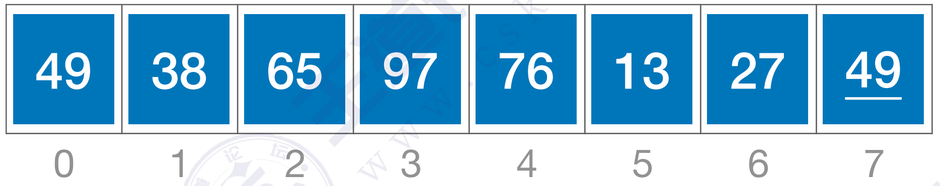
本节内容

插入排序

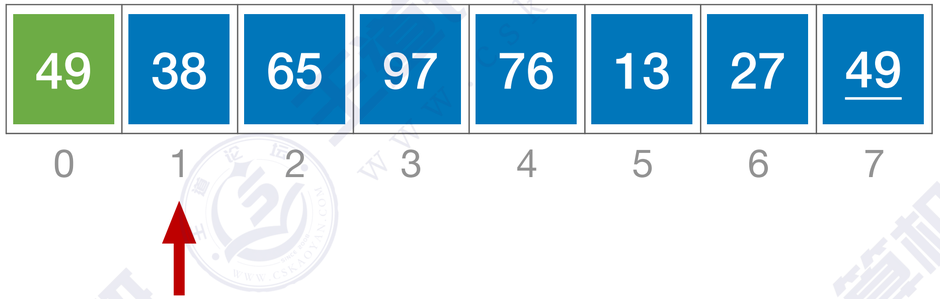
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



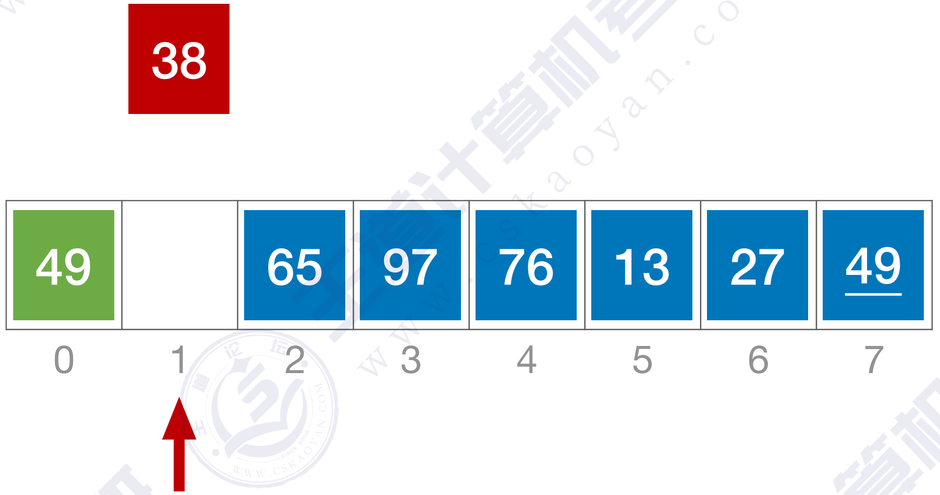
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



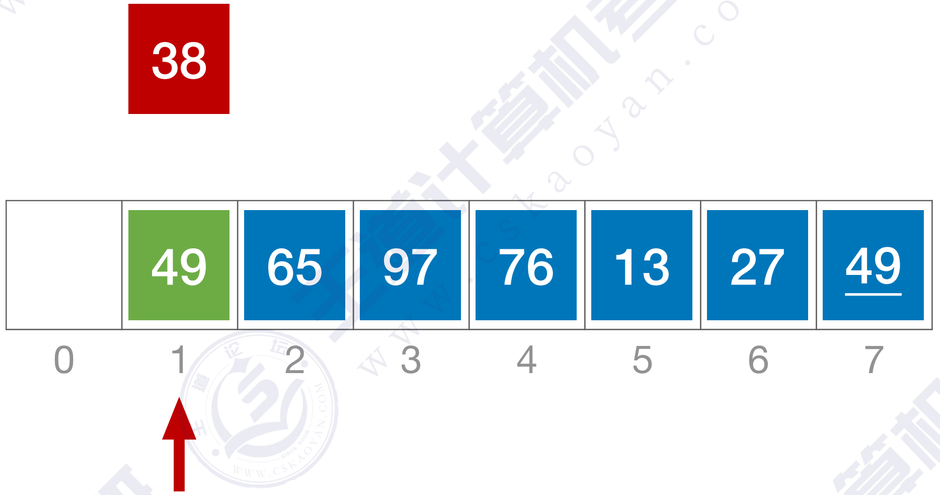
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



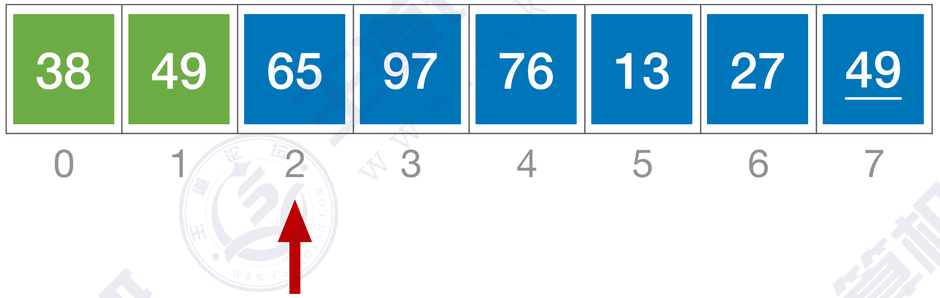
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



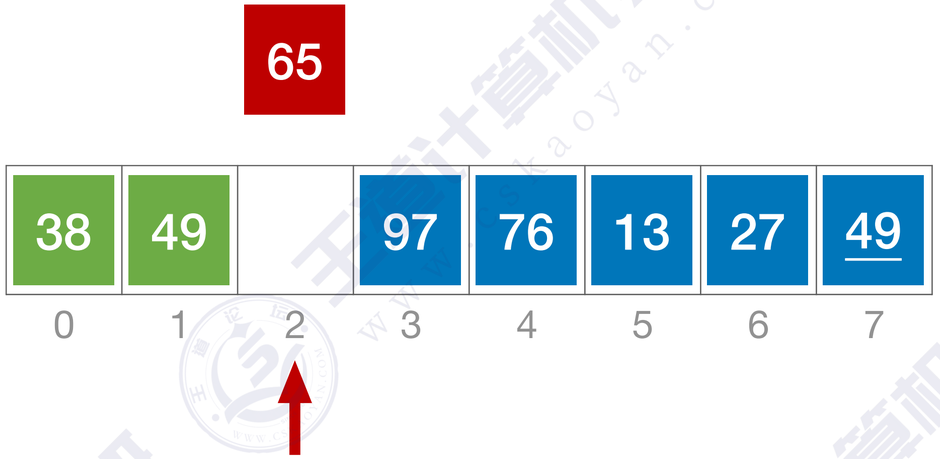
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



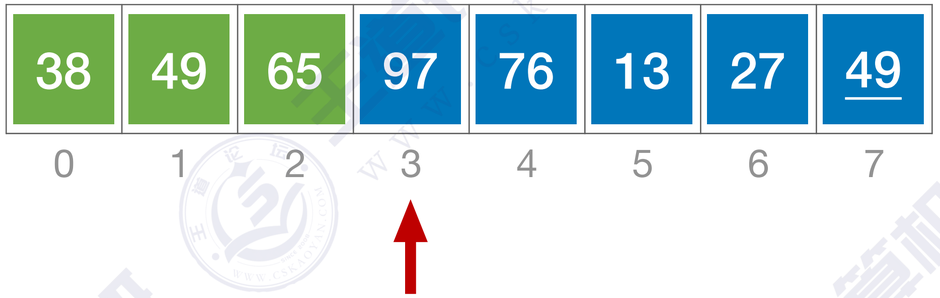
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



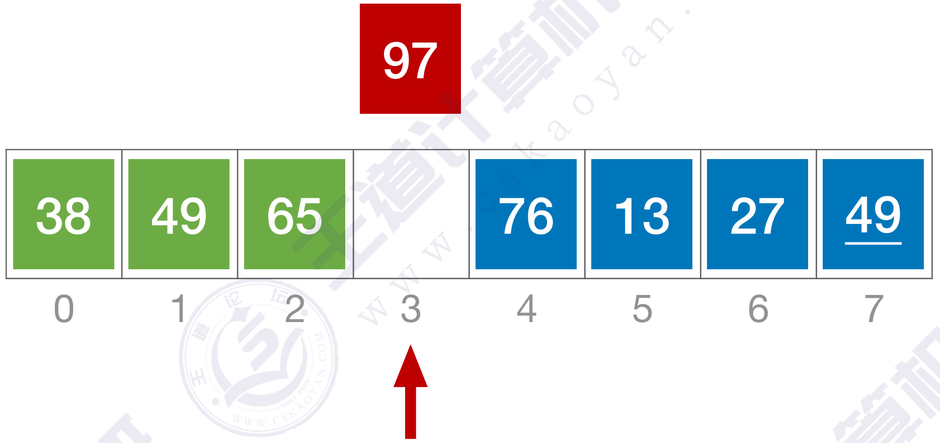
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



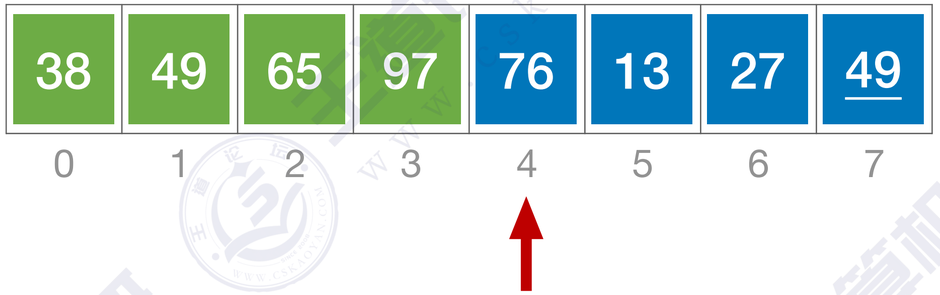
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



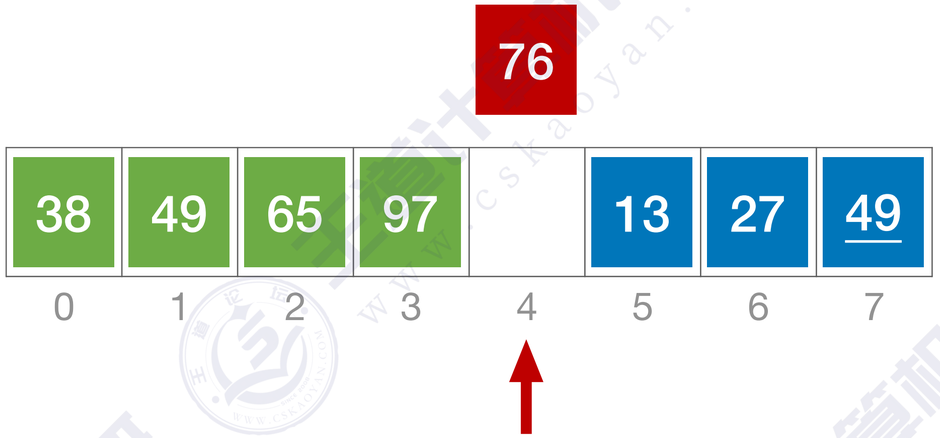
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



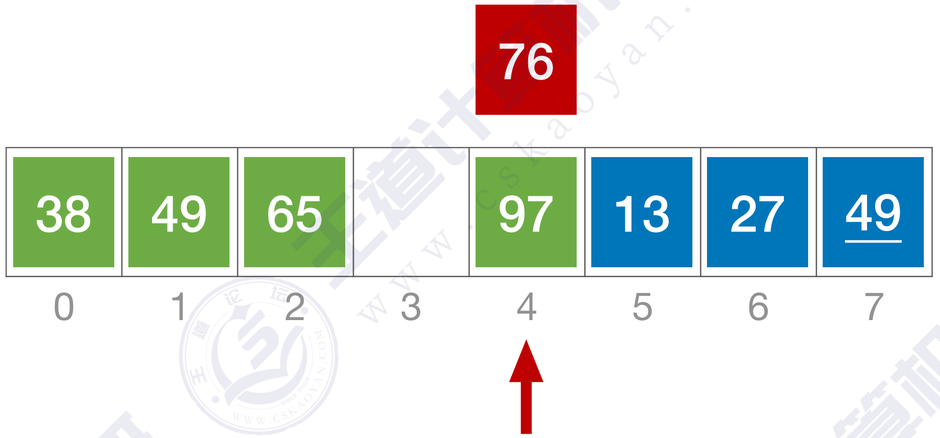
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



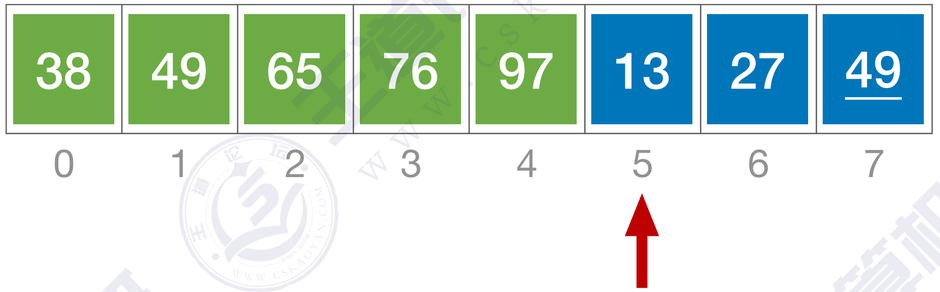
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



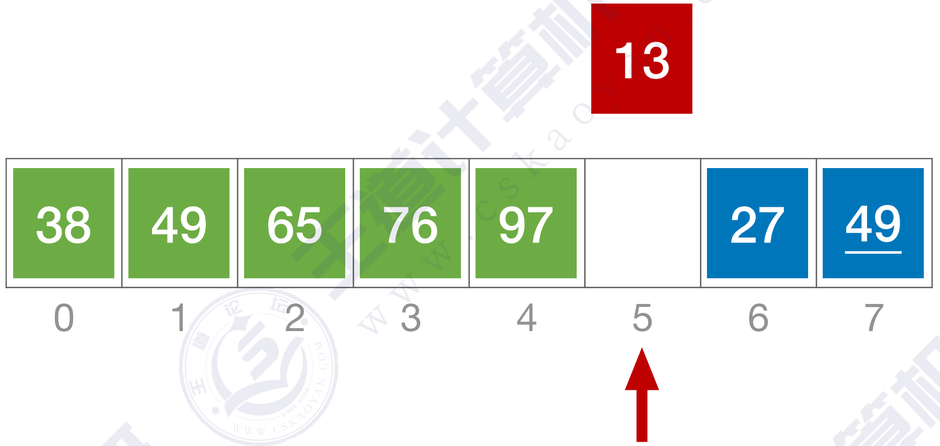
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



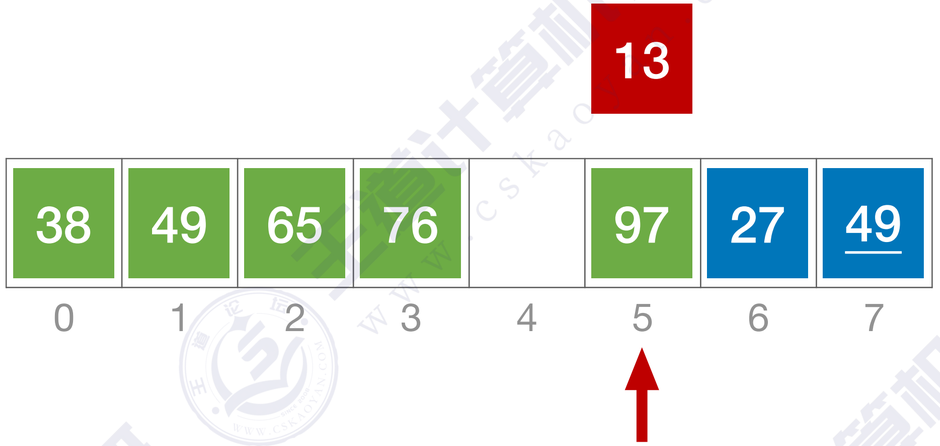
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



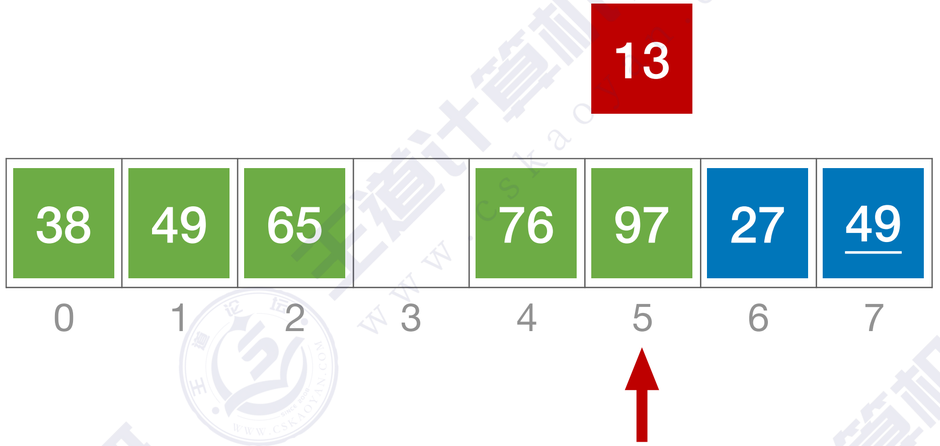
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



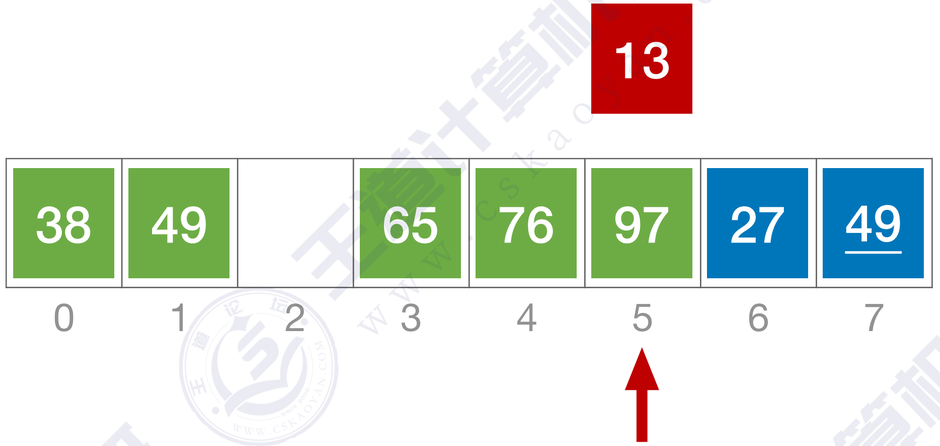
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



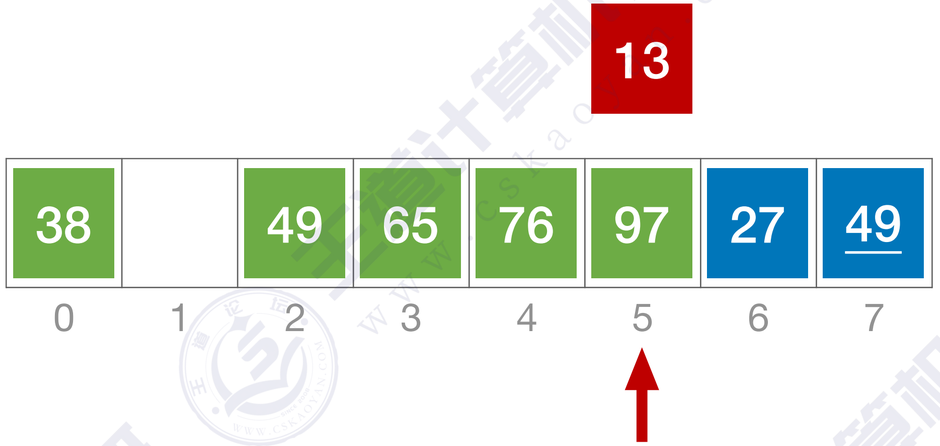
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



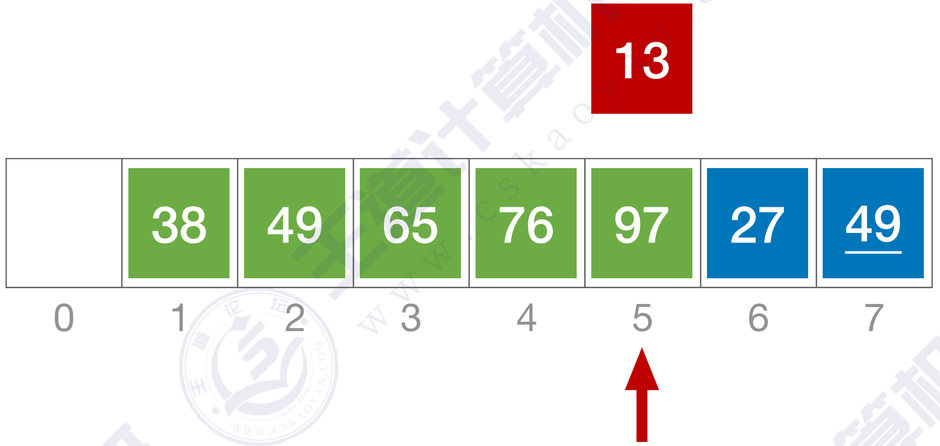
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



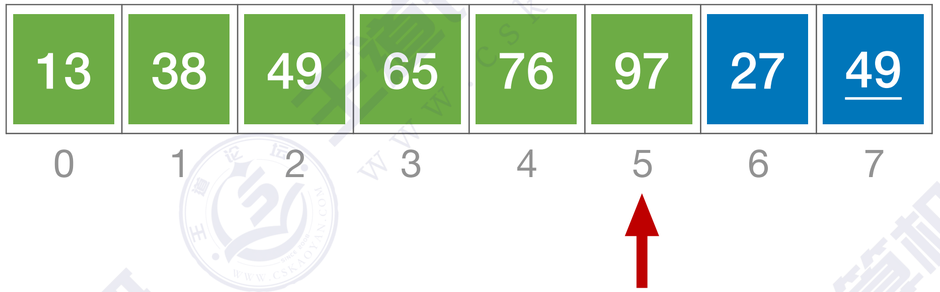
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



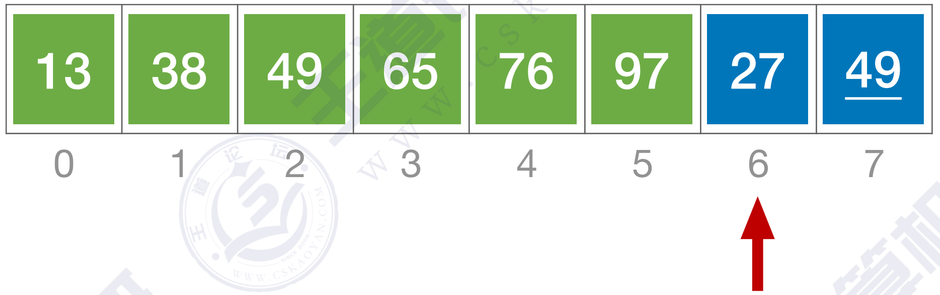
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



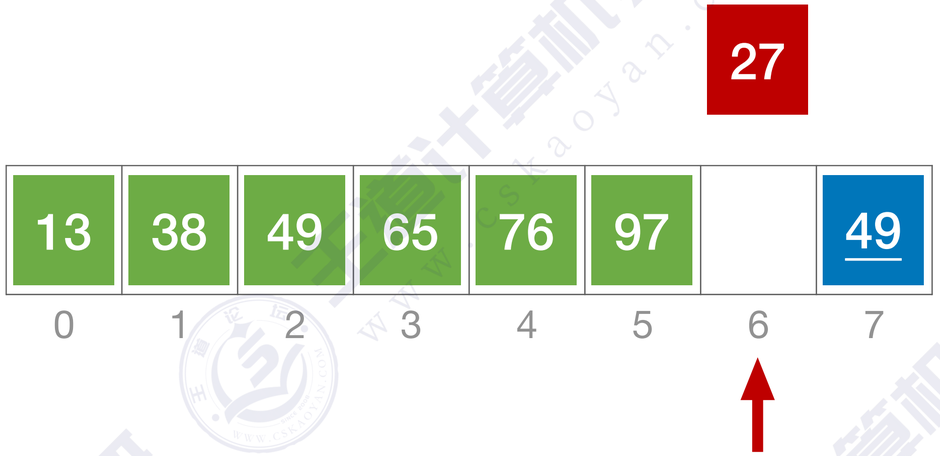
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



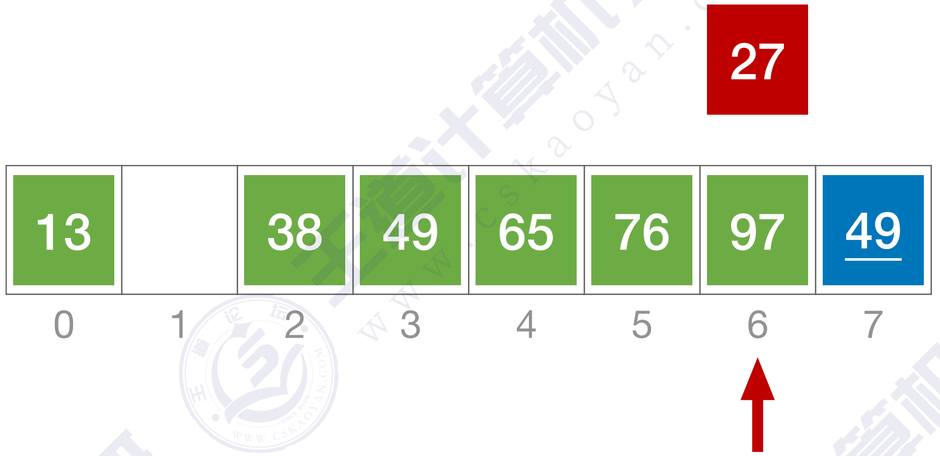
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



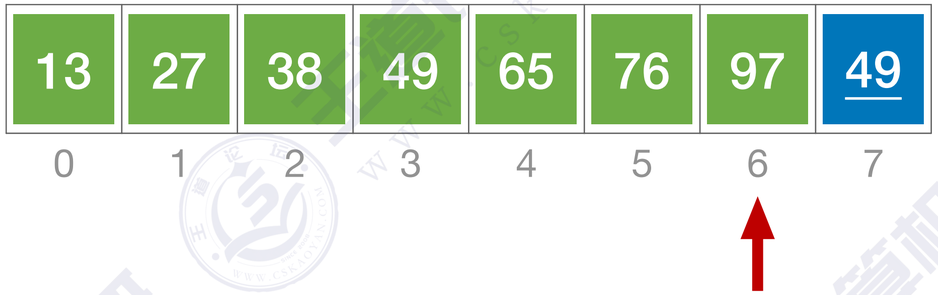
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



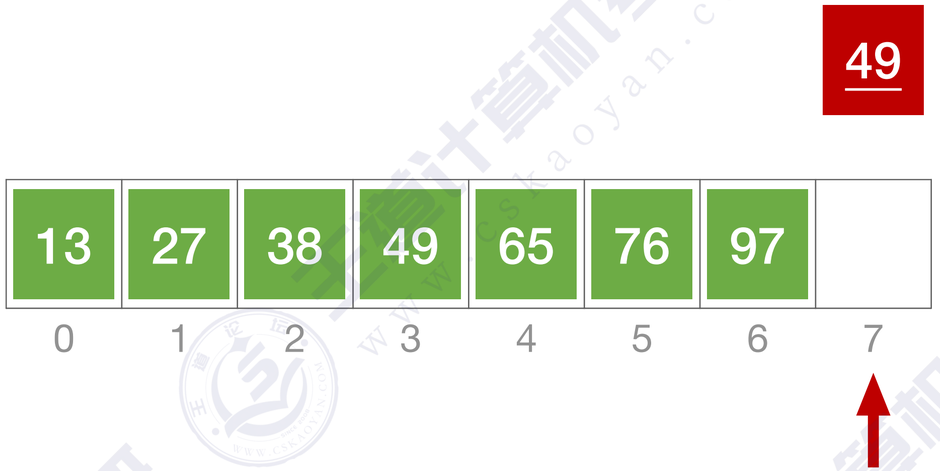
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



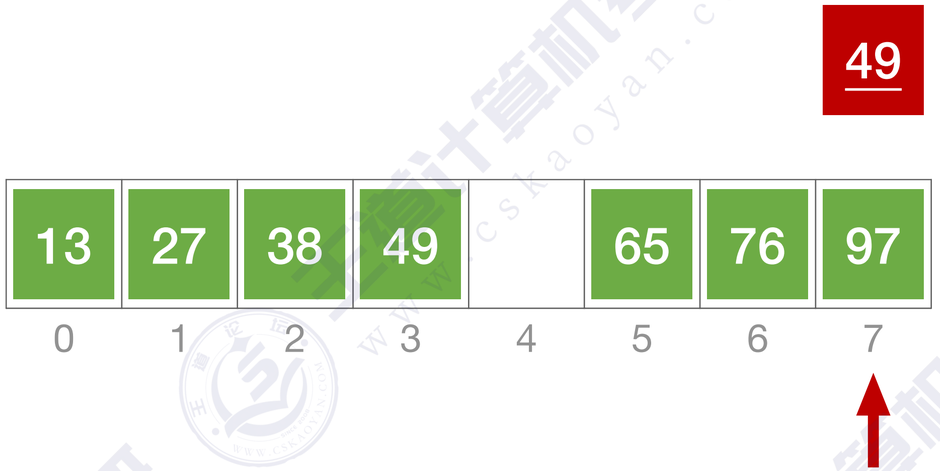
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



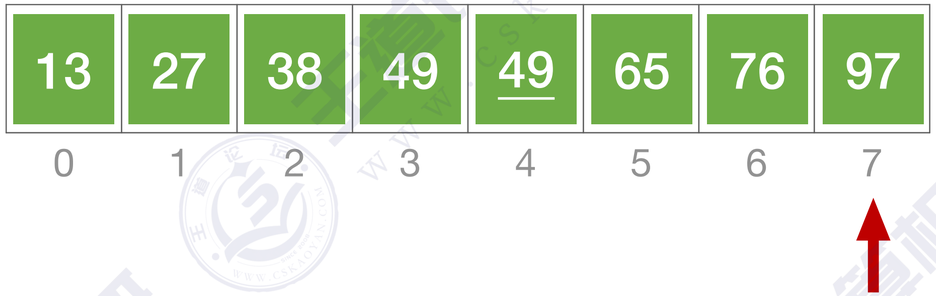
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



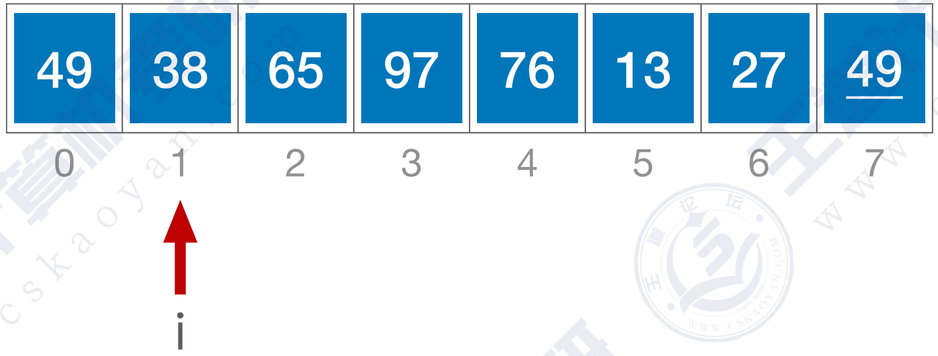
插入排序

算法思想：每次将一个待排序的记录按其关键字大小插入到前面已排好序的子序列中，直到全部记录插入完成。



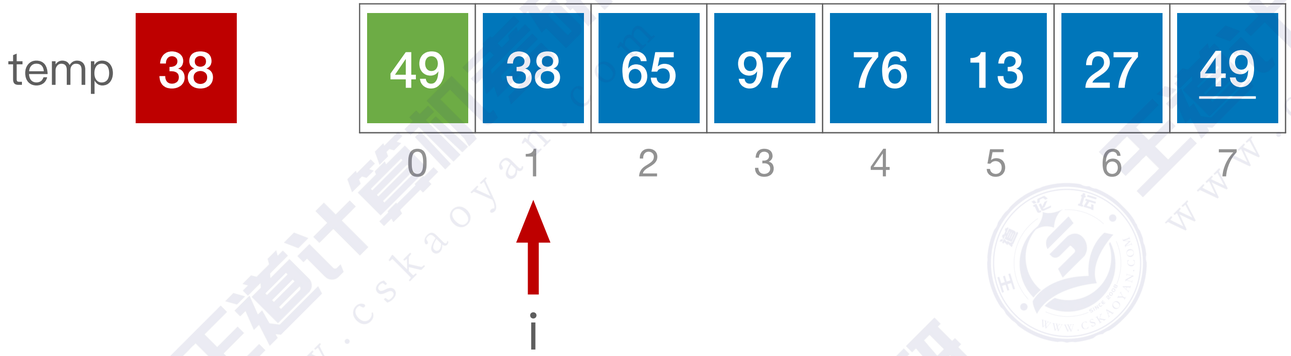
算法实现

```
//直接插入排序
void InsertSort(int A[],int n){
    int i,j,temp;
    for(i=1;i<n;i++)                    //将各元素插入已排好序的序列中
        if(A[i]<A[i-1]){                //若A[i]关键字小于前驱
            temp=A[i];                  //用temp暂存A[i]
            for(j=i-1;j>=0 && A[j]>temp;--j)    //检查所有前面已排好序的元素
                A[j+1]=A[j];            //所有大于temp的元素都向后挪位
            A[j+1]=temp;                //复制到插入位置
        }
}
```



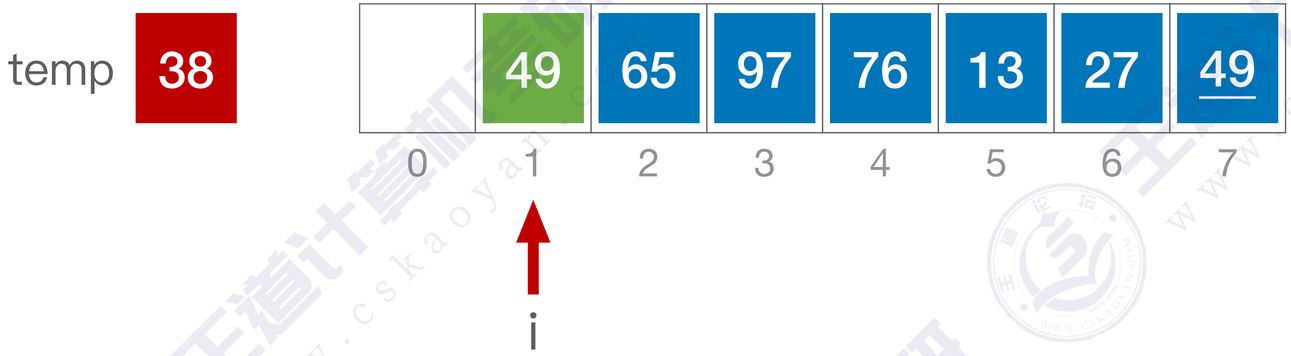
算法实现

```
//直接插入排序
void InsertSort(int A[],int n){
    int i,j,temp;
    for(i=1;i<n;i++)                    //将各元素插入已排好序的序列中
        if(A[i]<A[i-1]){                //若A[i]关键字小于前驱
            temp=A[i];                  //用temp暂存A[i]
            for(j=i-1;j>=0 && A[j]>temp;--j)    //检查所有前面已排好序的元素
                A[j+1]=A[j];            //所有大于temp的元素都向后挪位
            A[j+1]=temp;                //复制到插入位置
        }
}
```



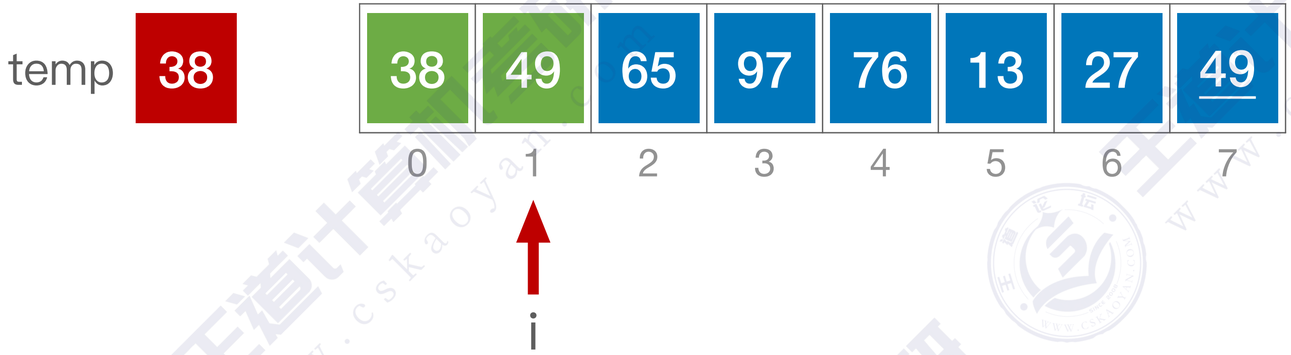
算法实现

```
//直接插入排序
void InsertSort(int A[],int n){
    int i,j,temp;
    for(i=1;i<n;i++)                    //将各元素插入已排好序的序列中
        if(A[i]<A[i-1]){                //若A[i]关键字小于前驱
            temp=A[i];                  //用temp暂存A[i]
            for(j=i-1;j>=0 && A[j]>temp;--j)    //检查所有前面已排好序的元素
                A[j+1]=A[j];            //所有大于temp的元素都向后挪位
            A[j+1]=temp;                //复制到插入位置
        }
}
```



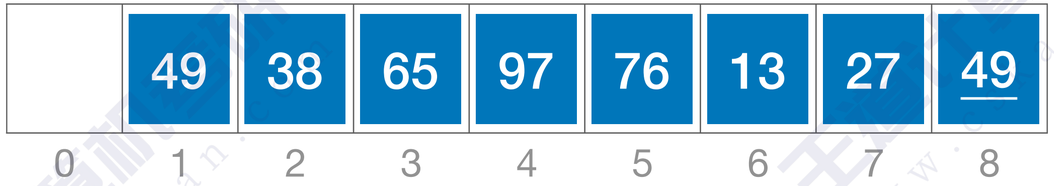
算法实现

```
//直接插入排序
void InsertSort(int A[],int n){
    int i,j,temp;
    for(i=1;i<n;i++)                    //将各元素插入已排好序的序列中
        if(A[i]<A[i-1]){                //若A[i]关键字小于前驱
            temp=A[i];                  //用temp暂存A[i]
            for(j=i-1;j>=0 && A[j]>temp;--j)    //检查所有前面已排好序的元素
                A[j+1]=A[j];            //所有大于temp的元素都向后挪位
            A[j+1]=temp;                //复制到插入位置
        }
}
```



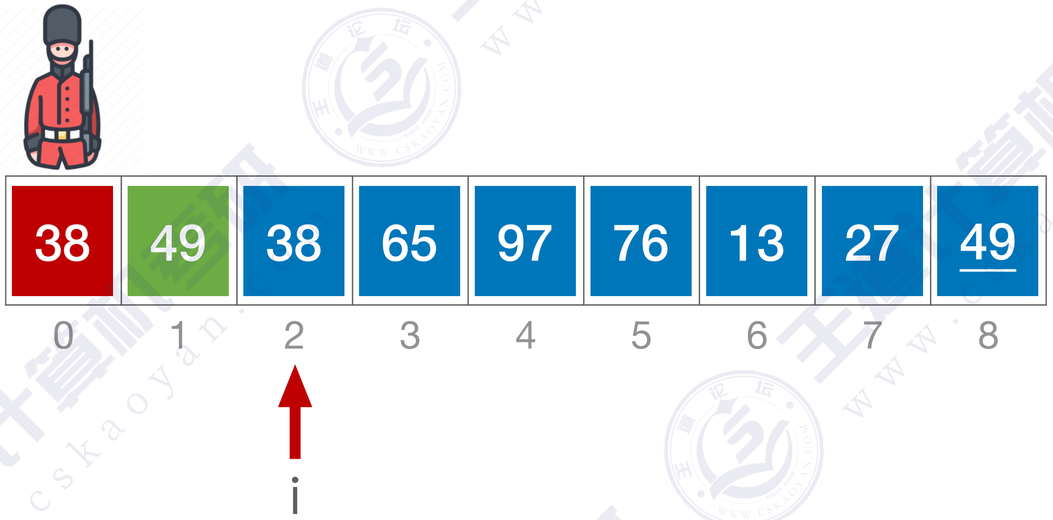
算法实现（带哨兵）

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){             //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];               //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];         //向后挪位
            A[j+1]=A[0];             //复制到插入位置
        }
}
```



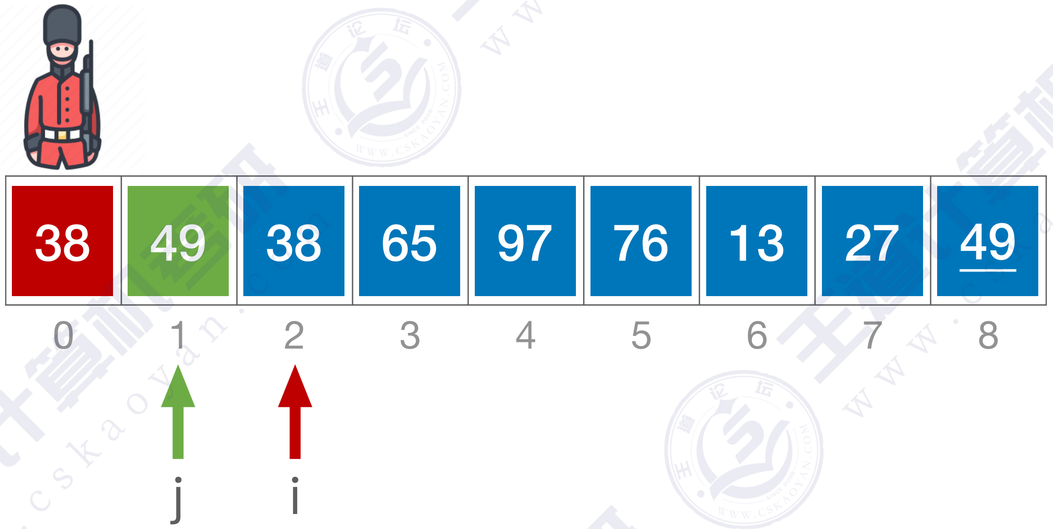
算法实现（带哨兵）

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){             //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];               //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];         //向后挪位
            A[j+1]=A[0];             //复制到插入位置
        }
}
```



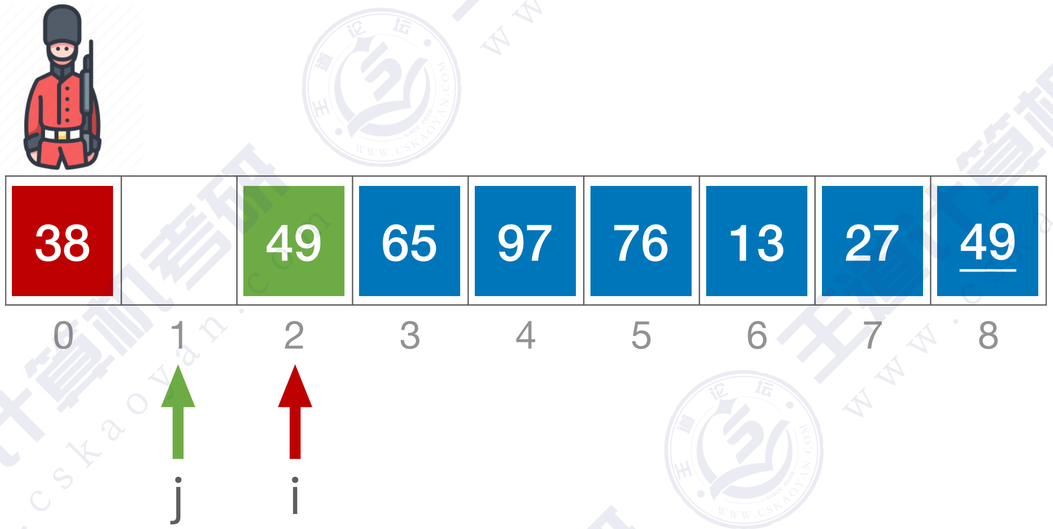
算法实现（带哨兵）

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                  //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){               //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];                 //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];           //向后挪位
            A[j+1]=A[0];               //复制到插入位置
        }
}
```



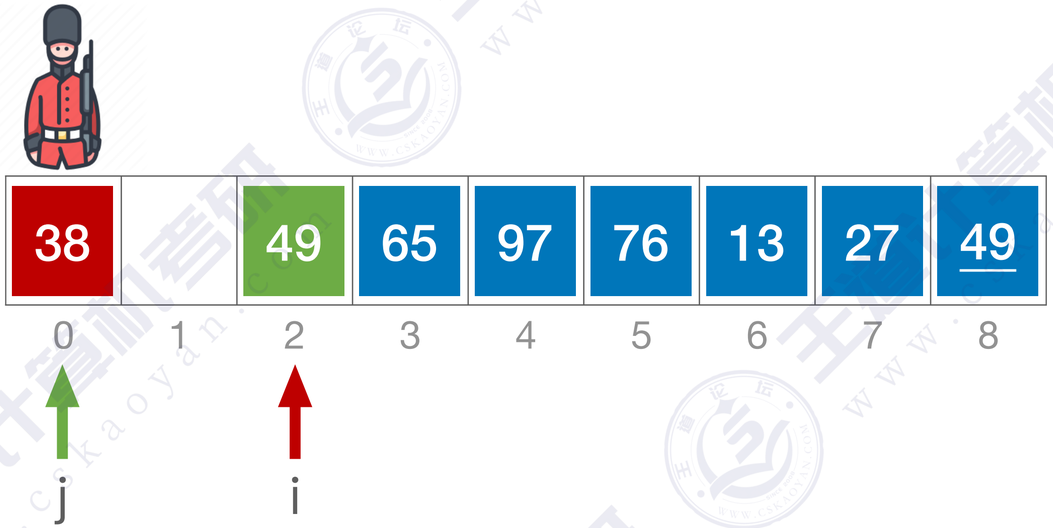
算法实现（带哨兵）

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                  //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){               //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];                 //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];           //向后挪位
            A[j+1]=A[0];               //复制到插入位置
        }
}
```



算法实现（带哨兵）

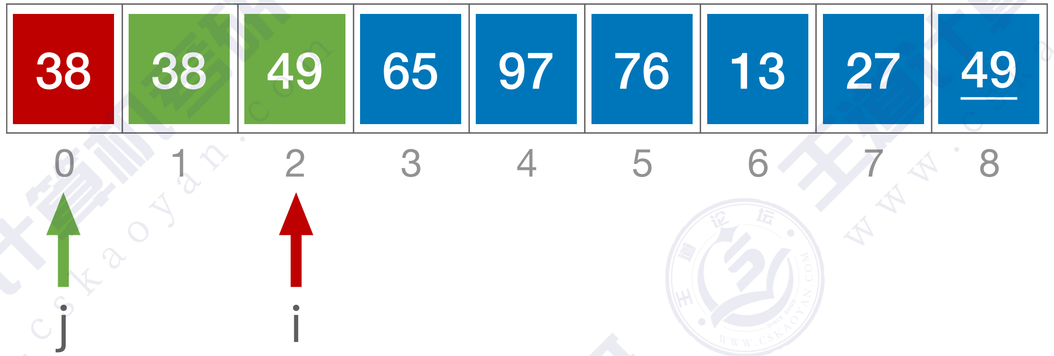
```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){             //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];               //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];         //向后挪位
            A[j+1]=A[0];             //复制到插入位置
        }
}
```



算法实现（带哨兵）

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){             //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];               //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];         //向后挪位
            A[j+1]=A[0];             //复制到插入位置
        }
}
```

优点：不用每轮循环都判断 j>=0



算法效率分析

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)              //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){           //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];             //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];       //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

空间复杂度：O(1)

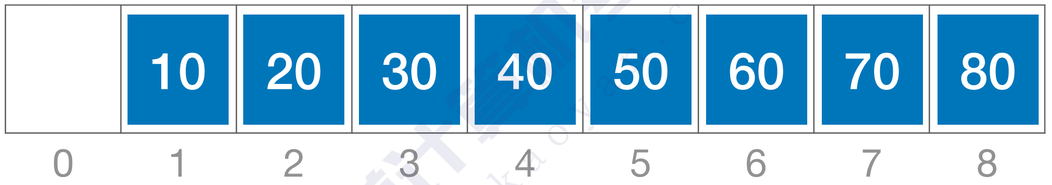
时间复杂度：主要来自对比关键字、移动元素

若有 n 个元素，则需要 n-1 趟处理

算法效率分析

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)              //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){           //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];             //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];       //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

最好情况：原本就有序



最好情况：

共n-1趟处理，每一趟只需要对比关键字1次，不用移动元素

最好时间复杂度—— O(n)

算法效率分析

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)              //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){           //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];             //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];       //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

最坏情况：原本为逆序

最坏情况：

第1趟：对比关键字2次，移动元素3次

第2趟：对比关键字3次，移动元素4次

…

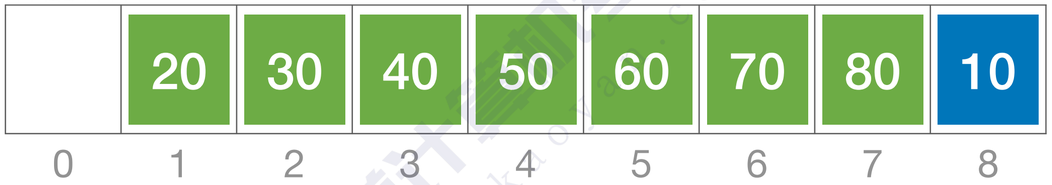
第 i 趟：对比关键字 i+1次，移动元素 i+2 次

…

算法效率分析

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)              //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){           //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];             //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];       //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

最坏情况：原本为逆序



最坏情况：

第1趟：对比关键字2次，移动元素3次

第2趟：对比关键字3次，移动元素4次

第 i 趟：对比关键字 i+1次，移动元素 i+2 次

…

第n−1趟：对比关键字 n 次，移动元素 n+1 次

最坏时间复杂度——$O(n^2)$

算法效率分析

```
//直接插入排序（带哨兵）
void InsertSort(int A[],int n){
    int i,j;
    for(i=2;i<=n;i++)              //依次将A[2]~A[n]插入到前面已排序序列
        if(A[i]<A[i-1]){           //若A[i]关键码小于其前驱，将A[i]插入有序表
            A[0]=A[i];             //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];       //向后挪位
            A[j+1]=A[0];           //复制到插入位置
        }
}
```

空间复杂度：O(1)

最好时间复杂度（全部有序）：O(n)

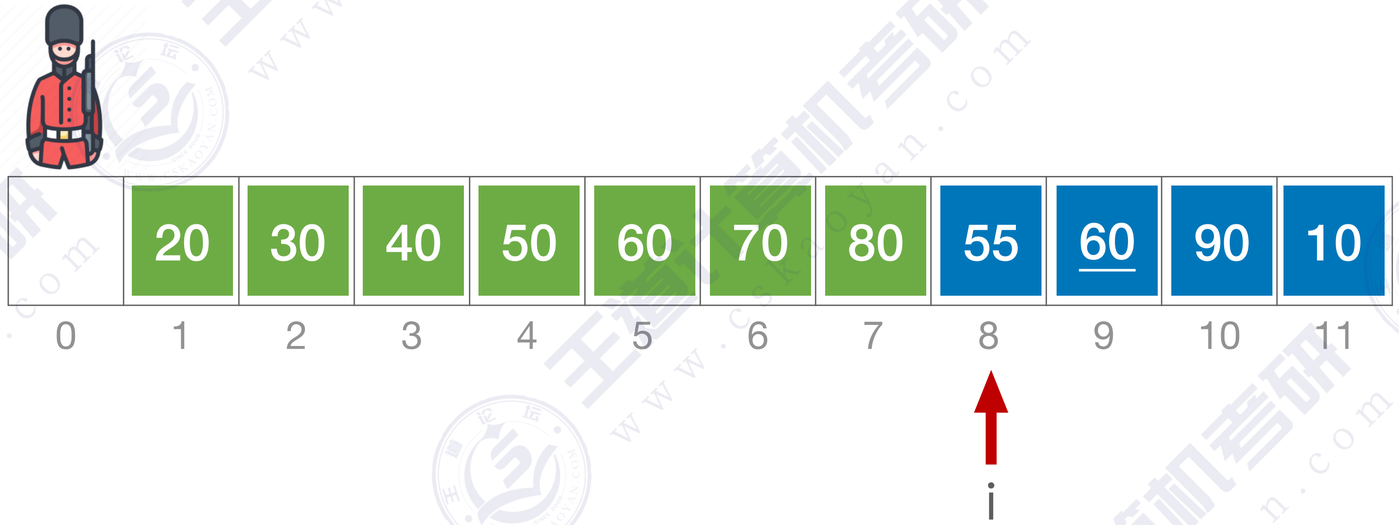
最坏时间复杂度（全部逆序）：$O(n^2)$

平均时间复杂度：$O(n^2)$

算法稳定性：稳定

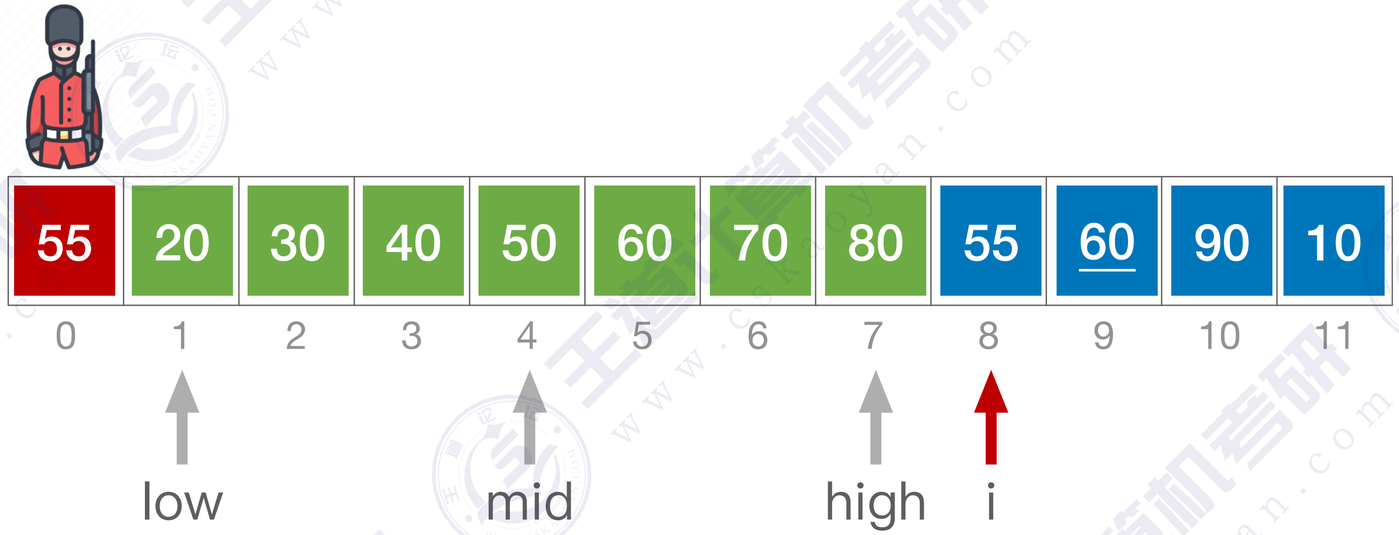
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



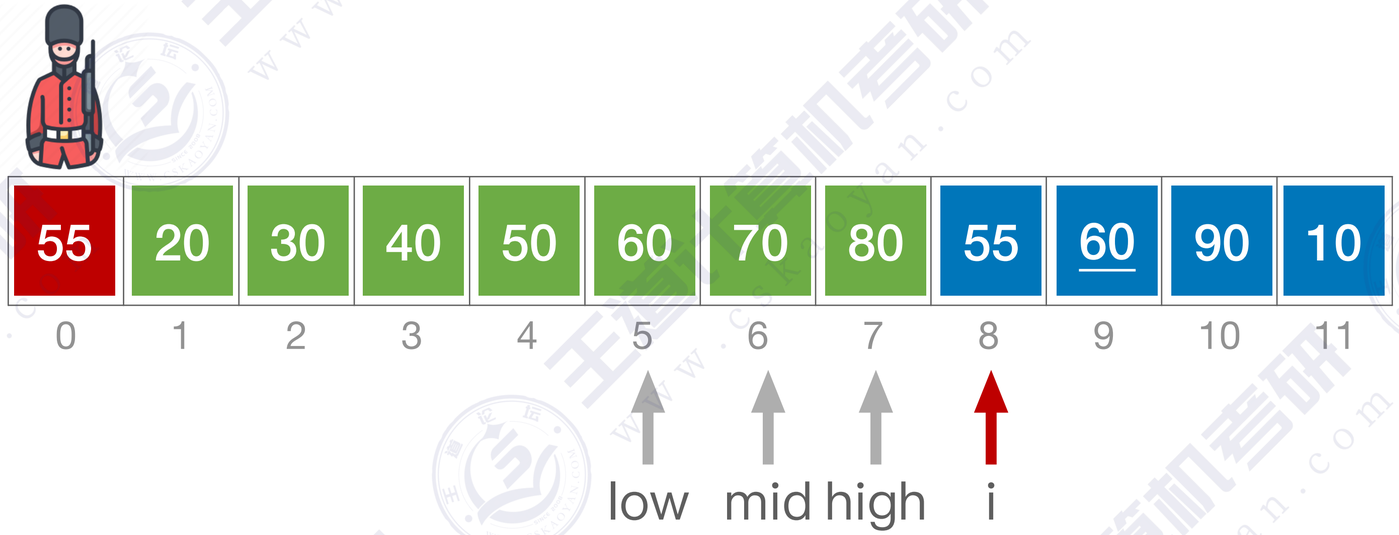
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



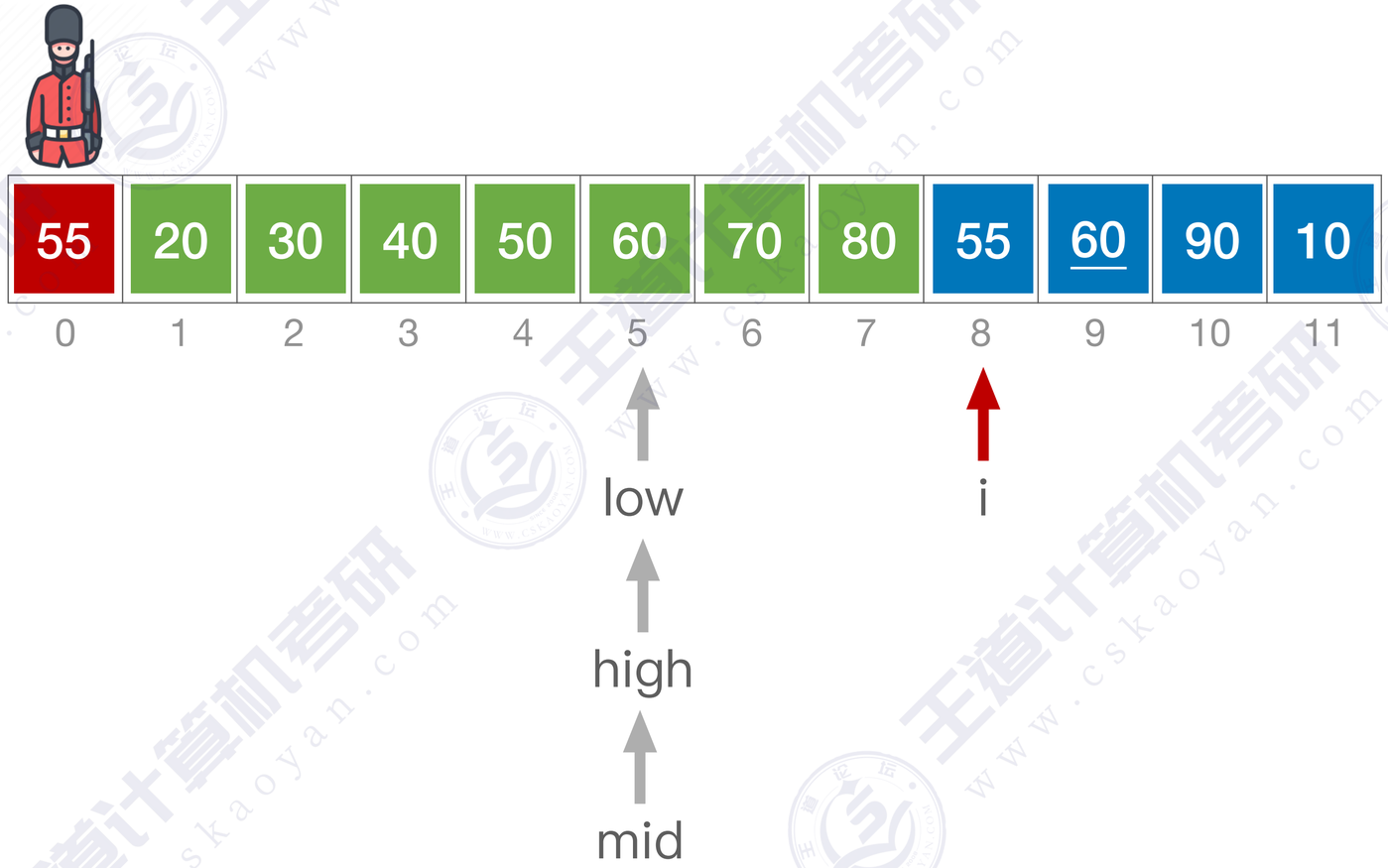
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



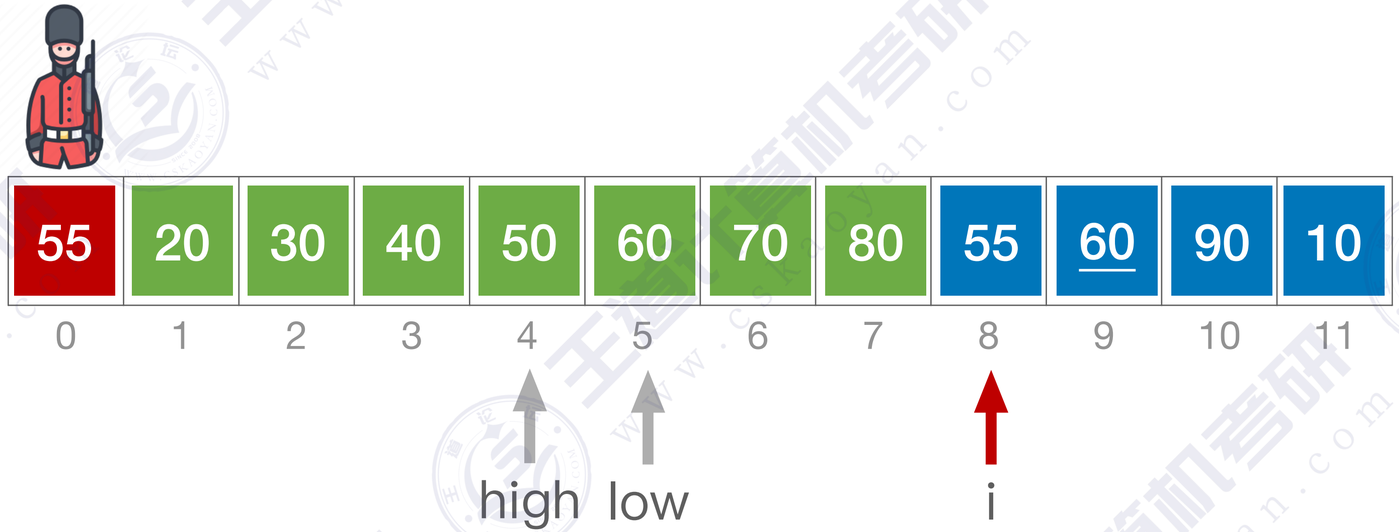
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



优化——折半插入排序

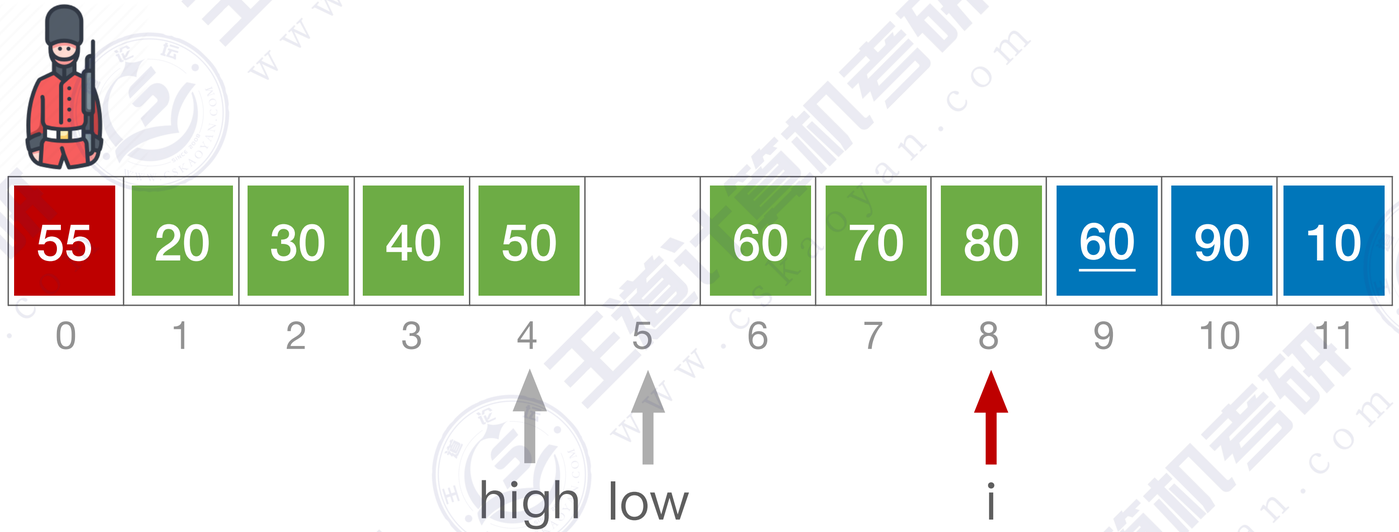
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

优化——折半插入排序

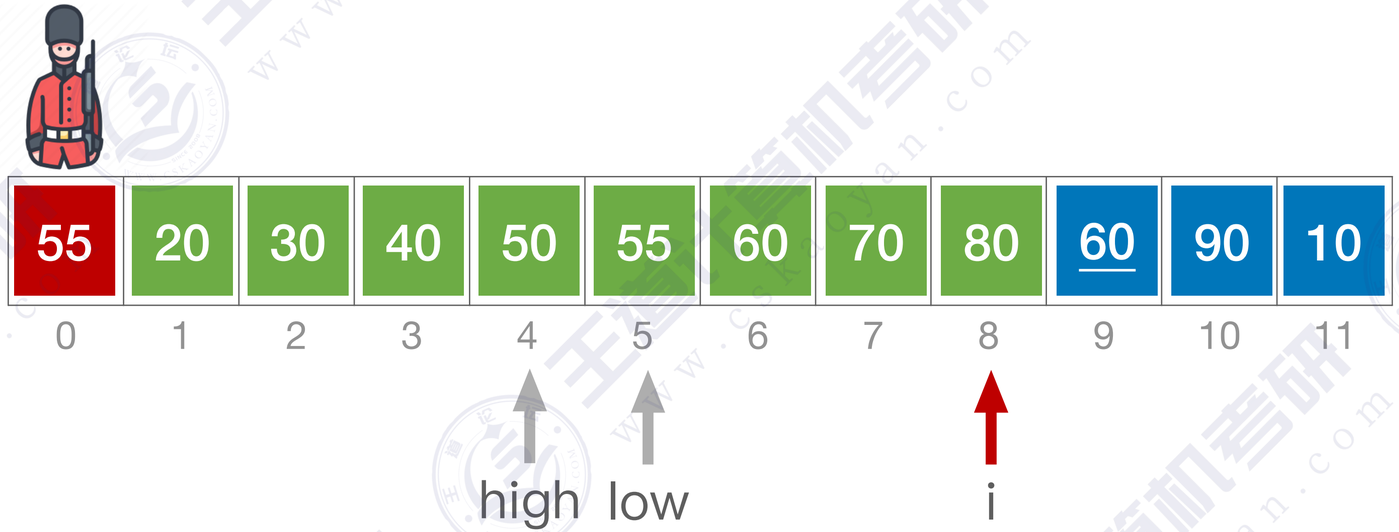
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

优化——折半插入排序

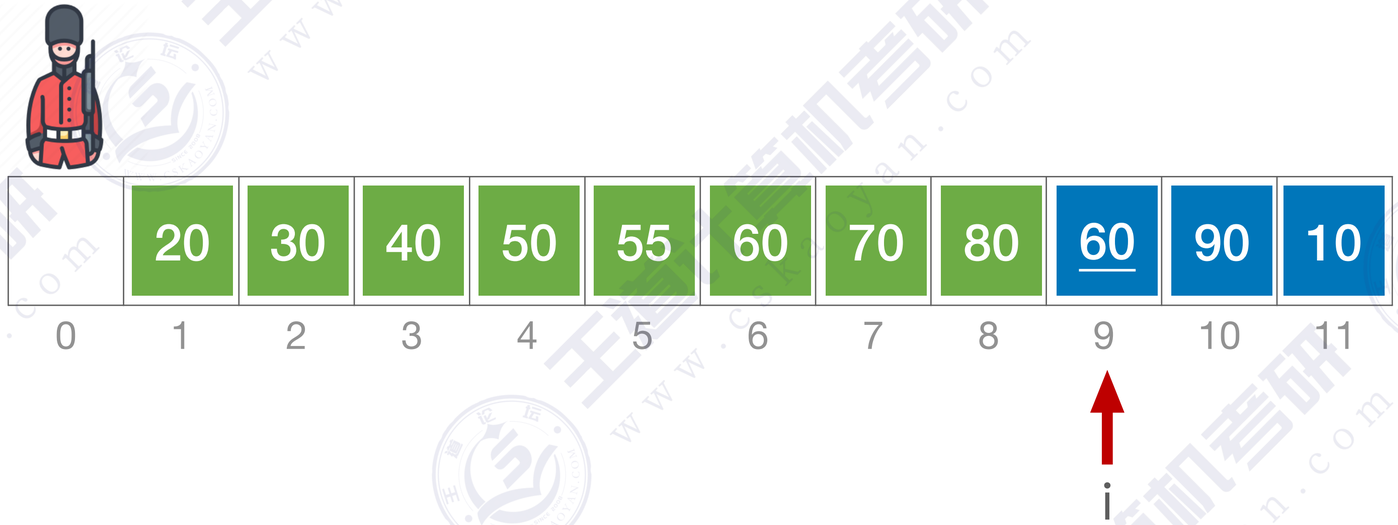
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

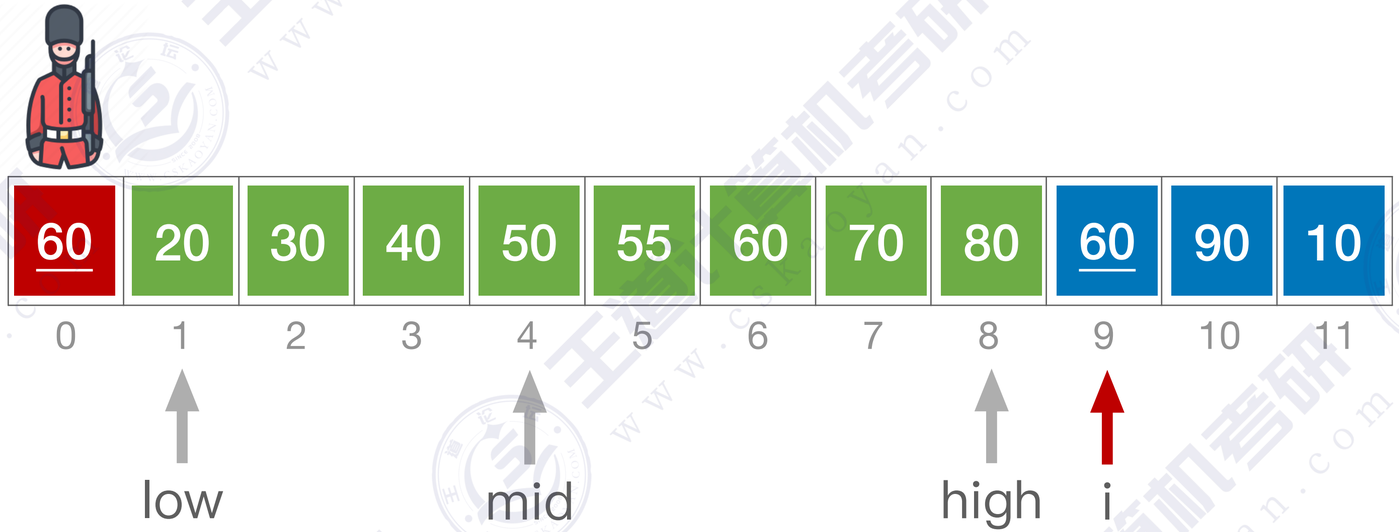
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



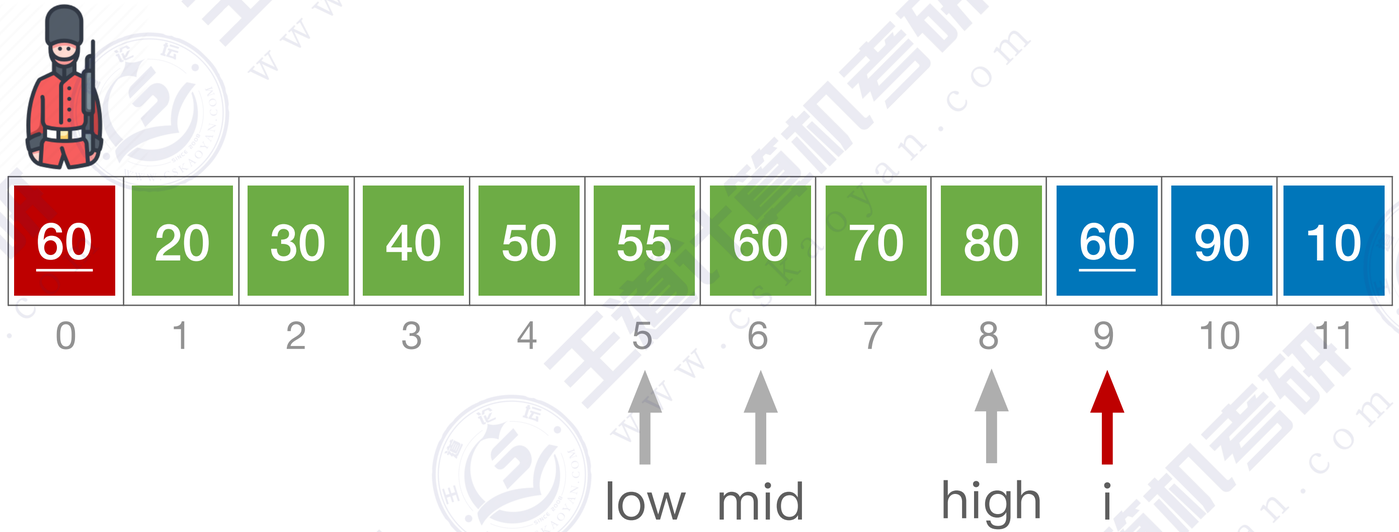
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



优化——折半插入排序

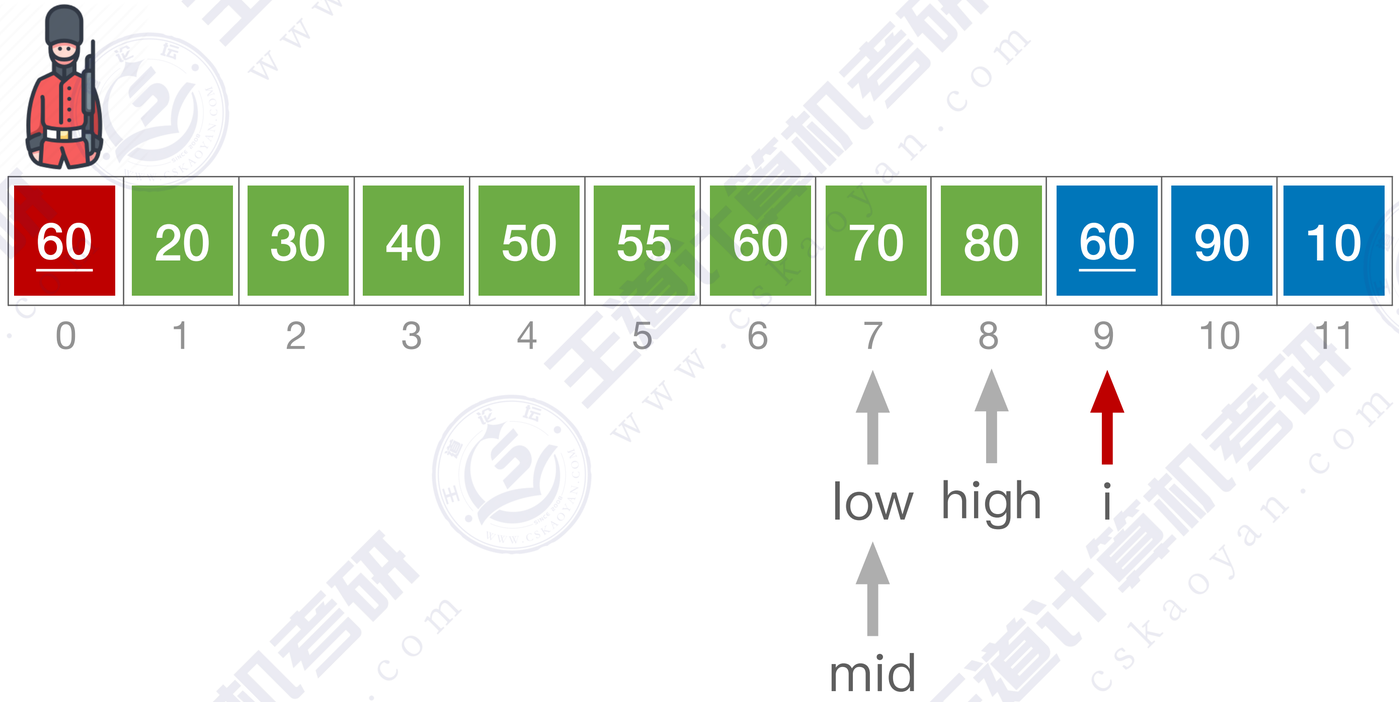
思路：先用折半查找找到应该插入的位置，再移动元素



当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

优化——折半插入排序

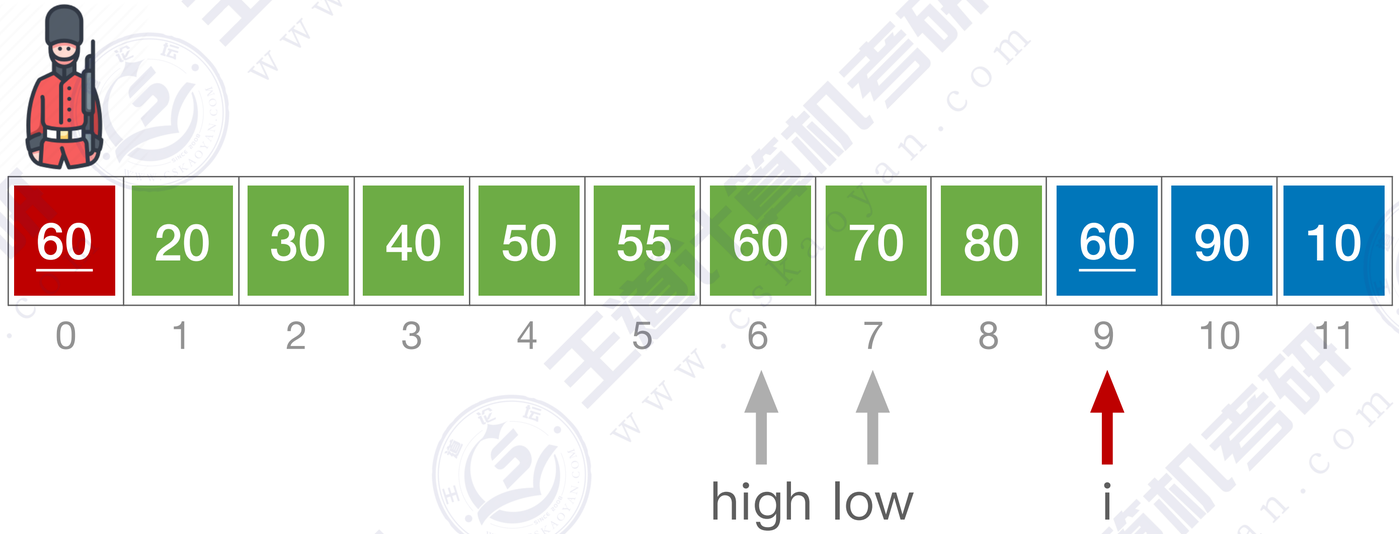
思路：先用折半查找找到应该插入的位置，再移动元素



当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素

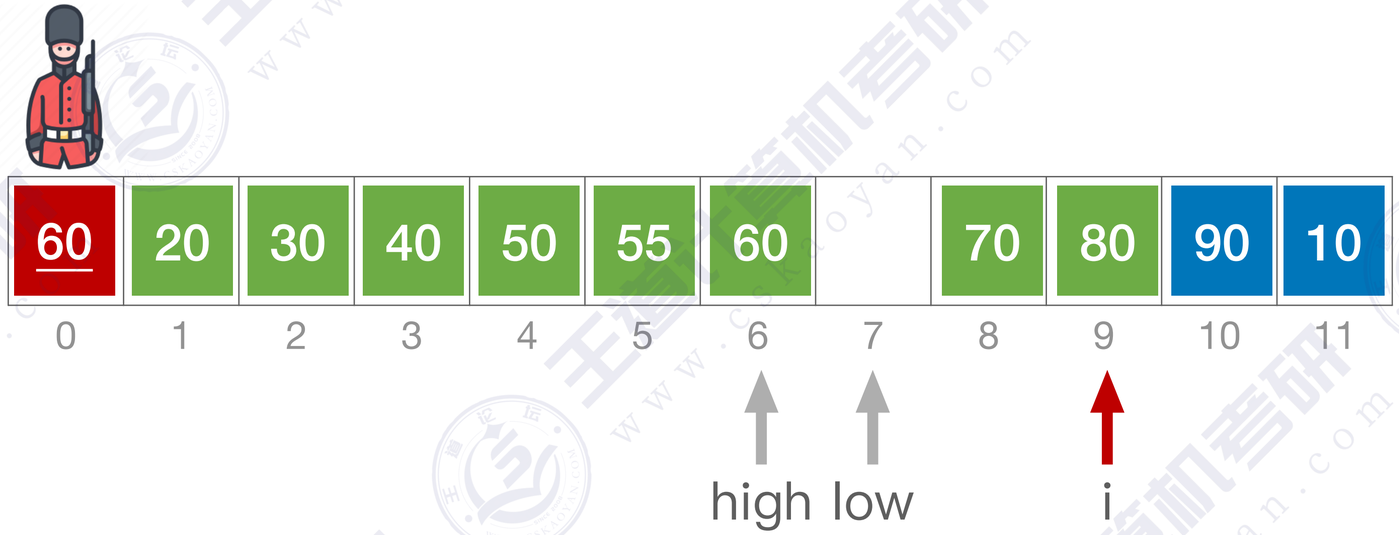


当 low>high 时折半查找停止，应将 [low, i–1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素

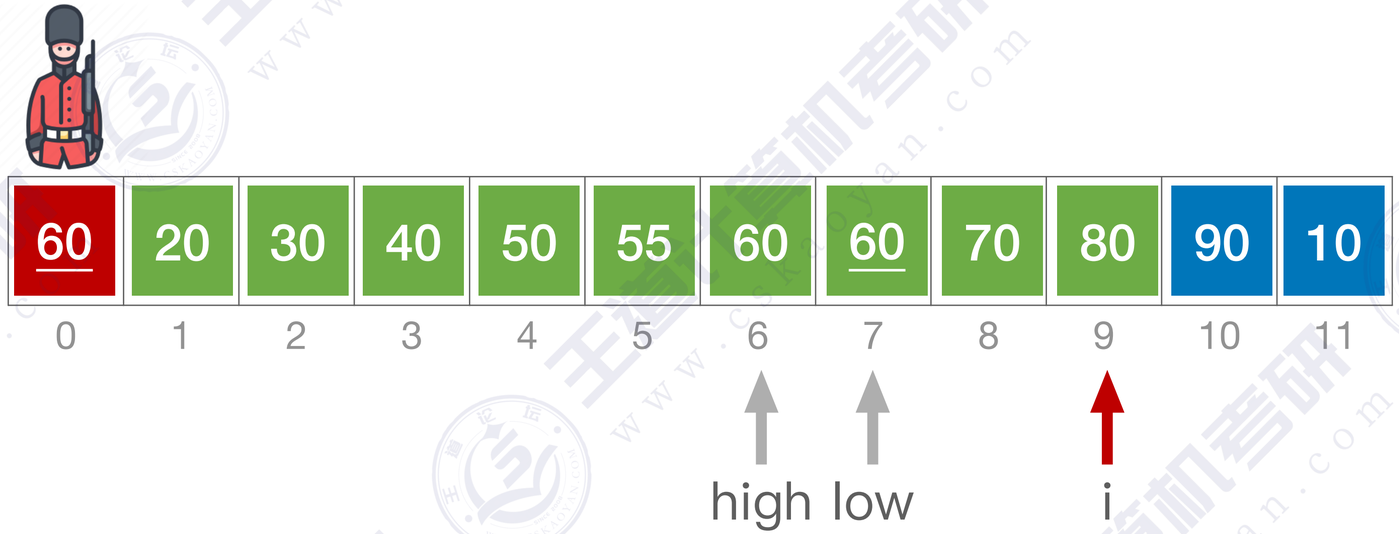


当 low>high 时折半查找停止，应将 [low, i–1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素

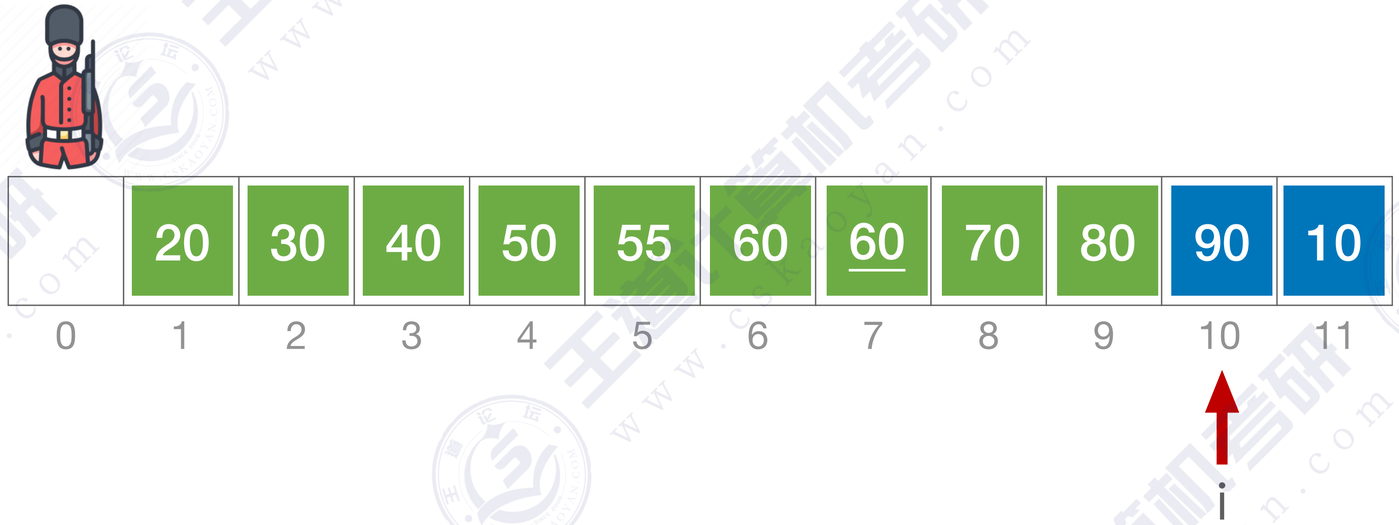


当 low>high 时折半查找停止，应将 [low, i–1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

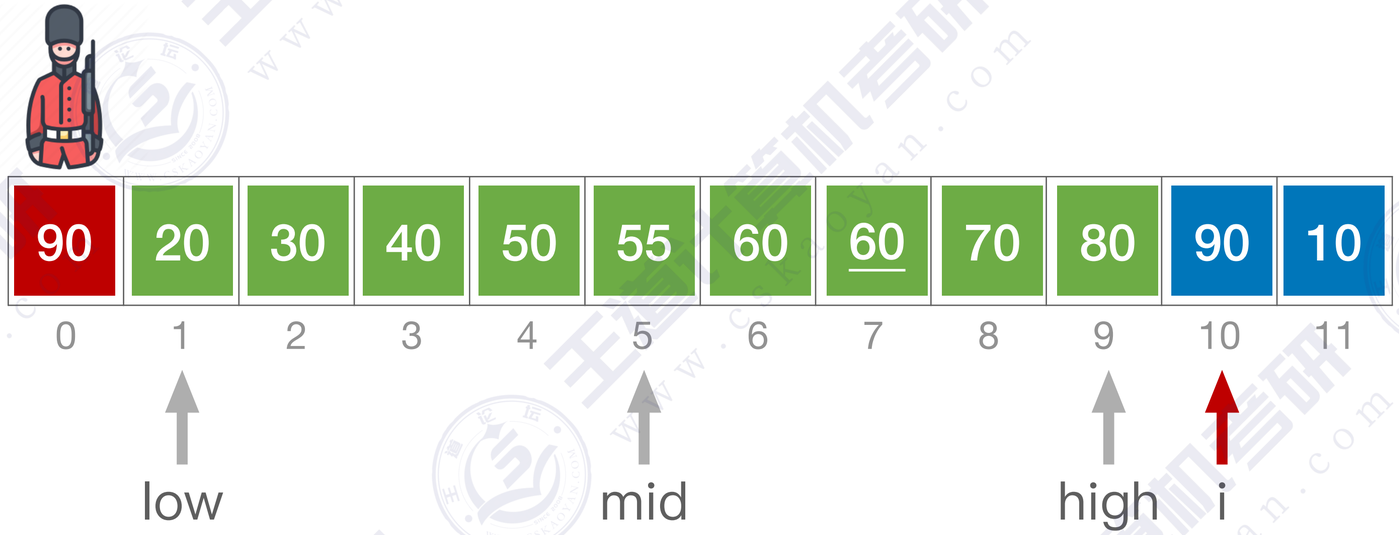
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



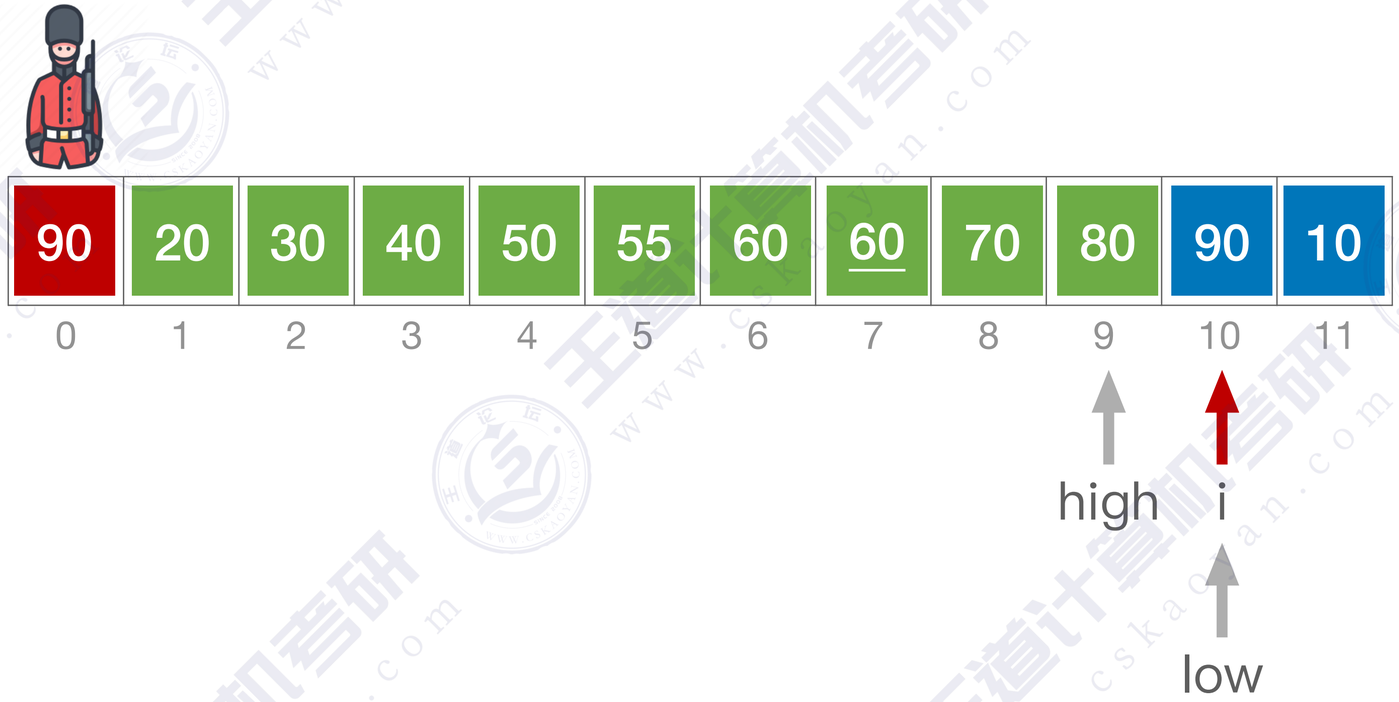
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



优化——折半插入排序

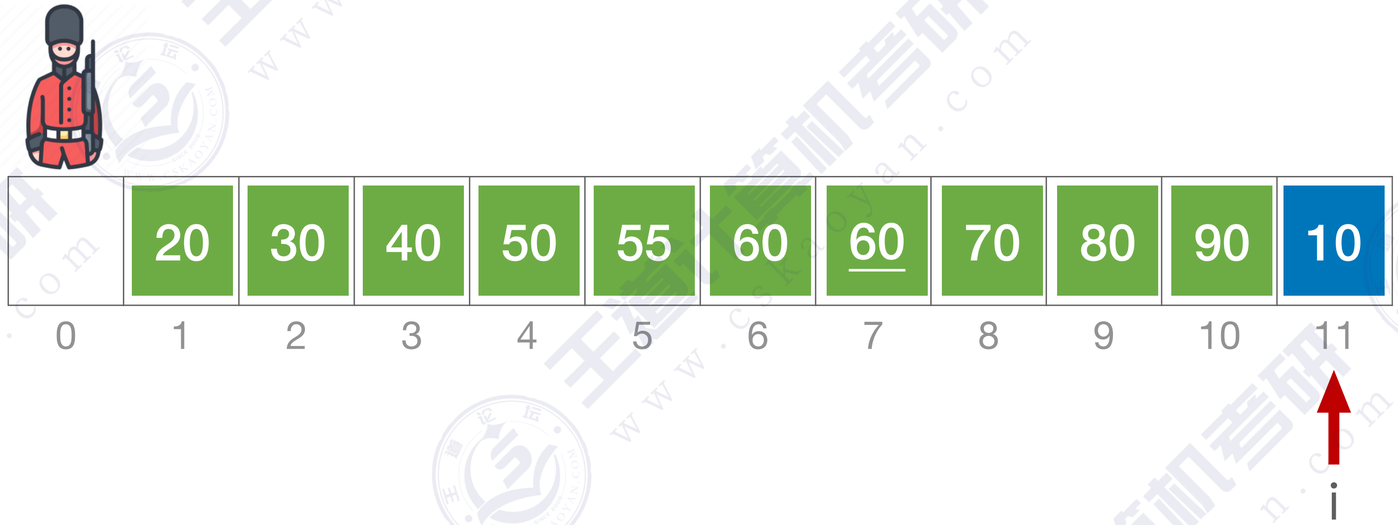
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将~~[low, i-1] 内~~的元素全部右移，并将 A[0] 复制到 low 所指位置

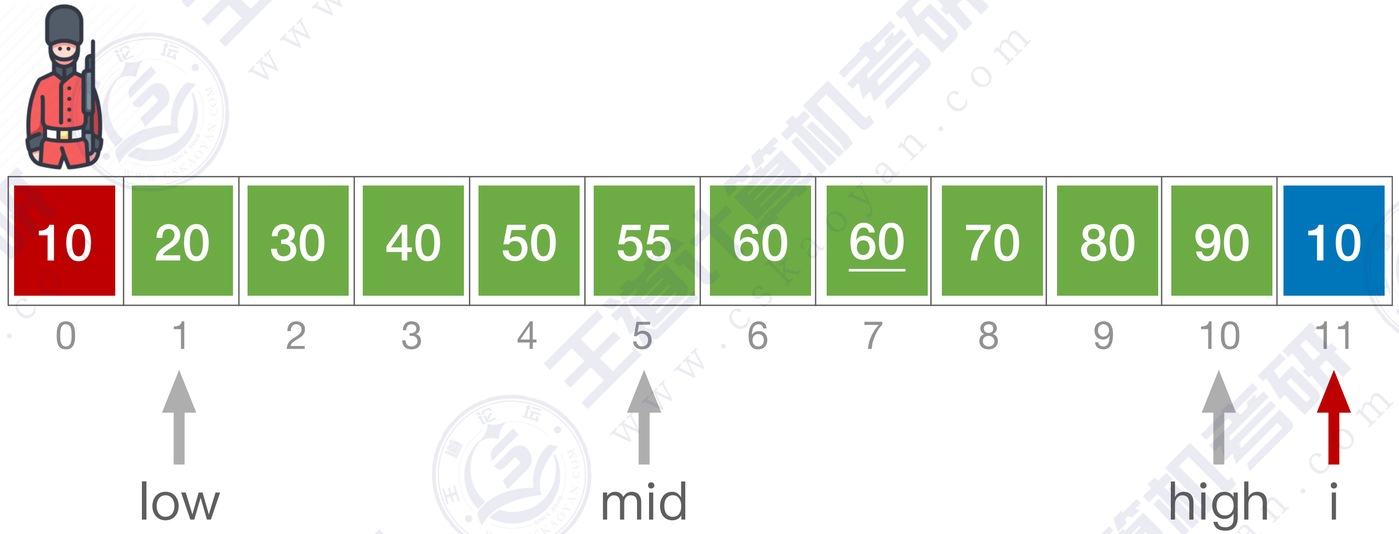
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



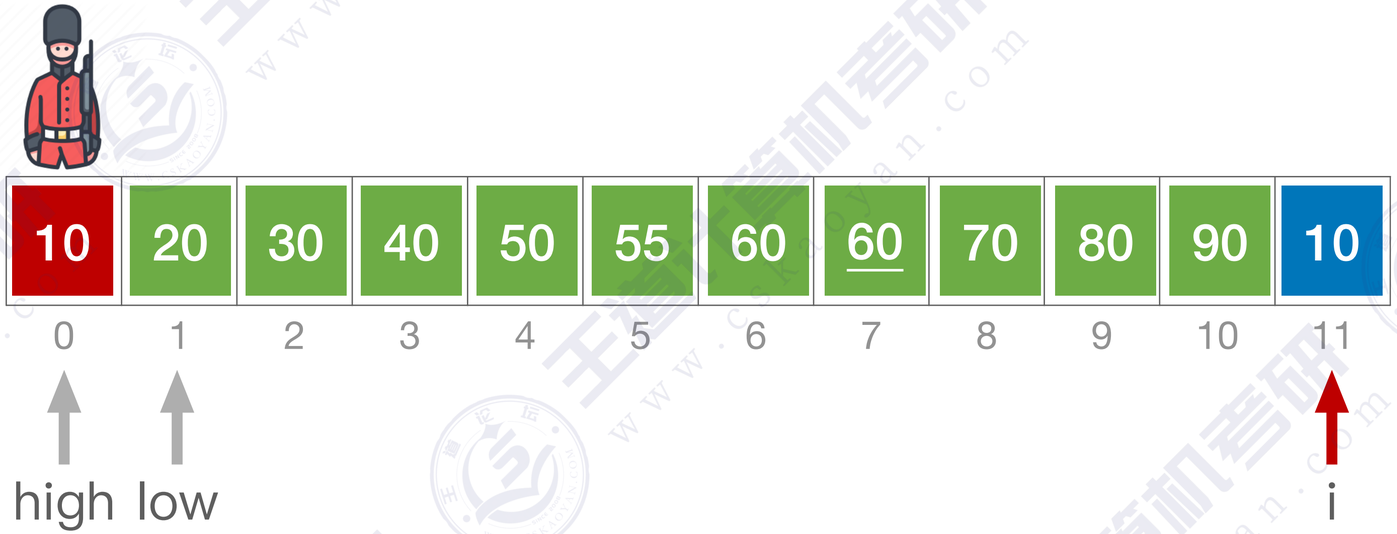
优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



优化——折半插入排序

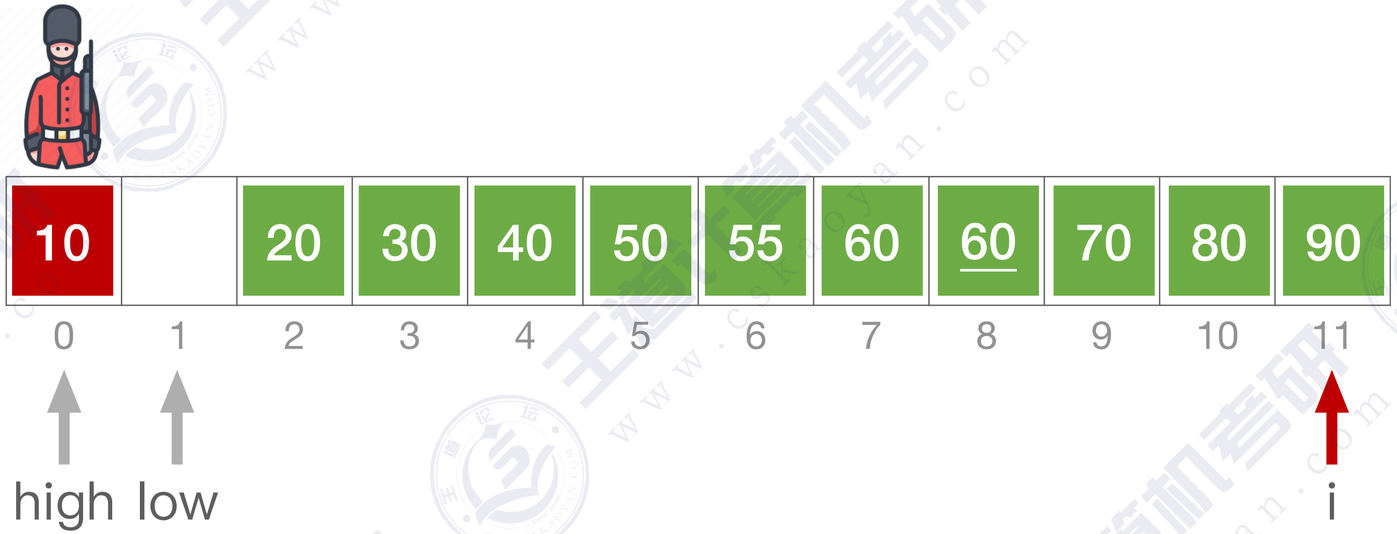
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

优化——折半插入排序

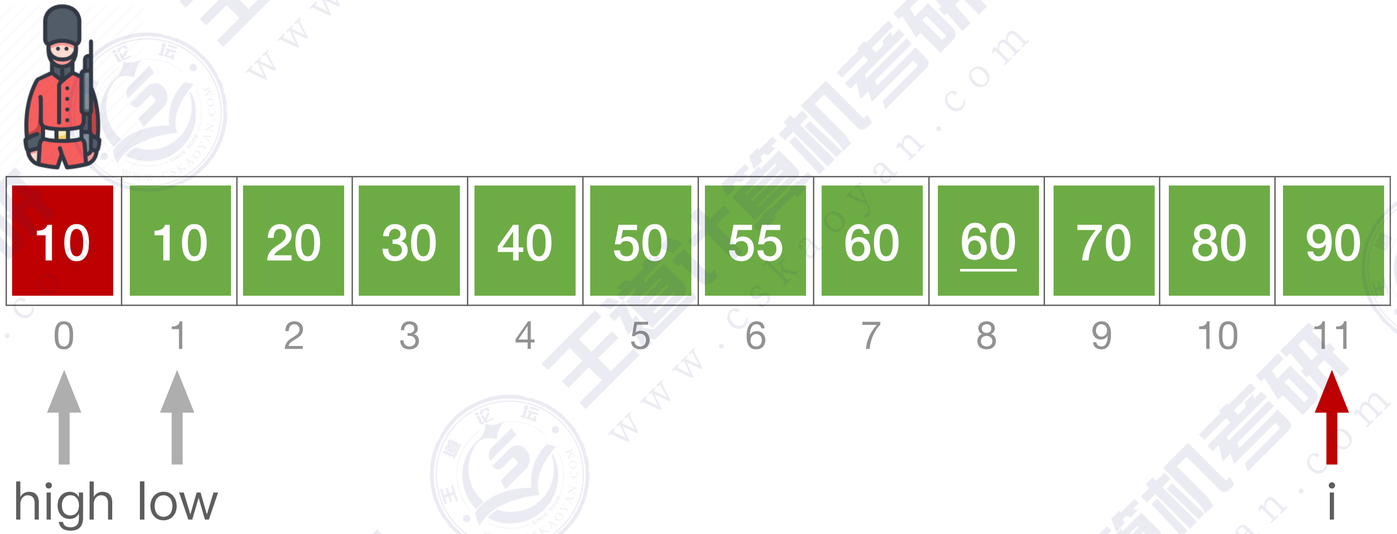
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

优化——折半插入排序

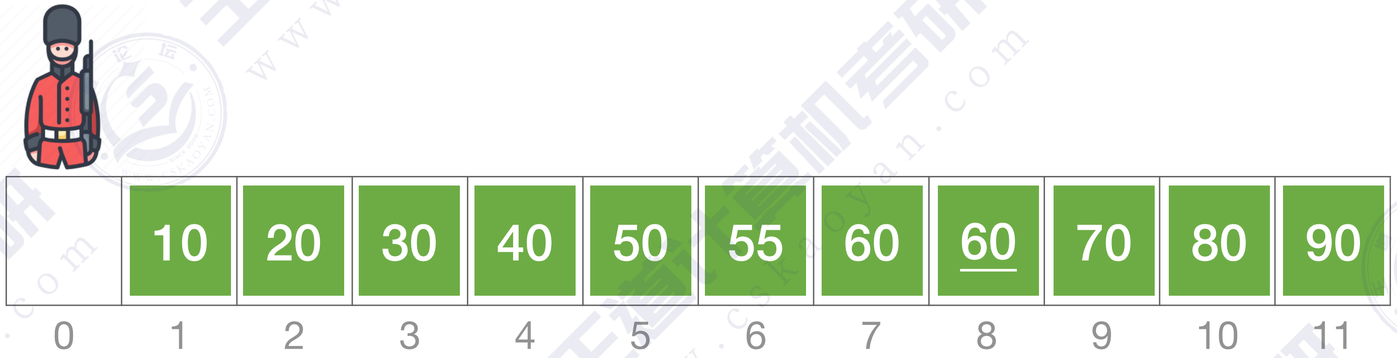
思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

优化——折半插入排序

思路：先用折半查找找到应该插入的位置，再移动元素



当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

优化——折半插入排序

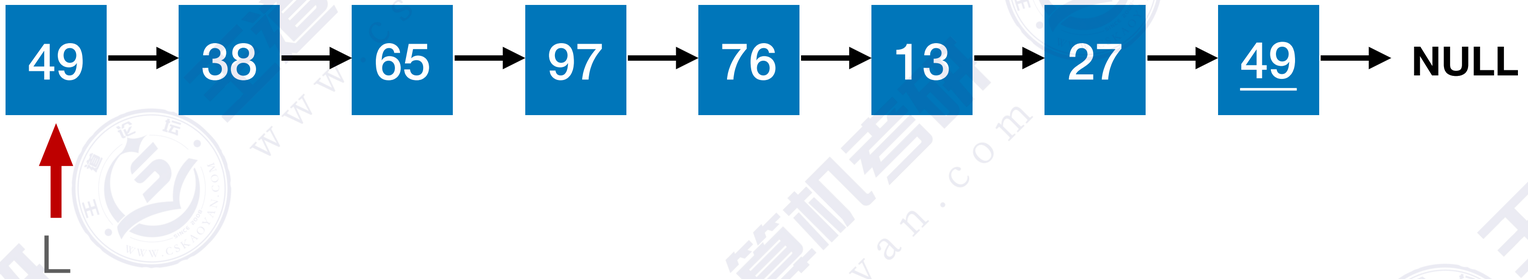
```
//折半插入排序
void InsertSort(int A[],int n){
    int i,j,low,high,mid;
    for(i=2;i<=n;i++){              //依次将A[2]~A[n]插入前面的已排序序列
        A[0]=A[i];                  //将A[i]暂存到A[0]
        low=1;high=i-1;             //设置折半查找的范围
        while(low<=high){           //折半查找(默认递增有序)
            mid=(low+high)/2;       //取中间点
            if(A[mid]>A[0]) high=mid-1; //查找左半子表
            else low=mid+1;         //查找右半子表
        }
        for(j=i-1;j>=high+1;--j)
            A[j+1]=A[j];            //统一后移元素，空出插入位置
        A[high+1]=A[0];             //插入操作
    }
}
```

比起“直接插入排序”，比较关键字的次数减少了，但是移动元素的次数没变，整体来看时间复杂度依然是$O(n^2)$

当 low>high 时折半查找停止，应将 [low, i−1] 内的元素全部右移，并将 A[0] 复制到 low 所指位置

当 A[mid]==A[0] 时，为了保证算法的“稳定性”，应继续在 mid 所指位置右边寻找插入位置

对链表进行插入排序



移动元素的次数变少了，但是关键字对比的次数依然是$O(n^2)$数量级，
整体来看时间复杂度依然是$O(n^2)$

知识回顾与重要考点