Title: Searching User-given Queries from two Separate Dictionaries
Authors: Liyan Ibrahim and Muhammad Wajahat Mirza

This assignment digs into three main cases of queries: Full-word Search, Prefix Search, and Wildcard Search. The code written by the authors of this report is split into two .cpp files: my dictionary file and binary search file. Description of different functions used in these two .cpp files is as followed:

Binary Search File: binary_search function

This binary_search function is used to search through the entire vector - vector<string> dic - to find the query given by the user. It takes three arguments: vector of strings, user input as a string, and reference to the counter which counts the number of comparisons made in the process of doing a binary search. Three integers are initialized to store the starting (left), middle, and ending (right) index values of the vector. For the size of the left side of the vector, it is initialized to 0 whereas, for the size of the right side of the vector, one is subtracted from the size of the vector. These initializations are valid because to start the binary search, we assume the left is 0 so we start from the first element and we then change the values for the left and the right in order to iterate through the entire vector
For the middle index of the vector, the following code of line is used:

```
18          middle = left + ((right - left)/2);
```

The above-given formula is used to find the middle to avoid overflow and out of range error. Subtracting left from the right will yield a smaller value, thus, no overflow can happen since every step of the operation is bounded by the value of the right. A while loop that will iterate as long as the left is smaller than the right. The counter is incremented because it counts the number of searches made and every time the loop is entered, a search is made. If the value of the user input matches the middle position is the vector (=0), then we return the value of the middle (the index of where the match is found in the vector of the dictionary strings). If the value of the first character of the user input is greater than the middle value (further down the alphabet), this means that the user input is further along in the vector, therefore, the left is now middle+1 (moves to the right). If neither of these conditions is true (the first character of the string entered is smaller than the middle value), then the right will be middle-. If this condition is not met, then this implies that the left value is equal to the string the user inputs, then the left is returned.

My Dictionary File: main function

In the main function, two vectors are used: vector “dict” to store all the words found in the user-given dictionary (whether dictionary1 or dictionary2) and the vector “output” that contains the words matched. From the vector output, the number of words would be displayed on the screen as per the value of the user, given after “-l.” In the main function, it is checked whether the file has been opened properly or not. If not, generate error output and Exit the code. Otherwise, the content of the user given dictionary (either 1 or 2) is

read into the vector dict through a while loops. Once the content of the file is read into the vector, file is closed.

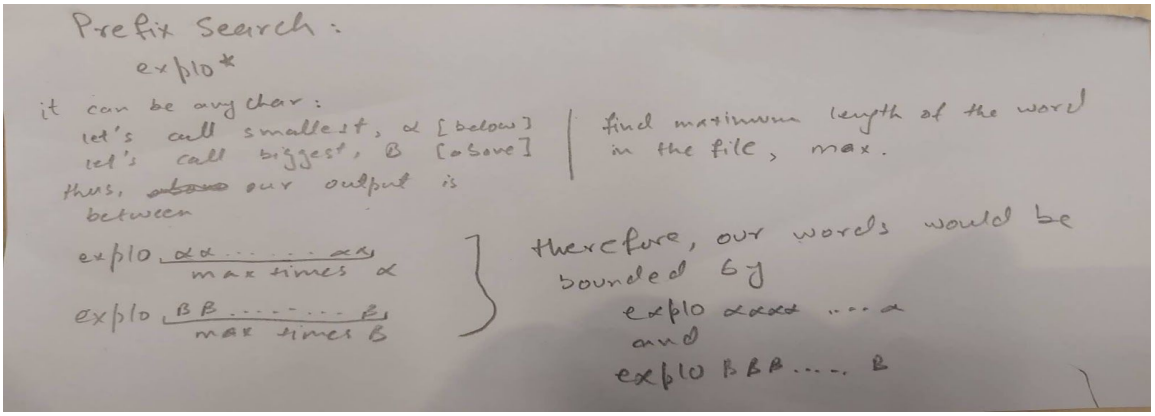
```
73      string str1;
74      while (fin)
75      {
76          fin >>str1;
77          dict.push_back(lower_string(str1));
78          max_length = max(max_length, (int)str1.length());
79      }
80      fin.close();
```

An infinite loop is then set up to constantly prompt the user for a query and is only terminated once the user types “quit.” The query of the user is stored in the variable “userI” and if “userI” is “quit” then the program is terminated. The function lower_string (described later) is used to de-capitalize “userI” as our code is case sensitive i.e. “explore” will be different “Explore.”

Full-word Search: In order to carry out a full-word search, our code checks whether the user input -“userI”- contains “*” or “?” (This condition is checked using “string::npos”). If not, code proceeds with full-word search using binary search function. Using the return value of binary search function, left, our code compares the “userI” with the “dict[left]” and if it is equal to 0, a message “Word Found” is displayed along with the number of comparisons made in the binary search to find the word. This process is carried out by calling the binary search function which returns the integer value of left. If the comparison of “userI” with the “dict[left]” is not zero, “word not found” is displayed.

```
103         if (userI.find('*')==string::npos && userI.find("?")==string::npos)
104         {
105             int left = binary_search(dict, userI, count);
106             if(userI.compare(dict[left]) == 0)
107             {
108                 cout << "Word found" << endl;
109                 cout << count << " word comparisons carried out" << endl;
110             }
111             else
112                 cout << "Word not found" << endl;
113         }
```

Prefix Search: In order to carry out a prefix search, we obtain two “userI” substrings from 0 to the position of “*”. Our code then iterates through these two substrings and are replaced with values from char 0 to 255. One substring starts from char 0 (below) and the other one starts from char 255 (above). These two substrings are “userI” input for the binary search: one substring from the above and one substring from the below. These substrings are then compared with the words in the vector dict. Refer to the following handwritten description to make an understanding of the above and below.




```
121             for (int i = 0; i < max_length - position;i++)
122             {
123                 string_1 += (char)0;
124                 string_2 += (char)255;
125             }
```

Between the range of below and above, those prefix-matched words are pushed back into the Output vector which is looped through a FOR loop to print "-l" number of words on the screen.

Wildcard Search: In order to carry out a wildcard search, our code finds the position of the index where "?" is present in the "userI." A FOR loop is iterated at that position of "?" which checks and replaces "?" with each of the ASCII code Char code between 33 to 126 (refer to Appendix A for ASCII Table). Each replacement of "?" with an ASCII code char is run through the binary search to compare with the words stored in the vector dict. If there is a match, that word is pushed back into the output vector which in turn goes through a FOR loop to display the number of matches depending on the user-given value of "-l." For instance, for a query "explo?e", our algorithm will check "explo!e", then "explo"e", and all the way to the 126th ASCII code. When it reaches "explo**d**e," and if a compare match is found with the vector dict, it will be pushed into vector output. For instance, for a query "explo?e", our algorithm will check "explo!e", then "explo"e", and all the way to the 126th ASCII code. As our algorithm iterates through all ASCII code, number of comparisons made are higher than the other two searches.

Asciitolower and lower string: These two functions that lie outside the main function check whether the first letter of the "userI" is between A to Z. If so, they perform the lower case operation to de-capitalize all the words as our code is case sensitive. For Asciitolower, we used Stack Overflow code available here.

Challenges and limitations:

In designing our algorithms and coding, we were challenged to perform the binary operation in the minimal comparisons. We achieved our goal for Full word and prefix search, however, our code for wildcard search makes too many comparisons to generate the result.

Appendices

Appendix A: This is the ASCII Code table that we referred to for our algorithmic coding.

CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX
[NUL]	0	00		32	20	@	64	40	`	96	60
[SOH]	1	01	!	33	21	A	65	41	a	97	61
[STX]	2	02	"	34	22	B	66	42	b	98	62
[ETX]	3	03	#	35	23	C	67	43	c	99	63
[EOT]	4	04	$	36	24	D	68	44	d	100	64
[ENQ]	5	05	%	37	25	E	69	45	e	101	65
[ACK]	6	06	&	38	26	F	70	46	f	102	66
[BEL]	7	07	'	39	27	G	71	47	g	103	67
[BS]	8	08	(	40	28	H	72	48	h	104	68
[HT]	9	09	)	41	29	I	73	49	i	105	69
[LF]	10	0A	*	42	2A	J	74	4A	j	106	6A
[VT]	11	0B	+	43	2B	K	75	4B	k	107	6B
[FF]	12	0C	,	44	2C	L	76	4C	l	108	6C
[CR]	13	0D	-	45	2D	M	77	4D	m	109	6D
[SO]	14	0E	.	46	2E	N	78	4E	n	110	6E
[SI]	15	0F	/	47	2F	O	79	4F	o	111	6F
[DLE]	16	10	0	48	30	P	80	50	p	112	70
[DC1]	17	11	1	49	31	Q	81	51	q	113	71
[DC2]	18	12	2	50	32	R	82	52	r	114	72
[DC3]	19	13	3	51	33	S	83	53	s	115	73
[DC4]	20	14	4	52	34	T	84	54	t	116	74
[NAK]	21	15	5	53	35	U	85	55	u	117	75
[SYN]	22	16	6	54	36	V	86	56	v	118	76
[ETB]	23	17	7	55	37	W	87	57	w	119	77
[CAN]	24	18	8	56	38	X	88	58	x	120	78
[EM]	25	19	9	57	39	Y	89	59	y	121	79
[SUB]	26	1A	:	58	3A	Z	90	5A	z	122	7A
[ESC]	27	1B	;	59	3B	[	91	5B	{	123	7B
[FS]	28	1C	<	60	3C	\	92	5C	|	124	7C
[GS]	29	1D	=	61	3D	]	93	5D	}	125	7D
[RS]	30	1E	>	62	3E	^	94	5E	~	126	7E
[US]	31	1F	?	63	3F	_	95	5F	[DEL]	127	7F