CHAPTER II

Data Structures and Algorithm Analysis in Java 3rd Edition by Mark Allen Weiss

Big-Oh and Other Notations in Algorithm Analysis

- Classifying Functions by Their Asymptotic Growth
- Theta, Little oh, Little omega
- Big Oh, Big Omega
- Rules to manipulate Big-Oh expressions
- Typical Growth Rates

Classifying Functions by Their Asymptotic Growth

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

.growing faster

.growing slower

Theta

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

$\lim(f(n) / g(n)) = c,$
$0 < c < \infty, \quad n \to \infty$

Notation: $f(n) = \Theta(g(n))$
pronounced "theta"

Little oh

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

$\lim(f(n) / g(n)) = 0, \quad n \to \infty$

Notation: f(n) = o(g(n))
pronounced "little oh"

Little omega

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

$\lim(f(n) / g(n)) = \infty, \quad n \to \infty$

Notation: $f(n) = \omega(g(n))$
pronounced "little omega"

Little omega and Little oh

if $g(n) = \mathbf{o}(f(n))$

then $f(n) = \boldsymbol{\omega}(g(n))$

Examples: Compare $\boldsymbol{n}$ and $\boldsymbol{n^2}$

$\lim(n/n^2) = 0,\ n \to \infty,\ n = o(n^2)$

$\lim(n^2/n) = \infty,\ n \to \infty,\ n^2 = \omega(n)$

Theta: Relation of Equivalence

R: "having the same rate of growth":
relation of equivalence,
gives a partition over the set of all differentiable functions - classes of equivalence.

Functions in one and the same class are equivalent with respect to their growth.

Algorithms with Same Complexity

Two algorithms have same complexity, if the functions representing the number of operations have same rate of growth.

Among all functions with same rate of growth we choose the simplest one to represent the complexity.

Examples

Compare ***n*** and ***(n+1)/2***

$\lim(n / ((n+1)/2)) = 2$,
same rate of growth

$(n+1)/2 = \mathbf{\Theta(n)}$
- rate of growth of a linear function

Examples

Compare n^2 and $n^2 + 6n$

$\lim(n^2 / (n^2 + 6n)) = 1$

same rate of growth.

$n^2+6n = \Theta(n^2)$

rate of growth of a quadratic function

Examples

Compare **log n** and **log n^2**

lim(log n / log n^2) = 1/2

same rate of growth.

log n^2 = **Θ(log n)**

logarithmic rate of growth

Examples

$\Theta(n^3)$: n^3

$5n^3 + 4n$

$105n^3 + 4n^2 \neq 6n$

$\Theta(n^2)$: n^2

$5n^2 + 4n + 6$

$n^2 + 5$

$\Theta(\log n)$: $\log n$

$\log n^2$

$\log (n + n^3)$

Comparing Functions

- same rate of growth: **$g(n) = \Theta(f(n))$**
- different rate of growth:

  either **$g(n) = o(f(n))$**

  g(n) grows slower than f(n),

  and hence **$f(n) = \omega(g(n))$**

  or **$g(n) = \omega(f(n))$**

  g(n) grows faster than f(n),

  and hence **$f(n) = o(g(n))$**

The Big-Oh Notation

f(n) = O(g(n))

if f(n) grows with

same rate or slower than g(n).

f(n) = Θ(g(n)) or
f(n) = o(g(n))

Example

$$n+5 = \Theta(n) = O(n) = O(n^2)$$
$$= O(n^3) = O(n^5)$$

the closest estimation: $n+5 = \Theta(n)$

the general practice is to use the Big-Oh notation:

$n+5 = O(n)$

The Big-Omega Notation

The inverse of Big-Oh is **Ω**

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

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

Rules to manipulate Big-Oh expressions

Rule 1:

a. If

$$T1(N) = O(f(N))$$

and

$$T2(N) = O(g(N))$$

then

$T1(N) + T2(N) =$
$\max(O(f(N)), O(g(N)))$

Rules to manipulate Big-Oh expressions

b. If

$$T1(N) = O(f(N))$$

and

$$T2(N) = O(g(N))$$

then

$T1(N) * T2(N) = O(f(N)* g(N))$

Rules to manipulate Big-Oh expressions

Rule 2:

If T(N) is a polynomial of degree **k**, then

$$\mathbf{T(N) = \Theta(N^k)}$$

Rule 3:

$\mathbf{\log^k N = O(N)}$ for any constant **k**.

Examples

$n^2 + n = O(n^2)$
we disregard any lower-order term

$nlog(n) = O(nlog(n))$

$n^2 + nlog(n) = O(n^2)$

Typical Growth Rates

C	constant, we write O(1)
logN	logarithmic
$\log^2 N$	log-squared
N	linear
NlogN	
N^2	quadratic
N^3	cubic
2^N	exponential
N!	factorial

Problems

$N^2 = O(N^2)$ **true**

$2N = O(N^2)$ **true**

$N = O(N^2)$ **true**

$N^2 = O(N)$ **false**

$2N = O(N)$ **true**

$N = O(N)$ **true**

Problems

$N^2 = \Theta(N^2)$	true	
$2N = \Theta(N^2)$		false
$N = \Theta(N^2)$		false
$N^2 = \Theta(N)$		false
$2N = \Theta(N)$	true	
$N = \Theta(N)$	true	

What to Analyze

- Running Time
  - Algorithm
  - Input

- Two Functions can be calculated on an algorithm on input of size N:
  - $T_{avg}(N)$
  - $T_{worst}(N)$

Maximum Subsequent Sum Problem

- Given integers A_1, A_2,…., A_N, find the maximum value of $\sum_{k=i}^{j} A_k$

- Ex: For input -2, 11, -4, 13,-5, -2 is 20

Input Size	Algorithm Time			
	1	2	3	4
	$O(N^3)$	$O(N^2)$	$O(N \log N)$	$O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Figure 2.2 Running times of several algorithms for maximum subsequence sum (in seconds)

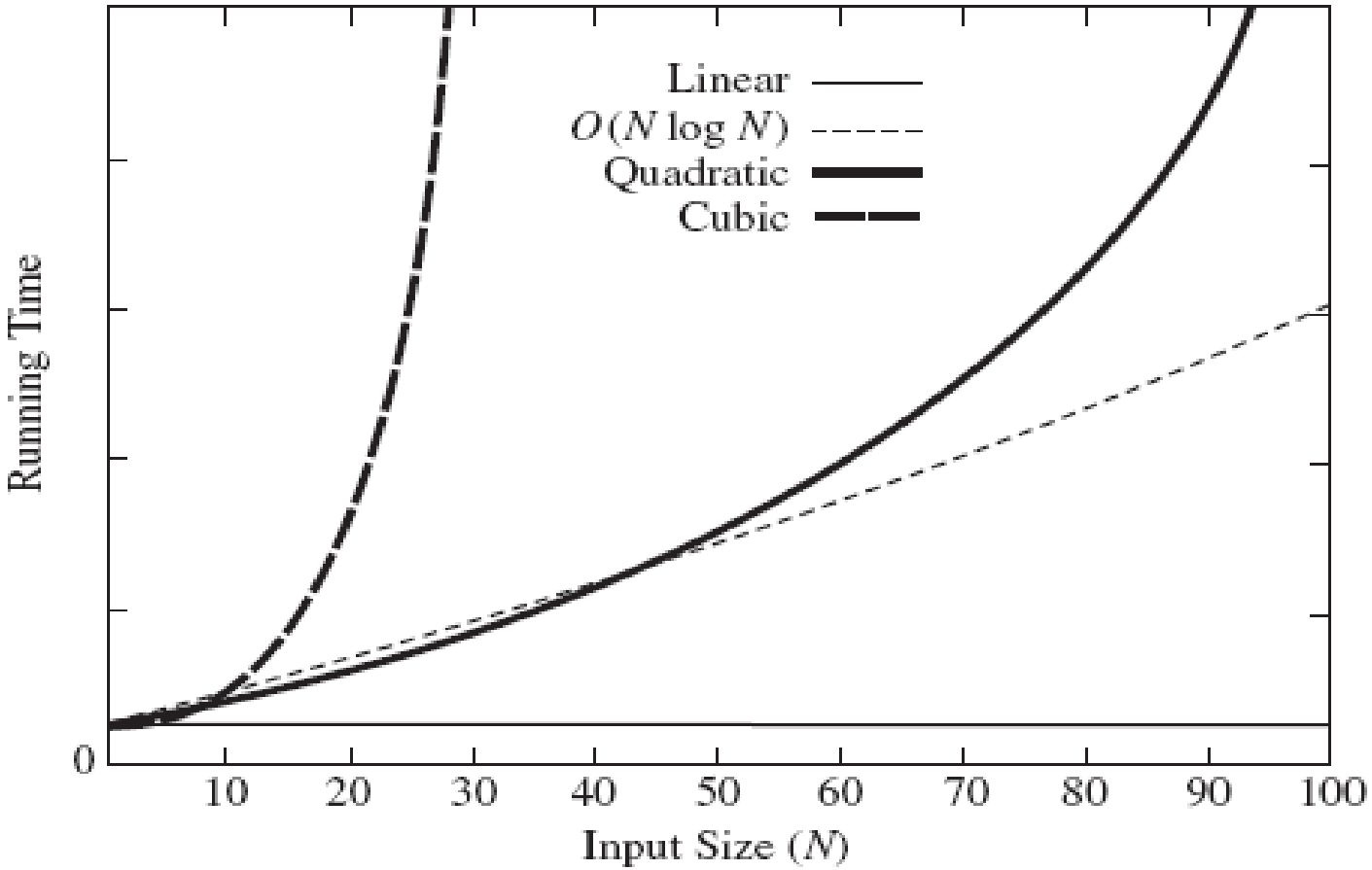

Figure 2.3 Plot (N vs. time) of various algorithms

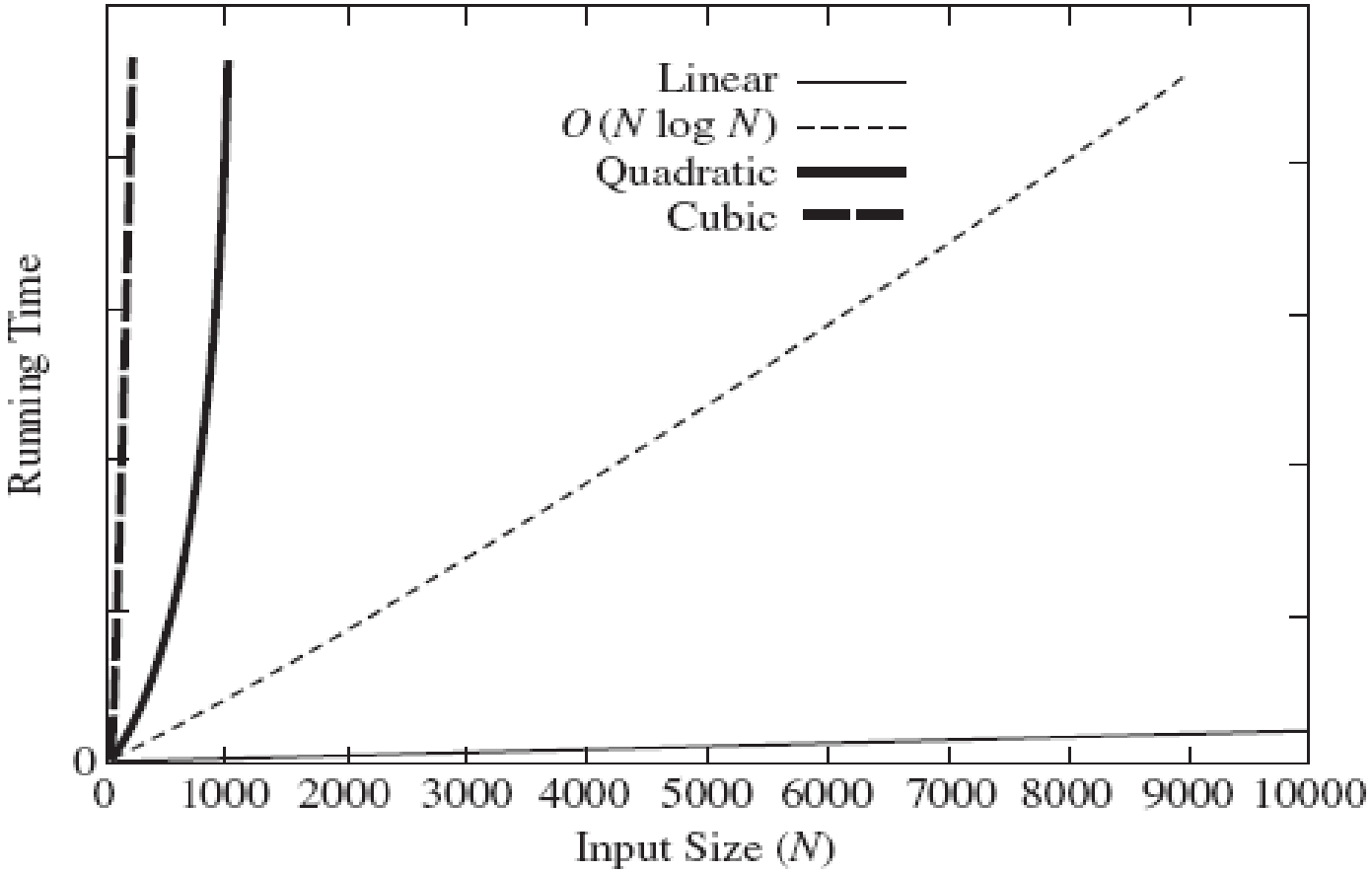

Figure 2.4 Plot (N vs. time) of various algorithms

Running Time Calculation

- Running time analysis can be done by calculating Big Oh running time.

- Big Oh guarantees that the program will terminate within a certain time period.

Example

- Sample Program to calculate

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

```
public static int sum( int n) {
    int partialSum;

    partialSum = 0;
    for(int i =1; i <=n ; i++)
        partialSum += i * i * i  ;
    return partialSum;
    }
```

General Rules

- Rule 1 : *for* loops
- Rule 2: Nested loops
- Rule 3: Consecutive Statements
- Rule 4- *if/else*

Rule 1: for loops

- Running time of a *for* loop =

  running time of statements inside the loop

  *

  number of iterations

Rule 2: Nested Loops

- Analyze inside out
- Example

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

The program fragment is $O(N^2)$

Rule 3: Consecutive Statements

- Add them

```
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;
```

O(N) work followed by $O(N^2)$ work, so use the maximum

O(N) Calculation

- Let T1(N) and T2(N) be the number of operations necessary to run two program segments respectively, and let T1 = O(g1(N)), T2 = O(g2(N)).

  $T1(N) + T2(N) = \max(O(g1(N)), O(g2(N)))$

- where max(O(g1(N)), O(g2(N))) is determined in the following way:

  If g1(N) = O(g2(N)), then max(O(g1(N)), O(g2(N))) = O(g2(N)) If g2(N) = O(g1(N)), then max(O(g1(N)), O(g2(N))) = O(g1(N))

  Example: $\max(O(n^2),O(n^3)) = O(n^3)$.

Rule 4 : *if/else*

For Example

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

Running time = running time of the test + max running time of S1&S2

Maximum Subsequence Sum Problem

```
1       /**
2        * Cubic maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum1( int [ ] a )
5       {
6           int maxSum = 0;
7
8           for( int i = 0; i < a.length; i++ )
9               for( int j = i; j < a.length; j++ )
10              {
11                  int thisSum = 0;
12
13                  for( int k = i; k <= j; k++ )
14                      thisSum += a[ k ];
15
16                  if( thisSum > maxSum )
17                      maxSum = thisSum;
18              }
19
20          return maxSum;
21      }
```

Figure 2.5 Algorithm 1

Analysis

- Each for loop executes to maximum of N times

- By rule of nested loops we multiple all the loop running times
- Running time of the algorithm = $O(N^3)$

Maximum Subsequence Sum Problem

```
1       /**
2        * Quadratic maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum2( int [ ] a )
5       {
6           int maxSum = 0;
7
8           for( int i = 0; i < a.length; i++ )
9           {
10              int thisSum = 0;
11              for( int j = i; j < a.length; j++ )
12              {
13                  thisSum += a[ j ];
14
15                  if( thisSum > maxSum )
16                      maxSum = thisSum;
17              }
18          }
19
20          return maxSum;
21      }
```

Figure 2.6 Algorithm 2

Analysis

- What happens if the inner loop does not start from 0?

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

- Here, the number of the times the inner loop is executed depends on the value of **i**:

  i = 0, inner loop runs n times

  i = 1, inner loop runs (n-1) times

  i = 2, inner loop runs (n-2) times ...

  i = n - 2, inner loop runs 2 times

  i = n – 1,inner loop runs 1 (once)

- Adding the right column, we get: (1 + 2 + … + n) = n*(n+1)/2 = **$O(n^2)$**

```
1         /**
2          * Recursive maximum contiguous subsequence sum algorithm.
3          * Finds maximum sum in subarray spanning a[left..right].
4          * Does not attempt to maintain actual best sequence.
5          */
6         private static int maxSumRec( int [ ] a, int left, int right )
7         {
8             if( left == right )  // Base case
9                 if( a[ left ] > 0 )
10                    return a[ left ];
11                else
12                    return 0;
13
14            int center = ( left + right ) / 2;
15            int maxLeftSum  = maxSumRec( a, left, center );
16            int maxRightSum = maxSumRec( a, center + 1, right );
17
18            int maxLeftBorderSum = 0, leftBorderSum = 0;
19            for( int i = center; i >= left; i-- )
20            {
21                leftBorderSum += a[ i ];
22                if( leftBorderSum > maxLeftBorderSum )
23                    maxLeftBorderSum = leftBorderSum;
24            }
25
26            int maxRightBorderSum = 0, rightBorderSum = 0;
27            for( int i = center + 1; i <= right; i++ )
28            {
29                rightBorderSum += a[ i ];
30                if( rightBorderSum > maxRightBorderSum )
31                    maxRightBorderSum = rightBorderSum;
32            }
33
34            return max3( maxLeftSum, maxRightSum,
35                         maxLeftBorderSum + maxRightBorderSum );
36        }
37
38        /**
39         * Driver for divide-and-conquer maximum contiguous
40         * subsequence sum algorithm.
41         */
42        public static int maxSubSum3( int [ ] a )
43        {
44            return maxSumRec( a, 0, a.length - 1 );
45        }
```

Figure 2.7 Algorithm 3

Recursive Algorithm

4 -3 5 -2 -1 2 6 -2

Maximum Subsequent sum for first half = 6

Maximum Subsequent sum for second half = 8

Max sum in first half that includes the last element = 4

Max sum in second half that includes the first element = 7

Max sum of the sequence = max(6,8,(4+7))

Recursive Algorithm Analysis

- The two *for* loops take O(N) time
- Let T(N) be time to solve the problem for N inputs
- When N=1 T(N) = 1
- Recursive call always divide the problem taking T(N/2) each
- T(N) = 2 * T(N/2) + O(N)

Cont..

- $T(N) = 2T(N/2) + N$
- $T(1) = 1$
- $T(2) = 4 = 2*2$
- $T(4) = 12 = 4*3$
- $T(8) = 32 = 8*4$
- $T(16) = 80 = 16*5$
- $T(N) = N*(k+1)$ where $N= 2^k$
- $T(N) = N\log N + N = O(N\log N)$

Maximum Subsequence Sum Problem

```
1       /**
2        * Linear-time maximum contiguous subsequence sum algorithm.
3        */
4       public static int maxSubSum4( int [ ] a )
5       {
6           int maxSum = 0, thisSum = 0;
7
8           for( int j = 0; j < a.length; j++ )
9           {
10              thisSum += a[ j ];
11
12              if( thisSum > maxSum )
13                  maxSum = thisSum;
14              else if( thisSum < 0 )
15                  thisSum = 0;
16          }
17
18          return maxSum;
19      }
```

Figure 2.8 Algorithm 4

Analysis

- Running time is O(N)
- This algorithm is called online algorithm:
  - Once data is read it does not need to be remembered
  - At any point in time, can correctly give an answer for the data it has already read

Logarithms in Running time

- An algorithm is O(logN) if it takes constant time O(1) to cut the problem size by a fractions (usually ½)
  - Binary Search
  - Euclid's Algorithm
  - Exponentiation.

Binary Search

- Given an integer X and integers A_0, A_1,… A_{N-1}, which is presorted find i such that A_i = X. or i= -1 if X is not found.
- Check if X is the middle element if so, return i
- else if X is smaller than the middle element , check X in the left half
- else check if X is in the right half

```
1       /**
2        * Performs the standard binary search.
3        * @return index where item is found, or -1 if not found.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       int binarySearch( AnyType [ ] a, AnyType x )
7       {
8           int low = 0, high = a.length - 1;
9
10          while( low <= high )
11          {
12              int mid = ( low + high ) / 2;
13
14              if( a[ mid ].compareTo( x ) < 0 )
15                  low = mid + 1;
16              else if( a[ mid ].compareTo( x ) > 0 )
17                  high = mid - 1;
18              else
19                  return mid;   // Found
20          }
21          return NOT_FOUND;     // NOT_FOUND is defined as -1
22      }
```

Figure 2.9 Binary search

Euclid's Algorithm

- For computing the greatest common divisor (gcd).
- The gcd of two integer is the largest integer that divides both.
- For example gcd (1989,1590) = 3

```
1       public static long gcd( long m, long n )
2       {
3           while( n != 0 )
4           {
5               long rem = m % n;
6               m = n;
7               n = rem;
8           }
9           return m;
10      }
```

Figure 2.10 Euclid's algorithm

Exponentiation

- To compute X^N we need N-1 multiplication
- Recursive algorithm is used to reduce the running time to 2logN
- For example to calculate X^{62}

$X^{62} = (X^{31})^2$

$X^{31} = (X^{15})^2 X$

$X^{15} = (X^7)^2 X$

$X^7 = (X^3)^2 X$

$X^3 = (X^2)X$

```
1       public static long pow( long x, int n )
2       {
3           if( n == 0 )
4               return 1;
5           if( n == 1 )
6               return x;
7           if( isEven( n ) )
8               return pow( x * x, n / 2 );
9           else
10              return pow( x * x, n / 2 ) * x;
11      }
```

Figure 2.11 Efficient exponentiation