# 算法复杂度&数据结构

# 算法(Algorithm)

算法是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。

# 算法的特征

- 有穷性(Finiteness)
- 确切性(Definiteness)
- 输入项(Input)
- 输出项(Output)
- 可行性/有效性(Effectiveness)

# 算法复杂度

- 同一问题可用不同算法解决，而一个算法的质量优劣将影响到算法乃至程序的效率。
- 算法复杂度是指算法在编写成可执行程序后，运行时所需要的资源，资源包括时间资源和内存资源。

- 时间复杂度(Time complexity)

评估执行程序所需的时间。可以估算出程序对处理器的使用程度。

- 空间复杂度(Space complexity)

评估执行程序所需的存储空间。可以估算出程序对计算机内存的使用程度。

# 时间复杂度-大O表示法

- T(n)：一个算法中的语句执行次数，称为语句频度或时间频度。
- T(n) = O(f(n))：其中f(n) 表示每行代码执行次数之和，而 O 表示正比例关系。称它为算法的渐进时间复杂度，简称算法复杂度。

定义：设$f(n)$和$g(n)$是定义域为自然数集$N$上的函数。若存在正数$c$和$n_0$，使得对一切$n \geq n_0$都有$0 \leq f(n) \leq cg(n)$成立，则称$f(n)$的渐进的上界是$g(n)$，记作$f(n) = O(g(n))$。通俗的说n满足一定条件范围内，函数$f(n)$的阶不高于函数$g(n)$。

根据符号大O的定义，使用大O表示法评估得出的算法复杂度只是问题规模（n）充分大时的上界值。这个上界的阶越低，评估越精确，越有价值。

# 渐近记号Θ、O、o、Ω、ω关系

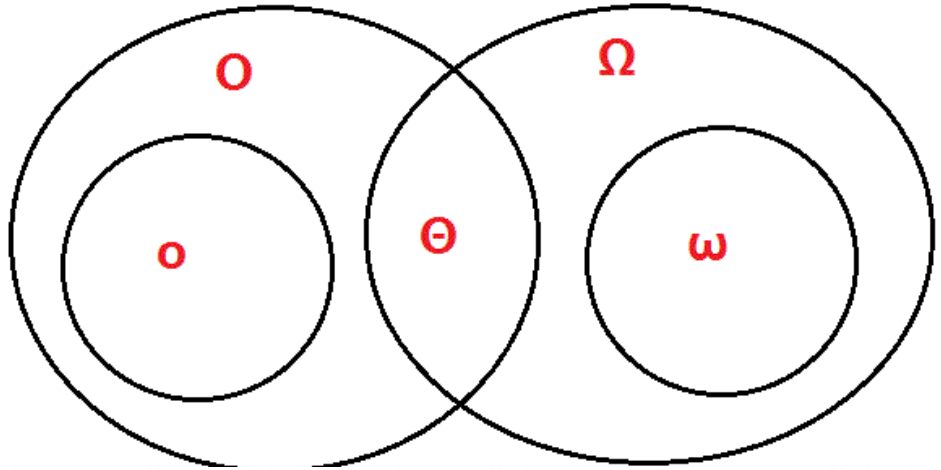
记号　　含义　　通俗理解

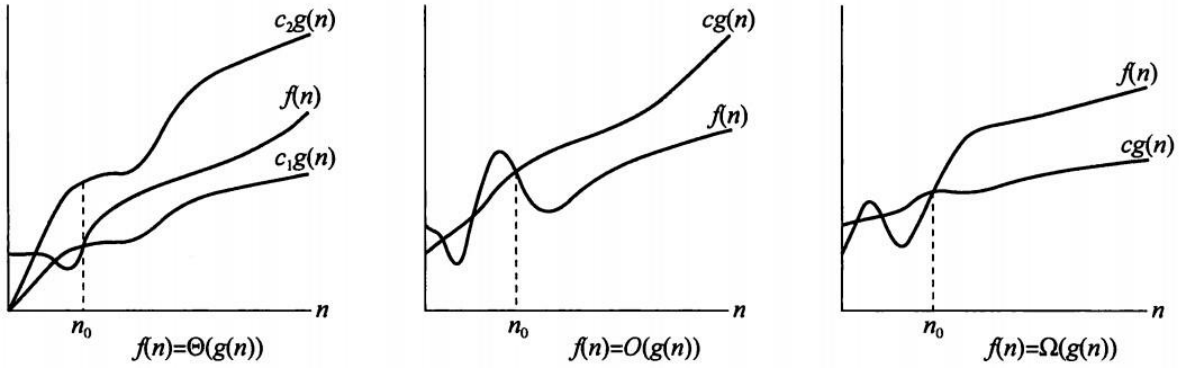
(1)Θ　　紧确界　　相当于"="

| Letter（字母） | Bound（限制） | Growth（增长） |
|---|---|---|
| (theta) Θ | upper and lower, tight[1] | equal[2] |
| (big-oh) O | upper, tightness unknown | less than or equal[3] |
| (small-oh) o | upper, not tight | less than |
| (big omega) Ω | lower, tightness unknown | greater than or equal |
| (small omega) ω | lower, not tight | greater than |

(5)ω 非紧的下界　　相当于">"

PS: Landau符号体系由德国数论学家保罗·巴赫曼（Paul Bachmann）在其1892年的著作《解析数论》首先引入，由另一位德国数论学家艾德蒙·朗道（Edmund Landau）推广。



# 时间复杂度

- 具体例子（伪代码（Pseudocode）是一种非正式的，类似于英语结构的，用于描述模块结构图的语言）
- 常数阶

```
int sum = 0, n = 100; //执行一次
sum = (1+n)*n/2; //执行一次
print (sum); //执行一次
```

- 线性阶

```
for(int i=0;i<n;i++){
    //时间复杂度为O(1)的算法
    ...
}
```

# 时间复杂度

- 对数阶

```
int number=1;
while(number<n){
        number=number*2;
        //时间复杂度为O(1)的算法
        ...
}
```

2^x = n

x=log~2~n

- 平方阶（循环嵌套&循环）

```
for(int i=0;i<n;i++){
   for(int j=0;j<n;i++){
     //复杂度为O(1)的算法
     ...
   }
}
```

# 时间复杂度

```
for(int i=0;i<n;i++){
    for(int j=i;j<n;i++){
      //复杂度为O(1)的算法
      ...
    }
 }
```

n+(n-1)+(n-2)+…+1
=n(n+1)/2
=n^2/2+n/2
=n^2/2
=n^2

作业：计算上次第一个作业（三角形）的时间复杂度

# 时间复杂度

常见的时间复杂度量级有:

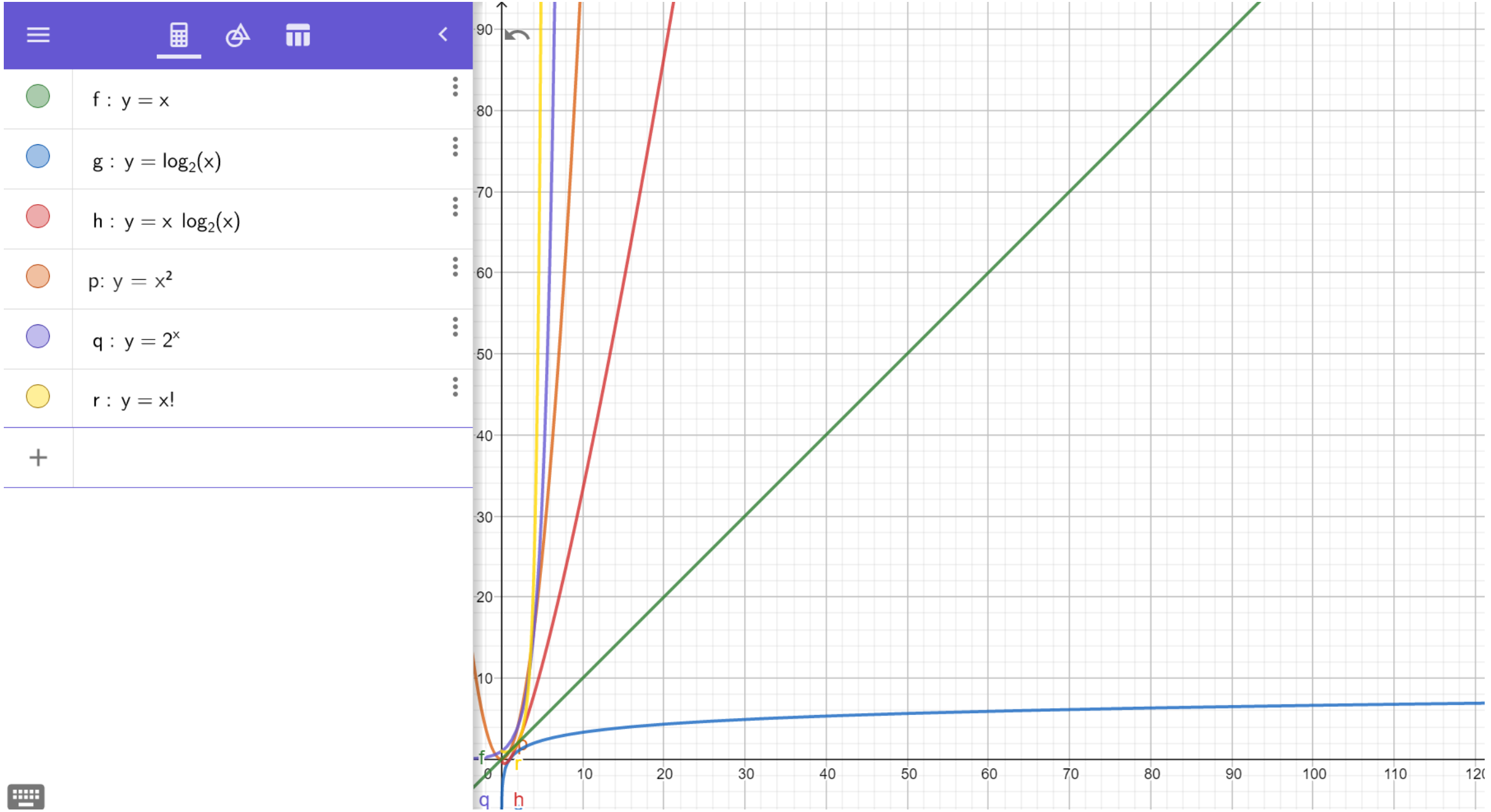
- 常数阶O(1)
- 对数阶O(logN)
- 线性阶O(n)
- 线性对数阶O(nlogN)
- 平方阶O($n^2$)
- 立方阶O($n^3$)
- K次方阶O(n^k)
- 指数阶O(2^n)
- 阶乘O(n!)

| n | logn | √n | nlogn | $n^2$ | $2^n$ | n! |
|---|---|---|---|---|---|---|
| 5 | 2 | 2 | 10 | 25 | 32 | 120 |
| 10 | 3 | 3 | 30 | 100 | 1024 | 3628800 |
| 50 | 5 | 7 | 250 | 2500 | 约10^15 | 约3.0*10^64 |
| 100 | 6 | 10 | 600 | 10000 | 约10^30 | 约9.3*10^157 |
| 1000 | 9 | 31 | 9000 | 1000 000 | 约10^300 | 约4.0*10^2567 |

从上至下依次的时间复杂度越来越大，执行的效率越来越低。

注：对数函数在没有底数时，默认底数为2。$\lg n = \log n = \log_2 n$

因为计算机中很多程序是用二分法实现的。


f : y = x
g : y = log2(x)
h : y = x log2(x)
p: y = x²
q : y = 2^x
r : y = x!
90
80
70
60
50
40
30
20
10
0
10
20
30
40
50
60
70
80
90
100
110
f
p
r
q
h

# 数据结构

- 数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。
- 研究方面：数据的逻辑结构；数据的存储结构；数据结构的运算。
- 数据的逻辑结构：指反映数据元素之间的逻辑关系的数据结构，其中的逻辑关系是指数据元素之间的前后件关系，而与他们在计算机中的存储位置无关。逻辑结构包括：集合，线性结构，树形结构，图形结构
- 数据的存储结构（物理结构）：指数据的逻辑结构在计算机存储空间的存放形式。不仅要存放各数据元素的信息，还要存放各数据元素之间的逻辑关系（前后件关系）。


数据结构
按逻辑结构分类
集合（无逻辑关系）
线性结构（线性表）
一维数组
队列
栈
非线性结构
树
图
多维数组
按存储结构分类
顺序存储结构
链式存储结构
索引存储结构
散列存储结构

# 线性结构与非线性结构

- 线性结构：数据元素之间存在着“一对一”的线性关系的数据结构。

- 非线性结构：相对应于线性结构，非线性结构的逻辑特征是一个结点元素可能对应多个直接前驱和多个后继。

# 数组（线性）

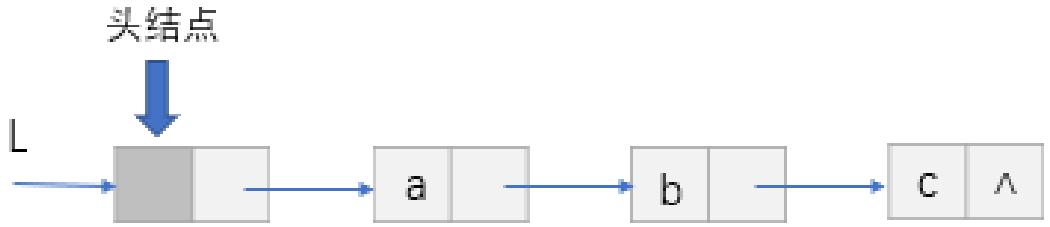
- https://www.w3school.com.cn/python/python_arrays.asp

- 数组的访问
- 数组的插入&删除

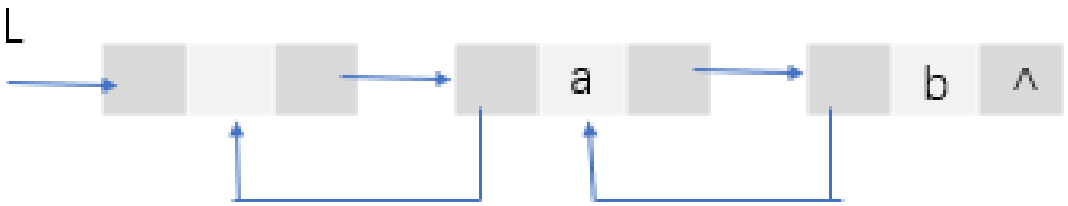
# 链表（线性）

- 链式表示的线性表也称链表，指用一组任意的存储单元存储线性表的数据元素。
- 特点：这组存储单元可连续可不连续，但每个结点除了存储对应的数据元素之外，还需存储一个直接后继的存储位置（最后一个结点也需指空即NULL）。其中存储数据元素信息的域称为数据域，存储直接后继存储位置的域称为指针域。

# 链表（单双循环）



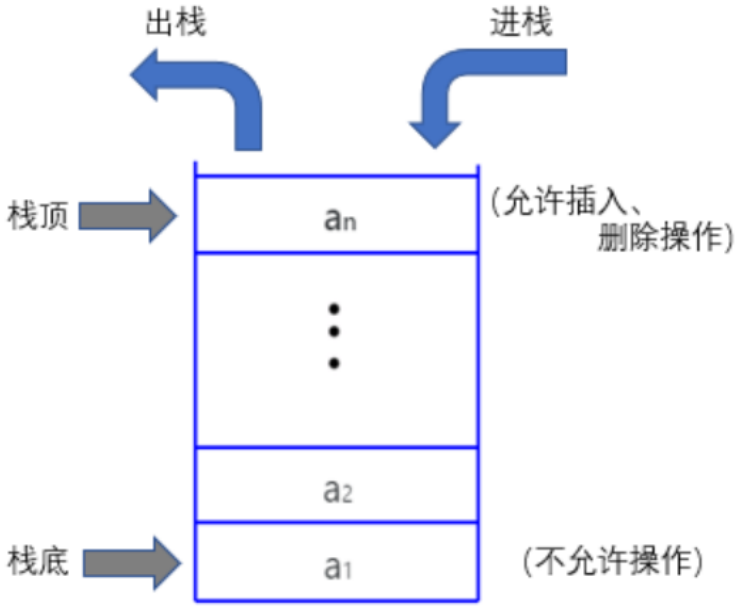

单链表




双向链表

# 栈（线性）

- 仅允许在表的一端进行插入和删除运算。这一端为栈顶，另一端为栈底。栈的特点：先进后出（后进先出）。
- 栈的元素：bottom（base）为栈底元素(非空情况下)，top为栈顶元素。top=0表示栈空。
- 栈的运算：进栈、入栈或压栈：插入新元素（把新元素放到栈顶元素的上面，使之成为新的栈顶元素）；出栈或退栈：删除元素（把栈顶元素删除掉，使其相邻的元素成为新的栈顶元素）。
- 读栈顶元素：将栈顶元素给一个指定的变量，此时指针无变化。




栈示意图

# 递归（算法）

- 程序调用自身的编程技巧称为递归（recursion）
  - 子问题须与原始问题为同样的事，且更为简单;
  - 不能无限制地调用本身，须有个出口，化简为非递归状况处理。
- n!，斐波那契数列，汉诺塔……

输入n>0:

```
Func(n){
        if(n=1){return n;}
        else{return n*Func(n-1)}
}
```

- 作业：函数&栈（类和对象）分别实现十进制到二进制的转换

# 递归练习(optional)

1. 倒序输出一个正整数
2. $m(n)=\frac{1}{2}+\frac{2}{3}+\ldots+\frac{n}{n+1}$
3. 斐波那契数列（递归和递推）
4. 汉诺塔

# 队列（线性）

- 只允许在表的前端进行删除操作，而在表的后端进行插入操作。进行插入操作的端称为队尾（rear指针），进行删除操作的端称为队头（front指针）。（在饭堂排队打饭）
- 队列的特点：先进先出（后进后出）
- 队列运算：
  - 入队运算：从队尾插入一个元素
  - 退队运算：从队头删除一个元素

队列示意图

- PS:栈和队列都是一种特殊的操作受限的线性表，只允许在端点处进行插入和删除。
- 因为队列先进先出的特点，在多线程阻塞队列管理中非常适用。

## Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ….

## Benefits.

- Client can't know details of implementation ⇒
  client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒
  many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

# 参考资料

- https://blog.csdn.net/so_geili/java/article/details/53353593