算法设计与分析

图南

2021/2

渐进分析与分治策略

接下来的课讲什么？

你将从中收获什么？

你将学会

- 算法的**设计**规范
  - 学会这些规范将帮助你设计各种问题的解决算法
- 严格的算法**分析**
  - 如何在保证正确性的同时让一个算法更高效？如何进行数学证明？
- 更好地**交流**技术
  - 我们如何描述一个算法？如何让别人理解我们的思路？

你还将学会

- 各种著名算法
- 技术面试的技巧
- 如何写伪代码

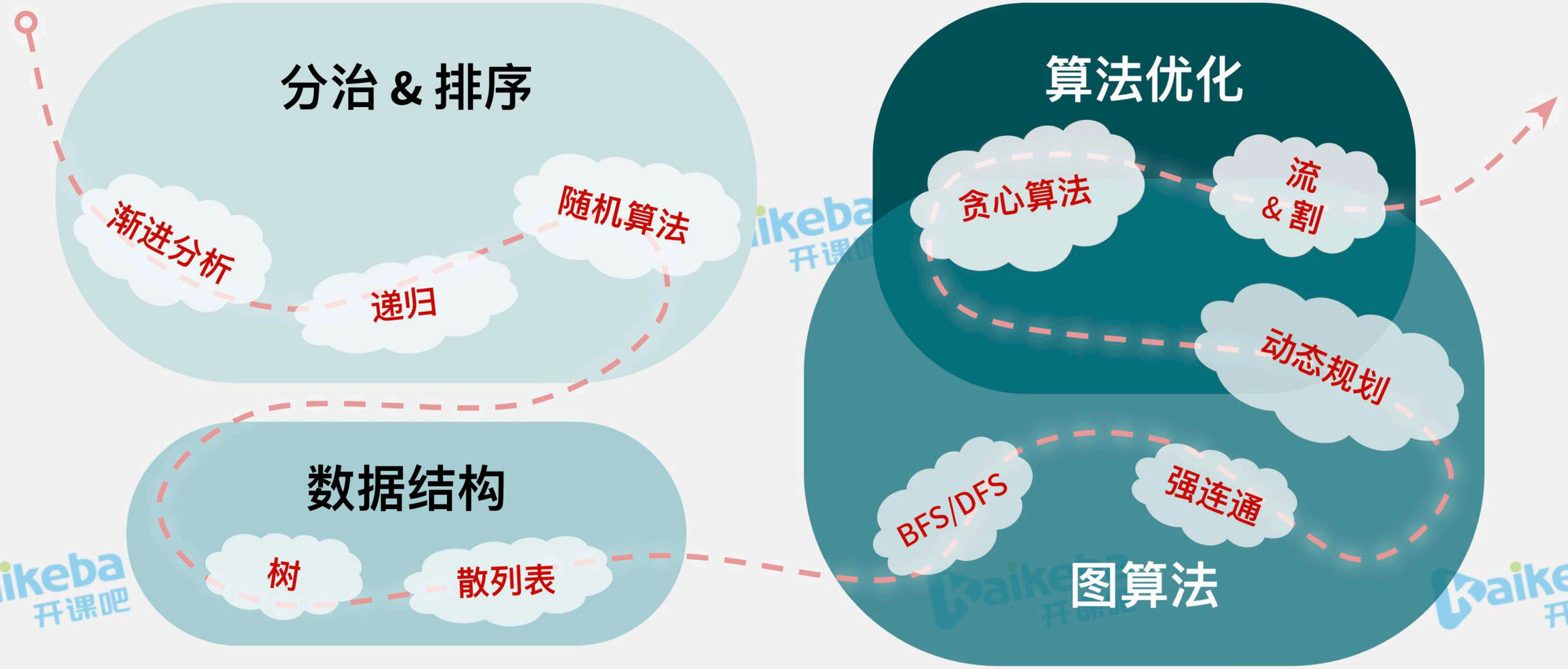
你还将学会

- 各种著名算法
- 技术面试的技巧
- 如何写伪代码

算法很有趣，

算法很重要!

我们的征途



乘法！

如何计算两个数的乘积?

乘法：问题定义

输入: 2 个非负的数, x 与 y (各有 n 位)

输出: 两数乘积 $x \cdot y$

$$
\begin{array}{r}
5678 \\
\times\ 1234 \\
\hline
7006652
\end{array}
$$

小学乘法

```
  45
x 63
----
 135
2700
----
2835
```

小学乘法

算法描述 (非正式*):

计算各个部分的乘积，并将这些积 (经过相应的位置移动) 求和

```
  45
x 63
----
 135
2700
----
2835
```

** 不要这样描述你的算法*

小学乘法

```
  45123456678093420581217332421
x 63782384198347750652091236423
--------------------------------
                              ):
```

小学乘法

n 位

```
  4512345667809342058121733242 1
x 6378238419834775065209123642 3
-------------------------------
                              ):
```

这个算法的效率如何?

(最坏情况需要多少次一位数操作?)

小学乘法

n 位

```
  4512345667809342058121733242 1
x 6378238419834775065209123642 3
```

)：

这个算法的效率如何?

(最坏情况需要多少次一位数操作?)

计算 n 个部分乘积: ~$2n^2$ 次操作 (至多 n 次一位数乘法 & n 次中间结果求和)

将 n 个部分乘积求和: ~$2n^2$ 次操作

(求和 & 进位)

小学乘法

n 位

```
  4512345667809342058121733242 1
x 6378238419834775065209123642 3
```

):

这个算法的效率如何?

(最坏情况需要多少次一位数操作?)

计算 n 个部分乘积: ~$2n^2$ 次操作 (至多 n 次一位数乘法 & n 次中间结果求和)

将 n 个部分乘积求和: ~$2n^2$ 次操作

(求和 & 进位)

最坏情况下需要 ~ $4n^2$ 次操作

小学乘法

n 位

4512345 17332421

x 63782 236423

):

这个算法的效率如何

(最坏情况需要多少次一位数操

问题来了...

如何才能做得更好?

乘积: ~$2n^2$ 次操作 (至多 n 次一法 & n 次中间结果求和)

部分乘积求和: ~$2n^2$ 次操作

(求和 & 进位)

最坏情况下需要 ~ $4n^2$ 次操作

如何定义“更好”

1000000n 比 $4n^2$ 更好吗?

$0.000001n^3$ 比 $4n^2$ 更好吗?

$3n^2$ 比 $4n^2$ 更好吗?

如何定义“更好”

$1000000n$ 比 $4n^2$ 更好吗?

$0.000001n^3$ 比 $4n^2$ 更好吗?

$3n^2$ 比 $4n^2$ 更好吗?

- **前两种情况取决于 n 的值...**
  - $1000000n < 4n^2$ 当且仅当 n 大于一定的值 (250000)
- **这里的常数系数没有一般性...**
  - 一次操作的速度可能与运算机器有关, 一台慢机器执行 $3n^2$ 次操作不一定比一台快机器执行 $4n^2$ 次操作快

如何定义“更好”

~~再一次~~隆重介绍...

渐进分析

如何定义“更好”

~~再一次~~隆重介绍...

渐进分析

- **一些指导性原则：**我们在乎的是运行时间/操作次数如何随着输入规模的变大而增加，我们需要一种与硬件、编程语言、内存等无关的评估方式。
  - 现实中，诸如硬件、编程语言、内存、编译器优化等等仍然是十分重要的因素。在学习算法分析的时候我们暂时忽略这些细节。

渐进分析

描述一个算法的渐进复杂度时我们使用

大-O 表示法

- 我们说小学乘法的 **“运行时间是 $O(n^2)$”** ← *“大-O n 方”或者“O n 方”**
  - 非正式地描述，运行时间像 n^2 一样增长
  - 稍后严格定义大-O表示法

** 表示非渐进紧确上界时使用小-o 表示法，由于分析复杂度的时候我们更关心渐进紧确界，口头表达“O”一般指大-O*

渐进分析

描述一个算法的渐进复杂度时我们使用

大-O 表示法

- 我们说小学乘法的 **“运行时间是 $O(n^2)$”** ← *“大-O n 方”或者“O n 方”**
  - 非正式地描述，运行时间像 n^2 一样增长
  - 稍后严格定义大-O表示法

渐进记号的意义

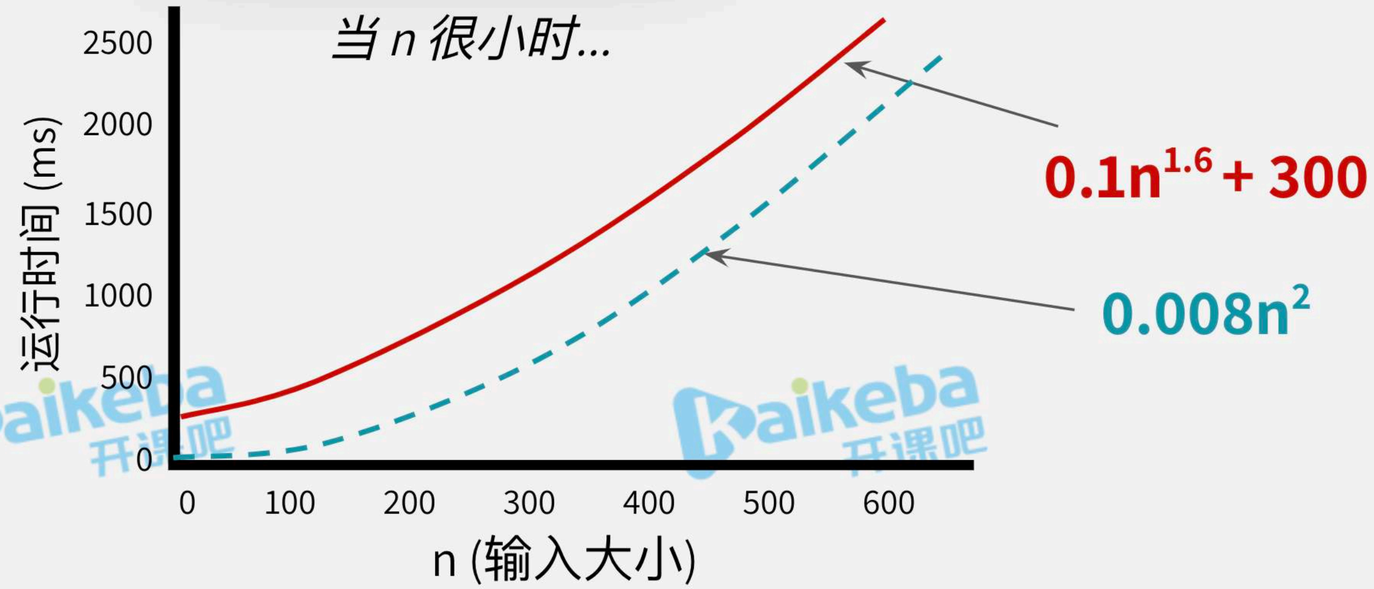
忽略 **常数项** *取决于运行环境* **和** **低阶项** *输入增大时不是主要增长因素*

渐进分析

渐进记号的意义

忽略 **常数项** **和** **低阶项**

取决于运行环境 *输入增大时不是主要增长因素*

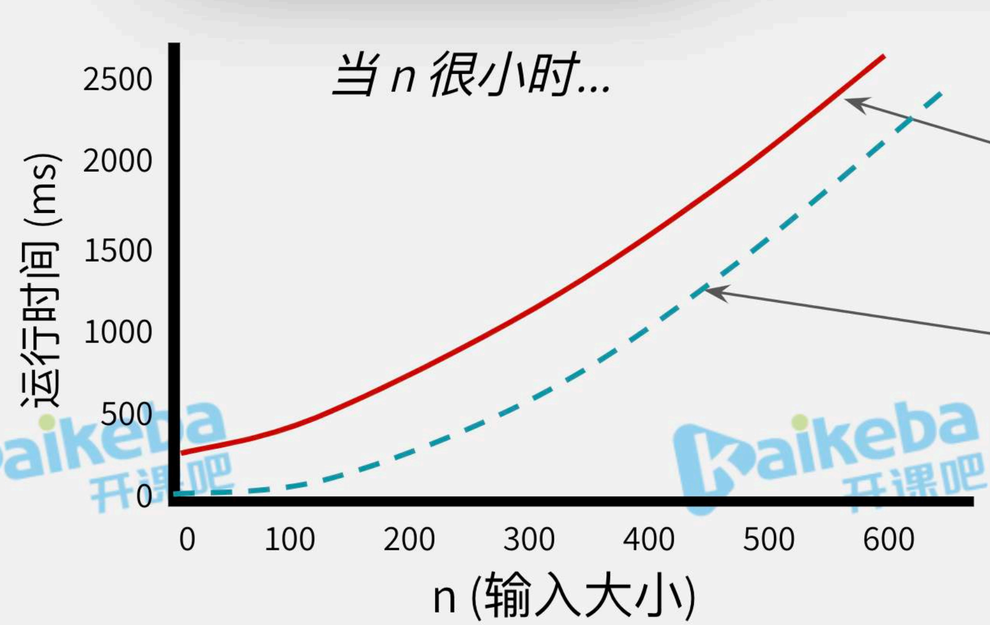


渐进分析

渐进记号的意义

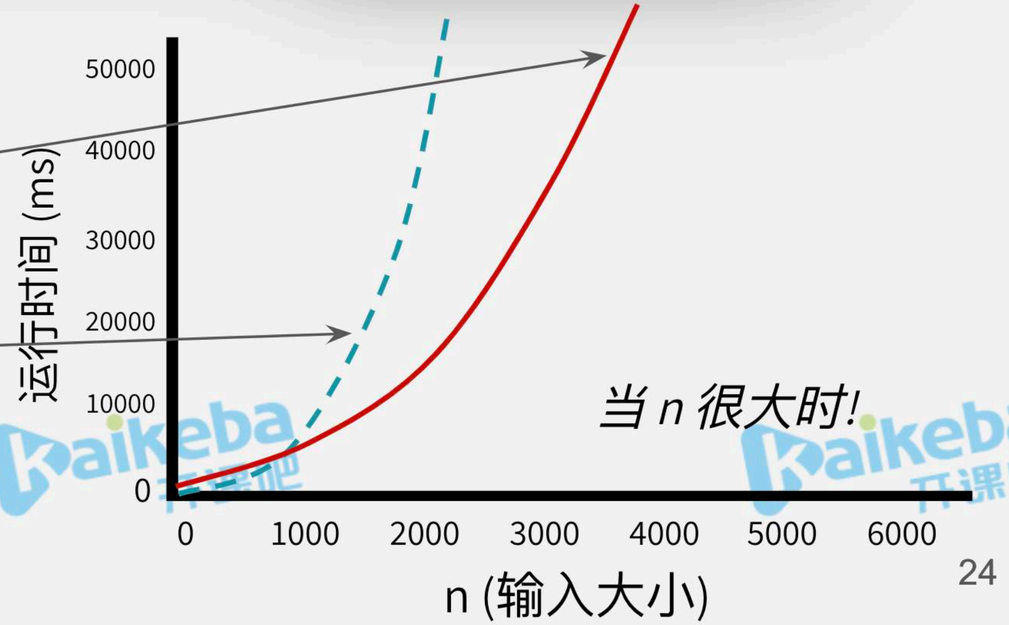
忽略 **常数项** **和** **低阶项**

取决于运行环境 *输入增大时不是主要增长因素*



$0.1n^{1.6} + 300$

$0.008n^2$

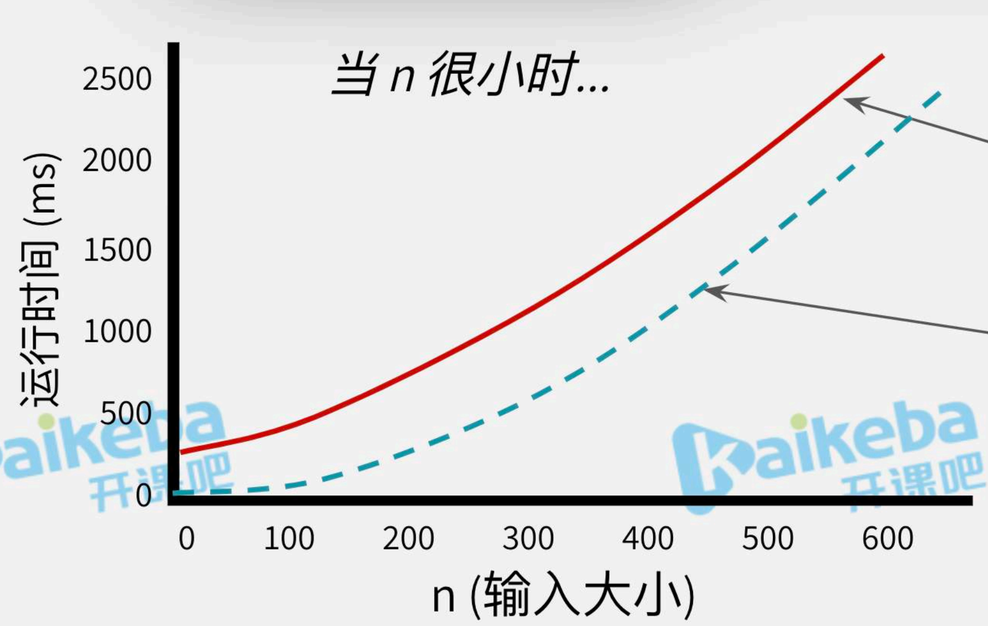


渐进分析

渐进记号的意义

忽略 **常数项** 和 **低阶项**

取决于运行环境 *输入增大时不是主要增长因素*

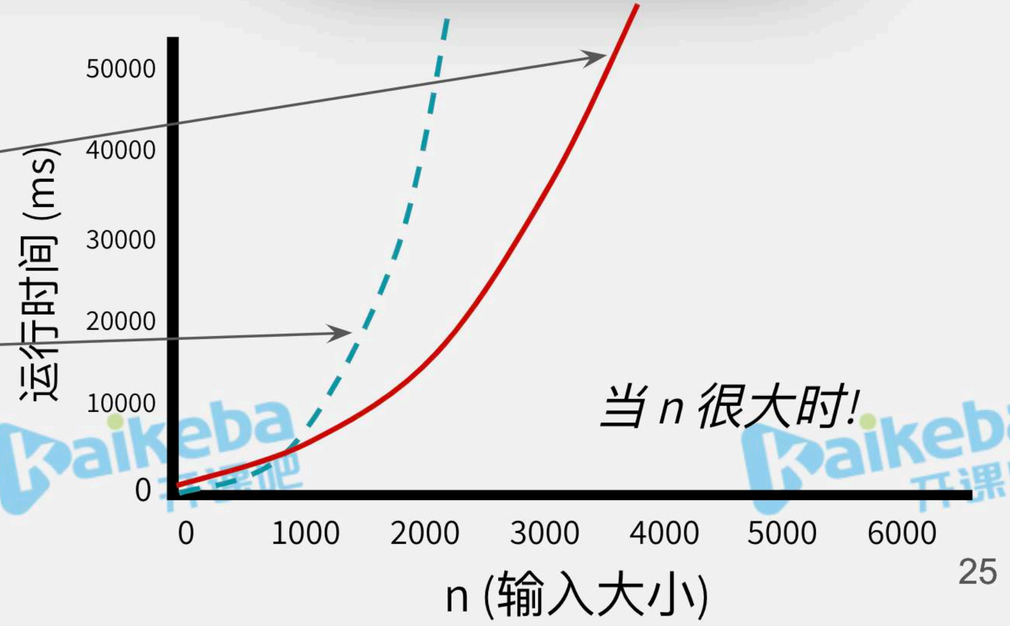


$O(n^{1.6})$

$0.1n^{1.6} + 300$

$0.008n^2$

$O(n^2)$



渐进分析

- 比较运行时间时，我们一般比较它们的 大-O 表达式
  - 例：$O(n^2)$ 的运行时间比 $O(n^3)$ 的运行时间“更好”
  - 例：$O(n^{1.6})$ 的运行时间比 $O(n^2)$ 的运行时间“更好”
  - 例：$O(1/n)$ 的运行时间比 $O(1)$ 的运行时间“更好”

渐进分析

- 比较运行时间时，我们一般比较它们的 大-O 表达式
  - 例：$O(n^2)$ 的运行时间比 $O(n^3)$ 的运行时间“更好”
  - 例：$O(n^{1.6})$ 的运行时间比 $O(n^2)$ 的运行时间“更好”
  - 例：$O(1/n)$ 的运行时间比 $O(1)$ 的运行时间“更好”

问题来了：

我们能不能比 $O(n^2)$ 更快地计算 n 位整数的乘法？

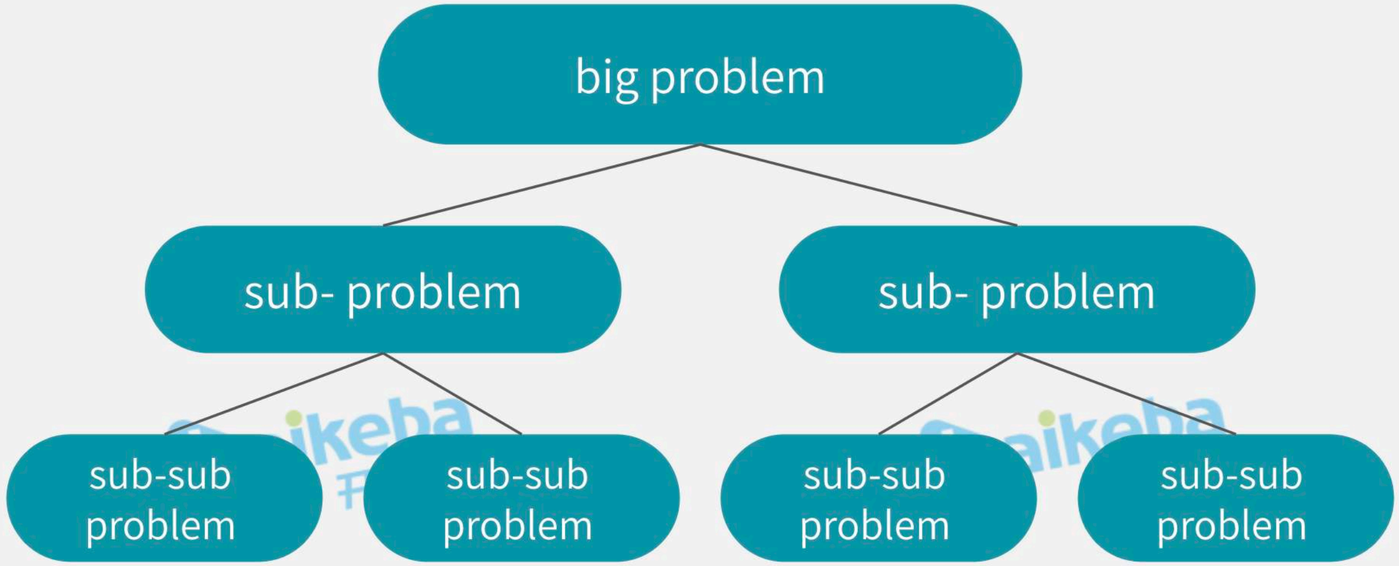
先思考这个问题，更多渐进分析和大-O表达式的内容稍后讨论

分治策略

我们的第一个算法设计范例

分治策略

- **分治算法 （Devide and Conquer）：**
  1. 分解(Devide)　将一个问题划分为一些子问题
  2. 解决(Conquer) *递归地* 解决这些子问题
  3. 合并(Combine) 将子问题的解组合成原问题的解



乘法的子问题

- **原问题:** 计算两个 n 位整数的乘积
- **如何划分子问题?**

乘法的子问题

- **原问题:** 计算两个 n 位整数的乘积
- **如何划分子问题?**

$$1234 \times 5678$$

$$= (12 \times 100 + 34) \times (56 \times 100 + 78)$$

$$= (12 \times 56)100^2 + (12 \times 78 + 34 \times 56)100 + (34 \times 78)$$

乘法的子问题

- **原问题:** 计算两个 n 位整数的乘积
- **如何划分子问题?**

$$1234 \times 5678$$

$$= (12 \times 100 + 34) \times (56 \times 100 + 78)$$

$$= (\underbrace{12 \times 56}_{1})100^2 + (\underbrace{12 \times 78}_{2} + \underbrace{34 \times 56}_{3})100 + (\underbrace{34 \times 78}_{4})$$

一个 4位数 乘法问题 → *四个 2位数 乘法子问题*

乘法的子问题

- **原问题:** 计算两个 n 位整数的乘积
- **如何划分子问题?** 更一般的...

$$\left[x_1 x_2 \cdots x_{n-1} x_n \right] \times \left[y_1 y_2 \cdots y_{n-1} y_n \right]$$

$$= (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$

$$= (\underbrace{a \times c}_{1}) 10^{n} + (\underbrace{a \times d + b \times c}_{2,3}) 10^{n/2} + (\underbrace{b \times d}_{4})$$

一个 n位数 问题 → *四个 (n/2)位数 子问题*

尝试写伪代码

MULTIPLY(x, y):

x & y 都是 n 位正整数

***注意**: 为了简化伪代码，这里我们假设 n 是 2 的幂*

尝试写伪代码

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y
```

x & y 都是 n 位正正整数

基本情况(base case): 我们甚至可以把九九乘法表存在内存里

注意: *为了简化伪代码，这里我们假设 n 是 2 的幂*

尝试写伪代码

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y

    write x as a·10^(n/2) + b
    write y as c·10^(n/2) + d
```

x & y 都是 n 位正整数

基本情况(base case): 我们甚至可以把九九乘法表存在内存里

a, b, c, d 都是 (n/2)位数

***注意**: 为了简化伪代码，这里我们假设 n 是 2 的幂*

尝试写伪代码

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y

    write x as a·10^(n/2) + b
    write y as c·10^(n/2) + d

    ac = MULTIPLY(a,c)
    ad = MULTIPLY(a,d)
    bc = MULTIPLY(b,c)
    bd = MULTIPLY(b,d)
```

x & y 都是 n 位正整数

基本情况(base case): 我们甚至可以把九九乘法表存在内存里

a, b, c, d 都是 (n/2)位数

递归调用 MULTIPLY 函数来求解子问题

注意: *为了简化伪代码，这里我们假设 n 是 2 的幂*

尝试写伪代码

x & y 都是 n 位正整数

```
MULTIPLY( x, y ):
    if (n = 1):
        return x·y

    write x as a·10^(n/2) + b
    write y as c·10^(n/2) + d

    ac = MULTIPLY(a,c)
    ad = MULTIPLY(a,d)
    bc = MULTIPLY(b,c)
    bd = MULTIPLY(b,d)

    return ac·10^n + (ad + bc)·10^(n/2) + bd
```

基本情况(base case): 我们甚至可以把九九乘法表存在内存里

a, b, c, d 都是 (n/2)位数

递归调用 MULTIPLY 函数来求解子问题

将子问题的解组合成最终的解!

注意: *为了简化伪代码，这里我们假设 n 是 2 的幂*

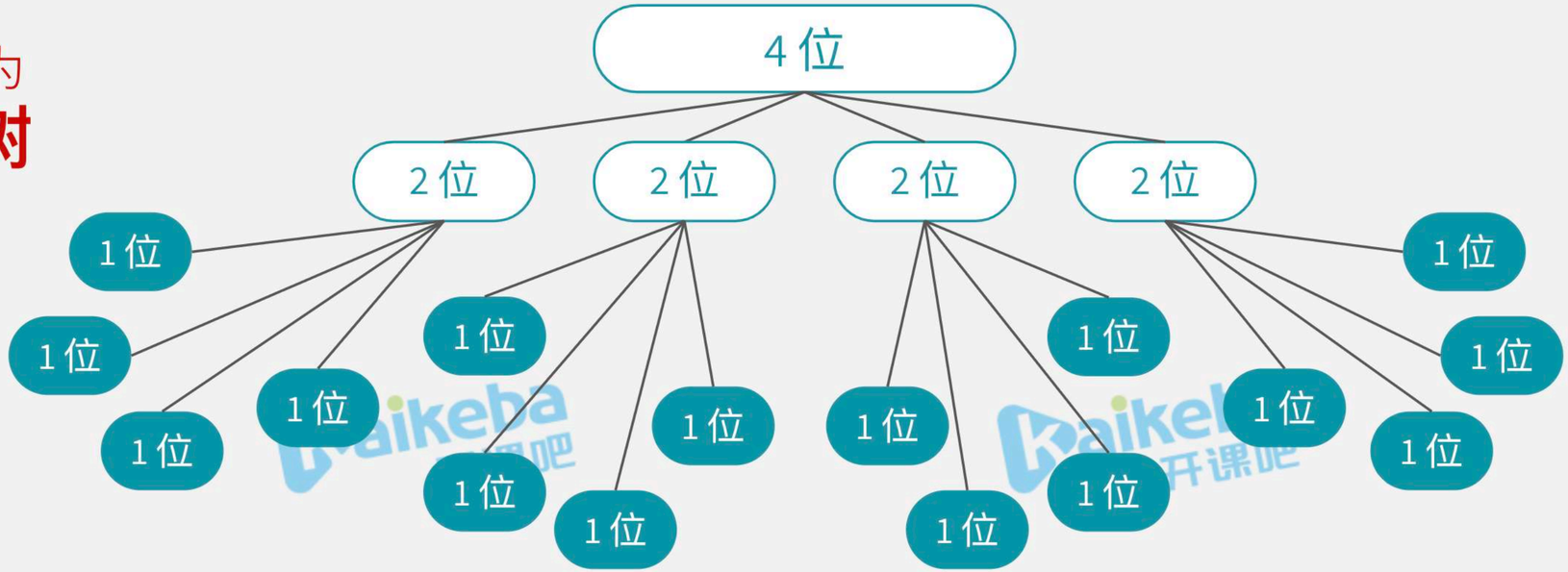
这个算法的性能如何?

- **先考虑小输入:** 如果用这个算法来计算两个4位数乘法，需要计算多少次1位数的乘积?
  - 即，我们会多少次触及到**基本情况**? 注意到只有在处理基本情况我们才会进行乘法计算
  - 这可以告诉我们所需操作数的下界

这个算法的性能如何?

- **先考虑小输入:** 如果用这个算法来计算两个4位数乘法， 需要计算多少次1位数的乘积?
  - 即，我们会多少次触及到**基本情况**? 注意到只有在处理基本情况我们才会进行乘法计算
  - 这可以告诉我们所需操作数的下界

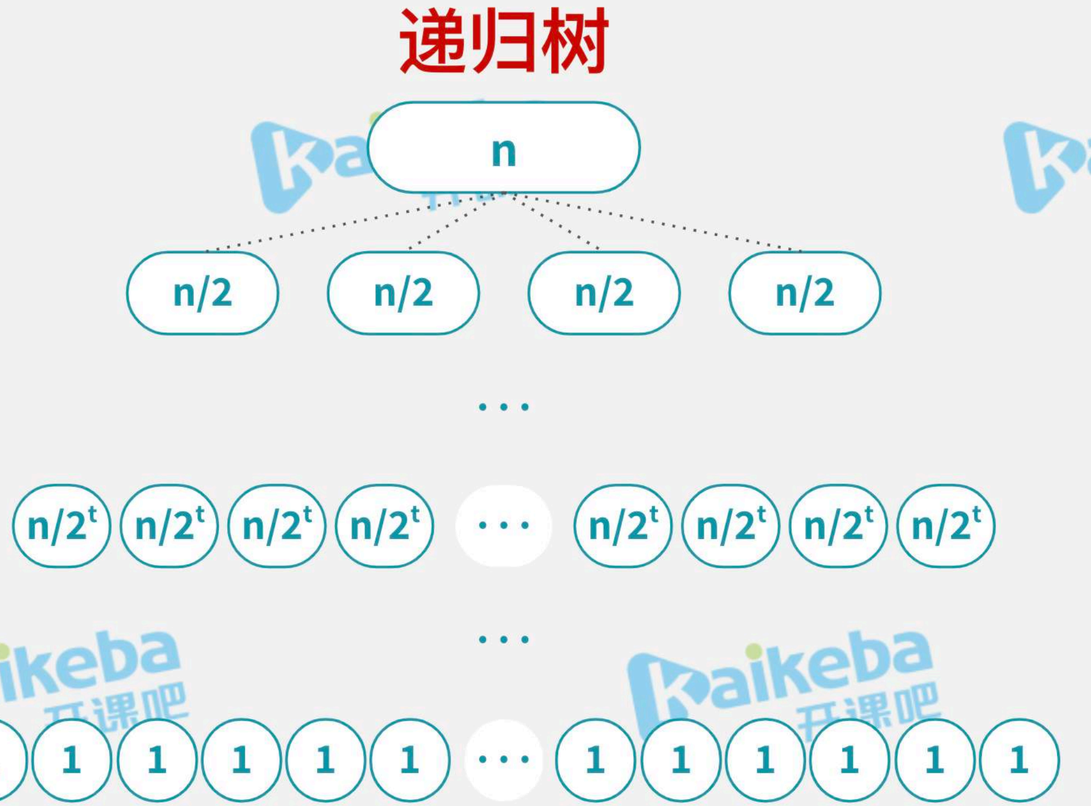
右图称为
递归树



十六次 1位 乘法计算

这个算法的性能如何?

- **更一般地:** 如果用这个算法来计算两个 n 位数乘法，需要计算多少次1位数的乘积?



Level 0: 1 个大小为 n 的问题

Level 1: 4^1 个大小为 n/2 的问题

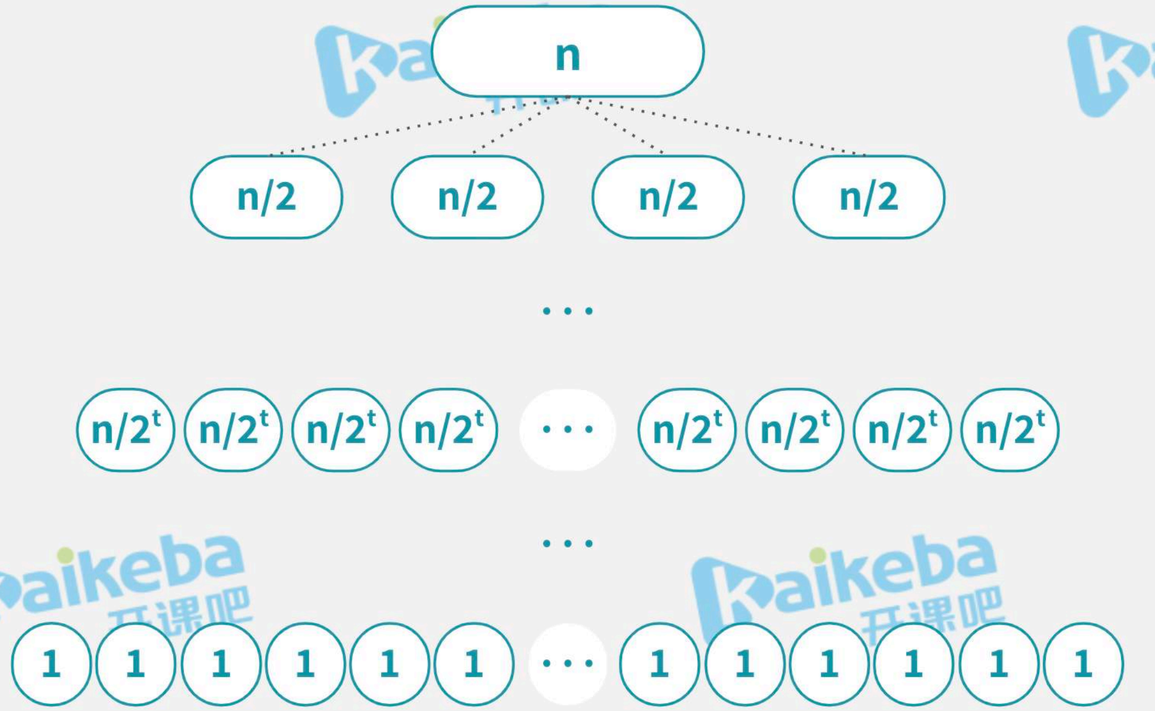
Level t: 4^t 个大小为 $n/2^t$ 的问题

Level $\log_2 n$: ____ 个大小为 1 的问题

这个算法的性能如何?

- **更一般地:** 如果用这个算法来计算两个 n 位数乘法，需要计算多少次1位数的乘积?

递归树



Level 0: 1 个大小为 n 的问题

Level 1: 4^1 个大小为 n/2 的问题

Level t: 4^t 个大小为 $n/2^t$ 的问题

Level $\log_2 n$: n^2 个大小为 1 的问题

$\log_2 n$ levels
(需要划分 $\log_2 n$ 次才能得到大小为 1 的子问题)

最后一层的问题数

$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

这个算法的性能如何?

使用分治策略的乘法算法所需运行时间至少是 $O(n^2)$!

递归树的最底层有 n^2 次操作，为什么我们还说运行时间**至少是** $O(n^2)$

这个算法的性能如何?

使用分治策略的乘法算法所需运行时间至少是 $O(n^2)$!

递归树的最底层有 n^2 次操作，为什么我们还说运行时间**至少是** $O(n^2)$

可是小学二次乘法就是 $O(n^2)$啊!

采用分治策略没有意义吗?

这个算法的性能如何?

使用分治策略的乘法算法所需运行时间至少是 $O(n^2)$!

递归树的最底层有 n^2 次操作，为什么我们还说运行时间**至少是** $O(n^2)$

可是小学二次乘法就是 $O(n^2)$啊!
采用分治策略没有意义吗?

Karatsuba says no!!!

KARATSUBA 算法

第一个比小学二次乘法渐进快速的乘法算法

更加明智地选择子问题

$$[x_1 x_2 \ldots x_{n-1} x_n] \times [y_1 y_2 \ldots y_{n-1} y_n]$$

$$= (a \times 10^{n/2} + b) \times (c \times 10^{n/2} + d)$$

$$= (a \times c)10^{n} + (a \times d + b \times c)10^{n/2} + (b \times d)$$

有效的子问题只需要提供:

ac $ad + bc$ bd

为了得到这三个值，之前的算法计算了4项: ac, ad, bc, & bd.

KARATSUBA的方法

$$\text{最终结果} = (\ ac\ )10^{n} + (\ ad + bc\ )10^{n/2} + (\ bd\ )$$

KARATSUBA的方法

最终结果 = (ac)10^n + (ad + bc)$10^{n/2}$ + (bd)

ac & bd 依然使用递归求解

ad + bc 等价于 (a+b)(c+d) - ac - bd

= (ac + ad + bc + bd) - ac - bd

= ad + bc

KARATSUBA的方法

最终结果 = (ac)10^n + (ad + bc)$10^{n/2}$ + (bd)

ac & bd 依然使用递归求解

ad + bc 等价于 (a+b)(c+d) - ac - bd

= (ac + ad + bc + bd) - ac - bd

= ad + bc

即, 可以通过一次递归调用来得到(a+b)(c+d)而不用两次递归调用来得到 ad & bc

三个子问题

所有需要的中间结果可以通过这三个子问题得到:

1. **ac**
2. **bd**
3. **(a+b)(c+d)**

(a+b) and (c+d) 均为n/2位数

⇩

子问题的大小依然为原问题的一半!

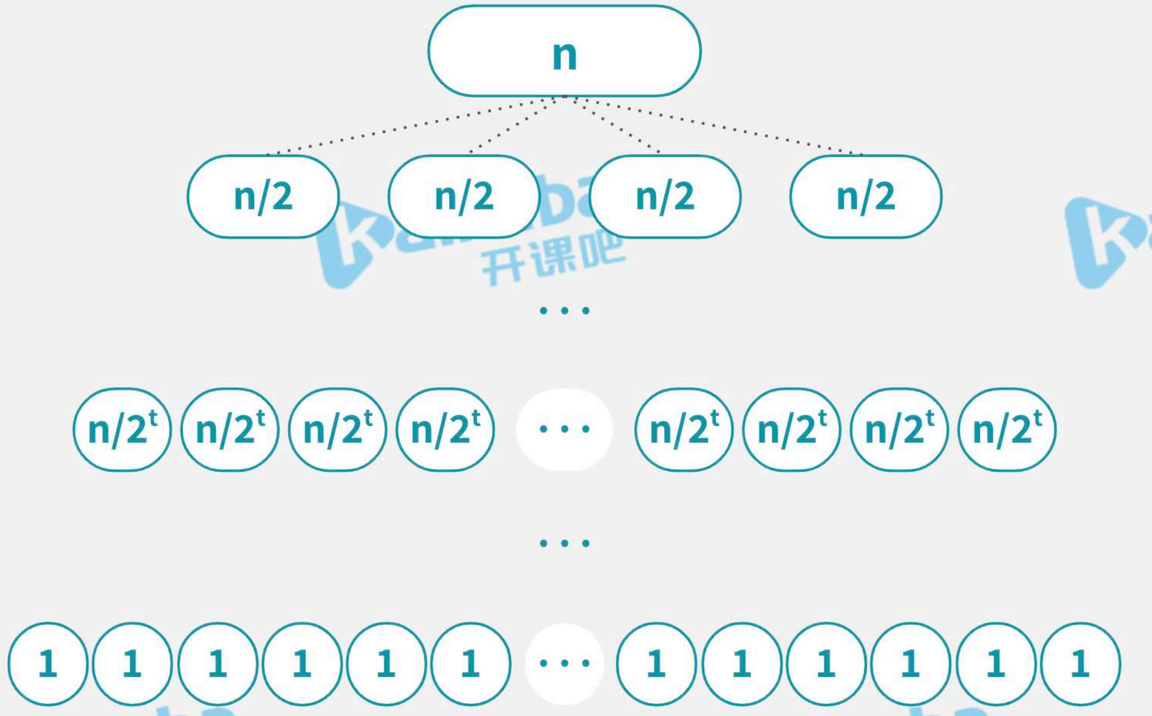
通过三个子问题的结果组合成原问题的结果:

$$(\ ac\ )10^{n} + (\ ad + bc\ )10^{n/2} + (\ bd\ )$$

1 ; 3 - 1 - 2 ; 2

算法运行时间如何?

先前分治算法的递归树和运行时间为:



Level 0: 1 个大小为 n 的问题

Level 1: 4^1 个大小为 n/2 的问题

Level t: 4^t 个大小为 $n/2^t$ 的问题

Level $\log_2 n$: ____ 个大小位 1 的问题

$\log_2 n$ levels
(需要划分 $\log_2 n$ 次才能得到大小为 1 的子问题)

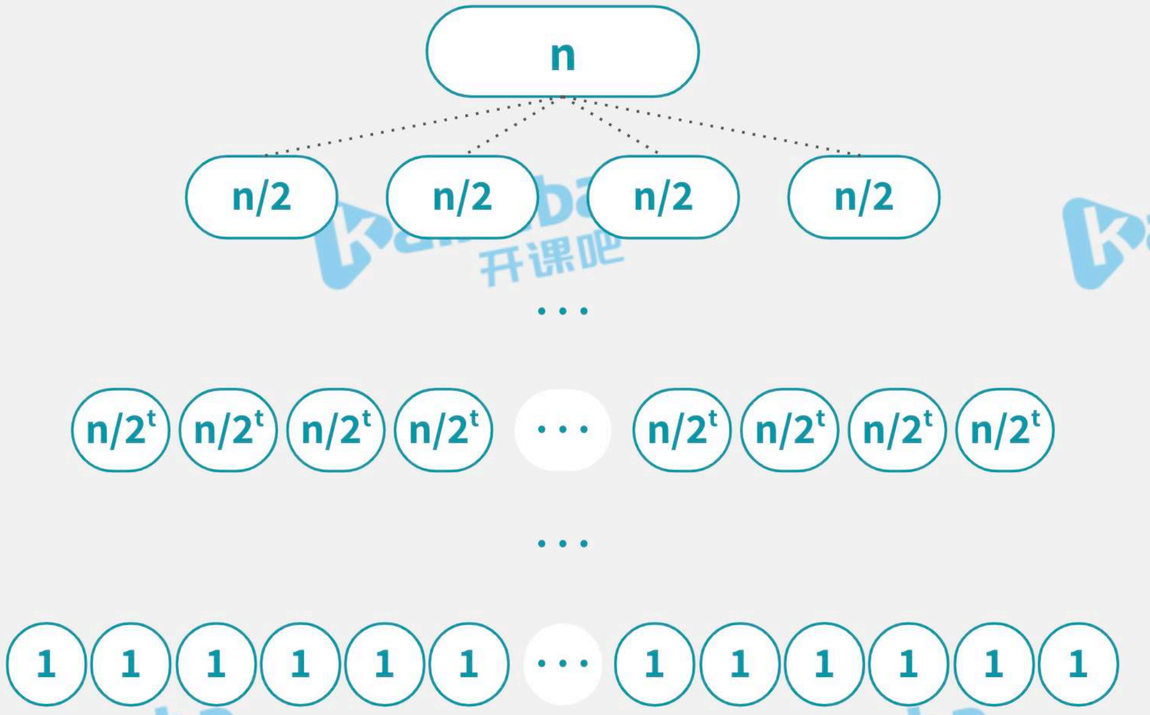
最后一层的问题数

$= 4^{\log_2 n} = n^{\log_2 4}$

$= \mathbf{n^2}$

算法运行时间如何?

先前分治算法的递归树和运行时间为:



Level 0: 1 个大小为 n 的问题

Level 1: 4^1 个大小为 n/2 的问题

Level t: 4^t 个大小为 $n/2^t$ 的问题

Level $\log_2 n$: $\underline{\quad n^2 \quad}$ 个大小位 1 的问题

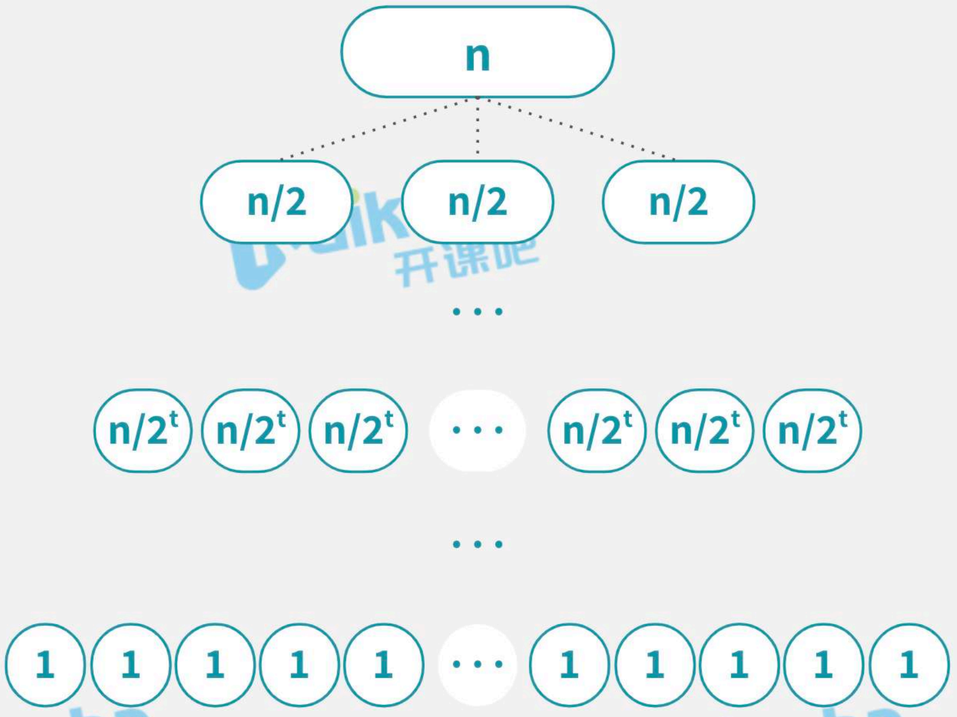
$\log_2 n$ levels
(需要划分 $\log_2 n$ 次才能得到大小为 1 的子问题)

最后一层的问题数
$= 4^{\log_2 n} = n^{\log_2 4}$
$= \mathbf{n^2}$

Karatsuba算法中, 每次划分的分支数由4变为3 ⇒

算法运行时间如何?

Karatsuba 算法的递归树



Level 0: 1 个大小为 n 的问题

Level 1: 3^1 个大小为 n/2 的问题

Level t: 3^t 个大小为 $n/2^t$ 的问题

Level $\log_2 n$: $n^{1.6}$ 个大小为 1 的问题

$\log_2 n$ levels
(需要划分 $\log_2 n$ 次才能得到大小为 1 的子问题)

最后一层的问题数
$= 3^{\log_2 n} = n^{\log_2 3}$
$\approx \mathbf{n^{1.6}}$

即, 算法运行时间为 $O(n^{1.6})$!

算法运行时间如何?

Karatsuba 算法的递归树

注意: 看起来我们似乎不怎么关心递归树的上层，稍后会解释为什么只有最下层起到了决定性的影响

Level 0: 1 个大小为 n 的问题

Level 1: 3^1 个大小为 n/2 的问题

Level t: 3^t 个大小为 $n/2^t$ 的问题

Level $\log_2 n$: $n^{1.6}$ 个大小为 1 的问题

$\log_2 n$ levels
(需要划分 $\log_2 n$ 次才能得到大小为 1 的子问题)

最后一层的问题数
$= 3^{\log_2 n} = n^{\log_2 3}$
$\approx \mathbf{n^{1.6}}$

即, 算法运行时间为 $O(n^{1.6})$!

还能更快吗？

- **Toom-Cook (1963): 另一种分治算法**，划分为五个 (n/3) 大小的子问题
  - $O(n^{1.465})$
- **Schönhage–Strassen (1971): 一种快速多项式乘法**
  - $O(n \log n \log \log n)$
- **Fürer (2007):** 使用傅立叶变换
  - $O(n \log(n) 2^{O(\log^*(n))})$
- **Harvey and van der Hoeven (2019!):** 太复杂了没看懂
  - $O(n \log(n))$

渐进分析

渐进记号(大-Ω 与 大-Θ)

刚才讲到

渐进记号的意义

忽略 **常数项** **和** **低阶项**

取决于运行环境 *输入增大时不是主要增长因素*

- **一些指导性原则：**我们在乎的是运行时间/操作次数如何随着输入规模的变大而增加，我们需要一种与硬件、编程语言、内存等无关的评估方式。
  - 现实中，诸如硬件、编程语言、内存、编译器优化等等仍然是十分重要的因素。在学习算法分析的时候我们暂时忽略这些细节。

运行时间分析

关于算法运行时间的分析有不同方式:

最坏情况分析:
输入是最坏情况的时候算法的运行时间如何?

最好情况分析:
输入是最好情况的时候算法的运行时间如何?

平均情况分析:
算法在平均情况下的运行时间如何?

我们重点关心最坏情况分析，这代表了任何输入情况下的运行时间上界

将在随机算法部分讨论

大-O表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

(令T(n)为最坏情况的算法运行时间)

“T(n) 是 O(f(n))”代表什么意思?

语言定义

图像定义

数学定义

大-O表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

(令T(n)为最坏情况的算法运行时间)

“T(n) 是 O(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*上界

图像定义

数学定义

大-O表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

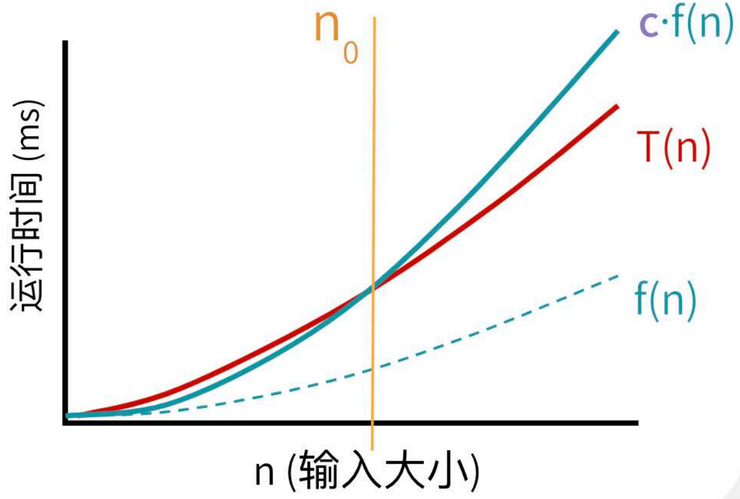
(令T(n)为最坏情况的算法运行时间)

“T(n) 是 O(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*上界

图像描述



数学定义

大-O表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

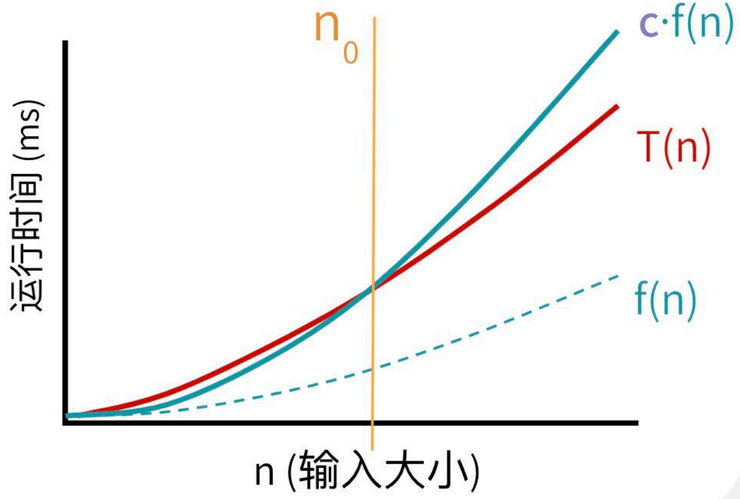
(令T(n)为最坏情况的算法运行时间)

“T(n) 是 O(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*上界

图像描述



数学描述

T(n) = O(f(n)) 当且仅当
存在正常量
c 和 n_0 使得对*所有* $n \geq n_0$

$$T(n) \leq c \cdot f(n)$$

大-O表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

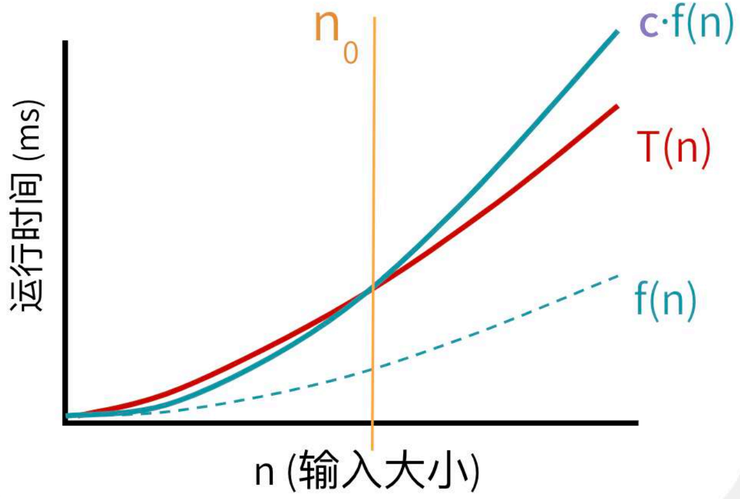
(令T(n)为最坏情况的算法运行时间)

“T(n) 是 O(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*上界

图像描述



数学描述

$$T(n) = O(f(n))$$

$$\Leftrightarrow$$

$$\exists\, c, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,$$

$$T(n) \leq c \cdot f(n)$$

大-O表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

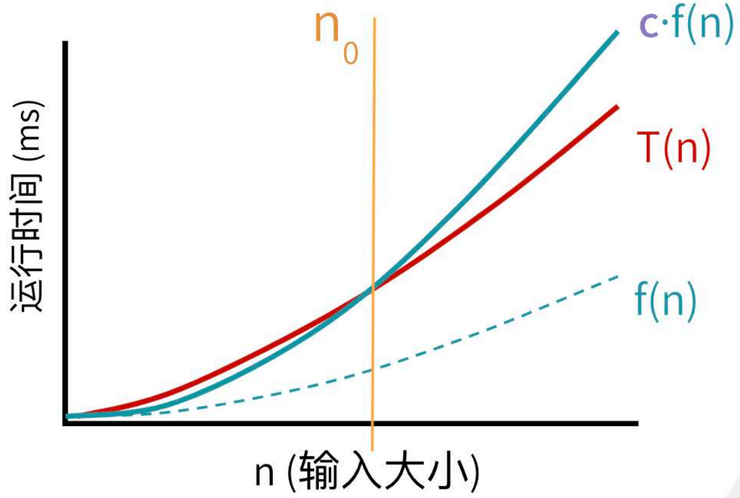
(令T(n)为最坏情况的算法运行时间)

“T(n) 是 O(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*上界

图像描述



数学描述

$$T(n) = O(f(n)) \Leftrightarrow \exists\, c, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,\ T(n) \leq c \cdot f(n)$$

“当且仅当” “所有” “存在” “使得 (such that)”

证明大-O边界

证明时使用数学描述:

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\, c, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,$$
$$\mathbf{T(n) \leq c \cdot f(n)}$$

- **证明** $T(n) = O(f(n))$ 前, 首先需要声明 c & n_0 ← c & n_0 必须和 n 无关
  - 观察给出的 f(n), 选取合适的 c & n_0 (正常量)
  - 然后给出证明! 证明的开头方式为:

“令 c = __， n_0 = __. 我们证明对所有 $n \geq n_0$ 有 $T(n) \leq c \cdot f(n)$”

证明大-O边界：例

$$T(n) = O(f(n))$$
$$\Leftrightarrow$$
$$\exists\, c, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,$$
$$T(n) \leq c \cdot f(n)$$

证明 $3n^2 + 5n = O(n^2)$.

令 $c = 4$， $n_0 = 5$. 我们证明对所有 $n \geq n_0$ 有 $3n^2 + 5n \leq c \cdot n^2$. 已知对所有 $n \geq n_0$, 有:

$$5 \leq n$$
$$5n \leq n^2$$
$$3n^2 + 5n \leq 4n^2$$

代入所选 c 和 n_0 后 $3n^2 + 5n \leq c \cdot n^2$ 对所有 $n \geq n_0$ 均成立。根据大-O的定义, 得 $3n^2 + 5n = O(n^2)$.

■

证伪大-O边界

需要证伪T(n) 为 O(f(n))时, 使用 **反例!**

首先假设 T(n) 为 O(f(n))。即,
存在某对 c & n_0 s.t. ∀ $n \geq n_0$, **$T(n) \leq c \cdot f(n)$**

代入 c & n_0, 推导出矛盾!

推出原假设错误, 即 **T(n) 不是 O(f(n))**.

证伪大-O边界：例

$T(n) = O(f(n))$
$\Leftrightarrow$
$\exists\, c, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,$
$T(n) \leq c \cdot f(n)$

证明 $3n^2 + 5n$ 不为 O(n).

假设 $3n^2 + 5n$ 为 O(n). 即存在这样的正常量 c & n_0 使得 $3n^2 + 5n \leq c \cdot n$ 对所有 $n \geq n_0$ 均成立。则我们有:

$$3n^2 + 5n \leq c \cdot n$$
$$3n + 5 \leq c$$
$$n \leq (c - 5)/3$$

然而, 由于 $(c - 5)/3$ 是常量, 当 $n = n_0 + c$: $n \geq n_0$ 成立, 但 $n > (c - 5)/3$, 矛盾，原假设不成立，即 $3n^2 + 5n$ 不为 O(n).

■

大-O表示法举例

忽略低阶部分

$$\log_2 n + 15 = O(\log_2 n)$$

大-O是上界!

$$3^n = O(4^n)$$

多项式

令 k 次多项式 $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$，$k \geq 1$.

则:

i. $p(n) = O(n^k)$
ii. $p(n)$ 不是 $O(n^{k-1})$

常数系数 & 忽略低阶部分

$$6n^3 + n \log_2 n = O(n^3)$$

$$25 = O(1)$$

$$[\text{任意常量}] = O(1)$$

大-Ω表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

(令T(n) 为最坏情况的算法运行时间)

“T(n) 是 Ω(f(n))”代表什么意思?

语言定义

图像定义

数学定义

大-Ω表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

(令T(n) 为最坏情况的算法运行时间)

“T(n) 是 Ω(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*下界

图像定义

数学定义

大-Ω表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

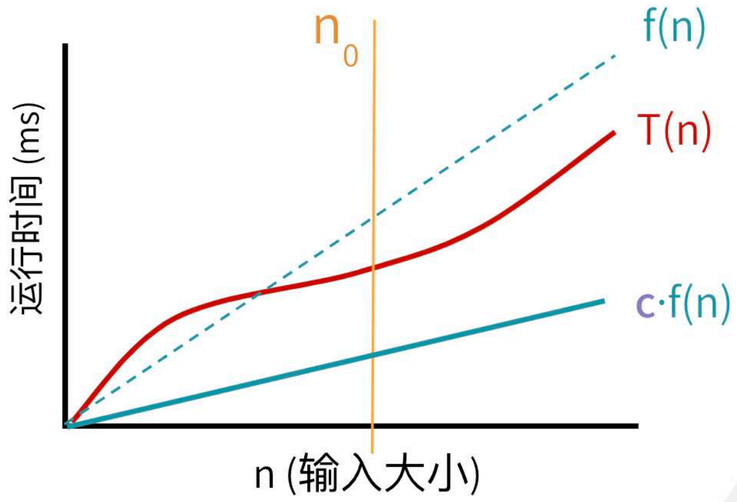
(令T(n) 为最坏情况的算法运行时间)

“T(n) 是 Ω(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*下界

图像描述



数学定义

大-Ω表示法

用来描述算法渐进运行时间的记号根据定义域为正整数集的函数 T(n) & f(n)来定义

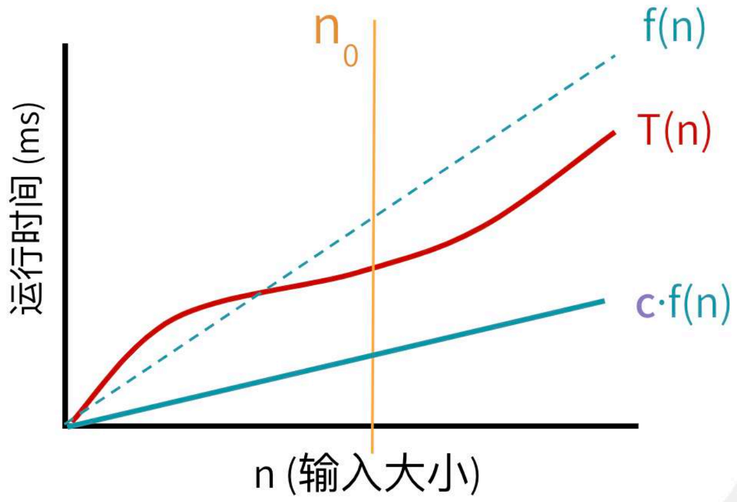
(令T(n) 为最坏情况的算法运行时间)

“T(n) 是 Ω(f(n))”代表什么意思?

语言描述

T(n) = O(f(n)) 当且仅当 T(n) 是 f(n) 乘以某个常数系数后的*渐进*下界

图像描述



数学描述

$$T(n) = \Omega(f(n))$$

$$\Leftrightarrow$$

$$\exists\, c, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,$$

$$\mathbf{T(n) \geq c \cdot f(n)}$$

大-Θ 表达式

“T(n) 为 Θ(f(n))” 当且仅当同时有

T(n) = O(f(n))

与

T(n) = Ω(f(n))

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

$$\Leftrightarrow$$

$$\exists\, c_1, c_2, n_0 > 0 \text{ s.t. } \forall\, n \geq n_0,$$

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$$

渐进记号总结

渐进界	定义 (如何证明)	意义
$T(n) = O(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0\ \text{ s.t. }\forall\ n \geq n_0,\ T(n) \leq c \cdot f(n)$	渐进上界
$T(n) = \Omega(f(n))$	$\exists\ c > 0,\ \exists\ n_0 > 0\ \text{ s.t. }\forall\ n \geq n_0,\ T(n) \geq c \cdot f(n)$	渐进下界
$T(n) = \Theta(f(n))$	$T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$	渐进紧确界

总结

- 讨论了如何 **设计**, **分析**, 以及 **交流** 算法
- 介绍了分治策略
- 分治算法举例：Karatsuba算法
- 使用渐进记号表示算法复杂度

接下来...

- 排序
- 更多分治算法
- 更多的运行时间分析

作业

- 实现 Karatsuba 算法
- 力扣
  - 50. Pow(x, n)
  - 932. 漂亮数组
  - 力扣 面试题 16.17. 连续数列
- *使用递归树法分析以上算法的运行时间*