i]$ 为结尾的子数组的最大利润（以下简称为 前 i 日的最大利润 ）。
- **转移方程**： 由于题目限定 “买卖该股票一次”， 因此前 i 日最大利润 $dp[i]$ 等于前 $i-1$ 日最大利润 $dp[i-1]$ 和第 i 日卖出的最大利润中的最大值。

  前 i 日最大利润 = max(前 $(i-1)$ 日最大利润, 第 i 日价格 - 前 i 日最低价格)

  $$dp[i] = \max(dp[i-1], prices[i] - \min(prices[0:i]))$$

- **初始状态**： $dp[0] = 0$ ， 即首日利润为 0 ；
- **返回值**： $dp[n-1]$ ， 其中 n 为 dp 列表长度。



例如前 4 日的最大利润为：

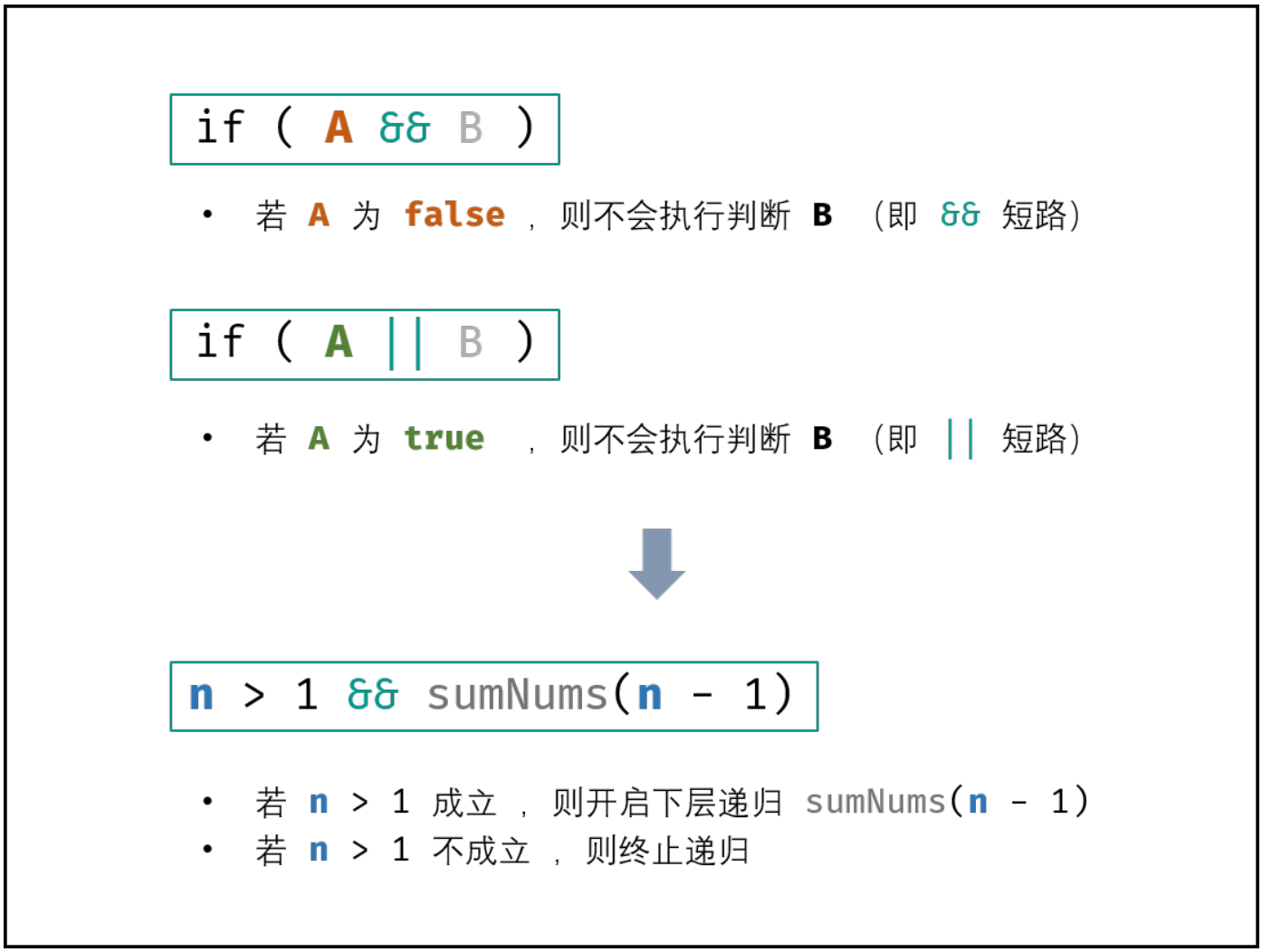
$$dp[4] = \max(dp[3], prices[4] - \min(prices[0:4]))$$
$$= \max(4, 6 - 1)$$
$$= 5$$

转移方程：

$$dp[i] = \max(\, dp[i-1], prices[i] - \min(prices[0:i])\,)$$

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        cost, profit = float("+inf"), 0
        for price in prices:
            cost = min(cost, price)
            profit = max(profit, price - cost)
        return profit

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/gu-piao-de-zui-da-li-run-lcof/solution/mian-shi-ti-63-gu-piao-de-zui-da-li-run-dong-tai-2/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

求 1 + 2 + ... + n:要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句（A?B:C）。



常见的逻辑运算符有三种，即“与 &&”，“或 ||”，“非 !”；而其有重要的短路效应，如下所示：

```
if(A && B)  // 若 A 为 false ，则 B 的判断不会执行（即短路），直接判定 A && B 为 false
if(A || B) // 若 A 为 true ，则 B 的判断不会执行（即短路），直接判定 A || B 为 true
```

本题需要实现“当 $n = 1$ 时终止递归”的需求，可通过短路效应实现。

```
n > 1 && sumNums(n - 1) // 当 n = 1 时 n > 1 不成立 ，此时 “短路” ，终止后续递归
```

```python
class Solution:
    def __init__(self):
        self.res = 0
    def sumNums(self, n: int) -> int:
        n > 1 and self.sumNums(n - 1)
        self.res += n
        return self.res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/qiu-12n-lcof/solution/mian-shi-ti-64-qiu-1-2-nluo-ji-fu-duan-lu-qing-xi-/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

不用加减乘除做加法:

本题考查位运算，我按照C++范例，写了一个Python的代码，结果输入负数样例超时了。

```python
class Solution:
    def add(self, a: int, b: int) -> int:
        sum, carry  = 0, 0
        while b != 0:
            sum = a^b
            carry = (a&b)<<1
            a = sum
            b = carry
        return a
```

Python 没有 `int` , `long` 等不同长度变量，即在编程时无变量位数的概念。

获取负数的补码： 需要将数字与十六进制数 `0xffffffff` 相与。可理解为舍去此数字 32 位以上的数字（将 32 位以上都变为 0 ），从无限长度变为一个 32 位整数。

返回前数字还原： 若补码 a 为负数（ `0x7fffffff` 是最大的正数的补码 ），需执行 `~(a ^ x)` 操作，将补码还原至 Python 的存储格式。 `a ^ x` 运算将 1 至 32 位按位取反； `~` 运算是将整个数字取反；因此， `~(a ^ x)` 是将 32 位以上的位取反，1 至 32 位不变。

```python
class Solution:
    def add(self, a: int, b: int) -> int:
        x = 0xffffffff
        a, b = a & x, b & x
        while b != 0:
            sum = a^b
            carry = (a&b)<<1
            a = sum
            b = carry & x
        return a if a <= 0x7fffffff else ~(a ^ x)
```

构建乘积数组：不能使用除法，于是用上下三角形

$$\boldsymbol{B}[i] = A[0] \times A[1] \times \cdots \times A[i-1] \times A[i-1] \times \cdots \times A[n-1] \times A[n]$$

列表格

$\boldsymbol{B}[0] =$	1	$A[1]$	$A[2]$	$\cdots$	$A[n-1]$	$A[n]$
$\boldsymbol{B}[1] =$	$A[0]$	1	$A[2]$	$\cdots$	$A[n-1]$	$A[n]$
$\boldsymbol{B}[2] =$	$A[0]$	$A[1]$	1	$\cdots$	$A[n-1]$	$A[n]$
$\ldots =$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$
$\boldsymbol{B}[N-1] =$	$A[0]$	$A[1]$	$A[2]$	$\cdots$	1	$A[n]$
$\boldsymbol{B}[N] =$	$A[0]$	$A[1]$	$A[2]$	$\cdots$	$A[n-1]$	1

解法

通过两轮循环，分别计算 下三角 和 上三角 的乘积，
即可在不使用除法的前提下获得结果。

```python
class Solution:
    def constructArr(self, a: List[int]) -> List[int]:
        if not a: return []
        elif len(a) == 1: return a
        elif len(a) == 2: return a[::-1]
        else:
            res_left, res_right, res = [1, a[0]], [1, a[-1]], []
            for i in range(1, len(a) - 1):
                res_left.append(res_left[-1] * a[i])
                res_right.append(res_right[-1] * a[len(a) - i - 1])
            for i in range(len(a)):
                res.append(res_left[i] * res_right[-1-i])
        return res
```

把字符串转换成整数:

本题可以用**自动机**解决，我试过但是仍旧处理不了一些特殊情况，程序就很臃肿了（其实整理思路是可以简短解决的），下面是K神的思路：

```python
class Solution:
    def strToInt(self, str: str) -> int:
        str = str.strip()                      # 删除首尾空格
        if not str: return 0                   # 字符串为空则直接返回
        res, i, sign = 0, 1, 1
        int_max, int_min, bndry = 2 ** 31 - 1, -2 ** 31, 2 ** 31 // 10
        if str[0] == '-': sign = -1            # 保存负号
        elif str[0] != '+': i = 0              # 若无符号位，则需从 i = 0 开始数字拼接
        for c in str[i:]:
            if not '0' <= c <= '9' : break     # 遇到非数字的字符则跳出
            if res > bndry or res == bndry and c > '7': return int_max if sign == 1
else int_min # 数字越界处理
            res = 10 * res + ord(c) - ord('0') # 数字拼接
        return sign * res

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/ba-zi-fu-chuan-zhuan-huan-cheng-zheng-shu-
lcof/solution/mian-shi-ti-67-ba-zi-fu-chuan-zhuan-huan-cheng-z-4/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

二叉搜索树的最近公共祖先:

因为是二叉搜索树，所以可以直接判断节点值来确定位置。注意测试用例有$p > q$的情况，这很扯。

```
class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') ->
'TreeNode':
        while root:
            if (p.val <= root.val and root.val <= q.val) or (p.val >= root.val and
root.val >= q.val): return root
            if p.val <= root.val and q.val <= root.val: return
self.lowestCommonAncestor(root.left,p,q)
            elif p.val >= root.val and q.val >= root.val: return
self.lowestCommonAncestor(root.right,p,q)
```

二叉树的最近公共祖先:

此时不是二叉搜索树了，需要用DFS，回溯法和后续遍历。这里我看了K神的解法：

```
class Solution:
    def lowestCommonAncestor(self, root: TreeNode, p: TreeNode, q: TreeNode) ->
TreeNode:
        if not root or root == p or root == q: return root
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)
        if not left: return right
        if not right: return left
        return root

# 作者：jyd
# 链接：https://leetcode-cn.com/problems/er-cha-shu-de-zui-jin-gong-gong-zu-xian-
lcof/solution/mian-shi-ti-68-ii-er-cha-shu-de-zui-jin-gong-gon-7/
# 来源：力扣（LeetCode）
# 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

-END-