# COT 5407: Introduction to Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/5407S19.html

# Homework

- Read Guidelines and Follow Instructions!
- Statement of Collaboration
  - Take it **seriously.**
  - If true, **reproduce** the statement faithfully.
  - For each problem, explain **separately** the sources and your collaborations with other people.
  - Your homework **will not be graded** without the statement.
- Extra Credit Problem
  - You can turn it in any time within a month or until last class day, whichever is earlier.
  - If you are not sure of your solution, don't waste my time.
  - You will NOT get partial credit on an extra credit problem.
  - Submit it separately and label it appropriately.

# Definition of big-Oh

- We say that $F(n) = O(G(n))$,
  - If there exists two positive constants, c and $n_0$, such that
  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$
- We say that $F(n) = \Omega(G(n))$,
  - If there exists two positive constants, c and $n_0$, such that
  - For all $n \geq n_0$, we have $F(n) \geq c\, G(n)$
- We say that $F(n) = \Theta(G(n))$,
  - If $F(n) = O(G(n))$ and $F(n) = \Omega(G(n))$
- We say that $F(n) = \omega(G(n))$,
  - If $F(n) = \Omega(G(n))$, but $F(n) \neq \Theta(G(n))$
- We say that $F(n) = o(G(n))$,
  - If $F(n) = O(G(n))$, but $F(n) \neq \Theta(G(n))$

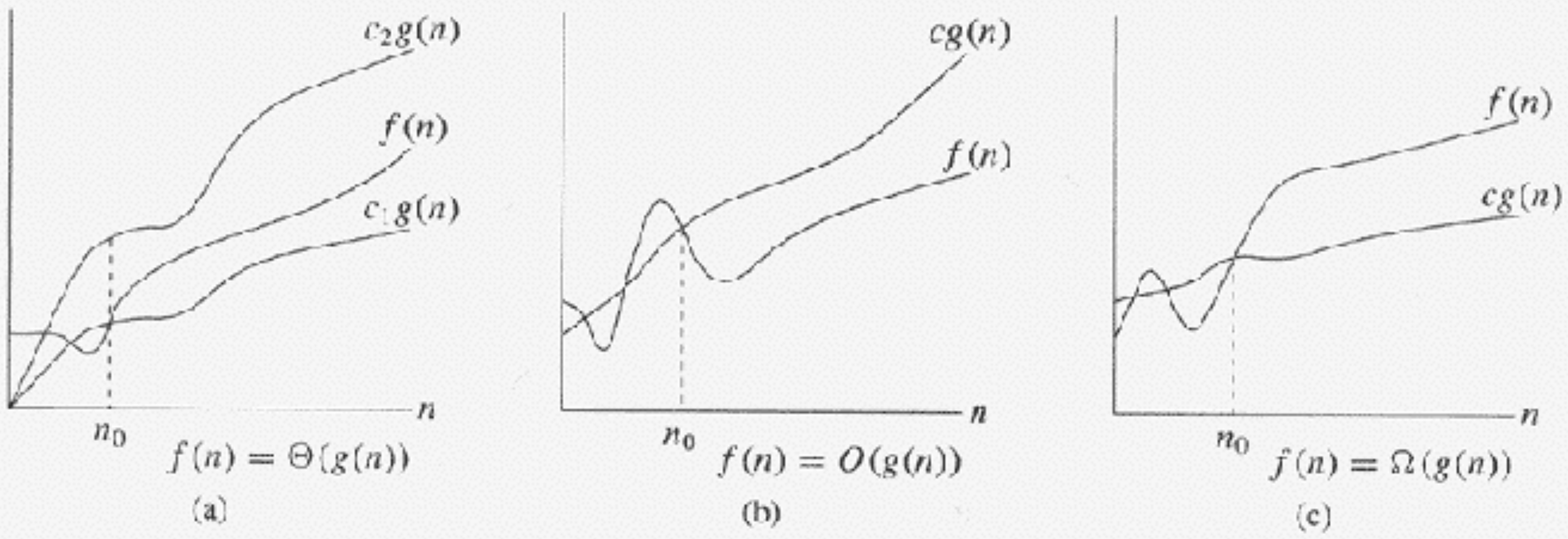
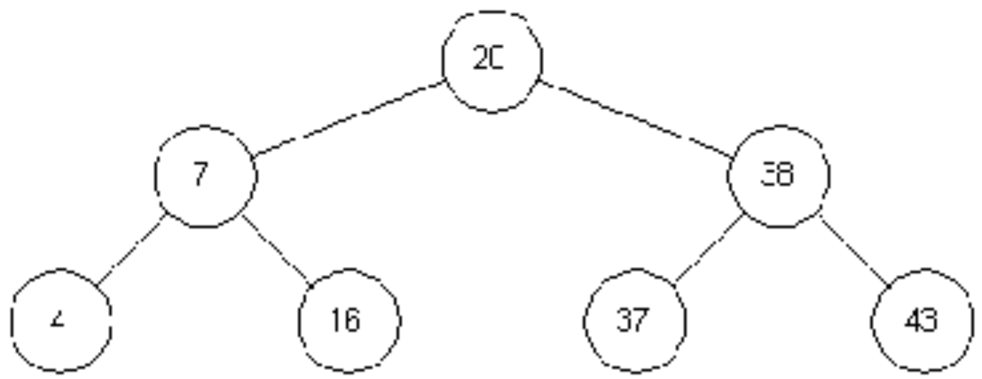



**Figure 3.1** Graphic examples of the $\Theta$, $O$, and $\Omega$ notations. In each part, the value of $n_0$ shown is the minimum possible value; any greater value would also work. **(a)** $\Theta$-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants $n_0$, $c_1$, and $c_2$ such that to the right of $n_0$, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** $O$-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** $\Omega$-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants $n_0$ and $c$ such that to the right of $n_0$, the value of $f(n)$ always lies on or above $cg(n)$.

# Storing binary trees as arrays




| | | | | | | |
|---|---|---|---|---|---|---|
| 20 | 7 | 38 | 4 | 16 | 37 | 43 |

# Heaps (Max-Heap)

| | | | | | | |
|---|---|---|---|---|---|---|
| 43 | 16 | 38 | 4 | 7 | 37 | 20 |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 43 | 16 | 38 | 4 | 7 | 37 | 20 | 2 | 3 | 6 | 1 | 30 |

HEAP represents a **complete** binary tree stored as an array such that:

- HEAP PROPERTY: Parent value is ≥ child's value

Complete Binary Tree:

- Tree is filled on all levels except the last level
- Last level is filled from left to right
- Left & right child of $i$ are in locations $2i$ and $2i+1$

# HeapSort

- First convert array into a heap (**BUILD-MAX-HEAP**, p157)
- Then convert heap into sorted array (**HEAPSORT**, p160)

# Animation Demos

**http://www-cse.uta.edu/~holder/courses/cse2320/lectures/applets/sort1/heapsort.html**

http://cg.scs.carleton.ca/~morin/misc/sortalg/

# HeapSort: Part 1

```
MAX-HEAPIFY(array A, int i)
    ▷ Assume subtree rooted at i is not a heap;
    ▷ but subtrees rooted at children of i are heaps
    l ← LEFT[i]
    r ← RIGHT[i]
    if ((l ≤ heap-size[A]) and (A[l] > A[i]))
        then largest ← l
        else largest ← i
    if ((r ≤ heap-size[A]) and (A[r] > A[largest]))
        then largest ← r
    if (largest ≠ i)
        then exchange A[i] ↔ A[largest]
             MAX-HEAPIFY(A, largest)
```

p154, CLRS

# Analysis of Max-Heapify

```
MAX-HEAPIFY(array A, int i)
    ▷ Assume subtree rooted at i is not a heap;
    ▷ but subtrees rooted at children of i are heaps
    l ← LEFT[i]
    r ← RIGHT[i]
    if ((l ≤ heap-size[A]) and (A[l] > A[i]))
        then largest ← l
        else largest ← i
    if ((r ≤ heap-size[A]) and (A[r] > A[largest]))
        then largest ← r
    if (largest ≠ i)
        then exchange A[i] ↔ A[largest]
                MAX-HEAPIFY(A, largest)
```

- $T(N) \leq T(2N/3) + O(1)$
- When called on node i, either it terminates with O(1) steps or makes a recursive call on node at lower level
- At most 1 call per level
- Time Complexity = O(level of node i) = $O(h_i)$ = O(log N)

# HeapSort: Part 2

```
BUILD-MAX-HEAP(array A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

# HeapSort: Part 2

```
BUILD-MAX-HEAP(array A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)

HEAPSORT(array A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
        do exchange A[1] ↔ A[i]
           heap-size[A] ← heap-size[A] − 1
           MAX-HEAPIFY(A, 1)
```

O(log n)

Total:
O(nlog n)

# HeapSort: Part 2

```
BUILD-MAX-HEAP(array A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

- For n/2 nodes, height is 1 and # of comparisons = 0,
- For n/4 nodes, height is 2 and # of comparisons = 1,
- For n/8 nodes, height is 3 and # of comparisons = 2, …
- Total = summation ((height -1) X # of nodes at that height)
- Total = summation ((height – 1) X $N/2^{height}$)
- Total ≤ summation (height X $N/2^{height}$)
- Total ≤ N X summation (height X $1/2^{height}$)

# Build-Max-Heap Analysis

We need to compute:

$$\boxed{n \times \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}}$$

Build-Max-Heap: O(n)

$$\text{We know that } \sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

$$\text{Differentiating both sides, we get } \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

$$\text{Multiplying both sides by } x\text{, we get } \sum_{k=0}^{\infty} kx^{k} = \frac{x}{(1-x)^2}$$

$$\text{Setting } x = 1/2\text{, we can show that } \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq 2$$

# HeapSort

```
BUILD-MAX-HEAP(array A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

```
HEAPSORT(array A)
BUILD-MAX-HEAP(A)
for i ← length[A] downto 2
        do exchange A[1] ↔ A[i]
            heap-size[A] ← heap-size[A] − 1
            MAX-HEAPIFY(A, 1)
```

- Single call to Max-Heapify runs in O(h) time
- However, Build-Max-Heap runs in O(n) time
- HeapSort runs in O(n log n) time

# Sorting Algorithms

- SelectionSort
- InsertionSort
- BubbleSort
- QuickSort
- MergeSort
- HeapSort
- Bucket & Radix Sort
- Counting Sort

Worst Case: $O(N^2)$

Worst Case: O(N logN)

Avg Case: O(N log N)

Lower Bound for Comparison-based Sorting

Worst Case: O(N); Not comparison-based

# Upper and Lower Bounds

- Time Complexity of a Problem
  - **Difficulty**: Since there can be many algorithms that solve a problem, what time complexity should we pick?
  - **Solution**: Define upper bounds and lower bounds within which the time complexity lies.
- What is the **upper** bound on time complexity of sorting?
  - **Answer**: Since SelectionSort runs in worst-case $O(N^2)$ and MergeSort runs in $O(N \log N)$, either one works as an upper bound.
  - **Critical Point**: Among all upper bounds, the best is the lowest possible upper bound, i.e., time complexity of the best algorithm.
- What is the **lower** bound on time complexity of sorting?
  - **Difficulty**: If we claim that lower bound is $O(f(N))$, then we have to prove that no algorithm that sorts N items can run in worst-case time $o(f(N))$.

# Lower Bounds

- It's possible to prove lower bounds for many comparison-based problems.
- For comparison-based problems, for inputs of length N, if there are P(N) possible solutions, then
  - any algorithm needs $\log_2(P(N))$ to solve the problem.
- Binary Search on a list of N items has at least N + 1 possible solutions. Hence lower bound is
  - $\log_2(N+1)$.
- Sorting a list of N items has at least N! possible solutions. Hence lower bound is
  - $\log_2(N!) = O(N \log N)$
- Thus, MergeSort is an optimal algorithm.
  - Because its worst-case time complexity equals lower bound!

# Beating the Lower Bound

- Bucket Sort
  - Runs in time O(N+K) given N integers in range [a+1, a+K]
  - If K = O(N), we are able to sort in O(N)
  - How is it possible to beat the lower bound?
  - Only because we know more about the data.
  - If nothing is know about the data, the lower bound holds.
- Radix Sort
  - Runs in time O(d(N+K)) given N items with d digits each in range [1,K]
- Counting Sort
  - Runs in time O(N+K) given N items in range [a+1, a+K]

# Bucket Sort

- N **integer** values in the range [a..a+m-1]
- For e.g., sort a list of 50 scores in the range [0..9].
- **Algorithm**
  - Make m buckets [a..a+m-1]
  - As you read elements throw into appropriate bucket
  - Output contents of buckets [0..m] in that order
- **Time O(N+m)**
- **Warning: <u>This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.</u>**

# Stable Sort

- A sort is **stable** if equal elements appear in the same order in both the input and the output.
- Which sorts are stable?

# Radix Sort

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3 5 9 | | 3 5 9 | | 3 3 6 | | 3 3 6 |
| 3 5 7 | | 3 5 7 | | 3 5 9 | | 3 5 1 |
| 3 5 1 | | 3 5 1 | | 3 5 7 | | 3 5 5 |
| 7 3 9 | → | 3 3 6 | → | 3 5 1 | → | 3 5 7 |
| 3 3 6 | | 3 5 5 | | 3 5 5 | | 3 5 9 |
| 7 2 0 | | 7 3 9 | | 7 2 0 | | 7 2 0 |
| 3 5 5 | | 7 2 0 | | 7 3 9 | | 8 3 9 |

**Algorithm**

**for** i = 1 **to** d **do**
    **sort** array A on digit i using any sorting algorithm

Time Complexity: $O((N+m) + (N+m^2) + \ldots + (N+m^d))$

Space Complexity: $O(m^d)$

# Radix Sort

| | | | |
|---|---|---|---|
| 3 2 9 | 7 2 0 | 7 2 0 | 3 2 9 |
| 4 5 7 | 3 5 5 | 3 2 9 | 3 5 5 |
| 6 5 7 | 4 3 6 | 4 3 6 | 4 3 6 |
| 8 3 9 | 4 5 7 | 8 3 9 | 4 5 7 |
| 4 3 6 | 6 5 7 | 3 5 5 | 6 5 7 |
| 7 2 0 | 3 2 9 | 4 5 7 | 7 2 0 |
| 3 5 5 | 8 3 9 | 6 5 7 | 8 3 9 |

**Algorithm**

Time Complexity: $O((n+m)d)$

**for** i = 1 **to** d **do**

**sort** array A on digit i using a stable sort algorithm

• **Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.

# Counting Sort

Initial Array

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

Counts

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 3 | 0 | 1 |

Cumulative Counts

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 2 | 4 | 7 | 7 | 8 |

- **Warning**: This algorithm cannot be used for "infinite-precision" real numbers, even if the range of values is specified.