1 算法的分析基础

1.1 算法的分析概念

问题的提出: 我们希望构造一个只与算法本身的特性有关的量，而与软硬件环境、计算机性能、实现的语言等无关的理想的情况. 其做法是: 将一些基本操作，如运算、赋值、比较等，令其时间代价均为 1。

但是, 算法会跟随输入规模的不同发生不同.

例 1.1 如下算法的运行速度和 n 有关:

输入: 数组 $A\,[a_1, a_2, \ldots, a_n]$

输出: 升序数组 $A'\,[a'_1, a'_2, \ldots, a'_n]$, 满足 $a'_1 \le a'_2 \le \ldots \le a'_n$

for $j \leftarrow 2$ to n do

$key \leftarrow A[j]$

$i \leftarrow j - 1$

while $i > 0$ and $A[i] >$ key do

$A[i+1] \leftarrow A[i]$

$i \leftarrow i - 1$

end

$A[i+1] \leftarrow$ key

end

我们的抽象可以认为算法运行时间仅仅依赖于问题输入规模 n，表示为 $T(n)$. 并且可以简化地用代码执行次数来估计算法运行时间.

但是不同的算法结果在表示上过于繁琐, 甚至难以表示. 不便于进行统一化的分析，所以我们对其进行简化.

- 忽略低阶项。因为当 n 足够大时，低阶项对算法时间的影响可忽略
- 忽略高阶项的常数系数。因为在考虑大规模输入下的计算效率时，相对于增长率而言，系数是次要的。
- 只剩最高阶项, 称作渐进记号.

渐进记号 渐进记号表示一类函数, 而非单独的一个函数. 其中, 满足如下的定义:

定义 1.1 对于给定的函数 $g(n)$，渐进上界 $O(g(n))$ 表示如下函数的集合:

$$O(g(n)) = \{T(n) : \exists c, n_0 > 0, \text{使得} \forall n \geq n_0, 0 \leq T(n) \leq cg(n)\}$$

有时候为了方便起见, 常用等于记号代替属于记号.

含义:

- 如果算法用 n 值不变的同一类数据在某台机器上运行时，所用的时间总是小于 $|g(n)|$ 的一个常数倍。所以 $g(n)$ 是计算时间 $T(n)$ 的一个上界函数。

例 1.2

$$
\begin{aligned}
&\cos(n) = O(1) \\
&\frac{n^2}{2} - 12n = O\left(n^2\right) \\
&\log_7^n = \log_2^n / \log_2^7 = O\left(\log_2^n\right) = O(\log n) \\
&\sum_{i=1}^{n} \frac{1}{i} \text{ (假设} n \text{ 是2 的整数幂)} \\
=&\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \ldots + \frac{1}{n-1} + \frac{1}{n} \\
<&\frac{1}{1} + \frac{1}{2} + \frac{1}{2} : 1\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} : \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \ldots + \frac{1}{n/2} + \frac{1}{n} \\
=&1 + 2 \cdot \frac{1}{2} + 4 \cdot \frac{1}{4} + 8 \cdot \frac{1}{8} + \ldots + \frac{n}{2} \cdot \frac{1}{n/2} + \frac{1}{n} \\
=&\log n + 1/n = O(\log n)
\end{aligned}
$$

定义 1.2 对于给定的函数 $g(n)$，渐进下界 $\Omega(g(n))$ 表示如下函数的集合：

$$\Omega(g(n)) = \{T(n) : \exists c, n_0 > 0, \text{使得} \forall n \geq n_0, 0 \leq cg(n) \leq T(n)\}$$

含义:

- 如果算法用 n 值不变的同一类数据在某台机器上运行时，所用的时间总是不小于 $|g(n)|$ 的一个常数倍。所以 $g(n)$ 是计算时间 $T(n)$ 的一个下界函数.

例 1.3

$$n^3 - 2n = \Omega\left(n^3\right)$$

$$n^2 + 12n = \Omega\left(n^2\right)$$

$$\sum_{i=1}^{n} \frac{1}{i} \text{ (假设} n \text{ 是2 的整数幂)}$$

$$= \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \frac{1}{6} + \frac{1}{7} + \frac{1}{8} + \frac{1}{9} + \frac{1}{10} + \ldots 1 + \frac{1}{n-1} + \frac{1}{n}$$

$$< \frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} + .1 + \frac{1}{n} + \frac{1}{n}$$

$$= 1 + \frac{1}{2} + 2 \cdot \frac{1}{4} + 4 \cdot \frac{1}{8} + \ldots + \frac{n}{2} \cdot \frac{1}{n}$$

$$= 1 + \frac{1}{2} \log n = \Omega(\log n)$$

定义 1.3 对于给定的函数 $g(n)$，渐进紧确界 $\Theta(g(n))$ 表示如下函数的集合:

$$\Theta(g(n)) = \{T(n) : \exists c_1, c_2, n_0 > 0, \text{使得} \forall n \geq n_0, c_1 g(n) \leq T(n) \leq c_2 g(n)\}$$

算法按运行时间分类

- 多项式时间算法:

$$O(1)<O(\log n)<O(n)<O(n\log n)<O\left(n^{2}\right)<O\left(n^{3}\right)$$

- 指数时间算法：计算时间用指数函数限界的算法.

$$O\left(2^{n}\right)<O(n!)<O\left(n^{n}\right)$$

问题 1.1 为什么可以抛弃多项式的剩余项来计算时间复杂度? 即证明: 若 $A(n)=a_{m}n^{m}+\ldots+a_{1}n+a_{0}$ 是一个 m 次多项式，则有 $A(n)=O\left(n^{m}\right)$

证明. 当 $n\geq 1$ 时, 有

$$\begin{aligned}
&|A(n)|\leq\left|a_{m}\right|n^{m}+\ldots+\left|a_{1}\right|n+\left|a_{0}\right|\\
&\leq\left(\left|a_{m}\right|+\left|a_{m-1}\right|/n+\ldots+\left|a_{0}\right|/n^{m}\right)n^{m}\\
&\leq\left(\left|a_{m}\right|+\left|a_{m-1}\right|+\ldots+\left|a_{0}\right|\right)n^{m}\\
&\text{令}c=\left|a_{m}\right|+\left|a_{m-1}\right|+\ldots+\left|a_{0}\right|
\end{aligned}$$

则, 定理得证。

1.2 递归算法的求解

递归的问题一般的模式 将规模为 n 的问题划分为 a 个子问题，每个子问题的规模为 n/b。划分原问题与合并答案的代价由函数 f(n) 来描述。也就是 $T(n)=aT(n/b)+f(n)$.

方法 1. 逐次展开 如式子 $T(n) = 3T(n/4) + O(n^2)$ 所示, 有如图1所示的内容. 希望把所有的树求和, 就有类似于等比数列的求和公式.

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + L + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + O\left(n^{\log_4^3}\right) \\
&= \frac{1-(3/16)^{\log_4 n}}{1-3/16}cn^2 + O\left(n^{\log_4^3}\right) = O\left(n^2\right) + O\left(n^{\log_4^3}\right) = O\left(n^2\right)
\end{aligned}
$$

主定理 我们给出一种方法, 使得可以求解 $T(n) = aT(n/b) + O(n^k)$ 的递归式. 我们同样按照方法 1 的描述展开, 得到如图1 的形式.

那么, 总共的时间代价为:

$$
\begin{aligned}
T(n) = cn^k &+ \left(\frac{a}{b^k}\right) \cdot cn^k + \left(\frac{a}{b^k}\right)^2 \cdot cn^k \\
&+ \cdots \left(\frac{a}{b^k}\right)^{\log_b n} \cdot cn^k \\
&= cn^k \frac{1-\left(\frac{a}{b^k}\right)^{\log_b^n}}{1-\left(\frac{a}{b^k}\right)}
\end{aligned}
$$

- 本质上是等比数列求和公式
- 看公比 q
  - $q < 1$, 由第一项决定: $T(n) > a_1(1-q) = O\left(a_1\right) = O\left(n^k\right)$
  - $q > 1$, 由后面的决定: $T(n) = O\left(\left(\frac{a}{b^k}\right)^{\log_b n} \cdot n^k\right) = O\left(n^{\log_b a}\right)$

- 等比系数等于 1，则每一项数一直这么大，则最终结果主要有第一项乘以项数: $T(n) = O\left(n^k \log_b n\right) = O\left(n^k \log n\right)$.

于是我们得到了主定理:

定理 1.1 (主定理) 形如 $T(n) = aT(n/b) + f(n), (a > 0, b > 0)$ 的递归调用最后有如下的总执行次数:

$$T(n) = \frac{1 - \left(\frac{a}{b^k}\right)^{\log_b^n}}{1 - \left(\frac{a}{b^k}\right)} \Theta\left(n^k\right)$$

更简洁地写作渐进的表达式, 有:

$$T(n) = \begin{cases} \Theta\left(n^k\right) & \text{if } a/b^k < 1 \\ \Theta\left(n^k \log n\right) & \text{if } a/b^k = 1 \\ \Theta\left(n^{\log_b^a}\right) & \text{if } a/b^k > 1 \end{cases}$$

例 1.4 求递归式 $T(n) = 3T(n/4) + O(n^2)$ 的复杂度: 注意到 $a = 3, b = 4, k = 2$, 则 $a/b^k = 3/16 < 1$, 则复杂度由第一项决定, 则 $T(n) = O(n^2)$

求递归式 $T(n) = 9T(n/3) + O(n)$ 的复杂度: $a = 9, b = 3, k = 1, a/b^k = 9/3 = 3 > 1$, 比例因子大于 1, 则复杂度由最后一项决定。因此 $T(n) = O\left(n^{\log_b a}\right) = O\left(n^{\log_3 9}\right) = O\left(n^2\right)$.

求递归式 $T(n) = T(2n/3) + O(1)$ 的复杂度: $a = 1, b = 3/2, k = 0$ 则 $a/b^k = 1$, 比例因子等于 1, 则复杂度由第一项乘以序列项数决定。因此, $T(n) = O(\log n)$

如果我们发现 $f(n)$ 不是多项式函数, 但是大体上的问题也是一样的. 所以我们可以仿照主定理推理出主定理的扩展形式:

定理 1.2 (主定理的扩展形式) 对于递推式 $T(n) = aT(n/b) + f(n)$, 假设 $af\left(\frac{n}{b}\right) = cf(n) + c_1$, 其中参数 a, b 与函数 f 已知, 那么有:

- 如果 $c < 1$, 则 $T(n) = \Theta(f(n))$
- 如果 c > 1, 则 $T(n) = \Theta\left(n^{\log_b a}\right)$
- 如果 c $= 1$, 则 $T(n) = \Theta(f(n) \log n)$

例 1.5 求 $T(n) = 3T(n/4) + n \log n$ 的复杂度: 注意到 $a = 3, \mathrm{b} = 4, \mathrm{f}(n) = n \log n$, 并且 $\mathrm{a} \cdot \mathrm{f}(n/\mathrm{b}) = 3 \cdot (n/4) \log(n/4) = 3/4 \cdot n \log(n)$-const. 则 $c < 1$,$T(n) = O(f(n)) = O(n \log n)$

2 排序算法

2.1 快速排序

- 基本思想: 分治法.
- 过程描述
  - 确定元素: 首先随机选取一个数 x;
  - 调整数组: 把小于 x 的放在这个位置的左边, 大于 x 的放在这个位置的右侧.
  - 递归执行: 一直这样递归下去.

问题: 如何解决 (2) 操作?

- 开两个额外的数组, $a[], b[]$, 如果当前的数小于等于 x, 就插入 a, 否则, 插入 b, 最后把 $a[], b[]$ 放入原先的数. – 简单但是不够优美.
- 用两个指针 i,j 分别向中间走, 如果 $i < x$, 向后移动, 直到 $i > x$, i 停下, j 也一样. 直到 i,j 都错位了, 现在把两个指针的内容交换一下就行了. 继续往中间走就行了.
- 正确性证明: 在任何时候, i 左边所有的数 $\leq x$(算法的语句保证), j 右边的数也是大于 x 的. 于是就可以分成相应的区间.

方便运行的代码:

```
void qst(int a[], int l, int r){
    if(l>=r) return;
    //(1)
    int x = a[l]; //Alterate r, (l+r)/2
    int i = l-1, j=r+1; //Move first, then compare and swap
    while(i<j){
        do i++; while(a[i]<x);
        do j--; while(a[j]>x);
        if(i<j) swap(a[i],a[j]);
    }
    qst(a, l, j); // If x:=r, then this line should be qst(a,l,i-1)
    qst(a,j+1,r); // and this line should be qst(a,i,r)
}
```

2.2 归并排序

- 基本思想: 分治.
- 过程描述:
  - 确定数组中心为分界点 $(l+r)/2$.
  - 递归排序左边和右边;
  - 把左边和右边两个有序的数组合二为一.
- 如何合二为一?
  - 假设有 a, b, res 数组, 开两个指针 i, j 指向 a, b 的头. 比较 $a[i]b[j]$ 中较小的那一个, 放到答案 res 数组中, 较小的那一个指针向后移动一位. (相同的时候需要把第一个移动到答案中, 这样做是稳定的 (Def. 相同值的会在相同的前后的顺序即可.))
  - 正确性: 因为取的是两个数组中的较小值, 每次总是能够按照从小到大的选取.

Code:

```
// We need a tmp array to store relations...
void mgsort(int a[], int tmp[], int l, int r){
    if(l>=r) return ;
    //(1)
    int mid = (l+r)>>1; // Bracket missing is also okay.
    mgsort(a, tmp, l, mid);
    mgsort(a, tmp, mid+1, r);
    int k = 0, i = l, j = mid+1;
```

```
    // merge
    while(i<=mid && j<=r){
        if(a[i]<=a[j]){
            tmp[k++] = a[i++];
        }else{
            tmp[k++] = a[j++];
        }
    }
    // Is there anything missing in the loop?
    while(i<=mid) tmp[k++] = a[i++];
    while(j<=r) tmp[k++] = a[j++];
    for(i=l,j=0; i<=r; i++, j++) a[i] = tmp[j];
}
```

2.3 基于选择的排序的时间分析

以比较为基础检索的时间下界.

问题 2.1 对于一个长度为 n 的有序数组 A, 要检索某个元素 x 是否在 A 中出现. 假设我们只允许进行元素之间的比较, 而不允许使用其他手段, 那需要比较次数的时间下限是多少？

- 以比较为基础的检索算法, 在执行过程中, 都可以用一个二元比较树 (见图3) 来描述.
- 每个内节点表示一个元素比较, 因此比较树中一定有 n 个内节点, 走到叶子节点, 表示检索完成.
- 任何一种以比较为基础的算法, 在最坏情况下的计算时间都不低于 $O(\log n)$. 因此, 不可能存在最

坏情况比二分检索数量级还低的算法.

- 二分检索是解决检索问题最坏情况下的最优算法.

以比较为基础分类的时间下界.

问题 2.2 我们能够说出对于一个排序问题, 其最坏的情况下, 最好的算法复杂度是什么吗?

- n 个元素, 在没有进行比较时, 所有可能的结果有 $n!$ 种.
- 排序问题 = 所有结果中, 否定错误结果, 找到唯一的正确结果
- 一次比较, 两个元素顺序被确定, 剩余的可能性为 $n!/2$ 种
- 进行 m 次比较后, 剩余的可能性为 $n!/2^m$ 种
- 由于估计 $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$
- 最坏情况复杂度是 $O(n\log(n))$

说明算法的筛选过程, 可以使用比较树 (图4) 的情况:

- 假设 n 个关键字 $A(1), A(2), \cdots, A(n)$ 互异.
- 任意两个关键字的比较必导致 $A(i) < A(j)$ 或 $A(i) > A(j)$ 的结果.
- 过程:
  - 若 $A(i) < A(j)$, 进入下一级的左分支
  - 若 $A(i) > A(j)$, 进入下一级的右分支
- 在叶子节点终止.

对于此的意义:

- 路径: 唯一的分类排序序列
- 路径长度: 该序列代表的分类表所需要的比较次数
- 最坏情况下界: 该算法对应的比较树的最小高度.

附: C++ 中的排序

C 中的排序 排序在 C 语言中就是一项常用的操作. C 给我们提供了 `qsort` 函数来进行排序. C 是高级的汇编语言, 自然要实现对每一个字节的精确控制. 这在标准库中也是体现的淋漓尽致的. 我们可以写出如下的代码:

```
int compare(const void *a, const void *b) {
    int num1 = *((int*)a);
    int num2 = *((int*)b);

    // sort by increasing order
    if (num1 < num2) return -1;
    if (num1 > num2) return 1;
    return 0;
}

int main() {
    int arr[] = {5, 2, 9, 1, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    // 调用qsort函数进行排序，传入数组、元素数量、每个元素的大小和比较函数
    qsort(arr, n, sizeof(int), compare);

    // 打印排序后的结果
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

对于这里的返回值的意思:

- 如果第一个元素小于第二个元素，compare 函数应该返回一个负整数（通常是负 1）。
- 如果第一个元素大于第二个元素，compare 函数应该返回一个正整数（通常是正 1）。
- 如果两个元素相等，compare 函数应该返回 0。

这样会很不方便. 于是, 在面向对象等一系列技术的加持下, C++ 中的标准库排序函数就好很多.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    // 创建一个整数向量
    std::vector<int> numbers = {5, 2, 9, 1, 5, 6};
```

```
    // 使用std::sort进行升序排序
    std::sort(numbers.begin(), numbers.end());

    // 打印排序后的结果
    for (int num : numbers) {
        std::cout << num << " ";
    }

    return 0;
}
```

对于 sort 第三个参数 – 比较参数

- 接受两个参数，然后根据自定义规则返回一个布尔值
- 第一个参数应该在第二个参数之前 (保持原位) = true
- 第一个参数应该在第二个参数之前 (更换位置) = false

附: 各个排序的稳定性

排序方法	时间复杂度 (平均)	时间复杂度 (最坏)	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O\left(n^2\right)$	$O\left(n^2\right)$	$O(n)$	$O(1)$	稳定
希尔排序	$O\left(n^{1.3}\right)$	$O\left(n^2\right)$	$O(n)$	$O(1)$	不稳定
选择排序	$O\left(n^2\right)$	$O\left(n^2\right)$	$O\left(n^2\right)$	$O(1)$	不稳定
堆排序	$O\left(n\log_2 n\right)$	$O\left(n\log_2 n\right)$	$O\left(n\log_2 n\right)$	$O(1)$	不稳定
冒泡排序	$O\left(n^2\right)$	$O\left(n^2\right)$	$O(n)$	$O(1)$	稳定
快速排序	$O\left(n\log_2 n\right)$	$O\left(n^2\right)$	$O\left(n\log_2 n\right)$	$O\left(n\log_2 n\right)$	不稳定
归并排序	$O\left(n\log_2 n\right)$	$O\left(n\log_2 n\right)$	$O\left(n\log_2 n\right)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O\left(n^2\right)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O\left(n^*k\right)$	$O\left(n^*k\right)$	$O\left(n^*k\right)$	$O(n+k)$	稳定

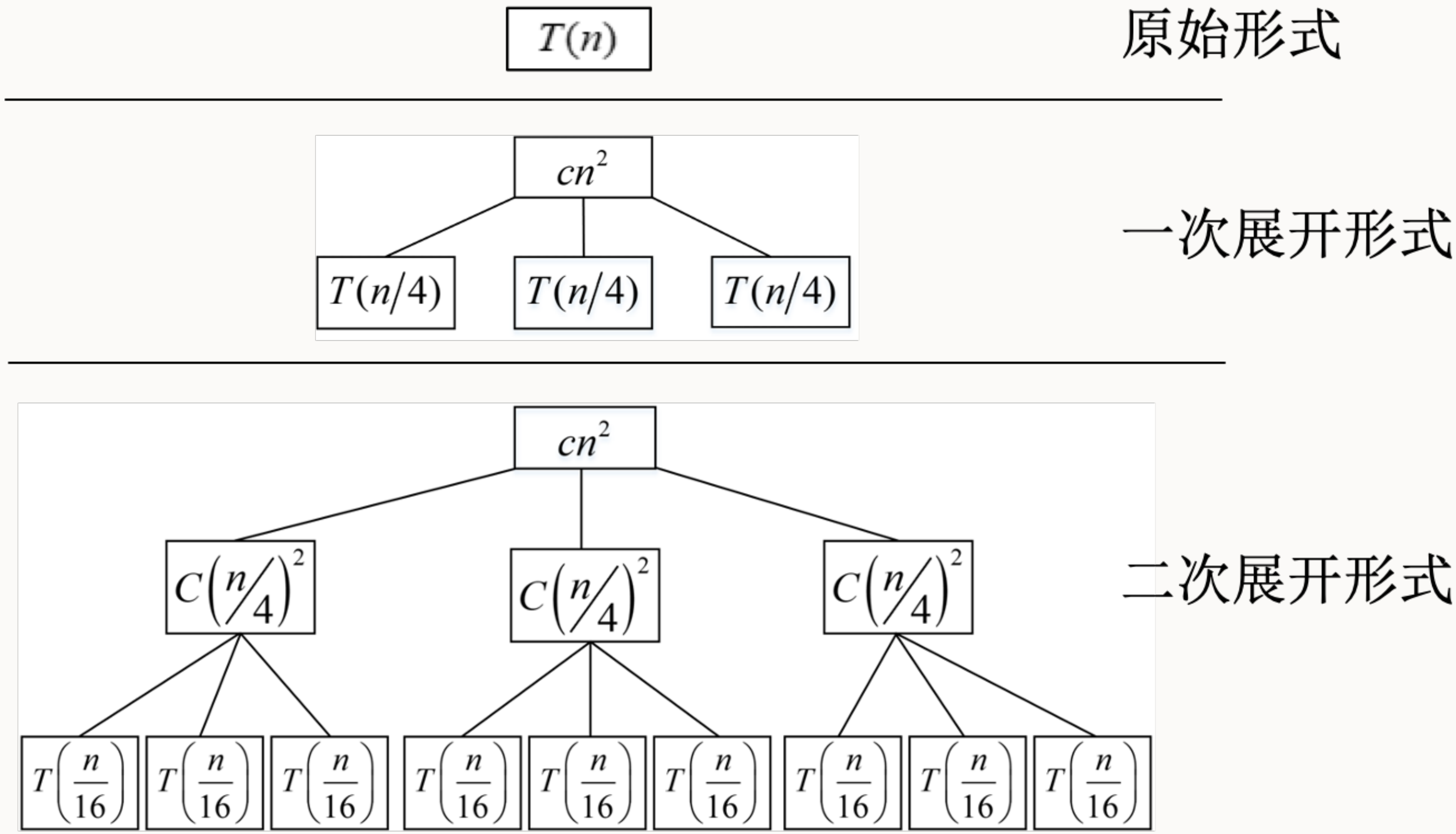
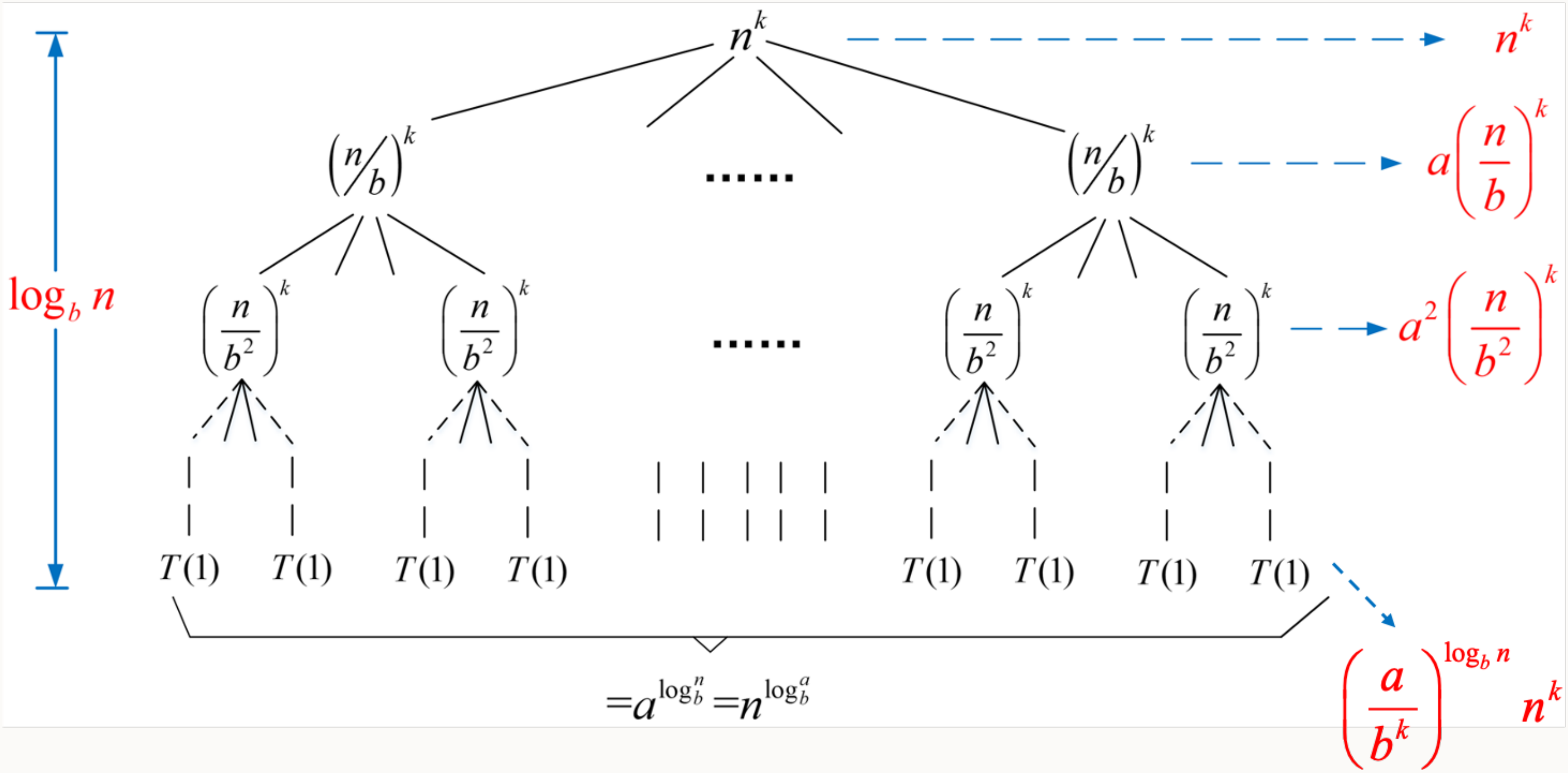

T(n)
原始形式
cn²
T(n/4)
T(n/4)
T(n/4)
一次展开形式
cn²
C(n/4)²
C(n/4)²
C(n/4)²
二次展开形式
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)
T(n/16)

图 1: 循环展开的递归树



n^k
$\left(\frac{n}{b}\right)^k$
$\left(\frac{n}{b}\right)^k$
......
$\left(\frac{n}{b^2}\right)^k$
$\left(\frac{n}{b^2}\right)^k$
......
$\left(\frac{n}{b^2}\right)^k$
$\left(\frac{n}{b^2}\right)^k$
$T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$ $T(1)$
$=a^{\log_b^n}=n^{\log_b^a}$
$\log_b n$
n^k
$a\left(\frac{n}{b}\right)^k$
$a^2\left(\frac{n}{b^2}\right)^k$
$\left(\frac{a}{b^k}\right)^{\log_b n} n^k$

图 2: 主定理的树

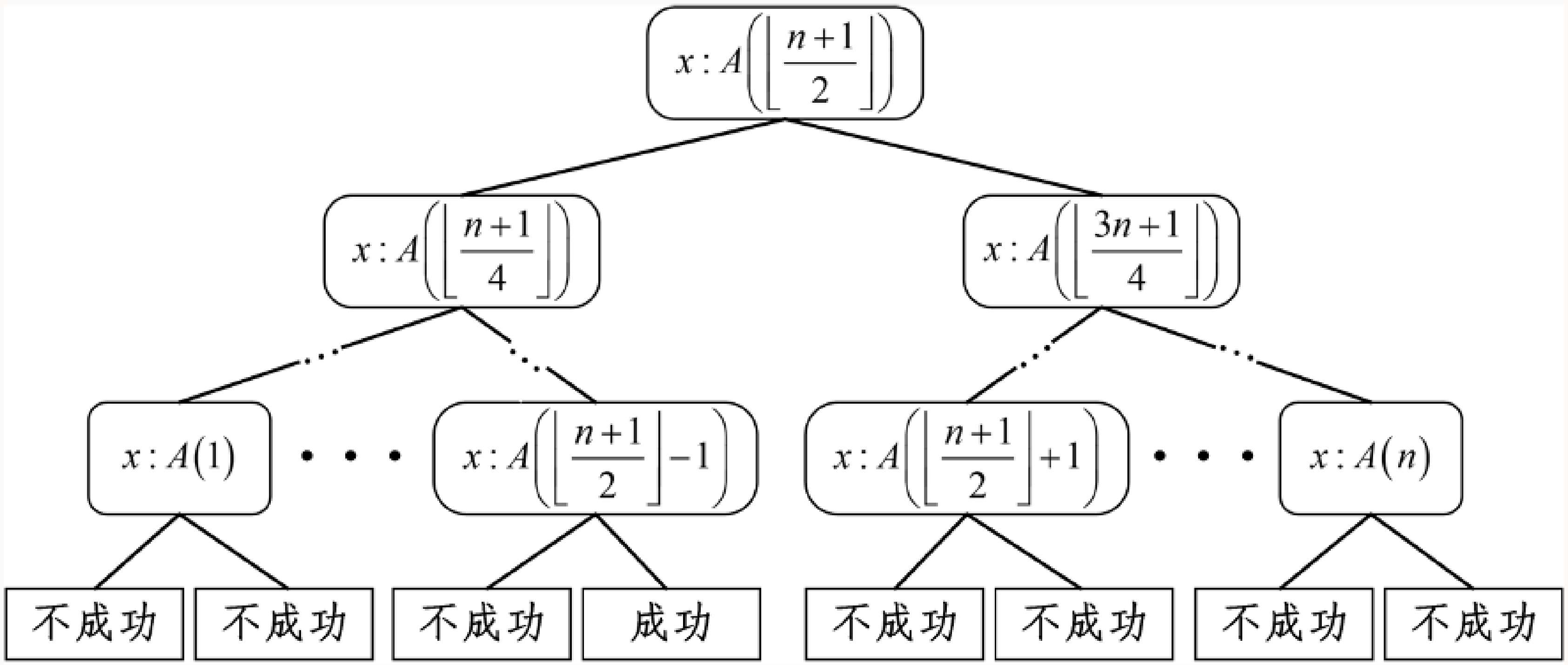

$x:A\left(\left\lfloor\frac{n+1}{2}\right\rfloor\right)$
$x:A\left(\left\lfloor\frac{n+1}{4}\right\rfloor\right)$
$x:A\left(\left\lfloor\frac{3n+1}{4}\right\rfloor\right)$
$x:A(1)$
$x:A\left(\left\lfloor\frac{n+1}{2}\right\rfloor-1\right)$
$x:A\left(\left\lfloor\frac{n+1}{2}\right\rfloor+1\right)$
$x:A(n)$
不成功
不成功
不成功
成功
不成功
不成功
不成功
不成功

图 3: 二元比较树例子

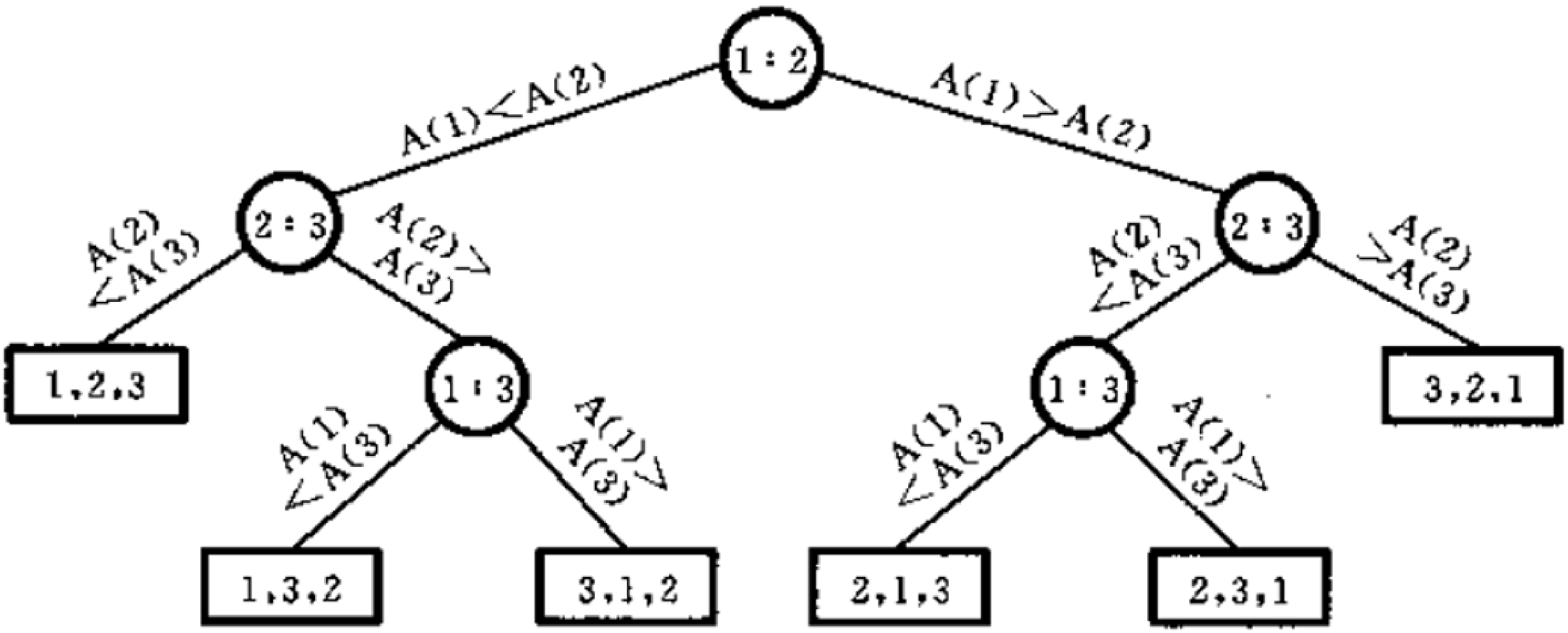


图 4.6 对 3 个关键字分类的比较树

图 4: 比较的平行时空