Problem 1:

(a) **High-level description of the algorithm:** Let \( t \) denote the total number of integers in the input and let \( a_1, a_2, \ldots, a_t \) denote the actual integer values (keys) themselves. Since each key has at least one digit, we have \( t \leq n \). The steps of the algorithm are as follows.

1. Let \( B[1..n] \) be an array of pointers (buckets). All pointers are initially NULL. \((B[i] \) will point to a list containing all the integers with exactly \( i \) digits, \( 1 \leq i \leq n \).\)

2. For each input value \( a_i \), determine the number of digits \( d_i \) in \( a_i \), and insert \( a_i \) at the head of the list pointed to by \( B[d_i] \) (\( 1 \leq i \leq t \)). Note that \( \sum_{i=1}^{t} d_i = n \).

3. For \( 1 \leq i \leq n \), sort each nonempty bucket \( B[i] \) into ascending order using Radix-Sort.

4. For \( 1 \leq i \leq n \), if \( B[i] \) is not NULL, output the sorted list pointed to by \( B[i] \).

(b) **Correctness:** The output is in ascending order since for every \( i \), all integers with \( i \) digits are output in ascending order before any integer with \( i + 1 \) digits.

(c) **Running time:**

   - Step 1: This step runs in \( O(n) \) time since we can initialize each bucket to NULL in \( O(1) \) time.

   - Step 2: For each input value \( a_i \), the number of digits \( d_i \) of \( a_i \) can be found in \( O(d_i) \) time using successive divisions by 10. So, the total time for finding the number of digits in all the input values is \( O(\sum_{i=1}^{t} d_i) = O(n) \). Once we have \( d_i \), key \( a_i \) can be inserted into bucket \( B[d_i] \) in \( O(1) \) time. So, the total time for inserting all the keys into buckets is \( O(n + t) = O(n) \) (since \( t \leq n \)).

   - Step 3: Consider bucket \( B[j] \). Let \( t_j \) denote the number of keys in \( B[j] \). Since each key in \( B[j] \) has \( j \) digits, Radix-Sort on \( B[j] \) takes \( O(j t_j) \) time. Therefore, the total time spent in Radix-Sort over all the buckets is \( O(\sum_{j=1}^{n} j t_j) \). Now, \( j t_j \) represents the total number of digits in all the \( t_j \) keys in \( B[j] \). Therefore, \( \sum_{j=1}^{n} j t_j \) gives the total number of digits over all the input values; that is, \( \sum_{j=1}^{n} j t_j = n \). In other words, the total time spent in Radix-Sort over all the buckets is \( O(n) \).

   - Step 4: The time needed to output all the keys is obviously \( O(n) \).

Since each of the four steps has a running time of \( O(n) \), the overall running time of the algorithm is also \( O(n) \).

Problem 2:

To avoid trivial cases, we assume that \( n \geq 2 \). We use a one-dimensional array \( C[1..n] \) of integers as our data structure. Thus, the additional space used is \( O(n) \). Using an appropriate preprocessing step, we ensure that for \( 1 \leq i \leq n \), \( C[i] \) holds the number of keys that are \( \leq i \). (This is similar to the algorithm used for Counting Sort.)
**Preprocessing Step:** We assume that the $m$ keys, each of which has a value in the range $[1 .. n]$, are stored in the array $S[1 .. m]$.

// Loop to initialize all counter values to 0.
1. for $i = 1$ to $n$ do
   $C[i] = 0$;

// Loop to ensure that $C[i]$ has the number of keys = $i$, for $1 <= i <= n$.
2. for $j = 1$ to $m$ do
   $C[S[j]] = C[S[j]] + 1$;

// Loop to ensure that $C[i]$ has the number of keys <= $i$, for $1 <= i <= n$.
3. for $i = 2$ to $n$ do
   $C[i] = C[i] + C[i-1]$;

It is obvious that after Step 2 above, $C[i]$ contains the number of keys whose value is $i$, $1 \leq i \leq n$. We can now prove by induction that at the end of Step 3, for $1 \leq i \leq n$, $C[i]$ contains the number of keys whose value is $\leq i$.

**Lemma 1:** At the end of Step 3 of the preprocessing step, for $1 \leq i \leq n$, $C[i]$ contains the number of keys whose value is $\leq i$.

**Proof by Induction:**

**Basis:** $i = 1$. At the end of Step 2, $C[1]$ contains the number of keys with value 1. Step 3 does not change $C[1]$. Since no key has value less than 1, $C[1]$ contains the number of keys with value $\leq 1$. Thus, basis holds.

**Inductive Hypothesis:** Assume that for some $k \geq 1$ and $k < n$, $C[k]$ contains at the end of Step 3, the number of keys whose value is $\leq k$.

**To prove:** At the end of Step 3, $C[k + 1]$ contains the number of keys whose value is $\leq k + 1$.

**Proof:** Since $1 \leq k < n$, we have $2 \leq k + 1 \leq n$. Thus, $C[k + 1]$ is a valid array element. In Step 3 before $C[k + 1]$ is changed, $C[k + 1]$ contains the number of keys with value equal to $k + 1$. By the inductive hypothesis, $C[k]$ contains the number of keys with value $\leq k$. Thus, the sum $C[k + 1] + C[k]$ is the number of keys with value $\leq k + 1$. Since $C[k + 1]$ is set to $C[k + 1] + C[k]$ in Step 3 and this value is not subsequently modified, it follows that $C[k + 1]$ contains the number of keys with value $\leq k + 1$. This completes the inductive proof.

**Running time of the preprocessing step:** Step 1 uses $O(n)$ time. Step 2 uses $O(m)$ time and Step 3 uses $O(n)$ time. Therefore, the total preprocessing time is $O(n + m)$.

With the above preprocessing step, the pseudo-code needed for each of the operations is straightforward.

(a) **Member($i$):** Returns `true` if there is at least one key with value $i$ and `false` otherwise.

1. if ($i == 1$) {
   if ($C[1] > 0$) return `true`;
   else return `false`;
}
else {
   if ($C[i] - C[i-1] > 0$) return `true`;
else return false;
}

(b) **Less(i):** Returns the number of keys which are strictly less than \( i \).

1. if (i == 1) return 0;
   
   else return \( C[i-1] \);

(c) **Range(i, j):** Returns the number of key values in the range \([i .. j]\). (It is assumed that \( i \leq j \)).

1. if (i == j) {
   
   if (i == 1) return \( C[1] \);
   
   else return \( C[i] - C[i-1] \);
   
   }

   else return \( C[j] - C[i-1] \).

For each of the operations above, the number of comparisons and other arithmetic operations is independent of the number \( m \) of keys and the value \( n \). Thus, the running time for each operation is \( O(1) \).

**Note:** If we add an extra element \( C[0] \) to the array \( C \) and set \( C[0] = 0 \), then we can simplify the pseudocode for the three operations. (In particular, there will be no need to check the special case \( i = 1 \).) This modification is left as an exercise.

---

**Problem 3:**

**Idea:** We use an array \( A[1 .. m] \). Suppose we were allowed to initialize each element of \( A \) to 0. Then, for each element \( x \) of \( T \), we can set \( A[x] = 1 \). Thus, \( A \) is the “bit vector” representation of set \( T \); that is, for each element \( x \in T \), \( A[x] = 1 \), and for each element \( x \notin T \), \( A[x] = 0 \). Once we have the bit vector representation, for each element \( y \in S \), we can check in \( O(1) \) time whether or not \( y \in T \).

However, we cannot initialize \( A \), since the running time of the algorithm must be \( O(n + r) \). We can achieve an “effective” bit vector representation without fully initializing \( A \) as follows. Let the set \( T \) be represented by the array \( T[1 .. r] \). Thus, \( T[1], T[2], \ldots, T[r] \) are the elements of the set \( T \). For each element \( T[j] \), we set \( A[T[j]] = j \), \( 1 \leq j \leq r \). (Thus, we initialize only \( r \) elements of \( A \); the other \( m - r \) elements of \( A \) contain unknown values.) We can now prove the following.

**Lemma 2:** An element \( x \in T \) if and only if there is an integer \( j \), \( 1 \leq j \leq r \), such that \( A[x] = j \) and \( T[j] = x \).

**Proof:** Suppose there is an integer \( j \), \( 1 \leq j \leq r \), such that \( A[x] = j \) and \( T[j] = x \). Since \( T[j] = x \), \( x \in T \). For the other direction, suppose \( x \in T \). Thus \( x = T[j] \) for some index \( j \), \( 1 \leq j \leq r \). Our method sets \( A[x] \), which is \( A[T[j]] \), to \( j \). Thus there is an integer \( j \), \( 1 \leq j \leq r \), such that \( A[x] = j \) and \( T[j] = x \).

**Description of the algorithm:** We will assume that sets \( T \) and \( S \) are both represented as arrays and that the auxiliary array \( A \) of size \( m \) (discussed above) is available.
// Construct an effective bit vector representation of T using A.
1. for j = 1 to r do
   A[T[j]] = j.

// Check whether each element of S is in T using A.
2. for i = 1 to n do
   (a) Let y = S[i] and z = A[y].

   // We must check whether y is in T. By Lemma 2, y is in T if
   // and only if z is in the range [1 .. r] and T[z] = y.

   (b) if ((z < 1) or (z > r) or (T[z] != y))
      print "S is not a subset of T" and stop.

// Here, we have verified that every element of S is also in T.
3. Print "S is a subset of T".

The correctness of the above algorithm is a consequence of Lemma 2. Step 1 takes $O(r)$ time, Step 2 takes $O(n)$ time, and Step 3 takes $O(1)$ time. So, the running time of the algorithm is $O(n + r)$. 