**Şevval ÇOLAK**

**string readFile(string filename) {return allText;}** 🡪 I wrote this function to use while reading both the “the\_truman\_show\_script.txt” and “question.txt”. It helped me to refuse code reputation and to have cleaner main. This function gets the file name to be opened as an input. It opens the file. Then it also opens “stop\_words.txt” file and reads all stop words and uses compute\_hash() function to put each word into a hash table. I used a hash table because I will check each words of the script and will only take the not stop-words one. If I would use and array each word of the Truman script (there is almost 13 thousand words) would be check with the array members until it finds one. Since most of the words of this script are not a stop words, anything rather than hash table would take very long time to compare. Then after finishing reading stopWords file it closes the file.  
As a next step function checks if the file name is question.txt or not. Because if it is the question.txt file it will do an extra work. It will create a new file called “filtered\_questions.txt”. In this step function will not read the words coming after “what”, “how” or “which”. Because these question words are causing a problem if they are combined with the words such as “time”, “color”, “many” etc. The second words are also a part of the question word but they are not counted as stop words. So later it will cause a problem while searching the answers in the text. We use the questions’ non-stopwords as a search key, therefore if we search for the color word in the text it will high probably show the wrong sentences. If the question is “what color…?” most of the time the answer sentence does not include the “color” word, only the name of the color such as lavender. Therefore we made this extra filtering in this step.  
On the other case filename can only be Truman script so it reads the word from script computes its hash index, checks if the word is stop word or not. If not stop word add it to the “allText” string. In the end return allText string.

**long long compute\_hash(std::string const& str){return hash\_val;} 🡪** This function used for calculating the index of the word to be put into the hash table. Since we use string hash table, this function makes complex calculation with each char and in the end finds and returns the hash index.

**vector<int> search\_next\_word(vector<int> match\_words, Boyer\_Moore b1, const string& text, string pat) {return temp;} 🡪** This function is getting the previously found words’ starting indexes as a vector as input. Also gets a boyer moore object to use its search function, text to be searched in and a pat to search. In the main we search each question’s first words in the script. Store them in matched\_words vector. Most of the time result of this search returns more than one index. To get closer to the answer sentence we need to search for second words of the question. No need to check second words inside the whole script. We search second words in matched\_words vector. We actually gets the index from that vector, go to this index of the script and then search second word of the question in this sentence. In the end temp vector will be returned. Having less index values than the first search. Later in the main we will use this function in this manner. Give the previously found indexes, make the new search on this limited part. With this way we reduced the number of searches.

**int \* find\_texts\_start(int index, string text) {return a;} 🡪** This function is used only inside search\_next\_word(). It gets the index of the found word’s starting letter. Then find the beginning of this sentence that includes the found word. Even though the name of the function says “find text start” it also finds the end of the sentence. Therefore, returns an array. Since returning array is not possible in C++ I returned a pointer that points to the first index of the array.

**string read\_sentence(int start, int end, string text) {return temp;} 🡪** This function is also only used in search\_next\_word(). It gets the start and end indexes from find\_texts\_start() function’s resulting array and reads the sentence into a string called temp. Returns temp.

**main: key search part🡪** In the main I read the files into strings by calling readFile() function. I split the question string into a 2D string vector. This vector stores the each word of the questions. For example question[0][0] means the first word of the first question, question[1][3] means fourth word of the second question. After that I called the above functions and used boyer moore object to search for words from question. Firstly we search for the first words of the questions, matched\_words vector stores the results of these searches. Then seconds words of the questions searched in previously found sentences, matched\_second\_words vector stores the result. And as a last step we search for third words of the questions in matched\_seconds\_words. This last step only happens if the matched\_second\_words vector’s size is greater than 1.  
Final\_match vector stores the indexes of the answer sentences’ startings. Then by using these indexes we go to the corresponding part of the script read the sentence into answer\_sentences 2D vector.

**1902201 - Şevval ÇOLAK ve 1905387 - Zeynep BABÜR**

**main: finding answers part 🡪** First we cleared punctuation from both question vector and answer\_sentences vector. After that we made set difference between question and answer sentence. With this way we would have the answer word. Because except the answer word question and answer sentences are same. We looped in question and answer sentences vectors in order to compare the words. If the word could not be found in the question vector, we push this word into answer vector by checking the flag. We came across some problems while arranging the indexes, such as reaching to the end of the question vector while answer\_sentence has not. Therefore, we used some if conditions to properly arrange the indexes. We realized some of the different words were not pushed into the answer vector when we reach to the end of the sentence. To solve this we used another flag named endFlag.

**bool IsEqual(string x, string y) {}🡪** To eliminate the differences caused by affixes we created this function. We check the shorter word inside the longer word. If it finds returns true. If the first characters are different it does not control the rest and returns false. We later realized even though the word of the answer\_sentence and question is same, it returned false. Because some of the words contain a hidden “\0” in the end. So to avoid this problem we skipped the last characters.

**main: printing the questions and answers** 🡪 We opened the questions.txt and read it to a string by using getline(). We print first the question and corresponding answer after that.

**Zeynep Babür**

**Division of Labor**

I did the text-searching part using Boyer-Moore’s searching algorithm.

**Algorithm of Boyer-Moore**

The class Boyer-Moore contains all relevant information about the search: a string text, pattern that is going to be searched in the text and a vector badChars that contains the shift table.

In order not to declare Boyer-Moore objects every time, I declared setTxt which changes the pattern that is searched and the text that is searched from.

Search function gets the pattern and returns a vector that contains the starting indexes of the founded patterns. If no pattern is found in the text, an empty vector is returned.

It uses 2 private functions, maxVal and badHeuristic.

* MaxVal compares the values that are send to the function and returns the bigger one. It makes sure that shift value is not negative.
* BadHeuristic fills the shift vector in terms of the pattern. If pattern contains some matching characters, shift table information of that characters are going to be calculated through substraction of the same character’s indexes.

When a pattern is searched, program will calculate the paths length and the texts length. These two information will be used in the shifting operations. BadHeuristic function will be called for updating the shift table to be compatible with the current pattern. Boyer-Moore’s algorithm will check the text starting from the patterns’ last character.

* If all the characters of the pattern are controlled and they’re all matched, starting index of the text that is checked will be pushed into the return vector.
* If there is any match but pattern is not found, index that we contolled is shifted according to the shift table values.
* If there is not any match, table is shifted in terms of the pattern length.

**Libraries that we used in the project🡪** #include <iostream>

#include <cctype>

#include <fstream>

#include <sstream>

**Average run time 🡪** 70ms

**Average correctness 🡪** We only tried the code with limited number of test questions. We expect %80 correctness. It perfectly works with the given questions.txt file.