Spring 2022 COP 3502 Section 2 Final Exam Solutions
Date: 4/29/2022

1) (8 pts) Anisa and Ben are trying to pick a name for their child. Based on their review of pseudo-science, they've decided that they want the shortest name possible. If there are multiple shortest names, they want the one that comes first alphabetically. Write a function that takes in a list of possible names (type char**) and returns the name (char*) that they should choose amongst the possible names in the input array, according to their criteria. Please use the appropriate functions in string.h and do NOT allocate memory, free memory, OR print anything. (You'll get an automatic 0 if you do any of these things.) You may assume that the strings stored in possibleNames only consist of lowercase letters. (Note: numNames represents the number of names stored in the array possibleNames.)

```
#include <string.h>

char* getBestName(char** possibleNames, int numNames) {

    int minLen = strlen(possibleNames[0]), i;   // Set up 2 pts
    char* res = possibleNames[0];

    for (i=1; i<numNames; i++) {                // Loop 1 pt

        int newLen = strlen(possibleNames[i]);  // if len 1 pt
                                                // if =len, cmp 2 pts
        if (newLen < minLen ||
           (newLen == minLen && strcmp(possibleNames[i], res) < 0)) {
            minLen = newLen;
            res = possibleNames[i];             // 1 pt updates in if
        }
    }

    return res;                                 // 1 pt return

}
```

2) (5 pts) Convert the number 838 in base 10 to base 3.

```
3 | 838
3 | 279 R 1
3 |  93 R 0
3 |  31 R 0
3 |  10 R 1
3 |   3 R 1
3 |   1 R 0
3 |   0 R 1
```

Reading the remainders backwards we get $\mathbf{1011001_3}$

Grading: 1 pt off per error, cap at 0/5.

3) (10 pts) Solve the following recurrence relation **exactly** (no Big-Oh answer) in terms of n:

$$T(n) = 3T(n-1) + 3^n, \text{ for all integers } n > 1$$
$$T(1) = 2$$

Your answer should be of the form $3^a(bn - c)$, where a is a function of n, and b and c are constants.

Iterate the formula:

$T(n) = 3T(n-1) + 3^n$ **Grading : 1 pt**

$= 3[3T(n-2) + 3^{n-1}] + 3^n$

$= 9T(n-2) + 3^n + 3^n$

$= 9T(n-2) + 2(3^n)$ **Grading: 2 pts**

$= 9[3T(n-3) + 3^{n-2}] + 2(3^n)$

$= 27T(n-3) + 3^n + 2(3^n)$

$= 27T(n-3) + 3(3^n)$ **Grading: 2 pts**

The results after 2 and 3 iterations respectively are underlined. Based on the first three iterations, we make the following guess for the recurrence after iterating k steps:

$T(n) = 3^kT(n-k) + k(3^n)$ **Grading: 2 pts**

Since we know T(1), we aim to plug in a value of k such that $n - k = 1$, which means that we should set $k = n - 1$ and plug into the formula above to yield

$T(n) = 3^{n-1}T(1) + (n-1)3^n$ **Grading : 1 pt**

Plug in T(1) = 2: **Grading : 1 pt**

$T(n) = 3^{n-1}(2) + (n-1)3^n$

Factor out 3^{n-1} to get:

$T(n) = 3^{n-1}(2 + (n-1)3)$

$T(n) = 3^{n-1}(3n - 3 + 2)$

$T(n) = 3^{n-1}(3n - 1)$ **Grading : 1 pt**

Thus, to fit the suggested form, $a = n - 1$, $b = 3$ and $c = 1$.

4) (5 pts) Show the result of each of the following bitwise operations:

(a) 43 & 25 = **9**

(b) 43 | 25 = **59**

(c) 43 ^ 25 = **50**

(d) 43 >> 2 = **10**

(e) 25 << 4 = **400**

Grading: 1 pt for each all or nothing.

5) (5 pts) The level of a black belt karate student is based on the skills they have achieved. Skills are numbered 0 through 19 and only count if achieved in order. For example, a student who has achieved skills 0, 3 and 7 is considered a level 1 black belt. (Your skill level is the lowest skill number you haven't achieved yet and someone who has achieved all of the skills is a level 20 blackbelt.) The local dojo stores information about the skills gained by each student in a single integer bitmask. (For example, the student who has achieved skills 0, 3 and 7 would have the integer $2^0+2^3+2^7 = 137$ stored in their entry.) Write a function that takes in the integer bitmask of a student and returns her level.

```
int getLevel(int skillMask) {

    int i = 0;                                  // 1 pt
    while ( (skillMask & (1<<i)) != 0)          // 2 pts
        i++;                                    // 1 pt

    return i;                                   // 1 pt
}
```

6) (6 pts) Write a recursive function that takes in a string (pointer to a character array) and a character, and returns the number of times the character appears in the string. (The key to this function is recalling that if str points to a character in a string, then str+1 points to the next character in the string and effectively represents the rest of the string starting at the second character.)

```
int numOccurrences(char* str, char target) {

    if (str[0] == '\0') return 0;               // 2 pts

    return (str[0] == target) + numOccurrences(str+1, target);

    // 1 pt return, 2 pts checking index 0 with target,
    // 1 pt adding rec call
}
```

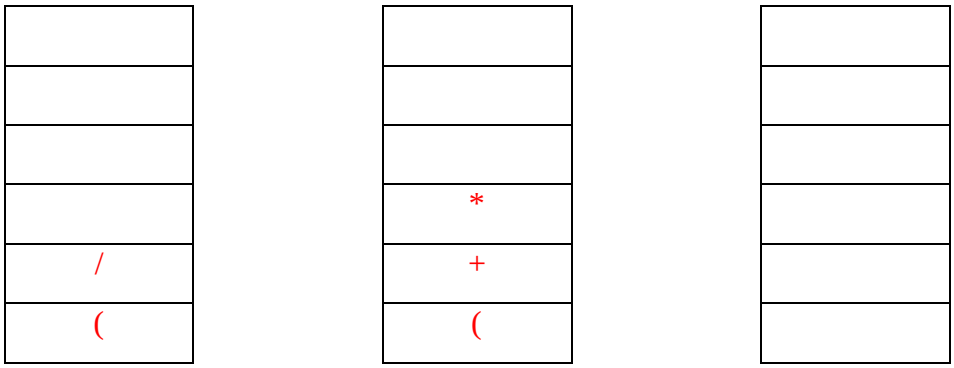
7) (9 pts) **Using the binary search technique**, write a function that takes in a long long, number, and returns the largest integer x such that $x^2 \leq$ number. You may assume that $0 \leq$ number $\leq 10^{18}$.

```
long long intSquareRoot(long long number) {

    long long low = 0, high = 100000000011;        // 2 pts

    while (low < high) {                           // 1 pt

        long long mid = (low+high+1)/2;            // 2 pts

        if (mid*mid > number)                      // 1 pt
            high = mid-1;                          // 1 pt
        else
            low = mid;                             // 1 pt
    }

    return low;                                    // 1 pt

}
```

8) (5 pts) Convert the following infix expression to postfix, showing the state of the operator stack at each of the indicated points and showing the output of the algorithm (equivalent postfix expression).

A B C

((3 + 7) / (8 - 6) + 4 * 3 + 4) / 7 - 1



A B C

Resulting postfix expression:

3	7	+	8	6	-	/	4	3	*	+	4	+	7	/	1	-				

Grading: 1 pt for each stack, 2 pts for expression (give 2 if correct, 1 if ½ way, 0 otherwise)

9) (5 pts) Complete the function below so that it adds 1 to the first item in the linked list pointed to by front, 2 to the second item, 3 to the third item and so forth. Write the function **iteratively.** The node struct and function prototype are provided below.

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

void addIndex(node* front) {

    int i = 1;                      // 1 pt
    while (front != NULL) {         // 1 pt
        front->data += i;           // 1 pt
        i++;                        // 1 pt
        front = front->next;        // 1 pt
    }

}
```

10) (15 pts) Consider the following problem:

Given the **differences** between successive elements in a permutation of 0, 1, 2, …, n-1, reconstruct the first lexicographical permutation that has the given differences.

For example, if you were given n=5 and the difference sequence 4, 3, 2, 1, then the correct answer would be the permutation [0, 4, 1, 3, 2], since $|0 - 4| = 4$, $|4 - 1| = 3$, $|1 - 3| = 2$ and $|3 - 2| = 1$. (Note that [4, 0, 3, 1, 2] also satisfies the difference sequence, but it comes after [0, 4, 1, 3, 2], lexicographically.)

The brute force solution would involve generating all n! permutations of 0, 1, 2, …, n-1 in lexicographical order and checking to see which is the first one that satisfies the difference sequence.

A better brute force solution would be to try all n values for the first item, but after that, using the difference information to just try two possible numbers for all future slots (adding the appropriate difference to the previous number and subtracting the appropriate difference from the previous number). For example, if we try the first term as 0, then the two possible second terms are $0 + 4 = 4$ and $0 - 4 = -4$ for the second term.

As the example above indicates, some of the sequences generated by trying both terms in some slots aren't valid permutations, either because they produce a number out of bounds, or they produce a number that has previously been used. Thus, the **backtracking** solution involves skipping over trying numbers that are invalid for either of the two reasons mentioned above.
Complete the solution that starts on this page and finishes on the next page to this problem. A lot of the code is support code. You only have to fill in some code for the recursive function called **go**, which is all on the next page. Some comments are provided to help you out. If you fill your

part in correctly, the code will print out the first lexicographical solution to the test case read in, if it exists, or the word “Impossible” if no solution is consistent with the difference data.

```
#include <stdio.h>
#include <stdlib.h>

int* wrapper(int* diff, int n);
int* go(int* diff, int* perm, int k, int* used, int n);

int main(void) {

    int n;
    scanf("%d", &n);
    int* diff = calloc(n-1, sizeof(int));
    for (int i=0; i<n-1; i++) scanf("%d", &diff[i]);

    int* res = wrapper(diff, n);

    if (res != NULL) {
        for (int i=0; i<n; i++)
            printf("%d ", res[i]);
        printf("\n");
        free(res);
    }
    else
        printf("Impossible\n");

    free(diff);
    return 0;
}

int* wrapper(int* diff, int n) {

    int* perm = calloc(n, sizeof(int));
    int* used = calloc(n, sizeof(int));

    for (int i=0; i<n; i++) {
        perm[0] = i;
        used[i] = 1;
        int* tmp = go(diff, perm, 1, used, n);
        if (tmp != NULL) {
            free(used);
            return tmp;
        }
        used[i] = 0;
    }

    free(perm);
    free(used);
    return NULL;
}

// k represents the current index into the perm array, k > 0.
int* go(int* diff, int* perm, int k, int* used, int n) {

    if (k == n) return perm;                    // 1 pt
```

```
    int step[2];

    // Stores subtracting from the previous term.

    step[0] = perm[k-1] - diff[k-1];            // 3 pts

    // Stores adding to the previous term.

    step[1] = perm[k-1] - diff[k-1];            // 3 pts

    // Go through both possible steps.
    for (int i=0; i<2; i++) {

        // We only try this step if it might lead to a possible solution.

        if ( step[i] >= 0 && step[i] < n && !used[step[i]] ) {  // 3 pts

            perm[k] = step[i];                  // 1 pt

            used[ step[i] ] = 1;                // 1 pt


            int* tmp = go(diff, perm, k+1, used, n);        // 2 pts

            if (tmp != NULL) return tmp;

            used[ step[i] ] = 0;                // 1 pt
        }
    }
    return NULL;
}
```

11) (5 pts) List the **worst-case** run times of each of the following operations, in terms of the appropriate variables in the prompt.

(a) Inserting an item into a linked list with ***n*** nodes. **O(n)**

(b) Deleting an item from an AVL tree with ***n*** nodes. **O(lg n)**

(c) Inserting ***k*** items in succession into a binary search tree which starts with ***n*** items already. **O(nk)**

(d) Removing the minimum element from a priority queue storing ***n*** items. **O(lg n)**

(e) Sorting ***n*** numbers using Quick Sort. **$O(n^2)$**

Grading: 1 pt each, all or nothing.

12) (12 pts) Define the string f(0) = 'a', and for each subsequent integer, x, we define the string f(x) based on the string f(x-1). In particular, we form f(x) by taking the $(x+1)^{th}$ letter and concatenating to the end of it two copies of f(x-1). For example, we have:

f(1) = "baa"
f(2) = "cbaabaa"
f(3) = "dcbaabaacbaabaa"

Write a function that takes in n, dynamically allocates space to store f(n), fills in the dynamically allocated array and returns a pointer to it. **(Note: This question is simultaneously testing dynamic memory management and recursion.)** In your function, you'll have to use recursion, allocate new memory, and free other memory once it's not needed. Also, you will find one of the string functions particularly helpful. Finally, the amount of space required to store f(n) is exactly 2^{n+1} characters because the string itself is $2^{n+1} - 1$ characters long and we need space for the null character. Some of the code has been provided below. Fill in the rest.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* f(int n) {

    // Allocate enough space to store the whole result.

    char* res = malloc(sizeof(char)*(1<<(n+1)));  // 2 pts

    // Just stores the first character of the result in res.
    res[0] = (char)('a'+n);
    res[1] = '\0';

    // Base case

    if ( n == 0 ) return res ;          // 2 pts

    char* tmp = f(n-1) ;                // 2 pts

    strcat(res, tmp) ;                  // 2 pts

    strcat(res, tmp) ;                  // 2 pts
                        // Note: last two lines could also
                        // be a for loop which runs twice…
    free( tmp );                        // 1 pt

    return res ;                        // 1 pt
}
```

13) (6 pts) An algorithm that runs in O(nlgn) time processes an input array of size n = 2^{20} in 190 milliseconds. How long is the algorithm expected to take when processing an input of size n = 2^{25}? **Please express your answer in seconds.**

Let T(n) represent the run time of the algorithm for an input size of n. Then for some constant c, we have T(n) = cnlgn. Plug in the given info:

$$T(2^{20}) = c2^{20}\lg(2^{20}) = 190\ ms$$

$$20c(2^{20})lg2 = 190ms$$

$$c = \frac{190ms}{20(2^{20})lg2}$$

Now, let's solve for T(2^{25}):

$$T(2^{25}) = \frac{190ms}{20(2^{20})lg2} \times 2^{25}\lg(2^{25}) = (190ms)(2^5) \times \frac{25\lg 2}{20lg2} = (190ms)(32)(\frac{5}{4})$$

$$= (190ms)(8)(5) = (190ms)(40) = 7600ms = \boldsymbol{7.6seconds}$$

Grading: 1 pt to write equation to solve for c. 2 pts to get a reasonable expression (not necessarily simplified) for c. 2 pts to plug in 2^{25} and simplify to ms. 1 pt to convert to seconds.

14) (4 pts) What meat is used to make the kid favorite, chicken nuggets?

chicken! (Give to all)