Computational Complexity of Algorithms

Complexity of an Algorithm

- Computers work fast but not infinitely fast. Execution of each instruction takes some (very short) time.
- The same problem can be solved by several different algorithms. The time of a computation (determined mostly by the number of executed instructions) can be different for different algorithms.
- We would like to compare different algorithms and choose a better one.
- We can implement the algorithms and then measure the time of their computation. By this we find out how long the computation takes on particular data on which we test the algorithm.
- We would like to have a more precise idea how long the computation takes on all possible input data.

Complexity of an Algorithm

- A running time is affected by many factors, e.g.:
  - the algorithm that is used
  - the amount of input data
  - used hardware (e.g., the frequency at which a CPU is running can be important)
  - the used programming language — its implementation (compiler/interpreter)
  - ...
- If we need to solve problem for "small" input data, the running time is usually negligible.
- With increasing amount of input data (the size of input), the running time can grow, sometimes significantly.

Complexity of an Algorithm

- **Time complexity of an algorithm** — how the running time of the algorithm depends on the amount of input data
- **Space complexity of an algorithm** — how the amount of a memory used during a computation grows with respect to the size of input

Remark: The precise definitions of these notion will be given in a moment.

Remark:

- There are also other types of computational complexity, which we will not discuss here (e.g., communication complexity).

Complexity of an Algorithm

Consider some particular machine executing some algorithm — e.g., a random-access machine, a Turing machine, ...

We will assume that for the given machine $\mathcal{M}$ we have somehow defined for every input w from the set of all possible inputs In the following two functions:

- $time_{\mathcal{M}} : In \to \mathbb{N}$ — it expresses the running time of machine $\mathcal{M}$ on input w
- $space_{\mathcal{M}} : In \to \mathbb{N}$ — it expresses the amount of memory used by machine $\mathcal{M}$ in a computation on input w

Remark: We assume that a computation on an arbitraty input w will halt after some finite number of steps.

Complexity of an Algorithm

Example:

- One-tape Turing machine $\mathcal{M}$:
  - $time_{\mathcal{M}}(w)$ — the number of steps performed by during a computation on word w
  - $space_{\mathcal{M}}(w)$ — the number of cells on the tape visited during a computation on input w

- Random-access machine:
  - $time_{\mathcal{M}}(w)$ — the number of steps performed by the given RAM in a computation on input w
  - $space_{\mathcal{M}}(w)$ — the number of memory cells that were used during a computation on input w (in they were written to or if a value was read from them)

Size of Input

For different input data the program performs a different number of instructions.

If we want to analyze somehow the number of performed instructions, it is useful to introduce the notion of the **size of an input**.

Typically, the size of an input is a number specifying how "big" is the given instance (a bigger number means a bigger instance).

Remark: We can define the size of an input as we like depending on what is useful for our analysis.

The size of an input is not strictly determinable but there are usually some natural choices based on the nature of the problem.

Size of Input

Examples:

- For the problem "Sorting", where the input is a sequence of numbers $a_1, a_2, \ldots, a_n$ and the output the same sequence sorted, we can take n as the size of the input.
- For the problem "Primality" where the input is a natural number x and where the question is whether x is a prime, we can take the number of bits of the number x as the size of the input.

  (The other possibility is to take directly the value x as the size of the input.)

Size of Input

Sometimes it is useful to describe the size of an input with several numbers.

For example for problems where the input is a graph, we can define the size of the input as a pair of numbers n, m where:

- n – the number of nodes of the graph
- m – the number of edges of the graph

Remark: The other possibility is to define the size of the input as one number $n + m$.

Size of Input

In general, we can define the size of an input for an arbitrary problem as follows:

- When the input is a word over some alphabet Σ:
  the length of word w
- When the input as a sequence of bits (i.e., a word over $\{0, 1\}$):
  the number of bits in this sequence
- When the input is a natural number x:
  the number of bits in the binary representation of x

Time Complexity

We want to analyze a particular algorithm (its particular implementation).

We want to know how many steps the algorithm performs when it gets an input of size $0, 1, 2, 3, 4, \ldots$.

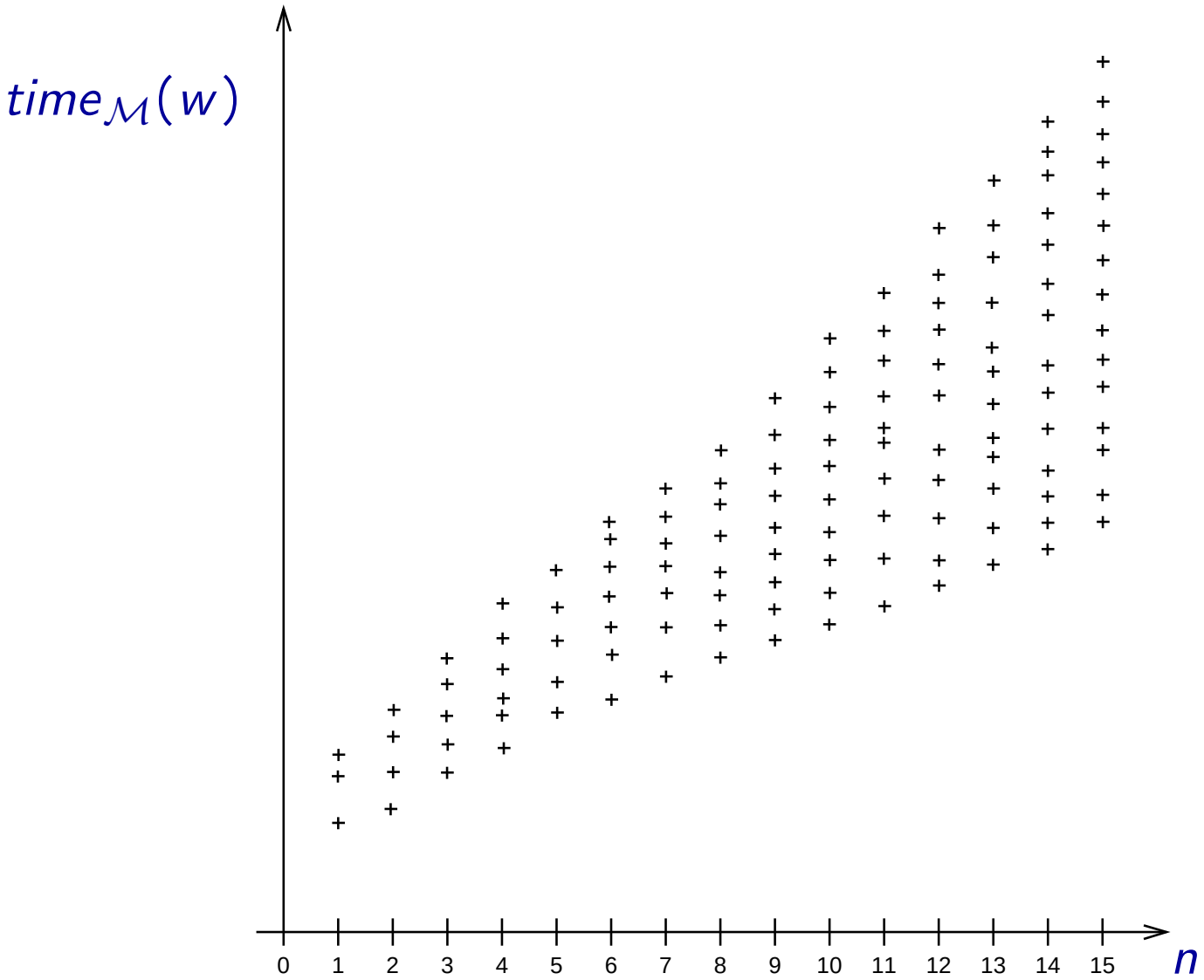
It is obvious that even for inputs of the same size the number of performed steps can be different.

Let us denote the size of input $w \in In$ as $size(w)$.

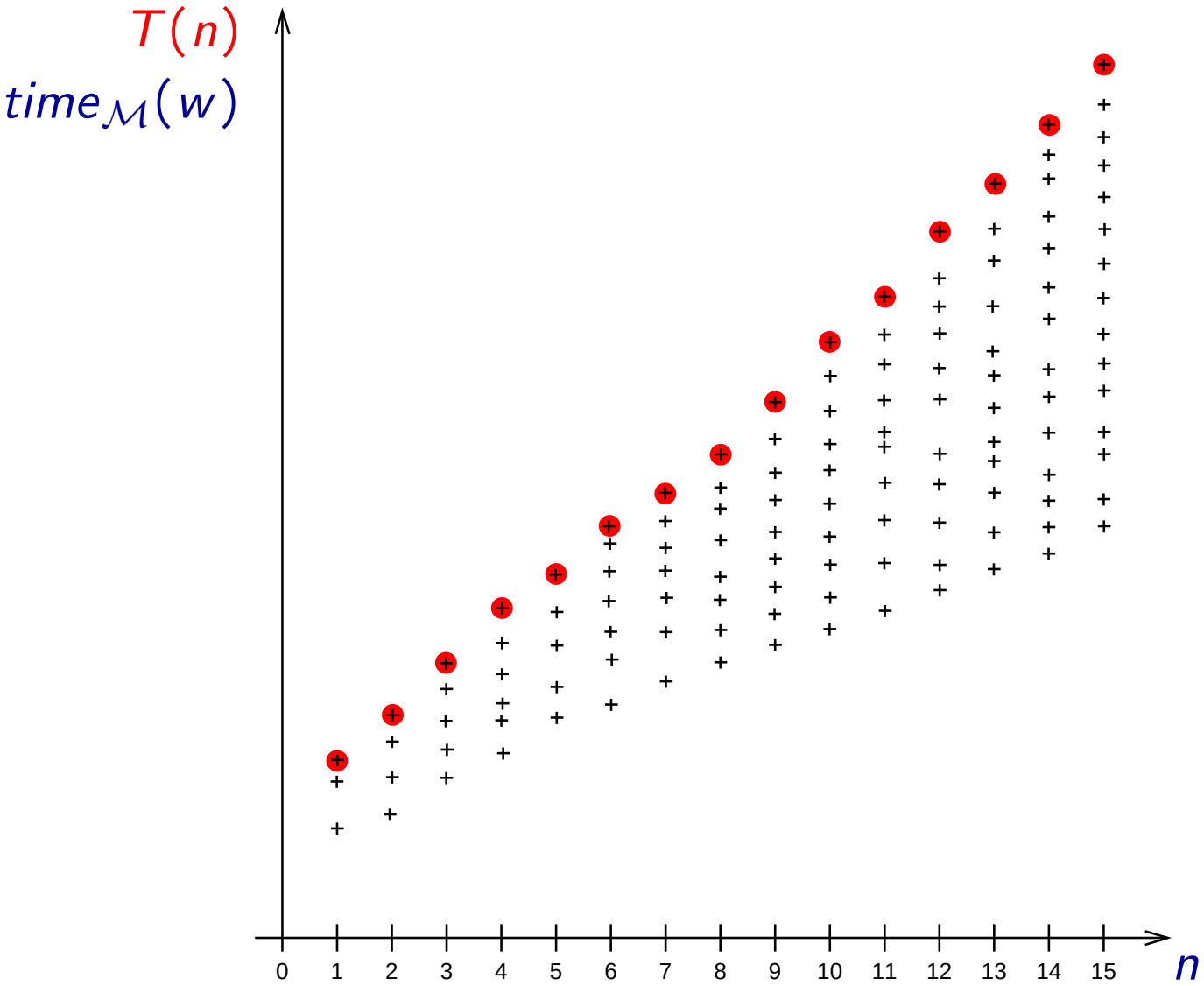
Now we define a function $T : \mathbb{N} \to \mathbb{N}$ such that for $n \in \mathbb{N}$ is

$$T(n) = \max\{\, time_{\mathcal{M}}(w) \mid w \in In,\ size(w) = n \,\}$$

Time Complexity in the Worst Case



Time Complexity in the Worst Case



Time Complexity in the Worst Case

Such function $T(n)$ (i.e., a function that for the given algorithm and the given definition of the size of an input assignes to every natural number n the maximal number of instructions performed by the algorithm if it obtains an input of size n) is called the **time complexity of the algorithm in the worst case**.

$$T(n) = \max\{\, time_{\mathcal{M}}(w) \mid w \in In,\ size(w) = n \,\}$$

Analogously, we can define **space complexity** of the algorithm in the worst case as a function $S(n)$ where:

$$S(n) = \max\{\, space_{\mathcal{M}}(w) \mid w \in In,\ size(w) = n \,\}$$

Time Complexity in an Average Case

Sometimes it make sense to analyze the time complexity **in an average case**.

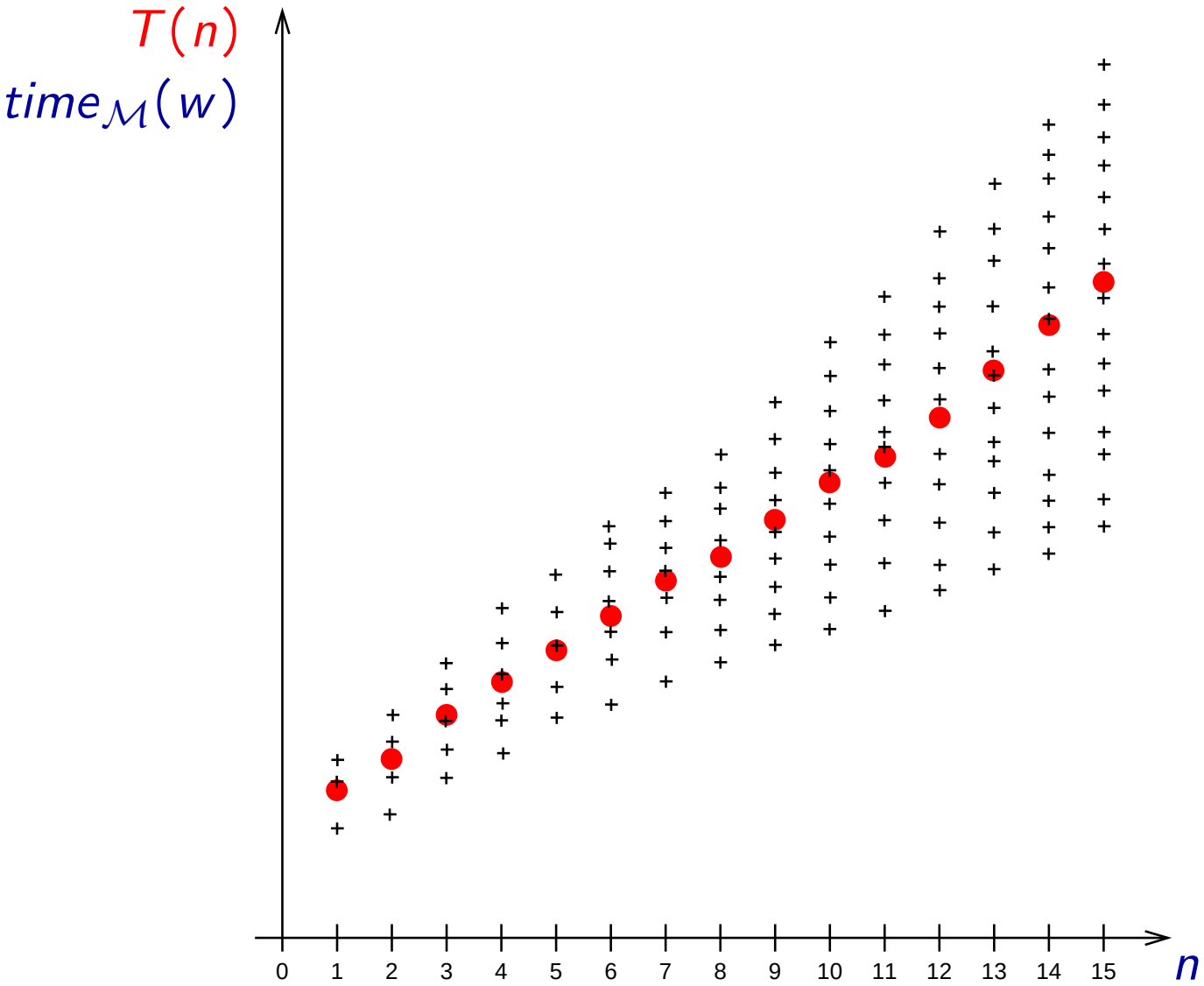
In this case, we do not define $T(n)$ as the maximum but as the arithmetic mean of the set

$$\{ time_{\mathcal{M}}(w) \mid w \in In, size(w) = n \}$$

- It is usually more difficult to determine the time complexity in an average case than to determine the time complexity in the worst case.
- Often, these two function are not very different but sometimes the difference is significant.

Remark: It usually makes no sense to analyze the time complexity in the best case.

Time Complexity in an Average Case



Computational Complexity of an Algorithm

It is obvious from this definition that the time complexity of an algorithm is a function whose precise values depend not only on the given algorithm Alg but also on the following things:

- on a machine $\mathcal{M}$, on which the algorithm Alg runs,
- on the precise definition of the running time $t(w)$ of algorithm Alg on machine $\mathcal{M}$ with input $w \in In$,
- on the precise definition of the size of an input (i.e., on the definition of function $size$).

Computational Complexity of an Algorithm

To determine the precise running time or the precise amount of used memory just by an analysis of an algorithm can be extremely difficult.

Usually the analysis of complexity of an algorithm involves many simplifications:

- It is usually not analysed how the running time or the amount of used memory depends precisely on particular input data but how they depend on the **size of the input**.
- Functions expressing how the running time or the amount of used memory grows depending on the size of the input are not computed precisely — instead **estimations** of these functions are computed.
- Estimations of these functions are usually expressed using **asymptotic notation** — e.g., it can be said that the running time of MergeSort is $O(n \log n)$, and that the running time of BubbleSort is $O(n^2)$.

Time Complexity of an Algorithm

An example of an analysis of the time complexity of algorithm **without** the use of asymptotic notation:

- Such precise analysis is almost never done in practice — it is too tedious and complicated.
- This illustrates what things are ignored in an analysis where asymptotic notation is used and how much the analysis is simplified by this.
- We will compute with constants $c_0, c_1, \ldots, c_k$, which specify the execution time of individual instructions — we won't compute with concrete numbers.

Running Time

Let us say that an algorithm is represented by a control-flow graph:

- To every instruction (i.e., to every edge) we assign a value specifying how long it takes to perform this instruction once.
- The execution time of different instructions can be different.
- For simplicity we assume that an execution of the same instruction takes always the same time — the value assigned to an instruction is a number from the set $\mathbb{R}_+$ (the set of nonnegative real numbers).

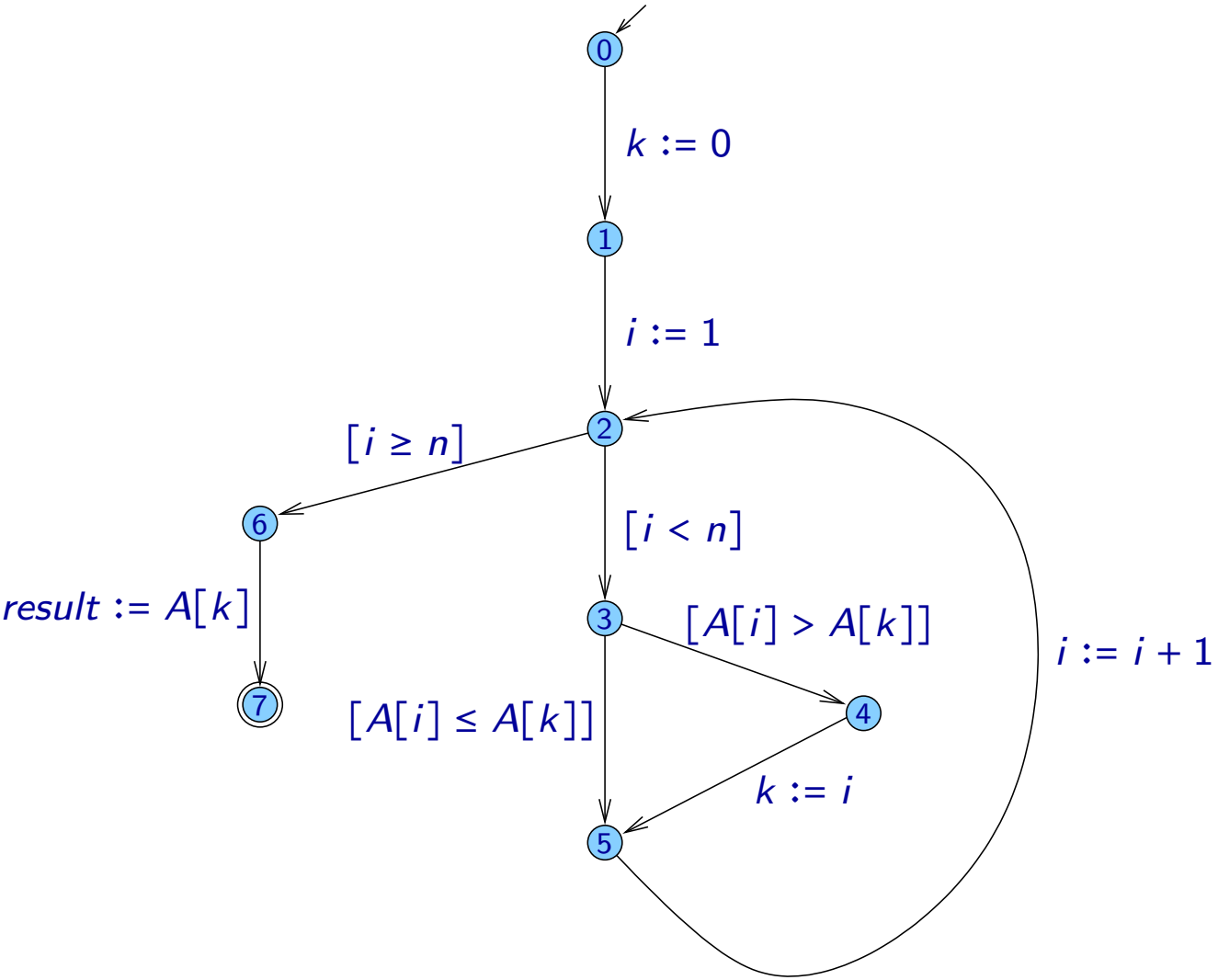
Running Time

Example:

Algorithm: Finding the maximal element in an array

```
FIND-MAX (A, n):
    k := 0
    for i := 1 to n − 1 do
        if A[i] > A[k] then
            k := i
    return A[k]
```

Running Time



Instr.	time
$k := 0$	c_0
$i := 1$	c_1
$[i < n]$	c_2
$[i \geq n]$	c_3
$[A[i] \leq A[k]]$	c_4
$[A[i] > A[k]]$	c_5
$k := i$	c_6
$i := i + 1$	c_7
$result := A[k]$	c_8

Running Time

Example: The execution times of individual instructions could be for example:

Instr.	symbol	time
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

For a particular input w, e.g., for $w = ([3, 8, 4, 5, 2], 5)$, we could simulate the computation and determine the precise running time $t(w)$.

Time Complexity of an Algorithm

The inputs are of the form (A, n), where A is an array and n is the number of elements in this array (where $n \geq 1$).

We take n as the size of input (A, n).

Consider now some particular input $w = (A, n)$ of size n:

- The running time $t(w)$ on input w can be expressed as

$$t(w) = c_0 \cdot m_0(w) + c_1 \cdot m_1(w) + \cdots + c_8 \cdot m_8(w),$$

where $m_0, m_1, \ldots, m_8$ are functions specifying how many times is each instruction performed in the computation on input w.

Time Complexity of an Algorithm

Instr.	time	occurences	value of $m_i(w)$
$k := 0$	c_0	$m_0(w)$	1
$i := 1$	c_1	$m_1(w)$	1
$[i < n]$	c_2	$m_2(w)$	$n-1$
$[i \geq n]$	c_3	$m_3(w)$	1
$[A[i] \leq A[k]]$	c_4	$m_4(w)$	$n-1-\ell$
$[A[i] > A[k]]$	c_5	$m_5(w)$	ℓ
$k := i$	c_6	$m_6(w)$	ℓ
$i := i+1$	c_7	$m_7(w)$	$n-1$
$result := A[k]$	c_8	$m_8(w)$	1

ℓ — the number of iterations of the cycle where $A[i] > A[k]$
(obviously $0 \leq \ell < n$)

Time Complexity of an Algorithm

By assigning values to

$$t(w) \;=\; c_0 \cdot m_0(w) + c_1 \cdot m_1(w) + \;\cdots\; + c_8 \cdot m_8(w),$$

we obtain

$$t(w) \;=\; d_1 \;+\; d_2 \cdot (n-1) \;+\; d_3 \cdot (n-1-\ell) \;+\; d_4 \cdot \ell,$$

where

$$d_1 = c_0 + c_1 + c_3 + c_8 \qquad d_3 = c_4$$
$$d_2 = c_2 + c_7 \qquad d_4 = c_5 + c_6$$

After simplification we have

$$t(w) \;=\; (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot \ell \;+\; (d_1 - d_2 - d_3)$$

Remark: $t(w)$ is not the time complexity but the running time for a particular input w

Time Complexity of an Algorithm

For example, if the execution times of instructions will be:

Instr.	symb.	time
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

then $d_1 = 25$, $d_2 = 16$, $d_3 = 14$, and $d_4 = 17$.

In this case is $t(w) = 30n + 3\ell - 5$.

For the input $w = ([3, 8, 4, 5, 2], 5)$ is $n = 5$ and $\ell = 1$, therefore $t(w) = 30 \cdot 5 + 3 \cdot 1 - 5 = 148$.

Time Complexity of an Algorithm

It can depend on details of implementation and on the precise values of constants, for which inputs of size n the compution takes the longest time (i.e., which are the worst cases):

The running time of algorithm FIND-MAX for an input $w = (A, n)$ of size n:

$$t(w) \;=\; (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot \ell \;+\; (d_1 - d_2 - d_3)$$

- If $d_3 \geq d_4$ — the worst cases are those where ℓ has the smallest value $\ell = 0$ — for example inputs of the form $[0, 0, \ldots, 0]$ or of the form $[n,\ n-1,\ n-2,\ \ldots,\ 2,\ 1]$
- If $d_3 \leq d_4$ — the worst are those cases where ℓ has the greatest value $\ell = n - 1$ — for example inputs of the form $[0, 1, \ldots, n-1]$

Time Complexity of an Algorithm

The time complexity $T(n)$ of algorithm FIND-MAX in the worst case is given as follows:

- If $d_3 \geq d_4$:

$$T(n) = (d_2 + d_3) \cdot n + (d_1 - d_2 - d_3)$$

- If $d_3 \leq d_4$:

$$\begin{aligned} T(n) &= (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot (n-1) + (d_1 - d_2 - d_3) \\ &= (d_2 + d_4) \cdot n + (d_1 - d_2 - d_4) \end{aligned}$$

Example: For $d_1 = 25$, $d_2 = 16$, $d_3 = 14$, $d_4 = 17$ is

$$\begin{aligned} T(n) &= (16 + 17) \cdot n + (25 - 16 - 17) \\ &= 33n - 8 \end{aligned}$$

Time Complexity of an Algorithm

In both cases (when $d_3 \geq d_4$ or when $d_3 \leq d_4$), the time complexity of the algorithm FIND-MAX is a function

$$T(n) = an + b$$

where a and b are some constants whose precise values depend on the execution time of individual instructions.

Remark: These constants could be expressed as

$$a = d_2 + \max\{d_3, d_4\} \qquad b = d_1 - d_2 - \max\{d_3, d_4\}$$

For example

$$T(n) = 33n - 8$$

Time Complexity of an Algorithm

If it would be sufficient to find out that the time complexity of the algorithm FIND-MAX is some function of the form

$$T(n) = an + b,$$

where the precise values of constants a and b would not be important for us, the whole analysis could be considerably simpler.

- In fact, we usually do not want to know precisely how function $T(n)$ look (in general, it can be a very complicated function), and it would be sufficient to know that values of the function $T(n)$ "approximately" correspond to values of a function $S(n) = an + b$, where a and b are some constants.

Time Complexity of an Algorithm

For a given function $T(n)$ expressing the time or space complexity, it is usually sufficient to express it approximately — to have an **estimation** where

- we ignore the less important parts

  (e.g., in function $T(n) = 15n^2 + 40n - 5$ we can ignore $40n$ and -5, and to consider function $T(n) = 15n^2$ instead of the original function),

- we ignore multiplication constants

  (e.g., instead of function $T(n) = 15n^2$ we will consider function $T(n) = n^2$)

- we won't ignore constants in exponents — for example there is a big difference between functions $T_1(n) = n^2$ and $T_2(n) = n^3$.

- we will be interested how function $T(n)$ behaves for "big" values of n, we can ignore its behaviour on small values

Growth of Functions

A program works on an input of size n.
Let us assume that for an input of size n, the program performs $T(n)$ operations and that an execution of one operation takes $1\,\mu s$ (10^{-6} s).

	n							
$T(n)$	20	40	60	80	100	200	500	1000
n	20 μs	40 μs	60 μs	80 μs	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	86 μs	0.213 ms	0.354 ms	0.506 ms	0.664 ms	1.528 ms	4.48 ms	9.96 ms
n^2	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms	40 ms	0.25 s	1 s
n^3	8 ms	64 ms	0.216 s	0.512 s	1 s	8 s	125 s	16.7 min.
n^4	0.16 s	2.56 s	12.96 s	42 s	100 s	26.6 min.	17.36 hours	11.57 days
2^n	1.05 s	12.75 days	36560 years	$38.3 \cdot 10^9$ years	$40.1 \cdot 10^{15}$ years	$50 \cdot 10^{45}$ years	$10.4 \cdot 10^{136}$ years	–
$n!$	77147 years	$2.59 \cdot 10^{34}$ years	$2.64 \cdot 10^{68}$ years	$2.27 \cdot 10^{105}$ years	$2.96 \cdot 10^{144}$ years	–	–	–

Growth of Functions

Let us consider 3 algorithms with complexities $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Our computer can do in a reasonable time (the time we are willing to wait) 10^{12} steps.

Complexity	Input size
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Growth of Functions

Let us consider 3 algorithms with complexities $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Our computer can do in a reasonable time (the time we are willing to wait) 10^{12} steps.

Complexity	Input size
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Now we speed up our computer 1000 times, meaning it can do 10^{15} steps.

Complexity	Input size	Growth
$T_1(n) = n$	10^{15}	$1000\times$
$T_2(n) = n^3$	10^5	$10\times$
$T_3(n) = 2^n$	50	$+10$

Asymptotic Notation

In the following, we will consider functions of the form $f : \mathbb{N} \to \mathbb{R}$, where:

- The values of $f(n)$ need not to be defined for all values of $n \in \mathbb{N}$ but there must exist some constant n_0 such that the value of $f(n)$ is defined for all $n \in \mathbb{N}$ such that $n \geq n_0$.

  Example: Function $f(n) = \log_2(n)$ is not defined for $n = 0$ but it is defined for all $n \geq 1$.

- There must exist a constant n_0 such that for all $n \in \mathbb{N}$, where $n \geq n_0$, is $f(n) \geq 0$.

  Example: It holds for function $f(n) = n^2 - 25$ that $f(n) \geq 0$ for all $n \geq 5$.

Asymptotic Notation

Let us take an arbitrary function $g : \mathbb{N} \to \mathbb{R}$. Expressions $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$, and $\omega(g)$ denote **sets of functions** of the type $\mathbb{N} \to \mathbb{R}$, where:

- $O(g)$ – the set of all functions that grow at most as fast as g
- $\Omega(g)$ – the set of all functions that grow at least as fast as g
- $\Theta(g)$ – the set of all functions that grow as fast as g
- $o(g)$ – the set of all fuctions that grow slower than function g
- $\omega(g)$ – the set of all functions that grow faster than function g

Remark: These are not definitions! The definitions will follow on the next slides.

- O – big "O"
- Ω – uppercase Greek letter "omega"
- Θ – uppercase Greek letter "theta"
- o – small "o"
- ω – small "omega"

Asymptotic Notation – Symbol O

Informally:

$O(g)$ – the set of all functions that grow at most as fast as g

How to define formally when $f \in O(g)$ holds?

The first try:

- to compare the values of the functions

$$(\forall n \in \mathbb{N})(f(n) \leq g(n))$$

A problem: This does not allow to ignore the values of constants, e.g., it is not true that $(\forall n \in \mathbb{N})(3n^2 \leq 2n^2)$.

Asymptotic Notation – Symbol O

Informally:

$O(g)$ – the set of all functions that grow at most as fast as g

How to define formally when $f \in O(g)$ holds?

The second try:

- to multiply function g with some big enough constant c

$$(\exists c > 0)(\forall n \in \mathbb{N})(f(n) \leq c \cdot g(n))$$

A problem: The inequality need not hold for some small values of n even after multiplying g by some arbitralily big value.

For example, function $g(n) = n^2$ grows faster than function $f(n) = n + 5$. However, not matter how big constant c is chosen, it will never be true that $n + 5 \leq c \cdot n^2$ for $n = 0$.

Asymptotic Notation – Symbol O

Informally:

$O(g)$ – the set of all functions that grow at most as fast as g

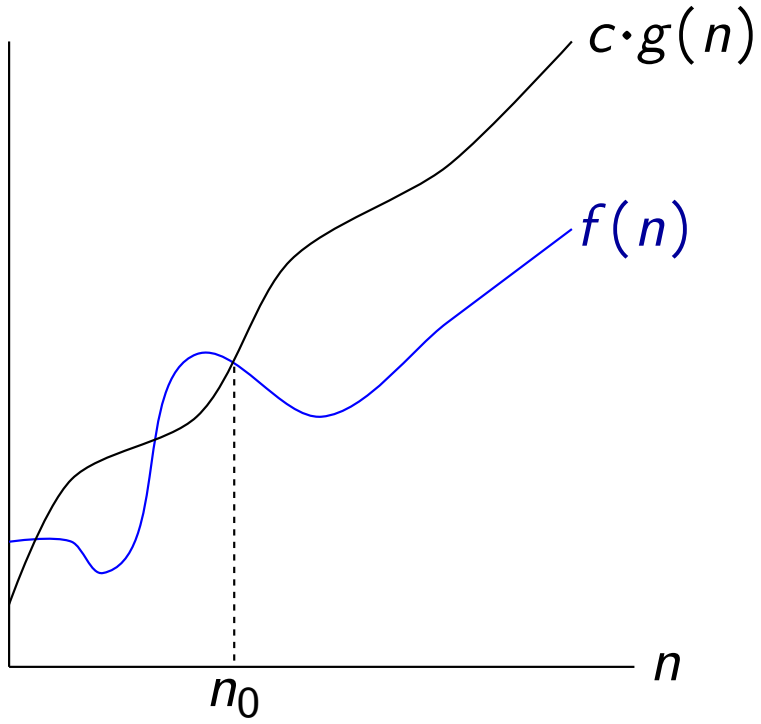
How to define formally when $f \in O(g)$ holds?

The third try:

- it is not required that the inequality holds for each n, it is sufficient when it holds for all values that are "big enough"

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(f(n) \leq c \cdot g(n))$$

Asymptotic Notation – Symbol O



Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{R}$. For a function $f : \mathbb{N} \to \mathbb{R}$ we have $f \in O(g)$ iff

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(f(n) \leq c \cdot g(n)\big).$$

Asymptotic Notation – Symbol O

Remarks:

- c is a posive real number (i.e., $c \in \mathbb{R}$ and $c > 0$)
- n_0 and n are natural numbers (i.e., $n_0 \in \mathbb{N}$ and $n \in \mathbb{N}$)

Asymptotic Notation – Symbol O

Example: Let us consider functions $f(n) = 2n^2 + 3n + 7$ and $g(n) = n^2$.

We want to show that $f \in O(g)$, i.e., $f \in O(n^2)$:

- **Approach 1:**

  Let us take for example $c = 3$.

  $$c \cdot g(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2$$

  We need to find some n_0 such that for all $n \geq n_0$ it holds that

  $$2n^2 \geq 2n^2 \qquad \frac{1}{2}n^2 \geq 3n \qquad \frac{1}{2}n^2 \geq 7$$

  We can easily check that for example $n_0 = 6$ satisfies this.

  For each $n \geq 6$ we have $c \cdot g(n) \geq f(n)$:

  $$cg(n) = 3n^2 = 2n^2 + \frac{1}{2}n^2 + \frac{1}{2}n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotic Notation – Symbol O

The example where $f(n) = 2n^2 + 3n + 7$ and $g(n) = n^2$:

- **Approach 2:**

  Let us take $c = 12$.

  $$c \cdot g(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2$$

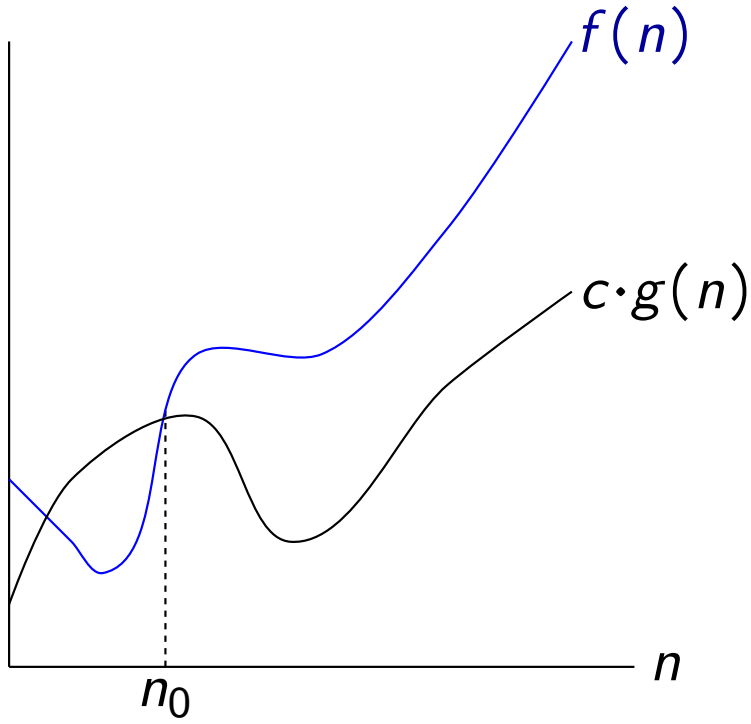
  We need to find some n_0 such that for all $n \geq n_0$ we have

  $$2n^2 \geq 2n^2 \qquad 3n^2 \geq 3n \qquad 7n^2 \geq 7$$

  These inequalities obviously hold for $n_0 = 1$, and so for each $n \geq 1$ we have $f(n) \leq c \cdot g(n)$:

  $$c \cdot g(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotic Notation – Symbol Ω



Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{R}$. For a function $f : \mathbb{N} \to \mathbb{R}$ we have $f \in \Omega(g)$ iff

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(c \cdot g(n) \leq f(n)\big).$$

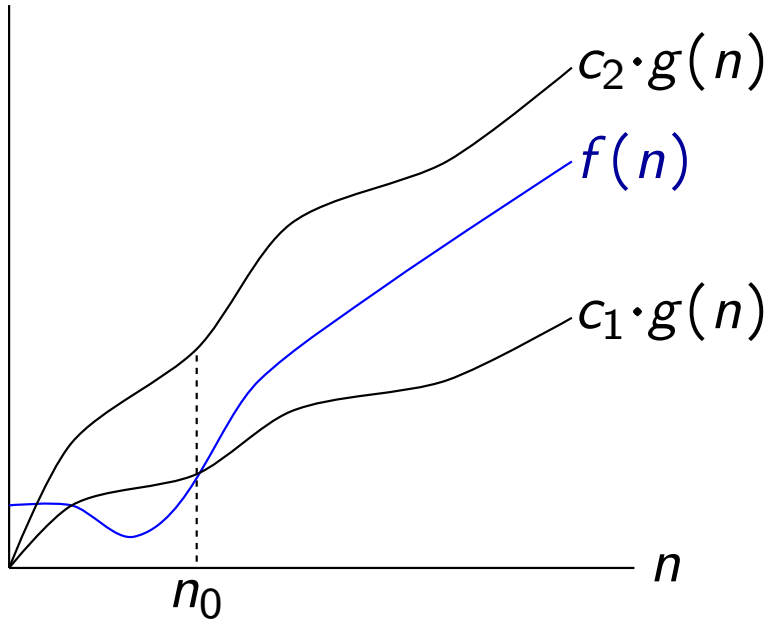
Asymptotic Notation – Symbol Ω

It is not difficult to prove the following proposition:

For arbitrary functions f and g we have:

$$f \in O(g) \qquad \text{iff} \qquad g \in \Omega(f)$$

Asymptotic Notation – Symbol Θ



Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{R}$. For a function $f : \mathbb{N} \to \mathbb{R}$ we have $f \in \Theta(g)$ iff

$$(\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\big).$$

Asymptotic Notation – Symbol Θ

The following easily follows from the definition of Θ:

For arbitrary functions f and g we have:

$$f \in \Theta(g) \quad \text{iff} \quad f \in O(g) \text{ and } f \in \Omega(g)$$

$$f \in \Theta(g) \quad \text{iff} \quad f \in O(g) \text{ and } g \in O(f)$$

$$f \in \Theta(g) \quad \text{iff} \quad g \in \Theta(f)$$

Asymptotic Notation – Symbols o and ω

Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{R}$. For a function $f : \mathbb{N} \to \mathbb{R}$ we have $f \in o(g)$ iff

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

Definition

Let us consider an arbitrary function $g : \mathbb{N} \to \mathbb{R}$. For a function $f : \mathbb{N} \to \mathbb{R}$ we have $f \in \omega(g)$ iff

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$$

Asymptotic Notation

For arbitrary functions f and g we have the following propositions:

If there exists a constant $c \geq 0$ such that

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = c$$

then $f \in O(g)$.

If there exists a constant $c \geq 0$ such that

$$\lim_{n \to +\infty} \frac{g(n)}{f(n)} = c$$

then $f \in \Omega(g)$.

Asymptotic Notation

It is obvious that:

- If $f \in o(g)$ then $f \in O(g)$.
- If $f \in \omega(g)$ then $f \in \Omega(g)$.

Asymptotic Notation

The asymptotic notation can be viewed as a certain kind of comparison of a **rate of growth** functions:

$f \in O(g)$	—	rate of growth of f "$\leq$" rate of growth of g
$f \in \Omega(g)$	—	rate of growth of f "$\geq$" rate of growth of g
$f \in \Theta(g)$	—	rate of growth of f "$=$" rate of growth of g
$f \in o(g)$	—	rate of growth of f "$<$" rate of growth of g
$f \in \omega(g)$	—	rate of growth of f "$>$" rate of growth of g

Remark:

- There are pairs of functions f and g such that

  $$f \notin O(g) \qquad \text{and} \qquad g \notin O(f),$$

  for example

  $$f(n) = n^2 \qquad g(n) = \begin{cases} n & \text{if } n \mod 2 = 1 \\ n^3 & \text{otherwise} \end{cases}$$

Asymptotic Notation

- A function f is called:

  linear, if $f(n) \in \Theta(n)$

  quadratic, if $f(n) \in \Theta(n^2)$

  cubic, if $f(n) \in \Theta(n^3)$

  polynomial, if $f(n) \in O(n^k)$ for some $k > 0$

  exponential, if $f(n) \in O(c^{n^k})$ for some $c > 1$ and $k > 0$

  logarithmic, if $f(n) \in \Theta(\log n)$

  polylogarithmic, if $f(n) \in \Theta(\log^k n)$ for some $k > 0$

- $O(1)$ is the set of all **bounded** functions, i.e., functions whose function values can be bounded from above by a constant.

- Exponential functions are often written in the form $2^{O(n^k)}$ when the asymptotic notation is used, since then we do not need to consider different bases.

Asymptotic Notation

In general, it holds that:

- every polylogarithmic function grows slower than any polynomial function
- every polynomial function grows slower than any exponential function
- to compare polynomial functions n^k and n^ℓ it is sufficient to compare values k and ℓ
- to compare polylogarithmic functions $\log^k n$ and $\log^\ell n$ it is sufficient to compare values k and ℓ
- to compare exponential functions $2^{p(n)}$ and $2^{q(n)}$ it is sufficient to compare polynomials $p(n)$ and $q(n)$.

Asymptotic Notation

Proposition

Let us assume that a and b are constants such that $a > 0$ and $b > 0$, and k and ℓ are some arbitrary constants where $k \geq 0$, $\ell \geq 0$ and $k \leq \ell$.

Let us consider functions

$$f(n) = a \cdot n^k \qquad\qquad g(n) = b \cdot n^\ell$$

For each such functions f and g it holds that $f \in O(g)$:

Proof: Let us take $c = \frac{a}{b}$.

Because for $n \geq 1$ we obviously have $n^k \leq n^\ell$ (since $k \leq \ell$), for $n \geq 1$ we have

$$c \cdot g(n) = \frac{a}{b} \cdot g(n) = \frac{a}{b} \cdot b \cdot n^\ell = a \cdot n^\ell \geq a \cdot n^k = f(n)$$

Asymptotic Notation

Proposition

For any $a, b > 1$ and any $n > 0$ we have

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Proof: From $n = a^{\log_a n}$ it follows that $\log_b n = \log_b(a^{\log_a n})$.
Since $\log_b(a^{\log_a n}) = \log_a n \cdot \log_b a$, we obtain $\log_b n = \log_a n \cdot \log_b a$, from which the above mentioned conclusion follows directly. □

Due to this observation, the base of a logarithm is often omited in the asymptotic notation: for example, instead of $\Theta(n \log_2 n)$ we can write $\Theta(n \log n)$.

Asymptotic Notation

Examples:

$n \in O(n^2)$

$1000n \in O(n)$

$2^{\log_2 n} \in \Theta(n)$

$n^3 \notin O(n^2)$

$n^2 \notin O(n)$

$n^3 + 2^n \notin O(n^2)$

$n^3 \in O(n^4)$

$0.00001n^2 - 10^{10}n \in \Theta(10^{10}n^2)$

$n^3 - n^2 \log_2^3 n + 1000n - 10^{100} \in \Theta(n^3)$

$n^3 + 1000n - 10^{100} \in O(n^3)$

$n^3 + n^2 \notin \Theta(n^2)$

$n! \notin O(2^n)$

Asymptotic Notation

For arbitrary functions f, g, and h we have:

- if $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
- if $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$
- if $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$

Asymptotic Notation

- For any function f a libovolnou konstantu $c > 0$ we have:
  - $c \cdot f \in \Theta(f)$
- For any pair of functions f, g we have:
  - $\max(f, g) \in \Theta(f + g)$
  - if $f \in O(g)$ then $f + g \in \Theta(g)$
- For any functions f_1, f_2, g_1, g_2 we have:
  - if $f_1 \in O(f_2)$ and $g_1 \in O(g_2)$ then $f_1 + g_1 \in O(f_2 + g_2)$ and $f_1 \cdot g_1 \in O(f_2 \cdot g_2)$
  - if $f_1 \in \Theta(f_2)$ and $g_1 \in \Theta(g_2)$ then $f_1 + g_1 \in \Theta(f_2 + g_2)$ and $f_1 \cdot g_1 \in \Theta(f_2 \cdot g_2)$

Asymptotic Notation

As mentioned before, expressions $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$, and $\omega(g)$ denote certain sets of functions.

In some texts, these expressions are sometimes used with a slightly different meaning:

- an expression $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ or $\omega(g)$ does not represent the corresponding set of functions but **some** function from this set.

This convention is often used in equations and inequations.

Example: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + O(n^2)$

When using this convention, we can for example write $f = O(g)$ instead of $f \in O(g)$.

Complexity of Algorithms

Let us say we would like to analyze the time complexity $T(n)$ of some algorithm consisting of instructions $I_1, I_2, \ldots, I_k$:

- Let us assume that how long it takes to execute each instruction is given by constants $c_1, c_2, \ldots, c_k$, i.e., the time it takes to execute instruction I_i once is specified by a constant c_i.
- Let us assume that In is the set of all possible inputes for the given algorithm.

  Let us define for each instruction I_i a corresponding function

  $$m_i : In \to \mathbb{N}$$

  specifying how many times instruction I_i will be executed during a computation over a given input, i.e., the value $m_i(w)$ specifies how many times instruction I_i will be executed during a computation over an input w.

Complexity of Algorithms

- Total running time of a computation over an input w:
$$t(w) = c_1 \cdot m_1(w) + c_2 \cdot m_2(w) + \cdots + c_k \cdot m_k(w).$$
- Let us racall that $T(n) = \max\{\, t(w) \mid size(w) = n \,\}$.
- For each of the functions $m_1, m_2, \ldots, m_k$ we can define a corresponding function $f_i : \mathbb{N} \to \mathbb{R}$, where
$$f_i(n) = \max\{\, m_i(w) \mid size(w) = n \,\}$$
is the maximum of numbers of executions of instruction I_i for all inputs of size n.
- It is obvious that $T \in O(f_1 + f_2 + \cdots + f_k)$.
- Let us recall that if $f_j \in O(f_i)$ then $c_i \cdot f_i + c_j \cdot f_j \in O(f_i)$.
- If there is a function f_i such that for all f_j, where $j \neq i$, we have $f_j \in O(f_i)$, then
$$T \in O(f_i).$$

Complexity of Algorithms

- Obviously, $T \in \Omega(f_i)$ for any function f_i.
- So in an analysis of a total running time $T(n)$, we can typically restrict our attention only to an analysis of how many times the most often executed instruction I_i is executed, i.e., on examination of a rate of groth of function $f_i(n)$ because

$$T \in \Theta(f_i).$$

- For other instructions I_j we just need to verify that

$$f_j \in O(f_i),$$

i.e., it is not necessary to determine precisely for them how fast they grow but rather it is sufficcient to determine for them that their rate of growth is not bigger than the rate of groth of f_i.

Example:

Algorithm: Finding the maximal element in an array

```
FIND-MAX (A, n):
    k := 0
    for i := 1 to n − 1 do
        if A[i] > A[k] then
            k := i
    return A[k]
```

Complexity of Algorithms

In the analysis of the complexity of the searching of a number in a sequence we obtained

$$f(n) = an + b\,.$$

If we would not like to do such a detailed analysis, we could deduce that the time complexity of the algorithm is $\Theta(n)$, because:

- The algorithm contains only one cycle, which is performed $(n-1)$ times for an input of size n, the number of iterations of the cycle is in $\Theta(n)$.
- Several instructions are performed in one iteration of the cycle. The number of these instructions is bounded from both above and below by some constant independent on the size of the input. So the time of execution of one iteration of the cycle is in $\Theta(1)$.
- Other instructions are executed just once. The time spent by their execution is in $\Theta(1)$.

Complexity of Algorithms

Let us try to analyze the time complexity of the following algorithm:

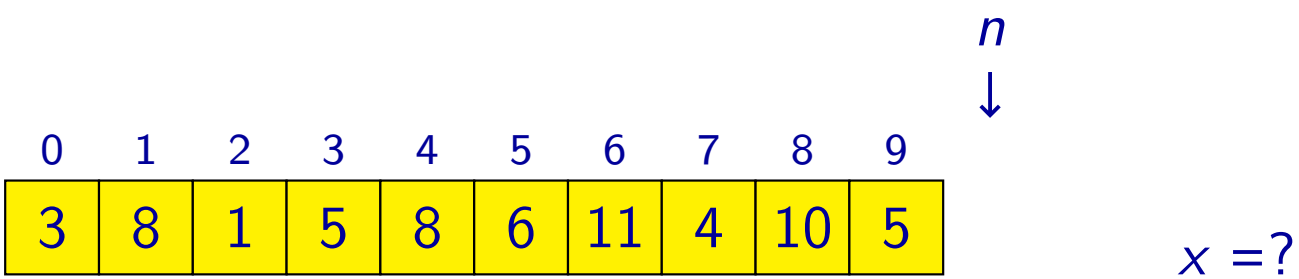
Algorithm: Insertion sort

```
INSERTION-SORT (A, n):
    for j := 1 to n − 1 do
        x := A[j]
        i := j − 1
        while i ≥ 0 and A[i] > x do
            A[i + 1] := A[i]
            i := i − 1
        A[i + 1] := x
```

I.e., we want to find a function $T(n)$ such that the time complexity of the algorithm INSERTION-SORT in the worst case is in $\Theta(T(n))$.

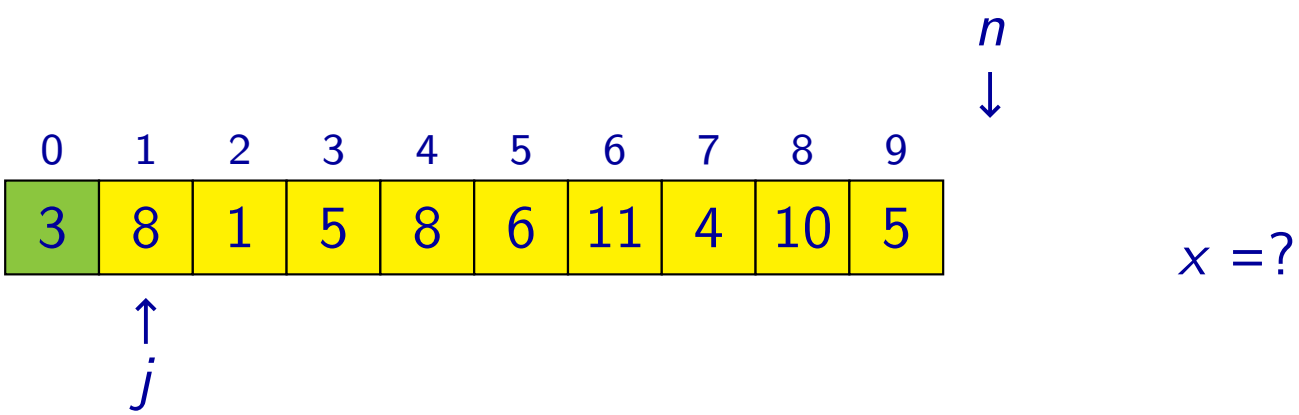
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



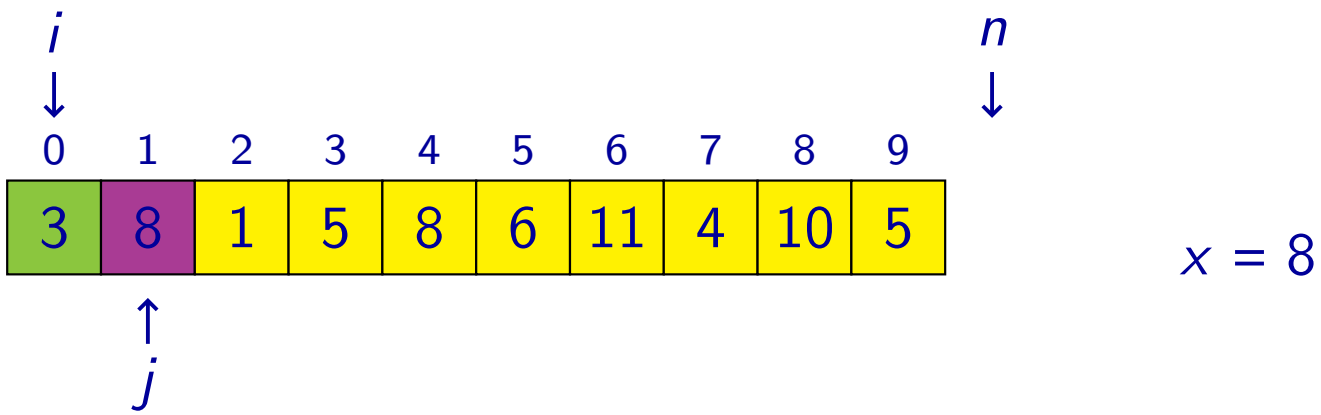
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



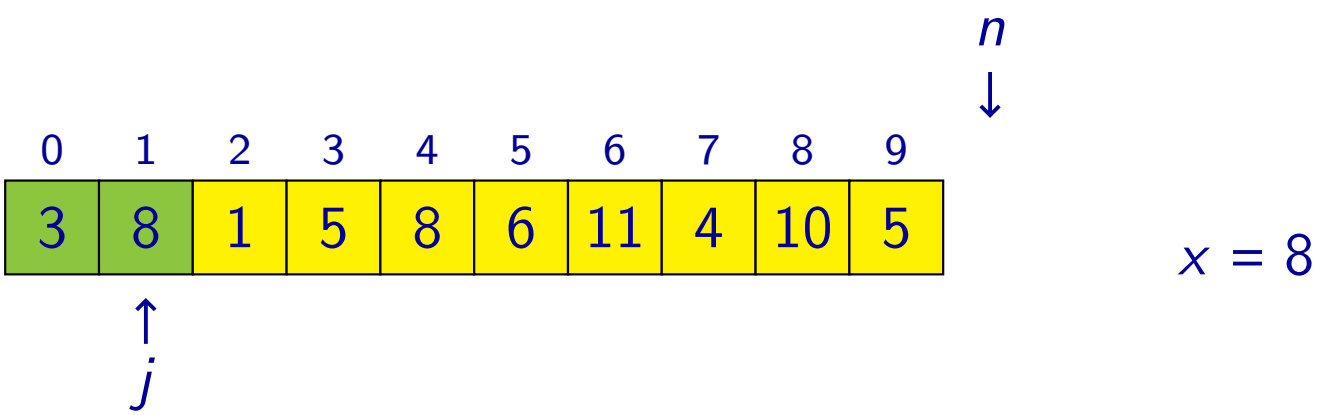
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



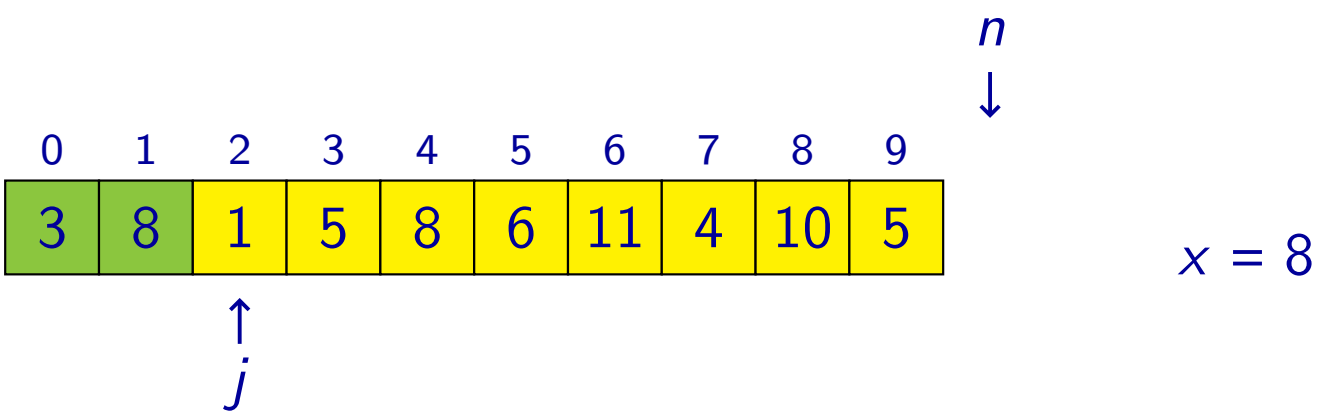
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



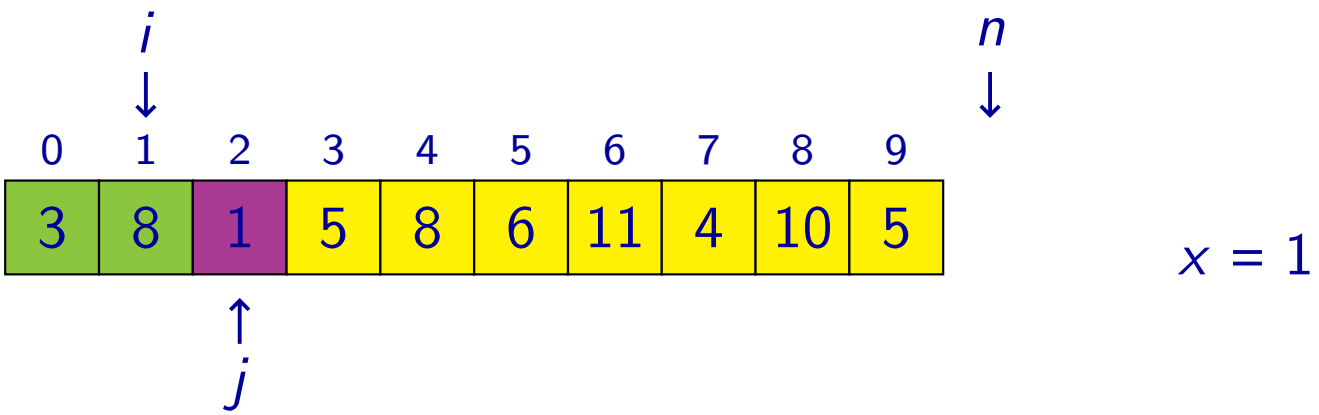
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



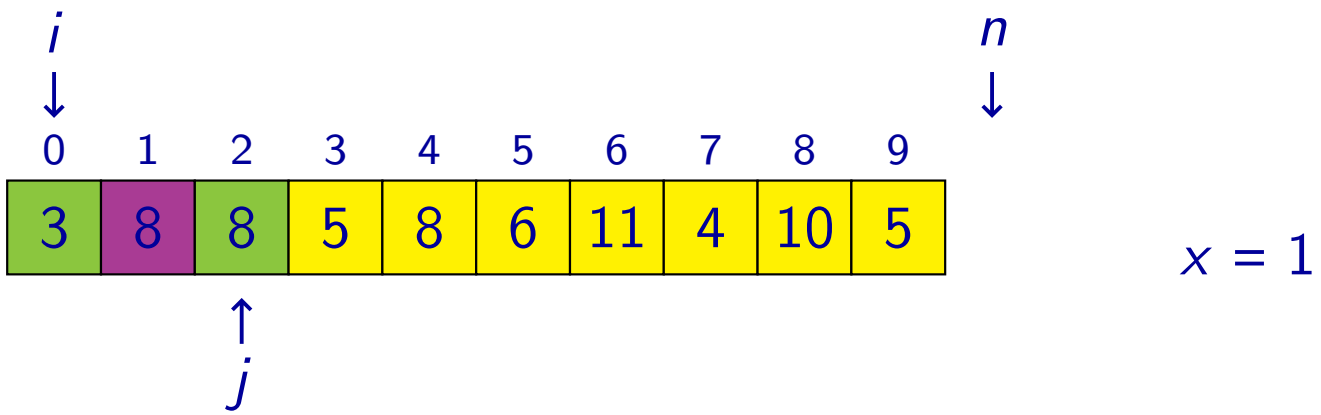
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



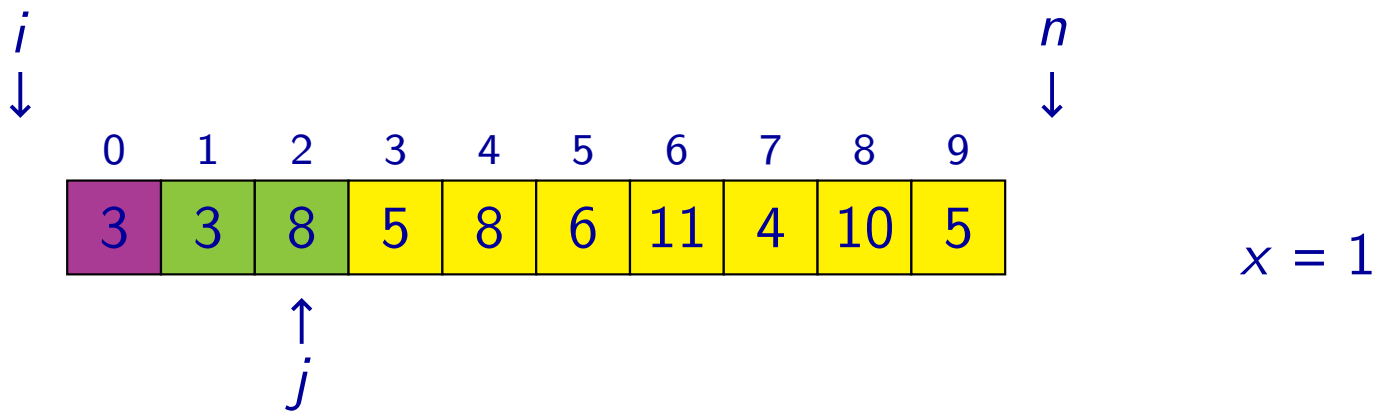
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



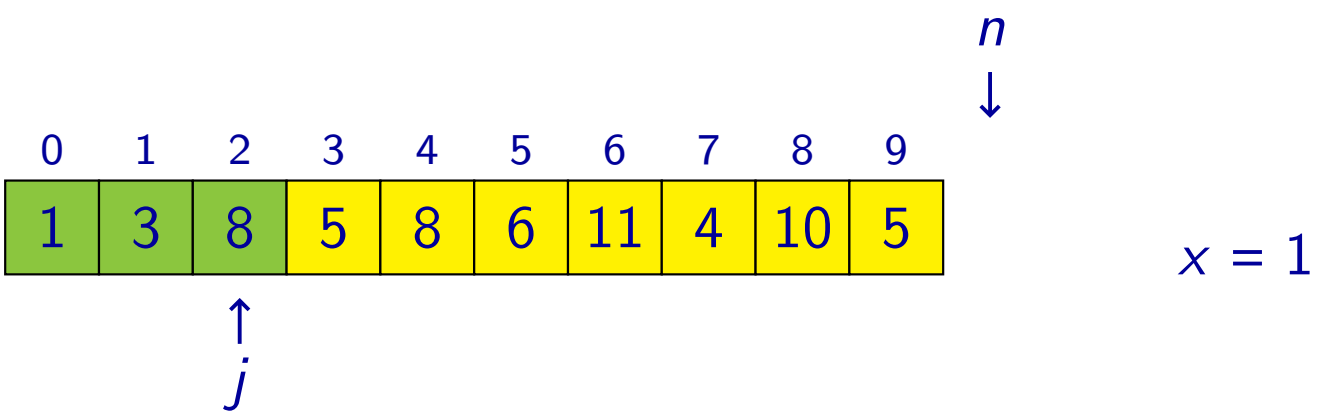
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



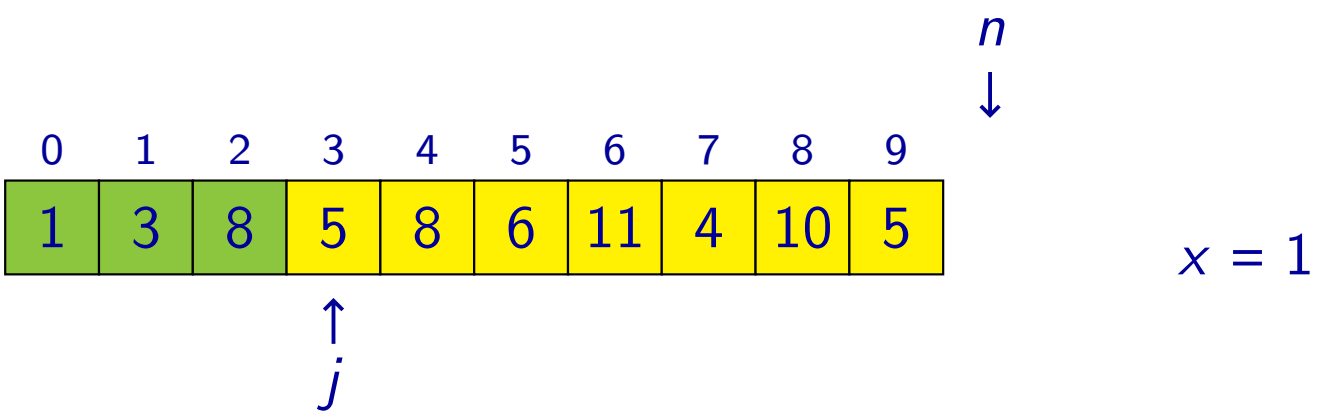
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



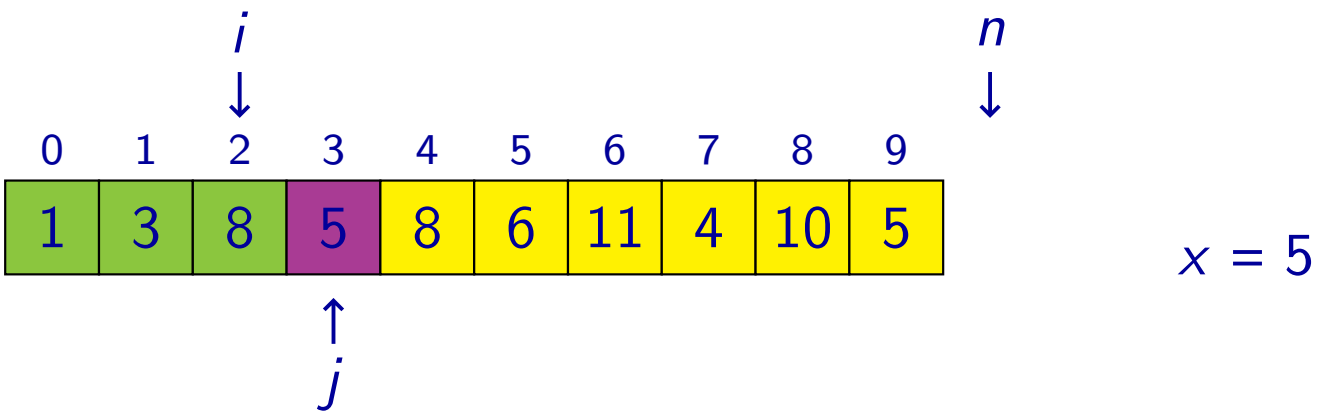
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



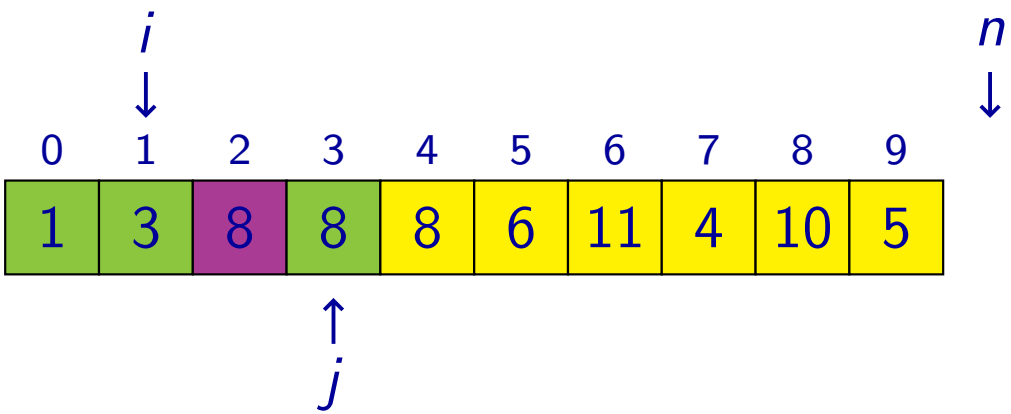
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



Example: A computation of INSERTION-SORT on input

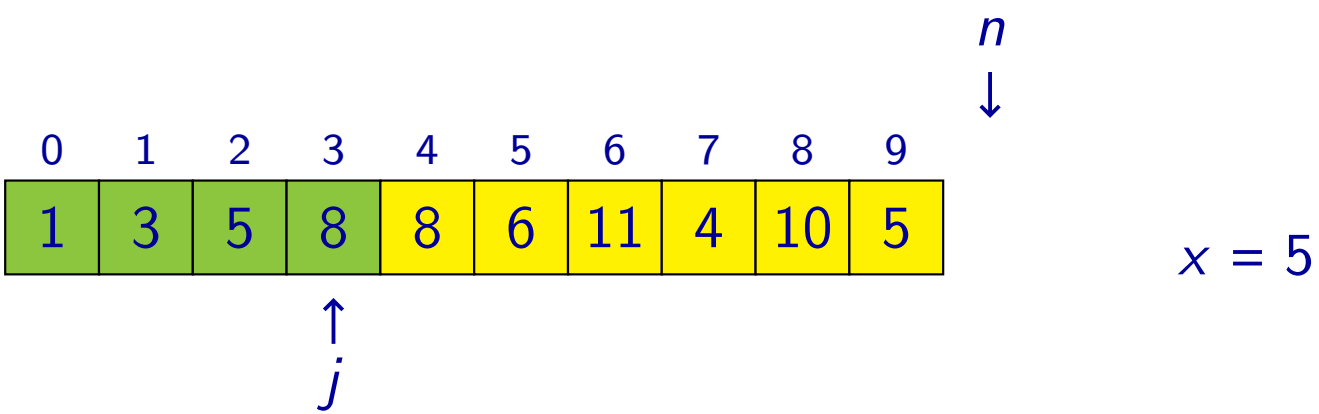
$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



$x = 5$

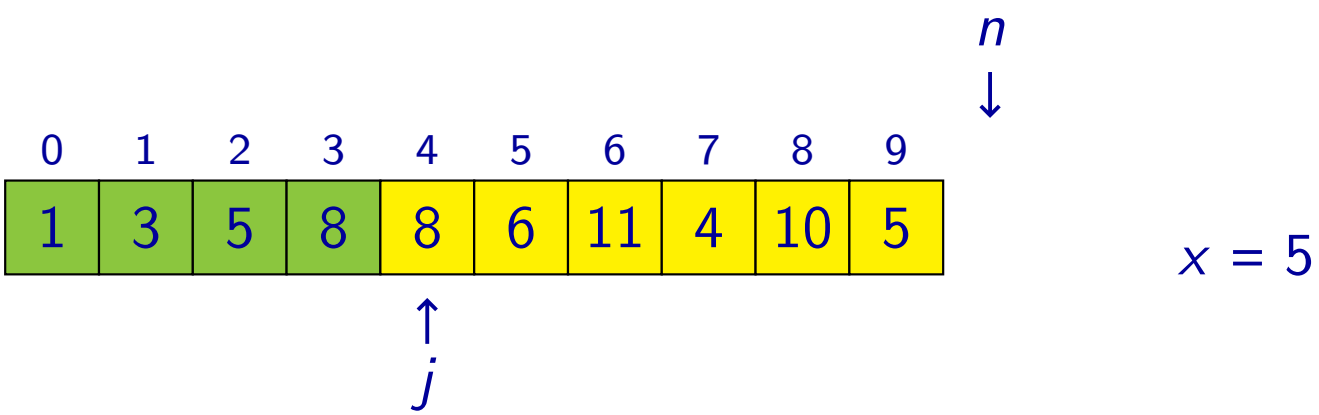
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



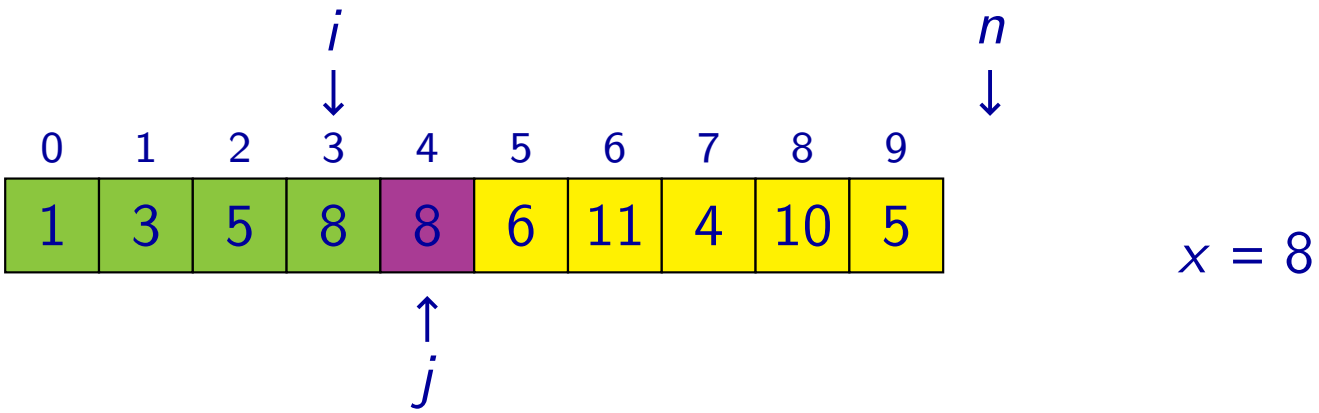
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



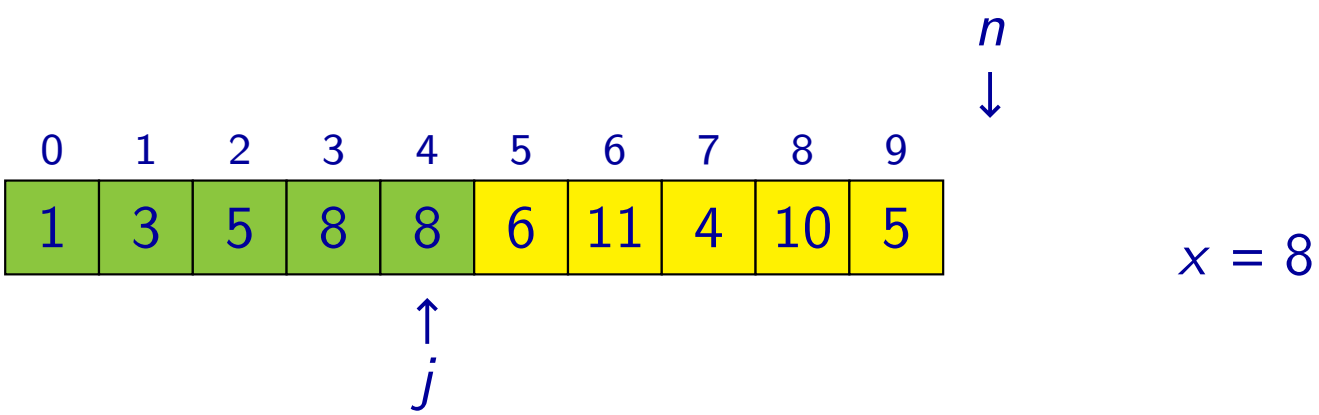
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



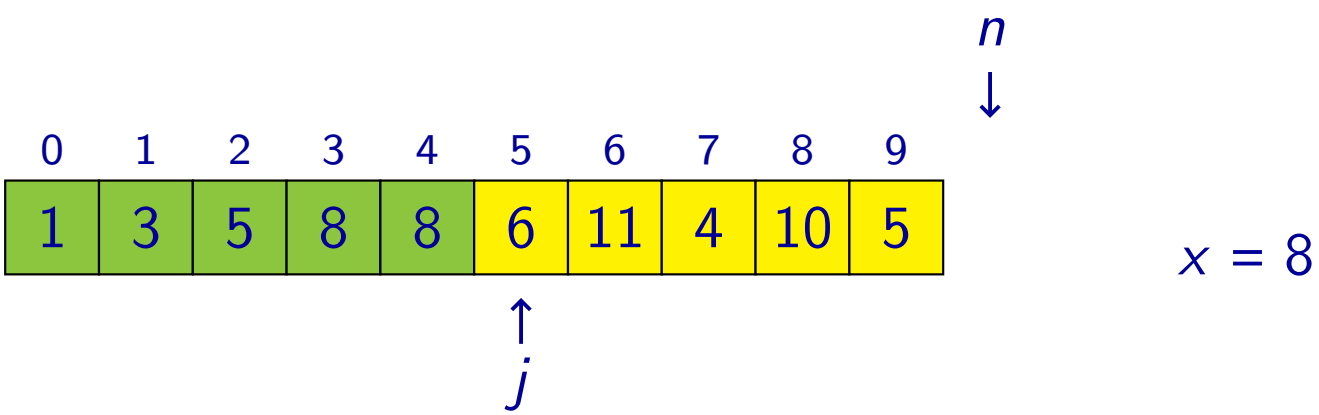
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



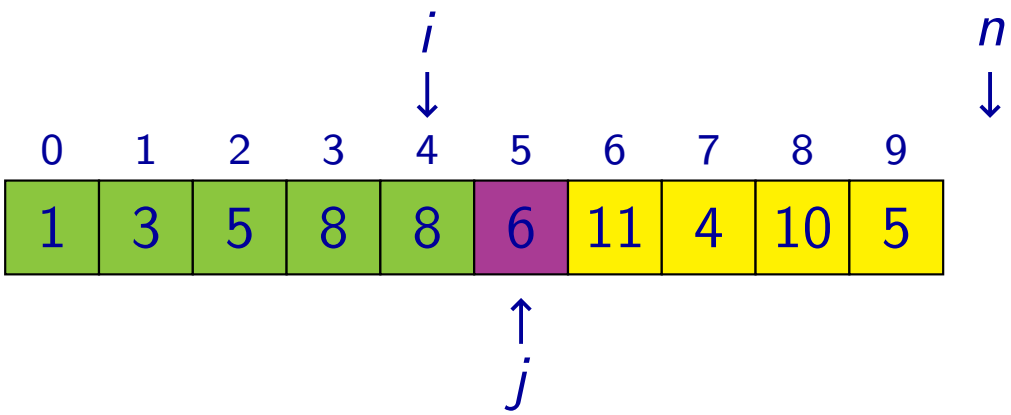
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



Example: A computation of INSERTION-SORT on input

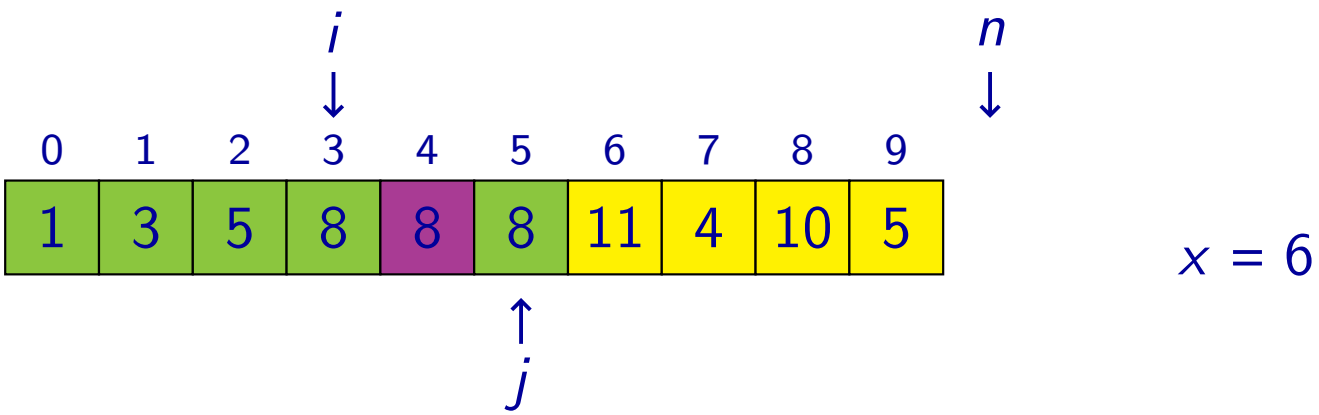
$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



$x = 6$

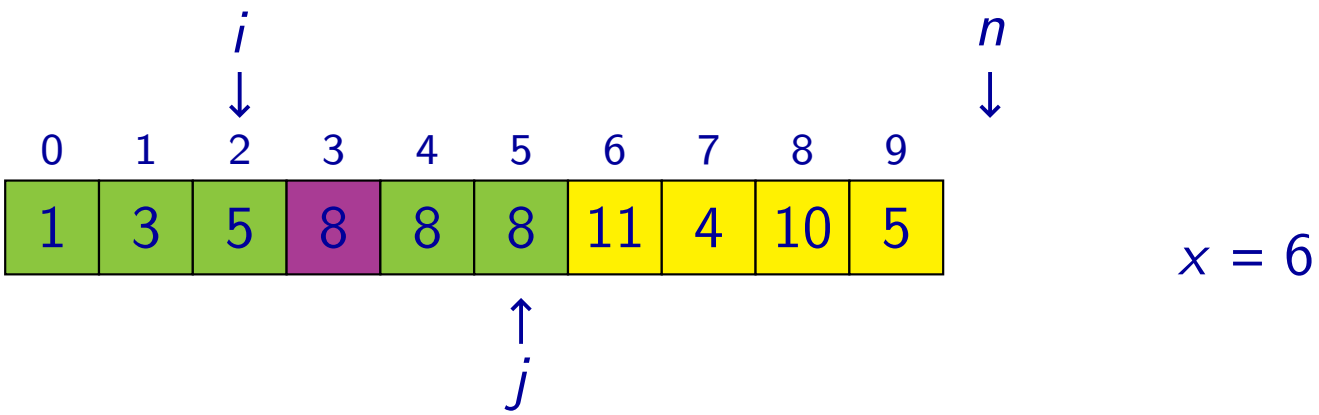
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



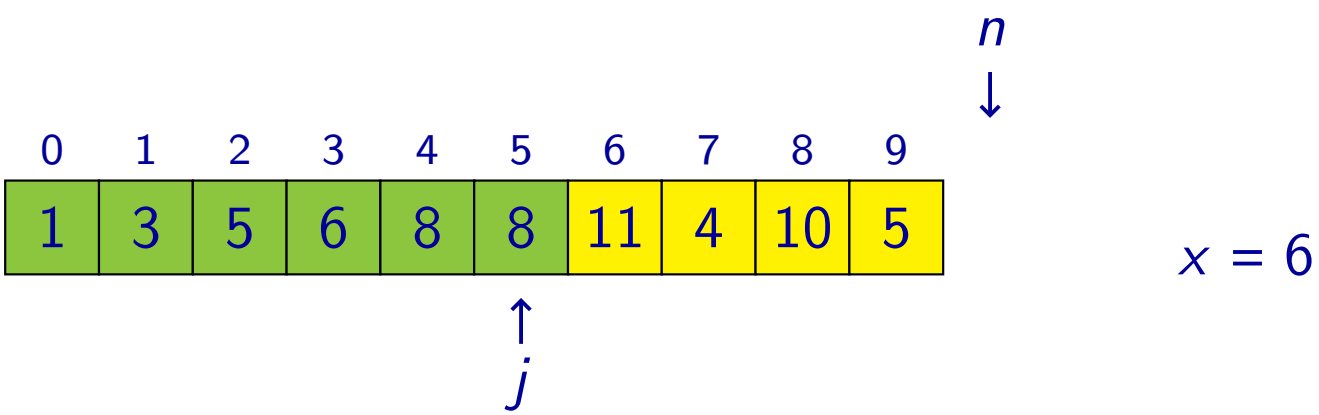
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



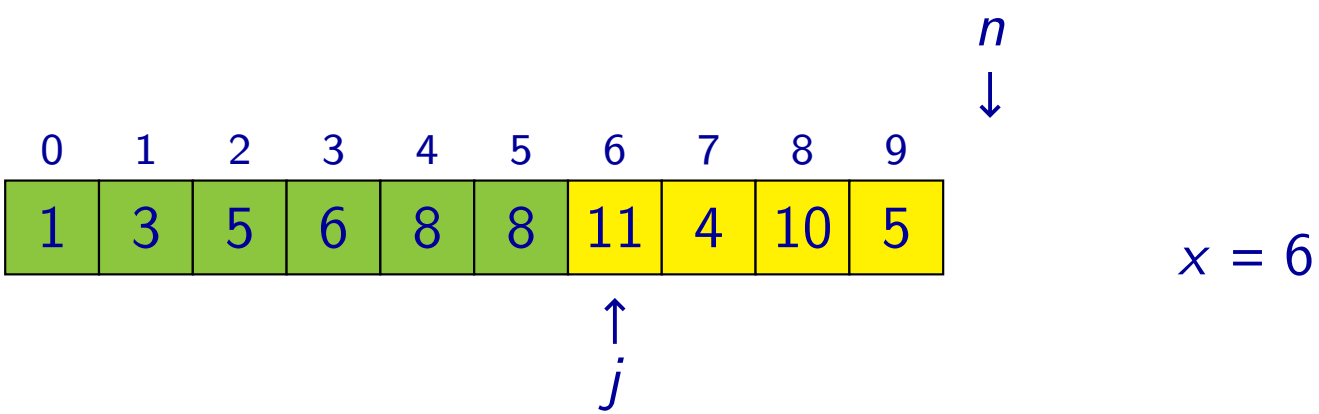
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



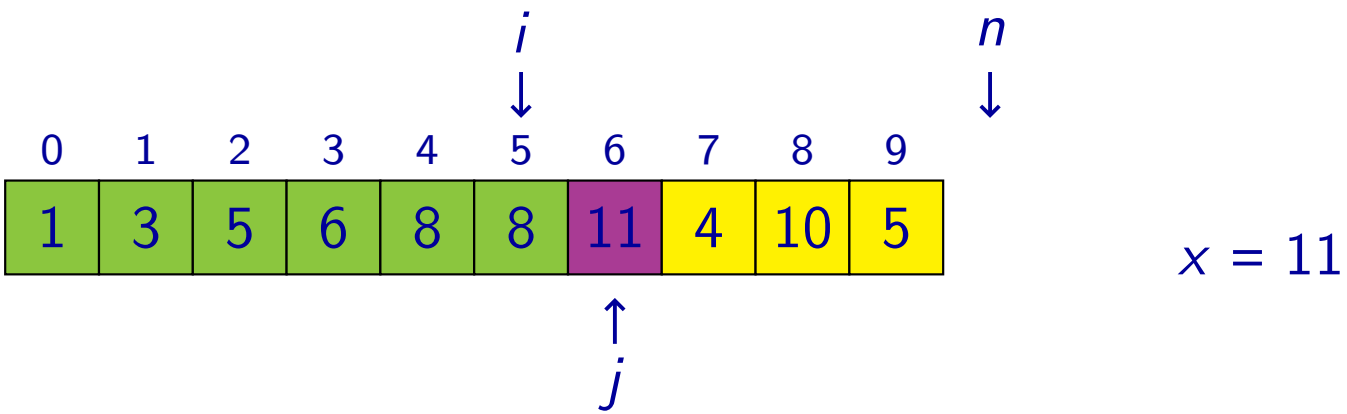
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



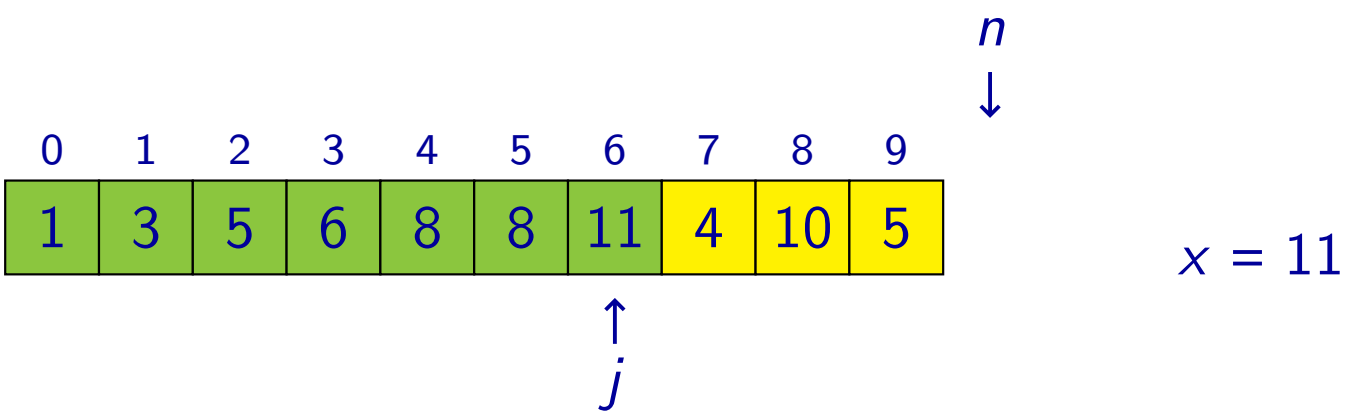
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



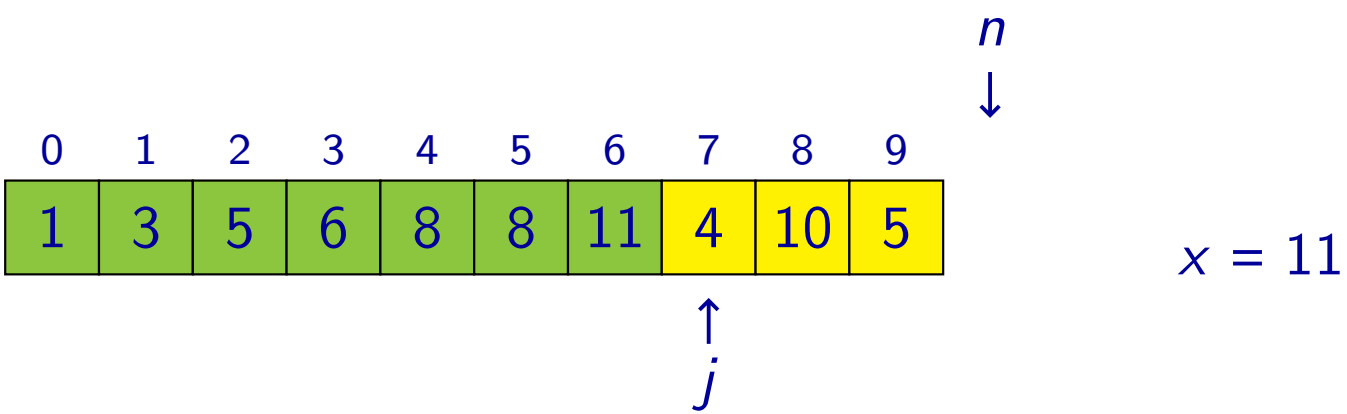
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



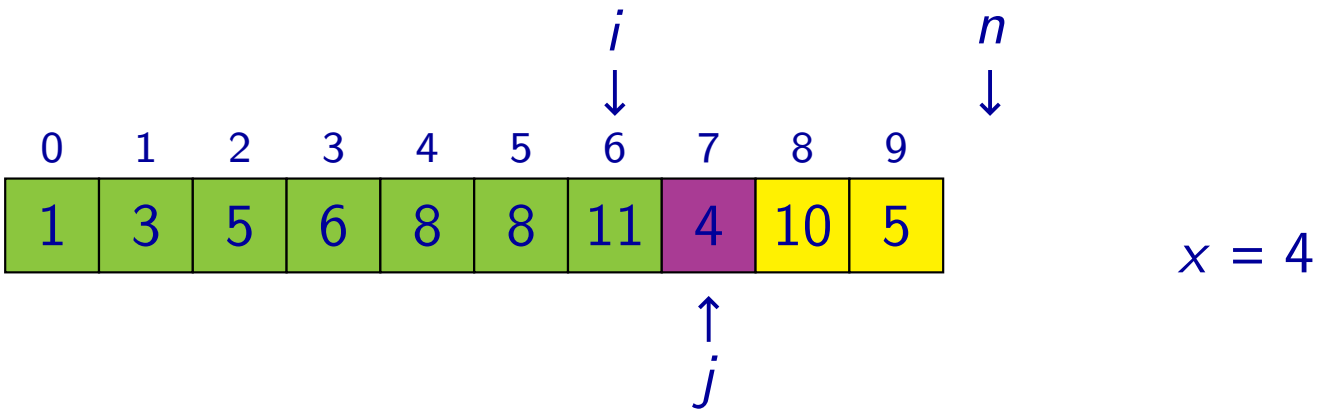
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



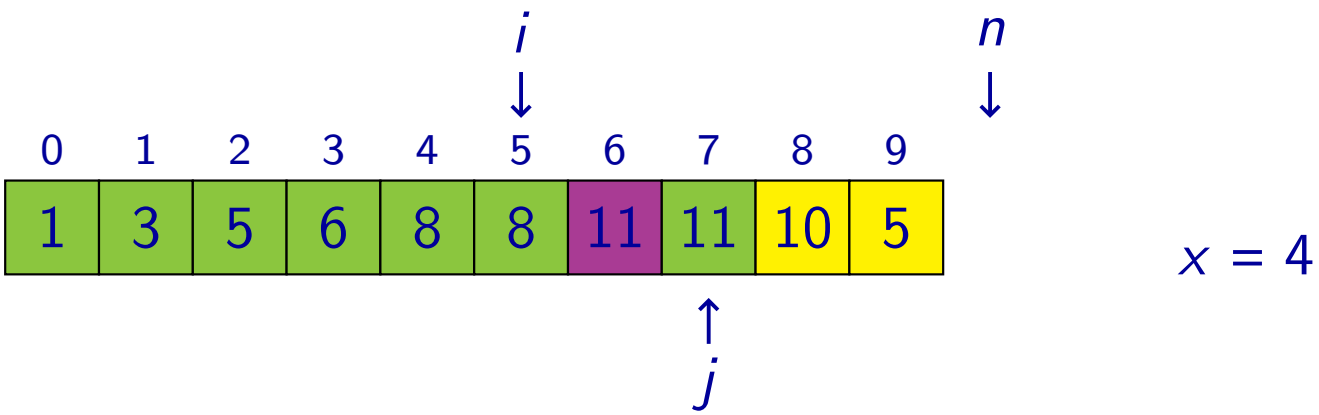
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



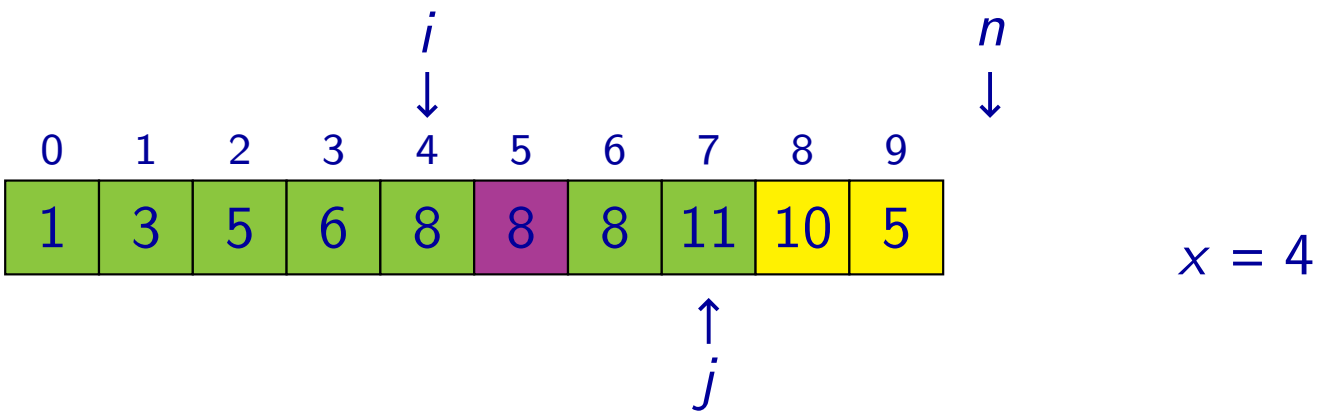
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



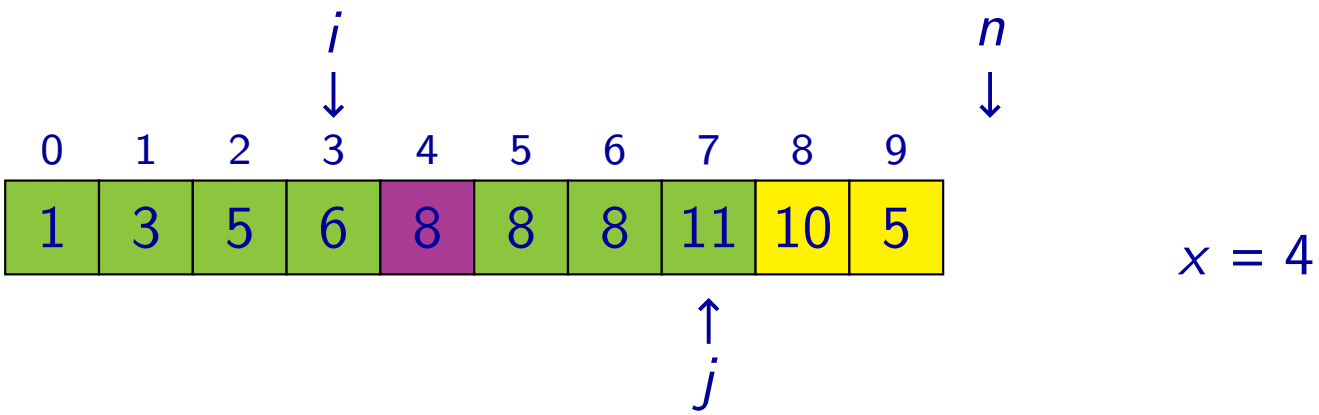
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



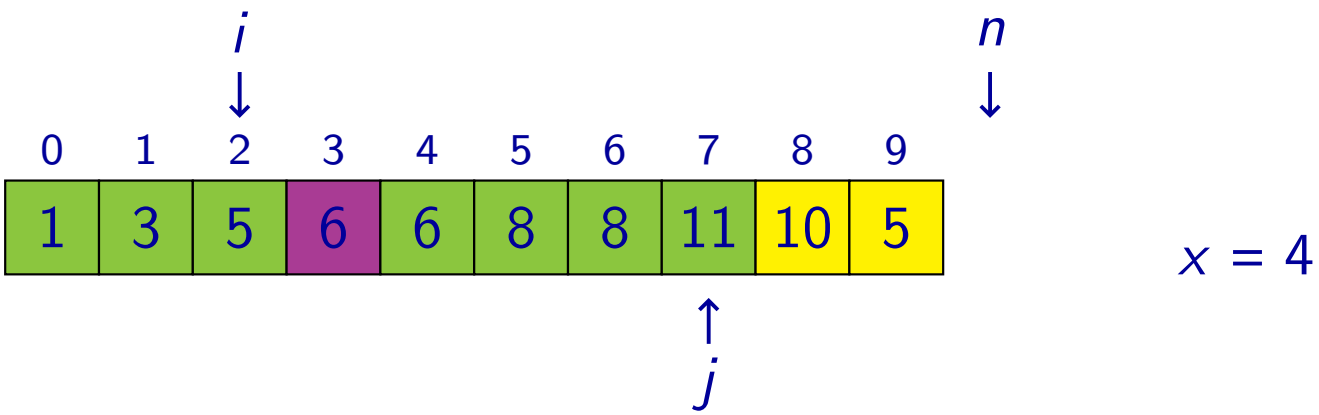
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



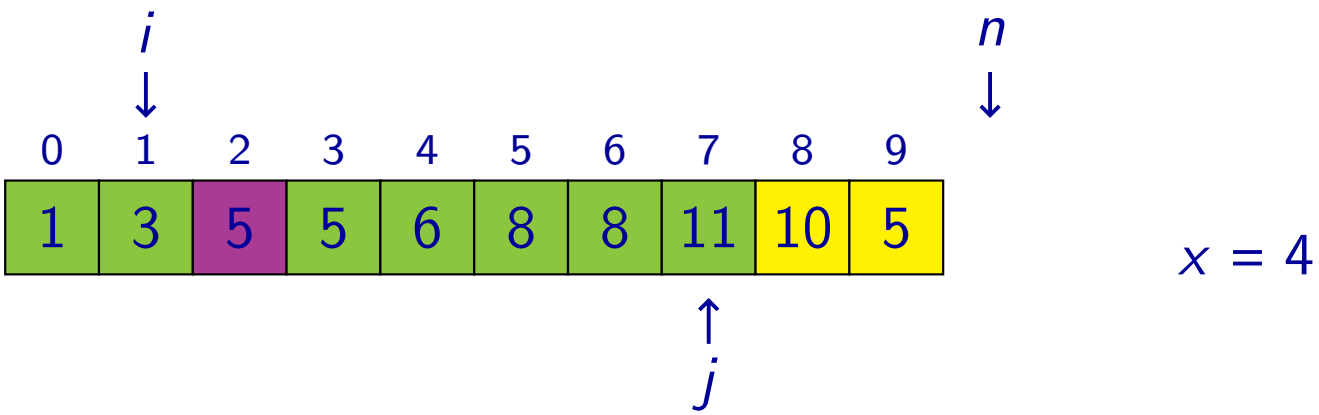
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



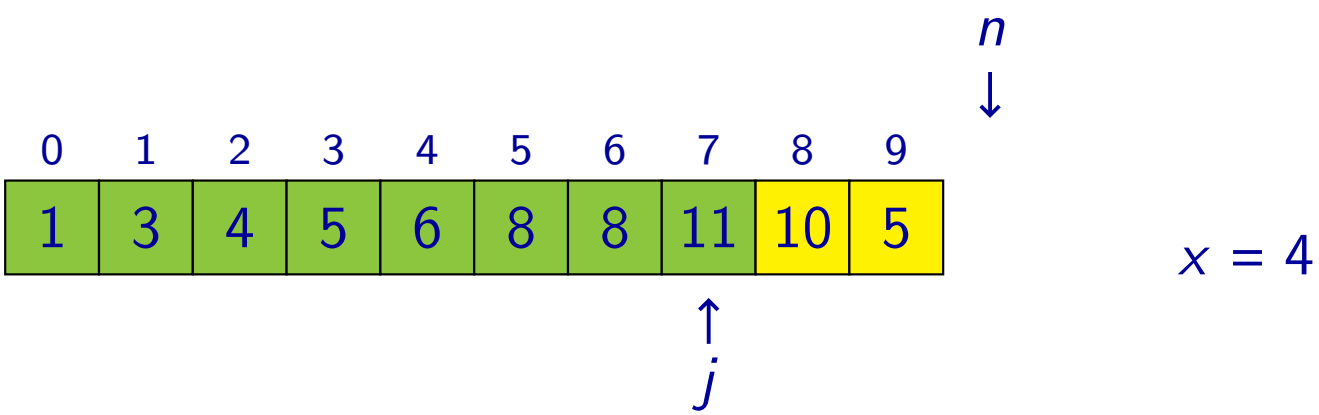
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



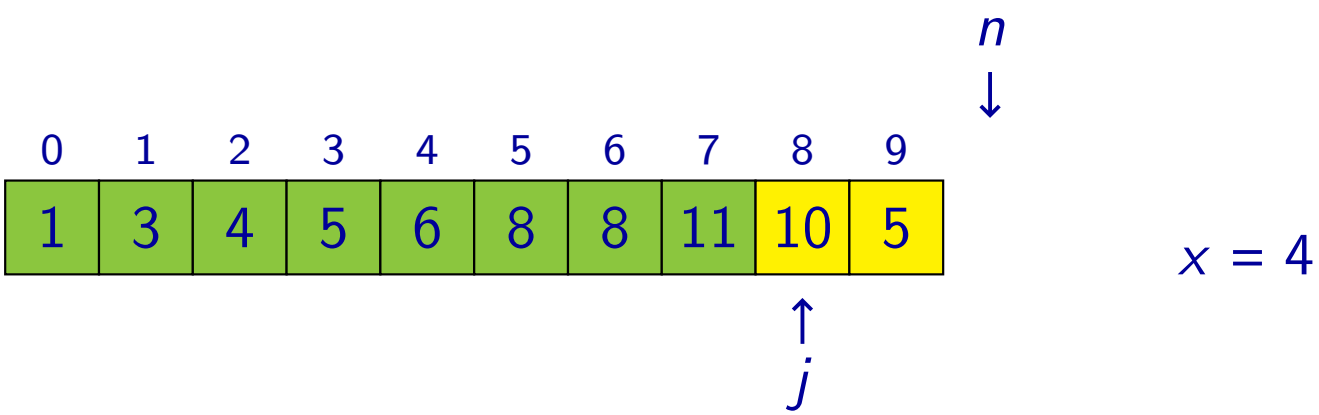
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



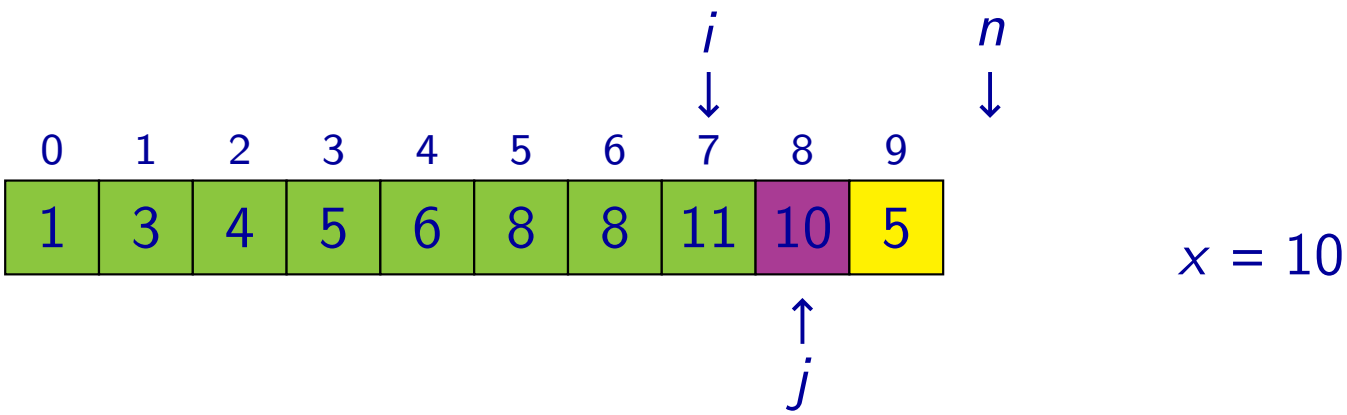
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



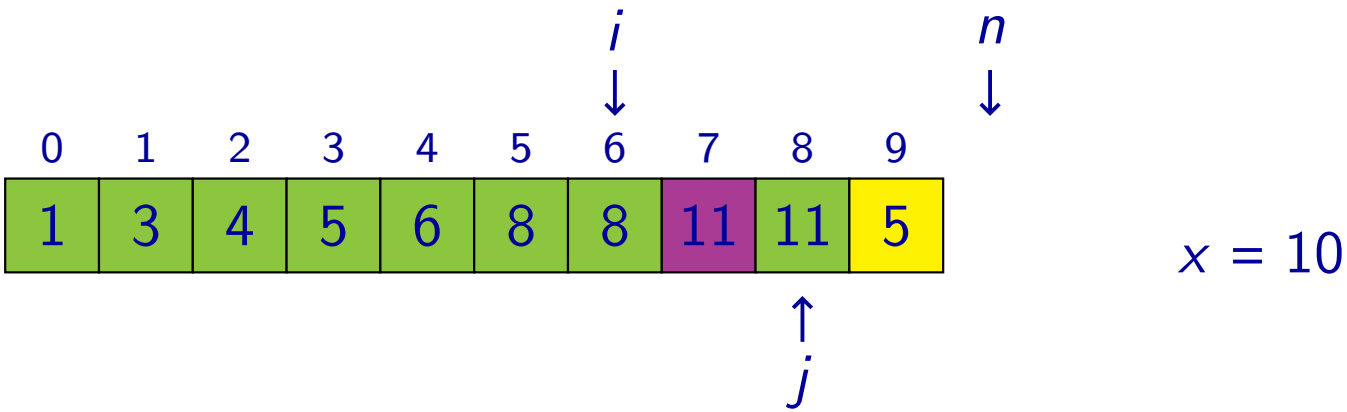
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



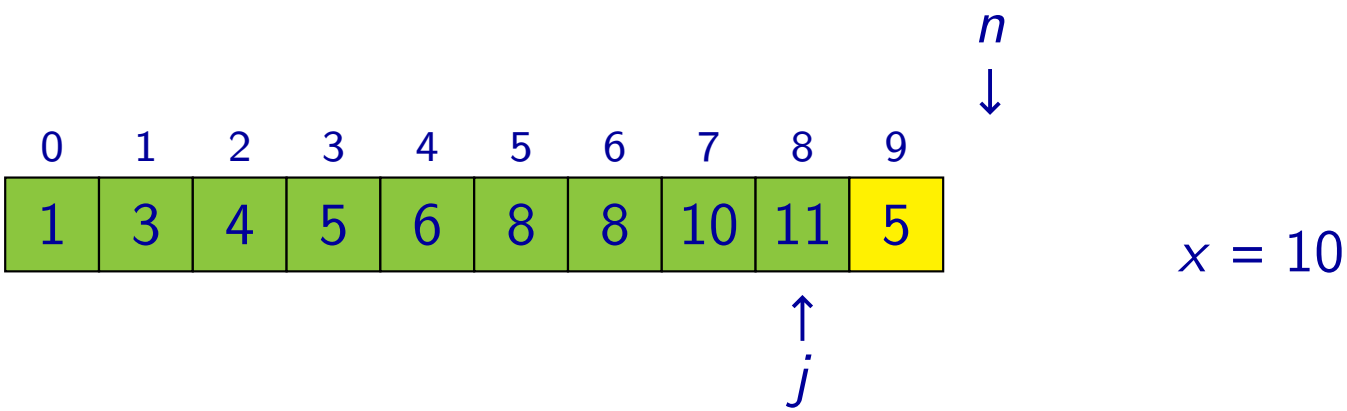
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



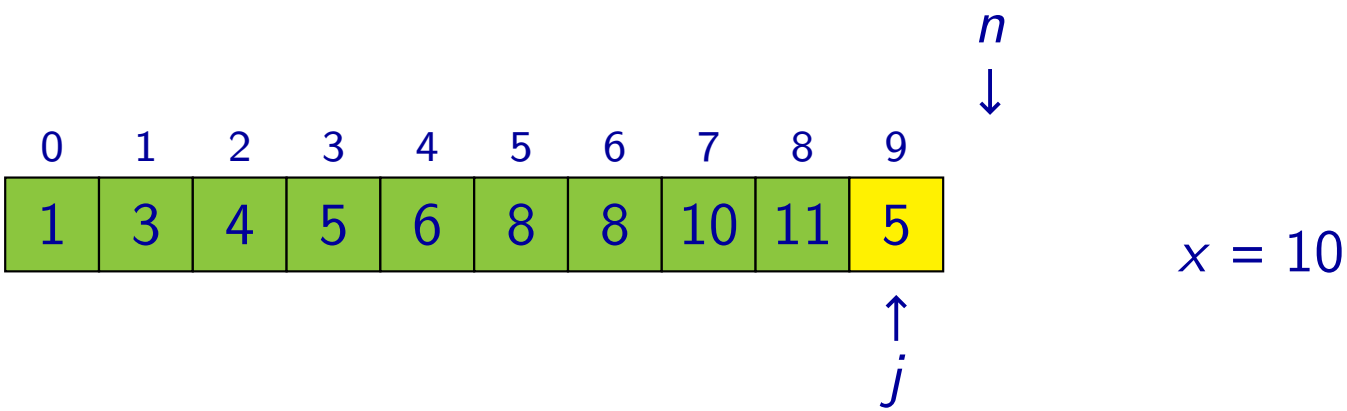
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



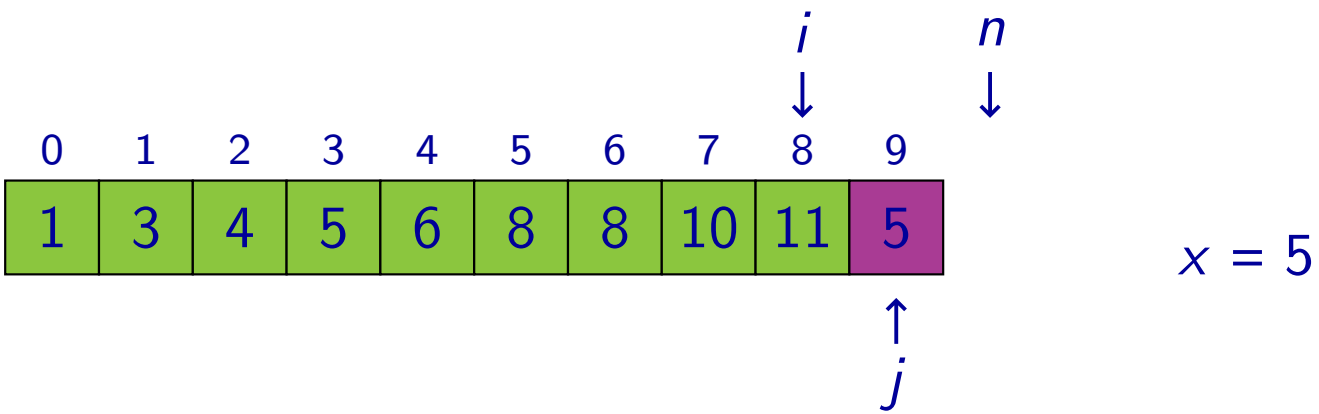
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



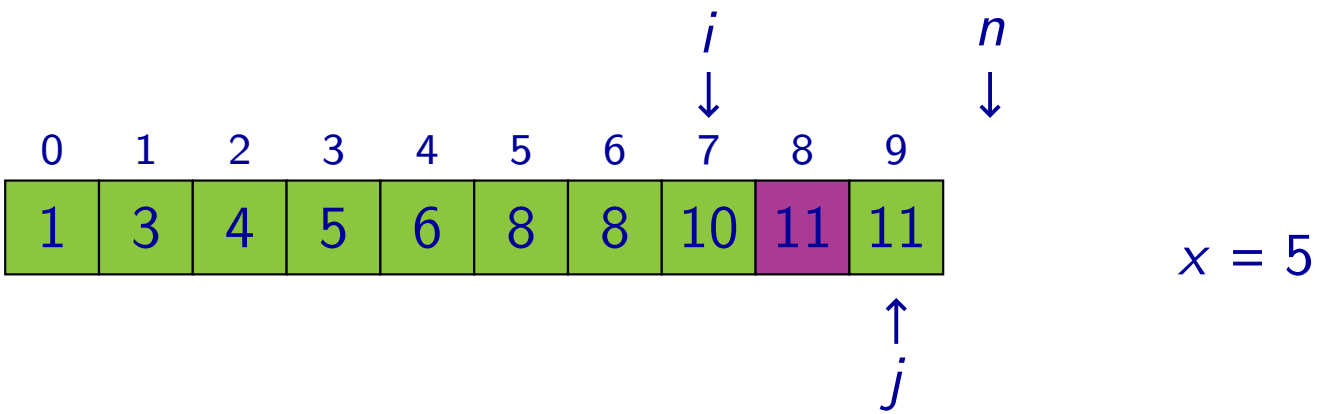
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



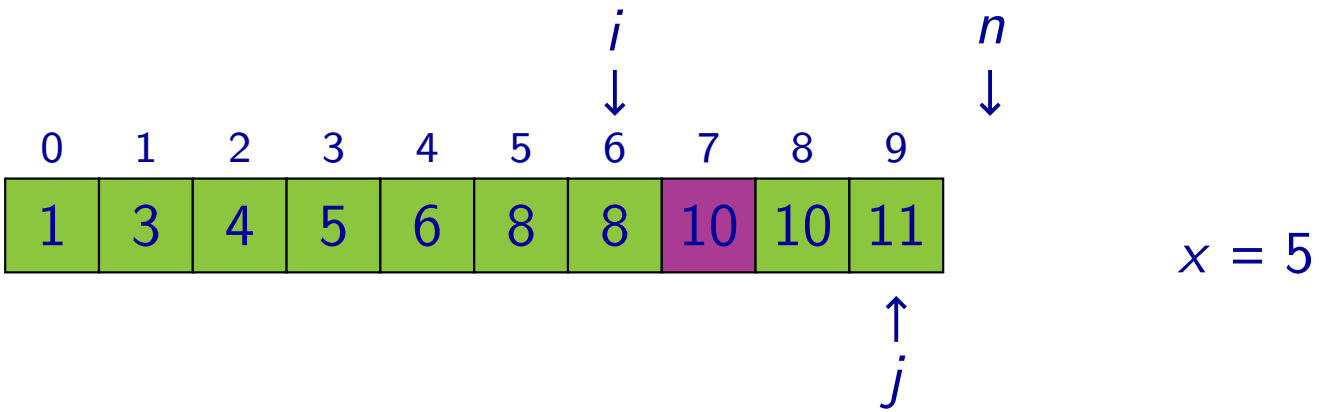
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



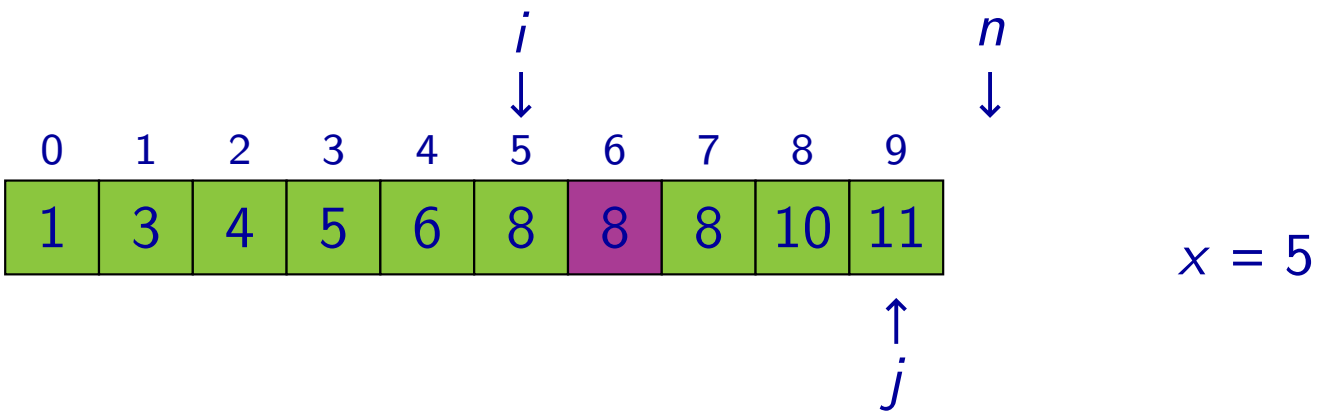
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



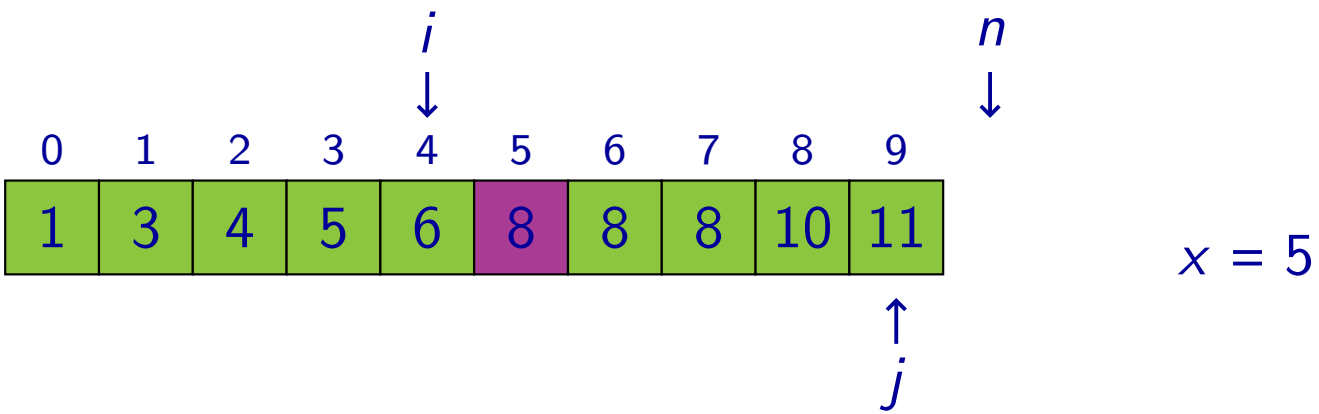
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



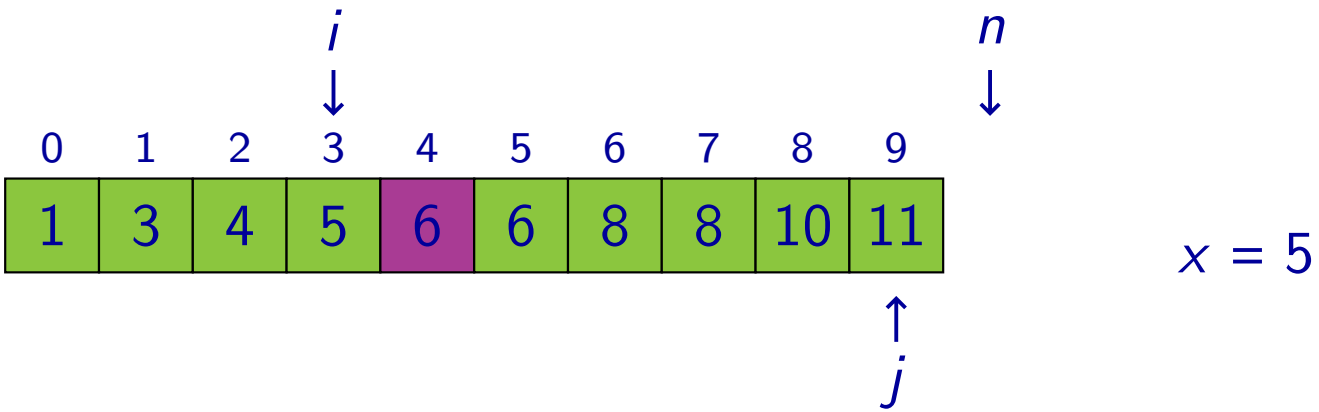
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



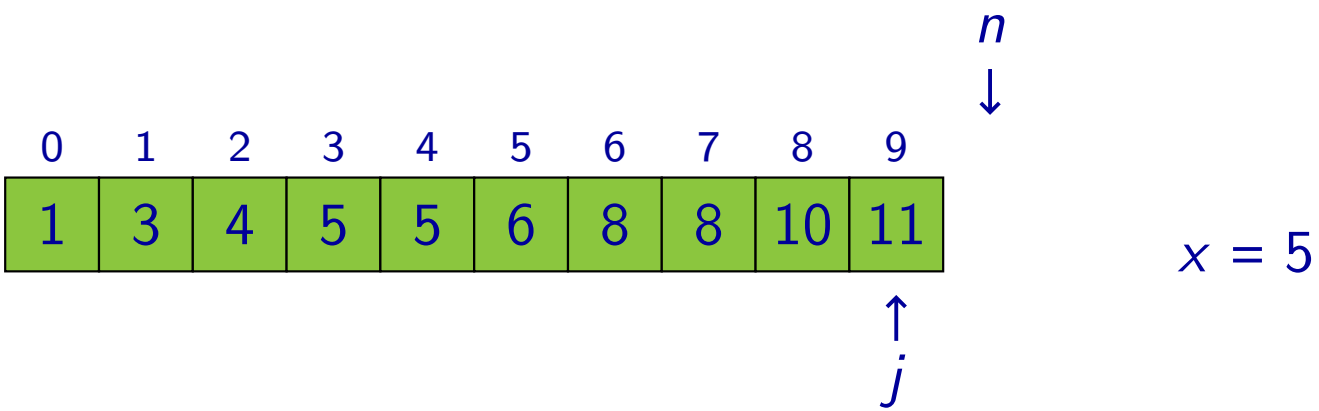
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



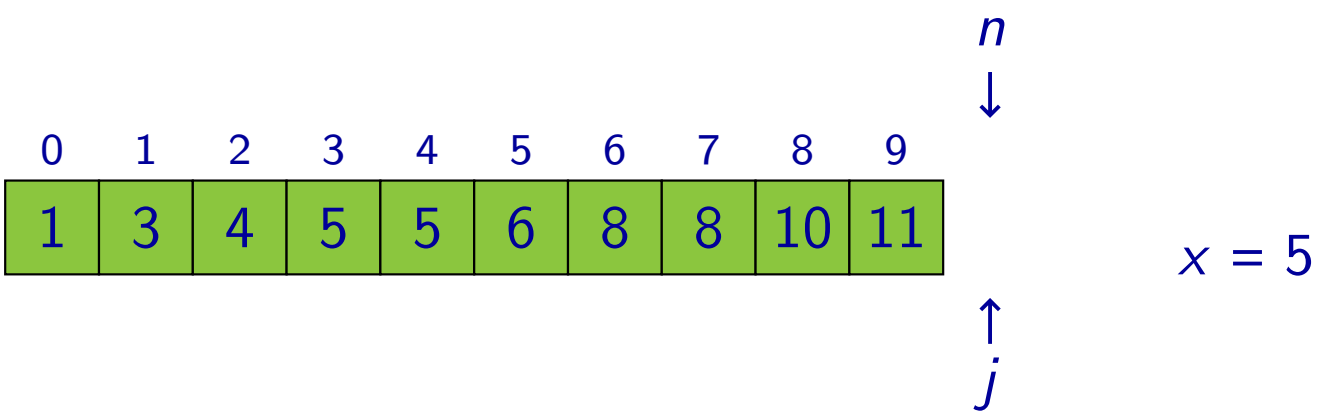
Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



Example: A computation of INSERTION-SORT on input

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



Complexity of Algorithms

Algorithm: Insertion sort

INSERTION-SORT (A, n):

```
for j := 1 to n − 1 do
    x := A[j]
    i := j − 1
    while i ≥ 0 and A[i] > x do
        A[i + 1] := A[i]
        i := i − 1
    A[i + 1] := x
```

Complexity of Algorithms

Let us consider inputs of size n:

- The outer cycle **for** is performed at most $n-1$ times.
  (Variable j takes values $1, 2, \ldots, n-1$.)
- The inner cycle **while** is performed at most j times for a given value j.
  (Variable i takes values $j-1, j-2, \ldots, 1, 0$.)
- There are inputs such that the cycle **while** is performed exactly j times for each value j from 1 to $n-1$.
- So in the worst case, the cycle **while** is performed exactly m times, where

$$m = 1 + 2 + \cdots + (n-1) = (1 + (n-1)) \cdot \frac{n-1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

- This means that the total running time of the algorithm INSERTION-SORT in the worst case is $\Theta(n^2)$.

Complexity of Algorithms

In the previous case, we have computed the total number of executions of the cycle **while** accurately.

This is not always possible in general, or it can be quite complicated. It is also not necessary, if we only want an asymptotic estimation.

Complexity of Algorithms

For example, if we were not able to compute the sum of the arithmetic progression, we could proceed as follows:

- The outer cycle **for** is not performed more than n times and the inner cycle **while** is performed at most n times in each iteration of the outer cycle.

  So we have $T \in O(n^2)$.

- For some inputs, the cycle **while** is performed at least $\lceil n/2 \rceil$ times in the last $\lfloor n/2 \rfloor$ iterations of the cycle **for**.

  So the cycle **while** is performed at least $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ times for some inputs.

  $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \geq (n/2 - 1) \cdot (n/2) = \frac{1}{4}n^2 - \frac{1}{2}n$

  This implies $T \in \Omega(n^2)$.

Complexity of Algorithms

- So far we considered that execution of a given instruction always takes the same time without regard the values, with which it works.
- So when asymptotic notation was used, the time how long it takes to execute an individual instruction played no role and it was only important how many times the given instruction is executed.
- For example, when RAMs are used as a model of computation, this corresponds to counting of instructions executed, i.e., an execution of one instruction takes 1 time unit.

  This is known as using the so called **uniform-cost measurement**.
- Estimations of the time complexity using the uniform-cost measurement correnspond to the running time on real computers under the assumption that operations, performed by the RAM, can be performed by a real computer in a constant time.

  This assumption holds, if numbers, the algorithm works with, are small (they can be stored, say, to 32 or 64 bits).

Complexity of Algorithms

- If the RAM works with "big" numbers (e.g., 1000 bit), the estimation using the uniform-cost measurement will be unrealistic in the sense that a computation on a real computer will take much more time.
- To analyse the time complexity of algorithms working with big numbers, we usually use so called **logarithmic-cost measurement**, where a duration of one instruction is not 1 but is proportional to the number of **bit operations**, which are necessary for an execution of this instruction.
- The duration of an exection of an instruction depends on the actual values of its operands.
- For example, a duration of an execution of instructions for addition and subtraction is equal to the sum of the numbers of bits of their operands.
- The duration of an execution of instructions multiplication and division is equal to the product of the numbers of bits of their operands.

Remark: The notation $blen(x)$ denotes the number of bits in a binary representation of a natural number x.

It holds that

$$blen(x) = \max(1, \lceil \log_2(x+1) \rceil)$$

Space Complexity of Algorithms

- So far we have considered only the time necessary for a computation
- Sometimes the size of the memory necessary for the computation is more critical.

For RAMs and their use of memory, we can again distinguish between the use of uniform-cost and logarithmic-cost measurement:

Amount of memory of a RAM $\mathcal{M}$ used for an input w is the number of memory cells that are used by $\mathcal{M}$ during its computation on w.

Definition

A **space complexity** of a RAM $\mathcal{M}$ (in the worst case) is the function $S : \mathbb{N} \to \mathbb{N}$, where $S(n)$ is the maximal amount of memory used by $\mathcal{M}$ for inputs of size n.

Space Complexity of Algorithms

- There can be two algorithms for a particular problem such that one of them has a smaller time complexity and the other a smaller space complexity.

- If the time-complexity of an algorithm is in $O(f(n))$ then also the space complexity is in $O(f(n))$ (note that a RAM uses at most three cells in each step — at most two for reading and at most one for writing).