(1) (a) (4 points) Let $T$ be the root node of a binary tree with $n$ nodes. Each node $x$ has pointers $left(x)$ and $right(x)$ to its children (these pointers can be NIL). Assume also that each node $x$ has an integer value $value(x)$. Write a recursive procedure for computing the sum of all the values in the tree rooted at $T$.

Solution:

integer Sum(T)
if T = NIL, return 0
return Sum(left(T)) + Sum(right(T)) + value(T)

(b) (3 points) Assume the tree is perfectly balanced: that is, for every node $x$ in $T$, the left subtree of $x$ has exactly the same size as the right subtree of $x$. Write a recurrence for the running time of your procedure in part (a) and solve this recurrence (in $\Theta$-notation). You can assume that addition takes constant time.

Solution: $A(n) = 2A((n-1)/2) + O(1)$. By the Master Theorem, we get $A(n) = \Theta(n)$.

Note: if you’re worried about applying the Master Theorem to $A((n-1)/2$ instead of $A(n/2)$, notice that $A(n) \leq A(n/2) + O(1)$ and $A(n) \geq A(n/2.1) + O(1)$ whenever $n$ gets big enough (i.e. greater than 21).

(c) (5 points) What is the worst-case running time (in $\Theta$-notation) when we don’t assume that the tree is balanced? Prove your answer.

Solution: Notice that, in general, $A(n) = A(L) + A(R) + O(1)$, where $L$ is the size of the left subtree and $R$ is the size of the right subtree. Clearly, since we visit every node of the tree $A(n) = \Omega(n)$. Now, we prove by induction that $A(n) \leq cn$ for some constant $c$. Assume that $A(1) = d$ for some constant $d$. Then, in the base case, where $n = 1$, we want $A(0) = d \leq c$. Since we get to choose $c$, this is no problem. Now assume that $\forall i < n, A(n) \leq cn$. First of all, $L = n - R - 1$, so $A(n) = A(n - R - 1) + A(R) + O(1)$. The values $n - R - 1$ and $R$ are each less than $n$, therefore, by the inductive hypothesis, $A(n) \leq c(n - R - 1) + cR + d'$ for some constant $d'$. Then $A(n) \leq cn - c + d'$. As long as $c \geq d'$, we’re done.
(2) Recall that a 3-coloring of a graph $G = (V,E)$ is a map $C : V \rightarrow \{ R, G, B \}$ such that any two adjacent nodes are colored differently. A 2-coloring is the same thing, except $C : V \rightarrow \{ R, G \}$. That is, there are only 2 possible colors.

(a) (6 points) Prove that a graph $G$ has a 2-coloring if and only if $G$ does not contain an odd-cycle (an odd-cycle is a cycle $(v_1,v_2), (v_2,v_3), \ldots, (v_k,v_1)$ where $k$ is odd).

Solution: First, assume that $G$ has an odd cycle of the form $(v_1,v_2), (v_2,v_3), \ldots, (v_k,v_1)$ where $k$ is odd. If we try coloring $v_1$ red, then $v_2$ has to be green, $v_3$ has to be red, etc. In general, $v_i$ will be red if $i$ is odd and green if $i$ is even. Hence $v_k$ will be red, but so is $v_1$, so there is a problem. The same problem happens when you try to color $v_1$ green. So there is no 2-coloring.

Now assume that there is no odd cycle. Use the following procedure to color the graph (repeat it on every connected component):

(1) pick a node $v_1$ and color it red
(2) color every neighbor of $v_1$ green
(3) color every uncolored neighbor of these nodes red, etc.

In general, we color a set of uncolored nodes $A_i$ in round (i) (if $i$ is odd, we color them red; if even, we color them green). Either this procedure leads to a valid coloring of the graph or there is an odd-cycle. Notice that the nodes in $A_i$ are all neighbors of nodes in $A_{i-1}$, but not neighbors of nodes in $A_1, A_2, \ldots, A_{i-2}$. Assume that at some point during this procedure, we get a conflict: that is, we color some node $v$ red (or green) and one of its neighbors $u$ red (or green) too. Then $u$ and $v$ must both be in $A_i$ for some particular $i$. Consider two paths from $v_1$ to $v$ and from $v_1$ to $u$ that both have length $i - 1$. Let $x$ be the last node that is on both of these paths. Then the paths from $x$ to $u$ and $x$ to $v$ have the same length as each other: say $k$. But this means there is a cycle of length $2k + 1$ using these two paths and the edge between $u$ and $v$. 

2
(b) (5 points) Describe how to modify the following code for DFS so that it returns “Yes” if G contains an odd-cycle and “No” if it doesn’t. That is, say what commands to add and where to insert them.

```plaintext
1 Frontier <- { x }
2 Visited[x] <- T
3 forall y != x
4 Visited[y] <- F
5 while Frontier not empty
6 y <- Pop(Frontier)
7 for z in Adj(y)
8 if Visited[z] = F
9 Push(Frontier, z)
10 Visited[z] <- T
```

Solution:

```plaintext
1 Frontier <- { x }
2 Visited[x] <- T
3 Time[x] <- T
4 forall y != x
5 Visited[y] <- F
6 while Frontier not empty
7 y <- Pop(Frontier)
8 for z in Adj(y)
9 if (Visited[z] = T AND Parent[y] != z AND
    AND (Time[y] + 1 - Time[z]) is odd)
   return ‘Yes’
10 if Visited[z] = F
   Parent[z] <- y
   Time[z] <- Time[y] + 1
11 Push(Frontier, z)
12 Visited[z] <- T
```

(c) (2 points) What is the worst-case running time of your algorithm?

**Solution:** We add only constant time operations, so the running time remains $O(n+m)$. 

(3) Given a graph \( G = (V, E) \), a vertex-cover is a subset of nodes \( S \subset V \) such that for every edge \( (x, y) \in E \), at least one of \( x \) and \( y \) is in \( S \). Our goal is to find a minimum vertex cover.

(a) (3 points) Describe an exhaustive search algorithm for vertex-cover. What is its running time?

**Solution:** A vertex cover is a subset of the nodes, so we simply go through all subsets of the nodes and check each one to see if it is a vertex cover. Whenever we find one that is, check to see if it is the smallest one we’ve seen so far. There are \( 2^n \) subsets of \( V \) and it takes \( O(n^2) \) to check if a set is a vertex cover: \( O(n^22^n) \).

(b) (6 points) Give a backtracking algorithm that runs significantly faster than the exhaustive search algorithm.

**Solution:** One solution is to notice that \( S \) is a vertex cover if and only if \( V - S \) is an independent set. Therefore, if \( T \subset V \) is a maximum independent set, then \( V - T \) is a minimum vertex cover. We could just run the backtracking algorithm for maximum independent set and take the complement of the solution.

More directly, notice that if \( x \) and \( y \) are two adjacent nodes, then we have to take at least one of them in the vertex cover. Furthermore, if \( x \) is a node with no neighbors, then we don’t want to take it.

\[
\text{VC}(G = (V,E))
\]

\[
\text{if } |V| = 0 \text{ returnemptyset}
\]

Choose some vertex \( x \)

If \( x \) has no neighbors

\[
\text{return VC}(G-x)
\]

\[
S\text{\_in} \gets \text{VC}(G-x) \cup \{x\}
\]

\[
S\text{\_out} \gets \text{VC}(G-x-\text{adj}(x)) \cup \{\text{adj}(x)\}
\]

If \( \text{size}(S\text{\_in}) > \text{size}(S\text{\_out}) \) return \( S\text{\_in} \)

return \( S\text{\_out} \)

(c) (5 points) What is the running time of the algorithm in part (b)? Explain.

**Solution:** The running time of the maximum independent set algorithm is \( O(nF_n) \). So is the running time for the second algorithm since \( T(n) \leq T(n-1) + T(n-2) + O(n) \).