Problem 1-1. (Understanding Memoization and Dynamic Programming)

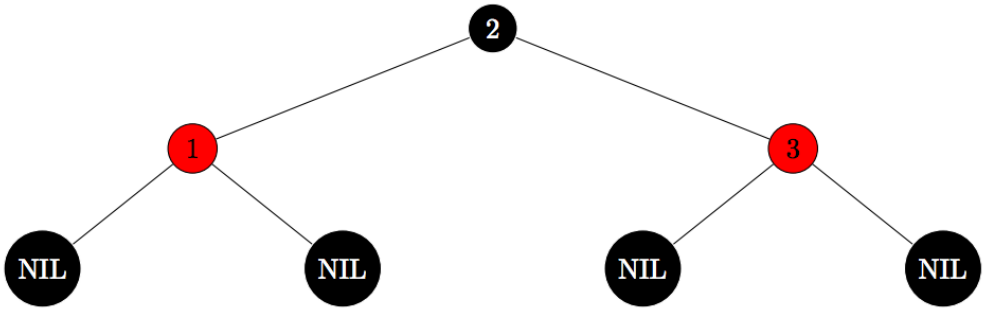
(a) Backtracking with memoization - Explain why memoization fails to speed up a good divide- and-conquer algorithm such as MERGE-SORT.
Solution: Because there are no overlapping subproblems. No duplicate work needs to be done.

(b) Party Planning - Professor G is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a friendliness rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend. Professor G is given the tree that describes the structure of the corporation. Each node of the tree holds the name of an employee and that employee's friendliness rating. Describe an algorithm to make up a guest list that maximizes the sum of the friendliness ratings of the guests. Analyze the running time of your algorithm. Does the degree of the tree matter for this runtime?
Solution: The problem exhibits optimal substructure in the following way: if node n is included in an optimal solution, then we must solve the optimal subproblems rooted at the grandchildren of n. If n is not included, then we must solve the optimal subproblems on trees rooted at the children of n. The dynamic programming algorithm to solve this problem works as follows: We make a table C indexed by vertices which tells us the optimal friendliness ranking of a guest list obtained from the subtree with root at that vertex. We also make a table G such that G[i] is the optimal guest list of the subtree rooted at i. Let T be the tree of guests. To solve the problem, we need to examine the guest list stored at G[T.root]. First solve the problem at each leaf L. If the conviviality ranking at L is positive, G[L] = L and C[L] = L.friendliness. Otherwise G[L] = $\emptyset$ and C[L] = 0. Iteratively solve the subproblems located at parents of nodes at which the subproblem has been solved. In general for node n, C[n] = max(sum of C[y] such that y is a child of n, sum of n.friendliness + C[y] such that y is a grandchild of n).
The runtime is $O(n)$ since each node appears in at most two of the sums (because each node has at most 1 parent and 1 grandparent) and each node is solved once. This does not depend on the degree.

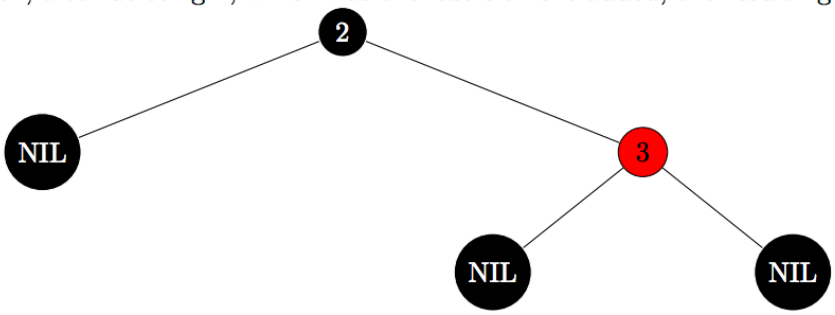
Problem 1-2. (Red-Black trees)

(a) Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree?
Solution: It is not the same. Suppose that we insert the elements 3, 2, 1 in that

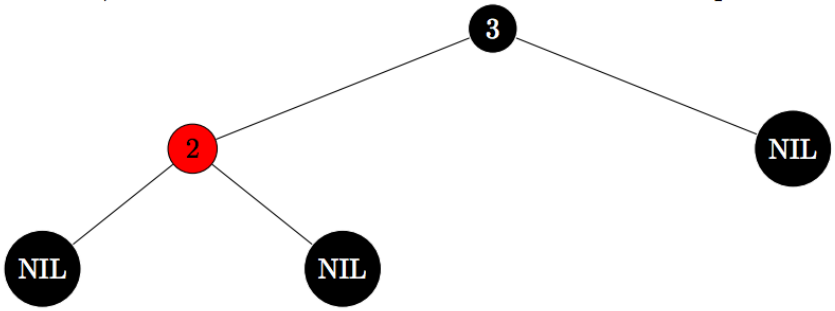
order, then, the resulting tree will look like



Then, after deleting 1, which was the last element added, the resulting tree is



however, the tree we had before we inserted 1 in the first place was



These two red black trees are clearly different

(b) What is the largest possible number of non-NIL nodes in a red-black tree with black- height k? What is the smallest possible number? **Solution:** In a path from root to leaf we can have at most one red node between any two black nodes, so the maximal height of such a tree is 2k + 1 (counting NIL in the tree height, but not the red-black height), where each path from root to leaf is alternating red and black nodes. To maximize non-NIL nodes, we make the tree complete, giving a total of $2^{height} - 1$ nodes, because we included nil in the height. This is equivalent to $2^{2k+1} - 1$ non-NIL nodes. The smallest possible number of internal nodes comes from a complete binary tree, where every node is black. This has $2^k - 1$ internal nodes, again assuming that the NIL is included in k.

Problem 1-3. (Amortized Analysis)

(a) Suppose we perform a sequence of stack operations on a standard stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.
Solution: Fo every stack operation, we charge 2 credits. First we charge the actual cost of the stack operation. Second we charge the cost of copying an element later on. Since we have the size of the stack never exceed k, and there are always k operations between backups, we always overpay by at least enough. So, the amortized cost of the operation is constant. So, the cost of the n operation is O(n).