中南大学
CENTRAL SOUTH UNIVERSITY

算法分析与实验设计
实验报告（一）

学　　院：计算机学院

专业班级：

学生姓名：R

学　　号：

指导教师：

年 月 日

目录

一

1. 问题

以 O(n)的复杂度在无序数组中查找第 k 大的数，分析算法的复杂度（50 分）

输入：

第一行输入 m, k，表示输入的数组有 m 个数，查找第 k 大的数

第二行输入 m 个整数

输出:

第 k 大的数

例子：

输入：

3, 2

4, 1, 3

输出：

3

2. 代码

```
#include <stdio.h>

// 快速排序函数
void quickSort(int arr[], int left, int right) {
    if (left < right) {
        int pivot = arr[left];   // 选择左边的元素作为枢纽元素
        int i = left, j = right;
        while (i < j) {
            // 从右边开始，找到第一个小于等于枢纽元素的元素
            while (i < j && arr[j] > pivot) {
                j--;
            }
            if (i < j) {
                arr[i++] = arr[j];   // 将找到的元素放到左边
            }
            // 从左边开始，找到第一个大于枢纽元素的元素
            while (i < j && arr[i] < pivot) {
                i++;
            }
            if (i < j) {
                arr[j--] = arr[i];   // 将找到的元素放到右边
            }
        }
        arr[i] = pivot;   // 将枢纽元素放到正确的位置
        // 递归调用，对枢纽元素左右两侧的子数组进行排序
        quickSort(arr, left, i - 1);
        quickSort(arr, i + 1, right);
    }
}

// 划分函数，用于快速排序中确定枢纽元素的位置
int partition(int arr[], int left, int right) {
    int pivot = arr[left];   // 选择左边的元素作为枢纽元素
    int i = left, j = right;
    while (i < j) {
        // 从右边开始，找到第一个小于等于枢纽元素的元素
        while (i < j && arr[j] > pivot) {
            j--;
```

```
        }
        if (i < j) {
            arr[i++] = arr[j];    //  将找到的元素放到左边
        }
        //  从左边开始，找到第一个大于枢纽元素的元素
        while (i < j && arr[i] < pivot) {
            i++;
        }
        if (i < j) {
            arr[j--] = arr[i];    //  将找到的元素放到右边
        }
    }
    arr[i] = pivot;    //  将枢纽元素放到正确的位置
    return i;
}

//  查找第 k 大的元素
int findKthLargest(int arr[], int left, int right, int k) {
    int index = partition(arr, left, right);    //  使用划分函数确定枢纽元素的位置
    if (index == k - 1) {
        return arr[index];    //  找到第 k 大的元素
    } else if (index < k - 1) {
        return findKthLargest(arr, index + 1, right, k);    //  在右侧递归查找第 k 大
的元素
    } else {
        return findKthLargest(arr, left, index - 1, k);    //  在左侧递归查找第 k 大的
元素
    }
}

int main() {
    int m, k;
    int a[100000];
    scanf("%d%d", &m, &k);    //  输入数组大小和 k 的值
    for (int i = 0; i < m; i++) {
        scanf("%d", &a[i]);    //  输入数组元素
    }
    quickSort(a, 0, m - 1);    //  对数组进行快速排序,排好后再给 findKthLargest
    int kthLargest = findKthLargest(a, 0, m - 1, m - k + 1);    //  查找第 k 大的元素
    printf("%d\n", kthLargest);    //  输出结果
    return 0;
}

//此代码中使用的是快速排序算法和查找第 k 大元素算法
```

```
//它们的时间复杂度分别为O(nlogn)和O(n)
//因此整个代码的时间复杂度为O(nlogn)
//其中，快速排序的平均时间复杂度为O(nlogn)，最差情况下为O(n^2)
```

3. 分析

该代码使用了快速排序算法和查找第 k 大元素算法。快速排序算法的时间复杂度为 O(nlogn)，其中 n 为数组的大小。查找第 k 大元素的算法通过使用快速排序的思想，将数组划分为左右两个部分，并根据划分后枢纽元素的位置来确定第 k 大元素所在的区间，从而进行递归查找。

另外，该代码还使用了一个划分函数，用于确定快速排序中枢纽元素的位置。划分函数的实现与快速排序算法中的部分代码相似，它通过不断交换元素的方式将小于枢纽元素的元素放到左边，大于枢纽元素的元素放到右边，并最终确定枢纽元素的位置。

综上所述，该代码实现了快速排序和在无序数组中查找第 k 大的数的功能。

复杂度分析：

对于代码中的快速排序部分，其时间复杂度取决于递归调用的次数和每次递归调用的操作复杂度。

在每次递归调用中，通过选择一个枢纽元素（pivot）将数组划分为两部分，左边的元素小于等于枢纽元素，右边的元素大于枢纽元素。然后，对左右两部分分别进行递归调用。划分操作的时间复杂度为 O(n)，其中 n 为当前划分的数组大小。

在最坏情况下，快速排序的递归深度为 n，即每次只能划分出一个元素，这时时间复杂度为 O(n^2)。然而，快速排序其平均时间复杂度为 O(nlogn)。

对于查找第 k 大元素的算法，它基于快速排序的思想。在每次递归调用中，通过调用 partition 函数将数组划分为两部分，并确定枢纽元素的位置。然后，根据枢纽元素的位置和 k 的大小关系，决定继续在左侧或右侧递归查找。归调用的次数取决于 k 和数组的大小，每次递归都会将数组的大小减半，因此，递归调用的次数最多为 logn。因为在 partition 函数中需要遍历一次数组，所以每次递归调用的时间复杂度为 O(n)。

综上所述，整段代码的时间复杂度为 O(nlogn)。

4. 测试

图 1：按样例输入

图 2：以 20 个随机数进行测试

二

1. 问题

给定两个序列 X[1..n]和 Y[1..m]，求解 X 和 Y 的最长公共子序列，输出所有可能的公共子序列（50 分）

输入：

第一行输入序列 X

第二行输入序列 Y

输出:

第一行输出最长公共子序列的长度

接下来每一行输出一个可能的公共子序列，直到输出所有公共子序列

例子：

输入：

ABCD

BAD

输出：

2

BD

AD

2. 代码

```
#include <stdio.h>
#include <string.h>

// 定义长整型
#define ll long long

// 定义常量
#define Size 1010
const int N = 3333;
const int MOD = 1e9;

// 大整数结构体
struct BigInt {
    ll *s; // 数组指针，用于存储大整数的每一位
    int c; // 有效位数

    // 初始化大整数
    void init() {
        s = new ll[20]; // 分配空间
        for (int i = 0; i < 20; i++)
            s[i] = 0; // 初始化每一位为 0
        c = 0; // 有效位数为 0
    }

    // 大整数与整数相加
    void add(int x) {
        s[0] += x; // 将整数加到最低位
        int i = 0;
        while (s[i] >= MOD) {
            s[i + 1] += s[i] / MOD; // 进位
            s[i] %= MOD; // 取模
            i++;
        }
        while (s[c + 1])
            c++; // 更新有效位数
    }

    // 大整数与另一个大整数相加
    void add(const BigInt &x) {
        int r = (c > x.c) ? c : x.c; // 取两个大整数有效位数的较大值
```

```
        for (int i = 0; i <= r; i++) {
            s[i] += x.s[i]; // 对应位相加
            if (s[i] >= MOD) {
                s[i + 1] += s[i] / MOD; // 进位
                s[i] %= MOD; // 取模
            }
        }
        c = (19 < r + 1) ? 19 : (r + 1); // 更新有效位数
        while (c && s[c] == 0)
            c--; // 去除高位的 0
    }
};

// 字符串匹配自动机
struct ZXLZDJ {
    int ch[N][58]; // 状态转移数组，ch[i][j]表示状态 i 在字符 j 下一个状态的编号

    // 构建自动机
    void build(char *s, int Len) {
        for (int i = Len; i; i--) {
            for (int j = 0; j < 58; j++)
                ch[i - 1][j] = ch[i][j]; // 复制上一个状态的转移信息
            ch[i - 1][s[i] - 'A'] = i; // 更新当前状态的转移信息
        }
    }

    // 重载[]运算符，用于快速访问状态转移数组
    int *operator[](const int &i) { return ch[i]; }
} A, B;

bool vis[N][N]; // 记录状态是否已访问
char s1[N], s2[N]; // 输入字符串
int n, m, op; // 输入字符串的长度
char ans[N]; // 存储结果

// 深度优先搜索，用于输出所有 LCS
void dfs(int u, int v, int tt) {
    ans[tt] = '\0'; // 在结果末尾添加字符串结束符
    if (strlen(ans + 1) > 1)
        printf("%s\n", ans + 1); // 输出 LCS
    for (int i = 0; i < 58; i++) {
        if (A[u][i] == 0 || B[v][i] == 0)
            continue; // 当前字符不存在于两个字符串中
        ans[tt] = i + 'A'; // 将字符添加到结果中
```

```
        dfs(A[u][i], B[v][i], tt + 1); // 继续搜索下一个状态
    }
}

// 最长公共子序列长度和方向数组
int DP[Size][Size];
int DIR[Size][Size];

// 求解最长公共子序列的长度
int LCS_length(char *a, char *b) {
    int M = strlen(a); // 字符串 a 的长度
    int N = strlen(b); // 字符串 b 的长度
    for (int i = 1; i <= M; i++) {
        for (int j = 1; j <= N; j++) {
            if (a[i - 1] == b[j - 1]) {
                DP[i][j] = DP[i - 1][j - 1] + 1; // 当前字符相等，LCS 长度加 1
                DIR[i][j] = 1; // 方向标记为 1，表示斜向上
            } else if (DP[i - 1][j] >= DP[i][j - 1]) {
                DP[i][j] = DP[i - 1][j]; // 当前字符不相等，LCS 长度不变，选择上方的状
态
                DIR[i][j] = 2; // 方向标记为 2，表示向上
            } else {
                DP[i][j] = DP[i][j - 1]; // 当前字符不相等，LCS 长度不变，选择左方的状
态
                DIR[i][j] = 3; // 方向标记为 3，表示向左
            }
        }
    }
    return DP[M][N]; // 返回最长公共子序列的长度
}

int main() {
    scanf(" %s", s1 + 1); // 输入字符串 1
    scanf(" %s", s2 + 1); // 输入字符串 2
    n = strlen(s1 + 1); // 计算字符串 1 的长度
    m = strlen(s2 + 1); // 计算字符串 2 的长度
    printf("%d\n", LCS_length(s1 + 1, s2 + 1)); // 输出最长公共子序列的长度
    A.build(s1, n); // 构建字符串 1 的自动机
    B.build(s2, m); // 构建字符串 2 的自动机
    dfs(0, 0, 1); // 深度优先搜索输出所有 LCS
    return 0;
}
//求最长公共子序列长度的时间复杂度为 o(m*n),空间复杂度为 o(m*n)
//求两个字符串的所有公共子序列的时间复杂度为 o(2^k),空间复杂度为 o(k).其
```

中 k 表示最长公共子序列的长度。

3. 分析

该程序使用了动态规划算法来计算最长公共子序列（LCS）的长度，并使用深度优先搜索算法来输出所有的最长公共子序列。

具体来说，使用动态规划算法求解 LCS 的长度，采用了一个二维的 DP 数组（DP[Size][Size]），其中 DP[i][j]表示字符串 A 的前 i 个字符和字符串 B 的前 j 个字符的 LCS 的长度。通过填充 DP 数组，可以逐步推导出最终的 LCS 长度。动态规划的状态转移方程如下：

若 A[i-1] == B[j-1]，则 DP[i][j] = DP[i-1][j-1] + 1（当前字符相等，LCS 长度加 1）
否则，DP[i][j] = max(DP[i-1][j], DP[i][j-1])（当前字符不相等，取上方或左方的最大值）

然后，利用 DIR 数组（DIR[Size][Size]）记录状态转移的方向，用于后续输出 LCS 时的回溯。DIR[i][j]的取值有三种情况：

1：表示从左上方（斜上方）转移得到 DP[i][j]
2：表示从上方转移得到 DP[i][j]
3：表示从左方转移得到 DP[i][j]

最后，使用深度优先搜索算法进行回溯，输出所有的 LCS。从 DIR 数组的右下角开始，根据 DIR 数组的指示，不断向左上方或上方或左方进行搜索，并将经过的字符添加到结果字符串中，直到搜索到左上角（DIR[0][0]）为止，即找到了一个完整的 LCS。

综上所述，该程序结合了动态规划和深度优先搜索算法来求解最长公共子序列，并输出所有的最长公共子序列。

复杂度分析：

字符串长度获取：获取输入字符串的长度，时间复杂度为 O(n+m)，其中 n 和 m 分别是输入字符串的长度。

构建自动机：构建两个字符串的自动机，时间复杂度为 O(n+m)。在构建自动机过程中，需要遍历字符串中的每个字符，将字符转化为对应的状态编号，因此时间复杂度与字符串的长度相关。

动态规划求解 LCS 长度：使用动态规划的方法计算两个字符串的最长公共子序列的长度，时间复杂度为 O(nm)，其中 n 和 m 分别是输入字符串的长度。在动态规划的过程中，需要计算 DP 数组的每个元素，每个元素的计算只依赖于前

面的元素，因此总共需要计算 nm 个元素。

深度优先搜索输出所有 LCS：通过深度优先搜索的方式输出所有的最长公共子序列，时间复杂度取决于最长公共子序列的数量。在最坏情况下，最长公共子序列的数量为指数级别，因此时间复杂度为 O(2^k)，其中 k 是最长公共子序列的数量。

所以，求最长公共子序列长度的时间复杂度为 o(m*n),空间复杂度为 o(m*n),求两个字符串的所有公共子序列的时间复杂度为 o(2^k),空间复杂度为 o(k).其中 k 表示最长公共子序列的长度。

4. 测试

图 3：按样例输入（输出的顺序与样例不一样，但题目并未要求输出顺序）

图 4：测试

三

1. 问题

有 n 项工作，工作 j 的开始时间是 sj，结束时间是 fj，完成工作 j 获得的报酬是 wj； 如果两项工作的时间没有重叠，则同一个人可以完成两项工作；目标：在同一个人可以完成的工作中，找出所获报酬最大的工作集合；

输入：

第一行输入工作的数量

第二行开始，每行输入一个工作编号和对应的报酬、开始时间以及结束时间，以空格隔开，时间按 XX:YY:ZZ 的格式

输出：

第一行输出所获报酬最大的工作编号集合

第二行输出对应的最大报酬

例子：

输入：

3
1 10 08:00:00 09:00:00
2 8 08:30:00 9:30:00
3 6 09:10:00 10:10:00

输出：

{1,3}
16

2.代码

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct Work { int id; int start; int end; int reward; };

int max(int a, int b){ return a > b ? a : b; }

int main() {
    int n; scanf("%d", &n); // 读取任务数量
    struct Work works[n]; // 声明一个包含 n 个 Work 结构体的数组

    // 读取每个任务的输入并初始化 works 数组
    for (int i = 0; i < n; i++) {
        int id, r, start_h, start_m, start_s, end_h, end_m, end_s;
        scanf("%d %d %d:%d:%d %d:%d:%d", &id, &r, &start_h, &start_m,
&start_s, &end_h, &end_m, &end_s);
        int start = start_h * 60 + start_m; // 将开始时间转换为分钟表示
        int end = end_h * 60 + end_m; // 将结束时间转换为分钟表示
        works[i].id = id;
        works[i].start = start;
        works[i].end = end;
        works[i].reward = r;
    }

    int pre[n], dp[n];
    memset(pre, -1, sizeof(pre)); // 初始化 pre 数组，表示每个任务的前一个任
务索引
    memset(dp, 0, sizeof(dp)); // 初始化 dp 数组，表示到达每个任务时的最大
奖励
    dp[0] = works[0].reward; // 第一个任务的最大奖励为其自身的奖励

    // 动态规划计算最大奖励
    for (int i = 1; i < n; i++) {
        dp[i] = works[i].reward; // 初始化当前任务的最大奖励为其自身的奖励
        for (int j = 0; j < i; j++) {
            if (works[j].end <= works[i].start) { // 如果任务 j 的结束时间早于等
于任务 i 的开始时间
                // 更新最大奖励，并记录前一个任务索引
                dp[i] = max(dp[i], dp[j] + works[i].reward);
```

```
                    if (dp[i] == dp[j] + works[i].reward) {
                        pre[i] = j; // 更新前一个任务索引为 j
                    }
                }
            }
            if (dp[i] < dp[i - 1]) { // 如果当前任务的最大奖励小于前一个任务的最大
奖励
                dp[i] = dp[i - 1]; // 取前一个任务的最大奖励作为当前任务的最大
奖励
                pre[i] = pre[i - 1]; // 更新前一个任务索引
            }
        }

        int maxSet[n], cnt = 0, idx = n - 1;

        // 构建最大奖励的任务集合
        while (pre[idx] != -1) {
            maxSet[cnt++] = works[idx].id;
            idx = pre[idx];
        }
        maxSet[cnt++] = works[idx].id;

        printf("{");
        for (int i = cnt - 1; i >= 0; i--) {
            printf("%d", maxSet[i]);
            if (i > 0) {
                printf(",");
            }
        }
        printf("}\n%d", dp[n - 1]); // 输出最大奖励及任务集合

        return 0;
}
```

3.分析

程序是使用动态规划解决任务调度问题的。它接受输入来表示一系列任务，包括任务的 ID、奖励以及开始和结束时间，并计算选择一组不重叠任务所能获得的最大奖励。

下面是代码的详细解释：

包含了必要的头文件：stdio.h、stdlib.h 和 string.h。
定义了一个结构体 Work，用于表示每个任务，包含 ID、开始时间、结束时间和奖励等字段。
max 函数是一个实用函数，返回两个整数中的最大值。
main 函数。

在 main 函数中：

从用户输入中读取任务数量 n。
声明一个包含 n 个 Work 结构体的数组。
读取每个任务的输入，包括任务的 ID、奖励、开始时间和结束时间。开始和结束时间被转换为分钟，并存储在相应的 Work 结构体的 start 和 end 字段中。
初始化两个数组 pre 和 dp，并将其元素置为初始值。
对于每个任务，计算选择该任务时可以获得的最大奖励，并记录其前一个任务的索引。
计算到达每个任务时的最大奖励，并记录选择的任务的前一个任务索引。
构建最大奖励的任务集合，并将其输出。
总体而言，该程序使用动态规划方法解决了任务调度问题，找到了一组不重叠的任务，使得总奖励最大化。

算法复杂度分析：

读取输入：O(n)

读取任务数量 n：O(1)
读取每个任务的输入：O(n)
初始化数组：O(n)

初始化 pre 数组：O(n)
初始化 dp 数组：O(n)
动态规划计算最大奖励：O(n^2)

外层循环：对于每个任务 i，执行一次循环，共需 O(n) 的时间。

内层循环：对于每个任务 i，遍历之前的任务 j，执行一次循环，共需 O(n) 的时间。

因此，动态规划计算的总时间复杂度为 O(n^2)。

构建最大奖励的任务集合：O(n)

最大奖励的任务集合的大小最多为 n，因此构建集合的时间复杂度为 O(n)。

输出结果：O(n)

输出最大奖励的任务集合：需要遍历集合中的每个元素，因此时间复杂度为 O(n)。

输出最大奖励的结果值 dp[n-1]：O(1)。

综上所述，总体时间复杂度为 O(n^2)，空间复杂度为 O(n)。其中 n 是任务的数量。

4.测试

图 5：按样例测试

图 6：随机测试