Honors Computer Science I (COP 3502) Exam #2 Solutions Date: 3/26/2026

1) (8 pts) Show the state of the array below after each iteration of Insertion Sort:

16	18	13	2	11	27	14	5	22	9
16	18	13	2	11	27	14	5	22	9
13	16	18	2	11	27	14	5	22	9
2	13	16	18	11	27	14	5	22	9
2	11	13	16	18	27	14	5	22	9
2	11	13	16	18	27	14	5	22	9
2	11	13	14	16	18	27	5	22	9
2	5	11	13	14	16	18	27	22	9
2	5	11	13	14	16	18	22	27	9
2	5	9	11	13	14	16	18	22	27

Grading: 1 pt per row, row has to be completely correct to get the point.

2) (4 pts) Why does Quick Sort, on average, run faster than Merge Sort, in practice?

The partition function, which Quick Sort uses works in place, meaning that during the whole running of the algorithm, no extra memory (well very little anyway) needs to be allocated whereas in Merge Sort, for every Merge, an entire extra array the size of the two merged arrays has to be allocated. **Grading: Grader's discretion!**

3) (6 pts) Consider running Merge Sort on the array below (of size 8). Over the course of the algorithm, 7 Merge operations occur. Show the state of the array after each of the first six merge operations:

Array	13	16	18	4	11	1	19	5
1st Merge	13	16	18	4	11	1	19	5
2nd Merge	13	16	4	18	11	1	19	5
3rd Merge	4	13	16	18	11	1	19	5
4th Merge	4	13	16	18	1	11	19	5
5th Merge	4	13	16	18	1	11	5	19
6th Merge	4	13	13	18	1	5	11	19
7th Merge	1	4	5	11	13	16	18	19

Grading: 1 pt per row, whole row has to be correct to get the point

4) (9 pts) Write down the solutions to each of these recurrence relations (Big-Oh bounds) using the Master Theorem: **Grading: 3 pts all or nothing except for (b), 1 pt for log form**

(a) $T(n) = 4T\left(\frac{n}{2}\right) + O(n^3)$ $\quad T(n) = O(n^3)$

(b) $T(n) = 8T\left(\frac{n}{4}\right) + O(n)$ $\quad T(n) = O\left(n^{\frac{3}{2}}\right), or\ O(n\sqrt{n})$

(c) $T(n) = 9T\left(\frac{n}{3}\right) + O(n^2)$ $\quad T(n) = O(n^2 lgn)$

5) (12 pts) Solve the following recurrence relation using the iteration technique shown in class. Please obtain a Big-Oh bound in terms of n for the recurrence relation.

$$T(n) = 8T\left(\frac{n}{2}\right) + O(n^2), \text{ for n} > 1$$
$$T(1) = 1$$

$\boldsymbol{T(n) = 8T\left(\frac{n}{2}\right) + cn^2}$

$T(n) = 8(8T\left(\frac{n}{4}\right) + c\left(\frac{n}{2}\right)^2) + cn^2$

$T(n) = 8(8T\left(\frac{n}{4}\right) + c(\frac{n^2}{4})) + cn^2$

$T(n) = 64T\left(\frac{n}{4}\right) + 2cn^2 + cn^2$

$\boldsymbol{T(n) = 64T\left(\frac{n}{4}\right) + 3cn^2}$

$T(n) = 64(8T\left(\frac{n}{8}\right) + c\left(\frac{n}{4}\right)^2) + 3cn^2$

$T(n) = 64(8T\left(\frac{n}{8}\right) + c(\frac{n^2}{16})) + 3cn^2$

$T(n) = 512T\left(\frac{n}{8}\right) + 4cn^2 + 3cn^2$

$\boldsymbol{T(n) = 512T\left(\frac{n}{8}\right) + 7cn^2}$

The first three iterations are bolded. More generally, after k iterations, we have:

$T(n) = 8^kT\left(\frac{n}{2^k}\right) + (2^k - 1)cn^2$

Let $\frac{n}{2^k} = 1$. We want to substitute for the value of k that makes this true. If $2^k = n$, then since $2^3 = 8$, it follows that $(2^k)^3 = n^3$, so $8^k = n^3$. Substituting for this value of k we get:

$\boldsymbol{T(n) = n^3T(1) + (n-1)cn^2 = n^3 + cn^3 - cn^2 = O(n^3)}$

Grading: 1 pt first iteration
2 pts second iteration
2 pts third iteration
3 pts general guess
1 pt value to plug in for k
2 pts figuring out that $8^k = n^3$ when $2^k = n$
1 pt to finish it

6) (12 pts) A common function to run on a binary search tree is the lower function, which returns a pointer to the node storing the largest value in a binary search tree strictly less than a given input value. If no such value exists, then the function should return NULL. Write an implementation of this function using the struct shown below:

```
typedef struct bstnode {
    int data;
    struct bstnode* left;
    struct bstnode* right;
} bstnode;

bstnode* lower(bstnode* root, int key) {

    // No answer.
    bstnode* res = NULL;                                    // 1 pt

    // Go down tree.
    while (root != NULL) {                                  // 1 pt

        // We want to go right to see if anything is bigger.
        if (root->data < key) {                             // 2 pts
            if (res == NULL || root->data > res->data)      // 3 pts
                res = root;                                 // 1 pt
            root = root->right;                             // 1 pt
        }

        // If we want to update our answer, we must only go left.
        else                                                // 1 pt
            root = root->left;                              // 1 pt
    }

    // Ta da!
    return res;                                             // 1 pt
}
```

7) (20 pts) Consider the following problem: given a list of integers, imagine inserting either plus signs or times signs between each integer, how many different values could the possible expression form? We can solve this problem by trying each possible combination of signs, and for each combination of signs, evaluating the expression, and marking, in a boolean array, each value produced. Then, we could just count the number of entries equal to 1 in the boolean array. On the next 2 pages is a solution to the problem which runs a small test case with two functions left empty for you to complete. Complete those functions.

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

int total(int* possible, int len);
void go(int* signs, int k, int* values, int* possible, int n);
int eval(int* signs, int* values, int* possible, int n);

int main() {
    int values[N+1] = {2,3,1,2,4,3};
    int prod  = 1;
    for (int i=0; i<=N; i++) prod *= values[i];
    int max = prod + N + 1;
    int* possible = calloc(max, sizeof(int));
    int* signs = calloc(N, sizeof(int));
    go(signs, 0, values, possible, N);
    printf("possible = %d\n", total(possible, max));
    free(possible);
    free(signs);
    return 0;
}

// Returns the number of values in the array possible equal to 1.
// The length of the array possible is len.
int total(int* possible, int len) {

    int res = 0;                            // 1 pt
    for (int i=0; i<len; i++)               // 1 pt
        res += possible[i];                 // 1 pts
    return res;                             // 1 pt
}

// Fills in the possible array with all possible values that can be
// formed given that the first k signs are fixed. signs[i] = 0 indicates
// that the ith sign is an addition sign. signs[i] = 1 indicates that
// the ith sign is a multiplication sign.
void go(int* signs, int k, int* values, int* possible, int n) {

    if (k == n) {                                           // 1 pt
        possible[eval(signs,values,possible,n)] = 1;        // 6 pts
        return;                                             // 1 pt
    }

    signs[k] = 0;                                           // 1 pt
    go(signs, k+1, values, possible, n);                    // 3 pts
    signs[k] = 1;                                           // 1 pt
    go(signs, k+1, values, possible, n);                    // 3 pts
    signs[k] = 0;                                           // 0 pts

}
```

```
int eval(int* signs, int* values, int* possible, int n) {

    int res = 0, curProd = values[0];
    for (int i=0; i<n; i++) {
        if (signs[i] == 0) {
            res += curProd;
            curProd = values[i+1];
        }
        else {
            curProd *= values[i+1];
        }
    }

    return res + curProd;
}
```

NOTE: Posted code has the possible parameter removed from the eval function and the corresponding changes. For the written exam, I'll allow either answer with or without the parameter.

8) (8 pts) Evaluate the following postfix expression, showing the contents of the stack as the expression is being evaluated at each of the different points shown (A, B and C) in the expression:

						A			B		C					
3	5	+	2	6	3	/	*	2	3	+	*	4	/	–		

3
6
2
8
Stack at A

2
4
8
Stack at B

5
4
8
Stack at C

Value of Expression = **3**

Grading: 2 pts for each stack, 2 pts for the answer

9) (20 pts) Consider using a linked list to store big integers, digit by digit. The linked list stores the digits in reverse order. For example, 1847 would be stored in the list → 7 → 4 → 8 → 1. The reason for this storage is it makes most mathematical operations easier to compute. Given the struct definition below, write an add method that takes in pointers to two nodes, and adds the two numbers represented storing the new number as a newly allocated linked list of nodes and returns

a pointer to the list storing the corresponding sum. **Note: since no carry value is passed into the function, you have to solve it iteratively.**

```
typedef struct node {
    int digit;
    struct node* next;
} node;

node* add(node* op1, node* op2) {

    node* res = NULL;                                   // 3 pts init
    node* back = NULL;
    int carry = 0;

    while (op1 != NULL || op2 != NULL) {                // 2 pts

        // Get two digits to add.
        int d1 = op1 != NULL ? op1->digit : 0;          // 5 pts
        int d2 = op2 != NULL ? op2->digit : 0;          // getting digit and
                                                        // carry
        // Figure out current digit and new carry.
        int tot = (d1+d2+carry)%10;
        carry = (d1+d2+carry)/10;

        node* tmp = malloc(sizeof(node));               // 2 pts
        tmp->digit = tot;
        tmp->next = NULL;

        if (res == NULL) {                              // 3 pts case
            res = tmp;
            back = tmp;
        }
        else {                                          // 2 pts case
            back->next = tmp;
            back = tmp;
        }

        if (op1 != NULL) op1 = op1->next;               // 2 pts
        if (op2 != NULL) op2 = op2->next;
    }

    if (carry > 0) {                                    // 1 pt for all of this
        node* tmp = malloc(sizeof(int));
        tmp->digit = carry;
        tmp->next = NULL;
        back->next = tmp;
    }

    return res;
}
```

10) (1 pt) The Miami Open, a tennis tournament, is currently being held at the grounds of Hard Rock Stadium. What company owns the naming rights to the stadium?

Hard Rock (Give to All)