CSE 548: (*Design and*) Analysis of Algorithms

Divide-and-conquer Algorithms

R. Sekar

Divide-and-Conquer: A versatile strategy

Steps

- Break a problem into subproblems that are smaller instances of the same problem
- Recursively solve these subproblems
- Combine these answers to obtain the solution to the problem

Benefits

Conceptual simplification

Speed up:

- rapidly (exponentially) reduce problem space
- exploit commonalities in subproblem solutions

Parallelism: Divide-and-conquer algorithms are amenable to parallelization

Locality: Their depth-first nature increases locality, extremely important for today's processors.

Topics

Binary Search

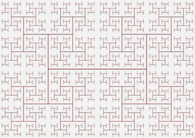
Problem: Find a key k in an ordered collection

Examples: Sorted array $A[n]$: Compare k with $A[n/2]$, then recursively search in $A[0 \cdots (n/2-1)]$ (if $k < A[n/2]$) or $A[n/2 \cdots n]$ (otherwise)

Binary search tree T: Compare k with $root(T)$, based on the result, recursively search left or right subtree of root.

B-Tree: Hybrid of the above two. Root stores an array M of m keys, and has $m+1$ children. Use binary search on M to identify which child can contain k, recursively search that subtree.

H-tree: Planar embedding of full binary tree



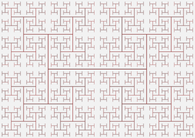
Key properties

- fractal geometry — divide-and-conquer structure
- n nodes in $O(n)$ area
- all root-to-leaf paths equal in length

Applications

- compact embedding of binary trees in VLSI
- hardware clock distribution

Divide-and-conquer Construction of H-tree



$MkHtree(l, b, r, t, n)$

$horizLine(\frac{b+t}{2}, l + \frac{l+r}{4}, r - \frac{l+r}{4})$
$vertLine(l + \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4})$
$vertLine(r - \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4})$
if $n \leq 4$ **return**
$MkHtree(l, \frac{b+t}{2}, \frac{l+r}{2}, t, \frac{n}{4})$
$MkHtree(\frac{l+r}{2}, \frac{b+t}{2}, r, t, \frac{n}{4})$
$MkHtree(l, b, \frac{l+r}{2}, \frac{b+t}{2}, \frac{n}{4})$
$MkHtree(\frac{l+r}{2}, b, r, \frac{b+t}{2}, \frac{n}{4})$

Divide-and-conquer Construction of H-tree

Questions

- How compact is the embedding
  - Ratio of minimum distance between nodes and the average area per node
- What is the root-to-leaf path length?
- Can we do better?
- Finally, how can we show that the algorithm is correct?

$MkHtree(l, b, r, t, n)$

$horizLine(\frac{b+t}{2}, l + \frac{l+r}{4}, r - \frac{l+r}{4})$
$vertLine(l + \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4})$
$vertLine(r - \frac{l+r}{4}, b + \frac{b+t}{4}, t - \frac{b+t}{4})$
if $n \leq 4$ **return**
$MkHtree(l, \frac{b+t}{2}, \frac{l+r}{2}, t, \frac{n}{4})$
$MkHtree(\frac{l+r}{2}, \frac{b+t}{2}, r, t, \frac{n}{4})$
$MkHtree(l, b, \frac{l+r}{2}, \frac{b+t}{2}, \frac{n}{4})$
$MkHtree(\frac{l+r}{2}, b, r, \frac{b+t}{2}, \frac{n}{4})$

Exponentiation

- How many multiplications are required to compute x^n?
- Can we use a divide-and-conquer approach to make it faster?

$ExpBySquaring(n, x)$

if $n > 1$
  $y = ExpBySquaring(\lfloor n/2 \rfloor, x^2)$
  if $odd(n)$ $y = x * y$
  return y
else return x

Merge Sort

function mergesort($a[1\dots n]$)
Input: An array of numbers $a[1\dots n]$
Output: A sorted version of this array

if $n > 1$:
  return merge(mergesort($a[1\dots\lfloor n/2\rfloor]$), mergesort($a[\lfloor n/2\rfloor+1\dots n]$))
else:
  return a

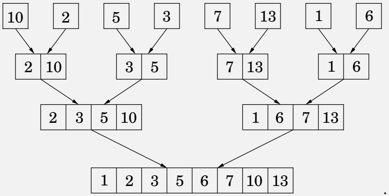
Warmup Sorting Selection Closest pair Multiplication FFT Mergesort Recurrences Fibonacci Numbers Quicksort Lower Bound Radix so

Merge Sort (Continued)

function merge($x[1\dots k], y[1\dots l]$)
if $k = 0$: return $y[1\dots l]$
if $l = 0$: return $x[1\dots k]$
if $x[1] \leq y[1]$:
  return $x[1] \circ$ merge($x[2\dots k], y[1\dots l]$)
else:
  return $y[1] \circ$ merge($x[1\dots k], y[2\dots l]$)

Warmup Sorting Selection Closest pair Multiplication FFT Mergesort Recurrences Fibonacci Numbers Quicksort Lower Bound Radix sort

Merge Sort Illustration



Warmup Sorting Selection Closest pair Multiplication FFT Mergesort Recurrences Fibonacci Numbers Quicksort Lower Bound Radix so

Merge sort time complexity

- *mergesort*(A) makes two recursive invocations of itself, each with an array half the size of A
- *merge*(A, B) takes time that is linear in $|A| + |B|$
- Thus, the runtime is given by the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- In divide-and-conquer algorithms, we often encounter recurrences of the form

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

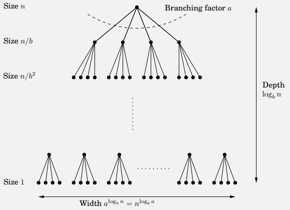
Can we solve them once for all?

Master Theorem

If $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$ for constants $a > 0, b > 1$, and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d), & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Proof of Master Theorem



Can be proved by induction, or by summing up the series where each term represents the work done at one level of this tree.

What if Master Theorem can't be appplied?

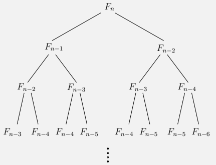
- Guess and check (prove by induction)
  - expand recursion for a few steps to make a guess
  - in principle, can be applied to any recurrence
- Akra-Bazzi method (not covered in class)
  - recurrences can be much more complex than that of Master theorem

More on time complexity: Fibonacci Numbers

function *fib1*(**int** *n*)
if $n = 0$ **return** 0;
if $n = 1$ **return** 1;
return $fib1(n-1) + fib1(n-2)$

- Is this algorithm correct? Yes: follows the defintion of Fibonacci
- What is its runtime?
  - $T(n) = T(n-1) + T(n-2) + 3$, with $T(k) \leq 2$ for $k < 2$
  - Solution is an exponential function . . .
    - Prove this by induction!
- Can we do better?

Structure of calls to *fib1*



- Complete binary tree of depth n, contains 2^n calls to *fib1*
- But there are only n distinct Fibonacci numbers being computed!
  - Each Fibonacci number computed an exponential number of times!

Improved Algorithm for Fibonacci

function *fib2*(n)
- **int** $f[max(2, n+1)]$;
- $f[0] = 0; f[1] = 1$;
- **for** $(i = 2; i \le n; i{+}{+})$
  - $f[i] = f[i-1] + f[i-2]$;
- **return** $f[n]$

- Linear-time algorithm!
- But wait! We are operating on very large numbers
  - n^{th} Fibonacci number requires approx. $0.694n$ bits
    - Prove *this* by induction!
  - Operation on k-bit numbers require k operations
  - i.e., Computing F_n requires $0.694n \log n$ operations

Quicksort

qs(A, l, h) /*sorts $A[l \ldots h]$*/
- **if** $l >= h$ **return**;
- $(h_1, l_2) =$
  - *partition*(A, l, h);
- *qs*(A, l, h_1);
- *qs*(A, l_2, h)

partition(A, l, h)
- $k = selectPivot(A, l, h)$; $p = A[k]$;
- *swap*(A, h, k);
- $i = l - 1$; $j = h$;
- **while** *true* **do**
  - **do** $i{+}{+}$ **while** $A[i] < p$;
  - **do** $j{-}{-}$ **while** $A[j] > p$;
  - **if** $i \ge j$ **break**;
  - *swap*(A, i, j);
- *swap*(A, i, h)
- **return** $(j, i+1)$

Analysis of Runtime of *qs*

General case: Given by the recurrence $T(n) = n + T(n_1) + T(n_2)$
where n_1 and n_2 are the sizes of the two sub-arrays after partition.

Best case: $n_1 = n_2 = n/2$. By master theorem, $T(n) = O(n \log n)$

Worst case: $n_1 = 1$, $n_2 = n - 1$. By master theorem, $T(n) = O(n^2)$

- *A fixed choice of pivot index, say, h, leads to worst-case behavior in common cases, e.g., input is sorted.*

Lucky/unlucky split: Alternate between best- and worst-case splits.

$$\begin{aligned} T(n) &= n + T(1) + \boxed{\text{T(n-1)}} + n \text{ (worst case split)} \\ &= n + 1 + \boxed{\text{(n-1) + 2T((n-1)/2)}} = 2n + 2T((n-1)/2) \end{aligned}$$

which has an $O(n \log n)$ solution.

Three-fourths split:

$$T(n) = n + T(0.25n) + T(0.75n) \le n + 2T(0.75n) = O(n \log n)$$

Average case analysis of *qs*

Define input distribution: All permutations equally likely

Simplifying assumption: all elements are distinct. (Nonessential assumption)

Set up the recurrence: When all permutations are qually likely, the selected pivot has an equal chance of ending up at the i^{th} position in the sorted order, for all $1 \leq i \leq n$. Thus, we have the following recurrence for the average case:

$$T(n) = n + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) + T(n-i))$$

Solve recurrence: Cannot apply the master theorem, but since it seems that we get an $O(n \log n)$ bound even in seemingly bad cases, we can try to establish a $cn \log n$ bound via induction.

Establishing average case of *qs*

- Establish base case. (Trivial.)
- Induction step involves summation of the form $\sum_{i=1}^{n-1} i \log i$.

  Attempt 1: bound $\log i$ above by $\log n$. (Induction fails.)

  Attempt 2: split the sum into two parts:

  $$\sum_{i=1}^{n/2} i \log i + \sum_{i=n/2+1}^{n-1} i \log i$$

  and apply the approx. to each half. (Succeeds with $c \geq 4$.)

  Attempt 3: replace the summation with the upper bound

  $$\int_{x=1}^{n} x \log x = \frac{x^2}{2}\left(\log x - \frac{1}{2}\right)\Bigg|_{x=1}^{n}$$

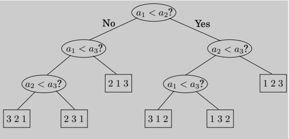
  (Succeeds with the constraint $c \geq 2$.)

Randomized Quicksort

- Picks a pivot at random
- What is its complexity?
  - For randomized algorithms, we talk about *expected complexity*, which is an average over all possible values of the random variable.
- If pivot index is picked uniformly at random over the interval $[l, h]$, then:
  - every array element is equally likely to be selected as the pivot
  - every partition is equally likely
  - thus, *expected* complexity of *randomized* quicksort is given by the same recurrence as the *average* case of *qs*.

Lower bounds for comparison-based sorting

- Sorting algorithms can be depicted as trees: each leaf identifies the input permutation that yields a sorted order.



- The tree has $n!$ leaves, and hence a height of $\log n!$. By Stirling's approximation, $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, so, $\log n! = O(n \log n)$
- No *comparison-based* sorting algorithm can do better!

Bucket sort

Overview

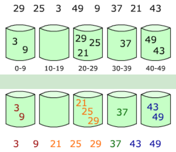
Divide: Partition input into intervals (buckets), based on key values

- Linear scan of input, drop into appropriate bucket

Recurse: Sort each bucket

Combine: Concatenate bin contents

Example



Images from Wikipedia commons

Bucket sort (Continued)

- Bucket sort generalizes quicksort to multiple partitions
  - Combination = concatenation
  - Worst case quadratic bound applies
  - But performance can be much better if input distribution is uniform. *Exercise:* What is the runtime in this case?
- Used by letter sorting machines in post offices

Counting Sort

Special case of bucket sort where each bin corresponds to an interval of size 1.

- No need to recurse. Divide = conquered!
- Makes sense only if range of key values is small (usually constant)
- Thus, counting sort can be done in $O(n)$ time!
  - *Hmm. How did we beat the $O(n \log n)$ lower bound?*

Radix Sorting

- Treat an integer as a sequence of digits
- Sort digits using counting sort

  LSD sorting: Sort first on least significant digit, and most significant digit last. After each round of counting sort, results can be simply concatenated, and given as input to the next stage.

  MSD sorting: Sort first on most significant digit, and least significant digit last. Unlike LSD sorting, we cannot concatenate after each stage.
- *Note:* Radix sort does not divide inputs into smaller subsets

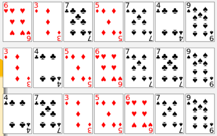
  If you think of input as multi-dimensional data, then we break down the problem to each dimension.

Stable sorting algorithms

- *Stable sorting algorithms:* don't change order of equal elements.
- Merge sort and LSD sort are stable. Quicksort is not stable.

Why is stability important?

- Effect of sorting on attribute A and then B is the same as sorting on $\langle B, A\rangle$
- LSD sort won't work without this property!
- Other examples: sorting spread sheets or tables on web pages



Images from Wikipedia Commons

Sorting strings

- Can use LSD or MSD sorting
  - Easy if all strings are of same length.
  - Requires a bit more care with variable-length strings.
    Starting point: use a special terminator character $t < a$ for all valid characters a.
- Easy to devise an $O(nl)$ algorithm, where n is the number of strings and l is the maximum size of any string.
  - But such an algorithm is *not* linear in input size.
- *Exercise:* Devise a linear-time string algorithm.
  Given a set $\mathcal{S}$ of strings, your algorithm should sort in $O(|\mathcal{S}|)$ time, where

$$|\mathcal{S}| = \sum_{s\in\mathcal{S}} |s|$$

Select k^{th} largest element

Obvious approach: Sort, pick k^{th} element — wasteful, $O(n \log n)$

Better approach: Recursive partitioning, search only on one side

$qsel(A, l, h, k)$

if $l = h$ **return** $A[l]$;
$(h_l, l_2) = partition(A, l, h)$;
if $k \le h_l$
  return $qsel(A, l, h_l, k)$
else return $qsel(A, l_2, h, k)$

Complexity

Best case: Splits are even:
$T(n) = n + T(n/2)$, which has an $O(n)$ solution.

Skewed 10%/90% $T(n) \le n + T(0.9n)$ — still linear

Worst case: $T(n) = n + T(n-1)$ — *quadratic!*

Worst-case $O(n)$ Selection

Intuition: Spend a bit more time to select a pivot that ensures reasonably balanced partitions

MoM Algorithm [Blum, Floyd, Pratt, Rivest and Tarjan 1973]

Time Bounds for Selection

by

Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan

Abstract

The number of comparisons required to select the i-th smallest of n numbers is shown to be at most a linear function of n by analysis of a new selection algorithm -- PICK. Specifically, no more than 5.4305 n comparisons are ever required. This bound is improved for

$O(n)$ Selection: MoM Algorithm

- Quick select (*qsel*) takes no time to pick a pivot, but then spends $O(n)$ to partition.
- Can we spend more time upfront to make a better selection of the pivot, so that we can avoid highly skewed splits?

Key Idea

- Use the selection algorithm itself to choose the pivot.
  - Divide into sets of 5 elements
  - Compute median of each set ($O(5)$, i.e., constant time)
  - Use selection recursively on these $n/5$ elements to pick their median
    - i.e., choose the median of medians (MoM) as the pivot
- Partition using MoM, and recurse to find kth largest element.

$O(n)$ Selection: MoM Algorithm

Theorem: MoM-based split won't be worse than 30%/70%

Result: Guaranteed linear-time algorithm!

Caveat: The constant factor is non-negligible; use as fall-back if random selection repeatedly yields unbalanced splits.

Selecting maximum element: Priority Queues

Heap

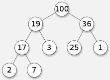
- A tree-based data structure for priority queues

  Heap property: H is a heap if for every subtree h of H

  $$\forall k \in keys(h) \quad root(h) \geq k$$

  where $keys(h)$ includes all keys appearing within h

  Note: No ordering of siblings or cousins
- Supports *insert*, *deleteMax* and *max*.
- Typically implemented using arrays, i.e., without an explicit tree data structure



Task of maintaining max is distributed to subsets of the entire set; alternatively, it can be thought of as maintaining several parallel queues with a single head.

Images from Wikimedia Commons

Binary heap

Array representation: Store heap elements in breadth-first order in the array. Node i's children are at indices $2 * i$ and $2 * i + 1$

- Conceptually, we are dealing with a balanced binary tree

Max: Element at the root of the array, extracted in $O(1)$ time

DeleteMax: Overwrite root with last element of heap. Fix heap – takes $O(\log n)$ time, since only the ancestors of the last node need to be fixed up.

Insert: Append element to the end of array, fix up heap

MkHeap: Fix up the entire heap. Takes $O(n)$ time.

Heapsort: $O(n \log n)$ algorithm, *MkHeap* followed by n calls to *DeleteMax*

Finding closest pair of points

Problem: Given a set of n points in a d-dimensional space, identify the two that have the smallest Euclidean distance between them.

Applications: A central problem in graphics, vision, air-traffic control, navigation, molecular modeling, and so on.

Images from Wikipedia Commons

Divide-and-conquer closest pair (2D)

Divide: Identify k such that the line $x = k$ divides the points evenly.
(Median computation, takes $O(n)$ time.)

Recursive case: Find closest pair in each half.

Combine:

- Can't just take the min of the closest pairs from two halves.
- Need to consider pairs across the divide line — seems that this will take $O(n^2)$ time!

Speeding up search for cross-region pairs

Observation (Key Observation 1)

- *Let δ_1 and δ_2 be the minimum distances in each half.*
- *Need only consider points within $\delta = \min(\delta_1, \delta_2)$ from the dividing line*

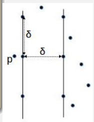
- We expect that only a small number of points will be within such a narrow strip.
- *But in the worst case, every point could be within the strip!*

Sparsity condition

Consider a point p on the left δ-strip. How many points $q_1, \ldots, q_r$ on the right δ-strip could be within δ from p?

Observation (Key Observation 2)

- *$q_1, \ldots, q_r$ should all be within a rectangular $2\delta \times \delta$ as shown*
- *r can't be too large: $q_1, \ldots, q_r$ will crowd together, closer than δ*
- *Theorem:* $r \leq 6$



We need to consider at most $6n$ cross-region pairs!

Remains $O(n)$ in higher dimensions as well

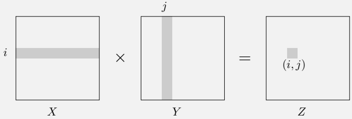
Closest pair: Summary

- *Recurrence:* $T(n) = 2T(n/2) + \Omega(n)$, since median computation is already linear-time. Thus, $T(n) = \Omega(n \log n)$.
- To get to $O(n \log n)$, need to
  1. compute the δ-strip in $O(n)$ time
     - Keep the points in each region sorted in x-dimension
     - Takes an additional $O(n \log n)$ time, no change to overall complexity
  2. compute $q_1, \ldots, q_6$ in $O(1)$ time.
     - keep points in each region sorted *also* in y-dimension
     - maintain this order while deleting points outside δ strip
     - in this list, for each p, consider only 12 neighbors — 6 on each side of divide

Matrix Multiplication

The product Z of two $n \times n$ matrices X and Y is given by

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj} \quad \text{— leads to an } O(n^3) \text{ algorithm.}$$



Divide-and-conquer Matrix Multiplication

Divide X and Y into four $n/2 \times n/2$ submatrices

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Recursively invoke matrix multiplication on these submatrices:

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

Divided, but did not conquer! $T(n) = 8T(n/2) + O(n^2)$, which is still $O(n^3)$

Strassen's Matrix Multiplication

Strassen showed that 7 multiplications are enough:

$$XY = \begin{bmatrix} P_6 + P_5 + P_4 - P_2 & P_1 + P_2 \\ P_3 + P_4 & P_1 - P_3 + P_5 - P_7 \end{bmatrix} \quad \text{where}$$

$$\begin{aligned} P_1 &= A(F - H) & P_5 &= (A + D)(E + H) \\ P_2 &= (A + B)H & P_6 &= (B - D)(G + H) \\ P_3 &= (C + D)E & P_7 &= (A - C)(E + F) \\ P_4 &= D(G - E) \end{aligned}$$

Now, the recurrence $T(n) = 7T(n/2) + O(n^2)$ has $O(n^{\log_2 7} = n^{2.81})$ solution!

Best-to-date complexity is about $O(n^{2.4})$, but this algorithm is not very practical.

Karatsuba's Algorithm

Same high-level strategy as Strassen — but predates Strassen.

Divide: n-digit numbers into halves, each with $n/2$-digits:

$$\begin{aligned} a &= \boxed{a_1}\boxed{a_0} = 2^{n/2}a_1 + a_0 \\ b &= \boxed{b_1}\boxed{b_0} = 2^{n/2}b_1 + b_0 \\ ab &= 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + b_1 a_0) + a_0 b_0 \end{aligned}$$

Key point — Instead of 4 multiplications, we can get by with 3 since:

$$a_1 b_0 + b_1 a_0 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0$$

Recursively compute $a_1 b_1$, $a_0 b_0$ and $(a_1 + a_0)(b_1 + b_0)$.

Recurrence $T(n) = 3T(n/2) + O(n)$ has an $O(n^{\log_2 3} = n^{1.59})$ solution!

Note: This trick for using 3 (not 4) multiplications noted by Gauss (1777-1855) in the context of complex numbers.

Toom-Cook Multiplication

- Generalize Karatsuba
  - Divide into $n > 2$ parts
- Can be more easily understood when integer multiplication is viewed as a polynomial multiplication.

Integer Multiplication Revisited

- An integer represented using digits
  $$a_{n-1} \ldots a_0$$
  over a base d (i.e., $0 \le a_i < d$) is very similar to the polynomial
  $$A(x) = \sum_{i=0}^{n-1} a_i x^i$$
  Specifically, the value of the integer is $A(d)$.
- Integer multiplication follows the same steps as polynomial multiplication:
  $$a_{n-1} \ldots a_0 \times b_{n-1} \ldots b_0 = (A(x) \times B(x))(d)$$

Polynomials: Basic Properties

Horner's rule

An n^{th} degree polynomial $\sum_{i=0}^{n} a_i x^i$ can be evaluated in $O(n)$ time:

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

Roots and Interpolation

- An n^{th} degree polynomial $A(x)$ has exactly n roots $r_1, \ldots, r_n$. In general, r_i's are complex and need not be distinct.
- It can be represented as a product of sums using these roots:
  $$A(x) = \sum_{i=1}^{n} a_i x^i = \prod_{i=0}^{n} (x_i - r_i)$$
- Alternatively, $A(x)$ can be specified uniquely by specifying $n+1$ points (x_i, y_i) on it, i.e., $A(x_i) = y_i$.

Operations on Polynomials

Representation	Add	Mult
Coefficients	$O(n)$	$O(n^2)$
Roots	?	$O(n)$
Points	$O(n)$	$O(n)$

Note: Point representation is the best for computation! But usually, only the coefficients are given

Solution: Convert to point form by *evaluating* $A(x)$ at selected points.

But conversion defeats the purpose: requires $O(n)$ evaluations, each taking $O(n)$ time, thus we are back to $O(n^2)$ total time.

Toom (and FFT) Idea: Choose evaluation points judiciously to speed up evaluation

Matrix representation of Polynomial Evaluation

Given a polynomial

$$A(x) = \sum_{t=0}^{n-1} a_t x^n$$

choose m points $x_0, \ldots, x_m$ for its evaluation.

Evaluation can be expressed using matrix multiplication:

$$\begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ \vdots \\ p_m \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Multiplication using Point Representation

- Let $A(x)$ and $B(x)$ be polynomials representing two numbers
- Evaluate both polynomials at chosen points $x_0, \ldots x_m$

  $$P = \mathbf{X}A \qquad Q = \mathbf{X}B$$

  where $P, \mathbf{X}, A, Q$ and B denote matrices as in last page
- Compute point-wise product

  $$\begin{bmatrix} r_0 \\ \vdots \\ r_m \end{bmatrix} = \begin{bmatrix} p_0 * q_0 \\ \vdots \\ p_m * q_m \end{bmatrix}$$
- Compute polynomial C corresponding to R

  $$R = \mathbf{X}C \Rightarrow C = \mathbf{X}^{-1}R$$
- To avoid overflow, m should be $degree(A) + degree(B) + 1$ for R

Improving complexity ...

- Key problem: Complexity of computing $\mathbf{X}$ and its inverse $\mathbf{X}^{-1}$
- Toom strategy:
  - Use low-degree polynomials e.g., Toom-2 = Karatsuba uses degree 1.
    - represents an n-bit number as a 2-digit number over a large base $d = 2^{n/2}$
  - Fix evaluation points for a given degree polynomial so that $\mathbf{X}$ and $\mathbf{X}^{-1}$ can be precomputed
    - For Toom-2, $x_0 = 0, x_1 = 1, x_2 = \infty$. (Define $A(\infty) = a_{n-1}$.)
  - Choose points so that expensive multiplications can be avoided while computing $P = \mathbf{X}A$, $Q = \mathbf{X}B$ and $C = \mathbf{X}^{-1}R$
- Toom-N on n-digit numbers needs $2N - 1$ multiplications on n/N digit numbers:

  $$T(n) = (2N-1)T(n/N) + O(n)$$

  which, by Master theorem, has a solution $O(n^{\log_N(2N-1)})$ solution

Karatsuba revisited as Toom-2

Given evaluation points $x_0 = 0, x_1 = 1, x_2 = \infty$,

$$\mathbf{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \qquad \mathbf{X}A = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ 0 \end{bmatrix} = \begin{bmatrix} a_0 \\ a_0 + a_1 \\ a_1 \end{bmatrix}$$

Similarly

$$\mathbf{X}B = \begin{bmatrix} b_0 \\ b_0 + b_1 \\ b_1 \end{bmatrix}$$

Point-wise multiplication yields:

$$R = \begin{bmatrix} a_0 b_0 \\ (a_0 + a_1)(b_0 + b_1) \\ a_1 b_1 \end{bmatrix}$$

and so on ...

Limitations of Toom

- In principle, complexity can be reduced to $n^{1+\epsilon}$ for arbitrarily small positive ϵ by increasing N
- In reality, the algorithm itself depends on the choice of N. Specifically, constant factors involved increase rapidly with N.
- As a practical matter, $N = 4$ or 5 is where we stop.
- Question: Can we go farther?

FFT and Schonhage-Strassen

- Key idea: evaluate polynomial on the complex plane
- Choose powers of Nth complex root of unity as the points for evaluation
- Enables sharing of operations in computing $\mathbf{X}A$ so that it can be done in $O(N \log N)$ time, rather than $O(N^2)$ time needed for the naive matrix-multiplication based approach

FFT to the Rescue!

Matrix form of DFT and interpretation as polynomial evaluation:

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{N-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{N-1} \end{bmatrix}$$

- *Voila!* FFT computes $A(x)$ at N points ($x_i = \omega^i$) in $O(N \log N)$ time!
- $O(N \log N)$ integer multiplication

Convert to point representation using FFT	$O(N \log N)$
Multiply on point representation	$O(N)$
Convert back to coefficients using FFT^{-1}	$O(N \log N)$

FFT to the Rescue!

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

FFT can be thought of as a clever way to choose points:

- Evaluations at many distinct points “collapse” together
- This is why we are left with $2T(n/2)$ work after division, instead of $4T(n/2)$ for a naive choice of points.

FFT-based multiplication: More careful analysis ...

- Computations on complex or real numbers can lose precision.
  - For integer operations, we should work in some other ring — usually, we choose a ring based on modulo arithmetic.
  - Ex: in mod 33 arithmetic, 2 is the 10^{th} root of 1, i.e., $2^{10} \equiv 1 \mod 33$

    More generally, 2 is the nth root of unity modulo $(2^{n/2}+1)$
- Point-wise additions and multiplications are not $O(1)$.
  - We are adding up to n numbers (“digits”) — we need $\Omega(\log n)$ bits
  - So, total cost increases by at least $\log n$, i.e., $O(n \log^2 n)$.
- [Schonhage-Strassen ’71] developed $O(n \log n \log \log n)$ algorithm: recursively apply their technique for “inner” operations.

Integer Multiplication Summary

- Algorithms implemented in libraries for arbitrary precision arithmetic, with applications in public key cryptography, computer algebra systems, etc.
- GNU MP is a popular library, uses various algorithms based on input size: naive, Karatsuba, Toom-3, Toom-4, or Schonhage-Strassen (at about 50K digits).
- Karatsuba is Toom-2. Toom-N is based on
  - Evaluating a polynomial at $2N$ points,
  - performing point-wise multiplication, and
  - interpolating to get back the polynomial, while
  - minimizing the operations needed for interpolation

Fast Fourier Transformation

One of the most widely used algorithms — yet most people are unaware of its use!

Solving differential equations: Applied to many computational problems in engineering, e.g., heat transfer

Audio: MP3, digital audio processors, music/speech synthesizers, speech recognition, ...

Image and video: JPEG, MPEG, vision, ...

Communication: modulation, filtering, radars, software-defined radios, H.264, ...

Medical diagnostics: MRI, PET, ultrasound, ...

Quantum computing: See text Ch. 10

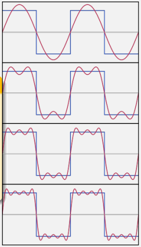
Other: Optics, data compression, seismology, ...

Fourier Series

Theorem (Fourier Theorem)

Any (sufficiently smooth) function with a period T can be expressed as a sum of series of sinusoids with periods T/n for integral n.

$$a(t) = \sum_{n=0}^{\infty} (d_n \sin(2\pi nt/T) + e_n \cos(2\pi nt/T))$$



Images from Wikimedia Commons.

Fourier Series

Using the identity

$$e^{ix} = \cos x + i \sin x$$

Fourier series becomes

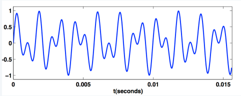
$$a(t) = \sum_{n=-\infty}^{\infty} c_n e^{2\pi int/T}$$

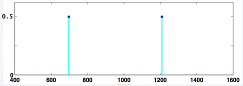
It can be shown

$$c_n = \int_0^T a(t) e^{-2\pi int} dt$$

For real $a(t)$, $c_n = c^*_{-n}$.

Example: Touch tone button 1





Images from Cleve Moler, Numerical computing with MATLAB

Fourier Transform

- What if a is not periodic?
- May be we can start with the Fourier series definition for c_n

$$c_n = \int_0^T a(t) e^{-2\pi int} dt$$

and let $T \to \infty$?

- Frequencies are not discrete any more, as the "fundamental frequency" $f = 1/T \to 0$
- Instead of discrete coefficients c_n, we will have a continuous function — call it $s(f)$.

$$s(f) = \int_{\infty}^{\infty} a(t) e^{-2\pi ift} dt$$

- $\mathcal{F}(a)$ denotes a's Fourier transform
- $\mathcal{F}$ is almost self-inverting: $\mathcal{F}(\mathcal{F}(a(t))) = a(-t)$

How do Fourier Series/Transform help?

Differential equations: Turn non-integrable functions into a sum of easily integrable ones.

Some problems easier to solve in frequency domain:

Filtering: filter out noise, tuning, ...

Compression: eliminate high frequency components, ...

Convolution: Convolution in time domain becomes (simpler) multiplication in frequency domain.

Definition (Convolution)

$$(a * b)(t) = \int_{-\infty}^{\infty} a(t - x) b(x) dx$$

Theorem (Convolution)

$$\mathcal{F}(a * b)(t) = \mathcal{F}(a(t))\mathcal{F}(b(t))$$

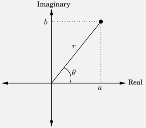
Discrete Fourier Transform

- Real-world signals are typically sampled
  - DFT is a formulation of FT applicable to such samples
- *Nyquist rate:* A signal with highest frequency $n/2$ can be losslessly reconstructed from n samples.
- DFT of time domain samples $a_0, \ldots, a_{n-1}$ yields frequency domain samples $s_0, \ldots, s_{n-1}$:

$$s_f = \sum_{t=0}^{n-1} a_t e^{-2\pi i f t/n} \qquad \text{cf. } s(f) = \int_{\infty}^{\infty} a(t) e^{-2\pi i f t} dt$$

Note: DFT formulation can be derived from FT by treating the sampling process as a multiplication by a sequence of impulse functions separated by the sampling interval

Background: Complex Plane, Polar Coordinates



The complex plane

$z = a + bi$ is plotted at position (a, b).

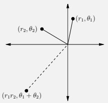
Polar coordinates: rewrite as $z = r(\cos\theta + i\sin\theta) = re^{i\theta}$, denoted (r, θ).

- *length* $r = \sqrt{a^2 + b^2}$.
- *angle* $\theta \in [0, 2\pi)$: $\cos\theta = a/r, \sin\theta = b/r$.
- θ can always be reduced modulo 2π.

Examples:

Number	-1	i	$5+5i$
Polar coords	$(1, \pi)$	$(1, \pi/2)$	$(5\sqrt{2}, \pi/4)$

Polar Coordinates and Multiplication

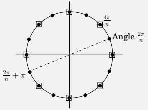


Multiply the lengths and add the angles:
$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.

For any $z = (r, \theta)$,

- $-z = (r, \theta + \pi)$ since $-1 = (1, \pi)$.
- If z is on the *unit circle* (i.e., $r = 1$), then $z^n = (1, n\theta)$.

Roots of unity on Complex Plane



Solutions to the equation $z^n = 1$.

By the multiplication rule: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$ (shown here for $n = 16$).

For even n:

- These numbers are *plus-minus paired*: $-(1, \theta) = (1, \theta + \pi)$.
- Their squares are the $(n/2)$nd roots of unity, shown here with boxes around them.

Matrix representation of DFT

- Given time domain samples a_t for $t = 0, 1, \ldots, n-1$,
- Compute frequency domain samples s_f for $f = 0, 1, \ldots, n-1$

$$s_f = \sum_{t=0}^{n-1} a_t e^{-2\pi i f t/n} = \sum_{t=0}^{n-1} a_t \left(e^{-2\pi i/n}\right)^{ft} = \sum_{t=0}^{n-1} a_t \omega^{ft}$$

where $\omega = e^{-2\pi i/n}$ is the nth complex root of unity

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Matrix representation of DFT

Note that $e^{-2\pi i/n}$ is represents the n^{th} root of 1, denoted ω.

$$\begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Possible interpretations of these matrix equations:

- Simultaneous equations that can be solved
- Change of basis (rotate coordinate system)
- Evaluation of polynomial $\sum_{k=0}^{n-1} a_k x^k$ at $x = \omega^j, 0 \le j < n$.

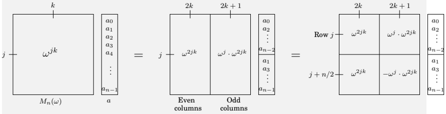
Speeding up FFT Computation

- Matrix multiplication formulation has an obvious divide-and-conquer implementation

$$M_n \vec{A}_{0\ldots(n-1)} = \begin{bmatrix} M^{11}_{n/2} & M^{12}_{n/2} \\ M^{21}_{n/2} & M^{22}_{n/2} \end{bmatrix} \begin{bmatrix} \vec{A}_{0\cdots\lfloor n/2\rfloor} \\ \vec{A}_{\lceil n/2\rceil\cdots(n-1)} \end{bmatrix}$$

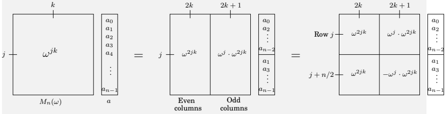
  But this algorithm still takes $O(n^2)$ time
- *... but wait!* — there are only $O(n)$ distinct elements in the square matrix M_n.
- $O(n)$ repetitions of each element in M_n, so there is significant scope for sharing operations on submatrices!

Observations about $\mathbf{M}(\omega)$

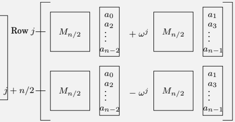


- Two successive colums differ by a factor ω^j in the j^{th} row
- Rows that are $n/2$ rows apart differ by a factor of $\omega^{kn/2}$ in the k^{th} column
  - Note that $\omega^{n/2} = -1$, so they differ by a factor of -1 on odd columns, and are identical on even columns.

DFT Matrix Multiplication, Rearranged ...



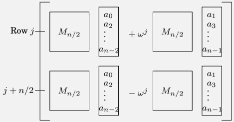
- Only two subproblems of size $n/2$: Multiply $M_{n/2}$ by $\overrightarrow{A}_{odd}$ and $\overrightarrow{A}_{even}$: $T(n) = 2T(n/2) + O(n)$, with an $O(n \log n)$ solution.
- *But wait!* M_n has $O(n^2)$ size, how can we operate on it in $O(n)$ time?



FFT Algorithm

function FFT(a, ω)
Input: An array $a = (a_0, a_1, \ldots, a_{n-1})$, for n a power of 2
A primitive nth root of unity, ω
Output: $M_n(\omega)\, a$

if $\omega = 1$: return a
$(s_0, s_1, \ldots, s_{n/2-1}) = \mathrm{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
$(s'_0, s'_1, \ldots, s'_{n/2-1}) = \mathrm{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
for $j = 0$ to $n/2 - 1$:
$r_j = s_j + \omega^j s'_j$
$r_{j+n/2} = s_j - \omega^j s'_j$
return $(r_0, r_1, \ldots, r_{n-1})$



Convolution in the Discrete World

$$(\overrightarrow{A_n} * \overrightarrow{B_m})_t = \sum_{x=0}^{m-1} a_{t-x} b_x \qquad \text{cf. } (a*b)(t) = \int_{-\infty}^{\infty} a(t-x)b(x)dx$$

Linear convolution: $a_{t-x} = 0$ if $x > t$

Circular convolution: $a_{t-x} = a_{t-x+n}$ if $x > n$. (Equivalent to treating A as a periodic function.)

Zero-extended convolution: First extend A and B to have $m + n - 1$ samples by letting $A_y = 0$ for $m \le y < m + n$ and $B_z = 0$ for $n \le z < m + n$.

With zero-extension, the definitions of linear and ciucular conventions match, and hence become eqivalent. Hence, we will deal only with zero-extended convolution.

Theorem (Discrete Convolution $\mathcal{F}(\overrightarrow{A_n} * \overrightarrow{B_m}) = \mathcal{F}(\overrightarrow{A_n})\mathcal{F}(\overrightarrow{B_m})$**)**

Why this fascination with convolution?

- Computationally, convolution is a loop to add products
- The convolution theorem says we can replace this $O(n)$ loop by a single operation on the DFT. *That is fascinating!*
  - *Wait a minute!* What about the cost of computing $\mathcal{F}$ first?
- If we use FFT, then we the computation of $\mathcal{F}$ and its inversion will still obe $O(n \log n)$, not quadratic.

- Can we use FFT as a building block to speed up algorithms for other problems?
  - Integer multiplication looks like a convolution, and usually takes $O(n^2)$. Can we make it $O(n \log n)$?

FFT to the Rescue!

Matrix form of DFT and interpretation as polynomial evaluation:

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

- *Voila!* FFT computes $A(x)$ at n points ($x_i = \omega^i$) in $O(n \log n)$ time!
- $O(n \log n)$ integer multiplication

Convert to point representation using FFT	$O(n \log n)$
Multiply on point representation	$O(n)$
Convert back to coefficients using FFT^{-1}	$O(n \log n)$

FFT to the Rescue!

$$\begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_j \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_j \\ \vdots \\ a_{n-1} \end{bmatrix}$$

FFT can be thought of as a clever way to choose points:

- Evaluations at many distinct points "collapse" together
- This is why we are left with $2T(n/2)$ work after division, instead of $4T(n/2)$ for a naive choice of points.

FFT-based Multiplication: Summary

- FFT works with 2^k points — Increases work by up to 2x.
  - Product of two n^{th}-degree polynomial has degree $2n$
    - We need to work with $2n$ points, i.e., 4x increase in time.
- Requires inverting to coefficient representation after multiplication:

$$\begin{aligned} \overrightarrow{S_n} &= M_n(\omega)\overrightarrow{A_n} \\ M_n^{-1}(\omega)\overrightarrow{S_n} &= M_n^{-1}(\omega)M_n(\omega)\overrightarrow{A_n} = \overrightarrow{A_n} \end{aligned}$$

It is easy to show that $M_n^{-1}(\omega) = M_n(-\omega)/n$, and hence:

$$\overrightarrow{A}_n * \overrightarrow{B}_n = FFT(FFT(\overrightarrow{A}_{2n}, \omega) \cdot FFT(\overrightarrow{B}_{2n}, \omega), \omega^{-1})/n$$

We are back to the convolution theorem!

More careful analysis ...

- Computations on complex or real numbers can lose precision.
  - For integer operations, we should work in some other ring — usually, we choose a ring based on modulo arithmetic.
  - Ex: in mod 33 arithmetic, 2 is the 10^{th} root of 1, i.e., $2^{10} \equiv 1 \mod 33$

More generally, 2 is the nth root of unity modulo $(2^{n/2}+1)$

- Point-wise additions and multiplications are not $O(1)$.
  - We are adding up to n numbers ("digits") — we need $\Omega(\log n)$ bits
  - So, total cost increases by at least $\log n$, i.e., $O(n \log^2 n)$.
- [Schonhage-Strassen '71] developed $O(n \log n \log\log n)$ algorithm: recursively apply their technique for "inner" operations.

Integer Multiplication Summary

- Algorithms implemented in libraries for arbitrary precision arithmetic, with applications in public key cryptography, computer algebra systems, etc.
- GNU MP is a popular library, uses various algorithms based on input size: naive, Karatsuba, Toom-3, Toom-4, or Schonhage-Strassen (at about 50K digits).
- Karatsuba is Toom-2. Toom-N is based on
  - Evaluating a polynomial at $2N$ points,
  - performing point-wise multiplication, and
  - interpolating to get back the polynomial, while
  - minimizing the operations needed for interpolation