ALGORITHMS AND DATA STRUCTURES I

Course 2

Previous course

- PROBLEM-SOLVING
- WHAT IS AN ALGORITHM?
- PROPERTIES OF AN ALGORITHM
- HOW TO DESCRIBE ALGORITHMS
- TYPES OF DATA USED
- SPECIFYING OPERATIONS IN AN ALGORITHM

Don’t forget to think at .... when an algorithm is used

- **Name & purpose**: one sentence on what problem it solves.
- **Inputs**: names, types, and constraints.
- **Outputs**: exact result and type.
- **Preconditions**: what must be true before running.
- **Postconditions**: what must be true after
- **High-level idea**: 2–4 lines explaining the strategy (intuition).
- **Pseudocode**: clear, language-agnostic steps - an actual implementation.
- **Complexity**: time and space, with the dominant term and brief reasoning.
- **Correctness sketch**: invariants or proof idea (why it works).
- **Edge cases**: empty inputs, duplicates, ties, overflow, non-ASCII, etc.
- **Example**: a tiny, worked input→output trace.

Course content

Some simple examples

Subalgoritms

Specification

Usage

… If first example is too simple … think at

- How to find Greatest Comon Divider for a sequence of numbers
  - Propose a variant to solve it
  - What properties of the date determine the operations number
  - Identify some cases when
    - The minim number of operations is done
    - The maximum number of operation is done

Gathering some information about students at the end of the semester

- Lets consider the following information about faculty students

- Requirement
  - Fill the state and mean columns based on the following rules
    - State is 1 if credits number is 60
    - State is 2 if credits number is between [30, 60)
    - State is 3 if credits less then 30
    - The mean is filled only if the student has 60 credits

No	ID	Mark 1	Mark 2	Mark 3	Credits no.	State	Mean
1	IE20251	8	6	7	60		
2	IE20252	5	-	8	40		
3	IE20253	10	10	10	60		
4	IE20254	5	6	5	60		
5	IE20255	-	10	-	20		

Gathering some information about students at the end of the semester

- Lets consider the following information about faculty students

- Requirement
  - Fill the state and mean columns based on the following rules
    - State is 1 if credits number is 60
    - State is 2 if credits number is between [30, 60)
    - State is 3 if credits less then 30
    - The mean is filled only if the student has 60 credits

No	ID	Mark 1	Mark 2	Mark 3	Credits no.	State	Mean
1	IE20251	8	6	7	60	1	7
2	IE20252	5	-	8	40	2	-
3	IE20253	10	10	10	60	1	10
4	IE20254	5	6	7	60	1	6
5	IE20255	-	10	-	20	3	-

The expected output for this example

Gathering some information about students at the end of the semester

- Which are the input data?
  - Marks values
    - Each student has 3 marks
  - Credits number
    - Each student has 1 credit number
- What data types to choose?
  - Bidimensional table with integer values for storing the marks
    - Integer marks[1..3][1..5]
  - Unidimensional table with integer values for storing the credits number
    - Integer credits[1..5]

No	ID	Mark 1	Mark 2	Mark 3	Credits no.	State	Mean
1	IE20251	8	6	7	60		
2	IE20252	5	-	8	40		
3	IE20253	10	10	10	60		
4	IE20254	5	6	5	60		
5	IE20255	-	10	-	20		

Gathering some information about students at the end of the semester

- Which are the output data?
  - Student state
  - Student mean

- What data types to choose?
  - Unidimensional table with integer values for storing the state
    - Integer state[1..3][1..5]
  - Unidimensional table with real values for storing the mean
    - Real mean[1..5]

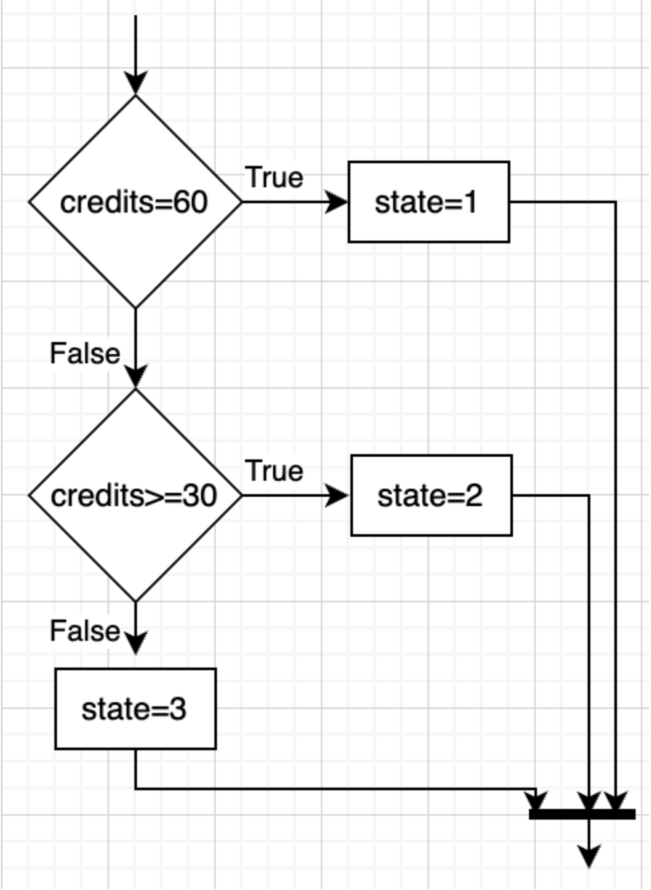
No	ID	Mark 1	Mark 2	Mark 3	Credits no.	State	Mean
1	IE20251	8	6	7	60		
2	IE20252	5	-	8	40		
3	IE20253	10	10	10	60		
4	IE20254	5	6	5	60		
5	IE20255	-	10	-	20		

Gathering some information about students at the end of the semester

- The rule to fill the information regarding sudent state

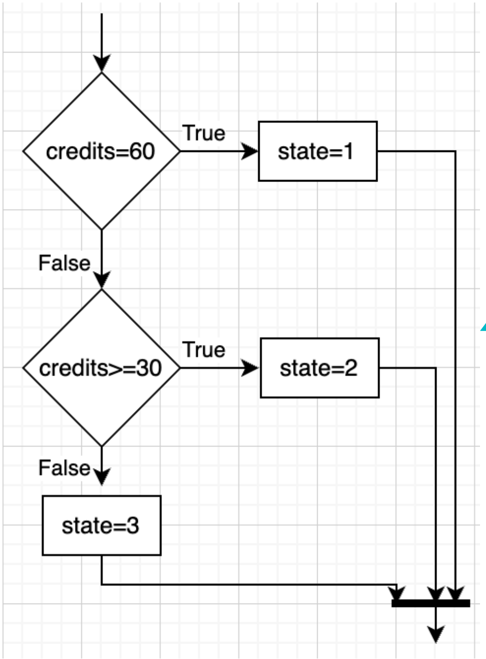
Pseudocode	Python

if credit = 60 then
  state ← 1
else
  if credit >=30 then
    state ← 2
 else
    state ← 3
 endif
endif
	if credit == 60:
	    state = 1
	elif state >= 30:
	    state = 2
	else:
	    state = 3

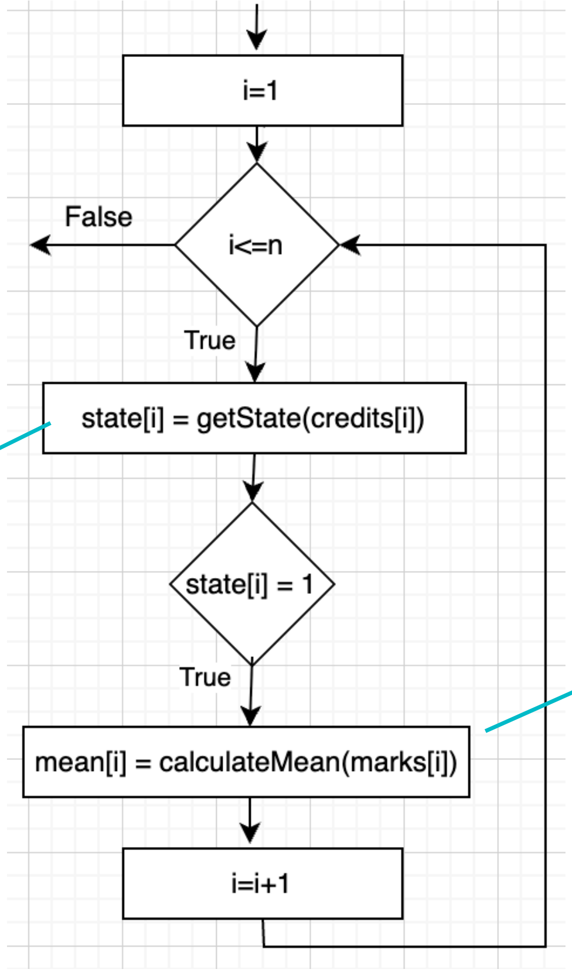


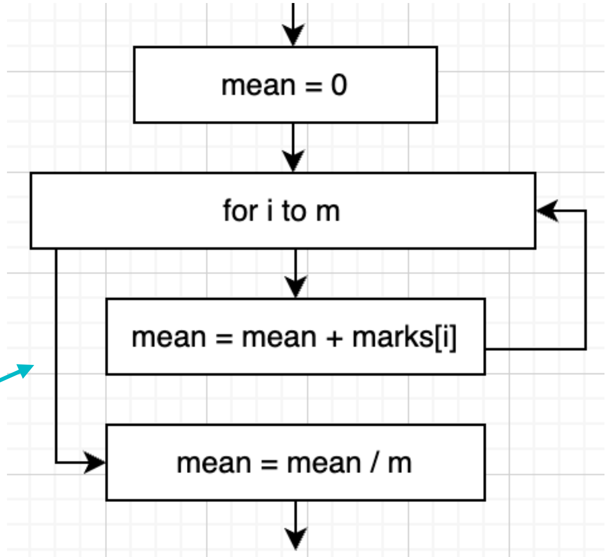
Gathering some information about students at the end of the semester

Applying the rule for all the students (in the example the number of students is equal with 5)



Sequence for finding the status





Sequence for calculating the mean of m marks (in the example the number of marks is equal with 3)

Gathering some information about students at the end of the semester

Applying the rule for all the students (in the example the number of students is equal with 5)

Step1: start from the first line in the table (i=1)

Step2: verifies if all lines were processed (i<n), if yes the algorithm stops

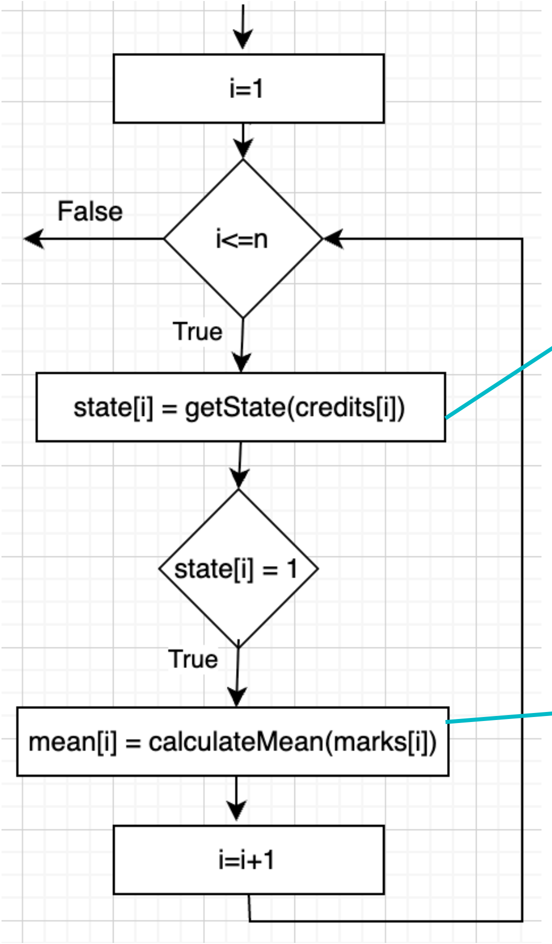
Step 3: calculates and fills the state of the student from line i (state[i]=getState(credits[i]))

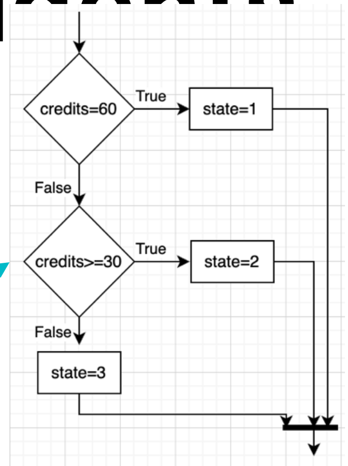
Step 4: verifies if the student i state is 1, if yes goto step 5

Step 5: calculates the mean of the marks of student i (mean[i]=calculateMean(marks[i]))

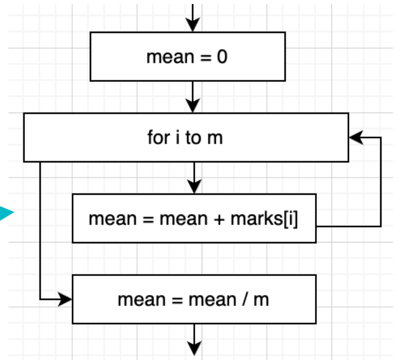
Step 6: calculates the index of the next line (student) (i=i+1)

Step 7: go to step 2





Sequence for finding the status



Sequence for calculating the mean of m marks

Gathering some information about students at the end of the semester

Applying the rule for all the students (in the example the number of students is equal with 5)

Step1: start from the first line in the table (i=1)

Step2: verifies if all lines were processed (i<n), if yes the algorithm stops

Step 3: calculates and fills the state of the student from line i (state[i]=getState(credits[i]))

Step 4: verifies if the student i state is 1, if yes goto step 5

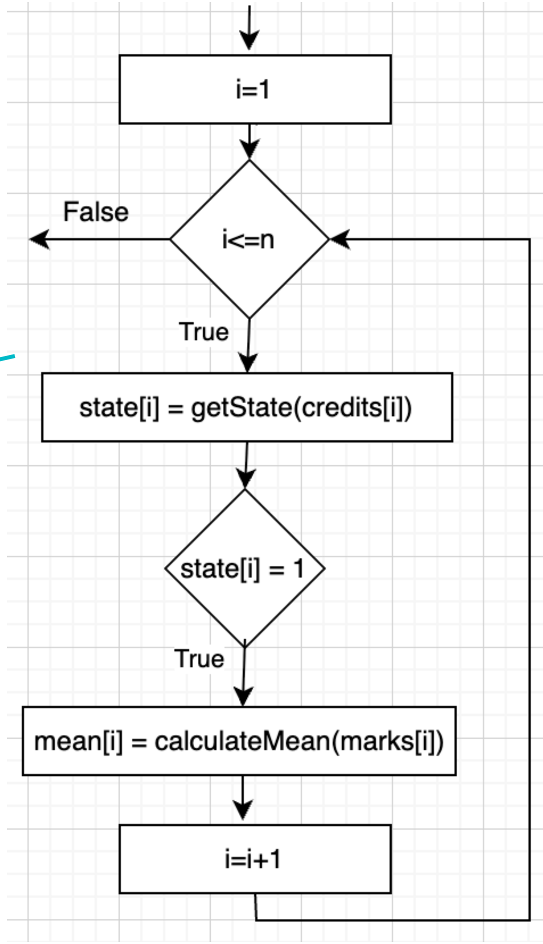
Step 5: calculates the mean of the marks of student i (mean[i]=calculateMean(marks[i]))

Step 6: calculates the index of the next line (student) (i=i+1)

Step 7: go to step 2

Pseudocode

```
integer marks[1..n][1..m], credits[1..n]
integer i, state[1..n]
Real mean[1..n]
i ← 1
while i <=n do
   if credits[i] = 60 then
       st ← 1
   else
        if credits[i] >=30 then
           st ← 2
        else
           st ← 3
        endif
   endif
   state[i] = st
   if state[i] = 1 then mean[i] = calculateMean(marks[i[)
endif
   i  ←  i+1
endwhile
```



Gathering some information about students at the end of the semester

Algorithm simplification using a subalgorithm (function/procedure)

Pseudocode

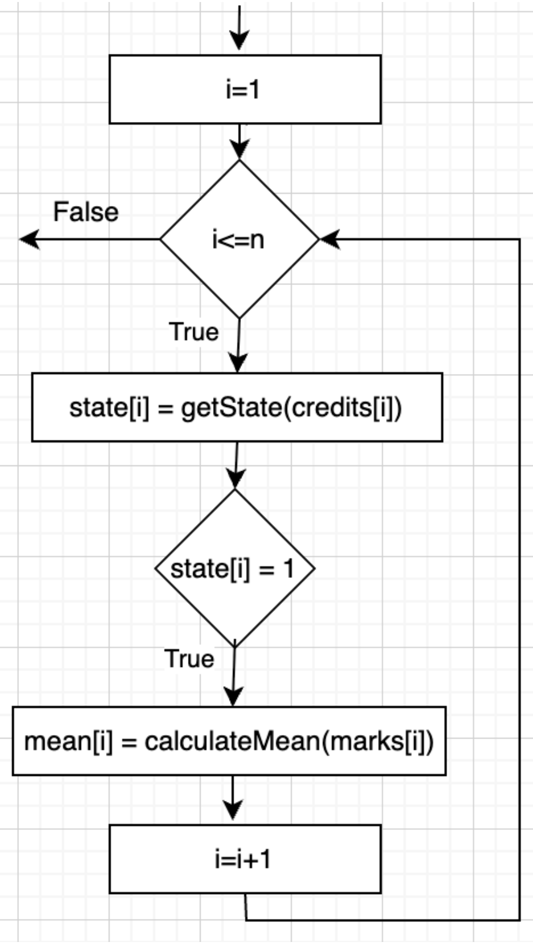
```
integer marks[1..n][1..m], credits[1..n]
integer i, state[1..n]
real mean[1..n]
i ← 1
while i<=n do
   state[i] = getState(credits[i])
   if state[i] = 1 then
      mean[i] = calculateMean(marks[i[)
  endif
   i  ←  i+1
endwhile
```

Pseudocode

```
getState(integer creditNo)
   integer st
   if creditNo = 60 then
      st ← 1
   else
      if creditNo >=30 then
         st ← 2
      else
         st ← 3
      endif
   endif
  return st
```

A subalgorithm is

- used to describe a operation that is performed on it’s input data (parameters)
- characterized by
  - A name (getState())
  - A list of parameters (the input values – integer creditNo)
  - A return type (the output values - st)



Subalgorithms

- The initial problem is decomposed into subproblems
- For each subproblem, an algorithm is designed (called subalgorithm or module or function or procedure)
- The processing within the subalgorithm is applied to generic data (called parameters) and possibly auxiliary data (called local variables)
- The processing specified within the subalgorithm is executed at the time of its call (when the generic parameters are replaced with concrete values)
- The effect of a subalgorithm consists of:
  - Returning one or more results
  - Modifying the values of some parameters (or global variables)

Subalgorithms

- Subalgorithm structure

  <subalgorithm name> (<formal parameters>)

    < local variables declarations>

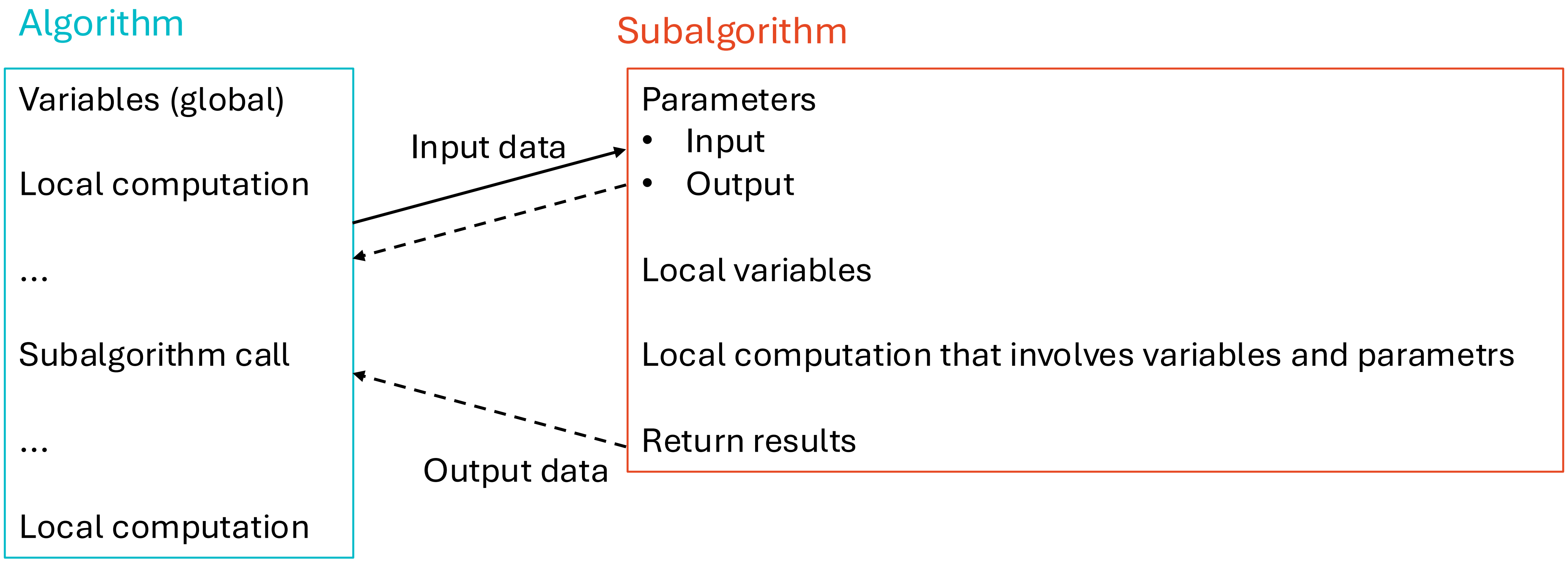
    < computation >

  RETURN <result>

- Subalgorithm usage (call)
  - < subalgorithm name > (<actual parameters>)

Subalgorithms

- How are algorithms and subalgorithms communicating?
  - Return values, parameters, global varaiables



Subalgorithms

- How are algorithms and subalgorithms communicating?
  - Return values, parameters, global varaiables

Input parameter

Algorithm

```
integer marks[1..n][1..m], credits[1..n]
integer i, state[1..n]
real mean.[1..n]
i ← 1
while i <= n do
   state[i] ← getState(credits[i])
   if state[i] = 1 then
       mean[i] ←
calculateMean(marks[i[)
  endif
   i  ←  i+1
endwhile
```

Input data

Output data

Subalgorithm

```
getState(integer creditNo): integer
  integer st
  if creditNo = 60 then st ← 1
  else
      if creditNo >=30 then st ← 2
      else st ← 3
      endif
  endif
 return st
```

Local variable

Return value (rezult)

Previous example. Pseudocode

```
integer marks[1..n][1..m], credits[1..n]

integer i, state[1..m]

real  mean[1..n]

i ← 1

while i <= n do

   state[i] ←  getState(credits[i])

   if state[i] = 1 then  mean[i] ←  calculateMean(marks[i[)  endif

   i  ←  i+1

endwhile
```

Equivalent variant using for instruction

```
integer marks[1..n][1..m], credits[1..n]
integer i, state
for i ← 1, n do
   state[i] ←  getState(credits[i])
   if state[i] = 1 then  mean[i] ←  calculateMean(marks[i])  endif
endfor
```

```
getState(integer creditNo): integer

   integer st

   if creditNo = 60 then st ← 1

   else if creditNo >=30 then st ← 2

   else st ← 3

   endif

   return st


calculateMean(marks[1..m]): real

   integer s

   s ← 0

   for i ← 1,m do

      s ← s+ marks[i]

   endfor

   return s / m
```

Previous example. Python

```
marks = [[8, 6, 7], [5, 0, 8], [10, 10, 10], [5, 6, 5], [0,10,0]]

credits = [60, 40, 60, 60, 20]

i = 0

while i <= n:

   state[i] = getState(credits[i])

   if state[i] == 1:

           mean[i] = calculateMean(marks[i[)
```

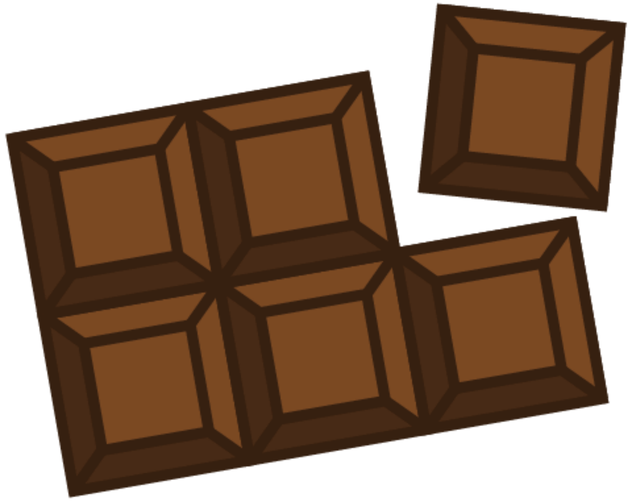
Equivalent variant using for instruction

```
for i in range(n):
   state[i] = getState(credits[i])
   if state[i] == 1:
        mean[i] = calculateMean(marks[i[)
```

```
def getState(creditNo):

   st =  3 //just initialize with a value

   if creditNo == 60: st = 1

   elif creditNo >=30: st = 2

   else st = 3

  return st


def calculateMean(marks):

  s = 0.0

  for i in range(0, len(marks)):

      s  = s+ marks[i]

return s / m
```

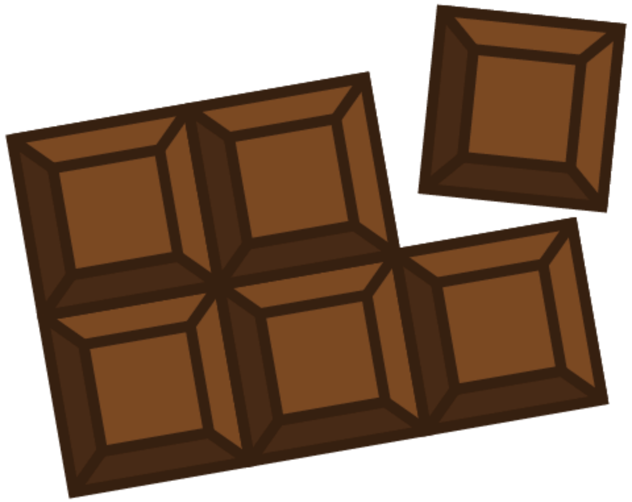
Chocolate Bar Puzzle

I have a chocolate bar that I want to break into pieces (in the case of a 4x6 bar, there are 24 such pieces). How many breaking moves are needed to separate the 24 pieces? (with each move I can break one piece into two more pieces – only along one of the dividing lines of the bar)



Chocolate Bar Puzzle

I have a chocolate bar that I want to break into pieces (in the case of a 4x6 bar, there are 24 such pieces). How many breaking moves are needed to separate the 24 pieces? (with each move I can break one piece into two more pieces – only along one of the dividing lines of the bar)



Answer: 23 (in the case of a mxn tablet the number ofmoves is mxn-1)

How can we prove ?

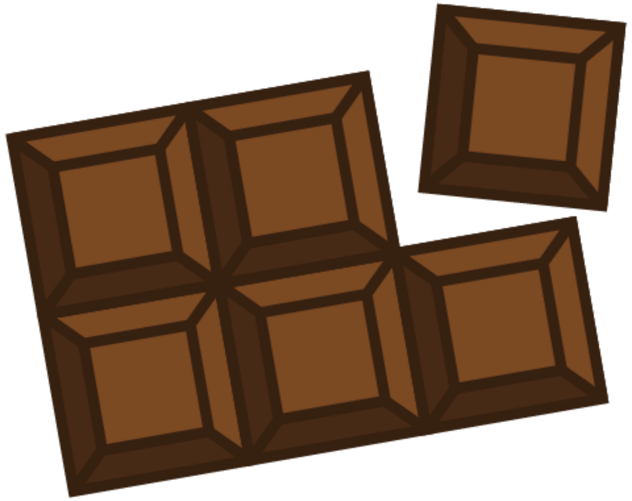
Chocolate Bar Puzzle

By mathematical induction (for a tablet with $N=nxm$

pieces)

Particular case: a single piece ($N=1$) - does not require any breaking (0)

Hypothesis: We assume that for any $K<N$, $K-1$ moves are necessary and sufficient.



To obtain N pieces, proceed as follows:

- Break the tablet into two pieces (with $K_1<N$ and $K_2<N$ pieces, $K_1+K_2=N$) – one move
- Break each of the two pieces into pieces ($K_1-1+K_2-1=K_1+K_2-2$ moves)
- Total: $K_1+K_2-2+1=K_1+K_2-1=N-1$ moves

Example – greatest common divisor

Problem: Let *a* and *b* be two non-zero natural numbers. Determine the greatest divisor of *a* and *b*: gcd(a,b)

Euclid's method (variant based on divisions):

- Calculate the remainder r of the division of a (dividend) by b (divisor)
- Replace
  - the value of the dividend (a) with the value of the divisor (b),
  - the value of the divisor (b) with the value of the remainder r and calculate the remainder of the division of a by b again
- The process continues until a remainder equal to 0 is obtained
- The previous remainder (which is obviously different from 0) will be gcd(a,b).

Example – greatest common divisor

How it works?

Step 1: $a=bq_1+r_1$, $0<=r_1<b$

Step 2: $b=r_1q_2+r_2$, $0<=r_2<r_1$

Step 3: $r_1=r_2q_3+r_3$, $0<=r_3<r_2$

…

Step i: $r_{i-2}=r_{i-1}q_i+r_i$, $0<=r_i<r_{i-1}$

…

Step n-1: $r_{n-3}=r_{n-2}q_{n-1}+r_{n-1}$, $0<=r_{n-1}<r_{n-2}$

Step n : $r_{n-2}=r_{n-1}q_n$, $r_n=0$

Remarks

- at each step the divisor takes the value of the old divider, and the new divisor takes the value of the old remainder
- the sequence of remainders is a strictly decreasing sequence of natural numbers, so there is a value *n* such that $r_n=0$ (the method is finite)
- using these relations it can be proved that r_{n-1} is indeed gcd(a,b)

Example – greatest common divisor

How it works?

Step 1: $a=bq_1+r_1$, $0<=r_1<b$

Step 2: $b=r_1q_2+r_2$, $0<=r_2<r_1$

Step 3: $r_1=r_2q_3+r_3$, $0<=r_3<r_2$

...

Step i: $r_{i-2}=r_{i-1}q_i+r_i$, $0<=r_i<r_{i-1}$

...

Step n-1: $r_{n-3}=r_{n-2}q_{n-1}+r_{n-1}$, $0<=r_{n-1}<r_{n-2}$

Step n : $r_{n-2}=r_{n-1}q_n$, $r_n=0$

Demonstration

- from the last relation it follows that r_{n-1} divides r_{n-2} , from the penultimate relation it results that r_{n-1} divides r_{n-3} etc.
- it results that r_{n-1} divides both a and b (so it is a common divisor)
- to show that r_{n-1} is a common divisor we consider that d is another common divisor for a and b; from the first relation it results that d divides r_1 ; from the second it results that d divides r_2 etc.
- from the penultimate relation it results that d divides r_{n-1}
- So any other common divisor d divides r_{n-1} so r_{n-1} is a common divisor

Example – greatest common divisor. Implementation variants

Using while

```
gcd(integer a,b)
  integer d, i, r
  d ← a
  i  ← b
  r ← d MOD i
  while r!=0 do
     d ← i
     i ← r
    r ← d MOD i
  endwhile
  return i
```

Using repeat-until

```
gcd(integer a,b)
   integer d, i, r
   d ← a
   i ← b
   repeat
      r ← d MOD i
      d ← i
      i ← r
   until r==0
   return d
```

Example – greatest common divisor for a sequence of values

- Problem: to determine the *gcd* of a sequence of nonzero natural numbers

- Example:
  - gcd(12,8,10) = gcd(gcd(12,8),10) = gcd(4,10)=2

- Input data: sequence of values (a_1, a_2,..., a_n)
- Output data (result): gcd (a_1, a_2,..., a_n)

- Idea:
  - The *gcd* of the first two elements is calculated, then the *gcd* is calculated for the previous result and the new value ...
  - ... it is natural to use a subalgorithm that calculates the *gcd*

Example – greatest common divisor for a sequence of values

Algoritmn structure

```
gcdSequence(integer a[1..n])
  integer d,i
  d ← gcd(a[1],a[2])
   for i ← 3,n do
     d ← gcd(d,a[i])
   endfor
return d
```

```
gcd(integer a,b)
  integer d,i,r
  d ← a
  i  ← b
  r ← d MOD i
  while r!=0 do
     d ← i
     i ← r
    r ← d MOD i
  endwhile
return i
```

Example - the successor problem

- Consider a number consisting of 10 distinct digits. Determine the next element in the ascending sequence of natural numbers consisting of 10 distinct digits.
- Example: x= 6309487521
- Input data: one-dimensional array with 10 elements containing the digits of the number: [6,3,0,9,4,8,7,5,2,1]
- What is the next number (in ascending order) containing 10 distinct digits?
- Answer: 6309512478

Example - the successor problem

- **Step 1**. Determine the largest index i having the property that x[i-1]<x[i] (it is considered that the first digit, i.e. 6, has index 1)

  Example: x= 6309487521 i=6

- **Step 2**. Determine the index, k, of the smallest element x[k] in the subarray x[i..n] that is greater than x[i-1]

  Example: x=6309487521 k=8

- **Step 3**. Interchange x[k] with x[i-1]

  Example: x=6309587421 (this value is greater than the previous one)

- **Step 4**. Sort x[i..n] in ascending order (to obtain the smallest number that satisfies the requirements)

  Example: x=6309512478 (it is sufficient to reverse the order of the elements in x[i..n])

Example - the successor problem

Subproblems / subalgorithms

- Identify: Identify the position i of the rightmost element x[i], which is greater than its left neighbor (x[i-1])
  - Input: x[1..n]
  - Output: i

- Minimum: Determine the index of the smallest element in the subarray x[i..n] that is greater than x[i-1]
  - Input: x[i-1..n]
  - Output: k

- Reverse: Reverse the order of the elements in x[i..n]
  - Input: x[i..n]
  - Output: x[i..n]

Example - the successor problem

Algorithm structure

```
Successor(integer x[1..n])
  integer i, k
  i ← Identify(x[1..n])
  if i==1
     then write "There is no successor!"
  else
     k ← Minimum(x[i-1..n])
     x[i-1] <->  x[k]
     x[i..n] ← Reverse(x[i..n])
     write x[1..n]
  endif
```

Remark

In general, exchanging the values of two variables requires 3 assignments and the use of an auxiliary variable (just as changing the liquid content of two glasses requires the use of another glass)

$a \leftrightarrow b$

is equivalent to

aux = a

a = b

b = aux

Example - the successor problem

```
identify(integer x[1..n])
  integer i
  i ← n
   while (i>1) and (x[i]<x[i-1]) do
        i ← i-1
   endwhile
  return i
```

```
minimum(integer x[i-1..n])
  integer j
  k ← i
  for j ← i+1,n do
     if x[j]<x[k] and x[j]>x[i-1] then
         k ← j
     endif
  endfor
  return k
```

```
reverse (integer x[left..right])
  integer i, j
  i ← left
  j ← right
  while i<j do
     x[i] <-> x[j]
     i ← i+1
      j ← j-1
  endwhile
  return x[left..right]
```

Example - the successor problem. Python code

```
def identify(x):
  n=len(x)
  i=n-1
   while (i>0)and(x[i-1]>x[i]):
       i=i-1
   return i
def minimum(x,i):
    n=len(x)
    k=i
   for j in range(i+1,n):
    if (x[j]<x[k])and (x[j]>x[i-1]):
         k=j
  return k
```

```
def reverse(x,left,right):
   i=left
   j=right
   while i<j:
     x[i],x[j]=x[j],x[i]
      i=i+1
      j=j-1
   return x
```

- Remark: In Python, the interchange of two variables a and b can be done by *a,b=b,a*

- Functions call

```
x=[6,3,0,9,4,8,7,5,2,1]
print ("The initial sequence:", x)
i=identify(x)
print ("i=", i)
k=minimum(x, i)
print ("k=", k)
x[i-1],x[k]=x[k],x[i-1]
print ("The sequence after switch:", x)
x=reverse(x, i, len(x)-1)
print ("The sequence after reversing:", x)
```

Summary

- Problems are decomposed into subproblems to which are associated subalgorithms
- A subalgorithm is characterized by:
  - Name
  - Parameters
  - Return values
  - Local variables
  - Processing
- Calling a subalgorithm:
  - Parameters are replaced with concrete values specified at the call
  - Processing in the algorithm is executed on concrete values and the results are returned
  - It is possible that processing is performed directly on global variables

Next course

- Algorithm Efficiency Analysis

- Execution Time Estimation
  - Best Case
  - Worst Case
  - Average Case

Q&A