**SOFTWARE SUPPORT EXERCISE**

**Problem:** "You are given the following bash script. This script is *intended* to find near-duplicate files in a directory based on file size and name (ignoring extensions). However, it's not very efficient, and it has some potential issues. Analyze the script, explain what it does, identify any problems, and propose an optimized version."

**Provided Script (Example - you can modify this):**

Bash

```
Unset
#!/bin/bash

dir="/path/to/directory"   # Directory to check

find "$dir" -type f -print0 | while IFS= read -r -d $'\0' file; do
    size=$(stat -c '%s' "$file") name=$(basename "$file")
    name="${name%.*}" # Remove extension

    find "$dir" -type f -size "$size" -print0 | while IFS= read -r
-d $'\0' other_file; do other_name=$(basename
        "$other_file")
        other_name="${other_name%.*}" # Remove extension

        if [ "$file" != "$other_file" ] && [ "$name" == "$other_name"
]; then
            echo "Possible duplicate found: $file and $other_file" # rm "$file"   #
            Comment this out initially!
            break   # Exit inner loop fi
    done
done
```

**Task:**

1. **Explain:** Describe what the script does step-by-step.

This script identifies potential near-duplicate files within a specified directory by comparing files based on their size and filename (excluding file extensions). Here's a breakdown of how it works:

**1. Set Directory:**
The script defines the target directory using a variable (e.g., dir="/path/to/directory"), which specifies where to search for files.

**2. Outer Loop – File Discovery:**
The command find "$dir" -type f -print0 lists all regular files within the directory, using null-terminated output (-print0) to safely handle filenames with spaces or special characters.
Each file is then read into the variable file via a while loop.

**3. Extract File Information:**
For each file:

- **Size:** Retrieved using stat -c '%s' "$file", which returns the file size in bytes.
- **Name (without extension):** Extracted with basename "$file" followed by ${name%.*} to strip the file extension.

**4. Inner Loop – Compare Files:**
The script then runs find "$dir" -type f -size "$size" to locate all files of the same size.
Each matching file (other_file) is checked as follows:

- Its name without extension is obtained as before.
- Two conditions are evaluated:
    - The files are not the same ("$file" != "$other_file").
    - The base names (without extensions) match ("$name" == "$other_name").

If both are true, the script outputs a message indicating a possible duplicate:

echo "Possible duplicate found: $file and $other_file"

It then breaks out of the inner loop after identifying the first match.

**5. Optional Deletion (Commented Out):**
A commented line (# rm "$file") indicates that duplicates could be deleted, but this action is disabled to prevent accidental data loss.

**Summary:**
The script scans for files with identical sizes and matching names (ignoring extensions), and flags them as potential duplicates.

==========
==========

2. **Identify Problems:** What are the inefficiencies or potential issues?

**1. Inefficient Use of Nested find Commands:**
The script performs a full directory search for each file using a nested find, resulting in $O(N^2)$ complexity. This drastically slows down execution on directories with many files.

**2. Redundant File Comparisons:**
Each file pair is compared twice (e.g., file1 vs file2 and file2 vs file1), leading to unnecessary processing. Additionally, it doesn't track which files have already been checked, increasing redundancy.

**3. Incorrect Use of -size Option:**
The find -size "$size" command expects sizes in 512-byte blocks by default, or specific units (e.g., -size 1024c for bytes). Using raw numbers may lead to incorrect or inconsistent behavior across different systems.

**4. Limited Duplicate Detection:**
The script exits the inner loop after finding the first match, potentially missing other duplicates with the same name and size.
Also, comparing only by name and size is unreliable—files may share these attributes but differ in content (e.g., photo.jpg vs photo.png).

**5. Lack of Error Handling:**
There are no checks to ensure the target directory exists or is accessible.
Errors from commands like stat or find (e.g., permission issues) are not handled, which could cause the script to fail silently or behave unpredictably.

**6. Hardcoded Directory Path:**
The script uses a fixed path (/path/to/directory), reducing flexibility. It's better to allow the user to specify the directory via a command-line argument.

**7. Portability Limitations:**
The use of stat -c '%s' assumes GNU stat. This syntax is incompatible with systems like macOS, where stat uses a different format, making the script non-portable.

**8. Safety Risks with Deletion:**
The commented-out rm "$file" command poses a risk. If enabled, it could delete files without verifying their contents or prompting the user, leading to potential data loss.

==========
==========

3. **Optimize:** Rewrite the script to be more efficient and robust.

```
#!/bin/bash

dir="/path/to/directory"
declare -A file_map

# Step 1: Build a map with key = name_without_ext + size, value = full path
while IFS= read -r -d $'\0' file; do
  size=$(stat -c '%s' "$file")
  name=$(basename "$file")
  name="${name%.*}"
  key="${name}_${size}"

  if [[ -n "${file_map[$key]}" ]]; then
    echo "Possible duplicate found:"
    echo "  ${file_map[$key]}"
    echo "  $file"
    echo ""
  else
    file_map["$key"]="$file"
  fi
done < <(find "$dir" -type f -print0)
```

**Deliverables:**

- Corrected/improved code.
- A clear explanation of the changes made.

---

- **Single-pass search**: We scan the directory only once and create a table using "size|filename" as the key.
- **Efficient storage**: We use a table (file_map) to store all files with the same key for quick access.
- **No repeated work**: Each key is processed once, avoiding duplicate comparisons.
- **Scales well**: Works much faster for large numbers of files compared to multiple searches.
- **Clear output**: Groups duplicate files together based on the matching criteria.

**Optional Improvements**

- Use checksums (like md5sum or sha1sum) for more accurate duplicate detection.
- Let users choose the directory to scan.
- Add options for previewing or deleting duplicates.

**Evaluation Criteria:**

- Correct implementation.
- Clear explanation of the changes and concepts behind it.
- Code quality and readability.