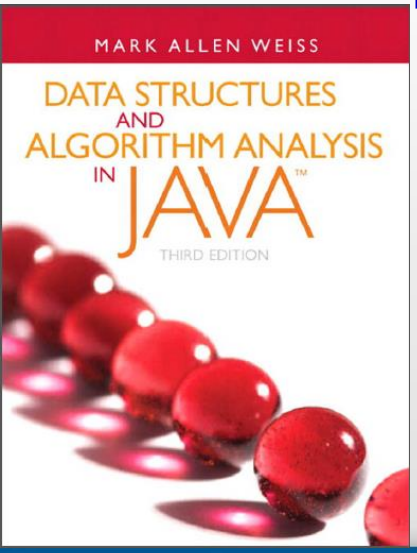
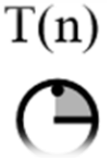
Algorithm Analysis: 2.1 Mathematical Background

On board
p. 30,31,32 textbook



Asymptotic Notation



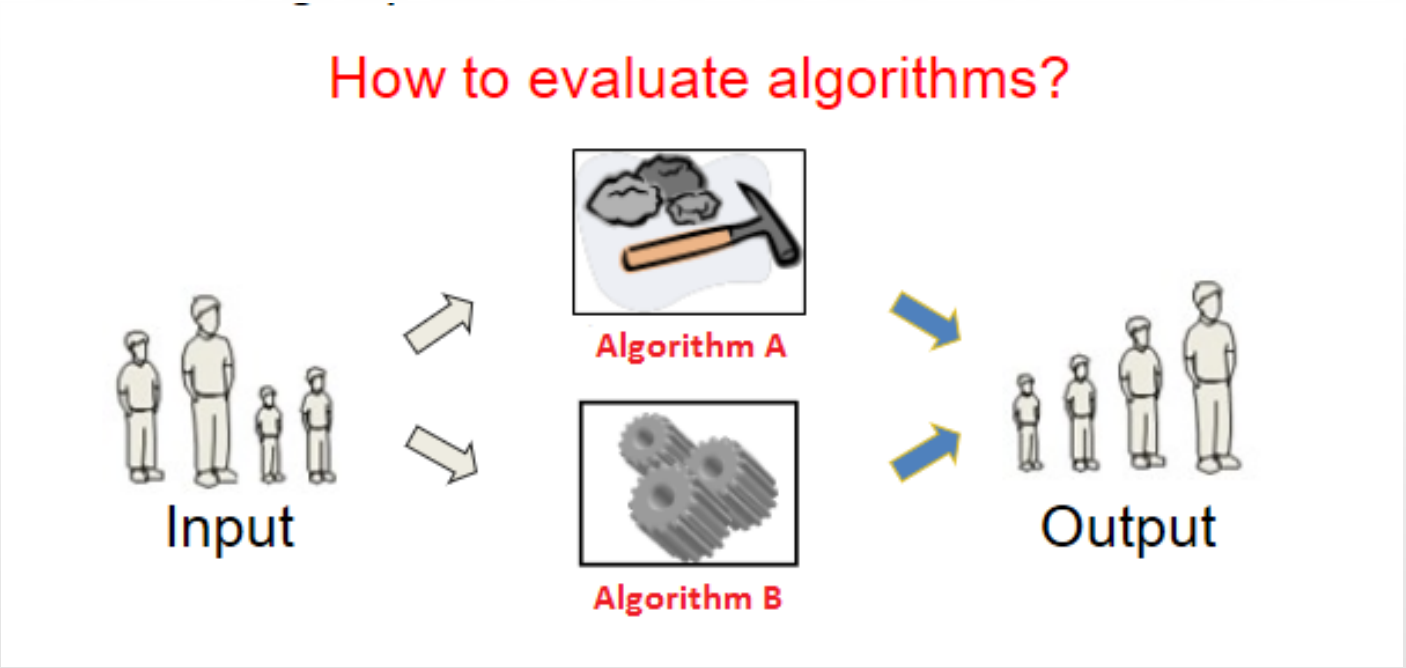
Asymptotic Notation is a formal notation for discussing and analyzing "classes of functions".

- Its use in analyzing runtimes

- " Big-O" notation : O(n)
- " Big-Omega of n " : Ω(n)
- " Theta of n" : Θ(n)

Asymptotic Notation

- Its use in analyzing runtimes.



Asymptotic Notation

We will refer to the running time as *T(N)*

Definition 2.1.

$T(N) = O(f(N))$ if there are positive *constants* c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.

Definition 2.2.

$T(N) = \Omega(g(N))$ if there are positive *constants* c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.

Definition 2.3.

$T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$.

Definition of Order Notation

- **Upper bound**: $T(n) = O(f(n))$ Big-O

  Exist constants c and n_0 such that

  $T(n) \leq c\, f(n)$ for all $n \geq n_0$

- **Lower bound**: $T(n) = \Omega(g(n))$ Omega

  Exist constants c and n_0 such that

  $T(n) \geq c\, g(n)$ for all $n \geq n_0$

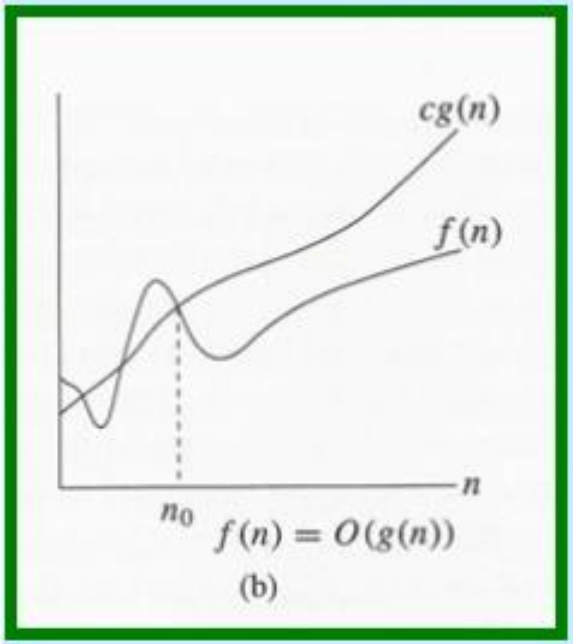
- **Tight bound**: $T(n) = \Theta(f(n))$ Theta

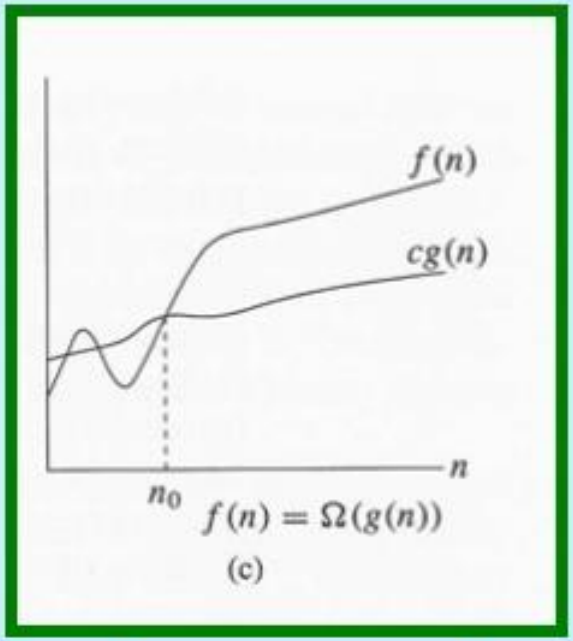
  When both hold:

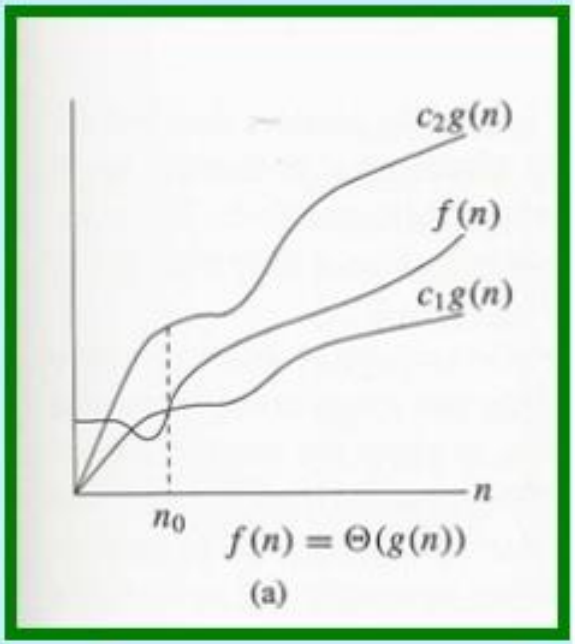
  $T(n) = O(f(n))$

  $T(n) = \Omega(f(n))$

Asymptotic Notation







Asymptotic Upper Bound

Big-O Notation

Asymptotic Lower Bound

Big-Ω Notation

Tight Asymptotic Bound

Big-Θ Notation

Example: Upper Bound

Claim: $n^2 + 100n = O(n^2)$

Proof: Must find c, n_0 such that for all $n > n_0$,

$$n^2 + 100n \leq cn^2$$

Let's try setting $c = 2$. Then

$$n^2 + 100n \leq 2n^2$$

$$100n \leq n^2$$

$$100 \leq n$$

So we can set $n_0 = 100$ and reverse the steps above.

Example: Lower Bound

Claim: $n^2 + 100n = \Omega(n^2)$

Proof: Must find c, n_0 such that for all $n > n_0$,

$$n^2 + 100n \geq cn^2$$

Let's try setting $c = 1$. Then

$$n^2 + 100n \geq n^2$$

$$n \geq 0$$

So we can set $n_0 = 0$ and reverse the steps above.

Thus we can also conclude $n^2 + 100n = \theta(n^2)$

Common time complexities

BETTER

- $O(1)$ constant time
- $O(\log n)$ log time
- $O(n)$ linear time
- $O(n \log n)$ log linear time
- $O(n^2)$ quadratic time
- $O(n^3)$ cubic time
- $O(2^n)$ exponential time

WORSE

Better

Comparing the asymptotic running time

Ex: O(log n) is better than O(n)

$$\log n << n << n^2 << n^3 << 2^n$$

Rules

1. Eliminate low order terms

- $4n + 5 \Rightarrow 4n$
- $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
- $2^n + n^3 + 3n \Rightarrow 2^n$

2. Eliminate constant coefficients

- $4n \Rightarrow n$
- $0.5\ n \log n \Rightarrow n \log n$
- $n \log (n^2) = 2\ n \log n \Rightarrow n \log n$

Rules

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

1. $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$

$$O(n^3) + O(n^4) = \max(O(n^3), O(n^4)) = O(n^4)$$

2. $T_1(n) * T_2(n) = O(f(n) * g(n))$

$$O(n^3) * O(n^4) = O(n^3 * n^4) = O(n^7)$$

Analyzing Code

I want to do some code examples, but first, how will we examine code.

- primitive operations
- consecutive statements
- function calls
- conditionals
- loops
- recursive functions

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

- if you have two loops; the outer and inner loop, and they are dependent on the problem size n, **the statements in the inner loop will be executed $O(n^2)$ times:**

```
for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < n; j++ ) {
        // these statements are executed O(n^2) times
    }
}
```

```
for ( int i = 0; i < n / 2; i++ ) {
    for ( int j = 0; j < n / 3; j++ ) {
        // these statements are also executed O(n^2) times
        // since both loops loop O(n) times, and
        // O(n) * O(n) = O(n^2)
    }
}
```

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

- if you have triply-nested loops, all of which are dependent on the problem size n, the statements in **the innermost loop will be executed $O(n^3)$ times**:

```
for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < n; j++ ) {
        for ( int k = 0; k < n; k++ ) {
            // these statements are executed O(n^3) times
        }
    }
}
```

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

imagine a case with doubly-nested loops where only the outer loop is dependent on the problem size n, and the inner loop always executes a constant number of times, say 3 times:

```
for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < 3; j++ ) {
        // these statements are executed O(n) times
    }
}
```

In this particular case, the inner loop will execute exactly 3 times for each of the n iterations of the outer loop, and so the total number of times the statements in the innermost loop will be executed is 3n or O(n) times, *not* $O(n^2)$ times.

Tips on Analyzing the Time Complexity of Algorithms that use Nested Loops

imagine a third case: you have doubly nested loops, and the outer loop is dependent on the problem size n, but the inner loop is dependent on the current value of the index variable of the outer loop:

```
for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < i; j++ ) {
        // these statements are executed O(n^2) times
    }
}
```

Analyzing Code

Simple statement

The simple statement takes O(1) time.

```
1 | int x= n + 12;
```

if condition

```
if (condition) {
    sequence of statements 1
}
else {
    sequence of statements 2
}
```

the worst-case time is the slowest of the two possibilities: max(time(sequence 1), time(sequence 2)). For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be O(N).

for/while loops

The loops take N time to complete and take O(n).

```
1 | for(int i=0;i<n;i++)
2 | {
3 | ..
4 | ..
5 | }
```

Nested loops

If the nested loops contain M and N size, the cost is O(MN)

```
1 | for(int i=0;i<n;i++)
2 | {
3 |     for(int i=0;i<m;i++){
4 |     ..
5 |     ..
6 |     }
7 | }
```

Analyzing Code

Example 1

O(N)

```
1  for(int i = 0; i < n; i++)
2      sum++;
```

Example 2

O(N)

```
1  for(int i = 0; i < n; i+=2)
2     sum++;
```

Example 3

O(N^2)

```
1  for(int i = 0; i < n; i++)
2      for( int j = 0; j < n; j++)
3          sum++;
```

Analyzing Code

Example 4

O(N)

```
1  for(int i = 0; i < n; i+=2)
2          sum++;
3      for(int j = 0; j < n; j++)
4          sum++;
```

Example 5

O(n^3)

```
1  for(int i = 0; i < n; i++)
2      for( int j = 0; j < n * n; j++)
3          sum++;
```

Example 6

O(N^2)

```
1  for(int i = 0; i < n; i++)
2      for( int j = 0; j < i; j++)
3          sum++;
```

Analyzing Code

Example 7

O(n^5)

```
1  for(int i = 0; i < n; i++)
2      for( int j = 0; j < n * n; j++)
3          for(int k = 0; k < j; k++)
4              sum++;
```

Example 8

O(log(n))

```
1  for(int i = 1; i < n; i = i * 2)
2      sum++;
```

Example 9

log(n)

```
1  while(n>1){
2      n=n/2;
3  }
```

Analyzing Code

Example 10

Find O (n) for the following:

$7n - 3$

$8n^2 \log n + 5n^2 + n$

Simple Rule: Drop lower order terms and constant factors.

- $7n - 3$ is $O(n)$
- $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$

Analyzing Code

Example 11

```
for (int i=0;i< n*n ;i++)
   for (int j=i;j< i*i;j++)
      for (int k=0;k<n;k++){
         Statement(s);
}
```

// these statement are executed O (n^7) times

More Examples

On board

Recurrence Relation

A recurrence relation, T(n), is a recursive function of an integer variable n.

Like all recursive functions, it has one or more recursive cases and one or more base cases.

Example:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn + c & \text{if } n > 1 \end{cases}$$

The portion of the definition that does not contain T is called the **base case** of the recurrence relation; the portion that contains T is called the **recurrent or recursive case**.

Forming Recurrence Relations

Example 1: Write the recurrence relation for the following method:

```
public void f (int n) {
    if (n==0)
        System.out.println(n)
    else{
        System.out.println(n);
        f(n-1);
    }
}
```

Recursive Function to print number from Given input down to 0.

1. The base case is reached when n = = 0.

2. When n > 0, the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter n – 1.

Forming Recurrence Relations

Example 1: Write the recurrence relation for the following method:

```
public void f (int n) {
    if (n==0)
        System.out.println(n)
    else{
        System.out.println(n);
        f(n-1);
    }
}
```

$$T(n) = \begin{cases} c & , n = 0 \\ b + T(n-1) & , n > 0 \end{cases}$$

Forming Recurrence Relations

Example 2: Write the recurrence relation for the following method:
:

```
long fibonacci (int n) { // Recursively calculates Fibonacci number
  if( n == 1 || n == 2)
       return 1;
  else
       return fibonacci(n – 1) + fibonacci(n – 2);
}
```

the recurrence relation is:

$$T(n) = \begin{cases} c & , n = 1 \text{ or } n = 2 \\ b + T(n-1) + T(n-2) & , n > 2 \end{cases}$$

Forming Recurrence Relations

Example 3: Write the recurrence relation for the following method:

```
long power (long x, long n) {
   if(n == 0)
     return 1;
   else if(n == 1)
        return x;
   else if ((n % 2) == 0)
    return power (x, n/2) * power (x, n/2);
   else
    return x * power (x, n/2) * power (x, n/2);
}
```

The recurrence relation is:

$$T(n) = \begin{cases} c & , n = 0 \text{ or } n = 1 \\ b + 2T(n/2) & , n > 2 \end{cases}$$

Solving Recurrence Relations

Example1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

```
long factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial (n – 1);
}
```

First: The recurrence relation is:

$$T(n) = \begin{cases} c & , n = 0 \\ b + T(n-1) & , n >= 1 \end{cases}$$

Second: By Substitution

Solving Recurrence Relations

T (n) =b + T (n-1)

T(n-1) = b + T (n-1-1) = b + T (n-2)

→ T(n-1) =b + T (n-2)

T (n) = b + [b + T (n-2)] = 2b + T (n-2)

T (n) = 2b + T (n-2)

T(n-2) = b + T (n-2-1) = b + T (n-3)

→ T (n-2) =b + T (n-3)

T (n) = 2b + [b + T (n-3)]

T (n) =3b + T (n-3)

T (n) = b + T (n-1)

T (n) = 2b + T (n-2)

T (n) = 3b + T (n-3)

.

.

T (n) =kb + T (n-k)

The base case is reached when n – k = 0
→ k = n, we then have:

T (n) = nb + T (n-n) = nb + T (0)

T (n) = nb +T (0)

T (n) = nb + c , c is constant

Therefore the method factorial is O(n)

Solving Recurrence Relations

Example 2: Analysis The following recurrence relation : (determine its big-O complexity)

$$T(n) = \begin{cases} c & , n = 1 \\ 2T(n/2) + n & , n > 1 \end{cases}$$

Solution: By Substitution

Solving Recurrence Relations

$T(n) = 2T(n/2) + n$

$T(n/2) = 2\,T(n/4) + n/2$

$T(n) = 2\,[\,2\,T(n/4) + n/2\,] + n = 4\,T(n/4) + 2n$

$T(n) = 4\,T(n/4) + 2n$

$T(n/4) = 2\,T(n/8) + n/4$

$T(n) = 4\,[\,2\,T(n/8) + n/4\,] + 2n = 8\,T(n/8) + 3n$

$T(n) = 8\,T(n/8) + 3n$

.

.

$T(n) = 2^k\,T(n/2^k) + k\,n$

$T(n) = 2^k\,T(n/2^k) + k\,n$

The base case is reached when $n/2^k = 1 \rightarrow n = 2^k$, we then have:

$T(n) = n\,T(1) + n \log_2 n$

$T(n) = nc + n \log_2 n$, c is constant

Therefore $T(n) = O(n \log_2 n)$

Definition: $\log_X B = A$ means $X^A = B$

Solving Recurrence Relations

Example 3: Analysis The following recurrence relation : (determine its big-O complexity)

$$T(n) = \begin{cases} d & , n=1 \\ 2\,T\left(\frac{n}{2}\right) + b & , n>1 \end{cases}$$

Solution: By Substitution

Solving Recurrence Relations

$T(n) = 2\,T(\frac{n}{2}) + b$

$T(\frac{n}{2}) = 2\,T(\frac{n}{2^2}) + b$

$T(n) = 2\,[\,2T(\frac{n}{2^2}) + b\,] + b$

$T(n) = 2^2\,T(\frac{n}{2^2}) + (2*b) + b$

$T(\frac{n}{2^2}) = 2\,T(\frac{n}{2^3}) + b$

$T(n) = 2^2\,[2\,T(\frac{n}{2^3}) + b] + (2*b) + b$

$T(n) = 2^3\,T(\frac{n}{2^3}) + (2^2 * b) + (2*b) + b$

$T(n) = 2^k\,T(\frac{n}{2^k}) + (2^{k-1} * b) + (2^{k-2} * b) + \ldots + 2^0 * b$

$T(n) = 2^k\,T(\frac{n}{2^k}) + b\sum_{i=0}^{k-1} 2^i$

$T(n) = 2^k\,T(\frac{n}{2^k}) + b\left[\frac{2^k-1}{2-1}\right]$

Base case : $T(d)=1 \rightarrow \frac{n}{2^k} = 1 \rightarrow n=2^k \rightarrow K=\log_2 n$

$T(n) = 2^k\,T(\frac{n}{2^k}) + b\left[\frac{2^k-1}{2-1}\right]$

$T(n) = n\,T(1) + b\left[\frac{n-1}{1}\right]$

$\rightarrow T(n) = n\,c + bn - b \;,\; c, b \text{ constants}$

Therefore T(n) = O(n)

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} \quad (x \neq 1)$$

Definition: $\log_x B = A$ means $x^A = B$