Chapter 1

ALGORITHM'S ANALYSIS

What is an Algorithm?

- A set of clear instructions to solve a problem.
- Algorithms are usually written in an English-like language called Pseudo code.
- Algorithms are not programs!

Data Structure

Data

Data structure

- Representation of the logical relationship existing between individual elements of data.
- A specialized format for organizing, managing and storing data in memory that considers not only the elements stored but also their relationship to each other.

Properties of Algorithms

Finiteness	terminates after a finite number of steps
Definiteness	Each step must be clear.
Input	Valid inputs must be clearly specified.
Output	The data that result upon completion of the algorithm must be specified.
Effectiveness	Steps must be sufficiently simple and basic.

How do you know an algorithm is correct

→ produces the correct output on every input

Comparison between two algorithms Computing gcd(m, n)

Euclid's algorithm	Consecutive integer checking algorithm

Repeat gcd(m,n) = gcd(n, m mod n)
until the second number becomes 0.
Example:
gcd(60,24) = gcd(24,12) = gcd(12,0) = 12
while $n \neq 0$ do
    $r \leftarrow m$ mod n
    $m \leftarrow n$
    $n \leftarrow r$
endWhile
return *m*
	Step 1: $t \leftarrow \min\{m,n\}$
	Step 2: if(m / t) == *0*, go to Step 3;
	    otherwise, go to Step 4
	Step 3 if(n / t) == *0*, return *t* and stop;
	    otherwise, go to Step 4
	Step 4 Decrease *t* by 1 and go to Step 2

Why Performance Matter?

Suppose $N = 10^6$

A PC can read/process N records in 1 sec.

If an algorithm does N*N computation, then it takes **1M seconds = 11 days!!!**

Faster Algorithm vs. Faster CPU

- Algorithm S1 sorts n keys in **2n2 instructions**
- Computer C1 executes **1 billion instruc/sec**
  - When n = 1 million
  - Number of instructions $2n^2$: $2 * (10^6)^2 = 2 * 10^{12}$
  - Time to execute: $2 * 10^{12} / 10^9 = 2000$ sec
- Algorithm S2 sorts n keys in **50nlog2n instructions**
- Computer C2 executes **10 million instruc/sec**
  - When n = 1 million
  - Number of instructions : $50 * 10^6 \log_2 (10^6)$
  - Time to execute:

$$50 * 10^6 \log_2 (10^6) / 10^6 = 50 \log_2 (10^6) = 1000 \text{ sec}$$

A faster algorithm running on a slower machine will always win for large enough instances

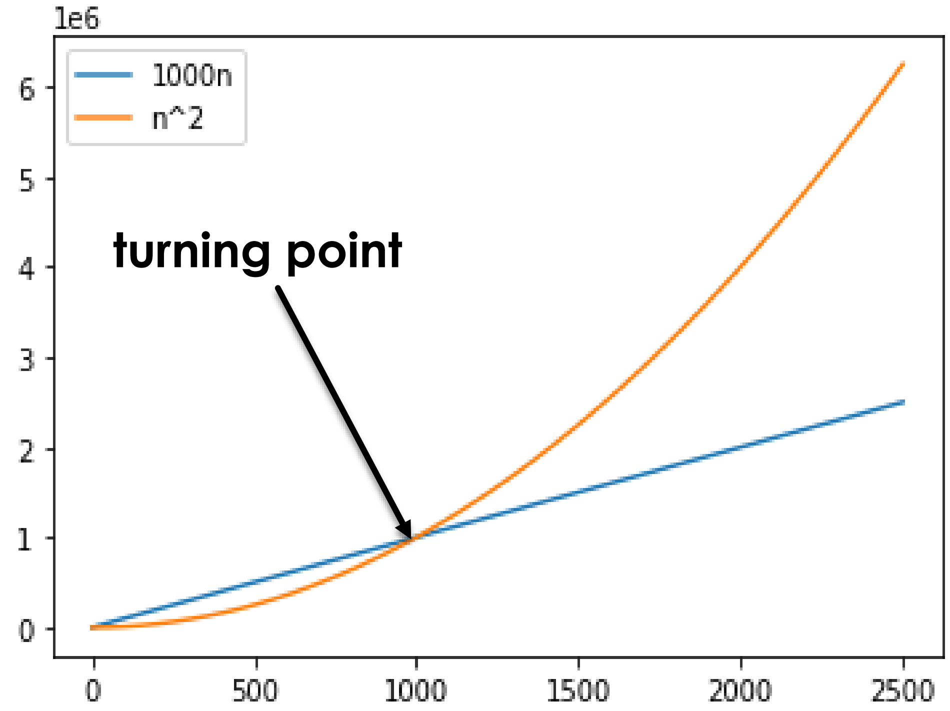
The growth rate

The rate at which running time increases as a function of input is called Rate of Growth.

Which is smaller f(n) = 1000 N or g(n) $= N^2$?

- For small values of N, we see that f(N) is greater than g(N).
- But g(N) grows at faster rate.
- Thus g(N) will eventually be the larger function.

The **turning point** is N =1000



N	F(n) = 1000 n	g(n) = N^2
100	100,000	10,000
500	500,000	250,000
700	700,000	490,000
1000	1000,000	1000,000
1500	1,500,000	2,250,000
2000	2,000,000	4,000,000
3000	3,000,000	9,000,000

Factors Affecting Choosing An Algorithm

- Running time (Speed)
- Storage usage
- Graphical User Interface (GUI). (Ease of use)
- Security
- Maintenance
- Portability
- Open Source

Algorithm Performance

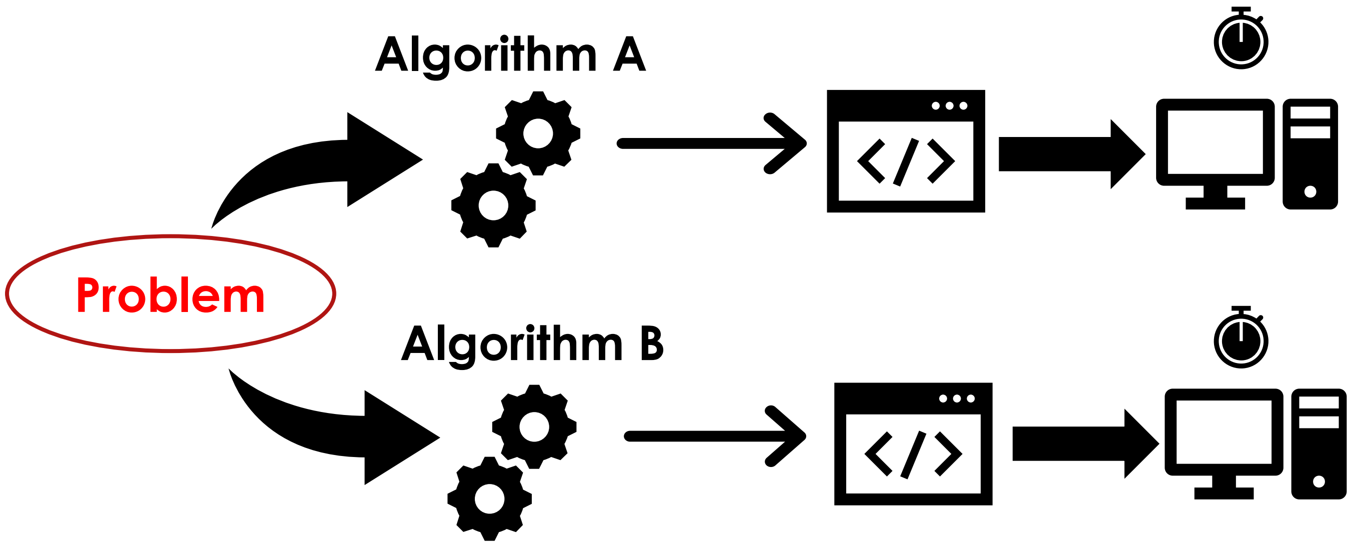
Performance = Efficiency = Complexity

↓

Two ways to compute an algorithm performance

Empirical Method.	**Analytical Method.**

Empirical Method



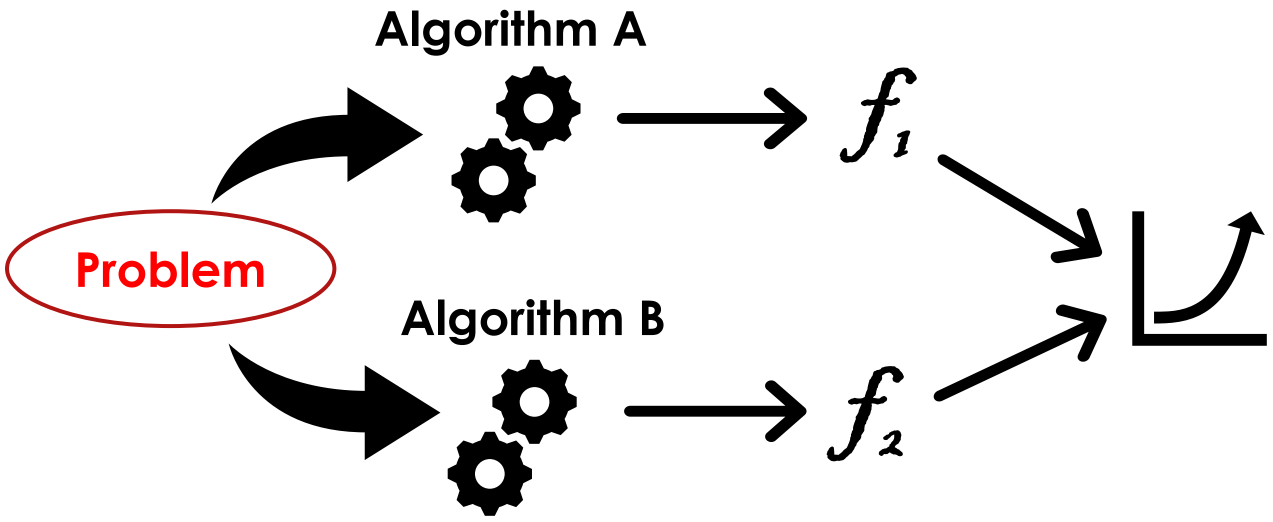
Factors affecting running speed:

- CPU speed.
- RAM size.
- Hard Disk Drive (HDD).
- Graphics Card.
- Operating System (OS).
- Compiler.
- Environment (Backup, Antivirus, ...etc).

! **Accuracy**

To compare two algorithms — Convert to the same programming language — Run on machines with same configuration — Compare the running time

Analytical Method



Each algorithm is mapped to a function

Functions are compared in terms of their growth rate to decide which is better.

Mathematical Review

Exponents

- $X^0 = 1$
- $X^a X^b = X^{(a+b)}$
- $X^a / X^b = X^{(a-b)}$
- $X^{-n} = 1 / X^n$
- $(X^a)^b = X^{ab}$

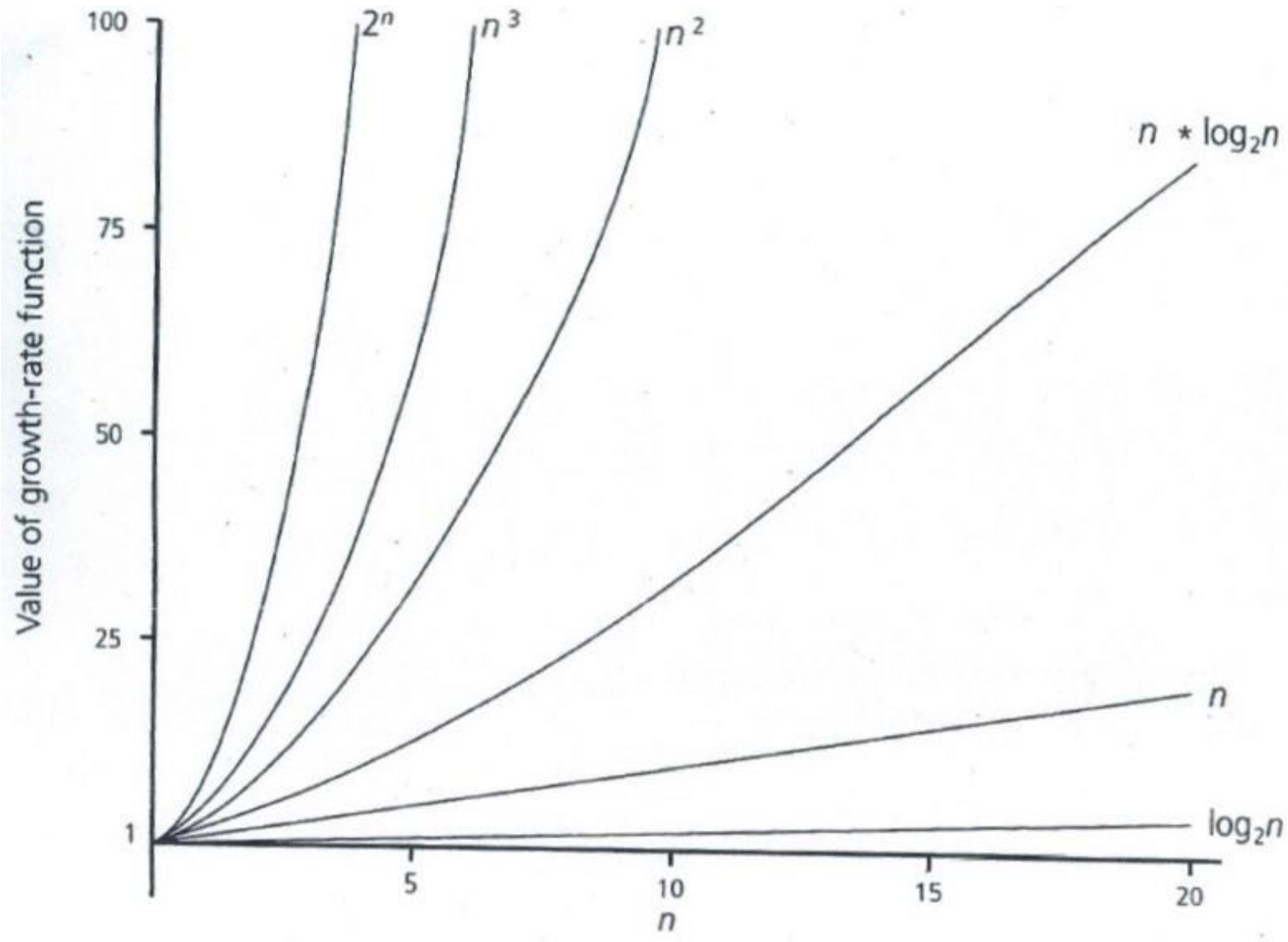
Logarithms

- $\log_a X = Y \Leftrightarrow a^Y = X$, $a > 0, X > 0$
- $\log_a 1 = 0$ because $a^0 = 1$
- $\log X$ means $\log_2 X$
- $\lg X$ means $\log_{10} X$
- $\ln X$ means $\log_e X$ ($e \approx 2.71828183$)

Logarithms

- $\log_a(XY) = \log_a X + \log_a Y$
- $\log_a(X/Y) = \log_a X - \log_a Y$
- $\log_a(X^n) = n\log_a X$
- $\log_a b = (\log_2 b)/ (\log_2 a)$
- $a^{\log_a x} = x$

Typical Function Families



Typical Functions: Examples

Function	n = 10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Growth rate	Name
C	Constant, we write O(1)
logN	Logarithmic
N	Linear
NlogN	Linearithmic
N^2	Quadratic
N^3	Cubic
2^N	Exponential
N!	Factorial

Typical Growth Rates

Function Growth

- $\lim_{n\to\infty}(n) = \infty$
- $\lim_{n\to\infty}(n^a) = \infty, a > 0$
- $\lim_{n\to\infty}\left(\frac{1}{n}\right) = 0$
- $\lim_{n\to\infty}\left(\frac{1}{n^a}\right) = 0, a > 0$
- $\lim_{n\to\infty}(log(n)) = \infty$
- $\lim_{n\to\infty}(a^n) = \infty, a > 0$

Function Growth

- $\lim(f(x) + g(x)) = \lim(f(x)) + \lim(g(x))$
- $\lim(f(x) \times g(x)) = \lim(f(x)) \times \lim(g(x))$
- $\lim(\frac{f(x)}{g(x)}) = \frac{\lim(f(x))}{\lim(g(x))}$
- $\lim(\frac{f(x)}{g(x)}) = \lim(\frac{f'(x)}{g'(x)})$

Examples

- $\lim_{n\to\infty}\left(\frac{n}{n^2}\right) = 0$
- $\lim_{n\to\infty}\left(\frac{n^2}{n}\right) = \infty$
- $\lim_{n\to\infty}\left(\frac{n^2}{n^3}\right) = 0$
- $\lim_{n\to\infty}\left(\frac{n^3}{n^2}\right) = \infty$
- $\lim_{n\to\infty}\left(\frac{n}{\frac{n+1}{2}}\right) = \infty$

Asymptotic Growth Classes

Asymptotic growth : **The rate of growth of a function**

Given a particular differentiable function f(n), all other differentiable functions fall into three classes:

growing with the same rate	Theta
growing faster	Little oh, Big Oh
growing slower	Little omega, Big Omega

Theta

f(n) and g(n) have the **same rate of growth**, if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \qquad 0 < c < \infty$$

Notation: f(n) = Θ(g(n))

Little oh

f(n) grows slower than g(n), (or g(n) grows faster than f(n)) if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Notation: f(n) = o(g(n))

Little omega

f(n) grows **faster** than g(n), (or g(n) grows slower than f(n)) if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

Notation: f(n) = ω(g(n))

if g(n) = o(f(n)) then f(n) = ω(g(n))

Examples

Compare log n and log n^2

- lim(log n / log n^2) = ½ ➔ **log n^2 = Θ(log n)**

Compare n and (n+1)/2

- lim(n / ((n+1)/2)) = 2 ➔ **(n+1)/2 = Θ(n)**

Compare n^2 and n^2+ 6n

- lim(n^2 / (n^2+ 6n))= 1 ➔ **n^2+6n = Θ(n^2)**

Θ(n^3)

- n^3
- $5n^3 + 4n$
- $105n^3 + 4n^2 + 6n$

Θ(n2)

- n^2
- $5n^2 + 4n + 6$
- $n^2 + 5$

Θ(log n)

- log n
- log n^2
- log (n + n^3)

Big oh

$f(n) = O(\ g(n)\ )$

if f(n) grows with

same rate or slower than g(n)

$f(n) = \Theta(g(n))$ **or** $f(n) = o(g(n))$

$N^2 = o(N^2)$ little oh ➔ False because $N^2 = \theta(N^2)$

$N^2 = O(N^2)$ Big oh ➔ True

Big omega

The inverse of Big-Oh is **Ω**

If $g(n) = O(f(n)) \rightarrow f(n) = \Omega\,(g(n))$

f(n) grows faster or with the same rate as g(n): **$f(n) = \Omega\,(g(n))$**

Rules

$$If\ T_1(N) = O(f(N))\ and\ T_2(N) = O(g(N)),\ then$$

$$(a)\ T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$$

$$(b)\ T_1(N) \times T_2(N) = O(f(N) \times g(N))$$

- **<u>For example</u>:**
  - **If $T_1(N) = O(N^2)$ and $T_2(N) = O(N)$ then**
  - **(a) $T_1(N) + T_2(N) = O(N^2)$**
  - **(b) $T_1(N) * T_2(N) = O(N^3)$**
- **Do not say $T(N) = O(2N^2)$ or $T(N) = O(N^2 + N)$. The correct form is $T(N) = O(N^2)$.**

Computing the Running Time

Computing $\sum_{i=1}^{N} i^3$

int sum (int N)	We ignore the costs of calling the function, thus the total is 1 to initialize *N*
{	
int i, partialSum ;	The declarations count for no time.
partialSum = 0;	The cost is 1 to initialize Sum.
for (i = 1; i <= N; i++)	1 to initialize i, N+1 for all the tests (i <= N), and N for all the increments, which is 2N + 2.
partialSum += i * i * i;	4 units per time executed and is executed N times, for a total of 4N units.
return partialSum;	We ignore the costs of returning.
}	
Total cost: $6N + 4 = O(N)$	

General Rules

Rule 1 the for loop:

The running time of a for loop is at most [the running time of the statements inside the for loop (including tests)] x [the number of iterations].

For example:

```
for ( i = 1; i <= n ; i++ )
    k++
```

- Total cost is 1 + N + 1 + N + N * 1 = 3N + 2, which has O(N).

General Rules

- **Rule 2 Nested for loops:**
- The total running time of a statement inside a group of nested loops is the running time of the statement multiplied by the product of the sizes of all the for loops.
- For example:

```
for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
        k++
```

- Total cost is:
  - 1 + (N+1) + N + N * (1 + (N+1) + N + N * 1) = 3N2 + 4N + 2, which has O(N2)

General Rules

- $O(N^2)$ program fragment:

```
sum = 0
for ( j = 0; j < n; j++ )
   for ( k = 0; k < j; k++ )
      sum++;
```

$$\sum_{j=0}^{N-1}\sum_{k=0}^{j-1} 1 = \sum_{j=0}^{N-1} j = \sum_{j=1}^{N-1} j = \frac{(N-1)N}{2}$$

$O(N^3)$ program fragment:

```
sum = 0
for ( j = 0; j < n; j++ )
   for ( k = 0; k < n * n; k++ )
      sum++;
```

$$\sum_{j=0}^{N-1}\sum_{k=0}^{N^2-1} 1 = \sum_{j=0}^{N-1} N^2 = N^2 N = N^3$$

General Rules

Rule 3 Consecutive Statements:

```
for (i = 0; i < n; i++)
    a[ i ] = 0;
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      a[ i ] += a[ j ] + i + j;
```

The program fragment, which has $O(N)$ work followed by $O(N^2)$ work, is also $O(N^2)$.

General Rules

- **Rule 4 Condition Statement:** For the fragment

```
if ( condition )
 S1
else
 S2
```

the running time of a if/else statement is never more the running time of the test plus the larger of the running times of S1 and S2.

Logarithms in the Running Time

Binary Search: Given an integer X and integers $A_0, A_1, \ldots, A_{N-1}$, which are presorted and already in memory, find i such that $A_i = X$, or return $i = -1$ if X is not in the input.

Binary search algorithm: Compare the search element with **the middle element** of the array, If not equal, then apply binary search to half of the array (if not empty) where the search element would be.

binarySearch Running Time

Recurrence equation

$$T(N) = T(\frac{N}{2}) + c$$

Solving the recurrence:

$$\begin{aligned} T(N) &= T(\frac{N}{2}) + c \\ &= T(\frac{N}{2^2}) + 2c \\ &= T(\frac{N}{2^3}) + 3c \\ &\vdots \\ &= T(\frac{N}{2^k}) + kc \end{aligned}$$

With k = log *N* (i.e. $2^k = N$), we have

Thus, the running time is O(log N).

```
int binarySearch( int a[ ], int  x )
{
        int low = 0, high = a.length( ) - 1;
        while( low <= high )
        {
                int mid = ( low + high ) / 2;
                if( a[ mid ] < x )
                        low = mid + 1;
                else if( a[ mid ] > x )
                        high = mid - 1;
                else
                        return mid; // Found
        }
        return -1;
}
```

Logarithms in the Running Time

Exponentiation: The following algorithm for computing X^N where X and N are two positive integers.

$$X^N = \begin{cases} 1 & \text{if } N = 0 \\ X & \text{if } N = 1 \\ X^{N/2}.X^{N/2} & \text{if } N \text{ is even} \\ X^{(N-1)/2}.X^{(N-1)/2}.X & \text{if } N \text{ is odd} \end{cases}$$

The algorithm:

Logarithms in the Running Time

Exponentiation: The following algorithm for computing X^N where X and N are two positive integers.

The algorithm:

```
long pow( long x, int n )
{
   if( n == 0 )
      return 1;
   if( n == 1 )
      return x;
   if( isEven( n ) )
      return pow( x * x, n / 2 );
   else
      return pow( x * x, n / 2 ) * x;
}
```

Running Time: $O(\log N)$

Worst Case, Best Case, and Average Case

When sorting a list of numbers in ascending order, we might have one of the following cases:

- Worst Case: The list is sorted in descending order, then maximum number of swaps is performed.
- Average Case: The list is unsorted, then average number of swaps is performed.
- Best Case: The list is sorted in ascending order, then minimum number of swaps is performed.

Best, Worst and Average Case Analysis Sequential Search

- Best case : Searching key element present at first index
- Best Case Time : 1 O(1), B(n) = O(1)
- Worst Case: Searching a key element, present at last index
- Worst Case Time= n , O(n), W(n)= O(n)
- Average case: (all possible case time) / no. of cases
- Avg. Time = (1+2+3+4+… +n)/n = ((n(n+1))/2)/n = (n+1)/2
- Average(n) = (n+1)/2

8	6	12	7	5	18	9

Useful summation formulas and rules

$$\sum_{i=a}^{b} constant = constant \times (b - a + 1)$$

$$\sum_{i=0}^{N} i = \sum_{i=1}^{N} i = \frac{N^2 + N}{2}$$

$$\sum_{i=0}^{N-1} i = \sum_{i=1}^{N-1} i = \frac{N^2 - N}{2}$$

$$\sum_{i=0}^{N+1} i = \sum_{i=1}^{N+1} i = \frac{N^2 + 3N + 2}{2}$$

$$\sum_{i=0}^{N} i^2 = \sum_{i=1}^{N} i^2 = \frac{2N^3 + 3N^2 + N}{6}$$

$$\sum_{i=1}^{N} 2^i = 2^{N+1} - 1$$