



同濟大學
TONGJI UNIVERSITY

项目说明文档

8种排序算法的比较案例

指导教师：张颖

1751984 王舸飞

1.分析

1.1 背景分析

1.2 功能分析

2.设计

2.1 总体框架设计

2.2 排序算法初始化设计

2.3 主函数初始化设计

2.4 主函数设计

3.实现

3.1 冒泡排序的实现

3.1.1 冒泡排序流程图

3.1.2 冒泡排序核心代码

3.1.3 冒泡排序性能分析

3.2 选择排序的实现

3.2.1 选择排序流程图

3.2.2 选择排序核心代码

3.2.3 选择排序性能分析

3.3 插入排序的实现

3.3.1 插入排序流程图

3.3.2 插入排序核心代码

3.3.3 插入排序性能分析

3.4 希尔排序的实现

3.4.1 希尔排序流程图

3.4.2 希尔排序核心代码

3.4.3 希尔排序性能分析

3.5 快速排序的实现

3.5.1 快速排序流程图

3.5.2 快速排序核心代码

3.5.3 快速排序性能分析

3.6 堆排序的实现

3.6.1 堆排序流程图

3.6.2 堆排序核心代码

3.6.3 堆排序性能分析

3.7 归并排序的实现

3.7.1 归并排序流程图

3.7.2 归并排序核心代码

3.7.3 归并排序性能分析

3.8 基数排序的实现

3.8.1 基数排序流程图(MSD)

3.8.2 基数排序核心代码

3.8.3 基数排序性能分析

4.比较测试

4.1 时间测试

4.1.1 小规模随机数测试

4.1.2 中等规模随机数测试

4.1.2 大规模随机数测试

4.2 内存消耗测试

4.2.1 大规模测试

4.3 特殊测试

4.3.1 产生固定一位数的随机数

4.3.1 产生固定三位数的随机数

5.总结

对上述结果进行总结如下：

1.分析

1.1 背景分析

排序问题在计算机数据处理中经常遇到，特别是在事务处理中，排序占了很大的比重。在日常的数据处理中，一般认为有四分之一的的时间用在排序上，而对于安装程序，多达一半的时间花费在对表的排序上。

因此，对于计算机排序的研究从来都不是少数，因此也产生了大量的风格不同、效率不同、适用于不同场合的排序方法。按照是否在内存排序，排序可以分成内排序和外排序两种，本次项目主要针对集中不同的内排序算法，通过实际测试比较出他们的特点和性能的优劣。

1.2 功能分析




要想观察出不同排序算法的性能差异，最直接的方法就是统计它们的运行时间，此外，还可以统计它们在一次算法内的交换次数。本次设计的排序算法有八种，对于待排序的数列，可以由随机数生成，通过实际对随机列的排序，得出八种算法的性能。

2.设计

2.1 总体框架设计

由于本次设计包含多种排序算法，每种排序算法也又可能涉及很多辅助函数，故要想将结构组织得清晰，就必须使用抽象分块的方法。

将主函数单独抽象出来，在剩余两个头文件Sort.h和Sort_support.h内分别定义排序算法及排序辅助函数

| | |
|--|---|
|  main.cpp | M |
|  sort.h | A |
|  Sort_Support.h | A |

2.2 排序算法初始化设计

Sort.h内严格规定只有八个排序算法的函数

在开始每个排序算法之前，需要对传入数据进行初始化如下：

```
clock_t start=clock();
long count=0;
srand(time(0));

for(int i=0;i<length;i++)List[i]=rand(); //赋值
```

- 1、记录排序的开始时间
- 2、创建长整形变量用于记录交换次数
- 3、设置随机数种子
- 4、对传入的数列进行随机数赋值

由于每次排序传入的都是数组指针，故在排序后序列会变得有序，这使得想要查看其他算法性能变得不可能。在每个函数前都插入上述代码，可以在每次执行时都创建一组新的数列排序，虽然不能进行直接横向比较，但在大量数据时取平均值，排序的比较还是有意义的。

2.3 主函数初始化设计

为简化主函数，将主函数涉及的初始化函数定义到Sort_Support.h内

Sort_support内涉及的主函数初始化有两个函数如下：

```
/*初始化函数*/

void Init(){
    cout<<"**          排序算法比较          **"<<endl;
    cout<<"-----"<<endl;
    cout<<"**          1 --- 冒泡排序          **"<<endl;
    cout<<"**          2 --- 选择排序          **"<<endl;
    cout<<"**          3 --- 直接插入排序        **"<<endl;
    cout<<"**          4 --- 希尔排序          **"<<endl;
    cout<<"**          5 --- 快速排序          **"<<endl;
    cout<<"**          6 --- 堆排序            **"<<endl;
    cout<<"**          7 --- 归并排序          **"<<endl;
    cout<<"**          8 --- 基数排序          **"<<endl;
    cout<<"**          9 --- 退出程序          **"<<endl;
    cout<<"-----"<<endl;
    cout<<endl;
    cout<<"请输入要产生随机数的个数： ";

}
```

```
/*输入长度的函数*/
```

```
int InputLength(){int length;  
    cin>>length;  
    while(length<=0){cout<<endl<<"输入的数据有误, 请重新输入: ";cin>>length;}  
    return length;  
}
```

分别用于给出初始提示信息 and 输入长度

此外, Sort_support内还定义了交换函数swap, 用于传入变量引用进行交换, 提高代码可读性。

```
/*交换的函数*/
```

```
void swap(int &a,int &b){int temp;  
    temp=a;a=b;b=temp;  
}
```

2.4 主函数设计

主函数代码如下:

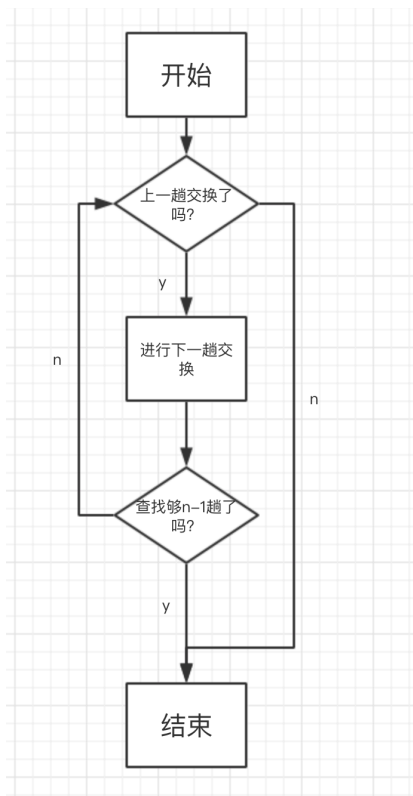
```
int main() {Init();  
  
    int length=InputLength();  
  
    int *List,Op=0;  
    List=(int*)malloc(length*sizeof(int));  
  
    cout<<endl<<"请选择排序算法: ";cin>>Op;  
    while(Op!=9){  
  
        switch(Op){  
            case 1:BubbleSort(List,length);break;  
            case 2:SelectSort(List,length);break;  
            case 3:InsertSort(List,length);break;  
            case 4:ShellSort(List,length);break;  
            case 5:QuickSort(List,length);break;  
            case 6:HeapSort(List,length);break;  
            case 7:MergeSort(List,length);break;  
            case 8:RadixSort(List,length);break;  
            default:cout<<"所选操作不存在"<<endl;  
        }  
  
        cout<<endl<<"请选择排序算法: ";cin>>Op;  
    }  
  
    free(List);  
  
    return 0;  
}
```

进入主函数后调用上文提到的初始化函数和长度输入函数，之后根据长度分配一个长度可变的数组，然后对用户所输入的操作码进行选择分支，当输入的操作码不为9时分别调用八个排序函数进行排序，否则直接退出程序。

3.实现

3.1 冒泡排序的实现

3.1.1 冒泡排序流程图



3.1.2 冒泡排序核心代码

```
for(int i=0;i<length-1;i++){
    bool exchange=false;
    for(int j=i;j<length-1;j++){
        if(List[j]>List[j+1]){tmp=List[j];
            List[j]=List[j+1];
            List[j+1]=tmp;
            count+=3;
            exchange=true;
        }
    }
    if(exchange==false)break;
}
cout<<"冒泡排序所用时间: "<<clock()-start<<endl;
cout<<"冒泡排序交换次数: "<<count<<endl;
```

3.1.3 冒泡排序性能分析

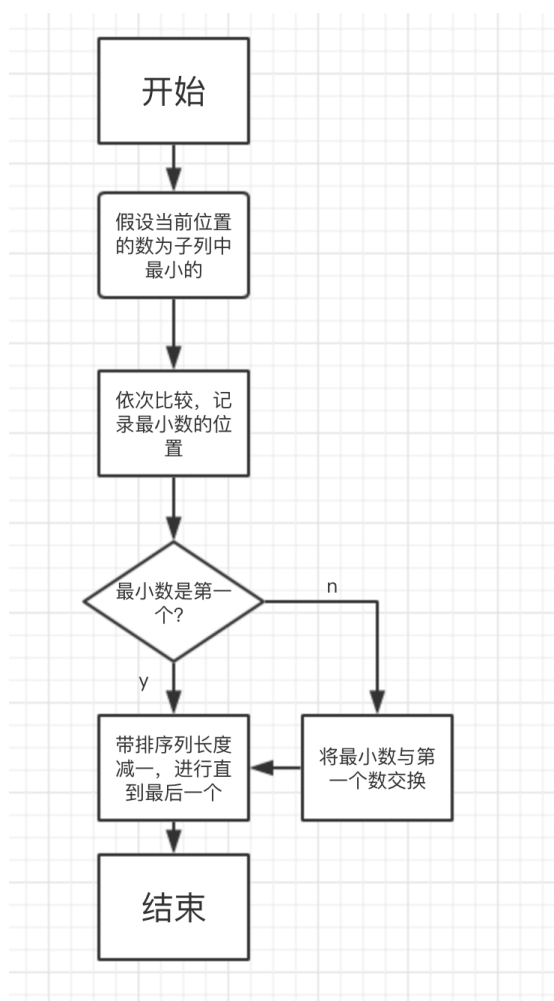
冒泡排序共要进行 $n-1$ 趟比较和交换操作，对于比较次数，它于序列本身的初始状态无关，而交换次数则与序列的初始状态密切相关：最好的情况下可以一次都不用交换，而最差的情况每一次都要交换。

将冒泡排序改进后，如果元素本身有序，那么只需要一趟排序就结束了。即使是这样，计算得到冒泡排序的时间复杂度仍然是平方级别。

由于不存在远距离交换，冒泡排序是一种稳定的排序算法。

3.2 选择排序的实现

3.2.1 选择排序流程图



3.2.2 选择排序核心代码


```

for(int j=0;j<length-1;j++){
    min=j;
    for(int k=j;k<length;k++){
        if(List[k]<List[min])min=k;
    }
    if(j!=min){tmp=List[j];
        List[j]=List[min];
        List[min]=tmp;
        count+=3;
    }
}
cout<<"选择排序所用时间: "<<clock()-start<<endl;
cout<<"选择排序交换次数: "<<count<<endl;
}

```

3.2.3 选择排序性能分析

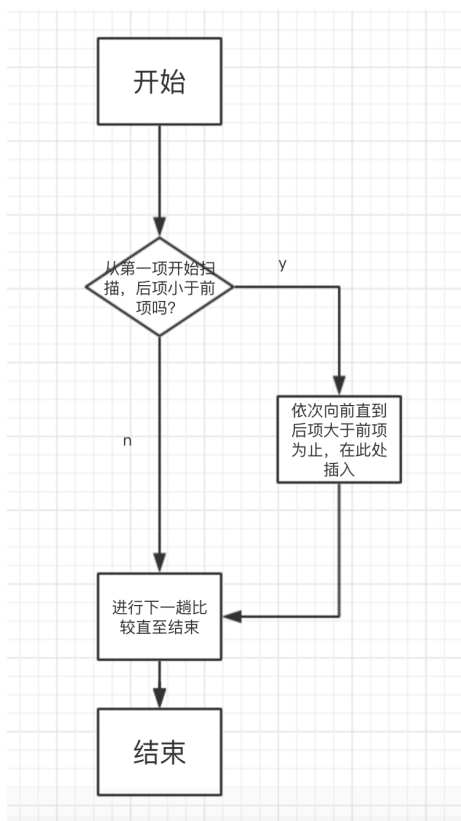
选择排序的比较次数与带排序列的初始状态无关，即每次都要比较相同的次数。对于交换次数，最好的情况下可以达到0，最坏的情况则为 $3(n-1)$ 。对于选择排序，序列的初始状态对排序的影响并不大，因此它的执行时间较为固定，它的时间复杂度为平方级别。

选择排序在序列数字较小但元素规模很大的时候有较好的效果，因为在这种情况下交换的时间会远远大于比较的时间，而选择排序涉及的交换次数很少。

由于存在远距离交换，选择排序是一种不稳定的排序算法。

3.3 插入排序的实现

3.3.1 插入排序流程图



3.3.2 插入排序核心代码

```
for(int i=1;i<length;i++){  
    if(List[i]<List[i-1]){  
        tmp=List[i];count++;int j=1;  
        while(tmp<List[i-j]){List[i-j+1]=List[i-j];  
            count++;  
            j++;  
        }  
        List[i-j+1]=tmp;  
        count++;  
    }  
}  
cout<<"直接插入排序所用时间:"<<clock()-start<<endl;  
cout<<"直接插入排序交换次数:"<<count<<endl;  
}
```

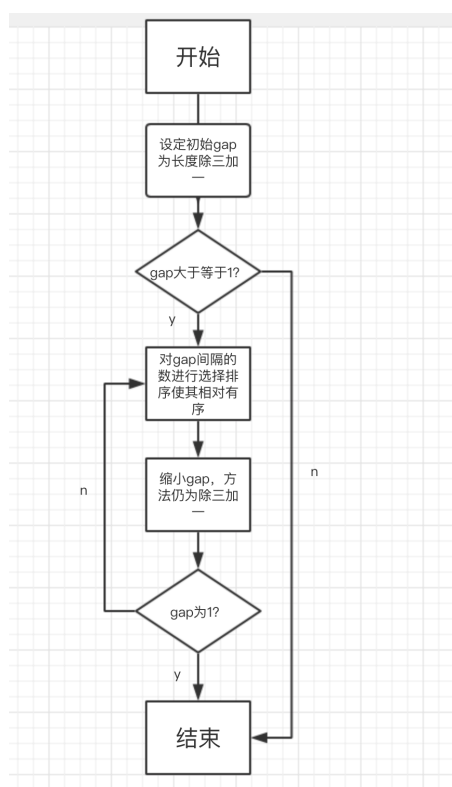
3.3.3 插入排序性能分析

插入排序的比较次数和交换次数都是和序列的初始状态相关的，在最好的情况下，只需要比较 $n-1$ 次并且不需要交换就可以完成对序列的排序，但在最坏的情况下，比较次数和交换次数均为平方级，故它的效率和序列的初始状态相关度较大，它的总时间复杂度也为平方级。

由于不存在远距离交换，插入排序是一种稳定的排序算法。

3.4 希尔排序的实现

3.4.1 希尔排序流程图



3.4.2 希尔排序核心代码

```
while(gap>=1){
    for(int i=gap;i<length;i++){
        if(List[i]<List[i-gap]){
            tmp=List[i];count++;int j=gap;
            while(tmp<List[i-j]){
                List[i-j+gap]=List[i-j];
                count++;
                j+=gap;
            }
            List[i-j+gap]=tmp;
            count++;
        }
    }

    if(gap==1)break;
    else gap=gap/3+1;
}

cout<<"希尔排序所用时间:"<<clock()-start<<endl;
cout<<"希尔排序交换次数:"<<count<<endl;
}
```

3.4.3 希尔排序性能分析

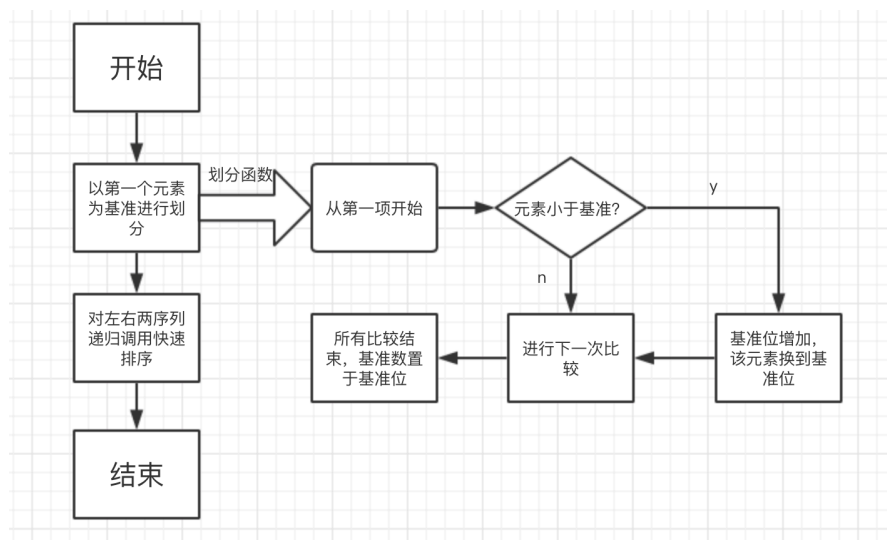
受选择排序性能虽序列有序度变化而变化的启发，当序列较有序时排序的速度会大大上升，故提出希尔排序方法。希尔排序是一种分析起来较为困难的方法，gap的取法也没有一个统一的定论，本次的gap取为了上一次的gap除三加一，这源于Knuth的想法。

根据Knuth所著的《计算机程序设计技巧》，大量统计显示希尔排序的时间复杂度在 n 的1.25次方到 $1.6n$ 的1.25次方之间。

由于存在远距离交换，希尔排序是一种不稳定的排序算法。

3.5 快速排序的实现

3.5.1 快速排序流程图



3.5.2 快速排序核心代码

```
/*快排用到的划分函数*/

int Partition(int *List,int low,int high,long &count){
    int pivotpos=low;
    int pivot=List[low];
    for(int i=low+1;i<=high;i++){
        if(List[i]<pivot){
            pivotpos++;
            if(pivotpos!=i)swap(List[pivotpos],List[i]);count+=3;
        }

    }
    List[low]=List[pivotpos];List[pivotpos]=pivot;count+=2;
    return pivotpos;
}

/*快排函数*/

void Quick(int *List,int left,int right,long &count){
    if(left<right){
        int pivotpos=Partition(List,left,right,count);
        Quick(List,left,pivotpos-1,count);
        Quick(List,pivotpos+1,right,count);
    }
}
```

3.5.3 快速排序性能分析

快速排序是目前应用最广的排序算法，好的快速排序算法在大多数的计算机上运行得都比其他排序算法快，且快速排序在空间上只使用一个辅助栈。

快速排序的速度取决于递归数的深度，如果左右两侧序列数字个数相近，快速排序可以达到它的最好性能。但如果是最坏情况，每次划分一侧只有一个自序列，那么快速排序的效果将会退化到普通排序的水平甚至比普通排序还慢。

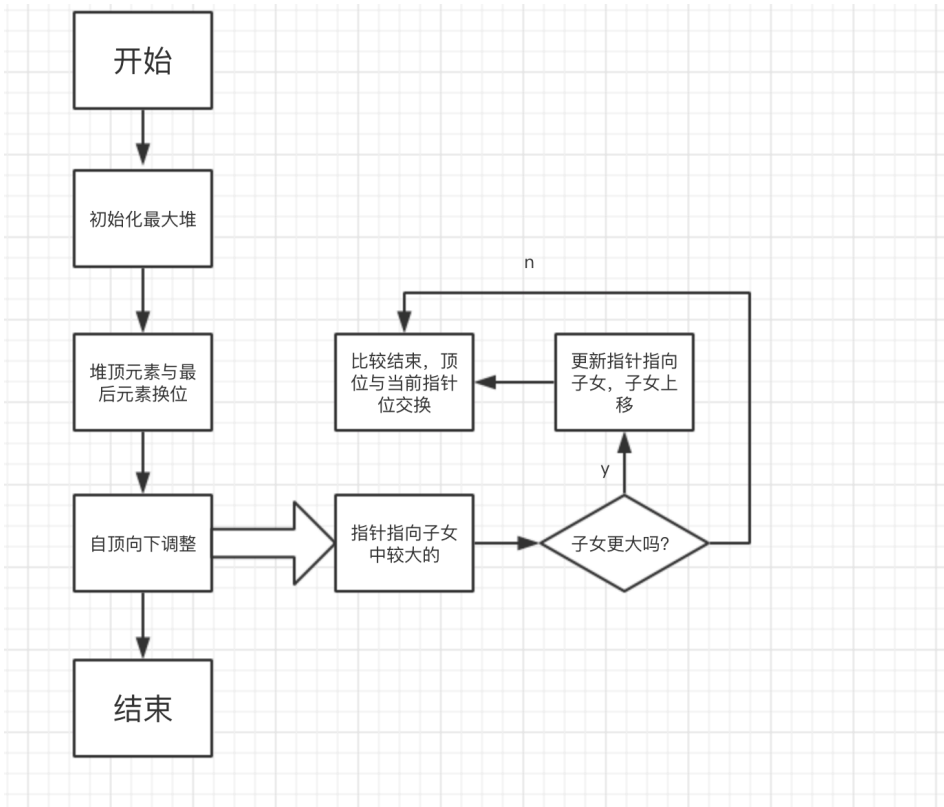
对于上述情况的改进措施是当序列的长度较小时直接采用稳定的插入排序。

但在随机产生的一般情况下，快速排序的时间复杂度位 n 乘对数级。

由于在划分时存在远距离交换，快速排序是一种不稳定的算法。

3.6 堆排序的实现

3.6.1 堆排序流程图



3.6.2 堆排序核心代码

/* 自顶向下调整的算法 */

```
void SiftDown(int *List, int start, int end, long &count){
    int i=start; int j=2*i+1; //j是i的左子女
    int tmp=List[i]; count++;
    while(j<=end){
        if(j<end&&List[j]<List[j+1])j++; //让j指向两子女中的较大者

        if(tmp>=List[j])break;
        else{
            List[i]=List[j]; count++;
            i=j; j=2*j+1;
        }
    }
    List[i]=tmp; count++;
}
```

```

for(int i=(length-2)/2;i>=0;i--)SiftDown(List,i,length-1,count);//循环调整
for(int i=length-1;i>=0;i--){swap(List[0],List[i]);count+=3;SiftDown(List,0,i-1,count);}//单独调整

cout<<"堆排序所用时间:"<<clock()-start<<endl;
cout<<"堆排序交换次数:"<<count<<endl;

```

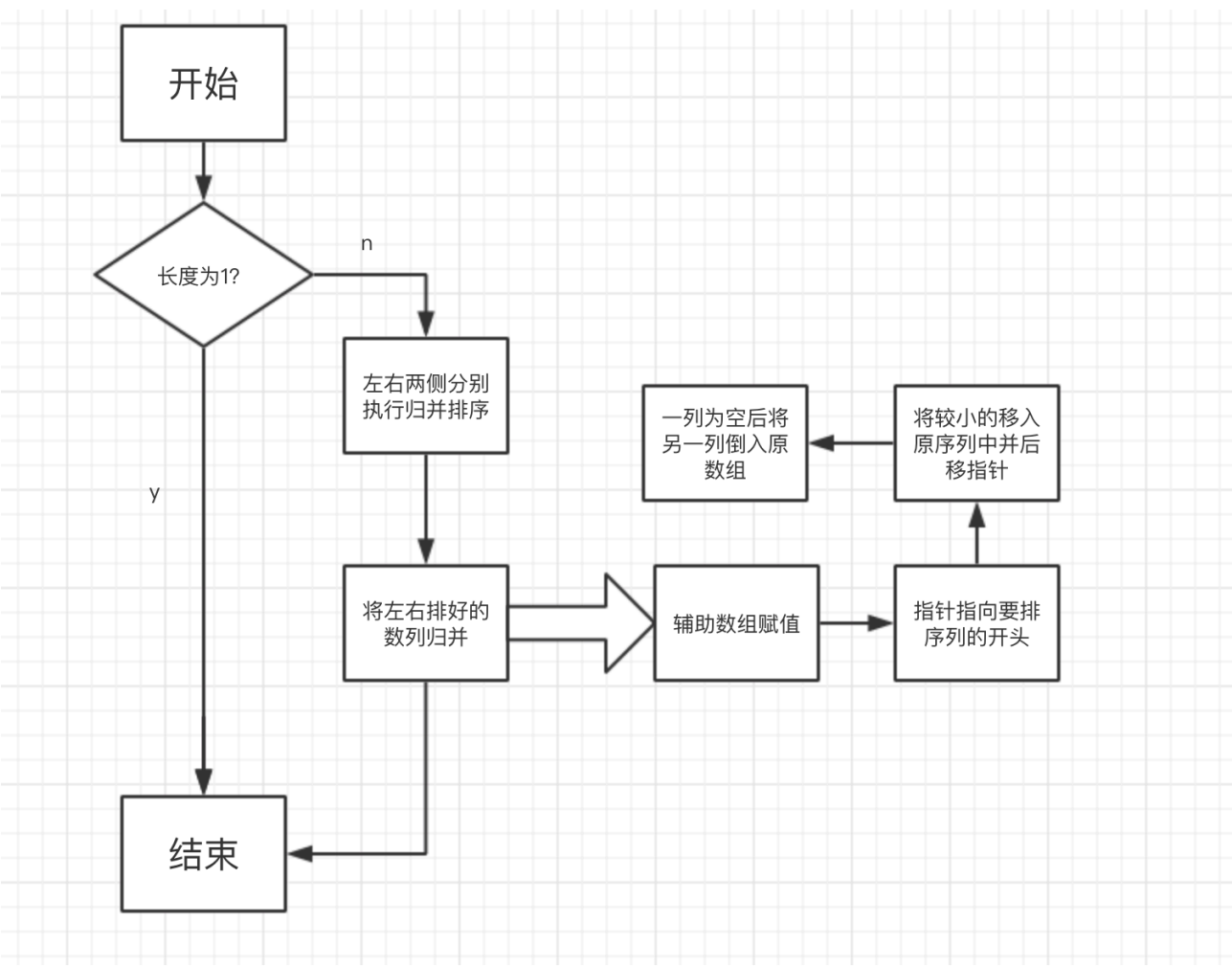
3.6.3 堆排序性能分析

堆排序需要建立最大堆，相应的需要最大堆的调整算法，它的时间复杂度为 $n \log n$ 的对数级，性能较好，如果采用数组下标作为堆号码，它并不需要额外的存储空间。

由于存在远距离交换，堆排序是一种不稳定的排序算法。

3.7 归并排序的实现

3.7.1 归并排序流程图



3.7.2 归并排序核心代码

```
/* 合并函数 */

int support_list[500000]; //辅助调整的全局数组

void merge(int *List,int *support_list,int left,int mid,int right,long &count){
    for(int i=left;i<=right;i++){support_list[i]=List[i];count++;} //辅助数组赋初值

    int s1=left,s2=mid+1,tmp_pos=left;

    while(s1<=mid&& s2<=right)
    {if(support_list[s1]<=support_list[s2])List[tmp_pos++]=support_list[s1++];
      else List[tmp_pos++]=support_list[s2++];
      count++;
    }

    while(s1<=mid){List[tmp_pos++]=support_list[s1++];count++;}
    while(s2<=right){List[tmp_pos++]=support_list[s2++];count++;}
}

/* 归并排序函数 */

void MerSort(int *List,int *support_list,int left,int right,long &count){
    if(left>=right)return;
    int mid=(left+right)/2;

    MerSort(List,support_list,left,mid,count);
    MerSort(List,support_list,mid+1,right,count);
    merge(List,support_list,left,mid,right,count);
}
```

3.7.3 归并排序性能分析

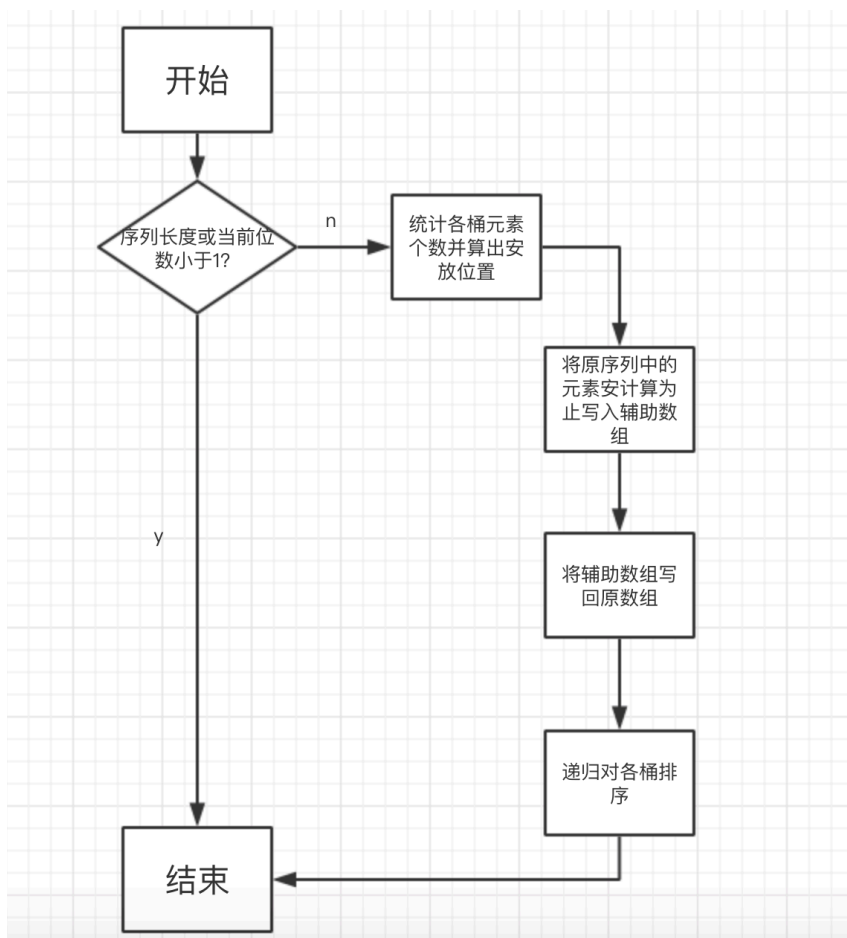
归并排序也是一种采用递归算法的排序方式，同上面分析的快速排序算法一样，它的复杂度也为 $n \log n$ 的对数级并且它不依赖初始的输入序列，始终将序列分成相等长度的两部分。

它的缺点是在归并的过程中需要开辟额外的辅助存储空间。

也正是由于辅助存储空间的指针移动是连续的，归并排序是一种稳定的排序算法。

3.8 基数排序的实现

3.8.1 基数排序流程图(MSD)



3.8.2 基数排序核心代码

/* 基数排序的辅助函数 */

```
int getdigit(int n,int d){
    for(int i=1;i<d;i++){n=n/10;}
    return n%10;
}
```

值得注意的是书上433页的文字与434页的代码并不对应不上，其次434页代码的最后部分也存在问题，下面的代码将错误用注释的形式标记了出来。


```

void RadSort(int *List,int left,int right,int d,long &tcount){
    if(d<=0||left>=right)return;

    int i,j,count[10],start,end;
    int sup[right-left+1];

    for(j=0;j<10;j++)count[j]=0;

    for(i=left;i<=right;i++)count[getdigit(List[i],d)]++;

    for(j=1;j<10;j++){count[j]=count[j]+count[j-1];}

    for(i=left;i<=right;i++){
        j=getdigit(List[i],d);
        sup[count[j]-1]=List[i];

        count[j]--;
        tcount++;
    }

    for(i=left,j=0;i<=right;i++,j++){List[i]=sup[j];tcount++;}

    for(j=0;j<9;j++){
        start=count[j]+left;//这里应该对加左端位置
        end=count[j+1]-1+left;
        RadSort(List,start,end,d-1,tcount);
    }
    RadSort(List,count[9]+left,right,d-1,tcount);//当排9号桶时终点应该取右端位置
}

```

3.8.3 基数排序性能分析

基数排序类似桶排序，是不需要交换的算法，对应的它需要较多的存储空间。基数排序的效率计算也很特别——它与带排序元素的位数有关，对于元素个数较多但排序码位数较少的情况，基数排序的效率非常好。

在元素个数多而位数少的情况常常会出现相同元素，而基数排序又恰好是一种稳定的排序函数，故在此情况下基数排序是最好的选择。

4.比较测试

4.1 时间测试

4.1.1 小规模随机数测试

| ** | 排序算法比较 | ** |
|----|--------------|----|
| ** | 1 --- 冒泡排序 | ** |
| ** | 2 --- 选择排序 | ** |
| ** | 3 --- 直接插入排序 | ** |
| ** | 4 --- 希尔排序 | ** |
| ** | 5 --- 快速排序 | ** |
| ** | 6 --- 堆排序 | ** |
| ** | 7 --- 归并排序 | ** |
| ** | 8 --- 基数排序 | ** |
| ** | 9 --- 退出程序 | ** |

请输入要产生随机数的个数：100

请选择排序算法：1
冒泡排序所用时间：66
冒泡排序交换次数：5454

请选择排序算法：2
选择排序所用时间：32
选择排序交换次数：285

请选择排序算法：3
直接插入排序所用时间：23
直接插入排序交换次数：2658

请选择排序算法：4
希尔排序所用时间：22
希尔排序交换次数：809

请选择排序算法：5
快速排序所用时间：22
快速排序交换次数：1014

请选择排序算法：6
堆排序所用时间：24
堆排序交换次数：1081

请选择排序算法：7
归并排序所用时间：27
归并排序交换次数：1344

请选择排序算法：8
基数排序所用时间：79
基数排序不涉及交换
它的值传递次数为：676

对它的分析如下：

- 1、在小规模测试时快速排序的算法并不具备优势，与常规算法的性能相近。
- 2、由于rand()产生的随机数为10位，故基数排序在这里要进行多位排序，效果很差。
- 3、冒泡排序即使在小规模排序中仍然效果不理想，交换次数是最多的。

4.1.2 中等规模随机数测试

请输入要产生随机数的个数：5000

请选择排序算法：1
冒泡排序所用时间：45441
冒泡排序交换次数：14070180

请选择排序算法：2
选择排序所用时间：31175
选择排序交换次数：14979

请选择排序算法：3
直接插入排序所用时间：**19894**
直接插入排序交换次数：**6245174**

请选择排序算法：4
希尔排序所用时间：**911**
希尔排序交换次数：**101235**

请选择排序算法：5
快速排序所用时间：**770**
快速排序交换次数：**109191**

请选择排序算法：6
堆排序所用时间：**830**
堆排序交换次数：**82144**

请选择排序算法：7
归并排序所用时间：**1006**
归并排序交换次数：**123616**

请选择排序算法：8
基数排序所用时间：**3866**
基数排序不涉及交换
它的值传递次数为：**51192**

对它的分析如下：

- 1、在中等规模时，时间复杂度较低的算法就已经展现出了较好且较快的时间。
- 2、选择排序的比较次数是这当中最少的。
- 3、基数排序在这时仍然没有体现出其优势。

4.1.2 大规模随机数测试

| ** | 排序算法比较 | ** |
|----|--------------|----|
| ** | 1 --- 冒泡排序 | ** |
| ** | 2 --- 选择排序 | ** |
| ** | 3 --- 直接插入排序 | ** |
| ** | 4 --- 希尔排序 | ** |
| ** | 5 --- 快速排序 | ** |
| ** | 6 --- 堆排序 | ** |
| ** | 7 --- 归并排序 | ** |
| ** | 8 --- 基数排序 | ** |
| ** | 9 --- 退出程序 | ** |

请输入要产生随机数的个数：100000

请选择排序算法：1

冒泡排序所用时间：**18574488**

冒泡排序交换次数：**5634446607**

请选择排序算法：2

选择排序所用时间：**9578817**

选择排序交换次数：**299979**

请选择排序算法：3

直接插入排序所用时间：**6148728**

直接插入排序交换次数：**2498020899**

请选择排序算法：4

希尔排序所用时间：**25794**

希尔排序交换次数：**3438921**

请选择排序算法：5

快速排序所用时间：**20320**

快速排序交换次数：**3148994**

请选择排序算法：6

堆排序所用时间：**21993**

堆排序交换次数：**2075178**

请选择排序算法：7

归并排序所用时间：**24704**

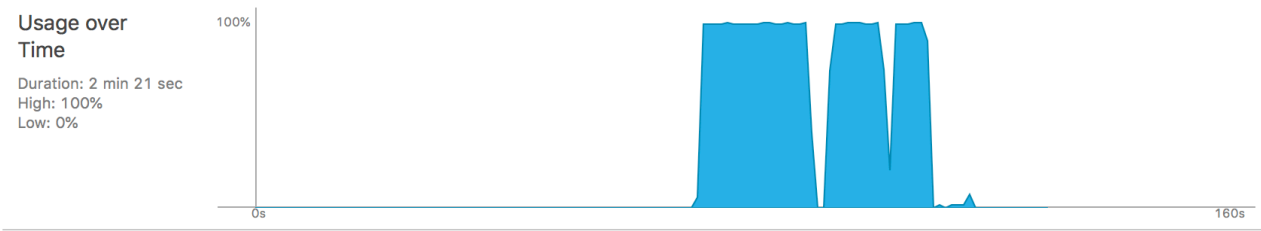
归并排序交换次数：**3337856**

请选择排序算法：8

基数排序所用时间：**78180**

基数排序不涉及交换

它的值传递次数为：**1282072**



上图为CPU的使用率随时间变化图，当采用前三个排序算法时CPU持续保持大功率运转
采用高级算法时，运转时间明显减少，到基数排序又有了一点点起伏

对它的分析如下：

- 1、当随机数个数达到十万个时，时间复杂度的重要性便体现了出来：冒泡排序时间最慢，交换次数最对，选择排序虽然交换次数仍为最少，但比较消耗了大量时间。与之相对，高级算法的运行时间与之形成了数量级差异。
- 2、基数排序仍为在此时体现出它的优势，虽然它的运行时间没有低级算法那么长，但也没有高级算法那么快而是介于两者之间。
- 3、快速排序在随机数达到十万个时并没有出现老师所说的减慢速度的情况，是因为在OSX环境下，产生的随机数位数是Windows环境的两倍，在这种情况下还是很难产生重复数字，故快速排序仍是最好最快的。

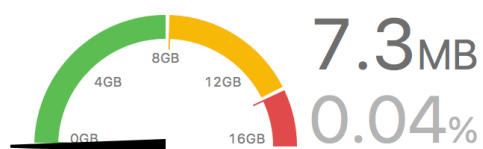
4.2 内存消耗测试

4.2.1 大规模测试

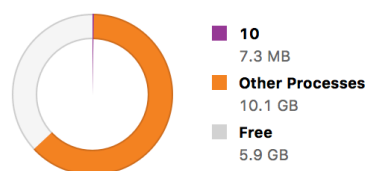
| ** | 排序算法比较 | ** |
|----|--------------|----|
| ** | 1 --- 冒泡排序 | ** |
| ** | 2 --- 选择排序 | ** |
| ** | 3 --- 直接插入排序 | ** |
| ** | 4 --- 希尔排序 | ** |
| ** | 5 --- 快速排序 | ** |
| ** | 6 --- 堆排序 | ** |
| ** | 7 --- 归并排序 | ** |
| ** | 8 --- 基数排序 | ** |
| ** | 9 --- 退出程序 | ** |

请输入要产生随机数的个数：100000

Memory Use



Usage Comparison



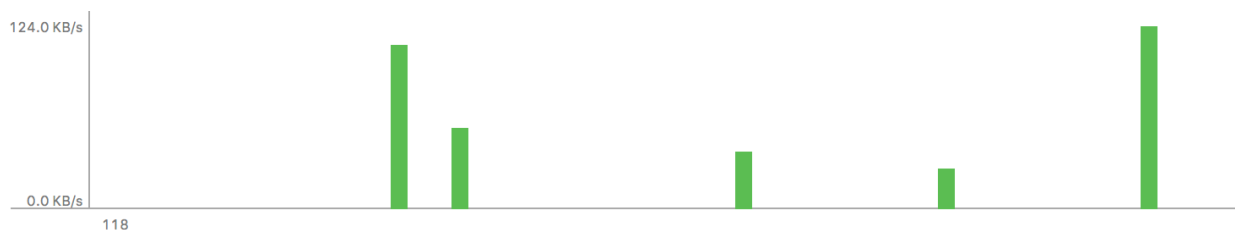
Memory

Duration: 2 min 41 sec
High: 7.3 MB
Low: Zero KB

7.3 MB



Reading and Writing Rates



分析：上图为运行过程中的内存使用率随时间变化情况图，其中后两个归并排序和基数排序由于都使用了辅助数组，故在执行他们时内存使用出现了明显的涨幅。

其中MSD基数排序由于使用了10个递归调用的桶，所以对内存开销较大。

4.3 特殊测试

4.3.1 产生固定一位数的随机数

```
for(int i=0;i<length;i++)List[i]=rand()%(10-1);
```

修改随机产生函数，同时修改基数排序位数，得出结果如下：

请选择排序算法：1

冒泡排序所用时间：**44349**

冒泡排序交换次数：**12996384**

请选择排序算法：2

选择排序所用时间：**30631**

选择排序交换次数：**13293**

请选择排序算法：3

直接插入排序所用时间：**17483**

直接插入排序交换次数：**5529291**

请选择排序算法：4

希尔排序所用时间：**547**

希尔排序交换次数：**44685**

请选择排序算法：5

快速排序所用时间：**4035**

快速排序交换次数：**39688**

请选择排序算法：6

堆排序所用时间：**674**

堆排序交换次数：**76341**

请选择排序算法：7

归并排序所用时间：**792**

归并排序交换次数：**123616**

请选择排序算法：8

基数排序所用时间：**146**

基数排序不涉及交换

它的值传递次数为：**10000**

1、选择排序在此时有了突出于其他低级算法的性能，因为对于数字较小个数较多的数列来说，选择排序比较次数多交换次数少。

2、基数排序由于只有一个桶，故效果是最好的。

3、快速排序在此时面对了大量相同的元素，故速度不如复杂度相同的归并排序和堆排序。

4.3.1 产生固定三位数的随机数

```
for(int i=0;i<length;i++)List[i]=rand()%(999-100); //赋值
```

```
RadSort(List,0,length-1,3,count);
```

修改随机生成函数和基数排序位数，得出结果如下：

请输入要产生随机数的个数：5000

请选择排序算法：1

冒泡排序所用时间：43239

冒泡排序交换次数：14042397

请选择排序算法：2

选择排序所用时间：31183

选择排序交换次数：14967

请选择排序算法：3

直接插入排序所用时间：19903

直接插入排序交换次数：6220796

请选择排序算法：4

希尔排序所用时间：911

希尔排序交换次数：98584

请选择排序算法：5

快速排序所用时间：832

快速排序交换次数：103145

请选择排序算法：6

堆排序所用时间：909

堆排序交换次数：82137

请选择排序算法：7

归并排序所用时间：1108

归并排序交换次数：123616

请选择排序算法：8

基数排序所用时间：615

基数排序不涉及交换

它的值传递次数为：30000

分析：1、由于随机数只有三位，故基数排序在速度上仍然有明显的优势

- 2、对于快速排序，它面对的不同元素减少了，性能又回到了除了基数排序以外最好的
- 3、选择排序的优势消失了，它的性能并不及同为平方级的插入排序

5.总结

对上述结果进行总结如下：

- 1、冒泡排序：始终是交换次数较多且较慢的，除了代码可读性高外没有明显优势
- 2、选择排序：交换次数始终是最少的，但比较次数多，不一定是最优算法，适用于元素排序码较小但个数较多的情况
- 3、插入排序：最稳定的平方级算法，时间不随带排序列改变
- 4、希尔排序：性能接近高级算法，gap取得好会有奇效
- 5、快速排序：面对重复简单元素时效率极差，但除此之外几乎是最好的内排序算法
- 6、堆排序：速度次于快速排序，但也是性能很高的算法
- 7、归并排序：不受序列影响的高性能算法，但要考虑内存开销
- 8、基数排序：在位数少数字多多情况下性能极好，优于所有高级排序算法。但在一般情况下性能介于低级算法和高级算法之间。