Divide and Conquer*

Nengjun Zhu

School of Computer Engineering and Science
Shanghai University, P.R.China

Algorithm Course @ Shanghai University

* Special thanks is given to Prof. Xiaofeng Gao sharing her teaching materials.

Outline

Outline

Divide-and-Conquer Strategy

The divide-and-conquer strategy solves a problem P by:

(1) Breaking P into smaller subproblems of the same type.

(2) Recursively solving these subproblems.

(3) Appropriately combining their answers.

Divide-and-Conquer Strategy

The divide-and-conquer strategy solves a problem P by:

(1) Breaking P into smaller subproblems of the same type.
(2) Recursively solving these subproblems.
(3) Appropriately combining their answers.

The key works lay in three different places:

(1) How to partition problem into subproblems.
(2) At the very tail end of the recursion, how to solve the smallest subproblems outright.
(3) How to glue together the partial answers.

Outline

Johann C.F. Gauss

Johann Carl Friedrich Gauss

1777 - 1855

Johann C.F. Gauss

Johann Carl Friedrich Gauss

1777 - 1855

$$1 + 2 + \cdots + 100 = \frac{100 \cdot (1 + 100)}{2} = 5050.$$

Multiplication for Complex Numbers

Gauss once noticed that although the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd.$$

Multiplication for Complex Numbers

Gauss once noticed that although the product of two complex numbers

$$(a + bi)(c + di) = ac - bd + (bc + ad)i$$

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and $(a + b)(c + d)$, since

$$bc + ad = (a + b)(c + d) - ac - bd.$$

In our big-O way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity. However, this modest improvement becomes *very significant when applied recursively*.

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma: $\forall n \in \mathbb{N}, \exists\, n'$ *with* $n \leq n' \leq 2n$ *such that* n' *is a power of* 2.

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma: $\forall n \in \mathbb{N}, \exists\, n'$ *with* $n \leq n' \leq 2n$ *such that* n' *is a power of* 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma: $\forall n \in \mathbb{N}, \exists\, n'$ *with* $n \leq n' \leq 2n$ *such that* n' *is a power of* 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$
\begin{aligned}
x &= \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R \\
y &= \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R.
\end{aligned}
$$

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma: $\forall n \in \mathbb{N}, \exists\, n'$ *with* $n \leq n' \leq 2n$ *such that* n' *is a power of* 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$\begin{aligned} x &= \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R \\ y &= \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R. \end{aligned}$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma: $\forall n \in \mathbb{N}, \exists\, n'$ *with* $n \le n' \le 2n$ *such that* n' *is a power of* 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$\begin{aligned} x &= \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R \\ y &= \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R. \end{aligned}$$

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

The additions take linear time, as do the multiplications by powers of 2 (merely left-shifts).

Multiplication for Integers

Suppose x and y are two n-bit integers, and assume for convenience that n is *a power of* 2.

Lemma: $\forall n \in \mathbb{N}, \exists\, n'$ *with* $n \le n' \le 2n$ *such that* n' *is a power of* 2.

As a first step toward multiplying x and y, we split each of them into their **left and right halves**, which are $n/2$ bits long:

$$\begin{aligned} x &= \boxed{\quad x_L \quad}\boxed{\quad x_R \quad} = 2^{n/2} x_L + x_R \\ y &= \boxed{\quad y_L \quad}\boxed{\quad y_R \quad} = 2^{n/2} y_L + y_R. \end{aligned}$$

$$xy = (2^{n/2} x_L + x_R)(2^{n/2} y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

The additions take linear time, as do the multiplications by powers of 2 (merely left-shifts). The significant operations are the four $n/2$**-bit multiplications**; these we can handle by *four recursive calls*.

Multiplication for Integers

Our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers, and then evaluates the preceding expression in $O(n)$ time.

Multiplication for Integers

Our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers, and then evaluates the preceding expression in $O(n)$ time.

Writing $T(n)$ for the overall running time on n-bit inputs, we get **the recurrence relation**:

$$T(n) = 4T(n/2) + O(n)$$

Multiplication for Integers

Our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers, and then evaluates the preceding expression in $O(n)$ time.

Writing $T(n)$ for the overall running time on n-bit inputs, we get **the recurrence relation**:

$$T(n) = 4T(n/2) + O(n)$$

Optimization: By **Gauss**'s trick, three multiplications, $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, suffice, as

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

A Divide-and-Conquer Algorithm for Integer Multiplication

Algorithm 1: MULTIPLY(x,y)

Input: Positive integers x and y, in binary.
Output: Their product xy.

```
n = max(size of x, size of y) rounded as a power of 2 ;
if n = 1 then
    return xy;
x_L, x_R = leftmost n/2, rightmost n/2 bits of x;
y_L, y_R = leftmost n/2, rightmost n/2 bits of y;
P_1 = MULTIPLY(x_L, y_L);
P_2 = MULTIPLY(x_R, y_R);
P_3 = MULTIPLY(x_L + x_R, y_L + y_R);
return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^(n/2) + P_2
```

$n = \max(\text{size of } x, \text{size of } y)$ rounded as a power of 2 ;
if $n = 1$ **then**
  return xy;
$x_L, x_R =$ leftmost $n/2$, rightmost $n/2$ bits of x;
$y_L, y_R =$ leftmost $n/2$, rightmost $n/2$ bits of y;
$P_1 = \text{MULTIPLY}(x_L, y_L)$;
$P_2 = \text{MULTIPLY}(x_R, y_R)$;
$P_3 = \text{MULTIPLY}(x_L + x_R, y_L + y_R)$;
return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$

A Divide-and-Conquer Algorithm for Integer Multiplication

Algorithm 1: MULTIPLY(x,y)

Input: Positive integers x and y, in binary.
Output: Their product xy.

```
n = max(size of x, size of y) rounded as a power of 2 ;
if n = 1 then
    return xy;
x_L, x_R = leftmost n/2, rightmost n/2 bits of x;
y_L, y_R = leftmost n/2, rightmost n/2 bits of y;
P_1 = MULTIPLY(x_L, y_L);
P_2 = MULTIPLY(x_R, y_R);
P_3 = MULTIPLY(x_L + x_R, y_L + y_R);
return P_1 × 2^n + (P_3 − P_1 − P_2) × 2^(n/2) + P_2
```

New recurrence relation: $T(n) = 3T(n/2) + O(n)$ $\rightarrow$ How well?

Outline

Master Theorem

Master Theorem

If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

for some constants $a > 0$, $b > 1$, and $d \geq 0$,

Master Theorem

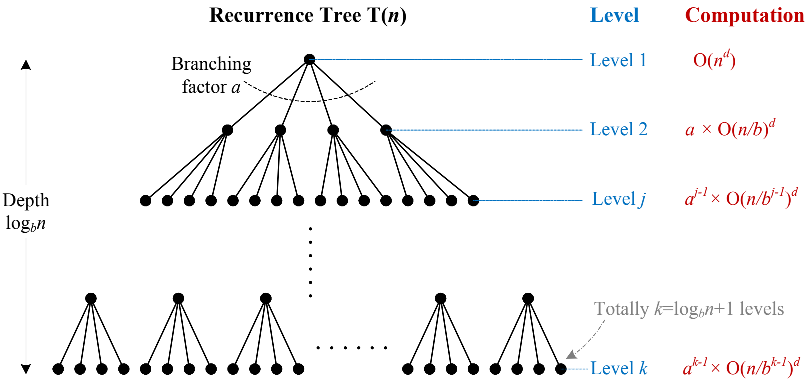
If

$$T(n) = aT(\lceil n/b \rceil) + O(n^d)$$

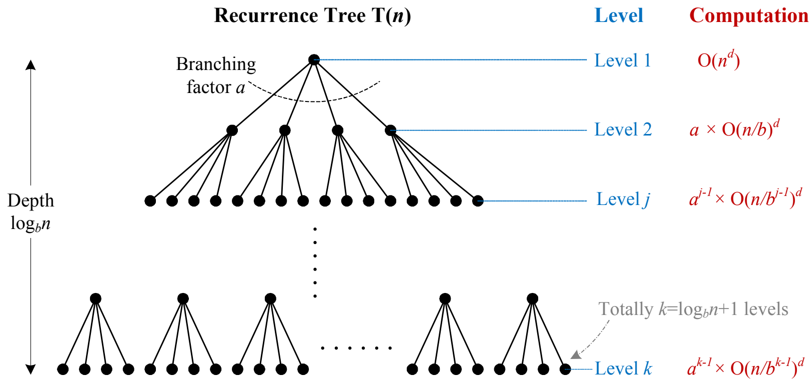
for some constants $a > 0$, $b > 1$, and $d \geq 0$, then

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

Proof of Master Theorem: $T(n) = aT(\lceil n/b \rceil) + O(n^d)$



Proof of Master Theorem: $T(n) = aT(\lceil n/b \rceil) + O(n^d)$



Complexity of $T(n)$ = Sum up all computations at each level.

Proof of Master Theorem

Proof of Master Theorem

Assume that n is a power of b. This will not influence the final bound in any important way: n is at most a multiplicative factor of b away from some power of b.

Proof of Master Theorem

Assume that n is a power of b. This will not influence the final bound in any important way: n is at most a multiplicative factor of b away from some power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and reaches the base case when

$$\frac{n}{b^{k-1}} = 1 \Rightarrow k = \log_b n + 1$$

(k is the level of the recursion tree, which equals to tree height + 1.)

Proof of Master Theorem

Assume that n is a power of b. This will not influence the final bound in any important way: n is at most a multiplicative factor of b away from some power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and reaches the base case when

$$\frac{n}{b^{k-1}} = 1 \Rightarrow k = \log_b n + 1$$

(k is the level of the recursion tree, which equals to tree height + 1.)

The branching factor of the recursion tree is a, so the j-th level of the tree is made up of a^{j-1} subproblems, each of size n/b^{j-1}.

Proof of Master Theorem

Assume that n is a power of b. This will not influence the final bound in any important way: n is at most a multiplicative factor of b away from some power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and reaches the base case when

$$\frac{n}{b^{k-1}} = 1 \Rightarrow k = \log_b n + 1$$

(k is the level of the recursion tree, which equals to tree height + 1.)

The branching factor of the recursion tree is a, so the j-th level of the tree is made up of a^{j-1} subproblems, each of size n/b^{j-1}.

The total work done at the j-th level is

$$a^{j-1} \times O\left(\frac{n}{b^{j-1}}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^{j-1}.$$

Proof of Master Theorem

Proof of Master Theorem

The total work done is

$$\sum_{j=1}^{\log_b n+1}\left(a^{j-1}\times O\left(\frac{n}{b^{j-1}}\right)^d\right)=\sum_{j=0}^{\log_b n}\left(O(n^d)\times\left(\frac{a}{b^d}\right)^j\right)=O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j.$$

Proof of Master Theorem

The total work done is

$$\sum_{j=1}^{\log_b n+1}\left(a^{j-1}\times O\left(\frac{n}{b^{j-1}}\right)^d\right)=\sum_{j=0}^{\log_b n}\left(O(n^d)\times\left(\frac{a}{b^d}\right)^j\right)=O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j.$$

It's the sum of a *geometric series (GS)* with ratio a/b^d.

Proof of Master Theorem

The total work done is

$$\sum_{j=1}^{\log_b n+1}\left(a^{j-1}\times O\left(\frac{n}{b^{j-1}}\right)^d\right)=\sum_{j=0}^{\log_b n}\left(O(n^d)\times\left(\frac{a}{b^d}\right)^j\right)=O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j.$$

It's the sum of a *geometric series (GS)* with ratio a/b^d.

(1) $\frac{a}{b^d}<1\Rightarrow d>\log_b a$:

$$O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j\le O(n^d)\frac{1}{1-\frac{a}{b^d}}=O(n^d).$$

(Sum of GS: $S_n=\sum_{j=1}^{n}a_1q^{j-1}=a_1\frac{1-q^n}{1-q}\le a_1\frac{1}{1-q}$ if $q<1$)

Proof of Master Theorem

(2) $\frac{a}{b^d} = 1 \Rightarrow d = \log_b a$:

$$O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j = O(n^d)(\log_b n + 1) = O(n^d \log_b n) = O(n^d \log n).$$

($\log_b n = \frac{\log n}{\log b} = \frac{1}{\log b}\log n = O(\log n)$ by changing the base)

Proof of Master Theorem

(3) $\dfrac{a}{b^d} > 1 \Rightarrow d < \log_b a$: (reverse the GS in decreasing order)

$$
\begin{aligned}
O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^j &= O(n^d)\sum_{j=0}^{\log_b n}\left(\frac{a}{b^d}\right)^{\log_b n}\cdot\left(\frac{b^d}{a}\right)^j \\
&= O(n^d)\sum_{j=0}^{\log_b n}\frac{a^{\log_b n}}{(b^{\log_b n})^d}\cdot\left(\frac{b^d}{a}\right)^j \\
&\leq O(n^d)\frac{n^{\log_b a}}{n^d}\cdot\frac{1}{1-\frac{b^d}{a}} \\
&= O(n^{\log_b a})
\end{aligned}
$$

$\left(a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}\right)$

Time Complexity of Multiplication

Original recurrence relation: $T(n) = 4T(n/2) + O(n)$

$a = 4, b = 2, d = 1, d < \log_b a.$

$\Rightarrow$ **Time complexity:** $O(n^{\log_b a}) = O(n^2).$

Time Complexity of Multiplication

Original recurrence relation: $T(n) = 4T(n/2) + O(n)$

$a = 4$, $b = 2$, $d = 1$, $d < \log_b a$.

$\Rightarrow$ **Time complexity:** $O(n^{\log_b a}) = O(n^2)$.

Optimized recurrence relation: $T(n) = 3T(n/2) + O(n)$

$a = 3$, $b = 2$, $d = 1$, $d < \log_b a$.

$\Rightarrow$ **Time complexity:** $O(n^{\log_2 3}) \approx O(n^{1.59})$.

Outline

Binary Search

Algorithm 2: BinarySearch

Input: An array $A[1..n]$ of n elements sorted in nondecreasing order and an element x.

Output: j if $x = A[j]$, $1 \le j \le n$, and 0 otherwise.

$low \leftarrow 1$; $high \leftarrow n$; $j \leftarrow 0$;
while $low \le high$ **and** $j = 0$ **do**
    $mid \leftarrow \lfloor (low + high)/2 \rfloor$;
    if $x = A[mid]$ **then**
        $j \leftarrow mid$ **break**;
    else if $x < A[mid]$ **then**
        $high \leftarrow mid - 1$;
    else
        $low \leftarrow mid + 1$;
return j;

Time Complexity

To find a key x in $A[1, \cdots, n]$ in sorted order, we first compare x with $A[n/2]$, and depending on the result we recurse either on the first half $A[1, \cdots, n/2-1]$, or on the second half $A[n/2+1, \cdots, n]$.

Time Complexity

To find a key x in $A[1, \cdots, n]$ in sorted order, we first compare x with $A[n/2]$, and depending on the result we recurse either on the first half $A[1, \cdots, n/2 - 1]$, or on the second half $A[n/2 + 1, \cdots, n]$.

The recurrence function is

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1),$$

Time Complexity

To find a key x in $A[1, \cdots, n]$ in sorted order, we first compare x with $A[n/2]$, and depending on the result we recurse either on the first half $A[1, \cdots, n/2-1]$, or on the second half $A[n/2+1, \cdots, n]$.

The recurrence function is

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + O(1),$$

By Master Theorem, $a = 1$, $b = 2$, $d = 0$, and thus the running time should be $O(\log n)$.

Outline

1. Divide-and-Conquer
   - Basic Technique
   - An Introductory Example: Multiplication
   - Recurrence Relations
2. Applications
   - Binary Search
   - Merge Sort
   - Matrix Multiplication

Merging Two Sorted Lists

Algorithm 3: Merge

Input: $A[1..m]$, p, q and r with $1 \leq p \leq q < r \leq m$.
Output: $A[p..r]$ (merging two sorted subarrays $A[p..q]$, $A[q+1..r]$).

$s \leftarrow p$; $t \leftarrow q+1$; $k \leftarrow p$;
while $s \leq q$ **and** $t \leq r$ **do**
    if $A[s] \leq A[t]$ **then**
        $B[k] \leftarrow A[s]$; $s \leftarrow s+1$; ($B[p..r]$ is an auxiliary array)
    else $B[k] \leftarrow A[t]$; $t \leftarrow t+1$;
    $k \leftarrow k+1$;
if $s = q+1$ **then**
    $B[k..r] \leftarrow A[t..r]$;
else $B[k..r] \leftarrow A[s..q]$;
return $A[p..r] \leftarrow B[p..r]$;

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m, n\}$;

E.g.

2	3	6

and

7	11	13	45	57

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m, n\}$;

  E.g. $\boxed{2}\boxed{3}\boxed{6}$ and $\boxed{7}\boxed{11}\boxed{13}\boxed{45}\boxed{57}$

- at most $m + n - 1$.

  E.g. $\boxed{2}\boxed{3}\boxed{66}$ and $\boxed{7}\boxed{11}\boxed{13}\boxed{45}\boxed{57}$

Analysis of Merge

Suppose $A[p..q]$ has m elements and $A[q+1..r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m, n\}$;

  E.g. $\boxed{2 \mid 3 \mid 6}$ and $\boxed{7 \mid 11 \mid 13 \mid 45 \mid 57}$

- at most $m + n - 1$.

  E.g. $\boxed{2 \mid 3 \mid 66}$ and $\boxed{7 \mid 11 \mid 13 \mid 45 \mid 57}$

If the two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons is between $\lfloor n/2 \rfloor$ and $n - 1$.

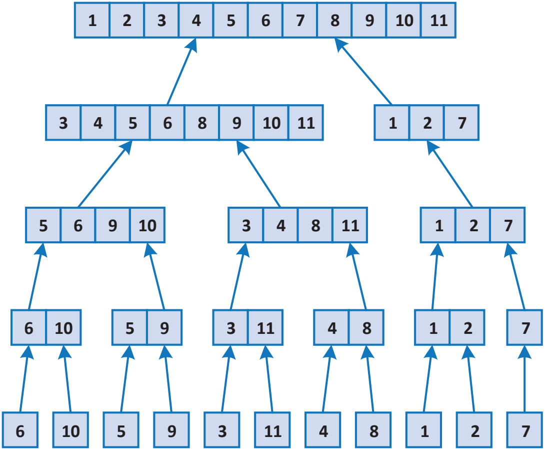
Bottom-Up MergeSort Algorithm

Algorithm 4: MergeSort

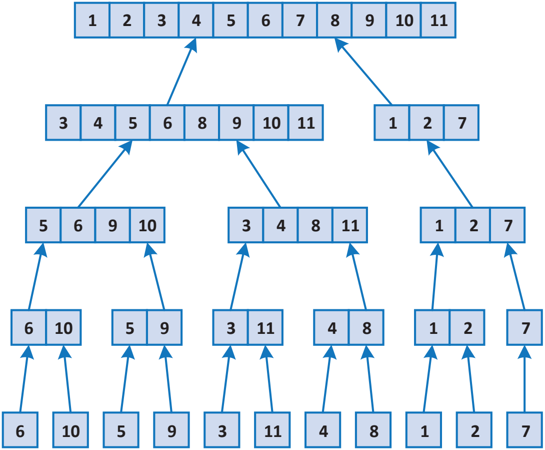
Input: An array $A[1..n]$ of n elements.
Output: $A[1..n]$ sorted in nondecreasing order.

$t \leftarrow 1$;
while $t < n$ **do**
    $s \leftarrow t$; $t \leftarrow 2s$; $i \leftarrow 0$;
    while $i + t \leq n$ **do**
        $Merge(A, i+1, i+s, i+t)$;
        $i \leftarrow i + t$;
    if $i + s < n$ **then**
        $Merge(A, i+1, i+s, n)$;
return $A[1..n]$;

An Example


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

An Example



Time Complexity:

Recurrence:

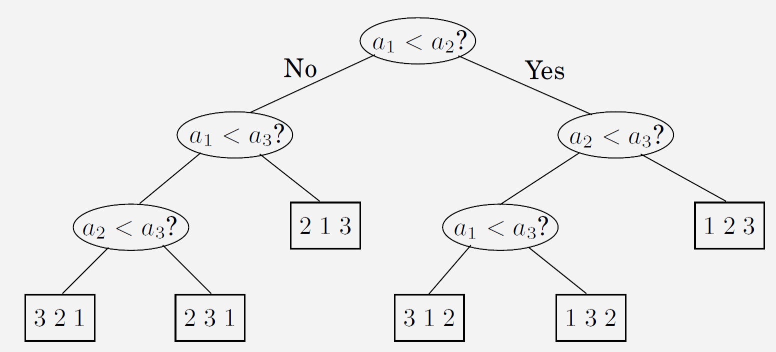
$$T(n) = 2T(n/2) + O(n);$$

By Master Theorem

$$T(n) = O(n \log n).$$

An $n \log n$ Lower Bound for Sorting

An example sorting permutation tree for $\{a_1, a_2, a_3\}$:



An $n \log n$ Lower Bound for Sorting

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a *permutation* of $\{1, 2, \ldots, n\}$.

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a *permutation* of $\{1, 2, \ldots, n\}$.

every permutation must appear as the label of a leaf.

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a *permutation* of $\{1, 2, \ldots, n\}$.

every permutation must appear as the label of a leaf.

This is a binary tree with $n!$ leaves.

An $n \log n$ Lower Bound for Sorting

Sorting algorithms can be depicted as **trees**.

The **depth** of the tree – the number of comparisons on the longest path from root to leaf, is exactly the worst-case time complexity of the algorithm.

Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a *permutation* of $\{1, 2, \ldots, n\}$.

every permutation must appear as the label of a leaf.

This is a binary tree with $n!$ leaves. Thus, the depth of our tree – and the complexity of our algorithm – must be at least

$$\log(n!) \approx \log\left(\sqrt{\pi\,(2n + 1/3)} \cdot n^n \cdot e^{-n}\right) \overset{?}{=} \Omega(n \log n),$$

where we use Stirling's formula.

Outline

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i,j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

That is, Z_{ij} is the dot product of the ith row of X with the jth column of Y.

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i,j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

That is, Z_{ij} is the dot product of the ith row of X with the jth column of Y.

In general, XY is not the same as YX; *matrix multiplication is not commutative*.

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry

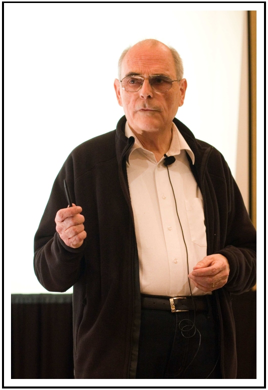
$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}$$

That is, Z_{ij} is the dot product of the ith row of X with the jth column of Y.

In general, XY is not the same as YX; *matrix multiplication is not commutative*.

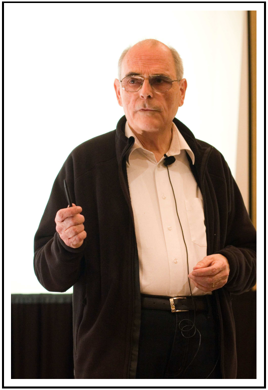
The preceding formula implies an $O(n^3)$ algorithm for matrix multiplication.

Volker Strassen



Volker Strassen (1936 –)

Volker Strassen



Volker Strassen (1936 –)

In 1969, the German mathematician **Volker Strassen** announced a surprising $O(n^{2.81})$ algorithm.

Divide and conquer

Divide and conquer

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

Divide and conquer

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

Then

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

Divide and conquer

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

Then

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

To compute the size-n product XY, recursively compute eight size-$n/2$ products AE, BG, AF, BH, CE, DG, CF, DH and then do some $O(n^2)$-time addition.

Divide and conquer

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

Then

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

To compute the size-n product XY, recursively compute eight size-$n/2$ products AE, BG, AF, BH, CE, DG, CF, DH and then do some $O(n^2)$-time addition.

The recurrence is

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

with solution $O(n^3)$.

Strassen’s trick

Strassen's trick

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

where

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

Strassen's trick

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

where

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

The recurrence is

$$T(n) = 7T(n/2) + O(n^2)$$

with solution $O(n^{\log_2 7}) \approx O(n^{2.81})$.