

## Spring 2019 COP 3502H Final Exam Solutions

### Questions 1-10: Writing Functions for Word Game

For the first ten questions of this exam, you will write 10 functions to complete an implementation of Word Game. Here is a high level description of the game:

1. You read in a valid dictionary of words for the game. The dictionary has  $n$  words which are each strings of lowercase letters (no more than length 99). Each word is stored in a struct and the struct stores the following information: the word itself, the length of the word and its score in Word Game.
2. You will sort the list of these valid words as follows: words with higher scores come first. If two words have the same score, then the tie will be broken alphabetically, with words that come first alphabetically coming before words of the same score that come later alphabetically.
3. You will generate a list of  $k$  random letters which can be used to form words.
4. You will print out a sorted list (as previously described) of all of the words from the valid dictionary that you can form using your  $k$  random letters. This will be a numbered list and you will print out each word, along with its corresponding score.

Details for each function you will write will be discussed in the prompt for each individual section. Included for you will be program upto the end of main (#includes, #defines, struct definition, global variables function prototypes). This will be on the second page of the exam. The questions will start on the third page.

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
typedef struct word {
    char* str;
    int len;
    int score;
} word;

#define MAX 100
#define ALPHASIZE 26

int numWords;

void printResults(char* rack, word** list);
word** getWordList(FILE* ifp);
void sort(word** words, int length);
int cmp(const word* w1, const word* w2);
int contains(const char* letters, const word* item);
void setScore(word* item);
int* getFreq(const char* letters);
void print(const word* item);
char* rndLetters(int n);
void freeWordList(word** list, int len);

int main(void) {
    char filename[MAX];
    printf("Enter the file with your dictionary of words.\n");
    scanf("%s", filename);
    FILE* ifp = fopen(filename, "r");
    word** mywords = getWordList(ifp);
    fclose(ifp);

    sort(mywords, numWords);

    int i, numLet;
    printf("How many letters do you want for the game?\n");
    scanf("%d", &numLet);

    char* rack = rndLetters(numLet);
    printf("The letters you got this round are %s.\n", rack);
    printResults(rack, mywords);
    free(rack);
    freeWordList(mywords, numWords);
    return 0;
}

```

1) (10 pts) The first function to write is the function that takes in a pointer to the file storing the words (this file has a single integer on the first line indicating the number of words in the file, followed by the words, one per line on the subsequent lines.) This function first reads in the number of words in the file into the global variable numWords. Then the function allocates space for an array of numWords pointers to words. Then, the function will read the words from the file, one by one. When reading them in initially, the words should be read into a fixed array of size MAX. Then, space should be allocated for a word struct to store this word. First, the length of the word should be set inside of the struct. Then, the pointer str should be dynamically allocated to point to an array of the proper size to just store the word and the associated null character. Then, the word should be copied into the struct. Finally, the score should be set for this word using the setScore function. (Even though you haven't written this function yet, calling it should be straight-forward, based on the function prototype provided.) Finally, you return a pointer to the array you have just created.

```
word** getWordList(FILE* ifp) {

    int i;
    fscanf(ifp, "%d", &numWords); // 1 pt

    word** list = malloc(numWords*sizeof(word*)); // 1 pt
    for (i=0; i<numWords; i++) { // 1 pt
        list[i] = malloc(sizeof(word)); // 1 pt
        char tmp[MAX];
        fscanf(ifp, "%s", tmp); // 1 pt
        list[i]->len = strlen(tmp); // 1 pt
        // 1 pt
        list[i]->str = malloc((list[i]->len+1)*sizeof(char));
        strcpy(list[i]->str, tmp); // 1 pt
        setScore(list[i]); // 1 pt
    }

    return list; // 1 pt
}
```

2) (3 pts) Write a function that takes in a pointer to an array of pointers to word as well as the length of the array and frees all of the associated memory.

```
void freeWordList(word** list, int len) {

    int i;
    for (i=0; i<len; i++) { // 0 pts
        free(list[i]->str); // 1 pt
        free(list[i]); // 1 pt
    }
    free(list); // 1 pt
}
```

3) (4 pts) Write a function that sets the score of a word struct by taking in a pointer to that struct. The scoring system is a bit strange. Define the value of an individual letter to be its alphabetic 0-based rank. Thus, 'a' has value 0, 'b' has value 1, ..., 'z' has value 25.

For a word with **lowercase** letters  $w_0w_1w_2\dots w_{k-1}$  (subscripts are the array indexes where the letters are), the score of the word is  $\sum_{i=0}^{k-1} |i - \text{val}(w_i)|$ . For example, consider the word "cat." The letter values are 2, 0 and 19, respectively. The score for this word is  $|0-2| + |1-0| + |2-19| = 20$ .

Your function takes in a pointer to a word and should set its score to 0, then it should go through the word itself and update the score accordingly.

```
void setScore(word* item) {  
  
    int i;  
    item->score = 0; // 1 pt  
    for (i=0; i<item->len; i++) // 1 pt  
        item->score += (abs(i-(item->str[i]-'a'))); // 2 pts  
  
}
```

4) (3 pts) Write a function that compares two word structs and returns a negative integer if the first "comes before" the second, 0 if they are equal, and a positive integer if the first "comes after" the second. We assume that the structs are completely filled (so the score component in them is accurate). We compare as follows: if one struct's score is higher, it comes BEFORE the other struct. If two struct's have the same score, then the one that comes first alphabetically comes BEFORE the other struct.

```
int cmp(const word* w1, const word* w2) {  
  
    if (w1->score != w2->score) // 1 pt  
        return w2->score - w1->score; // 1 pt  
    return strcmp(w1->str, w2->str); // 1 pt  
  
}
```

5) (7 pts) Pick your favorite  $O(n^2)$  sort from this list: Selection Sort, Insertion Sort, Bubble Sort, and implement it to sort an array of pointers to word structs. The sort should use the `cmp` function from question #4 only to determine ordering.

```
void sort(word** words, int length) {  
  
    int i, j;  
    for (i=length-1; i>=0; i--) { // 1 pt  
        for (j=0; j<i; j++) { // 1 pt  
            if (cmp(words[j], words[j+1]) > 0) { // 2 pts  
                word* tmp = words[j]; // 1 pt  
                words[j] = words[j+1]; // 1 pt  
                words[j+1] = tmp; // 1 pt  
            }  
        }  
    }  
}
```

6) (5 pts) Write a function that takes in an integer,  $n$ , and returns a string of  $n$  random letters. Note that your function must dynamically allocate space for  $n+1$  characters, fill the first  $n$  of them with randomly chosen lowercase letters. (Each letter should roughly have a chance of  $1/26$  of being picked each time.) Fill the last slot with the null character and return a pointer to the array created.

```
char* rndLetters(int n) {  
  
    char* res = malloc((n+1)*sizeof(char)); // 1 pt  
    int i;  
    for (i=0; i<n; i++) // 1 pt  
        res[i] = (char)(rand()%26+'a'); // 1 pt  
    res[n] = '\\0'; // 1 pt  
    return res; // 1 pt  
}
```



9) (1 pt) Write a one line function that takes in a pointer to a word struct and prints out the string in the struct, followed by a colon and a space, followed by the score of the word struct, followed by a new line character. Thus, for a word struct storing "cat", this should be printed:

```
cat: 20
```

```
void print(const word* item) {  
  
    printf("%s: %d\n", item->str, item->score);    // 1 pt  
  
}
```

10) (6 pts) Write the printResults function. This function should print out a numbered list of each word that can be printed using the letters in rack, of the words pointed to by list. Since the list is already ordered as previously described, this list will come out sorted the same way, listing words for which the player would get more points first. Let's say you could only print out the words "cat", "bat", "at" and "cab" with your letters (out of many more possible words), the corresponding output your function should produce is:

```
1. cat: 20  
2. bat: 19  
3. at: 18  
4. cab: 4
```

```
void printResults(char* rack, word** list) {  
  
    int listNo = 1, i;    // 1 pt  
  
    for (i=0; i<numWords; i++) {    // 1 pt  
  
        if (contains(rack, list[i])) {    // 1 pt  
            printf("%d. ", listNo);    // 1 pt  
            print(list[i]);    // 1 pt  
            listNo++;    // 1 pt  
        }  
    }  
  
}
```

11) (8 pts) I bet you thought you were done with the Word Game. Sorry to disappoint you, but you're not!!! Please do a detailed worst-case run-time analysis of the whole program, given that the input dictionary has  $n$  words, each of the words in the input dictionary are at most length  $k$ , the number of letters you get is at most length  $m$  and the alphabet size is  $a$ . Your analysis and final answer must include each of these four variables, along with a written explanation as to how long each piece of the code runs. So, your final answer ought to be the sum of a few terms, where each term represents the run-time of one portion of the code.

### Solution

We first run the function `getWordList`. This will run in  $O(nk)$  time, since it takes us  $O(k)$  time to read in each word, and then again, we go through each word, letter by letter, a second time to set the score.

Next we get to the sort. The for loop structure takes  $O(n^2)$  time and the compare function take  $O(k)$  time in the worst case since it may call the `strlen` function, which has a worst case run time of the shorter of the length of the two strings it is passed. Thus, the sort takes  $O(n^2k)$  time.

Generating your rack of tiles takes  $O(m)$  time, assuming `rand()` runs in constant time (which was my intention for this problem).

Lastly, we run the `printResults` function. The loop for this function runs  $n$  times. The main work in each loop iteration is the `contains` function (this takes longer than the `print` theoretically). The first thing the `contains` function does is make two calls to the `getFreq` function with strings of length  $k$  and  $m$ , in the worst case. The run time of `getFreq` is simply  $\max(a, m)$  for the first call and then  $\max(a, k)$ , for the second call. A simple way to express  $\max(a, m) + \max(a, k)$  in order notation is  $O(a + m + k)$ , or  $O(\max(a, k, m))$ . Both are equivalent and both (and other equivalent variants will be accepted for full credit). This means that this runs in time  $O(n(a + m + k))$  time.

Our total run time is:

$$O(nk + n^2k + m + n(a + m + k))$$

In this sum of four terms, the second term "supersedes" the first and the fourth supersedes the third, so if we wanted, we could simplify our expression to be:

$$O(n^2k + n(a + m + k))$$

For reasonable values of  $a$  and  $m$ , this would further simplify to  $O(n^2k)$ , but this simplification can't be made in the general case.

**Grading: 2 pts for get wordlist analysis, 3 pts for the sort, 0 pts for generating the rack, 2 pts for the last part and 1 pt for putting it all together. (Take off 2 pts if the  $k$  is missing from the sort analysis and take off 1 pt if  $m$  is missing from the printing analysis.)**

12) (8 pts) Solve the following recurrence relation defined for non-negative integers,  $n$ , using the iteration technique. Please solve the recurrence **exactly**, obtaining a closed-form solution for  $T(n)$ , in terms of  $n$ .

$$T(n) = 2T(n - 1) + 2^n, \text{ for } n > 0$$

$$T(0) = 1$$

**Solution**

$$T(n) = 2T(n - 1) + 2^n$$

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

$$T(n) = 4T(n - 2) + 2^n + 2^n$$

$$T(n) = 4T(n - 2) + 2(2^n)$$

$$T(n) = 4(2T(n - 3) + 2^{n-2}) + 2(2^n)$$

$$T(n) = 8T(n - 3) + 2^n + 2(2^n)$$

$$T(n) = 8T(n - 3) + 3(2^n)$$

Now that we've done three iterations, it's reasonably clear that after  $k$  iterations, we would get the following form:

$$T(n) = 2^k T(n - k) + k(2^n)$$

Since we are given  $T(0)$ , we should plug in  $n - k = 0$ , so  $k = n$  into this formula above to yield:

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

$$T(n) = 2^n T(0) + n(2^n)$$

$$T(n) = 2^n(1) + n(2^n)$$

$$\underline{T(n) = 2^n(n + 1)}$$

**Grading: 1 pt for each iteration, 2 pts for correct guess, 1 pt to plug in  $k = n$ , 2 pts to arrive at the correct final answer.**

13) (6 pts) Consider a system where applicants for a job fill out a T/F questionnaire of no more than 20 questions, indicating if they have certain skills or not. We can store the answer of a single questionnaire in an int variable, where bit  $i$  is set to 1 if the applicant has skill  $i$  and is set to 0 otherwise. For example, the number 25 indicates that the applicant has skills 0, 3 and 4, since  $25 = 2^4 + 2^3 + 2^0$ . For this problem, consider a situation where an array of integers of length  $len$  stores the questionnaire answers for  $len$  number of applicants. A company has some set of required skills, which can also be stored in a single integer. For example, if a company needs to hire an employee and who they hire must have skills 2, 4 and 5, then the requirements for that job can be stored as the integer  $52 = 2^5 + 2^4 + 2^2$ . Write a function that takes in an array of integers storing the answers to questionnaires, an integer,  $len$ , indicating the length of the input array, and an integer,  $req$ , representing the minimum requirements for a position. Have the function return the number of candidates who satisfy the minimum requirements for that position.

```
int getNumQualified(int* apps, int len, int req) {

    int i, res = 0;           // 0 pt
    for (i=0; i<len; i++)    // 1 pt
        if ((req & apps[i]) == req) // 3 pts
            res++;           // 1 pt
    return res;              // 1 pt

}
```

14) (5 pts) A sorting algorithm with a runtime of  $O(n \lg_2 n)$  takes .1 seconds to sort  $2^{20}$  integers. How long would the algorithm be expected to take if it were sorting  $2^{25}$  integers. Write your answer in seconds and put a box around your final answer.

### Solution

Let  $T(n)$  equal the run time of the algorithm with an input size of  $n$ . Then, for some constant  $c$ , we have:  $T(n) = cn \log_2 n$ . Using the data point given, solve for  $c$ :

$$T(2^{20}) = c(2^{20}) \log_2(2^{20}) = .1 \text{ sec}$$

$$c = \frac{.1 \text{ sec}}{20(2^{20})}$$

Now, solve for  $T(2^{25})$ :

$$T(2^{25}) = c(2^{25}) \log_2(2^{25})$$

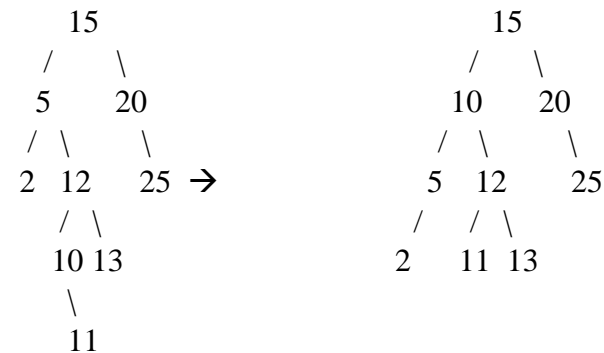
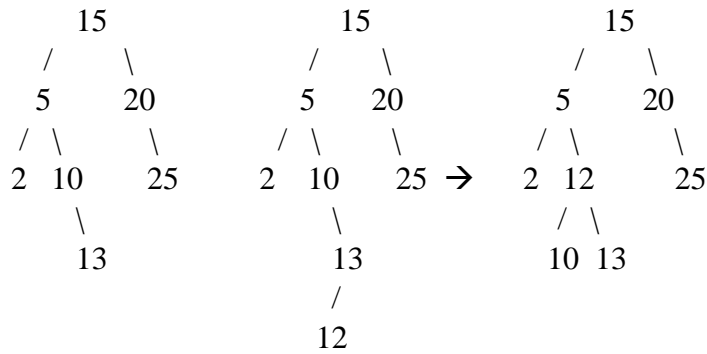
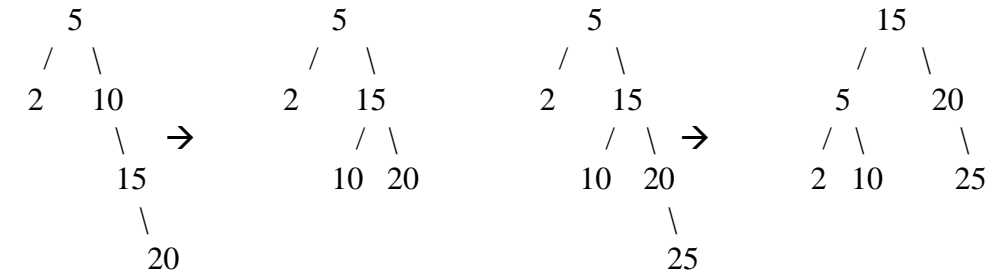
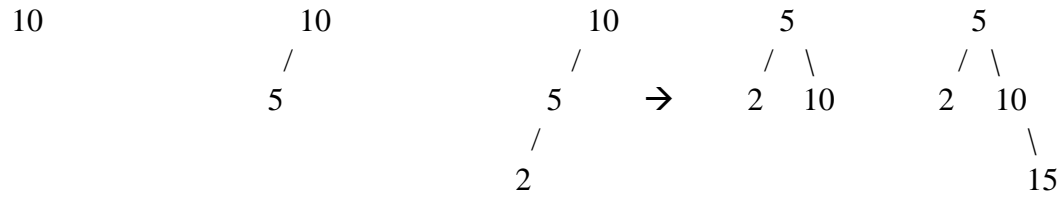
$$T(2^{25}) = \left(\frac{.1 \text{ sec}}{20(2^{20})}\right)(2^{25})(25)$$

$$T(2^{25}) = \left(\frac{.1 \text{ sec}}{4}\right)(2^5)(5) = (.1 \text{ sec})(8)(5) = \mathbf{4 \text{ sec}}$$

**Grading: 2 pts solve for  $c$ , 1 pt plug in  $2^{25}$ , 2 pts solve for final answer in sec.**

15) (8 pts) Insert the following integers into an AVL tree in the order shown and put a box around the result after EACH insertion: 10, 5, 2, 15, 20, 25, 13, 12 and 11.

**Solution**



**Grading: 1 pt per tree not counting the very first one.**

16) (8 pts) Consider inserting several integers into a hash table of size 13 using the following hash function and the quadratic probing method:

```
int hashfunc(int x) {
    int res = 0;
    while (x > 0) {
        res = (res + x%10)%13;
        x /= 10;
    }
    return res;
}
```

Show where each of the integers would end up in the table if these were the values inserted (in this order): 63, 1872, 12345, 11, 7223, 88263, 2344, and 77876

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	<b>2344</b>	<b>7223</b>	<b>12345</b>	<b>11</b>		<b>1872</b>				<b>63</b>	<b>88263</b>		<b>77876</b>

**Grading: 1 pt per value**

17) (4 pts) Write an **iterative** function that takes in a pointer to the first node in a linked list and returns 1 if the linked list is sorted from lowest to highest (ties allowed) and returns 0 otherwise.

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

int isSorted(node* head) {

    if (head == NULL) return 1;           // 1 pt
    while (head->next != NULL) {         // 1/2 pt
        if (head->data > head->next->data) // 1 pt
            return 0;                   // 1/2 pt
        head = head->next;               // 1/2 pt
    }
    return 1;                             // 1/2 pt
                                           // Round down
}
```

18) (1 pt) Panda Express serves Chinese food. From what country does the Panda hail? **China**  
(1 pt give to all)