Algorithm Design IV

Divide and Conquer I

Guoqiang Li
School of Software, Shanghai Jiao Tong University

SHANGHAI JIAO TONG UNIVERSITY

Divide-and-Conquer

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

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.
2. Recursively solving these subproblems.
3. Appropriately combining their answers.

Multiplication

Product of Complex Numbers

Carl Friedrich Gauss(1777-1855) 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

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

Product of Complex Numbers

Carl Friedrich Gauss(1777-1855) 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 big=O way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity.
- But this modest improvement becomes very significant when applied recursively.

Multiplication

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

Multiplication

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

[Hints: For every n there exists an n' with $n \leq n' \leq 2n$ such that n' a power of 2.]

Multiplication

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

[Hints: For every n there exists an n' with $n \leq n' \leq 2n$ such that n' 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

$$x = \boxed{x_L}\ \boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\ \boxed{y_R} = 2^{n/2}y_L + y_R$$

Multiplication

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

[Hints: For every n there exists an n' with $n \leq n' \leq 2n$ such that n' 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

$$x = \boxed{x_L}\ \boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\ \boxed{y_R} = 2^{n/2}y_L + y_R$$

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

Multiplication

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

[Hints: For every n there exists an n' with $n \leq n' \leq 2n$ such that n' 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

$$x = \boxed{x_L}\ \boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L}\ \boxed{y_R} = 2^{n/2}y_L + y_R$$

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

Additions and multiplications by powers of 2 take linear time.

Multiplication

The additions take linear time, as do multiplications by powers of 2 (that is, $O(n)$).

Multiplication

The additions take linear time, as do multiplications by powers of 2 (that is, $O(n)$).

The significant operations are the four $n/2$-bit multiplications: these can be handled by four recursive calls.

Multiplication

The additions take linear time, as do multiplications by powers of 2 (that is, $O(n)$).

The significant operations are the four $n/2$-bit multiplications: these can be handled by four recursive calls.

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

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

Multiplication

The additions take linear time, as do multiplications by powers of 2 (that is, $O(n)$).

The significant operations are the four $n/2$-bit multiplications: these can be handled by four recursive calls.

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

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

Solution: $O(n^2)$

Multiplication

The additions take linear time, as do multiplications by powers of 2 (that is, $O(n)$).

The significant operations are the four $n/2$-bit multiplications: these can be handled by four recursive calls.

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

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

Solution: $O(n^2)$

By Gauss's trick, three multiplications $x_L y_L$, $x_R y_R$, and $(x_L + x_R)(y_L + y_R)$ suffice.

Algorithm for Integer Multiplication

MULTIPLY(x, y)

Two positive integers x and y, in binary;

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;

$P1$=MULTIPLY(x_L, y_L);

$P2$=MULTIPLY(x_R, y_R);

$P3$=MULTIPLY($x_L + x_R, y_L + y_R$);

return($P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$)

Time Analysis

The recurrence relation

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

Time Analysis

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

Time Analysis

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

At each successive level of recursion the subproblems get halved in size.

Time Analysis

SHANGHAI JIAO TONG UNIVERSITY

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

At each successive level of recursion the subproblems get halved in size.

At the $(log_2 n)^{th}$ level, the subproblems get down to size 1, and so the recursion ends.

Time Analysis

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

At each successive level of recursion the subproblems get halved in size.

At the $(log_2 n)^{th}$ level, the subproblems get down to size 1, and so the recursion ends.

The height of the tree is $\log_2 n$.

Time Analysis

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

At each successive level of recursion the subproblems get halved in size.

At the $(log_2 n)^{th}$ level, the subproblems get down to size 1, and so the recursion ends.

The height of the tree is $\log_2 n$.

The branch factor is 3: each problem produces three smaller ones, with the result that at depth k there are 3^k subproblems, each of size $n/2^k$.

Time Analysis

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

At each successive level of recursion the subproblems get halved in size.

At the $(log_2 n)^{th}$ level, the subproblems get down to size 1, and so the recursion ends.

The height of the tree is $\log_2 n$.

The branch factor is 3: each problem produces three smaller ones, with the result that at depth k there are 3^k subproblems, each of size $n/2^k$.

For each subproblem, a linear amount of work is done in combining their answers.

Time Analysis

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

Time Analysis

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

At the top level, when $k = 0$, we need $O(n)$.

Time Analysis

SHANGHAI JIAO TONG UNIVERSITY

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

At the top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n}) = O(n^{\log_2 3})$

Time Analysis

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

At the top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n}) = O(n^{\log_2 3})$

The work done increases geometrically from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level.

Time Analysis

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

At the top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n}) = O(n^{\log_2 3})$

The work done increases geometrically from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level.

The sum of any increasing geometric series is, within a constant factor, the last term of the series.

Time Analysis

Shanghai Jiao Tong University

The total time spent at depth k in the tree is

$$3^k \times O(\frac{n}{2^k}) = (\frac{3}{2})^k \times O(n)$$

At the top level, when $k = 0$, we need $O(n)$.

At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n}) = O(n^{\log_2 3})$

The work done increases geometrically from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level.

The sum of any increasing geometric series is, within a constant factor, the last term of the series.

Therefore, the overall running time is

$$O(n^{\log_2 3}) \approx O(n^{1.59})$$

Time Analysis

Q: Can we do better?

Time Analysis

Q: Can we do better?

- Yes!

Recurrence Relations

Master Theorem

Master Theorem

If $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ for some constants $a > 0$, $b > 1$ and $d \geq 0$, then

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

The Proof of the Theorem

Proof:

The Proof of the Theorem

Proof:

Assume that n is a power of b.

The Proof of the Theorem

Proof:

Assume that n is a power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels - the the height of the recursion tree.

The Proof of the Theorem

Proof:

Assume that n is a power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels - the the height of the recursion tree.

Its branching factor is a, so the k-th level of the tree is made up of a^k subproblems, each of size n/b^k.

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$$

The Proof of the Theorem

Proof:

Assume that n is a power of b.

The size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels - the the height of the recursion tree.

Its branching factor is a, so the k-th level of the tree is made up of a^k subproblems, each of size n/b^k.

$$a^k \times O(\frac{n}{b^k})^d = O(n^d) \times (\frac{a}{b^d})^k$$

k goes from 0 to $\log_b n$, these numbers form a geometric series with ratio a/b^d, comes down to three cases.

The Proof of the Theorem

The ratio is less than 1.

Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

The Proof of the Theorem

The ratio is less than 1.

Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

The ratio is greater than 1.

The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$

The Proof of the Theorem

The ratio is less than 1.

Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

The ratio is greater than 1.

The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$

The ratio is exactly 1.

In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

Merge Sort

Merge Sort

The Algorithm


```
MERGESORT(a[1...n])
An array of numbers a[1...n];
if n > 1 then
    return(MERGE(MERGESORT(a[1...⌊n/2⌋]),
     MERGESORT(a[⌊n/2⌋ + 1...,n])));
    else return(a);
end
```

The Algorithm


```
MERGESORT(a[1...n])
An array of numbers a[1...n];
if n > 1 then
    return(MERGE(MERGESORT(a[1...⌊n/2⌋]),
     MERGESORT(a[⌊n/2⌋ + 1...,n])));
    else return(a);
end
```

```
MERGE(x[1...k], y[1...l])
if k = 0 then return y[1...l];
if l = 0 then return x[1...k];
if x[1] ≤ y[1] then return( x[1]∘MERGE(x[2...k], y[1...l]));
else return( y[1]∘MERGE(x[1...k], y[2...l]));
```

An Iterative Version

SHANGHAI JIAO TONG UNIVERSITY

```
ITERTIVE-MERGESORT(a[1...n])
An array of numbers a[1...n];
Q = [  ]empty queue;
for i = 1 to n do
    Inject(Q, [a[i]]);
end
while |Q| > 1 do
    Inject (Q,MERGE (Eject (Q),Eject (Q)));
end
return(Eject (Q));
```

The Time Analysis

The recurrence relation:

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

The Time Analysis

The recurrence relation:

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

By Master Theorem:

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

The Time Analysis

The recurrence relation:

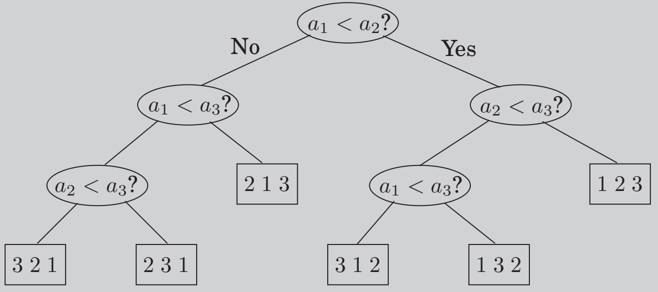
$$T(n) = 2T(n/2) + O(n)$$

By Master Theorem:

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

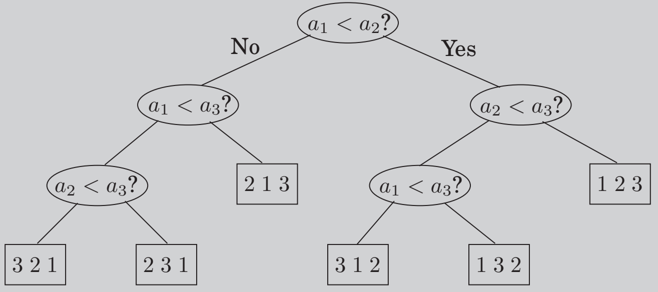
Q: Can we do better?

Sorting



Sorting algorithms can be depicted as trees.

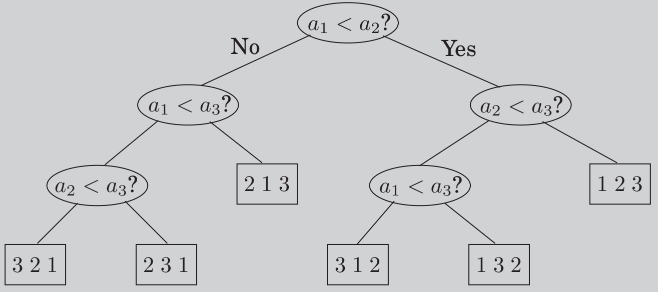
Sorting



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 the worst-case time complexity of the algorithm.

Sorting

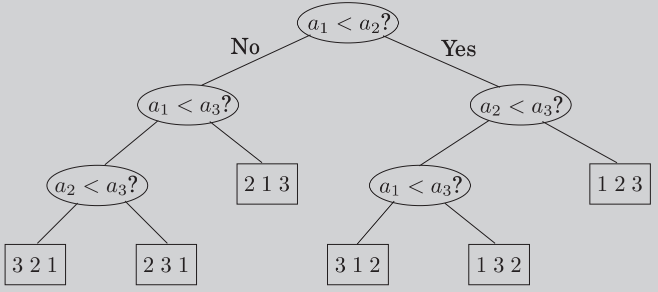


Sorting algorithms can be depicted as trees.

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

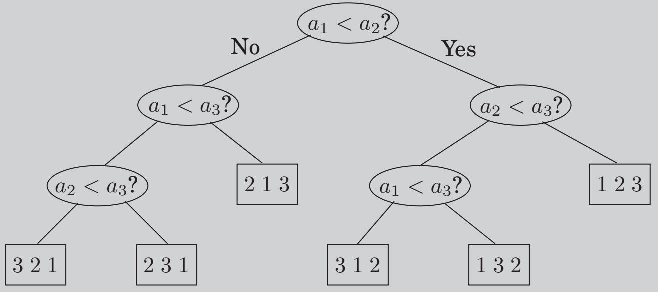
Assume n elements. Each of its leaves is labeled by a permutation of $\{1, 2, \ldots, n\}$.

Sorting



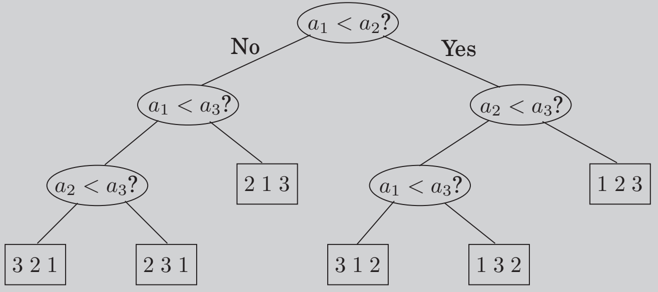
Every permutation must appear as the label of a leaf.

Sorting



Every permutation must appear as the label of a leaf.

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



Every permutation must appear as the label of a leaf.

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

So, the depth of the tree - and the complexity of the algorithm - must be at least

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

Median

Median

The median of a list of numbers is its $50th$ percentile: half the number are bigger than it, and half are smaller.

Median

The median of a list of numbers is its $50th$ percentile: half the number are bigger than it, and half are smaller.

If the list has even length, we pick the smaller one of the two.

Median

The median of a list of numbers is its $50th$ percentile: half the number are bigger than it, and half are smaller.

If the list has even length, we pick the smaller one of the two.

The purpose of the median is to summarize a set of numbers by a single typical value.

Median

The median of a list of numbers is its $50th$ percentile: half the number are bigger than it, and half are smaller.

If the list has even length, we pick the smaller one of the two.

The purpose of the median is to summarize a set of numbers by a single typical value.

Computing the median of n numbers is easy, just sort them. ($O(n \log n)$).

Median

The median of a list of numbers is its $50th$ percentile: half the number are bigger than it, and half are smaller.

If the list has even length, we pick the smaller one of the two.

The purpose of the median is to summarize a set of numbers by a single typical value.

Computing the median of n numbers is easy, just sort them. ($O(n \log n)$).

Q: Can we do better?

Selection

Input: A list of number S; an integer k.
Output: The $k\,th$ smallest element of S.

A Randomized Selection

For any number v, imagine splitting list S into three categories:

- elements smaller than v, i.e., S_L;
- those equal to v, i.e., S_v (there might be duplicates);
- and those greater than v, i.e., S_R; respectively.

A Randomized Selection

SHANGHAI JIAO TONG UNIVERSITY

For any number v, imagine splitting list S into three categories:

- elements smaller than v, i.e., S_L;
- those equal to v, i.e., S_v (there might be duplicates);
- and those greater than v, i.e., S_R; respectively.

$$selection(S,k) = \begin{cases} selection(S_L, k) & \text{if } k \le |S_L| \\ v & \text{if } |S_L| < k \le |S_L| + |S_v| \\ selection(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v| \end{cases}$$

How to Choose v?

How to Choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$| S_L |, | S_R | \approx \frac{| S |}{2}$$

How to Choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$\mid S_L \mid, \mid S_R \mid \approx \frac{\mid S \mid}{2}$$

If we could always guarantee this situation, we would get a running time of

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

How to Choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$| S_L |, | S_R | \approx \frac{| S |}{2}$$

If we could always guarantee this situation, we would get a running time of

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

But this requires picking v to be the median, which is our ultimate goal!

How to Choose v?

It should be picked quickly, and it should shrink the array substantially, the ideal situation being

$$|S_L|, |S_R| \approx \frac{|S|}{2}$$

If we could always guarantee this situation, we would get a running time of

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

But this requires picking v to be the median, which is our ultimate goal!

Instead, we pick v randomly from S!

How to Choose v?

Worst-case scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \ldots + \frac{n}{2} = \Theta(n^2)$$

How to Choose v?

Worst-case scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \ldots + \frac{n}{2} = \Theta(n^2)$$

Best-case scenario $O(n)$

The Efficiency Analysis

v is good if it lies within the $25th$ to $75th$ percentile of the array that it is chosen from.

The Efficiency Analysis

Shanghai Jiao Tong University

v is good if it lies within the $25th$ to $75th$ percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good.

The Efficiency Analysis

v is good if it lies within the $25th$ to $75th$ percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good.

Lemma

On average a fair coin needs to be tossed two times before a heads is seen.

The Efficiency Analysis

v is good if it lies within the $25th$ to $75th$ percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good.

Lemma

On average a fair coin needs to be tossed two times before a heads is seen.

Proof:

The Efficiency Analysis

v is good if it lies within the $25th$ to $75th$ percentile of the array that it is chosen from.

A randomly chosen v has a 50% chance of being good.

Lemma

On average a fair coin needs to be tossed two times before a heads is seen.

Proof:

Let E be the expected number of tosses before heads is seen.

$$E = 1 + \frac{1}{2}E$$

Therefore, $E = 2$.

The Efficiency Analysis

Let $T(n)$ be the expected running time on the array of size n, we get

$$T(n) \le T(3n/4) + O(n) = O(n)$$

Matrix Multiplication

Matrix

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

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

Matrix

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

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

In general, matrix multiplication is not commutative, say, $XY \neq YX$

Matrix

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

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

In general, matrix multiplication is not commutative, say, $XY \neq YX$

The running time for matrix multiplication is $O(n^3)$

- There are n^2 entries to be computed, and each takes $O(n)$ time.

Divide-and-Conquer

Matrix multiplication can be performed blockwise.

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

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

Divide-and-Conquer

Matrix multiplication can be performed blockwise.

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

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

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

$$T(n) = O(n^3)$$

Strassen Algorithm

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

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

Strassen Algorithm

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

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

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

Strassen Algorithm

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

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

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

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

Strassen Algorithm

SHANGHAI JIAO TONG UNIVERSITY

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

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

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

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

$$T(n) = O(n^{\log_2 7}) \approx O(n^{2.81})$$