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 $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.