RMIT Classification: Trusted

Algorithms Analysis and Design

RMIT UNIVERSITY

Learning objectives

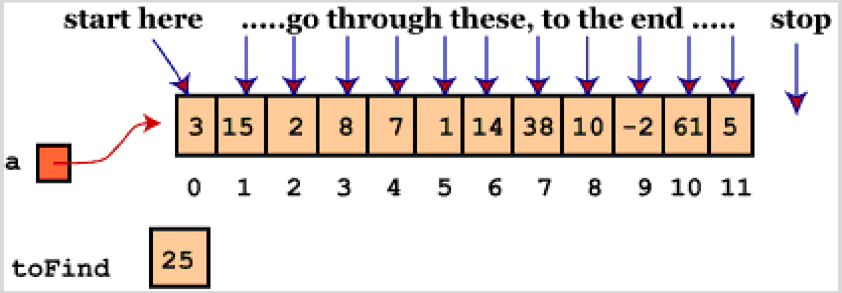
- Understand why it is important to be able to compare the complexity of algorithms
- Measure the complexity of algorithms
- Analyse the performance of algorithms
- Be able to perform empirical analyses of algorithms

Example: Which is Faster?

- **Scenario:** You work for a large medical company. Your boss comes and asks your opinion on which of two approaches is faster for employees to search over an unordered set of medicines.
- **Question:** Which of the following two approaches will you tell your boss is faster?
    1. Search one by one
    2. Sort then search

Array Search

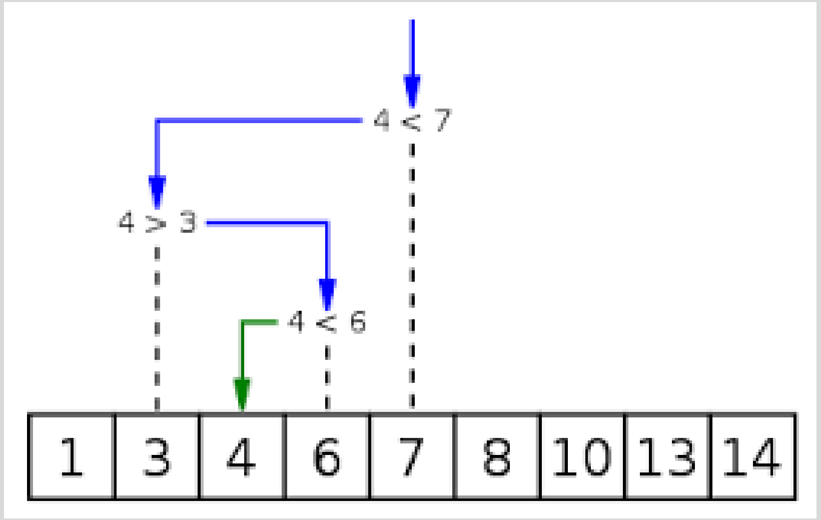
Items	Average number of tries
10	5
100	50
1000	500
10,000	5000
100,000	50,000
1,000,000	500,000
10,000,000	5,000,000



For a non-sorted array of size N, the number of tries to find a value is correlated to N

Binary Search

Items	Tries
10	4
100	7
1000	10
10,000	14
100,000	17
1,000,000	20
10,000,000	24



For a sorted array of size N, the number of tries to find a value is correlated to $\log_2 N$

Sorting

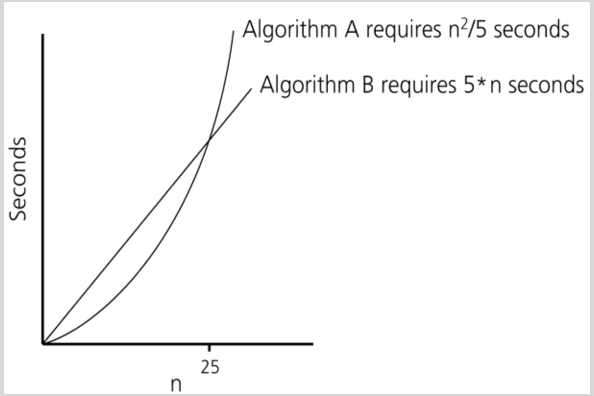
- This is a very interesting topic that we will cover extensively in class as many problems require using or generating sorted data
- But to answer the question on the medical company we need to be able to determine how fast sorting can be done

Which is Faster?

- Assume that linear search takes 2*n msec, sorting takes 10*n*log(n) msec and binary search takes 2*log(n) msec
- If you do the search 10 times on 10^6 which approach is best?
  1. $10*2*10^6 = 20*10^6$ msec = **20,000 sec**
  2. Sorting: $10*10^6*\log10^6 \approx 10*10^6*13.8$ msec ≈ **138,000 sec**
     Searching: $10*2*\log10^6 =$ **0.276 sec**
     Total ≈ **138,000 sec**
- If you do the search 1000 times and the data size is 10^6 then
  1. $1000*2*10^6 = 2000*10^6$ msec = **2,000,000** sec
  2. Sorting: $10 * 10^6 * \log 10^6 \approx 10*10^6 *13.8$ msec ≈ **138,000 sec**
     Searching: $1000*2*\log 10^6 =$ **27.6 sec**
     Total ≈ **138,027 sec**

Program Performance

- Assume that we can find a formula that, for a specific machine, determines how long a program takes to run on various input (N) sizes



- When do programs A and B become impractical?
  - If n = 10 then A takes 20 and B takes 50 secs
  - If n = 25 they both take 125 secs.
  - If n = 10^3 then A takes 200,000 secs and B takes 5,000 secs.
  - If n = 10^6 then A takes $2x10^{11}$ seconds and B take $5x10^6$ seconds
- **Algorithmic performance is critical when analyzing large data sets**

Comparing Performance

- Different algorithms that solve the same problem can have very different performance
  - How do we compare the performance of two programs that solve the same problem?
  - Should we just run them and compare their time requirements?
- Need address issues such as - what computer to use, what data to use and how much space is needed?
- An inefficient algorithm on a small data set (i.e., one that grows quadratically) would be unfeasible to run on a large data set

What to measure?

- In this lecture, we look at the ways of estimating the running time of a program and how to compare the running times of two programs **without ever implementing them**
- It is vital to analyse the resource use of an algorithm, well before it is implemented and deployed
- **Space** is also important, but we focus more on **time** in this course

RMIT Classification: Trusted

Time Efficiency

RMIT UNIVERSITY

How to Estimate Running Time?

- We employ mathematical techniques that analyze algorithms independently of specific implementations, computers or data
- Approach:
    - First, we start to count the number of significant operations in a particular solution to assess its efficiency
    - Then, we will express the efficiency of algorithms using growth functions

Theoretical Analysis of Time Efficiency

- **Idea:** An algorithm consists of some **operations** executed a number of times.
- Hence, an **estimate of the running time/time efficiency** of an algorithm can be obtained from
  - Determining these **operations**
  - **How long** to execute them, and
  - The **number of times** they are executed.
- These operations are called **basic operations,** and the number of times is based on the **input size** of the problem

Basic operation

Operation(s) that contribute most towards the total running time

- Examples:
  - Compare (i != j)
  - Add (i + j)
  - Multiply (i * j)
  - Divide (i / j)
  - Assignment (i = j)

Execution Time of Algorithms - 1

- Each operation in an algorithm (or a program) has a cost
    - count = count + 1; takes a certain amount of time, but it is constant
- Therefore, for a sequence of operations*:*

count = count + 1	Cost: c_1
sum = sum + count	Cost: c_2

Total Cost = $c_1 + c_2$

Execution Time of Algorithms - 2

Example: Simple If-Statement

	Cost	Occurrences
if (n < 0):	c1	1
absval = -n	c2	1
else:		
absval = n	c3	1

Total Cost <= c1 + max(c2,c3)

Execution Time of Algorithms - 3

Example*: *Simple Loop

	Cost	Occurrences
`i = 1`	c1	1
`sum = 0`	c2	1
`while (i <= n):`	c3	n+1
`    i = i + 1`	c4	n
`    sum = sum + i`	c5	n

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*c5

Execution Time of Algorithms - 4

Example: Nested Loop

	Cost	Occurrences
i = 1	c1	1
sum = 0	c2	1
while (i <= n):	c3	n+1
j=1	c4	n
while (j <= n):	c5	n*(n+1)
sum = sum + i	c6	n*n
j = j + 1	c7	n*n
i = i +1	c8	n

Total Cost = c1 + c2 + (n+1)*c3 + n*c4 + n*(n+1)*c5+n*n*c6+n*n*c7+n*c8

General Rules for Estimation

- **Consecutive Statements:** Just add the running times of those consecutive statements.
- **If/Else:** Never more than the running time of the test plus the larger of running times of S1 and S2.
- **Loops**: The running time of a loop is at most the running time of the statements inside of that loop times the number of iterations.
- **Nested Loops**: Running time of a nested loop containing a statement in the inner most loop is the running time of statement multiplied by the product of the sized of all loops.

Runtime Complexity

- **Worst Case** - Given an input of n items, what is the **maximum** running time for any possible input?
- **Best Case** - Given an input of n items, what is the **minimum** running time for any possible input?
- **Average Case** - Given an input of n items, what is the **average** running time across all possible inputs?

NOTE: **Average Case** is not the average of the worst and best case. Rather, it is the average performance across all possible inputs.

Runtime Complexity

- **Sequential Search: search for the key by traversing values in the array one-by-one**
- **Best-case:**
  - The best case input is when the item being searched for is the first item in the list, so Cbest (n) = 1.
- **Worst-case:**
  - The worst case input is when the item being searched for is not present in the list, so Cworst (n) = n.

Runtime Complexity

- **Average Case:** What does average case mean?
  - Recall: average across all possible inputs – how to analyse this?
  - Typically not straight forward

Runtime Complexity

Average Case Analysis: *p* is the probability of a successful search.

If search is successful

Cavg(n) = (1 + 2 + … + n)/n = (n + 1)/2

(Assume the probability of finding the search value at each element is the same)

If search is unsuccessful

Cavg(n) = n

Summary

- Input size, basic operation
- Time complexity estimate using input size and basic operation
- Best, worst, and average cases

RMIT Classification: Trusted

Asymptotic Complexity

Asymptotic Complexity

- **Problem:**
  - We now have a way to analyse the running time (a.k.a. **time complexity**) of an algorithm, but every algorithm has their own time complexity.
    - $T_1 = c_1 \cdot C(n_1), \quad T_2 = c_2 \cdot C(n_2), \quad T_3 = c_3 \cdot C(n_3)$ ...
  - How to **compare** in a meaningful way?

Asymptotic Complexity

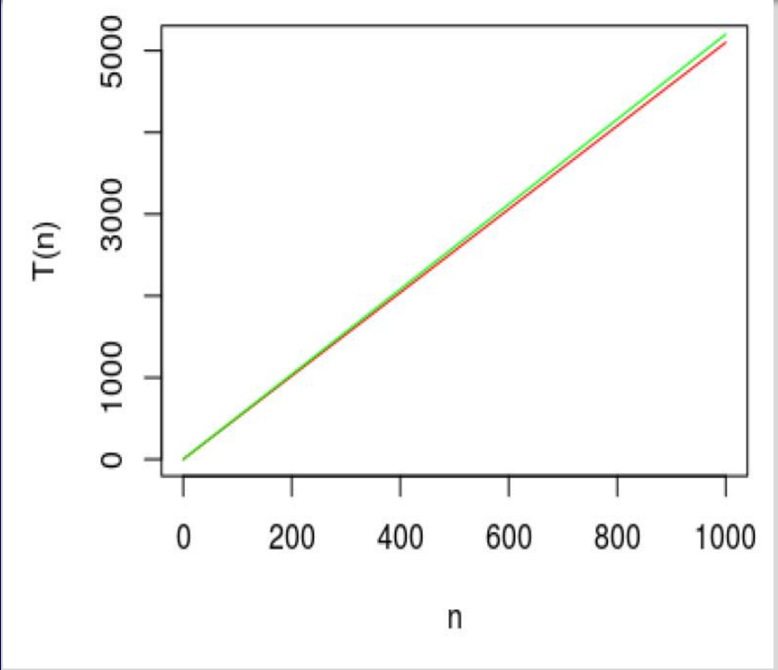
- **Solution:**
    - Group them into **equivalence classes** (for easier comparison and understanding), with respect to the input size
    - Focus of this part: **asymptotic complexity** and **equivalence classes**

Asymptotic Complexity

Consider the running time estimates of two algorithms:

- Algo 1: $T_1(n) = 5.1n$
- Algo 2: $T_2(n) = 5.2n$

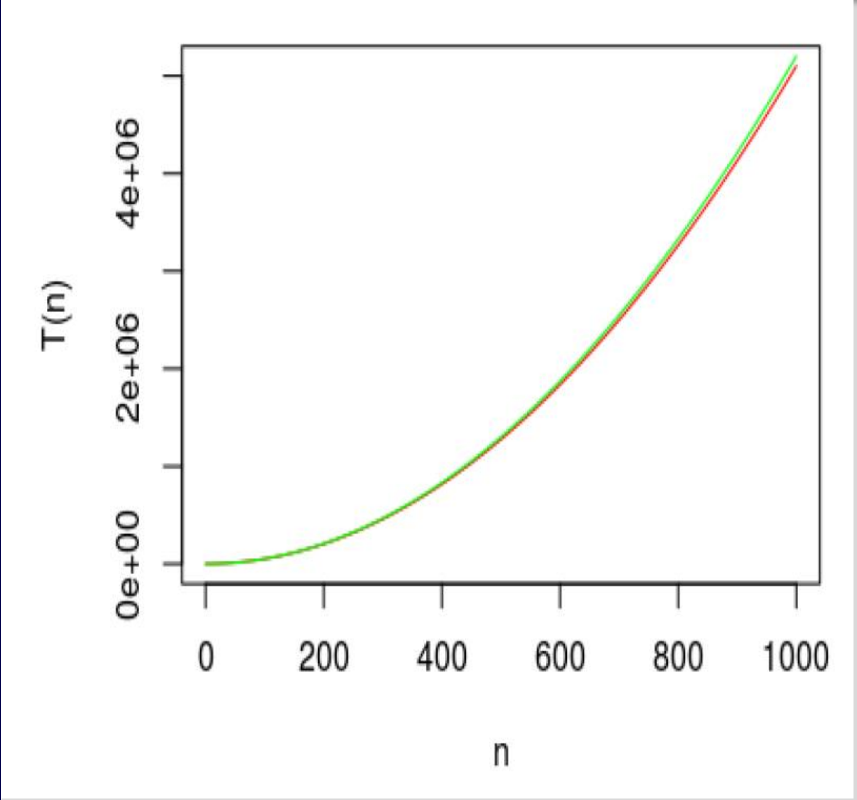
Do they have **similar** timing profiles as *n* grows?



Asymptotic Complexity

What about the followings:

- Algo 3: $T_3(n) = 5.1n^2$
- Algo 4: $T_4(n) = 5.2n^2$



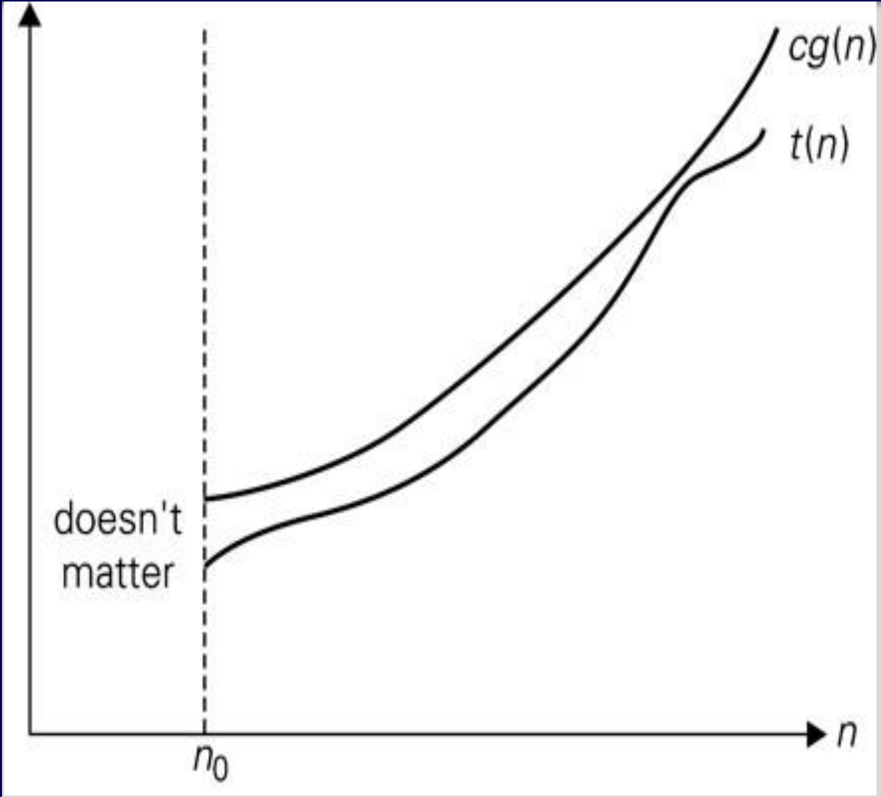
Asymptotic Complexity - Bounds

- **Idea:** Use **bounds** and asymptotic complexity
- In other words:
  - As *n* becomes large, what is the **dominant term** contributing to the running time $t(n)$?

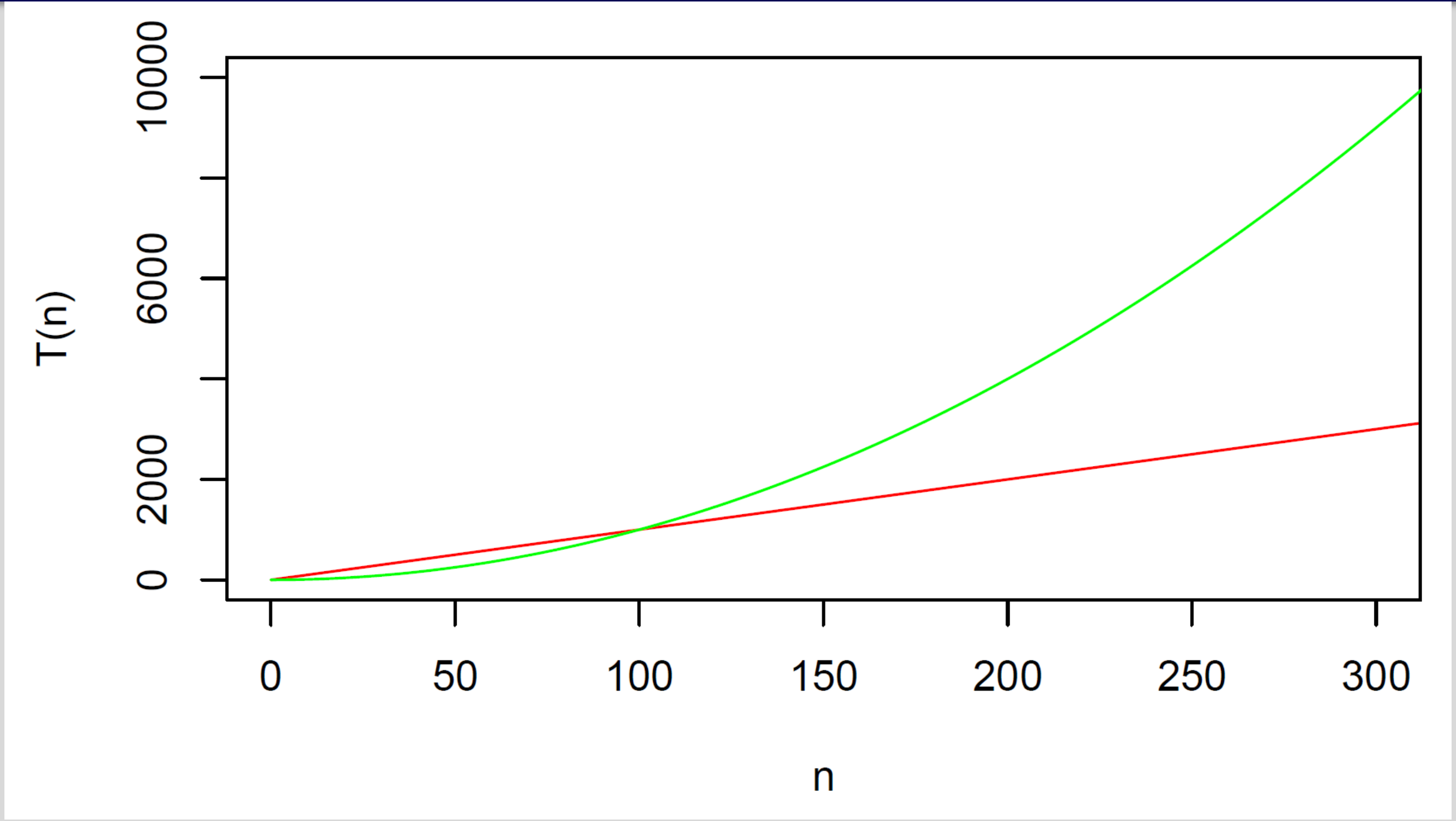
Asymptotic Complexity - Upper Bounds

Definition:

- Given a function $t(n)$ (i.e., the running time of an algorithm):
- Let $\boldsymbol{c \cdot g(n)}$ be a function that is an **upper** bound on $t(n)$ for some $c > 0$ and for *"large"* n.



Example: Upper Bounds



Asymptotic Complexity – O(*n*)

Rather than talking about individual upper bounds for each running time function, we seek to group them into **equivalence classes** that describe their **order of growth**.

Big-O notation: O(*n*)

Given a function $t(n)$,

- **Formally:** $t(n) \in O(g(n))$, if $g(n)$ is a function and $c \cdot g(n)$ is an upper bound on $t(n)$ for some $c > 0$ and for *"large"* n.
- **Informally:** $t(n) \in O(g(n))$ means $g(n)$ is a function that, as n increases, provides an upper bound for $t(n)$.

Asymptotic Complexity – O(*n*)

If $\boldsymbol{t(n) = 5.1n}$

- $g1(n) = n$
- $g2(n) = 0.001n - 6$
- $g3(n) = n^2$

What $\boldsymbol{g(n)}$ to use if $\boldsymbol{t(n) = 5.2n}$?

- Any of the above $g(n)$ functions are possible!

Asymptotic Complexity

If an algorithm requires $n^2 – 3*n+10$ seconds to solve a problem of size n. If constants c and n_0 exist such that

$$c*n^2 > n^2 – 3*n+10 \quad \text{for all } n \geq n_0$$

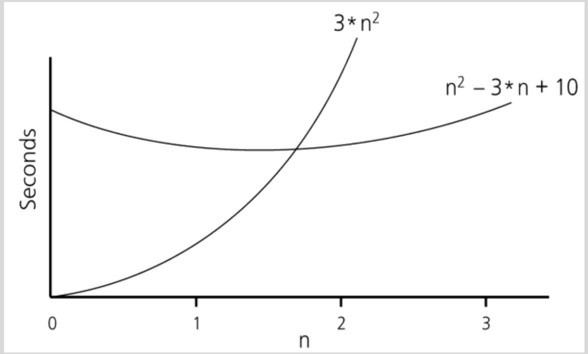
the algorithm is order n^2 (In fact, c is 3 and n_0 is 2)

$$3*n^2 > n^2 – 3*n+10 \quad \text{for all } n \geq 2$$

Thus, the algorithm requires no more than

$c*n^2$ time units for $n \geq n_0$ where $c = 3$ and $n_o = 2$

So, it is **O(n²)**

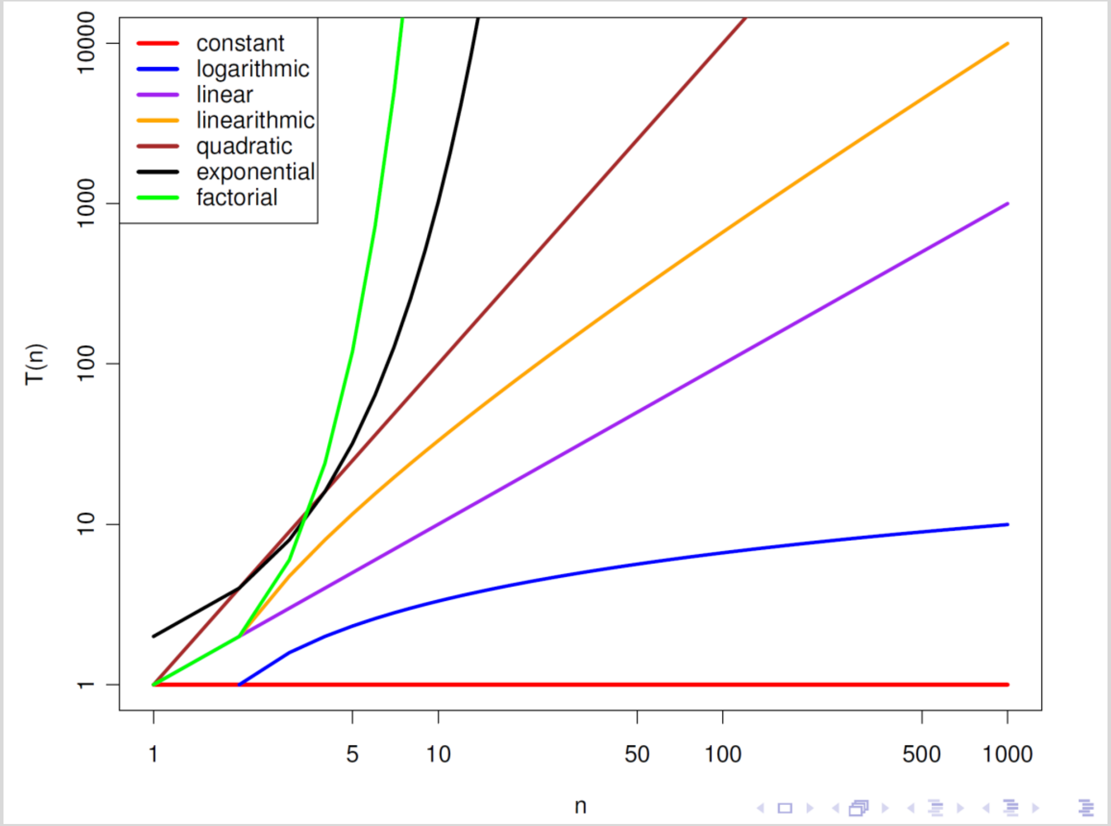


Common Equivalence Classes

$O(1)$	Constant (access array element)
$O(\log_2 n)$	Logarithmic (binary search)
$O(n)$	Linear (finding maximum in an array)
$O(n\log_2 n)$	Linearithmic (best sorting algorithms such as merge sort)
$O(n^2)$	Quadratic (simple sorting algorithms)
$O(n^3)$	Cubic (matrix multiplication)
NP	Non-deterministic polynomial (travelling salesman, zero-sum subset)
$O(2^n)$	Exponential (generating all subsets, solve the tower of Hanoi best known algorithms for travelling salesman and integer factorization)
$O(N!)$	Factorial – generating all permutations of N elements

List sorted from best to worst. Many other options e.g. sqrt(n), log(log(n)), etc.

Common Complexity Bounds



Common Complexity Bounds

Recall: We want to find ***equivalence function class*** that upper bounds different $t(n)$.

But we might be given an upper bound $g(n)$ that isn't quite in the form of the equivalence classes.

- In the previous slides, we mentioned multiple upper bounds, e.g., n, $0.001n - 6$, n^2

How to get the ***equivalence classes***?

Growth-Rate Functions

- Ignore low-order terms in the growth-rate function
  - If an algorithm is $O(n^3+4n^2+3n+5)$, it is also $O(n^3)$
- Ignore a multiplicative constant in the higher-order term
  - If an algorithm is $O(5n^3)$, it is also $O(n^3)$
- Combine two modules with growth rates $O(f(n))$ and $O(g(n))$ is $O(f(n)+g(n))$
  - If an algorithm is $O(n^3) + O(4n)$, it is also $O(n^3 +4n)$, so, it is $O(n^3)$
- We can also multiply two growth rates e.g. there is a loop inside a loop
  - So, a $O(n^2)$ loop inside $O(n)$ loop is $O(n^3)$

Common Complexity Bounds

Let $g(n) = 2n^4 + 43n^3 - n + 50$

be an **upper** bound for the running time $t(n)$ of an algorithm.

What **equivalence class** should you use to describe the **order of growth** of the algorithm?

Answer: O(n^4)

RMIT Classification: Trusted

Analysis of Algorithms

RMIT UNIVERSITY

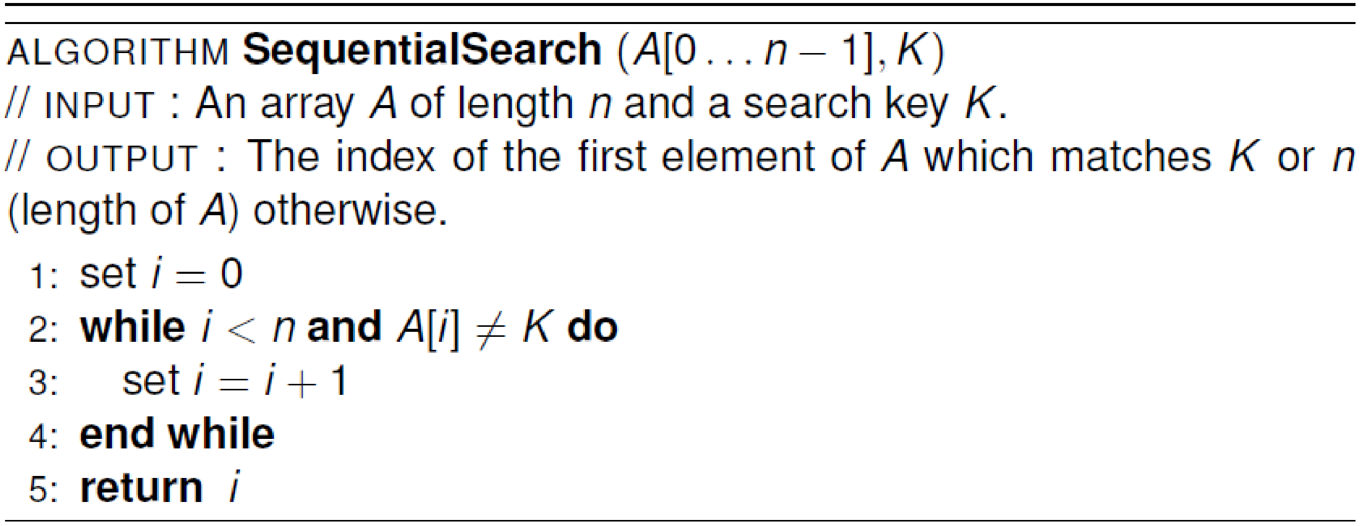
Time Efficiency of Algorithms

Typically, we are given the pseudo code of an algorithm, not a nice $t(n)$ function.

So how do we determine **bounds** on the **order of growth** of an algorithm?

Example: Sequential Search

Basic operations	comparison, addition, assignment
Input size	n



```
ALGORITHM SequentialSearch ($A[0 \dots n-1], K$)
// INPUT : An array $A$ of length $n$ and a search key $K$.
// OUTPUT : The index of the first element of $A$ which matches $K$ or $n$
(length of $A$) otherwise.
set $i = 0$
while $i < n$ and $A[i] \neq K$ do
    set $i = i + 1$
end while
return $i$
```

Example: a^n

```
// INPUT : $a, n$
// OUTPUT : $s = a^n$
1: set $s = 1$
2: for $i = 1$ to $n$ do
3:     $s = s * a$
4: end for
5: return
```

Input size: *n*

Basic operation: *multiplication*

C(*n*): $\underbrace{1 + 1 + 1 + \ldots + 1}_{n} = \sum_{i=1}^{n} 1$

= n

Example: Adding Matrices

Given two (square) matrices *A* and *B*, both of dimensions *n* by *n*, the following algorithm computes *C* = *A* + *B*.

```
for (int i = 0; i <= n−1; i++)  {
        for (int j = 0; j <= n−1; j++) {
                C[i, j] = A[i, j] + B[i, j];
        }
}
```

Input size: *n*

Basic op.: *addition*

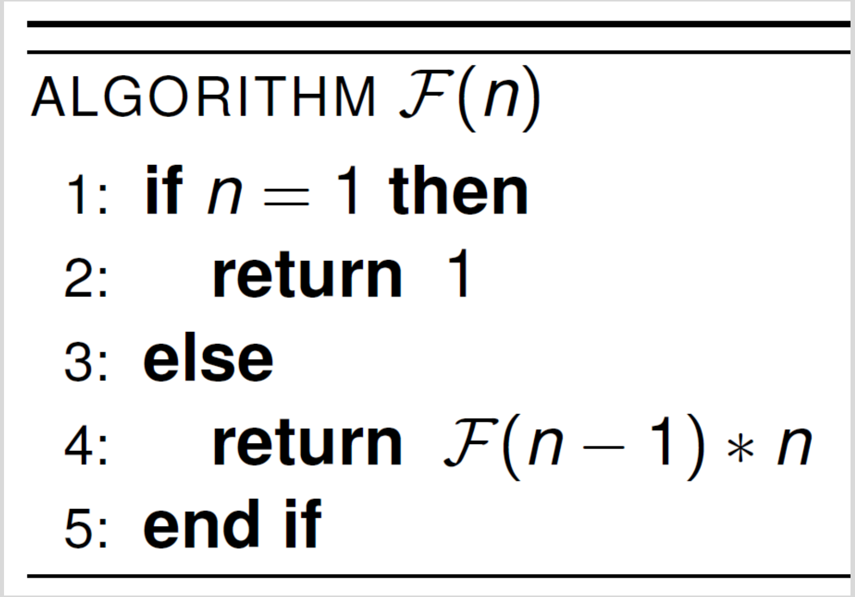
C(*n*): $\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$

= n^2

Recursion

- Recursion is fundamental tool in computer science.
  - A recursive program (or function) is one that calls itself.
  - It must have a termination condition defined.
- Many interesting algorithms are simply expressed with a recursive approach

Recursive Example: Factorial



```
ALGORITHM $\mathcal{F}(n)$
if $n = 1$ then
    return 1
else
    return $\mathcal{F}(n-1) * n$
end if
```

Input size: *n*

Basic operation: *multiplication*

Recursive Example: Factorial

```
ALGORITHM $\mathcal{F}(n)$
if $n = 1$ then
    return 1
else
    return $\mathcal{F}(n-1) * n$
end if
```

Recurrence relation and conditions:

The number of multiplications is

$C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

- +1 is the number of multiplication operations at each recursive step.
- When $n = 1$, we have our termination/base case, where we stop the recursion. When we reach this base case, the number of multiplications is 0, hence $C(1) = 0$.

Backward Substitution

Recurrence: $C(n) = C(n-1) + 1$ for $n > 1$, and $C(1) = 0$

Aim of simplification and backward substitution:

Convert $C(n) = C(n-1) + 1$ to $C(n) = function(n)$

For example, $C(n) = n + 1$

Backward Substitution

Aim of simplification and backward substitution: Convert $C(n) = C(n-1) + 1$ to $C(n) = function(n)$, e.g., $C(n) = n + 1$

1. Start with the recurrence relation: $C(n) = C(n-1) + \ldots$
2. Substitute $C(n-1)$ with its RHS $(C(n-1) = C(n-2) + \ldots)$ in original equation $C(n) = C(n-2) + \ldots$
3. Repeat Step 2
4. Spot the pattern of $C(n)$ and introduce a variable to express this pattern.
5. Determine when the value of this variable that relates $C(n) = C(1) + \ldots$
6. Substitute the value of $C(1)$ and get $C(n)$ in terms of some expression of n

Backward Substitution

Recurrence: $\boldsymbol{C(n) = C(n-1) + 1}$ **for** $\boldsymbol{n > 1}$**, and** $\boldsymbol{C(1) = 0}$

1. $C(n) = C(n-1) + 1$
2. Substitute $C(n-1) = C(n-2) + 1$ into original equation
3. $C(n) = [C(n-2) + 1] + 1 = C(n-2) + 2$
4. Substitute C(n − 2) = C(n − 3) + 1 into original equation
5. $C(n) = [C(n-3) + 1] + 2 = C(n-3) + 3$
6. We see the pattern $\boldsymbol{C(n) = C(n-i) + i}$ emerge, where $1 \leq i \leq n$
7. Now, we know $C(1) = 0$ and want to determine when $C(n-i) = C(1)$, or when $n - i = 1$. This value is $i = n - 1$

$$C(n) = C(n-i) + i = C(n-(n-1)) + n - 1 = C(1) + n - 1 = 0 + n - 1 = n - 1$$

Hence $\boldsymbol{t(n) = c_{op} \,.\, n \in O(n)}$

Empirical Analysis

- Theoretical analysis of the complexity of an algorithm gives an estimate of the running time and growth rate, but not the real time
- Measuring the actual time of an implementation takes in the real world is very important, especially when comparing two algorithms with the same time complexity

Estimating Execution Time

- With the basic operation and input size, to estimate the actual running time of an algorithm we apply the following:

$$t(n) \approx c_{op} \times C(n)$$

- $t(n)$ is the running time.
- n is the input size.
- c_{op} is the execution time for a basic operation.
- $C(n)$ is the number of times the basic operation is executed.

Estimating Execution Time

Example 1: Given sample N and Time values, we can generate a formula for calculating the performance as follow:

- $N = 10$, $T = 63.9$ (≈ 64)
- $N = 15$, $T = 88$
- $N = 20$, $T = 113$

As the growth appears to be linear, we can write the equations:

$64 = 10 * a + b$ (1) and $88 = 15 * a + b$ (2)

By subtracting the above equations, we get, $5a = 24 \Rightarrow a = 4.8$

By substituting a in (1), we get $48 + b = 64 \Rightarrow b = 16$

So, the approximations formula is $T = 4.8 * N + 16$

We can check this by substituting $N = 20 \Rightarrow 4.8 * 20 + 16 = 112$ (≈ 113)

Estimating Execution Time - 2

Example 2

- N = 5, T = 46.9 (≈ 47)
- N = 10, T = 143.1 (≈ 143)
- N = 20, T = 483
  - As the growth appears to be quadratic, we use $aN^2 + bN + c$
  - For N = 5, T = 47, the equation is $25a + 5b + c = 47$ (1)
  - For N = 10, T = 143, the equation is $100a + 10b + c = 143$ (2)
  - For N = 20, T = 483, the equation is $400a + 20b + c = 483$ (3)
- Can you solve (1), (2), and (b) to find a, b, and c?

Growth Rate Functions

If an algorithm takes 1 second to run with the problem size 8, approximately how long would it take for that algorithm with the problem size 16?

• $O(1)$	$T(n) = 1$ second
• $O(\log_2 n)$	$T(n) = 1*\log_2 16 / \log_2 8 = 4/3$ seconds
• $O(n)$	$T(n) = 1*16 / 8 = 2$ seconds
• $O(n*\log_2 n)$	$T(n) = 1*16*\log_2 16 / 8*\log_2 8 = 8/3$ secs
• $O(n^2)$	$T(n) = 1*16^2 / 8^2 = 4$ seconds
• $O(n^3)$	$T(n) = 1*16^3 / 8^3 = 8$ seconds
• $O(2^n)$	$T(n) = 1*2^{16} / 2^8 = 2^8$ seconds $= 256$ secs

Remember

if doubling the input size takes

- 2 times longer, its $O(n)$
- 4 times longer, its $O(n^2)$
- 8 times longer, is $O(n^3)$

if tripling the input size takes

- 3 times longer, its $O(n)$
- 9 times longer, its $O(n^2)$
- 27 times longer, is $O(n^3)$

Estimating Performance

- We can estimate the growth rate if we collect performance data on a number of inputs of different sizes
- We can then apply it to other input sizes

Input 1	Time 1	Input 2	Time 2	Big O	Input 3	Time 3
100	5	200	10	O(n)	1000	50
10	5	20	20	$O(n^2)$	1000	50,000
10	10	30	90	$O(n^2)$	10000	10^7
50	6	100	48	$O(n^3)$	5000	$6*10^6$
50	5	100	5	O(1)	10000	5
100	10	300	270	$O(n^3)$	1000000	10^{13}

Benchmarking Algorithms

Theoretical: Merge-sort and Quick-sort have O(n log(n)) complexity, while Selection-sort is O(n^2).

Empirical: Running Times (in seconds) for different sorting algorithms on a randomised list:

Input (list size)	**500**	**2,500**	**10,000**
Merge sort	0.8	8.1	39.8
Quick sort	0.3	1.3	5.3
Selection sort	1.5	35.0	534.7

Question

One of your data sources to predict the weather are images sent from satellites. When you receive this data, you find that for 5, 10 and 20 images your code took 24 msec, 99 msec and 401 msec respectively.

- What is order of complexity of your algorithm and justify your answer
- Your final input file has 1,000,000 images. Without using a calculator, can you determine approximately how many days it will take your code to run?

Theory vs. Practice

- Formal Analysis
  - Pros: Independent of implementation / hardware details
  - Cons: Hides constant factors. Only feasible on simple examples
- Empirical Analysis:
  - Pro: Measure and model the performance on working code. Can be used on complex examples
  - Con: Implementation specific. May be running on the “wrong” inputs.

RMIT Classification: Trusted

Other Topics (*)

Other Types of Bounds (*)

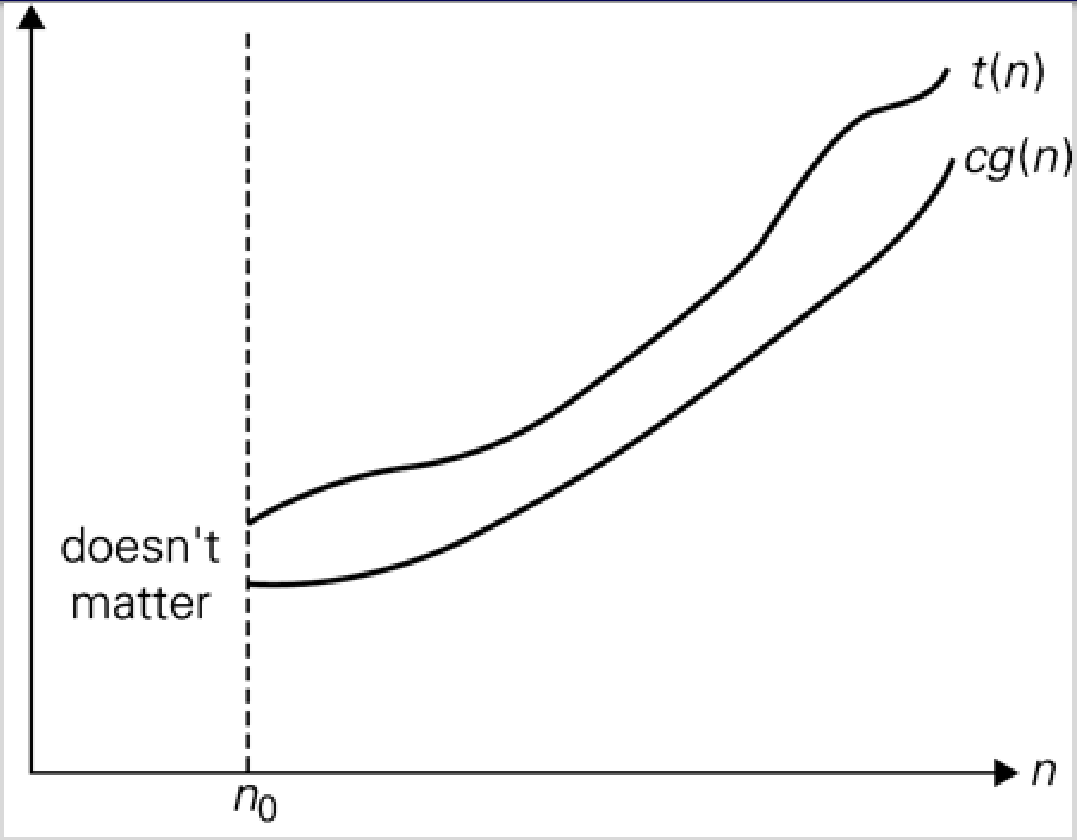
- So we talked about Big O notation for upper bounds…
- What about other types of bounds?

Lower Bound – Ω(*n*) (*)

Definition: given a function $t(n)$,

- Formally: $t(n) \in \Omega(g(n))$, if $g(n)$ is a function and $c \cdot g(n)$ is a **lower** bound on $t(n)$ for some $c > 0$ and for *"large"* n.
- Informally: $t(n) \in \Omega(g(n))$ means $g(n)$ is a function that, as n increases, is a lower bound of $t(n)$

Lower Bound – Ω(*n*) (*)

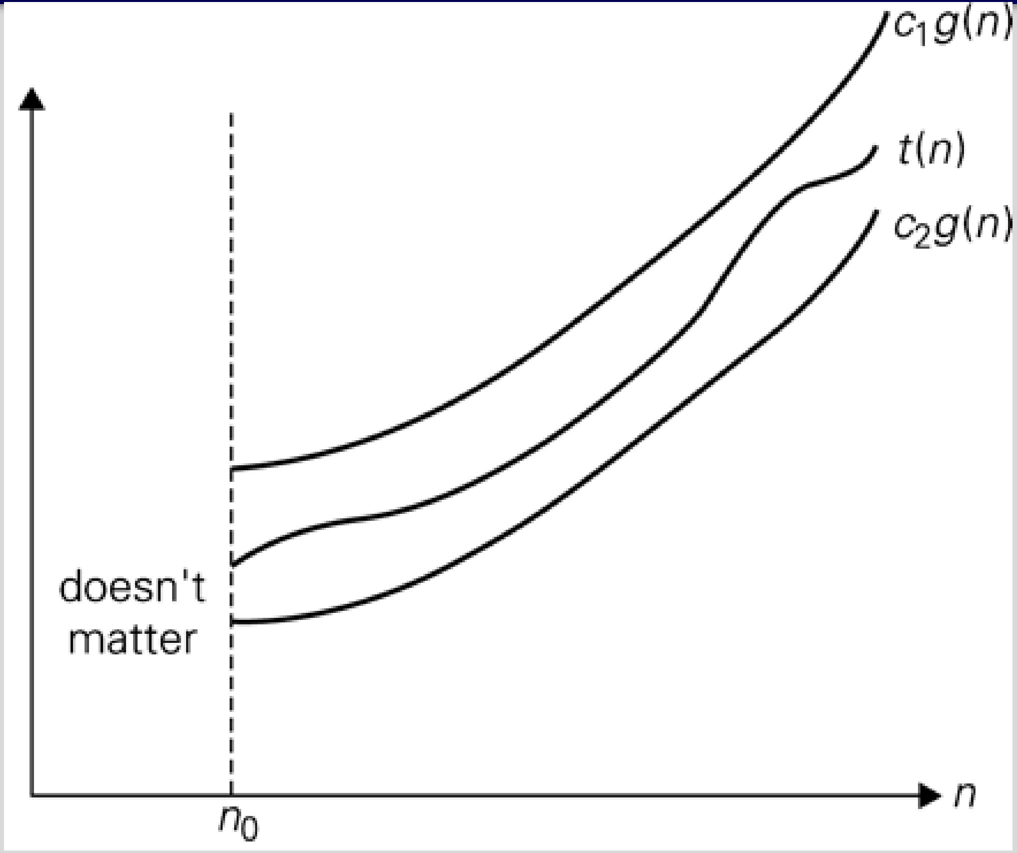


Exact Bounds – Θ(*n*) (*)

Exact Bound Definition: Given a function $t(n)$,

- Formally: $t(n) \in \Theta(g(n))$, if $g(n)$ is a function and $c_1 \cdot g(n)$ is an **upper** bound on $t(n)$ and $c_2 \cdot g(n)$ is an **lower** bound on $t(n)$, for some $c_1 > 0$ and $c_2 > 0$ and for *"large"* n
- Informally: $t(n) \in \Theta(g(n))$ means $g(n)$ is a function that, as n increases, is both an upper and a lower bound of $t(n)$

Exact Bounds – Θ(*n*) (*)



Examples (*)

	Upper bound			Lower bound		
$t(n)$	$O(n)$	$O(n^2)$	$O(n^3)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$
$\log_2 n$	T	T	T	F	F	F
$10n + 5$	T	T	T	T	F	F
$n(n-1)/2$	F	T	T	T	T	F
$(n+1)^3$	F	F	T	T	T	T
2^n	F	F	F	T	T	T

Some Clarifications…

- Generally O(n) is most commonly used…
- But exact bounds Θ(n) tell us the bounds are tight and the algorithm doesn't have anything outside what we expect
- Lower bounds Ω(n) are useful to describe the (theoretical) limits of whole classes of algorithms, and also sometimes useful to state how fast can the best case reach.

Some clarifications…

- O(n) is not the same thing as “Worst Case Efficiency”
- Ω(n) is not the same thing as “Best Case Efficiency”
- Θ(n) is not the same thing as “Average Case Efficiency”

However, it is perfectly reasonable to want the Ω(n) and O(n) worst case efficiency bounds for a class of algorithms.

RMIT Classification: Trusted

RMIT UNIVERSITY

Copyright © 2017 RMIT University Vietnam