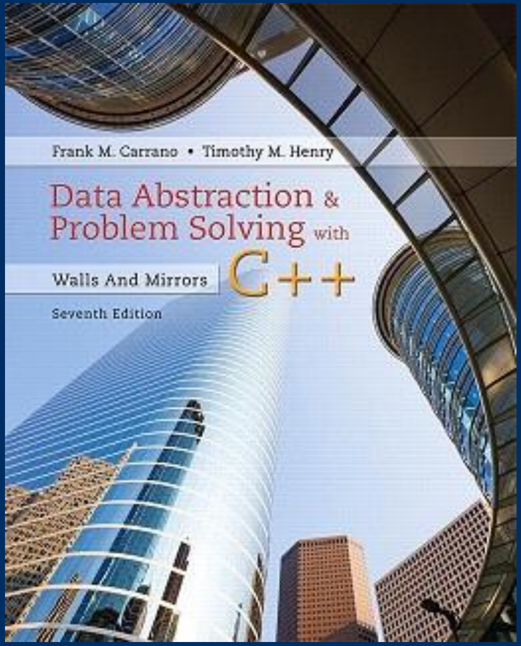
Chapter 10
Algorithm Efficiency

CS 302 - Data Structures

M. Abdullah Canbaz

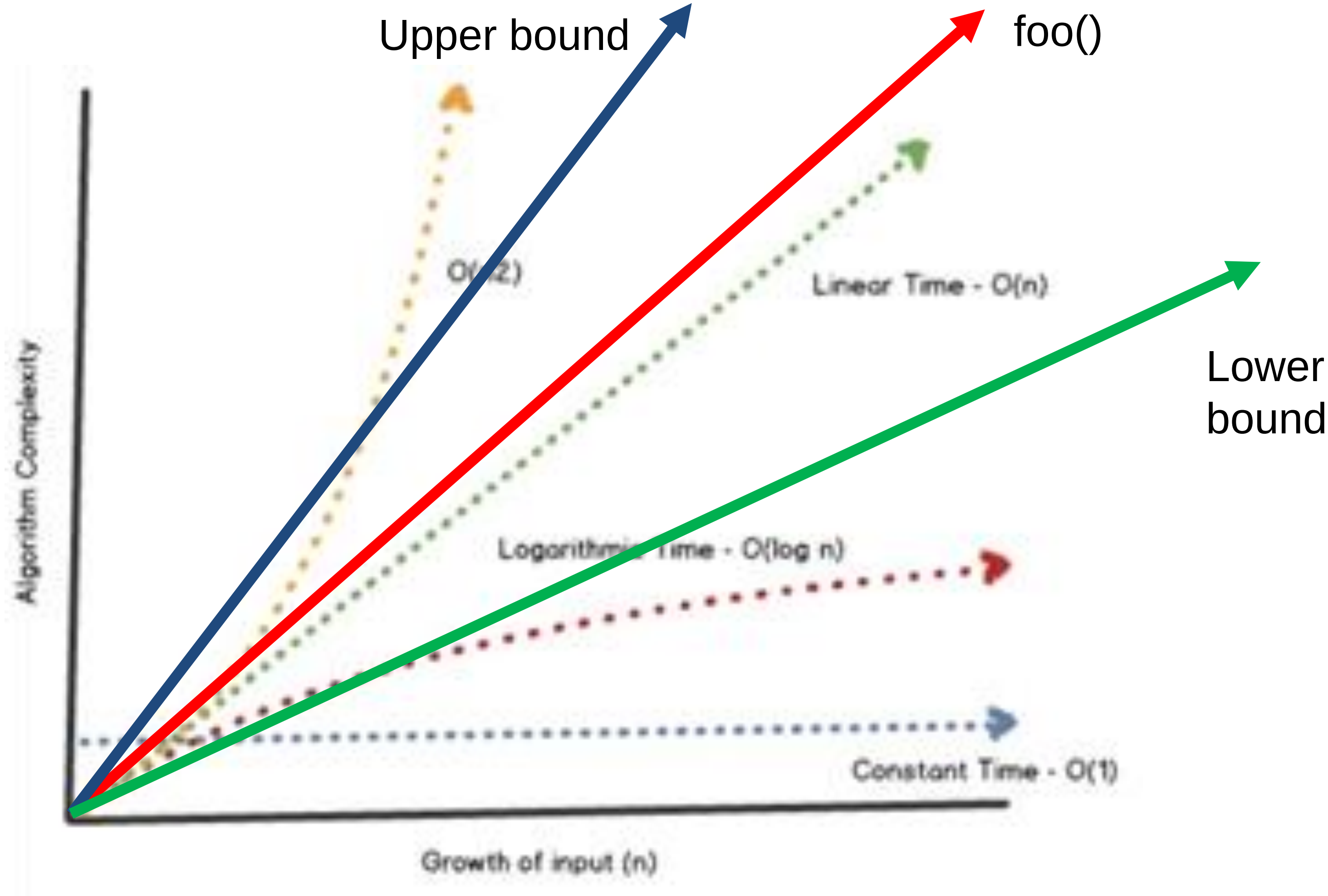


Modified from authors' slides

Reminders

- Assignment 3 is available
  - Due Feb 28th at 2pm
  - What to do?
- TA
  - Athanasia Katsila,
    Email: *akatsila [at] nevada {dot} unr {dot} edu*,
    Office Hours: Thursdays, 10:30 am - 12:30 pm at SEM 211
- Quiz 5 will be available
  - Wednesday between 4pm to 11:59pm

Complexity



Upper bound
foo()
Lower bound
Linear Time - O(n)
Logarithmic Time - O(log n)
Constant Time - O(1)
Algorithm Complexity
Growth of input (n)

Algorithm Efficiency

- There are often many approaches (algorithms) to solve a problem.
  - How do we choose between them?

- At the heart of computer program design are two (sometimes conflicting) goals
  - To design an algorithm that
    - is easy to understand, code, debug.
    - makes efficient use of the resources.

Algorithm Efficiency (cont)

- Goal (1) is the concern of Software Engineering.
- Goal (2) is the concern of data structures and algorithm analysis.
- When goal (2) is important,
  – how do we measure an algorithm's cost?

What Is a Good Solution?

- A program incurs a real and tangible cost.
  - Computing time
  - Memory required
  - Difficulties encountered by users
  - Consequences of incorrect actions by program
- A solution is good if …
  - The total cost incurs …
  - Over all phases of its life … is minimal

What Is a Good Solution?

- Important elements of the solution
  - Good structure
  - Good documentation
  - Efficiency
- Be concerned with efficiency when
  - Developing underlying algorithm
  - Choice of objects and design of interaction between those objects

Measuring Efficiency of Algorithms

- Important because
  - Choice of algorithm has significant impact

- Examples
  - Responsive word processors
  - Grocery checkout systems
  - Automatic teller machines
  - Video machines
  - Life support systems

Measuring Efficiency of Algorithms

- Analysis of algorithms
  - The area of computer science that provides tools for contrasting efficiency of different algorithms
  - Comparison of algorithms should focus on significant differences in efficiency
  - We consider comparisons of *algorithms*, not programs

Measuring Efficiency of Algorithms

- Difficulties with comparing programs (instead of algorithms)
  - How are the algorithms coded
  - What computer will be used
  - What data should the program use

- Algorithm analysis should be independent of
  - Specific implementations, computers, and data

The Execution Time of Algorithms

- An algorithm's execution time is related to number of operations it requires.
  - Algorithm's execution time is related to number of operations it requires.

- Example: Towers of Hanoi
  - Solution for *n* disks required $2^n - 1$ moves
  - If each move requires time *m*
  - Solution requires $(2^n - 1) \times m$ time units

Execution Time of Algorithm

- Traversal of linked nodes – example:

```
Node<ItemType>* curPtr = headPtr;           ← 1 assignment
while (curPtr != nullptr)                   ← n + 1 comparisons
{
   cout << curPtr->getItem() < endl;        ← n writes
   curPtr = curPtr->getNext();              ← n assignments
}  // end while
```

- Displaying data in linked chain of *n* nodes requires time proportional to *n*

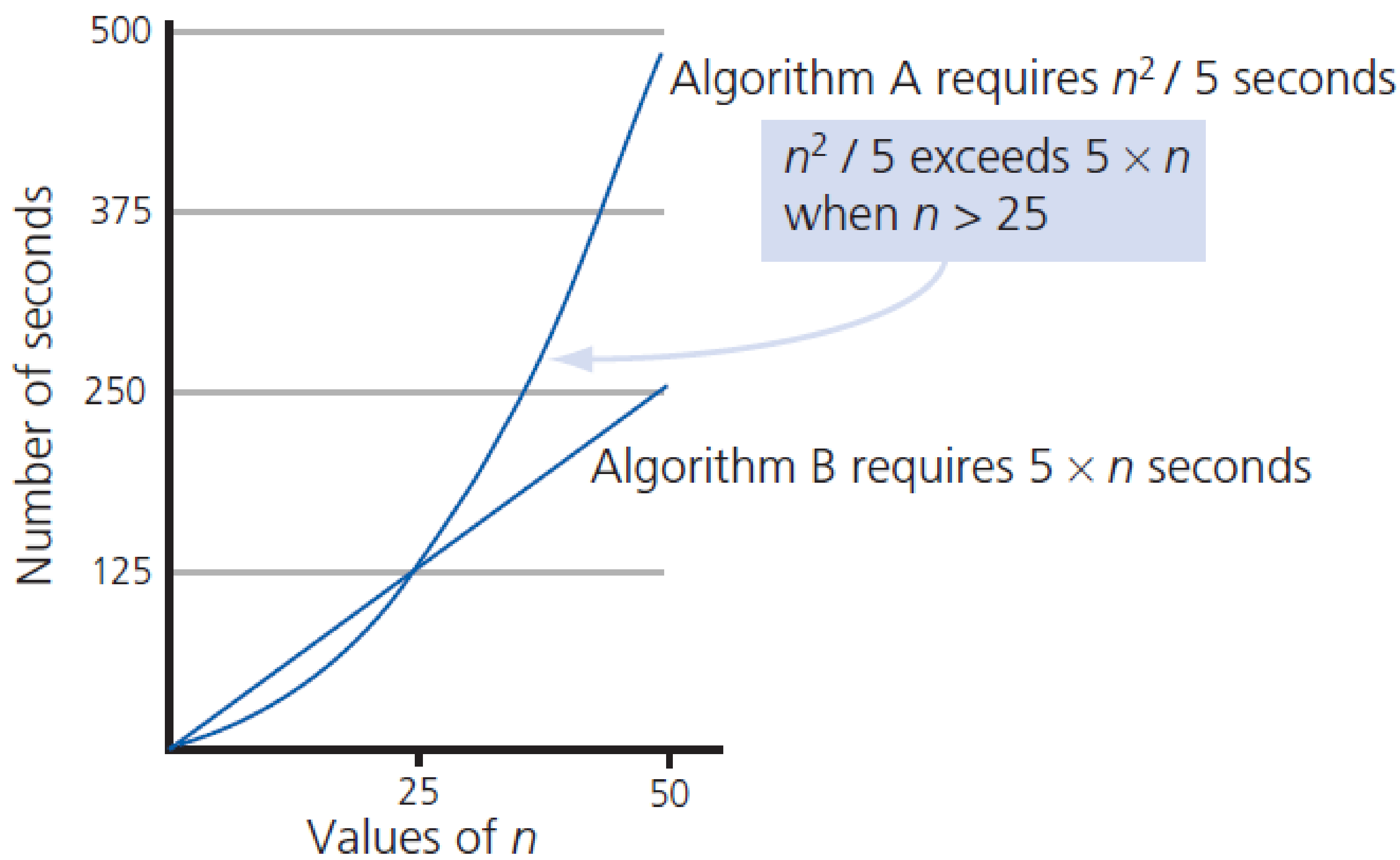
Algorithm Growth Rates

- Measure an algorithm's time requirement as function of problem size
- Most important thing to learn
  - How quickly algorithm's time requirement grows as a function of problem size

    Algorithm A requires time proportional to n^2
    Algorithm B requires time proportional to n

  - Demonstrates contrast in growth rates

Algorithm Growth Rates



Time requirements as a function of the problem size n

Best, Worst, Average Cases

- Not all inputs of a given size take the same time to run.
- Sequential search for K in an array of n integers:
  - Begin at first element in array and look at each element in turn until K is found
  - Best case:
  - Worst case:
  - Average case:

Time Analysis

- **Provides upper and lower bounds of running time.**

$$Lower\ Bound \le Running\ Time \le Upper\ Bound$$

Worst Case

- Provides an upper bound on running time.
- An absolute guarantee that the algorithm would not run longer, no matter what the inputs are.

$$\text{Lower Bound} \le \text{Running Time} \le \text{Upper Bound}$$

Best Case

- Provides a lower bound on running time.
- Input is the one for which the algorithm runs the fastest.

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

Average Case

- Provides an estimate of “average” running time.
- Assumes that the input is random.
- Useful when best/worst cases do not happen very often
  - i.e., few input cases lead to best/worst cases.

$$\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}$$

Which Analysis to Use?

- While average time appears to be the fairest measure,
  - it may be difficult to determine.

- When is the worst case time important?

How to Measure Efficiency?

- Critical resources:
  - Time, memory, battery, bandwidth, programmer effort, user effort

- Factors affecting running time:
  - For most algorithms, running time depends on "size" of the input.
  - Running time is expressed as T(n) for some function T on input size n.

How do we analyze an algorithm?

- Need to define objective measures.

(1) Compare execution times?

Empirical comparison (run programs)

Not good: times are specific to a particular machine.

(2) Count the number of statements?

Not good: number of statements varies with programming language and programming style.

How do we analyze an algorithm?

(3) Express running time t as a <u>function</u> of problem size n (i.e., $t=f(n)$)

Asymptotic Algorithm Analysis

- Given two algorithms having running times $f(n)$ and $g(n)$,
  - find which functions grows faster

- Such an analysis is <u>independent</u> of machine time, programming style, etc.

Comparing algorithms

- Given two algorithms having running times f(n) and g(n), how do we decide which one is faster?

- Compare "rates of growth" of f(n) and g(n)

Understanding Rate of Growth

- The low order terms of a function are relatively insignificant for **<u>large</u>** *n*

$$n^4 + 100n^2 + 10n + 50$$

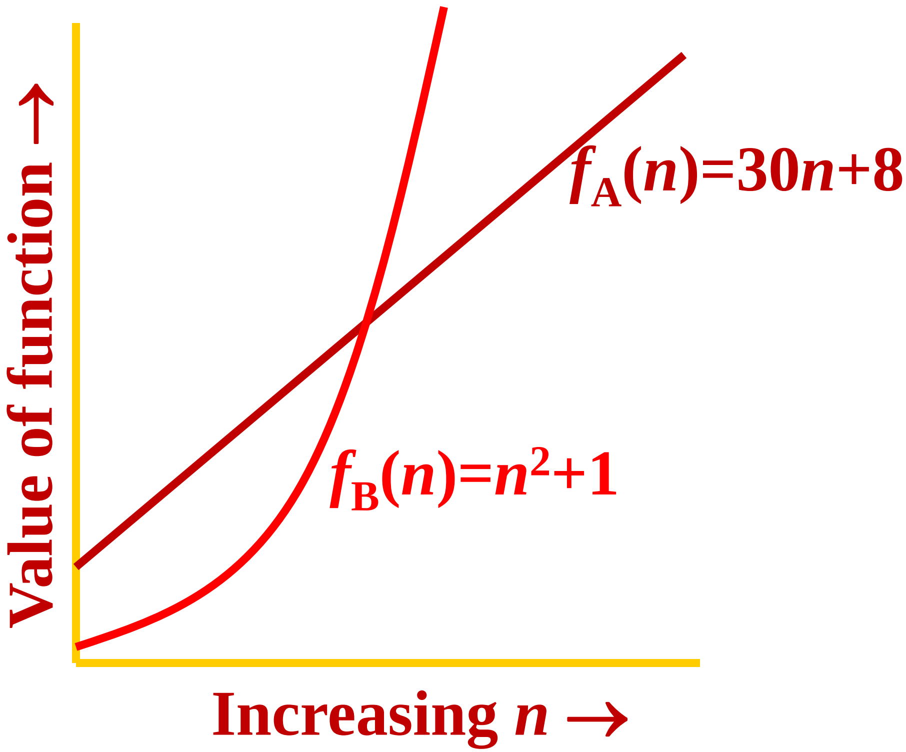
Approximation:

$$n^4$$

- Highest order term determines rate of growth!

Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...

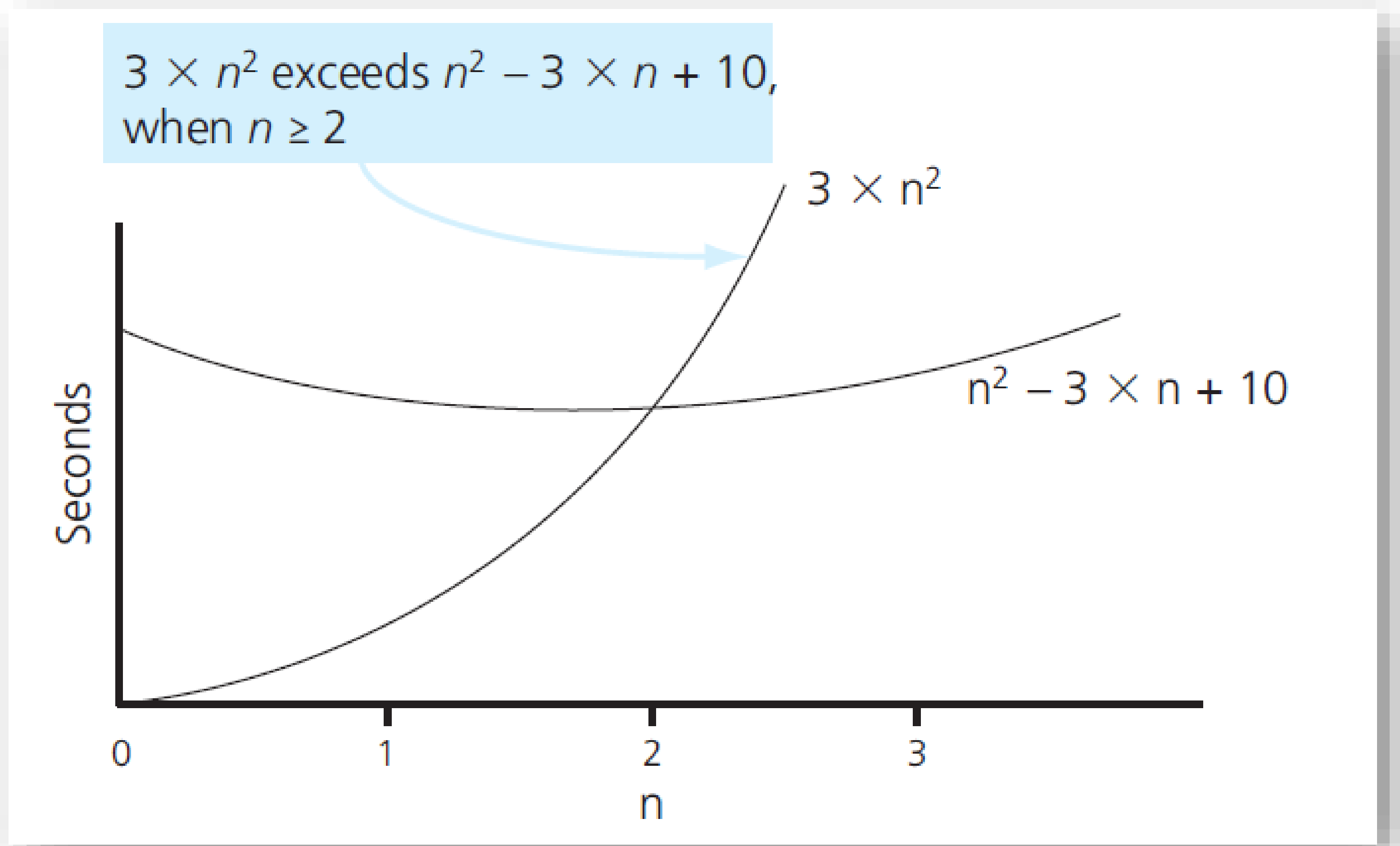


Analysis and Big O Notation

- Algorithm *A* is said to be order *f* (*n*),
  - Denoted as O(*f* (*n*))
  - Function f(n) called algorithm's growth rate function
  - Notation with capital O denotes *order*
- Algorithm *A* of order denoted O(f(n))
  - Constants k and n_0 exist such that
  - *A* requires no more than $k \times f(n)$ time units
  - For problem of size $n \geq n_0$

Rate of growth

- The graphs of $3 \times n^2$ and $n^2 - 3 \times n + 10$



Rate of Growth ≡ Asymptotic Analysis

- Using rate of growth as a measure to compare different functions implies comparing them asymptotically
  - i.e., as $n \rightarrow \infty$

- If f(x) is faster growing than g(x), then f(x) always eventually becomes larger than g(x) in the limit
  - i.e., for large enough values of x

Complexity

- Let us assume two algorithms A and B that solve the same class of problems.
- The time complexity of A is **5,000n**, the one for B is $\lceil \mathbf{1.1^n} \rceil$ for an input with **n** elements
- For n = 10,
  - A requires 50,000 steps,
  - but B only 3,
  - so B seems to be superior to A.
- For n = 1000, A requires $5 \cdot 10^6$ steps,
  - while B requires $2.5 \cdot 10^{41}$ steps.

Names of Orders of Magnitude

$O(1)$	**bounded (by a constant) time**
$O(\log_2 N)$	**logarithmic time**
$O(N)$	**linear time**
$O(N*\log_2 N)$	**$N*\log_2 N$ time**
$O(N^2)$	**quadratic time**
$O(N^3)$	**cubic time**
$O(2^N)$	**exponential time**

Asymptotic Notations

```
bool IsFirstElementNull(IList<string> elements)
{
    return elements[0] == null;
}
```

$O(1)$

```
bool ContainsValue(IList<string> elements, string value)
{
    foreach (var element in elements)
    {
        if (element == value) return true;
    }
    return false;
}
```

$O(n)$

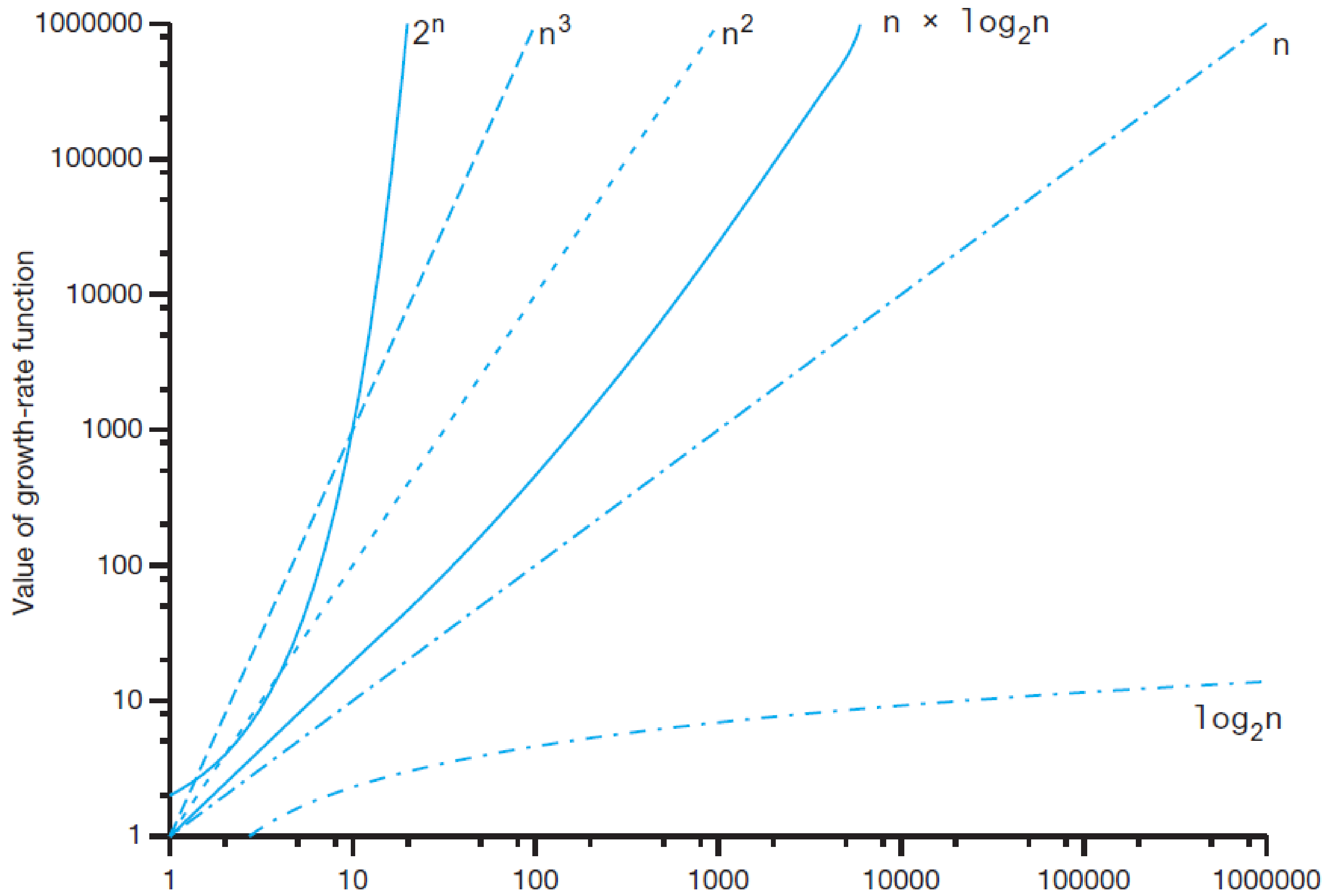
```
bool ContainsDuplicates(IList<string> elements)
{
    for (var outer = 0; outer < elements.Count; outer++)
    {
        for (var inner = 0; inner < elements.Count; inner++)
        {
            // Don't compare with self
            if (outer == inner) continue;

            if (elements[outer] == elements[inner]) return true;
        }
    }

    return false;
}
```

$O(n^2)$

Order of growth of common functions

$$O(1) < O(\log_2 n) < O(n) < O(n \times \log_2 n) <$$
$$O(n^2) < O(n^3) < O(2^n) < O(n!)$$

A comparison of growth-rate functions



N	$\log_2 N$	$N*\log_2 N$	N^2	2^N
1	0	0	1	2
2	1	2	4	4
4	2	8	16	16
8	3	24	64	256
16	4	64	256	65,536
32	5	160	1024	$4.29*10^{9}$
64	6	384	4096	$1.84*10^{19}$
128	7	896	16,384	$3.40*10^{38}$

A comparison of growth-rate functions

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n \times \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Example

Rank the following **functions** by order of growth from the slowest to the fastest

(<u>notation:</u> $\lg n = \log_2 n$)

$1000,\ 10\lg n,\ 4n^2,\ \ n^2,\ \ 2^n,\ \ 100n,\ \ 2^{\lg n}$

$1000 \quad 10\lg n \quad 2^{\lg n} \quad 100n \quad n^2 \quad 4n^2 \quad 2^n$

(Slowest → Fastest)

Sample Execution Times

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.005 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.

†1 ms = 10^{-3} second.

The Growth of Functions

- A problem that can be solved with polynomial worst-case complexity is called **tractable**

- Problems of higher complexity are called **intractable**

- Problems that no algorithm can solve are called **unsolvable**

Analysis and Big O Notation

- Definition:
  - Algorithm A is order $f(n)$
    - Denoted $O(f(n))$
  - If constants k and n_0 exist
  - Such that A requires no more than $k \times f(n)$ time units to solve a problem of size $n \geq n_0$.

Analysis and Big O Notation

- Worst-case analysis
  - Worst case analysis usually considered
  - Easier to calculate, thus more common
- Average-case analysis
  - More difficult to perform
  - Must determine relative probabilities of encountering problems of given size

Asymptotic Notation

O notation: asymptotic "less than":

f(n)=O(g(n)) implies: f(n) "≤" c g(n) in the limit*

c is a constant

(used in worst-case analysis)

***formal definition in CS477/677**

Asymptotic Notation

Ω notation: asymptotic "greater than":

f(n)= Ω (g(n)) <u>implies:</u> f(n) "≥" c g(n) in the limit*

c is a constant

(used in best-case analysis)

***formal definition in CS477/677**

Asymptotic Notation

Θ notation: asymptotic "equality":

$f(n) = \Theta(g(n))$ <u>implies:</u> $f(n)$ "=" $c\, g(n)$ in the limit*

c is a constant

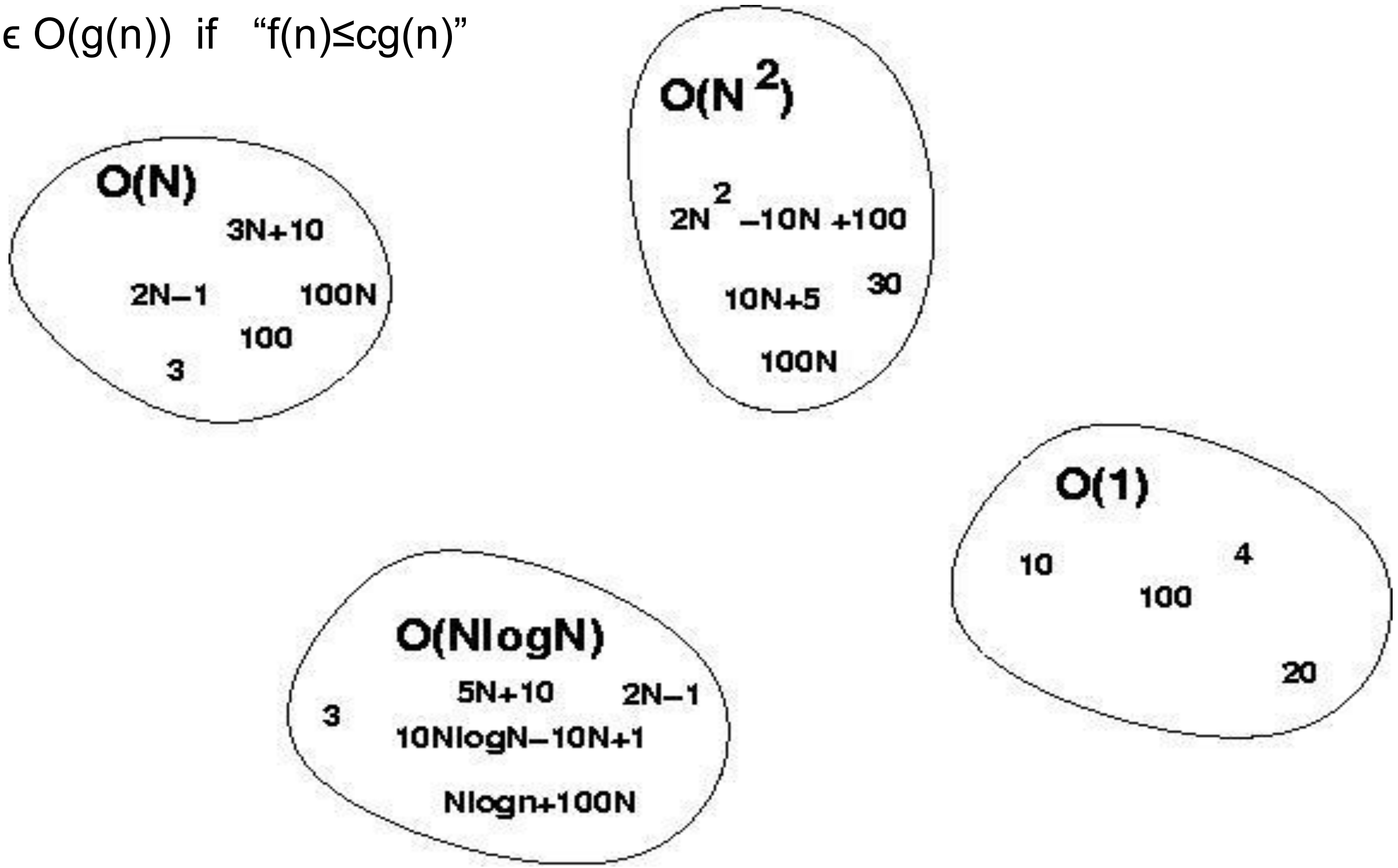
(provides a tight bound of running time)
(best and worst cases are same)

***formal definition in CS477/677**

More on big-O

O(g(n)) is a set of functions f(n)

f(n) ϵ O(g(n)) if "f(n)≤cg(n)"



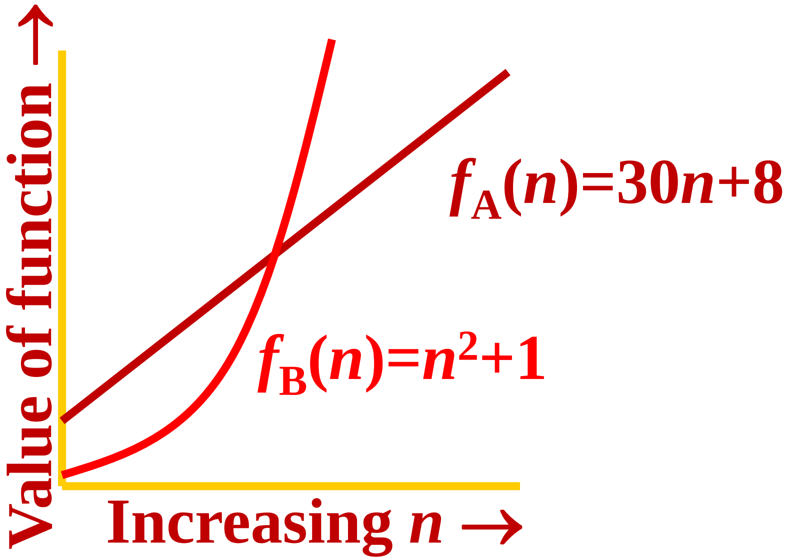
A Common Misunderstanding

- Confusing worst case with upper bound
- Upper bound refers to a growth rate
- Worst case refers to the worst input from among the choices for possible inputs of a given size

Algorithm speed vs function growth

- An O(n^2) algorithm will be slower than an O(n) algorithm (for large n).

- But an O(n^2) function will grow faster than an O(n) function.



Keeping Your Perspective

- Choosing implementation of ADT
  - Consider how frequently certain operations will occur
  - Seldom used but critical operations must also be efficient
- If problem size is always small
  - Possible to ignore algorithm's efficiency
- Weigh trade-offs between
  - Algorithm's time and memory requirements
- Compare algorithms for style and efficiency

Faster Computer or Algorithm?

- Suppose we buy a computer 10 times faster.
  - **n**: size of input that can be processed in one second on old computer
    - in 1000 computational units
  - **n'** : size of input that can be processed in one second on new computer

T(n)	n	n'	Change	n'/n
$10n$	100	1,000	$n' = 10n$	10
$10n^2$	10	31.6	$n' = \sqrt{10}n$	3.16
10^n	3	4	$n' = n + 1$	$1 + 1/n$

How do we find *f(n)*?

(1) Associate a "cost" with each statement

(2) Find total number of times each statement is executed

(3) Add up the costs

Properties of Growth-Rate Functions

- Ignore low-order terms
- Ignore a multiplicative constant in the high-order term
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
- Be aware of worst case, average case

Running Time Examples

```
i = 0;
while (i<N) {
    X=X+Y;                  // O(1)
    result = mystery(X);    // O(N)
    i++;                    // O(1)
}
```

- The body of the while loop: O(N)
- Loop is executed: <u>N times</u>

$N \times O(N) = O(N^2)$

Running Time Examples

```
if (i<j)
   for ( i=0; i<N; i++ )
       X = X+i;
else
   X=0;
```

O(N)

O(1)

Max (O(N), O(1)) = O (N)

Running Time Examples (cont.' d)

Algorithm 1 **Cost**

arr[0] = 0; c_1

arr[1] = 0; c_1

arr[2] = 0; c_1

...

arr[N-1] = 0; c_1

$c_1+c_1+...+c_1 = c_1 \times N$

Algorithm 2 **Cost**

```
for(i=0; i<N; i++)        c2
    arr[i] = 0;           c1
```

$(N+1) \times c_2 + N \times c_1 = (c_2 + c_1) \times N + c_2$

Running Time Examples (cont.'d)

	Cost
`sum = 0;`	c_1
`for(i=0; i<N; i++)`	c_2
`for(j=0; j<N; j++)`	c_2
`sum += arr[i][j];`	c_3

$c_1 + c_2 \times (N+1) + c_2 \times N \times (N+1) + c_3 \times N \times N$

56

Complexity Examples

What does the following algorithm compute?

```
int who_knows(int a[n])  {
    int m = 0;
    for {int i = 0; i<n; i++}
        for {int j = i+1; j<n; j++}
            if ( abs(a[i] - a[j]) > m )
                m = abs(a[i] - a[j]);
    return m;
}
```

returns the maximum difference between any two numbers in the input array

Comparisons: n-1 + n-2 + n-3 + … + 1 = (n-1)n/2 = $0.5n^2 - 0.5n$

Time complexity is $O(n^2)$

Complexity Examples

Another algorithm solving the same problem:

```
int max_diff(int a[n])  {
    int min = a[0];
    int max = a[0];
    for {int i = 1; i<n; i++}
        if ( a[i] < min )
            min = a[i];
        else if ( a[i] > max )
            max = a[i];
    return max-min;
}
```

Comparisons: 2n - 2

Time complexity is O(n)

Examples of Growth Rate

```
/* @return Position of largest value in "A“ */
static int largest(int[] A) {
  int currlarge = 0;  // Position of largest
  for (int i=1; i<A.length; i++)
    if (A[currlarge] < A[i])
       currlarge = i; // Remember position
  return currlarge;   // Return largest postn
}
```

Examples (cont)

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

Time Complexity Examples (1)

```
a = b;
```

This assignment takes constant time, so it is $\Theta(1)$.

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

Time Complexity Examples (2)

```
sum = 0;

for (j=1; j<=n; j++)
  for (i=1; i<=j; i++)
    sum++;

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

Time Complexity Examples (3)

```
sum1 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=i; j++)
      sum2++;
```

Time Complexity Examples (4)

```
sum1 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=n; j++)
    sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=k; j++)
    sum2++;
```

Example

Compare the two functions n^2 and $2^n/4$ for various values of n. Determine when the second becomes larger than the first.

$n^2 \geq 2^n/4$ for $n \leq 8$

So, $2^n/4$ becomes larger for $n \geq 9$

Analyze the complexity of the following code

(a) sum = 0; $O(1)$

$O(2n)$ for(i=1; i<=2*n; i++)

$O(1) \rightarrow$ sum = sum + 1;

Total: $O(2n) = O(n)$

(b) sum = 0; $O(1)$

$O(n^2)$ for(i=1; i<=n*n; i++)

$O(1) \rightarrow$ sum = sum + 1;

Total: $O(n^2)$

(c) sum=0; $O(1)$

$O(n)$ for(i=1; i<=n; i++)

$O(1) \rightarrow$ sum = sum + n;

Total: $O(n)$

(d) sum = 0; $O(1)$

$O(n)$ for(i=1; i<=n i++)

$O(i)$ for(j=1; j<=i; j++)

$O(1)$ sum = sum + i;

When $i=1$ $j \le 1$ 1 time

$i=2$ $j \le 2$ 2 times

$\vdots$

$i=n$ $j \le n$ n times

Total $1+2+\cdots+n = \frac{n(n+1)}{2} = O(n^2)$

(e

(e) sum = 0; $O(1)$

$O(1)$ for(i=1; i<=100; i++)

$O(n)$ for(j=1; j<=n; j++)

$O(1)$ sum = sum + i;

Total: $O(n)$

(f) sum = 0; $O(1)$

for(i=1; i<=n; i++) $O(n)$

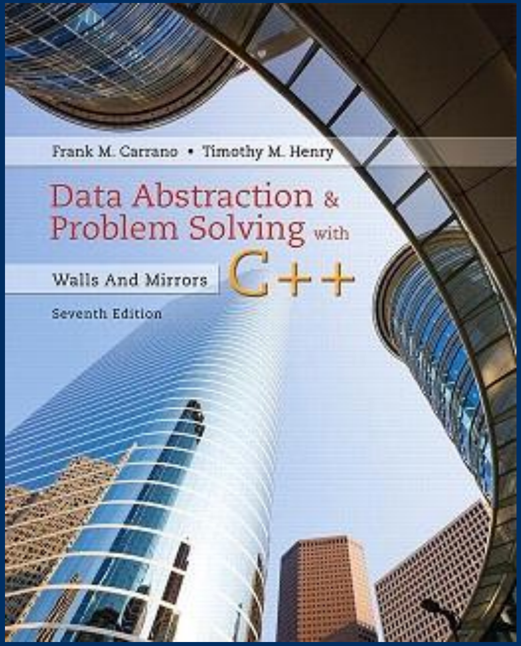
for(j=1; ~~i~~ j<=n; j*=2) $O(\lg n)$

sum = sum + 1; $O(1)$

Total: $O(n \lg n)$

Chapter 10
Algorithm Efficiency

The End



Modified from authors' slides

Out of the Box

Design a URL Shortener (TinyURL) System

From previous classes

Encode and Decode TinyURL

https://leetcode.com/problems/encode-and-decode-tinyurl/description/