Lec 11 Problem Solving with Pointers and Arrays

指针与多维数组

指针操作

- 对于二维数组

  `int a[2][3];`：

  - a 的类型是 `int (*)[3]`，即指向长度为3的整数数组的指针。
  - `a[i]` 是第 i 行的首地址，类型为 `int *`
- 访问元素的方式：
  - `a[i][j]`：标准下标访问。
  - `*(*(a + i) + j)`：指针解引用，等价于 `a[i][j]`。
  - `*(a[i] + j)`：先取第 i 行的首地址，再偏移 j 个元素。
  - `*((int*)a + i*COL + j)`：将数组视为一维数组，通过指针算术访问。

查找算法

线性查找

什么是线性查找?

线性查找（Linear Search）是一种最简单的查找方法。它的思路是从列表的第一个元素开始，一个一个检查每个元素，直到找到我们要找的目标值，或者检查到列表的最后一个元素也没找到为止。

特点

- **时间复杂度**：最坏情况下是$O(n)$，意思是如果目标值在最后或者不在列表中，我们需要检查所有n个元素。
- **适用场景**：适合小型数据或者列表没有排序的情况。

```c
#include <stdio.h>

int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i;  // 找到目标，返回索引
        }
    }
    return -1;  // 未找到目标，返回-1
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
```

```
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 10;
    int result = linearSearch(arr, n, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }
    return 0;
}
```

二分查找

什么是二分查找?

二分查找（Binary Search）是一种更快的查找方法，但要求数组必须是有序的（比如从小到大排列）。它每次检查数组中间的元素，如果中间元素不是目标，就根据大小关系把搜索范围缩小一半，重复这个过程，直到找到目标或确定目标不存在。

特点

- **时间复杂度**：$O(\log n)$，因为每次搜索范围减半，速度比线性查找快很多。
- **适用场景**：适合有序的大数据集。

```
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int target) {
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;  // 找到目标，返回索引
        }
        if (arr[mid] < target) {
            left = mid + 1;  // 在右半部分继续查找
        } else {
            right = mid - 1;  // 在左半部分继续查找
        }
    }
    return -1;  // 未找到目标，返回-1
}

int main() {
    int arr[] = {2, 3, 4, 10, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 10;
    int result = binarySearch(arr, 0, n - 1, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }
```

```
    return 0;
}
```

排序算法

冒泡排序

什么是冒泡排序?

冒泡排序（Bubble Sort）是一种简单的排序方法。它的思路是重复比较相邻的两个元素，如果前面的比后面的大，就交换它们，这样每一轮最大的元素会像气泡一样“冒”到最后。

特点

- **时间复杂度**：$O(n^2)$，因为要比较很多次，适合小数据。
- **适用场景**：数据量小或者基本有序时。

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) { // 外层循环，控制排序的轮数，总共n-1轮。
        for (int j = 0; j < n - i - 1; j++) { // 内层循环，比较相邻元素。n-i-1是因为每轮后，最后i个元素已经排好。
            if (arr[j] > arr[j + 1]) { // 如果当前元素比下一个大，需要交换。
                // 交换arr[j]和arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

插入排序

什么是插入排序?

插入排序（Insertion Sort）就像整理扑克牌。它从第二个元素开始，把它插入到前面已经排好序的部分，依次处理每个元素，直到整个数组有序。

特点

- **时间复杂度**：$O(n^2)$，适合小数据。
- **适用场景**：数据量小或基本有序时。

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i]; // 把当前元素存到key里，准备插入。
        int j = i - 1; // j指向前一个元素，开始比较。
        while (j >= 0 && arr[j] > key) { // 如果j还在数组范围内且前面的元素大于key，继续循环。
            arr[j + 1] = arr[j]; // 把前面的元素向后移一位。
            j = j - 1; // 向前检查前一个元素。
        }
        arr[j + 1] = key; // 找到key的位置，把它插入。
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
    insertionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```