Solution 1

a) // POST: Computes 0 + 1 + ... + n

b) Output: 10

Explanation: The call evaluates as follows:

\[ f(1000, 2) \]
\[ \rightarrow 1 + f(500, 2) \]
\[ \rightarrow 1 + f(250, 2) \]
\[ \rightarrow 1 + f(125, 2) \]
\[ \rightarrow 1 + f(62, 2) \]
\[ \rightarrow 1 + f(31, 2) \]
\[ \rightarrow 1 + f(15, 2) \]
\[ \rightarrow 1 + f(7, 2) \]
\[ \rightarrow 1 + f(3, 2) \]
\[ \rightarrow 1 + f(1, 2) \]
\[ \rightarrow 1 + f(0, 2) \]
\[ \rightarrow 0 \]

Thus the return value of the call \( f(1000, 2) \) is the sum of the 1s in the above diagram. The function computes the number of digits of \( i \) in the number system with base \( b \). For example for \( b=2 \) it simply counts the number of binary digits that \( i \) has. (There is one slight exception though: for \( i=0 \) the function evaluates to 0 which is not the number of base-b-digits of 0, as the latter requires 1 digit as well to be represented.)

c) We can just basically fill in the numbers as in the mathematical definition. Thus

\[
\begin{align*}
expr1 & : n < k \\
expr2 & : 0 \\
expr3 & : n >= k \&\& k == 0 \\
expr4 & : 1 \\
expr5 & : n * \text{binomial}(n-1, k-1) / k
\end{align*}
\]

The only thing we have to pay special attention to is that for expr5 integer-division plays a role. Writing \( n/k * \text{binomial}(n-1, k-1) \) is wrong as \( n \) might not be divisible by
k. For example, from the mathematical definition we know that \( \binom{3}{2} \) is 
\[ \frac{3}{2} \cdot \frac{2}{1} \cdot 1 = 3. \] However, if we were using \( \frac{n}{k} \times \binom{n-1}{k-1} \) for \( \text{expr5} \), we 
would obtain \( 1 \cdot 2 \cdot 1 = 2. \)

d) The gaps can be filled as follows:

\begin{verbatim}
expr1: true
expr2: (n % 2 == 0 && is_simple(n / 2)) ||
        (n % 3 == 0 && is_simple(n / 3))
\end{verbatim}

Explanation: A given number \( n \) is simple (i.e. of the form \( n = 2^k \cdot 3^l \) for some \( k, l \in \mathbb{N}_0 \)) if and only if it is

- divisible by 2, and \( \frac{n}{2} \) is simple again, or
- divisible by 3, and \( \frac{n}{3} \) is simple again, or
- equal to 1 otherwise.

Note that being divisible by 2 holds if and only if \( k > 0 \) (and analogously for 3 and \( l \)). In \( \text{expr2} \) we use this recursive formulation to reduce the exponents \( k \) and \( l \) both step by step down to 0, and if the remaining number is 1 then \( n \) is simple.

Remark: Notice that the above choice of \( \text{expr2} \) heavily uses short-circuit evaluation to its advantage! If the left-hand-side of the && is false it doesn’t initiate unnecessary recursive calls in the right-hand-side. And thanks to short-circuit-evaluation in the || it saves many recursive calls, especially when \( n \) is indeed simple.

Solution 2

```cpp
// Informatik — Serie 10 — Aufgabe 2
// Programm: subset_sum.cpp
// Autor: ... (Gruppe ...)

#include "tests.h"
#include <iostream>
#include <cassert>

// PRE: [begin, end) is a valid range, representing a (possibly empty) set X
// POST: returns whether t = \text{sum}(S) + offset, for some subset S of X, where
// \text{sum}(S) is the sum of all elements of S
bool is_subset_sum (int t, const int* begin, const int* end, int offset)
{
    if (begin == end)
        return t == offset;
    else
        return
            is_subset_sum (t, begin+1, end, offset) ||     // check sums without first element
            is_subset_sum (t, begin+1, end, offset + *begin); // check sums with first element
```

2
Solution 3

a) Valid are (ii) and (iv); both can be obtained by suitably substituting the EBNF rules. As an example: (ii) can be obtained as follows:

Tree --> '(' Branch { Branch } ')
--> '(' Label Tree { Branch } ')
--> '(' 'a' Tree { Branch } ')
--> '(' 'a' '(' Branch { Branch } ')' { Branch } ')
--> '(' 'a' '(' 'b' { Branch } ')' { Branch } ')
--> '(' 'a' '(' 'b' ')' { Branch } ')
--> '(' 'a' '(' 'b' ')' Branch ')
--> '(' 'a' '(' 'b' ')' Label ')' 
--> '(' 'a' '(' 'b' ')' 'c' ')
--> '(a(b)c)'

The others are not valid for the given EBNF:

(i) is not valid since every Tree must start on '(' and end on ')'.

(iii) is not valid since ((b) c) would imply that the outer Tree would have the Tree (b) as its first Branch, but this is not possible since every Branch must first have a Label,
which can’t be Trees according to the third rule.

b) The gaps can be filled as follows (but of course other possibilities might work as well):
   expr1: 0
   expr2: Branch(is)
   expr3: bdepth > depth
   expr4: depth = bdepth
   expr5: 1 + Tree(is)

Solution 4

a) Terminal symbols: <, >, -, ^ (i.e. 4 terminal symbols)

b) Nonterminal symbols: viaduct, bridge, landbridge, riverbridge (i.e. 4 nonterminal symbols)

c) The extension can for example be performed by modifying the rule for riverbridge:
   riverbridge = "^" [ "^" ]
   This way a riverbridge consists of one bridge-piece "^" guaranteed and another one optionally.

d) The following is an example of a BNF that describes the same:
   viaduct = bridge | bridge viaduct
   bridge = "<" landbridge ">" | "<" riverbridge ">"
   landbridge = "-" | "-" landbridge
   riverbridge = "^" | "^" riverbridge

Note that to write the extension from subtask c) as a BNF, we could for example write the rule for riverbridge as follows:
   riverbridge = "^" | "^" "^"

Remark: It is tempting to rewrite for example the EBNF-rule
   landbridge = "-" { "-" }
   as
   landbridge = "-" | "-" "-" | "-" "-" "-" | ...  
However, this is not a valid BNF because the latter doesn’t support an infinite number of alternatives in a rule. We can only use the three dots to save us from some finite amount of typing (like in the case of exercise 3, the rule of label).