作业 1

xwzeng

2023 年 3 月 22 日

1

[1]:
```
import time


def add_numbers(ls):
    total = 0
    for v in ls:
        total += v
    return total
```

[2]:
```
start = time.time()
add_numbers(range(1, 1000001))
end = time.time()
print(f'计算需要约{round((end - start) * 1000, 6)}毫秒。')
```

```
计算需要约 28.999805 毫秒。
```

2

分别计算下列函数的时间复杂度：

- $N \to O(N)$
- $N^{0.5} \to O(N^{0.5})$
- $N^{1.5} \to O(N^{1.5})$
- $N^2 \to O(N^2)$
- $N \log N \to O(N \log N)$
- $N \log(\log N) \to O(N \log(\log N))$
- $N(\log N)^2 \to O(N(\log N)^2)$
- $N \log(N^2) \to O(N \log N)$

- $2^N \to O(2^N)$
- $2^{N/2} \to O(\sqrt{2}^N)$
- $37 \to O(1)$
- $N^2 \log N \to O(N^2 \log N)$
- $N^3 \to O(N^3)$

排序原则为：

1. N 的阶数越高，增长速度越快。
2. 指数函数的增长速度始终大于 N 的多项式阶数。
3. 可以把 $\log N$ 视作 N 的正数次方，但是次数无限趋近于 0。
4. 通过求导可以得出 $\log(\log N)$ 比 $\log N$ 的增长速度慢的结论。
5. 把 $N \log(N^2)$ 中 $\log(N^2)$ 的 2 提出，可以证明 $N \log N$ 与之增长速度等价。

排序后得到：

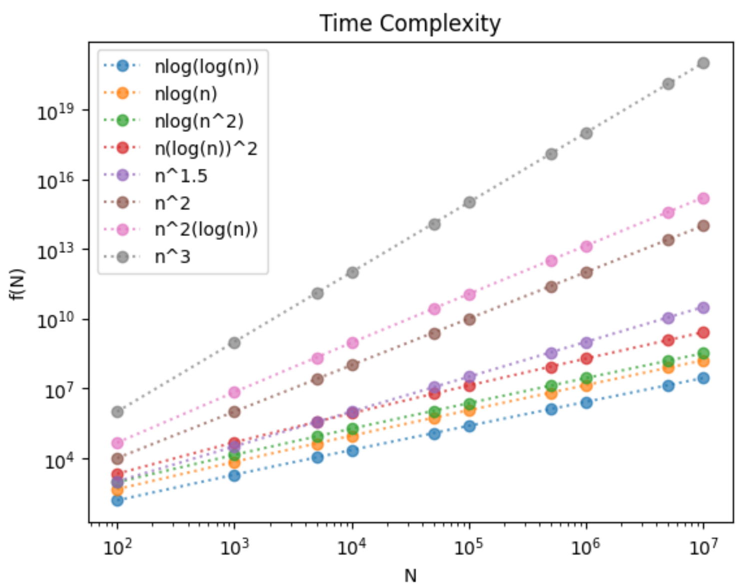
$$37, N^{0.5}, N, N\log(\log N), \{N\log N, N\log(N^2)\}, N(\log N)^2, N^{1.5}, N^2, N^2\log N, N^3, 2^{N/2}, 2^N$$

其中处于 {} 内的两个函数的增长速度相同。

我们也可以通过画出函数图像来得到相同的结论。

[3]:
```
import numpy as np
import matplotlib.pyplot as plt

Y = np.zeros((8, 10))
N_list = [100, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000,␣
 ↪10000000]
for i, n in enumerate(N_list):
    Y[0, i] = n * np.log(np.log(n))
    Y[1, i] = n * np.log(n)
    Y[2, i] = n * np.log(n ** 2)
    Y[3, i] = n * (np.log(n)) ** 2
    Y[4, i] = n ** 1.5
    Y[5, i] = n ** 2
    Y[6, i] = (n ** 2) * np.log(n)
    Y[7, i] = n ** 3
```

```
labels = ['nlog(log(n))', 'nlog(n)', 'nlog(n^2)', 'n(log(n))^2', 'n^1.5',␣
↪'n^2', 'n^2(log(n))', 'n^3']
for i, y in enumerate(Y):
    plt.loglog(N_list[:],y[:],linewidth =1.5, linestyle='dotted',label =␣
↪labels[i], alpha=0.7,marker='o')
plt.xlabel('N')
plt.ylabel('f(N)')
plt.title('Time Complexity')
plt.legend()
plt.show()
```



3

(1) 第 1 段程序:

[4]:
```
def Q3_1(N):
    sum = 0
    for i in range(N):
        sum += 1
    return sum
```

[5]:
```
def count_time_1(N):
    start = time.time()
```

```
    Q3_1(N)
    end = time.time()
    print(f'N = {N}, {round((end - start) * 1000, 6)}毫秒')
    return (end - start) * 1000
```

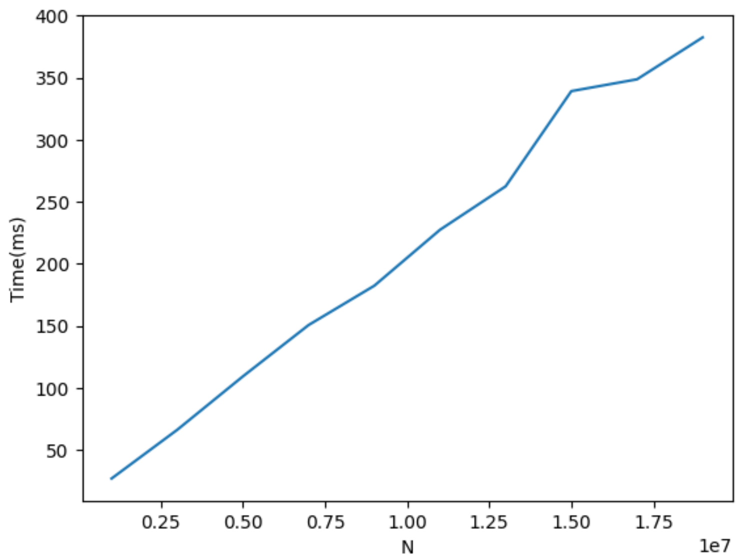
- (a) 运算时间为 $O(N)$，因为有一层 for 循环，操作了 N 次。
- (b) N 取 100000, 300000, ..., 1900000 时的运行时间如下：

[6]:
```
import numpy as np

t = [0] * 10
n = np.array(range(1000000, 20000000, 2000000))
for i, j in enumerate(n):
    t[i] = count_time_1(j)
```

```
N = 1000000, 26.952267 毫秒
N = 3000000, 66.239119 毫秒
N = 5000000, 109.296322 毫秒
N = 7000000, 150.666952 毫秒
N = 9000000, 182.343483 毫秒
N = 11000000, 227.408409 毫秒
N = 13000000, 262.466669 毫秒
N = 15000000, 339.18047 毫秒
N = 17000000, 348.71006 毫秒
N = 19000000, 382.492304 毫秒
```

- (c) 实际运行时间与 N 是很明显的线性关系，因此与 (a) 中时间复杂度的分析是相符的。

[7]:
```
import matplotlib.pyplot as plt

plt.plot(range(1000000, 20000000, 2000000), t)
plt.xlabel('N')
plt.ylabel('Time(ms)')
plt.show()
```



(2) 第 2 段程序:

[8]:
```
def Q3_2(N):
    sum = 0
    for i in range(N):
        for j in range(N):
            sum += 1
    return sum
```

[9]:
```
def count_time_2(N):
    start = time.time()
    Q3_2(N)
    end = time.time()
    print(f'N = {N}, {round((end - start) * 1000, 6)}毫秒')
    return (end - start) * 1000
```

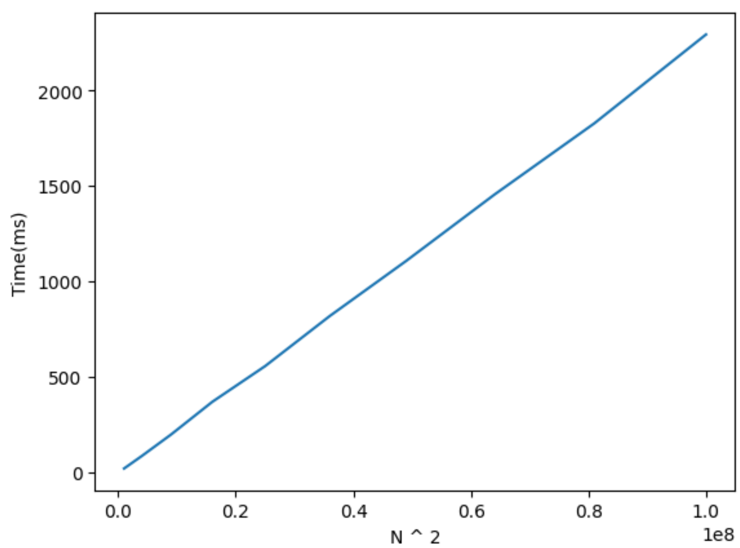
- (a) 运算时间为 $O(N^2)$，因为有两层循环，内层循环需要操作 N 次，外层则又进行了 N 次内层循环，$N \times N = N^2$。
- (b) N 取 1000, 2000, ..., 10000 时的运行时间如下:

[10]:
```
t = [0] * 10
n = np.array(range(1000, 11000, 1000))
for i, j in enumerate(n):
    t[i] = count_time_2(j)
```

```
N = 1000, 18.998861 毫秒
N = 2000, 84.131718 毫秒
N = 3000, 197.2363 毫秒
N = 4000, 368.148565 毫秒
N = 5000, 555.575609 毫秒
N = 6000, 818.181515 毫秒
N = 7000, 1106.965065 毫秒
N = 8000, 1454.161406 毫秒
N = 9000, 1827.555895 毫秒
N = 10000, 2293.26129 毫秒
```

- (c) 实际运行时间与 N^2 是很明显的线性关系，因此与 (a) 中时间复杂度的分析是相符的。

[11]:
```python
plt.plot(n * n, t)
plt.xlabel('N ^ 2')
plt.ylabel('Time(ms)')
plt.show()
```



(3) 第 3 段程序：

[12]:
```python
def Q3_3(N):
    sum = 0
    for i in range(N):
        for j in range(N * N):
            sum += 1
```

```python
    return sum
```

[13]:
```python
def count_time_3(N):
    start = time.time()
    Q3_3(N)
    end = time.time()
    print(f'N = {N}, {round((end - start) * 1000, 6)}毫秒')
    return (end - start) * 1000
```

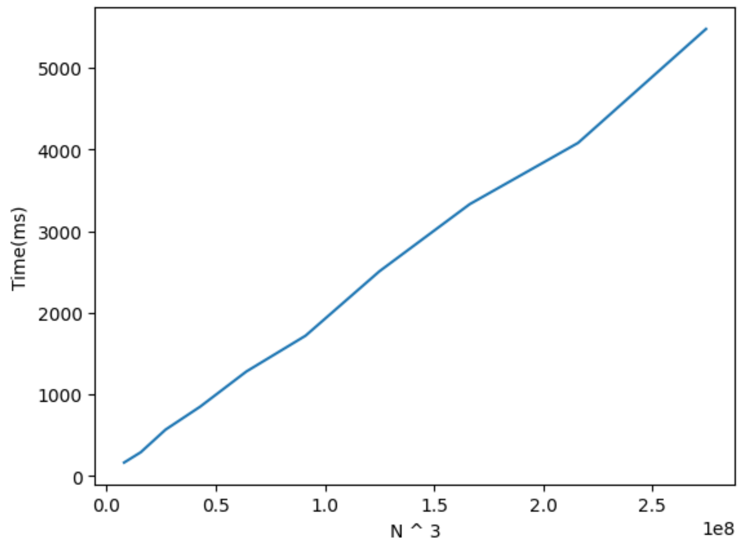
- (a) 运算时间为 $O(N^3)$，因为有两层循环，内层循环需要操作 N^2 次，外层则又进行了 N 次内层循环，$N \times N^2 = N^3$。
- (b) N 取 200, 250, ..., 650 时的运行时间如下：

[14]:
```python
t = [0] * 10
n = np.array(range(200, 700, 50))
for i, j in enumerate(n):
    t[i] = count_time_3(j)
```

```
N = 200, 162.38451 毫秒
N = 250, 288.565159 毫秒
N = 300, 566.053629 毫秒
N = 350, 847.87035 毫秒
N = 400, 1278.051853 毫秒
N = 450, 1716.883898 毫秒
N = 500, 2506.916761 毫秒
N = 550, 3329.619169 毫秒
N = 600, 4080.252409 毫秒
N = 650, 5476.906538 毫秒
```

- (c) 实际运行时间与 N^3 是很明显的线性关系，因此与 (a) 中时间复杂度的分析是相符的。

[15]:
```python
plt.plot(n ** 3, t)
plt.xlabel('N ^ 3')
plt.ylabel('Time(ms)')
plt.show()
```



(4) 第 4 段程序:

[16]:
```
def Q3_4(N):
    sum = 0
    for i in range(N):
        for j in range(i):
            sum += 1
    return sum
```

[17]:
```
def count_time_4(N):
    start = time.time()
    Q3_4(N)
    end = time.time()
    print(f'N = {N}, {round((end - start) * 1000, 6)}毫秒')
    return (end - start) * 1000
```

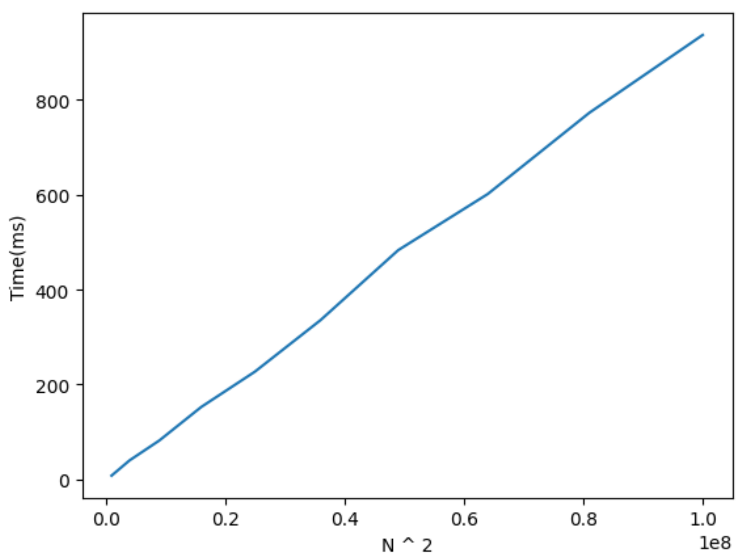
- (a) 运算时间为 $O(N^2)$，因为有两层循环，内层循环需要操作 i 次，外层则对 0 至 N-1 的每一个 i 操作 i 次，故一共进行 $N(N-1)/2$ 次操作。
- (b) N 取 1000, 2000, ..., 10000 时的运行时间如下:

[18]:
```
t = [0] * 10
n = np.array(range(1000, 11000, 1000))
for i, j in enumerate(n):
    t[i] = count_time_4(j)
```

```
N = 1000, 7.99942 毫秒
N = 2000, 40.000439 毫秒
N = 3000, 82.065582 毫秒
N = 4000, 152.549028 毫秒
N = 5000, 227.137089 毫秒
N = 6000, 335.86812 毫秒
N = 7000, 483.313084 毫秒
N = 8000, 601.728201 毫秒
N = 9000, 772.883177 毫秒
N = 10000, 937.061787 毫秒
```

- (c) 实际运行时间与 N^2 是很明显的线性关系，因此与 (a) 中时间复杂度的分析是相符的。

[19]:
```python
plt.plot(n * n, t)
plt.xlabel('N ^ 2')
plt.ylabel('Time(ms)')
plt.show()
```



(5) 第 5 段程序：

[20]:
```python
def Q3_5(N):
    sum = 0
    for i in range(N):
        for j in range(i * i):
            for k in range(j):
```

```python
                sum += 1
    return sum
```

[21]:
```python
def count_time_5(N):
    start = time.time()
    Q3_5(N)
    end = time.time()
    print(f'N = {N}, {round((end - start) * 1000, 6)}毫秒')
    return (end - start) * 1000
```

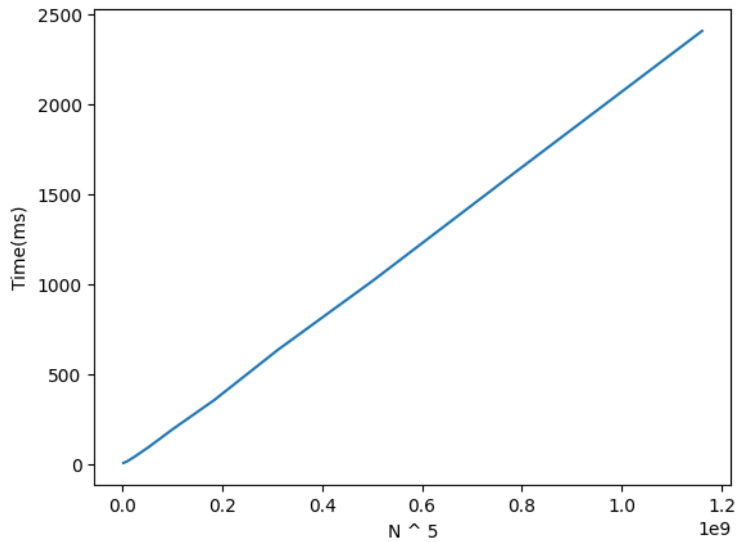
- (a) 运算时间为 $O(N^5)$，因为有三层循环，最里层一共操作 j 次，中间层对 0 至 i^2-1 中的每个 j 都操作 j 次，共进行 $0+1+2+...+(i^2-1) = i^2(i^2-1)/2$ 次操作，最外层对 0 至 N-1 的每个 i 进行 $i^2(i^2-1)/2$ 次操作，最终数量级达到 $O(N^5)$。
- (b) N 取 20, 25, 30, ..., 65 时的运行时间如下：

[22]:
```python
t = [0] * 10
n = np.array(range(20, 70, 5))
for i, j in enumerate(n):
    t[i] = count_time_5(j)
```

```
N = 20, 8.946657 毫秒
N = 25, 15.999556 毫秒
N = 30, 41.000605 毫秒
N = 35, 94.398499 毫秒
N = 40, 197.269917 毫秒
N = 45, 357.079029 毫秒
N = 50, 637.397528 毫秒
N = 55, 1022.819042 毫秒
N = 60, 1603.037596 毫秒
N = 65, 2406.447649 毫秒
```

- (c) 实际运行时间与 N^5 是很明显的线性关系，因此与 (a) 中时间复杂度的分析是相符的。

[23]:
```python
plt.plot(n ** 5, t)
plt.xlabel('N ^ 5')
plt.ylabel('Time(ms)')
plt.show()
```



(6) 第 6 段程序:

[24]:
```
def Q3_6(N):
    sum = 0
    for i in range(N):
        for j in range(i * i):
            if j % i == 0:
                for k in range(j):
                    sum += 1
    return sum
```

[25]:
```
def count_time_6(N):
    start = time.time()
    Q3_6(N)
    end = time.time()
    print(f'N = {N}, {round((end - start) * 1000, 6)}毫秒')
    return (end - start) * 1000
```

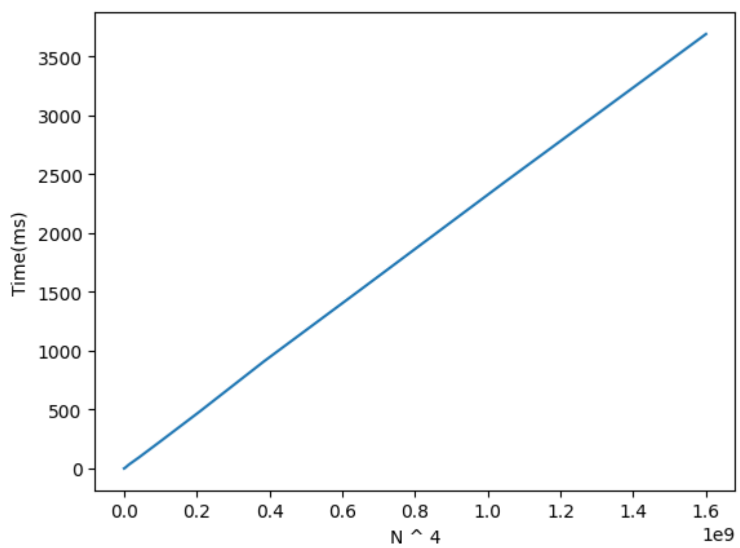
- (a) 运算时间为 $O(N^4)$，因为有三层循环，最里层一共操作 j 次，中间层对 0 至 i^2-1 中每个能被 i 整除的 j 都操作 j 次，相当于操作 $0+i+2i+...+(i-1)i=i^2(i-1)/2$ 次，最外层对 0 至 N-1 的每个 i 进行 $i^2(i-1)/2$ 次操作，最终数量级达到 $O(N^4)$。
- (b) N 取 20, 40, ..., 200 时的运行时间如下:

[26]:
```python
t = [0] * 10
n = np.array(range(20, 220, 20))
for i, j in enumerate(n):
    t[i] = count_time_6(j)
```

```
N = 20, 0.999451 毫秒
N = 40, 4.999638 毫秒
N = 60, 32.581806 毫秒
N = 80, 94.267607 毫秒
N = 100, 230.947495 毫秒
N = 120, 482.604265 毫秒
N = 140, 909.106493 毫秒
N = 160, 1531.010866 毫秒
N = 180, 2440.778971 毫秒
N = 200, 3691.1726 毫秒
```

- (c) 实际运行时间与 N^4 是很明显的线性关系，因此与 (a) 中时间复杂度的分析是相符的。

[27]:
```python
plt.plot(n ** 4, t)
plt.xlabel('N ^ 4')
plt.ylabel('Time(ms)')
plt.show()
```



4

汉诺塔问题。

[28]:
```python
class ArrayStack: # 基于 list 类实现栈类

    def __init__(self):
        self._data = []

    def __len__(self):
        return len(self._data)

    def is_empty(self):
        return len(self._data) == 0

    def push(self, e):
        self._data.append(e)

    def top(self):
        if self.is_empty():
            raise Empty('Stack is empty')

        return self._data[-1]

    def pop(self):
        if self.is_empty():
            raise Empty('Stack is empty')

        return self._data.pop()
```

[29]:
```python
def solve_hanoi(tower, N, x = 0, m = 1, y = 2): # 把 N 个盘子从柱子 0 移到柱子 2

    def move(i, j): # 从柱子 i 移动一个盘子到柱子 j
        global count
        count += 1
        p = tower[i].pop()
        tower[j].push(p)
        print(f'move({p}, {i}, {j})')
```

```python
    if N == 1:
        move(x, y)
    else:
        solve_hanoi(tower, N - 1, x, y, m) # 把 N-1 个盘子从柱子 0 移动到柱子 1
        move(x, y)
        solve_hanoi(tower, N - 1, m, x, y) # 把 N-1 个盘子从柱子 1 移动到柱子 2
```

[30]:
```python
N = 3 # 指定盘子数 N
tower = [ArrayStack(), ArrayStack(), ArrayStack()] # 定义 3 根柱子 0、1、2
for i in range(N, 0, -1): # 向柱子 0 从大到小压入 N 个盘子
    tower[0].push(i)

count = 0
solve_hanoi(tower, N)
print(f'把{N}个金盘从柱子 0 移动到柱子 2 总共需要移动{count}次。')
```

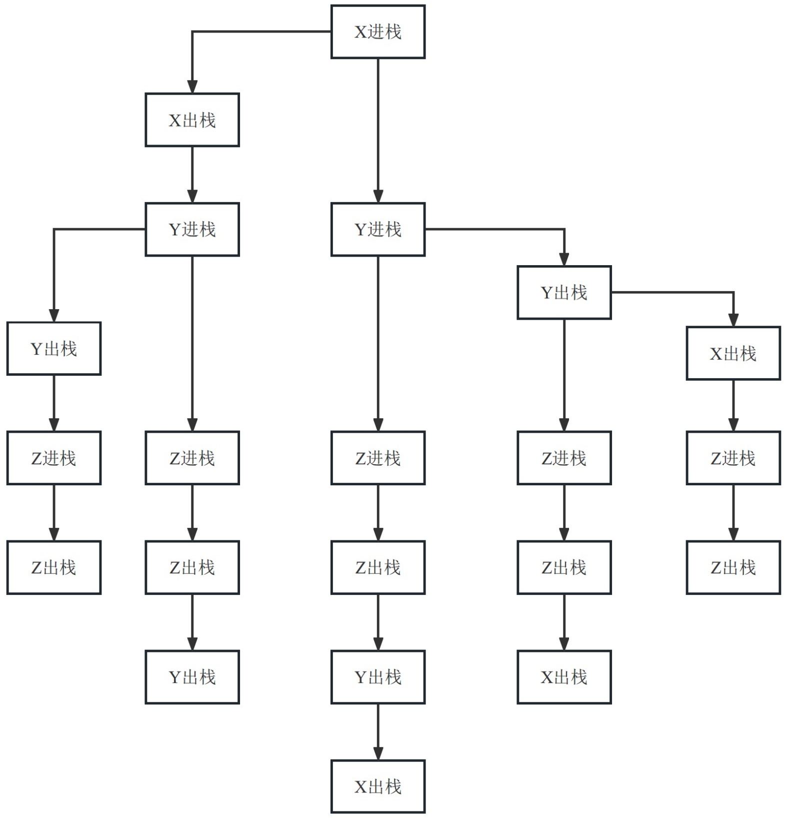
```
move(1, 0, 2)
move(2, 0, 1)
move(1, 2, 1)
move(3, 0, 2)
move(1, 1, 0)
move(2, 1, 2)
move(1, 0, 2)
把 3 个金盘从柱子 0 移动到柱子 2 总共需要移动 7 次。
```

5

如果 3 个元素进栈顺序为 X、Y、Z，一共可能有 5 种出栈顺序，分别为：

- X, Y, Z
- X, Z, Y
- Z, Y, X
- Y, Z, X
- Y, X, Z

具体情形如下图所示。



6

求解迷宫问题的思路为:

1. 令行数为 N，列数为 M，生成 $(N \times M)$ 的迷宫矩阵，并在其中随机生成 1/4 迷宫格数（期望值）的障碍物。矩阵中，0 代表可走的坐标，1 代表原有的墙或障碍物，-1 代表已走过的路，2 代表迷宫的解路径。令出口为 0，入口为 -1。
2. 定义迷宫的解路径 track 为栈类，因为当路走不通时，我们需要原路返回至上一步甚至更多步来搜索其他可能路径，符合栈类后进先出的特点。将入口坐标压入栈。
3. 从入口出发，按下、右、上、左的顺序进行搜索（因为入口在左上角，出口在右下角，理应优先搜索下方或右方的格子）。
4. 如果搜索到可走的迷宫格，则直接跳出搜索，将新坐标压入栈，赋值为 -1；如果没有搜索到可走的迷宫格，则将栈顶的坐标弹出。
5. 以新的栈顶的坐标为起点，按下、右、上、左的顺序进行搜索。
6. 重复 4、5 步，直到没有可走的点（无解），或栈顶坐标值为 (N, M)（找到出口），算法结束。

该算法的时间复杂度为 $O(MN)$，原因如下:

- 在最好情况下：该算法需要遍历 $(M + N - 2)$ 个格子，时间复杂度为 $O(M + N)$。

- 在最坏情况下：将访问一个格子时对相邻 4 格的搜索操作视为一个常数时间操作 $O(1)$，该算法需要访问除了障碍物之外的所有格子，且对每个格子最多进行 4 次操作（第一次访问时进行 1 次，第一、二、三次回溯时分别进行 1 次），时间复杂度应为 $O(MN)$。
- 不论在最坏情况还是平均情况，时间复杂度的最高阶都为 MN，故时间复杂度为 $O(MN)$。

[32]:
```
def solve_maze(N, M):
    # 构建迷宫，0 表示可走的路，1 表示墙壁/障碍物
    maze = np.ones((N + 2, M + 2), dtype = int)
    # 随机生成 1/4 迷宫格数的障碍物
    maze[1:(N + 1), 1:(M + 1)] = (np.random.rand(N, M) > 0.75).astype(int)
    maze[1, 1] = - 1 # 保证入口周围有路
    maze[N, M] = 0 # 保证出口周围有路
    track = ArrayStack() # 记录路线
    track.push((1, 1)) # 从入口开始

    def show_maze(maze): # 显示迷宫
        c = list(range(maze.shape[1]))
        k = 0
        for i in maze:
            for j in i:
                if j == 1:
                    print('口 ', end = '')
                elif j == 2:
                    print('路 ', end = '')
                else:
                    print('   ', end = '')
            print(f'{c[k]}')
            k += 1

    def move(track):
        i, j = track.top()
        for p, q in [(i + 1, j), (i, j + 1), (i - 1, j), (i, j - 1)]:
            if maze[p, q] == 0:
                track.push((p, q))
                maze[p, q] = - 1
                return True
```

```
            return False

    flag = True
    while True:
        if flag:
            flag = move(track)
        else:
            i, j = track.pop()
            if i == 1 & j == 1:
                show_maze(maze)
                return 'No solution'
            else:
                flag = True
        if track.top() == (N, M):
            break
    for i, j in track._data:
        maze[i, j] = 2
    show_maze(maze) # 展示迷宫形状与路线
    return 'Solved'
```

[33]:
```
solve_maze(8, 8)
```

```
□ □ □ □ □ □ □ □ □ □ 0
□ 路 　 　 　 　 　 □ 　 □ 1
□ 路 　 □ 　 　 □ □ 　 □ 2
□ 路 路 □ 　 　 　 □ □ □ 3
□ □ 路 路 　 　 　 　 　 □ 4
□ □ □ 路 □ 　 　 　 □ □ 5
□ 　 □ 路 □ 　 　 　 　 □ 6
□ 　 □ 路 □ □ 　 　 　 □ 7
□ 　 □ 路 路 路 路 路 路 □ 8
□ □ □ □ □ □ □ □ □ □ 9
```

[33]: 'Solved'

7

[34]:
```
def find_queen(board, start = 0):
    # board 是 N*N 的棋盘，0 表示可放置的格子，1 表示放置了皇后棋子的格子
    # start 表示从第几行开始搜索放置皇后的位置

    def check_queen(i, j): # 检查第 i,j 个格子是否可放置皇后
        # 检查所在行和列是否有皇后
        if (board[i, :] == 1).any() | (board[:, j] == 1).any():
            return False
        # 检查左上角是否有皇后
        for p, q in zip(list(range(i - 1, -1, -1)),list(range(j - 1, -1, -1))):
            if board[p, q] == 1:
                return False
        # 检查右上角是否有皇后
        for p, q in zip(list(range(i - 1, -1, -1)),list(range(j + 1, N))):
            if board[p, q] == 1:
                return False
        # 不用检查下方是否有皇后是因为算法是从上到下开始放的
        return True

    def show_board(board): # 显示棋盘和皇后所在位置
        for i in board:
            for j in i:
                if j == 1:
                    print('  ', end = '')
                else:
                    print('  ', end = '')
            print('\n', end = '')
        print('')

    if start > N - 1: # 超出棋盘范围后停止递归（基本情况）
        global count
        count += 1 # 解的数量加 1
        show_board(board)
    else:
        for j in range(N):
```

```python
            if check_queen(start, j):
                board[start, j] = 1 # 假设在当前位置放置皇后
                # 每行只能放置一个皇后，因此直接到下一行搜索放置皇后的位置
                find_queen(board, start + 1)
                board[start, j] = 0 # 撤销对棋盘的上一步操作
```

[35]:
```python
N = 8
board = np.zeros((N, N), dtype = int)
count = 0
find_queen(board)
```

由于结果太长，这里暂不显示，详见 week2_7.py。

[36]:
```python
print(f'{N}x{N}的国际象棋棋盘上共有{count}种符合条件的{N}个皇后的放置方法。')
```

8x8 的国际象棋棋盘上共有 92 种符合条件的 8 个皇后的放置方法。

附加题

在懒和尚汉诺塔问题中，总共需要 $N^2 - N + 1$ 次移动。

[37]:
```python
def lazy_hanoi(N):

    def show_move(p, i, j):
        global count
        count += 1
        print(f'move({p}, {i}, {j})')

    if N == 1:
        show_move(N, 0, 2)
    else:
        for i in range(1, N):
            show_move(i, 0, 2 - (N - i) % 2)
            for j in range(1, i):
                show_move(j, 1 + (N - i) % 2, 2 - (N - i) % 2)

        show_move(N, 0, 2)

```

```python
        for i in range(N - 1, 0, -1):
            for j in range(1, i):
                show_move(j, (N - i) % 2, 1 - (N - i) % 2)
            show_move(i, (N - i) % 2, 2)
```

[38]:

```python
N = 4
count = 0
lazy_hanoi(N)
print(f'在懒和尚问题中把{N}个金盘从柱子 0 移动到柱子 2 总共需要移动{count}次。')
```

```
move(1, 0, 1)
move(2, 0, 2)
move(1, 1, 2)
move(3, 0, 1)
move(1, 2, 1)
move(2, 2, 1)
move(4, 0, 2)
move(1, 1, 0)
move(2, 1, 0)
move(3, 1, 2)
move(1, 0, 1)
move(2, 0, 2)
move(1, 1, 2)
在懒和尚问题中把 4 个金盘从柱子 0 移动到柱子 2 总共需要移动 13 次。
```

[39]:

```python
N = 5
count = 0
lazy_hanoi(N)
print(f'在懒和尚问题中把{N}个金盘从柱子 0 移动到柱子 2 总共需要移动{count}次。')
```

```
move(1, 0, 2)
move(2, 0, 1)
move(1, 2, 1)
move(3, 0, 2)
move(1, 1, 2)
move(2, 1, 2)
move(4, 0, 1)
```

```
move(1, 2, 1)
move(2, 2, 1)
move(3, 2, 1)
move(5, 0, 2)
move(1, 1, 0)
move(2, 1, 0)
move(3, 1, 0)
move(4, 1, 2)
move(1, 0, 1)
move(2, 0, 1)
move(3, 0, 2)
move(1, 1, 0)
move(2, 1, 2)
move(1, 0, 2)
在懒和尚问题中把 5 个金盘从柱子 0 移动到柱子 2 总共需要移动 21 次。
```