Unit – 2

1) Linear Search - Algorithm

```
# Input    : Array A, element x
# Output   : First index of element x in A or -1 if not found

Algorithm: Linear_Search
for i = 1 to last index of A
      if A[i] equals element x
           return i
return -1
```

2) Bubble Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Bubble_Sort(A)
for i ← 1 to n-1 do                θ(n)
   for j ← 1 to n-i do
     if  A[j] > A[j+1] then
         temp ← A[j]                      θ(n^2)
         A[j] ← A[j+1]
         A[j+1] ← temp
```

3) Selection Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do                              θ(n)
      minj ← i;
      minx ← A[i];
      for j ← i + 1 to n do
            if A[j] < minx then
                  minj ← j;                        θ(n²)
                  minx ← A[j];
      A[minj] ← A[i];
      A[i] ← minx;
```

4) Insertion Sort - Algorithm

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
      x ← T[i];                                    θ(n)
      j ← i - 1;
      while x < T[j] and j > 0 do
            T[j+1] ← T[j];                         θ(n²)
            j ← j - 1;
      T[j+1] ← x;
```

5) Heap Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Heap_Sort(A[1,…,n])
     BUILD-MAX-HEAP(A)
     for i ← length[A] downto 2
          do exchange A[1] ↔ A[i]
          heap-size[A] ← heap-size[A] – 1
          MAX-HEAPIFY(A, 1, n)
```

```
Algorithm: Max-heapify(A, i, n)
                                        1
l ← LEFT(i)          l ← 2
r ← RIGHT(i)         r ← 3
if l ≤ n and A[l] > A[i]          Yes
   then largest ← l        largest ← 2
   else largest ← i
if r ≤ n and A[r] > A[largest]    Yes
   then largest ← r        largest ← 3
if largest ≠ i       Yes
   then exchange A[i] ↔ A[largest]
MAX-HEAPIFY(A, largest, n)
```

6) Radix Sort

```
Algorithm: RADIX-SORT(A, d)
 for i ← 1 to d
      do use a stable sort to sort array A on digit i
```

7) Bucket Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Bucket-Sort(A[1,…,n])
     n ← length[A]
     for i ← 1 to n do
          insert A[i] into bucket B[⌊A[i] ÷ n⌋]
     for i ← 0 to n – 1 do
          sort bucket B[i] with insertion sort
     concatenate the buckets B[0], B[1], . . ., B[n - 1] together in
     order.
```

8) Counting Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Counting-Sort(A[1,…,n], B[1,…,n], k)
for i ← 1 to k do
    c[i] ← 0
for j ← 1 to n do
    c[A[j]] ← c[A[j]] + 1
for i ← 2 to k do
    c[i] ← c[i] + c[i-1]
for j ← n downto 1 do
    B[c[A[j]]] ← A[j]
    c[A[j]] ← c[A[j]] - 1
```

Unit – 3

1)Binary Search – Iterative Algorithm

```
Algorithm: Function biniter(T[1,…,n], x)
    if x > T[n] then return n+1
    i ← 1;
    j ← n;
    while i < j do
        k ← (i + j ) ÷ 2
        if  x ≤ T [k]  then  j ← k
        else i ← k + 1
    return i
```

2)Binary Search – Recursive Algorithm

```
Algorithm: Function binsearch(T[1,…,n], x)
    if n = 0 or x > T[n] then return n + 1
        else return binrec(T[1,…,n], x)
    Function binrec(T[i,…,j], x)
        if i = j then return i
        k ← (i + j) ÷ 2
        if x ≤ T[k] then
            return binrec(T[i,…,k],x)
        else return binrec(T[k + 1,…,j], x)
```

3) Merge Sort – Algorithm

```
Procedure: mergesort(T[1,…,n])
if n is sufficiently small then
insert(T)
else
array U[1,…,1+n/2],V[1,…,1+n/2]
      U[1,…,n/2] ← T[1,…,n/2]
      V[1,…,n/2] ← T[n/2+1,…,n]
            mergesort(U[1,…,n/2])
            mergesort(V[1,…,n/2])
            merge(U, V, T)
```

```
Procedure:
merge(U[1,…,m+1],V[1,…,n+1],T[1,…,m+n])
i ← 1;
j ← 1;
U[m+1], V[n+1] ← ∞;
for k ← 1 to m + n do
      if U[i] < V[j]
            then T[k] ← U[i];
                  i ← i + 1;
      else T[k] ← V[j];
            j ← j + 1;
```

4) Quick Sort - Algorithm

```
Procedure: quicksort(T[i,…,j])
{Sorts subarray T[i,…,j] into
ascending order}
if j – i is sufficiently small
then insert (T[i,…,j])
else
      pivot(T[i,…,j],l)
      quicksort(T[i,…,l - 1])
      quicksort(T[l+1,…,j]
```

```
Procedure: pivot(T[i,…,j]; var l)
p ← T[i]
k ← i
l ← j + 1
repeat k ← k+1 until T[k] > p or k ≥ j
repeat l ← l-1 until T[l] ≤ p
while k < l do
      Swap T[k] and T[l]
      Repeat k ← k+1 until T[k] > p
      Repeat l ← l-1 until T[l] ≤ p
Swap T[i] and T[l]
```

Unit-4

1) Make Change Problem

To generate table $c[i][j]$ use following steps:

Step-1: Make $c[i][0] = 0 \; for \; 0 < i \leq n$

Repeat step-2 to step-4 for the remaining matrix values

Step-2: If $i = 1$ then $c[i][j] = 1 + c[1][j - d_1]$

Step-3: If $j < d_i$ then $c[i][j] = c[i-1][j]$

Step-4: Otherwise $c[i][j] = \min(c[i-1][j], 1 + c[i][j - d_i])$

2) 0/1 Knapsack Problem

To generate table $V[i][j]$ use following steps:

Step-1: Make $V[i][0] = 0$ for $0 < i \leq n$

Step-2: if $j < w_i$ then

$$V[i][j] = V[i-1][j]$$

Step-3: if $j \geq w_i$ then

$$V[i][j] = \max(V[i-1][j], V[i-1][j - w_i] + v_i)$$

3) Floyd's Algorithm

```
function Floyd(L[1..n, 1..n]):array [1..n, 1..n]
    array D[1..n, 1..n]
    D ← L
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i,j] ← min(D[i,j], D[i,k]+ D[k,j])
    return D
```

4)Matrix Chain Multiplication using Dynamic Programming

To generate $M[i][j]$ use following steps:

Step-1: if $i = j$ then $M[i][j] = 0$

Step-2: if $j = i + 1$ then $M[i][j] = P_{i-1} \cdot P_i \cdot P_{i+1}$

Step-3: if $i < j$ then

$$\boxed{M[i][j] = min(M[i][k] + M[k+1][j] + P_{i-1} \cdot P_k \cdot P_j)\ with\ i \le k < j}$$

5) longest common subsequence

To generate table $c[i][j]$ use following steps:

Step-1: Make $c[i][0] = 0$ and $c[0][j] = 0$

Step-2: if $x_i = y_j$ then $c[i, j] \leftarrow c[i-1, j-1] + 1$

Step-3: else $c[i, j] \leftarrow \max(c[i-1, j], c[i, j-1])$

Unit-5

1) Make Change - Algorithm

```
# Input: C = {10, 5, 2, 1, 0.5} //C is a candidate set
# Output: S: set of selected coins
Function make-change(n): set of coins
S ← Ø {S is a set that will hold the solution}
sum ← 0 {sum of the items in solution set S}
while sum ≠ n do
      x ← the largest item in C such that sum + x ≤ n
      if there is no such item then
            return "no solution found"
      S ← S U {a coin of value x}
      sum ← sum + x
return S
```

2) Kruskal's Algorithm for MST

```
Function Kruskal(G = (N, A))
Sort A by increasing length
n ← the number of nodes in N
T ← Ø {edges of the minimum spanning tree}
Define n sets, containing a different element of set N
repeat
   e ← {u, v} //e is the shortest edge not yet considered
   ucomp ← find(u)
   vcomp ← find(v)
   if ucomp ≠ vcomp then merge(ucomp, vcomp)
   T ← T U {e}
until T contains n - 1 edges
return T
```

find(u) tells in which connected component a node u is found

merge(ucomp, vcomp) is used to merge two connected components.

3) Prim's Algorithm

```
Function Prim(G = (N, A): graph; length: A — R+): set of edges
T ← ∅
B ← {an arbitrary member of N}
while B ≠ N do
     find e = {u, v} of minimum length such that
          u ∈ B and v ∈ N \ B
     T ← T U {e}
     B ← B U {v}
return T
```

4) Dijkstra's Algorithm

```
Function Dijkstra(L[1 .. n, 1 .. n]): array [2..n]
array D[2.. n]
C ← {2,3,…, n}
{S = N \ C exists only implicitly}
for i ← 2 to n do
  D[i] ← L[1, i]
repeat n - 2 times
  v ← some element of C minimizing D[v]
  C ← C \ {v} {and implicitly S ← S U {v}}
  for each w ∈ C do
     D[w] ← min(D[w], D[v] + L[v, w])
return D
```

5) Fractional Knapsack Problem - Algorithm

```
Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n do
      x[i] ← 0 ; weight ← 0
While weight < W do
      i ← the best remaining object
      if weight + w[i] ≤ W then
            x[i] ←1
            weight ← weight + w[i]
      else
            x[i] ← (W - weight) / w[i]
            weight ← W
return x
```

W = 100 and Current weight in knapsack= 60
Object weight = 50
The fraction of object to be included will be
(100 – 60) / 50 = 0.8

6) Activity Selection - Algorithm

```
Algorithm: Activity Selection
Step I: Sort the input activities by increasing finishing time.  f_1
≤  f_2 ≤  . . . ≤  f_n

Step II: Call GREEDY-ACTIVITY-SELECTOR (s, f)
  n = length [s]
  A = {i}
  j = 1
  for  i = 2  to  n
        do if   s_i ≥ f_j
             then  A = A U {i}
                      j = i
  return  set A
```

7) Job Scheduling with Deadlines - Algorithm

```
Algorithm: Job-Scheduling (P[1..n], D[1..n])
1. Sort all the n jobs in decreasing order of their profit.
2. Let total position P = min(n, max(d_i))
3. Each position 0, 1, 2…, P is in different set and T({i}) = i, for 0 ≤ i ≤
   P.
4. Find the set that contains d, let this set be K. if T(K) = 0 reject the
   job; otherwise:
   1. Assign the new job to position T(K).
   2. Find the set that contains T(K) – 1. Call this set L.
   3. Merge K and L. the value for this new set is the old value of T(L).
```

8) Huffman Codes - Algorithm

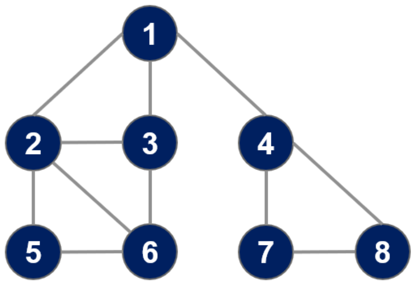
```
Algorithm: HUFFMAN (C)
n = |C|
Q = C
for i = 1 to n-1
        allocate a new node z
        z.left = x = EXTRACT-MIN(Q)
        z.right = y = EXTRACT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q,z)
return EXTRACT-MIN(Q)  // return the root of the tree
```

Unit-6

1) Depth-First Search Algorithm

```
procedure dfsearch(G)
   for each v ∈ N do
           mark[v] ← not-visited
   for each v ∈ N do
           if mark[v] ≠ visited
           then dfs(v)
procedure dfs(v)
   {Node v has not previously been visited}
   mark[v] ← visited
   for each node w adjacent to v do
           if mark[w] ≠ visited
           then dfs(w)
```

```
1. dfs(1)                Initial call
2.   dfs(2)              recursive call
3.     dfs(3)            recursive call
4.       dfs(6)          recursive call
5.         dfs(5)        recursive   call;
   progress is blocked
6.   dfs(4)              a  neighbour  of
   node 1 that has not been visited
7.     dfs(7)            recursive call
8.       dfs(8)          recursive call
9. There are no more nodes to visit
```




```
procedure dfs(v)
   mark[v] ← visited
   for each node w adjacent to v do
       if mark[w] ≠ visited
       then dfs(w)
```

2) Breadth First Search - Algorithm

```
procedure search(G)
    for each v ∈ N do
        mark[v] ← not visited
    for each v ∈ N do
        if mark[v] ≠ visited
        then bfs(v)
```

```
procedure bfs(v)
    Q ← empty-queue
    mark[v] ← visited
    enqueue v into Q
    while Q is not empty do
        u ← first(Q)
        dequeue u from Q
        for each node w adjacent to u do
                if mark[w] ≠ visited
                then mark[w] ← visited
         enqueue w into Q
```

Unit-7

1) N – Queen Algorithm

procedure queens (k, col, diag45, diag135)

{sol[1..k] is k-promising,

col = {sol[i] | 1≤i≤k},

diag45 = {sol[i]–i+1 | 1≤i≤k}, and

diag135 = {sol[i]+i–1 | 1≤i≤k}}

if k = 8 **then** {an 8-promising vector is a solution}

write sol

else {explore (k+1)-promising extensions of sol }

for j ← 1 **to** 8 **do**

if j ∉ col **and** j – k ∉ diag45 **and** j + k ∉ diag135 ∉ sol[k+1] ← j

then sol[k+1]← j

{sol[1..k+1] is (k+1)-promising}

queens(k + 1, col U {j}, diag45 U {j - k}, diag135 U {j + k})

2) 0/1 Knapsack Problem – Algorithm

```
function backpack(i, r)
      {Calculates the value of the best load that can be constructed
      using items of type i to n and whose total weight does not
      exceed r}
          b ← 0
          {Try each allowed kind of item in turn}
          for k ← i to n do
               if w[k] ≤ r then
                    b ← max(b, v[k] + backpack (k, r – w[k]))
          return b
```

Unit-8

1) Naive String Matching - Algorithm

```
NAIVE-STRING MATCHER (T,P)
n = T.length
m = P.length
for s = 0 to n-m
      if   p[1..m] == T[s+1..s+m]
           print   “Pattern   occurs   with
                shift” s
```

2) Rabin-Karp-Matcher

RABIN-KARP-MATCHER(T, P, d, q)

$n \leftarrow length[T];$

$m \leftarrow length[P];$

$h \leftarrow d^{m-1} \bmod q;$

$p \leftarrow 0;$

$t_0 \leftarrow 0;$

for $i \leftarrow 1$ to m do

  $p \leftarrow (d_p + P[i]) \bmod q$

  $t_0 \leftarrow (dt_0 + T[i]) \bmod q$

for $s \leftarrow 0$ to $n - m$ do

  if $p == t_s$ then

    if $P[1..m] == T[s+1..s+m]$ then

      print “pattern occurs with shift” s

  if $s < n\text{-}m$ then

    $t_{s+1} \leftarrow (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$

3) Compute Transition Function

```
COMPUTE-TRANSITION-FUNCTION(P, Σ )
m ← length[P]
for q ← 0 to m do
  for each character α Є Σ do
        k ← min(m + 1, q + 2)
        repeat k ← k – 1 until P_k ⊐ P_q α
        δ(q, α) ← k
return δ
```

4) Finite Automata Matcher

```
FINITE-AUTOMATON MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n do
    q ← δ(q, T[i])
    if q == m then
    print "Pattern occurs with shift" i – m
```

5) KMP- Compute Prefix Function

```
COMPUTE-PREFIX-FUNCTION(P)
      m ← length[P]
      π[1] ← 0
      k ← 0
      for q ← 2 to m
            while k > 0 and P[k + 1] ≠ P[q]
                    k ← π[k]
             end while
             if P[k + 1] == P[q] then
                    k ← k + 1
             end if
                    π[q] ← k
      return π
```

6) KMP-MATCHER

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                              //Number of characters matched.
for i ← 1 to n                     //Scan the text from left to right.
      while q > 0 and P[q + 1] ≠ T[i]
                q ← π[q]    //Next character does not match.
      if P[q + 1] == T[i] then
                then q ← q + 1      //Next character matches.
      if q == m then            //Is all of P matched?
                print "Pattern occurs with shift" i - m
                q ← π[q]    //Look for the next match.
```