Homework #1

Release Time: 2021/03/02 (Mon.)
Due Time: 2021/03/21 (Sun.) 21:59
Contact TAs: `vegetable@csie.ntu.edu.tw`

Instructions and Announcements

- **NO LATE SUBMISSION OR PLAGIARISM IS ALLOWED.**
- Discussions with others are encouraged. However, you should write down your solutions **in your own words**. In addition, for **each and every** problem you have to specify the references (the URL of the web page you consulted or the people you discussed with) on the first page of your solution to that problem.
- Some problems below may not have standard solutions. We will give you the points if your answer is followed by reasonable explanations.

Submission

- Please place your answers in the same order as the problem sheet and do not repeat problem descriptions, just organize them by problem number in a tidy manner.
- Please put your answers in NA part and your the brief report in SA part in one PDF named "`{your_student_id}.pdf`".
- Please zip all the files, including one PDF and three shell scripts, name the zip file "`{your_student_id}.zip`", and submit it through NTU COOL. The zip file should not contain any other files, and the directory layout should be the same as listed below:

```
{your_student_id}/
+-- {your_student_id}.pdf
+-- scripts/
    +-- p1.sh
    +-- p2.sh
    +-- p3.sh
```

Grading

- NA accounts for 100 points plus 2 bonus points while SA accounts for 100 points plus 2 bonus points. The final score is the average between them.
- It's possible you don't get full credits even if you have the correct answer. You should show how you get the answers step by step and list the references.
- Tidiness score: 3 bonus points, graded by TA.
- Final score = $\frac{\text{NA score+SA score}}{2}$ + tidiness score

Network Administration - Wireshark / tcpdump

嚴格禁止 DDoS 我們的服務！違者將會遭受到嚴重的懲罰！

Wireshark 和 tcpdump 都是功能強大的網路封包分析工具，其中 Wireshark 是圖形化介面，而 tcpdump 是終端機介面。在這份作業裡，你需要練習並熟悉這兩個工具。

野生的密碼難道會在網路上赤裸地奔馳著？(18%)

請執行下列步驟：

1. 用 Wireshark 或 tcpdump 開始擷取網路封包
2. 打開瀏覽器，前往 登入介面
3. 帳號輸入什麼都可以，密碼請輸入你的學號 (**不要真的打你的密碼！**)

然後，回答下列問題：

1. 找到那個含有你帳號密碼的封包並附上螢幕截圖 (9%)
2. 重複上列步驟，但這次請前往 登入介面・改。請找到那個含有你帳號密碼的封包並附上螢幕截圖，如果找不到的話請說明原因 (9%)

好玩遊戲也有暗潮洶湧的一面 (42% + 2%(bonus))

雙槍阿楷，那個與傳說中的韭神並列的名字，是一位台大資工系上神擋殺神、佛擋殺佛的傳奇人物。想當初他大一修程式設計時，把批改娘打到苦苦求饒，最後用五個月的排行榜第一名才換得 Judge 的安寧；後來被邀請成為 DSA 與 ADA 助教時，更是所向披靡，每次的 TA hour 都高朋滿座。除了這些必修科目之外，阿楷涉獵的領域還包含作業系統、機器學習、電腦圖學、資訊安全、古典經濟學、自駕車程式設計等，簡直就是全方位天才。

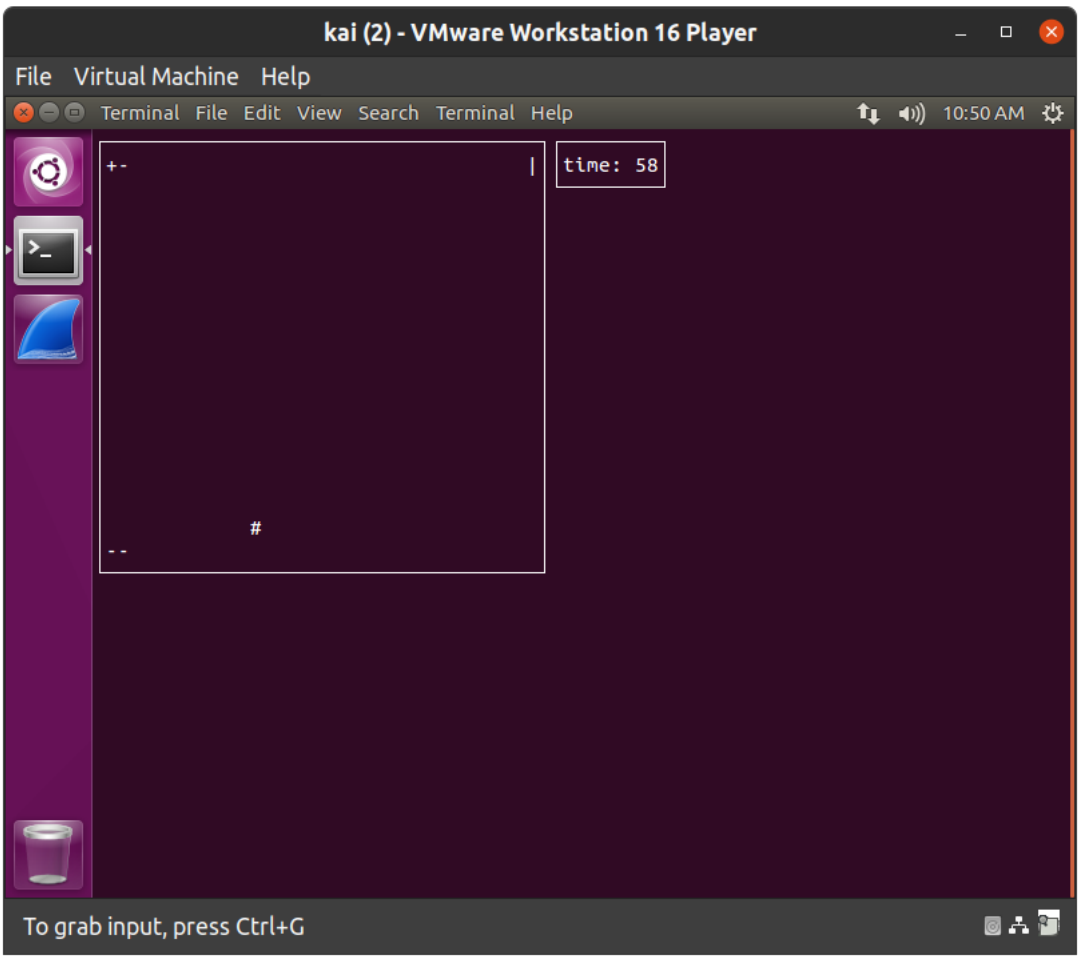
除了那只能用恐怖來形容的學業成就，雙槍阿楷也很喜歡玩遊戲，特別是終端機遊戲。舉凡「車火小 ls」或是「牛語術 - cowsay」，甚至是「星際大戰：終端機」這種大作，都在他的收藏之中。在學習完電腦圖學之後，阿楷還曾想過要在終端機上製作 3D 遊戲，還好後來沒有真的投入時間實做出來，不然阿楷做出來的遊戲，不知道會奪走多少青春學子的美好人生……

某天，雙槍阿楷收到了一個匿名仰慕者送的終端機遊戲。雖然再怎麼喜歡終端機遊戲，但稱霸台大資工兩大資安魔王課程的阿楷非常謹慎，不會輕易執行來路不明的程式。身為阿楷的超級仰慕者，你動用了各種關係來拿到這隻遊戲的程式。你想要幫阿楷鑑定一下這款遊戲有沒有問題，如果沒問題的話，阿楷就可以放心暢玩這個終端機遊戲了！

製作這終端機遊戲的匿名仰慕者知道雙槍阿楷平常都是使用 Linux 作業系統，所以遊戲程式只能在 64-bit Linux 作業系統上執行。下面連結下載虛擬機器，此虛擬機內包含了遊戲的伺服器和遊戲程式。

- 下載連結: double-gun-kai's terminal game server.ova
- sha256sum: `74ce6345d5b51273a3d306f9cad5b910493d8c6222a45d55270dc5313392f30f`
- 帳號: `kai`
- 密碼: `kai`

將虛擬機開機後，打開終端機，依照提示訊息執行 `./client-linux`。如果畫面如下圖，那麼你就成功了。遊戲操作請用上下左右鍵，離開遊戲請按 Ctrl+C



請回答下列問題，並詳細說明你是如何得到答案的：

1. 打開 Wireshark/tcpdump 擷取封包 (請選擇「Loopback: lo」或「lo」網路介面擷取封包)，然後開始玩遊戲。請根據擷取到的封包，說明遊戲程式是怎麼跟伺服器溝通的，還有遊戲程式跟伺服器交換了哪些遊戲資訊 (14%)
2. 這個遊戲其實是個惡意程式，它會把你一個有敏感資訊的檔案傳送給遊戲伺服器。請找出被偷傳出去的是哪個檔案 (14%)
3. 雙槍阿楷最後還是抵擋不住終端機遊戲的魅力，忍不住偷玩了一下這個遊戲。kai.pcapng 是他玩遊戲時的封包紀錄檔（是在他電腦上擷取的封包，不是在你拿到的虛擬機裡擷取的），請找出他電腦上架設的網站的管理員密碼 (14%)
4. (bonus) 雙槍阿楷是個追求速度感的油為青年，修日文課的第一週就跟全班分享「還要再更快」的日文。遊戲有個「極・勁速模式」，請使用 `./client-linux --game-mode fast` 來遊玩。請說明通關技巧並附上通關過後的螢幕截圖，如果沒有通關的話請說明你是如何拿到通關 flag 的 (flag 格式：`HW1{xxx}`) (1%)
5. (bonus) 雙槍阿楷一直夢想著要雙主修，但因為不管哪個科系的課對他來說都太簡單了，所以遲遲無法決定到底要選擇哪個科系雙主修。遊戲有個「魔・雙子模式」，請使用 `./client-linux --game-mode double` 來遊玩。請說明通關技巧並附上通關過後的螢幕截圖，如果沒有通關的話請說明你是如何拿到通關 flag 的 (flag 格式：`HW1{xxx}`) (1%)

這麼多的網路協定要是能全部都認識的話該有多好 (40%)

請用 Wireshark 或 tcpdump 擷取下列的封包。在以下的每題當中，請

- 附上封包的螢幕截圖
- 簡答該協定的用途
- 該協定運作在 TCP/IP 五層網路架構中的哪些層

題目：

1. ICMP echo request & echo reply (8%)
2. DNS query & response (8%)
3. ARP request & reply (8%)
4. DHCP discover & offer & request & acknowledge (16%)

System Administration - Shell Script

Moss Analyzer

Description

Moss (Measure of Software Similarity) is a common free online system to measure programs' similarity, which is also often used to find out program plagiarism. However, not every result warned by Moss turns out to be truly plagiarism, since programs with the same functionality will be somewhat similar. Thus, we might want to define a formula to calculate a score for each similarity result and filter out ones with score greater than or equal to a given threshold. Besides, in addition to the raw analysis result (pairs of similar programs), we might also want to gather some statistical information about them. Hence, in this problem, you are required to write (bash) shell scripts to do this task.
The problem is divided into three parts and for each subtask, you should write a script to accomplish the given requirement. And I have built a fake Moss server which should be almost the same as the real Moss server, except for the server address and the result. You can feel free to test your scripts using this server. The result of this server is generated randomly according to the filenames of the uploaded files; that is, the content of the files will not affect the result, and the same uploaded filenames will produce the same result.
You will get 5 files from here, which contains the correct answer's executables of the three tasks, an executable to upload files to the fake Moss server, which can be passed to **`p1.sh`** as the parameter of **`exec`**, and a text file **help.txt** containing the raw text of the help message in the task 1. You can use these files to test for the correct results (stdout, stderr, exit code) of each case, and you can also perform cross-testing using them. By the way, you can use the program you implemented in the lab to check them.

Agreement

- The filename of the three tasks should be called **`p1.sh`**, **`p2.sh`**, and **`p3.sh`** respectively, and set their permission to 755.
- All of your scripts will be called by `./p[123].sh args...`, thus, you should add a shebang for them to be sure they will be run by bash shell.
- You are forbidden to write any files on the system, including temporaries. Also, you will not have the root privilege.
- All of your scripts should be written in the pure shell script language, that is, you cannot embed any other programming language such as Python, Perl, and (compiled) C++ in it. However, **`sed`** and **`awk`** are allowed.
- Please do not upload to the real Moss server simply for debugging, or you will waste its resources.
- Do not try to reverse the given answer executables, attack the fake Moss server or put malicious codes in your scripts.

Grading rule

Your scripts will be tested automatically with some test cases, and you can get the full credits of each test case if the result is the same as the answer, including stdout, stderr, and the exit status. You can find the grading criteria under each subtask. Thus, please ensure that you have completely followed the requirement. (You can make some simple tests of the correctness yourself using the given answer's

executables.)
For each task, if the grade after auto testing is less than 80%, I will see your script and grade according to its completeness, and you can get at most 80% points. The total score of all the three tasks are 110 points, however, your final score will be clipped to 100 if you get more than 100 points in total. Besides, if all of your three scripts are totally correct, you can get 2 bonus points for reward, that is, the final score will be 102. You are suggested to briefly write down what you have implemented and what you have not in the report so I can understand your script easily and correctly to avoid misgrading.

Task 1 - Argument Parser and Checker (30%)

In this part, you should write a shell script to parse the arguments and perform a simple check of their validity, and output the value of each parameter at last.
The program will be run by: `./p1.sh [options] files...` where the **`files`** are the files to be submitted to the similarity test, and the **`options`** will be described below.

Parameters

- **`cmd`**: Store the command used to call the moss program, empty string by default.
- **`target`**: The target file to be compared with all the other files, empty string by default. If the target file is set, the program will only find out the similarity related to the target file.
- **`comp`**: Boolean value indicating the mode of the required statistical information, 0 by default. If it is set to 0, the program should find out all the files that occur at least once in the similarity result, while if it is set to 1, the program should find out every connected component of the similar files. A connected component here is defined as the maximal set of files such that every two files in it are related through a sequence of similar pairs. You may find out more details (the formal definition in graph theory) in https://en.wikipedia.org/wiki/Component_(graph_theory).
- **`thres`**: Integer value between 0 to 100 indicating the threshold to determine whether a result is plagiarism or not, 0 by default.
- **`url`**: The url of the Moss result to be analyzed, empty string by default. If this parameter is set, it should use this result directly rather than run the moss program to generate a new result.
- **`files`**: A list of filenames whose similarity is going to be analyzed, empty array by default.

Options

- **`-h, --help`**: If this option is present, print the help message in **resources/help.txt** and exit the program with exit status 1. (You should copy the help message into your script instead of reading the file at runtime.)
- **`-e, --exec <cmd>`**: If this option is present, it should read the next argument, set the value of **`cmd`** to it, and continue to parse the next argument.
- **`-f, --file <target>`**: If this option is present, it should read the next argument, set the value of **`target`** to it, and continue to parse the next argument.
- **`-c, --component`**: If this option is present, set the value of **`comp`** to 1, and continue to parse the next argument.

- **`-t, --threshold <thres>`**: If this option is present, it should read the next argument, set the value of **`thres`** to it, and continue to parse the next argument.
- **`-u, --url <url>`**: If this option is present, it should read the next argument, set the value of **`url`** to it, and continue to parse the next argument.
- If the next argument matches a filename, stop parsing the subsequent arguments. All of the arguments after it are regarded as the files to be analyzed and add them into the **`files`** array. Note that you do not need to test whether the subsequent arguments match the filename pattern. The pattern of a filename can be either relative or absolute path, and each directory name and basename without extension can consist of only alphanumeric and '_' characters. Besides, the directory name can also be "." or ".." indicating the current or parent directory. The extension of a filename is optional, which consists of a single '.' and some lowercase alphabet characters. The string that doesn't match this rule is not considered a filename, though it may be a legitimate filename for the system.
- If the next argument matches none of the above (i.e. neither an option nor a filename described above), print a line "Invalid argument <next argument>. Try -h or –help for more help." and exit the program with exit status 1.
- Also, if there is no argument remaining, stop the parsing process.

Argument Check

There are some validity checks on the parsed arguments, please check them one by one.

1. If both **`cmd`** and **`url`** are empty, print "Require -e argument to provide an executable command, or -u argument to provide a result url.\nTry -h or –help for more help.\n" and exit with exit code 1.
2. If **`target`** is provided, that is, non-empty, but the target file is either non-exist or not a readable regular file, print a line "Target file <target> does not exist or is not a readable regular file." and exit with exit code 1.
3. If the value of **`thres`** is not composed of only numeric characters, or the value is not within 0 to 100, print a line "Threshold value should be an integer within 0 to 100." and exit with exit status 1.
4. If the number of elements in **`files`** is 0, print "Require at least one file to be tested similarity.\nTry -h or –help for more help.\n" and exit with exit code 1.
5. Test for every file in **`files`** in order. If any of them is not an existed and readable regular file, print a line "File <filename> does not exist or is not a readable regular file." and exit with exit code 1.

Output

If it passes all of the above parsing and checks, output the value of each variable (may be empty):

```
cmd=<cmd>
target=<target>
comp=<comp>
thres=<thres>
url=<url>
files=(<files separated by single space>)
```

and exit normally at the end.

Grading Spec

- The standard output and the exit code should be exactly the same as the answer's.
- The program should terminate within 1 second (CPU time) and using no more than 64MB memory.

Hints

- How to print escaped string using **`echo`** (or maybe other command)?
- What pattern does the **`case`** command use? Is there any way to extend its usage to match more complex pattern?

Task 2 - Crawler and Filter (40%)

In this part, you are asked to write a shell script to run the moss program, crawl and parse the result website, filter the results and then print them out.
The program will be run by: **`./p2.sh <cmd>$ <target>$ <thres> <url>$ files...`** where the variables are the same as the previous problem. Since an empty string can not be passed as an argument, every possibly empty argument will append a '$' sign at the end when passing as an argument.
If the **`url`** is empty, it will run the moss program by calling **`<cmd> files...`** and get the result url from the last line of its output. However, if the moss program returns non-zero, which means an error occur when running it, print a line "Error when running <cmd>." and exit with exit status 1. Otherwise, if the **`url`** is given, do not run the moss program and use the **`url`** directly. To let us be able to see the result on the website, print the **`url`** in a line into standard error.
Next, you should parse the source code of the url and get every similarity found by Moss. To be more precisely, for each similarity result, you should extract its file1, ratio1, file2, ratio2, and lines, calculate the score using the formula: $\min(\lfloor 0.7 \cdot \frac{\text{ratio1}+\text{ratio2}}{2} + 0.3 \cdot \frac{\text{lines}}{2} \rfloor, 100)$, and then store the tuple (file1, ratio1, file2, ratio2, lines, score) into an array if the score is greater then or equal to the threshold value **`thres`**. Besides, if the **`target`** is provided, the results that are not related to the target file (i.e. both file1 and file2 are not equal to **`target`**) should be discarded.
Besides, to make the output more concise, remove all of the extensions stored in the array since all of them should be the same language, otherwise, the similarity among them is meaningless. Also, if all of the files in **`files`** and **`target`**, if provided, are in the same directory, removing their directories' names and storing their basenames (without extension) will be enough. Note that you should first transform all relative paths into absolute paths and use their absolute paths in the whole p2.sh program.
In addition, to make the output unique, for each entry, if the **`target`** is provided, file1 should be the target file, otherwise (if **`target`** is not provided), ratio1 should be larger than or equal to ratio2, and when they are equal, file1 should be lexicographically less than file2. You may need to swap

(file1,ratio1) and (file2,ratio2) to meet this condition.
After finishing them, you will get an array storing the information of each similar pair of files. If the array is empty, there is no need to do the following analysis, print a line "No plagiarism found." to standard error and then terminate the program with exit code 255.
At last, sort all the results according to their (-score, -ratio1, -ratio2, -lines, file1, file2), that is, sort them using their score from big to small, and for those having the same value of the score, sort them according to their ratio1 value from big to small, and so on. Note that the key score, ratio1, ratio2, and lines are compared numerically and sorted from big to small, while the key file1 and file2 are compared using dictionary order and sorted from small to big.
Finally, for each result, output one line "`<file1> <ratio1> <file2> <ratio2> <lines> <score>`", and exit normally at the end.

Grading Spec

- The standard output and the exit code should be exactly the same as the answer's.
- The standard error should be the same as the answer's except for the the result url if `<url>` is not provided.
- The program should terminate within 15 seconds (CPU time) and using no more than 64MB memory.

Hints

- How to parse the directory name, basename, or extension correctly and easily? And how to transform the relative path to the absolute path easily?
- How can the `curl` command work successfully to get the final result when receiving a redirection response?
- Are floating number arithmetic really needed?
- Clearly understand the usage of `IFS`, the difference between '@' and '*' symbol, and the difference between whether a string is double-quoted or not may be useful when operating on arrays.
- How to sort according to a single field? Have you heard of the difference between stable sort and unstable sort? It may be useful when performing multi-key sorting easily using only single key sorting algorithms.

Task 3 - Analyzer (40%)

In the last part, you are required to write a shell script to analyze the previous result and generate more informative outcomes. As described in task 1, according to the value of `comp`, the program should output different information.
This program will be run by `./p3.sh <target>$ <comp>`, the variable and the reason for the '$' symbol are the same as above. In addition, the output of task 2 will be sent to this program as its input.
If `comp` is set to 0, the program should output a line containing all the files that occur in the input, separated by space. Besides, the output files should be sorted in lexicographical order and each file should not appear more than once. However, if the `target` is set, the target file should be taken out and printed as the first file.
On the other hand, if `comp` is set to 1, and if `target` is non-empty, you should print a line containing

many space-separated files such that the first file is the target file, and the others are the files in the same connected component of the target file, sorted in lexicographical order.
Otherwise, the **comp** is equal to 1 and the **target** is empty, then you should print out every connected component of the input. The components are sorted in the ascending order of the lexicographically smallest file in each connected component. And for each component, print a line "<num>: <files>" where **<num>** is the number of files in this connected component, and **<files>** are the files in this connected component, also sorted in dictionary order and separated by space.
Below will propose a method to solve this task, however, you can also use other methods or algorithms to accomplish this task as long as you can get the same result using bash shell script language. You are highly recommended to study the provided algorithm and implement it by yourself since it is relatively simple, intuitive, and very important. However, I will also provide a pseudo code for it and you can finish them by simply translate it into shell script language without understanding them in detail.

Algorithm

Here provides an algorithm called DFS (Depth-First Search), which can find out every connected component in a graph. A graph is composed of some vertices and some edges where each edge connects two vertices. The main idea of this algorithm is that when visiting a vertex, it will traverse through all its adjacent vertices (the vertices directly connected to it), and if an adjacent vertex hasn't been visited yet, it will recursively visit that vertex and do the same procedure. Intuitively, all the vertices adjacent to the visited vertices will be reached, and as a result, all the vertices directly or indirectly connected to the starting vertex will be reached, which are also all vertices in the same connected component. Besides, the purpose of recording whether a vertex is visited or not is to avoid wasting time walking to the same vertices or infinite recursion.
With the DFS algorithm, you can efficiently find the connected component containing the starting vertex, and you can find all connected components by iterating through each vertex, if a vertex is unvisited, call the DFS function starting from it to get its connected component. Otherwise, if a vertex is visited then nothing to do since its connected component has been found before.
The final problem is, for a vertex, how to find its adjacent vertices easily. Since shell script does not support nested arrays (i.e. multiple dimensional arrays), here provides a data structure called forward star. The concept of the forward star is simple: sort all of the edges in a specific order such that for every vertex, its adjacent edges will lie in a continuous range. In fact, we can simply store all the edges of both directions ("v1 v2" "v2 v1") into an array and then sort them according to the first key. Then, trivially, all the edges start from a specific vertex will be together. Scan over the array and find out the range of each vertex, and store the left and right index of it into a dictionary for further usage.
You can see https://en.wikipedia.org/wiki/Depth-first_search for more detail, or see https://www.cs.usfca.edu/~galles/visualization/DFS.html for its animation.

Sketch

The whole program might be like this:

1. read all the input (file1, ratio1, file2, ratio2, lines, score), and add 'file1 file2' and 'file2 file1' into the edge array.

2. Sort the edge array according to the first element of each entry. For those the first elements are the same, sort them according to the second element.

3. Create two dictionaries **`left`** and **`right`** to store the edge range of each file, scan over the edge list one time to fill them.
4. For each file in dictionary order, print a line "<file>: <adjacent files>" to the standard error where **`<file>`** indicates the current file and **`<adjacent files>`** indicates the files adjacent to it. Note that the **`adjacent files`** should be in dictionary order, and the adjacent files are separated by space. (*)
5. Create a **`visit`** dictionary to store whether a file is visited, then, iterate through files in lexicographical order. For an unvisited file, call the DFS function start from it to find out its connected component.
6. Sort the files in each connected component and print them out according to the format mentioned previously.

- (*): This step is a checkpoint of your task, you can get partial points if its output is correct. However, you can ignore it if you are confident that you can finish the whole program correctly.

And below is the sketch of the DFS function:

1. Add the current file to the current connected component, and mark the file as visited.
2. Iterate through its adjacent files, if the file is not visited, recursively call DFS on it.

Pseudo code

- Forward Star

```
# Assume inputs be the array storing each line of the input
edges=empty array
for each line in inputs:
    extract file1 and file2 from line
    append (file1,file2) to edges
    append (file2,file1) to edges
sort(edges,key=(first element,second element))
set both left,right be an empty dictionary
for(i=0;i<len(edges);):
    j=i+1,cur=edges[i][0]
    while j<len(edges) and edges[i][0]==cur:
        j++
    left[cur]=i;
    right[cur]=j;
    i=j
# for each file, its adjacent edges is in range:
# [left[file],right[file])
# let each edge store only the second element for convenience:
for i=0 to len(edges)-1:
    edges[i]=edges[i][1]
```

- DFS

```
# Assume the forward star have been built
# Assume files array stores all the files occur in the result,
# sorted in dictionary order
visit=empty dictionary
component=empty array of string
size=empty array
function DFS(cur,id) -> int:
    component[id]+=cur+" "
    visit[cur]=id
    l=left[cur],r=right[cur]
    sz=1
    for i=l to r-1:
        nxt=edges[i]
        if nxt not in visit:
            sz+=DFS(nxt,id)
    return sz
id=0
for each file in files:
    if file not in visit:
        append an empty string into component
        append DFS(file,id) into size
        sort(component[id])
        # you may need to remove the trailing space in component[id]
        id++
# Then, component and size array will store all the result needed
# for output, in correct order
```

Grading Spec

- The standard output and should be exactly the same as the answer's.
- The exit code should be 0.
- If you want to get the points of the checkpoint when the program is not totally correct, the standard error should be exactly the same as the answer's.
- The program should terminate within 10 seconds (CPU time) and using no more than 64MB memory.

Hints

- How to read inputs (until EOF) in shell script?
- Be careful about the difference of local variable and global variable when using (recursive) functions. You are highly recommended to do a simple test to know the scope of each kind of variable (arguments, declared by assignment, and declared by **`declare`** command) before start writing the DFS function.
- If you want to finish it using other method, be careful about whether the order of your output meets the requirement. Also, if you use the method provided by the problem, you are encouraged

to think about why the files in each component should be sorted again even if the adjacent files of each file are sorted, and why the order of the components are correct without additional sorting.

Appendix - Combine Them Together

This is not a task of this problem, that is, you do not need to hand in the script of this section.
After finishing the above three tasks, the final thing is to combine them into a complete program. Here I briefly describe how to combine them, you can also modify or add more things to it depending on your need to help you debug or test the scripts, for example, you may print more messages or store some output into a file.

1. Run `p1.sh` and forwards all of the arguments to it.
2. If `p1.sh` returns non-zero, exit the program directly.
3. Evaluate the output of `p1.sh` using the command `eval` to set the value of each variable after argument parsing.
4. Call `p2.sh` as described in task 2.
5. If `p2.sh` returns 255 (which means no plagiarism found), exit normally. Otherwise, if it returns non-zero, print a message showing `p2.sh` failed in runtime, and then exit the program.
6. Run `p3.sh` as described in task 3, and send the output of `p2.sh` as its input (using pipes or here strings).
7. If `p3.sh` returns non-zero, print a message indicating `p3.sh` failed in runtime.

Hints

- `xargs` and `tee` command may be helpful in such kind of program.