How can we measure how good an algorithm is?

1. Give two different resources used by an algorithm that we might measure? [2]

Solution:

- Time
- Memory

2. Why do we usually measure the time taken by an algorithm rather than the time taken by an implementation of an algorithm? [2]

Solution:

- Not affected by language
- Not affected by hardware

3. What are the basic general principles behind measuring the time taken by some algorithm, expressed using pseudo-code, on some particular input? [3]

Solution:
It allows you to be language independent and not rely on features implemented in a specific programming language

4. In pseudo-code we assume that any 'basic' instruction takes c units of time to execute. What is this constant c supposed to reflect? [2]

Solution:
The time to do one operation on a given processor

5. What is the worst-case time complexity of an algorithm? Why is it expressed as a function on the natural numbers? [4]

Solution:

- Can't give a fixed number as a bigger input will always take more time
- The greatest amount of time that an algorithm will take for an input
- How the function reacts as the input increases

6. Define precisely what we mean when we say that two functions $f(n) : \mathbb{N} \to \mathbb{N}$ and $g(n) : \mathbb{N} \to \mathbb{N}$ are such that $f = O(g)$ [2]

Solution:

There exists $n_0 \in \mathbb{N}$ and $k \in Q$ such that $f(n) \leqslant k \cdot g(n)$ wherever $n \geqslant n_0$

7. Consider the following functions: [6]

- $f_1(n) = 345n^2 - n + 1$
- $f_2(n) = n^4 - n^2$
- $f_3(n) = \frac{2^n}{1000}$
- $f_4(n) = 10n^2 \log(n)$

For each pair of functions f_i and f_j, where $i \neq j$, say whether $f_i = O(f_j)$

Solution:

$$f_1 = O(f_2)$$
$$f_1 = O(f_3)$$
$$f_1 \neq O(f_4)$$

$$f_2 \neq O(f_1)$$
$$f_2 = O(f_3)$$
$$f_2 \neq O(f_4)$$

$$f_3 \neq O(f_1)$$
$$f_3 \neq O(f_2)$$
$$f_3 \neq O(f_4)$$

$$f_4 \neq O(f_1)$$
$$f_4 = O(f_2)$$
$$f_4 = O(f_3)$$

8. Suppose that you had two algorithms to solve some problem that had time complexities $O(n^3)$ and $O(n^2)$ and two computers, one of which was 100 times faster than the other. Suppose also that you could only run the slower algorithm on the fast machine and the faster algorithm on the slower machine. Say which configuration you would choose to solve your problem and why [4]

Solution:

n^3 will overtake n^2 and will take longer to run. For small inputs n^3 might be faster, but for large values n^2 will be faster, even on the slower machine

9. Define precisely what we mean when we say that two functions $f(n) : \mathbb{N} \to \mathbb{N}$ and $g(n) : \mathbb{N} \to \mathbb{N}$ are such that $f = \Omega(g)$ What does it mean in practice if we say that any algorithm solving the sorting problem has time complexity $\Omega(n \log n)$? [4]

Solution:

If there exists $n_0 \in \mathbb{N}$ and $k \in Q$ such that $g(n) \geqslant k \cdot f(n)$ wherever $n \geqslant n_0$. The algorithm can't take more than $n \log n$ steps

10. Define precisely what we mean when we say that two functions $f(n) : \mathbb{N} \to \mathbb{N}$ and $g(n) : \mathbb{N} \to \mathbb{N}$ are such that $f = \Theta(g)$. What does it mean in practice if we say that solving the sorting problem has time complexity $\Theta(n \log n)$ [4]

Solution:

$f = O(g)$ and $f = \Omega(g)$
For all values of n, the time taken will scale with $n \log n$

11. The algorithm Bubble-sort is as follows: [5]

```
change = true
WHILE change == true:
change = false
i = 0
WHILE i < n - 1:
IF A[i] > A[i + 1]:
swap the numbers in A[i] and A[i + 1]
change = true
i = i + 1
output A
```

What is the time complexity of bubble-sort? (you should explain your answer)

Solution:

- The inner while loop will loop n-1 times
- The outer loop will loop n-1 times before no swaps happen
- $O((n-1)n + n) = O(n^2)$
- Best case $\Omega(n)$

12. The algorithm Selection-sort is as follows: [6]

```
pass = 0
WHILE pass < n - 1:
x = A[pass]
i = pass + 1
WHILE i ⩽ n - 1:
IF A[i] < x:
```

```
swap x and A[i]
i = i + 1
A[pass] = x
pass = pass + 1
output A
```

What is the time complexity of Selection-sort? (You should explain your answer)

Solution:

- Assuming all instructions take constant time
- The dominating loop is n-1 of the outer while loop
- The inner while loop runs n-1 times
- So the complexity is $(n-1)(n-1)+n$ which is $O(n^2)$

13. Briefly explain the primary focus of the research area known as algorithm engineering. [2]

Solution:
Want fast or memory efficient algorithms, depending on the available resources