Solution 9.1  Minimum Spanning Trees.

a) We obtain the following spanning tree:

```
\begin{center}
\begin{tikzpicture}
\node[draw, circle] (1) at (0,0) {1};
\node[draw, circle] (2) at (1,1) {2};
\node[draw, circle] (3) at (1,-1) {3};
\node[draw, circle] (4) at (-1,-1) {4};
\node[draw, circle] (5) at (-1,1) {5};
\node[draw, circle] (6) at (2,0) {6};
\node[draw, circle] (7) at (2,2) {7};
\node[draw, circle] (8) at (-2,2) {8};
\node[draw, circle] (9) at (-2,0) {9};
\node[draw, circle] (10) at (-2,-2) {10};
\node[draw, circle] (11) at (2,-2) {11};
\node[draw, circle] (12) at (-1,2) {12};
\node[draw, circle] (13) at (1,2) {13};
\node[draw, circle] (14) at (3,0) {14};
\node[draw, circle] (15) at (3,3) {15};
\node[draw, circle] (16) at (3,-3) {16};
\node[draw, circle] (17) at (-3,3) {17};
\node[draw, circle] (18) at (-3,-3) {18};
\node[draw, circle] (19) at (-3,0) {19};
\node[draw, circle] (20) at (-3,3) {20};
\node[draw, circle] (21) at (3,3) {21};
\node[draw, circle] (22) at (3,0) {22};
\node[draw, circle] (23) at (3,-3) {23};
\node[draw, circle] (24) at (-3,0) {24};
\node[draw, circle] (25) at (-3,-3) {25};
\node[draw, circle] (26) at (-3,3) {26};
\node[draw, circle] (27) at (3,3) {27};
\node[draw, circle] (28) at (3,0) {28};
\node[draw, circle] (29) at (3,-3) {29};
\node[draw, circle] (30) at (-3,0) {30};
\node[draw, circle] (31) at (-3,-3) {31};
\node[draw, circle] (32) at (-3,3) {32};
\node[draw, circle] (33) at (3,3) {33};
\node[draw, circle] (34) at (3,0) {34};
\node[draw, circle] (35) at (3,-3) {35};
\node[draw, circle] (36) at (-3,0) {36};
\node[draw, circle] (37) at (-3,-3) {37};
\node[draw, circle] (38) at (-3,3) {38};
\node[draw, circle] (39) at (3,3) {39};
\node[draw, circle] (40) at (3,0) {40};
\node[draw, circle] (41) at (3,-3) {41};
\node[draw, circle] (42) at (-3,0) {42};
\node[draw, circle] (43) at (-3,-3) {43};
\node[draw, circle] (44) at (-3,3) {44};
\node[draw, circle] (45) at (3,3) {45};
\node[draw, circle] (46) at (3,0) {46};
\node[draw, circle] (47) at (3,-3) {47};
\node[draw, circle] (48) at (-3,0) {48};
\node[draw, circle] (49) at (-3,-3) {49};
\node[draw, circle] (50) at (-3,3) {50};
\node[draw, circle] (51) at (3,3) {51};
\node[draw, circle] (52) at (3,0) {52};
\node[draw, circle] (53) at (3,-3) {53};
\node[draw, circle] (54) at (-3,0) {54};
\node[draw, circle] (55) at (-3,-3) {55};
\node[draw, circle] (56) at (-3,3) {56};
\node[draw, circle] (57) at (3,3) {57};
\node[draw, circle] (58) at (3,0) {58};\end{tikzpicture}
\end{center}
```

b) We prove the statement by induction over the number of vertices |V|.

\textit{Base step} (|V| = 1): A graph with exactly one vertex does not have any edges, thus we have \(|E| = 0 \leq 1 - 1 = |V| - 1\).

\textit{Inductive hypothesis}: Suppose that every undirected acyclic graph with exactly \(|V| - 1\) vertices has at most \(|V| - 2\) edges.

\textit{Inductive step} (|V| - 1 \to |V|): Consider an undirected acyclic graph \(G = (V, E)\). Because it is acyclic, there exists at least one vertex \(w\) with degree 0 or 1. Let \(V' := V \setminus \{w\}\) and \(E' := \{\{u, v\} \in E \mid u, v \in V'\}\). Obviously we have \(|V'| = |V| - 1\), and, since there is at most one edge leaving \(w\), \(|E'| \geq |E| - 1\) (i.e., \(|E| \leq |E'| + 1\)). By inductive hypothesis, we have \(|E'| \leq |V'| - 1\), thus we obtain

\(|E| \leq |E'| + 1 \leq |V'| - 1 + 1 = |V'| = |V| - 1\). \quad (1)

\[\]

c) Let \(G = (V, E, w)\) be an undirected graph where no two edges \(e, e' \in E\) have the same weight. Suppose that \(G\) had two minimum spanning trees \(T_1 = (V, E_1)\) and \(T_2 = (V, E_2)\). The edges in \(E_1 \cap E_2\) occur both in \(T_1\) as well as in \(T_2\). Let \(E'_1 = E_1 \setminus E_2\) and \(E'_2 = E_2 \setminus E_1\) be the edges that are contained only in \(T_1\) or only in \(T_2\) respectively. Since the weights of all edges are different, the set \(E'_1 \cup E'_2\) contains a uniquely determined edge \(e^\perp\) with minimum weight. W.l.o.g., assume that it is contained in \(T_1\). If it was added to \(T_2\), then by part b) \(T_2\) would contain a cycle. This cycle would contain at least one edge \(e' \in E'_2\) (otherwise it would contain only edges in \(E_1 \cap E_2\) and \(e^\perp\), thus it would also be a cycle in \(T_1\)). If we remove \(e'\) from \(T_2\) and insert \(e^\perp\) instead, we obtain a subset of edges that is still acyclic and connected (i.e., a spanning tree), but it would have a smaller weight since \(w(e^\perp) < w(e')\). Therefore \(T_2\) was not a minimum spanning tree which is a contradiction to the assumption. \quad \blacksquare
**Solution 9.2**  \textit{Fibonacci Heaps.}

a) It is possible that a Fibonacci heap degenerates into a linear list. We start 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:

\textsc{Insert}(a); \textsc{Insert}(b); \textsc{Insert}(c); \textsc{Extract-Min}; \textsc{Decrease-Key}(c,a); \textsc{Extract-Min}

We first add the keys $a, b, c$ and then perform an \textsc{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 \textsc{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 \textsc{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 \textsc{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.

**Solution 9.3**  \textit{Union-Find Structures.}

If we append smaller trees to larger trees, then for a union-find structure with height $h$ and $n$ nodes the invariant $n \geq 2^h$ is valid (Lemma 6.3, Chapter 6.2.2).

The invariant states that a tree with height $h$ must contain at least $2^h$ nodes. We can construct a tree of height $h$ and exactly $2^h$ nodes as follows: a tree of height $h = 0$ consists of $n = 1 = 2^0$ nodes. To build a tree of height $h > 0$, we merge two trees of height $h - 1$, containing exactly $2^{h-1}$ nodes each. Since one of the two trees is appended to the other, the overall height increases by one. Therefore, the new tree has height $h$ and $2 \cdot 2^{h-1} = 2^h$ nodes. The number of the required \textsc{Union} operations is given by the following recursive equation:

\[ u(0) = 0, u(h) = 2 \cdot u(h-1) + 1. \quad (2) \]

Thus, the overall number of \textsc{Union} operations is $u(h) = \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$.

It is not possible to build a tree with $2^h$ nodes using less operations, since for each additional node in the tree we need at least one \textsc{Union} operation. Since a tree of height $h$ contains at least $2^h$ nodes, it is impossible to use less than $2^h - 1$ operations to obtain a tree with height $h$. 

2
Solution 9.4  Finding an Exit in a Maze.

a) The maze is modelled by an undirected graph \( G = (V, E) \) as follows. For every position of the maze we create a vertex which is connected to the vertex of every neighbored reachable position. Additionally we create a special vertex \( t \) that is connected with the vertex of the exit. For the example on the exercise sheet we obtain the following graph: 

Let \( s_R \) be the vertex that belongs to the start position \( P_R \). We use a modified breadth first search to determine the minimum required time to reach the exit. The search starts in \( s_R \) and assigns to every vertex \( v \) the time \( \alpha[v] \) when \( v \) is considered the first time. We set \( \alpha[s_R] \leftarrow 0 \), and if a vertex \( w \) is discovered when currently located at a vertex \( v \) then we set \( \alpha[w] \leftarrow \alpha[v] + 1 \). The minimum required time to reach the exit is stored in \( \alpha[t] \).

**MINIMUMTRAVELTIME**(\( s_R \); Vertex of the start position of the robot)
\[
1 \quad \alpha[s_R] \leftarrow 0; \quad \text{for each} \ v \in V \setminus \{s_R\} \quad \text{do} \quad \alpha[v] \leftarrow \infty \\
2 \quad Q \leftarrow \text{empty Queue}; \quad \text{ENQUEUE}(Q, s_R) \\
3 \quad \text{while} \ Q \text{ not empty} \quad \text{do} \\
4 \quad v \leftarrow \text{DEQUEUE}(Q) \\
5 \quad \text{for each} \ \{v, w\} \in E \quad \text{do} \\
6 \quad \quad \text{if} \ \alpha[w] = \infty \quad \text{then} \ \alpha[w] \leftarrow \alpha[v] + 1; \quad \text{ENQUEUE}(Q, w) \\
7 \quad \text{return} \ \alpha[t] \\
\]

If \( n \) is the number of positions in the maze then the graph has \( |V| = n + 1 \) vertices and \( |E| \leq 4(n + 1) \) edges (since at most four edges are adjacent to a vertex). Thus the running time of the algorithm is \( \mathcal{O}(|V| + |E|) = \mathcal{O}(n) \).

b) We perform two breadth first searches. In a first step we determine for every vertex \( v \in V \) the time \( \beta[v] \) when the corresponding position starts burning. Let \( s_F \) be the vertex of the position where the fire breaks out. We set \( \beta[s_F] \leftarrow 0 \) and proceed simultaneously to a).

As in a) we start a breadth first search from \( s_R \) and compute for every vertex \( v \in V \) the earliest time \( \alpha[v] \) when the robot can reach the position corresponding to \( v \). Obviously the robot can escape from the fire if and only if there is exists a path from \( s_R \) to \( t \) such that \( \alpha[v] < \beta[v] \) for every vertex \( v \) on this path. Such a path exists if and only if \( \alpha[t] < \beta[t] \). Thus it suffices to check the latter condition to answer the question. 

Since we only perform two breadth first searches, the overall running time is \( \mathcal{O}(n) \).

c) If the solution of b) is applied for every possible start position we obtain an algorithm with running time \( \mathcal{O}(n^2) \). The problem can be solved more efficient if we consider the following observation from b): from a given start position the robot can escape if and only if the
distance to the exit is smaller than the distance of the fire to the exit. Thus we start a
breadth first search from the vertex $t$ and determine the distance of every position to the
exit. We also determine the distance of the fire to the exit. Finally we return all positions
that have a smaller distance to the exit than the fire has. Obviously the running time is
still $O(n)$. 