CSC 148: Introduction to Computer Science

Week 12

Formalizing Big-O and Big-Theta

University of Toronto Mississauga,

Department of Mathematical and Computational Sciences

O(n)

- The stakes are very high when two algorithms …
  - Solve the same problem
  - But scale so differently with the size of the problem (we'll call that *n*).

- We want to express this scaling in a way that:
  - is simple
  - ignores the differences between different hardware, other processes on computer
  - ignores special behaviour for small n

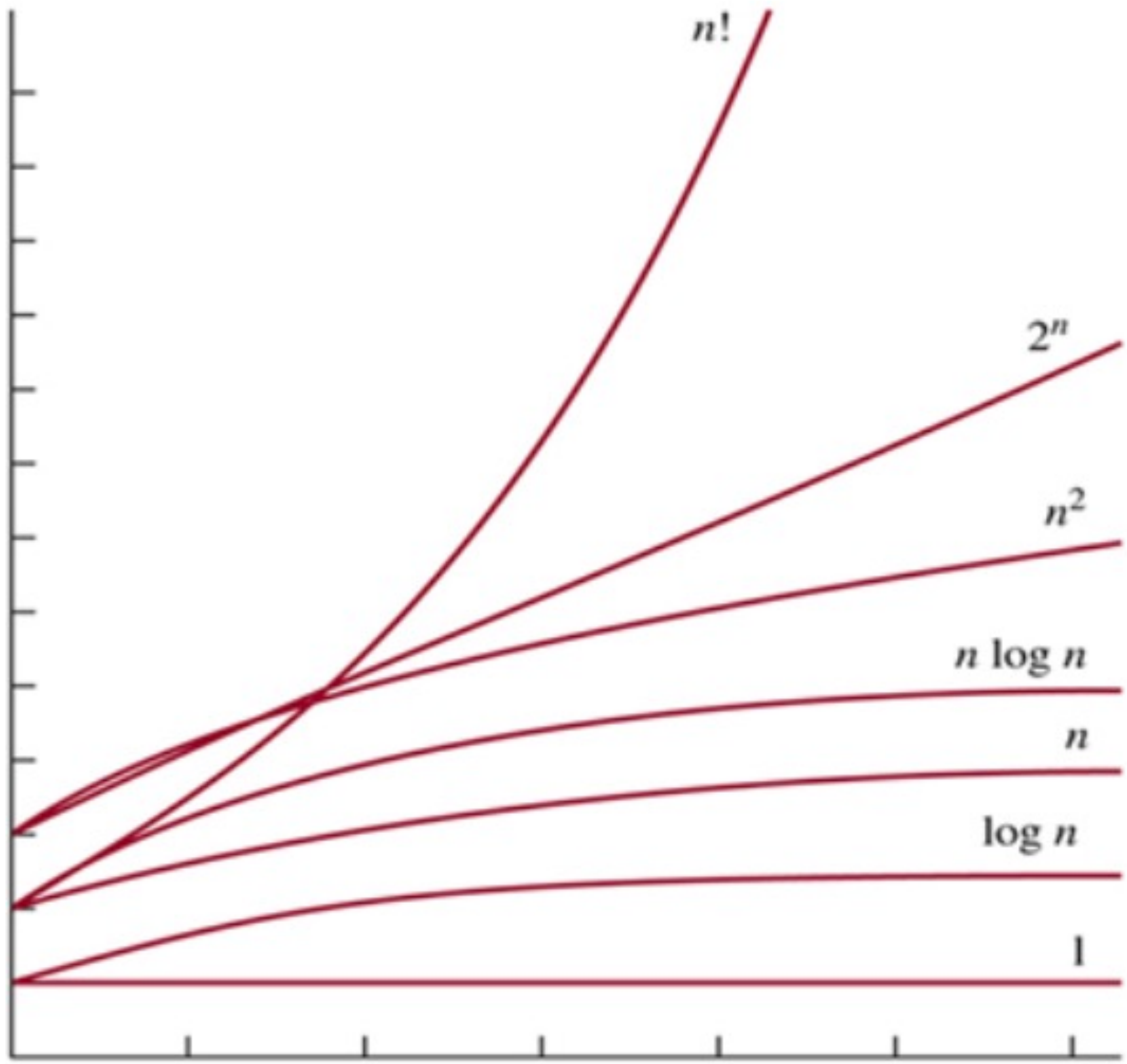
Big-O Definition

- Suppose that the number of "steps" (operations that don't depend on *n*, the input size) can be expressed as *f(n)*. We say that $f(n) \in O(g(n))$ if:
  - *there are positive constants c and n0, such that* $f(n) \leq c * g(n)$, *for any natural number* $n \geq n0$.

- Use graphing software:
  - $f(n) = 7n^2$
  - $f(n) = n^2 + 396$
  - $f(n) = 3960n + 4000$
- ... to see that the constant *c*, and the slower-growing terms don't change the scaling behaviour as *n* gets large

Efficiency of an algorithm matters!

- Comparison of function growth:



Considerations

- if f ∈ O(n), then it's also the case that f ∈ O(n lg n), f ∈ $O(n^2)$, and all larger bounds

- $O(1) \subseteq O(\lg(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(n!) \subseteq O(n^n)$

Disclaimer

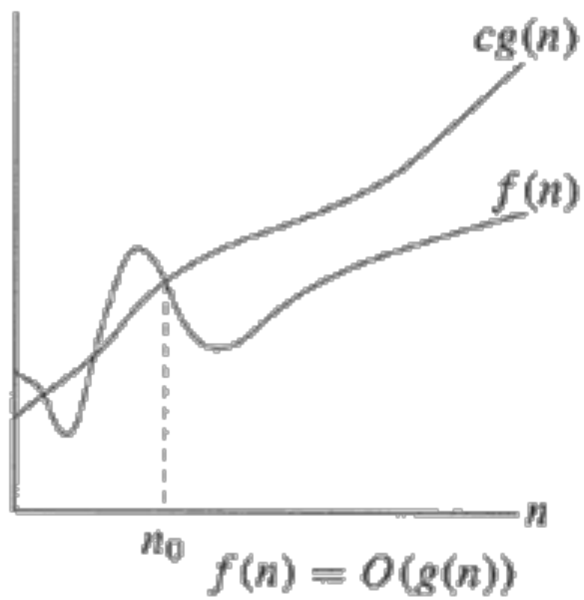
- We (computer scientists) commonly refer to O but often *mean* Θ

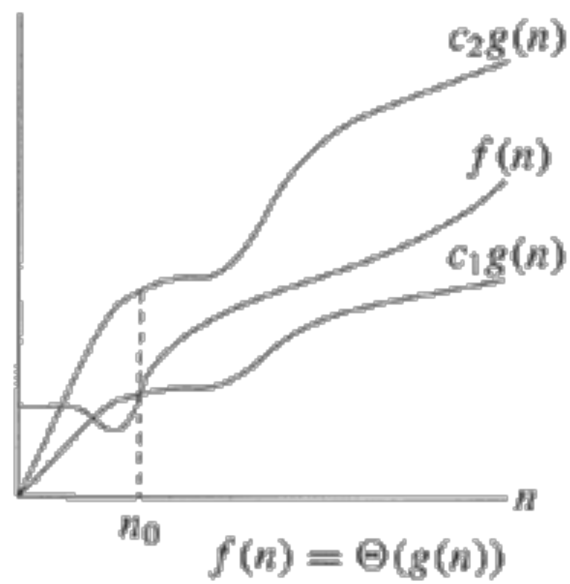
- What we're concerned about is the tightest upper bound
- So, while technically a function that has worst case running time proportional to *n log n* is also in $O(n^2)$, we wouldn't *say* that.

Big Theta

- Remember: we want the tightest bound
  - if an algorithm is $O(n)$ it's also $O(n^2)$
  - a $\Theta(n)$ algorithm is not $\Theta(n^2)$
- Big-theta: We say that $f(n) \in \Theta(g(n))$ if:
  - *there are positive constants c1, c2, and n0,*

    such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$, *for all* $n \geq n0$





Growth of Functions

- When n is arbitrarily large, growth of functions highly depends on the dominant term in the function:
  - $n + 42$
  - $n + 1000000$
  - $n^2 + n + 42$
  - $n^2 + 1000000n + 5$
  - $n^2 + n^3$
  - $n + \log n + n \log n$
  - $n + (\log n)^5 + n \log n$
  - $2^n + n^2$
  - $2^n + n^{200}$

Growth of Functions

- Ignore coefficients as well:
  - $20\ n + 42$ $\Theta(n)$
  - $20\ n + 1000000$ $\Theta(n)$
  - $100\ n^2 + n + 42$ $\Theta(n^2)$
  - $200\ n^2 + 1000000n + 5$ $\Theta(n^2)$
  - $n^2 + 50\ n^3$ $\Theta(n^3)$
  - $n + \log n + 500\ n \log n$ $\Theta(n \log n)$
  - $n + (\log n)^5 + 500\ n \log n$ $\Theta(n \log n)$
  - $2^n + 1000\ n^2$ $\Theta(2^n)$
  - $1000\ \ 2^n + 5000\ n^{200}$ $\Theta(2^n)$

Time Complexity of Algorithms

How time efficient is an algorithm, given input size of *n?*

- We measure time complexity in the order of **number of operations** an algorithm uses:
  - Big-O: an upper bound on the number of operations an algorithm conducts to solve a problem with input size of *n*
  - Big-Theta: a tighter bound on the number of operations an algorithm conducts to solve a problem with input size of n
- We will be looking at the **worst-case analysis** (number of operations in the worst case)
  - Don't confuse big-O, big-Theta with best/average/worst case analysis!

Time complexity: Example 1

```
def max(list):
    max  = list[0]
    for i  in range (len(list)):
        if  max < list[i]:    max = list[i]
    return  max
```

- Exact counting: count the number of comparisons
- Exactly $2n + 1$ comparisons are made
- Consider the dominant term, ignore constant coefficient
- => the time complexity of the max algorithm is $\Theta(n)$

Time complexity: Example 2

```
def max2(list):
    max  = list[0]
    i = 1
    while i  < len(list):
        if  max < list[i]:    max = list[i]
        i = i + 1
    return  max
```

- Exact counting: count the number of comparisons
- Exactly $2(n-1) + 1 = 2n - 1$ comparisons are made
- Consider the dominant term, ignore constant coefficient
- => the time complexity of the max2 algorithm is $\Theta(n)$

Time complexity: Example 3

```
def blah(n):
    n = 148 * n + n**(165)
    print ("n is equal to {}". format(n))
    if n > 1000:
        print ("n is over 1000")
    elif n > 100:
        print ("n is over 100")
    else:
        print ("n is under 100")
```

- Count the number of comparisons (assume that none happen in *print* or *format*)
- Exactly 2 comparisons => the time complexity of the *blah* function is $\Theta(1)$
- In general, consider the number of comparisons in any other functions called from *blah*! It may or may not depend on n!

Estimating Big-O and Big-Theta

- Instead of calculating the exact number of operations, and then using the dominant term, let's just focus on the dominant parts of the algorithm in the first place!

- Hint: look at loops and function calls!
- 2 things to watch:
  - 1. Carefully estimate the number of iterations in the loops in terms of algorithm's input size (i.e., n)
  - 2. If a called function depends on *n* (e.g., it has loops that have a number of iterations dependant on *n*), we should consider them in calculating the complexity

Time complexity: Example 1 (revisited)

```
def max(list):
    max  = list[0]
    for i  in range (len(list)):
        if  max < list[i]:    max = list[i]
    return  max
```

- Calculate big-Theta: focus on dominant part of this code
  - Assume len(list) = n
  - What's the dominant part of the algorithm?
  - What's the complexity of the dominant part of the algorithm?

=> the time complexity of the max algorithm is $\Theta(n)$

Time complexity: Example 2 (revisited)

```
def max2(list):
    max  = list[0]
    i = 1
    while i  < len(list):
        if  max < list[i]:    max = list[i]
        i = i + 1
    return  max
```

- Calculate big-Theta: focus on dominant part of this code
  - Assume len(list) = n
  - What's the dominant part of the algorithm?
  - What's the complexity of the dominant part of the algorithm?

=> the time complexity of the max2 algorithm is $\Theta(n)$

Time complexity: Example 3 (revisited)

```
def blah(n):
        n  = 148 * n + n**(165)
        print (“n is equal to {}”. format(n))
        if n  > 1000:
                print  (“n is over 1000”)
        elif n  > 100:
                print  (“n is over 100”)
        else:
                print  (“n is under 100”)
```

- Calculate big-Theta: focus on dominant part of this code
  - What’s the dominant part of this function? Any loops or function calls?
  - Are the dominant parts of code dependent on *n*?

=> the time complexity of the blah function is $\Theta(1)$

Time complexity: Example 4

```
def blah2(n):
    n  = 148 * n + n**(165)
    print (“n is equal to {}”. format(n))
    if n  > 1000:
        for i in range (n):   print  (“n is over 1000”)
    elif n  > 100:
        print  (“n is over 100”)
    else:
        print  (“n is under 100”)
```

- Calculate big-Theta ...
  - What are the dominant parts of code and what’s their complexity?
  - Keep in mind that we are looking for the worst-case complexity!

Time complexity: Example 5

```
def blah3(n):
    n  = 148 * n + n**(165)
    print ("n is equal to {}". format(n))
    if n  > 1000:
        print  ("n is over 1000")
    elif n  > 100:
        for i in range (n):   print  ("n is over 100")
    else:
        print  ("n is under 100")
```

- Calculate big-Theta ...
  - What are the dominant parts of code and what's their complexity?
  - In all cases, we are bound by a constant number of operations!

Time complexity: More examples ...

Ex 6. What is the time complexity of this piece of code?

```
sum = 0
for i in range (n//2):
    sum += i * n
```

How many times does the loop iterate?

½ n times

Ex 7. What is the time complexity of this piece of code?

```
sum = 0
for i in range (n//2):
    for j in range (n**2):
        sum += i * j
```

How many times does the loop iterate?

½ n * n^2 times

Time complexity: More examples ...

Ex 8. What is the time complexity of this piece of code?

```
sum = 0
for i in range (n//2):
    sum += i * n
for i in range (n//2):
    for j in range (n**2):
        sum += i * j
```

How many times does the loop iterate?

Total: $\frac{1}{2}n + \frac{1}{2}n * n^2$ times

Time complexity: More examples ...

Ex 9. What is the time complexity of this piece of code?

```
sum = 0
if n % 2 == 0:
   for i in range (n*n):
     sum += 1
else:
   for i in range (n+100):
          sum += i
```

For half of the values of n, complexity will be given by the n^2

For the other half proportional to n (keep in mind to discard the constants)

Time complexity: More examples ...

Ex 10. What is the time complexity of this piece of code?

```
i, sum = 0, 0
while i ** 2 < n:
    j = 0
    while j ** 2 < n:
        sum += i * j
        j += 1
    i += 1
```

The outer loop iterates $n^{1/2}$, the inner loop iterates $n^{1/2}$

=> $n^{1/2} * n^{1/2}$ => $\Theta(n)$

Time complexity: More examples ...

Ex 11. What is the time complexity of this piece of code?

```
i, j, sum  =  0, 0, 0
while i ** 2 < n:
    while j ** 2 < n:
        sum += i * j
        j += 1
    i += 1
```

The outer loop iterates $n^{1/2}$, but the inner loop does not reset j for each outer iteration!

=> $\Theta(n^{1/2})$

Time complexity: More examples ...

Ex 12. What is the time complexity of this piece of code?

```
count  =  0
while n > 1:
    n = n // 2
    count = count + 1
print(count)
```

The loop iterates **log(n)** times, because in each iteration n gets halved from what its value was in the previous iteration => $\Theta(\log n)$

Time complexity: More examples ...

Ex 13. What is the time complexity of this piece of code?

```
count, i  =  0, 1
while i < n:
        i = i * 2
        count = count + 1
print(count)
```

The loop iterates **log(n)** times, because in each iteration i gets doubled from what its value was in the previous iteration, hence reaching n in log(n) steps => $\Theta(\log n)$

Friday's lecture

- Odds & ends
- Tips for the future!
- Exam review

No class on Monday!