算法的渐进复杂性

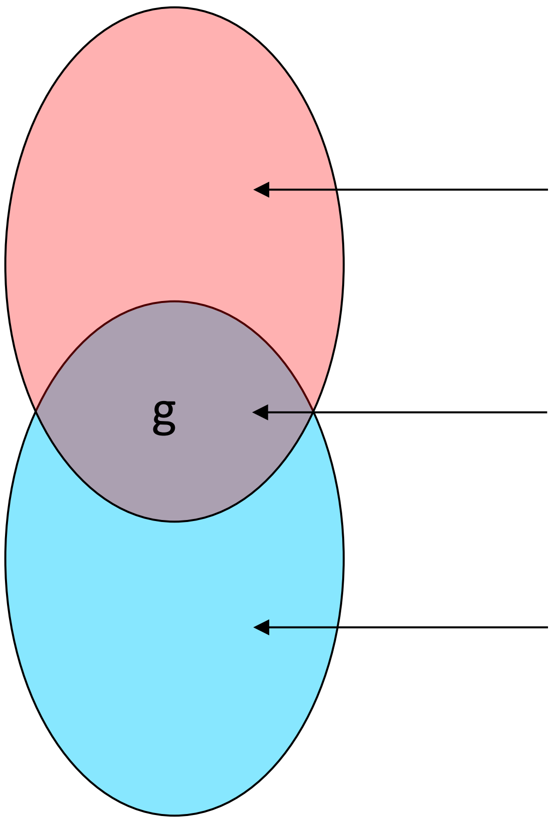
南京大学计算机系

赵建华

算法效率的比较方式

- 算法的效率通过它执行的关键操作的数量来度量，但是
  - 对于不同的具体输入，算法所需要的关键操作数量有所不同
  - 同样的操作在不同的计算机上需要的时间是不同的
- 算法复杂性表示为输入规模n的函数f(n)：平均情况，最坏情况
- 比较复杂性时主要看f(n)的渐进增长率（Asymptotic Growth Rate）
  - 如果算法A和算法B所需要的关键操作数量分别是f1(n)和f2(n)，A优于B是指当n足够大时，f1(n)必然小于f2(n)。

Relative Growth Rate



Ω(g):functions that grow at least as fast as g

Θ(g):functions that grow at the same rate as g

O(g):functions that grow no faster as g

"Big Oh"

- Basic idea
  - $f(n) \in O\big(g(n)\big)$ if for sufficiently large input n, g(n) ≥ f(n)

- Definition – "$\boldsymbol{\varepsilon - N}$"
  - Giving g: $N \rightarrow R^+$, then $O(g)$ is the set of f:$N \rightarrow R^+$, such that for some $c \in R^+$ and some $n_0 \in N$, $f(n) \leq cg(n)$ for all $n \geq n_0$

Example

- Let $f(n)=n^2$, $g(n)=n\log n$, then:
  - $f \notin O(g)$, since

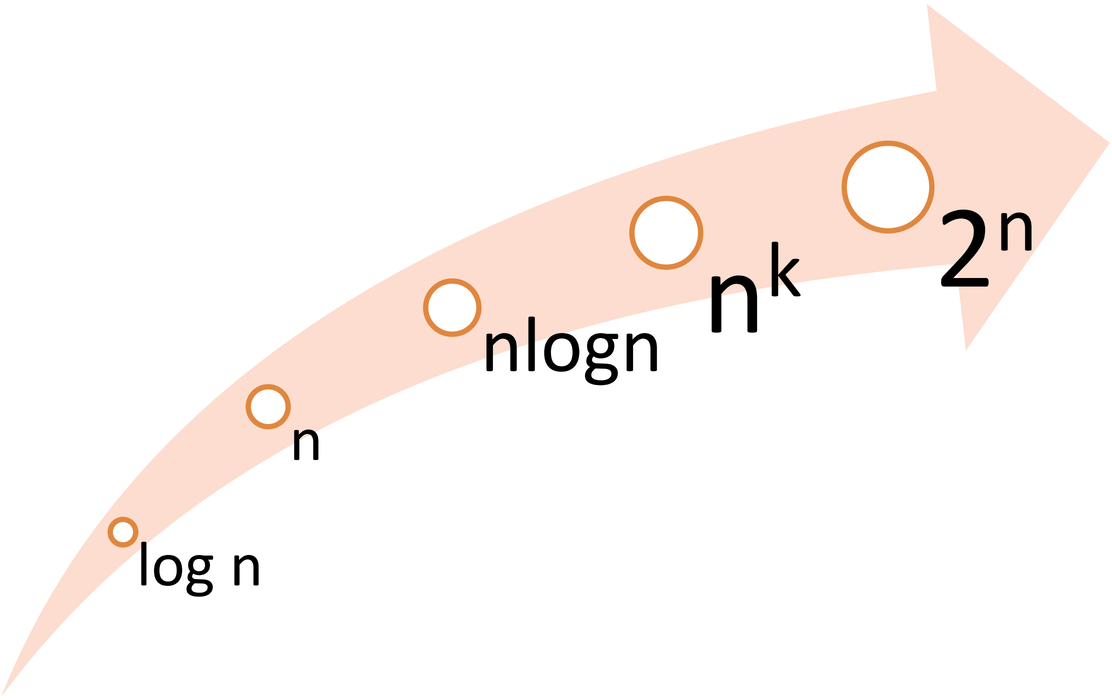
L'Hospital's rule

$$\lim_{n\to\infty} \frac{n^2}{n\log n} = \lim_{n\to\infty} \frac{n}{\log n} = \lim_{n\to\infty} \frac{1}{\frac{1}{n\ln 2}} = +\infty$$

  - $g \in O(f)$, since

$$\lim_{n\to\infty} \frac{n\log n}{n^2} = \lim_{n\to\infty} \frac{\log n}{n} = \lim_{n\to\infty} \frac{1}{n\ln 2} = 0$$

Asymptotic Growth Rate



Asymptotic Order

- Logarithm *log n*

$$\log n \in O(n^{\alpha}) \text{ for any } \alpha > 0$$

- Power n^k

$$n^{\mathrm{k}} \in O(c^n) \text{ for any } c > 1$$

- Factorial *n!*

$$\boldsymbol{n}! \approx \sqrt{\boldsymbol{2\pi n}}\left(\frac{\boldsymbol{n}}{\boldsymbol{e}}\right)^{\boldsymbol{n}} \text{(Stirling's formula)}$$

"Big Ω"

- Basic idea
  - $f(n) \in \Omega\big(g(n)\big)$ if for sufficiently large input n, $f(n) \geq g(n)$
  - Dual of "O"

- Definition – "$\boldsymbol{\varepsilon - N}$"
  - Giving g: $N \rightarrow R^+$, then $\Omega(g)$ is the set of f:$N \rightarrow R^+$, such that for some $c \in R^+$ and some $n_0 \in N$, $f(n) \geq c g(n)$ for all $n \geq n_0$

The Set Θ

- Basic idea of $f(n) \in \Theta(g(n))$
  - Roughly the same
  - $\Theta(g) = O(g) \cap \Omega(g)$
- Definition – “$\boldsymbol{\varepsilon - N}$”
  - Giving g:N→R^+, then $\Theta(g)$is the set of f:N→R^+, such that for some $c_1, c_2 \in R^+$ and some $n_0 \in N$,

    $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, for all $n \geq n_0$

Some Empirical Data

algorithm		1	2	3	4
Run time in *ns*		$1.3n^3$	$10n^2$	$47n\log n$	$48n$
time for size	10^3	$1.3s$	$10ms$	$0.4ms$	$0.05ms$
	10^4	$22m$	$1s$	$6ms$	$0.5ms$
	10^5	$15d$	$1.7m$	$78ms$	$5ms$
	10^6	$41yrs$	$2.8hrs$	$0.94s$	48ms
	10^7	$41mill$	$1.7wks$	$11s$	$0.48s$
max Size in time	sec	920	10,000	1.0×10^6	2.1×10^7
	min	3,600	77,000	4.9×10^7	1.3×10^9
	hr	14,000	6.0×10^5	2.4×10^9	7.6×10^{10}
	day	41,000	2.9×10^6	5.0×10^{10}	1.8×10^{12}
time for 10 times size		$\times1000$	$\times100$	$\times10+$	$\times10$

on 400Mhz Pentium II, in C
from: Jon Bentley: *Programming Pearls*

Properties of O, Ω and Θ

- Transitive property:
  - If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$
- Symmetric properties
  - $f \in O(g)$ if and only if $g \in \Omega(f)$
  - $f \in \Theta(g)$ if and only if $g \in \Theta(f)$
- Order of sum function
  - $O(f+g) = O(\max(f,g))$

“Little Oh”

- Basic idea of $f(n) \in o(g(n))$
  - Non-ignorable gap between f and its upper bound g
- Definition –“$\boldsymbol{\varepsilon - N}$”
  - Giving g:N→R$^+$, then o(g) is the set of f:N→R$^+$, such that for any c∈R$^+$, there exists some n_0∈N,

$$0 \leq f(n) < cg(n), \text{ for all } n \geq n_0$$

对比Big Oh的定义
Giving g: N→R$^+$, then $O(g)$ is the set of f:N→R$^+$, such that for some c∈R$^+$ and some n_0∈N, f(n)≤cg(n) for all n≥n_0

"Little ω"

- Basic idea of $f(n) \in \omega(g(n))$
  - Dual of "o"
- Definition – "$\boldsymbol{\varepsilon} - \boldsymbol{N}$"
  - Giving g:N→R$^+$, then ω(g) is the set of f:N→R$^+$, such that for any c∈R$^+$, there exists some n_0∈N,

$$0 \le cg(n) < f(n), \text{ for all } n \ge n_0$$

对比Big Omiga的定义

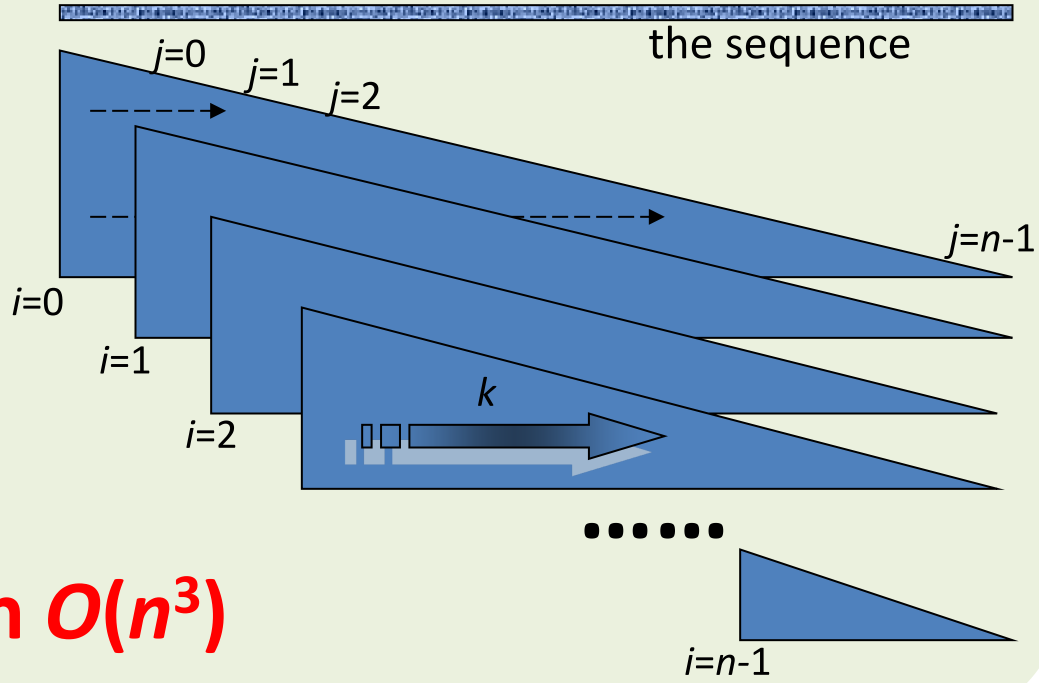
Giving g: N→R$^+$, then Ω(g) is the set of f:N→R$^+$, such that for some c∈R$^+$ and some n_0∈N, $f(n) \ge cg(n)$ for all $n \ge n_0$

Max-sum Subsequence

- The problem: Given a sequence S of integer, find the **largest sum** of a consecutive subsequence of S. (0, if all negative items)
  - An example: -2, 11, -4, 13, -5, -2; the result 20: (11, -4, 13)

A brute-force algorithm:

```
MaxSum = 0;
  for (i = 0; i < N; i++)
    for (j = i; j < N; j++)
    {
      ThisSum = 0;
      for (k = i; k <= j; k++)
            ThisSum += A[k];
      if (ThisSum > MaxSum)
            MaxSum = ThisSum;
    }
  return MaxSum;
```



in $O(n^3)$

More Precise Complexity

The total cost is : $\sum_{i=0}^{n-1}\sum_{j=i}^{n-1}\sum_{k=i}^{j} 1$

in $O(n^3)$

$$\sum_{k=i}^{j} 1 = j - i + 1$$

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

$$\sum_{i=0}^{n-1} \frac{(n - i + 1)(n - i)}{2} = \sum_{i=1}^{n} \frac{(n - i + 2)(n - i + 1)}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{n} i^2 - (n + \frac{3}{2})\sum_{i=1}^{n} i + \frac{1}{2}(n^2 + 3n + 2)\sum_{i=1}^{n} 1$$

$$= \frac{n^3 + 3n^2 + 2n}{6}$$

基本算法的改进

```
A brute-force algorithm:
MaxSum = 0;
  for (i = 0; i < N; i++)
    for (j = i; j < N; j++)
    {
      ThisSum = 0;
      for (k = i; k <= j; k++)
            ThisSum += A[k];
      if (ThisSum > MaxSum)
            MaxSum = ThisSum;
    }
  return MaxSum;
```

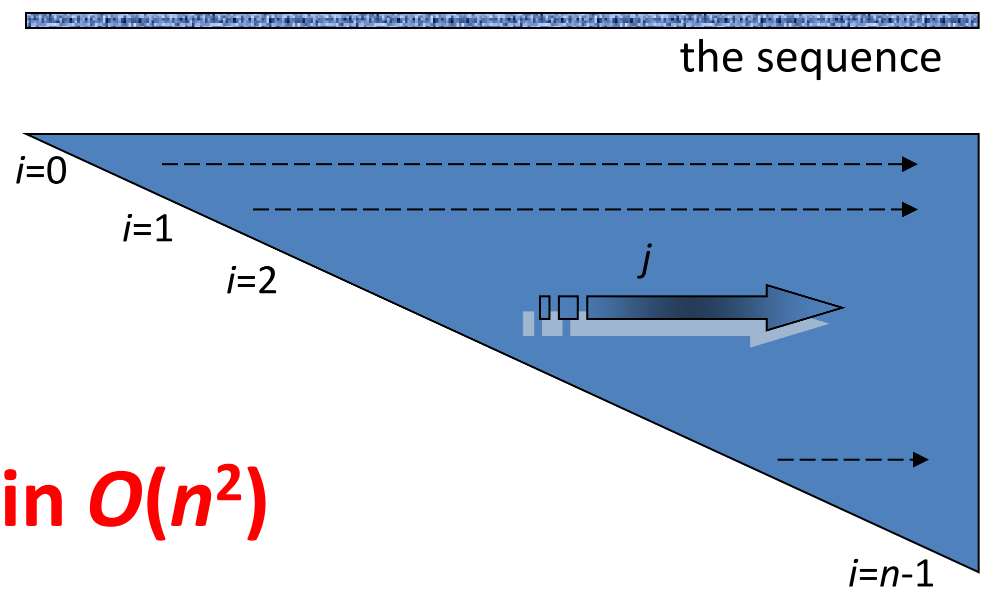
算法的里面两重循环计算了从i开始到各个连续区间的和。但是：

1. 当内层层循环计算完成A[i,j]的和之后，下一次迭代又从头开始计算A[i,j+1]的和。
2. 实际上我们只需要记住A[i,j]的和，再加上A[j+1]即可。
3. 也就是说，在第二层循环的每次迭代之后记住ThisSum，然后在加上A[j+1]即可。

Decreasing the Number of Loops

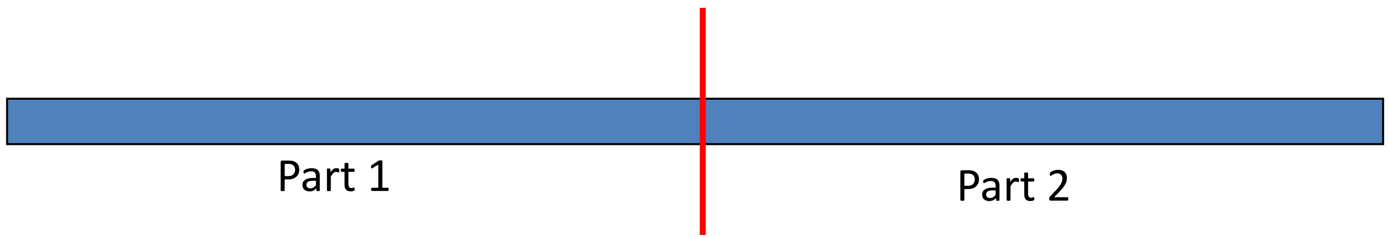
An improved algorithm:

```
MaxSum = 0;
for (i = 0; i < N; i++)
{
  ThisSum = 0;
  for (j = i; j < N; j++)
  {
       ThisSum += A[j];
       if (ThisSum > MaxSum)
           MaxSum = ThisSum;
  }
}
return MaxSum;
```

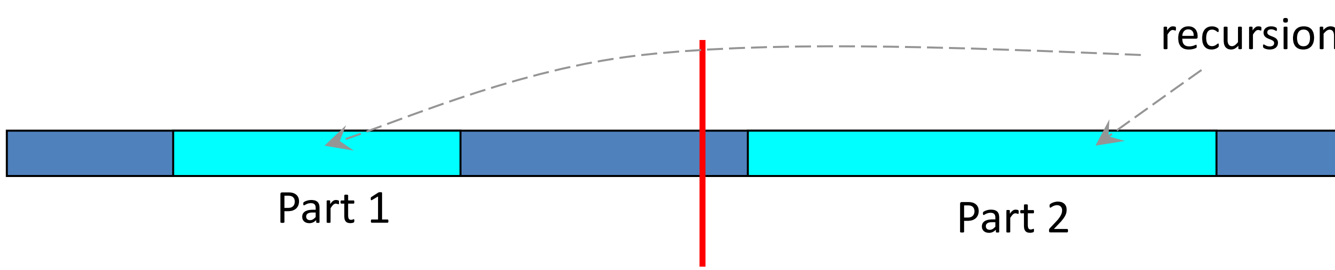


in $O(n^2)$

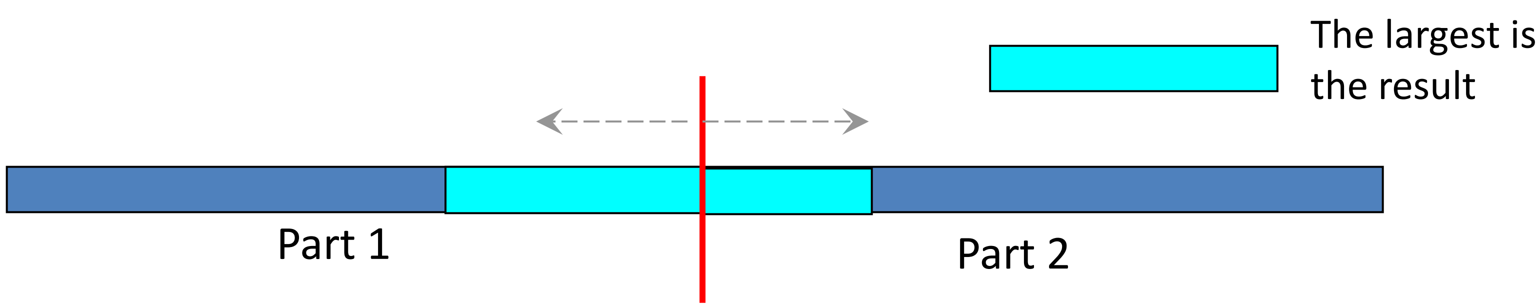
递归的思路：一分为二（1）



the sub with largest sum may be in:



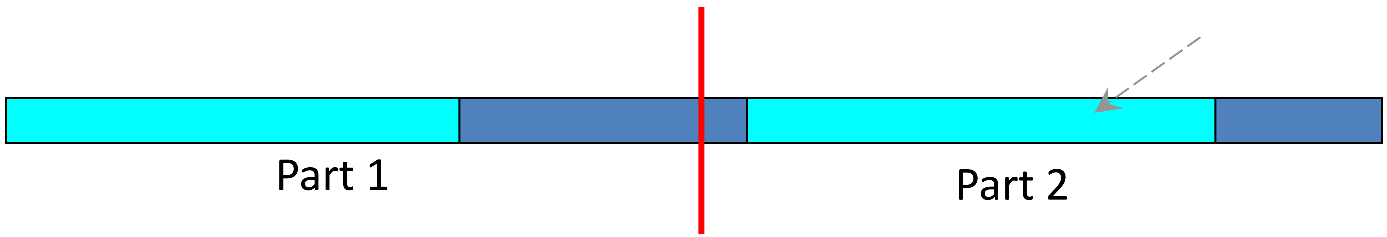
or:



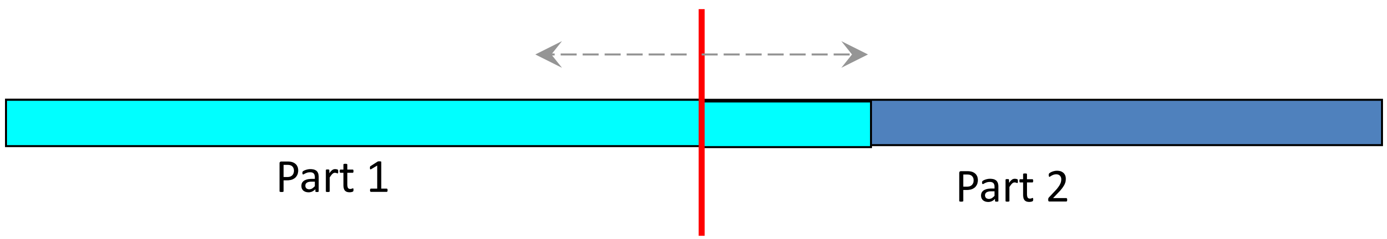
因此，对于每个分段，还需要计算从左边开始的最大值，和从右边开始的最大值

递归的思路：一分为二（2）

- 从左边开始的最大值可能是



or:



第二种情况：需要考虑左边部分全部元素的和

- 从右边部分开始的最大值类似

递归的思路：一分为二（3）

- All.sum = Left.sum + Right.sum
- All.leftMax = Max(Left.leftMax, Left.sum + Right.leftMax)
- All.rightMax = Max(Right.rightMax, Right.sum + Left.rightMax)
- All.maxSum =
  Max(Left.maxSum, Right.maxSum, Left.rightMax + Right.leftMax)

```
struct Result{
    int sum;
    int leftMax;
    int rightMax;
    int maxSum;
}
```

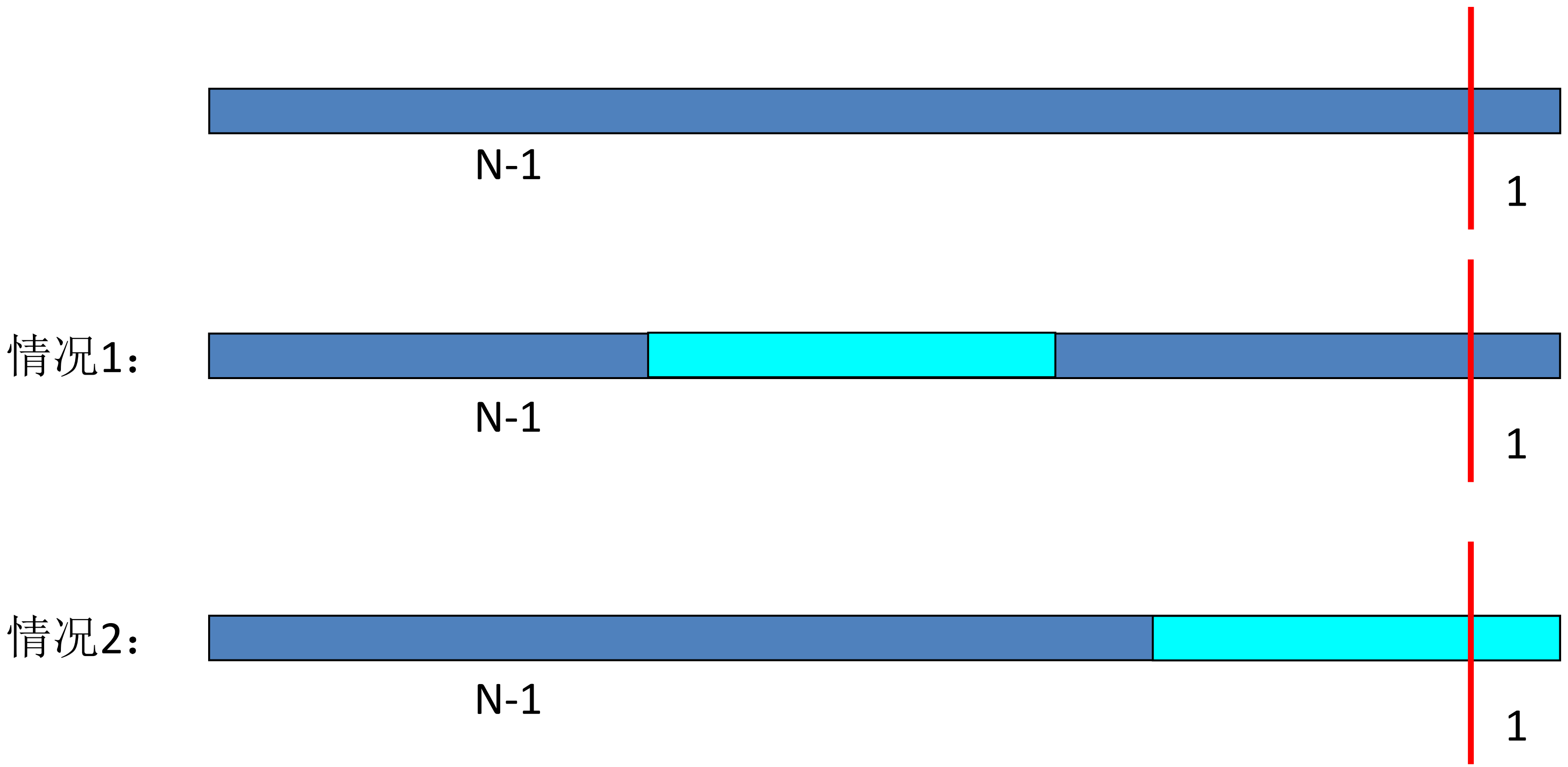
```
GetMaxSub(int start, int end)
{  Result ret;
   if(end == start + 1)//只有一个元素
   {
       ret.sum = A[start];
       ret.leftMax = A[start] > 0 ? A[start] : 0;
       ret.rightMax = A[start] > 0 ? A[start] : 0;
       ret.maxSum = A[start] > 0 ? A[start] : 0;
       return ret;
   }
   … … …
}
```

递归的思路：一分为二（4）

```
GetMaxSub(int start, int end)
{  Result ret;
   if(end == start + 1)//只有一个元素
   { … … ;return ret;}
   int center = (start + end)/2;
   Result lRet = GetMaxSum(start, center);
   Result rRet = GetMaxSum(center, end);
   ret.sum = lRet.sum + rRet.sum
   ret.leftMax = Max(lRet.leftMax, lRet.sum + rRet.leftMax)
   ret.rightMax = Max(rRet.rightMax, rRet.sum + lRet.rightMax)
   ret.maxSum =
      Max(lRet.maxSum, rRet.maxSum, lRet.rightMax + rRet.leftMax)
    return ret;
}
```

这个递归程序的复杂性是多少？

递归思路：1和N-1（1）



- 假设分成A[N-1]和A[0,N-2]，会有两种情况：
  - 最大子区间就是A[0,N-1)的最大子区间
  - 最大子区间包含了A[N-1]，也就是A[N-1] + A[0,N-1)的靠右的最大子区间

递归思路：1和N-1（2）

```
struct ReturnValue {
    int max;
    int rightMax;
}
```

```
MaxSubSum(int N)
{
    if(N == 0)
            return ReturnValue(0,0);
    ReturnValue r = MaxSubSum(N-1);
    ReturnValue ret;
    ret.rightMax = Max(0, r.rightMax + A[N-1]);
    ret.max = Max(r.max, ret.rightMax);
    return ret;
}
```

递归思路：1和N-1（迭代化）

```
rightMax = MaxSum = 0;
for (j = 0; j < N; j++)
{
    rightSum += A[j];
    if (rightSum < 0)
            rightSum = 0;

    if (MaxSum < rightSum)
            MaxSum = rightSum;
}
return MaxSum;
```

复杂性 $O(n)$

Recursion in Algorithm Design

- Counting the Number of Bits
  - Input: a positive decimal integer n
  - Output: the number of binary digits in n's binary representation

Int BitCounting (int n)
1. If(n==1) return 1;
2. Else
3. return BitCounting(n div 2) +1;

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

Analysis of Recursion

- Solving recurrence equations

- E.g., Bit counting
  - Critical operation: add
  - The recurrence relation

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + 1 & n > 1 \end{cases}$$

Analysis of Recursion: Backward substitutions

By the recursion equation : $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1$

For simplicity , let $n = 2^k$ (k is a nonnegativ e integer),

that is, $k = \log n$

$$T(n) = T\left(\frac{n}{2}\right) + 1 = T\left(\frac{n}{4}\right) + 1 + 1 = T\left(\frac{n}{8}\right) + 1 + 1 + 1 = \ldots\ldots$$

$$T(n) = T\left(\frac{n}{2^k}\right) + \log n = \log n \quad (T(1)=0)$$

Computing the Fibonacci Number

T(0)=0

T(1)=1

T(n)= *T(n-1)*+*T(n-2)*

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$a_n = r_1 a_{n-1} + r_2 a_{n-2} + \cdots + r_m a_{n-k}$$

is called linear homogeneous relation of degree k.

For the special case of Fibonacci: $a_n = a_{n-1} + a_{n-2}$, $r_1 = r_2 = 1$

Characteristic Equation

- For a linear homogeneous recurrence relation of degree k

$$a_n = r_1 a_{n-1} + r_2 a_{n-2} + \cdots + r_k a_{n-k}$$

the polynomial of degree k

$$x^k = r_1 x^{k-1} + r_2 x^{k-2} + \cdots + r_k$$

is called its characteristic equation.

- The characteristic equation of linear homogeneous recurrence relation of degree 2 is:

$$\boxed{x^2 - r_1 x - r_2 = 0}$$

Solution of Recurrence Relation

- If the characteristic equation $x^2 - r_1 x - r_2 = 0$ of the recurrence relation $a_n = r_1 a_{n-1} + r_2 a_{n-2}$ has two distinct roots s_1 and s_2, then

$$a_n = u s_1^n + v s_2^n$$

where u and v depend on the initial conditions, is the explicit formula for the sequence.

- If the equation has a single root s, then, both s_1 and s_2 in the formula above are replaced by s

Proof of the Solution

$$a_n = us_1^n + vs_2^n$$

S_1, S_2是$x^2 - r_1 x - r_2 = 0$的根

To prove that ：

$$us_1^n + vs_2^n = r_1 a_{n-1} + r_2 a_{n-2}$$

证：$us_1^n + vs_2^n = us_1^{n-2} s_1^2 + vs_2^{n-2} s_2^2$

$$= us_1^{n-2}(r_1 s_1 + r_2) + vs_2^{n-2}(r_1 s_2 + r_2)$$

$$= r_1 us_1^{n-1} + r_2 us_1^{n-2} + r_1 vs_2^{n-1} + r_2 vs_2^{n-2}$$

$$= r_1(us_1^{n-1} + vs_2^{n\ 1}) + r_2(us_1^{n-2} + vs_2^{n-2})$$

$$= r_1 a_{n-1} + r_2 a_{n-2}$$

Back to Fibonacci Sequence

$f_0=0$

$f_1=1$

$f_n = f_{n-1} + f_{n-2}$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ……

Explicit formula for Fibonacci Sequence

The characteristic equation is x^2-x-1=0, which has roots:

$$s_1 = \frac{1+\sqrt{5}}{2} \quad and \quad s_2 = \frac{1-\sqrt{5}}{2}$$

Note: (by initial conditions) $f_1 = us_1 + vs_2 = 1 \quad and \quad f_2 = us_1^{\,2} + vs_2^{\,2} = 1$

which means:

$$f_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Guess and Prove

将规模为n的问题分解成两个规模为n/2的子问题递归地求解，并且分解/合成时的总开销是O(n)的。

- Example: $T(n)=2T(\lfloor n/2 \rfloor) + n$
- Guess
  - $T(n) \in O(n)$?
    - $T(n) \le cn$, to be pro
  - $T(n) \in O(n^2)$?
    - $T(n) \le cn^2$, to be proved fo
  - **Or maybe**, $T(n) \in O(n\text{lo}$
    - $T(n) \le cn\log n$, to be prove
- Prove
  - by substitution

Try to prove $T(n) \le cn$:

$$T(n)=2T(\lfloor n/2 \rfloor)+n \le 2c(\lfloor n/2 \rfloor)+n$$
$$\le 2c(n/2)+n = (c+1)n \quad \textbf{Fail!}$$

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor)+n \\
&\le 2(c\lfloor n/2 \rfloor \log (\lfloor n/2 \rfloor))+n \\
&\le cn\log (n/2)+n \\
&= cn \log n - cn \log 2 + n \\
&= cn \log n - cn + n \\
&\le cn \log n \quad \text{for } c \ge 1
\end{aligned}
$$

这里C必须是明确给出的值

Divide and Conquer Recursions

- Divide and conquer
  - Divide the “big” problem to smaller ones
  - Solve the “small” problems by recursion
  - Combine results of small problems, and solve the original problem
- Divide and conquer recursion

将规模为n的问题分解为b个规模为n/c的子问题并递归地求解，并且分解/合成时的总开销是f(n)的。

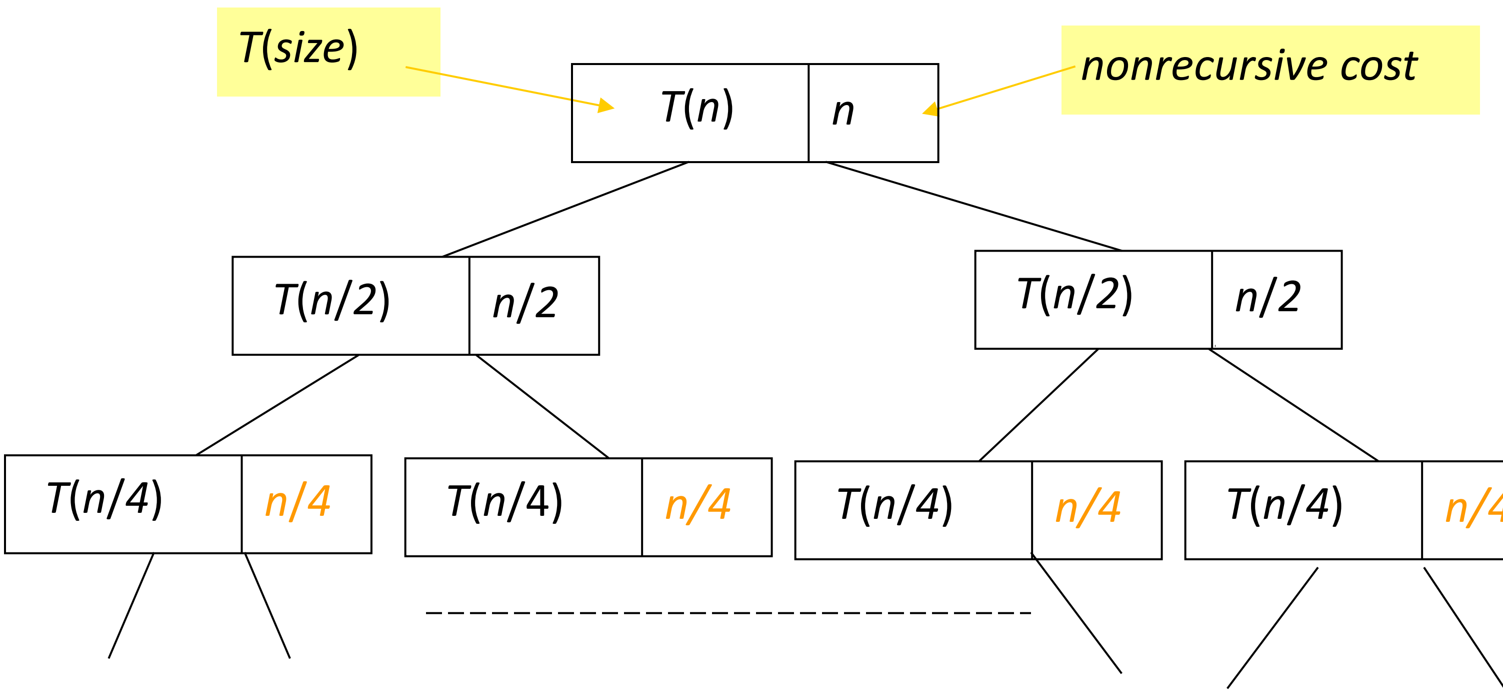
$$T(n) = a\,T(n/b) + f(n)$$

divide

Divide/combine

conquer

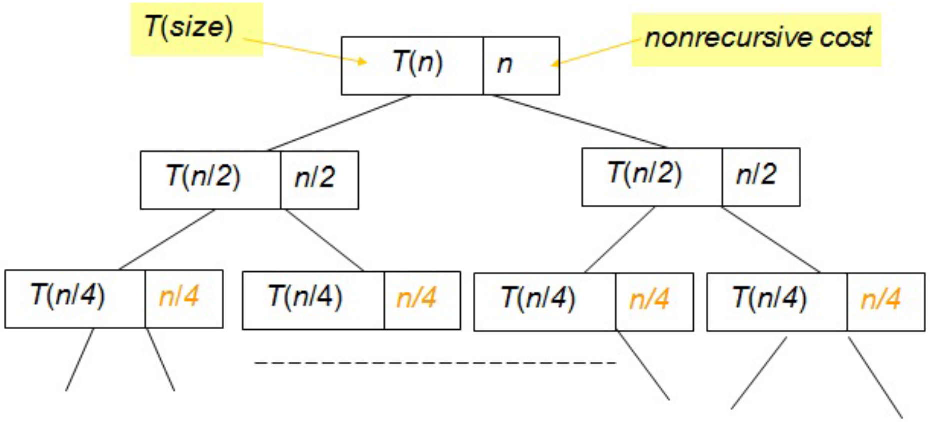
Recursion Tree



The recursion tree for $T(n) = 2T(n/2) + n$

Recursion Tree

- Node
  - Non-leaf
    - Non-recursive cost
    - Recursive cost
  - Leaf
    - Base case
- Edge
  - Recursion



The recursion tree for T(n)=T(n/2)+T(n/2)+n

Recursion Tree

- $T(n) = \overbrace{3T(n/4)}^{\text{Recursive cost}} + \overbrace{\Theta\ (n^2)}^{\text{Non-recursive cost}}$

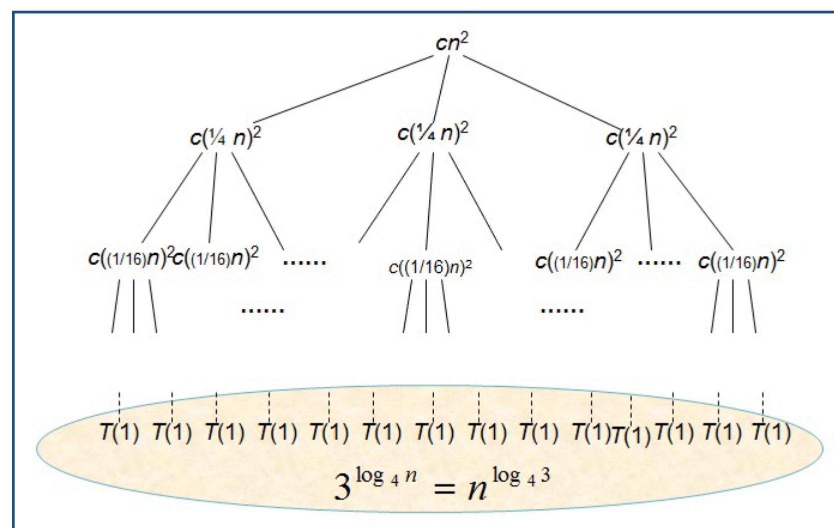
# of sub-problems

size of sub-problems

- Total cost

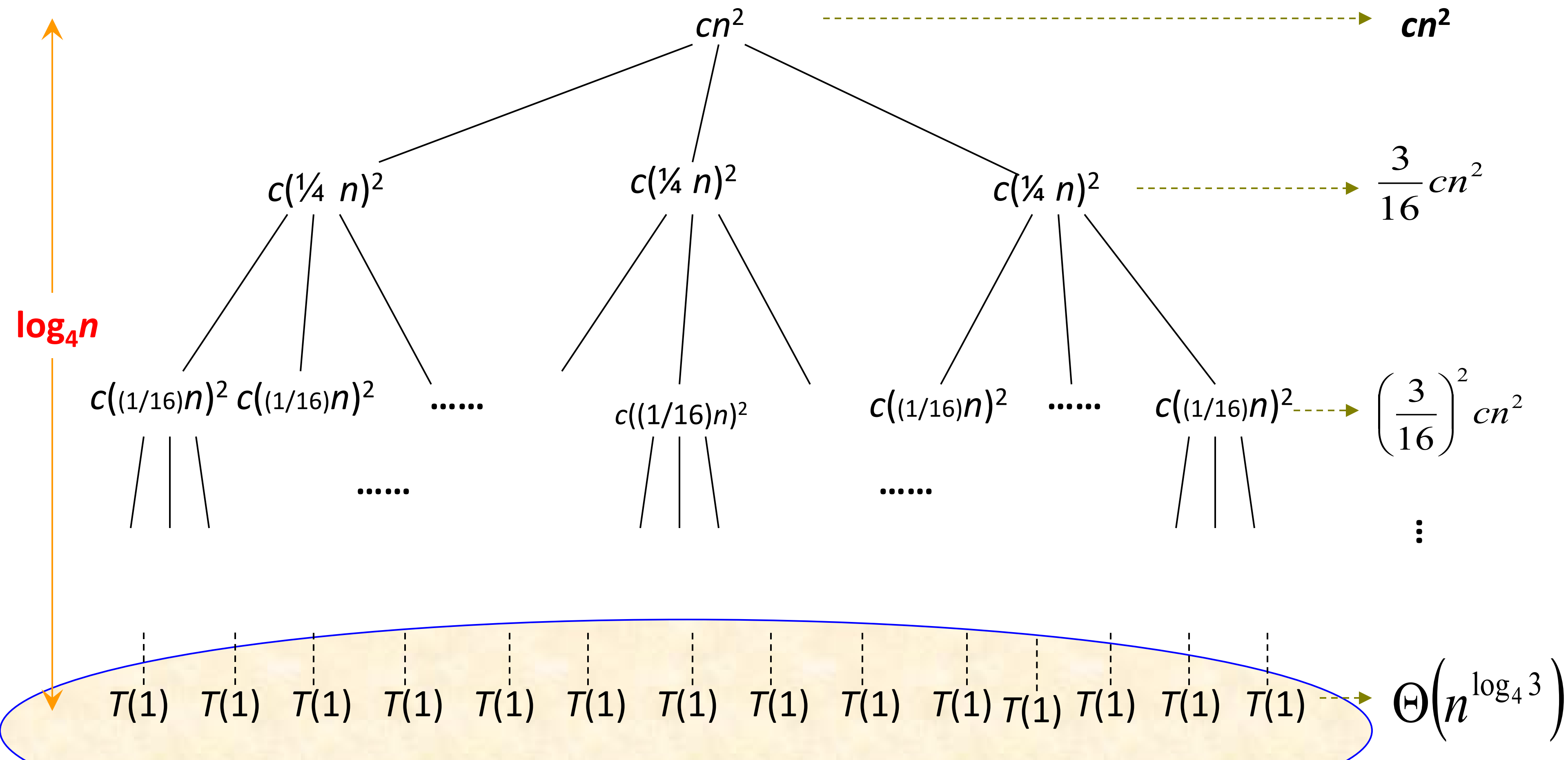
Σ

Sum of row sums



Sum of Row-sums

$T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$



Note: $3^{\log_4 n} = n^{\log_4 3}$

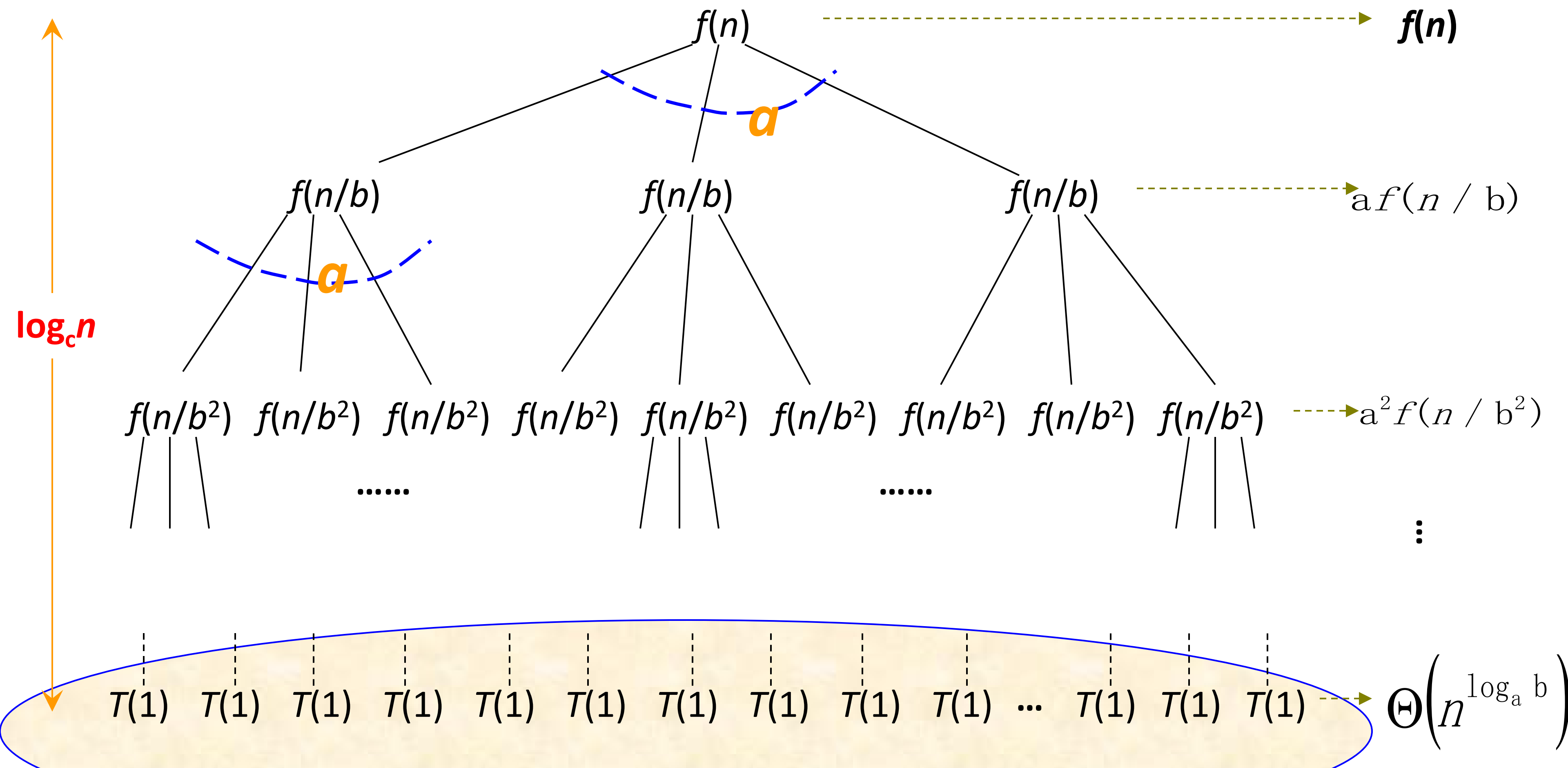
Total: $\Theta(n^2)$

Solving the Divide-and-Conquer Recurrence

- The recursion equation for divide-and-conquer, the general case: $T(n)=aT(n/b)+f(n)$
- Observations:
  - Let base-cases occur at depth D(leaf), then $n/b^{D}=1$, that is $D=\log(n)/\log(b)$
  - Let the number of leaves of the tree be L, then $L=a^{D}$, that is $L=a^{(\log(n)/\log(b))}$.
  - By a little algebra: $L=n^{E}$, where $E=\log(a)/\log(b)$, called *critical exponent*.

Recursion Tree for

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



Note: $a^{\log_b n} = n^{\log_a b}$ ***Total ?***

Divide-and-Conquer：the Solution

- The solution of divide-and-conquer equation is the non-recursive costs of all nodes in the tree, which is the sum of the row-sums
  - The recursion tree has depth $D=\log(n)/\log(b)$, so there are about that many row-sums.

- The 0^{th} row-sum
  - is $f(n)$, the nonrecursive cost of the root.
- The D^{th} row-sum
  - is n^E, assuming base cases cost 1, or $\Theta(n^E)$ in any event.

Solution by Row-sums

- [Little Master Theorem] Row-sums decide the solution of the equation for divide-and-conquer:
  - Increasing geometric series: $T(n) \in \Theta(n^E)$
  - Constant: $T(n) \in \Theta(f(n) \log n)$
  - Decreasing geometric series: $T(n) \in \Theta(f(n))$

This can be generalized to get a result not using explicitly row-sums.

Master Theorem

将f(n)和$n^{\log_b a}$进行比较
1、f(n)比较大，则选f(n)
2、$n^{\log_b a}$比较大，则选$n^{\log_b a}$
3、一致，则是$n \log n$

– Case 1: $f(n) \in O(n^{\log_b a-\varepsilon})$, ($\varepsilon>0$), then:

$$T(n) \in \Theta(n^{\log_b a})$$

– Case 2: $f(n) \in \Theta(n^{\log_b a})$, as all node depth contribute about equally:

$$T(n) \in \Theta(f(n)\log(n))$$

– case 3: $f(n) \in \Omega(n^{\log_b a+\varepsilon})$, ($\varepsilon>0$), and if $af(n/b) \leq \theta f(n)$ for some constant $\theta < 1$ and all sufficiently large n, then:

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

Using Master Theorem

Case 1: $f(n) \in O(n^{\log_b a - \varepsilon})$, ($\varepsilon > 0$), then:

$$T(n) \in \Theta(n^{\log_b a})$$

例子： $T(n) = 9T(n/3) + n$， 其中： $a = 9, b = 3, \log_b a = 2, f(n) = n \in O(n^{2-1})$

因此： $T(n) = $ Theta(n2)

Case 2: $f(n) \in \Theta(n^{\log_b a})$, as all node depth contribute about equally:

$$T(n) \in \Theta(f(n)\log(n))$$

例子： $T(n) = T(\frac{2n}{3}) + 1$， 其中： $a = 1, b = 3/2, \log_b a = 0, f(n) = 1 \in \Theta(n^0)$

因此： $T(n) = \Theta(\log n)$

Case 3: $f(n) \in \Omega(n^{\log_b a + \varepsilon})$, ($\varepsilon > 0$), and if $bf(n/c) \le \theta f(n)$ for some constant $\theta < 1$ and all sufficiently large n, then:

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

例子： $T(n) = 3T(n/4) + n\log n$， 其中： $a = 3, b = 4, \log_b a = \log_4 3$,

$f(n) = n\log n = \Omega(n^{\log_4 3 + \varepsilon})$, $3f(n/4) = 3((n/4)\log(n/4)) = (3/4)n\log n - 3/2*n$

因此: $T(n) = \Theta(n\log n)$

摊销分析
Amortized Analysis

Array Doubling

- Cost for search in a hash table is $\Theta(1+\alpha)$
  - If we can keep α constant, the cost will be $\Theta(1)$
- What if the hash table is more and more loaded?
  - Space allocation techniques such as array doubling may be needed
- The problem of "unusually expensive" individual operation

Looking at the Memory Allocation

- hashingInsert(HASHTABLE H, ITEM x)
- **integer** *size*=0, *num*=0;
- **if** *size*=0 **then** allocate a block of size 1; *size*=1;
- **if** *num*=*size* **then**
- allocate a block of size 2*size*;
- move all item into new table;
- *size*=2*size*;
- insert x into the table;
- *num*=*num*+1;
- **return**

Insertion with expansion: cost *size*

Elementary insertion: cost 1

Worst-case Analysis

- **For *n* execution of insertion operations**
  - A bad analysis: the worst case for one insertion is the case when expansion is required, up to n
  - So, the worst case cost is in $O(n^2)$.

Of course NOT!

- **Note the expansion is required during the *i*th operation only if $i=2^k$, and the cost of the *i*th operation**

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is exactly the power of 2} \\ 1 & otherwise \end{cases}$$

So the total cost is: $\sum_{i=1}^{n} c_i \leq n+\sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n+2n = 3n$

Amortized Analysis – Why?

- Unusually expensive operations
  - E.g., Insert-with-array-doubling
- Relation between expensive and usual operations
  - Each piece of the doubling cost corresponds to some previous insert

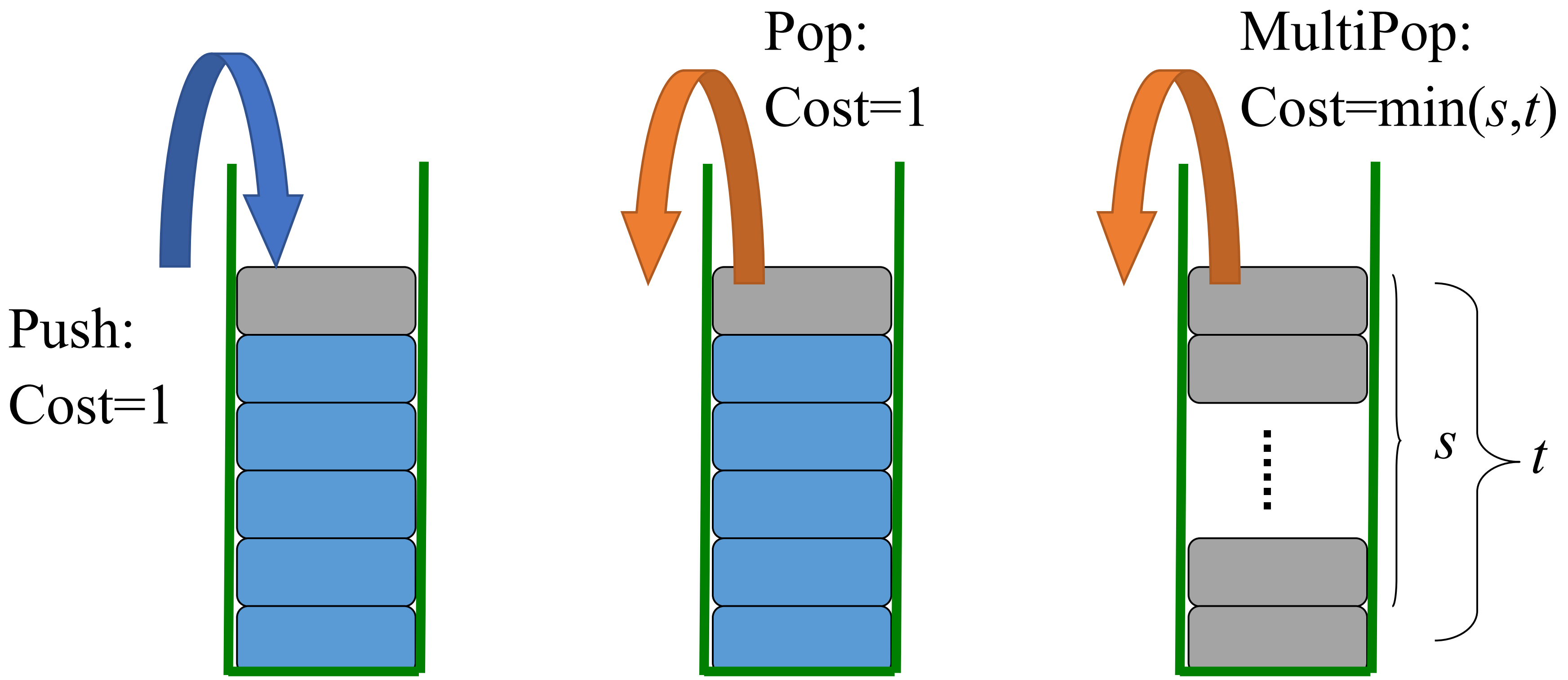
Amortized Analysis

- Amortized equation:

$$amortized\ cost = actual\ cost + accounting\ cost$$

- Design goals for accounting cost
  - In any legal sequence of operations, the sum of the accounting costs is nonnegative
  - The amortized cost of each operation is fairly regular, in spite of the wide fluctuate possible for the actual cost of individual operations

Amortized Analysis: MultiPop Stack



Amortized cost: push:2; pop, multipop: 0

Accounting Scheme for Stack Push

- Push operation with array doubling
  - No resize triggered: 1
  - Resize($n \rightarrow 2n$) triggered: $nt+1$ (t is a constant)
- Accounting scheme (specifying accounting cost)
  - No resize triggered: $2t$
  - Resize($n \rightarrow 2n$) triggered: $-nt+2t$
- So, the amortized cost of each individual push operation is $\mathbf{1+2t \in \Theta(1)}$

Amortized Analysis: Binary Counter

0	0 0 0 0 0 0 0 0	0
1	0 0 0 0 0 0 0 1	1
2	0 0 0 0 0 0 1 0	3
3	0 0 0 0 0 0 1 1	4
4	0 0 0 0 0 1 0 0	7
5	0 0 0 0 0 1 0 1	8
6	0 0 0 0 0 1 1 0	10
7	0 0 0 0 0 1 1 1	11
8	0 0 0 0 1 0 0 0	15
9	0 0 0 0 1 0 0 1	16
10	0 0 0 0 1 0 1 0	18
11	0 0 0 0 1 0 1 1	19
12	0 0 0 0 1 1 0 0	22
13	0 0 0 0 1 1 0 1	23
14	0 0 0 0 1 1 1 0	25
15	0 0 0 0 1 1 1 1	26
16	0 0 0 1 0 0 0 0	31

Cost measure: bit flip

amortized cost:

set 1: 2

set 0: 0