**Name: Joseph Hany** **ID: 900182870**

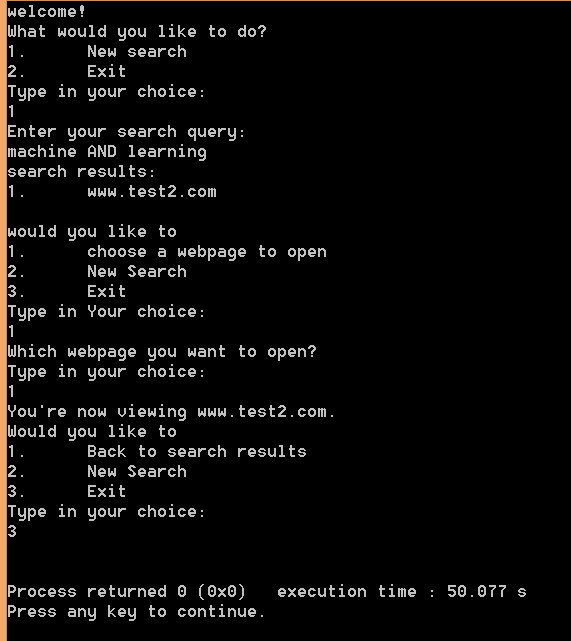
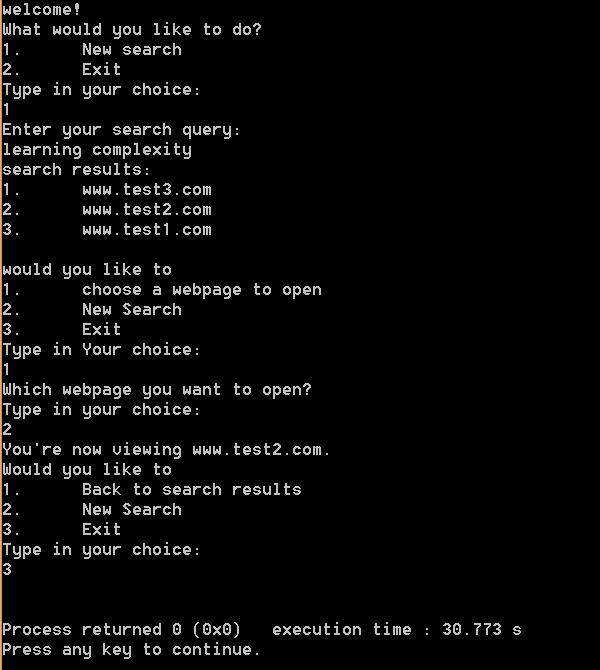
**Project Report**

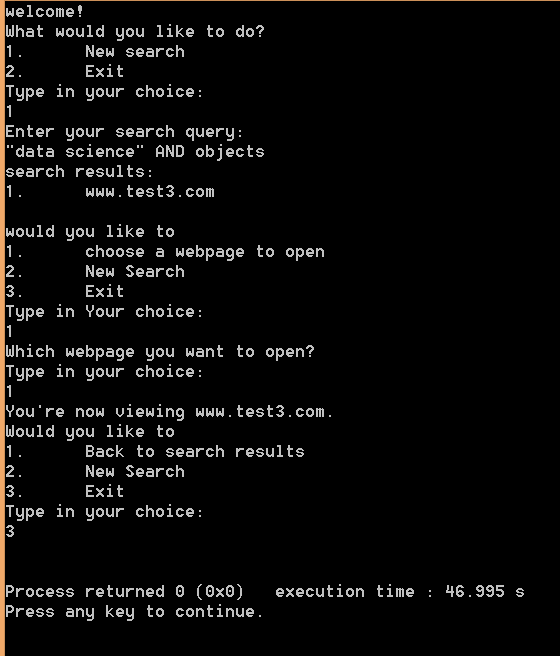
**VIP NOTE:**

To run the project properly please use any windows compiler (preferably code blocks), but don’t run it on linux because the txt or csv files format gets corrupted on linux. If you faced any problem running this program please send me on this email [joseph\_boulis@aucegypt.edu](mailto:joseph_boulis@aucegypt.edu).

* There is two cpp and excutable files one opening txt files and the other csv files choose whatever format that suits your testing

**Screenshots:**

As proof that the project is running properly see these screenshots:



1. **The pseudo-code for your indexing and ranking algorithms + time and space complexity analysis for indexing and ranking algorithms**
2. **Indexing algorithm pseudo code:**

**At first,** we initialize a map having its keys as all the keywords found in any website and its corresponding values are the strings representing the websites links which have this keyword.

**Vector<string> current search results;**

**for (int i=0 🡪 query keywords size) {**

**Vector<string> Current search results**

**for (int j=0 🡪 query keywords size) {**

**if (**the query keyword we are searching for found in the values of the map of all keywords in all websites**)**

current search results = **union** between all the websites links found for keywords **after** the **“OR“** or the **space** with last current search results which include the website links found **before** the **“OR”** or the **space**

**}**

Final search results = **intersection** between all the websites links found for query keywords **after** the **“AND”** with last Final search results which include the website links found **before** the **“AND”**

**}**

Then we **sort** the **Final search results** according to their **score** and view them to the user.

**Time complexity:**

This algorithm may seem to have a complexity of O(\*m) where n is the size of the query keywords list and m is the number of websites found in the entire web. **However,** since we will only have **simple queries** which means that there will never exist both “OR”s and “AND”s in the same query, this algorithm’s complexity will only be **O(n\*m) where n is the number of query keywords list and m is the maximum number of websites that can match to a single keywords. i.e. all the websites found in the entire web.**

Moreover, sorting after the nested loops will take at maximum **O(m\*log(m)).**

**Thus, the final complexity for this algorithm is O(m\*log(m) + n\*m)**

**Note:**

Here we cannot assume that **O(n\*m) > O(m\*log(m))** since m is the number of websites and can be much larger than the number of keywords.

i.e. let m = 200 (200 websites have the keywords found in the search query)

and let n = 2 (there are only 2 keywords in our search query)

Thus, **200\*2=400** is not larger than **200\*log(200)=460.205999**

**Space complexity:**

Since we only use a map with **keys** of type **strings** and **values** of type **vector<string>** and we will in the worst-case scenario map each keyword to all the websites found in the entire web. Thus, we will have **vectors of strings** with the number of keywords found in the entire web.

**Therefore,** the space complexity of this algorithm is **O(n\*m) where n is the number of query keywords list and m is the maximum number of websites that can match to a single keyword. i.e. all the websites found in the entire web.**

1. **Ranking algorithms pseudo code:**

**First, initializing the pagerank with 1/n**

int max\_iterations=10000;

double sigma\_tolerance=1e-12;

int n\_iteration=0;

float diff=1;

bool diff\_flag=true;

float min\_PR=FLT\_MAX;

float max\_PR=FLT\_MIN;

**//keep on looping until either we reach the max num of iterations or the pagerank converges to specific numbers and the diff flag is false**

while(**diff\_flag && n\_iteration<max\_iterations**){

old\_PageRank=PageRank;

diff\_flag=false;

**for(int i=0 🡪 page rank size){**

PageRank[i]=0;

**for(int j=0 🡪 number of websites pointing at the current website){**

float **Out\_Degree**=number of websites pointing to each website that points to the current website. i.e. out degree for all the nodes pointing at the current website

**PageRank**[i]+=(**old\_PageRank**[**In\_Degree**[i][j]]/**Out\_Degree**);

}

**//update max\_PR and min\_PR to be used in normalization of the pagerank**

if(PageRank[i]>max\_PR)max\_PR=PageRank[i];

if(PageRank[i]<min\_PR)min\_PR=PageRank[i];

**//update the diff and check whether it started to converge or not**

diff=abs(PageRank[i]-old\_PageRank[i]);

if(diff>sigma\_tolerance)diff\_flag=true;

}

**// update the number of iteration in order to stop if it exceeded 10000**

n\_iteration++;

}

**Then, we normalize the page rank**

**After that, we initialize the PageScore with the normalized pagerank**

**Time complexity:**

Since this algorithm has a while loop which at maximum iterates for 10000 times which is a constant time and has two nested loops, each of them will iterates at maximum m times, where m represents the max number of nodes or websites. Thus, the time complexity of this algorithm is **O()**

**Space complexity:**

Since we only use a vector of int with smax size equal to the max number of websites. Thus, the space complexity of this algorithm is **O(m)** where m is the number of websites.

1. **The main data structures used by my algorithm:**
   1. **Maps, used in the following:**
      1. map<string,int> links\_nodes 🡪 this links each link string to a node number
      2. map<string, vector<string>> links\_keywords 🡪 this links each keyword found in the entire web with all the websites that include this keyword
   2. **Vector:**
      1. vector<vector<int>> In\_Degree 🡪 indicates the number of nodes (websites) that points to a current website
      2. vector<vector<int>> adjList 🡪contains the relation between each node and the connected nodes to it
      3. vector<float> old\_PageRank 🡪the page rank vector that is used as a temp in order to compute the page rank values.
      4. vector<float> PageRank 🡪 will eventually contain the normalized page rank for each website
      5. vector<float> PageScore 🡪 initially initialized with the PageRank but then it is updated to help us arrange the websites in the search results view
      6. vector<pair <string,float>> search\_results 🡪 contain the current search results
      7. vector<vector<string>> parsed\_query 🡪 **VIP: in this data structure we consider each new row (vector) as a whole entity that represents ORing between keywords in a query and between each row and the other we consider having an AND**
      8. vector<int> impressions 🡪 includes how many times each website was viewed
      9. vector<string> all\_links 🡪includes all the links in a web
      10. vector<int> clicks 🡪 includes how many times a website link has been clicked on
2. **Any design tradeoffs you made along with their justifications**

**In the Indexing algorithm,** we needed to initialize a map which have as its keys all the keywords found in any website and its corresponding values are the strings representing the websites links which have this keyword. This increased our space complexity very much. To illustrate, instead only having a list of all the website in the web which has space complexity of **O(m)** where m is the number of websites in a web, we now have map which has a complexity of at most **O(m\*n)** where m is the number of websites and n is the number of keywords. However, this increase in complexity made us have a dramatic decrease in the time complexity , where instead of iterating over each and every website in the entire web and check whether it includes our search query keywords or not in **O(n)**, we will only check the map for the query keyword if it exists then we will directly know which websites has this keyword in a constant time **O(1)**.

**VIP Note:**

My parsing algorithm assumes that we can have nested ANDs and ORs and Quotations and Spaces.

And the indexing algorithm takes into consideration that this case might happen.

However, it works faster and properly on simple queries.