Name: ______ Login: ______ Section: ○ 1:30 ○ 3:00 v1.2

HW01: Complexity Analysis

Objectives: You will practice analyzing algorithms and expressing their asymptotic complexity in terms of Big-O. To help you understand the nature of asymptotic complexity more broadly, you will analyze some algorithms that are not expressed in C code.

Rules

Expressions of asymptotic complexity must be *tight* and *simplified*. To simplify, remove coefficients, as well as lower-order terms in any sum.

Example: Suppose you were asked to give a Big-O expression for f(n) where $f(n) = 6n^2 + 3n + 5$. Writing $O(6n^2 + 3n + 5)$ would not receive credit because that is not simplified. Writing $O(2^n)$ would also not receive credit because, although f(n) is $O(2^n)$ by the definition of Big-O, that is not a tight bound. The correct answer would be $O(n^2)$.

Please write very neatly. Answer must be contained within the box. Anything outside box will not be considered.

Do not use GenAI (i.e., ChatGPT, etc.) to obtain answers. You may ask general questions to GenAI (or people) to strengthen your understanding of concepts.

We deliberately chose some questions for which we know ChatGPT will give incorrect answers. (We tested.)

For questions that ask for an explanation, just give 1-2 sentences in your own words demonstrating that you understand your answer.

Submit via Gradescope by Wednesday, September 6, 2023 at 11:59 PM. Do not submit pages 5-7.

Q1. Unshuffle password (part 1)

You encrypt important documents using passwords, which you store in a separate file. A virus has sabotaged your file by randomly shuffling the characters in each password. You know which characters are included, but the order is unknown.

Other than that, your password file is intact. For each password, you know the characters contained, but not the order. To unshuffle an n-character password, you manually type every possible ordering until you find the one that can decrypt the corresponding document. You type at a rate of 3 characters/second (plus Enter). It takes 1 second to determine if a given password can decrypt the document. The system does not limit the number of attempts. Passwords must be at least 8 characters, but there is no maximum.

Example: For one of the documents, the original password was "pa$sword" (n=8). The shuffled password is "rsodp$aw". You go through each of the 40320 possible orderings until you find the one that can decrypt your file.

$f_{unshuffle}(n)$ is the time to unshuffle an n-character password using this method. Give the asymptotic complexity of f(n).

$f_{unshuffle}(n)$ is O(______).

Explain: ______

Q2. … (part 2) − Type possible passwords automatically

You automate the process using QuickTyper, a program that can automatically type text into other programs. Instead of you typing the passwords by hand, the script goes through each possible reordering until it finds the one that can decrypt your document. QuickTyper types the characters at a rate of 100 characters/second. (All other details remain the same.)

$f_{quicktyper}(n)$ is the time to unshuffle an n-character password using QuickTyper.

$f_{quicktyper}(n)$ is O( ).

Explain:

Q3. … (part 3) – Consider password length limit.

We now have InstantTyper, which can type the characters of a password in a single time step. For example, typing 5 characters will take the same time as typing 1,000,000 characters. (All other details remain the same.)

$f_{instanttyper}(n)$ is the time to unshuffle an n-character password using InstantTyper.

$f_{instanttyper}(n)$ is O( ).

Explain:

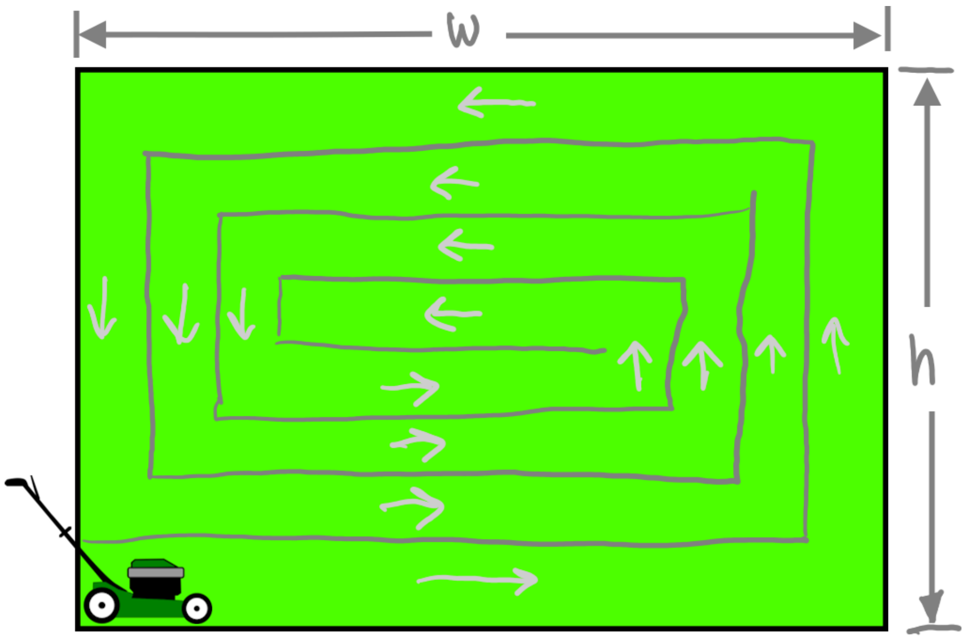
Q4. Lawn mowing

You mow rectangular yards with your lawn mower (pictured below), following the pattern shown below.

f(w,h) is the amount of gas (in gallons) used by the mower to mow a lawn of those dimensions.

f(w,h) is O(   ).

Explain:



Q5. Ternary search

On page 5, you will find an implementation of a ternary search. `is_string_in_array(…)` searches an array of strings and returns true or false depending on whether the given string was found. It works like binary search, except at each step, the array is partitioned into three subarrays instead of two.

We have instrumented the code to count the number of compare operations. Page 6 shows the output, which tells you how many times `strcmp(…)` is called when searching arrays of various lengths for the first element in the array.

The following questions refer to the ternary search algorithm, as implemented in `is_string_in_array(…)`. Assume the length of each string in the array is limited to some constant (i.e., calls to `strcmp(…)` are constant time operations).

$f_{strcmp}(n)$ is the number of compare operations (calls to `strcmp(…)`) to search an array of size n for the first element in that array.

$f_{strcmp}(n)$ is O( ).

Explain:

$f_{runtime}(n)$ is the total time to to search an array of size n for the first element.

$f_{runtime}(n)$ is O( ).

Explain:

$f_{any}(n)$ is the total time to search an array of size n for *any* element.

$f_{any}(n)$ is O( ).

Explain:

$F_{notpresent}(n)$ is the total time to to search an array of size n for an element that does not exist in that array (i.e., calling `is_string_in_array(strings, "NOT_PRESENT_IN_THE_ARRAY", n)`).

$f_{notpresent}$ (n) is O( ).

Explain:

ternary_search.c

```c
#include <stdio.h>   // for printf(…)
#include <stdlib.h>  // for EXIT_SUCCESS
#include <stdbool.h> // for bool, true, false
#include <string.h>  // for strcmp(…)
#include <assert.h>  // for assert(…)

// Instrument ternary search code to track the number of calls to strcmp(..).
#define strcmp (_g_strcmp_call_ct++, strcmp)
static unsigned int _g_strcmp_call_ct = 0;  // <<<<< GLOBAL MUTABLE - See note on page 7. <<<<<
// Rationale: Global/mutable unsigned int allows us to instrument code without modifying it.

bool is_string_in_array(char const* string, char const* const* strings, size_t array_len) {
    if(array_len == 0) {
        return false;
    }
    else if(array_len == 1) {
        return strcmp(strings[0], string) == 0;
    }
    else {
        // DIVIDE -- partition 'strings' into three equal or near-equal subarrays.
        // Each subarray size will be equal or one less than preceding subarray(s).
        //
        // First third of the array
        size_t sub_array_0_len = (array_len + 2) / 3;
        char const* const* sub_array_0 = strings;
        assert(sub_array_0_len >= 1);

        // Second third of the array
        char const* const* sub_array_1 = sub_array_0 + sub_array_0_len;
        size_t sub_array_1_len = (array_len + 1) / 3;
        assert(sub_array_1_len >= 1);
        assert(sub_array_0_len - sub_array_1_len <= 1);

        // Last third of the array
        char const* const* sub_array_2 = sub_array_1 + sub_array_1_len;
        size_t sub_array_2_len = array_len / 3;
        assert(sub_array_0_len - sub_array_2_len <= 1);
        assert(sub_array_1_len - sub_array_2_len <= 1);
        assert(sub_array_2_len >= 1 || (array_len == 2 && sub_array_2_len == 0));
        assert(sub_array_0_len + sub_array_1_len + sub_array_2_len == array_len);

        // CONQUER -- Recursive calls to obtain result
        if(strcmp(string, sub_array_1[0]) < 0) {
            // Search in first third of array.
            return is_string_in_array(string, sub_array_0, sub_array_0_len);
        }
        else if(strcmp(string, sub_array_2[0]) < 0) {
            // Search in second third of array.
            return is_string_in_array(string, sub_array_1, sub_array_1_len);
        }
        else {
            // Search in last third of array.
            return is_string_in_array(string, sub_array_2, sub_array_2_len);
        }
    }
}

int main(int argc, char* argv[]) {
    char const* const strings[] = { "apricot", "banana", "cherry", "date", "elderberry", "fig",
        "gooseberry", "huckleberry", "imbe", "jackfruit", "kiwi", "lime", "mango", "nectarine",
        "orange", "peach", "quince", "raspberry", "strawberry", "tangerine", "ugli fruit",
        "vanilla", "watermelon", "xigua", "yangmei", "zinfandel" };

    size_t max_array_len = sizeof strings / sizeof strings[0]; // only for arrays on stack
    assert(max_array_len == 26);  // There are 26 strings in the array.

    printf("array_len  num_compares\n");  // print header row
    for(size_t array_len = 1; array_len <= max_array_len; array_len++) {
        _g_strcmp_call_ct = 0;  // reset counter
        bool was_found = is_string_in_array("apricot", strings, array_len);
        assert(was_found);
        printf("%-9zd  %u\n", array_len, _g_strcmp_call_ct);
    }

    return EXIT_SUCCESS;
}
```

DO NOT SUBMIT THIS PAGE

Output from running ternary_search

```
$ gcc ternary_search.c -o ternary_search

$ ./ternary_search
array_len  num_compares
1          1
2          2
3          2
4          3
5          3
6          3
7          3
8          3
9          3
10         4
11         4
12         4
13         4
14         4
15         4
16         4
17         4
18         4
19         4
20         4
21         4
22         4
23         4
24         4
25         4
26         4

$
```

Translation

- Calling `is_string_in_array("apricot", strings, 1)` resulted in 1 call to `strcmp(…)`. In other words, searching an array of size 1 for the first element in that array resulted in 1 call to `strcmp(…)`.
- Calling `is_string_in_array("apricot", strings, 2)` resulted in 2 calls to `strcmp(…)`. In other words, searching an array of size 3 for the first element in that array resulted in 2 calls to `strcmp(…)`.
- Calling `is_string_in_array("apricot", strings, 3)` resulted in 2 calls to `strcmp(…)`. In other words, searching an array of size 3 for the first element in that array resulted in 2 calls to `strcmp(…)`.
- Calling `is_string_in_array("apricot", strings, 4)` resulted in 3 calls to `strcmp(…)`. In other words, searching an array of size 4 for the first element in that array resulted in 3 calls to `strcmp(…)`.
- Calling `is_string_in_array("apricot", strings, 5)` resulted in 3 calls to `strcmp(…)`. In other words, searching an array of size 5 for the first element in that array resulted in 3 calls to `strcmp(…)`.
- […]
- Calling `is_string_in_array("apricot", strings, 9)` resulted in 3 calls to `strcmp(…)`. In other words, searching an array of size 9 for the first element in that array resulted in 3 calls to `strcmp(…)`.
- Calling `is_string_in_array("apricot", strings, 10)` resulted in 4 calls to `strcmp(…)`. In other words, searching an array of size 9 for the first element in that array resulted in 4 calls to `strcmp(…)`.
- Calling `is_string_in_array("apricot", strings, 11)` resulted in 4 calls to `strcmp(…)`. In other words, searching an array of size 11 for the first element in that array resulted in 4 calls to `strcmp(…)`.
- […]
- Calling `is_string_in_array("apricot", strings, 26)` resulted in 4 calls to `strcmp(…)`. In other words, searching an array of size 26 for the first element in that array resulted in 4 calls to `strcmp(…)`.

DO NOT SUBMIT THIS PAGE

Explanation of the code in ternary_search.c

Q: What is `strcmp(…)`?

A: `strcmp(…)` is a standard library function that compares strings. `strcmp(s1, s2)` returns 0 if `s1` and `s2` are equal, a value < 0 if `s1` comes before s2, or a value > 0 if `s1` comes after s2.

Q: Why does `main(…)` pass the same array, but with different values for `array_len`?

A: It is simulating what would happen if you passed arrays of different length. The callee is unaware of any elements beyond `strings[array_len - 1]`.

Q: This code uses a global mutable (non-constant) variable (`_g_strcmp_call_ct`). Is that okay?

A: Yes. This global mutable variable (`_g_strcmp_call_ct`) allows us to track the behavior of `is_string_in_array(…)` without modifying its code. It is safe because it is only referred to within a narrow section of code in one file, and it is an `unsigned int` (no direct issues with memory faults or leaks).

Q: When may we use global mutable variables?

A: For this class, do not use global mutable variables, except for possibly instrumenting code (as we have done here), unless an assignment description specifically instructs you to do so (unlikely), and you follow a few rules to mitigate the risks: (1) Name the variable beginning with '_g_' to make it clear that the variable is global and private to this file. (2) Declare the variable as static to limit its scope to the current file. (3) State your rationale in a comment. (4) Global mutable variables may not refer to memory on the heap or anything else that must be freed or closed. This reduces the risk of leaks and memory faults. (Credit: Rule #3 is adapted from Google's C++ style guide.)

Q: Are global mutable variables always bad?

A: In the future, you might work on projects where global mutable variables are necessary (i.e., for background infrastructure). However, you should generally avoid using global mutable variables unless you can articulate why it is necessary and safe, and the people you are working with are okay with it.

Q: What about global constant variables (e.g., `const int MAX_STRING_LENGTH = 16;`)?

A: Global constants are fine and encouraged.

Q: How does `#define strcmp (_g_strcmp_call_ct++, strcmp)` work?

A: In short, that makes it so every time `strcmp(…)` is called, we first increment `_g_strcmp_call_ct`. You do not need to understand it any more deeply than that. For those who are curious, here is how it works. That line uses the preprocessor to replace every occurrence of `strcmp`—below that point in the current file only—with `(_g_strcmp_call_ct++, strcmp)`. It uses the comma operator, which is seldom used in C. When `(expr1, expr2)` is encountered in your code, both expressions are evaluated, but the value of the whole thing (`(expr1, expr2)`) is just the value of `expr2`; although `expr1` is evaluated, its resulting value is discarded. For example, if you had `int a = (foo(), bar());`, both `foo()` and `bar()` would be called, but regardless of what they returned, a would be set to the return value of `bar()`. In this case, it means that whenever `strcmp` is encountered, we first increment `_g_strcmp_call_ct`, but we don't do anything with its value at that point. Instead, `(_g_strcmp_call_ct++, strcmp)` evaluates to the address of the `strcmp(…)` function, so that the function call works the same as if we had just called `strcmp(…)` normally.

DO NOT SUBMIT THIS PAGE