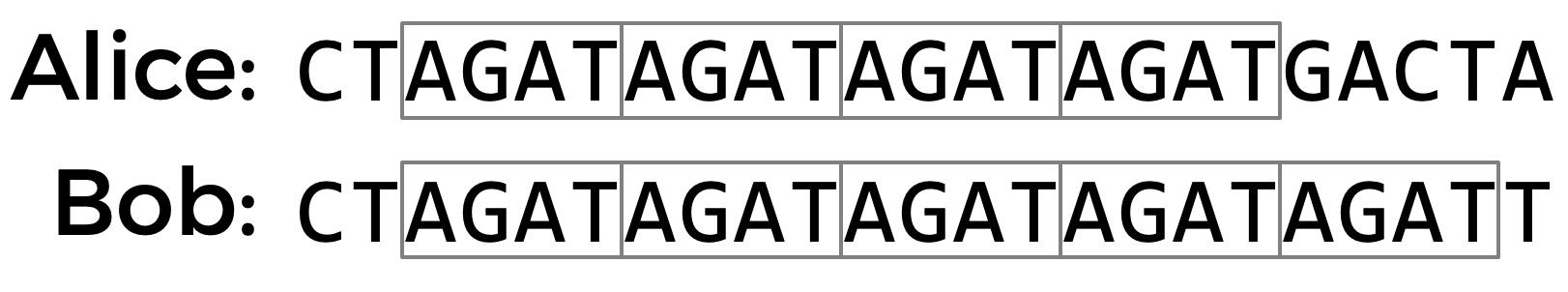
# DNA assignment

## Background

DNA, the carrier of genetic information in living things, has been used in criminal justice for decades. But how, exactly, does DNA profiling work? Given a sequence of DNA, how can forensic investigators identify to whom it belongs?

Well, DNA is really just a sequence of molecules called nucleotides, arranged into a particular shape (a double helix). Each nucleotide of DNA contains one of four different bases: adenine (A), cytosine (C), guanine (G), or thymine (T). Every human cell has billions of these nucleotides arranged in sequence. Some portions of this sequence (i.e. genome) are the same, or at least very similar, across almost all humans, but other portions of the sequence have a higher genetic diversity and thus vary more across the population.

One place where DNA tends to have high genetic diversity is in Short Tandem Repeats (STRs). An STR is a short sequence of DNA bases that tends to be repeated back-to-back numerous times at specific locations in DNA. The number of times any particular STR repeats varies a lot among different people. In the DNA samples below, for example, Alice has the STR AGAT repeated four times in her DNA, while Bob has the same STR repeated five times.



Using multiple STRs, rather than just one, can improve the accuracy of DNA profiling. If the probability that two people have the same number of repeats for a single STR is 5%, and the analyst looks at 10 different STRs, then the probability that two DNA samples match purely by chance is about 1 in 1 quadrillion (assuming all STRs are independent of each other). So if two DNA samples match in the number of repeats for each of the STRs, the analyst can be pretty confident they came from the same person. CODIS, The FBI's DNA database, uses 20 different STRs as part of its DNA profiling process.

What might such a DNA database look like? Well, in its simplest form, you could imagine formatting a DNA database as a text file, wherein each row corresponds to an individual, and each column corresponds to a particular STR. In this format each field is separated by a space

name AGAT AATG TATC

Alicia 28 42 14

Bob 17 22 19

Charlie 36 18 25

The data in the above file would suggest that Alicia has the sequence AGAT repeated 28 times consecutively somewhere in her DNA, the sequence AATG repeated 42 times, and TATC repeated 14 times. Bob, meanwhile, has those same three STRs repeated 17 times, 22 times, and 19 times, respectively. And Charlie has those same three STRs repeated 36, 18, and 25 times, respectively.

So given a sequence of DNA, how might you identify to whom it belongs? Well, imagine that you looked through the DNA sequence for the longest consecutive sequence of repeated AGATs and found that the longest sequence was 17 repeats long. If you then found that the longest sequence of AATGs is 22 repeats long, and the longest sequence of TATC is 19 repeats long, that would provide pretty good evidence that the DNA was Bob's. Of course, it's also possible that once you take the counts for each of the STRs, it doesn't match anyone in your DNA database, in which case you have no match.

In practice, since analysts know on which chromosome and at which location in the DNA an STR will be found, they can localize their search to just a narrow section of DNA. But we'll ignore that detail for this problem.

Your task is to write a program that will take a sequence of DNA and a CSV file containing STR counts for a list of individuals and then output to whom the DNA (most likely) belongs.

The overall approach will be to read in a text file that contains names and STR sequences and store each individual record in an array of Record objects. This will be your DNA database. You will next read in a DNA sample (from a different text file) and count the longest number of AGAT, AATG, and TATC sequences in that sample. You will finally go through the array of Record objects and see if there is a match (meaning the numbers listed for each sequence matches what you found to be the longest sequences in that sample. Your program will print out the read in DNA sample and information of the person if there is a match or NOT FOUND if there is no match.

## Requirements

1. You will create three classes:
   1. Create a class called **DNA** (filename is DNA.java)
   2. Create a class called **Profile** (filename is Profile.java)
   3. Create a class called **DNARecord** (filename is DNARecord.java)
2. For the Profile class,
   1. This represents the DNA profile of a sample (meaning the counts of three longest sequences of STRs). You will have three instance variables (variable name;datatype)
      1. agatCount; int - count of AGAT STRs
      2. aatgCount; int - count of AATG STRs
      3. tatcCount; int - count of TATC STRs
   2. create getters for all instance variables: getAgatCount; getAatgCount; getTatcCount
   3. create setters for all instance variables: setAgatCount; setAatgCount; setTatcCount
   4. override the **equals** method with the following signature which returns true if the passed in DNA profile object STR values match the STR instance variables of the object

| @Override  public boolean equals(Object o) |
| --- |

* 1. override the **toString** method with the following signature which returns the following formatted String "(AGAT = 4, AATG = 10, TATC = 14)" where the values represent the correct counts for each STR

| @Override  public String toString() |
| --- |

1. For the DNARecord class,
   1. you will have two instance variables (variable name;datatype)3
      1. name; String -name of the person
      2. profile; Profile- the DNA profile for that person
   2. Constructor with the following signature

| public DNARecord(String name, Profile profile) |
| --- |

* 1. create getters for all instance variables: getName, getProfile
  2. override the **toString** method with the following signature which returns the following formatted String "Fatema: (AGAT = 3, AATG = 5, TATC = 4)" where the values represent the correct name and counts for each STR in that record

| @Override  public String toString() |
| --- |

1. For the DNA class,
   1. you do not need any instance variables
2. Create the following methods**:**
   1. **readData**
      1. Method signature

| public static DNARecord[] readData(String fileName) |
| --- |

* + 1. Input parameter: String
       - String of filename to be read
    2. returns: DNARecord array
    3. The fileName will be a string path to a file in the following format. Note the maximum number of lines in the file will be 21(including the header). You can assume the file will be correctly formed and that there will be at least one data record in the file. Use a try/catch to open the file and return null if the file is not successfully opened

name AGAT AATG TATC

Alicia 28 42 14

Bob 17 22 19

Charlie 36 18 25

* + 1. Pseudocode

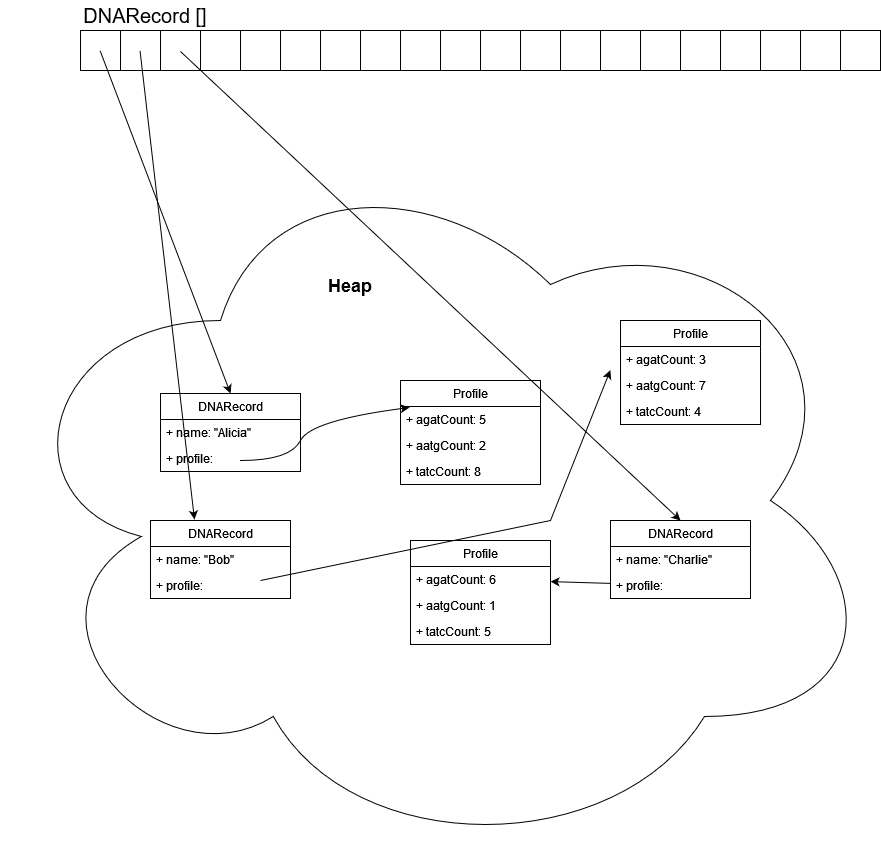
| create an empty array that can store 20 DNARecords  try to open the text file  if successful,  create a scanner,  skip the first line (header row)  for each line in text file  use the scanner to read in values  create a new DNARecord and add to the array  return the array  if not successful, return null |
| --- |

* + 1. Example

file data1.txt

| name AGAT AATG TATC  Alicia 5 2 8  Bob 3 7 4  Charlie 6 1 5 |
| --- |

**readData("data1.txt")** returns a DNARecords [] object (shown in a picture form)



* + 1. **countMaximumConsecutiveMatches**
    2. Method signature

| public static int countMaximumConsecutiveMatches(String sequence, String match) |
| --- |

* + 1. Input parameter: String, String
       - The sequence to check for consecutive matches
       - The match string
    2. returns: int
       - The count of maximum number of consecutive matches
    3. Below is the pseudocode for one approach to this method, feel free to come up with your own algorithm

Pseudocode

| Convert both the sequence and match strings to an array of characters (String class has a toCharArray method)  Set the index for the character position in the sequence to 0  Set maximum matches to 0  Set a match counter to 0  Set an offset to 0  While the sequence index plus offset is less than the sequence  length  Check to see if the character of the sequence plus offset and  match plus offset are equal  if equal  if offset equals match length minus 1  increase match counter by 1  update maximum matches if match counter is  greater than maximum matches  reset the offset to 0  increase sequence index by length of match string  otherwise  increase offset by 1  otherwise  reset match count to 0  increase sequence index by 1  reset offset to 0  return the max number of matches |
| --- |

* + 1. Examples
       - countMaximumConsecutiveMatches("catdog**catcat**fish","cat") returns 2
       - countMaximumConsecutiveMatches("dogfishdogfishfishdog**fishfishfishfish**","fish") returns 4
  1. **readSequence**
     1. Method signature

| public static Profile readSequence(String fileName) |
| --- |

* + 1. Input parameter: String
       - String of filename to be read for DNA sequence
    2. returns: Profile
       - An Profile object with STR counts for AGAT, AATG, and TATC for the sequence

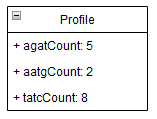
| Create a Profile object  try to open the file  if successful, using a scanner to read the first line  of the file  call countMaximumConsecutiveMatches on the sequence  with AGAT, set in profile object  call countMaximumConsecutiveMatches on the sequence  with AATG, set in profile object  call countMaximumConsecutiveMatches on the sequence  with TATC, set in profile object  return the profile object  if not successful then return null |
| --- |

* + 1. Example

file sequence1.txt

| AGACGGGTTACCATGACTATCTATCTATCTATCTATCTATCTATCTATCACGTACGTACGTATCGAGATAGATAGATAGATAGATCCTCGACTTCGATCGCAATGAATGCCAATAGACAAAA |
| --- |

**readSequence("sequence1.txt")** returns a Profile object (shown in a picture representation)

****

* 1. **main**
     1. Method signature

| public static void main(String[] args) |
| --- |

* + 1. Input parameter: String[]
       - An array of command line arguments.
    2. Your code will be run at the command line in the following manner

**java DNA data.txt sequence.txt**

where the first argument is the name of the file for the DNA "database" text file and the second is the file with the DNA sequence. You can assume that your program will always be run valid arguments for files that actually exist.

Pseudocode

| Save the data filename (args[0]) to a variable  Save the sequence filename(args[1]) to a variable  Create an array of DNARecords using the readData method  Create a DNA profile using the readSequence method  Loop through each record in the DNARecord array (make sure you handle null values)  if there is profile match, print the matching record  and exit  If after looping through all records there is no match, print out a no match message |
| --- |

* + 1. Example

file data1.txt

| name AGAT AATG TATC  Alicia 5 2 8  Bob 3 7 4  Charlie 6 1 5 |
| --- |

file sequence1.txt

| AGACGGGTTACCATGACTATCTATCTATCTATCTATCTATCTATCTATCACGTACGTACGTATCGAGATAGATAGATAGATAGATCCTCGACTTCGATCGCAATGAATGCCAATAGACAAAA |
| --- |

file sequence2.txt

| GGTACAGATGCAAAGATAGATAGATGTCGTCGAGCAATCGTTTCGATAATGAATGAATGAATGAATGAATGAATGACACACGTCGATGCTAGCGGCGGATCGTATATCTATCTATCTATCTATCAACCCCTAG |
| --- |

* java data1.txt sequence1.txt

**output**

| Matched Alicia: (AGAT = 5, AATG = 2, TATC = 8) |
| --- |

* java data1.txt sequence2.txt

**output**

| No match for (AGAT = 3, AATG = 7, TATC = 5) |
| --- |