Assignment 3 - Wacky Marketing

- Due Apr 8 by 11:59p.m.
- Points 149
- Submitting a file upload
- File Types c
- Available Mar 18 at 12a.m. - Apr 8 at 11:59p.m.

This assignment was locked Apr 8 at 11:59p.m..

Developing a business is crucial, and at The Wacky Store, we prioritize enhancing our sales. Our strategy involves suggesting new products and brands to customers, leveraging feedback from their friends. In this assignment, students enrolled in CSCA48 are tasked with creating an algorithm that identifies brand matches for our clients. To support this endeavor, we will be manipulating a graph network.

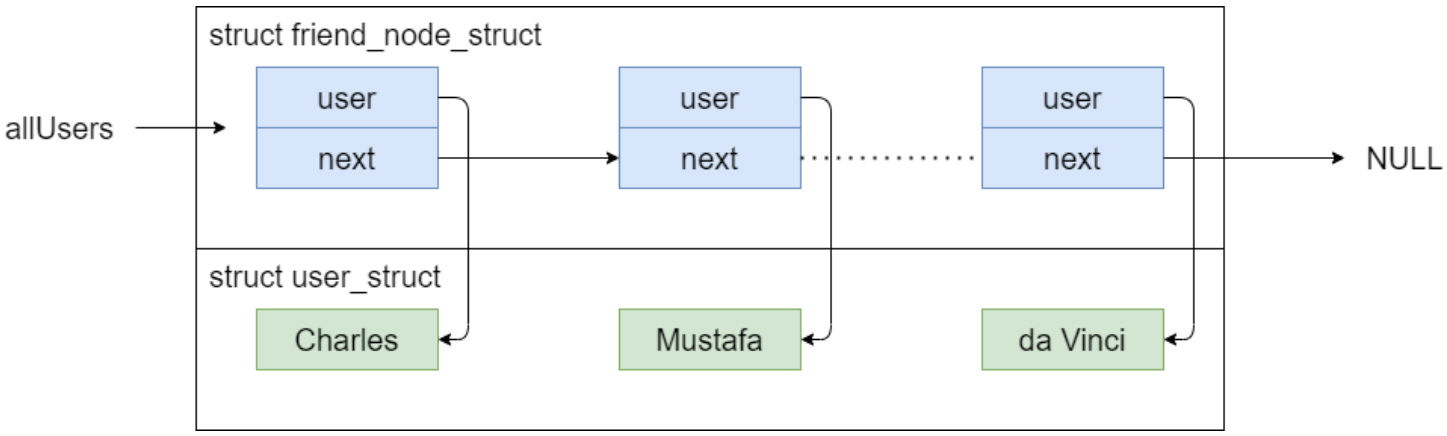
Get the starter code here: a3.zip (https://q.utoronto.ca/courses/332080/files/31021684?wrap=1) (https://q.utoronto.ca/courses/332080/files/31021684/download?download_frd=1)

Notes:

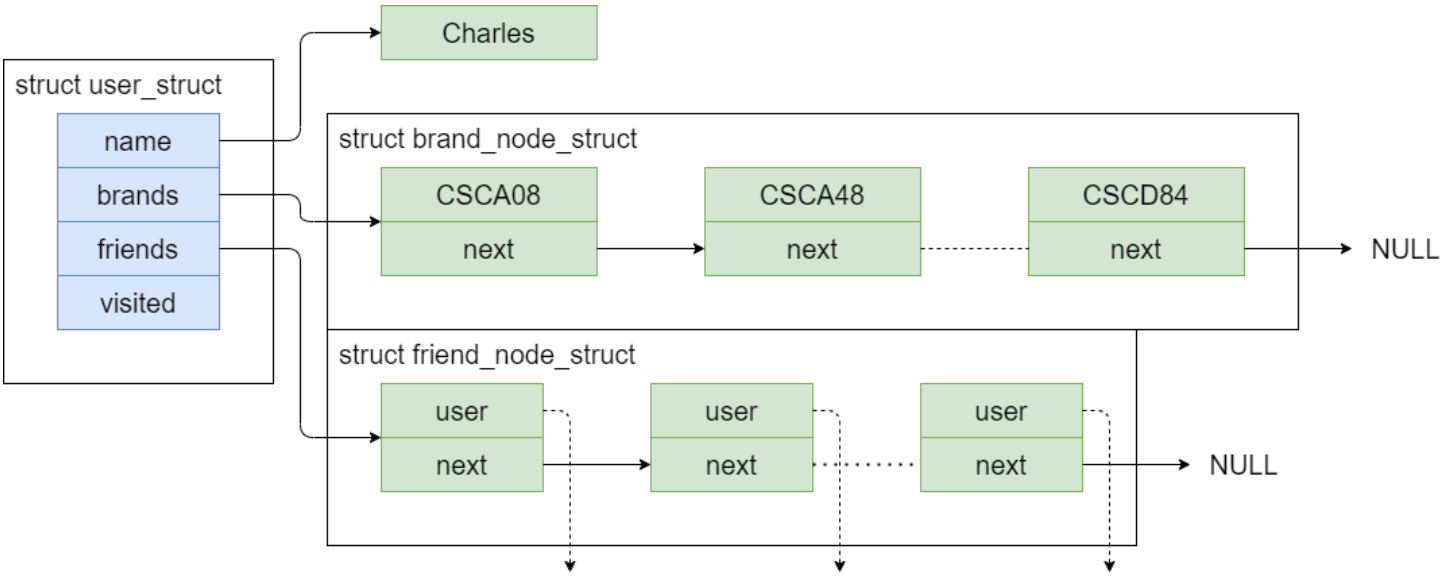
- When we mention alphabetical order, we mean ASCII sorted order by `strcmp()`.
- When testing, a brand cannot be similar to itself.
- You can make your own helper functions and define your own structs. However, when submitting, do not submit a file with a `main()` function.

Understanding the Data:

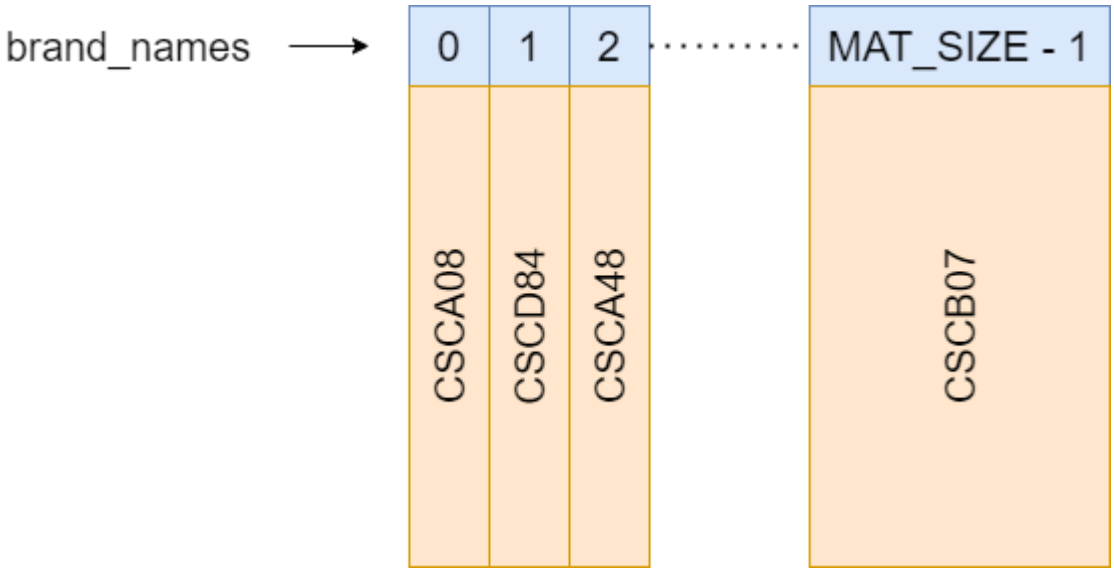
`allUsers` is global variable representing a linked list data structure holding a collection of all registered users on our platform. The linked list is sorted in alphabetical order by the user's name. Assume that all names in the linked list are and should remain unique.



Every `struct user_struct` represents a single user on the platform. For each user, we have a linked list of all the friends that they follow, as well as a linked list of all the brands that they like. Once again, both lists are sorted in alphabetical order by either the brand's name or the friend's name.

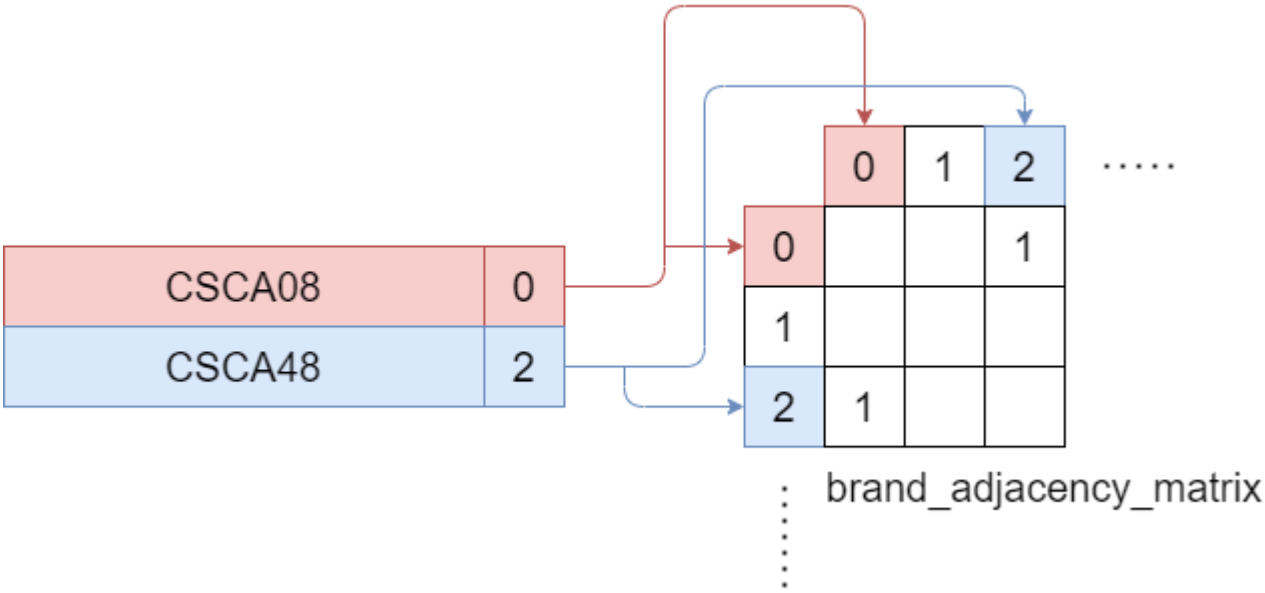


Every brand on the platform is stored inside an array of names under the variable `brand_names`. This array is not sorted in any way, shape, or form. There is guaranteed to be `MAT_SIZE` unique brand names. The image below gives an example of what the array of names might look like.



If we wanted to know if two brands are similar, we refer to the brand adjacency matrix.

For example, if we wanted to know if `CSCA08` and `CSCA48` were similar, we would check indices `0` and `2` (eg. `[0][2]` or `[2][0]`). If the brands were similar, the matrix would have a value of `1`. Otherwise, `0`.



Understanding the Starter Code:

To help out a bit, we've provided some helper functions. The following lists their respective documentation.

```
bool in_friend_list (FriendNode* head, User* node)
```

Given the head to a FriendNode linked list, returns `true` if a given user's name exists in the list. Returns `false` otherwise.

```
bool in_brand_list (BrandNode* head, char* name)
```

Given the head to a BrandNode linked list, returns `true` if a given brand's name exists in the list. Returns `false` otherwise.

```
FriendNode* insert_into_friend_list (FriendNode* head, User* node)
```

Given the head to a FriendNode linked list, inserts a given user into the linked list and returns the head of the new linked list. The insertion is done in alphabetical order. If the user already exists, no modifications are made and the list is returned as is.

```
BrandNode* insert_into_brand_list (BrandNode* head, char* node)
```

Given the head to a BrandNode linked list, inserts a given brand into the linked list and returns the head of the new linked list. The insertion is done in alphabetical order. If the brand already exists, no modifications are made and the list is returned as is.

```
FriendNode* delete_from_friend_list (FriendNode* head, User* node)
```

Given the head to a FriendNode linked list, removes a given user from the linked list and returns the head of the new linked list. If the user does not exist, no modifications are made and the list is returned as is.

```
BrandNode* delete_from_brand_list (BrandNode* head, char* node)
```

Given the head to a BrandNode linked list, removes a given brand from the linked list and returns the head of the new linked list. If the brand does not exist, no modifications are made and the list is returned as is.

```
void print_user_data (User* user)
```

Given a user, prints their name, friends, and liked brands.

```
void print_brand_data (char* brand_name)
```

Given a brand, prints their name, index (inside the `brand_names` array), and the names of other similar brands.

```
int get_brand_index (char* name)
```

Given a brand, returns the index of the brand inside the `brand_names` array. If it doesn't exist in the array, return `-1`.

Your Task:

You will be responsible for implementing a few functions. The following lists their respective documentation. A driver is also provided in the starter code to briefly test your implementation,

however, for more through testing, modifications to the code is necessary.

```
User* create_user (char* name)
```

Given a name, creates a new user on the platform and returns the created user. When the user is created, they must also be inserted into the `allUsers` linked list. If a user already existed with the same name, do nothing and return `NULL` instead.

```
int delete_user (User* user)
```

Removes a given user from the platform. The user must be removed from the `allUsers` linked list and the friend list of any users that they belong to. Return `0` if the user was successfully removed. If the user does not exist, return `-1` instead.

```
int add_friend (User* user, User* friend)
```

Given a pair of valid users, create a friendship. A user's friends list must remain in alphabetical order. Return `0` if the friendship was successfully created. Return `-1` if the pair were already friends.

```
int remove_friend (User* user, User* friend)
```

Given a paid of valid users, remove their friendship. A user's friends list must remain in alphabetical order. Return `0` if the pair are no longer friends. Return `-1` if the pair were not friends to begin with.

```
int follow_brand (User* user, char* brand_name)
```

Given a valid user and the name of a brand, create a link between the user and the brand. A user's brands list must remain in alphabetical order. Return `0` if the link was successfully created. Return `-1` if the link already existed or if the brand name is invalid.

```
int unfollow_brand (User* user, char* brand_name)
```

Given a valid user and the name of a brand, remove the link between the user and the brand. A user's brands list must remain in alphabetical order. Return `0` if the link was successfully removed. Return `-1` if the link did not previously exist or if the brand name is invalid.

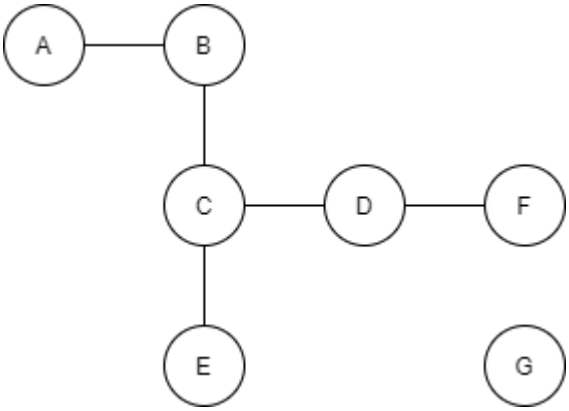
```
int get_mutual_friends (User* a, User* b)
```

Given a pair of valid users, return the number of mutual friends between them. A mutual friend is a user that exists in the friends list of both `User a` and `User b`.

```
int get_degree_of_connection (User* a, User* b)
```

Given a pair of valid users, return the degree of connection between both users. The degree of connection is the shortest number of steps it takes to get from one user to the other. If a connection cannot be formed, return `-1`.

Suppose this was a graph depicting the connections ("friendships") between 7 users on our platform. An example of inputs and expected outputs is provided below.



User A	User B	Degree of Connection	Explaination
A	F	4	A -> B -> C -> D -> F
A	A	0	A
A	B	1	A -> B
A	G	-1	***None***

You may have noticed that `struct user_struct` contains a `bool visited` variable. You may use this variable however you wish; the automarker will ignore the values inside these variables. You may also add additional variables to the struct as you wish (eg. `int distance;`). However, the automarker will not initialize these variables for you. You must initialize or reset these parameters at the start of the function should you wish to use it. Here is the psuedo-code of what we mean:

```
get_degrees_of_connection():
  // reset variables
  for all user in allUsers:
```

```
    set user.visited to false

  // perform algorithm ...
  do_stuff()
```

```
void connect_similar_brands (char* brandNameA, char* brandNameB)
```

Given two brand names, mark the two brands as similar in the `brand_adjacency_matrix` variable. If either brand name is invalid, do nothing.

```
void remove_similar_brands (char* brandNameA, char* brandNameB)
```

Given two brand names, mark the two brands as different (not similar) in the `brand_adjacency_matrix` variable. If either brand name is invalid, do nothing.

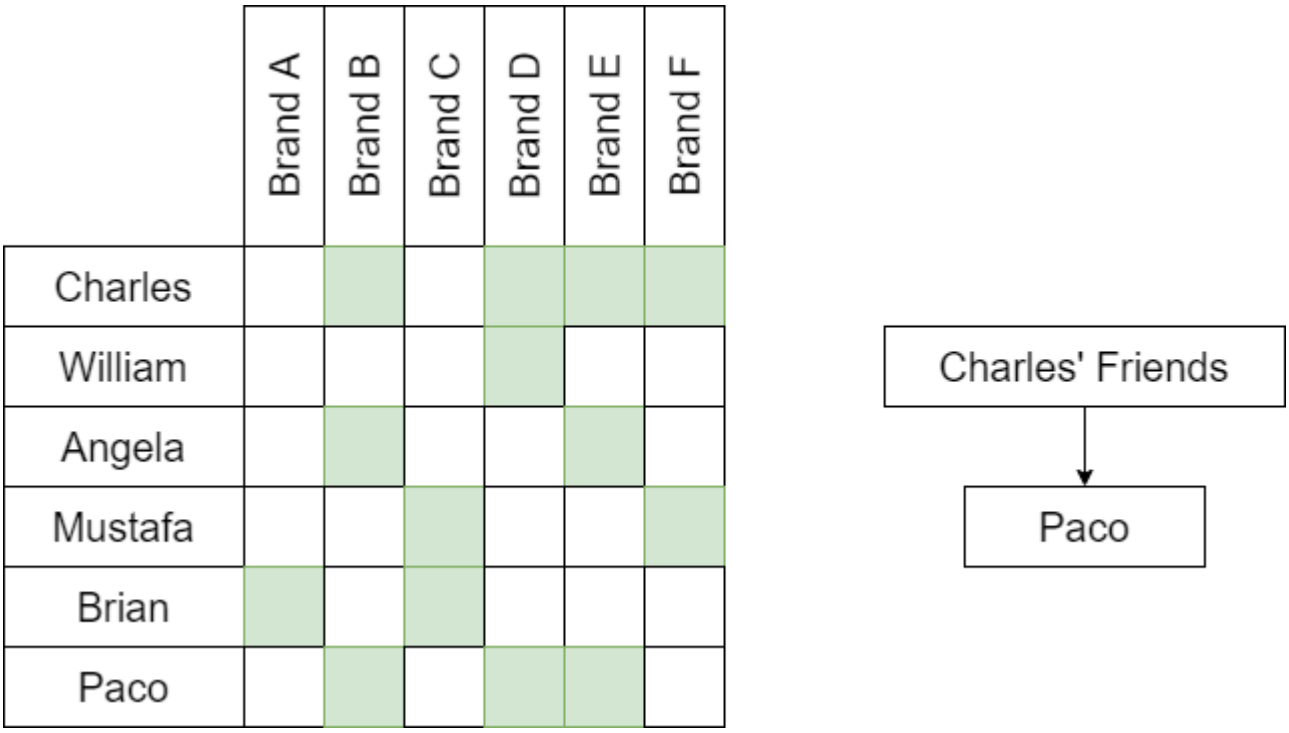
```
User* get_suggested_friend (User* user)
```

Given a user, suggest a new friend for them. To find the best match, the new suggested friend should have the highest number of mutually liked brands amongst all other valid candidates. If a tie needs to be broken, select the user with the name that comes first in **reverse-alphanumerical order**. The suggested friend must be a valid user, cannot be the user themself, nor cannot be some that they're already friends with. If the user is already friends with everyone on the platform, return `NULL`. Also, at times, you might add users with no mutually liked brands; worst comes to worse, at times, a similarity rating of 0 is the best option.

```
int add_suggested_friends (User* user, int n)
```

Given a user and a positive integer `n`, add `n` suggested friends using the `get_suggested_friend()` function. There might not be enough users on the platform to sastify `n` complete, so return the amount of friends successfully added.

Suppose our platform had the following user network:



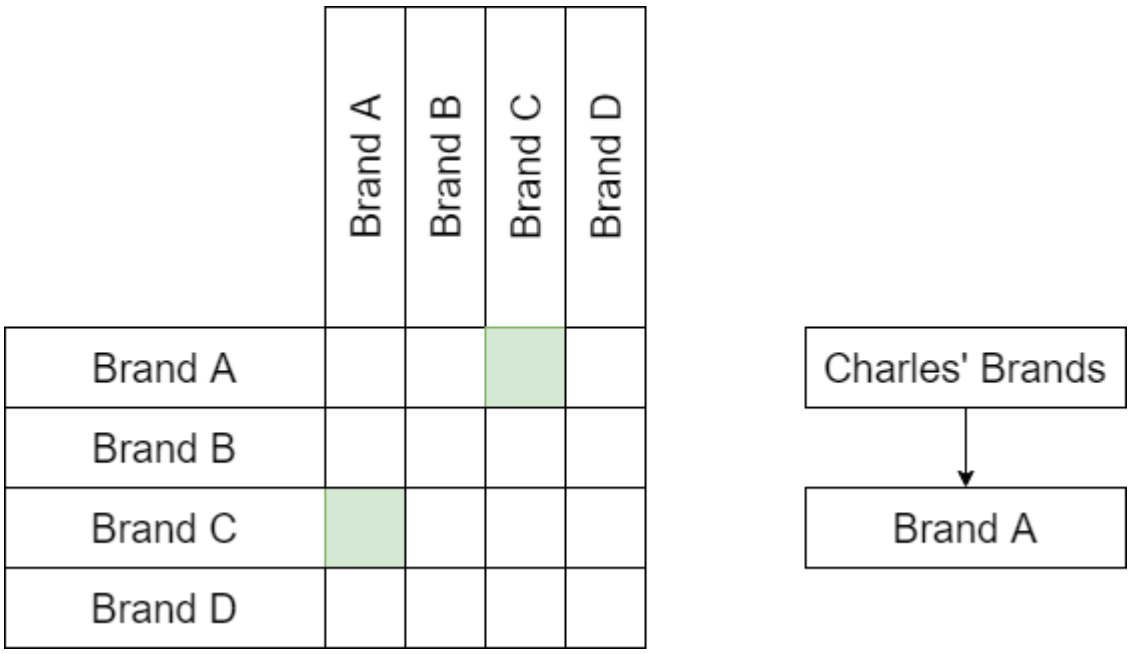
We want to find friends for `Charles`, who currently only has a single friend: `Paco`. `get_suggested_friend()` will return friends for `Charles` in the following order.

User	Similar Brands	Order	Explanation
Charles			You cannot add yourself as a friend.
William	1	2	Although William has the same count as Mustafa, William comes after the alphabet as compared to Mustafa, so William comes first.
Angela	2	1	Angela has the highest number of mutually liked brands as Charles, amongst all other valid candidates.
Mustafa	1	3	
Brian	0	4	Brian has the least number of mutually liked brands as Charles compared to all other valid candidates, so Brian is added last.
Paco			Paco is already Charles' friend.

```
int follow_suggested_brands (User* user, int n)
```

Given a user and a positive interger `n`, suggest `n` new brands for them. To find the best matches, suggested brands with the most similarities with the brands that the user already follows. If a tie needs to be broken, select the brand with the name that comes first in **reverse-alphanumerical order**. The suggested brand must be a valid brand and cannot be a brand that the user already follows. There might not be enough brands on the platform to sastify `n` complete, so return the amount of brands successfully followed. Like `add_suggested_friends`, sometimes, adding a brand with a similarity rating of 0 is the best option.

Suppose our platform had the following user network:



We want to find `2` brands for `Charles` to follow, who currently only follows a single brand: `Brand A`.

Brand	Similarity	Order	Explaination
Brand A			Charles already follows Brand A.
Brand B	0	3	Brand B is not similar to any of the brands that Charles follows.
Brand C	1	1	Brand C has a similarity rating of 1 (similar to Brand A, which Charles follows).
Brand D	0	2	Brand D is not similar to any of the brands that Charles follows. However, it comes first reverse-alphanumerical compared to Brand B, so it is added before Brand B.

As a result, the algorithm will prompt Charles to follow `2` brands: `Brand C` and `Brand D`.

Hint and Warning: `follow_suggested_brands(X, 4)` is not the same as calling `follow_suggested_brands(X, 1)` repeatively `4` different times. In the second example, 4 consecutive calls to `follow_suggested_brands(X, 1)` may result in better similarities for some brands as another brand is added. We **DO NOT** want this; you should be sorting the brand using similarities at one specific point in time, and this similarity does not change until the end of the function call. Please use the first interpretation `follow_suggested_brands(X, 4)`.

Here is an example:

Charles follows no brands. There are 4 brands in total (A, B, C, D). Only Brands A and D are similar with each other.

If I were to call `follow_suggested_brands(Charles, 2)`, then **Charles would follow Brands C and D**. This is because all brands are equally scored [0, 0, 0, 0], but **C and D are the last 2 in reverse order**.

However, if I were to call `2 x follow_suggested_brands(Charles, 1)`, then **Charles would follow Brands A and D**. This is because in **the first call**, Charles would follow Brand D, whileas in **the second call**, Charles would follow Brand A (as A is similar to D, thus making it more preferable than B and C).

Notice how each call to `follow_suggested_brands` **score all brands at one specific point in time**.

Final Remarks:

Use piazza and office hours wisely! Start early and work diligently. This assignment is a more challenging assignment than both A1 and A2, so you will need to work productively and not wait until the last minute.

As usual, please submit the contents of your `.c`. The file name does not matter.