西安電子科技大學

算法分析与设计（本科）
上机报告

学　　院： 计算机科学与技术学院

专　　业： 软件工程

方　　向： 嵌入式系统方向

姓　　名： 李云水

学　　号： 17130120116

排序篇：

1.插入排序

简介

插入排序与扑克排序相似，从数组的第二个数字开始直到数组末尾，在某一个数字进行排序时前面已经排好序，则排序时找到第一个比自身小的数，插入即可。

问题与思考

算法由于比较简单，所以没有遇到什么问题。

2.堆排序

简介

堆排序的基本方法，先构造一个大顶堆，将堆顶元素和末尾元素交换，再重新调整堆的结构，直到堆的大小到 1；

问题与思考

伪代码中 heap_size 的变化没有直接体现，在代码实现的过程中需要将 heap_size 作为参数传入 heapify 中。其次，在 heapify 的代码中，由于我的代码更新 index 值，我又在程序中将 index 和数值搞混了，导致花了一些时间 debug。

3.计数排序

简介

计算排序是线性的排序方法，但是有一定的限制，O(n)中 n 的大小取决于数组中的最大的数。具体的实施是先记录每一个元素出现的次数，在依次从 1~n，使得 C[i]+=C[i-1] 然后原始数组从后往前，根据 C 中的内容依次排序。

问题与思考

由于在课堂上和作业中都比较熟练的应用了此算法，所以没有遇到什么大的问题。

4.桶排序

简介

桶排序同样是线性排序方法，但其也有一定局限性，它适用于数据相对集中的一组数据。首先将数据分为各个桶中，在各个桶中使用一般的排序算法，最后将各个桶依次合并，就得到排好序的数组。

分治篇：

1.归并排序

简介

归并排序用到了分治的思想，将排序问题分解为小规模的排序问题，再合并小数组的排序结果，就得到了排序后的数组。

问题与思考

在 Merge 的时候，new 两个小数组的方法出现了一些问题，最后改变了策略，先 new 一个大数组，排好大数组后再更新原数组的值。

由于算法课上只是介绍基本的方法，还有很多代码本身的问题需要考虑，要注意数组的上下界。

2.最长子序和

简介

当最大子数组有 n 个数字时：

若 n==1，返回此元素。**left_sum** 为最大子数组前 n/2 个元素，在索引为 (left + right) / 2 的元素属于左子数组。**right_sum** 为最大子数组的右子数组，为最后 n/2 的元素。**cross_sum** 是包含左右子数组且含索引 (left + right) / 2 的最大值。

最后选取三者最大值即可。

问题与思考

这个是使用分治法解决问题的典型的例子，并且可以用与合并排序相似的算法求解

3.多数元素的求解（指在数组中个数大于 n/2 的元素）

简介

如果数 a 是数组 nums 的众数，如果我们将 nums 分成两部分，那么 a 必定是至少一部分的众数。

我们可以使用反证法来证明这个结论。假设 a 既不是左半部分的众数，也不是右半部分的众数，那么 a 出现的次数少于 l / 2 + r / 2 次，其中 l 和 r 分别是左半部分和右半部分的长度。由于 l / 2 + r / 2 <= (l + r) / 2，说明 a 也不是数组 nums 的众数，因此出现了矛盾。所以这个结论是正确的。

这样以来，我们就可以使用分治法解决这个问题：将数组分成左右两部分，分别求出左半部分的众数 a1 以及右半部分的众数 a2，随后在 a1 和 a2 中选出正确的众数。

问题与思考

我们使用经典的分治算法递归求解，直到所有的子问题都是长度为 1 的数组。长度为 1 的子数组中唯一的数显然是众数，直接返回即可。如果回溯后某区间的长度大于 1，我们必须将左右子区间的值合并。如果它们的众数相同，那么显然这一段区间的

众数是它们相同的值。否则，我们需要比较两个众数在整个区间内出现的次数来决定该区间的众数。

4.快速排序

简介

快速排序同样用到了分治的思想

快速排序的基本方法，找到一个数它的位置，再分别对位置前的数组和位置后的数组排序。

问题与思考

由于其实在刷题中已经手打了好几次代码，所以其实没有什么问题。

动态规划篇：

1.装配线调度问题

简介

这属于动态规划的典型问题，也是入门问题，我们可以很容易就找到它最优子结构:

- Have the following formulas:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j},\ f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

Using symmetric reasoning, we can get the fastest way through station $S_{2,j}$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j},\ f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

沿着装配线走到末尾就能得到最短花费

问题与思考

通过本次实验，我了解到动态规划算法的优越性，很明显使用动态规划算法之后，算法的复杂度降到了 O(n)，这大大降低了我们的运算时间，这道题的关键是我们想到用dp[]数组来保存我们的状态。

2. 矩阵链乘法问题

简介

用 m[i, j]示计算矩阵链 $\langle A_i, A_{i+1}, \cdots, A_j \rangle$ 所需标量乘法次数的最小值。如果 i = j，矩阵链中只有一个矩阵，显然 m[i, j] = 0。对于 i < j 的情况，上文提到，可以先将矩阵

链$\langle A_i, A_{i+1}, \cdots, A_j \rangle$划分为两个子链$\langle A_i, \cdots, A_k \rangle$和$\langle A_{k+1}, \cdots, A_j \rangle$, 这样我们很容易得到递归式

$$m[i][j]=\begin{cases}0 & , \\ \min\limits_{i\leqslant k<j}\{m[i][k]+m[k+1][j]+p[i-1]*p[k]*p[j]\} & ,\end{cases}$$

我们构造一个矩阵依次来填表

问题与思考

通过本次实验，加深了自己对于动态规划算法的理解。动态规划算法难在建立递归关系，也就是难在如何去划分最优的子结构，对于这个问题也就是难在你要想到用 k 去把子问题分解，然后把问题落在打表上，根据自己打出来的表，减少重复计算的次数，完成表格，也就是完成了整个动态规划的过程。

同时需要注意的是，虽然我们建立的是递归表达式，但是我们的代码是用循环来实现的。也就是说我们在找最优的解空间。

3.最长公共子序列问题（非连续子序列）

简介

一个序列 S，如果分别是两个或多个已知序列的子序列，且是符合此条件的子序列中最长的，则称 S 为已知序列的最长公共子序列。我们给定两个序列，求出这两个序列的最长公共子序列。

我们可以得到递推公式

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0\,, \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j\,, \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j\,.\end{cases}$$

问题与思考

通过本次实验，增加了自己对于动态规划的认识，通过 C[i][j]这个状态量，来寻找最优的解，然后每次记录计算的过程，把最优的解求出来，我们可以看出来，LCS 的复杂度是 O(mn)这比暴力枚举状态，降低了复杂度，这是因为我们之中避免了很多次的重复计算，要掌握动态规划划分子问题的技巧与一般的步骤。

4.Rod_Cutting 问题

简介

与矩阵链乘问题类似，稍有变化的是它的最优子结构表达式

$$r_n=\max_{1\le i\le n}(p_i+r_{n-i})$$

贪心篇：

1.活动选择问题

简介

一个简单的调度问题。给定作业 j1、j2……jn，它们的运行时间分别为 t1、t2……tn。我们只有一个处理器。为了使平均完成时间最小化，安排这些作业的最佳方法是什么?假设这是一种非抢占式调度:一旦作业启动，它必须运行到完成。

这是在操作系统中我们学过的一个问题，也就是作业的调度，为什么我们要采取最短作业优先这种调度方式呢？其实这正是一种贪心的策略，因为任何一个任务结束之前，其他任务都是无法工作的，也就是说这样后面的每个任务都会多一个前面任务的延时，可以很容易的想到，前面的任务一定是时间越短越好，这样计算平均时间时，对后面的任务造成的延时也比较小，所以这个问题的思路也打开了。

问题与思考

对于这个问题经过分析可以得到，越是靠近在前面的越短时间对最后的总延时造成的影响越小，所以这里我们采取贪心的策略，让每一步都选取最小的任务，我们可以通过分析来证明我们的策略是对的，这是贪心策略的经典任务，所以一定要具体问题具体分析，在合适的场景使用贪心算法。

2.最长子序列问题

简介

刚开始我们会想到用暴力去枚举这些所有的和，然后在这些和里面找出最大的，但是这样做，算法复杂度会来到 O(n^3),不是最优的解，刚好我们最近在学贪心算法，这道题利用贪心算法很好解决。

我们假设用 dp[n]来表示以第 n 个数结尾的最大连续子序列的和

我们便可以得到这两种情况

1.这个最大和的连续序列只有一个元素，即以 A[i] 开始，以 A[i] 结尾。

2.这个最大和的连续序列有多个元素，即从前面某处 A[p] 开始 (p<i)，一直到 A[i] 结尾。

对第一种情况，最大和就是 A[i] 本身。

对第二种情况，最大和是 dp[i-1]+A[i]。

于是我们得到状态转移方程

dp[i] = max{A[i], dp[i-1]+A[i]}

转化为贪心算法

```
    for(auto i:A){
        // sum=max(i,sum+i);
//          res=max(sum,res);
```

3.Dijkstra 算法

简介

我们使用 Dijkstra's 算法找到从源节点到所有节点的最短路径，Dijkstra's 算法是每次扩展一个距离最短的点，更新与其相邻点的距离

问题与思考

Dijkstra 算法 $O(N2)$ 的基本实现方法和 $O(NlogN)$用堆实现的方法。目前我实现了 $O(N2)$ Dijkstra 算法，因为 $O(NlogN)$ 中的优先队列目前在特定语言环境中还不知道如何实现，相信很快我就能实现。

4. Bellman-Ford 算法

简介

1.观察到邻接矩阵中有带负值的权，所以我们不能使用迪杰斯特拉算法进行来求出单源最短路径，所以我们选用 Bellman-Ford 算法来求出单源最短路径。首先要做初始化，把原点设置为 0，其余结点均是无穷。

2.Bellman-Ford 算法的要对，每一条边进行遍历，然后使用松弛时间来更新这条边两个节点上的值，也就是源点到这个点的路径值，在每次值进行更新之后，要重新记录出没有前驱的结点。

3. 要注意判断是否会出现负圈路径的问题

问题与思考

Bellman-Ford 算法解决了带负权值的问题，也就是说它解决了迪杰斯特拉算法不能解决的问题，Bellman-Ford 的核心是不断对每条边进行松弛操作，使得顶点集 V 中的每个顶点 v 的最短距离估计值逐步逼近最短距离，我们需要 v-1 次这样的迭代，那么为什么要循环 v-1 次呢？因为最短路径肯定是个简单路径而不是一个回路，如果包含回路，回路的权值是正的，那么去掉这个回路，如果回路值是负的话，那就没有解了。因为有 n 个点，不能有回路，所以最短路径最多有 n-1 条边。所以至多 relax n-1 次，如果还能继续更新，说明存在回路，就无法解决了。