CS 170 Handout: FFT in the Wild

By Jackie Lian, Jonathan Pei, Lance Mathias

Contents

1 Introduction

In lecture, you learned about using FFT as a subroutine to speedup polynomial multiplication from $O(n^2)$ to $O(n \log n)$ time[1]. Using polynomial multiplication (via FFT) as a black-box, we can do a lot of other cool stuff!

In this class, the procedure for FFT applications generally follows the outline below:

1. Construct two polynomials p, q whose product r gives useful information for solving the problem. Depending on the problem, the construction is performed by encoding information from the problem into either the coefficients or exponents of both polynomials.
2. Compute $r(x) = (p \cdot q)(x)$ via FFT (as a black-box).
3. Use the coefficients and/or exponents of r to solve the problem.

Then, full proofs of correctness involve arguing *why* the product polynomial r gives us useful information for solving the problem, and usually involve some manipulation of mathematical sums.

In this class, we apply FFT to solve the following types of problems:

- **Variations of k-SUM (2)**

  Example problems: triple sum (dis03), ways to get cups, adding coins (sp19 mt1 q12).

- **Convolutions (3)**

  Example problems: legoland (sp23 mt1 q9).

- **Cross correlation / Shifted Dot Product (4)**

  Example problems: string matching (dis03), counting k-inversions (hw03), ice-cream loving PNPenguins (hw03).

We will now walk through each application type.

[1] n is the max degree between the two polynomials being multiplied.

2 Variations of k-SUM

Main Idea: All problems of this type involve computing the number of ways to achieve some sum s, given a collection of elements with potential capacity constraints.

Common Strategy: Construct polynomial(s) by encoding the elements the exponents. Then, use FFT to multiply the polynomials to yield a product polynomial whose terms ax^b encode the following information: a is the number of ways to combine elements to achieve sum b.

2.1 Example Problem 1: Triple Sum

We are given an array $A[0, \dots, n-1]$ with n integers in the range $[0, n-1]$ (not necessarily all distinct!), and a non-negative integer s. We would like to know if there exist indices $0 \le i, j, k \le n-1$ (not necessarily distinct) such that

$$A[i] + A[j] + A[k] = s$$

(a) First, let us consider 2SUM, a simplified version of triple sum where you determine if there exist indices $0 \le i, j \le n-1$ (not necessarily distinct) such that

$$A[i] + A[j] = s$$

Suppose we're given the array $[1, 3, 5]$ and $n = 5$. What are all the possible 2SUMs?

Solution: We compute all possible 2SUMs below:

$$\begin{aligned} 1 + 1 &= 2 \\ 1 + 3 &= 4 \\ 3 + 1 &= 4 \\ 3 + 3 &= 6 \\ 1 + 5 &= 6 \\ 5 + 1 &= 6 \\ 3 + 5 &= 8 \\ 5 + 3 &= 8 \\ 5 + 5 &= 10 \end{aligned}$$

(b) Now try encoding the above array into a polynomial to solve 2SUM with one polynomial multiplication. Then, how would you encode an arbitrary array to solve 2SUM?

Hint: given $p(x) = x^1 + x^3 + x^5$, compute $p(x)^2$. What are the resulting coefficients and exponents in the product? Can they be used to solve 2SUM?

Solution: Using the hint, we have

$$p(x)^2 = (x^1 + x^3 + x^5)(x^1 + x^3 + x^5) = x^2 + 2x^4 + 3x^6 + 2x^8 + x^{10}$$

Comparing our resulting polynomial $p(x)^2$ to the 2SUMs computed in part (a), we see that $\{2, 4, 6, 8, 10\}$ show up as exponents, and the coefficient corresponding to each exponent is exactly the number of ways to achieve that 2SUM!

Thus, to solve 2SUM, we can follow this procedure:

(i) Encode the array A into a polynomial by setting all its elements as exponents:

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]} = \sum_{i=0}^{n-1} x^{A[i]}.$$

(ii) Use FFT to multiply $p(x)$ with itself to yield:

$$\begin{aligned} p(x)^2 &= \left(\sum_{i=0}^{n-1} x^{A[i]}\right)\left(\sum_{i=0}^{n-1} x^{A[i]}\right) \\ &= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} x^{A[i]}x^{A[j]} \\ &= \sum_{0\le i,j\le n-1} x^{A[i]+A[j]} \end{aligned}$$

(iii) Finally, to check whether or not s exists as a 2SUM for A, we simply need to check whether the term with x^s has a non-zero coefficient.

(c) Now, design an $\mathcal{O}(n \log n)$ time algorithm for triple sum. Note that you do not need to actually return the indices; just yes or no is enough.

Food for thought: is it possible to return the number of ways you can add 3 elements from A to equal n?

Solution: *Key: exponentiation converts multiplication to addition!*

Main idea.

Using similar idea to 2SUM, we define

$$p(x) = x^{A[0]} + x^{A[1]} + \cdots + x^{A[n-1]}.$$

Notice that $p(x)^3$ contains a sum of terms, where each term has the form

$$x^{A[i]} \cdot x^{A[j]} \cdot x^{A[k]} = x^{A[i]+A[j]+A[k]}.$$

Therefore, we just need to check whether $p(x)^3$ contains x^s as a term.

Proof of Correctness. Observe that

$$\begin{aligned} q(x) = p(x)^3 = \left(\sum_{0\le i<n} x^{A[i]}\right)^3 &= \left(\sum_{0\le i<n} x^{A[i]}\right)\cdot\left(\sum_{0\le j<n} x^{A[j]}\right)\cdot\left(\sum_{0\le k<n} x^{A[k]}\right) \\ = \sum_{0\le i,j,k<n} x^{A[i]}x^{A[j]}x^{A[k]} &= \sum_{0\le i,j,k<n} x^{A[i]+A[j]+A[k]}. \end{aligned}$$

Therefore, the coefficient of x^s in q is nonzero if and only if there exist indices i, j, k such that $A[i] + A[j] + A[k] = s$. Hence, the algorithm is correct.

Also, building off of what we explored in part (b), the coefficient of x^s also tells us exactly *how many* such triples (i, j, k) satisfy $A[i] + A[j] + A[k] = s$.

Runtime Analysis. Constructing $p(x)$ clearly takes $O(n)$ time. $p(x)$ is a polynomial of degree at most $n = O(n)$. Therefore doing the two multiplications to compute $q(x)$ takes $O(n \log n)$ time with the FFT. Finally, looking up the coefficient of x^s takes constant time, so overall the algorithm takes $O(n \log n)$ time.

2.2 Example Problem 2: Ways to Get Cups

You want to buy a total of s cups. The store sells cups in packages of integer size c_1, c_2, c_3, and c_4, all in the range $[1, s]$. The store, due to demand on cups, also places a restriction that you can only buy p of each package. Describe an efficient algorithm that allows you to compute the number of ways you can purchase exactly s cups, and analyze its runtime.

Solution:

Intuition. At a glance, this problem may seem to be quite different from Triple Sum—in Triple Sum you add together a fixed 3 elements from the array, while here the number of packages you can buy in each size can be varied. However, the end goal is still the same: you have a bunch of elements and want to figure out how to combine those elements to achieve some sum.

In this problem, to account for the varied number of cups you can use in each size, you can reframe the problem as follows:

> You are given four different integer arrays, each array containing all the possible number of cups that can be bought using a given package size, i.e.
>
> $$\begin{aligned} A_1 &= [0, c_1, 2c_1, \ldots, pc_1] \\ A_2 &= [0, c_2, 2c_2, \ldots, pc_2] \\ A_3 &= [0, c_3, 2c_3, \ldots, pc_2] \\ A_4 &= [0, c_4, 2c_4, \ldots, pc_4] \end{aligned}$$
>
> Determine the number ways you can choose indices $0 \leq i, j, k, \ell \leq n-1$ such that
>
> $$A_1[i] + A_2[j] + A_3[k] + A_4[\ell] = s.$$

Try to convince yourself that this is a valid way of rephrasing the problem. When framed in this way, it is easy to see that the problem is actually 4SUM in disguise!

Main Idea: We construct 4 different polynomials as follows:

$$f_1(x) = \sum_{i=0}^{p} x^{i \cdot c_1} = x^0 + x^{c_1} + x^{2c_1} + \cdots + x^{pc_1}$$
$$f_2(x) = \sum_{i=0}^{p} x^{i \cdot c_2} = x^0 + x^{c_2} + x^{2c_2} + \cdots + x^{pc_2}$$
$$f_3(x) = \sum_{i=0}^{p} x^{i \cdot c_3} = x^0 + x^{c_3} + x^{2c_3} + \cdots + x^{pc_3}$$
$$f_4(x) = \sum_{i=0}^{p} x^{i \cdot c_1} = x^0 + x^{c_4} + x^{2c_4} + \cdots + x^{pc_4}$$

where the exponent in polynomial f_α represents the possible number of cups you can buy with a package of size c_α. Now, we can run the FFT polynomial multiplication algorithm 3 times to obtain the final polynomial $q(x)$:

$$f_{12}(x) = (f_1 \cdot f_2)(x)$$
$$f_{34}(x) = (f_3 \cdot f_4)(x)$$
$$q(x) = f_{1234}(x) = (f_{12} \cdot f_{34})(x)$$

Note: it's completely valid to multiply them in a different order like $(f_1, f_2), (f_{12}, f_3), (f_{123}, f_4)$; the asymptotic runtime ends up being the same anyways.

Now, to get the number of ways to purchase exactly s cups, we simply retrieve the coefficient of the term with x^s in $q(x)$!

Proof of Correctness: We compute q as follows:

$$\begin{aligned} q(x) = (f_1 \cdot f_2 \cdot f_3 \cdot f_4)(x) &= \left(\sum_{i=0}^{p} x^{i \cdot c_1}\right)\left(\sum_{i=0}^{p} x^{i \cdot c_2}\right)\left(\sum_{i=0}^{p} x^{i \cdot c_3}\right)\left(\sum_{i=0}^{p} x^{i \cdot c_4}\right) \\ &= \sum_{0 \le i,j,k,\ell \le p} x^{i \cdot c_1} x^{j \cdot c_2} x^{k \cdot c_3} x^{\ell \cdot c_4} = \sum_{0 \le i,j,k,\ell \le p} x^{i \cdot c_1 + j \cdot c_2 + k \cdot c_3 + \ell \cdot c_4} \end{aligned}$$

Hence, the coefficient for a given term with x^s gives the number of ways to purchase exactly s cups, using up to p of each package type.

Runtime Analysis: Constructing polynomials takes $4 \cdot O(p) = O(p)$ time, and their degrees are at most $p \cdot \max(c_1, c_2, c_3, c_4) = ps$. Hence, performing three multiplications via FFT to compute $q(x)$ takes $O(ps \log(ps))$ time (why is this?). Finally, looking up the coefficient of x^s takes constant time, so the overall runtime of the algorithm is $O(ps \log(ps))$.

3 Convolution

Main Idea:

Note that the polynomial multiplication operation, at its core, is a convolution. To see why this is, consider multiplying two polynomials $p(x) = \sum_{i=0}^{m} a_i x^i$ and $q(x) = \sum_{j=0}^{n} b_j x^j$:

$$\begin{aligned} r(x) &= (p \cdot q)(x) \\ &= \left(\sum_{i=0}^{m} a_i x^i\right)\left(\sum_{j=0}^{n} b_j x^j\right) \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} a_i b_j x^i x^j \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} a_i b_{k-i} x^k && [k = i + j] \\ &= \sum_{k=0}^{m+n}\left(\sum_{i=0}^{k} a_i b_{k-i}\right) x^k \end{aligned}$$

Notice that the inside of the parentheses is the discrete convolution of the coefficients of p, q.

Hence, whenever you see a problem where you need to compute some form of

$$\sum_{i=0}^{k} a_i b_{k-i},$$

you should immediately think of polynomial multiplication (via FFT)!

Common Strategy:

Construct two polynomials by filling in their coefficients with some useful information from the problem. When doing this, you want to think about what things you want to multiply and then sum together; or alternatively, think about what a and b should represent in the summation $\sum_{i=0}^{k} a_i b_{k-i}$.

3.1 Example Problem: LegoLand

Legoland is open for $2n$ days in the summer, and visitors arrive at the park for the first n days. On day i, exactly a_i visitors are arrive at Legoland.

Visitors stay in Legoland for different lengths of time. More precisely, among visitors arriving on any given day, a p_t-fraction of visitors will leave after t-days at Legoland (i.e. they will spend t days at Legoland then leave the next day).

More formally, we are given the following input:

1. Number of arrivals $\{a_1, a_2, ... \, a_n\}$.
2. $\{p_1, p_2, \ldots, p_n\}$ where p_t is the fraction of visitors that will spend t days.

We want to design an algorithm to find the number of visitors leaving the park on each day.

Solution: Intuition: We believe that one of the best ways to approach this type of (opaque) problem is to start with some small examples and look for patterns. Even if you're able to recognize that the problem is probably an FFT application problem, don't jump straight into constructing polynomials because it's easy to lose focus along the way.

Thus, for this question where you're pretty sure you need to compute a series of shifted sums, but aren't sure exactly what you're supposed to sum up, we highly recommend thinking about what happens on day 1, day 2, etc.; start small and build up some ideas. Let's go through a little bit together:

> **How many people are leaving on day 1?** Well, no one is leaving on day 1 since everyone has to stay for at least one day (the p_i's start at p_1), and everyone there just arrived on that same day.
>
> **How many people are leaving on day 2?** On day 2, people who arrived on day 1 and only stay for 1 day will leave. Thus, the number of people leaving on day 2 is $a_1 p_1$.
>
> **How many people are leaving on day 3?** On day 3, people who arrived on day 1 and stay for 2 days will leave, and people who arrived on day 2 and stay for 1 day will also leave. Considering leaving visitors from both days 1 and 2, the total number of people leaving on day 3 is $a_1 p_2 + a_2 p_1$.
>
> **How many people are leaving on day 4?** Using the same logic as in previous, we can determine that the number of people leaving on day 4 is $a_1 p_3 + a_2 p_2 + a_3 p_1$.

Do we see a pattern here? If you keep on observing subsequent days, you'll be able to deduce that the number of people leaving on a given day d is

$$\sum_{1 \le i < d} a_i \cdot p_{d-i},$$

where we define a_i, p_i to be zero whenever $i > n$. *This sum looks just like the polynomial coefficient term we saw previously!*

Now that we've spotted the pattern, we are ready to dive into the construction of the polynomials.

Main Idea: We construct 2 polynomials as follows:

$$A(x) = \sum_{i=1}^{n} a_i x^i$$

$$P(x) = \sum_{j=1}^{n} p_j x^j$$

Then, we use FFT to multiply the two polynomials to yield

$$L(x) = (A \cdot P)(x) = \sum_{d=2}^{2n} c_d x^d,$$

where c_d represents the number of visitors leaving the park on day d! Thus, for our final answer we just return

$$[0, c_2, c_3, \ldots, c_{2n}].$$

Proof of Correctness: Observe that

$$\begin{aligned}
L(x) &= (A \cdot P)(x) && (1)\\
&= \left(\sum_{i=1}^{n} a_i x^i\right)\left(\sum_{j=1}^{n} p_j x^j\right) && (2)\\
&= \sum_{d=2}^{2n}\left(\sum_{i+j=d} a_i p_j\right) x^d && (3)\\
&= \sum_{d=2}^{2n}\left(\underbrace{\sum_{1 \le i < d} a_i \cdot p_{d-i}}_{c_d}\right) x^d && (4)
\end{aligned}$$

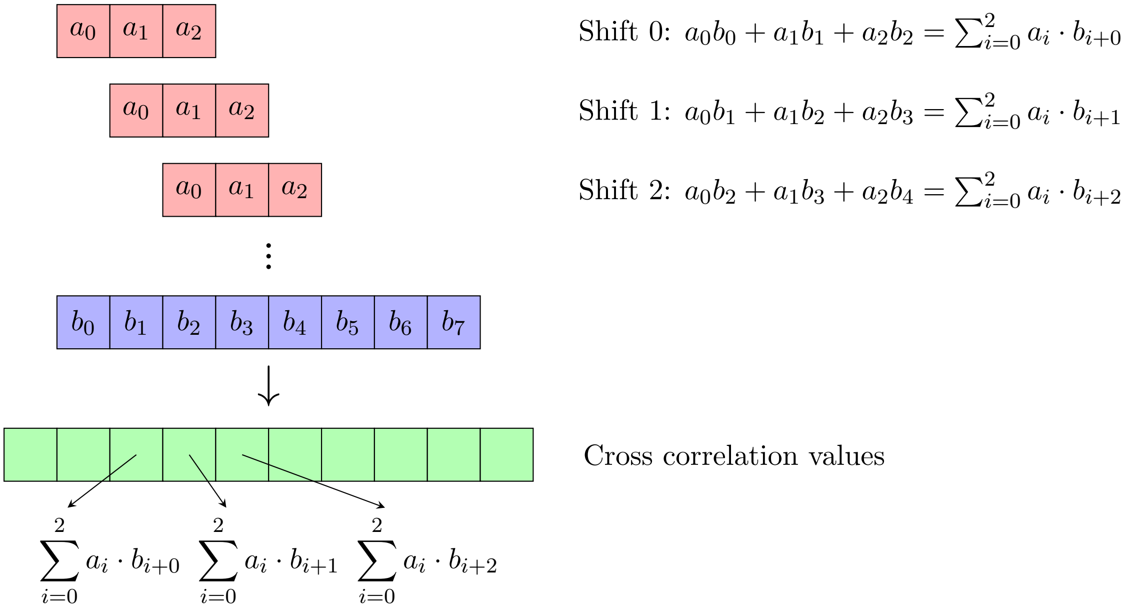
Hence, the coefficients correctly represent the number of visitors leaving on each day.

Runtime Analysis: Constructing both polynomials takes $2 \cdot O(n) = O(n)$ time, and both polynomials have degree n. Thus, multiplying them via FFT takes $O(n \log n)$ time. Finally, return all the coefficients takes $O(1)$ or $O(n)$ time, depending on how we implement array pointers. Thus, the overall runtime is $O(n \log n)$.

4 Cross Correlation

Main Idea:

Cross correlation is often referred to as a “sliding dot product” operation on two vectors a and b, which is an apt description: each coefficient in the output is of the form $\sum_{i=0}^{k} a_i b_{j+i}$, where k is the length of one of a, b. In Figure 1 below, we visualize computing the cross correlation of $a = [a_0, a_1, a_2]$ and $b = [b_0, \ldots, b_7]$:



Shift 0: $a_0b_0 + a_1b_1 + a_2b_2 = \sum_{i=0}^{2} a_i \cdot b_{i+0}$

Shift 1: $a_0b_1 + a_1b_2 + a_2b_3 = \sum_{i=0}^{2} a_i \cdot b_{i+1}$

Shift 2: $a_0b_2 + a_1b_3 + a_2b_4 = \sum_{i=0}^{2} a_i \cdot b_{i+2}$

Cross correlation values

$$\sum_{i=0}^{2} a_i \cdot b_{i+0} \quad \sum_{i=0}^{2} a_i \cdot b_{i+1} \quad \sum_{i=0}^{2} a_i \cdot b_{i+2}$$

Figure 1: Visual representation of Cross Correlation (“sliding dot product”)

To actually implement this type of operation, the key idea is to apply our intuition from convolution but reverse one of the polynomials so that instead of having coefficients of the form $\sum_{i=0}^{k} a_i \cdot b_{k-i}$, we will have something of the form $\sum_{i=0}^{k} a_i \cdot b_{k'+i}$. In other words, we can compute the cross correlation of two vectors a, b as follows:

1. Reverse b to yield b^R.
2. Generate the polynomials A and B^R from a and b^R, respectively, by treating them as coefficient vectors.
3. Use FFT to compute the product polynomial $C(x) = \left(A \cdot B^R\right)(x)$.
4. The coefficients of C are the cross-correlation of a and b!

We provide a proof of correctness for this method as follows:

Let us explicitly write out the polynomials A and B^R:

$$A(x) = a_0 + a_1x^1 + \cdots + a_mx^m$$
$$B^R(x) = b_n + b_{n-1}x^1 + \cdots + b_0x^n$$

Applying the formula derived in (3), the coefficient of x^k in the product polynomial $A \cdot B^R$ is then:

$$\sum_{j=0}^{k} a_j \cdot b_{n-(k-j)} = \sum_{j=0}^{k} a_j \cdot b_{(n-k)+j},$$

Notice that this is a cross-correlation between a and b at a shift of $n-k$! Thus, we have shown that the k-th term of the convolution between a and b^R is the $(n-k)$-th term of the cross-correlation between a and b.

We usually employ this technique when we want some kind of sliding window that computes a dot product at each possible starting location. Typical examples for this are matching problems where we want to measure the similarity (using a dot product) between one array and another at multiple different offsets or shifts, or in problems where we want to aggregate quantities over a sliding window.

Common Strategy:

Similar to the common strategy for convolution, you want to think about what a and b should represent in the summation $\sum_{i=1}^{k} a_i b_{k'+i}$. For instance, if the problem deals with finding matchings between bit strings, it may be helpful to construct "helpful" coefficients by mapping 0 → -1 and 1 → 1 such that the dot product can be used to properly measure similarity.

It's also useful to write out the resulting coefficients of your polynomial multiplication to double-check that you've set up your polynomials correctly.

4.1 Example Problem (String Matching)

Suppose we have a bitstring s of length $n+1$, and a pattern p (also a bitstring) of length $m+1 < n$. How do we efficiently find the (contiguous) substring of s which matches the pattern p at the largest number of positions?

Solution:

Intuition: First, we'll represent s and p in such a way that we can use a dot product to measure similarity between substrings.

Then, the question is essentially asking us to compute the cross-correlation of our representations of p and s, then find the index where the cross-correlation is highest. Since we want to perform sliding dot products over different slices of s, we'll represent p using our first polynomial and s with our second polynomial.

Main Idea: First, let's map the bits of s and p to numbers using the following function Φ:

$$\Phi(x) = \begin{cases} -1 : & x = 0 \\ 1 : & x = 1 \end{cases}$$

This way, $\Phi(p_i) \cdot \Phi(s_j) = 1$ if $p_i = s_j$ and is -1 otherwise, so larger dot products correspond with larger degrees of similarity. Now, the problem reduces to finding the index which maximizes the dot product of the resulting substrings, or in other words, finding the index that maximizes their cross-correlation.

Rather than just using cross-correlation as a black box, we'll approach this problem using only properties of FFT and polynomial multiplication. First, let's define polynomials to represent our arrays:

1. $P(x) = \Phi(p_0) + \Phi(p_1)x^1 + \Phi(p_2)x^2 + \cdots + \Phi(p_m)x^m$
2. $S(x) = \Phi(s_n) + \Phi(s_{n-1})x^1 + \Phi(s_{n-2})x^2 + \cdots + \Phi(s_0)x^n$

Notice that the coefficients of S are the entries of s in reverse order. Next, multiply those two polynomials using FFT. Finally, find the index k of the term with the largest leading coefficient, and return $(n-k)$.

Proof of Correctness: Let $R(x) = S(x) \cdot P(x)$. Suppose the term x^k in $R(x)$ has the largest leading coefficient c_k for some value of k, which can be expressed as.

$$c_k = \sum_{i=0}^{k} \Phi(p_i) \cdot \Phi(s_{n-(k-i)}) = \sum_{i=0}^{k} \Phi(p_i) \cdot \Phi(s_{(n-k)+i})$$

Which is exactly the dot product between the representations of the pattern p and the subarray of s starting at index $n-k$.

Runtime Analysis: Constructing the polynomials and padding them to the appropriate length takes $O(n+m)$ time. We then use FFT to multiply the two polynomials which have length $O(n)$ and $O(m)$ respectively. Thus the FFT and inverse FFT will take $O(n \log n)$ since to perform FFT polynomial multiplication, we need to pad both polynomials to have length $O(n+m) = O(n)$ (The n term dominates since $n > m$). The final searching process takes $O(m+n)$. Thus the final overall runtime will be $O(n \log n) + O(m+n) = O(n \log n)$.

5 Other helpful resources

Beyond the concepts, strategies, and examples we've included in this handout, we highly encourage you to review the following resources to bolster your understanding:

- Prof. Wright's lecture slides on FFT and applications: `https://drive.google.com/file/d/1g4mWiDi-YrAuf-RZPby33ORn4-64amtj/view?usp=sharing`.
- Awesome FFT Youtube Video by Reducible: `https://www.youtube.com/watch?v=h7apO7q16V0`.