CS302
Design and Analysis of Algorithm

Week-3

Comparisons of different sorting algorithms

Bubble Sort	**Insertion Sort**	**Selection Sort**
$\Theta(n^2)$ comparisons	$\Theta(n^2)$ comparisons	$\Theta(n^2)$ comparisons
$\Theta(n^2)$ swaps	$\Theta(n^2)$ writes	$\Theta(n)$ swaps
Adaptive: $O(n)$ running time when nearly sorted (Best case running time)	Adaptive: $O(n)$ running time when nearly sorted (Best case running time)	Not adaptive $\Theta(n^2)$ running time when nearly sorted (Best case running time)

Recurrence Relations

What is a recurrence relation?

- A recurrence relation, T(n), is a recursive function of integer variable n.
- Like all recursive functions, it has both recursive case and base case.
- Example:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n / 2) + bn & \text{if } n > 1 \end{cases}$$

- The portion of the definition that does not contain T is called the **base case** of the recurrence relation
- The part that contains T is called the **recurrent or recursive case**.

Forming Recurrence Relations

- For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- Example 1: Write the recurrence relation for the following method.

```
public void f (int n) {
if (n > 0) {
        System.out.println(n);
        f(n-1);
    }
}
```

- The base case is reached when n == 0. The method performs one comparison. Thus, the number of operations when n == 0, T(0), is some constant a.
- When n > 0, the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter n – 1.
- Therefore the recurrence relation is:

$$T(0) = a \qquad \text{for some constant a}$$
$$T(n) = b + T(n-1) \qquad \text{for a constant b}$$

Forming Recurrence Relations

Example 2: Write the recurrence relation for the following method.

```
public int g(int n) {
    if (n == 1)
        return 2;
    else
        return 3 * g(n / 2) + g( n / 2) + 5;
}
```

- The base case is reached when n == 1. The method performs one comparison and one return statement. Therefore, T(1), is constant **c**.
- When **n > 1**, the method performs **TWO** recursive calls, each with the parameter n **/ 2,** and some constant # of basic operations.
- Hence, the recurrence relation is:

$$T(1) = c \qquad \text{for some constant } c$$
$$T(n) = b + 2T(n/2) \qquad \text{for a constant } b$$

Solving Recurrence Relations

- Methods to solve recurrence relations that represent the running time of recursive methods:
    - Iteration method (*unrolling and summing)*
    - Recursion tree method
    - Master method

Iteration Method

Iteration Method

- Back Substitution method
- unrolling and summing
- Iteration consist of repeatedly substituting the recurrence into itself to obtain an summation expression

Analysis Of Recursive Factorial method

- Example: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

```
long factorial (int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial (n – 1);
}
```

$T(0) = c$

$T(n) = b + T(n - 1)$

$= b + b + T(n - 2)$

$= b + b + b + T(n - 3)$

$\ldots$

$= kb + T(n - k)$

When $n-k = 0$, we have: $n=k$

$T(n) = nb + T(n - n)$

$= bn + T(0)$

$= bn + c.$

Therefore method factorial is $O(n)$.

Analysis Of Recursive Binary Search

```
public int binarySearch (int target, int[] array,
                              int low, int high) {
   if (low > high)
      return -1;
   else {
      int middle = (low + high)/2;
      if (array[middle] == target)
         return middle;
      else if(array[middle] < target)
         return binarySearch(target, array, middle + 1, high);
      else
         return binarySearch(target, array, low, middle - 1);
     }
}
```

- The recurrence relation for the running time of the method is:

$T(1) = a$ if $n = 1$ (one element array)

$T(n) = T(n / 2) + b$ if $n > 1$

Analysis Of Recursive Binary Search

Expanding:

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

= [T(n / 4) + b] + b = T (n / 2^2) + 2b

= [T(n / 8) + b] + 2b = T(n / 2^3) + 3b

= ....

= T(n / 2^k) + kb

When n / 2^k = 1 □ n = 2^k □ k = log n, we have:

T(n) = T(1) + b log n

= a + b log n

Therefore, Recursive Binary Search is **O(log n)**

Task

$$T(n) = T(n-1) + bn$$

$$T(0) = c$$

Sol

$T(n) = T(n-1) + bn$

$= (T(n-2) + b(n-1)) + bn \quad = T(n-2) + bn+bn-b$

$= T(n-3) + b(n-2)) +b(n-1) + b(n) =T(n-3)+3bn-2b-b$

$= T(n-4) + b(n-3)+ b(n-2)) +b(n-1) + b(n) \quad = T(n-4)+4bn-3b-2b-b$

$=T(n-4)+4bn-b[3+2+1]$

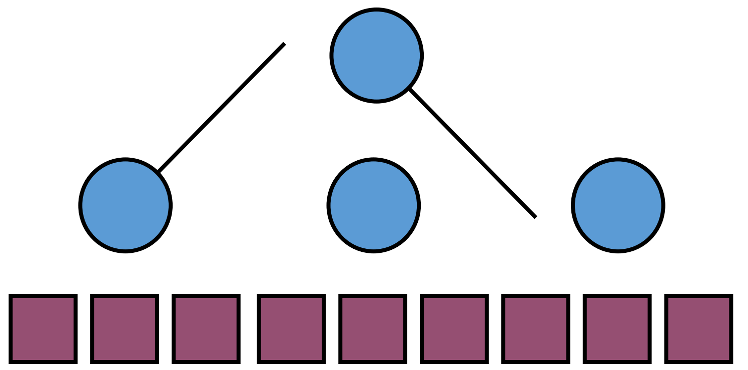
$= T(n-k)+kbn-b\{(k-1) (k)/2\}$

$n-k=0 \Rightarrow n=k$

$= T(0)+bn^2+b[(n-1)(n)/2]$

$=c+ bn^2+b[(n-1)(n)/2]$

$=O(n^2)$



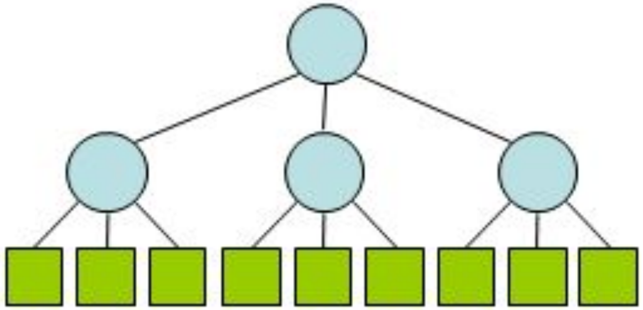
Recursion Tree Method

Recursion tree method

- Making good guess is sometimes difficult with the substitution method,
- Recursion Tree can be used to devise a good guess.
- solving recurrences
  - expanding the recurrence into a tree
  - summing the cost at each level (cost of leaf+internal nodes)
- Difference between depth and height of node

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets S_1, S_2, …
  - Recur: solve the sub problems recursively
  - Conquer: combine the solutions for S_1, S_2, …, into a solution for S
- The base case for the recursion are sub problems of constant size
- Analysis can be done using **recurrence equations**



Recursion tree

.Solve $T(n) = 2T(n/2) + c\,n,$ where $c > 0$ is constant

$$T(n)$$

Here Tree nodes represent costs incurred at various levels of the recursion

Recursion tree

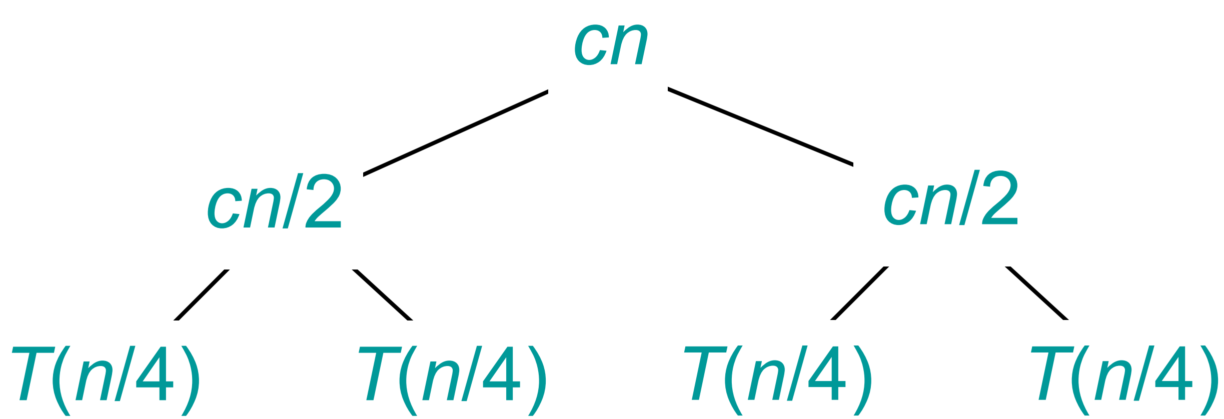
.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant

$$cn$$

$$T(n/2) \qquad\qquad T(n/2)$$

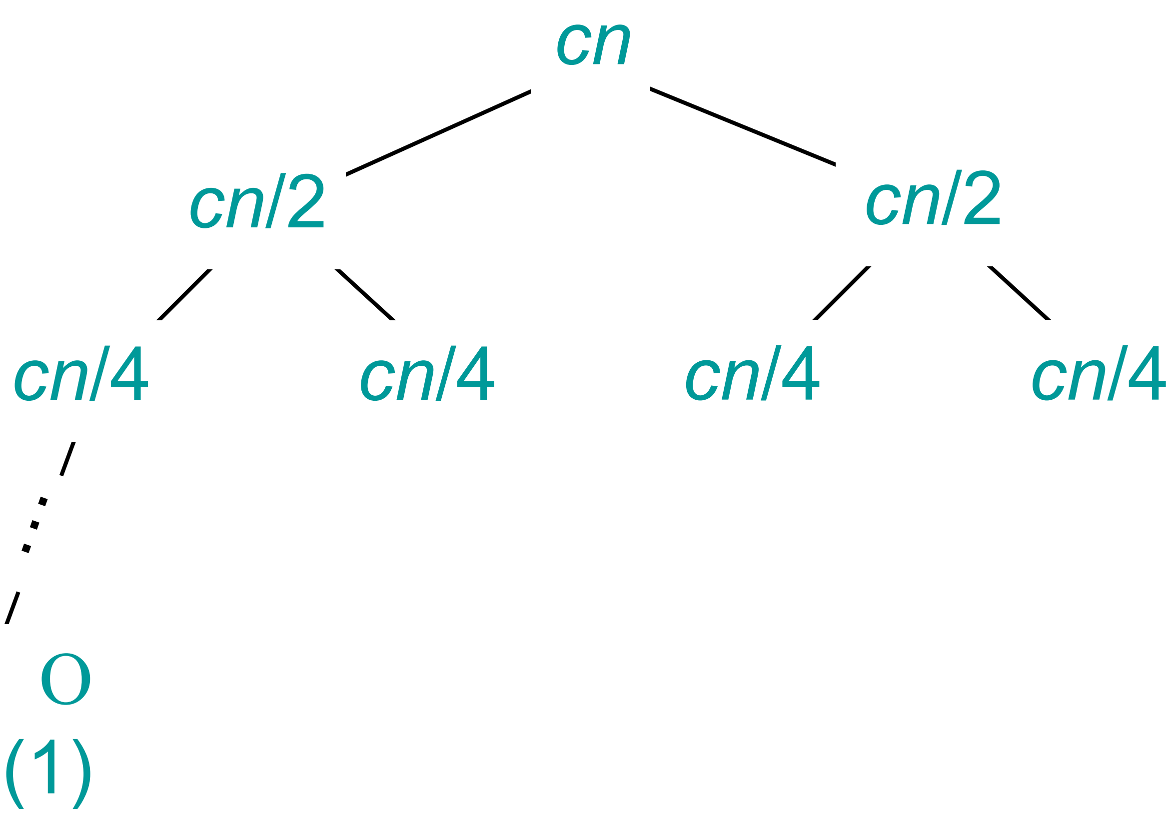
Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



2^1

Determining depth/height of tree

$$Level\ 0 = \frac{n}{(2)^0}$$

$$Level\ 1 = \frac{n}{(2)^1}$$

$$Level\ 2 = \frac{n}{(2)^2}.$$

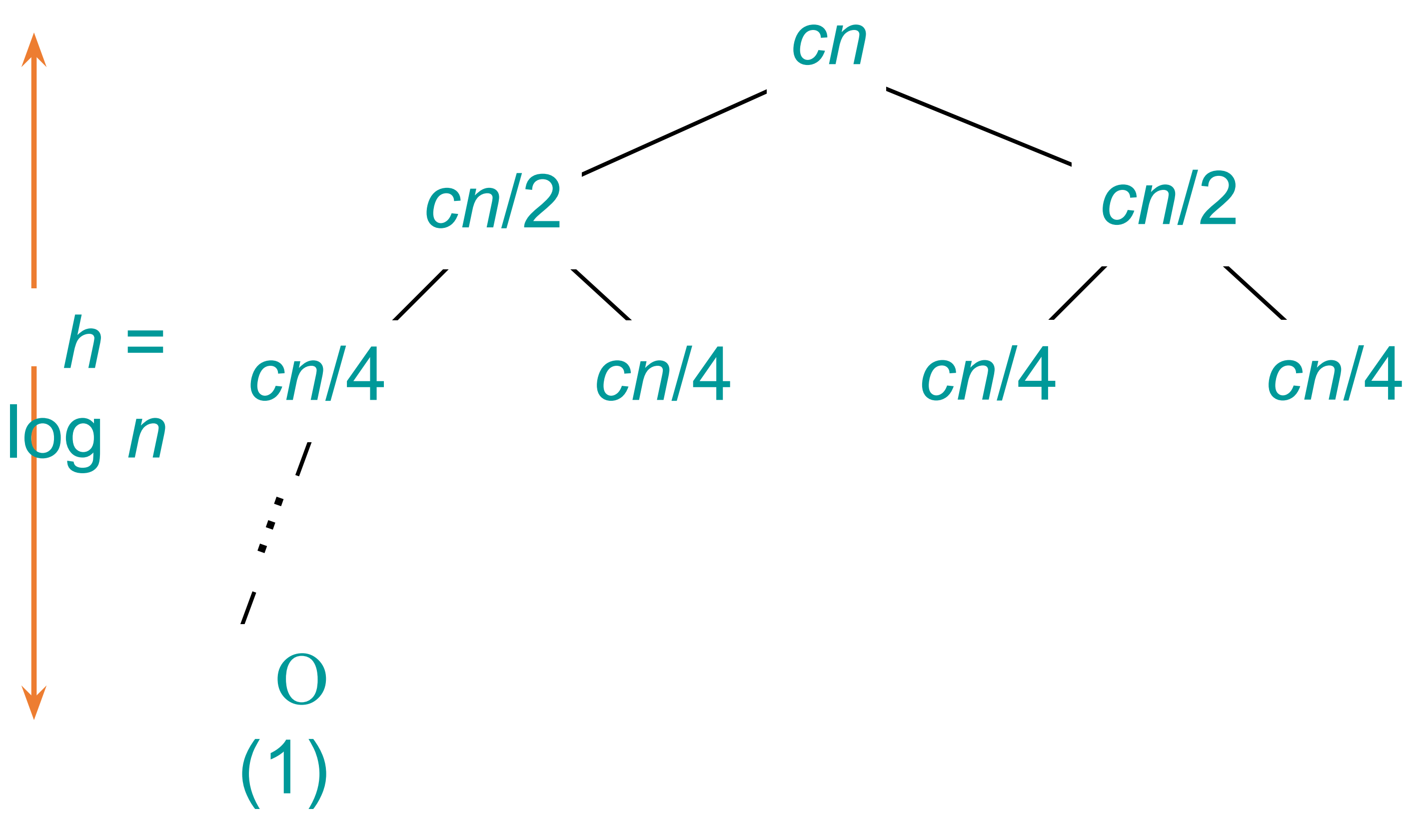
$$Level\ \mathrm{i} = \frac{n}{(2)^{\mathrm{i}}}$$

$$Level\ \mathrm{h} = \frac{n}{(2)^{\mathrm{h}}}$$

$$T(1) = \mathrm{T}\left(\frac{n}{(2)^{\mathrm{h}}}\right) \Rightarrow \frac{n}{(2)^{\mathrm{h}}} = 1 \Rightarrow n=(2)^{\mathrm{h}} \Rightarrow h=\log_2 n$$

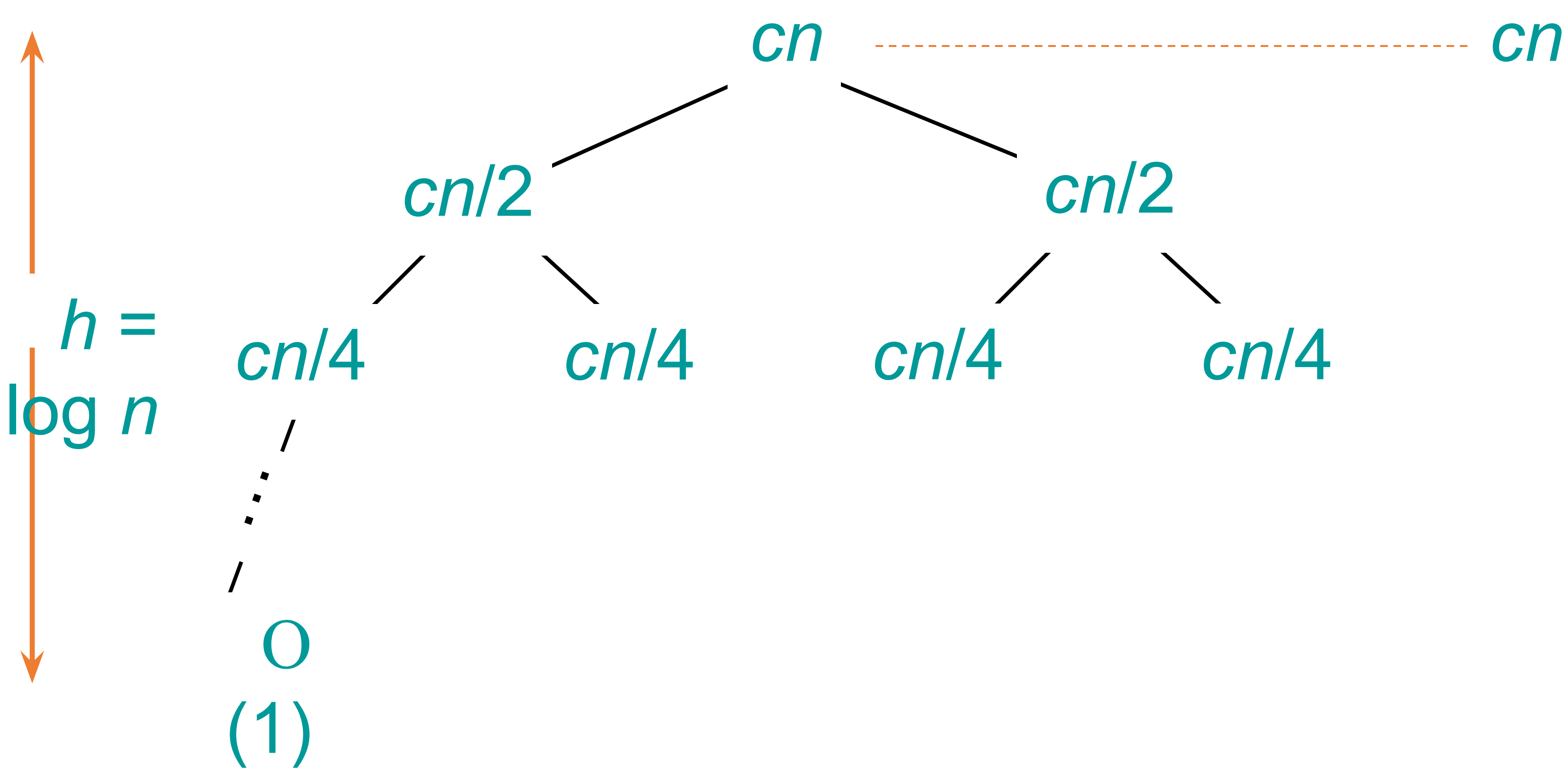
Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



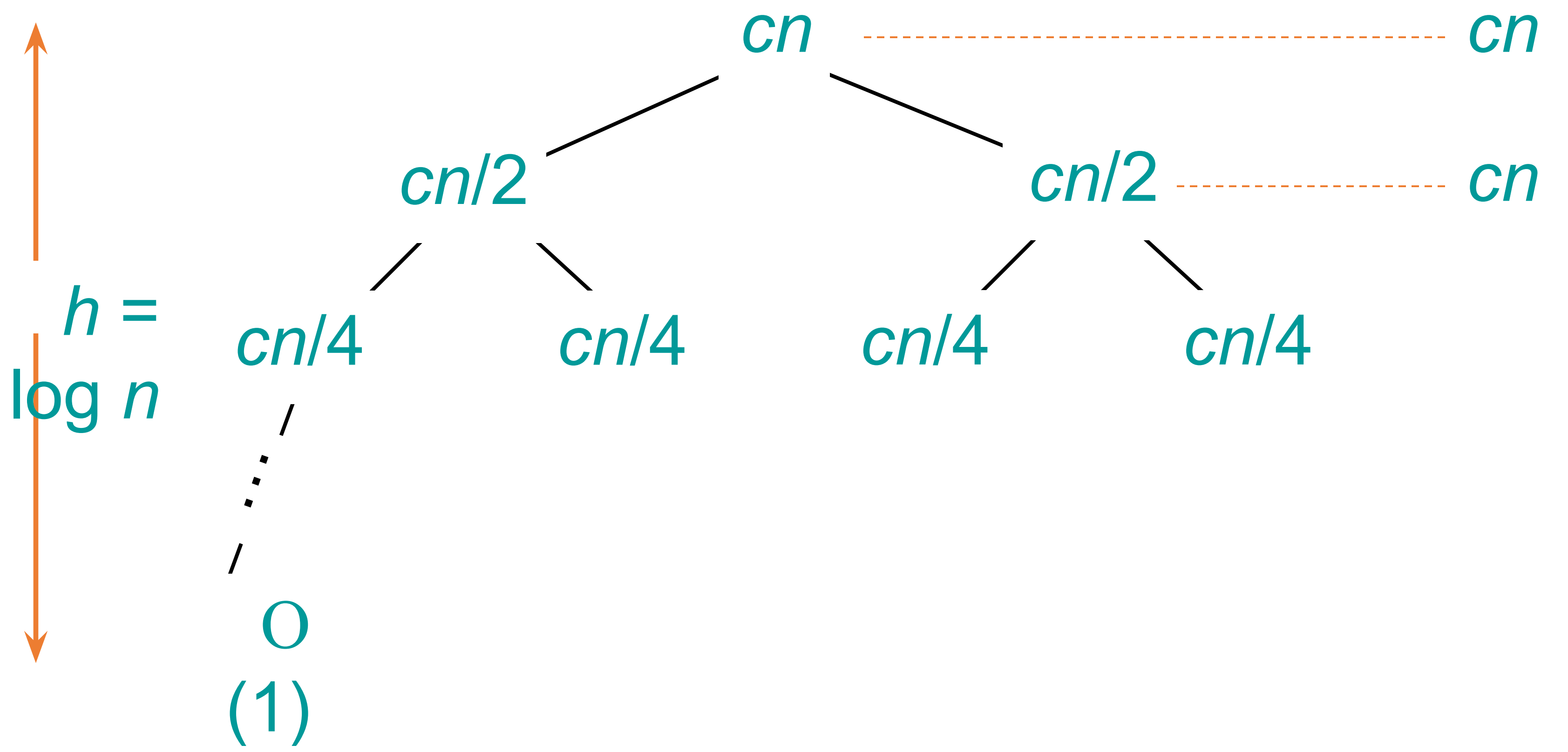
Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



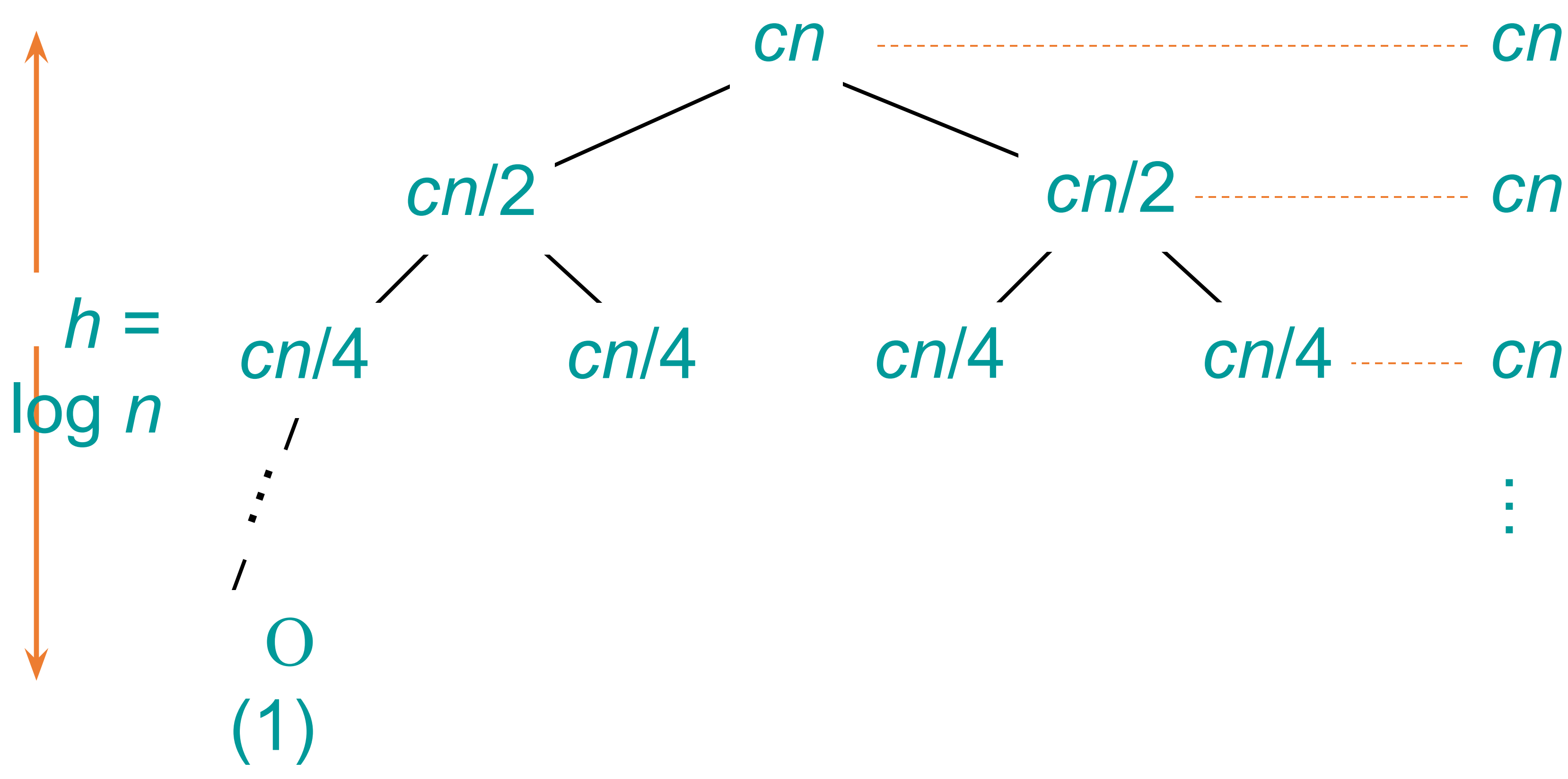
Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



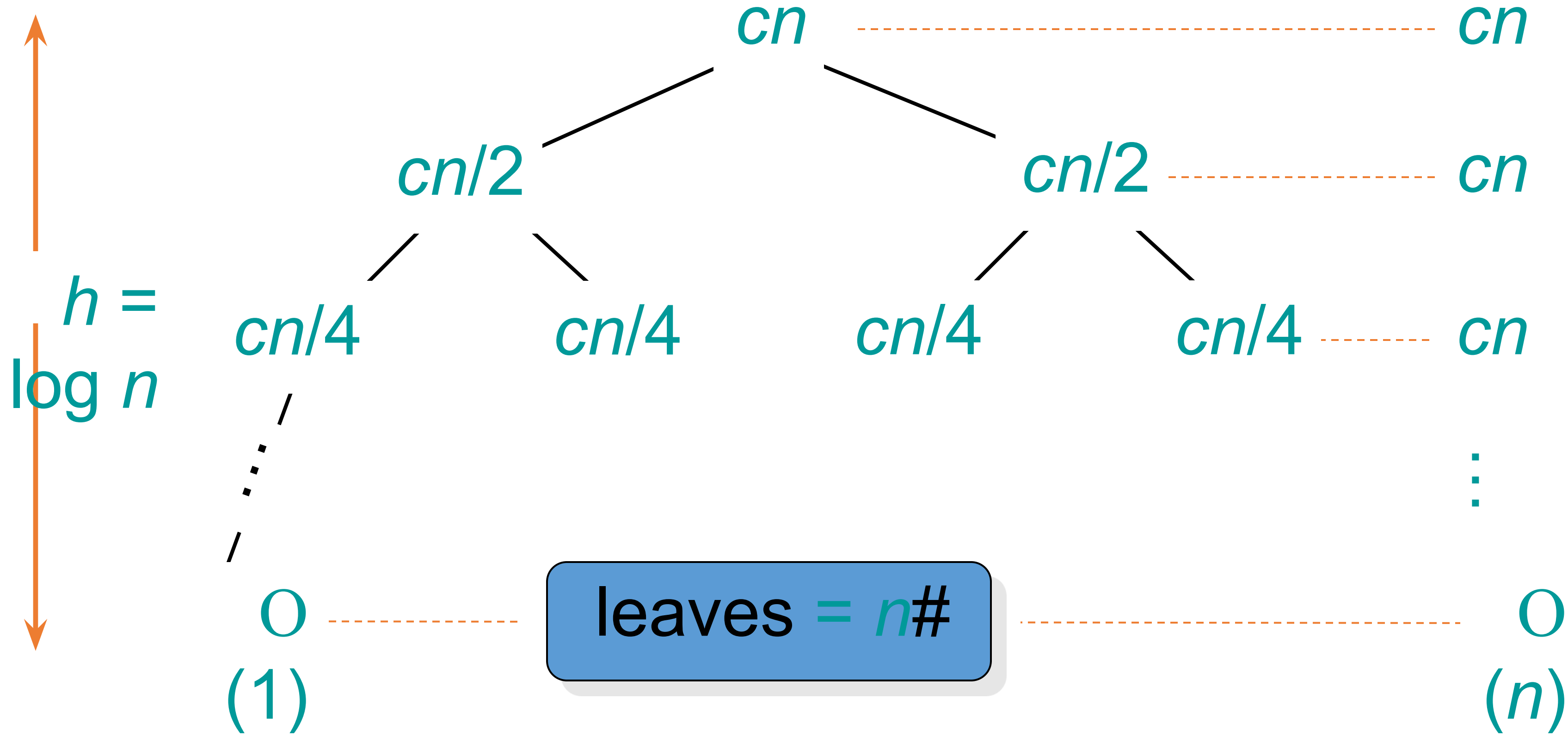
Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



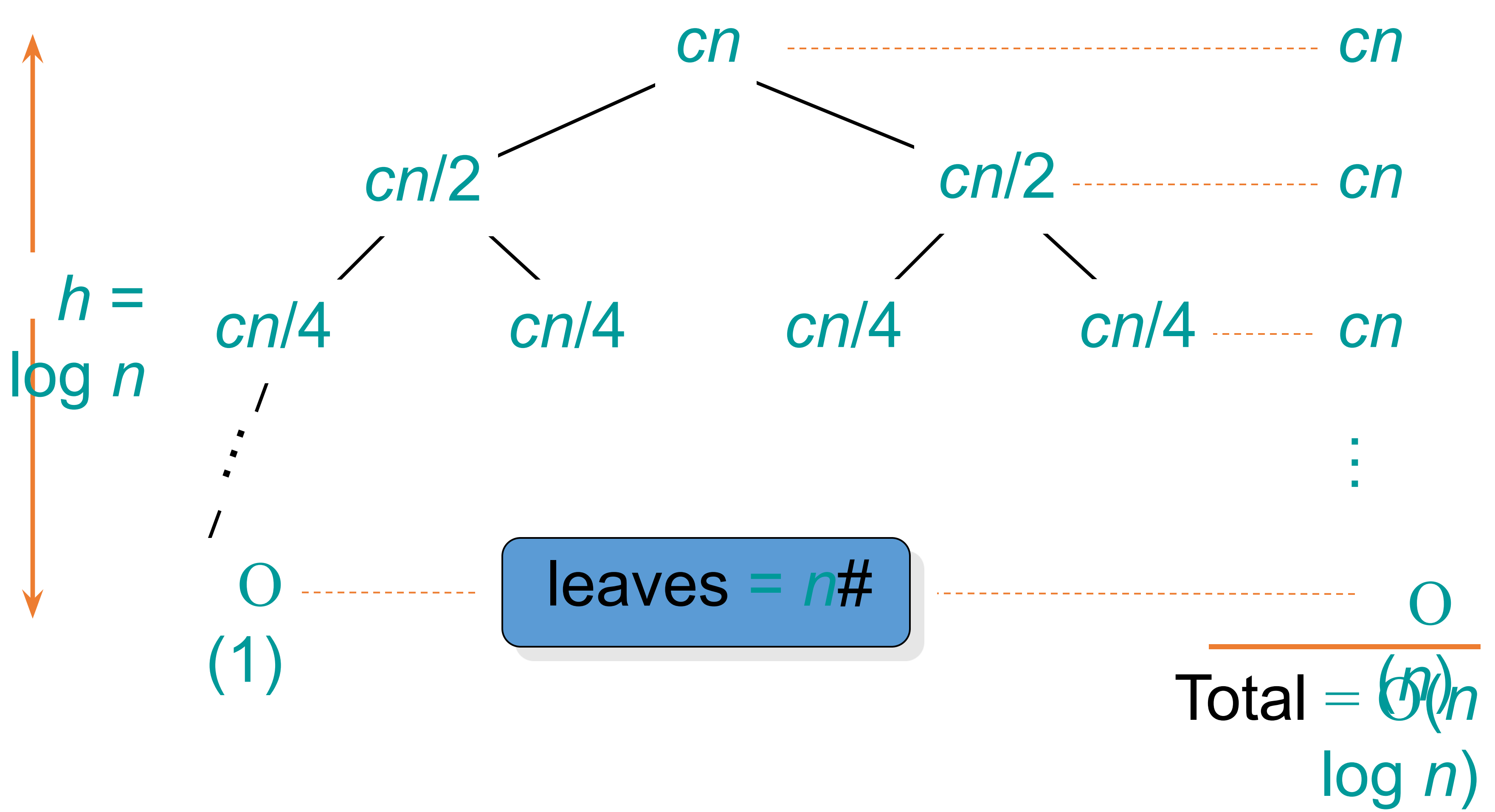
Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



Recursion tree

.Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant



at h level there will be 2^h nodes

$=2^h * T(1) + \sum_{i=0}^{h-1} cn$

$= n + n\sum_{i=0}^{h-1} n$

$=n+ n(h-1-0+1)$

$=n+ nh$

$=n+n\log_2 n$

$=O(n\log n)$

$$T(n) = T(n/3) + T(2n/3) + n.$$

Recursion Tree Method

(New Challenge)

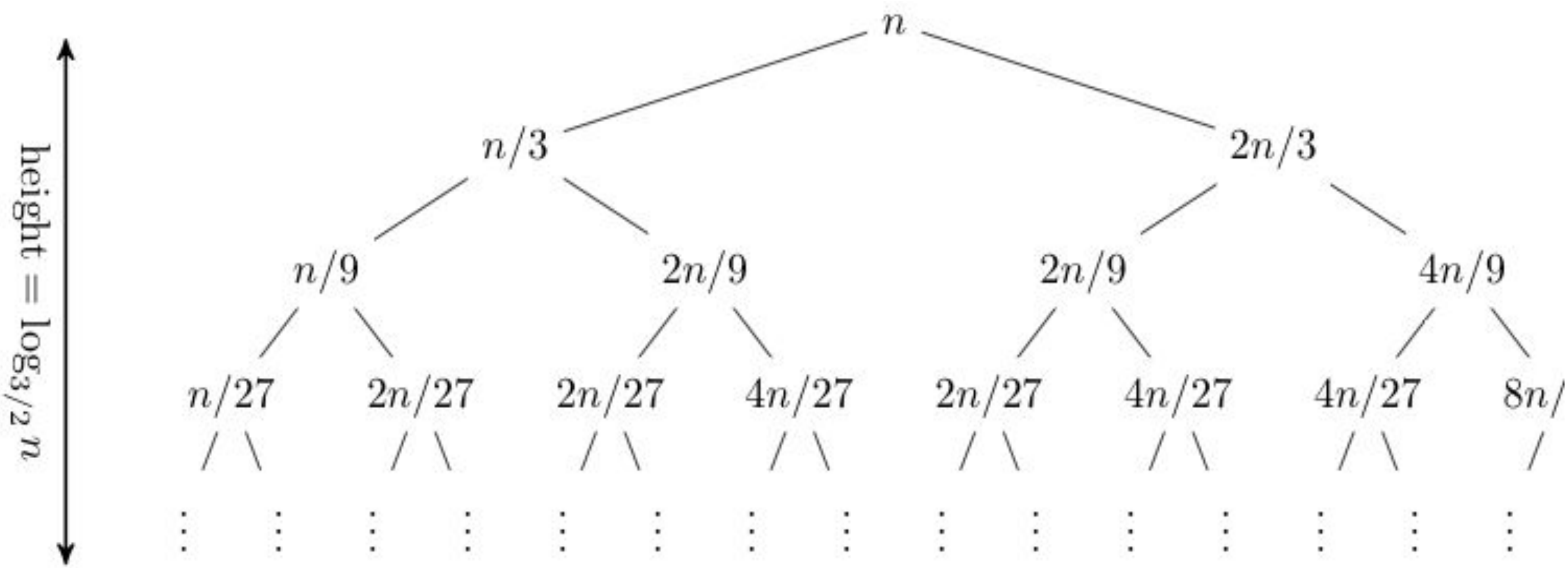
How to solve this?

$$T(n) = T(n/3) + T(2n/3) + n, \quad \text{with } T(1) = 1$$

What will be the recursion tree view?

T(n) = T(n/3) + T(2n/3) + n

- $T(n) = T(n/3) + T(2n/3) + n$.
- $T(1) = 1$
- Expanding out the first few levels, the recurrence tree is:



the closed form of this recurrence **is *O(n log n)*.**

Determining Height of tree

$Level\ 0 = \mathrm{n}(2/3)^0$

$Level\ 1 = \mathrm{n}(2/3)^1$

$Level\ 2 = \mathrm{n}(2/3)^2$

$Level\ \mathrm{i} = \mathrm{n}(2/3)^{\mathrm{i}}$

$Level\ 0 = \mathrm{n}(2/3)^{\mathrm{h}}$

$T(1) = \mathrm{T}(\mathrm{n}(2/3)^{\mathrm{h}}) \Rightarrow \mathrm{n}(2/3)^{\mathrm{h}} = 1 \Rightarrow \mathrm{n}=(3/2)^{\mathrm{h}} \Rightarrow \mathrm{h}=\log_{3/2} \mathrm{n}$

Home Task 1: Do it yourself

.Solve $T(n) = 3T(n/4) + cn^2$, where $c > 0$ is constant

$$=3^{h} * T(1) + n^2 \sum_{i=0}^{h-1} \left(\frac{3}{16}\right)^{i}$$

$$= c * 3^{\log_4 n} + n^2$$

$$= c * n^{\log_4 3} + n^2$$

$$= n + n^2$$

$$= O(n^2)$$

Home Task 2: Do it yourself

Solve $T(n) = 2T(n/2) + n^2$:

Thank you