**Algorithms lab manual**

**CSE 306**



**Department offering the LAB course**

Computer Science and Engineering

**Course Pre-requisites**

CSE 207

**Course Description**

|  |
| --- |
| This is a laboratory course based on its theory counterpart which is an introductory undergraduate course on the design and analysis of algorithms. |

**LAB Course Objectives**

1. To **emphasize** on efficient algorithm designing, solving practical problems through algorithmic techniques and data structures to be used in the implementations of algorithms.
2. To **expose** the students to a variety of techniques that have practical applications, while conducting detailed analysis of the requirements required by the algorithms.

**Intended learning outcomes of the course (ILOs)**

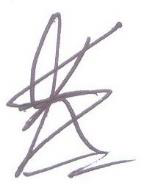
|  |  |
| --- | --- |
| SKILLS | **Solving practical problems using Data Structures in various applications** |
| 1. Apply programming skills and data structures to implement algorithms. |
| 1. Develop and implement algorithmic solutions to real-life problems. |

**Mapping of Course LO and PLO:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Learning Outcome**  **(LO) of the Course** | **Program Learning Outcome (PLO)** | | | | | | | | | | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| **ILO 1** | MJ |  | MJ |  |  |  |  |  | MJ | MN |  |  |
| **ILO 2** | MJ |  |  | MN |  |  |  |  | MJ | MJ |  |  |

**Lab Instructions:**

* Strictly observe the instructions given by the Faculty / Lab. Instructor.
* NO FOOD, DRINK, IN ANY FORM is allowed in the lab.
* TURN OFF CELL PHONES! If you need to use it, please keep it in bags.
* Avoid all horseplay in the laboratory. Do not misbehave in the computer laboratory. Work quietly.
* Save often and keep your files organized.
* Don’t change settings and surf safely.
* Do not reboot, turn off, or move any workstation or PC.
* Do not load any software on any lab computer (without prior permission of Faculty and Technical Support Personnel). Only Lab Operators and Technical Support Personnel are authorized to carry out these tasks.
* Do not reconfigure the cabling/equipment without prior permission.
* Do not play games on systems.
* Violation of the above rules and etiquette guidelines will result in disciplinary action.



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature of the faculty

**INSERTION SORT**

**Algorithm:**

**Step 1** − If it is the first element, it is already sorted. return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the

value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

**Pseudocode:**

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure

**MERGE SORT**

**Algorithm:**

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

**Pseudocode:**

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

end while

return c

end procedure

**QUICK SORT**

**Algorithm:**

**Pivot Algorithm**

**Step 1** − Choose the highest index value has pivot

**Step 2** − Take two variables to point left and right of the list excluding pivot

**Step 3** − left points to the low index

**Step 4** − right points to the high

**Step 5** − while value at left is less than pivot move right

**Step 6** − while value at right is greater than pivot move left

**Step 7** − if both step 5 and step 6 does not match swap left and right

**Step 8** − if left ≥ right, the point where they met is new pivot

**Quicksort Algorithm**

**Step 1** − Make the right-most index value pivot

**Step 2** − partition the array using pivot value

**Step 3** − quicksort left partition recursively

**Step 4** − quicksort right partition recursively

**Pseudocode:**

**Pivot Pseudocode**

function partitionFunc(left, right, pivot)

leftPointer = left

rightPointer = right - 1

while True do

while A[++leftPointer] < pivot do

//do-nothing

end while

while rightPointer > 0 && A[--rightPointer] > pivot do

//do-nothing

end while

if leftPointer >= rightPointer

break

else

swap leftPointer,rightPointer

end if

end while

swap leftPointer,right

return leftPointer

end function

**Quicksort Pseudocode**

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

end procedure

**Activity Selection Problem**

**Input:** A list of activity, and the number of elements in the list.

**Output −**The order of activities how they have been chosen.

**Algorithm:**

Begin

   initially sort the given activity List

   set i := 1

   display the ith activity //in this case it is the first activity

   for j := 1 to n-1 do

      if start time of act[j] >= end of act[i] then

         display the jth activity

         i := j

   done

End

**Fractional Knapsack Problem**

**Input:** set *S* of items with benefit *bi* and weight *wi*; max. weight *W*

**Output −**amount xi of each item i to maximize benefit w/ weight at most W

**Algorithm:**

for each item i in S←

xi ← 0

vi ← bi/ wi {value}

w ← 0 {total weight}

while w < W

remove item i with highest vi

xi ← min{wi , W - w}

w ← w + min{wi , W - w}

**Constructing a Huffman Code**

**Input:** a set C of n characters and each character has a frequency f(c)

**Output –**Huffman tree

**Algorithm:**

Alg.: HUFFMAN(C)

1. n ← ⎜C ⎜
2. Q ← C
3. **for** i ← 1 **to** n – 1
4. **do** allocate a new node z
5. left[z] ← x ← EXTRACT-MIN(Q)
6. right[z] ← y ← EXTRACT-MIN(Q)
7. f[z] ← f[x] + f[y]
8. INSERT (Q, z)
9. **return** EXTRACT-MIN(Q)

**0-1 Knapsack Problem**

**Input:** set *S* of items with benefit *bi* and weight *wi*; max. weight *W*

**Output −**find xi such that for all xi = {0, 1}, i = 1, 2, ..., n

∑ wixi ≤ W and

∑ xivi is maximum

**Algorithm:**

for w = 0 to W do

c[0, w] = 0

for i = 1 to n do

c[i, 0] = 0

for w = 1 to W do

if wi ≤ w then

if vi + c[i-1, w-wi] then

c[i, w] = vi + c[i-1, w-wi]

else c[i, w] = c[i-1, w]

else

c[i, w] = c[i-1, w]

## Longest Common Subsequence Problem

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

**Subsequence**

Let us consider a sequence S = <s1, s2, s3, s4, …,sn>.

A sequence Z = <z1, z2, z3, z4, …,zm> over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

**Common Subsequence**

Suppose, ***X*** and ***Y*** are two sequences over a finite set of elements. We can say that ***Z*** is a common subsequence of ***X*** and ***Y***, if ***Z*** is a subsequence of both ***X*** and ***Y***.

**Longest Common Subsequence**

If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.

**Algorithm:**

function LCS\_LENGTH(X,Y):

m := length(X)

n := length(Y)

for i = 1 to m do

C[i, 0] := 0

for j = 1 to n do

C[0, j] := 0

for i = 1 to m do

for j = 1 to n do

if xi = yj

C[i, j] := C[i - 1, j - 1] + 1

B[i, j] := ‘D’

else

if C[i -1, j] ≥ C[i, j -1]

C[i, j] := C[i - 1, j] + 1

B[i, j] := ‘U’

else

C[i, j] := C[i, j - 1]

B[i, j] := ‘L’

return C and B

function Print-LCS(B, X, i, j):

if i = 0 and j = 0

return

if B[i, j] = ‘D’

Print-LCS(B, X, i-1, j-1)

Print(xi)

else if B[i, j] = ‘U’

Print-LCS(B, X, i-1, j)

else

Print-LCS(B, X, i, j-1)

**Dijsktra’s Algorithm**

**Input:**

* A weighted, directed graph G = (V, E) where V is set of all vertices and E is set of all edges in the graph
* Edge weights cannot be negative

**Output −**find a shortest path from a given source vertex s to each vertex v ∈ V

**Algorithm:**



**Bellman Ford Algorithm**

**Input:**

* A weighted, directed graph G = (V, E) where V is set of all vertices and E is set of all edges in the graph
* Allow negative edge weights

**Output −**find a shortest path from a given source vertex s to each vertex v ∈ V. Detect negative weight cycles in the graph

**Algorithm:**

for each vertex v Є G.V

v.d := ∞

v.∏ := NIL

s.d := 0

for i = 1 to |G.V| - 1

for each edge (u, v) Є G.E

if v.d > u.d + w(u, v)

v.d := u.d +w(u, v)

v.∏ := u

for each edge (u, v) Є G.E

if v.d > u.d + w(u, v)

return FALSE

return TRUE