程序设计方法学

期末课程项目

2024-2025学年第二学期, 同济大学

课程编号：101031

任课教师：王小平 教授

课程期末项目任务：排序算法程序的正确性证明

- 选择以下排序算法之一（对一数组按升序排列）
  A. 选择排序（可用递归算法）
  B. 冒泡排序（可用递归算法）
  C. 插入排序（可用递归算法）
  D. 快速排序（用递归算法）
  E. 归并排序（用递归算法）
  F. 堆排序（用递归算法）
- 程序正确性证明方法选择以下一种：
  1. Floyd 的不变式断言法（部分正确性）
  2. Floyd的良序集方法(程序终止性))
  3. Hoare的公理化方法（正向或反向证明）
  4. Dijkstra的最弱前置谓词转换方法(完全正确性)
  5. 递归程序的良序归纳法(完全正确性)

课程期末项目任务：排序算法程序的正确性证明

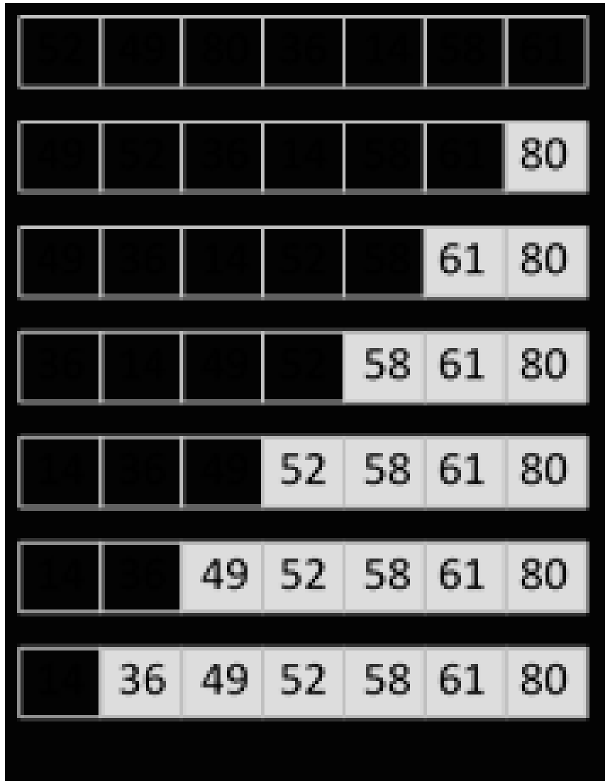
- 要求:
  - 写出问题描述和程序规范，编写排序算法的程序（伪代码形式）
  - 画出结构化程序的流程图
  - 写出完整的证明过程
- 提交形式
  - 第14周统一时间确认选题，一人一题，不允许重复。
  - 请于16周三前完成，提交一份PPT演示稿的电子版发给老师: xpwang01@163.com，准备课堂答疑和讨论。
  - 第17周三课堂汇报10分钟。
  - 课程结束前（17周周三课间）每人提交纸质版（至少10页）报告, 期末课程最终成绩按课程大作业和平时成绩综合评价

冒泡排序

(1) 设计思路

“依次比较相邻元素，‘逆序’则交换，重复n-1次” 。

例：冒泡排序(52，49，80，36，14，58，61)。



(2) 程序

(3) 分析

比较和交换总是发生在相邻元素之间，是稳定的排序算法。
时间复杂度$O(n^2)$。

冒泡排序

```
void bubble_sort(int a[],int length)
{
  int i,j,x,l;
 for (i=0;i<length;i++)
      for (j=0;j<len-i;j++)   {
          if (a[j]>a[j+1])  {
              temp=a[j+1];
              a[j+1]=a[j];
              a[j]=temp;  }
      }
}
```

```
Bubble_sort(A)
{
    for i=1 to n
        for j= n to i+1
            if A[j]<A[j-1]
                swap A[j]<->A[j-1]
}
```

冒泡排序

内循环不变式：在每次循环开始前，A[j]是A[j...n]中最小的元素。

初始：j=n，因此A[n]是A[n...n]的最小元素。

保持：当循环开始时，已知A[j]是A[j...n]的最小元素，将A[j]与A[j-1]比较，并将较小者放在j-1位置，因此能够说明A[j-1]是A[j-1...n]的最小元素，因此循环不变式保持。

终止：j=i，已知A[i]是A[i...n]中最小的元素，

外循环不变式：在每次循环之前，A[1...i-1]包含了A中最小的i-1个元素，且已排序：A[1]<=A[2]<=...<=A[i-1]。

初始：i=1，因此A[1..0]=空，因此成立。

保持：当循环开始时，已知A[1...i-1]是A中最小的i-1个元素，且A[1]<=A[2]<=...<=A[i-1]，根据内循环不变式，终止时A[i]是A[i...n]中最小的元素，因此A[1...i]包含了A中最小的i个元素，且A[1]<=A[2]<=...<=A[i-1]<=A[i]

终止：i=n+1，已知A[1...n]是A中最小的n个元素，且A[1]<=A[2]<=...<=A[n]

改进的冒泡排序

```
improved_bubble_sort(A)
{
    for i=1 to n-1
        if flag==false return;
        flag=false;
        for j=n to i+1
            if A[j]<A[j-1]
                swap A[j]<->A[j-1]
                flag = true;
}
```

递归版冒泡排序

```
recursive_bubblesort(A,p,q)
{
      if p<q
      {
            findmin(A,p,q); //Divide
            recursive_bubblesort(A,p+1,q);  //Conquer
      }
}
findmin(A,p,q)
{
      for i=q to p+1
            if A[i]<A[i-1]
                  swap A[i]<->A[i-1]
}
```

选择排序

(1) 思路

第i趟排序过程是在剩余的待排记录中选一个最小（大）的，放在第i个位置。

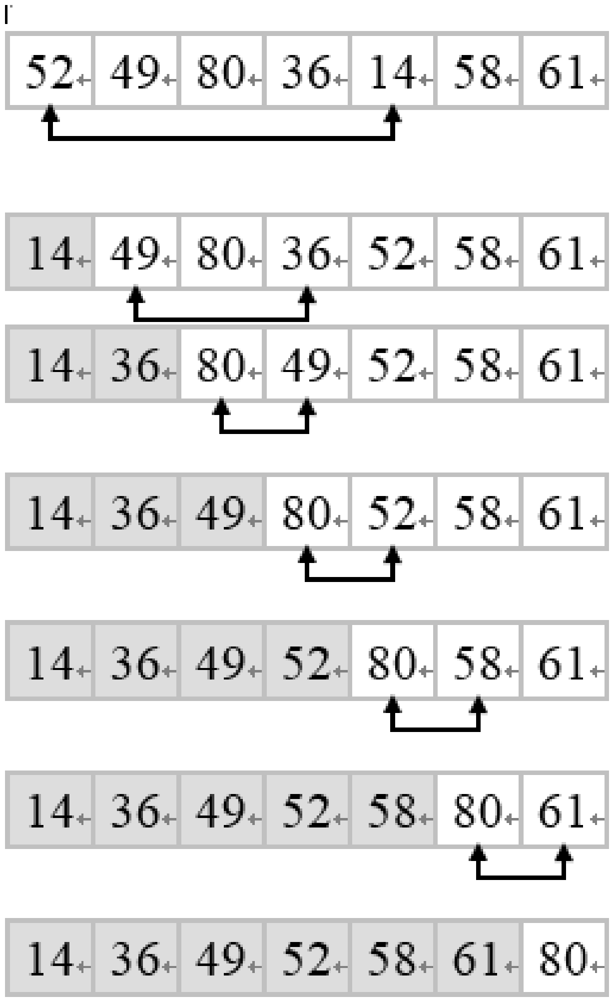
“在待排记录中选取最小的，交换到合适位置，重复n-1次”。

例：{52 49 80 36 14 58 61}简单选择排序。

(2) 程序

(3) 分析

时间复杂度$O(n^2)$，耗费在比较记录上，比较次数始终为n(n-1)/2，移动次数最小为0，最大3(n-1)，即n-1次交换。



选择排序

```
void select_sort(int a[],int length)
{
    int i,j,x,l;
    for(i=0;i<length;i++)  {
        x=a[i];
        l=i;
        for(j=i;j<length;j++)  {
            if(a[j]<x)   {
                x=a[j];
                l=j;  }
        }
        a[l]=a[i];
        a[i]=x;
    }
}
```

```
selection_sort(A)
{
    for i=1 to n-1
        min=i;
        for j=i+1 to n
            if A[min]>A[j]
                min = j;
        swap A[min]<->A[i]
}
```

选择排序

循环不变式：A[1...i-1]包含了A中最小的i-1个元素，且已排序。

初始：i=1，A[1...0]=空，因此成立。

保持：在某次迭代开始之前，保持循环不变式，即A[1...i-1]包含了A中最小的i-1个元素，且已排序，则进入循环体后，程序从A[i...n]中找出最小值放在A[i]处，因此A[1...i]包含了A中最小的i个元素，且已排序，而i++，因此下一次循环之前，保持：循环不变式：A[1..i-1]包含了A中最小的i-1个元素，且已排序。

终止：i=n，已知A[1...n-1]包含了A中最小的i-1个元素，且已排序，因此A[n]中的元素是最大的，因此A[1...n]已排序

递归版选择排序

```
recursive_selectionsort(A,p,q)
{
    if p<q
    {
        min=find_min(A,p,q);    //从A[p...q]中找出最小的与A[p]交换
        swap A[p]<->A[min]
        recursive_selectionsort(A,p+1,q);   //对A[p+1...q]排序
    }
}
find_min(A,p,q)
{
    min = p
    for i=p+1 to q
        if A[min]>A[i]
            min = i
    return min
}
```

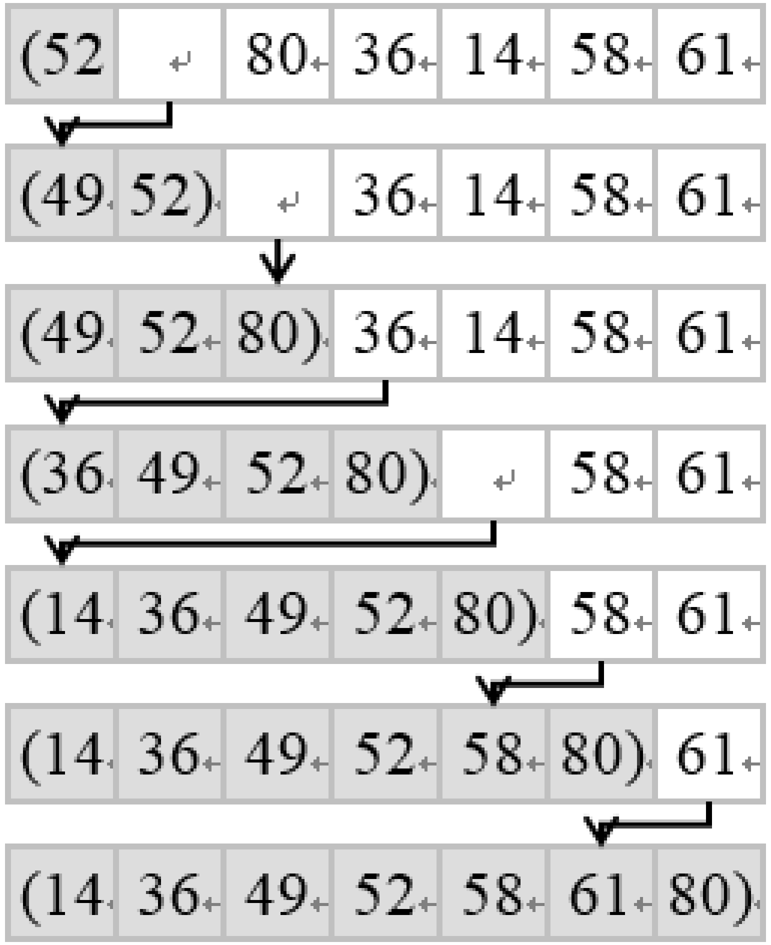
直接插入排序

思路

将待排序记录插入已排好的记录中，不断扩大有序序列

“将待排序记录插入有序序列，重复n-1次”。

例：52，49，80，36，14，58，61 进行直接插入排序。



	比较	移动	
记录顺序有序时	n-1	0	最好
记录逆序有序时	((n+2)(n-1))/2	((n+4)(n-1))/2	最坏

平均 $n^2/4$，算法的时间复杂度 $O(n^2)$。直接插入排序是稳定的排序算法。

```
Insertion_Sort(A)
{
    for i=2 to n
        j = i-1
        key = A[i]
        while j>0 && A[j]>key
            A[j+1] = A[j]
            j--
        A[j+1] = key
}
```

循环不变式：在每次循环开始前，A[1...i-1]包含了原来的A[1...i-1]的元素，并且已排序。

初始：i=2，A[1...1]已排序，成立。
保持：在迭代开始前，A[1...i-1]已排序，而循环体的目的是将A[i]插入A[1...i-1]中，使得A[1...i]排序，因此在下一轮迭代开始前，i++，因此现在A[1...i-1]排好序了，因此保持循环不变式。
终止：最后i=n+1，并且A[1...n]已排序，而A[1...n]就是整个数组

递归版插入排序

```
Recursive_InsertionSort(A,p,q)
{
    if p<q
        Recursive_InsertionSort(A,p,q-1);//递归将A[p...q-1]排序
        Insert(A,p,q-1);
}
Insert(A,p,q)
{
    key = A[q+1]
    j=q
    while j>0 && A[j]>key
        A[j+1]=A[j]
        j--
    A[j+1]=key
}
```

折半插入排序

(1) 思路

在直接插入排序中，查找插入位置时采用折半查找的方法。

(2) 程序

```
void BinInsertSort ( T a[], int n )
{
  for ( i=1; i<n; i++ ) {
    // 在a[0..i-1]中折半查找插入位置使a[high]≤a[i]<a[high+1..i-1]
    low = 0;  high = i-1;
    while ( low<=high ) {
      m = ( low+high )/2;
      if ( a[i]<a[m] )
        high = m-1;
      else
        low = m+1;
    }
    // 向后移动元素a[high+1..i-1]，在a[high+1]处插入a[i]
    x = a[i];
    for ( j=i-1; j>high; j-- )
      a[j+1] = a[j];
    a [high+1] = x;   // 完成插入
  }
}
```

(3) 分析

时间复杂度O(n2)。比直接插入排序减少了比较次数。折半插入排序是稳定的排序算法。

快速排序

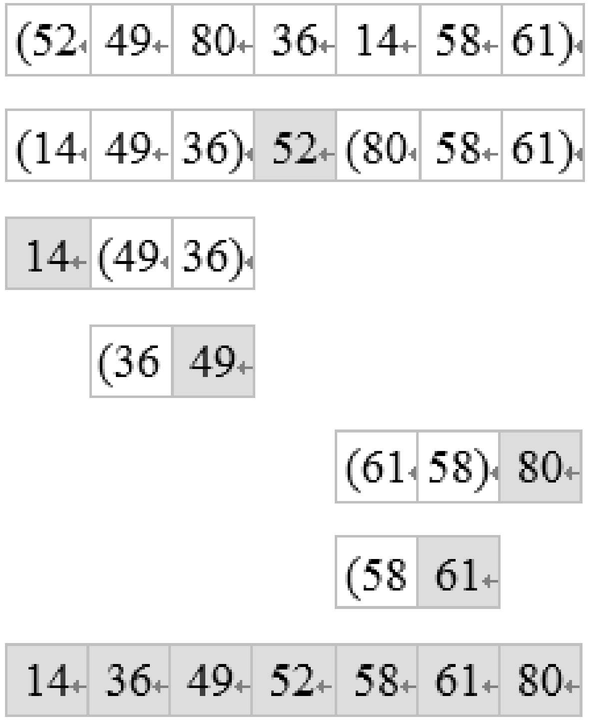
(1) 思路

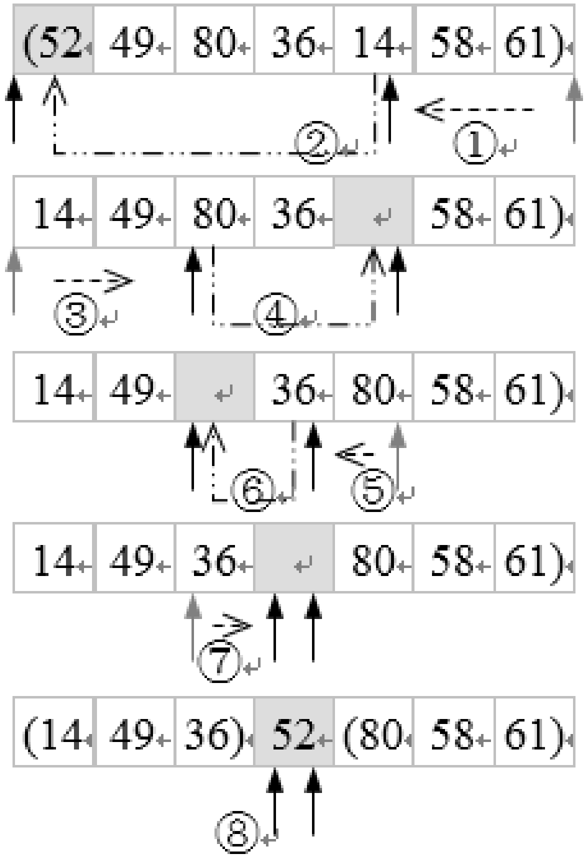
一趟排序把记录分割成独立的两部分，一部分关键字均比另一部分小，然后再分别对两部分快排。

例：{52 4

快速排序。

下面是一





技巧：

选第 1 个记录为轴，分别从后向前，从前向后扫描记录，后面“抓大放小”(如：①②)，前面“抓小放大”(如：③④)，交替进行(⑤-⑦)，最后将轴

快速排序

(2) 程序

```
void QuickSort ( T a[], int low, int high )
{
 if ( low < high ) {     // 划分
  pivot = a[low];
  i = low; j = high;
  while ( i < j ) {
   while ( i<j && a[j] >= pivot )  j--;
   a[i] = a[j];
   while ( i<j && a[i] <= pivot )  i++;
   a[j] = a[i];
  }
  a[i] = pivot;    // 对子序列快排
  QuickSort ( a, low, i-1);
  QuickSort ( a, i+1, high);
 }
}
```

快速排序

在最优情况下，Partition每次都划分得很均匀，如果排序n个关键字，其递归树的深度就为$\log_2 n+1$，即仅需递归$\log_2 n$次，假设需要时间为T（n）的话，第一次Partiation应该是需要对整个数组扫描一遍，做n次比较。然后，获得的枢轴将数组一分为二，那么各自还需要T（n/2）的时间（注意是最好情况，所以平分两半）。于是不断地划分下去，我们就有了下面的不等式推断：

$T(n)\le 2T(n/2)+n, \quad T(1)=0$

$T(n)\le 2(2T(n/4)+n/2)+n = 4T(n/4)+2n$

$T(n)\le 4(2T(n/8)+n/4)+2n=8T(n/8)+3n$

……

$T(n)\le nT(1)+(\log_2 n)\times n= O(n\log_2 n)$

快速排序

在最坏的情况下，待排序的序列为正序或者逆序，每次划分只得到一个比上一次划分少一个记录的子序列，注意另一个为空。如果递归树画出来，它就是一棵斜树。此时需要执行n-1次递归调用，且第i次划分需要经过n-i次关键字的比较才能找到第i个记录，也就是枢轴的位置，因此比较次数为

$$\sum_{i=1}^{n-1}(n-i)=n-1+n-2+\cdots+1=\frac{n(n-1)}{2}$$

最终其时间复杂度为O(n^2)。

快速排序

平均的情况，设枢轴的关键字应该在第k的位置（1≤k≤n），那么：

$$T(n)=\frac{1}{n}\sum_{k=1}^{n}\big(T(k-1)+T(n-k)\big)+n=\frac{2}{n}\sum_{k=1}^{n}T(k)+n$$

由数学归纳法可证明，其数量级为O(nlogn)。

就空间复杂度来说，主要是递归造成的栈空间的使用，最好情况递归树的深度为$\log_2 n$，其空间复杂度也就为$O(\log_2 n)$，最坏情况，需要进行n-1递归调用，其空间复杂度为O(n)，平均情况空间复杂度也为$O(\log_2 n)$。

快速排序

```
quick_sort(A)
{
    recursive_quicksort(A,1,length[A])
}
recursive_quicksort(A,p,q)
{
    if p<q
        r = partition(A,p,q)
        recursive_quicksort(A,p,r-1)
        recursive_quicksort(A,r+1,q)
}
partition(A,p,q)
{
    i = p-1
    pivot = A[q]
    for j=p to q-1
        if A[j]<=pivot
            i++
            swap A[i]<->A[j]
    swap A[i+1]<->A[q]
    return i+1
}
```

快速排序

对partition函数证明循环不变式：A[p...i]的所有元素小于等于pivot，A[i+1...j-1]的所有元素大于pivot。

初始：i=p-1,j=p，因此A[p...p-1]=空，A[p...p-1]=空，因此成立。

保持：当循环开始前，已知A[p...i]的所有元素小于等于pivot，A[i+1...j-1]的所有元素大于pivot，在循环体中，

- 如果A[j]>pivot，那么不动，j++，此时A[p...i]的所有元素小于等于pivot，A[i+1...j-1]的所有元素大于pivot。

- 如果A[j]<=pivot，则i++，A[i+1]>pivot，将A[i+1]和A[j]交换后，A[P...i]保持所有元素小于等于pivot，而A[i+1...j-1]的所有元素大于pivot。

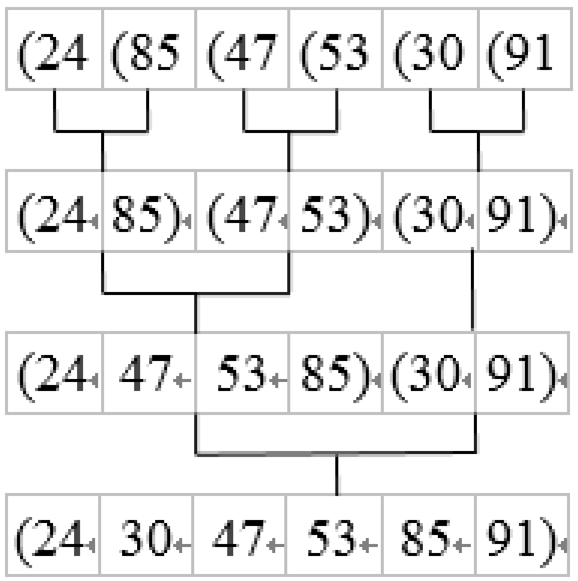
终止：j=r，因此A[p...i]的所有元素小于等于pivot，A[i+1...r-1]的所有元素大于pivot。

归并排序

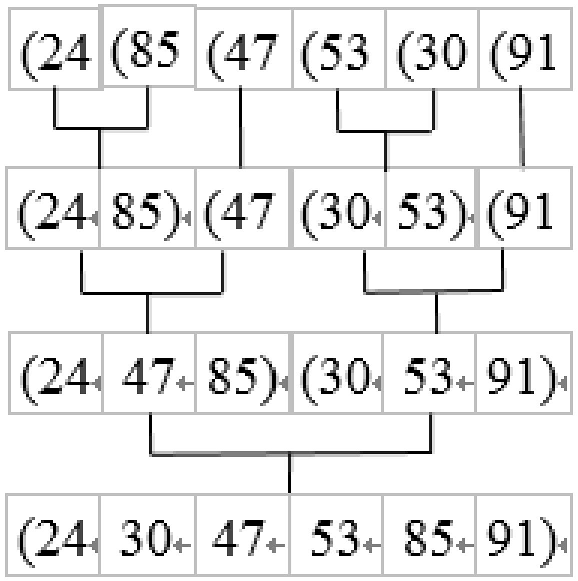
(1) 思路

两个或多个有序表合并成一个有序表。归并排序。

例：对{24, 85, 47, 53, 30, 91}归并排序。



自底向上归并排序



归并排序

归并排序

(2) 程序

归并排序（递归）：

```
void MergeSort ( T a[], int low, int high )
{
  if ( low>=high )  return;
  else {
    mid = (low+high)/2;
    MergeSort ( a, low, mid );
    MergeSort ( a, mid+1, high );
    Merge ( a, low, mid, high );
  }
}
```

自底向上的归并排序:

```
void MergeSort ( T a[], int n )
{
  t = 1;
  while ( t<n ) {
    s = t;  t = s*2;
    for ( i=0; i+t<=n; i+=t )
      Merge ( a, i, i+s-1, i+t-1 );
    if ( i+s<n )
      Merge ( a, i, i+s-1, n-1 );
  }
}
```

归并排序

```
Merge_sort(A)
{
    recursive_mergesort(A,1,length[A]);
}
recursive_mergesort(A,p,q)
{
    if p<q
        m = (p+q)/2
        recursive_mergesort(A,p,m)
        recursive_mergesort(A,m+1,q)
        merge(A,p,m,q)
}
merge(A,p,m,q)
{
    a = m-p+1
    b = q-m
    create array L[a+1] & R[b+1]
    for i=1 to a
        L[i] = A[p+i-1]
    for i=1 to b
        R[i] = A[m+i]
    L[a+1] = INFINITY
    R[b+1] = INFINITY
    i = j = 1
    for k = p to q
        if L[i]<R[j]
            A[k] = L[i]
            i++
        else if L[i]>R[j]
            A[k] = R[j]
            j++
}
```

归并排序

merge()函数的正确性证明中：

merge函数的主要步骤在第25~31行，可以看出是由一个循环构成。

循环不变式：每次循环之前，A[p...k-1]已排序，且L[i]和R[j]是L和R中剩下的元素中最小的两个元素。

初始：k=p，A[p...p-1]为空，因此已排序，成立。

保持：在第k次迭代之前，A[p...k-1]已经排序，而因为L[i]和R[j]是L和R中剩下的元素中最小的两个元素，因此只需要将L[i]和R[j]中最小的元素放到A[k]即可，在第k+1次迭代之前A[p...k]已排序，且L[i]和R[j]为剩下的最小的两个元素。

终止：k=q+1，且A[p...q]已排序

归并排序

Merge()，将有序序列a[low..mid]和a[mid+1..high]归并到a[low..high]。

```
void Merge ( T a[], int low, int mid, int high )
{
  // 归并到b[]
  i = low;  j = mid+1;  k = low;
  while ( i<=mid and j<=high ) {
    if ( a[i]<=a[j] ) { b[k] = a[i];  i++; }
    else  { b[k] = a[j];  j++; }
    k++;
  }
  // 归并剩余元素
  while ( i<=mid )  b[k++] = a[i++];
  while ( j<=high )  b[k++] = a[j++];
  // 从b[]复制回a[]
  a[low..high] = b[low..high];
}
```

时间复杂度O(nlogn)。需要空间多，空间复杂度O(n)。归并排序是稳定的排序。

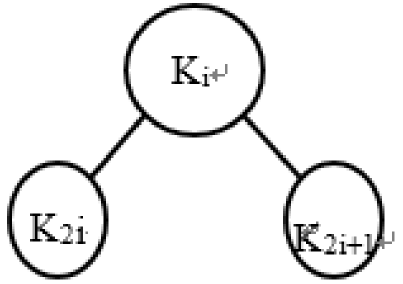
堆排序

(1) 堆及其特点

堆，小顶堆，大顶堆。

序列$\{K_1,K_2, ..., K_n\}$满足$K_i \le K_{2i}$，$Ki \le K_{2i+1}$，称为小顶堆；若满足$K_i \ge K_{2i}$，$K_i \ge K_{2i+1}$，称为大顶堆，其中$i=1, 2, ..., n/2$。

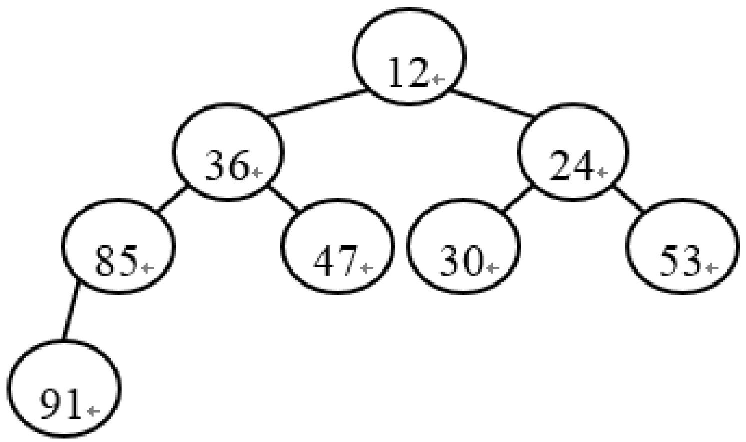
特点：小顶堆的堆顶(第一个元素)为最小元素，大顶堆的堆顶为最大元素



(2) 判断序列是否构成堆

方法：用Ki作为编号为i的结点，画一棵完全二叉树，比较双亲和孩子容易判断是否构成堆。

例：判断序列(12,36,24,85,47,30,53,91)是否构成堆。



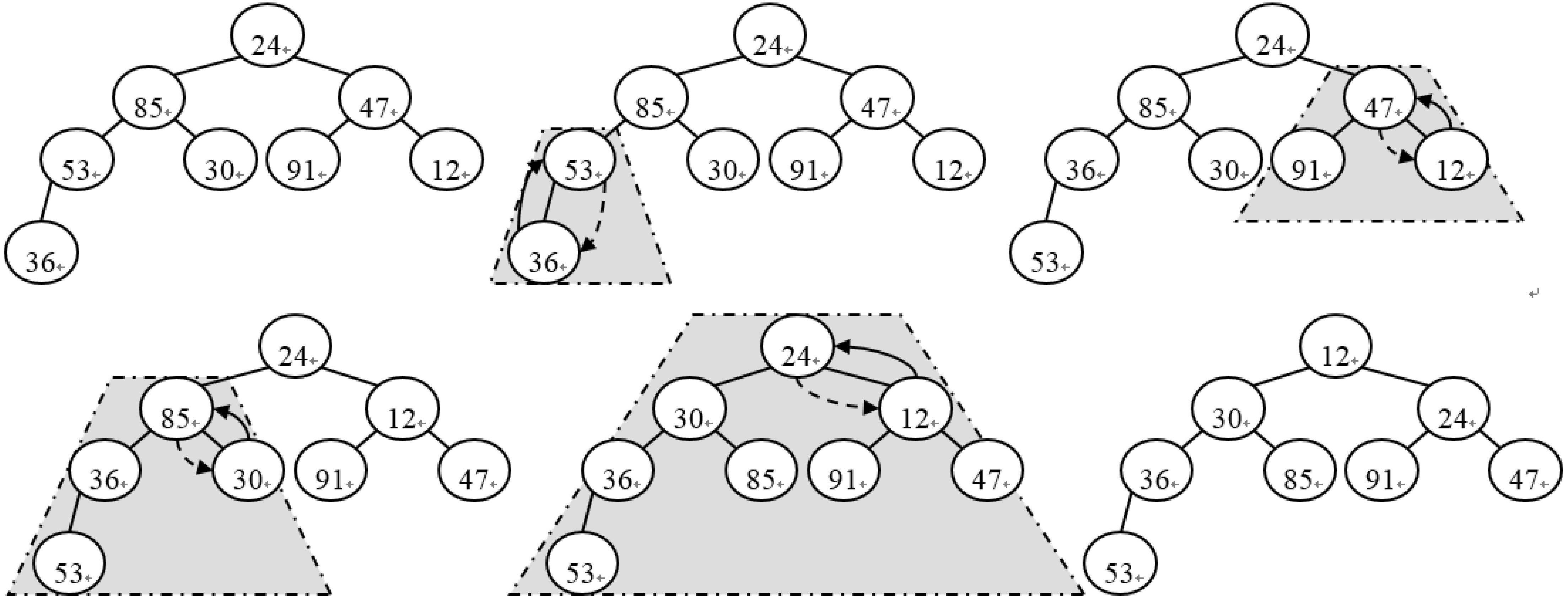
根据上图判断，该序列构成小顶堆。

堆排序

(3) 建立堆

“‘小堆’变‘大堆’，从[n/2] 变到1” 。第[n/2] 个是最后一个分支结点。

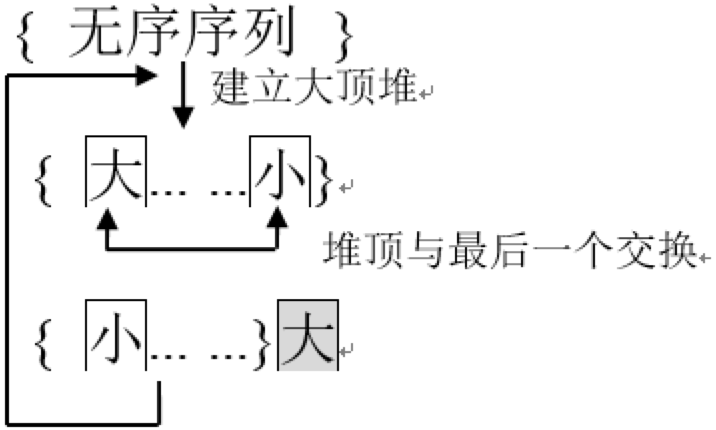
例：把(24,85,47,53,30,91,12,36)调整成小顶堆。

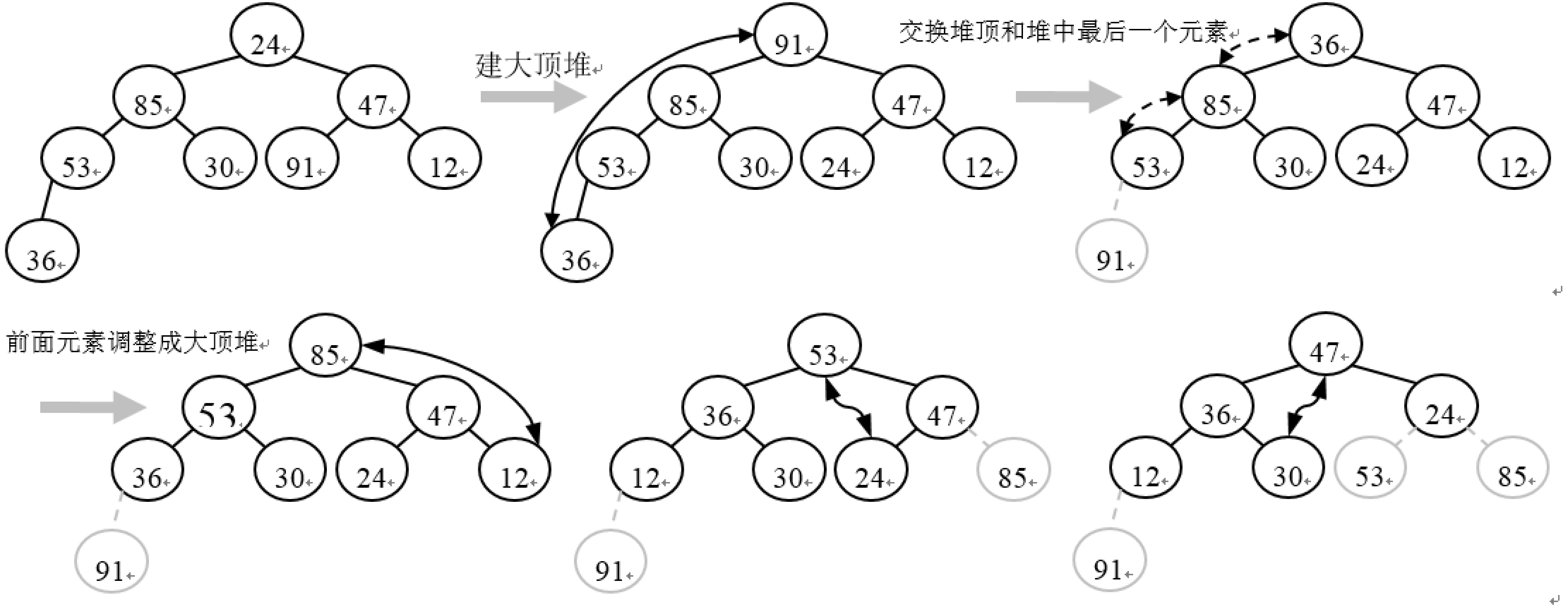


堆排序

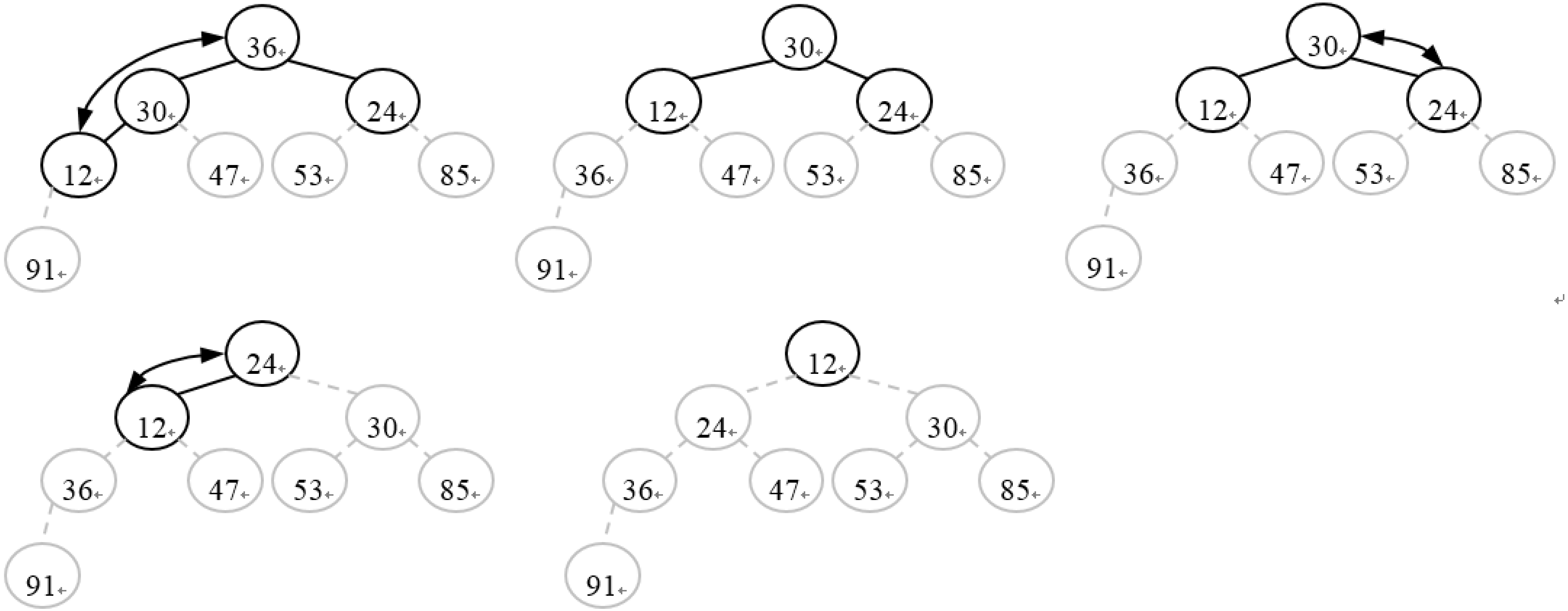
(4) 堆排序

思路:





堆排序



36
30
24
12
47
53
85
91
30
12
24
36
47
53
85
91
30
12
24
36
47
53
85
91
24
12
30
36
47
53
85
91
12
24
30
36
47
53
85
91

堆排序

整个排序步骤如下：

24	85	47	53	30	91	12	36

91	85	47	53	30	24	12	36

85	53	47	36	30	24	12	91

53	36	47	12	30	24	85	91

47	36	24	12	30	53	85	91

36	30	24	12	47	53	85	91

30	12	24	36	47	53	85	91

24	12	30	36	47	53	85	91

12	24	30	36	47	53	85	91

思想：运用了最小堆、最大堆这个数据结构，而堆还能用于构建优先队列。

优先队列应用于进程间调度、任务调度等。

最优时间：O(nlgn)
最差时间：O(nlgn)

堆排序

```
MAX_HEAPIFY(A,i)
{
    heapsize = heapsize[A];
    largest = i;
    if left(i)<= heapsize && A[largest]<A[left(i)]
        largest = left(i);
    else if right(i)<= heapsize && A[largest]<A[right(i)]
        largest = right(i);
    if(largest!=i)
        swap A[i]<->A[largest];
        MAX_HEAPIFY(A,largest);
}
build_max_heap(A)
{
    for i=floor(n/2) to 1
        MAX_HEAPIFY(A,i);
}
heapsort(A)
{
    build_max_heap(A);
    for i=n to 2
        swap A[1]<->A[heapsize]
        heapsize--;
        MAX_HEAPIFY(A,1);
}
```

堆排序

build_max_heap的正确性证明中：

循环不变式：每次循环开始前，A[i+1]、A[i+2]、…、A[n]分别为最大堆的根。

初始：i=floor(n/2)，则A[i+1]、…、A[n]都是叶子，因此成立。

保持：每次迭代开始前，已知A[i+1]、A[i+2]、…、A[n]分别为最大堆的根，在循环体中，因为A[i]的孩子的子树都是最大堆，因此执行完MAX_HEAPIFY(A,i)后，A[i]也是最大堆的根，因此保持循环不变式。

终止：i=0，已知A[1]、…、A[n]都是最大堆的根，得到了A[1]是最大堆的根

堆排序

heapsort的正确性证明中：

循环不变式：每次迭代前，A[i+1]、...、A[n]包含了A中最大的n-i个元素，且A[i+1]<=A[i+2]<=...<=A[n]，且A[1]是堆中最大的。

初始：i=n，A[n+1]...A[n]为空，成立。

保持：每次迭代开始前，A[i+1]、...、A[n]包含了A中最大的n-i个元素，且A[i+1]<=A[i+2]<=...<=A[n]，循环体内将A[1]与A[i]交换，因为A[1]是堆中最大的，因此A[i]、...、A[n]包含了A中最大的n-i+1个元素且A[i]<=A[i+1]<=A[i+2]<=...<=A[n]，因此保持循环不变式。

终止：i=1，已知A[2]、...、A[n]包含了A中最大的n-1个元素，且A[2]<=A[3]<=...<=A[n]，因此A[1]<=A[2]<=A[3]<=...<=A[n]

程序规范

对数组b[m：n]进行排序的程序。功能是把数组b[m：n]各元素的值从小到大排列起来，使得最后的数组满足b[i] ≤b[i+1]，i=m，…,n-1。

- **规范：**
  - P： $\{m \le n \land b[m:n]=u[m:n]\}$
  - Q： $\{m \le n \land perm(b[m:n],u[m:n]) \land$

$$(i: m \le i<n : b[i] \le b[i+1])\}$$

其中， u[m:n]代表b的任意可能初值； perm(b[m:n],u[m:n]) 表示b是u的一个置换。