Interval Scheduling:
Input: activieies w a start time s_i and a finish time f_i
Output: do as many activities as possible.
Optimal solution: earliest finish time.
Graph Algorithms:
BFS-layer by layer
DFS-go as far as you can, then backtrack
BFS(G,s) -> output is contains all vertices in layers
Also: L<-{s}
I<=1
Repeat
    Li <- all neighbors of vertices in Li-1 that are unmarked, and mark vertices in L_i
Until Li-1 = null
Runtime O(V+E)
DFS(G,s)
Mark s
For each neighbor v of s
    If unmarked, then DFS(G,v)
Runtime: O(V+E)
Dijkstra: Single source shortest path
Input: G=(V,E)
Output: For every vertex v in V, shortest path from s to v
Weighted with non negative weights
Algo:
Maintain S subset of V
S = {s}
D(s) = 0
Pick a v in V-S that minimizes d(u) + w(u,v)
s <- S U {v}
d(v) <- d(u) + w(u,v)
Priority Queue, maintain a set Q of elements where each element has a value key.
Insert(Q,x) insert element x with value key(x)
Minimum (Q) returns a pointer to Q with min key
Extract-Min(Q) returns an element with min key and extract it from Q
Decrease-key(Q,x,k) given a pointer to element x in S, change key(x) to k if key(x) > k
Algo:
For each v in V do d[v] <- infinity
D[s] = 0
Q – a binary heap with all v
While Q!= 0
    u <- Extract-Min(Q)
    For each neighbor v of u do
        If v in Q then decrease-key (Q,v,d[u]+q(u,v)
Runtime: O((m+n)lgn)

MST:
Input: A connected undirected graph G=(V,E) with weights w on edges
Output: A tree T that spans all vertices and minimized sum of weights
Prims Algo G=(V,E)
Q <- V
For all v in V, set key(v) to infinity
    s in V has key set to 0
While Q != 0
    u = Extract-Min(Q)
For each neighbor v of u
    If v in Q and w(u,v) < key(v)
        Key(v) = w(u,v)
        Pi(v) = u
Runtime: O((m+n)lgn)
Cut Lemma: Let C e a subset in V. Let (u,v) be a lightest edge connecting C to V-C. There exists a minimum spanning tree that contains (u,v).
Cycle Lemma: Let C be a cycle in G. Let (u,v) be heaviest max edge on C. Then there is an MST that does not contain (u,v).
If all edge weights are distinct, then MST is unique.
The actual edge weights don't matter, only the order between them.
Kruskal:
Sort edges according to their weights, and go over edges in order from lightest to heaviest.
If an edge e creates a cycle with the edges we already picked, discard it.
Otherwise, pick e to spanning tree.
Union Find: maintains a collection of disjoint sets.
Each set is identified by a representative element in the set.
Make-Set(x) create a set with x as rep.
Union(x,y) union sets represented by x and y
Find-Set(x) finds rep of set containing x.
G(V,E) kruskal
Sort edges e_1 < … e_m
T = null
For each v, make set (v)
For I = 1..m do
    If find-set(u) != find-set(v) where e_i = (u,v)
        Then T = T U {e_i}
            Union(Find-set(u, Find-set(v))
Takes O(mlgn) time
Clustering has a distance function added on.

1. Construct complete graph
2. Invoke kruskal until k disjoint sets are

Divide and Conquer:
Paradigm:
-break up problem into several problems on smaller input size
-solve each part recursively
-combine the solutions to a solution for the overall problem.
Example: mergesort
Recurrences:
Master theorem:
$T(n) = aT(n/b) + O(n^k)$
$a>1$, $b>1$, k is a non negative constant
$k < \log_b(a)$ then $T(n) = O(n ^ (\log_b(a))$
$k = \log_b(a)$ then $T(n) = O(n^(\log_b(a))\lg n)$
$k > \log_b(a)$ then $T(n) = O(n^k)$
Closest Pair of Points
Input: n points in the plane
Output: a pair of points that are closest in Euclidean distance.

1. Divide and find a vertical line that has half of the points on either side
2. Conquer and find the closest pair of points in each side.
3. Find the closest pair with one point on each side, another point on the other. Return the best pair among the two point.

Runtime: O(nlgn)