Fibonacci, Integer Multiplication

1. Yes, we can. First we find a_1b_1 and a_0b_0. After that, we compute the product $(a_1 + a_0)(b_1 + b_0)$. If we expand this, we get $(a_1b_1) + (a_0b_0) + (a_1b_0 + b_1a_0)$. Since we know the value of the first 2 terms already, we can just subtract their values from this 3^{rd} product to get $a_1b_0 + b_1a_0$. One small problem is that $a_1 + a_0$ and $b_1 + b_0$ might be $n/2 + 1$ bit integers, due to overflow, so instead of this we can multiply $a_1 - a_0$ and $b_1 - b_0$, and by doing the algebra, we can again get $a_1b_0 + b_1a_0$.

2. Doing this question with 6 squarings is easy, but doing it with 5 needs a bit more care. We have 5 terms, and 5 squarings are allowed, so we can imagine maybe having the additions and subtractions to solve 5 linear equations in 5 variables. One useful observation we can make is that the 5 terms we need are just the coefficients of $(Px^2 + Qx + R)^2$ (if we just take $2^{n/3}$ as x). And to get the coefficients of a d degree polynomial, we just need to evaluate it at $d + 1$ different points. And since the degree is 4, this is perfect for us. Also to evaluate it at each point, we need exactly 1 squaring. For example, at $x = 2$ the polynomial equals $(4P + 2Q + R)^2$. So let's evaluate this polynomial at $-2, -1, 0, 1, 2$.

$$\begin{bmatrix} 16 & -8 & 4 & -2 & 1 \\ 1 & -1 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 16 & 8 & 4 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} P^2 \\ 2PQ \\ Q^2 + 2PR \\ 2QR \\ R^2 \end{bmatrix} = \begin{bmatrix} (4P - 2Q + R)^2 \\ (P - Q + R)^2 \\ R^2 \\ (P + Q + R)^2 \\ (4P + 2Q + R)^2 \end{bmatrix}$$

The vector on the right hand side represents the 5 squarings we have to do. Now to solve x in $Ax = b$, since A is some fixed matrix just filled with constants, we could precompute A^{-1}, or we could do row operations if we want to just manipulate the linear equations we got. Either ways, we can solve for the coefficients i.e the vector x with few additions/subtractions.

3. If we are allowed 6 multiplications, we can come up with it manually, we can do something similar to Karatsuba. First we just calculate a_0b_0, a_1b_1, and a_2b_2. To compute $a_1b_0 + a_0b_1$, and $a_2b_1 + a_1b_2$, we do Karatsuba's trick and compute $(a_1 + a_0)(b_1 + b_0)$ and $(a_2 + a_1)(b_2 + b_1)$. For the middle term, since we already have a_1b_1, we just need $a_2b_0 + a_0b_2$, so we just compute $(a_2 + a_0)(b_2 + b_0)$, and we're done.

For 5 multiplications, we do what we did in the last question. We just need to find the coefficients of $(a_2x^2 + a_1x + a_0)(b_2x^2 + b_1x + b_0)$. For this we evaluate the polynomial at 5 different points and for each evaluation, we are multiplying 2 $n/3$ bit numbers.

4. Let's just generalize this, and solve $T(n) = aT(n/b) + O(n)$. To make things easier, assume $n = b^k$. Let's try to reduce the RHS to just $T(1)$ by repeatedly applying the recursion.

$$\begin{aligned}
T(b^k) &= aT(b^{k-1}) + O(b^k) \\
&= a(aT(b^{k-2}) + O(b^{k-1})) + O(b^k) \\
&= a^2T(b^{k-2}) + O(b^k + ab^{k-1}) \\
&= a^3T(b^{k-3}) + O(b^k + ab^{k-1} + a^2b^{k-2}) \\
&\vdots \\
&= a^kT(1) + O(b^k + ab^{k-1} + \cdots + a^{k-1}b) = a^kT(1) + O\left(b\left(\frac{a^k - b^k}{a-b}\right)\right)
\end{aligned}$$

Since in all the subdivisions $a > b$, we can assume the second term is just $O(a^k)$, so overall $T(b^k) = O(a^k)$. Substituting $k = \log_b n$, $T(n) = a^{\log_b n} = n^{\log_b a}$. To compare the complexities, we just need to compare log values, and it's easy to check that $\log_4 7 < \log_5 3 < \log_4 8 < \log_6 3$.

5. We can do something similar to Q2. We'll have to square n/k bit integers $2k-1$ times, so this takes $(2k-1)T(n/k)$ time. But the the other step we have to do is now solve $2k-1$ linear equations in $2k-1$ variables. If we do the normal row operations, i.e Gaussian elimination, this takes $O((2k-1)^3(n/k)) = O(k^2n)$. This is because each row might have to be subtracted from every row below it during elimination, and every row subtraction involves $2k-1$ individual subtractions. The n/k factor comes because the addition/subtraction is happening with n/k bit integers. So our recursion is $T(n) = (2k-1)T(n/k) + O(k^2)$. Now we can do something similar to the previous question, and keep using the recursion. Assume $n = k^c$.

$$\begin{aligned}
T(k^c) &= (2k-1)T(k^{c-1}) + O(k^2(k^c)) \\
&= (2k-1)((2k-1)T(k^{c-2}) + O(k^2(k^{c-1}))) + O(k^{c+2}) \\
&= (2k-1)^2T(k^{c-2}) + O(k^{c+2} + (2k-1)k^{c+1}) \\
&= (2k-1)^3T(k^{c-3}) + O(k^{c+2} + (2k-1)k^{c+1} + (2k-1)^2k^c) \\
&\vdots \\
&= (2k-1)^cT(1) + O(k^{c+2} + (2k-1)k^{c+1} + \cdots + (2k-1)^{c-1}k^3) \\
&= (2k-1)^cT(1) + O\left(k^3\left(\frac{(2k-1)^c - k^c}{(2k-1)-k}\right)\right) \\
&= (2k-1)^cT(1) + O(k^2((2k-1)^c - k^c))
\end{aligned}$$

Since $2k-1 > k$, the second term is $O(k^2(2k-1)^c)$, and the first term is clearly $O((2k-1)^c)$, so overall, $T(k^c) = O(k^2((2k-1)^c))$. Substituting $c = \log_k n$, we get $T(n) = O(k^2((2k-1)^{\log_k n})) = O(k^2n^{\log_k(2k-1)})$.

If we take $k = n/2$, $T(n) = O((n/2)^2n^{\log_{n/2}(n-1)})$, this is asymptotically as bad as (technically worse than) $O(n^2)$.

6. We have to show that $2^{\sqrt{\log n}}$ is better than $n^{0.01}$. Taking log on both sides, it suffices to show that $\sqrt{\log n}$ is better than $0.01\log n$. Just choose some $N > 2^{10,000}$. For

$n > N$, $\log n > 10,000$, so $\sqrt{\log n} > 100$, which would mean $0.01 \log n > \sqrt{\log n}$. This trick would work to show that $n2^{\sqrt{\log n}} > n^{1+\epsilon}$ for any positive ϵ, by just choosing $N > 2^{\frac{1}{\epsilon^2}}$.

7. Idk lol just code something

8. **(a)** It's just some algebra, but at the end we get the 4 sums as $a_1b_1 + a_2b_3, a_1b_2 + a_2b_4, a_3b_1 + a_4b_3, a_3b_2 + a_4b_4$.

(b) Because of how matrix multiplication works, treating the submatrices as numbers and multiplying like 2x2 matrices actually gives us the multiplication of the matrices if done normally.

$$C = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$$

(c) The divide and conquer is basically done from the previous parts. First we just divide A, B each into 4 submatrices, and we have 4 different terms to compute. But in order to compute these terms, we can compute the terms p_1 to p_7 but substituting submatrices instead of numbers. Note that each p_i will take exactly 1 matrix multiplication of size $n/2 \times n/2$. Now that the p_i's are calculated, we can do few additions and subtraction as we did in the first part to get the 4 terms that we need, they are the same as the terms that we need in the second part. So this is how we get C. The recursion for this is that $T(n) = 7T(n/8) + O(n^2)$ ($O(n^2)$ for additions and subtractions). Solving this recursion gives $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

Polynomial mulitplication, Fast Fourier transform

9. Yes we can, let's say we want to multiply $p(x)$ and $q(x)$. Split them as $p(x) = p_0(x) + x^{d/2}p_1(x)$ and $q(x) = q_0(x) + x^{d/2}q_1(x)$ So $p(x)q(x) = p_0(x)q_0(x) + x^{d/2} \times (p_0(x)q_1(x) + q_0(x)p_1(x)) + x^d \times p_1(x)q_1(x)$. We can do the same trick as Karatsuba to get these 3 terms in just 3 polynomial multiplications of size $d/2$. The recursion is $T(d) = 3T(d/2) + O(d)$, $T(d) = O(d^{1.58})$.

10. The probability mass function of the sum is just the convolution of the pmfs of the 2 random variables (assuming they are independent). Let's just explain this with an example. Let's say we have 2 weighted dice, A and B with some probabilities for the numbers 1 to 6. If we want to find $P(A + B) = 4$, we can case split this as $P(A + B = 4, B = 1) + P(A + B = 4, B = 2) + P(A + B = 4, B = 3)$. This is same as $P(A = 3, B = 1) + P(A = 2, B = 2) + P(A = 1, B = 3)$. Assuming indpendence of A, B, this is $P(A = 3)P(B = 1) + P(A = 2)P(B = 2) + P(A = 1)P(B = 3)$. This is a term of the convolution of the pmfs, and similarly we can show this for all terms.

11. Let's just take an example, say our array to FFT is $[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$ where a_i is the coefficient of x^i in our polynomial. We want our output to be the same array, with evaluation of this polynomial at the 8th roots of unity. The main idea of FFT is to separate the even coefficients and the odd coefficients, form polynomials of those, and solve the same problem for them. Using the solution of this subproblem, it's possible to combine its solutions to get the answer for our full problem. So a_0, a_2, a_4, a_6 is 1 group, a_1, a_3, a_5, a_7 is another group. Now we again break each group into even and odd indices. The first group is broken into a_0, a_4 and a_2, a_6. And the second group is broken into a_1, a_5 and a_3, a_7. We'll think about how to reorder this array in place later, for now let's assume that the array has been reordered as $[a_0, a_4, a_2, a_6, a_1, a_5, a_3, a_7]$.

First we have to combine the paired terms (the adjacent terms of the array). Since 2 coefficients just form a linear polynomial we evaluate it at 1, -1. So our transformed array is

$$[a_0 + a_4, a_0 - a_4, a_2 + a_6, a_2 - a_6, a_1 + a_5, a_1 - a_5, a_2 + a_6, a_2 - a_6]$$

Our next step is going to be to separately combine terms in the left half, and in the right half. Let's just look at the left half first. The 0th and 2nd index can be combined, $(a_0 + a_4) + 1(a_2 + a_6)$, $(a_0 + a_4) - 1(a_2 + a_6)$. We now replace the values at these 2 indices with the combined values. Similarly the terms and the first and third index can be combined. Call the terms t_1 and t_3. $t_1 + i * t_3$ and $t_1 - i * t_3$ give polynomial evaluations at $i, -i$ respectively. The same 4 combinations can be done in the right half of the array, to get polynomial evaluations at $1, i, -1, -i$ for the polynomial $a_1 + a_3x + a_5x^2 + a_7x^3$.

Now for the final step, combine the left half and the right half of the array. Let the latest terms in the array be $t_0 tot_7$. $t_0 + t_4$ and $t_0 - t_4$ evaluate the polynomial at $1, -1$ so we can replace them with this combination. $t_1 + \omega t_5$ and $t_1 - \omega t_5$ evaluate the polynomial at ω and $-\omega = \omega^5$ respectively. Following this pattern, $t_2 \pm \omega^2 t_6$ and $t_3 \pm \omega^3 t_7$ will evaluate the polynomial at the rest of the points.

How do we generalize this idea for $n = 2^k$? We first reorder the array so that elements are paired well. Then we do a merge sort like combining of the array. First we split the array into chunks of size 2 and combine them (if the elements in the block are a, b, we replace them with $a + b$ and $a - b$). Then we break the full array into chunks of 4 and combine them. If a chunk has terms t_0, t_1, t_2, t_3, we replace them by $t_0 + t_2, t_1 + it_3, t_0 - t_2, t_1 - it_3$. Then we do chunks of size 8. For each chunk of size 8, we combine t_0 and t_4 with $t_0 \pm t_4$, and similarly $t_1 \pm 1^{1/4}t_5, t_2 \pm 1^{2/4}t_6, t_3 \pm 1^{3/4}t_7$. In general, if we want to combine things in a chunk of size $2s$, we combine like this:

$$[t_0 + t_s, t_1 + 1^{1/s}t_{s+1}, t_2 + 1^{2/s}t_{s+2}, \ldots, t_0 - t_s, t_1 - 1^{1/s}t_{s+1}, t_2 - 1^{2/s}t_{s+2}, \ldots]$$

This is an iterative algorithm, which only uses constant space.

Now we come back to our question, how do we reorder these elements? Let's think about how exactly they are reordered. First we split them based on if they are even or not, moving all even elements to the front. This is same as partitioning based on the last digit in binary. Then in each subarray we reorder again based on even index or not. We are basically ignoring the last digit now, and partitioning based on second last digit in binary. This process goes on, we then do the third, fourth, last digit and so on. At the end of this process, if you read the indices in reverse when seen in binary, they will be in ascending order (we are doing something similar to a radix sort). So this gives a simple method to find where a_i goes: write i in binary, reverse it, and that's where it goes. Something neat is that if a_i goes to j, then a_j

goes to i (this is cause if i in reverse binary is j, j in reverse binary is i). So we can just swap a_i and a_j directly. So we just iterate through all elements and do the swap (to make sure something isn't swapped twice, we can make sure we only swap with something after it).