实验 1: Sorting

PB16111485 张劲暾

1. 实验要求

实验内容:排序 n 个元素,元素为随机生成的 1 到 65535 之间的整数, n 的取值为:

$$2^2, 2^5, 2^8, 2^{11}, 2^{14}, 2^{17};$$

算法:插入排序,堆排序,快速排序,基数排序,计数排序。

2. 实验环境

编译环境: gcc version 7.2.0 (Ubuntu 7.2.0-8ubuntu3.2)
机器内存: 8G
时钟主频: 2.50GHz × 4

3. 实验过程

(1) 编写简单的 Python 程序生成需要的随机乱序整数数据:

sorting.c | input_integer.py

```python
import numpy as np
output = open("./PB16111485-project1/ex1/input/input_integer.txt",'w')
input_integer = np.random.randint(0,65536,size=(pow(2,17),1))
for integer in input_integer:
    output.write(str(integer[0]) + "\n")
output.close()
```

(2) 编写 C 语言函数实现各种排序算法:

依次编写实现五种排序算法的核心代码与辅助函数，详见第四部分。

(3) 在 main 函数中对不同长度的数据集在不同的排序算法上依次测试并记录相关数据:

首先将整个数据集读入，然后依次截取要求长度的数据在排序算法函数上测试并输出排序结果和运行时间，一个典型的测试代码块如图所示:

截取测试数据

```
//---------------------------------------2^14
A = (int*)malloc(Length5 * sizeof(int));
for(i = 0;i < Length5;i++){ A[i] = Origin[i]; }
BeginTime = clock();
InsertionSort(A,Length5);
EndTime = clock();
InsertionSortFileResult = fopen("../output/InsertionSort/result_14.txt","w");
for(i = 0;i < Length5;i++){ fprintf(InsertionSortFileResult,"%d\n",A[i]); }
fclose(InsertionSortFileResult);
fprintf(InsertionSortFileTime,"n = 14,\ttime = %d ms\n",EndTime - BeginTime);
free(A);A = NULL;
```

输出排序结果和运行时间

运行排序算法

(4) 分析数据，对比分析：

对实验得到的排序结果和时间代价数据，检查发现结果基本排序正确，用时分析详见第五部分。

4. 实验关键代码截图（结合文字说明）

(1) 插入排序：

```
void InsertionSort(int* A,int length){
    int j;
    for(j = 1;j < length;j++){
        int key = A[j];
        //insert A[j] into the sorted sequence A[1..j-1]
        int i = j - 1;
        while(i >= 0 && A[i] > key){
            A[i + 1] = A[i];
            i = i - 1;
        }
        A[i + 1] = key;
    }
    return;
}
```

将 A[0..j-1] 已排序好的比 A[j] 大的元素依次后移一位，将 A[j] 插入到合适的位置，保持循环不变式

(2) 归并排序：

1. Merge 函数：

```
void Merge(int* A,int p,int q,int r){
    int LeftLength = q - p + 1;
    int RightLength = r - q;
    int Left[LeftLength + 1],Right[RightLength + 1];
    int i,j,k;
    for(i = 0;i < LeftLength;i++){
        Left[i] = A[p + i];
    }
    for(i = 0;i < RightLength;i++){
        Right[i] = A[q + i + 1];
    }
    Left[LeftLength] = __INT_MAX__;
    Right[RightLength] = __INT_MAX__;
    for(i = 0,j = 0,k = p;k <= r;k++){
        if(Left[i] <= Right[j]){
            A[k] = Left[i];
            i++;
        }
        else{
            A[k] = Right[j];
            j++;
        }
    }
    return;
}
```

哨兵

将已经排序好的左右两个数组归并成一个有序数组

2. MergeSort 函数：

```
void MergeSort(int* A,int p,int r){
    if(p < r){
        int q = (p + r) / 2;
        MergeSort(A,p,q);
        MergeSort(A,q + 1,r);
        //now A[p..q] and A[q+1..r] are both sorted array
        Merge(A,p,q,r);
    }
    return;
}
```

子问题分解并递归解决

子问题归并

(3) 快速排序：

1. Partition 函数：

```
int Partition(int* A,int p,int r){
    int key = A[r];
    int i = p - 1,j;
    for(j = p;j < r;j++){
        if(A[j] <= key){
            i++;
            Swap(&A[i],&A[j]);
        }
    }
    Swap(&A[i + 1],&A[r]);
    return i + 1;
}
```

扫描指针

小于主元的栈顶指针

2. QuickSort 函数：

```
void QuickSort(int* A,int p,int r){
    if(p < r){
        int q = Partition(A,p,r);
        QuickSort(A,p,q - 1);
        QuickSort(A,q + 1,r);
    }
    return;
}
```

递归求解子问题，将剩余的元素放到最终正确的位置

确定划分位置并将 A[r] 放到了最终正确的位置

(4) 计数排序：

```
void CountingSort(int* A,int* B,int lengthofA){
    int CountAndPosition[RANGE];
    int i;
    for(i = 0;i < RANGE;i++){
        CountAndPosition[i] = 0;
    }
    for(i = 0;i < lengthofA;i++){
        CountAndPosition[A[i]]++;
    }
    //now CountAndPosition[i] contains the number of elements equals to i
    for(i = 1;i < RANGE;i++){
        CountAndPosition[i] = CountAndPosition[i] + CountAndPosition[i - 1];
    }
    //now CountAndPosition[i] contains the position of elements equals to i to be insert into B
    for(i = lengthofA - 1;i >= 0;i--){
        B[ CountAndPosition[ A[i] ] - 1 ] = A[i];
        CountAndPosition[ A[i] ]--;
    }
    return;
}
```

CountAndposition[i] 统计等于 i 的元素个数

CountAndposition[i] 计算等于 i 的最后一个元素应该放入的位置

(5) 基数排序：

1. CountingSortForRadix 函数：

```
void CountingSortForRadix(int* A,int* B,int lengthofA,int d){
    //offer stable sort method to RadixSort
    int CountAndPosition[10];
    int i;
    for(i = 0;i < 10;i++){
        CountAndPosition[i] = 0;
    }
    for(i = 0;i < lengthofA;i++){
        CountAndPosition[ (A[i]/(int)pow(10,d-1))%10 ]++;
    }
    //now CountAndPosition[i] contains the number of elements equals to i
    for(i = 1;i < 10;i++){
        CountAndPosition[i] = CountAndPosition[i] + CountAndPosition[i - 1];
    }
    //now CountAndPosition[i] contains the position of elements equals to i to be insert into B
    for(i = lengthofA - 1;i >= 0;i--){
        B[ CountAndPosition[ (A[i]/(int)pow(10,d-1))%10 ] - 1 ] = A[i];
        CountAndPosition[ (A[i]/(int)pow(10,d-1))%10 ]--;
    }
    return;
}
```

不同点，不是按照 i 的值进行统计，而是按照 i 的第 d 位统计

2. RadixSort 函数：

```
void RadixSort(int* A,int d,int length){
    int i,j;
    int* B = malloc(length * sizeof(int));
    for(i = 1;i <= d;i++){
        CountingSortForRadix(A,B,length,i);
        for(j = 0;j < length;j++){
            A[j] = B[j];
        }
    }
    free(B);B = NULL;
    return;
}
```

以从低位到高位的值为键依次进行稳定排序

(6) 堆排序：

1. MaxHeapIfy 函数：

```
void MaxHeapIfy(int* A,int i,int length){
    int LeftChild = 2 * i + 1;
    int RightChild = 2 * i + 2;
    int Largest = i;
    if(LeftChild < length && A[LeftChild] > A[Largest]){
        Largest = LeftChild;
    }
    if(RightChild < length && A[RightChild] > A[Largest]){
        Largest = RightChild;
    }
    if(Largest != i){
        Swap(&A[Largest],&A[i]);
        MaxHeapIfy(A,Largest,length);
    }
    return;
}
```

从根节点和左右子堆堆顶元素中选取最大者

若最大者就是根节点，那么已然是最大堆，无需调整，否则将左右子堆堆顶较大者与根节点互换并调整对应的子堆

2. BuildMaxHeap 函数：

```
void BuildMaxHeap(int* A,int length){
    int i;
    for(i = (length - 1)/2;i >= 0;i--){
        MaxHeapIfy(A,i,length);
    }
    return;
}
```

从最底层的非叶子节点开始自底向上构建最大堆

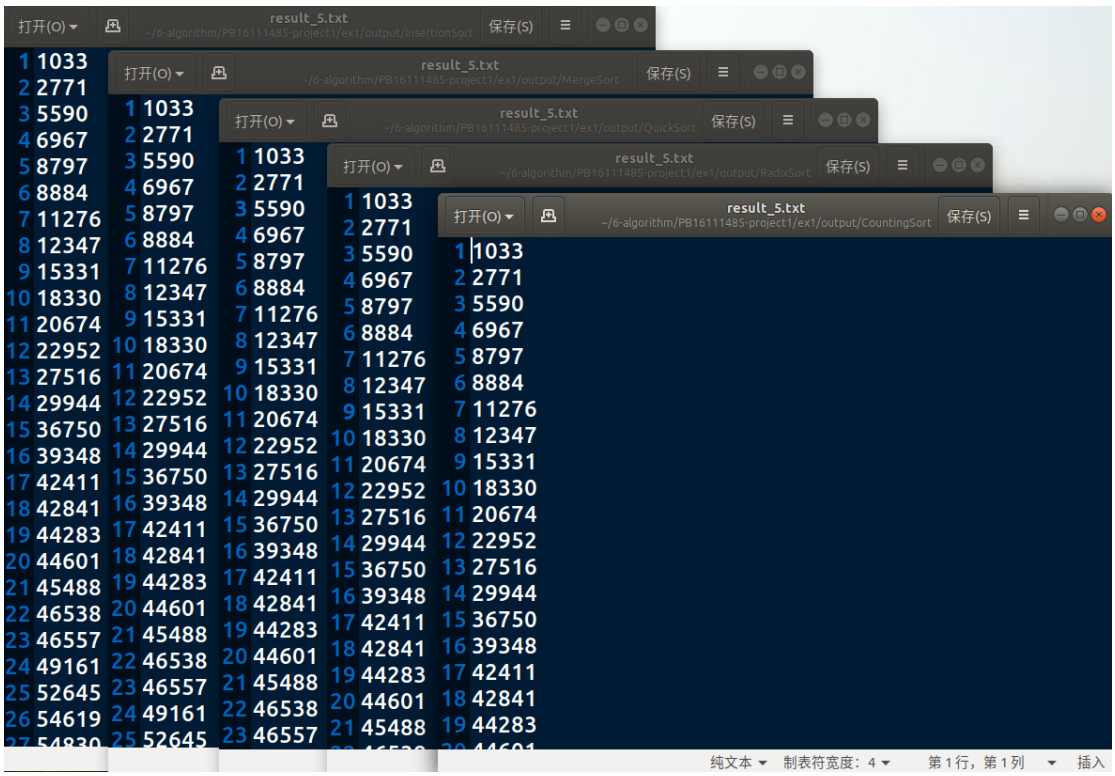
3. HeapSort 函数：

```
void HeapSort(int* A,int length){
    BuildMaxHeap(A,length);
    int i;
    for(i = length - 1;i > 0;i--){
        Swap(&A[i],&A[0]);
        int InOrder = length - i;
        MaxHeapIfy(A,0,length - InOrder);
    }
    return;
}
```

对还不是有序的部分 A[0,..,length – InOrder -1] 建堆后将堆顶最大元素划入有序部分 A[length – InOrder -1,…,length - 1]，如此循环往复完成排序

5. 实验结果、分析（结合相关数据图表分析）

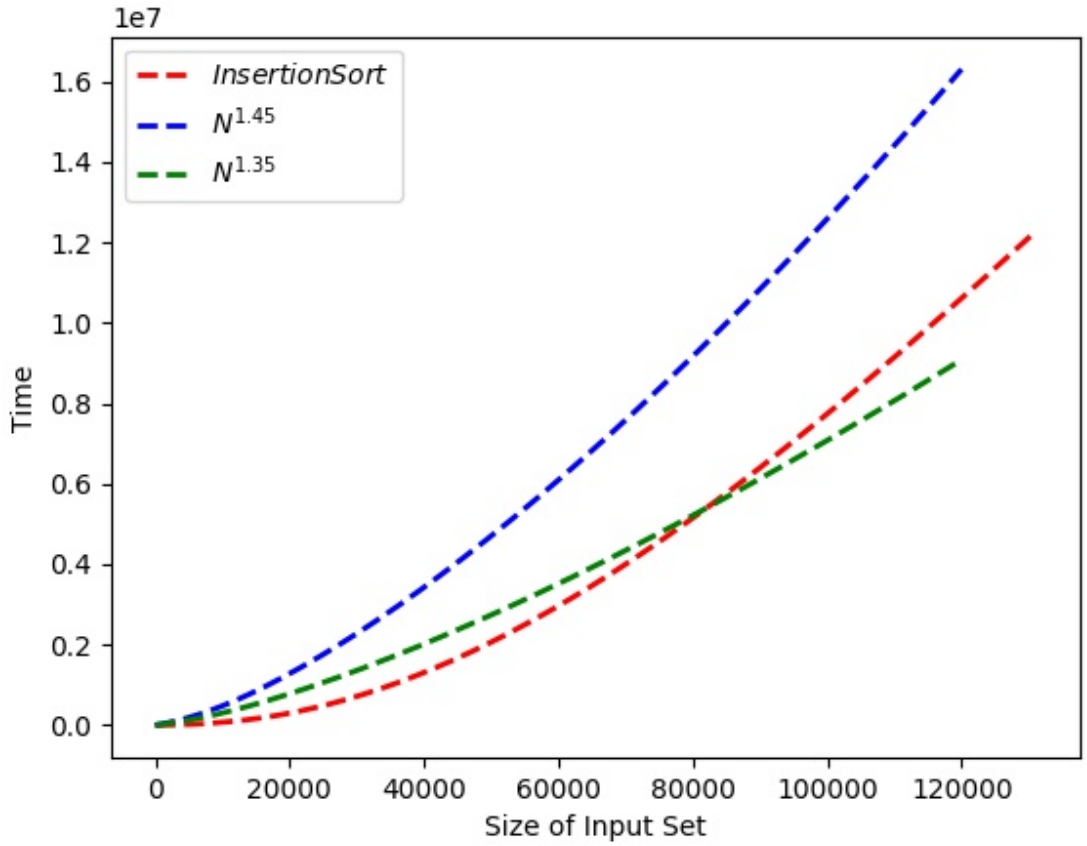
(1) 排序结果正确：（从左到右依次是插入排序,堆排序,快速排序,基数排序,计数排序在 2^5 数据集大小时的结果）



(2) 排序性能分析：

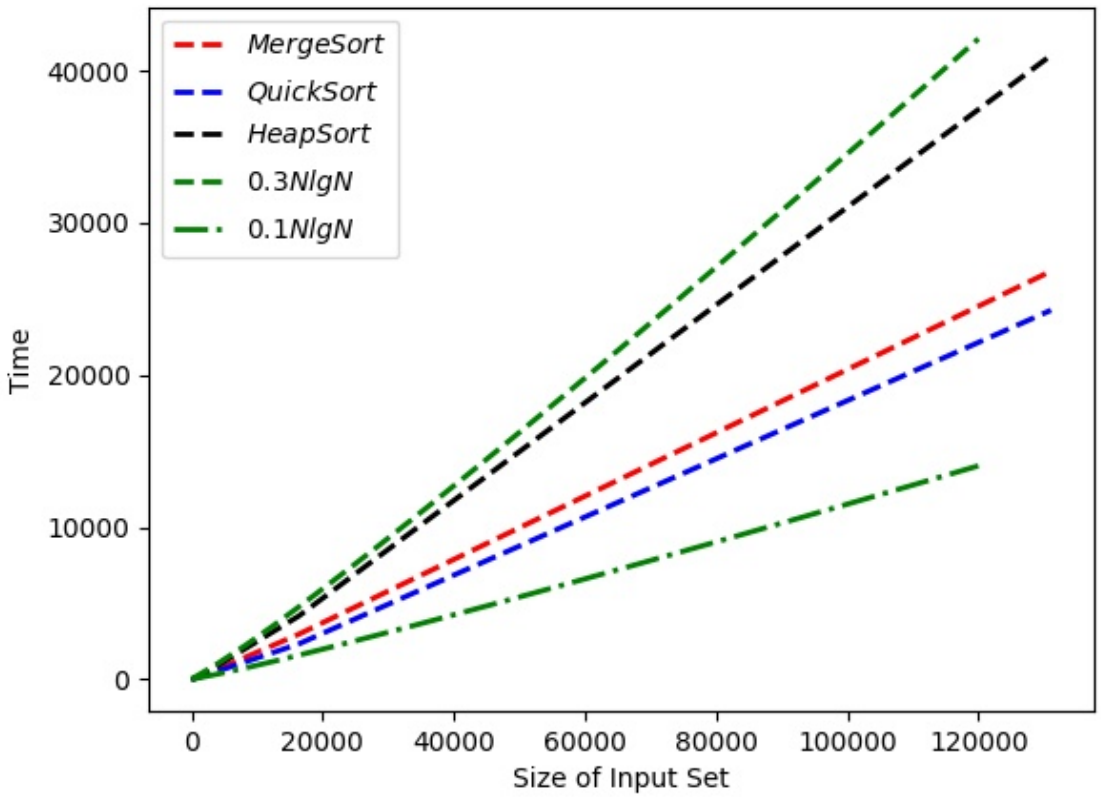
如图：(因为数量级相差过大，将五张表集成到一张表中观察体验极差，所以按数量级依次对比展示)**(注：以下所有图表时间单位为 ms)**

1. InsertionSort：$O(N^2)$



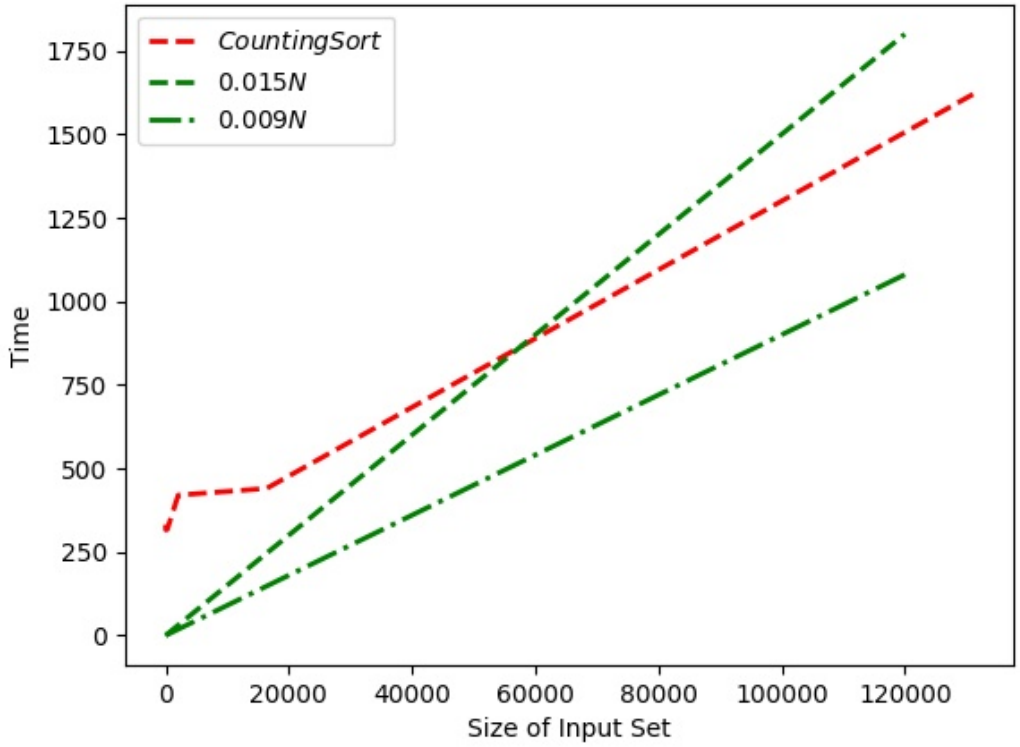
我们看到，虽然按照最坏情况插入排序的时间复杂度是 $O(N^2)$，但是对于随机的平均情况，表现出来的时间复杂度可能仅仅是介于 $O(N^{1.45})$与 $O(N^{1.35})$之间。

2. QuickSort, MergeSort, HeapSort：O(N*lgN)



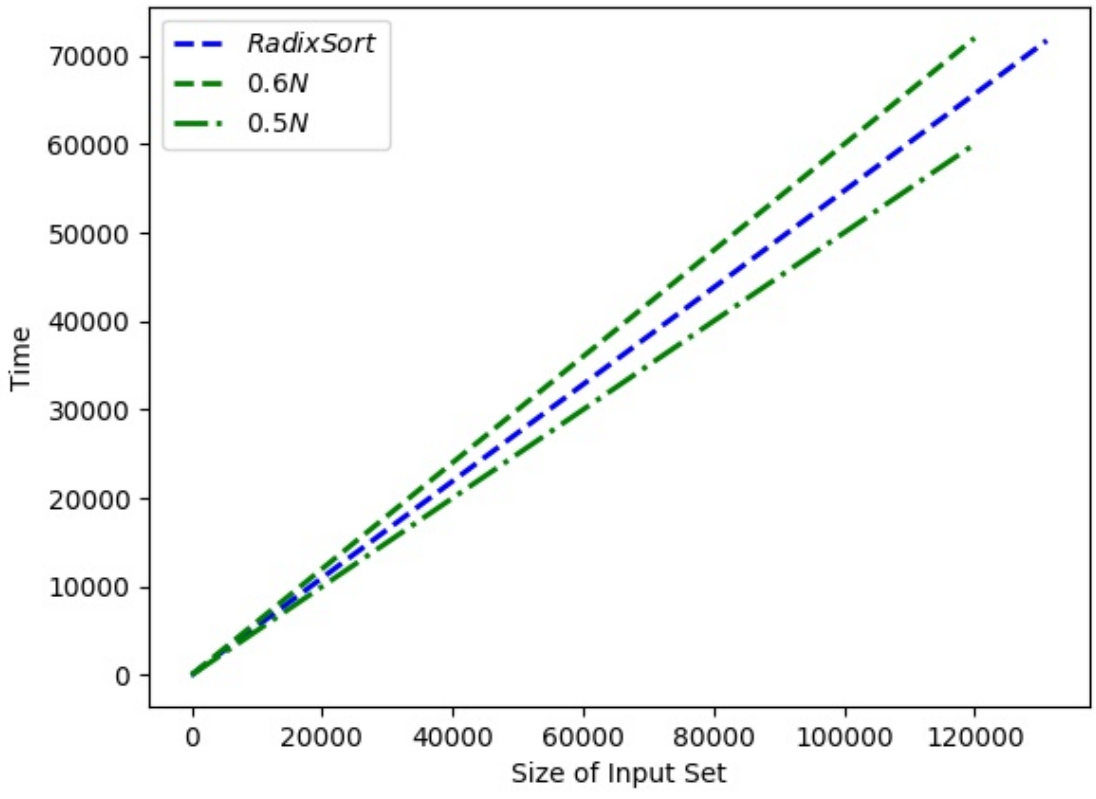
可以看到，这三种理论时间复杂度为 O(NlgN)的排序算法表现出的时间复杂度基本符合预期，介于 0.1NlgN 与 0.3NlgN 之间，相互对比会发现归并排序时间高于快速排序，主要是因为辅助空间开销大，对高速缓存不友好，堆排序代价最大，因为需要反复进行堆调整而且不能高效利用之前调整的有用比较信息，开销较大。

3. CountingSort：O(N) ≈ N



可以看到，虽然计数排序在渐近时间复杂度上表现为 O(N)，但是在数据集小于数据散布范围长度的时候，主要的时间的开销都是用于维护 CountAndPosition 数组造成的，而且当这个数组比较长的时候，对于高速缓存也非常不友好，表现为对于长分布小数据集开销大而且近似水平直线。

4. RadixSort：O(N) ≈ 5N



可以看到相比于计数排序，基数排序在这个实验主要表现为：

1. 渐近时间复杂度为 Θ(N)
2. 没有小数据集上的噪声，主要是每一位的取值只能是 0-9，不存在上面计数排序中分布过长的问题
3. 总体时间开销较大，约为计数排序的 20 倍，主要原因，一是有时间复杂度常数因子 d = 5，二是频繁的除法和取模运算时间开销较大。

6. 实验心得

(1) 熟悉了各种排序算法的编程实现，提高了编程能力，加深了对于各个排序算法的认识。

(2) 直观地体会了算法时间复杂度上的数量级差异巨大。

(3) 对于归并排序，堆排序，快速排序，相同理论时间复杂度的算法的具体表现情况与造成这种差异的算法特性有了进一步的思考与理解。

(4) 对于计数排序和基数排序在具体问题上的表现，复杂度噪声原因和优劣有了进一步的认识。

(5) 具体的算法和算法实现要适用于具体问题。

(6) 了解了 RAM 模型与实际高速缓存模型的差异及其对于算法性能和选择的影响。