

实验目标

- 1、熟悉贪心算法实现的基本方法和步骤。
- 2、理解贪心算法的实现方法和分析方法。

实验题目

问题 1：实现 Huffman 编码，其中测试数据：X={zhejiang university of technology, computer college}. Y={1310773597218806522025}.

问题 2：最小生成树，分别使用 Prim 和 Kruskal 算法实现。其中测试数据为：
啊

1.	0	2	8	1	0	0	0	0
2.	2	0	6	0	1	0	0	0
3.	8	6	0	7	5	1	2	0
4.	1	0	7	0	0	0	9	0
5.	0	1	5	0	0	3	0	8
6.	0	0	1	0	3	0	4	6
7.	0	0	2	9	0	4	0	3
8.	0	0	0	0	8	6	3	0

其中矩阵中的数据为节点之间的距离，比较并分析他们的算法复杂度（注意优先队列与邻接矩阵等不同实现方式）。

问题 2：课本 p115，4-4 磁盘文件最优存储问题。

4-4 磁盘文件最优存储问题。

问题描述：设磁盘上有 n 个文件 f_1, f_2, \dots, f_n ，每个文件占用磁盘上的 1 个磁道。这 n 个文件的检索概率分别是 p_1, p_2, \dots, p_n 且 $\sum_{i=1}^n p_i = 1$ 。磁头从当前磁道移到被检信息磁道所需的时间可用这两个磁道之间的径向距离来度量。如果文件 f_i 存放在第 i ($1 \leq i \leq n$) 道上，则检索这 n 个文件的期望时间是 $\sum_{1 \leq i \leq j \leq n} p_i p_j d(i, j)$ 。式中， $d(i, j)$ 是第 i 道与第 j 道之间的径向距离 $|i - j|$ 。

磁盘文件的最优存储问题要求确定这 n 个文件在磁盘上的存储位置，使期望检索时间达到最小。试设计一个解此问题的算法，并分析算法的正确性与计算复杂性。

算法设计：对于给定的文件检索概率，计算磁盘文件的最优存储方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行是正整数 n ，表示文件个数。第 2 行有 n 个正整数 a_i ，表示文件的检索概率。实际上第 k 个文件的检索概率应为 $a_k / \sum_{i=1}^n a_i$ 。

结果输出：将计算的最小期望检索时间输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
5	0.547396
33 55 22 11 9	

第一个实验 (Huffman 编码)

算法问题:

实现 Huffman 编码，其中测试数据：X={zhejiang university of technology, computer college}. Y={1310773597218806522025}. (输入：待编码的字符串，输出：编码，编码后的字符串，长度。)

算法原理

初步分析:

整个算法分成四个步骤：统计各个字符出现的频次，进行 Huffman 树的生成，进行 Huffman 树的编码，打印显示结果。这里只介绍 Huffman 树的生成方法和 Huffman 树的编码方法。

进行 Huffman 树的生成:

Huffman 树的生成符合贪心选择的性质，其原理为从当前的集合中 ($n \geq 2$) 选择频次出现最小的两个元素，将两个元素从源集合中取出，然后将两个元素组合在一起 (生成一个父节点，其左子树为左元素，其右子树为右元素，节点的 number 为 $\text{left.number} + \text{right.number}$)。从此往复，便可以生成一个满足条件的 Huffman 树。

由于需要在集合中选出频次出现最小的两个元素，在这里使用优先队列 priority_queue 可以解决问题。

进行 Huffman 树的编码:

从根节点出发 (设置 $\text{prefix}=""$)，当遇到叶子节点时，将该节点对应的字符设置编码为 prefix，否则，递归遍历左子节点 ($\text{prefix} += "0"$) 和右子节点 ($\text{prefix} += "1"$)。

伪代码:

```
# cal.value : node 表示此节点, cal.number : int 表示此节点的频数.
# priority_queue: 优先队列
def build_huffman(collection: priority_queue<cal>):
    while len(collection) >= 2: # 表示集合的个数>=2, 需要继续合并。
        # 提取两个最小的节点。
        first = collection.top()
        collection.pop()
        second = collection.second()
        collection.pop()
        # 合并节点
        new_node = combine(first, second)
        # 放回到优先队列
        collection.push(new_node)

# 存储编码
global encoding
```

```

# 遍历节点, 生成编码
def huffman_iter(node, prefix = ''):
    # 如果是叶子节点, 则输出
    if end_point(node):
        encoding(node) = prefix
    else
        huffman_iter(node.left, prefix + '0')
        huffman_iter(node.right, prefix + '1')

```

结果:

```

Run: an_huffman x
C:\Users\t1542\Program\C++\an_huffman\cmake-build-debug\an_huffman.exe
input the raw_string:
zhejiang university of technology, computer college
encoding:
5      001
,      1      110100
a      1      110101
c      3      0110
e      6      010
f      1      111001
g      3      11111
h      2      11101
i      3      0111
j      1      110011
l      3      1010
m      1      110010
n      3      1011
o      5      000
p      1      111000
r      2      11110
s      1      10010
t      3      1000
u      2      11000
v      1      110110
y      2      10011
z      1      110111

raw_string:zhejiang university of technology, computer college
encoded_string:1101111110101011001101111110101111111001110001011011111011001011110100100111100010011001000111001001100
001001101110110010100001111110011110100001011000011001011100011000101111000101100001010101011111010
length:216

```

图 2-1 Huffman 树实验结果 (1)

```
Run: an_huffman x
C:\Users\t1542\Program\C++\an_huffman\cmake-build-debug\an_huffman.exe
input the raw_string:
1310773597218806522025
encoding:
  0  3    101
  1  3    110
  2  4    111
  3  2    000
  5  3    100
  6  1    0100
  7  3    011
  8  2    001
  9  1    0101
raw_string:1310773597218806522025
encoded_string:110000110101011000100010101111110001001101010010011111101111100
length:68
Process finished with exit code 0
```

图 2-2 Huffman 树实验结果 2（1）

图 2-1 到 2-2 为 Huffman 树测试样例的实验结果，实验结果正确，符合预期。

分析：

贪心选择算法是一个条件最严苛，但是一般而言最优的算法。贪心算法的适应性不如动态规划广。在 Huffman 编码问题中，由于其满足贪心选择的性质（即如果其中一棵 Huffman 树是最优的，则可以通过替换是最小的两个节点位于最长编码的叶子节点处，其结果也是最优的。），因此能够使用贪心选择的算法来实现。在上面的问题中，统计、编码、迭代等结果的算法复杂度都是线性的。

第二个实验（最小生成树算法）

算法问题：

给定连接矩阵（连接矩阵对称，因为图为无向图），构造一棵连接所有节点的一棵树，使得各个边的权值最小。测试数据为：（输入：无向连接图矩阵，输出：无向连接图矩阵，长度）

测试数据：

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

0 0 0 0 8 6 3 0

算法原理:

Prim:

初始化边的优先队列 $O(e \cdot \log(e))$

Prim 算法将从一个初始点出发 (构造出一个点集 A), 一直进行下列步骤直到点集 A 与原图的所有点相等: $O(n)$

① 选择一条最小的边, 其满足其中一个点在 A 内, 另外一个点在 A 外。 $O(\log(e))$ //使用优先队列。

② 将此边加入已选择的边内, 将另外一个点加入 A 集。

③ 回到①, 直到满足退出条件。

时间复杂度 $O(e \cdot \log(e) + n \cdot \log(e))$

Kruskal:

Kruskal 将从一个点集集合 X 出发, 一直进行下列步骤直到 X 中所有点的个数与原图的点个数相等且 X 的个数为 1: $O(e)$

① 选择一条边, 若: $O(\log(e))$ //使用优先队列。

a. 两个点都在 X 包含的点外, 则 $X = X \cup \{\text{start}, \text{end}\}$, 加入这条边

b. 其中一个点在 X 内的某集合内, 另外一个点在 X 的某集合外, 则 $A = X[i] \cup \{\text{other}\}$, 并加入这条边。

c. 两个点都在 X 内, 且属于两个不同的集合, 则将这两个集合合并, 放回原集合, 并加入这条边。

d. 两个点都在 X 内, 且属于相同的集合, 则舍弃这条边。

时间复杂度 $O(e \cdot \log(e) + e \cdot \log(e))$

比较:

两个算法在初始化边的时候都会使用优先队列, 建立优先队列的时间为 $O(e \cdot \log(e))$, 然后再 Prim 算法中, 将在优先队列中查找满足的集合, 时间复杂度为 $O(\log(e))$, 循环的次数为 $O(n)$, 时间复杂度为 $O(n \cdot \log(e))$ 。

再 Kruskal 算法中, 由于从边的角度考虑问题, 所以时间复杂度为 $O(e \cdot \log(e))$ 。

由于再此问题中, 一定满足 $n \leq e \leq n^2$ 。因此两个算法的时间复杂度均为 $O(e \cdot \log(e))$

伪代码:

```
def prim(size, collection: priority<side>):
    a = {0} # 从 index=0 的点开始
    select_side = {}
    while a.size() < size:
        side = collection.top()
        collection.pop()
        # 表示此边满足其中一个点再内部, 另外一个再外部
        if type(side, p) == in_out :
            a.add(outer_point(p))
            select_side.add(side)
    return graph(size, select_side)
```

```

def kruskal(size, collection: priority<side>):
    a = {} # 从空节点开始, 其中的每个元素都将表示连通子图的点的集合。
    select_side = {}
    # 结束条件为选择了所有节点且 a 集合只有 1 个元素。
    while a.cal_point_count() < size or a.size() > 1:
        side = collection.top()
        collection.pop()
        # 取出两个点再集合中的下标, 若不存在, 则为 1
        index1, index2 = index_of(side)
        if index1 == -1 and index2 == -1 :
            a.add({side.start, side.end})
            select_side.add(side)
        else if index1 == -1 or index2 == -1:
            if index1 == -1:
                a[index2].add(side.end)
            else:
                a[index1].add(side.start)
            select_side.add(side)
        else if index1 != index2
            # 合并两个集合。
            combine(a, index1, index2)
            select_side.add(side)

```

结果:

```
Run: an_minimaltree x
C:\Users\t1542\Program\C++\an_minimaltree\cmake-build-debug\an_minimaltree.exe
input the matrix:
0 2 8 1 0 0 0 0
2 0 6 0 1 0 0 0
8 6 0 7 5 1 2 0
1 0 7 0 0 0 9 0
0 1 5 0 0 3 0 8
0 0 1 0 3 0 4 6
0 0 2 9 0 4 0 3
0 0 0 0 8 6 3 0

prim:
[ 0, 2, 0, 1, 0, 0, 0, 0,
  2, 0, 0, 0, 1, 0, 0, 0,
  0, 0, 0, 0, 0, 1, 2, 0,
  1, 0, 0, 0, 0, 0, 0, 0,
  0, 1, 0, 0, 0, 3, 0, 0,
  0, 0, 1, 0, 3, 0, 0, 0,
  0, 0, 2, 0, 0, 0, 0, 3,
  0, 0, 0, 0, 0, 0, 3, 0,]
prim_length:13
kruskal:
1
[ 0, 2, 0, 1, 0, 0, 0, 0,
  2, 0, 0, 0, 1, 0, 0, 0,
  0, 0, 0, 0, 0, 1, 2, 0,
  1, 0, 0, 0, 0, 0, 0, 0,
  0, 1, 0, 0, 0, 3, 0, 0,
  0, 0, 1, 0, 3, 0, 0, 0,
  0, 0, 2, 0, 0, 0, 0, 3,
  0, 0, 0, 0, 0, 0, 3, 0,]
kruskal_length:13
```

图 2-3 使用 Prim 算法和 Kruskal 算法解决最小生成树问题。

分析：

最小生成树算法相对来说使用 **Prim** 算法的实现更加简单一点（因为 **Kruskal** 算法需要维护多个连通子图），两者算法在使用优先队列的前提上，两者的时间复杂度是相同的。

第三个实验（磁盘最优存储问题）

算法问题：

4-4 磁盘文件最优存储问题。

问题描述：设磁盘上有 n 个文件 f_1, f_2, \dots, f_n ，每个文件占用磁盘上的 1 个磁道。这 n 个文件的检索概率分别是 p_1, p_2, \dots, p_n 且 $\sum_{i=1}^n p_i = 1$ 。磁头从当前磁道移到被检信息磁道所需的时间可用这两个磁道之间的径向距离来度量。如果文件 f_i 存放在第 i ($1 \leq i \leq n$) 道上，则检索这 n 个文件的期望时间是 $\sum_{1 \leq i \leq j \leq n} p_i p_j d(i, j)$ 。式中， $d(i, j)$ 是第 i 道与第 j 道之间的径向距离 $|i - j|$ 。

磁盘文件的最优存储问题要求确定这 n 个文件在磁盘上的存储位置，使期望检索时间达到最小。试设计一个解此问题的算法，并分析算法的正确性与计算复杂性。

算法设计：对于给定的文件检索概率，计算磁盘文件的最优存储方案。

数据输入：由文件 input.txt 给出输入数据。第 1 行是正整数 n ，表示文件个数。第 2 行有 n 个正整数 a_i ，表示文件的检索概率。实际上第 k 个文件的检索概率应为 $a_k / \sum_{i=1}^n a_i$ 。

结果输出：将计算的最小期望检索时间输出到文件 output.txt。

输入文件示例
input.txt
5
33 55 22 11 9

输出文件示例
output.txt
0.547396

测试样例

5
33 55 22 11 9

算法原理：

简单分析：

此问题的核心关键在于确定磁盘中 n 个文件的位置，目标函数是 $\sum_{1 \leq i \leq j \leq n} p_i p_j d(i, j)$ ，由题目

可以知道 $\sum_{1 \leq i \leq j \leq n} p_i p_j$ 和 $\sum_{1 \leq i \leq j \leq n} d(i, j)$ 都是相等的，一个比较自然的想法是在 $d(i, j)$ 最大的地

方放两个概率最小的两个文件。一次迭代，则可以想到在 $index = 0, 4, 1, 3, 2$ 一次从小到大放置文件。

下面将会证明在一个子问题中，在最两端放置概率最小的两个文件属于最优解：

假设一个最优的解的值为 l ，两个文件选中的概率分别为 p_a, p_b ，两端的概率为 p_c, p_d ，一开始两者的距离为 d ，最大的距离为 d_{max} 。之后将两个文件放在两端。

则 $\Delta l = +(d_{max} - d)(p_a p_b) - (d_{max} - d)(p_c p_d) < 0$ 。所以验证了贪心选择。

其次，在构造子问题中，如果将最小的排除在外，则子问题同时也适合贪心选择。

所以，使用交错从两端向中间从小达到排列时最优的一种方案。

算法复杂度分析：

构造优先队列的时间复杂度为 $O(n \log n)$ ，循环次数 $O(n)$ ，循环内部时间复杂度为 $O(1)$ ，计算平均时间的时长为 $O(n^2)$

伪代码：

```
def disk_sequence(vector<int> disks):
```



```

# 使用优先队列
queue = priority_queue(disks)
for i = 0 .. disks.size() - 1:
    arr = array[disk.size()]
    if even (i): # i 是偶数 0->0,2->1,...
        arr[i // 2] = queue.top()
    else: # i 是奇数 1->n-1,3->n-2
        arr[n - i // 2 - 1] = queue.top()
    queue.pop()

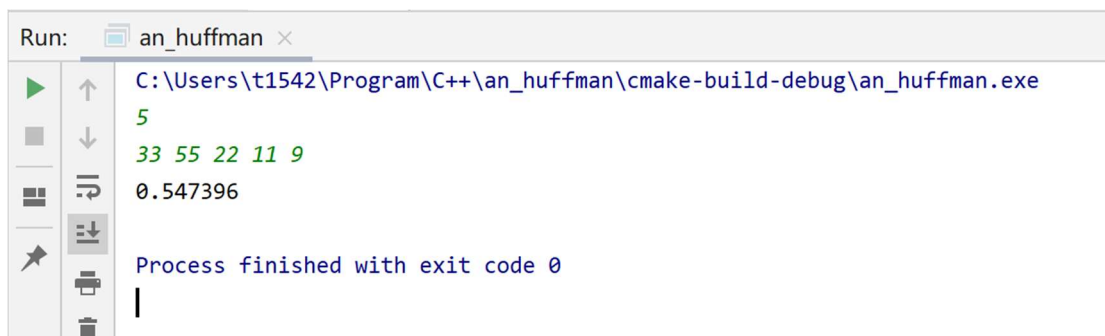
sum = sum(disks)
value = 0

# 计算长度
for i = 0 .. disks.size() - 2
    for j = i + 1.. disks.size() - 1
        value += (j-i) * arr[i] * arr[j]

return value / sum

```

结果:



```

Run: an_huffman x
C:\Users\t1542\Program\C++\an_huffman\cmake-build-debug\an_huffman.exe
5
33 55 22 11 9
0.547396
Process finished with exit code 0
|

```

图 2-4 磁盘最优存储最优问题的实验结果

分析:

此类有约束条件（通常为某些项是相等的）的最优值解法一般可以用局部最优来得到解决。此类问题能否能够使用贪心选择的关键在于其是否符合了贪心选择的性质，即是否能够构造出具有贪心性质的最优解，其解决方法是自顶向下的解法。如果问题与子问题的解是有关联的，子问题会制约之后所做出的选择时，那么就应该考虑使用动态规划的方法来得到解决。