ALGORITHMS IN ACTION - FFT

BASED ON LECTURES BY URI ZWICK AND HAIM KAPLAN

1. Discrete Fourier Transform

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} & \vdots & \\ \cdots & \omega_n^{jk} & \cdots \\ & \vdots & \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

Where ω_n is the n-th root of unity. For example, on $n = 4$

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & 1 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega \end{pmatrix} \begin{pmatrix} x_0 \\ y_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

and $\omega = \omega_4 = e^{2\pi i/4} = i$.

1.1. **DFT as polynomial evaluation.**

We can look at the DFT as an evaluation of a polynomial with coefficients $x_0, x_1, \ldots, x_n$ at the points $1, \omega, \omega^2, \omega^{n-1}$. If we denote this polynomial as X then:

$$X(z) = \sum_{k=0}^{n-1} x_k z^k = x_0 + x_1 z + \cdots + x_{n-1} z^{n-1}$$

and we can write the values y_j as

$$y_j = \sum_{k=0}^{n-1} \omega_n^{jk} x_k = X\left(\omega_n^k\right)$$

2. Fast Fourier Transform

The naive computation of the DFT required $\Theta\left(n^2\right)$ time. We will show an algorithm (FFT) which computes the DFT in $\Theta(n \log n)$ time. We will assume that $n = 2^k$.

Lemma 1. *If $x = (x_0, x_1, ..., x_{n-1}) \in \mathbb{R}^n$, $y = (y_0, y_1, \ldots, y_{n-1}) \in \mathbb{C}^n$ and $y = \mathrm{DFT}(x)$, then $y_{n-j} = y * j$, for $j = 1, .., n-1$.*

Proof. By definition,

$$y_j := \sum_{k=0}^{n-1} x_k \omega_n^{jk}$$

$$y_{n-j} = \sum_{k=0}^{n-1} x_k \omega_n^{(n-j)k} = \sum_{k=0}^{n-1} x_k \omega_n^{-jk} \left(w_n^n\right)^k \overset{\star}{=} \sum_{k=0}^{n-1} x_k \left(\omega_n^{jk}\right)^* = \left(\sum_{k=0}^{n-1} x_k \omega_n^{jk}\right)^* = y_j^*$$

From the Algorithms in Action website, Notes written by Arazim ©.

where $\star$ is due to the fact that $\omega_n^n = 1$ and that for all z such that $|z| = 1$, $z^* = z^{-1}$ $\square$

2.1. **Decomposing the DFT.** It's possible to compute the DFT of an even size n by computing two DFT's of size . Defining $x_{(0)} = (x_0, x_2, \dots, x_{n-2})$ and $x_{(1)} = (x_1, x_3, \dots, x_{n-1})$ as the even and odd parts of x. We can create the polynomials

$$X(z) = \sum_{j=0}^{n-1} x_j z^j \qquad X_{(0)}(z) = \sum_{j=0}^{n/2-1} x_{2j} z^j \qquad X(z) = \sum_{j=0}^{n/2-1} x_{2j+1} z^j$$

and thus, the value of this polynomial is equal to

$$X(z) = X_{(0)}\left(z^2\right) + zX_{(1)}\left(z^2\right)$$

Since we need to evaluate $X(z)$ at the points $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$, we will want to evaluate $X_{(0)}(z)$ and $X_{(1)}(z)$ at $\omega_n^0, \omega_n^2, \dots, \omega_n^{2(n-1)}$. These n points are exactly $\omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n-1}, \omega_{n/2}^0, \omega_{n/2}^1, \dots, \omega_{n/2}^{n-1}$ and as a result we only need to calculate DFT $\left(x_{(0)}\right)$ and DFT $\left(x_{(1)}\right)$, use each number twice and multiply the values of DFT $\left(x_{(1)}\right)$ by the appropriate powers of ω_n. Thus, we arrive at the following algorithm:

Algorithm 1 FFT($x_0, x_1, \dots, x_{n-1}$)

```
1: if n=2 then
2:     return $(x_0 + x_1, x_0 - x_1)$
3: end if
4: $(a_0, a_1, \dots, a_{n/2-1} \leftarrow$ FFT$(x_0, x_2, \dots, x_{n-2})$
5: $(b_0, b_1, \dots, b_{n/2-1} \leftarrow$ FFT$(x_1, x_3, \dots, x_{n-1})$
6: for $j \leftarrow 0$ to $n/2 - 1$ do
7:     $y_j \leftarrow a_j + \omega_n^j b_j$
8:     $y_{n/2+j} \leftarrow a_j - \omega_n^j b_j$
9: end for
10: return $(y_0, y_1, \dots, y_{n-1})$
```

2.2. **Complexity analysis of the FFT.** Denoting the cost of an FFT of size n as $T(n)$ we can easily see that

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

and as a result, $T(n) = \mathcal{O}(n \log n)$. Defining $A(n)$as the number of additions/subtractions in a FFT of size n, and $M(n)$as the number of multiplications in a FFT of size n, we have that:

$$\begin{array}{cc} A(2) = 2 & M(2) = 0 \\ A(n) = 2A(n/2) + n & M(n) = 2M(n/2) + n/2 \\ A(n) = n \log_2 n & M(n) = \frac{n}{2} \log_2 \frac{n}{2} \end{array}$$

2.3. **Circuit for FFT.** A major reason for the FFT algorithm is at which one can create a circuit which performs it. An example circuit is:

In this example, ω is known as the "twiddle factor" and another name for the whole diagram is a butterfly.

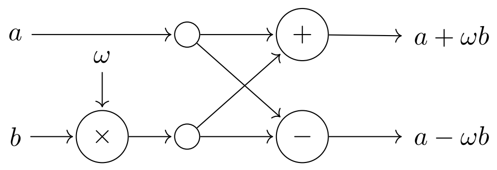

FIGURE 2.1. An FFT ciruit of size 2

3. The inverse DFT

The inverse of the DFT is very similar to the DFT:

$$\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} = \frac{1}{n} \begin{pmatrix} & \vdots & \\ \cdots & \omega_n^{-jk} & \cdots \\ & \vdots & \end{pmatrix} \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

3.1. **Proof of the inverse.** Recalling that if $C = AB$ then $c_{j,k} = \sum_{l=0}^{n-1} a_{j,l} b_{l,k}$. We will show that

$$\sum_{l=0}^{n-1} \omega_n^{jl} \omega_n^{-lk} = \begin{cases} n & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

In our case, $C = Id$ and A is the original DFT matrix. If $j = k$ then the claim is obvious. If $j \neq k$ then:

$$\sum_{l=0}^{n-1} \omega_n^{jl} \omega_n^{-jl} = \sum_{l=0}^{n-1} \omega_n^{(j-k)l} = \frac{\omega^{(j-k)n} - 1}{\omega_n^{j-k} - 1} = 0$$

Since $\left(\omega^{j-k}\right)^n = 1$ and $\omega^{j-k} \neq 1$ □

3.2. **DFT^{-1} as polynomial interpolation.** We will notice that $DFT\left(x\right)$ is an evaluation of the polynomial $X\left(z\right)$ at the points $1, \omega_n, \omega_n^2, \ldots, \omega_n^{n-1}$. This gives us that DFT$^{-1}\left(y\right)$ interpolates the coefficients of $X\left(z\right)$, since these are set uniquely. As DFTand DFT^{-1} are inverses of each other, the interpolation is unique.

3.3. **The fourier basis.** We can define a basis for the function $L^2([0,1], \mathbb{R})$ (the space of real valued square integrable functions on the interval $[0,1]$), using the fourier basis

$$f_j = \frac{1}{\sqrt{n}} \left(1, \omega_n^{-j}, \omega_n^{-2j}, \ldots, \omega_n^{-(n-1)j}\right)^T$$

This basis is orthonormal, since

$$f_j^* f_k = \begin{cases} 1 & \text{if } j = k \\ 0 & \text{otherwise} \end{cases}$$

And the DFT performs a change of bassi from the standard basis to the fourier basis.

Exercise 2. Let $x = \left(f\left(0\right), f\left(\frac{1}{32}\right), \ldots, f\left(\frac{31}{32}\right)\right)$, where $f\left(x\right) = \sin\left(9\left(2\pi x\right)\right) + 3\sin\left(2\left(2\pi x\right)\right) + 2\cos\left(2\pi x\right)$, what is $DFT\left(x\right)$?

Solution:

$$y_j = \sum_{k=0}^{31} x_k w^{jk} \overset{x \in \mathbb{R}^n}{=} \langle e_j, x \rangle$$

$$\begin{aligned} f(x) &= sin(9(2\pi x)) + 3sin(2(2\pi x)) + 2cos(2\pi x) \\ &= \frac{(e^{i9\cdot 2\pi x} - e^{-9\cdot 2\pi ix})}{2i} + 3\frac{(e^{2\cdot 2\pi ix} - e^{-2\cdot 2\pi ix})}{2i} + \frac{(e^{2\pi ix} + e^{-2\pi ix})}{2} \end{aligned}$$

$$\begin{aligned} y_j &= \sum_{k=0}^{31} x_k w^{jk} = \langle e_j, x\rangle = \left\langle e_j, \frac{1}{2i}e_9 - \frac{1}{2i}e_{-9} + \frac{3}{2i}e_2 - \frac{3}{2i}e_{-2} + \frac{1}{2}e_1 + \frac{1}{2}e_{-1} \right\rangle \\ &= \left\langle e_j, \frac{1}{2i}e_9 - \frac{1}{2i}e_{23} + \frac{3}{2i}e_2 - \frac{3}{2i}e_{30} + \frac{1}{2}e_1 + \frac{1}{2}e_{31} \right\rangle \end{aligned}$$

$$\begin{matrix} y_9 = \dfrac{1}{2i}\cdot 32 & y_{23} = -\frac{1}{2i}\cdot 32 & y_2 = \dfrac{3}{2i}\cdot 32 \\ y_{30} = -\dfrac{3}{2i}\cdot 32 & y_1 = \frac{1}{2}\cdot 32 & y_{31} = \dfrac{1}{2}\cdot 32 \end{matrix}$$

3.4. **Circuit for FFT^{-1}.** In order to compute the inverse, we need to run the network backwards.

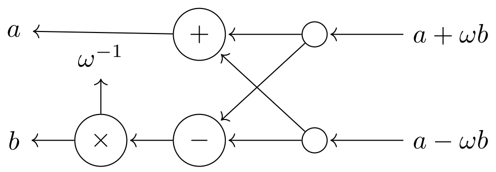

FIGURE 3.1. An FFT^{-1} ciruit of size 2

We can see that this is the inverse of 2.3

4. Convolution

Given two vectors $x = (x_0, x_1, \ldots, x_{n-1})$ and $y = (y_0, y_1, \ldots, y_{n-1})$ we can define the convolution as a vector of length $2n$ $x * y = z = (z_0, z_1, \ldots, z_{2n-1})$ such that for all $0 \le k \le 2n-1$ we have:

$$z_k = \sum_{i+j=k} x_i y_j = \sum_i x_i y_{k-i} \qquad \max\{0, k-n\} \le i \le \min\{k, n\}$$

Example 3. For $n = 4$ we have:

$$\begin{matrix} z_0 = x_0y_0 & z_1 = x_0y_1 + x_1y_0 \\ z_2 = x_0y_2 + x_1y_1 + x_2y_0 & z_3 = x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0 \\ z_4 = x_1y_3 + x_2y_2 + x_3y_1 & z_5 = x_2y_3 + x_3y_2 \\ z_6 = x_3y_3 & z_7 = 0 \end{matrix}$$

4.1. **Convolution and polynomial multiplication.** Defining the polynomials

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \qquad B(x) = \sum_{k=0}^{n-1} b_k x^k$$

We will notice that their product $A(x)B(x)$ obeys an interesting rule:

$$\begin{aligned}C(x) =& A(x)B(x)\\ =&\left(\sum_{j=0}^{n-1} a_j x^j\right)\left(\sum_{k=0}^{n-1} b_k x^k\right)\\ =&\sum_{i=0}^{2n-2}\left(\sum_{j+k=i} a_j b_k\right)x^i\\ =&\sum_{i=0}^{2k-1} c_i x^i\end{aligned}$$

Thus, $c = (c_1, c_2, \ldots, c_{2n-1}) = a * b$

4.2. **Cyclic convolution.** Similarily to the convolution, we can define $z = z \circledast y$ as the cyclic convolution,in this case z is defined as

$$z_k = \sum_{i+j\equiv k(\bmod n)} x_i y_j = \sum_{i+j=k} x_i y_j + \sum_{i+j=n+k} x_i y_j$$

Example 4. For $n = 4$ we have:

$$\begin{aligned}z_0 =& x_0y_0 + x_1y_3 + x_2y_2 + x_3y_1\\ z_1 =& x_0y_1 + x_1y_0 + x_2y_3 + x_3y_2\\ z_2 =& x_0y_2 + x_1y_1 + x_2y_0 + x_3y_3\\ z_3 =& x_0y_3 + x_1y_2 + x_2y_1 + x_3y_0\end{aligned}$$

4.2.1. *Relationship between convolution and cyclic convolution.* We can reduce the convolution to a cyclic convolution by padding:

$$x' = \left(x_0, x_1, \ldots, x_{n-1}, \overbrace{0, 0, \ldots, 0}^{n}\right) \qquad y' = \left(y_0, y_1, \ldots, y_{n-1}, \overbrace{0, 0, \ldots, 0}^{n}\right)$$

$$x * y = x' \circledast y'$$

4.3. **The convolution theorem.** This theorem enables us to quickly and efficiently compute the product of two polynomials.

Theorem 5.

$$x \circledast y = DFT^{-1}\left(DFT(x) \cdot DFT(y)\right)$$

Proof. We will define

$$XY(z) = \sum_{i=1}^{n-1}\left(\sum_{j+k\equiv i} x_i y_k\right) z^i$$

as the polynomial corresponding to $x \circledast y$ and show that $XY\left(\omega_n^l\right) = X\left(\omega_n^l\right)Y\left(\omega_n^l\right)$ for every $l = 0, 1, \ldots, n-1$ and as a result of the interpolation theorem, we have uniqueness and the polynomials are equal.

$$\begin{aligned}
X\left(\omega_n^l\right)Y\left(\omega_n^l\right) &= \left(\sum_{j=0}^{n-1} x_j\omega^{lj}\right)\left(\sum_{j=0}^{n-1} y_k\omega^{lk}\right)\\
&= \sum_{i=1}^{2n-2}\left(\sum_{j+k\equiv i} x_jy_k\right)\omega^{li}\\
&\overset{\star}{=} \sum_{i=0}^{n-1}\left(\sum_{j+k\equiv i} x_jy_k\right)\omega^{li}\\
&= XY\left(\omega^l\right)
\end{aligned}$$

where $\star$ comes from the fact that $\omega^{l(n+i)} = \omega^{li}$ and the claim follows. □

4.4. **The chirp transform.** Let z be an arbitrary complex number, the chirp transform of $x \in \mathbb{C}^n$ with respect to z is defined as

$$y_k = \sum_{j=0}^{n-1} x_jz^{jk} = X\left(z^k\right), \qquad k = 0, 1, \ldots, n-1$$

5. Polynomial arithmetic

For any two polynomials

$$A(x) = \sum_{j=0}^{n-1} a_jx^j \qquad B(x) = \sum_{k=0}^{n-1} b_kx^k$$

of degree $\leq n$, the coefficients of $A(x) + B(x)$ can be computed naively using n additions. However, a naive computation of the coefficients of $A(x)B(x)$ requires $\Theta\left(n^2\right)$. Using the FFT algorithm weare able to calculate the coefficients of $A(x)B(x)$ using only $\Theta(n\log n)$ operations.

5.1. **Karatsuba's algorithm.** For moderately large n, Karatsuba's algorithm works better in practice. Setting A_0, A_1, B_0 and B_1 to be:

$$A(x) = A_0(x) + x^{n/2}A_1(x) \qquad B(x) = B_0(x) + x^{n/2}B_1(x)$$

We will define the following:

$$\begin{aligned}
C_0(x) &= A_0(x)B_0(x)\\
C_1(x) &= \left(A_0(x) + A_1(x)\right)\left(B_0(x) + B_1(x)\right)\\
C_2(x) &= A_1(x)B_1(x)
\end{aligned}$$

And as a result,

$$A(x)B(x) = C_0(x) + x^{n/2}\left(C_1(x) - C_0(x) - C_2(x)\right) + x^nC_2(x)$$

Giving us the required result more efficiently.

5.1.1. *Complexity analysis.* eachC_i is of size $\frac{n}{2}$, the addtition factor is $\mathcal{O}(n)$. As a result, the total complexity is

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \Rightarrow T(n) = \mathcal{O}\left(n^{\log_2 3}\right) = \mathcal{O}\left(n^{1.59}\right)$$

5.2. **Integer polynomial multiplication.** So far, we assumed that all arithmetical operations are exact. This is not a realistic assumption, as ω_n is usually irrational. The FFT algorithm is well-behaved numerically, meaning the errors introduced if all operations are done using floating-point arithmetic are relatively small. In signal processing applications small errors are acceptable.

In our case, we want to add and multiply polynomials with integer coefficients and we want an exact result. If we use use high enough precision, we can use FFT and FFT^{-1} and round the results obtained to the nearest integers.

This isn't proven here but to multiply two polynomials of degree at most n with integer coefficients of absolute value at most n, $\mathcal{O}(\log n)$ bits of precision are enough.

6. Integer multiplication -Schonage-Strassen's algorithm

There are practical applications, e.g., cryptography, that require multiplying very large integers. The naïve method for multiplying two n-bit numbers requires $\Theta\left(n^2\right)$ bit operations. We can use FFTs to obtain a faster integer multplication algorithm since integer multplication can be reduced to polynomial multiplication.

6.1. **Integer multiplication and FFT.** The basic idea is to encode the numbers as polynomials and evaluate them at 2, then calculate their product.

$$\begin{aligned} x = (x_{n-1}, \ldots, x_1, x_0)_2 &= \sum_{i=0}^{n-1} x_i 2^i = X(2) \\ y = (y_{n-1}, \ldots, y_1, y_0)_2 &= \sum_{i=0}^{n-1} y_i 2^i = Y(2) \\ x \cdot y = z = (z_{2n-1}, \ldots, z_1, z_0)_2 &= \sum_{i=0}^{2n-1} z_i 2^i = Z(2) \end{aligned}$$

We can easily compute this since it is just polynomial multiplication. However we aren't done since the z_i aren't in binary, this is OK since $0 \le z_i < n$ this isn't a serious problem. We will show some clever stricks to speed the algorithm.

6.2. **Splitting into blocks.** Assuming $n = 2^k$ $(k = \log n)$ we will split the integers x, y of length n into n/k blocks of size k each, thus:

$$\begin{aligned} x &= (\overbrace{x_{n/k-1}}^{k=\log n}, \overbrace{x_{n/k-0}}^{k}, \ldots, \overbrace{x_0}^{k})_k = \sum_{i=0}^{n/k-1} x_i 2^{ki} = X\left(2^k\right) \\ y &= (\overbrace{y_{n/k-1}}^{k=\log n}, \overbrace{y_{n/k-0}}^{k}, \ldots, \overbrace{y_0}^{k})_k = \sum_{i=0}^{n/k-1} y_i 2^{ki} = Y\left(2^k\right) \end{aligned}$$

and as before, compute

$$x \cdot y = z = \left(z_{2n/k-1}, \ldots, z_1, z_0\right)_k = \sum_{i=0}^{2n/k-1} z_i 2^{ki} = Z\left(2^k\right)$$

Each z_i is equal to $\sum_{j+l=i} x_j y_l$ and since $0 \le x_j, y_k < 2^k = n$ we have that $0 \le z_i \le (n/\log n)\, n^2 < n^3$. Thus, each z_i is at most a three digit number base n.

As a result, by concatenating the z_i into three long integers as shown in the picture, we cna arrive at the result by adding these $2n$-bit numbers in $\mathcal{O}(n)$ time. In total, we performer two FFTs and one FFT^{-1} of size $n/k = n/\log n$.
Each input number is an integer between 0 and $n-1$.
Each output number is an integer between 0 and $n^3 - 1$.

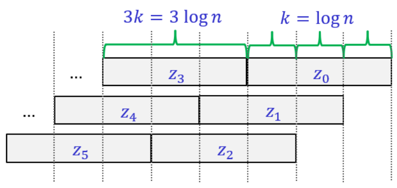

FIGURE 6.1. Adding the z_i

6.2.1. *Complexity analysis.* Defining $M(n)$as the total nummber of bit operations performed, we have that

$$M(n) = \mathcal{O}\left(\frac{n}{\log n} \log \frac{n}{\log n} \times M(\mathcal{O}(\log n))\right) = \mathcal{O}(nM(\mathcal{O}(\log n)))$$

Where $\frac{n}{\log n} \log \frac{n}{\log n}$ is the number of arithmetical operations performed in an FFT of size $n/\log n$. We can perform this recursively or we can choose to stop the recursion and perform the multiplication naively, thus

$$M(n) \in \left\{ \begin{array}{cc} \mathcal{O}\left(n^2\right) & \mathcal{O}\left(n \log^2 n\right) \\ \mathcal{O}\left(n \log n (\log\log n)^2\right) & \dots \end{array} \right\}$$

6.3. **The improved algorithm.** In 1971 Schönhage-Strassen showed an improved version of the algorithm, with a running time of $\mathcal{O}(n \log n (\log\log n))$. This improvemnt was obtained by performing the FFTs in an integer ring where $\omega = 2$ is a primitive root of unity.

Definition 6 (primitive root of unity)**.** An element ω is a primitive n-th root of unity in a ring R iff:

(1) $\omega^n = 1$.
(2) $\omega^k \ne 1$ for all $k = 0, 1, \dots, n-1$

For example, in $\mathbb{C}$, $e^{\frac{2\pi}{n}}$ is a primitive n-th root of unity, as are all $e^{\frac{2\pi i}{n}}$ if i is relatively prime to n. In $\mathbb{R}$ the only primitve root is -1, for $n = 2$.

6.3.1. *Number theoretic background.*

Theorem 7. *$\mathbb{Z}_m$ is a ring for all integers m and for any prime m, $\mathbb{Z}_m$ is a field.*

Theorem 8. *If p is prime, then $\mathbb{Z}_p$has a generator, an element g such that $g^{p-1} = 1$ but $g^i \ne 1$ for $i = 2, 3, \dots, p-2$*

Lemma 9. *If p is prime, $n | p-1$ and $k = (p-1)/n$ then $\omega = g^k$ is a primitive n-th root of unity.*

6.3.2. *FFT in prime fields.*

Example 10. Multiply two integer polynomials of degree < 512.

We need to compute FFT and FFT^{-1} with $n = 1024$. We would like to find a prime p such that $1024 | p-1$. We could take $p = 12 \cdot 1024 + 1 = 12{,}289$ but then we will get the coefficents modulo $12{,}289$ and the output coefficients are in the range $0, 1, \dots, 1024^3 - 1$.
In total we would like to find a prime $p > 1024^3$ such that $1024 | p-1$. For example, $p = \left(1024^2 + 8\right) \cdot 1024 + 1$ and then 5 is a generator and we can take $\omega = 5^{(p-1)/1024} = 381{,}780{,}781$ to be our primitive n-th root of unity.

The problem is that this isn't necessarily faster than working with floating point complex numbers, even though we don't need to worry about numerical errors. For this solution to work, we would need to find appropriate prime numbers and generators.

6.3.3. *FFT in rings.* Luckily for us, DFT and FFT does not have to be a field.

Lemma 11. *Let n and ω be positive powers of 2. ω is a primitive n-th root of unity in $\mathbb{Z}_m$ where $m = \omega^{n/2} + 1$.*

We will notice that multiplication by ω^k is just a shift and taking $\mod m$ is a simple operation since $m - 1 = 2^a$ for some $a \in \mathbb{N}$.

FFT performs $\mathcal{O}(n \log n)$ arithmetical operations. However, they are all either additions or multiplications by ω^k. To compute a convolution, we only need n multiplications, other than multiplications by ω^k. Break two n-bit integers into n_1 blocks of n_2-bits each.

$$M(n) = \mathcal{O}(n_1 \log n_1 \times M(\mathcal{O}(n_2)))$$

Since in a computer, multiplications by ω^k are essentially additions, then

$$M(n) = \mathcal{O}(n_1 \log n_1 \mathcal{O}(n_2) + n_1 M(\mathcal{O}(n_2)))$$

There are many technical problems to overcome. We have to choose $n_1 = \sqrt{n}$ rather than $n_1 = n/\log n$. The end result is an integer multiplication algorithm that performs only $\mathcal{O}\left(n \log n (\log\log n)^2\right)$ bit operations.

7. String matching

The classical problem presented here is: Given a text of length n and a pattern of length m, to find all occurrences of the pattern in the text. The naïve algorithm runs in $\mathcal{O}(mn)$ time. In previous courses we learnt of algorithms that run in $\mathcal{O}(m+n)$.

We will present solutions for a few variants of this problem:

(1) Counting the number of matches/mismatches in each alignment of the pattern with the text.
(2) Find all aligments with at most k mismatches.
(3) Allowing a wildcard that matches any (single) symbol in the pattern and/or text.

Traditional algorithms and techniques aren't so efficient for these variants.

7.1. **Cross-correlation.** The only difference between cross-correlation and convolution is a time reversal on one of the inputs, with a shift of indices.

$$z_k = \sum_i x_i y_{i-k} = \sum_j x_{j+k} y_j = \left(x * y^R\right)_{k+m-1}$$

If x is of length n and y has a length of m where $m \leq n$, then $k = 1-m, \ldots, n-1$. The correlation of two vectors of length n can be computed in $\mathcal{O}(n \log n)$ time.

Exercise 12. The correlation of two vectors of length n and m, where $m \leq n$, can be computed in $\mathcal{O}(n \log m)$ time.

Example 13. For $n = m = 4$ we have:

$$\begin{aligned} z_{-3} &= x_0 y_3 & z_{-2} &= x_0 y_1 + x_1 y_0 \\ z_{-1} &= x_0 y_1 + x_1 y_2 + x_2 y_3 & z_0 &= x_0 y_0 + x_1 y_1 + x_2 y_2 + x_3 y_3 \\ z_1 &= x_1 y_0 + x_3 y_2 + x_3 y_2 & z_2 &= x_2 y_0 + x_3 y_1 \\ z_2 &= x_3 y_0 \end{aligned}$$

7.1.1. *Counting mismatches.* Let Σ be the alphabet of the pattern and text. We may assume that $|\Sigma| \leq m+1$, since we only to check if it is equal to one of the characters in x or not in the pattern.

For every $a \in \Sigma$ we will create two boolean strings;

$$P_a[j] = 1 \Leftrightarrow P[j] = a \qquad T_a[i] = 1 \Leftrightarrow T[i] \neq a$$

The correlation of P_a and T_a counts mismatches involving a, by summing over all $a \in \Sigma$ we get the total number of mismatches.

Complexity: $\mathcal{O}(|\Sigma| \, n \log m)$ word operations. This is fast only if $|\Sigma|$ is small.

7.1.2. *Counting mismatches with wildcards.* Similarily to before, we will create two boolean strings

$$P_a[j] = 1 \Leftrightarrow P[j] = a \qquad T_a[i] = 1 \Leftrightarrow T[i] \neq a \wedge T[i] \neq *$$

Complexity: $\mathcal{O}(|\Sigma| \, n \log m)$ word operations. If we only want to find exact matches, we can replace each character by a $\lceil \log_2 |\Sigma| \rceil$ bit string and the complexity will drop to $\mathcal{O}(\log |\Sigma| \, n \log m)$.

7.2. L_2 **matching.** Standard string matching uses the Hamming distance. Two characters either match or they do not. The letter a is not closer to b than to z, even though alphabetically they are right next to each other. Suppose that each "character" is a real number. We want to find approximate matches. For each $k = 0, 1, \dots, n-m$ we want to compute

$$d_k = \sum_{j=0}^{m-1} (p_j - t_{k+j})^2$$

Complexity: We will split the above expression into seperate items

$$\sum_{j=0}^{m-1} (p_j - t_{k+j})^2 = \sum_{j=0}^{m-1} p_j^2 - 2\sum_{j=0}^{m-1} p_j t_{k+j} + \sum_{j=0}^{m-1} t_{k+j}^2$$

The first term can be calculated in $\mathcal{O}(m)$ the second is just the correlation which can be computed in $\mathcal{O}(n \log m)$ and the third in $\mathcal{O}(n)$. Thus the total complexity for L_2 matching is $\mathcal{O}(n \log m)$.

7.3. **Exact matches with wildcards (Clifford-Clifford).** Replace each character by a positive integer. Replace the wildcard by 0. For each $k \in [n-m]$ compute

$$d_k = \sum_{j=0}^{m-1} p_j t_{k+j} (p_j - t_{k+j})^2$$

There is an exact match at position k iff $d_k = 0$.

Complexity: As before, we will split the above expression into seperate items

$$d_k = \sum_{j=0}^{m-1} p_j t_{k+j} (p_j - t_{k+j})^2 = \sum_{j=0}^{m-1} p_j^3 t_{k+j} - 2\sum_{j=0}^{m-1} p_j^2 t_{k+j}^2 + \sum_{j=0}^{m-1} p_j t_{k+j}^3$$

We can compute the three correlations in $\mathcal{O}(m \log n)$, giving us a running time imdependent of $|\Sigma|$.