Date _____

## Insertion Sort $O(n^2)$

```
Insertion(int arr[], int n)
{
    int i, j, key;
    for (i=1; i<n; i++)
    {
        key = arr[i];
        j = i-1;

        while (j>=0 && arr[j] > key)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = key;
    }
}
```

Date _____

## Merge Sort  O(nlogn)

```
MergeSort(A)
{  n = len(A)
   if n <= 1;
     return A
   L = MergeSort(A[0: n/2 - 1])
   R = MergeSort(A[n/2 : n-1])
   return MERGE(L, R)
}

MERGE(L, R)
 { result = length n array
   L(n/2) = ∞ ; R(n/2) = ∞
   i = 0 ; j = 0
   for K in (0.... n-1)
          if L[i] < R[j]
             result[K] = L[i]
             i = i+1;
          else
             result[K] = R[j]
             j = j+1;
   return result;
 }
```

Date _____

# QUICK SORT

$W = O(n^2)$

$A = O(n \log n)$

```
QUICKSORT(A,p,r)
{ if {p<r:
      q = PARTITION(A,p,r)
      QUICKSORT(A,.p, q-1)
      QUICKSORT(A,q+1,r)
}
```

```
PARTITION(A,p,r)
    {
     x = A[r]
     i = p-1
     for j=p to r-1
          if A[j] ≤ x
           i = i+1
           exchange A[i] with A[j]

     exchange A[i+1] with A[r]
     return i+1
    }
```

Date

# BINARY SEARCH

```
BinarySearch(int arr[], int l, int r, int x)
{
  if (r >= l) {
    int mid = l + (r-1)/2;
    if (arr[mid] == x)
      return mid;
    if (arr[mid] > x)
      return BinarySearch(arr, l, mid-1, x);

    return BinarySearch(arr, mid+1, r, x);
    }

  return -1;
}
```

Date _____

# HEAP SORT

```
MaxHeapify(A, i, n)
{
    l ← LEFT(i)
    r ← RIGHT(i)
    if (l ≤ n && A[l] > A[i])
        largest ← l

    else largest ← i

    if (r ≤ n && A[r] > A[largest])
        largest ← r
    if (largest != i)
        exchange A[i] ⟷ A[largest]
        MAX-Heapify(A, largest, n)
}
```

```
BuildMaxHeap(A)
{   n = length[A]
    for (i ← [n/2] to 1)
        do MaxHeapify(A, i, n)
}
```

Date

# Count Sort

```
CountSort(A, B, K)
{
  let C[1...K] be a new array
  for (i=1 to K)
      C[i] = 0
  for (j=1 to A.length)
      C[A[j]] = C[A[j]] + 1

  for (i=2 to K)
      C[i] = C[i] + C[i-1]

  for (j = A.length to 1)
      B[C[A[j]]] = A[j]
      C[A[j]] = C[A[j]] - 1
}
```

Date _____

RADIX SORT (Similar to Count Sort)

Date

# BFS

```
BFS(s);
{ visit(s);
  queue.insert(s);
  while (queue is not empty)
     {
        u = queue.extractHead();
        for (each edge <u, d>)
           {
              if(d has not been visited)
              { visit(d);
                queue.insert(d);
           }
        }
     }
}
```

Date

# DFS

```
DFS(s)
  {
    S.underDFS = true;
    for (each edge <s, d>)
      {
        if (!d.underDFS and d have not been visited)
          {
            DFS(d);
          }
      }
    visit(s);
  }
```

Date _____

# PRIM'S ALGORITHM

```
PRIM (Graph(V,E), S)
{
    MST = {}
    visited = {s}
    for all v besides s: d[v] = ∞ and K[v] = NULL
    for each neighbor v of s: d[v] = w(s,v) and K[v] = s
    while len(visited) < n:
        x = unvisited vertex v with smallest d[v] value
        MST.add((K[x], x))
        for each unreached neighbor v of x:
            d[v] = min(w(x,v), d[v])
            if d[v] was updated : K[v] = x
        visited.add(x)

    return MST;

}
```

Date _____

# KRUSKAL'S ALGORITHM

```
Kruskal (Graph(V,E))
{
  E-Sorted = E sorted by weight in non-decreasing order
  MST = {}
  for v in V:
      Make-SET(v)
  for (u,v) in E-SORTED:
      if FIND(u) != FIND(v)
          MST.add ((u,v))
          UNION (u,v)

  return MST

}
```

Date

# BELLMAN-FORD ALGORITHIM

```
BellmanFord (G, s)
{
    d^(k) = [] for k=0, ...., n-1
    d^(0)[v] = ∞ for all v in V(except s)
    d^(0)[s] = 0
    for k = 1, ...., n-1;
        for b in V:
            d^(k)[b] ← min { d^(k-1)[b], min_a { d^(k-1)[a] + w(a,b) } }

    reborn d^(n-1)

}
```

Date_____

# DIJKSTRA'S ALGORITHM

```
Dijkstra(G, s)
{
    dist[s] ← 0
    for all v ε V-{s}
        do  dist[v] ← ∞

    S ← ∅
    Q ← V
    while Q ≠ ∅
     do  u ← mindistance(Q.dist)
         S ← S ε {u}
         for all v ε neighbor[u] do
                dist[v] = min(dist[v], dist[u] + ω(u,v))
    return dist;
}
```

Date_____

# FLOYD - WARSHALL ALGORITHM

```
Floyd ( W[1...n, 1..n])

{
  D ← W
  for k ← 1 to n do
      for i ← 1 to n d
          for j ← 1 to n do
              D[i,j] ← min{D[i,j], D[i,k] + D[k,j]}

  return D;

}
```

- **APPROX-VERTEX-COVER(G)**

```
C = ∅;
E'=G.E;
while(E' ≠ ∅ ){
    Randomly choose a edge (u,v) in E', put u and v into C;
    Remove all the edges that covered by u or v from E'
}
Return C;
```

# Traveling-salesman problem

**APPROX-TSP-TOUR**(G)

Minimum Spanning Tree;

Creating a Cycle;

Removing Redundant Visits;

# The set-covering problem

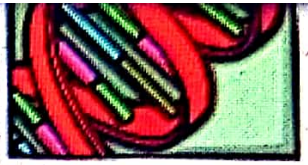
```
GREEDY-SET-COVER(X, F)
    U=X;
    C=Ø;
    While(U≠ Ø){
        Select S∈F that maximizes |S∩U|;
        U=U-S;
        C=CU{S};
    }
    return C;
```

```
GRAHAM-SCAN(Q)
 1  let p_0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
 2  let ⟨p_1, p_2, ..., p_m⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p_0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p_0)
 3  let S be an empty stack
 4  PUSH(p_0, S)
 5  PUSH(p_1, S)
 6  PUSH(p_2, S)
 7  for i = 3 to m
 8      while the angle formed by points NEXT-TO-TOP(S), TOP(S),
            and p_i makes a nonleft turn
 9          POP(S)
10      PUSH(p_i, S)
11  return S
```

## Closest Pair Algorithm

```
Closest-Pair(p1, …, pn)
{
    Compute separation line L such that half the points
    are on one side and half on the other side.                 O(N log N)

    δ1 = Closest-Pair(left half)
    δ2 = Closest-Pair(right half)                               2T(N / 2)
    δ  = min(δ1, δ2)

    Delete all points further than δ from separation line L     O(N)

    Sort remaining points by y-coordinate.                      O(N log N)

    Scan points in y-order and compare distance between
    each point and next 11 neighbors. If any of these          O(N)
    distances is less than δ, update δ.

    return δ.
}
```

# Brute-Force Algorithm



http://whocouldthat.be/visualizing-string-matching/

```
function brute_force(text[], pattern[])
{
  // let n be the size of the text and m the size of the
  // pattern

  for(i = 0; i < n; i++) {
    for(j = 0; j < m && i + j < n; j++)
      if(text[i + j] != pattern[j]) break;
      // mismatch found, break the inner loop
    if(j == m) // match found
  }
}
```

```
Algorithm KMPMatch(T, P)
    F ← failureFunction(P)
    i ← 0
    j ← 0
    while i < n
        if T[i] = P[j]
            if j = m − 1
                return i − j { match }
            else
                i ← i + 1
                j ← j + 1
        else
            if j > 0
                j ← F[j − 1]
            else
                i ← i + 1
    return −1 { no match }
```



# The Boyer-Moore

```
Algorithm BoyerMooreMatch(T, P, Σ)
    L ← lastOccurenceFunction(P, Σ)
    i ← m − 1
    j ← m − 1
    repeat
        if T[i] = P[j]
            if j = 0
                return i { match at i }
            else
                i ← i − 1
                j ← j − 1
        else
            { character-jump }
            l ← L[T[i]]
            i ← i + m − min(j, 1 + l)
            j ← m − 1
    until i > n − 1
    return −1 { no match }
```