Question 1:

Algorithm

Here is the modified KMP string matching algorithm to find the largest prefix of P that matches a substring or T:

1. Build the Next Table (Prefix Function) for P using the KMP algorithm, which is used to find all the prefixes of P that are also suffixes.
2. Define two indexes i and j and initialize i = 0 and j = 1
3. Do the following steps while $i < p.length \; and \; j \leq T.length$:
    a. Compare the first character of P with the first character of T:
        i. If $p[i] = T[j]$ increment both i and j and more to the next character
        ii. If $p[i] \neq T[j]$ set i to the value of the i-th entry in the Next table and compare the new $p[i]$ with $T[j]$
    b. Repeat the above process until either we have matched all the characters of P with some substring of T or we have reached the end of T:
        i. If matched all the characters of P: $i = p.length$ we return i which is the length of P
        ii. otherwise, we return $i - 1$, which is the length of the largest prefix of P that matches a substring of T
4. we are assuming that the first character of P (i.e., p1) is present in T. If not, return -1.

Correctness

The modified algorithm is correct because it finds the largest prefix of P that matches a substring of T. This is because, at each step, it only matches up to the current position in P, when we encounter a mismatch between $p[i] \; and \; T[j]$, we use the Next table to determine the next position to compare in P. The next position is guaranteed to be a prefix of P that is also a suffix of the current match we have make in T. Therefore, we are guaranteed to find the longest possible match up the current position in T.

Complexity

The time complexity of the modified algorithm is $O(m + n)$, where m is the length of P and n is the length of T

The algorithm iterates through T once, and each iteration takes $O(1)$ time, except for the cases where we need to use the Next table to determine the next position to compare in P. The number of times we use the Next table is at most m, since we can skip over at most $m - 1$ characters at each step. Therefore, the total time complexity is $O(m + n)$

Question 2:

Computer suffix array for string hippityhoppity

i	Suffix T[i:]	SA[i]	Suffix T[SA[i]:]
1	hippityhoppity	1	hippityhoppity
2	ippityhoppity	8	hoppity
3	ppityhoppity	2	ippityhoppity
4	pityhoppity	12	ity
5	ityhoppity	5	ityhoppity
6	tyhoppity	9	oppity
7	yhoppity	11	pity
8	hoppity	4	pityhoppity
9	oppity	10	ppity
10	ppity	3	ppityhoppity
11	pity	13	ty
12	ity	6	tyhoppity
13	ty	14	y
14	y	7	yhoppity

The suffix array for string hippityhoppity is [1, 8, 2, 12, 5, 9, 11, 4, 10, 3, 13, 6, 14, 7]

Question 3: Textbook, 14-4-2 (pp. 399)

Pseudocode LCS

$$
\begin{aligned}
&LCS(c, X, Y, i, j)\\
&\ if(i == 0 \ \| \ j == 0)\\
&\qquad return\\
&\ if(X[i] == Y[j]\\
&\qquad LCS(c, X, Y, i-1, j-1)\\
&\qquad print\ X[i]\\
&\ else\ if\ (c[i,j] == c[i-1, j])\\
&\qquad LCS(c, X, Y, i-1, j)\\
&\ else\\
&\qquad LCS(c, X, Y, i, j-1)
\end{aligned}
$$

Correctness

To reconstruct an LCS from the completed c table, this procedure examines the values of X[i] and Y[i]. If they are equal, the LCS will include this symbol, so the procedure recursively calls the LCS function with X[i-1] and Y[j-1], then prints X[i]. Otherwise, if c[i, j] equals c[i-1, j], the LCS procedure is called recursively with i-1 and j. If c[i, j] does not equal c[i-1,j], the procedure is called recursively with i and j-1 instead.

Complexity

The function includes a recursive function call that depends on the number of items in X (m) and the number of items in Y (n). Additionally, there is a constant time complexity of O(1) for the print operation. As a result, the total time complexity of the function is O(m+n).

Question 4: Textbook, 15.2-3 (pp. 430)

Algorithm

In this 0-1 knapsack problem, the order of increasing weight is the same as the order of decreasing value, which means that the value per weight ($\frac{v_i}{w_i}$) is in decreasing order ($\frac{v_i}{w_i} > \frac{v_{i+1}}{w_{i+1}}$). So, we can take the highest valued item until the total weight ≥ W (the capacity of knapsack)

Algorithm in details:

Assume the items are sorted in increasing order of weights (i.e., also in decreasing order of value).

Let i be the position of the item in the list, such that w_i= weight of item i and v_i = value of item i

As long as the capacity of knapsack W > 0, add $v_i$to the total value and subtract the item weight from capacity ($W = W - w_i$), keep doing this until $W \leq 0$. If W == 0, we have a full knapsack of items with total weight W; if W < 0, we remove the last item because it caused overflow in weight.

Correctness

This algorithm works because the approach is optimal for this particular problem. The goal is to maximize the total value of items that can be carried in a knapsack with a limited weight capacity **(**W).

Our algorithm always selects the highest valued item that can fit within the remaining weight capacity of the knapsack (W). Because the items are sorted in order of decreasing value per unit weight, we ensure that the highest valued items are also the lowest weight, and therefore this algorithm is guaranteed to choose the item with the greatest values.

Complexity

Assume the items are already sorted by increasing weight with decreasing value, the worst case is to iterate through the whole list which takes O(n)

If we need to do the sorting, the time complexity would be O(nlogn)

Question 5:

Algorithm

Let V be the set of vertices; E be the set of edges; and $u\ and\ v$ be the vertices in the graph

To find a maximum spanning tree, we can modify Kruskal's algorithm for finding a minimum spanning tree using the following steps:

1. Sort the edges in descending order using a max-heap
2. Pick the most expensive edge and mark it
3. Pick the most expensive edge available and mark it
4. Continue picking and marking the most expensive unmarked edge available that does not create a cycle
5. To check for cycles in the graph, use the union-find algorithm with $union\_find(u)\ and\ union\_find(v)$
6. To mark the selected edge use $mark\ (u, v)$
7. For each vertex v, use $make\_set(v)$ to put it in a set and connect the edges using $union(u, v)$

By following these steps, we can adapt Kruskal's algorithm to find the maximum spanning tree of a graph

Correctness

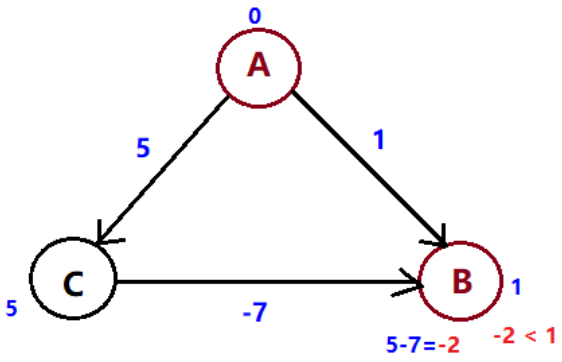
This algorithm is correct since in each step of the algorithm, the chosen edge is part of a maximum spanning tree. The edges are sorted in descending order using a max-heap, the first edge chosen will be the most expensive edge, which is guaranteed to be part of some maximum spanning tree. Subsequently, we choose the next most expensive edge available that does not create a cycle. Since we are adding the most expensive edges firs, and ensuring that at any given step of the algorithm, the edges selected so far are part of some maximum spanning tree.

Therefore, by the end of the algorithm, we have a spanning tree with the most expensive edges, which is a maximum spanning tree.

Complexity

This algorithm takes $O((|V| + |E|)\log(|V|))$ for heap operation and $O(|E|\log *(|V|))$ for $union\_find$ operation; therefore, the total time complexity is $O((|V| + |E|)\log(|V|))$

Question 6:



The counter example is provide by above graph. The graph has three vertices A, B, and C with edges $AB\ (weight = 1), AC(weight = 5), and\ CB(weight = -7)$. Starting from vertex A, there are two direct paths, AC and AB. Dijkstra's algorithm selects the edge with the smallest weight, which is AB, and marks it as the closed shortest path with edge weight 1. Next, AC is selected as the direct path. However, when we calculated the shortest path from $AC\ to\ CB$, we get a value of $5 - 7 = -2$, which is less than the weight of the closed shortest path 1. Therefore, Dijkstra's algorithm fails when there is negative weight edge in the graph.

Question 7:

Algorithm

We can use the Floyd-Warshall algorithm, which is an algorithm for finding the shortest paths between all pairs of vertices in a weighted graph. We can modify this algorithm to also find the minimum weight cycle in the graph. The algorithm is to compute a modified version of the distance matrix D where $D[i][j]$ represents the length of the shortest path from vertex i to vertex j that passes through at least one vertex in the set {1, 2…,k}. Initially, we set $D[i][j]$ to be the weight of the edge from $i\ to\ j$ if such an edge exists, and ∞ otherwise. Then, we update $D[i][j]$ as follows:

$$D[i][j] = \min\left(D[i][j], D[i][k] + D[k][j]\right)$$

for all $i, j, and\ k$. This updates $D[i][j]$ to be the length of the shortest path from $i\ to\ j$ that passes through either the vertices {1, 2…,k-1} or the vertex k. To find the minimum weight cycle, we can iterate over all pairs of vertices (i, j) and check if there is a path from $i\ to\ j$ in the modified distance matrix D that has weight less than the weight of the edge from $i\ to\ j$. If such a path exists, then there must be a cycle in the graph that includes the edge from $i\ to\ j$ and the path from $i\ to\ j$ in D. We can then recursively trace back the path in D to find the cycle with minimum weight.

Correctness

This algorithm is correct since the key idea is to use the Floyd-Warshall algorithm to compute a modified distance matrix D, which considers all possible paths that pass through at least one vertex from {1,2,…,k} and includes all cycles that use vertices from this set. This modification ensures that any cycle with minimum weight must be included in the matrix. The algorithm checks all pairs of vertices to find the cycle with minimum weigh that includes the edge from $i\ to\ j$ and the shortest path from $i\ to\ j$ in D.

Overall, the algorithm works by leveraging the Floyd-Warshall algorithm to efficiently compute a modified distance matrix that allows us to find the cycle with minimum weight in a graph with no negative cycle.

Complexity

The time complexity of this algorithm is $O((|V|)^3)$, since we need to compute and update the distance matrix D for all pairs of vertices, and iterate over all paris of vertices to find the minimum weight cycle

Question 8:

Following source code files are attached:

- Edge.java
- HeapADT.java
- Heap.java
- Dijkstra.java