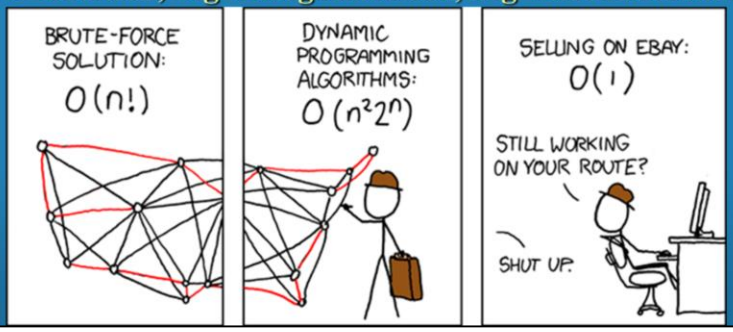
CS 395 – Analysis of Algorithms

Chapter 2 – Fundamentals of the Analysis of Algorithm Efficiency

Read 2.2

- Asymptotic Notations and Basic Efficiency Classes.
- Big-O notation, Big-Omega notation, Big-Theta notation



Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+ \cdots +1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n} i = 1+2+ \cdots +n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+ \cdots +n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

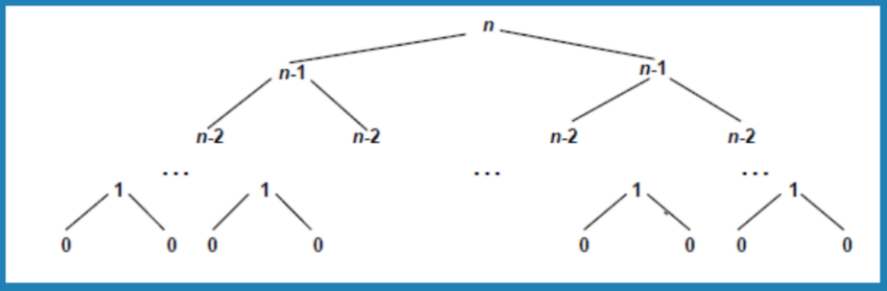
$\Sigma_{0 \le i \le n} a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i \qquad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Weekly Assignment 3

- Implement a recursive algorithm for computing 2^n for any nonnegative integer n that is based on the formula $2^n = 2^{(n-1)} + 2^{(n-1)}$.



3

How many times is this function called for input n?

Recall from Last Lecture

Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the positive reals.

Definition: "Big Oh" A function T(n) is in O(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \leq c * f(n)$.

Definition: "Omega" A function T(n) is in Ω(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \geq c * f(n)$.

Definition: "Theta" The set θ(g(n)) of functions consists of $\Omega(g(n)) \cap O(g(n))$.

Definition: "Big Oh" A function T(n) is in O(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \leq c * f(n)$.

If you need to prove f(n) is O(g(n)), it is sufficient to identify the n_0 and c so that the inequality holds.

- Choose a value for one of them (usually 1)
- Then find the minimum value for the other.

Try these yourself:

- $5x^2-3x+7 \in O(x^2)$
- $x^{20} \in O(x!)$
- $\log(x) \in O(x)$
- $f(x) \in O(f(x))$

Definitions

The following slides say exactly the same thing!

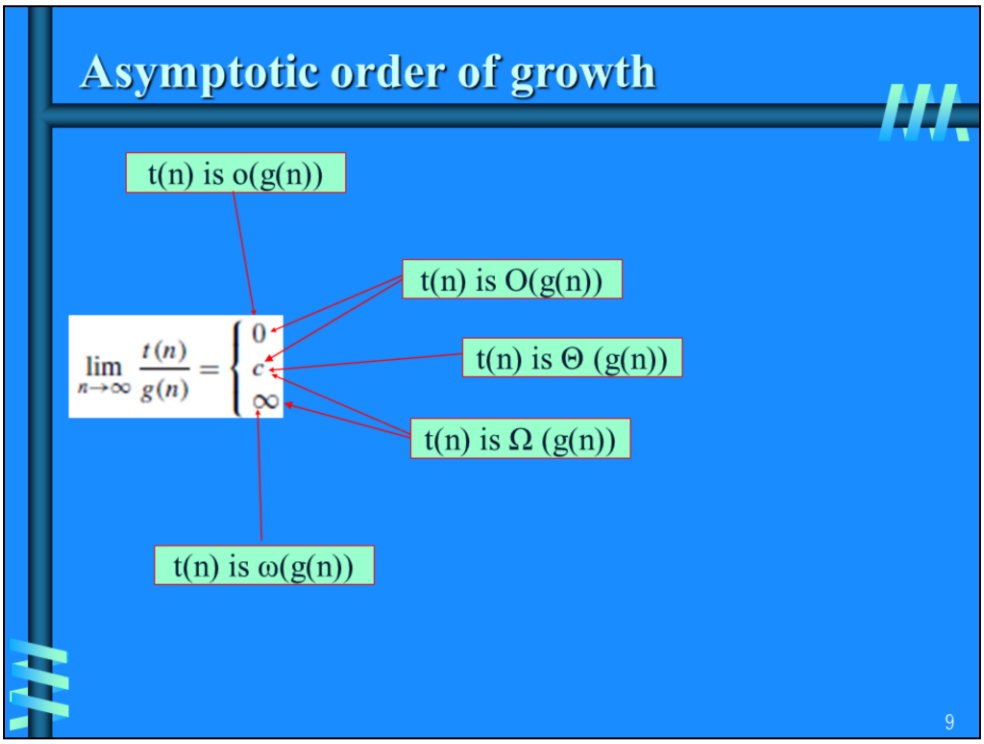
Find the one that makes sense to you and think of it that way

… but you need to know how to prove it both from the definition and from limits for the mid-term.

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 no faster than $g(n)$
- $o(g(n))$: class of functions $f(n)$ that grow slower than $g(n)$
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$
- $\omega(g(n))$: class of functions $f(n)$ that grow faster than $g(n)$



Limits:

0 -> Big O or C

C -> big Theta

Infinity -> Big Omega or C

Assume that f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the positive reals.

Definition: "Big Oh" A function f(n) is in O(g(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $f(n) \leq c * g(n)$.
Definition: "Little Oh" A function f(n) is o(g(n)) if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $0 \leq f(n) < c*g(n)$.

Definition: "Big Omega" A function f(n) is in $\Omega(g(n))$ if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $f(n) \geq c * g(n)$.
Definition: "Little Omega" A function f(n) is $\omega(g(n))$ if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $f(n) > c * g(n) \geq 0$ for every integer $n \geq n_0$.

Definition: "Theta" The set $\theta(g(n))$ of functions consists of $\Omega(g(n)) \cap O(g(n))$.

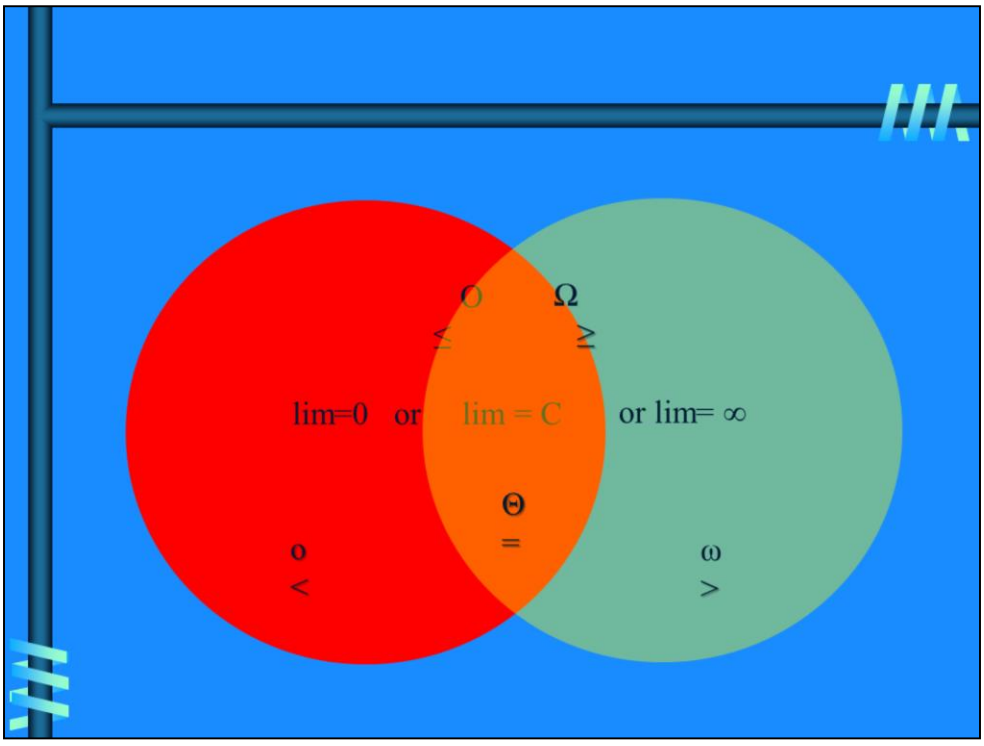
10

An isomorphic way of thinking of it:

- $O(g(n))$: $f(n) \leq g(n)$
- $o(g(n))$: $f(n) < g(n)$

- $\Theta(g(n))$: $f(n) \leq g(n)$ and $f(n) \geq g(n)$
  - therefore $f(n) = g(n)$

- $\Omega(g(n))$: $f(n) \geq g(n)$
- $\omega(g(n))$: $f(n) > g(n)$

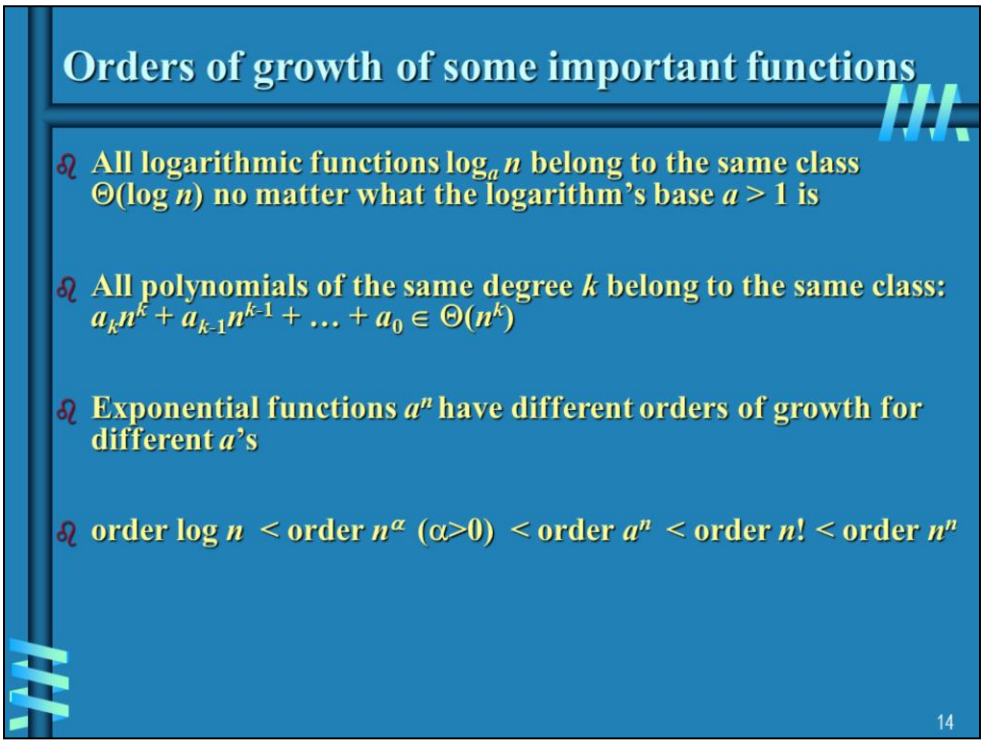

O
≤
Ω
≥
lim=0 or lim = C or lim= ∞
Θ
=
o
<
ω
>

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

  $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$



$\mathbf{Log}_b\ x = \mathbf{Log}_a\ x / \mathbf{Log}_a\ b$

Is $3x^2 + 2x + 4 \in \theta(x^2)$?

Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the positive reals.

Definition: "Big Oh" A function T(n) is in O(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, T(n) $\leq c * f(n)$

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies that } t(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } t(n) \text{ has the same order of growth as } g(n), \\ \infty & \text{implies that } t(n) \text{ has a larger order of growth than } g(n).^3 \end{cases}$$

exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, T(n) $\geq c * f(n)$.

Definition: "Theta" The set $\theta(g(n))$ of functions consists of $\Omega(g(n)) \cap O(g(n))$.

Now lets prove it using limits

15

Is $4n^{1.1} + 6\ n \log n \in O(n^{1.1})$?

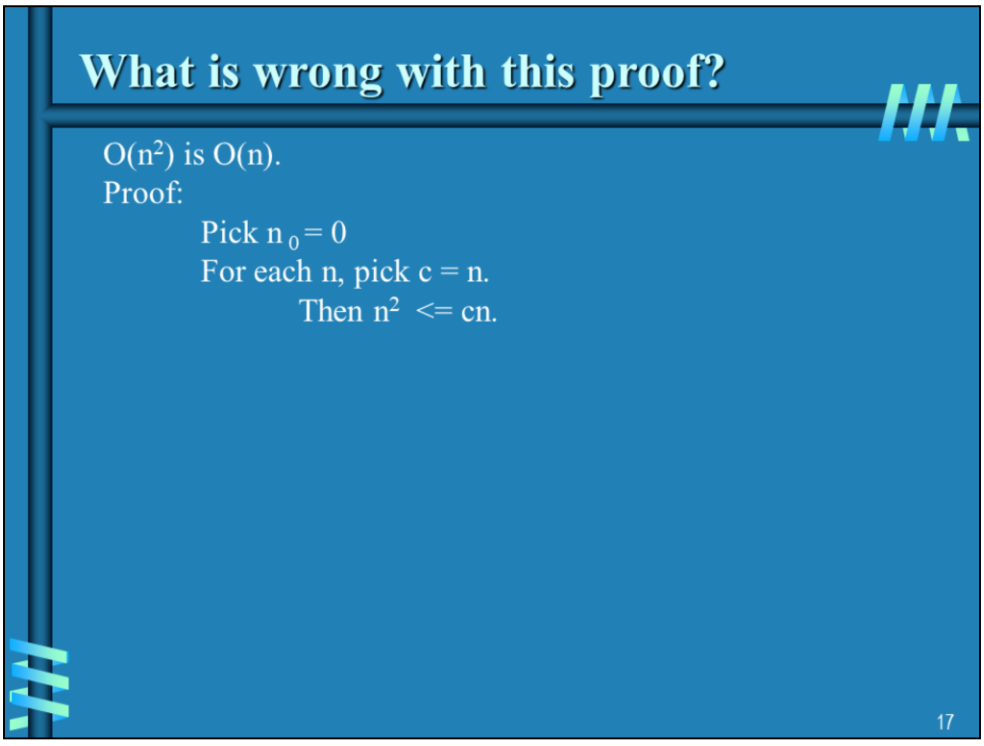
Assume that T, f and g are functions mapping the natural numbers {0, 1, 2, 3, ...} into the positive reals.

Definition: "Big Oh" A function T(n) is in O(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \leq c * f(n)$.

Definition: "Omega" A function T(n) is in Ω(f(n)) if there exist constants $n_0 \geq 0$, and $c > 0$, such that for all $n \geq n_0$, $T(n) \geq c * f(n)$.

Definition: "Theta" The set θ(g(n)) of functions consists of $\Omega(g(n)) \cap O(g(n))$.

16



Fallacious Arguments About Big-Oh

The definition of "big-oh" is tricky, in that it requires us, after examining T (n) and f(n), to pick witnesses n0 and c once and for all, and then to show that T (n) <= cf(n) for all n >= n0. It is not permitted to pick c and/or n0 anew for each value of n.

For example, one occasionally sees the following fallacious "proof" that n2 is O(n). "Pick n0 = 0, and for each n, pick c = n. Then n2 <= cn." This argument is invalid, because we are required to pick c once and for all, without knowing n.

Values of some important functions as $n \to \infty$

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	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
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}		

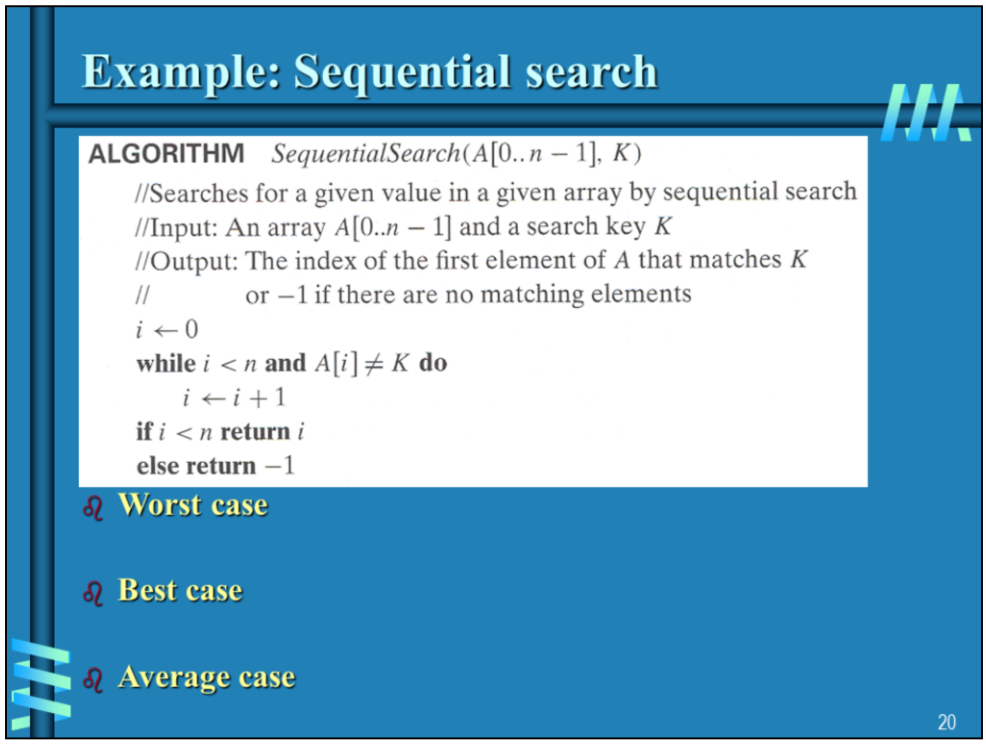
Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- **Worst case:** $C_{worst}(n)$ – maximum over inputs of size n
- **Best case:** $C_{best}(n)$ – minimum over inputs of size n
- **Average case:** $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

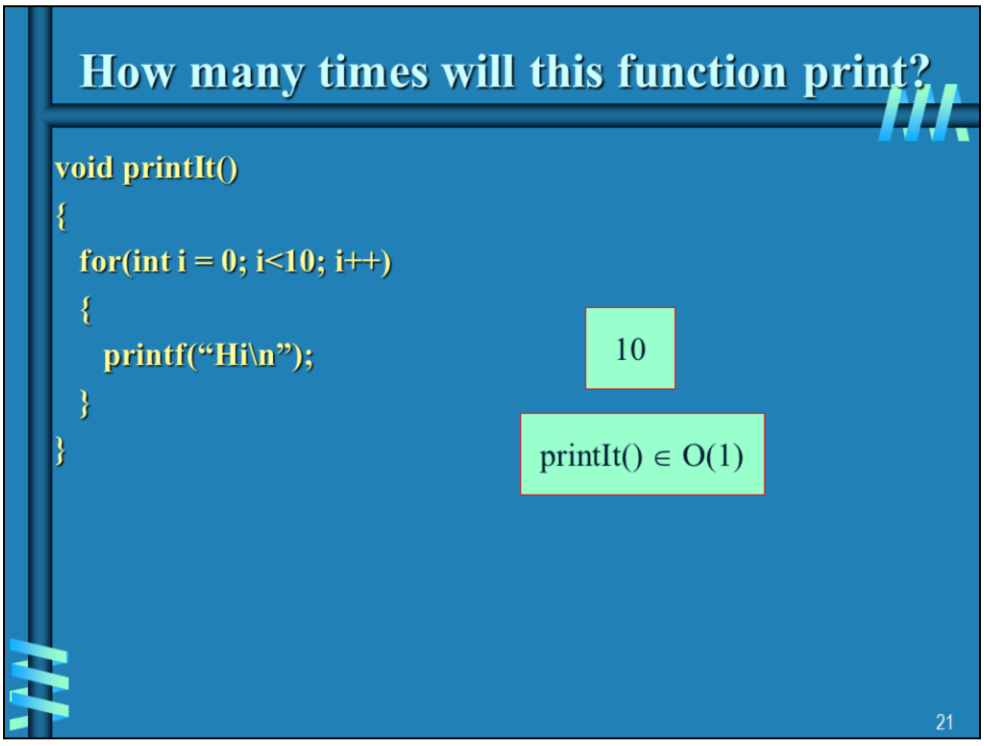
19



Worst: Not in the array = n

Best: First element = 1

Average: Mostly in the array=n/2. Mostly not found: n


How many times will this function print?
void printIt()
{
for(int i = 0; i<10; i++)
{
printf("Hi\n");
}
}
10
printIt() ∈ O(1)
21

10

How many times will this function print?

```
void printItN(int n)
{
  for(int i = 0; i<n; i++)
  {
    printf("Hi\n");
  }
}
```

printItN(int n) $\in$ O(n)

How many times will this function print?

```
void printItLots(int n)
{
  for(int i = 0; i<n; i++)
  {
    for(int j = 0; j<n; j++)
    {
      printf("Hi\n");
    }
  }
}
```

printItLots(int n) $\in O(n^2)$

How many times will this function print?

```
void printItMedium(int n)
{
  for(int i = 0; i<n; i++)
  {
    for(int j = 0; j<i; j++)
    {
      printf("Hi\n");
    }
  }
}
```

$0+1+2+\ldots+n-1 = n(n-1)/2$

printItMedium(int n) $\in O(n^2)$

What is the complexity of myProgram?

```
void main(int argc, char* argv[])
{
  int n = atoi(argv[1]);
  printIt();          O(1)
  printIt(n);         O(n)
  printItLots(n);     O(n^2)
  printItMedium(n);   O(n^2)
}
```

myProgram $\in O(2n^2 + n + 1)$

myProgram $\in O(n^2)$

25

Example: Sequential search

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

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//         or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

O(n)

What number am I thinking?

I am thinking of a number between 1 and 1000.
For each guess, I will tell you higher or lower.
Use the fewest number of guesses.

```
int numberGuesser(int lowerBound, int upperBound)
{
  int guess = (upperBound-lowerBound)/2 + lowerBound
  if(right(guess) {return(guess)}
  else if(lower(guess)) {return(numberGuesser(lowerBound, guess))}
  else {return(numberGuesser(guess, upperBound))}
}
```

numberGuesser() $\in$ O(logn)

27

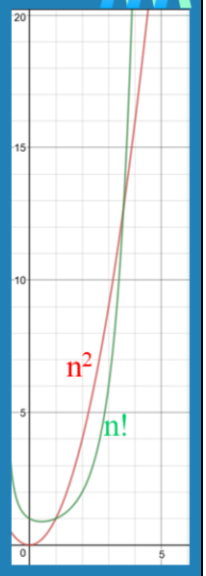
Is $n^2 \in O(n!)$ or is $n! \in O(n^2)$

n	n^2	n!
0	0	1
1	1	1
2	4	2
3	9	6
4	16	24
5	25	120
6	36	720

n! is off to a good start!

n^2 takes the lead

n! regains the lead.
Does it last?
lim $n^2/n!$



28

Could use Stirling's formula, but

n! >= n(n-1)(n-2) for n>=3

Values of some important functions as $n \to \infty$

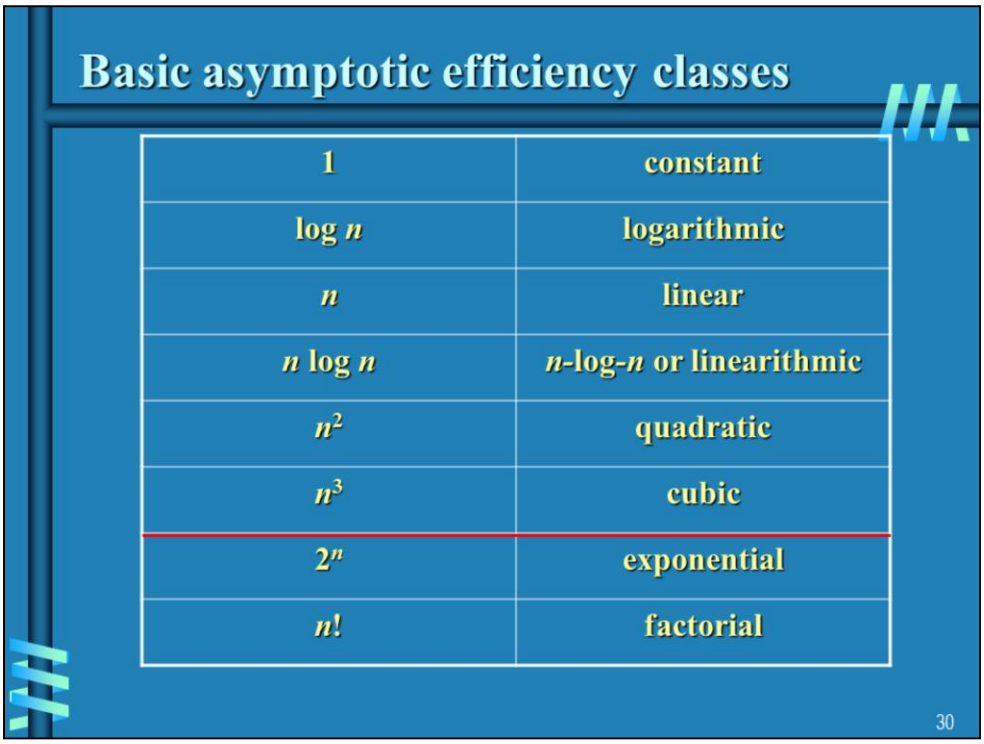
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$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

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

Think it through
Is $3^n \in O(2^n)$

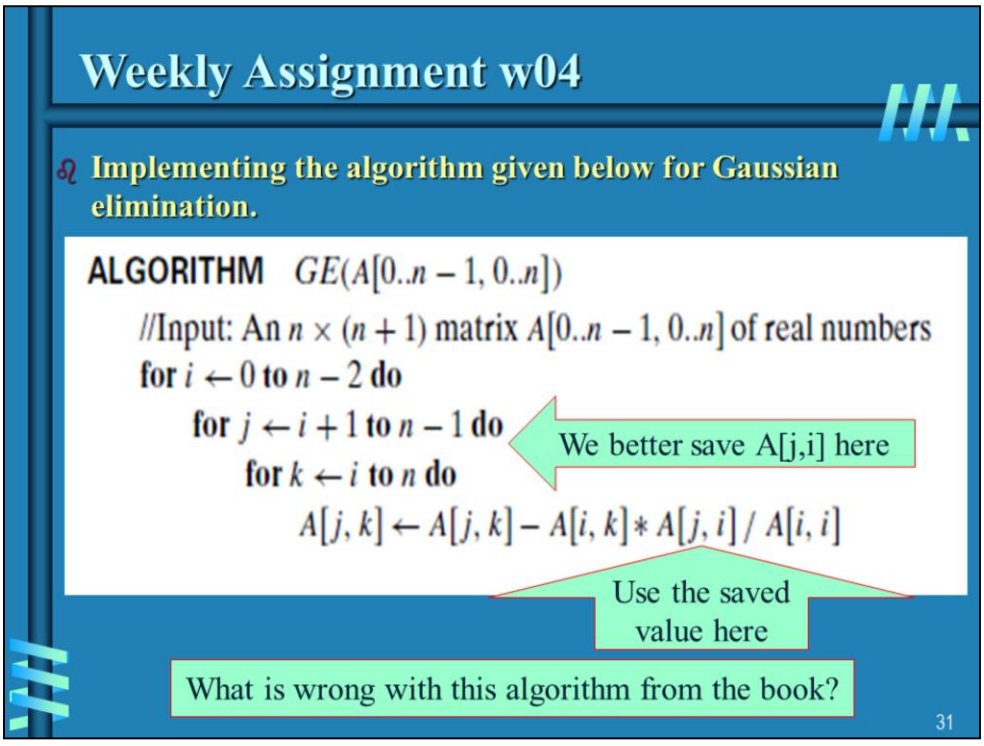
29

Find the limit of (2/3)^n



Below the red line the algorithms are NP hard.

NP stands for "nondeterministic polynomial time".



Limits:

0 -> Big O

C -> big Theta

Infinity -> Big Omega