Computer Science Department
College of Computer at Al-Lith

جـامـعـة أم الـقـرى
UMM AL-QURA UNIVERSITY

Data Structures

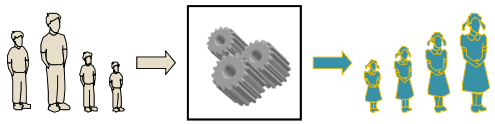
Chapter 1: Basics of algorithm analysis

Instructor Maher Hadiji
hdiji.maher@gmail.com

2015-2016

Outline

1.1 Introduction to Algorithms



Input **Algorithm** **Output**

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

1.1 Introduction to Algorithms

- Algorithms are all around us in everyday life.
- In the recipe of a cook book.
- In assembling a toy.
- In setting the table.
- In preparing a cup of tea.
- In calling your friend on the phone.
- ….There are countless examples!

1.2 Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

1. Understand the Problem (requirement analysis)
2. Select Data structure
3. Write Pseudo Code
4. Analyze Performance
5. Implement using suitable programming language
6. Test to resolve syntax and logic errors

1.2 Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

1. **Understand the Problem (requirement analysis)**
   - Gather data
   - Ask users
   - Carefully review any written requirements

1.2. Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

2. **Select Data structure:**
   To verify the appropriateness of the selected data structure:
   - Judge how well your data structure responds to user requirements (updates, questions)
   - Modify design as necessary

1.2. Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

3. **Write Pseudo Code**
   - Use pseudo code or flow chart
   - level of details of the pseudo code may vary

1.2 Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

4. **Analyze Performance**
   - Determine the feasibility of solution w.r.t. memory requirements, performance constraints … etc.
   - Manually review and validate pseudo code
   - Analyze the complexity of the algorithm using the big O notation to determine the complexity w.r.t. to time and storage.
   - Other criteria include: Clarity, Maintainability, Portability

1.2. Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

5. **Implement** using suitable programming language

1.2 Algorithm Design Basics

- Guidelines for Algorithm Designing and Analysis:

6. **Test to resolve syntax and logic errors.**
   Testing is divided into 2 main parts:
   - Trying to break the function of the program by entering unexpected data.
   - Debugging: It is concerned with finding out what is what caused the program to function in incorrectly.

1.3 Time Complexity of an Algorithm

- Time complexity is a main issue in **evaluating** an algorithm.
- It reflects **how the algorithm responds to the increase in data size (n)** it handles, by measuring the corresponding increase in number of instructions to be performed.
- Time complexity is meant to **classify algorithms** into categories.

1. 3. Time Complexity of an Algorithm

- To compare solutions, several points can be considered
  - Accuracy programs
  - Simplicity programs
  - Convergence and stability of programs
  - Program efficiency (it is desirable that our solutions are not slow, do not take considerable memory space)
- The **efficiency** of algorithms is our concern in this chapter.

1. 3. Time Complexity of an Algorithm

Program Execution → resource usage of the computer

► computation time to perform operations

► memory occupation (program + data)

Problem: we want to measure the efficiency of an algorithm, called the **complexity**

► to be able to predict execution time

► to estimate the resources that will be mobilized in a machine during its execution (amount of memory in particular)

► in order to compare it with another that does the same treatment in another way, in order to choose the best

1. 3. Time Complexity of an Algorithm

objective

"On any machine regardless of the language used, the algorithm A is better than B algorithm for large data"

why large??

Most algorithms transform inputs into one output

The time complexity of an algorithm is usually
Depending on the size of the inputs

Calculates 2^n → f (n)

1. 3. Time Complexity of an Algorithm

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
15	7 ns	100,000 ns
65	32 ns	150,000 ns
250	128 ns	200,000 ns
1,000	500 ns	250,000 ns

Based on this metric, it would be easy to jump to the conclusion that the algorithm A is far superior in efficiency than the algorithm B. However, ...

Additional data show us that our conclusion was wrong.

1. 3. Time Complexity of an Algorithm

n (list size)	Computer A run-time (in nanoseconds)	Computer B run-time (in nanoseconds)
15	7 ns	100,000 ns
65	32 ns	150,000 ns
250	125 ns	200,000 ns
1,000	500 ns	250,000 ns
...	...	...
1,000,000	500,000 ns	500,000 ns
4,000,000	2,000,000 ns	550,000 ns
16,000,000	8,000,000 ns	600,000 ns
...	...	...
$63{,}072 \times 10^{12}$	$31{,}536 \times 10^{12}$ ns, or 1 year	1,375,000 ns, or 1.375 milliseconds

1. 3. Time Complexity of an Algorithm

Analyzing Running Time

	Number of times executed
1. readl(n);	1
2. sum = 0 ;	1
3. i = 0 ;	1
4. while (i < n)	n+1
5. { readln(number);	n
6. sum = sum + number ;	n
7. i = i + 1 ; }	n
8. mean = sum / n ;	1

The computing time for this algorithm in terms on input size n is: T(n) = 4n + 5.

1.4 The Execution Time of Algorithms

Example: Simple If-Statement

	Cost	Times
`if (n < 0)`	c1	1
`absval = -n`	c2	1
`else`		
`absval = n;`	c3	1

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

1.4 The Execution Time of Algorithms

Example: Nested Loop

	Cost	Times
`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

➔ The time required for this algorithm is proportional to n^2

A Simple Example

- ***Linear Search***

INPUT: a sequence of *n* numbers, *key* to search for.
OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

```
boolean found = false;
   int T[n]
integer i, key;
   for ( i = 0; i < n ; i++ )
  {
     if ( key == T[i] )

       break;
  }
   if ( i < n )
    found = true;
    return found;
  }
```

Best case ?
Average case ?
Worst case ?

1.5 A Simple Example

In the simplest terms, for a problem where the input size is n:

- **Best case** = fastest time to complete, with optimal inputs chosen. For example, the best case for a sorting algorithm would be data that's already sorted.
- **Worst case** = slowest time to complete, with pessimal inputs chosen. For example, the worst case for a sorting algorithm might be data that's sorted in reverse order (but it depends on the particular algorithm).
- **Average case** = arithmetic mean. Run the algorithm many times, using many different inputs of size n that come from some distribution that generates these inputs (in the simplest case, all the possible inputs are equally likely), compute the total running time (by adding the individual times), and divide by the number of trials.

Simple Complexity Analysis: Loops

- We start by considering how to count operations in **for**-loops.
  - We use integer division throughout.
- First of all, we should know the number of iterations of the loop; say it is **x**.
  - Then the loop condition is executed **x + 1** times.
  - Each of the statements in the loop body is executed **x** times.
  - The loop-index update statement is executed **x** times.

Simple Complexity Analysis: Loops (with <)

- In the following for-loop:

```
for (int i = k; i < n; i = i + m){
      statement1;
      statement2;
 }
```

The number of iterations is: **(n – k) / m**

- The initialization statement, **i = k**, is executed **one** time.
- The condition, **i < n**, is executed **(n – k) / m + 1** times.
- The update statement, **i = i + m**, is executed **(n – k) / m** times.
- Each of **statement1** and **statement2** is executed **(n – k) / m** times

Simple Complexity Analysis : Loops (with <=)

- In the following for-loop:

```
for (int i = k; i <= n; i = i + m){
      statement1;
      statement2;
 }
```

- The number of iterations is: **(n – k) / m + 1**
- The initialization statement, **i = k**, is executed **one** time.
- The condition, **i <= n**, is executed **(n – k) / m + 2** times.
- The update statement, **i = i + m**, is executed **(n – k) / m + 1** times.
- Each of **statement1** and **statement2** is executed **(n – k) / m + 1** times.

Simple Complexity Analysis: Loop Example

- Find the exact number of basic operations in the following program fragment:

```
double x, y;
x = 2.5 ; y = 3.0;
for(int i = 0; i < n; i++){
   a[i] = x * y;
   x = 2.5 * x;
   y = y + a[i];
}
```

- There are 2 assignments outside the loop => 2 operations.
- The **for** loop actually comprises
- an assignment (i = 0) => 1 operation
- a test (i < n) => n + 1 operations
- an increment (i++) => 2 n operations
- the loop body that has three ***assignments***, two ***multiplications***, and an ***addition*** => 6 n operations

Thus the total number of basic operations is 6 * n + 2 * n + (n + 1) + 3
= **9n + 4**

Simple Complexity Analysis: Examples

- Suppose **n** is a multiple of 2. Determine the number of basic operations performed by of the method myMethod():

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

```
static int helper(int n){
    int sum = 0;
    for(int i = 1; i <= n; i++)
         sum = sum + i;
    return sum;
}
```

- Solution: The number of iterations of the loop:
  for(int i = 1; **i < n**; i = i * 2)
  sum = sum + i + *helper(i)*;
  is **$\log_2 n$** (A Proof will be given later)
  Hence the number of basic operations is:
  1 + 1 + (1 + log2 n) + *log2 n[2 + 4 + 1 + 1 + (n + 1) + n[2 + 2] + 1]* + 1
  = 3 + log2 n + log2 n[10 + 5n] + 1
  = 5 n log2 n + 11 log2 n + 4

Simple Complexity Analysis: Loops With Logarithmic Iterations

- In the following for-loop: (with <)

```
for (int i = k; i < n; i = i * m){
      statement1;
      statement2;
 }
```

  - The number of iterations is: $\lceil (\text{Log}_m (n / k)) \rceil$

- In the following for-loop: (with <=)

```
for (int i = k; i <= n; i = i * m){
      statement1;
      statement2;
 }
```

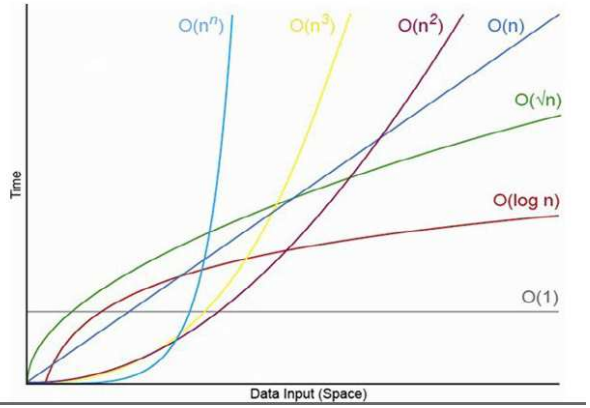
  - The number of iterations is: $\lfloor (\text{Log}_m (n / k) + 1) \rfloor$

The Growth of Functions

"Popular" functions g(n) are
n log n, 1, 2^n, n^2, n!, n, n^3, log n

Listed from slowest to fastest growth:

1) 1
2) log n
3) n
4) n log n
5) n^2
6) n^3
7) 2^n
8) n!



The Growth of Functions

	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
$O(1)$	$1\mu s$	$1\mu s$	$1\mu s$	$1\mu s$	$1\mu s$
$O(log(n))$	$6\mu s$	$10\mu s$	$13\mu s$	$17\mu s$	$20\mu s$
$O(n)$	$0.1ms$	$1ms$	$10ms$	$0.1s$	$1s$
$O(n\ log(n))$	$0.6ms$	$10ms$	$0.1s$	$1.6s$	$19.9s$
$O(n^2)$	$10ms$	$1s$	$100s$	$2.7h$	$11.5j$
$O(n^3)$	$1s$	$16.6min$	$11.5j$	$32a$	$32 * 10^3 a$
$O(2^n)$	$4 * 10^{16} a$	∞	∞	∞	∞