1.Recursive Algorithms（递归算法）

Recursion involves a procedure calling itself to solve subproblems of a smaller size. These smaller subproblems can then be combined in some way to get a solution to a larger problem.
递归即自己调用自己。

Recursive procedures require a base case that can be solved directly without using recursion.
即递归过程中的最后一步。

Recurrence Relations（递归关系）:

Recurrence relations sometimes allow us to define the running–time of an algorithm in the form of an equation. 以方程式的方式来展现算法的运行时间。

递归关系可能以多种形式出现，例如：

- $C(n) = 3 \cdot C(n-1) + 2 \cdot C(n-2) + C(n-3)$ where $C(1) = 1, C(2) = 3, C(3) = 5$
- 斐波那契数列（The Fibonacci numbers）

斐波那契数列：第三项等于前两项之和，其中前两项均为1。

伪码：

```
Algorithm: fibonacci numbers
Input: upper limit n
Output: The n-th term of Fibonacci
int fib (int n){
if n <=2
   return n;
else
   return fib(n-1)+fib(n-2);
}
```

有些情况下递归中调用的子问题会被多次运算，导致算法的运行时间大大增加。

在输入值很大时，电脑可能无法运行递归算法，因为重复的函数调用会占用太多内存。为了避免这种情况的出现，可以将计算的值先储存起来，等到需要时候再进行调用。

```
Algorithm: fibonacci numbers
Input: upper limit Nmax
Int f(int Nmax)
{
f1 ← 1;
f2 ←1;
 for n←3:(Nmax){
    fn← f2 + f1;
      f1 ← f2;
      f2 ← fn;
  return fn;
}}
```

2.Asymptotic notation（渐进表示法）

Asymptotic notation allows characterization of the main factors affecting running time.

即估计基本操作的数量，可以用来比较两个算法的运行时间。

An algorithm with an asymptotically slow running time is beaten in the long run by an algorithm with an asymptotically faster running time. 即随着时间的增加，运行速度可能会发生相应的改变。

增长率（Growth rate）：

Linear ≈n Quadratic ≈ n^2 Cubic ≈ n^3

增长率不受常数和低阶项的影响，例如 10^2n+10 5 is a linear function,10^5n^2 +108 $n$is a quadratic function。

大O标记法（“Big–Oh” Notation）：

定义如下图：

❖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 $O(n)$

例子如下：

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

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

Example: the function n^2 is not $O(n)$

- $n^2 \le cn$
- $n \le c$
- The above inequality cannot be satisfied since c must be a constant

大O标记法的例子：

- 7n-2
  7n-2 is **O(n)**
  need **c** > 0 and $n_0 \ge 1$ such that $7n - 2 \le cn$ for $n \ge n_0$
  this is true for **c** = 7 and $n_0 = 1$

- $3n^3 + 20n^2 + 5$
  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need **c** > 0 and $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le cn^3$ for $n \ge n_0$
  this is true for **c** = 4 and $n_0 = 21$

大O标记法给出了函数增长率的上界（upper bound），f(n) is O(g(n))意味着f(n)的增长率不比g(n)大。

We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ 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

常见的例子有：

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

如果f(n)是d次的多项式，那么f(n)的大O标记法表示O(n^d)，舍弃低次项和常数项：

2n is O(n) instead of O(n^2)，3n+5 is O(n) instead of O(3n)。更多例子：

1. $13n^3 + 7n \log n + 3$ is $O(\_\ n^3\ \_)$.

   Proof: $13n^3 + 7n \log n + 3 \leq 16n^3$, for n ≥1

2. $3 \log n + \log \log n$ is $O(\_\log n\_\_)$.

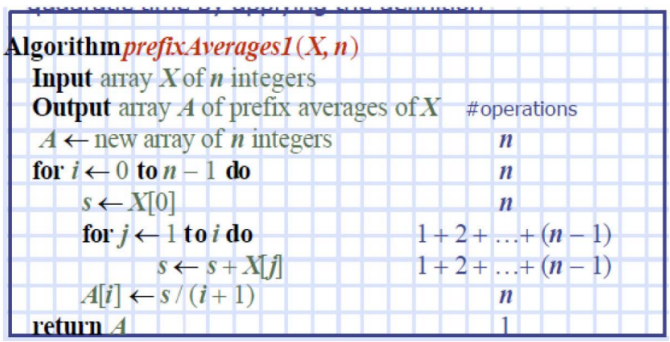
   Proof: $3 \log n + \log \log n \leq 4 \log n$, for n ≥ 2

3. 2^{70} is $O(\_\ 1\ \_)$.

   Proof: $2^{70} \leq 2^{70} * 1$, for n ≥1.

算法的渐进分析以大O标记法的方式来表示算法的运行时间。大O标记法表示的是在最差情况下需要的基本操作的数量。比如Lecture 1最后的例子中的O(n)是根据7n−2得出的。

再比如：



Algorithm prefixAverage1 runs in O(?)

该例子的结果为O(n^2)。

We say that f (n) is **Ω(g(n))** (big–Omega) if there are real constants c and n0 such that: **f (n) ⩾ cg(n)** for all n ⩾ n0. We say that f (n) is Θ(g(n)) (Theta) if f (n) is Ω(g(n)) and f (n) is also O(g(n)).

即分别决定上界和下界。

- 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)

3.Space Complexity（空间复杂度）

空间复杂度是对算法所需的储存大小的度量方法，即在最坏情况下算法需要的存储空间大小。空间复杂度主要涉及到在输入大小是N时，用大O标记法表示空间需求如何增长。例子如下：

```
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).

```
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).