Akshay Sud

CS 146 S7

Table of Contents

1 – Cover Page

2 – Table of Contents

3 – How to Run

4 – Overall Design

5 – BST Functionality

9 – Quicksort Functionality

11 – Bucket Sort Functionality

12 – Gradient Descent Functionality

13 – Class/Function Overviews

13 – pageRank

15 – Node

16 – bst

21 – Quicksort

22 – Bucket Sort

24 – Gradient Descent

28 – Spider

29 – SpiderLeg

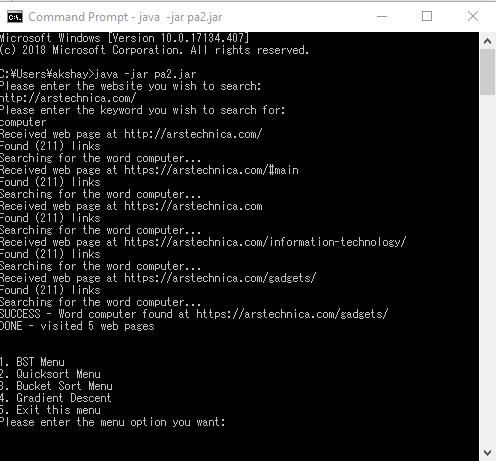
30 – Problems Encountered & Lessons Learned

How to run

The first thing to do once you have unzipped my program folder is to identify which folder you have extracted the pa1.jar file to. In my case here, it’s to C:/Users/akshay

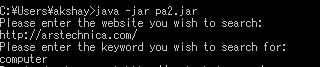
In order to run my program, you must first enter command line and navigate to the directory in which the jar file is located. Then, you must type into the command line “java –jar pa2.jar”, which will run the jar file as shown below.

In my test case here, I used the website <https://arstechnica.com/> and keyword to search for “computer”.



Overall Design

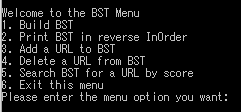
The way I designed my program was that first, the web crawler runs and asks you which website you wish to search, and which key word you wish to search the site for. After it comes up with all the URLs it has stored, I take the first 30 and pop them into an array.



Once it’s done that, it gives you the main menu where you can select the BST (Binary Search Tree) submenu, Quicksort submenu, Bucket Sort submenu, and Gradient Descent submenu.

The first 3 submenus; BST, Quicksort, and Bucket Sort are required for this program, but the addition of the Gradient Descent submenu is something that I decided to add into the program. It’s function will be discussed later in the documentation in the Gradient Descent section.

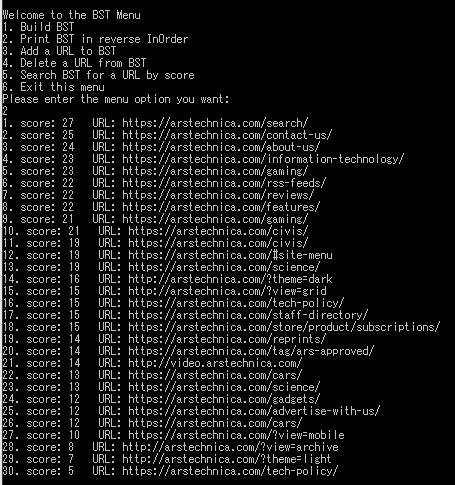
The BST menu, looking like this:



Gives you numerous options for interacting with the Binary Search Tree that was created at the beginning of the program. The first option allows you to build the BST, by inserting the 30 URLs into the tree by their associated rank values, which simulates the overall score of each URL.

**BST**

You can print the BST once it’s been built in reverse order, and the reason that it’s in reverse order is that my printing function is a modified InOrder print, but in reverse order. This makes it so that the top ranks are at the top of the list rather than at the bottom.



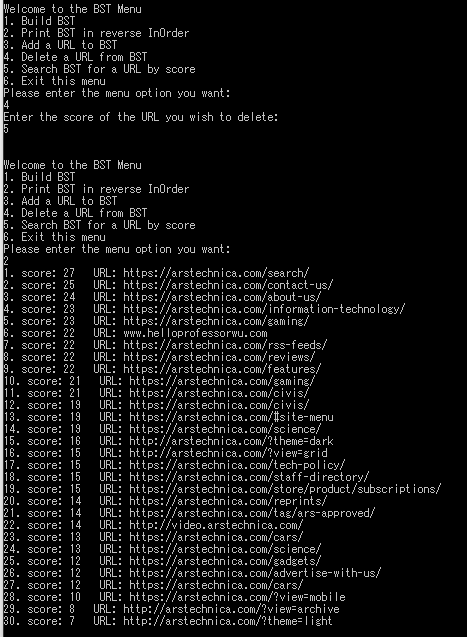
You can add a URL to the BST as well



If you look at #6, you can see the URL that I added into the list

We can also delete URLs from the BST like so

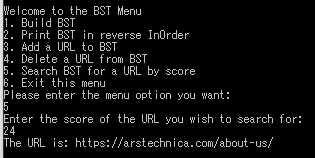
I want to delete the URL with score 5



As you can see, the URL is now gone from the list

If I want to search for a URL, I can do so by using its score

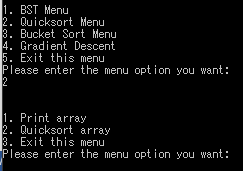
I want the URL with score 24, so I do this



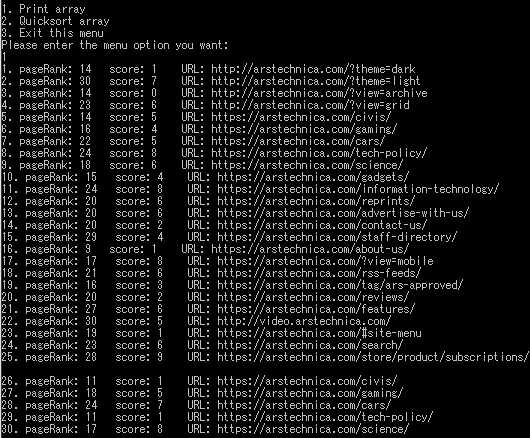
If you look at the previous screencap, you can see that this is the right URL

**QuickSort**

The second menu is for Quicksort

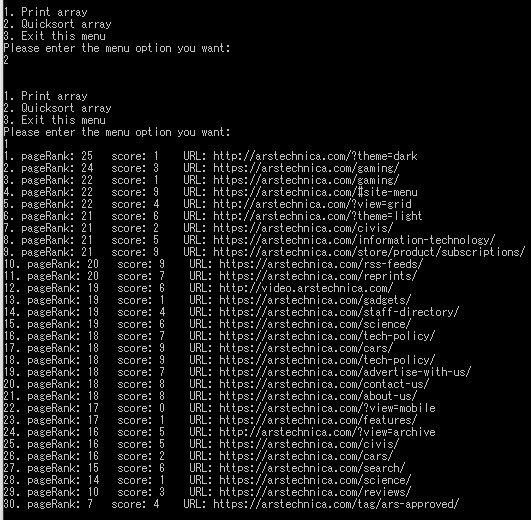


The first thing that you should do for this menu is to print the unsorted array.



You can see the values associated with each URL. The pageRank here is the total score of the URL with all 4 factors involved, and the score here represents the value of how much was paid to Google.

Once you get back to the Quicksort menu, you should Quicksort the array and print the newly sorted array like so

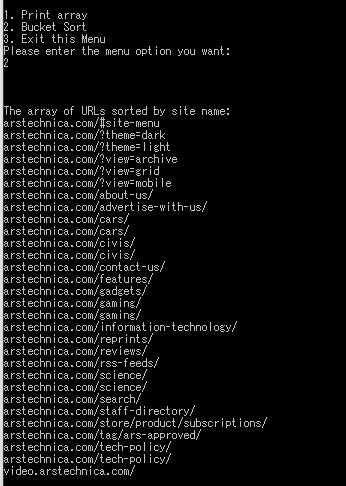


As you can see, the URLs have been sorted in order of highest score to lowest score, while still showing the score associated with how much the domain paid to Google

**Bucket Sort**

the Bucket Sort menu, the point of the print array function is to see that the URLs are not already sorted in alphabetical order.

When you Bucket Sort the array, it automatically prints the list of domains in alphabetical order by taking out the ” https://” substring from each URL, then sorting alphabetically using 26 buckets, each one corresponding to a letter in the alphabet from A-Z.



In my test case, I searched “arstechnica.com” for the word “computer”, so the vast majority of the URLs I was returned were to the same domain, so I couldn’t test this much, as only 2 buckets were utilized. However, the function works so if different URLs were returned by the web crawler, it would still work as intended.

**Gradient Descent**

The last menu option in the program is the Gradient Descent function.

To understand this function, you must first understand 2 important values associated with each URL, which are

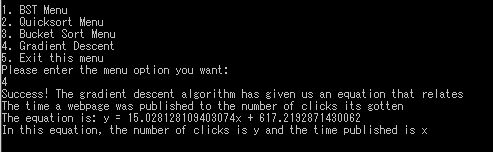
double time

double clicks

The variable of type double, time, is a randomly generated double between 0 and 24, which represents what time of day the URL was published. The variable of type double clicks however, is a randomly generated double between 1 and 1000, which represents how many thousands (or millions) of clicks the URL received after it was published.

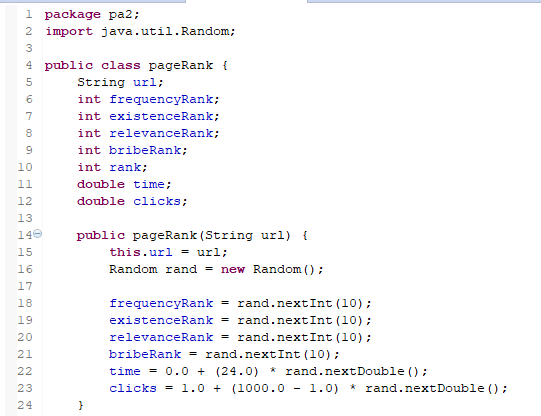
The point of the Gradient Descent function is to use the Machine Learning algorithm for Batch Gradient Descent to do linear regression on each data pair (time and clicks associated with each URL) to try and infer a linear relationship between the two variables, where clicks received by a page is a function of the time the page was posted. This is to represent a possible real-world problem a search indexing company like Google has probably dealt with and apply some basic Machine Learning to attempt to find a (very simple) solution.

When you select the Gradient Descent menu option, the Batch Gradient Descent function is run and puts the first 750 URLs the web crawler stored into an array, each with their own randomly generated time and clicks. The algorithm then attempts to find the linear relationship between the two and prints it before taking you back to the main menu.



To cite where I learned the algorithm, it was through a Machine Learning online lecture on Coursera taught by Professor Andrew Ng from Stanford University.

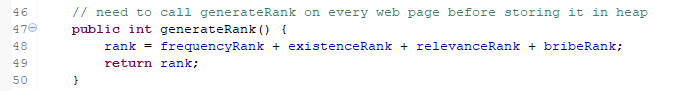
Class Overview – pageRank



The pageRank class creates a pageRank object for each individual URL, and assigns it 6 different values randomly. The first 4:

* frequencyRank
* existenceRank
* relevanceRank
* bribeRank

Are all added together to get the total rank for the page, which is generated for the pageRank when the function generateRank() is called.



Explained earlier on page 12 are the two randomly generated variables time and clicks. Time represents the time of day the page was published, and clicks represents how many clicks the page has gotten through the search company (Google), and in the Gradient Descent function, the program attempts to use the Gradient Descent algorithm to do linear regression on each data pair (URL) to find a relationship between the two.

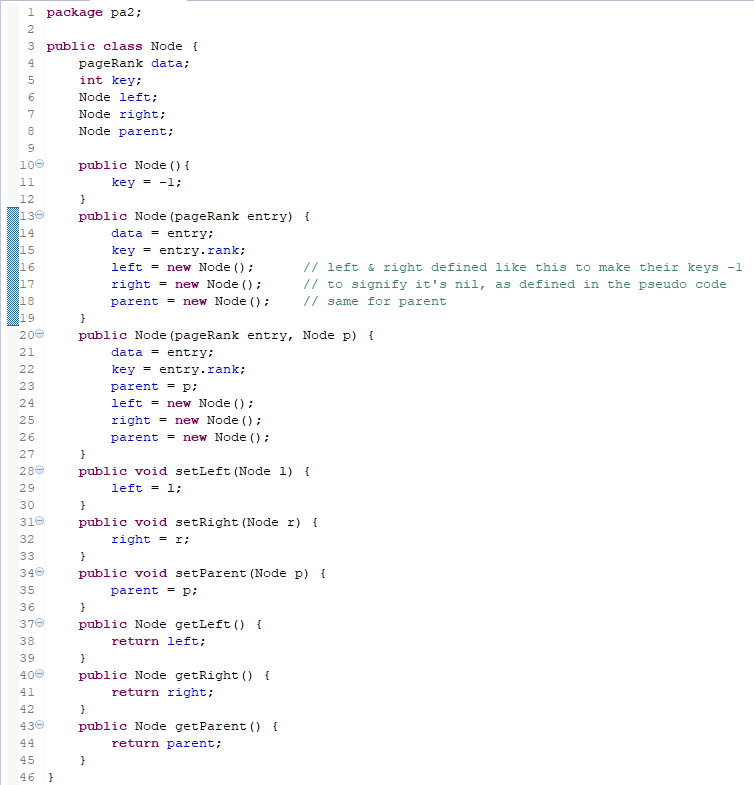
Time is generated as a random double between 0 and 24

Clicks is generated as a random double between 1 and 1000, and represents either thousands or millions of clicks received by a page, depending on the user.

The rank that can be accessed and edited by the user is the bribeRank, which corresponds to the rank assigned by companies such as Google based on how much a domain pays to the company in order to boost its rank on search results.

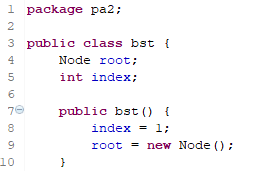
The rank is intrinsic to the URL once each pageRank object is created, but can be re-calculated using the generateRank function in case of bribeRank for the object being increased by the user.

Class Overview – Node



The Node class is simple, as its used in the Binary Search Tree function to store the individual pageRank objects, which each represent a different URL. The Node class has a pageRank assigned to it, a key (the total score for the page), and left, right and parent pointers for tree functions.

Class Overview – bst

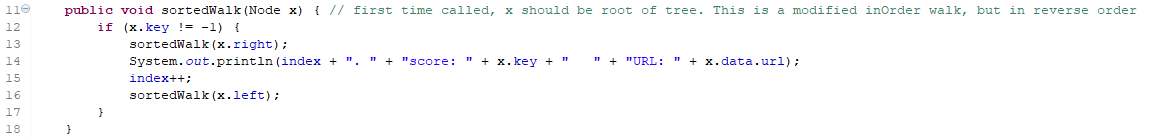


The bst class itself is simple, as each bst just contains a root. The index however, is something I used for my reverse InOrder printing function, and it will soon be apparent why it’s defined this way.

The class has 7 main functions

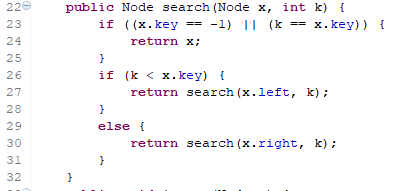
* sortedWalk
* search
* insert
* minimum
* maximum
* transplant
* delete

The first is sortedWalk



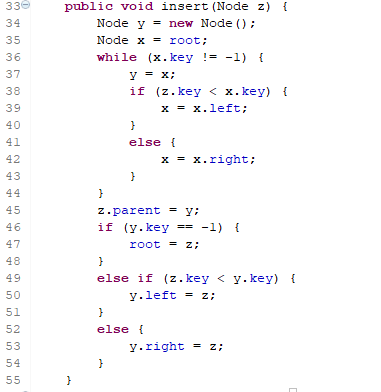
It’s a reversed InOrder walk, as I go through the right side of the tree before the left side, and increases the index counter each time its run, which prints the nodes rank in order.

The second is the search function, which is fairly simple

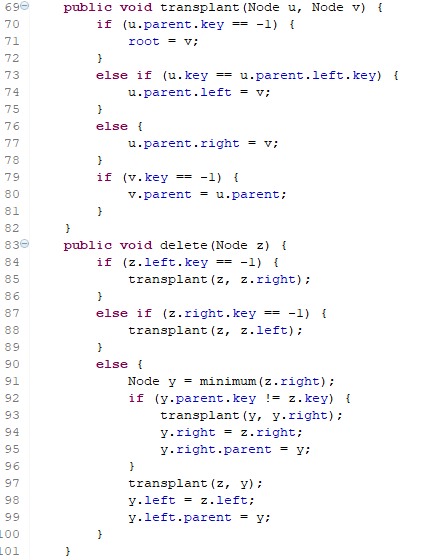


It’s mostly the same as the pseudo code in the textbook, and if the Node you’re looking for is less than the Node you’re looking at, you go left to a smaller Node. If it’s bigger, you go right to a larger Node until you find the right URL’s total score.

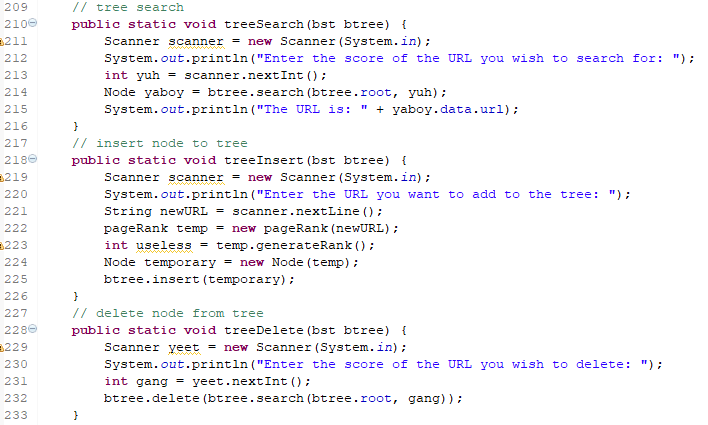
The next function, insert, is also really similar to the textbook pseudo code.



Transplant and Delete are also basically the same as the pseudo code, I just run it through the tree’s class function instead of an external function where I pass the tree in for simplicity’s sake.

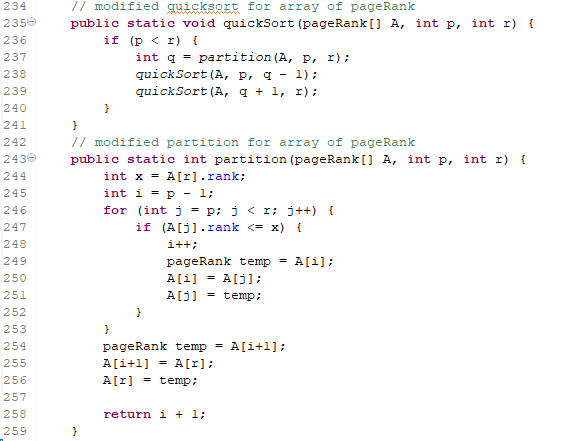


In the main function, there are a couple of helper functions that exist in order to simplify the code for readability’s sake



These are each called when the corresponding menu option is selected and shows how each function is called in the main.

Function Overview – QuickSort

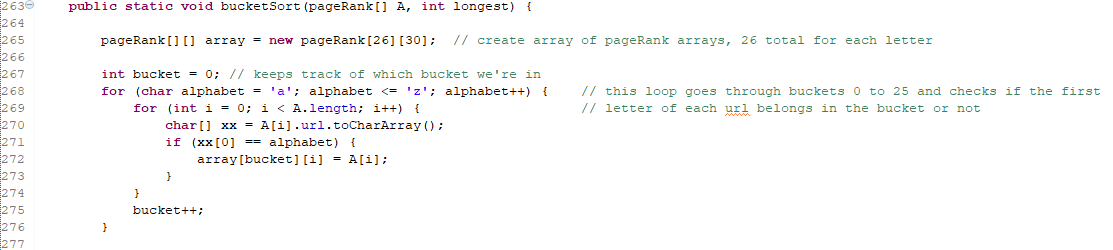


My QuickSort and partition functions are in the main.

They are both extremely similar to the pseudo code in the textbook, except the array passed in is an array of pageRank objects, each with a different total page score. The biggest changes were to the partition function, where I had to make a temporary pageRank to do swaps, and looked at each page’s rank to order them all.

Function Overview – Bucket Sort

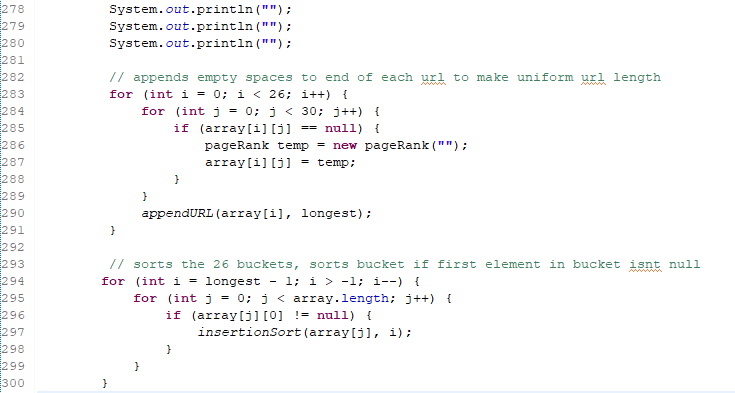
The code for Bucket Sort is quite long, so I’ll go over it here piece by piece



When Bucket Sort is called, it’s passed in an integer longest which represents the longest string in the array, for use later in Bucket Sort.

Bucket Sort is in the main, and is called using an array of pageRank objects. The first thing that the program does is create a 2 dimensional array of type pageRank, with 26 rows and 30 columns. This is done to deal with the (worst) case of all the values in one bucket, and one bucket created for each letter of the alphabet from A-Z.

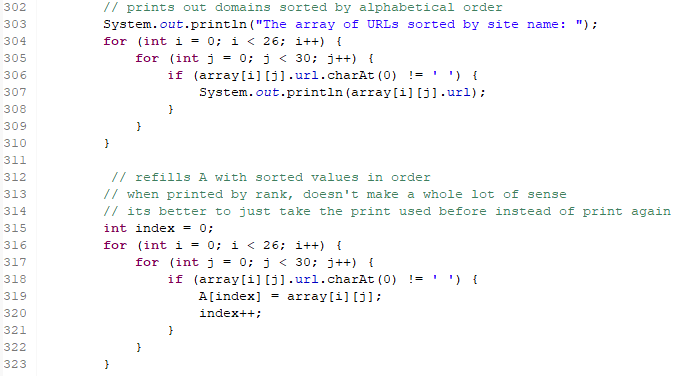
A nested for loop is used here to iterate through the different buckets and put each URL in the right bucket.



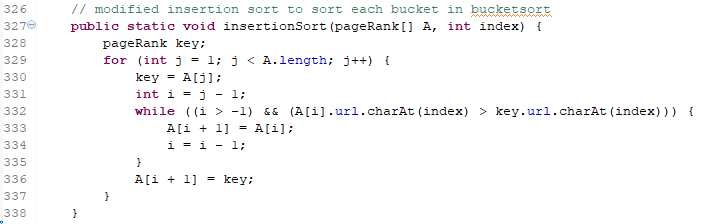
The next nested for loop goes through the 2d array of URLs and appends extra spaces onto each URL if it isn’t the same length as the longest string. This is done in order for radix sort to work on each URL, as they all need to be Strings of the same length to be sorted from the last character to the first.

The loop after that iterates through the buckets in order to insertion sort each bucket. The special part of this part of the function is that it doesn’t bother insertion sorting an empty bucket.

Next, we have the loops that print the domains in alphabetical order now that they’ve been insertion sorted from the last letter to the first in each bucket then combined again, thus making them Bucket Sorted.

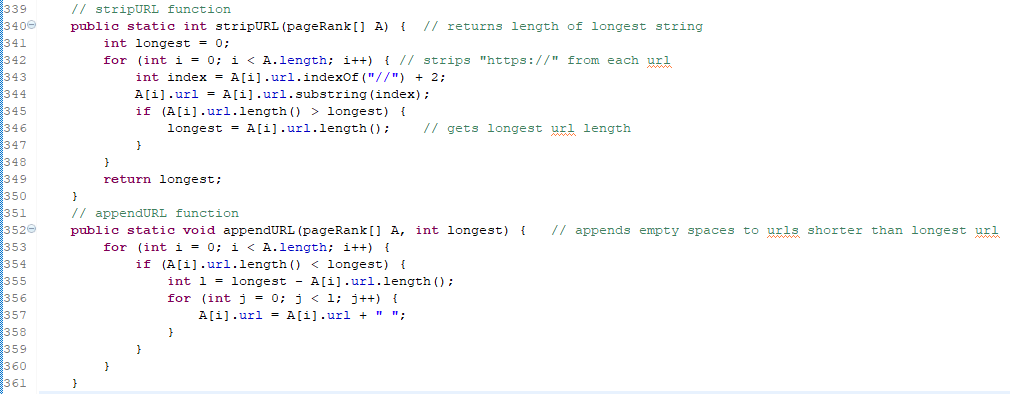


Once they’ve been printed, they get pushed back into the original array of URLs, skipping empty URL’s by checking the first character in the String. If the first character is an empty space, it doesn’t do anything to it, but if it’s anything else, it puts it in the array.



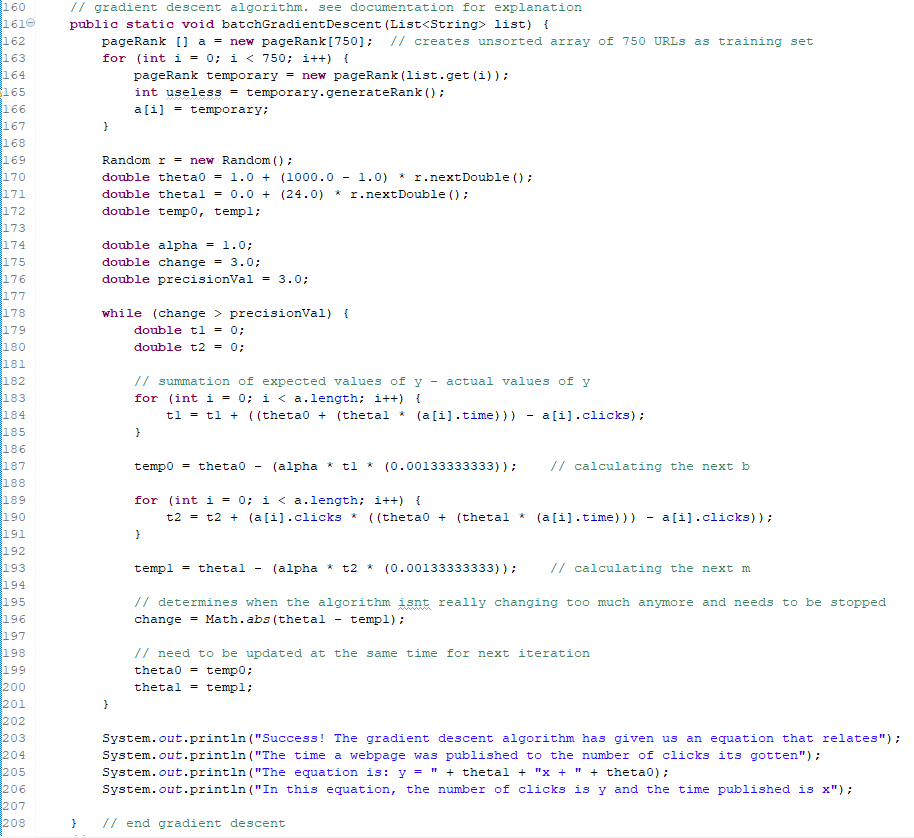
Insertion sort is used here for each bucket, and it’s been slightly modified from the pseudo code in the textbook. For the key, I made an entirely new pageRank object, and the integer index that gets passed in actually makes it so that each instance of insertion sort only sorts by character of the String given by the integer index. This is done from outside in a loop, sorting each bucket by the last letter of the URL to the first.

I have 2 more helper functions I made for the Bucket Sort algorithm to do some tasks



The stripURL function is used outside of the Bucket Sort function in the Bucket Sort menu in order to strip the “ https:// “ from each URL, to keep them all from being put into the h bucket. It also serves to find the longest URL of the array, and returns it for use in the Bucket Sort function.

Function Overview – Gradient Descent



The Gradient Descent function takes in the whole list of URLs returned by the web crawler because it needs to use as many as possible for the training set. This is to have as many learning cases as it can to predict the relationship between the time and clicks variables as accurately as possible. This is impossible here, as the variables are all generated randomly. However, with actual data like Google has access to, it could actually find a possible correlation, in theory.

I first generate random initial values for the first theta0 and theta1, where they correspond to

Y = mx + b

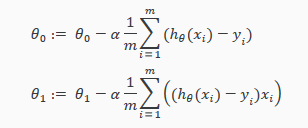
Theta0 is b (y intercept)

Theta1 is m (slope)

Here, alpha represents the learning rate, and my condition for the while loop is until “convergence”. However, I can’t really get a convergence from my randomly created data set, so I got as close as I could before the program started to get bugged.

Basically, the while loop ends as soon as the absolute value difference between the previously calculated value of theta1 and the new calculated value of theta1 are within 3 units of each other, which is as close to convergence as my program and likely my data set can get me before diverging to infinity.

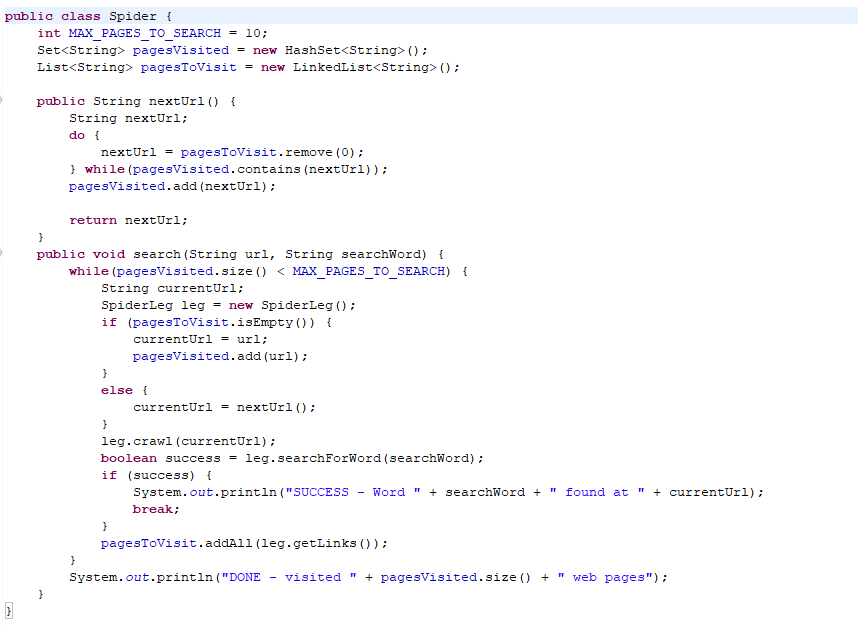
Inside the while loop, new theta0 and theta1 values are calculated using previous values, 1/750 (750 is # of URLs m), the summation of the differences between expected y (given from the initial theta0 and theta1 equation) and the actual y, which was the clicks for that particular URL. For theta1, it’s a little different, with that number being multiplied by the actual x value for that URL, which is the time variable.



The values are then calculated over and over again, until it can get 2 consecutive predictions where it has a certain amount of certainty in its prediction, essentially that it was within 3 units of its previous prediction. This is because with each trial, the Gradient Descent function learns from its previous trial, and tries to find another closer approximation for the linear relationship between the time and clicks.

This can be visualized if you imagine a golf course, where you want to find your way to the lowest level in the hilly course. You would take a step in a direction lower to you, then another step, and another, until you reach the first lowest location you can. Then, you go back up to a different hill and try to get down to the lowest point you can step by step the same way. This is what the Gradient Descent function is doing, and tries to find the lowest location (best approximation for the linear relationship) that it can using Linear Regression on the data set.

Class overview – Spider



The Spider class is what is first called with the url of the website you intend to search, as well as the keyword you wish to search for. Using these two strings, the class then uses the SpiderLeg class to actually crawl through the site and store the URLs into the Spider class’s List of Strings, where I extract 30 URLs from in most of the functions, and 750 from for Gradient Descent.

The code for the WebCrawler was all provided by

<http://www.netinstructions.com/how-to-make-a-simple-web-crawler-in-java/>

from author Steven (last name unknown). Thank you Steven!

Class overview – SpiderLeg



Like my Spider class, this was also provided by Steven from the link

<http://www.netinstructions.com/how-to-make-a-simple-web-crawler-in-java/>

The SpiderLeg class pretends to be a real user and makes requests to the site for URLs on the website. It crawls through the initial page for the keyword, then follows through to other webpages linked from the main page.

The SpiderLeg class actually interacts with the website, and in order to do this, Steven used the jsoup library to simplify things in the code and make it very easy to actually interact with the website.

Thanks again Steven, and jSoup!

Problems Encountered & Lessons Learned

The biggest problems I encountered during this project were with the Bucket Sort and with the Gradient Descent function. The Binary Search Tree was fairly easy for me because I understood the tree itself very well, and could easily follow the pseudo code. Modified Quicksort was also fairly easy, as I had already converted the Quicksort code to Java code previously, and had to just change it a little to work with sorting by the page’s overall score, rather than by an integer value in an array of integers.

However, Bucket Sort was incredibly difficult for me because of how many moving pieces were involved. It wasn’t just sorting the array, because I first had to do things like cutting out parts of each URL, find a way to append empty spaces onto the URL, then separate the URLs into buckets before sorting them somehow. My solutions for how I did almost anything for Bucket Sort was probably incredibly space and time inefficient compared to how it could have been done, but the good thing is that during the process, I learned things that would help me make a more efficient function to do the same thing again if I had to. I could have theoretically done all of the String cutting and appending as well as finding the longest string in one nested for loop, instead of having to do 3 separate ones. Even though doing 3 separate loops doesn’t affect time complexity, it’s clearly still unnecessary instructions that could have been avoided.

I also learned from Bucket Sort how to iterate selectively through 2D arrays, something I didn’t’ expect to have to do anytime soon. It was helpful though because of how often 2D arrays are used, and how often they’re useful, like this time.

A problem I encountered when doing Bucket Sort was trying to make an array of Lists instead of a 2D array. I originally wanted to follow the textbook pseudo code and use Lists because not every bucket had to have a determined number of spots for usability in a bigger set, but I couldn’t figure out how to sort a linked list letter by letter from the last to first without having to make my own Linked List class, which was something I figured was probably unnecessary.

In Gradient Descent, I learned a lot about Gradient Descent and linear regression by implementing both of them in this programming assignment. I had only really understood Gradient Descent and how you could use it for linear regression loosely in theory, but after having to implement them, especially on a time restraint, was very enlightening and taught me a lot about how both of them work. I had to learn Gradient Descent again at a higher level in order to actually implement it in Java, as opposed to a more ML-friendly environment like in Octave.