Some Classic Algorithm

1.

Description:

Find the jth smallest number in A (unsorted array).

Algorithm(for median):

1 Partition array A into ceiling(n/5) lists of 5 items each.
  L1 = {A[1]; A[2]; : : : ; A[5]}, L2 = {A[6]; : : : ; A[10]}, : : :,
  Li = {A[5i − 4]; : : : ; A[5i]}, : : :,
  L(ceiling(n/6))=5e = {A[5 ∗ ceiling(n/5) − 4]; : : : ; A[n]}.

2 For each find median bi of Li using brute-force in O(1) time. Total O(n) time

3 Let B = {b1; b2; : : : ; b_ceiling(n/5)}

4 Find median b of B

Code:

```
select(A, j):
Form lists L1; L2; : : : ; L_ceiling(n/5)
Find median bi of each Li using brute-force
Find median b of B = {b1; b2; : : : ; b_ceiling(n/5)}
Partition A into Aless and Agreater using b as pivot
if (|Aless|) = j return b
else if (|Aless|) > j)
        return select(Aless, j)
else
        return select(Agreater, j − |Aless|)
```

Running Time:

T(n) = T(n/5) + max{T(|Aless|), T(|Agreater|)} + O(n)
From Lemma,
T(n) <= T(ceiling(n/5)) + T(floor(7n/10 + 6)) + O(n)
and
T(n) = O(1) n < 10
conclude
T(n) = O(n)

2.

Description:

Find the longest increasing subsequence inside a given sequence A[].

Algorithm:

Use L[i] to record the longest increasing subsequence end in A[i]. And D[i] to record the previous element in this subsequence.

Outer loop through A[], and inner loop check every element before the current element to find the longest increasing loop end in current loop, fill A[i] and D[i].

Code:

```
LIS(A[1::n]):
Array L[1::n] (∗ L[i] stores the value LISEnding(A[1..i]) ∗)
Array D[1::n] (∗ D[i] stores how L[i] was computed ∗)
m = 0, h = 0 (∗ m is the length of the LIS∗)
for i = 1 to n do
```

```
                L[i] = 1
                D[i] = 0
                for j = 1 to i − 1 do
                        if (A[j] < A[i]) and (L[i] < 1 + L[j]) do
                                L[i] = 1 + L[j], D[i] = j
                        if (m < L[i]) m = L[i], h = i
        S = empty sequence
        while (h > 0) do
                add L[h] to front of S
                h = D[h]
        Output optimum value m, and an optimum subsequence S
```
Running time: O(n^2) Space: O(n).

3.

Description:

Check if a string w is in L^k, where L is a language and k is an non-negative integer.

Algorithm:

Loop through each element to see if the substring before it (inclusive) is in L, and the substring after it is in L^(k−1).

Code:
```
        IsStringinLk(A[1::n]; k):
        If (k = 0)
                If (n = 0) Output YES
                Else Ouput NO
        If (k = 1)
                Output IsStringinL(A[1::n])
        Else
                For (i = 1 to n ? 1) do
                        If (IsStringinL(A[1::i]) and IsStringinLk(A[i
+ 1::n]; k ? 1))
                                Output YES
        Output NO
```
Running time: O(n^2 ∗ k)

4.

Description:

Given two words X (x1, ..., xm)and Y (y1, ..., yn), and gap penalty S and mismatch costs Vpq. Find alignment of minimum cost.

Algorithm:

M (i; j) records the cost if before xi and yj is matched (inclusive).
```
                M(i; j) = min{
                                        V(xi, yj) + M[i−1][j−1]
(∗math this two∗)
                                        S + M[i−1][j] (∗match xi
with gap∗)
                                        S + M[i][j−1] (∗match yj
with gap∗)
                                        }
                Using two arrays, N[i, 0] records M[i−1], N[i, 1]
```

records M[i]. Fill the array.

Code:

```
for all i do N[i; 0] = i ∗ S
for j = 1 to n do
        N[0; 1] = j ∗ S (∗ corresponds to M(0; j) ∗)
for i = 1 to m do
        N[i; 1] = min{
                                                V(xi, yj) +
N[i−1; 0]
                                                S + N[i−1;
1]
                                                S + N[i; 0]
        }
for i = 1 to m do
        Copy N[i; 0] = N[i; 1]
```

Running Time:

O(mn), Space O(m).

5.

Description:

Find maximum weight independent set in T (T is a tree).

Algorithm:

See code.

Code:

```
MIS−Tree(T):
Let v1; v2; : : : ; vn be a post−order traversal of nodes of T
for i = 1 to n do
        M[vi] = max{
                                sum of M [vj], vj is the
child of vi
                                w(vi) + sum of M [vj], vj is
the grandchild of vi
        }
return M[vn] (∗ Note: vn is the root of T ∗)
```

Running Time:

O(n)

6.

Description:

A set of jobs with start and finish times to be scheduled on a resource. Schedule as many jobs as possible.

Algorithm:

Schedule the earliest finish time job.

Code:

```
R is the set of all requests
X is empty; (∗ X stores the jobs that will be scheduled ∗)
while R is not empty
        choose i in R such that finishing time of i is
smallest
        if i does not overlap with requests in X
                add i to X
        remove i from R
```

        return X
Running Time:
        O(n log n)
7.
        Some basic graph problems.
        Graph Search and Find Tree O(m+n):
```
        Explore(G,u):
        array Visited[1::n]
        Initialize: Set Visited[i] = FALSE for 1 <= i <= n
        List: ToExplore, S
        Add u to ToExplore and to S, Visited[u] = TRUE
        Make tree T with root as u
        while (ToExplore is non-empty) do
                Remove node x from ToExplore
                for each edge (x; y) in Adj(x) do
                        if (Visited[y] == FALSE)
                                Visited[y] = TRUE
                                Add y to ToExplore
                                Add y to S
                                Add y to T with edge (x; y)
        Output S
```
        BFS: use queue to implement ToExplore
        DFS: use stack to implement ToExplore
        Follow Ups:
        Q: Find all nodes that can reach u.
        A: Form Grev that reverse every edge of G. Do basic search on Grev.
        Q: Find all strongly connected components containing u. (SCC(u)).
        A: SCC(G; u) = rch(G; u) union rch(Grev; u)
        Q: Is G strongly connected?
        A: For any v check if |SCC(u)| = |V|.
8.
Description:
        Given G, is it a DAG? If it is, generate a topological sort. Else output a cycle C.
Algorithm:
        Record the time we visit each node when running DFS on G.
        1 Tree edges that belong to T
        2 A forward edge is a non-tree edges (x; y) such that pre(x) < pre(y) < post(y) < post(x).
        3 A backward edge is a non-tree edge (y; x) such that pre(x) < pre(y) < post(y) < post(x).
        4 A cross edge is a non-tree edges (x; y) such that the intervals [pre(x); post(x)] and [pre(y); post(y)] are disjoint.
        G has a cycle iff there is a back-edge in DFS(G).
        If there is a back edge e = (v; u) then G is not a DAG. Output cyclce C formed by path from u to v in T plus edge (v; u).
        Otherwise output nodes in decreasing post-visit order. Note: no need to sort, DFS(G) can output nodes in this order.

Code (for DFS):

```
        DFS(u)
                Mark u as visited
                pre(u) = ++time
                for each edge (u; v) in Out(u) do
                        if v is not visited
                                add edge (u; v) to T
                                DFS(v)
                post(u) = ++time
```

Running Time:
        O(m+n)

9.

:
        Find all SCC in a graph G and obtain meta-graph.

Algorithm:
        Form a topological sort then we can gurantee the source in always at front. DFS(u) only visited edges and vertices in SCC(u).

Code:

```
        do DFS(Grev) and output vertices in decreasing post order.
        Mark all nodes as unvisited
        for each u in the computed order do
                if u is not visited then
                        DFS(u)
                        Let Su be the nodes reached by u
                        Output Su as a strong connected component
                        Remove Su from G
```

Running Time:
        O(m + n)

10.

BFS with layers:

```
        BFSLayers(s):
                Mark all vertices as unvisited and initialize T to be
empty
                Mark s as visited and set L0 = {s}
                i = 0
                while Li is not empty do
                        initialize Li+1 to be an empty list
                        for each u in Li do
                                for each edge (u; v) in Adj(u) do
                                        if v is not visited
                                                mark v as visited
                                                add (u; v) to tree
T

                                                add v to Li+1
                        i = i + 1
```

Running time: O(n + m)

Properties:
        BFSLayers(s) outputs a BFS tree
        Li is the set of vertices at distance exactly i from s
        Every edge in the graph is either between two vertices that

are either (i) in the same layer (cross edge), or (ii) in two consecutive layers.

11.

Decription:

Find shortest paths for all pairs of nodes (all edges are non negative) for a given node u.

Algorithm:

Dijkstra's Algorithm: Loop through every node, put the computed node in X. Start from u, find the closest node v to X from V − X. Update the distance between s and every node in Adj(v).

Code:

```
        Initialize for each node v, dist(s; v) = infinity
        Initialize X = empty;, dist(s; s) = 0
        for i = 1 to |V| do
                Let v be such that dist(s; v) = min(u in V − X)
dist(s; u)
                X = X + {v}
                for each u in Adj(v) do
                        dist(s; u) = min{dist(s; u); dist(s; v) +
l(v; u)}
```

Code using Priority Queues:

```
        Q = makePQ()
        insert(Q, (s; 0))
        for each node u != s do
                insert(Q, (u, infinity))
        X = empty;
        for i = 1 to {V} do
                (v; dist(s; v)) = extractMin(Q)
                X = X + {v}
                for each u in Adj(v) do
                        decreaseKey{Q; (u, min(dist(s; u); dist(s; v)
+ l(v; u)))}
```

Code to find the shortest path:

```
        Q = makePQ()
        insert(Q, (s; 0))
        prev(s) = null
        for each node u != s do
                insert(Q, (u;infinity) )
                prev(u) = null
        X = empty;
        for i = 1 to |V| do
                (v; dist(s; v)) = extractMin(Q)
                X = X + {v}
        for each u in Adj(v) do
                if (dist(s; v) + l(v; u) < dist(s; u)) then
                        decreaseKey(Q, (u; dist(s; v) + l(v; u)))
                        prev(u) = v
```

The edge set (u; prev(u)) is the reverse of a shortest path tree rooted at s. For each u, the reverse of the path from u to s in the tree is a shortest path from s to u.

Running Time:
O((n + m) log n) with heaps
O(n log n + m) with Fibonacci Heaps

12.

Description:
Check if there is negative length cycle reachable from s, if not find the shortest path distances.

Algorithm:
Bellman-Ford Algorithm:
d(v; k): shortest walk length from s to v using at most k edges.
Note: dist(s; v) = d(v; n - 1). Recursion for d(v; k):

```
d(v; k) = min{
                                    min(u in V) (d(u; k - 1) + l(u; v))
                                    d(v; k - 1)
}
```

Base case: d(s; 0) = 0 and d(v; 0) = infinity for all v != s.

Code:

```
for each u in V do
        d(u) = infinity
d(s) = 0
for k = 1 to n - 1 do
        for each v in V do
                for each edge (u; v) in In(v) do
                        d(v) = min{d(v); d(u) + l(u; v)}
(* One more iteration to check if distances change *)
for each v in V do
        for each edge (u; v) in In(v) do
                if (d(v) > d(u) + l(u; v))
                        Output "Negative Cycle"
for each v in V do
        dist(s; v) = d(v)
```

Running Time:
O(mn) Space: O(m + n)

13.

Description:
Find shortest distance for everynode from s in a DAG.

Algorithm:
As there are no cycles in DAG, we can run shortest path for it.

Code:

```
for i = 1 to n do
        d(s; vi) = infinity
d(s; s) = 0
for i = 1 to n - 1 do
        for each edge (vi; vj) in Adj(vi) do
                d(s; vj) = min{d(s; vj); d(s; vi) + l(vi; vj)}
return d(s; ) values computed
```

Running Time:

O(m + n)

14.

Decription:

Find shortest paths for all pairs of nodes.

Algorithm:

Floyd−Warshall Algorithm:

Code:

```
for i = 1 to n do
	for j = 1 to n do
		dist(i; j; 0) = l(i; j) (∗ l(i; j) = infinity if (i; j) not in E, 0 if i = j ∗)
for k = 1 to n do
	for i = 1 to n do
		for j = 1 to n do
			dist(i; j; k) = min{dist(i; j; k − 1); dist(i; k; k − 1) + dist(k; j; k − 1)}
for i = 1 to n do
	if (dist(i; i; n) < 0) then
		Output that there is a negative length cycle in G
```

Running Time:

O(n^3) Space: O(n3).

15.

Decription:

Find the MST in a graph.

Algorithm:

Kruskal: Process edges in the order of their costs (starting from the least) and add edges to T as long as they don't form a cycle.

Prim: T maintained by algorithm will be a tree. Start with a node in T. In each iteration, pick edge with least attachment cost to T.

Reverse Delete Algorithm: Process edges in the order of their costs (starting from the greatest) and remove edges from T as long as they don't form disconnect.

Boruvka: While T is not spanning, for every connected component, add the cheapest edge connect to it to T.

Code:

```
Prim:
Prim ComputeMST
	E is the set of all edges in G
	S = {1}
	T is empty (∗ T will store edges of a MST ∗)
	for v not in S, a(v) = min (w in S) c(w; v)
	for v not in S, e(v) = w such that w in S and c(w; v) is minimum
	while S != V do
		pick v with minimum a(v)
		add (e(v); v) to T
		add v to S
```

```
                        update arrays a and e
                add e to T
                add w to S
                return the set T
        Maintain vertices in V \ S in a priority queue with key a(v).
        Kruskal:
                Kruskal ComputeMST
                        Sort edges in E based on cost
                        T is empty (∗ T will store edges of a MST ∗)
                        each vertex u is placed in a set by itself
                        while E is not empty do
                                pick e = (u; v) in E of minimum cost
                                if u and v belong to different sets
                                        add e to T
                                        merge the sets containing u
and v
                        return the set T
        Boruvka:
        T is empty; (∗ T will store edges of a MST ∗)
        while T is not spanning do
                X is empty;
                for each connected component S of T do
                        add to X the cheapest edge between S and V n
S
                Add edges in X to T
        return the set T
Running Time:
        Prim: O((m + n) log n) with standard heaps
                  O(n log n + m) with Fibonacci Heaps
        Kruskal: O((m + n) log m) with Union−Find data structure
        Boruvka: O(m log n)
```