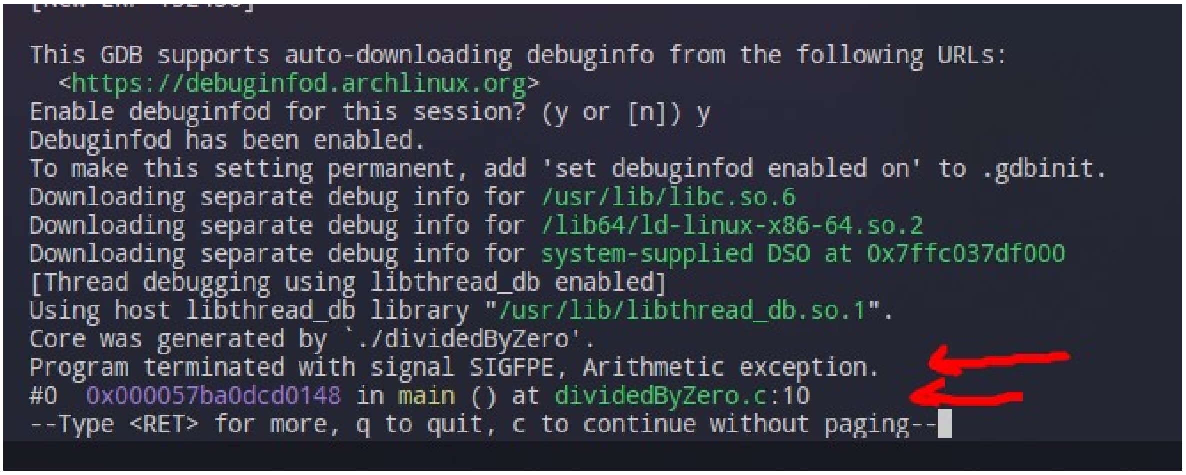
Operating Systems first task

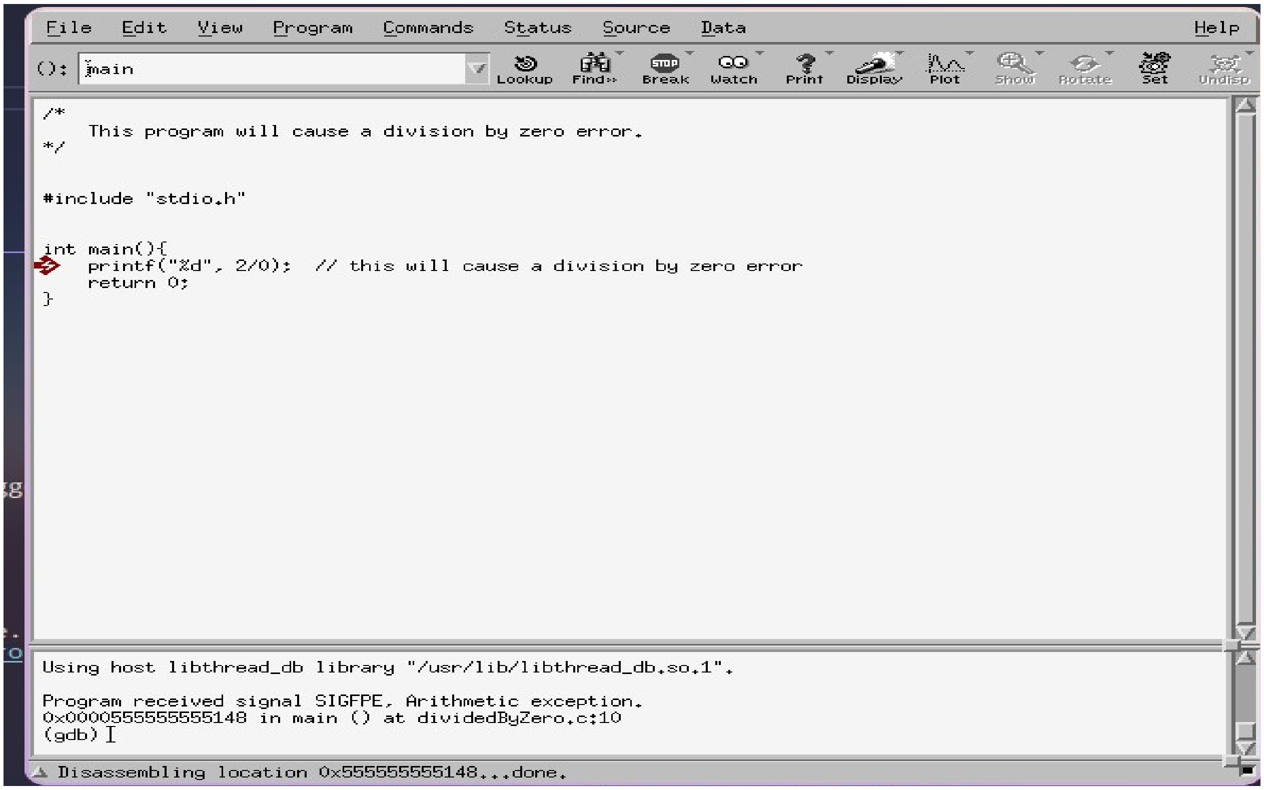
Made by: Tomer Shor and Adam Sin.
325511541, 322453689

Question 1:
Let’s make thing short and dive straight to the screeshots analysis:
RUN #1 – Division by Zero:
gdb debugger:
Command executed: $ coredumpctl gdb dividedByZero (dividedByZero is the executible).
Here we can see the gdb output. Lets break it down:
The first arrow indicates that the program failed due to arithmetic exception.
The second arrow shows that the issue accured in main function at line 10.



```
This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /usr/lib/libc.so.6
Downloading separate debug info for /lib64/ld-linux-x86-64.so.2
Downloading separate debug info for system-supplied DSO at 0x7ffc037df000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
Core was generated by `./dividedByZero'.
Program terminated with signal SIGFPE, Arithmetic exception.
#0  0x000057ba0dcd0148 in main () at dividedByZero.c:10
--Type <RET> for more, q to quit, c to continue without paging--
```

ddd debugger:
We used the command: $ ddd dividedByZero <coredump location>. We received this graphic debugger. We have been noticed by by the debugger that an error accured in the marked line, and the reason for it as printed at the box below is: “Arithmetic exception”.



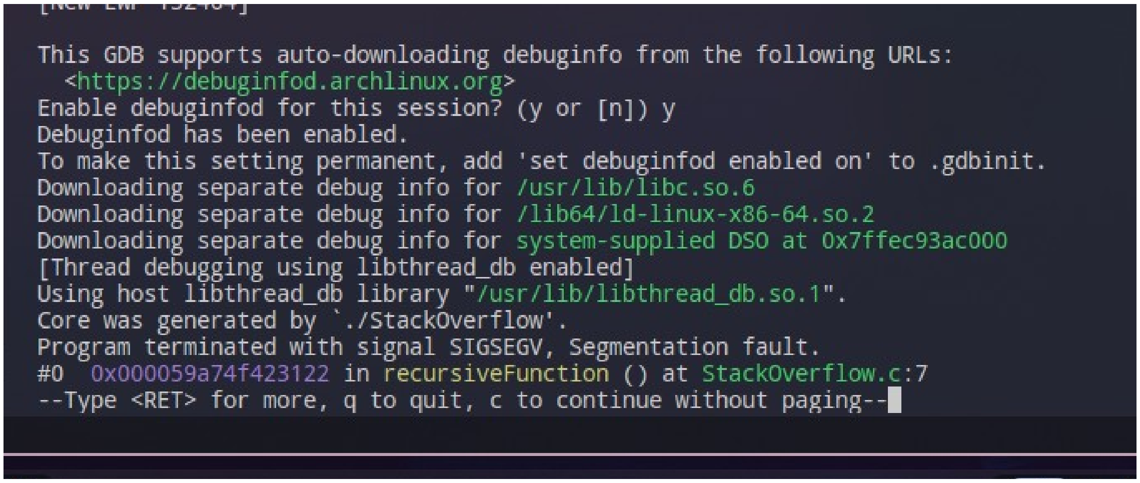
```
main
```

```
/*
    This program will cause a division by zero error.
*/

#include "stdio.h"

int main(){
    printf("%d", 2/0);  // this will cause a division by zero error
    return 0;
}
```

```
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGFPE, Arithmetic exception.
0x0000555555555148 in main () at dividedByZero.c:10
(gdb)
```

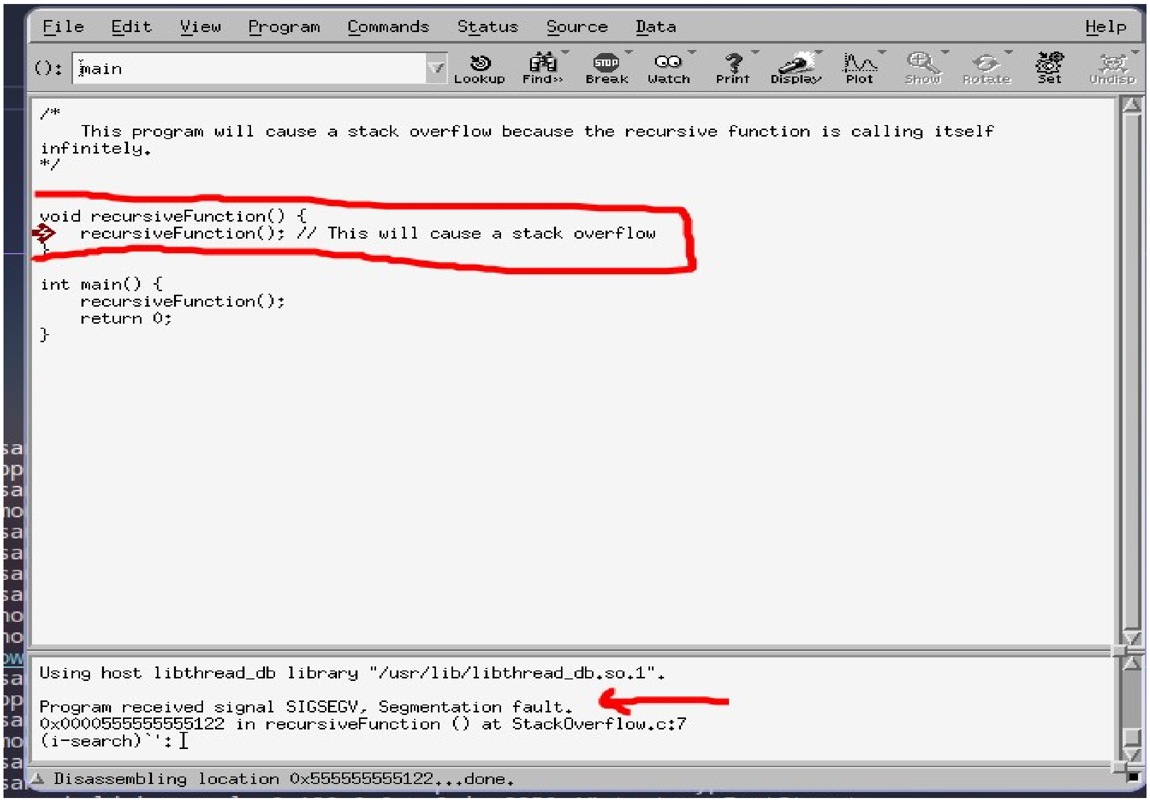
```
Disassembling location 0x555555555148...done.
```

RUN #2 – StackOverFlow (infinite recursion):
We ran the same command with the right adjustments.
The gdb debugger printed that the program crushed due to “Segmentation fault”.
The issue is loacted in recursivefunction at line number 7.



```
This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.archlinux.org>
Enable debuginfod for this session? (y or [n]) y
Debuginfod has been enabled.
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
Downloading separate debug info for /usr/lib/libc.so.6
Downloading separate debug info for /lib64/ld-linux-x86-64.so.2
Downloading separate debug info for system-supplied DSO at 0x7ffec93ac000
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/usr/lib/libthread_db.so.1".
Core was generated by `./StackOverflow'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000059a74f423122 in recursiveFunction () at StackOverflow.c:7
--Type <RET> for more, q to quit, c to continue without paging--
```

Then, we did the same proccess with ddd dubugger, and received this debug summary:
- The program failed due to the code wrriten in line marked.
- The reason is: “Segmentation fault”.



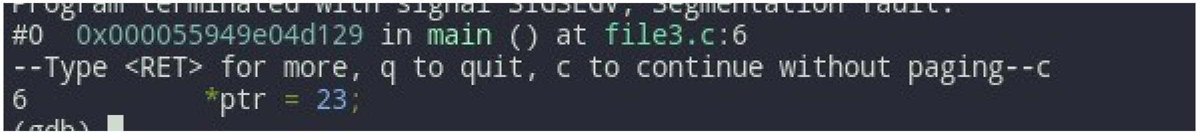
```
main
```

```
/*
    This program will cause a stack overflow because the recursive function is calling itself
infinitely.
*/

void recursiveFunction() {
    recursiveFunction(); // This will cause a stack overflow
}

int main() {
    recursiveFunction();
    return 0;
}
```

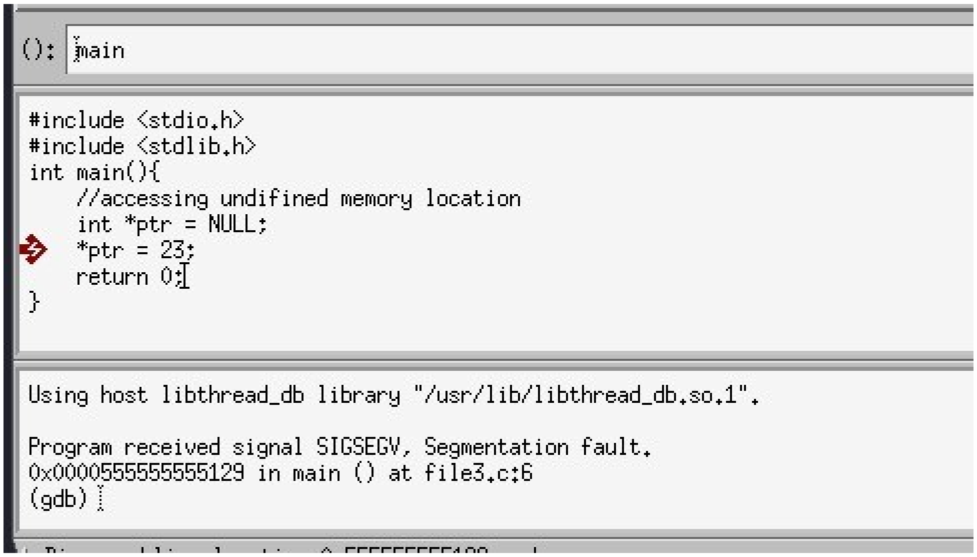
```
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555122 in recursiveFunction () at StackOverflow.c:7
(i-search)`':
```

```
Disassembling location 0x555555555122...done.
```

RUN #3 – Writing to undeclared address:
Just like in the previous examples:



```
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000055949e04d129 in main () at file3.c:6
--Type <RET> for more, q to quit, c to continue without paging--c
6           *ptr = 23;
(gdb) 
```



```
(): main
```

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    //accessing undifined memory location
    int *ptr = NULL;
    *ptr = 23;
    return 0;
}
```

```
Using host libthread_db library "/usr/lib/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x0000555555555129 in main () at file3.c:6
(gdb) 
```

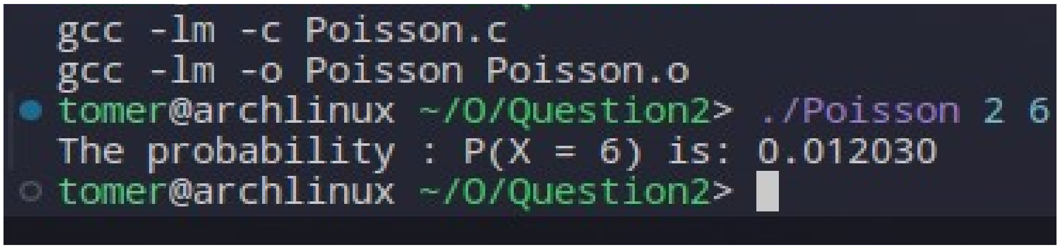
Question 2:

Starting up this question, we built the poisson function. Then, we made sure we received the arguments correctly.

Eventualy, we executed the following commands:

```
$ make
$ ./poisson 2 6
```

This is a Screenshot of a random test test using k=6, lambda = 2.



```
  gcc -lm -c Poisson.c
  gcc -lm -o Poisson Poisson.o
● tomer@archlinux ~/O/Question2> ./Poisson 2 6
  The probability : P(X = 6) is: 0.012030
○ tomer@archlinux ~/O/Question2> 
```

Question 3:

NOTE: There might be an issue with LD_LIBRARY_PATH. We linked the shared library by faced some issues running the main file. We have managed to run it after using the command:

```
$ export LD_LIBRARY_PATH=.
```

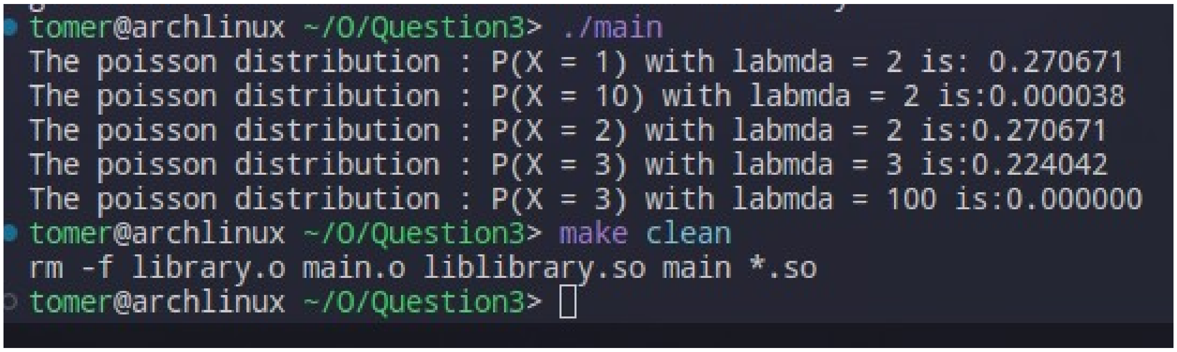
WE ALSO ADDED IT TO THE MAKEFILE. PLEASE MAKE SURE IT WORKS ON YOUR COMOUTER TOO!

Code explained:

We built library.c library. Then, we compiled it in the following way:

```
$ gcc -shared -o libmylib.so -fPIC library.c
$ gcc -o main main.c -L. -lmylib
$ ./main
```

The result was:



```
tomer@archlinux ~/O/Question3> ./main
The poisson distribution : P(X = 1) with labmda = 2 is: 0.270671
The poisson distribution : P(X = 10) with labmda = 2 is:0.000038
The poisson distribution : P(X = 2) with labmda = 2 is:0.270671
The poisson distribution : P(X = 3) with labmda = 3 is:0.224042
The poisson distribution : P(X = 3) with labmda = 100 is:0.000000
tomer@archlinux ~/O/Question3> make clean
rm -f library.o main.o liblibrary.so main *.so
tomer@archlinux ~/O/Question3>
```

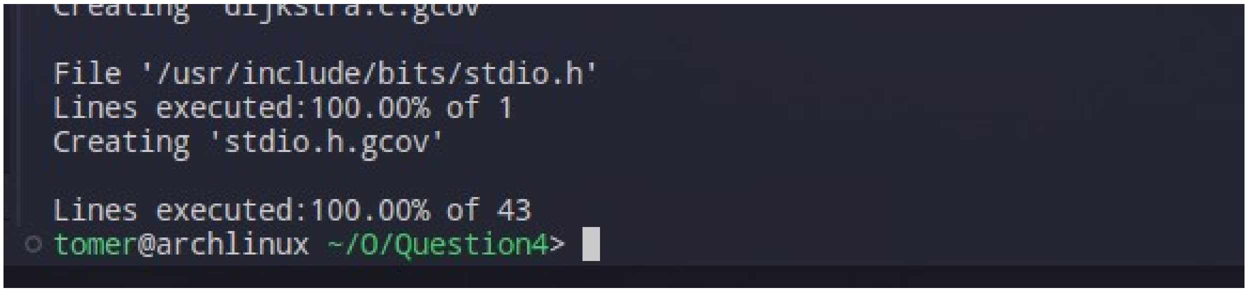
Put in mind that the makefile is built way pretier. We only showed the most basic commands in order for it to work.

Question 4:
Detailed explanations about implementation can be found in the code.
Let’s previue all the significant parts and changes we have made in our code. Parts to notice:

- We defined maximum number of vertices, and used dijkstra on user’s input.
- We have made a lot of test to check the code. The tests are in run_tests.sh file.

To run, we ran; $ make coverage
This command executed the code with all of the tests, and then added the command: gcov <code.c> which resulted the output shown. The output mainly shows that all of the code has been covered, that mean, we tested all wrong inputs, which have led the code to all of the error handler scopes.
This tool helped us confirm the necessity of the entire code.

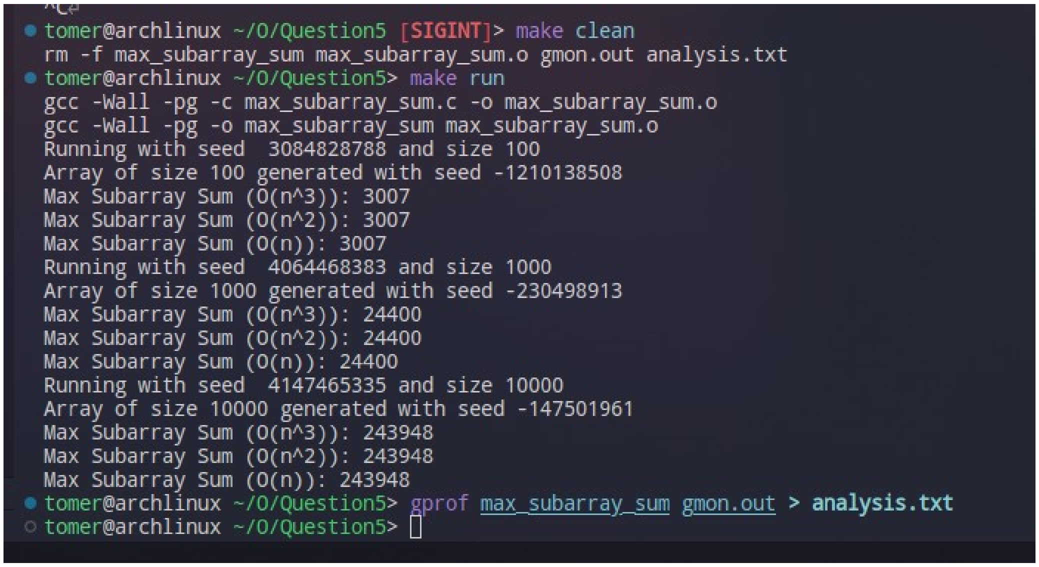
Output example:



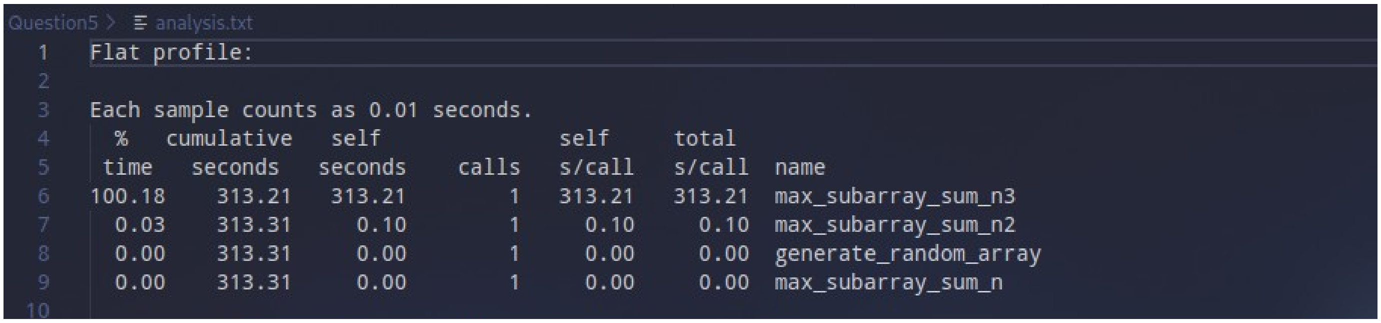
```
  Creating 'dijkstra.c.gcov'

  File '/usr/include/bits/stdio.h'
  Lines executed:100.00% of 1
  Creating 'stdio.h.gcov'

  Lines executed:100.00% of 43
○ tomer@archlinux ~/O/Question4>
```

Question 5:

This is the result of running max_sub_array problem with different ways to solve it.



```
^C⏎
● tomer@archlinux ~/O/Question5 [SIGINT]> make clean
  rm -f max_subarray_sum max_subarray_sum.o gmon.out analysis.txt
● tomer@archlinux ~/O/Question5> make run
  gcc -Wall -pg -c max_subarray_sum.c -o max_subarray_sum.o
  gcc -Wall -pg -o max_subarray_sum max_subarray_sum.o
  Running with seed  3084828788 and size 100
  Array of size 100 generated with seed -1210138508
  Max Subarray Sum (O(n^3)): 3007
  Max Subarray Sum (O(n^2)): 3007
  Max Subarray Sum (O(n)): 3007
  Running with seed  4064468383 and size 1000
  Array of size 1000 generated with seed -230498913
  Max Subarray Sum (O(n^3)): 24400
  Max Subarray Sum (O(n^2)): 24400
  Max Subarray Sum (O(n)): 24400
  Running with seed  4147465335 and size 10000
  Array of size 10000 generated with seed -147501961
  Max Subarray Sum (O(n^3)): 243948
  Max Subarray Sum (O(n^2)): 243948
  Max Subarray Sum (O(n)): 243948
● tomer@archlinux ~/O/Question5> gprof max_subarray_sum gmon.out > analysis.txt
○ tomer@archlinux ~/O/Question5> 
```



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total           
 time   seconds   seconds    calls   s/call   s/call  name    
100.18    313.21   313.21        1   313.21   313.21  max_subarray_sum_n3
  0.03    313.31     0.10        1     0.10     0.10  max_subarray_sum_n2
  0.00    313.31     0.00        1     0.00     0.00  generate_random_array
  0.00    313.31     0.00        1     0.00     0.00  max_subarray_sum_n
```

We can clearly see the difference in the running times of each function.

The max_sub_array problem which is implemented in running time of O(n^3), took 313 seconds to complete (way too much…). In compare to generating random array and the other methods to solve the problem, we can clearly see how bad this solving method is. Using profiling we were able to conculde these conclutions. theseconclusions

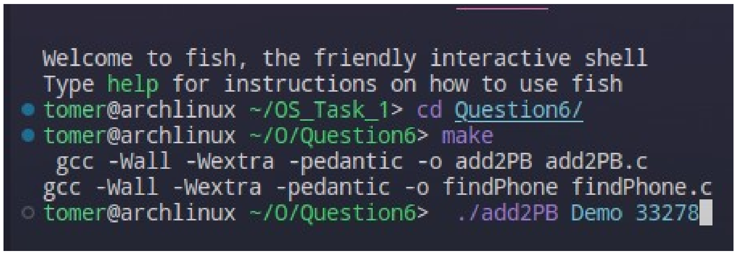
Question 6:

In this question we will cover the usage of some new function like: fork(), dup2(), execXX and few streams editors.

Implementation:
The program we built is responsible for adding people to the phone book and getting people's number from the note book.
We implemented this by using to programs: The first one is responsible for adding the people. The code is explained at the program's comments. (nothing new there).
The second program, which is responsible for extracting people's phone number, is implemented like so:
Firstly, we ask the user to enter a name (we assume that we receive a name only and not anything else). Then, a new process is created. In the added process, pipes are being used to transfer the output of program running in the process. Its being done by using dup2(), which redirects stdout to one side of the pipe. This way, after we perform execvp(grep, …), the output, which is usally being printed to stdout, will be stored in out pipe. After this, another process opens. In this process, we receive the data from the first pipe (by using read-end side of the pipe) and perform another function on it and store it in a pipe. (this time execvp(cut, …)) . Eventually, the data in the pipe (The phone number) is being transfered to the main process which prints it to the user.

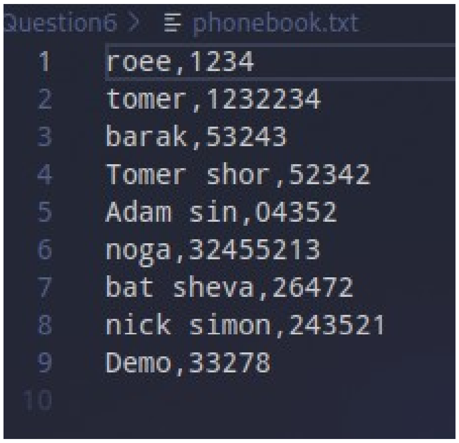
NOTE:
We have covered all of the reasonable cases of names (as mentioned in code comments).
We added a demo notebook.

Running example:
Lets add a random user to our phone book:



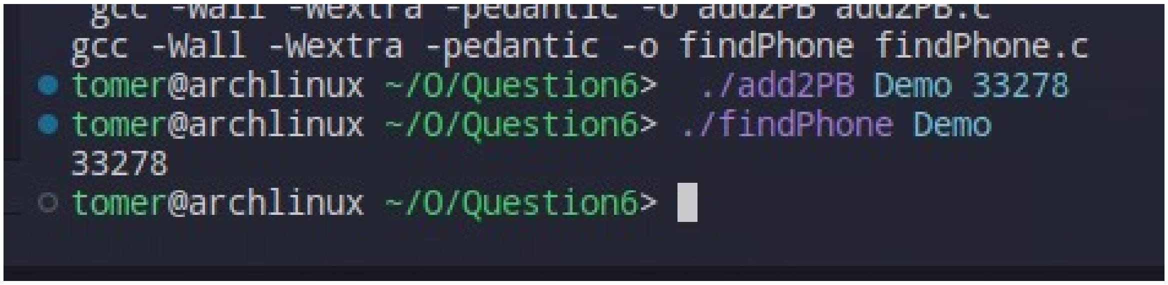
```
Welcome to fish, the friendly interactive shell
Type help for instructions on how to use fish
tomer@archlinux ~/OS_Task_1> cd Question6/
tomer@archlinux ~/O/Question6> make
 gcc -Wall -Wextra -pedantic -o add2PB add2PB.c
gcc -Wall -Wextra -pedantic -o findPhone findPhone.c
tomer@archlinux ~/O/Question6>  ./add2PB Demo 33278
```

Now, the phone book will store the user like so:



```
roee,1234
tomer,1232234
barak,53243
Tomer shor,52342
Adam sin,04352
noga,32455213
bat sheva,26472
nick simon,243521
Demo,33278

```

Now, lets try to get the number of Demo by using the following command:



```
 gcc -Wall -Wextra -pedantic -o add2PB add2PB.c
 gcc -Wall -Wextra -pedantic -o findPhone findPhone.c
● tomer@archlinux ~/O/Question6>  ./add2PB Demo 33278
● tomer@archlinux ~/O/Question6> ./findPhone Demo
 33278
○ tomer@archlinux ~/O/Question6> 
```

As we can see, we received the number of Demo as expected.