**An excursion into Fast Fourier Transform — Pattern Matching and Other Problems**

[[Abhijit Mondal](https://mecha-mind.medium.com/?source=post_page-----81461f125880--------------------------------)](https://mecha-mind.medium.com/?source=post_page-----81461f125880--------------------------------)

[Abhijit Mondal](https://mecha-mind.medium.com/?source=post_page-----81461f125880--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F3d860626bbc7&operation=register&redirect=https%3A%2F%2Fmecha-mind.medium.com%2Fan-excursion-into-fast-fourier-transform-part-2-81461f125880&user=Abhijit+Mondal&userId=3d860626bbc7&source=post_page-3d860626bbc7----81461f125880---------------------post_header-----------)

14 min read

·

Nov 3

5

In this post I am going to solve a bunch of algorithmic problems to demonstrate the power of FFT algorithm and how it extends beyond time series analysis.

*For understanding the basics of FFT go through my previous post*

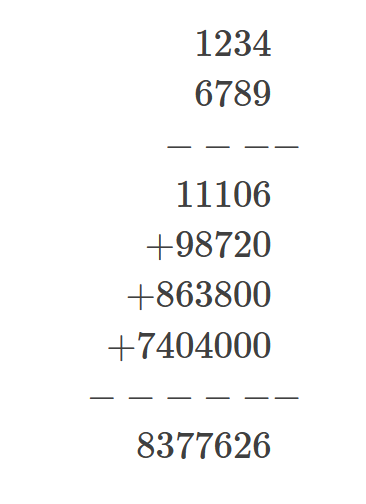
[An excursion into Fast Fourier Transform — The Basics and Time Series Analysis | by Abhijit Mondal | Nov, 2023 | Medium](https://medium.com/@mecha-mind/an-excursion-into-fast-fourier-transform-part-1-8a6498ee0c10)

The first problem we are going to solve is doing fast integer multiplication.

**Fast Integer Multiplication**

Everyone who has attended elementary school knows how to do multiplication. So what’s the big deal about it when it comes to FFT?

Lets revisit how would you do multiplication of 2 integers, for e.g. 1234 and 6789



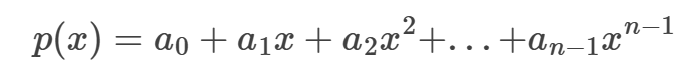
Multiply the 1st integer with each digit of the 2nd integer (starting from rightmost digit), left shift the multiplication result for each digit. Add all the results of multiplications.

*The time complexity of multiply 2 integers each with N digits is O(N²).*

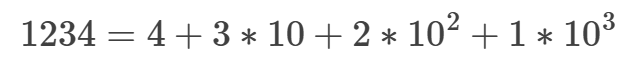
For very large integers i.e. N > O(10⁶), the multiplication is infeasible. For the moment assume that the computer hardware is capable of representing an integer of 10¹² digits (requiring around 40 bits to store it in memory).

Let’s take a diversion from integers to polynomials.

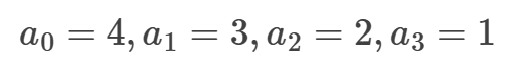
A polynomial p(x) of degree n-1 is represented as:



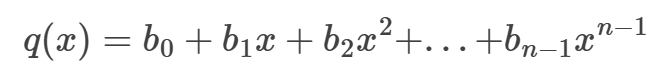
Any integer P can be represented as a polynomial. For e.g.



1234 is just p(x) evaluated as x=10 and n=4, with

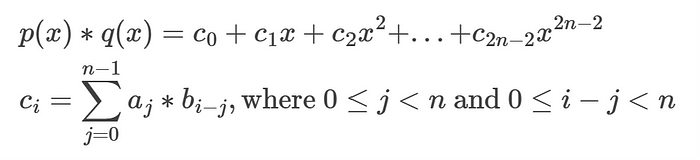


Similarly, let us define another polynomial q(x) of same degree n-1 as follows:



When we multiply p(x) with q(x), the resultant **polynomial is of degree at-most 2n-2** and each co-efficient ‘c’ of the resultant polynomial is evaluated as follows:

The **coefficient of x^i** in the multiplication is the summation of products of a\_j\*b\_(i-j) for all j ranging from 0 to n-1:



I would request the reader to verify and convince themselves the coefficients by multiplying the 2 polynomials by hand.

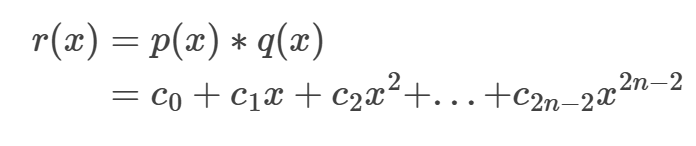
Thus multiplying 2 integers can also be represented as above, i.e. multiplying 2 polynomials and then evaluating the final polynomial at x=10. Here is a python code to implement multiplication using the polynomial representation.

def get\_coefficients(r):  
 coefficients = []  
   
 while r > 0:  
 coefficients += [r%10]  
 r = int(r/10)  
  
 return coefficients  
   
def multiply(p, q):  
 sign = (p >= 0 and q >= 0) or (p < 0 and q < 0)  
   
 p, q = abs(p), abs(q)  
   
 p\_coefficients = get\_coefficients(p)  
 q\_coefficients = get\_coefficients(q)  
  
 n, m = len(p\_coefficients), len(q\_coefficients)  
 new\_coefficients = [0]\*(n+m)  
   
 for i in range(n+m):  
 s = 0  
 for j in range(n):  
 if 0 <= j < n and 0 <= i-j < m:  
 s += p\_coefficients[j]\*q\_coefficients[i-j]  
   
 new\_coefficients[i] = s  
  
 h = 1  
 result = 0  
 for i in range(len(new\_coefficients)):  
 result += new\_coefficients[i]\*h  
 h \*= 10  
  
 return result if sign else -result  
  
p, q = 1234, 6789  
assert multiply(p, q) == p\*q

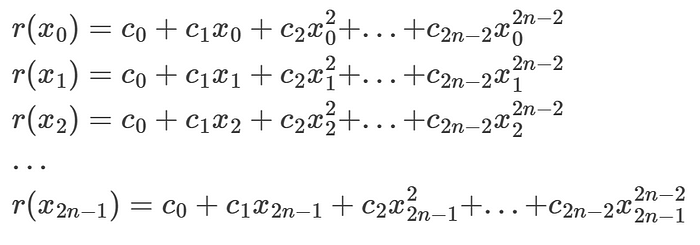
Huh !!! all those codes for just doing simple multiplication. The time complexity for the above code is clearly O(n²).

So how does fourier transform help us improving the time complexity from O(n²).

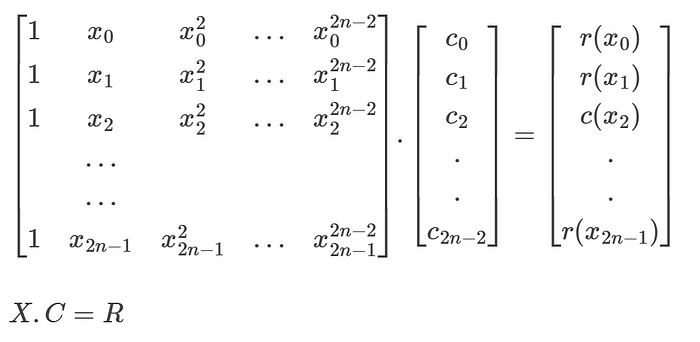
Let us rewrite the multiplication equation between p and q and denote by r.



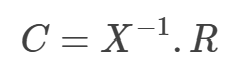
Now evaluate r at 2n different points x:



Representing the above equations in matrix notation, we get:



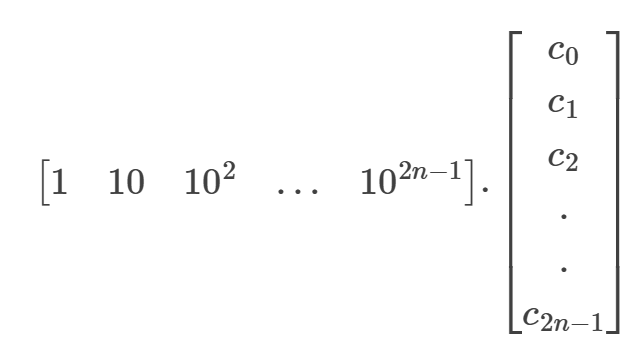
In order to obtain the column vector C, we need to multiply X inverse with R i.e.



Note that in order to find the inverse of the matrix X, it needs to be a square matrix but there are 2n-1 columns and 2n rows. So we add a dummy column corresponding to the 2n-1 th coefficient and add c\_(2n-1) = 0 in the C vector.

*It can be shown that the matrix C obtained using the above method is unique i.e. no two different C vectors can give X.C=R for the same X and R.*

Once we obtain the coefficients C, then we can obtain the multiplication results easily as seen earlier in our polynomial representation. Here is how we can do it using matrix multiplication.



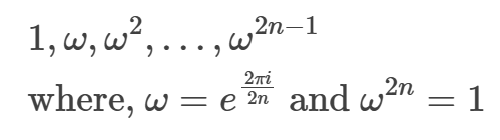
***Now the question arises how we can efficiently compute the C vector?***

Computing the R vector has a time complexity of O(n²) because the polynomial is of degree 2n-2 and there are 2n different values for the inputs X.

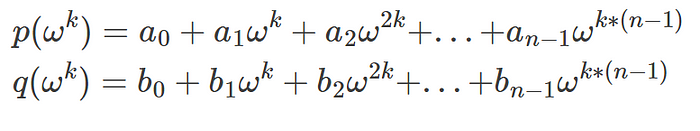
Computing the inverse of a generic matrix of size 2n x 2n has a time complexity of O(n³) and multiplying the inverse of size 2n x 2n with the R vector has complexity of O(n²).

*Thus time complexity to find C using the matrix route is****O(n³)****which is highly inefficient and sub-optimal compared to our python approach.*

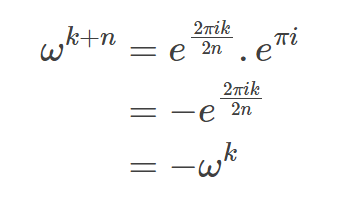
Instead of using any random 2n input points x for evaluating the R vectors, we are going to use the (2n) roots of unity, which are given by:



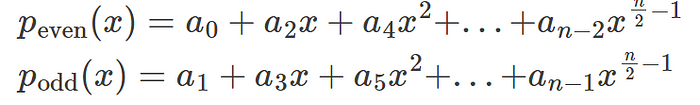
Thus we can write p and q for the k-th root of unity as:



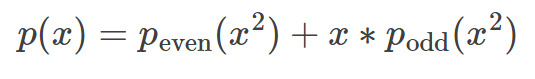
One intereseting property of the (2n) roots of unity is that:



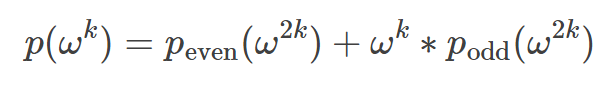
Similar to our previous DFT post, let us define 2 polynomials p\_even and p\_odd as follows, this is assuming that n is even.



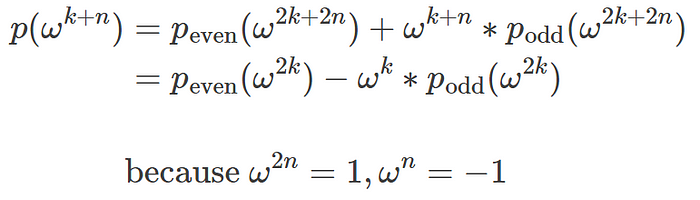
Then we can write p(x) using the above 2 polynomials as:



In terms of the (2n) roots of unity, for the k-th root, we have:



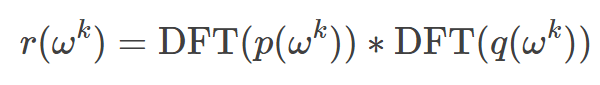
Similarly for the (k+n)-th root, we have:



Thus, once we compute p\_even(w²k) and p\_odd(w²k) for k=0 to n-1, we can compute p(w^k) for k = 0 to n-1, also we can re-use the same p\_even and p\_odd values to compute p(w^k) for k = n to 2n-1.

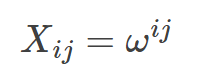
*This is same as computing the DFT of p(x) using Fast Fourier Transform. Similarly for q(x) we evaluate the DFT of q(x) for x equals to all the (2n) roots of unity.*

Thus, we can compute the values of r(x) which is the product of p(x) and q(x) for each x equals to the 2n roots of unity.

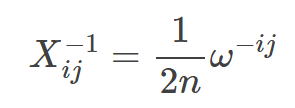


*Computing the R vector has****complexity of O(nlogn)****because both the DFTs has a complexity of O(nlogn).*

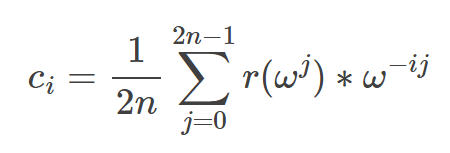
Using the 2n roots of unity, the X matrix is such that:



The good thing about this matrix is that the inverse of X is such that:



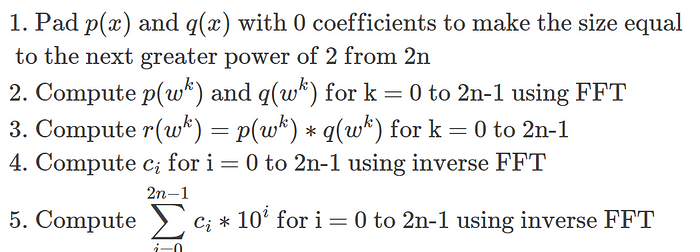
Then we can calculate the C vector as follows:



This is known as the **Inverse Fourier Transform**.

We can compute the inverse fourier transform using the same FFT algorithm in O(nlogn) time complexity. Instead of the coefficients a in p(x) or b in q(x) we are using the computed values r(w^j).

Here is the algorithm:



Here is the updated FFT code from the previous post.

def fft(poly, omega):  
 n = len(poly)  
   
 if n == 1:  
 return [poly[0], poly[0]]  
  
 even\_fft = fft(poly[::2], omega\*omega)  
 odd\_fft = fft(poly[1::2], omega\*omega)  
  
 m = int(n/2)  
 result = [0]\*2\*m  
 h = 1  
 for i in range(m):  
 result[i] = (even\_fft[i] + h\*odd\_fft[i])  
 result[i+m] = (even\_fft[i] - h\*odd\_fft[i])  
 h \*= omega  
   
 return result

Note that, in this implementation we are passing w as a paratemeter to the recursion because each sub-recursion uses w².

Python implementation for multiplying 2 integers using the FFT algoritm.

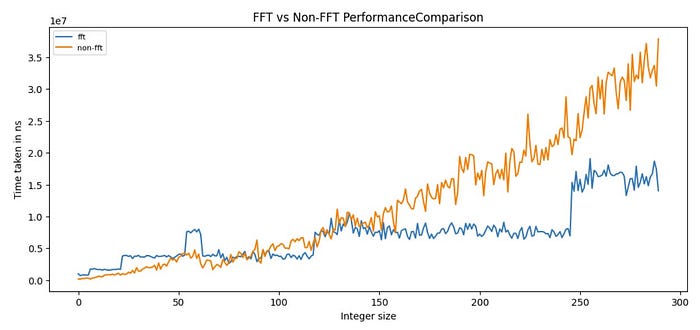
def get\_multiplication\_coefficients(p\_coefficients, q\_coefficients,   
 round\_decimal=0):  
  
 n, m = len(p\_coefficients), len(q\_coefficients)  
 g = n + m  
   
 # Pad p and q with enough 0s to make them equal to nearest power 2.  
 # The nearest power of 2 is computed w.r.t. the sum of the lengths   
 # of p and q  
 if g & (g-1) != 0:  
 b = int(math.log(g, 2))+1  
 g = (1 << b)  
   
 p\_coefficients += [0]\*(g-n)  
 q\_coefficients += [0]\*(g-m)  
  
 omega = np.exp(2j\*np.pi/g)  
 omega\_inv = np.exp(-2j\*np.pi/g)  
  
 # Compute DFT of p and q  
 p\_dft = fft(p\_coefficients, omega)  
 q\_dft = fft(q\_coefficients, omega)  
  
 # Compute the R vector  
 r = [p\_dft[i]\*q\_dft[i] for i in range(g)]  
   
 # Compute the inverse fourier transform using R vector  
 r\_idft = fft(r, omega\_inv)  
  
 # Final coefficient vector.  
 # Round the coefficients to nearest integer since for integer  
 # multiplication, coefficients must be integers.  
 c = [round(x.real/g, round\_decimal) for x in r\_idft]  
 return c  
  
def fft\_multiply(p, q):  
 sign = (p >= 0 and q >= 0) or (p < 0 and q < 0)  
   
 p, q = abs(p), abs(q)  
   
 p\_coefficients = get\_coefficients(p)  
 q\_coefficients = get\_coefficients(q)  
  
 c = get\_multiplication\_coefficients(p\_coefficients, q\_coefficients)  
  
 h = 1  
 result = 0  
 for i in range(len(c)):  
 result += c[i]\*h  
 h \*= 10  
   
 return result if sign else -result  
  
p, q = 1234, 6789  
assert fft\_multiply(p, q) == p\*q

The step where we round of the coefficients or the C vector to the nearest integer is an important step after inverse fourier transform, due to the following reasons.

1. In **integer multiplication, the coefficients must be integers**.
2. The coefficients without rounding off can have errors in the range of 10^-10 to 10^-15, and when we multiply these errors with power of 10s in the last step, if the power is greater than say 15 i.e. c[15]\*10¹⁵ then these errors can contribute significantly to the final value, making the final result significantly different from actual result.

Doing a performance benchmark of multiplication without using FFT and using FFT, we can see that the FFT algorithm starts to do better consistently for integers above 10¹⁵⁰.

The x-axis scales as 10^x in the below diagram.



import random  
  
for n in range(10, 300, 1):  
 p = random.randint(10\*\*n, 10\*\*(n+1))  
 q = random.randint(10\*\*n, 10\*\*(n+1))  
  
 print(n, p, q)  
  
 %timeit -n 10 -r 10 fft\_multiply(p, q)  
 %timeit -n 10 -r 10 multiply(p, q)  
 print()

*Also note that in the FFT algorithm, we have a requirement that the size of p + size of q should be equal to power of 2 and hence whenever the size of p + q is not a power of 2, it will pad with 0s to make it equal to the next power of 2.*

For e.g. if size of p = 16 and size of q = 15, then total size is 31 and the next greater power of 2 is 32. Thus it will pad p with 16 0s and q with 17 0s. But if size of q = 17, then total size is 33 and the next greater power of 2 is 64, thus it will have to pad p with 48 0s and q with 47 0s. Thus whenever size of p + size of q is 1 more than a power of 2, the time required becomes almost double as size requirement becomes double.

Let’s look at another problem.

**Combination Sums**

Given 2 integer arrays P and Q, consisting of positive integers, for each unique sum p+q where p is from P and q is from Q, find their frequencies.

For e.g. P=[1,2,3,4], Q=[3,1,5,4], then all possible p+q’s are :

1+3=4  
1+1=2  
1+5=6  
1+4=5  
2+3=5  
2+1=3  
2+5=7  
2+4=6  
3+3=6  
3+1=4  
3+5=8  
3+4=7  
4+3=7  
4+1=5  
4+5=9  
4+4=8  
  
The frequencies are:  
2:1  
3:1  
4:2  
5:3  
6:3  
7:3  
8:2  
9:1

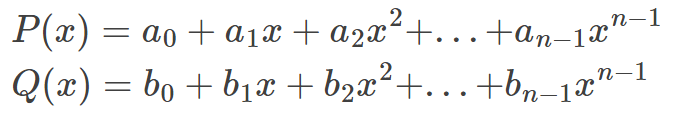
Straightforward approach is to sum each element of P with each element of Q, and use a hash table to count the frequencies. Time complexity is O(N²) where N is the size of P and Q.

Assuming that the maximum value of any element in P and Q combined is M.

Then the maximum value of the sum p+q is 2M. Thus, the different sums are in the range 0 to 2M. We can create a hash table of counts of elements in array Q and then looping over P, for each sum S, find the count of S-p in the hash table, add the count to frequency of sum S.

Time complexity of this approach is O(N\*M).

Another approach is to consider 2 polynomials P(x) and Q(x) as follows:



where a\_i = count of element i in array P and b\_i = count of element i in array Q.

*When we multiple P(x) and Q(x), then the coefficients of x^j gives the number of possible ways to get the sum j (or frequency of sum j). As seen above, we can compute the coefficients of the multiplication in O(MlogM) time complexity using the FFT algorithm.*

Following is the python implementation for the FFT solution.

def combination\_sum(p, q):  
 n = max(p)  
   
 p\_coefficients = [0]\*(n+1)  
 for i in range(len(p)):  
 p\_coefficients[p[i]] += 1  
  
 m = max(q)  
   
 q\_coefficients = [0]\*(m+1)  
 for i in range(len(q)):  
 q\_coefficients[q[i]] += 1  
  
 return get\_multiplication\_coefficients(p\_coefficients, q\_coefficients)

This approach is optimal only when M = O(N). If say M=O(N²) then the FFT based approach has time complexity of O(N²logN) which is greater than O(N²).

Let’s look at a slightly different problem.

**Pattern Search In String**

Given a string S of length N, consisting of lowercase english alphabets a-z and a pattern T of length M, again consisting of lowercase english alphabets. Return all start indices in S where T is a substring of S.

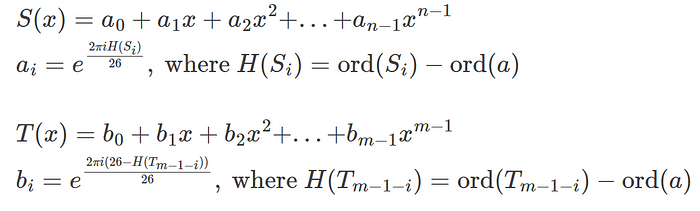
For e.g. if S = ‘mississippi‘ and T=’ssi’, then return [2, 5] because ‘ssi‘ occurs at indices 2 and 5 in S.

One straightforward approach is to use sliding window i.e. slide T over S and at each index U in S, check if T equals the substring S[U:U+M]. Time complexity is O(NM) because we are sliding over S of size N and each window is of size M.

The best algorithm known for this problem is the **KMP (Knuth-Morris-Pratt) algorithm** which solves the problem in **O(N+M) time complexity**. But we will anyways show a more optimized method compared to the sliding window approach but less optimized than KMP using the FFT algorithm.

Let S(x) be a polynomial of degree N-1 and T(x) be a polynomial of degree M-1.

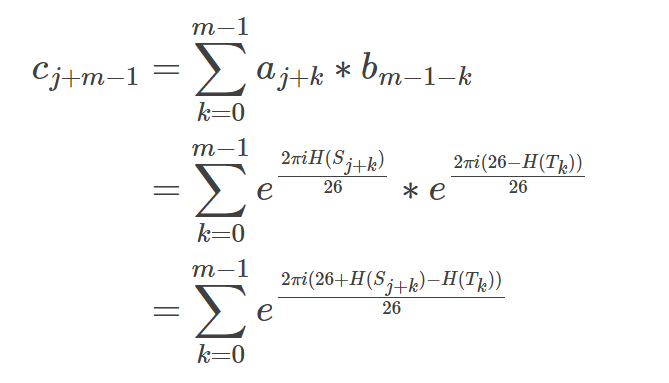
The coefficients are as follows:



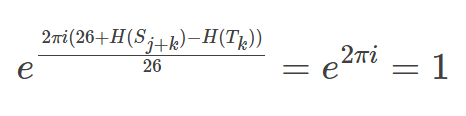
In the above ord() function calculates the ascii codes for the letters. For e.g. ord(a) = 97. We are scaling the codes in the range 0 to 25 from 97 to 122 using the function H().

*The coefficients b are such that the 1st coefficient b0 corresponds to the last letter in T, b1 corresponds to the 2nd last letter and so on. This is practically using the****sequence T in reverse****.*

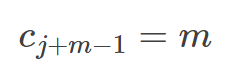
Now when we multiply S(x) with T(x), the coefficient of x^(j+m-1) is calculated to be:



If S[j+k] = T[k] in the above equation for some k i.e. the k-th letter in T is equal to the j+k th letter in S, we have:



If for all k=0 to m-1, S[j+k] = T[k] i.e. T is a substring of S, then we have:



Thus we multiply S(x) and T(x) using FFT and find the coefficient vector C using the inverse FFT and for all indices c\_(j+m-1) if it is equal to m, then the substring S[j:j+m] is equal to T and j is a staring index for a match. Find all such j.

Here is the python code:

def pattern\_search(s, pat):  
 p\_coefficients = []  
   
 for x in s:  
 h = ord(x)-ord('a')  
 p\_coefficients += [np.exp(2j\*np.pi\*h/26)]  
  
 q\_coefficients = []  
 for x in pat:  
 h = ord(x) - ord('a')  
 q\_coefficients = [np.exp(2j\*np.pi\*(26-h)/26)] + q\_coefficients  
  
 c = get\_multiplication\_coefficients(p\_coefficients, q\_coefficients,   
 round\_decimal=5)  
   
 output = []  
 for j in range(len(c)):  
 if j+len(pat)-1 < len(c) and c[j+len(pat)-1] == len(pat):   
 output += [j]  
 return output

Note that we are not rounding c values to the nearest integer because here **c need not be integers for all**, instead we are rounding c to the 5th decimal point just in case there are rounding off errors. For e.g. m = 5 but c=4.9999996.

***The time complexity of the above approach is O((N+M)log(N+M)).***

Similarly, we can compute the convolution of one vector with another using FFT. Recall that convolution is a very important operation in deep learning systems such as CNN networks.

Hence efficiently computing them is key to fast training and inference.

def convolution(p, q):  
 q = q[::-1]  
 c = get\_multiplication\_coefficients(p, q, round\_decimal=5)  
   
 return c

But in deep learning, we mostly use 2d convolution. To achieve 2d convolution, we must be able to do 2d FFT operations.

2d FFT is intuitively very simple: For each row of the 2d matrix, do 1d FFT. Then for each column do 1d FFT. Here is a python implementation for the same:

def fft\_2d(arr, omega\_x, omega\_y):  
 # FFT on each row of 2d matrix  
 new\_arr = []  
 for x in arr:  
 new\_arr += [fft(x, omega\_x)]  
  
 # Transpose the matrix from previous step  
 new\_arr\_t = [[0]\*len(new\_arr) for j in range(len(new\_arr[0]))]  
   
 for i in range(len(new\_arr\_t)):  
 for j in range(len(new\_arr\_t[0])):  
 new\_arr\_t[i][j] = new\_arr[j][i]  
  
 # Do FFT on each row of transposed matrix same as doing FFT on each   
 # column of original matrix.  
 new\_arr = []  
 for x in new\_arr\_t:  
 new\_arr += [fft(x, omega\_y)]  
  
 return new\_arr

The code for convolution with 2d matrices is updated as follows:

def convolution\_2d(p, q):  
 # p matrix has shape n1 x m1 and q matrix n2 x m2  
 n1, n2 = len(p), len(q)  
 m1, m2 = len(p[0]), len(q[0])  
  
 # 'Reverse' the q matrix as follows  
 q\_new = [[0]\*m2 for i in range(n2)]  
   
 for i in range(n2):  
 for j in range(m2):  
 q\_new[i][j] = q[n2-1-i][m2-1-j]  
  
 # Pad each row with 0s  
 g1 = m1 + m2  
   
 if g1 & (g1-1) != 0:  
 b = int(math.log(g1, 2))+1  
 g1 = (1 << b)  
  
 for i in range(n1):  
 p[i] += [0]\*(g1-m1)  
  
 for i in range(n2):  
 q\_new[i] += [0]\*(g1-m2)  
  
 # Pad each column with 0s  
 g2 = n1 + n2  
   
 if g2 & (g2-1) != 0:  
 b = int(math.log(g2, 2))+1  
 g2 = (1 << b)  
  
 p += [[0]\*g1 for i in range(g2-n1)]  
 q\_new += [[0]\*g1 for i in range(g2-n2)]  
  
 # Define the omega's for row and columns separately  
 omega\_x = np.exp(2j\*np.pi/g1)  
 omega\_x\_inv = np.exp(-2j\*np.pi/g1)  
  
 omega\_y = np.exp(2j\*np.pi/g2)  
 omega\_y\_inv = np.exp(-2j\*np.pi/g2)  
  
 # Perform 2d FFT on p and q  
 p\_dft = fft\_2d(p, omega\_x, omega\_y)  
 q\_dft = fft\_2d(q\_new, omega\_x, omega\_y)  
  
 # Do elementwise multiplication to get the R matrix  
 r = [[0]\*g1 for i in range(g2)]  
  
 for i in range(g2):  
 for j in range(g1):  
 r[i][j] = p\_dft[i][j]\*q\_dft[i][j]  
   
 # Do 2d inverse fourier transform on r  
 r\_idft = fft\_2d(r, omega\_x\_inv, omega\_y\_inv)  
  
 # Coefficient matrix  
 c = [[0]\*g1 for i in range(g2)]  
   
 for i in range(g2):  
 for j in range(g1):  
 x = r\_idft[i][j].real  
 c[i][j] = round(x/(g1\*g2), 5)  
  
 return c

*As you can see, that the problems involving FFT have a pattern. For any problem which involves****finding pairs (i, j) where i+j is some constant****or we can convert a problem into this form, we can apply FFT to solve those.*

I will conclude this post here. There are many similar algorithmic problems related to FFT you can find [here](https://cp-algorithms.com/algebra/fft.html). In the next post, I will dive into the optimization of the core FFT algorithm to make efficient use of CPU, memory and cache.