Chapter 2
Divide and conquer

The main idea for the divide and conquer is trying to divide a problem into several subproblems, which are similar to the original problem but smaller. Solving the subproblems recursively and combining these solutions will solve the original problem.

The divide and conquer paradigm involves three steps at each level of the recursion:

- Divide: break the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer: solve the subproblems recursively or straightforward if the subproblem sizes are small enough.
- Combine: solve the problem by combining the solutions of subproblems.

To analysis the efficiency of an divide and conquer algorithm, we will have to consider recurrences in many cases. A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. For example,

$$T(n) = 2T(n/2) + \Theta(n),\ T(n) = T(2n/3) + T(n/3) + \Theta(n).$$

To solve the recurrences, we will look at several methods.

- Substitution method: first guess a bound and then use mathematical induction to prove it.
- Recursion-tree method: convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. then use techniques for bounding summations to solve the recurrences.
- The Master method: suppose the bounds for recurrences is of the form

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

  where $a \geq 1, b > 1$, and $f(n)$ is a given function. We will use the maximum-subarray problem to explain this method.

The maximum-subarray problem

Example: The Figure 1 shows the price of a stoke A over a 17 day period. Our goal is to maximize the profit. Suppose we are only allowed to buy once and sell once during this period, what are the best dates?

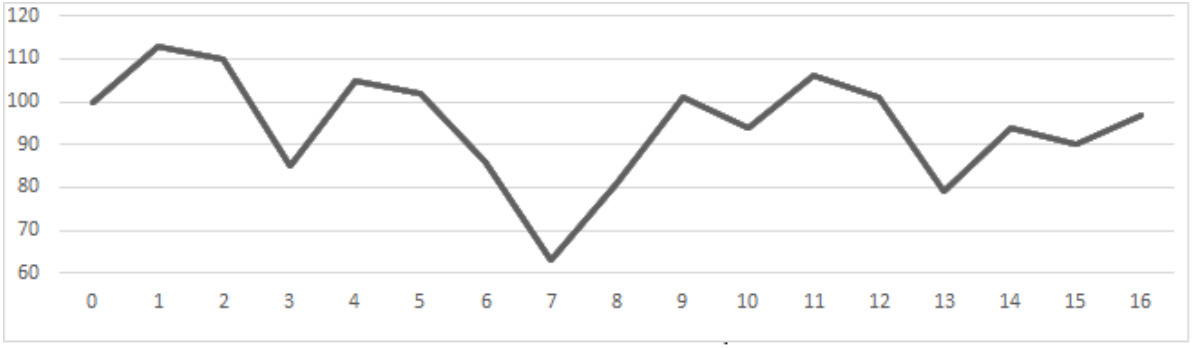

Figure 1: Stoke example

In general, it is not necessary that buy the stoke at the lowest price or sell the at the highest price will make the best profit. To simplify the discussion, we want to find two days from the stoke graph to get highest profit.

A straightforward way is calculating all the possibilities of the buying and selling days. Using this method, we need to compute $\binom{n}{2}$ pair of values. Therefore the rough time complicity is $\Omega(n^2)$.

We can find a better method. For this purpose, we do some transformation of the problem. Instead of computing the subtraction of the values of two days, we use the daily changes and compute the maximum subarray. In Table 1 we record the daily changes of the value of the stoke at the third row. So the question now has changed to find out the maximum subarry of the daily change array.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Table 1: Daily change array

Now we consider how to use the divide-and-conquer method to solve the maximum subarray problem. Suppose we have already found the maximum subarray. Then the subarray must be one of the following cases.

1. The subarray sets in the first half (left) of the original array.
2. The subarray sets in the second half (right) of the original array.
3. The subarray sets across the middle of the original array.

- If we can find the maximum subarrays of the above three cases, then we will be easily find the solution.
- the cases 1 and 2 is the problem similar to the original problem, but the data size is half of that of the original problem. This is the main idea about dividing the problem.
- To conquer the problems, we need to solve the case 3.

Suppose we have an array $A[low, \ldots, high]$ and the midpoint of the array is $A[mid]$. Assume that we have found such a subarray $A[i, \ldots, mid, mid + 1, \ldots, j]$ which is the maximum subarray of the array A.

Then it must be the case that $A[i, \ldots, mid]$ is greater than any subarray with the form $A[t, \ldots, mid]$ for any t, where $low \leq t \leq mid$, and $A[mid + 1, \ldots, high]$ is greater than any subarray with the form $A[mid + 1, \ldots, t]$ for any t, where $mid + 1 \leq t \leq high$.

It is not difficult to find the maximum subarray with the form $A[t, \ldots, mid]$ from the left half of array A and the maximum subarray with the form $A[mid + 1, high]$ from the right half of A.

We can start from the mid-point and go down to left one by one to calculate the sum. Keeping the largest sum and remember the position i such that the sum of $A[i, mid]$ is maximum.

Using a similar method we can find the maximum subarray with the form $A[mid + 1, \ldots, j]$.

Procedure FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$ finds the subarray, where A is the original array, and the indices low, mid and $high$ are know.

```
1:  procedure
    FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
2:      left-sum = −∞
3:      sum = 0
4:      for i = mid downto low do
5:          sum = sum +A[i]
6:          if sum > left-sum then
7:              left-sum = sum
8:              max-left = i
9:          end if
10:     end for
11:     right-sum = −∞
12:     sum = 0
```

```
13:     for j = mid + 1 to high do
14:         sum = sum + A[j]
15:         if sum > right-sum then
16:             right-sum = sum
17:             max-right = j
18:         end if
19:     end for
20:     return  (max-left, max-right, left-sum + right-sum)
21: end procedure
```

Running time for the procedure FIND-MAX-CROSSING-SUBARRAY.

In this procedure, the main running time is used by two **for** loops, one is the lines 4-10 and another is the lines 13 - 20. Other lines are for initializing variables and take constant time.

In each loop, a iteration also takes a constant time. Therefore we can compute the running time by counting the number of iterations.

In the first loop, it takes $mid - low + 1$ iterations and the second loop takes $high - mid$ iterations. Suppose the size of the array is n. Then the total number of iterations is
$(mid - low + 1) + (high - mid) = high - low - 1 = n$. We declare that the running time for this procedure is $\Theta(n)$.

With the above procedure, we can use the divide-and-conquer algorithm to solve the maximum subarray problem.

```
1: procedure FIND-MAXIMUM-SUBARRAY(A, low, high)
2:     if high == low then        // A only has one element
3:         return  (low, high, A[low])
4:     else
5:         mid = ⌊(low + high)/2⌋
6:         (left-low, left-high, left-sum) =
   FIND-MAXIMUM-SUBARRAY (A, low, mid)
7:         (right-low, right-high, right-sum) =
   FIND-MAXIMUM-SUBARRAY (A, mid + 1, high)
8:         (cross-low, cross-high, cross-sum) =
   FIND-MAX-CROSSING-SUBARRAY (A, low, mid, high)
9:     end if
10:    if  left-sum ≥right-sum and left-sum ≥ cross-sum then
11:        return  (left-low, left-high, left-sum)
```

```
12:     else if  right-sum ≥ left-sum and right-sum ≥ cross-sum
    then
13:         return  (right-low, right-high, right-sum)
14:     else
15:         return (cross-low, cross-high, cross-sum)
16:     end if
17: end procedure
```

Running time

In this procedure, lines 2 - 3 treats the base case, in which the array has one element. Since the recursion methods are used, it is important that there is a stop point of each recursion in the procedure. The base cases take constant running time.

Lines 10 - 16 returns the maximum subarray among the three subarrays found. It also takes constant running time.

Lines 6 and 7 use recursive methods. Suppose the running time for the FIND-MAXIMUM-SUBARRAY is $T(n)$, where n is the size of A. For simplicity, we assume that n is a power of 2.

Then line 6 and line 7 needs running time $2T(n/2)$. We already know that line 8 takes $\Theta(n)$ running time.

Therefore we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n). \end{aligned}$$

In general we have the running time $T(n)$ for
Find-Maximum-Subarray

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

We will decide that $T(n) = \Theta(n \log n)$ from the above recursive formula later. So the divide-and-conquer method for maximum subarray problem is better than the brute-force method which is $\Omega(n^2)$.

We note here that the divide-and-conquer method for the maximum subarray problem is not optimal. There is a linear algorithm to solve that problem.

Strassen's algorithm

Suppose we have two $n \times n$ matrices $A(a_{i,j})$ and $B(b_{i,j})$. Then the element $c_{i,j}$ of the product $C = A \cdot B$ is defined as:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}.$$

Using a straightforward way to compute the matrices multiplication will need $\Theta(n^3)$ running time.

```
1:  procedure SQUARE-MATRIX-MULTIPLY(A, B)
2:      n = A.rows
3:      let C be a new n × n matrix
4:      for i = 1 to n do
5:          for j = 1 to n do
6:              c_{i,j} = 0
7:              for k = 1 to n do
8:                  c_{i,j} = c_{i,j} + a_{i,k} · b_{k,j}
9:              end for
10:         end for
11:     end for
12:     return C
13: end procedure
```

Consider using divide-and-conquer.

Suppose two $n \times n$ matrices A and B are given and $C = A \cdot B$. We partition these matrices into four $n/2 \times n/2$ matrices.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

Since $C = A \cdot B$, we have

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}.$$

We obtain the following equations.

$$
\begin{aligned}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{aligned}
$$

Suppose the running time for the multiplying two $n \times n$ matrices is $T(n)$. Then the running time for the 8 multiplications of the $n/2 \times n/2$ is $8T(n/2)$. We also need to do the matrices additions. Adding two $n/2 \times n/2$ matrices need $n^2/4$ additions. So the total addition time is $\Theta(n^2)$. We have the recurrence equation of the running time

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

We will see later that the above simple divide-and-conquer method still takes $\Theta(n^3)$ time.

Strassen's method improves the above method by reducing the 8 multiplications of sub-matrices to 7.

The Strassen's method has four steps:

1. Divide the matrices A, B and C into $n/2 \times n/2$ submatrices as in the previous simple divide-and-conquer method.

2. Create 10 matrices by addition as follows:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

3. Do 7 multiplications of submatrices:

$$\begin{aligned} P_1 &= A_{11} \cdot S_1 \\ P_2 &= S_2 \cdot B_{22} \\ P_3 &= S_3 \cdot B_{11} \\ P_4 &= A_{22} \cdot S_4 \\ P_5 &= S_5 \cdot S_6 \\ P_6 &= S_7 \cdot S_8 \\ P_7 &= S_9 \cdot S_{10} \end{aligned}$$

4. Compute the matrix C as follows:

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

Running time Step 1 uses constant time, because we can just compute the index. Step 2 performs 10 matrices addition each of $n^2/4$ number additions. Therefore step 2 needs $10\frac{n^2}{4}$ addition, which is $\Theta(n^2)$. Step 3 needs running time $7T(n/2)$. Step 4 calculates 8 submatrices additions, which is also $\Theta(n^2)$.

So the running time for the Stassen's method is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T(n/2) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

Methods for solving recurrences

To calculate the running time for the divide-and-conquer method, usually we need to solve recurrences. Now, we look at several methods for solving recurrences.

The substitution method

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

We can use substitution method to establish either upper or lower bounds on a recurrence.

We use the following example to explain the method. Suppose we need to solve the recurrence (which is from the FIND-MAXIMUM-SUBARRAY procedure).

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n/2) + n & \text{if } n > 1 \end{cases}$$

- We guess that the solution is $T(n) = O(n \lg n)$. Next we need to prove the guess is correct.
- We need to prove the guess is correct for the boundary condition $T(n) \leq cn \lg n$ for some constant c.
- For the simplicity, we assume that $n = 2^k$ and use the mathematical induction.

First for the base case: $k = 1$. We have
$T(2) = 2T(1) + 2 = 4 \leq c2 \log_2 2$, where $c \geq 2$ a constant.

Now assume the claim is true for k. So we have
$T(2^k) \leq c2^k \log_2 2^k = ck2^k$. We nee to prove that the claim is true for $n = 2^{k+1}$. By the assumption, we have

$$\begin{aligned} T(2^{k+1}) &= 2T(2^k) + 2^{k+1} \\ &\leq 2ck2^k + 2^{k+1} \\ &= ck2^{k+1} + 2^{k+1} \\ &= 2^{k+1}(ck + 1) \\ &< c(k+1)2^{k+1} \\ &= c2^{k+1} \log_2 2^{k+1} \end{aligned}$$

From the mathematical induction, we have proved that
$T(n) = O(n \lg n)$.

To use the substitution method, the correct guess is the key work. Unfortunately, there is no general method to give a good guess. We will see how to using recursion-tree to get a good guess later.

Some methods are used to the proof of substitution method. We use examples to explain the methods.

Consider the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

We guess that $T(n) = O(n)$ and try to prove $T(n) \leq cn$ for some constant c. Assume it is true to any $m < n$, now we want to prove the case of n. By induction we have

$$\begin{aligned} T(n) &\leq c(\lfloor n/2 \rfloor) + c(\lceil n/2 \rceil) + 1 \\ &= cn + 1. \end{aligned}$$

But we are suppose to prove that $T(n) \leq cn$. Instead of assume $T(n) \leq cn$, we can assume that $T(n) \leq cn - d$ for some constant c and $d \geq 1$. Then

$$\begin{aligned} T(n) &\leq c(\lfloor n/2 \rfloor) - d + c(\lceil n/2 \rceil) - d + 1 \\ &\leq cn - d. \end{aligned}$$

So we have proved that $T(n) \leq cn - d$ for general n. (We omitted the proof of the base case). So we have $T(n) = O(n)$.

As another example, consider the recurrence

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n.$$

To simplify the proof, we can do some variable changing.
Renaming $m = \lg n$. Then we have

$$T(2^m) = 2T(2^{m-1}) + m.$$

Further we renaming $S(m) = T(2^m)$, then it becomes

$$S(m) = 2S(m/2) + m.$$

We already know that in this case $S(m) = O(m \lg m)$. So we have $T(n) = O(m \lg m) = O(\lg n \lg \lg n)$.

The recursion-tree method

- The recursion-tree are usually used to give a good guess of the recurrence.
- In a recursion tree, each node represents the costs of a single subproblem some where in the set of recursion function invocations.
- Sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of the recursion.

Suppose we want to guess the recurrence $T(n) = 3T(n/4) + \Theta(n^2)$.

- Create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$ for some constant c. We assume that n is a power of 4 for simplicity.
- In the first level, the cost is cn^2, the second level is $3T(n/4)$ which costs $3 \cdot c(\frac{n}{4})^2 = \frac{3}{16}cn^2$, the third level is $3^2T(\frac{n}{4^2})$ which costs $3^2 \cdot c(\frac{n}{4^2})^2 = (\frac{3}{16})^2cn^2, \ldots$. In general, the ith level costs $(\frac{3}{16})^{i-1}cn^2$.
- There are total $\log_4 n + 1$ levels and the last level is $3^{\log_4 n}T(1)$ which is $\Theta(n^{\log_4 3})$.

The total cost is

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

Since

$$\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i < \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i = \frac{1}{1-(3/16)} = \frac{16}{13},$$

We obtain

$$T(n) < \frac{16}{13}cn^2 + \Theta(n^{\log_4 3}) = O(n^2).$$

- We can see that $O(n^2)$ is the best possible, because $T(n) = 3T(n/4) + \Theta(n^2) \geq \Theta(n^2)$.
- Now we can use substitution method to prove that our guess is correct. So we want to prove that $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ is $O(n^2)$.
- Assume that the claim is true, that is, for all $m < n$, $T(m) \leq dm^2$ for some constant d. Then for n, we have

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn2 \\ &\leq 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\leq dn^2 \end{aligned}$$

The last step hold as long as $d \geq (16/13)c$.

As second example, we use the recursion-tree to guess the recurrence

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

- Let $T(n) = T(n/3) + T(2n/3) + cn$ for some constant c.
- At the top level of the recursion-tree, the cost is cn and the second level is $T(n/3) + T(2n/3)$ which costs $cn/3 + c2n/3 = cn$.
- In general, the cost of level i is cn if $i < \log_3 n$.
- For $i > \log_3 n$, the level cost less than cn.
- The height of the tree is $\log_{3/2} n$. So there are at most $2^{\log_{3/2} n}$ leaves. The cost of leaves is $\Theta(n^{\log_{3/2} 2})$.
- The total costs will be less than $cn \log_{3/2} n + \Theta(n^{\log_{3/2} 2})$. Since $1 < \log_{3/2} 2 < 2$ and we just want to guess the upper bound of $T(n)$, we may guess that $T(n) = O(n \lg n)$.

We can use substitution method to prove the correctness of the guess.

Assume that $T(m) \leq dm \lg m$ for any $m < n$. For n we have

$$\begin{aligned}
T(n) &\leq T(n/3) + T(2n/3) + cn \\
&\leq d(n/3)\lg(n/3) + 2d(n/3)\lg(2n/3) + cn \\
&= d(n/3)\lg n - d(n/3)\lg 3 + 2d(n/3)\lg n - 2d(n/3)\lg(3/2) + cn \\
&= dn\lg n - dn\lg 3 + 2d(n/3)\lg 2 + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\leq dn\lg n
\end{aligned}$$

The last step hold as long as $d \geq c/(\lg 3 - 2/3)$. This example shows that sometimes we do not need to calculate the exact value of the recursion-tree to get a correct guess.

The master method

Master Theorem Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, Then $T(n) = (n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

- In the above theorem, n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.
- Note that the theorem does not give a solution for all cases of the recurrences with the form $T(n) = aT(n/b) + f(n)$ because for some cases, we cannot find the constant ϵ for either case 1 or case 3, and it also does not fit case 2.
- The main idea for the proof of the Master theorem is calculating the exact cost of the the recursion tree. We omitted the details of the proof.

Examples

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

For this recurrence, $a = 1, b = 3, f(n) = n$, so we have
$n^{\log_b a} = n^{\log_3 9} = n^2$. Since $f(n) = n = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$,
we have the solution $T(n) = \Theta(n^2)$.

$T(n) = T(2n/3) + 1.$

In this case, $a = 1, b = 3/2, f(n) = 1$, and
$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Since
$f(n) = 1 = \Theta(n^{\log_b a}), T(n) = \Theta(\lg n)$.

$T(n) = 3T(n/4) + n \lg n$.

We have $a = 3, b = 4, f(n) = n \lg n$, and $n^{\log_b a} = n^4 3 = O(n^{0.793})$.
Since $f(n) = \Omega(n^{log_4 3+\epsilon})$, where $\epsilon > 0.2$, case 3 applies if we can find c such that $af(n/b) \leq cf(n)$.
$af(n/b) = (3/4)n \log(n/4) \leq (3/4)n \log n = cf(n)$ for $c = 3/4$. So $T(n) = \Theta(n \lg n)$.

FIND-MAXIMUM-SUBARRAY: $T(n) = 2T(n/2) + \Theta(n)$

We have $a = 2, b = 2, f(n) = \Theta(n)$, and $n^{\log_b a} = n^{\log_2 2} = n$. So $T(n) = \Theta(n \lg n)$ by the case 2 of the theorem.

Matrices Multiplication: $T(n) = 8T(n/2) + \Theta(n^2)$

We have $a = 8, b = 2, f(n) = \Theta(n^2)$, and $n^{\log_b a} = n^{\log_2 8} = n^3$.
Since $f(n) = O(n^{3-\epsilon})$, where $\epsilon = 1$, $T(n) = \Theta(n^3)$.

Strassen's method: $T(n) = 7T(n/2) + \Theta(n^2)$

We have $a = 7, b = 2, f(n) = \Theta(n^2)$, and $n^{\log_b a} = n^{\log_2 7}$. Since
$f(n) = O(n^{3-\epsilon})$, where $\epsilon = 0.8$, $(2.80 < \lg 7 < 0.81)$,
$T(n) = \Theta(n^{\lg 7})$.

Now we give an example to explain that in some cases, the Master method dose not work.

$T(n) = 2T(n/2) + n \lg n$.

In this cases, $a = 2, b = 2$, and $f(n) = n \lg n$. Now $n^{\log_b a} = n^{\log_2 2} = n < n \lg n$, we may guess the case 1 of the Theorem can be used. However, we cannot find $\epsilon > 0$ that satisfying $f(n) = n \lg n \geq dn^{1+\epsilon}$ for any constant d. This is because for any $\epsilon > 0$, we always can find some large n such that $\lg n < n^{\epsilon}$.

Quicksort

Quicksort algorithm applies the devide-and-conquer method. The procedure sort a subarray $A[p, \ldots, r]$, which can be described as follows.

Divide: Partition the array $A[p, \ldots, r]$ into two (possibly empty) subarray $A[p, \ldots, q-1]$ and $A[q, \ldots, r]$ such that each element of $A[p, \ldots, q-1]$ is less than or equal to $A[q]$ and $A[q]$ is less than or equal to each element of $A[q+1, \ldots, r]$. So $A[q]$ is at the correct position after this step.

Conquer: Sort the two subarrays $A[p, \ldots, q-1]$ and $A[q+1, \ldots, r]$ by recursive calls to quicksort.

Combine: Because the subarrays are already sorted, no work is needed to combine them. The entire array $A[p, \ldots, r]$ is sorted.

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q − 1)
5:         QUICKSORT(A, q + 1, r)
6:     end if
7: end procedure
```

```
1:  procedure PARTITION(A, p, r)
2:      x = A[r]
3:      i = p − 1
4:      for j = p to r − 1 do
5:          if A[j] ≤ x then
6:              i = i + 1
7:              exchange A[i] with A[j]
8:          end if
9:      end for
10:     exchange A[i + 1] with A[r]
11:     return i + 1
12: end procedure
```

The running time for PARTITION is $\Theta(n)$, because in each step of the loop, at most one swap is performed.

- The running time for quicksort depends on weather the partitions are balanced or not.
- The ideal partition is splitting at the middle and the worst case is splitting by 1 and $n-1$.
- Since the running time for partition is $\Theta(n)$, the worst case of the running time for quicksort is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n). \end{aligned}$$

To see the running time for the worst case, we consider the general case:

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n).$$

We guess that $T(n) \le cn^2$ for some constant c. Then

$$\begin{aligned} T(n) &= \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) \end{aligned}$$

The expression $q^2 + (n - q - 1)^2$ has positive second derivative with respect to q. Therefore the expression achieves the maximum value at either endpoint. So
$\max_{0 \le q \le n-1}(q^2 + (n - q - 1)^2) \le (n - 1)^2 = n^2 - 2n + 1$. We have

$$\begin{aligned} T(n) &= c(n^2 - 2n + 1) + \Theta(n) \\ &\le cn^2, \end{aligned}$$

if we pick the constant c large enough so that $c(2n - 1)$ greater than $\Theta(n)$. The above arguments shows $T(n) = O(n^2)$. On the other hand, it shows the case that has a solution of $T(n) = \Omega(n^2)$. That proves that in the worst case, $T(n) = \Theta(n^2)$.

In the best case of quicksort,

$$T(n) \quad = \quad 2T(n/2) + \Theta(n).$$

From master theorem, we have $T(n) = \Theta(n \lg n)$.