Assignment 4 CS395-S13 Due: Wed Mar 27 at 12:30 pm PST

Name: _____ Points: 170

Don't forget to put your name on your paper, even when you turn it in electronically. The algorithms below are described using Python as pseudocode. Some imports may have been left out for succinctness. Some of the code is not implemented optimally but implemented for clarity or to make a point.

1. (50 pts)

```
# SIMPLE QUICK SORT ALGORITHM
def sort(x) :
    sortaux(x, 0, len(x))

def sortaux(x, left, right) :
    if right-left > 1 :
        s = partitionaux(x, left, right)
        sortaux(x, left, s)
        sortaux(x, s+1, right)
```

In class we did many quicksort algorithms using the partition function. The partition function splits a list of elements between `left` and `right-1` inclusive around a **pivot element**. In our definitions of partition, the pivot was always the left most element. The partition function then rearranged the array in place putting all the elements less than pivot to the left of the pivot and all the elements greater than the pivot to the right of the pivot. With this we were wrote a quicksort algorithm like the one above.

Using the algorithm above as a starting point modify the algorithm to put the k largest elements in array x in the right hand side of the array. This is essentially a quick sort where any partition of the array that does not include some part of the k largest elements is simply not sorted. To do this modify the `sort` function to call `sortaux` with an extra argument that is the location of the location of the k^{th} largest element. Then for each of the recursive calls put in a test before it to decide if the recursive call is needed or not in order to put the k largest elements in the right most positions. Don't forget to change sort to `sort(x, k)` as well.

2. (30 pts) Consider this sorting algorithm that uses the partition code mentioned in the previous problem:

```
# SORT A
def sort(x) :
    sortaux(x, 0, len(x))
    return x

def sortaux(x, left, right) :
    while right-left > 1 :
        s = partitionaux(x, left, right)
        sortaux(x, left, s)
        left = s+1
```

Prove that it correctly sorts by proving that it is equivalent to the quicksort algorithm. Be clear and concise but complete.

3. (20 pts) Consider sort algorithms B and C below:

```
# SORT B
def sort(x) :
    sortaux(x, 0, len(x), 0)
    return x

def sortaux(x, left, right, depth) :
    while right-left > 1 :
        s = partitionaux(x, left, right)
        # sort larger half first
        if s-left > right-(s+1) :
            sortaux(x, left, s, depth+1)
            left = s+1
        else:
            sortaux(x, s+1, right, depth+1)
            right = s
```

```
# SORT C
def sort(x) :
    sortaux(x, 0, len(x), 0)
    return x

def sortaux(x, left, right, depth) :
    while right-left > 1 :
        s = partitionaux(x, left, right)
        # sort smaller half first
        if s-left > right-(s+1) :
            sortaux(x, s+1, right, depth+1)
            right = s
        else:
            sortaux(x, left, s, depth+1)
            left = s+1
```

The two algorithms inspect the size of the right and left "halves" created by the partition and based on which is larger they decide what they will sort. That is, the difference between the two algorithms is what they choose to sort first.

Both these algorithms are recursive. One of the properties of a recursive algorithm is how many pending calls are waiting to resume. For example:

```
def fact(n) :
    if n==1 : return 1
    else : return n*fact(n-1)
```

has n calls pending at the moment of deepest nesting. This is called the **depth of recursion**.

Compare the depth of recursion for the two algorithms SORT B and SORT C. Explain how this difference happens.

4. (20 pts) If you have an array of numbers in which all the elements are no more than k positions out of place, for small fixed k, argue that insertion sort gives you $O(n)$ execution time.

5. (50 pts)

   For simplicity, assume all nodes in a digraph are reachable from node 0. Here is a DFS algorithm like what we covered in class:

```
# simple depth first search in a graph or digraph
def dfs(graph) :
    visit = array(len(graph), False)
    dfsaux(graph, visit, 0)

def dfsaux(graph, visit, node) :
    visit[node] = True
    for n in graph[node] :
        if visit[n]==False : dfsaux(graph, visit, n)
```

   Modify this algorithm to return True if the graph is 2-colorable and False if it is not. By 2-colorable I mean that each node can be assigned one of 2 colors such that adjacent nodes are never colored the same color. That is, is the graph bipartite? See problem 8 on page 129 in your book. The problem I am proposing here is not as hard as in the book because of the assumption of the graphs connectedness from node 0. The required algorithm is similar to some algorithms we have seen. Hint: You might consider if there is a way to represent "color" in the visit array.