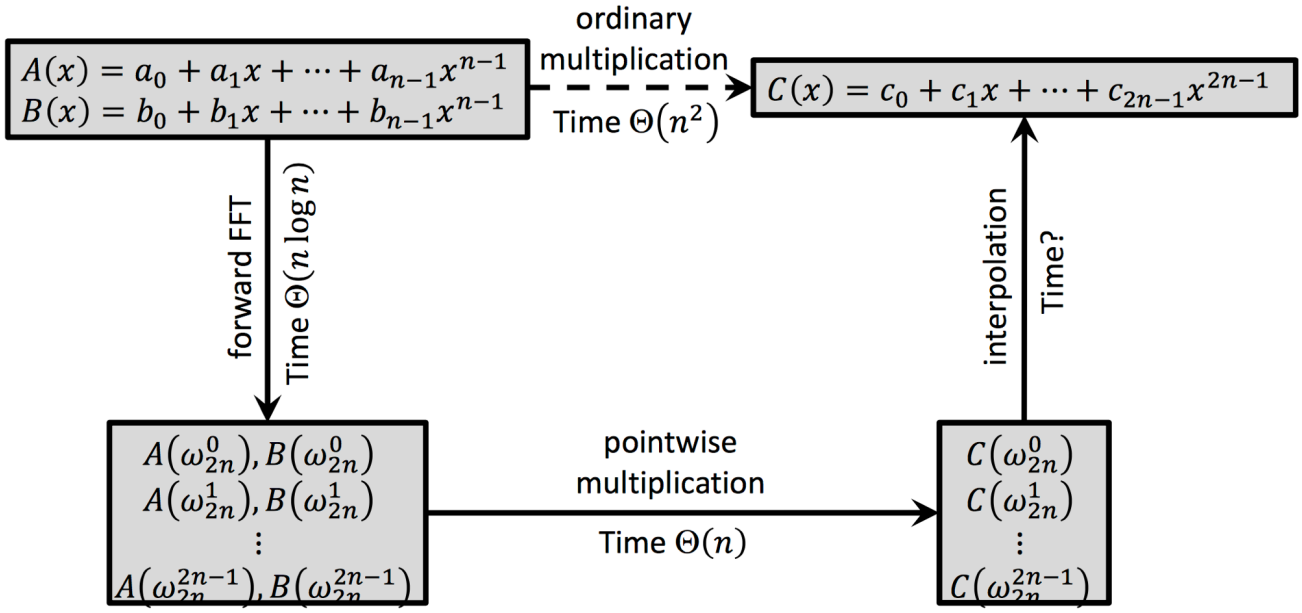
CSE548 & AMS542: Analysis of Algorithms, Fall 2017 **Lecturer:** Rezaul Chowdhury
Topic: Divide-and-Conquer Algorithms: Polynomial Multiplication **Date:** 9/18/17
Scribe: Harsh Gupta

6.1 Polynomial Multiplication : Point-Value Representation of Polynomial

As explained in previous lecture, when polynomials are represented in coefficient form, then product of two polynomials of degree bound n takes $\Theta\left(n^2\right)$time in traditional way and their addition takes $\Theta\left(n\right)$ time.
Another way to find the product of 2 polynomial is to transform it to point value form and then multiply it which takes $\Theta\left(n\right)$ and transform it back to coefficient form.
In the previous lecture we have seen that if a polynomial is given in coefficient form we can choose n points and convert it to point value form in $\Theta\left(n\log n\right)$ time.They are not any arbitrary set of points but must be n^{th} root of unity for degree bound n, then we will multiply polynomial and then go back to coefficient form. This has to be done in less than $\Theta\left(n^2\right)$ time.

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$$

$$A : (x_0, y_0), (x_1, y_1), ..., (x_n, y_n)$$



Forward equation (converting coefficient form to point value form) also works for inverse problem(converting point value form to coefficient form) with small modifications.

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ A(x_2) \\ . \\ . \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & x_0 & (x_0)^2 & \dots & (x_0)^{n-1} \\ 1 & x_1 & (x_1)^2 & \dots & (x_1)^{n-1} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ 1 & x_{n-1} & (x_{n-1}^2 & \dots & (x_{n-1})^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \\ a_{n-1} \end{bmatrix}$$

For degree bound n polynomial A and n distinct points we can find values of $A(x_i)$ where n distinct points are n^{th} root of unity.
$a_0, a_1 \dots$ are the coefficient of polynomial A(x). After replacing x_i with n^{th} root of unity we get y_i

$$\underbrace{\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & (\omega_n)^2 & \dots & (\omega_n)^{n-1} \\ 1 & \omega_n^2 & (\omega_n^2)^2 & \dots & (\omega_n^2)^{n-1} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ 1 & \omega_n^{n-1} & (\omega_n^{n-1})^2 & \dots & (\omega_n^{n-1})^{n-1} \end{bmatrix}}_{V(\omega_n)} \underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \\ a_{n-1} \end{bmatrix}}_{\overline{a}} = \underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ . \\ . \\ y_{n-1} \end{bmatrix}}_{\overline{y}}$$

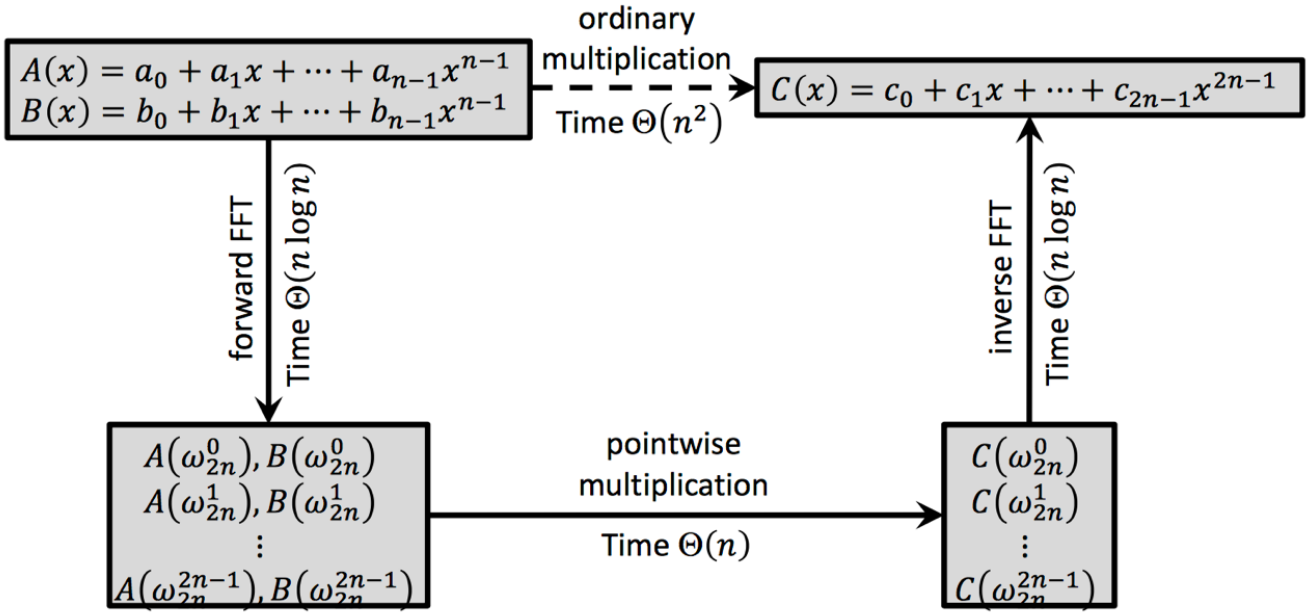
This above matrix is called $Vandermonde\ matrix$
In this matrix all the elements in the first row and first columns are 1. And other values are the powers of the n_{th} primitive roots of unity.
y vector can be computed in $\Theta\left(n \log n\right)$ time usinf FOrward Fast Fourier Transformation.

$\Rightarrow V(\omega_n).\overline{a} = \overline{y}$

Now the inverse of this problem is that we are given $\overline{y}$ which is obtained by evaluating coefficient form of polynomial at n^{th} roots of unity.
y_i is obtained by evaluating the polynomial at ω_n^i. We have to find the $\overline{a}$ which are the coefficients of the polynomial. We have to solve

$$\overline{a} = [V(\omega_n)]^{-1}.\overline{y}$$

It turns out the inverse of Vandermonde Matrix is $\frac{1}{n}V(\frac{1}{\omega_n})$
We have to just substitute this value and find $\overline{a}$. It looks similar with minor modification, need to add $\frac{1}{n}$ and replace ω_n by $\frac{1}{\omega_n}$



6.1.1 Proof for Inverse Vandermonde Matrix

Lets assume

$$U(\omega_n) = \frac{1}{n}V(\frac{1}{\omega_n})$$

We have to prove that

$$U(\omega_n).V(\omega_n) = I_n$$

where I_n is nxn Identity matrix, where diagonals are 1 and rest all values in the matrix are 0s. Entry at i^{th} row,j^{th} column in

$[V(\omega_n)]_{ij}$ = ω_n^{ij} and $[U(\omega_n)]_{ij} = \frac{1}{n}\frac{1}{\omega_n^{ij}} = \frac{1}{n}\omega_n^{-ij}$

Entry (p,q) of $U(\omega_n)V(\omega_n)$ will be :

$$[U(\omega_n)V(\omega_n)]_{pq} = \sum_{k=0}^{n-1}[U(\omega_n)]_{pk}[V(\omega_n)]_{kq} = \frac{1}{n}\sum_{k=0}^{n-1}\omega_n^{k(q-p)}$$

When p = q :

$$[U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n}\sum_{k=0}^{n-1}\omega_n^0 = \frac{1}{n}\sum_{k=0}^{n-1}1 = \frac{1}{n}n = 1$$

When p $\neq$ q :

$$[U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n}\sum_{k=0}^{n-1}(\omega_n^{q-p})^k = \frac{1}{n}\frac{(\omega_n^{q-p})^n - 1}{\omega_n^{q-p} - 1}$$

Taking the power n inside, ω_n^n = 1 :

$$[U(\omega_n)V(\omega_n)]_{pq} = \frac{1}{n}\frac{(\omega_n^n)^{q-p} - 1}{\omega_n^{q-p} - 1} = \frac{1}{n}\frac{(1)^{q-p} - 1}{\omega_n^{q-p} - 1} = 0$$

Hence it is proved that $U(\omega_n)V(\omega_n) = I_n$

$$\underbrace{\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ . \\ . \\ a_{n-1} \end{bmatrix}}_{\overline{a}} = \frac{1}{n}\underbrace{\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \frac{1}{\omega_n} & (\frac{1}{\omega_n})^2 & \dots & (\frac{1}{\omega_n})^{n-1} \\ 1 & \frac{1}{\omega_n^2} & (\frac{1}{\omega_n^2})^2 & \dots & (\frac{1}{\omega_n^2})^{n-1} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ 1 & \frac{1}{\omega_n^{n-1}} & (\frac{1}{\omega_n^{n-1}})^2 & \dots & (\frac{1}{\omega_n^{n-1}})^{n-1} \end{bmatrix}}_{[V(\omega_n)]^{-1}}\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ . \\ . \\ y_{n-1} \end{bmatrix}}_{\overline{y}}$$

This inverse problem is almost similar to the forward problem, and can be solved in $\Theta\left(n\log(n)\right)$ time using the same algorithm as the forward FFT with only minor modifications

6.2 Applications of Fourier Transform and FFT

Fast Fourier Transformation has application in variety of fields, some examples are discussed here.

6.2.1 Signal Processing

Any periodic signal can be represented as a sum of a series of sinusoidal (sin and cosine) waves.
Every Signal can be decomposed as a collection of Harmonics,
We have a sin wave $s(x)$, that is shown below

Figure 6.2.1:

Using FFT, we can convert this sin wave from time domain to frequency domain. In frequency domain, this sin wave is decomposed into 6 different sin waves S(f). Again using FFT, we can convert the waves back to time domain.

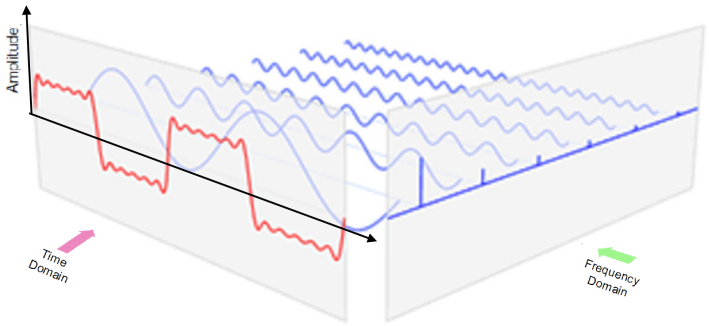


Figure 6.2.2:

This approach reveals the 6 different frequencies and their amplitude for the original sin wave.

Figure 6.2.3: Frequency

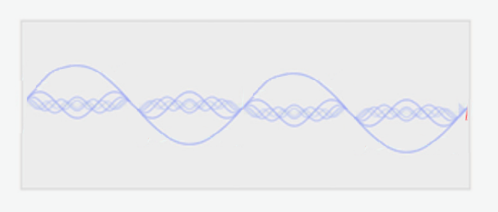


Figure 6.2.4: Composite Sine Waves that were part of original Sine Waves

Lets see how to calculate frequency from time domain and vice versa. Suppose $s(t)$ is the signal specified in the time domain. So

$$S(f) = s(t).e^{-2\pi ift}$$

This is a continuous wave so we will integrate on all the time stamps and result will be

$$S(f) = \int_{-\infty}^{\infty} s(t).e^{-2\pi ift} df$$

Value of $S(f)$ will be closer to zero if that signal is not among the 6 signals and it will have some value for these 6 signals.

The inverse of this problem is

$$s(t) = \int_{-\infty}^{\infty} S(f).e^{2\pi ift} df$$

From this we can find the signal for a particular frequency.

6.2.2 Noise Reduction

FFT can be used to reduce noise in photos when it is confined to only few frequencies. When the image is plotted in frequency domain then it becomes easy to remove these noise frequencies, once the noise frequencies are removed, the reverse FFT results in image without noise.

Figure 6.2.5: Original Image

Figure 6.2.6: Final Image without Noise

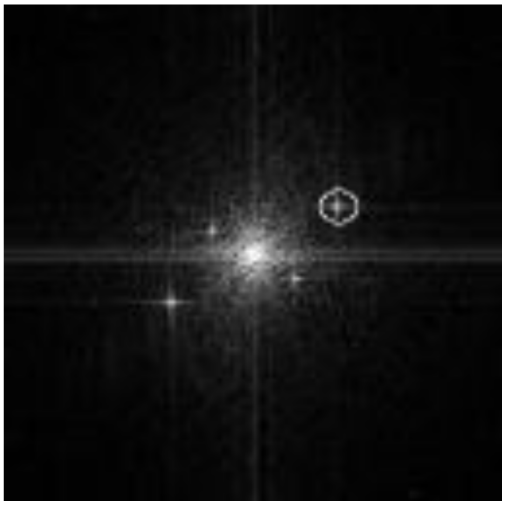


Figure 6.2.7: Frequency Domain with noise frequencies

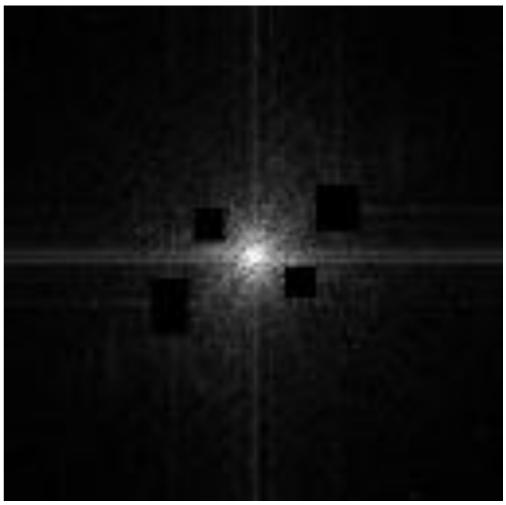


Figure 6.2.8: Frequency Domain after removing noise frequencies

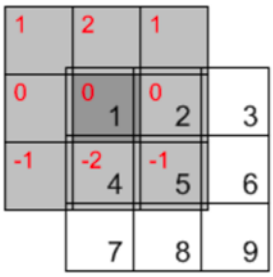
6.2.3 Data Compression

Discrete Cosine Transforms (DCT) are used for lossy data compression (e.g., MP3, JPEG, MPEG). DCT is a Fourier-related transform similar to DFT (Discrete Fourier Transform) but uses only real data (uses cosine waves only instead of both cosine and sine waves).Forward DCT transforms data from spatial to frequency domain. Each frequency component is represented using a fewer number of bits (i.e., truncated / quantized).Low amplitude high frequency components are also removed.Inverse DCT then transforms the data back to spatial domain.The resulting image compresses better

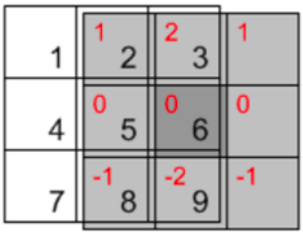
6.2.4 Protein-Protein Docking

For multiplying two uni-variate polynomials, the convolution approach takes $\Theta\left(n^2\right)$ time whereas by applying Fast Fourier transform the complexity reduces to $\Theta\left(n \log n\right)$. In case of bi-variate polynomial, if we want to find the largest value of coefficient in the product of two polynomials then by applying 2D convolution the time complexity is $\Theta\left(n^4\right)$.

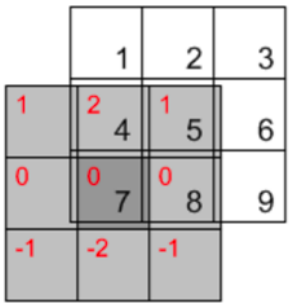
2D convolution in case of bi-variate polynomials is as explained below:



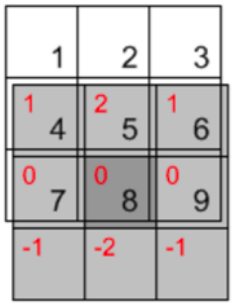
$$\begin{aligned} y[0,0] &= x[-1,-1]\cdot h[1,1] + x[0,-1]\cdot h[0,1] + x[1,-1]\cdot h[-1,1] \\ &\quad + x[-1,0]\cdot h[1,0] + x[0,0]\cdot h[0,0] + x[1,0]\cdot h[-1,0] \\ &\quad + x[-1,1]\cdot h[1,-1] + x[0,1]\cdot h[0,-1] + x[1,1]\cdot h[-1,-1] \\ &= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 0\cdot 0 + 1\cdot 0 + 2\cdot 0 + 0\cdot(-1) + 4\cdot(-2) + 5\cdot(-1) = -13 \end{aligned}$$



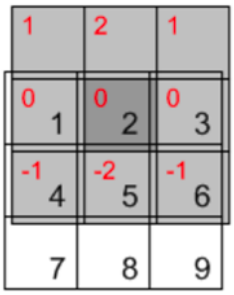
$$\begin{aligned} y[2,1] &= x[1,0]\cdot h[1,1] + x[2,0]\cdot h[0,1] + x[3,0]\cdot h[-1,1] \\ &\quad + x[1,1]\cdot h[1,0] + x[2,1]\cdot h[0,0] + x[3,1]\cdot h[-1,0] \\ &\quad + x[1,2]\cdot h[1,-1] + x[2,2]\cdot h[0,-1] + x[3,2]\cdot h[-1,-1] \\ &= 2\cdot 1 + 3\cdot 2 + 0\cdot 1 + 5\cdot 0 + 6\cdot 0 + 0\cdot 0 + 8\cdot(-1) + 9\cdot(-2) + 0\cdot(-1) = -18 \end{aligned}$$



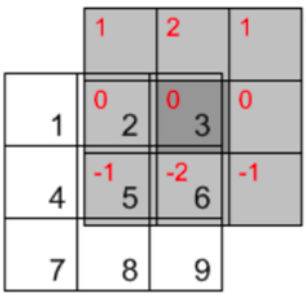
$$\begin{aligned} y[0,2] &= x[-1,1]\cdot h[1,1] + x[0,1]\cdot h[0,1] + x[1,1]\cdot h[-1,1] \\ &\quad + x[-1,2]\cdot h[1,0] + x[0,2]\cdot h[0,0] + x[1,2]\cdot h[-1,0] \\ &\quad + x[-1,3]\cdot h[1,-1] + x[0,3]\cdot h[0,-1] + x[1,3]\cdot h[-1,-1] \\ &= 0\cdot 1 + 4\cdot 2 + 5\cdot 1 + 0\cdot 0 + 7\cdot 0 + 8\cdot 0 + 0\cdot(-1) + 0\cdot(-2) + 0\cdot(-1) = 13 \end{aligned}$$



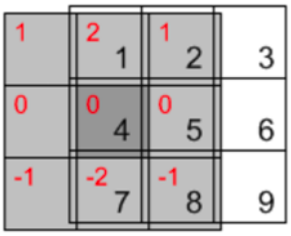
$$\begin{aligned} y[1,2] &= x[0,1]\cdot h[1,1] + x[1,1]\cdot h[0,1] + x[2,1]\cdot h[-1,1] \\ &\quad + x[0,2]\cdot h[1,0] + x[1,2]\cdot h[0,0] + x[2,2]\cdot h[-1,0] \\ &\quad + x[0,3]\cdot h[1,-1] + x[1,3]\cdot h[0,-1] + x[2,3]\cdot h[-1,-1] \\ &= 4\cdot 1 + 5\cdot 2 + 6\cdot 1 + 7\cdot 0 + 8\cdot 0 + 9\cdot 0 + 0\cdot(-1) + 0\cdot(-2) + 0\cdot(-1) = 20 \end{aligned}$$



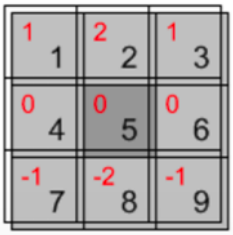
$$\begin{aligned} y[1,0] &= x[0,-1]\cdot h[1,1] + x[1,-1]\cdot h[0,1] + x[2,-1]\cdot h[-1,1] \\ &\quad + x[0,0]\cdot h[1,0] + x[1,0]\cdot h[0,0] + x[2,0]\cdot h[-1,0] \\ &\quad + x[0,1]\cdot h[1,-1] + x[1,1]\cdot h[0,-1] + x[2,1]\cdot h[-1,-1] \\ &= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 1\cdot 0 + 2\cdot 0 + 3\cdot 0 + 4\cdot(-1) + 5\cdot(-2) + 6\cdot(-1) = -20 \end{aligned}$$



$$\begin{aligned} y[2,0] &= x[1,-1]\cdot h[1,1] + x[2,-1]\cdot h[0,1] + x[3,-1]\cdot h[-1,1] \\ &\quad + x[1,0]\cdot h[1,0] + x[2,0]\cdot h[0,0] + x[3,0]\cdot h[-1,0] \\ &\quad + x[1,1]\cdot h[1,-1] + x[2,1]\cdot h[0,-1] + x[3,1]\cdot h[-1,-1] \\ &= 0\cdot 1 + 0\cdot 2 + 0\cdot 1 + 2\cdot 0 + 3\cdot 0 + 0\cdot 0 + 5\cdot(-1) + 6\cdot(-2) + 0\cdot(-1) = -17 \end{aligned}$$



$$y[0,1] = x[-1,0]\cdot h[1,1] + x[0,0]\cdot h[0,1] + x[1,0]\cdot h[-1,1]$$
$$+ x[-1,1]\cdot h[1,0] + x[0,1]\cdot h[0,0] + x[1,1]\cdot h[-1,0]$$
$$+ x[-1,2]\cdot h[1,-1] + x[0,2]\cdot h[0,-1] + x[1,2]\cdot h[-1,-1]$$
$$= 0\cdot 1 + 1\cdot 2 + 2\cdot 1 + 0\cdot 0 + 4\cdot 0 + 5\cdot 0 + 0\cdot(-1) + 7\cdot(-2) + 8\cdot(-1) = -18$$

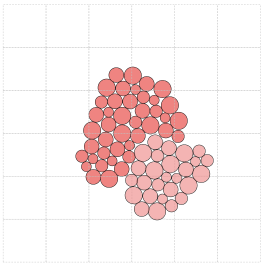


$$y[1,1] = x[0,0]\cdot h[1,1] + x[1,0]\cdot h[0,1] + x[2,0]\cdot h[-1,1]$$
$$+ x[0,1]\cdot h[1,0] + x[1,1]\cdot h[0,0] + x[2,1]\cdot h[-1,0]$$
$$+ x[0,2]\cdot h[1,-1] + x[1,2]\cdot h[0,-1] + x[2,2]\cdot h[-1,-1]$$
$$= 1\cdot 1 + 2\cdot 2 + 3\cdot 1 + 4\cdot 0 + 5\cdot 0 + 6\cdot 0 + 7\cdot(-1) + 8\cdot(-2) + 9\cdot(-1) = -24$$

This is the naive approach to solve this problem, the other way is to apply Fast Fourier Transform which gives a time complexity of $\Theta\left(n^2 \log n\right)$ time.

The application of FFT can be done in several ways in which the easiest way is to transform the 2D polynomial in 1D, then apply FFT and combine the results to form a 2D polynomial.

This is the **basis of Protein-Protein Docking problem**, in which given two polynomials,we need to find the best relative transformation and conformations to obtain a stable complex. In order to find this, we have to achieve shape complimentary.



Similarly in case of 3D, the time complexity will be $\Theta\left(n^3 \log n\right)$. Protein protein docking is a hard problem because of two reasons:

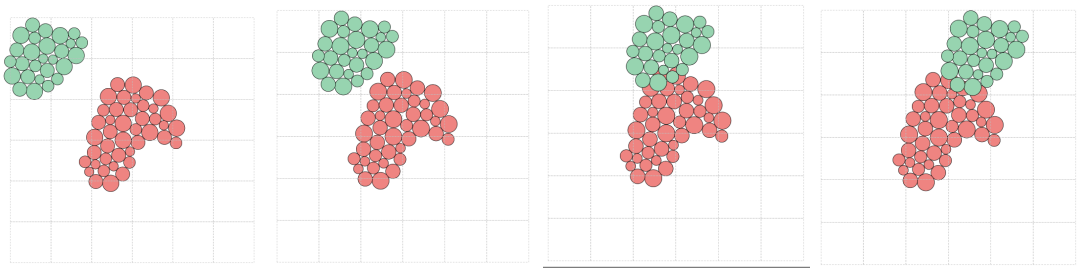
1. The search space is large. In case of rigid proteins, the two proteins will be in 3D thus the search space will be in 6D which is huge.
2. Protein flexibility.

To find the best possible docking position, we will plot the proteins in grid. For 2D, one protein will be in one grid and another protein in another grid, so total there will be two grids, and in case of 3D there will be six grids.

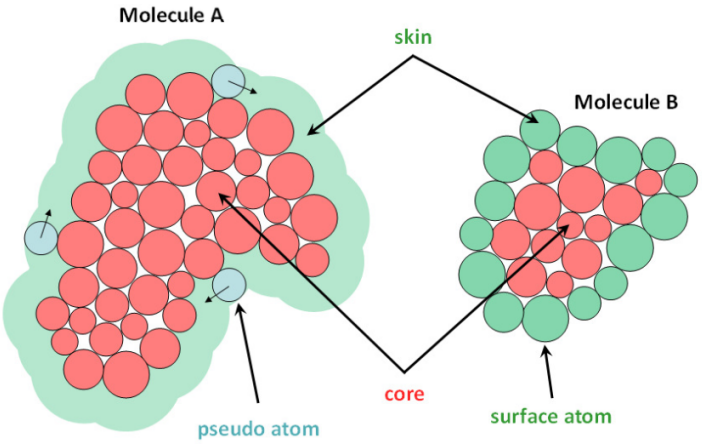
Each protein consists of various atoms. Each atom will have some influence around it, that will drop sharply if you move away from the radius.

Plot the proteins in 2D form as grids and for every grid point, figure out the contribution of all the atom to that grid point and it will be the value of that grid point. By looking at the convolution of each grid point,

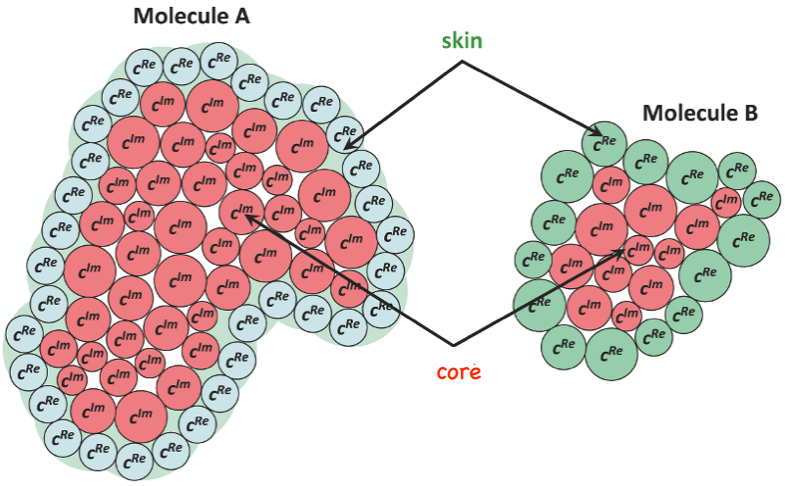
we can know the relation(distance) between the proteins. If they are near, the contribution is higher and if far it will be less. Our aim is to maximize the contribution.



To find the best position, we will fix the position of one protein; say Protein A and will move the other one; Protein B over Protein A. While overlapping it we will not rotate the Protein B. For each fixed rotation of Protein B, we will find the best overlapping position. If they have **shape complimentary**, score is good, if they are far, score is very less and if they overlap, the resultant score is worse. Our goal here is to find a rotation and a overlapping position that will result in highest score.



For protein A, create a pseudo skin around it by taking a pseudo item and rolling over the surface of the protein. That will be a thin artificial surface layer for protein A. For protein B, find the surface items, those items will act as the skin for protein B(real surface layer).Assign positive real weight to skin/surface atoms and large positive imaginary weight to core atoms, so that when core-core overlap the resultant product will be negative.



c^{Re} is real number and $c^{Im}$is imaginary number

Skin-skin overlap will result in positive product. Then embed both the proteins inside separate grid, so that each grid cell has a weight associated with it. if the grid contains the core atoms the weight will be a complex number. We will assign coefficients to each cell in the grid and will represent each protein as a bivariate polynomial. Now we have two polynomials, each representing the current position of the protein A and protein B, and the score of that position will be calculated by multiplying these two polynomials.

Since we have fixed the position of protein A, we will find all the possible rotations for Protein B. For each possible rotation, we will move protein B over protein A and will calculate a score for each move. Overlapping positions where skin-skin overlap is large and core-core overlap is less will result in high score. Skin-core or core-skin overlap will result in complex number that will have imaginary part, we can either ignore the imaginary part or can assign some low weight to them and subtract them from the overall score.

This approach if done in naive manner will result in $\Theta\left(n\right)^4$. The same calculation can also be done using FFT, by converting protein grid to bivariant polynomial.
We will get two polynomials; one static polynomial $P_A(x, y)$ representing the fixed position of protein A and another polynomial $P_A(x, y)$ representing a specific position of protein B in the grid.
These coefficient polynomials will be converted to point value form using FFT, then will do the multiplication and convert the product back to the coefficient form using reverse FFT. This approach will take only $\Theta\left(n^2 \log n\right)$.

To find the best docking position, we have to repeat the same approach for each rotation of protein B and the position with largest product will be the solution.

In case of 3D, the naive approach takes theta $\Theta\left(n\right)^6$ since the protein position will be represented as trivariant polynomial. Using Fourier Transform we can limit to $\Theta\left(n^4 \log n\right)$