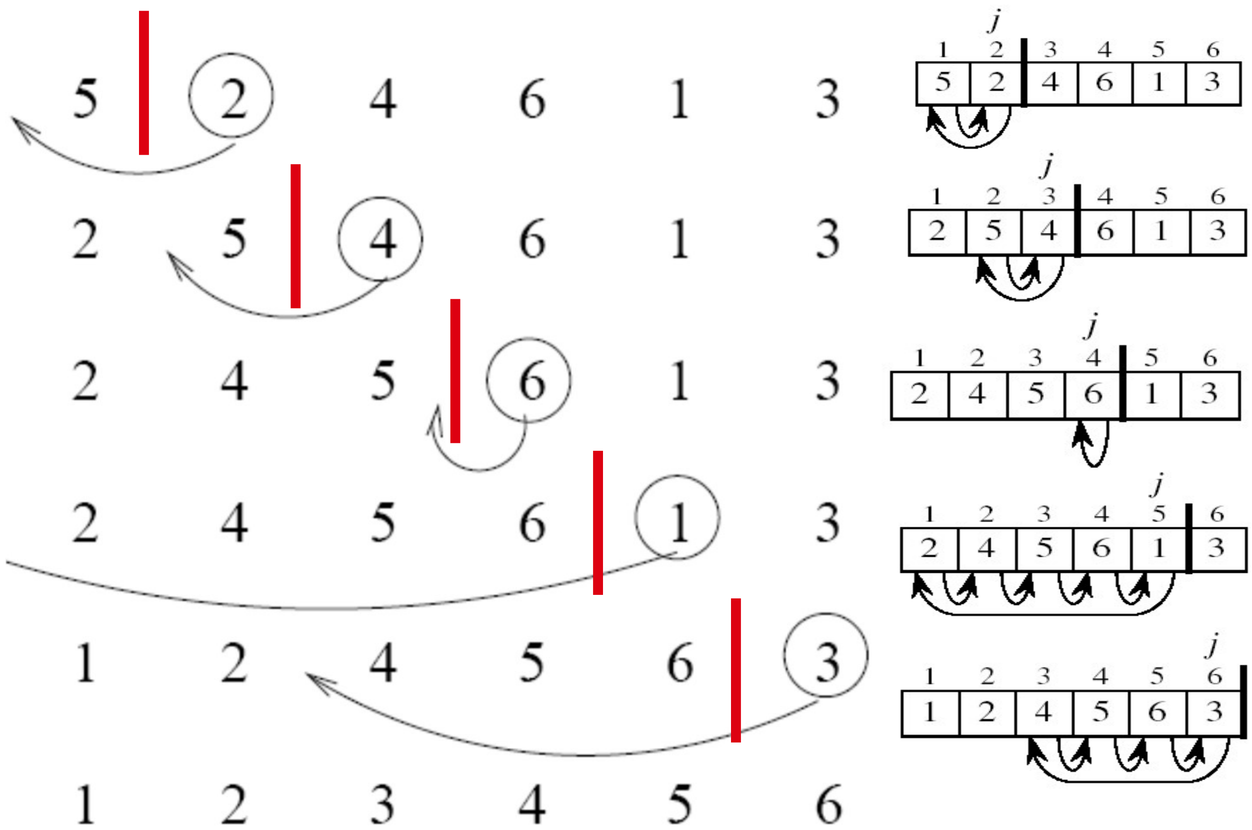
CS590/CPE590

Recurrence Relations

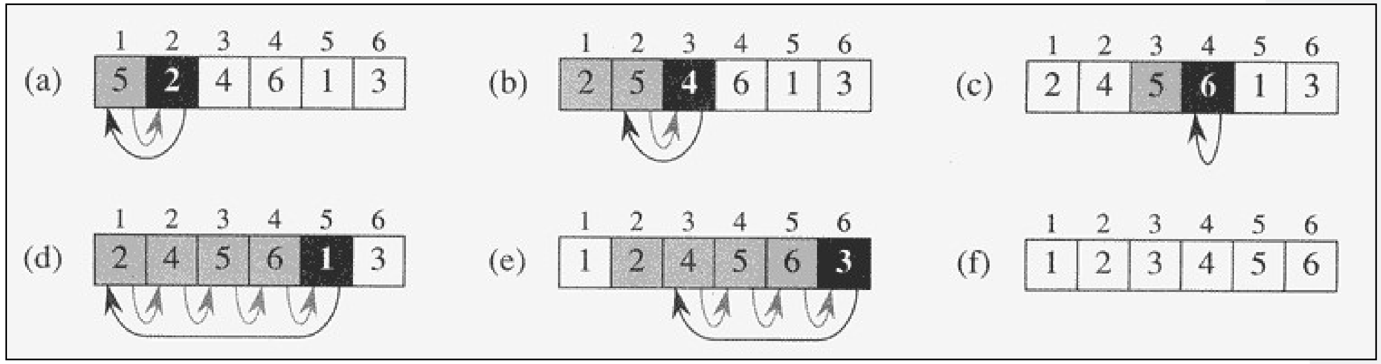
Kazi Lutful Kabir

Spring 2023

Insertion Sort again!



Insertion Sort



INSERTION-SORT(A)

```
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Insertion Sort (Recursive)

-- Base Case: Array size 1
-- Recursively sort first n-1 elements.
-- Insert last element at its correct position

```
void RecInSort(int A[], int n){
    int j, key;
    if (n==1)
    return;

    RecInSort(A, n-1);

    key = A[n-1];
    j = n-2;

    while(j>=0 and A[j] > key){
        A[j+1] = A[j];
        j = j-1;
    }
    A[j+1] = key;
}
```

Recurrences – Methods to Solve

There are several methods to solve recurrences by obtaining asymptotic bound on the solution.

Backward Substitution Method –Iteratively regenerate the terms in the recurrence to convert it into a form of a progression and then utilize the initial condition to find the full form of the function

Master Theorem – Provides bounds for recurrences. Mainly used for divide-and-conquer algorithms.

Recursion-tree Method – Convert the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. Solve the recurrence by bounding summations.

Recursion (General Form I)

Decrease-and-conquer Recurrence

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

- **a** is the number of times the recursive function is called in a single execution of the function
- **n-b** means this is a **decrease-and-conquer** algorithm
- **b** is the amount by which the input data is decreased in a recursive call
- **f(n)** is the amount of work performed in the function excluding the recursive calls

Recursion (General Form II)
Divide-and-conquer Recurrence

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

- a and b are constants, $a \geq 1$ and $b > 1$.
- a is the number of times the recursive call is made inside the function during a **single** execution of the function. Do not trace through to the base case. Simply count the number of times you see the function being called.
- b is the constant by which the input size is divided. The Master Theorem applies only if all recursive calls divide the input size by the same constant b.
- $f(n)$ is the amount of work that is performed in the function excluding the recursive calls.
- Assuming that n is a power of b simplifies the analysis.

Backward Substitution Method (5 steps)

Recurrence relation:

- x(n) = x(n-1) + (n-1), x(0) = 0

Backwards substitution

1) Replace n with n-1

x(n-1) = x(n-2) + (n-2)
x(n) = x(n-2) + (n-2) + (n-1)

2) Go back to the original and replace n with n-2

x(n-2) = x(n-3) + (n-3)
Go back one step and make the substitution
x(n) = x(n-3) + (n-3) + (n-2) + (n-1)

Solving Recurrence Relation (5 steps)

3) Write the general form of the equation

```
x(n) = x(n-i) + (n-i) + (n-i+1) + (n-i+2) + ... + (n-1)
```

4) Make use of the initial condition

```
n-i = 0
  i = n
```

5) Make the substitution and simplify

```
x(n) = x(n-n) + (n-n) + (n-n+1) + (n-n+2) + ... + (n-1)
     = 0 + 0 + 1 + 2 + 3 + ... + (n-1)
     = n(n-1)
       ------
          2
```

Example-1

$$\mathbf{x(n) = x(n-1) + n,\ \ x(0) = 0}$$

Step 1 - replace n by n-1

$$x(n-1) = \boxed{x(n-2) + n - 1}$$

$$\begin{aligned} x(n) &= x(n-1) + n \\ &= x(n-2) + (n-1) + n \end{aligned}$$

Step 2 - replace n by n-2

$$x(n-2) = \boxed{x(n-3) + n - 2}$$

$$\begin{aligned} x(n) &= x(n-2) + (n-1) + n \\ &= x(n-3) + (n-2) + (n-1) + n \end{aligned}$$

Step 3 - $x(n) = x(n-i) + (n-i+1) + (n-i+2) + ... + n$

Step 4 - initial condition: $x(0) = 0,$ so, $n - i = 0 \rightarrow \mathbf{i = n}$

Step 5 - $x(n) = 0 + 1 + 2 + 3 + ... + n = \dfrac{\mathbf{n(n+1)}}{\mathbf{2}}$

Example-2

$$\mathbf{x(n) = 2x(n/2) + n, \ \ x(1) = 1}$$

Step 1 - replace n by n/2

$$\begin{aligned} x(n/2) &= \boxed{2x(n/4) + (n/2)} \\ x(n) &= 2x(n/2) + n \\ &= 2[2x(n/4) + (n/2)] + n \\ &= 4x(n/4) + 2(n/2) + n \\ &= 4x(n/4) + 2n \end{aligned}$$

Step 2 - replace n by n/4

$$\begin{aligned} x(n/4) &= \boxed{2x(n/8) + (n/4)} \\ x(n) &= 4x(n/4) + 2n \\ &= 4[2x(n/8) + (n/4)] + 2n \\ &= 8x(n/8) + 4(n/4) + 2n \\ &= 8x(n/8) + 3n \end{aligned}$$

Step 3 - $x(n) = 2^k x(\frac{n}{2^k}) + kn$

Step 4 - want $x(1)$, so let $n = 2^k \rightarrow \mathbf{k = lg(n)}$

Step 5 - $x(n) = 2^{lg(n)} \cdot x(\frac{2^k}{2^k}) + lg(n) \cdot n$

$$\begin{aligned} &= n \cdot x(1) + n \cdot lg(n) \\ &= \mathbf{nlg(n) + n} \end{aligned}$$

Example-3

$$\mathbf{x(n)= x(\frac{n}{2}) + 1,\ x(1) = 1}$$ <u>Binary Search</u>

$$\textbf{Step 1 -}\ x(\frac{n}{2}) = x(\frac{n}{4}) + 1$$
$$x(n) = x(\frac{n}{4}) + 1 + 1$$
$$= x(\frac{n}{4}) + 2$$

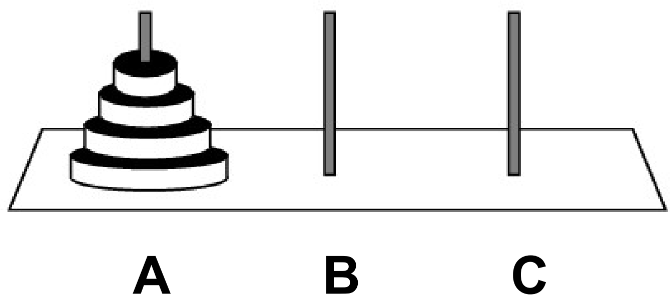
$$\textbf{Step 2 -}\ x(\frac{n}{4}) = x(\frac{n}{8}) + 1$$
$$x(n) = x(\frac{n}{8}) + 1 + 2$$
$$= x(\frac{n}{8}) + 3$$

$$\textbf{Step 3 -}\ x(n) = x(\frac{n}{2^k}) + k$$

$$\textbf{Step 4 -}\ x(1) = 1, \text{ let } n = 2^k \to \mathbf{k = lg(n)}$$

$$\textbf{Step 5 -}\ x(n) = x(\frac{2^k}{2^k}) + lg(n)$$
$$= x(1) + lg(n)$$
$$= \mathbf{lg(n) + 1}$$

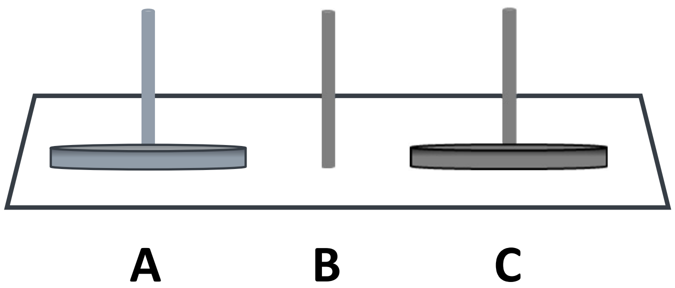
The Towers of Hanoi



Goal: Move the tower from peg A to peg C.
Move one disk at a time.
Never place a larger disk atop a smaller one.

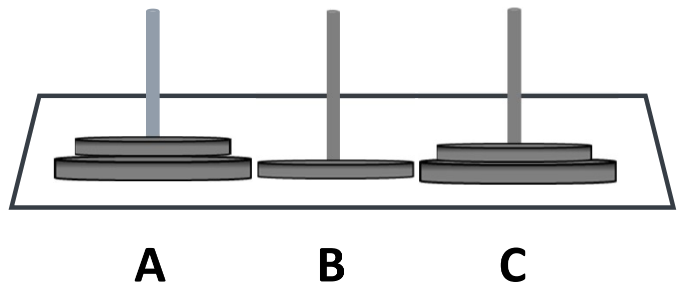
Questions: Can it always be done?
What is $M(n)$, the number of moves to move n disks?

The Towers of Hanoi



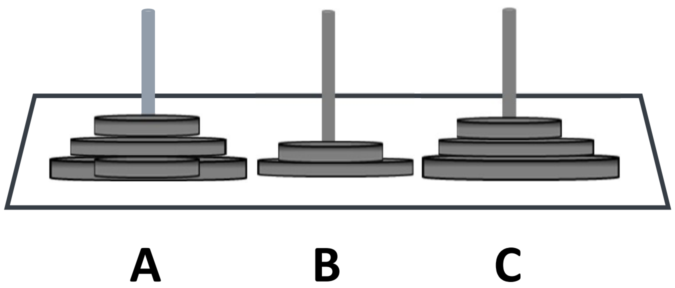
$$M(1) = 1$$

The Towers of Hanoi



$$M(2) = 3$$

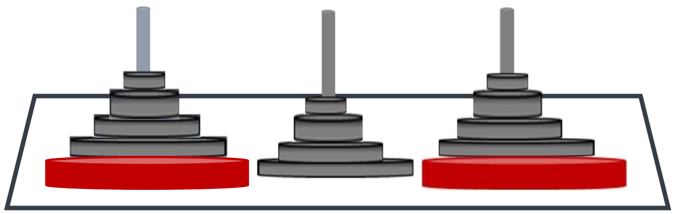
The Towers of Hanoi



$$M(3) = 7$$

1
2
3
4
5
6
7

The Towers of Hanoi



$M(n-1)$

1

$M(n-1)$

$$M(n) = 2M(n-1) + 1$$
$$M(1) = 1$$

This is a recurrence relation describing $M(n)$

The Towers of Hanoi

$$M(n) = 2M(n-1) + 1$$
$$M(1) = 1$$

Let's calculate a few values:

$$M(1) = 1$$
$$M(2) = 2 \cdot 1 + 1 = 3$$

$$M(3) = 2 \cdot 3 + 1 = 7$$
$$M(4) = 2 \cdot 7 + 1 = 15$$
$$M(5) = 2 \cdot 15 + 1 = 31$$

This suggests $M(n) = 2^n - 1$

Example-4

$$\mathbf{x(n)} = \mathbf{x(n-1) + 1 + x(n-1)},\ \mathbf{x(1) = 1}$$
$$= 2x(n-1) + 1$$

Step 1 - $x(n-1) = 2x(n-2) + 1$
$$x(n) = 2[2x(n-2) + 1] + 1$$
$$= 4x(n-2) + 3$$

Step 2 - $x(n-2) = 2x(n-3) + 1$
$$x(n) = 4[2x(n-3) + 1] + 3$$
$$= 8x(n-3) + 7$$

Step 3 - $x(n) = 2^k x(n-k) + 2^k - 1$

Step 4 - $x(1) = 1$, so $n = k + 1 \rightarrow \mathbf{k = n - 1}$

Step 5 - $x(n) = 2^{n-1} x(n-(n-1)) + 2^{n-1} - 1$
$$= 2^{n-1} \cdot x(1) + 2^{n-1} - 1$$
$$= 2^{n-1} + 2^{n-1} - 1$$
$$= \mathbf{2^n - 1}$$

Example-5

$$\mathbf{x(n)} = \mathbf{x(\frac{n}{2}) + 1,\ x(1) = 1}\ \underline{\text{Binary Search}}$$

Solve for $n = 2^k$

Step 0 - Rewrite the recurrence, making the substitution first

$$x(2^k) = x(2^{k-1}) + 1$$

Step 1 - replace 2^k with 2^{k-1}

$$x(2^{k-1}) = x(2^{k-2}) + 1$$

$$x(2^k) = x(2^{k-2}) + 1 + 1$$

$$= x(2^{k-2}) + 2$$

Step 2 - replace 2^k with 2^{k-2}

$$x(2^{k-2}) = x(2^{k-3}) + 1$$

$$x(2^k) = x(2^{k-3}) + 1 + 2$$

$$= x(2^{k-2}) + 3$$

Step 3 - $x(2^k) = x(2^{k-i}) + i$

Step 4 - $2^{k-i} = 1$

$$2^{k-i} = 2^0$$

$$k - i = 0$$

$$i = k$$

Step 5 - $x(2^k) = x(2^{k-i}) + i$

$$= x(2^{k-k}) + k$$

$$= x(2^0) + k$$

$$= x(1) + k$$

$$= 1 + k$$

$$n = 2^k$$

$$k = lg(n)$$

$$x(n) = 1 + lg(n)$$

$$= \mathbf{lg(n) + 1}$$

Recursion (General Form II)
Divide-and-conquer Recurrence

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

- a and b are constants, $a \geq 1$ and $b > 1$.
- a is the number of times the recursive call is made inside the function during a **single** execution of the function. Do not trace through to the base case. Simply count the number of times you see the function being called.
- b is the constant by which the input size is divided. The Master Theorem applies only if all recursive calls divide the input size by the same constant b.
- $f(n)$ is the amount of work that is performed in the function excluding the recursive calls.
- Assuming that n is a power of b simplifies the analysis.

Master Theorem

- If $f(n) \in \theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_b n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Analogous results hold for the O and Ω notations.
- Make sure you simplify your expressions.

Function 0

Which case of the Master Theorem, if any, applies?

```
int function0(int n) {
    int temp = 1;
    if (n <= 1) {
        return temp;
    }
    temp += function0(n - 1);
    return temp;
}
```

Function 0

Which case of the Master Theorem, if any, applies?

```
int function0(int n) {
    int temp = 1;
    if (n <= 1) {
        return temp;
    }
    temp += function0(n - 1);
    return temp;
}
```

None!
This is an example of decrease-and-conquer. A constant is being subtracted from the input in the recursive call.

Function 1

Which case of the Master Theorem, if any, applies?

```
int function1(int n) {
    int temp = 1;
    if (n <= 1) {
        return temp;
    }
    temp += function1(n / 2);
    temp += function1(n / 2);
    return temp;
}
```

Function 1

Which case of the Master Theorem, if any, applies?

```
int function1(int n) {
    int temp = 1;
    if (n <= 1) {
        return temp;
    }
    temp += function1(n / 2);
    temp += function1(n / 2);
    return temp;
}
```

$$a = 2$$
$$b = 2$$
$$f(n) = \theta(1) = n^0 \Rightarrow d = 0$$

$$a \_ b^d$$
$$2 > 2^0$$
$$T(n) \in \theta\left(n^{\log_2 2}\right) = \theta(n)$$

Function 2

Which case of the Master Theorem, if any, applies?

```
int function2(int n) {
    int temp = 0;
    if (n > 1) {
        for (int i = 1; i <= n; ++i) {
            ++temp;
        }
        temp += function2(n / 2);
    }
    return temp;
}
```

Function 2

Which case of the Master Theorem, if any, applies?

```
int function2(int n) {
    int temp = 0;
    if (n > 1) {
        for (int i = 1; i <= n; ++i) {
            ++temp;
        }
        temp += function2(n / 2);
    }
    return temp;
}
```

$$a = 1$$
$$b = 2$$
$$f(n) = \theta(n) = n^1 \Rightarrow d = 1$$

$$a \_ b^d$$
$$1 < 2^1$$
$$T(n) \in \theta(n^1) = \theta(n)$$

Function 3

Which case of the Master Theorem, if any, applies?

```
int function3(int n) {
    if (n <= 1) {
        return 0;
    }
    int temp = 0;
    for (int i = 1; i <= 8; ++i) {
        temp += function3(n / 2);
    }
    for (int i = 1, max = n * n * n; i <= max; ++i) {
        ++temp;
    }
    return temp;
}
```

Function 3

Which case of the Master Theorem, if any, applies?

```
int function3(int n) {
    if (n <= 1) {
        return 0;
    }
    int temp = 0;
    for (int i = 1; i <= 8; ++i) {
        temp += function3(n / 2);
    }
    for (int i = 1, max = n * n * n; i <= max; ++i) {
        ++temp;
    }
    return temp;
}
```

$$a = 8$$
$$b = 2$$
$$f(n) = \theta(n^3) \Rightarrow d = 3$$

$$a \_ b^d$$
$$8 = 2^3$$
$$T(n) \in \theta(n^3 \log_2 n) = \theta(n^3 \lg n)$$

Function 4

Which case of the Master Theorem, if any, applies?

```
int function4(int n) {
    int temp = 1;
    for (int i = 1; i <= n; ++i) {
        ++temp;
    }
    temp += function4(n / 2);
    temp += function4(n / 2);
    return temp;
}
```

Function 4

Which case of the Master Theorem, if any, applies?

```
int function4(int n) {
    int temp = 1;
    for (int i = 1; i <= n; ++i) {
        ++temp;
    }
    temp += function4(n / 2);
    temp += function4(n / 2);
    return temp;
}
```

$a = 2$

$b = 2$

$f(n) = \theta(n) \Rightarrow d = 1$

$a \,\_\, b^d$

$2 = 2^1$

$T(n) \in \theta(n^1 \log_2 n) = \theta(n \lg n)$

More Examples

Ex. $T(n) = 4T(n/2) + n$

$a = 4,\ b = 2,\ \text{d}=1 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n.$

Since, $\boldsymbol{a > b^d}$

$\therefore\ T(n) = \Theta(n^2).$

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4,\ b = 2,\ \text{d}=2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2.$

Since, $\boldsymbol{a = b^d}$

$\therefore\ T(n) = \Theta(n^2 \lg n).$

More Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2,\ \mathrm{d}{=}3 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

Since, $\boldsymbol{a < b^d}$

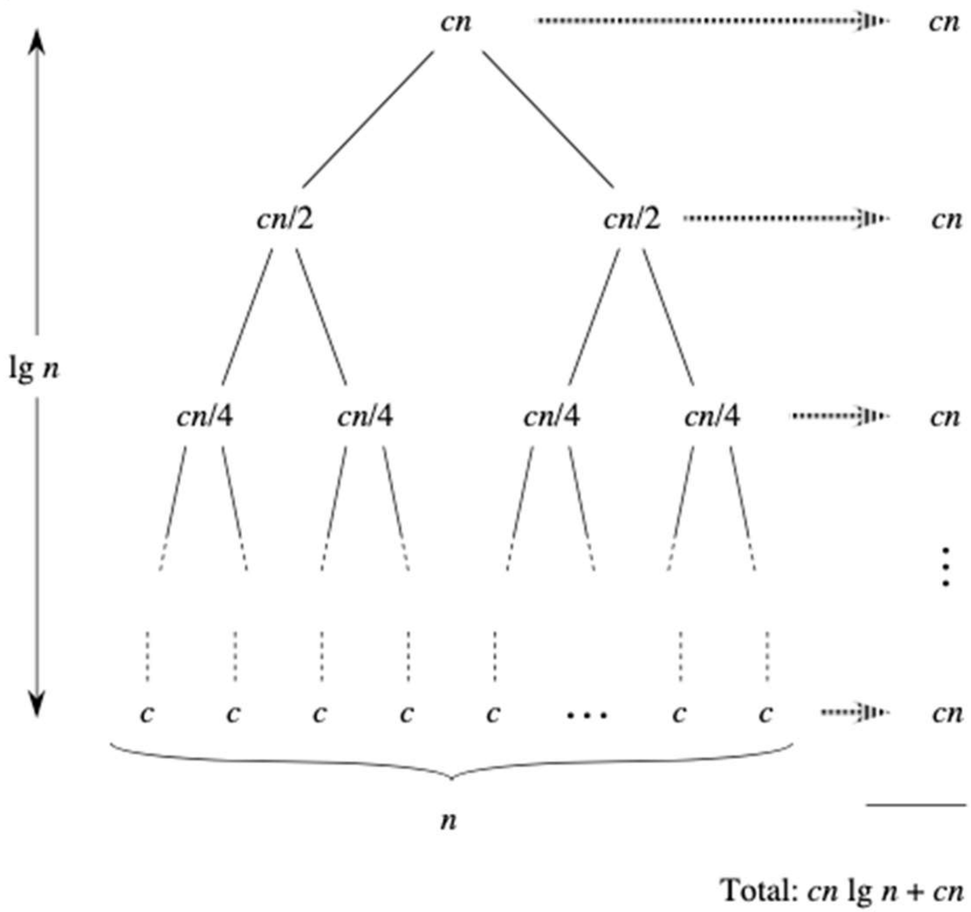
$\therefore\ T(n) = \Theta(n^3).$

Recursion Tree Method: Analyzing Merge Sort

- Running time for merge-sort is $T(n) = \Theta(n \lg n)$ where $\lg n = \log_2 n$
- Rewrite recurrence as

$$T(n) = \begin{cases} c & if\ (n = 1), \\ 2T\left(\frac{n}{2}\right) + cn & if\ (n > 1). \end{cases}$$

Analyzing Merge Sort

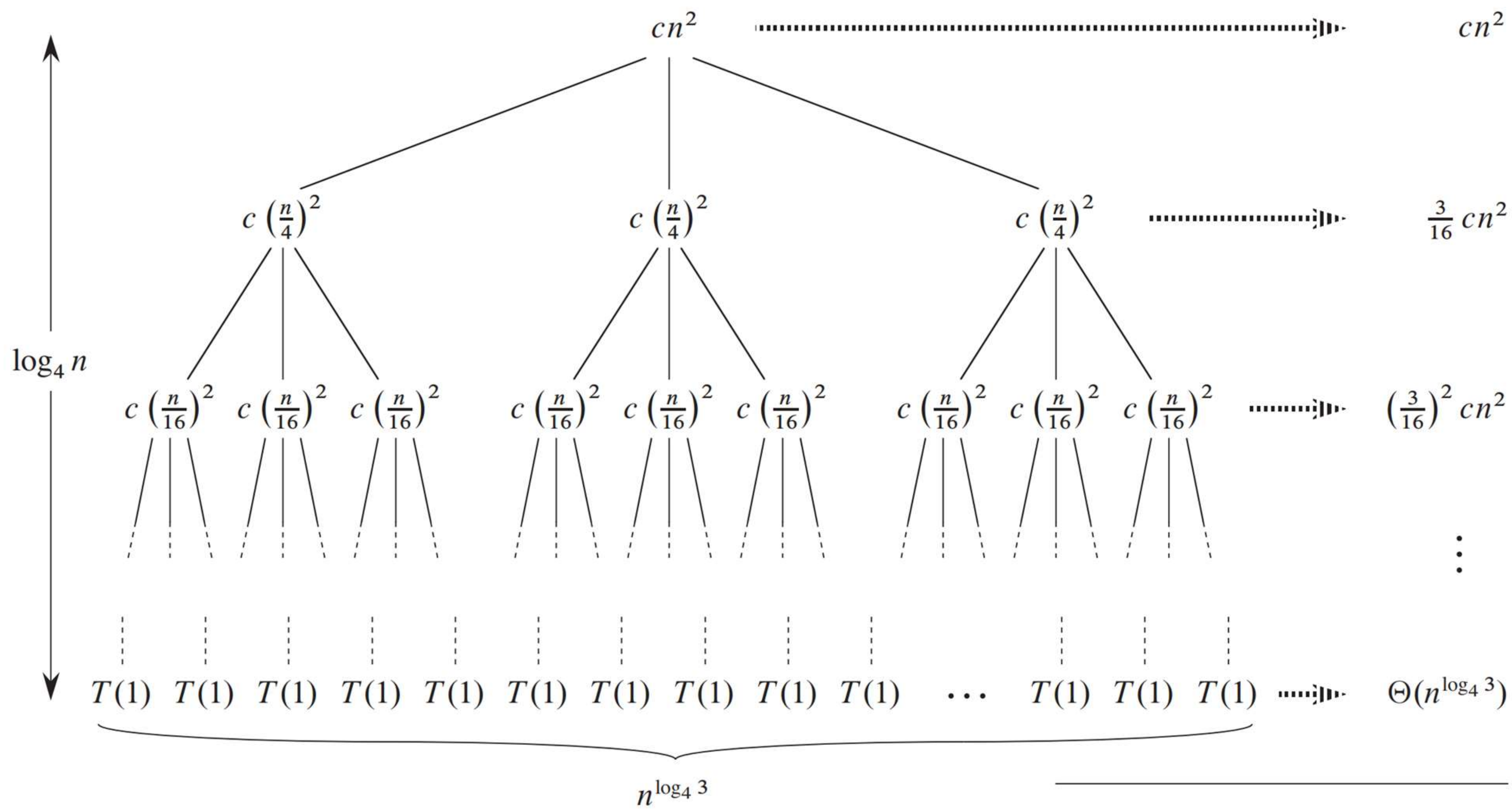


Each level has cost cn.

- Each time we go down one level, the number of subproblems doubles but the cost per subproblem halves $\Rightarrow$ cost/level stays the same.

There are $\lg n + 1$ levels (height is $\lg n$).
Total cost is sum of costs at each level.
Total cost is $cn \lg n + cn \Rightarrow \Theta(n \lg n)$.

$$\boxed{T(n) = 3T(n/4) + cn^2}$$



Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition. How far from the root do we reach one? The subproblem size for a node at depth i is $n/4^i$. Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$. Thus, the tree has $\log_4 n + 1$ levels (at depths $0, 1, 2, \ldots, \log_4 n$).

Next we determine the cost at each level of the tree. Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i. Because subproblem sizes reduce by a factor of 4 for each level we go down from the root, each node at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, has a cost of $c(n/4^i)^2$. Multiplying, we see that the total cost over all nodes at depth i, for $i = 0, 1, 2, \ldots, \log_4 n - 1$, is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta(n^{\log_4 3})$, since we assume that $T(1)$ is a constant.

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n-1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2) .
\end{aligned}
$$

Contents of this presentation are partially adapted from
My CS385 (Fall2022)
and from
Prof. In Suk Jang CS590 (Summer 2021 Lecture-4)
and are also based on
Book Chapter- 4, Introduction to Algorithms by *Cormen, Leiserson, Rivest,* & *Stein*

THANK **YOU**

Stevens Institute of Technology