CS/Math 240: Intro to Discrete Math 3/10/2011

Solutions to Homework 5

Instructor: Dieter van Melkebeek

Problem 1

Part a

We begin by examining the BubbleSort algorithm and making general comments about its behavior. We note that the algorithm works by 'pushing' larger elements to the right end of the array in waves; each wave is represented by an execution of the nested loop. During this wave, a pointer ℓ keeps track of the index where the last 'push' occured; this information determines the range of indices the next wave operates on. Intuitively, this implies that all array elements past the pointer ℓ at each iteration are in their correct sorted position. The algorithm terminates either when no pushes occur in a wave, or the only push that occurs is between the first two elements. It can be seenthat in either of these conditions, the array is sorted.

As stated earlier, the nested loop pushes larger elements to the right end of the array. More formally, we can state the following invariant holds at each execution of the while condition in line (5):

Invariant A: $0 \leq i \leq m-1$, $0 \leq \ell \leq i$, *and all elements from* $A[\ell ... i]$ *are* $\geq$ *than any element from* $A[0 ... \ell - 1]$ *and sorted amongst themselves.*

In the base case, P(0), we see that this holds; $i = 0$, $\ell = 0$, and $A[0 ... -1]$ is the empty array. Assuming P(k), we must show P(k+1).

We know that if $i = m-1$ at the $(k+1)$ iteration, the while loop would not be entered; so $i < m-1$ at the outset of the loop. i is incremented by one in the loop, so we have $i \leq m-1$. Thus the first condition of the invariant always holds. To prove the second half of the invariant, we will use a proof by cases.

Case One: $A[i] > A[i+1]$ To clarify our explanation, we will use i_k to denote our initial value of i at the outset of the loop, and i_{k+1} as the value at the end of the iteration. In this case, $A[i]$ is swapped with $A[i+1]$, and $l \leftarrow i+1$. After $i \leftarrow i+1$, $A[l ... i_{k+1}]$ is simply $A[i+1]$. From P(k), we know that $A[i+1]$ (which was formerly $A[i_k]$) is at least as large as any element from $0 ... i-1$. From the if condition, $A[i+1]$ is also $\geq$ than $A[i]$ after the swap. So $A[i+1]$ is larger than any element from $A[0 ... \ell - 1] = A[0 .... i]$, and is trivially sorted among itself.

As we previously stated, in this iteration we have $\ell \leftarrow i+1$, then $i \leftarrow i+1$; so $\ell = i$ at the execution of the while condition in line (5). Thus all conditions of the invariant are satisfied in this case.

Case Two: $A[i] \leq A[i+1]$ In this case, no swap occurs. ℓ is unchanged, and i inceases, so $0 \leq \ell \leq i$ holds.

From the case condition, we know that $A[i+1] \geq A[i]$ and by extensions all elements in the range $A[0 ... i]$. The inductive hypothesis gives us $A[\ell ... i]$ larger than any element from $A[0 ... \ell]$, and sorted within $A[\ell ... i]$. These two facts combined give us $A[\ell ... i+1]$ larger than any element from $A[0...\ell]$, and sorted within $A[\ell...i+1]$, which after $i \leftarrow i+1$ matches our invariant. So all conditions of the invariant hold.

We argue that since our invariant holds, on termination the subarray $A[\ell...m-1]$ is sorted amongst itself, and larger than any element from $A[0...\ell-1]$. We also argue that termination will occur, since $i \leftarrow i+1$ at each iteration. We note that from the invariant, $0 \leq i \leq m$; so on termination it must be the case that $i = m$.

We have now shown that each iteration of the nested loop rearranges the elements of A in such a way that the range $A[i...\ell-1]$ is in sorted order. However, in order to make larger claims about the correctness of BubbleSort, it will be necessary to prove a stronger statement; namely that the overall contents of the array A remain the same through each iterations. One can imagine an algorithm that sets all array elements to 1; this algorithm would fulfill our Invariant A, but clearly not give the desired output. With BubbleSort, the main danger is that a swap may occur out of range of the array A. With this in mind, we introduce the following additional condition to our invariant:

Invariant A_2: The array A is a permutation of the original input array.

We see that P(0) holds trivially; A is the input array. Assuming P(k), we must show P(k+1). We have previously argued that $i < m-1$ at the outset of our loop. A swap only occurs after this point, and before i is incremented; so the range for i is $0 \leq i < m-1$ during any swap. Since a swap occurs between $A[i]$ and $A[i+1]$, we see that all the swapped elements are in the range of A, and further in $A[0...m-1]$. So our swapped array is a permutation of A_k, which from P(k) is a permutation of the original array.

From the invariant of the nested loop, we can derive an invariant for the main loop as well. It goes as follows:

Invariant B: $0 \leq m \leq n$, *all elements from* $A[m...(n-1)]$ *are in sorted order, and all of these elements are* $\geq$ *any element in* $A[0...m-1]$.

In the base case, $m = n$, and the array $A[n...n-1]$ is empty, so P(0) holds. Assuming P(k), we show that P(k+1) holds. We know that the elements $A[m_k...n-1]$ are in sorted order from P(k); we can see that these elements are untouched in the (k+1) iteration since, as stated earlier, swaps only occur on elements from $A[0...m-1]$. From Invariant A, we know that the elements $A[\ell...m_k-1]$ are sorted amongst themselves at the end of this iteration. They are also smaller than any element from $[m_k...n-1]$ (since these are sorted among all elements in A), and larger than any element from $[0...\ell-1]$. So the elements $A[\ell...m_k-1]$ are sorted correctly; and thus the elements $A[\ell...n-1]$ are sorted correctly.

Since $m \leftarrow \ell$ at the end of each iteration, and $0 \leq \ell \leq i < m$ from invariant A, the second invariant holds.

Part b

Partial correctness for the algorithm follows from Invariant B; if the main loop is exited, either m = 0 or m = 1 (since $0 \leq m \leq n$ after all iterations, and the while condition is $m > 1$). In the first case ($m = 0$), all elements from $A[0...(n-1)]$ are sorted directly from the invariant. If $m = 1$, we see that $A[1...(n-1)$ is in sorted order; in addition, each element in this range is $\geq A[0]$. So all elements from $A[0...(n-1)]$ are in sorted order, fulfilling partial correctness in either case.

In order to argue termination, we note that after the nested loop exits, $\ell < m$ from Invariant A; and since $m \leftarrow \ell$ at the end of each main loop iteration, m is strictly decreasing. As ℓ (and thus m)

decrease, they will reach 0 or 1. When this happens, the program will terminate. So BubbleSort terminates, and the program is correct.

Part c

We define P(A, n) as the number of executions of line (5) on an array A of size n. The important thing to notice is that, after one iteration of the main loop, all elements from $A[\ell...(n-1)]$ are sorted, and m is set to ℓ; so P(A, n) will equal the number of executions of line (5) in the main loop + the number of executions of line (5) on an array A' of size ℓ. Or, since line (5) will execute n times in the first loop regardless of A's internal structure, P(A, n) $= n +$ P(A', ℓ). We also note that P(A, 1) $= 0$ for any array A; the main loop does not trigger, and thus the nested loop is never checked. Since $\ell < m$ at each step, and m is intially $= n$, we see that P(A, n) can be no larger than the sum of the integers $n, n-1, \ldots, 2$, or $\frac{n(n+1)}{2} - 1$.

We next argue that this upper bound can actually be obtained. We note that for $l = n-1$ after the first iteration, each iteration of the nested loop must 'push' an element to the right; the simplest case in which this happens is when $A[0]$ is the largest element in the array. Attempting to preserve this condition for all ranges 0....i in our array, we get an array that is in reverse order. We now prove that on this array the algorithm does execute the test in line (5) $\frac{n(n+1)}{2} - 1$ times..

Statement: Given an array A of size n, $n \geq 1$, where $\forall i A[i] >$ all elements $A[i...n]$, the number of executions of line (5) for BubbleSort on this array is $\frac{n(n+1)}{2} - 1$.

P(1) holds, since above we stated that P(A, 1) $= 0$ for any array A. Assuming P(n) holds, we attempt to prove P(n + 1). Since $A[0]$ is the largest element of A, there will be n pushes and $n+1$ executions of the test in line (5) during the first iteration; after this iteration, no elements have changed relative order other than $A[0] \rightarrow A[n]$. So $m \leftarrow \ell = n$, and now BubbleSort iterates on a subarray of size n, where $\forall i A[i] >$ all elements $A[i...n]$. From our inductive hypothesis, this subarray gives the maximum possible number of executions of line (5), or $\frac{n(n+1)}{2} - 1$. So the number of executions of line (5) for the full array is $n + 1 + \frac{n(n+1)}{2} - 1$, or $\frac{(n+1)(n+2)}{2} - 1$. Thus $P(n+1)$ holds.

Problem 2

This problem deals with exponentiation. We restate the algorithm below for completeness. We also make use of the function `Square` which returns $x \times x$ on input x.

```
Function FastExp($a, b$)
    Input: $a, b$ - integers, $a \neq 0$, $b \geq 0$
    Output: $a^b$
(1) if $b = 0$ then return 1
(2) if $b$ is even then return Square(FastExp($a$,$b/2$))
(3)             else return $a \cdot$ FastExp($a, b-1$)
```

Part a

In order to prove correctness of `FastExp`, we argue partial correctness and termination.

For partial correctness, we assume that all recursive calls return the correct value provided the preconditions are met, and argue by cases.

Case 1: $b = 0$. In this case, `FastExp` returns on line 1. The returned value is 1, which is the correct value because raising any nonzero integer to the zeroth power gives 1.

Case 2: $b \neq 0$ is even. If this is the case, the algorithm returns from line 2. We can write $b = 2k$ for some integer k, so $a^b = a^{2k} = (a^k)^2 = a^k \cdot a^k$. Now $a^k \cdot a^k = \texttt{Square(}a^k\texttt{)}$. Furthermore, $k = b/2 \geq 0$ is a non-negative integer, so k satisfies the preconditions of `FastExp`. Since a also satisfies the preconditions, we can assume that the recursive call to `FastExp` on line 2 returns the correct value, i.e., $\texttt{FastExp(}a, b/2\texttt{)}$ returns $a^{b/2} = a^k$. Hence, the original call returns $\texttt{Square(FastExp(}a, b/2\texttt{))} = \texttt{Square(}a^k\texttt{)} = a^{2k} = a^b$, which is correct.

Case 3: $b \neq 0$ is odd. If this is the case, the algorithm returns from line 3. Since $b \neq 0$, $b-1 \geq 0$, so $b-1$ satisfies the preconditions of `FastExp`. Since a also satisfies the preconditions of `FastExp`, the recursive call to `FastExp` return the correct value, i.e., $\texttt{FastExp(}a, b-1\texttt{)}$ returns a^{b-1}. Hence, the original call returns $a \cdot \texttt{FastExp(}a, b-1\texttt{)} = a \cdot a^{b-1} = a^b$, which is correct.

This completes the proof of partial correctness.

For termination, we prove by strong induction on b that the call $\texttt{FastExp(}a, b\texttt{)}$ terminates.

For the base case, $b = 0$, the algorithm executes line 1 and returns right away.

Now assume that a call to `FastExp` with second argument at most b terminates. Consider a call to $\texttt{FastExp(}a, b+1\texttt{)}$. Then $b \neq 0$, and there are two cases to consider.

Case 1: $b+1$ is even. In this case, the algorithm makes a call to $\texttt{FastExp(}a, (b+1)/2\texttt{)}$ on line 2. Note that the second argument to this call is at most b (because $(b+1)/2$ is an integer less than $b+1$), so this call terminates. The original call terminates right after that.

Case 2: $b+1$ is odd. Then the algorithm makes a call to $\texttt{FastExp(}a, b\texttt{)}$ on line 3. Note that the second argument to this call is at most b, so this call terminates. The original call terminates right after that.

This completes the proof of termination. Since we showed partial correctness earlier, this implies that `FastExp` meets its specification.

Part b

Let $R(k)$ be the number of recursive calls made by `FastExp` when the second argument is $b = 2^k$, and let $M(k)$ be the number of multiplications on such input.

First consider $k = 0$. Then the second argument to `FastExp` is $b = 2^0 = 1$. In that case, `FastExp` returns from line 3. It makes a recursive call to `FastExp` with second argument zero. That call returns 1 right away, and there are no more recursive calls after that. Thus, $R(0) = 1$. Similarly, the recursive call to `FastExp` makes no multiplications when the second argument is zero, so the only multiplication is the multiplication by a that happens after the recursive call to `FastExp` returns. Hence, $M(0) = 1$.

Now consider a call to `FastExp` with $b = 2^k$ where $k \geq 1$. Then b is even, and `FastExp` reaches line 2. It makes the recursive call $\texttt{FastExp(}a, b/2\texttt{)}$, squares the result, and returns. Squaring costs one multiplication, and all other multiplications come from the recursive call, so we have $M(k) = 1 + M(k-1)$. Similarly, we have $R(k) = 1 + R(k-1)$.

We now have the following two recurrences.

$$R(k) = R(k-1) + 1, \quad R(0) = 1 \tag{1}$$

$$M(k) = M(k-1) + 1, \quad M(0) = 1 \tag{2}$$

Let's solve them. We claim that $R(k) = M(k) = k + 1$.

We have already proved the base case by direct computation. We showed $R(0) = 1 = 0 + 1$ and $M(0) = 1 = 0 + 1$.

Now assume that $R(k) = M(k) = k + 1$ and consider a call to `FastExp` with $b = 2^{k+1}$. Note that $b/2 = 2^{k+1}/2 = 2^k$, so the recursive call makes $R(k) = k + 1$ additional recursive calls and $M(k) = k + 1$ additional multiplications by the induction hypothesis. By our recurrences, $R(k + 1) = R(k) + 1 = k + 2$, and $M(k + 1) = M(k) + 1 = k + 2$. This completes the proof.

Part c

Instead of using `Square`, we now call `FastExp` twice on line 2, and multiply the returned values. As we shall see, this causes a drastic slowdown in the algorithm's performance.

We obtain recurrences for the numbers of recursive calls and multiplications on input 2^k. We still have $R(0) = M(0) = 1$. When the second input is $b = 2^k$ with $k > 0$, `FastExp` goes to line 2, and makes 2 calls to `FastExp` with second argument 2^{k-1} and one multiplication. Thus, we get $R(k) = 2 + 2R(k - 1)$ and $M(k) = 1 + 2M(k - 1)$.

We now solve the recurrence for $R(k)$. The first few terms are listed in Table 1.

k	0	1	2	3	4	5	6	7
$R(k)$	1	4	10	22	46	94	190	382
2^k	1	2	4	8	16	32	64	128
$3 \cdot 2^k$	3	6	12	24	48	96	192	384

Table 1: A visual aid for solving the recurrence $R(k)$.

We see from Table 1 as well as from the recurrence that the value of $R(k)$ is roughly twice the value of $R(k - 1)$, so k should appear as an exponent of 2 somewhere in the solution. But as we can see from Table 1, 2^k is too small, and trying some other functions of k, such as $k + 1$ or $2k$ in the exponent doesn't quite work either. But notice that as k gets higher, we have $R(k)/2^k \approx 3$. As we see from Table 1, multiplying 2^k by 3 looks promising. In fact, adding 2 to the entries in the row corresponding to $R(k)$ gives us the row that corresponds to $3 \cdot 2^k$, so we conjecture that $R(k) = 3 \cdot 2^k - 2$. Let's prove this conjecture by induction.

Notice that $3 \cdot 2^0 - 2 = 3 - 2 = 1 = R(0)$, so the base case holds.

Now assume that $R(k) = 3 \cdot 2^k - 2$. Consider a call to `FastExp` with 2^{k+1} as the second argument. The number of recursive calls is $R(k + 1) = 2 + 2R(k) = 2 + 2 \cdot (3 \cdot 2^k - 2) = 2 + 2 \cdot 3 \cdot 2^{k+1} - 2 \cdot 2 = 3 \cdot 2^{k+1} - 2$. This proves the inductive step and solves our recurrence.

The recurrence for $M(k)$ is the same one we saw in lecture for the towers of Hanoi problem, except with the first term being 1 instead of 0. The recurrence was $N(k) = 2 \cdot N(k - 1) + 1$ with $N(0) = 1$ was $N(k) = 2^k - 1$, and its solution was $N(k) = 2^k - 1$. Now we have $M(0) = 1 = N(1)$. Thus, the sequence $M(k)$ is the same as the sequence $N(k)$ shifted by one position, that is, $M(k) = N(k + 1) = 2^{k+1} - 1$.

We see that the number of recursive calls increases exponentially in comparison to part (b) of this problem, and so does the number of multiplications.

Problem 3

Part a

We see that for any c such that $F_n = c^n$ satisfies the recurrence condition, we have $c^3 = c^2 + c$ from applying the reccurrence at $n = 3$. This gives the equation $c^3 - c^2 - c = 0$, or $c(c^2 - c - 1) = 0$. We will attempt to derive all possible values of c by first determining which values of c satisfy this equation.

We see immediately that $c = 0$ is a possible value satisfying $c(c^2 - c - 1) = 0$. Applying the quadratic formula to $c^2 - c - 1 = 0$ [Recall: for $ax^2 + bx + c = 0$, $x = \frac{-b \pm \sqrt{(b^2-4ac)}}{2a}$], we see that $c = \frac{1 \pm \sqrt{5}}{2}$.

We note that we can show that each of these c satisfy the reccurence for all F_k simply by multiplying each side of the equation $c^3 = c^2 + c$ by c the necessary number of times. So, for example, to show that $c = \frac{1+\sqrt{5}}{2}$ satisfies the recurrence for F_5, we take our equation $(\frac{1+\sqrt{5}}{2})^3 = (\frac{1+\sqrt{5}}{2})^2 + \frac{1+\sqrt{5}}{2}$, which was proven above, and multiply each side by $(\frac{1+\sqrt{5}}{2})^2$ to get $(\frac{1+\sqrt{5}}{2})^5 = (\frac{1+\sqrt{5}}{2})^4 + (\frac{1+\sqrt{5}}{2})^2$.

Part b

We assume that A_n and B_n satisfy the recurrence condition; that is, $\forall n \geq 3, A_n = A_{n-1} + A_{n-2}$ and $B_n = B_{n-1} + Bn - 2$. In order to prove that $X_n = \alpha A_n + \beta B_n$, we will use the defintions given by these recurrence conditions as part of a direct proof.

We first prove that for any real α, $F_n = \alpha A_n$ satisfies the recurrence condition. Note that from the recurrence of our original assumption $A_k = A_{k-1} + A_{k-2}$ for any $k \geq 3$. Multiplying each side by α, we have $\alpha A_k = \alpha A_{k-1} + \alpha A_{k-2} \forall k \geq 3$. So our proof is complete.

We now prove that for any C_n and D_n satisfying the recurrence condition, $F_n = C_n + D_n$ also satisfies the condition. We note that from the recurrences of our original assumption $C_k = C_{k-1} + C_{k-2}$ and $D_k = D_{k-1} + D_{k-2} \forall k \geq 3$. So $C_k + D_k = C_{k-1} + D_{k-1} + C_{k-2} + D_{k-2} \forall k \geq 3$, and our proof is complete.

Substituting $C_n = \alpha A_n$ and $D_n = \beta B_n$ gives us the desired statement, which from our above arguments satifies the recurrence condition if A_n and B_n do individually.

Part c

In order to show that the Fibonacci sequence satisfies (1), it is necessary for (1) to meet both the intial and recurrence conditions of the sequence. From part (a) we have two nonzero reals c where $X_n = c^n$ satisfies the reccurence conditions; however, neither of these satisfy the initial conditions. However, from part (b) we have shown that a linear combination of these two reals will also satisfy the reccurence conditions. Now we are essentially solving for two variables, α and β, where $\alpha(\frac{1+\sqrt{5}}{2}) + \beta(\frac{1-\sqrt{5}}{2}) = 1$ and $\alpha(\frac{1+\sqrt{5}}{2})^2 + \beta(\frac{1-\sqrt{5}}{2})^2 = 1$.

Our stated closed form gives us a hint as to how to set these variables; let $A_n = (\frac{1+\sqrt{5}}{2})^n$ and $B_n = (\frac{1-\sqrt{5}}{2})^n$. Note that $\frac{1-\sqrt{5}}{2}$ can be rewritten as $1 - \frac{1+\sqrt{5}}{2}$. Now, we define $F_n = \alpha A_n + \beta B_n$, where $\alpha = \frac{1}{\sqrt{5}}$ and $\beta = -\frac{1}{\sqrt{5}}$. Now we have our equation for the closed form for the Fibonacci sequence, which satisfies the recurrence conditions.

To verify that this is a valid closed form for the Fibonacci sequence, we must show that this closed form satisfies the initial conditions of the Fibonacci sequence as well. $F_1 = \frac{1+\sqrt{5}-(1-\sqrt{5})}{2} * \frac{1}{\sqrt{5}} = \frac{2\sqrt{5}}{2} * \frac{1}{\sqrt{5}} = 1$, which checks out. $F_2 = \frac{1+2\sqrt{5}+5-(1-2\sqrt{5}+5)}{4} * \frac{1}{\sqrt{5}} = \frac{4\sqrt{5}}{4} * \frac{1}{\sqrt{5}}$; so this checks out too. So our closed form matches both the initialization and recurrence conditions of the Fibonacci sequence; therefore the Fibonacci sequence satisfies our closed form.

Problem 4

There exist several methods for solving recurrences. We provide two different solutions for solving the given recurrence.

First Method: In this approach, we compute the first few terms of the sequence. From these terms, we try to guess the form of the solution. Then, we use induction to check whether or not our guess is the correct solution for the given recurrence.

We have $A_0 = 3$, $A_1 = 7$, $A_2 = 3A_1 - 2A_0 = 21 - 6 = 15$, $A_3 = 3 * 15 - 2 * 7 = 45 - 14 = 31$. Similarly, we can show that $A_4 = 63$, $A_5 = 127$. As one can observe, it seems that $A_n \sim 2A_{n-1}$. If we look more closely, we might conjecture $A_n = 2A_{n-1} + 1$. In fact, an even finer look would yield that A_n only takes the values one less than a power of 2. This can be refined to get the solution as $A_n = 2^{n+2} - 1$.

Now, let us do an inductive proof that our guess for the solution is indeed correct. We would like to prove that $P(n) : A_n = 2^{n+2} - 1$ holds $\forall n \geq 0$. We need to consider two base cases – $n = 0$ and $n = 1$. These hold as we have $A_0 = 3 = 4 - 1 = 2^{0+2} - 1$ and $A_1 = 7 = 8 - 1 = 2^{1+2} - 1$.

Now, suppose $P(n-1)$ and $P(n-2)$ hold true for $n \geq 2$. We would like to prove that P(n+1) holds, i.e. $A_n = 2^{n+2} - 1$. Now, from the recurrence we have that $A_n = 3A_{n-1} - 2A_{n-2}$. Applying the induction hypothesis, we get

$$\begin{aligned} A_n &= 3A_{n-1} - 2A_{n-2} = 3(2^{n-1+2} - 1) - 2(2^{n-2+2} - 1) \\ &= 3(2^{n+1} - 1) - 2(2^n - 1) = 3.2^{n+1} - 3 - 2.2^n + 2 \\ &= 2^n(3.2 - 2) - 1 = 2^n(4) - 1 = 2^{n+2} - 1 \end{aligned}$$

This completes the induction step of the proof. Hence, we have shown that $P(n)$ holds for all $n \geq 0$.

Second Method: In this solution, we follow the approach outlined in problem 3. So, let's assume $\exists\ c \in \mathbb{R}^+$ such that $A_n = c^n$. Then, we can use the recurrence relation to get the equation $c^n = 3c^{n-1} - 2c^{n-2}$. Dividing both sides by c^{n-2}, we get $c^2 - 3c + 2 = 0$. This equation has the solutions: $c = 1, 2$.

Now, assume that $A_n = X_n$ and $A_n = Y_n$ are solutions to the given recurrence relation; that is, $\forall n \geq 2, X_n = 3X_{n-1} - 2X_{n-2}$ and $Y_n = 3Y_{n-1} - 2Y_{n-2}$. We will now prove that for $\forall \alpha, \beta \in \mathbb{R}$, $A_n = \alpha X_n + \beta Y_n$ is also a solution to the recurrence relation. Substituting the values of A_{n-1} and A_{n-2} in to the RHS of the recurrence relation, we get that –

$$\begin{aligned} 3A_{n-1} - 2A_{n-2} &= 3(\alpha X_{n-1} + \beta Y_{n-1}) - 2(\alpha X_{n-2} + \beta Y_{n-2}) \\ &= 3\alpha X_{n-1} - 2\alpha X_{n-2} + 3\beta Y_{n-1} - 2\beta Y_{n-2} \\ &= \alpha X_n + \beta Y_n = A_n \end{aligned}$$

Hence, $\alpha X_n + \beta Y_n$ is also a solution to the recurrence.

Now, we will use the values obtained for c and the values A_0 and A_1, to derive a final expression for A_n. Let $X_n = 1^n$, $Y_n = 2^n$, $A_n = \alpha X_n + \beta Y_n$. We need to find the values of α and β such that $A_0 = 3$ and $A_1 = 7$. To figure out the values, we put $n = 0, 1$ in the expression for A_n. We get the following equations –

$$\begin{aligned} \alpha + \beta &= 3 \\ \alpha + 2\beta &= 7 \end{aligned}$$

Solving these equations, we get $\alpha = -1$, $\beta = 4$.

Any solution to a recurrence should satisfy both the recurrence relation and the initial conditions. We had earlier shown that $\alpha X_n + \beta Y_n$ is a solution for the given recurrence when ever X_n and Y_n are. We also derived the values X_n and Y_n as solutions to the recurrence. Now, we have used the initial conditions to arrive the values of α and β. Combining these, we can arrive at the final expression: $A_n = 2^{n+2} - 1$.