# CSC 212: Data Structures and Abstractions

## Big-O Notation

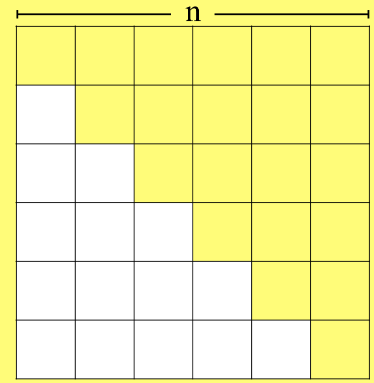
Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

# Practice




How many white squares as a function of n?

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$$



# Practice

How many cubes?

1 layer: 1
2 layers: 5
3 layers: 14
4 layers: 30

5 layers: ?

150 layers: ?

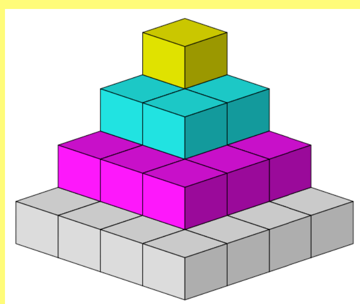


Image credit: Stanford's CS 106B lectures



# 2-sum (from lab)

- Problem
  - given an array of integers and a target, determine if there exist two elements in the array that add up to the target value

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 3 | -5 | 0 | 9 | -2 | 7 | 1 |

- Solutions
  - **brute-force**: examine all possible pairs (nested loops)
  - **sorting-based**: sort the array, then use two pointers, one starting at the beginning and the other at the end. Move the pointers inward based on the sum of the elements they point to
    - within the loop, calculate the sum, if sum < target we need a larger sum (move right), otherwise, we need a smaller sum (move left)

# 2-sum (from lab)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 3 | -5 | 0 | 9 | -2 | 7 | 1 |

$T(n) = \Theta(n^2)$

```
Algorithm TwoSumBrute(A, target, n)
    for i = 0 to n−2
        for j = i+1 to n−1
            if (A[i]+A[j]) == target
                return true
    return false
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| -5 | -2 | 0 | 1 | 3 | 4 | 7 | 9 |

$T(n) = \Theta(n \log n)$

dominated by the sorting operation

```
Algorithm TwoSumSort(A, target, n)
    Sort(A, n)
    p = 0
    q = n − 1
    while p < q
        sum = A[p] + A[q]
        if sum == target
            return true
        else if sum < target
            p = p + 1
        else
            q = q − 1
    return false
```



# Order of growth for different input sizes

| Size | T(n) = log n | T(n) = n | T(n) = n log n | T(n) = n² | T(n) = n³ |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 |
| 10 | 3 | 10 | 33 | 100 | 1,000 |
| 100 | 7 | 100 | 664 | 10,000 | 1,000,000 |
| 1,000 | 10 | 1,000 | 9,966 | 1,000,000 | 1,000,000,000 |
| 10,000 | 13 | 10,000 | 132,877 | 100,000,000 | 1,000,000,000,000 |
| 100,000 | 17 | 100,000 | 1,660,964 | 10,000,000,000 | 1,000,000,000,000,000 |
| 1,000,000 | 20 | 1,000,000 | 19,931,569 | 1,000,000,000,000 | 1,000,000,000,000,000,000 |
| 10,000,000 | 23 | 10,000,000 | 232,534,967 | 100,000,000,000,000 | 1,000,000,000,000,000,000,000 |
| | rounded | | rounded | | |



# 3-sum (from lab)

- Problem
  - given an array of integers and a target, determine if there exist three elements in the array that add up to the target value

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 3 | -5 | 0 | 9 | -2 | 7 | 1 |

- Solutions
  - **brute-force**: examine all possible triplets (three nested loops)
  - **sorting-based**: sort the array, then iterate through the array from left to right
    - for each element, use the 2-sum approach (two pointers) on the remaining part of the array to find if there are two other elements that sum up to the target minus the current element



# 3-sum (from lab)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 3 | -5 | 0 | 9 | -2 | 7 | 1 |

$T(n) = \Theta(n^3)$

```
Algorithm ThreeSumBrute(A, target, n)
    for i = 0 to n−3
        for j = i+1 to n−2
            for k = j+1 to n−1
                if (A[i]+A[j]+A[k]) == target
                    return true
    return false
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| -5 | -2 | 0 | 1 | 3 | 4 | 7 | 9 |

$T(n) = \Theta(n^2)$

```
Algorithm ThreeSumSorted(A, target, n)
    Sort(A, n)
    for i = 0 to n−3
        if TwoSumSorted(A[i+1:end], target−A[i])
            return true
    return false
```

NO NEED to sort within the TwoSumSorted function

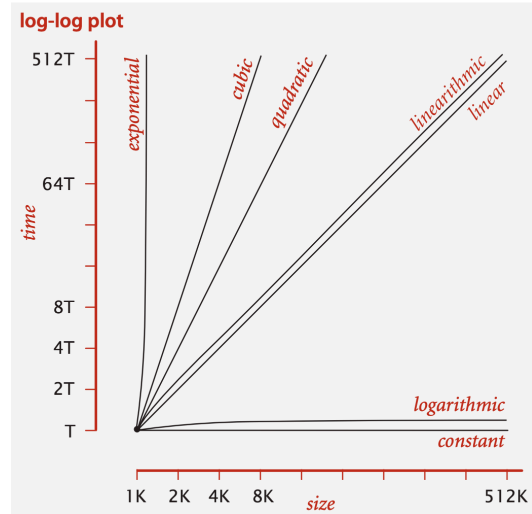
| description | order of growth | typical code framework | description | example |
|---|---|---|---|---|
| *constant* | 1 | `a = b + c;` | *statement* | *add two numbers* |
| *logarithmic* | $\log N$ | [ *see page 47* ] | *divide in half* | *binary search* |
| *linear* | $N$ | `double max = a[0];`<br>`for (int i = 1; i < N; i++)`<br>`   if (a[i] > max) max = a[i];` | *loop* | *find the maximum* |
| *linearithmic* | $N \log N$ | [ *see* ALGORITHM 2.4 ] | *divide and conquer* | *mergesort* |
| *quadratic* | $N^2$ | `for (int i = 0; i < N; i++)`<br>`   for (int j = i+1; j < N; j++)`<br>`      if (a[i] + a[j] == 0)`<br>`         cnt++;` | *double loop* | *check all pairs* |
| *cubic* | $N^3$ | `for (int i = 0; i < N; i++)`<br>`   for (int j = i+1; j < N; j++)`<br>`      for (int k = j+1; k < N; k++)`<br>`         if (a[i] + a[j] + a[k] == 0)`<br>`            cnt++;` | *triple loop* | *check all triples* |
| *exponential* | $2^N$ | [ *see* CHAPTER 6 ] | *exhaustive search* | *check all subsets* |

Common order of growth classifications

https://algs4.cs.princeton.edu/14analysis/



# Typical order of growth functions



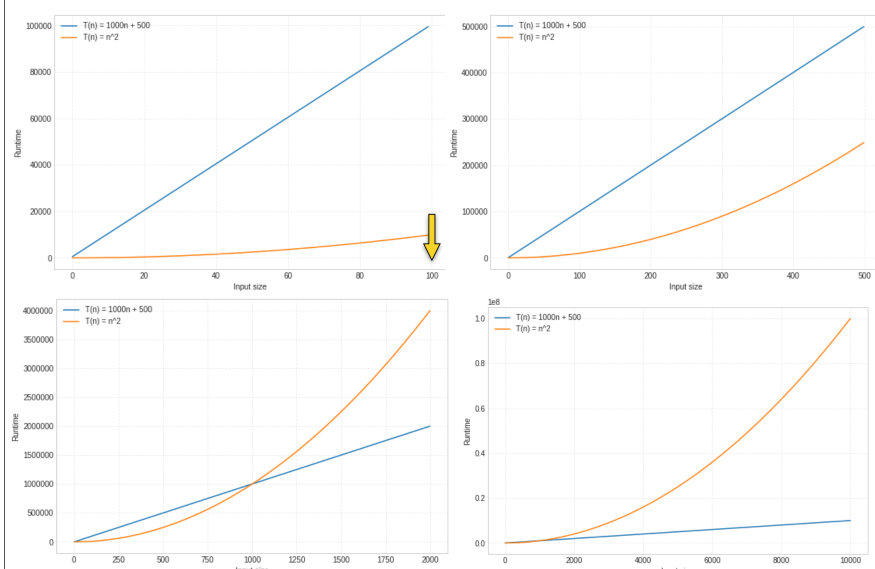

This set of functions is enough to describe the order of growth of the most common algorithms

https://algs4.cs.princeton.edu/lectures/keynote/14AnalysisOfAlgorithms.pdf



# Asymptotic notation

# Consider these functions



# Asymptotic analysis

- For a given algorithm, analyze $T(n)$ as the input size $n \to \infty$
  - we are interested in the **behavior** of the algorithm as the size of the input **grows**, NOT in the exact number of operations
- In practice:
  - may **ignore** constant factors (coefficients) and lower-order terms
    - when $n$ is large, constants and lower-order terms are negligible

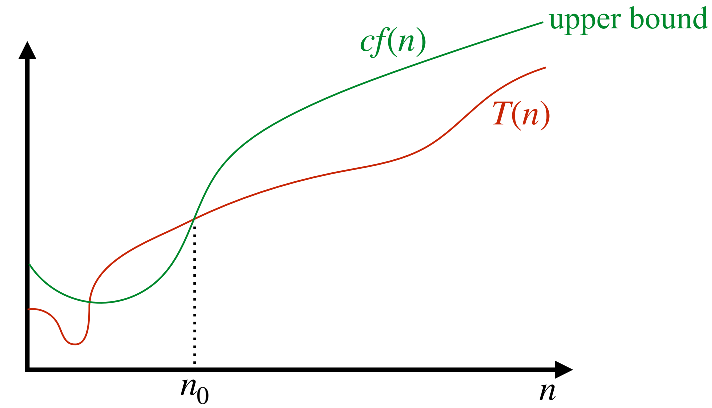
| | |
|---|---|
| $3n^3 + 50n + 24$ | $\Theta(n^3)$ |
| $10^{10}n + \frac{n^2}{1000} + 10^5$ | $\Theta(n^2)$ |
| $4n^5 + 2^n - \frac{16}{5}$ | $\Theta(2^n)$ |
| $4 \log n + n \log n$ | $\Theta(n \log n)$ |

Θ-notation used to describe tight bounds on the growth rate of functions



# Big O

- $T(n)$ grows no faster than $f(n)$ asymptotically




$T(n)$ is $O(f(n)) \iff \exists$ positive $c, n_0 \mid 0 \le T(n) \le cf(n), \forall n \ge n_0$



# Practice

- Mark true if $T(n) = O\left(f(n)\right)$

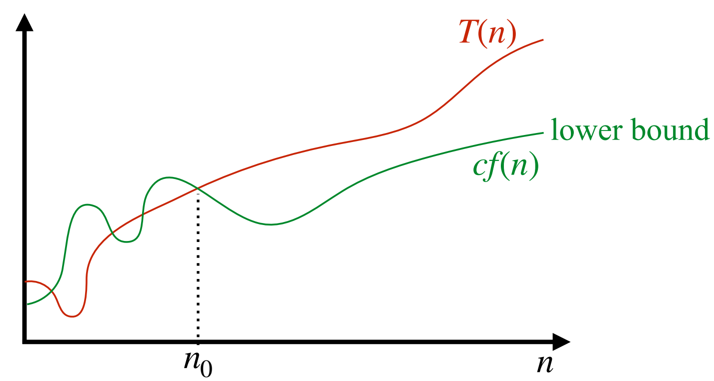
$f(n)$

| $T(n)$ | $n^2$ | $n^4$ | $2^n$ | $\log n$ |
|---|---|---|---|---|
| $10^2 + 3000n + 10$ | | | | |
| $21 \log n$ | | | | |
| $500 \log n + n^4$ | | | | |
| $\sqrt{n} + \log n^{50}$ | | | | |
| $4^n + n^{5000}$ | | | | |
| $3000n^3 + n^{3.5}$ | | | | |
| $2^5 + n!$ | | | | |



# Big Omega

- $T(n)$ grows at least as fast as $f(n)$ asymptotically




$T(n)$ is $\Omega(f(n)) \iff \exists$ positive $c, n_0 \mid 0 \le cf(n) \le T(n), \forall n \ge n_0$

# Practice

- Mark true if $T(n) = \Omega\left(f(n)\right)$

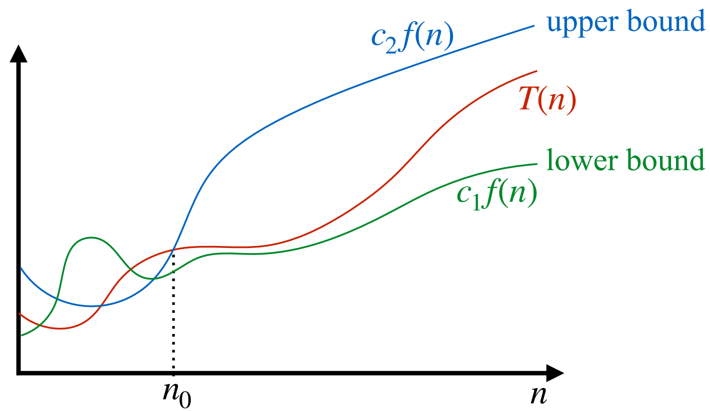
$f(n)$

| $T(n)$ | $n^2$ | $n^4$ | $2^n$ | $\log n$ |
|---|---|---|---|---|
| $10^2 + 3000n + 10$ | | | | |
| $21 \log n$ | | | | |
| $500 \log n + n^4$ | | | | |
| $\sqrt{n} + \log n^{50}$ | | | | |
| $4^n + n^{5000}$ | | | | |
| $3000n^3 + n^{3.5}$ | | | | |
| $2^5 + n!$ | | | | |



# Big Theta

- $T(n)$ grows at exactly the same rate as $f(n)$ asymptotically




$T(n)$ is $\Theta(f(n)) \iff T(n)$ is $O(f(n))$ and $T(n)$ is $\Omega(f(n))$



# Practice

- Mark true if $T(n) = \Theta\left(f(n)\right)$

$f(n)$

| $T(n)$ | $n^2$ | $n^4$ | $2^n$ | $\log n$ |
|---|---|---|---|---|
| $10^2 + 3000n + 10$ | | | | |
| $21 \log n$ | | | | |
| $500 \log n + n^4$ | | | | |
| $\sqrt{n} + \log n^{50}$ | | | | |
| $4^n + n^{5000}$ | | | | |
| $3000n^3 + n^{3.5}$ | | | | |
| $2^5 + n!$ | | | | |



# Growth rates in practice

- Key Insight
  - **<u>asymptotic analysis</u>** determines efficiency for large values of n
  - e.g., if $n = 100000$
    - $\Theta(n^2) = 10^{10}$ operations
    - $\Theta(n^3) = 10^{15}$ operations, much slower!
- Caveat
  - we shouldn't completely ignore asymptotically slower algorithms
    - they might have a lower constant factor and perform better for small inputs
    - they could be simpler to implement (hardware considerations relevant)
    - they could use less memory
- Takeaway
  - while asymptotic complexity matters for scalability, real-world performance depends on multiple factors!

# Growth rates in practice

- The question of Big-O versus Big-Θ notation
  - from a strictly mathematical perspective, Big-Θ notation provides a more precise bound
    - $\Theta(n^2)$ indicates $T(n)$ grows no faster and no slower than $n^2$ (up to constant factors)
    - $O(n^2)$ only specifies an upper bound
- Prevalence of Big-O notation in CS
  - in many cases where computer scientists use $O(f(n))$, they are actually describing a $\Theta(f(n))$ bound, but the community has implicitly accepted this *slight abuse of notation*
  - computer scientists are often more concerned with establishing worst-case upper bounds
  - in software engineering practice, focus is predominantly on ensuring performance doesn't exceed certain bounds, making Big-O notation more directly applicable