CS 2700 Programing and Data Structures.

Slot C (Mon 10.00am, Tues 9.00am, Wed 8.00am, Fri 12.00pm)

Instructor: Meghana Nasre (meghana@cse.iitm.ac.in)

Week 2: Complexity (Running time of Programs)

Tools for Two Aspects

1. Correctness
2. Complexity

Program / Algorithm Efficiency

Suppose we have **more than one algorithms / programs** for the same problem. Which one do you select? How?

- Are both of them correct? (correctness takes higher priority over other factors)
- Which one is **faster**?
  - Compare runs on a large set of inputs.
  - On the machine that you intend to run the program.
  - Compare running time in seconds / mins / hours.

Advantages:

You can estimate the maximum absolute time your program will need provided …

Disadvantages:

- Analysis is too tied up with the machine / hardware.
- How do other programs affect your program?
- Your inputs may not be representative.

An algorithm is a finite solution to infinitely many problems.

Lets take an example..

Compute gcd of two non-negative integers x and y.

```
gcd = 1; k = 1;
while (k <= x) {
 if ((x%k == 0) && (y%k == 0)) {
        gcd = k;
   }
   k++;
}
```

Idea1:

- Pick the smaller of the two, say x.
- Start checking for k ranging from 1 to x
- If k divides both x and y, then k is a candidate gcd.

Example continued..

- Compute gcd of two non-negative integers x and y where x >= y.

Idea2: (by Euclid)

- If y divides x, we are done.
- Else we have a smaller problem to solve.
  - gcd (x, y) = gcd (x % y, y)

Needs a proof!

```
if (y == 0) gcd = x;
while (x%y != 0) {
     x = x % y;
     if (x < y) {
        swap (x, y);
     }
}
```

Learning from the example..

- Implementing the algorithms in this case was easy enough – in general this may not be true.
- The running time varies across runs of the same program for the same set of inputs (need to take averages over a large number of runs!)
- The difference in the runtimes of the two algorithms is visible on "certain special" inputs. How does one find these?

- Can we avoid these altogether by doing some analysis without implemention?

Example 2 : Fibonacci numbers.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

n-th Fibonacci number is obtained from the (n-1)-th and (n-2)-th Fibonacci numbers.

fib(n) = fib(n-1) + fib(n-2)

```
int fib(n) {
if (n == 0 || n == 1)
    return n;
else
   return fib(n-1)+fib(n-2);
}
```

Is there a different way to write this program?

Learning from the example 2

- The same algorithm implemented in two different ways can lead to a large difference in the run times.
- Is recursion the issue? No! Euclids idea implemented recursively will still be faster than Idea1.

- We need some (mathematical) tools to analyze the running time of these programs / algorithms without relying on the implementation.

Recap from last class..

gcd

- Two different ideas
- One significantly faster than the other.
- Need to analyze the running times theoretically.

Fibonacci

- The same idea implemented in two different ways
- Recursive one may not even terminate successfully for large inputs.
- Needs analysis.

Study these snippets

```
x = x+y;
y = x-y;
x = x-y;
```

Proportional to constant

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

Proportional to n

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

Proportional to n^2

We would like to distinguish between the running times of these codes

Some more snippets..

```
x = x+y;
y = x-y;
x = x-y;
```

```
if (x<y)
    if (x<z) min =x;
    else min = z;
else
    if (y<z) min = y;
    else min = z;
```

```
for (i=0; i<100; i++)
    A[i] = 0;
```

All the codes are equally efficient. They all take constant time – the constants are different. We denote them O(1)

Big "Oh" notation

$T(n) = O(1)$ IF THERE EXISTS POSITIVE CONSTANTS c AND n_0 s.t

$T(n) \leq c$ FOR ALL $n \geq n_0$

$T(n) = O(n)$ IF THERE EXISTS POSITIVE CONSTANTS c and n_0 st

$T(n) \leq cn$ FORALL $n \geq n_0$

Big "Oh" notation

f(n) = O(g(n)) if there exists positive constants c and n_0 such that

$$f(n) \leq c\, g(n) \quad \forall\, n \geq n_0$$

Allows us to establish a relative order amongst the functions.

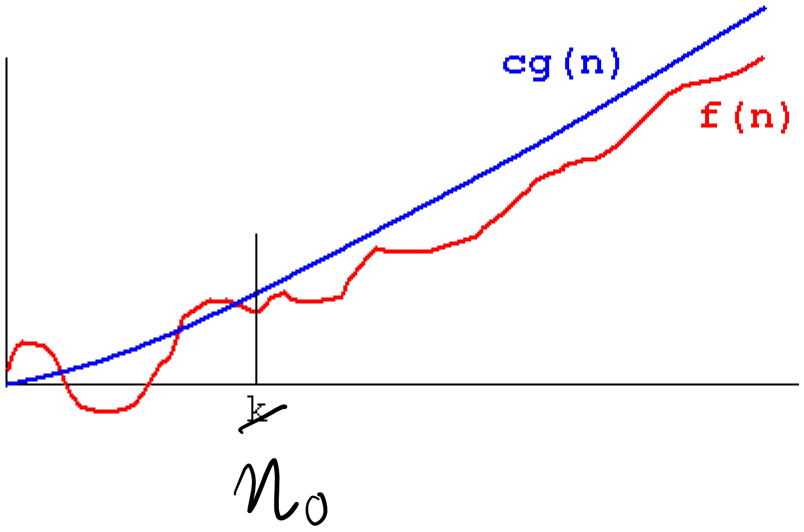
- $1000\, n > n^2$ for small values of n
- Yet we say $1000\, n = O(n^2)$ since we can select
  - $c = 1\ and\ n_0 = 1000$
  - $c = 10\ and\ n_0 = 100$

Big "Oh" notation

S1: $1000n = O(n^3)$

S2: $1000n = O(n^2)$

S3: $1000n = O(n)$



All these statements are true. f(n) = O(g(n)) means that f(n) grows at a rate no faster than g(n).

g(n) is an upper bound on f(n).

Back to these snippets

```
x = x+y;
y = x-y;
x = x-y;
```

O(1)

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

O(n)

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

O(n^2)

We would like to distinguish between the running times of these codes

One more example..

```
int fun(int n) {
    if (n==0) return 1;
    else return fun(n/3);
}
```

$T(n) = T(n/3) + c_1$

$T(1) = c_2$

c_1 AND c_2 ARE CONSTANTS.

$T(n) = T(n/3) + c_1$

$T(n) = T(n/9) + c_1 + c_1$

$= T(n/3^2) + 2 \cdot c_1$

$T(n) = T(n/3^k) + k \cdot c_1$

$\frac{n}{3^k} = 1 \Rightarrow k = \log_3 n$

$\therefore T(n) = c_2 + c_1 \cdot \log_3 n = O(\log n)$

"Oh", "Omega" and "Theta" notation

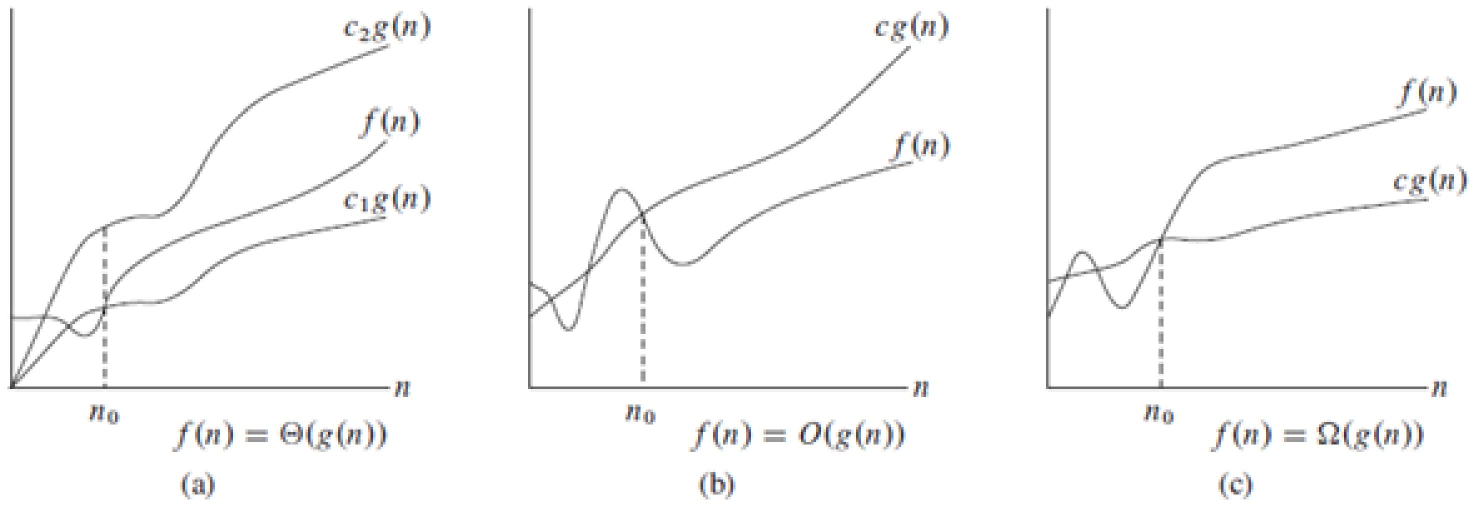
$f(n) = O(g(n))$
$\Rightarrow \ \exists\, c, n_0$ s.t.

$f(n) \le \ c\, g(n) \quad \forall\, n \ge n_0$

$f(n) = \Omega(g(n))$
$\Rightarrow \ \exists\, c, n_0$ s.t.

$f(n) \ge c\, g(n) \quad \forall\, n \ge n_0$

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



$f(n) = \Theta(g(n))$
iff
$f(n) = O(g(n))$
and
$f(n) = \Omega(g(n))$

Some commonly used functions in O estimates..

- O(1) : finding max of 3 integers, swapping two integers
- O(log(n)) : binary search kind of solutions
- O(n) : linear search, initializing an array
- O(n log(n)) : many sorting algorithms
- O(n^2) : initializing an n X n matrix, nested loops
- O(2^n) : all subsets of an n-sized array.

The Growth of Combinations of Functions

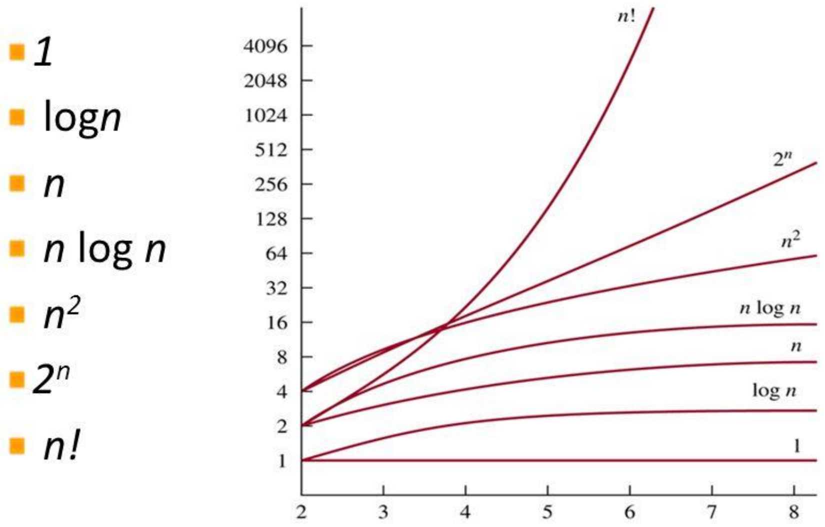


FIGURE 3 A Display of the Growth of Functions Commonly Used in Big-O Estimates.

Big O estimates…

- $n^2 + 2n + 1 = O(n^{\{k_1\}})$ $k_1 = 2$; $C = 4$, $n_0 = 1$ are witness for $O(n^2)$
  $C = 1$, $n_0 = 1$ are witness for $\Omega(n^2)$
- $n^2 + 0.0001\, n^3 = O(n^{\{k_2\}})$ $K_2 = 3$ SIMILARLY COME UP WITH CONSTANTS FOR $O(n^3)$ and $\Omega(n^3)$
- $3\log(n!) + (n+3)\log(n) = O(n^{\{k_3\}})$ $K_3 = 2$
  $f(n) = O(n^2)$ HOWEVER $f(n) \neq \Omega(n^2)$
- $n^{\{1+0.01\}}$ *is NOT* $O(n)$

$n^{1.001} \neq O(n)$ assume $\exists c, n_0$ s.t. $n^{1.001} \leq c \cdot n \ \forall n \geq n_0$
$\Rightarrow n^{0.001} \leq c \ \forall n \geq n_0$ CONTRADICTION

In general, if running time of an algorithm as $O(n^k)$ for any constant k, we call such an algorithm an efficient algorithm.

Big O as a relation..

- $O(g(n))$ is a set of functions $f(n)$ such that ..
- Hence it is technically more precise to say $f(n) \in O(g(n))$.

- What properties does the relation Big O satisfy?
  - Reflexive ✓ $f(n) = O(f(n))$
  - Transitive ✓ $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
  - Symmetric ✗ WE HAVE SEEN THIS EARLIER

- If $f(n) = O(g(n))$ then it need not be that $f(n)/g(n) = O(1)$

Similarly analyze Omega and Theta as relations

↳ IS SYMMETRIC.

Useful rules..

- $T_1(n) = O(f_1(n))$ and $T_2(n) = O(f_2(n))$ then
  - $T_1(\mathrm{n}) + \mathrm{T}_2(\mathrm{n})$ = max $(O(f_1(n)), O(f_2(\mathrm{n})))$
  - $T_1(\mathrm{n}) * \mathrm{T}_2(\mathrm{n})$ = ~~max~~ $(O(f_1(n) * f_2(n))$

- If $T(n)$ is a polynomial of degree k, then $T(n) = \Theta(n^k)$

- log(n) = O(n) and in fact for any constant k, (log(n))^k = O(n)

Little oh .. Yes there is one!

$$f(n) = o(g(n)) \Rightarrow \forall c > 0, \exists n_0 \text{ s.t.}$$
$$f(n) < c\,g(n) \quad \forall\, n \geq n_0$$

Note the two crucial changes

- The forall instead of there exists for the constant c.
- The < inequality versus the <= inequality between f(n) and c g(n).

Big O gives an upper bound, it may or may not be tight.

Little oh bound is always loose.

Is there a little omega? What about little theta?

$$2n^2 = O(n^2) \; but \; 2n^2 \neq o(n^2)$$

YES THERE IS LITTLE ω. NO LITTLE Θ

$$2n^2 = o(n^3)$$

Analogy with real numbers..

- $f(n) = O(g(n))$ is like $a \leq b$
- $f(n) = \Omega(g(n))$ is like $a \geq b$
- $f(n) = \Theta(g(n))$ is like $a = b$
- $f(n) = o(g(n))$ is like $a < b$
- $f(n) = \omega(g(n))$ is like $a > b$

Are there more??

Does the analogy break down?

THERE ARE FUNCTIONS $f(n)$ and $g(n)$ s.t. NEITHER $f(n) = O(g(n))$ NOR $g(n) = O(f(n))$.

FIND SUCH FUNCTIONS.

Efficient algorithms

Running time of an algorithm is the measured as the maximum time the algorithm requires on any input of size n. This is called as worst case analysis.

Some algorithms may not work differently for different inputs.

Some may do different things based on the input, for example, a sorting algorithm may do a check whether the input array is sorted.

We say an algorithm is efficient if it runs in time $O(n^c)$ for some constant c in the worst case.