

Algorithm Design IV

Divide and Conquer I

Guoqiang Li School of Software, Shanghai Jiao Tong University



Divide-and-Conquer



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

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

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.



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



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 $n \le n' \le 2n$ such that n' a power of 2.]



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 $n \le n' \le 2n$ 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

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

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



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 $n \le n' \le 2n$ 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

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

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

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



对于位数不够的情况,在前面添加0补位,由下面的HINT知道,最多扩展成2*n位,所以对于0时间性能无影响

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 $n \le n' \le 2n$ 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.



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



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.



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:

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



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: $O(n^2)$



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: $O(n^2)$

这里的0(n)指的是在每一步递归运算之后需要额外做的操作,这里主要是加法, 2的幂次的乘法为0(1)。

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 = \text{leftmost } n/2, rightmost n/2 bits of x;

y_L, y_R = \text{leftmost } n/2, rightmost n/2 bits of y;

P1 = \text{MULTIPLY } (x_L, y_L);

P2 = \text{MULTIPLY } (x_R, y_R);

P3 = \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)
```



The recurrence relation

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



The recurrence relation

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

The algorithm's recursive calls form a tree structure.



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.



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 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 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 $n/2^k$.



具体的树状结构可参考文件夹下的"分治算法树状图.png"

The recurrence relation

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

The algorithm's recursive calls form a tree structure.

在处理时我们可以直接将每一层的问题处理时间相加,是因为在这个情况下,每一个子问题在合并成父问题时花费的时间是线性的,即是与问题规模成正比的。

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.



The total time spent at depth k in the tree is

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



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).



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 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 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.



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

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



Q: Can we do better?



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 $d \ge 0$, 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.



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 .

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



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 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 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 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



```
\label{eq:mergesort} \begin{split} & \text{MERGESORT} \ (a[1 \dots n]) \\ & \textit{An array of numbers } a[1 \dots n]; \\ & \text{if } n > 1 \ \text{then} \\ & \quad \text{return} \ (\text{MERGE (MERGESORT} \ (a[1 \dots \lfloor n/2 \rfloor]) \ , \\ & \quad \text{MERGESORT} \ (a[\lfloor n/2 \rfloor + 1 \dots, n]) \ ) \ ); \\ & \quad \text{else} \ \text{return} \ (a) \ ; \\ & \quad \text{end} \end{split}
```

The Algorithm



```
 \begin{split} & \text{MERGESORT} \ (a[1 \dots n]) \\ & \textit{An array of numbers } a[1 \dots n]; \\ & \text{if } n > 1 \text{ then} \\ & \text{return} \ (\text{MERGE} \ (\text{MERGESORT} \ (a[1 \dots \lfloor n/2 \rfloor]) \ , \\ & \text{MERGESORT} \ (a[\lfloor n/2 \rfloor + 1 \dots, n]) \ ) \ ); \\ & \text{else} \ \text{return} \ (a); \\ & \text{end} \end{split}
```

```
\begin{split} & \text{MERGE}\,(x[1\ldots k],y[1\ldots l]) \\ & \text{if } k=0 \text{ then } \text{return } y[1\ldots l]; \\ & \text{if } l=0 \text{ then } \text{return } x[1\ldots k]; \\ & \text{if } x[1] \leq y[1] \text{ then } \text{return } (\,x[1] \text{oMERGE}\,(x[2\ldots k],y[1\ldots l])\,)\,; \\ & \text{else } \text{return } (\,y[1] \text{oMERGE}\,(x[1\ldots k],y[2\ldots l])\,)\,; \end{split}
```

An Iterative Version



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:

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

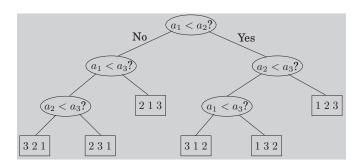
By Master Theorem:

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

对于基于比较的排序算法,0(nlogn)已经是最优的了。

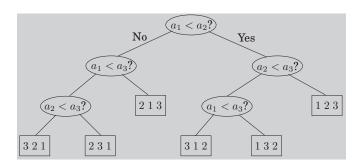
Q: Can we do better?





Sorting algorithms can be depicted as trees.



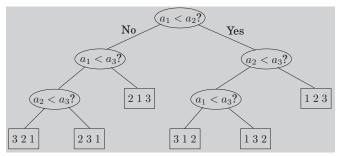


Sorting algorithms can be depicted as trees.

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.



将基于比较的排序算法 可能输入以树的叶子结点的 形式展现出来,则叶子结点 的个数为n!,即对于n个输入 数字的所有全排列的组合数 量。

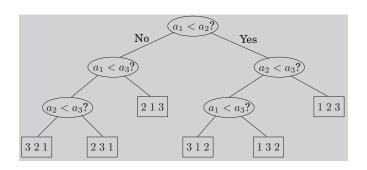


Sorting algorithms can be depicted as trees.

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.

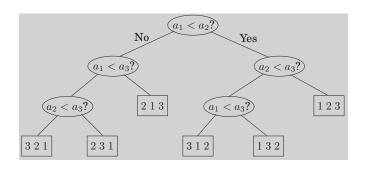
Assume n elements. Each of its leaves is labeled by a permutation of $\{1, 2, \ldots, n\}$.





Every permutation must appear as the label of a leaf.

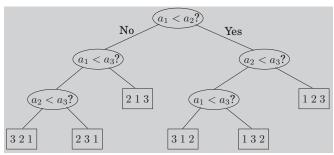




Every permutation must appear as the label of a leaf.

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.

我们是可以证明0(logn!) = (nlogn)的:使用n^n>=n!证明上界,使用(n/2)^(n/2) <=n!证明下界。两边取log即可。 而正是因为基于比较的排序算法的0(nlogn) 已经是 级别的时间复杂度了,所以已经 达到其上确界,所以就不能在优化了。

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

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



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



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 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.



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



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?

但是直接的排序会导致额外的不必要的操作,因为我们只需要排名为正中间 的那个数字。

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



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.

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





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

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



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

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



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!



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

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

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

Instead, we pick v randomly from S!



Worst-case scenario would force our selection algorithm to perform

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



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.



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.



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.



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:



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:

实际上就是求概率基为p的帕斯卡分布的数学期望,为1/p,故此处的E=2。

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

Therefore, E=2.



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

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

由于选中较好的v的gai I v为0.5,所以根据上一页的定理,则最终结果平均需要两次划分可以得到。 而两次划分之后,数组剩余的问题规模至多为原来的0.75倍,(在这种情况下指的是第一次划分正好 在25%或者75%位置发生,并且下一次的查找在剩下的3/4的部分中查找,并且在第二次的时候无需划 分就直接找到目标结果了)所以有上述公式。

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.

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

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

Divide-and-Conquer



Matrix multiplication can be performed blockwise.

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

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

$$T(n) = 8T(n/2) + {O(n^2) \over O(n^2)}$$
 (这里是 $0(n^2)$ 是因为每一个子矩阵都有 $n^2/4$ 个元素需要加在一起求和,而在这里认为加法和乘法都是单位时间的)



$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$
$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$



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

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

$$P_1 = A(F - H) \quad P_5 = (A + D)(E + H)$$

$$P_2 = (A + B)H \quad P_6 = (B - D)(G + H)$$

$$P_3 = (C + D)E \quad P_7 = (A - C)(E + F)$$

$$P_4 = D(G - E)$$



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

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

$$P_1 = A(F - H) \quad P_5 = (A + D)(E + H)$$

$$P_2 = (A + B)H \quad P_6 = (B - D)(G + H)$$

$$P_3 = (C + D)E \quad P_7 = (A - C)(E + F)$$

$$P_4 = D(G - E)$$

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



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

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

$$P_1 = A(F - H) \quad P_5 = (A + D)(E + H)$$

$$P_2 = (A + B)H \quad P_6 = (B - D)(G + H)$$

$$P_3 = (C + D)E \quad P_7 = (A - C)(E + F)$$

$$P_4 = D(G - E)$$

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

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