Complexity Analysis (Part II)

- **Asymptotic Complexity**
- ***Big-O (asymptotic) Notation***
- **Big-O Computation Rules**
- **Proving Big-O Complexity**
- **How to determine complexity of code structures**

Asymptotic Complexity

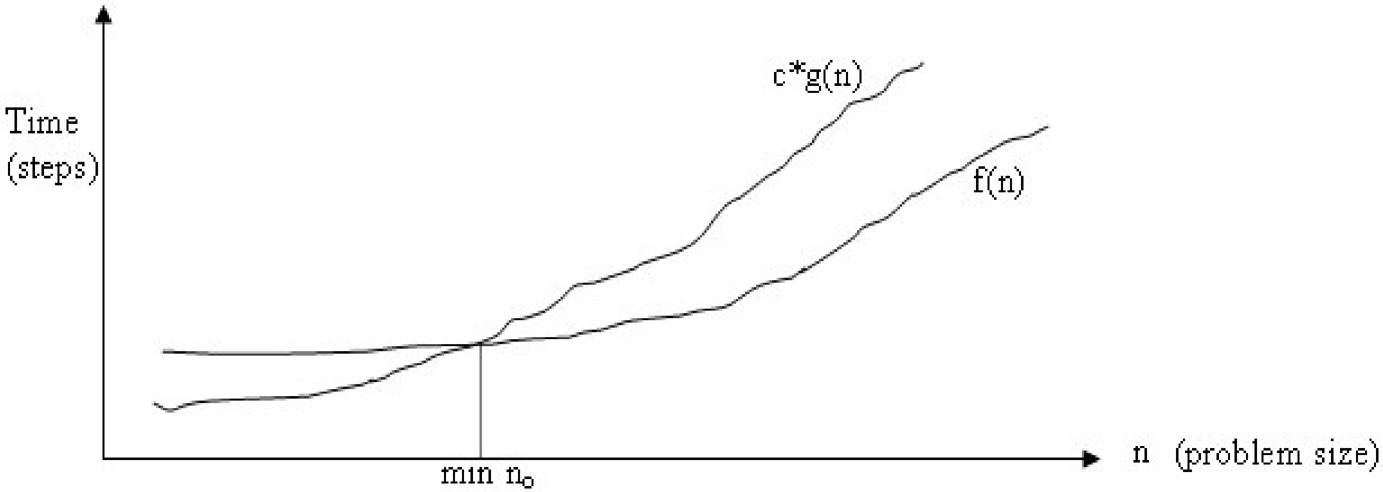
- Finding the exact complexity, **f(n) = number of basic operations**, of an algorithm is difficult.
- We approximate **f(n)** by a function **g(n)** in a way that does not substantially change the magnitude of **f(n)**. -- the function **g(n)** is sufficiently close to **f(n)** for <u>large values</u> of the input size **n**.
- This "approximate" measure of efficiency is called **asymptotic complexity**.
- Thus the ***asymptotic complexity*** measure does not give the exact number of operations of an algorithm, but it shows how that number grows with the size of the input.
- This gives us a measure that will work for different operating systems, compilers and CPUs.

Big-O (asymptotic) Notation

- The most commonly used notation for specifying asymptotic complexity is the big-O notation.
- The Big-O notation, **O(g(n)),** is used to give an upper bound (worst-case) on a positive runtime function f(n) where n is the input size.

Definition of Big-O:

- Consider a function **f(n)** that is non-negative $\forall$ n $\geq$ 0. We say that "f(n) is Big-O of g(n)" i.e., f(n) = O(g(n)), if $\exists$ n0 $\geq$ 0 and a constant c > 0 such that **f(n)** $\leq$ **cg(n),** $\forall$ **n** $\geq$ **n0**



Big-O (asymptotic) Notation

Implication of the definition:

- For all <u>sufficiently large</u> n, c *g(n) is an upper bound of f(n)

  Note: By the definition of Big-O:

  $f(n) = 3n + 4$ is $O(n)$

  it is also $O(n^2)$,

  it is also $O(n^3)$,

  . . .

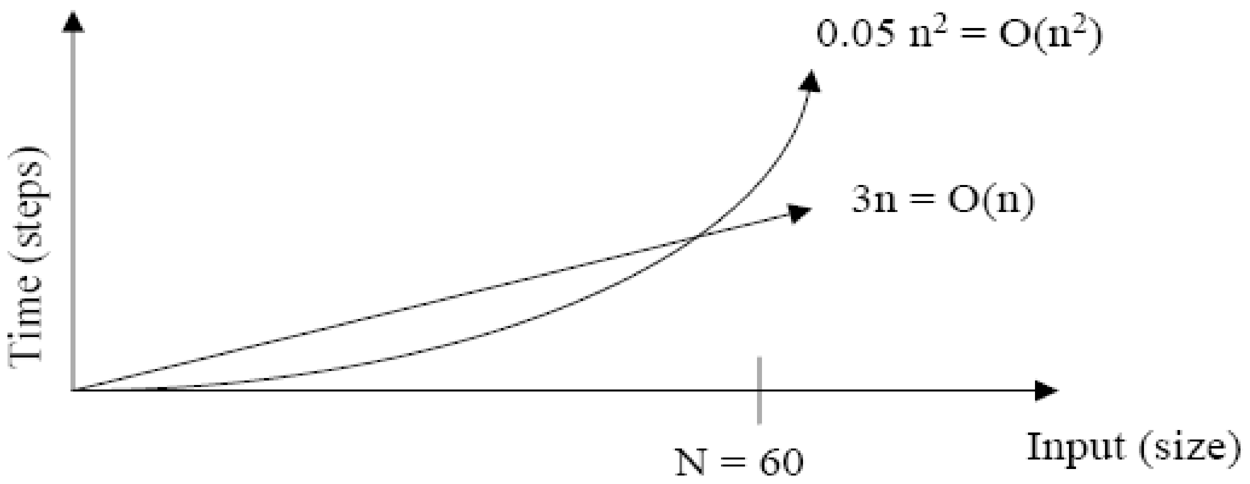
  it is also $O(n^n)$

- However when Big-O notation is used, the function g in the relationship f(n) is O(g(n)) is CHOOSEN TO BE AS SMALL AS POSSIBLE.
  - We call such a function g a *tight asymptotic bound* of f(n)

Big-O (asymptotic) Notation

Some Big-O complexity classes in order of magnitude from smallest to highest:

O(1)	Constant
O(log(n))	Logarithmic
O(n)	Linear
O(n log(n))	n log n
$O(n^x)$ {e.g., $O(n^2)$, $O(n^3)$, etc}	Polynomial
$O(a^n)$ {e.g., $O(1.6^n)$, $O(2^n)$, etc}	Exponential
O(n!)	Factorial
$O(n^n)$	



Examples of Algorithms and their big-O complexity

Big-O Notation	Examples of Algorithms
$O(1)$	Push, Pop, Enqueue (if there is a tail reference), Dequeue, Accessing an array element
$O(\log(n))$	Binary search
$O(n)$	Linear search
$O(n \log(n))$	Heap sort, Quick sort (average), Merge sort
$O(n^2)$	Selection sort, Insertion sort, Bubble sort
$O(n^3)$	Matrix multiplication
$O(2^n)$	Towers of Hanoi

Warnings about O-Notation

- Big-O notation cannot compare algorithms in the same complexity class.
- Big-O notation only gives sensible comparisons of algorithms in different complexity classes when **n** is large .
- Consider two algorithms for same task:
  Linear: $\mathbf{f(n) = 1000\ n}$
  Quadratic: $\mathbf{f'(n) = n^2/1000}$
  The quadratic one is faster for n < 1000000.

Rules for using big-O

- For large values of input n , the constants and terms with lower degree of n are ignored.

1. **Multiplicative Constants Rule**: Ignoring constant factors.
   $O(c\ f(n)) = O(f(n))$, where c is a constant;
   Example:
   $O(20\ n^3) = O(n^3)$

2. **Addition Rule:** Ignoring smaller terms.
   If $O(f(n)) < O(h(n))$ then $O(f(n) + h(n)) = O(h(n))$.
   Example:
   $O(n^2 \log n + n^3) = O(n^3)$
   $O(2000\ n^3 + 2n\ ! + n^{800} + 10n + 27n \log n + 5) = O(n\ !)$

3. **Multiplication Rule:** $O(f(n) * h(n)) = O(f(n)) * O(h(n))$
   Example:
   $O((n^3 + 2n^2 + 3n \log n + 7)(8n^2 + 5n + 2)) = O(n^5)$

Proving Big-O Complexity

To prove that **f(n)** is **O(g(n))** we find any pair of values $\mathbf{n_0}$ and **c** that satisfy:

$$\mathbf{f(n) \leq c * g(n)} \quad \text{for} \quad \forall\ \mathbf{n} \geq \mathbf{n_0}$$

Note: The pair **(n0, c)** is not unique. If such a pair exists then there is an **infinite** number of such pairs.

Example: Prove that $\mathbf{f(n) = 3n^2 + 5}$ is $\mathbf{O(n^2)}$

We try to find some values of n and c by solving the following inequality:

$$\mathbf{3n^2 + 5 \leq cn^2} \qquad \textbf{OR} \qquad \mathbf{3 + 5/n^2 \leq c}$$

(By putting different values for n, we get corresponding values for c)

$\mathbf{n_0}$	1	2	3	4	∞
c	8	4.25	3.55	3.3125	3

Proving Big-O Complexity

Example:

Prove that $\mathbf{f(n) = 3n^2 + 4n \log n + 10}$ is $\mathbf{O(n^2)}$ by finding appropriate values for **c** and $\mathbf{n_0}$

We try to find some values of n and c by solving the following inequality

$$\mathbf{3n^2 + 4n \log n + 10 \leq cn^2}$$

OR $$\mathbf{3 + 4 \log n / n + 10/n^2 \leq c}$$

(We used Log of base 2, but another base can be used as well)

$\mathbf{n_0}$	1	2	3	4	∞
c	13	7.5	6.22	5.62	3

How to determine complexity of code structures

Loops: for, while, and do-while:

Complexity is determined by the number of iterations in the loop times the complexity of the body of the loop.

Examples:

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

O(n)

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

O(n^2)

```
i=1;
while (i < n) {
      sum = sum + i;
      i = i*2
}
```

O(log n)

How to determine complexity of code structures

Nested Loops: Complexity of inner loop * complexity of outer loop.

Examples:

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

$O(n^2)$

```
i = 1;
while(i <= n) {
   j = 1;
   while(j <= n){
      statements of constant complexity
      j = j*2;
   }
   i = i+1;
}
```

O(n log n)

How to determine complexity of code structures

Sequence of statements: Use Addition rule

O(s1; s2; s3; … sk) = O(s1) + O(s2) + O(s3) + … + O(sk)

= O(max(s1, s2, s3, . . . , sk))

Example:

```
for (int j = 0; j < n * n; j++)
   sum = sum + j;
for (int k = 0; k < n; k++)
   sum = sum - l;
System.out.print("sum is now ” + sum);
```

Complexity is $O(n^2) + O(n) + O(1) = O(n^2)$

How to determine complexity of code structures

Switch: Take the complexity of the most expensive case

```
char key;
int[] X = new int[5];
int[][] Y = new int[10][10];
........
switch(key)  {
    case 'a':
       for(int i = 0; i < X.length; i++)
         sum += X[i];
    break;
    case 'b':
       for(int i = 0; i < Y.length; j++)
         for(int j = 0; j < Y[0].length; j++)
              sum += Y[i][j];
    break;
  }  // End of switch block
```

$o(n)$

$o(n^2)$

Overall Complexity: $o(n^2)$

How to determine complexity of code structures

If Statement: Take the complexity of the most expensive case **:**

```
char key;
   int[][] A = new int[5][5];
   int[][] B = new int[5][5];
   int[][] C = new int[5][5];
   ........
   if(key == '+')  {
      for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
           C[i][j] = A[i][j] + B[i][j];
   } // End of if block

   else if(key == 'x')
      C = matrixMult(A, B);

   else
      System.out.println("Error! Enter '+' or 'x'!");
```

O(n²)

O(n3)

O(1)

Overall complexity **O(n3)**

How to determine complexity of code structures

- Sometimes if-else statements must carefully be checked:

  O(if-else) = O(Condition)+ Max[O(if), O(else)]

```
int[] integers = new int[10];
........
if(hasPrimes(integers) == true)
     integers[0] = 20;
else
     integers[0] = -20;


public boolean hasPrimes(int[] arr) {
     for(int i = 0; i < arr.length; i++)
               ..........
               ..........
} // End of hasPrimes()
```

O(1)

O(1)

O(n)

O(if-else) = O(Condition) = ***O(n)***

How to determine complexity of code structures

- **Note:** Sometimes a loop may cause the if-else rule not to be applicable. Consider the following loop:

```
while (n > 0) {
   if (n % 2 = = 0) {
      System.out.println(n);
      n = n / 2;
   } else{
      System.out.println(n);
      System.out.println(n);
      n = n – 1;
    }
 }
```

The else-branch has more basic operations; therefore one may conclude that the loop is O(n). However the if-branch dominates. For example if n is 60, then the sequence of n is: 60, 30, 15, 14, 7, 6, 3, 2, 1, and 0. Hence the loop is logarithmic and its complexity is O(log n)