UCLA Computer Science 35L final exam - Winter 2020

Na[redacted]

Student ID: [redacted]

100 points total. Open book, open notes, open computer. Answer all questions yourself, without assistance from other students or outsiders. Although this exam is designed to take 180 minutes, under the circumstances it is simply a takehome exam.

Print this exam, write your answers on it, scan the completed exam, and upload your scans to CCLE Gradescope by 15:00 Monday (Los Angeles time). If you do not have easy access to a scanner, carefully photograph the sheets of paper with your cell phone and upload the photographs. Save your filled-out exam and do not give or show it to anybody other than an instructor or TA; we will send you instructions later about what to do with your filled-out exam.

If you lack access to a printer, read the exam on your laptop's screen, write your answers on blank sheets of paper (preferably 8½″×11″) with one page per question, and upload the scanned sheets of paper. Please answer every question on a new sheet of paper. At the end of the exam, you should have scanned and uploaded as many photographs as there are questions. If you do not answer a question, scan a blank sheet of paper as the answer.

As previously announced, the exam is open book and open notes, but due to circumstances we are also making it open computer. You can use your laptop to use a search engine for answers, and to run programs designed to help you answer questions. However, do not use your computer or any other method to communicate with other students or outsiders, or anything like that. Communicate only via CCLE and Gradescope to obtain your exam and upload your scanned results, or via Zoom or email with the instructor or TAs.

IMPORTANT Before submitting the exam, certify that you have read and abided by the above rules by signing and dating here:

Signature: [redacted] Date: 3/15/20

1 (1 point). Every directory in Linux is initialized with at least two other directories inside it. What are these two directories and why are they there?

These directories are . (current directory) and .. (parent directory).

They are there to allow the use of relative paths, so when moving between directories you don't have to type full paths but just . or .. for current or parent directory.

2 (4 points). Consider an inode that we'll call inode A, with a link count of 3. This inode holds information about the regular file 'foo'. Consider another inode B also with a link count of 3 that holds information about the directory 'bar'. Answer the following questions about inode A as if they were executed sequentially; e.g., if the link count increases from 3 to 6 in (1a), use the new link count of 6 for (1b).

2a. What happens to the link count for A when we create two new hard links (h1 and h2) to foo?

increases to 5 links

2b. What happens to the link count for A when we create a symbolic link s1 to 'bar' and then a symbolic link s2 to s1?

stays at 5 links

2c. Is it possible to create a symbolic link inside the directory 'bar' that causes a cycle? If so, write a command to create a cycle; if not, explain why not.

Yes, using the command:

```
ln -s . mycycle
```

Then one can keep using 'cd mycycle' to get an infinite path /bar/mycycle/mycycle/mycycle...

2d. What happens to link s1 when the directory 'bar' is removed?

s1 becomes an invalid link, following it will cause an error

3 (2 points). Consider the following Bash script:

```
#!/bin/bash
a=2
sed 's/2'${a}'}/f/g' < input.txt
```

Assuming the script is executable and that input.txt exists and is non-empty, what does this script do?

Prints the contents of input.txt to stdout, but replaces all occurances of the consecutive characters 22} with the letter f.

4 (2 points). Given an existing non-empty text file, myfile, what does the following command do?

```
ls -A | grep '^[.].*' >> myfile 2>&1
```

in the current directory not including . and ..

Appends a list of all hidden files/directories to the file myfile.
Additionally stderr is redirected to whatever stdout is outputting to, so error messages are also appended to myfile.

5 (4 points). Assume a Bash script named foo exists and is executable. Also assume foo has the following structure:

```
#! /bin/bash
echo ______
echo ______
echo ______
echo "line 4: ca${4}ts"
```

where the arguments of the first three echo commands have been left blank. Assume the following shell command is run in the directory where foo exists:

```
./foo I love 'eating rro'
```

The resulting stdout output is as follows:

```
line 1: I
line 2: $2
line 3: I love eating rro
line 4: cats
```

Complete the echo statements to have the script produce the given stdout output. Make sure that the script can run with arbitrary arguments. For example:

```
./foo John Smith loves rro
```

should produce the following stdout output:

```
line 1: John
line 2: $2
line 3: John Smith loves rro
line 4: carrots
```

```
echo "line 1: $1"
echo 'line 2: $2'
echo "line 3: $@"
echo "line 4: ca${4}ts"
```

6 (8 points). Write an extended regular expression that matches non-empty strings that consist of only characters A and B, in which the number of As is 3n, where n is a non-negative integer. You will get 4 points if your solution doesn't exclude the empty string but is otherwise correct. These are example matches, one example per line:

```
B
BAAA
AABBA
BBBB
ABABAB
AAAAAA
```

((B*A){3})+B*|B+

7 (8 points). Suppose you receive a comma-separated values (CSV) file "locations.csv", containing a list of locations and their coordinates. Here are the first 5 lines in the file:

```
id,location
1,"In-N-Out"
2,"Salt & Straw"
3,"Rocco's Tavern"
4,"Tatsu Ramen"
```

The first row contains the column names separated by commas. Each successive row contains a unique data entry.

A correctly formatted data line follows these rules:

* Exactly two fields separated by commas, i.e., each line has exactly one comma.

* The first field (i.e., id) is a non-empty string of ASCII digits.

* The second field (i.e., location) starts and ends with double quotation marks ("), and may use uppercase/lowercase ASCII letters, spaces ( ), hyphen-minuses (-), ampersands (&), or apostrophes (').

You notice after looking through the file that some entries don't follow the format listed above, and you'd like to remove them. Write a shell command that reads the file from stdin and prints only correctly formatted lines. Do not include the header line ("id,location") in the output of correctly formatted lines.

```
sed 1d < locations.csv | grep -E "^[0-9]+,\"[A-Za-z'& -]*\"$"
```

Space

Problems 8 through 10 use the Python 3 class below:

```
class FruitSalad:
  def __init__(self, fruits):
    self.fruits = fruits

  def add_fruits(self, new_fruits):
    self.fruits += new_fruits
    new_fruits = []

if __name__ == "__main__":
  x = ['apple', 'banana']
  y = ['strawberry', 'watermelon']

  salad = FruitSalad(x)
  salad.add_fruits(y)
```

8 (3 points). What is the value of the list y after calling salad.add_fruits()? Explain your reasoning in a couple sentences.

y = ['strawberry', 'watermellon']

In Python, object references are passed by value. So inside the add_fruits method, at first new_fruits refers to the same object as y. The line new_fruits = [] changes the object that new_fruits references, and since object references are passed by value, y still refers to the same object it originally did and that object is unchanged.

9 (3 points). Suppose we add a class variable melons to FruitSalad:

```
class FruitSalad:
  melons = ['watermelon', 'honeydew', 'cantaloupe']
  ...
```

Write an instance method find_melons() that returns the index of every melon from melons that is found in self.fruits. The return value should be a Python dictionary mapping the melon (a string) to the index. For example, running find_melons() on the prior code should return { 'watermelon': 3 }

```
def find_melons(self):
    d = dict()
    for melon in FruitSalad.melons:
        if melon in self.fruits:
            d[melon] = self.fruits.index(melon)
    return d
```

10 (3 points). Write an instance method remove_melons() that returns a copy of self.fruits with all the melons removed. The original self.fruits should not be modified. You can use your implementation of find_melons(). For example, running remove_melons() on the prior code should return ['apple', 'banana', 'strawberry'].

```
def remove_melons(self):
    non_melons = []
    for fruit in self.fruits:
        if fruit not in FruitSalad.melons:
            non_melons.append(fruit)

    return non_melons
```

11 (2 points). What's the problem with the following C code?
Fix the problem in a way that best matches the evident intent.

```
#include <stdio.h>

typedef struct {
    int x;
    int y;
} Point;

Point *create_point() {
    Point p;
    p.x = 0;
    p.y = 0;
    return &p;
}

int main() {
    Point *p = create_point();
    printf("(x, y) = (%d, %d)\n", p->x, p->y);
    return 0;
}
```

Problem:

p in create_point() is a local variable available only within the create_point function, once exiting the function the pointer to it is no longer valid, so p in main will have undefined behavior

1. at top, include:

```
#include <stdlib.h>
```

2. edit create_point function:

```
Point * create_point() {
    Point * p = (Point *) malloc (sizeof(Point));
    p->x = 0;
    p->y = 0;
    return p;
}
```

3. edit main function:

```
int main() {
    Point * p = create_point();
    printf("(x, y) = (%d, %d)\n", p->x, p->y);
    free(p);
    return 0;
}
```

12. Suppose we are worried that attackers have taken control of the SEASnet network (but not the SEASnet servers or anything else) and that the attackers want to prevent this final exam from being conducted fairly. We decide to conduct this exam via ssh in order to defeat the attackers, as follows. Students use ssh to get a copy of the exam from /home/eggert/cs35L/final.pdf, print it out, take the exam, scan the result into a file named 123-456-789.pdf (where 123-456-789 is their student ID), and then upload the file into /home/eggert/cs35L/submit/123-456-789.pdf and the professor and TAs take it from there.

12a (1 point). Describe a simple way the attackers can defeat this approach and prevent the exam from being conducted fairly anyway.

Since ssh is designed to be secure over an insecure network, it will be hard for the attackers to intercept, change, and resend any data being sent. But they can just block traffic on the network at certain times, thus preventing students trying to download or upload at that time from completing the test.

12b (1 point). Suppose the attackers are passive, i.e., they do not alter how the SEASnet network behaves, but can still get copies of all the data sent across the network. What information can they easily learn about the exam and how it was conducted? What information will it be hard for them to discover?

The attackers can see the amount of traffic on the network and the time at which data is sent, so they can easily infer the start time of the exam, the end time, and perhaps the size of the exam and the number of students taking the exam. Since the data is encrypted, it will be hard for them to understand the contents of what is being sent though.

12c (2 points). Changing the subject slightly, does it ever make sense to use SSH's 'scp' command to copy a file directly from one SEASnet GNU/Linux host to another? If so, explain why; if not, explain why not.

No, the Seasnet GNU/Linux hosts all have access to the same files (ie. if on lnxsrv07 I create a document under my home directory, I will be able to see it if I log onto lnxsrv09), so using 'scp' to copy from one host to another would be pointless.

13 (14 points total). You must be very familiar with grep by now. Why not write one yourself? In this problem you will use C to implement a grep variant with a different regular expression format.

The normal GNU grep command reads from the standard input and outputs any matching lines to standard output, with matching text highlighted if standard output is a terminal. In our implementation, grep will still read from standard input and output only the matching text from the matching lines, and output nothing if nothing is matched. The matching text for the current line should be output right after the program receives a newline or reaches EOF.

Your implementation will only support a small set of regular expression tokens. The supported list includes the following:

```
    .     \
```

Any characters other than those in the list will be literal. The backslash \ is an escape character that can make any single character after it literal; a valid regular expression cannot end in an unescaped backslash. Similar to homework 4, the input might NOT have a trailing newline. In such case, your program should still correctly process the last line and output the matching text.

For simplicity, you need only return the first match for every matching line. For example, if the regex is ‘a’, and the input is ‘aaaaa’, you only need to output ‘a’ so that you would need to parse the input only once to find the matching text.

We will assume there will be no errors while reading, writing, and allocating data so that you do not need to worry about outputting error messages and exiting the program for these cases.

Use getchar and putchar to do I/O, and use malloc, realloc, free to allocate memory for your input data.

Use the template on the next page to implement your program.

(continued from previous page)

```c
#include <stdio.h>
#include <stdlib.h>

const char LF = '\n';

int valid_regex(const char *regex);
void process(const char *regex);
void addchar(char *line, size_t *nchars, char ch);
char *match(const char *regex, char *line);

int main(int argc, char *argv[]) {
  if (argc != 2) {
    fprintf(stderr, "mygrep: there should be only one argument\n");
    return 1;
  }
  if (!valid_regex(argv[1])) {
    fprintf(stderr, "mygrep: invalid regular expression\n");
    return 1;
  }
  process(argv[1]);
  return 0;
}

void process(const char *regex) {
  char *matched = NULL;
  char *line = NULL;
  size_t nchars = 0;
  while (1) {
    int ch = getchar();
    if (ch == EOF) {
      if (nchars != 0) {
        line = addchar(line, &nchars, LF);
        line = match(regex, line);
        nchars = 0;
      }
      break;
    } else {
      line = addchar(line, &nchars, ch);
      if (ch == LF) {
        line = match(regex, line);
        nchars = 0;
      }
    }
  }
}
```

Your task is to write the functions valid_regex, addchar and match with the given signatures. The regex parameter is a static character array that contains the null-terminated regex string.

(continued on next page)

(continued from previous page)

13a (1 point). Implement the function ‘valid_regex’ that returns nonzero if its argument is a valid regular expression, zero otherwise.

```
int valid_regex(const char *regex)
/* Write your implementation here.  */
{
    int escaped=0;
    int i;
    for(i=0; regex[i]; i++){
        if(escaped)
            escaped=0;
        else if(regex[i]=='\\')
            escaped=1;
    }
    return (!escaped);
}
```

(continued from previous page)

13b (3 points). Implement the function 'addchar' that adds a character to the dynamically allocated array. You only need to use realloc in this function.

```
char *addchar (char *line, size_t *nchars, char ch)
/* Write your implementation here.  */
{
    (*nchars)++;
    line = (char *) realloc(line, *nchars * sizeof(char));
    line[*nchars-1] = ch;
    return line;
}
```

(continued from previous page)

13c (12 points). Implement the function 'match' that can match the line with the corresponding regex, and prints out the matching text inside the function. The function should always return NULL so that the line variable will get reset after matching. Please free all allocated memory inside the match function.

There are at least two approaches to implement this function. The first one would be storing the matching characters to a new dynamically allocated array and print out after iterating through the line. The other one would be having the start index and the end index, and then moving these indices to circle the range of text in the line that needs to be printed.

```
char *match (const char *regex, char *line)
/* Write your implementation here.  */
{
    int searchstart;
    for(searchstart=0; line[searchstart]!='\n'; searchstart++){
        int regexindex=0;
        int lineindex=searchstart;
        for( ; regex[regexindex] && line[lineindex]!='\n'; regexindex++, lineindex++){
            char r =regex[regexindex];
            char c =line[lineindex];
            if(r=='.')
                continue;
            if(r=='\\'){
                regexindex++;
                r=regex[regexindex];
            }
            if(r!=c)
                break;
        }
        if(!regex[regexindex]){
            int printindex;
            for(printindex=searchstart; printindex<lineindex; printindex++)
                putchar((int)line[printindex]);
            putchar('\n');
            break;
        }
    }free(line);
    return NULL;
}
```

14 (5 points). Use the 'read' system call with buffer size 1 to write a naive implementation of getchar() with the following function signature taken from <stdio.h>:

```
#define EOF (-1)
/* Return a character read from stdin, or EOF if no character/error.  */
int getchar(void);
```

```
int getchar(void){
    int c;
    ssize_t n;
    n = read(0, &c, 1);
    if (n != 1)
        return EOF;
    return c;
}
```

15 (10 points total). You are maintaining a C project, which should define how to compute the sum and difference of any two strings of the same length.

Your task is to write two files: main.c and Makefile. Please ignore integer overflow in this question.

You are given five files that you can assume work: computechar.h, addchar.c, subchar.c, addstr.c, substr.c. These files are as follows:

computechar.h:
```
  int addChar(char a, char b);
  int subChar(char a, char b);
  int addStr(char *a, char *b);
  int subStr(char *a, char *b);
```

addchar.c:

```
  #include <computechar.h>
  int addChar(char a, char b)
  {
    ...
  }
```

subchar.c:

```
  #include <computechar.h>
  int subChar(char a, char b)
  {
    ...
  }
```

addstr.c:

```
  #include <computechar.h>
  int addStr(char *a, char *b)
  {
    int result = 0;
    /* for index i between 0 and len(a)-1:
    result += addChar(a[i], b[i]); */
    ...
    return result;
  }
```

substr.c:

```
  #include <computechar.h>
  int subStr(char *a, char *b)
  {
    int result = 0;
    /* for index i between 0 and len(a)-1:
    result += subChar(a[i], b[i]); */
    ...
    return result;
  }
```

(continued on next page)

(continued from previous page)

main.c (4 points):

It takes three arguments <arg1> <arg2> <arg3> where <arg1> is either “add” or “sub”. Based on <arg1>, your main.c should dynamically link either libaddstr.so or libsubstr.so via dlopen(), dlsym(). Your main.c should output an integer value, which is the sum if <arg1> is “add” and the difference if <arg1> is “sub”. <arg2> and <arg3> are two non-empty strings with the same lengths.

Your main.c should also handle errors correctly, including incorrect <arg>’s, dlopen() errors, dlsym() errors, and dlclose() errors. Exit with status 1 if any of the errors above occurred.

Examples:

```
./main add Com Sci # Outputs an integer.
./main sub Oh ok   # Outputs an integer.
./main diff a b    # Exits with status 1 because “diff” is not valid.
```

```
/* Put your complete implementation of main.c below.  */
#include <string.h>
#include <stdio.h>
#include <dlfcn.h>
int main(int argc, char **argv){
        if(argc!=4 || strlen(argv[2])!=strlen(argv[3]))
            return(1);
        int(*opp)(char*a, char*b);
        void *handle;
        if(!strcmp(argv[1], "add")){
            handle=dlopen("libaddstr.so", RTLD_LAZY);
            if(!handle)
                return(1);
            opp=dlsym(handle, "addStr");
        }
        else if(!strcmp(argv[1], "sub")){
            handle=dlopen("libsubstr.so", RTLD_LAZY);
            if(!handle)
                return(1);
            opp=dlsym(handle, "subStr");
        }
        else
          return(1);
        int out = opp(arg[2], arg[3]);
        printf("%d\n", out);
        if(dlclose(handle))
            return(1);
        return(0);
}
```

```
if(dlerror()){
   dlclose(handle)
   return 1;
}
```

(continued from previous page)

Makefile (6 points):

It should at least include rules to:

(1). Compile addchar.c and subchar.c into an archive: libcomputechar.a.

(2). Compile addstr.c and substr.c into dynamic libraries libaddstr.so and libsubstr.so.

(3). Compile main.c into an executable 'main'.

You can assume all files are under the same directory.

Make sure each module includes the minimal set of include files.

```
OPTIMIZE = -O2
CC = gcc
CFLAGS = $(OPTIMIZE) -g3 -Wall -Wextra
         -march=native -mtune=native
         -mrdrnd
```

```
# Put your complete implementation of Makefile below.
default: main libaddstr.so libsubstr.so
main: main.o
  $(CC) $(CFLAGS) -ldl -Wl,-rpath=$(PWD) main.o -o main
main.o: main.c
  $(CC) $(CFLAGS) -c main.c -o main.o
libaddstr.so: addstr.o libcomputechar.a
  $(CC) $(CFLAGS) -shared addstr.o libcomputechar.a -o libaddstr.so
libsubstr.so: substr.o libcomputechar.a
  $(CC) $(CFLAGS) -shared substr.o libcomputechar.a -o libsubstr.so

addstr.o: addstr.c computechar.h
  $(CC) $(CFLAGS) -fPIC -c addstr.c -o addstr.o
substr.o: substr.c computechar.h
  $(CC) $(CFLAGS) -fPIC -c substr.c -o substr.o

libcomputechar.a: addchar.o subchar.o
   ar rcs libcomputechar.a addchar.o subchar.o
addchar.o: addchar.c computechar.h
 $(CC) $(CFLAGS) -fPIC -c addchar.c -o addchar.o
subchar.o: subchar.c computechar.h
 $(CC) $(CFLAGS) -fPIC -c subchar.c -o subchar.o
```

16 (10 points total). Imagine you are on a team of 3 software engineers. You are working together to create a new web application which will require 3 different python code files: server.py, config.py, data.py.

Your team plans for 10 distinct 'features' that will eventually need to be created for the application to be usable.

16a (1 point). This project will require version control. Say in one line what tool(s) would you use to enforce version-control.

I would use Git

16b (5 points). Explain in at most half a page what kind of workflow each software-engineer should follow. Below are potential discussion points, and try to use at least 3 points in your argument. (Your 3 points don't need to come from this list)

* Should engineers use branches, and if so - then how?
* If you plan to merge commits, how should merge conflicts be handled?
* If you plan to merge commits, will your team prefer 'git merge' or 'git rebase'?
* Will you use a 'centralized' remote repo or not?
* How should engineers make sure their work is in sync?

I will use a 'centralized' remote repository for engineers to push and pull changes from. Before starting a code update, an engineer will pull from the remote repository to ensure their local repository is up to date. They will then create and check out a new branch, make their changes on that branch, and commit them to that branch. Before merging they will fetch from the remote repository to ensure their repository is up to date, then swich to the master branch and merge their new changes. After this, they will push to the remote master. Small merge conflicts can be handled manually. If there is a large merge conflict, the changes one made can be saved locally, and then that person can revert to the older version where the problem occured, fix it, add back their changes, and then do the merge again.

16c (1 point). Explain in 2-3 sentences any concerns about your solution.

My solution results in a lot of pushing of commits to the remote repository. For a small team this should be fine, but if the team grows this may lead to far too many commits on the remote master, making it difficult to find older versions and possibly leading to a lot of merge conflicts if the engineers are editing the same files.

16d (3 points). Assume you are in the middle of the same project. You were given the task to create the feature 'Add new URL path to server.py'. What git commands would you use to emulate the workflow you described earlier? You can fill in your answer in a format like below:

```
# Any pre-coding Git commands here

// You make and test your changes to server.py. (You don't need
// to write anything else here).

# Any post-coding Git commands here.
```

pre-coding:

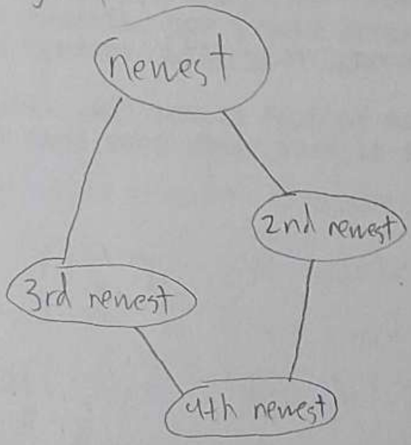
```
git pull
git branch new_url_path
git checkout new_url_path
```

post-coding:

```
git add server.py #also include any other files you changed
git commit -m "Add new URL path to server.py"
git fetch
git checkout master
git merge new_url_path
git push origin master
```

17 (2 points). When you run the command 'git log' on one branch, e.g., 'git log master', is it possible that one of the commits is not the parent of the commit that comes right before it in the output? If it's not possible, explain why. If it's possible, give a scenario where this can happen.

Yes, it is possible.
By default, git log will print commits in reverse chronological order. "git log master" will print all commits reachable from master. If you have the commit graph:



The order printed will be 1st, 2nd, 3rd, then 4th newest. But 3rd newest is not the parent of 2nd newest, so it is possible.

18 (4 points). Assume you have a program G that prints out a directed edge for every pair of parent and child commit hashes in a Git repository, where an edge will look like

```
h_parent -> h_child
```

where h_parent is the hash of the parent commit, h_child is the hash of the child commit.

Someone claims that they have a program P that outputs all the commit hashes in a topological order in which child commits are printed before ancestral commits.

Describe how can you use the output of program G to verify that the output of program P is a valid topological order. There is no need to write actual code.

For every pair of directed edges printed out by program G, ensure that as you iterate over the topological order output from program P from child commits to ancestral commits, you reach the h_child commit before you reach the h_parent commit. If this is the case for every directed edge outputted by program G, then the program P topological ordering is correct. But if you ever reach an h_parent commit before reaching the h_child commit, then the topological ordering is wrong.

19 (10 points). Consider the following text taken from:
Guzdial M. Beware of Hurting Our Weakest Students when Moving Classes Online. Blog@CACM. 2020-03-10.

> The biggest cost of moving our classes online will likely be the decreased learning and lower grades, particularly of our weakest students. Fully on-line classes can lead to less learning than face-to-face classes (see 2017 review paper here). Online learning has a differential impact on students. A 2013 paper studying 40,000 students found that students who come in with lower grades suffer the most in performance when moving online (see paper here). A 2018 New York Times article (see link here) describes about how online classes hurt students who most need help. Justin Reich (MIT) has written a great Twitter thread about all the evidence showing that moving classes online will hurt the most vulnerable students -- start here. His recommendation is to simply cancel classes until June or September. That would be better than creating greater disparity and inequity in our classes.

Discuss the relevance of the technology that you covered in your Assignment 10 presentation as a possible way of addressing the problems that Guzdial raises. If your Assignment 10 technology is largely irrelevant, state why it's irrelevant and discuss how you'd address the problems by using the Zoom features that have been employed during the past week at UCLA.

The topic I discussed in assignment 10 was a proposed new method of encryption that would be safe from being hacked by quantum computers. While this is very related to internet security, of which Zoom and other internet platforms rely on, it does not address the problems of how online classes will hurt students who need help the most.

To address these problems, Zoom will need to be used to its full potential. One advantage of using Zoom is that lectures can be recorded and rewatched later. So although an online lecture will tend to be weaker than an in-person lecture, being able to rewatch the lecture may be able to make up for this. To make the lectures as convenient as possible for students to rewatch, lecturers should provide timestamps for when certain topics were discussed. This would make the idea of going back and watching a lecture less daunting to students, as they could jump to the specific parts of the lecture they want. Additionally, office hours can be recorded. So while in-person office hours can be difficult to attend due to schedules, online hours can be more accesible