

Algorithm Complexity

"How long is this gonna take?"

Terence Parr
MSDS program
University of San Francisco



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)



Reasons to study algorithm complexity

- Get a feel for algorithm time and space 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;
Aside: For my PhD, I discovered an approximation to a useful algorithm that dropped complexity from $O(n^k)$ to $O(nk)$



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



Array sum example

- Let's count array accesses (memory is slow) and floating-point additions
- Charge two operations to a single element for each iteration (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]
```



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



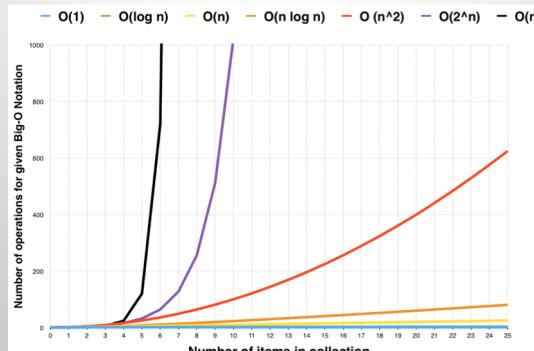
Sample execution times for $T(n)$

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	0.10ns	1.00ns	9.966ns	1ms	--	--	
10,000	0.13ns	10ns	130ns	100ms	--	--	
100,000	0.17ns	0.10ms	1.67ms	10sec	--	--	
1'000,000	0.20ns	1ms	19.93ms	16.7min	--	--	
10'000,000	0.23ns	0.01sec	0.23ms	1.16days	--	--	
100'000,000	0.27ns	0.10sec	2.66sec	115.7days	--	--	
1,000'000,000	0.30ns	1sec	29.90sec	1.8e+30	--	--	

From <http://coervo.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 $O(n)$
 - $T(n) = n^3 + kn^2 + n\log n$ 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 $n \times m$ matrix, for example
 - Even for $n \times m$, you could claim worst-case that n is bigger, so $n \times n$ is input size but we'll compute complexity as a function of n
- Define $T(n) = \dots$ then solve sum or recurrence for closed form
- 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 $i *= 2$)
- 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 $n \times m$ 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 $O(\log n)$ in balanced tree



Recursive algorithms are trickier

- Define initial condition, such as $T(0) = 0$
- Define recurrence relation for recursion then turn the crank
 $T(n) = 1 + T(n-1)$
 $T(n) = 1 + 1 + T(n-2)$
 $T(n) = 1 + 1 + 1 + T(n-3) = n + T(n-n) = n + T(0) = n + 0 = n$

```
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)



Don't count lines of code, count operations

- 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
```

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



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

- 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
```

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



## 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
```

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

UNIVERSITY OF SAN FRANCISCO

| 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                   | $O(k+n)$      |
| <u>Sort</u>                 | $O(n \log n)$ |
| Multiply                    | $O(nk)$       |
| $x \in s$                   | $O(n)$        |
| <code>min(s), max(s)</code> | $O(n)$        |
| 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>

UNIVERSITY OF SAN FRANCISCO

## 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?
- $T(1) = 0$
- $T(n) = 2 + T(n/2)$
- $= 2 + 2 + T(n/4)$
- $= 2 + 2 + 2 + T(n/2^3)$  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
```

3

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

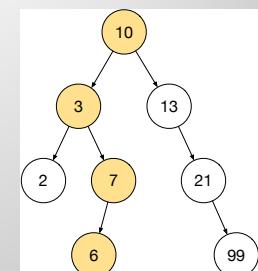
UNIVERSITY OF SAN FRANCISCO

## Faster than linear search via *binary search trees* (BST)

- BST: Nodes to left < current node, nodes to right are >
- 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
```

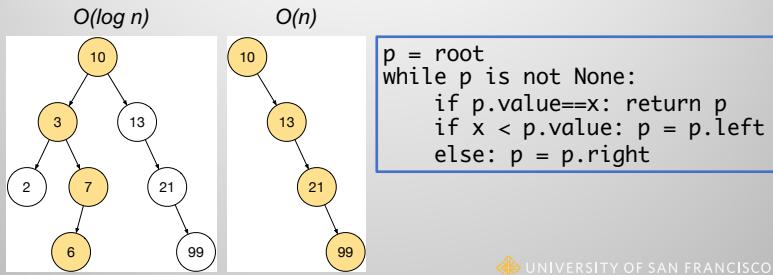
- What is average height? What is max height?



UNIVERSITY OF SAN FRANCISCO

## USUALLY faster than linear search

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



## 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                                                                                                      |
| $T(n) = n + T(n-1)$  | $T(n) = n + (n-1) + (n-2) + T(n-3) = n + (n-1) + (n-2) + \dots + 2 + 1 = n(n-1)/2 = n^2/2$ | $O(n^2)$      | Looping through all $n$ items, eliminating one from consideration each iteration 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                                                                    |
| $T(n) = n + T(n/2)$  | $T(n) = n + n/2 + n/4 + T(n/8) = n + n/2 + n/4 + \dots + 2 + 1 = 2n$                       | $O(n)$        | Cut amount of work in half each iteration, but examine $n$ items                                                                |
| $T(n) = n + 2T(n/2)$ | $T(n) = n + 2T(n/2) = n + 2(n/2) + 2T(n/4) = n + n + n + T(n/8) = n \log n$                | $O(n \log n)$ | Divide and conquer algs. Cut amount of work in half each iteration, but process both halves, the combine results in linear time |

UNIVERSITY OF SAN FRANCISCO

| Complexity    | Scenario                                                                                                                         | Sample operations                                                                                                 |
|---------------|----------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| $O(1)$        | Perform constant number of ops                                                                                                   | Hashtable lookup, access $a[i]$ , insert into middle of 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, find median |
| $O(n^2)$      | Looping through all $n$ items, eliminating one from consideration each iteration                                                 | Touch all elems of matrix, bubble sort, worst-case quicksort, process all pairs of $n$ items                      |
| $O(\log n)$   | 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, the combine results in linear time. | Average quicksort, merge sort, median by sorting/picking middle item                                              |

UNIVERSITY OF SAN FRANCISCO

## Compute complexity following our process

### Algorithm 2 – example 1

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

```

1: sum = 0
2: for i = 1 to n do
3: for j = 1 to n do
4: sum ← sum + 1
5: end for
6: end for
7: for k = 1 to n do
8: Xk ← k
9: end for
10: return X

```

- Identify unit of work
- Identify key size indicator
- Define  $T(n) = \dots$
- Reduce  $T(n)$  to closed form
- $O(n)$  is asymptotic behavior of  $T(n)$

UNIVERSITY OF SAN FRANCISCO

## Compute complexity following our process

---

**Algorithm 2 – example 1**


---

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

```

1: sum = 0
2: for i = 1 to n do
3: for j = 1 to n do
4: sum ← sum + 1
5: end for
6: end for
7: for k = 1 to n do
8: Xk ← k
9: end for
10: return X

```

---

- unit of work: assignment, addition
- Identify key size indicator:  $n$
- $T(n) = \sum_i^n \sum_j^n 1 + \sum_k^n 1$
- $T(n) = n^2 + n$  (closed form)
- $O(n^2)$  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

```

---

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



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

## Summary

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

