CSE 3100 Systems Programming Spring 2026

Practice Exam #1 (same as Fall 25 Exam #1)

This is a 110-minute exam. *Please read every question carefully.*
Please read the following list of important items on exam policy and make sure you understand it.

- The Makefile and starter code for the exam can be downloaded from HuskyCT.
- During the exam, you cannot communicate with and/or obtain help from people other than TAs and instructors on exam questions. Particularly, you cannot use any messaging applications.
- During the exam, you can only access data/files on HuskyCT, your github codespaces, man pages and files on your own laptop and on the lab computer you use. You cannot access data/files on other computers/servers. You cannot use your phone. You cannot use Chat-GPT or other generative AI.
- You must submit your code on Gradescope. We only grade submissions we have received on Gradescope.
- Note if your code does not compile, you will lose at least half of the points for that question.
- Do not modify the starter code.
- Submitting to Gradescope after leaving the exam room is not allowed.
- Return the question paper before you leave the exam room.
- After the exam, you cannot discuss/disclose exam problems and/or share your code with students who have not taken the exam.
- Failure to follow the rules will result in a zero on the exam and may cause you to fail the course.

Full Name: ______________________________ NetID: ______________

Lab Section: __________ Date: ____________________

Signature: ____________________________________

Note that a Makefile is provided and can be downloaded from HuskyCT.

Problem 1. (30 points) Find common characters in strings

The task is to identify all the characters that are common across a collection of lowercase strings. To accomplish this, the program relies on the idea of progressively narrowing down a set of candidate characters as it examines each string in the array.

We begin by assuming that every lowercase letter, from `a` to `z`, could potentially be common to all strings. This assumption is gradually refined as we process the input.

For each string, the program creates a temporary record to note which characters actually appear in that string. As the string is scanned character by character, the corresponding entry for each letter is marked. For instance, if the string is `apple`, then the entries for `a`, `p`, `l`, and `e` are set to true, while all others remain false.

Once the string has been examined, the program compares this temporary record with the running global record of common characters. Any letter that is absent from the current string is immediately disqualified from being considered common and is removed from the global record. In effect, the global record is continually updated as the intersection of character sets across all strings.

After all strings have been processed, the global record contains only those characters that were present in every string. These characters are then reported as the output. If no such characters remain, the program explicitly indicates this by printing `None`.

Below is the starter code for this question. (`common.c` in Husky CT). Do not modify starter code or add new additional functions.

```
// Do not modify starter code
#include <stdbool.h>
#include <stdio.h>
#include <string.h>

#define MAX_LEN 100

void commonChars(char arr[][MAX_LEN], int n) {
  int common[26];
  for (int i = 0; i < 26; i++) {
    common[i] = true;
  }
  for (int i = 0; i < n; i++) {

    // fill code here
  }

  printf("Common characters: ");

  // fill code here
  if (!found) {
    printf("None");
  }
  printf("\n");
}

int main(int argc, char *argv[]) {
    .......
}
```

Below is the test case for this question. Submit common.c to gradescope.

```
$./common applet ballet cattle
Common characters: a e l t

$./common app bow cut
Common characters: None
```

Problem 2. (40 points) Zip Linked List
The problem is reordering a non-empty, singly linked list. Given a list of K nodes, the goal is to rearrange its elements in such a way that the first node is followed by the last node, then the second node is followed by the second-last node, and so on, until the list has been completely rearranged.

Formally, the transformation produces the following sequence:

$$1^{\text{st}} \text{ node } \rightarrow \text{ K}^{\text{th}} \text{ node } \rightarrow 2^{\text{nd}} \text{ node } \rightarrow (K-1)^{\text{th}} \text{ node } \rightarrow 3^{\text{rd}} \text{ node } \rightarrow (K-2)^{\text{th}} \text{ node } \rightarrow \ldots$$

Example 1

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6$$

After zipping:

$$1 \rightarrow 6 \rightarrow 2 \rightarrow 5 \rightarrow 3 \rightarrow 4$$

Example 2

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$$

After zipping:

$$1 \rightarrow 5 \rightarrow 2 \rightarrow 4 \rightarrow 3$$

Below is the starter code for this question. (`ziplist.c` in Husky CT). Do not modify starter code. You may add additional helper functions if needed.

```
//Do not modify starter code
//You may add helper functions if needed
#include <stdio.h>
#include <stdlib.h>

// Node structure
typedef struct Node {
  int data;
  struct Node *next;
} Node;

// Create a new node
Node *createNode(int data) {...}
// Insert at end
void insertEnd(Node **head, int data) {...}
// Print list
void printList(Node *head) {...}

void zipList(Node **headRef) {
  // fill code here

}

int main(int argc, char *argv[]) {
  Node *head = NULL;
  if (argc < 2) {
    printf("Usage: %s <list of integers>\n", argv[0]);
    return 1;
  }
  for (int i = 1; i < argc; i++) {
    int val = atoi(argv[i]);
    insertEnd(&head, val);
  }
  printf("Original list:\n");
  printList(head);
  zipList(&head);
  printf("Zipped list:\n");
  printList(head);

  // fill code here

  return 0;
}
```

Below are some sample outputs. Submit ziplist.c to gradescope.

```
$ ./ziplist 1 2 3 4 5 6
Original list:
1 2 3 4 5 6
Zipped list:
1 6 2 5 3 4

$ ./ziplist 1 2 3 4 5
Original list:
1 2 3 4 5
Zipped list:
1 5 2 4 3
```