Dylan Dunsheath HW #4

Problem 1

This program was rather simple algorithmically to solve, allowing me to sort unsigned integers with radix sorting. I started off by declaring a preprocessor directive (#define MAXSIZE 100) which is essentially a constant value that I will be using to ensure the count that is inputted by the user is NOT more than MAXSIZE (or less than 1; which really wouldn’t make much sense to input regardless since it will just print out that value).

Within the main function, I declared three variables at the beginning, TOTAL\_BITS (of unsigned int) that holds the value 32; since there are 8 bits in a byte and each integer is 4-bytes long: 4 \* 8 = 32 bits. I also declared ‘i’ which is a loop-variable for my for-loop and numElements which holds the inputted number of elements we want to sort. When the program runs, the program asks the user how many values they want to sort (as unsigned integers). If the user enters a number less than 1 or more than 100; we prompt the user again for input as the input isn’t considered valid (and do so continuously until they enter valid input).

Once the user enters a valid value for the amount of elements, we declare an array to hold that EXACT number of elements of unsigned int type. This made more sense logically rather than declaring an array that can hold a MAX of MAXSIZE and only looping up-to what we want because it wouldn’t make sense to have n array that is ‘larger’ or holds more elements than we specifically need (or the user wants). After declaring the array that holds the number of elements that we want, we loop from index 0 – numElements – 1 (or one less than the number of elements that was inputted) and get the unsigned integers from the user for sorting (ideally, they would input the values out of order to demonstrate our program works). Afterwards, I then declared several other variables: counterBucketOne, counterBucketTwo, temp and ‘totallyCoolCounter’ along with ‘d’ and then the two unsigned integer ‘buckets’ (arrays) of size numElements to hold certain values based on the bitwise operation.

The code then enters a loop that runs for ‘TOTAL\_BITS’ (0 - 31) iterations which represent the number of bits in the integer being sorted; according to the current ‘d’-ith bit and then another inner loop that increments over the original array. Depending on the value of [temp = 1 & (values[i] >> d);] which essentially is a operation that performs a bitwise operation on the i-th element of the array ‘values’ by ‘d’ bits (in the outer loop) and performs a bitwise AND to get a value of either 0 or 1.

In the code, if the bit is ‘1’ it adds the value to the 2nd bucket and increments the counter of this bucket. Otherwise, it adds the value to the 1st bucket and increments the counter for bucketOne. After all the bits are analyzed and the correct values are in the correct bucket, it joins the buckets; starting off with bucketOne and then bucketTwo (which ensures it’s added AFTER the last element from bucketOne), especially with the ‘totallyCoolCounter’ variable. I then initialize these variables: counterBucketOne, counterBucketTwo, and totallyCoolCounter to 0 to ensure that they are reset to their initial state and to ensure that the previous iteration doesn’t carry over (or gives undesirable results).

Finally, I loop through this sorted array of type unsigned integer and then the value (Note: it is important to understand that negatives will appear AFTER positives because they are of unsigned type and that -2 will be ‘less’ than -1 in its unsigned integer representation).

**Problem 2**

This problem involved sorting with real or floating-point numbers, which is slightly different from sorting with unsigned integers like in the previous problem and much different from sorting with signed integers.

Similar to the previous problem, I initialized a ‘preprocessor directive’ or constant with #define MAXSIZE 100 🡪 this is used to ensure that there aren’t more than 100 elements in an array.

Within the main function, I initially stated off by declaring I and j variables for my for-loop (the version I was running on MobaXTerm with AFS required them to be declared beforehand) and numElements which contains the number of elements that an array will be holding (floats/real numbers). I check if the input is within the bounds of 1-100 and if not, I prompted the user again to re-input the number of elements. I then initialized an array of type float that held the exact number of elements we need (as requested previously from the user), a variable called currentNum that let the user what value/element we are on (compared to how many we need) and an unsigned pointer called FtoInt which then typecasts our float array to unsigned integer as well (otherwise the pointer wouldn’t work). After that, I then prompted the user for the floating point values (from 0 – numElements - 1) which was put in my array. The message that was displayed also read what the number was according to the array(first element, second element, third element… all the way to element N).

Now we officially begin the radix sort. Firstly, it is important to understand that this radix sort is performed on an array of ‘unsigned integers’ (hence we have an outer loop that works on 32 bits or 4 bytes… from 0 - 31). I then declared a mask that left shifts 1 by ‘i’ (as shown in the loop we did) and a variable called zeroCount, to count the number of zeroes. Then, I declared an inner loop with ‘j’ that goes through all the elements in fToInt ‘pointer’ and performed a bitwise AND with the mask to check if the bit was a zero and if it is, I incremented zeroCoint by 1.

While still being in the outer loop that goes from 0 – 31, I declared an array of type temp that also holds the exact number of elements, zeroIndex and oneIndex (which both represent the number of the position that the 0 bit and 1 bit will be placed). I then declared another inner loop that runs from 0 to the number of elements – 1 (so the number of elements is how long the loop iterates) that performs a bitwise AND on fToInt with the mask. If it’s 0, it places it in the temp array at the current position based on zeroIndex OR the bit of 1 at the oneIndex (both are incremented afterwards). The overall comparison is done based on the binary representation of the mask AND the binary of the fToInt value.

After we break out of the inner loop from 0 to 31, we moved the values from the temp array BACK to the unsigned pointer that is pointing to the float array and break out of the outer loop.

At this point, we have a new temporary array called tmp that holds the rearranged numbers since the array isn’t necessarily in sorted order (bits of ‘1’ are after bits of ‘0’). The tmp array allows us to re-sort the values in the order we want and place the negatives in front (and tmpIndex to keep track of where we are). We start AT the end because that’s where the negatives are in the fToInt array, the array, which mind you was changed after radix sorting based on bits…. And decrement ‘i’. If the value from I is negative, get the binary representation from fToInt and pass it to tmp (and increment the index of tmpIndex). Otherwise, we have another loop that starts at the BEGINNING and gets all the positives in ASCENDING order and places them ALSO in the tmp array at a specified index (which holds a different value at this point depending on the negatives).

So essentially, the temp array is what holds the radix sort that we did (where positives are in ASCENDING and the negatives are DESCENDING 🡪 -5.4, -3.2, -1.1…) and passes it to the unsigned pointer OF the float array. And as previously mentioned, I had to re-sort the values by going FROM the end to the ‘beginning’ and check if the values are negative and IF they are, place them in a tmp array. I then also had to start at the BEGINNING towards the END and check if the values are POSITIVES and place them also in the tmp array. Finally, I had a loop that placed the FINAL sorted values in the fToInt array (which again, points to the float array) and then I just had to print this float array out.

It performs a bitwise AND