Final Practice - CS136

Prepared and compiled by Tommy Pang, special thank to Benny Wu for review

Last update: April 13th, 2024

Disclaimer:
The practice problems is an independent effort and is **not affiliated** with the University of Waterloo. The materials provided are intended solely for practice purposes, as we recognize its potential benefits for first-year students. Should there be any concerns regarding rights or violations, please contact us immediately for removal.

Conceptual Questions

1. Give three advantages of using modularization and describe each.

2. Which of the following statement(s) are true?
   a. Clients may require functions from modules.
   b. Clients provide implementation to modules.
   c. Modules provide functions to clients.

3. Which of the following statement(s) are true?
   a. .o files represent source files that are compiled into machine code.
   b. Only one module is permitted in a program.
   c. We can combine multiple machine code files to build a program.
   d. A program must have exactly one function called main.

4. Discuss declaration vs. definition in C.

5. Describe the `extern` keyword.

6. True or False: A module `my_module.c` will not compile if you do not include `my_module.h`.

7. Consider the below module, label each identifier (`x`, `score`, `score_update`, `main`, `run_game`, `MAX_SCORE`) with their scopes (Local, Program, Module scope):

```c
// main.c
extern int score;
const int MAX_SCORE = 100;

void run_game(void);

int main(void) {
    run_game();
    printf("Score: %d", score);
}
```

```c
// module.c
int score = 0;
static int direction = 0;

static void score_update(int n)
    {
    score += n;
}

void run_game(void) {
    int x = 0;
    // ...
}
```

8. Describe an Opaque structure.

9. What constant(s) does the module `limits.h` provide?

10. What constant(s) does the module `stdlib.h` provide?

11. Compare Interface vs. implementation.

12. Why is the actual data structure & implementation hidden from the client in an ADT?

13. Is this a valid array definition: `char a[5] = "array";`?
14. Is this a valid array definition: `char a[5] = {0};`?
15. Is this a valid array definition: `int a[3] = {0, 1, 2, 3};`?
16. Given definition `char a[5] = {0};` what does `strlen(a)` return?

17. True or False: Given algorithm A has worst case complexity $O(n^2)$, and algorithm B has worst case complexity $O(n \log n)$, then A runs faster than B in all instances (assume of size n).
18. State the correct order for the running time: $100000 + 0.0001n + 0.001 \log n$.

19. Insertion Sort, Selection Sort, and Quicksort, which is more efficient?
    a. Insertion Sort
    b. Selection Sort
    c. Quicksort
    d. They all have the same efficiency
20. True/False: C has a String type that's built-in.
21. True/False: The length returned by `strlen` includes the null-terminator.
22. True/False: The character '0' is the null terminator.
23. State the result of the below `strcmp` calls:
    a. `strcmp("", "x");`
    b. `strcmp("2", "1");`
    c. `strcmp("abcd", "abc");`
24. What's the result of the length:

```
char arr[5] = {'a', 'x', '0', 'x'};
printf("%d\n", strlen(arr));
```

25. What's the output of the following code:

```
char s1[] = "str";
char s2[] = "str";

if (s1 == s2) {
    printf("equal");
}
else {
    printf("not equal");
}
```

26. Describe string literals.

27. Consider this code, write out the output produced, or up to the point of error occurrence if you think there's an error, and describe what's the error (e.g. heap-overflow):

```
int i = 0;
const char* s = "abc\0aaa\0bbb";
while (i < strlen(s)) {
    printf("%c", s[i]);
    i++;
}
for (int j = 1; j <= 3; ++j) {
    printf("%c", s[i + j]);
}
```

28. Why this is an issue:

```
void dumb_string_op(const char* a, const char* b) {
    strcpy(a, b);
}
```

29. In the following code snippet, which line will error occur at runtime in Edx?

```
int main(void) {
    int *j = malloc(sizeof(int)); // line 1
    free(j); // line 2
    *j = 43; // line 3
    return 0; // line 4
}
```

Options:

a. Line 1
b. Line 2
c. Line 3
d. Line 4

30. In the following code snippet, which line will error occur at runtime in Edx?

```
int main(void) {
    int *j = malloc(sizeof(int)); // line 1
    int *k = j; // line 2
    free(k); // line 3
    *j = 43; // line 4
    return 0; // line 5
}
```

Options:

a. Line 1
b. Line 2
c. Line 3
d. Line 4
e. Line 5

31. Why is it a good practice to set a pointer to NULL after freeing it?

32. What occurs when the malloc function is unable to allocate the requested amount of memory? Options:

    a. Program end with non 0 exit code
    b. malloc allocate the maximum # bytes that's affordable
    c. malloc returns NULL
    d. Undefined behavior

33. What is a dangling pointer, and provide an example.

34. Compare stack and heap data/memory.

35. List two advantages of using heap memory.

36. Is there anything wrong with the below function that destroys the linked list?

```
struct Node {
    const void* val;
    struct Node* next;
};

struct List {
    struct Node* front;
};

void destroy_linked_list(struct List* lst) {
    struct Node* cur = lst->front;
    while (cur) {
        free(cur);
        cur = cur->next;
    }
}
```

37. Will the below code result in a dangling pointer?

```
void bruhdanglingptr(int n) {
    char *a = malloc(n * sizeof(char));
    char *b = malloc(n * sizeof(char));
    b = a;
}
```

38. Will the below code result in a dangling pointer?

```
void bruhdanglingptr(int n) {
    char *a = malloc(n * sizeof(char));
    a = malloc(2 * n * sizeof(char));
}
```

39. Will the below code result in a dangling pointer?

```
void bruhdanglingptr(int n) {
    char *a = malloc(n * sizeof(char));
    char *c = realloc(a, 2 * n * sizeof(char));
}
```

40. Will the below code for sure result in a memory leak?

```
void bruhmemleak(int n) {
    char *a = malloc(n * sizeof(char));
    char *b = malloc(n * sizeof(char));
    b = a;
}
```

41. Will the below code for sure result in a memory leak?

```
void bruhmemleak(int n) {
    char *a = malloc(n * sizeof(char));
    char *c = realloc(a, 2 * n * sizeof(char));
}
```

42. Is it true that the worst case complexity of pushing to a stack ADT seen in class is $O(n)$?

43. Is it true that if a program runs $O(1)$ amortized, then its worst case complexity cannot be worse than $O(n)$?

44. Is it true that amortized analysis is only applicable to data structures and cannot be used for analyzing algorithms?

45. Is it true that the amortized cost of an operation is always equal to the worst case cost of that operation?

46. Is it true that the amortized runtime/cost of an operation is always better than the worst case cost/runtime of that operation?

47. Beside each print statement(assume they run independently), write the corresponding output (address), if that line will cause an error, describe why. See the code here: `https://pastebin.com/zgvVfmT3`.

Complexity Analysis

1. What's the runtime worst case running time in terms of parameter n for the following function:

```
bool is_prime(int n) {
    if (n <= 1) {
        return false;
    } else if (n == 2) {
        return true;
    }
    for (int i = 3; i * i <= n; i += 2) {
        if (n % i == 0) {
        return false;
        }
    }
    return true;
}
```

Options:

a. $O(n)$
b. $O(\log n)$
c. $O(\sqrt{n})$
d. $O(n^2)$

2. What's the worst case time complexity/efficiency of this code:

```
void subset_sums(int i, int n, int val, int a[]) {
    if (i == n) {
        printf("%d\n", val);
        return;
    }

    subset_sums(i + 1, n, val, a);
    if (i % 2 == 1) {
        subset_sums(i + 1, n, val + a[i], a);
    }
}
```

Options:

a. $O(n^2)$
b. $O(n)$
c. $O(n!)$
d. $O(2^n)$

3. What's the worst case time complexity/efficiency of this code:

```
int cringe_search() {
    int lo = 0, hi = 1000000, ans = -1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (mid % 2 == 1) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return ans;
}
```

Options:

a. $O(\log n)$
b. $O(n)$
c. $O(1)$
d. $O(n^2)$

4. What's the worst case complexity of the following pseudocode:

```
p = 1
s = 0
for i = 1 to n do
    p = p * 2
    for j = 1 to p do
        s = s + 1
```

You may find geometric series summation formula useful:

$$s_n = ar^0 + ar^1 + \ldots + ar^{n-1} = \sum_{k=0}^{n-1} ar^k = \sum_{k=1}^{n} ar^{k-1} = \begin{cases} a\frac{1-r^n}{1-r} & \text{if } r \neq 1 \\ an & \text{otherwise} \end{cases}$$

Options:

a. $O(2^n)$
b. $O(n \log n)$
c. $O(n2^n)$
d. $O(n^2)$

5. What's the runtime worst case running time in terms of parameter n:

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }

    int a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        int temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}
```

Options:

- $O(n)$
- $O(1)$
- $O(\sqrt{n})$
- $O(n^2)$

6. What's the runtime worst case running time in terms of parameter n:

```
void f(int n) {
    int a = 0;
    for (int i = 0; i < n; i++) {
        for (int j = n; j > i; j /= 2) {
            a = a + 1;
        }
    }
}
```

Options:

- $O(n)$
- $O(n \log n)$
- $O(n\sqrt{n})$
- $O(n^2)$

7. Give the exact number of iterations performed, i.e., what's the value printed:

```
void f(int n) {
    int value = 0;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            value += 1;
        }
    }
    printf("%d\n", value);
}
```

Options:

- $n(n+1)$
- $\frac{n(n-1)}{2}$
- n^2
- n

8. What's the worst case complexity of the function `dumb_binary_search`:

```
int binary_search(int arr[], int lo, int hi, int target) {
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            lo = mid + 1;
        }
        else {
            hi = mid - 1;
        }
    }
    return -1;
}

int dumb_binary_search(int arr[], int n, int target) {
    int lo = 0, hi = n - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            lo = mid + 1;
        }
        else {
            int result = binary_search(arr, lo, mid - 1, target
                );
            if (result != -1) {
                return -1;
            }
            hi = mid - 1;
        }
    }
    return -1;
}
```

Options:

- $O(\log n)$
- $O((\log n)^2)$
- $O(n \log n)$
- $O(\log(n^2))$

9. What's the worst case complexity of the function `weird` (be exact):

```
void weird(int n, int m) {
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j < i % m; j++) {
            printf("*");
        }
    }
}
```

Options:

- $O(n^2)$
- $O(n)$
- $O(m)$
- $O(nm)$

10. What is the worst case complexity of the following function `weird2`?

```
void weird2(int n, int m) {
    for (int i = 2; i <= n; i++) {
        for (int j = 1; j < i % 1000000; j++) {
            printf("*");
        }
    }
}
```

Options:

- $O(n)$
- $O(n^2)$
- $O(n \log n)$
- $O(\log n)$

11. What's the worst case complexity of the function `weird_recurrence2`? **Hint: geometric series sum**

```
void weird_recurrence2(int n) {
    if (n == 0) return;
    weird_recurrence(n / 2);

    int s = 0;

    for (int i = 0; i < n; ++i) {
        for (int j = i + 1; j < n; ++j) {
            s++;
        }
    }

    printf("%d\n", s);
}
```

Options:

- $O(n^2 \log n)$
- $O(n^3)$
- $O(n^2)$
- $O(n \log n)$

12. What's the worst case complexity of the function `a_string_function`?

```
void a_string_function(char* s) {
    const char* dummy = "abc";
    char *res = "";

    for (int i = 0; i < strlen(s); ++i) {
        if (strcmp(dummy, s)) {
            strcat(res, s);
        }
    }
    printf("gg %s", res);
}
```

Options:

- $O(n^2)$
- $O(n^3)$
- $O(n^4)$
- $O(n^5)$

13. What's the worst case complexity of the function `a_string_function2`?

```
void a_string_function2(char* s, char* s2, char*res) {
    const char* tmp = "sheeeesh";

    for (int i = 0; i < strlen(s); ++i) {
        for (int j = 0; j < strlen(tmp); j *= 2) {
            strcopy(res, s2);
        }
    }
    printf("%s", res);
}
```

Options:

- $O(nm^2)$
- $O(nm \log n)$
- $O(n^2m^2)$
- $O(n^2m)$
- $O(n^2m \log n)$

14. What's the worst case complexity of the function `two_ptr`?

```
int two_ptr(int nums[], int k, int n) {
    int ans = n * (n + 1) / 2, max = 0;
    for (int i = 0; i < n; i++) {
        if (nums[i] > max) {
            max = nums[i];
        }
    }

    int cnt[100005] = {0};

    for (int i = 0, j = 0; j < n; ++j) {
        cnt[nums[j]]++;
        while (i <= j && cnt[max] >= k) {
            cnt[nums[i]]--;
            i++;
        }
        ans -= (j - i + 1);
    }
    return ans;
}
```

Options:

- $O(n^2)$
- $O(n)$
- $O(n \log n)$
- $O(n\sqrt{n})$

15. What's the worst case complexity of the function nxt_greater, assuming a stack ADT implemented with dynamic array is available?

```
void nxt_greater(int nums[], int n) {
    struct stack* stk = stack_create();
    for (int i = n - 1; i >= 0; --i) {
        while (!stack_empty(stk) && stack_top(stk) <= nums[i])
            {
            stack_pop(stk);
        }
        if (stack_empty(stk))
            printf("%d has no next greater element\n", nums[i])
                ;
        else
            printf("%d has next greater element - %d \n", nums[
                i], stack_top(stk));
        stack_push(stk, nums[i]);
    }
}
```

Options:

- $O(n)$
- $O(n^2)$
- $O(n \log n)$

Programming Questions

1. Implement a recursive function `pow(b, p)` that calculates b^p in $O(\log p)$ time and $O(1)$ space. You may assume recursion stack does not count towards space memory. Hint: notice $p = 2k$ or $2k + 1$ for some integer k, and $b^{2k} = (b^k)^2$, $b^{(2k+1)} = b^{2k} \cdot b$.

```
int pow(int b, int p) {
}
```

2. Given an array A of length n such that $1 \leq A[i] \leq 1000$ for all $i = 0, 1, \ldots, n-1$, sort the array in $O(n)$ time.

```
void special_sort(int A[], int n) {
}
```

3. Given a string containing brackets '(', ')', '[', ']', '', '', and other alphabetic characters, check if the brackets sequence is valid, i.e. all opening bracket has a corresponding closing bracket of the same type, and vice-versa. The brackets are closed by the same type that opened them. For example, "()ab[{}b]" is valid whereas "){}[[d]]" and "c{{)}}[]c" are not. Hint: use stack, you may assume you have a correct Stack ADT implementation (below) is given to you, Runtime: $O(n)$, Space: $O(n)$.

```
struct stack;
// malloc a stack in heap memory
struct stack *create();
// check if stack is empty
bool empty(const struct stack *stk);
// push element to top of stack
void push(void *item, struct stack *stk);
// get the top element of the stack
const void *top(const struct stack *stk);
// remove the top element of the stack
void *pop(struct stack *stk);
// free the resources used by stack
void destroy(struct stack *s);
```

```
    bool balanced(char* s, int n) {}
```

4. Given a string containing printable ASCII characters, write a function `most_frequent_char` that returns the most frequently occurred character in $O(n)$ time and constant (i.e. $O(1)$) space.

```
char most_frequent(const char* s, int n) {

}
```

5. A hashmap can be thought of as an array of buckets of length M, and in our case the buckets will be LinkedLists. We will implement a generic hashmap that will support storing any type. And any given element will be mapped to an integer index $0 \leq i < M$, by a hash function H. Essentially range(H) = {0, 1, ..., M - 1}. You will support below operations:

   a. table_create: it takes in the number of buckets M, a hash function h, a compare function $cmp(a, b)$ that returns 0 iff object a and b equals, and $print(x)$ that prints the value to std out.

   b. table_insert: given a void typed value x, compute its hash value i and try to insert it into bucket i if x is not already present. We will chain multiple elements together with LinkedLists if more than one element has been inserted at bucket i

   c. table_search: given a void typed value x, return true if the item x is in the hash table, false otherwise.

   d. table_remove: given a void typed value x, return true if the item x is removed from the hash table, false otherwise.

   e. table_destroy: given a hash table, free all its resources.

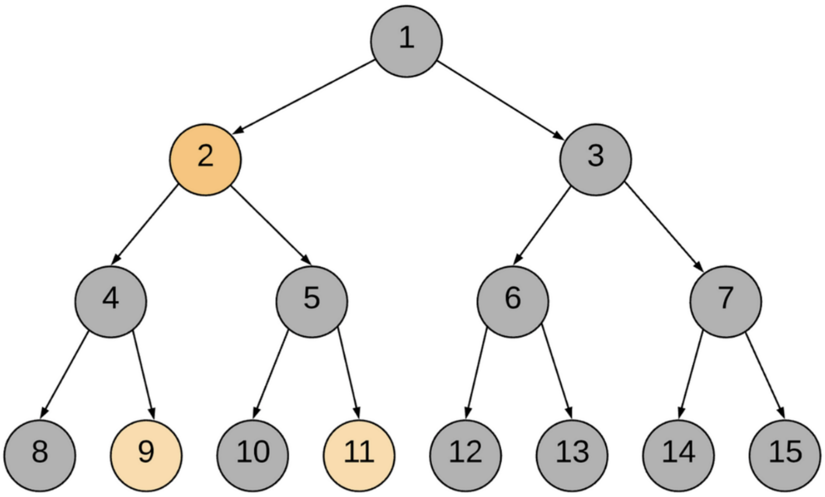
   f. table_print prints content of hash table, bucket by bucket, for each element in the bucket list, print its value.

   Starter code:

```c
struct Node {
    const void* val;
    struct Node* nxt;
};

struct List {
    struct Node* front;
};

struct hashtable {
    int size;
    int bucket_length;
    int (*hash_func)(const void *);
    int (*key_cmp)(const void *, const void *);
    void (*key_print)(const void *);
    struct List **buckets;
};

struct hashtable *table_create(int M, int (*hash_func)(const void
    *), int (*key_cmp)(const void *, const void *), void (*key_print
    )(const void *)) {}

bool table_insert(const void* x, struct hashtable* ht) {}

bool table_search(const void* x, struct hashtable* ht) {}

bool table_remove(const void* x, struct hashtable* ht) {}

void table_print(struct hashtable* ht) {}

void table_destroy(struct hashtable* ht) {}
```

6. Given two pointers $n1$ and $n2$ to distinct nodes in binary trees (not necessarily on the same tree), determine if there exists an LCA between $n1$ and $n2$. If so, return the pointer to LCA; otherwise, return NULL. The runtime should be $O(h)$, where h is the height of the binary tree. You may assume the root's parent is NULL.

```
struct Node {
    int val;
    struct Node* left;
    struct Node* right;
    struct Node* parent;
};

struct Node* find_lca(struct Node* n1, struct Node* n2) {}
```

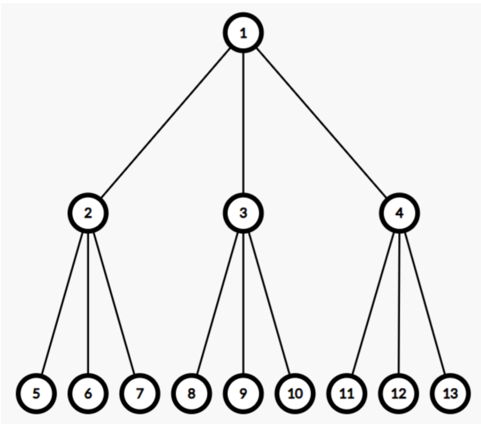


Lowest Common Ancestor for **Node 9** and **Node 11** is **Node 2**

7. Assume we have a variant of the binary tree, called a d-ary tree, where each node have exactly d children except the leaf nodes. Write a function `find_val` that returns the value of the kth ($1 \leq k \leq n$) element from the left in the jth level (root is level 1). The runtime should be $O(n)$ where n is the total number of elements. (**Challenge: Solve in O(h) where h is the height of the d-ary tree**)

```
/**
 * @param root The root of the d-ary tree.
 * @param j The depth of the element to find.
 * @param k The position of the element to find.
 */
int find_kth(struct DaryTreeNode* root, int k);
```

```
struct DaryTreeNode {
    int value;
    struct DaryTreeNode** children; // Array of pointers to
        children
    int numChildren;                // Actual number of
        children (d)
};


int find_kth(struct DaryTreeNode* root, int j, int k) {

}
```



For instance,

```
    find_kth(struct DaryTreeNode* root, 3, 7)
```

would yield 11 in the above example

8. Implement a substring function `substr(s, i, j)`. Given a string S of length n, and two indices i, j such that $i \leq j$, return the string containing characters of s in the range $[i, j)$, namely $S[i, i+1, \ldots, j-1]$. For example, if $S =$ "abc", $i = 0$, $j = 2$, the substr function should return "ab".

```
// Extracts a substring from s between two indices i, j - 1.
// Effect: allocate memory
char* substr(const char* s, int i, int j);
```

9. Implement a vending machine module capable of handling item stock and purchases with transaction history. Provide implementations for each of the following functions as described:

   Header: `https://pastebin.com/CQfCUuE6`
   Implementation & Starter: `https://pastebin.com/AbGJHDz4`