UNIT 1

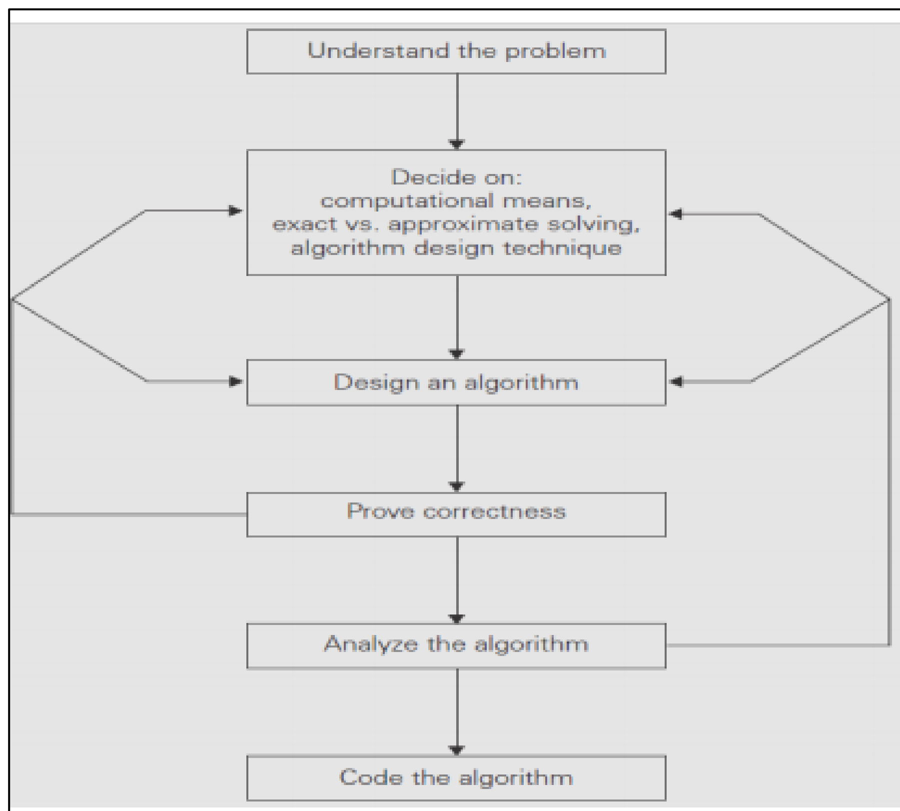
1.1 What is an Algorithm?

An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Characteristics of an algorithm

- Input: Zero / more quantities are externally supplied.
- Output: At least one quantity is produced.
- Definiteness: Each instruction is clear and unambiguous.
- Finiteness: The algorithm must terminates after a finite number of steps.
- Efficiency: Every instruction must be very basic and runs in short time.

1.2 Fundamentals of Algorithmic Problem Solving



Understanding the Problem:

Read the problem's description carefully and ask questions if you have any doubts about the problem, do a few small examples by hand, think about special cases, and ask questions again if needed.

An input to an algorithm specifies an instance of the problem the algorithm solves. If you fail to do this step, your algorithm may work correctly for a majority of inputs but crash on some "boundary" value. Remember a correct algorithm is not one that works most of the time, but one that works correctly for all legitimate inputs.

Ascertaining the Capabilities of the Computational Device:

RAM (random-access machine): instructions are executed one after another, one operation at a time, use sequential algorithms.New computers: execute operations concurrently, use parallel algorithms.

Also, consider the speed and amount of memory the algorithm would take for different situations.

Choosing between Exact and Approximate Problem Solving

An algorithm that can solve the problem exactly is called an exact algorithm.

The algorithm that can solve the problem approximately is called an approximation algorithm.

Ex:extracting square roots, solving nonlinear equations, and evaluating definite integrals etc

Algorithm Design Techniques

Provides guidance for designing algorithms for new problems or problems for which there is no satisfactory algorithm.

Designing an Algorithm and Data Structures

Data structures are important for both design and analysis of algorithms.

Methods of Specifying an Algorithm:

1. Pseudo code 2.Flowcharts

Proving an Algorithm's Correctness:

The algorithm yields a required result for every legitimate input in a finite amount of time.A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.For an approximation algorithm, we usually would like to show that the error produced by the algorithm does not exceed a predefined limit.

Analysing an Algorithm:

Time and space efficiency, simplicity, generality.

Coding an algorithm

As a rule, a good algorithm is a result of repeated effort and rework.

2. Fundamentals of the Analysis of Algorithm Efficiency

2.1 The Analysis Framework

The research experience has shown that for most problems, we can achieve much more spectacular progress in speed than in space.

- **Measuring an Input's Size**

When measuring input size for algorithms solving problems such as checking primality of a positive integer n. Here, the input is just one number, and it is this number's magnitude that determines the input size. In such situations, it is preferable to measure size by the number b of bits in the n's binary representation:

$$b=\lfloor \log 2n \rfloor+1$$

- **Units for Measuring Running Time**

Identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

The established framework for the analysis of an algorithm's time efficiency suggests measuring it by counting the number of times the algorithm's basic operation is executed on inputs of size n.

- **Orders of Growth**

$$\log a^{n} = \log a^{b} * \log b^{n}$$

Algorithms that require an exponential number of operations are practical for solving only problems of very small sizes.

- **Worst-Case, Best-Case, and Average-Case Efficiencies**

If the best-case efficiency of an algorithm is unsatisfactory, we can immediately discard it without further analysis.

The direct approach for investigating average-case efficiency involves dividing all instances of size n into several classes so that for each instance of the class the number of times the algorithm's basic operation is executed is the same. Then a probability distribution of inputs is obtained or assumed so that the expected value of the basic operation's count can be found.

Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.

The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.

2.2 Asymptotic Notations and Basic Efficiency Classes

O-notation

Definition: A function t(n) is said to be in O(g(n)), denoted $t(n) \in O(g(n))$, if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that $t(n) \leq cg(n)$ for all $n \geq n0$

Ω-notation

Definition: A function t(n) is said to be in Ω(g(n)), denoted t(n) ∈ Ω(g(n)), if t(n) is bounded below by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n0 such that t(n) ≥ cg(n) for all n ≥ n0

Θ-notation

Definition: A function t(n) is said to be in Θ(g(n)), denoted t(n) ∈ Θ(g(n)), if t(n) is bounded both above and below by some constant multiples of g(n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integer n0 such that c2g(n) ≤ t(n) ≤ c1g(n) for all n ≥ n0.

Useful Property Involving the Asymptotic Notations

- If t1(n) ∈ O(g1(n)) and t2(n) ∈ O(g2(n)), then t1(n) + t2(n) ∈ O(max{g1(n), g2(n)}) (also true for other two notations).

The definitions of the various asymptotic notations are closely related to the definition of a limit. As a result, $\lim_{n\to\infty} f(n) / g(n)$ reveals a lot about the asymptotic relationship between f and g, provided the limit exists. The table below translates facts about the limit of f / g into facts about the asymptotic relationship between f and g.

$\lim_{n\to\infty} f(n)/g(n) \neq 0, \infty$	$\Longrightarrow$	$f = \Theta(g)$	$\lim_{n\to\infty} f(n)/g(n) = 1$	$\Longrightarrow$	$f \sim g$
$\lim_{n\to\infty} f(n)/g(n) \neq \infty$	$\Longrightarrow$	$f = O(g)$	$\lim_{n\to\infty} f(n)/g(n) = 0$	$\Longrightarrow$	$f = o(g)$
$\lim_{n\to\infty} f(n)/g(n) \neq 0$	$\Longrightarrow$	$f = \Omega(g)$	$\lim_{n\to\infty} f(n)/g(n) = \infty$	$\Longrightarrow$	$f = \omega(g)$

- L'Hôpital's rule:

L'Hôpital's rule states that for functions f and g which are differentiable on an open interval I except possibly at a point c contained in I, if $\lim_{x\to c} f(x) = \lim_{x\to c} g(x) = 0$ or $\pm\infty$, $g'(x) \neq 0$ for all x in I with $x \neq c$, and $\lim_{x\to c} \frac{f'(x)}{g'(x)}$ exists, then

$$\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}.$$

Stirling's Formula

The factorial function $n!$ is approximated by

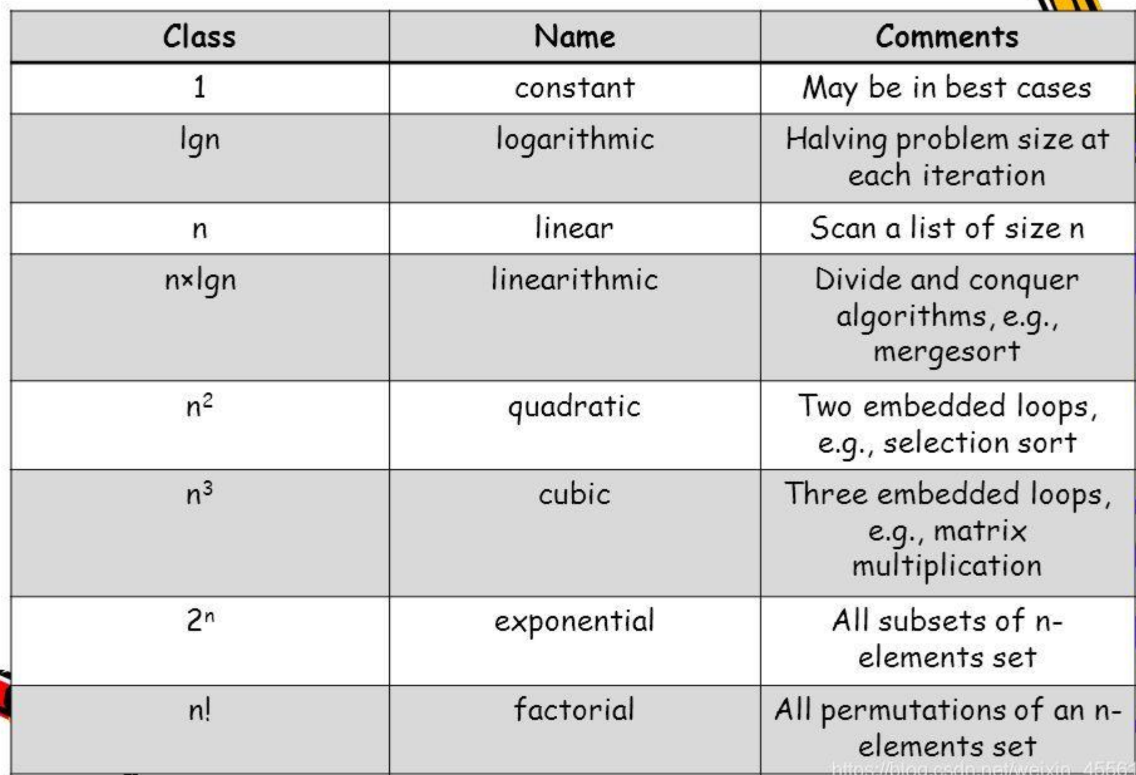
$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Furthermore, for any positive integer n, we have the bounds

$$\sqrt{2\pi}\, n^{n+\frac{1}{2}} e^{-n} \le n! \le e\, n^{n+\frac{1}{2}} e^{-n}.$$

https://blog.csdn.net/weixin_45561634

2 Basic efficiency classes



Basic Efficiency Classes

Class	Name	Comments
1	constant	May be in best cases
lgn	logarithmic	Halving problem size at each iteration
n	linear	Scan a list of size n
n×lgn	linearithmic	Divide and conquer algorithms, e.g., mergesort
n^2	quadratic	Two embedded loops, e.g., selection sort
n^3	cubic	Three embedded loops, e.g., matrix multiplication
2^n	exponential	All subsets of n-elements set
n!	factorial	All permutations of an n-elements set

3. Brute Force and Exhaustive Search

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection Sort
We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last $n - 1$ elements and exchange it with thesecond element, putting the second smallest element in its final position. After $n - 1$ passes, the list is sorted.

ALGORITHM Selection Sort(A[0..n − 1])
//Sorts a given array by selection sort

```
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
fori ← 0 to n − 2 do
        min ← i
        for j ← i + 1 to n − 1 do
                if A[j] < A[min]
                        min ← j
        swap A[i] and A[min]
```

The basic operation is the key comparison A [j] < A [m i n] A[j] < A[min]A[j]<A[min]. The number of times it is executed depends only on the array size n nn and is given by the following sum:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2}$$

Selection sort is a $\Theta(n^2)$algorithm on all inputs.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position
on the list. The next pass bubbles up the second largest element, and so on,
until after n − 1 passes the list is sorted.

```
ALGORITHM Bubble Sort(A[0..n − 1])
//Sorts a given array by bubble sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in non decreasing order
fori ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
                if A[j + 1] < A[j]
                        swap A[j] and A[j + 1]
```

The number of key comparisons:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} (n-1-i) = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Sequential search

```
ALGORITHM SequentialSearch2(A[0..n], K)
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] not equal to K do
        i ← i + 1
        ifi< n return i
else return −1
```

Exhaustive Search

Exhaustive search is simply a brute-force approach to combinatorial problems.

Travelling Salesman Problem

Find the shortest tour through a given set of n nn cities that visits each city exactly once before returning to the city where it started.

Weighted graph -> finding the shortest Hamiltonian circuit of the graph: a cycle that passes through all the vertices of the graph exactly once.

Knapsack Problem

Given n items of known weights w 1 , w 2 , … , w n
and values v 1 , v 2 , … , v n and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Generate all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets.

Assignment Problem

There are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the jth job is a known quantity C[i, j] for each pair i , j = 1 , 2 , … , n. The problem is to find an assignment with the minimum total cost.