Chapter 4 快速排序

Section 1: 分治法(Divide and Conquer)(D&C)

D&C的工作原理:(D&C只是一种解决问题的思路,并非某种确定的算法)

(1). 找出简单的基线条件;

(2). 确定如何缩小问题的规模,使其符合基线条件.

Q: 什么是基线条件?

A:

> 编写递归函数时，必须告诉它何时停止递归。正因为如此，每个递归函数都有两部分：基线条件（base case）和递归条件（recursive case）。递归条件指的是函数调用自己，而基线条件则指的是函数不再调用自己，从而避免形成无限循环。

Detailed Explaination for Base & Recursive cases:

1. **基线条件（Base Case）**

作用：终止递归，防止无限递归调用。

2. **递归条件（Recursive Case）**

作用：将问题分解为更小的子问题，继续递归调用。

使用分治法和递归的思想解决实际问题

Practice 1: 数组求和

```
#include <iostream>
#include <vector>
using namespace std;

// 递归方式实现数组求和
template<typename T>
T sum(const vector<T>& arr, size_t index = 0)
{
    if (index >= arr.size()) //基线条件
    {
        return T(); // 返回类型T的默认值
    }
    return arr[index] + sum(arr, index + 1);//递归条件
}
//什么叫作类型T的默认值?
/*
e.g. double
        return 0.0;
```

```
    float
        return 0.0;
    int
        return 0;
*/
int main()
{
    int t;
    cout << "请输入测试样例数量: ";
    cin >> t;

    while (t--)
    {
        cout << "请输入测试数据类型: ";
        cout << "整数(I), 单精度浮点数(F), 双精度浮点数(D): ";
        char op;
        cin >> op;

        cout << "请输入数组元素个数: ";
        int n;
        cin >> n;

        if (op == 'I')
        {
            vector<int> arr(n);
            cout << "请输入" << n << "个整数: ";
            for (int i = 0; i < n; i++)
            {
                cin >> arr[i];
            }
            cout << "数组和为: " << sum(arr) << endl;
        }
        else if (op == 'F')
        {
            vector<float> arr(n);
            cout << "请输入" << n << "个单精度浮点数: ";
            for (int i = 0; i < n; i++)
            {
                cin >> arr[i];
            }
            cout << "数组和为: " << sum(arr) << endl;
        }
        else if (op == 'D')
        {
            vector<double> arr(n);
            cout << "请输入" << n << "个双精度浮点数: ";
            for (int i = 0; i < n; i++)
            {
                cin >> arr[i];
            }
            cout << "数组和为: " << sum(arr) << endl;
        }
        else
        {
```

```
            cout << "无效的输入类型!" << endl;
        }
    }

    return 0;
}
```

comment:

(1). 常见的数组求和方法是使用循环,这里不展开说明.

(2). 编写涉及数组的递归函数时，基线条件通常是数组为空或只包含一个元素。

(3). 函数式编程中没有循环,故我们只能使用递归来实现循环.(略)

Pracatice 2: 使用递归方式count数组大小

```
//Way1:Traditional Recurrence
#include <iostream>
#include <vector>
using namespace std;

// 递归计算数组大小
template<typename T>
size_t recursiveSize(const vector<T>& arr, size_t index = 0)
{
   if (index >= arr.size()) // 基线条件
   {
       return 0;
   }
   return 1 + recursiveSize(arr, index + 1);  // 递归条件
}

int main()
{
   //e.g.
   vector<int> nums = { 1, 2, 3, 4, 5 };
   cout << "数组大小: " << recursiveSize(nums) << endl;  // 输出5
   //注:此处的index为默认参数,系统自动迭代+1,故在传参时不需要手动传入.("nums" is
enough)
   return 0;
}
```

```
//Way2:数组切片思想(more functionalized)
template<typename T>
size_t recursiveSize(const vector<T>& arr)
{
   if (arr.empty()) // 基线条件
```

```
    {
        return 0;
    }
    vector<T> subArr(arr.begin() + 1, arr.end());  // 创建子数组
    return 1 + recursiveSize(subArr);              // 递归条件
}
```

Practice 3: 使用递归的思想寻找数组中最大的元素

```
int findMaxPure(const vector<int>& arr)
{
    // 基线条件1：空数组
    if (arr.empty())
        throw invalid_argument("空数组无最大值");
    // 基线条件2：单个元素
    if (arr.size() == 1)
        return arr[0];
    // 递归条件：比较首元素与子数组最大值
    int subMax = findMaxPure(vector<int>(arr.begin() + 1, arr.end()));
    return (arr[0] > subMax) ? arr[0] : subMax;
}
```

Practice 4: 二分查找(binary search 中的基线条件和递归条件)

```
#include <iostream>
#include <vector>
using namespace std;

int binarySearch(const vector<int>& arr, int target, int left, int right)
{
    if (left > right) // 基线条件1：未找到
        return -1;

    int mid = left + (right - left) / 2;

    if (arr[mid] == target) // 基线条件2：找到目标
    {
        return mid;
    }
    else if (arr[mid] < target)
    {
        return binarySearch(arr, target, mid + 1, right);  // 递归条件1
    }
    else
    {
        return binarySearch(arr, target, left, mid - 1);  // 递归条件2
    }
}
```

```
int main()
{
    vector<int> arr = {1, 3, 5, 7, 9};
    int target = 7;
    int result = binarySearch(arr, target, 0, arr.size() - 1);

    if (result != -1)
    {
        cout << "元素 " << target << " 在索引 " << result << " 处" << endl;
    } else {
        cout << "元素未找到" << endl;
    }
    return 0;
}
```

Section 2: Quick sort(qsort)快速排序:

C++代码实现

```
#include <iostream>
#include <vector>
using namespace std;

// 分区函数 - 核心操作
int partition(vector<int>& arr, int left, int right)
{
    int pivot = arr[right];  // 选择最右元素作为基准
    int i = left - 1;        // i标记小于pivot的区域的右边界

    for (int j = left; j < right; j++)
    {
        if (arr[j] < pivot)
        {
            i++;
            swap(arr[i], arr[j]);  // 将小于pivot的元素移到左侧
        }
    }
    swap(arr[i + 1], arr[right]);  // 将pivot放到正确位置
    return i + 1;                  // 返回pivot的最终索引
}

// 递归快速排序主函数
void quickSort(vector<int>& arr, int left, int right)
{
    if (left < right) // 基线条件：子数组长度>1
    {
        int pivotIndex = partition(arr, left, right);

        // 递归条件：分治处理左右子数组
        quickSort(arr, left, pivotIndex - 1);   // 排序左半部分
```

```
        quickSort(arr, pivotIndex + 1, right);  // 排序右半部分
    }
}

// 封装接口（隐藏边界参数）
void quickSort(vector<int>& arr)
{
    if (!arr.empty())
    {
        quickSort(arr, 0, arr.size() - 1);
    }
}

// 测试用例
int main()
{
    vector<int> arr = { 10, 7, 8, 9, 1, 5 };

    cout << "排序前: ";
    for (int num : arr) cout << num << " ";

    quickSort(arr);

    cout << "\n排序后: ";
    for (int num : arr) cout << num << " ";

    return 0;
}
```

python代码实现(much more easier to understand)

```
def quicksort(array):
    if len(array) < 2:
        return array
    else:
        pivot = array[0]  # 基线条件：为空或只包含一个元素的数组是“有序”的
        less = [i for i in array[1:] if i <= pivot] #递归条件由所有小于基准值的元素组成的子数组
        greater = [i for i in array[1:] if i > pivot] #由所有大于基准值的元素组成的子数组

    return quicksort(less) + [pivot] + quicksort(greater)
# e.g.
# print (quicksort([10, 5, 2, 3]))
```