Algorithm Complexity

"How long is this gonna take?"

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

Reasons to study algorithm complexity

- Get a feel for algorithm time and space performance to operate on a specific data structure or structures
- Be able to meaningfully compare multiple algorithms' performance across a wide variety of input sizes
- Analyze best, typical, and worst-case behavior
- Reducing algorithm complexity is by far the most effective strategy for improving program performance;
  Aside: For my PhD, I discovered an approximation to a useful algorithm that dropped complexity from O(n^k) to O(nk)

Why can't we just time program execution?

- Execution time is a single snapshot that includes:
  - Choice of specific data structure(s) and algorithm(s)
  - Machine processor speed, memory bandwidth, possibly disk speed
  - Implementation language (in)efficiency (e.g., Python vs C)
  - One possible input (is it the best or worst-case scenario?)
  - One possible input size
- And, we have to actually implement an algorithm in order to time it
- (Measuring exec time is still useful)

Algorithm complexity to the rescue

- Complexity analysis encapsulates an algorithm’s performance across a wide variety of inputs and input sizes, *n*.
- In a sense, complexity analysis predicts future performance of your algorithm as, say, your company grows and the number of users on your website gets larger (be afraid of non-linear alg’s)
- We can compare performance of two algorithms without having to implement them
- Comparisons are independent of machine speed, implementation language, and any optimization work done by the programmer

Space vs time complexity

- **Space complexity** measures the amount of storage necessary to execute an algorithm as a function of input size
- **Time complexity** measures the amount of work ("time") necessary to execute an algorithm as a function of input size
- There is often a trade-off between using more memory and increasing speed
- Be aware that space complexity is a thing, but we will focus on time complexity in this class

If not exec time, what do we measure?

- We count fundamental operations of work; e.g., comparisons, floating-point operations, visiting nodes, traversing edges, swapping array elements, …
- For example, in sorting, we (usually) count the number of comparisons required to sort n elements
- Of primary interest is growth: how many more operations are required for each increase in input size
- If it takes 2 operations for input of size 2, how many operations are needed for input of size 3? Is it 2, 3, 4, 8, or worse?
- Define $T(n)$ = total operations required to operate on size n

Array sum example

- Let's count memory accesses (memory is slow) and floating-point additions
- Charge 2 operations to a single input element for each iteration (it's like accounting, charging work to input elements)
- $T(n) = \sum_{i=1}^{n} 2 = 2n$ which gives us great performance info!

```
s = 0.0
n = len(a)
for i in range(n):
    s = s + a[i]
```

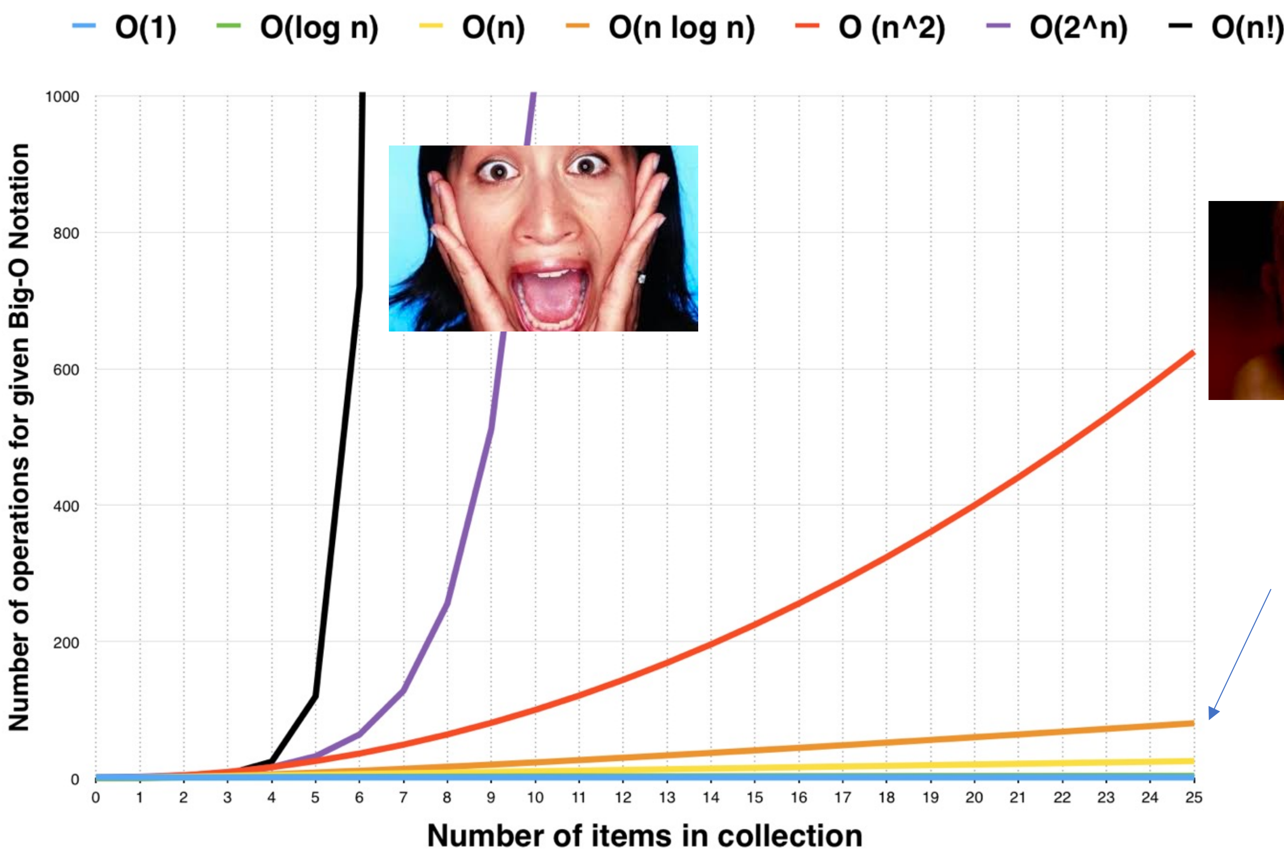
UNIVERSITY OF SAN FRANCISCO

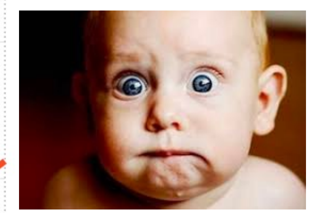
Sample execution times for T(n)

n f(n)	log n	n	n log n	n^2	2^n
10	0.003ns	0.01ns	0.033ns	0.1ns	1ns
20	0.004ns	0.02ns	0.086ns	0.4ns	1ms
30	0.005ns	0.03ns	0.147ns	0.9ns	1sec
40	0.005ns	0.04ns	0.213ns	1.6ns	18.3min
50	0.006ns	0.05ns	0.282ns	2.5ns	13days
100	0.07	0.1ns	0.644ns	0.10ns	$4x10^{13}$yrs
1,000	0.010ns	1.00ns	9.966ns	1ms	--
10,000	0.013ns	10ns	130ns	100ms	--
100,000	0.017ns	0.10ms	1.67ms	10sec	--
1'000,000	0.020ns	1ms	19.93ms	16.7min	--
10'000,000	0.023ns	0.01sec	0.23ms	1.16days	--
100'000,000	0.027ns	0.10sec	2.66sec	115.7days	--
1,000'000,000	0.030ns	1sec	29.90sec	31.7 years	--

From http://cooervo.github.io/Algorithms-DataStructures-BigONotation/index.html

Graphical view of growth





Should keep
your complexity
at or below O(n logn)

Plot from https://medium.freecodecamp.org/my-first-foray-into-technology-c5b6e83fe8f1

We care about asymptotic behavior

- We care about growth in effort given growth in input; i.e., what is the marginal cost to increase n to n+1?
- The best picture comes from imagining *n* getting very big and the worst-case input scenario
- This asymptotic behavior is called “big O” notation $O(n)$
- Therefore, ignore constants, keep only most important terms:
  - $T(n) = 2n$ implies $O(n)$
  - $T(n) = n^3 + kn^2 + nlogn$ implies $O(n^3)$
  - $T(n) = k$ for constant k implies $O(1)$
- E.g., $3n$! and $10n$! are indistinguishable asymptotically

Process

1. Identify what we are counting as a unit of work
2. Identify the key indicator(s) of problem size
   - Usually just some size *n*, but could be *n* ***and*** *m* if *n* x *m* matrix, etc…
   - Even for *n* x *m*, you could claim in worst-case that n is bigger, so *n* x *n* is input size but we usually compute complexity as a function of *n*
3. Define *T(n)* = … then solve sum or recurrence for closed form
4. Define *O(n)* as asymptotic behavior of *T(n)*

Tips

- With experience, you'll be able to go from algorithm description straight to *O(n)* by looking at max loop iterations etc…
- Look for loops and recursion
- Verify a loop steps by constant amount like 1 or k (not i *= 2)
- Look for patterns you know like binary search, sorting, traversing tree, etc…

Ask: What is the maximum amount of work?

- This approach often works great as we can focus on behavior rather than detailed analysis of the code
- Touching every element of a list means *O(n)*
- Touching every element of an *n* x *m* matrix means *O(nm)*=$O(n^2)$
- Touching every element of a tree with n nodes is *O(n)* but tracing the path from root to a leaf is *O(log n)* in balanced tree

Does it matter if tree is binary or trinary?

Nested loop examples

- Loops nested k deep, going around n times, are often O(n^k)

```
for i in range(n):
    for j in range(n):
        a = …
```

```
for i in range(n):
    for j in range(n):
        for k in range(n):
            a = …
```

What is cost of these loops assuming "a=…" costs k *operations?*

T(n) = $\sum_{i=1}^{n}\sum_{j=1}^{n} k = \sum_{i=1}^{n} nk = n^2k$, which is $O(n^2)$

UNIVERSITY OF SAN FRANCISCO

Recursive algorithms are trickier

- Define initial condition, such as T(0) = 0
- Define recurrence relation for recursion then turn the crank

$$T(n) = 1 + T(n-1)$$
$$T(n) = 1 + 1 + T(n-2)$$
$$T(n) = \underbrace{1 + 1 + 1}_{3} + T(n-3) = n + T(n-n) = n + T(0) = n + 0 = n$$

```
def sum(a): # recursive sum array
    if len(a)==0:
        return 0
    return a[0] + sum(a[1:])
```

Assumption: a[1:] takes constant time (not true in Python)

Linear search

```
def find(a,x): # find x in a
    n = len(a)
    for i in range(n):
         if a[i]==x: return i
    return -1
```

- Count comparisons
- Charge 1 comparison per loop iteration to each element
- *T(n)* is sum of *n* ones or *n*, giving *O(n),* same as sum(a)
- The intuition is that we have to touch every element of the input array once in the worst case
- What is complexity of max or argmax for array of size *n*?
- What is complexity to zero out an array of size *n*?
- Zero out matrix with n total elements? (careful)

Don’t count lines of code, count operations

- What is *O()* for findw()?
- Let *n* be len(words), *m* be len(a)

```
def findw(words, doc):
    c = 0
    for i in range(len(words)):
        if words[i] in doc:
            c += 1
    return c
```

- *T(n)* = $\sum_{i=1}^{n}(1 + cost\ of\ \text{in}\ operation)$
  = n + $\sum_{i=1}^{n} cost\ of\ \text{in}\ operation$
  = n + n * ???

Don't count lines of code

- What is *O()* for findw()?
- Let *n* be len(words), m be len(a)

```
def findw(words:list, doc:set):
    c = 0
    for i in range(len(words)):
        if words[i] in doc:
            c += 1
    return c
```

- $T(n) = \sum_{i=1}^{n} 1 + cost\ of \text{ in } operation$
  $= n + \sum_{i=1}^{n} cost\ of \text{ in } operation$
  $= n + \sum_{i=1}^{n} 1 = n + n = 2n$ which means this findw is O(n)

Don’t count lines of code

- What is *O()* for findw()?
- Let *n* be len(words), *m* be len(a)

```
def findw(words:list, doc:list):
    c = 0
    for i in range(len(words)):
        if words[i] in doc:
            c += 1
    return c
```

- $T(n) = \sum_{i=1}^{n} 1 + cost\ of \text{ in } operation$
  $= n + \sum_{i=1}^{n} cost\ of \text{ in } operation$
  $= n + \sum_{i=1}^{n} m = n + n \times m = n \times m$
- So, this findw is $O(nm)$ or, more commonly, $O(n^2)$

List operation	Worst Case
Copy	O(n)
Append[1]	O(1)
Pop last	O(1)
Pop intermediate	O(k)
Insert	O(n)
Get Item	O(1)
Set Item	O(1)
Delete Item	O(n)
Iteration	O(n)
Get Slice	O(k)
Set Slice	O(k+n)
Sort	O(n log n)
Multiply	O(nk)
x in s	O(n)
min(s), max(s)	O(n)
Get Length	O(1)

Set operation	Average Case	Worst Case
Copy	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration	O(n)	O(n)

From https://wiki.python.org/moin/TimeComplexity

Careful of loop iteration step size

- Let *n* be the input size
- Let's count math ops
- Charge 2 ops per iteration
- How many iterations?
- $T(1) = 0$
  $T(n) = 2 + T(n/2)$
  $= 2 + 2 + T(n/4)$
  $= \underbrace{2 + 2 + 2}_{3} + T(n/2^3)$ stop when 2^i reaches *n*, at T(n/n)=T(1)

```
def intlog2(n): # for n>=1
    if n == 1: return 0
    count = 0
    while n > 0:
        n = n // 2
        count += 1
    return count-1
```

Sum of log *n* twos = 2 *log n*, giving ***O(log n)***

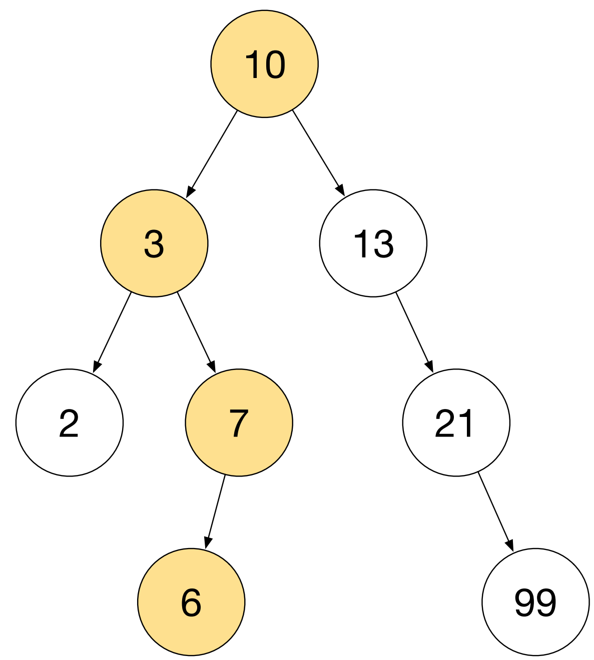
Ask how many times you can divide n by 2? log(n) times

Complexity of *binary search trees* (BST)

- BST: Nodes to left < current node, nodes to right are >
- Let *n* be num of values, count comparisons
- Charge 3 comparisons to each iteration
- How many iterations is key question?

```
p = root
while p is not None:
    if p.value==x: return p
    if x < p.value: p = p.left
    else: p = p.right
```

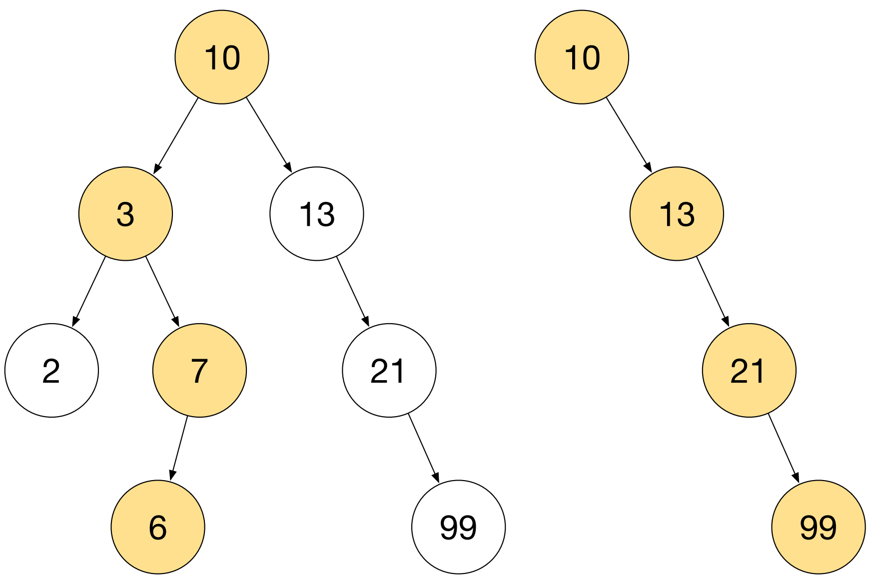
- What is average height? What is max height? (Use "what is max work" technique)



USUALLY faster than linear search

- Common case: T(n) = 3 + T(n/2), which we just saw is O(log n)
- Worst case: the tree is actually a linked list, which is O(n)

O(log n) *O(n)*




```
p = root
while p is not None:
    if p.value==x: return p
    if x < p.value: p = p.left
    else: p = p.right
```

Common recurrence relations / big O

Recurrence	Expanded	Complexity	Scenario
T(n) = 1 + T(n-1)	T(n) = 1 + 1 + 1 + T(n-3) = n	*O(n)*	Process one item then rest
T(n) = n + T(n-1)	T(n) = n + (n-1) + (n-2) + T(n-3) = n + (n-1) + (n-2) + … + 2 + 1 = n(n+1)/2 = $(n^2 + n)/2$ since $\sum_{i=1}^{n} i$ = n(n+1) / 2	$O(n^2)$	Looping through all *n* items, eliminating one from consideration each pass over items or nested loops
T(n) = 1 + T(n/2)	T(n) = 1 + 1 + 1 + T(n/8) = log n	*O(log n)*	Cut amount of work in half each iteration, doing 1 operation
T(n) = n + T(n/2)	T(n) = n + n/2 + n/4 + T(n/8) = = n + n/2 + n/4 + … + 2 + 1 = n(1 + ½ + ¼ + …) = n * 2	*O(n)*	Cut amount of work in half each iteration, but examine *n* items
T(n) = n+2T(n/2)	T(n) = n + 2T(n/2) = = n + 2(n/2 + 2T(n/4)) = n + 2n/2 + 4n/4 + 8T(n/8) = $\sum_{i=1}^{log n} n$ = n log n	*O(n log n)*	Divide and conquer algs. Cut amount of work in half each iteration, but process both halves, then combine results in linear time

Complexity	Scenario	Sample operations
O(1)	Perform constant number of ops	Hashtable lookup, access a[i], insert-after node in linked list
O(n)	Process one item then rest of items; or, cut amount of work in half each iteration, but examine *n* items	Linear search, zero an array, max, sum array, merge two sorted lists, insert into array, bucket sort
O(log n)	Cut amount of work in half each iteration, doing 1 operation	Binary search, search in binary search tree (BST), add to BST
O(n log n)	Divide and conquer algs. Cut amount of work in half each iteration, but process both halves, then combine results in linear time	Average quicksort, merge sort, median by sorting/picking middle item
$O(n^2)$	Looping through all *n* items, eliminating one from consideration each pass over the data	Touch all elems of *n* x *n* matrix, bubble sort, worst-case quicksort, process all pairs of *n* items

Compute complexity following our process

Algorithm 2 – example 1

Require: Input X with $|X| = n$

```
sum = 0
for i = 1 to n do
    for j = 1 to n do
        sum ← sum + 1
    end for
end for
for k = 1 to n do
    X_k ← k
end for
return X
```

1. Identify unit of work
2. Identify key size indicator
3. Define *T(n)* = …
4. Reduce *T(n)* to closed form
5. *O(n)* is asymptotic behavior of *T(n)*

Compute complexity following our process

Algorithm 2 – example 1

Require: Input X with $|X| = n$

```
sum = 0
for i = 1 to n do
    for j = 1 to n do
        sum ← sum + 1
    end for
end for
for k = 1 to n do
    X_k ← k
end for
return X
```

1. unit of work: assignment, addition
2. Identify key size indicator: n
3. *T(n)* = $\sum_{i=1}^{n}\sum_{j=1}^{n} 1 + \sum_{k=1}^{n} 1$
4. *T(n)* = n^2 + n (closed form)
5. *O(n^2)* asymptotic behavior

Compute complexities for these too

Algorithm 3 – example 2

```
1: sum1 = 0
2: for i = 1 to n do
3:     for j = 1 to n do
4:         sum1 ← sum1 + 1
5:     end for
6: end for
7: sum2 = 0
8: for i = 1 to n do
9:     for j = 1 to i do
10:        sum2 ← sum2 + 8
11:    end for
12: end for
```

- unit of work: assignment, addition
- Identify key size indicator: n
- $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 + \ldots$
- $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n} 1 + \sum_{i=1}^{n} \sum_{j=1}^{i} 1$
- $T(n)$ = n^2 + $\sum_{i=1}^{n} i$
- $T(n)$ = n^2 + $\sum_{i=1}^{n} i$
- $T(n)$ = n^2 + *n(n+1) / 2*
- $T(n)$ = n^2 + *n^2/2* + *n/2* = *3/2n^2* + *n/2*
- *O(n^2)* asymptotic behavior

lines 8, 9 are hallmark of matrix upper/lower triangle iteration

Summary

- O(…) is (tight) upper-bound on work done for given input size
- Independent of machine, language, algorithm details
- Process:
  1. Identify unit of work, key size indicator
  2. Define *T(n)* = … then find closed form
  3. Take asymptotic behavior of *T(n)* to get complexity