《数据结构》课程实践报告

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 院、系 | 计算机学院 | | 年级专业 | 23软工 | 姓名 | 王天予 | 学号 | 2362401031 |
| 实验布置日期 | | 2024.9.10 | | 提交  日期 |  | | 成绩 |  |

课程实践实验1-2

## 一、问题描述及要求

**1.** 假设S是整数序列 A0， A1，... An-1 （可能有负数），求 A0 ~ An-1 的一个子序列 Ai ~ Aj， 使得 Ai到 Aj 的和最大。

2. 比较递归算法和等效迭代算法的数据范围限制和运行时间

## 二、概要设计

**1.**尝试了3种不同的算法，并进行了进一步改进**。**

——————————————————————

ADT 连续子序列最大和问题 ——————————————————————

对象：

序列 s：存储整数的数组

整数 n：序列的长度

整数 max\_sum：当前最大子序列和

整数 start, finish：最大子序列的起始和结束位置

操作：

1. 初始化(s, n)：初始化序列 s 和其长度 n。
2. 应用算法1
3. 应用算法2
4. 应用算法3
5. 应用算法4

2.为了统计递归和迭代两种算法的运行时间，我们采用了chrono库中的高精度时间统计工具，利用终止时间减去初始时间的方式计算运行时间。

## 三、详细设计

为了在O(n)的时间复杂度下处解决问题并输出最大和子段起止点的编号，我们采用了基于前缀和的算法；改进后问题1的伪代码如下：

——————————————————————

算法4：连续子序列最大和问题

——————————————————————

输入：序列s，s.array[i]∈整数集

输出：序列s中最大的连续子序列之和max\_sum

1. s.max\_sum ← 0 //设置最大子序列和初值

2. for i←1 to s.n do //子序列起始位置

3. | for j←i to s.n do //子序列结束位置

4. | | this\_sum ← 0 //设置子序列和初值

5. | | for k←i to j do //求子序列和

6. | | | this\_sum ← this\_sum+s.array[k]

7. | | end

8. | | if this\_sum > s.max\_sum then //如当前子序列和更大

9. | | | s.max\_sum ← this\_sum //设置当前最大子序列和

10. | | | s.start ← i //设置当前最大子序列起始位置

11. | | | s.finish ← j //设置当前最大子序列结束位置

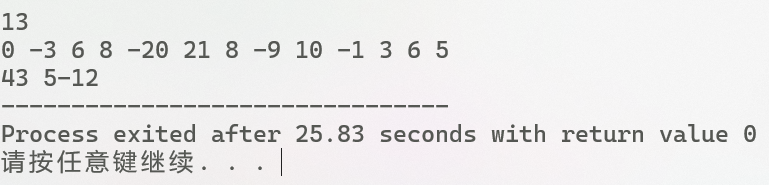
12. | | end

13. | end

14. end

## 四、实验结果

**1. 问题1**

****

**2. 问题2**

当n=43000时，测试程序有输出：

Average Time taken by RecursivePrint: 3363.9 ms

Average Time taken by IterativePrint: 3161.7 ms

## 五、实验分析与探讨

**1. 问题1**

(1) 请先编写程序验证这3个实现算法是否正确

答：均正确。

(2) 用事前分析法分析这3个算法的时间复杂度

答：分别为O(n^3)、O(n^2)、O(n)

(3) 生成随机数据并测试

答：经测试，3个算法的运行时间基本与估量的时间数量级一致。由于测试数据较大，此处不作摘录。

(4) 最后请给出你对这3个算法的分析结论

答：三个算法的空间复杂度相仿，算法3的时间复杂度最优秀。

(5) 改进后的算法4：

经测试，该算法可以在O(n)复杂度下正确实现区间的输出。

**2. 问题2**

(1) 测试当n达到多少时，算法2-1会崩溃，这时算法2-2工作正常吗？

答：对于算法2-1，n≤43000时可以正常工作；当n=43000时算法2-2可以正常工作。

(2) 当n取一个适当值时，请计算程序实际运行时间，看看这个结论是否正确。

答：当n=43000时，测试程序有输出：

Average Time taken by RecursivePrint: 3363.9 ms

Average Time taken by IterativePrint: 3161.7 ms

故结论是正确的。

## 六、小结

在本次实验中，对于问题1，通过实现和比较不同算法，解决了连续子序列最大和问题，并回答了相关问题。在此过程中，采用了递归和迭代两种方式来实现算法，并对其进行了详细的性能分析。通过引入基于前缀和的优化算法，成功将问题的时间复杂度降低到了O(n)，显著提高了效率。

对于问题2，实验结果表明，尽管递归算法在小规模数据下表现较好，但随着数据规模的增加，迭代算法展现了更为稳定的性能表现。最终测试结果也验证了对时间复杂度和运行时间的预估，充分体现了不同算法的优劣和适用场景。

本实验不仅加强了我对递归和迭代算法的理解，也使我学会了如何通过算法优化解决复杂问题，为今后的编程实践提供了宝贵的经验。

## 附录：源代码

1. **实验环境**：

编译环境：TDM-GCC 4.9.2 64bit release

命令行参数：-static-libgcc -std=c++11

**2、目录结构**

（1）Problem1.cpp

#include <bits/stdc++.h>

using namespace std;

const int maxn = 10001;

int n,s[maxn],min\_v,ans = 0;

pair<int,int> aii;

int main(){

cin>>n;

for (int i=1,t;i<=n;i++){

cin>>t;

//Calculate the prefix sum

s[i] = s[i-1] + t;

}

min\_v = 1;

for (int i=1,t=1;i<=n;i++){

//Calculate min value of each index

if (s[i] < s[min\_v])

min\_v = i;

if (s[i] - s[min\_v] > ans){

ans = s[i] - s[min\_v];

aii = make\_pair(min\_v+1,i);

}

}

//-1 since our array index begins at 0

cout<<ans<<' '<<aii.first-1<<'-'<<aii.second-1;

return 0;

}

（2）Problem2.cpp

#include <bits/stdc++.h>

#include <chrono>

using namespace std;

// Recursive Algorithm

void RecursivePrint(int n) {

if (n > 0) {

RecursivePrint(n - 1);

cout << n << " ";

}

}

// Iterative Algorithm

inline void IterativePrint(int n) {

for (register int i = 1; i <= n; ++i) {

cout << i << " ";

}

}

int main() {

int n;

int ave\_recursive\_duration = 0, ave\_iterative\_duration = 0;

cout << "Enter the value of n: ";

cin >> n;

for (int i=1;i<=10;i++){

cout<<"Running Test #"<<i<<":\n";

// Measure time for IterativePrint

auto start\_iterative = chrono::high\_resolution\_clock::now();

IterativePrint(n);

auto end\_iterative = chrono::high\_resolution\_clock::now();

cout << "\nIterativePrint completed.\n";

// Measure time for RecursivePrint

auto start\_recursive = chrono::high\_resolution\_clock::now();

RecursivePrint(n);

auto end\_recursive = chrono::high\_resolution\_clock::now();

cout << "\nRecursivePrint completed.\n";

// Calculate the elapsed time for both approaches

auto recursive\_duration = chrono::duration\_cast<chrono::milliseconds>(end\_recursive - start\_recursive).count();

auto iterative\_duration = chrono::duration\_cast<chrono::milliseconds>(end\_iterative - start\_iterative).count();

// Display time taken by both approaches

cout << "Time taken by RecursivePrint: " << recursive\_duration << " ms" << endl;

cout << "Time taken by IterativePrint: " << iterative\_duration << " ms" << endl;

ave\_recursive\_duration += recursive\_duration;

ave\_iterative\_duration += iterative\_duration;

}

cout << "Average Time taken by RecursivePrint: " << ave\_recursive\_duration/10.0 << " ms" << endl;

cout << "Average Time taken by IterativePrint: " << ave\_iterative\_duration/10.0 << " ms" << endl;

return 0;

}