Design and Analysis of Algorithms

Noor-ul-Ain

National University of Computer and Emerging Sciences, Islamabad

Algorithm Characteristics

The necessary features of an algorithm:

- *Definiteness*
  - The steps of an algorithm must be precisely defined.
- *Effectiveness*
  - Individual steps are all do-able.
- *Finiteness*
  - It won't take forever to produce the desired output for any input in the specified domain.
- *Output*
  - Information or data that goes out.

Algorithm Characteristics

Other important features of an algorithm:

- *Input*.
  - Information or data that comes in.
- *Correctness.*
  - Outputs correctly relate to inputs.
- *Generality.*
  - Works for many possible inputs.
- *Efficiency.*
  - Takes little time & memory to run.

Complexity Analysis

Want to achieve platform-independence

Use an abstract machine that uses *steps* of time and *units* of memory, instead of seconds or bytes

- ✓ each elementary operation takes **1 step**
- ✓ each elementary instance occupies **1 unit of memory**

Standard Analysis Techniques

- Constant time statements
- Analyzing Loops
- Analyzing Nested Loops
- Analyzing Sequence of Statements
- Analyzing Conditional Statements

Analysing an Algorithm

- **Simple statement sequence**

  $s_1; \; s_2; \; \ldots. \; ; \; s_k$

  - **Basic Step = 1 as long as k is constant**
- **Simple loops**

  ```
  for(i=0; i<n; i++) { s; }
  ```

  where `s` is Basic Step = 1

  - **Basic Steps : n**
- **Nested loops**

  ```
  for(i=0; i<n; i++)
      for(j=0; j<n; j++) { s; }
  ```

  - **Basic Steps : n^2**

Analysing an Algorithm

- **Loop index depends on outer loop index**

```
for(j=0;j<=n;j++)
    for(k=0;k<j;k++){
        s;
    }
```

  - **Inner loop executed**
    - **1, 2, 3, …., n times**

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$\therefore$ **Basic Steps : n^2**

Analysing an Algorithm

// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

```
int Sum(int A[], int N)
{
  int s=0;
  for (int i=0; i< N; i++)
    s = s + A[i];
  return s;
}
```

How should we analyse this?

Analysing an Algorithm

```
// Input: int A[N], array of N integers
// Output: Sum of all numbers in array A

int Sum(int A[], int N){
    int s=0;

    for (int i=0; i< N; i++)
        s = s + A[i];
    return s;
}
```

1,2,8: Once

3,4,5,6,7: Once per each iteration of for loop, N iteration

Total: 5N + 3

The *complexity function* of the algorithm is : $f(N) = 5N + 3$

Constant time statements

- **Simplest case: O(1) time statements**
- **Assignment statements of simple data types**

```
int x = y;
```

- **Arithmetic operations:**

```
x = 5 * y + 4 - z;
```

- **Array referencing:**

```
A[j] = 5;
```

- **Most conditional tests:**

```
if (x < 12) ...
```

Analyzing Loops

- **Any loop has two parts:**
  - **How many iterations are performed?**
  - **How many steps per iteration?**

```
int sum = 0,j;
for (j=0; j < N; j++)
  sum = sum +j;
```

Analyzing Loops

- **Any loop has two parts:**
  - **How many iterations are performed?**
  - **How many steps per iteration?**

```
int sum = 0,j;
for (j=0; j < N; j++)
  sum = sum +j;
```

  - **Loop executes N times (0..N-1)**
  - **O(1) steps per iteration**
- **Total time is N * O(1) = O(N*1) = O(N)**

Class Activity

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

Class Activity

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

$F(n) = n/2;$

$F(n) = O(n)$

Analyzing Loops

- **What about this for loop?**

```
int sum =0, j;
 for (j=0; j < 100; j++)
    sum = sum +j;
```

- **Loop executes 100 times**
- **O(1) steps per iteration**
- **Total time is 100 * O(1) = O(100 * 1) = O(100) = O(1)**

Analyzing Nested Loops

- **Treat just like a single loop and evaluate each level of nesting as needed:**

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

Analyzing Nested Loops

- **Treat just like a single loop and evaluate each level of nesting as needed:**

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

n+1 times

(n) * n+1 times

n * n times

$F(n) = 2n^2+2n+1$

$F(n) = O(n^2)$

Analyzing Nested Loops

- **Treat just like a single loop and evaluate each level of nesting as needed:**

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

- **Start with outer loop:**
  - **How many iterations? N**
  - **How much time per iteration? Need to evaluate inner loop**
- **Inner loop uses O(N) time**
- **Total time is N * O(N) = O(N*N) = O(N^2)**

Analyzing Nested Loops

- **Treat just like a single loop and evaluate each level of nesting as needed:**

```
int j,k;
for (j=0; j<N; j++)
    for (k=N; k>0; k--)
        sum += k+j;
```

?

Analyzing Nested Loops

- **Treat just like a single loop and evaluate each level of nesting as needed:**

```
int j,k;
for (j=0; j<N; j++)
    for (k=N; k>0; k--)
        sum += k+j;
```

- **Start with outer loop:**
  - **How many iterations? N**
  - **How much time per iteration? Need to evaluate inner loop**

- **Inner loop uses O(N) time**
- **Total time is N * O(N) = O(N*N) = O(N^2)**

Analyzing Nested Loops

- **Treat just like a single loop and evaluate each level of nesting as needed:**

```
int j,k;
for (j=0; j<N; j++)
    for (k=N; k>0; k--)
        sum += k+j;
```

- **Start with outer loop:**
  - **How many iterations? N**
  - **How much time per iteration? Need to evaluate inner loop**

- **Inner loop uses O(N) time**
- **Total time is N * O(N) = O(N*N) = O(N^2)**

Analyzing Nested Loops

- **What if the number of iterations of one loop depends on the counter of the other?**

```
int j,k;
for (j=0; j < N; j++)
    for (k=0; k < j; k++)
        sum += k+j;
```

- **Analyze inner and outer loop together:**
- **Number of iterations of the inner loop is:**
- $\mathbf{0 + 1 + 2 + ... + (N-1) = O(N^2)}$

Class Activity

```
void add( int A[ ], int B[ ], int n)
{
    for (i=0; i<n; i++)
      {
         for (j=0; j< n; j++)
             {
                    c[i,j]  =A[i][k]  + B[k][j];

              }
         }
}
```

Class Activity

```
void multiply( int A[ ], int B[ ], int n)
{
   for (i=0; i<n; i++)
    {
       for (j=0; j< n; j++)
          {
            c[i,j]= 0;
                for (k=0; k< n; ++)
                   {
                     c[i,j]+=A[i][k]*b[k][j];
                    }
          }
       }
}
```

Another Example

```
for (i=1; i < =n; i++)
    for (j =1; j < =i; j++)
        stmt;
```

Another Example

```
for (i=1; i <= n; i++)
    for (j =1; j <= i; j++)
      stmt;
```

i	j	No of iteration
1	1 2 x	1
2	1 2 3 X	2
3	1 2 3 4 x	3
. . .		
n	1 ….. n n+1 x	n

Steps:

$f(n) = 1+2+3+4+\ldots..n$

$f(n)=n(n+1)/2$

$f(n)=(n^2+2)/2$

$f(n)= O(n^2)$

Analyzing Sequence of Statements

```
for (j=0; j < N; j++)
   for (k =0; k < j; k++)
      sum = sum + j*k;
for (l=0; l < N; l++)
   sum = sum -l;
cout<<"Sum="<<sum;
```

Analyzing Sequence of Statements

- **For a sequence of statements, compute their complexity functions individually and add them up**

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

$O(N^2)$

```
for (l=0; l < N; l++)
   sum = sum -l;
```

$O(N)$

```
cout<<“Sum=”<<sum;
```

$O(1)$

Total cost is $O(N^2) + O(N) + O(1) = O(N^2)$

SUM RULE

Analysing an Algorithm

- **Loop index doesn't vary linearly**

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

Analysing an Algorithm

- **Loop index doesn't vary linearly**

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

  - ***i*** **takes values** $1, 2, 4, \ldots$ **until it exceeds** n

i

- **1**
- **$1x2=2$**
- **$2x2=2^2$**
- **$2^2x2=2^3$**
- **.**
- **.**
- **.**
- **2^k**

i

- **1**
- **1x2=2**
- **$2 \times 2 = 2^2$**
- **$2^2 \times 2 = 2^3$**
- **.**
- **.**
- **.**
- **2^k**

- **Assume**

- **$i >= n$**

  $i = 2^k$

  $2^k = n$

  $K = \log_2(n)$

Analyzing Conditional Statements

What about conditional statements such as

```
if (condition)
   statement1;
else
   statement2;
```

where statement1 runs in O(N) time and statement2 runs in $O(N^2)$ time?

We use "worst case" complexity: among all inputs of size N, that is the maximum running time?

The analysis for the example above is $O(N^2)$

Analysing an Algorithm

Growth of 5n+3

Estimated running time for different values of N:

N = 10	**=> 53 steps**
N = 100	**=> 503 steps**
N = 1,000	**=> 5003 steps**
N = 1,000,000	**=> 5,000,003 steps**

As N grows, the number of steps grow in *linear* proportion to N for this function *"Sum"*

Take Home Task

```
p=0;
for (i=0; p<n; i++)
    p=p+i
```

Some helpful mathematics

- $1 + 2 + 3 + 4 + \ldots + N$
  - $N(N+1)/2 = N^2/2 + N/2$ is $O(N^2)$

- $N + N + N + \ldots. + N$ (total of N times)
  - $N*N = N^2$ which is $O(N^2)$

- $1 + 2 + 4 + \ldots + 2^N$
  - $2^{N+1} - 1 = 2 \times 2^N - 1$ which is $O(2^N)$

10^6 instructions/sec, runtimes

N	$O(\log N)$	$O(N)$	$O(N \log N)$	$O(N^2)$
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

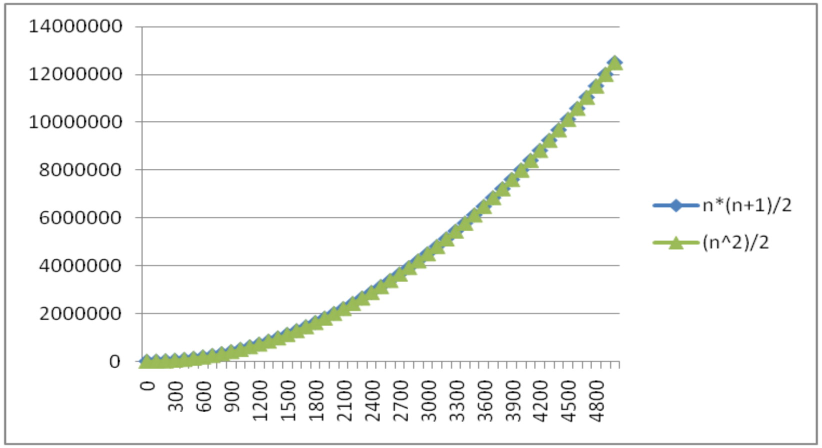
No Need To Be So Exact

Constants do not matter

- **Consider $6N^2$ and $20N^2$**
- **When N >> 20, the N^2 is what is driving the function's increase**

Lower-order terms are also less important

- **N*(N+1)/2 vs. just $N^2/2$**
- **The linear term is inconsequential**



We need a better notation for performance that focuses on the dominant terms only

What Dominates in Previous Example?

What about the +3 and 5 in 5N+3?

- **As N gets large, the +3 becomes insignificant**
- **5 is inaccurate, as different operations require varying amounts of time and also does not have any significant importance**

What is fundamental is that the time is *linear* in N.

Asymptotic Complexity: As N gets large, concentrate on the highest order term:

- ✓ **Drop lower order terms such as +3**
- ✓ **Drop the constant coefficient of the highest order term i.e. N**

Asymptotic Complexity

- **The 5N+3 time bound is said to "grow asymptotically" like N**
- **This gives us an approximation of the complexity of the algorithm**
- **Ignores lots of (machine dependent) details, concentrate on the bigger picture**

Asymptotic Analysis

- **Asymptotic analysis is an analysis of algorithms that focuses on**
  - **Analyzing problems of large input size**
  - **Consider only the leading term of the formula**
  - **Ignore the coefficient of the leading term**

Comparing Functions: Asymptotic Notation

- **Big Oh Notation: Upper bound**
- **Omega Notation: Lower bound**
- **Theta Notation: Tighter bound**

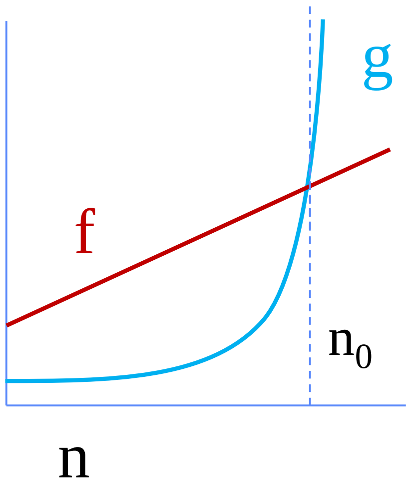
Big-Oh Rules

- **If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,**
    1. **Drop lower-order terms**
    2. **Drop constant factors**
- **Use the smallest possible class of functions**
    - **Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"**
- **Use the simplest expression of the class**
    - **Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"**

Big-Oh Notation

- **Given two functions f(n) & g(n) for input n, we say f(n) is in O(g(n)) iff there exist positive constants c and n_0 such that**

$$f(n) \leq c\, g(n) \quad \text{for all } n \geq n_0$$



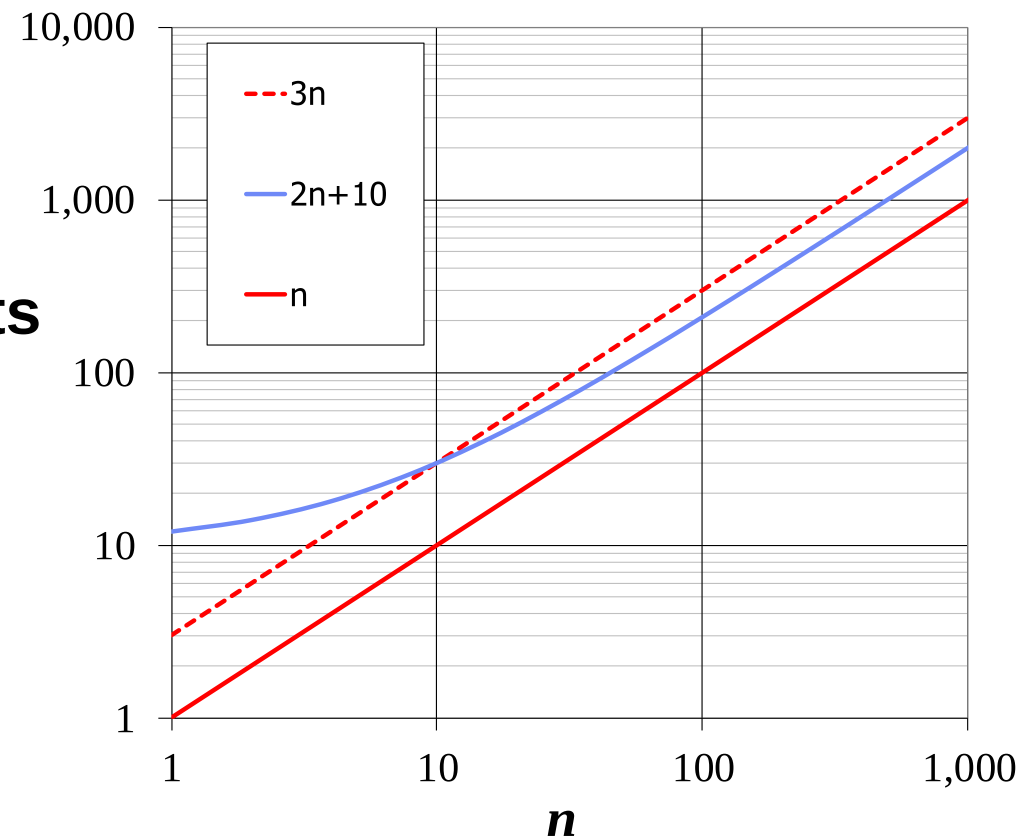
- **Basically, we want to find a function g(n) that is eventually always bigger than f(n)**

Big-Oh Notation (§3.4)

- **Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that**

  $$f(n) \leq cg(n) \quad \text{for } n \geq n_0$$

- **Example: $2n + 10$ is $O(n)$**
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Showing Order Briefly …

- **Show $10N^2 + 15N$ is $O(N^2)$**
- .

Showing Order Briefly …

- Show $10N^2 + 15N$ is $O(N^2)$
- Break into terms.

- $10N^2 \leq 10N^2$
- $15N \leq 15N^2$ for $N \geq 1$ (Now add)
- $10N^2 + 15N \leq 10N^2 + 15N^2$ for $N \geq 1$
- $10N^2 + 15N \leq 25N^2$ for $N \geq 1$
- $c = 25$, $N_0 = 1$
- Note, the choices for c and N_0 are not unique.

Take Home Task

- **Show $2N^3 + 10N$ is $O(N^3)$**

- **Show $2N^3 + 10N$ is not $O(N^2)$**

- **Show $2N^3 + 10N$ is $O(N^4)$**

-

The Gist of Big-Oh

Take functions f(n) & g(n), consider only the most significant term and remove constant multipliers:

- $5n+3 \rightarrow n$
- $7n+.5n^2+2000 \rightarrow n^2$
- $300n+12+n\log n \rightarrow n \log n$

Then compare the functions; if $f(n) \leq g(n)$, then $f(n)$ is in $O(g(n))$

Big Oh Notation

If f(N) and g(N) are two complexity functions, we say

$$f(N) = O(g(N))$$

(read "f(N) as order g(N)", or "f(N) is big-O of g(N)")

if there are constants c and N_0 such that for $N > N_0$,

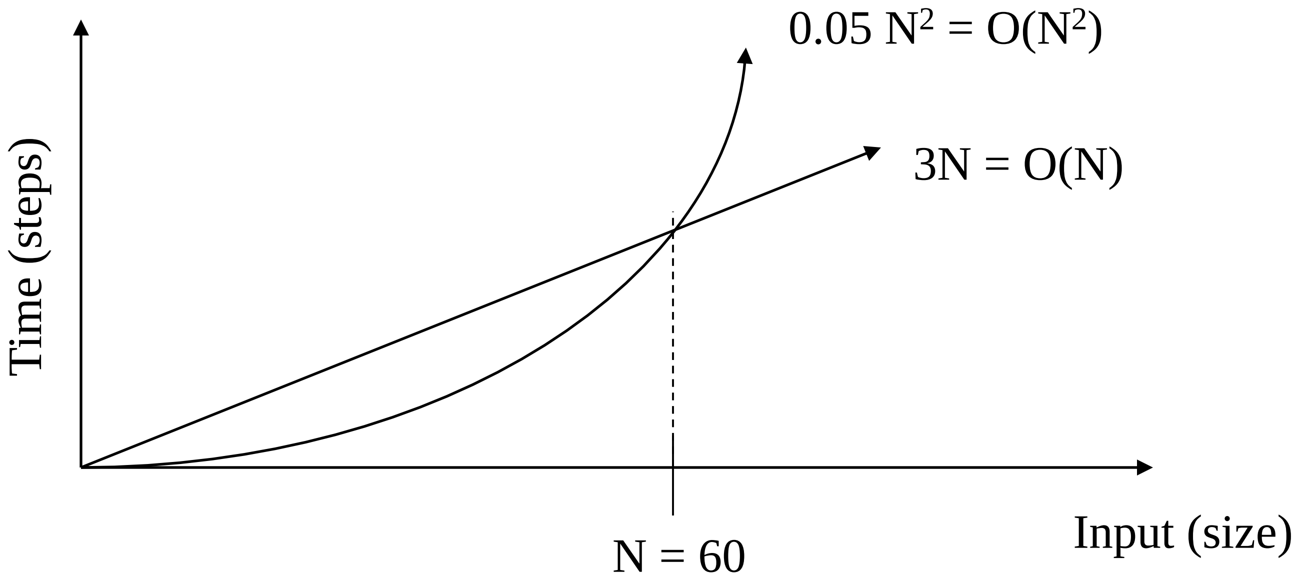
$$f(N) \le c * g(N)$$

for all sufficiently large N.

***big-O-Upper bound**

Comparing Functions

- **As inputs get larger, any algorithm of a smaller order will be more efficient than an algorithm of a larger order**



Big Omega Notation

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$

Example

- **Show 2n +3 is Ω(n)**

-

Example

- **Show $2n^2 + n + 1$ is $\Omega(n^2)$**

-

Example

- Show $2n^2 + n + 1$ is $\Omega(n^2)$
- $2n^2 + n + 1 >= cn^2$;for all $n>=1$
- $c=1$, $n_o=1$
-

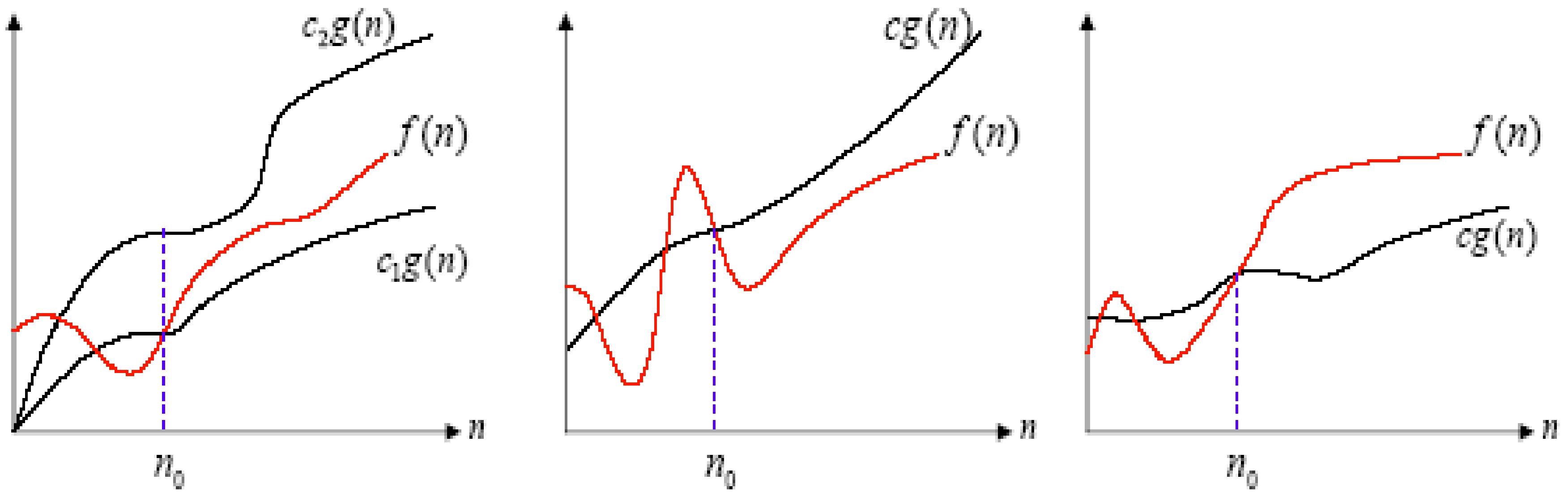
Take Home Task

- **Show $8n^3 + 5n^2 + 7$ is $\Omega(n^3)$**
-

Big Theta Notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$[1]

Asymptotic notation



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

- **Example:**
  - **$f(n) = 3n^5 + n^4 = \Theta(n^5)$**

Polynomial and Intractable Algorithms

- **Polynomial Time complexity**
  - **An algorithm is said to be polynomial if it is $O(n^d)$ for some integer d**
  - **Polynomial algorithms are said to be efficient**
    - **They solve problems in reasonable times!**

- **Intractable algorithms**
  - **Algorithms for which there is no *known* polynomial time algorithm**
  - ***We will come back to this important class later***

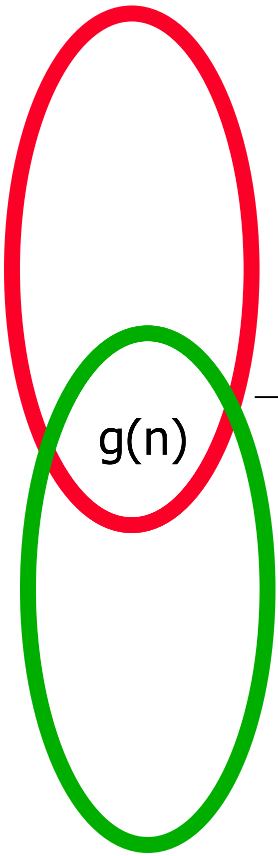
Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes

- **O(*g*(*n*)): class of functions *f*(*n*) that grow <u>no faster</u> than *g*(*n*)**

- **Θ(*g*(*n*)): class of functions *f*(*n*) that grow <u>at same rate</u> as *g*(*n*)**

- **Ω(*g*(*n*)): class of functions *f*(*n*) that grow <u>at least as fast</u> as *g*(*n*)**



>=

Ω(g(n)), functions that grow at least as fast as g(n)

=

Θ(g(n)), functions that grow at the same rate as g(n)

<=

O(g(n)), functions that grow no faster than g(n)

Performance Classification

f(*n*)	Classification
1	*Constant*: run time is fixed, and does not depend upon n. Most instructions are executed once, or only a few times, regardless of the amount of information being processed
`log` n	*Logarithmic*: when *n* increases, so does run time, but much slower. When *n* doubles, `log` *n* increases by a constant, but does not double until *n* increases to n^2. Common in programs which solve large problems by transforming them into smaller problems.
n	*Linear*: run time varies directly with *n*. Typically, a small amount of processing is done on each element.
n `log` n	When *n* doubles, run time slightly more than doubles. Common in programs which break a problem down into smaller sub-problems, solves them independently, then combines solutions
n^2	*Quadratic*: when *n* doubles, runtime increases fourfold. Practical only for small problems; typically the program processes all pairs of input (e.g. in a double nested loop).
n^3	*Cubic*: when n doubles, runtime increases eightfold
2^n	*Exponential*: when n doubles, run time squares. This is often the result of a natural, “brute force” solution.

TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

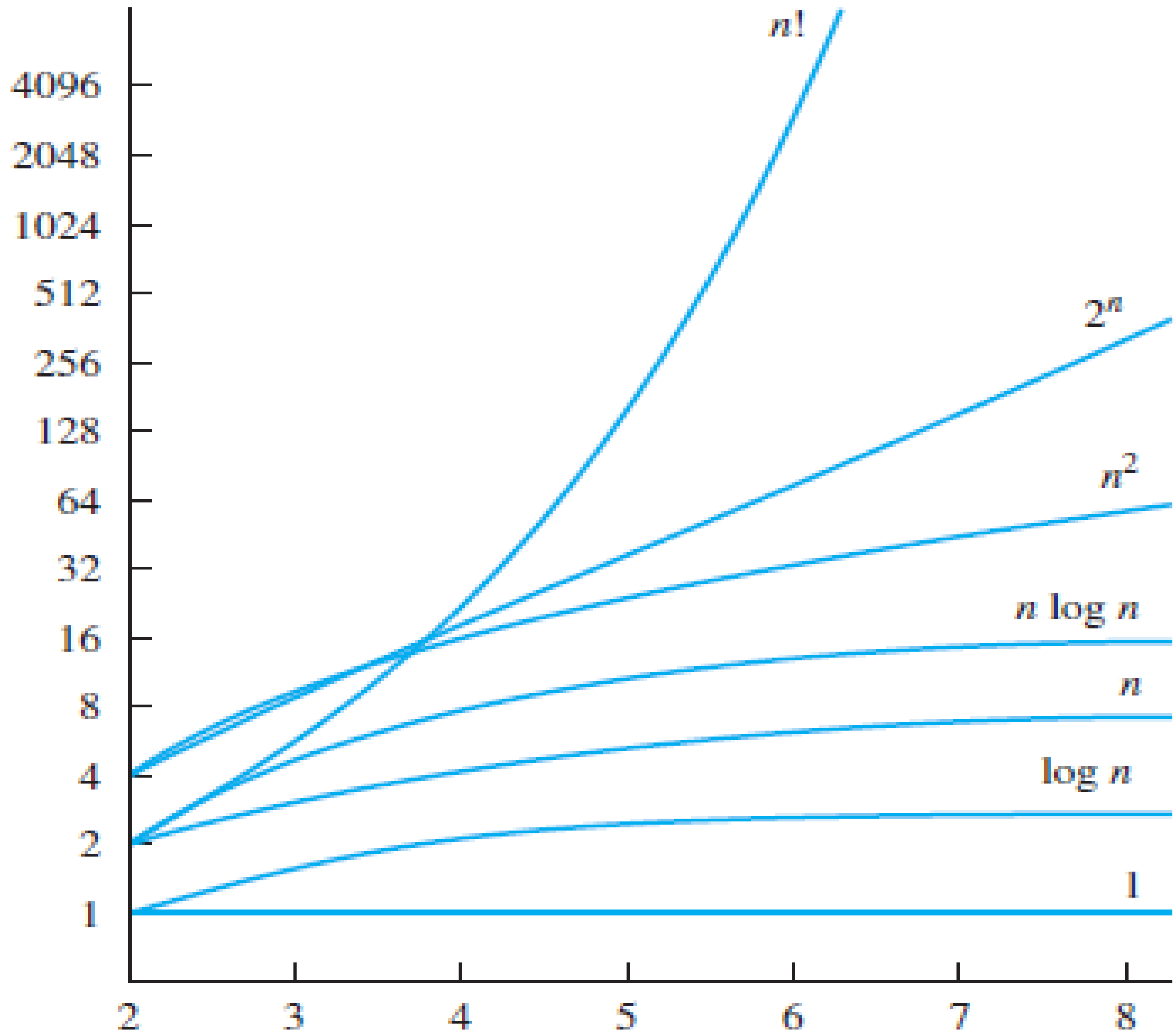
Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

Size does matter

What happens if we double the input size N?

N	$\log_2 N$	5N	$N \log_2 N$	N^2	2^N
8	3	40	24	64	256
16	4	80	64	256	65536
32	5	160	160	1024	$\sim 10^9$
64	6	320	384	4096	$\sim 10^{19}$
128	7	640	896	16384	$\sim 10^{38}$
256	8	1280	2048	65536	$\sim 10^{76}$

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



Typical Big O Functions – "Grades"

Function	Common Name
$N!$	factorial
2^N	Exponential
N^d, $d > 3$	Polynomial
N^3	Cubic
N^2	Quadratic
$N\sqrt{N}$	N Square root N
$N \log N$	N log N
N	Linear
$\sqrt{N}$	Root - n
$\log N$	Logarithmic
1	Constant

Running time grows 'quickly' with more input.

Running time grows 'slowly' with more input.

Theorem

- **If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.**
  - **The analogous assertions are true for the Ω-notation and Θ-notation.**

- **Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.**
  - **For example,** $5n^2 + 3n\log n \in O(n^2)$

- **Proof. There exist constants c_1, c_2, n_1, n_2 such that**
- $t1(n) \le c1*g1(n)$, **for all** $n \ge n1$
- $t2(n) \le c2*g2(n)$, **for all** $n \ge n2$
- **Define $c3 = c1 + c2$ and $n3 = \max\{n1,n2\}$. *Then***
- $t1(n) + t2(n) \le c3*\max\{g1(n), g2(n)\}$, **for all** $n \ge n3$

Review of Three Common Sets

$f(n) = O(g(n))$ means $c \times g(n)$ is an *Upper Bound* on $f(n)$

$f(n) = \Omega(g(n))$ means $c \times g(n)$ is a *Lower Bound* on $f(n)$

$f(n) = \Theta(g(n))$ means $c_1 \times g(n)$ is an *Upper Bound* on $f(n)$
and $c_2 \times g(n)$ is a *Lower Bound* on $f(n)$

These bounds hold for all inputs beyond some threshold n_0.

Properties of the O notation

- **Constant factors may be ignored**
  - $\forall\ k > 0,\ kf$ **is** $O(f)$
- **Higher powers grow faster**
  - n^r **is** $O(n^s)$ **if** $0 \leq r \leq s$

← **Fastest growing term dominates a sum**

- **If** f **is** $O(g)$, **then** $f + g$ **is** $O(g)$

  $eg\ \ an^4 + bn^3$ **is** $O(n^4)$

← **Polynomial's growth rate is determined by leading term**

- **If** f **is a polynomial of degree** d**,**
  then f **is** $O(n^d)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then fh is $O(gr)$
- Exponential functions grow faster than powers
  - n^k is $O(b^n)$ $\forall$ $b > 1$ and $k \geq 0$
    e.g. n^{20} *is* $O(1.05^n)$
- Logarithms grow more slowly than powers
  - $\log_b n$ is $O(n^k)$ $\forall$ $b > 1$ and $k > 0$
    e.g. $\log_2 n$ *is* $O(n^{0.5})$

Important!

Properties of the O notation

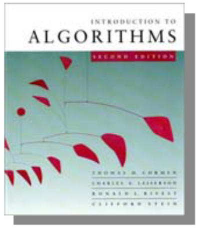
- **All logarithms grow at the same rate**
  - $\log_b n$ is $O(\log_d n)$ $\forall$ $b, d > 1$

- **Sum of first n r^{th} powers grows as the $(r+1)^{th}$ power**
  - $\sum_{k=1}^{n} k^r$ is $\Theta(n^{r+1})$

$$e.g.\quad \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \quad \text{is } \Theta(n^2)$$

Order of growth

- We usually consider one algorithm to be more efficient than another if its **worst case running time has a lower order of growth**.
- Due to constant factors and lower order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, $\Theta(n^2)$ algorithm, for example, will run more quickly in the worst case than $\Theta(n^3)$ algorithm



o-notation and ω-notation

O-notation and Ω-notation are like $\leq$ and $\geq$.
o-notation and ω-notation are like $<$ and $>$.

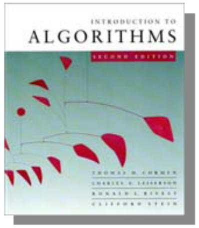
$o(g(n)) = \{ f(n)$: for any constant $c > 0$,
there is a constant $n_0 > 0$
such that $0 \leq f(n) < cg(n)$
for all $n \geq n_0$ $\}$

EXAMPLE: $2n^2 = o(n^3)$ $\quad (n_0 = 2/c)$

o-Notation

- **Examples**

$$n^{1.9999} = o(n^2)$$
$$n^2/\lg n = o(n^2)$$
$$n^2 \neq o(n^2) \text{ (just like } 2 \nless 2\text{)}$$
$$n^2/1000 \neq o(n^2)$$



o-notation and ω-notation

O-notation and Ω-notation are like $\leq$ and $\geq$.
o-notation and ω-notation are like $<$ and $>$.

$\omega(g(n)) = \{ f(n)$: for any constant $c > 0$,
there is a constant $n_0 > 0$
such that $0 \leq cg(n) < f(n)$
for all $n \geq n_0$ $\}$

EXAMPLE: $\sqrt{n} = \omega(\lg n)$ $\quad (n_0 = 1+1/c)$

ω- notation

$$n^{2.0001} = \omega(n^2)$$

$$n^2 \lg n = \omega(n^2)$$

$$n^2 \neq \omega(n^2)$$

Example

- $n^3 /1000 + 100n^2 + 1000n + 1000$

Example

- Give big-O estimates for the factorial function and the **logarithm of the factorial function** **log *n*!** , where

the factorial function $f(n) = n!$ is defined by

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdot \cdot \cdot \cdot n$$

Solution

- A big-O estimate for $n!$ can be obtained by noting that each term in the product does not exceed n. Hence,

$n! = 1 \cdot 2 \cdot 3 \cdot \cdot \cdot \cdot \cdot n$

$\leq n \cdot n \cdot n \cdot \cdot \cdot \cdot \cdot n$

$= n^n$

- This inequality shows that $n!$ is $O(n^n)$, taking $C = 1$ and $n_o = 1$ as witnesses. Taking logarithms of both sides of the inequality established for $n!$, we obtain
- **$\log n! \leq \log n^n = n \log n$**
- This implies that $\log n!$ is $O(n \log n)$, again taking $C = 1$ and $n_o = 1$ as witnesses.

Example

- Give a big-O estimate for
- $f(n) = 3n \log(n!) + (n^2 + 3) \log n$,
- where n is a positive integer.

Solution

- First, the product $3n \log(n!)$ will be estimated. From previous example we know that
- $\log(n!)$ is $O(n \log n)$.
- Using this estimate and the fact that $3n$ is $O(n)$,
- $3n \log(n!)$ is $O(n^2 \log n)$
- Next, the product $(n^2 + 3) \log n$ will be estimated. Because $(n^2 + 3) < 2n^2$ when $n > 2$, it follows that $n^2 + 3$ is $O(n^2)$.
- Thus it follows that $(n^2 + 3) \log n$ is $O(n^2 \log n)$.

- $f(n) = 3n \log(n!) + (n2 + 3) \log n$ is $\mathbf{O(n^2 \log n)}$