目录

1 无重复排序

```
val N = getInt();
val A = getIntTable(N);

fun merge([], ys) = ys : int list
  | merge(xs, []) = xs
  | merge(x::xs, y::ys) =
    if x < y then x::merge(xs, y::ys)
             else y::merge(x::xs, ys);

fun mergesort xs =
    let fun sort(0, xs) = ([], xs)
          | sort(1, x::xs) = ([x], xs)
          | sort(n, xs) =
                let val (l1, xs1) = sort((n+1) div 2, xs)
                    val (l2, xs2) = sort(n div 2, xs1)
                in (merge (l1,l2), xs2)
                end
        val (l, _) = sort (length xs, xs)
    in l end;

val B = mergesort(A);
printIntTable(B);
```

1.1 本关任务

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。

1.2 解法

我们可以采用归并排序算法解决。将数据分为即
复杂度为

$$W = O(n \log n) \quad S = O(\log^2 n)$$

Algorithm 1: MergeSort

```
1  MergeSort(⟨a_n⟩) =
2  if |a_n| = 1 then
3  |  a_1
4  else
5  |  let
6  |  |  L ⟵ MergeSort(⟨a_1, a_2, ..., a_{(n+1)/2}⟩);
7  |  |  R ⟵ MergeSort(⟨a_{(n+1)/2}, ..., a_n⟩);
8  |  in
9  |  |  Merge(L, R)
10 |  end
11 end
```

2 最短路

2.1 本关任务

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。

2.2 解法

可以采用 Dijkstra 算法解决该问题。

利用二维数组保存两点之间的路径长度，即 $A[i][j]$ 即是编号为 $i-1$ 和 $j-1$ 点之间的路径长度。若无给定长度，则用 ∞ 来表示无法两点之间无法到达。创建一个一维数组 dis，用以储存源点到每个点的距离。并且集合 B 为中只有源点 a。

(1) 在 dis 数组中寻找一个长度最短的值 dis[k]，并且保证 k 点不在集合 B 中。

(2) 再对每个不属于 B 集合中的点进行长度收缩: 如果 dis[i] 的值 dis[k]+A[a][k] 的大小，就将 dis[i] 的值更新为 dis[k]+A[a][k]。将 k 也加入集合 B 中。

(3) 重复上述步骤，直至集合 B 中有所有点。得到的数组 dis 的值就是源点到所有对应点的最短路径值。

在 SML 实际编写程序中，我用 99999 来替代 ∞，在比较长度的过程中也方便运算。在最后输出最短路之前，再将所有长度为 99999 的数据转化成 1 说明没有此路径。

2.3 分析与证明

Dijkstra 算法使用的迭代不超过 $n-1$ 次，在每次迭代的运算中，即比较与更新一共不超过 $2(n-1)$ 次，因此 $W = O(n^2)$。每一次的迭代都不能并行运算，但是在每一次的迭代中的比较与收缩长度操作可以同时进行，因此 $S = O(n) + 1$。

2.4 样例输入与输出

样例输入：

```
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
```

样例输出：

```
4 6 7 7 0 3 5
```

3 最大括号距离

3.1 本关任务

我们定义一个串 s 是闭合的，当且仅当它由'(' 和')' 构成，而且满足以下条件之一：

空串：这个串为空

连接：这个串由两个闭合的串连接构成

匹配：这个串是一个闭合的串外面包裹一个括号构成的

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。

3.2 解法

现在采用一个 stack 函数来储存括号的栈。

当栈为空时，判断下一个要入栈的括号的类型：如果是右括号，则必定不会有与之匹配的括号，直接丢弃该右括号；如果时左括号，就将左括号压栈。并在每次栈为空时更新当前最大的匹配括号长度。

当栈不为空时，如果栈顶元素是左括号，即将压栈的括号是右括号，就直接弹出栈顶元素，并增加当前括号距离 2。否则将新元素压栈。

由于这种解法只能够解决括号串整体匹配和右括号多于左括号的情况，因此需要将原来的括号串每个元素置换并颠倒，得到新的最大匹配长度不变但括号冗余情况完全相反的括号串。两个串同时经过 stack 函数的处理，得到的最小的值就是括号匹配的最大的距离。

关键函数 stack 的代码如下：

```
1 fun stack([], _, maxpair, tmppair) = Int.max(maxpair, tmppair)
2   | stack(x::xs, [], maxpair, tmppair) =
3         if(x=0)
4             then stack(xs, [x], Int.max(maxpair, tmppair), 0)
5             else stack(xs, [], Int.max(maxpair, tmppair), 0)
6   | stack(x::xs, y::ys, maxpair, tmppair) =
7         if((x=1) andalso (y=0))
8             then stack(xs, ys, maxpair, tmppair+2)
```

```
9 |             else stack(xs, x::y::ys, maxpair, tmppair);
```

3.3 分析与证明

字符串元素的置换和反转都需要 $O(n)$ 的 W 和 $O(1)$、$O(n)$ 的 S。

stack 函数只需要扫描整个 list 一遍，因此串行需要的 W 是 $O(n)$，S 也是 $O(n)$。

3.4 样例输入与输出

样例输入：

50 1 0 1 0 1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1 1 0 1 0 1 0 1 0 1 1 1 0 1 0 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0 0

样例输出：

6

4 括号匹配

4.1 本关任务

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。

4.2 解法

4.3 分析与证明

4.4 样例输入与输出

5 高精度整数

5.1 本关任务

给定两个任意精度的整数 a 和 b，满足 a≥b，求出 a+b, a−b, a×b 的值。

5.2 解法

5.2.1 加法的计算

将两个数列进行翻转，即将两个数的最后一位对齐，做相应的加法。

如果两个数相加大于 9，就将 c 置为 1，在下一位上加上 1。

Algorithm 2: Addition

```
1  a = (a_1 a_2 a_3 ... a_n)_10
2  b = (b_1 b_2 b_3 ... a_m)_10
3  带入该函数时需要将 a 和 b 进行翻转。
4  Addition(a, b, result, c, i) =
5  if (a_i = 0 andalso b_i = 0) then
6  |  result_i = c
7  else
8  |  let
9  |  |  result_i = a_i + b_i + c
10 |  |  c = (a_i + b_i + c) mod 10
11 |  in
12 |  |  Addition(a, b, result, c, i + 1)
13 |  end
14 end
```

5.2.2 减法的计算

减法的计算与加法同理，也是将最后一位对齐，从低位开始相减。并用 c 来表示在下一位是否要退位，如果 $a - b < 0$。算法如下。

Algorithm 3: subtraction

```
1  a = (a_1 a_2 a_3 ... a_n)_10
2  b = (b_1 b_2 b_3 ... a_m)_10
3  带入该函数时需要将 a 和 b 进行翻转。
4  Subtraction(a, b, result, c, i) =
5  if (a_i = 0 andalso b_i = 0) then
6  |  result_i = c
7  else
8  |  let
9  |  |  if a_i − b_i + c < 0 then
10 |  |  |  c = (a_i - b_i + c)
11 |  |  |  result_i = 10 + a_i − b_i + c
12 |  |  else
13 |  |  |  result_i = a_i − b_i + c
14 |  |  |  c = 0
15 |  |  end
16 |  in
17 |  |  Addition(a, b, result, c, i + 1)
18 |  end
19 end
```

5.2.3 乘法的计算

先将多位数相乘转换为多个一位数与多位数相乘的乘法。

与上述加法和减法的操作类似，从最末位开始相乘，并用 c 来记录下一位的进位数。

对于第 k 位数完成乘法运算后还要在末尾加上 $k-1$ 个零。

再将所有得到的结果相加。

```
fun onemultiply(a, [], c) = [c]
  | onemultiply(a, b::bs, c) =
        if (a * b + c> 9) then ((a*b+c)mod 10)::onemultiply(a, bs, (a*b
            +c) div 10)
                          else (a*b+c)::onemultiply(a, bs, 0);
fun multiply([], bs) = [0]
  | multiply(a::ass, bs) =
        Add(onemultiply(a, bs, 0), 0::multiply(ass, bs), 0);
```

5.3 分析与证明

对于加法来说:n 和 m 相加,需要线性扫描每一个数,因此 $W = O(max(|n|, |m|)+1)$。对于最优并行算法来说，可以用 map 操作对对应位数的两个数字相加，并记录每一位需要进位的个数。再将进位数相应相加。重复操作，直到所有每一位的进位数都为 0。所有的操作都是并行处理。

在第一次各位数相加时，

因此 $S = O(logn)$。

减法与加法同理，有同样的串行复杂度 W。

5.4 样例输入与输出

样例输入：

10 2 3 3 3 3 3 3 3 3 3 10 2 3 3 3 3 3 3 3 3 3

样例输出：

4 6 6 6 6 6 6 6 6 6 0 5 4 4 4 4 4 4 4 4 2 8 8 8 8 8 8 8 8 9

6 素性测试

6.1 本关任务

给定一个数 N，判断它是否是素数。

6.2 解法

利用 Miller-Rabin 算法来判断数字是否是素数。

费马小定理：当 p 为质数时，有 $a^{p-1} \equiv 1(mod\ p)$。

二次探测：如果 p 是一个素数，$0 < x < p$，则方程 $x^2 \equiv 1(mod\ p)$ 的解为 $x = 1$ 或 $x = p - 1$。

算法流程如下：

输入一个数 x。

(1) 当 x 是偶数或 2 时可以直接判断。

(2) 当 x 是奇数时，组一个比较小的质数 a, 计算 $2^s \cdot t = x - 1$。

(3) 先算出 a^t，然后进行 s 次平方进行二次探测。即如果在平方的过程中，当前数的平方模 x 与 1 同余并且该数不等于 1 或 $x-1$，那么 x 就是合数，直接退出循环。

(4) 根据费马定理判断 $a^{p-1} \equiv 1(mod\ p)$ 是否成立。如果成立，则从第 (2) 步重新挑选更大的质数 a 再次判断。

复杂度为

$$W = O(n \log n) \quad S = O(\log^2 n)$$

Algorithm 4: MergeSort

```
1  MergeSort(⟨a_n⟩) =
2  if |a_n| = 1 then
3  |  a_1
4  else
5  |  let
6  |  |  L ⟵ MergeSort(⟨a_1, a_2, ..., a_{(n+1)/2}⟩);
7  |  |  R ⟵ MergeSort(⟨a_{(n+1)/2}, ..., a_n⟩);
8  |  in
9  |  |  Merge(L, R)
10 |  end
11 end
```