实验一

一、 实验题目：贪心算法——编程实现最小生成树 MST 算法

二、实验目的与内容

1、实验目的：

1) 深入了解最小生成树的有关性质与经典算法。
2) 学习贪心算法的设计方法与实现方法。
3) 在具体实现时，学习如何使用的具体的数据结构完成一些算法操作（例如本问题中的 union 和 find）
4) 分析不同的最小生成树算法的性能差异。

2、实验内容：

1) 设计并编程实现最小生成树算法
   a) 输入：第一行有两个数，给出图的点数量 n 和边数量 m；接着有 m 行，每一行有三个数字，分别是边的两个端点 u 和 v，以及其对应的边权 w_i。
   b) 输出：输出边权和的最小值。
2) 自行设计测试用例测试数据，测试程序的正确性。
3) 分析算法的正确性与复杂度，并分析 kruskal 和 prim 算法的性能差异。

三、算法设计

1、算法描述

实验中我们采用 kruskal 算法，每次加入最小边权的边，并保证不成环，直到已经加入的边的数量已经到了 n-1 为止（n 为点的数量）。为了表示每个点的加入情况与连通情况，我们利用并查集标识每一个节点的从属关系，并用路径压缩技术减少并查集的查找时间。

下面用分步骤解释算法的具体实现：

STEP 1: 对输入的边权从小到大排序。

在这里我们可以调用 c++的 algorithm 库中的 sort 函数，这个函数采用的是快排的方法，复杂度为 O(nlogn)。

STEP 2: 初始化并查集，每个元素先各自为一个独立的连通块。

这里我们将我们所设置的并查集全部设置为 root[v]=v，即每个点的根节点都是自己，也就是各自为一个块。（并查集中把 root 一样的点称为一个块）

STEP 3: 按边权从小到大枚举所有边，直到加入 n-1 个边或者 m 条边循环完毕，跳转到 STEP 6。

我们开始循环，初次循环的时候，初始化 edgenum=0 记录已经加入的边。执行下列步骤。

STEP 4: 判断当前所判断的边的两个端点是否在一个测试块中，若是则合并。

这一步要实现并查集的 findroot 函数，我们可以采用递归的方式实现这个函数，具体实现如下：

$$findroot(v) = \begin{cases} v & if(root[v] = v) \\ findroot(root[v]) & if(root[v]! = v) \end{cases}$$

然而这种方法效率是比较低的，显然，我们访问过的递归链路上的 root 都应该是点 v 的 root，而不幸运的是，我们只将 v 节点的 root 进行了更新，没有将其他节点进行更新。所以，在这里提出路径压缩，就是保存下根节点的数值，将链路上的所有节点的更节点在访问过程中进行更新。我们只需要在递归过程中的时候插入以下逻辑即可：

$$root[v] = findroot(root[v]) \quad if(root[v]! = v)$$

要合并则要实现并查集的 union 函数，对于一条边的两个端点 u 和 v，我么先调用 findroot 函数寻找到了两个点的 root，分别记为 uroot 和 vroot，则有合并规则如下：

$$root[uroot] = vroot \quad if(uroot! = vroot)$$

可以看出，若两点 root 不同，将 uroot 合到 vroot 上，这里说明，由于并查集经过了路径压缩，所以将 vroot 合到 uroot 上也是一样的复杂度。我们在程序中统一向 vroot 合并。

STEP 5: 增加边权 sum，循环 STEP 3 到 STEP 5。

若上一步合并成功（即 uroot=vroot），将 sum 加上该边的权值。（sum 初始化为 0），edgenum 加 1。循环 STEP 3 到 STEP 5，一直到 edgenum 等于 n-1 或者 m 条边循环完毕，进行 STEP 6。

STEP 6: 输出结果

判断 edgenum 是否等于 n-1，若不等，证明图不连通，显示错误；若相等，输出 sum，即最小权值和。

总结上面所给出的步骤，算法伪代码如下：

```
krukal( ) {
    sum=最小生成树的边权之和  (init sum=0);
    edgenum=当前加入最小生成树的边的数量  (init edgenum=0);
    root[ ]=所有点的并查集  (init root[v]=v (v=0 to n-1 )
    for(从小到大枚举输入的边){
        if(当前所枚举的边的两个端点的 root 不同){
            该边加入最小生成树(合并并查集);
            sum=sum+边权;
```

```
                edgenum=edgenum+1;
            }
        }
        return sum;
}
```

2、算法流程图

算法描述的流程图如图 3.1：

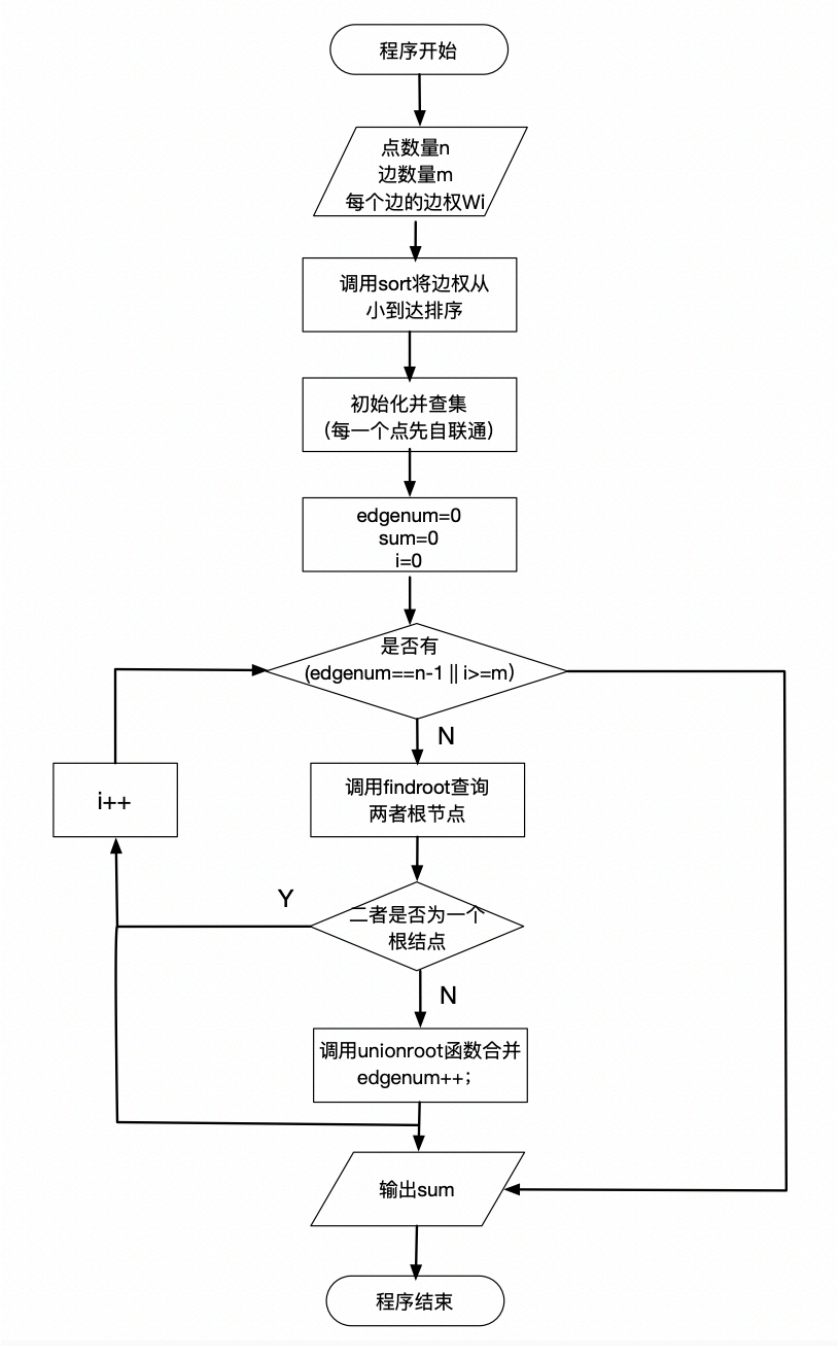

图 3.1 kruskal 算法流程图

四、实验环境

操作系统：Windows 10
编译环境：Dev C++ 5.11
编译器　：TDM-GCC 4.9.2 64-bit Release

五、实验过程

运用 kruskal 算法实现 MST 的源代码如下，每段程序代表的释义、每个变量的意义已经用注释详细标明，可以对照第三章算法设计进行对照查看：

kruskal 实现 MST

```
#include <iostream>
#include <algorithm>
#define Vmax 1000
#define Emax 500000
using namespace std;

struct edge{
    int u,v,w;   //u:一个端点；v:另一个端点；w:边权
}E[Emax];

bool cmp(edge a,edge b){
    return a.w<b.w;   //边按照从小到大排序
}

int m,n; //m:边的条数，n:点的个数
int sum,edgenum; //sum:边权和，edgenum:已经遍历的边的数量

//利用并查集实现 findroot 函数
int root[Vmax];
int findroot(int v){
    if(v==root[v]){   //找到了根节点
        return v;
    }
    //路径压缩
    else{
        int Root=findroot(root[v]); //寻找上一级的根节点
        root[v]=Root;   //更新路上所有节点接入根节点
        return Root;
    }
}
//实现 unionroot 函数(对于编号为 i 的边)
int unionroot(int i){
```

```
    int uroot=findroot(E[i].u);
    int vroot=findroot(E[i].v);
    if(uroot!=vroot){
        root[uroot]=vroot;
        sum=sum+E[i].w;
        edgenum=edgenum+1;
    }
}

int main(){
    //读入边数、点数、边的信息
    cin>>m>>n;
    for(int i=0;i<m;i++){
        cin>>E[i].u>>E[i].v>>E[i].w;
    }

    for(int i=0;i<n;i++){ //初始化并查集
        root[i]=i;
    }

    sort(E,E+m,cmp); //边从小到大排序

    for(int i=0;i<m;i++){ //遍历边，进行合并操作
        unionroot(i);
        if(edgenum==n-1){   //边的数量已经达到 n-1
            break;
        }
    }
    if(edgenum!=n-1){   //若图不连通
        cout<<"MST does not exist!"<<endl;
    }
    else{
        cout<<"The minimal sum is: "<<sum<<endl;
    }
    return 0;
}
```

六、算法测试

为了保证算法的正确性，我们设置三组测试样例，设置意义如下（我们的输入输出已经在第二节实验内容中作出了约定）：

1) 第一组，设置普通算例，6 个顶点、10 条边，没有重复的边权和为 0 的边权。
2) 第二组，测试特殊情况，4 个顶点、6 条边，有重复的边权。
3) 第三组，测试更边界的情况，4 个顶点、6 条边，有重复边权以及为 0 的边权。

测试样例 1

样例输入	理论输出	样例输出
6 10 0 1 4 0 4 1 0 5 2 1 2 1 1 5 3 2 3 6 2 5 5 3 4 5 3 5 4 4 5 3	11	6 10 0 1 4 0 4 1 0 5 2 1 2 1 1 5 3 2 3 6 2 5 5 3 4 5 3 5 4 4 5 3 the minimum sum of weights is: 11

由于理论输出与样例输出相符，所以测试样例 1 验证成功。

测试样例 2

样例输入	理论输出	样例输出
4 6 0 1 1 0 2 3 0 3 1 1 2 2 1 3 2 2 3 2	4	4 6 0 1 1 0 2 3 0 3 1 1 2 2 1 3 2 2 3 2 the minimum sum of weights is: 4

由于理论输出与样例输出相符，所以测试样例 2 验证成功。

测试样例 3

样例输入	理论输出	样例输出
4 6 0 1 0 0 2 3 0 3 3 1 2 2 1 3 2 2 3 0	2	4 6 0 1 0 0 2 3 0 3 3 1 2 2 1 3 2 2 3 0 the minimum sum of weights is: 2

由于理论输出与样例输出相符，所以测试样例 3 验证成功。

综上，算法通过所有样例的测试。

七、结果分析

1. 算法正确性证明

1) 证明该算法之前，先证明下面这个引理：

对于任何点集合 S 和 $V-S$，这两个点集合中权值最小的割边 $e=(v,w)$一定属于最小生成树。

证明如下：假设我们现有的生成树为T，对于任意点集合 V 和 V-S，设其割边为 $e=(v,w)$。假设有另一割边 $e'=(v',w')$,使得 $cost(e')\leq cost(e)$。我们可以构造一棵新的生成树 $T'=T-\{e\}+\{e'\}$。这棵树的权值和小于原生成树的权值和，即：$cost(T')=cost(T)-cost(e)+cost(e')\leq cost(T)$。也就是说，新树的总权值更优。现在证明新树能从点 v 路由到 w：在 S 和 $V-S$ 中，必然各自有路径满足从 v 到 v' 和从 w 到 w' 。所以，我们分析出 v 到 w 会有新路径：$(v,v')+e'+(w,w')$，满足 v 到 w 连通，故我们的新树是一棵生成树且权值更小，更接近最小生成树。综上，在最小生成树中，割边中权值最小的一定包含在最小生成树中。

2) 下面利用这一引理来证明 kruskal 算法：

对于 kruskal 添加的任何边 $e=(v,w)$ ，令 S 为 v 在添加 w 之前的点的集合，因为由 kruskal 算法性质有，加上边 e 不会产生环，即有 $v\in$ S且w $\notin$ S。而且没有从 S 到 $V-S$ 的边。那么，由 kruskal 算法添加，添加的边 e 一定是割边中最小的。由上面引理可知，该边属于最小生成树。即我们可以通过 kruskal 算法生成边构造最小生成树。

2. 算法复杂度分析

1) 我的算法的复杂度

该算法的算法主要分为两个方面，一个是对于边权的排序，另一个是 findroot 和 unionroot 的并查集操作过程。

对于边权的排序，在函数实现的时候是调用的 STL 的 algorithm 库中的 sort 函数，采用的是快速排序的方法，算法复杂度为 O(mlogm)。

对于并查集的 find 和 union 过程，每一次遍历的时候都会采取压缩策略，即压缩后，前序点的搜索都会变成 O(1)，降低了一部分复杂度，复杂度会从本身的 O(mlogm)（这里是一棵平衡树）变成了老师上课提到的 O(mα(m))。

所以算法的总复杂度为：

$$T(n)=O(mlogm)+O\big(m\alpha(m)\big)=O(mlogm)$$

2) 还可以用最小堆对于上面算法的复杂度进一步优化

我们观察上面复杂度也可以发现，这里的 sort 算法拖慢了并查集操作好不容易节省下来的复杂度。

所以我们也可以采用建最小堆的方式，每一次并查集操作时弹出堆顶元素，这样不用排序就可以完成由小到大取遍历边，建堆的复杂度仅为 O(m)，小于排序复杂度。这种情况下，复杂度为：

$$T(n)=O(m)+O\big(m\alpha(m)\big)=O\big(m\alpha(m)\big)$$

3. 关于 prim 和 krukal 对于 MST 问题的的性能简析

对于 prim 问题，是以点出发，每次寻找最小的不成环边，并将最小的边加入，对应的点加入集合，直到找到了 n 个点。故我们需要遍历 n 个点，遍历 n 个点的时候，每一次都要遍历 O(n)个点，故我们有，这样实现 prim 算法的复杂度为 $O(n^2)$。

对于 kruskal，我们上面已经提到了一般而言（不用最小堆情况下），有复杂度 O(mlogm)。

对于 m 和 n，我们有以下关系：

$$m \leq \frac{n(n-1)}{2}$$

如果是稠密图，比如我们取 m=n(n-1)/2 代入 kruskal 的复杂度，有：

$$\begin{aligned} T_{kruskal}(n) &= O(mlogm) \\ &= O\left(\frac{n(n-1)}{2}\log\left(\frac{n(n-1)}{2}\right)\right) \\ &= O\big((n^2-n)\log(n^2-n)\big) \\ &= O(n^2 logn^2) \\ &= O(n^2 logn\,) \end{aligned}$$

可以看出，在该稠密图中，kruskal 算法的复杂度是 prim 算法的 logn 倍。

如果是稀疏图，比如我们取 m=n 带入 kruskal 的复杂度，则有：

$$T_{kruskal}(n) = O(mlogm) = O(nlogn)$$

这个复杂度是要比 prim 算法的 $O(n^2)$要小的。

所以，我们可以看出 kruskal 适合解稠密图，prim 适合解稀疏图。

八、总结

在本次实验中，我深化了对最小生成树的算法的理解，并且利用了并查集、路径压缩等知识实现了这个算法，最后通过分析我所实现的 MST 的正确性与复杂度，进一步复习了算法正确性证明与复杂度分析的知识。最后，我还探究了如何在稀疏图和稠密图下 kruskal 算法和 prim 算法的性能差异。

实验中遇到的一个难点就是实现老师上课所提到的路径压缩，后来通过在网上和书籍中查阅资料发现可以用并查集操作实现路径压缩，具体就是在并查集的 find 函数寻找根时，实时更新每一个节点的 root，实现压缩操作。最后成功解决了这一问题。

实验一（选做）

一、实验题目：分治算法——找第 k 小元素，基于二次取中算法

二、实验目的与内容

1、实验目的：

1) 掌握分治算法的基本思想与设计方法。
2) 掌握基于分治的快速排序的实现方法。
3) 了解并探究基准点的选择对于快速排序算法的性能影响。

2、实验内容：

1) 利用分治算法和二次取中方法，设计并实现寻找第 k 小元素的算法。
   a) 输入：第一行有两个数字，分别是寻找的序号 k 和数组长度 n；然后第二行有 n 个数字，是待查找数组中的 n 个数字。
   b) 输出：第 k 小的数。
2) 设计数据测试程序正确性。
3) 分析算法复杂度。
4) 探索在二次取中时不同的分组数对于程序性能的影响。

三、算法设计

1、算法描述

本次实验主要运用快速排序的思想实现寻找第 k 小的元素，其中基准点的选取是采用了二次取中算法，降低最坏情况的算法复杂度。下述描述过程只针对分组 n=5 的情况进行说明，给出算法的思路，在结果分析的时候会具体说明分组数对于算法性能的影响。

STEP 1: 求出中位数的中位数，选取其作为快速排序的 pivot

设计函数 GetMid，将数组 a 下标从 start 到 end 的元素进行排序，取返回下标(start+end)/2。（即中位数）

设计函数 GetMidPivot，取分组后所得的中位数数组中的中位数。先将数组 a 按照 5 个一组进行划分，用 GetMid 函数取每组的中位数，存入 pivot 数组。对 pivot 数组再次调用 GetMid 函数，得到中位数的中位数。

STEP 2：设置 left=0，right=n-1。

注意：left 和 right 为当前要处理的数据段在整个数组中的左下标和右下标。

STEP 3: 数组分成大于 pivot 和小于 pivot 两部分。

以 pivot 为界，在 left 得到 right 范围内，比 pivot 小的放 pivot 左边，比 pivot

大的放 pivot 右边，记排序后 pivot 的下标为 i。

这里我们以挖坑法实现，主要步骤如下：

a) 选取 pivot，这里我们选取最后一个数，首先把 pivot 记为坑。
b) 令 i=left，j=right，即 i 和 j 最开始指向需要进行排序的序列两端。I 一直向后移动，找到一个大于 pivot 的值，填入坑中，坑位变为 a[i]。
c) J 一直向前移动，找到一个小于 pivot 的值，填入坑，坑位变为 a[j]。
d) 重复上面两个步骤到 i=j，将 pivot 填入 a[i]。

STEP 4: 若 i=k-1，返回 a[i]的值。

若 i=k-1，代表这个基准点是要求的第 k 小的数，返回 a[i]的值。

STEP 5: 若 i<k-1，在右边递归。

这种情况证明所求数在 pivot 右边，令设 left=i+1；right=right，回到 STEP 1。

STEP 6: 若 i>k-1，在左边递归。

这种情况证明所求数在 pivot 右边，令 left=left，right=i-1，回到 STEP 1。

2、算法流程图

算法描述的流程图如图 3.1：

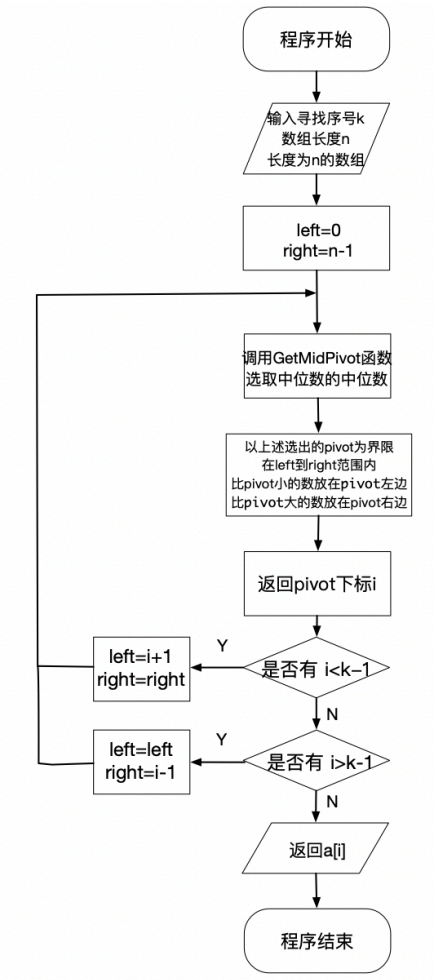

图 3.1 二次取中的选择算法找出第 k 小元素算法流程图

四、实验环境

操作系统：Windows 10
编译环境：Dev C++ 5.11
编译器　：TDM-GCC 4.9.2 64-bit Release

五、实验过程

利用二次取中的选择算法实现 K-top 问题的源代码如下，代码每部分含义、变量名已经在注释中标出：

二次取中的选择算法找出第 k 小元素

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

int GetMid(int a[],int start,int end){ //求出数组 a 在 start 到 end 位置上的中位数的下标
    sort(a+start,a+end+1); //start 到 end 排序
    int temp=start+end;
    int mid=(temp/2)+(temp%2); //取中位数的下标
    return mid;
}

int GetMidPivot(int a[],int start, int end){ //将数组 a 划分，求出中位数的中位数的下标
    int len=end-start+1;
    int group=len/5+(len%5==0?0:1); //求出组数，不足 5 个按照 5 个计算
    int pivot[group]; //存储每一组中位数的下标
    for(int k=0;k<group;k++){
        int i=start+k*5,w=i+4,j=min((i+4),end);
        pivot[k]=GetMid(a,i,j); //求出每一组的中位数下标
    }
    int index=GetMid(pivot,0,group-1); //取中位数下标数组的中位数的下标
    return pivot[index];
}

void bfqrt(int a[],int left,int right, int k){ //分治主体
    //挖坑法实现
    int i=left,j=right;
    int pivot_index=GetMidPivot(a,left,right); //求中位数的中位数的下标
    int pivot=a[pivot_index];
```

```
    swap(a[pivot_index],a[right]); //将基准值放在数组末尾
    while(i<j){
        while(i<j&&a[i]<=pivot){ //左边开始寻找比 pivot 大的数进行交换
            i++;
        }
        a[j]=a[i];
        while(i<j&&a[j]>=pivot){ //右边开始寻找比 pivot 小的数进行交换
            j--;
        }
        a[i]=a[j];
    }
    a[i]=pivot;
    if(i>k-1){
        bfqrt(a,left,i-1,k); //在左边继续搜索
    }
    else if(i<k-1){
        bfqrt(a,i+1,right,k); //在右边继续搜索
    }
    else{
        cout<<"The k-smallest number is: "<<a[i]<<endl;
        return ;
    }
}

int main(){
    int k,n;   //要寻找的序号 k,数组长度 n
    cin>>k>>n;   //读入序号 k，长度 n
    int a[n]; //要查找的数组 a
    for(int i=0;i<n;i++){ //读入数组 a[]
        cin>>a[i];
    }
    if(k>n){ //若要寻找的需要大于数组大小，报错
        cout<<"the k-smallest number does not exist!"<<endl;
        return 0;
    }
    bfqrt(a,0,n-1,k); //调用函数寻找第 k 小的数
    return 0;
}
```

六、算法测试

为了保证算法的正确性，我们设置三组测试样例，设置意义如下（我们的输入输出已经在第二节实验内容中作出了约定）：

1) 第一组：一般算例检验，7 个数，求第 4 小的数字，7 个数互相不相同。
2) 第二组：边界情况检验，7 个数，求第 4 小的数字，7 个数全部一样。
3) 第三组：边界情况检验，7 个数，求第四小的数字，其中有相同的数

测试样例 1

样例输入	理论输出	样例输出
4 7 5 2 7 3 8 1 9	5	4 7 5 2 7 3 8 1 9 The k-smallest number is: 5

由于理论输出与样例输出相符，所以测试样例 1 验证成功。

测试样例 2

样例输入	理论输出	样例输出
4 7 5 5 5 5 5 5 5	5	4 7 5 5 5 5 5 5 5 The k-smallest number is: 5

由于理论输出与样例输出相符，所以测试样例 2 验证成功。

测试样例 3

样例输入	理论输出	样例输出
4 7 3 6 3 5 2 6 8	6	4 7 3 6 3 5 2 6 8 The k-smallest number is: 5

由于理论输出与样例输出相符，所以测试样例 3 验证成功。

综上，算法通过所有样例的测试。

七、结果分析

1. 算法复杂度分析：

由于我们在算法实现的时候每组分成了五份，故算法复杂度为：

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + O(n) = O(n)$$

2. 简要探究不同分组数对复杂度的影响：

通过分析，我们发现，并不是每种分组方案都能够获得线性时间，分组 k 要满足一定的条件才行，简析如下：

因为每一组分为 k 个元素，所以大于中位数的元素至少为：

$$\left\lceil\frac{k}{2}\right\rceil\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{k}\right\rceil\right\rceil-2\right)\geq\frac{n}{4}-k$$

故小于的元素至多为 3n/4+k 个。

故得到递归式：

$$T(n)\leq T\left(\left\lceil\frac{n}{k}\right\rceil\right)+T\left(\frac{3n}{4}+k\right)+O(n)$$

假设该递推式由线性解，即有$T(n)\leq cn$，我们有：

$$\begin{aligned}T(n)&\leq c\left\lceil\frac{n}{k}\right\rceil+c\left(\frac{3n}{4}+k\right)+O(n)\\&\leq c\left(\frac{n}{k}+1\right)+c\left(\frac{3n}{4}+k\right)+c_0n\\&=cn\left(\frac{1}{k}+\frac{3}{4}\right)+c(1+k)+c_0n\end{aligned}$$

即我们要有：

$$cn\left(\frac{1}{k}+\frac{3}{4}\right)+c(1+k)+c_0n\leq cn$$

即最后我们要有：$k>4$

通过计算 $k=3$ 时，复杂度为 $O(nlogn)$

即综上 k>4 的时候才有该算法的线性性质，即复杂度为 O(n)

八、总结

这个实验让我熟悉了分治算法（快速排序）的基本思想，并且用快速排序的挖坑法实现了在 O(n)时间内将一组数据分为比 pivot 小和大两部分。最后我们进行了算法复杂度分析，并且探究了如何将取中分组的时候 k 取多少才能保证该算法最坏情况有线性时间解。

实验的两个关键就是二次取中操作以及如何在 O(n)时间内将一组数据分为比 pivot 大和小两部分。在进行二次取中操作时，我先开始的想法是直接对数据进行操作，后来发现如果移动了数据在第二次取中的时候会很麻烦，所以我实现的版本是返回二次取中的下标，这样能保证取到的下标对应的数一定是二次取中的数。在 O(n)时间内将数据分为比 pivot 大和小两部分我采用的是快速排序中的挖坑法的思想，解决了问题。