**一，实验内容**

用C/C++分别实现求解最大子数组问题的蛮力算法和分治算法。随机生成规模分别为5000，10000，100000，1000000的数组并比较两个算法对于上述实例运行的时间开销。

**二，实验目的**

我们通过这次实验可以更清晰地了解到相对于蛮力算法，分治算法的方便快捷之处，使得我们更好地认识到随着输入实例的规模不断扩大，相对于蛮力算法，快速算法的时间复杂性的变化情况。

**三，实验过程**

#include<iostream>

#include<vector>

#include<cstdlib>

#include<ctime>

#include<limits.h>

#define N 10000

using namespace std;

struct MAXSUB{

    int low;

    int high;

    long sum;

};

//蛮力算法

MAXSUB EXHAUSTIONMAXSUBArray(const vector<int>& R){

    MAXSUB m = { 0 ,0, R[0]};

    int n = R.size();

    for (int i=0;i<n;i++){

        long tempsum = 0;

        for(int j = i;j<n;j++){

            tempsum+=R[j];

            if (tempsum > m.sum){

                m.sum = tempsum;

                m.low = i;

                m.high = j;

            }

        }

    }

    return m;

}

//归并法

MAXSUB FINDMAXCROSSINGSUBARRAY(const vector <int>& R,int low,int mid,int high){

    long leftsum = R[mid];

    long sum = leftsum;

    int maxleft = mid;

    for(int i = mid - 1;i>=low;i--){

        sum +=R[i];

        if(sum > leftsum){

            leftsum = sum;

            maxleft = i;

        }

    }

    long rightsum = R[mid+1];

    sum = rightsum;

    int maxright = mid+1;

    for(int i = mid+2;i<=high;i++){

        sum+=R[i];

        if(sum > rightsum ){

            rightsum = sum;

            maxright =i;

        }

    }

    MAXSUB m;

    m.low = maxleft;

    m.high = maxright;

    m.sum = leftsum+rightsum;

    return m;

}

//进行合并

MAXSUB FINDMAXIMUMSUBARRAY(const vector<int>&R,int low,int high){

    if(high == low){

        return {low,high,R[low]};

    }

    const int THRESHOLD = 50;

    if (high -low +1<=THRESHOLD){

        MAXSUB m=EXHAUSTIONMAXSUBArray(vector<int> (R.begin()+low,R.begin()+high+1));

        m.low +=low;

        m.high+=low;

        return m;

    }

    int mid = (low+high)/2;

    MAXSUB mleft = FINDMAXIMUMSUBARRAY(R,low,mid);

    MAXSUB mright=FINDMAXIMUMSUBARRAY(R,mid+1,high);

    MAXSUB mcross = FINDMAXCROSSINGSUBARRAY(R,low,mid ,high);

    return (mleft.sum>=mright.sum&&mleft.sum>=mcross.sum)?mleft:(mright.sum>=mcross.sum)?mright:mcross;

}

//动态规划

MAXSUB DYNAMICMAXIMUMSUBARRAY(const vector<int>& R){

    MAXSUB m={0,0,R[0]};

    long currentsum = R[0];

    int currentlow= 0,currenthigh=0;

    for(int i =1;i<R.size();i++){

        if (R[i]>currentsum+R[i]){

            currentlow = currenthigh = i;

            currentsum = R[i];

        }else {

            currenthigh = i;

            currentsum += R[i];

        }if (currentsum > m.sum) {

            m.sum = currentsum;

            m.low = currentlow;

            m.high = currenthigh;

        }

    }

    return m;

}

int main() {

    vector<int> R(N);

    MAXSUB m;

    srand(static\_cast<unsigned int>(time(0)));

    for (int i = 0; i < N; i++)

        R[i] = rand() % 2000 - 1000;

    // 蛮力搜索

    clock\_t time\_start = clock();

    m = EXHAUSTIONMAXSUBArray(R);

    clock\_t time\_end = clock();

    cout << "蛮力搜索用时：" << (time\_end - time\_start) \* 1000.0 / CLOCKS\_PER\_SEC << " 毫秒\n";

    cout << "最大子数组范围：[" << m.low << ", " << m.high << "]，和：" << m.sum << "\n\n";

    // 分治法

    time\_start = clock();

    m = FINDMAXIMUMSUBARRAY(R, 0, N - 1);

    time\_end = clock();

    cout << "分治法用时：" << (time\_end - time\_start) \* 1000.0 / CLOCKS\_PER\_SEC << " 毫秒\n";

    cout << "最大子数组范围：[" << m.low << ", " << m.high << "]，和：" << m.sum << "\n\n";

    // 动态规划

    time\_start = clock();

    m = DYNAMICMAXIMUMSUBARRAY(R);

    time\_end = clock();

    cout << "动态规划用时：" << (time\_end - time\_start) \* 1000.0 / CLOCKS\_PER\_SEC << " 毫秒\n";

    cout << "最大子数组范围：[" << m.low << ", " << m.high << "]，和：" << m.sum << "\n\n";

    return 0;

}

**四，实验环境**

计算机硬件情况：

电脑截图

AI 生成的内容可能不正确。

使用的软件：

VS Code 1.97.2 版本。

**五，实验结果**

当数据的规模 N = 5000 时，运行程序得到结果：

文本, 信件

AI 生成的内容可能不正确。

当数据的规模 N = 10000时，运行程序得到结果：

文本

AI 生成的内容可能不正确。

当数据的规模 N = 100000时，运行程序得到结果：

文本, 信件

AI 生成的内容可能不正确。

当数据的规模 N = 1000000时，运行程序得到结果：



**六，实验结果说明**

通过实验结果，以及所给的表格之中，我们可以很直观地看到，当N的数量级并没有那么高的时候，三种方法地运行时间并没有十分大的差距。但是当数量级越变越大的过程之中，我们可以很明显地感觉到，蛮力算法虽然源代码好写，但是他运行百万数量级别所花费的时间是难以估量的。相比而言，分治递归，以及动态规划的方法，极大地改善了这一情况，花费时间是非常的简短。

图形用户界面, 应用程序

AI 生成的内容可能不正确。

图1 不同规模数据下三种算法花费时间

**七，实验总结**

通过这次实验，我们可以很清晰明白地看出不同地算法在处理同一个数量级问题时候，所花费的时间是及其地不平衡的。因此，在实验或者是后续解决问题的过程之中，并不是对于一个问题，我能够得出它的能解决这个问题的算法，就行的，而是要充分地考虑到，在极大的数量级之下，算法是否同样可以发挥其简便快速的作用，提醒着我们不仅要掌握代码，更要学会比较，找出更加优秀的代码。