# 实验四　排序与查找

## 一、实验目的

1. 复习 C 语言中指针、结构体、子程序调用等基础知识；
2. ..掌握常用的排序方法，并掌握用 C 语言实现排序算法的方法；
3. ..深刻理解排序的定义和各种排序方法的特点，并能加以灵活应用；
4. ..了解各种方法的排序过程及其时间复杂度的分析方法。

## 二、实验设备

微机

## 三、预习要求

1. 复习 C 语言中指针的用法，特别是结构体的指针的用法；
2. 理解各种常用的内部排序算法以及外部排序算法的一般步骤，如直接插入排序、希尔排序、简单选择排序、堆排序等等；
3. 复习各种方法的排序过程及其时间复杂度的分析方法；
4. 掌握各种排序算法在解决实际问题的一般方法和步骤。

## 四、实验内容（具体见 pintia）

1. 快速排序
2. 二分查找
3. 排序

## 五、注意事项

1. 提前准备实验的内容
2. 代码需要通过 pintia 测试
3. 将以上实验报告及程序代码**分别**在超星平台上提交。
    a) 实验报告：学号-姓名-实验 4.docx
    b) 代码：学号-姓名-实验 4.zip (包含 exp1.c、exp2.c 和 exp3.c 三个文件)

# 实验内容 1：快速排序

# 文件名：exp1.c

**一、 问题描述:**

给定包含 n 个元素的整型数组 a[1],a[2],...,a[n]，利用快速排序算法对其进行递增排序，请输出排序过程，即每次 Partition 之后的数组。最后输出排序后的数组。每次选择所处理的子数组的第一个元素作为基准元素。

**输入格式:**

输入为两行，第一行为一个整数 n（1<n≤1000），表示数组长度。第二行为 n 个空格间隔的整数，表示待排序的数组。

**输出格式:**

输出为若干行，每行依次输出 Partition 后的数组，每个元素后一个空格。最后一行输出排序后的数组。

**输入样例:**

```
5

4 5 3 2 1
```

**输出样例:**

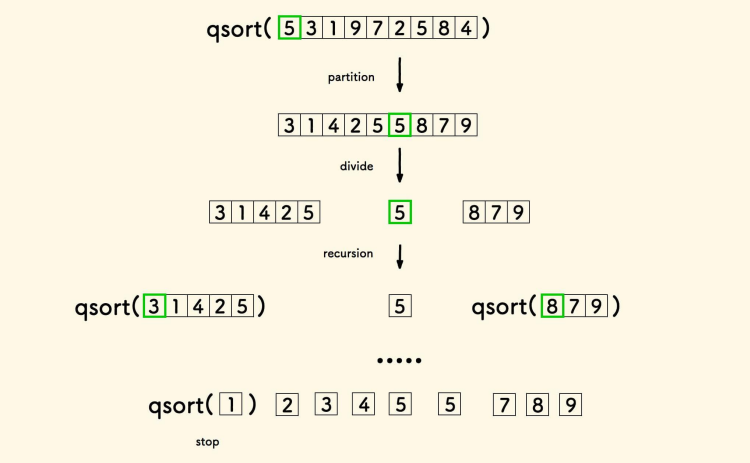
```
2 1 3 4 5

1 2 3 4 5

1 2 3 4 5
```

**二、 数据结构设计和核心算法（用图和文字描述清楚）：**

## 数据结构设计

整型数组 a[]：用于存储待排序的数列。
整数变量： n（数组长度）、low（排序子数组的起始下标）、high（排序子数组的终止下标）等，用于控制排序过程。

## 核心算法




选择基准（Pivot）：在本程序中，每次处理的子数组的第一个元素作为基准。

分区（Partition）：
将数组划分为两部分，使得：
左侧所有元素 <= 基准值。
右侧所有元素 > 基准值。
这一过程中，基准元素达到其最终位置。

递归排序：对左右两个子数组递归地进行同样的分区和排序过程。

终止条件：当子数组的长度为 1（或 0），即 low >= high，递归结束。

## 三、程序及注释

程序名： **exp1.c**

```
#include <stdio.h>

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

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int a[], int low, int high) {
```

```
    int pivot = a[low];  // 选择基准
    int i = low, j = high;
    while (i < j) {
        // while (i < j && a[j] >= pivot) j--; // 从右向左找小于等于基
        准的元素
        while (i < j && a[j] > pivot)
            j--;  // 与基准相等的元素不会移动到基准的左侧
        while (i < j && a[i] <= pivot) i++;
        swap(&a[i], &a[j]);
    }
    swap(&a[low], &a[i]);  // 基准记录到位
    return i;
}

void quickSort(int a[], int low, int high, int n) {
    if (low < high) {
        int pi = partition(a, low, high);
        printArray(a, n);
        quickSort(a, low, pi - 1, n);
        quickSort(a, pi + 1, high, n);
    }
}

int main() {
    int n;
    scanf("%d", &n);
    int a[n];
    for (int i = 0; i < n; i++) scanf("%d", &a[i]);
    quickSort(a, 0, n - 1, n);
    printArray(a, n);
    return 0;
}
```

四、运行结果：

```
(base) nanmener@Haotians-MacBook-Pro 实验4 % ./"1"
5
4 5 3 2 1
2 1 3 4 5
1 2 3 4 5
1 2 3 4 5
```

五、心得体会

通过本次快速排序实验，我深刻理解了快速排序算法的核心原理与实际应

用。在编写和调试程序的过程中，我逐渐领会了如何有效地选择基准元素，并将数组分成两部分的技巧。实际上手编程并观察每次分区后数组的变化，让我从理论走向了实践，这种经验是阅读书本无法替代的。

在编程过程中，我在处理数组边界和确保排序稳定性方面遇到了一些问题。通过不断调试和修改代码，我学会了如何识别和解决这些实际编程中的问题。这不仅提升了我的编程能力，也锻炼了我的逻辑思维和问题解决技巧。

这次实验加深了我对快速排序算法的理解，并扩展了我的知识面。我了解尽管快速排序在最坏情况下的时间复杂度为 $O(n^2)$，但它的平均时间复杂度为 $O(n \log n)$，在多数情况下表现优异。这种认识促使我在选择排序算法时，更加注重实际数据的特性和应用场景。

# 实验内容 2：二分查找

# 文件名：exp2.c

**一、 问题描述：**

输入 n 值(1<=n<=1000)、n 个非降序排列的整数以及要查找的数 x，使用二分查找算法查找 x，输出 x 所在的下标（0~n-1）及比较次数。若 x 不存在，输出-1 和比较次数。

**输入格式:**

输入共三行：
第一行是 n 值；
第二行是 n 个整数；
第三行是 x 值。

**输出格式:**

输出 x 所在的下标（0~n-1）及比较次数。若 x 不存在，输出-1 和比较次数。

**输入样例:**

```
4

1 2 3 4

1
```

**输出样例:**

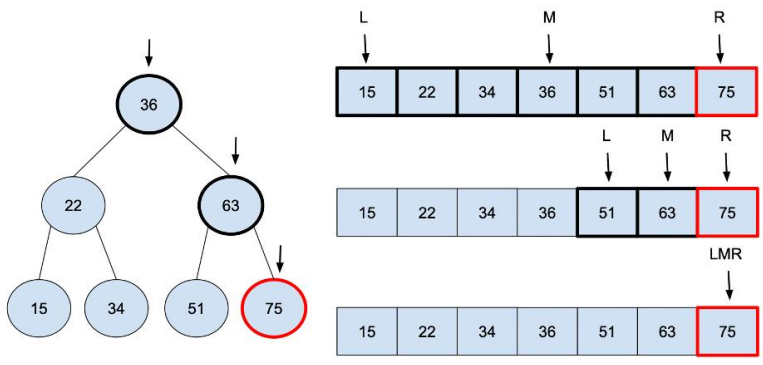
```
0

2
```

**二、 数据结构设计和核心算法（用图和文字描述清楚）：**

## 数据结构设计

数组 arr[]：用来存储输入的非降序排列的整数序列。
整型变量 n：表示数组 arr[] 的长度，即序列中整数的数量。
整型变量 x：代表要查找的数。
整型指针 count：用来记录比较的次数。

## 核心算法




二分查找算法：这是一种在有序数组中查找某一特定元素的搜索算法。搜索过程从数组的中间元素开始，如果中间元素正好是目标值，则搜索过程结束；如果目标值大于或小于中间元素，则搜索继续在数组较大或较小的半部分，以此类推，直到找到目标值或剩下的半部分为空。

## 三、程序及注释

程序名： **exp2.c**

```
#include <stdio.h>

int binarySearch(int arr[], int l, int r, int x, int *count) {
    while (l <= r) {
        int m = l + (r - l) / 2;
        (*count)++;
        if (arr[m] == x) return m;
        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }
    return -1;
}

int main() {
    int n, x;
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);
    scanf("%d", &x);
    int count = 0;
    int result = binarySearch(arr, 0, n - 1, x, &count);
    printf("%d\n%d\n", result, count);
```

```
    return 0;
}
```

**四、运行结果：**

```
(base) nanmener@Haotians-MacBook-Pro 实验4 % ./"2"
4
1 2 3 4
1
0
2
```

**五、心得体会**

通过这次实验，我对二分查找算法有了更深刻的理解。二分查找不仅是一种高效的搜索算法，它的优势在于大幅减少了在有序数据集中查找元素所需的比较次数。在实现这个算法的过程中，我深刻感受到算法逻辑的精妙与实现的严谨性。通过实际编写和测试代码，我学会了如何将理论转化为实际应用，这不仅仅是编程技能的提升，更是对问题解决方法的一种训练。

在编程实践中，我也遇到了一些问题，比如正确处理边界条件和理解指针的使用。初始时，对于如何精确地调整搜索的边界范围感到困惑，经过不断的测试和调试，我逐渐掌握了边界条件的处理方法。同时，通过这个实验，我对 C 语言中指针的作用有了更加深入的理解，尤其是在通过指针来记录数据比较次数的过程中。这不仅提高了我的编程能力，也增强了我解决复杂问题的信心。

这次实验加深了我对算法效率重要性的认识。在现实世界的应用中，优化算法以减少计算时间和资源消耗是至关重要的。通过比较二分查找与传统线性查找的效率，我更加明白为什么在大数据量的情况下，选择合适的算法至关重要。这种认识不仅限于学术层面，更是在未来职业生涯中，指导我进行更高效算法选择和优化的重要依据。

# 实验内容 3：排序

# 文件名：exp3.c

## 一、 问题描述：

给定 N 个（长整型范围内的）整数，要求输出从小到大排序后的结果。

本题旨在测试各种不同的排序算法在各种数据情况下的表现。各组测试数据特点如下：

数据 1：只有 1 个元素；

数据 2：11 个不相同的整数，测试基本正确性；

数据 3：$10^3$ 个随机整数；

数据 4：$10^4$ 个随机整数；

数据 5：$10^5$ 个随机整数；

数据 6：$10^5$ 个顺序整数；

数据 7：$10^5$ 个逆序整数；

数据 8：$10^5$ 个基本有序的整数；

数据 9：$10^5$ 个随机正整数，每个数字不超过 1000。

**输入格式：**

输入第一行给出正整数 N（$\leqslant 10^5$），随后一行给出 N 个（长整型范围内的）整数，其间以空格分隔。

**输出格式：**

在一行中输出从小到大排序后的结果，数字间以 1 个空格分隔，行末不得有多余空格。

**输入样例：**

11

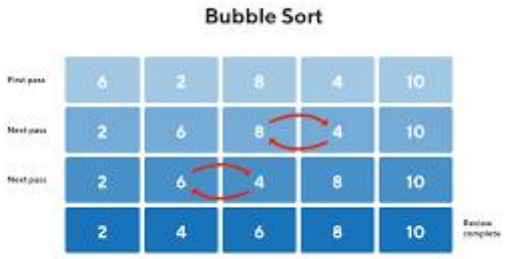
4 981 10 -17 0 -20 29 50 8 43 -5

**输出样例：**

-20 -17 -5 0 4 8 10 29 43 50 981

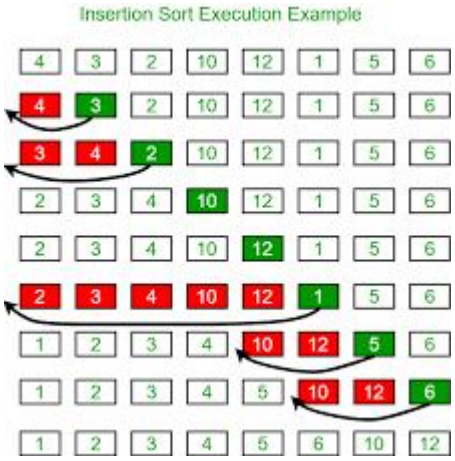
## 二、数据结构设计和核心算法（用图和文字描述清楚）：

## 冒泡排序



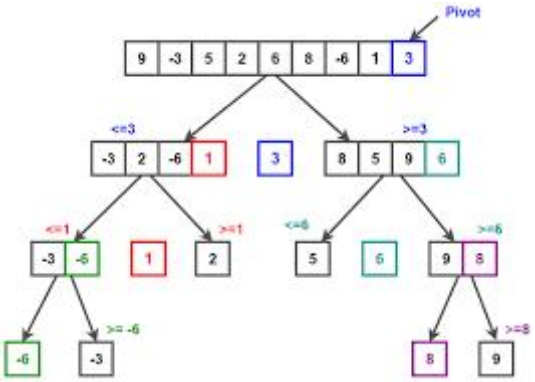

基本思想：通过不断比较相邻元素，如果顺序错误就交换它们，直到没有任何一对数字需要交换。
时间复杂度：平均和最差情况下均为 O(n^2)。
适用场景：小规模数据或几乎已经排序好的数据。

## 插入排序



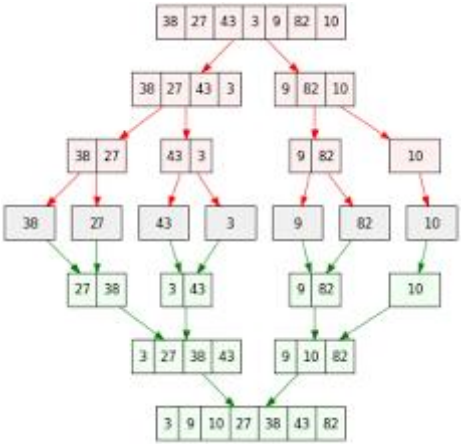

基本思想：构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。
时间复杂度：平均为 O(n^2)，最好情况下为 O(n)。
适用场景：小规模数据或几乎已经排序好的数据。

## 快速排序




基本思想：选择一个基准值，通过一趟排序将待排序的记录分隔成独立的两部分，其中一部分记录的键值均比另一部分的键值小，然后分别对这两部分记录继续进行排序，以达到整个序列有序。
时间复杂度：平均为 O(n log n)，最差情况下为 O(n^2)。
适用场景：大多数中等规模以上的数据。

## 归并排序




基本思想：采用分治法的一个非常典型的应用，将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。
时间复杂度：对所有情况均为 O(n log n)。
适用场景：大规模数据及对稳定排序有要求的场景。

### 三、程序及注释

程序名： **exp3.c**

```
#include <stdio.h>
#include <stdlib.h>
```

```c
void swap(int *a, int *b);
void bubbleSort(int arr[], int n);
void insertionSort(int arr[], int n);
void quickSort(int arr[], int left, int right);
void mergeSort(int arr[], int l, int r);
void merge(int arr[], int l, int m, int r);

int main() {
    int n;
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);

    bubbleSort(arr, n);
    // insertionSort(arr, n);
    // quickSort(arr, 0, n - 1);
    // mergeSort(arr, 0, n - 1);

    for (int i = 0; i < n - 1; i++) printf("%d ", arr[i]);
    printf("%d", arr[n - 1]);
    return 0;
}

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) swap(&arr[j], &arr[j + 1]);
}

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            // 大于 key 的元素右移，为 key 的插入腾出空间
            arr[j + 1] = arr[j];
            j--;
        }
```

```
        arr[j + 1] = key;  // 插入 key
    }
}

int partition(int a[], int low, int high) {
    int pivot = a[low];  // 选择基准
    int i = low, j = high;
    while (i < j) {
        // while (i < j && a[j] >= pivot) j--; // 从右向左找小于等于基
准的元素
        while (i < j && a[j] > pivot)
            j--;  // 与基准相等的元素不会移动到基准的左侧
        while (i < j && a[i] <= pivot) i++;
        swap(&a[i], &a[j]);
    }
    swap(&a[low], &a[i]);  // 基准记录到位
    return i;
}

void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);  // 分割、归并左边
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void merge(int arr[], int l, int m, int r) {
    int i, j, k;
    int n1 = m - l + 1, n2 = r - m;
    int L[n1], R[n2];  // 左、右的有序数组
    for (i = 0; i < n1; i++) L[i] = arr[l + i];
    for (j = 0; j < n2; j++) R[j] = arr[m + 1 + j];
    i = 0, j = 0, k = l;
    while (i < n1 && j < n2)  // 选择两个数组中较小的元素放入原数组
```

```
        if (L[i] <= R[j])
            arr[k++] = L[i++];
        else
            arr[k++] = R[j++];
    while (i < n1) arr[k++] = L[i++];  // 复制剩余元素
    while (j < n2) arr[k++] = R[j++];
}
```

## 四、运行结果：

数据 0：只有 1 个元素；
数据 1：11 个不相同的整数，测试基本正确性；
数据 2：1e3 个随机整数；
数据 3：1e4 个随机整数；
数据 4：1e5 个随机整数；
数据 5：1e5 个顺序整数；
数据 6：1e5 个逆序整数；
数据 7：1e5 个基本有序的整数；
数据 8：1e5 个随机正整数，每个数字不超过 1000。

# 快速排序

评测详情

| 测试点 | 提示 | 内存(KB) | 用时(ms) | 结果 | 得分 |
|---|---|---|---|---|---|
| 0 | | 352 | 4 | 答案正确 | 6 / 6 |
| 1 | | 320 | 4 | 答案正确 | 8 / 8 |
| 2 | | 352 | 4 | 答案正确 | 6 / 6 |
| 3 | | 476 | 7 | 答案正确 | 6 / 6 |
| 4 | | 1212 | 42 | 答案正确 | 2 / 2 |
| 5 | | 1280 | 3898 | 答案正确 | 2 / 2 |
| 6 | | 3692 | 3959 | 答案正确 | 2 / 2 |
| 7 | | 1424 | 2972 | 答案正确 | 2 / 2 |
| 8 | | 1080 | 31 | 答案正确 | 6 / 6 |

## 归并排序

评测详情

| 测试点 | 提示 | 内存(KB) | 用时(ms) | 结果 | 得分 |
|---|---|---|---|---|---|
| 0 | | 352 | 4 | 答案正确 | 6 / 6 |
| 1 | | 320 | 3 | 答案正确 | 8 / 8 |
| 2 | | 348 | 4 | 答案正确 | 6 / 6 |
| 3 | | 480 | 7 | 答案正确 | 6 / 6 |
| 4 | | 1676 | 34 | 答案正确 | 2 / 2 |
| 5 | | 1660 | 37 | 答案正确 | 2 / 2 |
| 6 | | 1728 | 30 | 答案正确 | 2 / 2 |
| 7 | | 1632 | 36 | 答案正确 | 2 / 2 |
| 8 | | 1492 | 36 | 答案正确 | 6 / 6 |

## 插入排序

评测详情

| 测试点 | 提示 | 内存(KB) | 用时(ms) | 结果 | 得分 |
|---|---|---|---|---|---|
| 0 | | 192 | 2 | 答案正确 | 6 / 6 |
| 1 | | 356 | 3 | 答案正确 | 8 / 8 |
| 2 | | 256 | 3 | 答案正确 | 6 / 6 |
| 3 | | 332 | 29 | 答案正确 | 6 / 6 |
| 4 | | 1200 | 1990 | 答案正确 | 2 / 2 |
| 5 | | 1216 | 20 | 答案正确 | 2 / 2 |
| 6 | | 1184 | 3988 | 答案正确 | 2 / 2 |
| 7 | | 1252 | 54 | 答案正确 | 2 / 2 |
| 8 | | 1128 | 2160 | 答案正确 | 6 / 6 |

## 冒泡排序

评测详情

| 测试点 | 提示 | 内存(KB) | 用时(ms) | 结果 | 得分 |
|---|---|---|---|---|---|
| 0 | | 324 | 2 | 答案正确 | 6 / 6 |
| 1 | | 196 | 2 | 答案正确 | 8 / 8 |
| 2 | | 352 | 3 | 答案正确 | 6 / 6 |
| 3 | | 308 | 137 | 答案正确 | 6 / 6 |
| 4 | | 688 | 10000 | 运行超时 | 0 / 2 |
| 5 | | 1180 | 4772 | 答案正确 | 2 / 2 |
| 6 | | 1212 | 7115 | 答案正确 | 2 / 2 |
| 7 | | 1248 | 4865 | 答案正确 | 2 / 2 |
| 8 | | 660 | 10000 | 运行超时 | 0 / 6 |

### 五、 心得体会

在这次实验中，我深刻体会到了不同排序算法的特性和适用场景。通过对冒泡排序、插入排序、快速排序和归并排序的编程实践，我不仅掌握了这些算法的基本原理和实现方法，还学会了如何根据数据的特点和规模选择合适的排序算法。例如，在处理小规模数据时，简单的冒泡或插入排序就非常高效；而面对大规模数据时，快速排序和归并排序展现出了它们的强大性能。这种算法与数据特性的匹配，让我对算法优化有了更深的理解。

实验过程中，我也意识到了算法效率对程序性能的影响。在处理大规模数据时，冒泡排序和插入排序的耗时远高于快速排序和归并排序，这让我明白在实际应用中选择合适的算法至关重要。同时，我也学会了如何分析算法的时间复杂度，这对于评估和优化程序性能提供了理论基础。通过对比不同排序算法在各种数据集上的表现，我更加清楚地认识到了算法设计的复杂性和挑战性。

这次实验也锻炼了我的编程能力和调试技巧。在编写和测试排序算法时，我遇到了一些逻辑错误和边界条件处理的问题。通过反复调试和修改，我不仅修正了错误，还提升了我的编程技巧。实验的过程虽然有时令人挫败，但每解决一个问题都让我感到巨大的成就感。通过这次实验，我更加坚信持续的实践和学习是提升编程技能的关键。