选择题

7

> 邻接表(node{vector<?>}) ≠ 邻接矩阵（array[n][n]）

8

二分图

节点由两个集合组成，且两个集合内部没有边的图。

二分图（Bipartite Graph）是一种特殊的图（graph），它的顶点集可以被划分为两个不相交的子集，使得每条边的两个端点分别属于这两个子集。简单来说，在二分图中，图的所有顶点可以分成两部分，且每条边都连接来自不同部分的顶点。

形式化定义：

- **G = (V, E)** 是一个图，其中 **V** 是顶点集，**E** 是边集。
- **G** 是二分图，当且仅当存在一个划分 **(V1, V2)**，使得 **V1** ∪ **V2 = V** 且 **V1** ∩ **V2 =** ∅（即 **V1** 和 **V2** 不重叠），并且对于所有的 **(u, v)** ∈ **E**，都有 **u** ∈ **V1** 且 **v** ∈ **V2** 或者 **u** ∈ **V2** 且 **v** ∈ **V1**。

二分图的性质

1. **无奇环**：二分图中不包含奇数长度的环。如果一个图中不包含奇环，那么它一定是二分图。
2. **着色**：二分图是可以用两种颜色对所有顶点进行着色的图，且相邻顶点不会有相同的颜色。

二分图的应用

- **匹配问题**：二分图常用于解决匹配问题（Matching），如最大匹配（Maximum Matching）和最大流问题中的二分图匹配。
- **分组问题**：二分图用于建模需要将元素分为两个组的情况，比如在网络中将用户和服务器分开。
- **任务分配**：在任务分配问题中，将任务和资源建模为二分图，找到最优分配方式。

如何判断一个图是否是二分图

常用的方法是**染色法**（BFS或DFS）：

- 从任意一个顶点开始，将其染成一种颜色（如红色），然后将与其相邻的所有顶点染成另一种颜色（如蓝色）。
- 对每个未被访问的顶点重复上述步骤。
- 如果过程中发现一个顶点需要染成两种不同的颜色，则图不是二分图；否则，图是二分图。

要计算 n 个顶点的二分图最多可以有多少条边，首先需要了解二分图的结构特性。二分图的顶点集可以划分为两个不相交的子集 V_1 和 V_2，并且每条边都连接来自不同子集的顶点。因此，二分图的边数受限于这两个子集中顶点的数目。

分析

设 V_1 和 V_2 是二分图的两个顶点集，其中 $|V_1| = n_1$ 和 $|V_2| = n_2$，且 $n_1 + n_2 = n$。在二分图中，所有的边都连接 V_1 中的顶点和 V_2 中的顶点，所以最大边数就是两个子集之间所有可能的连接数。

最大边数

最大边数 $E_{\max}$ 就是所有可能的边数，即：
$E_{\max} = n_1 \times n_2$

要使 $E_{\max}$ 最大化，我们需要尽量均匀地分配顶点给 V_1 和 V_2。

最大化边数的分配

为了最大化 $n_1 \times n_2$，我们应该让 n_1 和 n_2 尽量接近。数学上，给定 $n = n_1 + n_2$，如果 n 是偶数，则最佳分配是 $n_1 = n/2$ 和 $n_2 = n/2$；如果 n 是奇数，则最佳分配是 $n_1 = \lfloor n/2 \rfloor$ 和 $n_2 = \lceil n/2 \rceil$。

根据这些分配方式，我们可以总结如下：

1. **如果 n 是偶数**：

$$E_{\max} = \left(\frac{n}{2}\right) \times \left(\frac{n}{2}\right) = \frac{n^2}{4}$$

2. **如果 n 是奇数**：

$$E_{\max} = \left(\left\lfloor \frac{n}{2} \right\rfloor\right) \times \left(\left\lceil \frac{n}{2} \right\rceil\right) = \left(\frac{n-1}{2}\right) \times \left(\frac{n+1}{2}\right) = \frac{n^2 - 1}{4}$$

结论

综上所述，给定 n 个顶点的二分图最多可以有：

$$E_{\max} = \begin{cases} \frac{n^2}{4}, & \text{如果 } n \text{ 是偶数} \\ \frac{n^2-1}{4}, & \text{如果 } n \text{ 是奇数} \end{cases}$$

这意味着当 n 个顶点均匀地分配在两个子集中时，二分图能够包含最多的边。

14

迪杰斯特拉（Dijkstra）算法是一种用于计算单源最短路径的图算法。它可以找出从起始节点到其他所有节点的最短路径，是由荷兰计算机科学家艾兹赫尔·迪杰斯特拉（Edsger W. Dijkstra）于1956年提出的。

迪杰斯特拉算法的基本思想

1. **初始化**：
   - 从起点开始，设定起点的距离为0，其余所有节点的距离为无穷大。
   - 将所有节点标记为未访问。
2. **选择节点**：
   - 从未访问的节点中选择一个距离起点最近的节点（称为当前节点）。
3. **更新距离**：
   - 以当前节点为中介，更新其邻居节点的距离。
   - 如果通过当前节点可以使某个邻居节点的距离变得更短，则更新该邻居节点的距离值。
4. **标记已访问**：
   - 将当前节点标记为已访问，不再访问。
5. **重复步骤2至4**：
   - 继续选择未访问节点中距离最小的节点，重复更新操作，直到所有节点都被访问。

迪杰斯特拉算法的应用

迪杰斯特拉算法主要应用于以下场景：

- 网络路由：寻找最短路径以优化数据传输。
- 地图和导航系统：找出最短路径或最优路径。
- 资源管理：优化资源分配和调度。

伪代码

```
function Dijkstra(Graph, source):
    dist[source] ← 0
    for each vertex v in Graph:
        if v ≠ source:
            dist[v] ← INFINITY
        add v to unvisitedSet

    while unvisitedSet is not empty:
        u ← vertex in unvisitedSet with min dist[u]
        remove u from unvisitedSet

        for each neighbor v of u:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt

    return dist
```

复杂度

迪杰斯特拉算法的时间复杂度取决于实现方式。使用优先队列（二叉堆）实现的情况下，时间复杂度为 $O((V + E) \log V)$，其中 V 是节点数，E 是边数。

如果你有任何具体问题或者需要更详细的解释，请告诉我！

编程选择题

2

代码

```
#include <iostream>
#include <cstdlib>
using namespace std;

int n;
int d[10000];

int find(int L, int R, int k) {
    int x = rand() % (R - L + 1) + L;
    swap(d[L], d[x]);
    int a = L + 1, b = R;
    while (a <= b) {
        while (a <= b && d[a] < d[L])
            ++a;
        while (a <= b && d[b] >= d[L])
            --b;
        if (a < b) {
            swap(d[a], d[b]);
            ++a;
            --b;
        }
    }
    swap(d[L], d[b]);
    if (b - L + 1 == k)
        return d[b];
    if (b - L + 1 < k)
        return find(b + 1, R, k - (b - L + 1));
    return find(L, b - 1, k);
}

int main() {
    int k;
    cin >> n;
    cin >> k;
```

```
    for (int i = 0; i < n; ++i)
        cin >> d[i];
    cout << find(0, n - 1, k) << endl;
    return 0;
}
```

这段代码实现了**快速选择算法**（Quickselect），用于在未排序的数组中查找第 `k` 小的元素。

代码解释

1. **输入部分：**
   - `n` 是数组 `d` 的大小。
   - `k` 是我们要找的第 `k` 小的元素。
   - 用户从输入中读取 `n` 和 `k`，然后读取 `n` 个整数填充数组 `d`。
2. **`find` 函数：**
   - 这个函数通过递归的方式实现了快速选择算法。其参数 `L` 和 `R` 定义了搜索的区间，`k` 是要查找的第 `k` 小元素。
3. **算法的核心逻辑：**
   - 随机选取一个枢轴（`x`），将这个枢轴的值与 `d[L]` 交换，使得枢轴在 `d[L]` 处。
   - 使用两个指针 `a` 和 `b`，分别从 `L+1` 向右和从 `R` 向左遍历，重排元素使得枢轴左边的元素都小于它，右边的元素都大于它。
   - 最后，交换 `d[L]` 和 `d[b]`，使得 `d[b]` 成为枢轴元素，并且 `d[b]` 的位置是它在排序后应有的位置。
   - 根据枢轴的位置 `b` 与 `k` 的关系：
     - 如果 `b - L + 1 == k`，那么 `d[b]` 就是我们要找的第 `k` 小的元素。
     - 如果 `b - L + 1 < k`，则说明第 `k` 小的元素在枢轴右侧，因此递归搜索 `b + 1` 到 `R` 区间，寻找第 `k - (b - L + 1)` 小的元素。
     - 如果 `b - L + 1 > k`，则第 `k` 小的元素在枢轴左侧，因此递归搜索 `L` 到 `b - 1` 区间。
4. **程序的执行流程：**
   - 主函数 `main` 读取输入后调用 `find` 函数，输出结果即为数组中第 `k` 小的元素。

代码功能总结

该代码的主要功能是在一个未排序的数组中寻找第 `k` 小的元素，使用快速选择算法，它的平均时间复杂度为 `O(n)`。

如果数组中的数字是**单调递增**或**单调递减**，`find` 函数仍然可以正确地找到第 `k` 小的元素，但在最坏情况下，算法的性能可能会变差。

单调递增的情况

在数组单调递增的情况下：

1. 假设数组是 `[1, 2, 3, ..., n]`，且我们要找第 `k` 小的元素。
2. 在每次调用 `find` 函数时，随机选择的枢轴 `x` 如果总是选择 `d[L]` 或者接近 `d[L]`，将会导致以下情况：
   - 枢轴的值最小，每次都会将整个数组的一部分分给左边，右边是空的（`a` 会从 `L+1` 一路增加到 `R+1`）。
   - 因此，每次递归只减少一个元素，导致 `find` 函数的递归深度达到 `n`，退化成类似于选择排序的行为，时间复杂度变为 `O(n^2)`。

单调递减的情况

在数组单调递减的情况下：

1. 假设数组是 `[n, n-1, n-2, ..., 1]`，且我们要找第 `k` 小的元素。
2. 每次调用 `find` 函数时，随机选择的枢轴 `x` 如果总是选择 `d[L]` 或者接近 `d[L]`，会有类似单调递增的情况：
   - 枢轴的值最大，所有其他元素都被分到左边，右边为空（`b` 会从 `R` 一路减少到 `L`）。
   - 每次递归仍然只减少一个元素，导致递归深度达到 `n`，时间复杂度也退化为 `O(n^2)`。

总结

这份代码针对性地说，单调递增是$O(n \log n)$

虽然快速选择算法在随机数据或一般情况下具有 `O(n)` 的平均时间复杂度，但在输入数据是**单调递增**或**单调递减**且每次选择的枢轴都很不理想（例如总是选到最小或最大值）时，算法可能退化到最坏的情况，时间复杂度变为 `O(n^2)`。

要改善这种情况，可以采用随机化策略选择枢轴（如现在代码中所做的）或者使用“三点取中”的方式选择枢轴，以减少最坏情况发生的概率，从而保证算法的效率。

完善程序

分数背包

> 辗转相除法，最大公约数名曰gcd