Divide and Conquer
& Randomized Algorithms

CS218, Winter 2020

Outline

• Intro
• Integer Multiplication (Karatsuba)
• Randomized and Deterministic Select
• Polynomial Multiplication
• The DFT and the FFT Algorithm
• Randomized Polynomial Verification
• Integer Multiplication (FFT-based)
Intro

Divide and Conquer

- Divide: If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets
- Recur: Use divide and conquer to solve the subproblems associated with the data subsets
- Conquer: Take the solutions to the subproblems and “merge” these solutions into a solution for the original problem
Divide-and-Conquer Example

a problem of size $n$

subproblem 1 of size $n/k$

subproblem 2 of size $n/k$

... 

subproblem k of size $n/k$

a solution to subproblem 1

a solution to subproblem 2

... 

a solution to subproblem k

= a solution to the original problem

Integer multiplication (Karatsuba)
Integer multiplication

- Given positive integers $y$, $z$, compute $x = y \times z$
- A naïve multiplication algorithm is below

```python
def naive_mul(y, z):
    x = 0
    while z > 0:
        if z % 2 == 1:
            x += y
        y *= 2
        z /= 2
    return x
```

Remark: these two operations can be implemented as $O(1)$ shifts

Integer multiplication

Addition takes $O(n)$ bit operations, where $n$ is the number of bits in $y$ and $z$. The naïve multiplication algorithm takes $O(n)$ $n$-bit additions. Therefore, the naïve multiplication algorithm takes $O(n^2)$ bit operations.

Can we multiply using fewer bit operations?
Integer multiplication

Suppose $n$ is a power of 2. Divide $y$ and $z$ into two halves, each with $n/2$ bits.

\[
\begin{array}{c|c|c}
  y & a & b \\
  z & c & d \\
\end{array}
\]

Integer multiplication

Then

\[
y = a2^{n/2} + b \\
z = c2^{n/2} + d
\]

and so

\[
yz = (a2^{n/2} + b)(c2^{n/2} + d) = ac2^n + (ad + bc)2^{n/2} + bd
\]
Integer multiplication

This computes $yz$ with 4 multiplications of $n/2$ bit numbers, and some additions and shifts. Running time given by $T(1) = c$, $T(n) = 4T(n/2) + dn$, which has solution $O(n^2)$ by the General Theorem. No gain over naive algorithm!

**Example 5.7:** Consider the recurrence

\[
T(n) = 4T(n/2) + n.
\]

In this case, $n^{\log_2 4} = n^2$. Thus, we are in Case 1, for $f(n)$ is $O(n^{2-\varepsilon})$ for $\varepsilon = 1$. This means that $T(n)$ is $\Theta(n^2)$ by the master method.

Integer multiplication (Karatsuba’s algorithm)

- Consider the product
  \[(a + b)(d + c) = (ad + bc) + (ac + bd)\]
- It contains two of the products we need, namely $ad$ and $bc$
- Then
  \[yz = ac2^n + [(a + b)(d + c) - (ac + bd)]2^{n/2} + bd\]
- We need three multiplications of $n/2$ bits and $O(n)$ additional work
Integer multiplication (Karatsuba’s algorithm)

Therefore,

\[ T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  3T(n/2) + dn & \text{otherwise}
\end{cases} \]

where \( c, d \) are constants.

Therefore, by our general theorem, the divide and conquer multiplication algorithm uses

\[ T(n) = O(n^{\log 3}) = O(n^{1.59}) \]

bit operations.

Karatsuba’s algorithm

```python
def multiply(y, z):
    l = max(len(y), len(z))
    if l == 1:
        return [y[0] * z[0]]
    y = [0 for i in range(len(y), l)] + y;
    z = [0 for i in range(len(z), l)] + z;
    m0 = (l + 1) / 2
    a = y[:m0]
    b = y[m0:]
    c = z[:m0]
    d = z[m0:]
```

Remark: pad \( y \) and \( z \) so that they have the same length
Karatsuba’s algorithm (continued)

\[
p_0 = \text{multiply}(a, c) \\
p_1 = \text{multiply}(\text{add}(a, b), \text{add}(c, d)) \\
p_2 = \text{multiply}(b, d)
\]

\[
z_0 = p_0 \\
z_1 = \text{subtract}(p_1, \text{add}(p_0, p_2)) \\
z_2 = p_2
\]

\[
z_0 \text{ prod} = z_0 + [0 \text{ for } i \text{ in range}(0, l)] \\
z_1 \text{ prod} = z_1 + [0 \text{ for } i \text{ in range}(0, l / 2)]
\]

\[
\text{return add}(\text{add}(z_0 \text{ prod}, z_1 \text{ prod}), z_2)
\]

Remark: compute \(z_1 = p_1 - p_0 - p_2\)

Remark: compute \(z_0 2^l + z_1 2^{l/2} + z_2\)

Randomized Select
Selection problem

- **Problem:** Select the $i$-th smallest element in an unsorted array of size $n$  
  (assume distinct elements)
- **Trivial solution:** sort $A$, select $A[i]$  
  time complexity is $O(n \log n)$
- **Can we do it in linear time?**
- **Expected** linear time can be achieved using randomization + divide & conquer

RandSelect ($A$, rank)

1. pick a pivot element at random from $A$
2. partition array around pivot, splitting it into two arrays $L$  
  (elements smaller than pivot) and $R$ (elements bigger than pivot)
3. $k \leftarrow |L| + 1$
   - if ($rank = k$) then return pivot
   - else if ($rank < k$) then RandSelect ($L$, rank)
   - else RandSelect ($R$, rank – $k$)

Randomized select
Python implementation

```python
def randselect(a, rank):
    n = len(a)
    if n <= 1:
        return a[0]
    pivot = a[random.randint(0, n-1)]
    L, R = [], []
    for x in a:
        if x < pivot:
            L += [x]
        else:
            R += [x]
    if rank <= len(L):
        return randselect(L, rank)
    else:
        return randselect(R, rank - len(L))
```

Example

Let us run RandSelect(A, 11), where

A={12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 5, 18, 20, 33, 16, 33, 21, 30, 3, 47}

Suppose the random pivot=17
Example

After partitioning

\[ L = \{12, 0, 3, 4, 3, 2, 12, 5, 16, 3\} \]

\[ L \text{ contains } 10 \text{ elements smaller than } 17 \]

\[ \{17\} \text{ this is the } 11\text{-th smallest} \]

\[ R = \{34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47\} \]

\[ R \text{ contains } 17 \text{ elements bigger than } 17 \]

In this case, we have found the element; if we did not, we would have to recurse on \( L \) or \( R \)

Randomized select: Analysis

- The objective of the partition is to shrink the number of elements from \( |A| \) to \( \max\{|L|,|R|\} \), but the value \( \max\{|L|,|R|\} \) depends on the choice of the pivot
- (Lucky) the pivot is always the median, we get
  \[ T(n)=T(n/2)+O(n) \Rightarrow T(n) \text{ is } O(n) \]
- (Unlucky) the pivot is always the largest (or the smallest) element in \( A \); if we are looking for the median \( (rank=n/2) \), we get
  \[ T(n)=n+(n-1)+(n-2)+\ldots+n/2 \Rightarrow T(n) \text{ is } O(n^2) \]
Randomized select: Analysis

• We say that the pivot is “good” if it lies with the 25\textsuperscript{th} and the 75\textsuperscript{th} percentile of the array $A$
• A randomly chosen pivot has probability $p=0.5$ of being good
• Observation: if pivot is good, then $\max\{|L|,|R|\} \leq \frac{3}{4} |A|$
• How many pivots do we need to pick on average before getting a good one?

Randomized select: Analysis

• **Lemma**: On average a fair coin ($p=1/2$) need to be tossed twice before a “heads” is seen
• **Proof**: Let $X$ be the r.v. corresponding to the number of tosses needed before coming up heads, and $E[X]$ its expectation. We get heads with probability $p$ and tails with probability $1-p$. If get heads, we are done; otherwise we need to start again and do another $E[X]$ on average (hence we do a total of $E[X]+1$ tosses with probability $1-p$). Therefore $E[X] = p*1+(1-p)(E[X]+1) = 1+(1-p)E[X]$, which gives $E[X]=1/p$. When $p=1/2$, $E[X]=2$. 
Randomized select: Analysis

- According to the Lemma, on average after two split operations we will get a good pivot and thus shrink $A$ to at most $\frac{3}{4}|A|$
- Let $T(n)$ the expected running time on an array of size $n$, we have
  $$T(n) \leq T(3n/4) + O(n)$$
  which implies that $T(n)$ is $O(n)$ (MT case III)
- Example of a Las Vegas algorithm: always returns the correct answer, and has “low” expected running time

Deterministic Select
Linear-time select

- **Problem**: Select the \(i\)-th smallest element in an unsorted array of size \(n\) (assume distinct elements)
- We can solve this problem in expected linear time using randomization
- Can we do it in deterministic linear time? Yes, thanks to Blum, Floyd, Pratt, Rivest, and Tarjan

Linear-time select

\[\text{Select} \ (A, \ rank)\]

1. divide array \(A\) into \([n/5]\) groups of size 5 (and one leftover group if \(n \mod 5\) is not 0)
2. find the median of each group of size 5 by sorting the groups of 5 and then picking the middle element
3. call \text{Select} recursively to find \text{median}, the median of the \([n/5]\) medians
4. partition array around \text{median}, splitting it into two arrays \(L\) (elements smaller than \text{median}) and \(R\) (elements bigger than \text{median})
5. \(k \Leftarrow |L| + 1\)
   - if \((\text{rank} = k)\) then return \text{median}
   - else if \((\text{rank} < k)\) then \text{Select} \((L, \text{rank})\)
   - else \text{Select} \((R, \text{rank} - k)\)

\([r]\) means the ceiling (rounding to the next integer) of real number \(r\)
Python implementation

```python
def selection(a, rank):
    n = len(a)
    if n <= 5:
        return rank_by_sorting(a, rank)
    medians = [rank_by_sorting(a[i:i+5], 3)
                for i in range(0, n-4, 5)]
    median = selection(medians, (len(medians) + 1) // 2)
    L, R = [], []
    for x in a:
        if x < median:
            L += [x]
        else:
            R += [x]
    if rank <= len(L):
        return selection(L, rank)
    else:
        return selection(R, rank - len(L))
```

Example

Let us run Select(A,11), where

A={12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2, 19, 12, 5, 18, 20, 33, 16, 33, 21, 30, 3, 47}

Note that the elements in this example are not distinct.
**Example**

First make groups of 5

<table>
<thead>
<tr>
<th>12</th>
<th>4</th>
<th>43</th>
<th>2</th>
<th>20</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>17</td>
<td>82</td>
<td>19</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>32</td>
<td>25</td>
<td>12</td>
<td>16</td>
<td>47</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>33</td>
<td>12</td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>34</td>
<td>18</td>
<td>21</td>
<td>20</td>
</tr>
</tbody>
</table>

Then find medians in each group

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>25</th>
<th>2</th>
<th>20</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>34</td>
<td>12</td>
<td>21</td>
<td>47</td>
</tr>
<tr>
<td>34</td>
<td>32</td>
<td>43</td>
<td>19</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>82</td>
<td>18</td>
<td>33</td>
<td>16</td>
</tr>
</tbody>
</table>
Example

Then find median of medians

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>25</td>
<td>2</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>34</td>
<td>12</td>
<td>21</td>
<td>47</td>
</tr>
<tr>
<td>34</td>
<td>32</td>
<td>43</td>
<td>19</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>82</td>
<td>18</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

12,12,17,21,34,30

Example

Use 17 as the pivot value and partition original array

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>25</td>
<td>2</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>27</td>
<td>5</td>
<td>16</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>17</td>
<td>34</td>
<td>12</td>
<td>21</td>
<td>47</td>
</tr>
<tr>
<td>34</td>
<td>32</td>
<td>43</td>
<td>19</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>28</td>
<td>82</td>
<td>18</td>
<td>33</td>
<td></td>
</tr>
</tbody>
</table>

12,12,17,21,34,30
Example

After partitioning

\[ L = \{12, 0, 3, 4, 3, 2, 12, 5, 16, 3\} \]

\[ L \text{ contains 10 elements smaller than 17} \]

\[ \{17\} \text{ this is the 11-th smallest} \]

\[ R = \{34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47\} \]

\[ R \text{ contains 17 elements bigger than 17} \]

Linear-time select: Analysis

- Finding the median of medians guarantees that \( x \) causes a “good split”
- At least a constant fraction of the \( n \) elements \( \leq median \) and a constant fraction \( > median \)
- **Worst-case analysis**: we need to find the worst case for the size of \( L \) and \( R \)
Linear-time select: Analysis

Observation: At least half of the medians found in Step 2 are greater than median. So at least half of the \( n/5 \) groups contribute three elements that are bigger than median, except for the one group with less than five elements and the group with median itself.

\[
3\left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \geq \left( \frac{3n}{10} \right) - 6
\]

So worst-case split has \( \left( \frac{7n}{10} \right) + 6 \) elements in "big" section of the problem, that is

\[
\max\{|L|, |R|\} < \left( \frac{7n}{10} \right) + 6
\]
Linear-time select: Analysis

Running Time:
1. $O(n)$ (break into groups of 5)
2. $O(n)$ (sorting 5 numbers and finding median is $O(1)$ time)
3. $T([n/5])$ (recursive call to find median of medians)
4. $O(n)$ (partition is linear time)
5. $T(7n/10 + 6)$ (maximum size of subproblem)

Recurrence relation

$$T(n) = T([n/5]) + T(7n/10 + 6) + O(n) \quad n > 140$$
$$= \Theta(1) \quad n \leq 140$$

Linear-time select: Analysis

Fact: $T(n) = T([n/5]) + T(7n/10 + 6) + O(n)$ is $O(n)$

Proof:
Base case: easy (omitted).

$$T(n) = T([n/5]) + T(7n/10 + 6) + O(n)$$
$$\leq c[n/5] + c(7n/10 + 6) + O(n)$$
$$\leq c((n/5) + 1) + 7cn/10 + 6c + O(n)$$
$$= cn - [c(n/10 - 7) - dn]$$
$$\leq cn$$

Inequality $cn/10 - 7c - dn \geq 0$ is equivalent to $c \geq 10dn/(n-70)$ when $n > 70$. We can assume that $n \geq 140$, so that $n/(n-70) \geq 2$. In that case, choosing $c \geq 20d$ will satisfy the inequality (there is nothing special about choosing $n \geq 140$, a different choice of $n > 70$ will require to choose a different $c$)
DFT & FFT

Polynomials

\[ A(x) = \sum_{i=0}^{n-1} a_i x^i \]

or

\[ A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} \]

\( A(x) \) is of degree \( i \) if \( a_i \) is the highest non-zero coefficient

Operations: addition, multiplication, evaluation
Polynomial Multiplication Problem

- **Problem**: Given coefficients \((a_0,a_1,a_2,...,a_{n-1})\) and \((b_0,b_1,b_2,...,b_{n-1})\) defining polynomials \(A(x)\) and \(B(x)\) respectively, compute \(A(x)B(x)\)
- We have \(C(x) = A(x)B(x) = \sum_{i=0}^{2n-2} c_i x^i\)
  where \(c_i = \sum_{j=0}^{n} a_j b_{i-j}\)
  
  \(c\) is called the *discrete convolution* of \(a\) and \(b\)
- A naïve algorithm takes \(O(n^2)\) time
- Karatsuba-style algorithm takes \(O(n^{\log_2 3})\) time
- FFT can solve this problem in \(O(n \log n)\) time

**Example**

\[
A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3
\]
\[
B(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3
\]

\[
C(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0) x + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2 + (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) x^3 + (a_1 b_1 + a_2 b_0 + a_2 b_1 + a_3 b_1) x^4 + (a_2 b_3 + a_3 b_2) x^5 + a_3 b_3 x^6
\]
Example

\[ A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 \]
\[ B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 \]

\[ C(x) = a_0b_0 + (a_0b_1 + a_1b_0)x + (a_0b_2 + a_1b_1 + a_2b_0)x^2 + (a_0b_3 + a_1b_2 + a_2b_1 + a_3b_0)x^3 + (a_1b_3 + a_2b_2 + a_3b_1)x^4 + (a_2b_3 + a_3b_2)x^5 + a_3b_3x^6 \]
Example

\[ A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \]
\[ B(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \]

\[ C(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0) x + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2 +
(a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) x^3 + (a_1 b_3 + a_2 b_2 + a_3 b_1) x^4 +
(a_2 b_3 + a_3 b_2) x^5 + a_3 b_3 x^6 \]
Example

\[ A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \]
\[ B(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3 \]
\[ C(x) = a_0 b_0 + (a_0 b_1 + a_1 b_0) x + (a_0 b_2 + a_1 b_1 + a_2 b_0) x^2 + (a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0) x^3 + (a_1 b_3 + a_2 b_2 + a_3 b_1) x^4 + (a_2 b_3 + a_3 b_2) x^5 + a_3 b_3 x^6 \]
Circular discrete convolution

\[(a \otimes b)_i = \sum_{j=0}^{n-1} a_j b_{(i-j \mod n)}\]

\[
\begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
  b_0 & b_3 & b_2 & b_1 \\
  \end{array}
\]

\[
\begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
  b_1 & b_0 & b_3 & b_2 \\
  \end{array}
\]

\[
\begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
  b_2 & b_1 & b_0 & b_3 \\
  \end{array}
\]

\[
\begin{array}{cccc}
  a_0 & a_1 & a_2 & a_3 \\
  b_3 & b_2 & b_1 & b_0 \\
  \end{array}
\]

Representing a polynomial

- Coefficient representation \((a_0, a_1, a_2, \ldots, a_{n-1})\)
  - Convenient for certain operations, e.g., evaluation of \(A(x)\) at a specific value \(x_0\)

- Point-value representation \(\{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}\) where all \(x_k\) are distinct
  - Convenient for polynomial multiplication
Horner’s rule

• **Problem**: compute $A(x_0)$
• Given coefficients $(a_0, a_1, a_2, ..., a_{n-1})$, defining polynomial

$$A(x) = \sum_{i=0}^{n-1} a_i x^i$$

and given $x_0$, we can evaluate $A(x_0)$ in $\Theta(n)$ time using the equation

$$A(x_0) = a_0 + x_0 (a_1 + x_0 (a_2 + \cdots + x_0 (a_{n-2} + x_0 a_{n-1}) \cdots))$$

Point-value representation

• Instead of representing a polynomial by its coefficients we represent it by listing a set of $n$ points $\{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ where all $x_k$ are distinct
• In this representation, polynomial multiplication is $O(n)$
• Is this representation unique, i.e., does it uniquely identify a polynomial?
Point-value representation

![Graph of a polynomial function and points](image)

Interpolation theorem

- **Theorem**: For any set
  \[
  \{(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})\}
  \]
  of \(n\) point-value pairs such that all the \(x_k\) values are distinct, there is a unique polynomial \(A(x)\) of degree at most \(n-1\) such that \(y_k = A(x_k)\) for \(k = 0, 1, \ldots, n-1\)
Alternate algorithm for $C(x) = A(x)B(x)$

- Evaluate $A(x)$ on $2n$ $x$-values, $x_0, x_1, \ldots, x_{2n-1}$
- Evaluate $B(x)$ on $x_0, x_1, \ldots, x_{2n-1}$
- Compute $S = \{(x_0, A(x_0)B(x_0)), (x_1, A(x_1)B(x_1)), \ldots, (x_{2n-1}, A(x_{2n-1})B(x_{2n-1}))\}$ in $O(n)$ time
- Find the unique polynomial $C(x)$ that goes through the set $S$ of $2n$ points (interpolation)
A(x)

B(x)

x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8

A(x)B(x)

B(x)

A(x)

60

61
Alternate algorithm for $C(x) = A(x)B(x)$

• The conversion from coefficient to point-value representation takes $O(n^2)$ time because we need to use $O(n)$-time Horner’s Rule evaluation to $2n$ distinct points
• We have the freedom to choose the points
• FFT can do this in $O(n \log n)$ time, by picking $2n$ points that are “easy” to evaluate $\omega^{0}_{2n}, \omega^{1}_{2n}, \omega^{2}_{2n}, \ldots, \omega^{2^{n-1}}_{2n}$
• What are these special points?

$O(n \log n)$ polynomial-multiplication algorithm

![Diagram of the algorithm]
Algorithm outline

1. **Double degree-bound**: pad with zeroes the high degree coefficients of $A$ and $B$ to have $2n$ coefficients
2. **Evaluate**: compute point-value representations of $A(x)$ and $B(x)$ of length $2n$ through two applications of the FFT - fast evaluation using the $(2n)^{th}$ roots of unity
3. **Point-wise multiply**: compute the point-value representation of the product by multiplying the values of the two polynomials at the roots of unity
4. **Interpolate**: invert the DFT on the vector of the $2n$ value-pairs just computed

Complex Roots of Unity

- A **complex $n$-th root of unity** is a complex number $\omega_n$ such that $\left(\omega_n\right)^n = 1$
- There are exactly $n$ complex $n$-th roots of unity they are $\omega_k = e^{\frac{2\pi ik}{n}}$, where $k=0,1,...,n-1$
- Recall that $e^{iu} = \cos(u) + i \sin(u)$ [Euler]
Properties of Complex Roots of Unity

[Cancellation Lemma] For any integers $n \geq 0$, $k \geq 0$ and $d > 0$, $\omega_{dn}^d = \omega_n^k$.

Proof. From the definition of $\omega_n = e^{2\pi i/n}$,

$$\omega_{dn}^d = \left(e^{2\pi i/dn}\right)^d = \left(e^{2\pi i/n}\right)^k = \omega_n^k.$$  

[Corollary] For any even integer $n > 0$, $\omega_n^{n/2} = \omega_2 = -1$.

Properties of Complex Roots of Unity

[Halving Lemma] If $n > 0$ is even, then the squares of the $n$ complex $n^{th}$ roots of unity are the $n/2$ complex $(n/2)^{th}$ roots of unity.

Proof. The cancellation lemma gives $(\omega_n^k)^2 = \omega_n^{2k} = \omega_{n/2}^k$ for every integer $k \geq 0$. We need to show that all roots are accounted for: if we square all complex $n^{th}$ roots of unity, each $(n/2)^{th}$ root of unity appears exactly twice $(\omega_n^{k+n/2})^2 = \omega_n^{2k+n} = \omega_n^{2k}\omega_n^n = \omega_n^{2k} = (\omega_n^k)^2$.  

67
Discrete Fourier Transform

- We can now define the **Discrete Fourier Transform (DFT)**. Evaluate the polynomial

\[ A(x) = \sum_{j=0}^{n-1} a_j x^j \]

of degree bound \( n \) at the \( n \)th roots of unity

\[ \omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1} \]

We can assume \( n \) is a power of 2 (if not, just pad with zero coefficients). For each \( k, 0 \leq k \leq n-1 \),

\[ y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}, \]

is the \( k \)th component of the DFT of

\[ a = (a_0, a_1, \ldots, a_{n-1}). \]

---

Fast Fourier Transform (FFT)

- The divide and conquer algorithm FFT is based on splitting the original polynomial into its even powers part and odd powers part

\[ A(x) = A^{[0]}(x^2) \cdot x A^{[1]}(x^2), \]

where

\[ A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1} \]

\[ A^{[1]}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1} \]

- The problem of evaluating \( A(x) \) reduces to
  - Evaluating two \( n/2 \)-degree polynomials \( A^{[0]} \) and \( A^{[1]} \) at points \( \{\omega_n^0, \omega_n^1, \omega_n^2, \ldots, \omega_n^{n-1}\} \)
  - Combining the results
Fast Fourier Transform (FFT)

- Note that by the halving lemma \( (\omega_n^0)^2, (\omega_n^1)^2, (\omega_n^2)^2, \ldots, (\omega_n^{n-1})^2 \) consists of only \( n/2 \) complex \((n/2)^{th}\) roots of unity with each root occurring exactly twice.
- Evaluating \( A(x) \) at the \( n^{th} \) roots of unity reduces to the evaluation of \( A^{[0]}(x) \) and \( A^{[1]}(x) \) at the \( n/2 \) \((n/2)^{th}\) roots of unity, followed by a multiplication and an addition.
- The subproblems have exactly the same form as the original problem, but half the size.

Recursive FFT

```plaintext
RECURSIVE-FFT(a)
1  n ← length[a]  ⊳ n is a power of 2.
2  if n = 1
3      then return a
4  ωn ← e^{2πi/n}
5  ω ← 1
6  a^{[0]} ← (a_0, a_2, \ldots, a_{n-2})
7  a^{[1]} ← (a_1, a_3, \ldots, a_{n-1})
8  y^{[0]} ← RECURSIVE-FFT(a^{[0]})
9  y^{[1]} ← RECURSIVE-FFT(a^{[1]})
10 for k ← 0 to n/2 − 1
11    do y_k ← y_k^{[0]} + ω y_k^{[1]}
12    y_{k+(n/2)} ← y_k^{[0]} − ω y_k^{[1]}
13    ω ← ω \omega_n
14 return y  ⊳ y is assumed to be a column vector.
```
Recursion Tree

Time complexity Analysis

• Each call costs $O(n)$ - by the “splits” at lines 6 and 7, and by the “gluing things back together” loop at lines 10-13 plus the cost of the two recursive calls
• Recurrence relation: $T(n)=2T(n/2)+\Theta(n)$
• Solution: $T(n)$ is $\Theta(n \log n)$
Correctness

The termination at lines 2-3 is obvious: the DFT of a constant must be the same constant, so we simply return the coefficients.

At lines 8-9 we have two $n/2$-vectors $y[0]$ and $y[1]$. We must construct the correct $n$-vector. Notice that we have from $\text{DFT}_{n/2}$ and $k = 0, 1, \ldots, n/2-1$:

\[
y_k^0 = A^0[\omega_{n/2}^k], \quad y_k^1 = A^1[\omega_{n/2}^k]
\]

which, by the cancellation lemma, can be rewritten as

\[
y_k^0 = A^0[\omega_n^{2k}], \quad y_k^1 = A^1[\omega_n^{2k}]
\]

Correctness

For $y_0, y_1, \ldots, y_{n/2-1}$, line 11 yields

\[
y_k = y_k^0 + \omega_n^k y_k^1 = A^0[\omega_n^{2k}] + \omega_n^k A^1[\omega_n^{2k}] = A[\omega_n^k].
\]

This provides us with the first half of the result set. Regarding the other $y_{n/2}, y_{n/2+1}, \ldots, y_{n-1}$ elements.

First, observe that $\omega_n^{k+(n/2)} = -\omega_n^k$.

Line 12 gives, for $k = 0, 1, \ldots, n/2-1$

\[
y_{k+(n/2)} = y_k^0 - \omega_n^k y_k^1 = y_k^0 + \omega_n^{k+(n/2)} y_k^1 = A^0[\omega_n^{2k}] + \omega_n^{k+(n/2)} A^1[\omega_n^{2k}] = A[\omega_n^{k+(n/2)}]
\]
Inverse DFT

- We have an $\Theta(n \log n)$ way of moving from the coefficient domain to the point-value domain.
- How do we come back?
- We need convert from the point-value form back to the coefficient form by interpolation.

Inverse DFT

Note that we can write the DFT as a matrix product $y = V_n a$, as follows:

$$
\begin{pmatrix}
y_0 \\
y_1 \\
y_2 \\
y_3 \\
\vdots \\
y_{n-1}
\end{pmatrix} =
\begin{pmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\
1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\
1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)}
\end{pmatrix}
\begin{pmatrix}
a_0 \\
a_1 \\
a_2 \\
a_3 \\
\vdots \\
a_{n-1}
\end{pmatrix}
$$

The $(k, j)$ entry of $V_n$ is $\omega_n^{kj}$ for $j = 0, 1, \ldots, n-1$, and the exponents of the entries of $V_n$ form a multiplication table. The inverse DFT can be computed as $a = V_n^{-1} y$. We show how $V_n^{-1}$ can be obtained next.
Properties of Complex Roots of Unity

[Summation Lemma] For any integer \( n \geq 1 \) and \( k \geq 0 \) not divisible by \( n \),
\[
\sum_{j=0}^{n-1} (\omega_n^k)^j = 0.
\]

**Proof.** As long as \( \omega_n^k \neq 1 \), which holds since \( k \) is not divisible by \( n \), we simply sum a geometric progression:
\[
\sum_{j=0}^{n-1} (\omega_n^k)^j = (\omega_n^k)^n - 1 \quad \frac{\omega_n^k - 1}{\omega_n - 1} = \frac{1^k - 1}{\omega_n - 1} = 0.
\]

Inverse DFT

**Theorem.** For \( j, k = 0, 1, \ldots, n-1 \), the \((j, k)\) entry of \( V_n^{-1} \) is \( \omega_n^{kj}/n \).

**Proof.** We show that the product of the proposed inverse with \( V_n \) is \( I_n \). Consider the \((j, j')\) entry of \( V_n^{-1}V_n \):
\[
[V_n^{-1}V_n]_{j,j'} = \sum_{k=0}^{n-1} (\omega_n^{kj}/n)(\omega_n^{kj'}) = \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n.
\]

The summation equals 1 if \( j' = j \), and 0 otherwise (when we can apply the summation lemma). We observe that \(-(n-1) \leq j' - j \leq n - 1 \), so that \( j' - j \) is not divisible by \( n \), and the summation lemma applies.
Direct and Inverse DFT

\[ V_n = \begin{pmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{(n-1)} \\
1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(n-1)} \\
1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(n-1)} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{(n-1)} & \omega^{2(n-1)} & \omega^{3(n-1)} & \cdots & \omega^{(n-1)(n-1)}
\end{pmatrix} \]

\[ V_n^{-1} = \frac{1}{n} \begin{pmatrix}
1 & 1 & 1 & 1 & \cdots & 1 \\
1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \cdots & \omega^{-(n-1)} \\
1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \cdots & \omega^{-(2(n-1))} \\
1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \cdots & \omega^{-(3(n-1))} \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \omega^{-(n-1)} & \omega^{-(2(n-1))} & \omega^{-(3(n-1))} & \cdots & \omega^{-(n-1)(n-1)}
\end{pmatrix} \]

Inverse FFT algorithm

- Now that we know the exact form of \( V_n^{-1} \), we have
  \[ DFT^{-1}_n(y)_j = a_j = \frac{1}{n} \sum_{k=0}^{n-1} y_k \omega^{-kj}_n \text{ for } j = 0, 1, \ldots, n-1 \]

- Simple modification of the FFT algorithm
  1. switch the roles of \( a \) and \( y \)
  2. replace \( \omega_n \) with \( \omega_n^{-1} \)
  3. divide each element of the result by \( n \) (at the end)
- The inverse DFT can be computed in \( \Theta(n \log n) \) time
Python implementation

def fft(a):
    """The length of array a must be a power of two""
    n = len(a)
    if n == 1:
        return a
    y0 = fft(a[::2])
    y1 = fft(a[1::2])
    w, wn, y = 1, math.e ** (2j * math.pi / n), []
    for k in range(n // 2):
        y += [y0[k] + w * y1[k]]
        w *= wn
    w = 1
    for k in range(n // 2):
        y += [y0[k] - w * y1[k]]
        w *= wn
    return y

Python implementation

def inverse_fft(a):
    """The length of array a must be a power of two""
    n = len(a)
    if n == 1:
        return a
    y0 = inverse_fft(a[::2])
    y1 = inverse_fft(a[1::2])
    w, wn, y = 1, math.e ** (-2j * math.pi / n), []
    for k in range(n // 2):
        y += [y0[k] + w * y1[k]]
        w *= wn
    w = 1
    for k in range(n // 2):
        y += [y0[k] - w * y1[k]]
        w *= wn
    y = [j/2 for j in y]
    return y
Integer multiplication (FFT)

Integer Arithmetic

Multiply two $n$-digit binary numbers $a = a_{n-1} \ldots a_1a_0$, $b = b_{n-1} \ldots b_1b_0$.

- Form two degree $n$ polynomials with $a$ and $b$ as coefficients
- Observe that $a = A(2)$, $b = B(2)$
- Compute product using FFT in $O(n \log n)$ steps

\[ C(x) = A(x) \times B(x) \]

- Evaluate $C(2)$ in linear time, gives $a \times b$
FYI: integer arithmetic

- Instead of using complex numbers, which can result in loss of precision, use modular arithmetic
- Find a generator $x$ of $Z^*_p$, where $Z^*_p$ are the elements of $\{0, 1, \ldots, p-1\}$ which are relatively prime to $p$
- Then $\omega = x^c \mod p$ is a primitive root of unity in $Z^*_p$

Randomized Polynomial Verification
Verifying Polynomial Multiplication

• **Problem:** Given two polynomials $A(x)$ and $B(x)$ of degree $n-1$, and $C(x)$ of degree $2n-2$, verify that $C(x) = A(x)B(x)$
• We could compute $A(x)B(x)$ in $O(n \log n)$ time
• Consider the following randomized algorithm
  – 1) **choose** a $x$ at random from the set \{0,1,…, 100n-1\}
  – 2) **return true** if $A(x)B(x)-C(x)=0$, **false** otherwise
• The algorithm is $O(n)$ but is not always correct: it will never report false if the answer is yes, but it might report true when the answer is no

Verifying Polynomial Multiplication

• The algorithm will make a mistake when $x$ is a root of the polynomial $AB-C$, which has degree at most $2n-2$
• Since $AB-C$ can only have at most $2n-2$ roots, it follows that *(Schwartz–Zippel lemma)*
  Probability(wrong answer) = $(2n-2)/(100n) < 0.02$
• This is an example of a *Monte Carlo randomized algorithm*: runs quickly, and returns the correct answer with high probability (but not always)
• We can decrease the probability of error to any desired level by repeating the algorithm multiple times
Reading assignment

• Section 9.2, “Selection in expected linear time”
• Section 9.3, “Selection in worst-case linear time”
• Chapter 30, “Polynomials and the FFT”