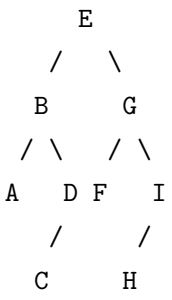
1 搜索

1.1 BFS 和 DFS 基础

构建二叉树




```
def build():
  a = node('a')
  b = node('b')
  c = node('c')
  d = node('d')
  e = node('e')
  f = node('f')
  g = node('g')
  h = node('h')
  i = node('i')
  root = e
  root.l = b
  root.r = g
  b.l = a
  b.r = d
  d.l = c
  g.l = f
  g.r = i
  i.l = h
  return root
```

1.1.1 BFS

代码实现 BFS 可以使用**队列**. 处理第 i 层的节点 a 时, 将 a 的第 i+1 层的所有孩子节点放入队尾.

```
class node():
  def __init__(self, v=None, l=None, r=None):
    self.value = v
    self.l = l
    self.r = r

# using queue to BFS
root = build()
q = deque()
q.append(root)
while len(q) > 0:
  temp = q.popleft()
  print(temp.value, end=' ')
  if temp.l:
    q.append(temp.l)
  if temp.r:
    q.append(temp.r)
```

```
e b g a d f i c h
```

BFS 遍历二叉树时, 每条边需要且只需要检查一次, 时间复杂度为 O(m), m 为边的数量; 每个点只进出队列一次, 空间复杂度为 O(n), n 为点的数量.

BFS 适用于寻找全局最优解. BFS 编码时需要注意去重, 待搜索的状态已经搜素过时, 不再将此状态放入队尾.

1.1.2 DFS

代码实现 DFS 可以使用递归, DFS 模板框架为

```
GLOBAL ans;
Function dfs(Layer NO., other parameters):
  IF (exit loop):
    ans <- updated ans;
    return; // back to previous layer
  ENDIF
  剪枝
  FOR (下一层所有可能情况):
    IF (used[i] == false): // have not been used
      used[i] = true // mark status
      dfs(layer + 1, other parameters)
      used[i] = false // recover status
    ENDIF
  ENDFOR
  return; // back to previous layer
```

DFS 常用操作

```python
      # DFS
root = build()

# 时间戳
dfn = []
dfn_timer = 0
def dfn_node(root):
  global dfn, dfn_timer
  if root:
    dfn_timer += 1
    dfn.append((root.value, dfn_timer))
    print('{}: {}'.format(root.value, dfn_timer), end=' ')
    if root.l:
      dfn_node(root.l)
    if root.r:
      dfn_node(root.r)

print('时间戳')
dfn_node(root)
print()

# DFS序
visit_timer = 0
def visit_order(root):
  global visit_timer
  if root:
    visit_timer += 1
    # 第一次 访问
    print('{}: {}'.format(root.value, visit_timer), end=' ')
    if root.l:
      visit_order(root.l)
    if root.r:
      visit_order(root.r)
    # 第二次 回溯
    visit_timer += 1
    print('{}: {}'.format(root.value, visit_timer), end=' ')

```

```
print('DFS序')
visit_order(root)
print()

# 树的深度
deep = dict()
deep_timer = 0
def deep_node(root):
  global deep_timer, deep
  if root:
    deep_timer += 1
    deep[root.value] = deep_timer
    print('{}: {}'.format(root.value, deep_timer), end=' ')
    if root.l:
      deep_node(root.l)
    if root.r:
      deep_node(root.r)
    deep_timer -= 1

print('树的深度')
deep_node(root)
print()

# 子树节点总数
num = dict()
def num_node(root):
  global num
  if not root:
    return 0
  else:
    num[root.value] = num_node(root.l) + num_node(root.r) + 1
    print('{}: {}'.format(root.value, num[root.value]), end=' ')
    return num[root.value]

print('子树节点总数')
num_node(root)
print()
```

时间戳

```
e: 1 b: 2 a: 3 d: 4 c: 5 g: 6 f: 7 i: 8 h: 9
```

DFS序

```
e: 1 b: 2 a: 3 a: 4 d: 5 c: 6 c: 7 d: 8 b: 9 g: 10 \
f: 11 f: 12 i: 13 h: 14 h: 15 i: 16 g: 17 e: 18
```

树的深度

```
e: 1 b: 2 a: 3 d: 3 c: 4 g: 2 f: 3 i: 3 h: 4
```

子树节点总数

```
a: 1 c: 1 d: 2 b: 4 f: 1 h: 1 i: 2 g: 4 e: 9
```

DFS 适用于寻找一个可行解. 编码时需要注意剪枝, 当某一个分支已经不符合要求时, 则不再在此分支上深入搜索.

1.2 剪枝与判重

1. 可行性剪枝: 检查当前状态, 出现条件不合法则剪枝

2. 搜索顺序剪枝: 搜索树有多个层次和分支, 不同的搜索顺序会造成不同的复杂度

3. 最优性剪枝: 如果当前花费的代价已经超过前面搜索到的最优解, 退出本分支. EX. 当前路径已经长于目前搜索到的最短路径

4. 排除等效冗余: 沿着当前节点搜索全部分支, 结果一样, 退出. 一般与组合问题有关, EX. 多个数字凑成一个大数, 数字的顺序不影响结果

5. 记忆化搜索: 将已经计算出的结果保存起来, 常用于 DP.

1.2.1 BFS 判重 蓝桥 642

有 9 只盘子，排成 1 个圆圈。其中 8 只盘子内装着 8 只蚱蜢，有一个是空盘。我们把这些蚱蜢顺时针编号为 1 8。

每只蚱蜢都可以跳到相邻的空盘中，也可以再用点力，越过一个相邻的蚱蜢跳到空盘中。

请你计算一下，如果要使得蚱蜢们的队形改为按照逆时针排列，并且保持空盘的位置不变（也就是 1−8 换位，2−7 换位,...），至少要经过多少次跳跃?

化圆为线, 将空盘子标记为 0, 从 12 点方向依顺时针将蚂蚱编号合并为字符串, 初始状态为 012345678. 若 1 跳到盘子里, 变为 1023456789; 若 7 越过 8 跳到盘子里, 变为 712345608. 目标状态为 087654321.

```
from collections import deque

q = deque()
mapp = set()

def solve():
 global q, mapp
 while len(q) > 0:
  now = q[0]
  q.popleft()
  s = now[0]
  step = now[1]
  if s == '087654321':
   print(step)
   return
  idx = 0
  for i in range(9):
   if s[i] == '0':
    idx = i
    break
  for j in range(idx-2, idx+3, 1):
   if j == idx:
    continue
   pos = (j + 9) % 9
   temp_lst = list(s)
   temp_lst[idx] = s[pos]
   temp_lst[pos] = s[idx]
   ns = ''.join(temp_lst)
   if ns not in mapp:
    mapp.add(ns)
    q.append((ns, step+1))

s = '012345678'
q.append((s, 0))
mapp.add(s)
solve()
```

1.2.2 可行性剪枝 poj3278

在一条直线上, 农夫在 N 位置, 奶牛在 K 位置. 农夫要抓到牛, 有 3 钟移动方法: 若位于 X, 可以移动到 X-1, X+1, 2X. 农夫需要走多少次才能从 N 到达 K.

Input: 两个整数 N, K. 0 <= N, K <= 100000

Output: 最少移动次数

使用可行性剪枝, 当 $X > K$ 时, 只能移动到 X-1, 不能扩大 X.

```
from collections import deque

n, k = map(int, input().split())
q = deque()
q.append((n, 0))
rst = []
mapp = set()

def solve():
  global q, n, k, mapp, rst
  while len(q) > 0:
    now = q[0]
    q.popleft()
    x = now[0]
    step = now[1]
    if x == k:
      rst.append(step)
    if x < 0:
      mapp.add(x)
      continue

    if x > k:
      if x-1 not in mapp:
        q.append((x-1, step+1))
        mapp.add(x-1)
    else:
      if x-1 not in mapp:
        q.append((x-1, step+1))
        mapp.add(x-1)
      if x+1 not in mapp:
        q.append((x+1, step+1))
        mapp.add(x+1)
      if 2*x not in mapp:
        q.append((2*x, step+1))
        mapp.add(2*x)


solve()
rst.sort()
print(rst[0])
```

```
>> 5 17
<< 4
```

1.2.3 最优性剪枝 洛谷 1118

输入一个 $1-n$ 的序列 a_i, 每次将相邻两个数相加, 形成新的序列, 直到剩下一个数字为止. EX.

```
3   1   2   4
  4   3   6
    7   9
      16
```

现在知道 n 和最终的数字 sum, 需要求出最初的序列 a. 若答案有多个, 则输出字典序最小的一个.

1. 从序列计算到最后一个数, 实际上是杨辉三角的应用, 可以先计算出来杨辉三角的系数.

2. 最优性剪枝: 若当前枚举到的序列中有一段, 这段子序列相加进行上述累加的结果已经超过 sum, 则整个序列都可以排除.

对于 Python: 生成全排列可以使用 itertools 中的 permutations; 列表可以直接进行大小比较, 放回的结果是第一个不同数字的大小比较结果; 查重可以将数字使用空格分隔再转换为字符串进行.

```python
from itertools import permutations
from copy import deepcopy

# prepare

cal_tab = dict()
base = [0] * 12
base[0] = 1
cal_tab[1] = base
pass_dict = set()

def move_add(a):
  i = 1
  rst = deepcopy(base)
  while i < len(a):
    if a[i-1] == 0:
      return rst
    rst[i] = a[i-1] + a[i]
    i += 1
  return rst

def tri_sum(a):
  global cal_tab
  n = len(a)
  tab = cal_tab[n]
  rst = 0
  for i in range(n):
    rst += a[i] * tab[i]
  return rst

for i in range(2, 13):
  temp = deepcopy(cal_tab[i-1])
  cal_tab[i] = move_add(temp)

# begin receive inputs
n, summ = map(int, input().split())
origin = [i for i in range(1, n+1)]
rst = []
for per in permutations(origin):
  per = list(per)
  check = ' '.join(map(str, per))
  flag = False
  for item in pass_dict:
    if item in check:
```

```
45          flag = True
46          break
47      if flag:
48          continue
49      if tri_sum(per) == summ:
50          if len(rst) == 0:
51              rst = per
52          elif per < rst:
53              rst = per
54
55      elif tri_sum(per) > summ:
56          per.pop()
57          while len(per) > 0:
58              if tri_sum(per) > summ:
59                  pass_dict.add(' '.join(map(str, per)))
60                  per.pop()
61              else:
62                  break
63
64  print(' '.join(map(str, rst)))
```

```
>> 4 16
<< 3 1 2 4
```

1.2.4 优化搜索顺序 排除等效冗余 洛谷 1120

乔治有一些同样长的小木棍，他把这些木棍随意砍成几段，直到每段的长都不超过 50。现在，他想把小木棍拼接成原来的样子，但是却忘记了自己开始时有多少根木棍和它们的长度。给出每段小木棍的长度，编程帮他找出原始木棍的最小可能长度。

输入格式

第一行是一个整数 n，表示小木棍的个数。

第二行有 n 个整数，表示各个木棍的长度 a_i

输出格式

输出一行一个整数表示答案。

1. 优化搜索顺序: 把小木棍按长度从大到小排列, 再按照从大到小的顺序尝试拼接.

2. 排除等效冗余: 即按照优化 1 进行优化, 因为在拼接中, 从大到小地拼接和从小到大的拼接一样.

3. 长度优化: 所有可行长度 D 是小木棍总长度的一个约数. 令木棍长度为 summ, 则 D 的范围为 [1, summ]. 再令 K 为原始木棍根数, 则 $summ = K \cdot D, K \in Z^+ \implies D \in \{n|summ/n \in Z^+\}$. 若 summ 不可分为多个 D 之和, 则该 D 不可用于分割.

```
n = int(input())
a = list(map(int, input().split()))
a.sort(reverse=True)
a_mark = [False for i in range(len(a))]
summ = sum(a)

def reset():
  global a_mark
  for i in range(len(a_mark)):
    a_mark[i] = False
def combine(length):
  global a, a_mark
  if length == 0:
    return True
  for i in range(len(a)):
    flag = False
    if a_mark[i]:
      continue
    else:
      if a[i] > length:
        continue
      else:
        a_mark[i] = True
        flag = combine(length - a[i])
        if not flag:
          a_mark[i] = False
    if flag:
      return True

for length in range(1, summ+1):
  reset()
  flag = True
  if summ % length != 0:
    continue
  else:
    k = summ // length
    for i in range(k):
      flag = combine(length)
      if not flag:
        reset()
        break
  if flag:
    print(length)
    break
```

```
>> 9
>> 5 2 1 5 2 1 5 2 1
<< 6
```

1.3 洪水填充

从一个种子点开始, 扩散到邻居, 再不断扩散到邻居的邻居的过程. 使用 BFS 和 DFS 都可以, DFS 更简单. EX.

```
import math
from time import sleep

def show(region):
  print()
  for i in range(15):
    for j in range(15):
      print(region[i][j], end=' ')
    print()

region = [[0 for _ in range(15)]for _ in range(15)]
for i in range(15):
  for j in range(15):
    if 4.5 < math.sqrt((i-7)**2+(j-7)**2) < 5.5:
      region[i][j] = 1

print('original')
show(region)
sleep(1)

'''
core
'''
def floodfill(x, y, new_color, old_color):
  global region
  if math.sqrt((x-7)**2+(y-7)**2) < 5.5 and region[y][x] == old_color \
    and 0 <= x <= 14 and 0 <= y <= 14:
    region[y][x] = new_color
    show(region)
    sleep(1)
    floodfill(x+1, y, 1, 0)
    floodfill(x-1, y, 1, 0)
    floodfill(x, y+1, 1, 0)
    floodfill(x, y-1, 1, 0)

floodfill(3, 5, 1, 0)
```

1.4 BFS 与最短路径

在所有相邻点的距离**相等**时, BFS 是最优的最短距离算法; 若距离**不相等**, 需要使用 Dijkstra 等通用的标准算法.

蓝桥 602 改

给出地图, 1 为障碍, 0 为可以通行的地方. 迷宫的左上角为入口, 右下角为出口. 找出一种通过迷宫的方式, 其使用的步数最少. 对每一步使用 DULR 表示, DULR 分别为下上左右.

```
010000
000100
001001
110000
```

```python
from collections import deque

class node:
  def __init__(self, x, y, path):
    self.x = x
    self.y = y
    self.path = path

rows = 30
cols = 50
a = \
'01010101001011001001010110010110100100001000101010' \
...(略)
'00111100001000010000000110111000000001000000001011' \
'10000001100111010111010001000110111010101101111000'
mapp = [[None for _ in range(cols)]for _ in range(rows)]
for r in range(rows):
  for c in range(cols):
    mapp[r][c] = int(a[c + r * cols])

vis = [[0 for _ in range(cols)]for _ in range(rows)]
op = [[1, 0],
   [-1, 0],
   [0, 1],
   [0, -1]]
P = ['R', 'L', 'D', 'U']

def bfs():
  start = node(0, 0, '')
  vis[0][0] = 1
  q = deque()
  q.append(start)
  while len(q) > 0:
    now = q.popleft()
    if now.x == cols-1 and now.y == rows-1:
      print(now.path)
    # broadcast
    for i in range(4):
      next = node(now.x + op[i][0], now.y + op[i][1], now.path+P[i])
      # out of range
      if next.x < 0 or next.y < 0 or next.x > cols-1 or next.y > rows-1:
        continue
      if vis[next.y][next.x] == 1 or mapp[next.y][next.x] == 1:
        continue
      vis[next.y][next.x] = 1
      q.append(next)

bfs()
```