华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析　　**年级**：22 级　　**上机实践成绩**：

指导教师：金澈清　　**姓名**：郭夏辉

上机实践名称：顺序统计量　　**学号**：10211900416　　**上机实践日期**：2023 年 3 月 30 日

上机实践编号：No.5　　**组号**：1-416

一、目的

1．熟悉算法设计的基本思想
2．掌握随机选择算法（rand select）的方法
3．掌握选择算法（SELECT）的方法

二、实验内容

1. 编写随机选择算法和 SELECT 算法；
2. 随机生成 1e2、1e3、1e4、1e5、1e6 个数，使用随机选择算法和 SELECT 算法找到第 0.5N 大的数输出，并画图描述不同情况下的运行时间差异；
3. 随机生成 1e6 个数，使用随机选择算法和 SELECT 算法找到第 0.2N、0.4N、0.6N、0.8N 大的数输出，并画图描述不同情况下的运行时间差异；
4. 递增生成 1e2、1e3、1e4、1e5、1e6 个数，使用随机选择算法和 SELECT 算法找到第 0.5N 大的数输出，并画图描述不同情况下的运行时间差异；
5. 随机生成 1e2、1e3、1e4、1e5、1e6 个数，使用 merge sort 找到第 0.5N 大的数输出，并画图描述不同情况下的运行时间差异；
6. 对比随机选择算法和 SELECT 算法以及 merge sort。

三、使用环境

推荐使用 C/C++集成编译环境。

四、实验过程

1. *写出随机选择算法以及 SELECT 算法的源代码。*
2. *测量不同情况下的运行时间。*
3. *用合适的统计图来表现你的数据。*

● 随机选择算法

rand-select 算法是一种基于快速排序思想的算法，用于在一个无序的数组中选择第 k 小的元素。具体步骤是这样的:

具体步骤如下：

1.随机选择一个数组中的元素 pivot 作为基准点。

2.将数组中小于 pivot 的元素移到 pivot 左边，大于 pivot 的元素移到 pivot 右边，得到左右两个子数组。

3.1 如果 k 等于 pivot 的下标，那么 pivot 就是要选择的元素，结束算法。

3.2 如果 k 小于 pivot 的下标，那么递归地在左子数组中寻找第 k 小的元素。

3.3 如果 k 大于 pivot 的下标，那么递归地在右子数组中寻找第(k-pivot 下标-1)小的元素。

4.重复上次操作，直到 k 等于 pivot 的下标为止。

虽然这个算法是以快速排序的 PARTITION 操作作为基础的，但是也与快速排序有一些明显的不同——快速排序需要处理划分的数组的两段但是该算法只用处理一段就可以了。这种“判定型”的算法相对于“改变型”的算法有明显的运行时间提升，快速排序的平均时间复杂度是 Θ(nlgn)但 rand-select 的平均时间复杂度却是 Θ(n).

至于 rand-select 为何平均情况下时间复杂度是 Θ(n)，《算法导论》上有较为清晰严谨的证明，此处就不多赘述了。（课本 122 页）然而我们需要知道的是，随机选择算法与快速排序算法类似，面对划分不均衡的情况(比如一个已经排好序的数组且选取的 pivot 一直是靠近两端的，划分的两段是极不均衡的)，算法会退化到最坏时间复杂度 Θ(n^2)。这里用到的随机化思想，可以有效地避免最坏情况的发生。

具体的代码如下所示:

```
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#include <string.h>
#define MAXN 1000006
using namespace std;
char filename[13][128];
char tmp1[]="./data/randnum-";
char tmp2[]=".txt";
char tmp3[10]="100";
char tmp4[]="-rand",tmp5[]="-increase";
int a[MAXN];

void swap(int *m, int *n){
    int tmp;tmp= *m;*m = *n;*n = tmp;
}
int partition(int l,int r){
    int x=a[r];
    int i=l-1;
    for(int j=l;j<r;j++){
        if(a[j]<=x){
            i++;
            swap(&a[i],&a[j]);
        }
    }
    swap(&a[i+1],&a[r]);
    return i+1;
}
int random_partition(int l, int r){
    int t=l+rand() % (r-l+1);
```

```
    swap(&a[t],&a[r]);
    return partition(l,r);
}
int random_select(int l,int r,int num){
    if(l==r)return a[l];
    int pivot=random_partition(l,r);
    int k=pivot-l+1;
    if(num==k)return a[pivot];
    else if(num<k)return random_select(l,pivot-1,num);
    else return random_select(pivot+1,r,num-k);
}
int main(){
    cout<<"This is the test for rand-select algorithm:"<<endl;
    srand(time(NULL));
    clock_t start, stop;
    for(int i=1;i<=10;i++)strcpy(filename[i],tmp1);
    for(int i=1;i<=5;i++){
        strcat(filename[i],tmp3);
        strcat(filename[i],tmp4);
        strcat(filename[i],tmp2);
        strcat(filename[i+5],tmp3);
        strcat(filename[i+5],tmp5);
        strcat(filename[i+5],tmp2);
        tmp3[i+2]='0';
        tmp3[i+3]='\0';
    }
    int tmp=100;
    cout<<"Task2:random 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:"<<endl;
    for(int i=1;i<=5;i++){
        ifstream fin(filename[i]);
        for(int j=0;j<tmp;j++)fin>>a[j];
        start=clock();
        cout<<"0.5N: "<<random_select(0,tmp-1,tmp/2);
        stop=clock();
        cout<<" Scale: "<<tmp<<" Time: "<<1000*((double) (stop - start) /
CLOCKS_PER_SEC)<<"ms"<<endl;
        fin.close();
        tmp*=10;
    }

    cout<<"Task3:random 1e6, get 0.2N,0.4N,0.6N,0.8N big num:"<<endl;
    tmp=1000000;
    ifstream fin1(filename[5]);
    for(int i=0;i<tmp;i++)fin1>>a[i];
    fin1.close();
    for(int i=1;i<=4;i++){
        start=clock();
        cout<<"0."<<(i*2)<<"N:"<<random_select(0,tmp-1,(tmp/5)*i)<<" ";
```

```
        stop=clock();
        cout<<"Time: "<<1000*((double) (stop - start) / CLOCKS_PER_SEC)<<"ms"<<endl;
    }
    cout<<"Task4:increase 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:"<<endl;
    tmp=100;
    for(int i=1;i<=5;i++){
        ifstream fin(filename[i+5]);
        for(int j=0;j<tmp;j++)fin>>a[j];
        start=clock();
        cout<<"0.5N: "<<random_select(0,tmp-1,tmp/2);
        stop=clock();
        cout<<" Scale: "<<tmp<<" Time: "<<1000*((double) (stop - start) /
CLOCKS_PER_SEC)<<"ms"<<endl;
        fin.close();
        tmp*=10;
    }
    return 0;
}
```

接下来是我的 rand-select 算法的运行结果:

This is the test for rand-select algorithm:
Task2:random 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:
0.5N: 41 Scale: 100 Time: 0ms
0.5N: 521 Scale: 1000 Time: 0ms
0.5N: 5030 Scale: 10000 Time: 1ms
0.5N: 49752 Scale: 100000 Time: 3ms
0.5N: 499039 Scale: 1000000 Time: 12ms
Task3:random 1e6, get 0.2N,0.4N,0.6N,0.8N big num:
0.2N:199775 Time: 16ms
0.4N:399091 Time: 20ms
0.6N:598896 Time: 42ms
0.8N:799799 Time: 33ms
Task4:increase 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:
0.5N: 41 Scale: 100 Time: 1ms
0.5N: 521 Scale: 1000 Time: 0ms
0.5N: 5030 Scale: 10000 Time: 0ms
0.5N: 49752 Scale: 100000 Time: 1ms
0.5N: 499039 Scale: 1000000 Time: 24ms

- SELECT 算法

SELECT 算法也是一种用于在一个无序数组中选择第 k 小的元素的算法。在我来看，它是通过递归的将数组划分成若干个子数组，直到找到第 k 小的元素为止。但是这个递归和随机选择的递归还不太一样，还利用了不等号的可传递性，通过中位数代表一个五元组，然后起到了明显的划分区间作用。它的具体步骤是这样的:

1. 将数组分成⌊n/5⌋组(当然有可能其中还会剩下一组有 n%5 个元素)，每组内部排序（对于小数组而言，快速排序并不占优，我选择的是插入排序），找出每组的中位数，共有⌊n/5⌋（或⌊n/5⌋+1）个中位数。

2.递归地调用 SELECT 算法来找到⌊n/5⌋（或⌊n/5⌋+1）个中位数的中位数 m。

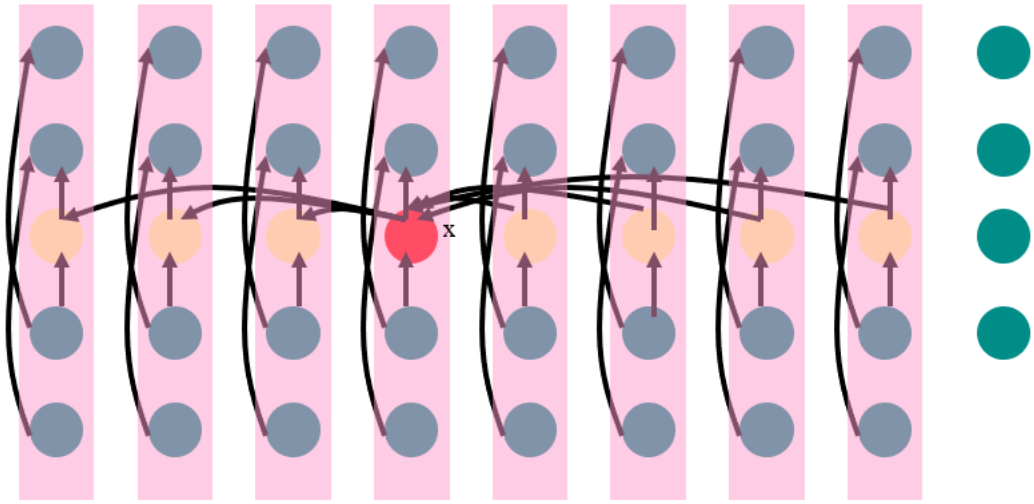
3.根据修改后的 partition 操作(这里的 partition 操作相对于快速排序中还是有些不同，并

没有随机化，而是将用于划分的 pivot 也作为了输入参数，是一个确定性的划分)按中位数的中位数 m 进行划分，可以很轻松地得到 m 是数组中第 k 小的元素，有 n-k 个元素在高区。

3.1 如果 k=num,则返回 m；

3.2 如果 k>num,则在低区递归调用相似的过程求其第 num 小的元素

3.3 如果 k<num,则在高区递归调用相似的过程求其第 num-k 小的元素



对我个人而言，这个算法的本质还是利用了不等号的可传递性，中间的细节则是仿效快速排序的 partition、配合归并排序等算法实现的。

至于 SELECT 算法的时间复杂度，我是这样来分析的。在算法的步骤 2 找到的中位数中，至少有一半是不低于中位数的中位数 m 的，在那⌈n/5⌉组中，除去最后那个在 n 不是 5 的倍数时遗留下来的组和包含 m 的那个组外，至少有一半的组中有 3 个元素大于 m。不算这两个组的化大于 m 的元素个数至少为 3(⌈⌈n/5⌉/2⌉-2)>=(3n/10-6).然后在第 3 步中，SELECT 接下来递归调用时最多作用于 7n/10+6 个元素。

$$T(n) \leqslant \begin{cases} O(1) & \text{若 } n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10+6) + O(n) & \text{若 } n \geqslant 140 \end{cases}$$

然后再利用代入法证明其时间复杂度:

$$\begin{aligned} T(n) &\leqslant c\lceil n/5 \rceil + c(7n/10+6) + an \\ &\leqslant cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

只要这个式子存在 c>0 使得 n>140 时成立就行了: $-cn/10+7c+an \leqslant 0$

因为假设 n>140,所以 n/(n-70)<=2,然后只要选择一个 c>=20a 就能满足不等式。

综上，我证明了最坏情况下 SELECT 算法的时间复杂度也是线性的，当然最好情况下的证明与之类似，也可以证明此时该算法的复杂度也是线性的。

有一个小小的问题，就是在这个证明过程中一定要保证 n>=140 吗？显然不是的，是可以选择足够大的 n 的，只不过选择之后要重新选 c 满足那个不等式的成立，每次选择都会发现相应的 c 存在，即该算法的最坏时间复杂度满足 Θ(n)。

为了便利化 select 操作的使用，我在实验过程中发现需要 select 递归返回两个值，所以我设置了一个结构体方便传值。我的代码如下所示:

```
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cstdlib>
```

```cpp
#include <ctime>
#include <algorithm>
#include <cmath>
#include <string.h>
#define MAXN 1000006
using namespace std;
struct PACK{int index,value;};
int a[MAXN];
char filename[13][128];
char tmp1[]="./data/randnum-";
char tmp2[]=".txt";
char tmp3[10]="100";
char tmp4[]="-rand",tmp5[]="-increase";
void swap(int *m, int *n);

int partition(int l,int r,int pivot){
    int tmp=a[pivot];
    int i=l-1;
    for(int j=l;j<=r;j++){
        if(a[j]<=tmp){
            i++;swap(&a[i], &a[j]);
        }
    }
    swap(&a[i],&a[pivot]);
    return i;
}
void insert_sort(int a[],int l, int r){
    int key,j;
    for(int i=l;i<=r;i++){
        key=a[i];
        j=i-1;
        while(j>=l && a[j]>key){
            a[j+1]=a[j];j--;
        }
        a[j+1]=key;
    }
}
PACK select(int l,int r,int num){
    if(r-l<=4){
        insert_sort(a,l,r);
        PACK pack1;
        pack1.index=l+num-1;
        pack1.value=a[pack1.index];
        return pack1;
    }
    if(r-l<140){
        insert_sort(a,l,r);
        PACK pack1;
```

```
        pack1.index=l+num-1;
        pack1.value=a[l+num-1];
        return pack1;
    }
    for(int i=0;i*5<=(r-l);i++){
        if(l+i*5+4>r){
            insert_sort(a,l+i*5,r);
            int mid=l+i*5+(r-(l+i*5))/2;
            swap(&a[mid],&a[l+i]);
        }else{
            insert_sort(a,l+i*5,l+i*5+4);
            int mid=l+i*5+2;
            swap(&a[mid],&a[l+i]);
        }
    }
    PACK pack2=select(l,l+(r-l)/5,((r-l)/5+1)/2);
    int pivot=partition(l,r,pack2.index);
    int k=pivot-l+1;
    if(num==k) {
        PACK pack3;
        pack3.index=pivot;pack3.value=a[pivot];
        return pack3;
    }else if(num<k)return select(l,pivot-1,num);
    else return select(pivot+1,r,num-k);
}

int main(){
    cout<<"This is the test for SELECT algorithm:"<<endl;
    srand(time(NULL));
    clock_t start, stop;
    for(int i=1;i<=10;i++)strcpy(filename[i],tmp1);
    for(int i=1;i<=5;i++){
        strcat(filename[i],tmp3);
        strcat(filename[i],tmp4);
        strcat(filename[i],tmp2);
        strcat(filename[i+5],tmp3);
        strcat(filename[i+5],tmp5);
        strcat(filename[i+5],tmp2);
        tmp3[i+2]='0';
        tmp3[i+3]='\0';
    }
    int tmp=100;
    cout<<"Task2:random 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:"<<endl;
    for(int i=1;i<=5;i++){
        ifstream fin(filename[i]);
        for(int j=0;j<tmp;j++)fin>>a[j];
        start=clock();
        cout<<"0.5N: "<<select(0,tmp-1,tmp/2).value;
```

```
        stop=clock();
        cout<<" Scale: "<<tmp<<" Time: "<<1000*((double) (stop - start) /
CLOCKS_PER_SEC)<<"ms"<<endl;
        fin.close();
        tmp*=10;
    }

    cout<<"Task3:random 1e6, get 0.2N,0.4N,0.6N,0.8N big num:"<<endl;
    tmp=1000000;
    ifstream fin1(filename[5]);
    for(int i=0;i<tmp;i++)fin1>>a[i];
    fin1.close();
    for(int i=1;i<=4;i++){
        start=clock();
        cout<<"0."<<(i*2)<<"N:"<<select(0,tmp-1,(tmp/5)*i).value<<" ";
        stop=clock();
        cout<<"Time: "<<1000*((double) (stop-start)/ CLOCKS_PER_SEC)<<"ms"<<endl;
    }


    cout<<"Task4:increase 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:"<<endl;
    tmp=100;
    for(int i=1;i<=5;i++){
        ifstream fin(filename[i+5]);
        for(int j=0;j<tmp;j++)fin>>a[j];
        start=clock();
        cout<<"0.5N: "<<select(0,tmp-1,tmp/2).value;
        stop=clock();
        cout<<" Scale: "<<tmp<<" Time: "<<1000*((double) (stop-start)/
CLOCKS_PER_SEC)<<"ms"<<endl;
        fin.close();
        tmp*=10;
    }


    return 0;
}
void swap(int *m, int *n){
    int tmp;tmp= *m;*m = *n;*n = tmp;
}
```

接下来是我的 SELECT 算法的运行结果:

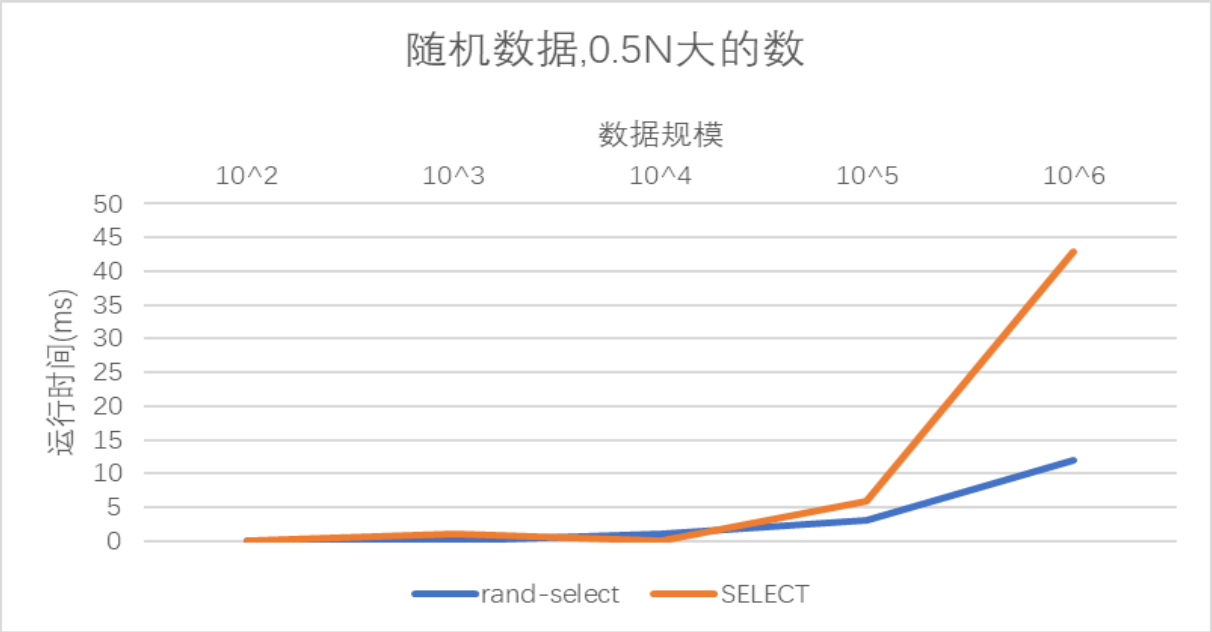
This is the test for SELECT algorithm:
Task2:random 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:
0.5N: 41 Scale: 100 Time: 0ms
0.5N: 521 Scale: 1000 Time: 1ms
0.5N: 5030 Scale: 10000 Time: 0ms
0.5N: 49752 Scale: 100000 Time: 6ms
0.5N: 499039 Scale: 1000000 Time: 43ms
Task3:random 1e6, get 0.2N,0.4N,0.6N,0.8N big num:

0.2N:199775 Time: 43ms
0.4N:399091 Time: 37ms
0.6N:598896 Time: 36ms
0.8N:799799 Time: 34ms
Task4:increase 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:
0.5N: 41 Scale: 100 Time: 0ms
0.5N: 521 Scale: 1000 Time: 0ms
0.5N: 5030 Scale: 10000 Time: 1ms
0.5N: 49752 Scale: 100000 Time: 4ms
0.5N: 499039 Scale: 1000000 Time: 24ms

- rand-select 与 SELECT 算法的各种情况下的对比

1. 随机生成 1e2、1e3、1e4、1e5、1e6 个数，使用随机选择算法和 SELECT 算法找到第 0.5N 大的数输出，并画图描述不同情况下的运行时间差异

	10^2	10^3	10^4	10^5	10^6
rand-select	0	0	1	3	12
SELECT	0	1	0	6	43

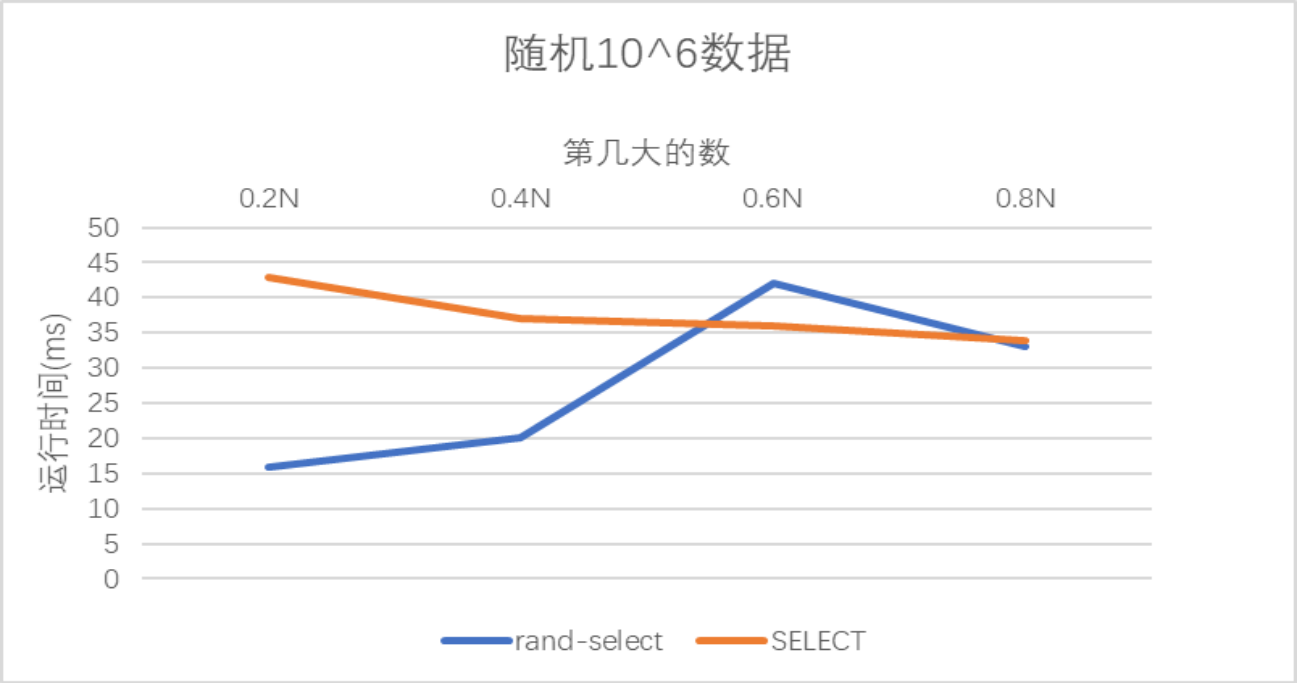


这个图还是挺出乎意料的，最坏情况下为线性时间复杂度的 SELECT 算法在实际情况中并不如平均情况下为线性时间复杂度的 rand-select 算法。具体的原因在后面总结部分。

2. 随机生成 1e6 个数，使用随机选择算法和 SELECT 算法找到第 0.2N、0.4N、0.6N、0.8N 大的数输出，并画图描述不同情况下的运行时间差异；

	0.2N	0.4N	0.6N	0.8N
rand-select	16	20	42	33

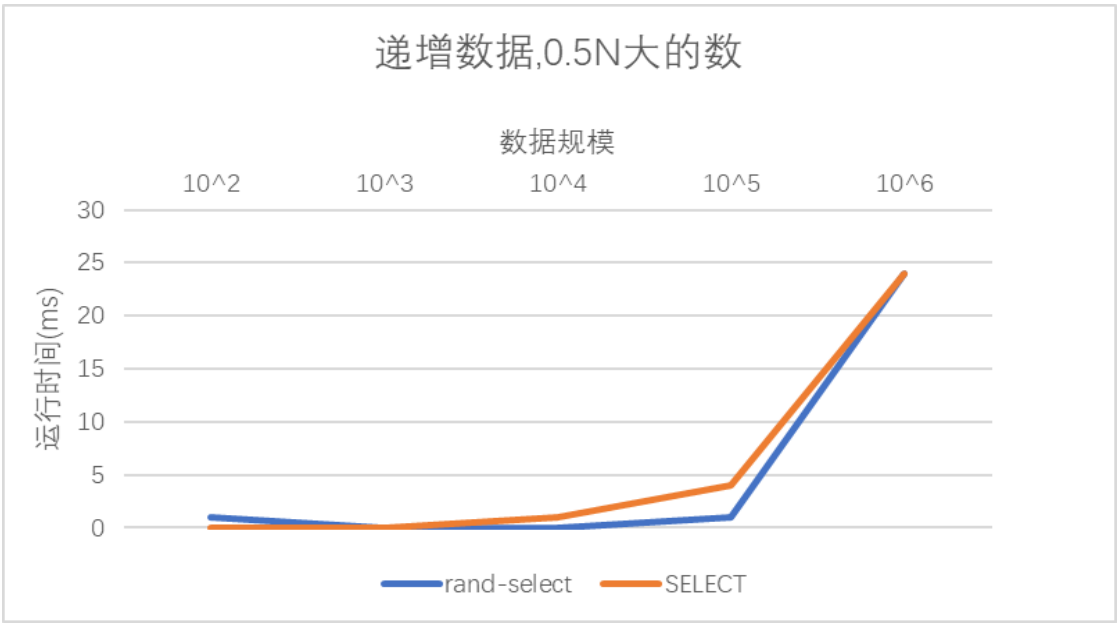
SELECT	43	37	36	34



在相同的数据规模下，可以看到两种算法的运行时间（尤其是 SELECT 算法）应该不受第几大的数的影响（其中 rand-select 可能是因为随机数的选择问题，出现了较大程度的抖动）。为何会出现这样的情况呢？因为两者都是在找一个 pivot 值后比较、拓展，决定时间的本质在于递归树的深度。由于数据是随机生成的，选择第 num 大并不会决定性地影响递归树的深度，应该有 Ω(n)的时间消耗。

3.递增生成 1e2、1e3、1e4、1e5、1e6 个数，使用随机选择算法和 SELECT 算法找到第 0.5N 大的数输出，并画图描述不同情况下的运行时间差异；

	10^2	10^3	10^4	10^5	10^6
rand-select	1	0	0	1	24
SELECT	0	0	1	4	24



通过与之前的那个随即情况下求各数据规模中位数的运行时间对比，可以发现递增情况下求中位数的时间明显缩小了。由于运行时间的增长之决定性因素在于递归树的深度，这又收到了数据分布是否均匀、pivot 值选取等因素的影响。在递增情况下，以 SELECT 算法为例，partition 操作可以更加均匀地划分数组，降低递归树的深度，进而优化了运行时间。

3.随机生成 1e2、1e3、1e4、1e5、1e6 个数，使用 merge sort 找到第 0.5N 大的数输出，并画图描述不同情况下的运行时间差异

补充一下利用归并排序得到的数据:（具体的代码见附件）

This is the test for Merge-sort algorithm:

Task:random 1e2,1e3,1e4,1e5,1e6, get 0.5N big num:

0.5N: 41 Scale: 100 Time: 0ms

0.5N: 522 Scale: 1000 Time: 0ms

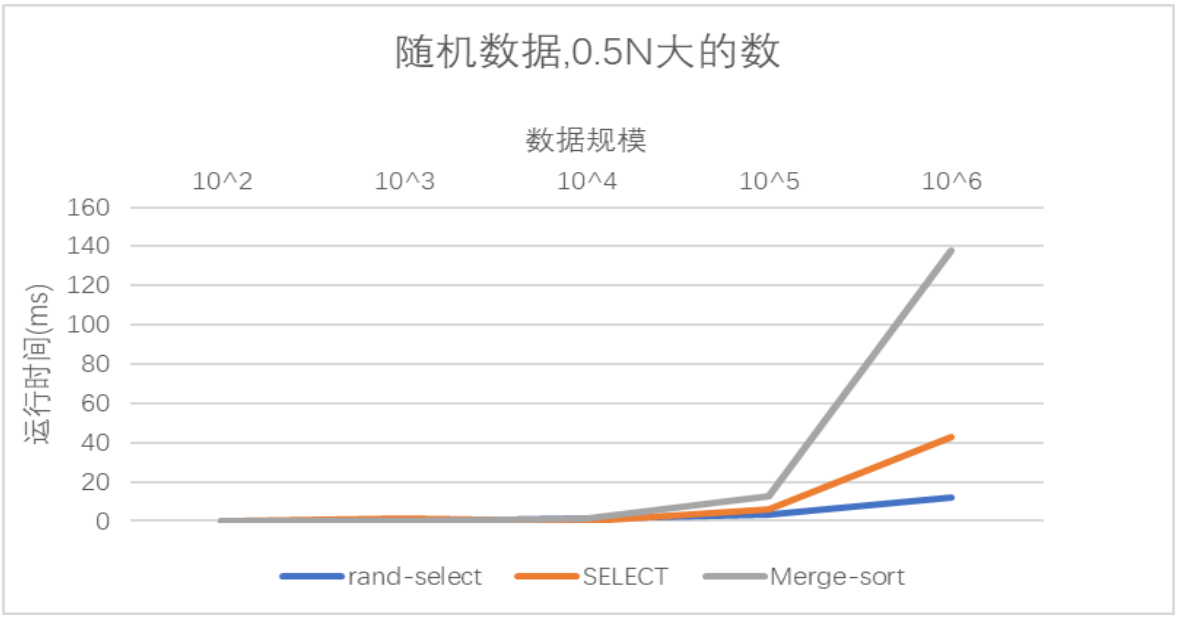
0.5N: 5032 Scale: 10000 Time: 1ms

0.5N: 49752 Scale: 100000 Time: 13ms

0.5N: 499039 Scale: 1000000 Time: 138ms

在一张表中对三种算法的运行时间进行对比

	10^2	10^3	10^4	10^5	10^6
rand-select	0	0	1	3	12
SELECT	0	1	0	6	43
Merge-sort	0	0	1	13	138



可以很清楚地看到，归并排序消耗的时间远远大于 rand-select，SELECT。这主要是因为归并排序对所有的数据都进行了排序，对很多数据的排序是不那么必要的，时间复杂度是 Θ(nlgn),但是 rand-select 和 SELECT 算法更多的是一种“选择一边就认准一边了，不考虑另外一边的情况了的算法”，较大程度地缩小了时间复杂度增长的规模。

五、总结

对上机实践结果进行分析，问题回答，上机的心得体会及改进意见。

- 对比 rand-select、SELECT 和 Merge-sort 算法

对于 rand-select，SELECT、Merge-sort 算法的运行原理，前文和之前的实验报告已经提及，这里就不再重复说明了，这里重点讨论细节上的差异。

理论上来说，rand-select 的平均时间复杂度是 Θ(n)，最坏时间复杂度是 Θ(n^2);而 SELECT 算法的最坏时间复杂度是 Θ(n)，优于 rand-select 算法。但是在实际运行过程中 SELECT 算法的表现并不尽如人意，我认为原因有以下几点:

1. 随机选择算法直接通过随机数确定了 pivot，而选择算法需要通过插入排序并递归取中位数的方式找出区间最优的 pivot，对数据的预处理耗时较大;
2. SELECT 算法在每次递归中都需要对数组进行分割，这个过程也需要进行计算，会增加常数因子;
3. 大部分情况下，rand-select 算法选择的 pivot 值是合适的，分割较为均衡的同时增快了算法的运行效率。

然而 Merge-sort 算法在实际运行的过程中存在大量的无用排序，我们只需要获得某个数组中的第 num 大的数即可，并不需要关注剩下的数的实际大小情况了。而且比较排序的下界 Θ(nlgn)限制住了 Merge-sort 算法的进一步优化。

在 C++的 STL 之 algorithm 库中的 nth_element 函数中实现了今日实验的操作，让我来看一下工程角度来讲和实验的区别与联系，以此来进一步优化思路。

```
template<class _RanIt>
void nth_element(_RanIt _First, _RanIt _Nth, _RanIt _Last);
```

其功能是对区间 [_First, _Last) 的元素进行重排，其中位于位置 _Nth 的元素与整个区间排序后位于位置 _Nth 的元素相同，并且满足在位置 _Nth 之前的所有元素都“不大于”它和位置 _Nth 之后的所有元素都“不小于”它，而且并不保证 _Nth 的前后两个区间的所有元

素保持有序。由于算法主要分两部分实现，第一部分是进行二分法弱分区，第二部分是对包含 _Nth 的位置的区间进行插入排序（STL 的阈值为 32）。当元素较多时平均时间复杂度为 O(N)，元素较少时最坏情况下时间复杂度为 O(N^2)。

研究相关的源代码（此处就不放了，主要是_Nth_element , _Unguarded_partition , _Median 和 _Med3 函数的实现）,我发现这个算法的核心还在于选择 pivot。nth_element 的选择 pivot 的策略是：把区间长度八等份记为 step，左端点开始按照 step 取三个点，记为 123，右端点倒序取 789，平均数中点一左一右取 456. 分别排序 123,456,789，也就是 2,5,8 是他们的中位数，然后再排序 258，5 这个位置也就是中位数的中位数，作为选择的 pivot。虽然有点类似于 BFPRT（SELECT 算法的另外称呼），不过 BFPRT 的思路是区间每 5 个数插入排序一下找到中位数，然后再找到中位数的中位数。nth_element 的三点排序只用了三次比较然后 swap，所以已经非常优化了，虽然这样获取的中位数肯定不如 BFPRT 那么近似，但在性能上十分优秀，减少了冗长的预处理得到 pivot 的时间。并且不依赖随机数，不像 rand-select 那样可能会出现意外的最坏情况。

总而言之，这次实验的过程还是比较有趣的，重点还在于如何选择 pivot，避免划分出来的子问题规模依旧很大之糟糕情况。