
CS2040 Data Structures and Algorithms

Lecture Note #3

Analysis of Algorithms

Objectives

1

- To introduce the theoretical basis for measuring the efficiency of algorithms

2

- To learn how to use such measure to compare the efficiency of different algorithms

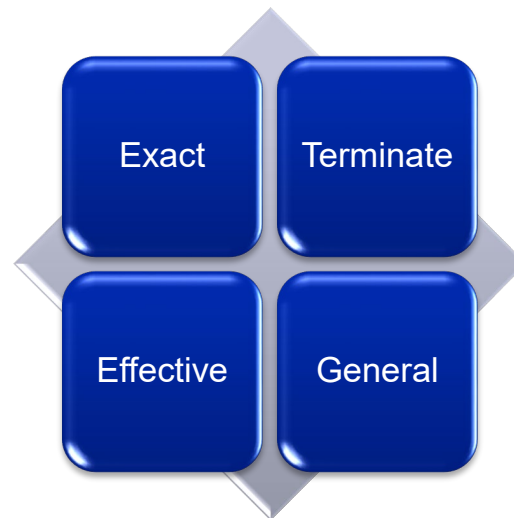
Outline

1. What is an **Algorithm**?
2. What do we mean by **Analysis of Algorithms**?
3. **Big-O** notation – Upper Bound
4. How to find the complexity of a program?

1 What is an algorithm?

1 Algorithm

- A step-by-step procedure for solving a problem.
- Properties of an algorithm:
 - Each step of an algorithm must be **exact**.
 - An algorithm must **terminate**.
 - An algorithm must be **effective**.
 - An algorithm should be **general**.



2 What do we mean by Analysis of Algorithms?

2.1 What is Analysis of Algorithms?

■ Analysis of algorithms

- Provides tools for comparing the efficiency of different methods of solution (rather than programs)
- Efficiency = Complexity of algorithms

■ A comparison of algorithms

- Should focus on significant differences in the efficiency of the algorithms
- Should not consider reductions in computing costs due to clever coding tricks. Tricks may reduce the readability of an algorithm.

2.2 Determining the Efficiency/Complexity of Algorithms

- To evaluate rigorously the resources (time and space) needed by an algorithm and represent the result of the analysis with a formula
- We will emphasize more on the time requirement rather than space requirement here
- The time requirement of an algorithm is also called its time complexity

2.3 By measuring the run time?

TimeTest.java

```
public class TimeTest {  
    public static void main(String[] args) {  
        long startTime = System.currentTimeMillis();  
        long total = 0;  
        for (int i = 0; i < 10000000; i++) {  
            total += i;  
        }  
        long stopTime = System.currentTimeMillis();  
        long elapsedTime = stopTime - startTime;  
        System.out.println(elapsedTime);  
    }  
}
```

Note: The run time depends on the compiler, the computer used, and the current work load of the computer.

2.4 Exact run time is not always needed

- ❑ Using exact run time is not meaningful when we want to **compare** two algorithms
 - coded in different languages,
 - using different data sets, or
 - running on different computers.

2.5 Determining the Efficiency of Algorithms

- Algorithm analysis should be independent of
 - ❑ Specific implementations
 - ❑ Compilers and their optimizers
 - ❑ Computers
 - ❑ Data

2.6 Execution Time of Algorithms

- Instead of working out the exact timing, we count the number of some or all of the **primitive operations** (e.g. **+**, **-**, *****, **/**, **assignment**, ...) needed.
- Counting an algorithm's **operations** is a way to assess its efficiency
 - An algorithm's execution time is related to the **number of operations** it requires.

2.7 Counting the number of statements

- To simplify the counting further, we can ignore
 - the different types of operations, and
 - different number of operations in a statement,and simply **count the number of statements executed**.

2.8 Computation cost of an algorithm

- How many operations are required?

```
for (int i=1; i<=n; i++) {  
    perform 100 operations;           // A  
    for (int j=1; j<=n; j++) {  
        perform 2 operations;        // B  
    }  
}
```

$$\text{Total Ops} = A + B = \sum_{i=1}^n 100 + \sum_{i=1}^n \left(\sum_{j=1}^n 2 \right)$$

$$= 100n + \sum_{i=1}^n 2n = 100n + 2n^2 = 2n^2 + 100n$$

2.9 Approximation of analysis results

- Very often, we are interested only in using a simple term to **indicate how efficient an algorithm is**. The exact formula of an algorithm's performance is not really needed.

- Example:

Given the formula: $2n^2 + 100n$

- the **dominating term** $2n^2$ can tell us approximately how the algorithm performs by providing us with a measure of the **growth rate** (how the number of operations executed grows as n increases in size) of the algorithm
- This is called asymptotic analysis of the algorithm

2.10 Asymptotic analysis

- **Asymptotic analysis** is an analysis of algorithms that focuses on
 - analyzing the problems of **large input size**,
 - considering only the **leading term** of the formula, and
 - **ignoring** the **coefficient** of the leading term
- Some notations are needed in asymptotic analysis

2.11 Algorithm Growth Rates (1/2)

- An algorithm's time requirement can be measured as a function of the **problem size**, say n
- An algorithm's **growth rate**
 - Enables the comparison of one algorithm with another
 - Examples
 - Algorithm A requires time proportional to n^2
 - Algorithm B requires time proportional to n

2.12 Algorithm Growth Rates (2/2)

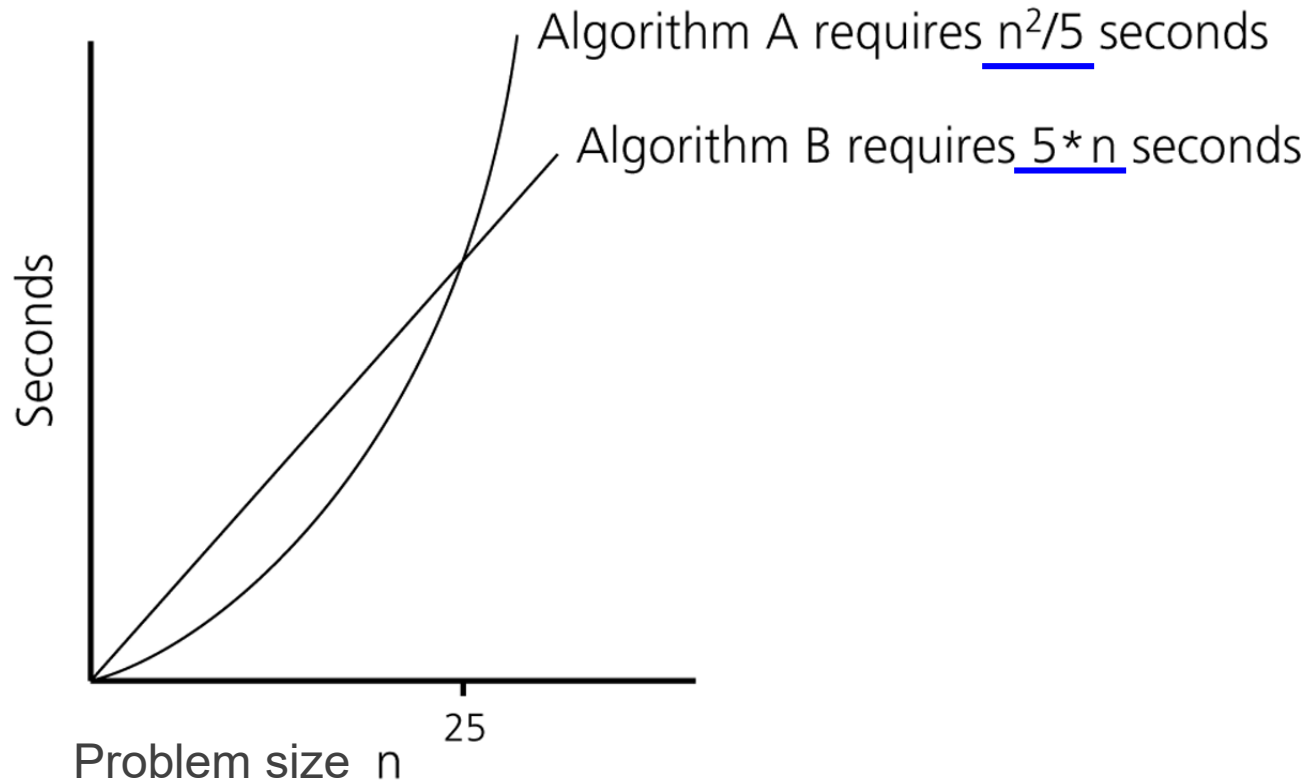


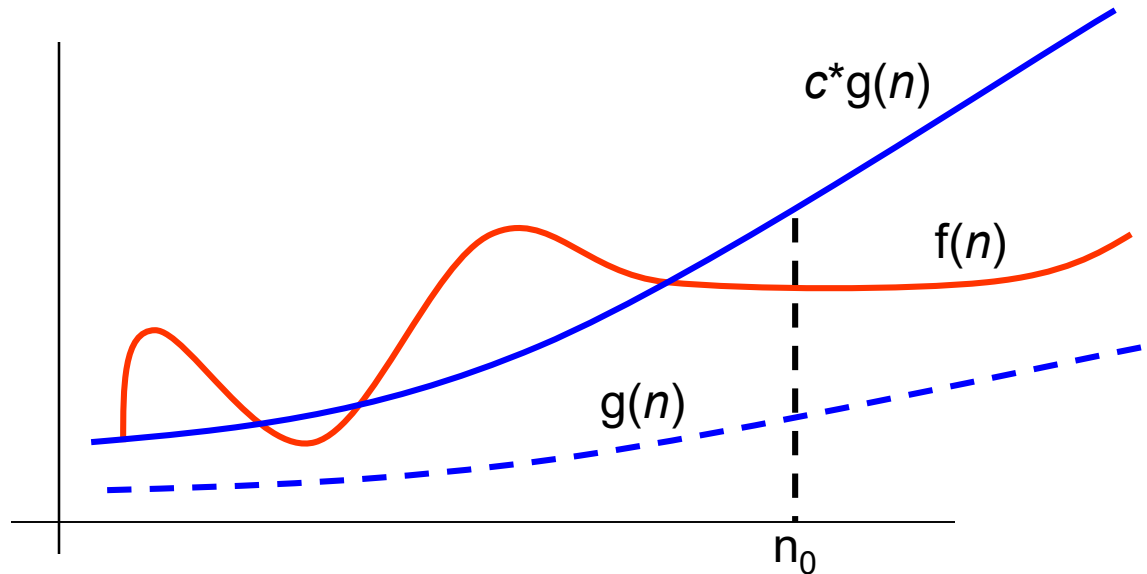
Figure - Time requirements as a function of the problem size n

- Algorithm efficiency is typically a concern for **large problems** only. **Why?**

3 Big O notation

3.1 Definition – Big O notation

- Given a function $f(n)$, we say $g(n)$ is an (asymptotic) **upper bound** of $f(n)$, denoted as $f(n) = O(g(n))$, if there exist a constant $c > 0$, and a positive integer n_0 such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.
- $f(n)$ is said to be **bounded from above** by $g(n)$.
- $O()$ is called the “big O” notation.



3.2 Ignore the coefficients of all terms

- Based on the definition, $2n^2$ and $30n^2$ have the same upper bound n^2 , i.e.,

- $2n^2 = O(n^2)$

Why?

$$f_1(n) = 2n^2; g(n) = n^2.$$

$$\text{Let } c=3 \text{ and } n_0=1, \text{ since } 2n^2 \leq cn^2 \quad \forall n \geq n_0$$

$$\text{Hence } f_1(n) = O(g(n))$$

- $30n^2 = O(n^2)$

$$f_2(n) = 30n^2; g(n) = n^2.$$

$$\text{Let } c=31 \text{ and } n_0=1, \text{ since } 30n^2 \leq cn^2 \quad \forall n \geq n_0$$

$$\text{Hence } f_2(n) = O(g(n))$$

They differ only in the choice of c .

- Therefore, in big O notation, we can omit the coefficients of all terms in a formula:
 - Example: $f(n) = 2n^2 + 100n = O(n^2) + O(n)$

3.3 Finding the constants c and n_0

- Given $f(n) = 2n^2 + 100n$, prove that $f(n) = O(n^2)$.

Observe that: $2n^2 + 100n \leq 2n^2 + n^2 = 3n^2$ whenever $n \geq 100$.

→ Set the constants to be $c = 3$ and $n_0 = 100$.

By definition, we have $f(n) = O(n^2)$.

Notes:

1. $n^2 \leq 2n^2 + 100n$ for all n , i.e., $g(n) \leq f(n)$, and yet $g(n)$ is an asymptotic upper bound of $f(n)$

2. c and n_0 are not unique.

For example, we can choose $c = 2 + 100 = 102$, and $n_0 = 1$ (because $f(n) \leq 102n^2 \forall n \geq 1$)

Q: Can we write $f(n) = O(n^3)$?

Yes

3.4 Is the bound tight?

- The complexity of an algorithm can be bounded by many functions.
- Example:
 - Let $f(n) = 2n^2 + 100n$.
 - $f(n)$ is bounded by n^2 , n^3 , n^4 and many others according to the definition of big O notation.
 - Hence, the following are all correct:
 - $f(n) = O(n^2)$; $f(n) = O(n^3)$; $f(n) = O(n^4)$
- However, we are more interested in the **tightest bound** which is n^2 for this case.

3.5 Growth Terms: Order-of-Magnitude

- In asymptotic analysis, a formula can be simplified to a single term with coefficient 1
- Such a term is called a growth term (rate of growth, order of growth, order-of-magnitude)
- The most common growth terms can be ordered as follows: (note: many others are not shown)

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$
“fastest” “slowest”

Note:

- “log” = log base 2, or \log_2 ; “ \log_{10} ” = log base 10; “ln” = log base e. In big O, all these log functions are the same. (Why?)

3.6 Examples on big O notation

- $f1(n) = \frac{1}{2}n + 4$
 $= O(n)$
- $f2(n) = 240n + 0.001n^2$
 $= O(n^2)$
- $f3(n) = n \log n + \log n + n \log (\log n)$
 $= O(n \log n)$

3.7 Order-of-Magnitude Analysis and Big O Notation (1/2)

(a)

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

Figure - Comparison of growth-rate functions in tabular form

3.8 Order-of-Magnitude Analysis and Big O Notation (2/2)

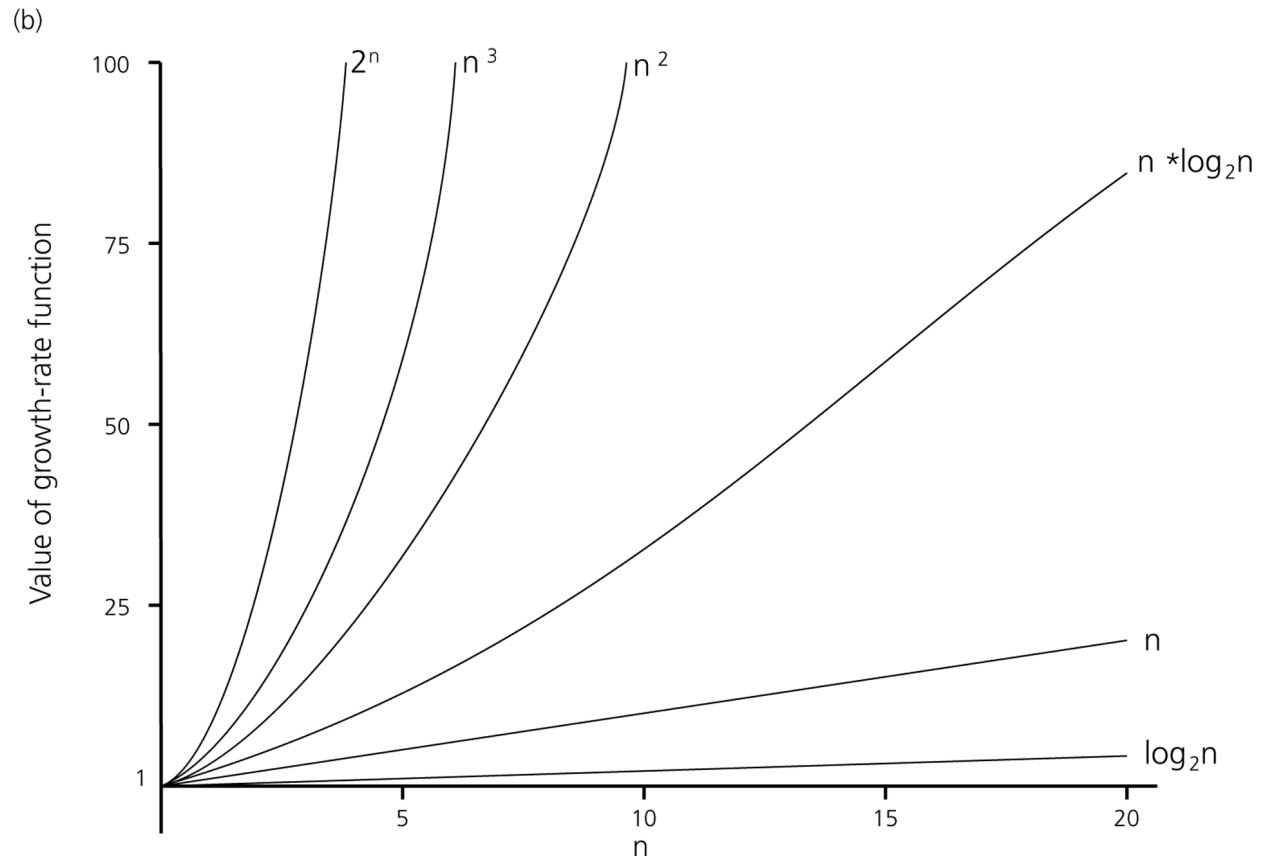


Figure - Comparison of growth-rate functions in graphical form

3.9 Summary: Order-of-Magnitude Analysis and Big O Notation

- Order of growth of some common functions:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$

- Properties of growth-rate functions

- You can ignore low-order terms
- You can ignore a multiplicative constant in the high-order term
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

4 How to find the complexity of a program?

4.1 Some rules of thumb and examples

- Basically just count the number of statements executed.
- If there are only a small number of simple statements in a program
 - $O(1)$
- If there is a 'for' loop dictated by a loop index that goes up to n
 - $O(n)$
- If there is a nested 'for' loop with outer one controlled by n and the inner one controlled by m
 - $O(n*m)$
- For a loop with a range of values n , and each iteration reduces the range by a fixed constant fraction (eg: $\frac{1}{2}$)
 - $O(\log n)$
- For a recursive method, each call is usually $O(1)$. So
 - if n calls are made – $O(n)$
 - if $n \log n$ calls are made – $O(n \log n)$

Infinite Geometric series
 $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \dots = 2$

4.2 Examples on finding complexity (1/2)

- What is the complexity of the following code fragment?

```
int sum = 0;
for (int i=1; i<n; i=i*2) {
    sum++;
}
```

- It is clear that **sum** is incremented only when

$$i = 1, 2, 4, 8, \dots, 2^k \text{ where } k = \lfloor \log_2 n \rfloor$$

There are $k+1$ iterations. So the complexity is $O(k)$ or $O(\log n)$

Note:

- In Computer Science, **log n** means $\log_2 n$.
- When 2 is replaced by 10 in the 'for' loop, the complexity is $O(\log_{10} n)$ which is the same as $O(\log_2 n)$. (Why?)
- $\log_{10} n = \log_2 n / \log_2 10$

4.2 Examples on finding complexity (2/2)

- What is the complexity of the following code fragment?
(For simplicity, let's assume that n is some power of 3.)

```
int sum = 0;
for (int i=1; i<n; i=i*3) {
    for (j=1; j<=i; j++) {
        sum++;
    }
}
```

- $$\begin{aligned} f(n) &= 1 + 3 + 9 + 27 + \dots + 3^{(\log_3 n)} \\ &= 1 + 3 + \dots + n/9 + n/3 + n \\ &= n + n/3 + n/9 + \dots + 3 + 1 \quad (\text{reversing the terms in previous step}) \\ &= n * (1 + 1/3 + 1/9 + \dots) \\ &\leq n * (3/2) \\ &= 3n/2 \\ &= O(n) \end{aligned}$$

Why is $(1 + 1/3 + 1/9 + \dots) \leq 3/2$?

This is sum of infinite geometric series see word file

“analysis_of_algorithms_useful_equalities.docx”

4.3 Non-recursive Binary Search Algorithm (1)

- Requires array to be **sorted** in ascending order
- Maintain subarray where **x** (the search key) might be located
- Repeatedly compare **x** with **m**, the middle element of current subarray
 - If **x** = **m**, found it!
 - If **x** > **m**, continue search in subarray after **m**
 - If **x** < **m**, continue search in subarray before **m**

4.3 Non-recursive Binary Search Algorithm (2)

- Data in the array `a[]` are sorted in ascending order

```
public static int binSearch(int[] a, int len, int x)
{
    int mid, low = 0;
    int high = len - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x == a[mid]) return mid;
        else if (x > a[mid]) low = mid + 1;
        else high = mid - 1;
    }
    return -1;
}
```

4.3 Non-recursive Binary Search Algorithm (3)

- At any point during binary search, part of array is “*alive*” (might contain the point *x*)
- Each iteration of loop eliminates at least *half* of previously “*alive*” elements
- At the beginning, all *n* elements are “*alive*”, and after
 - After 1 iteration, at most $n/2$ elements are left, or alive
 - After 2 iterations, at most $(n/2)/2 = n/4 = n/2^2$ are left
 - After 3 iterations, at most $(n/4)/2 = n/8 = n/2^3$ are left
 - :
 - After *i* iterations, at most $n/2^i$ are left
 - At the final iteration, at most **1** element is left

4.3 Non-recursive Binary Search Algorithm (4)

In the **worst case**, we have to search all the way up to the last iteration **k** with only one element left.

We have:

$$n/2^k = 1$$

$$2^k = n$$

$$k = \log n$$

Hence, the binary search algorithm takes $O(f(n))$, or $O(\log n)$ times

4.4 Time complexity of recursion: Fibonacci numbers

- Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, ...
 - The first two Fibonacci numbers are both 1 (arbitrary numbers)
 - The rest are obtained by adding the previous two together.
- Calculating the n^{th} Fibonacci number recursively:

$$\begin{aligned} \text{Fib}(n) &= 1 && \text{for } n=1, 2 \\ &= \text{Fib}(n-1) + \text{Fib}(n-2) && \text{for } n > 2 \end{aligned}$$

```
// Precond: n > 0
public static int fib(int n) {
    if (n <= 2)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

4.4 Time complexity of recursion: Fibonacci numbers

- Function for total number of operations is almost exactly like the recursive case of Fibonacci
 - $T(n) = T(n-1) + T(n-2) + 4$ (1 comparison, 1 addition, 2 subtraction)
 - $\Rightarrow T(n) < 2T(n-1) + c$ (replace 4 by some constant c)
 - $\Rightarrow < 2(2T(n-2) + c) + c$ (expand $T(n-1)$ to be $2 * T(n-2) + c$)
 - $\Rightarrow < 4T(n-2) + 2c + c$
 - $\Rightarrow < 8T(n-3) + 4c + 2c + c$ (cont. the expansion)
 - $\Rightarrow \dots$ (will stop when n is 1)
 - $\Rightarrow < 2^n(T(1)) + 2^{n-1}c + 2^{n-2}c + \dots + c$ ($T(1)$ is c)
 - $\Rightarrow < 2^n * (1 + 1/2 + 1/4 + 1/8 + \dots) * c$
 - $\Rightarrow = O(2^n)$

4.5 Analysis of Different Cases

Worst-Case Analysis

- ❑ Interested in the worst-case behaviour.
- ❑ A determination of the maximum amount of time that an algorithm requires to solve problems of size n

Best-Case Analysis

- ❑ Interested in the best-case behaviour
- ❑ Not useful

Average-Case Analysis

- ❑ A determination of the average amount of time that an algorithm requires to solve problems of size n
- ❑ Have to know the probability distribution
- ❑ The hardest

4.6 The Efficiency of Searching Algorithms

- Example: Efficiency of **Sequential Search** (data not sorted)
 - Worst case: $O(n)$
Which case?
 - Average case: $O(n)$
 - Best case: $O(1)$
Why? Which case?
 - Unsuccessful search?
- Q: What is the best case complexity of **Binary Search** (data sorted)?
 - Best case complexity is not interesting. Why?

4.7 Keeping Your Perspective

- If the problem size is always **small**, you can probably ignore an algorithm's efficiency
- Weigh the **trade-offs** between an algorithm's **time** requirements and its **memory** requirements
- Order-of-magnitude analysis focuses on **large** problems
- There are other measures, such as big Omega (Ω), big theta (Θ), little oh (o), and little omega (ω). These may be covered in more advanced module.

End of file