1. Solve the following recurrence relations. For parts (a) and (b), provide an exact solution. For parts (c) and (d), provide an asymptotic upper bound. For both cases, your solution must explain how you obtained the expression.

   (a) $A(n) = A(n-1) + 2n + 1; A(0) = 0$

Solution: We will unroll the recurrence:

$$\begin{aligned}
A(n) &= A(n-1) + 2n + 1 \\
&= A(n-2) + 2(n-1) + 1 + 2n + 1 &&= A(n-2) + 4n + 1 - 1 \\
&= A(n-3) + 2(n-2) + 1 + 4n &&= A(n-3) + 6n + 1 - 4 \\
&= A(n-4) + 2(n-3) + 1 + 6n - 3 &&= A(n-4) + 8n + 1 - 9 \\
&\cdots \\
&= A(n-k) + 2kn + 1 - (k-1)^2 \\
&= A(0) + 2n^2 + 1 - (n-1)^2 \\
&= n^2 + 2n
\end{aligned}$$

■

   (b) $B(n) = B(n-1) + n(n-1) - 1; B(0) = 0$

Solution: We will unroll the recurrence:

$$\begin{aligned}
B(n) &= B(n-1) + n^2 - n - 1 \\
&= B(n-2) + (n-1)^2 - (n-1) - 1 + n^2 - n - 1 \\
&= B(n-3) + (n-2)^2 - (n-2) - 1 + (n-1)^2 - (n-1) - 1 + n^2 - n - 1 \\
&\cdots \\
&= B(n-k) + \sum_{i=0}^{k-1} \left((n-i)^2 - (n-i) - 1\right) \\
&= B(0) + \sum_{i=0}^{n-1} \left((n-i)^2 - (n-i) - 1\right) = \sum_{m=1}^{n} \left(m^2 - m - 1\right) \\
&= n(n+1)(2n+1)/6 - n(n+1)/2 - n = n(n^2-4)/3
\end{aligned}$$

■

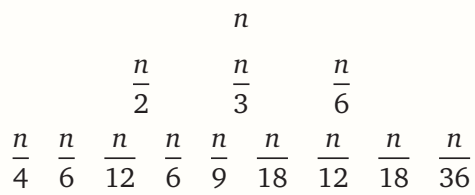
Solution:

$$\begin{aligned}
B(n) &= B(n-1)+n(n-1)-1 = B(n-1)+2\binom{n}{2}-1 \\
&= B(n-2)+(n-1)(n-2)-1-2\binom{n}{2}-1 \\
&= B(n-2)+2\binom{n-1}{2}+2\binom{n}{2}-2 \\
&\cdots \\
&= B(n-k)+2\sum_{i=0}^{k-1}\binom{n-i}{2}-k \\
&= B(0)+2\sum_{i=0}^{n-1}\binom{n-i}{2}-n = 2\sum_{m=1}^{n}\binom{m}{2}-n \\
&= 2(n^3-n)/6-n = (n^3-4n)/3
\end{aligned}$$

where we used the fact that: $\sum_{m=1}^{n}\binom{m}{2} = n(n^2-1)/6$. ■

(c) $C(n) = C(n/2)+C(n/3)+C(n/6)+n$

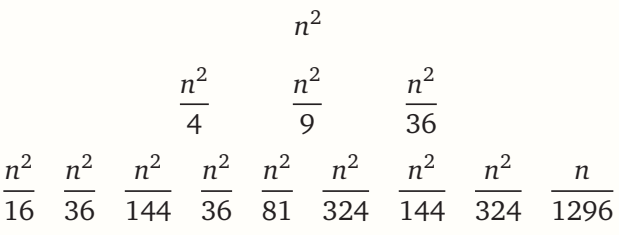
Solution: We model this recurrence as a recursion tree.



At each level, the nodes sum up to n. The maximum depth of the tree is $\log n$. Therefore, we have $\log n$ levels with n work per level, so the recurrence is $O(n \log n)$. ■

(d) $D(n) = D(n/2)+D(n/3)+D(n/6)+n^2$

Solution: We model this recurrence as a recursion tree.



At the root, the work sums up to n^2. At level 2, the work sums up to $\frac{7}{18}n^2$. At level 3, the work sums up to $\frac{49}{324}n^2$. At level i, the work sums up to

$(\frac{7}{18})^i n^2$. Hence, the work at each level is a decreasing geometric series. The overall recurrence is dominated by the root node. Therefore, the recurrence is $O(n^2)$. ■

2. Consider the following variants of the Towers of Hanoi. For each of variant, describe an algorithm to solve it in as few moves as possible. **Prove** that your algorithm is correct. Initially, all the n disks are on peg 1, and you need to move the disks to peg 2. In all the following variants, you are not allowed to put a bigger disk on top of a smaller disk.

   (a) **Hanoi 1**: Suppose you are forbidden to move any disk directly between peg 1 and peg 2, and every move must involve (the third peg) 0. Exactly (i.e., not asymptotically) how many moves does your algorithm make as a function of n?

Solution: The following recursive algorithm moves the top n disks from the source peg s (either 1 or 2) to the destination peg d (either 1 or 2), where every move uses peg 0. (The forbidden peg never changes, so we can hard-code it into the algorithm.)

```
Hanoi_1(n, s, d):
  if n = 0 then return

  Hanoi_1(n − 1, s, d)
  move disk n from peg s to peg 0
  Hanoi_1(n − 1, d, s)
  move disk n from peg 0 to peg d
  Hanoi_1(n − 1, s, d)
```

The initial call is $Hanoi_a(n, 1, 2)$.

The number of moves satisfies the recurrence $T(n) = 3T(n-1) + 2$, with $T(1) = 2$. We can easily verify by induction that $T(n) = 3^n - 1$.

As for proof of correctness, the trick is to be concise.

Proof: The procedure is clearly correct for $n = 1$. Assume it works correctly for $n < k$, and consider the case $n = k$. The first recursive calls succeeds by induction, and we only have to worry about the moves being legal. Moving disk n to from s to 0 succeeds, since all the first $n-1$ disks are on peg $d \neq 0$. Similar argumentation implies that the other recursive call are successful, and the move from peg 0 to peg d of n is valid (as all the first $n-1$ disks are on peg s at this point in time. As such, by induction, all the moves performed by the algorithm are legal, and clearly the first n disks end up on peg d, which implies the claim. □

(b) **Hanoi 2**: Suppose you are only allowed to move disks from peg 0 to peg 1, from peg 1 to peg 2, or from peg 2 to peg 0.

Provide an upper bound, as tight as possible, on the number of moves that your algorithm uses.

(One can derive the exact upper bound by solving the recurrence, but this is too tedious and not required here.)

Solution: Here $[\![n]\!] = \{1, 2, \ldots, n\}$. We can arrange the pegs in a circle. In order to move the disks one peg clockwise, you need to move the top $n-1$ discs two pegs clockwise, move the n^{th} disc one peg clockwise, then move the $n-1$ discs two pegs clockwise again. Hence, we can formulate the following two-part algorithm:

```
Move+1(n, s):
    if n = 0 then return
    Move+2(n−1, s) // Disks [[n−1]] are on peg s+2
    move disk n from peg s to peg (s+1) % 3      (L1)
    Move+2(n-1, (s+2) % 3)

Move+2(n, s):
    if n = 0 then return
    Move+2(n−1, s) // Disks [[n−1]] are on peg s+2
    move disk n from peg s to peg (s+1) % 3
    Move+1(n−1, (s+2) % 3) // Disks [[n−1]] are on peg s
    move disk n from peg (s+1) % 3 to peg (s+2)%3
    Move+2(n−1, s) // Disks [[n]] are on peg (s+2) % 3
```

The initial call is $\textsc{Move+1}(n, 1)$ which will move the first n disks from peg 1 to peg 2.

Proof (of correctness): We prove by induction on n that all the moves performed by both procedures are legal. The claim is immediate for $n = 1$. So assume both procedures works correctly for $n < k$, and consider the case $n = k$.

Arguing as above, when $\textsc{Move+1}(n, s)$ is being called, when reaching (L1), all the first $n-1$ disks are on peg $(s+2)$ % 3, as all moves performed by the call $\textsc{Move+2}(n-1, s)$ are valid by induction. As such, one can move disk n from peg s to peg $(s+1)$ % 3 safely. The rest of moves in the second recursive calls are also valid by induction. Which implies that $\textsc{Move+1}(n, s)$ performs only legal moves.

A similar inductive argument shows the correctness of $\textsc{Move+2}(n, s)$.

Now, since all the disk moves performed by the algorithm are valid, it easy to verify that $\textsc{Move+1}(n, 1)$ indeed moves the n disks from peg 1 to peg 2, as desired. □

Bounding the number of moves. Let $T_1(n)$ and $T_2(n)$ be the number of moves performed by $\textsc{Move+1}(n, \cdot)$ and $\textsc{Move+2}(n, \cdot)$, respectively.

We have the following two recurrence relations:

$$T_1(n) = 2T_2(n-1) + 1$$
$$T_2(n) = 2T_2(n-1) + T_1(n-1) + 2$$

Substituting the first equation into the second yields:

$$T_2(n) = 2T_2(n-1) + 2T_2(n-2) + 3.$$

The solution to this recurrence is

$$T_2(n) = \frac{(1+\sqrt{3})^{n+2} - (1-\sqrt{3})^{n+2}}{4\sqrt{3}} - 1 \leq (1+\sqrt{3})^n \leq 2.733^n.$$

This means $T_1(n) = \frac{(1+\sqrt{3})^{n+1} - (1-\sqrt{3})^{n+1}}{2\sqrt{3}} \leq 2.733^n$.

(c) **Hanoi 3**: Finally consider the disappearing Tower of Hanoi puzzle where the largest remaining disk will disappear if there is nothing on top of it. The goal here is to get all the disks to disappear and be left with three empty pegs (in as few moves as possible).
Provide an upper bound, as tight as possible, on the number of moves your algorithm uses.

Solution: The intuition for this puzzle is to move $n-2$ discs to another peg and then move the $n-1$ disk to the third empty peg. At this point the two largest disks will be on separate pegs without any disks on top of them and thus, disappear. You can use the classic Hanoi algorithm to move the first $n-2$ disks and use the algorithm below to complete the puzzle:

```
Hanoi_c(n, s, d):
  if n ≤ 1 then
    return
  t ← peg that is not s or d
  HanoiReg(n−2, s, t)
  Move disk n−1 from s to d
  //disks n and n−1 disappear
  Hanoi_c(n−2, t, d)
```

The initial call is $Hanoi_c(n, 1, 2)$. Here $HanoiReg(n, \cdot, \cdot)$ is the regular Hanoi algorithm which takes $2^n - 1$ moves.

Proof (of correctness): Follows readily by using the same argumentation as above. □

Knowing that the standard Hanoi function takes $2^n - 1$ moves, the recurrence for the number of moves performed by the algorithm is

$$T(n) = 2^{n-2} - 1 + T(n-2).$$

With $T(1) = 0$, $T(2) = 1$ and $T(3) = 2$. By repeated opening, we have that

$$T(n) = 2^{n-2} + 2^{n-4} - 2 + T(n-4) = \sum_{j=1}^{i} 2^{n-2j} - i + T(n-2i).$$

If $n = 2k+1$, then

$$T(n) = \sum_{j=1}^{k} 2^{n-2j} - k + T(n-2k) = 2(1+4+\cdots 4^{(k-1)}) - k + T(1) = \frac{2(4^k-1)}{3} - k$$
$$= \frac{2(4^{\lfloor n/2 \rfloor}-1)}{3} - \lfloor n/2 \rfloor \leq \frac{2^n}{3}.$$

If $n = 2k$, then

$$T(n) = \sum_{j=1}^{k-1} 2^{n-2j} - k + 1 + T(n-2(k-1))$$
$$= 4 + 4^2 + \cdots + 4^{(k-1)} - k + 1 + T(2) = \frac{4(4^{k-1}-1)}{3} - k + 2$$
$$= \frac{4^{\lfloor n/2 \rfloor}-4}{3} - \lfloor n/2 \rfloor + 2 \leq \frac{2^n}{3}.$$

3. Suppose we are given an array $A[1..n]$ of n integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] \leq A[2] \leq \cdots \leq A[n]$. Suppose we wanted to count the number of times some integer value x occurs in A. Describe an algorithm (as fast as possible) which returns the number of elements containing value x.

Solution: Dumb Approach: We could simply iterate through the array and count the number of times x appears. This would take $O(n)$ time.

Better Approach: First we can use binary search to find an instance of x. Then since A is sorted, All values of x appear next to one-another. Hence, if we find one instance of x, we can interate over the block of x instances and count the size. This will take $O(log(n)+k)$ time where k is the number of array elements containing x. The one issue is that if k is large, i.e. on the order of n, then the runtime reduces to $O(log(n)+k) = O(log(n)+O(n)) = O(n)$.

Best Approach: We can slightly modify binary search to find the leftmost array element that contains x (the left-bound of the array block):

```
FindLeftBound(A[1..n], x, i):
  if A[1] = x
      return i
  else
      if A[n/2] ≥ x
          return FindLeftBound(A[1, ..., n/2], x, i)
      else
          return FindLeftBound(A[n/2 + 1, ..., n], x, i + n/2)
```

i is a variable to keep track of the original position of the sub-array beign currently evaluated. We do the same to find the right bound and subtract the two values from one another to find the number of instances of x. ■

4. Given an arbitrary array $A[1..n]$, describe an algorithm to determine in $O(n)$ time whether A contains more than $n/4$ copies of any value.

Solution: The algorithm is formally described below. We use the fact that the selection problem can be solved in linear time. That is, given an unsorted array A of n values and an index j between 1 and n, we can find the j-th ranked element in A in $O(n)$ time. We denote this black box algorithm as SELECT(A[1..N], j) which returns the value of the j-th ranked element in A. To determine whether an element appears more than $n/4$ times, we select values with rank $n/4$, $2n/4$, and $3n/4$. If an element x appears more than $n/4$ times, it follows that at least one of these selected values is equal to x. Thus, we can scan and count the number of occurrences of each of these selected values.

```
Contains4Duplicates(A[1..N])
    x_1 ← SELECT(A, ⌈N/4⌉)
    x_2 ← SELECT(A, ⌈2N/4⌉)
    x_3 ← SELECT(A, ⌈3N/4⌉)
    for (i ← 1 : 3)
        count ← 0 for (j ← 1 : N)
            if (A[j] = x_i) then count++1
        if (count > N/4) then return True
    return False
```

Since SELECT runs in $O(n)$ time, finding x_1, x_2, and x_3 also takes $O(n)$ time. Looping over the array of length n a total of 3 times takes $O(n)$ time. Thus, this algorithm runs in the required $O(n)$ time.

To prove correctness of the algorithm, we must show that if an element appears more than $n/4$ times, it must be at least one of the selected values with rank $\lceil n/4 \rceil$, $\lceil 2n/4 \rceil$, or $\lceil 3n/4 \rceil$. Assume an element x appears $i > n/4$ times. Then, there must be consecutive ranks $j, ..., j+i-1$ with value x. Without loss of generality, consider the number of values of rank between $\lceil n/4 \rceil$ and $\lceil 2n/4 \rceil$ (excluding the outside values). Since $\lceil n/4 \rceil \geq n/4$ and $\lceil 2n/4 \rceil \leq 2n/4+1$, the maximum number of values is given by $(2n/4+1)-(n/4)-1 = n/4$. Thus, there are at most only $n/4$ spots for more than $n/4$ values. By pigeonhole principle, one of the selected values must be equal to x. ■