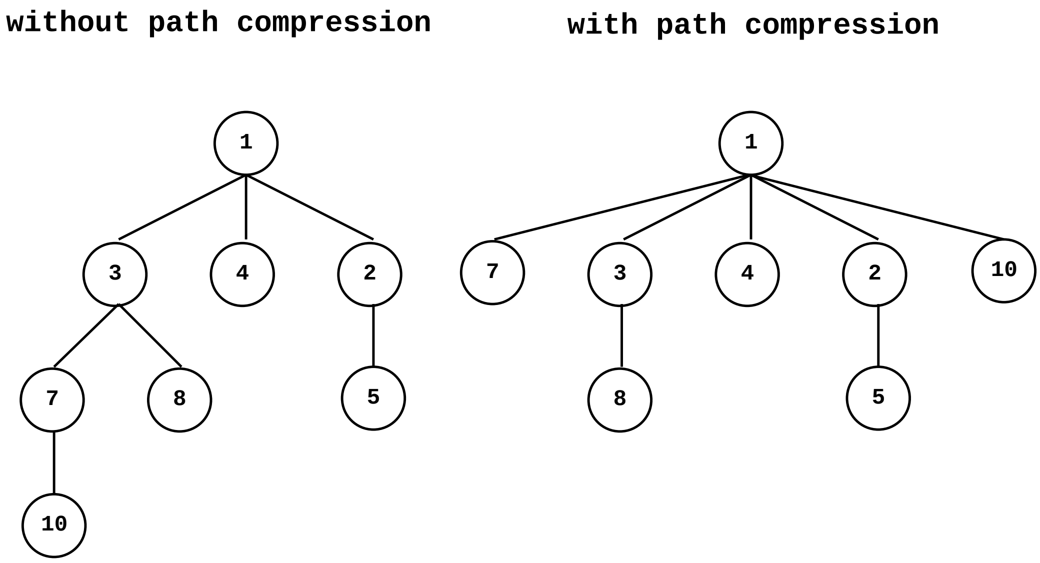
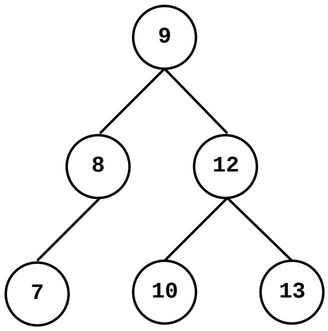
Question 1

a) Balancing means that the UNION points the root of the small tree to the root of the larger tree. Without path compression, FIND doesn't change the structure. With path compression, all nodes on a path are set to point to the root of the tree. Thus the trees look like



b) This is a LL imbalance, and a single right rotation at the root (this is the lowest point of imbalance) of the tree yields an AVL tree:



Question 2

(a) Let A be the array containing the elements we want to sort. If we pick the second largest element of A as a pivot (which takes at most $O(n)$ time), after performing the Quicksort partition of the elements (which also takes at most $O(n)$ time), we call Quicksort recursively on a sub-array which contains at least $n-2$ elements, and on an array that contain at most 1 element. An array of less than or equal to 1 elements is already sorted, so $T(1) = T(0) = 0$, if $T(n)$ is the time taken to Quicksort n elements using this pivot rule. Thus we have the following recurrence relation, for some constant $c > 0$:

$$T(n) \geq cn + T(n-2) = c(\sum_{i=0}^{\lfloor n/2 \rfloor}(n-2i)) \cong cn^2/4 = \Theta(n^2)$$

(b) Bucket sort takes time $O(n+m)$ to sort n elements which can assume at most m different values (that is, you need at most m "buckets" to sort the elements). In case $m = O(n)$, then Bucket Sort is linear on n. A contradiction to this fact might be the $\Omega(n \log n)$ lower bound which was proved in class for sorting n elements using some sorting algorithms. Note that this bound is still valid, even you know the range of the elements. But this bound was proven for comparison-based sorting algorithms only (like Quicksort, Mergesort, ...), and Bucket Sort is NOT such an algorithm.

Note here that the question asked about Bucket Sort, and not a variation of this algorithm, like Radix Sort. If you're considering Radix Sort (or some similar variation), in which you bound the number of "buckets" needed by $O(n)$, for ANY set of n elements, then the running time of your algorithm may not be linear on n anymore (e.g., for Radix Sort when you sort numbers by digits is $O(n \log n)$).

Question 3

This problem concerns $T(n,k)$, which we define to be the number of ways you can express a positive integer n as an ordered sum of k positive integers.

a) To prove: $T(n,k) = /Sigma_{i=1}^{i=n-1} T(n-i, k-1)$ for $n, k \geq 2$. Proof: The first integer in the sum must be one of $1, 2, \ldots n-1$. Once we pick this first term, which we can call i, we can chose any $k-1$ remaining terms that sum to $n-i$.

b) We can compute $T(n,k)$ by dynamic programming using an n by k table. We initialize $T(n,1) = 1$ and note the obvious fact that $T(n,k) = 0$ if $k > n$. We then use the recurrence to compute each of the $O(nk)$ values in the table in $O(n)$ time each, for a total time of $O(n^2 k)$.

We can do better by using the recurrence $T(n,k) = T(n-1, k-1) + T(n-1, k)$, which leads to an $O(nk)$ dynamic programming solution. Better still, $\binom{T(n,k)=}{n-1k-1}$.

Question 4

a) Let i be the least index such that $A[i] \geq 0$. Since $A[n] \geq 0$, such an index exists and since $A[1] < 0$, we have $i \geq 2$. By the definition of i, we have $A[i-1] < 0$ which implies that $A[i] < 2$, so we must have $A[i] = 0$ or 1.

b) The algorithm is based on binary search and is very similar to the algorithm for finding the zero of a polynomial (which we saw in class.)

The idea is : we try $A[\lfloor \frac{n}{2} \rfloor]$. If it is 0 or 1, we stop; if it is greater than 1, we know that the subarray $A[1..\lfloor \frac{n}{2} \rfloor]$ has the same property that $A[1..n]$ had and by a), it must have some i with $A[i] = 0$ or 1, so we can solve this $\lfloor \frac{n}{2} \rfloor$ size problem. A symmetric argument applies in the case that $A[\lfloor \frac{n}{2} \rfloor] < 0$.

Since we take $O(1)$ time to reduce the problem size by a factor of 2, the time taken is $O(\log n)$.

To summarize, here is the recursive function. (The program calls it with First=1 and Last=n.)

Function Search(A,First, Last)
$Mid := \lfloor \frac{First+Last}{2} \rfloor$;
If $A[Mid] < 0$ then return Search(A,Mid,Last);
else if $A[Mid] > 1$ then return Search(A,First,Mid)
else return Mid;