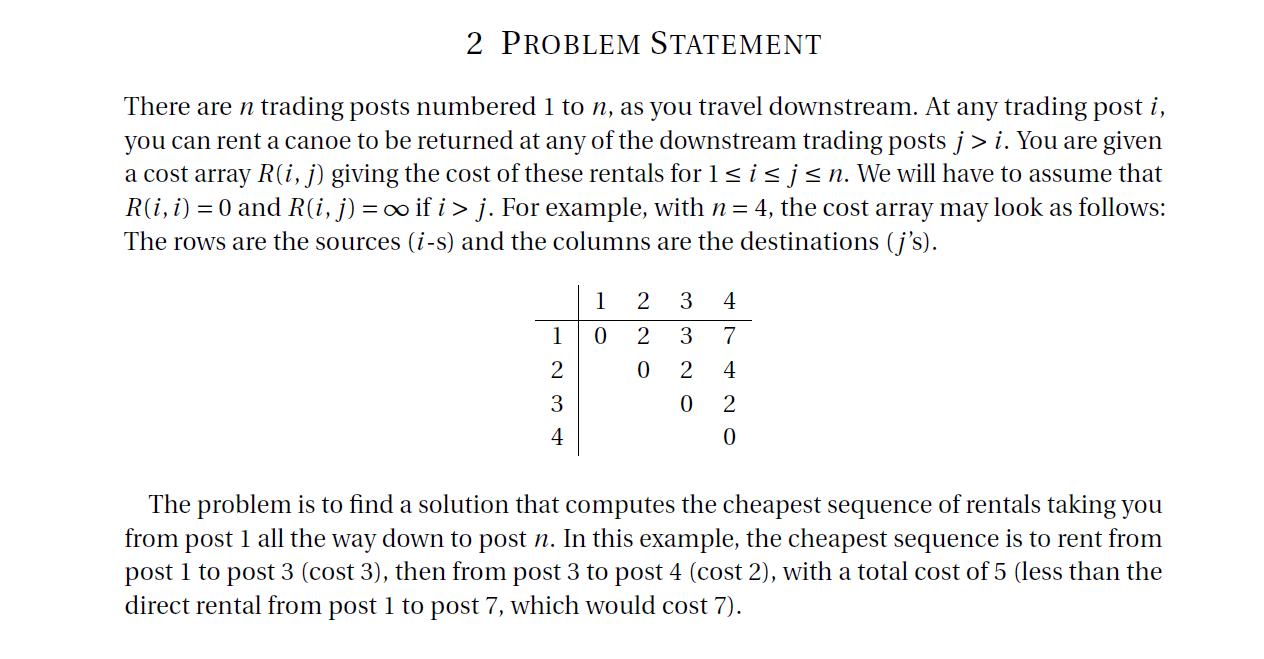
Aaron Chau, Tyler Brent, William Almond

TCSS 343

Spring 2016

Programming Project



Our group project sought to find and analyse the running times of three methods of programming solutions to the problem posted above. The first way we programmed was in a Brute force approach making the most thorough approach that we can considerting every possible solution. Secondly we attacked the problem with the divide and conquer method that divides down the problem into sub problems and solves the problem in that way. And lastly we used the dynamic programming solution that solves the last problem to consider and uses the stored values to solve the problem in the most efficient way of the three.

The labor on this project was divided into three parts, Aaron worked on the Brute Force and Divide and Conquer programming, Will Almond worked mostly on the Dynamic Programming and Documentation, and Tyler Brand worked on the Challenge question and the Testing.

The result of our analysis of each of the runtimes is as follows:

Brute Force:

The brute force algorithm loops through the size of the array twice (making O(n^2) time just for the loops), and compares each element up to k times. We know that k is at most equal to n so the runtime is O(n^3).

Divide and Conquer:

Would have run at a at the usual O(nlogn) time but because it had so many comparisons it ran in O(n^2) time.

Divide and conquer is a faster algorithm that

Dynamic programming:

Because it didn’t have to retrace it’s steps it ran near linear time. And generalizing this we know it runs in O(nm) time.

The results of our algorithms produced actual runtimes that were pretty scattered but had a general trend about them. We found that our brute force algroithm was more efficient than we expected and it wasn’t unusual to find times in the lower figures of zero milliseconds on all algorithms. However going into the upper numbers it becomes clear that the Brute Force algorithm is far slower than either of the two, and the Dynamic Programming algroithm is superior to them with its times growing the slowest as the n values grew.

Problems we had while coding this project mostly involved colaberation over the holiday weekend. There was much confusion especially at the end when everybody was pushing code to our code repository in the last few hours before the due date.

Another issue we had with this project was creating the Brute Force code. We developed the Brute Force code early on from our own intuition about programming, but it turned out we had codded it dynamically all along! So it was a struggle to turn out code that was not dynamic for the Brute Force code and it turned out that our Brute Force looks very similar to our Dynamic programming code.

The last complication of our program was deciphering the results. Perhaps it is because of our psudorandom values or the variable speeds of our laptops but our algorithms recorded many different values for each trial we ran. And we could only generally see trends in the data as the numbers and n values got large.

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.OutputStreamWriter;

**import** java.io.Writer;

**import** java.io.FileReader;

/\*\*

\* TCSS 343 Project

\* **@author** Aaron Chau

\* **@author** Will Almond

\* **@author** Tyler Brent

\* **@version** Spring 2016

\*/

**public** **class** **tcss343** {

/\*\*

\* Main method gets reads the file and drives each method.

\*/

**public** **static** **void** **main**(**String**[] args) {

**int**[][] **tradingPosts** = **null**;

// Create the sample files.

// createSampleFiles(new int[]{100, 200, 400, 600, 800});

// Read the input file

tradingPosts = *readFile*("input.txt"); // If this doesn't work you're not passing in "input.txt" as a param.

// printMatrix(tradingPosts);

*runCheapestAlgorithms*(tradingPosts);

// Read through sample files.

**for** (**int** **i** = 0; i < 5; i++) { // 5 sample inputs

tradingPosts = *readFile*("sample" + i + "input.txt");

*runCheapestAlgorithms*(tradingPosts);

}

}

/\*\*

\* Create the sample files given sample sizes.

\* **@param** sampleSizes an array of matrix sizes.

\*/

**public** **static** **void** **createSampleFiles**(**int**[] sampleSizes) {

**int**[][] **sample** = **null**;

// Create sample inputs

**for** (**int** **i** = 0; i < sampleSizes.length; i++) {

**try** (**Writer** **writer** = **new** BufferedWriter(**new** OutputStreamWriter(

**new** FileOutputStream("sample" + i + "input.txt"), "utf-8"))) {

sample = *tradingPostsFactory*(sampleSizes[i]);

*writePostToFile*(sample, writer);

**System**.***out***.println("Create file #" + i);

writer.close();

} **catch** (**IOException** **e**) {

e.printStackTrace();

}

}

}

/\*\*

\* Writes the trading post matrix to a file.

\* **@param** array the matrix

\* **@param** writer the writer to write to.

\* **@throws** IOException

\*/

**public** **static** **void** **writePostToFile**(**int**[][] array, **Writer** writer) **throws** **IOException** {

**for** (**int** **i** = 0; i < array.length; i++){

**StringBuilder** **sb** = **new** StringBuilder();

sb.append((array[i][0] == **Integer**.***MAX\_VALUE***) ? "NA" : array[i][0] + "");

**for** (**int** **j** = 1; j < array.length; j++) {

sb.append("\t"); // tab

sb.append(array[i][j] == **Integer**.***MAX\_VALUE*** ? "NA" : array[i][j]);

}

sb.append("\n"); // new line

writer.write(sb.toString());

}

}

/\*\*

\* Runs the brute, divide and conquer, and dynamic programming algorithms

\* given the trading post matrix.

\* **@param** tradingPosts a integer matrix.

\*/

**public** **static** **void** **runCheapestAlgorithms**(**int**[][] tradingPosts) {

**System**.***out***.println("Running length: " + tradingPosts.length);

// Brute force method

**long** **startingTime** = **System**.*currentTimeMillis*();

*brutePath*(tradingPosts);

**System**.***out***.println("Brute finished in: " + (**System**.*currentTimeMillis*() - startingTime) + "ms.");

// Divide and Conquer method

startingTime = **System**.*currentTimeMillis*();

*divideAndConquerPath*(tradingPosts);

**System**.***out***.println("Divide and conquer finished in: " + (**System**.*currentTimeMillis*() - startingTime) + "ms.");

// Dynamic method

startingTime = **System**.*currentTimeMillis*();

*dynamicPath*(tradingPosts);

**System**.***out***.println("Dynamic finished in: " + (**System**.*currentTimeMillis*() - startingTime) + "ms.");

**System**.***out***.println();

}

/\*\*

\* Brute force method of trading posts.

\*/

**public** **static** **void** **brutePath**(**int**[][] tradingPosts) {

// Number of trading posts.

**int** **size** = tradingPosts.length;

// Array to store paths.

**int**[] **path** = **new** **int**[size];

// Initialize new cost array with the price of traveling from the

// first post to the nth post.

**int**[] **cost** = **new** **int**[size];

**for** (**int** **i** = 0; i < cost.length; i++) {

cost[i] = tradingPosts[0][i];

path[i] = 0;

}

// Traveling to the first post cost $0. You're already there.

cost[0] = 0;

// For each other post.

**for** (**int** **j** = 1; j < size; j++) {

// Get the price

**int**[] **row** = tradingPosts[j];

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

// If the price at the previous post plus the price

// at the current post is less than the current cost at

// the current post then...

**if** (row[i] != 0 && row[i] != **Integer**.***MAX\_VALUE*** && row[i] + cost[i-1] < cost[i]) {

// Switch canoes

cost[i] = row[i] + cost[i-1];

// Track which post we switched to.

path[i] = j;

} // else stay on the same canoe.

}

} // O(n^2)

*printPath*(path, cost);

}

/\*\*

\* Use divide and conquer to solve the trading post problem.

\* **@param** tradingPosts A 2D array of integers that contains the cost to travel to each post.

\*/

**public** **static** **void** **divideAndConquerPath**(**int**[][] tradingPosts) {

// Create our cost and path array.

**int**[] **cost** = **new** **int**[tradingPosts.length];

**int**[] **path** = **new** **int**[tradingPosts.length];

// Call our helper method

*divideAndConquerPathHelper*(tradingPosts, 0, cost, path); // O(n^2)

// Print path

*printPath*(path, cost);

}

/\*\*

\* A helper method to do the recursion for the trading post problem.

\* **@param** tradingPosts A 2D array of integers that contains the cost to travel to each post.

\* **@param** index The index the algorithm is working on.

\* **@param** cost An array of current cost.

\* **@param** path An array of the current path.

\*/

**public** **static** **void** **divideAndConquerPathHelper**(**int**[][] tradingPosts, **int** index, **int**[] cost, **int**[] path) {

// If the index is less than the number of available post.

**if** (index < tradingPosts.length) {

// Get the prices to travel at our current post

**int**[] **row** = tradingPosts[index];

// GO through each post and assign the proper cost and path.

**for** (**int** **i** = index+1; i < tradingPosts.length; i++) {

**if** (index == 0) {

cost[i] = row[i];

path[i] = index;

} **else** **if** (row[i] + cost[i-1] < cost[i]) {

cost[i] = row[i] + cost[i-1];

path[i] = index;

}

}

// Call the helper method on the next trading post.

*divideAndConquerPathHelper*(tradingPosts, index+1, cost, path);

}

}

/\*\*

\* This is the algorithm to find the solution to the cheapest path.

\* **@param** tradingPosts

\* **@return** path array

\*/

**public** **static** **void** **dynamicPath**(**int**[][] tradingPosts) {

// Number of trading posts.

**int** **size** = tradingPosts.length;

// Array to store paths.

**int**[] **path** = **new** **int**[size];

// Initialize new cost array with the price of traveling from the

// first post to the nth post.

**int**[] **cost** = **new** **int**[size];

**for** (**int** **i** = 0; i < cost.length; i++) {

cost[i] = tradingPosts[0][i];

path[i] = 0;

}

// Traveling to the first post cost $0. You're already there.

cost[0] = 0;

// For each other post.

**for** (**int** **j** = 1; j < size; j++) {

// Get the price

**int**[] **row** = tradingPosts[j];

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

// If the price at the previous post plus the price

// at the current post is less than the current cost at

// the current post then...

**if** (row[i] + cost[i-1] < cost[i]) {

// Switch canoes

cost[i] = row[i] + cost[i-1];

// Track which post we switched to.

path[i] = j;

} // else stay on the same canoe.

}

} // O(n m)

*printPath*(path, cost);

}

/\*\*

\* Read a file and return a n by n matrix with the trading post prices.

\* NA values in the text file will use Integer.MAX\_VALUE.

\* **@param** file the file name as a string.

\* **@return** a trading post matrix

\*/

**public** **static** **int**[][] **readFile**(**String** file) {

// Get file to read

**FileReader** **fileReader**;

**BufferedReader** **buffReader**;

**int**[][] **tradingPosts** = **null**;

**try** {

fileReader = **new** FileReader(file);

buffReader = **new** BufferedReader(fileReader);

// Get the first line

**String** **line** = buffReader.readLine();

// Find out how many post we have.

**final** **int** **size** = line.split("\t").length;

tradingPosts = **new** **int**[size][size];

// Add the cost values of each trading post

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

**String**[] **values** = line.split("\t");

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

tradingPosts[i][j] = values[j].equals("NA") ?

**Integer**.***MAX\_VALUE*** : **Integer**.*parseInt*(values[j]);

}

line = buffReader.readLine();

}

buffReader.close();

fileReader.close();

} **catch** (**IOException** **e**) {

e.printStackTrace();

}

**return** tradingPosts;

}

/\*\*

\* Create a sudo random matrix that represents prices for the trading post problem.

\* **@param** size is the size of the n by n matrix.

\* **@return** a n by n matrix.

\*/

**public** **static** **int**[][] **tradingPostsFactory**(**int** size) {

**int**[][] **array** = **new** **int**[size][size];

**for** (**int** **row** = 0; row < array.length; row++) {

**for** (**int** **col** = 0; col < array.length; col++) {

**if** (row == col) {

array[row][col] = 0;

} **else** **if** (col < row) {

array[row][col] = **Integer**.***MAX\_VALUE***;

} **else** {

array[row][col] = array[row][col-1] + (**int**) **Math**.*ceil*(**Math**.*random*() \* 5);

}

}

}

**return** array;

}

/\*\*

\* Print the path and the cost given

\* **@param** path the path taken in an array

\* **@param** cost the cost along the way

\*/

**public** **static** **void** **printPath**(**int**[] path, **int**[] cost) {

**System**.***out***.print("The cheapest path is to travel from posts ");

**System**.***out***.print("1-");

**for** (**int** **i** = 1; i < path.length; i++) {

**if** (path[i-1] != path[i]) {

**System**.***out***.print(i);

**System**.***out***.print(", " + i + "-");

}

}

**System**.***out***.println(path.length + " and it cost $" + cost[cost.length-1] + ".");

}

/\*\*

\* Prints the values of an array.

\* **@param** arr the array that needs to be printed.

\*/

**public** **static** **void** **printArray**(**int**[] arr) {

**System**.***out***.print("[" + arr[0]);

**for** (**int** **i** = 1; i < arr.length; i++) {

**System**.***out***.print(", " + arr[i]);

}

**System**.***out***.println("]");

}

/\*\*

\* Prints a n by n matrix

\* **@param** array the matrix.

\*/

**public** **static** **void** **printMatrix**(**int**[][] array) {

**System**.***out***.println("[");

**for** (**int** **i** = 0; i < array.length; i++) {

**System**.***out***.print("[" + *num2String*(array[i][0]));

**for** (**int** **j** = 1; j < array.length; j++){

**System**.***out***.print(", " + *num2String*(array[i][j]));

}

**System**.***out***.println("]");

}

**System**.***out***.println("]\n");

}

/\*\*

\* Creates a string number and gives the number spaced paddings to the left.

\* **@param** number the number you want to be padded.

\* **@return** a string value of the number being printed with padding.

\*/

**public** **static** **String** **num2String**(**int** number) {

**int** **length** = (**int**) (**Math**.*log10*(number) + 1);

**int** **maxPadding** = (**int**) (**Math**.*log10*(**Integer**.***MAX\_VALUE***) + 1);

**int** **padding** = maxPadding - length;

**StringBuilder** **sb** = **new** StringBuilder();

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

sb.insert(0, " ");

}

sb.append(number);

**return** sb.toString();

}

}