# CS6.201: Introduction to Software Systems

Mid Lab Exam

## General Instructions

1. This exam consists of **4 questions**. Read all questions carefully before you begin.
2. **Submission filenames:** `1.sh`, `2.py`, `3.py`, `4.py` — for Questions 1 through 4 respectively.
3. **Allowed external libraries:**
   - `sys` and `math` are permitted for **all** Python questions.
   - `numpy` is permitted **only** for Question 2 (and is required for it).
   - **No other external libraries or modules may be imported** (e.g., `itertools`, `collections`, `pandas`, `openai`, etc. are all **prohibited** unless stated otherwise).
4. All Python programs must read input from **standard input** (`stdin`) and write output to **standard output** (`stdout`), unless the question explicitly specifies otherwise.
5. Ensure your scripts and programs produce the **exact** output format specified in each question.
6. Do **not** hardcode any test-case-specific values in your solutions.

### Offline Documentation

A `docs/` folder is provided with the following reference material:

- `docs/numpy.pdf` — NumPy documentation
- `docs/bash.pdf` — Bash documentation
- `docs/python/index.html` — Open this file in a browser to access the Python documentation

### VS Code Extension Setup

If the Python extension is not already installed, you can install it using either method:

- **Double-click** the `.vsix` file in the `extensions/` directory, **or**
- Run the following command in a terminal:

```
code --install-extension extensions/ms-python.python-2026.2.0-linux-x64.vsix
```

---

## Reference Material

The following concepts may be useful for this exam.

### Command Substitution (`$()`) — Bash Manual §3.5.4

Command substitution allows the output of a command to replace the command itself. The standard form is:

```
$(command)
```

You can capture the output of a command into a variable:

```
# Syntax
VARIABLE=$(command)

# Example
$ CURRENT_USER=$(whoami)

# Now try printing the variable
$ echo $CURRENT_USER
# Your current logged in username will be printed out
```

### AWK — Extracting Columns

`awk` is a text-processing command used to analyze, filter, and manipulate structured data such as logs, CSV files, and command output. It works by scanning input line by line and performing actions based on patterns and fields.

#### Built-in Field Variables

AWK's built-in field variables — `$1`, `$2`, `$3`, and so on (`$0` is the entire line) — break a line of text into individual fields (columns). Fields are split on whitespace by default.

To print a specific column:

```
$ awk '{print $n}' inputfile.txt

# or you can pipe the output of another command like this
$ echo -e "hello world\nhellow orld" | awk '{print $2}'
world
orld

$ ps | awk '{print $1, $4}'
# i wonder what this prints
```

### NumPy Matrix (Array) Slicing

In NumPy, slicing lets you extract parts of an array (rows, columns, or sub-matrices) using the syntax:

```
array[start:stop:step]
```

For 2D arrays, slicing is done as:

```
array[row_slice, column_slice]
```

For example, if you want the last two columns of a 3×3 matrix:

```python
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

print(A[:, 1:3])
# [[2 3]
#  [5 6]
#  [8 9]]
```

Note that you must still specify the row slice (`:`) to indicate that you want all available rows.

## Question 1: The File Auditor

**Language:** Bash | **Submit as:** `1.sh`

### Objective

Write a Bash script named `1.sh` that scans a specific directory for storage risks. The goal is to identify **all** large files owned by the current user that are **at or above** a specific size threshold.

### 1. Argument Handling & Validation

Your script must strictly validate the two inputs provided by the user.

- **Count Check:** The script must take exactly **two arguments**:
  1. The path to the directory.
  2. The size threshold in **bytes**.
  - If the argument count is incorrect, print:
    `USAGE: ./1.sh <directory> <size_in_bytes>`
- **Existence Check:** If the provided path does not exist on the system, print: `DIRECTORY DOES NOT EXIST`
- **Type Check:** If the path exists but is a file instead of a directory, print: `NOT A DIRECTORY`

  **Note:** In all error cases, the script must exit immediately after printing the message. Perform the checks in the order listed above (count → existence → type).

### 2. Audit Logic

For the valid directory provided, your script must find **all** files that meet **both** of the following criteria:

1. The file is **owned by the current user** running the script.
2. The file size is **greater than or equal to** the number of bytes provided in the second argument.

   **Clarification:** The size comparison is **not** strict — a file whose size is exactly equal to the threshold **does** match.

   **Hint:** The target directory may contain hidden files (files whose names begin with `.`). These should be included in the audit.

### 3. Output Requirements

- The **filenames only** (not full paths) of all matching files must be saved to a file named `audit_results.txt`.
- This results file must be created **inside the target directory** (the path provided as the first argument).
- The filenames must be listed **one per line**, sorted in **lexicographic order**.
- If the script is run multiple times, `audit_results.txt` must be **overwritten** with the latest results.
- The `audit_results.txt` file must be created even if no files match the criteria (i.e., it may be empty).

**Note:** The `audit_results.txt` file itself should **not** appear in the audit results.

## 4. Technical Constraints

- **No Spaces:** You are guaranteed that filenames and directory paths will **not** contain spaces. You do not need to handle special characters or whitespace.
- **Dynamic Ownership:** Do not hardcode your username. Use system commands (e.g., `whoami`) or environment variables to identify the current user.
- **Pure Bash / Core Utilities:** You may use any standard Bash commands and constructs (loops, redirection, `ls`, `grep`, `if/else`, `find`, `awk`, `stat`, etc.).

## Sample Test Case

There is only **1** sample test case for this question. A directory named `bash_test` is provided.

### Successful Run

```
$ ./1.sh ./bash_test 101
# Result: The file ./bash_test/audit_results.txt is generated.

$ cat ./bash_test/audit_results.txt
.hid.txt
aidf.txt
akdu.txt
aodj.txt
hcbd.txt
ibcu.txt
mciu.txt
```

## Question 2: Majat Makhanpal's Mystical Magical Matrix Multiplication Machine

**Language:** Python | **Required Library:** `numpy` | **Submit as:** `2.py`

Majat Makhanpal was recently hired at Moogle incorporated, where he was tasked with creating a matrix machine to perform mathematical operations for their latest AI model. Unfortunately Majat hasn't written a single line of code ever since he found out about the magical powers of Mopilot and has completely forgotten how to write code on his own. Forbidden from vibe coding and afraid of losing his job, Majat has now turned to you for help. Can you decipher his requirements and make a matrix machine capable of keeping Majat away from the fringes of unemployment?

### Task

Given a matrix of size $a \times b$ and a kernel of size $c \times d$, repeatedly apply the kernel to the matrix until the matrix becomes smaller than the kernel in at least one dimension.

#### Step 1 — Pad the Matrix

Pad the matrix with **zeroes** so that both dimensions become exact multiples of the kernel dimensions:

- Add rows to the **bottom** so that the total number of rows is the smallest multiple of $c$ that is $\geq a$.
- Add columns to the **right** so that the total number of columns is the smallest multiple of $d$ that is $\geq b$.
- If a dimension is already a multiple, no padding is needed along that dimension.

**Clarification:** Find the smallest integers $\lambda$ and $\eta$ such that $\lambda c \geq a$ and $\eta d \geq b$, then pad to reach exactly $\lambda c$ rows and $\eta d$ columns. All padded cells are filled with **0**.

#### Step 2 — Apply the Kernel

Imagine placing the kernel on top of the padded matrix, starting at the top-left corner. The kernel acts like a window of size $c \times d$ that you slide over the matrix in non-overlapping steps:

1. At each position, take the sub-matrix of the same size as the kernel.
2. Perform **element-wise multiplication** between the sub-matrix and the kernel (multiply corresponding entries).
3. **Sum** all the resulting values — this single number becomes one element of the output matrix.

Move the kernel across the matrix in **non-overlapping** steps:

- Move **right** by $d$ columns at a time.
- When you reach the right edge, move **down** by $c$ rows and start again from the left.

This produces a new, smaller output matrix of size $\lambda \times \eta$.

**Hint:** Element-wise multiplication of two NumPy arrays of the same shape can be done simply with `A * B`.

**Step 3 — Repeat**

Take the output matrix from Step 2 and treat it as the new input matrix. Go back to Step 1 (pad if needed) and apply the kernel again. **Repeat** until the resulting matrix is **smaller than the kernel in at least one dimension** (i.e., the number of rows $< c$ or the number of columns $< d$). At that point, stop and print the final matrix.

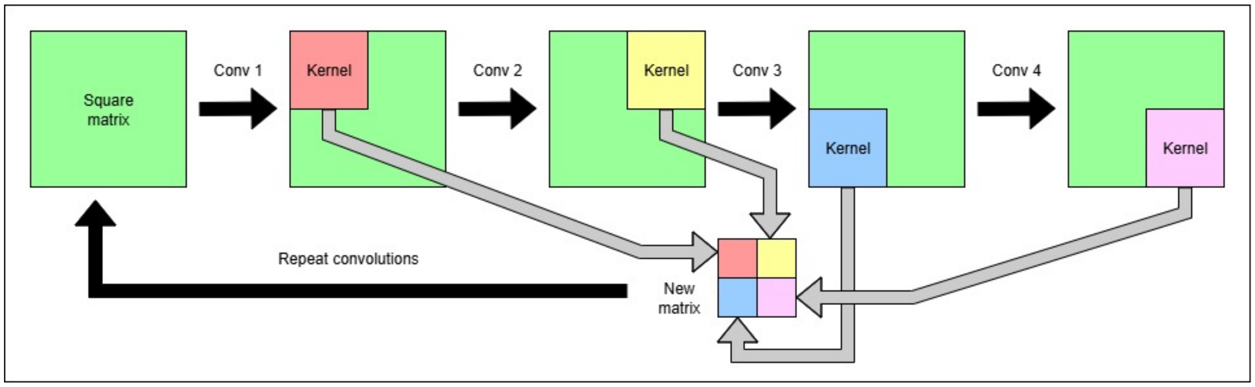

Figure 1: Illustration of kernel application

**Clarifications:**

- The kernel will always have at least one dimension greater than 1, i.e., $\max(c, d) > 1$. This guarantees the matrix shrinks and the process terminates.
- Padding is recomputed at each iteration based on the current matrix size.
- All input values (both matrix and kernel) are **integers**. The operations involved (element-wise multiplication and summation) preserve integer values, so the output will always consist of integers.
- Your output must be printed as **integers**. Printing `11.0` instead of `11` will be marked incorrect.
- When using NumPy, ensure your arrays use an **integer dtype** (e.g., `dtype=int`). If you accidentally parse input as floats, convert back to integers before printing (e.g., using `.astype(int)` or Python's `int()`).
- This algorithm involves only multiplication and addition — there are no division operations, so division-by-zero is not a concern.

## Constraints

| Variable | Range |
|---|---|
| $a$ (matrix rows) | $1 \leq a \leq 500$ |
| $b$ (matrix columns) | $1 \leq b \leq 500$ |
| $c$ (kernel rows) | $1 \leq c \leq a$ |
| $d$ (kernel columns) | $1 \leq d \leq b$ |
| $\max(c, d)$ | $> 1$ |

**Expected time complexity:** $O(a \times b)$

### Input Format

```
<matrix rows> <matrix cols>
<matrix row 1 values separated by spaces>
...
<matrix row A values separated by spaces>
<kernel rows> <kernel cols>
<kernel row 1 values separated by spaces>
...
<kernel row C values separated by spaces>
```

### Output Format

```
<row 1 values separated by spaces>
...
<row R values separated by spaces>
```

Elements within a row are separated by a single space. Each row ends with a newline.

### Sample Test Cases

#### Test Case 1

**Input:**

```
1 2
3 4
1 2
1 2
```

**Output:**

```
11
```

#### Test Case 2

**Input:**

```
2 2
1 2
3 4
2 2
1 0
0 1
```

**Output:**

```
5
```

#### Test Case 3

**Input:**

```
3 3
1 2 3
4 5 6
7 8 9
2 2
1 1
1 1
```

**Output:**

```
45
```

**Test Case 4**

**Input:**

```
4 4
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
2 2
1 2
3 4
```

**Output:**

```
1120
```

**Test Case 5**

**Input:**

```
4 6
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
2 3
1 0 -1
0 1 0
```

**Output:**

```
6 9
18 21
```

## Question 3: Substring Shenanigans

**Language:** Python | **Submit as:** `3.py`

Dreyas Sheb is an eccentric man with a peculiar taste in strings. He will only read a string if it satisfies a certain set of conditions. Being as considerate as he is, he reads the whole string before he discards parts of it for being "far too unpleasant to grace with his eyes". He uses this eccentricity to encode his notes to ensure no one else can study from them. Fortunately members of our *cult of in-class napping* were able to crack the code and now want you to create a program to decipher the notes.

### Task

Given a string $S$, remove all occurrences of characters whose total frequency in the **entire** string is **strictly less than** their specified threshold value. The resulting string must maintain the relative ordering of the original string.

- If a character has a defined threshold and its total count in $S$ is **strictly less than** the threshold, **remove all** occurrences of that character.
- If a character has **no** defined threshold, **preserve all** instances of that character.
- If after removal the resulting string is empty, print `-1`.

  **Clarifications:**

  - "Frequency" means the total count of that character across the **entire** original string, not within a substring.
  - The threshold comparison is **strict**: a character is removed only if its frequency is **strictly less than** its threshold. If its frequency equals the threshold, the character is **kept**.
  - Characters without a threshold are always kept.
  - The input string contains only lowercase English letters (`a`–`z`).

### Constraints

| Variable | Range |
|---|---|
| $\|S\|$ (length of input string) | $1 \leq \|S\| \leq 10^6$ |
| $M$ (number of conditional characters) | $0 \leq M \leq 26$ |
| Characters in $S$ | lowercase `a`–`z` only |

**Expected time complexity:** $O(|S|)$

### Input Format

```
<input string> string S
<number of conditional characters> int M
<conditional character 1> char C1 <frequency threshold> int F1
...
<conditional character M> char CM <frequency threshold> int FM
```

### Output Format

```
-1 (if the resulting string is empty)
OR
<resulting string>
```

### Sample Test Cases

#### Test Case 1

**Input:**

```
aabbcc
2
a 2
b 2
```

**Output:**

```
aabbcc
```

> **Explanation:** `a` appears 2 times (threshold 2, $2 \not< 2$, kept). `b` appears 2 times (threshold 2, kept). `c` has no threshold (kept).

#### Test Case 2

**Input:**

```
aabbc
2
a 3
b 2
```

**Output:**

```
bbc
```

> **Explanation:** `a` appears 2 times but threshold is 3 ($2 < 3$, removed). `b` appears 2 times (threshold 2, kept). `c` has no threshold (kept).

#### Test Case 3

**Input:**

```
abc
3
a 2
b 2
c 2
```

**Output:**

```
-1
```

> **Explanation:** Each character appears once, all thresholds are 2, so all characters are removed. Empty result $\rightarrow$ print `-1`.

**Test Case 4**

**Input:**

```
hello
0
```

**Output:**

```
hello
```

> **Explanation:** No thresholds defined at all, so every character is kept.

**Test Case 5**

**Input:**

```
banana
1
n 3
```

**Output:**

```
baaa
```

> **Explanation:** `n` appears 2 times but threshold is 3 ($2 < 3$, removed). `b` and `a` have no threshold (kept).

## Question 4: Parenthesis Generator

**Language:** Python | **Submit as:** `4.py`

I can't be bothered to write a story here, you get the point, someone is in some dire unescapable situation yada yada yada write code.

### Task

Given an even integer $N$, generate **all valid sequences** of parentheses of length $N$. Each sequence must consist solely of the characters `(` and `)`.

A valid sequence of parentheses is defined by the following rules:

1. Every opening bracket `(` has a corresponding closing bracket `)`.
2. The brackets are properly nested.
3. No closing bracket appears before its corresponding opening bracket.

#### Requirements

- Print all valid sequences, one per line.
- Sequences must be printed in **lexicographic order**.

  **Clarifications:**

  - $N$ is always **even**. An odd $N$ would make valid sequences impossible.
  - Lexicographic order means `(` comes before `)`. So `(())` comes before `()()`.

### Constraints

| Variable | Range |
|---|---|
| $N$ (length of string) | $2 \leq N \leq 20$ |
| $N$ | always even |

**Additional Constraints:**

- You are **not** allowed to use `itertools`.

### Input Format

```
<length of string> int N
```

A single integer on one line.

### Output Format

```
<sequence 1>
<sequence 2>
...
<sequence K>
```

One valid parenthesis sequence per line.

### Sample Test Cases

#### Test Case 1

**Input:**

```
2
```

**Output:**

```
()
```

#### Test Case 2

**Input:**

```
4
```

**Output:**

```
(())
()()
```

#### Test Case 3

**Input:**

```
6
```

**Output:**

```
((()))
(()())
(())()
()(())
()()()
```

#### Test Case 4

**Input:**

```
8
```

**Output:**

```
(((())))
((()()))
((())())
((()))()
(()(()))
(()()())
(()())()
(())(())
(())()()
()((()))
()(()())
```

```
()(())()
()()(())
()()()()
```

**Test Case 5**

**Input:**

```
10
```

**Output:**

```
((((()))))
(((()())))
(((())()))
(((()))())
(((())))()
((()(())))
((()()()))
((()())())
((()()))()
((())(()))
((())()())
((())())()
((()))(())
((()))()()
(()((())))
(()(()()))
(()(())())
(()(()))()
(()()(()))
(()()()())
(()()())()
(()())(())
(()())()()
(())((()))
(())(()())
(())(())()
(())()(())
(())()()()
()(((())))
()((()()))
()((())())
()((()))()
()(()(()))
()(()()())
()(()())()
()(())(())
()(())()()
```

```
()()((()))
()()(()())
()()(())()
()()()(())
()()()()()
```