Design and Analysis of Data Structures and Algorithms

Warin Wattanapornprom PhD.

กฎการฟังบรรยาย

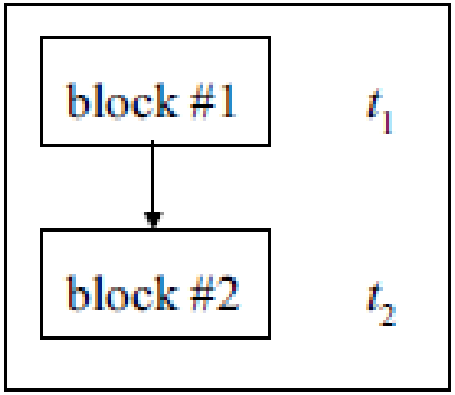
Algorithm and Complexity

PART 4:
ALGORITHM ANALYSIS:
ALGORITHM CONTROL STRUCTURE

Algorithm Control Structures

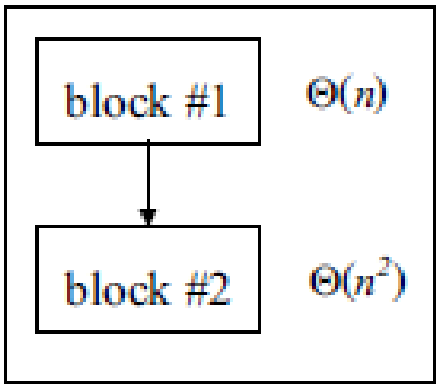
- Sequencing
- If-Then-Else
- “For” loop
- “While” loop
- Recursive calls

Sequencing



$$t_1 + t_2 = \max(t_1, t_2)$$

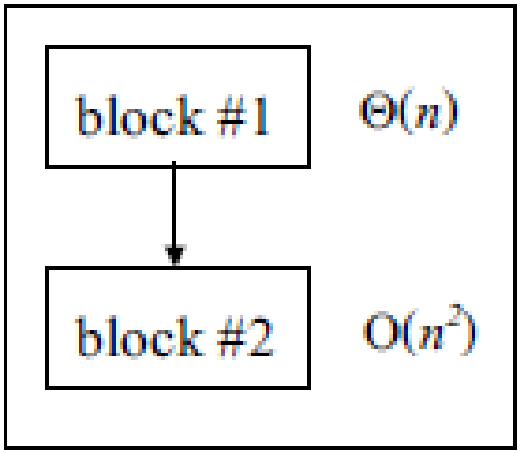
Sequencing



$$t_1 + t_2 = \max(t_1, t_2)$$

$$\Theta(n) + \Theta(n^2) = \max(\Theta(n), \Theta(n^2)) = \Theta(n^2)$$

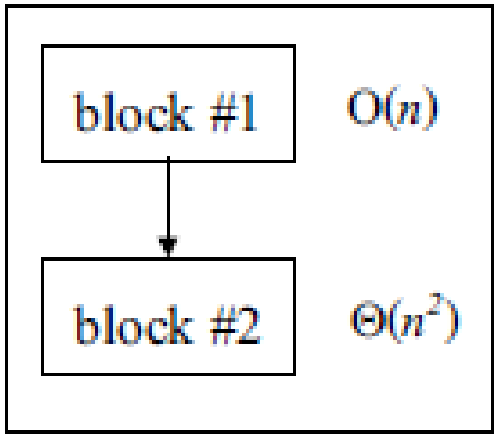
Sequencing



$$t_1 + t_2 = \max(t_1, t_2)$$

$$\Theta(n) + O(n^2) = \max(\Theta(n), O(n^2)) = O(n^2)$$

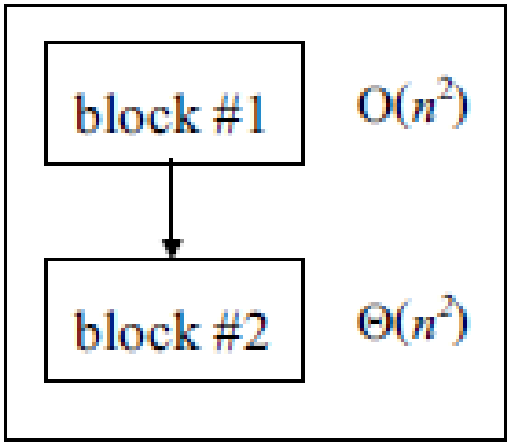
Sequencing



$$t_1 + t_2 = \max(t_1, t_2)$$

$$O(n) + \Theta(n^2) = \max(O(n), \Theta(n^2)) = \Theta(n^2)$$

Sequencing



$$t_1 + t_2 = \max(t_1, t_2)$$

$$O(n^2) + \Theta(n^2) = \max\big(O(n^2), \Theta(n^2)\big) = \Theta(n^2)$$

“For” Loop

```
for ( i = 1 to m )
{
        P(i)
}
```

$$\sum_{i=1}^{m} t_i$$

“For” Loop

```
for ( i = 1 to m )
    s += A[i][j]
```

$$\sum_{i=1}^{m} \Theta(1) = \Theta\left(\sum_{i=1}^{m} 1\right)$$

$$= \Theta(m)$$

“For” Loop

```
for ( i = 1 to m )
   for ( j = 1 to i )
      s += A[i][j]
```

$$\sum_{i=1}^{m}\sum_{j=1}^{m}\Theta(1) = \sum_{i=1}^{m}\Theta(m)$$

$$= \Theta\left(\sum_{i=1}^{m} m\right)$$

$$= \Theta(m^2)$$

“For” Loop

```
for ( i = 1 to m )
   for ( j = 1 to i )
      s += A[i][j]
```

$$\sum_{i=1}^{m-1}\sum_{j=1}^{i}\Theta(1) = \sum_{i=1}^{m-1}\Theta(i)$$

$$= \Theta\left(\sum_{i=1}^{m-1} i\right)$$

$$= \Theta\left(\frac{m(m+1)}{2}\right)$$

$$= \Theta(m^2)$$

“For” Loop

```
for ( i = 2 to m-1 )
   for ( j = 3 to i )
      s += A[i][j]
```

$$\sum_{i=2}^{m-1}\sum_{j=3}^{i}\Theta(1) = \sum_{i=2}^{m-1}\Theta(i)$$

$$= \Theta\left(\sum_{i=2}^{m-1} i\right)$$

$$= \Theta\left(\frac{m^2}{2} + \Theta(m)\right)$$

$$= \Theta(m^2)$$

“While” Loop

```
while (n>0) {
    ..
    n = n-1
}
```
$\Theta(n)$

```
while (n>0) {
    ..
    n = n/2
}
```
$\Theta(\log n)$

```
i=0; j=n;
while (i>j) {
    ..
    i++; j--;
}
```
$\Theta(n)$

“While” Loop: Insertion Sort

```
Insertion_Sort( A[1..n] )
   for j = 2 to n
   {
      key = A[j]
      i = j-1
      while i>0 and A[i]>key
      {
         A[i+1] = A[i]
         i = i-1
      }
      A[i+1] = key
   }
```

$$O(i) \rightarrow O(j)$$

$$\sum_{j=2}^{n} O(j) = O(n^2)$$

While Loop: Euclid's GCD

```
GCD( k, n )
{
    while k > 0
    {
        t = k
        k = n mod k
        n = t
    }
    return n
}
```

O(log n)

t	k	n
	88	128
88	40	88
40	8	40
8	0	8

ให้ $n \geq k$ จะพิสูจน์ว่า

$n \bmod k < n/2$ เสมอ

1: $k \leq n/2$

$n \bmod k < k \leq n/2$

2: $k > n/2$

$n/k < 2$, int(n/k) = 1

$n \bmod k = n - k * \text{int}(n/k)$

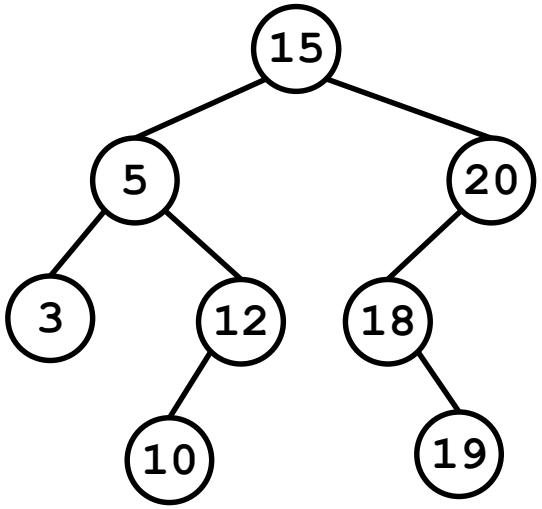
$= n - k$

$< n - n/2 = n/2$

“While” Loop: FindMin_BST

```
Position FindMin_BST( TREE T )
{
  Position p;
  p = T;
  if ( p != NULL ) {
    while ( p->left != NULL ) {
      p = p->left;
    }
  }
  return p;
}
```

O(n)



Recursive Calls

```
1:waste( n )
2:{
3:  if ( n = 0 ) return 0
4:  for ( i = 1 to n )
5:    for (j = 1 to i )
6:      print i,j,n
7:  for( i = 1 to 3 )
8:    waste( n/2 )
9:}
```

3: $\Theta(1)$
4 and 5: $\Theta(n^2)$
7 and 8: $3T(n/2)$

$$T(n) = 3T(n/2) + \Theta(n^2)$$

Meet the Recurrence

A **recurrence** relation is an equation that defines a sequence based on a rule that gives the next term as a function of the previous term(s)

It's a lot like recursive code:

- At least one base case and at least one recursive case
- Each case should include the values for n to which it corresponds
- The recursive case should reduce the input size in a way that eventually triggers the base case
- The cases of your recurrence usually correspond exactly to the cases of the code

A generic example of a recurrence:

$$T(n) = \begin{cases} 5 & \text{if } n < 3 \\ 2T\left(\frac{n}{2}\right) + 10 & \text{otherwise} \end{cases}$$

Writing Recurrences: Example 1

```
public int recurse(int n) {
  if (n < 3) {
    return 80;
  }

  int a = n * 2;

  int val1 = recurse(n / 3);
  int val2 = recurse(n / 3);

  return val1 + val2;
}
```

+2 Base Case

Recursive Case

Non-recursive Work: +4

Recursive Work: + 2*T(n/3)

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + 4 & \text{otherwise} \end{cases}$$

Writing Recurrences: Example 2

```
public int recurse(int n) {
  if (n < 3) {
    return 80;
  }

  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }

  int val1 = recurse(n / 3);
  int val2 = recurse(n / 3);

  return val1 + val2;
}
```

+2 Base Case

+n

+2

Recursive Case

Non-recursive Work: + n + 2

Recursive Work: + 2*T(n/3)

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 2T\left(\frac{n}{3}\right) + n + 2 & \text{otherwise} \end{cases}$$

Writing Recurrences: Example 3

```
public int recurse(int n) {
  if (n < 3) {
    return 80;
  }

  for (int i = 0; i < n; i++) {
    System.out.println(i);
  }

  int val1 = recurse(n / 4);
  int val2 = recurse(n / 4);
  int val3 = recurse(n / 4);


  return val1 + val2 * val3;
}
```

+2 Base Case

+n

+3

Recursive Case

Non-recursive Work: + n + 3

Recursive Work: + 3*T(n/4)

$$T(n) = \begin{cases} 2 & \text{if } n < 3 \\ 3T\left(\frac{n}{4}\right) + n + 3 & \text{otherwise} \end{cases}$$

Substitution method

("Guessing"+ Induction)

Guess: $T(n) = \mathrm{O}(n), T(n) \leq cnT(n)$
$= T(0.7n) + T(0.2n) + \mathrm{O}(n)$

Proof:

Basis: obvious

Induction: $T(n) \leq 0.7cn + 0.2cn + \mathrm{O}(n)$

$= 0.9cn + \mathrm{O}(n)$

$\leq 0.9cn + dn$

$= cn$ (choose $d = 0.1c$)

Theorem

if $\sum_{i-1}^{k} \alpha_i < 1$ then the solution to the recurrence

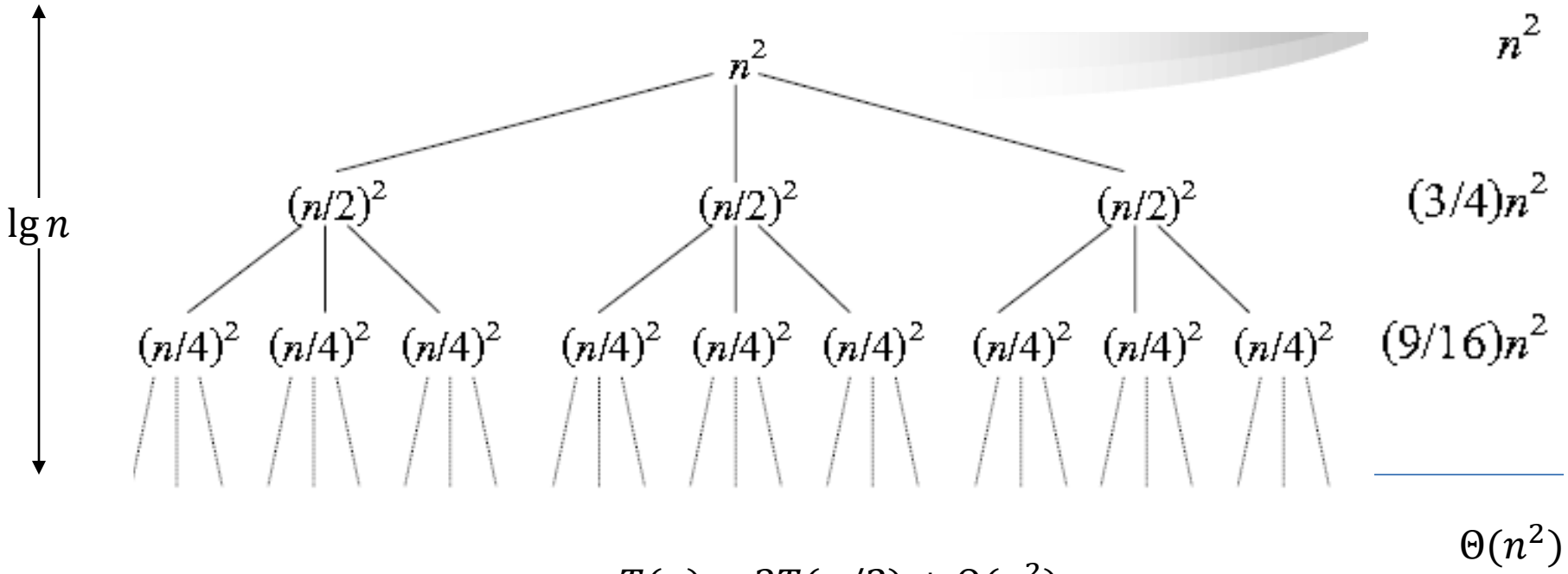
$$T(n) = \sum_{i=1}^{k} T(\alpha_i n) + O(n)$$

is $T(n) = O(n)$

$$T(n) = T(0.3n) + T(0.2n) + T(0.09n) + T(0.4n) + \mathrm{O}(n)$$

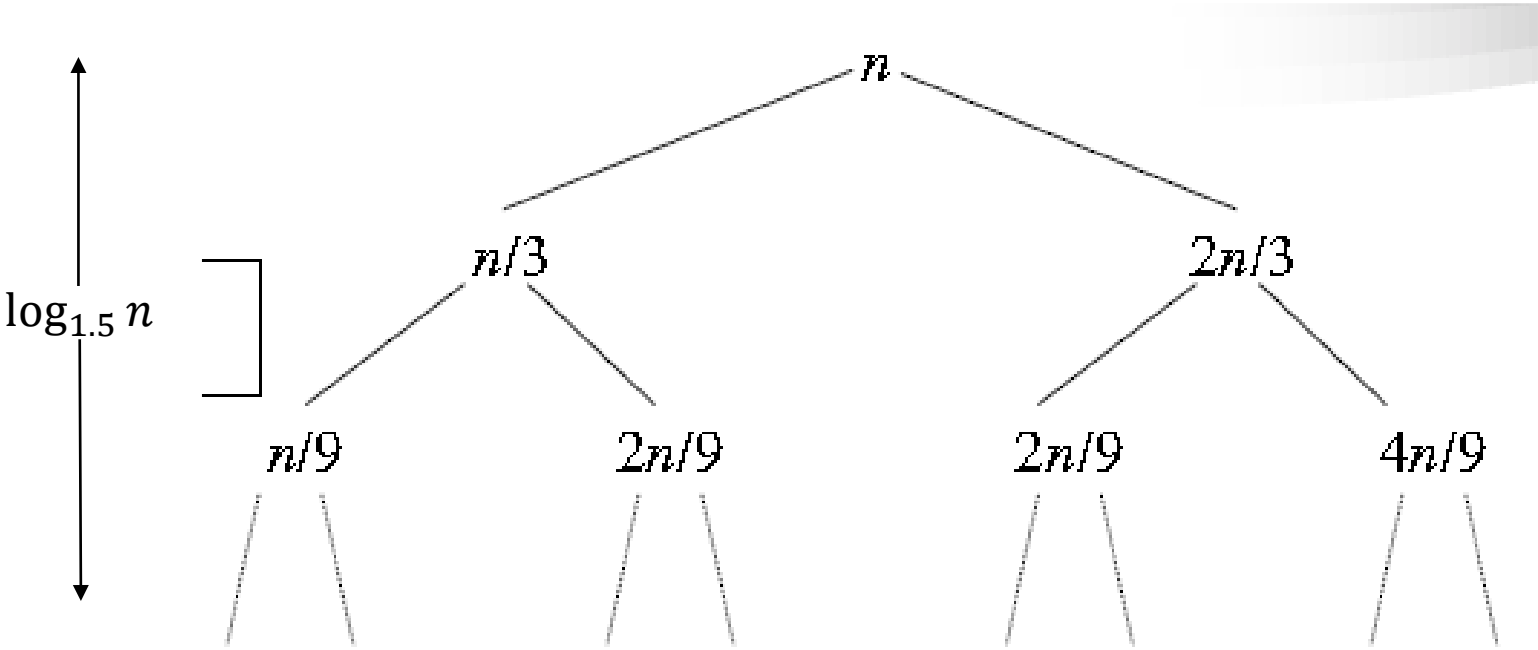
$$= \mathrm{O}(n)$$

Recursion Trees



$$T(n) - 3T(n/2) + \Theta(n^2)$$
$$T(n/2) - 3T(n/4) + \Theta((n/2)^2)$$

Recursion Trees



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

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

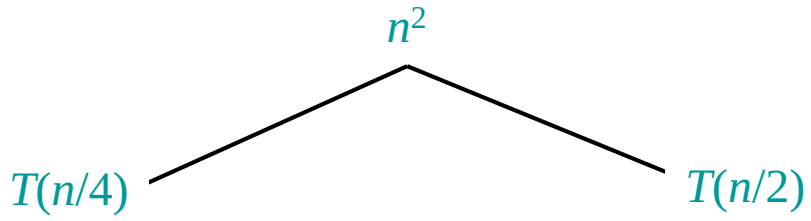
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$T(n)$

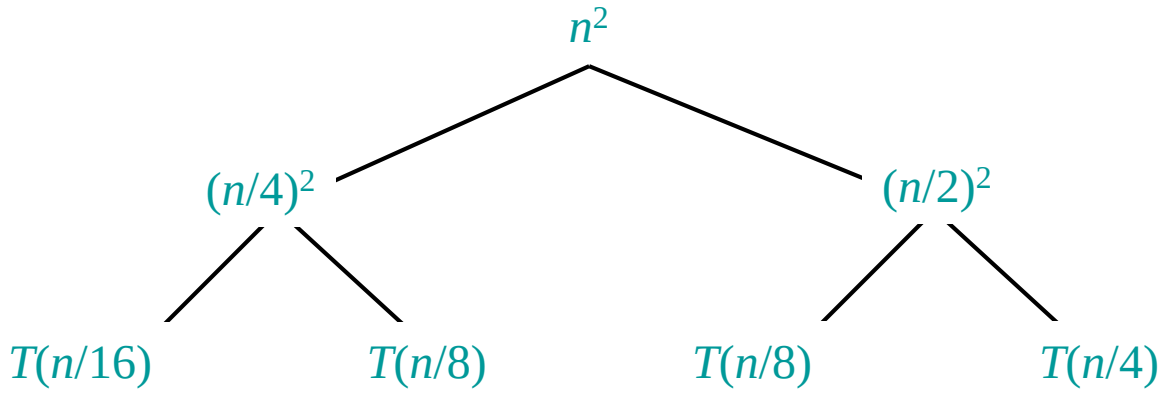
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



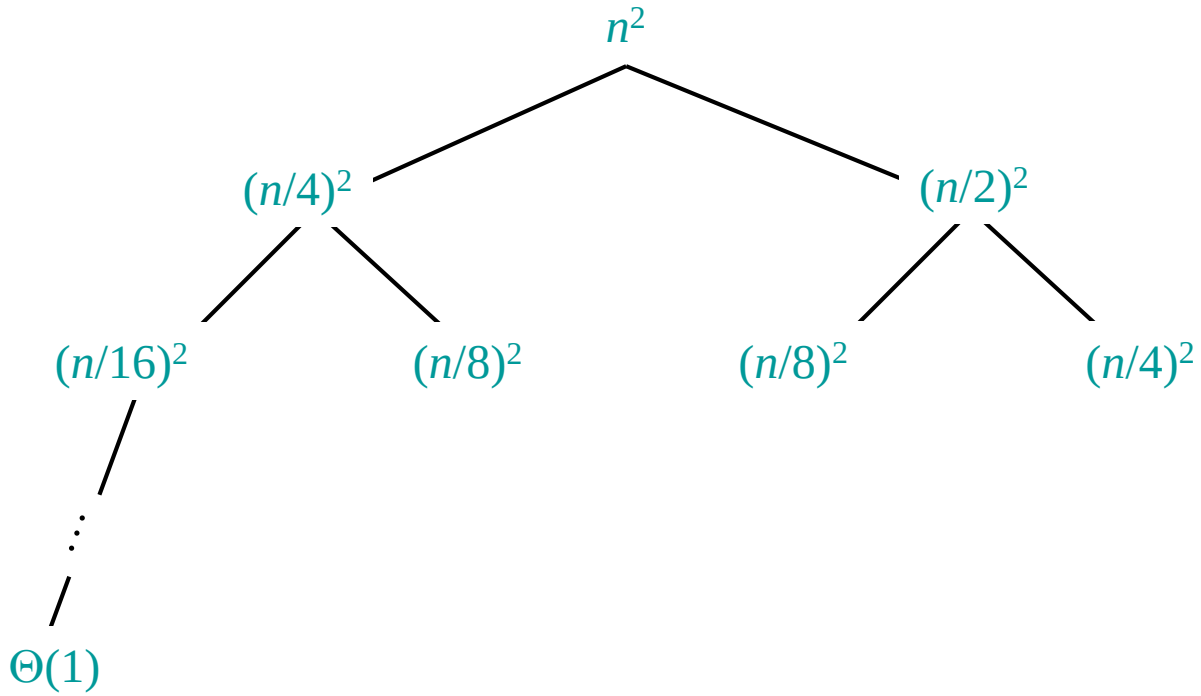
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



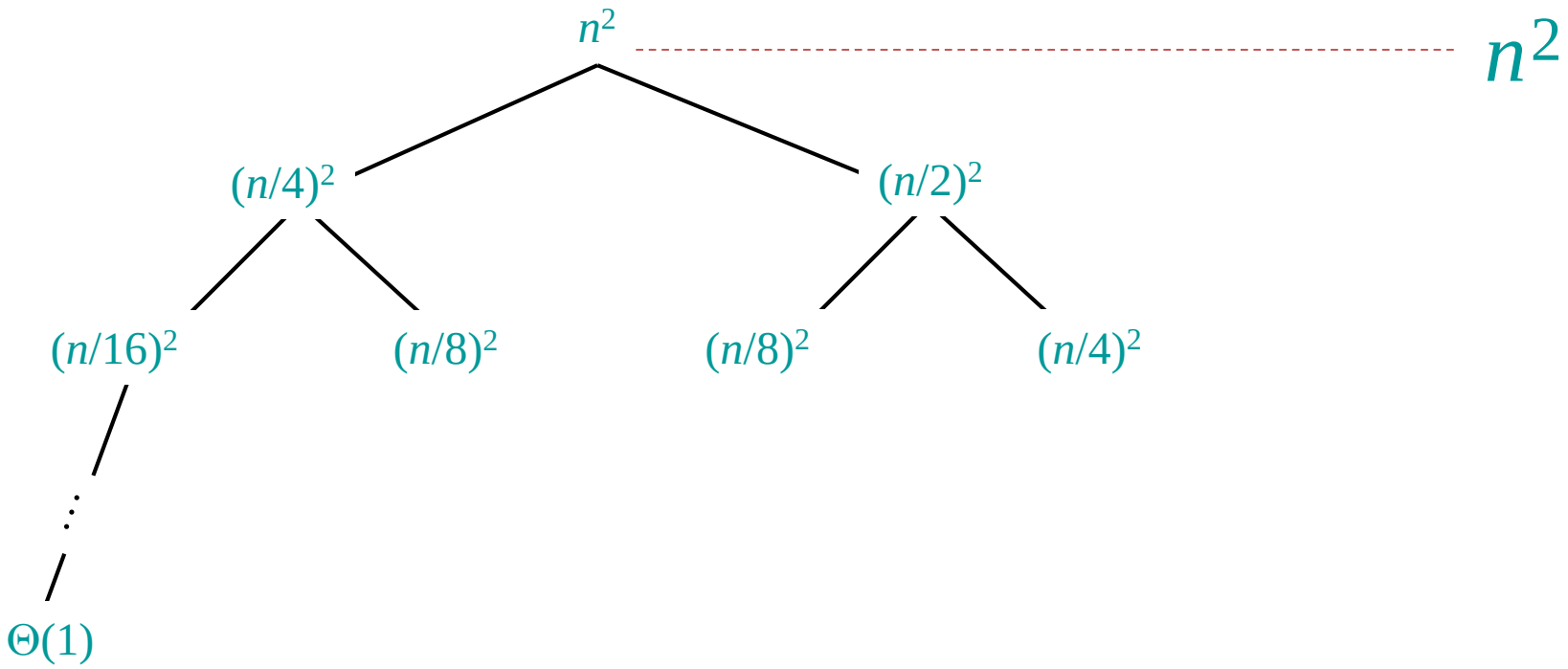
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



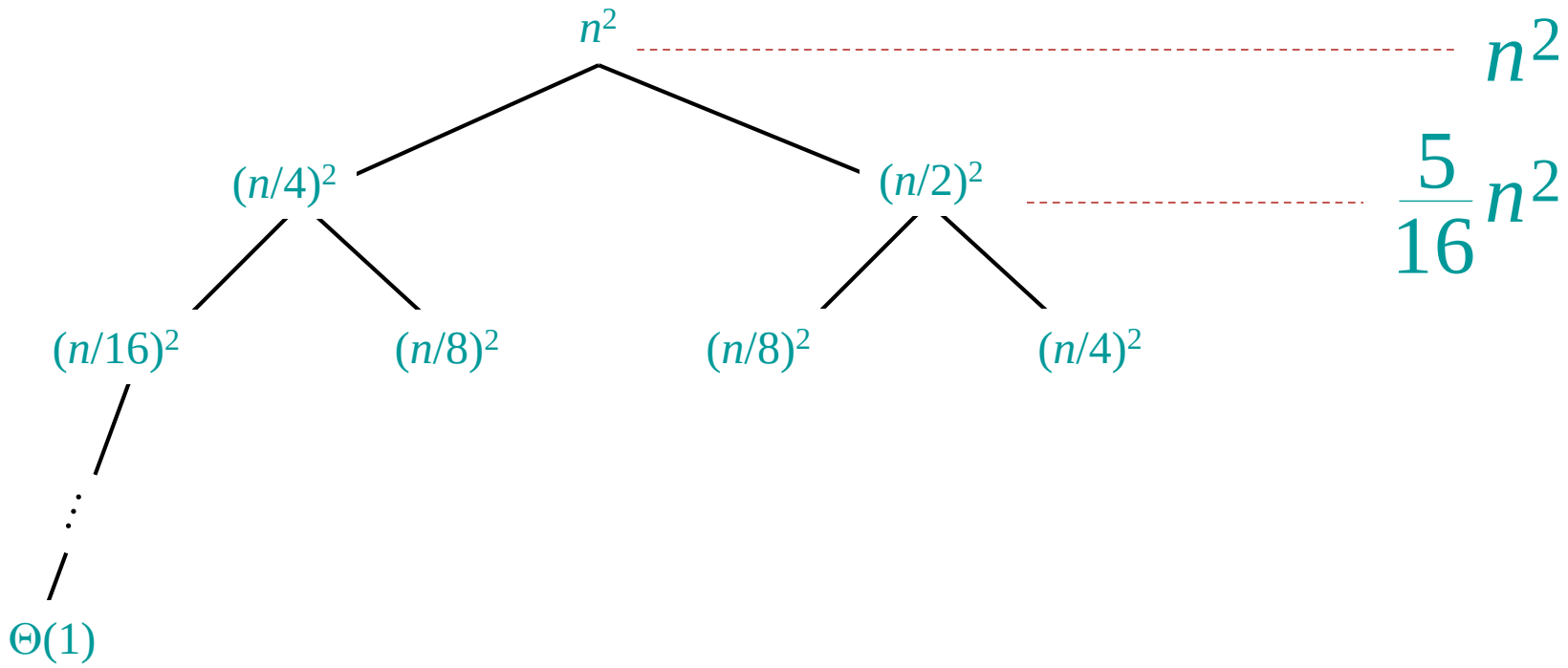
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



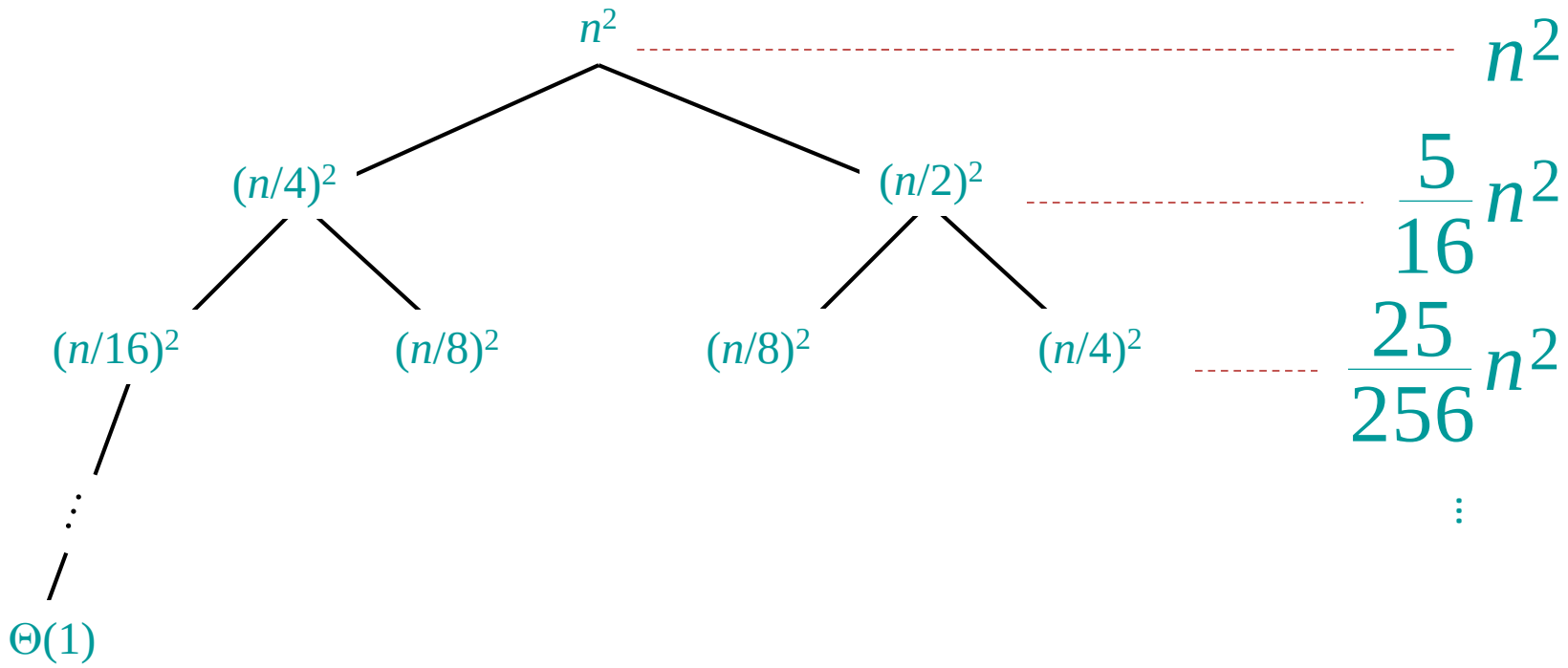
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



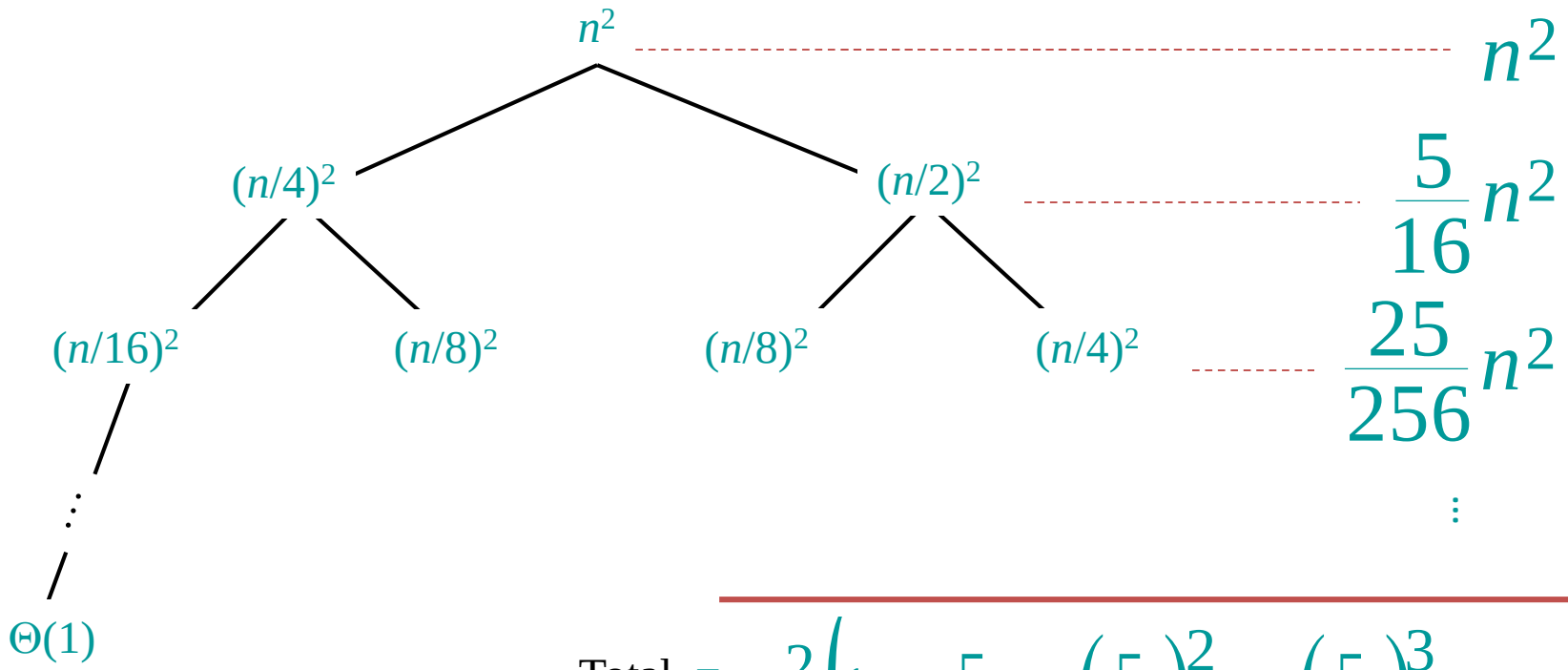
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$= \Theta(n^2)$ *geometric series*

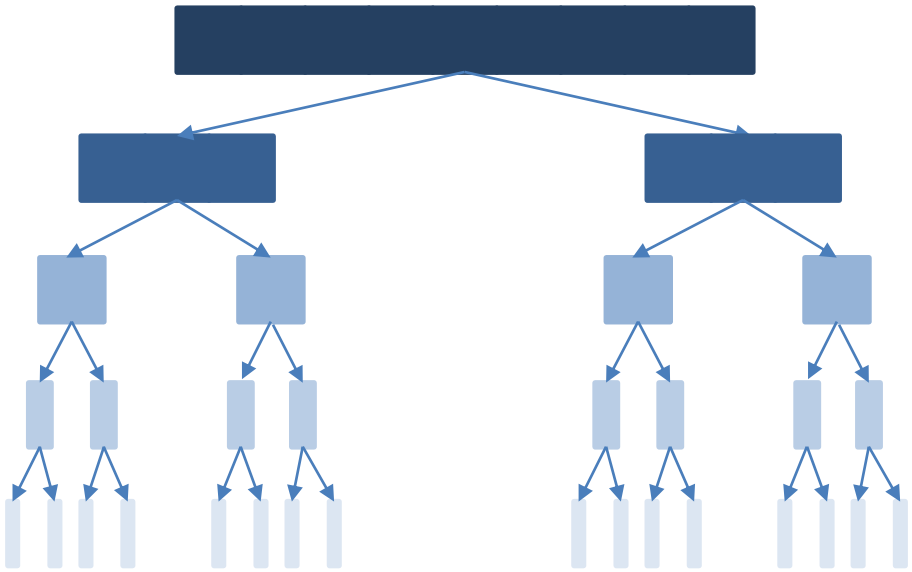
Appendix: geometric series

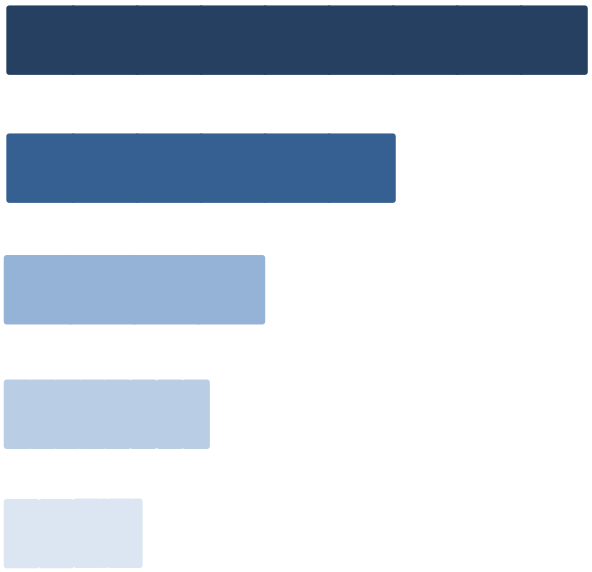
$$1 + x + x^2 + \cdots + x^n = \frac{1 - x^{n+1}}{1 - x} \quad \text{for } x \neq 1$$

$$1 + x + x^2 + \cdots = \frac{1}{1 - x} \quad \text{for } |x| < 1$$

Master Theorem

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

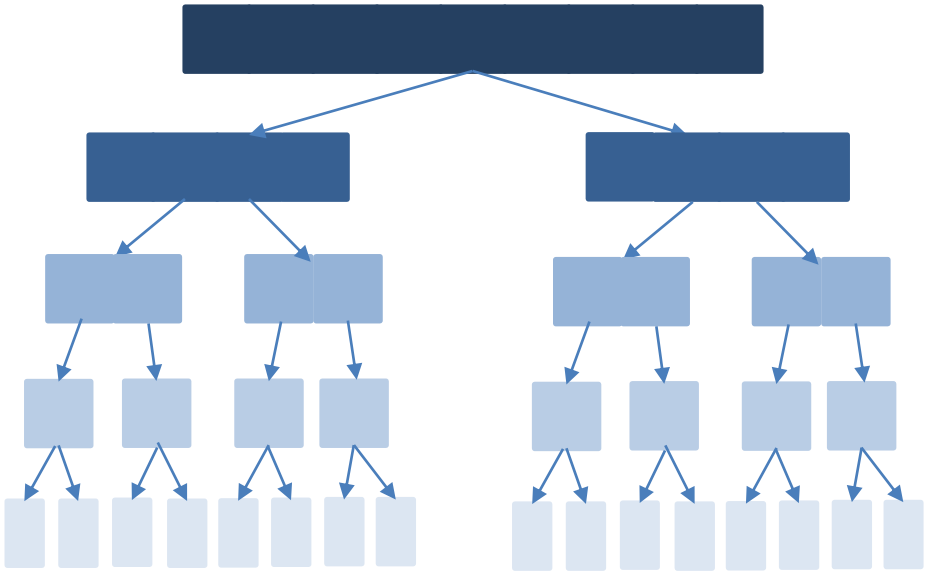


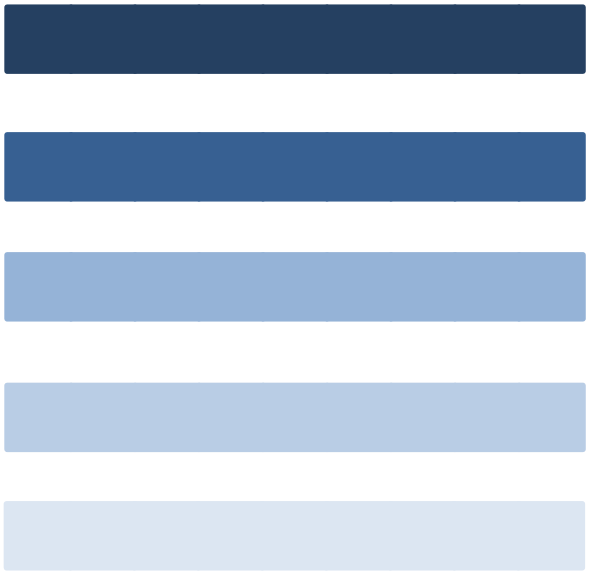


$$n \sum_{k=0}^{\log n} \left(\frac{2}{3}\right)^k < n\left(\frac{1}{1-\frac{2}{3}}\right) = 3n = \theta(n)$$

Master Theorem

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

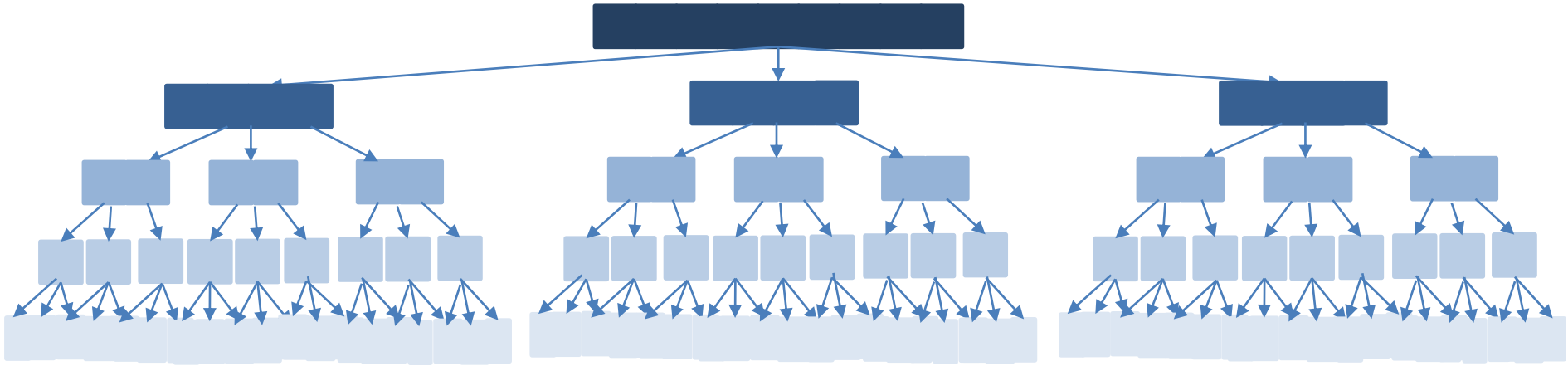




$$\sum_{k=0}^{\log_2 n} n = n \log_2 n = \theta(n \log n)$$

Master Theorem

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



$$n \sum_{k=0}^{\log n} \left(\frac{3}{2}\right)^k = n\left(\left(\frac{3}{2}\right)^{1+\log_2 n} - 1\right) = n\left(\left(\frac{3}{2}\left(\frac{3^{\log_2 n}}{2^{\log_2 n}}\right)\right) - 1\right)$$

$$= n\left(\left(\frac{3}{2}\left(\frac{n^{\log_2 3}}{n}\right)\right) - 1\right) = \frac{3}{2}n^{\log_2 3} - n = \theta\left(n^{\log_2 3}\right)$$

Master Theorem

$T(n) = aT(n/b) + f(n)$ $\quad a \geq 1, b > 1, \quad$ Let $c = \log_b a$

- If $f(n) = O(n^{c-\varepsilon})$ for some constant $\varepsilon > 0$, then
$$T(n) = \Theta(n^c)$$

- If $f(n) = \Theta(n^c)$ for some constant $\varepsilon > 0$, then
$$T(n) = \Theta(n^c \log n)$$

- If $f(n) = O(n^{c+\varepsilon})$ for some constant $\varepsilon > 0$
and if $a f(n/b) \leq d f(n)$ for some constant $d < 1$
and all sufficiently large n, then
$$T(n) = \Theta(f(n))$$

Master Theorem

$T(n) = aT(n/b) + f(n) \quad a \geq 1, b > 1$

Let $c = \log_b a$

- $f(n) = O(n^{c-\varepsilon}) \rightarrow T(n) = \Theta(n^c)$
- $f(n) = \Theta(n^c) \rightarrow T(n) = \Theta(n^c \log n)$
- $f(n) = \Omega(n^{c-\varepsilon}) \rightarrow T(n) = \Theta(f(n))$

Master Theorem : Example

$T(n) = 9T(n/3) + n$

$a = 9,\ b = 3,\ c = \log_b a = 2,\ n^c = n^2$

$f(n) = n = \mathrm{O}(\ n^{2-0.1}\ )$

$T(n) = \Theta(\ n^c\ ) = \Theta(n^2)$

Master Theorem : Example

$T(n) = 3T(n/4) + n \log n$

$a = 3,\ b = 4,\ c = \log_b a = < 0.793,\ n^c < n^{0.793}$

$f(n) = n \log n = \Omega(n^{0.793})$

$a\, f(n/b) = 3((n/4)\log(n/4)) \leq (3/4)\, n \log n = d\, f(n)$

$T(n) = \Theta(f(n)) = \Theta(n \log n)$

Exercises

- $T(n) = 2T(n/2) + 1$
- $T(n) = T(n/2) + 1$
- $T(n) = 4T(n/3) + n^3$

Master Theorem: Pitfalls

- You cannot use the Master Theorem if
  - $T(n)$ is not monotone, e.g. $T(n) = \sin(x)$
  - $f(n)$ is not a polynomial, e.g., $T(n) = 2T(n/2) + 2^n$
  - b cannot be expressed as a constant, e.g. $T(n) = \sqrt{n}$

- Note that the Master Theorem does not solve the recurrence equation
- Does the base case remain a concern?

Master Theorem

- $T(n) = 2T(n/2) + n \log n$

$a=2, b=2 , c =\log_2 2 = 1$

$n \log n \neq O(n^{1 - \varepsilon})$

$n \log n \neq \Theta(n)$

$n \log n \neq \Omega(n^{1 + \varepsilon})$

Failed

‘Fourth’ Condition

- Recall that we cannot use the Master Theorem if $f(n)$, the non-recursive cost, is not a polynomial
- There is a limited 4th condition of the Master Theorem that allows us to consider polylogarithmic functions
- **Corollary**: If $f(n) \in \Theta(n^{\log_b a} \log^k n)$ for some $k \geq 0$ then

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

- This final condition is fairly limited and we present it merely for sake of completeness.. Relax ☺

"Fourth" Condition: Example

- Say we have the following recurrence relation

$$T(n) = 2\,T(n/2) + n \log n$$

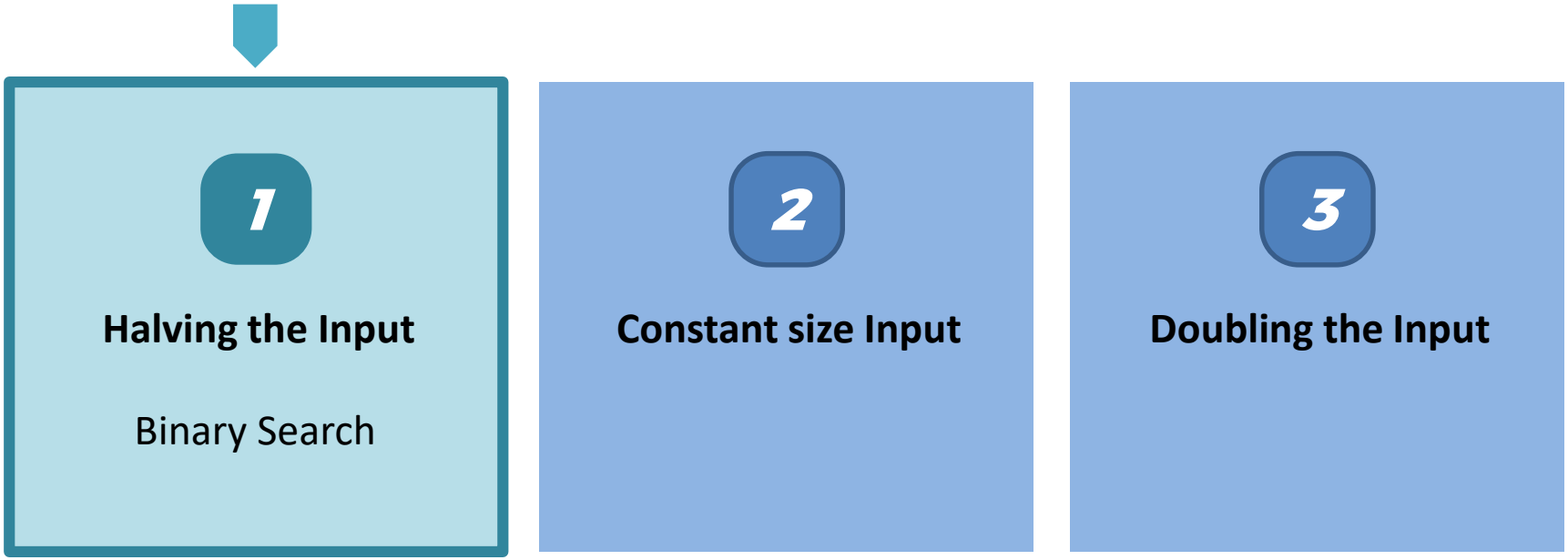
- Clearly, $a = 2, b = 2$, but $f(n)$ is not a polynomial. However, we have $f(n)$ $(n \log n), k = 1$
- Therefore by the 4th condition of the Master Theorem we can say that

$$T(n) \in \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(n^{\log_2 2} \log^2 n) = \Theta(n \log^2 n)$$

Case Study

- **Analyzing Recursive Code**

Recursive code usually falls into one of 3 common patterns:



Case Study: Binary Search

```
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
  if (hi < lo) {
    return -1;
  } else if (hi == lo) {
    if (arr[hi] == toFind) {
      return hi;
    }
    return -1;
  }

  int mid = (lo + hi) / 2;
  if (arr[mid] == toFind) {
    return mid;
  } else if (arr[mid] < toFind) {
    return binarySearch(arr, toFind, mid+1, hi);
  } else {
    return binarySearch(arr, toFind, lo, mid-1);
  }
}
```

Base Cases

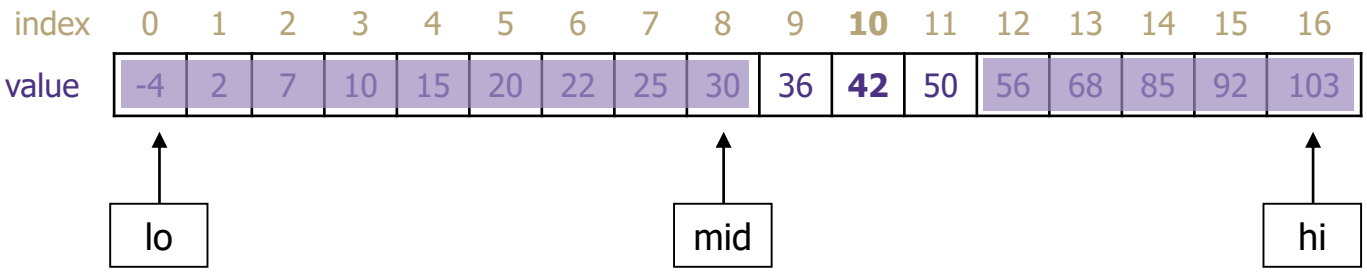
Recursive Cases

Note: the parameters passed to recursive call *reduce* the size of the problem!

Binary Search Runtime

Binary search: An algorithm to find a target value in a *sorted* array or list by successively eliminating half of the array from consideration.

- Example: Searching the array below for the value 42:



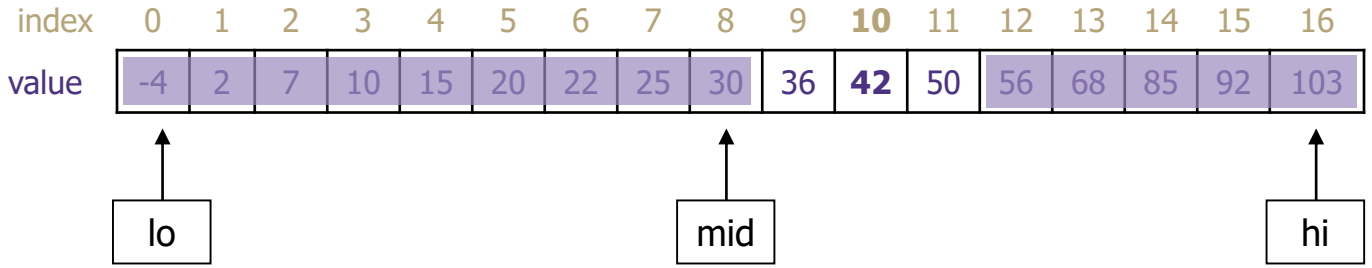
Let's consider the runtime of Binary Search

What's the first step?

Binary Search Runtime

Binary search: An algorithm to find a target value in a *sorted* array or list by successively eliminating half of the array from consideration.

- Example: Searching the array below for the value 42:



What's the Best Case?

Element found at first index examined (index 8) $f(n) = 1$ $\Rightarrow \Theta(1)$

What's the Worst Case?

Element not found, cut input in half, then in half again... **???**

1 **Halving the Input**

Binary Search Runtime

- For an array of size n, eliminate ½ until 1 element remains.

  $n, n/2, n/4, n/8, \ldots, 4, 2, 1$

  – How many divisions does that take?

- Think of it from the other direction:
  – How many times do I have to multiply by 2 to reach n?

  $1, 2, 4, 8, \ldots, n/4, n/2, n$

  – Call this number of multiplications "x".

  $2^x = n$

  $x = log_2 n$

- Binary search is in the **logarithmic** complexity class.

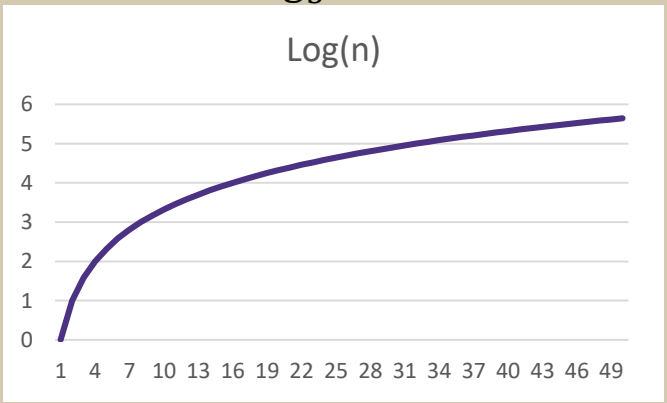
Logarithm – inverse of exponentials

$y = \log_b x$ *is equal to* $b^y = x$

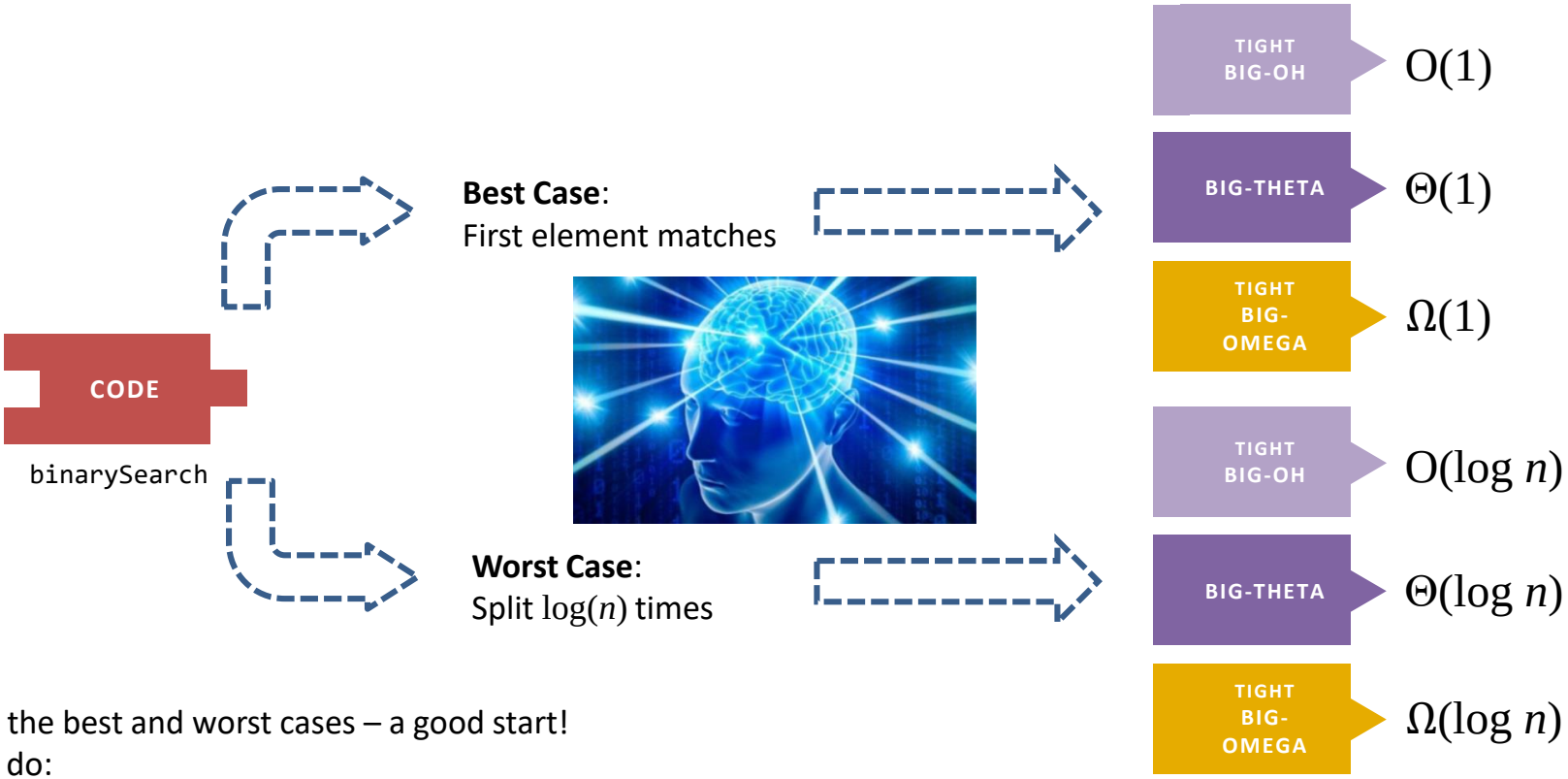
Examples:

$2^2 = 4 \Rightarrow 2 = \log_2 4$

$3^2 = 9 \Rightarrow 2 = \log_3 9$



We Just Saw: A Leap of Intuition



- We identified the best and worst cases – a good start!
- But we didn't do:
  - Step 1: model the code as a function
  - Step 2: analyze that function to find its bounds

Modeling Binary Search

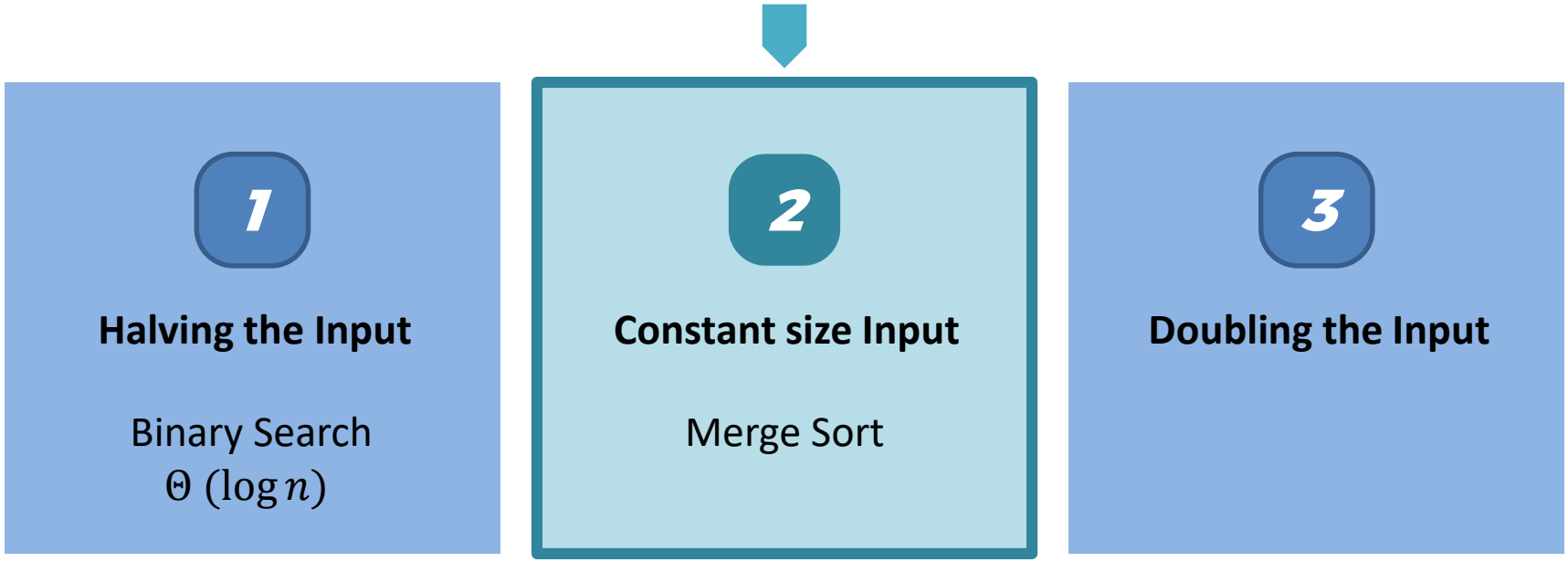
```
public int binarySearch(int[] arr, int toFind, int lo, int hi) {
  if (hi < lo) {
    return -1;
  } else if (hi == lo) {
    if (arr[hi] == toFind) {
      return hi;
    }
    return -1;
  }

  int mid = (lo + hi) / 2;
  if (arr[mid] == toFind) {
    return mid;
  } else if (arr[mid] < toFind) {
    return binarySearch(arr, toFind, mid+1, hi);
  } else {
    return binarySearch(arr, toFind, lo, mid-1);
  }
}
```

$$T(n) = T(n/2) + 1$$

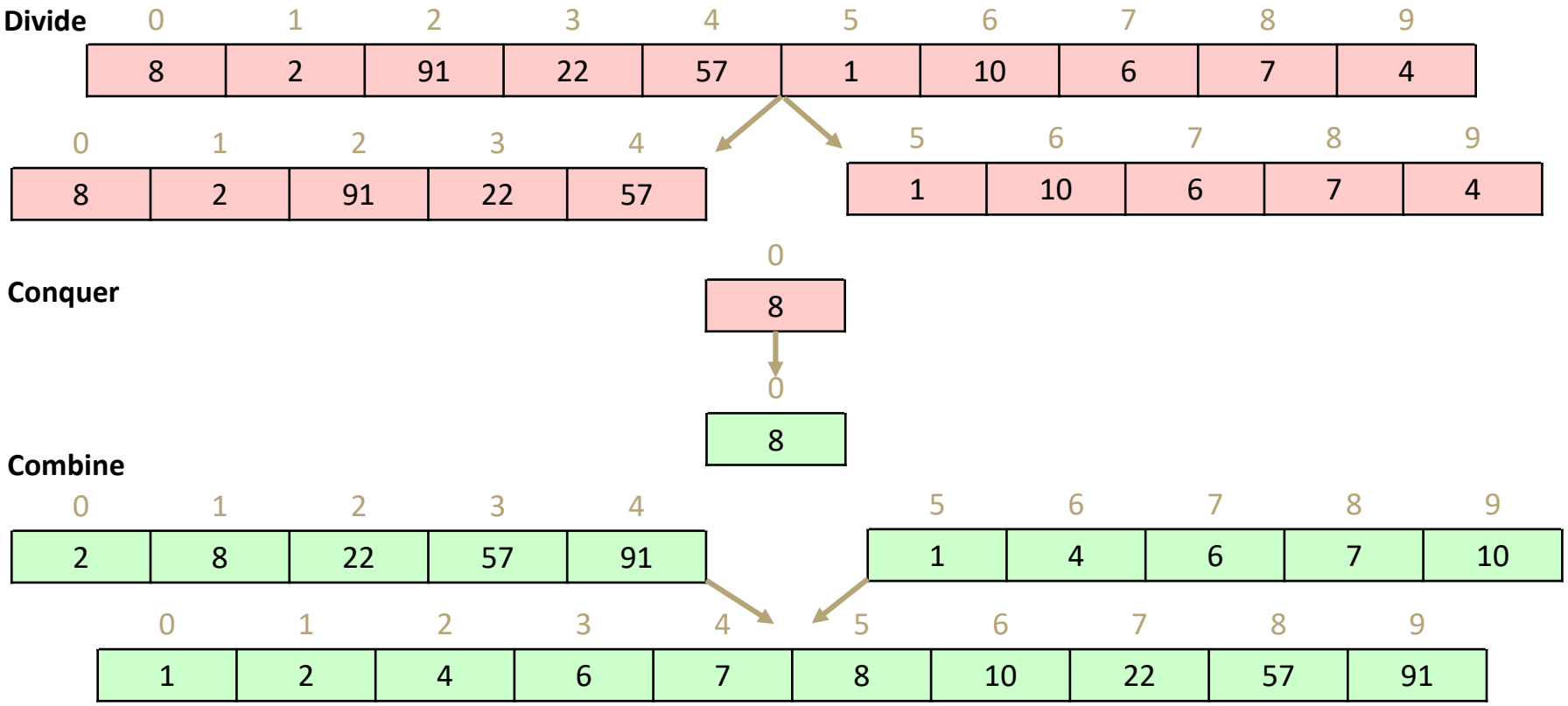
Case Study

- **Analyzing Recursive Code**

Recursive code usually falls into one of 3 common patterns:



Merge Sort

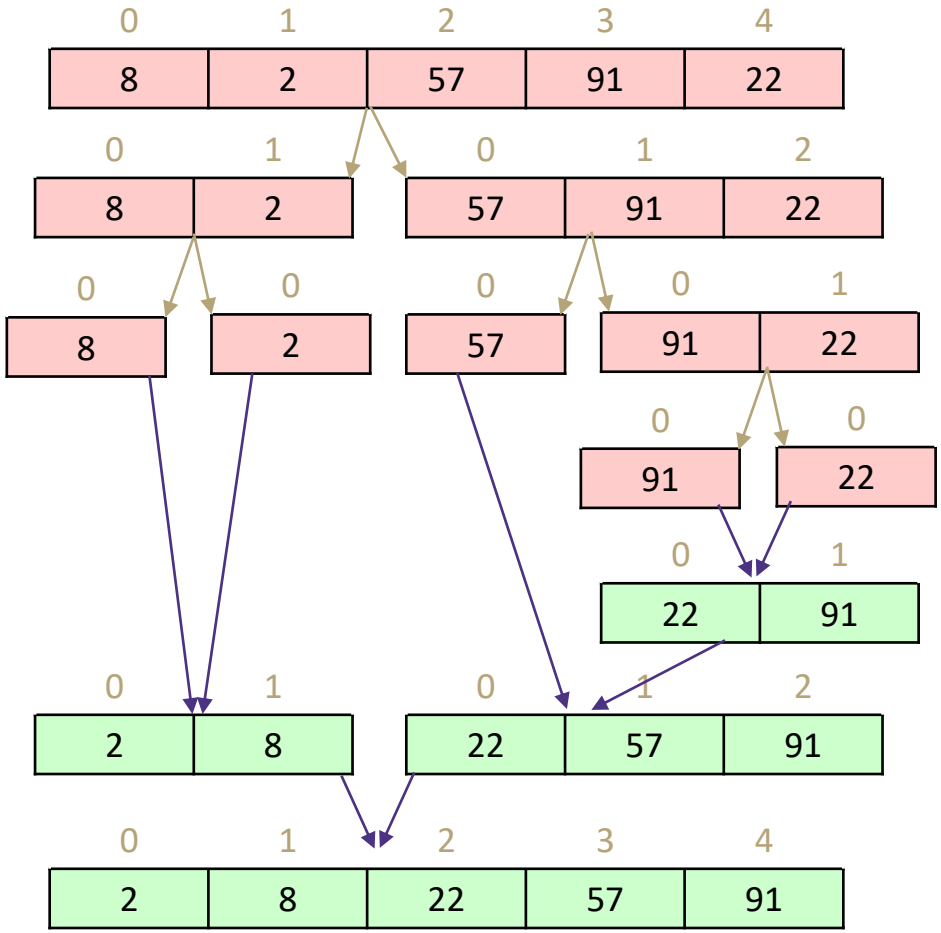

Divide
0 1 2 3 4 5 6 7 8 9
8 2 91 22 57 1 10 6 7 4
0 1 2 3 4
8 2 91 22 57
5 6 7 8 9
1 10 6 7 4
Conquer
0
8
0
8
Combine
0 1 2 3 4
2 8 22 57 91
5 6 7 8 9
1 4 6 7 10
0 1 2 3 4 5 6 7 8 9
1 2 4 6 7 8 10 22 57 91

Merge Sort

```
mergeSort(input) {
    if (input.length == 1)
        return
    else
        smallerHalf = mergeSort(new [0, ..., mid])
        largerHalf = mergeSort(new [mid + 1, ...])
        return merge(smallerHalf, largerHalf)
}
```

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

2 **Constant size Input**



Merge Sort

What is the Big-Theta of worst-case Merge Sort?

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

MASTER THEOREM

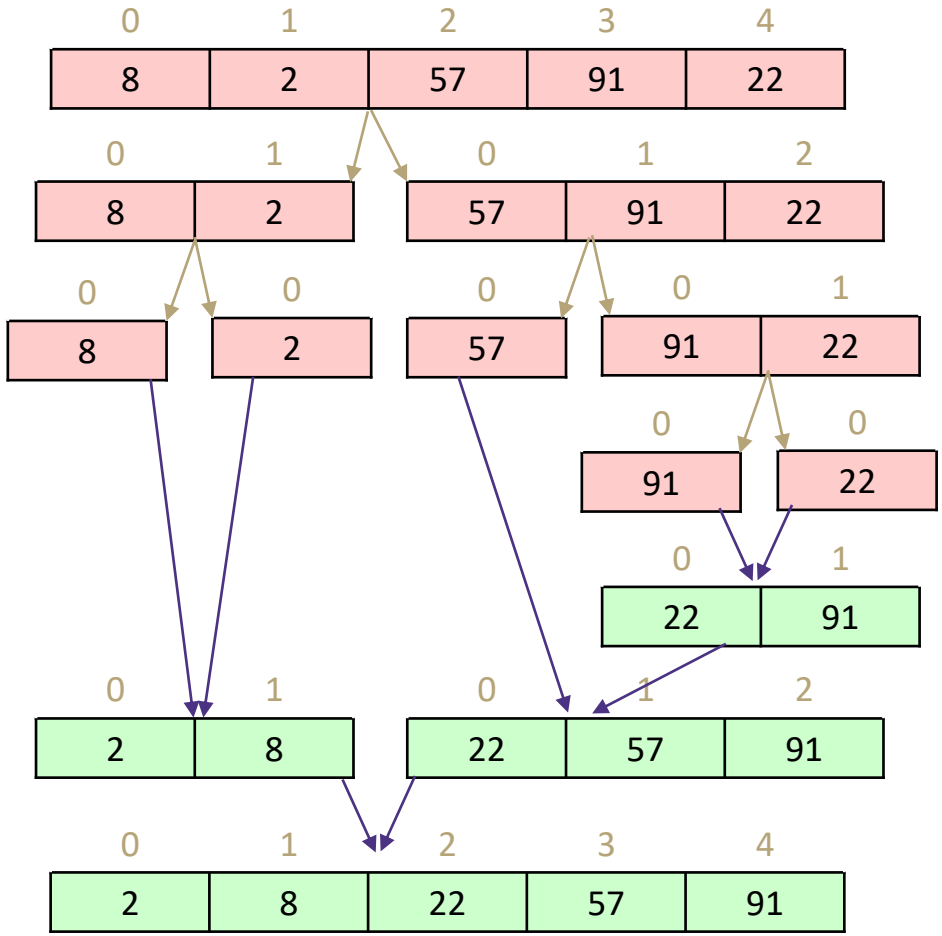
$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$



Merge Sort Recurrence to Big-Θ

$$T(n) = \begin{cases} 1 & \text{if } n \leq 3 \\ 2T\left(\frac{n}{2}\right) + n & \text{otherwise} \end{cases}$$

MASTER THEOREM

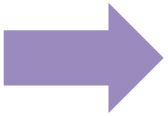
$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Where $f(n)$ is $\Theta(n^c)$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$



$a = 2 \quad b = 2 \quad and\ c = 1$

$\log_2 2 = 1$

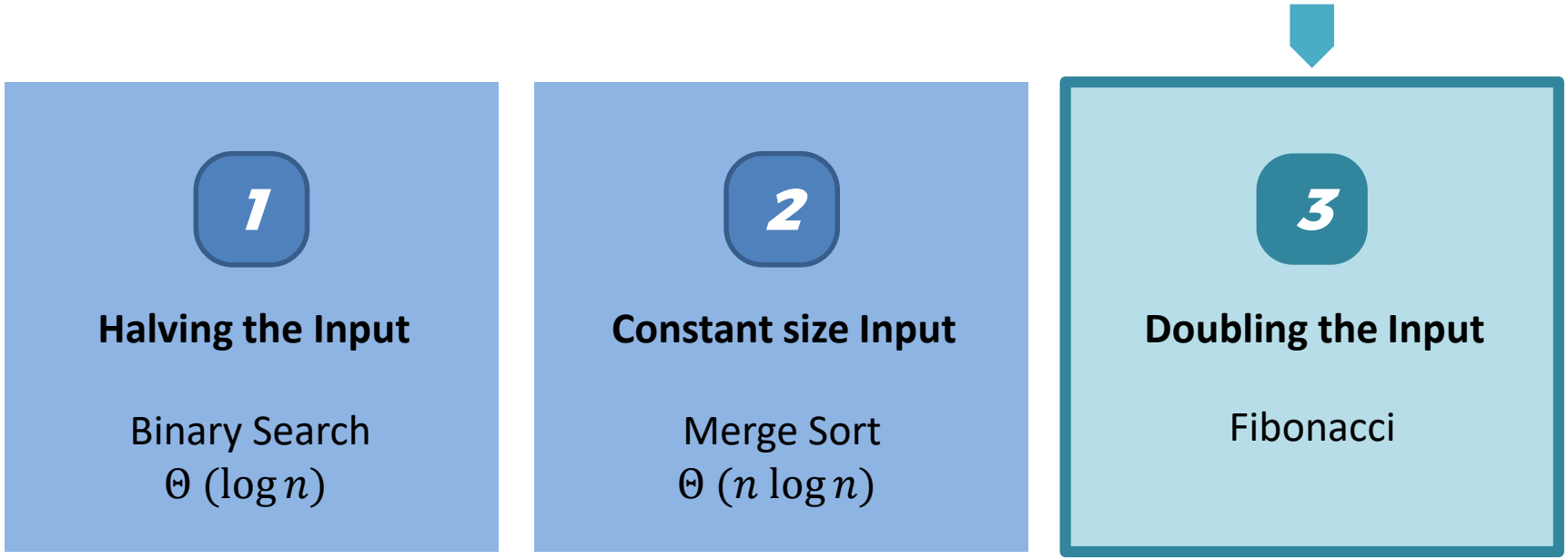
We're in case 2

$T(n) \in \Theta(n \log n)$

Case Study

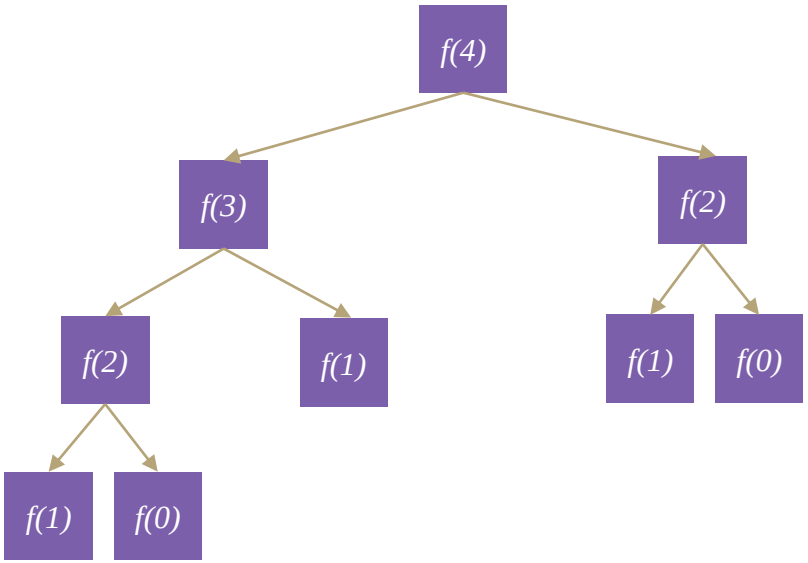
- **Analyzing Recursive Code**

Recursive code usually falls into one of 3 common patterns:



Calculating Fibonacci

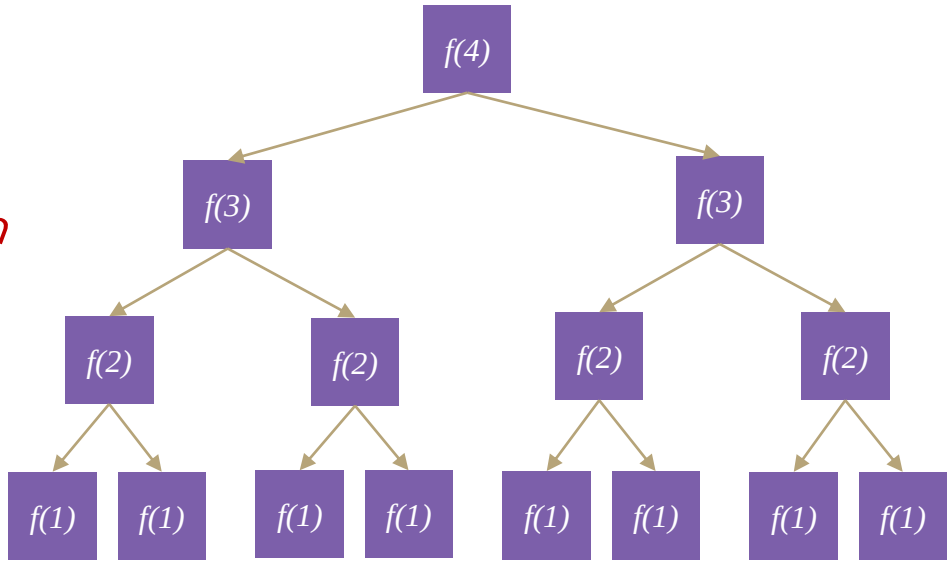
```
public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-2);
}
```



Calculating Fibonacci

```
public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-1);
}
```

Worsen



- Each call creates 2 more calls
- Each new call has a copy of the input, almost
- Almost doubling the input at each call

Almost

3 **Doubling the Input**

Fibonacci Recurrence to Big-Θ

```
public int fib(int n) {
    if (n <= 1) {
        return 1;
    }
    return fib(n-1) + fib(n-1);
}
```

d

2T(n-1) + c

$$T(n) = \begin{cases} d & \text{if } n \leq 1 \\ 2T(n-1) + c & \text{otherwise} \end{cases}$$

Can we use the Master Theorem?

MASTER THEOREM

$$T(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise} \end{cases}$$

Uh oh, our model doesn't match that format…

Can we intuit a pattern?

$T(1) = d$

$T(2) = 2T(2-1) + c = 2(d) + c$

$T(3) = 2T(3-1) + c = 2(2(d) + c) + c = 4d + 3c$

$T(4) = 2T(4-1) + c = 2(4d + 3c) + c = 8d + 7c$

$T(5) = 2T(5-1) + c = 2(8d + 7c) + c = 16d + 25c$

Looks like something's happening but it's tough

Maybe geometry can help!

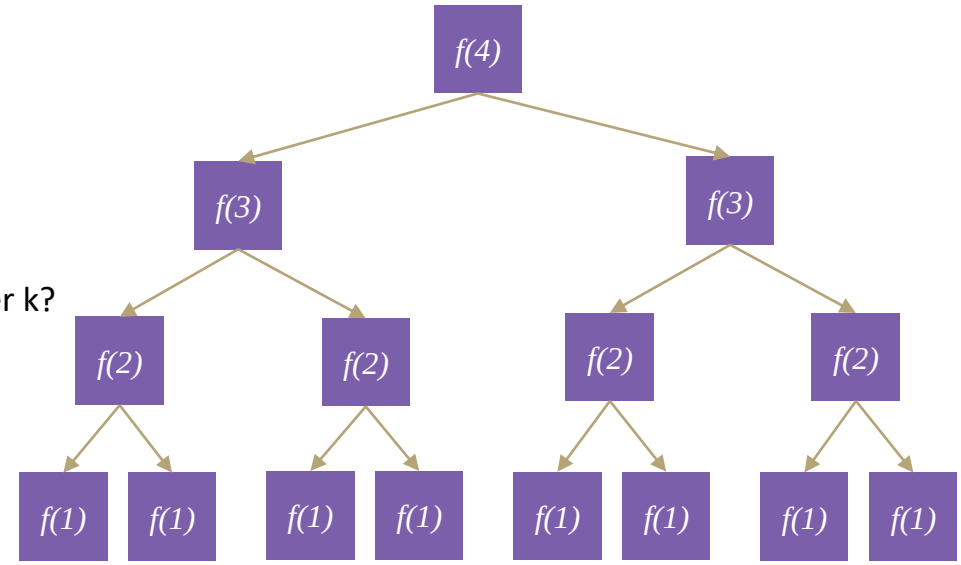
Fibonacci Recurrence to Big-Θ

How many layers in the function call tree?

How many layers will it take to transform "n" to the base case of "1" by subtracting 1

For our example, 4 -> Height = n

$$T(n) = \begin{cases} d \text{ when } n \leq 1 \\ 2T(n-1) + c \text{ otherwise} \end{cases}$$



How many function calls per layer?

LAYER	FUNCTION CALLS
1	1
2	2
3	4
4	8

How many function calls on layer k?

2^{k-1}

How many function calls TOTAL for a tree of k layers?

$1 + 2 + 3 + 4 + \ldots + 2^{k-1}$

Fibonacci Recurrence to Big-Θ

- Patterns found:

  How many layers in the function call tree? n

  How many function calls on layer k? 2^{k-1}

  How many function calls TOTAL for a tree of k layers?

  $1 + 2 + 4 + 8 + \ldots + 2^{k-1}$

  Total runtime = (total function calls) x (runtime of each function call)

  Total runtime = $(1 + 2 + 4 + 8 + \ldots + 2^{k-1})$ x (constant work)

  $1 + 2 + 4 + 8 + \ldots + 2^{k-1} = \sum_{i=1}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$

  Summation Identity
  Finite Geometric Series

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

  $\boldsymbol{T(n) = 2^n - 1 \in \Theta(2^n)}$

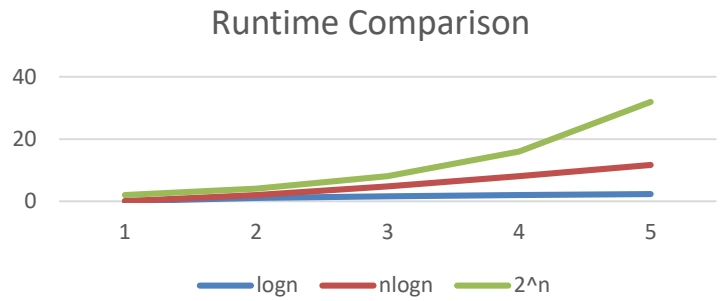
Fibonacci Recurrence to Big-Θ

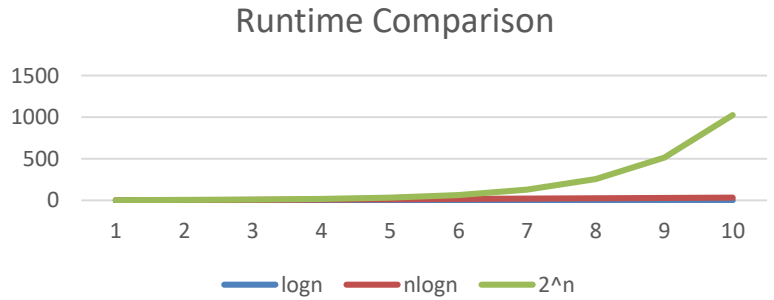
How many layers in the function call tree?	n
How many function calls on layer k?	2^{k-1}
How many function calls TOTAL for a tree of k layers?	$1 + 2 + 4 + 8 + \ldots + 2^{k-1}$
Total runtime = (total function calls) * (runtime of each function call)	$(1 + 2 + 4 + 8 + \ldots + 2^{k-1})$ x (constant work) $1 + 2 + 4 + 8 + \ldots + 2^{k-1} = \sum_{i=1}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1$ $\boldsymbol{T(n) = 2^n - 1 \in \Theta(2^n)}$

Summation Identity
Finite Geometric Series

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

3 Patterns for Recursive Code





1

Halving the Input

Binary Search

$\Theta(\log n)$

2

Constant size Input

Merge Sort

$\Theta(n \log n)$

3

Doubling the Input

Fibonacci

$\Theta(2^n)$

Conclusion

- แบ่งอัลกอริทึมออกเป็น blocks ตาม control structures
- วิเคราะห์แต่ละ block
- วิเคราะห์ความสัมพันธ์ของ blocks ตาม control structure
- จะง่ายขึ้นถ้าวิเคราะห์แบบ asymptotic
- ถ้าเจอ recurrence เริ่มด้วย Master Theorem
- ถ้าไม่เข้ากรณีใดเลยก็กลับไป recursion tree