Data Structures

The master method

- Assume that T(n) and f(n) be as defined previously
- The master theorem
  - If there is a small constant ε > 0, s. t. f(n) is O($n^{log_b a - ε}$), then T(n) is θ($n^{log_b a}$)
  - If there is a small constant k ≥ 0, s. t. f(n) is θ($n^{log_b a}\ log^k n$), then T(n) is θ($n^{log_b a}\ \ log^{k+1} n$) (log n is $\log_2$n)
  - If there are small constants ε > 0 and δ < 1, s. t. f(n) is Ω($n^{log_b a + ε}$) and af(n/b) ≤ δf(n), for n ≥ d, then T(n) is θ($f(n)$)

The master method

- Ex: Consider the recurrence

  T(n) = 4 T(n/2) + n

- $n^{log_b a}$ = $n^{log_2 4}$ = n^2
- f(n) is $O(n^{2-\varepsilon})$ for $\varepsilon = 1$
- T(n) is $\theta(n^2)$

The master method

- Ex:

  T(n) = 2 T(n/2) + n log n

- $n^{log_b a}$ = $n^{log_2 2}$ = n
- f(n) is n log n; with k = 1, f(n) is θ(n log n)
- T(n) is θ(n $\log^2$ n)

The master method

- Ex:

  T(n) = T(n/3) + n

- $n^{log_b a}$ = $n^{log_3 1}$ = n^0 = 1
- f(n) is $\Omega(n^{0+\varepsilon})$ for ε = 1; a f(n/b) = n/3 = (1/3)f(n)
- T(n) is θ(n)

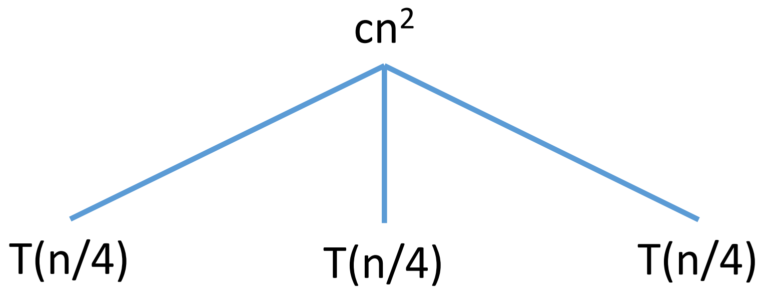
The master method

- Ex 1: $T(n) = 2^n T(n/2) + n^n$
- Ex 2: $T(n) = 2T(n/2) + n/\log n$

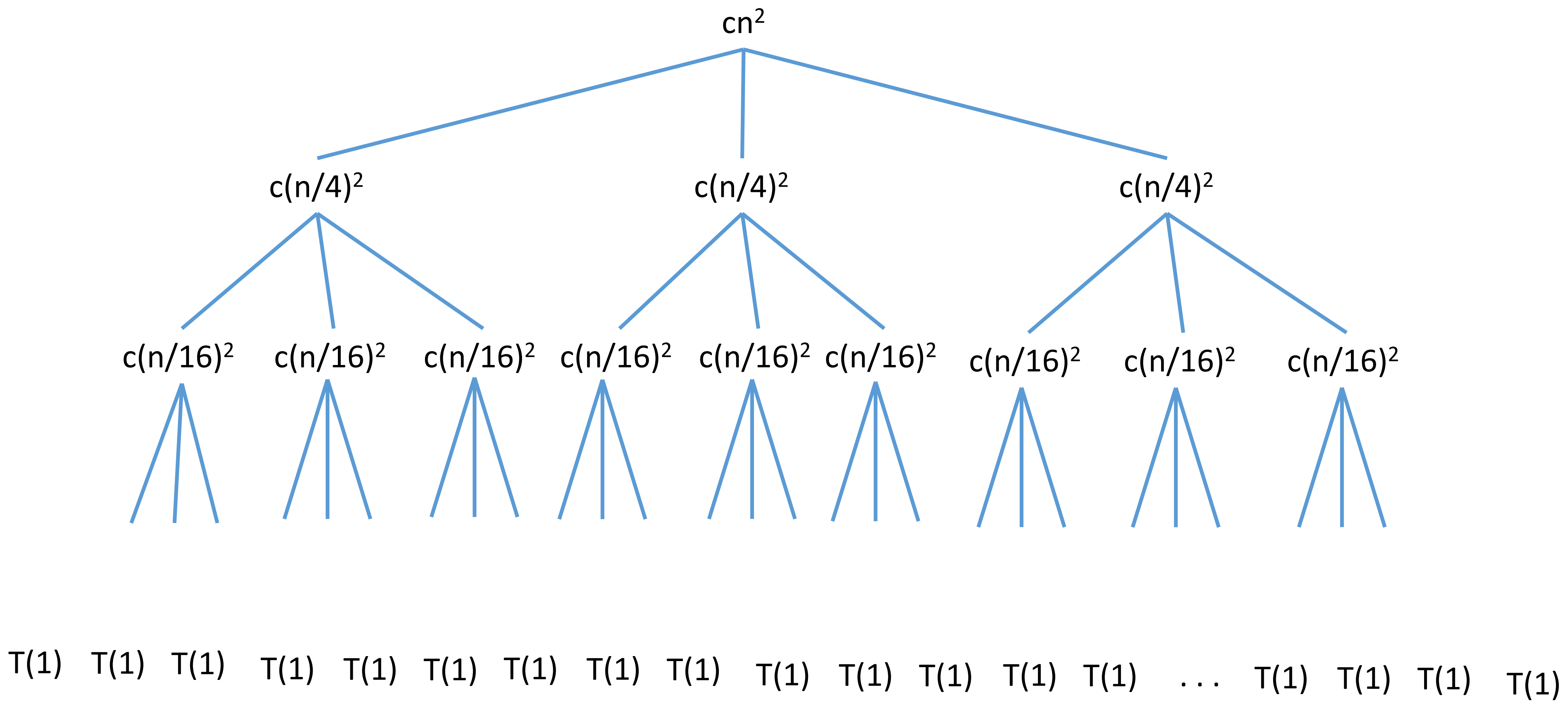
The recursive tree method

- Ex: consider the recurrence (n is an exact power of 4):

$$T(n) = \begin{cases} b & \text{if } n < 4, \\ 3T(n/4) + cn^2 & \text{otherwise} \end{cases}$$



The recursive tree method



The recursive tree method

- The height of tree is: $i = \log_4 n$
- $\log_4 n + 1$ levels at depths 0, 1, 2, . . ., $\log_4 n$
- Each level has three times more nodes than previous level
- A node at depth i has cost: $c(n/4^i)^2$
- Cost per depth: $3^i c(n/4^i)^2 = (3/16)^i cn^2$
- The bottom level has $3^{\log_4 n} = n^{\log_4 3}$ nodes and each node contributes cost T(1), hence the total cost at bottom level: $\theta(n^{\log_4 3})$

The recursive tree method

$$T(n) = cn^2 + \left(\frac{3}{16}\right) cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \theta(n^{\log_4 3})$$

$$= \frac{1}{1-\left(\frac{3}{16}\right)} cn^2 + \theta(n^{\log_4 3})$$

$$= (16/13)\ cn^2 + \theta(n^{\log_4 3})$$

$$= O(n^2)$$

Quick sort

- Though its worst case running time is $\theta(n^2)$, the expected running time in $\theta(n \log n)$
- In place sorting algorithm

Quick sort

- Uses divide and conquer approach
- Divide: partition (rearrange) the given sequence A[p. . . r] into two sub-sequences A[p . . .q-1] and A[q+1 . . . r] such that all elements in A[p . . .q-1] are less than or equal to A[q] and all elements in A[q+1 . . . r] are more than or equal to A[q]; compute the index of pivot
- Conquer: Recursively sort the two sub-sequences
- Combine: Since the subarrays are already sorted no work need to be done

Partitioning Algorithm

Algorithm Partition(A, p, r)

{Ensure that the algorithm works within the bounds of the input array}

```
    x ← A[r]
    i ← p-1
    j ← r+1
    for(;;)
        while(A[++i] ≤ x) { }
        while(A[--j] ≥ x) { }
        if(i < j)
            exchange(A[i], A[j])
        else
            break
    exchange(A[i], A[r])
    return i
```

	8	1	4	9	0	3	5	2	7	6	
i											j

Partitioning: Examples

	8	1	4	9	0	3	5	2	7	6	
i											j

8	1	4	9	0	3	5	2	7	6
↑							↑		
i							j		

2	1	4	9	0	3	5	8	7	6
↑							↑		
i							j		

2	1	4	9	0	3	5	8	7	6
			↑			↑			
			i			j			

2	1	4	5	0	3	9	8	7	6
			↑			↑			
			i			j			

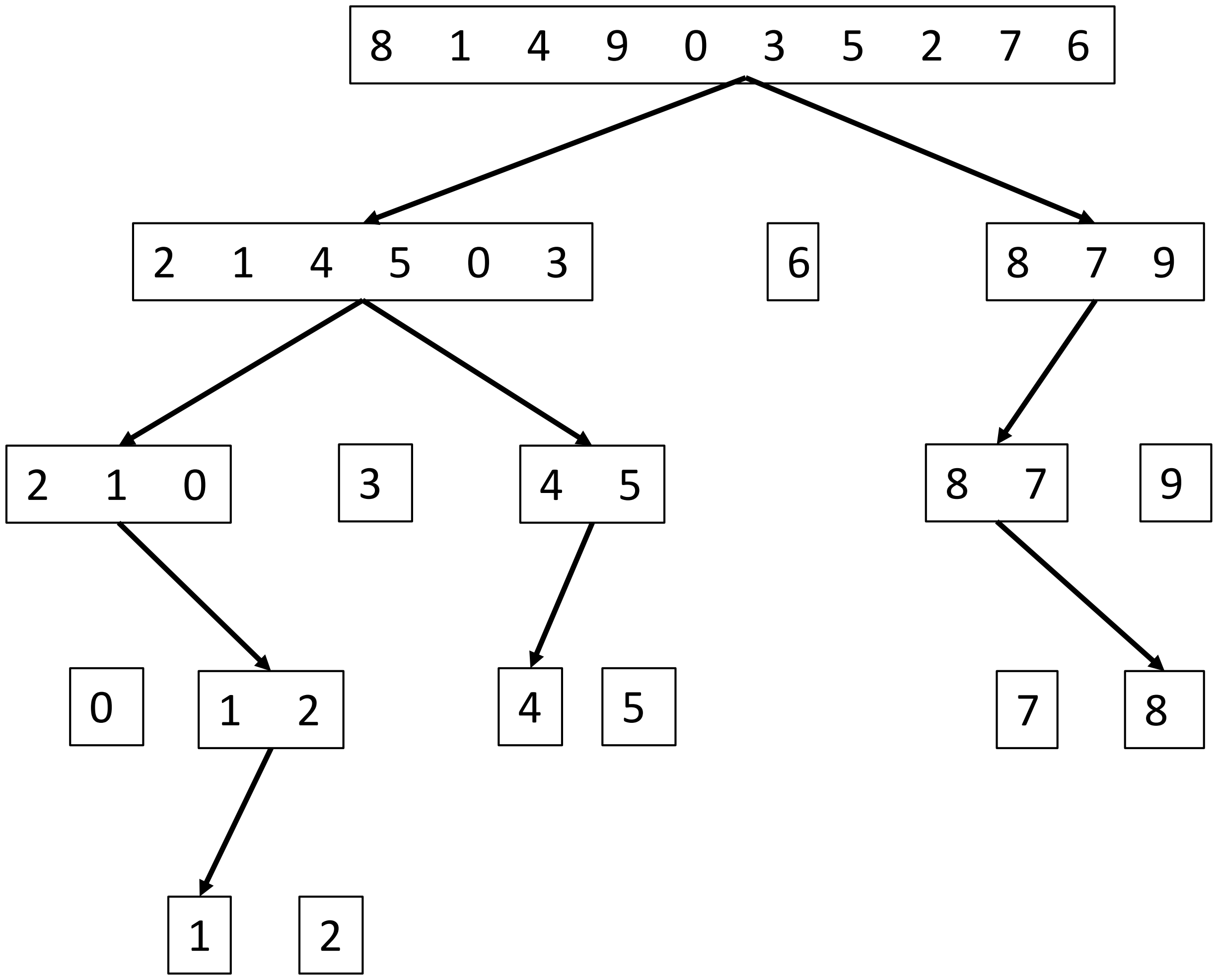
2	1	4	5	0	3	9	8	7	6
					↑	↑			
					j	i			

2	1	4	5	0	3	6	8	7	9
						↑			
						i			

Quick sort algorithm

```
Algorithm Quick_Sort(A, p, r)
        if(p < r)
                q = Partitition(A, p, r)
                Quick_Sort(A, p, q-1)
                Quick_Sort(A, q+1, r)
```

Quick sort example



Analysis of Partitioning Algorithm

Algorithm Partition(A, p, r)

{Ensure that the algorithm works within the bounds of the input array}

```
x ← A[r]
i ← p-1
j ← r+1
for(;;)
    while(A[++i] ≤ x) { }
    while(A[--j] ≥ x) { }
    if(i < j)
        swap(A[i], A[j])
    else
        break
exchange(A[i], A[r])
return i
```

Analysis of Quick Sort: Worst case

- Assumption: All elements are distinct
- Running time depends upon how the sub-sequences are distributed
- Consider a sequence with n elements
- Partitioning produces one sub-problem with “n-1” elements and another with “0” elements
- This unbalanced partitioning arises at each recursive call

$$T(n) = T(n\text{-}1) + T(0) + \theta(n)$$

- T(n) is $\theta(n^2)$