# Project 1: Google It

Members: Jason Durkee, Ryan Schick, Yash Chulki, Ty Rozell

## Project Description:

In this project, we are constructing C++ code that reads in a file with a collection of different documents, deletes certain stopwords, Porter stems the documents, and determines, based off of a user input query, which documents are most similar to the query. We will be determining the similarities using term frequency-inverse document frequency and cosine similarity and outputting the top 5 similar documents. The data read from the file is each document's identification number, title, author, and abstract inorder to perform the actions stated above. Information is passed between functions and handled separately in each group members function. This project was completed with a team of 4 that worked on independent IDEs and then coming together to put their work into one working solution.

Through hours of blood, sweat, and tears we bring you the completed version of Bing IT. This state of the art, miniature search engine is dedicated to providing the user with the best results possible.

Est. October 4, 2019

## Part 1: Storing the documents (Jason Durkee)

The very first part of this project starts with storing the documents. My first task was to create a class called "docStorage" and fill in the small components. In the docStorage class I created 4 string vectors, a basic constructor, and a constructor that sets up each string vector. There is a store and a print document function also in the class. When the storeDocs function is called, it asks for the document name. From there it creates a temp pointer object for the class and uses an if statement to start reading in from ".I". It then adds ID to the vector for the ID in the temp object. It repeats this for every title, author and abstract for the rest of the document file. Once everything has been read in, it pushes back the vector that holds all the documents and resets the temp pointer.

The print function takes in the docVector and then prints out each part of the first document in for loops until everything that is needed has been output to the screen.

# Part 2: Storing the stopwords and removing punctuation (Jason Durkee)

Once part 1 was implemented, storing stopwords was the next step. In the main, I created a vector to store the stopwords and then I created a store stopwords, print stopwords and a comparing function inside the docStorage class. I called the stopWordDocs function, which stores the stopwords, and it takes in the stopwords vector. From there I used the same code to make sure the correct file was used and then proceed to read each word in the file and store each word in the stopwords vector. The printStopwords function takes in the stopwords vector, counts the number of words in the vector, and outputs the number of currently known stopwords.

I created a remTitle function which removes the title of the document from the abstract. It does this by taking in the abstract and title of the document and erases the part of the abstract vector that contained the title vector.

I also created a remPunct function that takes in the abstract and uses the function “ispunct()” to detect if the character is a punctuation mark. If it is a punctuation mark then it deletes it from the vector. This function also removes digits that are throughout files using the same method as the punctuation removal.

# Part 3: Performing Porter stemming (Ty Rozell)

In this program, we employed a Porter stemming function. The purpose of the Porter stemmer is to allow for the contents of the documents to be easily compared to one another for similarity. What the Porter stemmer does is it truncates words based on a specific set of rules. It removes unnecessary endings on words and leaves behind the stem of the word. This allows for words with different suffixes to be converted to their respective stems and makes for easier comparisons between documents. For example, the words “carasses” and “caressed” have different suffixes but have the same stem word. The stemmer function converts these words to “caress”.

The Porter stemming function can be called with either the vector of document objects or the query string vector as parameters being passed by reference. Rather than writing two independent functions to handle the two types of data parameters, function overloading was used. The document vector version of the function contains the main stemming functionality. In the query version, a temporary vector of documents is created and the query is stored in this new vector as an abstract. The document vector function version is then called with the temporary document vector as the parameter. Doing this allows for code reuse, rather than creating an entire stemming function for the query alone.

Once inside the stemming function, the program will loop through and check for and remove the appropriate suffixes from the words in each of the documents' abstracts. Abstracts are vectors of strings. Each step in the stemming process is separated into its own function. This allows for the word to be returned at any time during the step's execution, which insures that a word will only be stemmed once per step. Another job of the stemming function is to output the first document's abstract before and after it has been stemmed. It also outputs the number of unique words in the stemmed and unstemmed abstract. This is done by removing all of the duplicate words in the abstract and seeing how many of the unique words remain. The output feature is skipped when the function is called with the query instead of the document vector. After this is complete, the function stems the rest of the abstracts that are stored in the document vector.

# Part 4: TF-IDF for Each Word in Document (Ryan Schick)

This part of the project starts to incorporate everything together in order to get a working "Google IT". The tf-idf of words in all the documents means the word's term frequency is the amount of times the current word shows in the current document, the word's inverse document frequency is a score that is based on the equation $\log_e(N/n_t)$ in order to compare documents in cosine similarity, and term frequency-inverse document frequency(tf-idf) is the word's term frequency multiplied by its inverse document frequency. Typically this score is used in search engines to rank a document's relevance given a query and is used to evaluate how important a word is to a document in a collection.

The TF-IDF portion of the code is made in a termFrequency class that holds multiple vectors of vectors that contain doubles and strings in order to store all the words in the abstracts of the documents along with their associated tf, idf, and TF-IDF score. This helps print out the correct output which is produced in a print function that uses the setw() function in order to make sure the spaces are consistent through all outputs. The class first stores the abstract of all documents and as it's added to the abstract vector, the function calculates and pushes the TF of the current word and deletes all other instances of the word and I did this solely for the purpose that my stored abstract always only has one instance of the word for output and that it's storing and count in one single function. IDF is the next calculation needed in order to move forward with the tf-IDF score, so the setidfAbstract() loops through the current document and compares it to the other documents and increases an int variable every time there's an instance of the current word in another document and breaks when it's found so that it's not counted twice in the same document. After calculating the idf, the

number is pushed to an idfAbstract vector which then leads to the final calculation of the tfIDF of the abstract words. This function loops through both tf and idf vectors and multiplies the elements and pushes the product to the tfIDFAbstract.

The tf-IDF of the query implements most of the code above; loops through the query and compares it to the all the documents and pushes the result to a vector. The printTFQuery() prints out any query words that didn't show up in any of the documents and also deletes the word from the query vector and the zero in tfidfQuery. I do this so that neither vector holds onto information that is no longer needed and that both vectors are still aligned word with associated number.

# Part 5: Storing the Query (Ryan Schick)

The major functionality of this portion of the project is to prompt the user for a query of words or sentence to search the file and store, parse, porterstem, the query and remove stop words and punctuation in the query. This is used in calculating tf-idf of the query and the cosine similarity as it's apart of dot product with the tf-idf for each word in each document. It's the backbone of the purpose for the project as it lets the user give keywords to search through all the documents to see which documents are the most similar to the words in the query, actually making it like a google query.

This portion of the project required me to create a Query class so that the data could be stored easier and create functions specifically catering to any Query object. The setQuery function doesn't take a parameter and prompts the user to input a list of words to be entered into the query to find the related documents. To function uses a toAllLowerCase function inorder to keep the words consistent and easier to compare to the document abstract. By storing the inputted query into a private string, it makes it each character readily accessible to loop through and separate all the words by spaces and pushes it to a public vector which sorts the query in alphabetical order. By having the query in a vector, in sets up great success in the program inorder to compare the contents of the query to the abstracts in all the documents. All data handled through this portion of the program was either a string or vector of strings in order to keep the type of data consistent. Much of the Query class was used to store and prepare the query to be ready for comparison and I did this so that other components weren't being used until the production of the final product for the project.

# Part 6: Calculating the cosine similarity and outputting the five most similar documents (Yash Chulki)

As the last part of the project, my job was to calculate the cosine similarity between the query and each document in the collection of documents using the equation given in the project description, then outputting, in descending order, the IDs and cosine similarities of the top five non-zero cosine similarity collection documents that are most similar to the query. To do this, I decided to split up each element of the equation and find their values individually, then plug those values in the equation as a simple division formula where I divided the numerator and denominator. The values for the numerator are A and B, which are the tf-idf of each word in the document and each word in the query, respectively. These were calculated in Part 4 by teammate Ryan, so multiplying A and B gave us the numerator. For the two square roots of sums in the denominator, I had a for loop that square and added together all A's and B's respectively, then independently took their square roots. Multiplying the square roots of the sums of A and B gave us the denominator. The last remaining task was to calculate the cosine similarity is to divide the numerator and denominator, which gives us our answers.

For the output, I ran a for loop as long the size of the collection of documents (up to 5), then output the cosine similarities and IDs of each document in the collection.