# DVA104

## Data Structures, Algorithms, and Program Development


Gabriele Capannini

gabriele.capannini@mdh.se

Mälardalen University


October 8, 2018

# Example

Assume the array `A` is longer than `1000`, how long does it take to run `f1`?

```
int f1(int * A) {
  int s=0;
  for(int i=0; i<1000; i=i+1)
    s = s+A[i];
  return s;
}
```

It depends. . . On what?

# What affects the execution time in a function?

- Hardware (HW)
  - Processor,
  - Memory (bandwidth, latency, cache, . . . )
  - . . .
- Software (SW)
  - Operating System
  - The compiler
  - The program itself (loops, recursion, etc.)

# Example

What if we measure the execution time of `f1` assuming to run it many times in the same identical HW/SW conditions?

```
int f1(int * A) {
  int s=0;
  for(int i=0; i<1000; i=i+1)
    s = s+A[i];
  return s;
}
```

Even in this case we can observe that times vary slightly!

There are some other random factors that we can't control.

# Computational Models

Similarly to what we have done with ADTs, we can use an abstraction that does not consider these *unwanted* details of the problem: HW, SW, and all the other possible factors affecting the performance.

We need an abstract executor, aka computational models.

For example, the Random Access Machine (RAM) is an abstraction of a real computer with:

- an unbounded amount of memory,
- a processor which executes every type of operation (assignments, calculations, comparisons, memory accesses, . . . ) in one unit of time.

# Time Complexity

Let's calculate the time complexity of `f1` by means of the RAM computational model: how many time units $t$ are required to run it:

```
int f1(int* A) {
  int s=0; //1t
  for(int i=0; i<1000; i=i+1) //1t+1t×1001+2t×1001
    s = s+A[i]; //2t×1000
  return s;
}
```

It turns out that the execution time is $5004\,t$. Note that this evaluation is not affected from other factors than the code.

What if, instead of `1000`, we had `10000` iterations?

# Time Complexity (continued)

What is the time complexity of the following function?

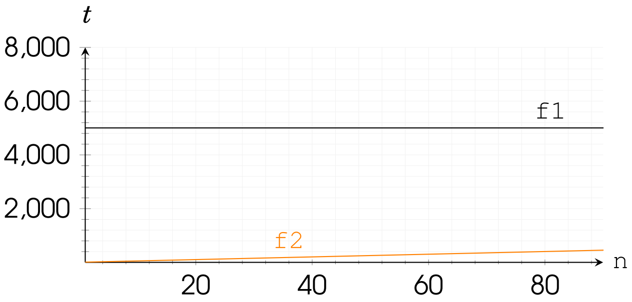
```
int f2(int* A, int n) {
  int s=0; //1t
  for(int i=0; i<n; i=i+1) //1t+1t×(n+1)+2t×(n+1)
    s = s+A[i]; //2t×n
  return s;
}
```

It turns out that the execution time of `f2` is $(5n+4)t$.

In this case the time complexity depends on the variable `n` which is a parameter of the computation.

# Time Complexity: graphically

Here we compare the time complexity of the two functions:




The time complexity of functions like `f1` is called constant since it doesn't vary.

The time complexity of functions like `f2` is called linear in `n` since it grows as `a×n+b` where `a=5` and `b=4`.

# Time Complexity: quatratic

There are also function with quadratic time complexity:

```
int f3(int** A, int n) {
  int s=0; //1t
  for(int i=0; i<n; i=i+1) //3t+3t×n
    for(int j=0; j<n; j=j+1) //(3t+3t×n)×n
      s = s+A[i][j]; //2n^2 t
  return s;
}
```

The time complexity equals ($t$ can be omitted):

$$1+(3n+3)(n+1)+2n^2 = 5n^2+6n+4$$

which can be described by the formula:

$$a \times n^2 + b \times n + c$$

with $a=5, b=6$ and $c=4$.

# Time Complexity: exponential

There are also functions with exponential time complexity:

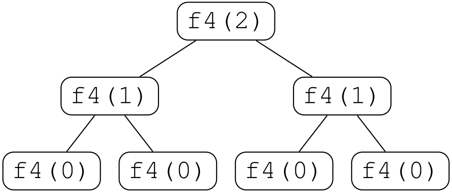
```
int f4(int n) {
  if(n>0) //1t
    return f4(n-1)+f4(n-1); //???
  else
    return 1;
}
```

Let $T_{\mathtt{f}}(\mathtt{n})$ the function which defines the time complexity of the function `f` computed on `n`: for example, $T_{\mathtt{f3}}(\mathtt{n}) = 5\mathtt{n}^2+6\mathtt{n}+4$.

$$T_{\mathtt{f4}}(\mathtt{n}) = \begin{cases} 2T_{\mathtt{f4}}(\mathtt{n}-1) + 3, & \text{if } \mathtt{n}>0 \\ 1, & \text{else} \end{cases}$$

# Time Complexity: exponential (continued)

For example, a sketch of running `f4(2)` looks like this:




so that its time complexity equals:

$$T_{\texttt{f4}}(2) = 2T_{\texttt{f4}}(1)+3 = 2(2T_{\texttt{f4}}(0)+3)+3 = 2(2(1)+3)+3 = 4+6+3$$

In the general case, we have:

$$T_{\texttt{f4}}(\texttt{n}) = 2^{\texttt{n}} + 3(\underbrace{2^{\texttt{n}-1} + 2^{\texttt{n}-2} + \ldots + 1}_{\sum_{i=0}^{\texttt{n}-1} 2^i = 2^n - 1}) = 4\times 2^{\texttt{n}} - 3$$

# Time Complexity: logarithmic

There are also functions with sublinear time complexity:

```
int f5(int n) {
  if(n>0) //1t
    return 1+f5(n/2); //???
  else
    return 0;
}
```

The time complexity equals:

$$T_{\mathtt{f5}}(\mathtt{n}) = \begin{cases} T_{\mathtt{f5}}(\frac{\mathtt{n}}{2}) + 2, & \text{if } \mathtt{n}{>}0 \\ 1, & \text{else} \end{cases}$$

# Time Complexity: logarithmic (continued)

For example, a sketch of running `f4(5)` looks like this:

$$\boxed{\texttt{f5}(5)} - \boxed{\texttt{f5}(2)} - \boxed{\texttt{f5}(1)} - \boxed{\texttt{f5}(0)}$$

so that its time complexity equals:

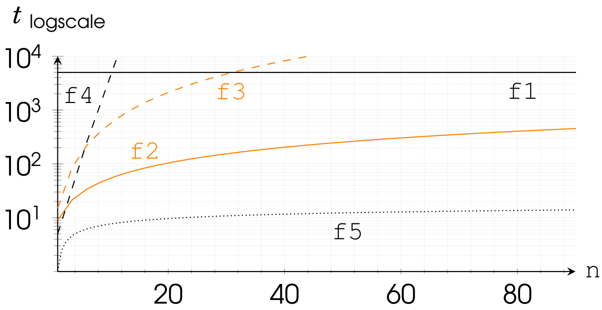
$$T_{\texttt{f5}}(5) = 2 + T_{\texttt{f5}}(2) = 2+2+T_{\texttt{f5}}(1) = 2+2+2+T_{\texttt{f5}}(0) = 2+2+2+1$$

In the general case, we have:

$$T_{\texttt{f5}}(\texttt{n}) = \underbrace{2+2+\ldots+2}_{\lfloor \log_2(\texttt{n}) \rfloor + 1 \text{ times}} + 1 = 2(\lfloor \log_2(\texttt{n}) \rfloor + 1) + 1$$

This kind of time complexity is called logarithmic.

# Time Complexity: recap




Note: functions like `f1`, `f2`, and `f3` are called polynomial time algorithms since their complexity is described by a polynomial.

A polynomial of degree *d* is can be expressed like: $\sum_{i=0}^{d} a_i \cdot n^i$

Examples: `d=0` constant, `d=1` linear, `d=2` quadratic, `d=3` cubic

# Time Complexity: test

1. What's the time complexity of a function searching for a given value. . .
   (a) . . . in an array?
   (b) . . . in a Linked List (LL)?
   (c) . . . in a balanced Binary Search Tree (BST)?

2. Given that `f4` has exponential time complexity, what is the complexity of `f4bis`?

```
int f4bis() {
  return f4(1000);
}
```

# Time Complexity: recap & examples

| Time complexity | Example |
|---|---|
| Constant | Evaluate an arithmetic expression |
| Logarithmic | Binary search on $n$ sorted values |
| Linear | Sum of $n$ values |
| Quadratic | Populate a matrix $n \times n$ |
| Exponential | Create a BST of $n$ levels |

increasing time complexity

# Big-O-notation

In this course we talked about `abstraction´ as a mean to simplify a problem by skipping the unnecessary details.

Can we do the same with Time Complexity?

**Yes!**

Big-O-notation is the answer: it allows us to classify algorithms based on their performance on large inputs.

# Big-O-notation: example #1

Let's consider `f3` again:

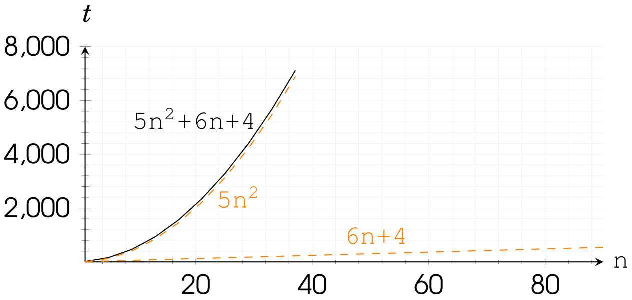
```
int f3(int** A, int n) {
  int s=0;
  for(int i=0; i<n; i=i+1)
    for(int j=0; j<n; j=j+1)
      s = s+A[i][j];
  return s;
}
```

Its complexity $T(n) = 5n^2+6n+4$ is given by the sum of 3 terms:

(a) a quadratic term: $5n^2$,

(b) a linear term: $6n$,

(c) a constant term: $4$.




There is a dominant term in the formula: $5n^2$.

We can state that the time complexity of this function grows proportionally to $n^2$ by discarding the coefficient $a=5$ and the remaining terms of the polynomial with lower degree.

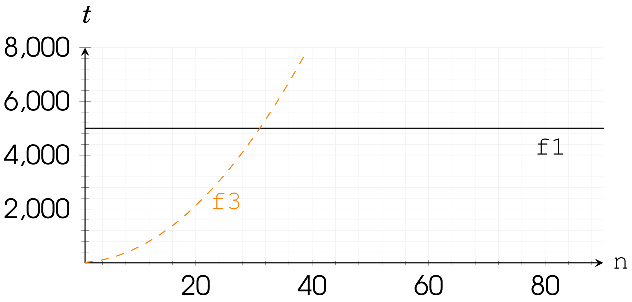
# Big-O-notation: example #2

Let us consider the previous functions `f1` and `f3` with time complexities $T_{\texttt{f1}}() = 5004$ and $T_{\texttt{f3}}(\mathtt{n}) = \mathtt{5n^2+6n+4}$, respectively.

| $\mathtt{n}$ | $T_{\texttt{f1}}()$ | $T_{\texttt{f3}}(\mathtt{n})$ |
|---|---|---|
| 1 | 5004 | 15 |
| 2 | 5004 | 36 |
| 3 | 5004 | 67 |
| 4 | 5004 | 108 |
| 5 | 5004 | 159 |
| ... | ... | ... |

It seems that `f3` behaves better than `f1`... However `f3` belongs to a worse complexity class than than `f1`! Why?

# Big-O-notation: example #2 (continued)




While $T_{\texttt{f1}}$ is constant (*it doesn't matter how large $n$ is*), `f3` requires more and more time as `n` grows.

# Big-O-notation: definition

Big-O-notation tells us the order of the maximum number of operations that might be performed when problem size grows.

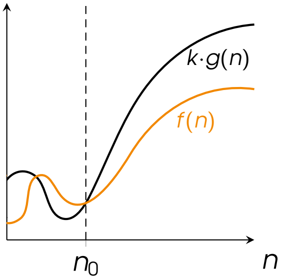
Its formal definition is:

Let the $f(n)$ and $g(n)$ be two functions. We say that

$$f(n) \in O(g(n))$$

if there exist two constants $n_0$ and $k$ such that

$f(n) \leq k \cdot g(n)$ for each $n \geq n_0$



# Big-O-notation: recap & examples

| f | Time complexity | Order |
|---|---|---|
| `f1()` | Constant | $O(1)$ |
| `f5(n)` | Logarithmic | $O(\log n)$ |
| `f2(n)` | Linear | $O(n)$ |
| `f3(n)` | Quadratic | $O(n^2)$ |
| ... | Cubic | $O(n^3)$ |
| ... | ... | $O(n^d)$ |
| `f4(n)` | Exponential | $O(2^n)$ |

increasing time complexity

# Big-O-notation: sum-rule

« If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 + f_2 \in O(\max\{g_1, g_2\})$ »

What is the time complexity of the following function?

```
int f6(int n) {
  int s=0;
  for(int i=0; i<n; i=i+1)
    s = s+i;
  for(int i=0; i<10000; i=i+1)
    s = s/(i+3);
  return s;
}
```

# Big-O-notation: product-rule

« If $f_1 \in O(g_1)$ and $f_2 \in O(g_2)$, then $f_1 \cdot f_2 \in O(g_1 \cdot g_2)$ »

What is the time complexity of the following function?

```
int f7(int n) {
  int s=0;
  for(int i=0; i<n; i=2*i)
    for(int j=0; j<n; j=j+1)
      s = s+i;
  return s;
}
```

# Big-O-notation: final remarks

- Big-O-notation allow us to easily compare two algorithms:
  - $O(T_{\texttt{f2}}) < O(T_{\texttt{f3}}) \Rightarrow$ "`f2()` is faster than `f3()`"
- To calculate Big-O function of an algorithm can be easily done (in general) just by looking (carefully) at the code.
  - For example, looking at `f3()` it's enough to observe that the most frequent calculation is performed $\texttt{n}^2$ times to state that the complexity of the whole function.
- Big-O function gives only an asymptotic upper bound of a function. However we are interested in to find the most tight possible one.
  - For example, to state that $T_{\texttt{f1}}$ is quadratic because $T_{\texttt{f2}} < n^2$ is technically correct, but not terribly precise.
  - There are similar notations (e.g., Θ-notation) that force us to be more precise.

# Question

In light of the previous definition, what is the time complexity of the following function?

```
void foo(int** A, int n) {
  int p = rand()%100;
  if(p%2==0) {
    for(int i=0; i<n; ++i)
      for(int j=0; j<n; ++j)
        A[i][j] = p;
  } else {
    for(int i=0; i<n; ++i)
      A[i][i] = p;
  }
}
```