3340 final asnmt

1. P = babbabbabbababbabb

For next table:

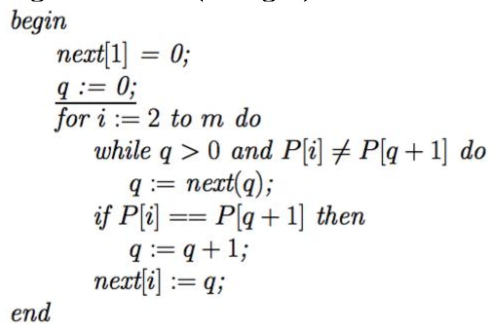
0	b
0	b a
1	b a b
1	b a b b
2	b a b b a
3	b a b b a b
4	b a b b a b b
5	b a b b a b b a
6	b a b b a b b a b
7	b a b b a b b a b b
8	b a b b a b b a b b a
9	b a b b a b b a b b a b
2	b a b b a b b a b b a b a
3	b a b b a b b a b b a b a b
4	b a b b a b b a b b a b a b b
5	b a b b a b b a b b a b a b b a
6	b a b b a b b a b b a b a b b a b
7	b a b b a b b a b b a b a b b a b b

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
P(i)	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
Next(i)	0	0	1	1	2	3	4	5	6	7	8	9	2	3	4	5	6	7

2.

Firstly, draw the next() table for string P. Next, we can start the KMP algorithm with a variable ‘count’ to count the matched elements each loop and also a global variable ‘max’ to compare with the count each time when reach a mismatch. Variable ‘count’ will be initialized by the next(i) value(in case it will not start over from P1, we need to count all matched characters before). when it reaches a mismatch or the string P has been traversed.

Algorithm next(string P):



```
begin
    next[1] = 0;
    q := 0;
    for i := 2 to m do
        while $q > 0$ and $P[i] \neq P[q+1]$ do
            q := next(q);
        if $P[i] == P[q+1]$ then
            q := q + 1;
        next[i] := q;
end
```

```
Algorithm Find_largest_match( n: len_of_T, m: len_of_P, T, P):
begin
i := 1 ;    //pointer for T
q := 0 ;   //pointer for P
count = 0;
Max=0;
while i ≤ n do
        if T[i] == P[q + 1] :
            then count++;
                  i := i + 1 ;
                  q := q + 1 ;
         else if q == 0:
               then i := i + 1 ;
         else:
               if(count>max):       //compare to the current max length of matched characters
                   max = count;
               q := next[q];
               count = q;   //count equals to the next(P[q])
         if q == m:
             then return m; //reached the longest possible matched string return the length of P

return max;  //didn't find a match for P in T return the max
end
```

3.

```
printLCS(c, x, y, i, j)
    if i = 0 || j = 0
        return
    if x[i] = y[j]
        printLCS (c, x, y, i-1, j-1)
        print x[i]
    elif c[i-1, j] >= c[i, j-1]
        printLCS (c, x, y, i-1, j)
    else
        printLCS (c, x, y, i, j-1)
```

4.

```
len_LCS(X, Y)
    m :=length[X]
    n :=length[Y]
    for i=1 to m do
        for j := 1 to n do
            c[i,j] := -1
        end for
    end for
    return find_len(X,Y,m,n)
```

```
find_len(X,Y,i,j)
    if c[i,j] > -1 then
        return c[i,j]
    end if
    if i = 0 or j = 0 then
        c[i,j] := 0
    else
        if X[i] = Y[j] then
            c[i,j] := find_len(X,Y,i-1,j-1)+1
        else
            c[i,j] := max(find_len(X,Y,i,j-1), find_len(X,Y,i-1,j))
        end if
    end if
    return c[i,j]
```

5.

By using dynamic programming, first we compute costs of all possible lines in a 2D table l[][]. The value l[i][j] indicates the cost to put words from i to j in a single line where i and j are indexes of words in the input sequences. If a sequence of words from i to j can't fit in a single line, then l[i][j] = infinite. Once we have the 2D table constructed, we can calculate total cost: c[] = min(c[i-1] + l[i, j]).

Dynamic Programming is used to store the results of subproblems. The array c[] can be computed from left to right, since each value depends only on earlier values.

The minimum cost will therefore be the cost of the line with the words plus the cost of the spaces. Since the algorithm will look through I and Wj until best cost is calculated, the time complexity of the algorithm will be O(n2).

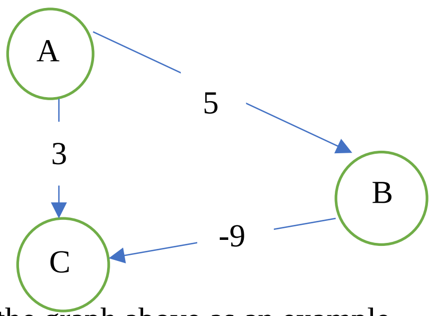
6.

To make Prim's minimum spanning tree algorithm into finding a maximum spanning tree, we only need to find the most weighted edge instead of the cheapest one each time during while loop.

```
modifiedMST Prim(G, w, r)
begin:
for each u ∈ V [G] do
    key[u] := ∞;
    π[u] := NIL;
key[r] := 0;
Q := V [G];
while Q = ∅ do
   u := Extract_Max(Q);
   for each v ∈ Adj[u] do:
      if v ∈ Q and w(u, v) < key[v] then
         π[v] := u;
         key[v] := w(u, v);
         update key[v] in Q
```

end

7.



Take the graph above as an example.
Based on Dijkstra's algorithm, we start with initializing the assigned value to each vertex:
A = 0; B = infinity; C = infinity;

Then extract vertex in Q = [A, B ,C] which has smallest value(vertex A), then the Q will be [B, C]. B and C are all adjacent to A, so we mark both of them with the values:
A = 0; B = 5; C = 3;

Then we choose the smaller one which is C. Now the Q will be [C].
A = 0; B = 5; C = 3;

Finally we extract C, after this the Q will be null. The path will be A -> B and A->C, therefore the total cost is: 3+5=8
However if we choose the path: A->B->C, the total cost will only be 5-9=4 less than 8.
So for this graph Dijkstra's Algorithm wrongly computes the distance from A to C.

8.
Yes, the all-pair shortest path still works with negative weights.
Floyd Warshall's all pairs shortest paths algorithm works for graphs with negative edge weights because the correctness of the algorithm does not depend on edge's weight being non-negative. Any path from onr vertex to another vertex, will go through any other vertex of the graph. Thus the shortest path from a to b is: min(shortest_path(a to c) + shortest_path(c to b)) for all vertices c in the graph.
As you can see there is no dependence on the graph's edges to be non-negative as long as the sub calls compute the paths correctly. And the sub calls compute the paths correctly as long as the base cases have been properly initialized.