COT 5407: Introduction to Algorithms

Giri Narasimhan

ECS 254A; Phone: x3748

giri@cis.fiu.edu

http://www.cis.fiu.edu/~giri/teach/5407S17.html

https://moodle.cis.fiu.edu/v3.1/course/view.php?id=1494

Evaluation

• Exams (2)	40%
• Quizzes	10%
• Homework Assignments	40%
• Semester Project	5%
• Class Participation	5%

http://www.cis.fiu.edu/~giri/teach/5407S17.html

https://moodle.cis.fiu.edu/v3.1/course/view.php?id=1494

What you should already know …

- Array Lists
- Linked Lists
- Sorted Lists
- Stacks and Queues
- Trees
- Binary Search Trees
- Heaps and Priority Queues
- Graphs
  - Adjacency Lists
  - Adjacency Matrices
- Basic Sorting Algorithms

Celebrity Problem

- A **Celebrity** is one that knows nobody and that everybody knows.

Celebrity Problem:

INPUT: n persons with a $n \times n$ information matrix.

OUTPUT: Find the "celebrity", if one exists.

MODEL: Only allowable questions are:

  - ***Does person i know person j?***

Only allowable answers are:

  - ***Yes*** or ***No?***

- Naive Algorithm: $O(n^2)$ Questions.

Celebrity Problem (Cont' d)

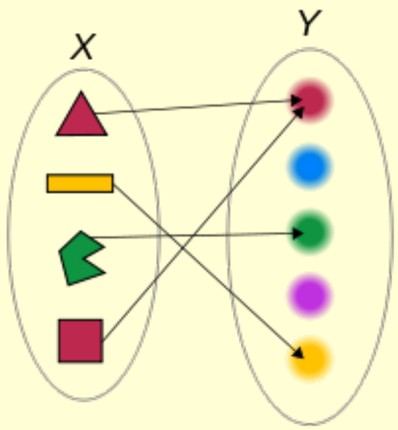
- Naive Algorithm: O(n^2) Questions.
  - Ask everyone of everyone else for a total of n(n-1) questions
- Using Divide-and-Conquer: O($n \log_2 n$) Questions.
  - Divide the people into two equal sets. Solve recursively and find two candidate celebrities from the two halves. Then verify which one (if any) is a celebrity by asking n-1 questions to each of them and n-1 questions to everyone else about them. This gives a recurrence for the total number of questions asked: $T(n) = 2T(n/2) + 2n$
- Improved solution?
  - How many questions are needed to find a non-celebrity?
  - Hint: What information do you gain by asking one question?
  - Do not proceed to next slide before thinking this through …

Information Gain from One Question

- Assume that you ask person A:
  - Do you know person B?
- If answer is No, then
  - B is clearly not a celebrity
    - Because everyone in the room knows a celebrity
- If answer is Yes, then
  - What can we infer?
  - A is clearly not a celebrity
    - Because a celebrity known nobody in the room
- In each question, we eliminate one person from being a "potential" celebrity
- We need 3(n-1) – 2 questions in the worst case to find the celebrity, if one exists. **Why**?

Definitions

Abstract Problem: defines a function from any allowable input to a corresponding output



Instance of a Problem: a specific input to abstract problem

Algorithm: well-defined computational procedure that takes an instance of a problem as input and produces the correct output

An Algorithm must <u>halt</u> on every input with <u>correct</u> output.

Sorting

- Input is a sequence of n items that can be compared.
- Output is an ordered list of those n items
  - I.e., a reordering or permutation of the input items such that the items are in sorted order
- Fundamental problem that has received a lot of attention over the years.
- Used in many applications.
- Scores of different algorithms exist.
- Task: To compare algorithms
  - On what bases?
    - Time
    - Space
    - Other

Sorting Algorithms

- Number of Comparisons
- Number of Data Movements
- Additional Space Requirements

Sorting Algorithms

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

Psuedocode

- Convention about statements
- Indentation
- Comments
- Parameters -- passed by value not reference
- And/or are short-circuiting

Invariants

- Extremely Useful tool for
  - Understanding an algorithm
  - Proving its correctness
  - Analyzing its time and space complexity

How to prove invariants & correctness

- **Initialization**: prove it is true at start
- **Maintenance**: prove it is maintained within iterative control structures
- **Termination**: show how to use it to prove correctness

SelectionSort: Algorithm Invariants

- At end of iteration k, the k smallest items are in their correct location
- NEED TO PROVE THE INVARIANT!!
  - Initialization: Is it true at $k = 0$?
  - Maintenance: Is it true at $k = j$, given that it is true for all values of $k < j$?
- Correctness of Algorithm is often proved by using invariant at termination
  - Termination: What happens at $k = N-1$?

More Invariants

- InsertionSort:
  - At the start of iteration k, the first k items are in sorted order
- BubbleSort:
  - At end of iteration k, the k smallest items are in their correct location

Algorithm Analysis

- Worst-case time complexity*
- (Worst-case) space complexity
- Average-case time complexity

Worst-Case Analysis

Two Techniques:

1. Count number of steps from pseudocode and add
2. Use invariant, write down recurrence relation and solve it

We will use big-Oh notation to write down time and space complexity (for both worst-case & average-case analyses).

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 $F(n) \neq \Theta(G(n))$

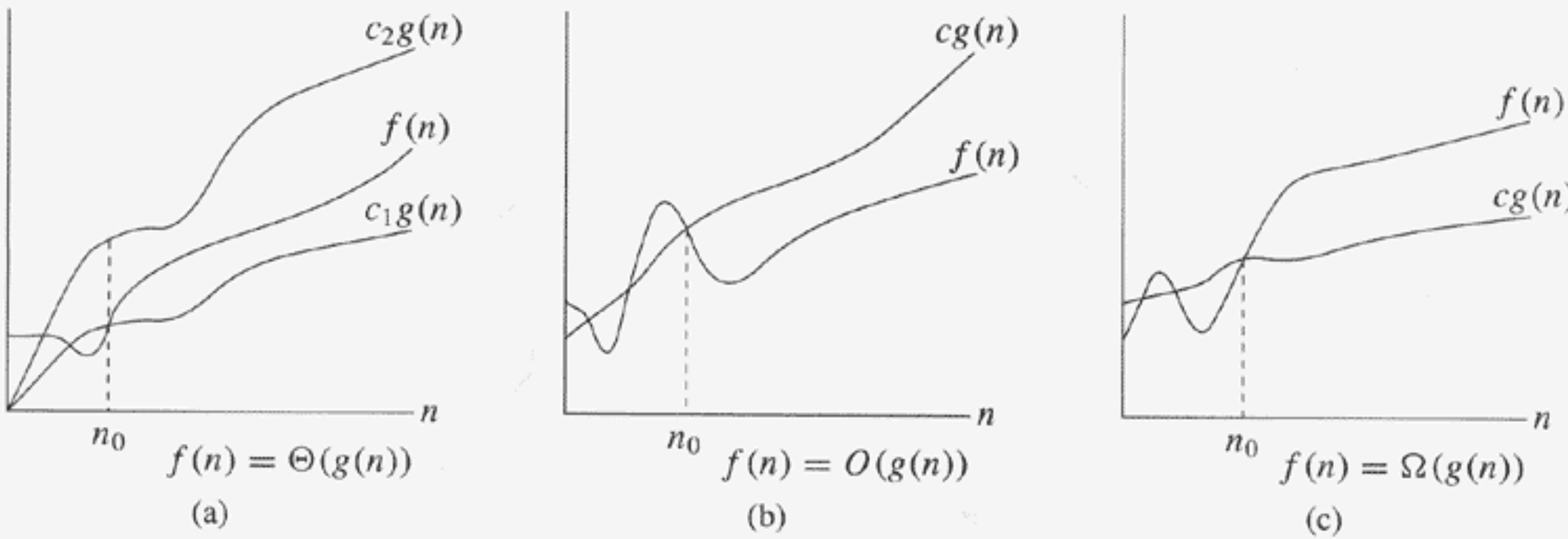


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ 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)** Ω-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 $cg(n)$.

Definition of big-Oh

- We say that
  - $F(n) = O(G(n))$

  If there exists two <u>positive</u> constants, c and n_0, such that
  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$
- Thus, to show that $F(n) = O(G(n))$, you need to find two positive constants that satisfy the condition mentioned above
- Also, to show that $F(n) \neq O(G(n))$, you need to show that for any value of c, there does not exist a positive constant n_0 that satisfies the condition mentioned above

SelectionSort – Worst-case analysis

```
SELECTIONSORT(array A)
N ← length[A]
for p ← 1 to N
        do ▷ Compute j
           j ← p
           for m ← p + 1 to N
                do if (A[m] < A[j])
                      then j ← m
           ▷ Swap A[p] and A[j]
           temp ← A[p]
           A[p] ← A[j]
           A[j] ← temp
```

N-p comparisons

3 data movements

SelectionSort – Worst-case analysis

```
SELECTIONSORT(array A)
N ← length[A]
for p ← 1 to N
        do ▷ Compute j
            j ← p
            for m ← p + 1 to N
                do if (A[m] < A[j])
                        then j ← m
            ▷ Swap A[p] and A[j]
            temp ← A[p]
            A[p] ← A[j]
            A[j] ← temp
```

- Data Movements
  - O(N)
- # Comparisons
  - Learn how to do sum of series!
  - $O(N^2)$
- Time Complexity
  - $O(N^2)$

SelectionSort – Worst-case space analysis

SELECTIONSORT(*array* A)

$N \leftarrow length[A]$
for $p \leftarrow 1$ **to** N
  do $\triangleright$ Compute j
    $j \leftarrow p$
    for $m \leftarrow p + 1$ **to** N
      do if $(A[m] < A[j])$
        then $j \leftarrow m$
    $\triangleright$ Swap $A[p]$ and $A[j]$
    $temp \leftarrow A[p]$
    $A[p] \leftarrow A[j]$
    $A[j] \leftarrow temp$

- Temp Space
  - No extra arrays or data structures
  - O(1)

Average-Case Analysis

- SelectionSort
  - Average-case time = Worst-case time
- InsertionSort
  - Average-case time < Worst-case time
  - On the average, in the k^{th} iteration, we will only compare $(k-1)/2$ items with the new item
  - However, average-case time complexity
    - Is still $O(N^2)$, even though constants are smaller

EXTRA SLIDES ON SORTING

SelectionSort

Array Position		0	1	2	3	4	5
Initial State		8	5	9	2	6	3
After Iteration 1		2	5	9	8	6	3
After Iteration 2		2	3	9	8	6	5
After Iteration 3		2	3	5	8	6	9
After Iteration 4		2	3	5	6	8	9
After Iteration 5		2	3	5	6	8	9

SelectionSort

SELECTIONSORT($array\ A$)

$N \leftarrow length[A]$
for $p \leftarrow 1$ **to** N
  do Compute j, the index of the smallest item in $A[p..N]$
  Swap $A[p]$ and $A[j]$

SelectionSort

```
SelectionSort(array A)
N ← length[A]
for p ← 1 to N
      do ▷ Compute j
         j ← p
         for m ← p + 1 to N
              do if (A[m] < A[j])
                   then j ← m
         ▷ Swap A[p] and A[j]
         temp ← A[p]
         A[p] ← A[j]
         A[j] ← temp
```

SelectionSort: Algorithm Invariants

- iteration k:
  - the k smallest items are in correct location
- NEED TO PROVE THE INVARIANT!!

How to prove invariants & correctness

- **Initialization**: prove it is true at start
- **Maintenance**: prove it is maintained within iterative control structures
- **Termination**: show how to use it to prove correctness

Algorithm Analysis

- Worst-case time complexity
- (Worst-case) space complexity
- Average-case time complexity

SelectionSort

```
SelectionSort(array A)
N ← length[A]
for p ← 1 to N
        do ▷ Compute j
           j ← p
           for m ← p + 1 to N
               do if (A[m] < A[j])
                     then j ← m
           ▷ Swap A[p] and A[j]
           temp ← A[p]
           A[p] ← A[j]
           A[j] ← temp
```

$O(n^2)$ time

$O(1)$ space

Solving Recurrence Relations

Page 62, [CLR]

Recurrence; Cond	Solution
$T(n) = T(n-1) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-1) + O(n)$	$T(n) = O(n^2)$
$T(n) = T(n-c) + O(1)$	$T(n) = O(n)$
$T(n) = T(n-c) + O(n)$	$T(n) = O(n^2)$
$T(n) = 2T(n/2) + O(n)$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a = b$	$T(n) = O(n \log n)$
$T(n) = aT(n/b) + O(n)$; $a < b$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a - \epsilon})$	$T(n) = O(n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = O(n^{\log_b a})$	$T(n) = \Theta(n^{\log_b a} \log n)$
$T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(f(n))$ $af(n/b) \leq cf(n)$	$T(n) = \Omega(n^{\log_b a} \log n)$

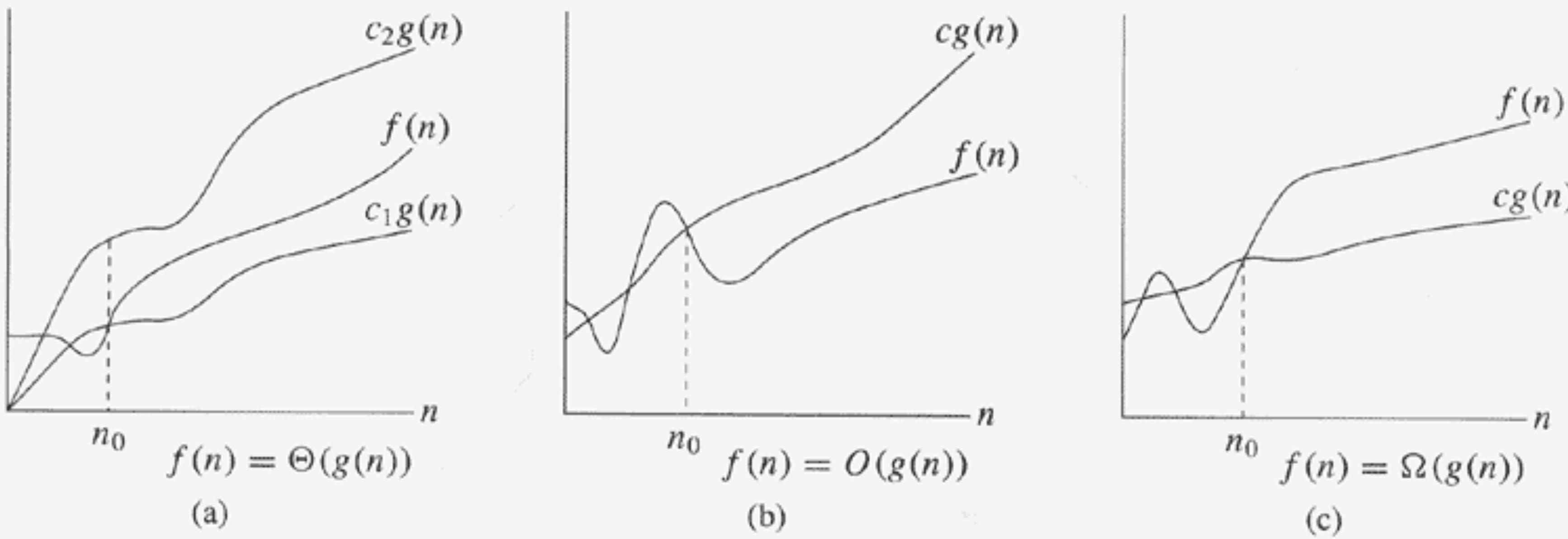


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that to the right of n_0, the value of $f(n)$ always lies between $c_1g(n)$ and $c_2g(n)$ 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)** Ω-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 $cg(n)$.

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        do key ← A[j]
           ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
           i ← j − 1
           while i > 0 and A[i] > key
               do A[i + 1] ← A[i]
                  i ← i − 1
           A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** $length[A]$	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
▷ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

$O(n^2)$ time

$O(1)$ space

InsertionSort: Algorithm Invariant

- iteration k:
  - the first k items are in sorted order.

Figure 8.3

Basic action of insertion sort (the shaded part is sorted)

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After a[0..1] is sorted	5	8	9	2	6	3
After a[0..2] is sorted	5	8	9	2	6	3
After a[0..3] is sorted	2	5	8	9	6	3
After a[0..4] is sorted	2	5	6	8	9	3
After a[0..5] is sorted	2	3	5	6	8	9

Figure 8.4

A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed).

Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After a[0..1] is sorted	5	8	**9**			
After a[0..2] is sorted	5	8	9	**2**		
After a[0..3] is sorted	2	5	8	9	**6**	
After a[0..4] is sorted	2	5	6	8	9	**3**
After a[0..5] is sorted	2	3	5	6	8	9

```
BUBBLESORT(A)
for i ← 1 to length[A]
    do for j ← length[A] downto i + 1
        do if A[j] < A[j − 1]
            then exchange A[j] ↔ A[j − 1]
```

$O(n^2)$ time

$O(1)$ space

BubbleSort: Algorithm Invariant

- In each pass, every item that does not have a smaller item after it, is moved as far up in the list as possible.
- Iteration k:
  - k smallest items are in the correct location.

Animation Demos

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

Comparing $O(n^2)$ Sorting Algorithms

- InsertionSort and SelectionSort (and ShakerSort) are roughly twice as fast as BubbleSort for small files.
- InsertionSort is the best for very small files.
- $O(n^2)$ sorting algorithms are **NOT** useful for large random files.
- If comparisons are very expensive, then among the $O(n^2)$ sorting algorithms, insertionsort is best.
- If data movements are very expensive, then among the $O(n^2)$ sorting algorithms, ?? is best.

Problems to think about!

- What is the least number of comparisons you need to sort a list of 3 elements? 4 elements? 5 elements?
- How to arrange a tennis tournament in order to find the tournament champion with the least number of matches? How many tennis matches are needed?