Solution 10.1  

**Fibonacci Heaps.**

a) It is possible that a Fibonacci heap degenerates into a linear list. We provide an iterative algorithm that produces such a structure. Starting with a heap containing a single tree with two nodes (such a heap can be obtained by inserting three elements and then extracting the minimum), we iteratively produce a longer list. After every iteration, we obtain a heap that has exactly one node in the root list to which a linear list is appended. Let \( k \) be the key of this node, and let \( a, b, c \) be three new keys such that \( a < b < c < k \). In every iteration, we perform the following operations:

\[
\text{INSERT}(a); \ \text{INSERT}(b); \ \text{INSERT}(c); \ \text{EXTRACT-MIN}; \ \text{DECREASE-KEY}(c,a); \ \text{EXTRACT-MIN}
\]

We first add the keys \( a, b, c \) and then perform an \text{EXTRACT-MIN} operation that extracts \( a \) and cleans up the root list. The other two inserted nodes \( b \) and \( c \) (with degree 0) are combined into a new heap of degree 1. The heap containing key \( k \) as root also has degree 1, and hence the two heaps are merged into a single heap. More precisely, the heap containing the key \( k \) will be appended to the one with root \( b \), since by definition \( b < k \). This increases the height of the chain by one. The root, however, now has two children, namely one with key \( k \) and one with key \( c \). We perform a \text{DECREASE-KEY} operation setting the value of \( c \) to \( a \). This separates \( c \) from its father \( b \) and appends it to the root list, marking \( b \) in the process. Finally, we perform an \text{EXTRACT-MIN} operation removing \( a \), and we are back in the original situation.

b) This is not possible, because otherwise we would be able to sort in linear time. This could be done by first inserting all \( n \) numbers as keys, and then performing \( n \) extractions of the minimum. If both operations run in amortized constant time, we would get an overall running time of \( O(n) \) for the insert operations, and \( O(n) \) for the \text{EXTRACT-MIN} operations. The total running time would be \( O(n) \). Since we know that, when only comparisons are allowed, the running time for sorting has a lower bound of \( \Omega(n \log n) \), this is not possible.

c) We start by inserting \( 2^n + 1 \) different keys in the Fibonacci heap. Since no extraction operation was performed, the root list then consists only of nodes with degree 0. By definition, each of these nodes is a binomial tree \( B_0 \).

Now we perform an \text{EXTRACT-MIN} operation. First, the smallest key is removed from the heap, then the root list is cleaned up. During this phase, all nodes with the same degree are merged pairwise. Each time two nodes are merged, one node becomes the child of the other. This is exactly how a binomial tree is built. We start with a tree \( B_0 \) (a node with degree 0 in the Fibonacci heap) and we build a tree \( B_1 \) by merging the nodes into a heap that now has degree 1. We proceed analogously for degree \( i \). This process continues until there are only two nodes left in the root list that have the same degree. By induction, all the elements of the root list are binomial trees (note again that we only insert and delete nodes).

It is easy to see that at the end we are left with only one node in the root list with degree \( n \) containing \( 2^n \) nodes. By induction, this is exactly \( B_n \).
Solution 10.2  Christofides’ Algorithm.

Let $G = (V, E)$ be the given graph.

1) Compute a minimum spanning tree $T = (V, E_T)$ in $G$.

![Minimum Spanning Tree](image)

2) The set of vertices with odd degree in $T$ is $V' = \{b, c, d, e\}$. Compute a minimum cost perfect matching in the subgraph of $G$ induced by $V'$.

![Minimum Cost Perfect Matching](image)

We obtain $M = \{\{b, e\}, \{c, d\}\}$.

3) Create the multigraph $G' = (V, E')$ with $E' = E_T \cup M$.

![Multigraph](image)

4) Compute an Eulerian tour in $G'$. We obtain $K = (a, c, d, b, e, a)$.

5) Delete multiple occurrences of vertices from $K$. Thus, we obtain the solution

$$\pi = (a, c, d, b, e)$$

with cost 7. This solution is indeed optimal.
**Solution 10.3  Shortest Paths.**

Consider for example the following graph:

We have $c_{\text{min}} = \min_{e \in E} c(e) = -1$, thus we add the value $1 - (-1) = 2$ to every edge cost. This results in the graph

A shortest $s$-$t$ path in this transformed graph is $\langle s, u, v, t \rangle$. In the original graph, there exists no shortest $s$-$t$ path since the vertices $\langle u, v, w, u \rangle$ form a cycle with negative cost. Therefore, if an arbitrary $s$-$t$ path in the original graph is given, a path with lower cost can be obtained by traversing the cycle one more time.

**Solution 10.4  Path Planning in Labyrinths.**

We define a directed graph with a node for each possible state of the system. A state is completely determined by three parameters:

1) the position where the robot is located,

2) the direction in which the robot is facing, and

3) whether the robot is standing or moving.

For every position where the robot can be located we then have 8 possible states (4 directions multiplied by 2 possibilities whether the robot is standing or moving). We construct our graph accordingly with 8 nodes per square in the labyrinth. In addition, we need a node for the target state, representing the escape from the labyrinth. We add an edge for each movement, rotation and stopping operation, with weights according to the given duration of the operations. A shortest path in the graph from the starting state of the robot to the target state corresponds to a sequence of operations of the robot that allows it to escape as quickly as possible.

The following picture illustrates this construction on a small section of a labyrinth. The labels of the nodes indicate the direction that the robot is facing; double arrows indicate that the robot is moving. Of course, we could remove the states that are not reachable.

Since, for each position in the labyrinth, we have used a constant number of nodes (precisely 8) and a constant number of edges (at most 20), the running time of Dijkstra’s algorithm on this graph is $O(n \log n)$, where $n$ is the number of squares in the labyrinth.