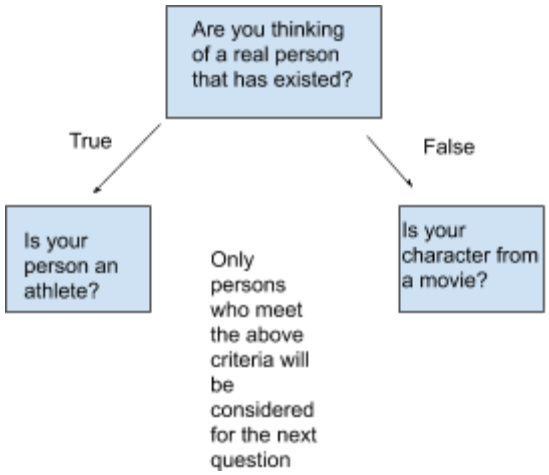
Project Description

My project was my own version of the famous Akinator game, minus the UI. The user plays from the *CommandLine* or *PowerShell*, and is prompted to respond to yes or no questions. The program eventually guesses who it thinks the person is thinking of.

The program essentially deals with one large database. The contents of the databases are the characters I have manually inputted and their unique descriptions. Eventually, that database thins itself out to only a couple of values based on the responses from the user from which the program randomly selects a character to guess. The following is a visual map to illustrate:



Application of Computational Thinking

The first step and one of the most crucial in my delivery of the problem was hollowing out the project to its core concept.

Abstraction of the Akinator:

The program begins from a starting point of storing mass data, then as the program runs it halves the data repeatedly until it can come across only one (or a select few) possible match(es).

Based on this abstraction, I then proceeded to brainstorm implementation ideas. I will only include the one that I determined to be most plausible and have the least drawbacks, which I will discuss in further detail later.

Implementation idea:

- **Brainstorm** about 50 characters and add them each with their **descriptions** into one file. Prompt the user to answer a question from a **question bank** with either True or False. Characters that don't share the same answer as the one the user inputs will be **deleted from a dictionary**.

Next, I **decomposed** the implementation idea into its deliverables, in order to get closer to an actual algorithm:

Deliverables:

1. **Brainstorm:**

Manually inputting all the characters the program will be able to choose from, into a file.

2. **Descriptions:**

Manually creating an appropriate description to identify and map out any specific character. These descriptions should double as questions the program can ask so as to make use of the efficiency of dictionary searching.

3. **Question Bank:**

Create and store a question set consisting of the aforementioned character descriptions. The key is figuring out how to ask the appropriate questions so as to render the game elegant and efficient (not brute force it by asking every single possible question in the game)

4. **Deletion from Dictionary:**

Delete characters from the dictionary that do not fit the user's description.

Evaluation

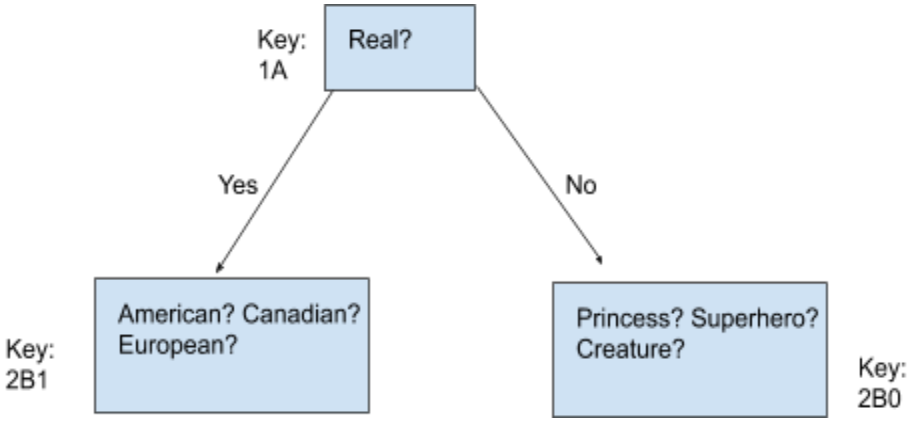
I quickly realized that deliverables 1), 2) and 4) were simple solves, but deliverable 3) was more complex. How would the program know what questions were appropriate to ask? This turned out to be the bulk of the effort that went into the program. I hypothesized that maybe some sort of tree structure we had briefly seen in class might work, which led me to discovering decision trees. I decided to decompose the decision tree idea to get closer to a concrete algorithm

Decomposition and Algorithm Design:

The parts below that are highlighted in blue represent the basis of my algorithm design. I often find that decomposition and algorithm design go hand in hand, almost done simultaneously, in my approach to solving CS problems.

1. Tree Structure
    a. The tree I designed contained clusters of questions. I realized that I could group the questions together in these clusters to minimize the quantity of branches the tree would require. This cluster system was organized off of the following 3 question types:
        - Flow changing questions: these are questions that, if true, change the flow of the program (i.e. each question points to a new and different cluster of questions). The order of these questions matter; if the answer to one of the questions is yes, the following questions do not get asked.
        - Yes or No questions: these questions thin the database but do not alter the flow of the program. Their order does not matter.
        - Sequential questions: these questions must be asked sequentially (order matters) but they do not alter the flow of the program. They all point to the same outcome (usually termination of the program).
    b. I created the tree in a file, and gave each cluster a unique identifier, or a key. This key was had the following structure:
        - First character of the key was a number representing its level in the hierarchy
        - Second character was a letter representing its type (A for flow changing, B for Yes or No and C for Sequential)

- All following digits were based on the path taken to get to that cluster. Essentially, after either a flow changing question or the last question of a Yes or No cluster had been asked, or a sequential question had been answered with *yes*, the cluster would have 0's and 1's to represent the answers to those questions (0 = no, 1 = yes). That way, clusters could be uniquely identified.
- The following is a visual map to better illustrate:



2. Question Flow

Now that I had an identification system for the questions, I just needed to find a way to navigate through it. I decomposed this into 3.5 mini solutions:

1. Key Making*** - getting a key in the form of a list and transforming it to a string.
2. Key Building - receiving a key and adding onto it to produce the next key. The next key points to a new cluster of questions, or the key is invalid and results in termination

    a. Key Checking - since the letter is non mathematical and can't be generated by a sequential/mathematical process, a helper needs to check for the appropriate letter and pass it to the key building process.

3. Path - entering a cluster, asking the questions within it and depending on the cluster type and response, building the appropriate key.

Pattern Recognition:

A notable use of pattern recognition ironically involved the identification of what looked like a pattern, but was in reality not a pattern. I initially thought that key building would be the same throughout the program, but quickly realized that was not the case. Since all 3 types of questions had a unique influence on the remaining clusters, their keys had to be built differently.

Runtime and Complexity Analysis:

```
def keyMaker(lis):#runtime
    key = str(lis[0])
    if len(lis) ==1:
        return str(lis[0])
    return key + keyMaker(lis[1:])
```

This recursive function runs in O(n). Its purpose is to build a string version of the key in list form that it receives. I could alternatively have kept every key as a string, however this would clutter up some of the functions below. Strings are immutable, so I can't change its type once I set it. I would have had to slice the keys repeatedly. Initially storing it as a list so I can easily change its values, then changing it to a string to use as a key in a dictionary was the solution that I preferred.

```
def keyCheck(key, filename): #runtime
    questions = questionReader(filename)
    for keyword in questions:
        counter = 0
        i = 0
        for letter in range(0,len(keyword)):
            if str(key[i]) == keyword[letter]:
                counter +=1
                if i == 1:
                    counter -=1
            if counter == len(key)-1:
                return keyword[1]

            i+=1
            if i == len(key):
                break
```

My keyCheck function runs in O(n*m). This function has to look through the dictionary containing the questions/tags, *n,* and compare each tag to the key it's been given. To do this, it needs to check every single character in the tags, *m*. Once it finds the tag that most matches the key, it returns the tag's type so the next function can add that type to the key.

```
126 def charcounter(filename):
127     counter = 0
128     file = open(filename, "r")
129     for i in file: #runtime
130         if i.strip() == "/":
131             counter +=1
132     return counter
133
```

The charcounter function runs in O(n), in order to find the amount of characters in the character bank. This serves as a helper function to the loadcharacters function below.

```
133
134 def loadcharacters(filename):
135     people = open(filename, "r")
136     line = "b"
137     counter = charcounter(filename)
138     person = {}
139
140     for i in range(counter): #runtime
141         characters = {}
142         line = people.readline().strip()
143         name = line
144         while line != "/":
145             line = people.readline().strip()
146             if line == '/':
147                 break
148             characters[line] = True
149
150         person[name] = characters
151
152     people.close()
153     return person
154
```

The loadcharacters function actually runs in O(n*m) time. The helper function above allows this function to offset its iterations. The outer loop runs n times, while the inner loop runs the m times. This function reads the characters from the database, saving their names as the key, and a dictionary containing their descriptions as the value.

```
156 def check(ques, choice, characters):
157     if choice.lower().strip() == "yes":
158         for p in list(characters):  #runtime
159             if ques not in characters[p]: #runtime
160                 del characters[p]
161         return characters
162     elif choice.lower().strip() == "no":
163         for p in list(characters):
164             if ques in characters[p]: #runtime
165                 del characters[p]
166     return characters
167
```

My check function runs in O(n). It receives the users choice, the question bank as well as a dictionary containing the remaining characters. I can't escape checking every single character, but my use of a dictionary allows me to hash search in O(1).

```
def pathC(key,filename,charactersList):
    keyword = keyMaker(key)
    questions = questionReader(filename)
    header = keyword

    dic = {}
    for i in range(len(questions[header])): #runtime
        choice = input("Does this tag fit your character?: " + questions[header][i] + "- ")
        while choice.lower() == "" or (choice.lower() != 'yes' and choice.lower() != "no"):
            choice = input("I SAID...does this tag fit your character?: " + questions[header][i]+ "- ")
        if choice.lower().strip() == "yes":
            dic = check(questions[header][i], choice, charactersList)
            key,keyword = keyBuilderC(key,filename)
            return key,keyword,dic
        if choice.lower().strip() == "no":
            dic = check(questions[header][i], choice, charactersList)

    key,keyword = keyBuilderC(key,filename)
    return key,keyword,dic
```

I'm illustrating the runtime using pathC, but all the path functions have the same runtime; O(n). This function repeatedly asks questions from the appropriate cluster. The choice dictionary and question asked are passed to the check function. It returns the dictionary, the new key and the key in string form.

```
while True: #runtime
    if key[1] == "B":
        key,keyword,characters = pathB(key,filename,characters)
        print(key)
    elif key[1] == "A":
        key,keyword,characters = pathA(key,filename,characters)
        print(key)
    elif key[1] == "C":
        key,keyword,characters = pathC(key,filename,characters)
        print(key)
```

I've included this snippet from my main function because there's something interesting about it. I could have substituted the while loop for a for loop that ran 6 times. This is because my decision tree has 6 levels. The program can never run past 6 levels, since clusters are also designed vertically not horizontally. Plus, each iteration processes a single cluster. However, for the purpose of plasticity, I keep the program running until an unknown key is created. That way, if I add questions and levels, I won't have to keep track of them and continuously update the for loop.

Trade-Offs

The first “trade-off” I considered was fairly easy. I knew I could brute force my way into making a guess, simply by asking every single question I had written down. The obvious advantage would be that this would be a fairly simple implementation, however it was clear that this lost the main goal of the program: elegance. The secret recipe of the Akinator is not that it guesses who you are thinking about, it is that it does so elegantly. It asks relevant questions, so the user is consistently surprised that the program “knows” what it's thinking. The only way to achieve this effect is through elegant program design, not brute force.
When it came down to execution, there weren’t many things I had to consider in terms of trade-off, however I did really weigh my initial implementation ideas. I had a total of 4, with the one mentioned above being the ultimate winner. These were the three others:

1. Many Different Files
- Brainstorm about 50 characters. Map each one of them to a series of categories, enterable by yes or no questions.

Ex. Michael B Jordan
Alive? Yes - MBJ is in this file
Actor? Yes - MBJ is in this file
Male? Yes - MBJ is in this file
Played in Action Movie? Yes - MBJ is in this file
Featured in MCU Movie? Yes - MBJ is in this file
American? Yes - MBJ is in this file
Randomly choose from this File
Problem with this implementation - though this specific question set maps out to MBJ, there may be many people included in the American file, so this wouldn't narrow anything down. **This implementation’s accuracy is inversely proportional to the size of the last file checked.**

2. One Big File
- Brainstorm about 50 characters, store specific information for each(list, dictionary?) (almost feels OOP ish). Eliminate characters that do not qualify to the question searched for, and ask a new one

Ex. Michael B Jordan
Alive? Yes - everyone who is not alive is erased
Actor? Yes - everyone who isn’t is erased
Male? Yes - everyone who isn’t is erased
Played in Action Movie? Yes - everyone remaining who didn’t is erased
Featured in MCU Movie? Yes - everyone remaining who didnt is erased
American? Yes - everyone remaining who is not american is erased
Problem with this implementation - we’ve seen many times in this course that the deletion of things from files is very taxing. Is this the most efficient solution?

3. Many subFiles

- Same as above, except instead of eliminating characters that do not qualify, I can add the characters that *do* to a subfile, and repeat the process until the file only contains a few people

Ex. Michael B Jordan
Alive? Yes - everyone who is alive is moved to a new file
Actor? Yes - everyone remaining who is is moved to a new file
Male? Yes - everyone remaining who is is moved to a new file
Played in Action Movie? Yes - everyone remaining who has is moved to a new file
Featured in MCU Movie? Yes - everyone remaining who has is moved to a new file
American? Yes - everyone remaining who is is moved to a new file
Problem with this implementation - it would be very tedious to continuously read all the character descriptions, as well as repeatedly creating files. Also, it would always add files to my directory which doesn't necessarily use more space, but it is a clutter.

Improvements and Extensions

Runtime:
Obviously, achieving an O(1) solution is the ideal for any program. However, I really don't think that is possible for the scope of this project. However, my keyCheck function runs in O(n^m). I think it may be possible to scale this down as low as O(logn). It looks a little like the searching algorithms we've seen in class, which we saw could be optimized to O(logn). I am not sure it can be done, but I think I could definitely attempt it.

Question Design:
I am not satisfied with some of my questions. On the "real" side, the questions are pretty layered, but I think I still could have added more depth to them, especially the not real side where all questions are type B and C. Since I solved the problem of question navigation, this would have been a manual adjustment, however I was short for time given my course load, and I wanted to add a lot of characters. I'll pursue further development of questions in my extensions.

Question Design:
In 1406, I may extend this project by adding and further developing/refining my questions. I want to add more type A questions because as mentioned earlier this is where the elegance of the problem lies. Also, this would allow the program to enlarge its character capacity, as more specific paths allows greater character differentiation.

Program Plasticity
A big extension I would pursue possibly in 1406 would be adding plasticity; allowing the program to grow and change based on user interaction and not my own manual input. I would like the program to be able to grow its database of characters by adding a character if the user

beats it . I would also like the user to be able to enter new questions they think would map out to their character and potentially others, and have the program incorporate those questions.

- Adding characters would be fairly easy to implement:
    - I would prompt them to enter the character's name, and add that character's name to the character file/database.
    - Have a dictionary save "yes" as the key, and a list of the questions that were answered with "yes" as the value.
    - I would print the list of questions to right underneath the character's name in the character file (these would be the character's descriptions/tags)
- Adding questions would be a little harder. I don't have a programmable method of determining a questions type independently, but I may be able to ask them how their question compares to the ones I have already. Depending on their response, I might be able to place the question in a cluster, however I am not quite sure how to incorporate the questions that are outside of the scope of my question bank WITHOUT relying too heavily on the user. It's important that I have maximal control over the process or a user could accidentally jamble my program.

Graphical User Interface:

An extension that I would without a doubt incorporate in a 1406 rendition of this project would be a user interface. I find that Java lends itself well to graphics work, so I could add a user friendly and appealing visual side to the project.