CSE 2320 Lab Assignment 1

Due September 23, 2010

Goal:

Application of binary searches to determine (in a logarithmic number of steps) which element in a pair of monotonically increasing sequences has a particular rank.

Requirements:

1. Write a Java program that will read the two sequences and then the ranks. For each rank you should trace the binary search (one line per "probe") and then indicate which element of which sequence has the desired rank. The first line of the input file will give `m`, `n`, and `p` where `m` is the number of elements in the first sequence, `n` is the number of elements in the second sequence, and `p` is the number of ranks for which binary searches will be performed. The sequence elements will be in the range 0 . . . 999,999. The ranks will be in the range 1 . . . `m + n`.

2. Email your program to `huawang2007@mavs.uta.edu` by 9:15 a.m. on September 23, 2010.

Getting Started:

1. Your program should read the input files via shell redirection (e.g. `java lab1 < lab1.dat`). You should dynamically allocate tables for storing the input. To simplify the binary search code, it is recommended that the first sequence be stored in subscripts 1 . . . `m` of its table (`a`) and that subscripts 0 and `m + 1` store "low" and "high" sentinel values, respectively. The second sequence (`b`) would be stored similarly.

2. The rank of an element in one of the two sequences is the number of the position that it would occupy if the two sequences were merged into one monotonically increasing sequence. Should equal-valued elements appear, those in the first sequence are copied to the output before those in the second sequence. The following (linear-time) code will compute (and store) the ranks for all elements if stored as suggested in (1.):

```
static void ranksByMerge()
{
  int i,j,k;

  i=j=k=1;
  while (i<=m && j<=n)
    if (a[i]<=b[j])
      aRank[i++]=k++;
    else
      bRank[j++]=k++;
  while (i<=m)
    aRank[i++]=k++;
  while (j<=n)
    bRank[j++]=k++;
}
```

These tables of ranks will allow testing your code in an exhaustive fashion, e.g. for all ranks in the range 1 . . . `m + n`. These tables should NOT be computed in the version you submit.

3. For a given rank, the corresponding element could be in either one of the two input sequences. If stored as suggested in (1.), the following (symmetric) observations allow a binary search to be used:

   a. If the corresponding element is at `a[i]`, then there must be an index `j` such that all of the following hold:

      1. `i + j == rank,`
      2. `a[i] > b[j],` and
      3. `a[i] <= b[j+1]`

   b. If the corresponding element is at `b[j]`, then there must be an index `i` such that all of the following hold:

      1. `i + j == rank,`
      2. `a[i] <= b[j],` and
      3. `a[i+1] > b[j]`

4. The binary search may be coded to adjust the value of `i` within the search range. Be careful in initializing the range (e.g. `low` and `high`). Otherwise, `i` and `j` could reference slots outside of their respective tables.

5. The output line for each probe should include at least the values of `low`, `high`, `i`, and `j`. You may also include other debugging information, such as the result of the probe.

6. Recursion should not be used.

i	a[i]	aRank[i]
0	-99999999	0
1	1	2
2	1	3
3	1	4
4	1	5
5	2	7
6	5	12
7	6	14
8	8	17
9	8	18
10	9	22
11	99999999	0

j	b[j]	bRank[j]
0	-99999999	0
1	0	1
2	1	6
3	2	8
4	3	9
5	4	10
6	4	11
7	5	13
8	6	15
9	7	16
10	8	19
11	8	20
12	8	21
13	9	23
14	9	24
15	9	25
16	99999999	0