计算机程序设计

Computer Programming

期末复习

主讲：吴锋

◎ 期末考试

- 拟定于2022年1月2日下午14:30-16:30进行期末考试。
- **期末考试范围：**
  - **考核内容：**循环、数组、结构（体）、函数、指针、链表（创建、插入、删除）、读递归程序、排序、查找等简单的算法应用、文本文件读写等
  - **不考内容：**补码、进制转换、很偏很难的表达式、位运算、枚举类型、带参数的宏定义、写递归程序、指向函数的指针、main函数参数、文件定位等
- **期末考试题型：**（选择题填答题卡上，需带2B铅笔）
  - 选择(总35分)：20题单选(1分/题)，4题单选(1.5分/题)，6题多选(1.5分/题)
  - 填空(总40分)：10题普通填空(1分/题)，20题程序填空(1.5分/题)
  - 编程(总25分)：算法、数据处理或计算+字符串处理+链表，至少给出函数名和功能（或输入输出样例）要求学生写函数

◎ 顺序查找

- **数据结构：** 以数组或链表表示
- **算法思想：**
  - 从一端开始向另一端，逐个进行记录的关键字和给定值的比较，若某个记录的关键字和给定值比较相等，则查找成功
  - 反之，若直至另一端，其关键字和给定值比较都不等，则表明表中没有所查记录，查找失败

```
int search(int arr[], int n, int key) {
    for (int i = 0; i < n; ++i)
        if (arr[i] == key) return i;
    return -1; // 查找失败
}
```

```
Node *search(Node *head, int key) {
    for (Node *p = head; p != NULL; p = p->next)
        if (p->value == key) return p; // 查找成功
    return NULL; // 查找失败
}
```

◎ 顺序查找

- **数据结构：** 以数组（链表）表示

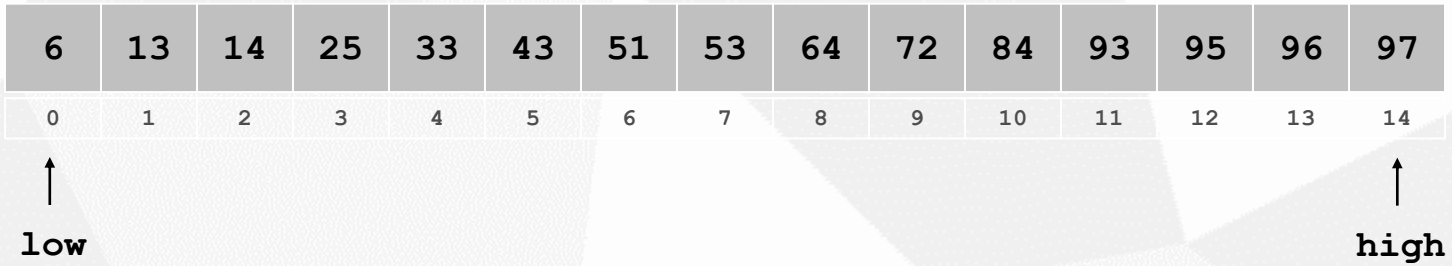
- **最坏情况：** 查找的元素在结构的末端（上面数组中找1）
- **改进思路：** 双向查找（双向链表）

```
int search(int arr[], int n, int key) {
    int left = 0, right = n – 1;
    while (left <= right) {
        if (arr[left] == key) return left;
        if (arr[right] == key) return right;
        ++left; --right;
    }
    return -1; // 查找失败
}
```

```
Node *search(Node *head, Node *tail, int key) {
    Node *left = head, *right = tail;
    while (left != right->next) {
        if (left->value == key) return left;
        if (right->value == key) return right;
        left = left->next; right = right->prev;
    }
    return NULL; // 查找失败
}
```

折半查找 (二分查找)

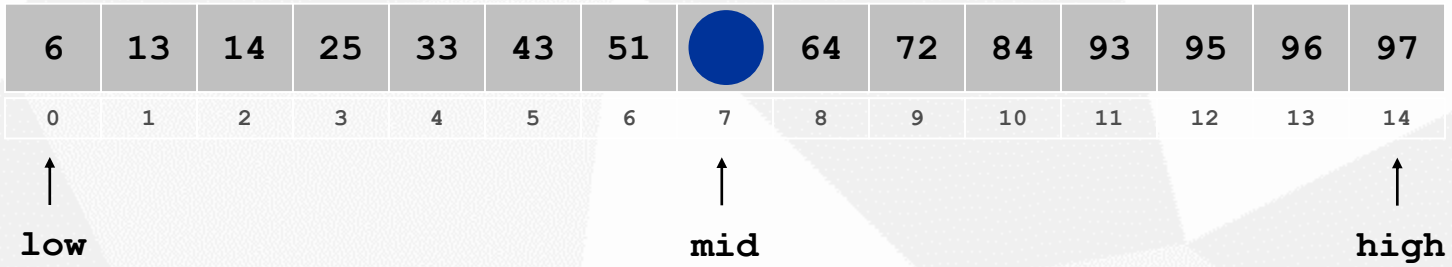
- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33



low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

◎ 折半查找 (二分查找)

- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列从小到大排列的数组arr中查找元素33

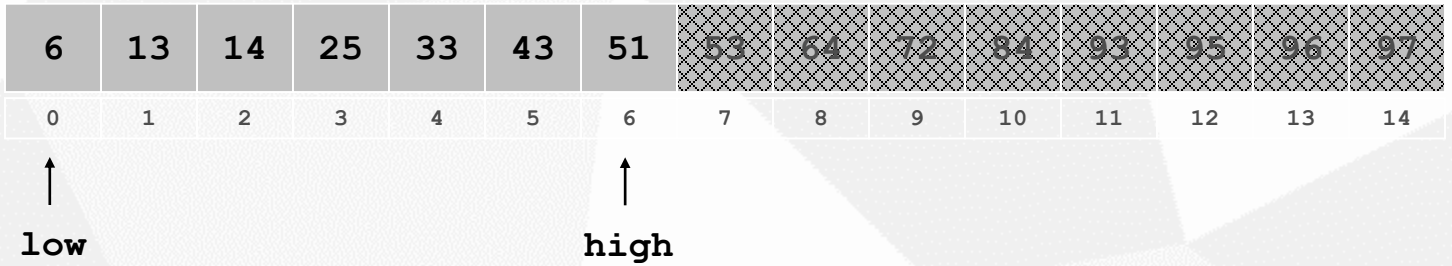


low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

由于 arr[mid] > 33，由于数组元素从小到大排列，33如果存在的话，只可能出现在左半段。

折半查找 (二分查找)

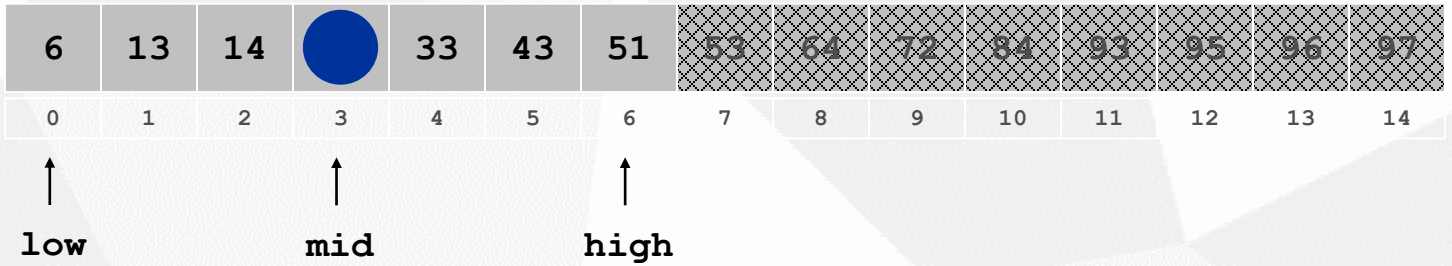
- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33



low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

◎ 折半查找 (二分查找)

- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33

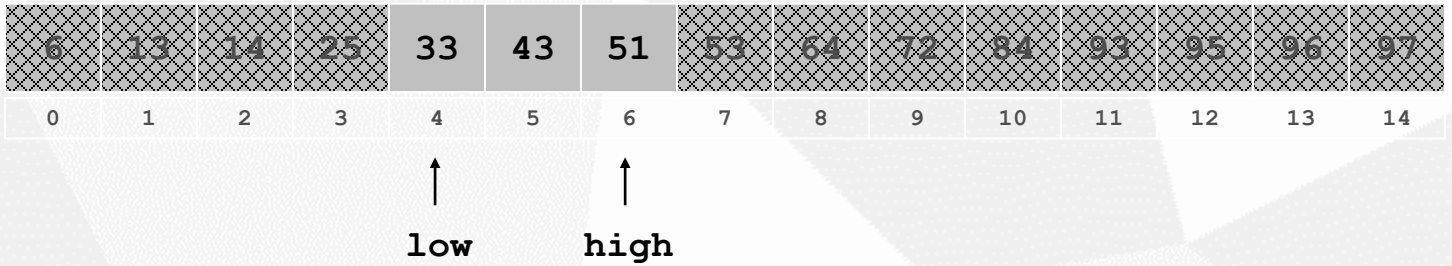


low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

由于 arr[mid] < 33，由于数组元素从小到大排列，33如果存在的话，只可能出现在右半段。

◎ 折半查找 (二分查找)

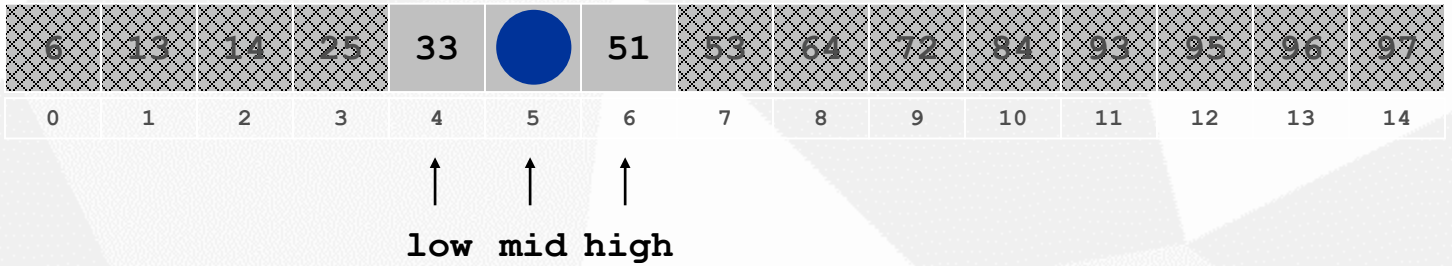
- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33



low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

◎ 折半查找 (二分查找)

- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33

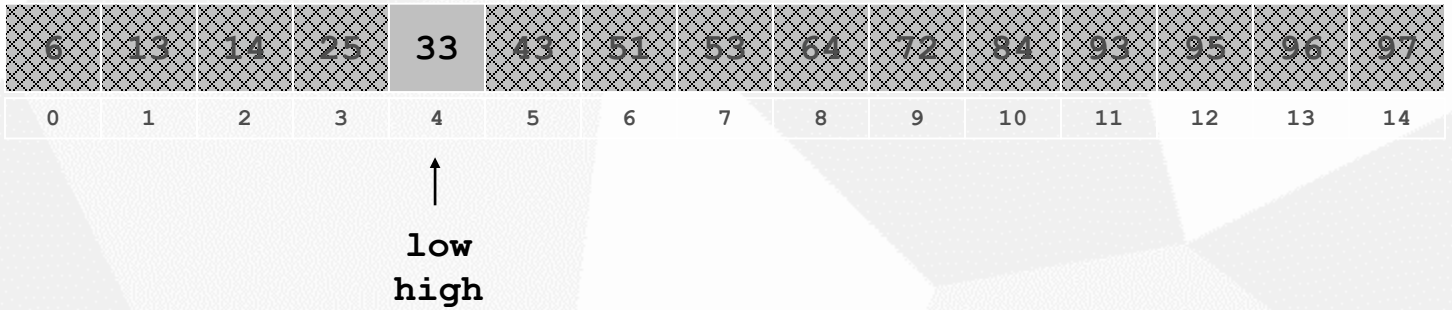


low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

由于 arr[mid] > 33，由于数组元素从小到大排列，33如果存在的话，只可能出现在左半段。

折半查找 (二分查找)

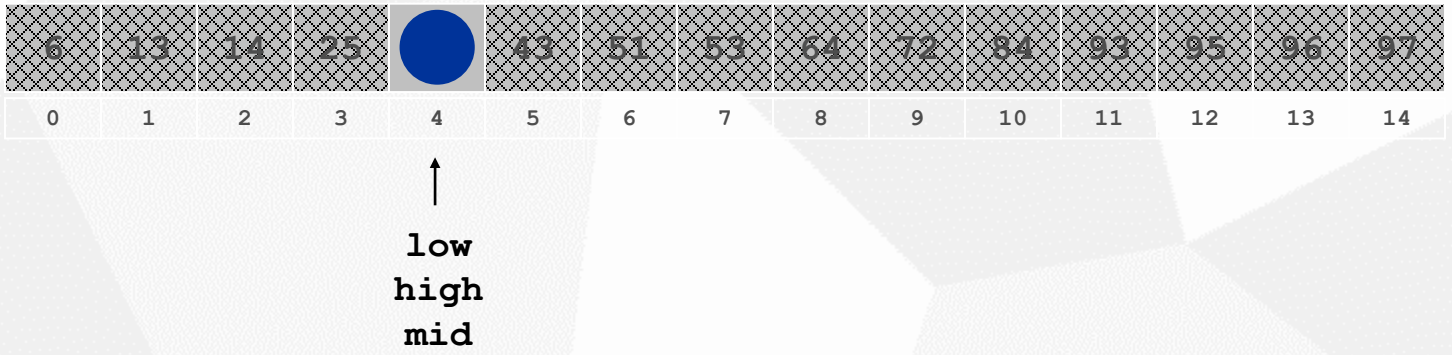
- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33



low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

折半查找 (二分查找)

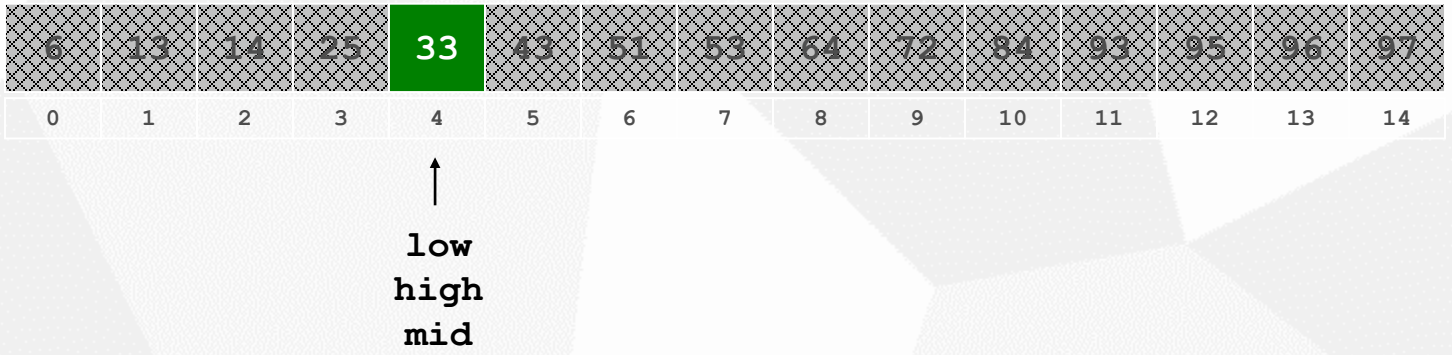
- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33



low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

折半查找 (二分查找)

- **数据结构**：排列有序（从小到大或从大到小）的数组
- **算法思想**：查找区间逐步减半，直到区间中只剩下查找的元素（查找成功）或区间为空（查找失败）
- **例**：在下列有序数组中查找元素33



low：指示查找区间的下界
high：指示查找区间的上界
mid = (low+high)/2

由于 arr[mid] == 33，查找成功；否则查找失败。

◎ 折半查找 (二分查找)

- **数据结构**：排列有序（从小到大或从大到小）的数组
- **时间复杂度**：$O(\log_2 n)$，**空间复杂度**：$O(1)$
- 基于循环的算法实现：

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n-1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid; // 查找成功
        else if (arr[mid] > key) high = mid -1;
        else low = mid +1;
    }
    return -1; // 查找失败，此时 low > high
}
```

注意：当前假设数组是从小到大排列，如果数组是从大到小排列，对high和low的更新要对调。

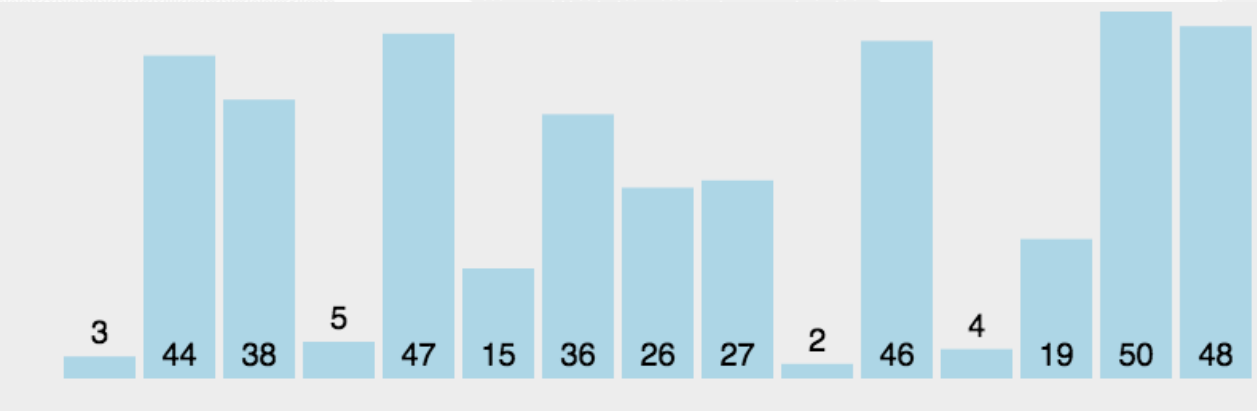
◎ 折半查找 (二分查找)

- **数据结构**：排列有序（从小到大或从大到小）的数组
- **时间复杂度**：$O(\log_2 n)$，**空间复杂度**：$O(1)$
- 基于递归的算法实现：

```
int binarySearch(int arr[], int n, int low, int high, int key) {
    if (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == key) return mid; // 查找成功
        else if (arr[mid] > key)
          return binarySearch(arr, n, low, mid-1, key);
        else return binarySearch(arr, n, mid+1, high, key);
    }
    return -1; // 查找失败，此时 low > high
}
```

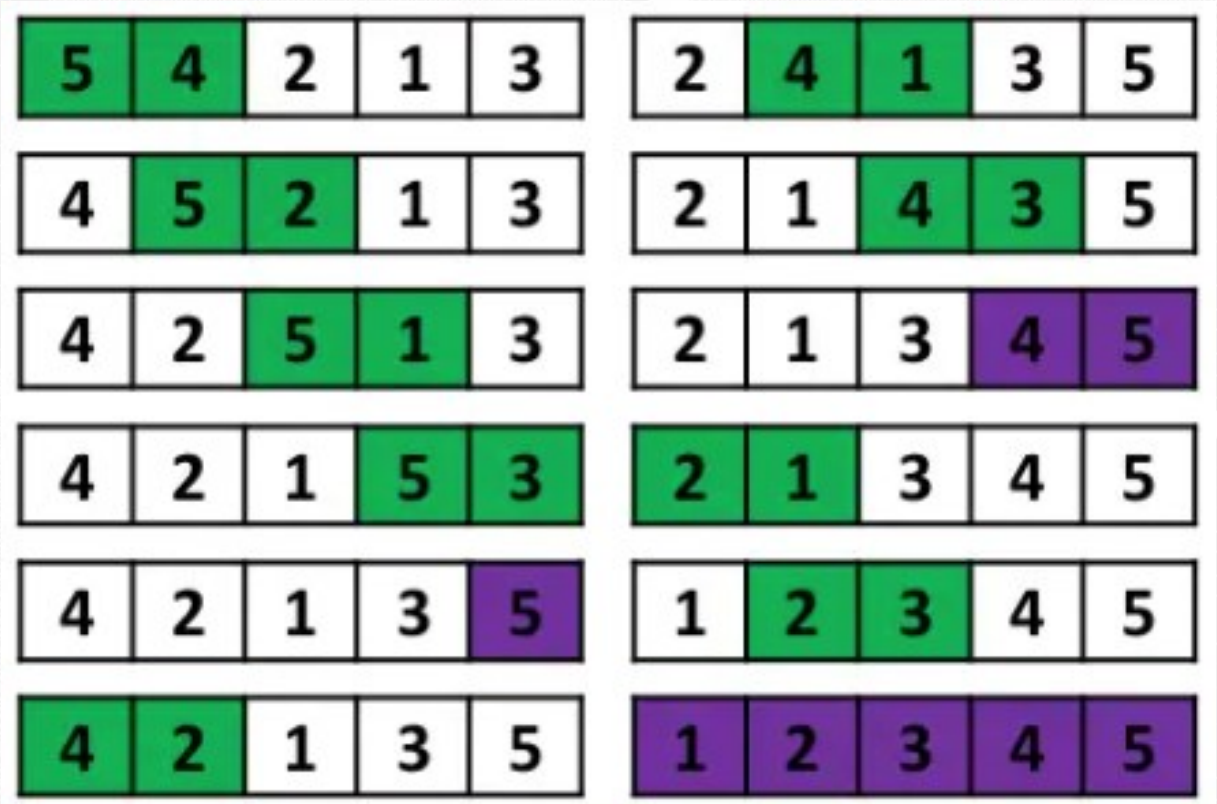
◎ 排序算法

- 冒泡排序（Bubble Sort）
  - 算法思想：反复比较相邻的位置，如果错序则进行交互
  - 算法步骤：
    1. 比较数组中每一对的相邻元素
    2. 如果这两个相邻元素是错序的，则进行交换
    3. 重复上述的1,2步，直到整个数组的元素都有序排列



◎ 排序算法

- 冒泡排序（Bubble Sort）
  - 例：给定下列5个元素的数组，用冒泡排序从小到大排列



排序算法

- 冒泡排序（Bubble Sort）
  - 例：给定数组 [13, 2, 9, 4, 18, 45, 37, 63]，用冒泡排序从小到大排列，经历第一次迭代后，元素的顺序是：

A. [2, 4, 9, 13, 18, 37, 45, 63]

B. [2, 9, 4, 13, 18, 37, 45, 63]

C. [13, 2, 4, 9, 18, 45, 37, 63]

D. [2, 4, 9, 13, 18, 45, 37, 63]

◎ 排序算法

- 冒泡排序（Bubble Sort）
  - 算法思想：反复比较相邻的位置，如果错序则进行交互
  - 算法实现：

```
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i)          // 进行 n-1 次迭代
        for (int j = 0; j < n - 1 - i; ++j) // n-i-1 到 n-1这i个元素已经有序
            if (arr[j] > arr[j+1]) {   // 如果相邻的元素错序则进行交换
                int temp = arr[j]; arr[j] = arr[j+1]; arr[j+1] = temp;
            }
}
```

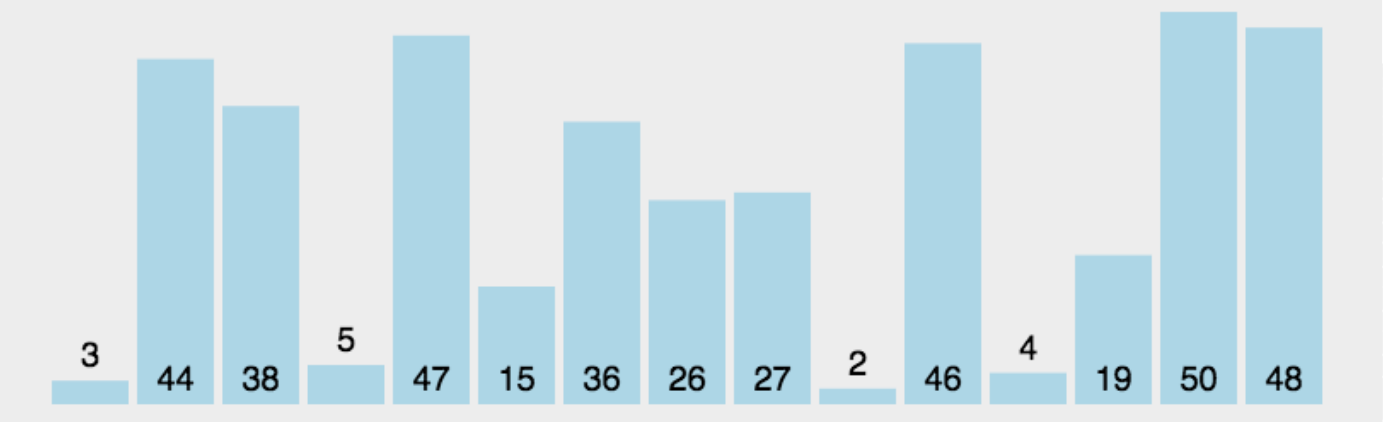
◎ 排序算法

- 冒泡排序（Bubble Sort）
  - 时间复杂度：$O(n^2)$，最好情况n，平均情况n^2，最坏情况n^2
  - 空间复杂度：O(1)
  - 稳定性：稳定
  - 最坏情况：
    - 例

5	4	3	2	1

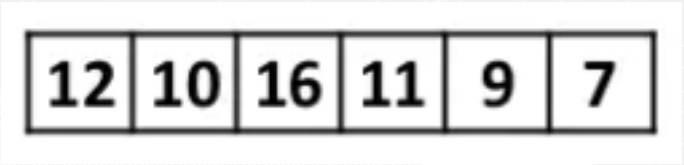
◎ 排序算法

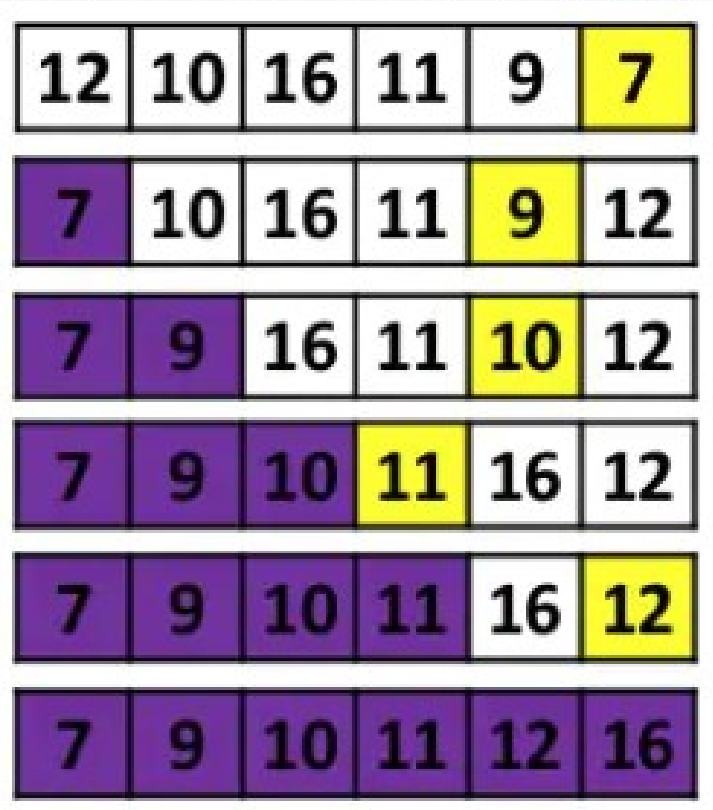
- 选择排序（Selection Sort）
  - 算法思想：不断从未排序部分找到最小元素，置于该部分首位
  - 算法步骤：
    1. 从当前位置到末尾，找到一个最小元素（假设从小到到排列）
    2. 将该元素与当前位置进行交互（如当前位置元素不是最小）
    3. 重复上述1,2步，直到整个数组的元素都有序排列



◎ 排序算法

- 选择排序（Selection Sort）
  - 例：给定下列6个元素的数组，用选择排序从小到大排列





◎ 排序算法

- 选择排序（Selection Sort）
  - 例：给定数组 [13, 2, 9, 4, 18, 45, 37, 63]，用选择排序从小到大排列，经历第二次迭代后，元素的顺序是：

A. [2, 4, 9, 13, 18, 37, 45, 63]

B. [2, 9, 4, 13, 18, 37, 45, 63]

C. [13, 2, 4, 9, 18, 45, 37, 63]

D. [2, 4, 9, 13, 18, 45, 37, 63]

◎ 排序算法

- 选择排序（Selection Sort）
  - 算法思想：不断从未排序部分找到最小元素，置于该部分首位
  - 算法实现：

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {  // 进行 n-1 次迭代
        int min = i;
        for (int j = i+1; j < n; ++j) // 找到i+1到n-1之间的最小元素
            if (arr[min] > arr[j]) min = j;
        if (i != min) {                // 如当前位置元素不是最小的则交换
            int temp = arr[i]; arr[i] = arr[min]; arr[min] = temp;
        }
}
```

◎ 排序算法

- 选择排序（Selection Sort）
  - 时间复杂度：$O(n^2)$，最好情况n^2，平均情况n^2，最坏情况n^2
  - 空间复杂度：O(1)
  - 稳定性：不稳定
  - 最坏情况：
    - 例

2	3	4	5	1

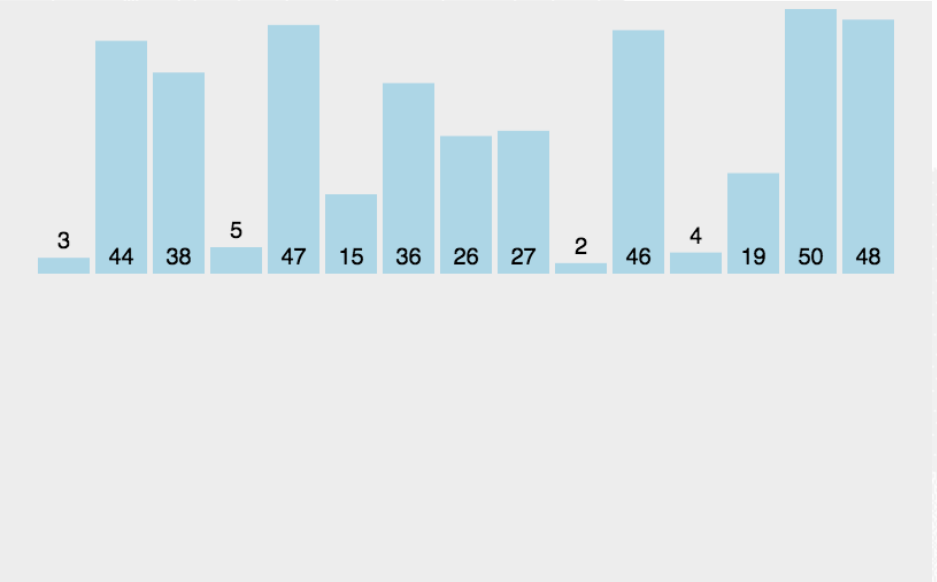
◎ 排序算法

- 交换排序（Exchange Sort）
  - 算法思想：是选择排序的一个更简单但低效的实现
  - 算法实现：

```
void exchangeSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)          // 进行 n-1 次迭代
        for (int j = i + 1; j < n; j++)      // arr[i] 最终将是i+1到n-1间的最小元素
            if (arr[i] > arr[j]) {           // 有可能进行了多次不必要的交换
                int temp = arr[j]; arr[j] = arr[i]; arr[i] = temp;
            }
}
```

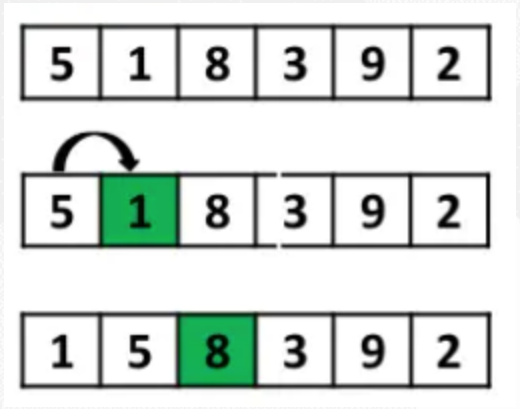
◎ 排序算法

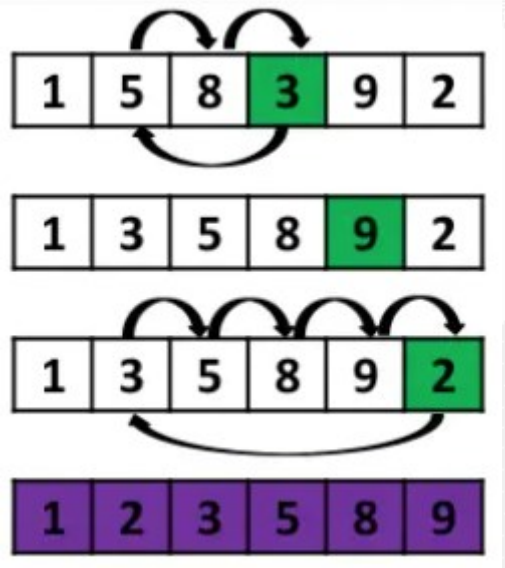
- 插入排序（Insertion Sort）
  - 算法思想：将乱序的元素插入到正确的位置，类似整理扑克牌
  - 算法步骤：
    1. 比较相邻的元素，如果一个元素乱序则将其插入到正确的位置
    2. 重复这个过程，直到所有的元素都变得有序为止



◎ 排序算法

- 插入排序（Insertion Sort）
  - 例：给定下列6个元素的数组，用选择排序从小到大排列





◎ 排序算法

- 插入排序（Insertion Sort）
  - 例：给定数组 [7, 3, 5, 1, 9, 8, 4, 6]，用选择排序从小到大排列，经历第二次迭代后，元素的顺序是：

A. [3, 5, 7, 1, 9, 8, 4, 6]

B. [1, 3, 7, 5, 9, 8, 4, 6]

C. [3, 4, 1, 5, 6, 8, 7, 9]

D. [1, 3, 4, 5, 6, 7, 8, 9]

◎ 排序算法

- 插入排序（Insertion Sort）
  - 算法思想：将乱序的元素插入到正确的位置，类似整理扑克牌
  - 算法实现：

```
void insertionSort(int arr[], int n) {
    for (i = 1; i < n; i++)  {        // 进行 n-1 次迭代
        int j, temp = arr[i];
        for (j = i - 1; j >= 0 && temp < arr[j]; j--)
            arr[j+1] = arr[j]; // 将i前面的元素不断的往后面挪
        arr[j+1] = temp;           // 将第i个元素插入到正确的位置
    }
}
```

◎ 排序算法

- 插入排序（Insertion Sort）
  - 时间复杂度：$O(n^2)$，最好情况n，平均情况n^2，最坏情况n^2
  - 空间复杂度：O(1)
  - 稳定性：稳定
  - 最坏情况：
    - 例