目录

[树结构 2](#_Toc6609252)

[排序 2](#_Toc6609253)

[单调队列 2](#_Toc6609254)

[栈 2](#_Toc6609255)

[堆 2](#_Toc6609256)

[矩阵乘法快速幂 2](#_Toc6609257)

[并查集 2](#_Toc6609258)

[动态规划 2](#_Toc6609259)

[最大值最小 2](#_Toc6609260)

[~~搜索~~ 2](#_Toc6609261)

[~~最短路径~~ 2](#_Toc6609262)

[输入描述: 3](#_Toc6609263)

[输入 3](#_Toc6609264)

[输出 3](#_Toc6609265)

[代码 3](#_Toc6609266)

[~~最小生成树~~ 5](#_Toc6609267)

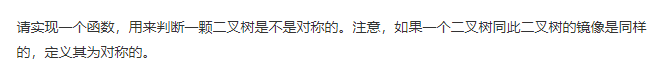
# 树结构

基本处理框架

1. **def** searchTree(root, val):
2. #边界条件
3. **if** root == None:
4. #处理边界，防止越界
5. **return** None
6. #左右子树的递归处理
7. a1 = searchTree(root.left, val2)
8. a2 = searchTree(root.right, val3)
9. #结合左右子树的结果对当前树的结果进行计算
10. a = a1+a2
11. **return** a

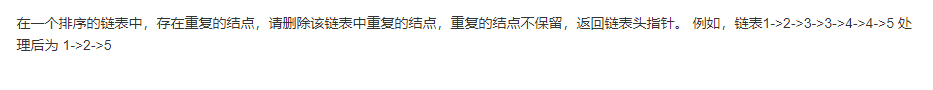


1. # -\*- coding:utf-8 -\*-
2. # class TreeNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
7. **class** Solution:
8. # 返回对应节点TreeNode
10. **def** calc(self, p, k):
11. **if** p == None:
12. **return** 0, None
13. ln, la = self.calc(p.left, k)
14. **if** ln == k - 1:
15. **return** k, p
16. **if** ln >= k:
17. **return** ln, la
18. rn, ra = self.calc(p.right, k - ln - 1)
19. **return** rn + ln + 1, ra
21. **def** KthNode(self, pRoot, k):
22. # write code here
23. **if** k == 0:
24. **return** None
25. num, ans = self.calc(pRoot, k)
26. **if** num >= k:
27. **return** ans
28. **else**:
29. **return** None



1. # -\*- coding:utf-8 -\*-
2. # class TreeNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.left = None
6. #         self.right = None
7. **class** Solution:
9. **def** calc(self, a, b):
10. **if** a == None **and** b == None:
11. **return** True
12. **if** a == None **or** b == None:
13. **return** False
14. **if** a.val != b.val:
15. **return** False
16. **return** self.calc(a.left, b.right) **and** self.calc(a.right, b.left)
18. **def** isSymmetrical(self, pRoot):
19. # write code here
20. **return** self.calc(pRoot, pRoot)

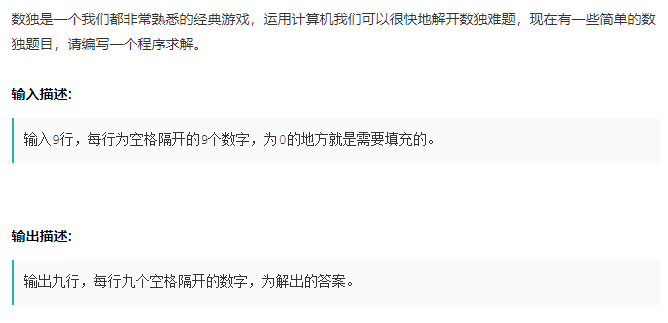
# 链表



1. # -\*- coding:utf-8 -\*-
2. # class ListNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
6. **class** Solution:
7. **def** deleteDuplication(self, pHead):
8. # write code here
9. **while** pHead!=None **and** pHead.next != None **and** pHead.val == pHead.next.val:
10. val = pHead.val
11. **while** pHead!= None **and** pHead.val == val:
12. pHead = pHead.next
13. **if** pHead != None:
14. pHead.next = self.deleteDuplication(pHead.next)
15. **return** pHead

# 回溯法

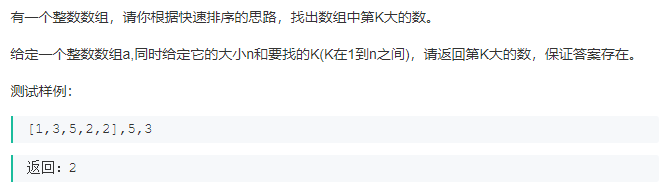
1. **def** search(state, step):
2. #边界条件
3. **if** step >= n:
4. #对结果进行判断，保存
5. **if** state:
6. ans.append(state)
7. **return**
8. #对所有可能性进行枚举
9. **for** i **in** range(n):
10. #判断当前参数是否可行
11. **if** **not** hash[i]:
12. **continue**
13. #修改剩余条件
14. hash[i] = True
15. state.append(i)
16. #进行下一层回溯
17. search(state, step+1)
18. #修改回原始条件
19. hash[i] = False
20. **del** state[-1]



1. data = []
2. **for** i **in** range(9):
3. data.append(map(int, raw\_input().split()))
4. h, l, g = [0]\*9, [0]\*9, [0]\*9
5. **for** i **in** range(9):
6. **for** j **in** range(9):
7. **if** data[i][j] == 0:
8. **continue**
9. h[i] |= 1 << (data[i][j] - 1)
10. l[j] |= 1 << (data[i][j] - 1)
11. g[i/3\*3+j/3] |= 1 << (data[i][j] - 1)

14. **def** dfs(a, b):
15. **global** data, h, l, g
16. **if** a >= 9:
17. **return** True
18. nb = b + 1
19. na = a + nb / 9
20. nb %= 9
21. **if** data[a][b] != 0:
22. **return** dfs(na, nb)
23. **else**:
24. **for** i **in** range(1, 10):
25. mark = 1<<(i-1)
26. **if** h[a] & mark > 0 **or** l[b] & mark > 0 **or** g[a/3\*3+b/3] & mark > 0:
27. **continue**
28. h[a] |= mark
29. l[b] |= mark
30. g[a/3\*3+b/3] |= mark
31. data[a][b] = i
32. **if** dfs(na, nb):
33. **return** True
34. data[a][b] = 0
35. h[a] -= mark
36. l[b] -= mark
37. g[a/3\*3+b/3] -= mark
38. **return** False
39. dfs(0, 0)
40. **for** i **in** data:
41. **print** " ".join(map(str, i))

# 排序

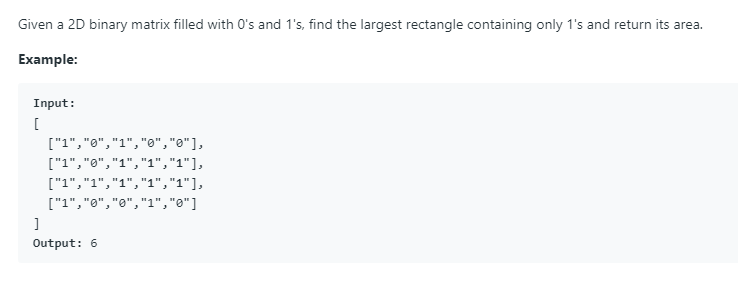


1. # -\*- coding:utf-8 -\*-
3. **class** Finder:
4. **def** findKth(self, a, n, K):
5. # write code here
6. now = a[0]
7. l = []
8. r = []
9. **for** i **in** a[1:]:
10. **if** i > now:
11. l.append(i)
12. **else**:
13. r.append(i)
14. **if** len(l)+1 == K:
15. **return** now
16. **if** len(l)>=K:
17. **return** self.findKth(l, len(l), K)
18. **else**:
19. **return** self.findKth(r, len(r), K-len(l)-1)



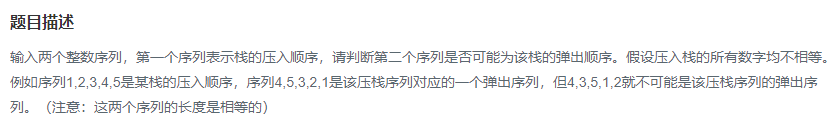
1. **import** java.util.ArrayList;
2. **public** **class** Solution {
3. **int** ans = 0;
4. **public** ArrayList<Integer> mergesort(ArrayList<Integer> array) {
5. **int** n = array.size();
6. **if**(n<=1) **return** array;
7. **int** m = n>>1;
8. ArrayList<Integer> l = **new** ArrayList<>();
9. ArrayList<Integer> r = **new** ArrayList<>();
10. ArrayList<Integer> temp = **new** ArrayList<>();
11. **for**(**int** i = 0; i < m; i++)
12. l.add(array.get(i));
13. **for**(**int** i = m; i < n; i++)
14. r.add(array.get(i));
15. l = mergesort(l);
16. r = mergesort(r);
17. **int** lenl = l.size();
18. **int** lenr = r.size();
19. **int** i = 0;
20. **int** j = 0;
21. **while** (i < lenl || j < lenr) {
22. **if**(j == lenr || i < lenl && l.get(i) <= r.get(j)) {
23. temp.add(l.get(i));
24. i++;
25. }
26. **else** {
27. temp.add(r.get(j));
28. ans = (ans + lenl - i) % 1000000007;
29. j++;
30. }
31. }
32. **return** temp;
33. }
34. **public** **int** InversePairs(**int** [] array) {
35. ans = 0;
36. ArrayList<Integer> data = **new** ArrayList<>();
37. **for** (**int** i : array) {
38. data.add(i);
39. }
40. mergesort(data);
41. **return** ans;
42. }
43. }

# 单调队列



1. **class** Solution(object):
2. **def** maximalRectangle(self, matrix):
3. """
4. :type matrix: List[List[str]]
5. :rtype: int
6. """
7. m = len(matrix)
8. **if** m == 0:
9. **return** 0
10. n = len(matrix[0])
11. u = []
12. **for** i **in** range(m):
13. u.append([])
14. **for** j **in** range(n):
15. matrix[i][j] = int(matrix[i][j])
16. **if** i == 0 **or** matrix[i][j] == 0:
17. u[i].append(matrix[i][j])
18. **continue**
19. u[i].append(matrix[i][j]+u[i-1][j])
21. l = [[0]\*n **for** i **in** range(m)]
22. r = [[0]\*n **for** i **in** range(m)]
23. **for** i **in** range(m):
24. queue = []
25. **for** j **in** range(n):
26. **while** len(queue) > 0 **and** u[i][queue[-1]] > u[i][j]:
27. r[i][queue[-1]] = j
28. **del** queue[-1]
29. queue.append(j)
30. **for** j **in** queue:
31. r[i][j] = n
32. queue = []
33. **for** j **in** range(n)[::-1]:
34. **while** len(queue) > 0 **and** u[i][queue[-1]] > u[i][j]:
35. l[i][queue[-1]] = j
36. **del** queue[-1]
37. queue.append(j)
38. **for** j **in** queue:
39. l[i][j] = -1
40. ans = 0
41. **for** i **in** range(m):
42. **for** j **in** range(n):
43. ans = max(ans, u[i][j]\*(r[i][j] - l[i][j]-1))
44. **return** ans

# 栈



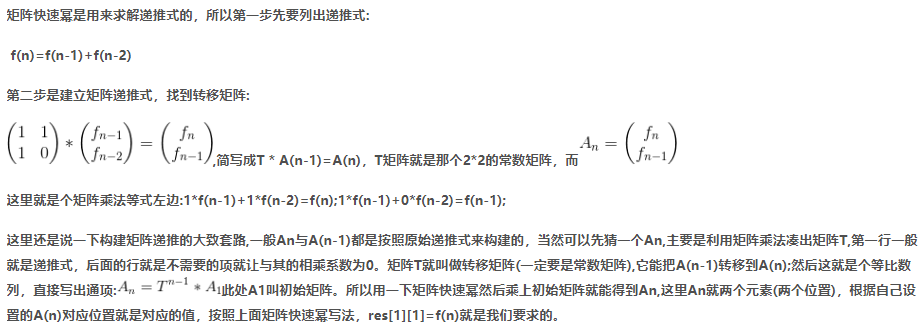
1. # -\*- coding:utf-8 -\*-
2. **class** Solution:
3. **def** IsPopOrder(self, pushV, popV):
4. # write code here
5. stack = []
6. **for** i **in** pushV:
7. stack.append(i)
8. **while** len(stack)>0 **and** len(popV)>0 **and** stack[-1] == popV[0]:
9. stack = stack[:-1]
10. popV = popV[1:]
11. **return** len(popV)==0

# 矩阵乘法快速幂

斐波那契数列及其相似数列的快速计算

图上N步随机游走的路径数量（概率）计算

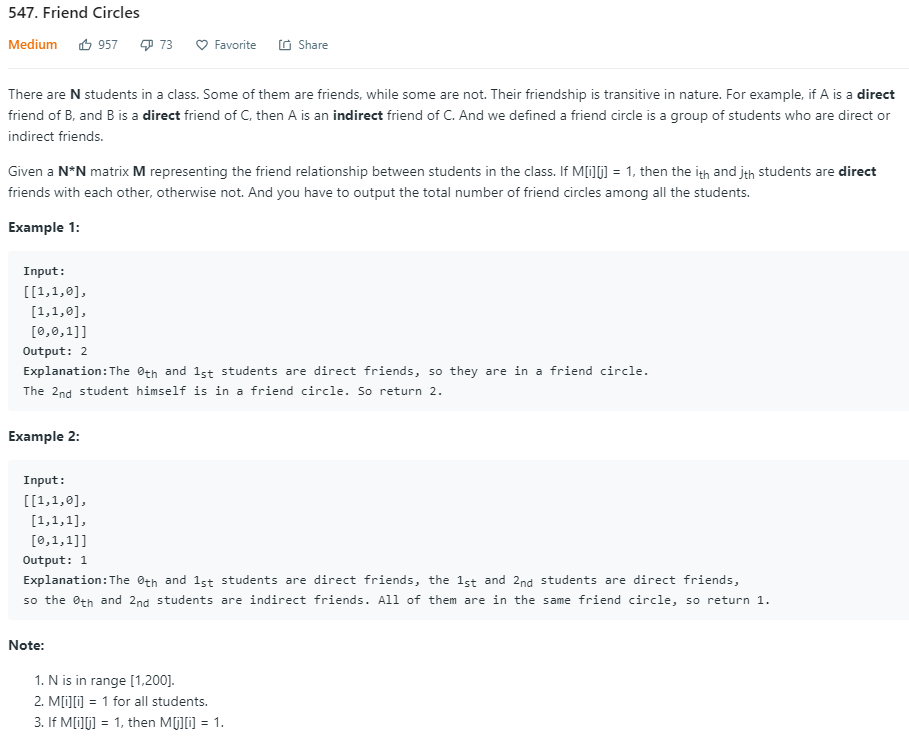
大家都知道斐波那契数列，现在要求输入一个整数n，请你输出斐波那契数列的第n项（从0开始，第0项为0）。



求从0到2^n-1的范围内所有满足i xor 2i = 3i的数的个数。

一只青蛙一次可以跳上1级台阶，也可以跳上2级。求该青蛙跳上一个n级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

# 并查集



1. **class** Solution(object):
2. f = []
3. **def** findCircleNum(self, M):
4. """
5. :type M: List[List[int]]
6. :rtype: int
7. """
8. n = len(M)
9. self.f = range(n)
10. **for** i **in** range(n):
11. **for** j **in** range(n):
12. **if** i==j **or** M[i][j]==0:**continue**
13. a = self.getf(i)
14. b = self.getf(j)
15. self.f[a] = b
16. ans = 0
17. **for** i,v **in** enumerate(self.f):
18. **if** i == v:
19. ans+=1
20. **return** ans
22. **def** getf(self, i):
23. **if** self.f[i] == i:
24. **return** i
25. self.f[i] = self.getf(self.f[i])
26. **return** self.f[i]

# 动态规划

能采用动态规划求解的问题的一般要具有3个性质：

（1）最优化原理：如果问题的最优解所包含的子问题的解也是最优的，就称该问题具有最优子结构，即满足最优化原理。

（2）无后效性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也就是说，某状态以后的过程不会影响以前的状态，只与当前状态有关。

（3）有重叠子问题：即子问题之间是不独立的，一个子问题在下一阶段决策中可能被多次使用到。（该性质并不是动态规划适用的必要条件，但是如果没有这条性质，动态规划算法同其他算法相比就不具备优势）

动态规划所处理的问题是一个多阶段决策问题，一般由初始状态开始，通过对中间阶段决策的选择，达到结束状态。这些决策形成了一个决策序列，同时确定了完成整个过程的一条活动路线(通常是求最优的活动路线)。如图所示。动态规划的设计都有着一定的模式，一般要经历以下几个步骤，如下图所示：

初始状态→│决策1│→│决策2│→…→│决策n│→结束状态

(1)划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。在划分阶段时，注意划分后的阶段一定要是有序的或者是可排序的，否则问题就无法求解。

(2)确定状态和状态变量：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。

(3)确定决策并写出状态转移方程：因为决策和状态转移有着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以如果确定了决策，状态转移方程也就可写出。但事实上常常是反过来做，根据相邻两个阶段的状态之间的关系来确定决策方法和状态转移方程。

(4)寻找边界条件：给出的状态转移方程是一个递推式，需要一个递推的终止条件或边界条件。

一般，只要解决问题的阶段、状态和状态转移决策确定了，就可以写出状态转移方程（包括边界条件）。实际应用中可以按以下几个简化的步骤进行设计：

（1）分析最优解的性质，并刻画其结构特征。

（2）递归的定义最优解。

（3）以自底向上或自顶向下的记忆化方式（备忘录法）计算出最优值

（4）根据计算最优值时得到的信息，构造问题的最优解



1. **import** java.util.Scanner;
3. **public** **class** Main {
4. **public** **static** **void** main(String[] args) {
5. Scanner sc = **new** Scanner(System.in);
6. **int** n = sc.nextInt();
7. **int** m = sc.nextInt();
8. **int** k = sc.nextInt();
9. **int**[] f = **new** **int**[2002];
10. **int**[] x = **new** **int**[1002];
11. **int**[] y = **new** **int**[1002];
12. **for**(**int** i = 0; i < m; i++) {
13. x[i] = sc.nextInt();
14. y[i] = sc.nextInt();
15. }
16. **for**(**int** i = 1; i <= n + 1001; i++) {
17. f[i] = i \* k;
18. }
19. **for**(**int** i = 0; i < m; i++) {
20. **for**(**int** j = y[i]; j <= n + 1001; j++) {
21. f[j] = Math.min(f[j], f[j - y[i]] + x[i]);
22. }
23. }
24. **int** ans = f[n + 1];
25. **for**(**int** i = 2; i <= 1001; i++) {
26. ans = Math.min(ans, f[i+n]);
27. }
28. System.out.println(ans);
29. sc.close();
30. }
31. }



1. **while** True:
2. **try**:
3. s = raw\_input()
4. t = s[::-1]
5. n = len(s)
6. f = [[10000]\*(n+1) **for** i **in** range(n+1)]
7. f[0][0] = 0
8. **for** i **in** range(n + 1):
9. **for** j **in** range(n + 1):
10. **if** i > 0:
11. f[i][j] = min(f[i][j], f[i-1][j] + 1)
12. **if** j > 0:
13. f[i][j] = min(f[i][j], f[i][j-1])
14. **if** i > 0 **and** j > 0 **and** s[i-1] == t[j-1]:
15. f[i][j] = min(f[i][j], f[i-1][j-1])
16. **print** f[n][n] - 1
17. **except**:
18. **break**



1. **import** sys
2. **try**:
3. **while** True:
4. line = sys.stdin.readline().strip()
5. **if** line == '':
6. **break**
7. lines = line.split()
8. n, m = map(int, lines)
9. stock = []
10. **for** i **in** range(n):
11. line = sys.stdin.readline().strip()
12. **if** line == '':
13. **break**
14. lines = line.split()
15. stock.append(map(int, lines))
16. f = [0] \* (m + 1)
17. **for** i **in** stock:
18. **for** j **in** range(i[0], m+1):
19. f[j] = max(f[j], f[j- i[0]] + i[1] - i[0])
20. **print** f[m]
21. **except**:
22. **pass**



1. **import** java.util.Scanner;
2. **public** **class** Main {
3. **int**[][] f;
4. **int** N;
5. **int**[] a;
6. **public** **int** search(**int** l, **int** r) {
7. **if**(f[l][r]!=-10007)
8. **return** f[l][r];
9. **if**(l == r) {
10. f[l][r] = a[l%N];
11. **return** f[l][r];
12. }
13. f[l][r] = Math.max(-search(l+1, r)+a[l%N], -search(l, r-1)+a[r%N]);
14. **return** f[l][r];
15. }
17. **public** **static** **void** main(String[] args) {
18. Main s = **new** Main();
19. s.calc();
20. }
22. **public** **void** calc() {
23. Scanner sc = **new** Scanner(System.in);
24. N = sc.nextInt();
25. a = **new** **int**[N];
26. **for**(**int** i = 0; i < N; i++) {
27. a[i] = sc.nextInt();
28. }
29. f = **new** **int**[N\*2][N\*2];
30. **for**(**int** i = 0; i < N\*2; i++) {
31. **for**(**int** j = 0; j <N\*2; j++)
32. f[i][j] = -10007;
33. }
34. **int** ans = 0;
35. **for**(**int** i = 0; i < N; i++) {
36. f[i][i+N-1] = search(i, i+N-1);
37. ans = Math.max(ans, f[i][i+N-1]);
38. }
39. System.out.println(ans);
40. }
41. }

# 最大值最小

二分查找+验证

# ~~最短路径~~

**Floyed-Warshall算法 O(N^3)**

分类：

多源最短路径算法。

作用：

1.求最短路。 2.判断一张图中的两点是否相连。

优点：

实现极为简单

缺点：

只有数据规模较小且时空复杂度都允许时才可以使用。

思想：

3层循环，第一层枚举中间点k，第二层与第三层枚举两个端点i，j。若有dis[i][j] > dis[i][k] + dis[k][j] 则把dis[i][j]更新成dis[i][k] + dis[k][j]

**Dijkstra迪杰斯特拉算法 O(N^2)**

分类：

单源最短路径算法。

适用于：

稠密图（侧重对点的处理）。

时间复杂度：

1.朴素：O(N^2)

2.堆优化：O(n \* logn)

缺点：

不能处理存在负边权的情况。

算法思想：

把点分为两类，一类是已经确定最短路径的点，称之为“标记点”；另一类则是还未确定最短路径的点，称之为“未标记点”。如果要求出一个点的最短路径，就是把这个“未标记点”变成“标记点”，从起点到“未标记点”的最短路径上的中转点在这个时刻只能是“标记点”。

Dijkstra的算法思想，就是一开始将起点到起点的距离标记为0，而后进行n次循环，每次找出一个到起点距离dis[u]最短的点u，将它从“未标记点”变为“标记的点”。随后枚举所有的“未标记的点”vi，如果以此“标记的点”为中转点到达“未标记的点”vi的路径dis[u] + w[u][vi]更短的话，这将它作为vi的“更短路径”dis[vi]（此时还不确定是不是vi的最短路径）。

就这样，我们每找到一个“标记的点”，就尝试着用它修改其他所有的：“未标记的点”，故每一个终点一定能被它的最后一个中转点所修改，而求得最短路径。

优化思想：

利用堆（优先队列），把冗杂的枚举查找变成更加快速的堆直接弹出。

**Bellman-Ford算法 O(NE)**

分类：

单源最短路径算法。

适用于：

稀疏图（侧重于对边的处理）。

优点：

可以求出存在负边权情况下的最短路径。

缺点：

无法解决存在负权回路的情况。

时间复杂度：

O(NE)，N是顶点数，E是边数。（因为和边有关，所以不适于稠密图）

算法思想：

很简单。一开始认为起点是“标记点”（dis[1] = 0），每一次都枚举所有的边，必然会有一些边，连接着“未标记的点”和“已标记的点”。因此每次都能用所有的“已标记的点”去修改所有的“未标记的点”，每次循环也必然会有至少一个“未标记的点”变为“已标记的点”。

**SPFA算法O(KE)**

适用于：

稀疏图（侧重于对边的处理）。

时间复杂度：

O(KE)，K是常数，平均值为二，E是边数。（因为和边有关，所以不适于稠密图）

来源：

SPFA是Bellman-Ford算法的一种队列实现，减少了不必要的冗余计算。

这个算法简单地说就是队列优化的Bellman-Ford，利用了每个点不会更新次数太多的特点发明的此算法。

SPFA在形式上和广度优先搜索非常类似，不同的是广度优先搜索中的一个点出了队列就不可能重新进入队列，但是SPFA中的一个点可能在出队列之后再次被放入队列，也就是说一个点修改过其他的点之后，过了一段时间可能会获得更短的路径，于是再次用来修改其他的点，这样反复进行下去。

优化方法：

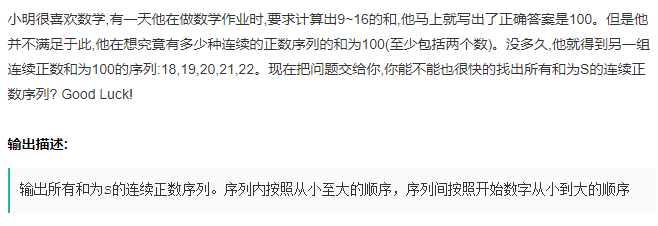
1.循环队列（可以降低队列大小）

2.SLF：Small Label First 策略，设要加入的节点是j，队首元素为i，若dist(j) < dist(i)，则将j插入队首，否则插入队尾。

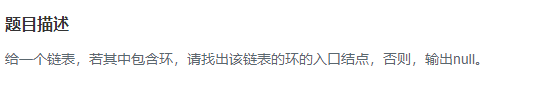


1. **import** java.util.ArrayList;
2. **import** java.util.Arrays;
3. **import** java.util.Scanner;
5. **public** **class** Main {
6. **public** **class** Edge {
7. **int** next;
8. **int** target;
9. **int** distance;
10. **public** Edge(**int** x, **int** y, **int** z) {
11. // TODO Auto-generated constructor stub
12. next = x;
13. target = y;
14. distance = z;
15. }
16. }
17. **int**[] head;
18. **int** n, m, s, t;
19. ArrayList<Edge> edges;
21. **public** **void** init() {
22. Scanner sc = **new** Scanner(System.in);
23. n = sc.nextInt();
24. m = sc.nextInt();
25. s = sc.nextInt();
26. t = sc.nextInt();
27. head = **new** **int**[n+1];
28. edges = **new** ArrayList<>();
29. **for**(**int** i = 1; i <= n; i++) {
30. head[i] = -1;
31. }
32. **for**(**int** i = 0; i < m; i++) {
33. **int** x = sc.nextInt();
34. **int** y = sc.nextInt();
35. **int** w = sc.nextInt();
36. edges.add(**new** Edge(head[x], y, w));
37. head[x] = edges.size() - 1;
38. }
39. sc.close();
40. }
42. **public** **static** **void** main(String[] args) {
43. Main main = **new** Main();
44. main.init();
45. System.out.println(main.calc());
46. }
48. **public** **int** calc() {
49. **return** spfa(s, t)+spfa(t, s);
50. }
52. **public** **int** spfa(**int** x, **int** y) {
53. ArrayList<Integer> queue = **new** ArrayList<>();
54. **int**[] best = **new** **int**[n+1];
55. **boolean**[] hash = **new** **boolean**[n+1];
56. Arrays.fill(best, Integer.MAX\_VALUE);
57. Arrays.fill(hash, **false**);
58. queue.add(x);
59. best[x] = 0;
60. hash[x] = **true**;
61. **while**(queue.size()>0) {
62. **int** h = queue.get(0);
63. queue.remove(0);
64. hash[h] = **false**;
65. **int** i = head[h];
66. **while**(i!=-1) {
67. Edge temp = edges.get(i);
68. i = temp.next;
69. **if** (best[h]+temp.distance >= best[temp.target])
70. **continue**;
72. best[temp.target] = best[h]+temp.distance;
73. **if** (hash[temp.target])
74. **continue**;
75. queue.add(temp.target);
76. hash[temp.target] = **true**;
77. }
78. }
79. **return** best[y];
80. }
81. }

# 常见题目



1. # -\*- coding:utf-8 -\*-
2. **class** Solution:
3. **def** FindContinuousSequence(self, tsum):
4. # write code here
5. i, s = 1, 0
6. ans = []
7. **for** num **in** range(1, tsum + 1):
8. **while** s + num > tsum:
9. s -= i
10. i += 1
11. s += num
12. **if** s == tsum **and** i < num:
13. ans.append(range(i, num+1))
14. **return** ans



1. # -\*- coding:utf-8 -\*-
2. # class ListNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
6. **class** Solution:
7. **def** EntryNodeOfLoop(self, pHead):
8. # write code here
9. **try**:
10. s1 = pHead
11. s2 = pHead
12. **while** True:
13. s1 = s1.next
14. s2 = s2.next.next
15. **if** s1 == s2:
16. s3 = pHead
17. **while** s3 != s2:
18. s3 = s3.next
19. s2 = s2.next
20. **return** s2
21. **except** Exception:
22. **return** None