THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學

COMP2013
Data Structures and Algorithms

Lecture 3
Divide and Conquer

Dr. Jieming Shi
COMP2013

Recap

- Merge sort as an example of Divide and Conquer
- Complexity of merge sort analysis by recursive tree
- O, Ω, Θ-notations and how to prove

Outline

- **Divide and Conquer Algorithms**
  - Binary Search
  - Maximum Subarray Sum
- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - Master method

Guess the Number Game

- Host: pick a secret integer X from 1 to 20
- Guest: guess V as the answer
- Host: “V is too low” / “V is too high” / “V is correct!”

- Brute force:
  - Test each integer in ascending order from 1 to 20
  - Running time $O(n)$

Guess the Number Game

- Host: pick a secret integer X from 1 to 20
- Guest: guess V as the answer
- Host: “V is too low” / “V is too high” / “V is correct!”

- Divide and Conquer:
  - Guess 10 -> too low
    - Indicate that X is between 11 and 20
  - Guess 15 -> too low
    - Indicate that X is between 16 and 20
  - Guess 18
    - Bingo
  - $O(\log n)$
    - Every time halve the problem size

Outline

- Divide and Conquer Algorithms
  - **Binary Search**
  - Maximum Subarray Sum
- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - Master method

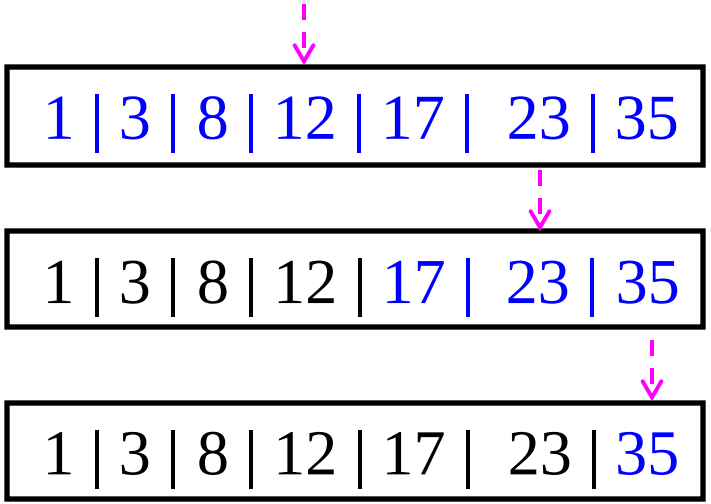
Binary Search

- Find an element in a sorted array

Example: find the key “35”

BinarySearch(Array *A*, *l*, *r*, *key*)

1. $m=\lfloor (l+r)/2 \rfloor$
2. if *A*[*m*] is *key*
3. return *m*
4. if *l* is *r*
5. return -1
6. if *key* < *A*[*m*]
7. return BinarySearch(*A*, *l*, *m-1*, *key*)
8. else
9. return BinarySearch(*A*, *m+1*, *r*, *key*)



Binary Search

- Find an element in a sorted array

```
BinarySearch(Array A, l, r, key)
1.  m=⌊(l + r)/2⌋
2.  if A[m] is key
3.      return m
4.  if l is r
5.      return -1
6.  if key < A[m]
7.      return BinarySearch(A, l, m-1, key)
8.  else
9.      return BinarySearch(A, m+1, r, key)
```

How many subproblem(s) do we break?

- Lines 1-6, O(1)
- Break a problem into 1 subproblem
- The combine cost is O(1)
- The divide cost is O(1)
- $T(n)=T(n/2)+O(1)$
- $T(n) = O(\log n)$

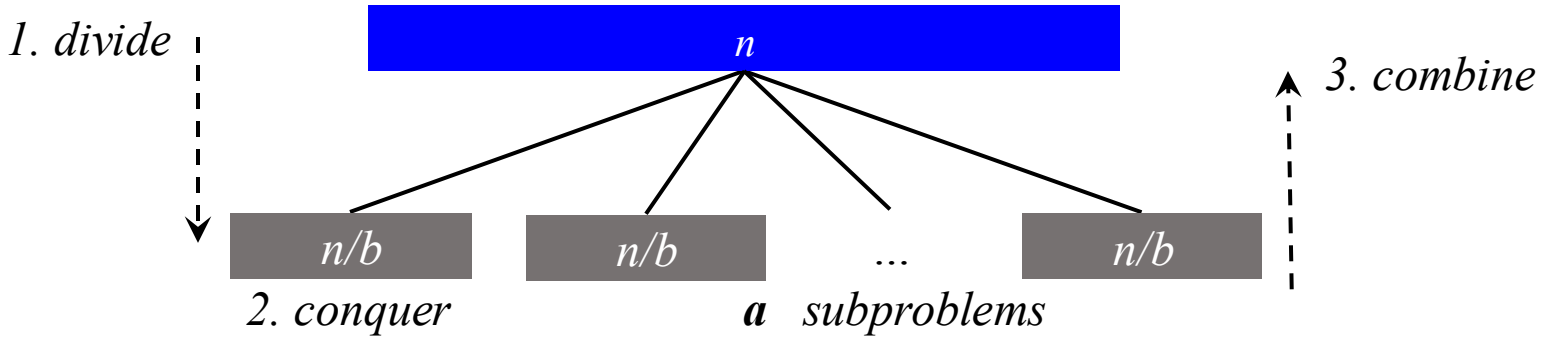
Divide and Conquer

- **Base case:** the problem is small enough to be solved directly without recursing
- **Recursive case**:
  - *Divide* the problem into one or more subproblems that are smaller instances of the same problem.
  - *Conquer* the subproblems by solving them recursively.
  - *Combine* the subproblem solutions to form a solution to the original problem.

Divide and Conquer

- T(n): time complexity of algorithm at input size **n**
  - Divide the problem into ***a*** subproblems
  - Size of each subproblem is **n/b**
    - ***a*** and **b** can be different
    - *It is also possible that subproblems can have different sizes*
  - Suppose that the combine and divide phases take *f*(n) time: $f(n) = D(n) + C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le n_0, \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise.} \end{cases}$$



Outline

- Divide and Conquer Algorithms
  - Binary Search
  - **Maximum Subarray Sum**
- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - Master method

Maximum Subarray Sum

- Problem
  - *Input:* an array A with n elements
  - *Output*: the largest possible sum of $\Sigma_{i=x..y} A[i]$ among all possible subarrays $A[x:y]$
- Example
  - The largest possible sum is 43 in the following array $A[1:16]$
    - Obtained from the subarray $A[8:11]$
- Application scenario
  - $A[i]$ denotes the stock closing price – the stock opening price
  - You can buy once and then sell once. How to earn the most?

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Maximum Subarray Sum: Brute Force

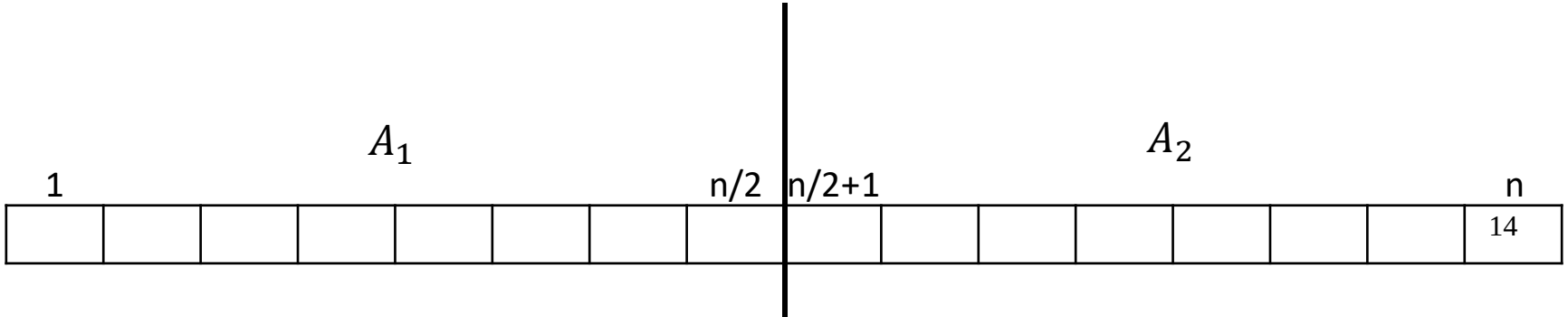
- Consider each start position x
- Find the sum from x to each position y
- Compare the maxSum and update if necessary
- Running time $O(n^2)$

```
BruteForce(Array A, n)
1. maxSum ← -∞
2. for x ← 1 to n
3.     sum ← 0
4.     for y ← x to n
5.         sum ← sum + A[y]
6.         if (sum>maxSum)
7.             maxSum ← sum
8. return maxSum
```

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
x=1:	13	10	-15	5	2	-14	-37	-19	1	-6	6	1	-21	-6	-10	-3
x=2:		-3	-28	-8	-11	-27	-50	-32	-12	-19	-7	-12	-34	-19	-23	-16
x=3:			-25	-5	-8	-24	-47	-29	-9	-16	-4	-9	-31	-16	-20	-13
x=4:	….															

Maximum Subarray Sum: Divide and Conquer

- If dividing A into two subarrays A_1 and A_2, where is the *subarray* with max sum for A?
- It must be either
  - In the left subarray A_1
    - i.e., find the maximum subarray sum in A_1 (a subproblem)
  - In the right subarray A_2,
    - i.e., find the maximum subarray sum in A_2 (a subproblem)
  - Across subarrays A_1 and A_2



Maximum Subarray Sum: Divide and Conquer

- If dividing A into two subarrays A_1 and A_2, where is the *subarray* with max sum for A?
- It must be either
  - In the left subarray A_1
    - i.e., find the maximum subarray sum in A_1 (a subproblem)
  - In the right subarray A_2,
    - i.e., find the maximum subarray sum in A_2 (a subproblem)
  - Across subarrays A_1 and A_2

The solution for A is the max value among the three cases

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

Maximum Subarray Sum: Divide and Conquer

- If dividing A into two subarrays A_1 and A_2, where is the *subarray* with max sum for A?
- It must be either
  - In the left subarray A_1
    - i.e., find the maximum subarray sum in A_1 (a subproblem)
  - In the right subarray A_2,
    - i.e., find the maximum subarray sum in A_2 (a subproblem)
  - Across subarrays A_1 and A_2
- The solution for A is the max value among the three cases

MaxSubarraySum (Array A, l, r)

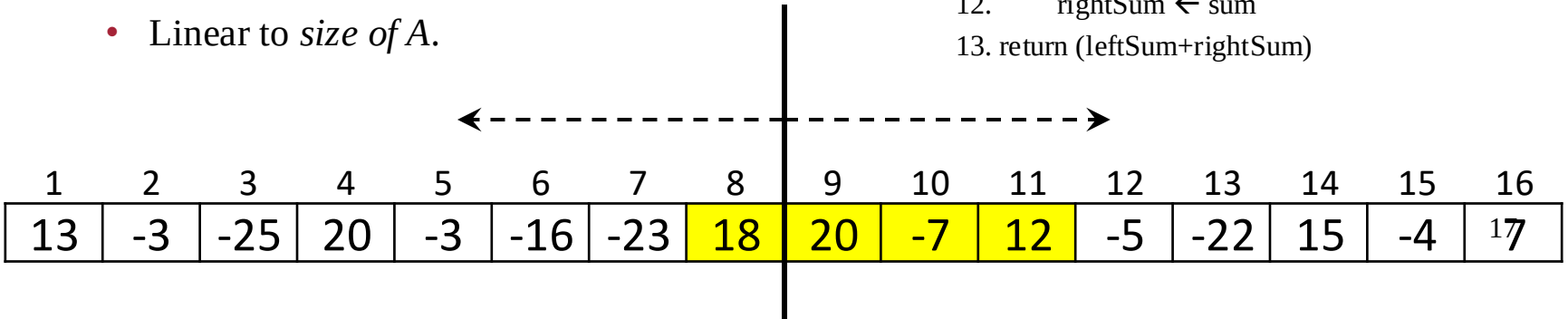
1. if l=r
2. return A[l]
3. else
4. m ← ⌊(l+r)/2⌋
5. leftMaxSum ← MaxSubarraySum(A,l,m)
6. rightMaxSum ← MaxSubarraySum(A,m+1,r)
7. crossMaxSum ← MaxCrossSum(A, l, m, r)
8. return max(leftMaxSum, rightMaxSum, crossMaxSum)

Maximum Subarray Sum: Divide and Conquer

- MaxCrossSum(*A*, *l*, *m*, *r*)
  - Find the max cross sum
  - Sum from mid to left, and keep the max sum obtained
  - Sum from mid to right, and keep the max sum obtained
  - Sum the two max sum values
  - What is the running time of this function?
    - Linear to *size of A*.

MaxCrossSum(*A*, *l*, *m*, *r*)

```
1.  leftSum ← -∞
2.  sum ← 0
3.  for i ← m to l
4.      sum ← sum + A[i]
5.      if (sum>leftSum)
6.          leftSum ← sum
7.  rightSum ← -∞
8.  sum ← 0
9.  for j ← m+1 to r
10.     sum ← sum + A[j]
11.     if (sum>rightSum)
12.         rightSum ← sum
13. return (leftSum+rightSum)
```



Maximum Subarray Sum: Divide and Conquer

- Divide 1 problem into ?
  - Two subproblems, each of size n/2
- The divide cost is ?
  - Constant
- The combine cost is?
  - Line 7 and 8
  - MaxCrossSum has time complexity?
    - O(n)
  - Line 8 has constant time
- How many subproblems?
- T(n)=2T(n/2)+O(n)
- T(n) = O(n log n)

MaxSubarraySum (Array A, l, r)

1. if l=r
2. return A[l]
3. else
4. m ← ⌊(l+r)/2⌋
5. leftMaxSum ← MaxSubarraySum(A,l,m)
6. rightMaxSum ← MaxSubarraySum(A,m+1,r)
7. crossMaxSum ← MaxCrossSum(A, l, m, r)
8. return max(leftMaxSum, rightMaxSum, crossMaxSum)

Outline

- Divide and Conquer Algorithms
  - Binary Search
  - Maximum Subarray Sum
- **Solving Recurrence**
  - Recursive-tree method
  - Substitution method
  - Master method

Algorithmic recurrence

- Recurrence $T(n)$ is algorithmic, if
  - For all $n \leq n_0$, $T(n) = \Theta(1)$
  - For all $n \geq n_0$, each path of recursion terminates in a defined base case within a finite number of recursive invocations
- Whenever a recurrence is stated without an explicit base case, we assume that the recurrence is algorithmic.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0, \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise.} \end{cases}$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Solving recurrences

- Recursion-tree method
- Substitution method
- Master method
- …

Overview of these methods

- Recursion tree method
  - Draw a tree with the time used at each recursion level, then sum up the times at all levels
  - ☹ need to draw a tree and find the sum; some trees are hard to draw
- Substitution method
  - Make a good guess (usually it should be correct)
  - Prove the guess is correct
  - ☹ wrong guess is in vain. Know how to prove.
- Master method
  - Obtain T(n) by using a theorem directly
  - ☹ not applicable in some cases

Recurrences

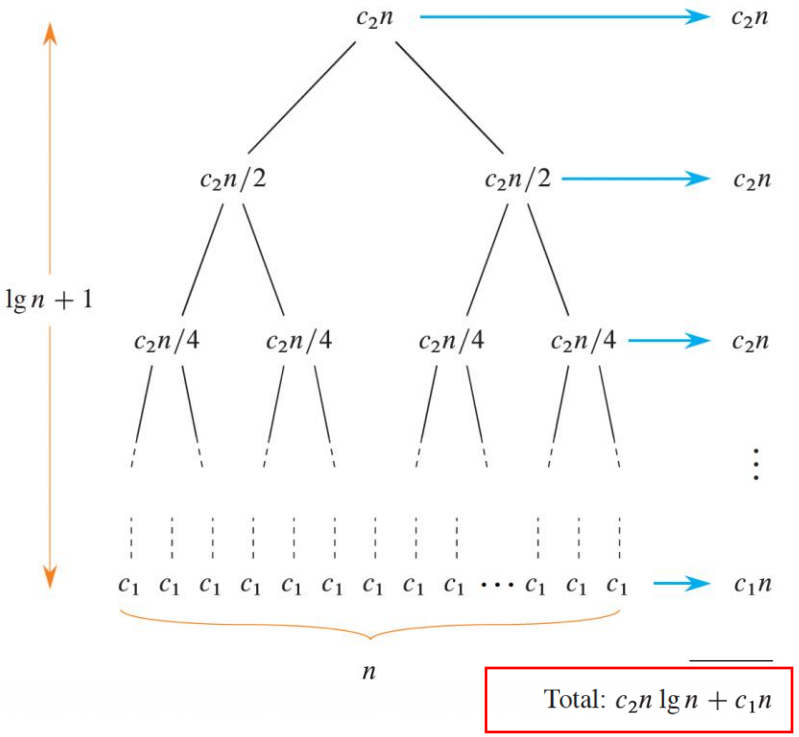
- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Express the solution using Θ-notation (using O is also ok)
- $T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Right side is an upper bound of $T(n)$, and express the solution using O-notation
- $T(n) \geq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - Right side is a lower bound of $T(n)$, and express the solution using Ω-notation

Outline

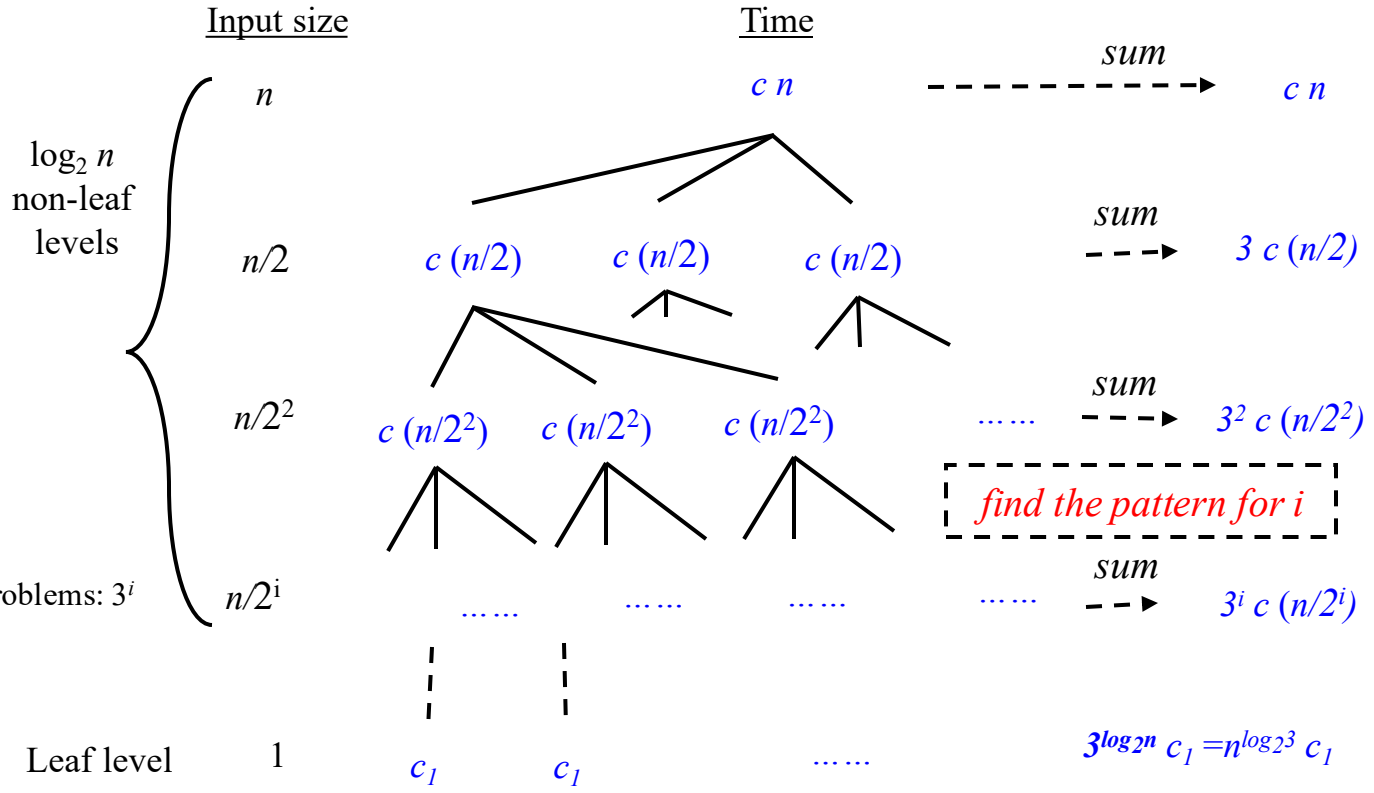
- Divide and Conquer Algorithms
  - Binary Search
  - Maximum Subarray Sum
- Solving Recurrence
  - **Recursive-tree method**
  - Substitution method
  - Master method

Recursion-tree method

- We have showed merge sort has running time $\Theta(n\ lg\ n)$ by recursion tree
- $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$



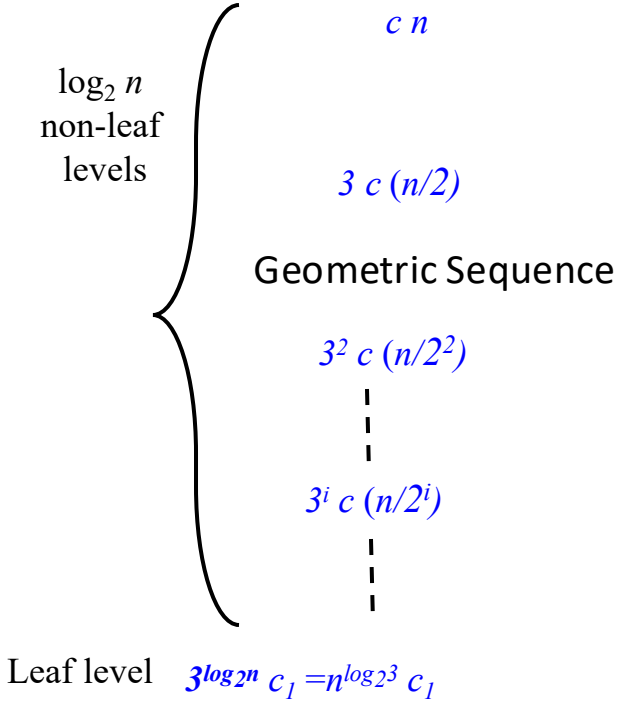
Recursion Tree for: T(n) = 3 T(n/2) + c n



- Number of subproblems: 3^i

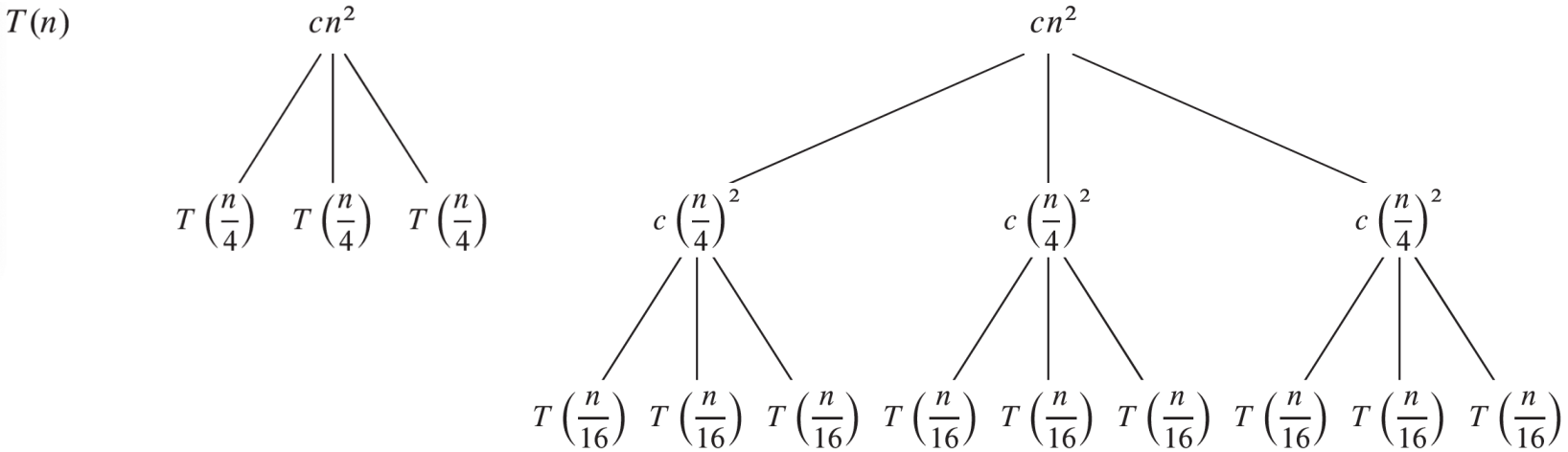
Recursion Tree for: T(n) = 3 T(n/2) + c n

- Time at the **leaf** level: $c_1 n^{\log_2 3}$
- Number of **non-leaf** levels L: $\log_2 n$
- Consider the non-left level i, i from 0 to $\log_2 n - 1$
  - level i=0 is the top level
  - Input size of a subproblem: $n/2^i$; Number of subproblems: 3^i
  - Divide&Combine time of a subproblem: $c\ (n/2^i)$
  - Total time at this level: $3^i\ c\ (n/2^i)$
- Time of non-leaf levels
  - $\sum_{i=0}^{\log_2 n - 1} 3^i c \left(\frac{n}{2^i}\right)$
  - $= 2cn\left(\left(\frac{3}{2}\right)^{\log_2 n} - 1\right) = 2cn\left((n)^{\log_2 \frac{3}{2}} - 1\right) = 2cn^{\log_2 3} - 2cn$
- Total time is $2cn^{\log_2 3} - 2cn + c_1 n^{\log_2 3}$
- $\Theta(n^{\log_2 3})$
  - $\log_2 3$ *is about 1.585*



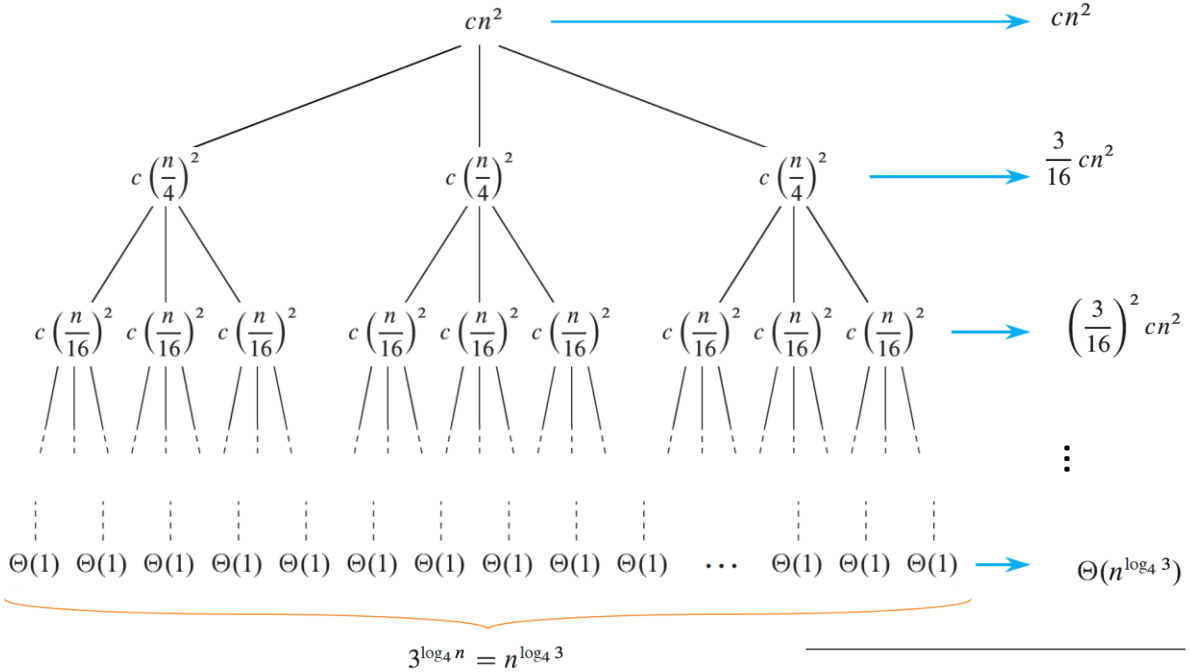
Recursion-tree method

- $T(n) = 3T(n/4) + \Theta(n^2)$
- $T(n) = 3T(n/4) + cn^2$, where $c > 0$



Recursion-tree method

- $T(n) = 3T(n/4) + \Theta(n^2)$
- $T(n) = 3T(n/4) + cn^2$, where $c > 0$
- $T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$



Geometric Sequence (internal levels)

Recursion-tree method

- $T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta\left(n^{\log_4 3}\right)$
- $= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$
- $< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right)$
- $= \frac{1}{1-\frac{3}{16}} cn^2 + \Theta\left(n^{\log_4 3}\right)$
  - ($\Theta\left(n^{\log_4 3}\right)$ is a lower-order term than n^2 since $\log_4 3<1<2$)
- $= O(n^2)$

Outline

- Divide and Conquer Algorithms
  - Binary Search
  - Maximum Subarray Sum
- Solving Recurrence
  - Recursive-tree method
  - **Substitution method**
  - Master method

Substitution method

- Given a recurrence
  - 1. Guess the form of the solution using symbolic constants.
    - Make a good guess about its non-recursive form
  - 2. Use mathematical *induction* to show that the solution works, and find the constants.
    - substitute the guessed solution for the function on smaller values
- If you want to get a Θ-bound
  - Better to first prove an O-bound and then prove an Ω-bound, and then you get a Θ-bound if O-bound and Ω-bound have the same rate of growth
- Use substitution to establish either an upper or a lower bound on a recurrence.
  - Usually best not to try to do both at the same time

Substitution method

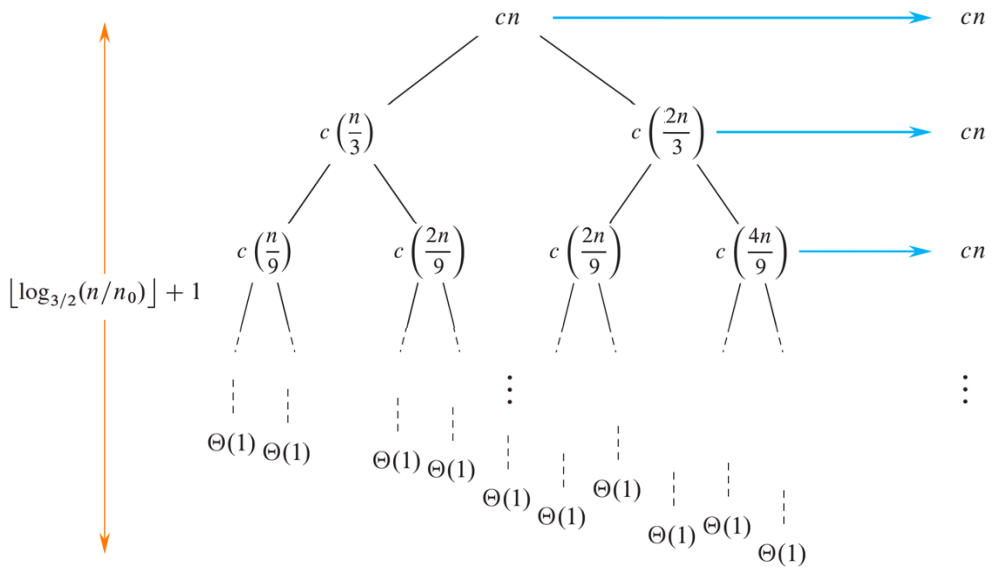
- $T(n) = 3T(n/4) + \Theta(n^2)$
- $T(n) = 3T(n/4) + cn^2$
- Guess $T(n) = O(n^2)$, i.e., $T(n) \leq dn^2$ for some $d > 0$
- Proof (by induction):
  - Assume $T(n') \leq dn'^2$ holds for any $n' < n$. We need to prove this holds for *n*.
  - Induction case (For $n > n_0$)
    - $T(n) = 3T(n/4) + cn^2$
    - $\leq 3d\left(\frac{n}{4}\right)^2 + cn^2 = \frac{3}{16}dn^2 + cn^2$
    - $\leq dn^2$, if $d \geq \frac{16}{13}c$
  - Base case (For $1 \leq n \leq n_0$)
    - implicit base case $T(n) = \Theta(1)$, i.e., $T(n) = c_1$
    - We can choose d to be large enough (dominating c_1) such that $T(n) \leq d \leq dn^2$ holds,
  - Completing the proof.

A challenging example

- $T(n) = T(n/3) + T(\frac{2n}{3}) + \Theta(n)$
  - Get its time complexity in *O-notation*
  - Two subproblems not in the same size
- Method 1: use a *rough recursive tree* to make a guess, and then prove it by *substitution*
- Method 2: use recursive tree to solve it

Make a guess by drafting a recursion tree

- $T(n) = T(n/3) + T\left(\frac{2n}{3}\right) + cn$
- The tree is not balanced
  - Left paths are shorter
  - Right most path is the longest
- How many levels?
  - (Hint: look at the longest path)
  - The right problem has size $\frac{2}{3}$ of its parent problem
  - $(\frac{2}{3})^h n < n_0 \leq \left(\frac{2}{3}\right)^{h-1} n$
  - $h = \left\lfloor \log_{\frac{3}{2}} \frac{n}{n_0} \right\rfloor + 1$
  - $h = \Theta(\log n)$
- Cost of Intermediate levels:
  - Each level has cost at most cn
  - $O(n \log n)$ for all internal nodes
- Cost of Leaf nodes:
  - **How many leaves are there**?
  - Hard to estimate/count (see textbook about how to get a tight upper bound about number of leaf nodes.)
- We make a guess about running time without estimating number of leaf nodes
  - $O(n \log n)$?
  - Prove it by substitution



Then prove it by substitution

- $T(n) = T(n/3) + T(\frac{2n}{3}) + \Theta(n)$
  - We guess it is $O(n\log n)$
- Proof:
  - Assume $T(n') \leq d\, n' \log n'$ holds for any $n' < n$ and for some constant $d>0$. We need to prove this holds for n.
  - Induction case:
    - $T(n) = T(n/3) + T\left(\frac{2n}{3}\right) + cn \leq d\frac{n}{3}\log\frac{n}{3} + d\frac{2n}{3}\log\frac{2n}{3} + cn$
    - (you finish the details here to find a *d* such that *T(n) is O(nlogn)*
  - Base case (For $1 \leq n \leq n_0$)
    - implicit base case $T(n) = \Theta(1)$, i.e., $T(n) = c_1$
    - (you finish the details here to find a *d* such that *T(n) is O(nlogn)*

Outline

- Divide and Conquer Algorithms
  - Binary Search
  - Maximum Subarray Sum
- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - **Master method**
    (next lecture)

- Thank you!