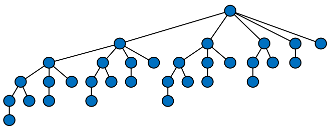
Lecture 7. Convolution and the Fast Fourier Transform

CpSc 8400: Algorithms and Data Structures

Brian C. Dean



School of Computing
Clemson University
Spring, 2016

Convolution

- The convolution of two length-n sequences of numbers $a_0\, a_1 \ldots a_{n-1}$ and $b_0\, b_1 \ldots b_{n-1}$ is:

  a_0b_0
  $a_0b_1 + a_1b_0$
  $a_0b_2 + a_1b_1 + a_2b_0$
  $a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$
  …
  $a_{n-2}b_{n-1} + a_{n-1}b_{n-2}$
  $a_{n-1}b_{n-1}$

- It has total length 2n – 1.

Examples of Convolution

- **Polynomial multiplication**
  - Consider polynomials

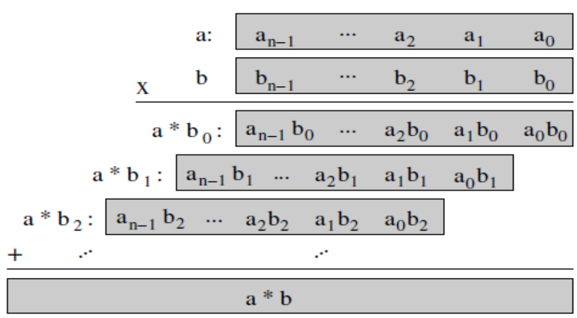
    $A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$

    $B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1}$

    The product of these polynomials A(x)B(x) will be a polynomial of degree 2n – 2 whose coefficients are exactly the convolution of $a_0a_1\ldots a_{n-1}$ and $b_0b_1\ldots b_{n-1}$.
- **Integer multiplication**
  - The base-10 integer 9876 can be represented by a polynomial $P(x) = 9x^3 + 8x^2 + 7x + 6$ evaluated at $x = 10$.
  - Multiplication of two integers (in any base) therefore corresponds roughly to polynomial multiplication (modulo a few carries after the fact to fix up overflowing digits).

3

Examples of Convolution



- **Integer multiplication**
  - The base-10 integer 9876 can be represented by a polynomial $P(x) = 9x^3 + 8x^2 + 7x + 6$ evaluated at $x = 10$.
  - Multiplication of two integers (in any base) therefore corresponds roughly to polynomial multiplication (modulo a few carries after the fact to fix up overflowing digits).

4

Examples of Convolution

- **Probability distributions of sums**
  - Suppose the number of boys in a class is distributed as:

    1: 1/6   2: 1/2   3: 1/3

    and the number of girls is distributed as:

    1: 1/4   2: 1/4   3: 1/2

    What is the probability distribution for the total number of students in the class?
- **Counting combinations**
  - How many different 6-piece fruit baskets can I construct with 2..4 apples, 1..2 bananas, and 1..3 pears?
- **Discrete-time signal processing**
  - What happens if we convolve a long signal with <1, 1>?

Convolution Algorithms

- The straightforward algorithm for convolving two length-n sequences $a_0a_1\dots a_{n-1}$ and $b_0b_1\dots b_{n-1}$ runs in $\Theta(n^2)$ time.

a_0b_0

$a_0b_1 + a_1b_0$

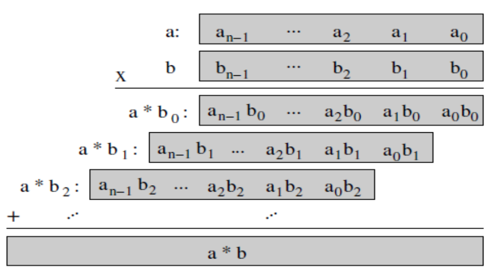
$a_0b_2 + a_1b_1 + a_2b_0$

$a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0$

…

$a_{n-2}b_{n-1} + a_{n-1}b_{n-2}$

$a_{n-1}b_{n-1}$



What About Divide and Conquer?

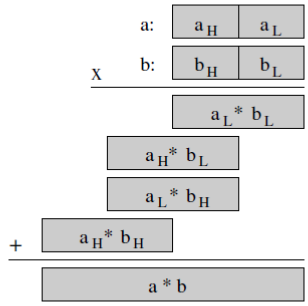
- We want to compute the coefficients of the product polynomial A(x)B(x), where

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$$
$$B(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1}$$

- Divide into two pieces:

$$A(x) = A_{low}(x) + x^{n/2} A_{high}(x)$$
$$B(x) = B_{low}(x) + x^{n/2} B_{high}(x)$$

- And multiply:



$$A(x)B(x) = A_{low}(x) B_{low}(x) + x^{n/2} (A_{high}(x) B_{low}(x) + A_{low}(x) B_{high}(x)) + x^n A_{high}(x) B_{high}(x)$$

- This uses 4 recursive multiplications of degree-n/2 polynomials (plus Θ(n) extra work): T(n) = 4T(n/2) + Θ(n)…

7

A Clever Trick…

- It turns out we can reduce the number of recursive multiplications from 4 down to 3…

$$A(x) = A_{low}(x) + x^{n/2} A_{high}(x) \qquad B(x) = B_{low}(x) + x^{n/2} B_{high}(x)$$

$$A(x)B(x) = A_{low}(x)B_{low}(x) + x^{n/2} (A_{high}(x) B_{low}(x) + A_{low}(x) B_{high}(x)) + x^n A_{high}(x) B_{high}(x)$$

- Let's compute these three products:

$$P_1(x) = A_{low}(x) B_{low}(x)$$
$$P_2(x) = A_{high}(x) B_{high}(x)$$
$$P_3(x) = (A_{low}(x) + A_{high}(x)) (B_{low}(x) + B_{high}(x))$$

- Why do these 3 products suffice?
- $T(n) = 3T(n/2) + \Theta(n)$ solves to $T(n) \approx \Theta(n^{1.58})$

8

The FFT

- The **Fast Fourier Transform (FFT)** helps us perform convolution in only O(n log n) time.
- Convolution and Fourier transforms play a fundamental role in in nearly every digital signal processing device, so the FFT is extremely important and widely-used in practice!
- For free, we also get O(n log n) algorithms for:
  - Multiplying two polynomials of degree n.
  - Multiplying two n-digit integers (very useful in cryptography, etc.).
  - Pattern matching with wildcards (later in this lecture).
  - ... And many other convolution-related problems…

9

Fun Aside: Secret Sharing

- Can I distribute a secret number among all N people in class so that:
  - Any two people, working together, can discover the secret.
  - Any one person, by themselves, knows effectively <u>nothing</u> about the secret.

- **Example:** In Russia in 1992, any two of the following three parties had to agree to order the arming of nuclear weapons: President Boris Yeltsin, Defense Minister Yevgeni Shaposhnikov, and the Defense Ministry chief of staff.

10

Fun Aside: Secret Sharing

- Can I distribute a secret number among all N people in class so that:
  - Any two people, working together, can discover the secret.
  - Any one person, by themselves, knows effectively <u>nothing</u> about the secret.
- **Easy Solution:** Make the secret the y-intercept of a line, and tell each person one point on the line.
- What if we need a group of 3+ to determine the secret, while any group of <3 should know nothing about it?

Fun Aside: Secret Sharing

- Can I distribute a secret number among all N people in class so that:
  - Any two people, working together, can discover the secret.
  - Any one person, by themselves, knows effectively <u>nothing</u> about the secret.
- **Easy Solution:** Make the secret the y-intercept of a line, and tell each person one point on the line.
- What if we need a group of 3+ to determine the secret, while any group of <3 should know nothing about it? **Use quadratics, not lines!**

Point-Value Representation

- A degree-(n-1) polynomial is uniquely represented by its value at n points!
- $A(x) = a_0 + a_1x^1 + \dots + a_{n-1}x^{n-1}$ (degree $n - 1$) can be uniquely specified in two different ways:
  - In terms of its n coefficients $a_0, a_1, \dots, a_{n-1}$.
  - In terms of n (point, value) pairs: $(x_1, A(x_1)), \dots, (x_n, A(x_n))$ (i.e., 2 points uniquely determine a line, 3 determine a parabola, 4 determine a cubic polynomial, etc.)
- Key observation: multiplication of two polynomials is much easier using (point, value) representations!

13

Multiplication Using Point-Value Representation

- When we multiply polynomials A(x) and B(x) (both of degree $n - 1$), we obtain a product polynomial C(x) of degree $2n - 2$.
- To compute C(x), we can do the following:
  - **Evaluate** A(x) and B(x) at a common set of $2n - 1$ points:

    $(x_1, A(x_1)), (x_2, A(x_2)), \dots, (x_{2n-1}, A(x_{2n-1}))$

    $(x_1, B(x_1)), (x_2, B(x_2)), \dots, (x_{2n-1}, B(x_{2n-1}))$
  - **Multiply** these values together in $\Theta(n)$ time:

    $(x_1, C(x_1)), (x_2, C(x_2)), \dots, (x_{2n-1}, C(x_{2n-1}))$
  - Convert C from (point, value) to coefficient form (known as **interpolation**).

14

Multiplying Polynomials by Changing Representation

Polynomials $A(x)$ and $B(x)$ each evaluated at $2n - 1$ points:

A: $(x_1, A(x_1)), (x_2, A(x_2)), \ldots, (x_{2n-1}, A(x_{2n-1}))$
B: $(x_1, B(x_1)), (x_2, B(x_2)), \ldots, (x_{2n-1}, B(x_{2n-1}))$

$\xrightarrow{\Theta(n)}$

Product polynomial $C(x) = A(x)B(x)$ evaluated at $2n - 1$ points:

$(x_1, C(x_1)), (x_2, C(x_2)), \ldots, (x_{2n-1}, C(x_{2n-1}))$

Evaluation

Interpolation

Polynomials $A(x)$ and $B(x)$ of degree $n - 1$ initially specified by n coefficients each:

$A(x) = a_0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$
$B(x) = b_0 + b_1x^1 + \ldots + b_{n-1}x^{n-1}$

Product polynomial $C(x) = A(x)B(x)$ of degree $2n - 2$ specified by $2n - 1$ coefficients:

$C(x) = c_0 + c_1x^1 + \ldots + c_{2n-2}x^{2n-2}$

The missing piece: how hard is evaluation and interpolation?...

15

Evaluation and Interpolation

- Evaluation of $A(x)$ at a specific point x takes only $\Theta(n)$ time using Horner's method:

$$\begin{aligned} A(x) &= a_0 + a_1x^1 + \ldots + a_{n-1}x^{n-1} \\ &= a_0 + x(a_1 + x(a_2 + \ldots + x(a_{n-2} + x(a_{n-1})))))) \end{aligned}$$

- So it takes $\Theta(n^2)$ time to evaluate at $2n - 1$ points.

- Interpolation boils down to solving a linear system with n equations and n variables, which takes:
  - $\Theta(n^3)$ time using straightforward Gaussian elimination, or
  - $\Theta(n^2)$ time with fancier techniques (since this linear system has very special structure).

16

Updated Picture

Polynomials A(x) and B(x) each evaluated at $2n - 1$ points:

A: $(x_1, A(x_1)), (x_2, A(x_2)), \ldots, (x_{2n-1}, A(x_{2n-1}))$
B: $(x_1, B(x_1)), (x_2, B(x_2)), \ldots, (x_{2n-1}, B(x_{2n-1}))$

$\Theta(n)$

Product polynomial $C(x) = A(x)B(x)$ evaluated at $2n - 1$ points:

$(x_1, C(x_1)), (x_2, C(x_2)), \ldots, (x_{2n-1}, C(x_{2n-1}))$

Evaluation
($\Theta(n^2)$ time)

Interpolation
($\Theta(n^2)$ time)

Polynomials A(x) and B(x) of degree $n - 1$ initially specified by n coefficients each:

$A(x) = a_0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$
$B(x) = b_0 + b_1x^1 + \ldots + b_{n-1}x^{n-1}$

Product polynomial $C(x) = A(x)B(x)$ of degree $2n - 2$ specified by $2n - 1$ coefficients:

$C(x) = c_0 + c_1x^1 + \ldots + c_{2n-2}x^{2n-2}$

So the complete algorithm still takes $\Theta(n^2)$ time. How do we fix this…

17

Using the FFT…

Polynomials A(x) and B(x) each evaluated at $2n - 1$ points:

A: $(x_1, A(x_1)), (x_2, A(x_2)), \ldots, (x_{2n-1}, A(x_{2n-1}))$
B: $(x_1, B(x_1)), (x_2, B(x_2)), \ldots, (x_{2n-1}, B(x_{2n-1}))$

$\Theta(n)$

Product polynomial $C(x) = A(x)B(x)$ evaluated at $2n - 1$ points:

$(x_1, C(x_1)), (x_2, C(x_2)), \ldots, (x_{2n-1}, C(x_{2n-1}))$

Evaluation
$\Theta(n \log n)$ time if we use the FFT to evaluate A and B at points $x_1 \ldots x_{2n-1}$ that are *roots of unity*.

Interpolation
$\Theta(n \log n)$ time if we use the inverse FFT, assuming $x_1 \ldots x_{2n-1}$ are *roots of unity*.

Polynomials A(x) and B(x) of degree $n - 1$ initially specified by n coefficients each:

$A(x) = a_0 + a_1x^1 + \ldots + a_{n-1}x^{n-1}$
$B(x) = b_0 + b_1x^1 + \ldots + b_{n-1}x^{n-1}$

Product polynomial $C(x) = A(x)B(x)$ of degree $2n - 2$ specified by $2n - 1$ coefficients:

$C(x) = c_0 + c_1x^1 + \ldots + c_{2n-2}x^{2n-2}$

Now the entire algorithm runs in only $\Theta(n \log n)$ time!

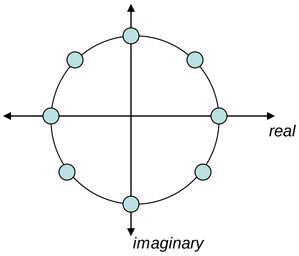
18

Complex Roots of Unity

- {+1, -1} are the two square roots of 1.
- {+1, -1, +i, -i} are the four 4^{th} roots of 1.
- {+1, -1, +i, -i, α(1+i), α(1-i), α(-1+i), α(-1-i)} are the eight 8^{th} roots of 1, where $\alpha = \sqrt{2}/2$.
- In general, there are k different k^{th} roots of 1:

  $e^{i\Theta} = \cos\Theta + i \sin\Theta$ for $\Theta = 0, 2\pi/k, 4\pi/k, \ldots, 2(k-1)\pi/k$

  (in other words, these are k equally-spaced points along the unit circle in the complex plane)
- Key property: Take the set of all k^{th} roots of unity. Square them. We get the set of all (k/2)th roots of unity (each counted twice).



A Divide and Conquer Approach…

- Consider $A(x) = a_0 + a_1x^1 + a_2x^2 + \ldots$ of degree < n.
- **Goal:** Evaluate A(x) at all n of the n^{th} roots of unity, where n is a power of 2.

  (note that we can assume n is a power of 2 without loss of generality…)

A Divide and Conquer Approach…

- Consider $A(x) = a_0 + a_1x^1 + a_2x^2 + \ldots$ of degree $< n$.
- **Goal:** Evaluate $A(x)$ at all n of the n^{th} roots of unity, where n is a power of 2.

- Divide A into its **odd** and **even** terms:

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

- For example:

$A(x) = 3 + 8x + 4x^2 + 9x^3 + 2x^4 + 7x^5 + 6x^6 – 5x^7$,

$A_{odd}(x) = 8 + 9x + 7x^2 – 5x^3$

$A_{even}(x) = 3 + 4x + 2x^2 + 6x^3$

- Note that A_{odd} and A_{even} have degree $< n/2$

A Divide and Conquer Approach…

- **Recall:** $A(x) = x\,A_{odd}(x^2) + A_{even}(x^2)$, where
  - $A(x)$: polynomial of degree $< n$
  - $A_{odd}(x)$ and $A_{even}(x)$: polynomials of degree $< n/2$
- **Goal:** Evaluate $A(x)$ at all n of the n^{th} roots of unity.
- For example, if $n = 8$, we want to compute:

$A(1)$
$A(-1)$
$A(i)$
$A(-i)$
$A(+\sqrt{2}/2 + i\sqrt{2}/2)$
$A(-\sqrt{2}/2 – i\sqrt{2}/2)$
$A(+\sqrt{2}/2 – i\sqrt{2}/2)$
$A(-\sqrt{2}/2 + i\sqrt{2}/2)$

A Divide and Conquer Approach…

- **Recall:** $A(x) = x\, A_{odd}(x^2) + A_{even}(x^2)$, where
  - $A(x)$: polynomial of degree $< n$
  - $A_{odd}(x)$ and $A_{even}(x)$: polynomials of degree $< n/2$
- **Goal:** Evaluate $A(x)$ at all n of the n^{th} roots of unity.
- For example, if $n = 8$, we want to compute:

$A(1) = (1)\, A_{odd}(1^2) + A_{even}(1^2)$

$A(-1) = (-1)\, A_{odd}((-1)^2) + A_{even}((-1)^2)$

$A(i) = (i)\, A_{odd}(i^2) + A_{even}(i^2)$

$A(-i) = (-i)\, A_{odd}((-i)^2) + A_{even}((-i)^2)$

$A(+\sqrt{2}/2 + i\sqrt{2}/2) = (+\sqrt{2}/2 + i\sqrt{2}/2)\, A_{odd}((+\sqrt{2}/2 + i\sqrt{2}/2)^2) + A_{even}(\ldots)$

$A(-\sqrt{2}/2 - i\sqrt{2}/2) = (-\sqrt{2}/2 - i\sqrt{2}/2)\, A_{odd}((-\sqrt{2}/2 - i\sqrt{2}/2)^2) + A_{even}(\ldots)$

$A(+\sqrt{2}/2 - i\sqrt{2}/2) = (+\sqrt{2}/2 - i\sqrt{2}/2)\, A_{odd}((+\sqrt{2}/2 - i\sqrt{2}/2)^2) + A_{even}(\ldots)$

$A(-\sqrt{2}/2 + i\sqrt{2}/2) = (-\sqrt{2}/2 + i\sqrt{2}/2)\, A_{odd}((-\sqrt{2}/2 + i\sqrt{2}/2)^2) + A_{even}(\ldots)$

A Divide and Conquer Approach…

- **Recall:** $A(x) = x\, A_{odd}(x^2) + A_{even}(x^2)$, where
  - $A(x)$: polynomial of degree $< n$
  - $A_{odd}(x)$ and $A_{even}(x)$: polynomials of degree $< n/2$
- **Goal:** Evaluate $A(x)$ at all n of the n^{th} roots of unity.
- For example, if $n = 8$, we want to compute:

$A(1) = (1)\, A_{odd}(1) + A_{even}(1)$

$A(-1) = (-1)\, A_{odd}(1) + A_{even}(1)$

$A(i) = (i)\, A_{odd}(-1) + A_{even}(-1)$

$A(-i) = (-i)\, A_{odd}(-1) + A_{even}(-1)$

$A(+\sqrt{2}/2 + i\sqrt{2}/2) = (+\sqrt{2}/2 + i\sqrt{2}/2)\, A_{odd}(i) + A_{even}(i)$

$A(-\sqrt{2}/2 - i\sqrt{2}/2) = (-\sqrt{2}/2 - i\sqrt{2}/2)\, A_{odd}(i) + A_{even}(i)$

$A(+\sqrt{2}/2 - i\sqrt{2}/2) = (+\sqrt{2}/2 - i\sqrt{2}/2)\, A_{odd}(-i) + A_{even}(-i)$

$A(-\sqrt{2}/2 + i\sqrt{2}/2) = (-\sqrt{2}/2 + i\sqrt{2}/2)\, A_{odd}(-i) + A_{even}(-i)$

A Divide and Conquer Approach…

- **Recall:** $A(x) = x\,A_{odd}(x^2) + A_{even}(x^2)$, where
  - $A(x)$: polynomial of degree < n
  - $A_{odd}(x)$ and $A_{even}(x)$: polynomials of degree < n/2
- **Goal:** Evaluate A(x) at all n of the n^{th} roots of unity.
- For example, if n = 8, we want to compute:

$$
\begin{aligned}
A(1) &= (1)\,A_{odd}(1) + A_{even}(1) \\
A(-1) &= (-1)\,A_{odd}(1) + A_{even}(1) \\
A(i) &= (i)\,A_{odd}(-1) + A_{even}(-1) \\
A(-i) &= (-i)\,A_{odd}(-1) + A_{even}(-1) \\
A(+\sqrt{2}/2 + i\sqrt{2}/2) &= (+\sqrt{2}/2 + i\sqrt{2}/2)\,A_{odd}(i) + A_{even}(i) \\
A(-\sqrt{2}/2 - i\sqrt{2}/2) &= (-\sqrt{2}/2 - i\sqrt{2}/2)\,A_{odd}(i) + A_{even}(i) \\
A(+\sqrt{2}/2 - i\sqrt{2}/2) &= (+\sqrt{2}/2 - i\sqrt{2}/2)\,A_{odd}(-i) + A_{even}(-i) \\
A(-\sqrt{2}/2 + i\sqrt{2}/2) &= (-\sqrt{2}/2 + i\sqrt{2}/2)\,A_{odd}(-i) + A_{even}(-i)
\end{aligned}
$$

So we need to evaluate A_{odd} and A_{even} (both of degree < n/2) at all n/2 of the (n/2)nd roots of unity… Time required: 2T(n/2)

…then we plug these results into the expressions above and evaluate them (in Θ(n) time) to obtain the value of A at all n of the n^{th} roots of unity.

25

The Inverse FFT

- The inverse FFT involves interpolating a degree-n polynomial that has been evaluated at the n roots of unity.
- Remarkably, we can perform an inverse FFT by running a standard FFT where:
  - Every root of unity ω is replaced with $-\omega$, and
  - We multiply the final n values computed as output by 1/n.

  (not difficult to prove by looking carefully at properties of the Fourier matrix)
- So the inverse FFT also takes Θ(n log n) time.

26

Avoiding Real Arithmetic

- Our approach so far uses complex numbers as roots of unity – this involves arithmetic on (potentially irrational) real numbers.
  – We're technically in the real RAM model of computation.
  – On an actual digital computer, we might encounter some round-off errors when implementing this approach.
- Fortunately, we can perform FFTs and inverse FFTs using only integer arithmetic on a RAM, if we use integer arithmetic modulo a suitable prime p.
  – Why a prime? This gives us the ability to use <u>division</u>!
  – Roots of unity still well-defined (and there's a way to compute them efficiently). For example, {1, 2, 4, 8, 9, 13, 15, 16} are the 8^{th} roots of unity in arithmetic modulo 17.

27

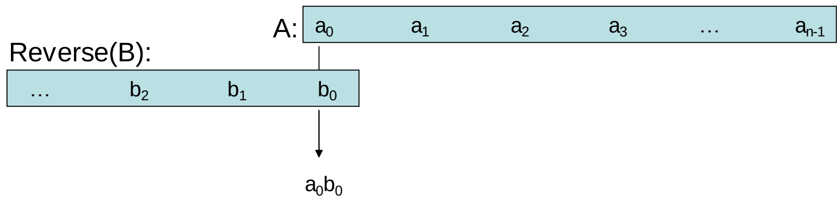
A Surprising Application of Convolution: Pattern Matching!

- **Problem**: find all occurrences of a short pattern P[1..m] within a long text T[1…n].
- For example: find all occurrences of **CAT** in:
  G<u>**CAT**</u>GTCAGTGCACGATCGAG<u>**CAT**</u>TCAGTCAGA<u>**CAT**</u>
- This is a classical problem in algorithms, and we know several elegant ways to solve it in O(n) time.
  (note that the naïve solution runs in O(mn) time)
- However, none of the fancy O(n) solutions allow "wildcard" characters. For example, find all matches of CA? in:
  G<u>**CAT**</u>GT<u>**CAG**</u>TG<u>**CAC**</u>GATCGAG<u>**CAT**</u>T<u>**CAG**</u>T<u>**CAG**</u>A<u>**CAT**</u>
- Using the FFT, we can solve this problem in Θ(n log n) time!
  (for binary strings)

28

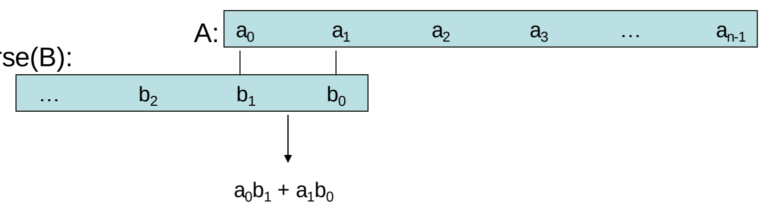
Another Way to Look at Convolution

- Let's convolve $A = a_0a_1\ldots a_{n-1}$ with $B = b_0b_1\ldots b_{n-1}$:



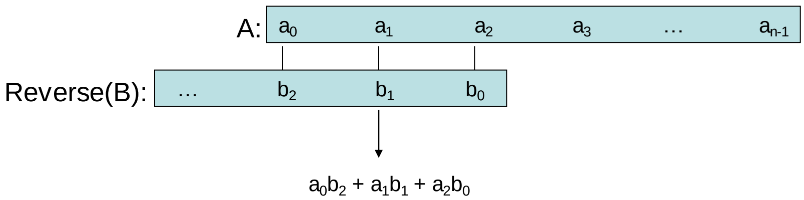
Another Way to Look at Convolution

- Let's convolve $A = a_0a_1\ldots a_{n-1}$ with $B = b_0b_1\ldots b_{n-1}$:



Another Way to Look at Convolution

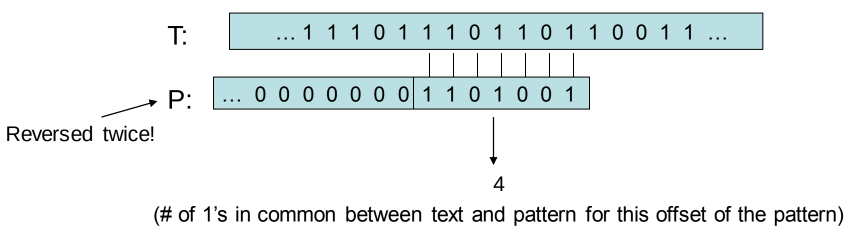
- Let's convolve $A = a_0a_1\ldots a_{n-1}$ with $B = b_0b_1\ldots b_{n-1}$:



- We can interpret the output of a convolution as the result of a series of "shifted dot products" between A and Reverse(B).
- This is starting to resemble pattern matching…!

Convolution with the FFT

- Take a binary pattern P[1..m] = 1101001 and a much longer text T[1..n].
- Reverse the pattern string.
- Add 0's after the pattern so it also has length n.
- Now convolve P[1..n] and T[1..n]. What does the output tell us?...

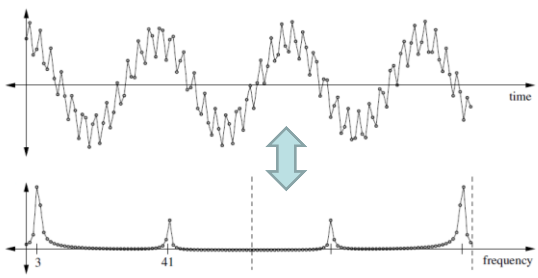


Convolution with the FFT

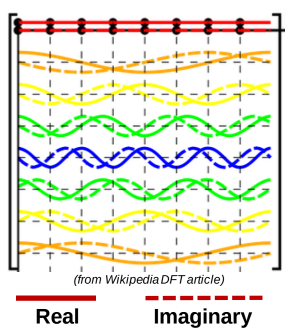
- Do the same thing as on the last slide, only first invert the 0's and 1's in the pattern and text.
- Now the elements of the convolution tell us the number of 0's in common at each shift of the pattern!
- So add the two up, and we know our length-m pattern matches at any offset where there are a total of m digits in common.
- How do we modify this approach to deal with wildcard characters in the pattern and text?
- What if our strings aren't binary?

33

Signal Processing: Frequency Space Interpretation of DFT Output



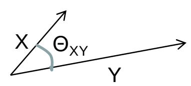
$$\begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \omega_n^{0\cdot 2} & \dots & \omega_n^{0\cdot(n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \omega_n^{1\cdot 2} & \dots & \omega_n^{1\cdot(n-1)} \\ \omega_n^{2\cdot 0} & \omega_n^{2\cdot 1} & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot(n-1)} \\ \vdots & & & & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \dots & \omega_n^{(n-1)\cdot(n-1)} \end{bmatrix} =$$



The Fourier matrix F_n. A discrete Fourier Transform (DFT) is equivalent to multiplying a vector by this matrix. Here, ω is the complex nth root of unity $e^{2\pi/n} = \cos(2\pi/n) + i \sin(2\pi/n)$.

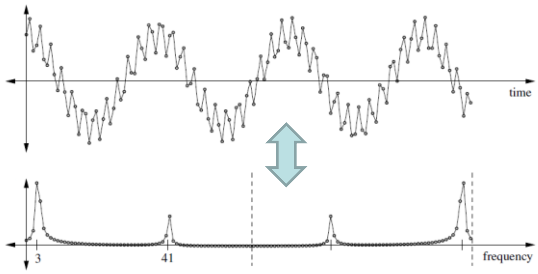
Similarity between two vectors X and Y (say, each of unit length) is often measured using their **correlation:**

$$X \bullet Y = \sum_i X_i Y_i = \cos \Theta_{XY}$$



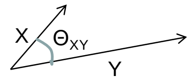
34

Signal Processing: Frequency Space Interpretation of DFT Output

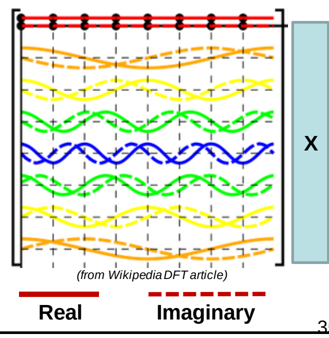


Similarity between two vectors X and Y (say, each of unit length) is often measured using their **correlation:**

$$X \bullet Y = \sum_i X_i Y_i = \cos \Theta_{XY}$$



$$\begin{bmatrix} \omega_n^{0\cdot 0} & \omega_n^{0\cdot 1} & \omega_n^{0\cdot 2} & \dots & \omega_n^{0\cdot (n-1)} \\ \omega_n^{1\cdot 0} & \omega_n^{1\cdot 1} & \omega_n^{1\cdot 2} & \dots & \omega_n^{1\cdot (n-1)} \\ \omega_n^{2\cdot 0} & \omega_n^{2\cdot 1} & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot (n-1)} \\ \vdots & & & & \vdots \\ \omega_n^{(n-1)\cdot 0} & \omega_n^{(n-1)\cdot 1} & \omega_n^{(n-1)\cdot 2} & \dots & \omega_n^{(n-1)\cdot (n-1)} \end{bmatrix} \; x \; =$$



The Fourier matrix F_n. A discrete Fourier Transform (DFT) is equivalent to multiplying a vector by this matrix. Here, ω is the complex nth root of unity $e^{2\pi/n} = \cos(2\pi/n) + i \sin(2\pi/n)$.