Questions of the day

- How can we use our understanding of Big-Oh to show that the algorithm below to the left always takes $\Theta(n^2)$ time?
- What about a recursive algorithm, like Binary Search?

Input: *data*: an array of integers to sort
Input: n: the number of values in *data*
Output: permutation of *data* such that $data[1] \leq \ldots \leq data[n]$

for $i = 1$ to $n - 1$ **do**
  Let m be the location of the min value in $data[i..n]$;
  Swap $data[i]$ and $data[n]$;
end
return *data*;

Input: *data*: array of n integer
Input: n: size of *data*
Input: t: target value to search for
Output: index i such that $data[i] = t$, or 0 if $t \notin data$
Algorithm: BinSearch
if $n = 1$ **then**
  return whether $data[1] = t$
else
  $mid = \lceil n/2 \rceil$
  if $data[mid] = t$ **then**
    return *mid*
  else if $data[mid] > t$ **then**
    return BinSearch($data[1..mid]$, t)
  else
    return BinSearch($data[mid + 1..n]$, t)
  end
end

Analysis

William Hendrix

Lecture 2

Today

- Review
  - Big-Oh notation
  - Big-Oh properties
- Analyzing algorithms
- Summation identities
- Recursive analysis

Review: Big-Oh formal definitions

- Provides a useful way to classify functions according to growth rate
  - Focuses on asymptotic (eventual) growth
- O(*g(n)*): grow no faster than *g(n)*
- Ω(*g(n)*): grow no slower than *g(n)*
- Θ(*g(n)*): grow at the same rate as *g(n)*

$f(n) = O(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

$f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

$f(n) = \Theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

- *To prove:* find positive constants that satisfy definition

Review: Big-Oh properties

- **Reflexive**
  - All functions are O/Ω/Θ of themselves
- **Symmetric (Θ only)**
  - If $f(n) = \Theta(g(n))$, $g(n) = \Theta(f(n))$
- **Antisymmetry (O and Ω)**
  - If f is O of g, g is Ω of f and vice versa
  - If f is both O and Ω of g, they are Θ of one another
- **Transitive**
  - If f is $O(g(n))$ and g is $O(h(n))$, f is $O(h(n))$
  - Same for Ω and Θ
- **"Envelopment"**
  - Sums and products of functions or bounds can be combined or split
- **Constant coefficients**
  - Multiplying by a constant doesn't change a function's growth rate
- **Largest term**
  - The dominant term in a sum determines the growth rate

Back to our previous example…

Input: $data$: array of integers
Input: n: size of $data$
Output: index min such that $data[min] \leq data[j]$, for all j from 1 to n

Algorithm: FindMin

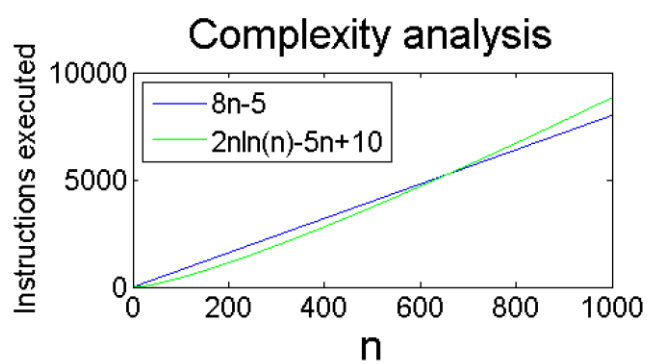
$min = 1$
for $i = 2$ to n **do**
    if $data[i] < data[min]$ **then**
        $min = i$
    end
end
return min

Ops per line … Times executed …

Total ops: $\leq 7(n-1) + 2 = 7n - 5$

Question: is this better or worse than an algorithm that takes at most $2n \ln n - 5n + 10$ ops?

Better unless n < 395



Worst case: *O(n)*

Don't need to count instructions!

Other algorithm: *O(n*lg*(n))*

Conclusion: Our algorithm is better for sufficiently large *n*.

Algorithm analysis

- Identify loops and function calls
  - Everything else is $\Theta(1)$
- *For loops*:
  - Estimate number of iterations
  - Estimate loop body running time
  - Does loop run time depend on iteration #?
    - If not: total = # iterations * time per iteration
    - Otherwise: total = sum of all iterations
- *For functions:*
  - Non-recursive: analyze separately
  - Recursive: set up a recurrence and solve (after exam)
- Overall complexity: sum of complexities
  - Largest loop/function call

Summations

- Summations show up frequently in analysis

- Notation
  - "Dotty notation": $1 + 2 + 3 + \ldots + n$
    - Easy to understand
  - Compact notation: $\sum_{i=1}^{n} i$
    - Easy to write

- Useful identities

$$\sum_{i=1}^{x} i = \Theta(x^2) \qquad \sum_{i=1}^{x} 2^i = \Theta(2^x)$$

$$\sum_{i=1}^{x} \frac{1}{i} = \Theta(\lg x) \qquad \sum_{i=1}^{x} \frac{1}{2^i} = \Theta(1)$$

Loop analysis example

- What is the worst-case time complexity for the following algorithm?
  - What is the best-case?

```
Input: data: array of integers
Input: n: length of data
Output: permutation of data such that
        data[1] ≤ data[2] ≤ ... ≤ data[n]
Algorithm: SelectionSort
for i = 1 to n − 1 do
    min = i
    for j = i + 1 to n do
        if data[j] < data[min] then
            min = j
        end
    end
    Swap data[i] and data[min]
end
return data
```

Loop analysis example

- SelectionSort is $\Theta(n^2)$.

Proof. Note that the **for** loop in lines 2–10 will iterate $n-1 = \Theta(n)$ times. Line 3 takes $\Theta(1)$ time, and line 9 takes $\Theta(1)$ (three assignment statements). The **for** loop in lines 4–8 iterates $n-i$ times. Lines 5 and 6 are both $\Theta(1)$, so each iteration of the **for** loop in lines 4–8 takes $\Theta(1)$ time. Since each iteration takes $\Theta(1)$ time and the loop iterates $n-i$ times, this **for** loop takes $\Theta(n-i)$ time. Hence, each iteration of the outer **for** loop takes $\Theta(1) + \Theta(n-i) + \Theta(1) = \Theta(n-i)$ time. Since the length of an iteration of the outer **for** loop depends on the iteration number, the total time will be $\sum_{i=1}^{n-1} \Theta(n-i) = \Theta\left(\sum_{i=1}^{n-1} i\right) = \Theta((n-1) + (n-2) + \ldots + 1)$ time. This sum equals $\Theta(\sum_{i=1}^{n-1} i)$, which is $\Theta(n^2)$. The **return** statement in line 11 takes $\Theta(1)$ time, so the total time for SelectionSort is $\Theta(n^2) + \Theta(1) = \Theta(n^2)$. $\square$

Analysis further practice

1. What is the *worst case* time complexity for the algorithm below? Show your work.
   - *Best case?*

Input: $data$: an array of integers to sort
Input: n: the number of values in $data$
Output: permutation of $data$ such that $data[1] \leq \ldots \leq data[n]$

Algorithm: InsertionSort

for $i = 2$ to n **do**
    $ins = data[i]$
    $j = i$
    while $j > 1$ and $data[j-1] > ins$ **do**
        $data[j] = data[j-1]$
        $j = j - 1$
    end
    $data[j] = ins$
end
return $data$

Analysis further practice

Input: *data*: an array of integers to sort
Input: n: the number of values in *data*
Output: permutation of *data* such that
$$data[1] \leq \ldots \leq data[n]$$

Algorithm: InsertionSort

$\Theta(n)$ iter. — **for** $i = 2$ to n **do**

$\Theta(1)$ —
  $ins = data[i]$
  $j = i$

  while $j > 1$ and $data[j-1] > ins$ **do**
    $data[j] = data[j-1]$
    $j = j - 1$
  end

$\Theta(1)$ —
  $data[j] = ins$

end

return *data*

Analysis further practice

Input: *data*: an array of integers to sort
Input: n: the number of values in *data*
Output: permutation of *data* such that
$data[1] \le \ldots \le data[n]$

Algorithm: InsertionSort

$\Theta(n)$ iter.	**for** $i = 2$ to n **do**
$\Theta(1)$	$ins = data[i]$
	$j = i$
$O(n), \Omega(1)$ iter.	**while** $j > 1$ and $data[j-1] > ins$ **do**
$\Theta(1)$	$data[j] = data[j-1]$
	$j = j - 1$
	end
$\Theta(1)$	$data[j] = ins$
	end
	return *data*

Analysis further practice

Input: *data*: an array of integers to sort
Input: n: the number of values in *data*
Output: permutation of *data* such that
$data[1] \leq \ldots \leq data[n]$

Algorithm: InsertionSort

$\Theta(n)$ iter.
for $i = 2$ to n **do**

$\Theta(1)$
  $ins = data[i]$
  $j = i$

$O(n), \Omega(1)$
  while $j > 1$ and $data[j-1] > ins$ **do**
    $data[j] = data[j-1]$
    $j = j - 1$
  end

$\Theta(1)$
  $data[j] = ins$
end
return *data*

Analysis further practice

Input: *data*: an array of integers to sort
Input: n: the number of values in *data*
Output: permutation of *data* such that
$data[1] \leq \ldots \leq data[n]$

```
Algorithm: InsertionSort
for i = 2 to n do
    ins = data[i]
    j = i
    while j > 1 and data[j − 1] > ins do
        data[j] = data[j − 1]
        j = j − 1
    end
    data[j] = ins
end
return data
```

$\Theta(n)(O(n), \Omega(1))$
$= O(n^2), \Omega(n)$

$\Theta(1)$

Analysis further practice

Input: $data$: an array of integers to sort
Input: n: the number of values in $data$
Output: permutation of $data$ such that
$data[1] \leq \ldots \leq data[n]$

Algorithm: InsertionSort

$O(n^2), \Omega(n)$

for $i = 2$ to n **do**
  $ins = data[i]$
  $j = i$
  while $j > 1$ and $data[j-1] > ins$ **do**
    $data[j] = data[j-1]$
    $j = j - 1$
  end
  $data[j] = ins$
end
return $data$

Tricky example

- Algorithm analysis is **not** simply counting the number of loops

Input: *data*: array of integers
Input: n: length of *data*
Output: permutation of *data* such that
$data[1] \leq data[2] \leq \ldots \leq data[n]$

Algorithm: BadSort

for $i = n - 1$ to 1 step -1 **do**
  for $j = 1$ to $n - i$ step i **do**
    if $data[j] > data[j + i]$ **then**
      Swap $data[j]$ and $data[j + i]$
    end
  end
end
return *data*

Tricky example

- Algorithm analysis is **not** simply counting the number of loops

Input: *data*: array of integers
Input: n: length of *data*
Output: permutation of *data* such that
$data[1] \le data[2] \le \ldots \le data[n]$

Algorithm: BadSort

for $i = n - 1$ to 1 step -1 **do**
    for $j = 1$ to $n - i$ step i **do**
        if $data[j] > data[j+i]$ **then**
            Swap $data[j]$ and $data[j+i]$
        end
    end
end
return *data*

$\frac{n-i}{i}$ iter. $= \Theta(\frac{n}{i})$

Tricky example

- Algorithm analysis is **not** simply counting the number of loops

Input: $data$: array of integers
Input: n: length of $data$
Output: permutation of $data$ such that $data[1] \leq data[2] \leq \ldots \leq data[n]$

Algorithm: BadSort

for $i = n - 1$ to 1 step -1 **do**
  for $j = 1$ to $n - i$ step i **do** — $\Theta(n/i)$ iter.
    if $data[j] > data[j+i]$ **then** — $\Theta(1)$
      Swap $data[j]$ and $data[j+i]$
    end
  end
end
return $data$

Tricky example

- Algorithm analysis is **not** simply counting the number of loops

$\Theta(n)$ iter.

$\Theta(n/i)$

Input: *data*: array of integers
Input: n: length of *data*
Output: permutation of *data* such that
$data[1] \leq data[2] \leq \ldots \leq data[n]$

Algorithm: BadSort
for $i = n - 1$ to 1 step -1 **do**
  for $j = 1$ to $n - i$ step i **do**
    if $data[j] > data[j + i]$ **then**
      Swap $data[j]$ and $data[j + i]$
    end
  end
end
return *data*

Tricky example

- Algorithm analysis is **not** simply counting the number of loops

Input: $data$: array of integers
Input: n: length of $data$
Output: permutation of $data$ such that
$data[1] \leq data[2] \leq \ldots \leq data[n]$

Algorithm: BadSort

for $i = n - 1$ to 1 step -1 **do**
    for $j = 1$ to $n - i$ step i **do**
        if $data[j] > data[j + i]$ **then**
            Swap $data[j]$ and $data[j + i]$
        end
    end
end
return $data$

$$\sum_{i=1}^{n-1} \Theta(n/i)$$

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

$$= \Theta(n \lg n)$$

Recursive analysis

- **Challenge:** recursive functions call themselves repeatedly (down to base case)
- **Main idea:** use recurrence to define runtime recursively
- **Example:** factorial
  - *T(n)*: time to compute *n*!

```
Input: n: number to calculate factorial
Output: n!
Algorithm: Factorial
if n = 0 then
|   return 1
else
|   temp = Factorial(n − 1)
|   return n ∗ temp
end
```

Recursive analysis

- **Challenge:** recursive functions call themselves repeatedly (down to base case)
- **Main idea:** use recurrence to define runtime recursively
  - Solve recurrence
- **Example:** factorial
  - *T(n)*: time to compute *n*!

Input: n: number to calculate factorial
Output: $n!$
Algorithm: Factorial

$\Theta(1)$
if $n = 0$ **then**
  return 1
else
$T(n-1)$
  $temp$ = Factorial$(n-1)$
$\Theta(1)$
  return $n * temp$
end

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 0 \\ T(n-1) + \Theta(1), & \text{otherwise} \end{cases}$$

Base case not important for asymptotic complexity

$$T(n) = T(n-1) + \Theta(1)$$

Recursion tree analysis

- Technique to analyze algorithm complexity of recursive algorithms
- **Main idea**
  - Sketch a tree that represents all recursive calls made
    - Start with n, split according to size and # of recursive calls
  - Add up complexity on each level of tree
    - Apply nonrecursive complexity to every node
    - Often helpful to estimate tree height
  - Add up the total complexity of all the nodes
- **Example**
  - $T(n) = T(n-1) + \Theta(n)$

Recursion tree analysis

- Technique to analyze algorithm complexity of recursive algorithms
- **Main idea**
  - Sketch a tree that represents all recursive calls made
    - Start with *n*, split according to size and # of recursive calls
  - Add up complexity on each level of tree
    - Apply nonrecursive complexity to every node
    - Often helpful to estimate tree height
  - Add up the total complexity of all the nodes
- **Example**
  - $T(n) = T(n\text{-}1) + \Theta(n)$

Recursive InsertionSort

		<u>Complexity per level</u>
$h = n$	n	$\Theta(n)$
	$n-1$	$\Theta(n-1)$
	$n-2$	$\Theta(n-2)$
	$\ddots$	$\vdots$
	1	$\Theta(1)$

Overall: $$\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

Recursion tree example

- Identify the complexity class for T(n) when

$$T(n) = 2T(n-1) + \Theta(1),\ T(1) = \Theta(1)$$

2 rec. calls of size *n*-1

Everything else in function

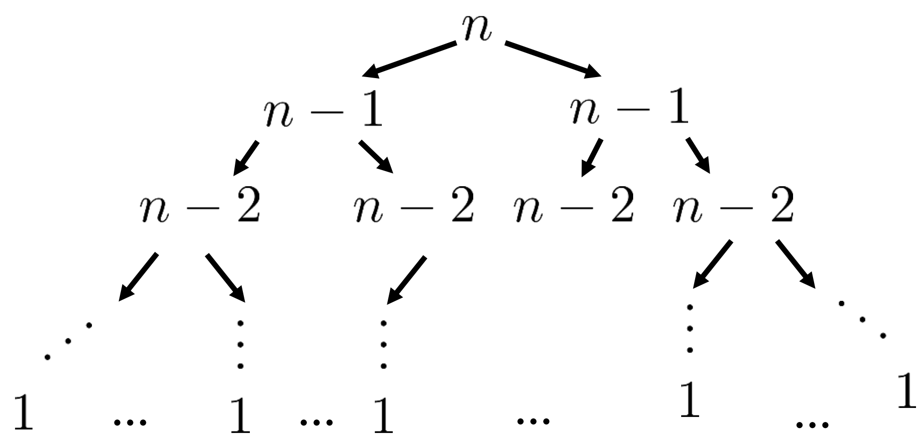
Recursion tree example

- Identify the complexity class for T(n) when

$$T(n) = 2T(n-1) + \Theta(1),\ T(1) = \Theta(1)$$

2 rec. calls of size *n*-1   Everything else in function

<u>Tree</u>



<u>Complexity per level</u>

$\Theta(1)$

$2\Theta(1)$

$4\Theta(1)$

$\vdots$

$2^n\Theta(1)$

<u>Total complexity:</u> $\sum_{i=1}^{n} \Theta(2^i)$

$= \Theta(2^n)$

Recursive analysis exercise

```
Input: data: array of n integer
Input: n: size of data
Input: t: target value to search for
Output: index i such that data[i] = t, or 0 if t ∉ data
Algorithm: BinSearch
if n = 1 then
|  return whether data[1] = t
else
|  mid = ⌈n/2⌉
|  if data[mid] = t then
|  |  return mid
|  else if data[mid] > t then
|  |  return BinSearch(data[1..mid], t)
|  else
|  |  return BinSearch(data[mid + 1..n], t)
|  end
end
```

1. Give a recurrence that describes the worst-case time complexity of BinSearch. (*Hint*: how many recursive calls will you make?)
2. Draw a recursion tree for your recurrence.
3. Solve the recurrence.

Recursive analysis exercise

Input: $data$: array of n integer
Input: n: size of $data$
Input: t: target value to search for
Output: index i such that $data[i] = t$, or 0 if $t \notin data$
Algorithm: BinSearch

if $n = 1$ **then**
    return whether $data[1] = t$
else
    $mid = \lceil n/2 \rceil$
    if $data[mid] = t$ **then**
        return mid
    else if $data[mid] > t$ **then**
        return BinSearch($data[1..mid]$, t)
    else
        return BinSearch($data[mid + 1..n]$, t)
    end
end

Recurrence: T(n) = T(n/2) + O(1)

Tree (above)
Total complexity:

Recursive analysis exercise

Every line except the recursive calls in lines 9 and 11 take $\Theta(1)$. The arrays in lines 9 and 11 are $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$, respectively, or roughly $n/2$ each. Thus, the recursive calls will take $T(n/2)$ time. However, due to the structure of the code, only *one* of the two calls can execute, for a total complexity of $T(n) = T(n/2) + \Theta(1)$.

$$
\begin{array}{cc}
\underline{\text{Tree}} & \underline{\text{Complexity}} \\
\begin{array}{r} h = \\ \lg n \end{array}
\left[\begin{array}{c} n \\ \downarrow \\ n/2 \\ \downarrow \\ n/4 \\ \downarrow \\ \vdots \\ \downarrow \\ 1 \end{array} \right. &
\begin{array}{c} \Theta(1) \\ \\ \Theta(1) \\ \\ \Theta(1) \\ \\ \vdots \\ \\ \underline{\Theta(1)} \\ \Theta(\lg n) \end{array}
\end{array}
$$

The Master Theorem

- Many recursive algorithms have complexity of the form:

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

  – n/b : size of recursive calls

  – a : number of recursive calls (often $a = b$)

  – $f(n)$: time required for other code

- **Master Theorem**

  – Gives complexity for $T(n)$ based on a, b, and $f(n)$

  1. Calculate $c = log_b(a)$
  2. Compare complexity of $f(n)$ to n^c
     - If $f(n) = \Theta(n^c)$, $T(n) = \Theta(f(n) \lg n)$
     - Otherwise, if $f(n)$ is *strictly smaller* than $O(n^c)$, $T(n) = \Theta(n^c)$
       – $f(n) = O(n^{c-e})$, for some $e > 0$
     - Otherwise, if $f(n) = \Omega(n^{c+e})$ <u>and</u> f is *regular*, $T(n) = \Theta(f(n))$
       – <u>Strictly more</u> than n^c
       – Regular: $af(n/b) < f(n)$, for large n

Formal statement of Master Theorem

Master Theorem. If T is an increasing function that satisfies the recurrence

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

where $a \geq 1$ and $b \geq 1$, then:

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \quad \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases},$$

where $c = \log_b(a)$.

Almost: $T(n) = \Theta(n^c + f(n))$ unless n^c and $f(n)$ are same size

For purposes of the Master Theorem, you may ignore floor and ceiling

$$\text{E.g., } T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n)$$
$$= 2T(n/2) + \Theta(n)$$

Warning: cases 1+3 must be polynomially different (<u>not log</u>)

Master Theorem application

Master Theorem. If $T(n) = aT(n/b) + f(n)$,

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases}$$

- $T(n) = T(n/2) + \Theta(1)$

1. Identify variables
2. Calculate *c*
3. Decide case
4. Report complexity (test regularity if case 3)

Master Theorem application

Master Theorem. If $T(n) = aT(n/b) + f(n)$,

$$T(n) = \begin{cases} \Theta(n^c), & \text{if } f(n) = O(n^{c-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^c \lg n), & \text{if } f(n) = \Theta(n^c) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{c+\epsilon}) \text{ for some } \epsilon > 0 \\ & \underline{\text{and}}\ af(n/b) < f(n) \text{ for large } n \end{cases}$$

- $T(n) = T(n/2) + \Theta(1)$

1. Identify variables
   - $a = 1, b = 2, f(n) = \Theta(1)$
2. Calculate c
   - $c = \log_b(a) = \log_2(1) = 0$
3. Decide case
   - n^c vs. $f(n)$? $f(n) = \Theta(n^c)$
4. Report complexity (test regularity if case 3)
   - $\Theta(n^c \lg n) = \boxed{\Theta(\lg n)}$

Master Theorem exercises

- What is *c* for the following recurrences?
- What case does *f(n)* fall under?
- What is the asymptotic complexity for the following recurrences? Write "n/a" if the Master Theorem does not apply.

1. $T(n) = 2T(n/2) + \Theta(1)$
2. $U(n) = 4U(n/2) + \Theta(n^2)$
3. $V(n) = 9V(n/9) + \Theta(n\lg n)$
4. $W(n) = 3W(n/3) + \Theta(\lg n)$
5. $X(n) = 2X(n/4) + \Theta(n\lg n)$
6. $Y(n) = 3Y(n/9) + \Theta(1)$
7. $Z(n) = Z(n/2) + \Theta(n)$

	c	Case	Complexity
$T(n)$			
$U(n)$			
$V(n)$			
$W(n)$			
$X(n)$			
$Y(n)$			
$Z(n)$			

Master Theorem exercises

- What is *c* for the following recurrences?
- What case does *f(n)* fall under?
- What is the asymptotic complexity for the following recurrences? Write "n/a" if the Master Theorem does not apply.

1. $T(n) = 2T(n/2) + \Theta(1)$
2. $U(n) = 4U(n/2) + \Theta(n^2)$
3. $V(n) = 9V(n/9) + \Theta(n\lg n)$
4. $W(n) = 3W(n/3) + \Theta(\lg n)$
5. $X(n) = 2X(n/4) + \Theta(n\lg n)$
6. $Y(n) = 3Y(n/9) + \Theta(1)$
7. $Z(n) = Z(n/2) + \Theta(n)$

	c	$f(n) =$	Complexity
$T(n)$	1	$O(n^{c-\epsilon})$	$\Theta(n)$
$U(n)$	2	$\Theta(n^c)$	$\Theta(n^2 \lg n)$
$V(n)$	1	n/a	n/a
$W(n)$	1	$O(n^{c-\epsilon})$	$\Theta(n)$
$X(n)$	0.5	$\Omega(n^{c+\epsilon})$	$\Theta(n \lg n)$
$Y(n)$	0.5	$O(n^{c-\epsilon})$	$\Theta(\sqrt{n})$
$Z(n)$	0	$\Omega(n^{c+\epsilon})$	$\Theta(n)$

Coming up

- Data structures
  - Lists
  - Stacks, queues, and deques
  - Trees
    - Binary search trees
    - Balanced BSTs

- **Recommended reading (today):** Section 11.1
  - *Practice problems:* R-11.1, C-11.1, C-11.3 (complexity only)
- **Recommended reading:** Sections 2.2, 1.4, and 2.1
  - *Practice problems:* R-2.3, R-2.2, R-2.6abc, C-2.12, C-2.19, A-2.1