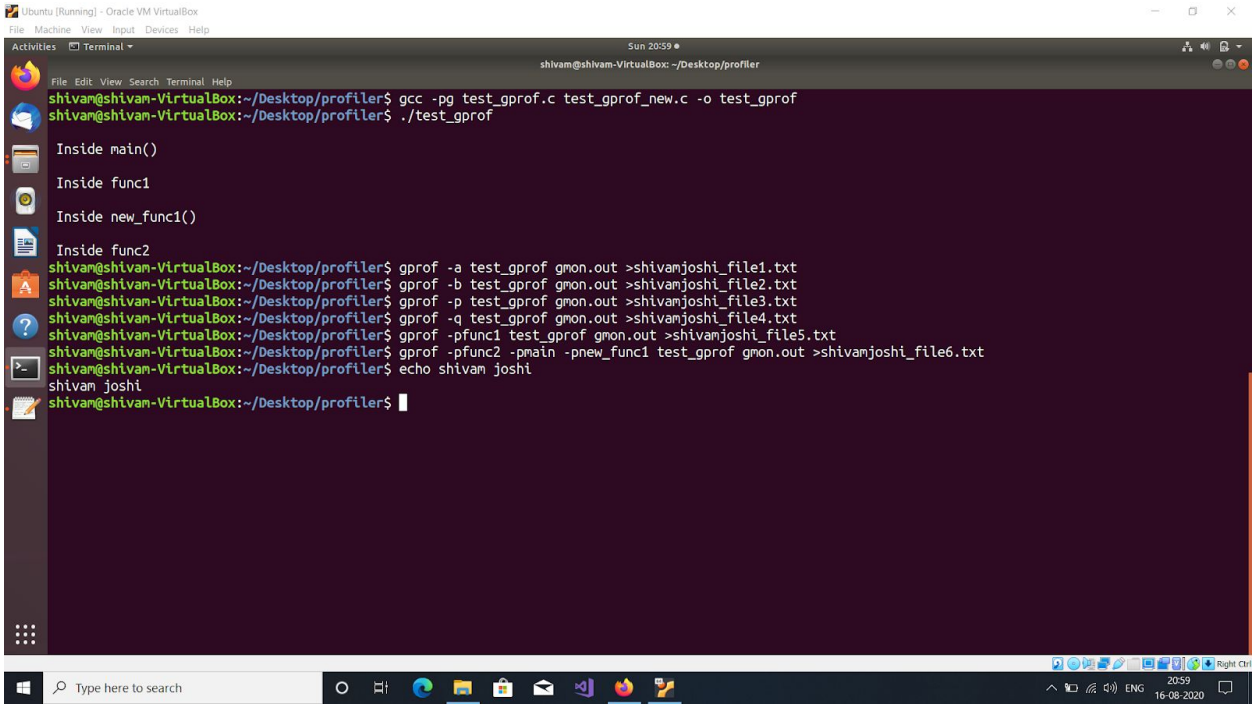
Question:1



Gprof -a is used to profile information of functions which are not declared statically.
Gprof -b Suppress verbose information that is generated in file
Gprof -p Generate only flat profile information
Gprof -q Generate only call graph information
Gprof pfunc1 Profiling information of only “func1”

Shivamjoshi_file1.txt:

Flat profile:
//shivam joshi
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
65.58	16.02	16.02	2	8.01	12.27	func1
34.87	24.53	8.52	1	8.52	8.52	new_func1
0.08	24.55	0.02				main

% time — the percentage of the total running time of the program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self   the number of seconds accounted for by this
seconds   function alone. This is the major sort for this
         listing.

calls   the number of times this function was invoked, if
         this function is profiled, else blank.

 self   the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
         else blank.

 total   the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
         function is profiled, else blank.

name   the name of the function. This is the minor sort
         for this listing. The index shows the location of
         the function in the gprof listing. If the index is
         in parenthesis it shows where it would appear in
         the gprof listing if it were to be printed.

                 Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.04% of 24.55 seconds

```
index % time   self  children   called  name
                                <spontaneous>
[1]      100.0  0.02   24.53            main [1]
         16.02  8.52    2/2             func1 [2]
-----------------------------------------------
         16.02  8.52    2/2             main [1]
[2]      99.9   16.02   8.52    2       func1 [2]
                8.52    0.00    1/1             new_func1 [3]
-----------------------------------------------
                8.52    0.00    1/1             func1 [2]
[3]      34.7   8.52    0.00    1       new_func1 [3]
-----------------------------------------------
```

 This table describes the call tree of the program, and was sorted by

the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:

index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children. Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the function into this parent.

children This is the amount of time that was propagated from
the function's children into this parent.

called This is the number of times this parent called the
function `/' the total number of times the function

was called. Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[2] func1 [1] main [3] new_func1

Shivamjoshi_file2.txt

Flat profile:
//shivam joshi
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
34.87		8.52	8.52	1	8.52	8.52 new_func1
34.38	16.91	8.39	1	8.39	8.39	func2
31.21	24.53	7.62	1	7.62	16.14	func1
0.08	24.55	0.02				main

Call graph

granularity: each sample hit covers 2 byte(s) for 0.04% of 24.55 seconds

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.02	24.53		main [1]
		7.62	8.52	1/1	func1 [2]
		8.39	0.00	1/1	func2 [4]

		7.62	8.52	1/1	main [1]
[2]	65.7	7.62	8.52	1	func1 [2]
		8.52	0.00	1/1	new_func1 [3]

		8.52	0.00	1/1	func1 [2]
[3]	34.7	8.52	0.00	1	new_func1 [3]

		8.39	0.00	1/1	main [1]
[4]	34.2	8.39	0.00	1	func2 [4]

Index by function name

[2] func1 [1] main
[4] func2 [3] new_func1

Shivamjoshi_file3.txt

Flat profile:
//shivam joshi
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
34.87		8.52	8.52	1	8.52	8.52 new_func1
34.38	16.91	8.39	1	8.39	8.39	func2
31.21	24.53	7.62	1	7.62	16.14	func1
0.08	24.55	0.02				main

% time — the percentage of the total running time of the program used by this function.

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call — the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name — the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in

the gprof listing if it were to be printed.

Shivamjoshi_file4.txt
//shivam joshi

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.04% of 24.55 seconds

```
index % time    self  children    called     name
                                             <spontaneous>
[1]     100.0    0.02   24.53                 main [1]
                 7.62    8.52       1/1           func1 [2]
                 8.39    0.00       1/1           func2 [4]
-----------------------------------------------
                 7.62    8.52       1/1           main [1]
[2]      65.7    7.62    8.52       1         func1 [2]
                 8.52    0.00       1/1           new_func1 [3]
-----------------------------------------------
                 8.52    0.00       1/1           func1 [2]
[3]      34.7    8.52    0.00       1         new_func1 [3]
-----------------------------------------------
                 8.39    0.00       1/1           main [1]
[4]      34.2    8.39    0.00       1         func2 [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:

index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this function by its children.

called This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

name The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function `/' the total number of times the function was called.  Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

Index by function name

[2] func1	[1] main
[4] func2	[3] new_func1

Shivamjoshi_file5.txt

Flat profile:
//shivam joshi
Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name

101.74 7.62 7.62 1 7.62 7.62 func1

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the gprof listing. If the index is
in parenthesis it shows where it would appear in
the gprof listing if it were to be printed.

Shivamjoshi_file6.txt

Flat profile:
//shivam joshi
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls s/call s/call name

50.69		8.52	8.52	1	8.52	8.52	new_func1
49.97	16.91	8.39	1	8.39	8.39	func2	
0.12	16.93	0.02			main		

% time — the percentage of the total running time of the program used by this function.

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

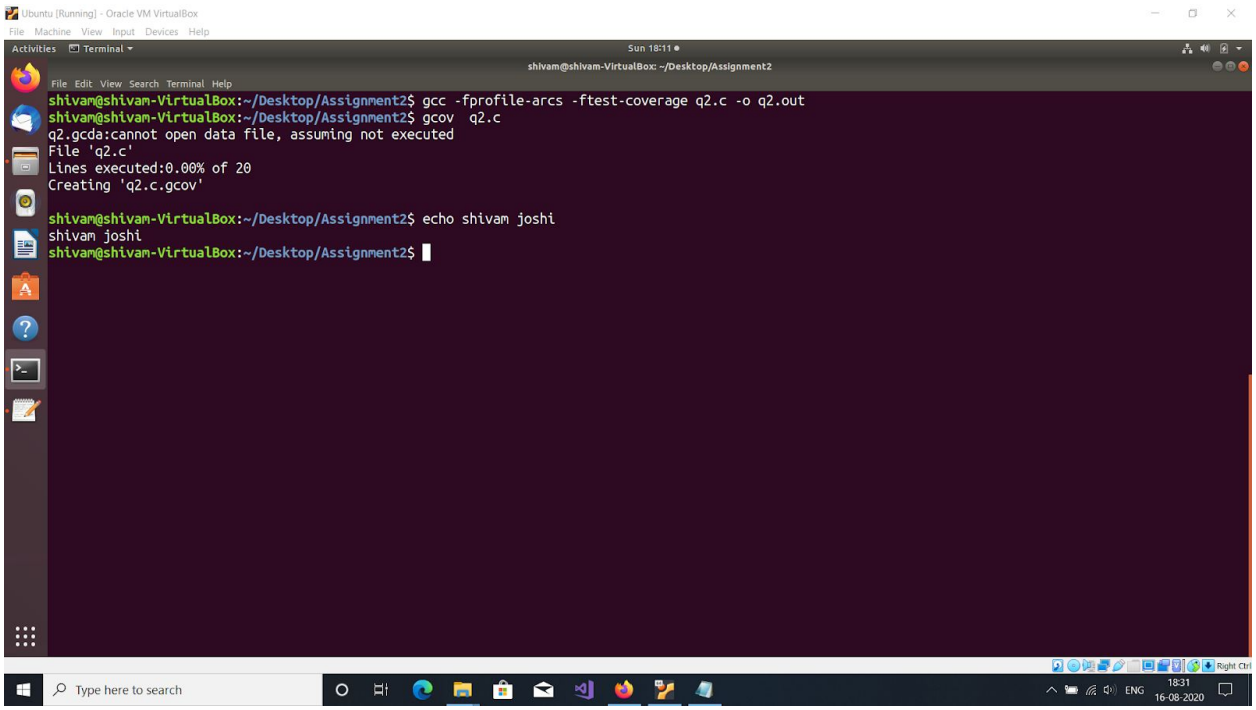
total ms/call — the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name — the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Question:2

```
//Shivam Joshi
#include <stdio.h>

int main (void)
{

    int i, total, num;

    total = 0;

    for (i = 0; i < 10; i++)
            total += i;

    if (total != 45)
                    printf ("Failure\n");
           else
           printf ("Success\n");

    num = 0;
    switch(num) {
    case 1: printf("1");
            break;

    case 2: printf("2");
            break;

    case 3: printf("3");
            break;

    case 4: printf("4");
            break;

    default: printf("0");
            break;
```

```
    }
    return 0;
}
```



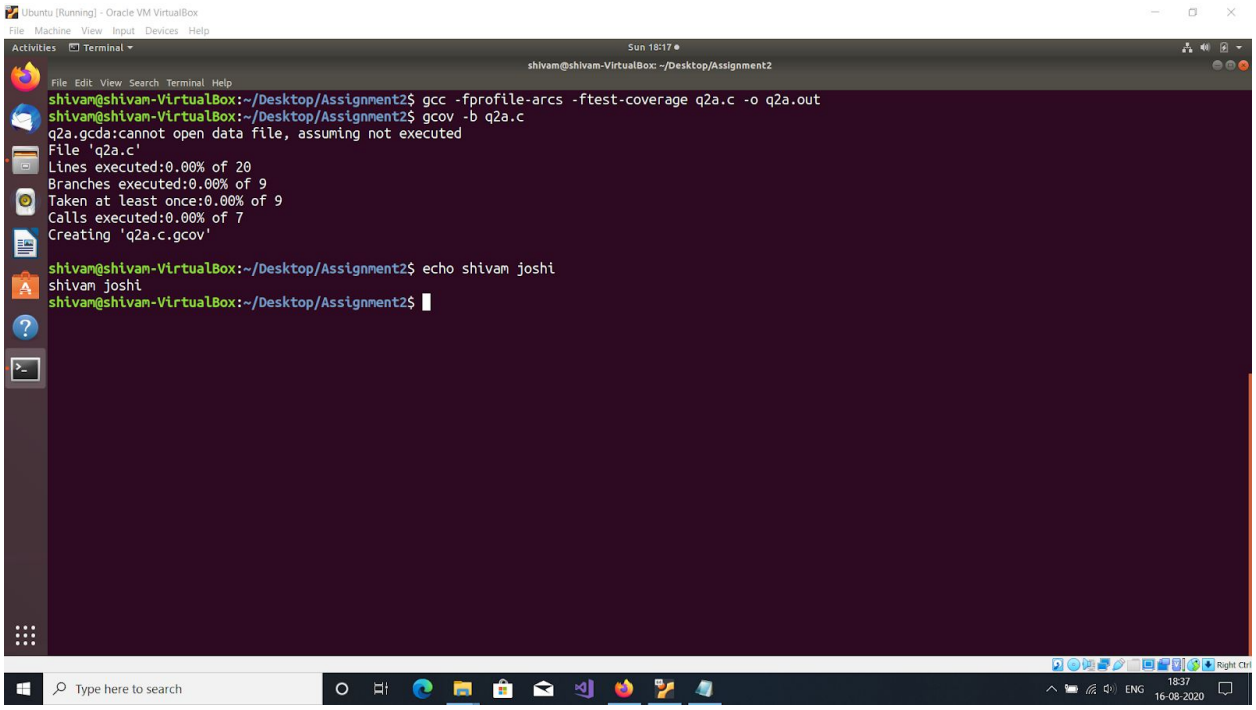
Compiler the file q2.c with Gcov with options **-fprofile-arcs -ftest-coverage**
This tells the compiler to generate additional information needed by gcov (basically a flow graph of the program) and also includes additional code in the object files for generating the extra profiling information needed by gcov.

Upon execution of instruction Gcov q2.c q2.c.Gcov file is created which is shown below

Output:The file *q2.c.gcov* contains output from **gcov**. Here is a sample:

```
        -:    0:Source:q2.c
        -:    0:Graph:q2.gcno
        -:    0:Data:-
        -:    0:Runs:0
        -:    0:Programs:0
        -:    1://Shivam Joshi
        -:    2:#include <stdio.h>
        -:    3:
    #####:    4:int main (void)
        -:    5:{
        -:    6:
        -:    7:    int i, total, num;
        -:    8:
    #####:    9:    total = 0;
        -:   10:
    #####:   11:    for (i = 0; i < 10; i++)
    #####:   12:      total += i;
        -:   13:
    #####:   14:    if (total != 45)
    #####:   15:                printf ("Failure\n");
        -:   16:            else
    #####:   17:                printf ("Success\n");
        -:   18:
    #####:   19:    num = 0;
    #####:   20:    switch(num) {
    #####:   21:    case 1: printf("1");
    #####:   22:      break;
        -:   23:
    #####:   24:    case 2: printf("2");
    #####:   25:      break;
        -:   26:
    #####:   27:    case 3: printf("3");
    #####:   28:      break;
        -:   29:
    #####:   30:    case 4: printf("4");
    #####:   31:      break;
```

```
-:   32:
#####:   33:    default: printf("0");
#####:   34:      break;
-:   35:
-:   36:   }
#####:   37:   return 0;
-:   38:}
-:   39:
```



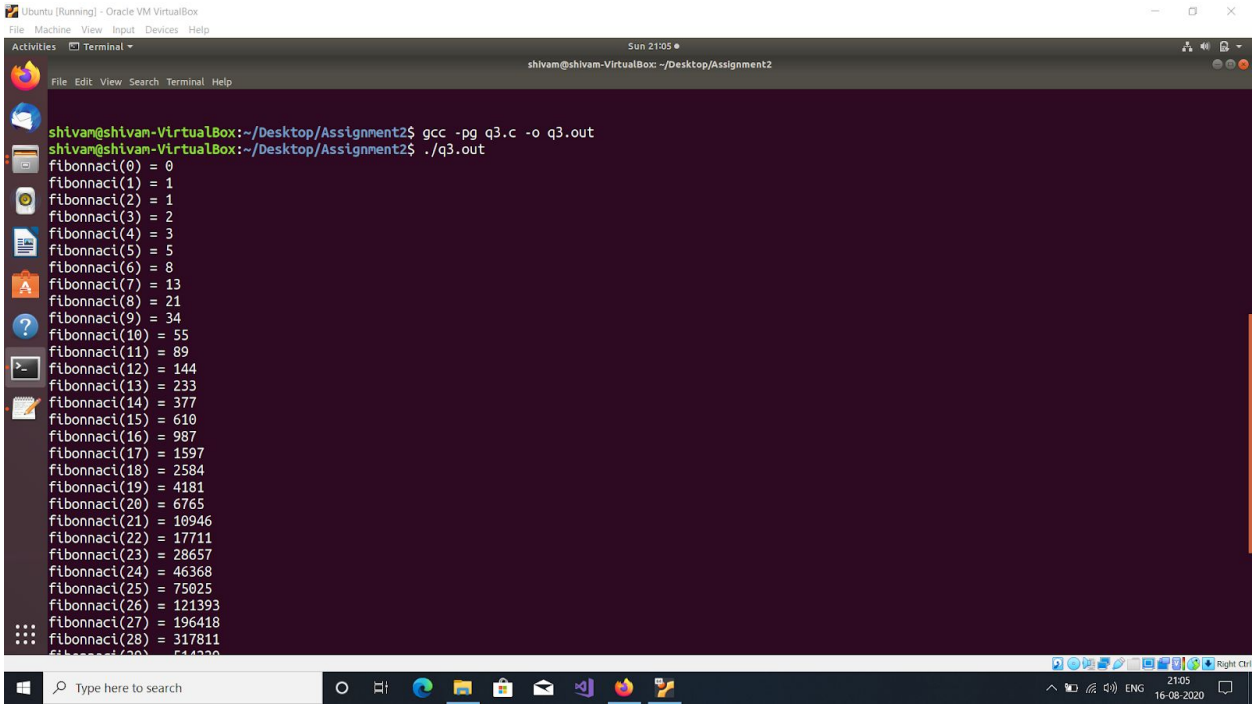
To get the branch and call counts after each block of code.
I have used gcov -b q2.c
For each function, a line is printed showing how many times the function is called, how many times it returns and what percentage of the function's blocks were executed

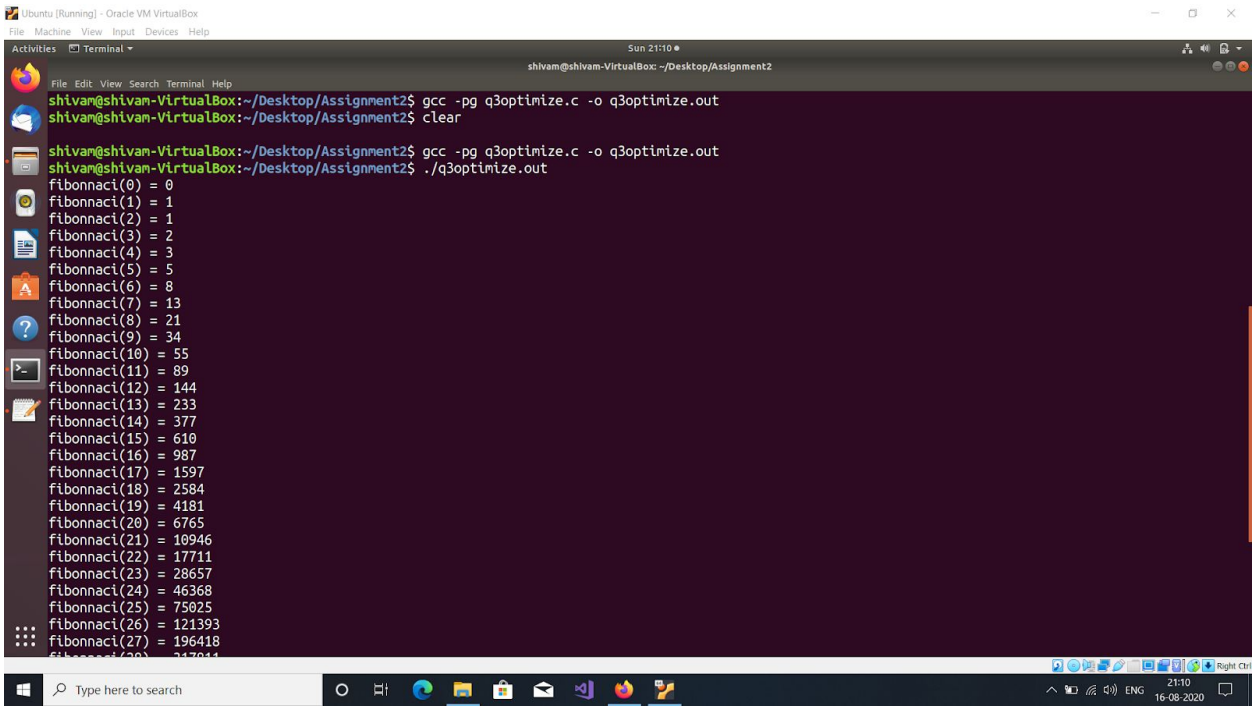
Output:The file *q2.c.gcov* contains output from **gcov**. Here is a sample:

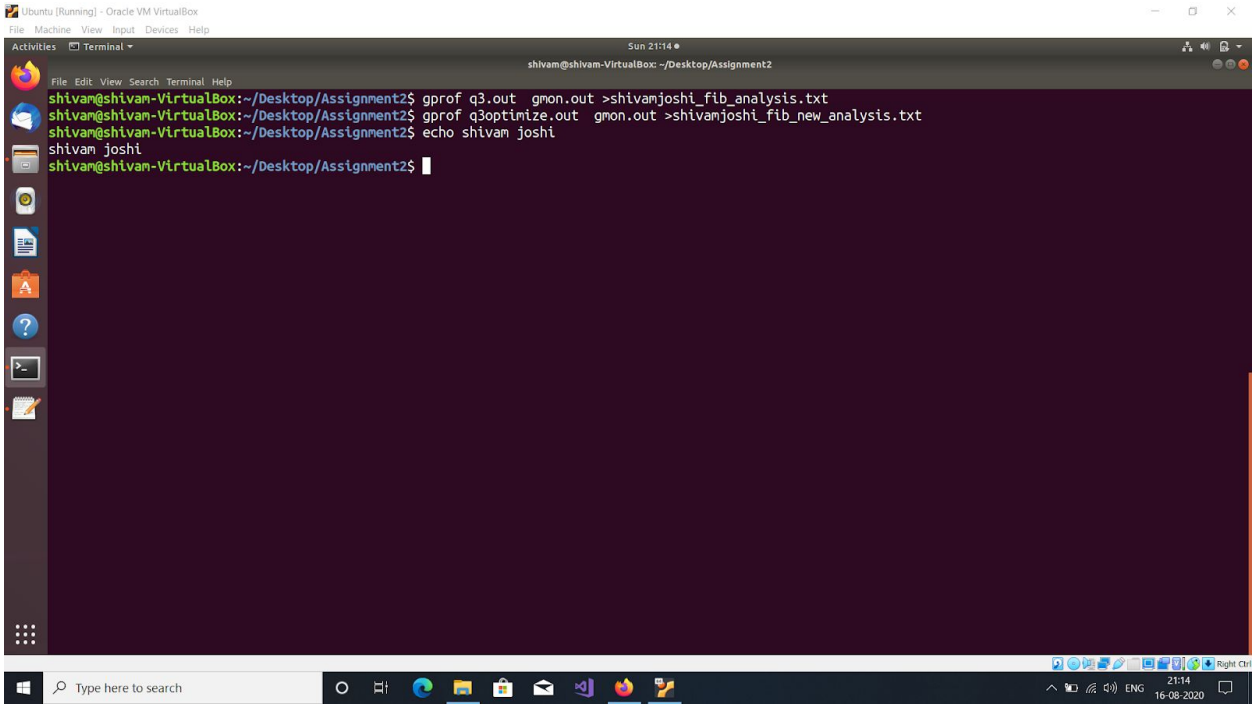
```
        -:    0:Source:q2a.c
        -:    0:Graph:q2a.gcno
        -:    0:Data:-
        -:    0:Runs:0
        -:    0:Programs:0
        -:    1://Shivam Joshi
        -:    2:#include <stdio.h>
        -:    3:
function main called 0 returned 0% blocks executed 0%
    #####:    4:int main (void)
        -:    5:{
        -:    6:
        -:    7:    int i, total, num;
        -:    8:
    #####:    9:    total = 0;
        -:   10:
    #####:   11:    for (i = 0; i < 10; i++)
branch  0 never executed
branch  1 never executed
    #####:   12:      total += i;
        -:   13:
    #####:   14:    if (total != 45)
branch  0 never executed
branch  1 never executed
    #####:   15:                printf ("Failure\n");
call    0 never executed
        -:   16:            else
    #####:   17:                printf ("Success\n");
call    0 never executed
        -:   18:
    #####:   19:    num = 0;
    #####:   20:    switch(num) {
branch  0 never executed
branch  1 never executed
branch  2 never executed
```

```
branch  3 never executed
branch  4 never executed
        #####:   21:    case 1: printf("1");
call    0 never executed
        #####:   22:      break;
        -:   23:
        #####:   24:    case 2: printf("2");
call    0 never executed
        #####:   25:      break;
        -:   26:
        #####:   27:    case 3: printf("3");
call    0 never executed
        #####:   28:      break;
        -:   29:
        #####:   30:    case 4: printf("4");
call    0 never executed
        #####:   31:      break;
        -:   32:
        #####:   33:    default: printf("0");
call    0 never executed
        #####:   34:      break;
        -:   35:
        -:   36:    }
        #####:   37:    return 0;
        -:   38:}
        -:   39:
```

Question:3


Ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Sun 21:05
shivam@shivam-VirtualBox: ~/Desktop/Assignment2
File Edit View Search Terminal Help
shivam@shivam-VirtualBox:~/Desktop/Assignment2$ gcc -pg q3.c -o q3.out
shivam@shivam-VirtualBox:~/Desktop/Assignment2$ ./q3.out
fibonnaci(0) = 0
fibonnaci(1) = 1
fibonnaci(2) = 1
fibonnaci(3) = 2
fibonnaci(4) = 3
fibonnaci(5) = 5
fibonnaci(6) = 8
fibonnaci(7) = 13
fibonnaci(8) = 21
fibonnaci(9) = 34
fibonnaci(10) = 55
fibonnaci(11) = 89
fibonnaci(12) = 144
fibonnaci(13) = 233
fibonnaci(14) = 377
fibonnaci(15) = 610
fibonnaci(16) = 987
fibonnaci(17) = 1597
fibonnaci(18) = 2584
fibonnaci(19) = 4181
fibonnaci(20) = 6765
fibonnaci(21) = 10946
fibonnaci(22) = 17711
fibonnaci(23) = 28657
fibonnaci(24) = 46368
fibonnaci(25) = 75025
fibonnaci(26) = 121393
fibonnaci(27) = 196418
fibonnaci(28) = 317811
Type here to search
ENG
21:05
16-08-2020


Ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Sun 21:10
shivam@shivam-VirtualBox: ~/Desktop/Assignment2
File Edit View Search Terminal Help
shivam@shivam-VirtualBox:~/Desktop/Assignment2$ gcc -pg q3optimize.c -o q3optimize.out
shivam@shivam-VirtualBox:~/Desktop/Assignment2$ clear
shivam@shivam-VirtualBox:~/Desktop/Assignment2$ gcc -pg q3optimize.c -o q3optimize.out
shivam@shivam-VirtualBox:~/Desktop/Assignment2$ ./q3optimize.out
fibonnaci(0) = 0
fibonnaci(1) = 1
fibonnaci(2) = 1
fibonnaci(3) = 2
fibonnaci(4) = 3
fibonnaci(5) = 5
fibonnaci(6) = 8
fibonnaci(7) = 13
fibonnaci(8) = 21
fibonnaci(9) = 34
fibonnaci(10) = 55
fibonnaci(11) = 89
fibonnaci(12) = 144
fibonnaci(13) = 233
fibonnaci(14) = 377
fibonnaci(15) = 610
fibonnaci(16) = 987
fibonnaci(17) = 1597
fibonnaci(18) = 2584
fibonnaci(19) = 4181
fibonnaci(20) = 6765
fibonnaci(21) = 10946
fibonnaci(22) = 17711
fibonnaci(23) = 28657
fibonnaci(24) = 46368
fibonnaci(25) = 75025
fibonnaci(26) = 121393
fibonnaci(27) = 196418
Type here to search
ENG
21:10
16-08-2020



OP of shivamjoshi_fib_analysis.txt

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
96.02	8.49	8.49	43	197.40	197.40	fibonacci
4.55	8.89	0.40				main

% time — the percentage of the total running time of the program used by this function.

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled,

else blank.

total ms/call the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) for 0.11% of 8.89 seconds

```
index % time    self  children    called     name
                                   <spontaneous>
[1]     100.0    0.40    8.49             main [1]
                 8.49    0.00     43/43          fibonacci [2]
-----------------------------------------------
                         2269806252          fibonacci [2]
                 8.49    0.00     43/43          main [1]
[2]      95.5    8.49    0.00     43+2269806252 fibonacci [2]
                         2269806252          fibonacci [2]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

index A unique number given to each element of the table.

Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children. Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the function into this parent.

children This is the amount of time that was propagated from
the function's children into this parent.

called This is the number of times this parent called the
function `/' the total number of times the function
was called. Recursive calls to the function are not
included in the number after the `/'.

name This is the name of the parent. The parent's index
number is printed after it. If the parent is a
member of a cycle, the cycle number is printed between
the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly
from the child into the function.

children This is the amount of time that was propagated from the
child's children to the function.

called This is the number of times the function called
this child `/' the total number of times the child
was called. Recursive calls by the child are not
listed in the number after the `/'.

name This is the name of the child. The child's index
number is printed after it. If the child is a
member of a cycle, the cycle number is printed
between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole. This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.

Index by function name

OP of shivamjoshi_fib_new_analysis.txt (optimization through Dp)
Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	43	0.00	0.00	fibonacci

% time — the percentage of the total running time of the program used by this function.

cumulative seconds — a running sum of the number of seconds accounted for by this function and those listed above it.

self seconds — the number of seconds accounted for by this function alone. This is the major sort for this listing.

calls — the number of times this function was invoked, if this function is profiled, else blank.

self ms/call — the average number of milliseconds spent in this function per call, if this function is profiled, else blank.

total ms/call — the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.

name — the name of the function. This is the minor sort for this listing. The index shows the location of the function in the gprof listing. If the index is in parenthesis it shows where it would appear in the gprof listing if it were to be printed.

Call graph (explanation follows)

granularity: each sample hit covers 2 byte(s) no time propagated

```
index % time    self  children    called     name
                              82                 fibonacci [1]
                0.00    0.00    43/43          main [7]
[1]     0.0     0.00    0.00    43+82          fibonacci [1]
                              82                 fibonacci [1]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:

index A unique number given to each element of the table.
Index numbers are sorted numerically.
The index number is printed next to every function name so
it is easier to look up where the function is in the table.

% time This is the percentage of the `total' time that was spent
in this function and its children. Note that due to
different viewpoints, functions excluded by options, etc,
these numbers will NOT add up to 100%.

self This is the total amount of time spent in this function.

children This is the total amount of time propagated into this
function by its children.

called This is the number of times the function was called.
If the function called itself recursively, the number
only includes non-recursive calls, and is followed by
a `+' and the number of recursive calls.

name The name of the current function. The index number is
printed after it. If the function is a member of a
cycle, the cycle number is printed between the
function's name and the index number.

For the function's parents, the fields have the following meanings:

self This is the amount of time that was propagated directly from the function into this parent.

children This is the amount of time that was propagated from the function's children into this parent.

called This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.

name This is the name of the parent. The parent's index number is printed after it. If the parent is a member of a cycle, the cycle number is printed between the name and the index number.

If the parents of the function cannot be determined, the word `<spontaneous>' is printed in the `name' field, and all the other fields are blank.

For the function's children, the fields have the following meanings:

self This is the amount of time that was propagated directly from the child into the function.

children This is the amount of time that was propagated from the child's children to the function.

called This is the number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.

name This is the name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.

If there are any cycles (circles) in the call graph, there is an entry for the cycle-as-a-whole. This entry shows who called the cycle (as parents) and the members of the cycle (as children.) The `+' recursive calls entry shows the number of function calls that

were internal to the cycle, and the calls entry for each member shows, for that member, how many times it was called from other members of the cycle.

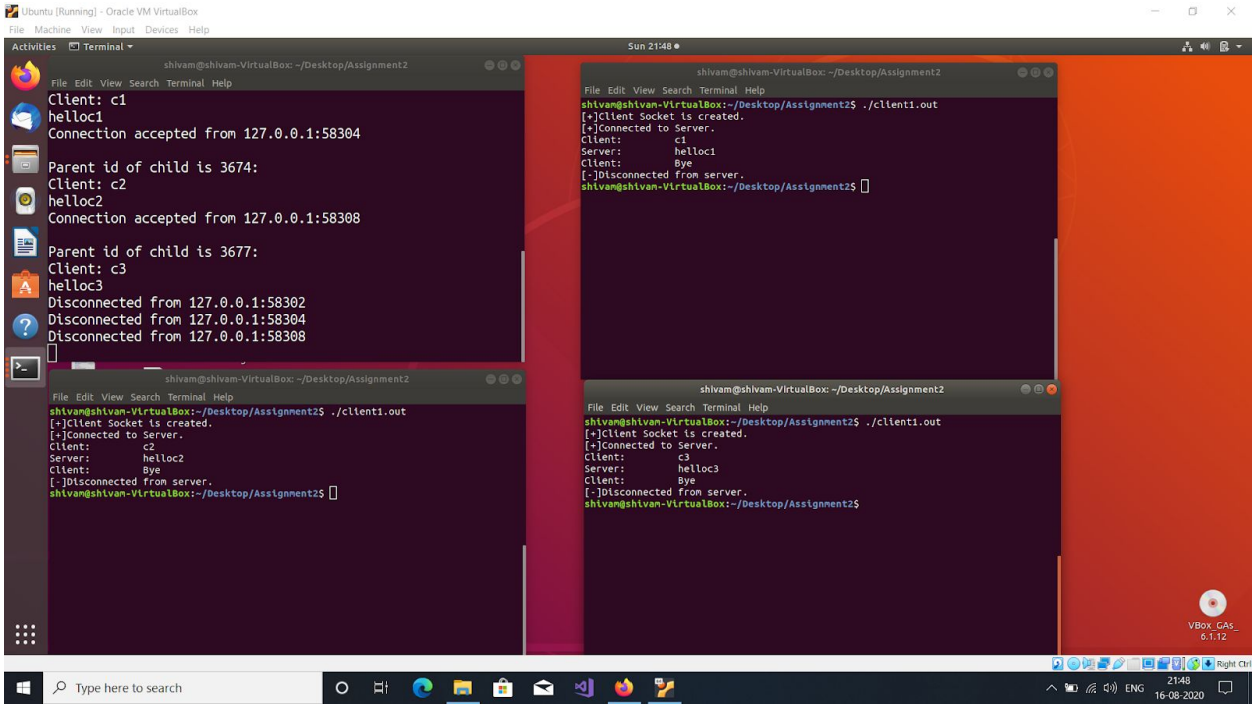
Index by function name

[1] fibonacci

Optimize code:

```
#include <stdio.h>
//shivam joshi
int fibonacci(int n);

int a[42];
int main (int argc, char **argv)
{
  int fib;
  int n;
   for(int i=0;i<=42;i++)
   {
          a[i]=-1;
   }
  for (n = 0; n <= 42; n++) {
         fib = fibonacci(n);
         printf("fibonnaci(%d) = %d\n", n, fib);
  }

  return 0;
}

int fibonacci(int n)
{
  if (n <= 0) {
   a[n]=n;
return a[n];
  }
  else if (n == 1) {
a[n]=n;
return a[n];
  }
if(a[n]!=-1)
{
return a[n];
}
         a[n] = fibonacci(n -1) + fibonacci(n - 2); // no of call will reduce from 2^n to 2n-1 T.C is
O(n)
return a[n];



}
```

Question 4



Client code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define PORT 4444

int main(){

    int clientSocket, ret;
    struct sockaddr_in serverAddr;
    char buffer[1024];

    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if(clientSocket < 0){
        printf("[-]Error in connection.\n");
        exit(1);
    }
    printf("[+]Client Socket is created.\n");

    memset(&serverAddr, '\0', sizeof(serverAddr));
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");

    ret = connect(clientSocket, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
    if(ret < 0){
        printf("Error in connection.\n");
        exit(1);
    }
    printf("[+]Connected to Server.\n");
    //read(clientSocket, buffer, 1024);
        //printf("Server: \t%s\n", buffer);
    //read(clientSocket, buffer, 1024);
        //printf("Server: \t%s\n", buffer);

    while(1){
        printf("Client: \t");
        bzero(buffer, sizeof(buffer));
        scanf("%s", buffer);
        send(clientSocket, buffer, strlen(buffer), 0);

        if(strcmp(buffer, "Bye") == 0){
            close(clientSocket);
            printf("[-]Disconnected from server.\n");
            exit(1);
        }

        if(read(clientSocket, buffer, 1024) < 0){
            printf("Error in receiving data.\n");
        }else{
            printf("Server: \t%s\n", buffer);
        }
    }

    return 0;
}
```

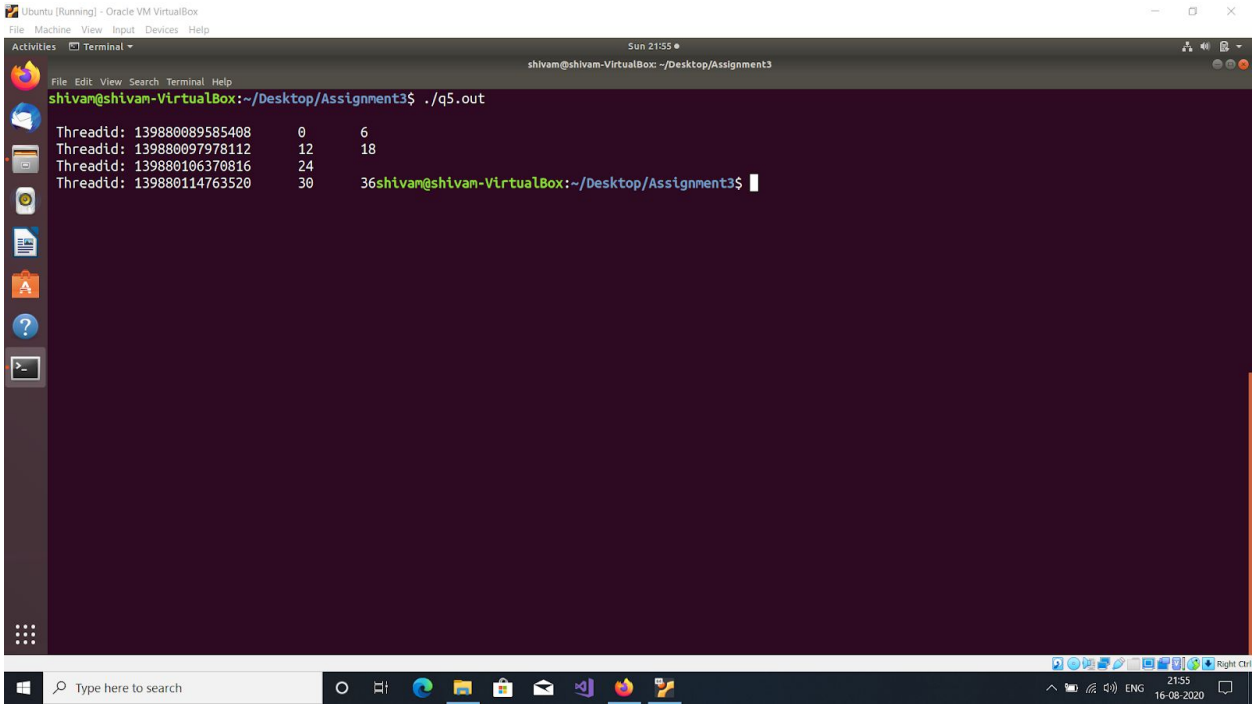
Server Code :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include<sys/wait.h>


#define PORT 4444
int temp,count;
int main(){

   int sockfd, ret;
    struct sockaddr_in serverAddr;

   int newSocket;
   struct sockaddr_in newAddr;

   socklen_t addr_size;

   char buffer[1024];
   char reply[1024];
   pid_t childpid;
         pthread_t thread1;
         //char*str="Hello Welocme to XYZ Bank ";
         //char*str1="Enter your Account number ";

   sockfd = socket(AF_INET, SOCK_STREAM, 0);
   if(sockfd < 0){
         printf("Error in connection.\n");
         exit(1);
   }
   printf("Socket creation: success.\n");

   memset(&serverAddr, '\0', sizeof(serverAddr));
   serverAddr.sin_family = AF_INET;
```

```
    serverAddr.sin_port = htons(PORT);
    serverAddr.sin_addr.s_addr = inet_addr("127.0.0.1");//previously we were using AnnyAddr
                                                        // now i have used inet_addr to give specific
ip

    ret = bind(sockfd, (struct sockaddr*)&serverAddr, sizeof(serverAddr));
    if(ret < 0){
            printf("Error in binding.\n");
            exit(1);
    }
    printf("Binding to %d\n", 4444);

    if(listen(sockfd, 2) == 0){
            printf("Listening..\n");
    }else{
            printf("Error in binding.\n");
    }


    while(1){
            newSocket = accept(sockfd, (struct sockaddr*)&newAddr, &addr_size);
            if(newSocket < 0){
                    exit(1);
            }
            printf("Connection accepted from %s:%d\n", inet_ntoa(newAddr.sin_addr),
ntohs(newAddr.sin_port));
            //send(newSocket, str, strlen(str), 0);
            //send(newSocket, str1, strlen(str1), 0);
            //printf("\nWelcome to xyz Bank ");
            //printf("\nEnter your Account Number :");
            childpid=fork();
            if(childpid  == 0){
                    close(sockfd);
                    count++;
                    if(count==1)
                    {
                            temp=childpid;  // it wont wait since its the first child

                    }
                    else{
                             waitpid(0,0,0);  //after creation of 2nd child this wait  condition will allow

                            //the exec in order of child creation i.e in the order of incoming request.
```

```
                }
                printf("\nParent id of child is %d:",getpid());//print the parent id who created the
child
                while(1){
                        read(newSocket, buffer, 1024);
                        if(strcmp(buffer, "Bye") == 0){
                                printf("Disconnected from %s:%d\n",
inet_ntoa(newAddr.sin_addr), ntohs(newAddr.sin_port));
                                break;
                        }else{

                                printf("\nClient: %s\n", buffer);
                                bzero(reply, sizeof(reply));
                                scanf("%s", reply);
                                //printf("\nserver waiting for 15 secs");
                                sleep(15);
                                send(newSocket, reply, strlen(reply), 0);
                                bzero(buffer, sizeof(buffer));

                        }
                }

        }


   }

   close(newSocket);


   return 0;
}
```

Question 5 :




```
//shivam joshi
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
void *quartercompare(void *arg);
int sum ,count;
int a[20];// global array shared between threads
int b[20];
int i=0,j=5,k=0;
int main(int argc, char **argv)
{
        for(int i=0;i<20;i++)
        {
        a[i]=i*2;      // multiple of2
        }
        for(int i=0;i<20;i++)
        {
        b[i]=i*3;   // multiple of 3
```

```
        }
        pthread_t thread1, thread2,thread3,thread4;
        pthread_create(&thread1, NULL, quartercompare, NULL);
        pthread_create(&thread2, NULL, quartercompare, NULL);
        pthread_create(&thread3, NULL, quartercompare, NULL);
        pthread_create(&thread4, NULL, quartercompare, NULL);

        sleep(5);//This is to ensure  that aa threads are created by putting main function in wait
                // else there will be possibility that none of thread will get executed
         printf("exit:");



}

void *quartercompare(void *arg)       // each thread compare ¼  of array
{
        printf("\n Threadid: %ld",pthread_self()); // prints the thread id whichever thread
                                                        //execute  this function

   while(i<j)
   {
        for(k=0 ;k<20;k++)
        {
        if(a[i]==b[k]){
        printf("\t%d",a[i]);
        k++;
        break;
        }
        }
        i++;

   }

   j+=5;

   pthread_exit(NULL);
}
```