Reasoning and asymptotic analysis

Lecture 1

Diptarka Chakraborty

Wing-Kin Sung

What is a good algorithm?

- Correct
  - The algorithm must be correct, including corner cases
- Efficient
  - Economical in use of time, space and resources
- Well-documented and with sufficient details
- Maintainable
  - Easy to understand, clear & concise (not tricky)
  - Easy to modify (if necessary)
  - Easily understood at different levels
  - Not computer-dependent
  - Usable as Modules by others

Correctness of an algorithm

- How do we reason if an algorithm is correct?
  - Depends on the types of algorithms
- Different types of algorithms:
  - Iterative algorithm
  - Recursive algorithm
  - Dynamic programming
  - Greedy algorithm
  - Randomized algorithm
  - Approximation algorithm
  - …

Correctness of iterative algorithm

Correctness of Iterative Algorithms

- Iterative algorithm is an algorithm which involves loop.

- The key step in the reasoning about the correctness of iterative algorithms is the invention of a condition called the ***loop invariant***, which is supposed to be
  - true at the beginning of an iteration, and
  - remains true at the beginning of the next iteration

- We will illustrate how to find loop invariant using insertion sort.

The problem of 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
```

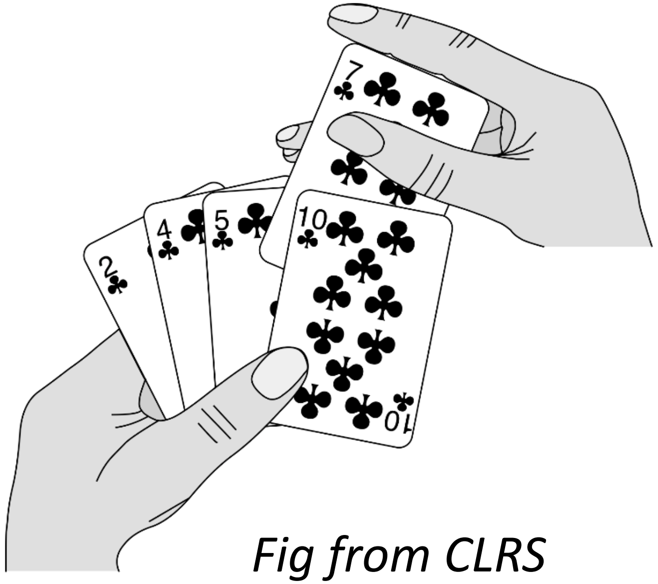


Fig from CLRS

Example: Step5 (while loop) of Insertion sort

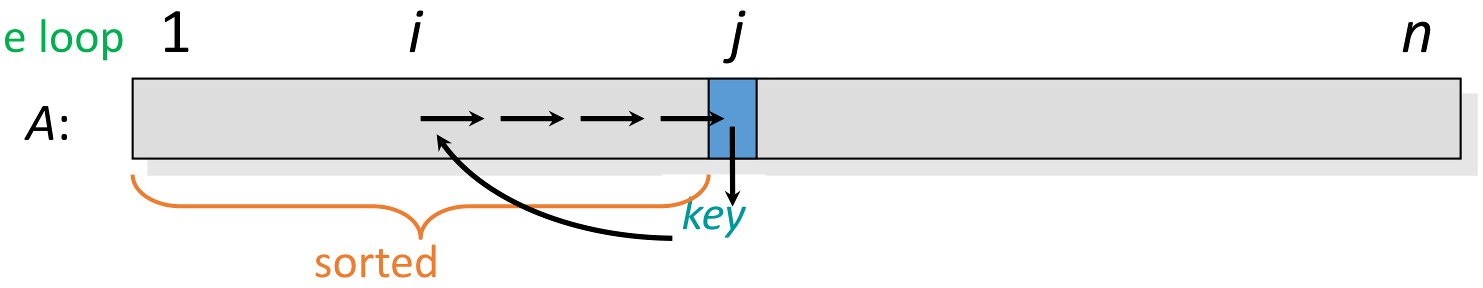
Suppose j=5.
Denote A’ be the array A immediately before the while loop (line 5)
Suppose A’=1, 4, 6, 9, 2, 7, 3 (i.e. key=A’[j]=A’[5]=2)

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

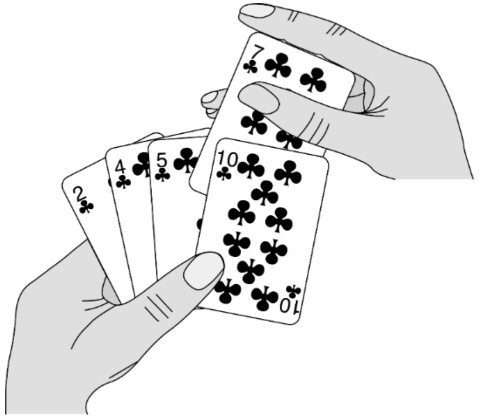
End of while loop

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*



Step 5 (while loop) of Insertion sort



Denote A’ be the array A immediately before the while loop (line 5)

We have the invariant:

1. ***A[1..i] = A’ [1..i]***

2. ***A[i+2..j] = A’ [i+1..j-1]***

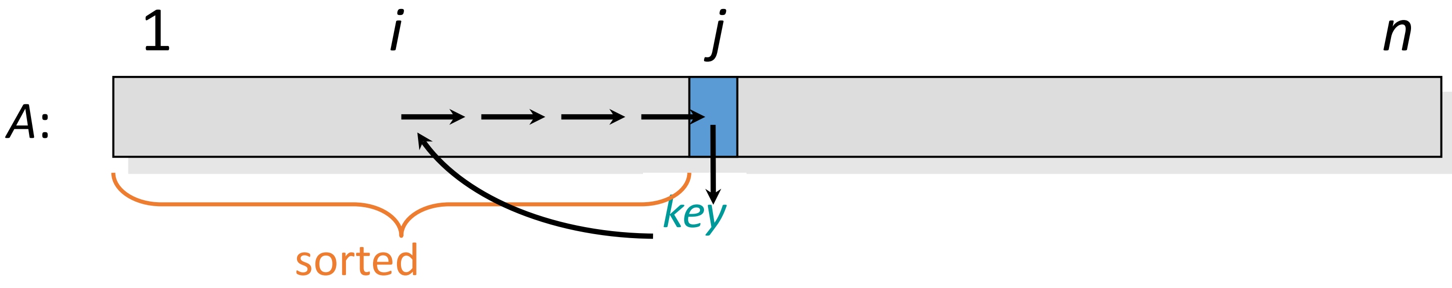
3. All elements in A[i+2..j] > key

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*

j=5
A’=1, 4, 6, 9, 2, 7, 3
(i.e. key=A’[j]=A’[5]=2)

i	A
4	1 4 6 9 2 7 3
3	1 4 6 9 9 7 3
2	1 4 6 6 9 7 3
1	1 4 4 6 9 7 3



Example: Step 1 (for loop) of Insertion srot

j		
	$\boxed{8}$ 2 4 9 3 6	0^{th} round of for loop
2	$\boxed{2\ 8}$ 4 9 3 6	1^{st} round of for loop
3	$\boxed{2\ 4\ 8}$ 9 3 6	2^{nd} round of for loop
4	$\boxed{2\ 4\ 8\ 9}$ 3 6	3^{rd} round of for loop
5	$\boxed{2\ 3\ 4\ 8\ 9}$ 6	4^{th} round of for loop
6	$\boxed{2\ 3\ 4\ 6\ 8\ 9}$	6^{th} round of for 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*

By inspection, the invariant is “A[1..j-1] is the sorted list of elements originally in A[1..j-1]”.

How to use invariant to show the correctness of an iterative algorithm?

To understand the correctness of an algorithm using an invariant, we need to show three things:

- **Initialization:** The invariant is true before the first iteration of the loop
- **Maintenance:** If the invariant is true before an iteration, it remains true before the next iteration
- **Termination:** When the algorithm terminates, the invariant provides a useful property for showing correctness.

Invariant: the subarray A[1 .. j-1] consists of the elements originally in A[1 .. j-1], but in sorted order

- **Initialization:** Before the start of the first iteration, *j* has been initialized to 2. The subarray *A*[1 .. *j*-1] is just *A*[1], which is trivially sorted.

```
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
```

Invariant: the subarray A[1 .. j-1] consists of the elements originally in A[1 .. j-1], but in sorted order

- **Maintenance:** (Sketch) By the property of the invariant, *A*[1 .. *j*-1] is sorted.
  - Line 2 assigns *A*[*j*] to *key*.
  - The **while** loop ensures that all array entries in *A*[1 .. *j*-1] larger than *key* is shifted one place to the right.
  - Line 8 assigns *key* to location created by shifts.
  - Then, A[1..j] is sorted!

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
```

Inv: *A*[*i*+2..*j*] sorted and > *key*.
A[1..*i*] = *A*′[1..*i*] and *A*[*i*+2..*j*] = *A*′[*i*+1..*j*-1]
where *A*′ is the array before the start of loop.

Invariant: the subarray A[1 .. j-1] consists of the elements originally in A[1 .. j-1], but in sorted order

- **Termination:** Array length is *n* and after the final loop, *j* is incremented to *n*+1. From the invariant, we have *A*[1 .. *j*-1] being sorted. Substituting *j*, the whole array is sorted.

```
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
```

Invariant of Iterative Algorithm

- Recap:
  - Invariant is a condition that is true at the beginning of every iteration
  - To show an invariant is true, we need to show that the invariant is true at initialization, is correctly maintained, and implies correctness with termination condition.

Initialization: $A[1..j\text{-}1]$ is empty, so invariant true.

Maintenance: Need invariant for inner loop stating that $A[smallest]$ is the smallest element in $A[j\ ..\ i\text{-}1]$.

When inner loop terminate, $i == n + 1$, so $A[smallest]$ is the smallest element in $A[j\ ..\ n]$.

If outer invariant is true before loop, it will be true after swapping on last line and incrementing j.

Correctness of recursive algorithm

Binary Search

Problem: Determine whether a number x is present in a *sorted* array $A[1\,.\,.\,N]$

```
BINARY-SEARCH (A, a, b, x)
if  a > b then
    return false
else
    mid = ⌊(a+b)/2⌋
if x == A[mid] then
    return true
if x < A[mid] then
    return BINARY-SEARCH (A, a, mid-1, x)
else
    return BINARY-SEARCH (A, mid+1, b, x)
```

Correctness of Recursive Algorithm

```
BINARY-SEARCH (A, a, b, x) ▻ A[a . . b]
if    a > b then
   return false
else mid = ⌊(a+b)/2⌋
if x == A[mid] then
   return true
if x < A[mid] then
   return BINARY-SEARCH (A, a, mid-1, x)
else
   return BINARY-SEARCH (A, mid+1, b, x)
```

- Usually use **mathematical induction** on size of problem
- P(n): Binary-search(A, a, b, x) return correct answer when b-a+1=n.

- **Base case:** array size $n = b - a + 1 = 0$
- Since $a = b + 1$, the subarray A[a..b] is empty!
- The test $a > b$ succeeds and the algorithm correctly returns false

Correctness of Recursive Algorithm

- **Inductive step**: array size $n = b - a + 1 > 0$
- By strong induction, we assume **BINARY-SEARCH** (A, a', b', x) returns the correct value for all j such that $0 \leq j \leq n\text{-}1$ where $j = b' - a' + 1$.
- As a≤b, the algorithm first calculates $mid = \lfloor (a+b)/2 \rfloor$, thus $a \leq mid \leq b$.
- If $x == A[mid]$, clearly $x \in A[a..b]$ and the algorithm correctly returns true.

```
BINARY-SEARCH (A, a, b, x) ⊳ A[a . . b]
if   a > b then
    return false
else mid = ⌊(a+b)/2⌋
if x == A[mid] then
    return true
if x < A[mid] then
    return BINARY-SEARCH (A, a, mid-1, x)
else
    return BINARY-SEARCH (A, mid+1, b, x)
```

Correctness of Recursive Algorithm

- If $x < A[mid]$, x is in $A[a..b]$ if and only if $x \in A[a..mid\text{-}1]$.
- By the *inductive hypothesis*, **BINARY-SEARCH** $(A, a, mid\text{-}1, x)$ will return the correct value since $0 \leq (mid\text{-}1) - a + 1 \leq n\text{-}1$.
- The case $x > A[mid]$ is similar.
- Hence, Binary-search(A, a, b, x) always returns correct answer!

```
BINARY-SEARCH (A, a, b, x) ⊳ A[a . . b]
if    a > b then
   return false
else mid = ⌊(a+b)/2⌋
if x = A[mid] then
   return true
if x < A[mid] then
   return BINARY-SEARCH (A, a, mid-1, x)
else
   return BINARY-SEARCH (A, mid+1, b, x)
```

Reasoning About Recursive Algorithm

Recap

- Use strong induction
- Prove base cases
- Show algorithm works correctly assuming algorithm works correctly for all smaller cases

Efficiency

Simplicity versus Efficiency

- Two goals in designing an algorithm:

Simplicity and/or Efficiency

Simplicity	Efficiency
Easy to code	Faster
Easy to understand	Use less space
Easy to debug	

- Usually, the two goals contradict each other!
  - A naïve simple algorithm is slower
  - A fast algorithm trends to be complicated

How to design?

- It depends on two questions:

1. How often does the problem occur?
    - Only a few times? or very often?

2. How big is the problem?
    - Small, medium or big?

How to design? (II)

- When the problem only occurs a few times and small,
  - We prefer a simple algorithm
- When the problem occurs many times and big,
  - We prefer efficient algorithm
- In between, you need to make your decision.
- To address these issues, we need to know if the algorithm is time and space efficient.

Analysis of an algorithm

- There are two issues:
  - The running time of the algorithm
  - The amount of storage used by the algorithm
- We focus on running time. Storage can be analyzed similarly.

- Two ways to analyze an algorithm
  - Simulation: Run the algorithm many times and measure the running time
    - Note: Input data used may be biased!
  - Mathematical analysis: Calculate the running time.

Running time

- The running time depends on the input.
- Parameterize the running time by the size of the input.
- Generally, we seek upper bounds on the running time, because everybody likes a guarantee.

Kinds of analyses

- Worst-case:
  - T(n) = maximum time of algorithm on any input of size n.
- Average-case:
  - T(n) = expected time of algorithm over all inputs of size n.
  - Need assumption of statistical distribution of inputs.
- Best-case:
  - Cheat with a slow algorithm that works fast on some input.

statement	cost
cin, cout	5
for loop	3
assignment	20
if statement	6

Example

	code	Cost	Times executed
1	cin >> n;	5	1
2	if (n is odd) then	6	1
3	for i = 1 to n	3	n
4	cin >> X[i];	5	n
5	cout << X[i]*X[i] << endl;	5	n
6	cout << “End” << endl;	5	1

- Worst case $T(n) = 5+6+3n+5n+5n+5 = 13n+16$
- Best case $T(n) = 5+6+5 = 11$
- Average case $T(n) = (13n+16)/2 + 11/2 = 6.5n+13.5$

Example

	code	Cost	Times executed
1	cin >> n;	5	1
2	for i = 1 to n	3	n
3	cin >> X[i];	5	n
4	sum = 0;	20	1
5	for i = 1 to n	3	n
6	for j = 1 to n	3	n^2
7	sum = sum + X[i]*X[j];	20	n^2
8	cout << sum << endl;	5	1

statement	cost
cin, cout	5
for loop	3
assignment	20
if statement	6

- T(n) is $5+3n+5n+20+3n+3n^2+20n^2+5$, which is $23n^2+11n+30$.
- The time is the same for worst case, average case and best case.

Is it useful to analyze the running time?

- **Testing "*" operation of your CPU**
- Q: How to test that the "*" operation of your CPU is correct?
- A: Check exhaustively a*b for all *a*, *b.*

	code
1	n=2^{32};
2	for a = 1 to n
3	for b = 1 to n
4	Test if a*b is correct;

- How long will it take?
- Any guesses?

Testing “*” operation of your CPU

- For 32-bit machine, this algorithm runs n^2 operations where n is 2^{32}.
- Assume we use a 100G-Flop CPU
  - I.e. it runs 100 billion operations per sec.

code

```
n=2^32;
for a = 1 to n
    for b = 1 to n
        Test if a*b is correct;
```

- How long does it take to test all cases?
- Time taken = n^2 / $(100*10^9)$ = $2^{32} * 2^{32}$ / $(100*10^9) \approx$ 6 years!

A better algorithm to test “*” operation

- Suppose we developed an algorithm to test “*” operation that runs in n log n time.

- How long does it take?
- Time taken = n log n / $(100 * 10^9)$
  $= 2^{32} * \log(2^{32}) / (100 * 10^9)$ sec ≈ 1.3 sec.

Another example: web-service

- You developed a web-application (say SMS server) that runs 0.03 seconds per click.

- Suppose there are 10 clicks per second. One server can handle it.
    - The process time is 0.03*10 = 0.3 seconds.
- Suppose there are 1000 clicks per second. One server is not enough.
    - The process time is 0.03 * 1000 = 30 seconds.
    - You need at least 30 servers.

- Hence, it is important to estimate the running time.

Why study algorithms and performance?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behaviour.
- Performance is the ***currency*** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

Asymptotic Analysis
(Machine independent analysis)

Different machine has different cost for the statements

statement	cost
cin, cout	c_1
for loop	c_2
assignment	c_3
if statement	c_4

	code	Cost	Times executed
1	cin >> n;	c_1	1
2	for i = 1 to n	c_2	n
3	cin >> X[i];	c_1	n
4	sum = 0;	c_3	1
5	for i = 1 to n	c_2	n
6	for j = 1 to n	c_2	n^2
7	sum = sum + X[i]*X[j];	c_3	n^2
8	cout << sum << endl;	c_1	1

- $T(n) = c_1+c_2n+c_1n+c_3+c_2n+c_2n^2+c_3n^2+c_1 = (c_2+c_3)n^2+(c_1+2c_2)n+(2c_1+c_3)$.
- Changes in costs of the statements affect the coefficients of the expression.

Machine independent 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 $1000n^2$!

Asymptotic Analysis

- Asymptotic Analysis is a method of describing the limiting behavior.
- Example:
  - $f(n)=5n^2+4n+3$.
  - When n is big enough, we have $5n^2 \leq f(n) \leq 6n^2$.
  - The coefficient 5 or 6 is machine dependent. To compare rate-of-growth, we ignore it and we say f(n) is in the order of n^2. (i.e. $f(n)=\Theta(n^2)$.)

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 $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

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 $2n^2 = O(n^3)$.

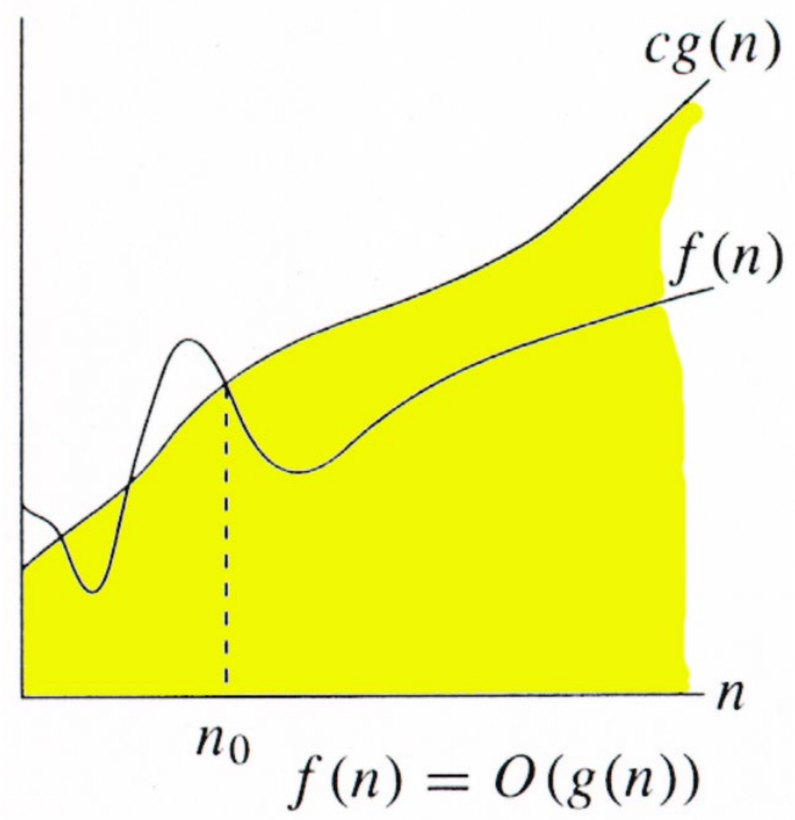
Set definition of O-notation

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

- 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.

Graphical explanation of O-notation

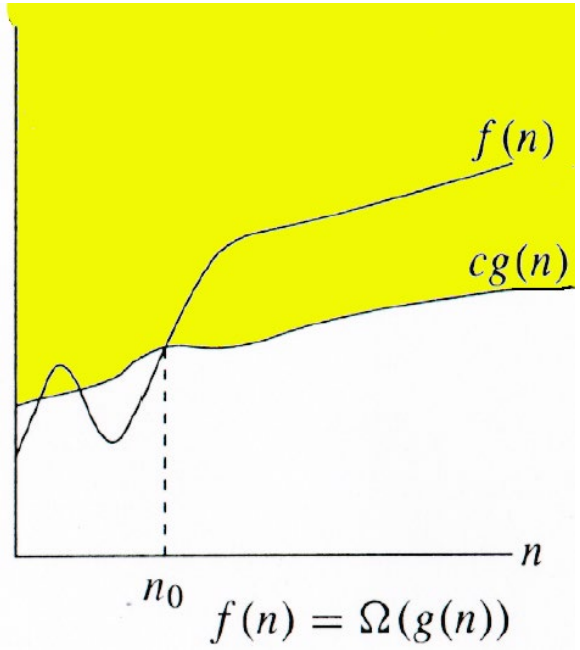
We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.



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

Ω-notation (lower bounds)

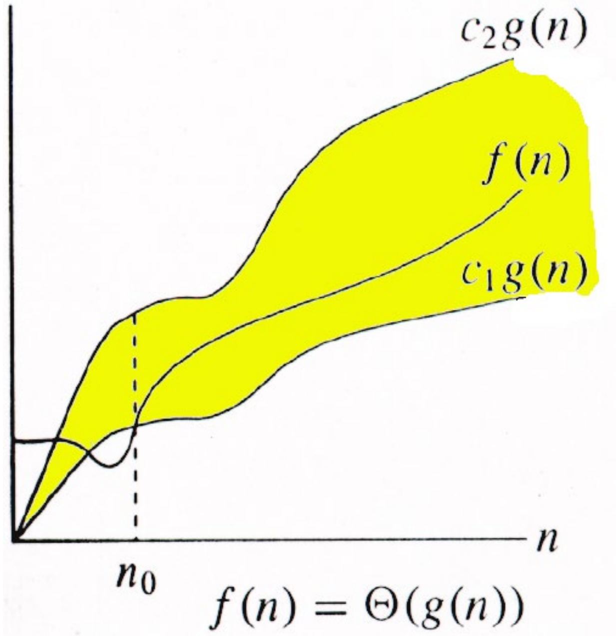
$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ $\}$



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

Θ-notation (tight bounds)

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



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

O, Ω and Θ

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

- Can you prove it?

o-notation and ω-notation

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

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

- Example: $0 \leq n < 2n^2$ 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?

o-notation and ω-notation

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

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

- Example:
  - $0 \le n < (n^2 - n)$ for n≥3 (i.e. c=1, n_0=3)
  - Hence, $n^2 - n = \omega(n)$

Acknowledgement

- The slides are modified from
  - the slides from Prof. Erik D. Demaine and Prof. Charles E. Leiserson
  - the slides from Prof. Leong Hon Wai
  - the slides from Prof. Lee Wee Sun