**HW3**

* Due: Friday, October 18th, 11:59pm PST
* Directory name in your github repository (<https://github.com/usc-csci104-fall2019/hw-username>) for this homework (case sensitive): hw3
* Place your answers to problems 1,2,3 in a file named hw3.txt
* Place your code to problem 4 in permutations.cpp, problem 5 in cave.cpp, and problem 6 in company.h and company.cpp.
* Include a README.md under hw3 directory that explains how to compile and run your program for each coding problem. Also include any other information we need to consider when running your programs.

**Problem 1 (Abstract Data Types, 10%)**

For each of the following data storage needs, describe which abstract data types you would suggest using. Natural choices would include list, set, map, stack, and queue but also any simpler data types that you may have learned about before.

Try to be specific, e.g., rather than just saying "a list", say "a list of integers" or "a list of structs consisting of a name (string) and a GPA (double)". Also, please give a brief explanation for your choice: we are grading you at least as much on your justification as on the correctness of the answer. Also, if you give a wrong answer, when you include an explanation, we'll know whether it was a minor error or a major one, and can give you appropriate partial credit. There may be multiple equally good options, so your justification may get you full credit.

1. a data type that stores the histories of all of The Rani's subjects, ordered by their number in the initial subject pool (an integer from 1 to n).
2. a data type that stores all of the students that earned an A grade in CSCI 103.
3. a data type that stores all of the students in CSCI 104: given a student name, it brings up the student with that name.
4. a data type that stores all of the students that were in CSCI 103 in Spring 2019. Given a grade, it brings up all of the students that earned that grade.

**Problem 2 (Linked Lists, Recursion, 10%)**

Consider the following C++ code. What linked list is returned if funcA is called with the input linked list 1,2,3,4,5? All of the points for this problem will be assigned based on your explanation, since we have full faith in your ability to run this program and copy down the answer. We strongly recommend solving this by hand, and only using a compiler to verify your answer.

**struct** Node {

int value;

Node \*next;

};

Node\* funcA(Node\* in) {

Node \*out = in;

**while** (out->next != **nullptr**) {

out = out->next;

}

funcB(in)->next = NULL;

**return** out;

}

Node\* funcB(Node\* in) {

**if** (in->next != **nullptr**) {

funcB(in->next)->next = in;

}

**return** in;

}

**Problem 3 (Runtime Analysis, 20%)**

In Big-Θ notation, analyze the running time of the following pieces of code/pseudo-code. Describe the running time as a function of the input size (here, n). You should **always** explain your work when solving mathematics problems.

**Part (a)**

**for** (int i = 0; i < n; i ++)

{ *// A is an array of integers of sufficient size*

**if** (A[i] == 0) {

**for** (int j = 0; j <= i; j++) {

**if** (A[i] == 0) {

**for** (int k = 0; k <= j; k++) {

A[i] = 1; *// this is no typo*

}

}

}

}

}

**Part (b)**

**for** (int i = 1; i < n; i \*= 2)

{

func(i);

}

void func(int x) {

**if** (x <= 1) **return**;

func(x-1);

}

**Part (c)**

*// This problem uses the same singly linked list Node structure you've seen a lot*

*// A is an array of integers of sufficiently large size*

Node \*head = **new** Node;

Node \*curr = head;

head->data = 0;

head->next = **nullptr**;

**for** (int i = 1; i < n; i++)

{

curr->next = **new** Node;

curr = curr->next;

curr->data = i;

curr->next = **nullptr**;

}

**for** (int i = 1; i <= n; i++) {

curr = head;

**while** (curr != **nullptr**) {

**if** (curr->data % i == 0) {

**for** (int j = 0; j < n; j++) {

A[j] ++;

}

}

curr = curr->next;

}

}

**Part (d)**

Notice that this code is very similar to what happens if you keep inserting into a vector.

double \*a = **new** double [3];

int size = 3;

**for** (int i = 0; i < n; i ++)

{

**if** (i == size)

{

int newsize = 3 \* size;

double \*b = **new** double [newsize];

**for** (int j = 0; j < size; j++) b[j] = a[j];

**delete** [] a;

a = b;

size = newsize;

}

a[i] = sqrt(i);

}

**Problem 4 (Recursion, 15%)**

Write a function that takes in a string, and outputs all possible permutations of the input, one per line. A permutation is a shuffling of the characters.

If the input is USC, then the output would be (in any order):

USC

UCS

SUC

SCU

CUS

CSU

If the input string is length n, then there should be exactly n! output strings. If the input string has no repeat letters, then there should be no repeat output strings.

If the input is CSC, then there will be repeat output (CCS shows up twice, once when one C is the first character, the other when the other C is the first character):

CSC

CCS

SCC

SCC

CSC

CCS

Here is the function you should implement:

void permutations(std::string in)

While we will only test your permutations function, you will probably want to write some main code to actually test it. Your submission should be in a file called permutations.cpp, and it should only contain your implementation of the function (don't include your main/testing code).

You may use loops and/or recursion in your implementation (Hint: you probably want to combine the two). You may not use the STL on this problem, other than the vector (if you so choose).

**Problem 5 (Cave Exploration, 15%)**

You are trying to explore a cave. In order to keep track of the way out, you carry a very long piece of string wrapped around a spool. When you take a step away from the direction you just came from, you let out a little more string. When you retrace a step and move back the way you came, you roll up a bit of string again. Your goal in this problem is to process a sequence of move and find out how much string is not rolled up.

Each input file will consist of several lines, and each line contains exactly one character: N, S, E, or W, describing the direction in which you took a step. Here is an example:

N

S

E

N

You can see that you are two steps away from the entrance, and that is also how much string is unrolled. Here is another example:

N

N

E

S

W

S

For this example, you end up back at the entrance, but the string will not be rolled up. For instance, if there is a big rock, the string is wrapped around the rock and there's no way to roll up the string now. There are six units of unrolled string - that's the number of steps you'd have to take to roll it up completely. Here is another example:

N

E

N

S

W

W

E

S

Here, you **do** end up at the entrance with the string rolled up, so the string distance is 0. (The easiest way to see this is to draw a little grid and simulate the moves and the trail of string.)

You should write a program that will process such sequences and always output a single integer, the number of units of string that are unrolled. We promise you that the input will always follow the given format. However, some of our test cases may be as long as somewhere around 2,000,000 steps, so you need to make sure that your solution is very fast (O(n)).

Contrary to what you may have come to expect from us, you cannot use any recursion in this problem (helper functions are always okay, as long as they remain non-recursive).

Your Makefile should be set up so that when we type make cave, it produces an executable named cave. This executable should take a single command-line parameter, which is the name of the input file to read from.

In this problem you are strongly encouraged to make use of the STL, in any manner you see fit. If you think carefully about the problem, and use the correct STL container, you will be able to solve this problem in well under 50 lines of code. If you don't, you will likely produce a mammoth solution that still does not solve all edge cases correctly.

**Problem 6 (Startup Companies, 30%)**

Startups these days are merging so fast, it's hard to keep track of who is in what company. Company A merges with Company B, and Company C merges with Company D, and then the two merged companies merge, and suddenly, employees from companies A and C find themselves being colleagues. This is getting sufficiently difficult to follow that we will have you write a Startup Tracker.

Here is how this will work. You have n students. Initially, each starts a startup company by himself/herself. Then, companies may merge. When you have a merge command, you will be given the numbers of two representative students, one from each company. Then, for each of those two students, you find the "biggest" company they are in, i.e., companies that are not subcompanies of any bigger company; let's call them A and B. Those two companies A and B are then merged into a new company; let's call it C. C will become the parent company of A and B.

Sometimes, companies also split up again. When we call split, we will again give you a representative student. Then, you find the biggest company that the student is in - let's call it A. As a result of the split, A should be completely removed, and the two companies that were at some point in the past merged to become A will now be individual companies without a parent again. (If the student is only in their own 1-person company, split does nothing.)

You will build a data structure that allows you to process a sequence of merges and splits, and answer queries about whether two students are in the same company.

To illustrate this, consider the following example with 5 students. After each command, we are showing you the structure of the nested companies with braces. The notation { {1}, { {2}, {3} } } means that we have a company with two subcompanies: the first subcompany is just student 1, while the second subcompany again has two subcompanies, one consisting of just student 2, the other consisting of just student 3.

merge (0,1) => { {0}, {1} }, {2}, {3}, {4}

merge (2,3) => { {0}, {1} }, { {2}, {3} }, {4}

merge (0,3) => { { {0}, {1} }, { {2}, {3} } }, {4}

split (2) => { {0}, {1} }, { {2},{3} }, {4}

split (2) => { {0}, {1} }, {2}, {3}, {4}

merge (2,4) => { {0}, {1} }, { {2}, {4} }, {3}

split (0) => {0}, {1}, { {2}, {4} }, {3}

A company is captured by the following

**struct** Company {

Company \*parent; *// the parent company, or nullptr if it has no parent*

Company \*merge1, \*merge2;

*/\* the subcompanies that were merged to obtain this company, or*

*nullptr if this is a 1-student company \*/*

Company ()

{ parent = **nullptr**; merge1 = **nullptr**; merge2 = **nullptr**; }

Company (Company \*m1, Company \*m2)

{ parent = **nullptr**; merge1 = m1; merge2 = m2; }

};

Your task is to implement the following data structure:

**class** **CompanyTracker** {

**public**:

CompanyTracker (int n);

*// initializes the tracker with n students and their 1-person companies*

~CompanyTracker (); *// deallocates all dynamically allocated memory*

void merge (int i, int j);

*/\* Merges the largest companies containing students i and j,*

*according to the rules described above.*

*That is, it generates a new Company object which will*

*become the parent company of the largest companies currently*

*containing students i and j.*

*If i and j already belong to the same company (or are the same),*

*merge doesn't do anything.*

*If either i or j are out of range, merge doesn't do anything. \*/*

void split (int i);

*/\* Splits the largest company that student i belongs to,*

*according to the rules described above.*

*That is, it deletes that Company object, and makes sure that*

*the two subcompanies have no parent afterwards.*

*If i's largest company is a 1-person company, split doesn't do anything.*

*If i is out of range, split doesn't do anything. \*/*

bool inSameCompany (int i, int j);

*/\* Returns whether students i and j are currently in the same company.*

*Returns true if i==j.*

*Returns false if i or j (or both) is out of range. \*/*

**private**:

int numCompanies; *// the number of companies you are tracking*

Company\*\* companies;

*/\* an array (allocated once in the constructor)*

*to keep pointers to all the 1-person companies.*

*Will not contain the merged companies. \*/*

*/\* Feel free to add private helper functions as you see fit.*

*In particular, you may want a function to find the largest company*

*that a student i is part of. \*/*

};

The signature above is given to you as a file company.h in the homework-resources repo. There, we also give you a bit of skeleton code that you are welcome to use to simplify your life a little bit. You may add private helper functions to CompanyTracker, but you cannot change the signatures of any of the functions we gave you - otherwise, we cannot test your solution, and that would be bad for your score. Each of your functions should run in no worse than O(n) time.

In order to test your CompanyTracker implementation, you will almost certainly want to write another file that includes it, and has a main function. You can then either read from a file, or hard-wire test-cases. Your test cases and main will not be graded, though; you should submit only your company.cpp file and company.h file (even if you didn't change it from our provided file).

You may not use any containers from the STL in this problem, other than the vector (if you so choose).

**Chocolate Problem (1 Chocolate Bar)**

We will from time to time assign "Chocolate Problems" on homeworks. These are entirely optional: solving them will not in any way affect your grade in this class. The reward is literally chocolate: if your solution is correct (or close enough), you will get the number of chocolate bars listed with the problem. If you have preferences/allergies, feel free to specify those with your submission. You can solve chocolate problems in teams, and will then share the chocolate as a team. The number of chocolate bars is a rough estimate of how difficult or time-consuming we think the problem is, but keep in mind that even a "1 chocolate bar" problem is more challenging than most standard homework problems. If you solved a chocolate problem, do not submit it via github - instead, just send all necessary files as an email attachment to Aaron Cote. Similarly, if you want to discuss ideas or get hints on chocolate problems, don't ask the course producers and TAs (they have enough to do helping your classmates with regular homework); instead, ask Aaron directly in person or on Piazza. Homework deadlines do not explicitly apply to chocolate problems, but please do not submit more than a couple days after the actual HW deadline.

Use recursion to write a program playing **perfect** Tic Tac Toe. Feel free to design any reasonable user interface (input and output). Your program should never lose, and win whenever given a chance.

(Note: Contrary to the [famous silly clip from the movie War Games](http://www.youtube.com/watch?v=F7qOV8xonfY), even if you let your program play itself, it will not learn that sometimes in life, one has to lose.)