

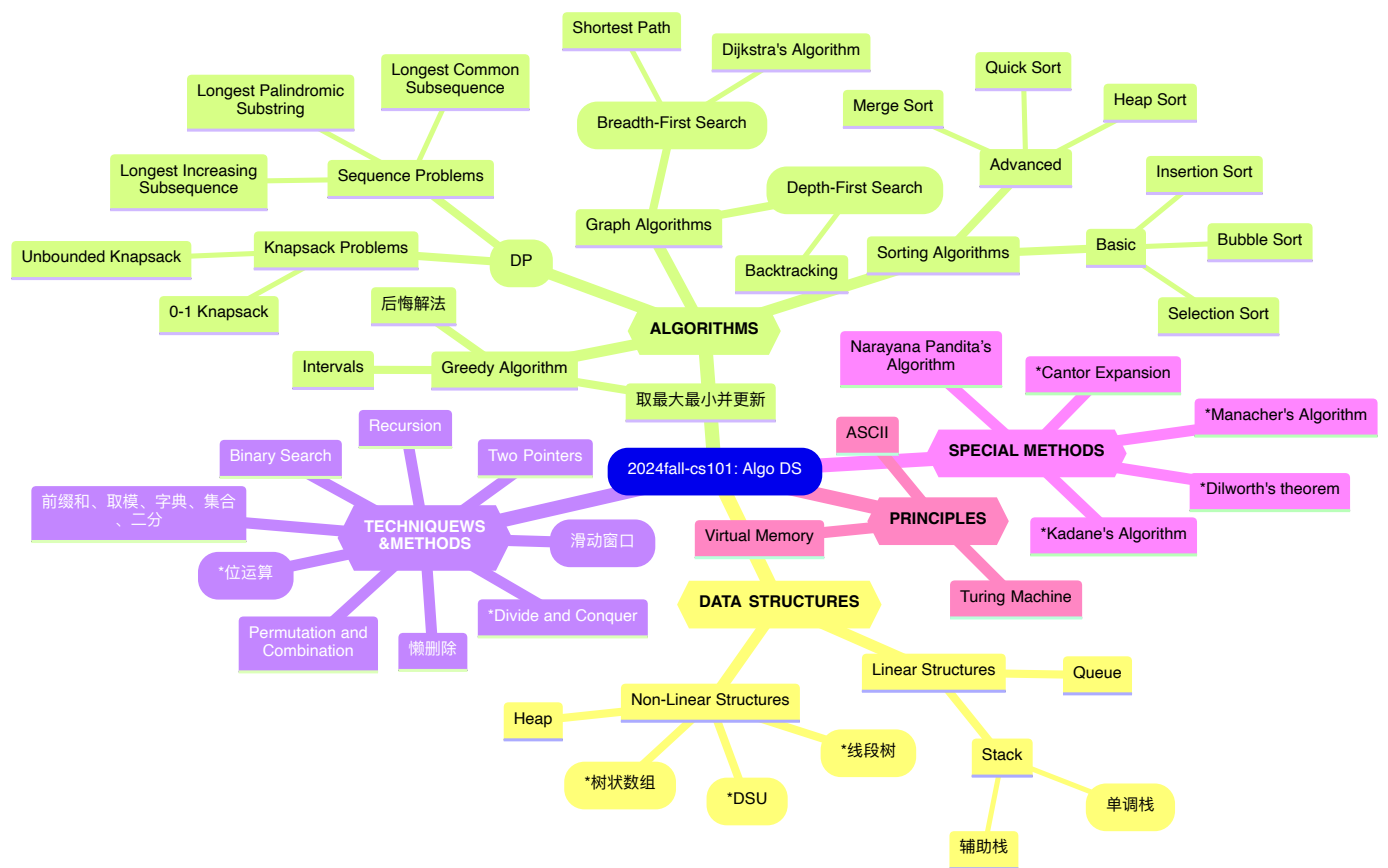
20241202-Week13 计概知识图谱

Updated 1446 GMT+8 Dec 03, 2024

2024 fall, Compiled by Hongfei Yan

Logs:

查看: https://github.com/GMyhf/2023fall-cs101/blob/main/cheatsheet/review_and_thoughts-202312-HURuicheng.md
<https://github.com/GMyhf/2023fall-cs101/blob/main/cheatsheet/cheatsheet-20231226-JIANGZixuan.md>
<https://github.com/GMyhf/2023fall-cs101/blob/main/cheatsheet/DailyOption-202312-DENGJinwen.md>



Knowledge Graph of 2024fall-cs101: Algo DS

Recap

作业示例474D. Flowers

dp, *1700, <https://codeforces.com/contest/474/problem/D>

We saw the little game Marmot made for Mole's lunch. Now it's Marmot's dinner time and, as we all know, Marmot eats flowers. At every dinner he eats some red and white flowers. Therefore a dinner can be represented as a sequence of several flowers, some of them white and some of them red.

But, for a dinner to be tasty, there is a rule: Marmot wants to eat white flowers only in groups of size k .

Now Marmot wonders in how many ways he can eat between a and b flowers. As the number of ways could be very large, print it modulo $1000000007(10^9 + 7)$.

Input

Input contains several test cases.

The first line contains two integers t and k ($1 \leq t, k \leq 10^5$), where t represents the number of test cases.

The next t lines contain two integers a_i and b_i ($1 \leq a_i \leq b_i \leq 10^5$), describing the i -th test.

Output

Print t lines to the standard output. The i -th line should contain the number of ways in which Marmot can eat between a_i and b_i flowers at dinner modulo $1000000007(10^9 + 7)$.

Examples

Input

1	3 2
2	1 3
3	2 3
4	4 4

Output

1	6
2	5
3	5

Note

- For $K = 2$ and length 1 Marmot can eat (R).
- For $K = 2$ and length 2 Marmot can eat (RR) and (WW).
- For $K = 2$ and length 3 Marmot can eat (RRR), (RWW) and (WWR).

- For $K = 2$ and length 4 Marmot can eat, for example, (WWWW) or (RWWR), but for example he can't eat (WWWR).

思路：题目本身就是一个普通的“上楼梯”，但是这里不用前缀和来查询会超时

```

1 MAX = 1000000007
2 t, k = map(int, input().split())
3 MOD = int(1e9+7)
4 MAXN = 100001
5 dp = [0]*MAXN
6 s = [0]*MAXN
7 dp[0] = 1
8 s[0] = 1
9 for i in range(1, MAXN):
10     if i >= k:
11         dp[i] = (dp[i-1]+dp[i-k]) % MOD
12     else:
13         dp[i] = dp[i-1] % MOD
14     s[i] = (s[i-1]+dp[i]) % MOD
15
16 for _ in range(t):
17     a, b = map(int, input().split())
18     print((s[b]-s[a-1]+MOD) % MOD)
19

```

作业示例12029: 水淹七军

bfs, dfs, <http://cs101.openjudge.cn/practice/12029/>

随着最后通牒的递出，C国的总攻也开始了，由于C国在地形上的优势，C国总司令下令采用水攻，剿灭A国最后的有生力量。

地形图是一个 $M \times N$ 的矩阵，矩阵上每一个点都对应着当前点的高度。C国总司令将选择若干个点进行放水。根据水往低处流的特性，水可以往四个方向的流动，被淹的地方的水面高度便和放水点的高度一样。然而，A国不是一马平川的，所以总会有地方是淹没不到的。你的任务很简单，判断一下A国司令部会不会被淹没掉。

我们将给你完整的地形图，然后给出A国司令部所在位置，给出C国将在哪几个点进行放水操作。你所需要的，就是给出A国司令部会不会被水淹。

输入

第一行：一个整数 K ，代表数据组数。

对于每一组数据：

第1行：符合题目描述的两个整数， $M(0 < M \leq 200)$ 、 $N(0 < N \leq 200)$ 。

第2行至 $M+1$ 行：每行 N 个数，以空格分开，代表这个矩阵上的各点的高度值 $H(0 \leq H \leq 1000)$ 。

第 $M+2$ 行：两个整数 $I(0 < I \leq M)$ 、 $J(0 < J \leq N)$ ，代表司令部所在位置。

第 $M+3$ 行：一个整数 $P(0 < P \leq M \times N)$ ，代表放水点个数。

第 $M+4$ 行至 $M+P+4$ 行：每行两个整数 $X(0 < X \leq M)$ 、 $Y(0 < Y \leq N)$ ，代表放水点。

输出

对于每组数据，输出一行，如果被淹则输出Yes，没有则输出No。

样例输入

```
1 1
2 5 5
3 1 1 1 1 1
4 1 0 0 0 1
5 1 0 1 0 1
6 1 0 0 0 1
7 1 1 1 1 1
8 3 3
9 2
10 1 1
11 2 2
```

样例输出

```
1 No
```

提示

样例中左上角的位置是(1, 1), 右上角的位置是(1, 5), 右下角的位置是(5, 5)

根据样例，可以这样理解：如果司令部与周围水等高，不算淹没。

不用visited的原因，有的在某些情况下也需要重新遍历。比如之前淹没的高度为h，之后放水的高度 $H > h$ ，此时就需要重新淹没。即可以不用visited，直接用water_height矩阵（每次洪泛更新），只要扩展点的高度小于当前water_height_value。

bfs实现

```
1 from collections import deque
2 import sys
3 input = sys.stdin.read
4
5 # 判断坐标是否有效
6 def is_valid(x, y, m, n):
7     return 0 <= x < m and 0 <= y < n
8
9 # 广度优先搜索模拟水流
10 def bfs(start_x, start_y, start_height, m, n, h, water_height):
11     dx = [-1, 1, 0, 0]
12     dy = [0, 0, -1, 1]
13     q = deque([(start_x, start_y, start_height)])
14     water_height[start_x][start_y] = start_height
15
16     while q:
```

```

17     x, y, height = q.popleft()
18     for i in range(4):
19         nx, ny = x + dx[i], y + dy[i]
20         if is_valid(nx, ny, m, n) and h[nx][ny] < height:
21             if water_height[nx][ny] < height:
22                 water_height[nx][ny] = height
23                 q.append((nx, ny, height))
24
25 # 主函数
26 def main():
27     data = input().split() # 快速读取所有输入数据
28     idx = 0
29     k = int(data[idx])
30     idx += 1
31     results = []
32
33     for _ in range(k):
34         m, n = map(int, data[idx:idx + 2])
35         idx += 2
36         h = []
37         for i in range(m):
38             h.append(list(map(int, data[idx:idx + n])))
39             idx += n
40         water_height = [[0] * n for _ in range(m)]
41
42         i, j = map(int, data[idx:idx + 2])
43         idx += 2
44         i, j = i - 1, j - 1
45
46         p = int(data[idx])
47         idx += 1
48
49         for _ in range(p):
50             x, y = map(int, data[idx:idx + 2])
51             idx += 2
52             x, y = x - 1, y - 1
53             if h[x][y] <= h[i][j]:
54                 continue
55             bfs(x, y, h[x][y], m, n, h, water_height)
56
57         results.append("Yes" if water_height[i][j] > 0 else "No")
58
59     sys.stdout.write("\n".join(results) + "\n")
60
61 if __name__ == "__main__":
62     main()

```

作业示例5.最长回文子串

dp, two pointers, string, <https://leetcode.cn/problems/longest-palindromic-substring/>

给你一个字符串 `s`，找到 `s` 中最长的

回文子串。

示例 1:

```
1  输入: s = "babad"
2  输出: "bab"
3  解释: "aba" 同样是符合题意的答案。
```

示例 2:

```
1  输入: s = "cbbd"
2  输出: "bb"
```

提示:

- `1 <= s.length <= 1000`
- `s` 仅由数字和英文字母组成

Plan

1. Initialize a 2D list `dp` where `dp[i][j]` will be `True` if the substring `s[i:j+1]` is a palindrome.
2. Iterate through the string in reverse order to fill the `dp` table.
3. For each character, check if the substring is a palindrome by comparing the characters at the ends and using the previously computed values in `dp`.
4. Keep track of the start and end indices of the longest palindromic substring found.
5. Return the substring defined by the start and end indices.

对于一个子串而言，如果它是回文串，并且长度大于 2，那么将它首尾的两个字母去除之后，它仍然是个回文串。

状态: `dp[i][j]` 表示子串 `s[i:j+1]` 是否为回文子串

状态转移方程: `dp[i][j] = dp[i+1][j-1] ^ (s[i] == s[j])`

动态规划中的边界条件，即子串的长度为 1 或 2。对于长度为 1 的子串，它显然是个回文串；对于长度为 2 的子串，只要它的两个字母相同，它就是一个回文串。

```
1  class Solution:
2      def longestPalindrome(self, s: str) -> str:
3          n = len(s)
4          if n == 0:
5              return ""
6
7          # Initialize the dp table
8          dp = [[False] * n for _ in range(n)]
9          start, max_length = 0, 1
```

```

10
11     # Every single character is a palindrome
12     for i in range(n):
13         dp[i][i] = True
14
15     # Check for palindromes of length 2
16     for i in range(n - 1):
17         if s[i] == s[i + 1]:
18             dp[i][i + 1] = True
19             start = i
20             max_length = 2
21
22     # Check for palindromes of length greater than 2
23     for length in range(3, n + 1):
24         for i in range(n - length + 1):
25             j = i + length - 1
26             if s[i] == s[j] and dp[i + 1][j - 1]:
27                 dp[i][j] = True
28                 start = i
29                 max_length = length
30
31     return s[start:start + max_length]
32
33 if __name__ == "__main__":
34     sol = Solution()
35     print(sol.longestPalindrome("babad")) # Output: "bab" or "aba"
36     print(sol.longestPalindrome("cbbd")) # Output: "bb"

```

Plan

1. Initialize variables to store the start and end indices of the longest palindromic substring.
2. Iterate through each character in the string, treating each character and each pair of consecutive characters as potential centers of palindromes.
3. For each center, expand outwards while the characters on both sides are equal.
4. Update the start and end indices if a longer palindrome is found.
5. Return the substring defined by the start and end indices.

```

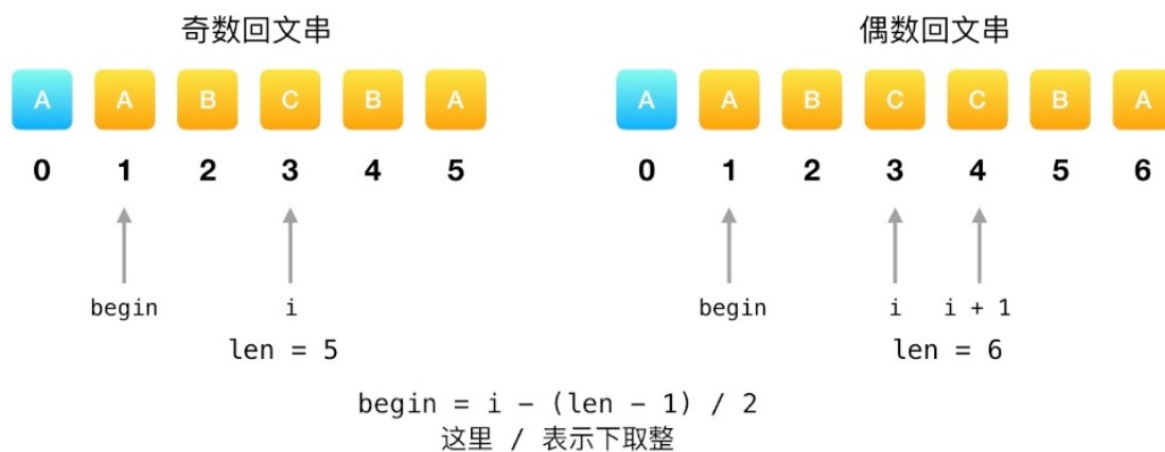
1  class Solution:
2      def longestPalindrome(self, s: str) -> str:
3          if not s:
4              return ""
5
6          start, end = 0, 0
7
8          for i in range(len(s)):
9              odd_len = self.expandAroundCenter(s, i, i)
10             even_len = self.expandAroundCenter(s, i, i + 1)
11             max_len = max(odd_len, even_len)

```

```

12         if max_len > end - start:
13             start = i - (max_len - 1) // 2
14             end = i + max_len // 2
15
16         return s[start:end + 1]
17
18     def expandAroundCenter(self, s: str, left: int, right: int) -> int:
19         while left >= 0 and right < len(s) and s[left] == s[right]:
20             left -= 1
21             right += 1
22         return right - left - 1
23
24 if __name__ == "__main__":
25     sol = Solution()
26     print(sol.longestPalindrome("babad")) # Output: "bab" or "aba"
27     print(sol.longestPalindrome("cbbd")) # Output: "bb"

```



这个双指针是从中间往两边跑。

Manacher算法

<https://leetcode.cn/problems/longest-palindromic-substring/solutions/255195/zui-chang-hui-wen-zi-chuan-by-leetcode-solution/>

```

1  class Solution:
2      def expand(self, s, left, right):
3          while left >= 0 and right < len(s) and s[left] == s[right]:
4              left -= 1
5              right += 1
6          return (right - left - 2) // 2
7
8      def longestPalindrome(self, s: str) -> str:
9          end, start = -1, 0
10         s = '#' + '#'.join(list(s)) + '#'
11         arm_len = []
12         right = -1
13         j = -1
14         for i in range(len(s)):

```



```

15         if right >= i:
16             i_sym = 2 * j - i
17             min_arm_len = min(arm_len[i_sym], right - i)
18             cur_arm_len = self.expand(s, i - min_arm_len, i + min_arm_len)
19         else:
20             cur_arm_len = self.expand(s, i, i)
21         arm_len.append(cur_arm_len)
22         if i + cur_arm_len > right:
23             j = i
24             right = i + cur_arm_len
25         if 2 * cur_arm_len + 1 > end - start:
26             start = i - cur_arm_len
27             end = i + cur_arm_len
28     return s[start+1:end+1:2]
29

```

这段代码是用于解决“最长回文子串”问题的一个Python实现。它使用了Manacher算法的变种，通过在每个字符间插入特殊字符（这里是#）来处理奇数和偶数长度的回文字符串。让我们逐段解析这个解决方案：

扩展函数 `expand`

- 此函数尝试从给定的中心点向两边扩展，以找到以该中心点为中心的最大回文半径。
- 如果左右指针指向的字符相等且都在字符串范围内，则继续向外扩展。
- 返回值为最大回文半径，即 $(right - left - 2) // 2$ 。

主函数 `longestPalindrome`

- 首先，将原始字符串 `s` 转换成一个新字符串，其中每个字符之间都插入了一个 `#`，这样可以确保所有可能的回文子串都是奇数长度，便于处理。
- `end` 和 `start` 变量用于记录最长回文子串的边界。
- `arm_len` 列表存储每个位置的回文臂长。
- `right` 和 `j` 分别表示当前已知最右端的回文子串的右边界及其对应的中心点。
- 在遍历新字符串时，如果当前索引 `i` 位于 `right` 内，那么利用对称性，计算出 `i` 关于 `j` 的对称点 `i_sym`，并确定最小臂长。
- 使用 `expand` 函数计算当前位置的实际臂长，并更新 `arm_len` 列表。
- 如果当前臂长加上 `i` 超过了 `right`，则更新 `right` 和 `j`。
- 如果当前找到的回文子串比之前记录的更长，则更新 `start` 和 `end`。
- 最后，返回原始字符串中对应最长回文子串的部分，注意去掉添加的 `#` 符号。

整个算法的核心思想是利用已经计算过的回文信息来减少重复计算，从而提高效率。这种方法的时间复杂度接近 $O(n)$ ，空间复杂度为 $O(n)$ 。

`min_arm_len = min(arm_len[i_sym], right - i)` 如何理解？

`min_arm_len = min(arm_len[i_sym], right - i)` 这一行代码是用来确定从当前位置 `i` 开始进行中心扩展时，可以跳过的最小臂长。这里有两个关键点需要理解：

1. `arm_len[i_sym]`:

- `i_sym` 是位置 `i` 关于当前已知最右边回文子串中心 `j` 的对称点，计算方式为 `2 * j - i`。
- `arm_len[i_sym]` 表示的是对称点 `i_sym` 处的回文臂长。如果 `i_sym` 位置的回文臂长足够大，那么根据对称性，我们可以在 `i` 位置直接获得至少相同的臂长，因为这两个位置关于 `j` 对称。

2. `right - i`:

- `right` 是当前已知最右端回文子串的边界。
- `right - i` 表示从位置 `i` 到当前已知最右端的距离。由于我们已经知道在 `right` 之前的字符都是回文的一部分，因此在 `i` 位置进行扩展时，我们至少可以达到 `right` 位置，而不需要重新验证这些字符。

综合以上两点，`min_arm_len` 取两者中的较小值，确保了以下几点：

- 如果 `i_sym` 位置的臂长小于或等于 `right - i`，我们可以直接使用 `i_sym` 位置的臂长，因为它不会超出已知的回文区域。
- 如果 `i_sym` 位置的臂长大于 `right - i`，则我们只能保证到 `right` 位置是回文，所以取 `right - i` 作为最小臂长。

这样做的目的是为了减少不必要的重复计算，通过利用之前计算的结果（即 `arm_len` 中的信息）来加速找到当前位置的最长回文臂长的过程。这实际上是Manacher算法中的一种优化手段，它允许我们在某些情况下快速跳过已经确认的部分，从而提高算法的整体效率。

作业示例02802: 小游戏

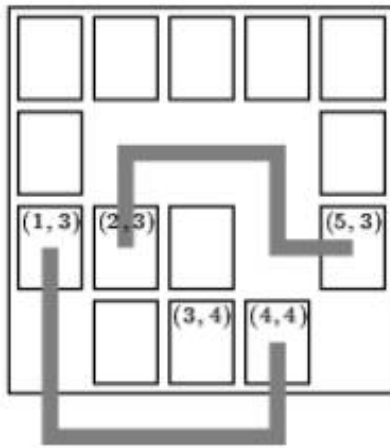
bfs, <http://cs101.openjudge.cn/practice/02802/>

一天早上，你起床的时候想：“我编程这么牛，为什么不能靠这个赚点小钱呢？”因此你决定编写一个小游戏。

游戏在一个分割成 $w * h$ 个正方格子的矩形板上进行。如图所示，每个正方格子上可以有一张游戏卡片，当然也可以没有。

当下面的情况满足时，我们认为两个游戏卡片之间有一条路径相连：

路径只包含水平或者竖直的直线段。路径不能穿过别的游戏卡片。但是允许路径临时的离开矩形板。下面是一个例子：



这里在 (1, 3) 和 (4, 4) 处的游戏卡片是可以相连的。而在 (2, 3) 和 (3, 4) 处的游戏卡是不相连的，因为连接他们的每条路径都必须穿过别的游戏卡片。

你现在要在小游戏里面判断是否存在一条满足题意的路径能连接给定的两个游戏卡片。

输入

输入包括多组数据。一个矩形板对应一组数据。每组数据包括的第一行包括两个整数w和h ($1 \leq w, h \leq 75$), 分别表示矩形板的宽度和长度。下面的h行, 每行包括w个字符, 表示矩形板上的游戏卡片分布情况。使用'X'表示这个地方有一个游戏卡片; 使用空格表示这个地方没有游戏卡片。

之后的若干行上每行上包括4个整数 x_1, y_1, x_2, y_2 ($1 \leq x_1, x_2 \leq w, 1 \leq y_1, y_2 \leq h$)。给出两个卡片在矩形板上的位置（注意：矩形板左上角的坐标是(1, 1)）。输入保证这两个游戏卡片所处的位置是不相同的。如果一行上有4个0，表示这组测试数据的结束。

如果一行上给出 $w = h = 0$ ，那么表示所有的输入结束了。

输出

对每一个矩形板，输出一行“Board #n:”，这里n是输入数据的编号。然后对每一组需要测试的游戏卡片输出一行。这一行的开头是“Pair m:”，这里m是测试卡片的编号（对每个矩形板，编号都从1开始）。接下来，如果可以相连，找到连接这两个卡片的所有路径中包括线段数最少的路径，输出“k segments.”，这里k是找到的最优路径中包括的线段的数目；如果不能相连，输出“impossible.”。

每组数据之后输出一个空行。

样例输入

1	5	4			
2	X	X	X	X	X
3	X			X	
4	X	X	X	X	
5		X	X	X	
6	2	3	5	3	
7	1	3	4	4	
8	2	3	3	4	
9	0	0	0	0	
10	0	0			

样例输出

```
1 Board #1:
2 Pair 1: 4 segments.
3 Pair 2: 3 segments.
4 Pair 3: impossible.
```

来源：翻译自Mid-Central European Regional Contest 1999的试题

bfs

这个题目比较麻烦，因为外圈还可以走，需要在输入矩阵包一圈。另外，就是行列与我们平时练习行列刚好反着。因为没有走到end之前的线段最短，不能保证总的线段最短。需要穷举队列，找到的最短都append到ans列表，最后min(ans)。

```
1 from collections import deque
2
3
4 def bfs(start, end, grid, h, w):
5     queue = deque([start])
6     in_queue = set()
7     dirs = [(0, -1), (-1, 0), (0, 1), (1, 0)]
8
9     ans = []
10    while queue:
11        x, y, d_i_r, seg = queue.popleft()
12        # print(x,y,end)
13        if (x, y) == end:
14            # return seg
15            ans.append(seg)
16            break
17
18        for i, (dx, dy) in enumerate(dirs):
19            nx, ny = x + dx, y + dy
20
21            if 0 <= nx < h + 2 and 0 <= ny < w + 2 and ((nx, ny, i) not in in_queue):
22                new_dir = i
23                new_seg = seg if new_dir == d_i_r else seg + 1
24                if (nx, ny) == end:
25                    # return new_seg
26                    ans.append(new_seg)
27                    continue
28
29                if grid[nx][ny] != 'X':
30                    in_queue.add((nx, ny, i))
31                    queue.append((nx, ny, new_dir, new_seg))
32
33    if len(ans) == 0:
34        return -1
35    else:
36        return min(ans)
```

```

37
38
39 board_num = 1
40 while True:
41     w, h = map(int, input().split())
42     if w == h == 0:
43         break
44
45     # grid = [[' ']*(w+2)] + \
46     # [[' ']+list(input())+' ' for _ in range(h)] + \
47     # [[' ']*(w+2)]
48     grid = [' '*(w+2)] + [' '+input()+' ' for _ in range(h)] + [' '*(w+
2)]
49     print(f"Board #{board_num}:")
50     pair_num = 1
51     while True:
52         y1, x1, y2, x2 = map(int, input().split())
53         if x1 == y1 == x2 == y2 == 0:
54             break
55
56         start = (x1, y1, -1, 0)
57         end = (x2, y2)
58
59         seg = bfs(start, end, grid, h, w)
60         if seg == -1:
61             print(f"Pair {pair_num}: impossible.")
62         else:
63             print(f"Pair {pair_num}: {seg} segments.")
64             pair_num += 1
65
66     print()
67     board_num += 1

```

《算法基础。。》上面讲到4.3例题：小游戏，书上给出的是dfs。但是经过同学和助教调试，发现dfs与先沿着哪个邻居出发有关，导致剪枝可能失效。因为可能拿不到一个相对较好的结果，便于比较剪枝。所以最好用bfs完成。

其实所有求最短、最长的问题都能用heapq实现，在图搜索中搭配bfs尤其好用。

利用heapq优先队列的做法，因为每次都取当前队列中线段最小值前进，可以保证最后总的线段最短。这个实际上是Dijkstra。

```

1  # 23 工学院 苏王捷
2  import heapq
3
4  num1 = 1
5  while True:
6      w, h = map(int, input().split())
7      if w == 0 and h == 0:

```

```

8         break
9     print(f"Board #{num1}:")
10    martix = [[" "]*(w+2)] + [[" "]+list(input())+" " for _ in range(h)] +
11    [[" "]*(w+2)]
12    dir = [(0, 1), (0, -1), (1, 0), (-1, 0)]
13    num2 = 1
14    while True:
15        x1, y1, x2, y2 = map(int, input().split())
16        if x1 == 0 and x2 == 0 and y1 == 0 and y2 == 0:
17            break
18        queue, flag = [], False
19        in_queue = set()
20        heapq.heappush(queue, (0, x1, y1, -1))
21        martix[y2][x2] = " "
22        in_queue.add((-1, x1, y1))
23        while queue:
24            step, x, y, dirs = heapq.heappop(queue)
25            if x == x2 and y == y2:
26                flag = True
27                break
28            for i, (dx, dy) in enumerate(dir):
29                px, py = x + dx, y + dy
30                if 0 <= px <= w + 1 and 0 <= py <= h + 1 and (i, px, py) not in
31                in_queue and martix[py][px] != "X":
32                    in_queue.add((i, px, py))
33                    heapq.heappush(queue, (step + (dirs != i), px, py, i))
34            if flag:
35                print(f"Pair {num2}: {step} segments.")
36            else:
37                print(f"Pair {num2}: impossible.")
38            martix[y2][x2] = "X"
39            num2 += 1
40    print()
41    num1 += 1

```

1 能申请到 10^{18} 内存吗?

我的机器是macOS Sonoma 14.6.1，最大可以申请到 276.00 GB（即接近于 2^{38} ）。计算方法如下所述。

10^{18} 有多大

要将 10^{18} 字节转换为更常见的存储单位，如GB（吉字节）或TB（太字节），我们需要了解这些单位之间的换算关系。在二进制表示中，这些单位是基于2的幂来定义的，但在十进制表示中，它们通常基于10的幂来定义。

- 1 GB (Gigabyte, 吉字节) = 10^9 字节
- 1 TB (Terabyte, 太字节) = 10^{12} 字节

因此， 10^{18} 字节可以被直接转换为：

- $10^{18} \div 10^9 = 10^9$ GB
- $10^{18} \div 10^{12} = 10^6$ TB

这表示 10^{18} 字节等于 10亿 GB 或者 1百万 TB。

判断使用的Python解释器是否64位

要判断你正在使用的Python解释器是32位还是64位，可以通过几种方法来完成。这里我将展示一种简单的方法，即使用Python内置的 `platform` 模块或 `sys` 模块来获取相关信息。

方法 1: 使用 `platform` 模块

`platform` 模块提供了很多关于平台的信息，包括体系结构（architecture），可以用来区分32位和64位版本。

```
1 import platform
2
3 print(platform.architecture())
```

这个函数会返回一个元组，其中第一个元素表示Python解释器的位数（例如 '32bit' 或 '64bit'）。

方法 2: 使用 `sys` 模块

`sys` 模块中的 `maxsize` 属性也可以用来间接判断Python解释器的位数。对于32位Python，`sys.maxsize` 的值通常为 $2^{31} - 1$ (即 2147483647)，而对于64位Python，这个值通常为 $2^{63} - 1$ (即 9223372036854775807)。

```
1 import sys
2
3 if sys.maxsize > 2**32:
4     print("64-bit Python")
5 else:
6     print("32-bit Python")
```

下面我将执行一段代码来演示如何使用这两种方法之一来确定当前Python环境的位数。

```
1 import platform
2 print(platform.architecture()[0])
```

或者使用 `sys` 模块的方法：

```
1 import sys
2 if sys.maxsize > 2**32:
3     print("64-bit Python")
4 else:
5     print("32-bit Python")
```

测试Python程序可以申请的最大内存量

python能够申请到的最大内存

Python程序能够申请到的最大内存量主要取决于几个因素，包括操作系统、硬件配置以及Python解释器本身的限制。下面是一些关键点：

1. 操作系统限制：

- 不同的操作系统对进程可以使用的最大内存有不同的默认限制。例如，在Linux上，可以通过 `ulimit -a` 命令查看当前用户的资源限制，其中包括了最大虚拟内存大小（`virtual memory`）。
- 在Windows上，理论上一个32位进程最多可以访问4GB的地址空间，而64位进程则可以访问更大的地址空间。

2. 硬件限制：

- 物理内存（RAM）的数量直接限制了程序可用的实际物理内存。即使操作系统允许更大的虚拟内存，实际能使用的内存仍然受限于物理内存加上交换空间（swap space）的总量。
- 交换空间可以在一定程度上扩展可用内存，但使用大量交换空间会导致性能显著下降。

3. Python解释器限制：

- Python本身并没有设置一个固定的内存上限，但它会受到上述因素的影响。
- 在32位Python版本中，由于地址空间的限制，通常最大可分配内存为2-4GB左右。
- 64位Python版本理论上可以访问更多内存，但是实际可用量依然受限于操作系统的限制和可用的物理及虚拟内存。

4. Python内部机制：

- Python对象的内存管理由Python解释器负责，它使用垃圾回收机制来自动释放不再使用的内存。
- 对于非常大的数据集，可能需要考虑使用更高效的数据结构或第三方库（如NumPy, Pandas等），这些库针对大数据处理进行了优化，并且支持在内存之外存储数据（例如，使用磁盘上的文件作为后端）。

5. 增加内存限制的方法：

- 在Linux上，可以通过修改 `/etc/security/limits.conf` 文件或使用 `ulimit` 命令来提高单个进程的内存限制。
- 对于需要处理特别大数据集的情况，可以考虑使用分布式计算框架（如Apache Spark）、数据库或者其他专门设计用于处理大规模数据的技术。

总之，Python程序能够申请到的最大内存量不是一个固定值，而是依赖于运行环境的具体条件。

```
% ulimit -a
```

```
-t: cpu time (seconds)    unlimited
-f: file size (blocks)    unlimited
-d: data seg size (kbytes) unlimited
-s: stack size (kbytes)   8176
-c: core file size (blocks) 0
-v: address space (kbytes) unlimited
-l: locked-in-memory size (kbytes) unlimited
-u: processes             10666
-n: file descriptors      2560
```

虚拟地址空间（计算机原理2/3）

三大计算机原理之一，@Book_my_flight_v0.3.md

计算机的基础架构自从 20 世纪 40 年代起就已经形成规范，包括处理器、存储指令和数据的内存、输入和输出设备。它通常叫作冯·诺依曼架构，以约翰·冯·诺依曼（德语：John Von Neumann，1903 年 12 月 28 日－1957 年 2 月 8 日）的名字来命名，他在 1946 年发表的论文里描述了这一架构。论文的开头句，用现在的专门术语来说就是，CPU 提供算法和控制，而 RAM 和磁盘则是记忆存储，键盘、鼠标和显示器与操作人员交互。其中需要重点理解的是与存储相关的进程的虚拟地址空间。

虚拟存储器是一个抽象概念，它为每个进程提供了一个假象，好像每个进程都在独占地使用主存。每个进程看到的存储器都是一致的，称之为虚拟地址空间。如图 1-15 所示的是 Linux 进程的虚拟地址空间（其他 Unix 系统的设计与此类似）。在 Linux 中，最上面的四分之一的地址空间是预留给操作系统中的代码和数据的，这对所有进程都一样。底部的四分之三的地址空间用来存放用户进程定义的代码和数据。请注意，图中的地址是从下往上增大的。

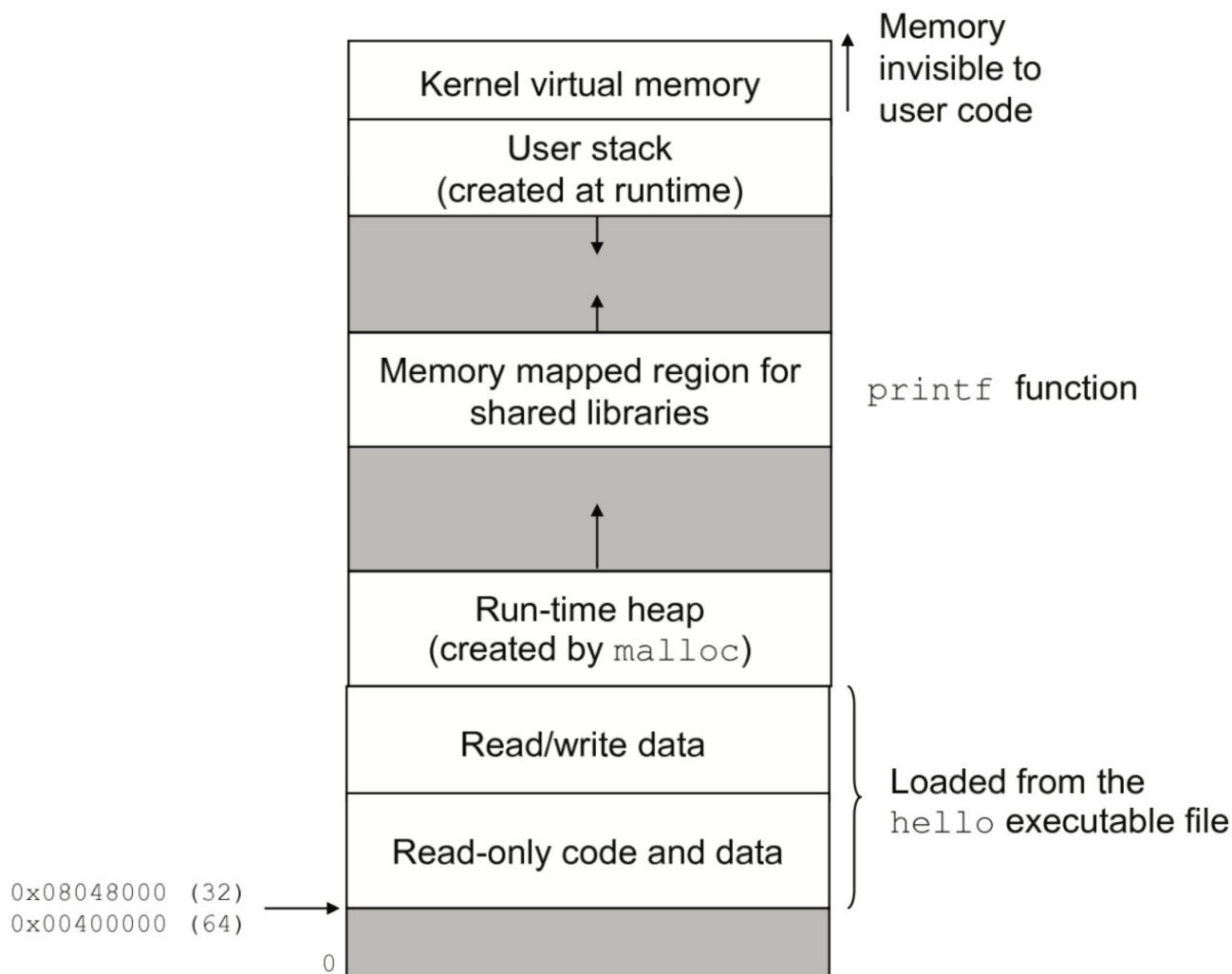


图1-15 进程的虚拟地址空间 (Process virtual address space) (注: 图片来源为 Randal Bryant[8], 2015 年3月)

每个进程看到的虚拟地址空间由准确定义的区 (area) 构成, 每个区都有专门的功能。简单看下每一个区, 从最低的地址开始, 逐步向上研究。

- 程序代码和数据 (code and data)。代码是从同一固定地址开始, 紧接着的是和全局变量相对应的数据区。代码和数据区是由可执行目标文件直接初始化的, 示例中就是可执行文件hello。
- 堆 (heap)。紧随代码和数据区之后的是运行时堆 (Run-time heap)。代码和数据区是在进程一旦开始运行时就被指定了大小的, 与此不同, 作为调用像 malloc 和 free 这样的 C 标准库函数的结果, 堆可以在运行时动态地扩展和收缩。
- 共享库 (shared libraries)。在地址空间的中间附近是一块用来存放像标准库和数学库这样共享库的代码和数据的区域。共享库的概念非常强大。
- 栈 (stack)。位于用户虚拟地址空间顶部的是用户栈, 编译器用它来实现函数调用。和堆一样, 用户栈 (User stack) 在程序执行期间可以动态地扩展和收缩。特别地, 每次我们调用一个函数时, 栈就会增长。每次我们从函数返回时, 栈就会收缩。
- 内核虚拟存储器 (kernel virtual memory)。内核是操作系统总是驻留在存储器中的部分。地址空间顶部是为内核预留的。应用程序不允许读写这个区域的内容或者直接调用内核代码定义的函数。

虚拟存储器的运作需要硬件和操作系统软件间的精密复杂的互相合作，包括对处理器生成的每个地址的硬件翻译。基本思想是把一个进程虚拟存储器的内容存储在磁盘上，然后用主存作为磁盘的高速缓存。

要测试Python程序可以申请的最大内存量，你可以编写一个简单的脚本，该脚本会尝试分配越来越多的内存，直到达到系统限制或Python解释器本身的限制。这个过程通常涉及到创建一个越来越大的列表（或其他数据结构），并填充它，直到内存不足。

请注意，这样的测试可能会导致你的系统变得非常慢，甚至可能崩溃，因为它会消耗大量的RAM。因此，在进行这种测试之前，请确保你了解风险，并且最好在受控环境中执行此操作，例如虚拟机或有足够空闲资源的机器上。

```
1 import os
2 import sys
3 import gc # 垃圾回收模块
4
5
6 def allocate_memory(chunk_size=1024 * 1024 * 1024, max_attempts=1000):
7     """
8     尝试分配内存，每次增加chunk_size字节，直到无法分配更多。
9
10    :param chunk_size: 每次尝试分配的内存大小（以字节为单位）
11    :param max_attempts: 最大尝试次数
12    """
13    data = []
14    total_allocated = 0
15    for i in range(max_attempts):
16        try:
17            # 尝试分配额外的内存
18            data.append(' ' * chunk_size)
19            total_allocated += chunk_size
20            print(f"Allocated {total_allocated / (1024 * 1024 * 1024):.2f} GB")
21        except MemoryError:
22            print("Memory allocation failed.")
23            break
24        finally:
25            # 强制垃圾回收
26            gc.collect()
27
28    print(f"Total memory allocated: {total_allocated / (1024 * 1024 * 1024):.2f} GB")
29
30
31 # 运行测试
32 allocate_memory()
```

运行结果，mac机器

Allocated 274.00 GB

Allocated 275.00 GB

Allocated 276.00 GB

Process finished with exit code 137 (interrupted by signal 9:SIGKILL)

要找出276GB是2的多少次幂，首先需要将276GB转换为字节，因为通常在计算中使用的是二进制单位。1GB等于 2^{30} 字节（在二进制表示中）。因此，276GB可以表示为 $276 * 2^{30}$ 字节。

接下来，我们需要找到一个指数 x ，使得 2^x 等于 $276 * 2^{30}$ 。这可以通过对数运算来解决：

$$x = \log_2(276 \times 2^{30})$$

$$\log_2(276 \times 2^{30}) = \log_2(276) + \log_2(2^{30})$$

$$\log_2(276) + 30 \approx 8.1073 + 30 = 38.1073$$

这意味着276GB大约等于 $2^{38.1073}$ 字节。由于幂次通常是一个整数，我们可以认为276GB最接近于 2^{38} 字节，但略大于这个值。如果你需要更精确的结果，可以使用科学计算器来获得更准确的对数值。

2 最短路径Dijkstra

示例sy386: 最短距离 简单

<https://sunnywhy.com/sfbj/10/4/386>

现有一个共 n 个顶点（代表城市）、 m 条边（代表道路）的无向图（假设顶点编号为从 0 到 $n-1$ ），每条边有各自的边权，代表两个城市之间的距离。求从 s 号城市出发到达 t 号城市的最短距离。

输入

第一行四个整数 n 、 m 、 s 、 t ($1 \leq n \leq 100, 0 \leq m \leq \frac{n(n-1)}{2}, 0 \leq s \leq n-1, 0 \leq t \leq n-1$)，分别表示顶点数、边数、起始编号、终点编号；

接下来 m 行，每行三个整数 u 、 v 、 w ($0 \leq u \leq n-1, 0 \leq v \leq n-1, u \neq v, 1 \leq w \leq 100$)，表示一条边的两个端点的编号及边权距离。数据保证不会有重边。

输出

输出一个整数，表示最短距离。如果无法到达，那么输出 -1 。

样例1

输入

```
1 6 6 0 2
2 0 1 2
3 0 2 5
4 0 3 1
5 2 3 2
6 1 2 1
7 4 5 1
```

输出

```
1 3
```

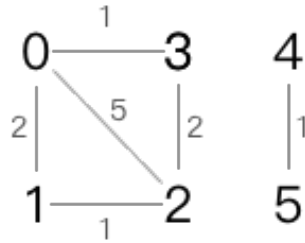
解释

对应的无向图如下图所示。

共有 3 条从 0 号顶点到 2 号顶点的路径：

1. $0 \rightarrow 3 \rightarrow 2$ ：距离为 3 ；
2. $0 \rightarrow 2$ ：距离为 5 ；
3. $0 \rightarrow 1 \rightarrow 2$ ：距离为 3 。

因此最短距离为 3 。



样例2

输入

```

1 6 6 0 5
2 0 1 2
3 0 2 5
4 0 3 1
5 2 3 2
6 1 2 1
7 4 5 1

```

输出

```

1 -1

```

解释

和第一个样例相同的图，终点换成了 5 号顶点，显然从 0 号无法到达 5 号。

需要找到从给定的起始城市到目标城市的最短距离。可以使用 Dijkstra 算法来解决这个问题。Dijkstra 算法是一种用于在图中找到最短路径的算法。它从起始节点开始，然后逐步扩展到所有可达的节点，每次选择当前最短的路径进行扩展。

以下是使用 Python 实现 Dijkstra 算法来解决这个问题的示例代码：

```

1 import heapq
2
3 def dijkstra(n, edges, s, t):
4     graph = [[] for _ in range(n)]
5     for u, v, w in edges:
6         graph[u].append((v, w))
7         graph[v].append((u, w))
8
9     pq = [(0, s)] # (distance, node)
10    visited = set()
11    distances = [float('inf')] * n
12    distances[s] = 0
13
14    while pq:
15        dist, node = heapq.heappop(pq)
16        if node == t:

```

```

17         return dist
18     if node in visited:
19         continue
20     visited.add(node)
21     for neighbor, weight in graph[node]:
22         if neighbor not in visited:
23             new_dist = dist + weight
24             if new_dist < distances[neighbor]:
25                 distances[neighbor] = new_dist
26                 heapq.heappush(pq, (new_dist, neighbor))
27     return -1
28
29 # Read input
30 n, m, s, t = map(int, input().split())
31 edges = [list(map(int, input().split())) for _ in range(m)]
32
33 # Solve the problem and print the result
34 result = dijkstra(n, edges, s, t)
35 print(result)

```

这段代码实现了 Dijkstra 算法来求解从起点到终点的最短路径。首先构建了一个图，然后使用优先队列来选择下一个要探索的节点，并在探索过程中更新最短距离。最后返回从起点到终点的最短距离。

这个版本的 Dijkstra 算法使用了一个集合 `visited` 来记录已经访问过的节点，这样可以避免对同一个节点的重复处理。当我们从优先队列中取出一个节点时，如果这个节点已经在 `visited` 集合中，那么我们就跳过这个节点，处理下一个节点。这样可以提高算法的效率。

此外，这个版本的 Dijkstra 算法还在找到目标节点 `t` 时就立即返回结果，而不是等到遍历完所有节点。这是因为 Dijkstra 算法保证了每次从优先队列中取出的节点就是当前距离最短的节点，所以当我们找到目标节点 `t` 时，就已经找到了从起始节点 `s` 到 `t` 的最短路径，无需再继续搜索。

这个版本的 Dijkstra 算法的时间复杂度仍然是 $O((V+E)\log V)$ ，其中 V 是顶点数， E 是边数。这是因为每个节点最多会被加入到优先队列中一次（当找到一条更短的路径时），并且每条边都会被处理一次（在遍历节点的邻居时）。优先队列的插入和删除操作的时间复杂度都是 $O(\log V)$ ，所以总的时间复杂度是 $O((V+E)\log V)$ 。

Dijkstra 算法是一种经典的图算法，它综合运用了多种技术，包括邻接表、集合、优先队列（堆）、贪心算法和动态规划的思想。例题：最短距离，<https://sunnywhy.com/sfbj/10/4/386>

- 邻接表：Dijkstra 算法通常使用邻接表来表示图的结构，这样可以高效地存储图中的节点和边。
- 集合：在算法中需要跟踪已经访问过的节点，以避免重复访问，这一般使用集合（或哈希集合）来实现。
- 优先队列（堆）：Dijkstra 算法中需要选择下一个要探索的节点，通常使用优先队列（堆）来维护当前候选节点的集合，并确保每次都能快速找到距离起点最近的节点。
- 贪心算法：Dijkstra 算法每次选择距离起点最近的节点作为下一个要探索的节点，这是一种贪心策略，即每次做出局部最优的选择，期望最终能达到全局最优。
- 动态规划：Dijkstra 算法通过不断地更新节点的最短距离来逐步得到从起点到各个节点的最短路径，这是一种动态规划的思想，即将原问题拆解成若干子问题，并以最优子结构来解决。

综合运用这些技术，Dijkstra 算法能够高效地求解单源最短路径问题，对于解决许多实际问题具有重要意义。

第2种写法，没有用set记录访问过的结点。

```
1 import heapq
2
3 def dijkstra(n, s, t, edges):
4     graph = [[] for _ in range(n)]
5     for u, v, w in edges:
6         graph[u].append((v, w))
7         graph[v].append((u, w))
8
9     distance = [float('inf')] * n
10    distance[s] = 0
11
12    queue = [(0, s)]
13    while queue:
14        dist, node = heapq.heappop(queue)
15        if dist != distance[node]:
16            continue
17        for neighbor, weight in graph[node]:
18            if distance[node] + weight < distance[neighbor]:
19                distance[neighbor] = distance[node] + weight
20                heapq.heappush(queue, (distance[neighbor], neighbor))
21
22    return distance[t] if distance[t] != float('inf') else -1
23
24 # 接收数据
25 n, m, s, t = map(int, input().split())
26 edges = []
27 for _ in range(m):
28     u, v, w = map(int, input().split())
29     edges.append((u, v, w))
30
31 # 调用函数
32 min_distance = dijkstra(n, s, t, edges)
33 print(min_distance)
```

第15行的判断 `if dist != distance[node]: continue` 的作用是跳过已经找到更短路径的节点。

在Dijkstra算法中，我们使用优先队列（在Python中是heapq）来存储待处理的节点，每次从队列中取出当前距离最短的节点进行处理。但是在处理过程中，有可能会多次将同一个节点加入到队列中，因为我们可能会通过不同的路径到达同一个节点，每次到达时都会将其加入到队列中。

因此，当我们从队列中取出一个节点时，需要判断这个节点当前的最短距离是否与队列中存储的距离相同。如果不同，说明这个节点在队列中等待处理的时候，已经有了一条更短的路径，所以我们可以跳过这个节点，处理下一个节点。

Dijkstra正确性证明

Proof of Dijkstra's Correctness

1 详细解释

Dijkstra算法的正确性证明主要基于贪心选择性质和最优子结构性质。下面是对Dijkstra算法正确性的详细解释：

贪心选择性质

Dijkstra算法在每一步中总是选择当前已知最短路径的顶点，并且更新其邻居顶点的距离。这种选择方式确保了每次添加到最终解中的顶点都是当前最优的选择。

最优子结构

如果从起点 s 到某个顶点 v 的最短路径是通过顶点 u ，那么从 s 到 u 的部分也必须是最短路径。这保证了局部最优解可以组合成全局最优解。

证明步骤

1. 定义：

- 让 S 表示已经确定了最短路径的顶点集合。
- 让 $V-S$ 表示尚未确定最短路径的顶点集合。
- $d[v]$ 表示从起点 s 到顶点 v 的当前已知最短距离。
- $\delta(s, v)$ 表示从起点 s 到顶点 v 的实际最短距离。

2. 初始状态：

- 算法开始时， $S = \{s\}$ ，即只包含起点 s 。
- 对于所有顶点 $v \in V - S$ ，初始化 $d[v]$ 为从 s 到 v 的直接边的权重（如果存在），否则为无穷大。

3. 不变量：

- 在每一步执行之前，对于所有 $u \in S$ ，有 $d[u] = \delta(s, u)$ 。
- 对于所有 $v \in V - S$ ，有 $d[v] \geq \delta(s, v)$ 。

4. 迭代过程：

- 在每一步中，选择 $V-S$ 中 $d[v]$ 最小的顶点 u 加入 S 。
- 更新 u 的所有邻居 v 的 $d[v]$ 值，如果通过 u 到达 v 的新路径更短，则更新 $d[v]$ 。

5. Dijkstra正确性证明，如何理解？：

- 假设在某一步骤中，我们选择了 u 加入 S ，并且 $u \neq s$ 。
- 由于 u 是 $V-S$ 中 d 值最小的顶点，因此 $d[u] \leq d[v]$ 对于所有 $v \in V - S$ 成立。
- 根据不变量， $d[u] \geq \delta(s, u)$ 。
- 如果 $d[u] > \delta(s, u)$ ，则存在一条从 s 到 u 的更短路径，但这条路径必须经过 $V-S$ 中的某个顶点 w （因为 u 是第一个被加入 S 的顶点）。

- 由于 $d[w] \geq \delta(s, w)$ ，且 $\delta(s, w) + \text{weight}(w, u) \geq \delta(s, u)$ ，所以 $d[u]$ 不可能大于 $\delta(s, u)$ 。

由于我们假设了存在一条更短的路径，即 $d[u] > \delta(s, u)$ ，那么按照Dijkstra算法更新规则， $d[u]$ 应该被更新为 $d[w] + \text{weight}(w, u)$ 或更小的值。这与 $d[u] > \delta(s, u)$ 相矛盾，因为这样会导致 $d[u]$ 不大于 $\delta(s, u)$ 。

- 因此， $d[u] = \delta(s, u)$ 。

6. 终止条件：

- 当所有顶点都被加入 S 时，算法结束。
- 此时，对于所有顶点 v ， $d[v] = \delta(s, v)$ 。

结论

通过上述证明，我们可以得出结论：Dijkstra算法能够正确地找到从单个源点到图中所有其他顶点的最短路径。该算法依赖于非负权重边的假设，如果图中存在负权重边，Dijkstra算法可能会给出错误的结果。在这种情况下，可以使用Bellman-Ford算法来处理。

2 进一步解释

Dijkstra 算法的正确性证明基于以下核心逻辑：每次将一个顶点 u 加入已确定最短路径集合 S 时， $d[u]$ 必然等于从起点 s 到该顶点 u 的真实最短路径权值 $\delta(s, u)$ 。以下是如何理解这一证明步骤的关键点：

1. u 的选择保证了它的最小性

- 在算法中，每次选择 u 时，其 $d[u]$ 是所有 $V-S$ 中 d 值最小的。
- 换句话说，在尚未被处理的顶点中， u 是当前最接近起点 s 的顶点。

因此， $d[u] \leq d[v]$ 对于所有 $v \in V - S$ 。

2. 不变量： $d[u] \geq \delta(s, u)$

- 算法的初始化确保了对所有顶点 v ， $d[v]$ 是从起点 s 出发到达该顶点的最短路径的一个上界（初始化时， $d[s] = 0$ ，其余顶点 $d[v] = \infty$ ）。
- 在算法每一步中，通过松弛操作不断缩小 $d[v]$ 的值，但始终保持 $d[v] \geq \delta(s, v)$ 。

3. 假设反证法：如果 $d[u] > \delta(s, u)$

如果 $d[u] > \delta(s, u)$ ，意味着存在更短的路径从 s 到达 u 。设这条路径为 $s \rightarrow w \rightarrow u$ ，其中 $w \in V - S$ 是路径上未处理的某个顶点。

矛盾点分析

- 根据不变量， $d[w] \geq \delta(s, w)$ 。
- 由于 u 是当前 $V-S$ 中 d 最小的顶点，因此 $d[u] \leq d[w]$ 。

- 另一方面，路径 $s \rightarrow w \rightarrow u$ 的真实距离为 $\delta(s, w) + \text{weight}(w, u)$ ，而 $\delta(s, w) + \text{weight}(w, u) \geq \delta(s, u)$ 。

由于我们假设了存在一条更短的路径，即 $d[u] > \delta(s, u)$ ，那么按照Dijkstra算法更新规则， $d[u]$ 应该被更新为 $d[w] + \text{weight}(w, u)$ 或更小的值。这与 $d[u] > \delta(s, u)$ 相矛盾，因为这样会导致 $d[u]$ 不大于 $\delta(s, u)$ 。

- 综合以上推导可知， $d[u] \geq \delta(s, u)$ 。

但 $d[u] > \delta(s, u)$ 的假设与上述结论矛盾。

4. 结论： $d[u] = \delta(s, u)$

由于不存在更短路径未被考虑，因此 $d[u]$ 必等于从 s 到 u 的真实最短路径权值 $\delta(s, u)$ 。

3 直观理解

可以将 Dijkstra 算法看作“逐步揭露最短路径”的过程：

1. 每次处理一个顶点 u ，它已经是离 s 最近的、尚未处理的顶点。
2. 对于 u ，我们确认其最短路径值为 $d[u] = \delta(s, u)$ ，并将其固定在 s 中。
3. 此后更新其邻接顶点的 d 值，使得其他顶点的潜在路径长度不断逼近真实最短路径。

这种逐步扩展的方式确保了算法的正确性。

3 滑动窗口

示例3.无重复字符的最长子串

sliding window, <https://leetcode.cn/problems/longest-substring-without-repeating-characters/>

给定一个字符串 `s`，请你找出其中不含有重复字符的 **最长子串** 的长度。

示例 1:

```
1 输入: s = "abcabcbb"
2 输出: 3
3 解释: 因为无重复字符的最长子串是 "abc", 所以其长度为 3。
```

示例 2:

```
1 输入: s = "bbbbb"
2 输出: 1
3 解释: 因为无重复字符的最长子串是 "b", 所以其长度为 1。
```

示例 3:

```
1 输入: s = "pwwkew"
2 输出: 3
3 解释: 因为无重复字符的最长子串是 "wke", 所以其长度为 3。
4     请注意, 你的答案必须是 子串 的长度, "pwke" 是一个子序列, 不是子串。
```

提示:

- `0 <= s.length <= 5 * 104`
- `s` 由英文字母、数字、符号和空格组成

滑动窗口

是一个队列，比如例题中的 `abcabcbb`，进入这个队列（窗口）为 `abc` 满足题目要求，当再进入 `a`，队列变成了 `abca`，这时候不满足要求。所以，我们要移动这个队列！如何移动？我们只要把队列的左边的元素移出就行了，直到满足题目要求！

一直维持这样的队列，找出队列出现最长的长度时候！时间复杂度： $O(n)$

滑动窗口是一种常用的算法技巧，用于解决数组或字符串中的子数组或子字符串问题。在下面的代码中，滑动窗口的概念体现在通过移动两个指针（起始指针和结束指针）来维护一个当前的无重复子串。

滑动窗口的基本思想

1. 初始化:

- 维护一个窗口 `[start + 1, i]`，表示当前的无重复子串。
- 使用一个字典 `char_index` 来记录每个字符最近一次出现的位置。

2. 扩展窗口:

- 遍历字符串，逐个字符地扩展窗口的右边界 `i`。

3. 收缩窗口:

- 如果当前字符 `c` 在字典中且其上次出现的位置在当前窗口内，则需要收缩窗口的左边界 `start`，使其不包含重复字符。

```
1 class Solution:
2     def lengthOfLongestSubstring(self, s: str) -> int:
3         # 初始化变量
4         start = -1 # 当前无重复子串的起始位置的前一个位置
5         max_length = 0 # 最长无重复子串的长度
6         char_index = {} # 字典，记录每个字符最近一次出现的位置
7
8         # 遍历字符串
9         for i, char in enumerate(s):
10             # 如果字符在字典中且上次出现的位置大于当前无重复子串的起始位置
11             if char in char_index and char_index[char] > start:
12                 # 更新起始位置为该字符上次出现的位置
13                 start = char_index[char]
14
15             # 更新字典中字符的位置
16             char_index[char] = i
17
18             # 计算当前无重复子串的长度，并更新最大长度
19             current_length = i - start
20             max_length = max(max_length, current_length)
21
22         return max_length
```

代码解读

- `k`: 记录当前无重复子串的起始位置的前一个位置，初始值为 -1。
- `res`: 记录最长无重复子串的长度，初始值为 0。
- `c_dict`: 一个字典，用于记录每个字符最近一次出现的位置。

处理字符

```

1         if c in c_dict and c_dict[c] > k: # 字符c在字典中 且 上次出现的下标大于当前长度的起始下标
2             k = c_dict[c]
3             c_dict[c] = i
4         else:
5             c_dict[c] = i
6             res = max(res, i - k)

```

- 条件判断：

- `if c in c_dict and c_dict[c] > k`：检查当前字符 `c` 是否在字典中，并且该字符上次出现的位置是否大于当前无重复子串的起始位置的前一个位置 `k`。
- 如果条件成立，说明当前字符 `c` 在之前的子串中已经出现过，且该位置在当前无重复子串的范围之内，因此需要更新 `k` 为该字符上次出现的位置。
- `k = c_dict[c]`：更新 `k` 为字符 `c` 上次出现的位置。
- `c_dict[c] = i`：更新字典中字符 `c` 的位置为当前索引 `i`。

- 否则：

- `c_dict[c] = i`：更新字典中字符 `c` 的位置为当前索引 `i`。
- `res = max(res, i - k)`：计算当前无重复子串的长度 `i - k`，并更新 `res` 为当前最大值。

4 并查集 (Disjoint Set)

Disjoint Set (Union-Find Algorithm)

<https://www.geeksforgeeks.org/introduction-to-disjoint-set-data-structure-or-union-find-algorithm/>

What is a Disjoint set data structure?

Two sets are called **disjoint sets** if they don't have any element in common, the intersection of sets is a null set.

A data structure that stores non overlapping or disjoint subset of elements is called disjoint set data structure. The disjoint set data structure supports following operations:

- Adding new sets to the disjoint set.
- Merging disjoint sets to a single disjoint set using **Union** operation.
- Finding representative of a disjoint set using **Find** operation.
- Check if two sets are disjoint or not.

Consider a situation with a number of persons and the following tasks to be performed on them:

- Add a **new friendship relation**, i.e. a person x becomes the friend of another person y i.e adding new element to a set.
- Find whether individual **x is a friend of individual y** (direct or indirect friend)

Examples:

We are given 10 individuals say, a, b, c, d, e, f, g, h, i, j

Following are relationships to be added:

a <-> b

b <-> d

c <-> f

c <-> i

j <-> e

g <-> j

Given queries like whether a is a friend of d or not. We basically need to create following 4 groups and maintain a quickly accessible connection among group items:

G1 = {a, b, d}

G2 = {c, f, i}

G3 = {e, g, j}

G4 = {h}

Find whether x and y belong to the same group or not, i.e. to find if x and y are direct/indirect friends.

Partitioning the individuals into different sets according to the groups in which they fall. This method is known as a **Disjoint set Union** which maintains a collection of **Disjoint sets** and each set is represented by one of its members.

To answer the above question two key points to be considered are:

- **How to Resolve sets?** Initially, all elements belong to different sets. After working on the given relations, we select a member as a **representative**. There can be many ways to select a representative, a simple one is to select with the biggest index.
- **Check if 2 persons are in the same group?** If representatives of two individuals are the same, then they'll become friends.

Data Structures used are:

Array: An array of integers is called **Parent[]**. If we are dealing with **N** items, *i*'th element of the array represents the *i*'th item. More precisely, the *i*'th element of the Parent[] array is the parent of the *i*'th item. These relationships create one or more virtual trees.

Tree: It is a **Disjoint set**. If two elements are in the same tree, then they are in the same **Disjoint set**. The root node (or the topmost node) of each tree is called the **representative** of the set. There is always a single **unique representative** of each set. A simple rule to identify a representative is if 'i' is the representative of a set, then **Parent[i] = i**. If i is not the representative of his set, then it can be found by traveling up the tree until we find the representative.

4.1 Operations on Disjoint Set

操作包括 Find 和 Union。

4.1.1 Find

Can be implemented by recursively traversing the parent array until we hit a node that is the parent of itself.

```
1  # Finds the representative of the set
2  # that i is an element of
3
4  def find(i):
5
6      # If i is the parent of itself
7      if (parent[i] == i):
8
9          # Then i is the representative of
10         # this set
11         return i
12     else:
13
14         # Else if i is not the parent of
15         # itself, then i is not the
```



```

16     # representative of his set. So we
17     # recursively call Find on its parent
18     return find(parent[i])
19
20 # The code is contributed by Nidhi goel
21

```

Time complexity: This approach is inefficient and can take $O(n)$ time in worst case.

7.1.2 Union

It takes **two elements** as input and finds the representatives of their sets using the **Find** operation, and finally puts either one of the trees (representing the set) under the root node of the other tree.

```

1  # Unites the set that includes i
2  # and the set that includes j
3
4  def union(parent, rank, i, j):
5      # Find the representatives
6      # (or the root nodes) for the set
7      # that includes i
8      irep = find(parent, i)
9
10     # And do the same for the set
11     # that includes j
12     jrep = find(parent, j)
13
14     # Make the parent of i's representative
15     # be j's representative effectively
16     # moving all of i's set into j's set)
17
18     parent[irep] = jrep
19

```

Time complexity: This approach is inefficient and could lead to tree of length $O(n)$ in worst case.

4.2 Optimizations (Union by Rank/Size and Path Compression)

The efficiency depends heavily on which tree get attached to the other. There are 2 ways in which it can be done. First is Union by Rank, which considers height of the tree as the factor and Second is Union by Size, which considers size of the tree as the factor while attaching one tree to the other . This method along with Path Compression gives complexity of nearly constant time.

4.2.1 Path Compression

Modifications to Find():

It speeds up the data structure by **compressing the height** of the trees. It can be achieved by inserting a small caching mechanism into the **Find** operation. Take a look at the code for more details:

```
1  # Finds the representative of the set that i
2  # is an element of.
3
4
5  def find(i):
6
7      # If i is the parent of itself
8      if Parent[i] == i:
9
10         # Then i is the representative
11         return i
12     else:
13
14         # Recursively find the representative.
15         result = find(Parent[i])
16
17         # We cache the result by moving i's node
18         # directly under the representative of this
19         # set
20         Parent[i] = result
21
22         # And then we return the result
23         return result
24
25 # The code is contributed by Arushi Jindal.
26
```

Time Complexity: $O(\log n)$ on average per call.

4.2.2 Union by Rank

First of all, we need a new array of integers called **rank[]**. The size of this array is the same as the parent array **Parent[]**. If i is a representative of a set, **rank[i]** is the height of the tree representing the set. Now recall that in the Union operation, it doesn't matter which of the two trees is moved under the other. Now what we want to do is minimize the height of the resulting tree. If we are uniting two trees (or sets), let's call them left and right, then it all depends on the **rank of left** and the **rank of right**.

- If the rank of **left** is less than the rank of **right**, then it's best to move **left under right**, because that won't change the rank of right (while moving right under left would increase the height). In the same way, if the rank of right is less than the rank of left, then we should move right under left.
- If the ranks are equal, it doesn't matter which tree goes under the other, but the rank of the result will always be one greater than the rank of the trees.

4.2.3 Path compression and union by rank

Below is the complete implementation of disjoint set with path compression and union by rank.

```
1 class DisjSet:
2     def __init__(self, n):
3         # Constructor to create and initialize sets of n items
4         self.rank = [1] * n
5         self.parent = [i for i in range(n)]
6
7
8     # Finds set of given item x
9     def find(self, x):
10
11         # Finds the representative of the set that x is an element of
12         if (self.parent[x] != x):
13
14             # if x is not the parent of itself
15             # Then x is not the representative of its set
16             self.parent[x] = self.find(self.parent[x])
17
18             # so we recursively call Find on its parent
19             # and move i's node directly under the
20             # representative of this set
21
22         return self.parent[x]
23
24
25     # Do union of two sets represented by x and y.
26     def Union(self, x, y):
27
28         # Find current sets of x and y
29         xset = self.find(x)
30         yset = self.find(y)
31
32         # If they are already in same set
33         if xset == yset:
34             return
35
36         # Put smaller ranked item under
37         # bigger ranked item if ranks are different
38         if self.rank[xset] < self.rank[yset]:
39             self.parent[xset] = yset
40
41         elif self.rank[xset] > self.rank[yset]:
42             self.parent[yset] = xset
43
44         # If ranks are same, then move y under x (doesn't matter
45         # which one goes where) and increment rank of x's tree
46         else:
```

```

47     self.parent[yset] = xset
48     self.rank[xset] = self.rank[xset] + 1
49
50 # Driver code
51 obj = DisjSet(5)
52 obj.Union(0, 2)
53 obj.Union(4, 2)
54 obj.Union(3, 1)
55 if obj.find(4) == obj.find(0):
56     print('Yes')
57 else:
58     print('No')
59 if obj.find(1) == obj.find(0):
60     print('Yes')
61 else:
62     print('No')
63
64
65 """
66 Yes
67 No
68 """

```

Time complexity: $O(n)$ for creating n single item sets. The two techniques -path compression with the union by rank/size, the time complexity will reach nearly constant time. It turns out, that the final [amortized time complexity](#) is $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which grows very steadily (it does not even exceed for $n < 10^{600}$ approximately).

Space complexity: $O(n)$ because we need to store n elements in the Disjoint Set Data Structure.

4.2.4 Union by Size

Again, we need a new array of integers called **size[]**. The size of this array is the same as the parent array **Parent[]**. If i is a representative of a set, **size[i]** is the number of the elements in the tree representing the set.

Now we are uniting two trees (or sets), let's call them left and right, then in this case it all depends on the **size of left** and the **size of right** tree (or set).

- If the size of **left** is less than the size of **right**, then it's best to move **left under right** and increase size of right by size of left. In the same way, if the size of right is less than the size of left, then we should move right under left. and increase size of left by size of right.
- If the sizes are equal, it doesn't matter which tree goes under the other.

```

1 class UnionFind:
2     def __init__(self, n):
3         self.Parent = list(range(n))
4         self.Size = [1] * n

```

```

5
6 # Function to find the representative (or the root node) for the set that includes
i
7 def find(self, i):
8     if self.Parent[i] != i:
9         # Path compression: Make the parent of i the root of the set
10        self.Parent[i] = self.find(self.Parent[i])
11        return self.Parent[i]
12
13 # Unites the set that includes i and the set that includes j by size
14 def unionBySize(self, i, j):
15     # Find the representatives (or the root nodes) for the set that includes i
16     irep = self.find(i)
17
18     # And do the same for the set that includes j
19     jrep = self.find(j)
20
21     # Elements are in the same set, no need to unite anything.
22     if irep == jrep:
23         return
24
25     # Get the size of i's tree
26     isize = self.Size[irep]
27
28     # Get the size of j's tree
29     jsize = self.Size[jrep]
30
31     # If i's size is less than j's size
32     if isize < jsize:
33         # Then move i under j
34         self.Parent[irep] = jrep
35
36         # Increment j's size by i's size
37         self.Size[jrep] += self.Size[irep]
38     # Else if j's size is less than i's size
39     else:
40         # Then move j under i
41         self.Parent[jrep] = irep
42
43         # Increment i's size by j's size
44         self.Size[irep] += self.Size[jrep]
45
46 # Example usage
47 n = 5
48 unionFind = UnionFind(n)
49
50 # Perform union operations
51 unionFind.unionBySize(0, 1)
52 unionFind.unionBySize(2, 3)
53 unionFind.unionBySize(0, 4)
54
55 # Print the representative of each element after unions

```

```

56 for i in range(n):
57     print("Element {}: Representative = {}".format(i, unionFind.find(i)))
58
59 # This code is contributed by Susobhan Akhuli
60
61 """
62 Element 0: Representative = 0
63 Element 1: Representative = 0
64 Element 2: Representative = 2
65 Element 3: Representative = 2
66 Element 4: Representative = 0
67 """

```

Time complexity: $O(\log n)$ without Path Compression.

4.3 编程题目

示例晴问9.6.1 学校的班级个数 (1)

<https://sunnywhy.com/sfbj/9/6/360>

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生 A 和学生 B 处于一个班级，学生 B 和学生 C 处于一个班级，那么我们称学生 A 和学生 C 也处于一个班级。

现已知学校中共 n 个学生（编号为从 1 到 n ），并给出 m 组学生关系（指定两个学生处于一个班级），问总共有多少个班级。

输入

第一行两个整数 m 、 n ($1 \leq n \leq 100, 1 \leq m \leq 100$)，分别表示学生个数、学生关系个数；

接下来 m 行，每行两个整数 a 和 b ($1 \leq a \leq n, 1 \leq b \leq n, a \neq b$)，表示编号为 a 的学生和编号为 b 的学生处于一个班级。

输出

输出一个整数，表示班级个数。

样例1

输入

```

1 5 3
2 4 2
3 1 3
4 2 5

```

输出

解释

编号 2、4、5 的学生在同一个班级，编号 1、3 的学生在同一个班级，因此共有两个班级。

To solve this problem, you can use the concept of disjoint sets (also known as union-find). Each student is initially in their own set (or class). When a relationship between two students is given, you unite their sets. The number of disjoint sets at the end will be the number of classes.

Here is a step-by-step plan:

1. Initialize an array `parent` of size `n+1` where `parent[i]` is initially `i`. This means each student is in their own class.
2. For each student relationship, unite their sets using the union operation.
3. Count the number of unique parents. This will be the number of classes.

Here is the Python code that implements this plan:

```

1  def find(x):
2      if parent[x] != x: # 如果不是根结点，继续循环
3          parent[x] = find(parent[x])
4      return parent[x]
5
6  def union(x, y):
7      parent[find(x)] = find(y)
8
9  n, m = map(int, input().split())
10 parent = list(range(n + 1)) # parent[i] == i, 则说明元素i是该集合的根结点
11
12 for _ in range(m):
13     a, b = map(int, input().split())
14     union(a, b)
15
16 classes = set(find(x) for x in range(1, n + 1))
17 print(len(classes))

```

This code reads the number of students and relationships from the input, initializes each student to be in their own class, unites the classes of each pair of students that have a relationship, and then counts and prints the number of unique classes.

示例晴问9.6.2 学校的班级人数 (2)

<https://sunnywhy.com/sfbj/9/6/361>

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生 A 和学生 B 处于一个班级，学生 B 和学生 C 处于一个班级，那么我们称学生 A 和学生 C 也处于一个班级。

现已知学校中共 n 个学生（编号为从 1 到 n），并给出 m 组学生关系（指定两个学生处于一个班级），问总共有多少个班级，并按降序给出每个班级的人数。

输入

第一行两个整数 m 、 n ($1 \leq n \leq 100, 1 \leq m \leq 100$)，分别表示学生个数、学生关系个数；

接下来 m 行，每行两个整数 a 和 b ($1 \leq a \leq n, 1 \leq b \leq n, a \neq b$)，表示编号为 a 的学生和编号为 b 的学生处于一个班级。

输出

第一行输出一个整数，表示班级个数；

第二行若干个整数，按降序给出每个班级的人数。整数之间用空格隔开，行末不允许有多余的空格。

样例1

输入

1	5	3
2	4	2
3	1	3
4	2	5

输出

1	2
2	3 2

解释

编号 2、4、5 的学生在同一个班级，编号 1、3 的学生在同一个班级，因此共有两个班级，人数分别是 3 和 2。

To solve this problem, you can use the concept of disjoint sets (also known as union-find). Each student is initially in their own set (or class). When a relationship between two students is given, you unite their sets. The number of disjoint sets at the end will be the number of classes. You can also maintain a count of the number of students in each class.

Here is a step-by-step plan:

1. Initialize an array `parent` of size `n+1` where `parent[i]` is initially `i`. This means each student is in their own class.
2. Initialize an array `size` of size `n+1` where `size[i]` is initially `1`. This means each class initially has one student.

3. For each student relationship, unite their sets using the union operation and update the size of the new set.
4. Count the number of unique parents. This will be the number of classes.
5. Print the sizes of the classes in descending order.

Here is the Python code that implements this plan:

```
1 def find(x):
2     if parent[x] != x:
3         parent[x] = find(parent[x])
4     return parent[x]
5
6 def union(x, y):
7     root_x = find(x)
8     root_y = find(y)
9     if root_x != root_y:
10        parent[root_x] = root_y
11        size[root_y] += size[root_x]
12
13 n, m = map(int, input().split())
14 parent = list(range(n + 1))
15 size = [1] * (n + 1)
16
17 for _ in range(m):
18     a, b = map(int, input().split())
19     union(a, b)
20
21 #classes = [size[find(x)] for x in range(1, n + 1) if x == parent[x]]
22 classes = [size[x] for x in range(1, n + 1) if x == parent[x]]
23 print(len(classes))
24 print(' '.join(map(str, sorted(classes, reverse=True))))
```

This code reads the number of students and relationships from the input, initializes each student to be in their own class, unites the classes of each pair of students that have a relationship, counts and prints the number of unique classes, and prints the sizes of the classes in descending order.

示例问题9.6.3 是否相同班级

<https://sunnywhy.com/sfbj/9/6/362>

现有一个学校，学校中有若干个班级，每个班级中有若干个学生，每个学生只会存在于一个班级中。如果学生A和学生B处于一个班级，学生B和学生C处于一个班级，那么我们称学生A和学生C也处于一个班级。

现已知学校中共n个学生（编号为从1到n），并给出m组学生关系（指定两个学生处于一个班级）。然后给出k个查询，每个查询询问两个学生是否在同一个班级。

输入

第一行两个整数 n 、 m ($1 \leq n \leq 10^5, 1 \leq m \leq 10^5$), 分别表示学生个数、学生关系个数;

接下来 m 行, 每行两个整数 a 和 b ($1 \leq a \leq n, 1 \leq b \leq n, a \neq b$), 表示编号为 a 的学生和编号为 b 的学生处于一个班级。

然后一个整数 k ($1 \leq k \leq 10^3$), 表示查询个数;

接下来 k 行, 每行两个整数 a 和 b ($1 \leq a \leq n, 1 \leq b \leq n$), 表示询问编号为 a 的学生和编号为 b 的学生是否在同一个班级。

输出

每个查询输出一行, 如果在同一个班级, 那么输出 `Yes`, 否则输出 `No`。

样例1

输入

```
1 5 3
2 4 2
3 1 3
4 2 5
5 2
6 4 5
7 1 2
```

输出

```
1 Yes
2 No
```

解释

编号 2、4、5 的学生在同一个班级, 编号 1、3 的学生在同一个班级, 因此编号 4 和 5 的学生在同一个班级, 编号 1 和 2 的学生不在同一个班级。

To solve this problem, you can use the concept of disjoint sets (also known as union-find). Each student is initially in their own set (or class). When a relationship between two students is given, you unite their sets. For each query, you check if the two students are in the same set.

Here is a step-by-step plan:

1. Initialize an array `parent` of size `n+1` where `parent[i]` is initially `i`. This means each student is in their own class.
2. For each student relationship, unite their sets using the union operation.
3. For each query, check if the two students are in the same set using the find operation.

Here is the Python code that implements this plan:

```

1 def find(x):
2     if parent[x] != x:
3         parent[x] = find(parent[x])
4     return parent[x]
5
6 def union(x, y):
7     parent[find(x)] = find(y)
8
9 n, m = map(int, input().split())
10 parent = list(range(n + 1))
11
12 for _ in range(m):
13     a, b = map(int, input().split())
14     union(a, b)
15
16 k = int(input())
17 for _ in range(k):
18     a, b = map(int, input().split())
19     if find(a) == find(b):
20         print('Yes')
21     else:
22         print('No')

```

This code reads the number of students and relationships from the input, initializes each student to be in their own class, unites the classes of each pair of students that have a relationship, and then for each query, checks and prints whether the two students are in the same class.

示例晴问9.6.4 迷宫连通性

<https://sunnywhy.com/sfbj/9/6/363>

现有一个迷宫，迷宫中有 n 个房间（编号为从 1 到 n ），房间与房间之间可能连通。如果房间 A 和房间 B 连通，房间 B 和房间 C 连通，那么我们称房间 A 和房间 C 也连通。给定 m 组连通关系（指定两个房间连通），问迷宫中的所有房间是否连通。

输入

第一行两个整数 n 、 m ($1 \leq n \leq 100, 1 \leq m \leq 100$)，分别表示房间个数、连通关系个数；

接下来行，每行两个整数 a 和 b ($1 \leq a \leq n, 1 \leq b \leq n$)，表示编号为 a 的房间和编号为 b 的房间是连通的。

输出

如果所有房间连通，那么输出 Yes，否则输出 No。

样例1

输入

```
1 5 4
2 4 2
3 1 3
4 2 5
5 1 5
```

输出

```
1 Yes
```

解释

所有房间都连通，因此输出 `Yes`。

样例2

输入

```
1 5 3
2 4 2
3 1 3
4 2 5
```

输出

```
1 No
```

解释

编号 `2`、`4`、`5` 的房间互相连通，编号 `1`、`3` 的房间互相连通，因此没有全部互相连通，输出 `No`。

To solve this problem, you can use the concept of disjoint sets (also known as union-find). Each room is initially in its own set. When a connection between two rooms is given, you unite their sets. If at the end there is only one set, then all rooms are connected.

Here is a step-by-step plan:

1. Initialize an array `parent` of size `n+1` where `parent[i]` is initially `i`. This means each room is in its own set.
2. For each connection, unite their sets using the union operation.
3. Check if all rooms are in the same set.

Here is the Python code that implements this plan:

```
1 def find(x):
2     if parent[x] != x:
3         parent[x] = find(parent[x])
4     return parent[x]
```

```

5
6 def union(x, y):
7     parent[find(x)] = find(y)
8
9 n, m = map(int, input().split())
10 parent = list(range(n + 1))
11
12 for _ in range(m):
13     a, b = map(int, input().split())
14     union(a, b)
15
16 sets = set(find(x) for x in range(1, n + 1))
17 if len(sets) == 1:
18     print('Yes')
19 else:
20     print('No')

```

This code reads the number of rooms and connections from the input, initializes each room to be in its own set, unites the sets of each pair of rooms that have a connection, and then checks and prints whether all rooms are in the same set.

5 辅助栈、懒删除

示例22067: 快速堆猪

辅助栈, <http://cs101.openjudge.cn/practice/22067/>

小明有很多猪, 他喜欢玩叠猪游戏, 就是将猪一头头叠起来。猪叠上去后, 还可以把顶上的猪拿下来。小明知道每头猪的重量, 而且他还随时想知道叠在那里的猪最轻的是多少斤。

输入

有三种输入

1. push n
n是整数($0 \leq n \leq 20000$), 表示叠上一头重量是n斤的新猪
2. pop
表示将猪堆顶的猪赶走。如果猪堆没猪, 就啥也不干
3. min
表示问现在猪堆里最轻的猪多重。如果猪堆没猪, 就啥也不干

输入总数不超过100000条

输出

对每个min输入, 输出答案。如果猪堆没猪, 就啥也不干

样例输入

```
1 pop
2 min
3 push 5
4 push 2
5 push 3
6 min
7 push 4
8 min
```

样例输出

```
1 2
2 2
```

来源: Guo wei

用辅助栈: 用一个单调栈维护最小值, 再用另外一个栈维护其余的值。

每次push时, 在辅助栈中加入当前最轻的猪的体重, pop时也同步pop, 这样栈顶始终是当前猪堆中最轻的体重, 查询时直接输出即可

```

1  a = []
2  m = []
3
4  while True:
5      try:
6          s = input().split()
7
8          if s[0] == "pop":
9              if a:
10                 a.pop()
11                 if m:
12                     m.pop()
13             elif s[0] == "min":
14                 if m:
15                     print(m[-1])
16             else:
17                 h = int(s[1])
18                 a.append(h)
19                 if not m:
20                     m.append(h)
21                 else:
22                     k = m[-1]
23                     m.append(min(k, h))
24         except EOFError:
25             break

```

```

1  pig, pigmin = [], []
2  while True:
3      try:
4          *line, = input().split()
5          if "pop" in line:
6              if len(pig) == 0:
7                  continue
8
9              val = pig.pop()
10             if len(pigmin) > 0 and val == pigmin[-1]:
11                 pigmin.pop()
12         elif "push" in line:
13             val = int(line[1])
14             pig.append(val)
15             if len(pigmin) == 0 or val <= pigmin[-1]:
16                 pigmin.append(val)
17         elif "min" in line:
18             if len(pig) == 0:
19                 continue
20             else:
21                 print(pigmin[-1])
22     except EOFError:
23         break

```

字典标记, 懒删除

```
1 import heapq
2 from collections import defaultdict
3
4 out = defaultdict(int)
5 pigs_heap = []
6 pigs_stack = []
7
8 while True:
9     try:
10         s = input()
11     except EOFError:
12         break
13
14     if s == "pop":
15         if pigs_stack:
16             out[pigs_stack.pop()] += 1
17     elif s == "min":
18         if pigs_stack:
19             while True:
20                 x = heapq.heappop(pigs_heap)
21                 if not out[x]:
22                     heapq.heappush(pigs_heap, x)
23                     print(x)
24                     break
25             out[x] -= 1
26     else:
27         y = int(s.split()[1])
28         pigs_stack.append(y)
29         heapq.heappush(pigs_heap, y)
```

集合标记, 懒删除。如果有重复项就麻烦了, 可能刚好赶上题目数据友好。

```
1 import heapq
2
3 class PigStack:
4     def __init__(self):
5         self.stack = []
6         self.min_heap = []
7         self.popped = set()
8
9     def push(self, weight):
10         self.stack.append(weight)
11         heapq.heappush(self.min_heap, weight)
12
13     def pop(self):
14         if self.stack:
```



```

15         weight = self.stack.pop()
16         self.popped.add(weight)
17
18     def min(self):
19         while self.min_heap and self.min_heap[0] in self.popped:
20             self.popped.remove(heapq.heappop(self.min_heap))
21         if self.min_heap:
22             return self.min_heap[0]
23         else:
24             return None
25
26 pig_stack = PigStack()
27
28 while True:
29     try:
30         command = input().split()
31         if command[0] == 'push':
32             pig_stack.push(int(command[1]))
33         elif command[0] == 'pop':
34             pig_stack.pop()
35         elif command[0] == 'min':
36             min_weight = pig_stack.min()
37             if min_weight is not None:
38                 print(min_weight)
39     except EOFError:
40         break

```

6 单调栈

示例42.接雨水

monotonic stack, <https://leetcode.cn/problems/trapping-rain-water/>

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

示例 1:



```
1 输入: height = [0,1,0,2,1,0,1,3,2,1,2,1]
2 输出: 6
3 解释: 上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。
```

示例 2:

```
1 输入: height = [4,2,0,3,2,5]
2 输出: 9
```

提示:

- $n == \text{height.length}$
- $1 \leq n \leq 2 * 10^4$
- $0 \leq \text{height}[i] \leq 10^5$

单调栈其实就是在栈的基础上，维持一个栈内元素单调。

<https://github.com/SharingSource/LogicStack-LeetCode>

在这道题，由于需要找某个位置两侧比其高的柱子（只有两侧有比当前位置高的柱子，当前位置才能接下雨水），我们可以维持栈内元素的单调递减。

PS.找某侧最近一个比其大的值，使用单调栈维持栈内元素递减；找某侧最近一个比其小的值使用单调栈维持栈内元素递增

当某个位置的元素弹出栈时，例如位置 `a`，我们自然可以得到 `a` 位置两侧比 `a` 高的柱子：

- 一个是导致 `a` 位置元素弹出的柱子(`a` 右侧比 `a` 高的柱子)
- 一个是 `a` 弹出后的栈顶元素(`a` 左侧比 `a` 高的柱子)

当有了 `a` 左右两侧比 `a` 高的柱子后，便可计算 `a` 位置可接下的雨水量。

```
1 class Solution:
2     def trap(self, height: List[int]) -> int:
3         stack = []
4         water = 0
5         for i in range(len(height)):
6             while stack and height[i] > height[stack[-1]]:
7                 top = stack.pop()
8                 if not stack:
9                     break
10                distance = i - stack[-1] - 1
11                bounded_height = min(height[i], height[stack[-1]]) - height[top]
12                water += distance * bounded_height
13            stack.append(i)
14        return water
```

这段代码实现了一个算法，用于计算给定高度数组 `height` 中可以 trapping 的雨水总量。这个算法使用了栈来高效地解决这个问题。

代码解读

处理栈中的元素

```
1 while stack and height[i] > height[stack[-1]]:
```

- 当栈不为空且当前高度 `height[i]` 大于栈顶元素对应的高度 `height[stack[-1]]` 时，进入循环。

弹出栈顶元素

```
1 top = stack.pop()
```

- 弹出栈顶元素 `top`，`top` 是当前高度较低的柱子的索引。

检查栈是否为空

```
1 if not stack:
2     break
```

- 如果栈为空，说明没有更高的柱子可以形成积水区域，跳出循环。

计算积水区域

```

1         distance = i - stack[-1] - 1
2         bounded_height = min(height[i], height[stack[-1]]) - height[top]
3         water += distance * bounded_height

```

- `distance`：计算当前柱子 `i` 和栈顶柱子 `stack[-1]` 之间的距离，减去 1 是因为不包括两端的柱子。
- `bounded_height`：计算当前柱子 `i` 和栈顶柱子 `stack[-1]` 之间的最小高度，减去弹出的柱子 `top` 的高度，得到积水的高度。
- `water += distance * bounded_height`：计算当前积水区域的水量，并累加到 `water` 中。

双指针

```

1 class Solution:
2     def trap(self, height: List[int]) -> int:
3         ans = left = pre_max = suf_max = 0
4         right = len(height) - 1
5         while left < right:
6             pre_max = max(pre_max, height[left])
7             suf_max = max(suf_max, height[right])
8             if pre_max < suf_max:
9                 ans += pre_max - height[left]
10                left += 1
11            else:
12                ans += suf_max - height[right]
13                right -= 1
14        return ans
15

```

单调栈相当于按行接雨水，双指针相当于按列接雨水？

确实，单调栈和双指针方法在处理“接雨水”问题时采用了不同的思路，但最终都能正确计算出能够接住的雨水总量。下面简要分析这两种方法的工作原理：

单调栈（按行接雨水）

单调栈方法通过维护一个递减栈来找到每个位置左侧和右侧的第一个更高柱子，进而计算该位置上方能接住的雨水量。这种方法可以看作是逐行计算雨水量，因为每次从栈中弹出一个元素时，实际上是在计算该元素上方的水平层（即一行）的雨水量。具体来说，当遇到一个比栈顶元素高的柱子时，就找到了一个可以蓄水的区域，然后根据左右边界的高度差和距离计算出该行的雨水量。

双指针（按列接雨水）

双指针方法则是通过两个指针从数组的两端向中间移动，同时记录左右两边的最大高度。在每一步中，选择较短的一边进行处理，因为水位总是由较短的一边决定的。如果左边高度小于右边高度，那么左边当前柱子上方能接住的雨水量就是左边最大高度减去当前高度；反之亦然。这种方法可以看作是逐列计算雨水量，因为每次移动指针时，实际上是在计算当前指针所指向柱子上方的雨水量。

比较

- **单调栈** 更适理解解为按照行来计算雨水量，因为它关注的是每个局部凹陷处（即两根较高柱子之间的一根或几根较低柱子）的雨水量，这些凹陷处可以想象成一个个水平的水层。
- **双指针** 则更适理解解为按照列来计算雨水量，因为它直接计算每个柱子上方能接住的雨水量，而不需要显式地找出每个凹陷处。

两种方法虽然计算方式不同，但是都能有效地解决问题，并且时间复杂度都是 $O(n)$ ，其中 n 是高度数组的长度。空间复杂度方面，单调栈方法需要额外的空间来存储栈，而双指针方法只需要常数级别的额外空间。

7 概念关联理解

马拉车算法

马拉车算法与KMP算法有异曲同工之效

马拉车算法（Manacher's Algorithm）和KMP算法（Knuth-Morris-Pratt Algorithm）都是字符串处理中的经典算法，但它们解决的问题不同，尽管在某些方面有相似之处。

马拉车算法

- **用途：**用于查找一个字符串中最长的回文子串。
- **核心思想：**利用已知的回文信息来避免不必要的重复计算。通过动态规划的思想，它可以在 $O(n)$ 的时间复杂度内完成任务。
- **特点：**
 - 通过中心扩展法寻找回文，但巧妙地利用了已经找到的回文信息来减少不必要的比较。
 - 使用一个辅助数组来存储每个位置作为中心时的最长回文半径长度。
 - 通过对称性以及最右边界来优化查找过程，从而达到线性时间复杂度。

KMP算法

- **用途：**用于在一个文本中查找某个模式串是否存在，并返回其起始位置。
- **核心思想：**利用模式串本身的结构信息来避免不必要的字符比较，即构建一个部分匹配表（也称为“失败函数”或“前缀函数”），这个表记录了模式串的前缀和后缀重合的信息。
- **特点：**
 - 通过预处理模式串，构建出一个能够快速跳过已知不匹配部分的表。
 - 在搜索过程中，如果遇到不匹配的情况，可以根据部分匹配表直接移动到下一个可能匹配的位置，而不是从头开始重新匹配。
 - 时间复杂度为 $O(n + m)$ ，其中 n 是文本串的长度， m 是模式串的长度。

异曲同工之效

- **预处理：**两者都通过某种形式的预处理来加速后续的操作。马拉车算法通过预处理辅助数组来加速回文子串的查找；KMP算法则通过预处理部分匹配表来加速模式匹配。
- **避免重复工作：**两种算法都试图利用已有信息来避免重复的工作，从而提高效率。马拉车算法利用已知的回文信息来减少不必要的中心扩展；KMP算法利用模式串的部分匹配信息来避免不必要的字符比较。
- **线性时间复杂度：**两者都可以在线性时间内完成各自的任务，这使得它们在处理大规模数据时非常高效。

虽然马拉车算法和KMP算法解决的是不同的问题，但它们都体现了利用已有信息来优化算法执行过程的思想，这是它们共有的“异曲同工之效”。

马拉车算法是双指针+dp

马拉车算法（Manacher's Algorithm）是一种用于寻找字符串中最长回文子串的高效算法，其时间复杂度为 $O(n)$ 。这个算法巧妙地结合了双指针技术和动态规划的思想，但并不直接使用传统的动态规划表来解决问题。

双指针

- **中心扩展法**：基本思想是从每一个字符开始，尝试向两边扩展，检查是否构成回文。但是直接这样做会导致效率低下。
- **优化后的双指针**：马拉车算法通过一些技巧避免了重复计算。它利用已经处理过的部分信息来加速后续的查找过程。

动态规划的思想

- **状态定义**：在传统动态规划方法中，我们会定义一个二维数组 $dp[i][j]$ 表示从 i 到 j 的子串是否为回文。但对于马拉车算法来说，我们并不直接使用这样的二维数组。
- **状态转移**：虽然没有显式地构建 dp 数组，但马拉车算法隐含地使用了之前计算过的信息来决定当前可以跳过多少个不需要检查的位置，这实际上是一种基于已知信息推导新信息的过程，类似于动态规划中的状态转移。

马拉车算法的关键点

1. **预处理**：为了处理奇数长度和偶数长度的回文问题，通常会在每个字符之间插入一个特殊符号（如#），这样所有可能的回文都会变成奇数长度。
2. **P数组**：定义一个数组 P ，其中 $P[i]$ 表示以位置 i 为中心的最大回文半径。这个数组用来记录每个位置作为中心时的最长回文子串信息。
3. **中心和右边界**：维护当前找到的最远的回文子串的中心 C 以及对应的最右边界的 R 。这些信息帮助我们快速跳过不必要的检查。
4. **镜像对称性**：对于每一个新的中心，如果它位于当前右边界 R 内，那么我们可以利用它的对称位置的 P 值来减少不必要的比较。

综上所述，马拉车算法确实结合了双指针的方法来进行高效的回文判断，并且利用了之前计算的结果来避免重复工作，这体现了动态规划的核心思想之一——即通过保存并利用之前的状态来优化当前问题的解决过程。因此，描述马拉车算法为“双指针+dp”是合理的，这里的dp指的是动态规划的思想而非具体的dp数组实现。

单调栈和Greedy关系

单调栈（Monotonic Stack）和贪心算法（Greedy Algorithm）是两种不同的数据结构和算法，它们在解决特定类型问题时有着各自的特点和应用场景。

单调栈

- **定义**：单调栈是一种特殊的栈，它保持栈内的元素按照某种顺序（递增或递减）排列。当新元素入栈时，如果破坏了这种顺序，则会弹出栈顶元素，直到可以保持顺序为止。
- **用途**：单调栈常用于解决涉及数组或序列的问题，尤其是那些需要找到某个元素的下一个更大/更小元素、或者需要维护一个窗口内最大/最小值等问题。

- **特点：**通过维护一个有序的栈，可以在 $O(1)$ 时间内获取某些特定的信息，从而使得整体时间复杂度降低到 $O(n)$ 。

贪心算法

- **定义：**贪心算法是一种在每一步选择中都采取当前状态下最好或最优的选择，从而希望导致结果是全局最好或最优的算法。
- **用途：**适用于那些可以通过局部最优解来达到全局最优解的问题。例如，找零钱问题、活动选择问题等。
- **特点：**贪心算法通常简单直观，但并不总是能够保证得到全局最优解。对于某些问题，贪心算法可能只提供近似解而非精确解。

两者之间的关系

虽然单调栈和贪心算法都可以用来优化问题的解决方案，但它们并不是同一类技术。单调栈更多地被看作是一种数据结构，而贪心算法则是一种解决问题的策略。在某些情况下，你可能会结合使用这两种技术来解决一个问题，但它们本身代表了不同的概念：

- **单调栈：**关注的是如何高效地维护和访问一组有序的数据。
- **贪心算法：**关注的是如何通过一系列局部最优决策来尝试达到全局最优。

因此，不能简单地说单调栈就是贪心算法。它们在解决问题时扮演着不同的角色，并且有各自的适用场景。

树状数组和线段树关系

树状数组（Binary Indexed Tree, BIT）和线段树（Segment Tree）虽然都是用于高效处理区间查询和更新的数据结构，但它们在实现方式、空间复杂度以及某些操作的效率上有所不同。下面是对这两种数据结构的简要比较：

树状数组 (Binary Indexed Tree, BIT)

- **定义：**树状数组是一种可以高效地进行前缀和计算以及单点更新的数据结构。
- **用途：**主要用于解决动态范围求和问题，如区间求和、频繁的点更新等。
- **特点：**
 - 空间复杂度为 $O(n)$ ，与输入数组大小相同。
 - 支持快速的单点更新和前缀和查询，时间复杂度均为 $O(\log n)$ 。
 - 实现相对简单，易于理解和编码。
 - 不支持区间更新（除非使用更复杂的变种）。

线段树 (Segment Tree)

- **定义：**线段树是一种二叉树结构，每个节点代表一个区间，通常用于处理区间查询和更新。
- **用途：**适用于多种区间操作，包括区间求和、区间最大/最小值、区间更新等。
- **特点：**
 - 空间复杂度为 $O(4n)$ 或 $O(2n)$ ，因为每个节点可能需要额外的空间来存储信息。
 - 支持快速的区间查询和更新，时间复杂度通常为 $O(\log n)$ 。

- 可以处理更复杂的区间操作，如区间更新和延迟传播（Lazy Propagation）。
- 实现相对复杂，需要更多的代码量。

区别

- **功能：**线段树比树状数组功能更强大，能够处理更多类型的区间操作。
- **空间复杂度：**树状数组的空间复杂度更低，而线段树则需要更多的空间。
- **实现难度：**树状数组的实现较为简单，而线段树的实现相对复杂，特别是当涉及到懒惰标记（Lazy Propagation）时。
- **适用场景：**如果只需要处理前缀和或者简单的点更新，树状数组是更好的选择；如果需要处理复杂的区间操作，尤其是区间更新，线段树通常是更好的选择。

总结来说，树状数组和线段树不是同一种数据结构。尽管它们都用于处理区间相关的查询和更新问题，但它们各自有其优势和适用场景。在实际应用中，根据具体需求选择合适的数据结构是很重要的。

递归和算法关系

递归（Recursion）本身是一种编程技术或方法，而不是一种具体的算法。它是一种解决问题的策略，通过将问题分解为更小的、相似的子问题来求解。递归通常涉及一个函数直接或间接地调用自身。

递归的关键特性：

1. **基准情况（Base Case）：**这是递归终止的条件，确保递归不会无限进行下去。
2. **递归步骤（Recursive Step）：**在这一部分，函数会调用自身，但每次调用都会处理规模更小的问题。

递归的应用

递归可以用于实现各种算法，例如：

- **排序算法：**如快速排序（Quick Sort）和归并排序（Merge Sort）。
- **搜索算法：**如深度优先搜索（DFS）、回溯算法等。
- **动态规划：**某些动态规划问题可以通过递归来解决，尽管通常会使用记忆化（Memoization）来优化性能。
- **数学问题：**计算阶乘、斐波那契数列等。

递归与算法的关系

虽然递归不是一种特定的算法，但它是一种强大的工具，可以用来构建算法。很多经典算法都是基于递归思想设计的。例如：

- **快速排序：**选择一个基准元素，将数组分成两部分，一部分小于基准，另一部分大于基准，然后对这两部分分别递归排序。
- **归并排序：**将数组分成两个子数组，分别对它们进行排序，然后再合并两个有序的子数组。
- **二叉树遍历：**前序遍历、中序遍历、后序遍历等都可以通过递归方式简洁地实现。

递归的优点

- **代码简洁**：递归可以使代码更加简洁易懂。
- **逻辑清晰**：对于一些问题，递归能够自然地表达问题的结构。

递归的缺点

- **性能问题**：递归可能导致大量的函数调用开销，尤其是在没有优化的情况下（如尾递归优化）。
- **栈溢出**：如果递归层次太深，可能会导致栈溢出错误。

递归的优化

- **尾递归优化**：在某些编程语言中，编译器或解释器可以优化尾递归，使其不占用额外的栈空间。
- **记忆化**：存储已经计算过的结果，避免重复计算，提高效率。

总之，递归是一种重要的编程技术和解决问题的方法，它可以用来实现多种高效的算法。理解递归的思想对于掌握许多高级算法和数据结构非常有帮助。

双指针和Greedy关系

双指针（Two Pointers）是一种编程技巧，而不是一种具体的算法。它通常用于解决数组或链表中的问题，特别是在需要同时处理两个位置的数据时。双指针方法可以有效地减少时间复杂度，提高程序的效率。

双指针的特点

- **定义**：双指针是指在遍历数据结构时使用两个指针来追踪不同的位置。
- **用途**：常用于数组、链表等线性数据结构中，以优化查找、排序、合并等问题。
- **优点**：
 - 可以在一次遍历中完成多个操作。
 - 通常能将时间复杂度从 $O(n^2)$ 降低到 $O(n)$ 。

常见应用场景

1. **寻找两数之和**：例如，在一个有序数组中找到两个数使它们的和等于给定的目标值。
2. **合并两个有序数组**：例如，合并两个已排序的数组。
3. **删除重复元素**：例如，在一个数组中移除重复的元素。
4. **反转字符串**：例如，反转一个字符串中的字符。
5. **滑动窗口**：例如，在一个数组中找到满足特定条件的最长子数组。

双指针与贪心算法的关系

虽然双指针是一种编程技巧，但它可以被用来实现某些贪心算法。贪心算法是一种在每一步选择中都采取当前状态下最好或最优的选择，从而希望导致结果是全局最好或最优的算法。双指针可以在某些情况下帮助实现这种局部最优选择，从而达到全局最优解。

示例

假设我们要在一个有序数组中找到两个数，使它们的和等于给定的目标值。我们可以使用双指针方法：

```
1 def two_sum(nums, target):
2     left, right = 0, len(nums) - 1
3     while left < right:
4         current_sum = nums[left] + nums[right]
5         if current_sum == target:
6             return [left, right]
7         elif current_sum < target:
8             left += 1
9         else:
10            right -= 1
11    return [-1, -1]
12
13 # 示例
14 nums = [1, 2, 3, 4, 6]
15 target = 6
16 print(two_sum(nums, target)) # 输出: [1, 3]
```

在这个例子中，双指针方法通过同时移动左右指针来高效地找到目标值。

总结

- 双指针是一种编程技巧，用于优化数组或链表的操作。
- 贪心算法是一种解决问题的策略，通过局部最优选择来尝试达到全局最优解。
- 双指针可以作为一种实现贪心算法的方法，但双指针本身并不局限于贪心算法，它可以用于多种类型的算法和问题。

因此，双指针更多地被视为一种编程技巧，而不是一种具体的算法实现方式。它在很多场景下都能提高代码的效率和简洁性。

回溯、递归和深搜关系

回溯（Backtracking）是一种算法技术，它通常用于解决那些需要探索所有可能解空间的问题。回溯算法本质上是递归的，并且经常使用深度优先搜索（DFS, Depth-First Search）来实现。因此，回溯可以被看作是递归和搜索的一种结合。

回溯与递归

- 递归：递归是一种编程技术，通过函数调用自身来解决问题。递归的关键在于定义一个基准情况（Base Case），以及如何将问题分解为更小的子问题。
- 回溯：回溯是一种系统地搜索问题解空间的方法。它通过尝试每一种可能的选择，如果发现当前选择不能得到解，则撤销该选择（即“回溯”），并尝试下一个选择。回溯通常是通过递归来实现的，因为递归提供了一种自然的方式来处理子问题。

回溯与深度优先搜索（DFS）

- **深度优先搜索（DFS）**：DFS是一种用于遍历或搜索树或图的算法。它从根节点开始，尽可能深地搜索每个分支，直到无法继续为止，然后回溯到上一个节点，继续搜索其他分支。
- **回溯**：回溯算法在很多情况下使用DFS来实现，因为它需要系统地探索所有可能的解路径。回溯中的“回溯”步骤实际上就是DFS中的回溯过程。

归类

1. **递归**：回溯算法本质上是递归的，因为它通过递归调用来探索解空间。因此，回溯可以被视为递归的一种应用。
2. **搜索**：回溯也可以被视为一种搜索方法，特别是当它使用DFS来实现时。回溯算法通过DFS的方式系统地探索解空间，找到所有可能的解。

总结

- **回溯**：既可以归类为递归，也可以归类为搜索中的深度优先搜索。
- **递归**：是回溯的基础，提供了实现回溯的技术手段。
- **深度优先搜索（DFS）**：是回溯中常用的具体搜索策略，用于系统地探索解空间。

因此，回溯可以同时归类到递归和搜索中的深度优先搜索。具体归类取决于你关注的是它的实现方式（递归）还是它的搜索策略（DFS）。在实际应用中，这两种视角都是正确的，而且它们是相辅相成的。

并查集

并查集（Disjoint Set Union, DSU），也被称为Union-Find结构，是一种用于处理一些不相交集合并及查询问题的数据结构。它支持两种主要操作：查找（Find）和合并（Union）。这种数据结构在解决图论中的连通性问题时非常有用，比如判断两个节点是否属于同一个连通块、将两个连通块合并成一个等。

基本概念

- **集合**：并查集中的每个元素都属于某个集合。
- **代表元**：每个集合都有一个代表元素，用来唯一标识这个集合。
- **路径压缩**：一种优化手段，通过让查找路径上的所有节点直接指向根节点来减少树的高度，从而加速后续的查找操作。
- **按秩合并**：另一种优化手段，在合并两个集合时总是把较小深度的树挂到较大深度的树上，以保持树的平衡。

主要操作

1. **初始化**：为每个元素创建一个单独的集合，此时每个元素都是自己所在集合的代表。
2. **查找 (Find)**：确定某元素所属的集合，并返回该集合的代表元素。在此过程中可以实施路径压缩。
3. **合并 (Union)**：将两个元素所在的集合合并成一个新的集合。如果两个元素已经在同一个集合中，则无需执行任何操作。此过程可利用按秩合并进行优化。

实现细节

- 使用数组或者哈希表来存储每个元素的父节点指针以及集合的大小或秩信息。

- 初始化时，每个元素的父节点设置为其自身。
- 查找操作通常会递归地向上寻找直到找到根节点，并且可能更新路径上的每个节点使其直接指向根节点（路径压缩）。
- 合并操作先找到两者的根节点，然后根据一定的策略（如按秩合并）连接这两个根节点。

应用场景

- 网络连接问题：判断网络中任意两点之间是否存在路径。
- 图像处理：在某些图像分割算法中应用。
- 编译器优化：检测循环依赖关系。
- 游戏开发：实现游戏世界中的区域划分等功能。

并查集因其高效性和简洁性，在许多领域都有广泛的应用。希望以上信息对你有所帮助！如果你有更具体的问题或需要进一步的例子，请告诉我。

Dijkstra

Dijkstra是bfs+greedy

Dijkstra算法和广度优先搜索（BFS）确实有一些相似之处，但它们解决的问题类型和具体实现方式有所不同。Dijkstra算法用于在加权图中找到从一个起始顶点到所有其他顶点的最短路径，而BFS通常用于无权图中查找最短路径或层次遍历。

可以说Dijkstra算法结合了BFS的一些特性以及贪心算法的思想：

1. **与BFS的相似性**：就像BFS一样，Dijkstra算法也是逐步向外扩展搜索范围，从起点开始，逐步探索离起点最近的未访问节点。这种逐步扩展的方式保证了当某个节点第一次被访问时，所记录的距离就是从起点到该节点的最短距离。
2. **贪心选择**：Dijkstra算法在每一步都选择当前已知距离最小的顶点进行处理，并更新其邻居节点的距离值。这种总是选取“最优”选择来推进的方法体现了贪心算法的特点。通过这种方式，它能够确保一旦确定了一个顶点的最短路径长度，就不会再改变这个值了。
3. **优先队列/堆的应用**：为了高效地选取下一个待处理的节点（即距离最小的节点），Dijkstra算法通常使用优先队列（或者说是二叉堆）来存储待处理节点。这样可以快速获取距离最小的节点并维护数据结构的有序性。

然而，需要注意的是，虽然Dijkstra算法具有上述特点，但它并不完全等同于BFS加上贪心策略。Dijkstra算法适用于边有权重的情况，且这些权重必须是非负的；如果图中存在负权重边，则可能需要使用其他算法如Bellman-Ford算法。此外，Dijkstra算法在处理每个节点时考虑了边上的权重，这一点使得它比简单的BFS更复杂也更强大。因此，将Dijkstra简单地描述为"BFS+greedy"是对其特性的简化管理，但有助于初学者抓住其核心思想。

"每一步....，并更新其邻居节点的距离值"，更新其邻居节点的距离值也可以理解为dp

确实，Dijkstra算法中的“更新其邻居节点的距离值”这一过程可以被看作是一种动态规划（Dynamic Programming, DP）的思想。在动态规划中，一个问题被分解成更小的子问题来解决，解决方案通常基于先前计算的结果来构建。对于Dijkstra算法而言：

- **状态定义**：每个节点的状态可以定义为从起点到该节点的最短路径长度。
- **状态转移**：当我们处理一个节点时，我们尝试通过当前节点到达它的邻居，并检查是否可以通过这条路径得到更短的总距离。如果可以，我们就更新这个邻居节点的状态（即更新其最短路径长度）。这实际上就是一种状态转移的过程，在这里，新状态是基于旧状态和边的权重计算出来的。

因此，Dijkstra算法中的这种逐步更新最短路径估计的过程体现了动态规划的核心思想——利用已知信息来优化未知信息。不过，与经典的DP方法相比，Dijkstra算法更侧重于图论中的最短路径问题，并且使用了优先队列来高效地选择下一个要处理的节点。

总结来说，虽然Dijkstra算法主要归类为贪心算法的一种应用，但其核心操作确实包含了动态规划的思想。这是因为两者都涉及到了基于已有信息做出决策以达到全局最优解的过程。将Dijkstra算法的部分机制理解为动态规划，可以帮助更好地理解算法如何保证找到最短路径。

Dijkstra是bfs+greedy+dp。经典

Dijkstra是一种用于计算图中单源最短路径的经典算法。总结一下Dijkstra算法是如何结合BFS（广度优先搜索）、贪心选择以及动态规划思想的：

1. **BFS特性**：在无权图中，BFS可以用来找到从起点到所有其他节点的最短路径。Dijkstra算法在加权图中工作时，也采用了类似逐步扩展的方式，从起点开始逐层向外探索，确保一旦某个节点被确定为已访问状态，其最短路径就已经找到了。
2. **贪心选择**：每一步，Dijkstra算法总是选择当前距离最小且未处理过的节点进行处理，并基于这个节点更新其邻居节点的距离。这种每次都选择局部最优解以期望达到全局最优解的方法体现了贪心策略。
3. **动态规划(DP)思想**：在更新邻居节点的距离时，Dijkstra算法实际上是在执行一种形式的状态转移。每个节点的最短路径长度是基于已经计算出的较近节点的最短路径来更新的。这符合动态规划的核心思想，即通过解决子问题来构建整体问题的解决方案。

因此，说Dijkstra算法融合了BFS、贪心策略和动态规划的思想是非常恰当的。它利用了BFS的基本框架，通过贪心地选择下一步要处理的最佳节点，同时运用了类似于DP的状态更新机制来优化路径长度。不过，需要注意的是，在实际应用中，Dijkstra算法通常使用优先队列来高效管理待处理节点，从而保证算法的整体效率。

综上所述，虽然严格来说Dijkstra是一种贪心算法，但它确实巧妙地结合了上述多种算法思想的优点，使得它成为解决最短路径问题的有效工具。

heap = queue+ greedy

将堆（Heap）描述为“队列+贪心”是一种简化的表述方式，旨在强调堆在某些算法中的角色和作用。这种说法有助于理解堆如何结合了队列的先进先出（FIFO）特性与优先级选择的概念。具体来说：

- **队列**：标准队列遵循先进先出原则，即最先加入队列的元素也是最先被处理的。这与BFS等算法中逐层扩展节点的方式相吻合。

- **贪心**：贪心算法的核心思想是在每一步都做出局部最优的选择，以期望达到全局最优解。在涉及到优先级或成本的情况下，总是选择当前最佳选项。

当我们将这两个概念结合起来时，可以这样理解堆的作用：

- **优先队列**：堆本质上是一个实现优先队列的数据结构，它允许我们根据元素的优先级来访问它们，而不是简单地按照插入顺序。在最小堆中，每次都能快速获取到具有最小值的元素；而在最大堆中，则是最大值。这种基于优先级的访问方式体现了贪心策略的思想——总是选择当前最好的选项。
- **高效管理**：堆通过其特殊的树形结构保证了插入、删除以及查找最值操作的时间复杂度为 $O(\log n)$ ，这使得它非常适合于需要频繁进行这些操作的应用场景，如Dijkstra算法中的距离更新过程。

因此，说“heap = queue + greedy”实际上是指堆作为优先队列的一种实现，它不仅保持了队列的基本功能，还引入了基于优先级的贪心选择机制。这种组合使得堆成为解决诸如最短路径问题、任务调度等多种问题的强大工具。不过，需要注意的是，这是一种概念上的简化表达，实际中堆有着更丰富的特性和应用范围。