玩儿转算法面试

liuyubobobo

面试中的时间复杂度分析

时间复杂度

到底什么是大O

n表示数据规模

O(f(n)) 表示运行算法所需要执行的指令数，和f(n)成正比。

到底什么是Big O

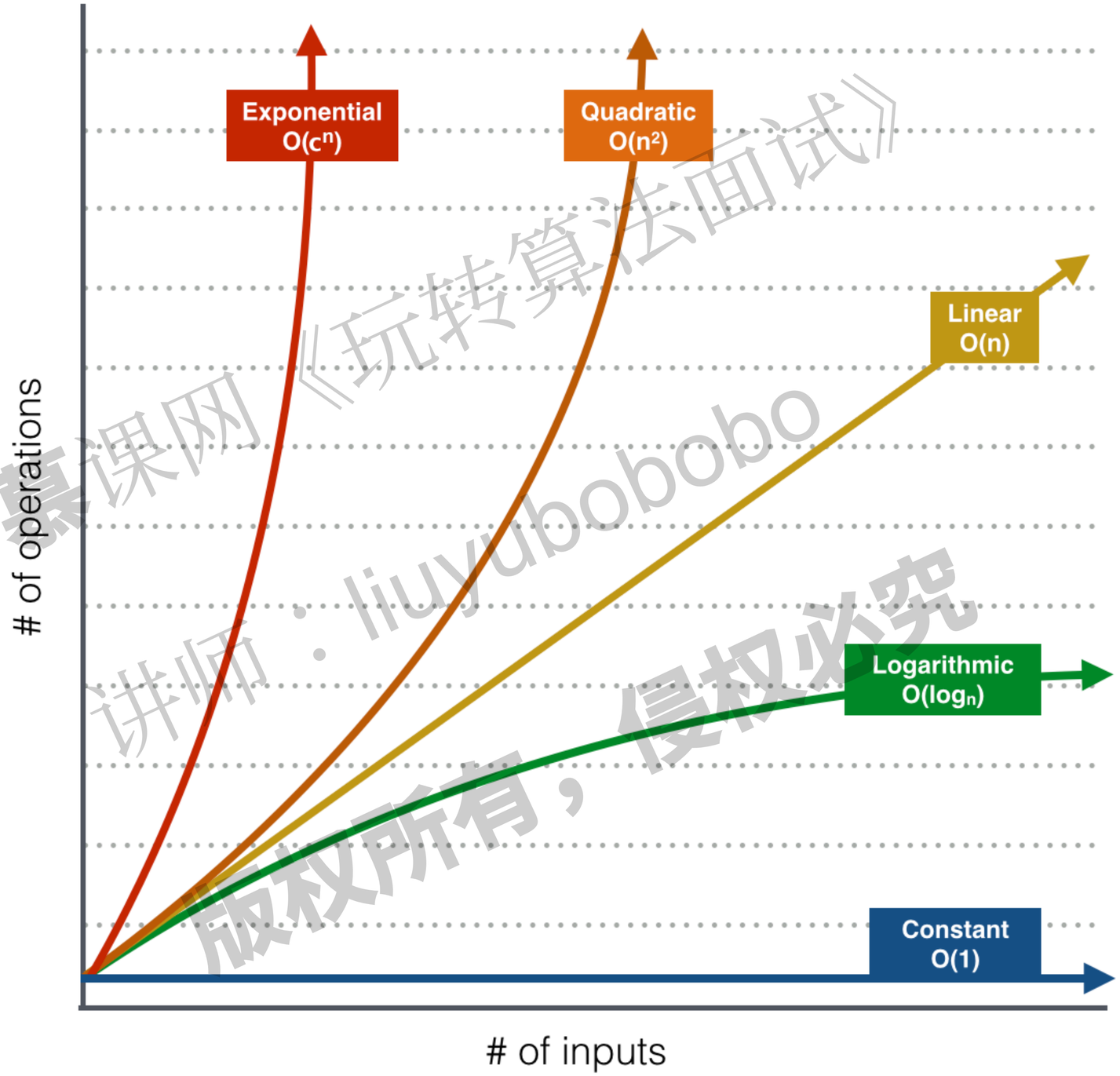
二分查找法 O(logn)	所需执行指令数：a*logn
寻找数组中的最大/最小值 O(n)	所需执行指令数：b*n
归并排序算法 O(nlogn)	所需指令数：c*nlogn
选择排序法 O(n^2)	所需指令数：d*n^2

到底什么是Big O

算法A：O(n)　　所需执行指令数：10000*n

算法B：O(n^2)　　所需执行指令数：10*n^2

n	A指令数 10000n	B指令数 10n^2	倍数
10	10^5	10^3	100
100	10^6	10^5	10
1000	10^7	10^7	1
10000	10^8	10^9	0.1
10^5	10^9	10^11	0.01
10^6	10^10	10^13	0.001



到底什么是大O

在学术界，严格地讲，O(f(n))表示算法执行的上界

归并排序算法的时间复杂度是O(nlogn)的，同时也是O(n^2)

在业界，我们就使用O来表示算法执行的最低上界

我们一般不会说归并排序是O(n^2)的

到底什么是大O

$$O(nlogn + n) = O(nlogn)$$

$$O(nlogn + n^2) = O(n^2)$$

到底什么是大O

O(AlogA + B)

O(AlogA + B^2)

对邻接表实现的图进行遍历：

时间复杂度：O(V + E)

一个时间复杂度的问题

有一个字符串数组，将数组中的每一个字符串按照字母序排序；之后再将整个字符串数组按照字典序排序。整个操作的时间复杂度?

O(n*nlogn + nlogn) = O(n^2logn)

一个时间复杂度的问题

有一个字符串数组，将数组中的每一个字符串按照字母序排序；之后再将整个字符串数组按照字典序排序。整个操作的时间复杂度?

假设最长的字符串长度为s；数组中有n个字符串

对每个字符串排序：O(slogs)

将数组中的每一个字符串按照字母序排序：O(n*slog(s))

将整个字符串数组按照字典序排序：O(s*nlog(n))

一个时间复杂度的问题

有一个字符串数组，将数组中的每一个字符串按照字母序排序；之后再将整个字符串数组按照字典序排序。整个操作的时间复杂度?

$$O(n*slog(s)) + O(s*nlog(n)) = O(n*s*logs + s*n*logn)$$
$$= O(n*s*(logs+logn))$$

算法复杂度在有些情况是用例相关的

插入排序算法 O(n^2)

最差情况：O(n^2)

最好情况：O(n)

平均情况：O(n^2)

快速排序算法 O(nlogn)

最差情况：O(n^2)

最好情况：O(nlogn)

平均情况：O(nlogn)

对数据规模有一个概念

数据规模的概念

对 10^5 的数据进行选择排序，结果计算机假死?

数据规模的概念

如果要想在1s之内解决问题：

O(n^2)的算法可以处理大约10^4级别的数据；

O(n)的算法可以处理大约10^8级别的数据；

O(nlogn)的算法可以处理大约10^7级别的数据

空间复杂度

多开一个辅助的数组：O(n)

多开一个辅助的二维数组：O(n^2)

多开常数空间：O(1)

空间复杂度

递归调用是有空间代价的

空间复杂度O(1)

```
int sum1( int n ){

    assert( n >= 0 );
    int ret = 0;
    for( int i = 0 ; i <= n ; i ++ )
        ret += i;
    return ret;
}
```

空间复杂度O(n)

```
int sum2( int n ){

    assert( n >= 0 );
    if( n == 0 )
        return 0;

    return n + sum2(n-1);
}
```

常见的复杂度分析

O(1)

```
void swapTwoInts( int &a , int &b ){

    int temp = a;
    a = b;
    b = temp;

}
```

O(n)

```
int sum( int n ){

    int ret = 0;
    for( int i = 0 ; i <= n ; i ++ )
        ret += i;
    return ret;
}
```

O(n)

```
void reverse( string &s ){

    int n = s.size();
    for( int i = 0 ; i < n/2 ; i ++ )
        swap( s[i] , s[n-1-i] );
}
```

1/2*n次swap操作：O(n)

O(n^2)

```
void selectionSort(int arr[], int n){

    for(int i = 0 ; i < n ; i ++){
        int minIndex = i;
        for( int j = i + 1 ; j < n ; j ++ )
            if( arr[j] < arr[minIndex] )
                minIndex = j;

        swap( arr[i] , arr[minIndex] );
    }
}
```

(n-1) + (n-2) + (n-3) + ... + 0

= (0+n-1)*n/2

= (1/2)n*(n-1)

= 1/2*n^2 - 1/2*n

= O(n^2)

O(n^2)?

```
void printInformation(int n){

    for( int i = 1 ; i <= n ; i ++ )
        for( int j = 1 ; j <= 30 ; j ++ )
            cout<<"Class "<<i<<" - "<<"No. "<<j<<endl;
    return;
}
```

30n次基本操作：O(n)

O(logn)

```
int binarySearch(int arr[], int n, int target){

    int l = 0, r = n-1;
    while( l <= r ){
        int mid = l + (r-l)/2;
        if( arr[mid] == target ) return mid;
        if( arr[mid] > target ) r = mid - 1;
        else l = mid + 1;
    }
    return -1;
}
```

O(logn)

二分查找法的时间复杂度是O(logn)的

在 n 个元素中寻找

在 n/2 个元素中寻找

在 n/4 个元素中寻找

……

在 1 个元素中寻找

n经过几次“除以2”操作后，等于1?

$\log_2 n = O(\log n)$

O(logn)

```
string intToString( int num ){

    string s = "";

    while( num ){
        s += '0' + num%10;
        num /= 10;
    }

    reverse(s);
    return s;
}
```

n经过几次“除以10”操作后，等于0?

$\log_{10}n = O(\log n)$

O(logn)

$\log_a N$ $\log_b N$

$\log_a N = \log_a b * \log_b N$

（常数）

O(logn)

整形转成字符串

```
string intToString( int num ){

    string s = "";

    while( num ){
        s += '0' + num%10;
        num /= 10;
    }

    reverse(s);
    return s;
}
```

O(n^2) ?

```
void hello(int n){

    for( int sz = 1 ; sz < n ; sz += sz )
        for( int i = 1 ; i < n ; i ++ )
            cout<<"Hello, Algorithm!"<<endl;
}
```

O(sqrt(n))

```
bool isPrime( int n ){

    for( int x = 2 ; x*x <= n ; x ++ )
        if( n%x == 0 )
            return false;
    return true;
}
```

复杂度实验

复杂度试验

我们自以为写出了一个O(nlogn)的算法，但实际是O(n^2)的算法?

数据规模的概念

如果要想在1s之内解决问题：

O(n^2)的算法可以处理大约10^4级别的数据；

O(n)的算法可以处理大约10^8级别的数据；

O(nlogn)的算法可以处理大约10^7级别的数据

复杂度试验

实验，观察趋势

每次将数据规模提高两倍，看时间的变化

实践：复杂度实验 O(n)，O(n^2)

复杂度试验

$$\log_2 N / \log N$$

$$= (\log 2 + \log N)/\log N$$

$$= 1 + \log 2/\log N$$

实践：复杂度实验 O(logn), O(nlogn)

递归算法的复杂度分析

递归算法的复杂度分析

不是有递归的函数就一定是O(nlogn)!

递归中进行一次递归调用的复杂度分析

```
int binarySearch(int arr[], int l, int r, int target){

    if( l > r )
        return -1;

    int mid = l + (r-l)/2;
    if( arr[mid] == target )
        return mid;
    else if( arr[mid] > target )
        return binarySearch(arr, l, mid-1, target);
    else
        return binarySearch(arr, mid+1, r, target);
}
```

时间复杂度：O(logn)

递归中进行一次递归调用的复杂度分析

如果递归函数中，只进行一次递归调用，

递归深度为depth;

在每个递归函数中，时间复杂度为T;

则总体的时间复杂度为O(T * depth)

递归中进行一次递归调用的复杂度分析

```
int sum( int n ){

    assert( n >= 0 );

    if( n == 0 )
        return 0;
    return n + sum(n-1);
}
```

递归深度：n

时间复杂度：O(n)

递归中进行一次递归调用的复杂度分析

```
double pow( double x, int n ){

    assert( n >= 0 );

    if( n == 0 )
        return 1.0;

    double t = pow(x, n/2);
    if( n%2 )
        return x*t*t;

    return t*t;
}
```

递归深度：logn

时间复杂度：O(logn)

递归中进行多次递归调用

```
int f(int n){

    assert( n >= 0 );

    if( n == 0 )
        return 1;

    return f(n-1) + f(n-1);
}
```

计算调用的次数

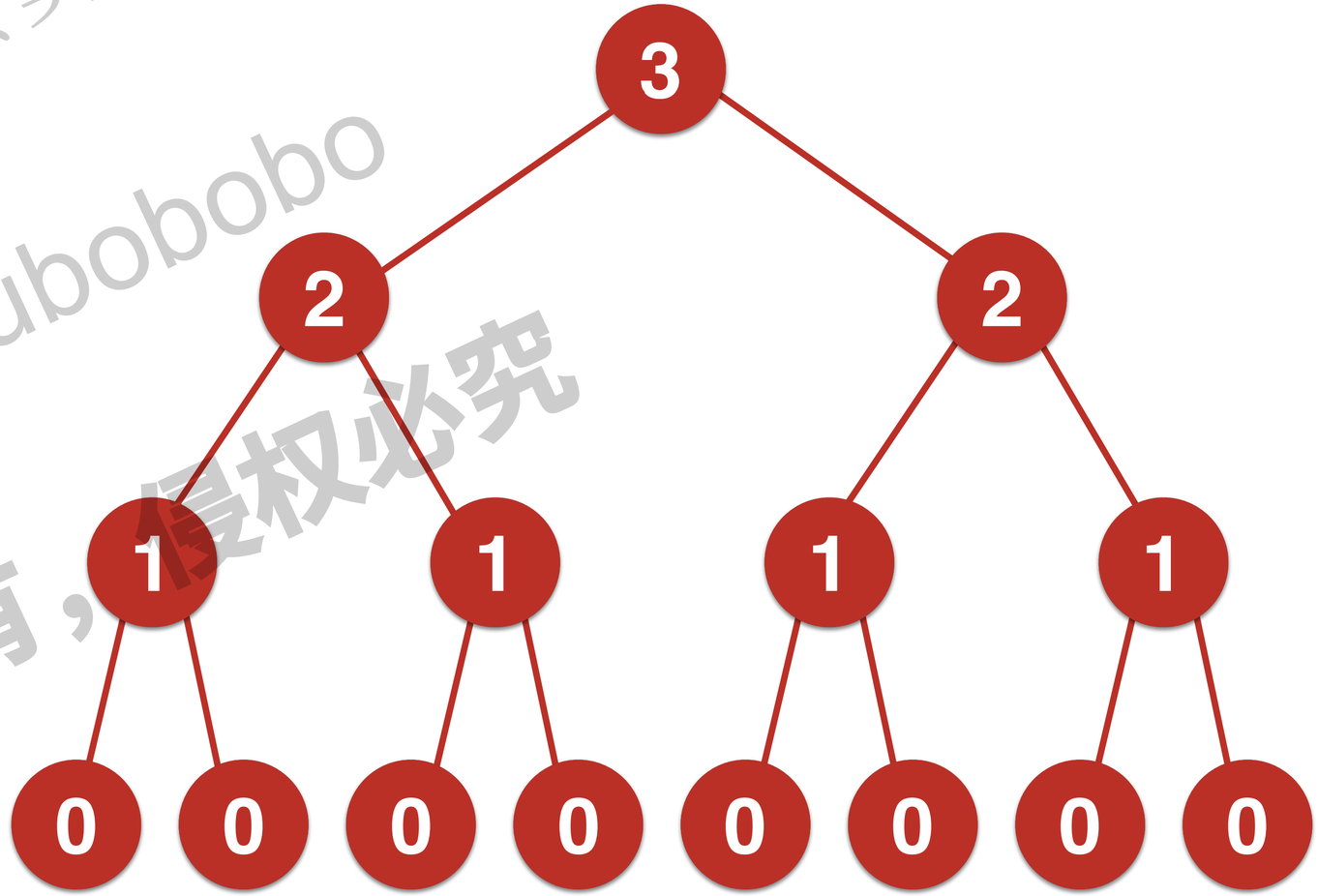
递归中进行多次递归调用

```
int f(int n){

    assert( n >= 0 );

    if( n == 0 )
        return 1;

    return f(n-1) + f(n-1);
}
```



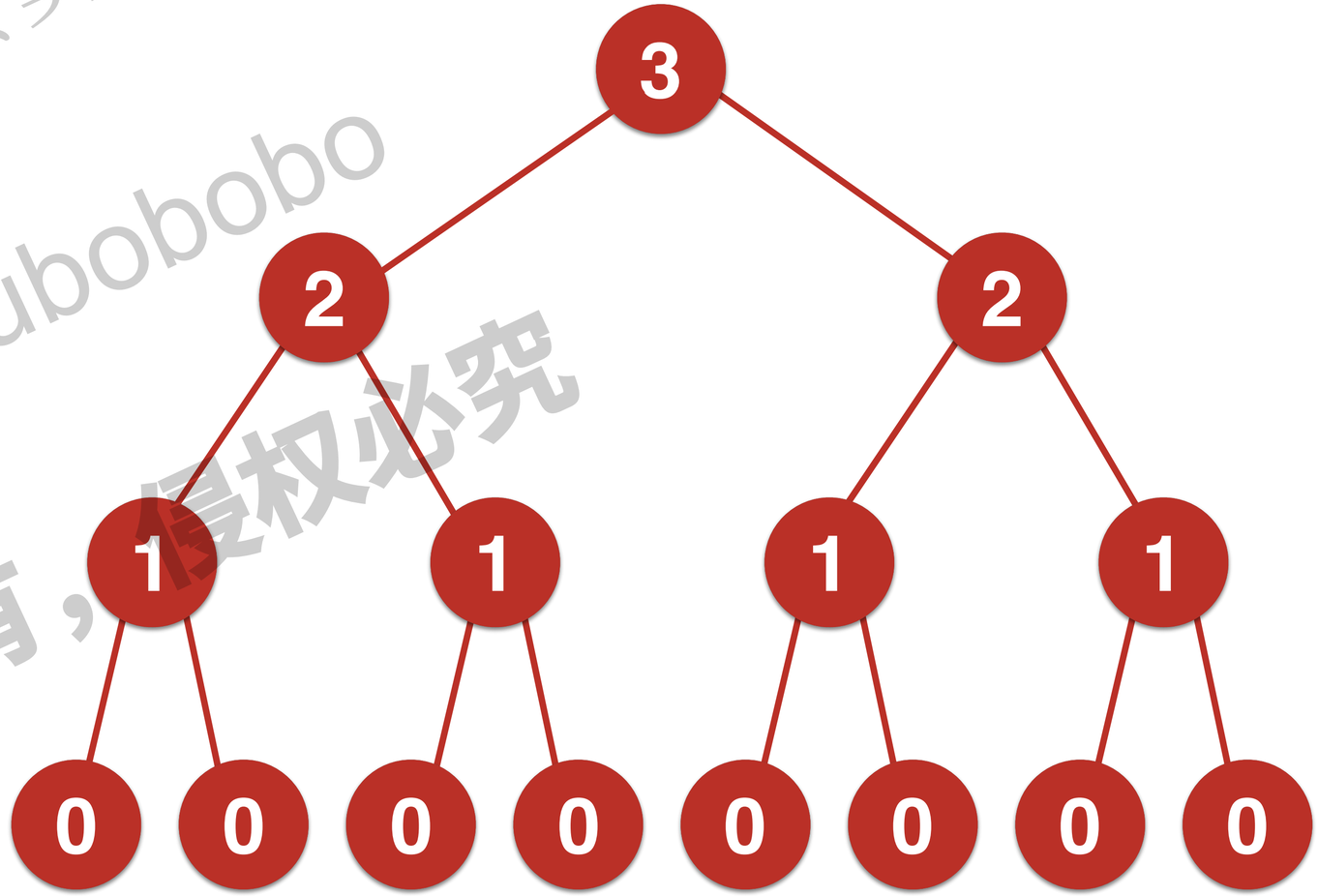
递归中进行多次递归调用

$1 + 2 + 4 + 8 = 15$

$2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^n$

$= 2^{n+1} - 1$

$= O(2^n)$

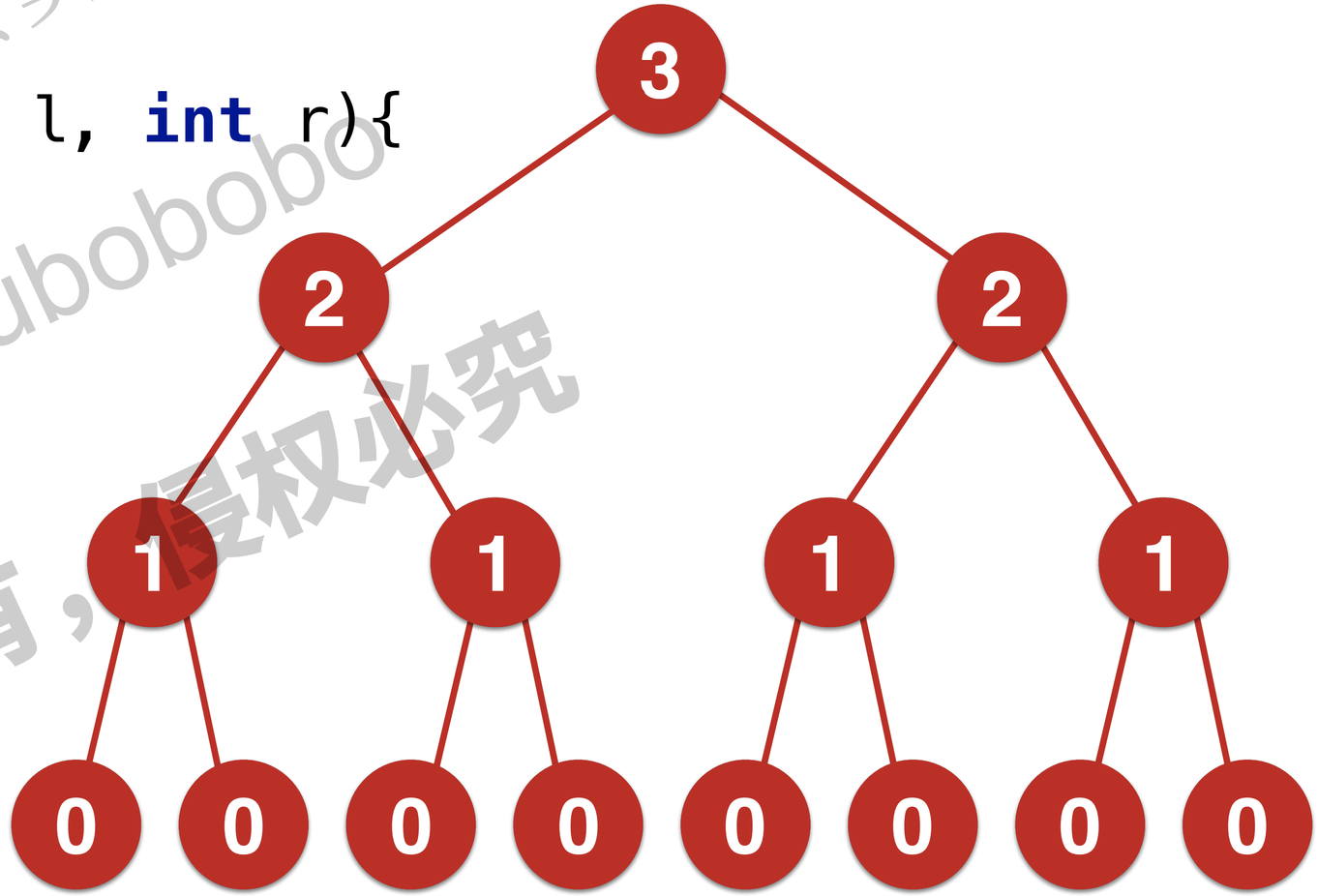


递归中进行多次递归调用

```
void mergeSort(int arr[], int l, int r){

    if( l >= r )
        return;

    int mid = (l+r)/2;
    mergeSort(arr, l, mid);
    mergeSort(arr, mid+1, r);
    merge(arr, l, mid, r);
}
```



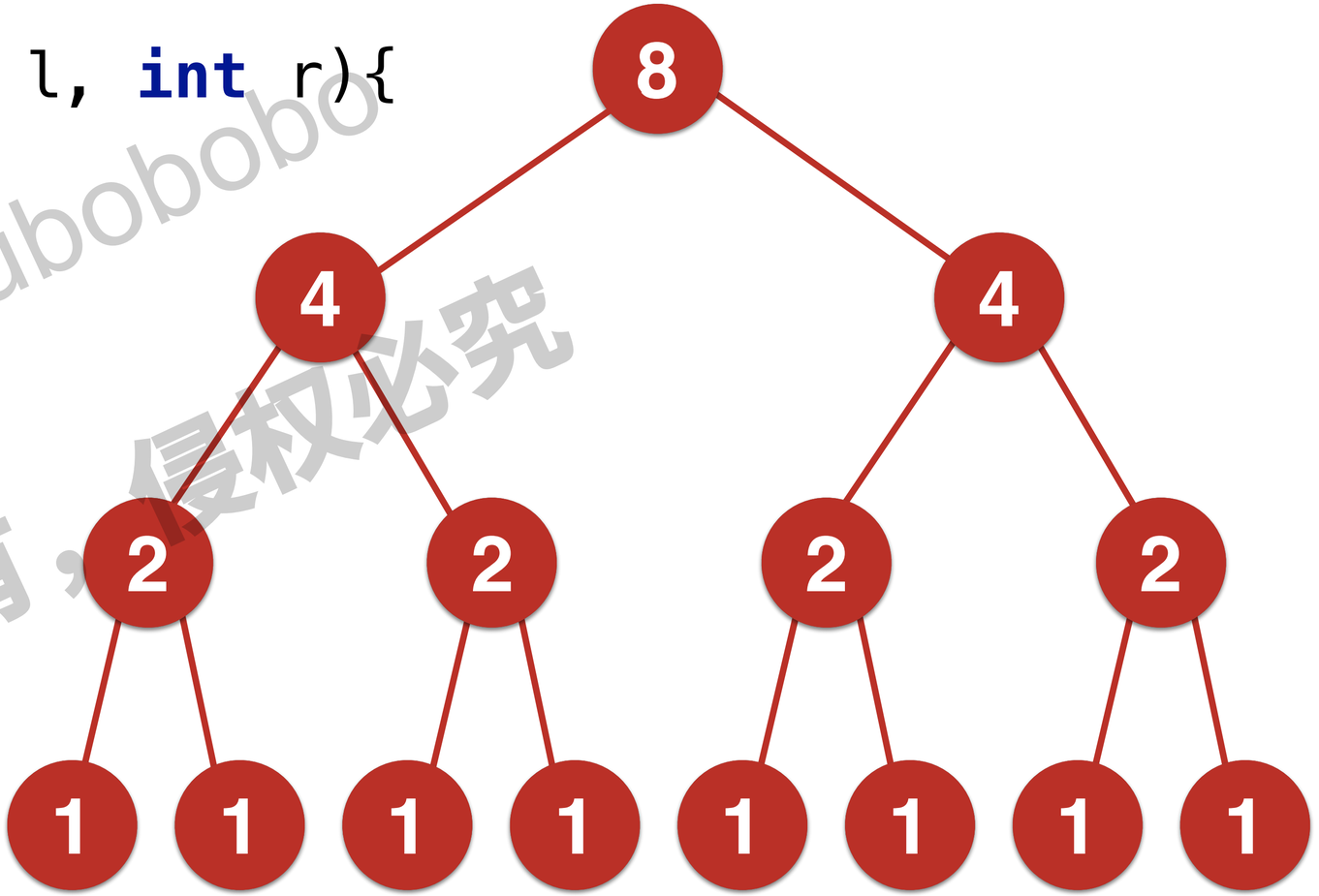
递归中进行多次递归调用

```
void mergeSort(int arr[], int l, int r){

    if( l >= r )
        return;

    int mid = (l+r)/2;
    mergeSort(arr, l, mid);
    mergeSort(arr, mid+1, r);
    merge(arr, l, mid, r);
}
```

时间复杂度：O(nlogn)



递归函数的时间复杂度

主定理

均摊复杂度分析
Amortized Time

动态数组 (Vector)

实践：编写动态数组

均摊复杂度分析

假设当前数组容量为n

1　1　1　　每一次添加耗费为O(1)　1　n

n

2　平均来看：O(1)

实践：验证均摊复杂度

实践：删除元素运用resize

均摊复杂度分析

假设当前数组容量为2*n

n+1　每一次删除耗费为O(1)　1　1

2　平均来看：O(1)

防止复杂度的震荡

假设当前数组容量为2*n

添加元素：n

删除元素：n

重复这个过程，无法均摊，复杂度为O(n)

复杂度震荡的解决方案

假设当前数组容量为2*n

当元素个数为数组容量的1/4时，resize

为再添加元素留出余地

实践：修改删除元素的resize

实践：验证均摊复杂度

均摊复杂度

动态数组

动态栈

动态队列

其他

欢迎大家关注我的个人公众号：是不是很酷

玩儿转算法面试

liuyubobobo