# CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

Department of Computer Science
University of Miami



## Outline I

## Divide-and-Conquer Approach

- Insertion Sort – **incremental approach**
  - It increased length of sorted section by one in each iteration
- **Divide-and-conquer approach**
  - **Divide** the problem into a number of smaller sub-problems of the same problem
  - **Conquer** the sub-problems by solving them recursively
  - **Combine** the solutions to the sub-problems into the solution of the original problem



## Divide-and-Conquer for Sorting

- **Problem**: An array **A[1..n]** has $n$ unsorted elements, **A[1], A[2], ..., A[n]**
- Find a **divide-and-conquer** algorithm to sort the array
- Here is the algorithm
  - **Divide: A[1], A[2], ..., A[n]** into two subarrays
    - **A[1..k]** and **A[k+1 .. n]**
    - Best if $k$ is the middle element of **A[1..n]**
  - **Conquer**: recursively sort **A[1..k]** and **A[k+1 .. n]**
  - **Combine**: merge **A[1..k]** and **A[k+1 .. n]**, if necessary
- Time complexity $T(n)$ ??
  - Time for dividing the problem in to two subproblems $T_{divide}(n)$
  - Time for sorting **A[1..k]** $T(k)$
  - Time for sorting **A[k+1..n]** $T(n-k)$
  - Time for combining sorted subarrays **A[1..k]** and **A[k+1 .. n]** $T_{combine}(n)$
  - $T(n) = T_{divide}(n) + [T(k) + T(n-k)] + T_{combine}(n)$

# Divide-and-Conquer: Example 1 — Merge Sort

- Data is in array **A**
- We want to sort elements in **A[p .. r]**
- **Divide** the problem into two sub-problems
  - Find **A[q]** such that, **A[q]** is approximately at the middle of **A[p .. r]**
  - We compute $q = \lfloor (p + r)/2 \rfloor$
  - Divide the problem into two subproblems **A[p .. q]** and **A[q+1 .. r]**
- **Conquer** — Solve each subproblem recursively
  - Merge sort items in **A[p .. q]**, and **A[q+1 .. r]**
  - Merge sort items in **A[q+1 .. r]**

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Figure: Merge sort algorithm

- **Combine** sorted items in **A[p .. q]** and **A[ q + 1 .. r ]**
  - We use $\text{MERGE}(A, p, 1, r)$ function for combining
  - $T(n) = T_{divide}(n) + [T(k) + T(n - k)] + T_{combine}(n)$
  - $T(n) = O(1) + [T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)] + O(n)$



# Combine two sorted subarrays: Merge procedure

```
MERGE(A, p, q, r)
1  n1 = q - p + 1
2  n2 = r - q
3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
4  for i = 1 to n1
5      L[i] = A[p + i - 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1 + 1] = ∞
9  R[n2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

- We want to merge elements in
  - **A[p .. q]** and **A[q+1 .. r]**

- Lines 1 and 2: determine lengths of two subarrays
- line 3: creates two new arrays L and R
- Lines 4-5: copy A[*p .. q*] in L
- Lines 6-7: copy A[*q+1 .. r*] in R
- Lines 8-9: mark right ends of L and R
- Lines 10-11: i and j are pointers for arrays L and R; initial values 1 and 1
- Line 12 to 17: merge items in L and R stores in A
- What is the time complexity?
- Use **loop invariant** to proof correctness of the procedure
- We need to consider only lines 12 to 17 for inductive proof

# Divide-and-Conquer: Example 2 — Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

- The procedure sorts elements in **A[p .. r]**
- Line 1: if number of items more than one, partition at $q$
- Line 2: the array is partitioned
- $q$ is determined by PARTITION(A,p,r)
- After partition
  - Items in **A[p .. q-1]** are $\leq$ than **A[q]**
  - Items in **A[q+1 .. r]** are $>$ than **A[q]**
- Line 3 Sort items in **A[p .. q-1]** recursively
- Line 4: Sorts items in **A[q+1 .. r]** recursively

Designing Algorithms: Divide-and-Conquer Asymptotic and standard notations, and common functions

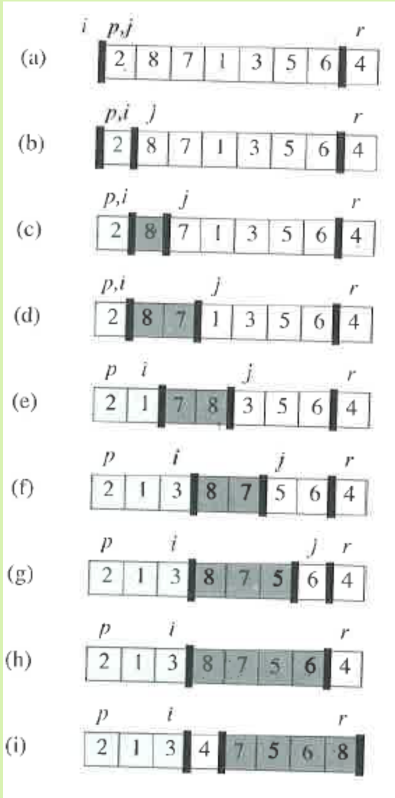
# Divide-and-conquer: Divide the Problem

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

- Use item in **A[r]** as pivot element
- Items $\leq$ **A[r]** in **A[p, r]** are moved to the left
- Items $>$ **A[r]** in **A[p, r]** are moved to the right
- This will leave a position open, where **A[r]** is inserted

# Partitioning: An example

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```






# Why we need $O$, $\Omega$, and $\Theta$?

- Time complexity $T(n)$ of the MERGE SORT algorithm shown to the right can be a recurrence equation
- $T(n) = c_1 + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2 n$
- Solving the equation above is simplified if we use only $T(\lfloor n/2 \rfloor)$ or $T(\lceil n/2 \rceil)$
- But that will lead to inequalities
  - $T(n) \geq c_1 + 2T(\lfloor n/2 \rfloor) + c_2 n$, and
  - $T(n) \leq c_1 + 2T(\lceil n/2 \rceil) + c_2 n$,
- If we solve the first inequality, the expression obtained would be lower than what would be obtained if we solve the original equation
  - This expression would be $\Omega(T(n))$, **asymptotic lower bound**
- If we solve the second inequality, the expression obtained would be higher than what would be obtained if we solve the original equation
  - This expression would be $O(T(n))$ **asymptotic upper bound**

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Figure: Merge Sort Algorithm

- If we solve the original equation, we would get $\Theta(T(n))$.
- Note: lower bound for problem is NOT **asymptotic lower bound** of an algorithm
- Note: upper bound for problem is NOT **asymptotic upper bound** of an algorithm

# $O$ — asymptotic upper bound

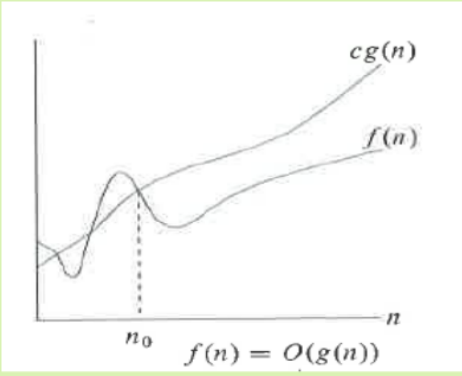
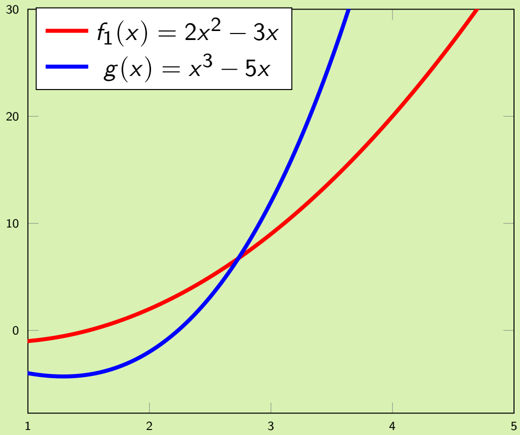



Figure: Illustration of big 'oh' concept

For a given function $g(n)$, the $O(g(n))$ denotes a set of functions,

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




- $O(g(x)) = O(x^3 - 5x) = x^3 - 5x$
- For $x = 2$, $f_1(2) = 2(2^2) - 3 \times 2 = 2$
- For $x = 2$, $g(2) = 2^3 - 5 \times 2 = -2$
- For $x = 3$, $f_1(3) = 2(3^2) - 3 \times 3 = 9$
- For $x = 3$, $g(3) = 3^3 - 5 \times 3 = 12$
- Thus, $f_1(3) < g(3)$
- In general, for $x \geq 3$, $f_1(x) < g(x)$
- Therefore, $O(g(x)) = f_1(x)$



# $O$ — asymptotic upper bound

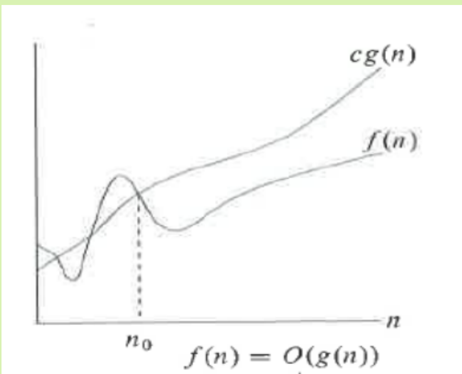
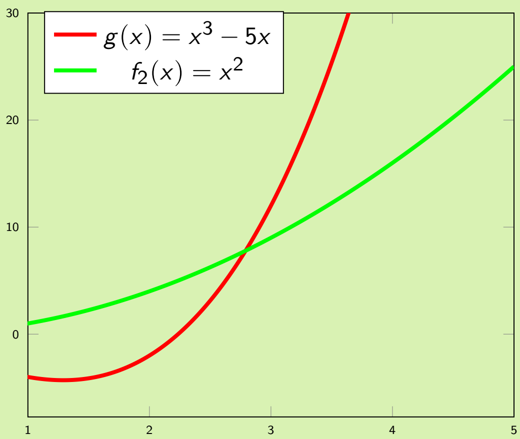



Figure: Illustration of big 'oh' concept

For a given function $g(n)$, the $O(g(n))$ denotes a set of functions,

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




- $O(g(x)) = O(x^3 - 5x) = x^3 - 5x$
- For $x = 2$, $f_2(2) = (2^2) = 4$
- For $x = 2$, $g(2) = 2^3 - 5 \times 2 = -2$
- For $x = 3$, $f_2(3) = (3^2) = 9$
- For $x = 3$, $g(3) = 3^3 - 5 \times 3 = 12$
- Thus, $f_2(3) < g(3)$
- In general, for $x \geq 3$, $f_2(x) < g(x)$
- Therefore, $O(g(x)) = f_2(x)$

# $\Omega$ — asymptotic lower bound

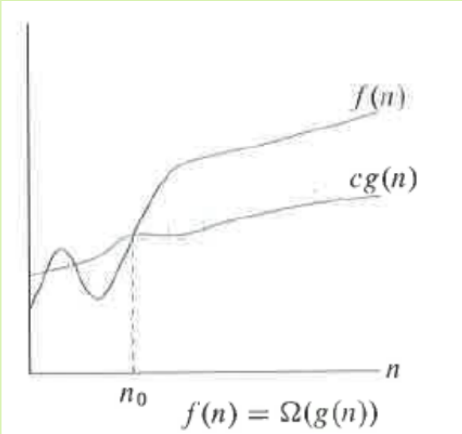
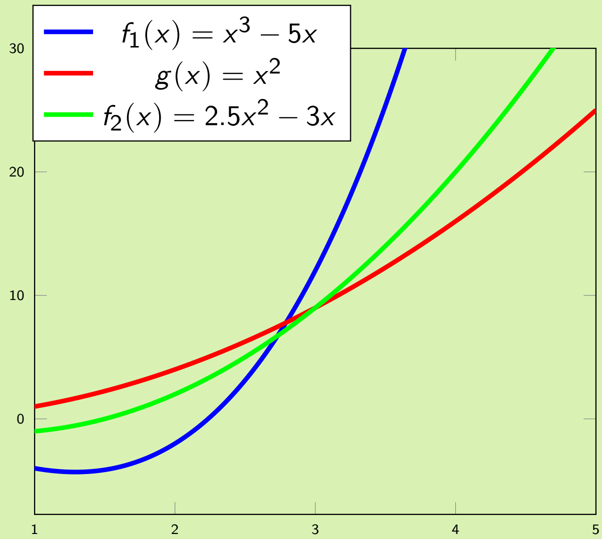



Figure: Big-$\Omega$ concept

For a given function $g(n)$, the $\Omega(g(n))$ denotes a set of functions,

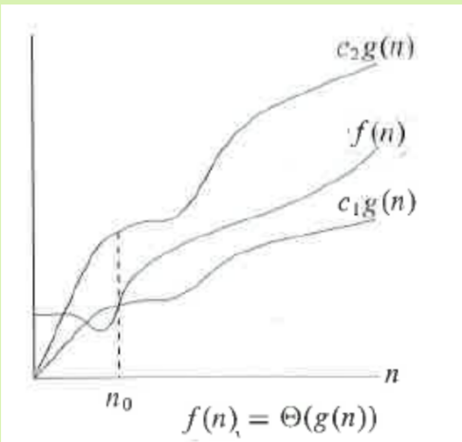
$$\Omega(g(n)) = \{f(n) : \text{there exist } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$




- $\Omega(g(x)) = \Omega(x^2) = f_1(x)$
- $\Omega(g(x)) = \Omega(x^2) = f_2(x)$

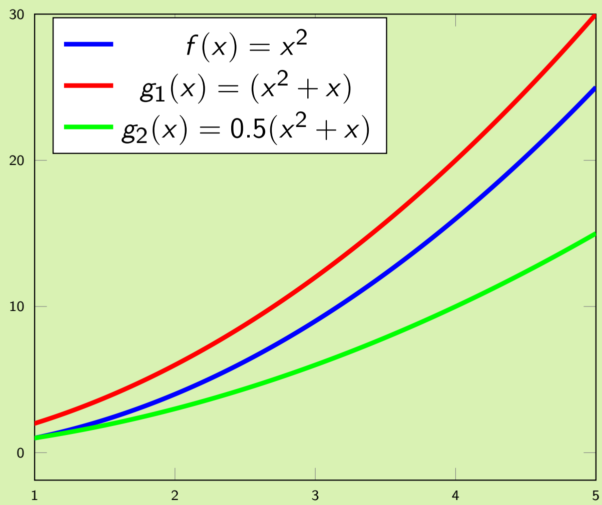


# $\Theta$ — asymptotic tight bound




For a given function $g(n)$, the $\Theta(g(n))$ denotes a set of functions,

$$\Theta(g(n)) = \{f(n) : \text{there exist } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$$




- $\Theta(x^2 + x) = x^2$

## $O$, $\Omega$, and $\Theta$ at a glance

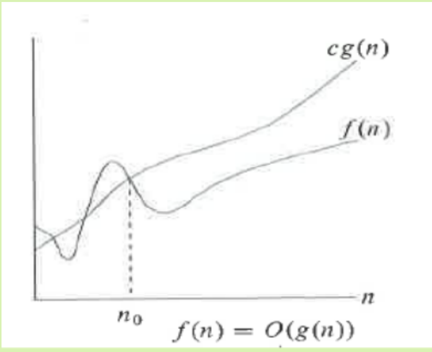



Figure: $O$ concept

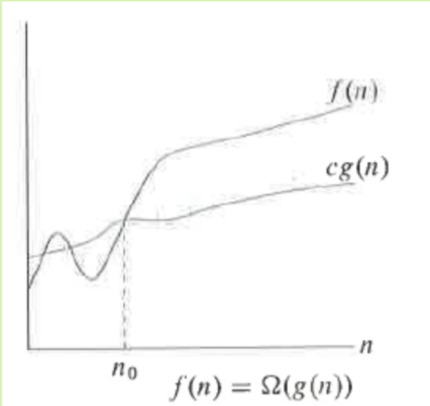



Figure: $\Omega$ concept

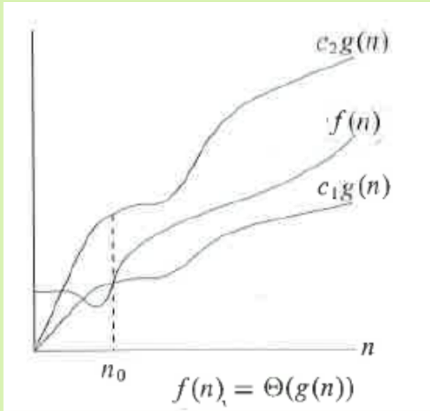



Figure: $\Theta$ concept

**Theorem 3.1** For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.



## Standard notations and common functions

- Monotonicity
  - **Monotonically increasing** function $f(n)$: for $m \leq n \Rightarrow f(m) \leq f(n)$.
  - **Monotonically decreasing** function $f(n)$: for $m \leq n \Rightarrow f(m) \geq f(n)$.
  - **Strictly increasing** function $f(n)$: for $m < n \Rightarrow f(m) < f(n)$.
  - **Strictly decreasing** function $f(n)$: for $m < n \Rightarrow f(m) > f(n)$.
- Floors and celings
- Modular arithmetic
- Polynomials
- Exponential
- Logarithms
- Factorials $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$
- Functional iteration

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 \\ f(f^{(i-1)}(n)) & \text{if } i > 0 \end{cases} \tag{1}$$

For example, if $f(n) = 2n$, then $f^{(2)}(n) = f(f^{(1)}(n)) = f(f(f^{(0)}(n))) = f(f(n)) = f(2n) = 2 \times 2n = 2^2 n$

- The iterated logarithmic function

$$\lg^* n = \min\{i \geq 0 : \lg^{(i)} n \leq 1\}$$

  - Examples, $\lg^* 16 = 3$, $\lg^* 65536 = 4$ $\lg^*(2^65536) = 5$
  - The value of this function is **really really** small, even for a very large number.