

CSC 148 Winter 2017

Week 11

Efficiency of algorithms

Bogdan Simion

bogdan@cs.toronto.edu

<http://www.cs.toronto.edu/~bogdan>



University of Toronto, Department of Computer Science



Outline

- Big-O notation and Big-Theta notation
- more big- Θ on paper
- big- Θ examples
- hash tables



$O(n)$

- 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



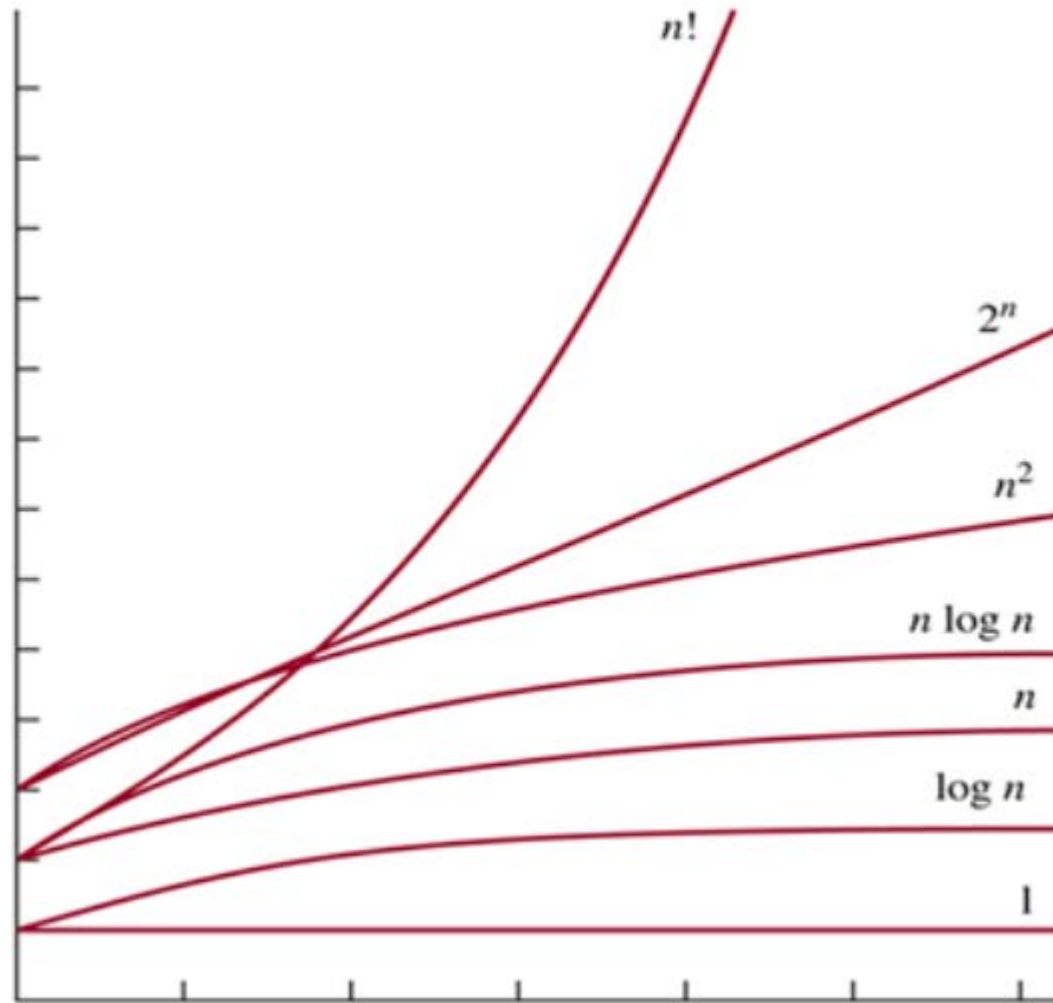
big-O definition

- 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 n_0 , such that $f(n) \leq c * g(n)$, for any natural number $n \geq n_0$.*
- 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 $f \in O(n)$, then it's also the case that $f \in O(n \lg n)$, $f \in O(n^2)$, and all larger bounds
- $O(1) \subseteq O(\lg(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n) \subseteq O(n!) \subseteq O(n^n)$



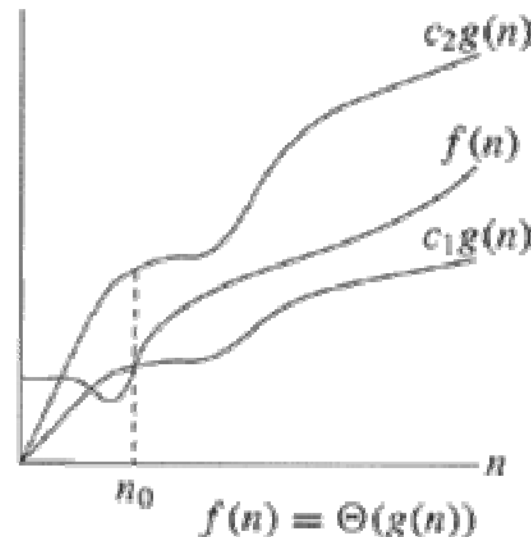
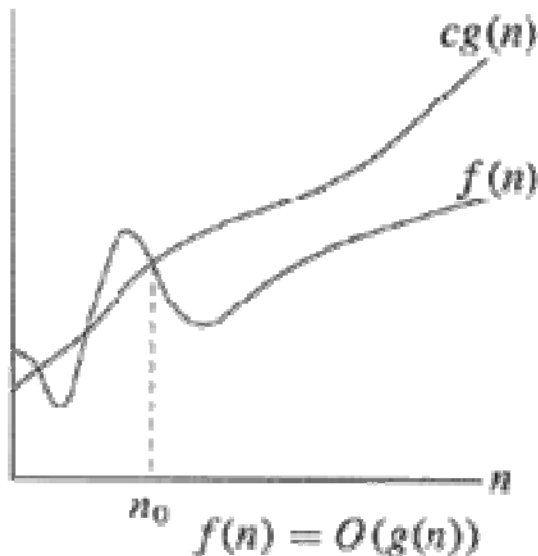
Disclaimer (for those of you taking CSC165...)

- We (computer scientists) commonly refer to O , but often mean Θ
- What we're concerned about is the **tighest** upper bound
- So, while technically a function that has worst case running time proportional to $n \log n$ is 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 c_1 , c_2 , and n_0 , such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$, for all $n \geq n_0$*





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



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
- \Rightarrow 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
- \Rightarrow 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 \Rightarrow 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 $\text{len}(\text{list}) = n$
 - 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 $\text{len}(\text{list}) = n$
 - 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?

$\frac{1}{2} n$ 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?

The outer loop iterates $\frac{1}{2} n * n^2$ 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?

The outer loop iterates $\frac{1}{2} n + \frac{1}{2} n * n^2$ 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 $n^{1/2}$

$\Rightarrow n^{1/2} * n^{1/2} \Rightarrow \theta(n)$



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 $\Rightarrow \Theta(\log n)$



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 $\Rightarrow \Theta(\log n)$

More complexity considerations

Hashing



University of Toronto, Department of Computer Science



Recall: python lists and our linked lists

- Python list is a contiguous data structure
 - Lookup is fast
 - Insertion and deletion is slow
- Our linked list is not a contiguous data structure
 - Lookup is slow
 - Insertion and deletion (**when it does not require lookup**) is fast

	lookup	insert	delete
Python lists	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Linked lists	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$



Recall: balanced BST

- BST can be implemented using linked lists
- Yet, it has a property that makes it more efficient when it comes to lookups: searching only takes $\log n$ steps!
- However, this comes at a price for insertion and deletion

	lookup	insert	delete
Python lists	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Linked lists	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

- Can we do better?



Can we do better?

- Assume an ideal “black box”:
 - Input: a key
 - Output: its index value in a list
- Turns out, we do have such a thing:
 - A pair of (key, index)
 - The key is the value that we want to search, insert, or delete, and the index is the location in the list
 - This is called a **hash function**



Hash function

- A hash function first “converts” a key to an integer value
- Then, it “compresses” that value into an index
- Just as a simple example:
 - The **conversion** can be done by applying some functions to binary values of the characters of the key
 - The **compression** can be done by some modulo operations.



Example (insertion)

- A class roster of up to 10 students:
 - We want to enroll “ANA”
 - Hash function:
 - Conversion component: for instance, returns sum of ASCII codes:
 $65 + 78 + 65 = 208$
 - Compression component, for instance, returns $208 \bmod 10 = 8$

=> We insert “ANA” at index 8 of the roster

- Similarly, if we want to enroll “ADAM”
 - Conversion? $65 + 68 + 65 + 77 = 275$
 - Compression? $275 \% 10 = 5$

=> We insert it at index 5 of the roster
- This is essentially how you can view a **hash table!**



Example (lookup)

- We want to lookup “ANA”
 - Hash function:
 - Conversion component returns sum of ASCII codes: $65+78+65 = 208$
 - Compression component, returns $208 \bmod 10 = 8$
 - => We check “ANA” at index 8 of the roster
- Similarly, if we want to lookup “ADAM”
 - Use hash function:
 - Conversion? $65 + 68 + 65 + 77 = 275$
 - Compression? $275 \% 10 = 5$
 - => We check index 5 of the roster



Python hash function

- Python has a built-in hash function. It's an error to use it on a list (try it!)
- An immutable list would be a tuple:

```
>>> hash((0, 1))  
1106628192
```
- Once you have hashed an object to a number, you can easily use part of that number as an index into a list to store the object, or something related to that object.
 - If the list is of length n , you might store information about object o at index $\text{hash}(o) \% n$



Recall: Complexity comparison

	lookup	insert	delete
Python lists	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Linked lists	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
BST	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash Table	$\Theta(1)^*$	$\Theta(1)^*$	$\Theta(1)^*$

- *Caveat: if there is no “collision”
 - What happens if several names hash to the same index?
 - We have to search several names at that index .. worst case: all of them!



Collisions

- How can collisions happen?
- What can we do when there is a collision?
 - **Chaining** (A bucket can contain a list (or other data structure) to hold more than 1 value)
 - **Open addressing** (probe next available bucket in some way)
 - Various ways of probing, Double-hashing, etc.
- There are overheads in each approach
 - Rule-of-thumb: if the number of items in the hash table exceeds 70% of the number of buckets, the overhead is too big, so we must increase the number of buckets



Collisions

- How can collisions happen?
- What can we do when there is a collision?
 - **Chaining** (A bucket can contain a list (or other data structure) to hold more than 1 value)
 - **Open addressing** (probe next available bucket in some way)
 - Various ways of probing, Double-hashing, etc.
- There are overheads in each approach
 - Rule-of-thumb: if the number of items in the hash table exceeds 70% of the number of buckets, the overhead is too big, so we must increase the number of buckets



Python dictionaries

- Python dictionaries are implemented using hash tables and probing. The cost of collisions is kept small by enlarging the underlying table when necessary, and the cost of enlarging is amortized over many dictionary accesses
 - e.g., double it when $> 2/3$ full
- Access to a dictionary element is $O(1)$, essentially the time it takes to access a list element.
- Shrinking the dictionary is tricky – not after a delete (to avoid thrashing, if the stale slot is needed soon), but a long sequence of deletes followed by an add may trigger shrinking the dictionary.



Course evaluations

- Available at:
 - <http://www.uoft.me/course-evs>