1, Algorithms and Data Structures

2, Primary interest: Running time (time-complexity). Secondary interest: Space (or "memory") usage(space-complexity).

3, Running-time depends on: input (size & instance), algorithm, software, hardware.

4, Big O

❖Given two positive functions $f(n)$ and $g(n)$ (defined on the nonnegative integers), we say f (n) is O(g(n)), written f (n) ∈O(g(n)), if there are constants c and n_0 such that:

$$f(n) \le c \cdot g(n) \text{ for all } n \ge n_0.$$

Example: $2n + 10$ is $O(n)$

- $2n + 10 \le cn$
- $(c - 2)\, n \ge 10$
- $n \ge 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$

5, The statement "f(n) is O(g(n))" means that the growth rate of f(n) is no more than the growth rate of g(n).

	$f(n)$ is $O(g(n))$	g(n) is O(f(n))
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

6, Constant O(1), Logarithmic O(log n), Linear O(n), Log-linear O(nlogn), Quadratic $O(n^2)$, Cubic $O(n^3)$, Polynomial $O(n^k)$, Exponential $O(a^n)$ (a >1), Factorial O(n!)

7, 时间复杂度排序（小 -> 大）

$c < \log N < n < n\log N < n^2 < n^3 < 2^n < 3^n < n!$ (对于同样的输入 n，复杂度越高的算法，速度越快)

8,

- Big-Oh

  f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)
- Big-Omega

  f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)
- Big-Theta

  f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

9, space complexity

```
int sum(int a[], int n) {
  int r = 0;
  for (int i = 0; i < n; ++i) {
     r += a[i];
   }
  return r;
}
```

requires N units for a, plus space for n, r and i, so it's O(N).

```
int sum(int x, int y, int z) {
   int r = x + y + z;
   return r;
}
```

requires 3 units of space for the parameters and 1 for the local variable, and this never changes, so this is O(1).

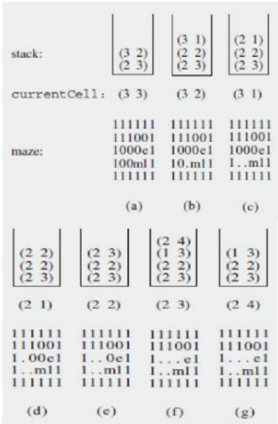
10, Stack (Last-in, first-out)

- *push(Obj)* : Insert object *Obj* onto the top of the stack.
- *pop()* : Remove (and return) the object from the top of the stack. An error occurs if the stack is empty.
- *initialize() : initialize a stack*
- *isEmpty()* : returns a **true** if stack is empty, **false** otherwise.
- *isFull()* : returns a true if stack is full, false otherwise.

11, infix notation: 4 + 3 * 9; postfix notation: x y + z * ((x + y) * z)

12, Exiting a Maze

```
 012345
0111111
1111001
21000e1
3100m11
4111111
```



13, queue (first-in, first-out)

▶ *enqueue(Obj)*: inserts object *Object* the *rear* of the queue.

▶ *dequeue()*: removes and returns the object from the *front* of the queue. An error occurs if the queue is *empty*.

• *size(): Return the number of objects in the queue.*

• *isEmpty()*: Returns true if queue is empty, and false otherwise.

• isFull(): Returns true if queue is full, and false otherwise.

• front(): Return, but do not remove, the object at the front of the queue. An error is returned if the queue is empty.

14, list

▶ *first()*: Return position of first element; error occurs if list S is empty.

▶ *last()*: Return the position of the last element; error occurs if list S is empty.

▶ *isFirst(p)*: Return **true** if element *p* is first item in list, **false** otherwise.

▶ *isLast(p)*: Return **true** is element *p* is last element in list, **false** otherwise.

▶ *before(p)*: Return the position of the element in S preceding the one at position *p*; error if *p* is first element.

▶ *after(p)*: Return the position of the element in S following the one at position *p*; error if *p* is last element.

▶ *replaceElement(p,e)*: *p* - position, *e* - element.

▶ *swapElements(p,q)*: *p,q* - positions.

▶ *insertFirst(e)*: *e* - element.

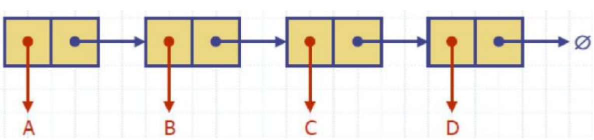
▶ *insertLast(e)*: *e* - element.

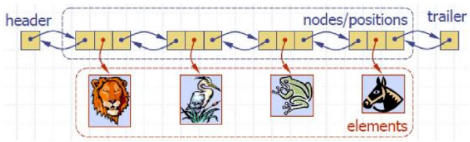
▶ *insertBefore(p,e)*: *p* - position, *e* - element.

▶ *insertAfter(p,e)*: *p* - position, *e* - element.

▶ *remove(p)*: *p* - position.

15, singly-linked list, doubly-linked list





16, tree (root, parent, child, siblings, leaf (external), internal node)

17, binary tree (rooted ordered tree; every node has at most 2 children; is **proper** if each internal node has exactly 2 children; labeled as left child or aright child)

▶ *root()*: return the root of the tree.

▶ *parent(v)*: return parent of *v*.

▶ *children(v)*: return links to *v*'s children.

▶ *size()*: return the number of nodes in the tree.

▶ *elements()*: return a list of all elements.

▶ *positions()*: return a list of addresses of all elements.

▶ *swapElements(u,v)*: swap elements stored at positions *u* and *v*.

▶ *replaceElements(v,e)*: replace element at address *v* with element *e*.

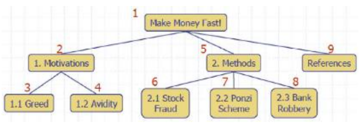
▶ *isInternal(v)*: test whether *v* is internal node.

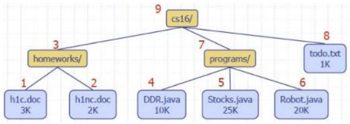
▶ *isExternal(v)*: test whether *v* is external node.

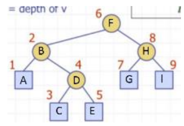
▶ *isRoot(v)*: test whether *v* is the root.

18, depth (v 的 depth: v 祖先的数量(除去 v)), height (一个 tree 的 height: tree 中最大的 leaf 的 depth)

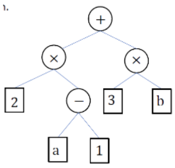
19, tree traversal (Binary trees have three kinds of traversals: preorder, postorder, inorder)







20, Binary tree associated with an arithmetic expression



external node 是 变量 或 常数; internal node 是 运算符号;

使用 inorder 遍历; 同一个 parent 下的 node 优先运算;

右图结果为 (2 * (a - 1) + (3 * b))

21, store binary tree in array

Children of A[i]: A[2*i+1], A[2*i+2]; parent of A[i]: A[$\lfloor (i-1)/2 \rfloor$] ($\lfloor 5/2 \rfloor$=2)

22, Ordered Dictionary (ordered by key)

findElement(k): position *k*
insertElement(k,e): position *k* , element *e*
removeElement(k): position *k*

23, binary search (O(logn))

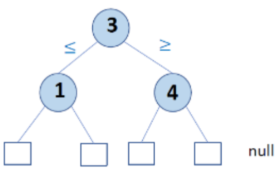
$mid = \lfloor (low + high)/2 \rfloor$

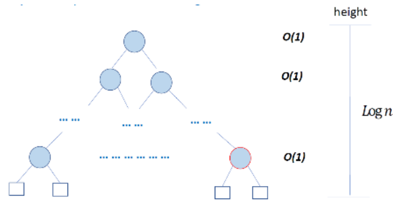
Initially, low = 1 and high = n

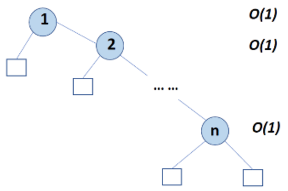
$k = key(mid)$, the search is completed successfully.
$k < key(mid)$, search continued with $high = mid - 1$.
$k > key(mid)$, search continued with $low = mid + 1$.

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ T(n/2) + b & \text{otherwise} \end{cases}$$

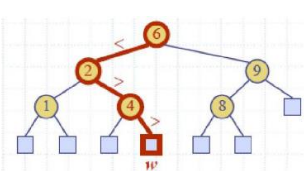
24, binary search tree (O(logn) – O(n))

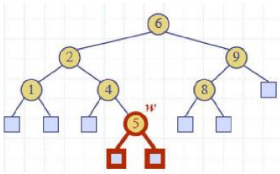




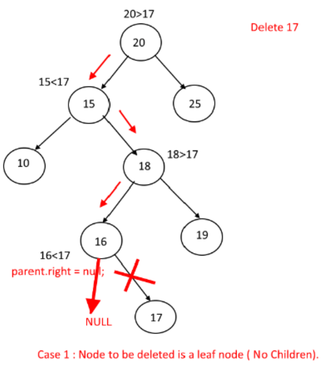


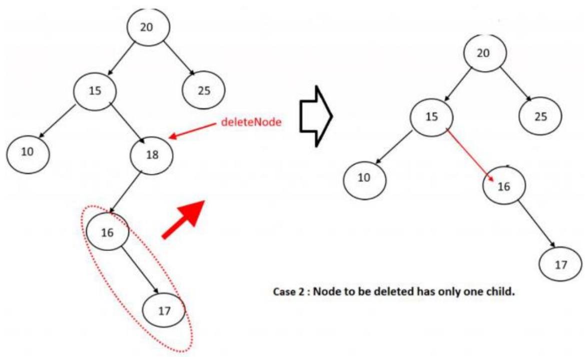
Insertion in BST (O(logn) – O(n))



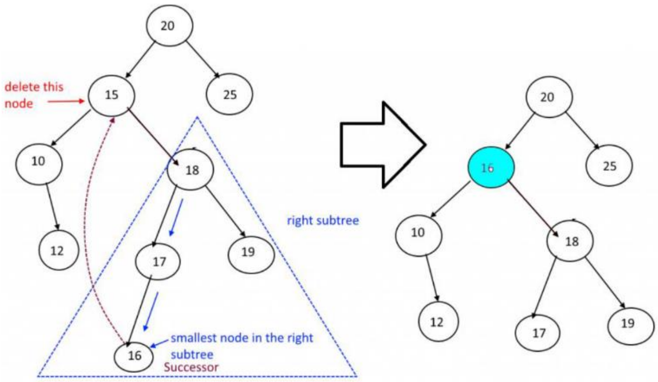


Deletion in BST (3 cases: leaf, one child, otherwise)

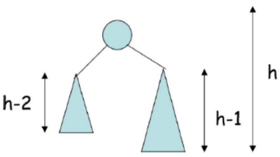




Otherwise, use inorder successor of removeElement to replace:



25, AVL-tree (无 order, O(logn))

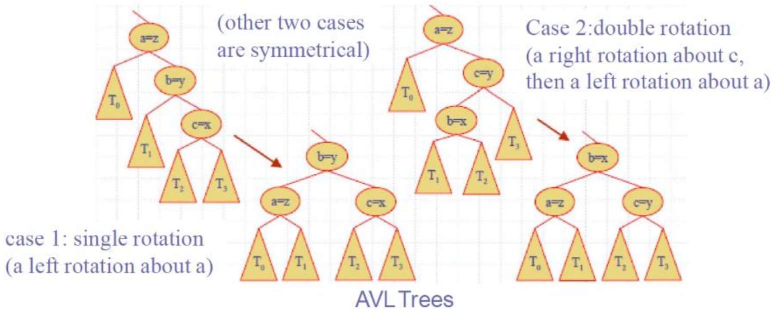


- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
  n(h) > 2n(h-2), n(h-2) > 2n(h-4), n(h-4) > 2n(h-6) => n(h) > 2^in(h-2i)
- Solving the base case we get: n(h) > $2^{h/2-1}$ h-2i = 1 or 2, depending on if h is even or odd: i = ⌈h/2⌉-1.
  Taking logarithms: h < 2log n(h) +2 log n(h) > log $2^{h/2-1}$, log n(h)>h/2-1
- Thus the height of an AVL tree is O(log n)

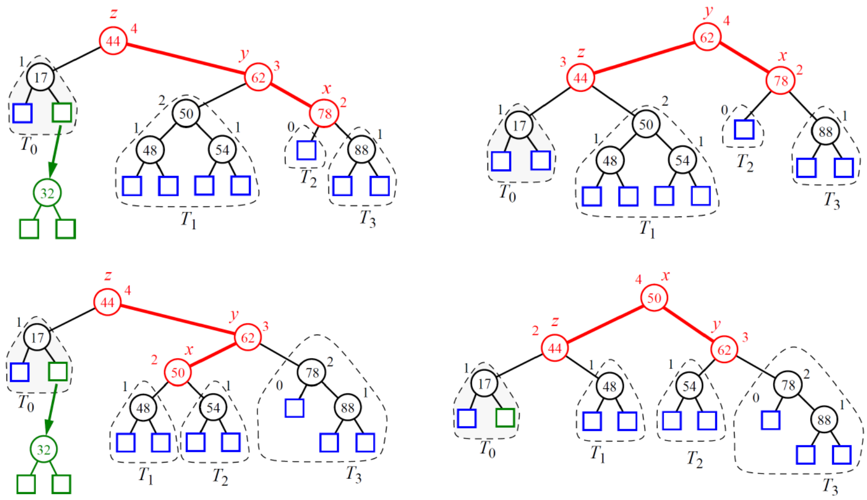
26, insertion in AVL-tree (O(logn))

find the first node x such that

its grandparent z is unbalanced node



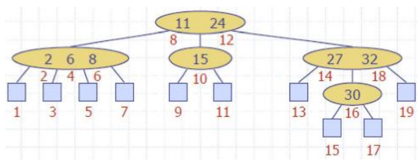
27, removal in AVL-tree (same as insertion) (O(logn))



28, Multi-Way Search Tree (inorder traversal)

Each internal node has at least 2 children, stores d-1

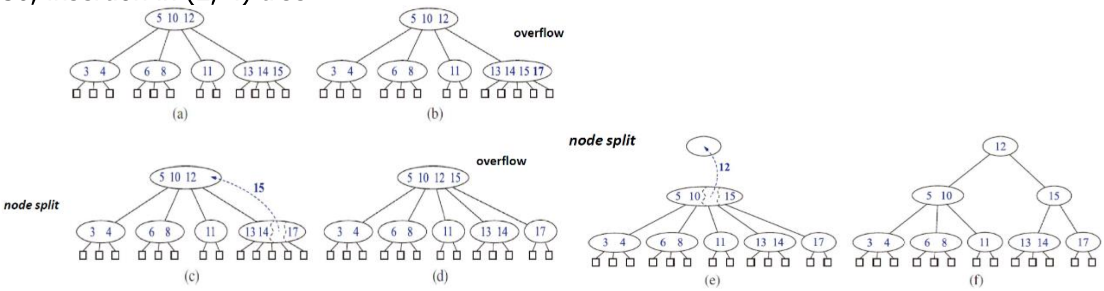
key-element items where d is the number of children



29, (2, 4) tree (Multi-Way Search Tree) (O(logn))

every internal node has 2-4 children (蓝框), all the external nodes have the same depth

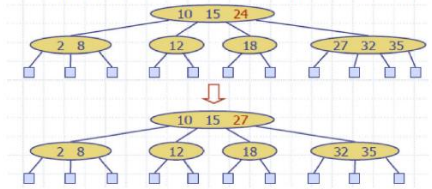
30, insertion in (2, 4) tree



31, deletion in (2, 4) tree

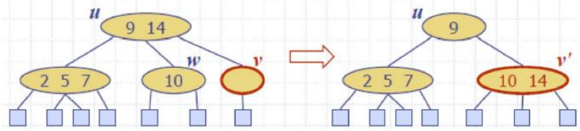
a), replace the item with its inorder successor or inorder predecessor, then delete.

delete key 24, we replace it with 27 (inorder successor)



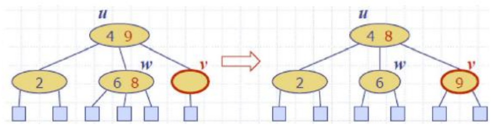
Case 1: the adjacent siblings of v are 2-nodes

- Fusion operation: we merge v with an adjacent sibling w and move an item from u to the merged node v'
- After a fusion, the underflow may propagate to the parent u



Case 2: an adjacent sibling w of v is a 3-node or a 4-node

- Transfer operation
  1. we move a child of w to v
  2. we move an item from u to v
  3. we move an item from w to u
- After a transfer, no underflow occurs



32, Priority Queues

insertItem(k,e): insert element *e* having key *k* into PQ.

removeMin(): remove minimum element.

minElement(): return minimum element.

minKey(): return key of minimum element.

How can we use a priority queue to perform sorting on a set *C*?
Do this in two phases:

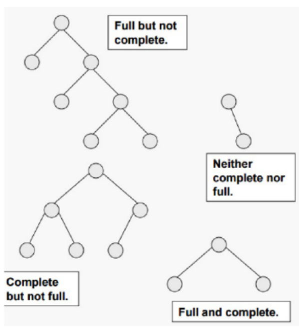
- *First phase*: *Put* elements of *C* into an initially empty priority queue, *P*, by a series of *n* insertItemoperations.
- *Second phase*: *Extract* the elements from *P* in *non-decreasing* order using a series of *n* removeMin operations.

33, heap (In a heap the elements and their keys are stored in an almost complete binary tree.)

34, complete & full binary tree

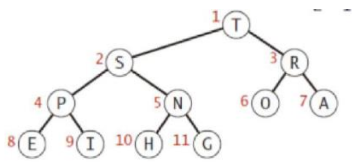
Definition: a binary tree T is *full* if each node is either a leaf or possesses exactly two child nodes.

Definition: a binary tree T with n levels is *complete* if all levels except possibly the last are completely full, and the last level has all its nodes to the left side

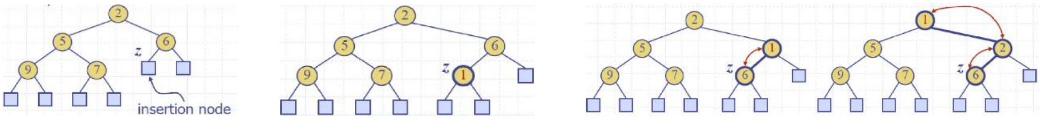


35, heap-order (max heap & min heap, binary heap) (O(logn))

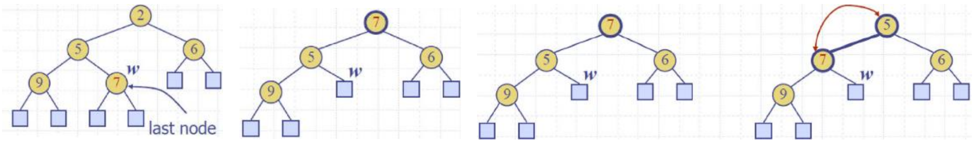
For any given node at position i:
- Its **Left Child** is at **[2*i]** if available.
- Its **Right Child** is at **[2*i+1]** if available.
- Its **Parent Node** is at **[⌊i/2⌋]** if available.



36, insertion in heap (O(logn))



37, deletion in heap (removeMin for min-heap: replace root key with **last node**) (O(logn))



38, heap: methods size, isEmpty, minKey, and minElement: O(1),

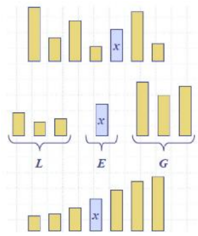
sort heap-based priority queue: O(nlogn)

39, Divide-and-Conquer

MergeSort: O(nlogn), n + 2 * n/2 + 4 * n/4 + ... = n * logn (height is logn) = nlogn

QuickSort: O(nlogn), worst-case: O(n^2)

- Divide: pick a random element x (called pivot) and partition S into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- Recur: sort L and G
- Conquer: join L,E and G



$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn & \text{otherwise} \end{cases}$$

$$T(n) = bn + bn\log n$$

40, The Master method

$$T(n) = \begin{cases} c & \text{if } n < d \\ aT(n/b) + f(n) & \text{otherwise.} \end{cases}$$

wherein $d \geq 1, a > 0, c > 0, b > 1$

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

In the **The Master Theorem**:

1. Case 1: applies where $f(n)$ is polynomially smaller than the special function $n^{\log_b a}$.
2. Case 2: applies where $f(n)$ is asymptotically close to the special function $n^{\log_b a}$.
3. Case 3: applies where $f(n)$ is polynomially larger than the special function $n^{\log_b a}$.

*f(n) is polynomially smaller than g(n) if f(n)=O(g(n)/n^ϵ) for some ϵ>0.

*f(n) is polynomially larger than g(n) if f(n)=Ω(g(n)n^ϵ) for some ϵ>0

$$T(n) = 2T(n^{1/2}) + \log n.$$

Unfortunately, this equation is not in a form that allows us to use the master method. We can put it into such a form, however, by introducing the variable $k = \log n$, which lets us write

$$T(n) = T(2^k) = 2T(2^{k/2}) + k.$$

Substituting into this the equation $S(k) = T(2^k)$, we get that

$$S(k) = 2S(k/2) + k.$$

Now, this recurrence equation allows us to use master method, which specifies that $S(k)$ is $O(k\log k)$. Substituting back for $T(n)$ implies $T(n)$ is $O(\log n \log\log n)$.

41, greedy method (does not always lead to an optimal solution)

42, The Knapsack Problem (heap: O(logn), greedy: O(nlogn))

Object:	1	2	3	4
Benefit:	7	9	9	2
Weight:	3	4	5	2
Value index:	2.33	2.25	1.8	1

Value index = bi / wi

43, Interval Scheduling (O(nlogn)): Select "finish first"

44, Dynamic Programming:

{0 –1} Knapsack Problem (worst-case: O(2n))

Given an integer W and a set S of n items, each of which has a positive benefit and a positive integer weight, we can find the highest benefit subset of S with total weight at most W in *O(nW)* time.

Let $S_k = \{i \mid i = 1, 2, \ldots, k\}$ denotes a set containing the first k items, and define $S_0 = \emptyset$.

Let $B[k, w]$ be the *maximum total benefit* obtained using a subset of S_k, and having total weight *at most* w.

Then we have $B[0, w] = 0$, for each $w \le W$, and

$$B[k,w] = \begin{cases} B[k-1, w] & \text{if } w < w_k \\ \max\{B[k-1, w], b_k + B[k-1, w - w_k]\} & \text{otherwise} \end{cases}$$

Example: x-axis: w, y-axis: k; when B[4, 10], wk = w4 = 6

i	1	2	3	4
b_i	25	15	20	36
w_i	7	2	3	6

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	25	25	25	25
2	0	0	15	15	15	15	15	25	25	40	40
3	0	0	15	20	20	35	35	35	35	40	45
4	0	0	15	20	20	35	36	36	51	56	56

45, 公式

等比等差求和: $S_n = \frac{a_1(1-q^n)}{1-q} \, (q \ne 1)$ $\quad S_n = \frac{n \times (a_1 + a_n)}{2}$

求和: $\sum_{k=1}^{n} k = 1+2+3+\cdots\cdots+(n-1)+n = \frac{n(n+1)}{2}$

$\sum_{k=1}^{n} k(k+1) = (1\times2)+(2\times3)+(3\times4)+\cdots\cdots+(n-1)\times n + n\times(n+1) = \frac{n(n+1)(n+2)}{3}$

$\sum_{k=1}^{n} k^2 = 1^2+2^2+3^2+\cdots\cdots+n^2 = \frac{n(n+1)(2n+1)}{6}$

$\sum_{k=1}^{n} 1 = n \quad \sum_{k=i}^{n} k = \sum_{k=1}^{n} k - \sum_{k=1}^{i-1} k \quad \sum_{k=0}^{n-1} (ar^k) = a\left(\frac{1-r^n}{1-r}\right)$

对数运算: $log_a(MN) = log_a M + log_a N$ $\quad log_a(1/N) = -log_a N$ $\quad \log_a b = \frac{\log_c b}{\log_c a}$

$log_a(M/N) = log_a M - log_a N$ $\quad log_a M^n = n log_a M$ $\quad \log_a b * \log_b a = 1$