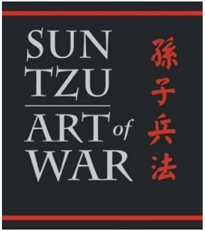
Algorithm Design and Analysis

Day 3
Divide and Conquer

2015, AUT - CJLU

Day 3: Divide and Conquer

Part I: Divide-and-Conquer – Integer Multiplication

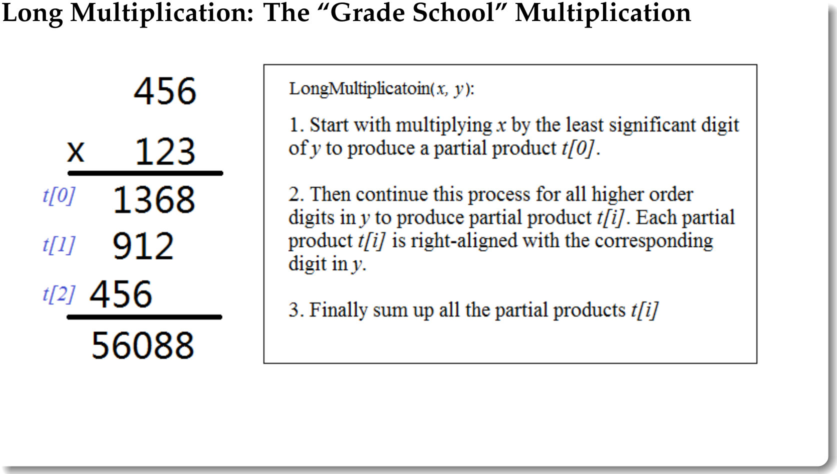


故用兵之法，十則圍之，五則攻之，倍則分之， 敵則能戰之，少則能守之，不若則能避之。

``It is the rule in war, if ten times the enemy's strength, surround them; if five times, attack them; if double, be able to divide them; if equal,engage them; if fewer, be able to evade them; if weaker, be able to avoid them.''

---``Chapter III Strategic Attack" 500BC

Multiplication



Long Multiplication: The "Grade School" Multiplication

```
          456
x         123
-------------
t[0]     1368
t[1]      912
t[2]  456
-------------
        56088
```

LongMultiplicatoin(x, y):

1. Start with multiplying x by the least significant digit of y to produce a partial product *t[0]*.

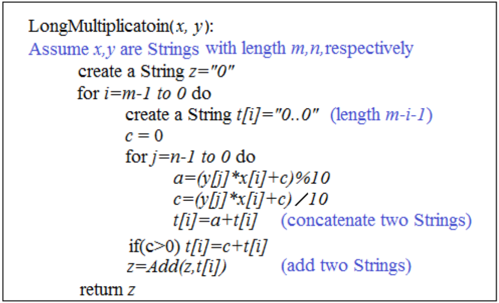
2. Then continue this process for all higher order digits in y to produce partial product *t[i]*. Each partial product *t[i]* is right-aligned with the corresponding digit in y.

3. Finally sum up all the partial products *t[i]*

Multiplication

Long Multiplication: The “Grade School” Multiplication

$$
\begin{array}{rr}
 & 456 \\
\times & 123 \\
\hline
t[0] & 1368 \\
t[1] & 9120 \\
t[2] & 45600 \\
\hline
 & 56088
\end{array}
$$

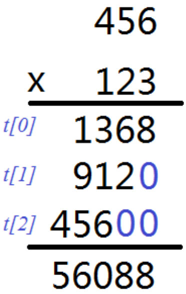


```
LongMultiplicatoin(x, y):
Assume x,y are Strings with length m,n,respectively
    create a String z="0"
    for i=m-1 to 0 do
        create a String t[i]="0..0"  (length m-i-1)
        c = 0
        for j=n-1 to 0 do
            a=(y[j]*x[i]+c)%10
            c=(y[j]*x[i]+c)/10
            t[i]=a+t[i]    (concatenate two Strings)
        if(c>0) t[i]=c+t[i]
        z=Add(z,t[i])      (add two Strings)
    return z
```

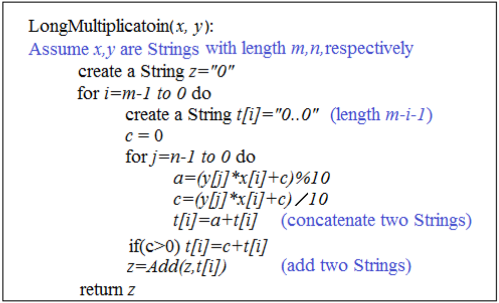
Complexity: What is the running time?

Multiplication

Long Multiplication: The "Grade School" Multiplication



```
         456
   x     123
   ---------
t[0]    1368
t[1]    9120
t[2]   45600
   ---------
       56088
```

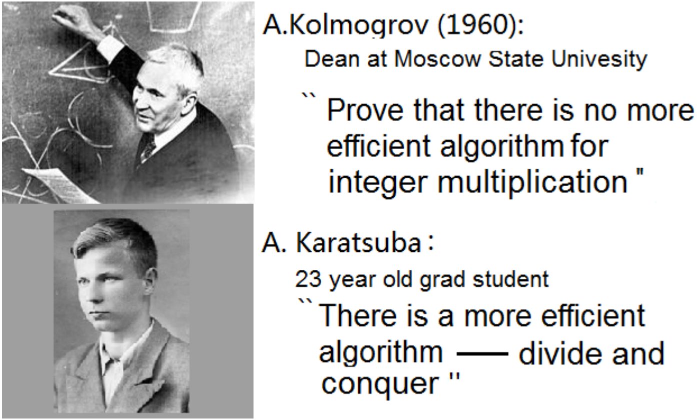


```
LongMultiplicatoin(x, y):
Assume x,y are Strings with length m,n,respectively
    create a String z="0"
    for i=m-1 to 0 do
        create a String t[i]="0..0"  (length m-i-1)
        c = 0
        for j=n-1 to 0 do
            a=(y[j]*x[i]+c)%10
            c=(y[j]*x[i]+c)/10
            t[i]=a+t[i]    (concatenate two Strings)
        if(c>0) t[i]=c+t[i]
        z=Add(z,t[i])      (add two Strings)
    return z
```

Complexity: What is the running time? $O(n^2)$

Multiplication

Kolmogorov's Conjecture



A.Kolmogrov (1960):

Dean at Moscow State Univesity

`` Prove that there is no more efficient algorithm for integer multiplication "

A. Karatsuba :

23 year old grad student

`` There is a more efficient algorithm —— divide and conquer "

Faster Multiplication

Observation

- $(2x + 3)(5x + 7) = 2 \times 5x^2 + (2 \times 7 + 3 \times 5)x + 3 \times 7$

Faster Multiplication

Observation

- $(2x+3)(5x+7) = 2\times 5x^2 + (2\times 7 + 3\times 5)x + 3\times 7$

  4 multiplications, 3 additions

Faster Multiplication

Observation

- $(2x+3)(5x+7) = 2\times 5x^2 + (2\times 7 + 3\times 5)x + 3\times 7$

  4 multiplications, 3 additions
- $(2+3)(5+7) = 2\times 5 + 3\times 7 + 2\times 7 + 3\times 5$

Faster Multiplication

Observation

- $(2x+3)(5x+7) = 2\times 5x^2 + (2\times 7 + 3\times 5)x + 3\times 7$

  4 multiplications, 3 additions

- $(2+3)(5+7) = 2\times 5 + 3\times 7 + 2\times 7 + 3\times 5$
  $\Rightarrow 2\times 7 + 3\times 5 = (2+3)\times(5+7) - 2\times 5 - 3\times 7$

Faster Multiplication

Observation

- $(2x+3)(5x+7) = 2\times 5x^2 + (2\times 7 + 3\times 5)x + 3\times 7$

  4 multiplications, 3 additions

- $(2+3)(5+7) = 2\times 5 + 3\times 7 + 2\times 7 + 3\times 5$
  $\Rightarrow 2\times 7 + 3\times 5 = (2+3)\times(5+7) - 2\times 5 - 3\times 7$
  $\Rightarrow (2x+3)(5x+7) =$
  $2\times 5x^2 + 3\times 7 + ((2+3)\times(5+7) - 2\times 5 - 3\times 7)x$

Faster Multiplication

Observation

- $(2x+3)(5x+7) = 2\times 5x^2 + (2\times 7 + 3\times 5)x + 3\times 7$

  4 multiplications, 3 additions

- $(2+3)(5+7) = 2\times 5 + 3\times 7 + 2\times 7 + 3\times 5$
  $\Rightarrow 2\times 7 + 3\times 5 = (2+3)\times(5+7) - 2\times 5 - 3\times 7$
  $\Rightarrow (2x+3)(5x+7) =$
  $2\times 5x^2 + 3\times 7 + ((2+3)\times(5+7) - 2\times 5 - 3\times 7)x$

  3 multiplications, 6 additions

Faster Multiplication

Observation

- $(2x+3)(5x+7) = 2\times 5x^2 + (2\times 7 + 3\times 5)x + 3\times 7$

  4 multiplications, 3 additions

- $(2+3)(5+7) = 2\times 5 + 3\times 7 + 2\times 7 + 3\times 5$
  $\Rightarrow 2\times 7 + 3\times 5 = (2+3)\times(5+7) - 2\times 5 - 3\times 7$
  $\Rightarrow (2x+3)(5x+7) =$
  $2\times 5x^2 + 3\times 7 + ((2+3)\times(5+7) - 2\times 5 - 3\times 7)x$

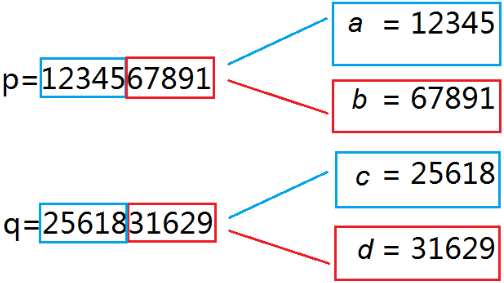
  3 multiplications, 6 additions

General Version

$$(ax+b)(cx+d) = acx^2 + bd + ((a+b)(c+d) - ac - bd)x$$

Karatsuba's Algorithm

Example:



$p = 1234567891$

$a = 12345$

$b = 67891$

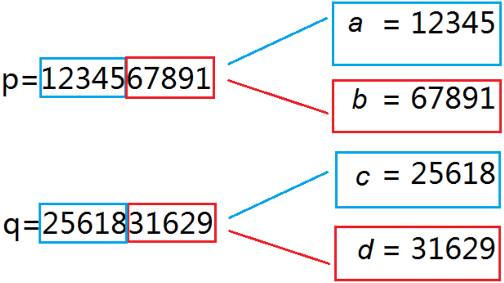
$q = 2561831629$

$c = 25618$

$d = 31629$

Karatsuba's Algorithm

Example:



p=1234567891

a = 12345

b = 67891

q=2561831629

c = 25618

d = 31629

$$\begin{aligned} p \times q &= (a \times 10^5 + b) \times (c \times 10^5 + d) \\ &= ac \times 10^{10} + bd + ((a+b)(c+d) - ac - bd) \times 10^5 \end{aligned}$$

Karatsuba’s Algorithm

```
Algorithm multiply($x, y$)
INPUT: $x, y$ are length-$n$ (string representations of) integers
OUTPUT: The (string representation of) product of $x, y$
    if $n = 1$ return $xy$
    $a = leftmost\ \lceil n/2 \rceil$ bits of $x$, $b = rightmost\ \lfloor n/2 \rfloor$ bits of $x$
    $c = leftmost\ \lceil n/2 \rceil$ bits of $y$, $d = rightmost\ \lfloor n/2 \rfloor$ bits of $y$
    $p_1 \leftarrow multiply(a, c)$
    $p_2 \leftarrow multiply(b, d)$
    $p_3 \leftarrow multiply(a + b, c + d)$
    $p_3 \leftarrow p_3 - p_1 - p_2$
    Shift $p_1$ to the left by $n$-bits, $p_3$ to the left by $\lfloor n/2 \rfloor$-bits
    return $p_1 + p_2 + p_3$
```

Karatsuba's Algorithm

Time Complexity

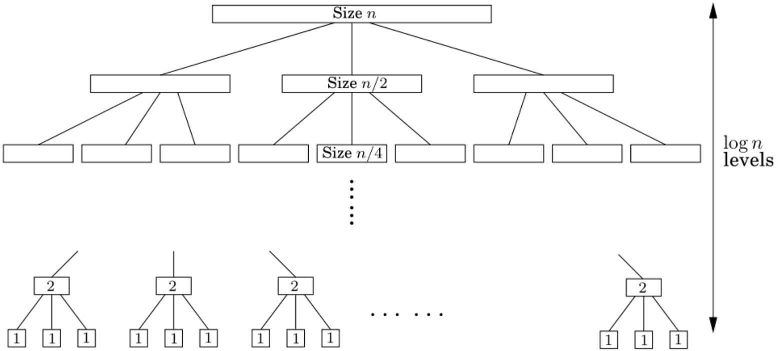
Let $T(n)$ be the time complexity of multiplying two length-n integers.

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

where c is a constant.

Divide-and-Conquer

Tree of Recursive Calls

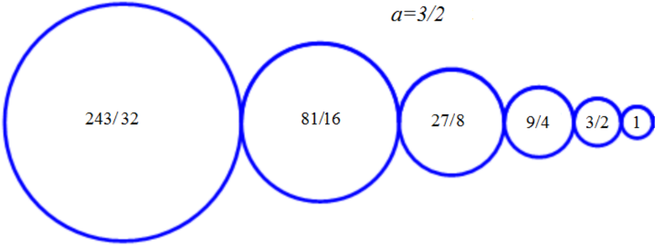


Solving Recurrence

Time Complexity

$$T(n) = nc(1 + \frac{3}{2} + \frac{3^2}{2^2} + \ldots + \frac{3^{k-1}}{2^{k-1}} + \frac{3^k}{2^k})$$

Geometric Series



Solving Recurrence

Time Complexity

$$T(n) = nc(1 + \frac{3}{2} + \frac{3^2}{2^2} + \ldots + \frac{3^{k-1}}{2^{k-1}} + \frac{3^k}{2^k})$$

Geometric Series

Let $m = 1 + a + a^2 + a^3 + \ldots + a^k$, where $a > 0, a \neq 1$, be a geometric series.

Solving Recurrence

Time Complexity

$$T(n) = nc(1 + \frac{3}{2} + \frac{3^2}{2^2} + \ldots + \frac{3^{k-1}}{2^{k-1}} + \frac{3^k}{2^k})$$

Geometric Series

Let $m = 1 + a + a^2 + a^3 + \ldots + a^k$, where $a > 0, a \neq 1$, be a geometric series.
$\Rightarrow am = a + a^2 + a^3 + \ldots + a^{k+1}$
$\Rightarrow am - m = a^{k+1} - 1$
$\Rightarrow m(a-1) = a^{k+1} - 1$
$\Rightarrow m = \frac{a^{k+1}-1}{a-1}$

Solving Recurrence

Time Complexity

$$T(n) = nc(1 + \frac{3}{2} + \frac{3^2}{2^2} + \ldots + \frac{3^{k-1}}{2^{k-1}} + \frac{3^k}{2^k})$$

Geometric Series

Let $m = 1 + a + a^2 + a^3 + \ldots + a^k$, where $a > 0, a \neq 1$, be a geometric series.

$\Rightarrow am = a + a^2 + a^3 + \ldots + a^{k+1}$

$\Rightarrow am - m = a^{k+1} - 1$

$\Rightarrow m(a-1) = a^{k+1} - 1$

$\Rightarrow m = \frac{a^{k+1}-1}{a-1}$

Therefore

$$T(n) = nc\frac{(3/2)^{k+1} - 1}{3/2 - 1} \leq 3nc\frac{3^k}{2^k}$$

Solving Recurrence

Time Complexity

Recall $n = 2^k$. Therefore $k = \log_2 n$. We have

Solving Recurrence

Time Complexity

Recall $n = 2^k$. Therefore $k = \log_2 n$. We have

$$T(n) \leq 3nc\frac{3^{\log_2 n}}{2^{\log_2 n}} = 3nc\frac{3^{\log_2 n}}{n} = 3c \times 3^{\log_2 n}$$

Solving Recurrence

Time Complexity

Recall $n = 2^k$. Therefore $k = \log_2 n$. We have

$$T(n) \leq 3nc\frac{3^{\log_2 n}}{2^{\log_2 n}} = 3nc\frac{3^{\log_2 n}}{n} = 3c \times 3^{\log_2 n}$$

Note that $3 = 2^{\log_2 3}$. So

$$\begin{aligned} T(n) &\leq 3c \times (2^{\log_2 3})^{\log_2 n} \\ &= 3c \times 2^{\log_2 3 \times \log_2 n} \\ &= 3c \times 2^{\log_2 n \times \log_2 3} \\ &= 3c \times (2^{\log_2 n})^{\log_2 3} \\ &= 3c \times n^{\log_2 3} \\ &\leq 3c \times n^{1.59} \end{aligned}$$

Solving Recurrence

Time Complexity

Recall $n = 2^k$. Therefore $k = \log_2 n$. We have

$$T(n) \leq 3nc\frac{3^{\log_2 n}}{2^{\log_2 n}} = 3nc\frac{3^{\log_2 n}}{n} = 3c \times 3^{\log_2 n}$$

Note that $3 = 2^{\log_2 3}$. So

$$\begin{aligned} T(n) &\leq 3c \times (2^{\log_2 3})^{\log_2 n} \\ &= 3c \times 2^{\log_2 3 \times \log_2 n} \\ &= 3c \times 2^{\log_2 n \times \log_2 3} \\ &= 3c \times (2^{\log_2 n})^{\log_2 3} \\ &= 3c \times n^{\log_2 3} \\ &\leq 3c \times n^{1.59} \end{aligned}$$

Therefore $T(n)$ is $O(n^{1.59})$, a big improvement from $O(n^2)$!!

Divide-and-Conquer

- Karatsuba's algorithm solves integer multiplication in time $O(n^{1.59})$.
- The technique used is called divide-and-conquer

Divide-and-Conquer

- Karatsuba's algorithm solves integer multiplication in time $O(n^{1.59})$.
- The technique used is called divide-and-conquer

Divide-and-Conquer as an algorithm design technique

The divide-and-conquer technique solves a computational problem by dividing it into one or more subprograms of smaller size, conquering each of them by solving them recursively, and then combining their solutions into a solution for the original problem.

Divide-and-Conquer

General Divide-and-Conquer Strategy

```
if $n \leq n_0$ then
    directly solve problem without dividing
else
    divide problem into $a$ subproblems of size $n/b$ each
    for $i \leftarrow 0$ to $a-1$ do
        recursively solve the $i$th subproblem
    combine the $a$ solutions into a solution of the original
    problem
```

Divide-and-Conquer

General Divide-and-Conquer Strategy

```
if $n \le n_0$ then
    directly solve problem without dividing
else
    divide problem into $a$ subproblems of size $n/b$ each
    for $i \leftarrow 0$ to $a - 1$ do
        recursively solve the $i$th subproblem
    combine the $a$ solutions into a solution of the original problem
```

Running Time Analysis

Come up with a recurrence: $T(n) = aT(n/b) + f(n)$

Example 1: Karatsuba's Algorithm

Karatsuba's algorithm

Given two input numbers x, y:

```
if $x, y$ both have length 1 then
    directly multiply $x, y$
else
    divide each $x$ and $y$ into two numbers and
    obtain 3 subproblems of size $n/2$ each
    for each subprogram do
        recursively solve the $i$th subproblem
    add the 3 solutions
```

Time complexity: $T(n) = 3T(n/2) + cn$.

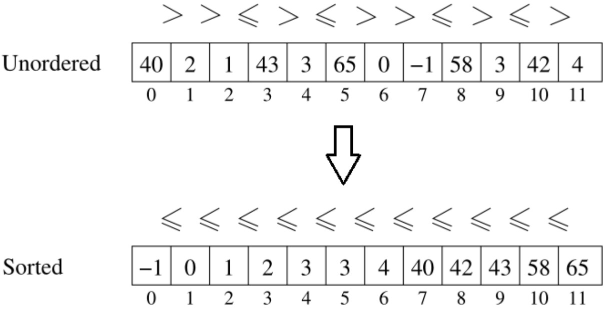
Day 3: Divide and Conquer

Part II: Sorting

The Sorting Problem

The Sorting Problem

Arrange an array of integers so that every adjacent pair of values is in the correct order.



Unordered

$> \quad > \quad \leqslant \quad > \quad \leqslant \quad > \quad > \quad \leqslant \quad > \quad \leqslant \quad >$

40	2	1	43	3	65	0	−1	58	3	42	4
0	1	2	3	4	5	6	7	8	9	10	11

⇩

Sorted

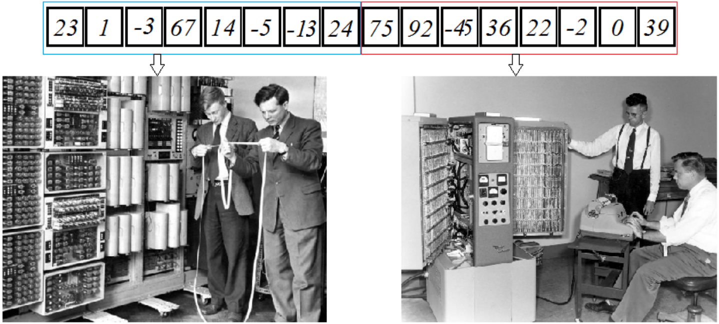
$\leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant \quad \leqslant$

−1	0	1	2	3	3	4	40	42	43	58	65
0	1	2	3	4	5	6	7	8	9	10	11

Merge Sort

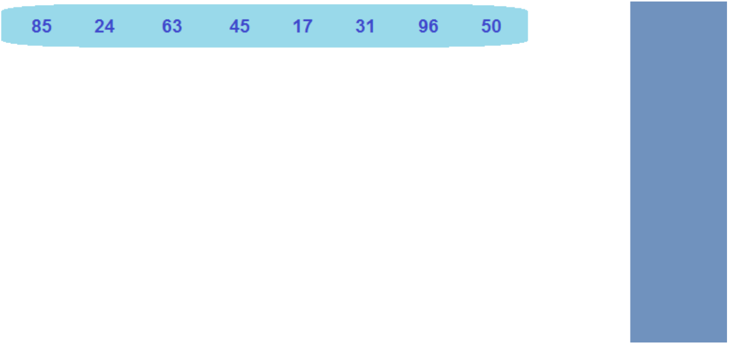
Merge Sort Algorithm

1. To sort an array, partition it into two parts; and give each part to a different machine
2. Each machine sorts its part recursively
3. Merge the two solutions



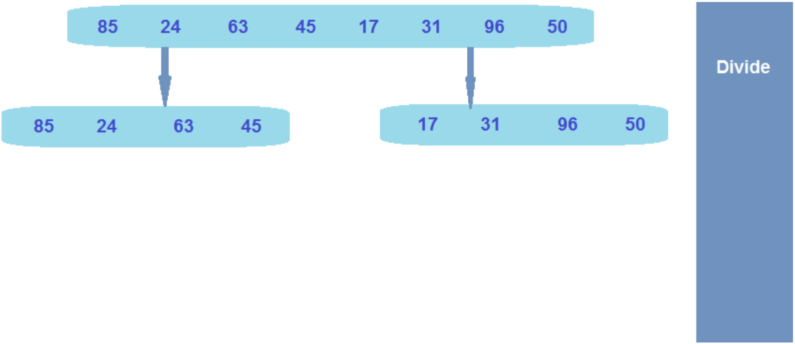
23	1	-3	67	14	-5	-13	24	75	92	-45	36	22	-2	0	39

Merge Sort



85 24 63 45 17 31 96 50

Merge Sort



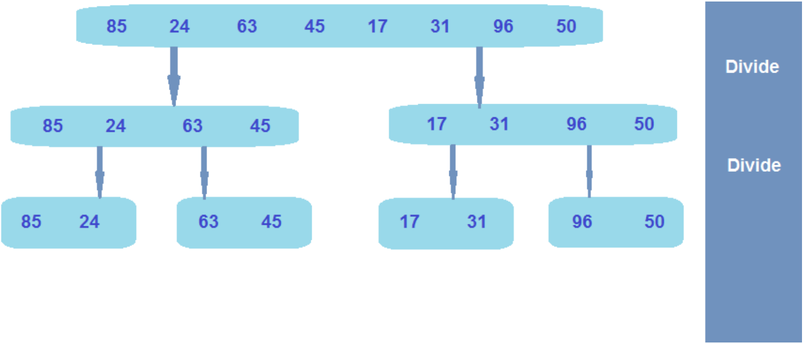
85 24 63 45 17 31 96 50

85 24 63 45

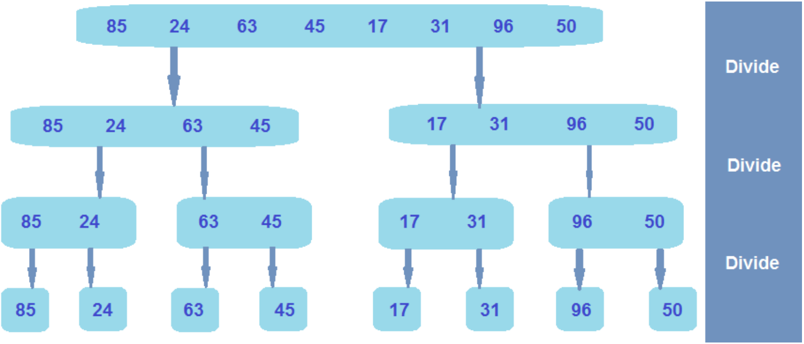
17 31 96 50

Divide

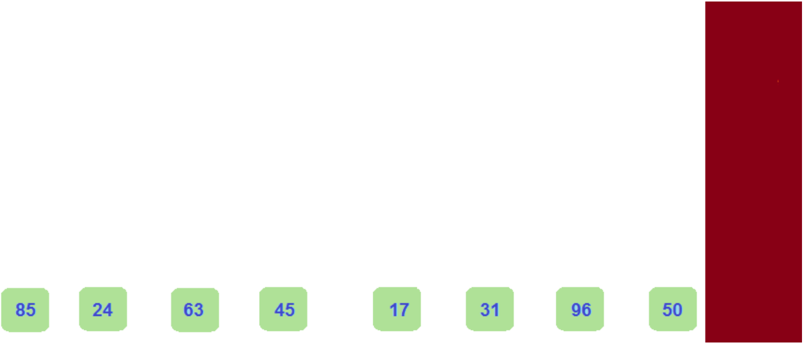
Merge Sort


85 24 63 45 17 31 96 50
85 24 63 45
17 31 96 50
85 24
63 45
17 31
96 50
Divide
Divide

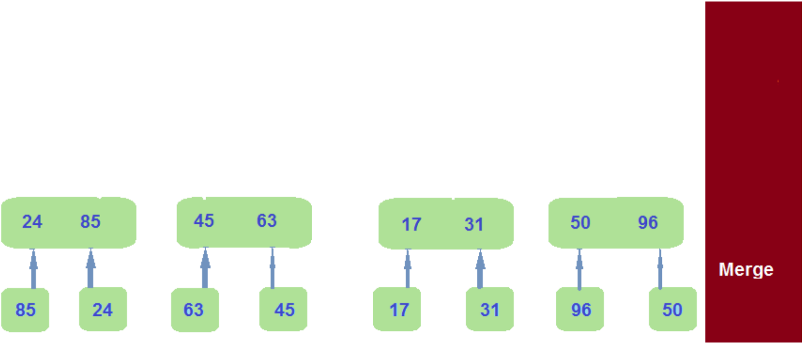
Merge Sort


85 24 63 45 17 31 96 50
85 24 63 45
17 31 96 50
85 24
63 45
17 31
96 50
85
24
63
45
17
31
96
50
Divide
Divide
Divide

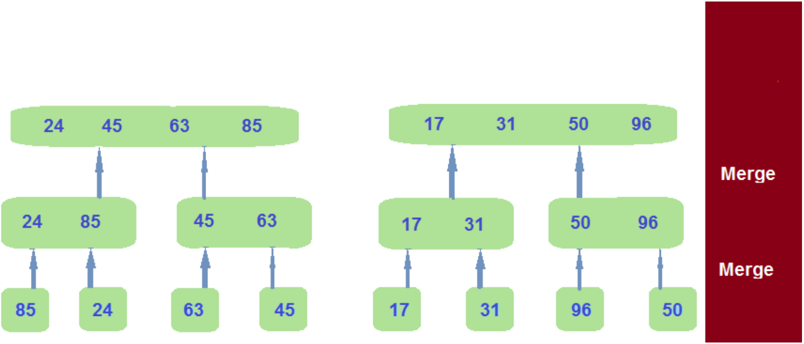
Merge Sort



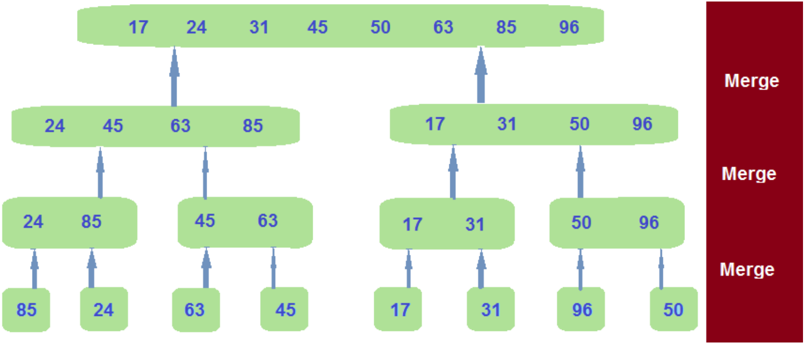
Merge Sort



Merge Sort


24 45 63 85
17 31 50 96
24 85
45 63
17 31
50 96
85
24
63
45
17
31
96
50
Merge
Merge

Merge Sort


17 24 31 45 50 63 85 96
24 45 63 85
17 31 50 96
24 85
45 63
17 31
50 96
85
24
63
45
17
31
96
50
Merge
Merge
Merge

Merge Sort

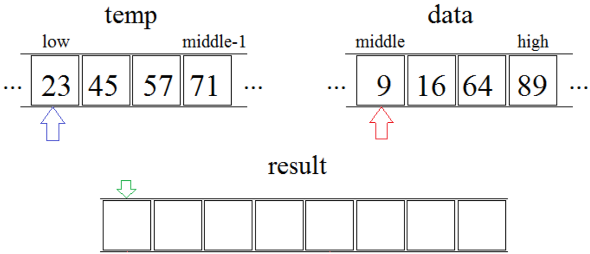
The Partition Procedures

```
private static void mergeSortRecursive(int data[], int temp[],
                                        int low, int high)
// pre: 0 <= low <= high < data.length
// post: values in data[low..high] are in ascending order
{
    int n = high-low+1;
    int middle = low + n/2;
    int i;
    if (n < 2) return;
    // move lower half of data into temporary storage
    for (i = low; i < middle; i++)
        temp[i] = data[i];
    // sort lower half of array
    mergeSortRecursive(temp,data,low,middle-1);
    // sort upper half of array
    mergeSortRecursive(data,temp,middle,high);
    // merge halves together
    merge(data,temp,low,middle,high);
}
```

The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending

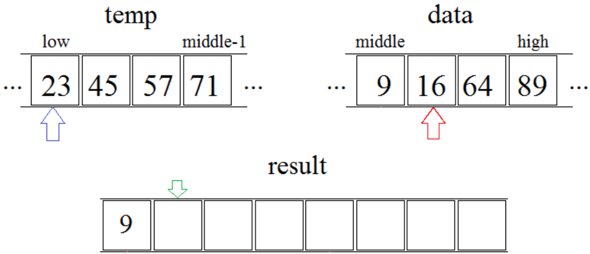


Merge Sort

The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending



temp

low … middle-1

… 23 45 57 71 …

data

middle … high

… 9 16 64 89 …

result

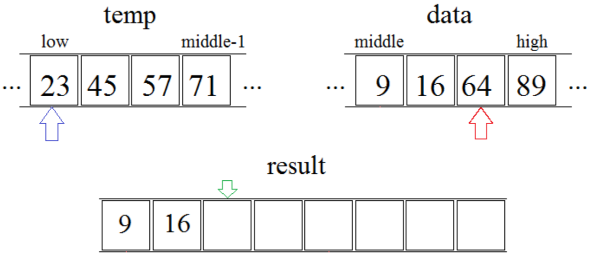
9

Merge Sort

The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending

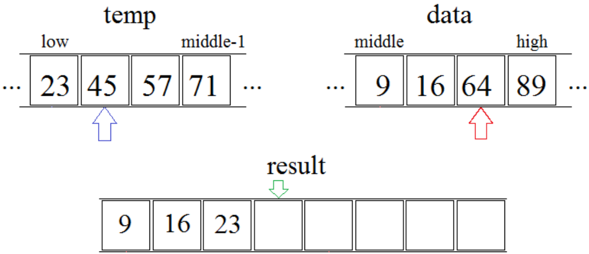


Merge Sort

The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending

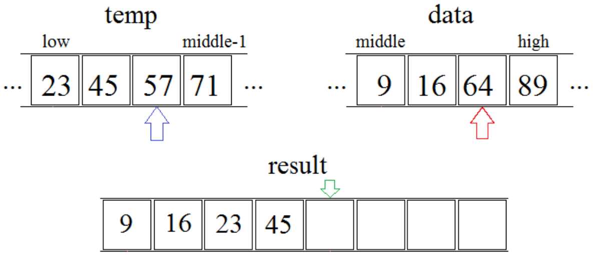


Merge Sort

The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending

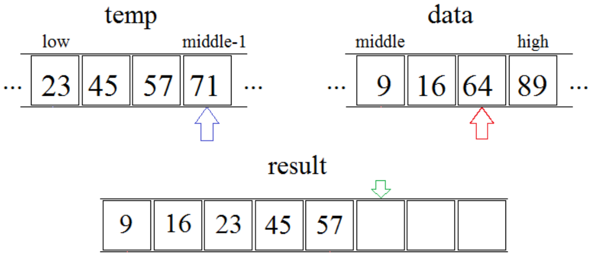


Merge Sort

The Merge Procedures

Suppose

- temp[low..middle − 1] are ascending
- data[middle..high] are ascending

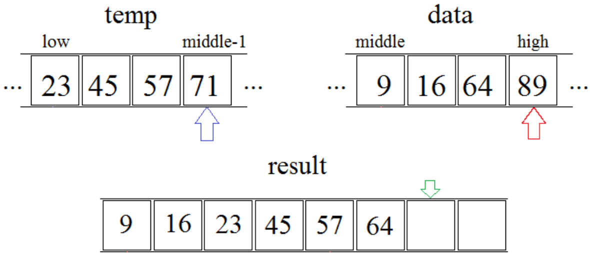


Merge Sort

The Merge Procedures

Suppose

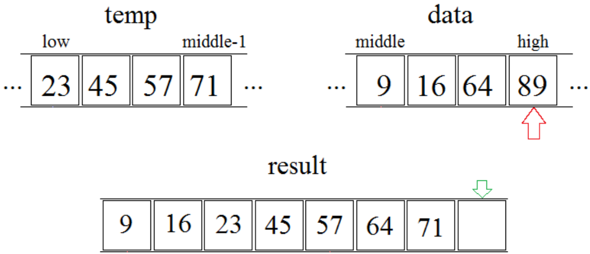
- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending



The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending

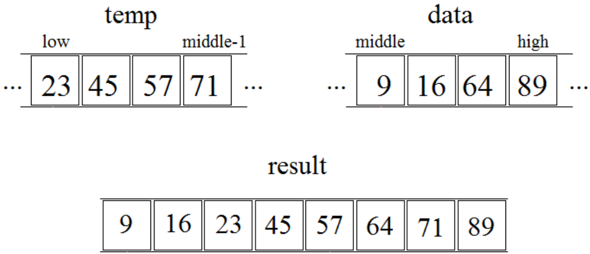


Merge Sort

The Merge Procedures

Suppose

- `temp[low..middle − 1]` are ascending
- `data[middle..high]` are ascending



temp

low … middle-1

… 23 | 45 | 57 | 71 …

data

middle … high

… 9 | 16 | 64 | 89 …

result

9 | 16 | 23 | 45 | 57 | 64 | 71 | 89

Merge Sort

The Merge Procedures

```
private static void merge(int data[], int temp[],
                          int low, int middle, int high){
    int ri=0; //result index
    int ti=low;//temp index
    int di=middle;//data index
    int[] result = new int[high-low+1];
    while (ti<middle && di<=high){
        if (data[di]<temp[ti]){
            result[ri++] = data[di++];//smaller is in data
        } else{
            result[ri++] = temp[ti++];//smaller is in temp
        }
    }     while(ti<middle) result[ri++]=temp[ti++];
    while(di<=high) result[ri++]=data[di++];
    for(int i=0;i<high;i++) data[low+i]=result[i];
}
```

Merge Sort

```
public static void mergeSort(int data[], int n)
//pre:  0<=n <=data.length
//post:  values in data[0..n-1] are in ascending order
{
    mergeSortRecursive(data, new int[n], 0, n-1);
}
```

Merge Sort: Complexity

- The Partitioning Stage
  - The number of partitioning is $n - 1$.
- The Merging Stage
  - The number of operations is $O(n \log n)$.

The overall complexity of the algorithm: $O(n \log n)$

Merge Sort: Complexity

Optimality

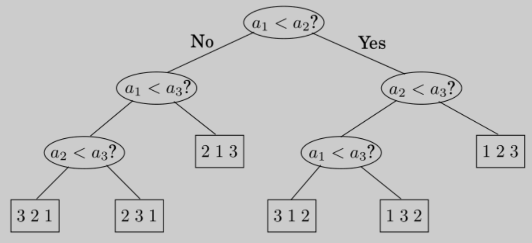
Instead of asking "Is merge sort the most efficient algorithm for sorting?",
we ask "What is the shortest time must any sorting algorithms run in?"

Comparison-Based Sorting

Merge sort is a comparison-based sorting algorithm, i.e., it sorts numbers based on a sequence of comparisons between elements from the input arrays.

Sort: Time Lower Bound

Comparison Tree



- The depth or height of the tree is the number of comparisons on the longest path from the root to a leaf.
- For $n > 0$, let $h(n)$ be the height of the comparison tree for n numbers.

Sorting: Time Lower Bound

Fact

Any comparison-based sorting algorithm uses at least $h(n)$ comparisons in the worst case.

Why? Otherwise there must be some sequence on which the algorithm fails.

A Lower Bound on Times of Comparison

What is $h(n)$ for any $n > 0$?

Sorting: Time Lower Bound

Observation 1

- The comparison tree for n numbers is a binary tree
- A binary tree with k leaves has at least height $\log k$

$\Rightarrow h(n) \geq \log k$ where k is the number of leaves.

Sorting: Time Lower Bound

Observation 1

- The comparison tree for n numbers is a binary tree
- A binary tree with k leaves has at least height $\log k$

$\Rightarrow h(n) \geq \log k$ where k is the number of leaves.

Observation 2

- Every leaf in the comparison tree is a permutation of n numbers
- There are $n!$ number of permutations with n numbers

$\Rightarrow k = n!$

Sorting: Time Lower Bound

Observation 1

- The comparison tree for n numbers is a binary tree
- A binary tree with k leaves has at least height $\log k$

$\Rightarrow h(n) \geq \log k$ where k is the number of leaves.

Observation 2

- Every leaf in the comparison tree is a permutation of n numbers
- There are $n!$ number of permutations with n numbers

$\Rightarrow k = n!$

$\Rightarrow h(n) \geq \log n!$

Sorting: Time Lower Bound

Observation 3

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \ldots \times n-1 \times n \\ &> \lfloor n/2 \rfloor \times \ldots \times n-1 \times n \\ &> (n/2)^{n/2} \end{aligned}$$

$\Rightarrow \log n! > \log(n/2)^{n/2} = n/2(\log n - 1)$
$\Rightarrow h(n)$ is $\Omega(n \log n)$

Sorting: Time Lower Bound

Observation 3

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \ldots \times n-1 \times n \\ &> \lfloor n/2 \rfloor \times \ldots \times n-1 \times n \\ &> (n/2)^{n/2} \end{aligned}$$

$\Rightarrow \log n! > \log(n/2)^{n/2} = n/2(\log n - 1)$
$\Rightarrow h(n)$ is $\Omega(n \log n)$

Conclusion

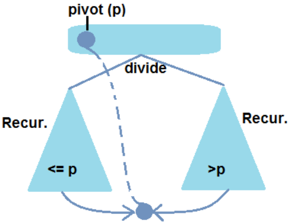
Any comparison-based sorting algorithm must use $\Omega(n \log n)$ number of comparisons in the worst case.
$\Rightarrow$ The best time complexity for any comparison-based sorting algorithm is $\Theta(n \log n)$.
$\Rightarrow$ Merge sort is optimal.

Quick Sort

Just like merge sort, quick sort is also a divide-and-conquer algorithm.

Idea

1. Divide: If S has 0 or 1 element, do nothing. Otherwise, pick a pivot/partitionElement from the array (first element). Rearrange the other elements into two parts, **those $\leq$ the pivot** and **those $>$ the pivot**.
2. Recur: Recursively sort these two parts
3. Conquer: Put the two resulting sorted arrays and the pivot in order.



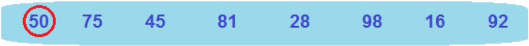
Quick Sort

Example:

50 75 45 81 28 98 16 92

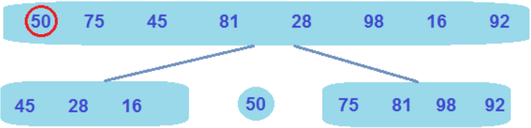
Quick Sort

Example:



Quick Sort

Example:



50 75 45 81 28 98 16 92

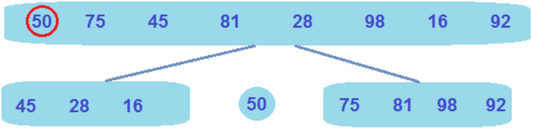
45 28 16

50

75 81 98 92

Quick Sort

Example:



50 75 45 81 28 98 16 92

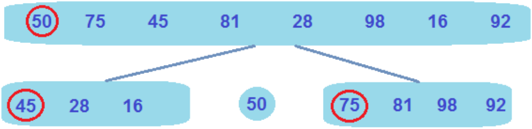
45 28 16

50

75 81 98 92

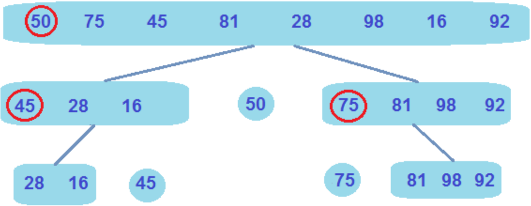
Quick Sort

Example:



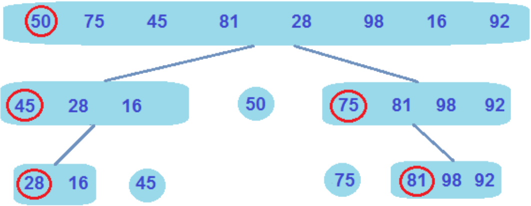
Quick Sort

Example:



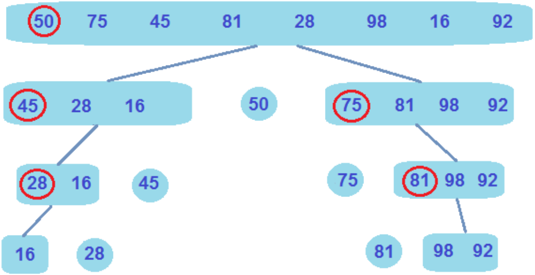
Quick Sort

Example:



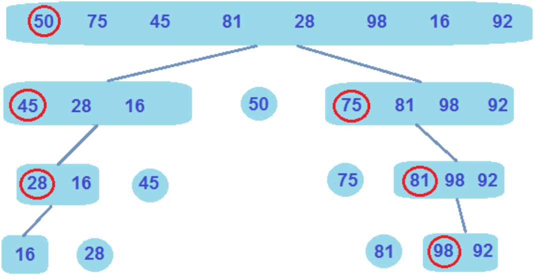
Quick Sort

Example:



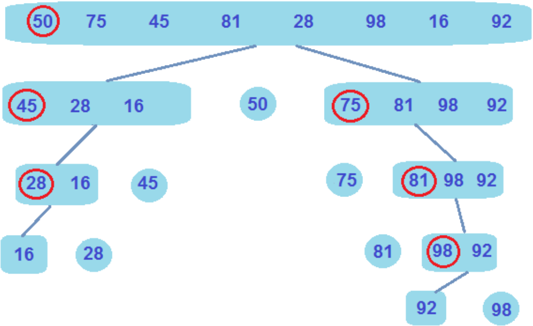
Quick Sort

Example:



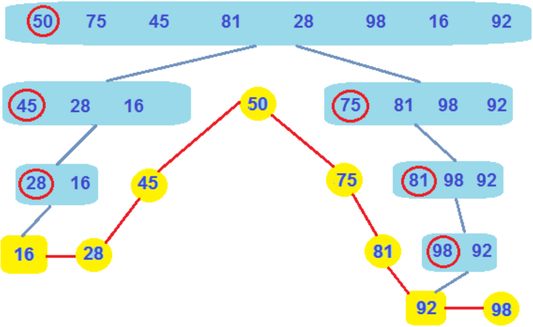
Quick Sort

Example:



Quick Sort

Example:



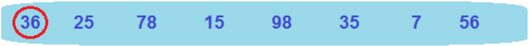
Quick Sort

Example 2:

36	25	78	15	98	35	7	56

Quick Sort

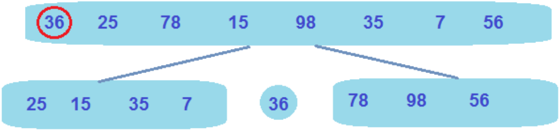
Example 2:



36 25 78 15 98 35 7 56

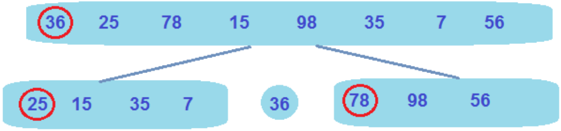
Quick Sort

Example 2:



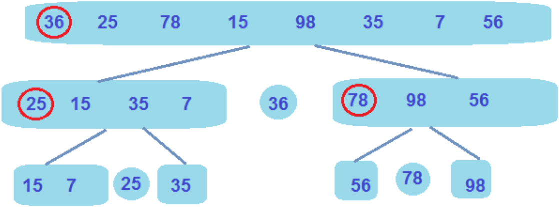
Quick Sort

Example 2:



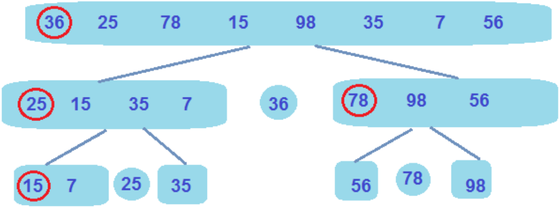
Quick Sort

Example 2:



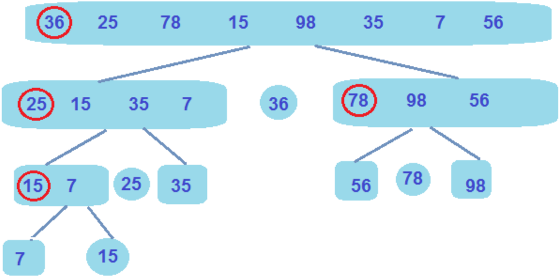
Quick Sort

Example 2:



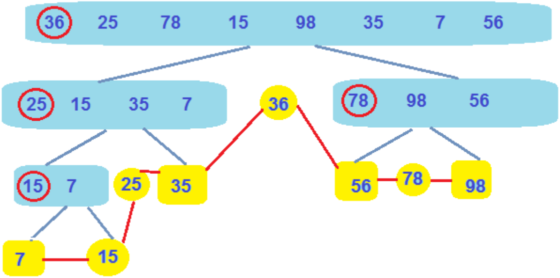
Quick Sort

Example 2:



Quick Sort

Example 2:



In-Place Quick Sort

- The most straight-forward implementation of quick sort requires creating two new arrays at each recursion step (otherwise we may need a lot of shiftings)

  ⇒ Takes too much memory.

- We would like to work only on the input array (without creating new arrays), just like selection, insertion and bubble sort.

In-Place Quick Sort

- The most straight-forward implementation of quick sort requires creating two new arrays at each recursion step (otherwise we may need a lot of shiftings)

  ⇒ Takes too much memory.

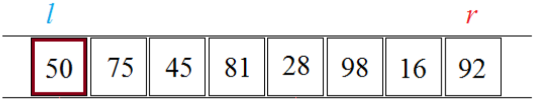
- We would like to work only on the input array (without creating new arrays), just like selection, insertion and bubble sort.
- In-place quick sort is a way of implementing quick sort so that it only works on the input array.

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `left`



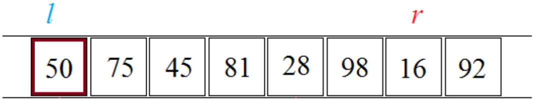
l							r
50	75	45	81	28	98	16	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `left`

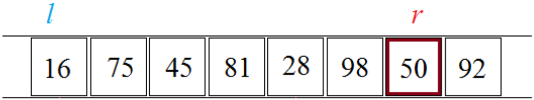


In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements $\leq$ pivot to its left; all elements $>$ pivot to its right

Pivot on `right`



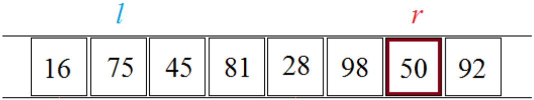
l						r	
16	75	45	81	28	98	50	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `right`



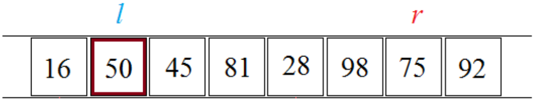
	l					r	
16	75	45	81	28	98	50	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements $\leq$ pivot to its left; all elements $>$ pivot to its right

Pivot on `left`

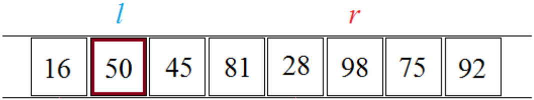


In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `left`



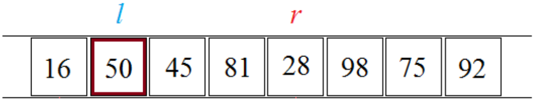
	l				r		
16	50	45	81	28	98	75	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements $\leq$ pivot to its left; all elements $>$ pivot to its right

Pivot on `left`



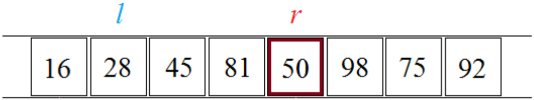
	l			r			
16	50	45	81	28	98	75	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements $\leq$ pivot to its left; all elements $>$ pivot to its right

Pivot on `right`



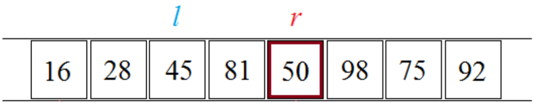
	l			r			
16	28	45	81	50	98	75	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `right`

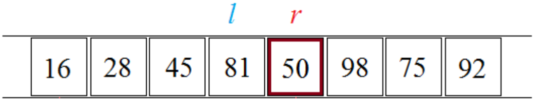


In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `right`

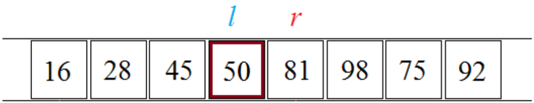


In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `left`



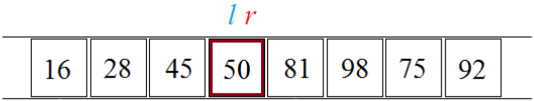
			l	*r*			
16	28	45	50	81	98	75	92

In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Pivot on `left` and `right`

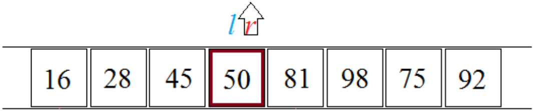


In-Place Quick Sort

The Partition Procedures

- Given the `data[]` array, and `left`, `right` pointers.
- Set `data[left]` as the pivot
- We want to arrange all elements ≤ pivot to its left; all elements > pivot to its right

Return `right`



16	28	45	50	81	98	75	92

In-Place Quick Sort

The Partition Procedures

```
private static int partition(int data[], int left, int
right)
//pre:  left<= right
//post:  data[left] placed in the correct location
{
    while(true){
        //move right "pointer" towards left
        while(left<right && data[left] <data[right])
            right--;
        if (left<right) swap(data,left++,right);
        //move left pointer towards right
        while(left<right && data[left]<data[right])
            left++;
        if(left<right) swap(data,left,right--);
        else return right;
    }
}
```

In-Place Quick Sort

The Combine Procedures

```
private static void quickSortRecursive(int data[], int
left, int right)
//pre:  left<=right
//post:  data[left..right] in ascending order
{
    int pivot;
    if (left>=right) return;
    pivot=partition(data,left,right); //Partition
    quickSortRecursive(data,left,pivot-1); //Sort small
    quickSortRecursive(data,pivot+1,right); //Sort large
}
```

```
public static void quickSort(int data[], int n){
    quickSortRecursive(data,0,n-1);
}
```

Quick Sort: Complexity

Worst Case

- If the input array is already sorted, one side of the pivot is always empty, and the other side is $n - 1$
- There are n levels of recursion.
- Therefore $O(n^2)$.

Quick Sort: Complexity

Worst Case

- If the input array is already sorted, one side of the pivot is always empty, and the other side is $n - 1$
- There are n levels of recursion.
- Therefore $O(n^2)$.

In the average case, quick sort runs very fast

Quick Sort: Complexity

Quick Sort: Complexity

- Fix an input *list* of size n.
- Let $T_{list}(n)$ denote the running time for sorting *list*.
- The *pivot* is chosen as the first element in *list*.

Quick Sort: Complexity

- Fix an input *list* of size n.
- Let $T_{list}(n)$ denote the running time for sorting *list*.
- The *pivot* is chosen as the first element in *list*.

A Recurrence for $T_{list}(n)$

- Let n_{small} be the number of elements smaller than *pivot*.
- Let n_{big} be the number of elements bigger than *pivot*.
- Then we have

$$T_{list}(n) = T_{list}(n_{small}) + T_{list}(n_{big}) + cn$$

Quick Sort: Complexity

Average-case analysis

Quick Sort: Complexity

Average-case analysis

- $T(n)$: the average-case running time of quick sort.

Quick Sort: Complexity

Average-case analysis

- $T(n)$: the average-case running time of quick sort.
- Then $T(n) = T(n_{small}) + T(n_{big}) + cn$.
- n_{small} may take any value in $0, 1, 2, \ldots, n-1$.
- $n_{big} = n - n_{small} - 1$.

Quick Sort: Complexity

Average-case analysis

- $T(n)$: the average-case running time of quick sort.
- Then $T(n) = T(n_{small}) + T(n_{big}) + cn$.
- n_{small} may take any value in $0, 1, 2, \ldots, n-1$.
- $n_{big} = n - n_{small} - 1$.

Equi-probability assumption

Assume that all initial orderings appear with equal probability.
$\Rightarrow$ For any $i, j \in \{0, 1, 2, \ldots, n-1\}$, $Pr[n_{small} = i] = Pr[n_{big} = j]$

Quick Sort: Complexity

Average-case analysis

- $T(n)$: the average-case running time of quick sort.
- Then $T(n) = T(n_{small}) + T(n_{big}) + cn$.
- n_{small} may take any value in $0, 1, 2, \ldots, n-1$.
- $n_{big} = n - n_{small} - 1$.

Equi-probability assumption

Assume that all initial orderings appear with equal probability.
$\Rightarrow$ For any $i, j \in \{0, 1, 2, \ldots, n-1\}$, $Pr[n_{small} = i] = Pr[n_{big} = j]$

Therefore on average,

$$T(n_{small}) = [T(0) + T(1) + \ldots + T(n-1)] \div n$$
$$T(n_{big}) = [T(n-1) + T(n-2) + \ldots + T(0)] \div n$$

Quick Sort: Complexity

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$T(n) = 2([T(0) + \cdots + T(n-1)]) \div n + cn$$

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$T(n) = 2([T(0) + \cdots + T(n-1)]) \div n + cn$$
$$nT(n) = 2([T(0) + \cdots + T(n-1)]) + cn^2$$

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$
\begin{aligned}
T(n) &= 2([T(0) + \cdots + T(n-1)]) \div n + cn \\
nT(n) &= 2([T(0) + \cdots + T(n-1)]) + cn^2 \\
(n-1)T(n-1) &= 2([T(0) + \cdots + T(n-2)]) + c(n-1)^2
\end{aligned}
$$

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$\begin{aligned} T(n) &= 2([T(0) + \cdots + T(n-1)]) \div n + cn \\ nT(n) &= 2([T(0) + \cdots + T(n-1)]) + cn^2 \\ (n-1)T(n-1) &= 2([T(0) + \cdots + T(n-2)]) + c(n-1)^2 \\ nT(n) - (n-1)T(n-1) &= 2T(n-1) - 2cn + c \end{aligned}$$

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$\begin{aligned} T(n) &= 2([T(0) + \cdots + T(n-1)]) \div n + cn \\ nT(n) &= 2([T(0) + \cdots + T(n-1)]) + cn^2 \\ (n-1)T(n-1) &= 2([T(0) + \cdots + T(n-2)]) + c(n-1)^2 \\ nT(n) - (n-1)T(n-1) &= 2T(n-1) - 2cn + c \\ nT(n) &= (n+1)T(n-1) - c(2n-1) \end{aligned}$$

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$\begin{aligned}
T(n) &= 2([T(0) + \cdots + T(n-1)]) \div n + cn \\
nT(n) &= 2([T(0) + \cdots + T(n-1)]) + cn^2 \\
(n-1)T(n-1) &= 2([T(0) + \cdots + T(n-2)]) + c(n-1)^2 \\
nT(n) - (n-1)T(n-1) &= 2T(n-1) - 2cn + c \\
nT(n) &= (n+1)T(n-1) - c(2n-1) \\
\frac{T(n)}{(n+1)} &= \frac{T(n-1)}{n} - \frac{c(2n-1)}{n(n+1)}
\end{aligned}$$

Quick Sort: Complexity

Average-case Analysis (continued.)

By the above arguments,

$$\begin{aligned}
T(n) &= 2([T(0) + \cdots + T(n-1)]) \div n + cn \\
nT(n) &= 2([T(0) + \cdots + T(n-1)]) + cn^2 \\
(n-1)T(n-1) &= 2([T(0) + \cdots + T(n-2)]) + c(n-1)^2 \\
nT(n) - (n-1)T(n-1) &= 2T(n-1) - 2cn + c \\
nT(n) &= (n+1)T(n-1) - c(2n-1) \\
\frac{T(n)}{(n+1)} &= \frac{T(n-1)}{n} - \frac{c(2n-1)}{n(n+1)} \\
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{3c}{n+1} - \frac{c}{n}
\end{aligned}$$

Quick Sort: Complexity

Average-case Analysis (continued.)

Quick Sort: Complexity

Average-case Analysis (continued.)

Telescoping on $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{3c}{n+1} - \frac{c}{n}$

Quick Sort: Complexity

Average-case Analysis (continued.)

Telescoping on $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{3c}{n+1} - \frac{c}{n}$

$$\begin{array}{llll}
\frac{T(n)}{n+1} & = \frac{T(n-1)}{n} & + \frac{3c}{n+1} & - \frac{c}{n} \\
\frac{T(n-1)}{n} & = \frac{T(n-2)}{n-1} & + \frac{3c}{n} & - \frac{c}{n-1} \\
\frac{T(n-2)}{n-1} & = \frac{T(n-3)}{n-2} & + \frac{3c}{n-1} & - \frac{c}{n-2} \\
\cdots & = \cdots & \cdots & \cdots \\
\frac{T(1)}{2} & = \frac{T(0)}{1} & + \frac{3c}{2} & - \frac{c}{1}
\end{array}$$

Quick Sort: Complexity

Average-case Analysis (continued.)

Telescoping on $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{3c}{n+1} - \frac{c}{n}$

$$\begin{array}{llll}
\frac{T(n)}{n+1} & = \frac{T(n-1)}{n} & + \frac{3c}{n+1} & - \frac{c}{n} \\
\frac{T(n-1)}{n} & = \frac{T(n-2)}{n-1} & + \frac{3c}{n} & - \frac{c}{n-1} \\
\frac{T(n-2)}{n-1} & = \frac{T(n-3)}{n-2} & + \frac{3c}{n-1} & - \frac{c}{n-2} \\
\dots & = \dots & \dots & \dots \\
\frac{T(1)}{2} & = \frac{T(0)}{1} & + \frac{3c}{2} & - \frac{c}{1}
\end{array}$$

Cancel out the common terms, we have:

$$\frac{T(n)}{n+1} = 3c(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}) - c(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + \frac{1}{1})$$

Quick Sort: Complexity

Harmonic number

The n-th harmonic number is $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$.
Fact: H_n is $O(\log n)$.

Quick Sort: Complexity

Harmonic number

The n-th harmonic number is $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$.
Fact: H_n is $O(\log n)$.

Average-case Analysis (continued.)

$$\frac{T(n)}{n+1} = 3c(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}) - c(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + \frac{1}{1})$$

Quick Sort: Complexity

Harmonic number

The n-th harmonic number is $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$.
Fact: H_n is $O(\log n)$.

Average-case Analysis (continued.)

$$\begin{aligned}\frac{T(n)}{n+1} &= 3c(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}) - c(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + \frac{1}{1}) \\ &= 3c(H_{n+1} - 1) + c(H_n)\end{aligned}$$

Quick Sort: Complexity

Harmonic number

The n-th harmonic number is $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$.
Fact: H_n is $O(\log n)$.

Average-case Analysis (continued.)

$$\begin{aligned}
\frac{T(n)}{n+1} &= 3c(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}) - c(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + \frac{1}{1}) \\
&= 3c(H_{n+1} - 1) + c(H_n) \\
&= 4cH_{n+1} - 3c - \frac{c}{n+1}
\end{aligned}$$

Quick Sort: Complexity

Harmonic number

The *n*-th harmonic number is $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$.
Fact: H_n is $O(\log n)$.

Average-case Analysis (continued.)

$$\begin{aligned}
\frac{T(n)}{n+1} &= 3c(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}) - c(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + \frac{1}{1}) \\
&= 3c(H_{n+1} - 1) + c(H_n) \\
&= 4cH_{n+1} - 3c - \frac{c}{n+1} \\
T(n) &= 4c(n+1)H_{n+1} - 3c(n+1) - c
\end{aligned}$$

Quick Sort: Complexity

Harmonic number

The n-th harmonic number is $H_n = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n}$.
Fact: H_n is $O(\log n)$.

Average-case Analysis (continued.)

$$\begin{aligned}
\frac{T(n)}{n+1} &= 3c(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{2}) - c(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} + \frac{1}{1}) \\
&= 3c(H_{n+1} - 1) + c(H_n) \\
&= 4cH_{n+1} - 3c - \frac{c}{n+1} \\
T(n) &= 4c(n+1)H_{n+1} - 3c(n+1) - c
\end{aligned}$$

Therefore $T(n)$ is $\Theta(n \log n)$.

Sorting Algorithms

Algorithms	Worst Case Time	Average Case Time
Merge Sort	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n^2)$	$O(n \log n)$

Lower bound for comparison-base sorting: $O(n \log n) \Rightarrow$

- MergeSort has optimal worst case complexity
- QuickSort has optimal average case complexity

Day 3: Divide and Conquer

Part III: Analysis of Divide and Conquer Algorithms

Runtime Analysis

Divide and Conquer

The *running time* $T(n)$ of a Divide-and-Conquer algorithm can normally be specified by

$$T(n) = aT(n/b) + f(n).$$

The problem is entirely mathematical: Solve the above recursion.

Master Theorem

What is the master theorem?

The master theorem provides a direct way to solve recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ are constants and $f(n)$ is a positive function.

Master Theorem

What is the master theorem?

The master theorem provides a direct way to solve recurrence of the form

$$T(n) = aT(n/b) + f(n)$$

where $a \geq 1$, $b > 1$ are constants and $f(n)$ is a positive function.

Examples

- $T(n) = 3T(n/2) + n^2$
- $T(n) = 16T(n/2) + 3n \log n$
- $T(n) = T(n/2) + 3$
- $T(n) = \sqrt{2}T(n/4) + n^{0.51}$

Master Theorem

Intuitive Version

The master theorem allows us to solve the recurrence

$$T(n) = aT(n/b) + f(n)$$

by comparing the function $f(n)$ with $n^{\log_b a}$.
There are three cases:

Master Theorem

Intuitive Version

The master theorem allows us to solve the recurrence

$$T(n) = aT(n/b) + f(n)$$

by comparing the function $f(n)$ with $n^{\log_b a}$.
There are three cases:

- Case 1: $f(n)$ is much smaller than $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a}$.

Master Theorem

Intuitive Version

The master theorem allows us to solve the recurrence

$$T(n) = aT(n/b) + f(n)$$

by comparing the function $f(n)$ with $n^{\log_b a}$.
There are three cases:

- Case 1: $f(n)$ is much smaller than $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a}$.
- Case 2: $f(n)$ is the same with $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a} \log n$.

Master Theorem

Intuitive Version

The master theorem allows us to solve the recurrence

$$T(n) = aT(n/b) + f(n)$$

by comparing the function $f(n)$ with $n^{\log_b a}$.
There are three cases:

- Case 1: $f(n)$ is much smaller than $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a}$.
- Case 2: $f(n)$ is the same with $n^{\log_b a}$.
  Then $T(n)$ has complexity $n^{\log_b a} \log n$.
- Case 3: $f(n)$ is much bigger than $n^{\log_b a}$.
  Then $T(n)$ has complexity $f(n)$.

Master Theorem

Master theorem

Let $\alpha \geq 1$ and $b > 1$, let $f(n)$ be a positive function, and let $T(n)$ be defined as:

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then there are three cases:

Master Theorem

Master theorem

Let $\alpha \geq 1$ and $b > 1$, let $f(n)$ be a positive function, and let $T(n)$ be defined as:

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then there are three cases:

1. If $f(n)$ is $O(n^{\log_b a - e})$ for some constant $e > 0$, then

$$T(n) = \Theta(n^{\log_b a}).$$

Master Theorem

Master theorem

Let $\alpha \geq 1$ and $b > 1$, let $f(n)$ be a positive function, and let $T(n)$ be defined as:

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then there are three cases:

1. If $f(n)$ is $O(n^{\log_b a - e})$ for some constant $e > 0$, then

$$T(n) = \Theta(n^{\log_b a}).$$

2. If $f(n)$ is $\Theta(n^{\log_b a})$, then

$$T(n) = \Theta(n^{\log_b a} \log n).$$

Master Theorem

Master theorem

Let $\alpha \geq 1$ and $b > 1$, let $f(n)$ be a positive function, and let $T(n)$ be defined as:

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then there are three cases:

1. If $f(n)$ is $O(n^{\log_b a - e})$ for some constant $e > 0$, then
$$T(n) = \Theta(n^{\log_b a}).$$
2. If $f(n)$ is $\Theta(n^{\log_b a})$, then
$$T(n) = \Theta(n^{\log_b a} \log n).$$
3. If $f(n)$ is $\Omega(n^{\log_b a + e})$ for some constant $e > 0$, and the regularity condition $af(n/b) \leq rf(n)$ for some $r < 1$ holds, then
$$T(n) = \Theta(f(n)).$$

Note: Most of the functions we see satisfy the regularity condition.

Master Theorem

Examples

- $T(n) = 9T(n/3) + n.$

- $T(n) = 4T(n/2) + n^2.$

- $T(n) = 3T(n/3) + n^2.$

Master Theorem

Examples

- $T(n) = 9T(n/3) + n.$
  $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2.$
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2.$
- $T(n) = 3T(n/3) + n^2.$

Master Theorem

Examples

- $T(n) = 9T(n/3) + n$.
  $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2$.
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$.
  $a = 4, b = 2, f(n) = n^2, n^{\log_b a} = n^2$.
  $f(n)$ is $\Theta(n^2)$. Hence we apply case 2.
  $T(n)$ is $\Theta(n^2 \log n)$.
- $T(n) = 3T(n/3) + n^2$.

Master Theorem

Examples

- $T(n) = 9T(n/3) + n$.
  $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2$.
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$.
  $a = 4, b = 2, f(n) = n^2, n^{\log_b a} = n^2$.
  $f(n)$ is $\Theta(n^2)$. Hence we apply case 2.
  $T(n)$ is $\Theta(n^2 \log n)$.
- $T(n) = 3T(n/3) + n^2$.
  $a = 3, b = 3, f(n) = n^2, n^{\log_b a} = n^{\log_3 3} = n$.
  $f(n)$ is $\Omega(n^e)$ for some e (say $e = 0.5$).
  Hence we apply case 3.
  $T(n)$ is $\Theta(n^2)$.

Master Theorem

Examples

- $T(n) = 9T(n/3) + n$.
  $a = 9, b = 3, f(n) = n, n^{\log_b a} = n^2$.
  $f(n)$ is $O(n^{2-e})$ for some e (say $e = 0.5$).
  Hence we apply case 1.
  $T(n)$ is $\Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$.
  $a = 4, b = 2, f(n) = n^2, n^{\log_b a} = n^2$.
  $f(n)$ is $\Theta(n^2)$. Hence we apply case 2.
  $T(n)$ is $\Theta(n^2 \log n)$.
- $T(n) = 3T(n/3) + n^2$.
  $a = 3, b = 3, f(n) = n^2, n^{\log_b a} = n^{\log_3 3} = n$.
  $f(n)$ is $\Omega(n^e)$ for some e (say $e = 0.5$).
  Hence we apply case 3.
  $T(n)$ is $\Theta(n^2)$.

Note: Master theorem holds without assuming n is a power of b.

Master Theorem

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.

Master Theorem

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.
- When $f(n)$ is negative.
  e.g. $T(n) = 2T(n/2) - \log n$.

Master Theorem

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.
- When $f(n)$ is negative.
  e.g. $T(n) = 2T(n/2) - \log n$.
- When $f(n)$ is smaller than $n^{\log_b a}$ but is not small enough:
  ($f(n)$ is $O(n^{\log_b a})$ but $f(n)$ is not $O(n^{\log_b a - e})$ for any $e > 0$)
  e.g. $T(n) = 2T(n/2) + n/\log n$.

Master Theorem

Cases where master theorem doesn't work

- When $a < 1$.
  e.g. $T(n) = 0.5T(n/2) + n$.
- When $f(n)$ is negative.
  e.g. $T(n) = 2T(n/2) - \log n$.
- When $f(n)$ is smaller than $n^{\log_b a}$ but is not small enough:
  ($f(n)$ is $O(n^{\log_b a})$ but $f(n)$ is not $O(n^{\log_b a-e})$ for any $e > 0$)
  e.g. $T(n) = 2T(n/2) + n/\log n$.
- When $f(n)$ is bigger than $n^{\log_b a}$ but is not big enough:
  ($f(n)$ is $\Omega(n^{\log_b a})$ but $f(n)$ is not $\Omega(n^{\log_b a+e})$ for any $e > 0$)
  e.g. $T(n) = 2T(n/2) + n \log n$.

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

- $n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$
- $f(n) = cn$

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

- $n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$
- $f(n) = cn$
- cn is $O(n^{1.59-0.1})$

Example 1: Karatsuba's Algorithm

Running time: $T(n) = 3T(n/2) + cn$

- $n^{\log_b a} = n^{\log_2 3} \approx n^{1.59}$
- $f(n) = cn$
- cn is $O(n^{1.59-0.1})$

Case 1: cn is much smaller than $n^{\log_2 3}$.
$\Rightarrow T(n)$ is $\Theta(n^{\log_2 3})$

Example 2: Merge Sort

Running time: $T(n) = 2T(n/2) + cn$

Example 2: Merge Sort

Running time: $T(n) = 2T(n/2) + cn$

- $n^{\log_2 2} = n$
- $f(n) = cn$

Example 2: Merge Sort

Running time: $T(n) = 2T(n/2) + cn$

- $n^{\log_2 2} = n$
- $f(n) = cn$
- cn is $\Theta(n)$

Example 2: Merge Sort

Running time: $T(n) = 2T(n/2) + cn$

- $n^{\log_2 2} = n$
- $f(n) = cn$
- cn is $\Theta(n)$

Case 2: cn has the same asymptotic growth as n.
$\Rightarrow T(n)$ is $\Theta(n \log n)$

Day 3: Divide and Conquer

Part III: Matrix Multiplication and Strassen's Algorithm

Matrix Multiplications

Problem

INPUT: Two $n \times n$ matrices A, B
OUTPUT: Their product matrix $A \times B$.

This is a crucial process in

- computer graphics
- Linear programming
- Linear dynamical systems
- etc.

Matrix Multiplications

Standard Multiplication Algorithm

The (i, j)-entry of $A \times B$ is $\sum_{k=1}^{n} A[i,k]B[k,j]$, i.e.,

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{pmatrix} \times \begin{pmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ g_2 & h_2 & i_2 \end{pmatrix} =$$

$$\begin{pmatrix} a_1a_2 + b_1d_2 + c_1g_2 & a_1b_2 + b_1e_2 + c_1h_2 & a_1c_2 + b_1f_2 + c_1i_2 \\ d_1a_2 + e_1d_2 + f_1g_2 & d_1b_2 + e_1e_2 + f_1h_2 & d_1c_2 + e_1f_2 + f_1i_2 \\ g_1a_2 + h_1d_2 + i_1g_2 & g_1b_2 + h_1e_2 + i_1h_2 & g_1c_2 + h_1f_2 + i_1i_2 \end{pmatrix}$$

Matrix Multiplications

Standard Multiplication Algorithm

The (i, j)-entry of $A \times B$ is $\sum_{k=1}^{n} A[i,k]B[k,j]$, i.e.,

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{pmatrix} \times \begin{pmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ g_2 & h_2 & i_2 \end{pmatrix} =$$

$$\begin{pmatrix} a_1a_2 + b_1d_2 + c_1g_2 & a_1b_2 + b_1e_2 + c_1h_2 & a_1c_2 + b_1f_2 + c_1i_2 \\ d_1a_2 + e_1d_2 + f_1g_2 & d_1b_2 + e_1e_2 + f_1h_2 & d_1c_2 + e_1f_2 + f_1i_2 \\ g_1a_2 + h_1d_2 + i_1g_2 & g_1b_2 + h_1e_2 + i_1h_2 & g_1c_2 + h_1f_2 + i_1i_2 \end{pmatrix}$$

Standard algorithm: a three-nested loop.

- Inner-most loop: Compute value for an entry
- Middle loop: Compute values in a row
- Outer-most loop: Compute values in all rows

Matrix Multiplications

Standard Multiplication Algorithm

The (i, j)-entry of $A \times B$ is $\sum_{k=1}^{n} A[i,k]B[k,j]$, i.e.,

$$\begin{pmatrix} a_1 & b_1 & c_1 \\ d_1 & e_1 & f_1 \\ g_1 & h_1 & i_1 \end{pmatrix} \times \begin{pmatrix} a_2 & b_2 & c_2 \\ d_2 & e_2 & f_2 \\ g_2 & h_2 & i_2 \end{pmatrix} =$$

$$\begin{pmatrix} a_1a_2 + b_1d_2 + c_1g_2 & a_1b_2 + b_1e_2 + c_1h_2 & a_1c_2 + b_1f_2 + c_1i_2 \\ d_1a_2 + e_1d_2 + f_1g_2 & d_1b_2 + e_1e_2 + f_1h_2 & d_1c_2 + e_1f_2 + f_1i_2 \\ g_1a_2 + h_1d_2 + i_1g_2 & g_1b_2 + h_1e_2 + i_1h_2 & g_1c_2 + h_1f_2 + i_1i_2 \end{pmatrix}$$

Standard algorithm: a three-nested loop.

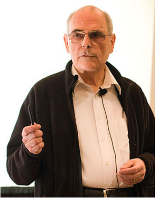
- Inner-most loop: Compute value for an entry
- Middle loop: Compute values in a row
- Outer-most loop: Compute values in all rows

Time complexity $\Theta(n^3)$

Matrix Multiplications

Let's try
Divide-and-Conquer
on this problem

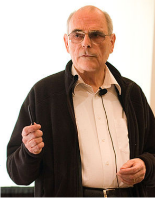
Volker Strassen (1969)
Professor of Math and Stats
University of Konstanz, Germany
Knuth Prize Winner 2008



Matrix Multiplications

Let's try
Divide-and-Conquer
on this problem

Volker Strassen (1969)
Professor of Math and Stats
University of Konstanz, Germany
Knuth Prize Winner 2008



Observations

We may divide a $2n \times 2n$ matrix into four $n \times n$ sub-matrices.

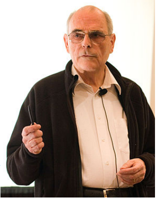
$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{pmatrix} =$$

$$\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}$$

Matrix Multiplications

Let's try
Divide-and-Conquer
on this problem

Volker Strassen (1969)
Professor of Math and Stats
University of Konstanz, Germany
Knuth Prize Winner 2008



Observations

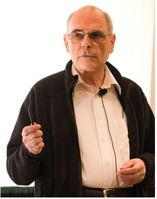
We may divide a $2n \times 2n$ matrix into four $n \times n$ sub-matrices.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{pmatrix} =$$

$$\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}$$

Matrix Multiplications

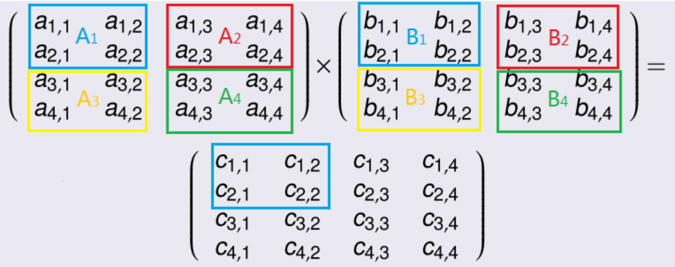
Let's try
Divide-and-Conquer
on this problem



Volker Strassen (1969)
Professor of Math and Stats
University of Konstanz, Germany
Knuth Prize Winner 2008

Observations

We may divide a $2n \times 2n$ matrix into four $n \times n$ sub-matrices.



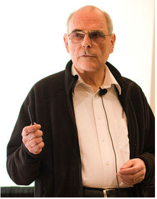
$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{pmatrix} =$$

$$\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}$$

Matrix Multiplications

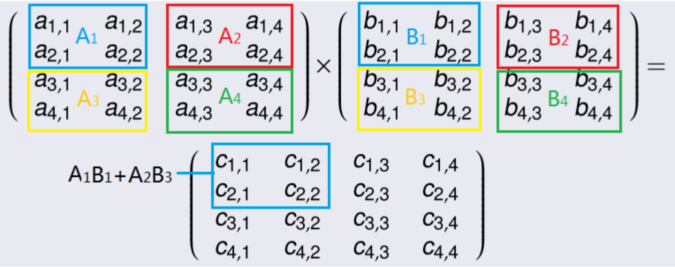
Let's try
Divide-and-Conquer
on this problem

Volker Strassen (1969)
Professor of Math and Stats
University of Konstanz, Germany
Knuth Prize Winner 2008



Observations

We may divide a $2n \times 2n$ matrix into four $n \times n$ sub-matrices.



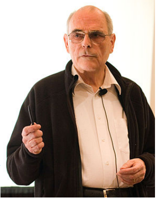
$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \times \begin{pmatrix} b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \\ b_{4,1} & b_{4,2} & b_{4,3} & b_{4,4} \end{pmatrix} =$$

$$A_1B_1+A_2B_3 \quad \begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}$$

Matrix Multiplications

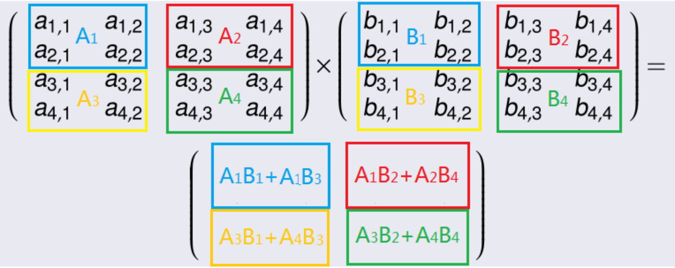
Let's try
Divide-and-Conquer
on this problem

Volker Strassen (1969)
Professor of Math and Stats
University of Konstanz, Germany
Knuth Prize Winner 2008



Observations

We may divide a $2n \times 2n$ matrix into four $n \times n$ sub-matrices.



$$\begin{pmatrix} \boxed{\begin{matrix} a_{1,1} & A_1 & a_{1,2} \\ a_{2,1} & & a_{2,2} \end{matrix}} & \boxed{\begin{matrix} a_{1,3} & A_2 & a_{1,4} \\ a_{2,3} & & a_{2,4} \end{matrix}} \\ \boxed{\begin{matrix} a_{3,1} & A_3 & a_{3,2} \\ a_{4,1} & & a_{4,2} \end{matrix}} & \boxed{\begin{matrix} a_{3,3} & A_4 & a_{3,4} \\ a_{4,3} & & a_{4,4} \end{matrix}} \end{pmatrix} \times \begin{pmatrix} \boxed{\begin{matrix} b_{1,1} & B_1 & b_{1,2} \\ b_{2,1} & & b_{2,2} \end{matrix}} & \boxed{\begin{matrix} b_{1,3} & B_2 & b_{1,4} \\ b_{2,3} & & b_{2,4} \end{matrix}} \\ \boxed{\begin{matrix} b_{3,1} & B_3 & b_{3,2} \\ b_{4,1} & & b_{4,2} \end{matrix}} & \boxed{\begin{matrix} b_{3,3} & B_4 & b_{3,4} \\ b_{4,3} & & b_{4,4} \end{matrix}} \end{pmatrix} =$$

$$\begin{pmatrix} \boxed{A_1B_1 + A_1B_3} & \boxed{A_1B_2 + A_2B_4} \\ \boxed{A_3B_1 + A_4B_3} & \boxed{A_3B_2 + A_4B_4} \end{pmatrix}$$

Matrix Multiplications

Example:

$$\begin{pmatrix} 1 & 4 & 3 & -1 \\ 0 & 2 & -2 & 4 \\ -1 & 0 & 1 & 0 \\ 5 & 2 & 1 & -2 \end{pmatrix} \times \begin{pmatrix} 3 & 1 & -1 & 1 \\ 1 & 0 & -2 & 3 \\ 2 & 3 & 1 & -3 \\ -1 & -2 & 0 & 1 \end{pmatrix}$$

Result:

$$\begin{pmatrix} 14 & 12 & -6 & 3 \\ -6 & -14 & -6 & 16 \\ -1 & 2 & 2 & -4 \\ 21 & 12 & -8 & 8 \end{pmatrix}$$

Matrix Multiplications

First Attempt

Recursively solve the 8 sub-matrices multiplications:

$$A_1B_1, A_2B_3, A_1B_2, A_2B_4, A_3B_1, A_4B_3, A_3B_2, A_4B_4$$

Then some additions ($\Theta(n^2)$-time).
Thus $T(n) = 8T(n/2) + cn^2$.

Matrix Multiplications

First Attempt

Recursively solve the 8 sub-matrices multiplications:

$$A_1B_1, A_2B_3, A_1B_2, A_2B_4, A_3B_1, A_4B_3, A_3B_2, A_4B_4$$

Then some additions ($\Theta(n^2)$-time).
Thus $T(n) = 8T(n/2) + cn^2$.
By Master theorem, $T(n)$ is $\Theta(n^3)$.
No improvement from the standard way.
First attempt fails.

Matrix Multiplications

First Attempt

Recursively solve the 8 sub-matrices multiplications:

$$A_1B_1, A_2B_3, A_1B_2, A_2B_4, A_3B_1, A_4B_3, A_3B_2, A_4B_4$$

Then some additions ($\Theta(n^2)$-time).
Thus $T(n) = 8T(n/2) + cn^2$.
By Master theorem, $T(n)$ is $\Theta(n^3)$.
No improvement from the standard way.
First attempt fails.

Goal: "Group" some of the multiplications together so we need < 8 sub-matrix multiplication.

Strassen’s Algorithm

- $P_1 = A_1(B_2 - B_4)$
- $P_2 = (A_1 + A_2)B_4$
- $P_3 = (A_3 + A_4)B_1$
- $P_4 = A_4(B_3 - B_1)$
- $P_5 = (A_1 + A_4)(B_1 + B_4)$
- $P_6 = (A_2 - A_4)(B_3 + B_4)$
- $P_7 = (A_1 - A_3)(B_1 + B_2)$

$$\left(\begin{array}{c|c} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ \hline A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{array}\right) =$$

$$\left(\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{array}\right)$$

Thus we only need 7 multiplications of sub-matrices.

Strassen's Algorithm

Strassen's Algorithm

Given two input $n \times n$ matrices A, B, do the following:

- If A, B have very small dimensions, directly multiply them
- Otherwise divide A,B into $A_1, \ldots, A_4$, $B_1, \ldots, B_4$.
- Compute $P_1, \ldots, P_7$, each use one recursive call.
- Then add and subtract P_i's to get the output matrix $A \times B$

Strassen's Algorithm

Strassen's Algorithm

Given two input $n \times n$ matrices A, B, do the following:

- If A, B have very small dimensions, directly multiply them
- Otherwise divide A,B into $A_1, \ldots, A_4$, $B_1, \ldots, B_4$.
- Compute $P_1, \ldots, P_7$, each use one recursive call.
- Then add and subtract P_i's to get the output matrix $A \times B$

Complexity

Let $T(n)$ be the time it takes to multiples two $n \times n$ matrices.
We have $T(n) = 7T(n/2) + cn^2$
By Master theorem, $T(n)$ is $\Theta(n^{\log 7}) \approx \Theta(n^{2.808})$
This is asymptotically better than $O(n^3)$!

Divide and Conquer Summary

- Algorithm design technique: Divide and Conquer
  - Partition the problems into subproblems
  - Combine sub-solutions to overall solution
- Analysis Technique for Divide and Conquer: Master Theorem
- Karatsuba's algorithm (Integer multiplication): $O(n^{1.59})$
- Strassen's algorithm (Matrix multiplication): $O(n^{2.808})$