Decrease and Conquer Practice

**Algorithm**: INSERTIONSORT(A)

//Input: an array A[0..n − 1]

for i ← 0..n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] < v do
        j ← j − 1
    A[j + 1] ← v

In-place? Stable?

Size Metric:

Basic operation:

Best/average/worst cases?
Number of comparisons, worst case:
\[ C_{\text{worst}}(n) = \]

Number of comparisons, best case:
\[ C_{\text{best}}(n) = \]

How likely is this? In what use cases is this more likely?

Number of comparisons, average case (on random or-dered data):
\[ C_{\text{avg}}(n) \approx \]
ALGORITHM FACTORIAL(n)
    if \( n = 0 \) then
        return 1
    else
        return \( n \cdot \text{FACTORIAL}(n - 1) \)

Recurrence:
\[ M(n) = \]

Base case/initial condition:
\[ M(\quad) = \]

Back substitution steps:

Pattern:
\[ M(n) = \]

Application of base case, and result:
\[ M(n) = \]
Towers of Hanoi
Recall that solving an instance of this problem for $n$ disks involves solving an instance of the problem of size $n - 1$, moving a single disk, then again solving an instance of the problem of size $n - 1$. We denote the number of moves to solve the problem for $n$ disks as $M(n)$.

Recurrence with base case:

$M(n) =$

$M( ) =$

Backward substitution:

$M(n) =$

Pattern:

$M(n) =$

Application of base case and result:

$M(n) =$
ALGORITHM BinDigits(n)
    if \( n = 1 \) then
        return 1
    else
        return BinDigits(\( \lfloor \frac{n}{2} \rfloor \)) + 1

Recurrence with base case:
\[
A(n) = \\
A(\quad) =
\]

Convert to a power of 2:
\[
A(2^k) = \\
A(2 \quad) =
\]

Backward substitution:
\[
A(2^k) =
\]

Pattern:
\[
A(2^k) =
\]

Application of base case and result, convert back to \( n \):
\[
A(2^k) = \\
A(n) =
\]
Another example of a common pattern

\[ C(n) = 2C(n/2) + 2 \]

when \( n > 0 \), and a base case of \( C(1) = 0 \).
**ALGORITHM INSERTIONSORT(A)**

//Input: an array A[0..n − 1]
recInsertionSort(A, n − 1)

**ALGORITHM RECINSERTIONSORT(A, max)**

//Input: an array A[0..n − 1]
//Input: upper index limit to sort max
// Base case: a 1-element array
if max = 0 then
    return
// Recursive case: sort first max − 1
RecInsertionSort(A, max − 1)
// now insert max’th in correct location
v ← A[max]
j ← max − 1
while j ≥ 0 and A[j] > v do
j ← j − 1
A[j + 1] ← v

Worst case recursive analysis for the number of comparisons: