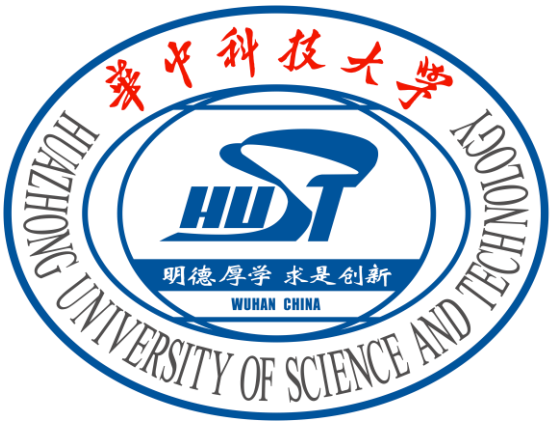

华中科技大学

HUAZHONG UNIVERSITY SCIENCE AND TECHNOLOGY

课 程 实 验 报 告

（实验一）

课程名称 ： 算法分析与设计

院　　系 ： 新闻与信息传播学院

专业班级 ： 传播学 2201 班

学　　号 ： U202217034

姓　　名 ： 余易昕

指导教师 ： 王多强

2025 年 5 月 23 日

实验一：最大子数组问题

https://github.com/sakaaanaYu/CS_Experiments/blob/main/algorithm_learning/code/exp_1.ipynb

1. 需求分析

本次实验需要完成以下内容：

1. 分别使用分治算法和非递归线性时间算法实现最大子数组问题的求解，实现两个算法的功能函数；
2. 将两种算法集成在一个程序中，实现编号选择调用；
3. 对于存在多种最大子数组的情况，分别实现返回一个结果和返回全部结果；
4. 对比两种算法在计算时间上的差异

2. 源码及说明

a）分治算法

对于求解数组 A[low, …, high]的最大子数组，分治算法的求解思路为：

1. 找到找到 A 的 mid
2. 求解 A[low..mid]和 A[mid+1..high]的最大子数组
3. 找到跨越中点的最大子数组
4. 取和最大者

代码实现如下。

【输出一个结果】

```
# 跨越中点的最大子数组
def FIND_MAX_CROSSSING_SUBARRAY(A, low, mid, high):
    left_sum = float('-inf')
    sum = 0
    max_left = mid  # 初始化 max_left
    for i in range(mid, low-1, -1):
        sum += A[i]
        if sum > left_sum:
            left_sum = sum
            max_left = i
    right_sum = float('-inf')
    sum = 0
    max_right = mid + 1  # 初始化 max_right
    for j in range(mid+1, high+1):
        sum += A[j]
        if sum > right_sum:
```

```
            right_sum = sum
            max_right = j
    return (max_left, max_right, left_sum + right_sum)

# 找最大子数组
def FIND_MAX_SUBARRAY(A, low, high):
    if high == low:
        return (low, high, A[low])
    else:
        mid = (low + high) // 2  # 使用整除
        (left_low, left_high, left_sum) = FIND_MAX_SUBARRAY(A, low, mid)
        (right_low, right_high, right_sum) = FIND_MAX_SUBARRAY(A, mid+1, high)
        (cross_low, cross_high, cross_sum) = FIND_MAX_CROSSSING_SUBARRAY(A, low,
mid, high)
        if left_sum >= right_sum and left_sum >= cross_sum:
            return (left_low, left_high, left_sum)
        elif right_sum >= left_sum and right_sum >= cross_sum:
            return (right_low, right_high, right_sum)
        else:
            return (cross_low, cross_high, cross_sum)
```

【输出多个结果】

```
def FIND_MAX_CROSSSING_SUBARRAY_NEW(A, low, mid, high):
    # 向左扫描
    left_sum = float('-inf')
    sum = 0
    left_sums = []

    for i in range(mid, low-1, -1):
        sum += A[i]
        if sum > left_sum:
            left_sum = sum
            left_sums = [(i, sum)]
        elif sum == left_sum:
            left_sums.append((i, sum))

    # 向右扫描
    right_sum = float('-inf')
    sum = 0
    right_sums = []

    for j in range(mid+1, high+1):
        sum += A[j]
        if sum > right_sum:
```

```
            right_sum = sum
            right_sums = [(j, sum)]
        elif sum == right_sum:
                right_sums.append((j, sum))
    # 找出所有可能的组合
    results = []
    max_cross_sum = float('-inf')

    # 计算所有可能的组合和值
    for left_idx, left_val in left_sums:
        for right_idx, right_val in right_sums:
            cross_sum = left_val + right_val
            if cross_sum > max_cross_sum:
                max_cross_sum = cross_sum
                results = [(left_idx, right_idx, cross_sum)]
            elif cross_sum == max_cross_sum:
                results.append((left_idx, right_idx, cross_sum))

    return results

def FIND_MAX_SUBARRAY_NEW(A, low, high):
    if high == low:
        return [(low, high, A[low])]  # 返回列表而不是元组

    mid = (low + high) // 2
    left_results = FIND_MAX_SUBARRAY_NEW(A, low, mid)
    right_results = FIND_MAX_SUBARRAY_NEW(A, mid+1, high)
    cross_results = FIND_MAX_CROSSSING_SUBARRAY_NEW(A, low, mid, high)

    # 合并所有结果
    all_results = left_results + right_results + cross_results

    # 找出最大和值
    max_sum = max(result[2] for result in all_results)

    # 返回所有和值等于最大和值的结果
        return [result for result in all_results if result[2] == max_sum]
```

b）非递归线性时间算法

【输出一个结果】

```
def FIND_MAX_SUBARRAY_LINEAR(A):
    max_sum = float('-inf')
    current_sum = 0
```

```
    start = 0
    max_start = 0
    max_end = 0

    for i in range(len(A)):
        current_sum += A[i]
        if current_sum > max_sum:
            max_sum = current_sum
            max_start = start
            max_end = i
        if current_sum < 0:
            current_sum = 0
            start = i + 1

    return (max_start, max_end, max_sum)
```

【输出多个结果】

```
def FIND_MAX_SUBARRAY_LINEAR_NEW(A):
    max_sum = float('-inf')
    current_sum = 0
    start = 0
    results = []  # 存储所有最大子数组

    for i in range(len(A)):
        current_sum += A[i]
        if current_sum > max_sum:
            max_sum = current_sum
            results = [(start, i, current_sum)]  # 重置为新的最大值
        elif current_sum == max_sum:
            results.append((start, i, current_sum))  # 添加相同和值的结果
        if current_sum < 0:
            current_sum = 0
            start = i + 1

    return results
```

c）编号选择调用

```
def menu():
    print("请选择要使用的算法：")
    print("1. 分治法")
    print("2. 线性法")
    print("0. 退出")
    choice = input("请选择要使用的算法：")
    return choice
```

d）主函数

```
def read_dataset_from_file(file_path):
    with open(file_path, 'r') as file:
        data = list(map(int, file.read().strip().split()))
    n = data[0]
    if n == 0:
        return []  # 如果 n 为 0，返回空数组
    return data[1:]  # 返回后续的 n 个整数


def main_with_menu(file_path):
    A = read_dataset_from_file(file_path)

    if not A:
        print("数据集为空或无效。")
        return

    n = len(A)

    while True:
        choice = menu()

        if choice == '1':
            # 使用分治法
            start_time = time.time()
            low, high, max_sum = FIND_MAX_SUBARRAY(A, 0, n-1)
            end_time = time.time()
            if n == 0:
                print(0)
            else:
                print(f"{max_sum} {low} {high} {n}")

        elif choice == '2':
            # 使用线性法
            start_time = time.time()
            low, high, max_sum = FIND_MAX_SUBARRAY_LINEAR(A)
            end_time = time.time()
            if n == 0:
                print(0)
            else:
                print(f"{max_sum} {low} {high} {n}")

        elif choice == '0':
            print("程序已退出。")
```

```
            break

        else:
            print("无效的选择，请重新输入。")
```

e）时间对比

```
    # 测试分治法
    start_time = time.time()
    divide_results = FIND_MAX_SUBARRAY_NEW(A, 0, n-1)
    end_time = time.time()
    divide_time = end_time - start_time

    # 测试线性法
    start_time = time.time()
    linear_results = FIND_MAX_SUBARRAY_LINEAR_NEW(A)
    end_time = time.time()
    linear_time = end_time - start_time

    # 输出对比结果
    print(f"\n数据集大小: {n}")
    print(f"分治法执行时间: {divide_time:.6f} 秒")
    print(f"线性法执行时间: {linear_time:.6f} 秒")
    print(f"时间差异: {abs(divide_time - linear_time):.6f} 秒")
    if linear_time < 1e-6:  # 如果线性法执行时间小于1微秒
        print("线性法执行时间过短，无法计算准确的时间比例")
    else:
        print(f"分治法/线性法时间比: {divide_time/linear_time:.2f}倍")
```

3. 代码测试

【输出一个结果】

第一次的是分治法，第二次的二是线性法。

```
请选择要使用的算法:
1. 分治法
2. 线性法
0. 退出
78 7 26 38
请选择要使用的算法:
1. 分治法
2. 线性法
0. 退出
78 7 26 38
请选择要使用的算法:
1. 分治法
2. 线性法
0. 退出
程序已退出。
```

【输出多个结果】

第一次的是分治法，第二次的二是线性法。

```
请选择要使用的算法:
1. 分治法
2. 线性法
0. 退出
找到 2 个最大子数组:
3 0 1 5
3 3 4 5
请选择要使用的算法:
1. 分治法
2. 线性法
0. 退出
找到 2 个最大子数组:
3 0 1 5
3 3 4 5
请选择要使用的算法:
1. 分治法
2. 线性法
0. 退出
程序已退出。
```

【时间对比】

```
请选择要使用的算法：
1. 分治法
2. 线性法
3. 对比两种算法
0. 退出

开始算法性能对比...

数据集大小: 1000
分治法执行时间: 0.007381 秒
线性法执行时间: 0.000000 秒
时间差异: 0.007381 秒
线性法执行时间过短，无法计算准确的时间比例

请选择要使用的算法：
1. 分治法
2. 线性法
3. 对比两种算法
0. 退出
程序已退出。
```

线性法在计算时间上明显优于分治法。