“Sloppy” Counting

Thus –

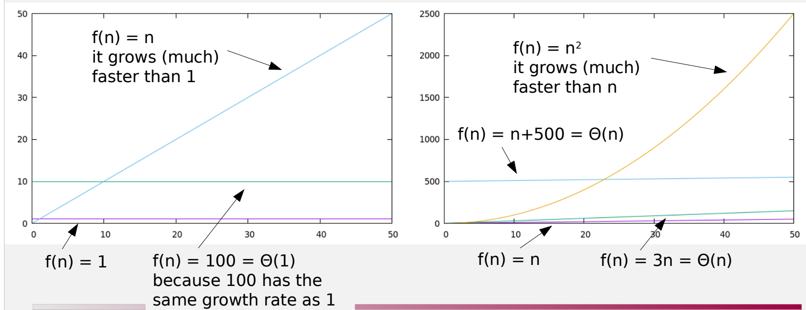
- focus on loops, and how the number of loop repetitions depends on the size of the input
  - identify what repeats the most

But –

- be aware of hidden loops – a method call is not one line of code, but rather all of the lines of code in its body

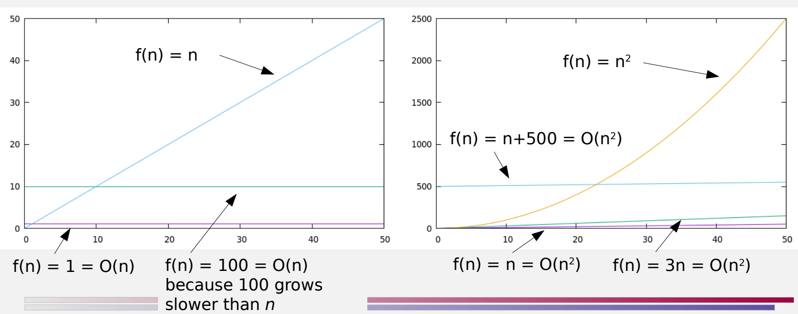
Notation

- Θ – “big-Theta”
  - f(n) = Θ(g(n)) means that f(n) and g(n) grow at the same rate – i.e. they have the same shape
  - guideline: drop constant multiples and lower-order (slower-growing) terms from f(n) to get a simpler g(n)



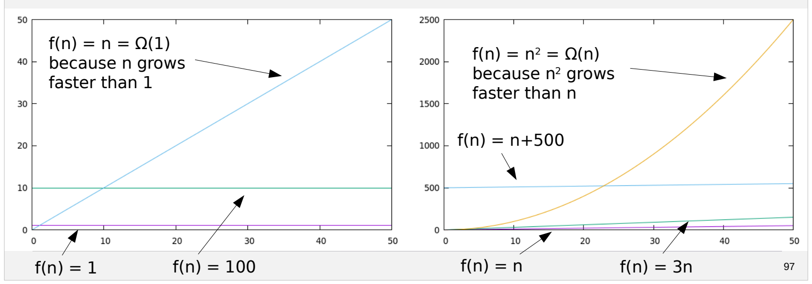
Notation

- O – “big-Oh”
  - f(n) = O(g(n)) means that f(n) grows no faster than g(n) – i.e. the growth rate of g(n) is an upper bound on the growth rate of f(n)
  - guideline: for algorithms, use O only if f(n) might grow slower than g(n) – perhaps it does in some cases, or the analysis to determine a more precise running time is too complex



Notation

- Ω – “big-Omega”
  - f(n) = Ω(g(n)) means that f(n) grows no slower than g(n) – i.e. the growth rate of g(n) is a lower bound on the growth rate of f(n)
  - guidelines for algorithms
    - use Ω only if f(n) might grow faster than g(n) – perhaps it does in some cases, or the analysis to determine a more precise running time is too complex
    - Ω is less commonly used because we are usually interested in the worst case – an upper bound on how long things will take



g(n) being a lower bound for f(n) means g(n)'s value stays below – f(n) can grow at the same rate or faster, but if g(n) grows faster its value will eventually be bigger than f(n)

Question	Defines a lower bound on the running time; the actual growth rate can be faster.	Defines a tight bound (both upper and lower) on the running time.	Defines an upper bound on the running time; the actual growth rate can be slower.	Defines a lower bound on the running time; the actual growth rate can be slower.	Defines an upper bound on the running time; the actual growth rate can be faster.
Ω(f(n))	✓ 5	0	0	5	0
Θ(f(n))	0	✓ 10	0	0	0
O(f(n))	0	0	✓ 6	0	4

g(n) being an upper bound for f(n) means g(n)'s value stays above – f(n) can grow at the same rate or slower, but if f(n) grows faster its value will eventually be bigger than f(n)

Self-Test

For each of the following functions f(n), find a simple function g(n) such that f(n) = Θ(g(n)).

– choose g(n) from 1, log n, √n, n, n log n, n^2, n^3, 2^n, n!

f(n)	g(n)
10 log n	
log n + 5	
3n + 4n log n	
$5n^2 + n + 100$	
20	
log 5	
$30n^2 + n^3$	
$100n + n^2$	
$n^2 + 2^n + 5$	
n + 5n + 10	
$10n \log n + 2n^2$	

guidelines –

– remember the order 1, log n, …

– drop constant multiples and lower-order (slower-growing) terms from f(n) to get a simpler g(n)

Self-Test

For each of the following functions f(n), find a simple function g(n) such that f(n) = Θ(g(n)).

– choose g(n) from 1, log n, √n, n, n log n, n^2, n^3, 2^n, n!

f(n)	g(n)
10 log n	Θ(log n)
log n + 5	Θ(log n)
3n + 4n log n	Θ(n log n)
$5n^2 + n + 100$	$\Theta(n^2)$
20	Θ(1)
log 5	Θ(1)
$30n^2 + n^3$	$\Theta(n^3)$
$100n + n^2$	$\Theta(n^2)$
$n^2 + 2^n + 5$	$\Theta(2^n)$
n + 5n + 10	Θ(n)
$10n \log n + 2n^2$	$\Theta(n^2)$

Examples

'numbers' is an array containing n elements

```
numbers[size] = elt;
size++;
```

only 2 steps no matter the size of 'numbers' → T(n) = Θ(1)

```
for ( int i = 0 ; i < numbers.length ; i++ ) {
  System.out.print(numbers[i] + " ");
}
```

loop body repeats n times → T(n) = Θ(n)

```
int count = 0;
for ( int i = 0 ; i < numbers.length ; i++ ) {
  for ( int j = i+1 ; j < numbers.length ; j++ ) {
    if ( numbers[i] > numbers[j] ) { count++; }
  }
}
```

inner loop body repeats n-1 + n-2 + n-3 + … + 1 times → $T(n) = \Theta(n^2)$

if you don't know how to compute this sum, you can also observe that the inner loop never repeats more than n times each time through the outer loop and the outer loop repeats no more than n times → n x n = $O(n^2)$
(in this case, using O instead of Θ reflects the fact that we are overcounting the number of repetitions of the inner loop – sometimes by quite a lot – and we don't know if we are overcounting by so much that it changes the growth rate of the function)

Self-Test

Give a big-Oh or big-Theta characterization, in terms of *n*, of the running time of each of the following functions.

```
public void ex1 ( int n ) {
  int a = 0;
  for ( int i = 0 ; i < n ; i++ ) {
    a = i;
  }
}
```

$\Theta(n)$

```
public void ex2 ( int n ) {
  int a = 0;
  for ( int i = 0 ; i < n ; i += 5 ) {
    a = i;
  }
}
```

$\Theta(n)$

```
public void ex3 ( int n ) {
  int a = 0;
  for ( int i = 0 ; i < n*n ; i++ ) {
    a = i;
  }
}
```

$\Theta(n^2)$

Self-Test

Give a big-Oh or big-Theta characterization, in terms of *n*, of the running time of each of the following functions.

```
public void ex4 ( int n ) {
  int a = 0, b = 0;
  for ( int i = 0 ; i < n ; i++ ) {
    a = i;
  }
  for ( int i = 0 ; i < n ; i++ ) {
    b = i;
  }
}
```

$\Theta(n)$

```
public void ex5 ( int n ) {
  int a = 0;
  for ( int i = 0 ; i < n ; i++ ) {
    for ( int j = 0 ; j <= i ; j++ ) {
      a = i*j;
    }
  }
}
```

counting the exact number of repetitions of the j loop – $\Theta(1+2+3+\ldots+n) = \Theta(n^2)$

useful sum:
$1+\ldots+n = n(n-1)/2$ $= n^2/2-n/2 =$ **$\Theta(n^2)$**

counting the j loop as "at most n repetitions each time" – $O(n^2)$

using O instead of Θ –
O means that the running time could grow slower than n^2 – we overcounted after all
Θ means it grows at the same rate as n^2 – which we only know if we counted carefully enough

Self-Test

Give a big-Oh or big-Theta characterization, in terms of *n*, of the running time of each of the following functions.

```
public void ex6 ( int n ) {
  int a = 0;
  for ( int i = 0 ; i < n*n ; i++ ) {
    for ( int j = 0 ; j <= i ; j++ ) {
      a = i*j;
    }
  }
}
```

$\Theta(1)$

n^2 times

$O(n^2)$ because the most times this loop will execute is n^2
(though most of the time it will be less than that)

$O(n^4)$ based on n^2 repetitions of the *i* loop and $O(n^2)$ each time for the *j* loop

but the j loop doesn't actually repeat n^2 times every time – maybe we are way overcounting

counting more carefully, the *j* loop repeats 1 time when i=0, 2 times when i=1, etc, resulting in a total time of
$1+2+3+\ldots+n^2 = \Theta(n^4)$
where we use the fact that the sum $1+2+\ldots+x = \Theta(x^2)$
(so we didn't overcount by too much)

Self-Test

Give a big-Oh or big-Theta characterization, in terms of *n*, of the running time of each of the following functions.

```
public void ex7 ( int n ) {
  for ( int i = 0 ; i < n ; i++ ) {
    ex1(i);
  }
}
```

$\Theta(i)$ if we count carefully, $O(n)$ if we observe that the biggest i gets is n

```
public void ex1 ( int n ) {
  int a = 0;
  for ( int i = 0 ; i < n ; i++ ) {
    a = i;
  }
}
```

$\Theta(n^2)$ based on ex1 taking time *i* $(1+2+3+\ldots+n = \Theta(n^2))$

$O(n^2)$ based on ex1 taking time *n* – n repetitions x $O(n)$ work per repetition = $O(n^2)$

Self-Test

Give a big-Oh or big-Theta characterization, in terms of *n*, of the running time of each of the following functions.

```
public void ex8 ( int n ) {
  int a = 0;     Θ(1)
  for ( int i = n ; i > 1 ; i /= 2 ) {  ? repetitions
    a = i;   Θ(1)
  }
}
```

$\Theta(\log n)$

when i = n/1 → Θ(1) work
when i = n/2 → Θ(1) work
when i = n/4 → Θ(1) work
when i = n/8 → Θ(1) work
when i = n/16 → Θ(1) work
…
when i = 2 → Θ(1) work
when i = 1 → stop

how many repetitions?
at rep j, i = $n/2^j$ where j = 0, 1, 2, …
which j gives $n/2^j = 1$?
solve for j…
$n = 2^j$
$\log_2 n = j \rightarrow j = \log n$

Which Input?

- *worst case* – the longest the algorithm could take on an input of a given size
  - most common measure but may not give an accurate picture if the worst case is slow but rare
- *best case* – the shortest the algorithm could take on an input of a given size
  - typically reported if it is different from the worst case
- *average case / expected case / typical case*
  - what's typical?
  - less common – requires knowing how likely different possible inputs are

Examples

'numbers' is an array containing n elements

```
int count = 0;
for ( int i = 0 ; i < numbers.length ; i++ ) {
  for ( int j = i+1 ; j < numbers.length ; j++ ) {
    if ( numbers[i] > numbers[j] ) { count++; }
  }
}
```

inner loop body repeats n-1 + n-2 + n-3 + … + 1 times → $T(n) = \Theta(n^2)$

```
int count = 0;
for ( int i = 0 ; i < numbers.length ; i++ ) {
  for ( int j = i+1 ; j < numbers.length ; j++ ) {
    if ( numbers[i] > numbers[j] ) { count++; break; }
  }
}
```

if numbers[i] > numbers[j] is never true, the inner loop body repeats n-1 + n-2 + n-3 + … + 1 times → $T(n) = \Theta(n^2)$

if numbers[i] > numbers[j] is true the first time, the inner loop body repeats 1 + 1 + 1 + … + 1 times → $T(n) = \Theta(n)$

→ $T(n) = O(n^2)$ – best case $\Theta(n)$, worst case $\Theta(n^2)$

Which Input?

- *worst case* – the longest the algorithm could take **on an input of a given size**
  - most common measure but may not give an accurate picture if the worst case is slow but rare
- *best case* – the shortest the algorithm could take **on an input of a given size**
  - typically reported if it is different from the worst case

Note: "…on an input of a given size"

- the best case is *not* the smallest possible input size
  - running time is always smaller – or at least not larger – for smaller inputs
- best and worst case are about the particular input instance
  - e.g. two slides ago, best case is decreasing order, worst case is increasing order

True or false: the worst case running time for an algorithm is for large values of *n*.

	Answer	Respondents	Percentage
✕	True	4	40%
✓	False	6	60%

an algorithm never takes less time on larger inputs than on small ones

best and worst case reflects differences for a given size of problem

```
int count = 0;
for ( int i = 0 ; i < numbers.length ; i++ ) {
  for ( int j = i+1 ; j < numbers.length ; j++ ) {
    if ( numbers[i] > numbers[j] ) { count++; break; }
  }
}
```

the inner (`j`) loop can repeat anywhere from 1 to `numbers.length-i+1` times depending on the exact values in `numbers`

More Sophistication

When the running time depends on more than just *n*, it can be meaningful to note that.

```
/**
 * Print the first k values in numbers.
 */
public static void print ( int k, int[] numbers ) {
  for ( int i = 0 ; i < k ; i++ ) {
    System.out.println(numbers[i]);
  }
}
```

- this could be described as worst case Θ(n) (because *k* might be `numbers.length`) and best case Θ(1) (because *k* might be 0) – O(n) in general
- but since the time really depends on *k* rather than *n*, Θ(k) is more meaningful

Key Points

- the idea of measuring running time in terms of input size
- the idea of counting loop repetitions
  - (including hidden loops – a method call takes the time of its method body even though the call itself is just one statement)
- big-Oh compares growth rates, not actual running times
  - "sloppy counting" ignores multiplicative factors and lower-order terms but these matter for actual running times
  - knowing $T_A(n) = O(T_B(n))$ doesn't tell you that program A will run faster than program B on any particular input
    - it does tell you that A's running time won't blow up faster than B's (so A will remain practical as long as or longer than B)
    - (though A is also likely to be faster than B when n is large enough)
- the ordering of typical growth rate functions from slower- to faster-growing
  - 1, log n, n, n log n, n^2, n^3, 2^n
- a sense of how much better/worse each is