

Name: Bhargavi Dhage

Class: T.Y.CSE CORE 1

Roll No: 2193079

Enrollment No: MITU19BTCS0084

ASSIGNMENT_1

AIM:

To write a program to generate random numbers in an array and to sort them using Quick sort.

THEORY:

Quicksort is a type of divide and conquer algorithm for sorting an array, based on a partitioning routine. Applied to a range of at least two elements, partitioning produces a division into two consecutive non empty sub-ranges, in such a way that no element of the first sub-range is greater than any element of the second sub-range.

Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

This then results in a sorted array.

- **Best Case Complexity** - In Quicksort, the best-case occurs when the pivot element is the middle element or near to the middle element. The best-case time complexity of quicksort is **$O(n \cdot \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of quicksort is **$O(n \cdot \log n)$** .
- **Worst Case Complexity** - In quick sort, worst case occurs when the pivot element is either greatest or smallest element. Suppose, if the pivot element is always the last element of the array, the worst case would occur when the given array is sorted already in ascending or descending order. The worst-case time complexity of quicksort is **$O(n^2)$**

Quicksort Algorithm:

1. QUICKSORT (array A, start, end)
2. {
3. if (start < end)
4. {
5. p = partition(A, start, end)
6. QUICKSORT (A, start, p - 1)
7. QUICKSORT (A, p + 1, end)
8. }
9. }

Partition Algorithm:

The partition algorithm rearranges the sub-arrays in a place.

1. PARTITION (array A, start, end)
2. {
3. pivot ? A[end]
4. i ? start-1
5. for j ? start to end -1 {
6. do if (A[j] < pivot) {
7. then i ? i + 1
8. swap A[i] with A[j]
9. }}
10. swap A[i+1] with A[end]
11. return i+1
12. }

CODE:

```
#program of quick sort
```

```
# to generate random variables
```

```
import random
```

```
# function for quick sort
```

```
# returns the sorted list
```

```
def partition(lis, left, right):
```

```
    i = (left-1)
```

```
    pivot = lis[right]
```

```
    for j in range(left, right):
```

```
        if lis[j] <= pivot:
```

```
            i = i+1
```

```
            lis[i], lis[j] = lis[j], lis[i]
```

```
    lis[i+1], lis[right] = lis[right], lis[i+1]
```

```
    return (i+1)
```

```
# Function to do Quick sort
```

```
def quicksort(lis, left, right):
```

```
    if len(lis) == 1:
```

```
        return lis
```

```
    if left < right:
```

```
pa = partition(lis, left, right)
```

```
quicksort(lis, left, pa-1)
```

```
quicksort(lis, pa+1, right)
```

```
size = int(input("Enter the size of list :"))
```

```
lower =int(input("Enter the lower range of random numbers to fill the list :"))
```

```
higher =int(input("Enter the higher range of random numbers to fill the list :"))
```

```
lis = []
```

```
# always making it lower for smaller number
```

```
if(lower>higher):
```

```
    temp = lower
```

```
    lower = higher
```

```
    higher = temp
```

```
for i in range(0,size,1):
```

```
    lis.append(random.randint(lower,higher))
```

```
# to print the list
```

```
print("\n\nThe unsorted list",end='\n')
```

```
print(lis)
```

```
quicksort(lis,0,len(lis)-1)
```

```
# to print the list
```

```
print("\n\nThe sorted list",end='\n')
```

```
print(lis)
```

OUTPUT:

```
Shell
Enter the size of list :12
Enter the lower range of random numbers to fill the list :10
Enter the higher range of random numbers to fill the list :100

The unsorted list
[44, 35, 38, 54, 76, 78, 68, 81, 76, 47, 20, 29]

The sorted list
[20, 29, 35, 38, 44, 47, 54, 68, 76, 76, 78, 81]
> >
> >
```

CONCLUSION:

The unsorted randomly generated numbers in the array were sorted using quick sort to form an array of sorted numbers in ascending order.

Name: Bhargavi Dhage

Class: T.Y.CSE CORE 1

Roll No: 2193079

Enrollment No: MITU19BTCS0084

ASSIGNMENT - 2

AIM:

To write a program to generate random numbers in an array and to sort them using Merge sort.

THEORY:

Merge sort uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves according to the values of the elements.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

- **Best Case Complexity** - It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of merge sort is **$O(n \cdot \log n)$** .
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of merge sort is **$O(n \cdot \log n)$** .
- **Worst Case Complexity** - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of merge sort is **$O(n \cdot \log n)$** .

Merge sort Algorithm:

1. MERGE_SORT(arr, beg, end)
- 2.
3. **if** beg < end
4. set mid = (beg + end)/2
5. MERGE_SORT(arr, beg, mid)
6. MERGE_SORT(arr, mid + 1, end)
7. MERGE (arr, beg, mid, end)
8. end of **if**
- 9.
10. END MERGE_SORT

CODE:

```
#program of merge sort
```

```
# to generate random variables
```

```
import random
```

```
def merge(lis, l, m, r):
```

```
    n1 = m - l + 1
```

```
    n2 = r - m
```

```
    # create temp arrays
```

```
    L = [0] * (n1)
```

```
    R = [0] * (n2)
```

```
    # Copy data to temp arrays L[] and R[]
```

```
    for i in range(0, n1):
```

```
        L[i] = lis[l + i]
```

```
    for j in range(0, n2):
```

```
        R[j] = lis[m + 1 + j]
```

```
    # Merge the temp arrays back into lis[l..r]
```

```
    i = 0    # Initial index of first subarray
```

```
    j = 0    # Initial index of second subarray
```



```
k = l    # Initial index of merged subarray
```

```
while i < n1 and j < n2:
```

```
    if L[i] <= R[j]:
```

```
        lis[k] = L[i]
```

```
        i += 1
```

```
    else:
```

```
        lis[k] = R[j]
```

```
        j += 1
```

```
    k += 1
```

```
# Copy the remaining elements of L[], if there
```

```
# are any
```

```
while i < n1:
```

```
    lis[k] = L[i]
```

```
    i += 1
```

```
    k += 1
```

```
# Copy the remaining elements of R[], if there
```

```
# are any
```

```
while j < n2:
```

```
    lis[k] = R[j]
```

```
    j += 1
```

```
    k += 1
```

```
# l is for left index and r is right index of the
```

```
# sub-array of arr to be sorted
```

```
def mergesort(lis, l, r):
```

```
    if l < r:
```

```
# Same as (l+r)//2, but avoids overflow for
```

```
# large l and h
```

```
m = l+(r-l)//2
```

```
# Sort first and second halves
```

```
mergesort(lis, l, m)
```

```
mergesort(lis, m+1, r)
```

```
merge(lis, l, m, r)
```

```
size = int(input("Enter the size of list :"))
```

```
lower =int(input("Enter the lower range of random numbers to fill the list :"))
```

```
higher =int(input("Enter the higher range of random numbers to fill the list :"))
```

```
lis = []
```

```
# always making it lower for smaller number
```

```
if(lower>higher):
```

```
    temp = lower
```

```
    lower = higher
```

```
    higher = temp
```

```
for i in range(0,size,1):
```

```
    lis.append(random.randint(lower,higher))
```

```
# to print the list
```

```
print("\n\nThe unsorted list",end='\n')
```

```
print(lis)
```

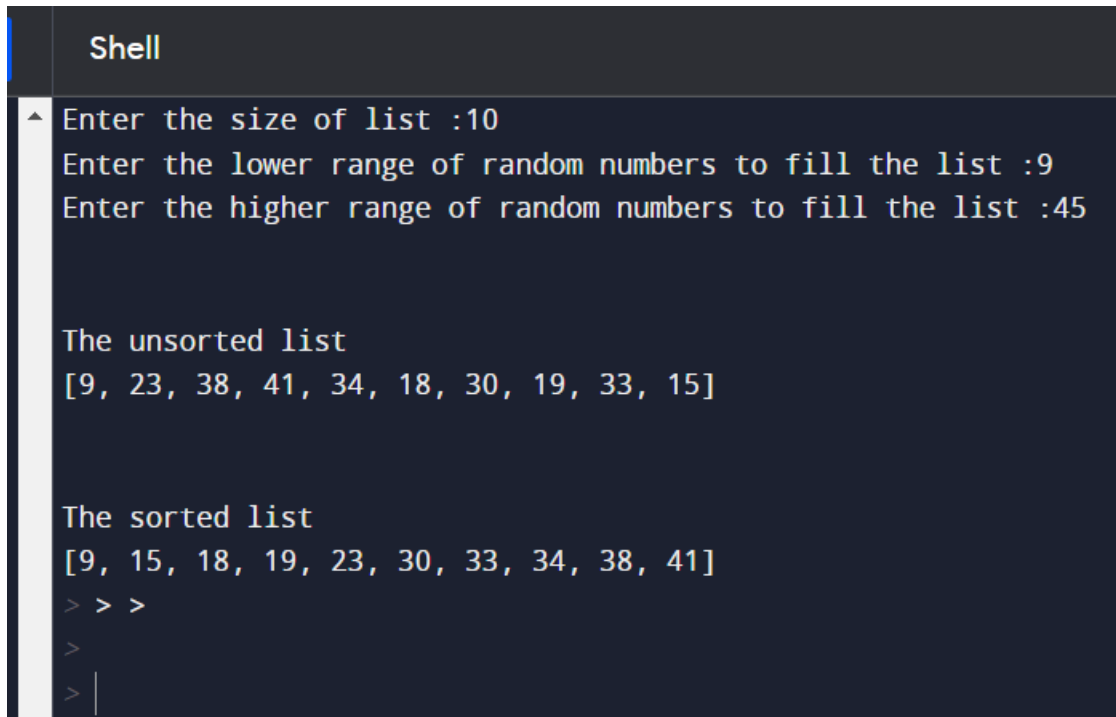
```
mergesort(lis,0,len(lis)-1)
```

to print the list

```
print("\n\nThe sorted list",end='\n')
```

```
print(lis)
```

OUTPUT:



```
Shell
^ Enter the size of list :10
Enter the lower range of random numbers to fill the list :9
Enter the higher range of random numbers to fill the list :45

The unsorted list
[9, 23, 38, 41, 34, 18, 30, 19, 33, 15]

The sorted list
[9, 15, 18, 19, 23, 30, 33, 34, 38, 41]
> > >
>
> |
```

CONCLUSION:

The unsorted randomly generated numbers in the array were sorted using Merge sort to form an array of sorted numbers in ascending order.

Name: Bhargavi Dhage

Class: T.Y.CSE CORE 1

Roll No: 2193079

Enrollment No: MITU19BTCS0084

ASSIGNMENT - 3

AIM:

To write a program to find the sum of subsets for a given set of numbers which match up to the target value using dynamic programming.

THEORY:

The problem is given an A set of integers a_1, a_2, \dots, a_n up to n integers. The question arises that is there a non-empty subset such that the sum of the subset is given as M integer? For example, the set is given as $[5, 2, 1, 3, 9]$, and the sum of the subset is 9; the answer is YES as the sum of the subset $[5, 3, 1]$ is equal to 9.

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Algorithm:

1. SumOfSubSets(index, weight, total,W,x,M)
2. if promising(index, weight)
3. if (weight == M)
4. i=0 loop till i =N
5. If x[i] == 1
6. print(W[i+1]," ','Solution is ', x)
7. End if
8. End loop
9. End if
- 10.End if
- 11.Else
- 12.x[index] = 1
- 13.SumOfSubSets(index+1, weight + W[index+1], total - W[index+1],W,x,M)
- 14.x[index] = 0
- 15.SumOfSubSets(index+1, weight, total - W[index+1],W,x,M)
- 16.End Else
- 17.End SumOfSubSets

Promising Algorithm:-

The Promising function returns if the index value is useable or not

1. Promising (array A, start, end)
2. If index > N-1
3. Return False
4. End if
5. Else if index == N-1
6. return ((weight + total >= M) and (weight == M))
7. End Else if
8. Else
9. return ((weight + total >= M) and ((weight == M) or (weight + W[index+1] <= M)))

10.End Else

11.End promising

CODE:

```
def SumOfSubSets(index, weight, total, W, x, M):  
    if promising(index, weight):  
  
        if weight == M:  
            for i in range(N-1):  
                if x[i] == 1:  
  
                    print(W[i+1], end=" ")  
                print("")  
                print('Solution is ', x)  
                print("")  
  
            else:  
                x[index] = 1  
                SumOfSubSets(index+1, weight + W[index+1], total - W[index+1], W, x, M)  
  
                x[index] = 0  
                SumOfSubSets(index+1, weight, total - W[index+1], W, x, M)  
  
def promising(index, weight):  
    if index > N-1:  
        return False
```

```
elif index == N-1:
```

```
    return ((weight + total >= M) and (weight == M))
```

```
else:
```

```
    return ( (weight + total >= M) and ( (weight == M) or (weight + W[index+1] <= M) ))
```

```
size = int(input("Enter the size of subset :"))
```

```
W = []
```

```
for i in range (0,size,1):
```

```
    print("Enter element ",i+1," :", end = "")
```

```
    W.append(int(input()))
```

```
W.insert(0,0)
```

```
N = len(W)
```

```
M = int(input("Enter the target value :"))
```

```
total = sum(W)
```

```
index = 0
```

```
weight = 0
```

```
x = [0] * (N-1)
```

```
print("Solutions:")
```

```
SumOfSubSets(index, weight, total,W,x,M)
```

OUTPUT:

```
Shell
Enter the size of subset :4
Enter element 1 :1
Enter element 2 :2
Enter element 3 :3
Enter element 4 :4
Enter the target value :4
Solutions:
1 3
Solution is [1, 0, 1, 0]

4
Solution is [0, 0, 0, 1]

> |
```

CONCLUSION:

The solution of the given sum of subset problem were found using dynamic programming.

Name: Bhargavi Dhage

Class: T.Y.CSE CORE 1

Roll No: 2193079

Enrollment No: MITU19BTCS0084

ASSIGNMENT - 4

AIM:

To write a program to find the shortest path while covering all cities in the travelling salesman problem.

THEORY:

Travelling Salesman Problem (TSP): Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

These are the following steps, which we use to implement the TSP

1. We consider a city as the starting and ending point. We can use any city as a starting point because the route is cyclic.
2. In the DFS way, we start traversing from the source to its adjacent nodes.
3. Find the cost of each traversal and keep track of minimum cost and keep on updating the value of minimum cost stored value.
4. In the end, return the permutation with minimum cost.

Backtracking is an algorithmic technique where the goal is to get all solutions to a problem using the brute force approach. It consists of building a set of all the solutions incrementally. Since a problem would have constraints, the solutions that fail to satisfy them will be removed. Backtracking algorithm uses the depth-first search method. When it starts exploring the solutions, a bounding function is applied so that the algorithm can check if the so-far built solution satisfies the constraints. If it does, it continues searching. If it doesn't, the branch would be eliminated, and the algorithm goes back to the level before.

Algorithm:

Data: s : starting point; N : a subset of input cities; $dist()$: distance among the cities

Result: $Cost$: TSP result

$Visited[N] = 0;$

$Cost = 0;$

Procedure TSP(N, s)

- $Visited[s] = 1;$
- if** $|N| = 2$ **and** $k \neq s$ **then**
 - $Cost(N, k) = dist(s, k);$
 - Return** $Cost;$
- else**
 - for** $j \in N$ **do**
 - for** $i \in N$ **and** $visited[i] = 0$ **do**
 - if** $j \neq i$ **and** $j \neq s$ **then**
 - $Cost(N, j) = \min (TSP(N - \{i\}, j) + dist(j, i))$
 - $Visited[j] = 1;$
 - end**
 - end**
 - end**
- end**
- Return** $Cost;$

end

CODE:

```
answer = []
```

```
sol = []
```

```
# Function to find the minimum weight
```

```
def tsp(graph, v, currPos, n, count, cost, order):
```

```
    if (count == n and graph[currPos][0]):
```

```
        answer.append(cost + graph[currPos][0])
```

```
        order.append(0)
```

```
        lsc = order.copy()
```

```
        sol.append(lsc)
```

```
        order.pop()
```

```
        return
```

```
# BACKTRACKING STEP
```

```
# Loop to traverse the adjacency list
```

```
# of currPos node and increasing the count
```

```
# by 1 and cost by graph[currPos][i] value
```

```
for i in range(n):
```

```
    if (v[i] == False and graph[currPos][i]):
```

```
        # Mark as visited
```

```
        v[i] = True
```

```
        order.append(i)
```

```
        tsp(graph, v, i, n, count + 1,
```

```
            cost + graph[currPos][i], order)
```

```
# Mark ith node as unvisited
```

```
v[i] = False
```

```
order.pop()
```

```
graph = []
```

```
r = []
```

```
n = int(input("Enter the number of cities :"))
```

```
for row in range(0,n,1):
```

```
    r=[]
```

```
    #making 0s
```

```
    for col in range(0,row+1,1):
```

```
        r.append(0)
```

```
    for col in range(row+1,n,1):
```

```
        ss = "Enter the distance between city " + str(row) + " and city "+str(col)+" :"
```

```
        r.append(int(input(ss)))
```

```
graph.append(r)
```

```
#making the mirror image
```

```
for row in range(0,n,1):
```

```
    for col in range(0,n,1):
```

```
        graph[col][row] = graph[row][col]
```

```
# Boolean array to check if a node
```

```
# has been visited or not
```

```
v = [False for i in range(n)]
```

```
# Mark 0th node as visited
```

```
v[0] = True
```

```
order = [0]
```

```
# Find the minimum weight Hamiltonian Cycle
```

```
tsp(graph, v, 0, n, 1, 0, order)
```

```
# ans is the minimum weight Hamiltonian Cycle
```

```
short = min(answer)
```

```
option = answer.index(short)
```

```
print("\n\nAnswer:", short, " is the least distance on path ", end = "")
```

```
print(sol[option])
```

OUTPUT:

```
Shell
^ Enter the number of cities :5
Enter the distance between city 0 and city 1 :1
Enter the distance between city 0 and city 2 :2
Enter the distance between city 0 and city 3 :3
Enter the distance between city 0 and city 4 :4
Enter the distance between city 1 and city 2 :4
Enter the distance between city 1 and city 3 :3
Enter the distance between city 1 and city 4 :2
Enter the distance between city 2 and city 3 :1
Enter the distance between city 2 and city 4 :1
Enter the distance between city 3 and city 4 :2

Answer: 8 is the least distance on path [0, 1, 4, 2, 3, 0]
> > > |
```

CONCLUSION:

The solution of the given Travelling salesman problem was found using backtracking approach.

Name: Bhargavi Dhage

Class: T.Y.CSE CORE 1

Roll No: 2193079

Enrollment No: MITU19BTCS0084

ASSIGNMENT - 5

AIM:

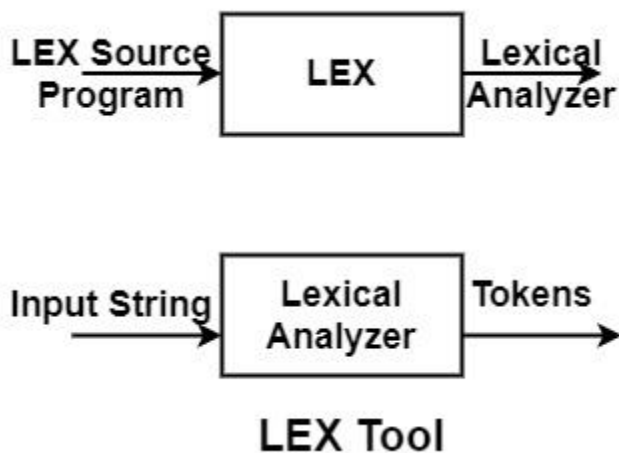
To write a lex program to find if the character entered is a letter or digit.

THEORY:

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

LEX is a program generator designed for lexical processing of character input/output stream. Anything from simple text search program that looks for pattern in its input-output file to a C compiler that transforms a program into optimized code.

In program with structure input-output two tasks occurs over and over. It can divide the input-output into meaningful units and then discovering the relationships among the units for C program (the units are variable names, constants, and strings). This division into units (called tokens) is known as lexical analyzer or LEXING. LEX helps by taking a set of descriptions of possible tokens n producing a routine called a lexical analyzer or LEXER or Scanner.



CODE:

%{

%}

letter [a-zA-Z]

digit [0-9]

id {letter}({letter}|{digit})*

num {digit}+(\.{digit}+)?

%%

if|else|then|while|do {printf ("Keyword");}

{num} {printf ("Number");}

{id} {printf ("Identifier");}

%%

int main()

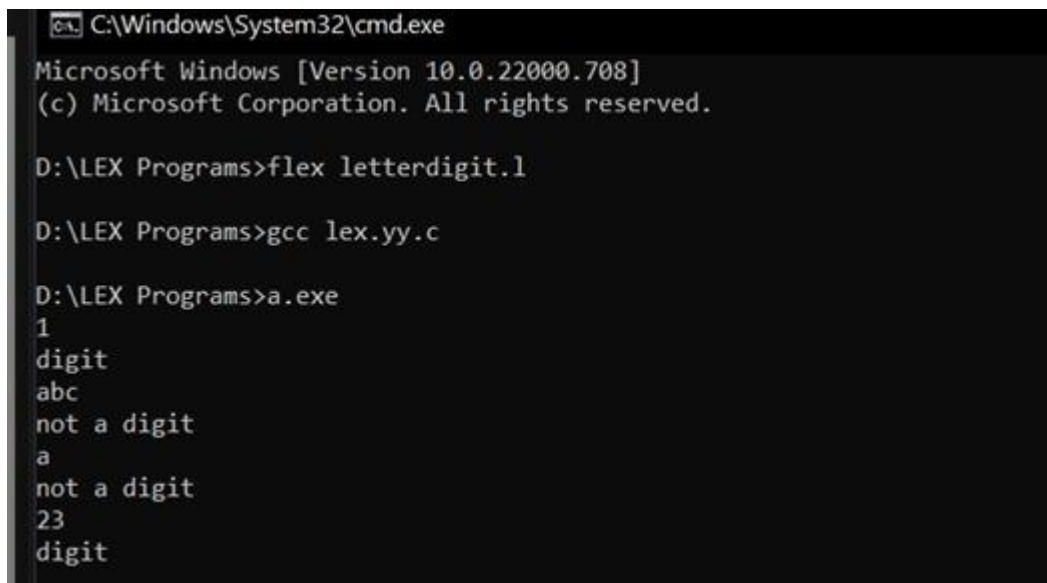
{ yylex();

return 0;

}

int yywrap(){return(1);}

OUTPUT:

A screenshot of a Windows command prompt window. The title bar at the top reads 'C:\Windows\System32\cmd.exe'. The window content shows the following text: 'Microsoft Windows [Version 10.0.22000.708]' followed by '(c) Microsoft Corporation. All rights reserved.' on the next line. The prompt 'D:\LEX Programs>' is followed by the command 'flex letterdigit.l'. The next line shows the prompt 'D:\LEX Programs>' followed by 'gcc lex.yy.c'. The following line shows the prompt 'D:\LEX Programs>' followed by 'a.exe'. The output of the program is displayed on the next lines: '1', 'digit', 'abc', 'not a digit', 'a', 'not a digit', '23', and 'digit'.

CONCLUSION:

The lex program to find if the character entered is a letter or digit was successfully written and executed.

Name: Bhargavi Dhage

Class: T.Y.CSE CORE 1

Roll No: 2193079

Enrollment No: MITU19BTCS0084

ASSIGNMENT - 6

AIM:

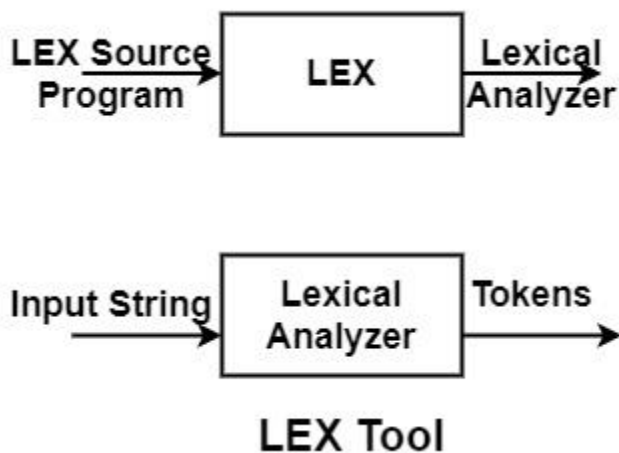
To write a lex program to find if the word entered is a verb or not a verb.

THEORY:

Lexical analysis is the first phase of a compiler. It takes modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

LEX is a program generator designed for lexical processing of character input/output stream. Anything from simple text search program that looks for pattern in its input-output file to a C compiler that transforms a program into optimized code.

In program with structure input-output two tasks occurs over and over. It can divide the input-output into meaningful units and then discovering the relationships among the units for C program (the units are variable names, constants, and strings). This division into units (called tokens) is known as lexical analyzer or LEXING. LEX helps by taking a set of descriptions of possible tokens n producing a routine called a lexical analyzer or LEXER or Scanner.



CODE:

%{

%}

%%

[|t|+

is |

am |

are |

were |

was |

be |

being |

been |

do |

does |

did |

will |

would |

should |

can |

could |

has |

have |

had |

go {printf("%s: is a verb\n", yytext);}

[a-zA-Z]+ {printf("%s: is not a verb\n", yytext);}

.\|\\n {ECHO ;}

%%

int main()

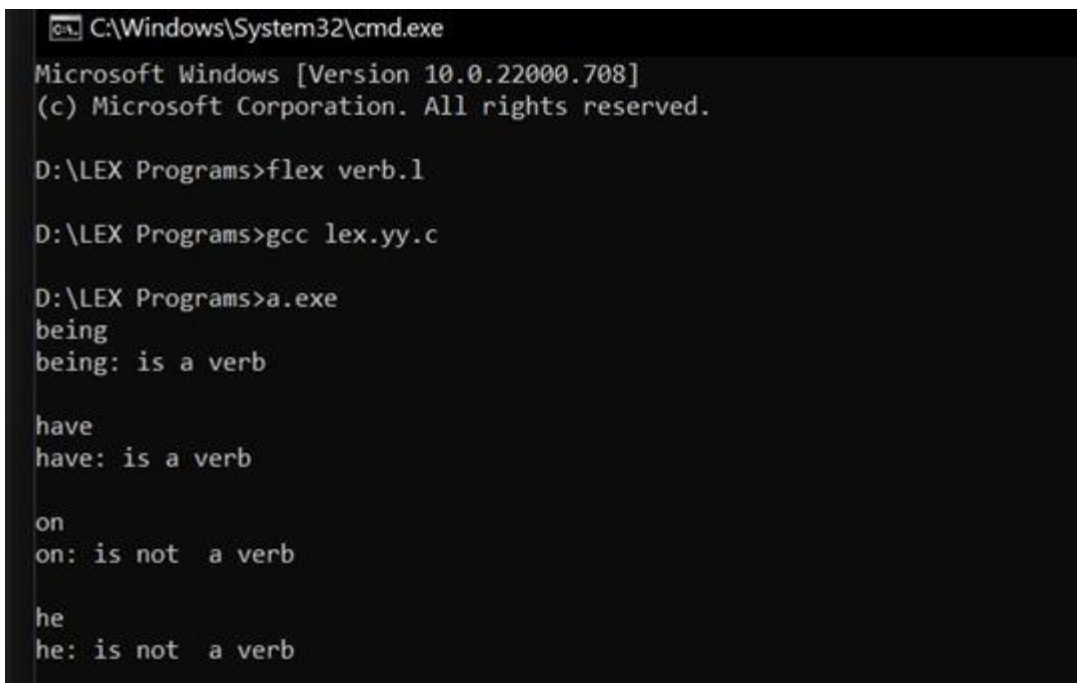
{ yylex();

return 0;

}

int yywrap(){return(1);}

OUTPUT:



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.22000.708]
(c) Microsoft Corporation. All rights reserved.

D:\LEX Programs>flex verb.l

D:\LEX Programs>gcc lex.yy.c

D:\LEX Programs>a.exe
being
being: is a verb

have
have: is a verb

on
on: is not a verb

he
he: is not a verb
```

CONCLUSION:

The lex program to find if the word entered is a verb or not a verb was successfully written and executed.