CSC 212: Data Structures and Abstractions

04: Big-O Notation

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2026

2-sum

- Problem
  - given an array of integers and a <u>target</u>, determine if there exist <u>two elements</u> 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

```cpp
bool two_sum_bf(const std::vector<int>& A, int tgt) {
    for (size_t i = 0 ; i < A.size() ; ++i) {
        for (size_t j = i + 1 ; j < A.size() ; ++j) {
            if (A[i] + A[j] == tgt) {
                return true;
            }
        }
    }
    return false;
}
```

$$T(n) = (n-1) + (n-2) + \ldots + 1 + 0$$

$$= \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$

worst-case analysis

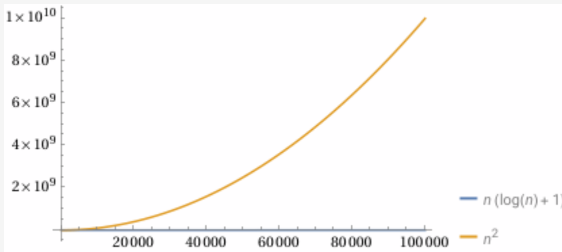
2-sum

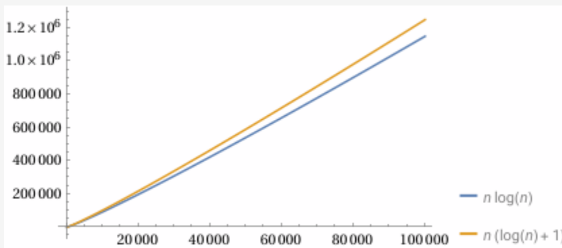
```cpp
bool two_sum_tp(const std::vector<int>& A, int tgt) {
    const size_t n = A.size();
    if (n < 2) return false;
    std::vector<int> B = A;
    std::sort(B.begin(), B.end());
    size_t left = 0, right = n - 1;
    while (left < right) {
        int current_sum = B[left] + B[right];
        if (current_sum == tgt) return true;
        else if (current_sum < tgt) left ++;
        else right --;
    }
    return false;
}
```

$$T(n) = T_{sort}(n) + n$$

$$= n \log n + n$$

worst-case analysis





https://www.wolframalpha.com/

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 4 mins
100,000	17	100,000	1,660,964	10,000,000,000	1,000,000,000,000,000 3 days
1,000,000	20	1,000,000	19,931,569	1,000,000,000,000	1,000,000,000,000,000,000 8 years
10,000,000	23	10,000,000	232,534,967	100,000,000,000,000	1,000,000,000,000,000,000,000 7900 years
	rounded		rounded		assume a basic 4Ghz processor

3-sum

- 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

```
function ThreeSumBrute(A, target)
    n = length(A)
    for i = 0 to n-1
        for j = i+1 to n-1
            for k = j+1 to n-1
                if (A[i]+A[j]+A[k]) == target)
                    return true
    return false
```

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}\sum_{k=j+1}^{n-1} 1 = \dots = \frac{n(n-1)(n-2)}{6}$$

```
function ThreeSumSorted(A, target)
    n = length(A)
    B = copy(A)
    Sort(B)
    for i = 0 to n-3
        if TwoSumSorted(B[i+1:n-1], target-B[i])
            return true
    return false
```

NO NEED to sort within the TwoSumSorted function

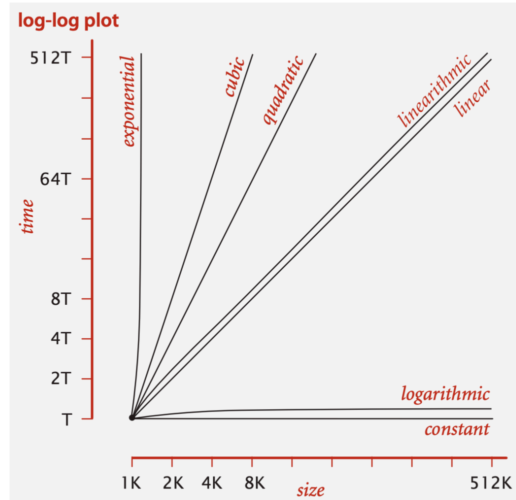
$$T(n) = n\log n + \sum_{i=0}^{n-3}(n-i-1) = \dots = n\log n + \frac{n^2-n-2}{2}$$

Typical order of growth functions

Function	Name	Example algorithm(s)
1	Constant	Array element access, push/pop on stack
log n	Logarithmic	Binary search
n	Linear	Linear search, traversing a list
n log n	Linearithmic	Merge sort, Heap sort
n^2	Quadratic	Bubble sort, Insertion sort, Selection sort
n^3	Cubic	Naive matrix multiplication
2^n	Exponential	Recursive Fibonacci
n!	Factorial	Generating all permutations

`We seek to characterize an algorithm's cost` <u>`not by its exact function`</u>`, but by its` **`asymptotic growth rate`**`, capturing how the function scales with input size while abstracting away constant factors and lower-order terms.`

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

Asymptotic analysis

- How does an algorithm's performance scale with input size?
  - ✓ machine-independent analysis
  - ✓ want to analyze the behavior of $T(n)$ as $n \to \infty$, NOT the exact number of operations
  - ✓ example: is $T(n) = 1000n$ better than $T(n) = n^2$ for large n?
- Asymptotic growth intuition
  - ✓ for sufficiently large inputs, the highest order term dominates
  - ✓ example:
    - consider $T(n) = 3n^2 + 100n + 500$

n	$3n^2$	100n	T(n)	$3n^2$ / T(n)
10	300	1k	1.8k	0.16
100	30k	10k	40.5k	0.74
1000	3M	100k	3.1M	0.97

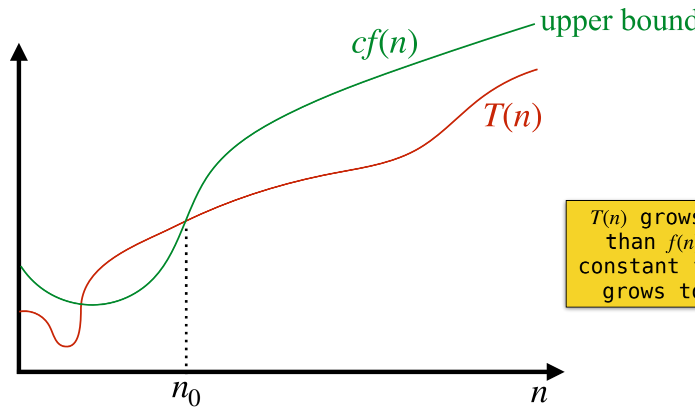
Asymptotic analysis

- In practice:
  - **<u>ignore</u>** constant factors (coefficients) and lower-order terms
    - when n is large, constants and lower-order terms are negligible

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

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

Big O



$T(n)$ grows no faster than $f(n)$ (up to a constant factor) as n grows to infinity

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

Practice

- Prove that the function $T(n) = 8n - 2$ is $O(n)$
  - find positive constants c, n_0 such that $0 \le 8n - 2 \le cn$ for every integer $n \ge n_0$
  - possible choice:
    - $c = 8, \quad n_0 = 1$

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

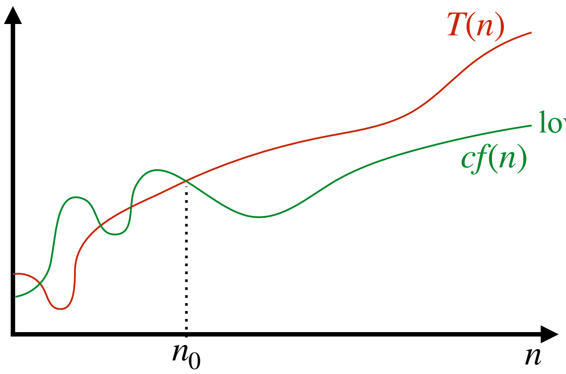
Practice

- 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 $f(n)$ (up to a constant factor) as n grows to infinity

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

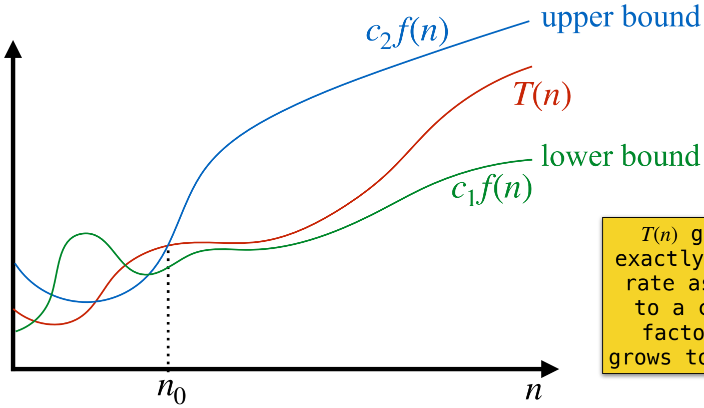
Practice

- 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 $f(n)$ (up to a constant factor) as n grows to infinity

$T(n)$ 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

- **Asymptotic analysis** determines efficiency for large values of n
  - ✓ e.g., two algorithms perform $T_A(n) = 100n$ and $T_B(n) = n^2$ operations respectively
    - for large values of n, algorithm A is superior as $\Theta(n) \ll \Theta(n^2)$
    - $n = 100000$
      - $T_A(n) = 10^7$ operations
      - $T_B(n) = 10^{10}$ operations, much slower!
- However, asymptotically slower algorithms may still be preferable, when they:
  - ✓ have significant lower constant factors and/or operate on small inputs
  - ✓ are simpler to implement
  - ✓ require substantially 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
  - ✓ big-Θ notation provides tight bounds
    - $T(n)$ is $\Theta(n^2)$ means $T(n)$ grows at the same rate as n^2
  - ✓ big-O notation provides upper bounds only
    - $T(n)$ is $O(n^2)$ means $T(n)$ grows no faster than n^2
- Prevalence of Big-O notation in CS
  - ✓ computer scientists routinely use $O(f(n))$ when discussing algorithm complexity, even when the actual complexity is $\Theta(f(n))$, because the field has adopted Big-O as the conventional notation for expressing algorithmic efficiency regardless of bound tightness