Divide and Conquer

1.a)

```
function maxSubsequenceDC(A, low, high)
  // Base case: if the subarray has only one element, return that element
  if low == high
    return A[low]

  // Divide the array into two halves
  mid = (low + high) / 2

  // Recursively find the maximum sum subarray in the left and right halves
  leftSum = maxSubsequenceDC(A, low, mid)
  rightSum = maxSubsequenceDC(A, mid + 1, high)

  // Find the maximum sum subarray that crosses the midpoint
  leftMax = NEGATIVE_INFINITY
  rightMax = NEGATIVE_INFINITY
  sum = 0

  for i from mid downto low
    sum = sum + A[i]
    leftMax = max(leftMax, sum)

  sum = 0
  for i from mid + 1 to high
    sum = sum + A[i]
    rightMax = max(rightMax, sum)

  crossMax = leftMax + rightMax

  // Return the maximum of leftSum, rightSum, and crossMax
  return max(leftSum, rightSum, crossMax)
```

b)

The master theorem is a useful tool for analyzing the time complexity of divide-and-conquer algorithms. It provides a straightforward way to determine the time complexity based on the recurrence relation of the algorithm.

Let's analyze the time complexity of the proposed algorithm using the master theorem:

The recurrence relation for the algorithm can be expressed as:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Where:

- $T(n)$ is the time taken to solve a problem of size n.
- $O(n)$ represents the time taken to combine the solutions of the subproblems (finding the maximum sum subarray that crosses the midpoint).

According to the master theorem, if a recurrence relation can be expressed in the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where:

- a represents the number of subproblems.
- b represents the factor by which the problem size is reduced in each recursive call.
- $f(n)$ represents the time ta' to divide the problem and combine the solutions.

Then, the time complexity of the algorithm can be determined as follows:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af\left(\frac{n}{b}\right) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

In our case:

- $a = 2$ (because we divide the problem into two subproblems).
- $b = 2$ (because we divide the problem size in half in each recursive call).
- $f(n) = O(n)$ (because the time taken to combine the solutions is linear).

So, $\log_b a = \log_2 2 = 1$.

Since $f(n) = O(n) = O(n^{\log_b a-\epsilon})$ for $\epsilon = 0$, we can apply case 1 of the master theorem.

Therefore, the time complexity of the algorithm is $T(n) = \Theta(n^{\log_b a}) = \Theta(n^1) = \Theta(n)$.

So, the time complexity of the ↓ rithm is $O(n)$.

2.a)

function hanoi(n, source, destination, auxiliary):

```
    if n == 1:
        // Move the single disk from source to destination
        print "Move disk 1 from", source, "to", destination
        return
    hanoi(n-1, source, auxiliary, destination) // Move top n-1 disks from source to auxiliary
    print "Move disk", n, "from", source, "to", destination // Move the largest disk from source to destination
    hanoi(n-1, auxiliary, destination, source) // Move top n-1 disks from auxiliary to destination
```

2.b)

To prove that for n pegs, at most $2^n - 1$ moves are needed, we can use mathematical induction.

Base Case: When $n = 1$, there is only one disk. In this case, the minimum number of moves needed is 1, which matches the formula $2^1 - 1 = 1$. So, the base case holds true.

Inductive Step: Assume that for $n = k$, the number of moves needed is $2^k - 1$.

Now, let's consider $n = k + 1$. We can solve the Tower of Hanoi problem with $k + 1$ disks using the following steps:

1. Move the top k disks from the source peg to an auxiliary peg.
2. Move the largest disk from the source peg to the destination peg.
3. Move the k disks from the auxiliary peg to the destination peg.

Steps 1 and 3 involve moving k disks, which, by the inductive hypothesis, requires $2^k - 1$ moves each. Step 2 involves moving one disk, which requires 1 move. So, the total number of moves for $n = k + 1$ is:

$$2 \times (2^k - 1) + 1 = 2^{k+1} - 2 + 1 = 2^{k+1} - 1$$

Thus, by induction, we have proved that for n pegs, at most $2^n - 1$ moves are needed.

Thus, by induction, we have proved that for n pegs, at most $2^n - 1$ moves are needed.

Regarding the algorithm's time complexity with respect to the number of disks, n, each recursive call involves solving two subproblems with $n - 1$ disks, which results in a time complexity of $O(2^n)$. This is because each recursive call branches into two recursive calls, and the number of levels in the recursion tree is n. Therefore, the time complexity of the algorithm is exponential, $O(2^n)$.

3)

Let's analyze the running times of each algorithm:

1. **Algorithm A**:
   - It divides the problem into 5 subproblems of half the size, meaning the problem size reduces to $\frac{n}{2}$ in each recursive call.
   - Recursively solving each subproblem takes $T(\frac{n}{2})$ time.
   - Combining the solutions in linear time takes $O(n)$ time.
   - So, the recurrence relation for Algorithm A is $T(n) = 5T(\frac{n}{2}) + O(n)$.
   - By the Master Theorem, the running time of Algorithm A is $O(n^{\log_2 5})$, which is approximately $O(n^{2.32})$.
2. **Algorithm B**:
   - It divides the problem into 2 subproblems of size $n-1$, meaning the problem size reduces by 1 in each recursive call.
   - Recursively solving each subproblem takes $T(n-1)$ time.
   - Combining the solutions in constant time takes $O(1)$ time.
   - So, the recurrence relation for Algorithm B is $T(n) = 2T(n-1) + O(1)$.
   - By expanding the recurrence relation, we can see that Algorithm B runs in $O(2^n)$ time.

3. **Algorithm C**:
   - It divides the problem into 9 subproblems of size $\frac{n}{3}$, meaning the problem size reduces to $\frac{n}{3}$ in each recursive call.
   - Recursively solving each subproblem takes $T(\frac{n}{3})$ time.
   - Combining the solutions in $O(n^2)$ time.
   - So, the recurrence relation for Algorithm C is $T(n) = 9T(\frac{n}{3}) + O(n^2)$.
   - By the Master Theorem, the running time of Algorithm C is $O(n^2)$.

For large values of n, Algorithm A would be the most efficient choice, as it has a faster growth rate compared to Algorithms B and C. However, it's important to note that the actual performance can vary based on constant factors, memory requirements, and other practical considerations.