

# CSEN102

## Introduction to Computer Science

### Lecture 6:

### Algorithm Discovery: Attributes of Algorithms, Measuring Efficiency

Prof. Dr. Slim Abdennadher

Dr. Aysha Alsafty, [slim.abdennadher@guc.edu.eg](mailto:slim.abdennadher@guc.edu.eg),  
[aysha.alsafty@guc.edu.eg](mailto:aysha.alsafty@guc.edu.eg)

German University Cairo, Department of Media Engineering and Technology

25.11.2017 - 30.11.2017

## 1 Algorithm discovery

### 1.1 Attributes of algorithms

#### Attributes of algorithms

Which *characteristics* mark a good algorithm?

- *Correctness*
  - Give a correct solution to the problem!
- *Ease of understanding*
  - Clarity and ease of handling
- *Program maintenance*
  - Fix errors
  - Extend the program to meet new requirements
- *Efficiency*
  - *Time*: How long does it take to solve the problem?
  - *Space*: How much memory is needed?

## 1.2 A choice of algorithms

### A choice of algorithms

It is possible to come up with a *several different* algorithms to solve one and the *same* problem

- Which one is the best?
  - Most efficient: Time vs. Space
  - Easiest to maintain
- How do we measure time efficiency?
  - Running time?
  - Number of executed operations?

## 1.3 Measuring efficiency

### Measuring Efficiency

- Need a *metric* to measure efficiency of algorithms
  - Running Time: How long does it take to solve the problem?
    - \* Depends on machine speed
  - Number of Operations: How many operations does the algorithm execute?
    - \* Better metric but a lot of work to count all operations
  - Number of Fundamental Operations: How many “*fundamental operations*” does the algorithm execute?
- Depending on *size* and *type* of input, we are interested in:
  - *Best-case, Worst-case, Average-case* behavior
- *Need to analyze the algorithm!*

## 1.4 Examples: Average

### Example: Average of $n$ numbers – counting

*Problem: Find the average of  $n$  numbers*

```
1 list_A = eval(input())
2 n = len(list_A)
3 sum = 0
4 i = 0
5 while i < n:
6     sum = (sum + list_A[i])
7     i = (i + 1)
8 average = sum/n
9 print(average)
```

- How many steps does the algorithm execute?
  - Steps 1, 2, 3, 4, 8, and 9 are executed *once*.
  - Steps 5, 6, and 7 *depend on the input size  $n$* .

### Example: Average of $n$ numbers – doing the sum

Problem: Find the average of  $n$  numbers

```
1 list_A = eval(input()) 1 operation => executed once
2 n = len(list_A)        1 operation => executed once
3 sum = 0                 1 operation => executed once
4 i = 0                   1 operation => executed once
5 while i < n:           1 operation => (n+1) repetitions
6     sum = sum + list_A[i] 1 operation => n repetitions
7     i = i + 1          1 operation => n repetitions
8 average = sum/n        1 operation => executed once
9 print (average)        1 operation => executed once
```

### Total number of executed operations

$$1 + 1 + 1 + 1 + 1 + (n + 1) + n + n + 1 + 1 = 3n + 7$$

## 1.5 Examples Sequential search

### Sequential search – Analysis

```
1 Name = eval(input())
2 list_N = eval(input())
3 list_T = eval(input())
4 n = len(list_N)
5 i = 0
6 Found = False
7 while Found == False and i < n:
8     if Name == list_N[i]:
9         print(list_T[i])
10        Found = True
11    else:
12        i = i + 1
13 if Found == False:
14    print("sorry, name is not in directory")
```

How many steps does the algorithm execute?

### Sequential search – Analysis

How many steps does the algorithm execute?

- Steps 1, 2, 3, 4, 5, 6, 9, 10, 13 and 14 are executed *at most* once.
- Step 7 is executed *at most*  $n + 1$  times.
- Steps 8 and 12 are executed *at most*  $n$  times.

So what is the running time now? (focus on the loop)

- Worst case: Steps 8 and 12 are executed at most  $n$  times and step 7  $n + 1$  times.
- Best case: Step 7 is executed exactly *twice*. Step 8 is executed only *once*.
- Average case: Steps 8 and 12 are executed approximately  $\frac{n}{2}$  times.

## Sequential search – Worst case behavior

```
1 Name = eval(input())           1 ops => executed once
2 list_N = eval(input())         1 ops => executed once
3 list_T = eval(input())         1 ops => executed once
4 n = len(list_N)                1 ops => executed once
5 i = 0; Found = False           2 ops => executed once
6 while (i < n and Found == False): 1 op => (n+1) reps
7     if Name == list_N[i]:      1 op => n repetitions
8         print(list_T[i])       skipped...
9         Found == True          skipped...
10    else: i = i + 1             1 op => n reps
11    if Found == False:         1 op => executed once
12    print("Sorry, name not in directory") 1 op => executed once
```

Assume the name is *not* in the list!

### Total number of executed operations

$$1 + 1 + 1 + 1 + 2 + (n + 1) + n + n + 1 + 1 = 3n + 9$$

## Sequential search – Worst case behavior

```
1 Name = eval(input())           1 ops => executed once
2 list_N = eval(input())         1 ops => executed once
3 list_T = eval(input())         1 ops => executed once
4 n = len(list_N)                1 op => executed once
5 i = 0; Found = False           2 ops => executed once
6 while (i < n and Found == False) 1 op => (n+1) reps
7     if Name = list_N[i]:       1 op => n repetitions
8         print(list_T[i])       1 op => executed once
9         Found = YES            1 op => executed once
10    else: i = i + 1             1 op => (n-1) reps
11    if Found == False:         1 op => executed once
12    print("Sorry, name not in directory") skipped...
```

Assume the name is *the last one* in the list!

### Total number of executed operations

$$1 + 1 + 1 + 1 + 2 + (n + 1) + n + 1 + 1 + (n - 1) + 1 = 3n + 9$$

## 2 Order of magnitude

### 2.1 What really matters

#### What really matters

- We are
  - *not* interested in knowing the *exact number* of operations the algorithm performs.
  - mainly interested in knowing how the number of operations *grows* with increased input size!
- Why?

- Given large enough input, the algorithm with faster growth will execute more operations.

Consider algorithms A and B with input size  $n = 10,000$ :

- Algorithm A has a running time of  $570n + 920$  operations  $\rightarrow 5,700,920$  operations.
- Algorithm B has a running time of  $n^2 + 21$  operations  $\rightarrow 100,000,021$  operations.
- Algorithm B takes already *twenty times* longer!

## 2.2 Big-O notation

### The “Order of Magnitude”

The Order of Magnitude is a *formula* that *strips away all ballast*. For example...

- $n$
- $6n$
- $6n + 278$
- $5000n + 2000$

Are all in the Order of Magnitude of  $n$

- $n^3$
- $200n^3 + 150n + 20$
- $50n^3 + n^2$

Are all in the Order of Magnitude of  $n^3$

### The “Big O notation”

- We write  $O(n)$  for the Order of Magnitude of  $n$ .
- Generally, if a function  $g$  is in the Order of Magnitude of a function  $f$ , we write ( $g = O(f)$ ).

### For your reference...

Formally, the Order of Magnitude is defined as:

$$\exists c. \exists n_0. \forall n > n_0. c \cdot f(n) > g(n) \Leftrightarrow g = O(f)$$

You are only interested into the *fastest growing part* of the function you have!

### Beware of fast growth!

Imagine algorithms A, B, and C, with

- A in  $O(n)$ ,
- B in  $O(n^2)$ , and
- C in  $O(2^n)$

Consider an operation taking  $\frac{1}{100}s$

$n$	10	20	30	40
A	$\frac{1}{10}s$	$\frac{2}{10}s$	$\frac{3}{10}s$	$\frac{4}{10}s$
B	1s	4s	9s	16s
C	ca. 10s	ca. 3h	ca. 4 months	ca. 348 years

### Some terms

- Algorithms in  $O(1)$  are called *constant*
- Algorithms in  $O(n)$  are called *linear*
- Generally, algorithms in  $O(n^x)$  for some  $x$  are called *polynomial* (constant, linear, quadratic, cubic, ...)
- Algorithms in  $O(2^n)$  are called *exponential*

## 3 Summary

### Summary

- We are concerned about the efficiency of algorithms
  - Time and space efficiency
  - An analysis of the algorithm is necessary
- The order of magnitude (Big O notation) measures efficiency
  - $O(1)$ ,  $O(n)$ ,  $O(n^2)$ , ...
  - Measures the growth with increasing input size  $n$
  - Aim for low growth rates