華中科技大學

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：U**202115666**

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

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。

实现代码如下：

```
val N = getInt();
val a = getIntTable(N);
fun quickSort [] = []
| quickSort(x::xs) =
let val (left, right) = List.partition(fn y => y < x) xs
in
quickSort(left) @ [x] @ quickSort(right) end
val res = quickSort(a);
printIntTable(res);
```

1.3 复杂度分析

$partition$ 的 $Work$ 为 $O(n)$,$Span$ 为 $O(lgn)$, 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

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

$$S(n) = 2S(\frac{n}{2}) + S(lgn)$$

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

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

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

$$S(n) = S(n-1) + S(lgn)$$

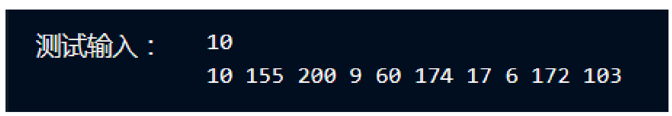
由主定理 $W(n) = O(n^2), S(n) = O(nlgn)$

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 $W(n) = O(nlgn), S(n) = O(lg^2 n)$

由于只需要一个长度为 n 的数组来保存输入，故算法的空间复杂度为 $O(n)$

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



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 ≠ source:
        distance [i] = ∞
visited[source] = true
for k = 1 to N − 1 :
    minDist = ∞
    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 $O(lgn)$ 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 $Span$

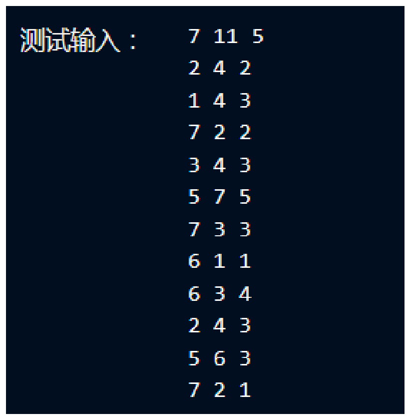
$Dijkstra$ 是一个典型的串行算法, 它的每次计算都依赖前面的结果

那么总的 $Work$ 为 $O(n^2)$,$Span$ 为 $O(nlgn)$

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



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≤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：

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

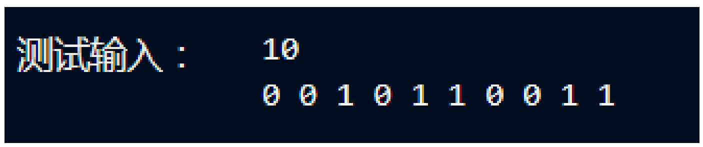
其中 pos 记录括号的位置，x 是当前读到的括号序列的值，stack 是一个序列用来实现栈

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



算法过程如下：

1. 栈为空，max=0
2. 第零个元素是 0，pos=0 入栈
3. 第一个元素是 0，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，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，pos=6 入栈
9. 第七个元素是 0，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
具体实现思路：

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

故算法总的 $Work = Span = O(nlgn)$

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

```
4
1 3 4
3 2 11
6 6 8
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：

```
(*****Begin*****)
val N = getInt();
val s = getIntTable(N);
fun match([], state) = state
| match(x::xs, state) =
    if state = ~1 then ~1
    else if x = 1 then match(xs, state - 1)
    else match(xs, state + 1);
val res:int = if match(s, 0) = 0 then 1 else 0;
printInt(res);
(*****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
- 将读入的 a 和 b 逆序存入相应的数组
- 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 mod 10 存入结果数组，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：将结果逆序输出，最高位的 0 不输出

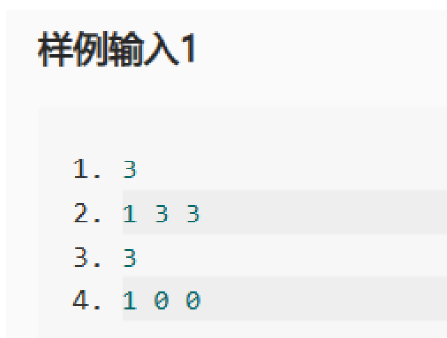
6.3 复杂度分析

定义存储 a 的数组长度为 n，存储 b 的数组长度为 m
各高精度计算的复杂度如下：

- 根据结果的最大长度，加法和减法的空间复杂度 $O(n)$，如果乘法的累加过程使用 reduce 的话我们需要保存位与位相乘的结果，故其空间复杂度为 $O(n*m)$
- 加法：串行计算每一位，故加法的 $Work = Span = O(n)$
- 减法：串行计算每一位，故减法的 $Work = Span = O(n)$
- 乘法:b 的每一位与 a 做乘法再累加,因为此时不需要考虑进位,可以并行计算,则每次 $Work = O(n), Span = O(1)$,一共做 m 次,$Work = O(n*m), Span = O(n)$,处理进位的过程可以用 reduce 来进行累加每位的值，但进位仍需串行则此部分的 $Work = O(n^2), Span = O(nlgn)$，故总的 $Work = O(n^2), Span = O(nlgn)$

6.4 样例分析

测试样例如下图所示, 其中 a 为 133，b 为 100



算法过程如下：

1. 读入 a 得到序列 $[3, 3, 1]$, 读入 b 得到序列 $[0, 0, 1]$
2. 加法：
   (a) $3 + 0 + 0 = 3$ 存入结果数组

   (b) $3+0+0=3$ 存入结果数组
   (c) $1+1+0=1$ 存入结果数组
   (d) 结束得到结果 $[3,3,2]$

3. 减法：
   (a) $3-0-0=3$ 存入结果数组
   (b) $3-0-0=3$ 存入结果数组
   (c) $1-1-0=0$ 存入结果数组
   (d) 结束得到结果 $[3,3,0]$

4. 乘法：
   (a) 0 与 $[3,3,1]$ 相乘, 累加进结果数组，得到 $[0,0,0]$
   (b) 0 与 $[3,3,1]$ 相乘, 累加进结果数组，得到 $[0,0,0,0]$
   (c) 1 与 $[3,3,1]$ 相乘, 累加进结果数组，得到 $[0,0,3,3,1]$
   (d) 结束得到结果 $[0,0,3,3,1]$

5. 将上述结果逆序输出，最高位的 0 不输出，得到 233，33，13300

7 实验七：割点和割边

7.1 题目描述

给定一个具有 N 个顶点和 M 条边的图，找出图中的割点和割边的数量。割点是指删除该顶点后，图会被分成多个连通分量。割边是指删除该边后，图会被分成多个连通分量。

7.2 算法流程

用 Tarjan 算法求解图中的割点和割边，DFS 得到深搜顺序 dfn，同时记录 low(从当前点出发能到达的点的 dfn 的最小值)，根据 dfn 和 low 来判别割点和割边，即假设对于 A、B 两个节点，$dfn[A] \leq low[B]$，说明 A 与 B 只可能按 dfs 序连接，也就说 A、B 之间的边为割边并且 A 是割点。同时判断根节点的度，如果度大于 1 则是割点。

具体实现思路：

1. 初始化：为每个节点设置一个初始的 DFS 序号为 0，并将所有节点标记为未访问状态
2. 从图中的任意未访问节点开始进行深度优先搜索（DFS）
3. 对于当前节点 v，设置其 DFS 序号为当前的 DFS 计数，设置其低链接值（low-link value）为其 DFS 序号，并将其标记为已访问
4. 对于当前顶点 v 的每个邻接顶点 u, 如果 u 未被访问，则进行递归 DFS，然后更新当前节点 v 的低链接值为 $min(v\text{的低链接值}, u\text{的低链接值})$
5. 进行如下判断：
   (a) 当前顶点 v 是图的根节点，并且度大于 1, 则 v 是一个割点
   (b) 对于当前节点 v 的每个邻居节点 u，如果 u 的低链接值大于等于 v 的 DFS 序号，则 (v, u) 是一个割边, 且 v 是一个割点
6. 重复步骤 2—5，直到图中的所有节点都被访问
7. 输出统计的割点和割边的数量

7.3 复杂度分析

该算法只需要一次深搜就能解决问题，如果是稀疏图，深搜的复杂度可以用节点的个数 n 和边的个数 m 表示，其 $Work = Span = O(n+m)$, 故整个算法也是 $Work = Span = O(n+m)$

算法需要邻接矩阵存储图结构，两个长度为 n 的数组 dfn、low 来保存深搜顺序和回溯节点，故总的空间复杂度为 $O(n^2)$

7.4 样例分析

测试样例如下图所示, 它是一张有 5 个节点，4 条边的图

#####样例输入1

```
5 4
1 2
1 3
1 4
1 5
```

算法过程如下：

1. 我们假设从编号为 1 的节点开始 DFS
2. 当前序号为 (1)，编号为 1 的节点的 dfn 为 (1)，同时 low 也为 (1)，假设按照编号顺序对其未访问过的邻接节点递归地进行 DFS
3. 当前序号为 (2)，编号为 2 的节点的 dfn 为 (2)，同时 low 也为 (2)，DFS 返回
4. 当前序号为 (3)，编号为 3 的节点的 dfn 为 (3)，同时 low 也为 (3)，DFS 返回
5. 当前序号为 (3)，编号为 3 的节点的 dfn 为 (3)，同时 low 也为 (3)，DFS 返回
6. 当前序号为 (3)，编号为 3 的节点的 dfn 为 (3)，同时 low 也为 (3)，DFS 返回
7. 序号为 (1) 的节点的 low 仍为 (1)，进行判断发现点 1 为割点，边 1−2, 1−3, 1−4, 1−5 为割边

8 实验八：静态区间查询

8.1 题目描述

给定一个长度为 N 的数列，和 M 次询问，求出每一次询问的区间内数字的最大值。输入为 N 和 M 以及数列以及 M 个查询的左右区间值，输出为分别查询出来的最大值。

8.2 算法流程

对于静态区间查询最值问题我们可以使用 ST 算法，做法是初始化一个 ST 表，$st[i][j]$ 表示从 i 到 2^{j-1} 范围内的最大值，则可以构建递归表达式

$$st[i][j] = maxst[i][j-1], st[i + (1 << (j-1))][j-1]$$

由于数据规模满足 $N \leq 1000$, 只需要初始化一个大小为 st[1000][10] 的数组
具体实现思路：

1. 将输入存入一个长度为 1000 的数组 A
2. 初始化一个大小为 st[1000][10] 的数组
3. 初始化 ST 的第一列，即将 A 中的每个元素复制到 ST 的第一列
4. 从第二列开始，每一列的元素值通过表达式

   $$st[i][j] = maxst[i][j-1], st[i + (1 << (j-1))][j-1]$$

   来计算
5. 对于每个查询
   (a) 找到最大的 j，满足 $2^j <= len = Right - Left + 1$
   (b) 最大值为 $max(ST[L][j], ST[R - 2^j + 1][j])$
6. 输出统计的割点和割边的数量

8.3 复杂度分析

由于创建 ST 表时，每一列只跟前一列有依赖关系，每一列的 n 个元素是可以并行计算的，一共有 lgn 列故创建 ST 表过程的 $Work = O(nlgn), Span = O(lgn)$

对于每个查询完全可以并行，且创建好 ST 表之后只需要常数级的查询代价, 故查询过程的 $Work = O(m), Span = O(1)$

算法总的 $Work = O(nlgn), Span = O(lgn)$

ST 表的规模为 nlgn，故算法的空间复杂度为 O(nlgn)

8.4 样例分析

测试样例如下图所示, 它对一个长度为 8 的区间进行了 8 次查询

#####样例输入1

```
8 8
9 3 1 7 5 6 0 8
1 6
1 5
2 7
2 6
1 8
4 8
3 7
1 8
```

算法过程如下 (只给出创建 ST 表和第一个查询的过程):

1. 初始化 ST 表的第一列为 [9 3 1 7 5 6 0 8]
2. 根据递推式计算得到 ST 表的第二列为 [9 3 7 7 6 6 8 8]
3. 根据递推式计算得到 ST 表的第三列为 [9 7 7 7 6 8 8 8]
4. 根据递推式计算得到 ST 表的第四列为 [9 7 7 7 8 8 8 8]
5. 对于第一个查询 $[1, 6]$, 只需比较 st[1][2] 和 st[3][2] 得到 9

9 实验九：素性测试

9.1 题目描述

给定一个数 N，判断它是否是素数

输入：一行，为一个任意大的正整数 N，满足 N≥2

输出：如果是素数输出 True，否则输出 False

9.2 算法流程

使用 Miller-Rabin 米勒罗宾算法来判断是否为质数。考虑到对于一个质数 p,$a^{p-1} mod\, p = 1$(费马小定理), 将其与二次探测定理结合，将 $n-1$ 分解为 2 的幂与其他数 u 的积，

对 a 求 $a^u \mod n = v$, 若 $a^{u*2^s} = n-1 \mod n$ 则 $true$，否则 $false$

根据情况选择 a 进行检验：

- if n < 2,047, it is enough to test a = 2;
- if n < 1,373,653, it is enough to test a = 2 and 3;
- if n < 9,080,191, it is enough to test a = 31 and 73;
- if n < 25,326,001, it is enough to test a = 2, 3, and 5;
- if n < 3,215,031,751, it is enough to test a = 2, 3, 5, and 7;
- if n < 4,759,123,141, it is enough to test a = 2, 7, and 61;
- if n < 1,122,004,669,633, it is enough to test a = 2, 13, 23, and 1662803;
- if n < 2,152,302,898,747, it is enough to test a = 2, 3, 5, 7, and 11;
- if n < 3,474,749,660,383, it is enough to test a = 2, 3, 5, 7, 11, and 13;
- if n < 341,550,071,728,321, it is enough to test a = 2, 3, 5, 7, 11, 13, and 17.

具体实现思路：

1. 输入一个待检测的正整数 n
2. 将 $n-1$ 表示为 $2^s * d$，其中 d 是一个奇数
3. 选择相应的 a
4. 计算 $x = a^d \, mod\, n$
5. 如果 x 等于 1 或者 x 等于 n-1，则跳转到步骤 9

6. 重复执行 s-1 次以下步骤
    (a) 计算 $x = x^2 \ mod \ n$
    (b) 如果 x 等于 1，则返回结果为 $false$
    (c) 如果 x 等于 n-1，则跳转到步骤 9
7. 返回 $false$
8. 重复步骤 3 到步骤 7，执行 k 次，其中 k 是一个参数，用于控制算法的准确性
9. 返回 $true$

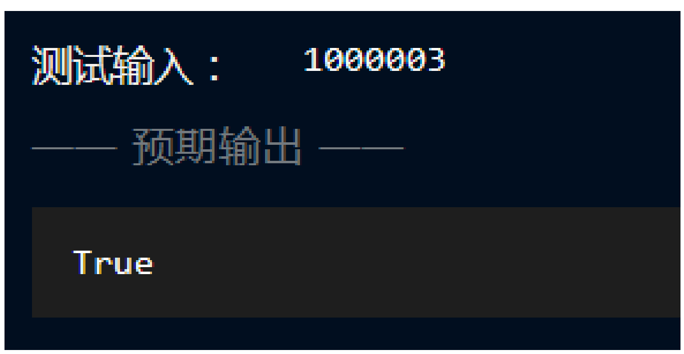
9.3 复杂度分析

费马小定理判断过程中，求快速幂和分解 n-1 的算法，$Work = Span = O(lgn)$，二次判断的过程，只需要常数级 $Work, Span$, 一共 $O(lgn)$ 次这样的判断，故该过程 $Work = Span = O(lgn)$，所以算法总的 $Work = Span = O(lgn)$

算法所需的空间复杂度为常量级 $O(1)$

9.4 样例分析

测试样例如下图所示, 它检验 1000003 是否为素数



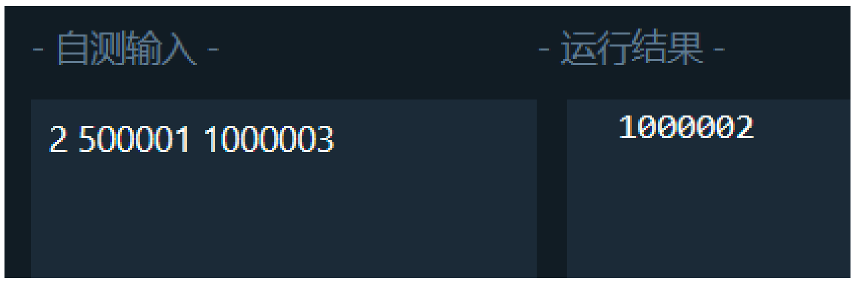
算法过程如下：

1. 输入 n=1000003

2. 分解 n-1，得到 $1000002 = 2^1 * 500001$

3. 选择 a=2,3

4. 计算 2^{500001} mod 1000003，得到 x=1000002，计算 3^{500001} mod 1000003，得到 x=1000002



5.

6.

7. 故返回 $true$