

华中科技大学

课程实验报告

课程名称： 串行与并行数据结构及算法

专业班级： 计算机本硕博 2101

学 号： U202115666

姓 名： 刘文博

指导教师： 陆枫

报告日期： 2021.6.15

计算机科学与技术学院

目 录	2
-----	---

目录

1 实验一：无重复排序	3
2 实验二：最短路	5
3 实验三：最大括号距离	9
4 实验四：天际线	11
5 实验五：括号匹配	13
6 实验六：高精度整数	15
7 实验七：割点和割边	17
8 实验八：静态区间查询	18
9 实验九：素性测试	19

1 实验一：无重复排序

1.1 题目描述

给出一个具有 N 个互不相同元素的数组，请对它进行升序排序。第一行为一个整数 N ，表示元素的个数。第二行为 N 个整数，表示这 N 个元素，保证每个元素均在 `int` 范围内

1.2 算法流程

本题使用快速排序，做法是用 *sml* 的 *List.partition* 函数将序列分为小于 x 和大于 x 的两部分，进行递归求解，然后将两个子序列连接得到原问题的解。

由于 *partition* 本身就做了比较操作，故递归边界是子问题规模为 0 时返回一个空序列。

```

1  val N = getInt();
2  val a = getIntTable(N);
3  fun quickSort [] = []
4  | quickSort(x::xs) =
5  let val (left, right) = List.partition(fn y => y < x) xs
6  in
7  quickSort(left) @ [x] @ quickSort(right) end
8  val res = quickSort(a);
9  printIntTable(res);

```

1.3 复杂度分析

partition 的 *Work* 为 $O(n)$, *Span* 为 $O(\lg n)$ ，我们每次选择序列的第一个元素为 *pivot*。

最好情况下每次都能将序列划分为相等的两部分

$$W(n) = 2W\left(\frac{n}{2}\right) + O(n)$$

$$S(n) = 2S\left(\frac{n}{2}\right) + S(\lg n)$$

由主定理 $W(n) = O(n \lg n)$, $S(n) = O(\lg^2 n)$

最坏情况下序列基本有序, 总是将序列划分为一个大小为 $n-1$ 的子序列和一个空序列

$$W(n) = W(n-1) + O(n)$$

$$S(n) = S(n-1) + S(\lg n)$$

由主定理 $W(n) = O(n^2)$, $S(n) = O(n \lg n)$

在平均情况下, 假设输入数组的元素是随机分布的, 选择第一个元素作为枢轴的快速排序的平均复杂度分析为 $W(n) = O(n \lg n)$, $S(n) = O(\lg^2 n)$

1.4 样例分析

测试样例如下图所示, 它是一个长度为 10 的序列

```
测试输入： 10
            10 155 200 9 60 174 17 6 172 103
```

首先选择 10 作为基准元素, 进行划分得到 $[[9\ 6]\ 10\ [155\ 200\ 60\ 174\ 17\ 172\ 103]]$

然后对于左边和右边的再次进行同样的步骤, 左边以 9 作为基准元素, 右边以 155 作为基准元素, 进行划分得到 $[[[6]\ 9\ []]\ [[60\ 17\ 103]\ 155\ [200\ 174\ 172]]]$

继续这样不断递归, 直到子序列长度为 0 时终止递归, 返回结果, 按照左边 + 基准元素 + 右边的方式将其连接并返回

最终返回结果 $[6, 9, 10, 17, 60, 103, 155, 172, 174, 200]$

2 实验二：最短路

2.1 题目描述

给定一个带权无向图，一个源点，权值在边上。计算从源点到其他各点的最短路径。输入格式为：

- 第一行:3 个由空格隔开的整数: N, M, T_s 。其中 N 表示结点的数量 (从 1 到 N), M 表示边的数量, T_s 表示源点
- 第 2 到第 $M + 1$ 行: 描述每条边, 每行包含 3 个由空格隔开的整数: R_s, R_e, C_i , 其中 R_s 和 R_e 是两个结点的编号, C_i 是它们之间的边的权值

输出格式为：

- N 个整数, 表示从源点 T_s 到各顶点的最短路径长度。如果到某个顶点不连通, 对应最短路径长度输出 -1。

2.2 算法流程

本题需要计算从源点到其他各点的最短路径, 因为不存在路径长度为负的情况, 可以使用 Dijkstra 算法来解决

算法描述如下：

1. 创建一个初始值足够大的长度为 N 的数组 $distance[]$ 来记录记录源点到所有点的最短距离
2. 创建一个初始值为 `false` 的长度为 N 的布尔数组 $visited[]$ 来记录该点是否已被访问
3. 在未访问的顶点中, 找到距离源点最近的顶点, 将其标记为已访问
4. 更新与该顶点相邻的未访问顶点的最短路径长度, 如果经过当前顶点到达相邻顶点的路径长度比原先的路径长度短, 则更新最短路径长度
5. 如果还有节点没有访问, 回到步骤 3
6. 此时 $distance[]$ 中的值即为源点到各顶点的最短路径长度, 输出该值

算法伪代码描述如下：

```
function Dijkstra(graph, source):
    distance[source] = 0
    for  $i = 1$  to  $N$  :
        if  $i \neq \text{source}$ :
            distance [ $i$ ] =  $\infty$ 
    visited[source] = true
    for  $k = 1$  to  $N - 1$  :
        minDist =  $\infty$ 
         $u = -1$ 
        for  $v = 1$  to  $N$  :
            if visited [ $v$ ] = false and distance [ $v$ ] < minDist:
                minDist = distance [ $v$ ]
                 $u = v$ 
        if  $u = -1$  :
            break
        visited [ $u$ ] = true
        for  $v = 1$  to  $N$  :
            if visited [ $v$ ] = false and graph [ $u$ ] [ $v$ ] > 0 :
                if distance [ $u$ ] + graph [ $u$ ] [ $v$ ] < distance [ $v$ ] :
                    distance [ $v$ ] = distance [ $u$ ] + graph [ $u$ ] [ $v$ ]
    return distance
```

2.3 复杂度分析

步骤三在未访问的顶点中，找到距离源点最近的顶点需要 $O(n)$ 的 *Work* 和 $O(\lg n)$ 的 *Span*

步骤四更新与该顶点相邻的未访问顶点的最短路径长度需要 $O(n)$ 的 *Work* 和 $O(1)$ 的 *Span*

Dijkstra 是一个典型的串行算法, 它的每次计算都依赖前面的结果

那么总的 *Work* 为 $O(n^2)$, *Span* 为 $O(\lg n)$

用邻接矩阵保存图，用两个长度为 N 的数组来记录距离和标记访问，则总的空间复杂度为 $O(n^2)$

2.4 样例分析

测试样例如下图所示，它是一个包含 7 个节点，15 条边的图，其中源点为 5 号节点

```
测试输入：  7 11 5
             2 4 2
             1 4 3
             7 2 2
             3 4 3
             5 7 5
             7 3 3
             6 1 1
             6 3 4
             2 4 3
             5 6 3
             7 2 1
```

算法过程如下：

1. $distance$ 为 $[\infty, \infty, \infty, \infty, 0, \infty, \infty]$
2. $visited$ 为 $[false, false, false, false, true, false, false]$
3. $minDist$ 为未访问的 6 号节点到源点的距离 3, 更新 $distance$ 为 $[\infty, \infty, \infty, \infty, 0, 3, \infty]$,
 $visited$ 为 $[false, false, false, false, true, true, false]$
4. 更新与 6 号节点相邻的未访问顶点的最短路径长度, 更新 $distance$ 为 $[4, \infty, 7, \infty, 0, 3, \infty]$
5. 还有节点未访问, $minDist$ 为未访问的 1 号节点到源点的距离 4, 更新 $visited$ 为
 $[true, false, false, false, true, true, false]$
6. 更新与 1 号节点相邻的未访问顶点的最短路径长度, 更新 $distance$ 为 $[4, \infty, 7, 7, 0, 3, \infty]$
7. 还有节点未访问, $minDist$ 为未访问的 7 号节点到源点的距离 5, 更新 $distance$
为 $[4, \infty, 7, 7, 0, 3, 5]$, $visited$ 为 $[true, false, false, false, true, true, true]$

8. 更新与 7 号节点相邻的未访问顶点的最短路径长度,更新 *distance* 为 [4, 6, 7, 7, 0, 3, 5]
9. 还有节点未访问, *minDist* 为未访问的 2 号节点到源点的距离 6, 更新 *visited* 为 [true, true, false, false, true, true, false]
10. 更新与 4 号节点相邻的未访问顶点的最短路径长度, *distance* 不变, 此时我们其实已经得到了最终答案
11. 重复上面的过程, 直到所有节点都被访问
12. 此时 *distance*[] 中的值即为源点到各顶点的最短路径长度, 输出 [4, 6, 7, 7, 0, 3, 5]

3 实验三：最大括号距离

3.1 题目描述

现在给你一个串，你需要找出所有这个串中匹配的子串（一个闭合的串，并且外侧由括号包裹）中最长的那个，输出它的长度。第一行输入一个数 N ，表示序列的长度，满足 $N \leq 30000$ 。接下来一行输入 N 个数，表示这个括号序列，0 代表左括号，1 代表右括号。

3.2 算法流程

我们可以用栈来解决该问题，遍历括号序列，当遇到左括号时，将其位置入栈；当遇到右括号时，从栈顶去除相对应的左括号的位置，计算括号的距离（右括号减左括号加 1），并更新最大距离。最后返回最大距离即可

实现代码如下

```
1  (*****Begin*****)
2  val N = getInt();
3  val s = ListPair.zip(List.tabulate(N, fn x => x), getIntTable(N));
4  fun parenDist((pos, x), (stack, max)) =
5      if x = 0 then (pos::stack, max)
6      else if stack = [] then (stack, max)
7      else
8          let val top = hd stack
9          val tmp = Int.max(max, pos - top + 1)
10         in
11             (tl stack, tmp) end;
12  val res = #2(foldl parenDist ([], 0) s);
13  printInt(res);
14  (*****End*****)
```

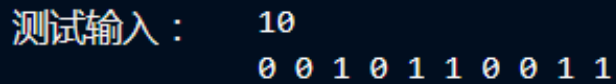
其中 pos 记录括号的位置， x 是当前读到的括号序列的值， $stack$ 是一个序列用来实现栈

3.3 复杂度分析

该算法遍历输入序列，故其 $Work = Span = O(n)$ ，使用栈来模式匹配，空间复杂度为 $O(n)$

3.4 样例分析

测试样例如下图所示，它是一个长度为 10 的序列



```
测试输入： 10
           0 0 1 0 1 1 0 0 1 1
```

算法过程如下：

1. 栈为空， $\max=0$
2. 第零个元素是 0， $\text{pos}=0$ 入栈
3. 第一个元素是 0， $\text{pos}=1$ 入栈
4. 第二个元素是 1，出栈并计算当前括号的距离为 $2-1+1=2$ ，更新 $\max=2$
5. 第三个元素是 0， $\text{pos}=3$ 入栈
6. 第四个元素是 1，出栈并计算当前括号的距离为 $4-3+1=2$
7. 第五个元素是 1，出栈并计算当前括号的距离为 $5-0+1=6$ ，更新 $\max=2$
8. 第六个元素是 0， $\text{pos}=6$ 入栈
9. 第七个元素是 0， $\text{pos}=7$ 入栈
10. 第八个元素是 1，出栈并计算当前括号的距离为 $8-7+1=2$
11. 第九个元素是 1，出栈并计算当前括号的距离为 $9-6+1=4$
12. 得到 $\max=6$

4 实验四：天际线

4.1 题目描述

第一行输入一个整数 N ，表示建筑的数量。接下来 N 行，每行输入 3 个整数 li, hi, ri 分别为建筑的左边界坐标，高度，右边界坐标

输出天际线的轮廓，即在建筑高度发生突变的位置输出建筑坐标以及建筑的高度 (在建筑重合的地方我们可以知道低的建筑可以被高的建筑阻挡，因此我们只需注意每个坐标的最高建筑的高度即可)

4.2 算法流程

采用扫描线的算法, 只在边界点进行比较, 使用优先队列维护最大高度, 使用一个轮廓列表来存储轮廓结果

1. 在输入数据的基础上加入地平线
2. 使用快速排序按左边界坐标对输入的建筑物进行排序
3. 对所有边界点进行快排
4. 遍历排序后的边界点序列, 将左边界等于当前边界点的建筑的高度入队, 将右边界等于当前边界点的建筑的高度出队, 当最大高度 (优先队列队首元素) 改变时将当前坐标和新的最大高度存入轮廓列表
5. 输出轮廓序列

4.3 复杂度分析

步骤一的快速排序的 $Work$ 为 $O(n \lg n)$, $Span$ 为 $O(\lg^2 n)$

步骤二串行扫描, 每次更新需要保持优先队列有序, 则 $Work = Span = O(\lg n)$, 一共 $2n$ 次, 所以 $Work = Span = O(n \lg n)$

故算法总的 $Work = Span = O(n \lg n)$

算法的空间复杂度易知为 $O(n)$

4.4 样例分析

测试样例如下图所示, 它包含 4 个建筑物

样例输入1				
1.	4			
2.	1	3	4	
3.	3	2	11	
4.	6	6	8	
5.	7	4	10	

算法过程如下：

1. 边界点序列为 $[1, 3, 4, 6, 7, 8, 10, 11]$, 同时得到有序的建筑物序列
2. 当前边界点为 1, 地平线 $(1,0,11)$ 和 $(1,3,4)$ 入队, 队首元素为 $(1,3,4), (1,3)$ 存入轮廓序列
3. 当前边界点为 3, $(3,2,11)$ 入队
4. 当前边界点为 4, $(1,3,4)$ 出队, 队首元素为 $(3,2,11), (4,2)$ 存入轮廓序列
5. 当前边界点为 6, $(6,6,8)$ 入队, 队首元素为 $(6,6,8), (6,6)$ 存入轮廓序列
6. 当前边界点为 7, $(7,4,10)$ 入队
7. 当前边界点为 8, $(6,6,8)$ 出队, 队首元素为 $(7,4,10), (8,4)$ 存入轮廓序列
8. 当前边界点为 10, $(7,4,10)$ 出队, 队首元素为 $(3,2,11), (10,2)$ 存入轮廓序列
9. 当前边界点为 11, $(3,2,11)$ 出队, 队首元素为 $(1,0,11), (11,0)$ 存入轮廓序列
10. 输出轮廓序列

5 实验五：括号匹配

5.1 题目描述

给定一个括号序列，判断它是否是匹配的。注意 $()()$ 在本题也当做匹配处理。
第一行输入一个整数 N ，满足 $N \leq 20000$ ，表示括号的个数。第二行输入 N 个整数 0 或 1，0 表示左括号，1 表示右括号。如果匹配，输出 1，否则输出 0。

5.2 算法流程

这一题相对比较简单，我们用一个 `state` 来记录括号匹配的状态。

我们遍历括号序列，当遇到左括号 `state` 就加 1，遇到右括号 `state` 就减 1，当遍历完序列时 `state` 恰好还是 0，那么就匹配上了。当然如果遍历过程中出现 `state` 小于 0 的情况那么说明已经出错了，最后直接输出 0 就可以了。

实现代码如下：

```
1  (*****Begin*****)
2  val N = getInt();
3  val s = getIntTable(N);
4  fun match([], state) = state
5  | match(x::xs, state) =
6      if state = ~1 then ~1
7      else if x = 1 then match(xs, state - 1)
8      else match(xs, state + 1);
9  val res:int = if match(s, 0) = 0 then 1 else 0;
10 printInt(res);
11 (*****End*****)
```

5.3 复杂度分析

我们用一次遍历过程来得到结果，故该算法的 $Work = Span = O(n)$

使用一个长度为 n 的序列来保存括号序列，故空间复杂度为 $O(n)$

5.4 样例分析

测试样例如下图所示,它是一个长度为 6 的序列

#####样例输入1

```
1. 6
2. 0 0 0 1 1 1
```

算法过程如下：

1. state 初始为 0
2. 第一个元素为 0，state 加 1，为 1
3. 第二个元素为 0，state 加 1，为 2
4. 第三个元素为 0，state 加 1，为 3
5. 第四个元素为 1，state 减 1，为 2
6. 第五个元素为 1，state 减 1，为 1
7. 第六个元素为 1，state 减 1，为 0
8. match 函数返回 0，则 res 为 1，输出 1

6 实验六：高精度整数

6.1 题目描述

给定两个任意精度的整数 a 和 b ，满足 $a \leq b$ ，求出 $a + b$, $a - b$, $a \times b$ 的值。顺序均为从高到低

6.2 算法流程

使用一维数组来存储高精度数，采用类似竖式计算的方式将结果写入到另外一个数组中，最后输出

各种计算的具体实现思路：

- 定义一个初始值全 0 的结果数组
- 加法：定义一个初始为 0 的进位值，从最低位开始，对两个加数的每位及进位值进行加法运算，结果 $\text{mod } 10$ 存入结果数组， $\text{div } 10$ 作为进位值，当 b 取到最高位后取 0 参与运算，当 a 取到最高位结束
- 减法：定义一个初始为 0 的借位值，从最低位开始，对两个数的每位及借位值进行减法运算，借位值此时变为 0，结果为正就存入结果数组，结果为负就再加 10 存入，同时借位值变为 1，当 b 取到最高位后取 0 参与运算，当 a 取到最高位时结束
- 乘法：从 b 的最低位开始，每一位与 a 相乘后与结果数组中的对应值累加，当 b 的最高位计算结束之后处理结果数组中的进位，方法和上面的加法进位类似，当某一位结果为 0 且无进位值时结束
- 输出结果：

6.3 复杂度分析

6.4 样例分析

测试样例如下图所示, 其中 a 为 133, b 为 100

样例输入1

```
1. 3
2. 1 3 3
3. 3
4. 1 0 0
```

算法过程如下：

7 实验七：割点和割边

7.1 题目描述

7.2 算法流程

7.3 复杂度分析

7.4 样例分析

8 实验八：静态区间查询

8.1 题目描述

8.2 算法流程

8.3 复杂度分析

8.4 样例分析

9 实验九：素性测试

9.1 题目描述

9.2 算法流程

9.3 复杂度分析

9.4 样例分析