Data Structure Lecture II: Divide and Conquer

Part1. Recurrences Complexity Analyses

> ✍️ Three methods to analyze the complexity of recursive algorithm:
>
> 1. Substitution method(数学归纳法)
> 2. Recursive-Tree method
> 3. Master Method(大师法，有使用条件)
>
>   The master method comes from the ituition of Recursive-Tree

1.Substitution method

- step1:**Guess** the form of the solution
- step2:**Verify** by induction
- step3:**Solve** for constants

e.g1:

$$T(n) = 4T(\frac{n}{2}) + n$$

1. **First Guess:** $O(n^3)$ **:**

   Base case:

   $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant

   Induction:

   Assume that $T(k) \leq ck^3$ for $k < n$, we need to prove that $T(n) \leq cn^3$

$$\begin{aligned} T(n) \quad & = 4T(\frac{n}{2}) + n \\ & \leq 4c(\frac{n}{2})^3 + n \\ & = cn^3 - (\frac{1}{2}cn^3 - n) \\ & \leq cn^3 \end{aligned}$$

   Notice that when $\frac{1}{2}cn^3 - n \geq 0$, the above nonequality is true.

   We can set $c \geq 2$, and $n \geq 1$ (Solve the constants)

2. **Second Guess:** $O(n^2)$:

if we model the above method

> *Base case:*
>
> $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant
>
> *Induction:*
>
> Assume that $T(k) \le ck^2$ for $k < n$, we need to prove that $T(n) \le cn^2$
>
> $$\begin{aligned} T(n) \quad &= 4T(\frac{n}{2}) + n \\ &\le 4c(\frac{n}{2})^2 + n \\ &= cn^2 + n \end{aligned}$$
>
> We find that we can' t prove $T(n) \le cn^2$. We must make some changes.

Base case:

$T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant

Induction:

Assume that $T(k) \le c_1k^2 - c_2k$ for $k < n$, we need to prove that $T(n) \le c_1n^2 - c_2n$

$$\begin{aligned} T(n) \quad &= 4T(\frac{n}{2}) + n \\ &\le 4c_1(\frac{n}{2})^2 - 2c_2n + n \\ &= c_1n^2 - c_2n - (c_2n - n) \\ &\le c_1n^2 \end{aligned}$$

Notice that when $c_2n - n \ge 0$, the above nonequality is true.

We can set $c_2 \ge 1$ (Solve the constants)

e.g2:

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

注意：向下取整拿掉无伤大雅...

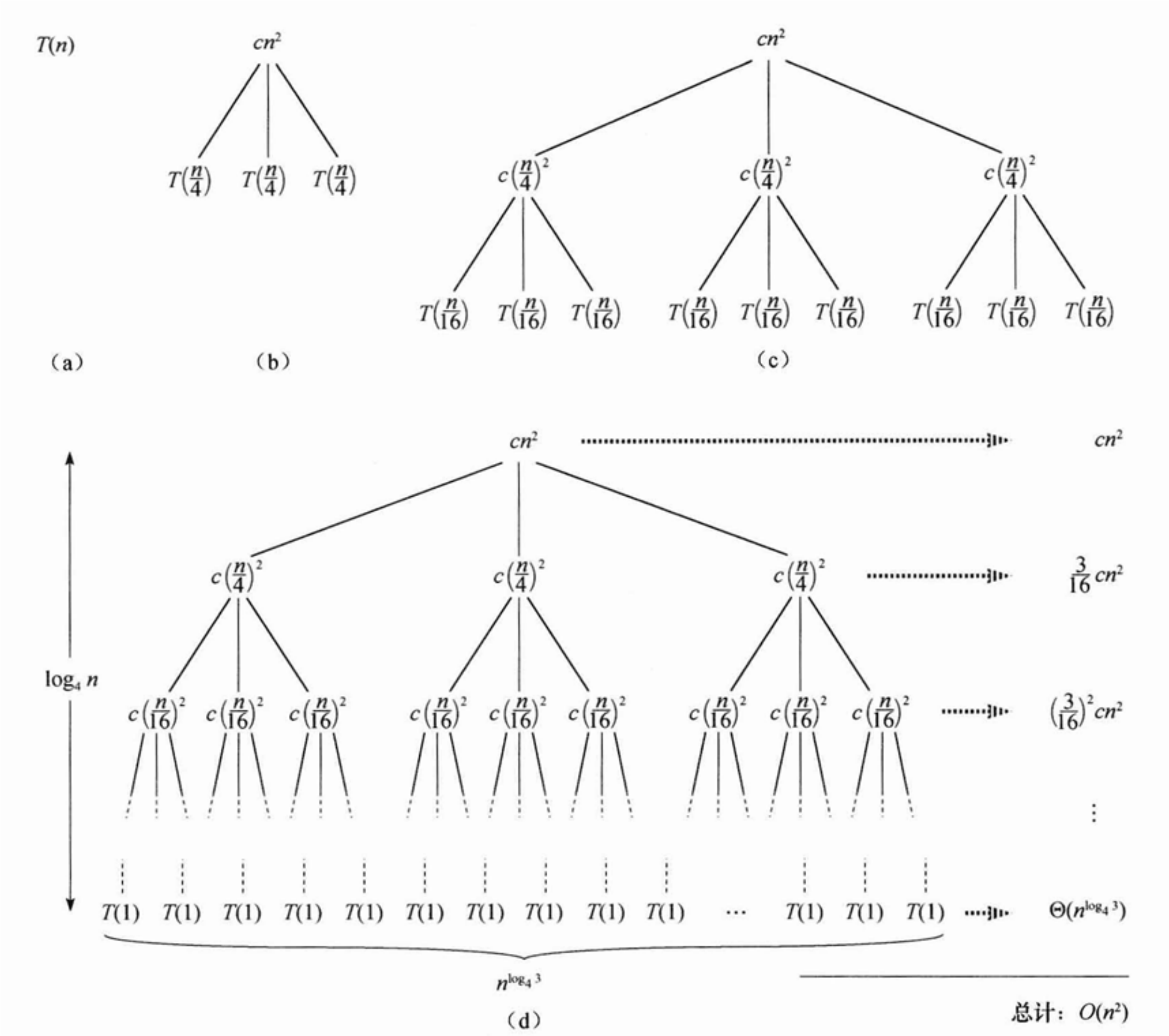
我们用一种巧妙的指对变换来求解这个问题

- Renaming $m = lgn$ yields
  $T(2^m) = 2T(2^{m/2}) + m$
- Rename $S(m) = T(2^m)$ to produce the new recurrence
  $S(m) = 2S(m/2) + m$
  $S(m) = O(m\ lgm)$
- Changing back from $S(m)$ to $T(n)$
  $T(n) = T(2^m) = S(m) = O(m\ lgm) = O(lgn\ lglgn)$

2.Recursive-tree method

We use an example to show the idea of recursive tree

$$T(n) = 3T(\lfloor\frac{n}{4}\rfloor) + n^2$$



$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{3}{16})^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

We can compute each layers' complexitites to divide and conquer the question into smaller questions, adding the last layer's cost.

注意我们在求和过程里面的等比数列是收敛的，这对于我们求解这个问题非常重要。

说白了，求递归（分治算法）的时间复杂度，就是 `each layers' complexitites to divide and conquer` 和 `last layer's cost` 两方面力量的较量，谁大就取谁。

3.Master method

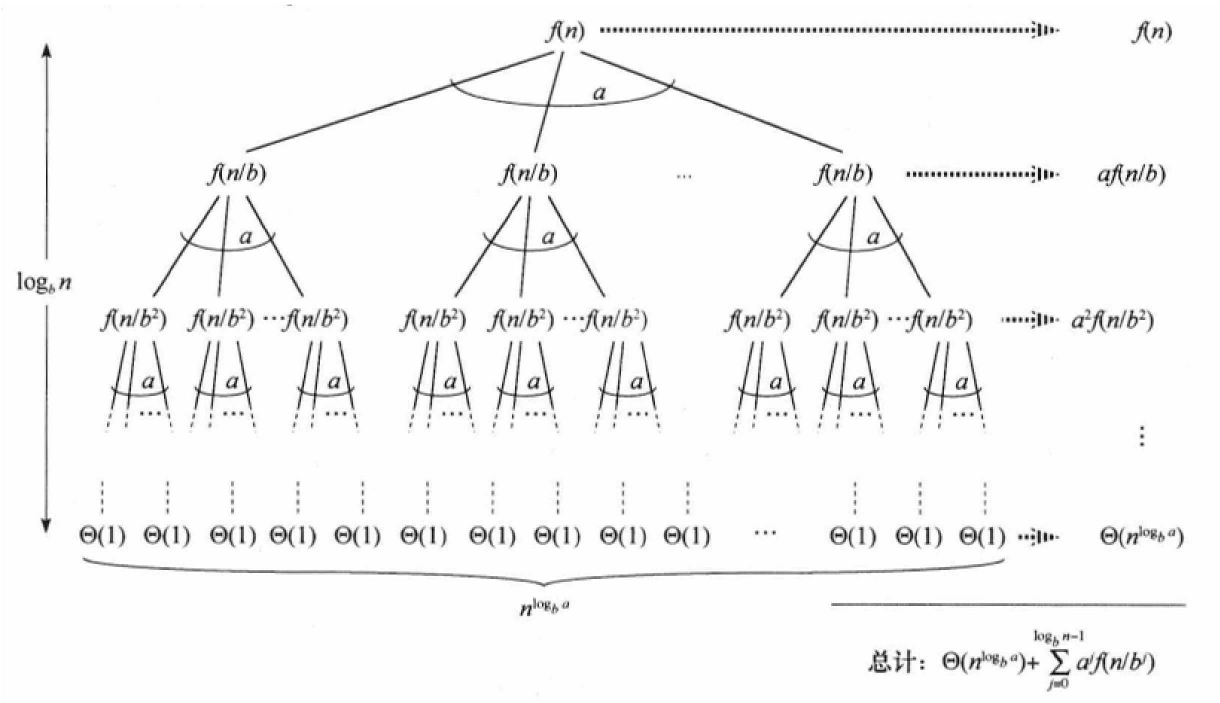
With the intuition coming from recursive tree above, we can introduce the master method.

3.1.Master method

We assume the question has the following form:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Where $a \geq 1$, $b > 1$, and f is asymptotically(渐进) positive.



In the last layer, we have cost of $\Theta(n^{log_b^a})$, and each layer we have the complexity of $f(x)$ to divide and conquer.

We can compare $f(n)$ with $n^{log_b^a}$

1. $f(n) = O(n^{log_b^a - \epsilon})$ for some constant $\epsilon > 0$.

   $f(n)$ grows **polynomially slower** than $n^{log_b^a}$ (by an n^ϵ factor)

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

2. $f(n)$ and $n^{log_b^a}$ grow at similar rates

   Solution: $T(n) = \Theta(n^{log_b^a} lg(n))$

3. $f(n) = \Omega(n^{log_b^a + \epsilon})$ for some constant $\epsilon > 0$.

   $f(n)$ grows **polynomially faster** than $n^{log_b^a}$ (by an n^ϵ factor) and $f(n)$ satisfies the **regularity condition** that $af(\frac{n}{b}) \leq c(f(n))$ for some constant $c < 1$ (保证求和数列能够收敛)

Solution: $T(n) = \Theta(f(n))$

Proof

$$
\begin{aligned}
g(n) &= \sum_{j=0}^{\log_b n-1} a^j f(n/b^j) \\
&\leq \sum_{j=0}^{\log_b n-1} c^j f(n) \quad \text{if } af(n/b) \leq cf(n) \\
&\leq f(n) \sum_{j=0}^{\log_b n-1} c^j \\
&= f(n)(\frac{1}{1-c}) \\
&= O(f(n))
\end{aligned}
$$

3.2 Example of Master method

EX.1 $T(n) = 4T(\frac{n}{2}) + n$

$$a = 4, b = 2, n^{log_b^a} = n^2$$

$$f(n) = n \implies T(n) = \Theta(n^2)$$

EX.2 $T(n) = 4T(\frac{n}{2}) + n^2$

$$a = 4, b = 2, n^{log_b^a} = n^2$$

$$f(n) = n^2 \implies T(n) = \Theta(n^2 lg(n))$$

EX.3 $T(n) = 4T(\frac{n}{2}) + n^3$

$$a = 4, b = 2, n^{log_b^a} = n^2$$

$$f(n) = n^3 \implies T(n) = \Theta(n^3)$$

EX.4 $T(n) = 4T(\frac{n}{2}) + \frac{n^2}{lg(n)}$

$$a = 4, b = 2, n^{log_b^a} = n^2$$

$f(n) = \frac{n^2}{lg(n)} \implies f(n)$ doesn't grow **polynomially slower** than $n^{log_b^a}$

$\implies$ Master method does not apply

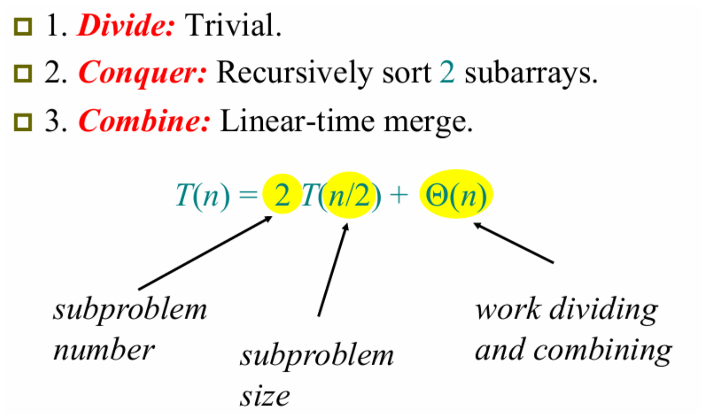
Part2. Divide and Conquer Algorithm

1.Divide and Conquer design paradigm

1. Divide the problem(instance) into subproblems
2. Conquer the subproblems by solving them recursively.
3. Combine subproblem solutions

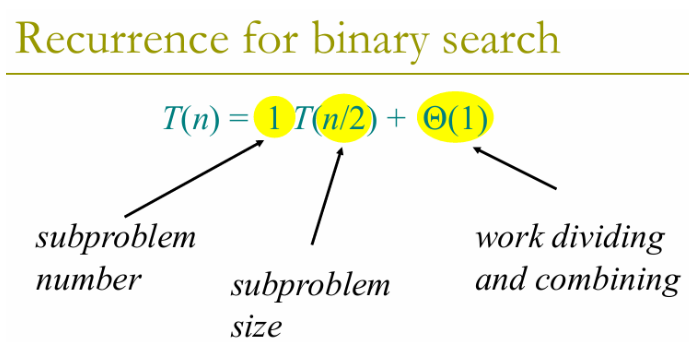
2.Divide and Conquer Examples:

2.1 Merge Sort:



$$a = 2, b = 2, f(n) = n, n^{log_b^a} = n \implies \Theta(nlg(n))$$

2.2 Binary Search:



$$a = 1, b = 2, f(n) = n, n^{log_b^a} = 1 \implies \Theta(lg(n))$$

2.3 Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(lgn)$$

2.4 Fibonacci numbers(Recursive squaring)

Fibonacci numbers:

$F(0) = 0, F(1) = 1 \quad F(x) = F(x-1) + F(x-2), x \geq 2$

- algorithm1: Naive recursive algorithms:

Naive recursive algorithm: $\Omega(\phi^n)$

(exponential time), where $\phi = (1+\sqrt{5})/2$

is the ***golden ratio***.

- algorithm2:Recursive squaring:

  We can conclude that:

$$\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$\begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$
$$\Longrightarrow \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$$

Then we can **see the powering of the matrix** $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ **as the powering of a number**, similar to 2.3.

$$\Longrightarrow \Theta(lg(n))$$

- algorithm3:dynamic programming:

$$\Longrightarrow \Theta(n)$$

2.5 Matrix multiplication

Input: $A = [a_{ij}],\ B = [b_{ij}].$
Output: $C = [c_{ij}] = A \cdot B.$ $\Big\}\ i, j = 1, 2, \dots, n.$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

- algorithm1: standard algorithm

for $i \leftarrow 1$ **to** n
  do for $j \leftarrow 1$ **to** n
    do $c_{ij} \leftarrow 0$
      for $k \leftarrow 1$ **to** n
        do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Running time = $\Theta(n^3)$

- algorithm2: naive divide and conquer algoritm

IDEA:
$n \times n$ matrix = 2×2 matrix of $(n/2) \times (n/2)$ submatrices:

$$\left[\begin{array}{c|c} r & s \\ \hline t & u \end{array}\right] = \left[\begin{array}{c|c} a & b \\ \hline c & d \end{array}\right] \cdot \left[\begin{array}{c|c} e & f \\ \hline g & h \end{array}\right]$$

$$\boldsymbol{C} \quad = \quad \boldsymbol{A} \quad \cdot \quad \boldsymbol{B}$$

$$\left.\begin{aligned} r &= ae + bg \\ s &= af + bh \\ t &= ce + dg \\ u &= cf + dh \end{aligned}\right\}$$

recursive
8 mults of $(n/2) \times (n/2)$ submatrices
4 adds of $(n/2) \times (n/2)$ submatrices

$$T(n) = 8\,T(n/2) + \Theta(n^2)$$

submatrices number *submatrices size* *work dividing submatrices*

$$n^{\log_b a} = n^{\log_2 8} = n^3 \Rightarrow \textbf{CASE 1} \Rightarrow T(n) = \Theta(n^3)$$

No better than the ordinary algorithm.

- algorithm3: Strassen's algorithm

- Multiply 2×2 matrices with only 7 recursive mults.

$P_1 = a \cdot (f - h)$

$P_2 = (a + b) \cdot h$

$P_3 = (c + d) \cdot e$

$P_4 = d \cdot (g - e)$

$P_5 = (a + d) \cdot (e + h)$

$P_6 = (b - d) \cdot (g + h)$

$P_7 = (a - c) \cdot (e + f)$

$r = P_5 + P_4 - P_2 + P_6$

$s = P_1 + P_2$

$t = P_3 + P_4$

$u = P_5 + P_1 - P_3 - P_7$

7 mults, 18 adds/subs.
Note: No reliance on commutativity of mult!

1. ***Divide:*** Partition A and B into $(n/2) \times (n/2)$ submatrices. Form terms to be multiplied using $+$ and $-$.
2. ***Conquer:*** Perform 7 multiplications of $(n/2) \times (n/2)$ submatrices recursively.
3. ***Combine:*** Form C using $+$ and $-$ on $(n/2) \times (n/2)$ submatrices.

$$T(n) = 7\,T(n/2) + \Theta(n^2)$$

$$n^{\log_b a} = n^{\log_2 7} = n^{2.81} \Rightarrow \textbf{CASE 1} \Rightarrow T(n) = \Theta(n^{\lg 7})$$

The number 2.81 may not seem much smaller than 3, but because the difference is in the exponent, the impact on running time is significant. In fact, Strassen's algorithm beats the ordinary algorithm on today's machines for $n \geq 32$ or so.

2.6 Chip Problem

4-5 （芯片检测） Diogenes 教授有 n 片可能完全一样的集成电路芯片，原理上可以用来相互检测。教授的测试夹具同时只能容纳两块芯片。当夹具装载上时，每块芯片都检测另一块，并报告它是好是坏。一块好的芯片总能准确报告另一块芯片的好坏，但教授不能信任坏芯片报告的结果。因此，4 种可能的测试结果如下：

芯片 A 的结果	芯片 B 的结果	结　论
B 是好的	A 是好的	两片都是好的，或都是坏的
B 是好的	A 是坏的	至少一块是坏的
B 是坏的	A 是好的	至少一块是坏的
B 是坏的	A 是坏的	至少一块是坏的

a. 证明：如果超过 $n/2$ 块芯片是坏的，使用任何基于这种逐对检测操作的策略，教授都不能确定哪些芯片是好的。假定坏芯片可以合谋欺骗教授。

b. 考虑从 n 块芯片中寻找一块好芯片的问题，假定超过 $n/2$ 块芯片是好的。证明：进行 $\lfloor n/2 \rfloor$ 次逐对检测足以将问题规模减半。

c. 假定超过 $n/2$ 块芯片是好的，证明：可以用 $\Theta(n)$ 次逐对检测找出好的芯片。给出描述检测次数的递归式，并求解它。

Problem c:

首要目标是**找到一个好芯片**， 这样我们就可以检测出其他芯片的好坏了。

我们可以把所有芯片两两配对（如果是奇数可能会剩一个芯片X）

如果配对的一对芯片检测结果是： `B坏A好` 或 `B好A坏` 或 `B坏A坏` ，那么我们就可以确定 `A` , `B` 里面至少有一个芯片损坏。

我们如果把这些芯片对都丢弃， 也不会影响好芯片对于坏芯片的格局

丢弃这些芯片对了以后，剩下检测结果为 `B好A好` 的芯片对，他们要么都是好芯片，要么都是坏芯片。如果是奇数个芯片，则还有芯片 `X` .我们假设 `B好A好` 的芯片对有 `m` 对好芯片, `n` 对坏芯片。

对于没有 `X` 的情况，丢弃每一对里面的一个就好，$2m > 2n \implies m > n$。

我们分类讨论一下有 `X` 的情况：

- 如果 `X` 是好芯片

  则有 $2m+1 > 2n \implies m > n - \frac{1}{2}$

  - 如果 $m+n$ 是奇数：

    m, n 异奇偶，可知 $m > n$

  - 如果 $m+n$ 是偶数：

    m, n 同奇偶，仅可知 $m \geq n$

    此时 `X` 确定是好的芯片， 我们丢弃每一对中的一个，**再加上X**(这样才能保证好芯片数还是多于坏芯片数, $m+1 > n$)， 得到数目基本为原来一半的芯片集合。问题规模减半。

- 如果 `X` 是坏芯片

  则有 $2m-1>2n \implies m>n+\frac{1}{2}$

  - 如果 $m+n$ 是奇数：

    m,n 异奇偶，可知 $m>n$

  - 如果 $m+n$ 是偶数：

    m,n 同奇偶，可知 $m \geq n+2$

    此时 `X` 确定是坏的芯片， 我们丢弃每一对中的一个，**再加上X** (这样还是能保证好芯片数还是多于坏芯片数, $m>n+1$)， 得到数目基本为原来一半的芯片集合。问题规模减半。

总结我们的算法：

代码块

```
while (! 找到好芯片) {
    丢弃所有检测结果是：“B坏A好”或 “B好A坏”或 “B坏A坏” 的芯片对
    if(没有X){
        丢弃剩下每个芯片对里面的一个
    }
    else{
        if(芯片对对数是奇数){
            丢弃剩下每个芯片对里面的一个，丢弃X
        }
        else{
            丢弃剩下每个芯片对里面的一个，保留X
        }
    }
}
```

考虑每次配对都有 $O(n)$ 的复杂度

我们可知：

$$T(n) \leq T(\frac{n}{2})+n$$

$$a=1,b=2,n^{log_b^a}=1,f(n)=n \implies T(n)=\Theta(n)$$