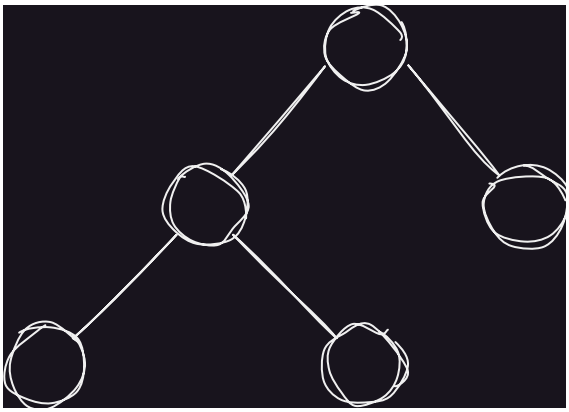
目录

二叉树经典问题

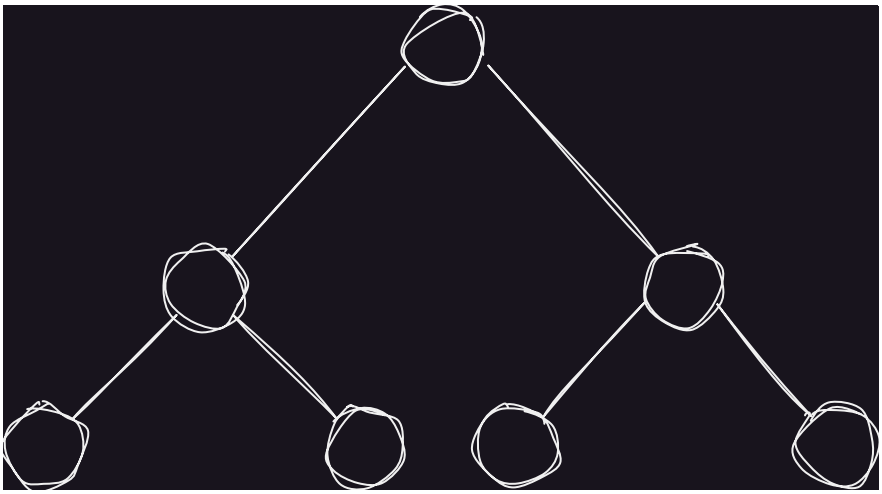
1. 什么是满二叉树、完全二叉树、完美二叉树、二叉搜索树和平衡二叉树?

下面逐个解释这些概念。

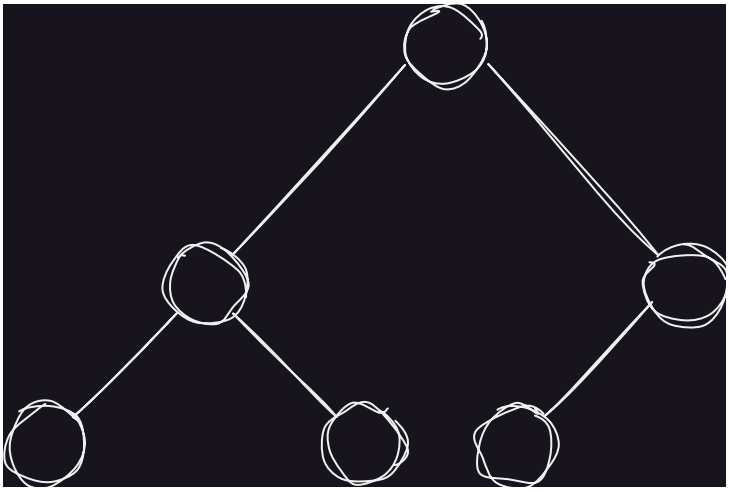
满二叉树（Full Binary Tree）是一种特殊的二叉树结构，**其中每个节点要么是叶子节点（没有子节点），要么有两个子节点。** 这意味着每一层上的节点都是完全填满的。



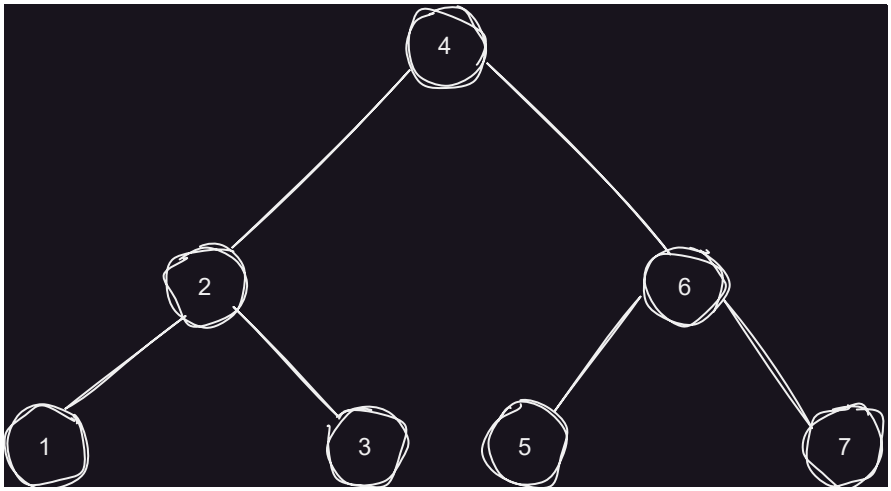
完美二叉树（Perfect Binary Tree）是一种特殊的二叉树结构，**其中每个节点要么是叶子节点（没有子节点），要么有两个子节点，并且左右子树都完全相同。** 这意味着每一层上的节点都是完全填满的。和满二叉树最大的不同在于，完美二叉树不是节点粒度上的填满，而是层粒度上的填满。



完全二叉树（Complete Binary Tree）是一种特殊的二叉树结构，**其中除了最后一层节点，其他层节点都是满的，并且最后一层节点从左向右依次排布。** 这是一种完美二叉树的弱化版本，因为完美二叉树尽管拥有很多很好的性质，但是最后一层的节点数量会指数上升。而完全二叉树是一种可以动态决定最后一层节点数量的二叉树结构，同时保证了完美二叉树的一些良好性质（平衡性），主要用于实现堆数据结构。



二叉搜索树是一种有序的二叉树，其中每个节点的值都满足一定的排序规则：**对于任意一个节点，其左子树上所有节点的值都小于该节点的值，而右子树上所有节点的值都大于该节点的值。** 二叉搜索树是一种常见的数据结构，其中每一个子树也是二叉搜索树，利用这个性质可以递归构造二叉搜索树。



平衡二叉树是一种特殊的二叉搜索树，它保持二叉树的高度尽可能小，从而确保查找、插入、删除操作的时间复杂度尽量接近 $O(\log n)$ 。平衡二叉树需要定义**平衡**规则，而由于二叉树的操作时间复杂度和树的高度直接相关，所以平衡二叉树的规则基本都是限制住树的高度。最常见的两种平衡二叉树是 AVL 树和红黑树，在 AVL 树中，任何节点的两个子树的高度差最多为 1，而红黑树更加复杂，想了解更多可以参考深入理解红黑树。

2. 给定节点总数 n，完全二叉树可能的最小高度是多少?

高度（从1开始）为h的完美二叉树有 2^h-1 个节点。那么可以得到高度为 h 的完全二叉树的节点数量 n 满足 $2^{h-1} \leq n \leq 2^h-1$ 。于是给定节点数量 n，我们试图找到第一个 h，其满足 $2^h-1 \geq n$，那么就可以得到最小的高度 $h_{min}=\lceil \log_2(n+1) \rceil$ 。

3. 给定节点总数 n，这有可能是一棵满二叉树吗?

满二叉树的节点数量 $n=2*h-1$ ，所以只需要判断 $n+1$ 是否为偶数即可。

4. 二叉搜索树的前驱后继怎么找（迭代实现）?

下面给出二叉树的寻找前驱的迭代实现。

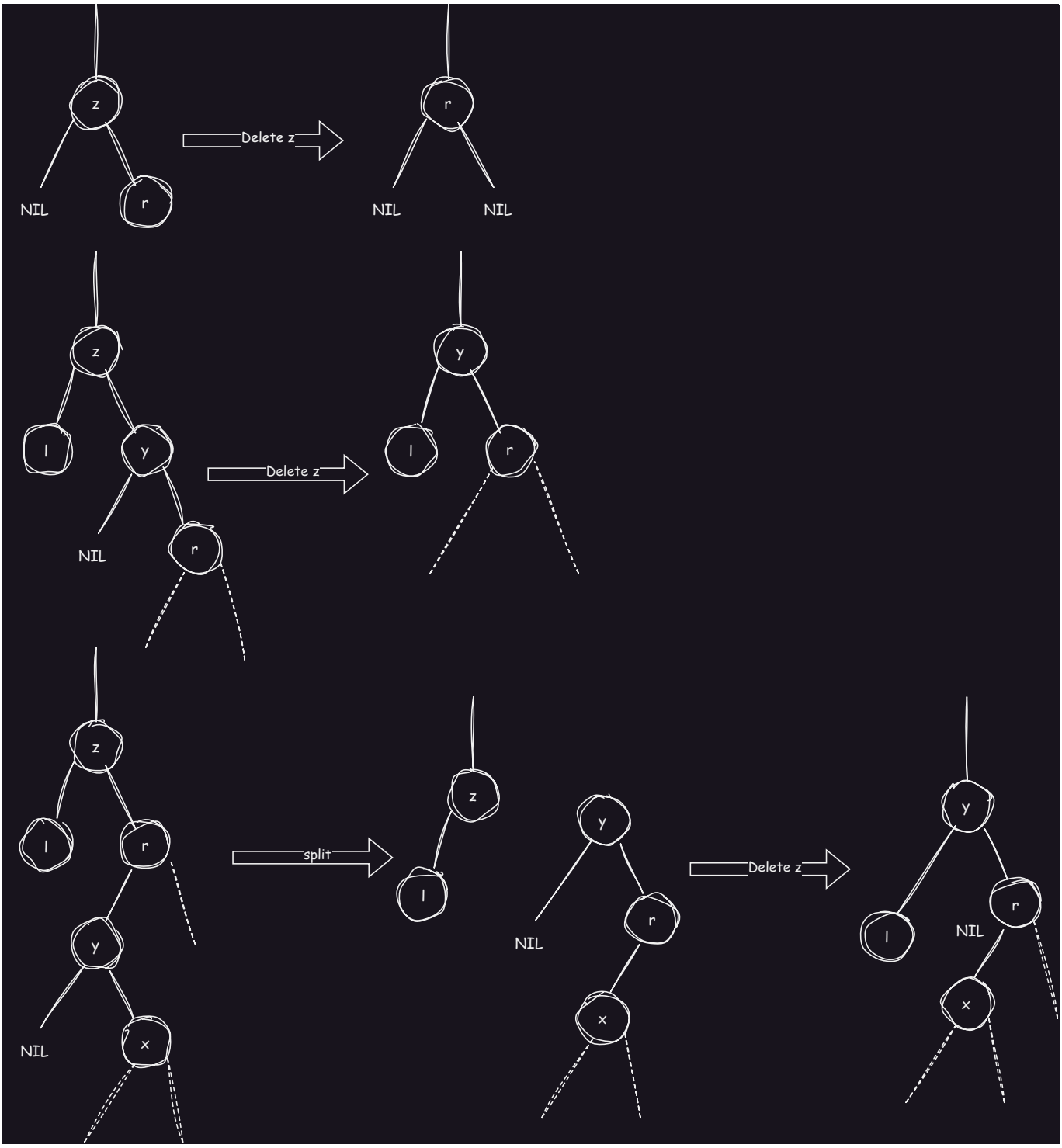
```
class TreeNode:
    def __init__(self, val=0, parent=None, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
        self.parent = parent

def find_preprocessor(root, val):
    cur = self.root
    # 找到val的位置
    while cur and cur.val != val:
        if val < cur.val:
            cur = cur.left
        else:
            cur = cur.right
    if not cur:
        return None
    if cur.left:
        # 如果存在左子树，那么前驱就是左子树的最右节点
        cur = cur.left
        while cur.right:
            cur = cur.right
        return cur
    # 否则一直往上走，直到 cur != cur.parent.left
    pa = cur.parent
    while pa and pa.left != cur:
        cur = pa
        pa = cur.parent
    return pa
```

5. 二叉搜索树的怎么删除一个节点?

二叉搜索树的删除操作没有那么简单，总共可以分为以下几种情况：

1. 删除的节点是叶子节点，直接删除即可。
2. 删除的节点只有一个子节点，那么直接将子节点替换到删除节点的位置即可。
3. 删除的节点有两个子节点，那么需要找到这个节点的右子树中的最小节点，将这个节点的值替换到删除节点的位置，然后删除这个最小节点即可。



下面是实现的参考代码：

```
def tranplant(root, u, v):
    # 移植子树 v 到 u 的位置，并且调整关系
    if u.parent is None:
        root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    if v:
        v.parent = u.parent
```

```
def delete(root, val):
    cur = root
    # 找到 cur 的位置
    while cur and cur.val != val:
        if val < cur.val:
            cur = cur.left
        else:
            cur = cur.right
    if not cur:
        return None
    if cur.left is None:
        tranplant(root, cur, cur.right)
    elif cur.right is None:
        tranplant(root, cur, cur.left)
    else:
        y = cur.right
        while y.left:
            y = y.left
        if y.parent != cur:
            tranplant(root, y, y.right)
            y.right = cur.right
            y.right.parent = y
        tranplant(root, cur, y)
        y.left = cur.left
        y.left.parent = y
```

6. 结点数量为 n 的二叉树，有多少种不同的结构?

由于不同的二叉树结构决定了递归的顺序问题（出栈和入栈），令 1 表示进栈， 0 表示出栈，则可转化为求一个 $2n$ 位、含 n 个 1、 n 个 0 的二进制数，满足从左往右扫描到任意一位时，经过的 0 数不多于 1 数。显然含 n 个 1 、 n 个 0 的 $2n$ 位二进制数共有 $\binom{2n}{n}$ 个，下面考虑不满足要求的数目。

假设其中不合法的序列在位置 $2m+1$ 处，此时恰好 0 的数量比 1 多一位，那么必然后面的有 1 的数量比 0 多一位，具体而言， 0 有 $n-m-1$ 位， 1 有 $n-m$ 位。我们将 $2m+2$ 及之后的序列进行反转，即可得到一个包含了 $n+1$ 个 0， $n-1$ 个 1 的序列。注意这是一个双射的过程，即一个不合法的序列经过构造始终得到唯一的一个包含了 $n+1$ 个 0， $n-1$ 个 1 的序列，而反过来该序列唯一对应一个不合法的序列。下面证明：

定义映射 f ：从不满足条件的 n 个 1 和 n 个 0 的序列到 $n+1$ 个 0 和 $n-1$ 个 1 的序列。 映射 f 的构造：找到第一个违反条件的位置（称为关键位置）将此位置之后的所有 0 变 1， 1 变 0 。

证明 f 是单射（一对一）：

假设两个不同的不满足条件的序列 A 和 B 映射到同一个序列

- A 和 B 的关键位置必然相同（否则映射结果会不同）
- 如果 A 和 B 在关键位置之前有任何不同，映射后仍然不同
- 如果 A 和 B 在关键位置之后有任何不同，由于0和1互换，映射后仍然不同

因此，不可能有两个不同的序列映射到同一个序列。

证明 f 是满射（映上）：

对于任何 $n+1$ 个 0 和 $n-1$ 个 1 的序列 S，从左到右扫描，必然存在一个位置， 0 的数量比 1 的数量多 2（因为总共多2个0）。这个位置就是我们寻找的关键位置， 将此位置之后的 0 和 1 互换，得到一个 n 个 1 和 n 个 0 的序列 T。 T 在关键位置之前满足条件，在关键位置不满足条件，因此 T 是一个不满足原条件的序列，且 $f(T)=S$ 。

证明 f 的逆映射：

对于任何 $n+1$ 个 0 和 $n-1$ 个 1 的序列，找到 0 比 1 多2的位置（一定存在且唯一）。将此位置之后的 0 和 1 互换，这个过程是上述映射的逆过程。

证毕。

所以合法的序列（也就是二叉树不同结构数量）等于：

$$\binom{2n}{n}-\binom{2n}{n+1}$$

也就是卡特兰数，其实卡特兰数还满足以下的性质：

$$C_0=1, C_{n+1}=\sum_{i=0}^{n} C_i C_{n-i}$$

C_i 可以看成是左子树的数量， C_{n-i} 可以看成是右子树的数量，根据乘法原理即可得到总的数量。

数组经典问题（双指针、滑动窗口、二分）

1. 二分查找如何实现？开区间写法、闭区间写法？

二分查找是在有序数组中进行高效查找的简单算法，而且可以更加广义的使用在单调（非严格、严格）的函数上，于是可以得到算法模板常用的二分答案，其中最重要的事实在于证明计算的结果是具有某种单调性质的。

不同于二分查找的思想，二分查找的实现具有大量的细节。下面主要以开闭区间为依据介绍三种常见的实现。

闭区间的写法是最为经典也是最为常见的实现，其中核心的形式为 $l \le r \le n$ ，其中 l 和 r 分别为左右边界，n 为数组长度。

我们可以定义一个函数 $f(x)$ ，如果 $f(x)$ 为真，则 x 属于右侧合法区间，否则 x 属于不合法区间。

如果区间的中点值 $mid=(l+r)/2$ 满足 $f(mid)$ ，那么更新 $r=mid-1$ ，否则更新 $l=mid+1$ 。

基于此，我们可以观察到两个循环不变量 $[0,l)$ 都是非法的， $(r,n-1]$ 都是合法的，而 $[l,r]$ 是待确定的，那么结束的时候也可以知道 l 或者 $r+1$ 是答案。

```python
# 举一个例子，我们寻找有序数组中第一个大于等于 target 的位置
# f(x) = nums[x] >= target

def lower_bound(nums, target):
    n = len(nums)
    l, r = 0, n-1
    while l <= r: # 保证区间不为空
        mid = (l+r)//2
        if nums[mid] >= target:
```

```
            r = mid-1
        else:
            l = mid+1
    return l
```

练习题目：在排序数组中查找元素的第一个和最后一个位置

那么基于上述的闭区间的写法，我们可以推导得到等价的开区间写法和左闭右开区间写法。

对于开区间而言，我们的循环不变量就是 $[0,l]$ 都是非法的， $[r,n-1]$ 都是合法的，那么其中的 (l,r) 是不确定的。

```
# 开区间写法

def lower_bound(nums, target):
    n = len(nums)
    l, r = -1, n  # 注意这里 l, r 的取值范围，l = l-1, r = r+1
    while l+1 < r:
        mid = (l+r)//2
        if nums[mid] >= target:
            r = mid
        else:
            l = mid
    return r
```

对于左闭右开区间而言，循环不变量就变成了 $[0,l)$ 是非法的， $[r,n-1]$ 都是合法的，那么其中的 $[l,r)$ 是不确定的。

```
# 左闭右开写法

def lower_bound(nums, target):
    n = len(nums)
    l, r = 0, n
    while l < r:
        mid = (l+r)//2
        if nums[mid] >= target:
            r = mid
        else:
            l = mid+1
    return l # l == r
```

2. 滑动窗口算法与二分答案算法

题目：假设有一个长度为 n 的非负整数的数组，我们需要找到其中和不大于 sum 的最长子数组的长度。

上述题目是最经典的滑动窗口算法和二分答案算法的应用。先讲其中的二分答案算法。

首先我们可以知道非负整数子数组的和总是越长和就越大，不会因为增加了长度反而和变小了。那么假设我们找到了一个长度为 L 的子数组，我们是否可以找到一个大小介于 $[0, L)$ 的子数组呢？显然可以，因为我们只需要不断减少这个子数组的长度即可。因此我们可以知道这个答案是具有单调性质的，可以使用二分的方法加速寻找答案。

```
# 使用二分答案算法

def get_longgest_subarray(nums, sum):
    n = len(nums)
    pre = [0] * (n+1)
    for i in range(1, n+1):
        pre[i] = pre[i-1] + nums[i-1]
    def check(val) -> bool:
        # 检查是否存在一个长度为val的合法子数组
        nonlocal pre
        for i in range(1, n-val+2):
            # [i, i+val-1]
            if pre[i+val-1] - pre[i-1] <= sum:
                return True
        return False

    l, r = -1, n+1 # 开区间写法
    while l+1 < r:
        mid = (l+r)//2
        if check(mid):
            l = mid
        else:
            r = mid
    return l  # 根据循环不变量 [0,l] 是合法的
```

使用了二分答案之后可以很明显感受到一点：这个最长的子数组是不是可以动态维护？比如说对于以 $nums[i]$ 结尾的子数组，我们知道了它的左边界最远为 $left$，那么对于以 $nums[i+1]$ 结尾的子数组，它的最远左边界能够是多少呢？因为刚才已经推导过了，对于长度越长的子数组，和会保持不减，所以其左边界最远就是 $left$，这可以使用反证法证明，这里就不啰嗦了。因此我们可以使用两个指针维护当前的最大子数组长度，也就是经典的双指针算法。

```
# 双指针（滑动窗口）

def get_longgest_subarray(nums, sum):
    n = len(nums)
    l, s = 0, 0
    ans = 0
    for i in range(n):
        s += nums[i]
        while s > sum:
            s -= nums[l]
            l += 1
        if i - l + 1 > ans:
```

```
                ans = i - l + 1
        return ans
```

二分答案经常用来解决一种答案具有单调性质的问题，而根据计算复杂度理论，检查是否成立比求解简单。而滑动窗口算法则是一种全局的优化，本质上是动态规划的思想。二分答案在面试算法题中是一种非常重要的算法。

回溯算法经典问题

回溯算法是一种通过穷举所有可能的解来求解问题的方法，广泛应用于各种经典的数学和计算机科学问题中。在面试中也是比较容易考察到的算法题目类型。而这种算法由于基于搜索，因此有很强的套路，需要熟练掌握。这类题目在思考的过程中可以遵循下面的思考步骤：

1. 确定搜索的空间。
2. 确定搜素的策略。
3. 确定搜素的终止条件。

枚举子集

定义：给定一个集合，求出其所有子集（或者统计其信息）。

子集问题的核心在于每个元素都有两种状态：要么被选入当前子集，要么不被选入。因此，子集问题的规模为 2^n ，即每个元素的选择组合。

练习题目1：17. 电话号码的字母组合

分析一下题目，我们对于其中的每一个数字按键都有多种选择，形式上如果我们知道了按键长度，是可以使用迭代方法获取所有方案的，但是由于我们不知道按键的长度，这个时候就需要回溯算法来实现动态的搜索。

```
choice = ["", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]

class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        n = len(digits)
        ans = []
        cur = [''] * n
        if n == 0:
            return ans
        def dfs(i):
            if i == n: # 确定好终止条件，这是很重要的，防止无限递归
                ans.append(''.join(cur))
                return
            for num in choice[int(digits[i])]:
                cur[i] = num
                dfs(i+1)
        dfs(0)
        return ans
```

练习题目2：78. 子集

对于每一个数字我们可以选或者不选，因为每一个元素都是不相同的，所以这保证了所有方案的独立性，不需要去重。

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans = []
        cur = []
        def dfs(i):
            if i == n:
                ans.append(cur.copy())
                return
            # 选择
            cur.append(nums[i])
            dfs(i+1)
            # 回溯
            cur.pop()

            # 不选择
            dfs(i+1)
        dfs(0)
        return ans
```

练习题目3：131. 分割回文串

回溯算法的思路是，对于每一个位置，我们尝试选择或者不选择，如果选择，那么判断是否为回文串，如果不是回文串，那么就直接剪枝。

```python
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        n = len(s)
        ans = []
        cur = []
        def dfs(i):
            if i == n:
                ans.append(cur.copy())
                return
            for j in range(i,n):
                t = s[i:j+1]
                if t == t[::-1]: # 判断回文串
                    cur.append(t)
                    dfs(j+1)
                    cur.pop()
        dfs(0)
        return ans
```

解决组合问题

定义：从给定的元素中选取一定数量的元素，求出所有可能的组合。

组合问题的核心在于，每次选择时后面的元素不能再被重新选择，且**每个组合的顺序不影响结果**。这是区别于排列问题的最为关键的点。

练习题目1：77. 组合

这道题目就是最为经典的组合问题，我们需要在原有的子集问题基础上加多一个选择次数的限制，而利用这些限制，我们可以在原有的简单的搜索策略加上一些启发式的规则，统称为剪枝。

```
class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        ans = []
        cur = []
        def dfs(i):
            if len(cur) == k:
                ans.append(cur.copy())
                return
            # 剪枝，[i,n] 还有 n-i+1个数字
            if n-i+1 < k-len(cur):
                return
            for j in range(i,n+1):
                cur.append(j)
                dfs(j+1)
                cur.pop()
        dfs(1)
        return ans
```

练习题目2：216. 组合总和 III

这道题目和练习题目1十分类似，但是多出了一个限制，那就是最后选择的元素和要等于一个制定的数字，那么我们就可以将这个限制放在搜索的策略中，并且制定对应的剪枝方法，从而实现高效的搜索。

```
class Solution:
    def combinationSum3(self, k: int, n: int) -> List[List[int]]:
        ans = []
        cur = []
        s = 0 # 当前和
        def dfs(i):
            nonlocal s
            # 剪枝1，[i,n]，个数限制
            if n-i+1 < k-len(cur):
                return
            # 剪枝2，i,i+1,i+2...n，数字和限制
            if (n+i)*(n-i+1)//2 < n-s or s > n:
                return
            if len(cur)==k and s == n:
                ans.append(cur.copy())
                return
            for j in range(i,10):
                cur.append(j)
                s += j
```

```
                dfs(j+1)
                s -= j
                cur.pop()
        dfs(1)
        return ans
```

解决排列问题

定义：从给定的元素中选取一定数量的元素，使得元素间满足一定的顺序，求出所有可能的排列。

排列问题和组合问题的最大区别在于，排列问题需要考虑元素间的顺序，而组合问题只需要考虑元素是否被选择。

练习题目1：46. 全排列

```
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        n = len(nums)
        ans = []
        cur = []
        def dfs(s):
            if s == 0:
                ans.append(cur.copy())
                return
            for j in range(n):
                if s & (1<<j):
                    cur.append(nums[j])
                    dfs(s ^ (1<<j))
                    cur.pop()
        dfs((1<<n)-1)
        return ans
```

这里使用了二进制集合的方法表示了当前的选择的数字以及还没有选择的数字，比如数字`nums[i]`就对应了 s 中的第 i 位，0表示已经选择了，否则就表示还没有选择。

练习题目2：51. N 皇后

此题的难点在于检查皇后是否处于同一行、同一列、同一对角线上。我们可以将对角线进行一个哈希编码，从而用于快速判断在某条对角线上是否存在皇后。

```
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        ans = []
        board = ["." * n for _ in range(n)]
        if n == 0:
            return ans
        column = [False] * n
        ldiag = [False] * (2 * n - 1)  # 左对角线
        rdiag = [False] * (2 * n - 1)  # 右对角线
```

```
        def backtrack(row):
            if row == n:
                ans.append(board.copy())
                return
            for col in range(n):
                # 计算左对角线和右对角线的索引
                l_diag_index = row - col + (n - 1)
                r_diag_index = row + col
                # 检查当前位置是否被攻击
                if column[col] or ldiag[l_diag_index] or rdiag[r_diag_index]:
                    continue
                # 放置皇后
                board[row] = board[row][:col] + 'Q' + board[row][col+1:]
                column[col] = ldiag[l_diag_index] = rdiag[r_diag_index] = True
                # 递归到下一行
                backtrack(row + 1)
                # 移除皇后（回溯）
                board[row] = board[row][:col] + '.' + board[row][col+1:]
                column[col] = ldiag[l_diag_index] = rdiag[r_diag_index] = False

        backtrack(0)
        return ans
```

动态规划经典问题

背包问题

背包问题是一类经典的可以使用动态规划解决的问题，它包含大量的变形，对于一般面试而言，只需要掌握其中的01背包和完全背包问题即可。

01背包问题和完全背包问题与以上的回溯算法思想非常类似，主要围绕着**选还是不选**做规划。对于动态规划的题目，一定要多加练习才能够掌握其中的精髓。

01背包

1. 问题描述

给定 N 件物品和一个容量为 V 的背包。第 i 件物品的费用为 C_i ，价值为 W_i 。每种物品只能选择一次，问如何选择物品使得总价值最大，且总费用不超过背包容量。

2. 动态规划求解

- **状态定义**: 设 $F[i][v]$ 表示前 i 件物品放入容量为 v 的背包时的最大价值。
- **状态转移方程**:

$$F[i][v] = \max(F[i-1][v], F[i-1][v-C_i] + W_i)$$

- **优化空间复杂度**: 可以使用一维数组 $F[v]$ ，并采用**逆序遍历**来避免覆盖问题。

3. 基础Python代码模板

```python
def knapsack_01(weights, values, V):
    n = len(weights)
    dp = [0] * (V + 1)
    for i in range(n):
        for v in range(V, weights[i] - 1, -1):
            dp[v] = max(dp[v], dp[v - weights[i]] + values[i])
    return dp[V]


weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
V = 8
max_value = knapsack_01(weights, values, V)
print(f"01背包问题的最大价值为: {max_value}")
```

完全背包问题

1. 问题描述

与01背包不同，每种物品可以选择无限次。仍给定 N 件物品和一个容量为 V 的背包，求在总费用不超过背包容量的情况下，如何选择物品使得总价值最大。

2. 动态规划求解

- **状态定义**: 设 $F[v]$ 表示容量为 v 的背包可以获得的最大价值。
- **状态转移方程**:

$$F[v] = \max(F[v], F[v - C_i] + W_i)$$

- **算法优化**: 采用顺序遍历更新 $F[v]$，以允许同一物品多次选择。

3. 基础的Python代码模板

```python
def knapsack_complete(weights, values, V):
    n = len(weights)
    dp = [0] * (V + 1)
    for i in range(n):
        for v in range(weights[i], V + 1):
            dp[v] = max(dp[v], dp[v - weights[i]] + values[i])
    return dp[V]


weights = [2, 3, 4, 5]
values = [3, 4, 5, 6]
V = 8
max_value = knapsack_complete(weights, values, V)
print(f"完全背包问题的最大价值为: {max_value}")
```

递推顺序的问题

可以发现一点，01背包和完全背包的实现代码十分类似，主要不同在于第二层循环的容量枚举顺序。在01背包中，枚举顺序是逆序的而在完全背包中则是顺序的。

为什么这样会可以呢？01背包限定了每一种物品选择的次数只能为1次，我们使用逆序的方法主要是为了首先更新大容量的dp值，保证了当前容量的dp值是仅仅依赖于前面的状态。

在完全背包中，每种物品可以被选取任意次，因此在更新 $F[v]$ 时，需要使用可能已经包含当前物品的状态。顺序遍历 v 可以确保 $F[v-C_i]$ 已经包含了当前物品 i 的选取情况。

更加具体的，如果我们按从 C_i 到 V 的顺序进行遍历，那么 $F[v]$ 的更新过程相当于在考虑"再选一次第 i 种物品"的情况。这种更新方式允许我们在已经选取若干次当前物品的基础上，再次选取，从而得到正确的解。

练习题目

1、目标和

对于每一个数字我们可以选择使用它的正数或者是负数，形式上，我们可以假设将不同的数字放到两个堆里面，相当于是01选择。我们可以假设其中的正数和为 sp ，然后其中的负数的绝对值和为 sn ，那么就有：

$$sp - sn = target$$

由于 $sp + sn = s$ ， s 是数组的和，就可以得到：

$$s - sn - sn = target \; s - target = 2 * sn$$

因此我们可以得到其中的 $sn = (s - target)/2$ ，这是一个固定的值，那么问题就转换为了从一堆数字中选取和恰好为 sn 的方案数，这是一个经典的01背包题目。

参考代码：

```
class Solution:
    def findTargetSumWays(self, nums: List[int], target: int) -> int:
        s = sum(nums) - target
        if s < 0 or s % 2 != 0:
            return 0
        sn = s // 2

        # 01 knapsack
        n = len(nums)
        dp = [0] * (sn + 1)
        dp[0] = 1
        for i in range(n):
            for j in range(sn, nums[i]-1, -1):
                dp[j] += dp[j-nums[i]]

        return dp[sn]
```

2、分割等和子集

此题和上面的题目非常类似，读者可以自己体会。

参考代码：

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        s = sum(nums)
        if s % 2 != 0:
            return False
        s //= 2
        n = len(nums)
        dp = [0] * (s + 1)
        dp[0] = 1
        for i in range(n):
            for j in range(s, nums[i]-1, -1):
                dp[j] |= dp[j-nums[i]]
                if j == s and dp[j]:
                    return True
        return False
```

这道题目中的dp值只包含了0或者1，所以可以使用bitset优化。对于位置 j 而言， 每一次 $dp[j]$ 都是或上了前面的 $nums[i]$ 位置上的数字 $dp[j-nums[i]]$ 而已。

因此我们可以将整个数组看成是一个bitset去优化，并行计算所有的位置。

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        total = sum(nums)
        if total % 2 != 1:
            target = total // 2
            dp = 1
            for num in nums:
                dp |= dp << num
            return (dp >> target) & 1 == 1
        return False
```

3、零钱兑换 I

本题就是最为经典的完全背包问题。

参考代码：

```python
inf = 0x3f3f3f3f

class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [inf] * (amount + 1)
        dp[0] = 0
```

```
        for c in coins:
            for v in range(c, amount+1):
                dp[v] = min(dp[v], dp[v-c]+1)
        return dp[amount] if dp[amount] < inf else -1
```

4、零钱兑换 II

这道题目和上题有所区别，这道题目更加关注的是不同的方案数量，但是也是属于完全背包的问题范畴，不过最优化目标函数变化了。

```
class Solution:
    def change(self, amount: int, coins: List[int]) -> int:
        dp = [0] *  (amount + 1)
        dp[0] = 1
        for c in coins:
            for v in range(c, amount+1):
                dp[v] += dp[v-c]
        return dp[amount]
```

基础数据结构经典问题（链表、队列、栈）

字符串经典问题

基础图论问题

基础计算几何问题