1. 编写程序实现归并排序算法 MergeSortL 和快速排序算法 QuickSort;

2. 用长分别为 100、200、300、400、500、600、700、800、900、1000 的 10 个数组的排列来统计这两种算法的时间复杂性;

3. 讨论归并排序算法 MergeSort 的空间复杂性。

4. 说明算法 PartSelect 的平均时间复杂性为 $O(n)$。

Merge Sort L

```
}
private static void sort(int[] arr,int left,int right,int []temp){
    if(left<right){
        int mid = (left+right)/2;
        sort(arr,left,mid,temp);//左边归并排序，使得左子序列有序
        sort(arr,mid+1,right,temp);//右边归并排序，使得右子序列有序
        merge(arr,left,mid,right,temp);//将两个有序子数组合并操作
    }
}
private static void merge(int[] arr,int left,int mid,int right,int[] temp){
    int i = left;//左序列指针
    int j = mid+1;//右序列指针
    int t = 0;//临时数组指针
    while (i<=mid && j<=right){
        if(arr[i]<=arr[j]){
            temp[t++] = arr[i++];
        }else {
            temp[t++] = arr[j++];
        }
    }
    while(i<=mid){//将左边剩余元素填充进temp中
        temp[t++] = arr[i++];
    }
    while(j<=right){//将右序列剩余元素填充进temp中
        temp[t++] = arr[j++];
    }
    t = 0;
    //将temp中的元素全部拷贝到原数组中
    while(left <= right){
        arr[left++] = temp[t++];
    }
}
```

Quicksort

```c
//arr:需要排序的数组，begin:需要排序的区间左边界，end:需要排序的区间的右边界
void quickSort(int *arr,int begin,int end)
{
        //如果区间不只一个数
        if(begin < end)
        {
                int temp = arr[begin]; //将区间的第一个数作为基准数
                int i = begin; //从左到右进行查找时的“指针”，指示当前左位置
                int j = end; //从右到左进行查找时的“指针”，指示当前右位置
                //不重复遍历
                while(i < j)
                {
                        //当右边的数大于基准数时，略过，继续向左查找
                        //不满足条件时跳出循环，此时的j对应的元素是小于基准元素的
                        while(i<j && arr[j] > temp)
                                j--;
                        //将右边小于等于基准元素的数填入右边相应位置
                        arr[i] = arr[j];
                        //当左边的数小于等于基准数时，略过，继续向右查找
                        //(重复的基准元素集合到左区间)
                        //不满足条件时跳出循环，此时的i对应的元素是大于等于基准元素的
                        while(i<j && arr[i] <= temp)
                                i++;
                        //将左边大于基准元素的数填入左边相应位置
                        arr[j] = arr[i];
                }
                //将基准元素填入相应位置
                arr[i] = temp;
                //此时的i即为基准元素的位置
                //对基准元素的左边子区间进行相似的快速排序
                quickSort(arr,begin,i-1);
                //对基准元素的右边子区间进行相似的快速排序
                quickSort(arr,i+1,end);
        }
        //如果区间只有一个数，则返回
        else
                return;
}
```

改进插入排序算法(第三章 ppt No.6)，在插入元素 a(i)时使用二分查找代替顺序查找，将这个算法记做 ModInsertSort，估计算法在最坏情况下的时间复杂度。

```
ModInsertSort(A[1..n])
    for i:=2 to n do
        x:=A[i]
        k:=BinarySearch(A[1..i-1],x)    //在 A[1..i-1]查找 x 应该插入的位置 k
        A[k]:=x
```

```
int BinarySearch(int *a,int left, int right,int k)//k是要找的数字

{

    int m = left + (right - left) / 2;

    if (left > right)//查找完毕没有找到答案，返回-1

        return -1;

    else

    {

        if (a[m] == k)

            return m;//找到!返回位置.

        else if (a[m] > k)

            return BinarySearch(a,left, m - 1,k);//找左边

        else

            return BinarySearch(a,m + 1, right,k);//找右边

    }
```


复杂度分析：外 for 循环 n-1 次，在 i-1 规模的数组中二分比较次数为<=log(i-1)+1，因此总比较次数为：（但移动次数没节省）

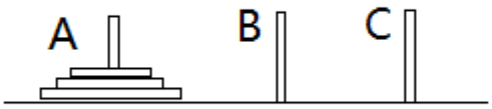
$$\sum_{i=2}^{n} \log(i-1)+1) = n-1+\sum_{i=1}^{n-1} \log i \leq n-1+\sum_{i=1}^{n-1} \log n = n-1+(n-1)\log n = O(n\log n)$$

设 A 是 n 个非 0 实数构成的数组，设计一个算法重新排列数组中的数，使得负数都排在正数前面，要求算法复杂度为 O(n)。

类似快速排序的划分过程。从后向前把每个数与 0 比较，找到第一个负数 A[p];从前向后把每个数与 0 比较，找到第一个正数 A[q]，如果 P>q，则将 A[p]与 A[q]交换。交换后如果 p-q=1，算法停止，否则继续这个过程。

```
void reorder(int array[], int length)
{
        int low = 0, high = length - 1;
        int temp;
        while (low < high)
        {
            while (array[low] < 0 && low < high)
                low++;
            while (array[high] > 0 && low < high)
                high--;
            temp = array[low];
            array[low] = array[high];
            array[high] = temp;
        }
```

Hanoi 塔问题。图中有 A、B、C 三根柱子，在 A 柱上放着 n 个圆盘，其中小圆盘放在大圆盘的上边。从 A 柱将这些圆盘移到 C 柱上去，在移动和放置时允许使用 B 柱，但不能把大盘放到小盘的下面。设计算法解决此问题，分析算法复杂度。




```
void hanoi(int i , char A , char B , char C)
{
    if(i == 1)
    {
        move(i , A , C);
    }
    else
    {
        hanoi(i - 1 , A , C , B);   //函数递归调用
        move(i , A , C);
        hanoi(i - 1 , B , A , C);
    }
}
```

算法复杂度：T(n)=2T(n-1)+1=4T(n-2)+2+1=2n-1+2n-2+…+1=2n-1

给定含有 n 个不同数的数组 L={x1,x2,…,xn}，如果 L 中存在 xi，使得 x1<x2<…<xi-1<xi>xi+1>…>xn，则称 L 是单峰的，并称 xi 是 L 的峰顶。假设 L 是单峰的，设计一个优于 O(n)的算法找到 L 的峰顶。

因为 L 中存在峰顶元素，因此|L|>=3。使用二分查找算法。如元素数等于 3，则 L[2]是峰顶元素，当元素数 n>3 时，令 k=⌊n/2⌋，比较 L[k]与它左边和右边相邻的项，如果 L[k]>L[k-1]且 L[k]>L[k+1]则 L[k]为峰顶元素；否则，如果 L[k-1]>L[k]>L[k+1]，则继续搜索 L[1..k-1]，如果 L[k-1]<L[k]<L[k+1]则继续搜索 L[k+1..n]的范围。每比较两次，搜索范围减半，直到元素数小于 3 停止递归调用。

时间复杂度 T(n)=T(n/2)+2，根据主定理，T(n)=O(logn)。

```
int siglePeak(int arr[],int left,int right) {

    int mid = left + (right- left)/2;//x + (y - x) / 2

    if(arr[mid]> arr[mid-1]&& arr[mid]> arr[mid+1]) return arr[mid];

    else{

        if(arr[mid-1] < arr[mid]&& arr[mid-2]< arr[mid-1])

            siglePeak(arr,mid+1,right); // 右边

        else if(arr[mid] > arr[mid+1] && arr[mid +1] > arr[mid+2]){

            siglePeak(arr,left,mid-1);
```

设 A 是 n 个不同的排好序的数组，给定数 L 和 U，L<U，设计一个优于 O(n)的算法，找到 A 中满足 L<x<U 的所有数 x。

在 A 中使用二分查找算法找 L，如果 L=A[i]，找到 L 的位置 i，然后把 i 加 1;如果 L 不在 A 中，那么找到大于 L 的最小数的位置 i。类似地，找到 U 的位置 A[j]，j=j+1，或小于 U 的最大数 A[j]。输出 A 中 i 到 j 的全体数。

```
void lower_index(int *a, int start, int end, int l, int u)
{
    //find the lower.
    if (start > end)
        return;
    else if (start == end)
    {
        if (a[start] > l)
            lower = start;
        return;
    }
    int mid = (start + end) / 2;
    if (a[mid] == l)
    {
        lower = mid + 1;
        return;
    }
    else if (a[mid] < l)
    {
        lower_index(a, mid + 1, end, l, u);
        return;
    }
    else
    {
        lower_index(a, start, mid, l, u);
        return;
    }
}

void higher_index(int *a, int start, int end, int l, int u)
{
    //cout << start << " " << end << " " << endl;
    //find the higher.
    if (start > end)
        return;
    else if (start == end)
    {
        if (a[start] < u)
            higher = start;
        return;
    }
    int mid = (start + end) / 2 + 1;
    if (a[mid] == u)
    {
        higher = mid - 1;
        return;
    }
    else if (a[mid] < u)
    {
        higher_index(a, mid, end, l, u);
        return;
    }
    else
    {
        higher_index(a, start, mid - 1, l, u);
        return;
    }
}
void cal(int *a, int start, int end, int l, int u)
{
    if (start <= end)
    {
        int mid = (start + end) / 2;
        if (u <= a[mid])
        {
            cal(a, start, mid - 1, l, u);
            return;
        }
        else if (l >= a[mid])
        {
            cal(a, mid + 1, end, l, u);
            return;
        }
        else
        {
            result[len++] = a[mid];
            cal(a, start, mid - 1, l, u);
            cal(a, mid + 1, end, l, u);
            return;
        }
    }
}
```

.设 A={a1,a2,…,an},B={b1,b2,…,bm}是整数集合，其中 m=O(logn)，设计一个优于 O(nm)的算法法找出集合 C=A∩B。(即找出一个优于 nlogn 的方法）所得方法为 nlogm

3. A∩B

设A={a1,a2,…,an},B={b1,b2,…,bm}是整数集合，其中m=O(logn)，设计一个优于O(nm)的算法找出集合C=A∩B。

答：可以将长度较小的数组排序(由于m=O(logn)，可知m较小)，可选择复杂度较小的排序算法如归并排序，所需时间复杂度为O(mlogm)，再将每个B集合的每个元素作为Key，利用二分搜索去排好序的A数组中寻找，查找for循环运行n次，内部的二分搜索O(logm)，for循环关键操作O(nlogm)，总的时间复杂度T(n)= O(mlogm)+ O(nlogm)= O((m+n)logm)= O(nlogm)

```
void mergeSort(int *arr,int low, int high) {

    // arr[low..high]是一个全程数组，含有high-low+1个待排序元素

    if (low < high) {

        int mid = (low + high) / 2;

        mergeSort(arr,low, mid);

        mergeSort(arr,mid + 1, high);

        merge(arr,low, mid, high);
```

```
void merge(int *arr,int low, int mid, int high) {

    //merge函数将两部分，（分别排好序的子数组），合并成一个，再存于原数组arr

    int h, i, j, k;

    h = low; i = low; j = mid + 1; // h,j作为游标，i指向temp存放数组元素的游标

    vector<int> temp(5); // 这个temp数组是local的,长度为数组arr长度

    while (h <= mid && j <= high) { // 当两个集合都没有取尽时

        if (arr[h] <= arr[j]) {

            temp[i] = arr[h];

            h = h + 1;

        }

        else {

            temp[i] = arr[j];

            j++;

        }

        i++;

    }

    if (h > mid) { // 当第一子组元素被 取尽，而第二组元素未被取尽时

        for (int k = j; k <= high; k++) {

            temp[i] = arr[k];

            i++;

        }

    }

    else { // 当第2组取尽，第一组未被取尽时

        for (int k = h; k <= mid; k++) {

            temp[i] = arr[k];

            i++;

        }

    }

    for (int k = low; k <= high; k++) {//将临时数组temp中的元素再赋值给arr

        arr[k] = temp[k];

    }

}
```

```
int BinarySearch(int *a,int left, int right,int k)//k是要找的数字

{

    int m = left + (right - left) / 2;

    if (left > right)//查找完毕没有找到答案，返回-1

        return -1;

    else

    {

        if (a[m] == k)

            return m;//找到!返回位置.

        else if (a[m] > k)

            return BinarySearch(a,left, m - 1,k);//找左边

        else

            return BinarySearch(a,m + 1, right,k);//找右边

    }
```

算法复杂度：找中位数和划分都是 O(n)，所以 T(n)=O(n)。

补充算法实现

```
int main() {

    int a[] = { 1,5,4,7,8 };

    int b[] = { 4,5,6,10,1,2,3,8,12 };

    vector<int> c ;

    int count=0;

    mergeSort(a,0,4); //归并排序 O(nlogn)

    for (int i = 0; i < 9;i++) {

        int temp=BinarySearch(a, 0, 4, b[i]);

        if (temp != -1) {

            c.push_back(a[temp]);

            count++;

        }

    }

    for (int i = 0; i < c.size();i++) {

        cout << c[i]<<" , ";

    }

         return 0;

}
```

设 S 是 n 个不等的正整数的集合，n 为偶数，给出一个算法将 S 划分为子集 S1 和 S2，使得 |S1|=|S2|且 $|\sum_{x \in S_1} x - \sum_{x \in S_2} x|$ 达到最大，即两个子集元素之和的差达到最大。(要求：T(n)=O(n))。

规定 S 的中位数 x 是从小到大排序的第 n/2 个数，用 x 划分 S，比 x 小的整数属于 S1，x 本身也放到 S1，其余的放到 S2，由于 n 是偶数，|S1|=|S2|，易见这样的集合满足要求。

算法复杂度：找中位数和划分都是 O(n)，所以 T(n)=O(n)。

补充算法实现

考虑第三章 PPT NO.17 Select(A,k)算法：

（1）如果初始元素分组 r=3，算法的时间复杂度如何?

(2) 如果初始元素分组 r=7，算法的时间复杂度如何?

选择问题

- 改进划分选择算法:改变划分元素,使每次划分规模减半
  - Select(A,k) //输入A[1..n]，输出其第k小元素
  - 1. 将A划分为r(如r=5)个一组，共$\lceil n/5 \rceil$ 个组(取$\lfloor n/5 \rfloor$个中位)。
  - 2. 每组找一个中位数，把它们放到集合M中。
    //可使用排序法,时间复杂度O(C)
  - 3. m*:=select(M,$\lceil |M|/2 \rceil$)　　//选M的中位数,A[m]=m*
  - 4. 用m*划分A，小的A1，大的A2；　　//Partition(m,j)
  - 5. if k=|A1|+1 then 输出m*;
  - 6. else if k ≤ |A1| then
  - 7. Select(A1,k)
  - 8. else Select(A2,k-|A1|-1)

选择问题

Select的另一种描述

```
proc Select(A, m, p, k) // 返回一个i值，
//使得A[i]是A[m..p]中第k小元素。 r常数.
  global r;   integer n, i, j;
  if p-m+1≤r then
    InSort(A, m, p);
    return(m+k-1);
  end{if}
  loop
    n:=p-m+1;
    for i =1 to ⌊n/r⌋ do //计算中间值
      InSort(A, m+(i-1)*r, m+i*r-1);
      //对五元组进行排序
      //将中间值收集到A[m..p]的前部:
      Swap(A[m+i-1], A[m+(i-1)*r+⌈r/2⌉-1]);
将每个五元组里面的值放到头部
    end{for}
    j:=Select(A, m, m+⌊n/r⌋-1, ⌈⌊n/r⌋/2⌉);
    Swap(A[m], A[j]); //产生划分元素
    j:=p+1;
    Partition(m, j);
    case:
      j-m+1=k :  return(j);
      j-m+1>k :  p:=j-1;
      else  k:=k-(j-m+1); m:=j+1;
    end{case}
  end{loop}
  end{Select}
```

选择问题

- Select的复杂度分析
  - 设数组A中的元素都是互不相同的，取 r=5。
  - 则5个元素一组的中间值u是该数组的第3小元素，此数组至少有3个元素不大于u；$\lfloor n/5 \rfloor$个中间值中至少有$\lceil \lfloor n/5 \rfloor /2 \rceil$个不大于这些中间值的中间值v。因而，
  - 在数组A中至少有$3*\lceil \lfloor n/5 \rfloor /2 \rceil \geq 1.5*\lfloor n/5 \rfloor \geq 1.5*(n/5-1)$个元素不大于v。即，A中至多有 $n-1.5*(n/5-1) = 0.7n+1.5$个元素大于v。
  - 同理，至多有0.7n+1.5个元素小于v。
  - 这样，以v为划分元素所产生的新的数组至多有0.7n+1.5个元素. 当n≥30时,$0.7n+1.5 \leq 0.75n=3n/4$，即子问题的规模为3n/4。
  - Select中，j:= m*:=select(M, $\lceil |M|/2 \rceil$)规模为n/5;IF语句后的循环调用规模3n/4,Partition和Insort关键操作cn。得：
  - $T(n) \leq T(n/5)+T(3n/4)+cn \leq cn+0.95cn+0.95^2cn+\ldots+0.95^kcn$
    所以 **T(n)=O(n)**　　$=cn(1-0.95^{k+1})/(1-0.95) \leq 20cn$　（n≥30）

(1)r=3 ，不妨设 n 是 3 的倍数。每组至少 2 个元素不大于 u，A 中至少 $2*\lceil \lfloor n/3 \rfloor /2 \rceil >= \lfloor n/3 \rfloor = n/3$ 个不大于 v， 即 A 中至多 $n-\lfloor n/3 \rfloor <= n-n/3 = 2n/3$ 个元素大于 v。同理，至多有 2n/3 个元素小于 v。即子问题的规模小于 2n/3。所以， T(n)=T(n/3)+T(2n/3)+O(n)，得 T(n)=O(nlogn)。与直接排序方法的复杂度一样。

(2)问题变为 $4*\lceil \lfloor n/7 \rfloor /2 \rceil >= 2\lfloor n/7 \rfloor$，子问题规模小于 n-2n/7=5n/7（不妨设 n 是 7 的倍数）

T(n)=T(n/7)+T(5n/7)+O(n)　=n(1+6/7+(6/7)2+…+(6/7)k)+O(n)=O(n)

11.对玻璃瓶做强度试验，设地面高度为 0，从 0 向上有 n 个高度，记为 1,2,…,n，其中任何两个高度之间的距离都相等。如果一个玻璃瓶从高度 i 落到地上没有摔碎，但从高度 i+1 落到地上摔碎了，那么就将玻璃瓶的强度记为 i。

(1)假设每种玻璃瓶只有 1 个测试样品，设计算法来测试出每种玻璃瓶的强度。以测试次数作为算法的时间复杂度量度，估计算法的复杂度。

(2)假设每种玻璃瓶有足够多的相同的测试样品，设计算法使用最少的测试次数来完成测试。

(3)假设每种玻璃瓶只有 2 个相同的测试样品，设计次数尽可能少的算法完成测试。

解：(1)只好顺序从下到上测试，一次一个高度，最坏 T(n)=O(n)

(2)二分查找法。取 n/2 高度进行第一次测试，如果瓶子没有摔碎，则强度在[n/2+1,n]之间，否则在[1,n/2]之间。每次测试后可能一个瓶子的代价，测试范围减半，最坏时间复杂度 T(n)=O(logn)。

(3) 为简单起见，不妨设 $\sqrt{n}$ 为整数，将高度 1,2,..,n 分为 $\sqrt{n}$ 个组，每组 $\sqrt{n}$ 高度，取第一个瓶子从下到上测试每组的最大高度，即高度 $\sqrt{n}$,2$\sqrt{n}$ …n，如果 k-1 组没碎，k 组碎了，那么玻璃瓶子的强度在第 k 组内，于是，再经至多 $\sqrt{n}$ 次测试，就可以得到瓶子的强度。

T(n)=O($\sqrt{n}$)+O($\sqrt{n}$)=O($\sqrt{n}$)

12. 使用主定理求解以下递归方程：

(1) $\begin{cases} T(n)=9T(n/3)+n \\ T(1)=1 \end{cases}$ (2) $\begin{cases} T(n)=5T(n/2)+(n\log n)^2 \\ T(1)=1 \end{cases}$

(3) $\begin{cases} T(n)=2T(n/2)+n^2\log n \\ T(1)=1 \end{cases}$

(1)a=9,b=3,f(n)=n，log39=2,f(n)的阶低于 nlogba,符合情况 1，T(n)=Θ(nlogba)= Θ(n2)。

(2)a=5,b=2,f(n)=n2log2n=O(nlog25-ε)，方法 T(n)= Θ(nlog25)

(3)a=2,b=2,f(n)=n2logn,取 c=3/4 则

af(n/b)=2(n/2)2log(n/2)=(n2/2)(logn-1)≤(n2/2)logn≤cn2logn=cf(n)

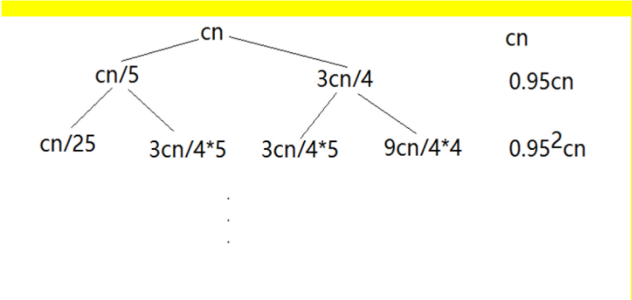
于是，符合情况 3，T(n)= Θ(n2logn)

使用递归树求解：$\begin{cases} T(n)=T(n/2)+T(n/4)+cn \\ T(1)=1 \end{cases}$

T(n)=cn+3cn/4+(3/4)2cn+(3/4)3cn+…..=[1+3/4+(3/4)2+(3/4)3+…]cn=Θ(n)

例：

T(n) ≤T(n/5)+T(3n/4)+cn ≤cn+0.95cn+0.95^2cn+…+0.95^kcn



T(n)的递归树

使用迭代递归法求解：

(1) $\begin{cases} T(n) = T(n-1) + \log 3^n \\ T(1) = 1 \end{cases}$ (2) $\begin{cases} \mathrm{T}(n) = T(n-1) + 1/n \\ T(1) = 1 \end{cases}$

(1)T(n)=nlog3+(n-1)log3+T(n-2)=log3(n+(n-1)+(n-2)+…+1)= Θ(n2)

(2)T(n)=T(n-1)+1/n=T(n-2)+1/(n-1)+1/n=…=**1/n+1/(n-1)+…+1/2+1=Θ(logn)—相等**