书面题

1.

顺序查找：顺序查找是通过按顺序遍历数组中的每个元素，与所查找的值进行比较，直到找到相应的值或遍历完整个数组为止。对于一个包含 N 个元素的数组，平均需要比较 $N/2$ 次（最坏的情况是 N 次，最好的情况是 1 次，所以平均 $N/2$ 次）。

折半查找：折半查找前提是数组已经是有序的，每次查找都会将搜索范围减半。这意味着第一次查找后，搜索范围减少为 $N/2$，第二次查找后进一步减少为 $N/4$，以此类推，直到找到所查找的值或者范围减少到0。折半查找的时间复杂度是 $O(logN)$。

假设数组大小为 $N = 10000000$ 元素。

对于顺序查找，平均需要的比较次数约为 $N/2 = 5000000$ 次。

对于折半查找，平均需要的比较次数可以用公式$\log_2 N$来计算。

在最好的情况下，折半查找和顺序查找都是 1 次查找。加速比为 1 倍。而在最坏的情况下，顺序查找需要 $N = 10000000$ 次，折半查找需要 $log_2 N = 23.25$ 次。加速比约为 430108 倍。在平均情况下，折半查找比顺序查找平均快大约215,021倍。这表明在处理大量有序数据时，折半查找的效率远远超过顺序查找，特别是在数据量庞大时差异尤为显著。

2.

题目要点:

1. 小船每次可以运送两名水手，或者一名探险家。
2. 船必须始终由至少一名水手操作。
3. 目标是让所有探险家都安全渡河，同时两名水手留在初始侧以继续操作船。

实现思路:

当士兵只有1人时，渡河次数为 4 次，即 $F(1) = 4$；
当士兵有N人时，渡河次数应该为 $F(n) = F(n-1) + 4$
最终解得 $F(n) = 4 * N$

3.

1. 前序遍历 : a, b, d, c, e, f
2. 中序遍历 : d, b, a, e, c, f
3. 后序遍历 : d, b, e, f, c, a

4.

1. 枚举所有的子图：从图 G 中枚举所有可能的大小为 k 的子图。
2. 检查每个子图是否完全：对于枚举出的每个子图，检查它是否是一个完全子图。一个大小为 k 的完全子图是指其所有的 k 个节点都彼此相连，即任意两个不同节点之间都有边。
3. 返回结果：如果在枚举的子图中找到了至少一个完全子图，则算法返回 true，表示存在大小为 k 的完备子图；否则，返回 false。

```cpp
#include <iostream>
#include <vector>

using namespace std;

// 函数用于判断是否所有选定的节点都彼此相连
bool isComplete(const vector<vector<int>>& graph, const vector<int>& nodes) {
    for (int i = 0; i < nodes.size(); ++i) {
        for (int j = i + 1; j < nodes.size(); ++j) {
            if (!graph[nodes[i]][nodes[j]]) {
                // 如果节点 i 和节点 j 之间没有边，则不是完全子图
                return false;
            }
        }
    }
    return true; // 所有选定的节点都彼此相连
}

// 主函数，用于检测图中是否存在大小为 k 的完备子图
bool hasCompleteSubgraph(const vector<vector<int>>& graph, int k) {
    int n = graph.size(); // 图中的节点数量
    vector<int> nodes(n); // 节点的索引数组
    for (int i = 0; i < n; ++i) nodes[i] = i;

    vector<int> subset; // 用于存放当前选择的子集
    function<void(int, int)> search = [&](int start, int depth) {
        if (depth == k) {
            // 当前选择的子集大小等于 k，检查是否是完备子图
            if (isComplete(graph, subset)) {
                throw true; // 使用异常机制来提前退出所有递归调用
            }
            return;
        }

        for (int i = start; i <= n - k + depth; ++i) {
            subset.push_back(nodes[i]);
            search(i + 1, depth + 1);
            subset.pop_back(); // 回溯
        }
    };

    try {
        search(0, 0);
```

```
    } catch (bool found) {
        return found;
    }
    return false;
}
```

5.

(a) 在邻接矩阵中，矩阵的行和列对应图中的顶点，如果顶点 i 和顶点 j 之间有边，则 $matrix[i][j]$（以及 $matrix[j][i]$）为 1；如果没有边，则为 0。
a
在邻接链表中，每个顶点都对应一个链表，链表中包含所有与该顶点直接相连的其他顶点。

邻接矩阵：

	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	0	1	0
c	1	0	0	0	0	0	1
d	1	1	0	0	0	1	0
e	1	0	0	0	0	0	1
f	0	1	0	1	0	0	0
g	0	0	1	0	1	0	0

邻接链表：

a: b, c, d, e

b: a, d, f

c: a, g

d: a, b, f

e: a, g

f: b, d

g: c, e

(b) 使用深度优先搜索（DFS）遍历图时，我们按照字母顺序访问未被访问过的顶点。在遍历过程中，会有两个重要的时间点：

1. 顶点第一次被发现并加入遍历栈的时间点（入栈顺序）。
2. 顶点的所有相邻顶点都被探索完毕，从栈中移除的时间点（出栈顺序）。

首先，我们初始化两个空列表，一个用于记录顶点入栈顺序（发现顺序），另一个用于记录顶点出栈顺序（终点顺序）。然后，我们将遵循以下规则进行DFS遍历：

访问当前顶点，并将其添加到入栈顺序列表中。

递归地访问当前顶点的所有未被访问的相邻顶点，选择字母顺序最小的顶点先访问。

当一个顶点的所有相邻顶点都被访问后，将它添加到出栈顺序列表中。

具体过程：

1. 访问 a (入栈 a)
2. 访问 a 的邻接点 b (入栈 b)
3. 从 b 继续，访问 b 的邻接点 d (入栈 d)
4. 从 d 继续，访问 d 的邻接点 f (入栈 f)
5. f 没有未访问的邻接点 (出栈 f)
6. 回到 d，d 也没有未访问的邻接点 (出栈 d)
7. 回到 b，b 也没有未访问的邻接点 (出栈 b)
8. 回到 a，访问 a 的下一个邻接点 c (入栈 c)
9. 从 c 继续，访问 c 的邻接点 g (入栈 g)
10. 从 g 继续，访问 g 的邻接点 e (入栈 e)
11. e 没有未访问的邻接点 (出栈 e)
12. 回到 g，g 也没有未访问的邻接点 (出栈 g)
13. 回到 c，c 也没有未访问的邻接点 (出栈 c)
14. 回到 a，所有邻接点都已经被访问 (出栈 a)

入栈（发现）顺序: a, b, d, f, c, g, e

出栈（完成）顺序: f, d, b, e, g, c, a