FAQs - DAA Lab (UE16CS255)

[Announcements](#_esx2n1cwqkt4)

[FAQs](#_2x3i2dgwm5zv)

[Things to do before coming to the lab:](#_xeoe2m60qwpe)

[Things to do in the lab:](#_1orjfqee349r)

[Programming language to be used:](#_5axvr31yvwv)

[How to build an executable from 3-file approach?](#_bbrbiefg430)

[Reading a string with spaces until a newline character](#_xcxrtpszw405)

[Non-standard feature of allocating array in stack](#_cs3496r6tc0c)

[Why doesn’t floor() and ceil() functions work](#_seimlwga1329)

[Passing a 2D array across functions safely](#_y4ubxm7js78g)

[What do we gain by using these input/output files?](#_bwmilrwt365u)

[Debugging tips](#_4csydqrrkv3r)

# Announcements

# FAQs

### Things to do before coming to the lab:

* Read the questions well enough.
* Prepare/download the input/output files for the test-cases.
* Write the program and see if it works for the given test-cases.
* Write the core function that implements the algorithm in the observation book.
* Carry input/output files and your source-code into your memory-stick/usb-drive.

### Things to do in the lab:

* If you’ve got permission to show output of previous lab, do it in the first ten minutes.
* Remaining time to be spent only on the task of session.
* Listen to any more instructions from the lab instructors.
* Execute and show output to the lab instructor. On approval, write the output (or any specific things asked) in your observation book. For larger test-cases, just write the input size and the time taken for the execution. Get it signed by the lab instructor.
* If the lab instructor asks for any more changes, do it.
* Answer the viva questions, if asked.

### Programming language to be used:

* C programming language is used for implementing all the algorithms in the lab.

### How to build an executable from 3-file approach?

* 1. Compile to generate ss.o object file

gcc -Wall ss.c -c

* 1. Compile to generate the executable file ss\_client.out

gcc -Wall ss\_client.c ss.o -o ss\_client.out -lrt

* 1. Execute by feeding the input file ss\_ip1.txt and output written to tmp

./ss\_client.out < s2\_ip1.txt > tmp

* 1. Test for correctness with the expected output

diff -wB tmp ss\_op1.txt

### Reading a string with spaces until a newline character

* 1. // to read a string into str up to 1000 characters.

// The string will have ‘\0’ in the end and not ‘\n’.

char str[1001];

scanf(" %[^\n]", str);

### Non-standard feature of allocating array in stack

* 1. Why does initializing array doesn’t work after reading its size by scanf()?

scanf("%d",&n);

int s[n];

Don’t do this even though it may work in a few C compiler because it’s a non-standard feature. Generally, keep all the declarative statements above any executable statement because decision about declarative statements is made at compile-time. So, s[n] could have initialized with some “unpredictable” value of n, not the you’ve fed into. A better way of achieving the same is,

int \*s;

scanf("%d",&n);

s = (int \*) malloc(n \* sizeof(int));

Make sure you “free(s);” when you don’t need that piece of memory block.

### Why doesn’t floor() and ceil() functions work

the way we expect in the following code?

p = ((int)(floor(n/2)));

q = ((int)(ceil(n/2)));

* 1. If n is an int, then the statement with ceil() doesn’t work the way you expect. That’s because n/2 is an integer division operation which is automatically floored. So, you’ll never get the actual ceil when n is odd. A correct way of achieving the same is as follows.

p = n/2;

q = ((int)(ceil(n/2.0)));

* 1. Another way of achieving the same is:

p = n/2;

q = n - p;

### Passing a 2D array across functions safely

* 1. Here’s one of the ways of getting it done. A 2D array of ints would have datatype as “**int \*\***”. Allocate an array of r number of “**int \***”, where r is the number of rows. That’s a pointer for each row. Now allocate c number of “**int**” for each row, where c is the number of columns. That way accessing an element **a[i][j]** is equivalent to **\*(\*(a+i)+j)**.
  2. //Allocates 2D array of ints having r rows and c columns  
     int\*\* alloc\_array2d\_int(int r, int c) {  
      int \*\*a;  
      int i;  
      a = (int \*\*) malloc(r \* sizeof(int \*)); //r rows  
      for(i = 0; i < c; i++) {  
      a[i]=(int\*)malloc(c\*sizeof(int)); //cols for ith row  
      }  
      return a;  
     }  
       
     //Frees an allocated 2D array having r rows  
     void free\_array2d(int \*\*a, int r) {  
      int i;  
      for(i = 0; i < r; i++) {  
      free(a[i]);  
      }  
      free(a);  
     }

### What do we gain by using these input/output files?

It was simpler to give inputs manually. Isn’t it?

* 1. Large inputs cannot be tested by manually typing in the terminal. Large inputs are necessary to observe the time taken by the algorithms. Algorithmic analysis is all for executing faster on large inputs.
  2. Multiple test-cases covering border cases can be tested in one go.
  3. To observe order of growth functions you studies in theory by increasing the input size systematically and measuring the executing time.
  4. Sooner than later you need to get used to it for other projects, interviews, and programming contests.

### Debugging tips

* 1. Extra printf statements help to catch the state of variables at the instances you want to observe. Make sure to put ‘\n’ at end of every such printf statements because sometimes printf statements doesn’t display on the terminal until a full line is sent to the stdout.
  2. Use gdb, which is GNU debugger. It’s a tool which takes an executable file and runs it in a controlled way, which allows you to analyze what happens step by step. For that, while compiling using gcc, add an option -g to your command. That adds necessary debugging information to your executable file (a.out in most of your cases). Then run gdb command with a.out as a command line parameter. That puts you into a gdb prompt. gdb has its own commands, which work in gdb prompt. Learn more about how to debug in gdb.