Advanced Algorithms

Piyush Kumar

(Lecture 6: Div & Conquer)

Welcome to COP4531

Source: Kevin Wayne, Harold Prokop.

Divide and Conquer

- Inversions
- Closest Point problems
- Integer multiplication
- Matrix Multiplication
- Cache Aware algorithms/ Cache oblivious Algorithms
- Static Van Emde Boas Layout

5.3 Counting Inversions

Counting Inversions

- Music site tries to match your song preferences with others.
  - You rank n songs.
  - Music site consults database to find people with similar tastes.
- Similarity metric: number of inversions between two rankings.
  - My rank: 1, 2, …, n.
  - Your rank: $a_1, a_2, \ldots, a_n$.
  - Songs i and j inverted if $i < j$, but $a_i > a_j$.

Songs

	A	B	C	D	E
Me	1	2	3	4	5
You	1	3	4	2	5

Inversions

3-2, 4-2

- Brute force: check all $\Theta(n^2)$ pairs i and j.

- Applications.
  - Voting theory.
  - Collaborative filtering.
  - Measuring the "sortedness" of an array.
  - Sensitivity analysis of Google's ranking function.
  - Rank aggregation for meta-searching on the Web.
  - Nonparametric statistics (e.g., Kendall's Tau distance).

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.

1	5	4	8	10	2	6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2		6	9	12	11	3	7

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2		6	9	12	11	3	7

Conquer: 2T(n / 2)

5 blue-blue inversions
5-4, 5-2, 4-2, 8-2, 10-2

8 green-green inversions
6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Counting Inversions: Divide-and-Conquer

- Divide-and-conquer.
  - Divide: separate list into two pieces.
  - Conquer: recursively count inversions in each half.
  - Combine: count inversions where a_i and a_j are in different halves, and return sum of three quantities.

1	5	4	8	10	2	6	9	12	11	3	7

Divide: O(1).

1	5	4	8	10	2		6	9	12	11	3	7

Conquer: 2T(n / 2)

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Combine: ???

Total = 5 + 8 + 9 = 22.

Counting Inversions: Combine

- Combine: count blue-green inversions
  - Assume each half is sorted.
  - Count inversions where a_i and a_j are in different halves.
  - Merge two sorted halves into sorted whole.

to maintain sorted invariant

3	7	10	14	18	19		2	11	16	17	23	25
							6	3	2	2	0	0

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: O(n)

2	3	7	10	11	14	16	17	18	19	23	25

Merge: O(n)

$$T(n) \le T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \Rightarrow T(n) = O(n \log n)$$

Counting Inversions: Implementation

- Pre-condition. [Merge-and-Count] A and B are sorted.
- Post-condition. [Sort-and-Count] L is sorted.

```
Sort-and-Count(L) {
   if list L has one element
      return 0 and the list L

   Divide the list into two halves A and B
   (rA, A) ← Sort-and-Count(A)
   (rB, B) ← Sort-and-Count(B)
   (r, L) ← Merge-and-Count(A, B)

   return r = rA + rB + r and the sorted list L
}
```

5.4 Closest Pair of Points

Closest Pair of Points

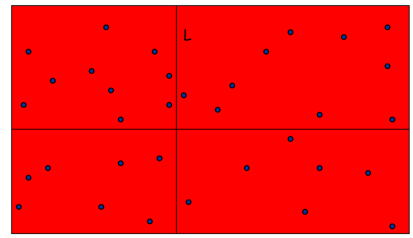
- Closest pair. Given n points in the plane, find a pair with smallest Euclidean distance between them.
- Fundamental geometric primitive.
  - Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
  - Special case of nearest neighbor, Euclidean MST, Voronoi.

    fast closest pair inspired fast algorithms for these problems
- Brute force. Check all pairs of points p and q with $\Theta(n^2)$ comparisons.
- 1-D version. O(n log n) easy if points are on a line.
- Assumption. No two points have same x coordinate.

  ↑ to make presentation cleaner

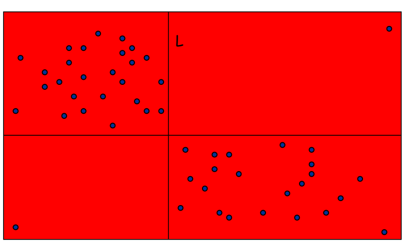
Closest Pair of Points: First Attempt

- Divide. Sub-divide region into 4 quadrants.



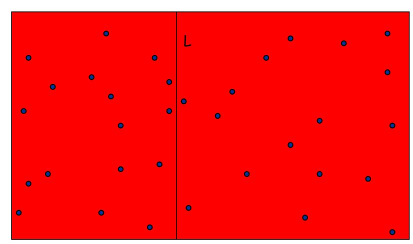
Closest Pair of Points: First Attempt

- Divide. Sub-divide region into 4 quadrants.
- Obstacle. Impossible to ensure n/4 points in each piece.



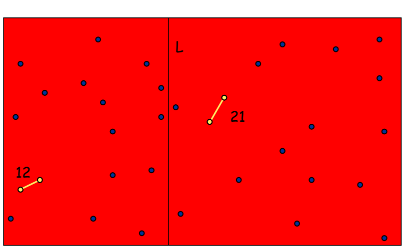
Closest Pair of Points

- Algorithm.
  - Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.



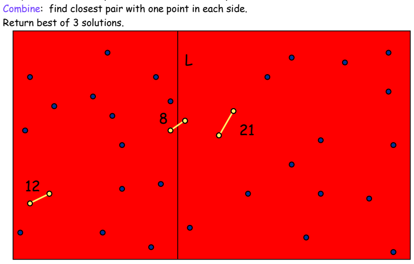
Closest Pair of Points

- Algorithm.
  - Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
  - Conquer: find closest pair in each side recursively.



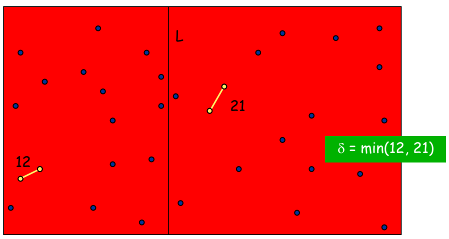
Closest Pair of Points

- Algorithm.
  - Divide: draw vertical line L so that roughly $\frac{1}{2}n$ points on each side.
  - Conquer: find closest pair in each side recursively.
  - Combine: find closest pair with one point in each side. ← seems like $\Theta(n^2)$
  - Return best of 3 solutions.



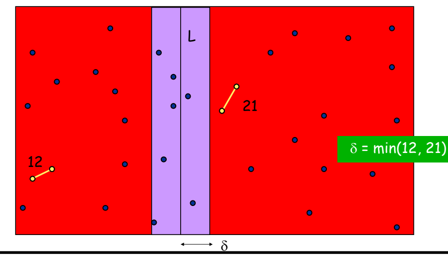
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.



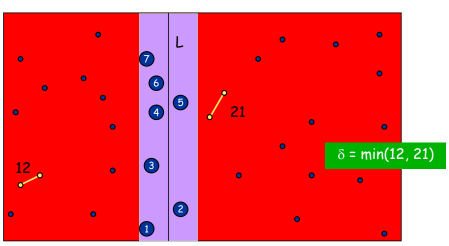
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.
  - Observation: only need to consider points within δ of line L.



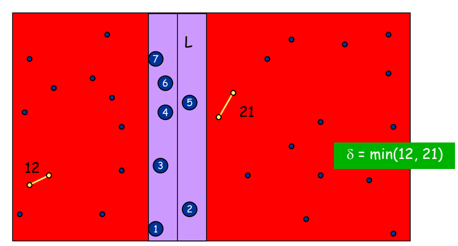
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.
  - Observation: only need to consider points within δ of line L.
  - Sort points in 2δ-strip by their y coordinate.



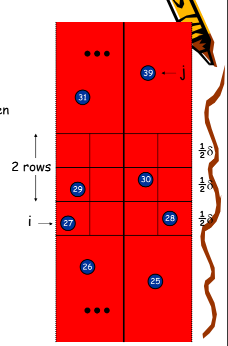
Closest Pair of Points

- Find closest pair with one point in each side, assuming that distance < δ.
  - Observation: only need to consider points within δ of line L.
  - Sort points in 2δ-strip by their y coordinate.
  - Only check distances of those within 11 positions in sorted list!



Closest Pair of Points

- Def. Let s_i be the point in the 2δ-strip, with the i^{th} smallest y-coordinate.
- Claim. If $|i - j| \geq 12$, then the distance between s_i and s_j is at least δ.
- Pf.
  - No two points lie in same $\frac{1}{2}\delta$-by-$\frac{1}{2}\delta$ box.
  - Two points at least 2 rows apart have distance $\geq 2(\frac{1}{2}\delta)$. ▪
- Fact. Still true if we replace 12 with 7.



Closest Pair Algorithm

```
Closest-Pair(p1, …, pn) {
   Compute separation line L such that half the points        O(n log n)
   are on one side and half on the other side.

   δ1 = Closest-Pair(left half)                                2T(n / 2)
   δ2 = Closest-Pair(right half)
   δ  = min(δ1, δ2)

   Delete all points further than δ from separation line L    O(n)

   Sort remaining points by y-coordinate.                      O(n log n)

   Scan points in y-order and compare distance between        O(n)
   each point and next 11 neighbors. If any of these
   distances is less than δ, update δ.

   return δ.
}
```

Closest Pair of Points: Analysis

- Running time.

$$T(n) \le 2T(n/2) + O(n \log n) \;\Rightarrow\; T(n) = O(n \log^2 n)$$

- Q. Can we achieve O(n log n)?

- A. Yes. Don't sort points in strip from scratch each time.
  - Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
  - Sort by merging two pre-sorted lists.

$$T(n) \le 2T(n/2) + O(n) \;\Rightarrow\; T(n) = O(n \log n)$$

5.5 Integer Multiplication

26

Integer Arithmetic

- Add. Given two n-digit integers a and b, compute a + b.
  - O(n) bit operations.

- Multiply. Given two n-digit integers a and b, compute a × b.
  - Brute force solution: $\Theta(n^2)$ bit operations.

```
 1 1 1 1 1 1 0 1
   1 1 0 1 0 1 0 1
 + 0 1 1 1 1 1 0 1
 1 0 1 0 1 0 0 1 0
```

Add

```
                         1 1 0 1 0 1 0 1
                       * 0 1 1 1 1 1 0 1
                       1 1 0 1 0 1 0 1 0
                     0 0 0 0 0 0 0 0 0
                   1 1 0 1 0 1 0 1 0
                 1 1 0 1 0 1 0 1 0
               1 1 0 1 0 1 0 1 0
             1 1 0 1 0 1 0 1 0
           1 1 0 1 0 1 0 1 0
         0 0 0 0 0 0 0 0 0
       0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 1 0
```

Multiply

Divide-and-Conquer Multiplication: Warmup

- To multiply two n-digit integers:
  - Multiply four ½n-digit integers.
  - Add two ½n-digit integers, and shift to obtain result.

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= \left(2^{n/2} \cdot x_1 + x_0\right)\left(2^{n/2} \cdot y_1 + y_0\right) = 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \end{aligned}$$

$$T(n) = \underbrace{4T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, shift}} \;\Rightarrow\; T(n) = \Theta(n^2)$$

↑ assumes n is a power of 2

Karatsuba Multiplication

- To multiply two n-digit integers:
  - Add two ½n digit integers.
  - Multiply three ½n-digit integers.
  - Add, subtract, and shift ½n-digit integers to obtain result.

$$\begin{aligned} x &= 2^{n/2} \cdot x_1 + x_0 \\ y &= 2^{n/2} \cdot y_1 + y_0 \\ xy &= 2^n \cdot x_1 y_1 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + x_0 y_0 \\ &= 2^n \cdot \underset{A}{x_1 y_1} + 2^{n/2} \cdot \big(\underset{B}{(x_1 + x_0)(y_1 + y_0)} - \underset{A}{x_1 y_1} - \underset{C}{x_0 y_0}\big) + \underset{C}{x_0 y_0} \end{aligned}$$

- Theorem. [Karatsuba-Ofman, 1962] Can multiply two n-digit integers in $O(n^{1.585})$ bit operations.

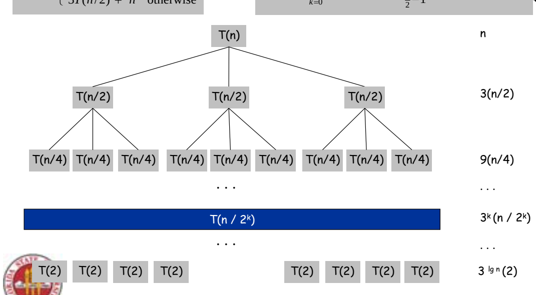
$$T(n) \le \underbrace{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(1 + \lceil n/2 \rceil)}_{\text{recursive calls}} + \underbrace{\Theta(n)}_{\text{add, subtract, shift}}$$

$$\Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

Karatsuba: Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 3T(n/2) + n & \text{otherwise} \end{cases}$$

$$T(n) = \sum_{k=0}^{\log_2 n} n \left(\tfrac{3}{2}\right)^k = \frac{\left(\tfrac{3}{2}\right)^{1+\log_2 n} - 1}{\tfrac{3}{2} - 1} = 3n^{\log_2 3} - 2$$



Matrix Multiplication

Courtesy Harald Prokop

Matrix Multiplication (MM)

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{matrix} c_{11} & c_{12} & \Lambda & c_{1n} \\ c_{21} & c_{22} & \Lambda & c_{2n} \\ M & M & O & M \\ c_{n1} & c_{n2} & \Lambda & c_{nn} \end{matrix} = \begin{matrix} a_{11} & a_{12} & \Lambda & a_{1n} \\ a_{21} & a_{22} & \Lambda & a_{2n} \\ M & M & O & M \\ a_{n1} & a_{n2} & \Lambda & a_{nn} \end{matrix} \times \begin{matrix} b_{11} & b_{12} & \Lambda & b_{1n} \\ b_{21} & b_{22} & \Lambda & b_{2n} \\ M & M & O & M \\ b_{n1} & b_{n2} & \Lambda & b_{nn} \end{matrix}$$

C A B

Matrix Multiplication

- Matrix multiplication. Given two n-by-n matrices A and B, compute C = AB.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

- Brute force. $\Theta(n^3)$ arithmetic operations.
- Fundamental question. Can we improve upon brute force?

Recursive Matrix Multiplication

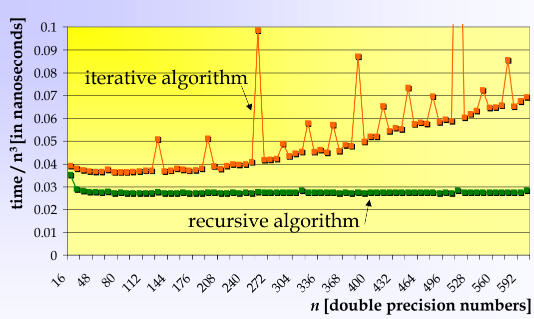
Divide and conquer on $n \times n$ matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

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

8 multiplications of $(n/2) \times (n/2)$ matrices.
1 addition of $n \times n$ matrices.

$\Theta(n^3)$-*Matrix Multiplication* $(n,n) \times (n,n)$



450-MHz AMD K6-III processor with 32kB L1-cache, 64kB L2-cache, and 1MB L3-cache.

Matrix Multiplication: Warmup

- Divide-and-conquer.
  - Divide: partition A and B into ½n-by-½n blocks.
  - Conquer: multiply 8 ½n-by-½n recursively.
  - Combine: add appropriate products using 4 matrix additions.

$$T(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow T(n) = \Theta(n^3)$$

Matrix Multiplication: Key Idea

- Key idea. multiply 2-by-2 block matrices with only 7 multiplications.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$

- 7 multiplications.
- 18 = 10 + 8 additions (or subtractions).

Fast Matrix Multiplication

- Fast matrix multiplication. (Strassen, 1969)
  - Divide: partition A and B into ½n-by-½n blocks.
  - Compute: 14 ½n-by-½n matrices via 10 matrix additions.
  - Conquer: multiply 7 ½n-by-½n matrices recursively.
  - Combine: 7 products into 4 terms using 8 matrix additions.
- Analysis.
  - Assume n is a power of 2.
  - T(n) = # arithmetic operations.

$$T(n) = \underbrace{7T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, subtract}} \Rightarrow T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})$$

Fast Matrix Multiplication in Practice

- Implementation issues.
  - Sparsity.
  - Caching effects.
  - Numerical stability.
  - Odd matrix dimensions.
  - Crossover to classical algorithm around n = 128.
- Common misperception: "Strassen is only a theoretical curiosity."
  - Advanced Computation Group at Apple Computer reports 8x speedup on G4 Velocity Engine when n ~ 2,500.
  - Range of instances where it's useful is a subject of controversy.
- Remark. Can "Strassenize" Ax=b, determinant, eigenvalues, and other matrix ops.

Fast Matrix Multiplication in Theory

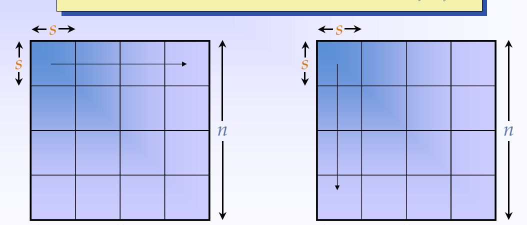
- Q. Multiply two 2-by-2 matrices with only 7 scalar multiplications?
- A. Yes! [Strassen, 1969] $\Theta(n^{\log_2 7}) = O(n^{2.81})$
- Q. Multiply two 2-by-2 matrices with only 6 scalar multiplications?
- A. Impossible. [Hopcroft and Kerr, 1971] $\Theta(n^{\log_2 6}) = O(n^{2.59})$
- Q. Two 3-by-3 matrices with only 21 scalar multiplications?
- A. Also impossible. $\Theta(n^{\log_3 21}) = O(n^{2.77})$
- Q. Two 70-by-70 matrices with only 143,640 scalar multiplications?
- A. Yes! [Pan, 1980] $\Theta(n^{\log_{70} 143640}) = O(n^{2.80})$
- Decimal wars.
  - December, 1979: $O(n^{2.521813})$.
  - January, 1980: $O(n^{2.521801})$.

Fast Matrix Multiplication in Theory

- Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987.]
- Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.
- Caveat. Theoretical improvements to Strassen are progressively less practical.

Cache-Aware MM

```
BLOCK-MULT (A,B,C,n)
  for i ← 1 to n/s
     do for j ← 1 to n/s
           do for k ← 1 to n/s
                 do ORD-MULT (A_ik, B_kj, C_ij, s)
```



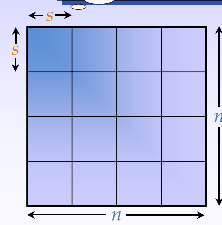
Towards faster matrix multiplication…
(Blocked version)

- Advantages
  - Exploit locality using blocking
  - Do not assume that each access to memory is O(1)
  - Can be extended to multiple levels of cache
  - Usually the fastest algorithm after tuning.
- Disadvantages
  - Needs tuning every time it runs on a new machine.
  - Usually "s" is a voodoo parameter that is unknown.

Cache-Aware Matrix Multiplication

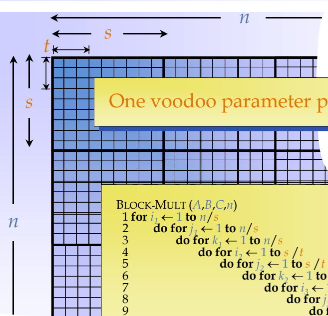
```
BLOCK-MULT (A,B,C,n)
f…
…ULT (A_ik,B_kj,C_ij,s)
```

Voodoo!



- Tune s so that A_{ik}, B_{kj}, and C_{ij} just fit into cache? $s = \Theta(\sqrt{Z})$
- If $n > s$, then
  $$Q(n) = \Theta\left((n/s)^3 (s^2/L)\right) = \Theta\left(n^3 / L\sqrt{Z}\right).$$
- Optimal [HK81].

Three-Level Cache



One voodoo parameter per caching level!

```
BLOCK-MULT (A,B,C,n)
for i1 ← 1 to n/s
  do for j1 ← 1 to n/s
    do for k1 ← 1 to n/s
      do for i2 ← 1 to s/t
        do for j2 ← 1 to s/t
          do for k2 ← 1 to s/t
            do for i3 ← 1 to t/u
              do for j3 ← 1 to t/u
                do for k3 ← 1 to t/u
                  do ORD-MULT (A_ik,B_kj,C_ij,u)
```

Recursive Matrix Multiplication

Cache Oblivious

Divide and conquer on $n \times n$ matrices.

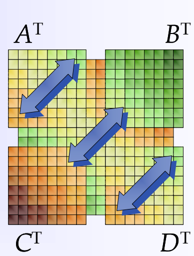
$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

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

8 multiplications of $(n/2) \times (n/2)$ matrices.
1 addition of $n \times n$ matrices.

Recursive Transpose

1. Partition matrix in 4 submatrices A, B, C, and D
2. Recursively transpose A.
3. Recursively transpose and swap B and C.
4. Recursively transpose D.

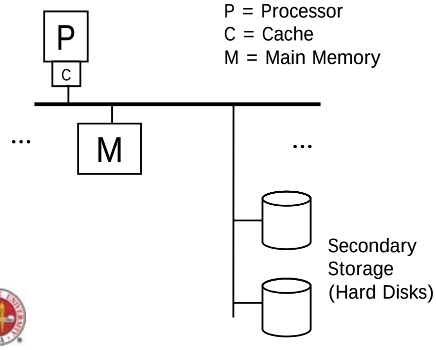


$\Theta(n^2 / L)$ cache misses, which is optimal. Used as a subroutine in our optimal cache-oblivious FFT [HK81].

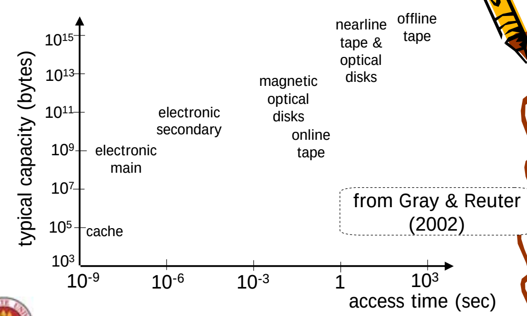
Algorithms for External Memory

Computer

P = Processor
C = Cache
M = Main Memory

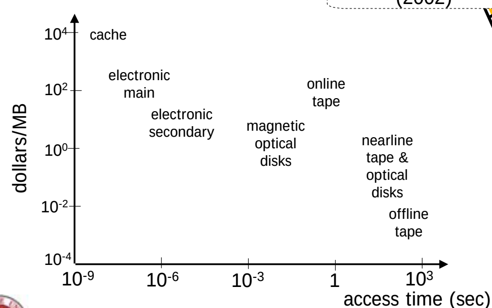


Storage Capacity



Storage Cost

from Gray & Reuter (2002)



Disk Access Time

- Block X Request → Block in Memory
- Time = Seek Time +
  Rotational Delay +
  Transfer Time +
  Other

Typical Numbers (for random block access)
Seek Time = 10ms
Rot Delay = 4ms (7200rpm)
transfer rate = 50MB/sec
Other = CPU time to access delays+
Contention for controllers
Contention for Bus
Multiple copies of the same data

Rules for EM Algorithms

- Sequential IO is cheap compared to Random IO
- 1kb block
  - Seq : 1ms
  - Random : 20ms
- The difference becomes smaller and smaller as the block size becomes larger.

The DAM Model

- Count the number of IOs.
- Explicitly control which Blocks are in memory.
- Two level memory model.
- Notation:
  - M = Size of memory.
  - B = size of disk block.
  - N = size of data.

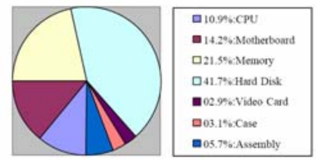
Question: How many block transfers for one scan of the data set?

Problem

- Mergesort
  - How many Block IOs does it need to sort N numbers (when the size of the data is extremely large compared to M)
- Can we do better?

External Memory Sorting

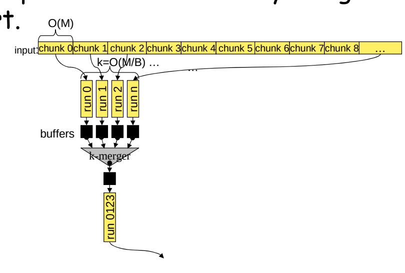
- Penny Sort Competition



40GB , 433 million records,
1541 seconds on a 614$ Linux/AMD system

EM Sorting

- Two pass external memory merge sort.



The CO Memory Model

- Cache-oblivious memory model
  - Reason about two-level, but prove results for unknown multilevel memory models
  - Parameters B and M are unknown, thus optimized for all levels of memory hierarchy
- B = L , M = Z?

Matrix Transpose: DAM n CO

- What is the number of blocks you need to move in a transpose of a large matrix?
  - In DAM
  - In CO

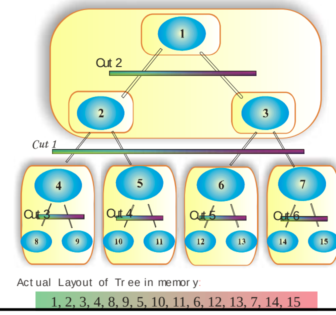
Static Searches

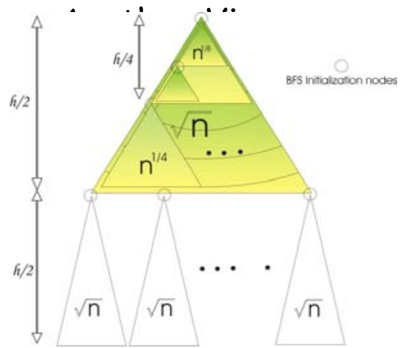
- Only for balanced binary trees
- Assume there are no insertions and deletions
- Only searches
- Can we speed up such seaches?

What is a layout?

- Mapping of nodes of a tree to the Memory
- Different kinds of layouts
  - In-order
  - Post-order
  - Pre-order
  - Van Emde Boas
- *Main Idea* : Store Recursive subtrees in contiguous memory

Example of Van Emde Boas



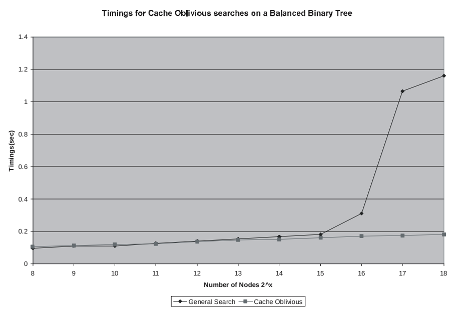


Theoretical Guarantees?

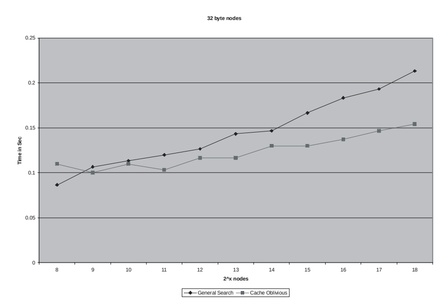
- Cache Complexity $Q(n) = O(\log_L n)$
- Work Complexity $W(n) = O(\log n)$

From Prokop's Thesis

In Practice??



In Practice II



In Practice!

- Matrix Operations by Morton Ordering , David S.Wise
  (Cache oblivious Practical Matrix operation results)
- Bender, Duan, Wu
  (Cache oblivious dictionaries)
- Rahman, Cole, Raman (CO B-Trees)

Known Optimal Results

- Matrix Multiplication
- Matrix Transpose
- n-point FFT
- LUP Decomposition
- Sorting
- Searching

Other Results Known

Priority Q	$O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$
List Ranking	$O(sort(V))$
Tree Algos	$O(sort(V))$
Directed BFS/DFS	$O((V+\frac{E}{B})\log_2 V + sort(E))$
Undirected BFS	$O(V + sort(E))$
MSF	$O(sort(E)+\log_2\log_2 V)$

Introduction to Streaming Algorithms

Brain Teaser

- Let P = { 1…n }. Let P' = P \ {x}
  - x in P
- Paul shows Carole elements from P'
- Carole can only use O(log n) bits memory to answer the question in the end.

Now what about P''?

Streaming Algorithms

- Data that computers are being asked to process is growing astronomically.
- Infeasible to store
- If I cant store the data I am looking at, how do I compute a summary of this data?

Another Brain Teaser

- Given a set of numbers in a large array.
- In one pass, decide if some item is in majority (assume you only have constant size memory to work with).

2		7	6	4		3		

N = 12; item 9 is majority

Courtesy : Subhash Suri

Misra-Gries Algorithm ('82)

- A counter and an ID.
  - If new item is same as stored ID, increment counter.
  - Otherwise, decrement the counter.
  - If counter 0, store new item with count = 1.
- If counter > 0, then its item is the only candidate for majority.

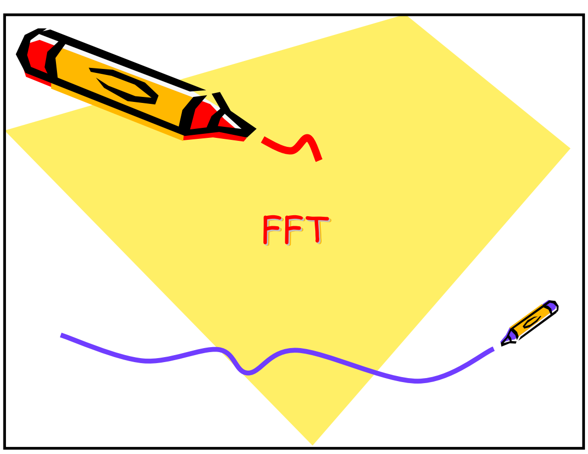
	2	9	9	9	7	6	4	9	9	9	3	9
ID		2	9	9	9	9		4	9	9	9	
count		0	1	2	1	0		0	1	2	1	

Data Stream Algorithms

- Majority and Frequent are examples of data stream algorithms.
- Data arrives as an online sequence $x_1, x_2, \ldots,$ potentially infinite.
- Algorithm processes data in one pass (in given order)
- Algorithm's memory is significantly smaller than input data
- Summarize the data: compute useful patterns

Streaming Data Sources

- Internet traffic monitoring
- New Computer Graphics hardware
- Web logs and click streams
- Financial and stock market data
- Retail and credit card transactions
- Telecom calling records
- Sensor networks, surveillance
- RFID
- Instruction profiling in microprocessors
- Data warehouses (random access too expensive).



Fast Fourier Transform: Applications

- Applications.
  - Optics, acoustics, quantum physics, telecommunications, control systems, signal processing, speech recognition, data compression, image processing.
  - DVD, JPEG, MP3, MRI, CAT scan.
  - Numerical solutions to Poisson's equation.

> The FFT is one of the truly great computational developments of this [20th] century. It has changed the face of science and engineering so much that it is not an exaggeration to say that life as we know it would be very different without the FFT. *-Charles van Loan*

Fast Fourier Transform: Brief History

- Gauss (1805, 1866). Analyzed periodic motion of asteroid Ceres.
- Runge-König (1924). Laid theoretical groundwork.
- Danielson-Lanczos (1942). Efficient algorithm.
- Cooley-Tukey (1965). Monitoring nuclear tests in Soviet Union and tracking submarines. Rediscovered and popularized FFT.
- Importance not fully realized until advent of digital computers.

Polynomials: Coefficient Representation

- Polynomial. [coefficient representation]

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

$$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

- Add: O(n) arithmetic operations.

$$A(x) + B(x) = (a_0 + b_0) + (a_1 + b_1)x + \cdots + (a_{n-1} + b_{n-1})x^{n-1}$$

- Evaluate: O(n) using Horner's method.

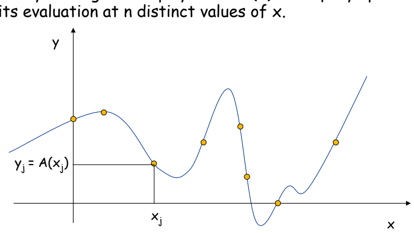
$$A(x) = a_0 + (x(a_1 + x(a_2 + \cdots + x(a_{n-2} + x(a_{n-1}))\cdots))$$

- Multiply (convolve): O(n^2) using brute force.

$$A(x) \times B(x) = \sum_{i=0}^{2n-2} c_i x^i, \text{ where } c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

Polynomials: Point-Value Representation

- Fundamental theorem of algebra. [Gauss, PhD thesis] A degree n polynomial with complex coefficients has n complex roots.
- Corollary. A degree n-1 polynomial A(x) is uniquely specified by its evaluation at n distinct values of x.



Polynomials: Point-Value Representation

- Polynomial. [point-value representation]

$$A(x): (x_0, y_0), \cdots, (x_{n-1}, y_{n-1})$$

$$B(x): (x_0, z_0), \cdots, (x_{n-1}, z_{n-1})$$

- Add: O(n) arithmetic operations.

$$A(x) + B(x): (x_0, y_0 + z_0), \ldots, (x_{n-1}, y_{n-1} + z_{n-1})$$

- Multiply: O(n), but need 2n-1 points.

$$A(x) \times B(x): (x_0, y_0 \times z_0), \cdots, (x_{2n-1}, y_{2n-1} \times z_{2n-1})$$

- Evaluate: O(n^2) using Lagrange's formula.

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Converting Between Two Polynomial Representations

•Tradeoff. Fast evaluation or fast multiplication. We want both!

Representation	Multiply	Evaluate
Coefficient	O(n^2)	O(n)
Point-value	O(n)	O(n^2)

•Goal. Make all ops fast by efficiently converting between two representations.

$a_0, a_1, \ldots, a_{n-1}$ ⇄ $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$

coefficient representation — point-value representation

Converting Between Two Polynomial Representations: Brute Force

•Coefficient to point-value. Given a polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, \ldots, x_{n-1}$.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

O(n^2) for matrix-vector multiply

O(n^3) for Gaussian elimination

Vandermonde matrix is invertible iff x_i distinct

•Point-value to coefficient. Given n distinct points $x_0, \ldots, x_{n-1}$ and values $y_0, \ldots, y_{n-1}$, find unique polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ that has given values at given points.

Coefficient to Point-Value Representation: Intuition

- Coefficient to point-value. Given a polynomial $a_0 + a_1 x + ... + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, ... , x_{n-1}$.
- Divide. Break polynomial up into even and odd powers.
  - $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$.
  - $A_{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$.
  - $A_{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$.
  - $A(x) = A_{even}(x^2) + x\,A_{odd}(x^2)$.
  - $A(-x) = A_{even}(x^2) - x\,A_{odd}(x^2)$.
- Intuition. Choose two points to be ± 1.
  - $A(1) = A_{even}(1) + 1\,A_{odd}(1)$.
  - $A(-1) = A_{even}(1) - 1\,A_{odd}(1)$.

Can evaluate polynomial of degree $\le n$ at 2 points by evaluating two polynomials of degree $\le \frac{1}{2}n$ at 1 point.

Coefficient to Point-Value Representation: Intuition

- Coefficient to point-value. Given a polynomial $a_0 + a_1 x + ... + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, ... , x_{n-1}$.
- Divide. Break polynomial up into even and odd powers.
  - $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$.
  - $A_{even}(x) = a_0 + a_2x + a_4x^2 + a_6x^3$.
  - $A_{odd}(x) = a_1 + a_3x + a_5x^2 + a_7x^3$.
  - $A(x) = A_{even}(x^2) + x\,A_{odd}(x^2)$.
  - $A(-x) = A_{even}(x^2) - x\,A_{odd}(x^2)$.
- Intuition. Choose four points to be $\pm 1, \pm i$.
  - $A(1) = A_{even}(1) + 1\,A_{odd}(1)$.
  - $A(-1) = A_{even}(1) - 1\,A_{odd}(1)$.
  - $A(i) = A_{even}(-1) + i\,A_{odd}(-1)$.
  - $A(-i) = A_{even}(-1) - i\,A_{odd}(-1)$.

Can evaluate polynomial of degree $\le n$ at 4 points by evaluating two polynomials of degree $\le \frac{1}{2}n$ at 2 points.

Discrete Fourier Transform

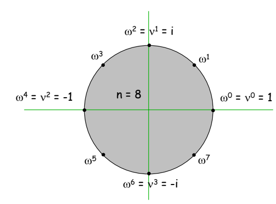
- Coefficient to point-value. Given a polynomial $a_0 + a_1 x + ... + a_{n-1} x^{n-1}$, evaluate it at n distinct points $x_0, ... , x_{n-1}$.
- Key idea: choose $x_k = \omega^k$ where ω is principal n^{th} root of unity.

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

↑ Discrete Fourier transform ↑ Fourier matrix F_n

Roots of Unity

- Def. An n^{th} root of unity is a complex number x such that $x^n = 1$.
- Fact. The n^{th} roots of unity are: $\omega^0, \omega^1, ..., \omega^{n-1}$ where $\omega = e^{2\pi i / n}$.
- Pf. $(\omega^k)^n = (e^{2\pi i k / n})^n = (e^{\pi i})^{2k} = (-1)^{2k} = 1$.
- Fact. The $\frac{1}{2}n^{th}$ roots of unity are: $\nu^0, \nu^1, ..., \nu^{n/2-1}$ where $\nu = e^{4\pi i / n}$.
- Fact. $\omega^2 = \nu$ and $(\omega^2)^k = \nu^k$.



Fast Fourier Transform

- Goal. Evaluate a degree n-1 polynomial $A(x) = a_0 + ... + a_{n-1} x^{n-1}$ at its n^{th} roots of unity: $\omega^0, \omega^1, ..., \omega^{n-1}$.
- Divide. Break polynomial up into even and odd powers.
  - $A_{even}(x) = a_0 + a_2x + a_4x^2 + ... + a_{n/2-2}\,x^{(n-1)/2}$.
  - $A_{odd}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n/2-1}\,x^{(n-1)/2}$.
  - $A(x) = A_{even}(x^2) + x\,A_{odd}(x^2)$.
- Conquer. Evaluate degree $A_{even}(x)$ and $A_{odd}(x)$ at the $\frac{1}{2}n^{th}$ roots of unity: $\nu^0, \nu^1, ..., \nu^{n/2-1}$.
- Combine.
  - $A(\omega^k) = A_{even}(\nu^k) + \omega^k A_{odd}(\nu^k), \quad 0 \le k < n/2$
  - $A(\omega^{k+n/2}) = A_{even}(\nu^k) - \omega^k A_{odd}(\nu^k), \quad 0 \le k < n/2$

↑ $\omega^{k+n/2} = -\omega^k$

$\nu^k = (\omega^k)^2 = (\omega^{k+n/2})^2$

FFT Algorithm

```
fft(n, a₀,a₁,…,aₙ₋₁) {
   if (n == 1) return a₀

   (e₀,e₁,…,eₙ/₂₋₁) ← FFT(n/2, a₀,a₂,a₄,…,aₙ₋₂)
   (d₀,d₁,…,dₙ/₂₋₁) ← FFT(n/2, a₁,a₃,a₅,…,aₙ₋₁)

   for k = 0 to n/2 - 1 {
      ωᵏ ← e^(2πik/n)
      y_(k+n/2) ← e_k + ωᵏ d_k
      y_(k+n/2) ← e_k - ωᵏ d_k
   }

   return (y₀,y₁,…,yₙ₋₁)
}
```

FFT Summary

- Theorem. FFT algorithm evaluates a degree n-1 polynomial at each of the n^{th} roots of unity in O(n log n) steps.
  ↑ assumes n is a power of 2
- Running time. T(2n) = 2T(n) + O(n) ⇒ T(n) = O(n log n).

O(n log n)

$a_0, a_1, \ldots, a_{n-1}$ → $(\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1})$

coefficient representation — point-value representation

Point-Value to Coefficient Representation: Inverse DFT

- Goal. Given the values $y_0, \ldots, y_{n-1}$ of a degree n-1 polynomial at the n points $\omega^0, \omega^1, \ldots, \omega^{n-1}$, find unique polynomial $a_0 + a_1 x + \ldots + a_{n-1} x^{n-1}$ that has given values at given points.

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

↑ Inverse DFT — ↑ Fourier matrix inverse $(F_n)^{-1}$

Inverse FFT

- Claim. Inverse of Fourier matrix is given by following formula.

$$G_n = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \omega^{-3} & \cdots & \omega^{-(n-1)} \\ 1 & \omega^{-2} & \omega^{-4} & \omega^{-6} & \cdots & \omega^{-2(n-1)} \\ 1 & \omega^{-3} & \omega^{-6} & \omega^{-9} & \cdots & \omega^{-3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \omega^{-3(n-1)} & \cdots & \omega^{-(n-1)(n-1)} \end{bmatrix}$$

Inverse FFT: Proof of Correctness

- Claim. F_n and G_n are inverses.
- Pf.

$$(F_n G_n)_{k k'} = \frac{1}{n} \sum_{j=0}^{n-1} \omega^{kj} \omega^{-jk'} = \frac{1}{n} \sum_{j=0}^{n-1} \omega^{(k-k')j} = \begin{cases} 1 & \text{if } k = k' \\ 0 & \text{otherwise} \end{cases}$$

↑ summation lemma

- Summation lemma. Let ω be a principal n^{th} root of unity. Then

$$\sum_{j=0}^{n-1} \omega^{kj} = \begin{cases} n & \text{if } k \equiv 0 \bmod n \\ 0 & \text{otherwise} \end{cases}$$

- Pf.
  - If k is a multiple of n then $\omega^k = 1$ ⇒ sums to n.
  - Each n^{th} root of unity ω^k is a root of $x^n - 1 = (x - 1)(1 + x + x^2 + \ldots + x^{n-1})$.
  - if $\omega^k \neq 1$ we have: $1 + \omega^k + \omega^{k(2)} + \ldots + \omega^{k(n-1)} = 0$ ⇒ sums to 0. ▪

Inverse FFT: Algorithm

```
ifft(n, a0,a1,…,an-1) {
   if (n == 1) return a0

   (e0,e1,…,en/2-1) ← FFT(n/2, a0,a2,a4,…,an-2)
   (d0,d1,…,dn/2-1) ← FFT(n/2, a1,a3,a5,…,an-1)

   for k = 0 to n/2 - 1 {
      ωk ← e^(-2πik/n)
      yk+n/2 ← (ek + ωk dk) / n
      yk+n/2 ← (ek - ωk dk) / n
   }

   return (y0,y1,…,yn-1)
}
```

Inverse FFT Summary

- Theorem. Inverse FFT algorithm interpolates a degree n-1 polynomial given values at each of the n^{th} roots of unity in O(n log n) steps.
  ↑ assumes n is a power of 2

O(n log n)

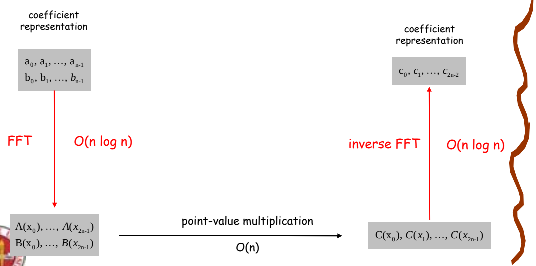
$a_0, a_1, \ldots, a_{n-1}$ ⇄ $(\omega^0, y_0), \ldots, (\omega^{n-1}, y_{n-1})$

O(n log n)

coefficient representation — point-value representation

Polynomial Multiplication

• Theorem. Can multiply two degree n-1 polynomials in O(n log n) steps.

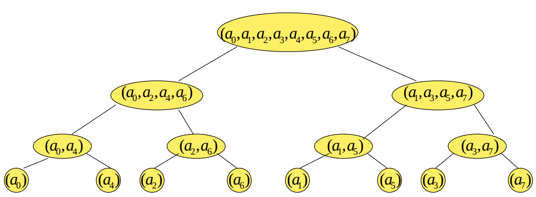


FFT in Practice

- Fastest Fourier transform in the West. [Frigo and Johnson]
  - Optimized C library.
  - Features: DFT, DCT, real, complex, any size, any dimension.
  - Won 1999 Wilkinson Prize for Numerical Software.
  - Portable, competitive with vendor-tuned code.
- Implementation details.
  - Instead of executing predetermined algorithm, it evaluates your hardware and uses a special-purpose compiler to generate an optimized algorithm catered to "shape" of the problem.
  - Core algorithm is nonrecursive version of Cooley-Tukey radix 2 FFT.
  - O(n log n), even for prime sizes.

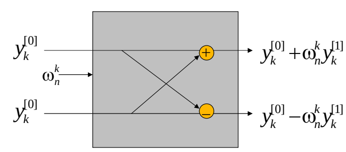
Reference: http://www.fftw.org

More on FFT

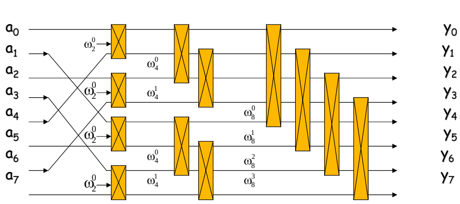


0,4,2,6,1,5,3,7

FFT



Parallel FFT



Integer Multiplication

- Integer multiplication. Given two n bit integers $a = a_{n-1} \ldots a_1 a_0$ and $b = b_{n-1} \ldots b_1 b_0$, compute their product $c = a \times b$.
- Convolution algorithm.
  - Form two polynomials.
  - Note: a = A(2), b = B(2).
  - Compute C(x) = A(x) × B(x).
  - Evaluate C(2) = a × b.
  - Running time: O(n log n) complex arithmetic steps.

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

- Theory. [Schönhage-Strassen 1971] O(n log n log log n) bit operations.
- Practice. [GNU Multiple Precision Arithmetic Library] GMP proclaims to be "the fastest bignum library on the planet." It uses brute force, Karatsuba, and FFT, depending on the size of n.