Design and Analysis of Algorithms

National University of Singapore

CS3230

Week 2

Asymptotic Analysis

Warut Suksompong

Wordle

Goal: Guess a 5-letter word in as few turns as possible

Green = letter is in correct location
Yellow = letter appears in word but is in wrong location
Gray = letter does not appear in word

https://www.powerlanguage.co.uk/wordle/

Adversarial Wordle

ABSURDLE by qntm

This is an adversarial version of the excellent Wordle.

"Adversarial" means that Absurdle is actively trying to avoid giving you the answer. With each guess, Absurdle reveals as little information as possible, changing the secret word if need be. (Well, sort of: here is a detailed explanation.)

Other than that, the rules are the same as Wordle's, except you have unlimited guesses. You'll need them! The best possible score in Absurdle is 4 guesses.

https://qntm.org/files/wordle/index.html

Adversarial Wordle

Suppose your first guess is "TERNS".

Absurdle considers every word in its list of 2,315 possible secret words and figures out what its response would be in each case.

If the secret word was...	...then Absurdle's response would be...
"CIGAR"	⬜⬜🟨⬜⬜
"REBUT"	🟨🟩🟨⬜⬜
"SISSY"	⬜⬜⬜⬜🟨
"HUMPH"	⬜⬜⬜⬜⬜
...	...
"TERNS"	🟩🟩🟩🟩🟩

https://qntm.org/wordle

Adversarial Wordle

This has the effect of separating all of the 2,315 possible secret words up into 110 different *buckets* depending on the possible response:

Response	Secret words in this bucket	Number of secret words in this bucket
⬜⬜🟨⬜⬜	"CIGAR", "DWARF", "MAJOR", ...	136
🟨🟩🟨⬜⬜	"REBUT", "REACT", "RETCH", ...	12
⬜⬜⬜⬜🟨	"SISSY", "BLUSH", "BASIC", ...	119
⬜⬜⬜⬜⬜	"HUMPH", "FOCAL", "CLUCK", ...	256
...	...	...
🟩🟩🟩🟩🟩	"TERNS"	1

The smallest bucket is of course the 🟩🟩🟩🟩🟩 bucket, which has a single word in it, "TERNS". If Absurdle were feeling nice then it could simply announce, "Success! The word was 'TERNS'! You won in a single guess!" But Absurdle instead does the opposite, and selects the *largest* bucket, which is ⬜⬜⬜⬜⬜ and has 256 words in it. It

https://qntm.org/wordle

Introduction

What is an algorithm?

A finite sequence of "well-defined" instructions to solve a given computational problem

A prime objective of the course: Design of **efficient** algorithms

- Focus on running time

Example - Fibonacci Number F(n)

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n>1$

Problem 1: Given n, m, compute $F(n) mod\ m$

- Recursive algorithm
- Iterative algorithm

Two algorithms for Fibonacci numbers (mod m)

Recursive Algorithm

```
RFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1) + RFIB(n-2)) mod m);
}
```

Iterative Algorithm

```
IFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else {
                a ← 0;   b ← 1;
                For(i=2 to n) do
                {
                temp ← b;
                b ← (a+b) mod m;
                a ← temp; }
        }
        return b;}
```

How to analyze running time?

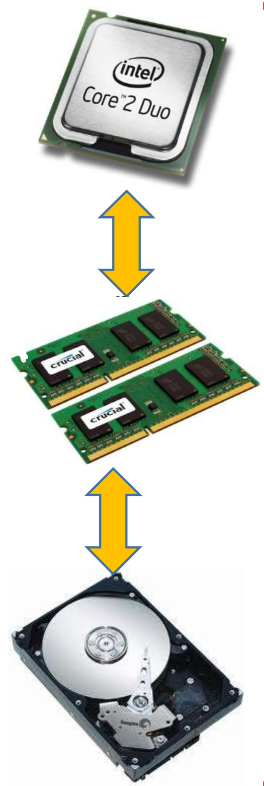
- **Simulation:** Run the algorithm many times and measure the running time

Machine dependent

Input dependent

- **Mathematical analysis**

Model of Computation



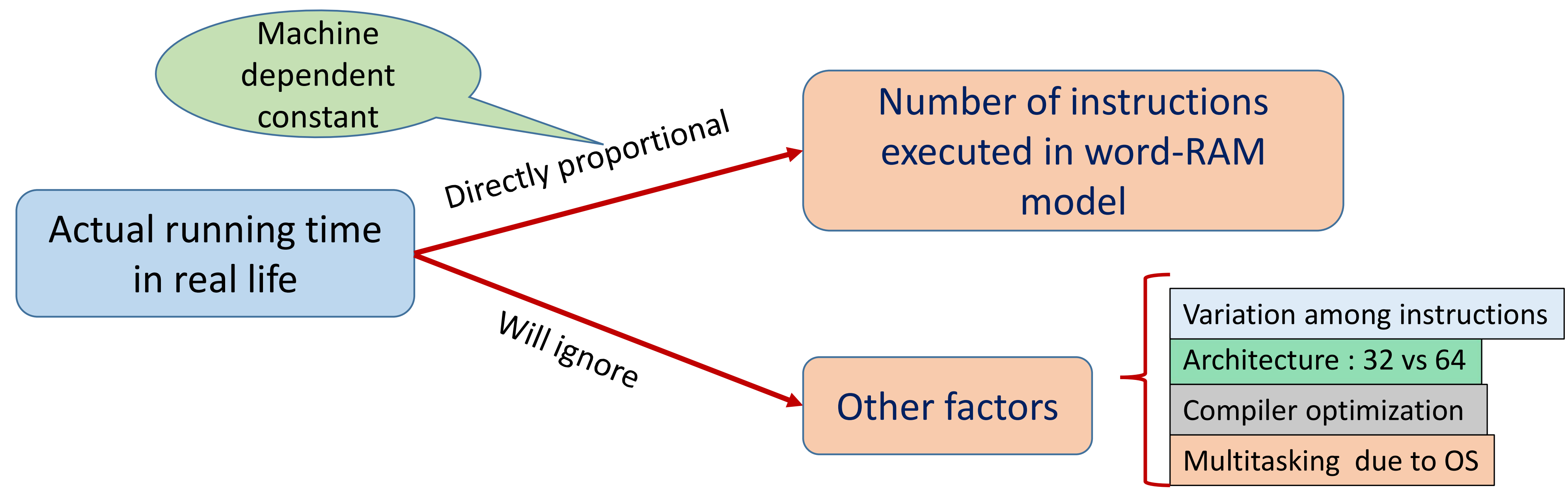
(Mathematical) Model of Computation

1. Simple
2. Close to real computers

How to measure Running Time?

This is what we will count (mathematically)

- Number of instructions taken in word-RAM model (+, -, x, /, mod, <, > , if, return, …)



Analyze algorithms for $F(n)$ mod m

First analyze the Recursive Algorithm

```
RFIB(n,m)
{
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1,m) + RFIB(n-2,m))
mod m);
}
```

Let $R(n)$ be the number of instructions executed by **RFIB**(n,m)

$R(0) = 2$

Analyze algorithms for F(n) mod m

First analyze the Recursive Algorithm

```
RFIB(n,m)
{
        if n=0 return 0;
        else if n=1 return 1;
       else return((RFIB(n-1,m) + RFIB(n-2,m))
mod m);
}
```

Let $R(n)$ be the number of instructions executed by **RFIB**(n,m)

$R(0) = 2$

$R(1) = 3$

Analyze algorithms for F(n) mod m

First analyze the Recursive Algorithm

```
RFIB(n,m)
{
        if n=0 return 0;
        else if n=1 return 1;
        else return((RFIB(n-1,m) + RFIB(n-2,m))
mod m);
}
```

No. of instructions $\geq \mathbf{2^{(n-2)/2}}$

Let R(n) be the number of instructions executed by **RFIB**(n,m)

R(0) = 2

R(1) = 3

R(n) = 7 + R(n-1) + R(n-2) for n>1

Exercise: Use induction to show, for all $\boldsymbol{n \geq 4}$

1. $R(n) \geq F(n)$
2. $F(n) \geq 2^{(n-2)/2}$

Analyze algorithms for F(n) mod m

Now analyze the Iterative Algorithm

No. of instructions =

```
IFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else {      a ← 0;   b ← 1;
              For(i=2 to n) do
              {     temp ← b;
                    b ← (a+b) mod m ;
                    a ← temp; }
            }
  return b;}
```

Analyze algorithms for F(n) mod m

Now analyze the Iterative Algorithm

No. of instructions $\leq 4 + 5(n-1) + 1$
$= 5n$

Worst-case time

```
IFIB(n,m) {
        if n=0 return 0;
        else if n=1 return 1;
        else {      a ← 0;   b← 1;
              For(i=2 to n) do
              {     temp ← b;
                    b← (a+b) mod m ;
                    a← temp; }
              }
    return b;}
```

4 instructions

n-1 iterations

5 instructions

The final instruction

Example: Checking a Number is Prime or not

```
IsPrime(k)
{
        For(i = 2 to k − 1) do
        {
                If(k%i = 0) return “Not Prime”;
        }
        return “Prime”;
}
```

Worst-case time=

Best-case time=

Example: Checking a Number is Prime or not

```
IsPrime(k)
{
    For(i = 2 to k − 1) do        ⟸ k-2 iterations
    {
        If(k%i = 0) return "Not Prime";   } 1 instruction
    }
    return "Prime";               ⟸ The final instruction
}
```

Worst-case time $\approx k$

Best-case time= 2

Comparing efficiency of two algorithms

Algorithm 1
$T(n) = 10n + 1000$

Algorithm 2
$T(n) = n^2 + 1000$

Which one is more efficient?

Comparing efficiency of two algorithms

Algorithm 1
$T(n) = 10n + 1000$

Algorithm 2
$T(n) = n^2 + 1000$

Which one is more efficient?

Algorithm 2 when $n < 10$
Algorithm 1 when $n > 10$

Time complexity
really matters only
for large-sized input

Comparing efficiency of two algorithms

Algorithm 1
$T(n) = 10n + 1000$

is more efficient than

Algorithm 2
$T(n) = n^2 + 1000$

Only compare for asymptotically large values of input size

Asymptotic analysis for running time

- Different machines have different running time.
- We do not measure *actual run-time*.
- We estimate the rate-of-growth of running time by asymptotic analysis.
  - Example: $0.01n^3$ grows faster than $1000n^2$!
- To compare running time of two different algorithms we need to see which is more efficient (or fast) for large inputs

Machine Independent

Asymptotic notations

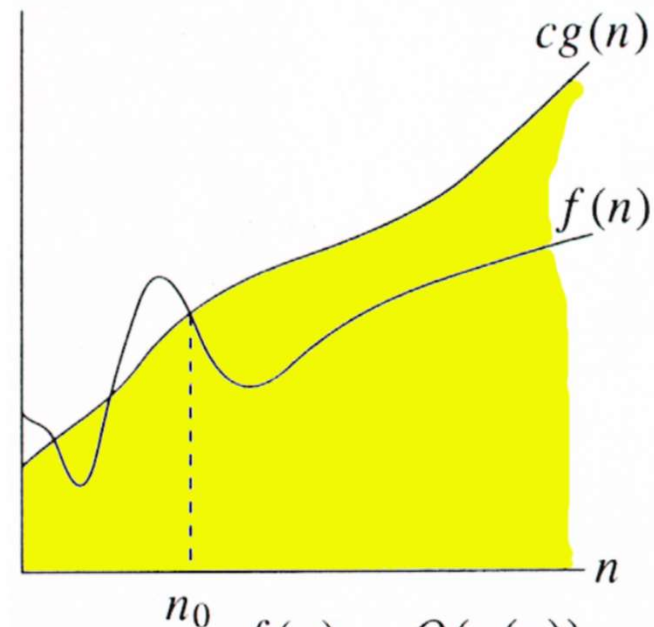
- O-notation (upper bound)
- Ω-notation (lower bound)
- Θ-notation (tight bound)

Formal definition: O-notation [upper bound]

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

Graphical explanation of O-notation

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.



$f(n) = O(g(n))$

O-notation is an *upper-bound* notation. It makes **no sense** to say $f(n)$ is <u>at least</u> $O(n^2)$.

Example

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- **Claim:** $2n^2 = O(n^3)$
- **Proof:** Let $f(n)=2n^2$.
  - Note that $f(n)=2n^2 \leq n^3$ when $n \geq 2$.
  - Set $c=1$ and $n_0=2$.
  - We have $f(n)=2n^2 \leq c \cdot n^3$ for $n \geq n_0$.
  - By definition $2n^2 = O(n^3)$.

Question 1

- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- We want to prove that $f(n) = O(g(n))$ by showing that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- What should be c and n_0? (There may be more than one correct answer.)

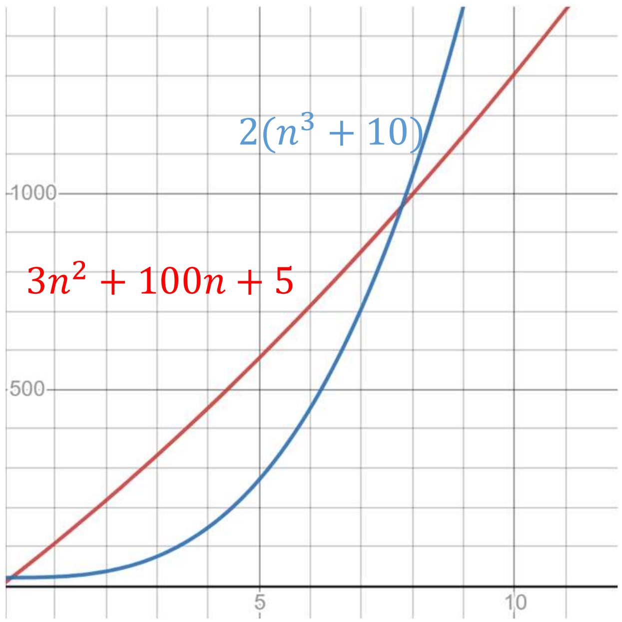
- (A) $c=2$, $n_0=10$
- (B) $c=1$, $n_0=12$
- (C) $c=5$, $n_0=2$
- (D) $c=1$, $n_0=10$

Answer of Question 1

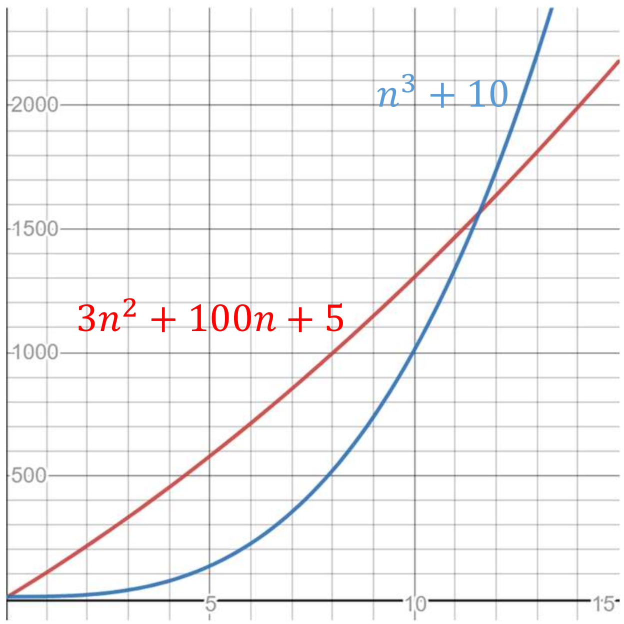
- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- (C) is false
  - When $c=5$, $n_0=2$,
  - $f(2)=3\cdot 2^2+100(2)+5=217$ and $g(2)=2^3+10=18$
  - Hence, $f(n) > c\ g(n)$ when $n=n_0=2$ and $c = 2$.
- (D) is false
  - When $c=1$, $n_0=10$,
  - $f(10)=3\cdot 10^2+100(10)+5=1305$ and $g(10)=10^3+10=1010$
  - Hence, $f(n) > c\ g(n)$ when $n=n_0=10$ and $c = 1$.

Answer of Question 1

- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- (A) is true



- (B) is true



Question 1

- Let $f(n)=3n^2+100n+5$.
- Let $g(n)=n^3+10$.
- We want to prove that $f(n) = O(g(n))$ by showing that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- What should be c and n_0? (There may be more than one correct answer.)

- (A) $c=2$, $n_0=10$
- (B) $c=1$, $n_0=12$
- (C) $c=5$, $n_0=2$
- (D) $c=1$, $n_0=10$

Set definition of O-notation

$$O(g(n)) = \{ f(n) : \text{there exist constants } c > 0,\ n_0 > 0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \}$$

- $O(g(n))$ is actually a set of functions.
- Although we write $f(n)=O(g(n))$, we mean $f(n) \in O(g(n))$

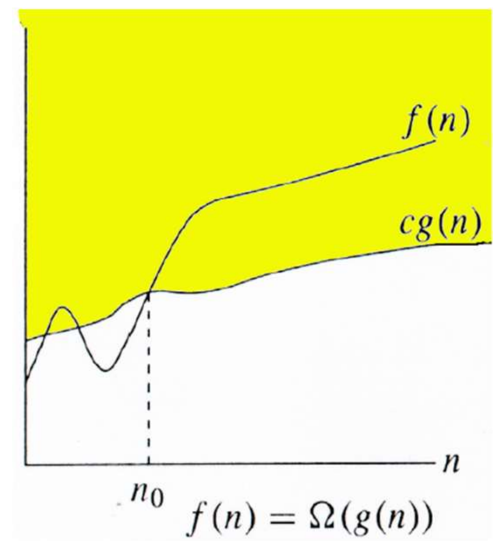
- Example, $2n^2 = O(n^3)$, $3n+4 = O(n^3)$, etc.

Formal definition: O-notation [upper bound]

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

Ω-notation (lower bound)

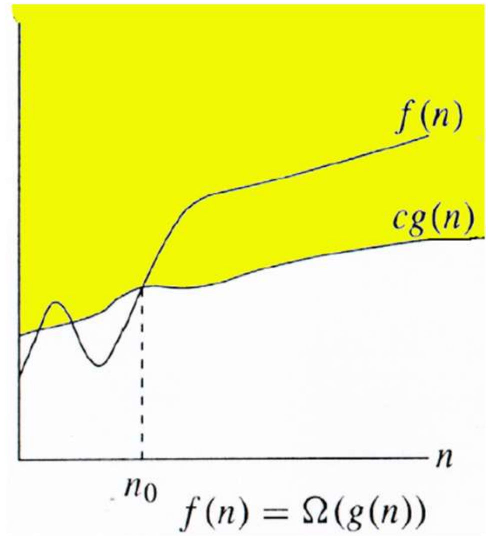
We write $f(n) = \Omega(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.



- **Example:** $n^2 - n = \Omega(n^2)$
  - $0 \leq \frac{1}{2}n^2 \leq (n^2 - n)$ for $n \geq 2$ (i.e. $c=1/2$, $n_0=2$)
  - Hence, $n^2 - n = \Omega(n^2)$

Ω-notation (lower bound)

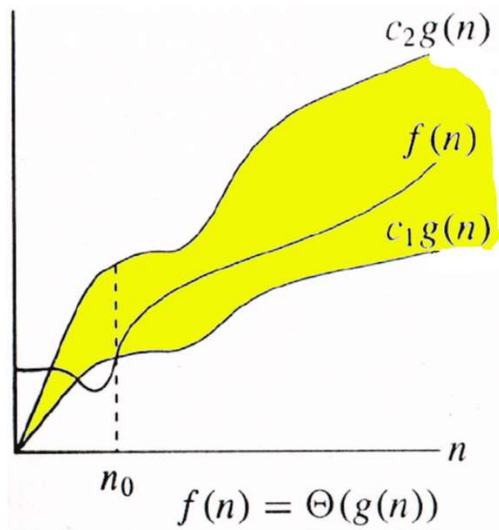
$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$ $\}$



- **Example:** $n^2 - n = \Omega(n^2)$
  - $0 \le \frac{1}{2}n^2 \le (n^2 - n)$ for $n \ge 2$ (i.e. $c=1/2$, $n_0=2$)
  - Hence, $n^2 - n = \Omega(n^2)$

Θ-notation (tight bound)

$\Theta(g(n)) = \{ f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ $\}$



- **Example:** $n^2 - n = \Theta(n^2)$
  - $0 \leq \frac{1}{2}n^2 \leq (n^2 - n) \leq n^2$ for $n \geq 2$ (i.e. $c_1=1/2$, $c_2=1$, $n_0=2$)
  - Hence, $n^2 - n = \Theta(n^2)$

O, Ω and Θ

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

Exercise: Prove the above using the definitions.

o-notation and ω-notation

O-notation and Ω-notation are like ≤ and ≥.
o-notation and ω-notation are like **<** and **>**.

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

- **Example:** $n = o(n^2)$
  - **E.g.,** $0 \le n < 2n^2$ for $n \ge 1$ (i.e., for $c=2$, $n_0=1$, similarly for any c we can find n_0)
  - Hence, $n = o(n^2)$
- However, $n^2 - n \ne o(n^2)$. Why?

o-notation and ω-notation

O-notation and Ω-notation are like ≤ and ≥.

o-notation and ω-notation are like **<** and **>**.

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

- **Example:** $n^2 - n = \omega(n)$
  - E.g., $0 \leq n < (n^2 - n)$ for $n \geq 3$ (i.e. $c=1$, $n_0=3$, similarly for any c we can find n_0)
  - Hence, $n^2 - n = \omega(n)$

Limit

- Assume f(n), g(n)>0.

- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f(n) = o\big(g(n)\big)$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = O\big(g(n)\big)$
- $0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = \Theta\big(g(n)\big)$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \Rightarrow f(n) = \Omega\big(g(n)\big)$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \Rightarrow f(n) = \omega\big(g(n)\big)$

$$\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 0 \rightarrow \; f(n) = o\big(g(n)\big)$$

- **Proof:**
  - Since $\lim_{n \to \infty} \left(\frac{f(n)}{g(n)} \right) = 0$, by definition, we have:
    - For all $\varepsilon > 0$, <u>there exists</u> $\delta > 0$ such that $\frac{f(n)}{g(n)} < \varepsilon$ for $n > \delta$.
  - Set $c = \varepsilon$ and $n_0 = \delta$. We have:
    - For all $c > 0$, there exists $n_0 > 0$ such that $\frac{f(n)}{g(n)} < c$ for $n > n_0$.
    - Hence, for all $c > 0$, there exists $n_0 > 0$ such that $f(n) < c \cdot g(n)$ for $n > n_0$.
    - By definition, $f(n) = o\big(g(n)\big)$.

o-notation and ω-notation

O-notation and Ω-notation are like ≤ and ≥.
o-notation and ω-notation are like **<** and **>**.

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

- **Example:** $n = o(n^2)$
  - $0 \leq n < 2n^2$ for $n \geq 1$ (i.e. $c=2$, $n_0=1$)
  - Hence, $n = o(n^2)$
- However, $n^2 - n \neq o(n^2)$. Why?

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \rightarrow \quad f(n) = o\big(g(n)\big)$$

- **Proof:**
  - Since $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$, by definition, we have:
    - For all $\varepsilon>0$, there exists $\delta>0$ such that $\frac{f(n)}{g(n)} < \varepsilon$ for $n > \delta$.
  - Set $c = \varepsilon$ and $n_0=\delta$. We have:
    - For all $c>0$, there exists $n_0>0$ such that $\frac{f(n)}{g(n)} < c$ for $n > n_0$.
    - Hence, for all $c>0$, there exists $n_0>0$ such that $f(n) < c \cdot g(n)$ for $n > n_0$.
    - By definition, $f(n) = o\big(g(n)\big)$.

Limit

- Assume f(n), g(n)>0.

- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0 \Rightarrow f(n) = o\big(g(n)\big)$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = O\big(g(n)\big)$
- $0 < \lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) < \infty \Rightarrow f(n) = \Theta\big(g(n)\big)$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) > 0 \Rightarrow f(n) = \Omega\big(g(n)\big)$
- $\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \infty \Rightarrow f(n) = \omega\big(g(n)\big)$

Similar to the first one

Example

- **Question:** By limit, show that $n^3 + 3n^2 + 4n + 1 = \omega(n^2)$.

- **Proof:**
  - $\lim_{n\to\infty}\left(\frac{n^3+3n^2+4n+1}{n^2}\right) = \lim_{n\to\infty}\left(n + 3 + \frac{4}{n} + \frac{1}{n^2}\right) = \infty$.
  - Hence, $n^3 + 3n^2 + 4n + 1 = \omega(n^2)$

Properties of big-O

- **Transitivity**

$$f(n) = \Theta(g(n)) \;\&\; g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$
$$f(n) = O(g(n)) \;\&\; g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$
$$f(n) = \Omega(g(n)) \;\&\; g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$$
$$f(n) = o\,(g(n)) \;\&\; g(n) = o\,(h(n)) \Rightarrow f(n) = o\,(h(n))$$
$$f(n) = \omega(g(n)) \;\&\; g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$$

- **Reflexivity**

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

Properties of big-O

- **Symmetry**

$$f(n) = \Theta(g(n)) \text{ iff } g(n) = \Theta(f(n))$$

- **Complementarity**

$$f(n) = O(g(n)) \text{ iff } g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \text{ iff } g(n) = \omega((f(n))$$

Example: Sorting

- ***Input:*** sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.
- ***Output:*** permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

- Example:
  - ***Input:*** 8 2 4 9 3 6
  - ***Output:*** 2 3 4 6 8 9

Insertion Sort

```
INSERTION-SORT(A[1..n])
for j = 2 to n
    key = A[j]
    // Insert A[j] into sorted seq A[1 .. j-1]
    i = j –1
    while i > 0 and A[i] > key
        A[i+1] = A[i]
        i = i –1
    A[i+1] = key
```

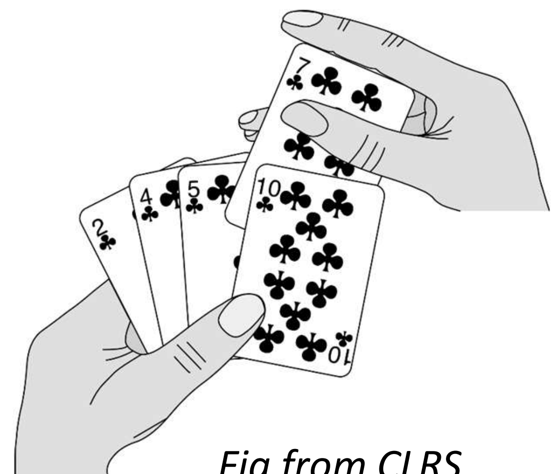


Fig from CLRS

Example: Step5 (while loop) of Insertion sort

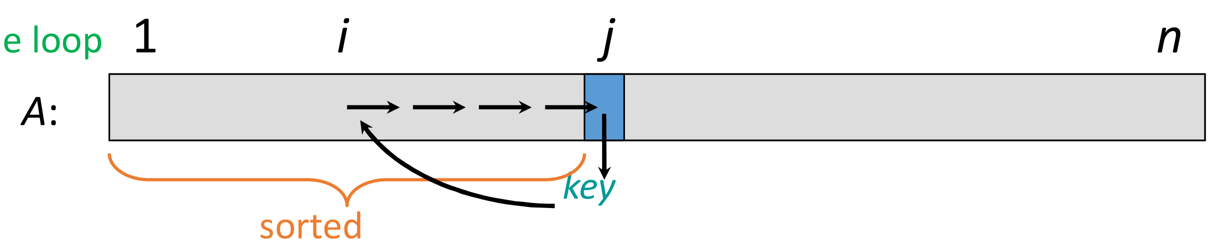
Suppose j=5.
Denote by A’ the array A immediately before the while loop (line 5)
Suppose A’=1, 4, 6, 9, 2, 7, 3 (i.e. key=A’[j]=A’[5]=2)

i		
4	1 4 6 9 2 7 3	0^{th} round of while loop
3	1 4 6 9 $\boxed{9}$ 7 3	1^{st} round of while loop
2	1 4 6 $\boxed{6\ 9}$ 7 3	2^{nd} round of while loop
1	1 4 $\boxed{4\ 6\ 9}$ 7 3	3^{rd} round of while loop

End of while loop

INSERTION-SORT(*A[1..n]*)

1. **for** *j* = 2 **to** *n*
2. *key* = *A*[*j*]
3. // Insert *A*[*j*] into sorted seq *A*[1 .. *j*-1]
4. *i* = *j* –1
5. **while** *i* > 0 and *A*[*i*] > *key*
6. *A*[*i*+1] = *A*[*i*]
7. *i* = *i* –1
8. *A*[*i*+1] = *key*



Insertion Sort running time

- The inner while loop is iterated at most $j - 1$ times
- Each while loop iteration has constant number of instructions
- Total runtime proportional to $\sum_{j=2}^{n}(j-1) < n^2$. Hence, the running time is $O(n^2)$.

- If the array is in reverse sorted order, then the runtime is also $\Omega(n^2)$.
- Hence, the worst-case runtime is $\Theta(n^2)$.

Acknowledgement

- The slides are modified from
  - the slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - the slides from Prof. Surender Baswana
  - the slides from Prof. Leong Hon Wai
  - the slides from Prof. Lee Wee Sun
  - the slides from Prof. Ken Sung
  - the slides from Prof. Diptarka Chakraborty
  - the slides from Prof. Arnab Bhattacharyya