USTC

Chapter 1
Algorithm Analysis Methods

王子磊（Zilei Wang）

Email: zlwang@ustc.edu.cn

http://vim.ustc.edu.cn/

学习要点

- 算法复杂度分析
- 递归算法的复杂性分析
  - 代入法、递归树、主定理
- 摊还分析方法
  - 聚合分析、核算法、势能法

算法复杂度分析

❖ 例：顺序搜索算法

```
template<class Type>
int seqSearch(Type *a, int n, Type k)
{
    for(int i=0; i<n; i++)
        if (a[i]==k) return i;
    return -1;
}
```

（1） $T_{\max}(n) = \max\{\ T(\mathrm{I}) \mid \mathrm{size}(\mathrm{I})=n\ \}=O(n)$

（2） $T_{\min}(n) = \min\{\ T(\mathrm{I}) \mid \mathrm{size}(\mathrm{I})=n\ \}=O(1)$

（3）在平均情况下，假设：

- (a) 搜索成功的概率为 p ($0 \le p \le 1$)
- (b) 在数组的每个位置 i ($0 \le i < n$)搜索成功的概率相同，均为 p/n

$$T_{avg}(n) = \sum_{size(I)=n} p(I)T(I)$$

$$= \left(1\cdot\frac{p}{n} + 2\cdot\frac{p}{n} + 3\cdot\frac{p}{n} + \cdots + n\cdot\frac{p}{n}\right) + n\cdot(1-p)$$

$$= \frac{p}{n}\sum_{i=1}^{n} i + n(1-p) = \frac{p(n+1)}{2} + n(1-p) = \Theta(n)$$

算法分析的基本法则

❖ **非递归算法：**

(1) for / while 循环

- 循环体内计算时间*循环次数；

(2) 嵌套循环

- 循环体内计算时间*所有循环次数；

(3) 顺序语句

- 各语句计算时间相加；

(4) if-else语句

- if语句计算时间和else语句计算时间的**较大者**

插入排序的分析计算

```
template<class Type>
void insertion_sort(Type *a, int n)
{
    Type key;                               //  cost    times
    for (int i = 1; i < n; i++){            //   c1     n
        key=a[i];                           //   c2     n-1
        int j=i-1;                          //   c3     n-1
        while( j>=0 && a[j]>key ){          //   c4     sum of ti
         a[j+1]=a[j];                       //   c5     sum of (ti-1)
         j--;                               //   c6     sum of (ti-1)
        }
       a[j+1]=key;                          //   c7     n-1
    }
}
```

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{i=1}^{n-1} t_i + c_5\sum_{i=1}^{n-1}(t_i - 1) + c_6\sum_{i=1}^{n-1}(t_i - 1) + c_7(n-1)$$

- 在最好情况下，$t_i = 1$, for $1 \le i < n$;

$$T_{\min}(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) = O(n)$$

- 在最坏情况下，$t_i \le i+1$, for $1 \le i < n$;

$$\sum_{i=1}^{n-1}(i+1) = \frac{n(n+1)}{2} - 1 \qquad \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$T_{\max}(n) \le c_1 n + c_2(n-1) + c_3(n-1) +$$

$$c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= \frac{c_4 + c_5 + c_6}{2} n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

$$= O(n^2)$$

- 对于输入数据 $a[i]= n-i$, i =0,1,…,n-1，算法 insertion_sort 达到其最坏情形。因此，

$$T_{\max}(n) \geq \frac{c_4+c_5+c_6}{2}n^2+\left(c_1+c_2+c_3+\frac{c_4-c_5-c_6}{2}+c_7\right)n$$
$$-(c_2+c_3+c_4+c_7)$$
$$=\Omega(n^2)$$

- 由此可见，$T_{\max}(n)=\Theta(n^2)$

最优算法

- 问题的计算时间下界为 $\Omega(f(n))$，则计算时间复杂性为 $O(f(n))$ 的算法是最优算法
- 例如：
  - 比较排序问题的计算时间下界为 $\Omega(n\log n)$，计算时间复杂性为 $O(n\log n)$ 的排序算法是最优算法
- 堆排序算法是最优算法

递归算法复杂度分析

递归算法复杂性分析

❖ 举例：

```
int factorial(int n)
 {
     if (n == 0) return 1;
     return n*factorial(n-1);
  }
```

❖ 复杂性分析：

$$T(n)=\begin{cases} 0 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

$$T(n)=n$$

归并排序

MERGE-SORT $A[1\,.\,.\,n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1\,.\,.\quad n/2\quad]$

   and $A[\,n/2 + 1\,.\,.\,n\,]$.
3. "***Merge***" the 2 sorted lists.

   Key subroutine: **MERGE**

序列的合并

20	12
13	11
7	9
2	1

序列的合并

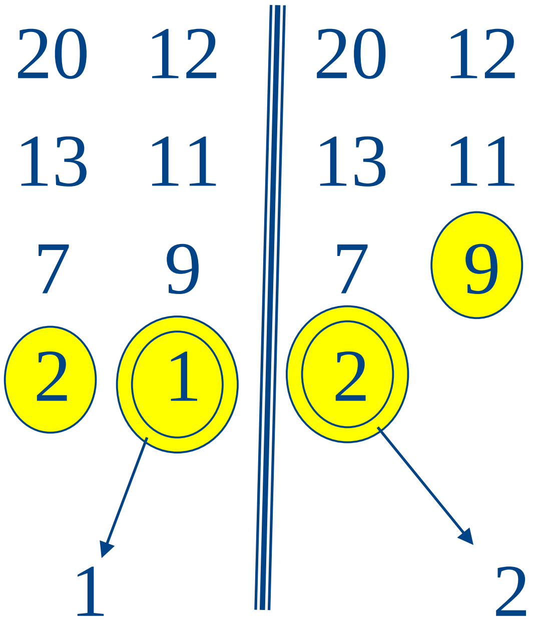
20	12
13	11
7	9
2	1

1

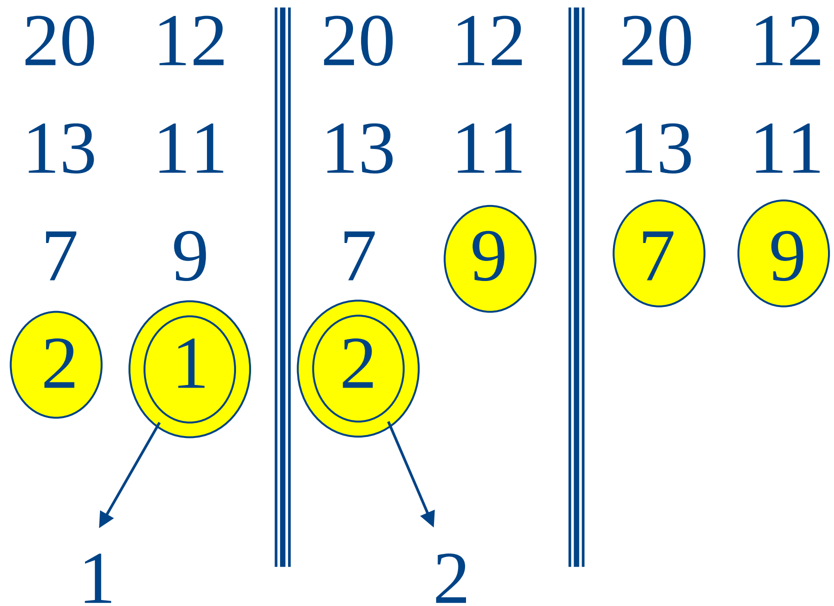
序列的合并

20	12	‖	20	12
13	11	‖	13	11
7	9	‖	7	9
2	1	‖	2	
1		‖		

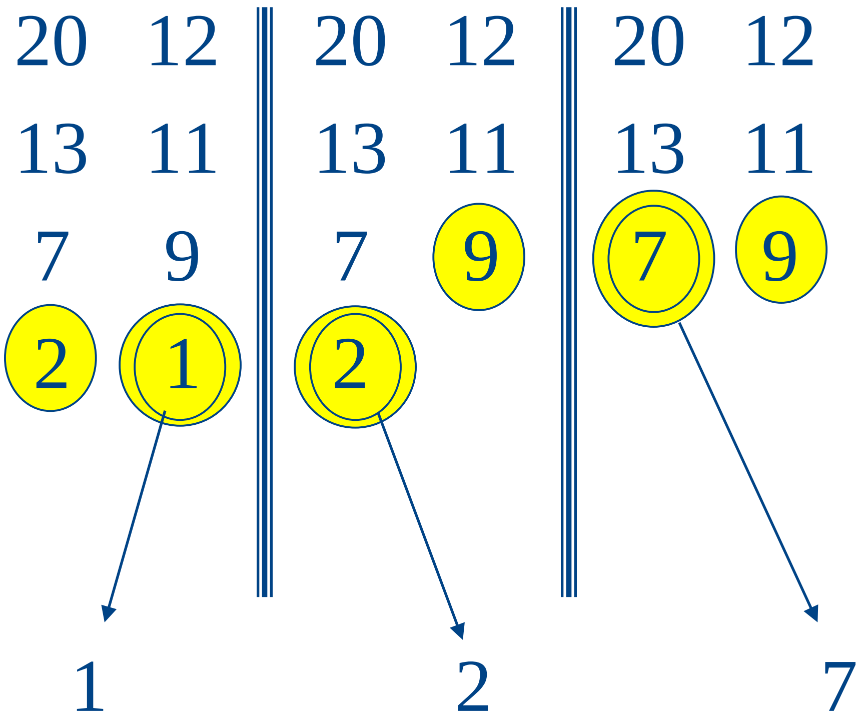
序列的合并



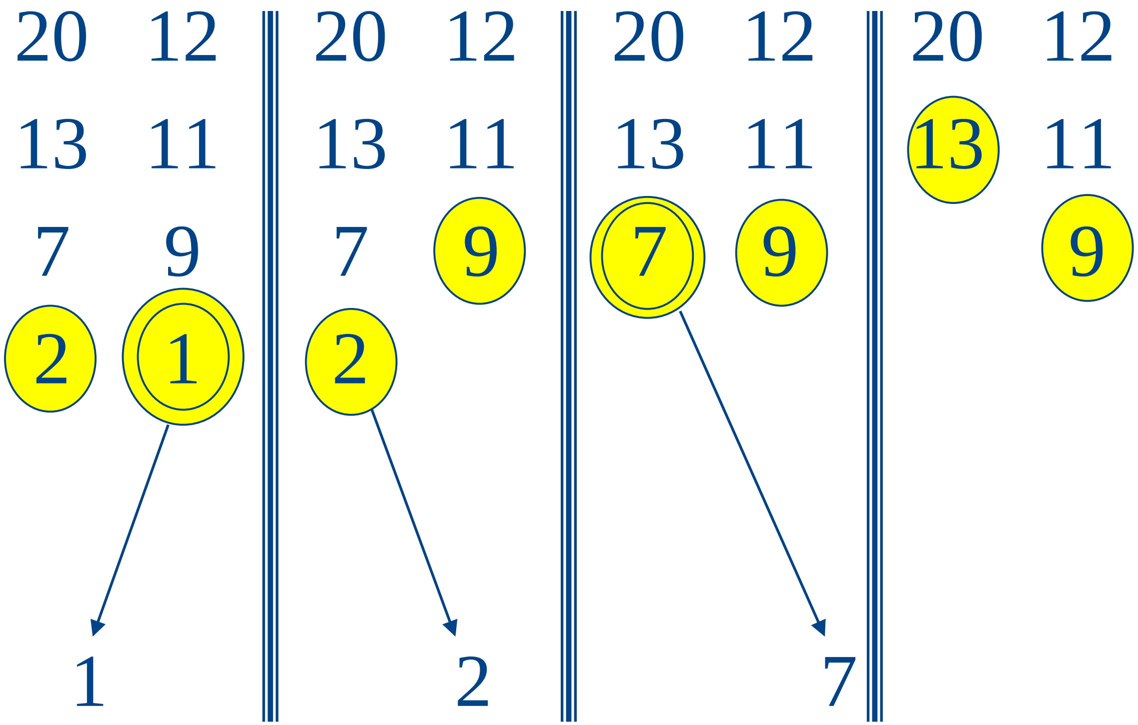
序列的合并



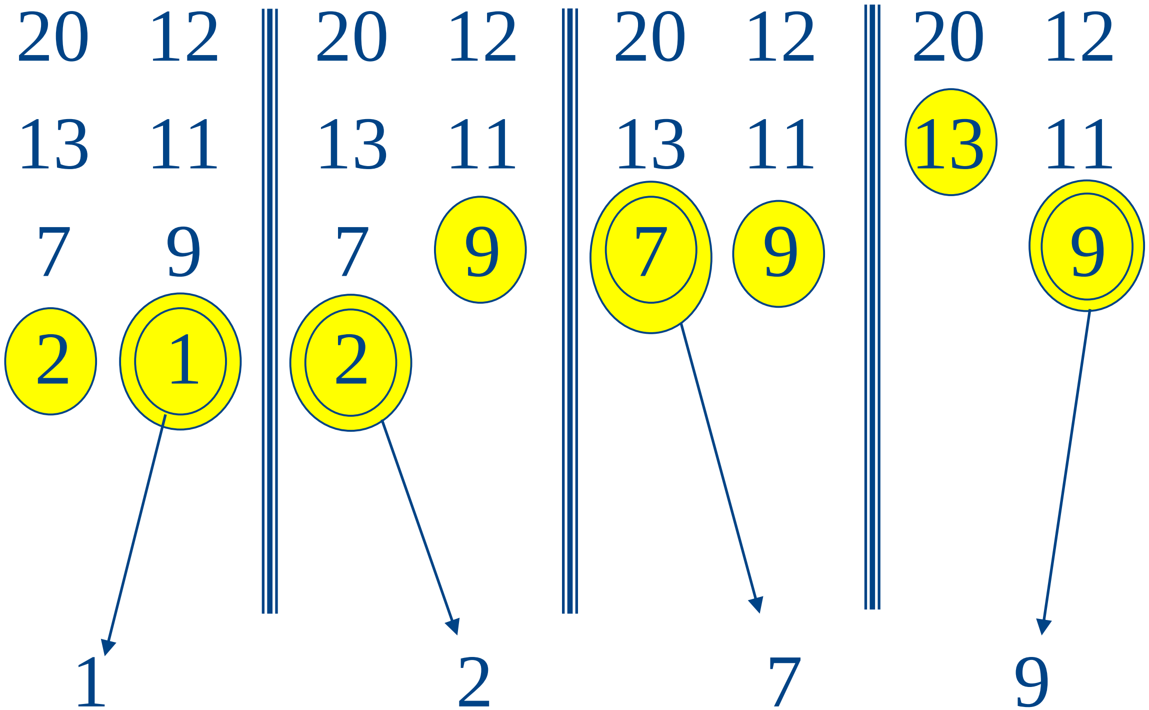
序列的合并



序列的合并



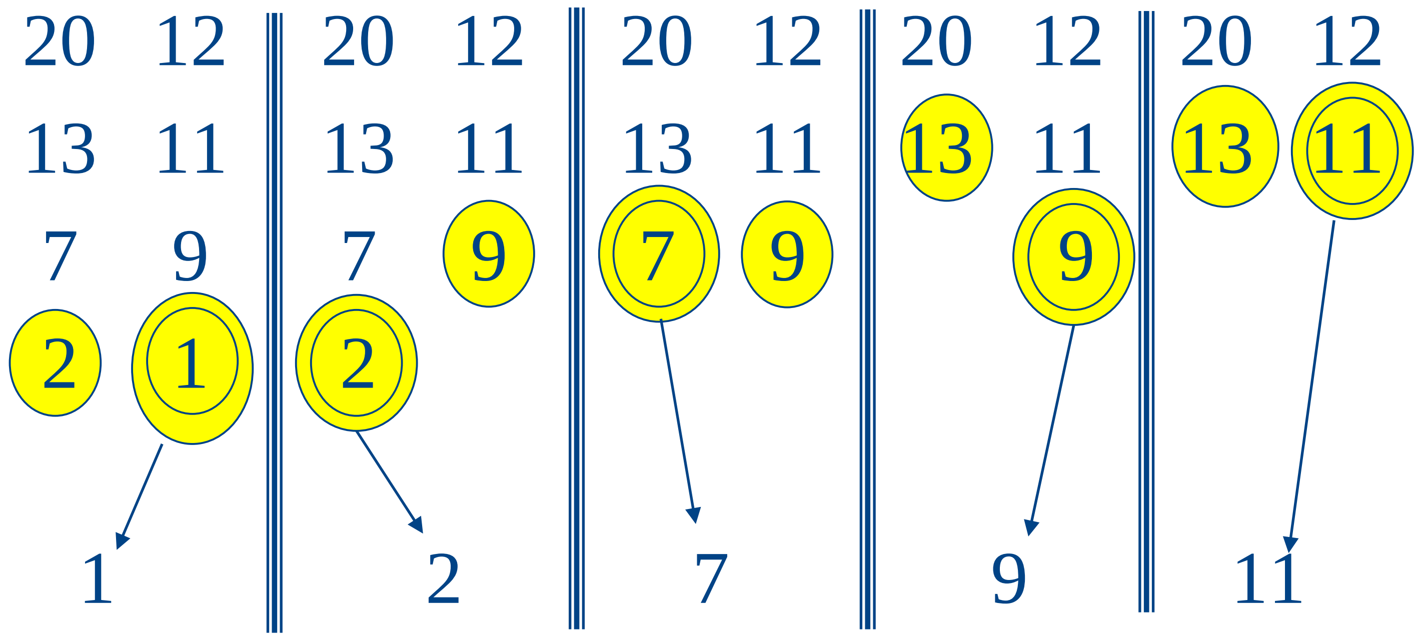
序列的合并



序列的合并

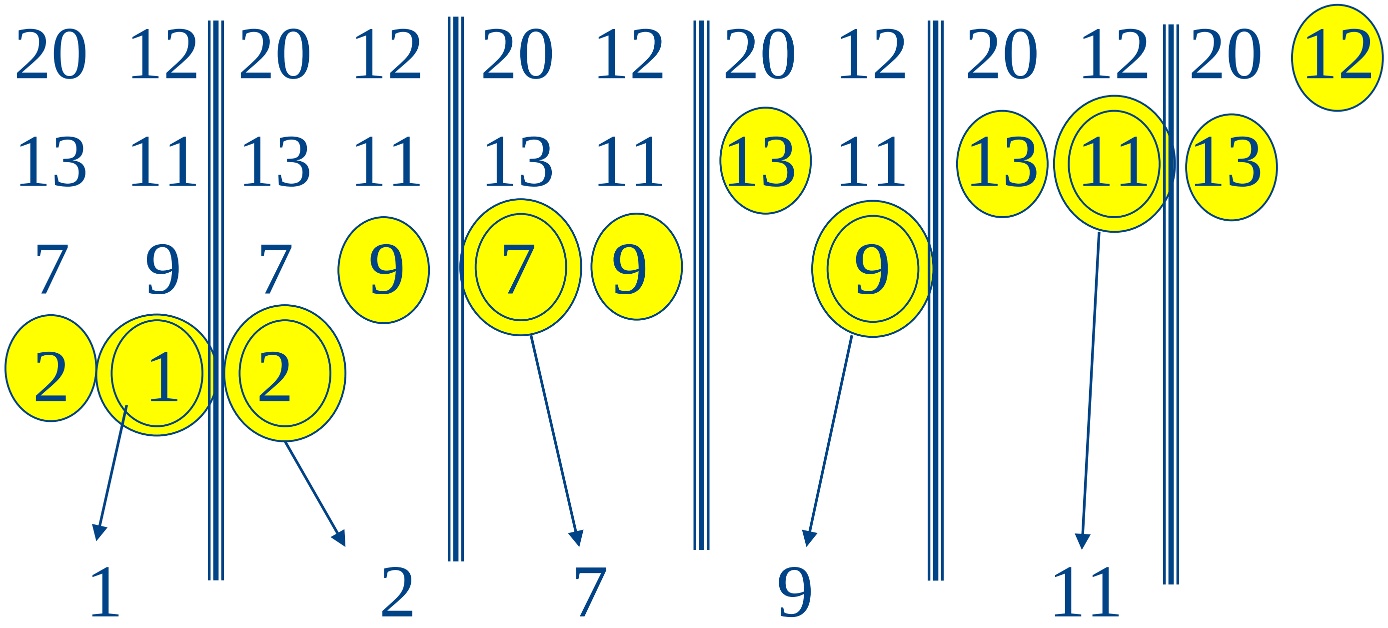
20	12	20	12	20	12	20	12	20	12
13	11	13	11	13	11	13	11	13	11
7	9	7	9	7	9		9		
2	1	2							
1		2		7		9			

序列的合并



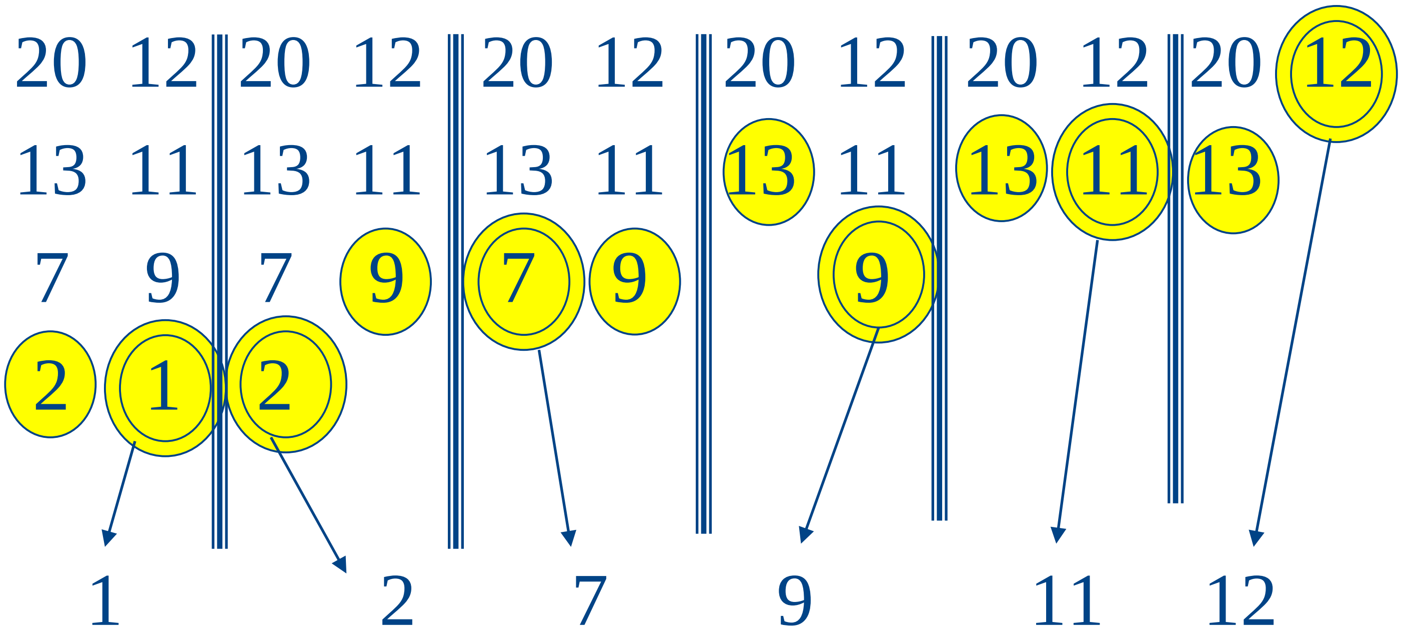
20 12 | 20 12 | 20 12 | 20 12 | 20 12
13 11 | 13 11 | 13 11 | 13 11 | 13 11
7 9 | 7 9 | 7 9 | 9
2 1 | 2
1 | 2 | 7 | 9 | 11

序列的合并



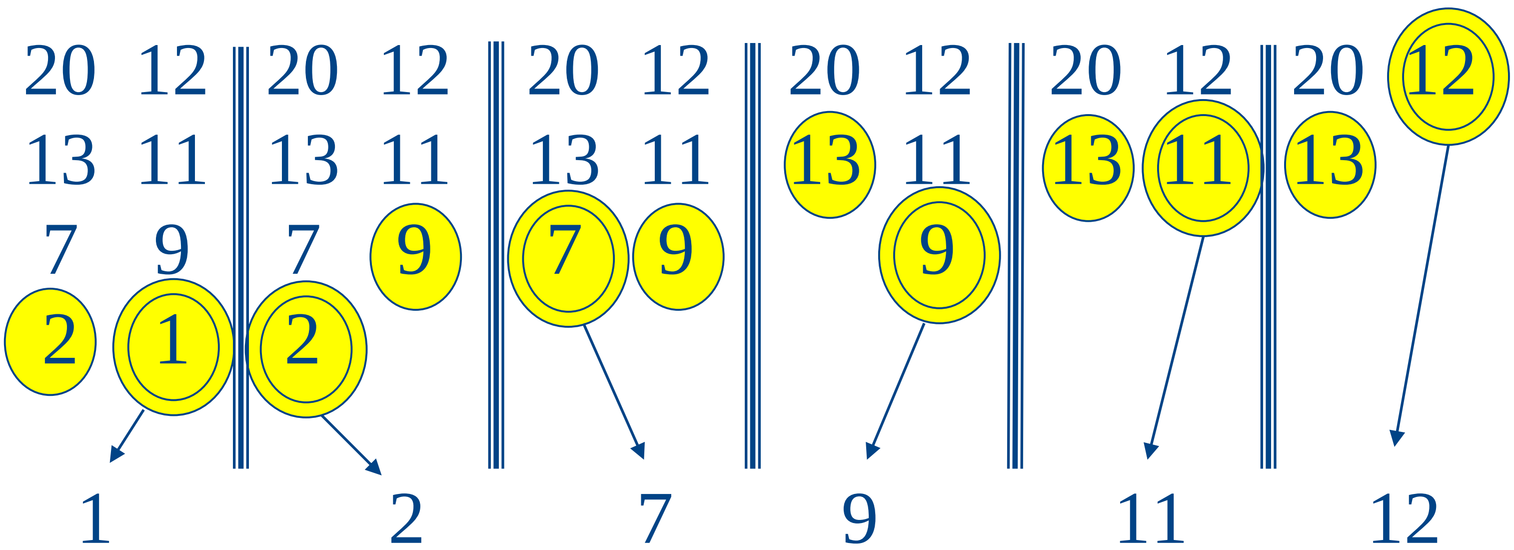
20 12 20 12 20 12 20 12 20 12 20 12
13 11 13 11 13 11 13 11 13 11 13
7 9 7 9 7 9 9
2 1 2
1 2 7 9 11

序列的合并



20 12 | 20 12 | 20 12 | 20 12 | 20 12 | 20 12
13 11 | 13 11 | 13 11 | 13 11 | 13 11 | 13
7 9 | 7 9 | 7 9 | 9
2 1 | 2
1 2 7 9 11 12

序列的合并



Time = $\Theta(n)$ to merge a total of n elements —— linear time

归并算法分析

$T(n)$	**MERGE-SORT** $A[1\,..\,n]$
$\Theta(1)$	1. If $n = 1$, done.
$2T(n/2)$	2. Recursively sort $A[\,1\,..\,n/2\,]$
Abuse	and $A[\,n/2+1\,..\,n\,]$.
$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

归并排序的递归

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

当 $T(n)=\Theta(1)$时，我们通常不陈述足够小 n 下的基础情况，但是要求对迭代的渐进解没有影响

- 我们后面分析出 $T(n)$ 的上界

递归树

求解 $T(n) = 2T(n/2) + cn$, 这里 $c > 0$ 是常数

递归树

求解 $T(n) = 2T(n/2) + cn$, 这里 $c > 0$ 是常数

$$T(n)$$

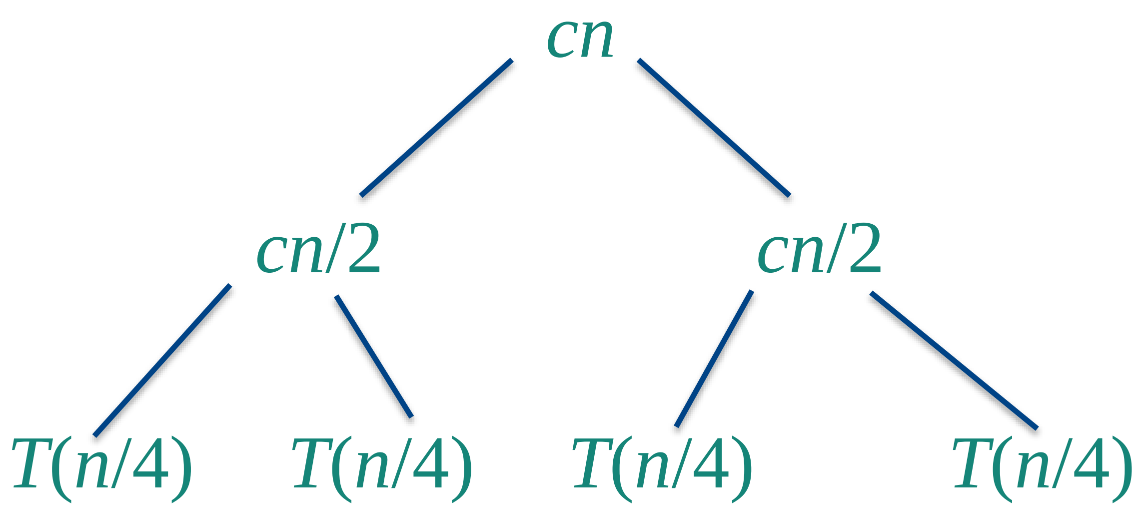
递归树

求解 $T(n) = 2T(n/2) + cn$, 这里 $c > 0$ 是常数

$$
\begin{array}{ccc}
 & cn & \\
\swarrow & & \searrow \\
T(n/2) & & T(n/2)
\end{array}
$$

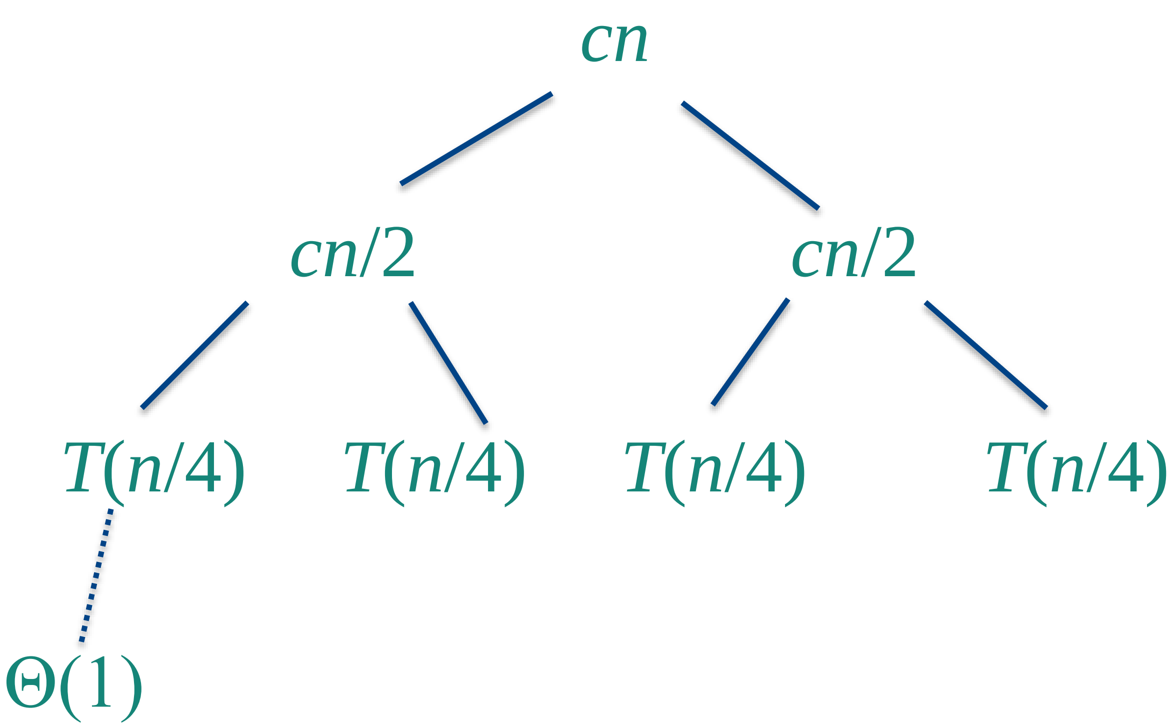
递归树

求解 $T(n) = 2T(n/2) + cn$, 这里 $c > 0$ 是常数



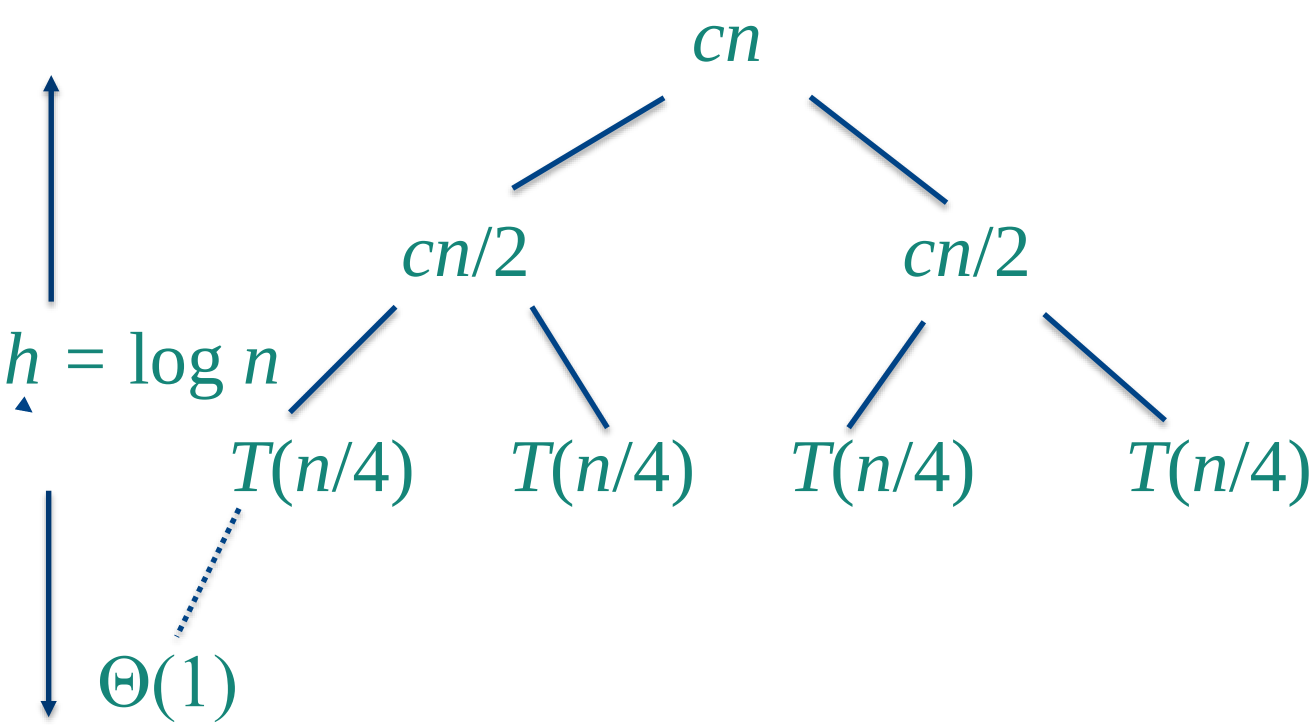
递归树

求解 $T(n) = 2T(n/2) + cn$, 这里 $c > 0$ 是常数



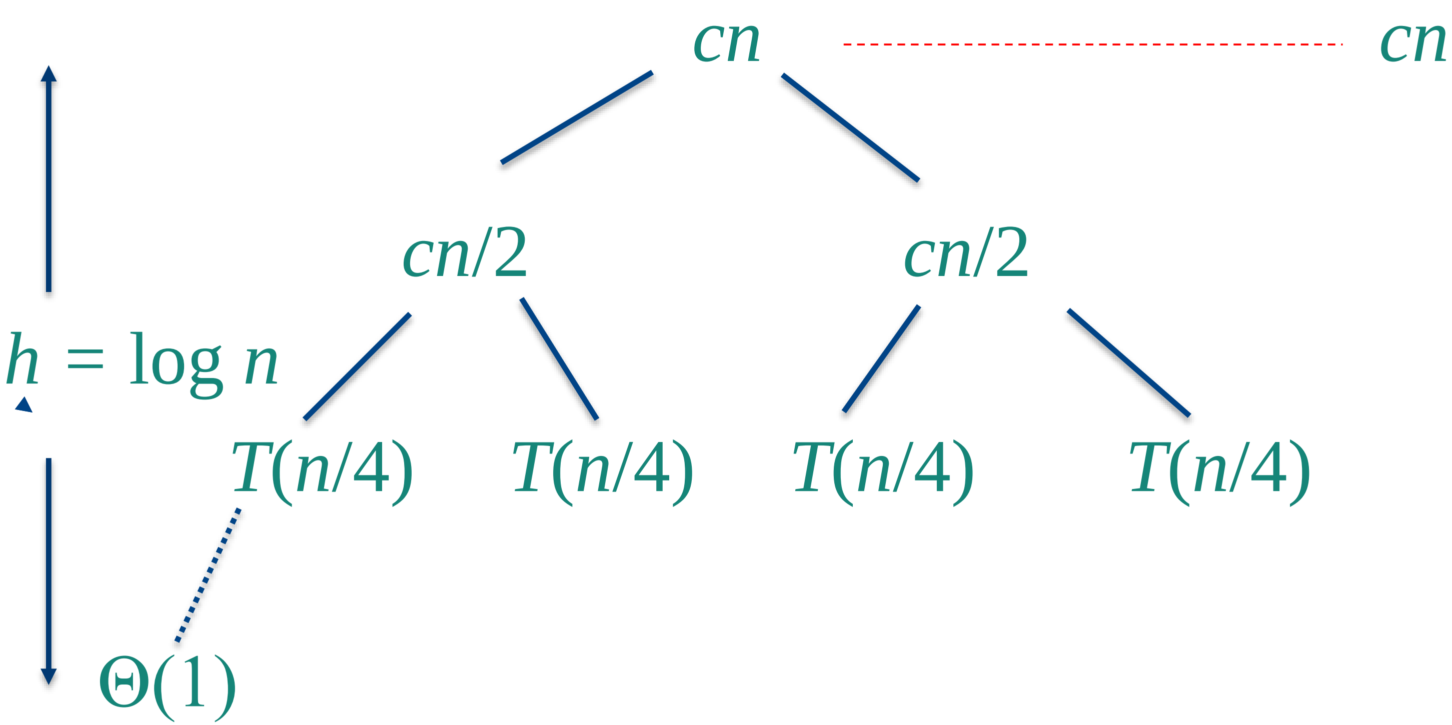
递归树

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



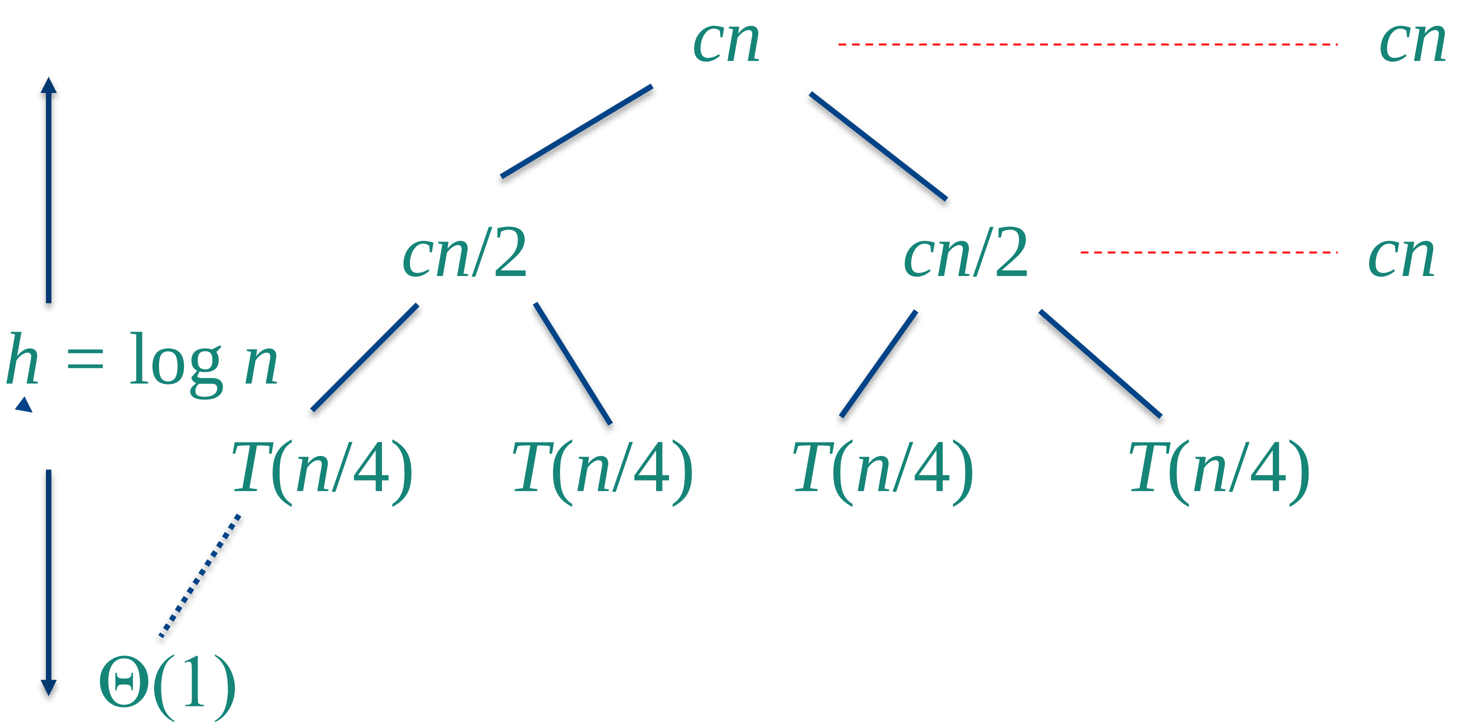
递归树

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



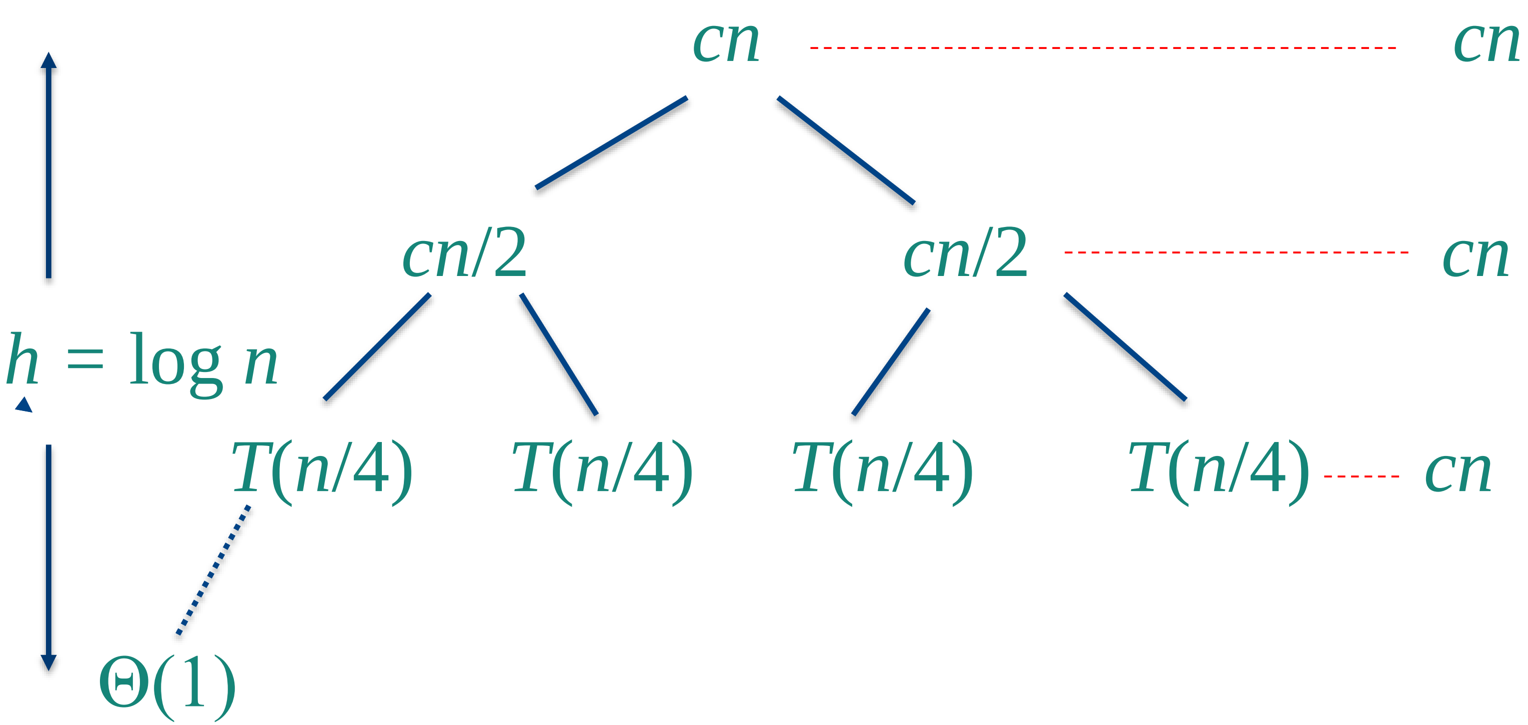
递归树

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



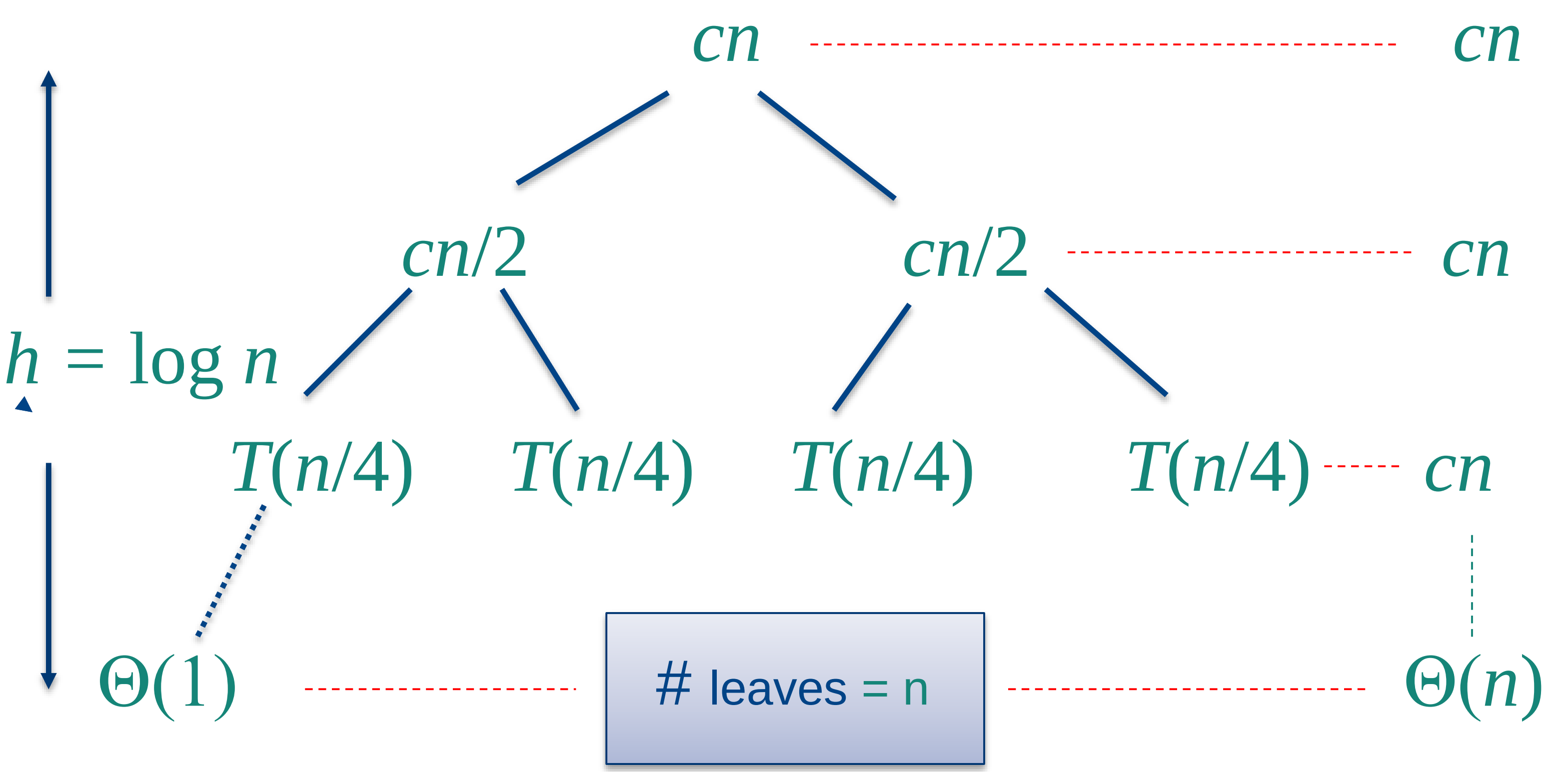
递归树

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



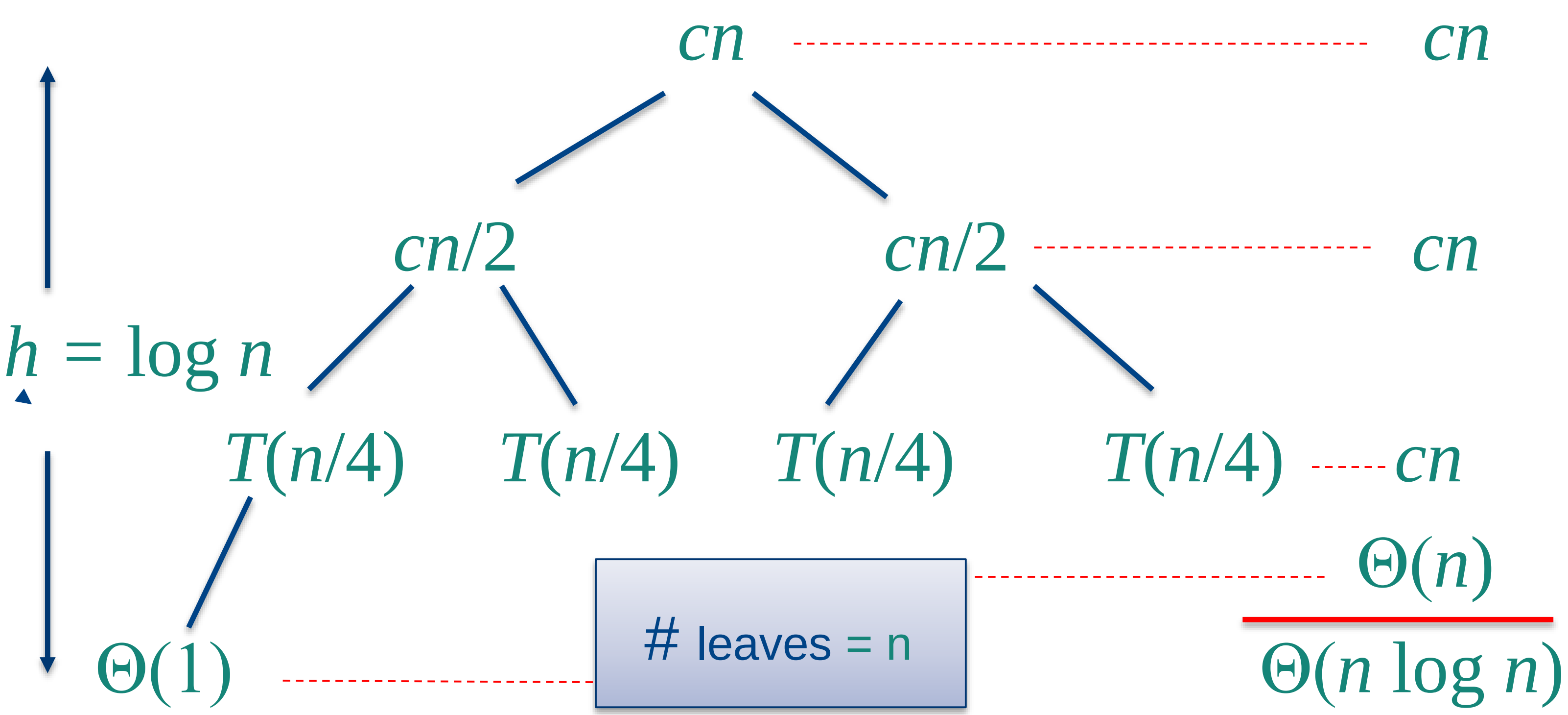
递归树

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



递归树

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



与插入排序对比

- $\Theta(n \log n)$ 的增长速度慢于 $\Theta(n^2)$
  - 在最坏情况下，归并排序算法渐进意义上优于插入排序算法
- 在实际应用中，通常是在 $n>30$ 左右后归并排序才开始打败插入排序
  - 可以在自己电脑上做个测试！

迭代算法分析的一般方法

- 代入法
- 递归树方法
- 主定理

代入法

代入法求解递归式的一般步骤：

1. **猜测**解的形式
2. **验证**解的形式，并**计算**常数

示例： $T(n) = 4T(n/2) + n$

→ 形式猜测？

- [假定 $T(1) = \Theta(1)$]
- 猜测 $T(n)=O(n^3)$ （可分别证明 O 和 Ω）
  - 假设 $T(k) \le ck^3$ for $k < n$
  - 推理法证明 $T(n) \le cn^3$

代入法示例

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

$$\leqq 4c(n/2)^3 + n$$

$$= (c/2)n^3 + n$$

$$= cn^3 - ((c/2)n^3 - n) \leftarrow \textbf{desired} - \textbf{residual}$$

$$\leqq cn^3 \leftarrow \textbf{desired}$$

当 $(c/2)n^3 - n \geq 0$, 比如： $c \geq 2$ and $n \geq 1$

↖

residual

代入法示例

- 这证明过程中，我们必须处理初始条件，也就是基本情况的推理
  - 基本情况: $T(n) = \Theta(1)$ for all $n < n_0$, where n_0 is a suitable constant.
  - For $1 \le n < n_0$, we have "$\Theta(1)$" $\le cn^3$, if we pick c big enough.
- 然而，这是一个非紧界！

更紧的上界?

我们现在证明 $T(n) = O(n^2)$

假定 $T(k) \le ck^2$ for $k < n$ ::

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

$$\le cn^2 + n$$

$= \cancel{O(n)}$ ***Wrong!*** We must prove the I.H. $\boxed{O(n) = O(1)?}$

$= cn^2 - (-n)$ [desired –residual]

$$\le cn^2$$

→不存在 $c > 0$ 满足上述条件，失败!?

更紧的上界？

- IDEA: 强化推理猜测
- 在假定中减去一个低阶项
  - 在证明过程中，当常数无法获取时，通常是减去一个低阶项而不是增加
  - 推理假定：$T(k) \leq c_1 k^2 - c_2 k$ for $k < n$.

$$
\begin{aligned}
T(n) &= 4T(n/2) + n \\
&\leq 4(c_1(n/2)^2 - c_2(n/2)) + n \\
&= c_1 n^2 - 2 c_2 n - (c_2 n - n) \\
&\leq c_1 n^2 - c_2 n \quad \text{if } c_2 < 1
\end{aligned}
$$

- 这里，c_1 取足够大就能够处理初始的基本情况

递归树方法

- 递归树模型刻画了一个算法在递归执行时的时间消耗
- 递归树方法是产生代入法猜测的一个很好工具
  - 递归树方法并不是总是可靠的(非万能)
  - 递归树方法非常直观

递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：

递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：

$$\boldsymbol{T(n)}$$

递归树示例

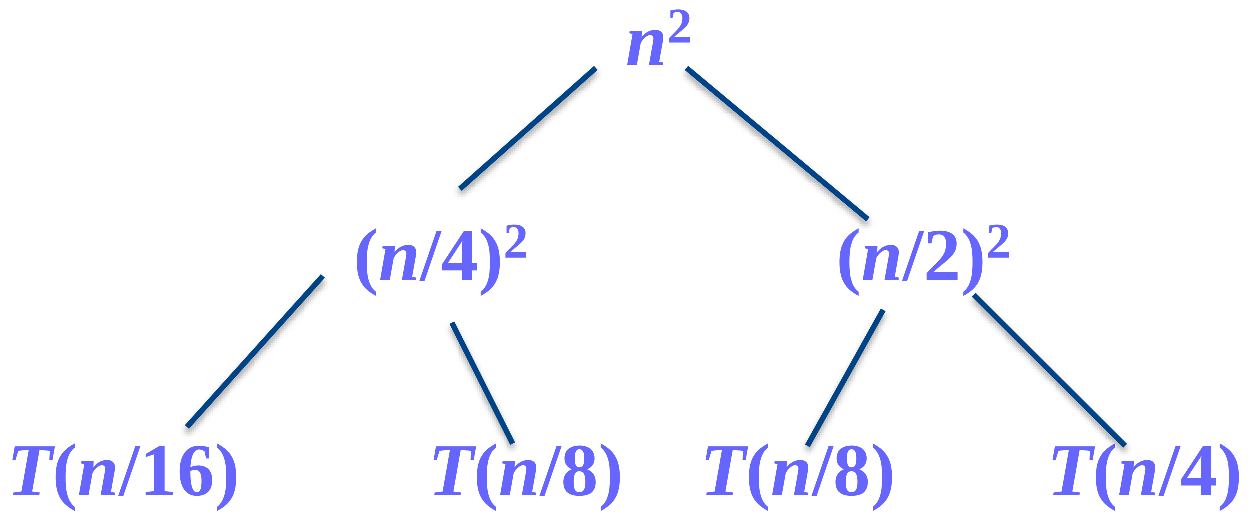
求解 $T(n) = T(n/4) + T(n/2) + n^2$：

$$n^2$$

$$T(n/4) \qquad\qquad T(n/2)$$

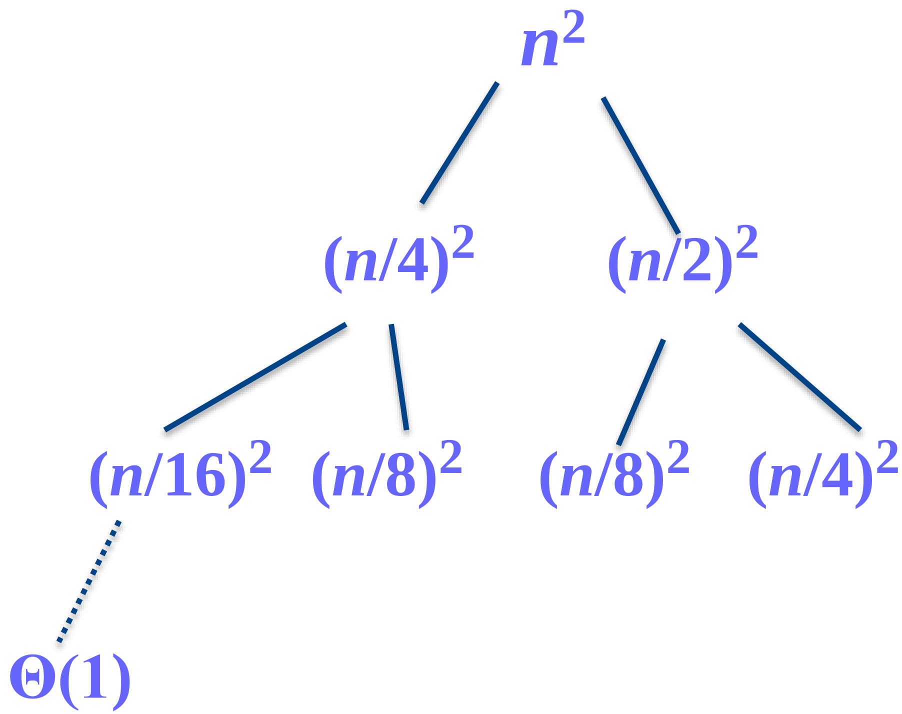
递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：



递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：



递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：

$$n^2 \qquad\qquad n^2$$

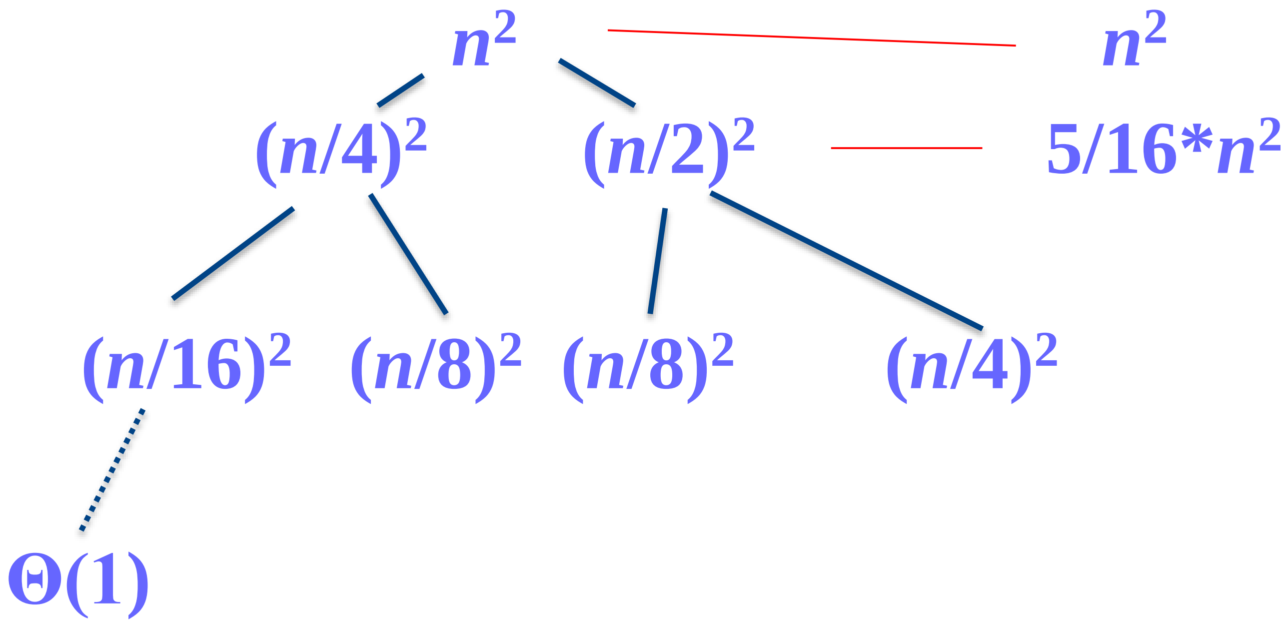
$$(n/4)^2 \qquad (n/2)^2$$

$$(n/16)^2 \quad (n/8)^2 \quad (n/8)^2 \quad (n/4)^2$$

$$\Theta(1)$$

递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：



递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：

n^2 —— n^2

$(n/4)^2$ $(n/2)^2$ —— $5/16*n^2$

$(n/16)^2$ $(n/8)^2$ $(n/8)^2$ $(n/4)^2$ —— $25/256*n^2$

$\Theta(1)$

递归树示例

求解 $T(n) = T(n/4) + T(n/2) + n^2$：

$$n^2 \quad\text{———}\quad n^2$$

$$(n/4)^2 \qquad (n/2)^2 \quad\text{———}\quad 5/16*n^2$$

$$(n/16)^2 \quad (n/8)^2 \quad (n/8)^2 \quad (n/4)^2 \quad\text{———}\quad 25/256*n^2$$

$$\Theta(1)$$

$$\text{Total} = n^2(1+5/16+(5/16)^2+(5/16)^3+\ldots)$$

$$= \Theta(n^2)$$

主定理方法

❖ 主定理方法应用于如下的递归形式

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

其中，$a \geq 1, b > 1, f$ 是渐近正的

主定理的三种情况

- 比较 $f(n)$ 和 $n^{\log_b a}$：
  - $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$
    - $f(n)$ 的增长速度比 $n^{\log_b a}$ 慢一个 n^{ε} 因子
    - Solution: $T(n) = \Theta(n^{\log_b a})$
  - $f(n) = \Theta(n^{\log_b a} \log^k n)$ for some constant $k \geq 0$
    - $f(n)$ 与 $n^{\log_b a} \log^k n$ 具有相似的增长速度
    - Solution: $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

主定理的三种情况

❖ 比较 $f(n)$ 和 $n^{\log_b a}$：

- $f(n) = \Omega(n^{\log_b a+\varepsilon})$ for some constant $\varepsilon > 0$
    - $f(n)$的增长速度比 $n^{\log_b a}$ 快一个 n^{ε} 因子
- 且 $f(n)$满足正则条件，即：

$$af(\frac{n}{b}) \leq cf(n) \text{ for some constant } 0 < c < 1$$

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

示例

Ex. $T(n) = 4T(n/2) + n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2,\ f(n) = n.$

→ CASE 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$

$\therefore T(n) = \Theta(\boldsymbol{n}^2)$

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2$

→ CASE 2: $f(n) = \Theta(n^2 \log^0 n)$, that is, $k = 0$

$\therefore T(n) = \Theta(n^2 \log n)$

示例

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2;\quad f(n) = n^3$

→ **CASE 3:** $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(n/2)^3 \leq cn^3$ (reg. cond.) for $c = 1/2$

$\therefore T(n) = \Theta(n^3)$

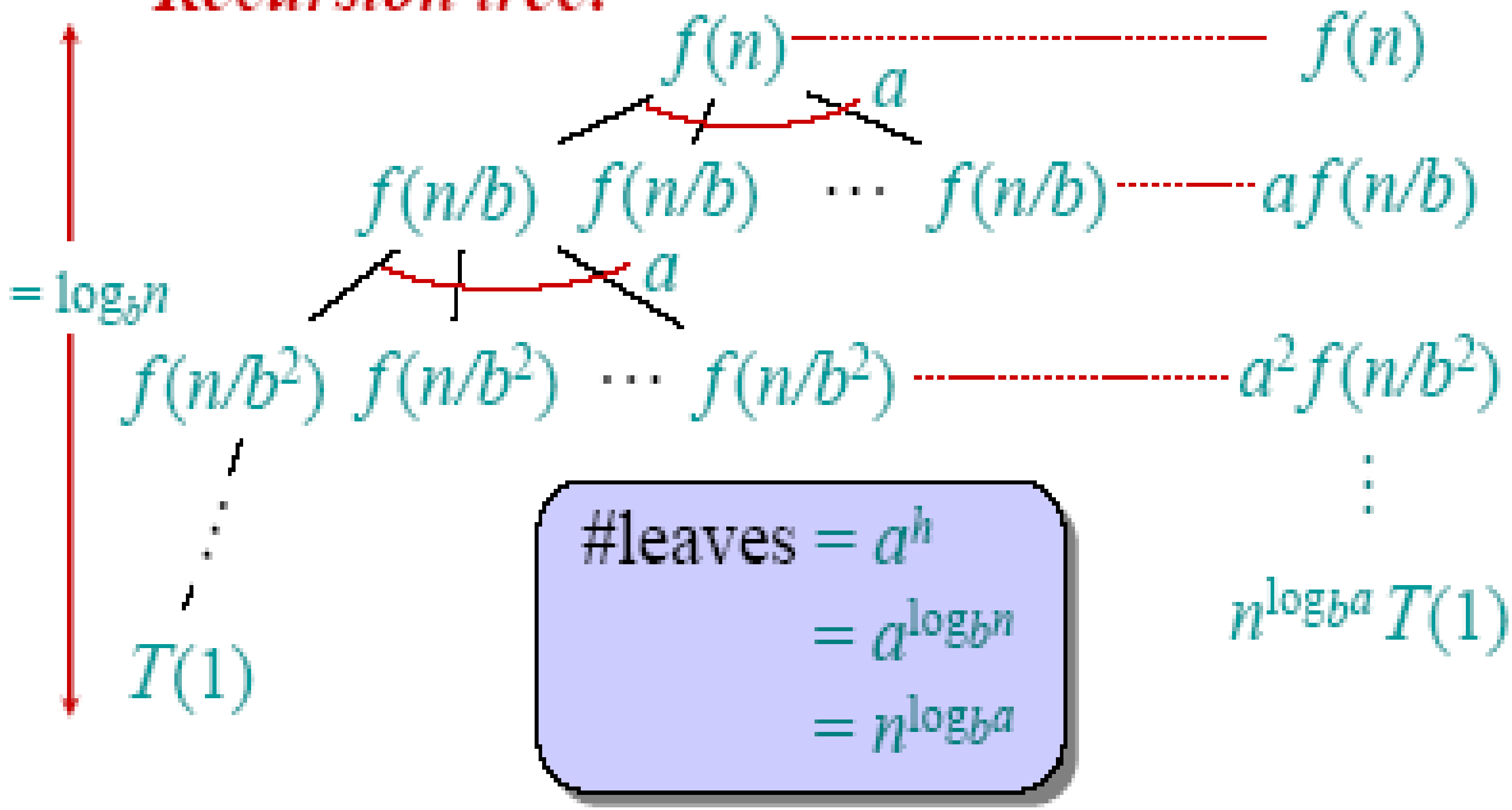
Ex. $T(n) = 4T(n/2) + n^2/\log n$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\log n$

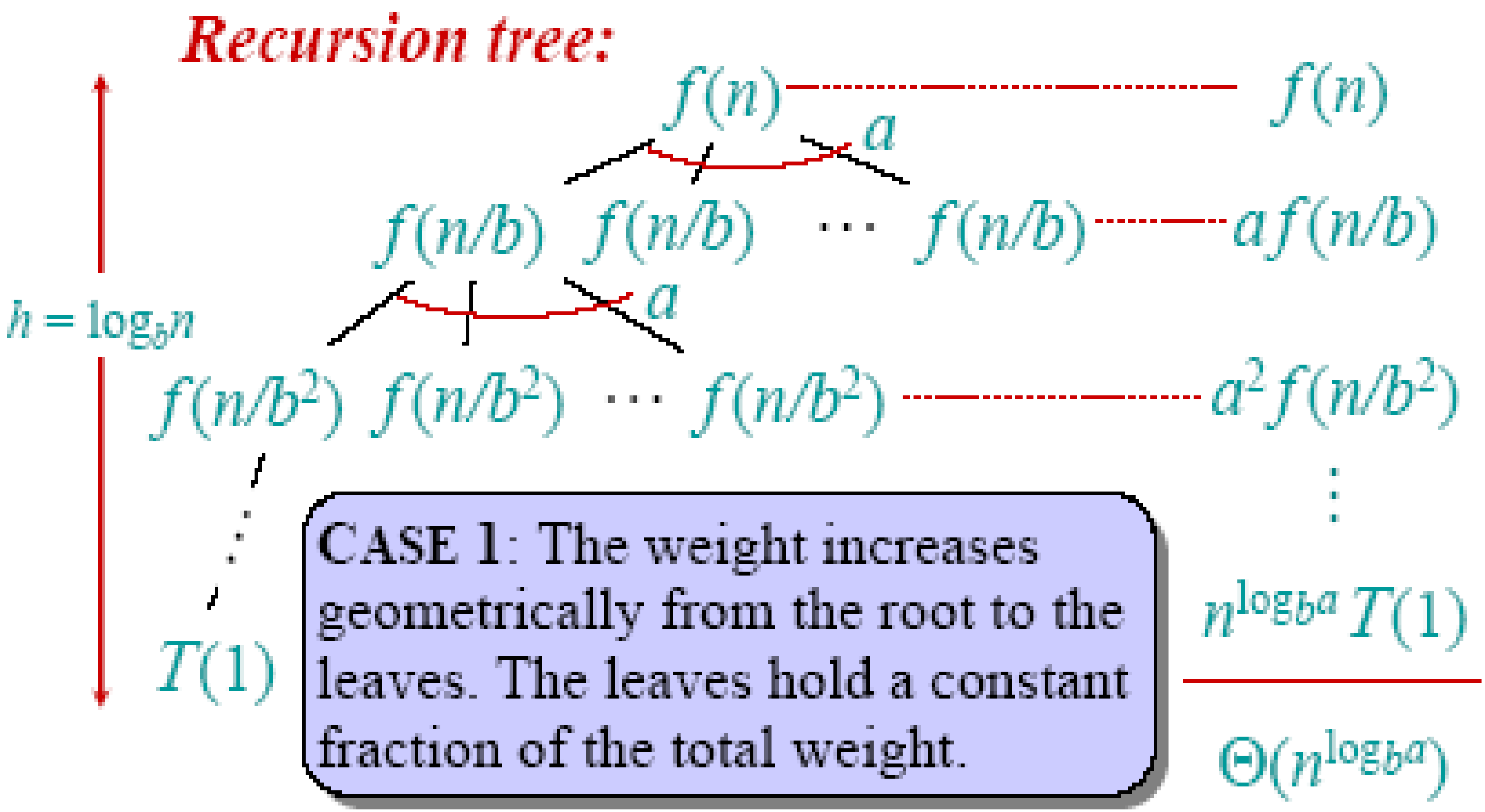
主定理对这种情况不再适用，特别地，对常数 $\varepsilon > 0$, 有 $n^\varepsilon = \omega(\log n)$

主定理的思想

Recursion tree:

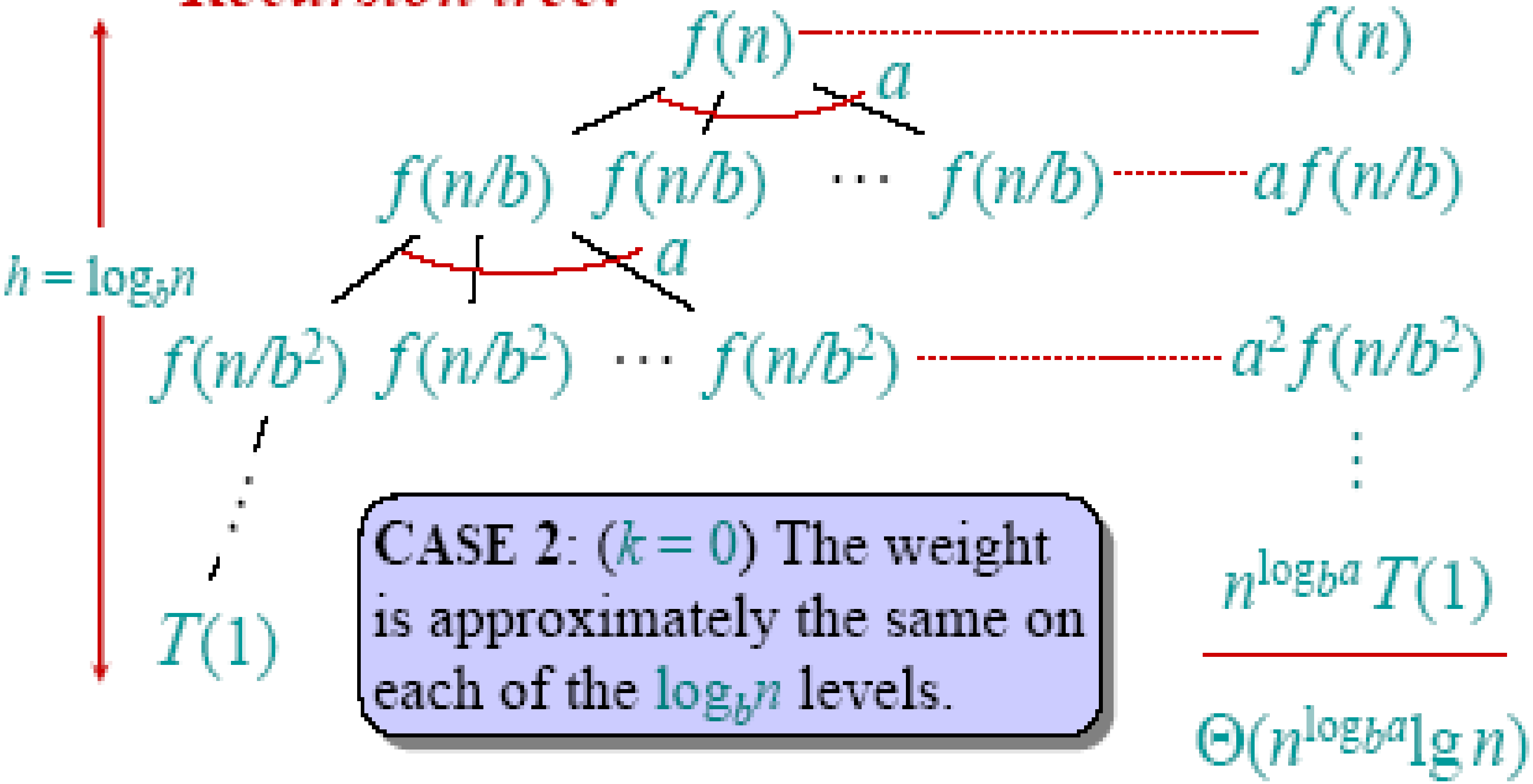


主定理的思想


Recursion tree:
$f(n)$
a
$f(n/b)$ $f(n/b)$ $\cdots$ $f(n/b)$
a
$h = \log_b n$
$f(n/b^2)$ $f(n/b^2)$ $\cdots$ $f(n/b^2)$
$T(1)$
CASE 1: The weight increases geometrically from the root to the leaves. The leaves hold a constant fraction of the total weight.
$f(n)$
$af(n/b)$
$a^2 f(n/b^2)$
$\vdots$
$n^{\log_b a} T(1)$
$\Theta(n^{\log_b a})$

主定理的思想

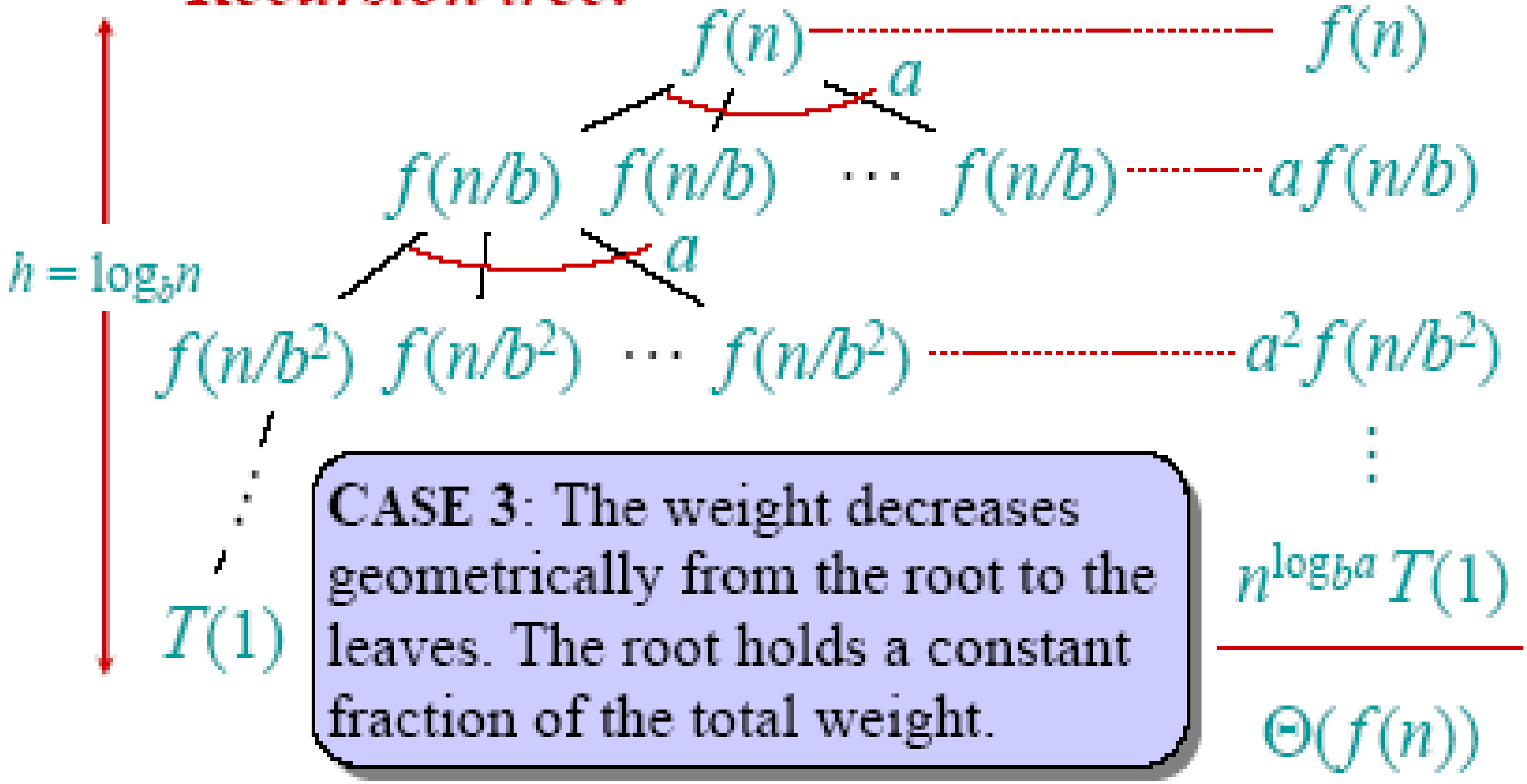
Recursion tree:



CASE 2: ($k = 0$) The weight is approximately the same on each of the $\log_b n$ levels.

主定理的思想

Recursion tree:



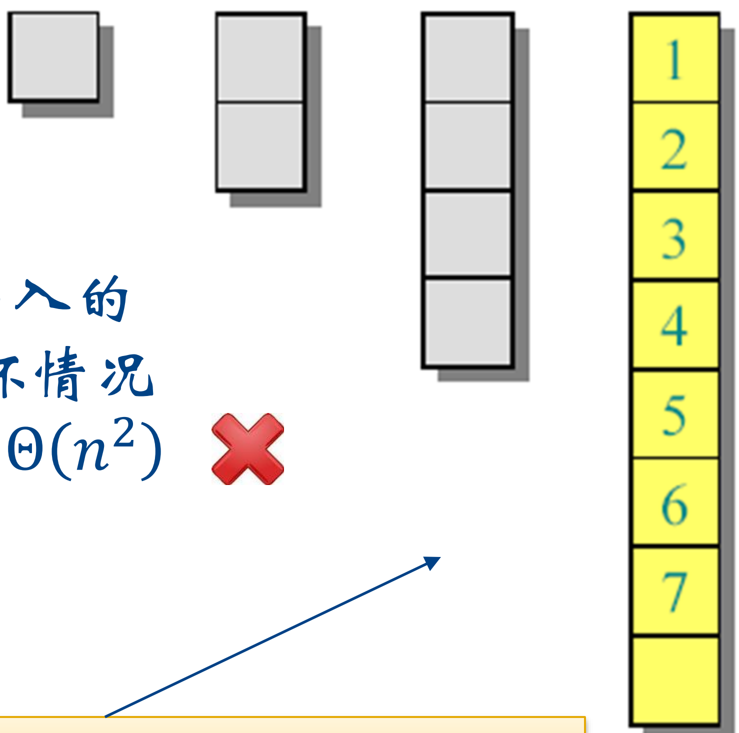
摊还分析

摊还分析

- 摊还分析 (Amortized analysis)
  - 求取数据结构的一个操作序列中所执行的所有操作的平均时间，来评价操作的代价
  - 不同于平均情况分析，它并不涉及概率，可以保证最坏情况下每个操作的平均性能
  - 即使操作序列中某个单一操作的代价很高，但其平均代价可能是很低的
- 基本方法
  - 聚合分析 (aggregate analysis)
  - 核算法 (accounting method)
  - 势能法 (potential method)

动态表

- 动态表
  - 表格大小根据插入的数据动态分配大小
  - 假定按 2 的幂次进行分配
- 最坏情况下的分析
  - 考虑插入 n 个元素，且单次插入的最坏情况为 $\Theta(n)$，因此，最坏情况下的总插入代价为 $n \cdot \Theta(n) = \Theta(n^2)$
  - 实际总代价为 $\Theta(n)$



INSERT: 1, 2, 3, 4, 5, 6, 6, 5, ….

聚合分析

- n 个操作序列则最坏情况下的总花费 $T(n)$
  - 最坏情况下，单个操作的摊还代价为 $T(n)/n$
- 动态表插入操作的分析
  - 定义第 i 个插入操作的代价 c_i 为

$$c_i = \begin{cases} i & \text{如果 } i \text{ 首次是2的幂次} + 1 \\ 1 & \text{其它} \end{cases}$$

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1

聚合分析

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	1 1	1 2	1	1 4	1	1	1	1 8	1

- 总代价

$$\sum_{i=1}^{n} c_i \le n + \sum_{j=0}^{\lfloor \log n - 1 \rfloor} 2^j \le 3n = \Theta(n)$$

- 摊还代价

$$\frac{\Theta(n)}{n} = \Theta(1)$$

核算法

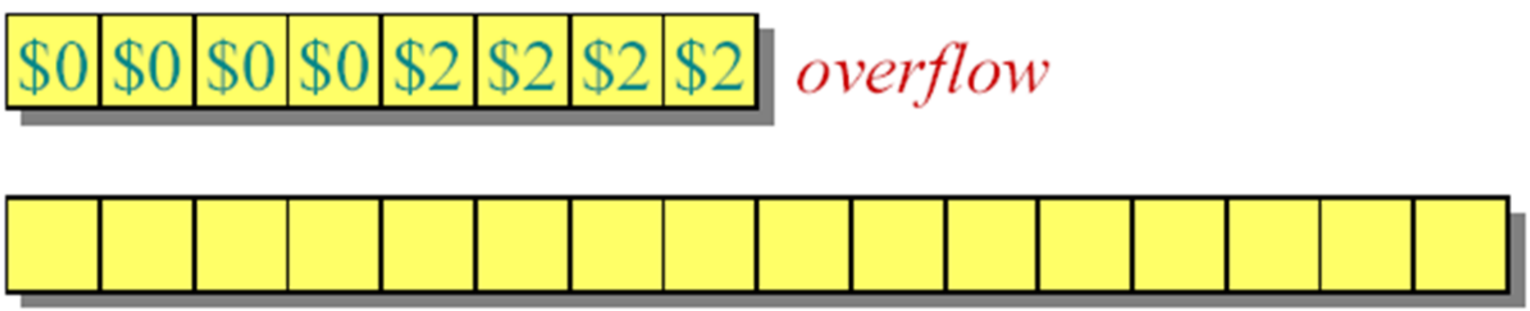
- Accounting method
  - 对不同的操作赋予不同费用(称之为摊还代价)，其赋予费用可能多于或少于其实际代价
  - 当摊还代价超出实际代价时，将差额(即**信用**)存入，当后续摊还代价小于实际代价时，信用可用来支付差额
  - 不同于聚合分析中所有操作都赋予相同的摊还代价
- 摊还代价的赋予
  - 确保操作序列的总摊还代价是总真实代价的**上界**
  - 保持数据结构中的**总信用**永远非负
  - 第i个操作的摊还代价表示为$\hat{c}_i$，真实代价为c_i，则

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

核算法

❖ 动态表分析

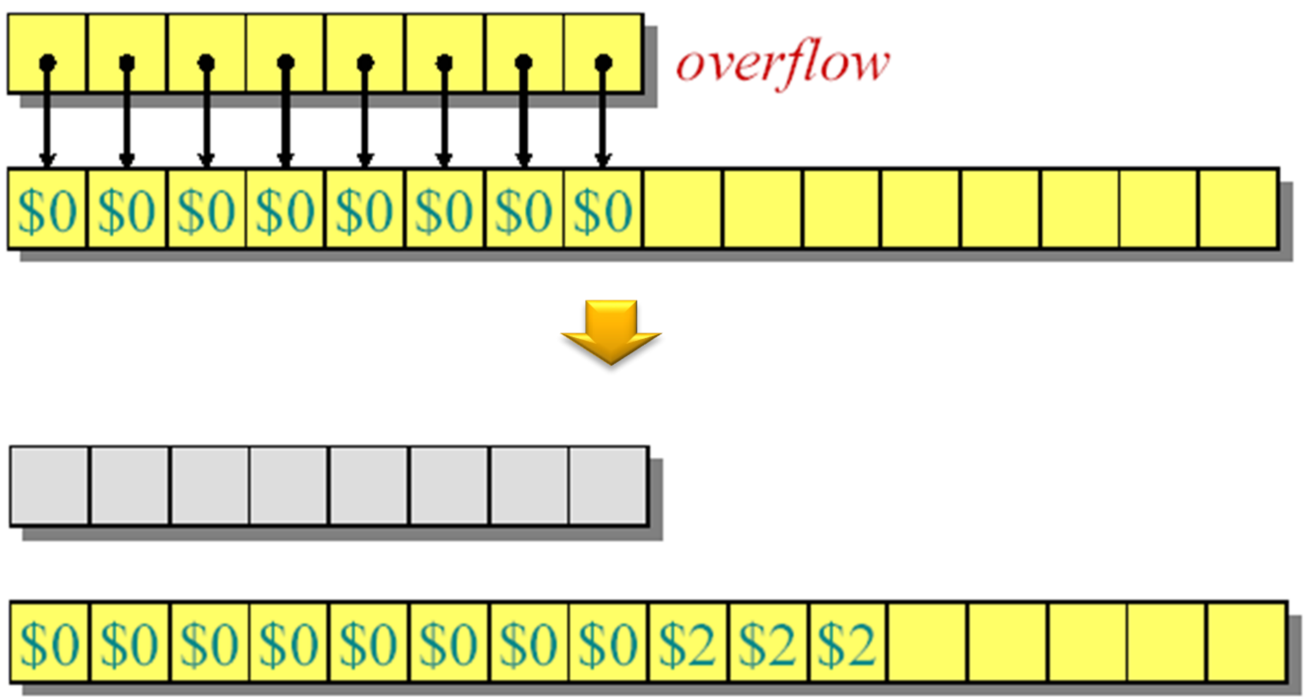
- 赋予每个插入操作3元，即$\hat{c}_i = 3$，其中
  - 1 元用于支付当前的插入操作
  - 2 元用于存入用于后续的表格翻倍处理
- 当表格翻倍时，1 元用于移动最新项，1 元用于移动旧项



核算法

❖ 动态表分析

- 当表格翻倍时，1 元用于移动最新项，1 元用于移动旧项



核算法

❖ 关键点分析

- 存入的信用永远是非负的，因而摊还代价的总和提供了实际代价的一个上界

i	1	2	3	4	5	6	7	8	9	10
$size_i$	1	2	4	4	8	8	8	8	16	16
c_i	1	2	3	1	5	1	1	1	9	1
$\hat{c}_i$	2*	3	3	3	3	3	3	3	3	3
$bank_i$	1	2	2	4	2	4	6	8	2	4

势能法

- 将动态集合存入银行账户里的总信用作为势能
  - 与整个数据结构关联，而不是特定的对象
  - 势能释放即可用来支付未来操作的代价
- 框架
  - 初始数据结构为 D_0
  - 操作 i 将数据结构从D_{i-1}变换成D_i
  - 操作 i 的代价为 c_i
  - 定义**势能函数** Φ: $\{D_i\} \rightarrow \mathcal{R}$, 且 $\Phi(D_0) = 0, \Phi(D_i) \geq 0$
  - 对应Φ的**摊还代价**$\hat{c}_i$ 定义为

    $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$

势能法

- 对势能的理解
  - 如果$\Delta\Phi_i > 0$，则$\hat{c}_i > c_i$，操作 i 在数据结构中存入能量以便以后使用
  - 如果$\Delta\Phi_i < 0$，则$\hat{c}_i < c_i$，数据结构为操作 i 提供能量以执行

$$\hat{c}_i = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\textit{potential difference } \Delta\Phi_i}$$

势能法

- 对势能的理解
  - n 个操作的总摊还代价为

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} \left(c_i + \Phi(D_i) - \Phi(D_{i-1}) \right)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

$$\geq \sum_{i=1}^{n} c_i \qquad \text{since } \Phi(D_n) \geq 0 \text{ and } \Phi(D_0) = 0.$$

势能法

- 动态表的分析
  - 定义势能函数为 $\Phi(D_i) = 2 \cdot i - 2^{[\log i]}$
  - 假定 $2^{[\log 0]} = 0$
  - 第 i 个插入操作的摊还代价为：

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= \begin{cases}
i + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right) \\
\qquad \text{if } i-1 \text{ is an exact power of } 2, \\
1 + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right) \\
\qquad \text{otherwise.}
\end{cases}
\end{aligned}
$$

势能法

- Case 1

$$\begin{aligned}\hat{c}_i &= i + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)\\ &= i + 2 - \left(2^{\lceil \lg i \rceil} - 2^{\lceil \lg (i-1) \rceil}\right)\\ &= i + 2 - \left(2(i-1) - (i-1)\right)\\ &= i + 2 - 2i + 2 + i - 1\\ &= 3\end{aligned}$$

- Case 2

$$\begin{aligned}\hat{c}_i &= 1 + \left(2i - 2^{\lceil \lg i \rceil}\right) - \left(2(i-1) - 2^{\lceil \lg (i-1) \rceil}\right)\\ &= 1 + 2 - \left(2^{\lceil \lg i \rceil} - 2^{\lceil \lg (i-1) \rceil}\right)\\ &= 3\end{aligned}$$

动态表

❖ 性能分析

- 除插入(扩展操作)外，收缩操作可以进行类似分析
- 在一个动态表上执行任意 n 个操作的实际运行时间为 $O(n)$
  - 每个操作的摊还代价的上界是一个常数

Next

- 递归与分治策略
  - Recursion
  - Divide and Conquer