CSE 421: Introduction to Algorithms

Divide and Conquer

Paul Beame

Algorithm Design Techniques

- Divide & Conquer
  - Reduce problem to one or more sub-problems of the same type
  - Typically, each sub-problem is at most a constant fraction of the size of the original problem
    - e.g. Mergesort, Binary Search, Strassen’s Algorithm, Quicksort (kind of)

Fast exponentiation

- **Power**($a,n$)
  - **Input:** integer $n$ and number $a$
  - **Output:** $a^n$

- Obvious algorithm
  - $n$-1 multiplications

- Observation:
  - if $n$ is even, $n=2m$, then $a^n=a^m\cdot a^m$

Divide & Conquer Algorithm

- **Power**($a,n$)
  - if $n=0$ then return($1$)
  - else if $n=1$ then return($a$)
  - else
    - $x \leftarrow$ **Power**($a,\lfloor n/2 \rfloor$)
    - if $n$ is even then
      - return($x\cdot x$)
    - else
      - return($a\cdot x\cdot x$)
Analysis

- Worst-case recurrence
  - \( T(n) = T(\lfloor n/2 \rfloor) + 2 \) for \( n \geq 1 \)
  - \( T(1) = 0 \)
- Time
  - \( T(n) = T(\lfloor n/2 \rfloor) + 2 \leq T(\lfloor n/4 \rfloor) + 2 + 2 \leq \cdots \leq T(1) + 2 + \cdots + 2 = 2 \log_2 n \) copies
- More precise analysis:
  - \( T(n) = \lceil \log_2 n \rceil + \# \text{ of } 1\text{'s in } n\text{'s binary representation} \)

A Practical Application- RSA

- Instead of \( a^n \) want \( a^n \mod N \)
  - \( a^{i+j} \mod N = (a^i \mod N) \cdot (a^j \mod N) \mod N \)
  - same algorithm applies with each \( x \cdot y \) replaced by
    - \( ((x \mod N) \cdot (y \mod N)) \mod N \)
- In RSA cryptosystem (widely used for security)
  - need \( a^n \mod N \) where \( a, n, N \) each typically have 1024 bits
  - Power: at most 2048 multiplies of 1024 bit numbers
    - relatively easy for modern machines
  - Naive algorithm: \( 2^{1024} \) multiplies

Binary search for roots (bisection method)

- Given:
  - continuous function \( f \) and two points \( a < b \) with \( f(a) \leq 0 \) and \( f(b) > 0 \)
- Find:
  - approximation to \( c \) s.t. \( f(c) = 0 \) and \( a < c < b \)

Bisection method

Bisection\((a, b, \varepsilon)\)

- if \( (a-b) < \varepsilon \) then
  - return\((a)\)
- else
  - \( c \leftarrow (a+b)/2 \)
  - if \( f(c) \leq 0 \) then
    - return\( (Bisection(c,b,\varepsilon)) \)
  - else
    - return\( (Bisection(a,c,\varepsilon)) \)
**Time Analysis**

- At each step we halved the size of the interval
- It started at size $b-a$
- It ended at size $\varepsilon$
- \# of calls to $f$ is $\log_2( (b-a)/\varepsilon)$

**Old favorites**

- **Binary search**
  - One subproblem of half size plus one comparison
  - Recurrence $T(n) = T(\lceil n/2 \rceil) + 1$ for $n \geq 2$
  - $T(1) = 0$
  - So $T(n)$ is $\lceil \log_2 n \rceil + 1$
- **Mergesort**
  - Two subproblems of half size plus merge cost of $n-1$ comparisons
  - Recurrence $T(n) \leq 2T(\lceil n/2 \rceil) + n-1$ for $n \geq 2$
  - $T(1) = 0$
  - Roughly $n$ comparisons at each of $\log_2 n$ levels of recursion
  - So $T(n)$ is roughly $2n \log_2 n$

**Euclidean Closest Pair**

- **Given** a set $P$ of $n$ points $p_1, \ldots, p_n$ with real-valued coordinates
- **Find** the pair of points $p_i, p_j \in P$ such that the Euclidean distance $d(p_i, p_j)$ is minimized
- $\Theta(n^2)$ possible pairs
- In one dimension: easy $O(n \log n)$ algorithm
  - Sort the points
  - Compare consecutive elements in the sorted list
- What about points in the plane?

**Closest Pair in the Plane**

No single direction along which one can sort points to guarantee success!
Closest Pair In the Plane: Divide and Conquer

- Sort the points by their x coordinates
- Split the points into two sets of n/2 points L and R by x coordinate
- Recursively compute
  - closest pair of points in L, (p_L, q_L)
  - closest pair of points in R, (p_R, q_R)
- Let δ = \min\{d(p_L, q_L), d(p_R, q_R)\} and let (p, q) be the pair of points that has distance δ
- But this may not be enough
  - Closest pair of points may involve one point from L and the other from R!

A clever geometric idea

Any pair of points p ∈ L and q ∈ R with d(p, q) < δ must lie in band

No two points can be in the same green box

Only need to check pairs of points up to 2 rows apart - At most a constant # of other points!
Closest Pair Recombining

- Sort points by y coordinate ahead of time
- On recombination only compare each point in δ-band of L ∪ R to the 11 points in δ-band of L ∪ R above it in the y sorted order
  - If any of those distances is better than δ replace (p,q) by the best of those pairs
- O(n log n) for x and y sorting at start
- Two recursive calls on problems on half size
- O(n) recombination
- Total O(n log n)

Sometimes two sub-problems aren’t enough

- More general divide and conquer
  - You’ve broken the problem into a different sub-problems
  - Each has size at most n/b
  - The cost of the break-up and recombing the sub-problem solutions is O(n^k)

Recurrence
- T(n) ≤ a·T(n/b)+c·n^k

Master Divide and Conquer Recurrence

- If T(n) ≤ a·T(n/b)+c·n^k for n>b then
  - if a>b^k then T(n) is Θ(n^log_a a)
  - if a<b^k then T(n) is Θ(n^k)
  - if a=b^k then T(n) is Θ(n^k log n)
- Works even if it is ⌊n/b⌋ instead of n/b.

Proving Master recurrence
Proving Master recurrence

Problem size

\[ T(n) = a \cdot T(n/b) + c \cdot n^k \]  # probs

\[ n \]
\[ n/b \]
\[ n/b^2 \]
\[ b \]
\[ 1 \]

\[ T(1) = c \]

\[ a \]
\[ a^2 \]
\[ a^d \]

\[ d = \log_b n \]

Proving Master recurrence

Problem size

\[ T(n) = a \cdot T(n/b) + c \cdot n^k \]  # probs

\[ n \]
\[ n/b \]
\[ n/b^2 \]
\[ b \]
\[ 1 \]

\[ T(1) = c \]

\[ a \]
\[ a^2 \]
\[ a^d \]

\[ c \cdot n^k \]

Geometric Series

- \( S = t + tr + tr^2 + \ldots + tr^{n-1} \)
- \( r \cdot S = tr + tr^2 + \ldots + tr^{n-1} + tr^n \)
- \( (r-1)S = tr^n - t \)
- so \( S = t (r^n - 1)/(r-1) \) if \( r \neq 1 \).

Simple rule

- If \( r \neq 1 \) then \( S \) is a constant times largest term in series

Total Cost

- Geometric series
  - ratio \( a/b^k \)
  - \( d + 1 = \log_b n + 1 \) terms
  - first term \( cn^k \), last term \( ca^d \)
- If \( a/b^k = 1 \)
  - all terms are equal \( T(n) \) is \( \Theta(n^k \log n) \)
- If \( a/b^k < 1 \)
  - first term is largest \( T(n) \) is \( \Theta(n^k) \)
- If \( a/b^k > 1 \)
  - last term is largest \( T(n) \) is \( \Theta(a^d) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) \)
  (To see this take \( \log_b \) of both sides)
Multiplying Matrices

\[\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} = \begin{bmatrix} a_1 b_{11} & a_1 b_{12} & a_1 b_{13} & a_1 b_{14} \\ a_2 b_{21} & a_2 b_{22} & a_2 b_{23} & a_2 b_{24} \\ a_3 b_{31} & a_3 b_{32} & a_3 b_{33} & a_3 b_{34} \\ a_4 b_{41} & a_4 b_{42} & a_4 b_{43} & a_4 b_{44} \end{bmatrix} + \begin{bmatrix} a_1 b_{12} & a_1 b_{13} & a_1 b_{14} & a_1 b_{14} \\ a_2 b_{22} & a_2 b_{23} & a_2 b_{24} & a_2 b_{24} \\ a_3 b_{32} & a_3 b_{33} & a_3 b_{34} & a_3 b_{34} \\ a_4 b_{42} & a_4 b_{43} & a_4 b_{44} & a_4 b_{44} \end{bmatrix} = \begin{bmatrix} a_1 b_{11} + a_1 b_{12} + a_1 b_{13} + a_1 b_{14} \\ a_2 b_{21} + a_2 b_{22} + a_2 b_{23} + a_2 b_{24} \\ a_3 b_{31} + a_3 b_{32} + a_3 b_{33} + a_3 b_{34} \\ a_4 b_{41} + a_4 b_{42} + a_4 b_{43} + a_4 b_{44} \end{bmatrix} + \begin{bmatrix} a_1 b_{12} + a_1 b_{13} + a_1 b_{14} & a_1 b_{12} + a_1 b_{13} + a_1 b_{14} \\ a_2 b_{22} + a_2 b_{23} + a_2 b_{24} & a_2 b_{22} + a_2 b_{23} + a_2 b_{24} \\ a_3 b_{32} + a_3 b_{33} + a_3 b_{34} & a_3 b_{32} + a_3 b_{33} + a_3 b_{34} \\ a_4 b_{42} + a_4 b_{43} + a_4 b_{44} & a_4 b_{42} + a_4 b_{43} + a_4 b_{44} \end{bmatrix}
\]

- \(n^3\) multiplications, \(n^3-n^2\) additions
### Multiplying Matrices

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} & a_{14} \\
  a_{21} & a_{22} & a_{23} & a_{24} \\
  a_{31} & a_{32} & a_{33} & a_{34} \\
  a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
  b_{11} & b_{12} & b_{13} & b_{14} \\
  b_{21} & b_{22} & b_{23} & b_{24} \\
  b_{31} & b_{32} & b_{33} & b_{34} \\
  b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
= \begin{bmatrix}
  a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31}+a_{14}b_{41} \\
  a_{21}b_{11}+a_{22}b_{21}+a_{23}b_{31}+a_{24}b_{41} \\
  a_{31}b_{11}+a_{32}b_{21}+a_{33}b_{31}+a_{34}b_{41} \\
  a_{41}b_{11}+a_{42}b_{21}+a_{43}b_{31}+a_{44}b_{41}
\end{bmatrix}
\]

### Simple Divide and Conquer

\[
\begin{bmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{bmatrix}
\begin{bmatrix}
  B_{11} & B_{12} \\
  B_{21} & B_{22}
\end{bmatrix}
= \begin{bmatrix}
  A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\
  A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22}
\end{bmatrix}
\]

- \( T(n) = 8T(n/2) + 4(n/2)^2 = 8T(n/2) + n^2 \)
- \( 8 > 2^2 \) so \( T(n) \) is \( \Theta(n^{\log_2 8}) = \Theta(n^3) \)

### Strassen’s Divide and Conquer Algorithm

- Strassen’s algorithm
  - Multiply 2x2 matrices using 7 instead of 8 multiplications (and lots more than 4 additions)
  - \( T(n) = 7T(n/2) + cn^2 \)
  - \( 7 > 2^2 \) so \( T(n) \) is \( \Theta(n^{\log_2 7}) \) which is \( O(n^{2.81...}) \)

- Fastest algorithms theoretically use \( O(n^{2.376}) \) time
  - not practical but Strassen’s is practical provided calculations are exact and we stop recursion when matrix has size somewhere between 10 and 100

### The algorithm

\[
\begin{align*}
P_1 & \leftarrow A_{12}(B_{11} + B_{21}) ; & P_2 & \leftarrow A_{21}(B_{12} + B_{22}) \\
P_3 & \leftarrow (A_{11} - A_{12})B_{11} ; & P_4 & \leftarrow (A_{22} - A_{21})B_{22} \\
P_5 & \leftarrow (A_{22} - A_{12})(B_{21} - B_{22}) ; & P_6 & \leftarrow (A_{11} - A_{21})(B_{12} - B_{11}) \\
P_7 & \leftarrow (A_{21} - A_{12})(B_{11} + B_{22}).
\end{align*}
\]

\[
\begin{align*}
C_{11} & \leftarrow P_1 + P_3 ; & C_{12} & \leftarrow P_2 + P_3 + P_6 - P_7 \\
C_{21} & \leftarrow P_1 + P_4 + P_5 + P_7 ; & C_{22} & \leftarrow P_2 + P_4
\end{align*}
\]
Another Divide & Conquer Example: Multiplying Faster

If you analyze our usual grade school algorithm for multiplying numbers

- $\Theta(n^2)$ time
- On real machines each “digit” is, e.g., 32 bits long but still get $\Theta(n^2)$ running time with this algorithm when run on n-bit multiplication

We can do better!

- We’ll describe the basic ideas by multiplying polynomials rather than integers
- Advantage is we don’t get confused by worrying about carries at first

Notes on Polynomials

These are just formal sequences of coefficients

- when we show something multiplied by $x^k$ it just means shifted $k$ places to the left – basically no work

Usual polynomial multiplication

<table>
<thead>
<tr>
<th>4$x^2$ + 2$x$ + 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2$ - 3$x$ + 1</td>
</tr>
</tbody>
</table>

Polynomial Multiplication

- Given:
  - Degree n-1 polynomials $P$ and $Q$
  - $P = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-2}x^{n-2} + a_{n-1}x^{n-1}$
  - $Q = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-2}x^{n-2} + b_{n-1}x^{n-1}$

- Compute:
  - Degree 2n-2 Polynomial $PQ$
  - $PQ = a_0b_0 + (a_0b_1+a_1b_0)x + (a_0b_2+a_1b_1+a_2b_0)x^2 + \ldots + (a_{n-2}b_{n-1}+a_{n-1}b_{n-2})x^{2n-3} + a_{n-1}b_{n-1}x^{2n-2}$

- Obvious Algorithm:
  - Compute all $a_i b_j$ and collect terms
  - $\Theta(n^2)$ time

Naive Divide and Conquer

- Assume $n=2k$
  - $P = (a_0 + a_1 x + a_2 x^2 + \ldots + a_k x^k + a_{k+1} x^{k+1}) + (a_k + a_{k+1} x + \ldots + a_{n-2}x^{k-2} + a_{n-1}x^{k-1})x^k = P_0 + P_1 x^k$ where $P_0$ and $P_1$ are degree $k-1$ polynomials
  - Similarly $Q = Q_0 + Q_1 x^k$
  - $PQ = (P_0+P_1x^k)(Q_0+Q_1x^k)$
  - $P_0Q_0 + (P_1Q_0 + P_0Q_1)x^k + P_1Q_1x^{2k}$

- 4 sub-problems of size $k=n/2$ plus linear combining
  - $T(n)=4T(n/2)+cn$ Solution $T(n) = \Theta(n^2)$
Karatsuba’s Algorithm

- A better way to compute the terms
  - Compute
    - \( A \leftarrow P_0Q_0 \)
    - \( B \leftarrow P_1Q_1 \)
    - \( C \leftarrow (P_0 + P_1)(Q_0 + Q_1) = P_0Q_0 + P_1Q_0 + P_0Q_1 + P_1Q_1 \)
  - Then
    - \( P_0Q_1 + P_1Q_0 = C - A - B \)
    - So \( PQ = A + (C - A - B)x^k + Bx^{2k} \)
- 3 sub-problems of size \( n/2 \) plus \( O(n) \) work
  - \( T(n) = 3T(n/2) + cn \)
  - \( T(n) = O(n^\alpha) \) where \( \alpha = \log_2 3 \approx 1.59... \)

Karatsuba: Details

PolyMul(P, Q):

```
// P, Q are length n = 2k vectors, with P[i], Q[i] being
// the coefficient of x^i in polynomials P, Q respectively.
// Let Pzero be elements 0..k-1 of P; Pone be elements k..n-1
// Qzero, Qone : similar
if n=1 then Return(P[0]*Q[0]) else
  A ← PolyMul(Pzero, Qzero); // result is a (2k-1)-vector
  B ← PolyMul(Pone, Qone);  // ditto
  Psum ← Pzero + Pone;      // add corresponding elements
  Qsum ← Qzero + Qone;      // ditto
  C ← polyMul(Psum, Qsum); // another (2k-1)-vector
  Mid ← C – A – B;         // subtract correspond elements
  R ← A + Shift(Mid, n/2) + Shift(B, n) // a (2n-1)-vector
  Return( R);```

Multiplication

- Polynomials
  - Naïve: \( \Theta(n^2) \)
  - Karatsuba: \( \Theta(n^{1.59...}) \)
  - Best known: \( \Theta(n \log n) \)
    - ”Fast Fourier Transform”
    - FFT widely used for signal processing
- Integers
  - Similar, but some ugly details re: carries, etc. due to
    Schonhage-Strassen in 1971 gives \( \Theta(n \log n \log \log n) \)
  - Improvement in 2007 due to Furer gives \( \Theta(n \log n 2^{\log^* n}) \)
  - Used in practice in symbolic manipulation systems like Maple

Hints towards FFT: Interpolation

```
Given set of values at 5 points```

37 38 39 40
Hints towards FFT: Interpolation

Given set of values at 5 points
Can find unique degree 4 polynomial going through these points

Multiplying Polynomials by Evaluation & Interpolation

- Any degree $n-1$ polynomial $R(y)$ is determined by $R(y_0), \ldots, R(y_{n-1})$ for any $n$ distinct $y_0, \ldots, y_{n-1}$
- To compute $PQ$ (assume degree at most $n-1$)
  - Evaluate $P(y_0), \ldots, P(y_{n-1})$
  - Evaluate $Q(y_0), \ldots, Q(y_{n-1})$
  - Multiply values $P(y_i)Q(y_i)$ for $i=0,\ldots,n-1$
  - Interpolate to recover $PQ$

Interpolation

- Given values of degree $n-1$ polynomial $R$ at $n$ distinct points $y_0, \ldots, y_{n-1}$
  - $R(y_0), \ldots, R(y_{n-1})$
- Compute coefficients $c_0, \ldots, c_{n-1}$ such that
  - $R(x)=c_0+c_1x+c_2x^2+\ldots+c_{n-1}x^{n-1}$
- System of linear equations in $c_0, \ldots, c_{n-1}$
  - $c_0+y_0^2+\ldots+y_0^{n-1}=R(y_0)$ known
  - $c_0+y_1^2+\ldots+y_1^{n-1}=R(y_1)$
  - $\ldots$ unknown
  - $c_0+y_{n-1}^2+\ldots+y_{n-1}^{n-1}=R(y_{n-1})$

Interpolation: $n$ equations in $n$ unknowns

- Matrix form of the linear system
  $$\begin{bmatrix}1 & y_0 & y_0^2 & \ldots & y_0^{n-1} \\ 1 & y_1 & y_1^2 & \ldots & y_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & y_{n-1} & y_{n-1}^2 & \ldots & y_{n-1}^{n-1}\end{bmatrix}\begin{bmatrix}c_0 \\ c_1 \\ \vdots \\ c_{n-1}\end{bmatrix}=\begin{bmatrix}R(y_0) \\ R(y_1) \\ \vdots \\ R(y_{n-1})\end{bmatrix}$$
- Fact: Determinant of the matrix is $\prod_{i<j} (y_i-y_j)$ which is not 0 since points are distinct
  - System has a unique solution $c_0, \ldots, c_{n-1}$
**Hints towards FFT: Evaluation & Interpolation**

- Evaluation of polynomial at 1 point takes $O(n)$ time
  - So 2n points (naively) takes $O(n^2)$—no savings
  - But the algorithm works no matter what the points are...
- So...choose points that are related to each other so that evaluation problems can share subproblems

**Karatsuba’s algorithm and evaluation and interpolation**

- Strassen gave a way of doing 2x2 matrix multiplies
- Karatsuba’s algorithm can be thought of as a way of multiplying degree 1 polynomials (which have 2 coefficients) using fewer multiplications
  - $PQ = (P_0 + P_1 z)(Q_0 + Q_1 z)
    = P_0 Q_0 + (P_0 + P_1)(Q_0 + Q_1)z + P_1 Q_1 z^2$
- Evaluate at 0,1,-1 (Could also use other points)
  - $A = P(0)Q(0) = P_0 Q_0$
  - $C = P(1)Q(1) = P_0 + P_1(Q_0 + Q_1)$
  - $D = P(-1)Q(-1) = (P_0 - P_1)(Q_0 - Q_1)$
- Interpolating, Karatsuba’s $Mid = (C-D)/2$ and $B = (C+D)/2 - A$

**Evaluation at Special Points**

- Evaluation of polynomial at 1 point
- So 2n points (naively) takes $O(n^2)$—no savings
- But the algorithm works no matter what the points are...
- So...choose points that are related to each other so that evaluation problems can share subproblems

**The key idea: Evaluate at related points**

- $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 +...+ a_{n-1} x^{n-1}$
  - $= a_0 + a_2 x^2 + a_4 x^4 +...+ a_{n-2} x^{n-2}$
  - $+ a_1 x + a_3 x^3 + a_5 x^5 +...+ a_{n-1} x^{n-1}$
  - $= P_{even}(x^2) + x P_{odd}(x^2)$

- $P(-x) = a_0 - a_1 x + a_2 x^2 - a_3 x^3 + a_4 x^4 -...- a_{n-1} x^{n-1}$
  - $= a_0 + a_2 x^2 + a_4 x^4 +...+ a_{n-2} x^{n-2}$
  - $- (a_1 x + a_3 x^3 + a_5 x^5 +...+ a_{n-1} x^{n-1})$
  - $= P_{even}(x^2) - x P_{odd}(x^2)$

where $P_{even}(x) = a_0 + a_2 x + a_4 x^2 +...+ a_{n-2} x^{n-2-1}$

and $P_{odd}(x) = a_1 + a_3 x + a_5 x^2 +...+ a_{n-1} x^{n-2-1}$
The key idea:
Evaluate at related points

- So... if we have half the points as negatives of the other half
  - i.e., \( y_{n/2} = -y_0, y_{n/2+1} = -y_1, \ldots, y_{n-1} = -y_{n/2-1} \)
then we can reduce the size \( n \) problem of evaluating degree \( n-1 \) polynomial \( P \) at \( n \) points to evaluating 2 degree \( n/2 - 1 \) polynomials \( P_{\text{even}} \) and \( P_{\text{odd}} \) at \( n/2 \) points \( y_0^2, y_{n/2}^2 \) and recombine answers with \( O(1) \) extra work per point

- But to use this idea recursively we need half of \( y_0^2, y_{n/4}, \ldots, y_{n/2-1}^2 \) to be negatives of the other half
  - If \( y_{n/4}^2 = -y_0^2 \), say, then \( (y_{n/4}/y_0)^2 = -1 \)
- Motivates use of complex numbers as evaluation points

\[ e^{2\pi i} = 1 \]
\[ e^{\pi i} = -1 \]

**Complex Numbers**

\[ i^2 = -1 \]

To multiply complex numbers:
1. add angles
2. multiply lengths
   (all length 1 here)
\[ a+bi = (a+bi)(c+di) \]
\[ e^{\theta i} = \cos \theta + i \sin \theta = e^{\theta \pi i} \]
\[ c+di = \cos \phi + i \sin \phi = e^{\phi \pi i} \]
\[ e^{\theta i} = \cos (\theta + \phi) + i \sin (\theta + \phi) = e^{(\theta + \phi) \pi i} \]

**Primitive n\(^{th}\) root of 1** \( \omega = \omega_n = e^{i2\pi/n} \)

- Let \( \omega = \omega_n = e^{i2\pi/n} = \cos \left(\frac{2\pi n}{n}\right) + i \sin \left(\frac{2\pi n}{n}\right) \)
- \( \omega^2 = i \)
- \( \omega^3 = \omega \)
- \( \omega^4 = -1 \)
- \( \omega^5 = \omega^0 = 1 = \omega^8 \)
- \( \omega^6 = -i \)
- \( \omega^7 \)

\( \rho^2 = -1 \)
\( e^{2\pi i} = 1 \)

**Facts about \( \omega = e^{2\pi i/n} \) for even n**

- \( \omega = e^{2\pi i/n}, i = \sqrt{-1} \)
- \( \omega^n = 1 \)
- \( \omega^{n/2} = -1 \)
- \( \omega^{n/2+k} = -\omega^k \) for all values of \( k \)
- \( \omega^2 = \omega^{2\pi i/m} \) where \( m = n/2 \)
- \( \omega^k = \cos(2k\pi/n) + i \sin(2k\pi/n) \) so can compute with powers of \( \omega \)
- \( \omega^k \) is a root of \( x^{n-1} = (x-1)(x^{n-1}+x^{n-2}+\ldots+1) = 0 \)
  but for \( k \neq 0 \), \( \omega^k \neq 1 \) so \( \omega^k(n-1)+\omega^k(n-2)+\ldots+1 = 0 \)
The key idea for $n$ even

- $P(\omega) = a_0 + a_1 \omega + a_2 \omega^2 + a_3 \omega^3 + a_4 \omega^4 + \ldots + a_{n-1} \omega^{n-1}$
  
  $= a_0 + a_2 \omega^2 + a_4 \omega^4 + \ldots + a_{n-2} \omega^{n-2}$
  
  $+ a_1 \omega + a_3 \omega^3 + a_5 \omega^5 + \ldots + a_{n-1} \omega^{n-1}$

$= P_{\text{even}}(\omega^2) + \omega P_{\text{odd}}(\omega^2)$

- $P(-\omega) = a_0 - a_1 \omega + a_2 \omega^2 - a_3 \omega^3 + a_4 \omega^4 - \ldots - a_{n-1} \omega^{n-1}$

  $= a_0 + a_2 \omega^2 + a_4 \omega^4 + \ldots + a_{n-2} \omega^{n-2}$

  $- (a_1 \omega + a_3 \omega^3 + a_5 \omega^5 + \ldots + a_{n-1} \omega^{n-1})$

$= P_{\text{even}}(\omega^2) - \omega P_{\text{odd}}(\omega^2)$

where $P_{\text{even}}(x) = a_0 + a_2 x + a_4 x^2 + \ldots + a_{n-2} x^{n/2-1}$

and $P_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \ldots + a_{n-1} x^{n/2-1}$

---

Analysis and more

- Run-time
  
  $T(n) = 2 \cdot T(n/2) + cn$ so $T(n) = O(n \log n)$

- So much for evaluation ... what about interpolation?

  - Given
    
    $r_0 = R(1), \ r_1 = R(\omega), \ r_2 = R(\omega^2), \ldots, \ r_{n-1} = R(\omega^{n-1})$

  - Compute
    
    $c_0, \ c_1, \ldots, c_{n-1}$ s.t. $R(x) = c_0 + c_1 x + \ldots + c_{n-1} x^{n-1}$

---

The recursive idea for $n$ a power of 2

- Goal:
  
  Evaluate $P$ at $1, \omega, \omega^2, \ldots, \omega^{n-1}$

- Now
  
  $P_{\text{even}}$ and $P_{\text{odd}}$ have degree $n/2-1$ where

  $P(\omega^k) = P_{\text{even}}(\omega^{2k}) + \omega^k P_{\text{odd}}(\omega^{2k})$

  $P(-\omega^k) = P_{\text{even}}(\omega^{2k}) - \omega^k P_{\text{odd}}(\omega^{2k})$

- Recursive Algorithm
  
  - Evaluate $P_{\text{even}}$ at $1, \omega^2, \omega^4, \ldots, \omega^{n-2}$
  
  - Evaluate $P_{\text{odd}}$ at $1, \omega^2, \omega^4, \ldots, \omega^{n-2}$
  
  - Combine to compute $P$ at $1, \omega, \omega^2, \ldots, \omega^{n/2-1}$
  
  - Combine to compute $P$ at $-1, -\omega, -\omega^2, \ldots, -\omega^{n/2-1}$

    (i.e. at $\omega^{n/2}, \omega^{n/2+1}, \omega^{n/2+2}, \ldots, \omega^{n-1}$)

---

Interpolation $\approx$ Evaluation: strange but true

- Weird fact:
  
  If we define a new polynomial

  $S(x) = r_0 + r_1 x + r_2 x^2 + \ldots + r_{n-1} x^{n-1}$

  where $r_0, r_1, \ldots, r_{n-1}$ are the evaluations of $R$ at $1, \omega, \ldots, \omega^{n-1}$

  Then $c_k = S(\omega^k)/n$ for $k = 0, \ldots, n-1$

- So...
  
  evaluate $S$ at $1, \omega^1, \omega^2, \ldots, \omega^{(n-1)}$ then divide each answer by $n$ to get the $c_0, \ldots, c_{n-1}$

  $\omega^1$ behaves just like $\omega$ did so the same $O(n \log n)$ evaluation algorithm applies!
Divide and Conquer Summary

- Powerful technique, when applicable
- Divide large problem into a few smaller problems of the same type
- Choosing sub-problems of roughly equal size is usually critical
- Examples:
  - Merge sort, quicksort (sort of), polynomial multiplication, FFT, Strassen's matrix multiplication algorithm, powering, binary search, root finding by bisection, …

Why this is called the discrete Fourier transform

- Real Fourier series
  - Given a real valued function $f$ defined on $[0,2\pi]$ the Fourier series for $f$ is given by
    \[ f(x) = a_0 + a_1 \cos(x) + a_2 \cos(2x) + ... + a_m \cos(mx) + ... \]
    where
    \[
    a_m = \frac{1}{2\pi} \int_0^{2\pi} f(x) \cos(mx) \, dx
    \]
  - is the component of $f$ of frequency $m$
  - In signal processing and data compression one ignores all but the components with large $a_m$ and there aren’t many since

- Complex Fourier series
  - Given a function $f$ defined on $[0,2\pi]$ the complex Fourier series for $f$ is given by
    \[ f(z) = b_0 + b_1 e^{iz} + b_2 e^{2iz} + ... + b_m e^{miz} + ... \]
    where
    \[
    b_m = \frac{1}{2\pi} \int_0^{2\pi} f(z) e^{-miz} \, dz
    \]
  - is the component of $f$ of frequency $m$
  - If we discretize this integral using values at $n$ equally spaced points between $0$ and $2\pi$ we get
    \[
    b_m = \frac{1}{n} \sum_{k=0}^{n-1} f_k e^{-2k\pi i/n} = \frac{1}{n} \sum_{k=0}^{n-1} f_k \omega^{-km} \text{ where } f_k = f(2k\pi/n)
    \]
    just like interpolation!
Today

- Divide and conquer examples
  - Simple, randomized median algorithm
    - Expected $O(n)$ time
  - Not so simple, deterministic median algorithm
    - Worst case $O(n)$ time
  - Expected time analysis for Randomized QuickSort
    - Expected $O(n \log n)$ time

Order problems: Find the $k^{th}$ largest

- Runtime models
  - Machine Instructions
  - Comparisons
- Maximum
  - $O(n)$ time
  - $n-1$ comparisons
- 2nd Largest
  - $O(n)$ time
  - $?$ comparisons

Median Problem

- $k^{th}$ largest for $k = n/2$
- Easily done in $O(n \log n)$ time with sorting
  - How can the problem be solved in $O(n)$ time?
- Select($k, n$) – find the $k$-th smallest from a list of length $n$

Divide and Conquer

- $T(n) = n + T(\alpha n)$ for $\alpha < 1$
- Linear time solution
- Select algorithm – in linear time, reduce the problem from selecting the $k$-th largest of $n$ to the $j$-th smallest of $\alpha n$, for $\alpha < 1$
Quick Select

QSelect(\(k, S\))
Choose element \(x\) from \(S\)
\(S_L = \{y \in S \mid y < x\}\)
\(S_E = \{y \in S \mid y = x\}\)
\(S_G = \{y \in S \mid y > x\}\)

if \(|S_L| \geq k\)
    return QSelect(\(k, S_L\))
else if \(|S_L| + |S_E| \geq k\)
    return \(x\)
else
    return QSelect(\(k - |S_L| - |S_E|, S_G\))

Implementing “Choose an element \(x\)”

- Ideally, we would choose an \(x\) in the middle, to reduce both sets in half and guarantee progress
- Method 1
  - Select an element at random
- Method 2
  - BFPRT Algorithm
  - Select an element by a complicated, but linear time method that guarantees a good split

Random Selection

Consider a call to QSelect(\(k, S\)), and let \(S'\) be the elements passed to the recursive call.

With probability at least \(\frac{1}{2}\), \(|S'| < \frac{3}{4}|S|\)

↑ bad \(x\) ↑ good \(x\) ↑ good \(x\) ↑ bad \(x\)

elements of \(S\) listed in sorted order

⇒ On average only 2 recursive calls before the size of \(S'\) is at most \(3n/4\)

Expected runtime is \(O(n)\)

- Given \(x\), one pass over \(S\) to determine \(S_L, S_E,\) and \(S_G\) and their sizes: \(cn\) time.
  - Expect \(2cn\) cost before size of \(S'\) drops to at most \(3|S|/4\)

- Let \(T(n)\) be the expected running time
  - \(T(n) \leq T(3n/4) + 2cn\)
    \(\leq 2cn + (\frac{3}{4}) 2cn + (\frac{3}{4})^2 2cn + \ldots\)
    \(\leq 2cn (1 + (\frac{3}{4}) + (\frac{3}{4})^2 + \ldots)\)
Making the algorithm deterministic

- In $O(n)$ time, find an element that guarantees that the larger set in the split has size at most $\frac{3}{4}n$

Blum-Floyd-Pratt-Rivest-Tarjan Algorithm

- Divide $S$ into $n/5$ sets of size 5
- Sort each of these sets of size 5
- Let $M$ be the set of all medians of the sets of size 5
- Let $x$ be the median of $M$
- $S_L = \{y \in S \mid y < x\}$, $S_G = \{y \in S \mid y > x\}$
- Claim: $|S_L| < \frac{3}{4}|S|$, $|S_G| < \frac{3}{4}|S|$

BFPRT, Step 1: Construct sets of size 5, sort each set

BFPRT, Step 2: Find median of column medians

<table>
<thead>
<tr>
<th>95</th>
<th>86</th>
<th>81</th>
<th>81</th>
<th>69</th>
<th>91</th>
<th>98</th>
<th>51</th>
<th>77</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>65</td>
<td>62</td>
<td>73</td>
<td>47</td>
<td>81</td>
<td>96</td>
<td>36</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>45</td>
<td>52</td>
<td>32</td>
<td>9</td>
<td>42</td>
<td>91</td>
<td>21</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>16</td>
<td>41</td>
<td>25</td>
<td>8</td>
<td>25</td>
<td>64</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>32</td>
<td>12</td>
<td>7</td>
<td>18</td>
<td>6</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>
BFPRT Recurrence

- Sorting all \( n/5 \) lists of size 5
  - \( cn \) time

- Finding median of set \( M \) of medians
  - Recursive computation: \( T(n/5) \)

- Computing sets \( S_L, S_E, S_G \) and \( S' \)
  - \( c'n \) time

- Solving selection problem on \( S' \)
  - Recursive computation: \( T(3n/4) \) since \( |S'| \leq \frac{3}{4} n \)

\[ T(n) \leq cn + T(n/5) + T(3n/4) \] is \( O(n) \)

- Key property
  - \( 3/4 + 1/5 < 1 \) (The sum is \( 19/20 \))

- Sum of problem sizes decreases by \( 19/20 \) factor per level of recursion

- Overhead per level is linear in the sum of the problem sizes
  - Overhead decreases by \( 19/20 \) factor per level of recursion
  - Total overhead is linear (sum of geometric series with constant ratio and linear largest term)

Quick Sort

QuickSort(\( S \))

  if \( S \) is empty, return
  Choose element \( x \) from \( S \) "pivot"
  \( S_L = \{ y \ in \ S \mid y < x \} \)
  \( S_E = \{ y \ in \ S \mid y = x \} \)
  \( S_G = \{ y \ in \ S \mid y > x \} \)
  return [QuickSort(\( S_L \)), \( S_E \), QuickSort(\( S_G \))]
**Expected run time for QuickSort:**

“Global analysis”

- Count comparisons
- \( a_i, a_j \) – elements in positions \( i \) and \( j \) in the **final** sorted list. \( p_{ij} \) the probability that \( a_i \) and \( a_j \) are compared
- Expected number of comparisons:

\[
\sum_{i<j} p_{ij}
\]

**Lemma:** \( P_{ij} \leq \frac{2}{(j-i+1)} \)

If \( a_i \) and \( a_j \) are compared then it must be during the call when they end up in different subproblems

- Before that, they aren’t compared to each other
- After they aren’t compared to each other

During this step they are only compared if one of them is the pivot

Since all elements between \( a_i \) and \( a_j \) are also in the subproblem this is 2 out of at least \( j-i+1 \) choices

**Average runtime is** \( 2n \ln n \)

\[
\sum_{i<j} p_{ij} \leq \sum_{i<j} \frac{2}{(j-i+1)} \quad \text{write } j=k+i
\]

\[
= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{(k+1)}
\]

\[
\leq 2(n-1)(H_n-1)
\]

where \( H_n=1+1/2+1/3+1/4+...+1/n \)

\[
= \ln n + O(1)
\]

\[
\leq 2n \ln n + O(n) \leq 1.387n \log_2 n
\]