# Worksheet 1: Analysis and design of algorithms

**Task 1**

1. (a) Complete the following table to show that as n becomes large, in a function

f(n) = 5n2 + 10n + 2

only the n2 term has a significant effect on the size of f(n)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **n** | **n2** | **5n2** | **10n** | **2** | **f(n)= 5n2+ 10n + 2** |
| **10** |  |  |  | 2 |  |
| **100** |  |  |  | 2 |  |
| **1000** |  |  |  | 2 |  |
| **10,000** |  |  |  | 2 |  |

(b) What is the time complexity of an algorithm that has 5n2 + 10n + 2 steps, expressed in the Big-O notation?

2. Calculate the number of assignment statements in each of the following pseudocode fragments. Hence calculate the Big-O time complexity of each algorithm.

(a) a = 1000

for i = 1 to n

x = a + i

next i

(b) for i = 0 to n

for j = 0 to n

k = n\*2 + i \* j

next j

next i

(c) for i = 1 to n

x = 5 + i \* i

next i

for j = 1 to n

y = 10 – j

next j

**Task 2**

1. (a) How many different permutations would have to be checked using a “brute force” method, on average, to crack a password, if it is known to be 4 uppercase alphabetic characters?

(b) How many would have to be checked, in the worst case scenario?

(c) What is the Big-O time complexity of this algorithm?

2. Look at the following graphs. A graph which has an edge connecting every vertex is said to be “complete”.

How many edges are there in each graph?

Verify that for a graph of n vertices, there are n(n-1)/2 edges.

This is the same problem as the following:

“There are n people at a party. Each person shakes hands with every other person. How many handshakes take place?”

(a) Assuming that each traversal of an edge is just one operation, what is the time complexity of traversing each edge in a “complete” graph just once?

(b) A computer game has 5 doors which have to be opened in a particular sequence in order to progress to the next step. In how many different orders can the doors be opened?

(c) What is the order of complexity of the problem if there are n doors?

(d) Suppose there were 7 doors. How many doors would need to be opened, on average, to find the correct door using a “Brute Force” method?

3. In the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8 … , each number after the first two is the sum of the two previous numbers.

What will be the next two numbers in the sequence?

Here are two subroutines written in Python each designed to find the first n numbers in the Fibonacci series.

**Subroutine 1: (a recursive routine)**

def fibonacci(n):

if n == 0:

return 0

if n == 1:

return 1

return fibonacci(n-1) + fibonacci(n-2)

**Subroutine 2: (an iterative routine)**

def fibonacci2(n):

fibNumbers = [0,1] *#list of first two Fibonacci numbers*

*# now append the sum of the two previous numbers to the list*

for i in range(2, n+1):

fibNumbers.append(fibNumbers[i-1] + fibNumbers[i-2])

return fib[n]

Which subroutine do you think has the lower time complexity?

If you can, run the Python program *Fibonacci.py*, or write your own code in a language you are familiar with, fill in the times in the following table:

|  |  |  |
| --- | --- | --- |
|  | **Time to execute recursive subroutine (milliseconds)** | **Time to execute iterative subroutine (milliseconds)** |
| **n = 10** |  |  |
| **n = 20** |  |  |
| **n = 25** |  |  |
| **n = 30** |  |  |
| **n = 35** |  |  |
| **n = 40 (estimate)** |  |  |
| **N = 100 (estimate)** |  |  |

Which of O(n) O(n2) O(2n) O (log n) is the approximate time complexity of:

1. the recursive subroutine?
2. the iterative subroutine?

**Quick quiz on Big-O notation**

Tick the statements which are correct:

(a) The aim of the Big-O notation is to give a rough idea of how time and/or memory requirements will grow as the problem gets bigger

(b) The statement *“The worst case run-time complexity of algorithm A is O(n2)”* means that *“Algorithm A takes at most c x n2 steps (where c is a constant) to solve a problem of size n (for large n)”*

(c) Problems of complexity O(1) have only one statement, however large the problem

(d) A problem with complexity O(100n) is of a different order of magnitude from one of complexity O(n)

(e) An algorithm of complexity O(n3) is useless for any practical purpose

(f) Hashing is an example of a problem of time complexity O(1)

(g) Some problems may be O(n) for small values of n but O(n2) for large values of n

(h) “Divide and conquer” algorithms typically have time complexity O(log n) and are   
very efficient