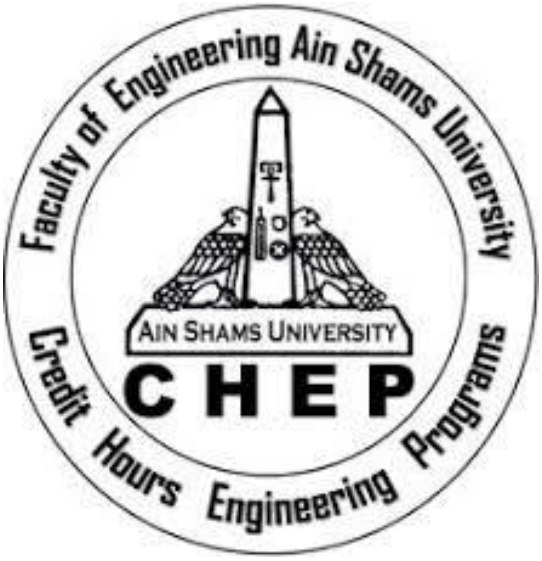


Name: Mohamed Wagih Ahmed

ID: 14p6080

CSE423 Software Performance Evaluation Term Project

In this document I'll explain what my code does and how it works.

Environment: you need Python 3.7.1, Anaconda to run the Jupyter notebook and some other libraries installed like the ones below.

Or you can simply open the .html file to only view the code and check the results without running the code (recommended).

Libraries and Global variables

tabulate library: Prints the table view

matplotlib: Bar chart visualization

networkx and hierarchy: tree graph visualization, hierarchy is an open source piece of code. It is a workaround the networkx library as it doesn't support hierarchal trees directly.

In [1]:
```
from tabulate import tabulate
import re , os
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
from hierarchy import hierarchy_pos
```

In [2]:
```
#Global Variables
call_stack = list()
trace_stack = list()
interrupted_functions = list()
waiting_queue = list()
```

Functions

1. **show_stack_content(…):** Prints the stack's content over time in a tabular form.

```
In [3]: def show_stack_content(tabular_data):
            print(tabulate(tabular_data, headers=['Operation' , 'Current Context' , 'Time'] , tablefmt='fancy_grid'))
```

2. **peek(…):** Returns the top of the stack.

```
In [4]: def peek(stack):
            if(stack == []):
                return None
            else:
                return stack[-1]
```

3. **get_call_stack_trace(…)**: Responsible for showing the stack's content over time and to fill the call context tree nodes values.

```
In [5]: def get_call_stack_trace(operation , functions , time , nodes):
            found_new = False
            found_key = ''
            if(peek(trace_stack) != None):
                index = re.split('call' , str(peek(trace_stack)) , flags=re.IGNORECASE)[1].strip()

                if('Return' in operation and index in interrupted_functions):
                    interrupted_functions.pop()
                    call_stack.pop()

                elif('Call' in operation and 'Call' in str(peek(trace_stack))):
                    interrupted_functions.append(index)

                elif('Return' in operation and index not in interrupted_functions):
                    call_stack.pop()

            if(operation != 'Return'):
                key = re.split('call' , operation , flags=re.IGNORECASE)[1].strip()
                call_stack.append(key)

                if(peek(trace_stack) != None):
                    index = re.split('call' , str(peek(trace_stack)) , flags=re.IGNORECASE)[1].strip()
                    for node in nodes[index]:
                        if(node[0] != key and node[0] != ''):
                            found_new = True
                            found_key = key
                        else:
                            found_new = False
                            node[0] = key
                            node[1] += 1
                            break

                    if(found_new):
                        nodes[index].append([found_key , 1])

            return call_stack
```

4. **read_log_file(…):** Responsible for reading the logs text file content, write the code_flow variable's content and to check if the entire program recorded in the logs file ended correctly or not.

In [6]:
```
def read_log_file(file_path):
    try:
        logs_file = open(file_path , 'r')
        logs_lines = logs_file.read().split('\n')
        tabular_data = list()
        functions = dict()
        nodes = dict()
        code_flow = list()
        number_of_calls = 0
        number_of_returns = 0
        logs_file_corrupted = False


        for line in logs_lines:
            logs = line.split(' - ')

            if(logs[0].lower() != 'return'):
                fn_name = logs[0].split('Call')[1].strip()
                functions[fn_name] = [0,0,0] #call/return time , inclusive time , exclusive time
                nodes[fn_name] = [['',0]]
                number_of_calls += 1
            else:
                number_of_returns += 1

        if(number_of_calls != number_of_returns):
            logs_file_corrupted = True

        for i in range(len(logs_lines)-1):
            logs_current = logs_lines[i].split(' - ')
            logs_next = logs_lines[i+1].split(' - ')
            current_context = get_call_stack_trace(logs_current[0] , functions , int(logs_current[1].strip()) , nodes)
            tabular_data.append([logs_current[0] , '<' + ' , '.join(current_context) + '>' , logs_current[1]])
            if(logs_current[0] == 'Return'):
                code_flow.append(['return'])
            code_flow.append([logs_current[1] , peek(current_context) , logs_next[1]])

            if(i == len(logs_lines)-2):
                code_flow.append(['return'])
                current_context = get_call_stack_trace(logs_next[0] , functions , int(logs_next[1].strip()) , nodes)
                tabular_data.append([logs_next[0] , '<' + ' , '.join(current_context) + '>' , logs_next[1]])


            if(logs_current[0]!= 'Return'):
                trace_stack.append(logs_current[0])
            else:
                trace_stack.pop()

        return tabular_data , functions , nodes , code_flow , logs_file_corrupted

    except IOError:
        print(file_path , 'is not valid!')
```

The code_flow variable content looks like this after reading the logs file:

```
[['0', 'main', '5'], ['5', 'a', '6'], ['6', 'b', '8'], ['return'], ['8', 'a', '9'], ['9', 'b', '10'], ['return'], ['10', 'a',
'12'], ['return'], ['12', 'main', '15'], ['15', 'a', '16'], ['16', 'c', '18'], ['return'], ['18', 'a', '19'], ['return'], ['1
9', 'main', '20'], ['return']]
```

The code_flow variable is any array of arrays of strings.

Each array is an instruction which has a start time, function name/return and a finish time.

5. **calculate_inclusive_and_exclusive_times(…):** Responsible for calculating the inclusive and the exclusive time of each functions, This was the toughest part in the project.
   The trick behind this function is I simulated how each function will work in the OS by adding each interrupted function in waiting_queue and by reading the code_flow variable values I can add which function should be added to the queue for example if the main called function a that mean the main got interrupted by function a as in the code_flow there was no return statement after the main was called initially and so on..

In [7]:
```
def calculate_inclusive_and_exclusive_times(fn_name , code_flow):
    for i in range(len(code_flow)-1):
        item = code_flow[i]

        if(code_flow[i] != ['return']):
            if(item[1] == fn_name and code_flow[i+1] != ['return']):
                if(fn_name not in waiting_queue):
                    functions[fn_name][0] = int(item[0])
                    waiting_queue.append(fn_name)

                functions[fn_name][2] += int(item[2]) - int(item[0])

            #Calculate inclusive and exclusive times for interrupted functions
            elif(item[1] == fn_name and code_flow[i+1] == ['return'] and fn_name in waiting_queue):
                functions[fn_name][1] += int(item[2]) - functions[fn_name][0]
                functions[fn_name][2] += int(item[2]) - int(item[0])
                waiting_queue.pop()
                #functions['a'][0] = int(item[0])

            #Calculate exclusive time for one-shot functions
            elif(item[1] == fn_name and code_flow[i+1] == ['return'] and fn_name not in waiting_queue):
                functions[fn_name][0] = int(item[0])
                functions[fn_name][2] += int(item[2]) - int(item[0])

    return functions
```

6. **draw_bar_chart(…)**: Responsible for the visualization of the results of the inclusive and exclusive times of each function in a bar chart.

In [8]:
```
def draw_bar_chart(functions , code_flow):
    for fn_name in functions:
        functions = calculate_inclusive_and_exclusive_times(fn_name , code_flow)

    functions_names = list()
    inclusive_times = list()
    exclusive_times = list()

    #Data to plot
    n_groups = len(functions)

    for key in functions:
        inclusive_times.append(functions[key][1])
        exclusive_times.append(functions[key][2])

        functions_names.append(key)


    #Create plot
    fig, ax = plt.subplots()
    index = np.arange(n_groups)
    bar_width = 0.35
    opacity = 0.8

    inclusive_bar = plt.barh(index, inclusive_times, bar_width,
    alpha=opacity,
    color='b',
    label='Inclusive Time')

    exclusive_bar = plt.barh(index + bar_width, exclusive_times, bar_width,
    alpha=opacity,
    color='g',
    label='Exclusive Time')

    plt.xlabel('Time')
    plt.ylabel('Functions')
    plt.title('Inclusive vs Exclusive Bar Chart')
    plt.yticks(index + bar_width, functions_names)
    plt.legend()

    plt.tight_layout()
    plt.show()
```

7. **draw_CCT(…):** Responsible for the visualization of the Call Context Tree

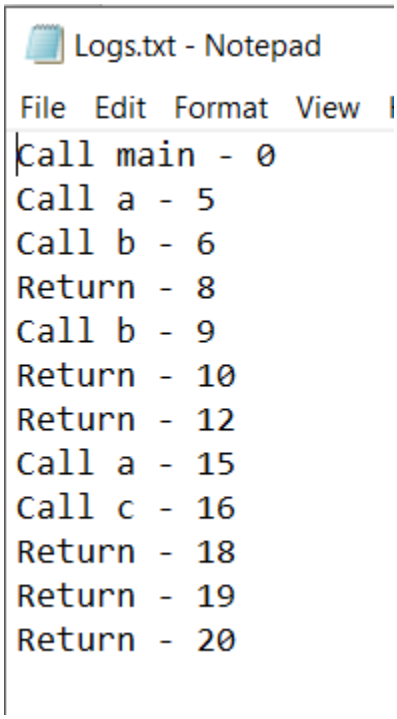
In [9]:
```
def draw_CCT(nodes):
    G = nx.DiGraph()
    e_labels = dict()
    max_weight = 0

    for key,node in nodes.items():
        for subnode in node:
            for i in range(len(subnode)//2):
                if(subnode[0] != ''):
                    if(max_weight < subnode[1]):
                        max_weight = subnode[1]

                    G.add_edge(key , subnode[0] , weight=subnode[1])
                    e_labels[(key , subnode[0])] = subnode[1]


    elarge = [(u, v) for (u, v, d) in G.edges(data=True) if d['weight'] >= max_weight]
    esmall = [(u, v) for (u, v, d) in G.edges(data=True) if d['weight'] < max_weight]

    pos = hierarchy_pos(G , None , 1 , 1)

    #Nodes
    nx.draw_networkx_nodes(G, pos, node_size=500 , node_color='g')

    #Edges
    nx.draw_networkx_edges(G, pos, edgelist=elarge,
                           width=4, alpha=0.8 , edge_color='r')
    nx.draw_networkx_edges(G, pos, edgelist=esmall,
                           width=4, alpha=1.0 , style='dashed')

    #Labels
    nx.draw_networkx_labels(G, pos , font_size=20, font_family='sans-serif')
    nx.draw_networkx_edge_labels(G, pos , edge_labels=e_labels , font_size=15, font_family='sans-serif')

    plt.title('Contex Call Tree')
    plt.axis('off')
    plt.show()
```

Logs File

1. The path variable gets the current working directory and by changing the name of the logs file you can change the file being read.
   Note that the logs file has to be in the same directory of the notebook or the python file.

```
In [10]: path = os.getcwd() + '\Logs.txt'
         tabular_data , functions , nodes , code_flow , logs_file_corrupted = read_log_file(path)
```

2. The Logs file should be a .txt file and the format is like the following:
   FUNCTION_NAME WHITE_SPACE DASH WHITE_SPACE TIME_STAMP



Results

After reading the previous Logs.txt file and making sure that it is valid here are the results:

```
In [11]: if(not logs_file_corrupted):
             show_stack_content(tabular_data)
             draw_bar_chart(functions , code_flow)
             draw_CCT(nodes)
         else:
             print('Logs file is not valid or corrupted!')
```

1. **Call Stack Content over time**

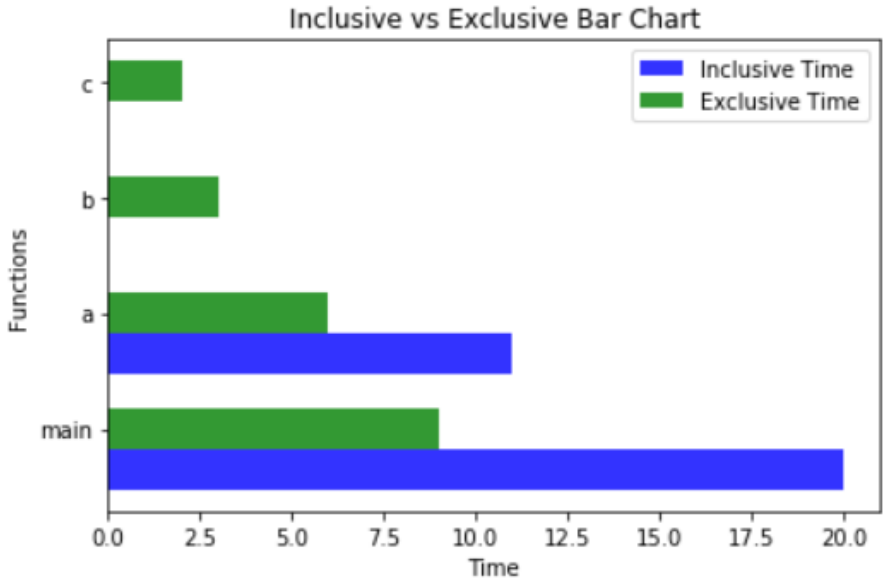
```
show_stack_content(tabular_data)
```

Operation	Current Context	Time
Call main	<main>	0
Call a	<main , a>	5
Call b	<main , a , b>	6
Return	<main , a>	8
Call b	<main , a , b>	9
Return	<main , a>	10
Return	<main>	12
Call a	<main , a>	15
Call c	<main , a , c>	16
Return	<main , a>	18
Return	<main>	19
Return	<>	20

2. **Bar Chart**

   Note that both functions b and c have no inclusive times that's why the inclusive blue bar is not visible

```
draw_bar_chart(functions , code_flow)
```

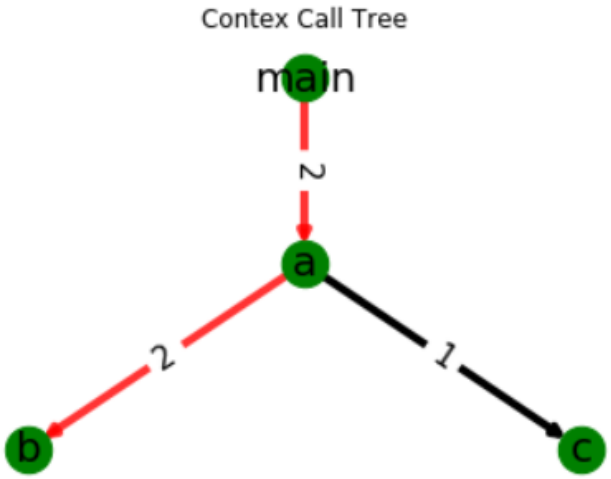


3. **Call Context Tree**

   The red color on the path: main > a > b indicates that this is the hottest path (most frequent)

   The black color indicates a normal (non-hot) path like main > a > c

```
draw_CCT(nodes)
```

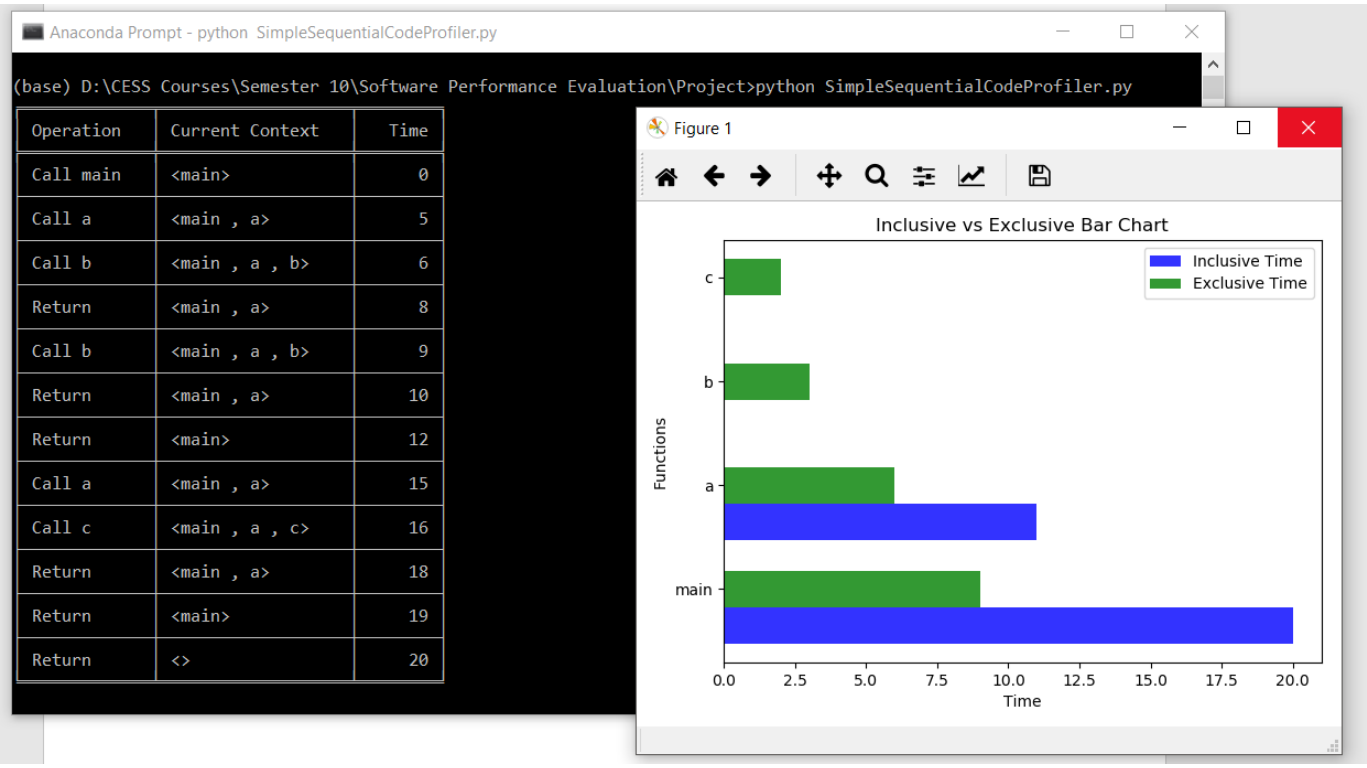


You can also try to run the code on the terminal BUT you won't be able to change the logs file directly from the terminal you've to change it in the code I'll change this later as I don't have much time in my hands now.

Also, make sure that the logs file is in the same directory as the python file and the libraries mentioned above are installed in order to make the code run.

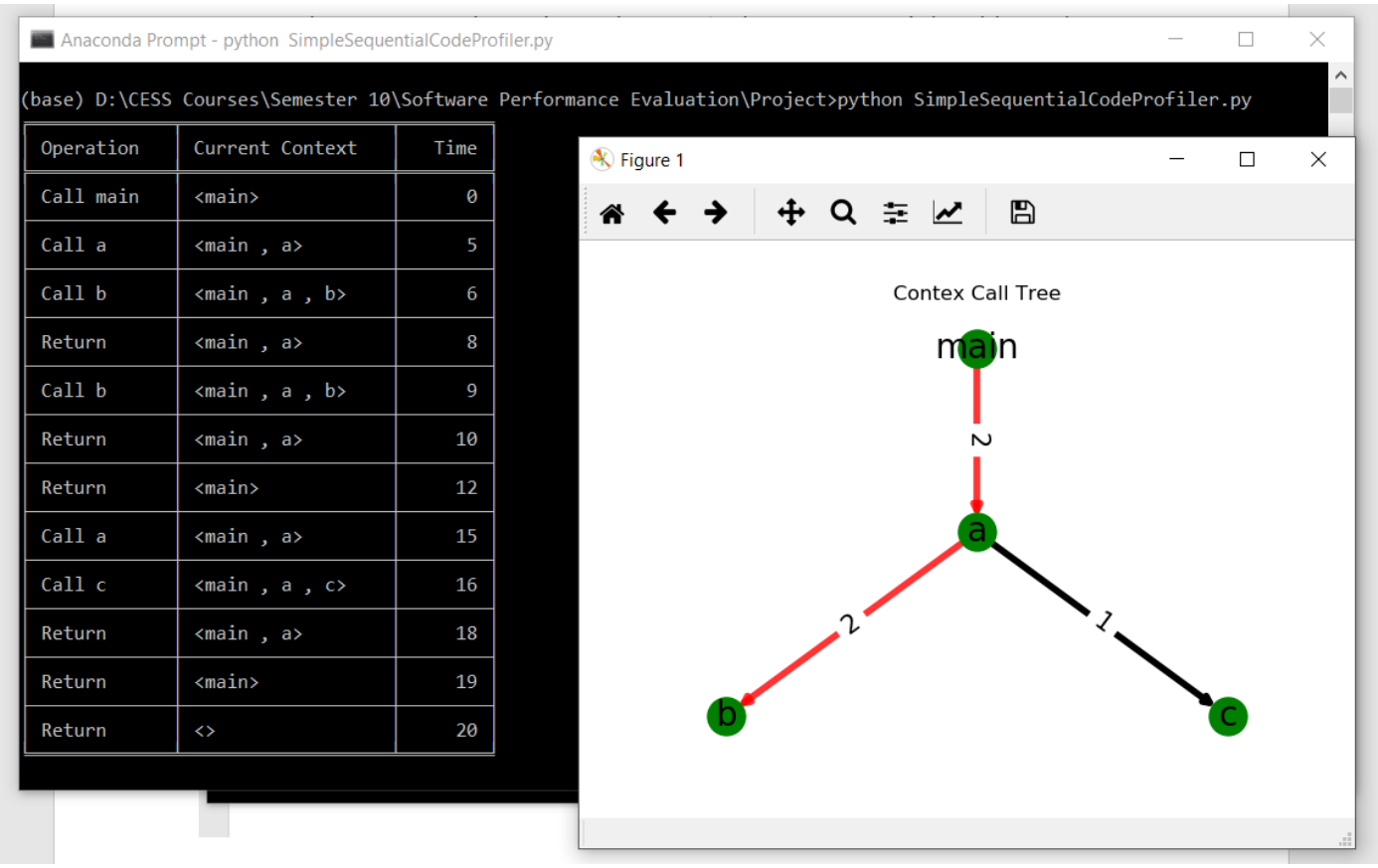
Type in the following command on the terminal in the python file directory folder

```
>python SimpleSequentialCodeProfiler.py
```



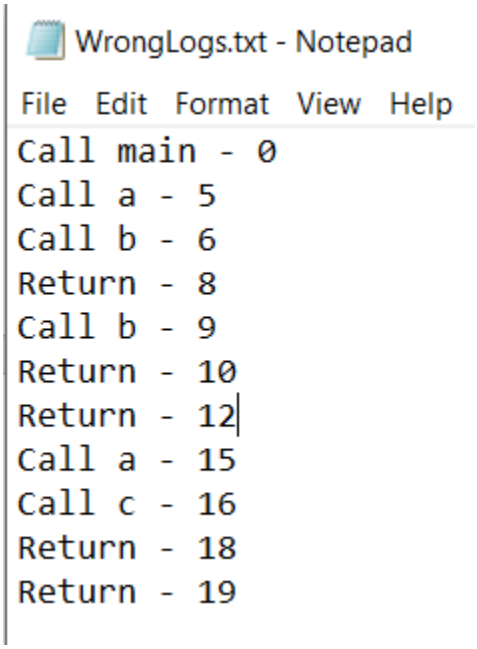
Operation	Current Context	Time
Call main	<main>	0
Call a	<main , a>	5
Call b	<main , a , b>	6
Return	<main , a>	8
Call b	<main , a , b>	9
Return	<main , a>	10
Return	<main>	12
Call a	<main , a>	15
Call c	<main , a , c>	16
Return	<main , a>	18
Return	<main>	19
Return	<>	20

Press the ‘X’ button on the previous Figure1 window to make the next figure appears.



Operation	Current Context	Time
Call main	<main>	0
Call a	<main , a>	5
Call b	<main , a , b>	6
Return	<main , a>	8
Call b	<main , a , b>	9
Return	<main , a>	10
Return	<main>	12
Call a	<main , a>	15
Call c	<main , a , c>	16
Return	<main , a>	18
Return	<main>	19
Return	<>	20

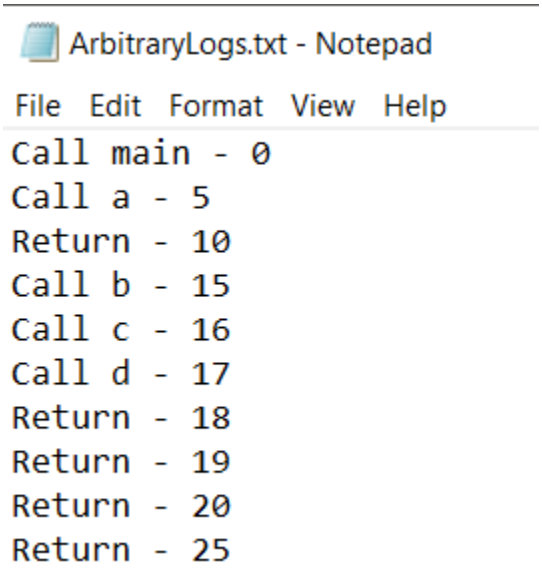
Next, I tried here to read a corrupted logs file named WrongLogs.txt by removing the main's return statement.



And here are the results:

```
In [10]: path = os.getcwd() + '\WrongLogs.txt'
         tabular_data , functions , nodes , code_flow , logs_file_corrupted = read_log_file(path)

In [11]: if(not logs_file_corrupted):
             show_stack_content(tabular_data)
             draw_bar_chart(functions , code_flow)
             draw_CCT(nodes)
         else:
             print('Logs file is not valid or corrupted!')

         Logs file is not valid or corrupted!
```

Finally, I tried another logs file I created to make sure that everything is working correctly.




```
In [10]: path = os.getcwd() + '\ArbitraryLogs.txt'
         tabular_data , functions , nodes , code_flow , logs_file_corrupted = read_log_file(path)
```

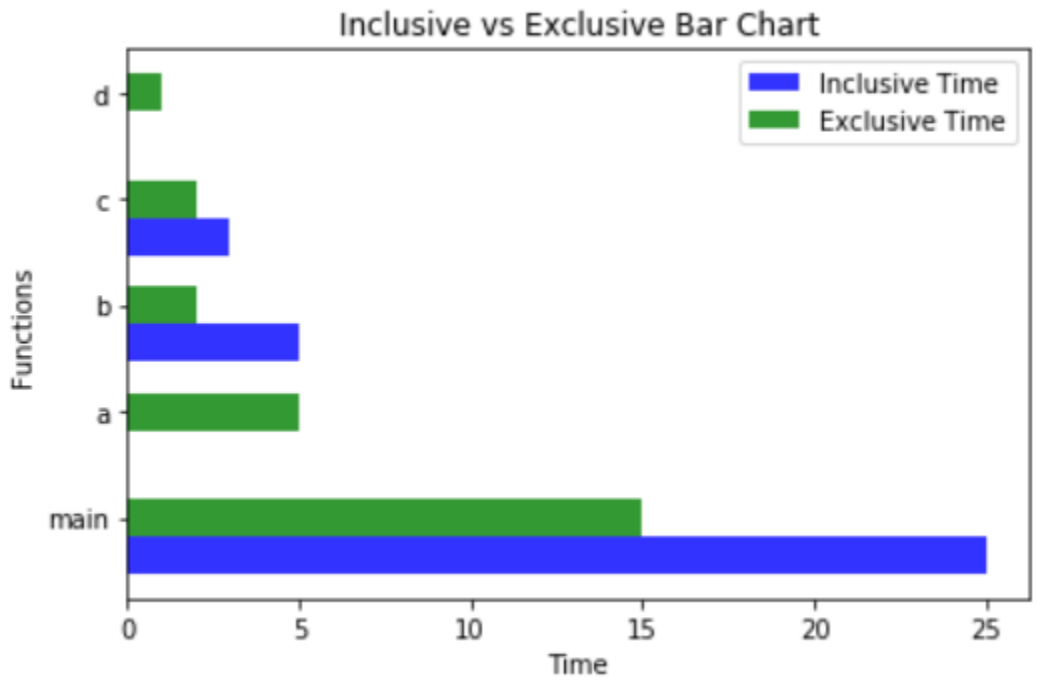
```
In [11]: if(not logs_file_corrupted):
             show_stack_content(tabular_data)
             draw_bar_chart(functions , code_flow)
             draw_CCT(nodes)
         else:
             print('Logs file is not valid or corrupted!')
```

And here are the results:

1. **Call Stack Content over time**

Operation	Current Context	Time
Call main	<main>	0
Call a	<main , a>	5
Return	<main>	10
Call b	<main , b>	15
Call c	<main , b , c>	16
Call d	<main , b , c , d>	17
Return	<main , b , c>	18
Return	<main , b>	19
Return	<main>	20
Return	<>	25

2. **Bar Chart**



3. **Call Context Tree**