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：ACM1701

学　　号：U201714722

姓　　名：计瀚东

指导教师：陆枫

报告日期：2018.11.18

计算机科学与技术学院

Lab1 括弧匹配实验

1. 实验要求

给定一个由括号构成的串，若该串是合法匹配的，返回串中所有匹配的括号对中左右括号距离的最大值；否则返回 NONE。左右括号的距离定义为串中二者之间字符的数量，即 *max { j - i + 1 | (si, sj)* 是串 s 中一对匹配的括号}。要求分别使用枚举法和分治法求解。

2. 实验思路

2.1 分治法求解思路

2.1.1、问题分析

根据观察我们可以发现，对任何一个括号串都可以用下面的结构来表示：

...)x)x)X(x(x(...

其中未匹配的右括号在“左侧”，未匹配的左括号在“右侧”，如上所示之间会有一些已经相互匹配的括号对，把它们用 x 表示，其中 X 如图表示最右未匹配右括号到最左未匹配左括号之间的部分。

综上使用五个量来描述一个括号串的信息：M,L,R,LLEN,RLEN：

（1）M：表示这个串内“中间位置”的括号距离最大值;

（2）L：没有匹配的右括号数量；

（3）R：没有匹配的左括号数量；

（4）LLEN：上面模型中 X 左侧的全部括号数（长度）；

（5）RLEN：上面模型中 X 右侧的全部括号数。

2.1.2、算法设计与算法正确性证明

使用 showt 过程将括号串以树的形式展开，并在每个节点提取两颗子树的状态（上面五个变量），并归并得到整体的状态。

（1）初态：

（a）空串：五个量均为 0;

（b）左括号：0，0，1，0，1；符合定义；

（c）右括号：0，1，0，1，0；符合定义；

（2）合并：对每个内部节点和根节点，假设我们已经返回了两颗子树的状态：

左子树 s1：M1，L1，R1，LLEN1，RLEN1

右子树 s2：M2，L1，R2，LLEN2，RLEN2

通过上面的信息我们需要推导出两颗子树合并之后的状态：

合并后：M，L，R，LLEN，RLEN

（a）R1 = L2，必然有

L = L1, R = R2, LLEN = LLEN1, RLEN = RLEN2

新产生的最长括号距离为 RLEN1+LLEN2，所以我们可以得知：

M = max(M1, M2, RLEN1+LLEN2)

（b）R1 > L2，必然有:

M = M1, L = L1，R = R2 + R1 - L2,

LLEN = LLEN1, RLEN = RLEN1 + length（s2）；

（c）R1 < L2，与情况（b）同理：

M = M2, L = L1 + L2 - R1, R = R2,

LLEN = length(s1) + X2, RLEN = RLEN2

（3）结果：最终我们得到 M，L，R，LLEN，RLEN，检查 L 和 R:

（a）L!=0 或 R！=0，此时语法不正确

（b）L=R=0，此时语法正确，按照上面的模型我们可以认为此时 LLEN=RLEN=0，即中间的 X 部分，而 M 表示中间 X 部分的最长括号距离，所以 M 可以表示语法正确时的最长括号距离。

3. 回答问题

3.1 关于枚举法求解

Task 5.2 (5%). What is the work and span of your brute-force solution? You should assume subseq has *O(1)* work and span

我的设计是，对任何一个括号串，如果它的首元素为左括号，则这个括号串的最长括号距离等于这个左括号与其匹配右括号的距离，或者尾串中的最长括号距离。对每个左括号要找到它匹配的右括号，我使用了迭代的设计，迭代有 work=span，所以有以下递归式：

$$W(n) = W(n-1) + \Theta(1) + \Theta(n)$$

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

解得Θ(n^2)，综上所述，这个算法计算了每一对括号之间的距离，并返回了其中最大的。

3.2 关于分治法求解

Task 5.4 (20%). The specification in Task 5.3 stated that the work of your solution must follow a recurrence that was parametric in the work it takes to view a sequence as a tree. Naturally, this depends on the implementation of SEQUENCE.

1. Solve the work recurrence with the assumption that $W_{showt} \in \Theta(\lg n)$ where n is the length of the input sequence.
2. Solve the work recurrence with the assumption that $W_{showt} \in \Theta(n)$ where n is the length of the input sequence.
3. In two or three sentences, describe a data structure to implement the sequence α seq that allows showt to have $\Theta(\lg n)$ work.
4. In two or three sentences, describe a data structure to implement the sequence α seq that allows showt to have $\Theta(n)$ work.

1、 $W(n) = 2W(n/2) + \Theta(\lg n) + \Theta(1)$ 根据算法导论 master theory, a = 2, b = 2, $\Theta(n) > \Theta(\lg n)$，根节点占据主体部分。 $W(n) = \Theta(n)$。
2、 $W(n) = 2W(n/2) + \Theta(n) + \Theta(1)$ 根据算法导论 master theory, a = 2, b = 2, $\Theta(n) = \Theta(n)$，属于平衡情况。 $W(n) = \Theta(n\lg n)$。
3、 使用保存了节点数目的 BST。 首先找到位于串中间的元素需要花费$\Theta(\lg n)$级别的时间，然后使用中间元素进行 split 操作即可把树分成两半，split 的 work 有$\Theta(\lg n)$。 综上，复杂度为$\Theta(\lg n)$。
4、 用一个保存了长度的双向链表就行了，如果不保存也可以花费$\Theta(n)$的时间计算一下。 已知长度为 n，take 和 drop 分别访问前后 n/2 的长度即可，这样的实现有复杂度$\Theta(n)$。

3.3 关于渐进复杂度分析

Task 6.1 (5%). Rearrange the list of functions below so that it is ordered with respect to O—that is, for every index *i*, all of the functions with index less than i are in big-O of the function at index *i*. You can just state the ordering; you don't need to prove anything.

1. $f(n) = n^{\log(n^2)}$
2. $f(n) = 2n^{1.5}$
3. $f(n) =(n^n)!$
4. $f(n) = 43^n$
5. $f(n) = lg(lg(lg(lg(n))))$
6. $f(n) = 36n^{52} + 15n^{18} + n^2$
7. $f(n) = n^{n!}$

5 < 2 < 6 < 1 < 4 < 7 < 3

Task 6.2 (15%). Carefully prove each of the following statements, or provide a counterexample and prove that it is in fact a counterexample. You should refer to the definition of big-O. Remember that verbose proofs are not necessarily careful proofs.
1. *O* is a transitive relation on functions. That is to say, for any functions *f*, *g*, *h*, if *f∈ O(g)* and *g∈ O(h)*, then *f∈ O(h)*.
2. *O* is a symmetric relation on functions. That is to say, for any functions *f* and *g* , if *f∈ O(g)* , then *g∈ O(f)* .
3. *O* is an anti-symmetric relation on functions. That is to say, for any functions *f* and *g* , if *f∈ O(g)* and *g∈ O(f)*, then *f=g*.

1、 正确，证明如下： f∈ O(g)所以存在 c1>0, n1>0 使对任意 n>n1， 0<f(n) < c1*g(n); g∈ O(h)所以存在 c2>0, n2>0 使对任意 n>n2， 0<g(n) < c2*h(n); 所以存在 c=c1*c1>0 使对任意 n>n1 且 n>n2， 0<f(n) < c*h(n); 所以 f∈ O(h)
2、 错误，反例如下： f(n)=lgn, g(n)=n 存在 c=1, n0=1，对任意 n>n0，存在 0<f(n)<c*g(n)，所以 f∈ O(g)。 对任意 c>0, n0>0, 都存在 n>n0，使 g(n)>c*f(n)，所以 g∈ O(h)不成立。
3、 错误，举出反例 f=2n, g=3n, 显然 f∈ O(g) and g∈ O(f)，但是 f=g 不成立。

Lab3 大整数运算

1、实验要求

实现n位二进制大整数的加法运算。输入a, b和输出s都是二进制位的串。要求算法的时间复杂度满足work=O(n)，span=O(log n)。

2、实验思路

2.1、加法计算

加法计算的难点在于保存进位信息，所以我们把结果分成两个部分：“朴素和”和“进位”；其中“朴素和”表示1+1，1+0，0+0三种情况，“进位”表示对高位的影响。

不难发现每一位的相加都会决定下一位是否有进位，这种“错位传递”的特点和scan的特点十分相似，所以我们利用”朴素和“配合scan来得到“进位”的信息。

最终我们只要将“朴素和”与“进位”相加，并根据是否有溢出来决定是否要在高位补1。

2.2、减法计算

我们知道x>y，所以结果的长度必然不会超过x，这使得实现十分简单；我们只需要对y进行取反加一得到去除符号位的补码，然后两数相加，通过舍弃溢出位实现模运算，就可以得到正确的差。

PS：关于去除首位零的方法慢慢想......

2.3、乘法计算

乘法计算中我们可以使用分治法的思想将$A*B$看作$(p2^{n/2}+q)*(r2^{n/2}+s)$，从而分解成更小的乘法，如果我们直接分解为：$pr2^n+(ps+rq)2^{n/2}+qs$则递归式为：

$$W_{**}(n)=4W_{**}(n/2)+O(n), W_{**}(n)=O(n^2)$$

我们发现这个递归树的分支太多了，如果把4减少则可以得到更低的复杂度。如果我们把乘式分解为$pr2^n+[(p+q)*(r+s)-pr-qs]2^{n/2}+qs$，乘法就减少到了3次，那么复杂度降低：

$$W_{**}(n)=3W_{**}(n/2)+O(n), W_{**}(n)=O(n^{log_2 3})$$

3、回答问题

3.1、加法计算

Task 4.1 (35%). Implement the addition function

++ : bignum * bignum -> bignum

in the functor MkBigNumAdd in MkBigNumAdd.sml. For full credit, on input with m and n bits, yoursolution must have $O(m+n)$ work and $O(lg(m+n))$ span. Our solution has under 40 lines with comments.

```
(*
 * 大整数相加
 * 1) 首先把两个bit串补为相同长度，方便后续计算；
 * 2) 然后使用carry串储存“朴素和”：
 *      1+1用GEN表示；1+0用PROB表示；0+0用STOP表示
 * 3) 然后在2的基础上使用scan对“朴素和”进行分析，使用的结合函数为：
 *      _, GEN => GEN      如果后一位是GEN则下一位必然会进位；
 *      _, STOP => STOP    如果后一位是STOP则下以为必然不会进位；
 *      some, PROP => some 如果后一位是PROB则它传递之前的进位情况；
 *    可以得出满足结合性；
 *    scan之后得到了每一位的进位情况：
 *      GEN表示进位，STOP表示不进位，PROP不会存在；
 *    多出的一位同时表示是否溢出；
 * 4) 然后把“朴素和”和“进位信息”进行map2可以得到结果：
 * 5) 最后看一下有没有溢出，如果就就在高位补充一个ONE就行了。
 *)
 fun x ++ y =
   case (length(x), length(y))
     of (0, 0) => empty()
      | (0, _) => y
      | (_, 0) => x
      | _ =>
   let
     (*1，高位补零使两串等长*)
     fun with0(a : bit seq, b : bit seq) =
       let val n = Int.max(length(a), length(b))
           val taila = tabulate (fn i => ZERO) (n - length a)
           val tailb = tabulate (fn i => ZERO) (n - length b)
       in (append(a, taila), append(b, tailb))
       end;
     (*2，得到“朴素和”*)
     fun getRawResult(x : bit seq, y : bit seq) =
       map2 (fn (i, j) => case (i, j)
                          of (ONE, ONE) => GEN
                           | (ZERO, ZERO) => STOP
                           | _ => PROP)
             x y;
     (*3，推导carry信息*)
```

```
    fun getCarryResult(x : carry seq) =
      scan (fn (i, j) => case (i, j)
                         of (_, GEN) => GEN
                          | (_, STOP) => STOP
                          | (some, PROP) => some)
            STOP x;
    (*4, 直接得到结果*)
    fun getResult(x : carry seq, y : carry seq) =
      map2 (fn (i, j) => case (i, j)
                         of (PROP, GEN) => ZERO
                          | (PROP, STOP) => ONE
                          | (_, GEN) => ONE
                          | (_, STOP) => ZERO
                          | (_, _) => raise BugInGetResult)
            x y;
  in
    let
      val (cx, cy) = with0(x, y)
      val rawResult = getRawResult(cx, cy)
      val (carryResult, high) = getCarryResult(rawResult)
      val result = getResult(rawResult, carryResult)
    in
      (*5, 判断高位是否溢出*)
      if high = GEN then append(result, singleton ONE)
      else result
    end
  end;
```

3.2、减法计算

Task 4.2 (15%). Implement the subtraction function

-- : bignum * bignum -> bignum

in the functor MkBigNumSubtract in MkBigNumSubtract.sml, where *x* -- *y* computes the number

obtained by subtracting *y* from *x*. We will assume that $x \geq y$; that is, the resulting number will always be non-negative. You should also assume for this problem that ++ has been implemented correctly. For full credit, if *x* has *n* bits, your solution must have $O(n)$ work and $O(\lg n)$ span. Our solution has fewerthan 20 lines with comments.

```
(*
  * 大整数相减
  * 条件：x和y均为正数且x>y。
  * 1）给y高位补零使x和y等长，便于计算；
  * 2）对y进行取反加一，不考虑符号位；
  * 3）x与y的补码模相加，结果消除多余零；
  * 以上三个步骤分别使用下面的三个“过程”表示
```

```
  *)
  fun x -- y =
    if length y = 0 then x else
    if length y = 1 andalso nth y 0 = ZERO then x else
    let
      (*1、补0使之等长*)
      fun sameLen(x : bit seq, y : bit seq) =
        let val tail = tabulate (fn i => ZERO) (length(x)-length(y))
        in append(y, tail) end;
      (*2、取补码*)
      fun trueToComp(x : bit seq) : bit seq =
        (map (fn i => if i = ONE then ZERO else ONE) x) ++ singleton ONE;
      (*3、模相加并消零*)
      fun compSub (x : bit seq, cy: bit seq) : bit seq =
        take(x ++ cy, length(x));
      (*4、判断相减后是否为零*)
      fun isZero (x : bit seq) =
        (reduce (fn (i,j) => if i = ZERO andalso j = ZERO then ZERO else
ONE)ZERO x) = ZERO;
      val ans = compSub(x, trueToComp(sameLen(x, y)))
    in
      if isZero(ans) then empty() else ans
    end;
```

3.3、乘法计算

Task 4.3 (30%). Implement the function

** : bignum * bignum -> bignum

in MkBigNumMultiply.sml. For full credit, if the larger number has n bits, your solution must satisfy $W(n) = W(n/2) + O(n)$and have $O(lg^2 n)$ span. You should use the following function in the Primitives structure:

val par3 : (unit -> 'a) * (unit -> 'b) * (unit -> 'c) -> 'a * 'b * 'c

to indicate three-way parallelism in your implementation of **. You should assume for this problem that++ and -- have been implemented correctly, and meet their work and span requirements. Our solutionhas 40 lines with comments.

```
 (*
  * 大整数相乘
  * 使用分治法可以将n级别的乘法分成4个n/2级别的乘法，但是利用类似Strenssen?矩阵乘法的
  * 的技巧，可以用“便宜的”加减来换乘法，从而只需要3个n/2级别的乘法，work由O(n^2)下降
  * 为O(n^lg3)；span由于乘法可并行，保持为O(n)
  *)
  fun x ** y =
```

```
    case (length(x), length(y))
      of (0, _) => empty()
       | (_, 0) => empty()
       | (1, _) => if nth x 0 = ZERO then empty() else y
       | (_, 1) => if nth y 0 = ZERO then empty() else x
       | _ =>
    let
      (*补零为等长*)
      fun with0(a : bit seq, b : bit seq) =
        let
          val len = Int.max(length a, length b)
          val taila = tabulate (fn _ => ZERO) (len - length a)
          val tailb = tabulate (fn _ => ZERO) (len - length b)
        in (append(a, taila), append(b, tailb))
        end;
      val (nx, ny) = with0(x, y)
      (*取得半长*)
      val half = length(nx) div 2
      val q = take(nx, half)
      val p = drop(nx, half)
      val s = take(ny, half)
      val r = drop(ny, half)
      (*3次并行乘法*)
      val (p1, p2, p3) =
          par3(fn _ => p ** r,
               fn _ => q ** s,
               fn _ => (p ++ q) ** (r ++ s))
      val mm = tabulate (fn _ => ZERO) (half*2)
      val m  = tabulate (fn _ => ZERO) half
    in
      append(mm,p1) ++ append(m,p3 -- (p1 ++ p2)) ++ p2
    end;
```

3.4、迭代计算复杂度分析

Task 5.1 (15%). Determine the complexity of the following recurrences. Give tight Θ-bounds, and

justify your steps to argue that your bound is correct. Recall that $f \in \Theta(g)$ if and only if $f \in O(g)$ and $g \in O(f)$. You may use any method (brick method, tree method, or substitution) to show that your bound is correct, except that you must use the substitution method for problem 3.

$T(n) = 3T(n/2) + \Theta(n)$

$T(n) = 2T(n/4) + \Theta(\sqrt{n})$

$T(n) = 4T(n/4) + \Theta(\sqrt{n})$(Prove by substitution)

（1）$T(n) = 3T(n/2) + \Theta(n)$

根据主方法，$\Theta(n^{log_2 3}) > \Theta(n)$，叶节点掌控，故$T(n) = \Theta(n^{log_2 3})$

（2）$T(n) = 2T(n/4) + \Theta(\sqrt{n})$

根据主方法，$\Theta(n^{log_4 2}) = \Theta(n^{1/2}) = \Theta(\sqrt{n})$，平衡态，故$T(n) = \Theta(\sqrt{n} lg(n))$

（3）$T(n) = 4T(n/4) + \Theta(\sqrt{n})$

不妨令$\Theta(\sqrt{n}) = c_0\sqrt{n} + d_0$

不妨假设若对$n = N/4$有$T(n) \leq c_1 n + c_2\sqrt{n} + d_1 = \Theta(n), c_1 > 0$

则$T(N) \leq 4(c_1\frac{N}{4} + c_2\sqrt{\frac{N}{4}} + d_1) + c_0\sqrt{N} + d_0$

不妨取$c_2 = -c_0, d_1 = -\frac{d_0}{3}$

则$T(N) \leq c_1 N + c_2\sqrt{N} + d_1$也满足$T(n) \leq c_1 n + c_2\sqrt{n} + d_1$

由于显然存在$n_0 > 0 \to c_1 n_0 + c_2\sqrt{n_0} + d_1$

所以以上递归式成立

$T(n) = \Theta(n)$

Lab8-范围搜索实验

2、回答问题

2.1、完成函数first和last，简述

思路：根据BST的性质，沿着左侧或者右侧递归下降就可以达到最小和最大的key所在节点。

```
(*左递归下降*)
fun first (T : 'a table) : (key * 'a) option =
  case Tree.expose T
    of NONE => NONE
     | SOME {key, value, left, right} =>
         case Tree.expose left
           of NONE => SOME(key, value)
```

```
               | _ => first(left);

    (*右递归下降*)
    fun last (T : 'a table) : (key * 'a) option =
      case Tree.expose T
        of NONE => NONE
         | SOME {key, value, left, right} =>
            case Tree.expose right
              of NONE => SOME(key, value)
               | _ => last(right);
```

测试样例：均通过

```
    val ordSet1 = % [5, 7, 2, 8, 9, 1]
    val ordSet2 = % [~5, ~7, ~2, 8, 9, 1]

    val testsFirst = [
      ordSet1,      测试一般情况
      ordSet2,      测试含负数
      % []          测试空集
    ]
    val testsLast = [
      ordSet1,
      % []
    ]
```

2.2、完成函数**previous**和**next**，简述

思路：对previous，如果一个节点右子树的key等于我们给出的k，那么左子树的last就是前驱节点；对next，如果一个节点左子树的key等于我们给出的k，那么右子树的first就是后继节点。

```
    (*前驱节点*)
    fun previous (T : 'a table) (k : key) : (key * 'a) option =
      case Tree.expose T
        of NONE => NONE
         | _ =>
            let
              val (left, _, _) = Tree.splitAt(T, k)
            in
              last left
            end;

    (*后继节点*)
    fun next (T : 'a table) (k : key) : (key * 'a) option =
      case Tree.expose T
        of NONE => NONE
         | _ =>
            let
              val (_, _, right) = Tree.splitAt(T, k)
            in
```

```
        first right
      end;
```

测试样例：均通过

```
    val testsPrev = [
      (ordSet1, 8),    测试中间一般情况
      (ordSet1, 1),    测试边缘
      (ordSet2, ~2),   测试含负数
      (% [], 8)        测试空集
    ]
    val testsNext = [ 同上
      (ordSet1, 8),
      (ordSet1, 9),
      (ordSet2, ~2),
      (% [], 8)
    ]
```

2.3、完成函数join和split，简述

思路：使用已有的轮子

```
    fun join (L : 'a table, R : 'a table) : 'a table =
      Tree.join(L, R)

    fun split (T : 'a table, k : key) : 'a table * 'a option * 'a table =
      Tree.splitAt(T, k)
```

测试样例：

```
    val testsJoin = [
      (ordSet1, % [100]), 测试长点集和单点集合并
      (ordSet1, % [3]),   单点集的点在长点集中间
      (ordSet2, % [100]), 测试含负数
      (ordSet1, ordset2), 测试来两个长点集合并
      (% [], % [100]),    测试空点集和单点集
      (% [], % ordSet2),  测试空点集和长点集
      (% [], % [])        测试两个空点集
    ]
    val testsSplit = [
      (ordSet1, 7),       测试切集合中的元素
      (ordSet1, 100),     测试不在集合中且完全“远离”集合的元素
      (% [], 7)           测试切空集
    ]
```

PS：其中join与union不同，不具有去重功能和要求；要求在调用时左树元素严格小与右树元素。

2.4、完成函数getRange，详叙

思路：对low和high边界分别使用split，再基于两次split的结果判断边界的key是否在table中存在，如果存在就把边界的singleton加入结果。

```
fun getRange (T : 'a table) (low : key, high : key) : 'a table =
  let
    val (_, m1, cut1) = split(T, low)
    val (cut2, m2, _) = split(cut1, high)
    fun complete(tree, m, board) =
      case m
        of NONE => tree
         | SOME(value) => join(tree, singleton(board, value));
  in
    complete(complete(cut2, m2, high), m1, low)
  end;
```

测试样例：均通过

```
val mySet1 = % [3, 2, 14, 13, 34, 8, 1, 60, 21, 5]

val testsRange = [
  (ordSet1, (5,8)),    边界是集合元素
  (ordSet1, (10,12)), 边界不是集合元素
  (ordSet1, (0,100)), 边界远远跨越集合元素
  (mySet1, (5,40)),    集合元素间间隔很大
  (% [], (5,8))        空集合
]
```

2.5、完成函数makeCountTable

思路：

1. 首先进行(x,y)排序，排序基于x，从左到右的点x从小到大；
2. 把点(x,y)映射为{(y,x)}后，使用scan join可以得到在x从小到大的基础上，比每个点靠左的点的集合；
3. 由于可能有多个点有相同的x，所以对每个x我们取最大的点集，以此来去重；
4. 最终可以得到每个x左边的点数目；

```
fun makeCountTable (S : point seq) : countTable =
  let
    val ps = Seq.sort (fn ((i, _), (j, _)) => compareKey(i, j)) S
    (* exchange position of x and y, *)
    val ss = Seq.map (fn (i, j) => singleton(j, i)) ps
    val yts = Seq.scani join (empty()) ss
    (* get pair of x * points sets whose points are left to x *)
    val xsyt = Seq.map2 (fn((x, _), t) => (x, t)) ps yts
    (* we collect x * point sets togehter, and keep only one point set, the
bigest*)
    val xcyts = Seq.collect compareKey xsyt
    val xcyt = Seq.map (fn (x, s) => singleton(x, Seq.nth s
(Seq.length(s)-1))) xcyts
```

```
    val xtyt = Seq.reduce join (empty()) xcyt
  in
    xtyt
  end;
```

复杂度分析：（含放大处理）

1. sort：$W = O(nlgn), S = O((lg^2 n)$;比较函数复杂度$O(1)$归并排序的复杂度
2. map：$W = O(n), S = O(1)$；映射函数复杂度为$O(1)$
3. scan join：$W = O(n), S = O(lg^2 n)$；scan的复杂度按照定义
4. map2：$W = O(n), S = O(1)$
5. collect：$W = O(nlgn), S = O(lg^2 n)$；处理的规模小于n，不妨放大为n，下同
6. map：$W = O(n), S = O(1)$
7. reduce：$W = O(n), S = O(lg^2 n)$
8. 总体：$W = O(nlgn), S = O(lg^2 n)$

   此外，考虑到中间结果，根据算法需要占用的最大空间可得出空间复杂度为$O(n^2)$

2.5、完成函数count，做相关分析

思路：

1. 根据countTable的定义，得到x1左侧的点集，使用getRange可以得到其中y1~y2的部分；
2. 同理得到x2左侧的点集，使用getRange可以得到其中y1~y2的部分；
3. 其中在计算x2左侧的点集时需要特别判断一下，如果split正好切中了x2，就不应该使用last(l)的点集合，而是x2的点集合；而x1就不会有这样的问题。

```
fun count (T : countTable) ((x1, y2) : point, (x2, y1) : point) :int  =
    if size(T) = 0 then 0 else
    let
      fun getsize t = size(getRange t (y1, y2))
      (*得到小于x1且纵坐标满足范围限制的点数*)
      fun getcount1(t : countTable, x1) =
        let
          val (l, m, _) = split(t, x1)
        in
          case last(l)
            of NONE => 0
             | SOME(_,yt) => getsize yt
        end;

      (*得到小于等于x2且纵坐标满足范围限制的点数*)
      fun getcount2(t : countTable, x2) =
        let
          val (l, m, _) = split(t, x2)
        in
          (*判断是否切中x2*)
```

```
            case m
              of SOME yt => getsize yt
               | NONE =>
                case last(l)
                  of NONE => 0
                   | SOME(_,yt) => getsize yt
          end;
        val c1 = getcount1(T, x1)
        val c2 = getcount2(T, x2)
      in
        c2 - c1   (*两者相减*)
      end;
```

复杂度分析：（含放大处理，设countTable规模为O(n)）

1. getRange ：$W = O(lgn), S = O(lgn)$，有限次split和join操作，下同
2. getcount1：$W = O(lgn), S = O(lgn)$
3. getcount2：$W = O(lgn), S = O(lgn)$
4. 总计：$W = O(lgn), S = O(lgn)$

测试样例：

```
    val points1 = % [(0,0),(1,2),(3,3),(4,4),(5,1)] 普通点集
    val points2 : point seq = % []                  空集
    val points3 = % [(10000,10000),(0,0)]           小而分散的点集
    val points4 = tabulate (fn i => (i,i)) 1000     大而整齐的点集
    val points5 = % [(~1,~1),(~2,~3),(~3,4),(0,0),(1,2),(2,3),(5,1)]含负数的普通
点集
    val points6 = % [(~1,~1),(0,0),(1,1)]           小而密集的含负数点集

    (points1, ((1,3),(5,1))),
    (points1, ((2,4),(4,2))),
    (points1, ((100,101),(101,100))),    靠近的两点
    (points2, ((0,10),(10,0))),
    (points3, ((0,10000),(10000,0))),    远离的两点
    (points4, ((0,500),(1000,0))),
    (points5, ((~1,~1),(3,3))),
    (points5, ((~2,5),(4,0))),
    (points6, ((~1,2),(0,~1))),
    (points6, ((0,0),(0,0))),            同一个点
```