Portland State University

W'21 CS 584/684 Algorithm Design & Analysis

Fang Song

Lecture 2

- Recursion
- Mergesort
- Divide-&-Conquer
- Multiplication

Logistics

- **TA office hours**
  - M/W 11 - 12. Zoom links on Calendar.
- **Calendar: updated title on due dates**

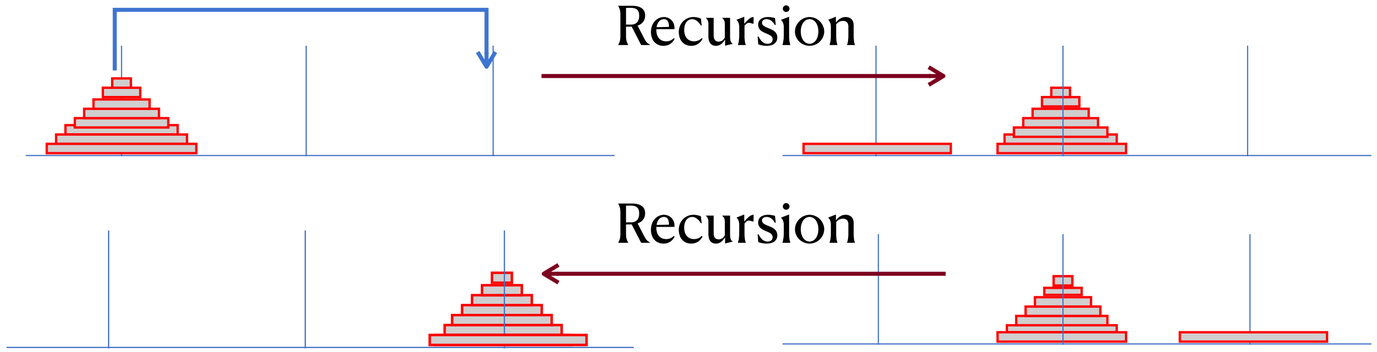
Recall: algorithmic techniques

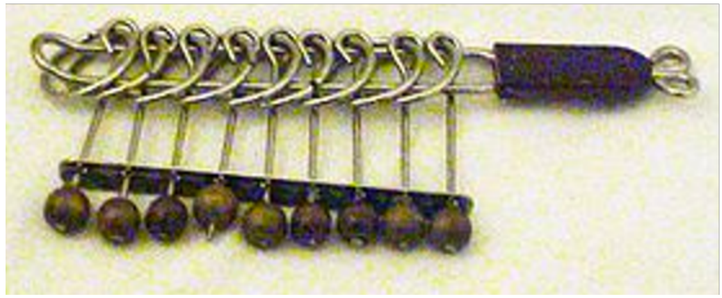
★Reduction: a meta technique, always keep in mind.

- Brute force
- Divide and conquer
  - Decompose a problem into smaller sub-problems and compose the solutions.
- Dynamic programming
  - Memorize soln's to subproblems that occur repeatedly.
- Greediness
  - Make a local optimal choice for subproblems.
- Randomization
- ... Creativity

Recursion: "self" reduction

- Hanoi tower





https://youtu.be/cHxJFMsvAII

- Simplify and delegate
  - If the given instance of the problem can be solved directly, do it!
  - Otherwise, reduce it to one or more simpler instances of the same problem.
- Induction (in disguise)
  - Base case
  - Induction hypothesis

... and think no further. Imagine a Recursion Fairy solves the subproblems for you.

Sorting

Given: n elements (numbers, letters, ...) $a[1, \ldots, n]$

Goal: rearrange in ascending order

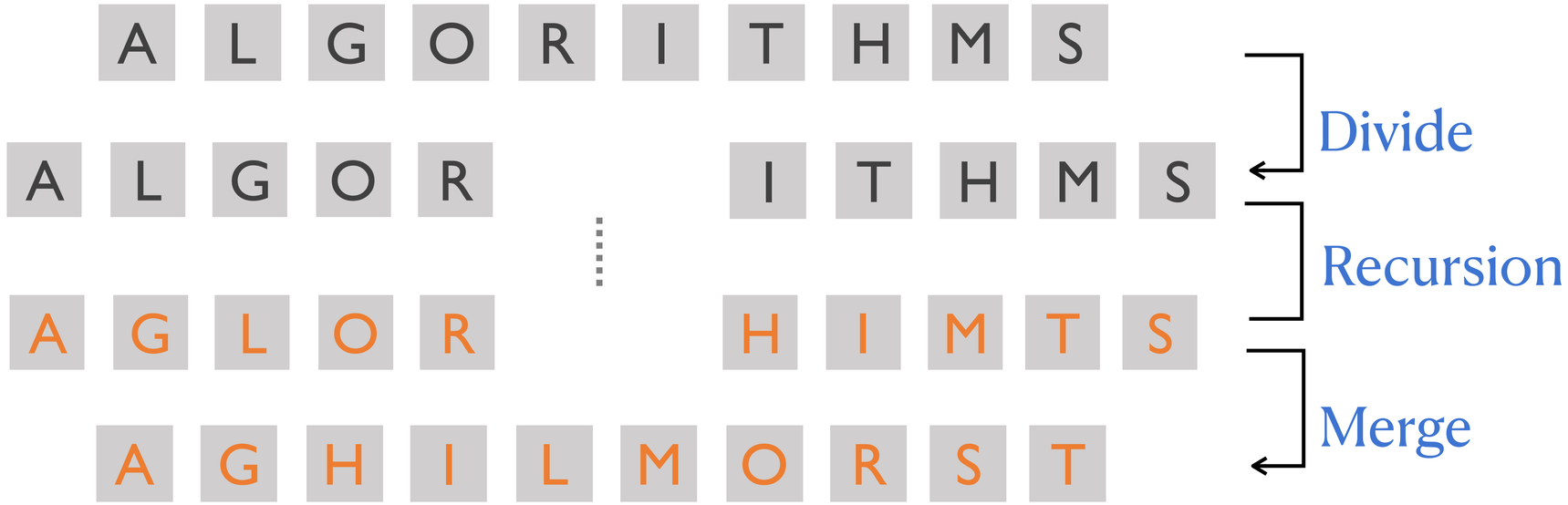
- **Applications**
  - Display Google page rank results
  - Find the median
  - Binary search in a database
  - Data compression
  - Computational biology
  - ...

Exercise. Name your familiar sorting algorithms.

Merge sort

- **Main idea**
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make a sorted whole.

John von Neumann, 1945



Merging

Given: two **sorted** sublists

Goal: combine into a **sorted** whole

- **How to merge efficiently?**
  - Use temporary array
  - Store smaller of L/R, and continue to the next in that list

Write up your algorithm

Problem description

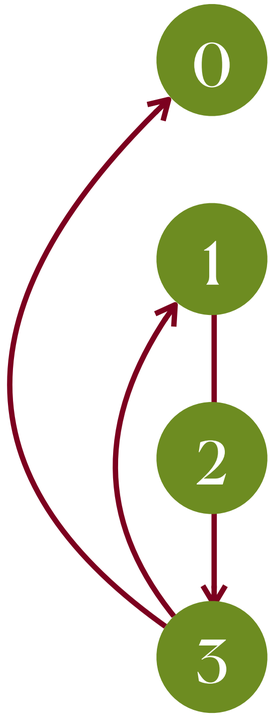
- **Input:** a list of n letters $a[1,\ldots,n]$
- **Output :** $a[1,\ldots,n]$ sorted, i.e.,

$a[i] \le a[j], \forall i < j \in [n]$

```
MergeSort(a[1,...,n]):
 if n > 1
    m ← ⌊n/2⌋
    MergeSort(a[1,...,m] // recursion
    MergeSort(a[m + 1,...,n]) // recursion
    Merge(a[1,...,n], m)
```

```
Merge(a[1,...,n], m):
 i ← 1; j ← m + 1
 for k ← 1 to n
  if j > n
     b[k] ← a[i]; i ← i + 1
  else if i > m
     b[k] ← a[j]; j ← j + 1
  else if a[i] < a[j]
     b[k] ← a[i]; i ← i + 1
  else
     b[k] ← a[j]; j ← j + 1
 for k ← 1 to n
  a[k] ← b[k]
```

Recall: Principal questions



What problem to solve?

Is the algorithm correct?

How much resource it costs?

Can we do better?

DESIGN ✓

ANALYSIS

Correctness of MergeSort

Think of reduction again

```
MergeSort(a[1,…,n]):
 if n > 1
   m ← ⌊n/2⌋
   MergeSort(a[1,…,m] // recursion
   MergeSort(a[m + 1,…,n]) // recursion
   Merge(a[1,…,n], m)
```

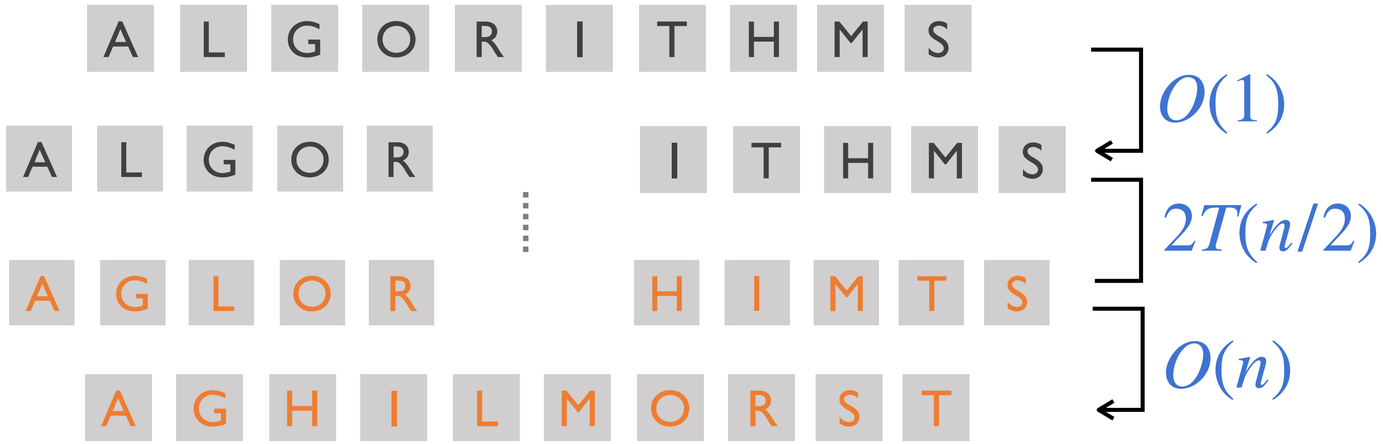
```
Merge(a[1,…,n], m):
 i ← 1; j ← m + 1
 for k ← 1 to n
  if j > n
     b[k] ← a[i]; i ← i + 1
  ……
```

1. **Correctness of Mergesort(), assuming Merge() is correct**
   - Follows by induction on n.
2. **Correctness of Merge()**
   - Loop invariant: $b[k]$ is the smallest of $a[i, \ldots, m]$ and $a[j, \ldots, n]$. Read CLRS.

Resource analysis of MergeSort

- **Running time** $T(n)$

$T(n) = 2T(n/2) + O(n) = O(n \log n)$ [will show. You can verify by induction.]



- **Space (memory)** $S(n)$
  - $S(n) = O(n)$. $a[\,\cdot\,], b[\,\cdot\,]$

Exercise. Can you merge in place, without temporary array?

Template for a complete algorithm

Problem description

- Input: $a[1,\ldots,n]$ ...
- Output : ...

Algorithm description

- Idea: divide, sort, merge ...
- Pseudocode:

```
MergeSort(a[1,...,n]):
  if n > 1 .....
```

Algorithm analysis

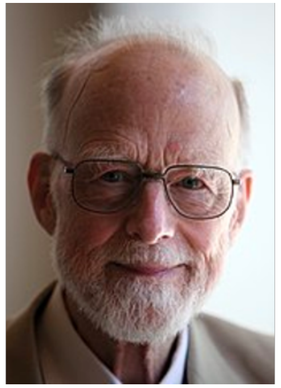
- correctness: ...
- Running time: ...
- Other analysis when needed: space ...

Quicksort

- **Main idea**
  - Divide array into two halves.
  - Recursively sort each half.
  - Merge two halves to make a sorted whole.

with condition $L \leq pivot \leq R$

trivial



Tony Hoare, 1959

- **Demo**

Analysis of Quicksort

- **Correctness**

- **Running time**
  - Best-case partition: $T(n) = 2T(n/2) + O(n)$

Cost in divide, not merge

- **A lot more into quicksort, we'll come back to it.**

Name your familiar sorting algorithms

Algorithms	$T(n)$
Insertion	$O(n^2)$
Bubble	$O(n^2)$
Merge	$O(n \log n)$
Quick (Randomized)	$O(n \log n)$
Non-comparison algorithms (Counting radix, bucket, ...)	

Exercise. Can you improve on $O(n \log n)$ for comparison-based sorting?

Divide-&-Conquer

You can see a pattern ...

1. **Divide**
   - Divide the given instance of the problem into several independent smaller instances of the same problem.
2. **Delegate**
   - Solve smaller instances recursively, i.e., delegate each smaller instance to the Recursion Fairy.
3. **Combine**
   - Combine solutions of smaller instance into the final solution for the given instance.

Multiplication

- **Input**: n-bit integers a, b (in binary)
- **Output** : $c = a \cdot b$

13=(1101)_2,14=(1110)_2

13×14=(1101)_2×(1110)_2=(10110110)_2=182

subscript 2 means binary rep.

- **Recall: the grade-school algorithm**
  - Compute n intermediate products.
  - Do n additions
  - Running time: $\Theta(n^2)$
- **Can we do better?**

```
          1 1 0 1
×         1 1 1 0
-----------------------
          0 0 0 0
        1 1 0 1 0
      1 1 0 1 0 0
+   1 1 0 1 0 0 0
-----------------------
  1 0 1 1 0 1 1 0
```

Multiplication by divide-&-conquer

- **Attempt #1**
  - Write $a = a_1 2^{n/2} + a_0, b = b_1 2^{n/2} + b_0$.
  - Observe $ab = a_1 b_1 2^n + (a_1 b_0 + a_0 b_1) 2^{n/2} + a_0 b_0$.
  - OK! Multiply $n/2$-bit integers recursively.
- **Running time**
  - Exercise. Work out the recurrence relation.
    $T(n) = 4T(n/2) + \Theta(n)$.
  - Alas! This is still $\Theta(n^2)$.

Karatsuba's idea

- From 4 to 3

$$a = a_1 2^{n/2} + a_0, b = b_1 2^{n/2} + b_0.$$
$$ab = a_1 b_1 2^n + (a_1 b_0 + a_0 b_1) 2^{n/2} + a_0 b_0.$$
$$= x 2^n + (z - x - y) 2^{n/2} + y.$$

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

z x y

- Running time
  - $T(n) = 3T(n/2) + \Theta(n) = O(n^{1.59})$.
  - Significantly faster than n^2 when n is big

Karatsuba's fast multiplication algorithm

- **Input**: n-bit integers a, b (in binary)
- **Output** : $c = a \cdot b$

FastMultipy(a, b, n): // Assume n is a power of 2 for simplicity
 if $n = 1$
  Return $x \cdot y$
 else
  $a_1 \leftarrow a/2^{n/2}, b_1 \leftarrow b/2^{n/2}, a_0 \leftarrow a \bmod 2^{n/2}, b_0 \leftarrow \bmod b 2^{n/2}$
  $x \leftarrow$ FastMultiply$(a_1, b_1, n/2)$
  $y \leftarrow$ FastMultiply$(a_0, b_0, n/2)$
  $z \leftarrow$ FastMultiply$(a_1 + a_0, b_1 + b_0, n/2)$
  Return $x^{2n} + (z - x - y)2^{n/2} + y$

… faster multiplication

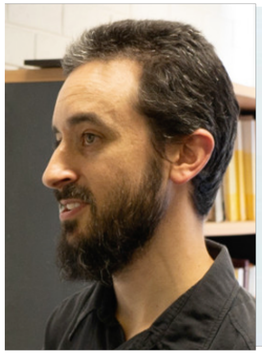
Anatolii Karatsuba **1960**

$O(n^{1.585})$

Arnold Schönhage, Volker Strassen **1971**

$O(n \log n \log\log n)$

David Harvey,Joris van der Hoeven **2019**



$O(n \log n)$

$O(n^{1+1/\log k})$ k-fold Karatsuba

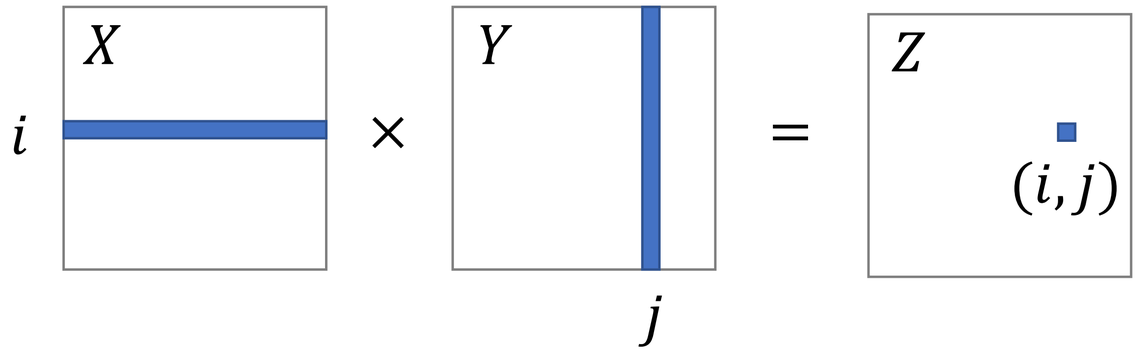
Andrei Toom, Stephen Cook **1966**

$O(n \log ne^{\log^* n})$

Martin Fürer **2007**

Matrix multiplication

- **Input**: square matrices $X = [x_{ij}]$, $Y = [y_{ij}]$, $i, j = 1, \ldots, n$.
- **Output** : $Z = [z_{ij}] = XY$. $z_{i,j} = \sum_{k=1}^{n} x_{ik} y_{kj}$



- **Standard algorithm**
  - Running time: $\Theta(n^3)$.

```
MatrixMult(X, Y, Z, n):
  for i ← 1 to n
    for j ← 1 to n
      z_ij ← 0
      for k ← 1 to n  z_ij = Σ_k x_ik y_kj
```

Matrix multiplication by divide-&-conquer

- **Idea:** block-wise multiplication
  - $n \times n$ matrix $= 2 \times 2$ matrix of $(n/2) \times (n/2)$ sub-matrices.

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix} \qquad \Rightarrow Z = XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

- **Running time**
  - 8 recursive multiplications of $(n/2) \times (n/2)$ submatrices.
  - 4 additions of $(n/2) \times (n/2)$ submatrices.

$$T(n) = 8T(n/2) + O(n^2) = O(n^3)$$

submatrices

Submatrix size

Adding submatrices

Strassen's algorithm $Z = XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$

- From 8 to 7

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

$$P_1 = A(F - H) \quad P_2 = (A + B)H$$
$$P_3 = (C + D)E \quad P_4 = D(G - E)$$
$$P_5 = (A + D)(E + H)$$
$$P_6 = (B - D)(G + H)$$
$$P_7 = (A - C)(E + F)$$

- Running time
  - 7 recursive multiplications of $(n/2) \times (n/2)$ submatrices.
  - 18 additions of $(n/2) \times (n/2)$ submatrices.
  - Significant improvement in "big"-data setting

$$T(n) = 7T(n/2) + O(n^2) \approx O(n^{2.81})$$

... faster matrix multiplication

Volker Strassen **1969**

$O(n^{2.81})$

Virginia Vassilevska Williams **2011**

$O(n^{2.3729})$

Josh Alman, VVM **2021**

$O(n^{2.3728596})$

$O(n^{2.376})$

Don Coppersmith, Shmuel Winograd, **1990**

$O(n^{2.3728639})$

François Le Gall **2014**

Scratch