Assignment 1 – Working with Recursive Functions (4 points)

a) State PRE- and POST-conditions for the following function:

```c++
unsigned int f(const unsigned int n) {
    if (n <= 1) return n;
    else return f(n-1) + n;
}
```

[based on: Exam Summer 2011, ex.5]

b) What does the following function output for the given `main` function?

```c++
// PRE: ...  
// POST: ... 
unsigned int f (const unsigned int i, const unsigned int b) {
    if (i == 0) return 0;
    return 1 + f (i/b, b);
}

int main() {
    std::cout << f (1000, 2);
    return 0;
}
```

[based on: Exam Summer 2015, ex. 3.b]

b) What does the following function output for the given `main` function?

```c++
// PRE: ...  
// POST: ... 
unsigned int f (const unsigned int i, const unsigned int b) {
    if (i == 0) return 0;
    return 1 + f (i/b, b);
}

int main() {
    std::cout << f (1000, 2);
    return 0;
}
```

[c) Binomial coefficients can be defined in multiple ways. For example:

\[
\binom{n}{k} := \begin{cases} 
0 & \text{if } n < k \\
1 & \text{if } n \geq k, k = 0 \\
\frac{n}{k} \binom{n-1}{k-1} & \text{if } n \geq k, k > 0 
\end{cases}
\]

State expressions `expr1`, ..., `expr5` so that the resulting function computes the binomial coefficient and thus fulfills the given POST-condition!

[based on Script Exercise 125.a]
unsigned int binomial(unsigned int n, unsigned int k) {
    if (expr1) return expr2;
    if (expr3) return expr4;
    return expr5;
}

// POST: returns true if and only if n is simple, that means n is of the form 2^k * 3^l for some natural numbers k, l >= 0
bool is_simple(unsigned int n) {
    if (n == 1) return expr1;
    else return expr2;
}

Assignment 2 – Subset Sum (4 points)

The subset sum problem is the following: you are given n integers \(a_1, a_2, \ldots, a_n\) and an integer t. The question is whether there exists a subset \(S \subseteq \{1, 2, \ldots, n\}\) such that

\[ t = \sum_{i \in S} a_i \quad \text{(or } t = 0 \text{ if } S = \emptyset) \]

This problem is well-known in theoretical computer science as one of the many known NP-complete problems. Roughly speaking, an NP-complete problem is a hard problem in the sense that so far, and despite substantial efforts, no one has found an efficient algorithm for solving it; moreover, it is unlikely that such an algorithm will ever be found, since this would imply the existence of an efficient algorithm for all NP-complete problems (and there are hundreds of them).

Now what does “efficient” really mean? There is a precise definition, but in many cases, one would already be quite happy with something a bit faster than the obvious. In case of the subset sum problem, the obvious is to go through all subsets \(S \subseteq \{1, 2, \ldots, n\}\), and for each of them check whether the elements in \(S\) sum up to \(t\). As the number of subsets is \(2^n\), this will be very slow already for moderate values of \(n\). For example, suppose that \(n = 100\), and that you could check \(10^{15}\) subsets per second (this is a very optimistic estimate of what the currently fastest computer in the world can do). As there are \(2^{100} \approx 10^{30}\) subsets, you would still need \(10^{15}\) seconds, or 31 million years, to complete the task.

Well, for smaller values of \(n\), up to \(n = 10\), say, the obvious method is not so bad after all and finishes almost instantly on a normal computer. Write a program subset_sum.cpp that first asks the user for the int \(t\), and then inputs 10 ints. Then it shall determine and output whether the number \(t\) occurs among the subset sums of the 10 ints (in the I/O example below this is the case because \(3 = 1 + 2\)). The assignment here is to implement the obvious method in form of the following function:

```cpp
// PRE: n > 0
// POST: returns true if and only if n is simple, that means n is of the form 2^k * 3^l for some natural numbers k, l >= 0
bool is_simple(unsigned int n) {
    if (n == 1) return expr1;
    else return expr2;
}
```
// PRE: [begin, end) is a valid range, representing a set X of integers
// POST: returns whether t = \sum(S), for some subset S of X, where \sum(S) is
// the sum of all elements of S (or 0 if S is empty).
bool is_subset_sum (int t, const int* begin, const int* end);

You may use one or more helper functions.

<table>
<thead>
<tr>
<th>I/O-Examples</th>
<th>(Explanation: <a href="http://lec.inf.ethz.ch/ifmp/2016/codeboard.html">link</a>)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1 0 0 2 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Substitution: [link](https://codeboard.ethz.ch/ifmp16E10T2)

Assignment 3 – Prefix Trees (4 points) [based on Exam Summer 16, ex. 6]

The following EBNF defines a language for the description of prefix trees. These can be used to efficiently store a set of words by sharing common beginnings. An expression in parentheses stands for a prefix tree, characters designate edge labels. An illustrating example is shown below (bold: the words stored in the leaves of the prefix tree; every leaf of the tree corresponds to a stored word).

\[
\text{Tree} = '(', \text{Branch} \{\text{Branch}\} ')'
\]

\[
\text{Branch} = \text{Label} [\text{Tree}]
\]

\[
\text{Label} = 'a' | 'b' | \ldots | 'z'
\]

Example:

\[
(t (e (n a) o) i (n (n) f)) 
\]

\[
\iff
\]

a) Which of the following strings are valid Trees according to the EBNF?

(i) a 
(ii) (a (b) c) 
(iii) (a ((b) c) d) 
(iv) (a b (c d e) f g (h i j) k) 

b) Fill in the gaps such that the following main function returns the depth of a valid prefix tree that is provided at the input stream according to this EBNF. The depth of a tree is the length of the longest path from the root to a leaf. In the example above, the depth is 3, and this corresponds to the length of the longest stored words ten, tea, and inn. In the following code the definition of lookahead, all #include<...>, and a separate declaration of Branch before Tree are not printed due to space constraints but are present in the program!

```cpp
// POST: leading whitespace characters are extracted from is, and the
// first non-whitespace character is returned (0 if there is none)
char lookahead (std::istream& is);

// PRE: Tree = '(', Branch { Branch } ')
// POST: Extracts tree from is and returns its depth
int Tree (std::istream& is) {
    char c; is >> c; // extract '('
    
```
int depth = [expr1];
while (lookahead(is) != ')') {
    const int bdepth = [expr2];
    if ([expr3]) [expr4];
    is >> c; // extract ')
    return depth;
}

// PRE: Branch = Label [ Tree ]
// POST: Extracts single branch from is and returns its depth
int Branch (std::istream& is) {
    char c; is >> c; // extract label
    if (lookahead(is) == '(') return [expr5];
    return 1;
}

int main() {
    const int depth = Tree(std::cin);
    std::cout << "Longest stored word has length " << depth << "\n";
    return 0;
}

Assignment 4 – Bridges (4 points)
[based on: Exam Summer 2015, (new EBNF)]

The following EBNF describes simple viaducts (connections of bridges)

viaduct = bridge { bridge }
bridge = "<" landbridge ">
      | "<" riverbridge ">
landbridge = "-" { "-" }
riverbridge = "ˆ" { "ˆ" }

For example, <----><--><ˆˆˆ><---> describes a valid viaduct according to the above EBNF.
Answer the following questions:

a) List the terminal symbols in the above EBNF!

b) List the nonterminal symbols in the above EBNF!

c) How do you have to extend the above EBNF to allow for riverbridges to consist of either one or two bridge pieces "ˆ" but not more?

d) Write the above EBNF as a BNF.

Submission: https://codeboard.ethz.ch/ifmp16E10T4