Lecture 1

COMP 3760

Textbook: 1.1, 1.2, 1.3, 2.1

Math Review

Be sure to review those slides for **stuff you should know**

You'll need to know it to do lab and quiz problems!

Ask me questions on Discord (or DM or email) any time

Session topics

- Why do we care? What are we doing here?
- Define algorithm
- Examine time efficiency and space efficiency
- Determine the basic operation for a given algorithm represented in pseudocode
- Determine running time of an algorithm
- Define asymptotic notations (big-O)

Why do we care about algorithms?

- Algorithms are at the core of computer programming
- There are many important, standard algorithms
- We want to design new algorithms and analyze their efficiency

Important problem types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Numerical problems
- Optimization problems

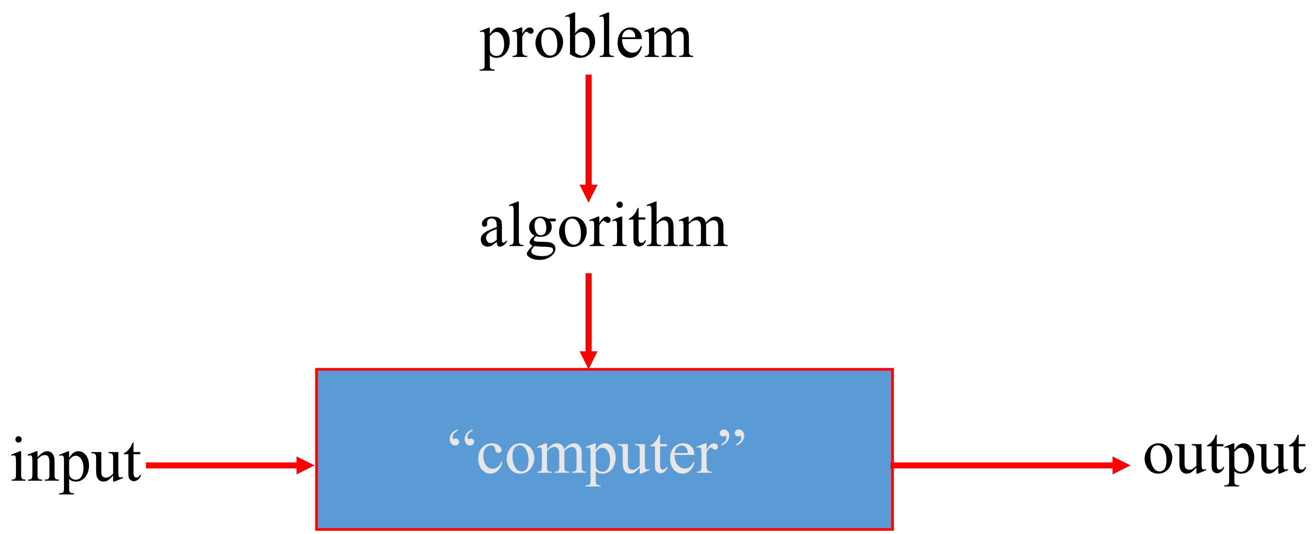
Algorithm design techniques

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

What is an Algorithm?

- Definition:

  An algorithm is a sequence of ***unambiguous instructions*** *for* ***obtaining a required output*** *for* ***any legitimate input*** *in a* ***finite amount of time***



There can’t be only one

- There is always more than one algorithm for the same problem
- We care about several characteristics:
  - Is it *correct*?
  - Is it *time-efficient*?
  - Is it *space-efficient*?

```
Algorithm DoSomething()
    count = 0
    do
        x = count+1
        secret = 100*x + 10*x + x
        if secret % 37 == 0
            count++
        endif
    while x < 100
    return count
END
```

```
Algorithm CountSomething2()
    count = 0
    for a = 1 to 100 do
        for b = 1 to 100 do
            val = 100*a + b
            if MathLib.isPrime(val)
                count++
            endif
        endfor
    endfor
    return count
END
```

```
Algorithm CountSomethingElse(int N)
    A = new array[1..N]
    for i = 1 to N do
        sum1 = 0
        for j = 1 to i do
            sum1 = sum1 + j
        endfor
        A[i] = sum1
    endfor
    sum2 = 0
    for i = 1 to N do
        sum2 = sum2 + A[i]
    endfor
    return sum2
END
```

Example

- Here is a pseudocode algorithm:

```
Algo: find( A[0…n-1] )
    m ← A[0]
    for i ← 1 to n-1 do
        if A[i] > m
            m ← A[i]
    return m
```

- What does it do?

It finds the largest element of an array

Correctness

- Will *find* work correctly?
  - for any possible input? (how many are there?)
  - within a finite amount of time?

- How would you argue this rigorously?

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
           m ← A[i]
   return m
```

Time Efficiency

- Is *find* a time-efficient algorithm?

- Seems good
  - To find the largest, you need to check each array element exactly once

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Space Efficiency

- Is *find* a space-efficient algorithm? (amount of memory)

- Again… it seems reasonable
  - Two temp variables introduced

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Variation of the problem

- What if you are guaranteed that A is pre-sorted?
- Is this *find()* algorithm still efficient?
- Could you do better?

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Why do we care?

- Think about computing the n^{th} Fibonacci number:
  - 0, 1, 1, 2, 3, 5, 8, 13, …

First algorithm

```
Algo: fib( n )
   if n ≤ 1
      return n
   else
      return fib( n-1 ) + fib( n-2 )
```

Java implementation

```
public static int fib(int n)  {
   if (n<=1)
      return n;
   else
      return ( fib(n-1) + fib(n-2) );
}
```

Why do we care, Part 2

- Now look at a different algorithm

Second algorithm

```
Algo: fib2( n )
   F[0] ← 0; F[1] ← 1;
   for i ← 2 to n do
      F[i] ← F[i-1] + F[i-2]
   return F[n]
```

```
public static int fib2(int n) {

   int[] f = new int[n+1];

   f[0] = 0;
   f[1] = 1;
   for (int i=2; i<=n; i++)
       f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

Difference

- First approach
  - Recursively calls the Fib function over and over again
- Second approach
  - Stores successive results so we don't have to re-compute them
- Very soon the second approach is much, much faster

N	Fib1 (ms)	Fib2 (ms)
30	9	0
31	11	0
32	22	0
33	83	0
34	90	0
35	148	0
36	237	0
37	429	0
38	722	0
39	1105	0
40	1627	0

So?

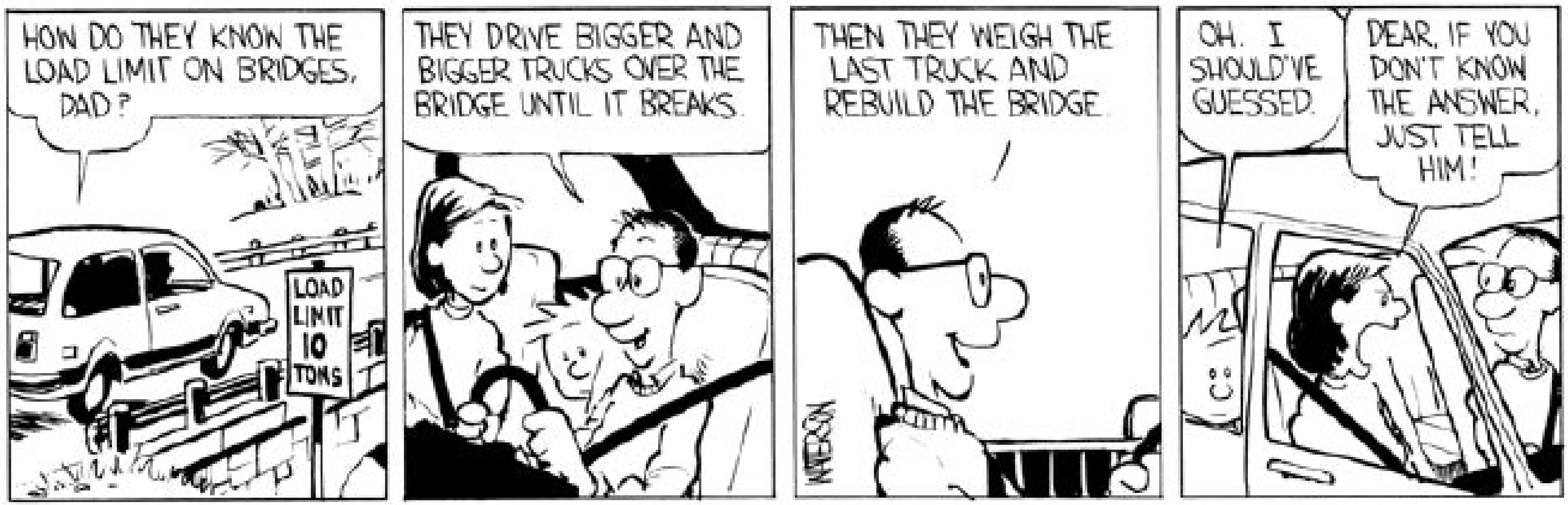
- Fib is a basic example of why we care about algorithm efficiency

- A well thought out algorithm can run much faster

- There can be big variation in efficiency

How to determine efficiency

- Could do it experimentally
  - i.e. Write a bunch of implementations, see which one is fastest



How to determine efficiency

- Could do it experimentally
    - i.e. Write a bunch of implementations, see which one is fastest
- Problem?
    - Time consuming and expensive
    - It is not accurate

- Another idea: estimate efficiency before writing code

How to determine efficiency

- What we know:

  1. Running time (efficiency) of an algorithm depends on the input size

  2. The total execution time for any algorithm depends primarily on the number of instructions executed
     - Different execution times of specific instructions is of secondary importance

Example

- Here’s “find” again:

```
1. Algo: find( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.        if A[i] > m
5.           m ← A[i]
6.     return m
```

for n=3

stmt	#times
1	0? 1?
2	1
3	2
4	2
5	2
6	1

- How many instructions are executed if n=3?

$$f(3) = 1 + 3*(3-1) + 1$$

Example

- What about n=8?

```
1. Algo: find( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.         if A[i] > m
5.             m ← A[i]
6.     return m
```

for n=8

stmt	#times
1	0
2	1
3	7
4	7
5	7
6	1

- f(8) = 1 + 3*(8-1) + 1
- For input of size n, the running time is

$$f(n) = 1 + 3*(n-1) + 1$$
$$= 3n - 1$$

Basic operations

- Which instruction in *find* gets executed the most?

```
1. Algo: find( A[0…n-1] )
2.       m ← A[0]
3.       for i ← 1 to n-1 do
4.           if A[i] > m
5.              m ← A[i]
6.       return m
```

stmt	(n=3) #times	(n=10) #times	(n=100) #times
1	0	0	0
2	1	1	1
3	2	9	99
4	2	9	99
5	2	9	99
6	1	1	1

- We define the basic operation of an algorithm as the statement that gets executed most frequently
  - Tiebreakers: deepest inside the loop; which one is more “expensive”; or maybe sometimes we don’t care

Basic operations

This is the fundamental concept we use to analyze algorithmic efficiency:

count the number of basic operations
executed for an input of size n

- Using this idea, we would say the efficiency of *find* is:

$$f(n) = n-1$$

- We don't count instructions that are not basic operations

Example 1

- Consider this algorithm:

```
1. Mystery1(n)  // n > 0
2.   S ← 0
3.   for i ← 1 to n do
4.       S ← S + i * i
5.   return S
```

1. What does this algorithm do? Calculates: $1^2 + 2^2 + 3^2 + \ldots + n^2$

2. What is the basic operation? It's line 4

3. How many times is the basic operation executed for input size n?

How many times?

```
Mystery(n)   // n > 0
  S ← 0
  for i ← 1 to n do
       S ← S + i * i
  return S
```

- Basic operation is executed once each time through the loop
  - 1^{st} time: 1
  - 2^{nd} time: 1
  - …
  - n^{th} time: 1
- So you have a sum: $\sum_{i=1}^{n} 1$
- What does this equal?

  1+ 1 + 1 … + 1 (n times)

  = n

Example 2

- Consider this algorithm:

```
1. Mystery2(A[0..n-1][0..n-1])  // n > 0
2.  S ← 0
3.  for i ← 0 to n-1 do
4.      for j ← 0 to n-1 do
5.          S ← S + A[i][j];
6.  return S
```

1. What does this algorithm do? Calculates sum of the elements in array A
2. What is the basic operation? Addition on line 5
3. How many times is the basic operation executed for input size n?

Example 2

```
Mystery2(A[0..n-1][0..n-1])   // n > 0
  S ← 0
  for i ← 0 to n-1 do
      for j← 0 to n-1 do
           S ← S + A[i][j];
  return S
```

- The outer loop
  - i goes from 0 to n-1
  - So we have:

$$\sum_{i=0}^{n-1} (whatever\ the\ inner\ loop\ is)$$

Example 2

```
Mystery2(A[0..n-1][0..n-1])   // n > 0
  S ← 0
  for i ← 0 to n-1 do
      for j← 0 to n-1 do
          S ← S + A[i][j];
  return S
```

- The inner loop:
  - j goes from 0 to n-1
  - At each iteration, we do one basic operation
  - So for the inner loop we have

$$\sum_{j=0}^{n-1} 1$$

  - We do this for each iteration of the outer loop

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

Simplifying the sum

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

- The inner summation is:

$$\sum_{j=0}^{n-1} 1 = 1 + 1 + \ \dots + 1 = n$$

- So the outer summation is:

$$\sum_{i=0}^{n-1} n = n + n + \ \dots + n = n^2$$

Example 3

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

- What does this algorithm do?

- What is the basic operation?

- How many times is the operation executed for input size n?

What does it do?

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

5	**2**	4	6	1	3
2	5	**4**	6	1	3
2	4	5	**6**	1	3
2	4	5	6	**1**	3
1	2	4	5	6	**3**
1	2	3	4	5	6

Basic operation

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

- Two options:
  - There are variable assignments and comparisons
  - Most people would say the basic operation is the key comparison *A[j]>v*
  - Why?
    - It is really the key thing being checked in each loop
    - “Data” comparisons are often considered more expensive than simple numerical comparisons or assignments

Example 3 analysis

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

- Look at outer loop first
- There is a variable *i* getting incremented from *1* up to *n-1*
- So we have: $\sum_{i=1}^{n-1} (something)$

Example 3 analysis

```
Loops(A[0..n-1])
  for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← v
```

- The inner loop:
  - j goes from i-1 down to 0
  - At each iteration, we do one basic operation
  - Mathematically, the number of steps is: $$\sum_{j=0}^{i-1} 1$$
- We do this for each iteration of the outer loop
- So the total number of basic operations is:

$$\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1$$

Simplifying the sum

- We know:

$$\sum_{j=0}^{i-1} 1 = i$$

- So:

$$\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i$$

- Which equals:

$$\frac{(n-1)n}{2}$$

(See math review for the formula this answer comes from … also in the textbook Appendix A.)

Basic operations: tie-breakers

1. Function calls (growing with N)
2. Function calls (constant time)
3. Key comparisons (comparing data)
4. Assignments (copying data)
5. Expression evaluations

- Arithmetic tie-breakers:
    1. Multiplication/division
    2. Addition/subtraction
- These are all more like *guidelines* than strict rules

Pause and Reflect

- SO FAR: we learned how to determine the *running time* aka the *efficiency* of an algorithm
  - Non-recursive algorithms only
  - Count the statements, or the basic operations
  - The result is a function of n (input size)

An algorithm for analyzing algorithms?

- Given an algorithm (as input):
- Decide on the basic operation
  - May require tie-breakers
- Count how many times the basic operation is executed
  - Set up summations
  - Simplify to an expression (function) that depends on N
  - This is the running time
- Determine big-O class of the running time function

Running times of algorithms are functions.

$$f(n) = n$$

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

$$f(n) = \frac{(n-1)n}{2}$$

There are LOTS of functions in the world.

8675309

$4n^2+10$

3^n

$\log_2 n$

$50n^3+20n+4$

$4n\log_2 n$

2^n-1

$1+\log_2 6$

$3\log_2 n+1$

$3737n$

$n(2n+1)$

$3\log_2 n+n$

$5!+3^2$

n^2+3n^3

$\log_2 n+9n!$

Comparing functions

- Some functions are bigger than others
- What does “bigger” mean?
- We need a formalized way to talk about this

From earlier:

1. Efficiency of an algorithm depends on **input size**
2. Efficiency of an algorithm also depends on **basic operation**
3. Efficiency can be expressed by **counting** the basic operation

This is the algorithm from “Example 3” in Part 1

```
Loops(A[0..n-1])
  for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← v
```

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} \approx \frac{n^2}{2}$$

Example 1

- Problem: find the largest element in a list
- Input size measure:
  - *Number of list items, i.e. n*
- Basic operation:
  - *If statement / comparison*

```
ALGORITHM MaxElement(A[0..n − 1])
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]
    return maxval
```

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1$$

Example 2

- Problem: Multiplication of two matrices
- Input size measure:
  - *Matrix* dimension *(elements per row/col)*
- Basic operation:
  - *Innermost expression and assignment*

ALGORITHM $MatrixMultiplication(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

    for $i \leftarrow 0$ **to** $n-1$ **do**

        for $j \leftarrow 0$ **to** $n-1$ **do**

            $C[i, j] \leftarrow 0.0$

            for $k \leftarrow 0$ **to** $n-1$ **do**

                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

    return C

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3$$

Example 3

- Problem: calculating an unusual sum
- Input size measure:
  - *Number n*
- Basic operation:
  - *Division & assignment on line 6*
    - (but note that div-by-2 is actually a super-fast op)

```
1. Example3(n)
2.   sum ← 0
3.   i ← n
4.   while i ≥ 1
5.       sum ← sum + 1
6.       i ← i/2
7.   return sum
```

$$C(n) = \log n$$

Example 4

- Problem: Searching for key in a list of n items
- Input size measure:
  - *Number of list items, i.e. n*
- Basic operation:
  - *Key comparison / while loop*

ALGORITHM $SequentialSearch(A[0..n-1], K)$

$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
  $i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

Depends on order of input

$C_{worst}(n) = n$
$C_{best}(n) = 1$

Worst case, average case, best case

- Worst case:
  - Most possible number of steps needed by an algorithm
- Average case:
  - Number of steps needed “on average”
- Best case:
  - Number of steps needed if you “get lucky” with a particular input

- Consider the problem of finding an element in an unsorted list

Which to use: best, worst, average?

- We will usually focus on worst-case analysis
  - Unless otherwise specified, you should always analyze the worst case

- There are many situations where best case = worst case
  - Example: find the *largest* element in an unsorted list

Running time/efficiency can be many different functions

- $C(n) = n(n\text{-}1)/2$
- $C(n) \approx 0.5n^2$
- $C(n) = log\ n + 5$
- $C(n) =$ n!

8675309

3^n $4n^2+10$ $\log_2 n$

$50n^3+20n+4$

$4n\log_2 n$

2^n-1 $1+\log_2 6$

$3\log_2 n+1$ $3737n$

$n(2n+1)$ $3\log_2 n+n$

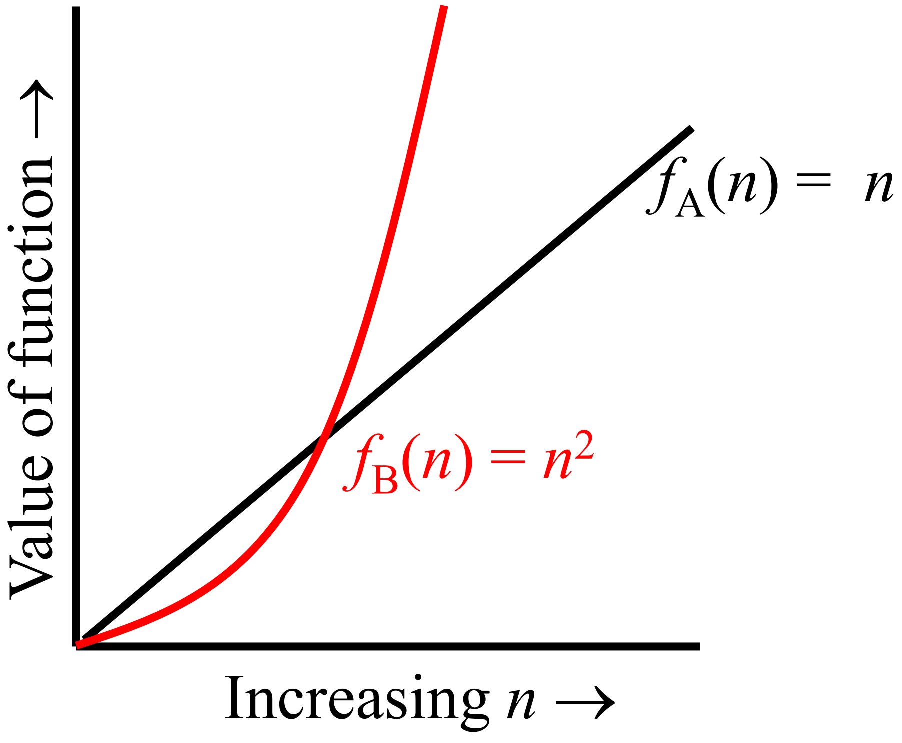
$5!+3^2$ n^2+3n^3 $\log_2 n+9n!$

- Which one is the better algorithm?

Let’s look at some functions

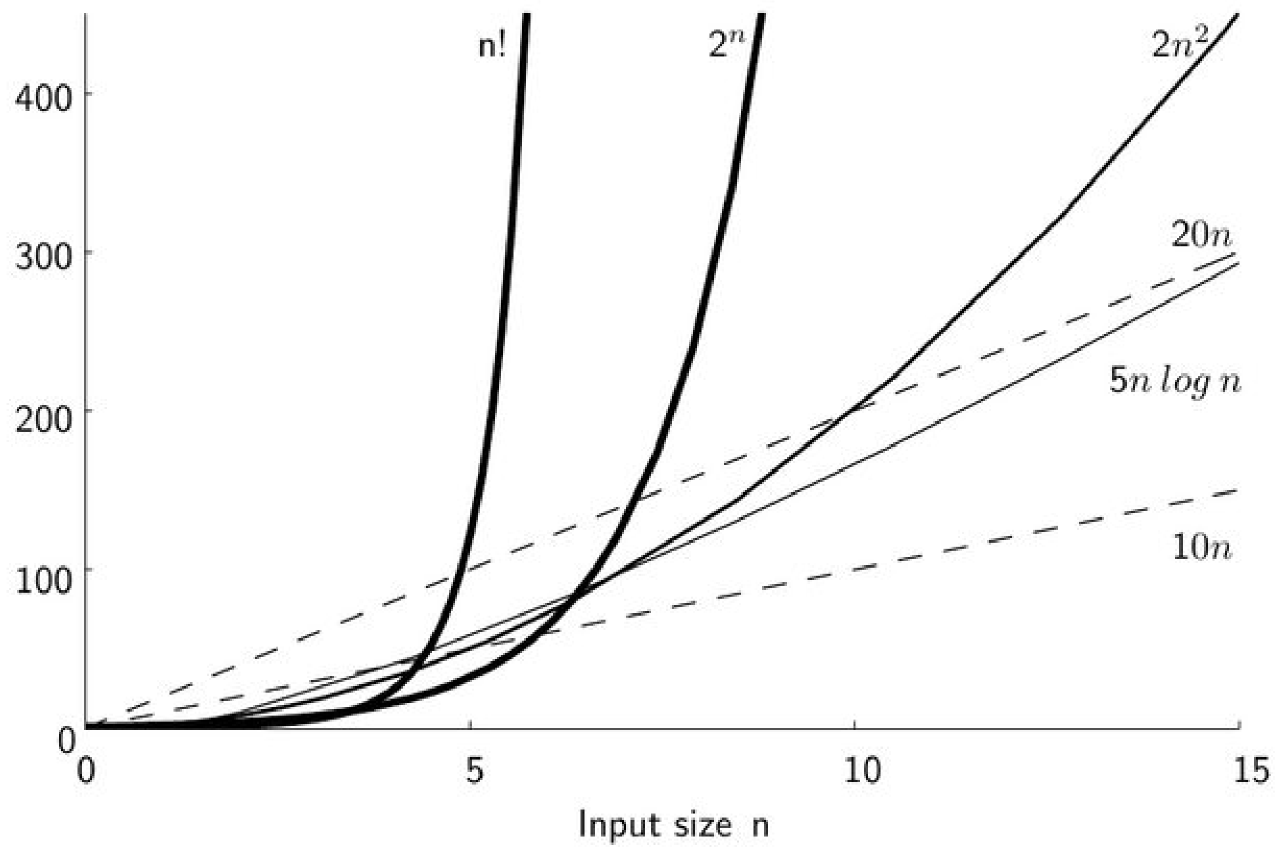
- DESMOS

Order of growth



Order of growth

- What we really care about:
  - Order of growth as $n \rightarrow \infty$



Orders of growth

these represent possible functions that classify basic ops counts

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$\boxed{9.3 \cdot 10^{157}}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

1.5×10^{133} years on the world's fastest supercomputer

Common efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.

Common efficiency classes (cont.)

n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

General strategy for analysis of non-recursive algorithms

From the textbook (p62):

1. Decide on a parameter indicating the input's size.
2. Identify the algorithm's basic operation.
3. Be sure the number of times the basic operation is executed depends only on the size of the input.
   - If it depends on some other property, the best/worst/average case efficiencies must be investigated separately
4. Set up a sum expressing the number of times the basic operation is executed.
5. Use summation algebra to find a closed-form expression for the sum from step 4 above.
6. Determine the efficiency class of the algorithm using asymptotic notations

Asymptotic order of growth

A way of comparing functions

- Big O (Pronounced “big oh”)
- Big Ω
- Big Θ

Some functions are essentially *the same*

8675309

$4n^2+10$

3^n

$\log_2 n$

$50n^3+20n+4$

$4n\log_2 n$

2^n-1

$1+\log_2 6$

$3\log_2 n+1$

$3737n$

$n(2n+1)$

$3\log_2 n+n$

$5!+3^2$

n^2+3n^3

$\log_2 n+9n!$

3^n

2^n-1

$50n^3+20n+4$
n^2+3n^3

$4n^2+10$
$n(2n+1)$

$\log_2 n+9n!$

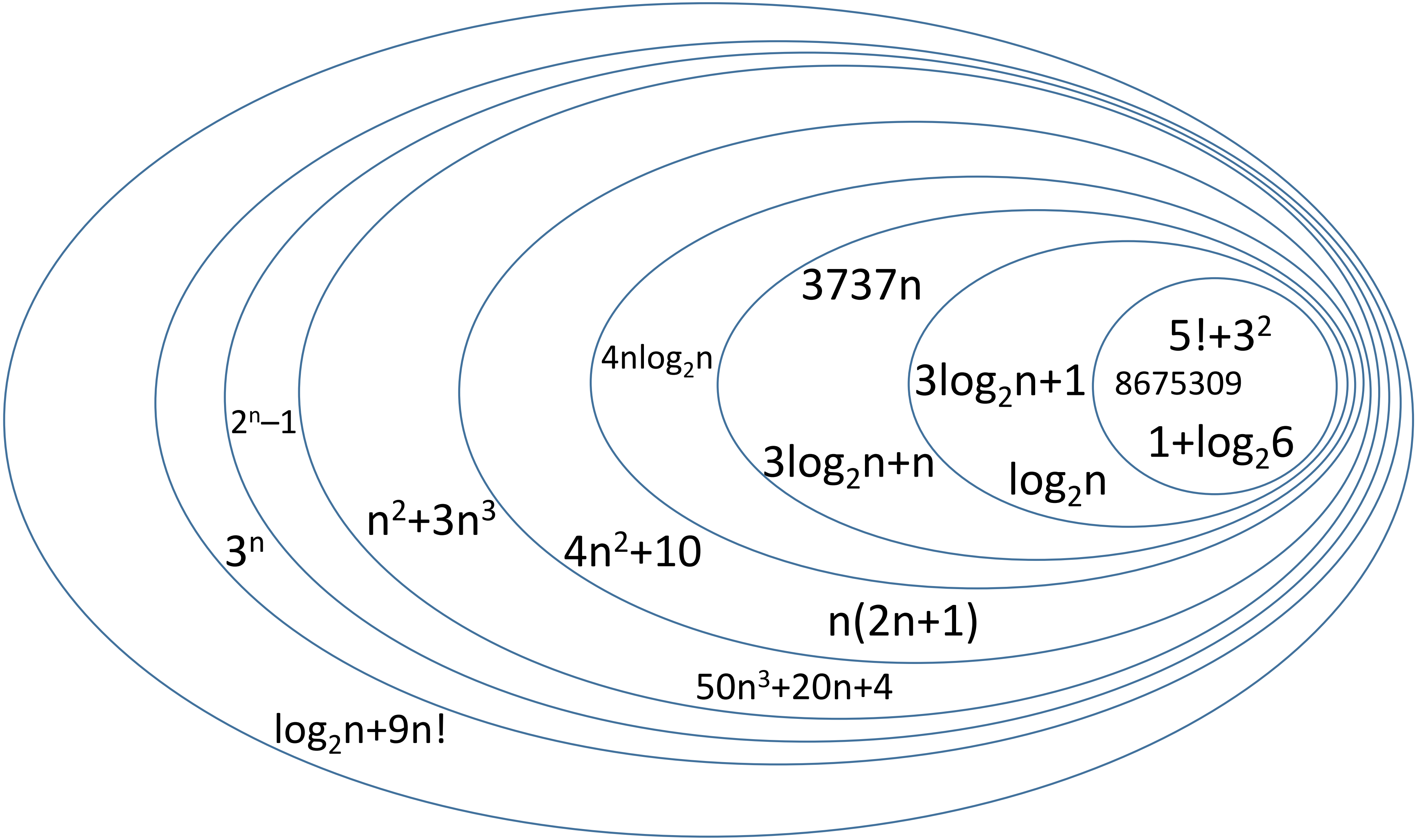
$5!+3^2$
8675309
$1+\log_2 6$

$4n\log_2 n$

$3\log_2 n+1$
$\log_2 n$

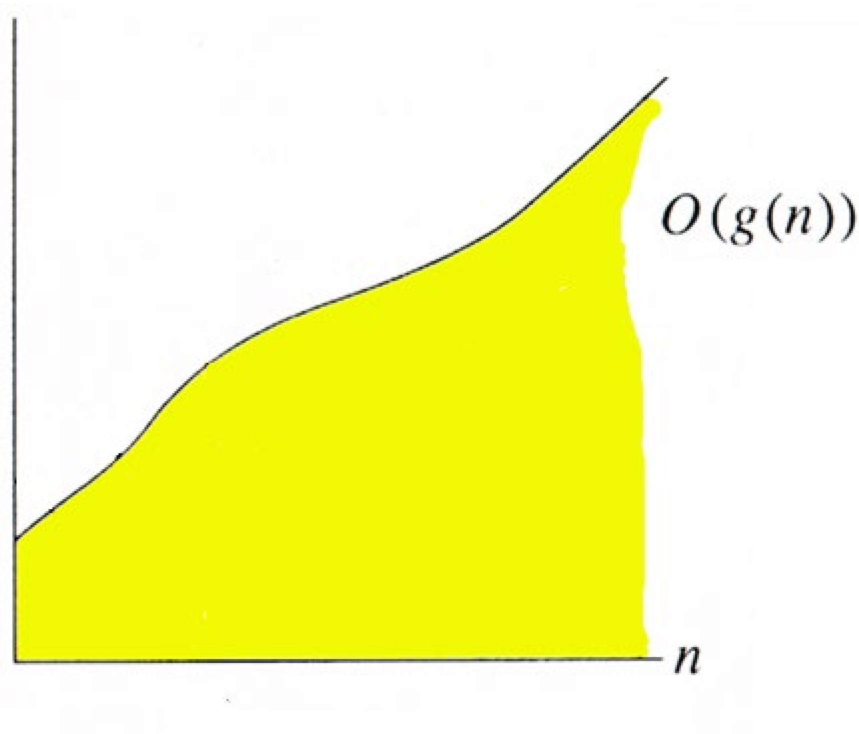
$3737n$
$3\log_2 n+n$

Even better



3737n
$5!+3^2$
4nlog$_2$n
$3\log_2 n+1$
8675309
2^n-1
$1+\log_2 6$
$3\log_2 n+n$
$\log_2 n$
n^2+3n^3
3^n
$4n^2+10$
n(2n+1)
$50n^3+20n+4$
$\log_2 n+9n!$

Big-O in pictures

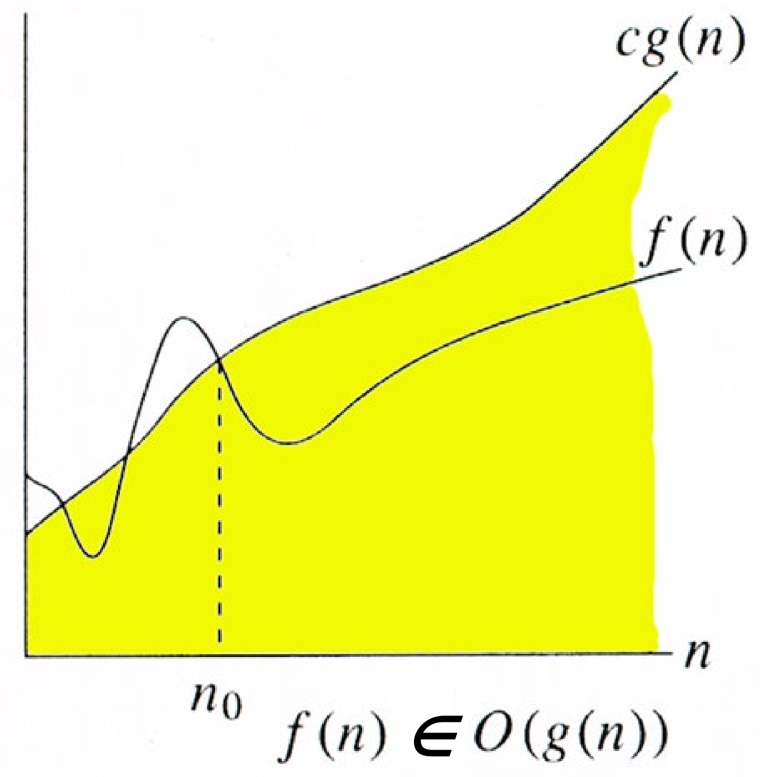


Set of all functions whose *rate of growth* is the same as or lower than that of *g*(*n*).

We also say “f(n) is *bounded above* by a constant multiple of g(n)”

or (carelessly) just “f(n) is bounded by g(n)”

Big-O in pictures



$f(n) \leq c * g(n)$, for all $n \geq n_0$

Big-O (formal definition)

<u>Definition:</u>

- a function **f(n)** is in the set **O(g(n))** [*denoted: f(n) ∈ O(g(n))*] if there is a constant **c** and a positive integer **n_0** such that

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

i.e. f(n) is bounded above by some constant multiple of g(n)

Example

- Is $f(n) = 2n+6 \in O(n)$?
- By the definition:
  - Need to find a constant c and a constant n_0 such that $f(n) \leq cg(n)$ for all $n > n_0$
- Many will work
  - Use $c = 4$ and $n_0 = 3$
- → $f(n)$ is $\in O(n)$

n	f(n)	c*g(n)
1	8	4
2	10	8
3	12	12
4	14	16
5	16	20
6	18	24
…	…	…

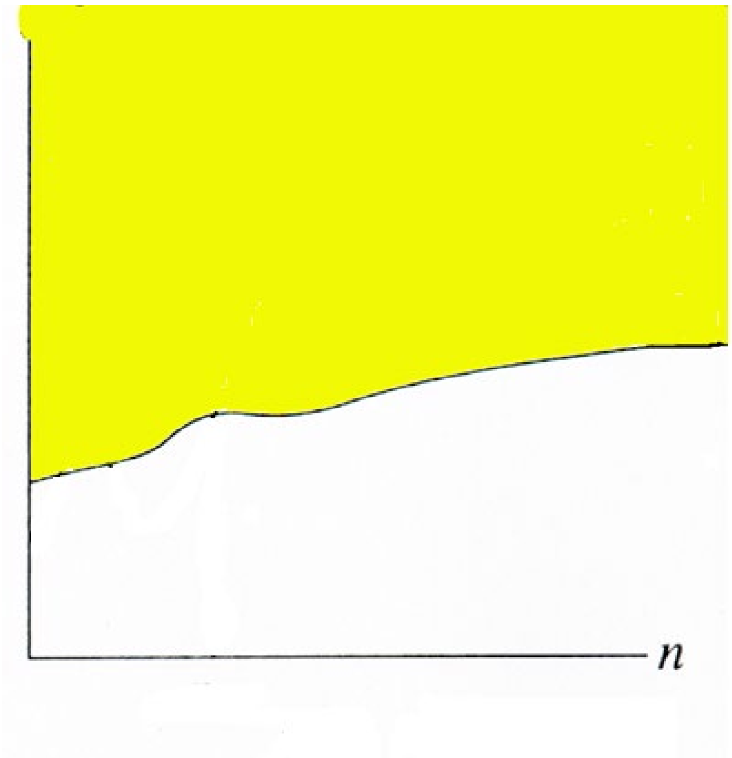
Looks good from here down

Big-O

- Simple Rule: Drop lower order terms and constant factors

1. $50n^3 + 20n + 4$ $\in O(n^3)$
2. $4n^2 + 10$ $\in O(n^2)$
3. $n(2n + 1)$ $\in O(n^2)$
4. $3\log n + 1$ $\in O(\log n)$
5. $3\log n + n$ $\in O(n)$
6. $1 + \log 6$ $\in O(1)$
7. $5! + 3^2$ $\in O(1)$

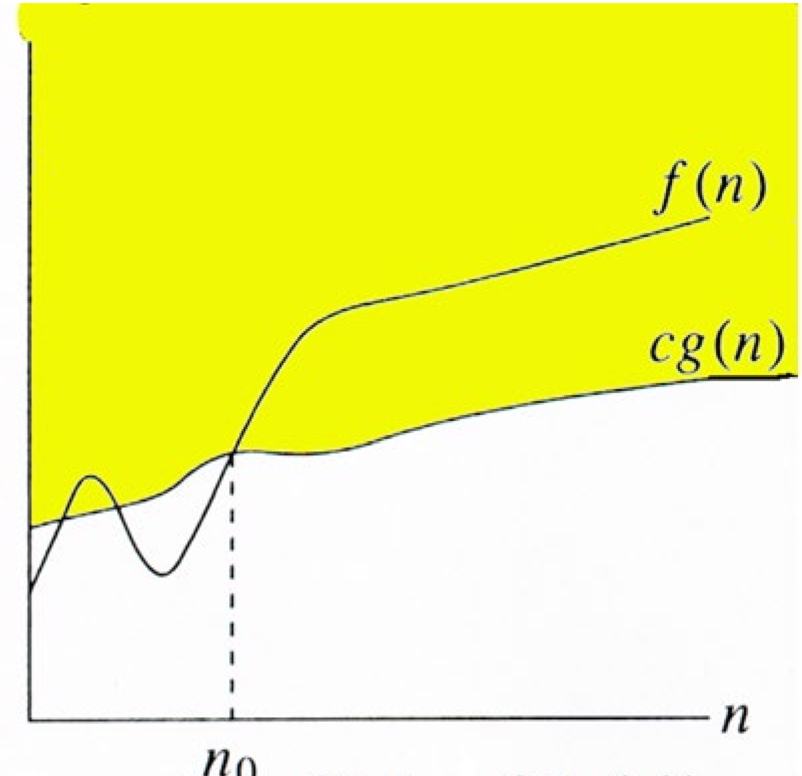
Big Omega



$\Omega(g(n))$

Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

Big Omega



$f(n) \in \Omega(g(n))$

$f(n) \geq c * g(n)$, for all $n \geq n_0$

Big Omega

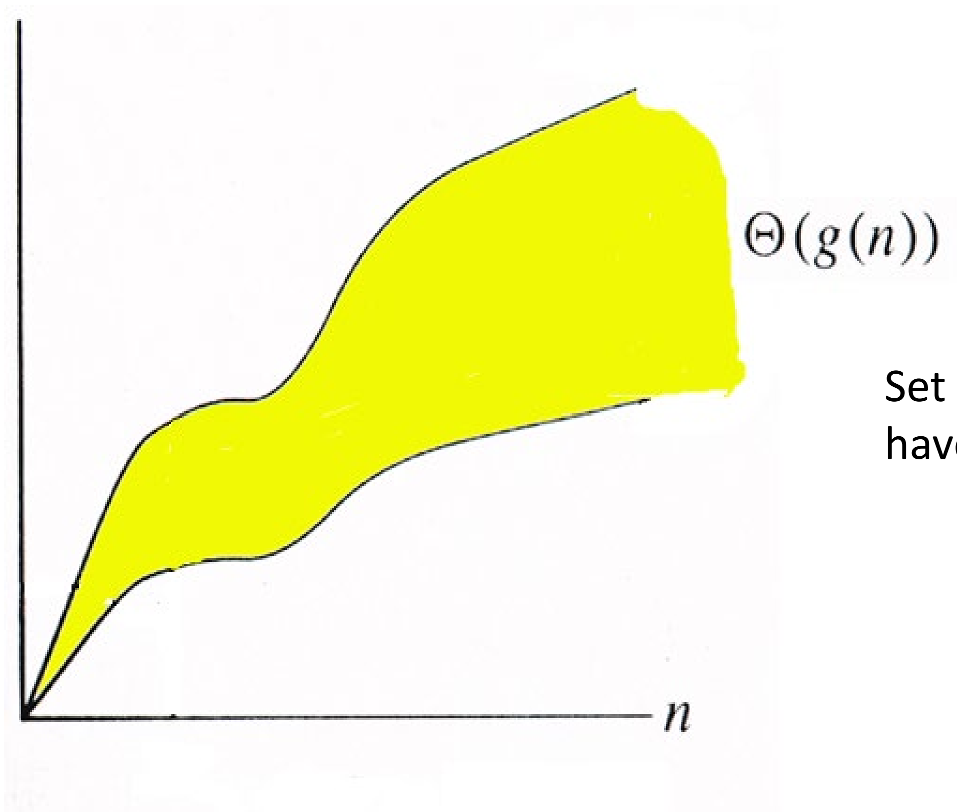
<u>Definition:</u>

- a function **f(n)** is in the set **Ω(g(n))** [*denoted: f(n) ∈ Ω(g(n))*] if there is a constant **c** and a positive integer $\mathbf{n_0}$ such that

$$\mathbf{f(n) \geq c * g(n)\ ,\ for\ all\ n \geq n_0}$$

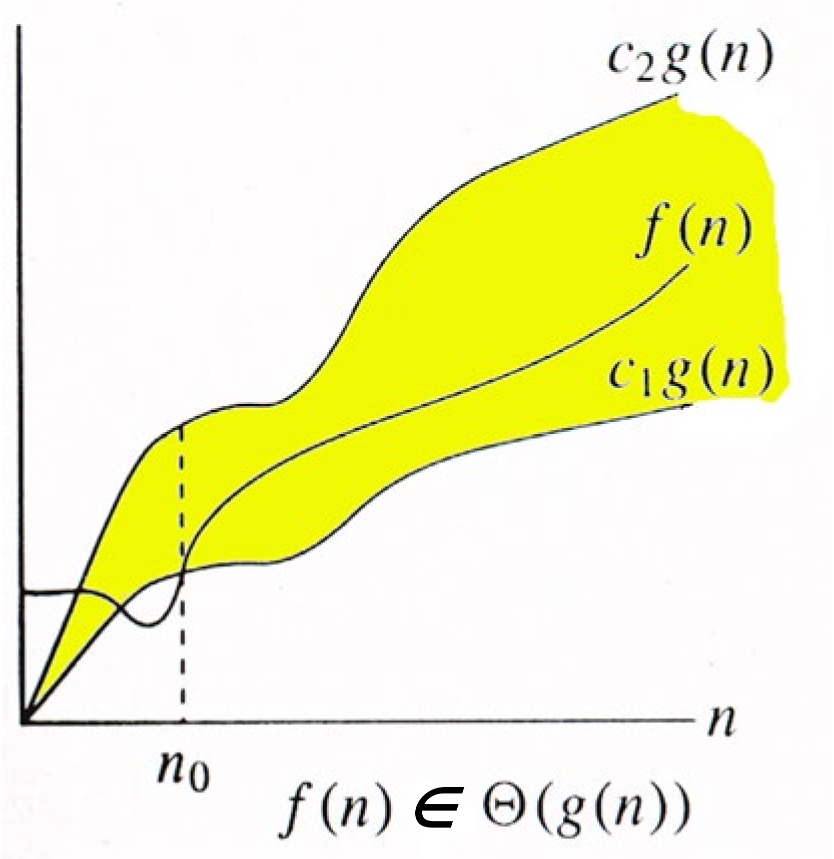
- *i.e.* f(n) is bounded below by some constant multiple of g(n)

Big Theta



Set of all functions that
have the same *rate of growth* as *g*(*n*).

Big Theta



$c_2\ g(n) \le f(n) \le c_1\ g(n)$, for all $n \ge n_0$

Big Theta

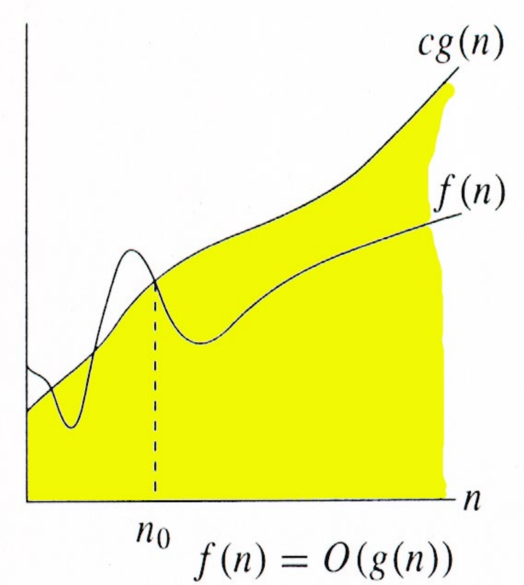
<u>Definition:</u>

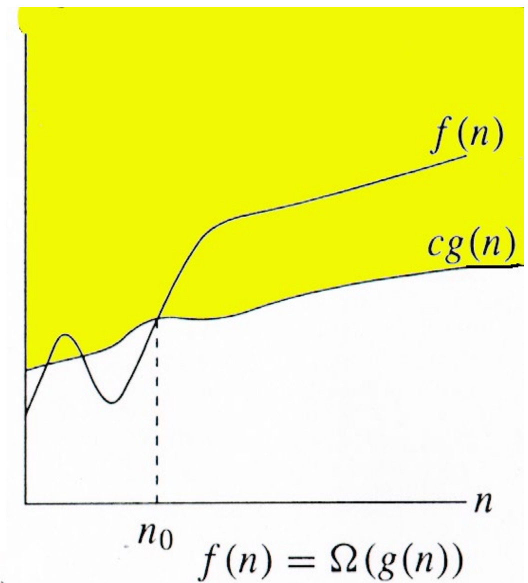
- a function **$f(n)$** is in the set **$\Theta(g(n))$** [*denoted: $f(n) \in \Theta(g(n))$*] if there are constants **c_1 and c_2**, and a positive integer **n_0** such that

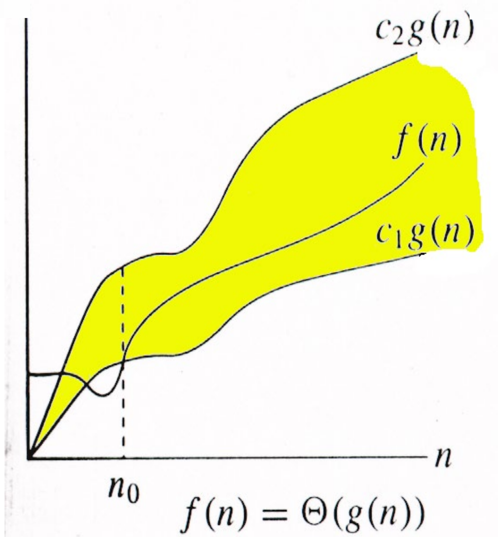
$$c_2\, g(n) \le f(n) \le c_1\, g(n)\ ,\ \text{for all } n \ge n_0$$

- *i.e.* f(n) is bounded both above and below by constant multiples of g(n)

Summary of notations - pictorial







Summary of notations - intuition

- Big-O → execution will take *at MOST that long*
- Big-Ω → execution will take *at LEAST that long*
- Big-Θ → execution *will take THAT long*

In general…

- We will usually focus on Big-O
- Why?
  - Focuses on worst case efficiency
  - Most common when people talk about algorithms

Examples

What is the efficiency class of the following functions?

- $10n$ O(n)
- $5n^2 + 20$ $O(n^2)$
- $10000n + 2^n$ $O(2^n)$
- log(n) * (1 + n) O(nlog(n))

Example 1

- Problem: find the max element in a list
- Input size measure:
  - *Number of list items, i.e. n*
- Basic operation:
  - *Comparison*

ALGORITHM *MaxElement*$(A[0..n-1])$

$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in O(n)$$

Example 2

- Problem: *Multiplication of two matrices*
- Input size measure:
  - *Matrix dimensions or total number of elements*
- Basic operation:
  - *Multiplication of two numbers*

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

    for $i \leftarrow 0$ **to** $n-1$ **do**

        for $j \leftarrow 0$ **to** $n-1$ **do**

            $C[i, j] \leftarrow 0.0$

            for $k \leftarrow 0$ **to** $n-1$ **do**

                $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

    return C

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = n^3 \in O(n^3)$$

Example 3: Element uniqueness problem

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

Example 3

ALGORITHM *UniqueElements*$(A[0..n-1])$
//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

- Parameter for input size:

  n, the size of the array

- Basic operation:

  Comparison in the innermost loop

- Worst case efficiency count… nested loop:

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i-1+1) = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = n(n-1) - (n-1) - (n-2)(n-1)/2$$

$$= n^2 - n - n + 1 - n^2/2 + 3n/2 - 1$$

$$= n^2/2 - n/2 \in O(n^2)$$

Practice problems

- Chapter 1.1 page 8, question 5
- Chapter 1.2 page 18, question 9
- Chapter 1.3 page 23, question 1
- Chapter 2.1, page 50, question 2
- Chapter 2.2, page 60, question 5
- Chapter 2.3, page 68, questions 5, 6

More practice problems

For each of the following problems, write an algorithm and then determine:

a. its basic operation
b. basic operation count
c. if basic op count depends on input form

1. Computing the sum of a set of numbers
2. Computing n! (n factorial)
3. Checking whether all elements in a given array are distinct