CS 513
Solutions to #3

1. Suppose that you are given a k-*sorted* array, in which no element is farther than k positions away from its final (sorted) position. Give an algorithm which will sort such an array. Prove its correctness. Analyse its running time. Note: your algorithm should run faster than $\Theta(n \log n)$, that is, it should take advantage of the fact that the array is k-sorted.

Answer: The algorithm is as follows:

k-**sort** (A)

Divide input array A into blocks of size k: $B_i = A[(i-1)k+1] \cdots A[ik]$.
for $i = 1$ to n/k do
Mergesort(B_i, B_{i+1})
end

end algorithm

Running time: Each call to **mergesort** is on an array of size $2k$, so it takes $\Theta(k \log k)$. Since n/k such calls are made, the total time is $\Theta(n \log k)$.

Correctness:

Claim 1 *After* **mergesort**(B_i, B_{i+1}) *has excuted, blocks* $B_1, \ldots, B_i$ *are correctly sorted and blocks* $B_{i+1}, \ldots, B_{n/k}$ *are* k*-sorted.*

Proof of claim: Base case: For $i = 0$, that is before we call **mergesort** at all, the claim is trivially true.

Inductive hypothesis: Suppose the claim is true for $i' < i$. By induction, $B_1, \ldots, B_{i-1}$ are correctly sorted. After calling **mergesort**(B_i, B_{i+1}), $B_{i+2}, \ldots, B_{n/k}$ are still k-sorted. So we need to show that B_i is sorted and B_{i+1} is still k-sorted.

B_i is sorted because the elements that end up in B_i are the k smallest elements of B_i and B_{i+1}. **mergesort**(B_i, B_{i+1}) will corectly put the k smallest elements in their place.

Now, suppose that B_{i+1} is no longer k-sorted. Let j be the index of the rightmost element of B_{i+1} that is more than k away from its correct position. First, j cannot be the last position, since each element in B_{i+1} must end up in either B_{i+1} or B_{i+2} (and all these positions are within k of the last position). Now if there are l things greater than $A[j]$ in B_{i+1}, then $A[j]$'s final position cannot be within l of the right-hand side of B_{i+2}. But then its final position is no more than k away, so $A[j]$ *is* within k of its final position, so B_{i+1} is k-sorted.□

We conclude by noting that after calling **mergesort**$(B_{n/k-1}, B_{n/k})$. We have that $B_1, \ldots, B_{n/k-1}$ is sorted and $B_{n/k}$ is k-sorted. But clearly $B_{n/k}$ is also sorted, giving that the entired array is sorted. □

2. Consider once again the k-sorted array of the previous problem. Show that any comparison based algorithm for sorting an almost sorted array makes $\Omega(n \log k)$ comparisons.

   Proof: We must count the number of distinct k-sorted arrays. We will give a lower bound for this number. We will count all the k-sorted arrays so that elements need not cross blocks (as defined in algorithm above). We call all such arrays *block k-sorted.* To recap, in these arrays, when going from a block k-sorted array to a sorted array, the correct position for all elements in block B_i is still within block B_i.

   Now, within each block, any ordering of the elements is possible, so there are $k!$ ways to order each block. That means that the number of distinct block k-sorted arrays is $(k!)^{n/k}$. So the number of k-sorted arrays is al least $(k!)^{n/k}$. So a decision tree for this problem has depth $\Omega(\log(k!)^{n/k}) = \Omega((n/k)\log(k!)) = \Omega((n/k)k \log k) = \Omega(n \log k)$.□

3. Prove that if an n node graph has two of the following properties, it has the third:

   (a) It has $n-1$ edges.

   (b) It is connected.

   (c) It is accyclic.

   Answer: First show that (2),(3) imply (1).

   Proof: Of course, by induction on n.

   Base case: If $n = 1$, then trivially there are no edges, and $0 = 1 - 1$.

   Inductive hypothesis: Suppose that each tree T' with $n' < n$ nodes has $n' - 1$ edges. Let T be an n node tree with e edges. Pick an arbitrary edge $\overline{xy}$ in T. Then removing $\overline{xy}$ leaves T disconnected into two trees. In particular, we know that x and y can no longer be connected, since if they where, adding $\overline{xy}$ back would form a cycle in T. Any node v must use $\overline{xy}$ in its path in T to either x or y (otherwise, we once again get a cycle), so we get two trees T_x and T_y, both of which have fewer than n nodes. T_x is the tree of nodes still connected x and T_y is the tree of nodes still connected to y. Let T_x have n_x nodes and e_x edges. Let T_y have n_y nodes and e_y edges. Then $e_x = n_x - 1$ and $e_y = n_y - 1$, by induction. Finally $n = n_x + n_y$ and $e = e_x + e_y + 1$ (the last one is for $\overline{xy}$), so $e = e_x + e_y + 1 = n_x - 1 + n_y - 1 + 1 = n - 1$, as desired.□

   Now show that (1),(3) imply (2).

Proof: Suppose otherwise. Let T be an accyclic graph with $n-1$ edges and k connected components $T_1, \ldots T_k$. Let T_i have n_i nodes. Each T_i is connected and accyclic so it is a tree (from first part of proof). So it has $n_i - 1$ edges. The total edges for T is $\sum_{i=1}^{k} n - i - 1 = n - k$. Therefore $k = 1$ and the graph is connected.$\Box$

Finally, (1),(2) imply (3).

Proof: Suppose otherwise. Let T be a connected graph with $n-1$ edges, and with a cycle. Pick any edge from a cycle in T and remove it. The new graph T' is still connected, but it has $n-2$ edges. If it still has a cycle, remove another edge from that cycle, and so on, until the remaining graph no longer has a cycle. This will be a tree, since it is still connected, but it will have at most $n-2$ edges, thus contradicting the first part of this proof.$\Box$

4. (a) Prove that the number of subtrees of a complete binary tree is not polynomial in the number of nodes.

   Proof: We will underestimate the number of subtrees. Let T be an n leaf binary tree. It then has $2n-1$ total nodes. Consider the class of subtrees S of T which has all the nodes in all levels but the bottom. In the bottom level, there is one subtree in S for each subset of the leaves. Clearly, each graph in S is connected, so it is a subtree of T, and clearly, T has at least $|S|$ subtrees. So the question is, how big is S? But this is the same questions as asking, how many subsets of n leaves are there? There are 2^n such subsets, so $|S| = 2^n$. So the question now is, is there a contant c such that 2^n is $O(n^c)$ – this is what it means to be polynomial in n.

   Claim 2 *For any constant c, 2^n is not $O(n^c)$.*

   Proof of claim: $\lim_{n\to\infty} \frac{2^n}{n^c} = \lim_{n\to\infty} \frac{2^n}{2^{c\log n}} = \lim_{n\to\infty} 2^{n-c\log n} = \infty$.

   (b) Give an example of a class of trees $\{T_n\}$ where the number of subtrees is a polynomial in the number of nodes.

   Answer: Let T_n be the class of n node chains. That is, if the nodes are $1, \ldots, n$, then i is connected to $i-1$ and $i+1$, for $1 < i < n$. Clearly, T_n is connected and acyclic, so it's a tree. Now, any subtree must start at some i and end at some j and include all the nodes between, so there are $\binom{n}{2}$ subtrees for T_n. This is $\Theta(n^2)$.

5. Let F_i be the ith Fibonacci number (that is $F(0) = 0, F(1) = 1, F(n+2) = F(n) + F(n+1)$). Show that $F_{n+2} = 1 + \sum_{i=0}^{n} F_i$.

Proof: By induction, the base case $F_2 = 1$ is trivial.

The inductive hypothesis is that the claim holds for all $n' < n+2$. Then $F_{n+2} = F_{n+1} + F_n = F_n + 1 + \sum_{i=0}^{n-1} F_i = 1 + \sum_{i=0}^{n} F_i$.$\Box$

6. Show that if you have a polynomial time algorithm for Hamiltonian Path, that you have a polynomial time algorithm for sorting.

   Answer: We already have a polynomial time algorithm for sorting, e.g. $O(n \log n)$ for mergersort. Thus, the implication we must prove is vacously true.$\square$

7. The *Bounded Degree Spanning Tree* (BDST) problem is the following:

   Input: Graph G and integer k.

   Output: Yes, if G has a spanning tree where every node has degree at most k, No, otherwise.

   Suppose there is no polynomial time algorithm for Hamilonian Path. Show that there is no polynomial time algorithm for BDST.

   Proof: We must give a poly time algorithm for HamP using BDST. Suppose we want a Hamiltonian path from x to y in G. Create G' from G by adding nodes x' and y' and hooking x' to x and y' to y. Now $HamP(G, x, y)$ is true iff $BDST(G', 2)$ is true. Why, because a spanning tree of degree two is exactly a hamiltonian path, and x' and y' for the spanning tree to start and end at these nodes.$\square$