Operating Systems
Programming Assignment - 02

Introduction and Learning Objectives

This assignment is a hands-on project that takes you deep into the workings of a core Unix utility: `ls`. Starting with a basic C implementation, you will incrementally add advanced features, mirroring the capabilities of the standard `ls` command in a GNU/Linux environment. This project is designed to strengthen your understanding of file systems, system calls, and the professional practices used to build robust command-line tools.

The goal is to move beyond basic programming and master the design and implementation of system-level software. You will learn to parse command-line options, manage file metadata, format output professionally, and manage the project's evolution with a disciplined Git workflow, including versioned releases for each new feature.

By the end of this assignment, you will have demonstrated proficiency in:

- **System Call Programming:** Using essential system calls like `stat()`, `lstat()`, `readdir()`, `getpwuid()`, and `getgrgid()` to interact with the file system.
- **Data Structures & Algorithms:** Using dynamic arrays and standard library sorting algorithms (`qsort`) to manage and present file system data.
- **Command-Line Argument Parsing:** Implementing logic to handle command-line options such as `-l` and `-R`.
- **Output Formatting:** Crafting detailed, column-aligned, and colorized output similar to professional command-line utilities.
- **Build Automation:** Using a `Makefile` to manage the compilation and build process of your C application.
- **Professional Git Workflow:** Using branches for features, and creating versioned tags and releases on GitHub.

The Scenario: Re-engineering the ls Utility

You will begin with a simple, functional `ls` program and evolve it by implementing a series of advanced features in distinct stages. Each stage will introduce a new capability—such as the long listing format (`-l`), alphabetical sorting, colorized output, and recursive listing (`-R`). A key constraint is to **extend the same base code** throughout the project without making major structural changes.

To prepare for this journey and to assist you in your development, the following resources are provided:

- It is mandatory to first watch the **making of ls video** to grasp the foundational concepts of how the utility is built from the ground up.
- Your starting point is the `ls-v1.0.0` program, which is available for download directly from the **instructor's GitHub account**.
- For implementing complex features like the long-listing format, a repository of **helper code examples** is provided. This repository contains small, focused programs that demonstrate how to:
    - Retrieve file metadata (permissions, link count, size) using the `stat()` system call.

- Resolve user and group IDs into human-readable names using **`getpwuid()`** and **`getgrgid()`**.
- Format file modification timestamps into a standard date and time string.

Do not blindly copy-paste code from AI. You are expected to understand the 'how' and 'why' of every line you submit, and failure to demonstrate this to a TA will result in a zero for the assignment.

Feature-1: Project Setup and Initial Build (05 Marks)

Concepts Covered: Git/GitHub, Project Scaffolding, Basic Makefile Usage

What you will do: Your first task is to set up your personal GitHub repository, populate it with the instructor's starter code, and confirm that you can build the initial version. This establishes the foundational project structure that you will extend in all subsequent features.

Task 1. Create and Clone Your GitHub Repository:

- Create a new **public** repository on GitHub. Name it exactly: **`ROLL_NO-OS-A02`** (e.g., `BSDSF23A001-OS-A02` (ALL UPPERCASE)).
- Initialize this repository with a `README.md` file. This is important as it makes the repository immediately cloneable.
- Use the **`git clone`** command to **clone** this new, empty repository of yours to your local machine. You will now have a local directory named `ROLL_NO-OS-A02`.

Task 2. Populate Your Project with the Starter Code:

- Fork, download or clone the **`ls-v1.0.0`** starter project from the instructor's resources to a temporary location on your computer.
- From the starter project, **copy** the following items directly into your `ROLL_NO-OS-A02` local repository folder:
  - The entire **`src`** directory (which contains `ls-v1.0.0.c`).
  - The **`Makefile`**.

Task 3. Create the Complete Project Structure:

- Navigate inside your **`ROLL_NO-OS-A02`** directory. You have already copied over **`src`** and **`Makefile`**.
- Now, use the standard Linux command for **making new directories** to create the remaining folders needed for the assignment.
- Finally, use the appropriate command to **create an empty file** named **`REPORT.md`**.
- After this step, your **`ROLL_NO-OS-A02`** directory must have the following complete structure:

```
ROLL_NO-OS-A02/
├── src/
│   └── ls-v1.0.0.c
├── bin/           # This directory will hold the compiled program
├── obj/           # This will be used for object files in later features
├── man/           # This will be used for the bonus man page task
├── Makefile       # The build script for compilation
├── README.md      # From when you created the repository
└── REPORT.md      # Your analysis report for answering questions
```

Task 4. Initial Build and Test:

- Before committing, it's crucial to verify that the project works in its new home. Use the **`make`** utility to compile the starter code.
- After a successful build, an executable will be created in the `bin/` directory. Run this executable (e.g., **`./bin/ls`**) to confirm it functions as expected.

Task 5. Commit and Push Your Initial Project:

- Now that your project is fully structured and tested, save this initial state to GitHub.
- Use the standard **`git`** workflow:
  1. **Add** all the new files and directories (`src`, `Makefile`, `bin`, `obj`, `man`, `REPORT.md`) to the staging area.
  2. **Commit** your staged changes with a clear and descriptive message, such as **`"feat: Initial project setup with starter code"`**.
  3. **Push** your commit to the `main` branch on GitHub.

Deliverables:

- A GitHub repository URL that follows the specified naming convention and contains the complete, structured initial project, including the starter code and the **`REPORT.md`** file.

Feature-2: ls-v1.1.0 - Complete Long Listing Format (15 Marks)

Concepts Covered: File Metadata, System Calls (`stat`, `lstat`), User/Group Resolution (`getpwuid`, `getgrgid`), Time Formatting, Command-Line Argument Parsing

What you will do: You will add a feature that allows the user to use the **`-l`** option with **`ls`** to display files inside a directory in a long listing format. The final output of this version should resemble the **`ls -l`** format exactly. This will be version **`1.1.0`** of your utility.

Marks Distribution (15 Marks):

- **Code Implementation (7 Marks):** Correctly uses the specified system calls (`stat`, `getpwuid`, etc.) and logic to gather all required metadata.
- **Output Correctness (3 Marks):** The output is correctly formatted and properly aligned, exactly resembling the real **`ls -l`**.
- **Git Workflow (2 Marks):** A separate branch is used for the feature and work is committed correctly.
- **Tag and Release (3 Marks):** The **`v1.1.0`** tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Create a Development Branch:

- From your `main` branch, create and switch to a new branch for this feature. A suitable name would be `feature-long-listing-v1.1.0`.

Task 2. Implement Argument Parsing:

- Modify the **`main`** function in your C source file to detect if a **`-l`** command-line argument was provided. The standard C library function **`getopt()`** is highly recommended for this.

- Your program's logic should now branch: if `-l` is present, it should call a new function to handle the long listing format; otherwise, it should fall back to the original simple display logic.

Task 3. Implement the Long Listing Logic:

- To implement the long listing format, you will need to create a new C function that gathers and prints detailed information for each file. Your implementation must use the following:
    - `stat()` **or** `lstat()` for retrieving file metadata.
    - `getpwuid()` **and** `getgrgid()` for resolving user and group ID numbers into human-readable names.
    - `ctime()` for formatting the file's modification timestamp.
    - Custom **permission formatting logic** to convert the `st_mode` integer into the familiar `rwx` string format, including handling of special bits.

Task 4. Build, Run, and Commit:

- Update your `Makefile` if necessary to accommodate any new files or changes.
- Build the entire project using `make` and run the resulting executable with the `-l` option to test your implementation thoroughly. The output must be properly aligned in columns.
- Once working, use the standard `git` workflow to add all your modified files to the staging area, and then commit them to your feature branch with a clear message.

Task 5. Tag and Release Version 1.1.0:

- First, merge your completed feature branch back into your `main` branch.
- Use the `git tag` command to create a new **annotated tag**. Name it exactly `v1.1.0`.
- Push your new tag to your remote repository on GitHub.
- On GitHub, navigate to the "Releases" section and draft a new release titled `Version 1.1.0: Complete Long Listing Format`.
- Select the `v1.1.0` tag you just pushed.
- In the "Attach binaries" section, upload your compiled `ls` executable.
- Publish the release.

Report Questions (for REPORT.md):

- What is the crucial difference between the `stat()` and `lstat()` system calls? In the context of the `ls` command, when is it more appropriate to use `lstat()`?
- The `st_mode` field in `struct stat` is an integer that contains both the file type (e.g., regular file, directory) and the permission bits. Explain how you can use bitwise operators (like `&`) and predefined macros (like `S_IFDIR` or `S_IRUSR`) to extract this information.

Feature-3: ls-v1.2.0 – Column Display (Down Then Across) (15 Marks)

Concepts Covered: Output Formatting, Terminal I/O (`ioctl`), Data Structures, Dynamic Memory

What you will do: You will upgrade the default output of `ls` (when used without any options). Instead of a single column, your program will now display files in multiple columns, formatted "down then across," automatically adjusting to the width of the terminal and the length of the filenames. The final output should mimic the default behavior of the standard `ls` utility. This will be version `1.2.0`.

Marks Distribution (15 Marks):

- **Code Implementation (7 Marks):** Logic correctly determines the number of columns and rows based on terminal width and filename lengths.
- **Output Correctness (3 Marks):** The output is correctly formatted in columns, aligned, and adjusts properly to different terminal sizes.
- **Git Workflow (2 Marks):** A separate branch is used for the feature and work is committed correctly.
- **Tag and Release (3 Marks):** The `v1.2.0` tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Merge and Create a New Branch:

- First, ensure your `main` branch is up-to-date by merging your completed long-listing feature branch into it.
- From your `main` branch, create and switch to a new branch for this feature, for example: `feature-column-display-v1.2.0`.

Task 2. Gather All Filenames First:

- The logic for column display requires knowing all filenames and the longest filename *before* you can start printing.
- Modify your program's default behavior:
    - Read all directory entries into a dynamically allocated array of strings.
    - While doing so, keep track of the length of the longest filename.

Task 3. Calculate Column Layout:

- Your program needs to determine how many columns can fit on the screen. The calculation depends on:
    - **Terminal Width:** You can get the terminal width programmatically. The `ioctl` system call with the `TIOCGWINSZ` request is the standard way to do this. If `ioctl` is unavailable, you can fall back to a fixed width (e.g., 80 characters).
    - **Maximum Filename Length:** Use the length of the longest filename you found in the previous step. Add a small buffer for spacing between columns (e.g., 2 spaces).
    - **Calculation:** The number of columns is `terminal_width / (max_filename_length + spacing)`. The number of rows needed can then be calculated based on the total number of files.

Task 4. Implement "Down Then Across" Printing:

- This is the most complex part of the logic. You cannot simply iterate through your array of filenames and print them.
- You must iterate row by row. For each row, you will print the items that belong in that row from each column. For example, to print the first row, you would print `filenames[0]`, then `filenames[0 + num_rows]`, then `filenames[0 + 2*num_rows]`, and so on, for each column.
- Ensure each item is padded with spaces to align with the next column correctly.

Task 5. Build, Run, and Commit:

- Build the project using **`make`** and run your **`ls`** command without any options.
- Resize your terminal window and run it again to ensure the column layout adapts correctly.
- Once the output is correct, commit your changes to your feature branch.

Task 6. Tag and Release Version 1.2.0:

- Merge your completed feature branch back into `main`.
- Create a new **annotated tag** named exactly **`v1.2.0`**.
- Push the tag to GitHub.
- On GitHub, draft a new release titled **`Version 1.2.0: Column Display`**.
- Select the **`v1.2.0`** tag, attach your compiled **`ls`** binary, and publish the release.

Report Questions (for **`REPORT.md`**):

- Explain the general logic for printing items in a "down then across" columnar format. Why is a simple single loop through the list of filenames insufficient for this task?
- What is the purpose of the **`ioctl`** system call in this context? What would be the limitations of your program if you only used a fixed-width fallback (e.g., 80 columns) instead of detecting the terminal size?

Feature-4: ls-v1.3.0 – Horizontal Column Display (-x Option) (15 Marks)

Concepts Covered: Output Formatting Logic, Command-Line Argument Parsing, State Management

What you will do: You will extend your `ls` utility by adding a new display mode: a simple horizontal (row-major) column layout, triggered by the `-x` flag. Unlike the default "down then across" format, this mode lists files from left to right, wrapping to the next line only when the current line is full. This provides a clear contrast in implementation logic and mimics another standard `ls` behavior. This will be version `1.3.0`.

Marks Distribution (15 Marks):

- **Code Implementation (7 Marks):** Argument parsing correctly handles the new `-x` flag, and the horizontal printing logic is implemented correctly.
- **Output Correctness (3 Marks):** The `ls -x` output is correctly formatted horizontally and wraps based on terminal width. The default `ls` output remains "down then across".
- **Git Workflow (2 Marks):** A separate branch is used for the feature and work is committed correctly.
- **Tag and Release (3 Marks):** The `v1.3.0` tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Merge and Create a New Branch:

- First, ensure your `main` branch is up-to-date by merging your completed column-display feature branch (`v1.2.0`) into it.
- From your `main` branch, create and switch to a new branch for this feature, for example: `feature-horizontal-display-v1.3.0`.

Task 2. Extend Argument Parsing for the `-x` Flag:

- Modify your `getopt()` loop in the `main` function to recognize the new `-x` option.
- You will need a way to manage the state of which display mode is selected (e.g., long-listing, horizontal, or default vertical). A simple integer or enum flag variable is a good way to track this. The presence of `-x` should set this flag.

Task 3. Implement the Horizontal Display Logic:

- Create a new C function dedicated to printing in the horizontal format. This logic is significantly different and simpler than the "down then across" method.
- The implementation should:
    1. Determine the terminal width and calculate the width of each column (based on the longest filename, just as in the previous feature).
    2. Loop through your array of filenames.
    3. For each filename, print it, padded with spaces to fill the column width.
    4. Keep track of the current horizontal position on the screen. If printing the next filename would exceed the terminal width, print a newline character (`\n`) and reset your horizontal position counter before printing the filename.

Task 4. Integrate the New Display Mode:

- In your main `do_ls` function (or equivalent), after you have read all the filenames into an array, you must now check your display mode flag.
- Based on the flag, call the correct display function: call your long-listing function if `-l` was present, your new horizontal display function if `-x` was present, or your original "down then across" function if no display option was given.

Task 5. Build, Run, and Commit:

- Build the project using `make`. Test your program thoroughly:
  - `./bin/ls` (should show "down then across" columns)
  - `./bin/ls -l` (should show long listing format)
  - `./bin/ls -x` (should now show horizontal "across" columns)
- Once all modes work correctly, commit your changes to your feature branch.

Task 6. Tag and Release Version 1.3.0:

- Merge your completed feature branch back into `main`.
- Create a new **annotated tag** named exactly `v1.3.0`.
- Push the tag to GitHub.
- On GitHub, draft a new release titled `Version 1.3.0: Horizontal Column Display (-x)`.
- Select the `v1.3.0` tag, attach your compiled `ls` binary, and publish the release.

Report Questions (for `REPORT.md`):

- Compare the implementation complexity of the "down then across" (vertical) printing logic versus the "across" (horizontal) printing logic. Which one requires more pre-calculation and why?
- Describe the strategy you used in your code to manage the different display modes (`-l`, `-x`, and default). How did your program decide which function to call for printing?

Feature-5: ls-v1.4.0 – Alphabetical Sort (15 Marks)

Concepts Covered: Dynamic Memory, Arrays of Pointers, Sorting Algorithms (`qsort`), Function Pointers

What you will do: You will modify the default behavior of `ls` to display directory contents in alphabetical order. This is a standard feature that users expect. The implementation requires reading all directory entries into memory, sorting them, and then displaying the result using your existing column-display logic. This will be version `1.4.0`.

Marks Distribution (15 Marks):

- **Code Implementation (7 Marks):** Correctly reads all entries into a dynamic array and uses `qsort()` with a proper string comparison function.
- **Output Correctness (3 Marks):** The output of the default `ls` command (and `-x` and `-l`) is now verifiably sorted alphabetically.
- **Git Workflow (2 Marks):** A separate branch is used for the feature and work is committed correctly.
- **Tag and Release (3 Marks):** The `v1.4.0` tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Merge and Create a New Branch:

- First, ensure your `main` branch is up-to-date by merging your completed horizontal-display feature branch (`v1.3.0`) into it.
- From your `main` branch, create and switch to a new branch for this feature, for example: `feature-alphabetical-sort-v1.4.0`.

Task 2. Integrate Sorting into the Directory Reading Logic:

- Your logic from previous features already reads all filenames into a dynamic array. Now, you will add the sorting step immediately after reading is complete, but before any display function is called.
- Use the standard C library function `qsort()` to sort the array of filenames.
- To use `qsort()`, you must provide it with a **comparison function**. You will need to write a simple helper function that takes two `const void *` pointers, casts them to string pointers (`char **`), and uses `strcmp()` to determine their alphabetical order. This comparison function is the key to telling `qsort` how to sort your specific data type (strings).

Task 3. Ensure Sorting Applies to All Display Modes:

- After the array of filenames has been sorted by `qsort()`, your existing logic will pass this sorted array to the appropriate display function (long-listing, vertical column, or horizontal column). No changes should be needed for the display functions themselves; they will now simply render the pre-sorted list.
- Ensure you properly manage memory by freeing the array of strings and the array itself after they have been displayed.

Task 4. (Optional) Handle Hidden Files:

- Your base `ls` code currently skips hidden files (those starting with `.`). The real `ls` command only shows them when the `-a` flag is used. While implementing `-a` is not required for this

feature, ensure your sorting logic would work correctly even if hidden files were included in the list.

Task 5. Build, Run, and Commit:

- Build the project using `make` and run your `ls` command with no options, `-x`, and `-l` on various directories to verify that the output is always alphabetically sorted.
- Once the output is correct, commit your changes to your feature branch with a clear message.

Task 6. Tag and Release Version 1.4.0:

- Merge your completed feature branch back into `main`.
- Create a new **annotated tag** named exactly `v1.4.0`.
- Push the tag to GitHub.
- On GitHub, draft a new release titled `Version 1.4.0: Alphabetical Sort`.
- Select the `v1.4.0` tag, attach your compiled `ls` binary, and publish the release.

Report Questions (for `REPORT.md`):

- Why is it necessary to read all directory entries into memory before you can sort them? What are the potential drawbacks of this approach for directories containing millions of files?
- Explain the purpose and signature of the comparison function required by `qsort()`. How does it work, and why must it take `const void *` arguments?

Feature-6: ls-v1.5.0 – Colorized Output Based on File Type (10 Marks)

Concepts Covered: File Metadata (`stat`), ANSI Escape Codes, String Manipulation

What you will do: You will enhance `ls` to provide a more user-friendly, colorized output. Based on the file type, each filename will be printed in a specific color or style, making it easier to distinguish between directories, executables, and other file types at a glance. This will be version `1.5.0`.

Marks Distribution (10 Marks):

- **Code Implementation (5 Marks):** Logic correctly identifies file types using `stat` and applies the correct ANSI escape codes for coloring.
- **Output Correctness (2 Marks):** The output of `ls` correctly displays colors for different file types as specified.
- **Git Workflow (1 Marks):** A separate branch is used for the feature and work is committed correctly.
- **Tag and Release (2 Marks):** The `v1.5.0` tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Merge and Create a New Branch:

- First, ensure your `main` branch is up-to-date by merging your completed alphabetical-sort feature branch (`v1.4.0`) into it.
- From your `main` branch, create and switch to a new branch for this feature, for example: `feature-colorized-output-v1.5.0`.

Task 2. Understand ANSI Escape Codes:

- Color in the Linux terminal is achieved by printing special character sequences known as **ANSI escape codes**. For example, to print text in blue, you would print `"\033[0;34m"`, then your text, and then `"\033[0m"` to reset the color back to default.
- Create a set of `#define` macros or a helper function in your code to manage these color codes, making your printing logic cleaner.

Task 3. Determine File Type Before Printing:

- To know which color to use, you must determine the type of each file before you print its name. This requires using the `stat()` or `lstat()` system call to get the `st_mode` for each file.
- You will need to integrate this `stat()` call into your display loops. Before printing any filename, get its `st_mode`.

Task 4. Implement the Coloring Logic:

- Write a function that takes a filename and its `st_mode` and prints it with the correct color based on the following rules:
  - **Directory:** Blue
  - **Executable:** Green (Hint: Check the execute permission bits in `st_mode`).
  - **Tarballs** (`.tar`, `.gz`, `.zip`): Red (Hint: Use `strstr` or check the filename extension).
  - **Symbolic Links:** Pink (Hint: Use `lstat` and the `S_ISLNK()` macro).
  - **Special Files** (e.g., device files, sockets): Reverse Video

Task 5. Build, Run, and Commit:

- Build the project using `make` and run your `ls` command on a directory that contains a mix of file types (directories, executables, archives, etc.) to verify that the coloring works as specified.
- Once the output is correct, commit your changes to your feature branch.

Task 6. Tag and Release Version 1.5.0:

- Merge your completed feature branch back into `main`.
- Create a new **annotated tag** named exactly `v1.5.0`.
- Push the tag to GitHub.
- On GitHub, draft a new release titled `Version 1.5.0: Colorized Output`.
- Select the `v1.5.0` tag, attach your compiled `ls` binary, and publish the release.

Report Questions (for `REPORT.md`):

- How do ANSI escape codes work to produce color in a standard Linux terminal? Show the specific code sequence for printing text in green.
- To color an executable file, you need to check its permission bits. Explain which bits in the `st_mode` field you need to check to determine if a file is executable by the owner, group, or others.

Feature-7: ls-v1.6.0 – Recursive Listing (-R Option) (20 Marks)

Concepts Covered: Recursion, String Manipulation (Path Construction), File Metadata (`stat`), Command-Line Argument Parsing

What you will do: You will implement one of the most powerful features of `ls`: recursive listing, triggered by the `-R` flag. When this option is used, your program will list the contents of the initial directory, and then for every subdirectory it finds, it will recursively descend into that subdirectory and list its contents as well, continuing until all nested directories have been visited. This will be version `1.6.0`.

Marks Distribution (20 Marks):

- **Code Implementation (10 Marks):** The recursive logic is correctly implemented, and path construction is handled properly for nested directories.
- **Output Correctness (5 Marks):** The output correctly mimics the structure of the standard `ls -R` command, printing directory headers.
- **Git Workflow (2 Marks):** A separate branch is used for the feature and work is committed correctly.
- **Tag and Release (3 Marks):** The `v1.6.0` tag and corresponding GitHub Release with the binary asset are created successfully.

Task 1. Merge and Create a New Branch:

- First, ensure your `main` branch is up-to-date by merging your completed colorized-output feature branch (`v1.5.0`) into it.
- From your `main` branch, create and switch to a new branch for this feature, for example: `feature-recursive-listing-v1.6.0`.

Task 2. Extend Argument Parsing for the -R Flag:

- Modify your `getopt()` loop in the `main` function to recognize the new `-R` option.
- Use a flag variable (e.g., `int recursive_flag = 0;`) to keep track of whether this option has been enabled by the user.

Task 3. Modify the Core do_ls Function for Recursion:

- The core of this feature is to make your main directory-listing function recursive. It should now perform the following steps:
    - First, print the name of the directory it is about to list (e.g., `printf("%s:\n", dirname);`). This is standard `ls -R` behavior.
    - Read all directory entries into a dynamically allocated array and sort them (as you did in previous features).
    - Loop through the sorted list of entries to display them using your existing display logic (e.g., column, long-listing, etc.).
    - **Crucially**, inside this same loop, after printing an entry's name, you must check if that entry is a directory.
        - To do this, you must first construct the **full path** to the entry (e.g., by combining the current directory's path and the entry's name, like `dirname/entryname`).

    - Use `lstat()` on this full path to get the file's metadata.
    - Use the `S_ISDIR()` macro on the `st_mode` field to check if it's a directory.

- If the entry is a directory (and its name is not `.` or `..`), you must make a **recursive call** to your `do_ls` function, passing the newly constructed full path as the argument.

Task 4. Build, Run, and Commit:

- Build the project using `make`.
- Test your implementation by running it on a directory that contains nested subdirectories (e.g., `./bin/ls -R src`). Verify that it correctly lists the top-level directory and then recursively lists the contents of the subdirectories.
- Once the output is correct, commit your changes to your feature branch.

Task 5. Tag and Release Version 1.6.0:

- Merge your completed feature branch back into `main`.
- Create a new **annotated tag** named exactly `v1.6.0`.
- Push the tag to GitHub.
- On GitHub, draft a new release titled `Version 1.6.0: Recursive Listing (-R)`.
- Select the `v1.6.0` tag, attach your compiled `ls` binary, and publish the release.

Report Questions (for `REPORT.md`):

- In a recursive function, what is a "base case"? In the context of your recursive `ls`, what is the base case that stops the recursion from continuing forever?
- Explain why it is essential to construct a full path (e.g., "parent_dir/subdir") before making a recursive call. What would happen if you simply called `do_ls("subdir")` from within the `do_ls("parent_dir")` function call?

Viva-Voce and Final Submission (5 Marks)

What you will do: This final part covers the steps to merge all your work, ensure your report is complete, and submit the project for grading. The submission will be followed by a viva-voce where you will be expected to defend and explain your work.

Marks Distribution (10 Marks):

- **Final Report (5 Marks):** All questions in `REPORT.md` are answered completely, clearly, and thoughtfully.
- **Final Git Workflow (5 Marks):** The final feature branch is merged correctly, and all branches are pushed to GitHub.

Task 1. Finalize your REPORT.md file

- Ensure all questions from all preceding features of the assignment are answered. Review your answers to make sure they reflect a clear understanding of the concepts you've implemented.

Task 2. Merge Your Final Branch

- Merge your final completed feature branch (e.g., `feature-recursive-listing-v1.6.0` or whichever is your last one) back into your `main` branch. This ensures that **`main`** contains the complete and final version of your project.

Task 3. Push Everything to GitHub

- For grading, your instructor needs to see not only the final code but also the history of your development process contained in your feature branches.
- Use the appropriate **`git`** command to push your updated `main` branch to GitHub.
- Also, use the **`git push`** command to push **all** of your feature branches to your remote repository. This includes branches like `feature-long-listing-v1.1.0`, `feature-column-display-v1.2.0`, `feature-alphabetical-sort-v1.4.0`, etc.

Grading Rubric (Total: 100 Marks)

Part	Task	Marks
Part 1	Project Setup and Initial Build	5
Part 2	**ls-v1.1.0** – Complete Long Listing Format	15
Part 3	**ls-v1.2.0** – Column Display (Down Then Across)	15
Part 4	**ls-v1.3.0** – Horizontal Column Display (-x)	15
Part 5	**ls-v1.4.0** – Alphabetical Sort	15
Part 6	**ls-v1.5.0** – Colorized Output	10
Part 7	**ls-v1.6.0** – Recursive Listing (-R)	20
Part 8	Final Submission (Report, Final Git Workflow) & Viva	5
Total		**100**

Happy Learning with Arif Butt