Exam 2 Review - Fall 2020

# Exam Logistics

Exam information is sent to students by email and made available on eCampus. Make sure that you read the information already available and pay attention to any additional logistic documents released by the instructors.

If you have questions about any logistic details for Exam 2, please ask on Piazza.

# Exam Study Strategies

The strategies we delineated for Exam 1 continue to apply:

* Understanding at a high level will help you code better during an exam. So, as you study, ask yourself…
  + *What is happening?*
    - *High level: What is the goal?*
    - *Code level: What are the effects and consequences of statements, etc.?*
  + *Why is it that way?*
    - *High level: Why are certain design decisions made? Why is it designed that way?*
    - *Code level: What is the code that accomplishes a higher-level goal? Why is it done that way?*
  + *How is it done?*
    - *High level: How do you accomplish a high-level goal?*
    - *Code level: How do you write the code?*
* Study Strategy: Prioritize how you study to maximize your ability to demonstrate concept mastery while not overwhelming yourself.
  + For a solid passing grade focus on class topics, slides, examples, homework and labwork.
    - Make sure you understand anything coded in homework, labwork, and in-class examples.
      * What general (higher level) principles were illustrated in a labwork/homework?
        + Could you explain it in general?  
          (i.e. what/why rather than how)
        + Be able to solve and write code for similar problems.
  + After you’ve done all you can do in the “for a solid passing grade” above, then focus on zyBook Challenge Activities and Participation Activities.
    - If a challenge or activity seems fuzzy, review the text for clarification.
    - Key terms are bolded.
  + After you’ve done all you can from the previous approaches and if you are still fuzzy about a topic, read about it in a different source or textbook. You can also talk to someone who can explain it in a different way.

# Exam Topics

## Problem solving requires accumulative knowledge of all topics covered in the course. *This exam will focus on:*

## Input/Output Streams

* File I/O
* String streams

## Parameter passing

* Pass by value
* Pass by reference (a variable or a pointer variable)
* const modifier

## Dynamic Memory

* Operations
  + Allocation
  + Deallocation
  + Access
* Where allocated
* Memory Management
  + Garbage Collection
  + Memory Leaks: definition, prevention strategies
* Dynamic 1D and 2D Arrays
  + create (declare, define/allocate, initialize)
  + read (access, search)
  + update (insert, remove, resize)
  + destroy (deallocate)
  + Traversal

# Practice Problems

Note that practice problems are generally greater than or equal in difficulty to what you will find on the exam. Exam questions are designed to be challenging and doable in the time provided. Some of the practice questions are longer than what you would see on an exam, but should give you good practice.

Additional practice problems can be doing the parts of the labworks that you did not work on.

## PathLength

Write a function with declaration

**double pathLength(double\*\* distance, int n, int\* path, int m)**where

* distance is a n by n 2d-array such that position distance[i][j] stores the road distance in miles from city i to city j;
* path is an integer array with m elements that stores a sequence of cities visited in a trip, i.e., 0 <= path[i] < n for all i such that 0 <= i < m
* n and m are greater than zero.

The function should return the length in miles of the path.

For example, for n = 5, distance the matrix below

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0.0 | 30.0 | 10.0 | 70.0 | 10.0 |
| 30.0 | 0.0 | 45.0 | 100.0 | 50.0 |
| 10.0 | 45.0 | 0.0 | 85.0 | 20.0 |
| 70.0 | 100.0 | 85.0 | 0.0 | 100.0 |
| 10.0 | 50.0 | 20.0 | 100.0 | 0.0 |

and m = 3, path

|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |

pathLength(distance, n, path, m) returns 30 + 45 = 75.

For same n and distance, but with m = 6 and path

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 3 | 2 | 0 |

pathLength(distance, n, path, m) returns 30 + 30 + 70 + 85 + 10 = 225.

## 

## AvgMatrix

Write a function with declaration  
void avgMatrix (double\*\* inArray, int rows, int columns, double\*\* outArray)

that gets a 2d-array inArray with rows >= 1 rows and columns >= 1 columns with double elements and calculates outArray as the average matrix, i.e., outArray is an rows by columns matrix where each element is obtained by the average of the (at most 8) neighbor elements in inArray.

Example:

For the 3 by 4 matrix:

|  |  |  |  |
| --- | --- | --- | --- |
| 0.5 | 2.0 | 1.2 | 3.0 |
| -1.0 | 1.5 | 3.0 | 2.4 |
| 0.0 | 1.0 | 1.5 | 2.0 |

the outArray would be:

|  |  |  |  |
| --- | --- | --- | --- |
| 0.833 | 1.04 | 2.38 | 2.2 |
| 1.0 | 1.025 | 1.825 | 2.14 |
| 0.5 | 1.0 | 1.98 | 2.3 |

## Minus Odd Column

Write a function with declaration  
void minusOddColumn(int\*\* mat, int n)

where n >= 1 and mat is an n by n 2d-array of non-negative integers.

The function should find the column in mat with more odd numbers and replace all of

its elements with -1. If multiple columns have the highest presence of odd numbers,

the first of them (i.e., the one with the lowest index) should be chosen to have its

elements replaced by -1.

For example, for n = 5 and mat:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 30 | 10 | 70 | 10 |
| 30 | 0 | 45 | 100 | 50 |
| 10 | 45 | 0 | 85 | 20 |
| 70 | 100 | 85 | 0 | 100 |
| 10 | 50 | 20 | 100 | 0 |

the function will result in mat

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 30 | -1 | 70 | 10 |
| 30 | 0 | -1 | 100 | 50 |
| 10 | 45 | -1 | 85 | 20 |
| 70 | 100 | -1 | 0 | 100 |
| 10 | 50 | -1 | 100 | 0 |

For n = 2 and mat

|  |  |
| --- | --- |
| 0 | 30 |
| 30 | 0 |

the function will result in mat

|  |  |
| --- | --- |
| -1 | 30 |
| -1 | 0 |

# get\_even\_numbers

Write a function that receives an array **A** of **n** integers (all elements >= 0) and returns a dynamically allocated array containing all the even elements in A. It also returns in the parameter **m** the number of elements in the created array.  
The signature of the function is:  
  
int\* get\_even\_numbers(int\* A, unsigned int n,   
 unsigned int& m);

# Separate even from odds

Write a function that receives an array **A** of **n** integers (all elements >= 0) and returns a dynamically allocated array containing all even elements of A first, then all odd elements of A. The signature of the function is:  
  
int\* separate\_even\_odd(int\* A, unsigned int n);

# 

# Array ordered by frequency

Write the function and the main program as specified below:

* 1. write a function with the following signature:  
     int\* get\_array\_ordered\_by\_frequency (int \*a,   
      unsigned int size\_a  
      unsigned int& new\_array\_size);  
     that, given an array **a** of **size\_a** elements, it returns a dynamically allocated array such that the new array contains all elements of a (without repetition), ordered by how frequently they appear in a.  
     Function parameters:
     1. **a** is an array of integers
     2. **size\_a** is the number of elements in array a
     3. **new\_array\_size** is the number of elements in the array being created by the function

The function returns the address of a dynamically created array such as the new array contains the numbers that appear in **a,** ordered by frequency  
  
Example:  
For array a = {-1, 3, -1, 1, 1, 4, 3, 3}, the function returns array {3, -1, 1, 4} or {3, 1, -1, 4} (since -1 and 1 appear the same number of times) and new\_array\_size will be 4.

* 1. write the main program that reads **n**, an array of **n** integers, and prints the list of numbers that appear in the given array, ordered by how frequently these numbers appear in the input array. Use arrays that are dynamically allocated and make sure your program is not leaking memory.

# Remove all-zeros rows

Write a function that, given a 2D array, returns a new 2D array containing only the rows that have at least one non-zero element in the row.  
int\*\* remove\_allzeros\_rows(int\*\* matrix,   
 int nrows,   
 int ncolumns,   
 int& new\_nrows);

If nrow or ncolumns is zero, throw an exception.