Lab 3: Fundamental Algorithms

CSE/IT 113

CSE NMT

> Programming is learned by writing programs.
>
> — Brian Kernighan

> [Computer science] is not really about computers – and it's not about computers in the same sense that physics is not really about particle accelerators, and biology is not about microscopes and Petri dishes...and geometry isn't really about using surveying instruments. Now the reason that we think computer science is about computers is pretty much the same reason that the Egyptians thought geometry was about surveying instruments: when some field is just getting started and you don't really understand it very well, it's very easy to confuse the essence of what you're doing with the tools that you use."
>
> — Hal Abelson, 1986

> The purpose of computing is insight, not numbers.
>
> — Richard Hamming, 1962

Introduction

In this lab you will perform various fundamental tasks on a list. You will use arrays to represent the list. Later in this class you will learn different ways to represent lists (i.e. linked list, doubly linked list).

Preliminaries

Header Files

The tarball contains three files you will use for this lab: `lab3.c`, `array.h`, and `array.c`. The `*.c` files are source code files, and the `*.h` file is known as a header file. Header files contain function prototypes, `#define`s and other preprocessor directives.

You will split your code across multiple files in the following way:

1. `lab3.c` will contain only the `main()` function of your program. That is the logic of your program. Notice how it has a line at the top of the file to include the `array.h` file. This tells the compiler to add the function prototypes from `array.h`
2. `array.h` contains the function prototypes of the functions you will implement.
3. The definition of the functions are in `array.c`. Note that `array.c` includes `array.h`, this is so you can put preprocessor directives in `array.h` and they can be used in `array.c`

A couple of things to note about the given files. They include more Doxygen annotations (@file, etc) at the top of the file. Every C file you turn in from now on needs to have this comment block with the tags filled out at the top of file. Make sure you fill these tags out with the correct information. **You should update the tags now**.

The header file `array.h` includes what is known as an include guard, which prevents multiple copies of the file being processed by the compiler. At line 19 in `array.h`, if the variable `ARRAY_H_` is undefined the preprocessor will continue processing the file `array.h`. The first time the file is processed the variable `ARRAY_H_` is defined preventing further reads of the file by other files that are being compiled. At line 27, the `#ifndef` block is terminated.

Compiling Programs with Multiple Files

When your program is split across multiple files, you have to compile the program in a different manner. The first step is to create an *object* file for every C source file (`*.c`) that does not have a main function.

```
1 $ gcc -g -Wall -c array.c
```

The command creates the file `array.o`. You can see this by doing a `ls` in the current directory.

Once you have created all the object files your `main()` function uses, the next step is to link in all the object files (`*.o`) to the file with the `main()` function.

```
$ gcc -g -Wall lab3.c array.o -o lab3
```

The command creates the executable file named `lab3`, which you can run with the following command:

```
$ ./lab3
```

Pseudo Code

Pseudo code is an informal high-level description of an algorithm. Pseudo code is computer language neutral with the idea being a programmer can translate the algorithm to the programming language of their choice. Pseudo code focuses on what is important in the algorithm to solve a problem, not the implementation details. While it is programming language neutral, pseudo code still uses the ideas of variables and assignment, flow control (if-then) and looping (while or for loops).

Variables are represented with a single letter, but the data type is not given. Often standard math symbols are used. For instance x, y, z are used for real numbers and m and n for integers. Often the type is implicit from the context of the algorithm.

Assignment of variables is made via the `=` operator. For example to assign 5 to x you would write `x = 5`.

Conditional execution (flow-control) is represented by `if - else if - else` blocks. `//` is a comment. The body of the if, else if, and else is indicated by indentation

```
if condition
    //steps to perform if the condition is true
else if condition
    //steps to do if the else-if condition is true
else
   //steps to perform if the conditions in the
   //if and else-if are false
```

For conditions logical operators are spelled out (and, or), and math notation is often used for relational operators ($<, \leq, >, \geq$)

Looping is accomplished via `for` statements or `while` while statements.

```
for i = 1 to n  //in C for (i = 1; i <= n; i++)
    j  = j + i
```

```
//or as a while loop
i = 1
while (i ≤ n)
    j = j + i
    i = i + 1
```

Note there is usually no C-shorthand syntax in pseudo-code as it is supposed to be language independent and as the programmer you should be able to translate `j = j + i` to C syntax `j += i;` and the line `i = i + 1` to the C increment operator `i++`.

Again the indentation indicates the body of the `for` or `while` loop.

Arrays are represented using capital or lower case letters followed by a pair of brackets. The brackets are used in two different ways. The following syntax indicates an array which contains N elements `A[1..N]` or `a[1..n]`. If there is just a single number or letter in between the brackets, it indicates a specific element of the array. For example, `A[3]` indicates the third element of array A, while `a[i]` indicates the i-th element of array a.

It is important to note that in pseudo code array indexing begins with 1, while in C it begins with 0.

The Lab

You will write four programs for this lab. The first program implements a number of fundamental array algorithms. You will split your code across three files. The second program is a variation of the first. The third program implements the game Rock-paper-scissors-lizard-Spock. The last program determines if credit cards are valid or not.

The steps of the lab are in *italics*.

Part I

1 Compile the sample code

1. *Untar the file lab3.tar.gz to a directory of your choice. Open the files* ***`array.c`****,* ***`array.h`****, and* ***`lab3.c`*** *with a text editor.*

2. *Compile the given code with the following two commands:*

```
$ gcc -g -Wall -c array.c
$ gcc -g -Wall lab3.c array.o -o lab3
```

3. *Run the program you just compiled.*

```
$ ./lab3
```

2 The size of the array

In `main()`, which is in the `lab3.c` file, the size of the array is determined and stored in a variable named `size`, which is of type `size_t`. You need to pass this value as an argument to all functions involving arrays.

3 Printing summary statistics

4. *Write a function* ***`print_summary()`*** *that prints out information about the array. Place in* ***`lab3.c`***

   As you keep working through the lab, keep adding parameters to the function `print_summary()` and print out the statistic with a label.

   At the moment, `print_summary()` should look like this:

```c
void print_summary(size_t size)
{
        /* use %zu for printing size_t types */
        printf("size of array = %zu\n", size);
}
```

4 Find the maximum of an array

You often want to know the maximum element in a list. To find the maximum element of two numbers a and b you can do the following:

Pseudo Code to find the maximum of two numbers

```
//find the max of two numbers a and b
max = a //set max to an initial value

if a < b
   max = b
```

For arrays you extend this idea, but you test max against every element in the array:

Pseudo Code to find the maximum element in an array

```
//A[1..N] an array with N elements
max = A[1] //set max to the value in the first array element
for i = 2 to N
   if A[i] > max
      max = A[i]
```

Function Prototypes

Functions always need a function prototype which you will put in `array.h` and a defintion, which you put in `array.c`. The definition contains the body of the function.

Finally, you call the function in `main()` or in other functions. Here place the call to `find_max()` in `main()` which is located in `lab3.c`. Make sure you store the return value of the function. You will use it to print the maximum value out. For example, in main do something like the following:

```
1 int max;
2 max = find_max(a, size);
3 print_summary(size, max);
```

5. *Write a function (**`find_max()`**) that finds the largest element in the array and returns its value. Store this value in **`main()`**. Print the maximum using **`print_summary()`**.*

5 Testing find_max() and creating test arrays

As you write and test `find_max()`, it is useful to use arrays with known answers. This makes finding errors in your functions easy. In general, to test your functions and that is all you are doing in `main()`, it is best to use a number of different arrays. Good test arrays are ones that consist of all zeros, all ones, simple data that is linearly ordered and randomly ordered. An array size of 10 is sufficient to test with.

It is a good idea to test the boundary cases of the array (first and last elements) and the middle of the array (everything else). For example, to test the find max function, test it with an array that has the maximum as the first element, the maximum as the last element, and the maximum somewhere in the middle.

6. *Add the following integer arrays of size 10 to **`main()`**: all zeros, all ones, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], and [10, 9, 8, 7, 6, 5, 4, 3, 2, 1], and [5, 7, 8, 4, 3, 10, 1, 4, 4, 5]. Test **`find_max()`** with all these arrays.*

6 Find the minimum value in an array

Finding the minimum value in a list is similar to finding the maximum element of an array.

7. *Write a function (**find_min()**) to find the smallest element in the array and return its value. Store this value in **main()**. Print the minimum using **print_summary()**. Test with various array data.*

7 Find the midpoint

The midpoint of two numbers is defined as

$$(a + b)/2$$

8. *Write a function (**midpoint()**) that finds the midpoint of min and max. Use integer division and return an integer. Store this value in **main()**. Print the midpoint using **print_summary()**. Test with various array data.*

8 Counting Elements

Counting mechanisms are used frequently in programming. Find the number of elements greater than a given number; find the number of elements less than a given number, etc. For instance, to find the number of A's in a given list of grades you count the number of scores greater than or equal to 90.

Pseudo Code Counting

Note: the for loop usually begins at 1 not 0 in pseudo code

```
//counts the number of elements in an array
//that is equal to x
//A[1..N], an array with N elements
count = 0

for i = 1 to N
   if A[i] == x
      count = count + 1
```

Notice how the value of `count` is independent of the array index, i.

Algorithms to find the number of items in an array with other ordering relationships is very similar.

9. *In your README, write pseudo code to show how you can find a count of elements that are 1) less than, 2) less than or equal to, 3) greater or equal to, or 4) greater than a given element.*

10. *Write a function* ***`get_count()`*** *that given a value finds the number of elements in the array that are less than, less than or equal, equal, greater than or equal to, or greater than the value in the array. Test with a variety of array data and values.*

Notes on programming `get_count()`

C lets you define constants using the preprocessor directive – `#define`. For example,

```
#define SIZE 10
```

When the you compile, the preprocessor replaces wherever it finds SIZE in your source code with the value of 10. A global find and replace feature. This lets you avoid using magic numbers in your code and makes your code much more readable and maintainable. You wanted to change SIZE to 100, you only need to change it one place. Define statements are typically added to header files (*.h).

For `get_count()` you are going to use constants as *flags* that let you determine the type of count (<, <=, ==, >=, >) you want to perform. The flags than can be used as part of switch statement.

To get you started, you could do the following:

```c
/* @file array.h */
/* the value of LT - less than - is immaterial as long as
   its an integer which can be used in a switch statement */

#define LT 10
```

```c
/* @file array.c */
 * @param type what kind of count to perform either LT, LE,
 *        EQ, GE, or GT
*/
int get_count(int a[], size_t size, int type, int x) {

        switch (type) {
        case LT: /* less than count */
                /* todo */
                break
        /* todo other cases */
        }

```

```
14          return count;
15  }
```

```
1   /* @file lab3.c */
2   int main(void)
3   {
4           /* the count of the numbers <= 7 in the array */
5           int le7 = 0;
6
7           int a[] = {9, 7, 5, 3, 2, 77, 8, 12, 4, 5, 11};
8
9           size_t size = sizeof(a)/sizeof(int);
10
11          /* finds the count of the elements in array <= 7 */
12          le7 = get_count(a, size, LE, 7);
13          return 0;
14  }
```

11. *Find all elements in the array that are less then the midpoint of the list and greater than the midpoint of the list. Store both values. Print the counts using* ***`print_summary()`***.

9 Linear search of an array

One way to see if an element is in the list, is to do a linear search. A linear search begins at the beginning of the list and moves through the list element by element looking for the desired element. If the element is found it returns the index of the element in the array otherwise it return -1 which indicates the element wasn't found.

Pseudo Code for a linear search

```
//x is the element you are searching for in the list A
//A[1..N], A has N elements

i = 1 //i is the index into an array of size N

while i ≤ N and x ≠ A[i]
   i = i + 1

if i ≤ n
   index = i //x was found in the list at A[i]
else
   index = -1 //x was not found in the list
```

12. *Write a function (**linear_search()**) that determines if a given value is in the list or not. The linear search function returns the index of the element if found; - 1 if not. See if the midpoint is in the list. Store the result in **main()**. Print the result using **print_summary()**, and print either "midpoint found in list" or "midpoint not found in list" as the case may be.*

10 Sum the elements in a list

Summing the elements in a list (an array) is similar to counting.

Pseudo Code Sum the elements in the list

```
//A[1..N], an array with N elements

sum = 0

for i = 1 to N
    sum = sum + A[i]
next i
```

13. *Write a function to find the sum of the elements in the array and return its value. Store this value in **main()**. Print the sum using **print_summary()**. Test with a variety of test arrays.*

11 Find the average

To find the average you first find the sum of all elements and divide by number of elements. As there is no guarantee that the division will result in an integer, the average is a real number.

Pseudo Code to find the average

Calling functions in pseudo code, is a lot like calling functions in C. The parenthesis after a name indicates it is a function and parameters are a comma separated list inside the parenthesis. Unlike C functions, no data types are explicitly given in pseudo code.

```
//A[1..N], an array of N elements

sum = sum(A[1..N]) //sums the elements of array A[1..n]
average = sum / N
```

14. *Write a function to find the average of the elements in the array and return its value. Store this value in* ***main()****. Print the average using* ***print_summary()****. Test with a variety of test arrays.*

12 Sorting

Sorting is fundamental to computer science. Once a list is sorted you can find an element in the list very efficiently using a binary search. You have to use a linear search approach if your data is not sorted.

Given the importance of sorting there are numerous sorting algorithms. You are going to implement two: a bubble sort and an insertion sort. Wikipedia has some nice animations of the sorts. Check them out. They will make the pseudo code easier to understand.

Pseudo code for Bubble Sort

One sorting algorithm is the bubble sort. So called because smaller elements "bubble" to the top of the list. While it is not the most efficient sorting algorithm, it is easy to implement. It uses a nest loop and swaps elements.

```
//A[1..N], an array of N elements

for i  = 1 to (N - 1)
    for j = N to (i + 1)
        if A[j - 1] > A[j]
            //swap elements
            tmp = A[j - 1]
            A[j - 1] = A[j]
            A[j] = tmp
```

15. *Write a function (****bubble_sort()****) to sort the elements of the array using a bubble sort. This function returns void, but your array will come back sorted. After you sort your array, print the array to verify that it was sorted. Test with a variety of test arrays.*

Pseudo code for Insertion Sort

Insertion sort accomplishes the same thing as bubble sort – an array in ascending order – it just does it differently.

Insertion sort works by comparing a[k] to the previous items in the array (a[1], a[2], ..., a[k - 1]), starting with a[k - 1] and moving towards a[i]. Whenever a[k] is less than

a preceding element, increment the index of the preceding element and move it one position to the right. As soon as a[k] is greater than or equal to an array element, insert a[k] to the right of the element. This is easier said in pseudo code.

```
//A[1..N], an array of N elements

for k  = 2 to N
    x = A[k]
    j = k - 1

    while j > 0 and A[j] > x
            a[j + 1] = a[j]
            j = j - 1

    a[j + 1] = x
```

16. *Write a function (`insertion_sort()`) to sort the elements of the array using an insertion sort. This function returns void, but your array will come back sorted. After you sort your array, print the array to verify that it was sorted. Test with a variety of test arrays.*

13 Reverse an array

At times you would like to reverse the order of the elements in array. For instance, if you have an array in ascending order, you can place them in descending order by reversing the elements. To do this for an array a of size n, you swap elements a[1] with a[n], a[2] with a[n - 1], etc. To swap elements you do the following:

```
//swap a and b

tmp = a
a = b
b = tmp
```

17. *In your README, write pseudo code to show how you can reverse an array.*

18. *Write a function `reverse()` that implements your pseudo code. After you reverse your array, print the array to verify that it was reversed. Test with a variety of test arrays.*

14 Find the median

Besides allowing for fast searching, sorting is useful in other settings. Once you have a *sorted* array, you can find the median of a list of numbers. Let n be the size of the array and using integer division, the median is defined as

if n is odd then the median (M) is the value of $((n+1)/2)$-th term.

If n is even then the median (M) is the value of
$[((n)/2)$-th term + $((n)/2+1)$-th term $]/2$

Pseudo code median

```
//A[1..N], an array with N elements
//A is sorted

if N mod 2 == 0 //n is even
    median = (A[N / 2] + A[(N / 2) + 1]) / 2 //integer division
else
    median = A[(N + 1) / 2] //integer division
```

19. *Write a function to find the median of the elements in the array and return its value. Store this value in* ***main()****. Print the median using* ***print_summary()****. Test with a variety of test arrays.*

15 Find the count of even terms

20. *In your README, write pseudo code to show how you can find the count of all even terms in an array.*

21. *Write a function (****even_count()****) that finds the count of all even numbers in the array. Store this value in* ***main()****. Print the even count using* ***print_summary()****. Test with a variety of test arrays.*

16 Find the count of odd terms

22. *In your README, write pseudo code to show how you can find the count of all odd terms in an array.*

23. *Write a function (****odd_count()****) that finds the count of all odd numbers in the array. Store this value in* ***main()****. Print the odd count using* ***print_summary()****. Test with a variety of test arrays.*

17 Find the count of numbers that are divisible by a given number

24. *In your README, write pseudo code to show how you can find the count of all terms in an array that are divisible by a number.*

25. *Write a function (`divisible_count()`) that finds the number of elements of an array that are divisible by a given number. Test this function with the given number set to the minimum value of the array. Store this value in `main()`. Print the divisible count using `print_summary()`. Test with a variety of test arrays.*

18 Running lab3

Make sure your `print_summary()` function has 13 parameters: the size, the max, the min, the midpoint, count less than the midpoint, count greater than the midpoint, the midpoint linear search, the sum, the average, the median, then number of even elements, the number of odd elements, the number of elements divisible by the min value.

After you have tested your functions and satisfied that they work with small data sets, open the file named `test_array.c` and copy the array named `a[]` into your main function and run your program. This is the data you will use when you run your script.

Part II

Adding structure to your data

Copy the file `lab3.c` to a file named `lab3_structure.c`.

Having to pass around 13 parameters to a function is a bit of a drag to say the least. And as you have probably noticed, all the summary information pertains to one array. If you ran the program with a different array you would have a different set of values. It is difficult to keep track of all the individual variables. To correct this situation, C has the ability to bundle related data into a cohesive unit. Such things are called structures in C. For instance, you could define the following structure to hold the size (number of elements in the array), the min, and max value of an array.

```
1 struct summary_t {
2         size_t size;
3         int min;
4         int max;
```

```
};
```

To use the structure, which is really just a user-defined data type. You would do something like this in `lab3_structure.c`

```c
#include <stdio.h>

struct summary_t {
        size_t size; /* use %zu for printing */
        int min;
        int max;
};

/* can pass structures to functions */
void print_summary(struct summary_t summary);

int main(void)
{
        /* this declares summary to be a variable
           of type struct summary_t */
        struct summary_t summary;

        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

        /* to access the members of the structure (or fields)
         * you use dot notation */
        summary.size = sizeof(a)/sizeof(int);
        summary.max = find_max(a, summary.size);
        summary.min = find_min(a, summary.size);
        print_summary(summary);

        return 0;
}

void print_summary(struct summary_t summary)
{
        printf("size = %zu\n", summary.size);
        printf("max = %d\n", summary.max);
        printf("min = %d\n", summary.min);
}
```

For the new version of lab3, you are to

1. *Create a structure that holds all 13 data types for the summary. This can go in the file `lab3_structure.c`*

2. *Modify the* ***print_summary()*** *function to use the structure rather than individual parameters.*

3. *Move all function calls from main to a function named* ***get_summary()****. You will have to pass the summary structure, the array, and the size of the array to* ***get_summary()****.* ***get_summary()*** *will call all the functions main previously did, including* ***print_summary()****.*

19 Running lab3_structure

Like before, test the changes with simple arrays. Once you are satisfied the changes work with small data sets, open the file named `test_array.c` and copy the array named `a[]` into your `main()` function and run your program. This is the data you will use when you run your script.

Part III

Rock-paper-scissors-lizard-Spock

The tarball contains a file named `rock-spock.c`. The file contains source code for the game rock-paper-scissors-lizard-Spock. Unfortunately, the game is not finished. Your job is to finish the program. First run the code to get a feel for what it does. Then add code to do the following in `rock-spock.c`:

1. Check to make sure the player enters a correct move.

2. Use `switch` statements to determine the winner of Rock-paper-scissors-lizard-Spock. See `http://en.wikipedia.org/wiki/Rock-paper-scissors-lizard-Spock` for the rules of the game. Implement the function `winner()`. The prototype is given. Do not change it. This function returns the winner.

3. Print out the winner. Implement the function `print_winner()`. The prototype is given. Do not change it. Use the provided phrases.

4. Do not add any other functions to the code. You don't need any more.

5. Capture the output of at least 10 plays in a script file named `rock-spock.script`

Part IV

Luhn's algorithm

Luhn's algorithm (`http://en.wikipedia.org/wiki/Luhn_algorithm`) provides a quick way to check if a credit card is valid or not. The algorithm consists of three steps:

1. Starting with the second to last digit (tens column digit), multiply every other digit by two.
2. Sum the digits of the products together with the sum of the undoubled digits. The sum of the digits of the products means if the doubled value is 14, the sum of digits is $1 + 4 = 5$.
3. If the total sum modulo by 10 is zero, then the card is valid; otherwise it is invalid.

For example, to check that the Diners Club card 38520000023237 is valid, you would start at 3, double it and double every other digit to give, writing the credit card number as separate digits:

6, 8, 10, 2, 0, 0, 0, 0, 0, 2, 6, 2, 6, 7

Next you would sum the digits of the products with the sum of the undoubled digits:

$6 + 8 + (1 + 0) + 2 + 0 + 0 + 0 + 0 + 0 + 2 + 6 + 2 + 6 + 7 = 40$

Note for 10, since it was doubled you sum its digits $(1 + 0)$.

The last step is to check if $40 \; mod \; 10 = 0$, which it does. So the card is valid.

Program Luhn's Algorithm

Write a program that implements Luhn's Algorithm for validating credit cards. The tarball contains a file named `credit.c` that gives you a start on solving this problem. A few points:

1. NUM_DIGITS is the number of digits in the credit card. You have to count the number of digits manually. Later, you will learn a way so the machine does this for you.
2. The variable `char visa[]` is a string. The function `convert_card` converts the string to a numeric array named `card`. The numeric array `card` is what you manipulate with your functions.

3. Remember to pass the size (or length) of the `card` array to every function you pass `card` to. The function `print_card`, prints out the array value at each index. This is useful for debugging.
4. Remember to separate the steps of the algorithm into separate functions.
5. Print a message to the screen if the card is valid or not.
6. Capture the output of the program into a script named `credit.script`

The card given is valid. If you change the last digit of the card to 2 it will become invalid. If you want to test with other credit cards, google "test credit cards". Just make sure that when you replace the card, you keep the new card in double quotes and that you change NUM_DIGITS to match the number of digits the card has.

Submission guidelines

Make sure you comment your code. Follow the Coding Style for comment style. Comment functions in the source code file.

For submission, you will upload a tar archive containing the following items:

- `lab3.c`
- `array.h`
- `array.c`
- `lab3.script`
- `lab3_structure.c`
- `lab3_structure.script`
- `rock-spock.c`
- `rock-spock.script`
- `credit.c`
- `credit.script`
- a `README`, make sure it includes pseudo code for the following algorithms:

  #9 Count LT

  #9 Count LE

  #9 Count GE

#9 Count GT

#17 Reverse Array

#20 Count Evens

#22 Count Odds

#24 Count elements divisible by a given number

The name of your tar archive and your source code file should be

`cse113_firstname_lastname_lab3.tar.gz`

Upload the tarball to Moodle before the start of the due date.