COT 6405: Analysis of Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/6405F19.html

Momentos

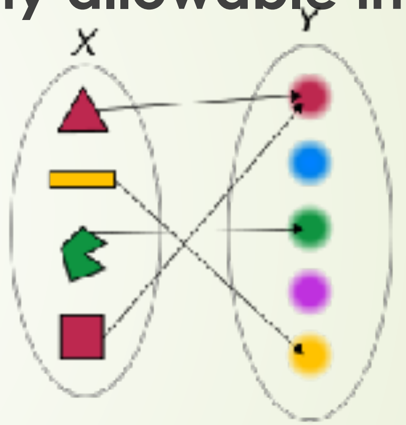
- Slides and Audio online
- Need to register
  - Go to https://fiu.momentos.life
  - If you don't already have an account
    - Click on "Sign up"
    - Follow instructions & use referral code: **5T6LSV**
  - If you have an account, "Add Course" with code **5T6LSV**
  - Verify account using link sent to email

Homework 1 is ready

- Read Submission Guidelines before starting on homework.

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 halt on every input with correct 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

Worst-Case Time Analysis

- Two Techniques:
  1. Counts and Summations:
     - Count number of steps from pseudocode and add
  2. Recurrence Relations:
     - 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).
- Compute worst possible time of all input instances of length N.

Definition of big-Oh

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

If there exists **positive** constants, c and n_0, such that

  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$

To prove big-Oh relationships

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

  If there exists **positive** constants, c and n_0, such that
  - For all $n \geq n_0$, we have $F(n) \leq c\, G(n)$
- To show that $F(n) = O(G(n))$, you need to find two positive constants that satisfy the condition mentioned above

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
  - $F(n) \neq O(G(n))$

If for any positive constant, c, such that

  - There exists $n \geq n_0$, we have $F(n) > c\, G(n)$

To disprove big-Oh relationships

- We say that
  - $F(n) \neq O(G(n))$

  If for any **positive** constant, c, such that
  - There exists $n \geq n_0$, we have $F(n) > c\, G(n)$
- To show that $F(n) \neq O(G(n))$,
  - need to show that for any positive 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

- Data Movements

$$= \sum_{p=1}^{N} 3 = 3 \times N = O(N)$$

- Number of Comparisons

$$\begin{aligned} &= \sum_{p=1}^{N} (N - p) \\ &= \sum_{p=1}^{N} N - \sum_{p=1}^{N} p \\ &= (N \times N) - (N)(N+1)/2 \\ &= O(N^2) \end{aligned}$$

Learn how to sum series

- Time Complexity = $O(N^2)$
- Homework: Show it is not O(N)

SelectionSort – Space Complexity

```
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
```

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

Solving Recurrence Relations

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

Solving Recurrences: Recursion-tree method

- Substitution method fails when a good guess is not available
- Recursion-tree method works in those cases
  - Write down the recurrence as a tree with recursive calls as the children
  - Expand the children
  - Add up each level
  - Sum up the levels
- Useful for analyzing divide-and-conquer algorithms
- Also useful for generating good guesses to be used by substitution method

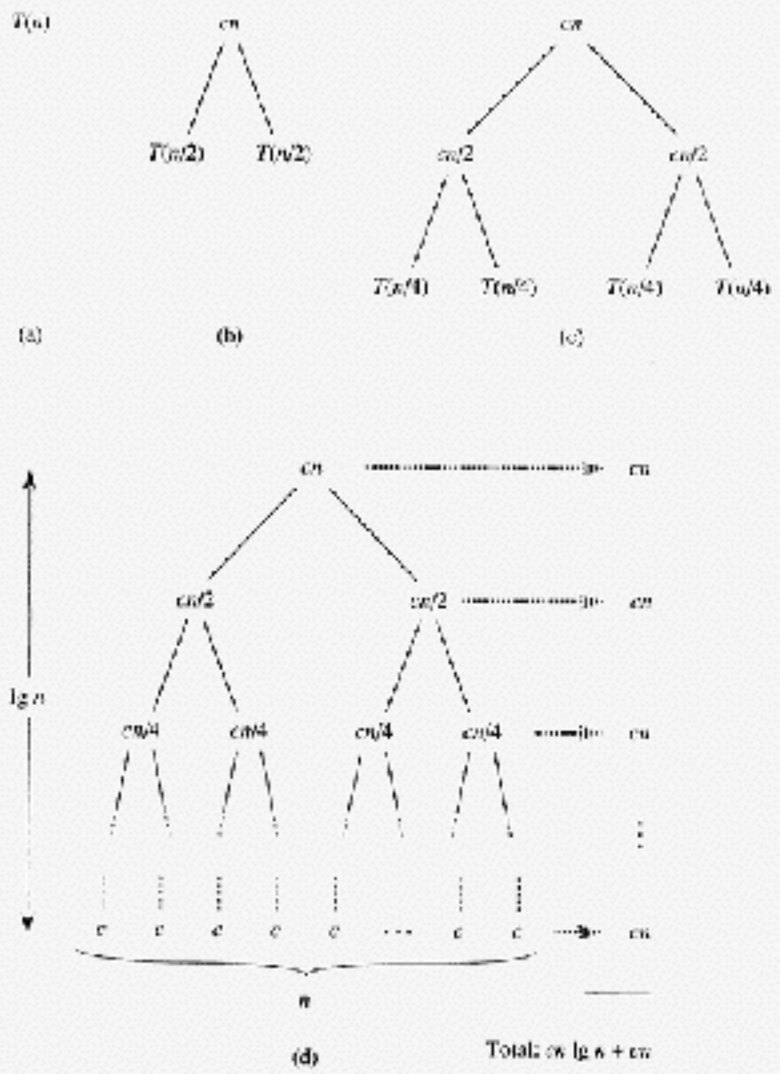


Figure 2.5 The construction of a recursion tree for the recurrence $T(n) = 2T(n/2) + cn$. Part **(a)** shows $T(n)$, which is progressively expanded in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has $\lg n + 1$ levels (i.e., it has height $\lg n$, as indicated), and each level contributes a total cost of cn. The total cost, therefore, is $cn \lg n + cn$, which is $\Theta(n \lg n)$.

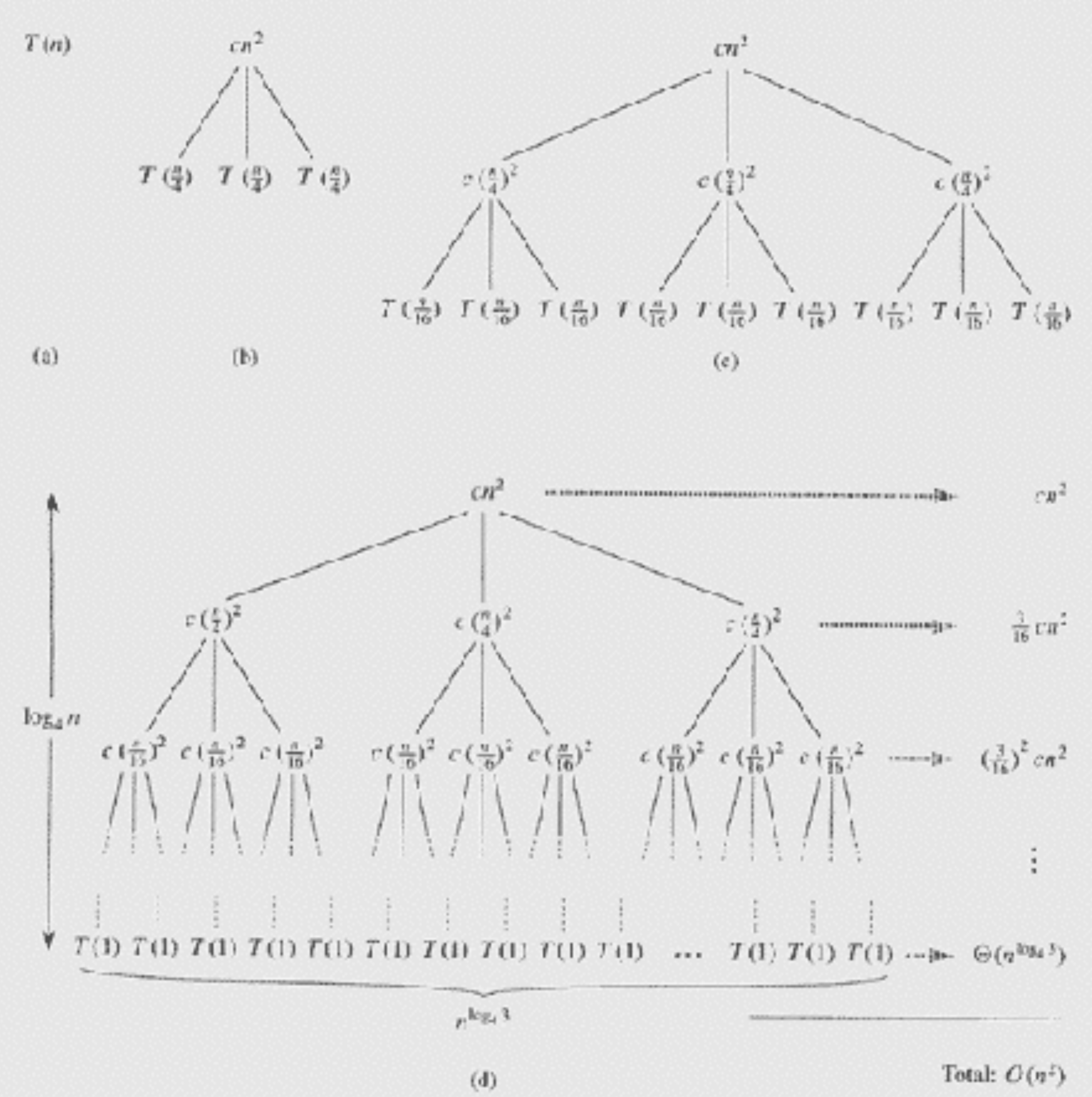


Figure 4.1 The construction of a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part (a) shows $T(n)$, which is progressively expanded in (b)–(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

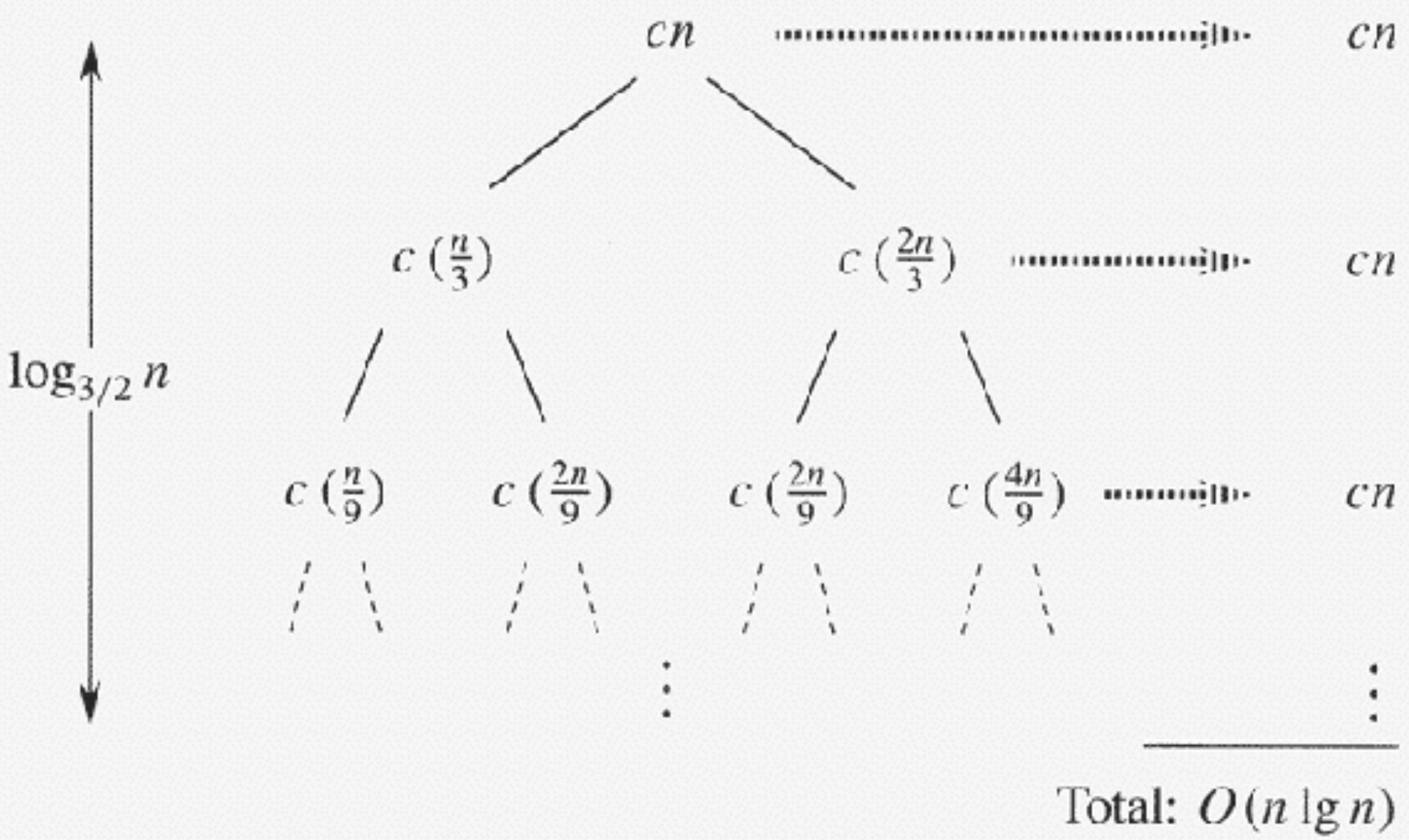


Figure 4.2 A recursion tree for the recurrence $T(n) = T(n/3) + T(2n/3) + cn$.

Solving Recurrences using Master Theorem

Master Theorem:

Let a,b >= 1 be constants, let f(n) be a function, and let

$T(n) = aT(n/b) + f(n)$

1. If $f(n) = O(n^{\log_b a-e})$ for some constant $e>0$, then
   - $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then
   - $T(n) = \Theta(n^{\log_b a} \log n)$
3. If $f(n) = \Omega(n^{\log_b a+e})$ for some constant $e>0$, then
   - $T(n) = \Theta(f(n))$

QuickSort

Page 146, CLR

```
QuickSort(A, p, r)
    if (p < r) then
        q = Partition(A, p, r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)

Partition(A, p, r)
  x = A[r]
  i = p-1
  for j = p to r-1 do
     if A[j] <= x) then
         i++
         exchange(A[i], A[j])
exchange(A[i+1], A[r])
return i+1
```

HeapSort Analysis

For the HeapSort analysis, we need to compute:

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

We know from the formula for geometric series that

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

Differentiating both sides, we get

$$\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$$

Multiplying both sides by x we get

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

Now replace $x = 1/2$ to show that

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \leq \frac{1}{2}$$

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

Invariant for SelectionSort

- An appropriate invariant has a parameter related to the progress of the algorithm (e.g., iteration number)
- An appropriate invariant helps in proving algorithm is correct
- "At the end of iteration p, the p smallest items are in their correct location"

Algorithm Invariants

- Selection Sort
  - iteration k: the k smallest items are in correct location.
- Insertion Sort
  - iteration k: the first k items are in sorted order.
- Bubble Sort
  - 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.
- Shaker Sort
  - In each odd (even) numbered pass, every item that does not have a smaller (larger) item after it, is moved as far up (down) in the list as possible.
  - Iteration k: the k/2 smallest and largest items are in the correct location.

Algorithm Invariants (Cont'd)

- Merge (many lists)
  - Iteration k: the k smallest items from the lists are merged.
- Heapify
  - Iteration with i = k: Subtrees with roots at indices k or larger satisfy the heap property.
- HeapSort
  - Iteration k: Largest k items are in the right location.
- Partition (two sublists)
  - Iteration k (with pointers at i and j): items in locations [1..I] (locations [i+1..j]) are at least as small (large) as the pivot.

Readings for next class

- All sorting algorithms
- QuickSort in particular
- Recurrence relations for divide-and-conquer algorithms
- Substitution method for solving recurrence relations