

2024-2025 《数据结构》大作业报告

学号：231220105

姓名：刘笑姪

院系：计算机学院计算机科学与技术系

一、第一题 魔法解谜

1. 解题思路：

- 使用数组存储的二叉树表示蜡烛阵，二叉树节点 `TreeNode` 表示蜡烛。

```
1 struct TreeNode
2 {
3     bool out;    //表示蜡烛亮灭状态：灭 true，亮false
4     int left;    //指向该节点的左子树
5     int right;   //指向该节点的右子树
6
7     TreeNode() :out(false), left(-1), right(-1) {}
8 };
9
10 class Tree
11 {
12 public:
13     Tree();
14     void solution();
15
16 private:
17     TreeNode* treeList; //数组存储二叉树，数组下标即为节点编号
18     int n, m;    // n 表示蜡烛阵的蜡烛数量， m 表示集合 S 的大小
19     int cnt;     //存储翻转操作的次数
20
21     void reverse(int i);    //翻转操作：将蜡烛节点 i 及其子树中所有蜡烛节点的亮灭状态
    翻转
22     void check(int i);    //从节点 i 开始依次向后检查蜡烛节点
23 };
```

- 从根开始依次遍历各节点，遇到熄灭的蜡烛就点亮。

```
1 int main()
2 {
3     Tree tree;
4     tree.solution();
5
6     return 0;
7 }
8
9 void Tree::solution()
10 {
```

```

11     cnt = 0;
12
13     check(1);
14     cout << cnt;
15 }

```

- 时间复杂度分析

- 构建树的过程需要 $O(n)$ 时间，设置蜡烛初始状态需要 $O(m)$ 时间；
- 对任意节点 i 进行 `reverse()` 操作所需要的最坏时间复杂度为 $O(n)$ ；
- 进行 `check()` 操作的最坏情况需要进行 $O(n)$ 次 `reverse()` 操作；
- 所以总的时间复杂度在最坏情况下为 $O(n^2)$ 。

2. 核心代码+注释:

```

1  void Tree::reverse(int i)    //翻转操作：将蜡烛节点 i 及其子树中所有蜡烛节点的亮灭状态
    翻转
2  {
3      if (i == -1)            //若节点为空，则结束递归；否则按前序排序进行递归
4
5          return;
6
7      treeList[i].out = !treeList[i].out; //改变当前节点的亮灭状态
8      //依次对左右子树递归进行翻转操作
9      reverse(treeList[i].left);
10     reverse(treeList[i].right);
11 }
12
13 void Tree::check(int i) //从节点 i 开始依次向后检查蜡烛节点
14 {
15     if (i == -1)            //若节点为空，则结束递归；否则按前序排序进行递归
16         return;
17
18     if (treeList[i].out)     //若当前节点状态为灭，翻转该节点并计数加 1
19     {
20         cnt++;
21         reverse(i);
22     }
23     // 依次对左右子树递归进行检查
24     check(treeList[i].left);
25     check(treeList[i].right);
26 }

```

3. OJ运行结果截图

Test #1:	score: 10	Accepted	time: 4ms	memory: 3444kb
Test #2:	score: 10	Accepted	time: 1ms	memory: 3328kb
Test #3:	score: 10	Accepted	time: 5ms	memory: 3388kb
Test #4:	score: 10	Accepted	time: 6ms	memory: 3472kb
Test #5:	score: 10	Accepted	time: 0ms	memory: 3392kb
Test #6:	score: 10	Accepted	time: 1ms	memory: 3456kb
Test #7:	score: 10	Accepted	time: 5ms	memory: 3384kb
Test #8:	score: 10	Accepted	time: 4ms	memory: 3452kb
Test #9:	score: 10	Accepted	time: 32ms	memory: 4508kb
Test #10:	score: 10	Accepted	time: 318ms	memory: 14000kb

二、第二题 小蓝鲸的战斗力

1. 解题思路：

- 使用数组存储的最大堆 MaxHeap 记录战斗结果

```
1  const int DefaultSize = 128;
2
3  class MaxHeap
4  {
5  public:
6      MaxHeap(int sz = DefaultSize);
7      MaxHeap(int arr[], int n);
8      ~MaxHeap() { delete[] heap; }
9
10     int size() { return currentSize; }
11     int top() { return heap[0]; }
12     bool push(int& x);
13     bool pop();
14     bool pop(int& x);
15     bool empty() const { return currentSize == 0; }
16     bool full() const { return currentSize == maxHeapSize; }
17     void clear() { currentSize = 0; }
18
19 private:
20     int* heap;
21     int currentSize;
22     int maxHeapSize;
23     void siftDown(int start, int m);
24     void siftUp(int start);
25 }
```

```

26 | };
27 |

```

• 时间复杂度分析

- 读取输入数据的时间复杂度为 $O(n + q)$
- 初始化堆 `heap_1` 和 `heap_2` 的时间复杂度为 $O(n + r)$
- 将新战斗结果插入 `heap_1` 的总时间为 $O(\log n)$,
- 对每个检查点进行检查时,

$$\text{top_index} = \left\lceil \frac{\text{check_points}[i]}{m} \right\rceil$$
 将最近的 `r` 个战斗结果插入 `heap_2` 的时间为 $O(r \log r)$, 从 `heap_2` 中弹出 `k` 个最大值并累加的时间为 $O(k \log r)$, 将弹出的 `top_index` 个元素重新插入 `heap_1` 的时间为 $O(\text{top_index} \log n)$
- 对每次查询, 时间复杂度近似为 $O\left(\frac{n}{m} \cdot \log n + (r + k) \cdot \log r\right)$, 所以 `q` 次查询的总的时间复杂度为 $O\left(q \cdot \left(\frac{n}{m} \cdot \log n + (r + k) \cdot \log r\right)\right)$ 。
- 总的时间复杂度为 $O\left(n \cdot \log n + q \cdot \left(\frac{n}{m} \cdot \log n + (r + k) \cdot \log r\right)\right)$

2. 核心代码+注释:

```

1  int main()
2  {
3      // n 代表战斗总次数
4      // 已经过 N 次战斗时, 最高的 (N / m)取上整 个战斗结果中最低的战斗结果为 x,
5      //          最近的 r 个战斗结果中最高的 k 个战斗结果为 y[1], ... ,
6      //          y[k] (不足 r 个用 0 补齐)
7      // 当前战斗力为 x * k + Σy
8      // 输出 q 次即时查询的结果
9
10     //初始化
11     int n, m, r, k, q;
12     cin >> n >> m >> r >> k >> q;
13
14     int* combat_results = new int[n + 1]; //存储所有战斗的结果
15     combat_results[0] = 0;
16     for (int i = 1; i <= n; i++)
17         cin >> combat_results[i];
18
19     int* check_points = new int[q + 1]; //存储查询的时刻
20     check_points[0] = 0;
21     for (int i = 1; i <= q; i++)
22         cin >> check_points[i];
23
24     MaxHeap heap_1(n); //用于计算位于查询点时最高的 (N / m)取上整 个战斗结果
25     MaxHeap heap_2(r); //用于计算最近的 r 个战斗结果中最高的 k 个战斗结果
26
27     for (int i = 1; i <= q; i++)
28     {
29         heap_2.clear();
30
31         int top_index = (check_points[i] + m - 1) / m;
32         int x, y = 0;
33         int* popped_x = new int[top_index];

```

```

33
34     // 求 x
35     for (int j = check_points[i - 1] + 1; j <= check_points[i]; j++)
36         heap_1.push(combat_results[j]);
37     // 保存被 pop 的战斗结果
38     for (int j = 0; j < top_index; j++)
39     {
40         heap_1.pop(x);
41         popped_x[j] = x;
42     }
43
44     // 求 y[1] ... y[k]
45     for (int j = 0; j < r; j++)
46     {
47         int index = check_points[i] - j;
48         if (index <= 0)
49             break;
50
51         heap_2.push(combat_results[index]);
52     }
53
54     for (int j = 0; j < min(k, check_points[i]); j++)
55     {
56         y += heap_2.top();
57         heap_2.pop();
58     }
59
60     // 打印当前检查点的结果
61     cout << x * k + y << ' ';
62
63     // 恢复堆, 将计算 x 时被 pop 的战斗结果 push 回 heap_1
64     for (int j = 0; j < top_index; j++)
65         heap_1.push(popped_x[j]);
66 }
67
68 return 0;
69 }

```

3. OJ运行结果截图

Test #1:	score: 10	Accepted	time: 3ms	memory: 3380kb
Test #2:	score: 10	Accepted	time: 3ms	memory: 3272kb
Test #3:	score: 10	Accepted	time: 1ms	memory: 3376kb
Test #4:	score: 10	Accepted	time: 0ms	memory: 3380kb
Test #5:	score: 10	Accepted	time: 3ms	memory: 3356kb
Test #6:	score: 10	Accepted	time: 3ms	memory: 3292kb
Test #7:	score: 10	Accepted	time: 6ms	memory: 3536kb
Test #8:	score: 10	Accepted	time: 45ms	memory: 4396kb
Test #9:	score: 10	Accepted	time: 500ms	memory: 21336kb
Test #10:	score: 10	Accepted	time: 251ms	memory: 11796kb

三、第三题 小蓝鲸学传送

1. 解题思路：

- 使用邻接表存储的图记录小岛、绳索和传送门

```

1  struct edge
2  {
3      int dest;
4      int cost;
5      edge* link;
6
7      edge(int d, int c, edge* l = nullptr) :dest(d), cost(c), link(l) {};
8  };
9
10 struct vertex
11 {
12     edge* adj;
13
14     vertex(edge* a = nullptr) :adj(a) {};
15 };
16
17 class graph
18 {
19 public:
20     graph();
21
22     int getweight(int u, int v);
23     void solution();
24
25 private:

```

```

26     int n, m, s, t, q;
27     vertex* web;      //邻接表
28     vertex* rweb;     //转置邻接表
29     int* dist;        //记录从起点 s 到各顶点的最短路径
30     int* rdist;       //记录从各顶点到终点 t 的最短路径
31
32     void dijkstra(int v); //计算 dist 数组
33     void r_dijkstra(int v); //计算 rdist 数组
34 };

```

- 使用数组存储的最小堆 MinHeap 以及结构体 HeapNode 辅助实现Dijkstra算法

```

1  const int DefaultSize = 128;
2
3  template<class T>
4  class MinHeap
5  {
6  public:
7      MinHeap(int sz = DefaultSize);
8      MinHeap(T arr[], int n);
9      ~MinHeap() { delete[] heap; }
10     T top() { return heap[0]; }
11     bool push(T& x);
12     bool pop();
13     bool pop(T& x);
14     bool empty() const { return curSize == 0; }
15     bool full() const { return curSize == maxHeapSize; }
16     void clear() { curSize = 0; }
17
18 private:
19     T* heap;
20     int curSize;
21     int maxHeapSize;
22     void siftDown(int start, int m);
23     void siftUp(int start);
24
25 };
26
27 //最小堆的节点，成员为小岛编号和当前最短距离
28 struct HeapNode {
29     int vertex;
30     int distance;
31
32     HeapNode(int v = 0, int d = 0) : vertex(v), distance(d) {};
33
34     bool operator<(const HeapNode& h) const
35     {
36         return distance < h.distance;
37     }
38
39     bool operator<=(const HeapNode& h) const
40     {
41         return distance <= h.distance;
42     }
43

```

```

44     bool operator>(const HeapNode& h) const
45     {
46         return distance > h.distance;
47     }
48
49     bool operator>=(const HeapNode& h) const
50     {
51         return distance >= h.distance;
52     }
53 };

```

- 时间复杂度分析

- 初始化图的时间复杂度为 $O(m)$
- Dijkstra算法最多进行 $O(m)$ 次插入操作, $O(n)$ 次删除操作, 每次操作的时间复杂度为 $O(\log n)$, 总的时间复杂度为 $O((n + m) \log n)$
- q 次查询的时间复杂度为 $O(q)$
- 总的时间复杂度为 $O((n + m) \log n + q)$

2. 核心代码+注释:

```

1  void graph::solution()
2  {
3      //计算从起点 s 到所有顶点的最短距离
4      dijkstra(s);
5      // 计算从终点 t 到所有顶点的反向最短距离
6      r_dijkstra(t);
7
8      cin >> q;
9      for (int i = 0; i < q; i++)
10     {
11         int u, v, w;
12         cin >> u >> v >> w;
13
14         //初始答案为当前从 s 到 t 的最短距离
15         int ans = dist[t];
16
17         //检查是否存在从 s 到 u 的路径和从 v 到 t 的路径, 并且检查添加边 u->v 后的路径
           长度是否更短
18         if (dist[u] < MAX && rdist[v] < MAX && dist[u] + w + rdist[v] < ans)
19             ans = dist[u] + w + rdist[v];
20
21         //输出结果, 如果无法到达则输出 -1, 否则输出最短路径长度
22         if (ans == MAX)
23             cout << "- 1\n";
24         else
25             cout << ans << "\n";
26     }
27 }
28
29 void graph::dijkstra(int v)
30 {
31     fill(dist, dist + n + 1, MAX);

```



```

32     dist[v] = 0;
33
34     MinHeap<HeapNode> heap(n * 2);
35     HeapNode h(v, 0);
36     heap.push(h);
37
38     while (!heap.empty())
39     {
40         HeapNode cur = heap.top(); //使用最小堆选择当前距离最小的顶点
41         heap.pop();
42
43         int u = cur.vertex;
44         int cur_dist = cur.distance;
45
46         //遍历该顶点的所有邻接边，更新相邻顶点的最短距离，并将更新后的顶点重新插入堆中
47         edge* e = web[u].adj;
48         while (e)
49         {
50             int v = e->dest;
51             int w = e->cost;
52             if (dist[u] < MAX && dist[u] + w < dist[v])
53             {
54                 dist[v] = dist[u] + w;
55                 HeapNode node(v, dist[v]);
56                 heap.push(node);
57             }
58             e = e->link;
59         }
60     }
61 }
62
63 void r_dijkstra(int v) //使用转置邻接表 rweb 计算数组 rdist, 与 dijkstra(int v)
    函数的实现相仿

```

3. OJ运行结果截图

Test #1:	score: 10	Accepted	time: 5ms	memory: 3376kb
Test #2:	score: 10	Accepted	time: 5ms	memory: 3448kb
Test #3:	score: 10	Accepted	time: 1ms	memory: 3380kb
Test #4:	score: 10	Accepted	time: 1ms	memory: 3484kb
Test #5:	score: 10	Accepted	time: 3ms	memory: 3560kb
Test #6:	score: 10	Accepted	time: 3ms	memory: 3560kb
Test #7:	score: 10	Accepted	time: 9ms	memory: 3620kb
Test #8:	score: 10	Accepted	time: 4ms	memory: 3608kb
Test #9:	score: 10	Accepted	time: 159ms	memory: 5008kb
Test #10:	score: 10	Accepted	time: 233ms	memory: 5648kb

四、第四题 最终挑战

1. 解题思路:

- 使用邻接表存储的图记录小岛、绳索和传送门

```

1  struct EtherDrops {
2      int index;
3      int num;
4      int demon;
5
6      EtherDrops(int i = 0, int n = 0, int d = 0) : index(i), num(n), demon(d)
7      {};
8
9  struct edge {
10     int dest;
11     int cost;
12     edge* link;
13
14     edge(int d, int c, edge* l = nullptr) : dest(d), cost(c), link(l) {};
15 };
16
17 struct vertex {
18     edge* adj;
19
20     vertex(edge* a = nullptr) : adj(a) {};
21 };
22
23 class graph {
24 public:

```

```

25     graph(int n, int m);
26     int getweight(int u, int v);
27     void solution();
28
29 private:
30     int n, m, s, t, x0, p;
31     vertex* web;
32     EtherDrops* etherInfo;
33     int** a;
34     int* important;
35     int* dist;
36     int* parent;
37     bool* visited;
38     int path[MAX_PATH_SIZE]; // 存储路径的数组
39     int pathIndex; // 路径的当前索引
40
41     void floyd();
42     void dfs(int v, int prev, int totalCost, int currentPower);
43     int fightTime(int x, int y) {
44         return max(100 - (x - y), 0);
45     }
46 };
47

```

- 使用数组存储的最小堆 `MinHeap` 以及结构体 `HeapNode` 辅助实现Dijkstra算法

```

1  const int DefaultSize = 128;
2
3  template<class T>
4  class MinHeap
5  {
6  public:
7      MinHeap(int sz = DefaultSize);
8      MinHeap(T arr[], int n);
9      ~MinHeap() { delete[] heap; }
10     T top() { return heap[0]; }
11     bool push(T& x);
12     bool pop();
13     bool pop(T& x); template<class T>
14     class MinHeap {
15     public:
16         MinHeap(int sz = DefaultSize);
17         MinHeap(T arr[], int n);
18         ~MinHeap() { delete[] heap; }
19         T top() { return heap[0]; }
20         bool push(T& x);
21         bool pop();
22         bool pop(T& x);
23         bool empty() const { return curSize == 0; }
24         bool full() const { return curSize == maxHeapSize; }
25         void clear() { curSize = 0; }
26
27     private:
28         T* heap;
29         int curSize;

```

```

30     int maxHeapSize;
31     void siftDown(int start, int m);
32     void siftUp(int start);
33 };
34
35 struct HeapNode {
36     int vertex;
37     int distance;
38
39     HeapNode(int v = 0, int d = 0) : vertex(v), distance(d) {};
40
41     bool operator<(const HeapNode& h) const {
42         return distance < h.distance;
43     }
44
45     bool operator<=(const HeapNode& h) const {
46         return distance <= h.distance;
47     }
48
49     bool operator>(const HeapNode& h) const {
50         return distance > h.distance;
51     }
52
53     bool operator>=(const HeapNode& h) const {
54         return distance >= h.distance;
55     }
56 };
57
58 bool empty() const { return curSize == 0; }
59 bool full() const { return curSize == maxHeapSize; }
60 void clear() { curSize = 0; }
61
62 private:
63     T* heap;
64     int curSize;
65     int maxHeapSize;
66     void siftDown(int start, int m);
67     void siftUp(int start);
68 };
69
70 //最小堆的节点，成员为小岛编号和当前最短距离
71 struct HeapNode {
72     int vertex;
73     int distance;
74
75     HeapNode(int v = 0, int d = 0) : vertex(v), distance(d) {};
76
77     bool operator<(const HeapNode& h) const
78     {
79         return distance < h.distance;
80     }
81
82     bool operator<=(const HeapNode& h) const
83     {
84         return distance <= h.distance;
85     }

```

```

86
87     bool operator>(const HeapNode& h) const
88     {
89         return distance > h.distance;
90     }
91
92     bool operator>=(const HeapNode& h) const
93     {
94         return distance >= h.distance;
95     }
96 };

```

- 时间复杂度分析

- 初始化图和相关数据结构的总时间复杂度是 $O(n^2 + m + q)$
- Floyd算法的时间复杂度是 $O(n^3)$
- dfs算法最坏情况下每个节点会被访问一次，且每条边会被访问一次，时间复杂度是 $O(n + m)$
- 程序的总体时间复杂度为 $O(n^3 + n^2 + m + q)$ ，可简化为 $O(n^3)$

2. 核心代码+注释:

```

1  void graph::solution() {
2      cin >> s >> t;
3      cin >> x0 >> p;
4
5      important = new int[p + 2];
6      important[p] = s;
7      important[p + 1] = t;
8
9      etherInfo = new EtherDrops[p];
10     for (int i = 0; i < p; ++i) {
11         int idx, num, demon;
12         cin >> idx >> num >> demon;
13         etherInfo[i] = EtherDrops(idx, num, demon);
14
15         important[i] = idx;
16     }
17
18     for (int i = 0; i < m; ++i) {
19         int u, v, w;
20         cin >> u >> v >> w;
21         web[u].adj = new edge(v, w, web[u].adj);
22     }
23
24     floyd();
25     dist[s] = 0;
26     dfs(s, -1, 0, x0);
27
28     if (dist[t] == MAX) {
29         cout << -1 << endl;
30         return;
31     }

```

```

32     else {
33         // 输出最短路径
34         int node = t;
35         pathIndex = 0; // 重置路径索引
36         while (node != -1) {
37             path[pathIndex++] = node;
38             node = parent[node];
39         }
40
41         for (int i = pathIndex - 1; i >= 0; --i) {
42             cout << path[i] << " ";
43         }
44         cout << endl;
45         cout << dist[t] << endl;
46     }
47 }
48
49 void graph::floyd()
50 {
51     for (int k = 1; k <= n; k++)
52         for (int i = 1; i <= n; i++)
53             for (int j = 1; j <= n; j++)
54                 if (a[i][k] < MAX && a[k][j] < MAX && a[i][k] + a[k][j] <
a[i][j])
55                     a[i][j] = a[i][k] + a[k][j];
56 }
57
58 void graph::dfs(int v, int prev, int totalCost, int currentPower) {
59     visited[v] = true;
60
61     if (v == t) {
62         if (totalCost < dist[t]) {
63             dist[t] = totalCost;
64             parent[t] = prev;
65         }
66         visited[v] = false;
67         return;
68     }
69
70     for (edge* e = web[v].adj; e != nullptr; e = e->link) {
71         int next = e->dest;
72         int newCost = totalCost + e->cost;
73
74         if (visited[next]) continue;
75
76         // 更新战斗力
77         int newPower = currentPower;
78         for (int i = 0; i < p; ++i) {
79             if (etherInfo[i].index == next) {
80                 newPower += etherInfo[i].num;
81                 break;
82             }
83         }
84
85         // 计算战斗时间
86         int fightTimeCost = fightTime(currentPower, etherInfo[v].demon);

```

```

87         newCost += fightTimeCost;
88
89         if (newCost < dist[next]) {
90             dist[next] = newCost;
91             parent[next] = v;
92             dfs(next, v, newCost, newPower);
93         }
94     }
95
96     visited[v] = false;
97 }

```

3. OJ运行结果截图

Test #1:	score: 0	Wrong Answer	time: 0ms	memory: 3468kb
Test #2:	score: 0	Wrong Answer	time: 4ms	memory: 3500kb
Test #3:	score: 0	Wrong Answer	time: 5ms	memory: 3372kb
Test #4:	score: 0	Wrong Answer	time: 1ms	memory: 3496kb
Test #5:	score: 10	Accepted	time: 4ms	memory: 3440kb
Test #6:	score: 0	Wrong Answer	time: 231ms	memory: 4376kb
Test #7:	score: 0	Time Limit Exceeded		
Test #8:	score: 0	Time Limit Exceeded		
Test #9:	score: 0	Memory Limit Exceeded		
Test #10:	score: 0	Memory Limit Exceeded		