Recitation 5: Fast Fourier Transform (FFT), Streaming Algorithms

1 Fast Fourier Transform (FFT)

1.1 Motivation: why do we care?

In lecture, we discussed the Fast Fourier Transform as an algorithm for multiplying two polynomials in $O(n \log n)$ time. Fundamentally, though, FFT is a powerful technique to efficiently transform signals between different domains.

Polynomial multiplication with FFT makes use of this by switching between point-value and coefficient representations of a polynomial, but FFT is also extremely important in the field of signal processing to transform a signal (let's say a sound wave) between the time and frequency domains. A sound wave can be decomposed into sinusoids of different frequencies that are added together and taking the FFT of a signal can tell you exactly what frequencies are present in your sound wave and in what proportions. This has a variety of applications including detecting pitches from a sound file, easily convolving signals, removing or filtering different frequencies from a signal, etc. For example, consider the problem of trying to apply a lowpass filter (i.e. remove all high frequencies) from an audio signal. In the frequency domain, this would be easy: simply remove frequencies above a given threshold. How would you do this in the time domain? Using FFT, we can easily convert a difficult time domain problem into a simpler and more tractable one in the frequency domain.

1.2 Algorithm and Analysis

Let's review the FFT algorithm for multiplying two polynomials. Let A and B be two degree bound n (degree $\leq n$) polynomials expressed as follows.

$$A(x) = a_0 + a_1 x + a_2 x^2 + .. + a_{n-1} x^{n-1} \tag{1}$$
$$B(x) = b_0 + b_1 x + b_2 x^2 + .. + b_{n-1} x^{n-1} \tag{2}$$

We call the length n vector $(a_0, a_1, .., a_{n-1})$ the coefficient vector representation for A and similarly, $(b_0, b_1, .., b_{n-1})$ is the coefficient vector representation for B. Equivalently, since any polynomial P is uniquely determined by its evaluations on $\geq degree(P)+1$ points, A

and B can also be represented by n points evaluated on the polynomial. We call this the point-value representation.

The product polynomial $C(x) = A(x) \cdot B(x)$ has degree $2 \cdot (n-1) < 2n$. Note that the first coefficient of $C(x)$ is $c_0 = a_0 b_0$, the second coefficient is $c_1 = a_0 b_1 + a_1 b_0$, ..., and the last coefficient is $c_{2n-2} = a_{n-1} b_{n-1}$. In general the coefficient c_k is given as follows.

$$c_k = \sum_{i=0}^{k} a_i b_{k-i} \tag{3}$$

Equation (3) is also known as the *convolution* of the coefficient vectors of A and B. Multiplying polynomials A and B, can now be expressed more mathematically as convolving their coefficient vectors. Note that trivially, it would take $O(n^2)$ time to compute this convolution. However, if A and B were evaluated at the same $2n$ points i.e. they were in point-value representation, multiplying them is easy. Simply do a point-wise multiplication in $O(n)$ time. Using this trick, as we saw in lecture, the Fast Fourier Transform allows us to compute this convolution in time $O(n \log n)$.

$$\begin{array}{ccc} A(x), B(x) \text{ coefficient representation} & \xrightarrow[O(n^2)]{\text{Naively}} & C(x) \text{ coefficient representation} \\ \text{FFT} \Big\downarrow O(n \log n) & & O(n \log n) \Big\uparrow \text{Inverse FFT} \\ A(x), B(x) \text{ point-value representation} & \xrightarrow[O(n)]{C(x) = A(x) \cdot B(x)} & C(x) \text{ point-value representation} \end{array}$$

Recall our algorithm for polynomial multiplication:

1. Evaluate polynomials $A(x)$ and $B(x)$ on the same $2n$ locations $\{x_0, x_1, .., x_{2n-1}\}$. ($O(n \log n)$ time using FFT)
2. Point-wise multiply the polynomials to obtain a point-value representation of $C(x) = A(x) \cdot B(x)$. ($O(n)$ time)
3. Compute the coefficient vector for C from the point-value representation. This is called *interpolation*. ($O(n \log n)$ time using Inverse FFT)

For step (1), we will choose the N-th roots of unity, where $N = 2^k$ is a power of 2 so that $2n <= N < 4n$ (Note that $N = O(n)$ and we could also have more than $2n$ points). The N-th roots of unity are given as $\{w^0, w^1, .., w^{N-1}\}$ where $w = e^{\frac{2\pi i}{N}}$. Note that $(w^k)^N = 1$ for $k \in \{0 .. N-1\}$.

Let y_k be value of A at $x_k = w^k$, we wish to evaluate A at all the roots of unity and find $y_0, y_1, .., y_{N-1}$ as follows:

$$y_k = \sum_{j=0}^{n-1} a_j e^{\frac{2\pi i k j}{N}} \tag{4}$$

Equation (4) is called the Discrete Fourier Transform (DFT) of A. The Fast Fourier Transform computes the DFT in $O(n \log n)$ time. The FFT to evaluate A at all the N-th roots of unity (also called DFT) is a divide and conquer algorithm given as follows:

1. Define

$$A_{even}(x) = a_0 + a_2 x + a_4 x^2 + .. + a_{n-2} x^{(n-2)/2} \tag{5}$$
$$A_{odd}(x) = a_1 + a_3 x + a_5 x^2 + .. + a_{n-1} x^{(n-2)/2} \tag{6}$$

   Hence

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2) \tag{7}$$

   Note that this is simply algebraic manipulation. Nothing special going on here. However, note that A_{even} and A_{odd} are degree bound $\frac{n}{2}$ polynomials.

2. Recursively evaluate $A_{even}(x)$ and $A_{odd}(x)$ on the *squares* of the N-th roots of unity.

$$\{(w^0)^2, (w^1)^2, .., (w_{N-1})^2\}$$

   But note that this set is the $\frac{N}{2}$-th roots of unity and has size $\frac{N}{2}$ (all the values in the second half of the set are a repeat of the first half, you can verify this easily). Hence, only $\frac{N}{2}$ evaluations need to be performed on $A_{even}(x)$ and $A_{odd}(x)$.

3. Evaluate $A(x)$ on all the N-th roots of unity using (7) and the recursively computed values for $A_{even}(x)$ and $A_{odd}(x)$.

We are now ready to compute the runtime for the FFT. Step (1) divides the problem into 2 problems of half the size (both in the degree of the polynomials and the number of evaluations). And it takes $O(N)$ time to perform the necessary additions in step (3). Hence, we have $T(N) = 2T(\frac{N}{2}) + O(N)$ which evaluates to $T(N) = \Theta(N \log N) = \Theta(n \log n)$ since $N = \Theta(n)$.

Hence, going back to our polynomial multiplication algorithm, we have performed step (1) in $O(n \log n)$ time. Step (2) is trivially $O(n)$ time to perform the point-wise multiplication. For step (3), we need to interpolate the evaluations of C to find the coefficient vector for C. The coefficient c_k can be obtained from the evaluations of C, y_t's, as follows:

$$c_k = \frac{1}{N} \sum_{t=0}^{N-1} y_t e^{\frac{-2\pi i t k}{N}} \tag{8}$$

This can easily be proven by plugging in the value of $y_t = \sum_{j=0}^{N-1} c_j e^{\frac{2\pi ijt}{N}}$ since we know the y_t's are the evaluations of C at the N-th roots of unity.

$$c_k = \frac{1}{N}\sum_{t=0}^{N-1} y_t e^{\frac{-2\pi itk}{N}} \tag{9}$$

$$= \frac{1}{N}\sum_{t=0}^{N-1}(\sum_{j=0}^{N-1} c_j e^{\frac{2\pi ijt}{N}})e^{\frac{-2\pi itk}{N}} \tag{10}$$

$$= \frac{1}{N}\sum_{j=0}^{N-1}(c_j \sum_{t=0}^{N-1} e^{\frac{2\pi it(j-k)}{N}}) \tag{11}$$

$$= \frac{1}{N}c_k N = c_k \tag{12}$$

Note that in (11), $\sum_{t=0}^{N-1} e^{\frac{2\pi it(j-k)}{N}} = N \cdot \delta_{j-k}$ is N if $j = k$ and 0 otherwise.[1]

Hence, (8) is known as the Inverse DFT, which is the same as the DFT but with a scaling factor and a minus sign. This can also be used with the Fast Fourier Transform using as evaluation points $\{w^0, w^{-1}, \cdots, w^{-(N-1)}\}$ which is also a collapsing set of the roots of unity.

Hence, in step (3) of our polynomial multiplication algorithm, the interpolation takes $O(n \log n)$ to obtain the polynomial $C(x)$ in coefficient form using the Fast Fourier Transform. Hence overall, we can multiply the two polynomials in $O(n \log n)$ time.

1.3 Minkowski Sum Using FFT

Let X and Y be length n sets of integers in the range $\{0, .., m-1\}$. The Minkowski Sum $X + Y$ is defined as the set:

$$X + Y = \{x + y \mid x \in X, y \in Y\} \tag{13}$$

We want to compute the size $|X + Y|$.

Naively, we can look through all n^2 pairs $(x, y)|x \in X, y \in Y$ and sum them up. We can then maintain a binary search tree of the values in order to avoid duplicate entries. Initialize a counter $i = 0$. Each time we insert a new value into the BST, increment i by 1. At the end of the process, i will be the size of $X + Y$. This process takes $O(n^2 \log n)$. If

[1] Here, δ is the Kronecker delta, where $\delta_i = 1$ if $i = 0$ and $\delta_i = 0$ otherwise when $i \neq 0$.

we use a different, perhaps randomized (Hash Table) method of avoiding duplicates, we could get a better runtime, but even this will take $\Omega(n^2)$ time.

We will now use the Fast Fourier Transform to solve this in $O(m \log m)$ time instead. The algorithm is as follows:

1. Construct a degree bound m polynomial $P_X(x)$ that has coefficient 1 for term x^k if $k \in X$, 0 otherwise.
2. Similarly, construct a degree bound m polynomial $P_Y(x)$ that has coefficient 1 for term x^k if $k \in Y$, 0 otherwise.
3. Multiply polynomials $P_{X+Y} = P_X(x) \cdot P_Y(x)$ using FFT.
4. Return the number of terms in P_{X+Y} that have a non-zero coefficient.

Note that by our construction, the coefficient of term $a_k x^k$ in P_{X+Y} is exactly the number of ways that numbers in X and Y can sum to k. If a_k is greater than 0, that means there exists values $x \in X$ and $y \in Y$ such that $x + y = k$. Hence, $|X + Y|$ is equal to the number of non-zero coefficients in P_{X+Y}.

Steps (1), (2) and (4) take $O(m)$ time and step (2) takes $O(m \log m)$ time. Hence, the overall cost of this algorithm is $O(m \log m)$. Note that if $m = O(n)$, this is $O(n \log n)$.

Note: One common mistake is to try to use FFT when the integers in X and Y are not bounded. If so, the integers in X and Y can get arbitrarily large, so the polynomials corresponding to X and Y will also be very large. Using FFT on those high degree polynomials could then take a lot of time.

1.4 Binary Strings

Alyssa P. Hacker has two binary strings (strings which contain only the characters '0' and '1'). Her first string A is of length $2n$ and her second string B is of length n. She wishes to find the locations of all substrings of A which have a Hamming distance from B which is less than or equal to d, for some nonnegative integer d. (The Hamming distance between two strings of the same length is defined as the number of positions at which they have different characters.) For example, when $A =$ '10101011', $B =$ '1010', and $d = 2$, the locations of all such substrings in A are indices 0, 2, and 4, since '1010' has a Hamming distance of 0 from B and '1011' has a Hamming distance of 1 from B.

Find an efficient algorithm for the problem. (Hint: Find a good polynomial representation for strings. It may help to first think about the case when $d = 0$.)

Pre-Solution: We start by trying to think of a good polynomial representation for binary strings. The most obvious representation to try would be to use a coefficient of 0 when the character is 0 and a coefficient of 1 when the character is 1. However, we also want

the product of two same characters to be some number x and the product of two different characters to be some different number y. This leads us to using the coefficient -1 for the character 0 and the coefficient 1 for the character 1 (or vice versa). With this representation, if two characters are the same, their product is 1, and if two characters are different, their product is -1. Note that there are also other representations that work.

We also wish to capture each Hamming distance in a coefficient within the product of our two polynomials. Since each coefficient of the product of two polynomials $a(x)b(x)$ is the dot product of the coefficients of $a(x)$ and the coefficients of $b(x)$ reversed, when we represent our strings as polynomials, we reverse the order of one of the strings. This gives us our solution below.

Solution: We find a polynomial representation for strings A and B such that multiplying these two polynomials solves our problem. Our representation is as follows.

Let $a_i = 1$ if the i^{th} character of A is '1', and let $a_i = -1$ if it is '0', for $0 \leq i < 2n$. Let b_i be the same, for $0 \leq i < n$. We represent these strings as polynomials as follows:

$$a(x) = a_0 + a_1 x + \ldots a_{2n-1} x^{2n-1}$$
$$b(x) = b_{n-1} + b_{n-2} x + \ldots b_0 x^{n-1}$$

Note that the coefficients of $b(x)$ are in reversed order. To facilitate index tracking, we define a polynomial $b'(x) = b'_0 + \ldots + b'_{n-1} x^{n-1}$ where $b'_i = b_{n-1-i}$. Compute $c(x) = a(x)b'(x)$ using FFT. Return the set $\{j - n + 1\}$ where j ranges over all indices $j \geq n - 1$ where $c_j \geq n - 2d$.

Correctness: follows from the following lemma.

Lemma 1 *For $j \geq n - 1$, $c_j \geq n - 2d$ if and only if the substring of A of length n ending with the j^{th} character has a Hamming distance $\leq d$ from B.*

Proof.

Assume that $j \geq n-1$ and write $j = n-1+k$. Note that the coefficient c_j is the convolution of the vector $(a_k, \ldots a_{n-1+k})$ and $(b'_0, \ldots b'_{n-1})$. This in turn is the same as the dot product of the vector $(a_k, \ldots a_{n-1+k})$ and $(b_0, \ldots b_{n-1})$. Note that this the dot product of the substring of A of length n *ending* in $n - 1 + k$ and B (with 0 and 1 mapped to -1 and 1 respectively).

What is the dot product equal to? When the characters matche, the product of the corresponding a_i and b_i is 1; otherwise, it is -1. If there are x non-matching characters, and $n - x$ matching characters, then the dot product is $n - 2x$.

Therefore, a coefficient $c_j \geq n - 2d$ represents the end of a substring of A with Hamming distance $\leq d$ from B. Such a substring starts on index $j - n - 1$.

Runtime analysis: The runtime of converting strings to polynomials is $O(n)$, performing FFT is $O(n \log n)$, and finding the large coefficients is $O(n)$. So our total runtime is $O(n \log n)$ as desired.

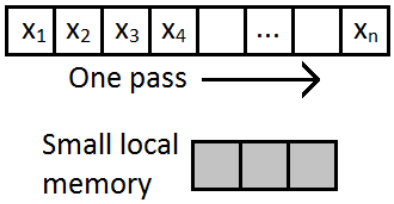
2 Streaming Algorithms

2.1 Overview

For most of 6.046 till now, we have been looking at algorithms that can see the entire input and make multiple passes over it if needed. In many cases, however, this may not be feasible. Some algorithms are "online" – they need to make decisions without seeing the whole input (*e.g.*, paging or caching). We will look at yet another class of algorithms: streaming algorithms. These are used in scenarios when memory is extremely limited compared to the size of the input, and where (typically) we can only make one pass over the input. In other words, we are not allowed to go back and re-look at any inputs. Examples of such scenarios can be found in routers processing IP packets, video and text streams, and gaming.

Note that streaming and online algorithms are not *quite* the same. Online algorithms have no space restrictions, but may need to produce partial output without seeing the rest of the input. Streaming algorithms have limited space ($O(n)$ or even $O(\log n)$), but only produce output after seeing the entire input.

Of course, these are very general guidelines, as there may be algorithms which cross over. Both classes, however, may relax correctness or optimality. For instance, we may sometimes relax the "one pass" rule to have $O(1)$ passes. These algorithms are sometimes exact, but more often only *probably approximately correct*.



2.2 Simple Statistics

Some examples of simple statistics that are often desired in contexts involving streaming:

- **Average**: we are asked to compute $\frac{\sum x_i}{n}$. Keep a running partial sum and counter, then divide at the end of the stream. This takes a single pass and uses logarithmic space.
- **Max element**: we are asked to compute $max(x_i)$. Keep a running partial answer (notice how this is basically online). This also takes a single pass and uses logarithmic space.

2.3 Majority Element

We are told that there exists an element which appears $> n/2$ times in the input. Below is a simple algorithm to find it (refer to lecture notes for more details).

```
counter = 0
majorityElement = 0
for each item in list:
  if item == majorityElement
    counter++
  else
    if counter == 0
      majorityElement = item
      counter = 1
    else
      counter--
return majorityElement
```

2.4 Number of Distinct Elements

The problem we want to solve here is to find an $(\epsilon, \delta)-$approximation for the number of distinct elements in the stream, which we will denote as F_o. That is, with a probability of at least $(1 - \delta)$ we want to output $\hat{F}_o$ such that

$$(1 - \epsilon)F_o \leq \hat{F}_o \leq (1 + \epsilon)F_o.$$

As we saw in lecture, we start with a simpler problem: to determine whether some estimate D for $F_o = D^*$ is reasonably close. More specifically, we want an algorithm that:

- outputs *YES* if $D^* \geq 2D$
- outputs *NO* if $D^* \leq D$
- outputs anything if otherwise

The idea is to randomly hash elements randomly to $\{0, 1, \ldots B-1\}$ where $B = 4d$. If there are many elements ($D^* \geq 2D$), then it is highly likely that at least one element will be mapped to some particular value we fix, say 0. If there are few elements ($D^* \leq D$), then it is more likely that none will be mapped to 0. Let's analyze this in more detail.

If $D^* \leq D$, then

$$\begin{aligned} Pr[YES] &= Pr[\text{at least one element mapped to 0}] \\ &\leq (\text{number of distinct elements}) \cdot Pr[\text{a given element is mapped to 0}] \\ &= D^* \cdot \frac{1}{B} \\ &= \frac{D^*}{4D} \leq \frac{D}{4D} = \frac{1}{4} \end{aligned}$$

So the probability that the algorithm *incorrectly* outputs *YES* in this case is at most $1/4$.

If $D^* \geq 2D$, then we want to lower bound the probability of *correctly* outputting *YES*. The worst case is when $D^* = 2D$, and then we can use the following Bonferroni inequality which complements the union-bound:

$$Pr[\cup_i A_i] \geq \sum_i Pr[A_i] - \sum_{i<j} Pr[A_i \cap A_j]$$

Let the elements in the stream be $m_1, \ldots m_{D^*}$ and let A_i be the event that the ith element is hashed to 0. Then,

$$\begin{aligned} Pr[YES] &= Pr[\cup_i A_i] \\ &\geq \sum_i Pr[A_i] - \sum_{i<j} Pr[A_i \cap A_j] \\ &= D^*\left(\frac{1}{B}\right) - \frac{D^*(D^*-1)}{2}\left(\frac{1}{B^2}\right) \\ &\geq \frac{D^*}{B} - \frac{(D^*)^2}{2B^2} \\ &= \frac{D^*}{B}\left(1 - \frac{D^*}{2B}\right) \\ &= \frac{2D}{4D}\left(1 - \frac{2D}{8D}\right) \\ &= \frac{1}{2}\left(1 - \frac{1}{4}\right) = \frac{3}{8} \end{aligned}$$

In summary, we correctly output *YES* with probability at least $3/8$ when $D^* \geq 2D$ and incorrectly so with probability at most $1/4$ when $D^* \leq D$. To amplify the probability of success, we can do this k times in parallel. In case of $D^* \geq 2D$, we expect $\frac{3}{8}k$ *YES* answers.

Otherwise, if $D^* < D$, we expect only $\frac{1}{4}k$ *YES* answers. We can use a threshold in the middle ($\frac{5}{16}k$) to decide whether we output *YES* or *NO*. Finally, we can set k as a function of δ to make sure the error is smaller than δ – try this as an exercise.

What happens if we don't know D? In that case, we test $D = 1, 2, 4, \ldots 2^r, \ldots m$ in parallel which takes $\log m$ parallel runs.

One last note is that we cannot use a truly random function because that takes over m space to remember the mapped values for all elements. Instead, we actually only need hash functions from a family that is *pairwise independent*, which means that for any two distinct elements x_1, x_2 and distinct hash values y_1, y_2, the following holds:

$$Pr_{h \in \mathcal{H}}[h(x_1) = y_1 \quad \text{and} \quad h(x_2) = y_2] = \frac{1}{m^2}$$

Pairwise independent is actually sufficient for the bounds we derived above.

2.5 Reservoir Sampling

Suppose we want to sample a uniformly random element from a stream of (a-priori) unknown size. That is, for each i, we want to output x_i with probability $1/n$. Unfortunately, the problem is that until the stream ends, we don't know what n is.

At each point, we must assume that it is possible for the current element is the last one (the next input we get might be NULL, ending the sequence). So, our probability of keeping elements at each stage looks like this:

- With only one element, x_1, we must keep it with probability 1. We will write our probability distribution over the elements seen in the stream so far as $\{1\}$.
- With two elements, we need to keep x_1 with a reduced probability of $1/2$ and x_2 also with probability $1/2$. We will write this as $\{\frac{1}{2}, \frac{1}{2}\}$.
- With three elements, we want our probabilities to be $\{\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\}$, and so on.

At any time i, we have a random sample from $x_1, x_2, \ldots, x_i$, with each element having been kept with probability $\frac{1}{i}$. So, what should we do when we see x_{i+1} in order to get $\frac{1}{i+1}$ for each element?

The answer is to keep each element we already have with probability $\frac{i}{i+1}$. Since the probability that each of the first i elements is kept is $\frac{1}{i}$, after this step the probability that each of the first $i+1$ elements is kept becomes $\frac{1}{i} \times \frac{i}{i+1} = \frac{1}{i+1}$, as desired. The new element, x_{i+1}, of course, should be kept with probability $\frac{1}{i+1}$. Now, all of the first $i+1$ elements are kept with probability $\frac{1}{i+1}$ each.

Now, suppose we want to sample not a single element but instead k random elements. We can generalize the above streaming algorithm to k samples by keeping a "reservoir"

of k elements at all times. The first k elements are automatically included. After the kth element, for each new x_{i+1}, we keep it with probability $\frac{k}{i+1}$ and remove a random element from the k in the reservoir.

Reservoir sampling may also be generalized to include weighted sampling, where each x_i comes with a weight w_i. The problem then becomes that of sampling each x_i with probability $\frac{w_i}{\sum_{j=1}^{n} w_j}$. How would you modify your algorithm in this scenario?

3 Appendix: Example of FFT

In this section, we will walk through how the FFT converts the following polynomial into point-value form:

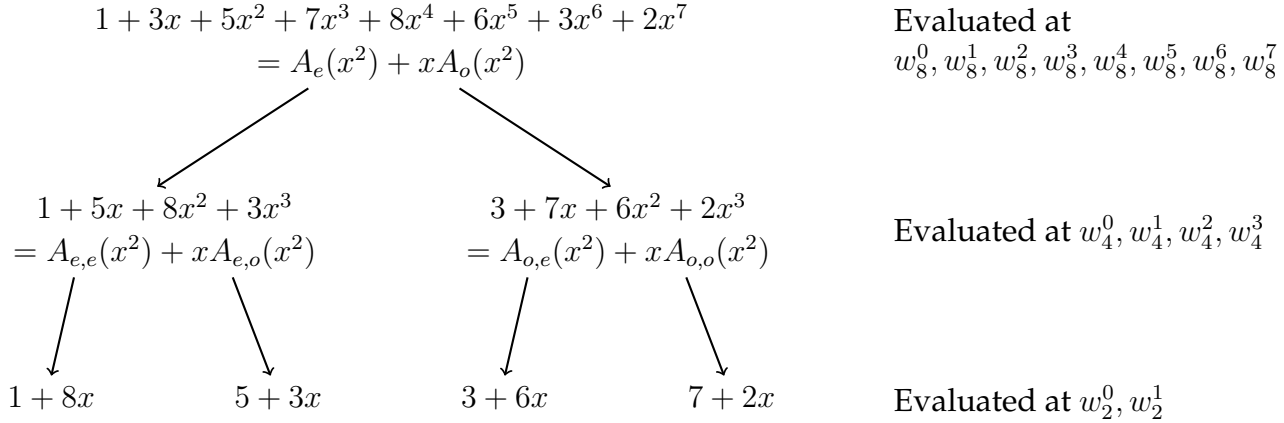
$$P(x) = 1 + 3x + 5x^2 + 7x^3 + 8x^4 + 6x^5 + 3x^6 + 2x^7$$

This is a degree-bound 8 polynomial, and we will evaluate it at the eighth roots of unity $(w_8^0, w_8^1, w_8^2, w_8^3, w_8^4, w_8^5, w_8^6, w_8^7)$.

By definition, these roots of unity are equal to

$$\begin{aligned}
w_8^0 &= 1 \\
w_8^1 &= \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}} \\
w_8^2 &= i \\
w_8^3 &= -\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}} \\
w_8^4 &= -1 \\
w_8^5 &= -\frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}} \\
w_8^6 &= -i \\
w_8^7 &= \frac{1}{\sqrt{2}} - \frac{i}{\sqrt{2}}
\end{aligned}$$

This diagram depicts how the FFT evaluates $P(x)$ at the roots of unity. For the sake of clarity, we treat the recursive calls as bottoming out when the polynomial is of degree-bound 2.



This yields the following results:

Evaluating the leaves of the recursion tree:

x	$A_{e,e}(x)$	$A_{e,o}(x)$	$A_{o,e}(x)$	$A_{o,o}(x)$
$w_2^0 = 1$	$1 + 8(1) = 9$	$5 + 3(1) = 8$	$3 + 6(1) = 9$	$7 + 2(1) = 9$
$w_2^1 = -1$	$1 + 8(-1) = -7$	$5 + 3(-1) = 2$	$3 + 6(-1) = 3$	$7 + 2(-1) = 5$

Evaluating the middle level of the recursion tree:

x	$A_{e,e}(x^2) + xA_{e,o}(x^2) = A_e(x)$	$A_{o,e}(x^2) + xA_{o,o}(x^2) = A_o(x)$
$w_4^0 = 1$	$9 + 1(8) = 17$	$9 + 1(9) = 18$
$w_4^1 = i$	$-7 + i(2)$	$-3 + i(5)$
$w_4^2 = -1$	$9 - 1(8) = 1$	$9 - 1(9) = 0$
$w_4^3 = -i$	$-7 - i(2)$	$-3 - i(5)$

Evaluating the root of the recursion tree:

x	$A_e(x^2) + xA_o(x^2) = P(x)$
w_8^0	$17 + 1(18) = 35$
w_8^1	$-7 + 2i + (1/\sqrt{2} + i/\sqrt{2})(-3 + 5i)$
w_8^2	$1 + i(0) = 1$
w_8^3	$-7 - 2i + (-1/\sqrt{2} + i/\sqrt{2})(-3 - 5i)$
w_8^4	$17 - 18 = -1$
w_8^5	$-7 + 2i + (-1/\sqrt{2} - i/\sqrt{2})(-3 + 5i)$
w_8^6	$1 - i(0) = 1$
w_8^7	$-7 - 2i + (1/\sqrt{2} - i/\sqrt{2})(-3 - 5i)$

This final table is a point-value representation of $P(x)$. Since $P(x)$ has degree-bound 8 and is evaluated at 8 distinct points, these point-value pairs uniquely identify P. You can verify that these values are correct by plugging and solving by hand (or more realistically by asking WolframAlpha to evaluate $P(x)$ at these points). If you wanted, you could now apply the inverse FFT to these points and recover the original coefficients of P. If you are still confused about the inverse FFT, it may be useful to do this exercise.