Data Structures and Algorithms

Complexity Analysis

Objectives

- Introduction
- Define an algorithm
- Define growth rate of an algorithm as a function of input size
- Classify functions based on growth rate
- Define growth rates: Big O, Theta, and Omega

Algorithm

- What’s the definition of Algorithm?

Algorithm

- What’s the definition of Algorithm?

  \- Step-by-step procedure to solve a problem

- Components of a problem:

$$\text{Input} \xrightarrow{\text{Algorithms (steps)}} \text{Output}$$

Computational Problem

- Components of a problem:

Inpu
t

Algorithms
(steps)

Outp
ut

Computational Problem

- Components of a problem:

Input $\xrightarrow{\text{Algorithms (steps)}}$ Output

Input size parameter : n

Computational Problem

- Components of a problem:

Inpu
t

Algorithms
(steps)

Outp
ut

Input size parameter : n

- Ex: Sorting problem
    - Input : unsorted array A of n numbers
    - Output: sorted array A

Computational Problem

- Components of a problem:

Input → Algorithms (steps) → Output

Input size parameter : n

- Ex: Sorting problem
    - Input : unsorted array A of n numbers
    - Output: sorted array A

Input size : $|A| = n$

Complexity Function

- **Time complexity** describes the amount of time an algorithm takes in terms of the amount of input

- **Space complexity** describes the amount of memory (space) an algorithm takes in terms of the amount of input

- **Time Complexity function**: Any function that maps the positive integers to the nonnegative reals. It shows number of basic operations of an algorithm based on the input size (Ex : $f(n) = 5n + 1000$)

Comparing Complexity Functions

EX:

$T_1(n) = n^2$

$+3n +5 \; T_2(n)$

$= 100\, n + 7$

Comparing Complexity Functions

EX
:
$T_1(n) = n^2 +3n +5$

$T_2(n) = 100\,n + 7$

	$T_1(n)$	$T_2(n)$
n = 10	135	1007
n = 100	10305	10007
n = 1000	10^6	10^5
n = 10000	10^8	10^6

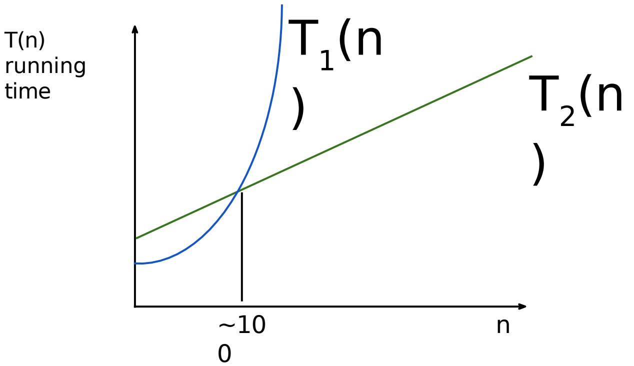
Comparing Complexity Functions

EX:

$T_1(n) = n^2 + 3n + 5$

$T_2(n) = 100\,n + 7$

	$T_1(n)$	$T_2(n)$
n = 10	135	1007
n = 100	10305	10007
n = 1000	10^6	10^5
n = 10000	10^8	10^6
⋮	⋮	⋮



Comparing Complexity Functions

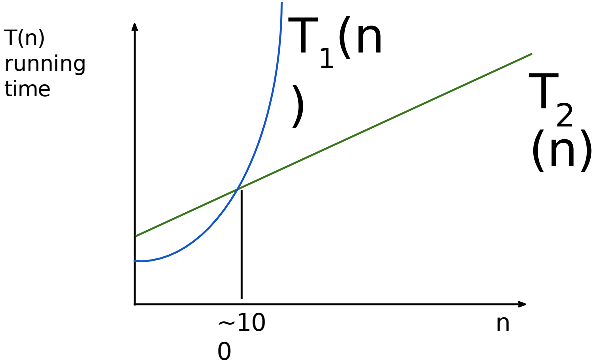
EX:

$T_1(n) = n^2 + 3n + 5$

$T_2(n) = 100\,n + 7$

	$T_1(n)$	$T_2(n)$
n = 10	135	1007
n = 100	10305	10007
n = 1000	10^6	10^5
n = 10000	10^8	10^6

- Why don’t we care about small input size?



Comparing Algorithms Efficiencies

- We are interested in the algorithm’s **asymptotic** complexity

What does asymptotic mean?

- Compare the complexity functions for the large input size!
- when n (number of input items) goes to infinity, what happens to the algorithm’s performance?

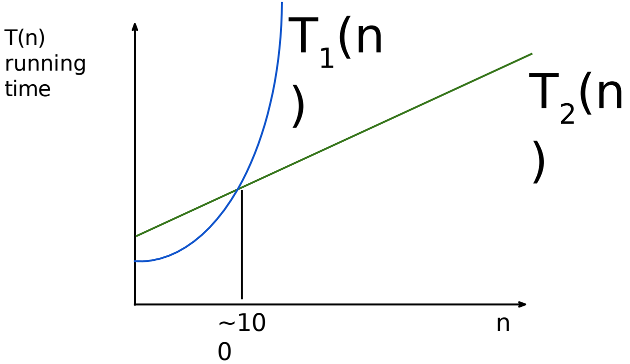
Comparing Complexity Functions

EX:

$T_1(n) = n^2 + 3n + 5$

$T_2(n) = 100\,n + 7$

	$T_1(n)$	$T_2(n)$
n = 10	135	1007
n = 100	10305	10007
n = 1000	10^6	10^5
n = 10000	10^8	10^6
⋮	⋮	⋮



Comparing Complexity Functions

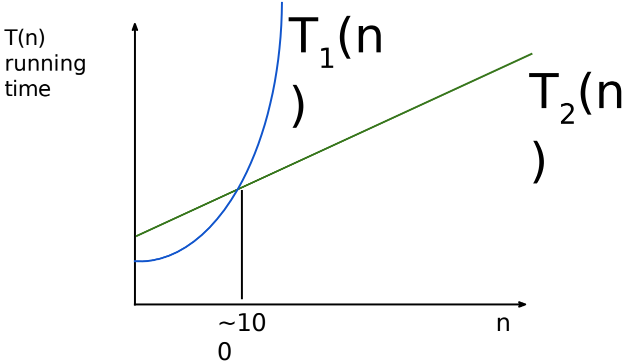
EX:

$T_1(n) = n^2 + 3n + 5$ — Dominating term

$T_2(n) = 100\,n + 7$ — Dominating term

	$T_1(n)$	$T_2(n)$
n = 10	135	1007
n = 100	10305	10007
n = 1000	10^6	10^5
n = 10000	10^8	10^6
⋮	⋮	⋮

When n goes to **infinity**, **dominating** terms have the most contribution to the value of the complexities functions



Comparing Complexity Functions

EX:

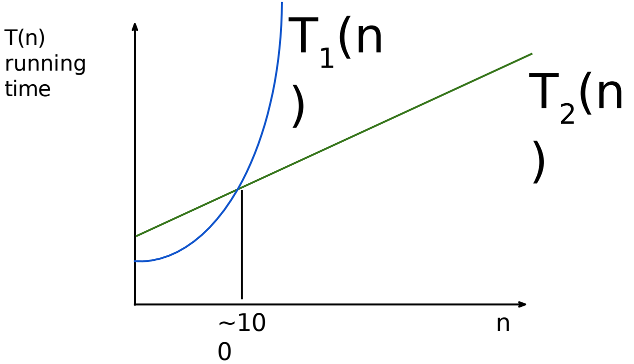
$T_1(n) = n^2 + 3n + 5$

Dominating term

$T_2(n) = 100\,n + 7$

Dominating term

	$T_1(n)$	$T_2(n)$
n = 10	135	1007
n = 100	10305	10007
n = 1000	10^6	10^5
n = 10000	10^8	10^6
⋮	⋮	⋮



When n goes to **infinity**, **dominating** terms have the most contribution to the value of the complexities functions

$T_1(n) \in O(n^2)$

Big-O Notation (*Asymptotic Upper Bounds)*

- The most commonly used notation for asymptotic complexity used is "big-O"
- Big-O notation can be described as an upper bound for a complexity function
- In the previous example we would say $n^2 + 3n + 5 = O(n^2)$
- Knowing where a function lies within the big-O class of functions lets us compare it quickly with other functions
- Thus we have an idea of which algorithm has the best time performance

Big-O Notation

- Ex:
- $f(n) = n^2 + 5n + 1000$
- $f(n) = n^3 + 40n^2 + n \log n + 2$
- $f(n) = n^{1.999} + 100n$
- $f(n) = n\log n + n$

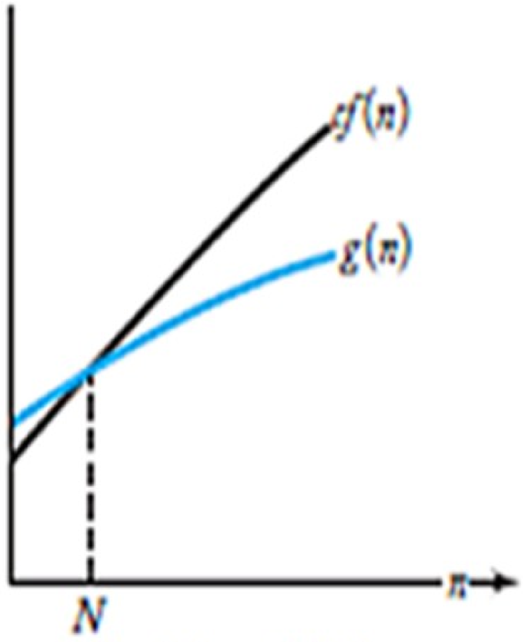
Big O

Let $T(n)$ be a function. Given another function $f(n)$, we say that $T(n)$ *is* $O(f(n))$ (read as “$T(n)$ is order $f(n)$”) if, for sufficiently large n, the function $T(n)$ is bounded above by a constant multiple of $f(n)$. We will also sometimes write this as $T(n) = O(f(n))$.

$T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ so that for all $n \geq n_0$,
we have $T(n) \leq c \cdot f(n)$. In this case, we will say that T is *asymptotically upper bounded by f*. It is important to note that this definition requires a constant
to exist that works for *all n*; in particular, c cannot depend on n.

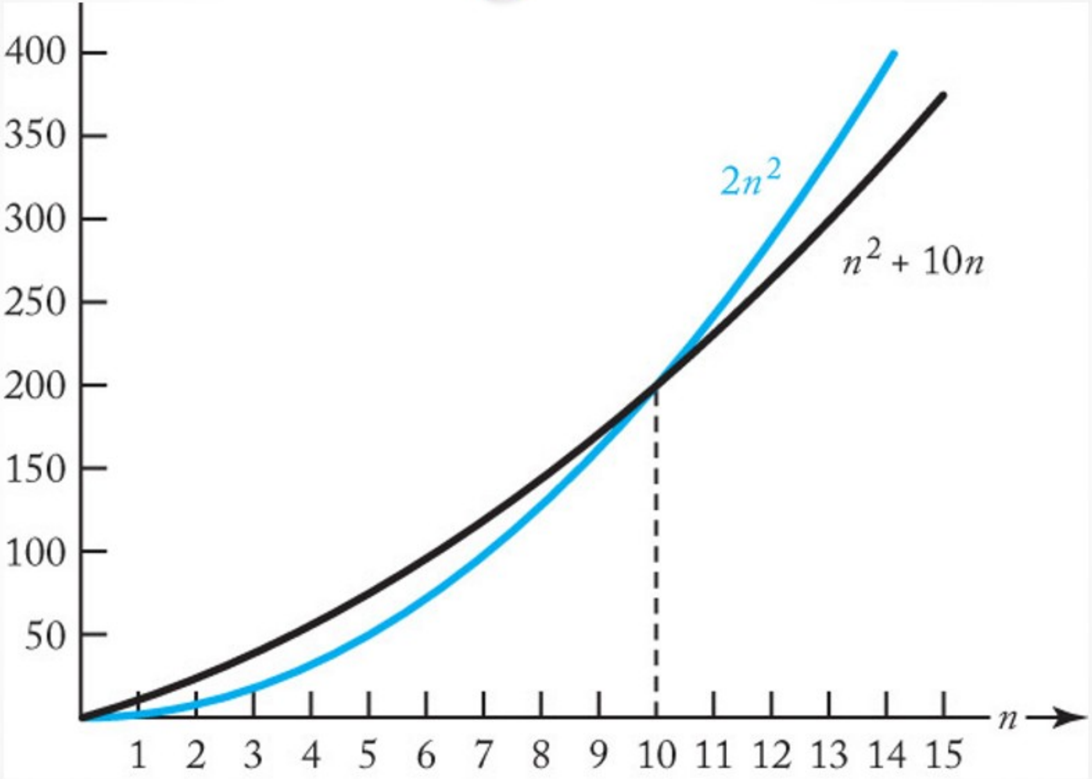
Big O

- For a given complexity function f (n), O(f(n)) is **the set of complexity functions g(n)** for which there exists **some positive real constant** c and some nonnegative integer N such that for all $n \geq N$,
- $g(n) \leq c \times f(n)$
- $g(n) \in O(f(n))$
- "big O" puts an asymptotic upper bound on a function.



(a) $g(n) \in O(f(n))$

Finding c and N



c = 2 and N = 10 in the definition of "big O"

Big-O Notation

- Ex:
- $f(n) = n^2 + 5n + 1000 \in O(n^2)$
- $f(n) = n^3 + 40n^2 + n \log n + 2 \in O(n^3)$
- $f(n) = n^{1.999} + 100n \in O(n^2)$
- $f(n) = n\log n + n \in O(n \log n)$

$n^2 + n$, $4n^2 - n \log n + 12$, $n^2/5 - 100n$, $n \log n$, and so forth are all $O(n^2)$

Exercise

$T(n) = pn^2 + qn + r$ for positive constants p, q, and r.

Show that $T(n)$ is in $O(n^2)$

Exercise

$T(n) = pn^2 + qn + r$ for positive constants p, q, and r.

Show that $T(n)$ is in $O(n^2)$

Solution

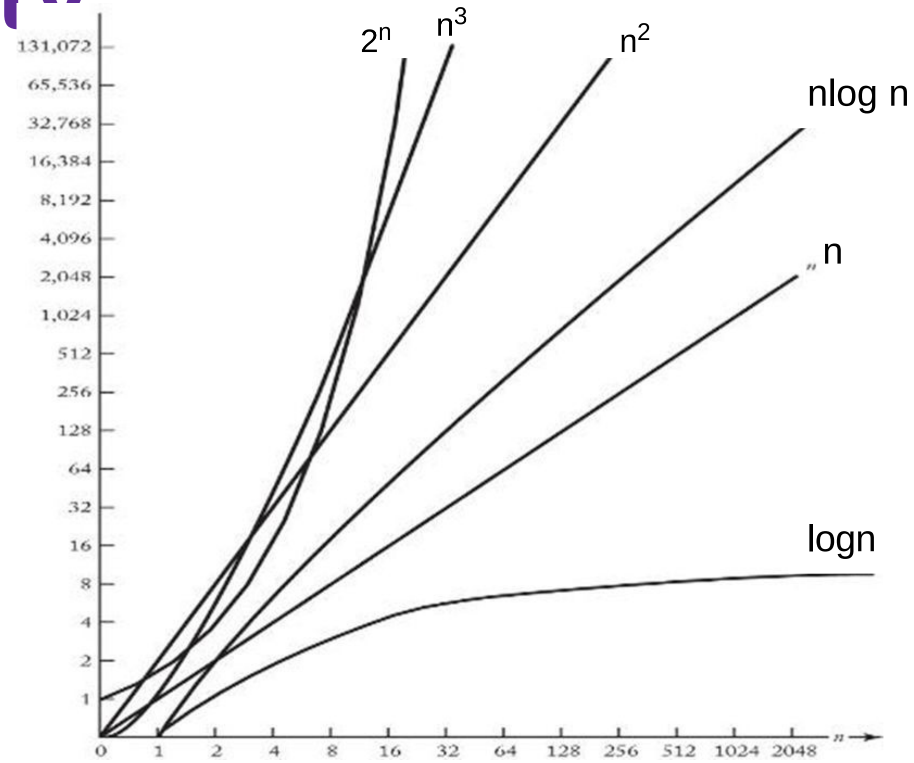
We have $qn \leq qn^2$, and $r \leq rn^2$. So we can write

$$T(n) = pn^2 + qn + r \leq pn^2 + qn^2 + rn^2 = (p + q + r)n2$$

for all $n \geq 1$. This inequality is exactly what the definition of $O(\cdot)$ requires:

$T(n) \leq cn2$, where $c = p + q + r$.

Growth Rates of Common Complexity Functions



Big-O Notation

- Definition: Given a function $g(n)$, we denote $O(g(n))$ to be the set of functions

{ $f(n)$ | there exists positive constants **c** and $\mathbf{n_0}$ such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$}

Rough Meaning: $O(g(n))$ includes all functions that are upper bounded by $g(n)$

EX: $4n = O(n)$ $n_0 = 1$,

Note

Note that $O(\cdot)$ expresses only an upper bound, not the exact growth rate of the function. For example, just as we claimed that the function $T(n) = pn2 + qn + r$ is $O(n2)$, it's also correct to say that it's $O(n3)$.

Big-Ω Notation *Asymptotic Lower Bounds*

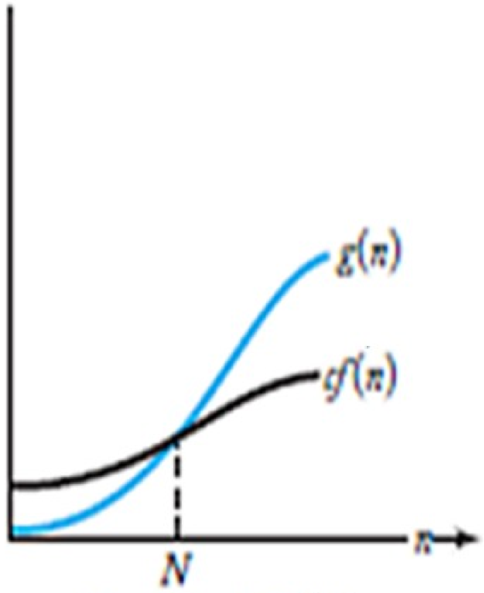
Let's say we have just proven that its worst-case running time $T(n)$ is $O(n^2)$
we want to show that this upper bound is the best one possible.

To do this, we want to express the notion that for arbitrarily large input sizes n, the function $T(n)$ is *at least* a constant multiple of some specific function $f(n)$.

$T(n)$ is $\Omega(f(n))$ (also written $T(n) = \Omega(f(n))$) if there

Omega

- For a given complexity function f(n), Ω(f (n)) is **the set of complexity functions g (n)** for which there exists some positive real constant c and some nonnegative integer N such that, for all n ≥ N,
- $g(n) \geq c \times f(n)$
- $g(n) \in \Omega(f(n))$
- "Omega" puts an asymptotic lower bound on a function.



(b) $g(n) \in \Omega(f(n))$

Big-Ω Notation

- Definition: Given a function $g(n)$, we denote $\Omega(g(n))$ to be the set of functions

$\{ f(n) \mid$ there exists positive constants $\mathbf{c}$ and $\mathbf{n_0}$ such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0 \}$

Rough Meaning: $\Omega(g(n))$ includes all functions that are lower bounded by $g(n)$

EX: $2n = \Omega(n)$

Exercise

$T(n) = pn^2 + qn + r$, where p, q, and r. Is $T(n) = \Omega(n^2)$?

Show that for n>=that a given integer n_0, what about for $n_0 = 0$?

$pn^2 + qn + r >= cn^2$

Big-Θ Notation *Asymptotically Tight Bounds*

If we can show that a running time $T(n)$ is both $O(f(n))$ and also $\Omega(f(n))$, then in a natural sense we've found the "right" bound $T(n)$ grows exactly like $f(n)$ to within a constant factor.

This, for example, is the conclusion we can draw from the fact that $T(n) = pn2 + qn + r$
is both $O(n^2)$ and $\Omega(n^2)$.

Big-Θ Notation *Asymptotically Tight Bounds*

- Definition: Given a function $g(n)$, we denote $\Theta(g(n))$ to be the set of functions

{ $f(n)$ | there exists positive constants c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$ }

$f(n) = \Theta(g(n))$ $\Longleftrightarrow$ $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$

Meaning: Those functions which can be both upper bounded and lower bounded by g(n)

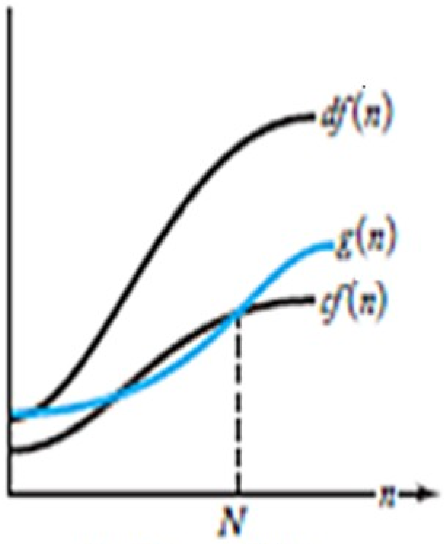
EX: $4n = \Theta(n)$

Properties of Big-O Notation

- Transitive: if $f(n) = O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n) = O(h(n))$
- If $f(n) = O(h(n))$ and $g(n)$ is $O(h(n))$, then $f(n) + g(n) = O(h(n))$
- A function $an^k = O(n^k)$ for any $a > 0$
- Any kth degree polynomial is $O(n^{k+j})$ for any $j > 0$
- $\log_a n = O(\log_b n)$ for any $a, b > 1$. we don't care what base our logarithms are

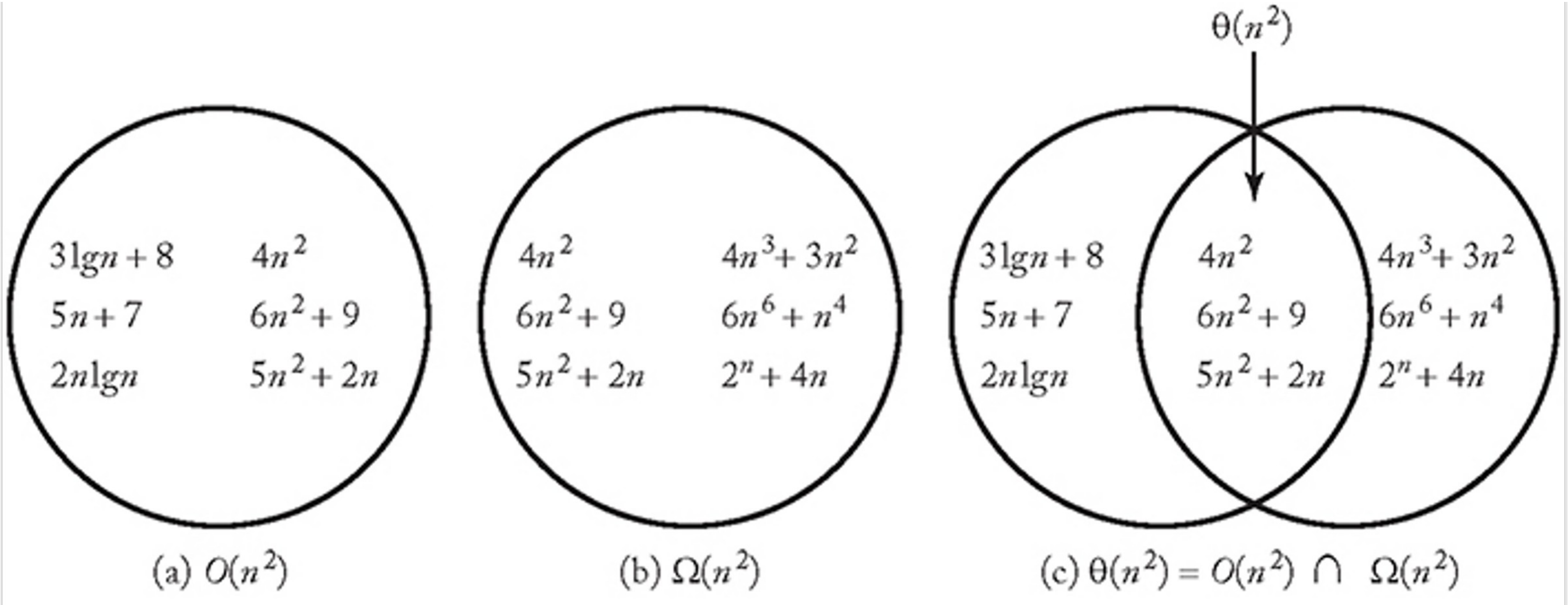
Theta

- For a given complexity function f(n), $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- This means that $\Theta(f(n))$ is **the set of complexity functions** g(n) for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$,
- $c \times f(n) \leq g(n) \leq d \times f(n)$.
- $g(n) \in \Theta(f(n))$



(c) $g(n) \in \Theta(f(n))$

Examples of O(),Ω (), Θ ()



Properties of Common Growth Functions

- An algorithm that runs in polynomial time is (eventually) preferable to an algorithm that runs in exponential time

- An algorithm that runs in logarithmic time is (eventually) preferable to an algorithm that runs in polynomial (or from above, exponential) time

- Some of important growth functions in order: $n!$, 2^n, n^3, n^2, $n\log n$, n, $\log n$, 1

Finding Asymptotic Complexity of An Algorithm

1. Find the basic operations (such as assignments, comparisons, etc.) in the program

2. Compute the total number of basic operations in the program

3. Find the dominating term

4. Represent with big O notation

Finding Asymptotic Complexity of An Algorithm

- Ex:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

Finding Asymptotic Complexity of An Algorithm

- Ex:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

1 2 3 4 5

Finding Asymptotic Complexity of An Algorithm

- Ex:

1

```
sum = 0;
```

3

4

2

```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

5

1: assignment operation (=)

Number of executions

1 time

Finding Asymptotic Complexity of An Algorithm

- Ex
  :

1

```
sum = 0;
```

3

4

2

```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

5

1: assignment operation (=) 2: assignment operation (=)

Number of executions

1 time
1 time

Finding Asymptotic Complexity of An Algorithm

- Ex
  :

1

```
sum = 0;
```

3

4

2

```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

5

1: assignment operation (=) 2: assignment operation (=) 3: comparison operation(<)

Number of executions

1 time
1 time
n+1 times

Finding Asymptotic Complexity of An Algorithm

- Ex
  :

1

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

2 3 4 5

1: assignment operation (=) 2: assignment operation (=) 3: comparison operation(<)
4: increment and assignment (++)

Number of executions

1 time
1 time
n+1 times n times

Finding Asymptotic Complexity of An Algorithm

- Ex:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

1 2 3 4 5

	Number of executions
1: assignment operation (=)	1 time
2: assignment operation (=)	1 time
3: comparison operation(<)	n+1 times
4: increment and assignment (++)	n times
5: assignment and addition	n times

Finding Asymptotic Complexity of An Algorithm

- Ex
  :

1

```
sum = 0;
```

3

4

2

```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

5

1: assignment operation (=) 2: assignment operation (=) 3: comparison operation(<)
4: increment and assignment (++) 5: assignment and addition

Number of executions

1 time
1 time
n+1 times n
times
n times

Total number of executions

3n + 3

Finding Asymptotic Complexity of An Algorithm

- Ex
:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

1 2 3 4 5

1: assignment operation (=) 2: assignment operation (=) 3: comparison operation(<)
4: increment and assignment (++) 5: assignment and addition

Number of executions

1 time
1 time
n+1 times
n times
n times

Total number of executions

3n + 3 = O(n)

Finding Asymptotic Complexity of An Algorithm

- Ex:

```
                 1
sum = 0;
                3
for (i = 0; i < n; i++)
2              4
    sum = sum + a[i];    5
```

1: assignment operation (=) 2: assignment operation (=) 3: comparison operation(<)
4: increment and assignment (++) 5: assignment and addition

Number of executions

1 time
1 time
n+1 times n times
n times

Total number of executions

3n + 3 = O(n)

We don’t care about the exact number of operations. We just care about the complexity of algorithms in terms of big O notation

Finding Asymptotic Complexity of An Algorithm

- Ex:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

Consider the operations inside the for loop only which are proportional to the size of the input array (n)
Other operations take constant time

How many times the for loop will be executed?

Finding Asymptotic Complexity of An Algorithm

- Ex:

```
sum = 0;
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

Consider the operations inside the for loop only which are proportional to the size of the input array (n)
Other operations take constant time

How many times the for loop will be executed? O(n)

Worse Case, Best Case, Average Case

- By Default, we analyze the algorithms for the worse case scenario.

- In worse case analysis, we consider the input parameters

  such that our algorithm takes the most number of

  operations to run

Ex: Sequential search

Input: An array of size n, and a value x

Output: return the index of x, return -1 if x does not exist

Worse Case

- By Default, we analyze the algorithms for the worse case scenario.

- In worse case analysis, we consider the input parameters

  such that our algorithm takes the most number of

  operations to run

Ex: Sequential search

Input: An array A of size n, and a value x

Output: return the index of x, return -1 if x does not exist

Best Case

- In best case analysis, we consider the case such that our algorithm takes the

  least number of operations to run

Ex: Sequential search

Input: An array A of size n, and a value x

Output: return the index of x, return -1 if x does not exist

Best Case

- In best case analysis, we consider the case such that our algorithm takes the

  least number of operations to run

Ex: Sequential search

Input: An array A of size n, and a value x

Output: return the index of x, return -1 if x does not exist

Best Case: is when x exists in A[0] (Only one step) → O(1)

Average Case

- In average case analysis, we consider the average number of steps for running

  the algorithm

Ex: Sequential search

Input: An array A of size n, and a value x

Output: return the index of x, return -1 if x does not exist

Average Case

- In average case analysis, we consider the average number of steps for running

  the algorithm

Ex: Sequential search

Input: An array A of size n, and a value x

Output: return the index of x, return -1 if x does not exist

we would find the target in the first location with $p = 1/n$, in the second location with $p = 1/n$, etc.

Since the number of steps required to get to each location is the same as the location

itself, our sum becomes: $1/n * (1 + 2 + \ldots + n) = (n + 1) / 2$