New York University in Abu Dhabi
Saad Teeti

In this project, I have developed an application that helps a user find an English word or words in a chosen dictionary. I have created the program using the data structures elements and topics discussed in class such as the use of loops, vectors, function definition, recursive function etc.

The program runs in the way that it takes the name of the dictionary the user wants to look up and the number of words that the user wants to be found in the command line. The program checks if there is such a dictionary and if there is one, it opens it. Now when the dictionary is opened properly, the program asks the user to input specific word(s) he/she wants to find. There are 3 possible options for the outcome:

1) Full-Word-Search: User types a word and the program tells the word is found in the chosen dictionary (example: word)

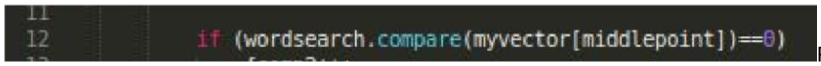
2) Prefix-Search: User only types the beginning of the word and puts an asterisk on it, and the program lists all the words of the dictionary that start with the provided prefix (example: wor*)

3) Wildcard-Search: User puts a question mark instead of one letter, and the program lists all possible the words of the dictionary that could match the searched word (example: w?rd)

In each option, the program should report if the word/s was/were found and the number of word comparisons the program performed.

The first move was to copy all words from the chosen dictionary into a string vector, called myvector as we can make faster, flexible and more efficient operations using vectors.
I made 3 different functions for each outcome: nomalbinarysearch (for full-word search), starbinary (for prefix search) and quesbinary (for wildcard-search). All three are similar, but they vary according to the wanted performance. Starbinary and quesbinary are basically more specific normalbinarysearch functions.

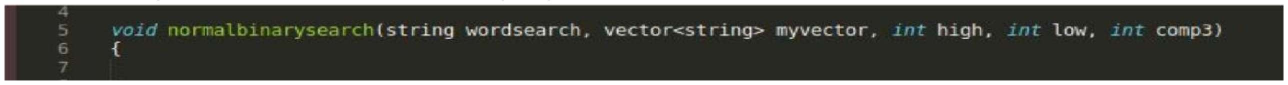
One of the functions that I used is *compare(),* which member function of string class. This function compares the value of the string, which is user’s word stored as “wordsearch” to the sequence of characters specified by the arguments, which is the word stored in “myvector[middlepoint]” in here. If these two are equal, their comparison will be equal to 0. If they are not equal, then one value is bigger than the other, and this investigated in each binary search function on its own.



```
        if (wordsearch.compare(myvector[middlepoint])==0)
```

Figure 0

Normalbinarysearch takes in the following arguments:

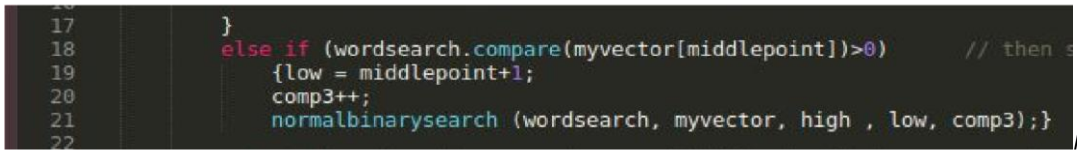


```
void normalbinarysearch(string wordsearch, vector<string> myvector, int high, int low, int comp3)
{
```

Figure 1

where wordsearch is the user-inputTed word, myvector is a vector containing all words from the dictionary, high are low are the start and end points of the myvector and comp3 is the number of times the searched word is compared inside of the myvector.

One of the important parts of our assignment was figuring out how to use the binary search. The description of the main idea was provided in the Data Structures & Algorithms book (pp 395). The binary search finds the position of a key value within a sorted array. Each time our binary search function runs, it halfs the dictionary vector and choses to search only through the half to which the key should belong to, until the key is found.
The value of the middle of array is found by calculating the index of the middle point as middle point=(how+low)/2. The binary search compares user's word with the word in the middle of the vector. If the words do not match, then the program assesses if the word belongs to the upper half or lower half of the vector. At this point, the normalbinarysearch function is called again (recursion), so the search through the halved vectors goes on again, all until the word is found.

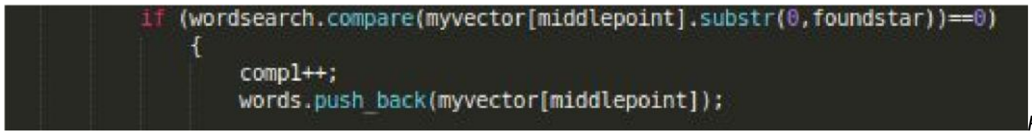


```
            }
            else if (wordsearch.compare(myvector[middlepoint])>0)      // then s
                {low = middlepoint+1;
                comp3++;
                normalbinarysearch (wordsearch, myvector, high , low, comp3);}
```

Figure 2

It is important to explain that each time a vector is halved, either high or low value will change. In figure 2, the program recognised that the word belongs to the upper half, hence the high (end point) value will stay the same while the middlepoint+1 will become the new low (start point) value. Each time word is compared, the comparison variable comp3 is incremented so that the total number of comparisons can be calculated and printed to the user.

In **starbinary**, one more argument was added: that is the vector to store the found words because I expected to have one or more than one word matching. The function uses the same logic for binary search as in normalbinarysearch. The difference is that only a part of the word is being compared, hence I used the "substr" comparison. In substr(0, foundstar), 0 is the beginning of the word while foundstar is the index where the asterisk is found. Futher in Figure 3., I can see the comp1, which counts the comparisons all until the first match. All words found were stored into vector words to be displayed later.

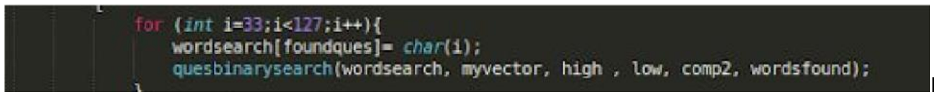


```
        if (wordsearch.compare(myvector[middlepoint].substr(0,foundstar))==0)
            {
                comp1++;
                words.push_back(myvector[middlepoint]);
```

Figure 3

For the moment the program finds the first word, I created a special loop. This loops checks for all neighboring words above and below our found word to see which of those satisfy the user-provided prefix.

In **quesbinary,** the main logic of binary search from normalbinarysearch is still used. The comparison value here accounts for all comparisons executed. The additional vector called "wordsfound" is added, as I are expecting more than one word to be matching the user's word. What is specific in this Wildcard-search part are the following lines in the main function:

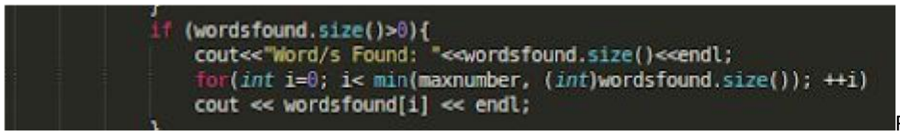


```
        for (int i=33;i<127;i++){
            wordsearch[foundques]= char(i);
            quesbinarysearch(wordsearch, myvector, high , low, comp2, wordsfound);
```

Figure 4

Foundques is the index where the question mark is found in the word. The second line changes the question mark into a character. The character depends on the value of i, which I found in the ASCII chart (see references). Then, the quesbinarysearch is called for every value of char(i).

In starbinary and quesbinary, it might be the case that I found more or less words to match the user-inputed words than the maximum number <MaxNumOfWordsInOutput> entered as the command of that user wants us to print. In this case, I did not want to overprint or to print invalid value, so I created the following:



```
        if (wordsfound.size()>0){
            cout<<"Word/s Found: "<<wordsfound.size()<<endl;
            for(int i=0; i< min(maxnumber, (int)wordsfound.size()); ++i)
            cout << wordsfound[i] << endl;
```

Figure 5

In this loop, I compared the number of the words found and the number of words that the user wants. Function “min” determines which number is smaller and only prints that amount of words.

If the input is invalid, I created a while loop that checks for all possible invalid inputs. If notifies the user that his input is invalid and requires him to type a new word for search.
If the user’s word is not found in any of the possible outcomes, then the program displays “Word not found”.