Problem solved by an algorithm. Algorithm Complexity

Șt. Ciobâcă, Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
stefan.ciobaca@info.uaic.ro, dlucanu@info.uaic.ro

PA 2019/2020

1. Recap
2. Problem solved by an algorithm
3. Algorithm Complexity
4. The worst case complexity

Anunț

Cursurile din data de 03.03.2020 (marti) se vor tine in avans sambata 29.02.2020, incepand cu ora 8:00, sala C2.

Modificarea apare si in orar:
https://profs.info.uaic.ro/ orar/discipline/orar_pa.html

Plan

Recap from the previous lecture

- Alk = a language for describing algorithms
  - syntax
  - semantics
- algorithm execution (computation)
- cost functions:
  - size of values,
  - time of operations,
  - time of executions (calculation)

Recap: Alk - syntax

- expressions
  - `a * b + 2`, `a < 5`, `(a < 5) && (a > -1)`
  - `l.update(2,55)`, `l.size()`, `f(a*2, b+5)`
  - ...
- statements:
  - X `=` E`;`
  - `if (` E `)` St_1 `else` St_2
  - `while (` E `)` St
  - St_1 St_2
  - $\{\ Sts\ \}$
  - ...

Recap: Alk - semantics

- state: $\sigma = \mathtt{a} \mapsto 3\ \mathtt{b} \mapsto 5\ \mathtt{c} \mapsto -12$
- expressions are evaluated
  $[\![\mathtt{a} + \mathtt{b} * 2]\!](\sigma) = [\![\mathtt{a}]\!](\sigma) +_{Int} [\![\mathtt{b} * 2]\!](\sigma) = 3 +_{Int} [\![b]\!](\sigma) *_{Int} [\![2]\!](\sigma) = 3 +_{Int} 5 *_{Int} 2 = 13$
- statements define execution steps:
  $\langle \mathtt{x = } E\mathtt{;} S, \sigma\}\rangle \Rightarrow \langle S, \sigma[\mathtt{x} \mapsto [\![E]\!](\sigma)]\rangle$
  $\langle \mathtt{if\ (}E\mathtt{)}\ S\ \mathtt{else}\ S'\ \ S'', \sigma\rangle \Rightarrow \langle S\ \ S'', \sigma\rangle$ if $[\![E]\!](\sigma) = \mathit{true}$
  $\langle \mathtt{if\ (}E\mathtt{)}\ S\ \mathtt{else}\ S'\ \ S'', \sigma\rangle \Rightarrow \langle S'\ \ S'', \sigma\rangle$ if $[\![E]\!](\sigma) = \mathit{false}$
- execution (computation) = sequence of execution steps:
  $\tau = \langle S_1, \sigma_1\rangle \Rightarrow \langle S_2, \sigma_2\rangle \Rightarrow \langle S_3, \sigma_3\rangle \Rightarrow \ldots$

Recap: cost functions

- size of values:
  - $|n|_{\mathrm{unif}} = O(1)$, $|n|_{\mathrm{log}} = \log_2 abs(n)$, $|n|_{\mathrm{lin}} = \log_2 abs(n)$
  - $|\langle v_0, v_1, \ldots, v_{n-1}\rangle|_d = |v_0|_d + |v_1|_d + \cdots + |v_{n-1}|_d$, $d \in \{\mathrm{unif}, \mathrm{log}, \mathrm{lin}\}$
- time of operations:
  - $time_{\mathrm{unif}}(a *_{Int} b) = O(1)$, $time_{\mathrm{log}}(a *_{Int} b) = O(\max(\log a, \log b)^{1.545})$
  - $time_{\mathrm{unif}}(L.\mathtt{insert}(i,x)) = O(i)$,
    $time_{\mathrm{log}}(L.\mathtt{insert}(i,x)) = O(\log(1 + \cdots + i) + |x|_{\mathrm{log}})$
- time of execution steps:
  $time_d(\langle \mathtt{if}\ \ (E)\ \ S'\ \ \mathtt{else}\ \ S''\ \ S, \sigma\rangle \Rightarrow \langle \_, \sigma\rangle) = time_d([\![E]\!](\sigma))$
  $d \in \{\mathrm{unif}, \mathrm{log}\}$
- time of executions:

$$\tau = \langle S_1, \sigma_1\rangle \Rightarrow \langle S_2, \sigma_2\rangle \Rightarrow \langle S_3, \sigma_3\rangle \Rightarrow \ldots$$
$$time_d(\tau) = \sum_i time_d(\langle S_i, \sigma_i\rangle \Rightarrow \langle S_{i+1}, \sigma_{i+1}\rangle),\ d \in \{unif, log\}$$

NB Notations $|v|_d$ and $size_d(v)$ are equivalent.

Uniform Time vs. Logarithmic Time in Digits

```
s = 0 ;
while (n > 0) {
  s = s + n;
  n = n - 1;
}
```

Logaritmic time: 33384

Uniform time: 3001

Plan

Computational Problem

A problem to be solved by an algorithm is represented by:

- problem domain
- a pair $(input, output)$

Notation: $p \in P$ denotes the fact that p is an *instance* (input component) of the problem P and $P(p)$ denotes the result (output component)

Example: Problem Plateau 1/2

Problem domain: Let $a = (a_0, \ldots, a_{n-1})$ a finite sequence of integers.
A segment $a[i..j]$ of a is the sequence $(a_i, \ldots, a_j)$, where $i \leq j$.
If $i > j$ then we may assume that $a[i..j]$ is the empty sequence.
The length of a segment $a[i..j]$ is $j + 1 - i$.
A plateau is a segment whose elements are equal.
The sequence a is nondecreasing if $a_0 \leq \ldots \leq a_{n-1}$.

Input: An nondecreasing sequence $a = (a_0, \ldots, a_{n-1})$ of integers of length n.

Output: The length of the longest plateau of a.

Example: Problem Plateau 2/2

Often it is helpful to represent the pair $(input, output)$ using predicates:

$plateau(a, i, j)$: $(\forall k) i \leq k \leq j \implies a_i = a_k$

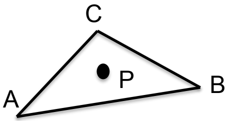
$nondecreasing(a)$: $a_0 \leq \ldots \leq a_{n-1}$
The next implication helps to find a simple solution:
$nondecreasing(a) \implies (plateau(a, i, j) \wedge i \leq j \iff a_i == a_j)$

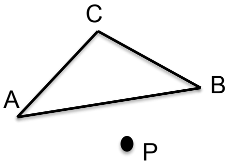
Input: $a = (a_0, \ldots, a_{n-1}) \wedge nondecreasing(a)$.

Output: $q \in \mathbb{Z} \wedge$
$(\exists 0 \leq i \leq j < n) plateau(a, i, j) \wedge q = j + 1 - i \wedge$
$(\forall 0 \leq k \leq \ell < n) plateau(a, k, \ell) \implies q \geq (\ell + 1 - k)$.

formal *input* $\equiv$ precondition
formal *output* $\equiv$ postcondition
(*precondition*, *postcondition*) $\equiv$ specification

Example: Position of a point w.r.t. a Triangle 1/5





Input: A triangle (A, B, C) and a point P, both in the plane.
Output: The answer to the question: Does P lie inside the triangle?

Example: Position of a point w.r.t. a Triangle 2/5

It is essential to investigate the problem domain!!!

Let A, B, C be three points.

$$det(A, B, C) = \begin{vmatrix} \texttt{A.x} & \texttt{A.y} & 1 \\ \texttt{B.x} & \texttt{B.y} & 1 \\ \texttt{C.x} & \texttt{C.y} & 1 \end{vmatrix}$$

- $det(A, B, C) > 0$: A, B, C form a counter-clock-wise cycle (left turn)
- $det(A, B, C) < 0$: A, B, C form a clock-wise cycle (right turn)
- $det(A, B, C) = 0$: A, B, C are colinear

Convention: $det(A, B, C) \equiv \texttt{sign2xTriArea}(\texttt{A}, \texttt{B}, \texttt{C})$

Example: Position of a point w.r.t. a Triangle 3/5

```
sign2xTriArea(A, B, C) {
  d1 = B.y * A.x + C.y * B.x + A.y * C.x;
  d2 = C.x * B.y + B.x * A.y + A.x * C.y;
  return d1 - d2;
}

ccw(A, B, C)
/*
  turn left = +1;
  turn right = -1;
  colinear = 0;
*/
{
  ax2 = sign2xTriArea(A, B, C);
  if (ax2 > 0.0) return 1;
  if (ax2 < 0.0) return -1;
  return 0;
}
```

Example: Position of a point w.r.t. a Triangle 4/5

A test:

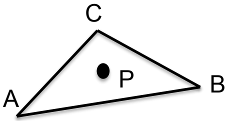
```
A = {x -> 1.0 y -> 1.0};
B = {x -> 3.0 y -> 1.0};
C = {x -> 2.0 y -> 2.0};

print(ccw(A, B, C));
print(ccw(A, C, B));
print(ccw(A, B, B));
```

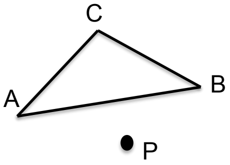
Running the test:

```
$ ../../Linux_Mac/alki.sh -a triangle.alk
1
-1
0
A |-> {x->1.0 y->1.0}
B |-> {x->3.0 y->1.0}
C |-> {x->2.0 y->2.0}
```

Example: Position of a point w.r.t. a Triangle 5/5



`ccw(P, A, B)`, `ccw(P, B, C)` and `ccw(P, C, A)` have the same sign



`ccw(P, A, B)`, `ccw(P, B, C)` and `ccw(P, C, A)` do NOT have the same sign Exercise. Write the algorithm that decide the position of a point w.r.t. a triangle.

Problem solved by an algorithm

An algorithm A solves a problem P if:

- for any instance (input) p of P, there is an initial configuration $\langle A, \sigma_p \rangle$ such that σ_p includes data structures describing p;
- the execution starting from the initial configuration $\langle A, \sigma_p \rangle$ ends into a final configuration $\langle \cdot, \sigma' \rangle$, write $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$; and
- σ' includes data structures that describes the output $P(p)$.

Problem solved by an algorithm, more formally

An asertion ϕ is a formula with predicates whose variables are program variables occurring in the algorithm.

Let $\sigma \models \phi$ denote the fact the variables values given by σ satisfy ϕ.
Example: if $\sigma = \mathrm{x} \mapsto 3\ \mathrm{y} \mapsto 5$, then $\sigma \models 2 * \mathrm{x} > \mathrm{y}$ și $\sigma \not\models \mathrm{x} + \mathrm{y} < 0$.

Preconditions and postconditions are assertions.

A solves *P*, specified by $(pre, post)$, iff for any state σ, if $\sigma \models pre$ then there is σ' such that $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$ and $\sigma' \models post$.

The pair $(pre, post)$ represents the specification (*precondition*,*postcondition*).

Solving versus Correctness

If A solves P given by (*precondition*,*postcondition*), we often say that A is correct (w.r.t. (*precondition*,*postcondition*)).

The two notions are not equivalent.

There are two kinds of correctness:

total correctness: for any state σ, if $\sigma \models pre$ then there is σ' such that $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$ and $\sigma' \models post$.

partial correctness: for any state σ, if $\sigma \models pre$ and there is σ' such that $\langle A, \sigma \rangle \Rightarrow^* \langle ., \sigma' \rangle$, then $\sigma' \models post$.

For total correctness the termination of the execution must be proved, for the partial correctness the final configuration satisfies the *postcondition* only when the execution starting from σ is finite.

Solving is equivalent with the total correctness.

The algorithm PlateauAlg

Assume that the sequence a is represented by the array a $\mapsto [a_0, \ldots, a_{n-1}]$. An algorithm proposed to solve Plateau is:

```
lg = 1;
i = 1;
while (i < n) {
  if (a[i] == a[i - lg]) lg = lg+1;
  i = i + 1;
}
```

The relationship between PlateauAlg and the specification of the Plateau problem

To prove that PlateauAlg solves Plateau, we have to show that any execution starting from initial configuration:

$\langle \mathit{PlateauAlg},\ \mathtt{n} \mapsto n\ \mathtt{a} \mapsto [a_0, \ldots, a_{n-1}]\rangle$

with $a_0 \leq \cdots \leq a_{n-1} \wedge n \geq 1$ (i.e., it satisfies the precondition)

stops in the final configuration:

$\langle \cdot,\ \mathtt{n} \mapsto n\ \mathtt{a} \mapsto [a_0, \ldots, a_{n-1}]\ \mathtt{i} \mapsto n\ \mathtt{lg} \mapsto q\rangle$

and ℓ is the length of the longest plateau in a:

$$(\exists 0 \leq i \leq j < n) \mathit{plateau}(a, i, j) \wedge q = j + 1 - i\ \wedge$$
$$(\forall 0 \leq k \leq \ell < n) \mathit{plateau}(a, k, \ell) \implies q \geq (\ell + 1 - k)$$

(i.e., it satisfies the postcondition)

How the correctness is proved?

How do we prove that PlateauAlg solves indeed Plateau?

A possible solution:

- we prove that that at the beginning and at the end of the while loop the property "`lg` the length of the longest plateau in a$[0..i-1]$" holds

$$\exists i_0, j_0 . 0 \leq i_0 \leq j_0 < i \implies platou(a, i_0, j_0) \wedge \mathtt{lg} = j_0 + 1 - i_0 \wedge$$
$$\forall k, \ell . 0 \leq k \leq \ell < i \implies platou(a, k, \ell) \implies \mathtt{lg} \geq (\ell + 1 - k)$$

- this property is called a loop invariant

- at the end of the while statement, the invariant and the negation of the while condition ($i \geq n$) hold; if we show that also $i \leq n$ is an invariant, then after the execution of while we have $i = n$ and the invariant becomes the postcondition.

How the invariant is proved?

There are two cases:

1. $j_0 = i - 1$ (the longest plateau in $a[0..i-1]$ ends in $i-1$). There are two subcases:

1.1 it holds $plateau(a, i_0, i)$, the plateau $a[i_0..j_0]$ increases with one unit, and $a[i_0..i]$ the longest plateau in $a[0..i]$ (why?); hence `lg` is incremented;

1.2 $plateau(a, i_0, i)$ does not hold, i.e. $a[i-1] < a[i]$; the logest plateau in $a[0..i-1]$ will be the longest plateau in $a[0..i]$ as well; hence `lg` is not modified;

2. the longest plateau in $a[0..i-1]$ DOES NOT end in $i-1$. It follows that the longest plateau that ends in $i-1$ is $<$ `lg`, hence the length of the plateau that ends in i is $\leq$ `lg`; therefore `lg` is not modified.

Solvable (Computable) Problem

A problem P is solvable (computable) if there is an algorithm A that solves P.

A problem P is non-solvable (non-computable) if it DOES NOT exists an algorithm A that solves P.

Decision Problems

A decision problem P has the answer (output) of the form "YES" or "NO" (equivalently, "true" or "false"). More precisely, for any instance $p \in P$, $P(p) \in \{"YES", "NO"\}$ ($P(p) \in \{"true", "false"\}$).

A decision problem is generally presented by a pair (instance, question).

O decidable problem is a decision problem that is solvable.

O undecidable problem is a decision problem that is unsolvable.

Are all the computational problems decidable/solvable?

At the beginning of the 20th century, the mathematieciens believed that yes.

In 1931, Kurt Gödel shocked by proving that this is impossible. He showed that if we have a system with a well-defined behaviour and strong enough to include the mathematical reasoning, then there are statements that cannot be proved with this system as being true, even if they are indeed true. (the famous the Gödel's incompleteness theorem).

Later, Alan Turing proved the same thing using the notion of algorithm (Turing machine).

On the next slides we present an example of undecidable problem.

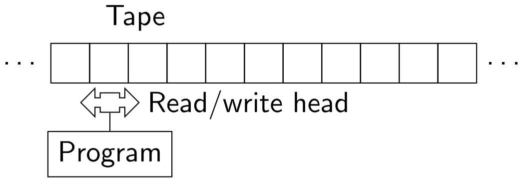
The universal program (algorithm)

Starting from the notion of universal Turing machine, we may define the notion of universal program (algoritm): this has as input a program (algorithm) A and an input x for A (i.e. the initial configuration $\langle A, \sigma_x \rangle$), simulates the behaviour of A on x (starting from $\langle A, \sigma_x \rangle$).

The programs (algorithms) can be inputs for other algorithms!

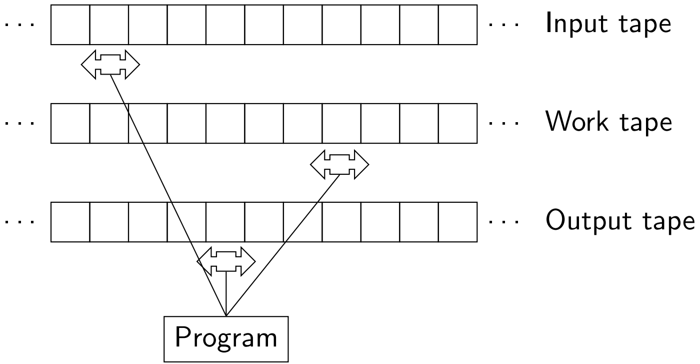
Universal Turing Machine 1/3

Turing Machine:



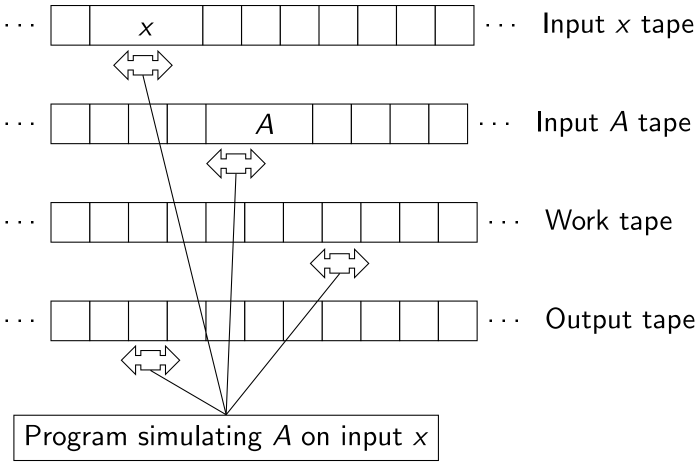
Universal Turing Machine 2/3

Turing Machine with multiple tapes:



Universal Turing Machine 3/3

Universal Turing Machine:



Example of undecidable problem

Halting Problem:
Instance: A configuration $\langle A, \sigma_0 \rangle$, where A is an algorithm.
Question: Does the execution starting from $\langle A, \sigma_0 \rangle$ terminates?

Teoremă

There is no algorithm solving Halting Problem.

The proving idea 1/2

Reductio ad absurdum. Assume that there is an algorithm H that solves Halting Problem. A call of H on the input A, x is denoted by $H(A, x)$.

Consider the following algorithm calling H:

```
NewH(A) {
  if H(A, A) return true;
  else return false;
}
```

and another one calling NewH:

```
HaltsOnSelf (A) {
  if NewH(A) while (true) {}
  else return false;
}
```

The proving idea 2/2

What happens when `HaltsOnSelf(HaltsOnSelf)` is called?

The execution of `HaltsOnSelf(HaltsOnSelf)` does not terminate; it follows that `NewH(HaltsOnSelf)` returns `true`, which implies that `H(HaltsOnSelf,HaltsOnSelf)` returns `true`. Contradiction.

The execution of `HaltsOnSelf(HaltsOnSelf)` terminates and returns `true`; it follows that `NewH(HaltsOnSelf)` returns `false`, which implies that `H(HaltsOnSelf,HaltsOnSelf)` returns `false`. Contradiction again.

The above theorem is strong related to the following logic paradox (Russel's paradox):
The barber is the "one who shaves all those, and those only, who do not shave themselves." The question is, does the barber shave himself? Who shaves the barber?"

Other examples of undescidable problems

Equivalence of programs

Totality: if a program (algorithm) stops on all its inputs

Total correctness

Hilbert's tenth problem

...

Theorem (Rice,1953)

All non-trivial questions about the behaviour of programs from a universal programming language are undecidable.

Partial solvabil (computable, decidable)

A decision problem is partial computable (semi-decidable) if there is an algorithm that stops with the answer "YES" for all the inputs with the answer "YES".

Is Halting Problem semi-decidable?

Plan

The time of an execution (recall)

Let $E = \langle A_0, \sigma_0 \rangle \Rightarrow \cdots \Rightarrow \langle A_n, \sigma_n \rangle$ an execution.

The time required by this execution is:

$$time_d(E) = \sum_{i=0}^{n-1} time_d(\langle A_i, \sigma_i \rangle \Rightarrow \langle A_{i+1}, \sigma_{i+1} \rangle)$$

where $d \in \{log, unif, lin\}$.

The execution time for an instance: the deterministic case

Deterministic algorithm: for any reachable configuration $\langle A_i, \sigma_i \rangle$ there is at most a successor configuration, i.e., at most a $\langle A', \sigma' \rangle$ such that $\langle A_i, \sigma_i \rangle \Rightarrow \langle A', \sigma' \rangle$.

All algorithms we considered up to now are deterministic! Later we shall consider nondeterministic algorithms as well.

Let P be a problem and A a deterministic algorithm that solves P.

For each $p \in P$ there is an unique execution path E_p.

The time for computing $P(p)$ is

$$time_d(A, p) = time_d(E_p)$$

where $d \in \{log, unif, lin\}$.

Plan

The size of an instance

The dimension of a state σ is

$$size_d(\sigma) = \sum_{\mathtt{x} \mapsto v \in \sigma} size_d(v)$$

The dimension of a configuration is

$$size_d(\langle A, \sigma \rangle) = size_d(\sigma)$$

where $d \in \{log, unif, lin\}$.

Let P be a problem, $p \in P$, and A a deterministic algorithm that solves P.

The size of p is the the size of its intial configuration:

$$size_d(p) = size_d(\langle A, \sigma_p \rangle) \quad (= size(\sigma_p))$$

where $d \in \{log, unif, lin\}$.

The worst case time complexity

Let P be a problem and A a determinstic algorithm that solves P and fix $d \in \{log, unif, lin\}$.

Group the instances p of P into equivalences classes: p and p' are in the same equivalence class iff $size(p) = size(p')$.

A natural number n can be seen as the equivalence class of instances p of size n ($size_d(p) = n$).

The worst case time complexity:

$$T_{A,d}(n) = \max\{time_d(A, p) \mid p \in P, size_d(p) = n\}$$

Space complexity

Let $E = \langle A_0, \sigma_0 \rangle \Rightarrow \cdots \Rightarrow \langle A_n, \sigma_n \rangle$ be an execution and fix $d \in \{log, unif, lin\}$

The space used by this execution is:

$$space_d(E) = \max_{i=0}^{n} size_d(\langle A_i, \sigma_i \rangle)\}$$

The space required by the algorithm A solve the instance $p \in P$ is

$$space_d(A, p) = space_d(E_p)$$

where E_p is the execution corresponding to p.

The worst case space complexity:

$$S_{A,d}(n) = \max\{space_d(A, p) \mid size_d(p) = n\}$$

Computing the worst case time complexity 1/3

- A is an expression E without function (algorithm) calls:
  $T_{A,d}(n) = \max\{time_d(\llbracket E \rrbracket(\sigma)) \mid size_d(\sigma|_{var(x)}) = n\}$ (a kind of)
- A is an assignment $X = E$;
  $T_{A,d}(n) = T_{E,d}(n)$
- A is `if (`E`)` S_1 `else` S_2:
  $T_{A,d}(n) = \max\{T_{S_1,d}(n), T_{S_2,d}(n)\} + T_{E,d}(n)$
- A is a sequential composition $S_1\ S_2$:
  $T_{A,d}(n) = T_{S_1,d}(n) + T_{S_2,d}(n)$

Computing the worst case time complexity 2/3

- A is an iterative instruction (e.g.,`while, for`): only an approximation can be computed
  - solution 1 (better approximation):
    - compute the maximum number of iterations $nMax$
    - compute the worst case time complexity for each iteration, say $T_1, \ldots, T_{nMax}$
    - take $T_{A,d}(n) = T_1 + \ldots + T_{nMax}$
  - solution 2 (coarser approximation):
    - compute the maximum number of iterations $nMax$
    - find the worst case iteration and compute the worst case time complexity for this iteration, say T_{itMax}
    - take $T_{A,d}(n) = nMax \times T_{itMax}$

Computing the worst case time complexity 3/3

- Atention to the cases of lists, sets, . . . :

```
s = 0;
for(i = 0; i < l.size(); ++i)  // l is a linear list
  s = s + l.at(i);
```

```
s = emptySet;
forall x in a   // a is a set
  if (x % 2 == 0) s = s U singletonSet(x);
```

You have to mention the complexity of each operation.

- Function (algorithm) calls:
  - estimate the size of arguments as a function depending on the size of instance n
  - use the worst case time of the called algorithm computed for the estimated size of the arguments

Computing the worst case time complexity in practice

- usualy the uniform case is used
- only a part of operations are counted (e.g., comparisons, assignments)
- the function classes $O(f(n))$, $\Omega(f(n))$,$\Theta(f(n))$ are used to give approximations for $T_{a,d}(n)$

Recall:

$$O(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\}$$
$$\Omega(f(n)) = \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\}$$
$$\Theta(f(n)) = \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0)c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}$$

Example 1

input: $n, (a_0, \ldots, a_{n-1}), z$ integers.

output: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{if } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{otherwise.} \end{cases}$

```
i = 0;
while (a[i] != z) and (i < n-1)
  i = i+1;
if (a[i] == z) poz = i;
else poz = -1;
```

Analiza Exemplul 1

- type of cost: uniform
- size of an instance: n
- operations counted: comparisons between array elements
- the worst case: z ocurs first time on position $n - 1$ or does not occur in a
- a while loop: 1 comparație
- the number of iterations for the worst case: $n - 1$
- execution time for the worst case: $T_A(n) = (n - 1) + 1 = n$

Example 2

input: $n, (a_0, \ldots, a_{n-1})$ integers.
output: $max = \max\{a_i \mid 0 \leq i \leq n-1\}$.

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

Discussion on the blackboard.

Example 3

input: $n, (a_0, \ldots, a_{n-1})$ integers.
output: $(a_{i_0}, \ldots, a_{i_{n-1}})$ where $(i_0, \ldots, i_{n-1})$ is a permutation of the sequence $(0, \ldots, n-1)$ and $a_{i_j} \leq a_{i_{j+1}}, \forall j \in \{0, \ldots, n-2\}$.

```
for (k = 1; k < n;  k++) {
  temp = a[k];
  i = k - 1;
  while (i >= 0 and a[i] > temp) {
    a[i+1] = a[i];
    i = i-1;
  }
  a[i+1] = temp;
}
```

Discussion on the blackboard.

Example 4

input: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi;
$(a_0, \ldots, a_{n-1})$ is an increasing sequence,

output: $poz = \begin{cases} k \in \{i \mid a_i = z\} & \text{if } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{otherwise.} \end{cases}$

```
  istg = 0;
  idr = n - 1;
  while (istg <= idr ) {
    imed = (istg + idr) / 2;
    if (a[imed] == z)
      return imed
    else if (a[imed] > z)
      idr = imed-1;
    else
      istg = imed + 1;
  }
return -1
```

Discussion on the blackboard.