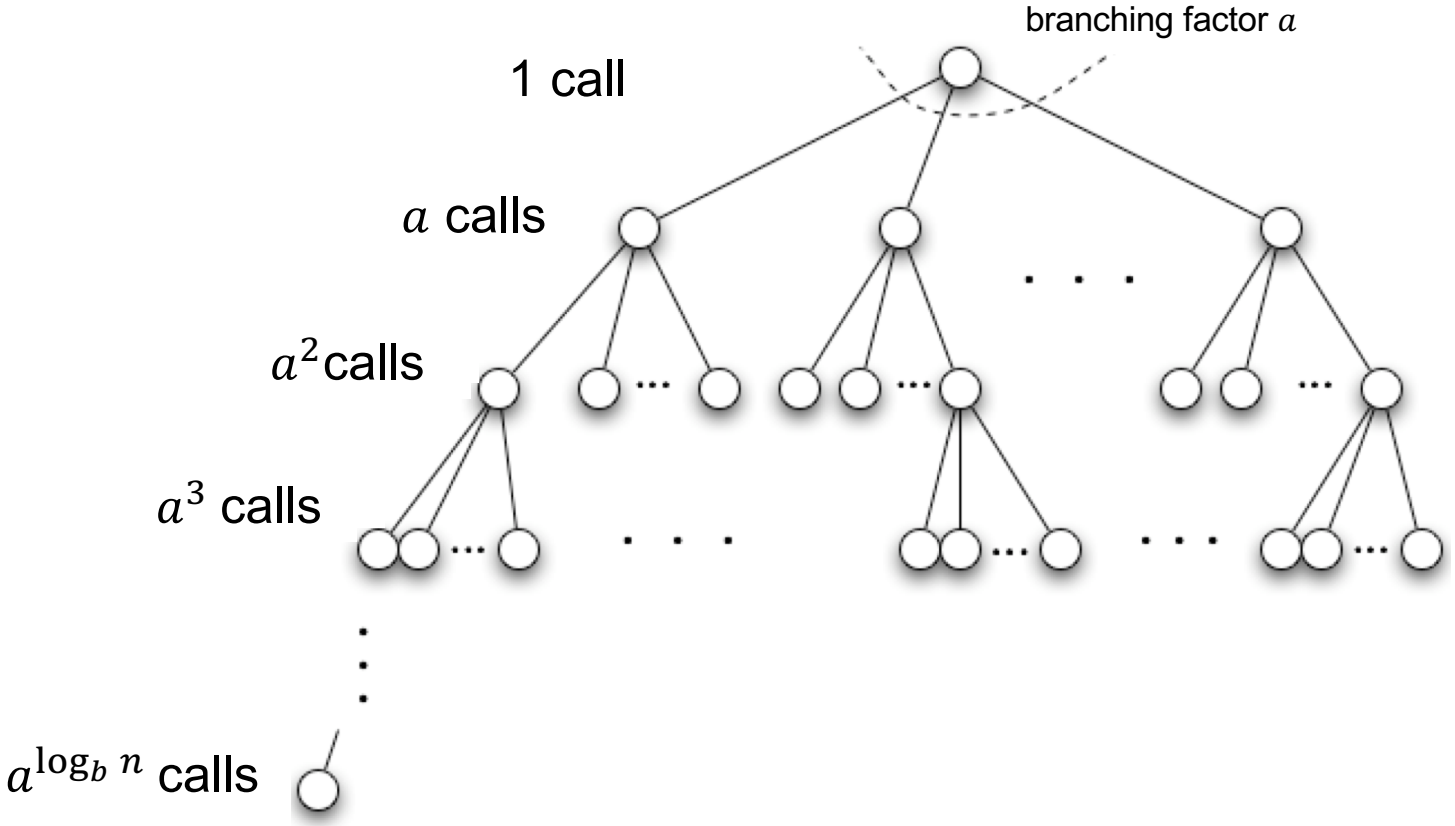
EECS 203 Stickers are here!
Come pick yours up before/after class.

L28: (More) Complexity & Divide and Conquer Recurrences



Announcements

- **EECS 203 Stickers are here!**
  - **Grab one before or after class**
- **[Ungraded] Homework 11 will post tonight**
  - **Do it! It's the best way to master recurrence relations and Big-O, etc. before the exam.**
  - **There's also a "Discussion 11" to give you more practice/guidance**
  - **Nothing to submit for hw 11**

- **Today: HW/GW 10 due**
- **Tomorrow: Surveys due**
  - **2 are part of your grade**
  - **1 is for extra credit**
- **Friday**
  - **Grading of GW 10 due**
  - **Grade Review deadline**

Exam 3 Planning

	Sunday	Mon	Tuesday	Weds.	Thursday	Friday	Saturday
This week			*Last day of classes* Today HW 10 due Practice Exam Solutions out	[Ungraded] HW 11 released Surveys due Special Topics Review #1 4-6pm	Special Topics Review #2 4-6pm	GW 10 Grading due Deadline to review your grades	Review session #1: 1-4 pm
Next week	Review session #2: 1-4 pm		**4/26:** **Exam 2: 7-9pm**				

- Exam 3 covers: Graphs, Counting, Probability, Recurrence Relations, Complexity
  - (HW 8 – 11)
- Please note: the material is *inherently cumulative*
  - (e.g., you could be asked to count bijections, or *prove* something about graphs/counting/probability/etc
- See "Exam 3 Information" doc on Canvas and related Piazza post

Note: Submitting late on Gradescope = a violation of the Honor Code.

- You must join the zoom helpline before 9:30 pm to avoid a penalty and potential honor council report

Learning Objectives

After today's lecture (and associated readings, discussion & homework), you should be able to:

- Determine the runtime of an algorithm, given it's pseudocode
- Recognize a divide-and-conquer algorithm and identify its key features
  - e.g., number of subproblems, size of each subproblem
- Apply the Master Theorem to determine the runtime of divide-and-conquer algorithms

Outline

- **Complexity Recap**
- Runtimes of algorithms (with pseudocode)

- Divide and Conquer recurrences
  - Example: Merge Sort

- Master Theorem
  - Key features of a Divide and Conquer algorithm
  - Master Theorem: runtimes for divide and conquer algorithms
  - (Optional) Derive Master Theorem

Recurrence Relations Recap

- Recurrence Relations
  - $T(n)$ is defined in terms of previous values, i.e., $T(k), k < n$
  - We need initial value(s) of T to fully define the recurrence
  - Recursion = induction

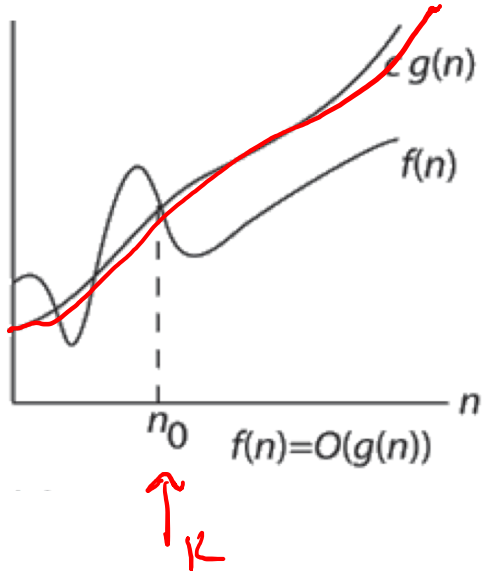
- Last time we saw some *linear* recurrences
  - ToH: $T(n) = 2T(n-1) + 1$
  - Stair climbing: $s(n) = s(n-1) + s(n-2)$
  - Pandemic outfits: $c(n) = 2c(n-1) + c2(n-2)$

- Today we'll add Divide & Conquer recurrences
  - Which take the form: $T(n) = aT\left(\frac{n}{b}\right) + cn^d$

Runtime Recap

- Let $f, g : \mathbb{N} \to \mathbb{R}$ (or can be $f, g: \mathbb{R} \to \mathbb{R}$)
- **Big-O**
  - "f is $O(g)$" means "f grows no faster than g"

$$\exists k, C > 0 \text{ such that } \forall n > k$$
$$|f(n)| \leq C|g(n)|$$

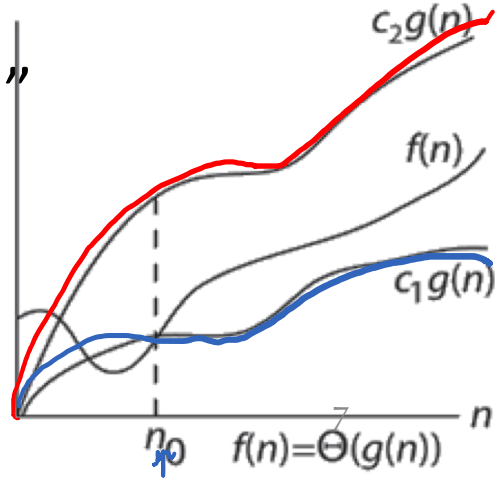


- **Big-Omega**
  - "f is $\Omega(g)$" means "f grows at least as fast as g"

$$\exists k \exists C \text{ such that } \forall n > k\text{: } |f(n)| \geq C|g(n)|$$

- ****Big-Theta****
  - "f is $\Theta(g)$" means "f grows at the same rate as g"
  - f is $\Theta(g)$ iff $f = O(g)$ and $f = \Omega(g)$

$$\exists k \exists C_1 \exists C_2 \text{ such that } \forall n > k$$
$$C_1|g(n)| \leq |f(n)| \leq C_2|g(n)|$$



Runtime Recap

$$O(1),\ \log n,\ n,\ n\log n,\ n^2, n^3, \text{(maybe } n^4, \ldots),\ 2^n,\ n!$$

Grows slowly (**better**)

Grows quickly (**worse**)

Consider positive-valued functions $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$.

- **Addition**

$$(f_1 + f_2)(n) = \Theta(\max(g_1(n), g_2(n)))$$

Keep the bigger one

- **Scalar multiplication**

$$af(n) = \Theta(f(n))$$

Doesn't change

- **Product**

$$(f_1 \cdot f_2)(n) = \Theta(g_1(n) \cdot g_2(n))$$

Multiply their runtimes

Outline

- Complexity Recap
- **Runtimes of algorithms (with pseudocode)**
- Divide and Conquer recurrences
  - Example: Merge Sort
- Master Theorem
  - Key features of a Divide and Conquer algorithm
  - Master Theorem: runtimes for divide and conquer algorithms
  - (Optional) Derive Master Theorem

Time Complexity with Pseudocode, Ex. #1

```
procedure hello(n: integer)
    sum := 0
    for i := 1 to 2n
        print “hello world”
        sum := sum + 2
```

Time Complexity with Pseudocode, Ex. #1

```
procedure hello(n: integer)
    sum := 0                          ← Θ(1)
    for i := 1 to 2n
        print "hello world"           ← Θ(1)
        sum := sum + 2                ← Θ(1)
```

Loop executes $2n$ times

$\Longrightarrow \Theta(n)$

- The loop dominates the runtime.
- Loop executes $2n$ times
- So $\Theta(n)$

Time Complexity with Pseudocode, Ex. #2

```
procedure hello_goodbye(n: integer)
    sum := 0
    for i := 1 to 2n
        print “hello world”
        sum := sum + 2

    for i := 1 to n
        for j := 1 to n
            print “goodbye”
            sum := sum + 1
```

Time Complexity with Pseudocode, Ex. #2

```
procedure hello_goodbye(n: integer)
    sum := 0
    for i := 1 to 2n
        print "hello world"
        sum := sum + 2

    for i := 1 to n
        for j := 1 to n
            print "goodbye"
            sum := sum + 1
```

Loop executes $2n$ times

$\Longrightarrow \quad \Theta(n)$

For each iteration of the outer loop, the inner loop executes n times

Outer loop executes n times

$\Longrightarrow \quad \Theta(n^2)$

n^2 grows faster than n, so the nested loops dominate the runtime. The algorithm is $\Theta(n^2)$

Time Complexity with Pseudocode, Ex. #3

```
procedure foo(n: integer)
    a := 0
    i := 1
    while i < n
        a := a + i
        i := i * 2
    return a
```

Time Complexity with Pseudocode, Ex. #3

```
procedure foo(n: integer)
    a := 0
    i := 1
    while i < n
        a := a + i
        i := i * 2
    return a
```

$\log n$

Note the update on `i`:

```
    i = 1
    i = 2
    i = 4
    i = 8
      ⋮
    until i >= n
```

Q: How many times does the loop execute?

A: $\log_2 n$

So this algorithm is $\Theta(\log n)$

Time Complexity of Matrix Multiplication

```
procedure square_matrix_mult(A: nxn matrix, B: nxn matrix)
    for i := 1 to n
        for j := 1 to n
            c_ij := 0
            for k := 1 to n
                c_ij := c_ij + a_ik * b_kj

    return C
```

Time Complexity of Matrix Multiplication

```
procedure square_matrix_mult(A: nxn matrix, B: nxn matrix)
    for i := 1 to n
        for j := 1 to n
            c_ij := 0
            for k := 1 to n
                c_ij := c_ij + a_ik * b_kj
    return C
```

n (innermost loop)

n (middle loop)

n (outer loop)

$$\Theta(n^3)$$

Time Complexity of Linear Search

```
procedure linear_search(x: integer, a1, a2, …, an: integers)
    i := 1

    while(i ≤ n and x ≠ a_i )
        i := i + 1

    if i ≤ n then location := i
    else location := 0

    return location
```

Time Complexity of Linear Search

```
procedure linear_search(x: integer, a1, a2, …, an: integers)
    i := 1

    while(i ≤ n and x ≠ a_i )
        i := i + 1

    if i ≤ n then location := i
    else location := 0

    return location
```

Loop executes at most n times

$\Longrightarrow \quad \Theta(n)$

Time Complexity of Binary Search

```
procedure binary_search (x: integer, a_1, a_2, ... , a_n:
                                          increasing integers)
    i := 1    {i is left endpoint of search interval}
    j := n    {j is right endpoint of search interval}
    while i < j
        m := ⌊(i + j)/2⌋          find the midpoint
        if x > a_m  then i:=m+1   update i to midpoint, or
        else j := m                  update j to midpoint
    if x = a_i  then location := i
    else location := 0
    return location {location is the subscript i of the term a_i
                     equal to x, or 0 if x is not found}
```

Time Complexity of Binary Search

```
procedure binary_search (x: integer, a_1, a_2, ... , a_n:
                                        increasing integers)
    i := 1     {i is left endpoint of search interval}
    j := n     {j is right endpoint of search interval}
    while i < j
        m := ⌊(i + j)/2⌋                 find the midpoint
        if x > a_m  then i:=m+1          update i to midpoint, or
        else j := m                         update j to midpoint
    if x = a_i  then location := i
    else location := 0
    return location {location is the subscript i of the term a_i
                     equal to x, or 0 if x is not found}
```

Updates cut remaining list in half each time:

j-i≈n, then n/2, then n/4, …

So loop iterates $\log n$ times.

$\Longrightarrow \quad \Theta(\log n)$

Time Complexity with Pseudocode, Ex. #4

```
procedure bar(n: integer)
    a := (n * n - 7) / 2
    for i := 1 to n
        j := n
        while j > 0
            print “hi”
            j := j / 2
    print “bye”

    for i := 1 to 500n
        print “203 is fun!”
```

Time Complexity with Pseudocode, Ex. #4

```
procedure bar(n: integer)
    a := (n * n - 7) / 2
    for i := 1 to n
        j := n
        while j > 0
            print "hi"
            j := j / 2
    print "bye"

    for i := 1 to 500n
        print "203 is fun!"
```

$\log n$ $\quad n \quad \Longrightarrow \quad \Theta(n \log n)$

$500n \quad \Longrightarrow \quad \Theta(n)$

Putting it all together, the algorithm is $\boldsymbol{\Theta(n \log n)}$

Time Complexity of Algorithms

- Lots more examples in Rosen 3.3!

Outline

- Complexity Recap
- Runtimes of algorithms (with pseudocode)

- **Divide and Conquer recurrences**
  - **Example: Merge Sort**

- Master Theorem
  - Key features of a Divide and Conquer algorithm
  - Master Theorem: runtimes for divide and conquer algorithms
  - (Optional) Derive Master Theorem

Divide-and-Conquer Recurrences

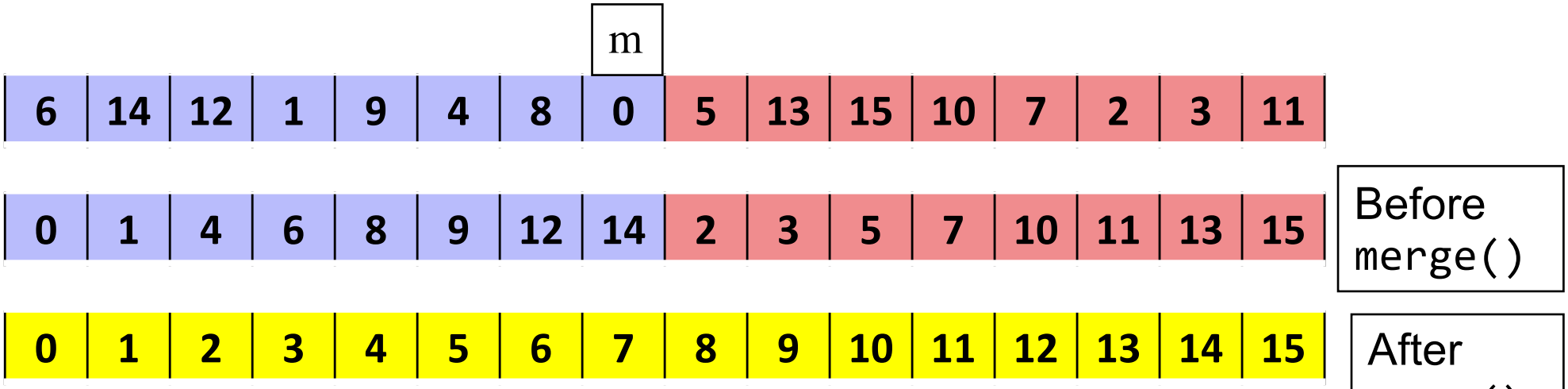
- **Divide-and-Conquer** algorithms divide a problem into non-overlapping sub-problems.
  - Subproblems are smaller, and easier to conquer.

- This leads to recurrences of the form:
  - $T(n) = aT(n/b) + \Theta(n^d)$ to estimate resource use.

- We want to find $\Theta(T(n))$.
  - The Master Theorem gives us simple formulas.

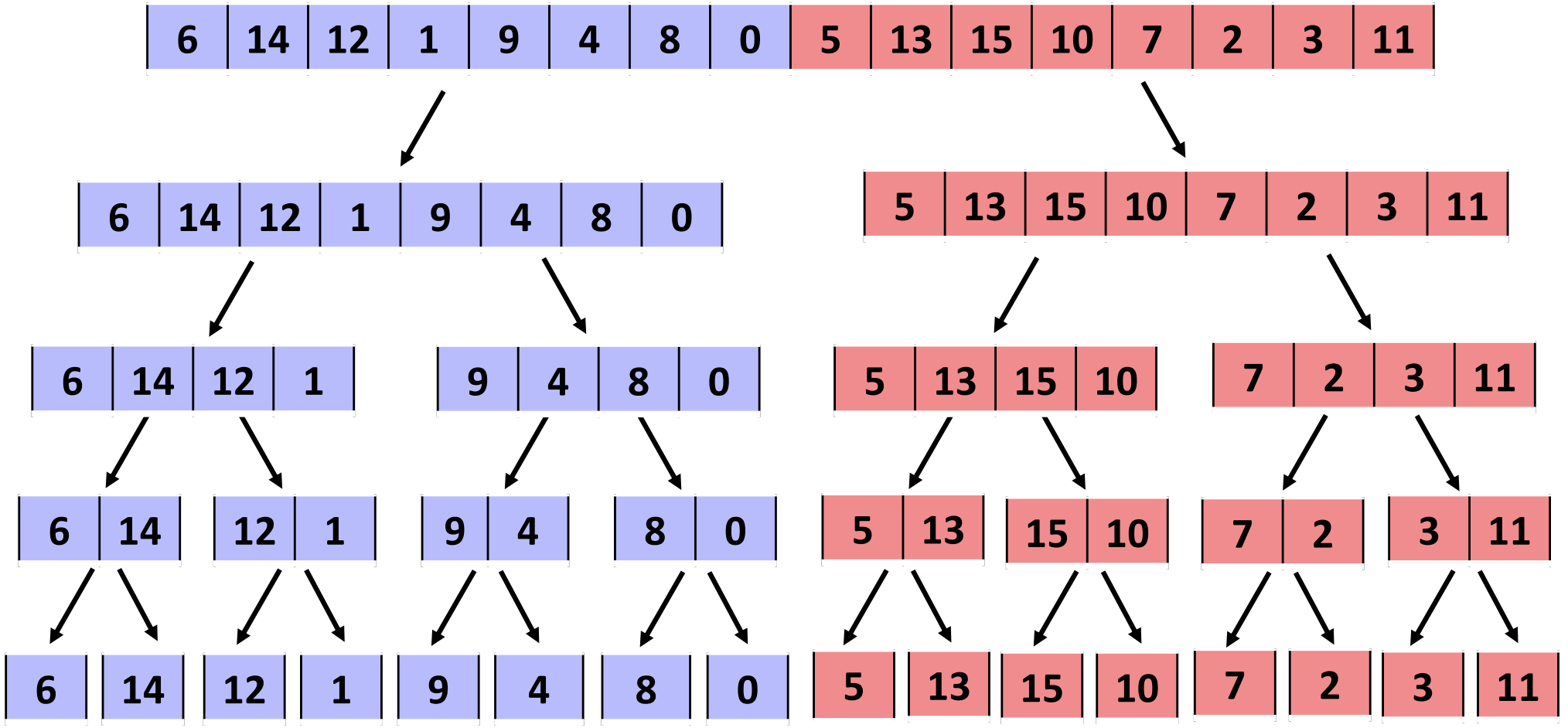
MergeSort (divide, conquer, and combine)

```
procedure MergeSort(a[1..n] : array of n integers)
    if n = 1, return;                  list of length 1 is already sorted!
    m := floor(n/2);                   find mid point
    MergeSort(a[1..m]);                sort first half recursively
    MergeSort(a[m+1..n]);              sort second half recursively
    merge(a[1..m], a[m+1..n])          combine two sorted lists: Θ(n)
    return a;
```



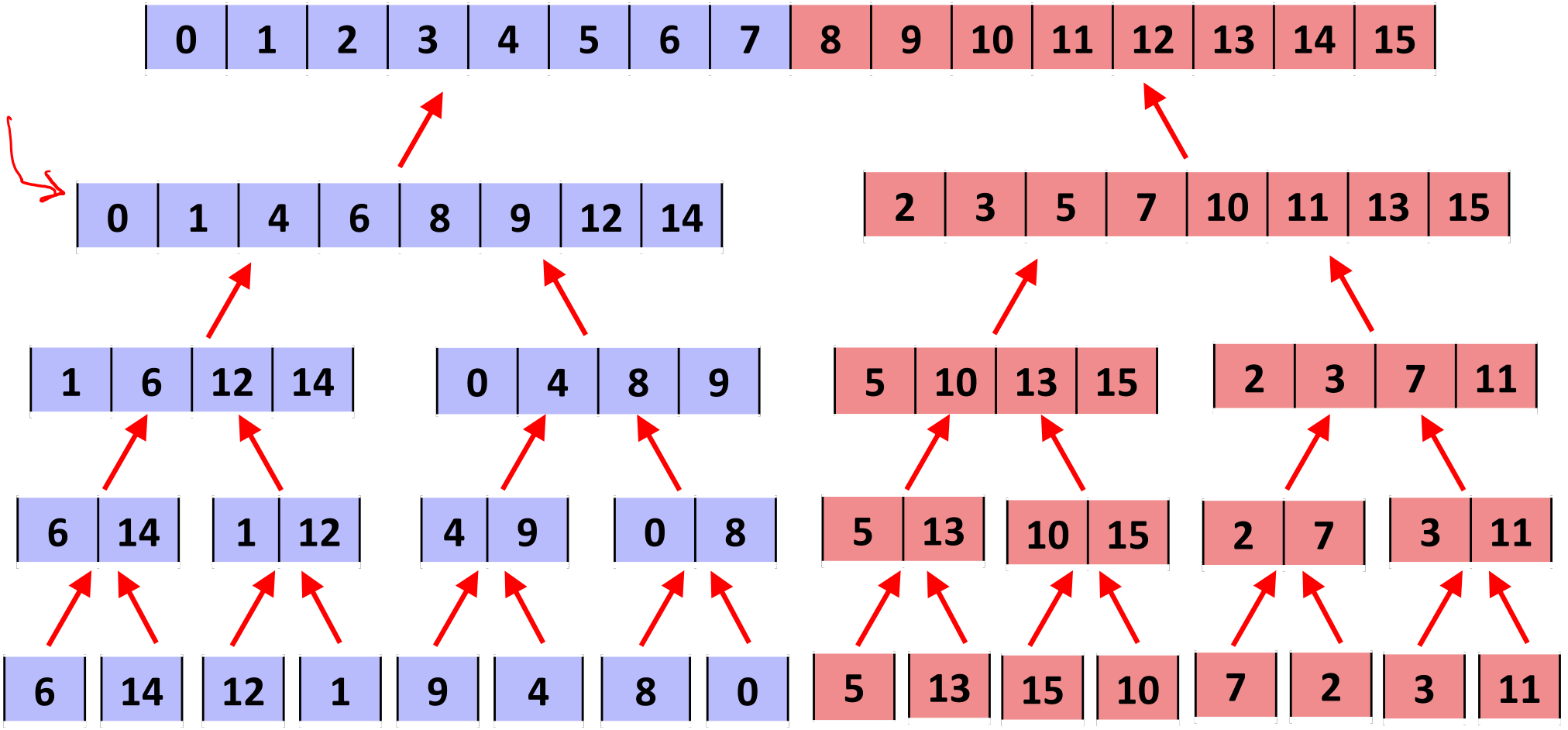
Merge Sort (the **Divide** part)

Recursive calls to *`Mergesort()`* with smaller and smaller subproblems



Merge Sort (the **Conquer**/reassemble part)

Putting the results of the subproblems back together using *merge()*



MergeSort in Action!

https://www.youtube.com/watch?v=ZRPoEKHXTJg

Runtime of MergeSort

```
procedure MergeSort(a[1..n] : array of n integers)
    if n = 1, return;          list of length 1 is already sorted!
    m := floor(n/2);           find mid point
    MergeSort(a[1..m]);        sort first half recursively
    MergeSort(a[m+1..n]);      sort second half recursively
    merge(a[1..m], a[m+1..n])  combine two sorted lists: Θ(n)
    return a;
```

- T(n) = runtime of MergeSort on inputs of size n.
  - $T(1) = \Theta(1)$
  - $T(n) = 2T(n/2) + \Theta(n)$

What is the runtime of this algorithm?

Divide-and-Conquer Algorithms

- Consider the recurrence: $T(n) = aT(n/b) + \Theta(n^d)$
- Divide-and-conquer breaks big problem $T(n)$
  - into a smaller problems,
  - each smaller problem $T(n/b)$ is of size n/b,
  - with cost of $\Theta(n^d)$ to put together the results of the smaller problems.
- Continue breaking into smaller and smaller problems
  - Until you reach depth k, where $T\left(\frac{n}{b^k}\right) = 1$
    - So $k = \log_b n$
  - (For the analysis, imagine that n is a power of b.)
    - (Rosen 8.3, explains why this is OK.)

Divide-and-Conquer Recursion

$$T(n) = aT(n/b) + \Theta(n^d)$$

- Divide-and-conquer breaks big problem $T(n)$
  - into a smaller problems,
  - each one of size n/b,
  - with cost $\Theta(n^d)$ to put the smaller results back together.

- Break smaller problems into even smaller ones
  - Until the size of each subproblem is 1
    - until $T(n/b^k) = T(1)$
  - Depth of tree = $k = \log_b n$

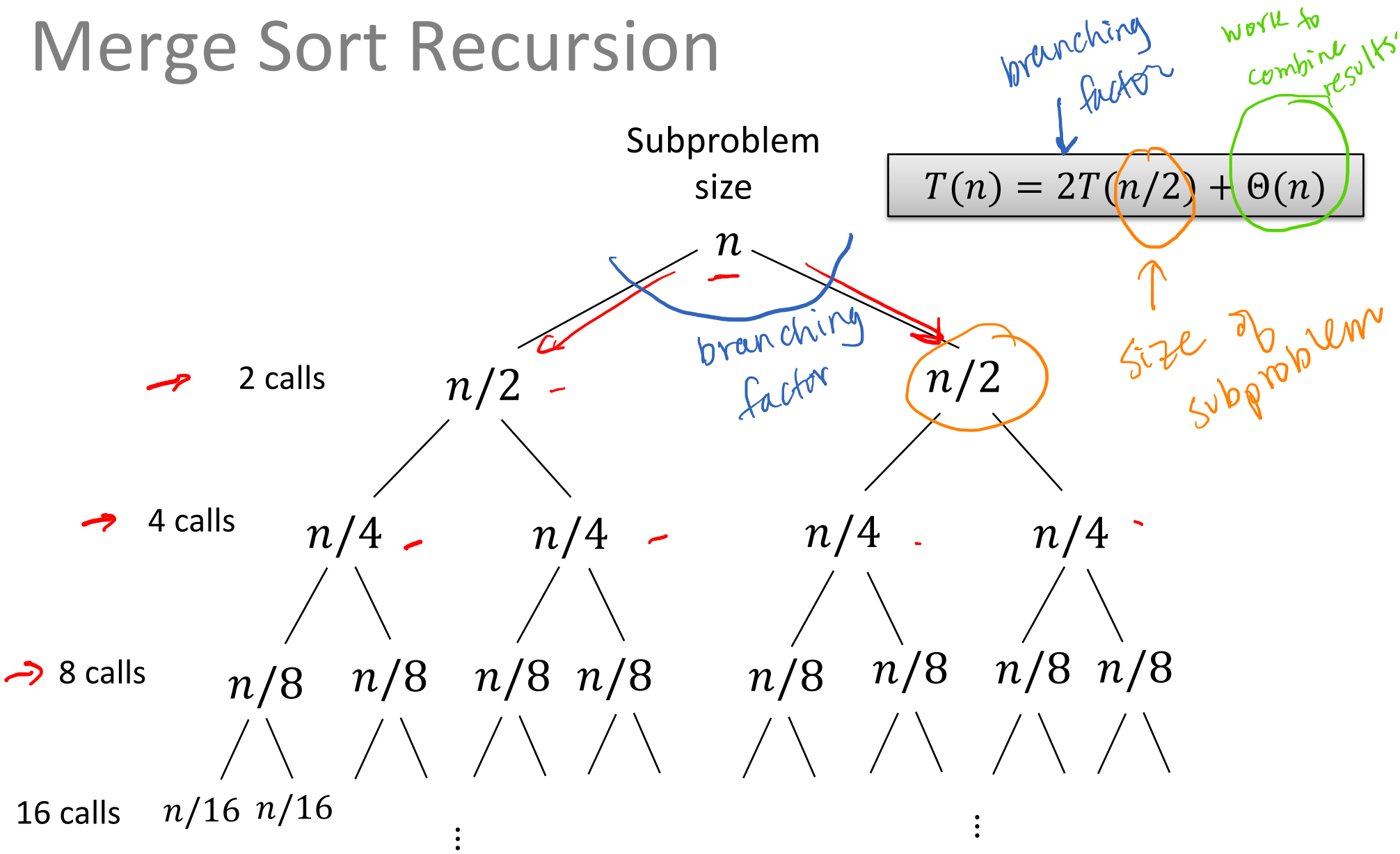
Outline

- Complexity Recap
- Runtimes of algorithms (with pseudocode)

- Divide and Conquer recurrences
  - Example: Merge Sort

- **Master Theorem**
  - **Key features of a Divide and Conquer algorithm**
  - **Master Theorem: runtimes for divide and conquer algorithms**
  - (Optional) Derive Master Theorem

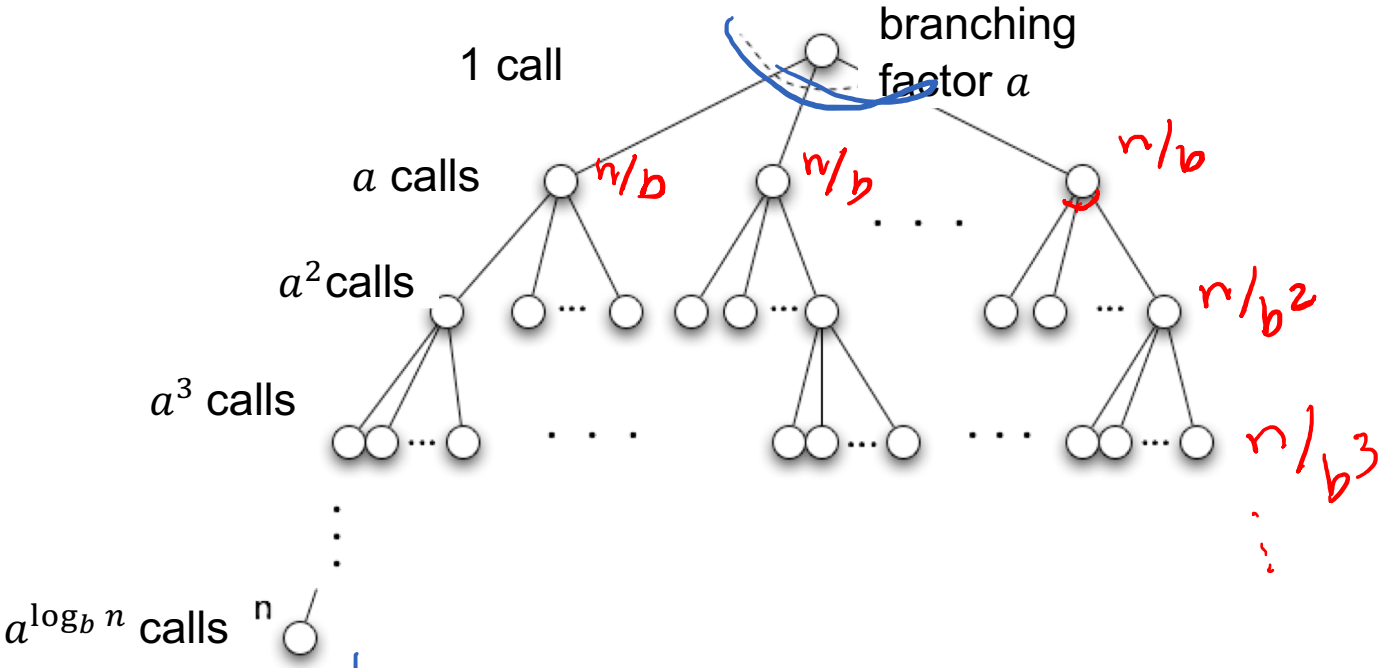
Merge Sort Recursion



Key features:

- Branching factor = 2
- Size of subproblems = n/2
- Work to combine results = $\Theta(n)$

Divide-and-Conquer Recursion

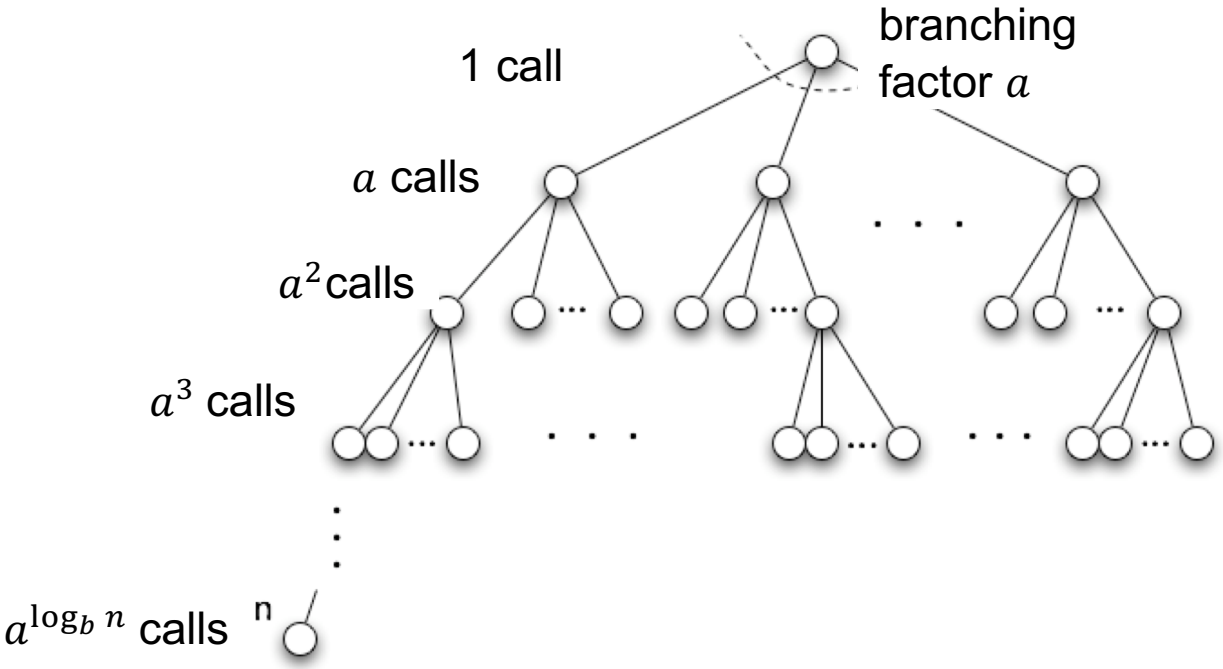


$$T(n) = aT(n/b) + \Theta(n^d)$$

Key features:

- Number of subproblems : a
- Size of subproblems : n/b
- Work needed to combine results : $\Theta(n^d)$

Divide-and-Conquer Recursion



$$\boldsymbol{T(n) = aT(n/b) + \Theta(n^d)}$$

Key features:

- Number of subproblems: a
- Size of subproblems: n/b
- Work needed to combine results: $\Theta(n^d)$

Divide-and-Conquer Runtime

The Master Theorem: If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } (a/b^d) < 1 \\ \Theta(n^d \log n) & \text{if } (a/b^d) = 1 \\ \Theta(n^{\log_b a}) & \text{if } (a/b^d) > 1 \end{cases}$$

Note:
Rosen uses a slightly different version of the Master Theorem, with

- $T(n) = aT(n/b) + \boldsymbol{cn^d}$
- Big-O instead of Big-Θ for the 3 cases of runtimes for $T(n)$
  - Ex: $T(n) = \boldsymbol{O}(n^d)$ if $(a/b^d) < 1$

Master Theorem and MergeSort

The Master Theorem: If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } (a/b^d) < 1 \\ \Theta(n^d \log n) & \text{if } (a/b^d) = 1 \\ \Theta(n^{\log_b a}) & \text{if } (a/b^d) > 1 \end{cases}$$

- The recurrence for MergeSort:
  - $T(n) = 2T(n/2) + \Theta(n)$ and $T(1) = 1$.

Master Theorem and MergeSort

The Master Theorem: If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } (a/b^d) < 1 \\ \Theta(n^d \log n) & \text{if } (a/b^d) = 1 \\ \Theta(n^{\log_b a}) & \text{if } (a/b^d) > 1 \end{cases}$$

- The recurrence for MergeSort:
  - $T(n) = 2T(n/2) + \Theta(n)$ and $T(1) = 1$.
  - $T(n) = aT(n/b) + \Theta(n^d)$
- So $a = 2,\ b = 2,\ d = 1$. Therefore $a/b^d = 1$.
- The Master Theorem tells us that
  - $T(n)$ is in $\Theta(n^d \log n) = \Theta(n \log n)$.

Master Theorem Practice

The Master Theorem: If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } (a/b^d) < 1 \\ \Theta(n^d \log n) & \text{if } (a/b^d) = 1 \\ \Theta(n^{\log_b a}) & \text{if } (a/b^d) > 1 \end{cases}$$

Recurrence	Big-Θ of F(n)
1. $F(n) = 2F(n/4) + \sqrt{n}$	$\Theta(n^d \log n) = \Theta(\sqrt{n} \log n)$
2. $F(n) = 8F(n/2) + 5n^2$	$\Theta(n^{\log_b a}) = \Theta(n^{\log_2 8}) = \Theta(n^3)$
3. $F(n) = 4F(n/2) + 2n^3$	$\Theta(n^3)$
4. $F(n) = 2F(n/2) + 2$	$\Theta(n)$

1. $a = 2,\ b = 4,\ d = 1/2$; $a/b^d = 2/\sqrt{4} = 1$
2. $a = 8,\ b = 2,\ d = 2$; $\frac{a}{b^d} = \frac{8}{2^2} > 1$

Master Theorem Practice

The Master Theorem: If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } (a/b^d) < 1 \\ \Theta(n^d \log n) & \text{if } (a/b^d) = 1 \\ \Theta(n^{\log_b a}) & \text{if } (a/b^d) > 1 \end{cases}$$

	Recurrence	**Big-Θ of F(n)**
1.	$F(n) = 2F(n/4) + \sqrt{n}$	$\Theta(\sqrt{n}\log(n))$
2.	$F(n) = 8F(n/2) + 5n^2$	$\Theta(n^3)$
3.	$F(n) = 4F(n/2) + 2n^3$	$\Theta(n^3)$
4.	$F(n) = 2F(n/2) + 2$	$\Theta(n)$

Master Theorem with Pseudocode

Find the Big-Θ runtime of the following algorithm:

```
procedure farewell(n: integer)
    if n=0, stop
    farewell(n/3)
    farewell(n/3)
    farewell(n/3)
    farewell(n/3)

    for i := 1 to n/4
        for j := 1 to n
            print "good luck with finals!"
```

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

4 subproblems

$a = 4,\ b = 3,\ d = 2$

master thm $\Rightarrow \Theta(n^2)$

$n/4 \equiv n$

n

$\Rightarrow \Theta(n^2)$

Master Theorem with Pseudocode

Find the Big-Θ runtime of the following algorithm:

```
procedure farewell(n: integer)
    if n=0, stop
    farewell(n/3)
    farewell(n/3)
    farewell(n/3)
    farewell(n/3)

    for i := 1 to n/4
        for j := 1 to n
            print "good luck with finals!"
```

4 subcalls of size n/3

n

$n/4 \equiv n$

$\Longrightarrow \Theta(n^2)$

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

- Master Theorem with: $a = 4, b = 3, d = 2$
- $(a/b^d) = 4/3^2 = 4/9 < 1$
- $T(n) = \Theta(n^d) = \Theta(n^2)$

That's a wrap on EECS 203

- We know you've learned a lot
- We hope you've enjoyed yourself too!

- Good luck on finals

- Have a great summer!

Outline

- Complexity Recap
- Runtimes of algorithms (with pseudocode)

- Divide and Conquer recurrences
  - Example: Merge Sort

- Master Theorem
  - Key features of a Divide and Conquer algorithm
  - Master Theorem: runtimes for divide and conquer algorithms
  - **(Optional) Derive Master Theorem**

Analyzing complexity: Example 1

- $T(n) = 3T(n/3) + n^2$
- Find O(T(n))

Analyzing complexity: Example 1

- $T(n) = 3T(n/3) + n^2$
- Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0				
1				
2				
3				
…				

Analyzing complexity: Example 1

- $T(n) = 3T(n/3) + n^2$
- Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0	1	n	n^2	n^2
1	3	$n/3$	$\left(\frac{n}{3}\right)^2$	$n^2/3$
2	9	$n/9$	$\left(\frac{n}{9}\right)^2$	$n^2/9$
3	27	$n/27$	$\left(\frac{n}{27}\right)^2$	$n^2/27$
…				

$$T(n) = n^2\left(1 + \tfrac{1}{3} + \tfrac{1}{9} + \tfrac{1}{27} + \cdots + \{last\ term\}\right)$$

Geometric sum with $r = 1/3$

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

$$\boldsymbol{T(n) = O(n^2)}$$

Q: Could I do less work than n^2?

A: No. $n^2 \le T(n) < \frac{3}{2}n^2$

So, $T(n) = \theta(n^2)$

Analyzing complexity: Example 2

- $T(n) = 5T(n/4) + n.$ Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0				
1				
2				
3				
⋮				
k				
⋮				

Analyzing complexity: Example 2

- $T(n) = 5T(n/4) + n.$ Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0	1	n	n	n
1	5	$n/4$	$n/4$	$5n/4$
2	25	$n/16$	$n/16$	$25n/16$
3	125	$n/64$	$n/64$	$n \cdot 5^3/4^3$
⋮				
k	5^k	$n/4^k$	$n/4^k$	$n(5/4)^k$
⋮				

$$T(n) = n(1 + \tfrac{5}{4} + \left(\tfrac{5}{4}\right)^2 + \left(\tfrac{5}{4}\right)^3 + \cdots)$$

Geometric sum with $|r| > 1$.
Infinite sum doesn't converge!

Thankfully, this sum is finite.
But when does it end?

Analyzing complexity: Example 2

- $T(n) = 5T(n/4) + n.$ Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0	1	n	n	n
1	5	$n/4$	$n/4$	$5n/4$
2	25	$n/16$	$n/16$	$25n/16$
3	125	$n/64$	$n/64$	$n \cdot 5^3/4^3$
⋮				
k	5^k	$n/4^k$	$n/4^k$	$n(5/4)^k$
⋮				
x		**1**		

$$1 = \frac{n}{4^x}$$
$$4^x = n$$
$$x = \log_4 n$$

Analyzing complexity: Example 2

- $T(n) = 5T(n/4) + n.$ Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0	1	n	n	n
1	5	$n/4$	$n/4$	$5n/4$
2	25	$n/16$	$n/16$	$25n/16$
3	125	$n/64$	$n/64$	$n \cdot 5^3/4^3$
⋮				
k	5^k	$n/4^k$	$n/4^k$	$n(5/4)^k$
⋮				
$x = \log_4 n$	$5^{\log_4 n}$	1	1	$n(5/4)^{\log_4 n} = 5^{\log_4 n}$

Analyzing complexity: Example 2

- $T(n) = 5T(n/4) + n.$ Find O(T(n))

Depth	# of calls	Size of calls	Work/call	Total work/depth
0	1	n	n	n
k	5^k	$n/4^k$	$n/4^k$	$n(5/4)^k$
$x = \log_4 n$	$5^{\log_4 n}$	$1 = \frac{n}{4^x}$	1	$n(5/4)^{\log_4 n} = 5^{\log_4 n}$

$$T(n) = n\left[1 + \tfrac{5}{4} + \left(\tfrac{5}{4}\right)^2 + \left(\tfrac{5}{4}\right)^3 + \cdots + \left(\tfrac{5}{4}\right)^x\right]$$ (Geom. Sum, $r = 5/4$)

$$= n\left[\left(\tfrac{5}{4}\right)^x + \left(\tfrac{5}{4}\right)^{x-1} + \cdots + \left(\tfrac{5}{4}\right)^3 + \left(\tfrac{5}{4}\right)^2 + \tfrac{5}{4} + 1\right]$$ Trick: reverse the order of the terms

$$< n\left(\tfrac{5}{4}\right)^x\left[1 + \tfrac{4}{5} + \left(\tfrac{4}{5}\right)^2 + \left(\tfrac{4}{5}\right)^3 + \cdots + \left(\tfrac{4}{5}\right)^{9999} + \cdots\right]$$ (Geom. Sum, $r = 4/5$)

$$= n\left(\tfrac{5}{4}\right)^x \tfrac{1}{1-4/5}$$

$$= 5n\left(\tfrac{5}{4}\right)^x$$

$$= 5 \cdot 5^{\log_4 n} \quad (x = \log_4 n)$$

$$= 5 \cdot n^{\log_4 5} \quad (z^{\log_b y} = y^{\log_b z})$$

$T(n)$ is $O(n^{\log_4 5})$

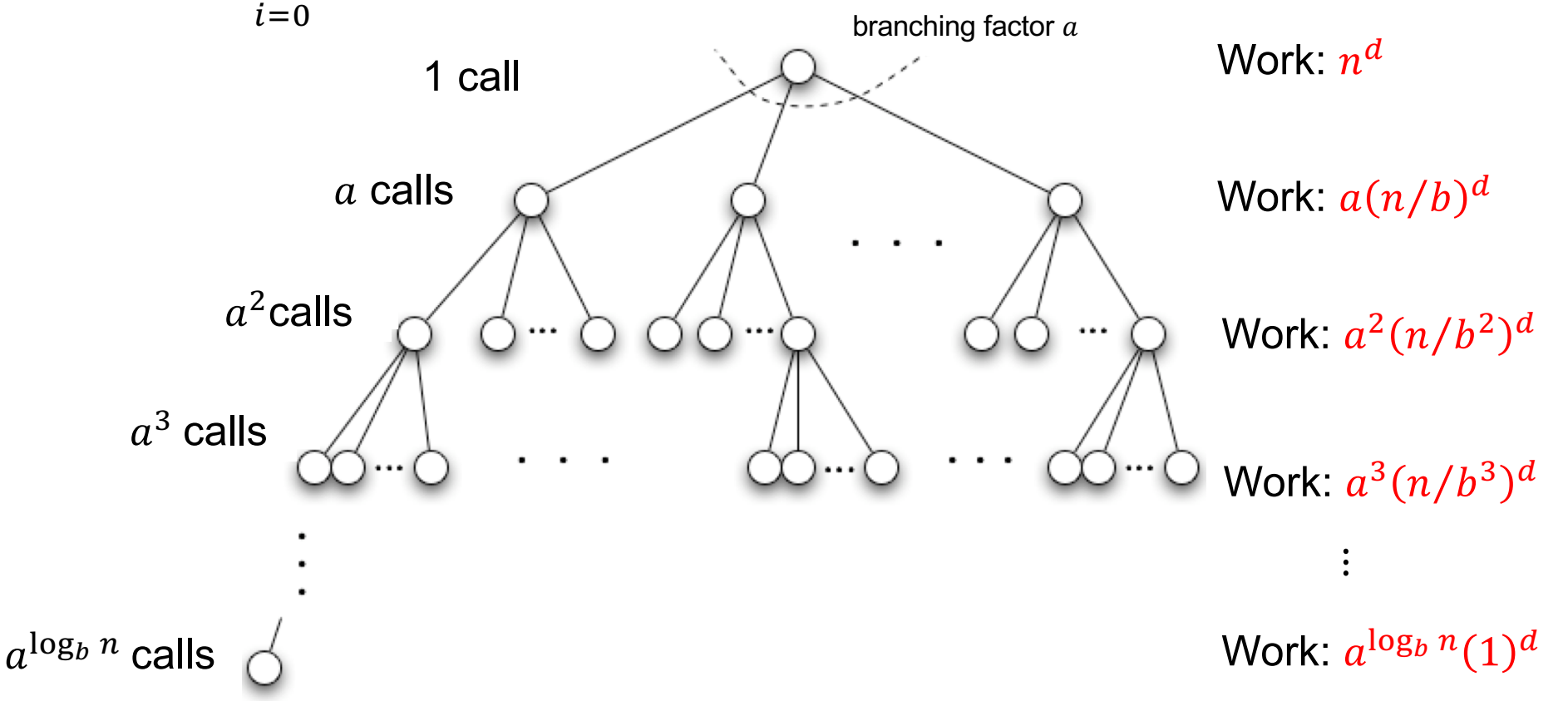
Also $\Theta(n^{\log_4 5})$

Closed Form for $T(n)$

- $T(n) = aT(n/b) + n^d$ (recursive form)

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}(1)^d \quad \text{(closed form)}$$

$$= \sum_{i=0}^{\log_b n} a^i \left(\frac{n}{b^i}\right)^d T(n)$$



Estimate $\Theta(T(n))$

- $T(n) = aT(n/b) + n^d$ (recursive form)

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}(1)^d$$

$$= n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^d$$

$$= n^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$

geometric sum with base (a/b^d)

- Simplification depends on whether the base is > 0, < 0, or $= 0$

Estimate $\Theta(T(n))$

- $T(n) = aT(n/b) + n^d$ (recursive form)

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}(1)^d$$

$$= n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^d$$

$$= n^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$

geometric sum with base (a/b^d)

- If $\boldsymbol{a/b^d < 1}$
  - the terms in the sum are getting smaller, so the **first** term dominates the sum
  - $\Theta(T(n)) = \Theta(n^d(1)) = \Theta(n^d)$

Estimate $\Theta(T(n))$

- $T(n) = aT(n/b) + n^d$ (recursive form)

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}(1)^d$$

$$= n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^d$$

$$= n^d\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$

geometric sum with base (a/b^d)

- If $\boldsymbol{a/b^d > 1}$
  - the terms in the sum are getting larger, so the **last** term dominates the sum
  - $\Theta(T(n)) = \Theta\left(n^d\left(\frac{a}{b^d}\right)^{\log_b n}\right) = \ \ldots\ = \Theta\left(n^{\log_b a}\right)$

Estimate $\Theta(T(n))$

- $T(n) = aT(n/b) + n^d$ (recursive form)

$$T(n) = n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}(1)^d$$

$$= n^d + a\left(\frac{n}{b}\right)^d + a^2\left(\frac{n}{b^2}\right)^d + \cdots + a^{\log_b n}\left(\frac{n}{b^{\log_b n}}\right)^d$$

$$= n^d \underbrace{\left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)}$$

geometric sum with base (a/b^d)

- If $\boldsymbol{a/b^d = 1}$
  - the terms in the sum are all 1, so the **number of terms** impacts $\Theta(T(n))$
  - $\Theta(T(n)) = \Theta(n^d(1 + 1 + 1 + \cdots + 1)) = \Theta\left(n^d \log_b n\right)$

The Master Theorem

- $T(n) = aT(n/b) + n^d$ (recursive form)

$$T(n) = n^d \left(1 + \frac{a}{b^d} + \left(\frac{a}{b^d}\right)^2 + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right)$$

geometric sum with base (a/b^d)

- If $a/b^d < 1$ the **first** term dominates $\Theta(T(n))$.
- If $a/b^d > 1$ the **last** term dominates $\Theta(T(n))$.
- If $a/b^d = 1$ the **number** of terms impacts $\Theta(T(n))$.

The Master Theorem: If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } (a/b^d) < 1 \\ \Theta(n^d \log n) & \text{if } (a/b^d) = 1 \\ \Theta(n^{\log_b a}) & \text{if } (a/b^d) > 1 \end{cases}$$