算法设计与分析实验1

计科2205 刘志垚 202208010512

1.分治法查找最大最小值

问题描述

利用分治法查找数组元素的最大值和最小值，并计算出程序运行所需要的时间。

思路

分治法的核心就是「分而治之」，大概的流程即：分解->解决->合并。先将原问题分解为结构相同的子问题，接着根据某个边界条件对子问题进行递归求解，最终合并子问题的解。本题可以将数组分成两个部分，递归地在这两个部分中查找最大值和最小值。在合并结果时，比较两个部分的最大值和最小值，得到全局的最大值和最小值。

算法步骤

如果数组只有一个元素，返回该元素作为最大值和最小值。

如果数组有两个元素，比较这两个元素，返回较大的作为最大值，较小的作为最小值。

如果数组有三个或更多元素，进行如下操作：

1. 将数组分成两个子数组。
2. 递归调用该方法，分别计算左子数组的最大值和最小值，右子数组的最大值和最小值。
3. 比较左侧和右侧的最大值与最小值，得出全局的最大值和最小值。

代码

```
//分治法查找最大最小值
#include <bits/stdc++.h>
#define endl '\n'
typedef long long ll;
using namespace std;
pair<ll, ll> dcMaxMin(const vector<ll>& arr, ll low, ll high) { //const传递常数引用，避免拷贝开销
    if(low == high) return {arr[low], arr[high]};

    if(low + 1 == high) {
        if(arr[low] > arr[high])
            return {arr[low], arr[high]};
        else
            return {arr[high], arr[low]};
    }

    ll mid = (low + high) / 2;

    //递归查找左右子数组的最大最小值
    auto leftMaxMin = dcMaxMin(arr, low, mid);
    auto rightMaxMin = dcMaxMin(arr, mid + 1, high);
```

```
    return {max(leftMaxMin.first, rightMaxMin.first), min(leftMaxMin.second,
rightMaxMin.second)};
}
int main() {
    ifstream ifs("testData/small_input1.txt");
    ofstream ofs("output1.txt");
    vector <ll> arr;
    ll number;
    while (ifs >> number) { // 逐行读取整数
        arr.push_back(number); // 将读取的数字推入向量
    }
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    auto[MaxValue, MinValue] = dcMaxMin(arr, 0, arr.size() - 1);
    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    ofs << "time: " << duration.count() << " ms" << endl;
    ofs << "Max:" << MaxValue << ' ' << "Min:" << MinValue;
    ifs.close();
    ofs.close();
    return 0;
}
```

测试结果

小规模：10个数据，范围1~100

```
time: 0.0008 ms
Max:99 Min:0
```

中规模：1万个数据，范围1~2141425

```
time: 0.1224 ms
Max:2141398 Min:120
```

大规模：100万个数据，范围1~2147483600

```
time: 12.4158 ms
Max:1073738451 Min:563
```

算法分析

时间复杂度：$O(n)$，因为每个元素都被访问一次。

空间复杂度：$O(\log n)$，由于递归调用栈的深度。

2.分治法实现归并排序

问题描述

利用用分治法实现合并排序，并计算出程序运行所需要的时间。

思路

将待排序数组分成大小大致相同的2个子数组，分别对2个子数组进行排序，最终将排好序的子数组合并成为所要求的排好序的数组。`先自上而下分解问题，后自下而上解决问题`。

算法步骤

1. 如果数组的长度小于或等于 1，直接返回数组。
2. 找到数组的中间索引，将数组分成左右两部分。
3. 递归地对左半部分和右半部分进行合并排序。
4. 合并两个已排序的部分。

代码

```
//分治法实现合并排序
#include <bits/stdc++.h>
typedef long long ll;
const int N = 8;
using namespace std;
void merge(vector<ll>& arr, ll left, ll mid, ll right) {
    ll n1 = mid - left + 1; //左数组大小
    ll n2 = right - mid;    //右数组大小

    vector<ll> L(n1), R(n2); //创建左右临时数组
    // 拷贝数据到 L[] 和 R[]
    for(int i = 0; i < n1; ++i)L[i] = arr[left + i];
    for(int j = 0; j < n2; ++j)R[j] = arr[mid + 1 + j];

    //合并数组
    ll i = 0, j = 0, k = left;
    while(i < n1 && j < n2) {
        if(L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    //拷贝剩余元素
    while(i < n1) arr[k++] = L[i++];
    while(j < n2) arr[k++] = R[j++];
}
void mergeSort(vector<ll>& arr, ll left, ll right) {
    if(left < right) {
        ll mid = left + (right - left) / 2; //防止溢出
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
int main() {
```

```
    ifstream ifs("testData/large_input2.txt");
    ofstream ofs("output2.txt");
    vector <ll> arr;
    ll number;
    while (ifs >> number) { // 逐行读取整数
        arr.push_back(number); // 将读取的数字推入向量
    }
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    mergeSort(arr, 0, arr.size() - 1);
    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    ofs << "time: " << duration.count() << " ms" << endl;
    for(auto y : arr) {
        ofs << y << ' ' ;
    }
    ifs.close();
    ofs.close();
    return 0;
}
```

测试结果

小规模：10个数据，范围1~100

```
time: 0.0107 ms
0 1 22 33 34 40 49 49 59 62 64 70 75 76 77 78 82 85 95 99
```

中规模：1万个数据，范围1~2141425

```
time: 3.0158 ms
52 127 384 498 524 747 812 956 1128 1157 1184 1443 1758 1895 2188 2300 2501 2732 2779 3005 3101 3436 3622 3841 3860
3876 3974 4046 4772 4924 5012 5040 5138 5202 5737 5808 5978 6497 6883 6891 6966 7099 7533 7558 7618 7675 7734 7866
8022 8244 8251 8277 8320 8391 8480 8509 9343 9366 9920 10496 10747 11103 11164 11361 11498 12250 12326 12404 12501
12531 13461 13741 13953 14086 14266 14302 15037 15168 15669 15738 16285 16450 16538 16623 16943 17689 17699 18166
18424 18876 18994 19134 19236 19298 19575 19680 19814 19939 20175 20192 20249 21027 21154 21323 21556 21890 22184
22817 22877 22941 23202 23220 23366 23625 24060 24148 24362 24395 24672 25024 25260 25502 25549 25590 25989 26815
27039 27277 27283 27531 28004 28037 28441 28511 28536 28690 28783 29143 29915 30213 30582 30847 31185 31382 32186
32302 32327 32645 32654 32706 33024 33041 33079 33173 33632 34036 34145 34275 34334 34574 34970 35068 35777 35849
36434 37535 37582 37891 38289 38298 38333 38787 39525 39692 39773 39822 39851 40358 40706 41034 41309 41402 41447
41578 41744 41778 41903 42155 42187 42224 42467 42750 42805 42977 43163 43195 43234 43700 43746 43922 44125 44474
44607 44639 44809 45002 45802 46140 47226 47299 47582 47611 47668 47710 47844 47847 47852 47889 47964 48038 48398
48500 48698 49202 49393 49925 50215 50294 50433 50482 51474 51831 51895 51968 52167 52342 52585 53146 53254 53394
53638 53647 53697 54169 54451 54770 54903 56135 56144 56204 56810 56906 57148 57812 57974 58200 58206 58358 58437
58809 58882 58893 59031 59298 59368 59476 59617 59823 59856 60579 60906 61021 61105 61106 61155 61157 61216 61533
61649 61657 61957 62136 62236 62400 62610 62684 62691 62713 62737 62738 63198 63258 63294 63554 63585 63894 64410
64629 64853 64960 65033 65037 65437 65480 65659 65672 65861 66028 66093 67619 67659 67898 68148 68517 68565 68815
```

大规模：1000万个数据，范围1~2147483600。

```
time: 367.544 ms
1708 1874 2357 3372 4366 5163 5541 6731 8038 10598 10868 12238 12945 13493 13858 13897
15681 16023 16919 17862 19146 19210 21281 22154 24261 24542 25742 26303 26657 27083
28416 28540 28691 29172 29325 29341 29885 30462 31293 32430 33966 35500 35930 37372
38291 40055 40248 43361 45227 45618 46103 47020 47656 47782 48136 48984 52883 53465
53810 54950 55443 55797 56030 56093 57752 59700 62023 62770 64655 65650 65712 68759
70194 70610 70611 71347 71682 72182 73222 73682 74166 74272 75579 76338 77031 77302
80762 82413 82638 83112 83148 83186 83290 85587 86987 87121 87421 88071 90343 91106
91921 91943 92076 92677 93684 94100 94560 96415 100686 100819 101267 101322 102293
102394 102703 105741 107532 107760 107884 108746 109193 110290 110384 110702 110714
110721 110987 111840 113751 113944 115337 116132 116906 117019 117951 118755 118973
119284 119379 121437 121814 122369 122608 123038 124198 124505 125842 128352 130274
130545 131286 131917 135015 138864 141658 143143 143529 144425 144477 149575 150048
150743 150801 151562 153422 154530 154893 156883 158679 159445 160685 161552 163187
```

算法分析

时间复杂度：$O(n\log n)$，因为每次合并需要 $O(n)$的时间，而递归深度为 $\log n$。

空间复杂度：$O(n)$，用于临时数组。

3.实现题1-3 最大约数问题

问题描述

正整数 x 的约数是能整除 x 的正整数。正整数 x 的约数个数记为 div(x)。例如1、2、5、10 都是正整数 10 的约数，且 div(10)=4。设 a 和b 是 2 个正整数，a≤b，找出 a 和 b 之间约数个数最多的数 x 及其最多约数个数 div(x)。

Ⅰ.方法1（暴力求解）

思路

遍历区间 $[a,b]$ 内所有数，对每个数 n 通过遍历 1 到 $\sqrt{n}$ 的所有整数求其约数。

算法步骤

1. 遍历区间$[a,b]$内的所有数。
2. 对每个数计算其约数个数。对于每一个数 N，其约数个数的计算方法是遍历从 1 到 $\sqrt{N}$ 的所有整数，对于每一个能够整除 N 的数 i，如果 $i\times i=N$，则只计入一个约数；否则，计入两个（i 和 N/i 都是 N 的约数）。
3. 找到约数个数最多的数，如果有多个，返回最小的那个数。

代码

```cpp
#include <iostream>
#include <cmath>
#include <bits/stdc++.h>
using namespace std;
int a, b;
int sum = 0; // 最大的约数个数
int number = 0; // 对应的数
// 计算某个数 n 的约数个数
int div(int n) {
    int count = 0;
    for (int i = 1; i <= sqrt(n); ++i) {
        if (n % i == 0) {
            if (i == n / i) // 如果 i 是 n 的平方根，只计算一次
```

```
                count++;
            else // 否则，i 和 n/i 都是约数
                count += 2;
        }
    }
    return count;
}

int main() {
    ifstream ifs("testData/large_input3.txt");
    ofstream ofs("output3.txt");
    ifs >> a >> b;
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    // 遍历区间 [a, b]
    for (int i = a; i <= b; ++i) {
        int k = div(i); // 计算 i 的约数个数
        // 更新最大约数数及其对应的数
        if (k > sum) {
            sum = k;
            number = i;
        } else if (k == sum && i < number) {
            number = i;
        }
    }

    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    // 输出结果
    ofs << "time: " << duration.count() << " ms" << endl;
    ofs << "x: " << number << " div(x): " << sum << endl;
    ifs.close();
    ofs.close();
    return 0;
}
```

测试结果

小规模：[1, 1000]

```
time: 0.1234 ms
x: 840 div(x): 32
```

中规模：[1, 100000]

```
time: 70.1601 ms
x: 83160 div(x): 128
```

大规模：[1, 10000000]

```
time: 66992.3 ms
x: 8648640 div(x): 448
```

算法分析

每次计算约数的时间复杂度为 $O(\sqrt{i})$，最坏情况下为 $O(\sqrt{b})$，所以总的时间复杂度为

$$O(N \cdot \sqrt{b})$$

N为区间长度，$N = b - a + 1$。该算法的效率会随着区间长度和区间上界的增大而显著降低。

II.方法2（筛）

思路

可以使用`筛法`来优化计算每个数的约数个数。筛法类似于埃氏筛算法，它的核心思想是遍历所有数时，直接标记其`倍数的约数个数`，避免重复计算。

算法步骤

1. 筛法预处理：对于每一个数 i，将它作为其所有倍数的约数。因此，我们可以从 1 遍历到 b，`然后对每个 $i$ 的倍数 $k$，将 $k$ 的约数个数加 1`。这样，我们可以避免逐个数去计算约数个数，大大提升了效率。
2. 区间查询：我们需要在 $[a, b]$ 区间内找到约数个数最多的数。这意味着我们可以先筛出 1 到 b 的所有约数个数，然后再根据 a 和 b 的范围取出相应的结果。

代码

```
#include <iostream>
#include <cmath>
#include <bits/stdc++.h>
using namespace std;
#define endl '\n'
int a, b;
int sum = 0; // 最大的约数个数
int number = 0; // 对应的数
int main() {
    ifstream ifs("testData/large_input3.txt");
    ofstream ofs("output3.txt");
    ifs >> a >> b;
    vector<int> divi(b + 1, 0);
    // 开始计时
    auto start = chrono::high_resolution_clock::now();
    for(int i = 1; i <= b; ++i)
        for(int j = i; j <= b; j += i)
            divi[j]++;
    // 遍历区间 [a, b]
    for (int i = a; i <= b; ++i) {
        int k = divi[i]; // 计算 i 的约数个数
        // 更新最大约数数及其对应的数
        if (k > sum) {
            sum = k;
            number = i;
        } else if (k == sum && i < number) {
            number = i;
        }
    }
    // 结束计时
    auto end = chrono::high_resolution_clock::now();
```

```
    chrono::duration<double, std::milli> duration = end - start;
    // 输出结果
    ofs << "time: " << duration.count() << " ms" << endl;
    ofs << "x: " << number << " div(x): " << sum << endl;
    ifs.close();
    ofs.close();
    return 0;
}
```

测试结果

小规模：[1, 1000]

```
time: 0.0182 ms
x: 840 div(x): 32
```

中规模：[1, 100000]

```
time: 2.1342 ms
x: 83160 div(x): 128
```

大规模：

[1, 10000000]

```
time: 961.849 ms
x: 8648640 div(x): 448
```

[1, 100000000]

```
time: 13507.3 ms
x: 73513440 div(x): 768
```

算法分析

使用筛法计算 1 到 b 的所有数的约数个数，因为每个数 i 将处理 $\frac{b}{i}$ 次。将所有 i 的贡献累加起来，即：

$$\sum_{i=1}^{b} \frac{b}{i}$$

这个和实际上是一个调和级数，它的近似值为$b \log b$。因此筛的时间复杂度是$O(b \log b)$。

查询部分的复杂度为$O(b - a + 1)$，相对于预处理部分$O(b \log b)$来说比较小，尤其是b较大的情况。因此，总的时间复杂度是：

$$O(b \log b)$$

Ⅲ.方法3（筛+约数定理）

算法思路

使用 约数定理 的办法可以进一步优化计算约数个数。约数定理表明，如果一个数 n 的质因数分解为：

$$n = p_1^{e_1} \times p_2^{e_2} \times \cdots \times p_k^{e_k}$$

那么 n 的约数个数为：

$$d(n) = (e_1 + 1) \times (e_2 + 1) \times \cdots \times (e_k + 1)$$

因此，通过质因数分解，可以更高效地计算一个数的约数个数。针对题目，我们可以结合方法2的**预筛法**和**约数定理**来提高效率。

算法步骤：

1. 预处理小质数：使用`埃氏筛`预处理一定范围内的质数，方便快速分解每个数的质因数。
2. 质因数分解：对于每个数 i，通过预处理的质数列表进行质因数分解，利用`约数定理公式`计算其约数个数。
3. 搜索区间：在区间 $[a, b]$ 中遍历所有的数，使用质因数分解法计算每个数的约数个数，并记录约数最多的那个数。

代码

```
//筛+约数定理
#include <bits/stdc++.h>
using namespace std;
const int N = 1e9 + 5;
bool vis[N];                                                    //vis[i] = true表示i
被筛掉，不是素数
int prime[50000];
int a, b;
int sum = 0; // 最大的约数个数
int number = 0; // 对应的数
int k = 0;
void optimal_E_sieve(int n){
    for(int i = 2; i <= sqrt(n); i++)                            //筛选掉非素数
        if(!vis[i])
            for(int j = i * i; j <= n; j += i)vis[j] = true;  //标记非素数
    //下面来记录素数
    for(int i = 2; i <= n; i++)
        if(!vis[i]) prime[++k] = i;                              //记录素数
prime[1]=2, prime[2]=3, ......
    return;
}
int div(int _n) {
    int ans = 1;
    int n = _n;

    for(int i = 1; i <= k; ++i) {
        int y = prime[i];
        if(y * y > n) break; // 如果质数大于 sqrt(n)，结束循环
        int cnt = 0;
        while(n % y == 0) n /= y, cnt++; // 分解质因数，cnt 记录质数幂
        ans *= (cnt + 1); //使用约数定理公式
    }

    // 若n > 1，则剩下的 n 为一个幂次为1的质数
    if(n > 1) ans *= 2;
    return ans;
}
int main() {
    ifstream ifs("testData/small_input3.txt");
    ofstream ofs("output3.txt");
```

```
    ifs >> a >> b;
    int sqrt_b = sqrt(b);
    // 开始计时
    auto start = chrono::high_resolution_clock::now();

    optimal_E_sieve(sqrt_b); //只需要筛到 sqrt(b) 范围内的质数
    // 遍历区间 [a, b]
    for (int i = a; i <= b; ++i) {
        int k = div(i); // 计算 i 的约数个数
        // 更新最大约数数及其对应的数
        if (k > sum) {
            sum = k;
            number = i;
        } else if (k == sum && i < number) {
            number = i;
        }
    }

    // 结束计时
    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, std::milli> duration = end - start;
    // 输出结果
    ofs << "time: " << duration.count() << " ms" << endl;
    ofs << "x: " << number << " div(x): " << sum << endl;
    ifs.close();
    ofs.close();
    return 0;
}
```

测试结果

小规模：[1, 1000]

```
time: 0.0491 ms
x: 840 div(x): 32
```

中规模：[1, 100000]

```
time: 8.386 ms
x: 83160 div(x): 128
```

大规模：

[1, 10000000]

```
time: 2129.95 ms
x: 8648640 div(x): 448
```

[1, 100000000]

```
time: 41107.5 ms
x: 73513440 div(x): 768
```

算法分析

通过筛法预处理所有不超过 $\sqrt{b}$ 的质数，时间复杂度为 $O(\sqrt{b}\log\log\sqrt{b})$，这样可以在后续进行质因数分解时避免重复计算。

对每个数 i 进行质因数分解，时间复杂度为 $O(\log i)$，因为只需要检查 $\sqrt{i}$ 内的质数。 因此，对于区间内 $b-a+1$ 个数，因为区间中的最大数为b，我们可以将对每个数的质因数分解复杂度近似为 $\log b$，总的质因数分解复杂度是：

$$O(\sum_{i=a}^{b}\log i)\approx O((b-a+1)\log b)$$

对于大范围的查询，预处理部分 $O(\sqrt{b}\log\log\sqrt{b})$ 相对较小，主要开销来自于质因数分解部分。故总的时间复杂度为：

$$O((b-a+1)\log b)$$

三种方法的时间对比:

[a,b]	time(1)	time(2)	time(3)
[1,1000]	0.1234ms	0.0182ms	0.0491ms
[1,100000]	70.1601ms	2.1342ms	8.386ms
[1,10000000]	66992.3ms	961.849ms	2129.95ms

实验心得

实验中通过使用分治法来解决具体问题，使我对分治法的实际应用有了更加深刻的理解。在设计和调试算法时，我感受到分治法的优雅之处，即通过递归地分解复杂问题，将其转换为更简单的子问题来解决。然而，分治法的效果不仅取决于问题的分解方式，还与合并结果的效率密切相关。这使得在分解和合并步骤中，我们必须慎重考虑时间复杂度和实现细节。

分治算法的核心思想就是`「分而治之」`。大概的流程可以分为三步：分解 -> 解决 -> 合并。

分解（Divide）：将原问题分解为若干个子问题，通常是将问题划分为若干个规模较小的子问题。

解决（Conquer）：递归地解决这些子问题，当子问题足够小时，直接求解。

合并（Combine）：将子问题的解组合起来，得到原问题的解。

在实验中，分治法虽然能够很好地分解问题，但随着问题规模增加，递归的深度和重复计算问题可能会导致性能问题。通过引入动态规划或记忆化搜索，可以有效减少子问题的重复计算，提升递归算法的性能。