1. We define a simple sorting algorithm as one which sorts $n$ elements into ascending order in $O(n^2)$ time.

An array $A$ holds $N = 3^p$ integers (for integer $p$). Consider the following sorting algorithm for the array $A$:

Define two overlapping sub-arrays of $A$: $A_L = A[1 \ldots \frac{2}{3}N]$, and $A_H = A[\frac{1}{3}N + 1 \ldots N]$.

1: Sort the elements of $A_L$ using a simple sorting algorithm.

2: Sort the elements of $A_H$ using a simple sorting algorithm.

3: Sort the elements of $A_L$ using a simple sorting algorithm.

(a) Prove that this algorithm does sort the elements of $A$ into ascending order.

**Answer:**

Number the array indices $1 \cdots N$. Call $R_1$ the region with indices $1 \cdots N/3$, $R_2$ the region with indices $N/3 + 1 \cdots 2N/3$, and $R_3$ the region with indices $2N/3 + 1 \cdots N$.

Assume the algorithm doesn’t work and form a contradiction.

Consider value $a_i$ that should end up in $R_3$.

If it begins in $R_1$, in the worst case, after sorting $A_L$, this value won’t make it into $R_2$ and therefore will never get into $R_3$.

To make the algorithm fail, we want to have as many values initially in $R_1$ that are larger than $a_i$. There cannot be more than $N/3 - 1$ such values if $a_i$ belongs in $R_3$.

In the worst case, $a_i$ begins in $R_1$ along with $N/3 - 1$ values larger than it. After sorting $A_L$, $a_i$ must be in position $N/3 + 1$. That puts it in $R_2$ along with the $N/3 - 1$ values larger than it than began in $R_1$.

After sorting $A_H$, the value $a_i$ must be in position $2N/3 + 1$, where it belongs.
By symmetry, any value that should end up in $R_1$ will do so due to the two steps: sort($A_H$) followed by sort($A_L$).

It’s trivial to show that any value that should end up in $R_2$ will do so.

We have a contradiction. The algorithm correctly sorts A.

(b) Give an equation for the running time of this algorithm.

**Answer:**

$$T(N) = 3k \left( \frac{2N}{3} \right)^2$$

Try just one step:

With the simple sorting algorithm, the time was $kN^2$.
With the above algorithm the time is $3k(\frac{2N}{3})^2$.

If we divide the first by the second, we get $3/4$. The new algorithm is 33% slower than the simple sorting algorithm.

(c) Give an equation for the running time of an algorithm that applies the above strategy recursively.

**Answer:** Using the equation for $T(N)$ above, and guessing $T(N) = kN^p$, we solve for $p$:

$$T(N) = kN^p = 3k \left( \frac{2N}{3} \right)^p$$

$$p = \frac{\log(1/3)}{\log(2/3)} = 2.71$$

$$T(N) = O(N^{2.71})$$

Or, by repeated substitution:

$$T(N) = 3[3T\left( \left( \frac{2}{3} \right)^2 N \right)]$$

$$= 3^2 T\left( \left( \frac{2}{3} \right)^2 N \right)$$

$$= 3^p T\left( \left( \frac{2}{3} \right)^p N \right)$$
Eventually

\[ T \left( \left( \frac{2}{3} \right)^p N \right) = T(1) \]

Then

\[ p = \frac{\log N}{\log \left( \frac{2}{3} \right)} \]

and

\[ T(N) = 3^{\log N / \log \left( \frac{2}{3} \right)} = N^{\log 3 / \log \left( \frac{2}{3} \right)} = N^{3.71} \]
2. Here is a declaration of the class Node to be used in a linked list:

```cpp
class Node {
    Thing *x;
    Node *next;
public:
    Node(Thing *myThing, Node *nextNode=NULL): x(myThing), next(nextNode) {}    
    Node *getNext() {
        return next;
    }
    void putNext(Node *x) {
        next = x;
    }
    Thing *getThing() {
        return x;
    }
    void putThing(Thing *newThing) {
        x = newThing;
    }
};
```

And here is a partial declaration of the class LinkedList:

```cpp
class LinkedList {
    Node *head;
public:
    LinkedList();
    bool find(const Thing *x) const;
    void insertAtHead(Thing *x);
    Thing *removeFromHead();  // searches for x and removes it,
    bool remove(Thing *x);    // returns true if successful
    void setUnion(LinkedList &x);  // mutates *this
};
```
(a) Write the function `remove`. You can assume that class `Thing` includes a function:

```
bool isEqual(const Thing *x)
```

**Answer:**

```cpp
bool remove(Thing *x) { // searches for x and removes it, // returns true if successful
    Node *p = head;
    Node *q = NULL;
    while(p!=NULL && !p->getThing()->isEqual(x)) {
        q = p; // make q trail 1 step behind p
        p=p->getNext();
    }
    if(p==NULL)
        return false;
    if(q==NULL) { // must be first element
        q = head;
        head = head->getNext();
        delete q;
    } else {
        q->putNext(p->getNext()); // this unstitches the node.
        p.delete();
    }
    return true;
}
```
(b) Assume that the LinkedList class is used to implement a set, and assume that the relational operators < and > do not apply to objects of the class Thing.

Write the member function `setUnion` for the LinkedList class that forms the union of two sets.

**Answer:**

```cpp
void setUnion(LinkedList &x) { // this version empties x
    Thing *t;
    while((t = x.removeFromHead())!=NULL)
        if(!find(t))
            insertAtHead(t);
}
```

3. Write a function:

```cpp
bool isAVL(Node *root)
```

that takes a pointer to the root of a binary tree and returns `true` if and only if the tree satisfies all the requirements of being an AVL tree.

Assume that the class Node has the following member functions:

```cpp
int getKey();
Node *getLeftChild();
Node *getRightChild();
```

**Answer:**

```cpp
bool isAVL(Node *p) {
    if(p==NULL)
        return true;
    return isBST(p) &&
        abs(getHeight(p->getLeftChild())-
            getHeight(p->getRightChild()))<2 &&
        isAVL(p->getLeftChild()) &&
        isAVL(p->getRightChild());
}
```
bool isBST(Node *p) { // doesn't allow repeat values
    if (p == NULL)
        return true;
    return getMaxKey(p->getLeftChild()) < p->getKey() &&
           getMinKey(p->getRightChild()) > p->getKey() &&
           isBST(p->getLeftChild()) &&
           isBST(p->getRightChild());
}

int getHeight(Node *p) {
    if (p == NULL)
        return 0;
    return 1 + max(getHeight(p->getLeftChild()),
                    getHeight(p->getRightChild()));
}

int getMaxKey(Node *p) {
    if (p == NULL)
        return -1;
    if (p->getLeftChild() == NULL && p->getRightChild() == NULL)
        return p->getKey();
    return max(getMaxKey(p->getLeftChild()),
               getMaxKey(p->getRightChild()));
}

int getMinKey(Node *p) {
    if (p == NULL)
        return MAX_INT; // largest possible key value
    if (p->getLeftChild() == NULL && p->getRightChild() == NULL)
        return p->getKey();
    return min(getMinKey(p->getLeftChild()),
               getMinKey(p->getRightChild()));
}