Summer 2020 COP 3502 Section 2 Final Exam - Part D Solution

1) (5 pts) Consider the problem of finding the mode (most frequently occurring value) in an array. John attempts to solve the problem recursively. His strategy is to recursively call his mode function on the left half of the array and the right half of the array, and then use these answers to calculate the mode of the whole array. Assuming that his function only returns the mode and nothing else, why is his strategy doomed to fail?

Solution

If we know the mode of the left half of the array, and we know the mode of the right half of the array, that is not enough information to determine the mode of the array. It's possible that the mode is neither value. Consider the following example of an array of size 10:

3 3 3 2 2 2 2 4 4 4

The mode of the left half of the array is 3 and the right half of the array is 4, but the mode of the whole array is 2. There is no way to determine this by only knowing that the mode of the left is 3 and that the mode of the right is 4. At a minimum, one would have to know the frequencies of all the values on the left and the right to make an accurate determination of the mode.

Grading: Full credit only if they explain why knowing the modes of the left and right is insufficient in calculating the mode of the whole array. Give partial credit accordingly.

2) Sorting

(a) (5 pts) In a Merge Sort of 8 elements, the Merge function gets called 7 times. Consider a Merge Sort being executed on the array shown below. What does the array look like right **AFTER** the sixth call to the Merge function completes?

index	0	1	2	3	4	5	6	7
value	40	27	12	18	11	99	31	16

Index	0	1	2	3	4	5	6	7
Value	12	18	27	40	11	16	31	99

Grading: 5 pts for the correct answer, 3 pts for showing the array after any of the merges except the last two, 2 pts for showing each pair sorted, 0 otherwise.

(b) (5 pts) Consider sorting the array below using a bubble sort. Show the contents of the array after each iteration of the algorithm completes. (Note: after the first iteration, the maximum array value should be in its correct spot.)

Index	0	1	2	3	4	5
Original	33	18	22	9	15	14
After 1st	18	22	9	15	14	33
After 2nd	18	9	15	14	22	33
After 3rd	9	15	14	18	22	33
After 4th	9	14	15	18	22	33
After 5th	9	14	15	18	22	33

Grading: 1 pt per row, row must be fully correct to get the point.

3) (10 pts) For the purposes of this question, a permutation of size n is any ordering of the integers 0, 1, 2, …, n-1. We define a spaced-out permutation of size n to be a permutation such that two consecutive terms in the permutation differ by at least 2. For example, [0, 2, 4, 1, 3] is a spaced out permutation of size 5, and [5, 2, 4, 0, 3, 1] is a spaced out permutation of size 6, but [3, 0, 2, 1] is not a spaced out permutation since the 2 and 1 are adjacent and differ by only 1. Fill in the blanks below so that all spaced out permutations of size N are printed out. (Note: You may use the abs function from the math library. The function takes in a single integer and returns its absolute value.)

```
#include <stdio.h>
#include <math.h>
#define N 6

int main(void) {
    int perm[N], used[N];
    for (int i=0; i<N; i++) used[i] = 0;
    printSpaced(perm, 0, used);
    return 0;
}

void printSpaced(int perm[], int k, int used[]) {
    if (k == N) {
        for (int i=0; i<N; i++) printf("%d ", perm[i]);
        printf("\n");
        return;
    }
    for (int i=0; i<N; i++) {
        if ( k==0 || (!used[i] && abs(perm[k-1]-i)>=2 ) ) {
            used[i] =  1;
            perm[k] =  i;
            printSpaced( perm, k+1, used );
            used[i] = 0;
        }
    }
}
```

Grading: 1 pt each slot except big, that's 4 pts(1 used, 3 abs)