15-451: Algorithm Design and Analysis, Carnegie Mellon University

Applications the FFT and Convolutions

1 Preliminaries

The Fast Fourier Transform (FFT) is a very important algorithm. It plays a key role in many fields, including digital signal processing, digital photography, video processing, x-ray tomography, physics simulations, high-precision arithmetic, string matching, evalating generating functions for counting things, and more. It's a useful tool to have in your algorithms tool box.

This lecture is about what the FFT computes, and how to apply it. In a subsequent lecture we will describe and analyze the FFT algorithm.

The computation done by the FFT is most easily understood in the context of the problem of multiplying polynomials[1]. So let $A(x)$ and $B(x)$ be two polynomials of degree d

$$\begin{aligned} A(x) &= a_0 + a_1 x + a_2 x^2 + \cdots + a_d x^d \\ B(x) &= b_0 + b_1 x + b_2 x^2 + \cdots + b_d x^d \end{aligned}$$

Let $C(x)$ be the polynomial that is the product of $A(x)$ and $B(x)$. The coefficients of $C(x)$ are $c_0, \ldots c_{2d}$, so

$$C(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_{2d} x^{2d}.$$

These coefficients can be expressed in the following summation:

$$c_k = \sum_{\substack{0 \le i,j \le d \\ i+j=k}} a_i * b_j = \sum_{0 \le i \le k} a_i * b_{k-i}$$

If we think of A and B as vectors of numbers, then the C-vector is called the *convolution* of A and B. (In the latter sum we have assumed that the coefficients of A and B are padded with zero coefficients up to degree $2d$. We shall do this throughout the lecture whenever convenient.)

So, given vectors $A = [a_0, \ldots, a_d]$ and $B = [b_0, \ldots, b_d]$ we want to compute $C = [c_0, \ldots, c_{2d}]$ where c_i is defined as above.

This can be done naively in $O(n^2)$. The FFT can be used to do it in $O(n \log n)$.

In the rest of this lecture we present some interesting ways that we can make use of convolutions.

[1] These polynomials can be over the field of complex numbers, or the finite field of integers over a properly chosen prime modulus. The latter method avoids the issue of round-off errors, and is particularly important in the applications of the FFT to discrete algorithms.

2 Multiplying Large Numbers

Suppose we have two n-bit numbers P and Q, written in binary. We want to compute the product of P and Q, which will have at most $2n$ bits. The brute-force algorithm will take $O(n^2)$ time.

Karatsuba's algorithm is divide and conquer. It splits P and Q each into two $n/2$ bit numbers, then does three multiplications (recursively) of the $n/2$ bit numbers, then combines them together for the answer in another $O(n)$ time. Thus the recurrence is:

$$T(n) = 3T(n/2) + n$$

which solves to $T(n) = O(n^{\log_2 3}) = O(n^{1.584\cdots})$.

Here we present an algorithm based on the FFT which runs in time $O(n \log n)$.[2] [3] To see how this is going to work, let's write the two input numbers as follows:

$$P = \sum_{0 \le i < n} 2^i a_i \qquad Q = \sum_{0 \le i < n} 2^i b_i$$

Now, setting $d = n - 1$, we see that

$$P = A(2) \qquad Q = B(2)$$

where $A(x)$ and $B(x)$ are the two polynomials defined above. It immediately follows that

$$C(2) = A(2) * B(2) = P * Q$$

The coefficients of the polynomial C can be computed in $O(n \log n)$ time. And we have

$$C(2) = 2^0 c_0 + 2^1 c_1 x + \cdots + 2^{2n-2} c_{2n-2} = P * Q$$

The numbers c_i are integers in the range $[0, n]$. These can therefore be represented in binary with $1 + \lfloor \log_2 n \rfloor$ bits. So let's assume we can do arithmetic on these numbers in $O(1)$ time. The remaining problem is to convert these c_i into bits to represent the same number as $C(2)$.

The following simple algorithm does this conversion.

[2]Here we use the usual model of computing from this course, where if you have problem where the input is of size n you can assume that you can do arithmetic with numbers of $\log_2 n$ bits in constant time. If you count the complexity in bits then you can achieve a running time of $O(n \log n \log\log n)$. This is achieved by the Schönhage-Strassen algorithm.

[3]The three algorithms mentioned above (as well as some others) are all used in the GMP multi-precision arithmetic package. For the largest numbers an FFT-based algorithm is used. See `https://gmplib.org/manual/Multiplication-Algorithms`.

Algorithm: Convert $c_0 \ldots c_{2n-2}$ to Binary

```
$\mathbf{AC} \leftarrow 0$
for $i = 0$ to $2n-2$ do
    $\mathbf{AC} \leftarrow \mathbf{AC} + c_i$
    $c_i \leftarrow \mathbf{AC} \bmod 2$
    $\mathbf{AC} \leftarrow \lfloor \mathbf{AC}/2 \rfloor$
```

The invariant that holds at the beginning of the loop is:

$$2^i \mathbf{AC} + \sum_{0 \le j \le 2n-2} 2^j c_j = P * Q$$

This invariant follows inductively from the fact that

$$\mathbf{AC} + c_i = 2\mathbf{AC}' + c_i'$$

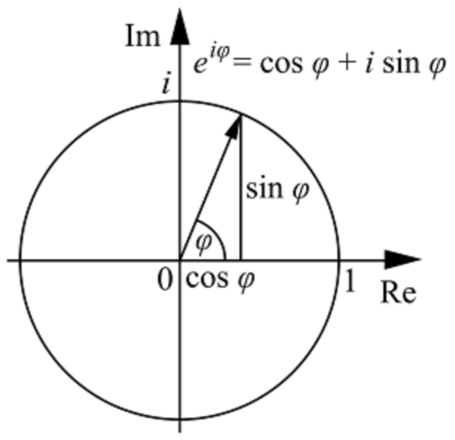
where $\mathbf{AC}$ and c_i represent those variables at the beginning of the loop and $\mathbf{AC}'$ and c_i' represent those variables at the end of the loop. When the algorithm terminates $\mathbf{AC}$ must be zero, because at most $2n-2$ bits are required to represent the answer.

At the end each of the c_i variables is either 0 or 1, and $\mathbf{AC}$ is zero. It follows from the invariant that this is the correct binary representation of the product $P * Q$.

3 String Matching

Suppose we are given a pattern $P = p_0 p_1 \ldots p_{m-1}$ and a text $T = t_0 t_1 \ldots t_{n-1}$ (with $m < n$). All the characters of T come from an alphabet $\Sigma = \{0, 1, \ldots, \ell - 1\}$. The characters of P come from $\Sigma \cup *$, where $*$ is a wild-card character which matches any character of Σ. We can find all occurrences of P in T in $O(n \log n)$ time using the convolution.

The solution presented here is based on the use of complex numbers on the unit circle.



The figure above shows where the point $e^{i\phi}$ is on the unit circle. (The symbol $i = \sqrt{-1}$ is the imaginary unit.) If we have the two points on the circle, $e^{i\alpha}$ and $e^{i\beta}$ ($-\pi < \alpha, \beta < \pi$) and we multiply them together we get:

$$e^{i\alpha} * e^{i\beta} = e^{i(\alpha+\beta)}$$

It follows that $e^{i\alpha} * e^{i\beta} = 1$ if and only if $\alpha = -\beta$.

Now, to encode the problem of string matching we will rewrite the pattern and the text as sequence of complex numbers. Here's how we rewrite the pattern. For all $0 \le j \le n-1$ we have:

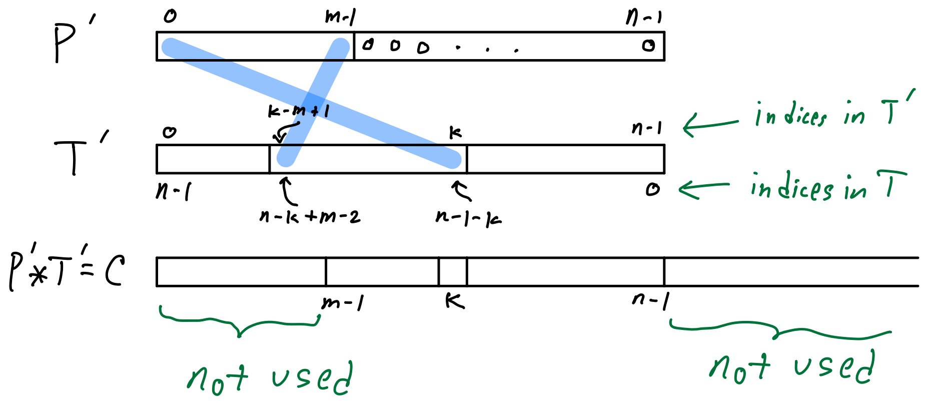
$$p'_j = \begin{cases} 0 & \text{if } j \ge m \\ 0 & \text{if } p_j = * \\ \exp(i\pi \frac{p_j}{\ell}) & \text{otherwise} \end{cases}$$

For the text, we use the following replacement for $0 \le j \le n-1$:

$$t'_j = \exp(-i\pi \frac{t_{n-1-j}}{\ell})$$

Notice the way in which the order of the characters has been reversed. Now we take the convolution of the p' vector and the t' vector. The kth element (for $m-1 \le k \le n-1$) of the convolution c_k is as follows:

$$\begin{aligned} c_k &= \sum_{h+j=k} p'_h * t'_j = \sum_{0\le h\le m-1} p'_h * t'_{k-h} \\ &= \sum_{0\le h\le m-1} \exp(i\pi \frac{p_h}{\ell}) * \exp(-i\pi \frac{t_{n-1-k+h}}{\ell}) \end{aligned}$$



Let w be the number of wild-card characters in P. We're interested in the convolution elements for $m-1 \le k \le n-1$. For these values we have $c_k = m - w$ if and only if the string P matches the string $[t_{n-1-k} \dots t_{n-k+m-2}]$. Why? Each wild-card of P produces 0 in the convolution. So if

$c_k = m - w$ it means that all the non-wild-card characters of P must be matching a character of T. Because when the characters match it contributes a 1 to the real part of that element of the convolution. If there is a mismatch, it contributes something less than 1 to the real part. So if there is a missmatch, then the real part of that element of the convolution will be strictly less than $m - w$.

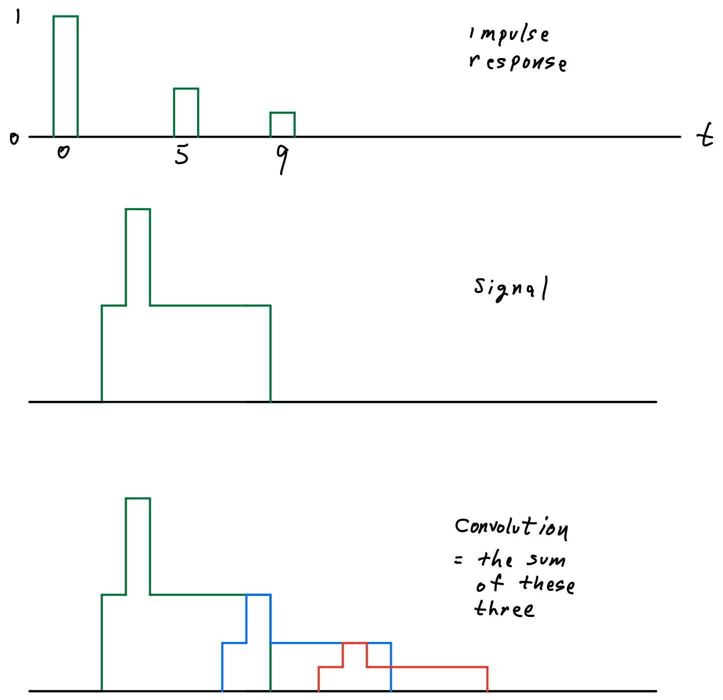
4 Digital Filtering

Suppose you wanted to simulate what it would sound like to be in a dungeon (or concert hall, or a forest, or anywhere else). You want to take a recorded sound and transform it to what it sounds like in some other environment. Here's one way to go about doing that.

You could go to the dungeon and measure what is called an *impulse response* of the environment. This is done by generating a loud very short bang (the impulse) and recording for the next several seconds what comes back (the response) as a result of that impulse.

The input to this problem is a signal of length n, which is a sequence of numbers, $S = s_0, s_1, \ldots, s_{n-1}$. Let the response also be a sequence of n numbers (pad it out if it's too short), and call it $R = r_0, r_1, \ldots, r_{n-1}$. Now the convolution of the two sequences is

$$c_k = \sum_{\substack{0 \le i,j \le k \\ i+j=k}} s_i * r_j$$



Making a practial digital filter for applications is much more complicated. But at the core of these algorithms is an FFT-based convolution algorithm. This is needed to make it run fast enough.

5 Three Evenly Spaced 1s within a Binary String

Given a binary string S of length n, let $S[i]$ denote the ith bit, for $0 \le i < n$. And let $L = \{i \mid S[i] = 1\}$, be the indices with ones. For this input we have to decide whether there are three evenly spaced 1s within S (we don't need to find the locations of the 1s should they exist). For example, 11100000, 110110010 both have three evenly spaced 1s, while 1011 does not.

1. What is a naive $O(n^2)$ solution to this problem? Just try all starting indices i and all gap lengths d, and for each (i,d) pair, check in $O(1)$ if $S[i] = S[i+d] = S[i+2d] = 1$. There are at most n choices for i and for d, so total runtime is $O(n^2)$.

2. Here's the $O(n \log n)$ solution using convolutions. Construct a polynomial $p(x) = \sum_{i=0}^{n-1} S[i]x^i$. So for the second example above this is:

$$p(x) = x^0 + x^1 + x^3 + x^4 + x^7.$$

   This step is $O(n)$.

   Compute $p(x)^2$ using FFT multiplication. For the example above:

$$p(x)^2 = x^0 + 2x^1 + x^2 + 2x^3 + 4x^4 + 2x^5 + x^6 + 4x^7 + 3x^8 + 2x^{10} + 2x^{11} + x^{14}.$$

   The coefficient of x^t is the number of ordered pairs (a,c) such that $a,c \in L$ and $a+c = t$. This step is $O(n \log n)$.

   Now we look at the terms x^{2b} for $b \in L$. Note that the pair (b,b) always contributes 1 to the coefficient of x^{2b}. If there exists some other pair (a,c) where $a \neq c$, $a+c = 2b$, and $a,c \in L$, then the coefficient of x^{2b} will be at least 3. Thus if the coefficient of x^{2b} is 3 or more for any $b \in L$ then output YES. Otherwise, output NO. In the example above, $4 \in L$ we see that the coefficient of x^8 is 3 implying that there exists three evenly spaced 1s, centered at 4.