Algorithm Complexity

"How long is this gonna take?"

Terence Parr
MSDS program
University of San Francisco

UNIVERSITY OF SAN FRANCISCO

The goal

- Recall “algorithms + data structures = programs”
- Get a feel for algorithm performance operating 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 algorithm performance

Why can’t we just time program execution?

- Execution time is a single snapshot that measures:
    - Choice of specific data structure(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 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

If not exec time, what do we measure?

- We count fundamental operations of work; e.g., comparisons, floating-point operations, visiting nodes, 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 3, 4, 8, or worse?
- Define *T(n)* = total operations required to operate on size *n*

Array sum example

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

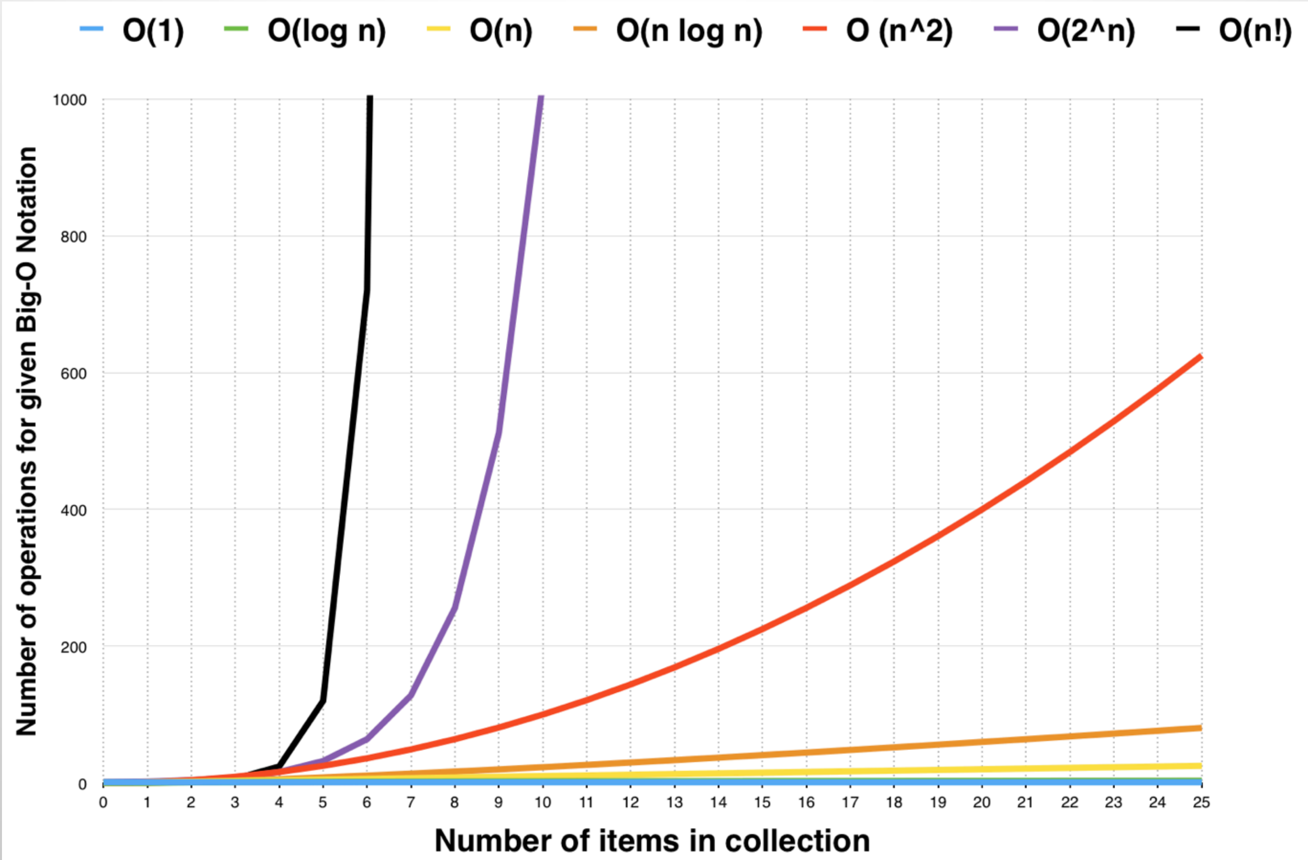
```
s = 0.0
n = len(a)
for i in range(n):
    s = s + a[i]
```

Sample execution times for T(n)

n f(n)	log n	n	n log n	n^2	2^n	n!
10	0.003ns	0.01ns	0.033ns	0.1ns	1ns	3.65ms
20	0.004ns	0.02ns	0.086ns	0.4ns	1ms	77years
30	0.005ns	0.03ns	0.147ns	0.9ns	1sec	8.4×10^{15}yrs
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	4×10^{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



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

Asymptotic behavior

- We count operations, not time, to make comparisons independent of algorithm impl language, machine speed, etc.
- We care about growth in effort given growth in input
- 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 implies O(1)

Process

- Identify what we are counting as a unit of work
- Identify the key indicator(s) of problem size
  - Usually just some size *n*, but could be *n*, *m* if *n* x *m* matrix, for example
  - Even for *n* x *m*, you could claim worst-case that n is bigger, so *n* x *n* is input size but we'll compute complexity as a function of *n*
- Define T(n) = …
- 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
- Looks for loops and recursion
- Verify loop steps by constant amount like 1 or k (e.g., not i *= 2)
- Loops nested k deep, going around n times, are often O(n^k)
- Ask yourself what the maximum amount of work is
  - Touching every element of the list means *O(n)*, touching every element of an *n* x *m* matrix means *O(nm)* or *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 worst-case *O(log n)*

UNIVERSITY OF SAN FRANCISCO

Recursive algorithms are trickier

- Define initial condition: $T(0) = 0$
- 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:])
```

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
- *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

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

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

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

Don't count lines of code

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

```
def findw(words:list, a:set):
    c = 0
    for i in range(len(a)):
        if words[i] in a:
            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(n)* for findw()?
- Let *n* be len(words), *m* be len(a)

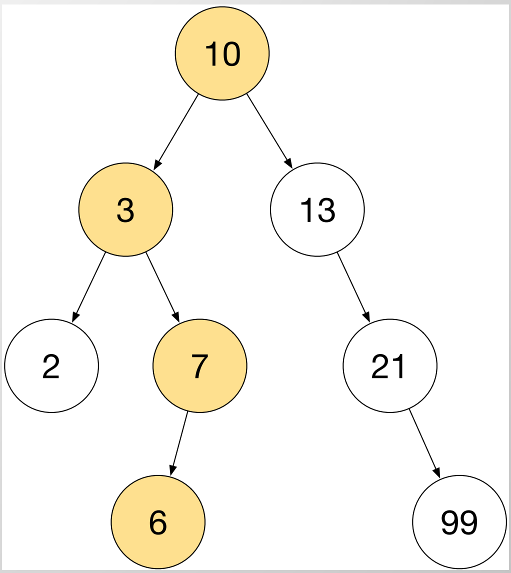
```
def findw(words:list, a:list):
    c = 0
    for i in range(len(a)):
        if words[i] in a:
            c += 1
    return c
```

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

Faster than linear search via trees…

- Let *n* be num of values, count comparisons
- Charge 2 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
```

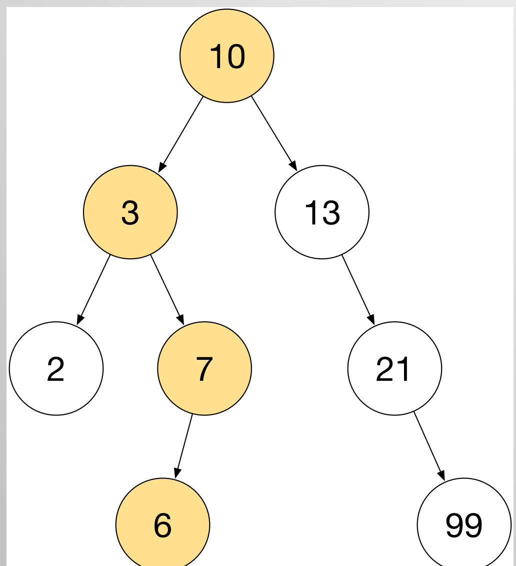


- What is average height? What is max height?

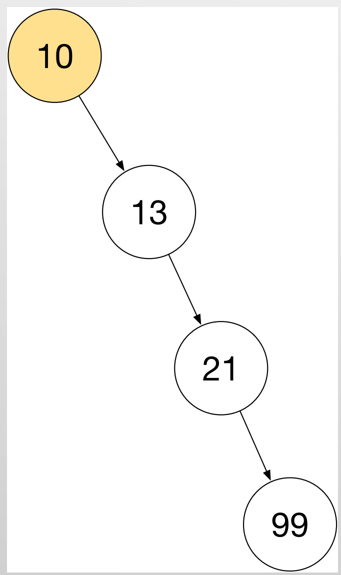
USUALLY faster than linear search

- What is average height? What is max height?

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
```

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 = int(n / 2)
        count += 1
    return count-1
```

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