1. A *complete* binary tree is one where every non-leaf vertex has exactly two children and every leaf has the same depth. Prove by induction that for a complete binary tree of height $h$ (height 0 is a single vertex, height 1 is a vertex with two children, etc.), the number of vertices is:

$$2^{h+1} - 1$$

*Theorem:* A complete binary tree of height $h$ has $2^{h+1} - 1$ vertices.

*Proof:* We use induction on $h$.

*basis:*  
If $h = 0$ then the tree has a single vertex, and there are $1 = 2^{0+1} - 1$ vertices.

*inductive step:*  
Now for height $h > 0$ assume the theorem is true for smaller heights.  
Then the tree of height $h$ is formed from a root and two sub-trees of height $h - 1$.  
By induction, each of these two subtrees has $2^{(h-1)+1} - 1 = 2^h - 1$ vertices

*conclusion:*  
Hence the entire tree has $2(2^k - 1) + 1 = 2^{k+1} - 1$ vertices.
2. Give a non-recursive algorithm that performs pre-order traversal of an arbitrary (unordered) tree of \( n \) vertices by employing a stack.

Assume that you have available the stack operations initializing an empty stack, testing if the stack is empty, PUSH, POP, and TOP (where TOP returns the top element but does not remove it). Assume also that an array representation of stacks is being used.

Recall the recursive procedure:

```
procedure PRE(v):
    {visit v}
    for each child w of v do PRE(w)
end
```

We can mimic its actions by pushing onto a stack vertices that need to be processed:

```
Initialize an empty stack \( S \).
PUSH(root)
while \( S \) is not empty do begin
    POP a vertex \( v \)
    {visit v}
    for each child \( w \) of \( v \) do PUSH(\( w \))
end
```

Or, at a lower level, we can work directly with the LCHILD-RSIB representation (and visit children of a vertex in the same order they appear in the representation):

```
Initialize an empty stack \( S \).
PUSH(root)
while \( S \) is not empty do begin
    POP a vertex \( v \)
    {visit \( v \)}
    if \( v \) has a right sibling then PUSH(right sibling of \( v \))
    if \( v \) is not a leaf then PUSH(leftmost child of \( v \))
end
```

**Time:** \( O(n) \) time since each vertex is visited once.

**Space:** Since we are making no assumptions about the shape of the tree, it could have height \( n \), and the best we can say is that in the worst case, since no vertex is pushed on the stack more than once, at most \( n \) vertices can be on the stack at any time. So to prevent overflow no matter what tree is being searched, the stack must employ an array of size \( n \), and hence \( O(n) \) space is used for the stack, in addition to the space for the tree.
3. Define the *child number* of a tree to be the maximum number of children that any vertex has. Let $T$ be an unordered tree stored with the LMCHILD-RSIB representation. Describe in English and give pseudo-code (using the PARENT, LMCHILD, and RSIB functions) for a recursive algorithm to compute the child number of $T$. Analyze the asymptotic time and space used.

We can recursively traverse a tree of $n$ vertices, and at each vertex we can take the maximum of the child number returned from each of the calls on the children and the number of children:

```plaintext
function CNUM(v):
    i := 1
    m := 0
    w := LMCHILD(v)
    while (w is not nil) do begin
        i := i+1
        m = MAX{m, CNUM(w)}
        w := RSIB(w)
    end
    return MAX{i, m}
end
```

**Time:** This is basically just a modified pre-order traversal of the tree that takes $O(n)$ time.

**Space:** Although the algorithm only explicitly uses $O(1)$ space in addition to the $O(n)$ space for the tree itself, space may also be consumed by the stack used to implement the recursion, which is proportional to the deepest vertex visited, which (since we do not know anything about the shape of the tree) could be $O(n)$.
4. For a vertex $v$ in a binary tree, let $LCHILD(v)$ denote the left child of $v$ (or $nil$ if $v$ does not have a left child) and $RCHILD(v)$ denote the right child of $v$ (or $nil$ if $v$ does not have a right child). Give a proof by induction that the following algorithm computes the height of the subtree rooted at $v$ in a binary tree (or returns $-1$ if $v$ is $nil$).

```
function BHEIGHT(v):
    if v=nil then return -1
    else return 1+MAXIMUM{BHEIGHT(LCHILD(v)),BHEIGHT(RCHILD(v))}
end
```

*Theorem:* BHEIGHT computes the height of the subtree rooted at $v$, or returns $-1$ if $v$ is $nil$.

*Proof:* We use induction on the height $h$ of $v$.

**basis:**
If $h=0$ then both recursive calls return $-1$ and hence BHEIGHT returns $0$.

**inductive step:**
Now for $h>0$ assume that the theorem is true for vertices of height less than $h$, and consider a vertex $v$ of height $h$.

A non empty subtree of $v$ has height $h-1$.

Since $v$ must have at least one subtree that is not $nil$ (since $h>0$), a return of $-1$ for a $nil$ subtree will have no effect on the MAXIMUM computation.

Hence, by induction, the two recursive calls correctly compute the heights of the sub-trees of $v$ (or return $-1$ for an empty sub-tree).

Hence, BHEIGHT correctly returns the height of $v$ as one greater than the maximum height of a subtree of $v$. 