CSE332 Week 8 Section Worksheet

For problems 1 & 2 you don’t need to write pseudo-code – just describe what to do at the sequential cut-off step, and how to merge results.

* 1. Given a large array of integers, describe a fork-join program to compute the number of elements that are less than 7.

*When describing how to do a fork-join, you usually just need to specify how to handle the sequential cutoff step, and how to handle merging.*

*For this problem, once we hit the sequential cutoff we just count the number of 7’s in our portion of the array and return that. We merge cases by adding the left & right results together and returning that.*

* 1. Given a large array of strings (each element is a string of relatively small length), describe a fork-join program that counts the number of elements that exactly match the string “parallel”.

*This seems like a different problem than (a), but is really quite similar. For the sequential cutoff case we just count the number of strings that match “parallel” and return that count. The merging step is the same as (a); we just add the counts of left & right and return that. Both (a) & (b) are really just parallel-adds with special sequential cutoff parts.*

* 1. Given a large array of integers, return ‘true’ if there are an even number of even numbers, or ‘false’ if not. For instance, on [1, 7, 4, 3, 6] it would return ‘true’, as we have 2 even numbers, whereas on [6, 5, 4, 3, 2, 1] it would return ‘false’.

*We can also do this one like a parallel add; at the sequential cutoff we return the number of even numbers in our portion of the array. Merging is just adding left & right together. When we get the total number of even numbers (after the fork-join finishes), we return true if it is even, else return false. Note: you could also return a Boolean value as the result, instead of the count; it would just require a slightly different merge.*

1. Given an array that represents one enormous bit string (each element is a 0 or 1), describe a fork-join program that finds the length of the longest string of consecutive 1’s. For instance, [0,1,1,0,1,1,1,0] would return 3.

*This problem is quite a bit harder than problem #1. For this, one solution is to return a tuple containing a) how many 1’s are on my left border, b) the length of my longest string of 1’s, c) the # of 1’s on my right border and d) a Boolean: true if my whole portion consists of 1’s, false otherwise; I’ll write these as {a,b,c,d}. To compute the case at a sequential cutoff of 1, if the number is 0 we return {0,0,0,false}; if it’s 1 we return {1,1,1,true}.*

 *To merge left & right results (call them left: {a,b,c,d} and right: {e,f,g,h}), return {i,j,k,n} where:*

 *i= if d is true then a+e, else a*

 *j= max(b,f,c+e)*

 *k= if h is true then g+c, else g*

 *n= d && h*

*It’s difficult to just look at these equations and understand them, so to get a feeling how it works, I recommend trying them out on some bit strings.*

1. Write pseudo-code for a ForkJoin framework program that can traverse a balanced binary tree (but without the BST ordering property) to find whether a given element exists in the tree or not. This should have a log(n) span, provided that the tree is balanced.

*I won’t provide the pseudo-code, but conceptually this will be very similar to the usual fork-join program, except that instead of passing half of our portion of the array to the left child, and half to the right, we pass the left node to the left child, and the right node to the right. At each step we need to check whether the key at our current node is the desired key; return true if so. Else, if the node has no children, return false. Else return the OR of its children’s results (an empty child counts as ‘false’).*