Problemset 1

Albert Ye

February 6, 2023

1 Study Group

Zipeng Lin, 3036516523

2 Decimal to Binary

Let's assume we have an n-digit number x. Then, similar to in Karatsuba's Algorithm, let's cut the number into two equally-sized smaller numbers x_1 and x_2. We'd then have $x = 10^{\frac{n}{2}} x_1 + x_2$.

Our first step is trying to precompute the exponents of 10. Note that we can recursively find 10^k as $10^{\frac{k}{2}} \cdot 10^{\frac{k}{2}}$, so we can find $10^{\frac{k}{2}}$ in binary first and multiply the result by itself to find 10^k. Because the time complexity of Karatsuba is $O(k^{\log_2 3})$ to multiply two k-digit numbers and there are $\log n$ multiplication operations to complete, we have an $O(n^{\log_2 3} \log n)$ algorithm for this step.

Next, we look at the rest of the equation. We can use the same divide and conquer equation for x_1 and x_2 to find their binary representations, and the merging algorithm would take $O(n^{\log_2 3})$ time for this layer. However, we claim that each layer has an $O(n^{\log_2 3})$ runtime.

We claim that if we merge two $\frac{k}{2}$ digit numbers in one half (x_{11}, x_{12}) and two $\frac{k}{2}$ digit numbers in the other half (x_{21}, x_{22}), it would take the same time as merging their results x_1, x_2. Each half has a size of $\frac{k}{2}$, and merging each would take $(\frac{k}{2})^{\log_2 3}$ operations. Adding both of them together would give $\left(\frac{k}{2}\right)^{\log_2 3} + \left(\frac{k}{2}\right)^{\log_2 3} \leq k^{\log_2 3}$ operations. Merging two k-digit numbers x_1 and x_2 together would also lead to $k^{\log_2 3}$, so the runtimes are roughly equal.

Therefore, the algorithm is as follows. First, recursively precompute the exponents of 10 in binary. Then, for the larger number x wth n digits, split the number into two $\frac{n}{2}$-digit numbers x_1, x_2, where $x = 10^n x_1 + x_2$. After finding the binary representations of x_1 and x_2 using the same procedure as is currently being described, we plug x_1 and x_2 into the aforementioned equation for x, and use Karatsuba multiplication to find x. Therefore, we have an $O(n^{\log_2 3} \log n)$ algorithm.

3 Modular Fourier Transform

a) This is essentially taking everything to the power of 4 and checking if the result has residue 1 modulo 5. We have that $1^4 = 1 \pmod 5$, $2^4 = 16 \equiv 1 \pmod 5$, $3^4 = 81 \equiv 1 \pmod 5$, and $4^4 = 256 \equiv 1 \pmod 5$.

   Moreover, $2 \cdot 2 \equiv 4$, $4 \cdot 4 \equiv 1$, and $3 \cdot 3 \equiv 1 \pmod 5$, so 2 replaces i, 3 replaces $-i$, and 4 replaces -1.

   We have that $1 + 2 + 2^2 + 2^3 = 1 + 2 + 4 + 8 = 15 \equiv 0 \pmod 5$, so $1 + \omega + \omega^2 + \omega^3 \equiv 0 \pmod 5$.

b) We initially start with the $FFT([0, 2, 3, 0], [1, 2, 4, 3])$, and then try to reduce from there.

   As per our FFT algorithm, we can split our list A into even and odd indices (lists A_e and A_o, respectively), take the even powers from our roots of unity list, and run two more smaller FFT's on the even and odd indices. We would then have to combine them with the formula $A(x) = A_e(x^2) + xA_o(x^2)$. Our FFTs would be $FFT([0, 3], [1, 4])$ and $FFT([2, 0], [1, 4])$.

   $FFT([0, 3], [1, 4])$ once again gets split into $FFT([0], [1])$ which returns the value of 0, and $FFT([3], [1])$ which returns the value of 3. Then, we can combine the two with our algorithm $A(x) = A_e(x^2) + xA_o(x^2)$ to get return values of $[(1, 0 + 3), (4, 0 + 4 \cdot 3)] = [(1, 3), (4, 2)]$.

   Similarly, $FFT([2, 0], [1, 4])$ splits into $FFT([2], [1])$ which returns 2 and $FFT([0], [1])$ which returns 0. Combining the two as in the previous paragraph, we get $[(1, 2 + 0), (4, 2 + 4 * 0)] = [(1, 2), (4, 2)]$.

   Plugging these back into our formula for the larger set $FFT([0, 2, 3, 0], [1, 2, 4, 3])$ gives us:

$$\begin{aligned}
A(1) &= A_e(1) + 1 \cdot A_o(1) \\
&= 3 + 1 \cdot 2 = \boxed{0}. \\
A(2) &= A_e(2^2) + 2 \cdot A_o(2^2) \\
&= A_e(4) + 2 \cdot A_o(4) \\
&= 2 + 2 \cdot 2 = \boxed{1}. \\
A(3) &= A_e(3^2) + 2 \cdot A_o(3^2) \\
&= A_e(4) + 3 \cdot A_o(4) \\
&= 2 + 3 \cdot 2 = \boxed{3}. \\
A(4) &= A_e(4^2) + 2 \cdot A_o(3^2) \\
&= A_e(1) + 4 \cdot A_o(1) \\
&= 3 + 4 \cdot 2 = \boxed{1}.
\end{aligned}$$

   Therefore, our sequence $[0, 2, 3, 0]$ becomes $[0, 1, 3, 1]$ after FFT. This can be sanity checked by directly plugging $1, 2, 3, 4$ into the polynomial $A(x) = 2x + 3x^2$.

c) We essentially do the same thing again, except using ω^{-1} instead of ω. This means we take $FFT([0, 1, 3, 1], [1, 3, 4, 2])$.

   We once again split this into odd and even groups. Our splits are $FFT([0, 3], [1, 4])$ and $FFT([1, 1], [1, 4])$. The first split is identical to part (b), giving us $[(1, 3), (4, 2)]$. The second, however, splits into $FFT([1], [1])$ twice. Each time, it returns 1. Then, this split must return $[(1, 2), (4, 0)]$.

Plugging this back into $FFT([0,1,3,1],[1,3,4,2])$, we have:

$$\begin{aligned}
A(1) &= A_e(1) + 1 \cdot A_o(1) \\
&= 3 + 1 \cdot 2 = \boxed{0}. \\
A(2) &= A_e(2^2) + 2 \cdot A_o(2^2) \\
&= A_e(4) + 2 \cdot A_o(4) \\
&= 2 + 2 \cdot 0 = \boxed{3}. \\
A(3) &= A_e(3^2) + 3 \cdot A_o(3^2) \\
&= A_e(4) + 3 \cdot A_o(4) \\
&= 2 + 3 \cdot 0 = \boxed{3}. \\
A(4) &= A_e(4^2) + 4 \cdot A_o(3^2) \\
&= A_e(1) + 4 \cdot A_o(1) \\
&= 3 + 4 \cdot 2 = \boxed{1}.
\end{aligned}$$

4 Triple Sum

We encode the list into a polynomial $P(x) = x^{a_1} + x^{a_2} + x^{a_3} + \cdots + x^{a_n}$. If we multiply $P(x)$ by itself three times, we end up with a polynomial $P(x)^3 = (x^{a_1}+x^{a_2}+\cdots+x^{a_n})(x^{a_1}+x^{a_2}+\cdots+x^{a_n})(x^{a_1}+x^{a_2}+\cdots+x^{a_n})$. Note that for any three coefficients a_i, a_j, a_k, we can find a term of $x^{a_i}x^{a_j}x^{a_k} = x^{a_i+a_j+a_k}$ in the resulting polynomial. This is done by taking the power of a_i in the first polynomial, a_j in the second polynomial, and a_k in the third polynomial.

We know how to multiply two polynomials of degree n quickly using Fast Fourier Transform. Then, we will have a polynomial of degree $2n$ and another one of degree n. We can still apply FFT on these two polynomials, just setting the powers of $n+1$ to $2n$ in the second polynomial to 0. Given the final polynomial in coefficient form, we can easily find if n is a valid sum by checking if the coefficient of x^n in the resulting array is equal to 0 or not. This works by the way $P(x)^3$ is constructed, because the powers of some term in $P(x)^3$ must be the sum of three powers in $P(x)$.

We know that $a_i \leq n \forall i$, so the FFT operations will thus take $O(n \log n)$.

5 Pattern Matching

a) For each position i, we search the substring $s[i : i+n]$. If at least k characters match, we add the index to our answer, otherwise we do not and just search at the next i. We have to go through $m-n$ total positions, and we check n integers each time, so our overall answer is $O(nm)$.

b) First, we reverse the pattern string s. Then, We convert the strings into polynomials, where each power has a coefficient of $+x^i$ if the ith character in the string is 1, and $-x^i$ if the ith character in the string is 0. For example, for the pattern string 0111 we would end up with the polynomial $S(x) = 1+x+x^2-x^3$, but the same sequence string 0111 would make the polynomial $G(x) = -1+x+x^2+x^3$. Our algorithm is to use FFT to multiply $G(x)$ and $S(x)$, and then loop over all the terms $a_{n+i-1}x^{n+i-1}$ to $a_m x^m$. For each term $a_j x^j$, we check if $a_j \geq n-2k$, and if so, we add it to our answer.

Lemma 5.1. If we multiply the polynomials we get for g and s together ($GS(x)$), the coefficient of x^{n+i-1} alone can determine whether the position i is a match.

Proof. Let's just use two arbitrary degree-3 polynomials as an example, where we have $G(x)S(x) = (a_0 1 + a_1 x + a_2 x^2 + a_3 x^3)(b_0 1 + b_1 x + b_2 x^2 + b_3 x^3)$. Notice that the x^3 term in $G(x)S(x)$ is calculated by adding $a_0 x^0 \cdot b_3 x^3 + a_1 x^1 \cdot b_2 x^2 + a_2 x^2 \cdot b_1 x^2 + a_3 x^3 \cdot b_0 x^0$. Therefore, the coefficient of the b^3 term is equal to $a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0$.

By definition, each of the a_i and b_i can only take on two values: 1 or -1. Note that for each term $a_i b_j$, if $a_i = b_{3-i}$, then the result is 1 and otherwise the result is -1. Because $S(x)$ is reversed, we know that the x^3 coefficient of $S(x)$ corresponds to the character at position 0, and the x^0 coefficient of $S(x)$ corresponds to the character at position 3. Therefore, $a_i b_{3-i} = 1$ if the ith characters of g and s match.

We can generalize to two arbitrarily degree $G(x)S(x) = (a_0 1 + a_1 x + \cdots + a_{m-1}x^{m-1})(b_0 1 + b_1 x + \cdots + b_{n-1}x^{n-1})$. Then,

$$x^{n+i-1} = a_i x^i b_{n-1} x^{n-1} + a_{i+1} x^{i+1} b_{n-2} x^{n-2} + \cdots + a_{i+n-2} x^{i+n-2} b_1 x^1 + a_{i+n-1} x^{i+n-1} b_0 x^0.$$

In the jth term ($j \in [0, n)$), we are comparing $a_{i+j} x^{i+j} b_{n-j-1} x^{n-j-1}$, which is essentially taking g_{i+j} and comparing it with s_j. ■

Remember that if $g_{i+j} = s_j$, then we add 1 to the x^{n+i-1} term, and otherwise we subtract 1 from the x^{n+i-1} term. If all n numbers match, then we will have an x^{n+i-1} coefficient of n. Each time we have a difference, we replace a 1 with a -1, so with k differences we will have an x^{n+i-1} coefficient of $n-2k$.

Therefore, our algorithm will indeed return all indices i such that pattern matches at i.

The time complexity is $O(m \log m)$ for an FFT to multiply a polynomial of degree m and a polynomial of degree n because $m \geq n$. Looping and checking the coefficients takes $O(m)$, which is insignificant compared to the $m \log m$ term, so we can exclude it from our runtime. Therefore, we see that our algorithm works and runs in $O(m \log m)$. ■