EECE1012

Net-Centric Introduction to Computing

Computational Thinking

Amirhossein Chinaei

Winter 2019

Office Hours: T 9:30-10:30 W 11:30-12:30 LAS3048

ahchinaei@cse.yorku.ca

overview

- **computational thinking**
  - the thought process involved in expressing a solution in such a way that a computer can effectively execute it
- **algorithm**
  - an unambiguous specification of how to solve a problem in finite steps and finite memory
- **flowchart**
  - visualization of an algorithm
- **other design tools**
  - uml, pseudocode, etc.

software development life-cycle

- **requirements**
  - understand/analyze what the problem/project is
- **design**
  - find/define a solution and present it in a schematic way
- **implementation**
  - code it in a programming language
- **verification**
  - verify the solution for test cases and fix possible flaws

our focus

learn more about: waterfall, agile, spiral, v-model, etc.

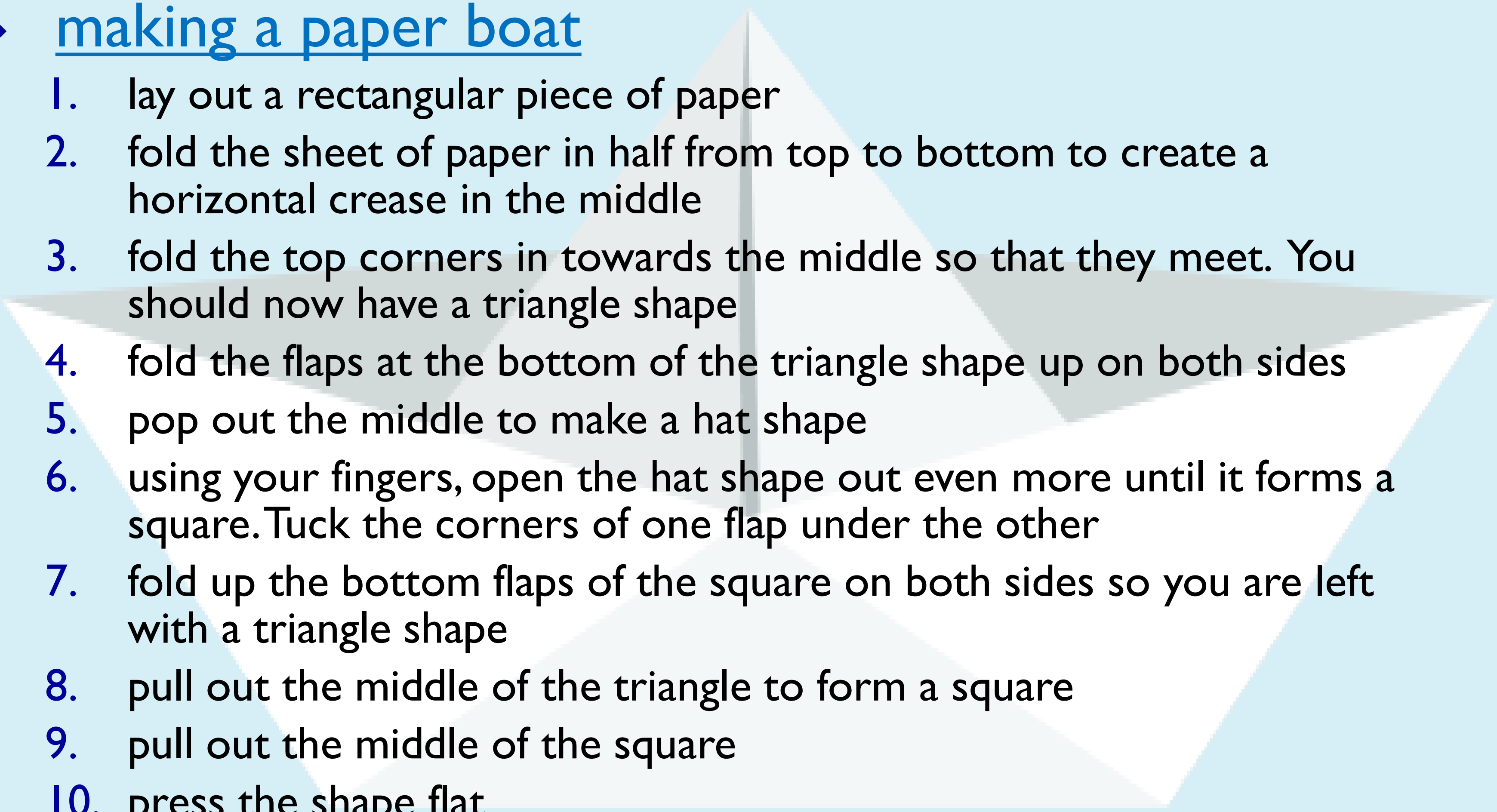
algorithm

- an unambiguous specification of how to solve a problem in finite steps and finite memory
- it’s like a recipe for problem solving
- it’s the *design* phase of writing a program
  - **you don't want to miss this phase**
- history
  - the concept existed for centuries
    - example: Euclid's method of computing the greatest common divisor (gcd)
  - the term derives from the 9th Century mathematician Muḥammad ibn Mūsā al'Khwārizmī, latinized 'Algoritmi'

recipe example

- **making a bagel sandwich**
  1. place bagel in toaster and toast as desired
  2. over medium heat, fry egg in frying pan
  3. turn off heat but, leave pan on burner
  4. place meat on top of egg and then the cheese on top of the meat
  5. cover with a lid and let the heat melt the cheese
  6. place egg, meat, and cheese pile on one half of bagel and top with the other half
  7. enjoy!

example 2, try it, what do you think?



❖ making a paper boat

1. lay out a rectangular piece of paper
2. fold the sheet of paper in half from top to bottom to create a horizontal crease in the middle
3. fold the top corners in towards the middle so that they meet. You should now have a triangle shape
4. fold the flaps at the bottom of the triangle shape up on both sides
5. pop out the middle to make a hat shape
6. using your fingers, open the hat shape out even more until it forms a square. Tuck the corners of one flap under the other
7. fold up the bottom flaps of the square on both sides so you are left with a triangle shape
8. pull out the middle of the triangle to form a square
9. pull out the middle of the square
10. press the shape flat
11. open out from the bottom to assemble your boat shape

algorithm example 3

- **problem:** write a program that receives two numbers and calculate their sum.
- **solution:** instead of rushing to implementation, the idea is to provide an **algorithm** first:

Steps

1. start
2. receive a number from the end-user
3. receive another number from the end-user
4. calculate sum of the two numbers
5. show the sum to the end-user
6. stop

algorithm example 3 revisited

- **problem:** write a program that receives two numbers and calculate their sum.
- we try to make our algorithm a bit more specific and less ambiguous

Steps

1. start
2. receive a number from the end-user and store it as *a*
3. receive a number from the end-user and store it as *b*
4. calculate sum of *a* and *b*
5. show the sum to the end-user
6. stop

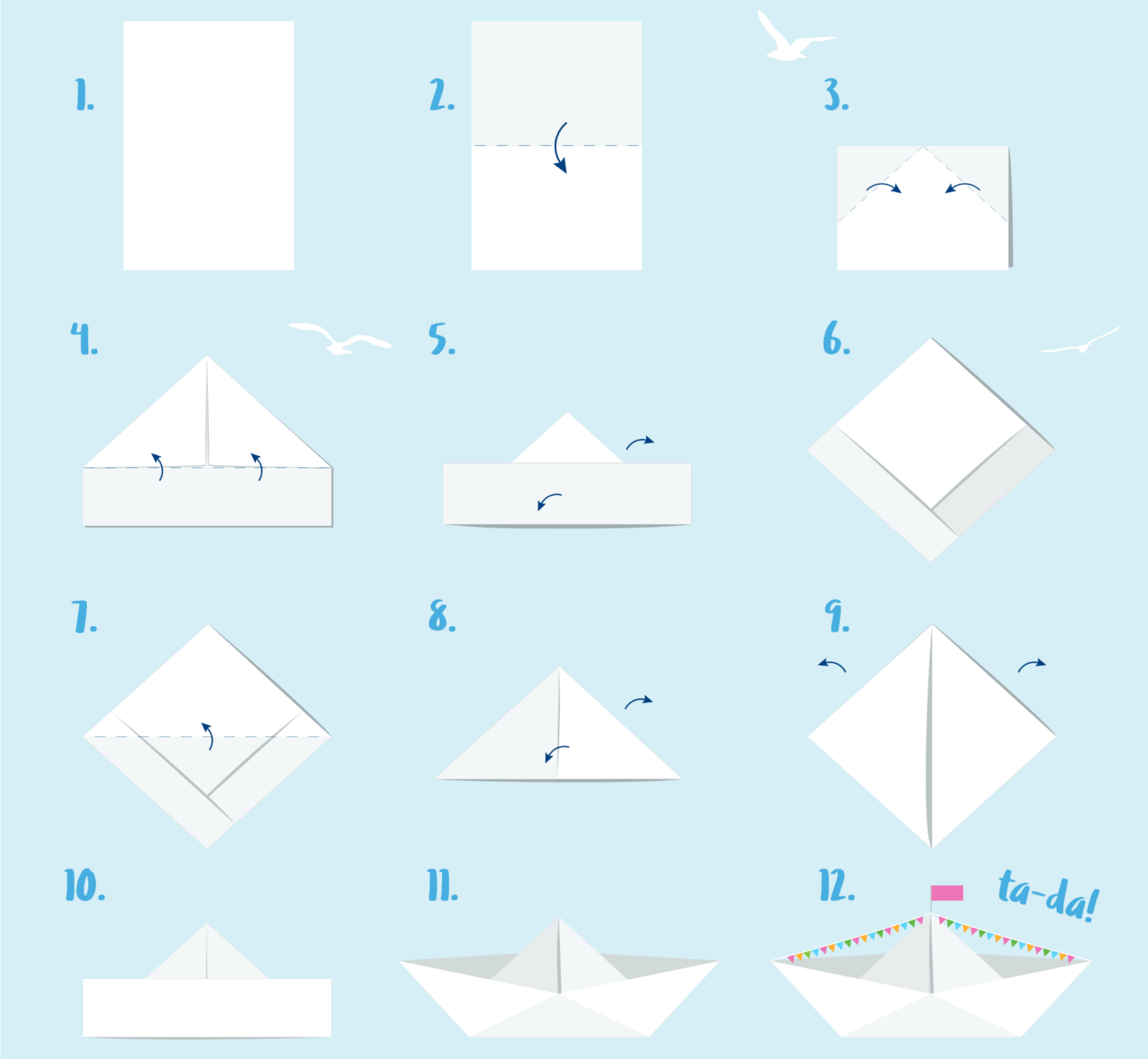
algorithm example 4

- **problem:** write a program that calculates sum of numbers 10, 11, 12, …, 30.
- **solution:** instead of rushing to implementation, the idea is to provide an **algorithm** first:

1. start
2. store 10 in a memory space and call it *a*
3. store 0 in a memory space and call it *s*
4. add *a* to *s* and store it in *s*
5. increment the value stored in *a*
6. if *a*<31 go to step 4; otherwise, go to the next step
7. show *s* to the end-user
8. stop

flowchart

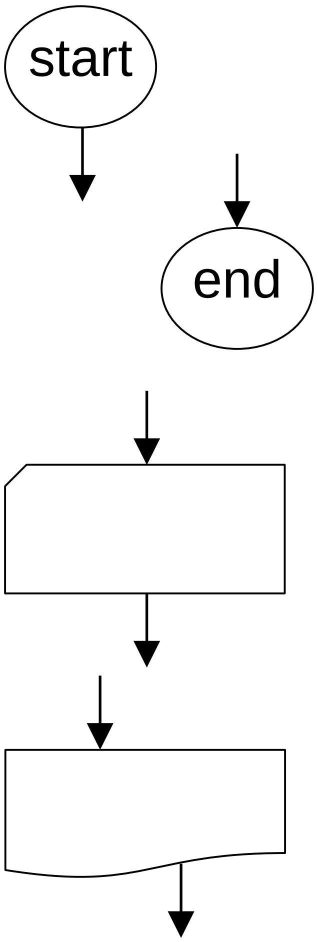
- visual presentation of algorithms
- like a blueprint outlines an architecture/design details visually, a flowchart visualizes an algorithm



a picture is worth a thousand words

start, end, input, output

early computers used punched cards for data input, and papers for data output



start the process

end the process

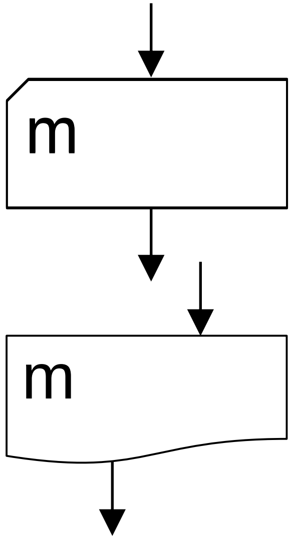
input data

and store

output data

typically the stored data

examples:



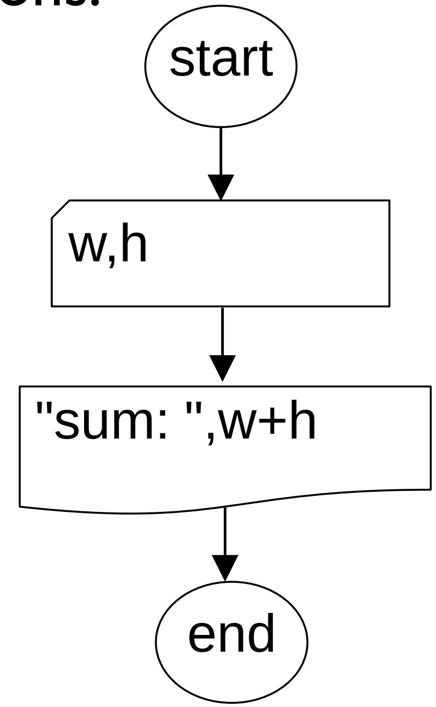
input data &
store it as ***m***

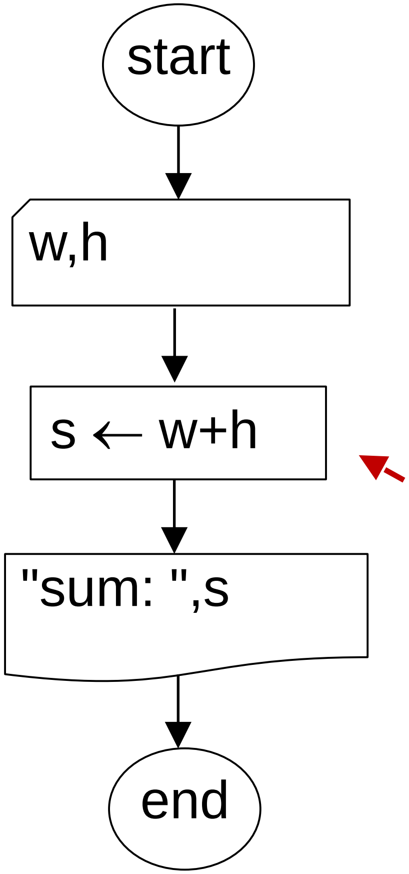
output ***m***

example 5 ***visual** approach to example 3*

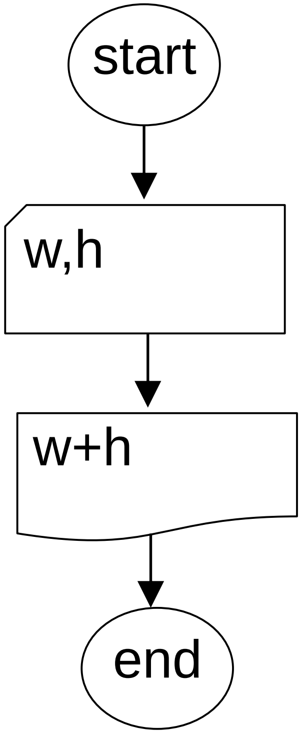
- receive two numbers and show their sum.

- other solutions:

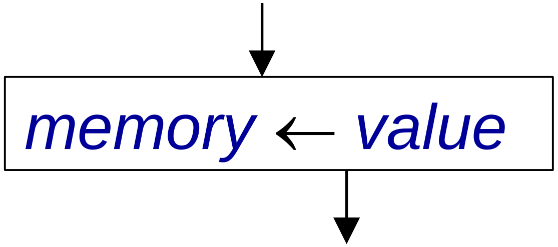




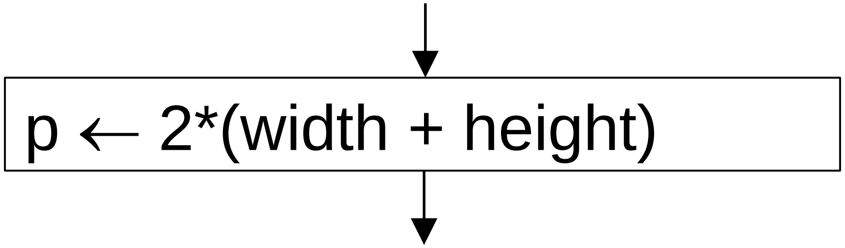
assignment box



assignment


memory ← *value*

- it's used to assign a *value* to a *memory* space
- the *memory* space that we want to assign the *value* to it, should be on the left side
- the *value* normally is an *expression*
- **example:**


p ← 2*(width + height)

  - the value stored in memory space called *width* is added to the value stored in *height*; the result multiply by 2 generates a new *value* that is stored in *p*
  - use meaningful names for memory spaces
  - p in the example above is not a good name, why?
  - also, it's highly ***discouraged*** to use = instead of ←

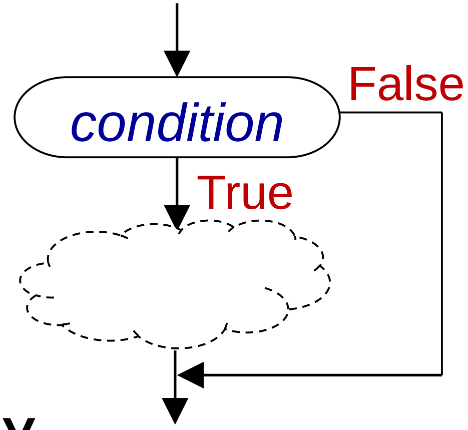
example 6

- **problem:** write a program that receives numerical coefficients of a quadratic equation and calculates its roots. (assume coefficients are good enough such that a solution exists.)
- **solution:**
  - first, let's understand/analyze what the project is
  - we may need to review/learn about quadratic equations
    - if the equation is in form of $ax^2 + bx + c = 0$, the roots are calculated as follows:

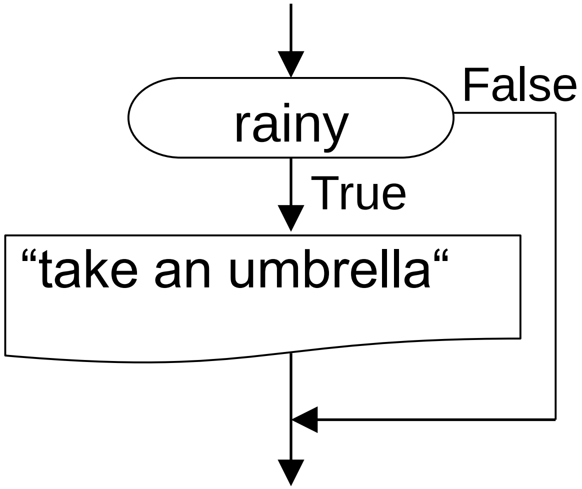
$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

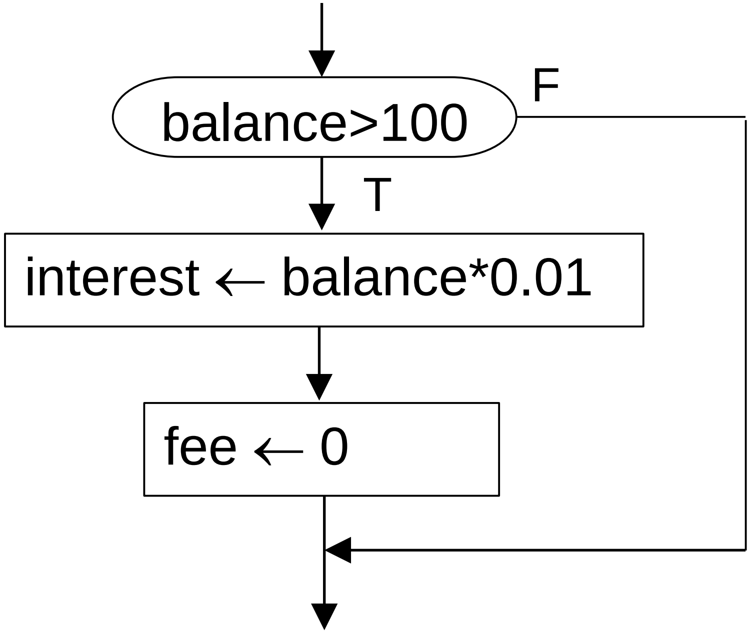
  - then, draw a flowchart for that

branching (if-then)



- it’s used to branch the flow of steps based on certain *boolean conditions*
- the cloud part ☁ is actually replaced by any combination of flowchart constructs
- **examples:**





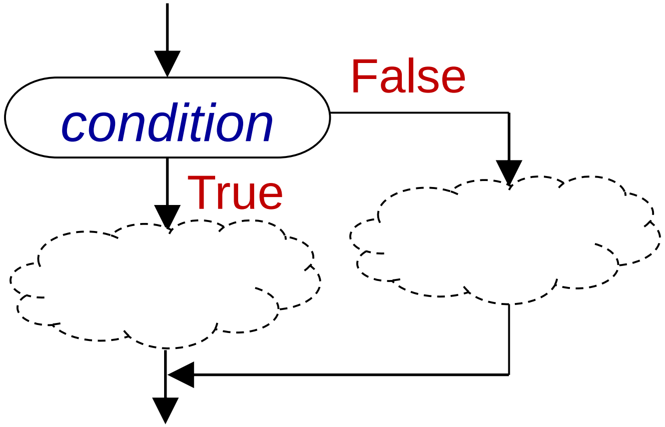
example 7

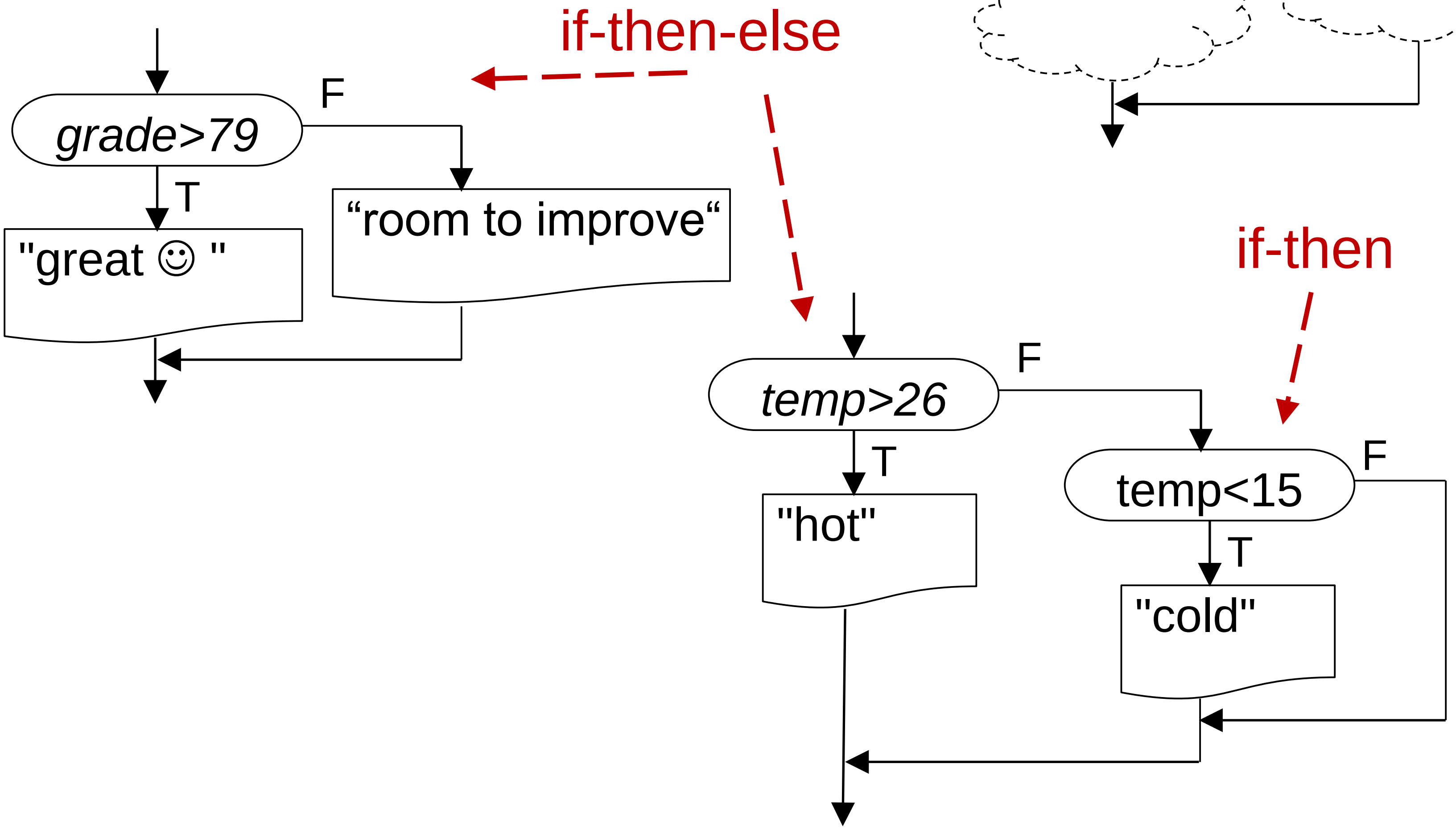
- **problem:** write a program that receives numerical coefficients of a quadratic equation and calculates its roots if a solution exists.
- **solution:**

  …

branching (if-then-else)

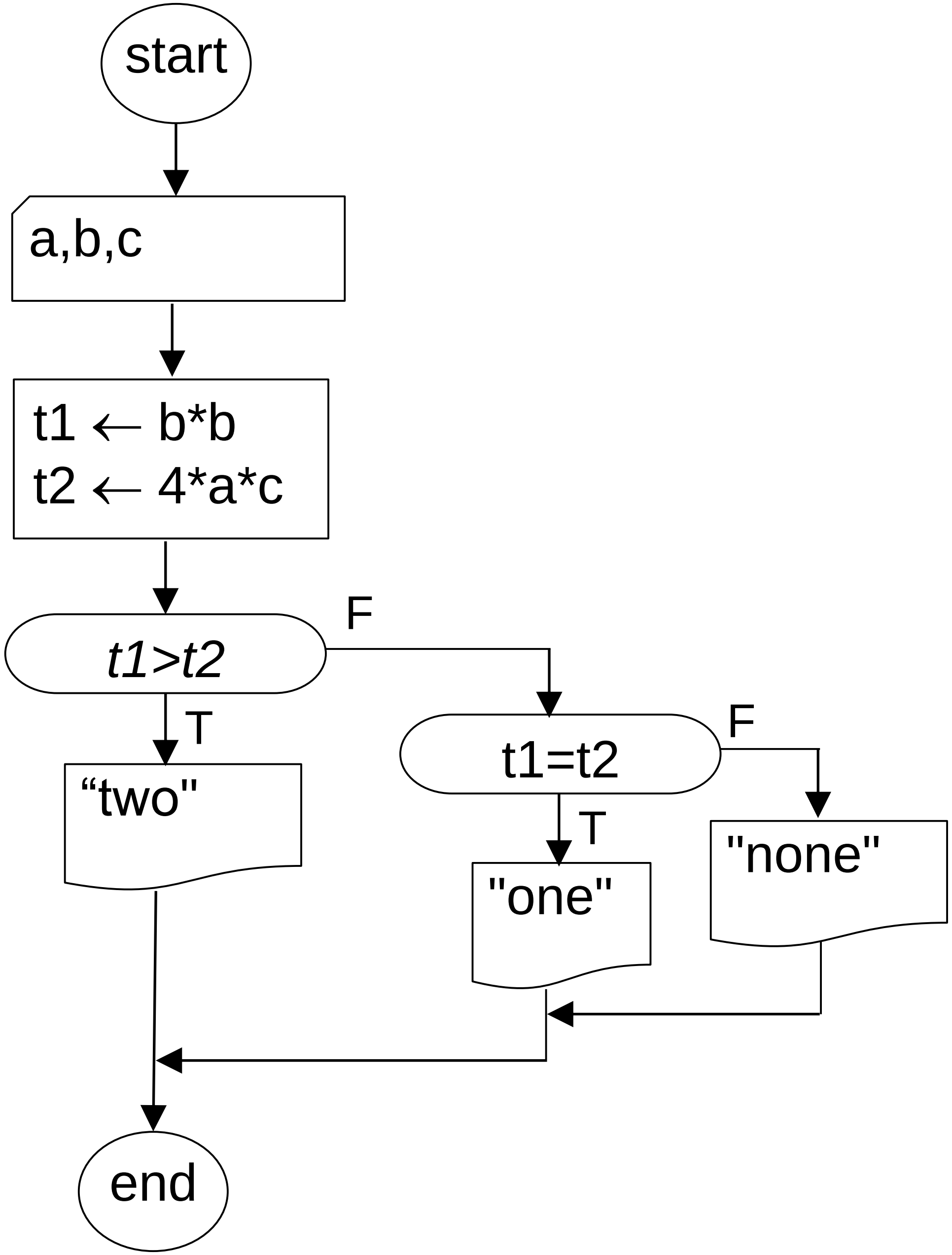
- **examples:**





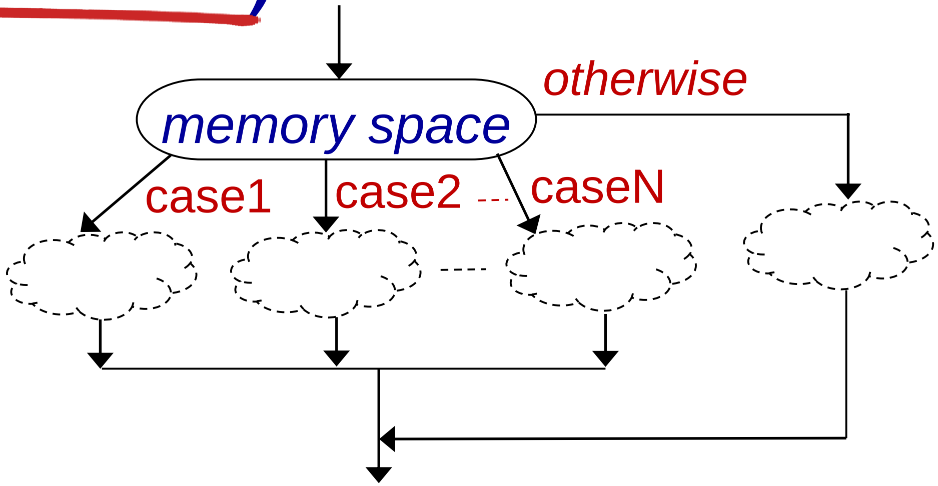
example 8

- draw a flowchart for a program to receive numerical coefficients of a quadratic equation and determine if it has two distinct roots, one root, or no root.

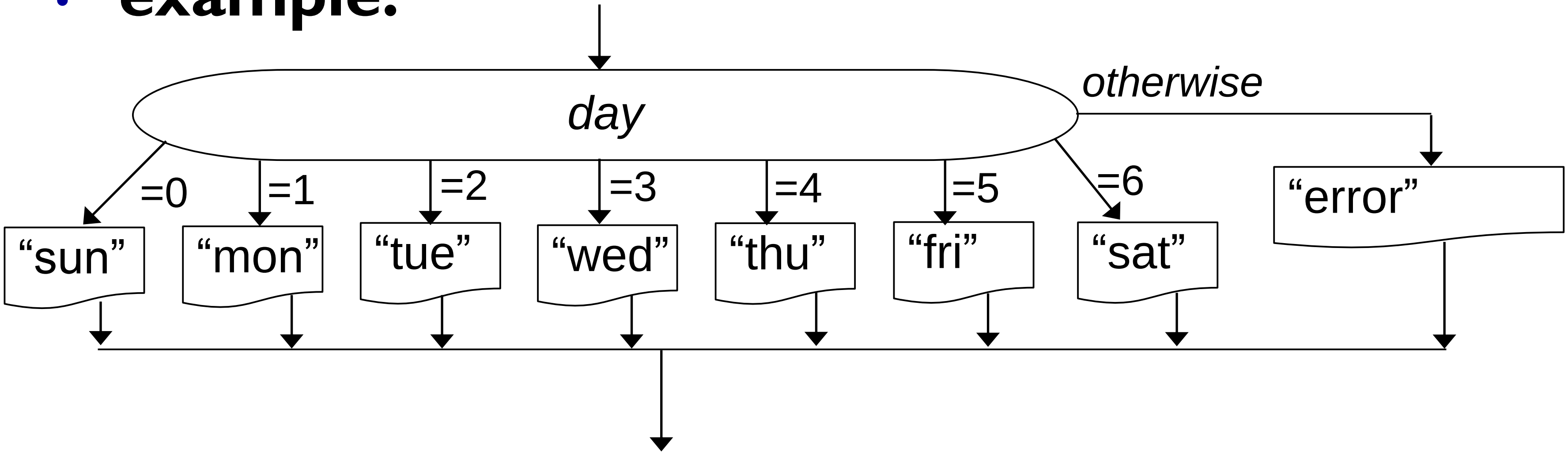


many branches (cases)

- deciding based on different values of a memory space or expression



- **example:**



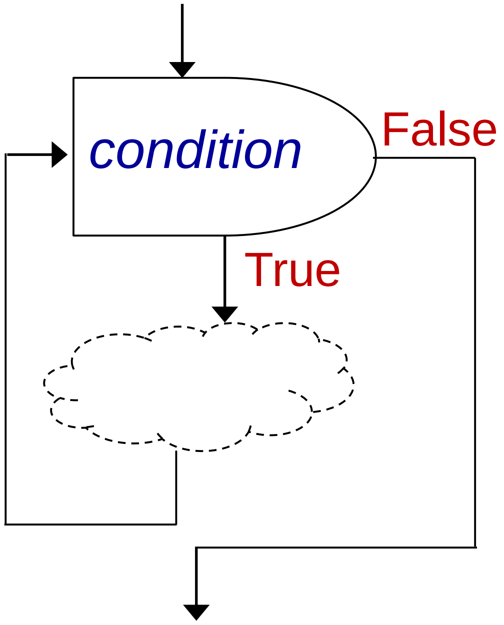
example 9

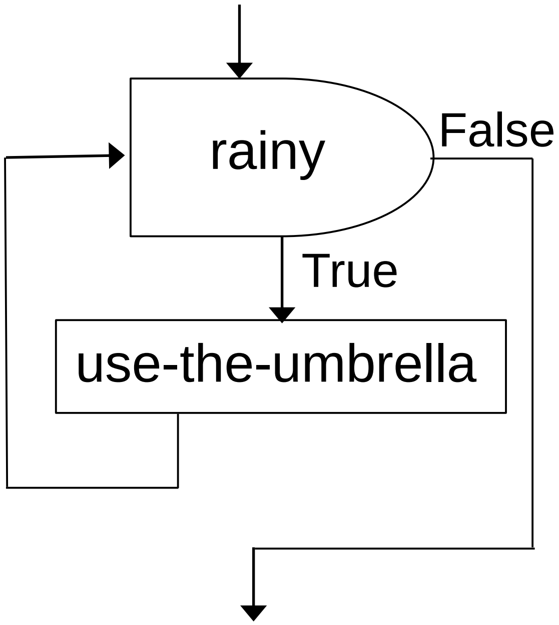
- draw a flowchart for a program to receive a grade and map it to a letter based on York standard.

…

iteration (while-do loop)

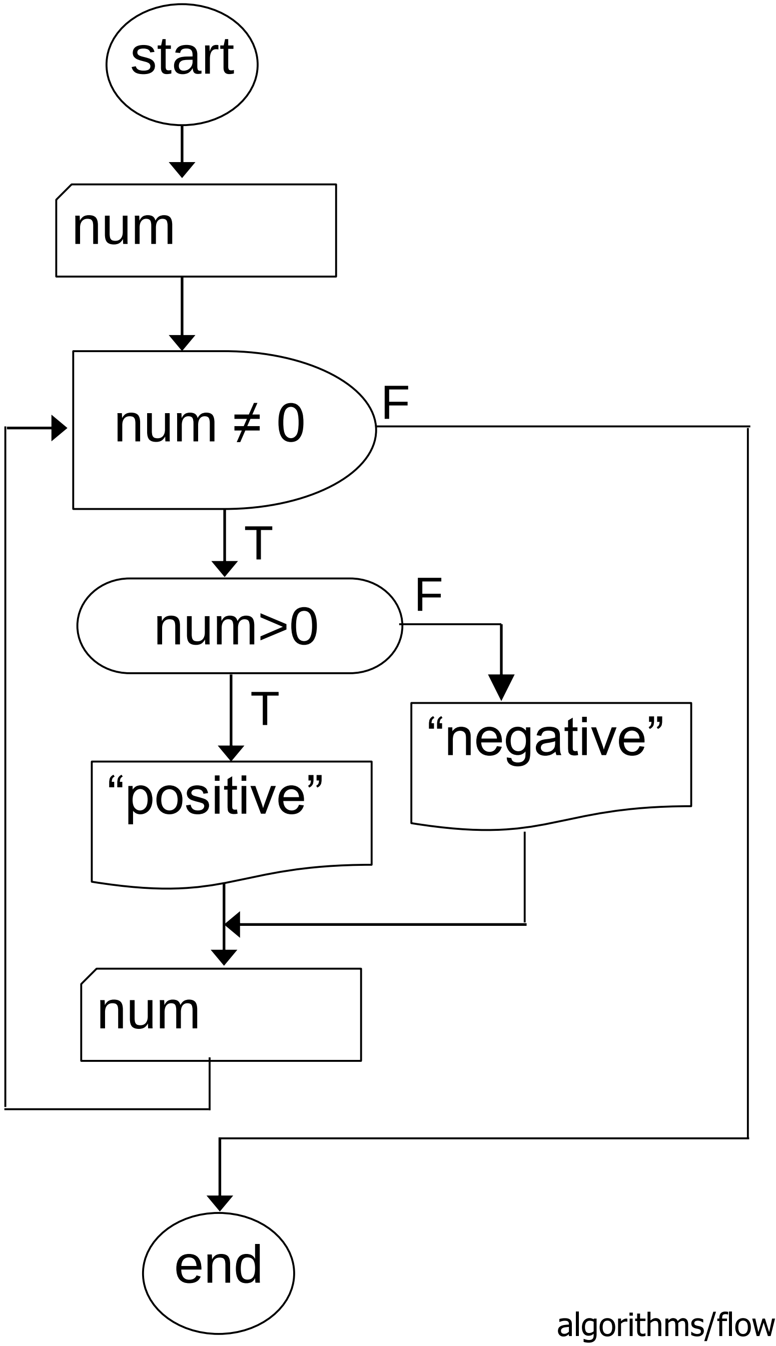
- as long as the *boolean* *condition* holds, the steps in the cloud part are iterated
- normally it's used for **non deterministic** iterations
- **example:**





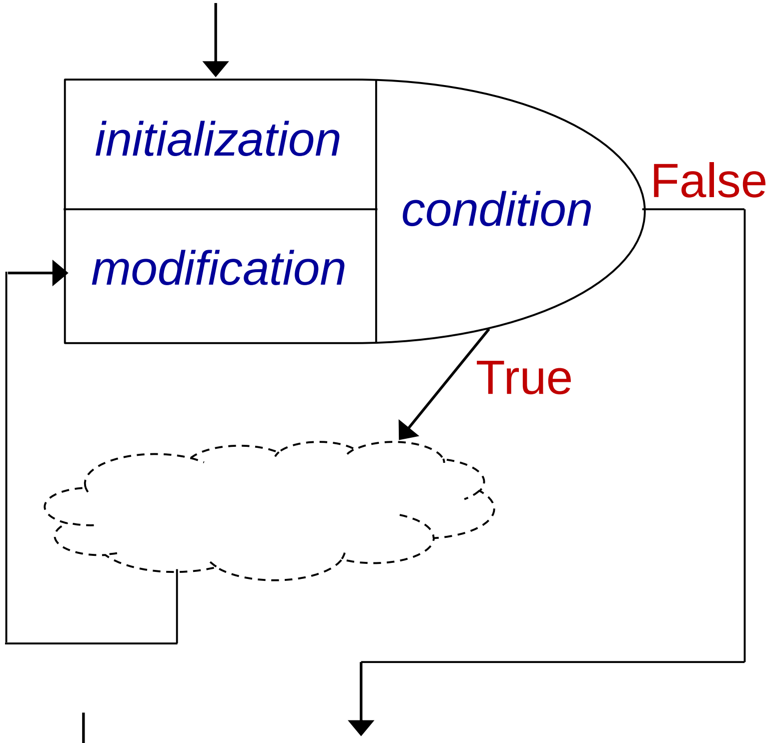
example 10

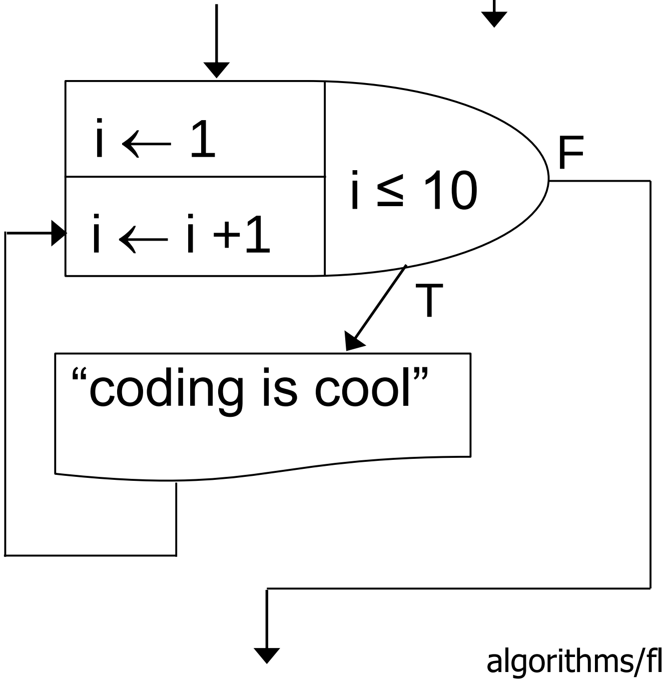
draw a flowchart to keep receiving numbers from the end-user and determine if they are positive or negative, until a zero is received.



iteration (for loop)

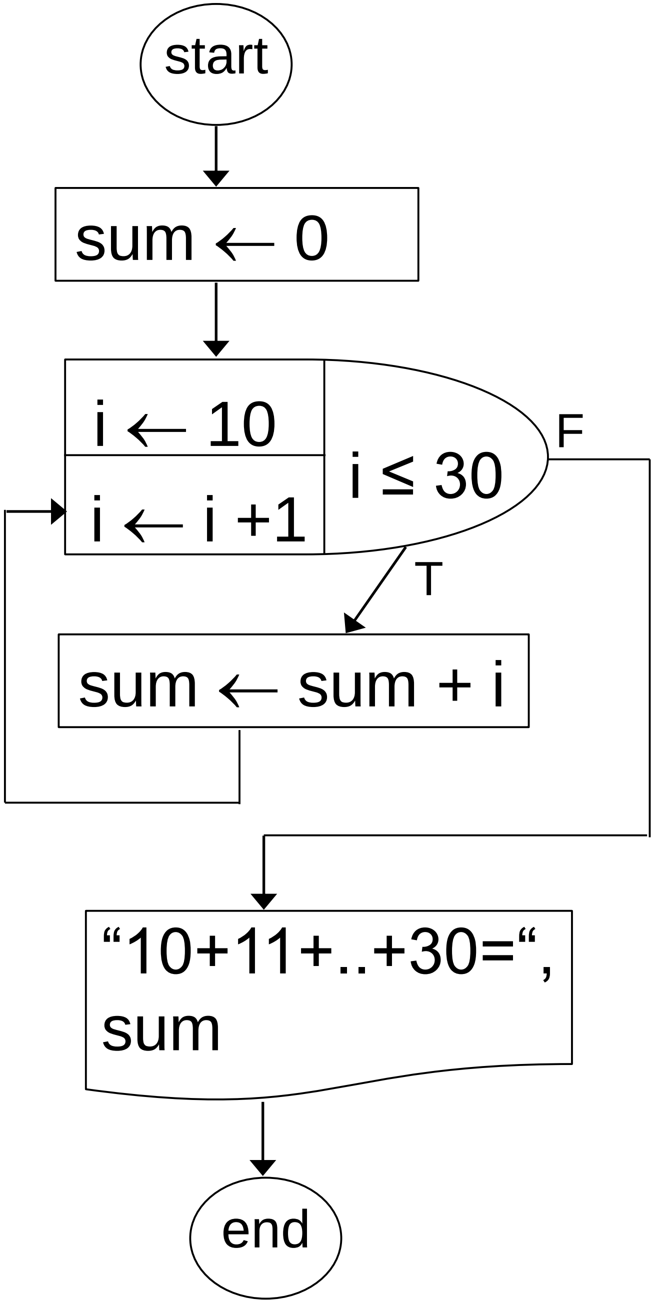
- as long as the *boolean condition* holds, the steps in the cloud part are iterated
- normally it's used for **deterministic** iterations
- *initialize* a memory space
- specify how you want that memory space be *modified* after each iteration
- **example:**





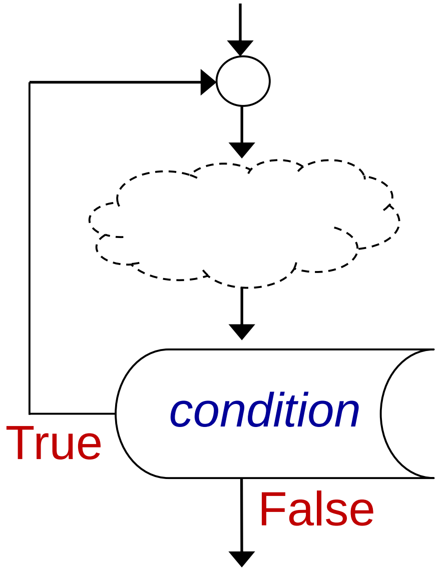
example 11

draw a flowchart to compute sum of numbers between 10 and 30, inclusively.



iteration (do-while loop)

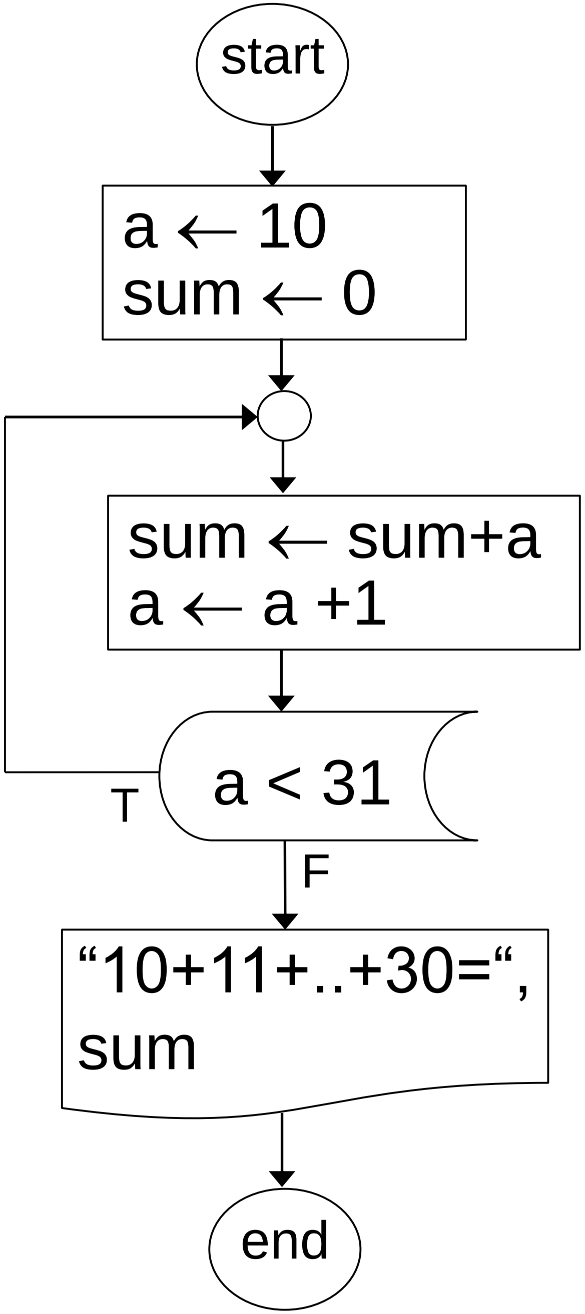
- the steps in the cloud part are iterated as long as the *boolean condition* holds
- it's similar to the while-do loop, however the cloud part will be iterated at least once
- similar to while-do, it's used for **non deterministic** iterations



- **exercise:** draw a flowchart to keep receiving numbers and adds the positive numbers and negative numbers separately, until a zero is received. compare the structure of your flowchart with that of example 10.

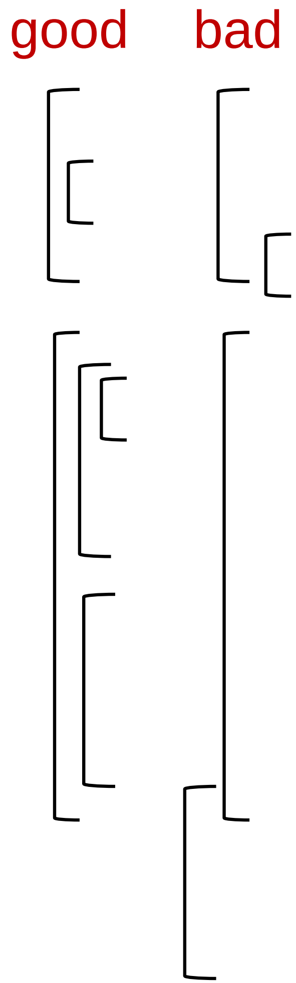
example 12

draw a flowchart for the algorithm presented in example 4: *sum of 10, 11,..., 30*



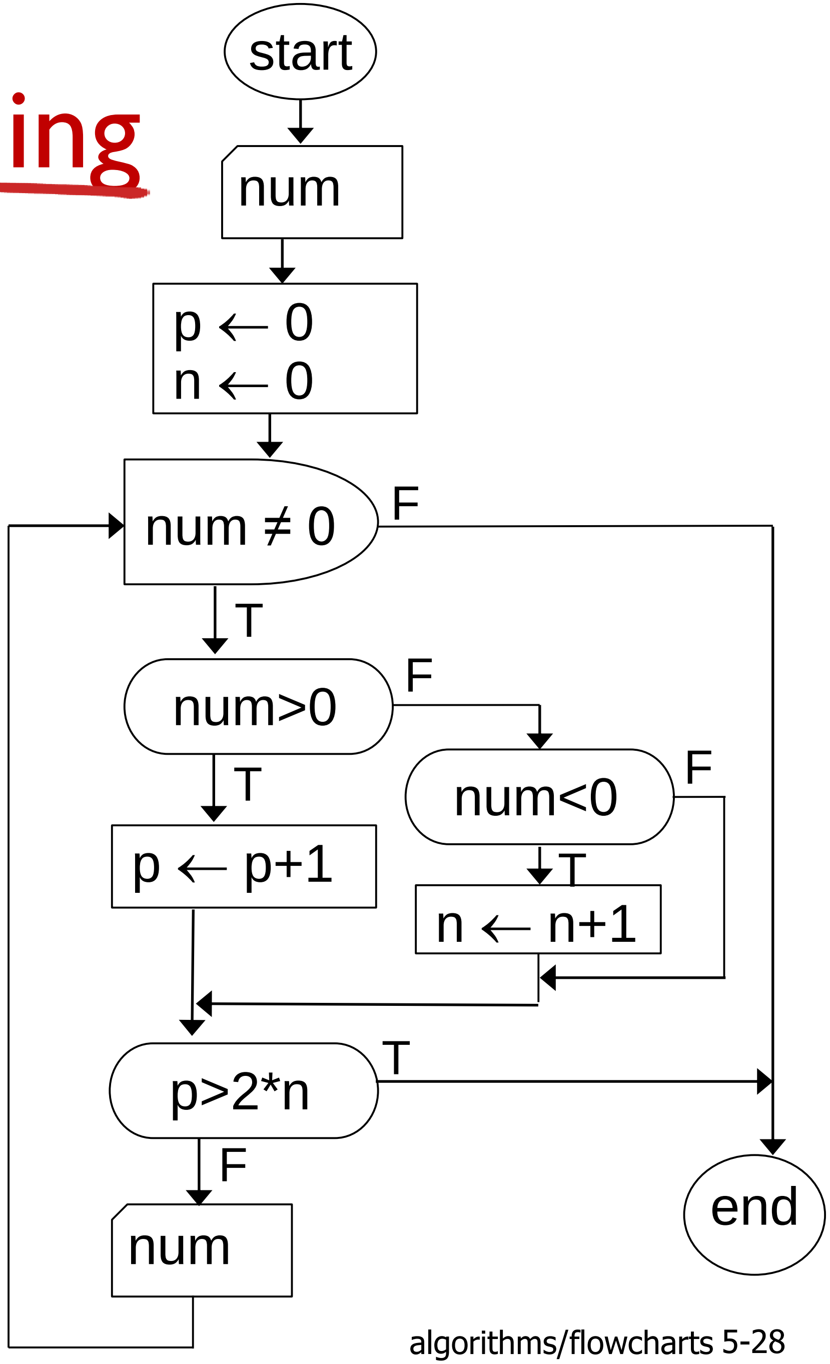
proper nesting

- **common constructs**
  - the flowchart constructs introduced so far are supported in most well-known languages
- **nesting**
  - you can use a construct inside another construct as long as they are **properly nested**
- **bad practice**
  - jumping into a construct or jumping outside a construct is considered a **bad practice**
  - in general, jumping is considered a bad practice



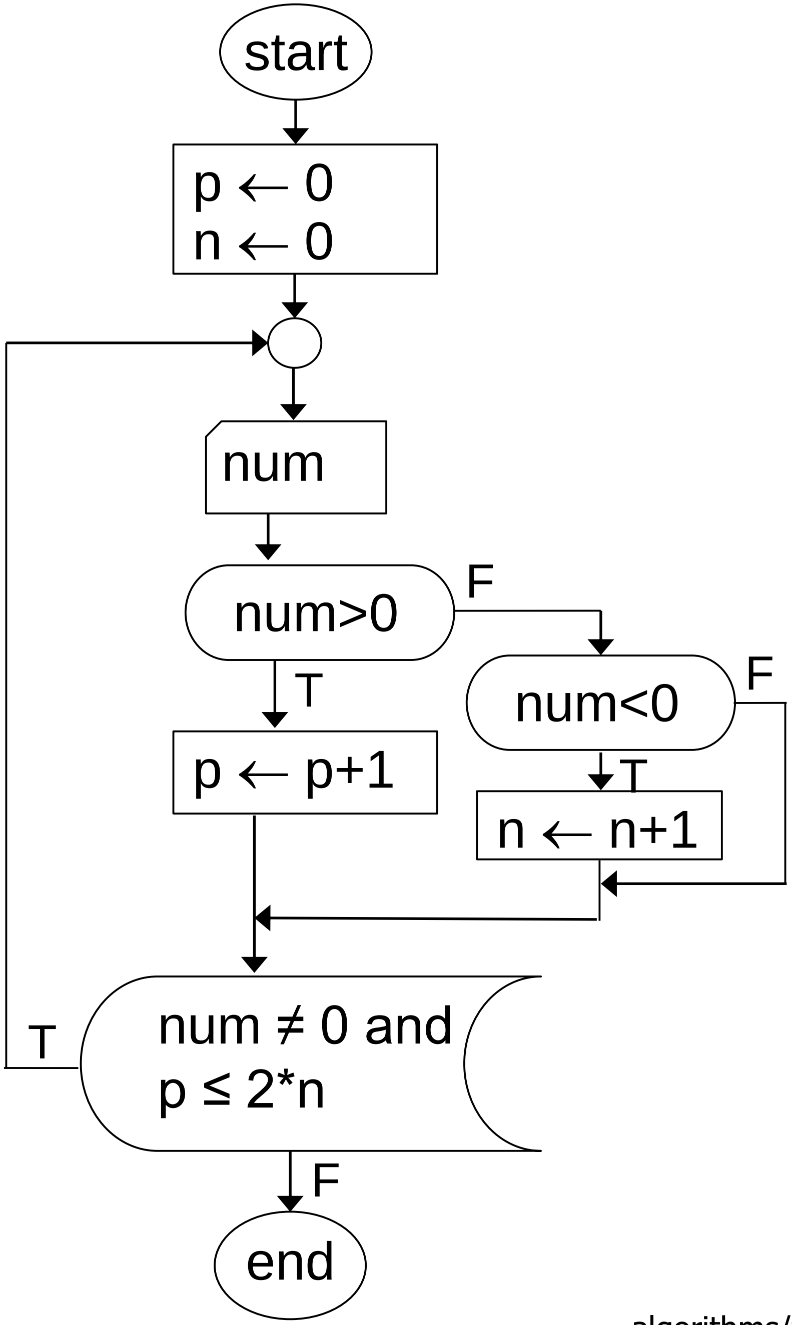
example of a bad thing

draw a flowchart to receive numbers and count positive and negative ones, until a zero is received. Also, at any time the number of positives gets more than twice the number of negatives, the program should stop.



example 14

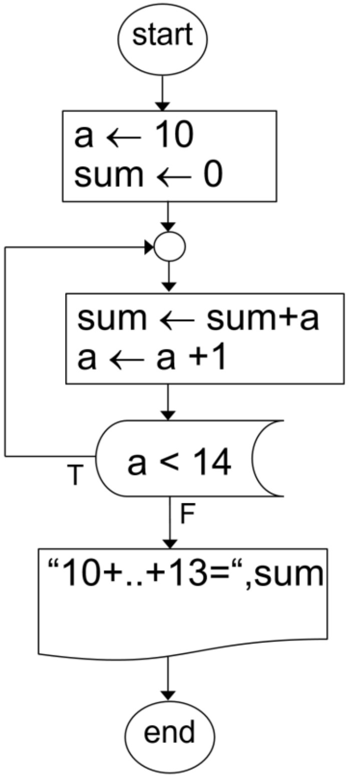
- good thing
- proper nesting



verification

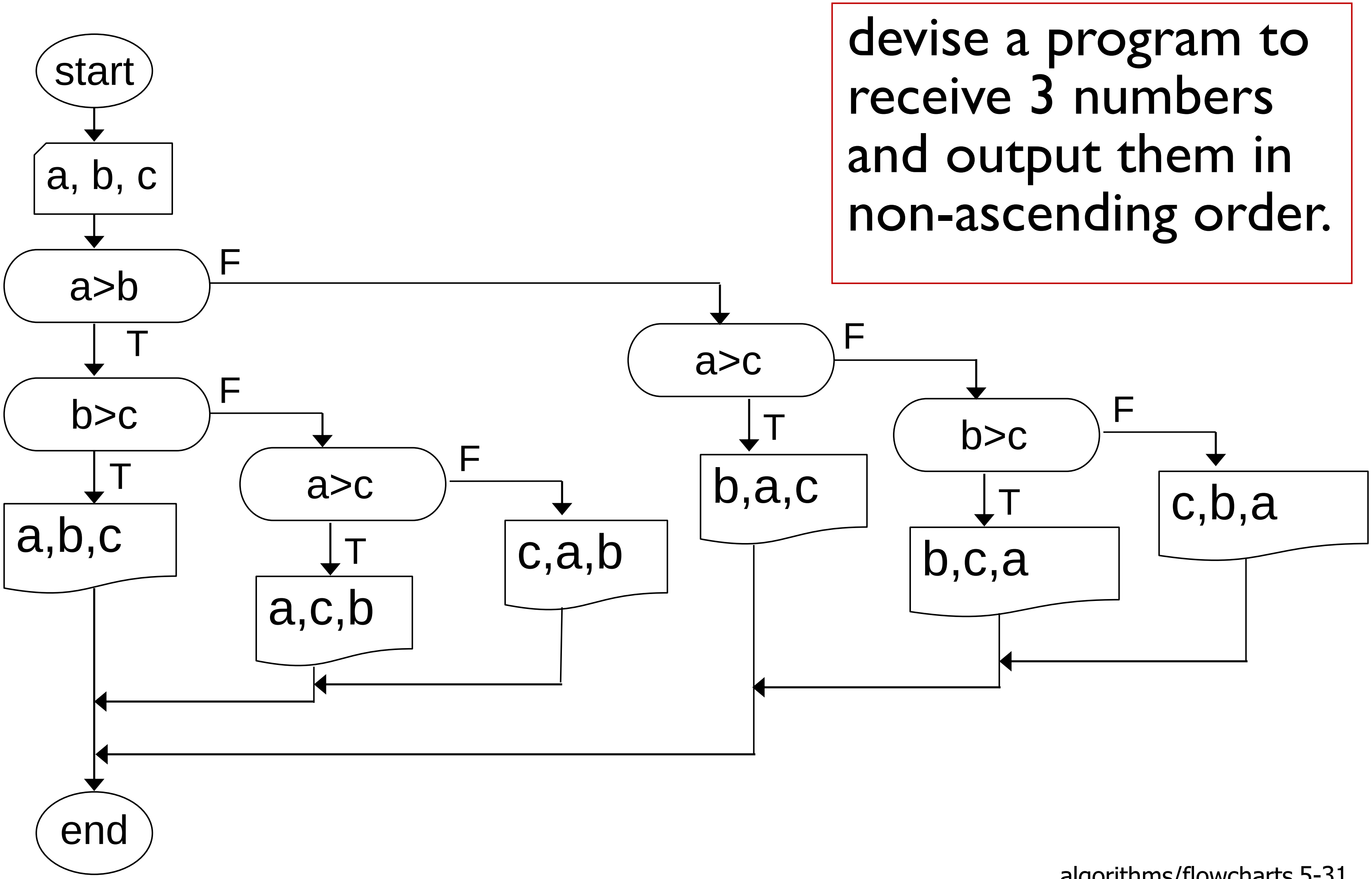
- **design tip**
  - it’s also a bad practice to develop algorithms/flowcharts and rush into implementation without testing them first
- **trace**
  - to test and debug our solutions
- let’s trace example 12, for sum of 10 to 13

a	sum	output
10		
	0	
	10	
11		
	21	
12		
	33	
13		
	46	
14		
		46



advanced example I

devise a program to receive 3 numbers and output them in non-ascending order.



trace it

let's trace it when the input is 9, 6, 7

a	b	c	output
9	6	7	
			9 7 6

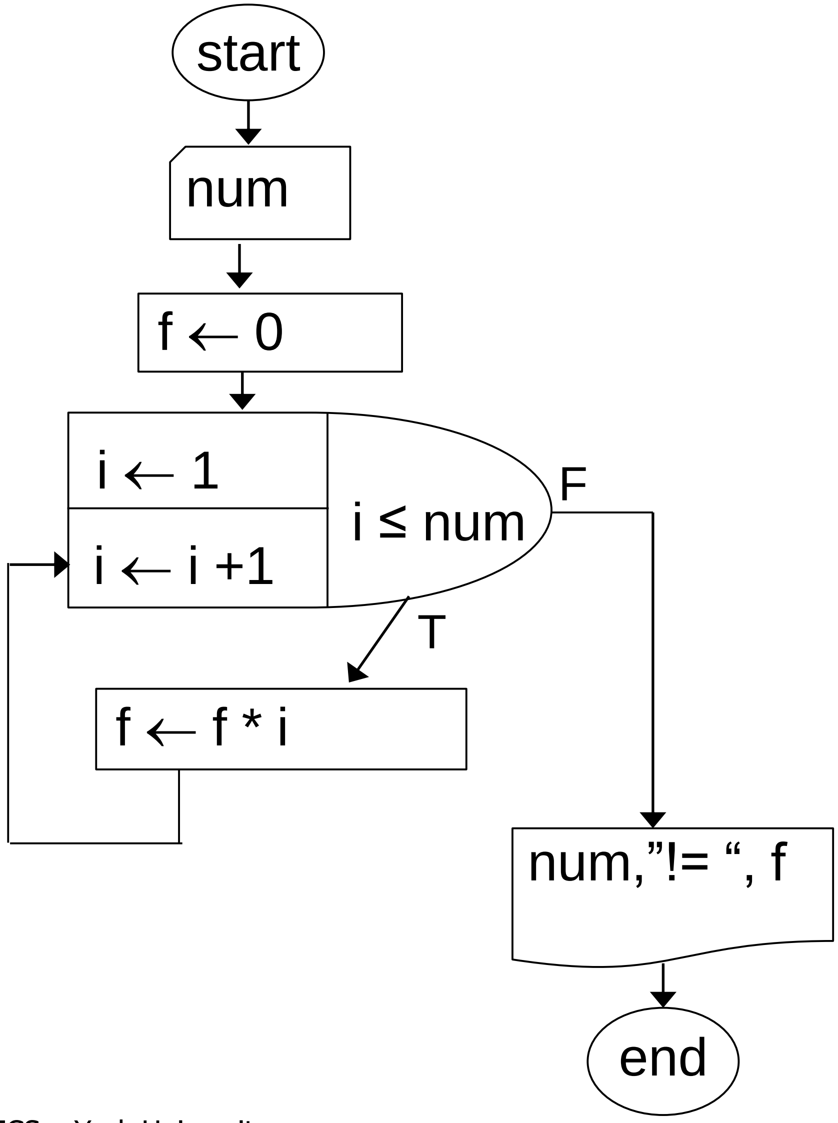
let's trace it when the input is 5, 8, 5

a	b	c	output
5	8	5	
			8 5 5

let's trace it when the input is 6, 7, 9

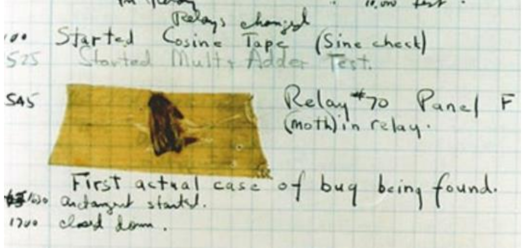
a	b	c	output
6	7	9	
			9 7 6

advanced example 2



devise a program to receive a positive whole number, *n*, and compute *n!*

trace it



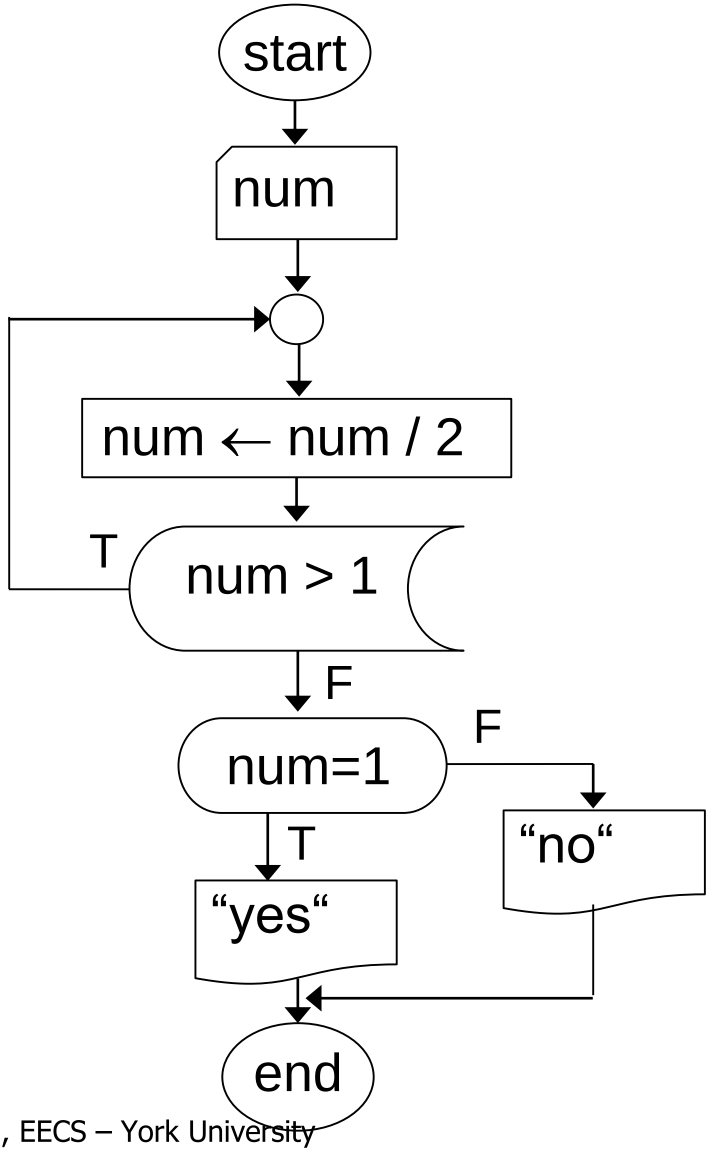
Relays changed
100 Started Cosine Tape (Sine check)
525 Started Mult + Adder Test.
545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 arctangent started.
1700 closed down.

- let’s trace our solution for when the input is 3

num	f	i	output
3			
	0		
		1	
	0		
		2	
	0		
		3	
	0		
		4	
			0

- we know 3! is 6. but, our solution outputs 0. where is the bug?

advanced example 3



devise a program to receive a number greater than 1 and determine if it is a power of 2 or not

trace it

- let’s trace our solution for when the input is 8

num	output
8	
4	
2	
1	
	yes

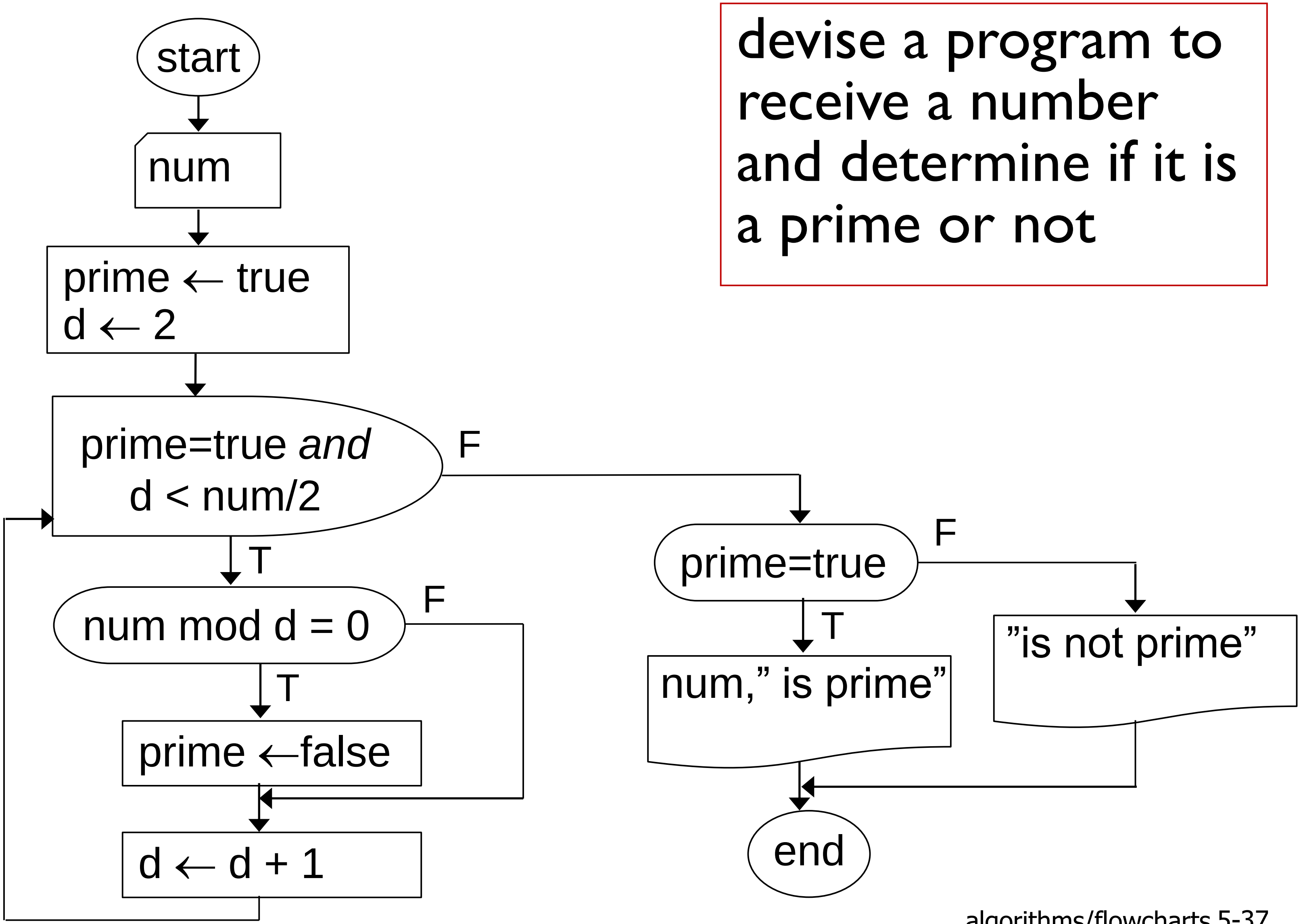
- let’s do another trace, for when the input is 18

num	output
18	
9	
4.5	
2.25	
1.125	
0.5625	
	no

- **exercises:** provide a solution using the “while-do” symbol; and, one with the “for” symbol.
- let’s modify the question a bit: devise a program to receive a number & determine if it is a power of 2 or not
- now modify and compare the 3 solutions

advanced example 4

devise a program to receive a number and determine if it is a prime or not



trace it

- let’s trace our solution for when the input is 7

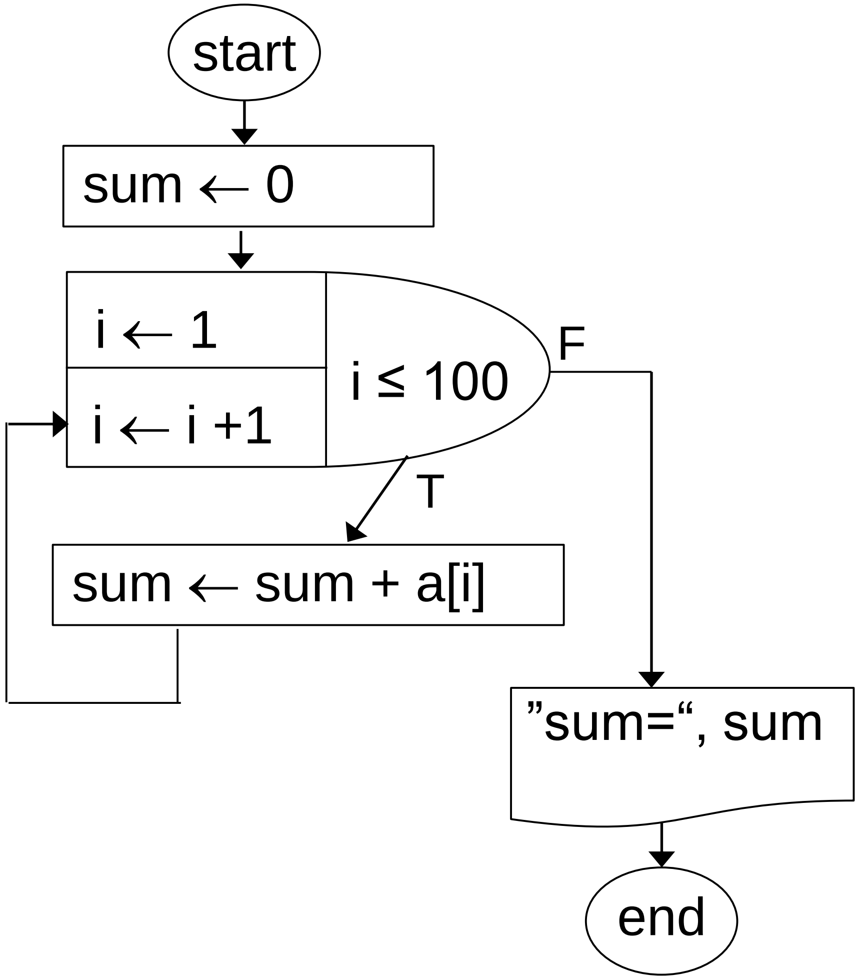
num	prime	d	output
7			
	true		
		2	
		3	
		4	
			7 is prime

- let’s do another trace, for when the input is 8

num	prime	d	output
8			
	true		
		2	
	false		
		3	
			is not prime

- **exercises:** trace the algorithm for when the number is 1 (or 0). Note that 1 (or 0) is not a prime number.
- when you find a bug, debug it.

advanced example 5

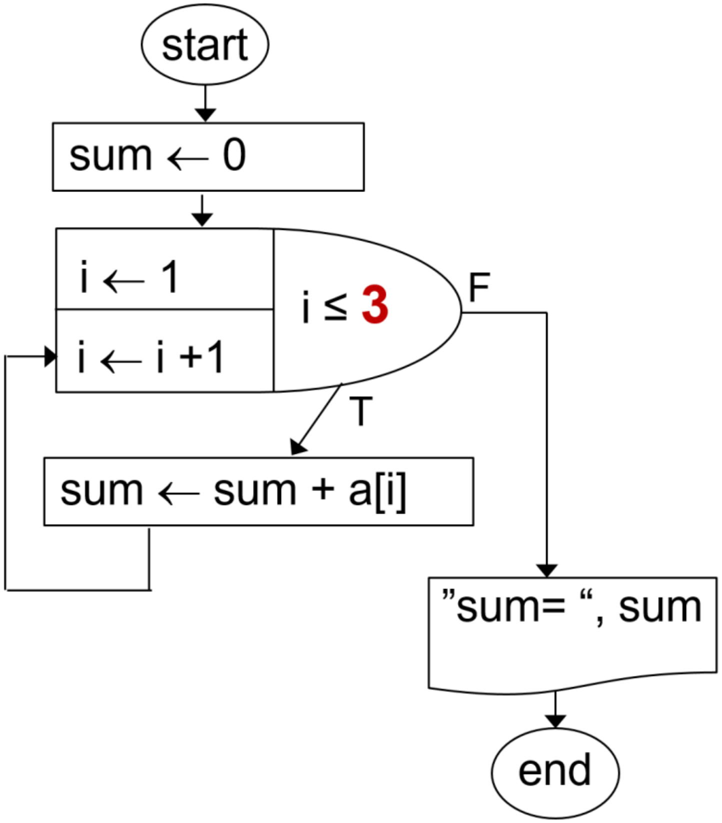


devise a program to output sum of values stored in an array of 100 elements

trace it

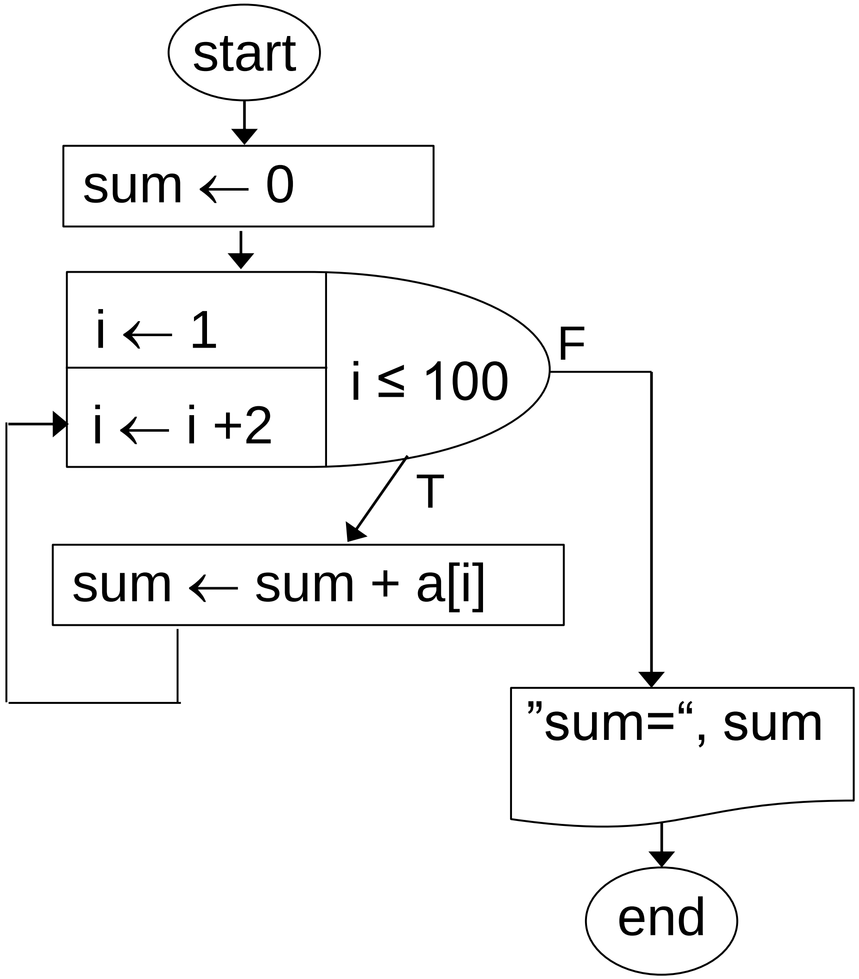
to trace our solution, we do it for an array of a smaller size. let's assume size of the array is **3**, and indexing starts from 1. also, assume the values stored in the array are 6, 3, and 12

a[1]	a[2]	a[3]	sum	i	output
6	**3**	**12**			
			0		
				1	
			6		
				2	
			9		
				3	
			21		
				4	
					sum=21



exercise: assume indexing starts from 0, modify the flowchart accordingly

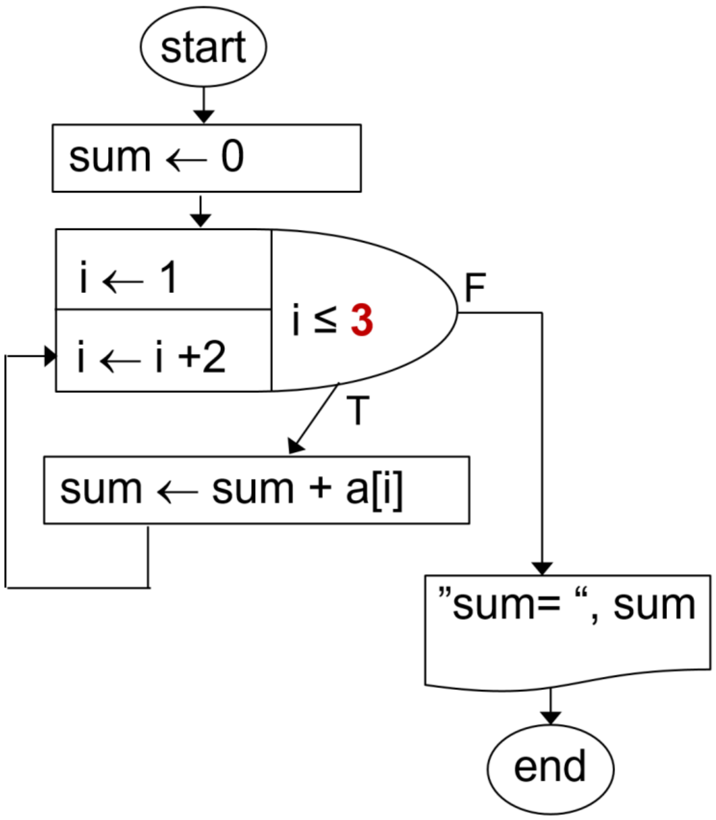
advanced example 6



devise a program to output sum of values stored at odd positions of an array of 100 elements

trace it

to trace our solution, we do it for an array of a smaller size. let's assume size of the array is **3**, and indexing starts from 1. also, assume the values stored in the array are 6, 3, and 12



a[1]	a[2]	a[3]	sum	i	output
6	**3**	**12**			
			0		
				1	
			6		
				3	
			18		
				5	
					sum=18

advanced example 7

devise a program to output sum of even values stored in an array of 100 elements

Do it yourself. The flowchart should look like the previous examples, except for—inside the loop—you need to check if the value is odd or even; add it to sum only if it's even.

trace it

to trace your solution, do it for an array of a smaller size. Let’s assume size of the array is 5, and indexing starts from 0. Also, assume the values stored in the array are 3, 2, 22, 11, and 12. Your trace should like the following table

a[0]	a[1]	a[2]	a[3]	a[4]	sum	i	output
3	2	22	11	12			
					0		
						0	
						1	
					2		
						2	
					24		
						3	
						4	
					36		
						5	
							sum=36

advanced example 8

devise a program to receive two positive whole numbers and output their GCD

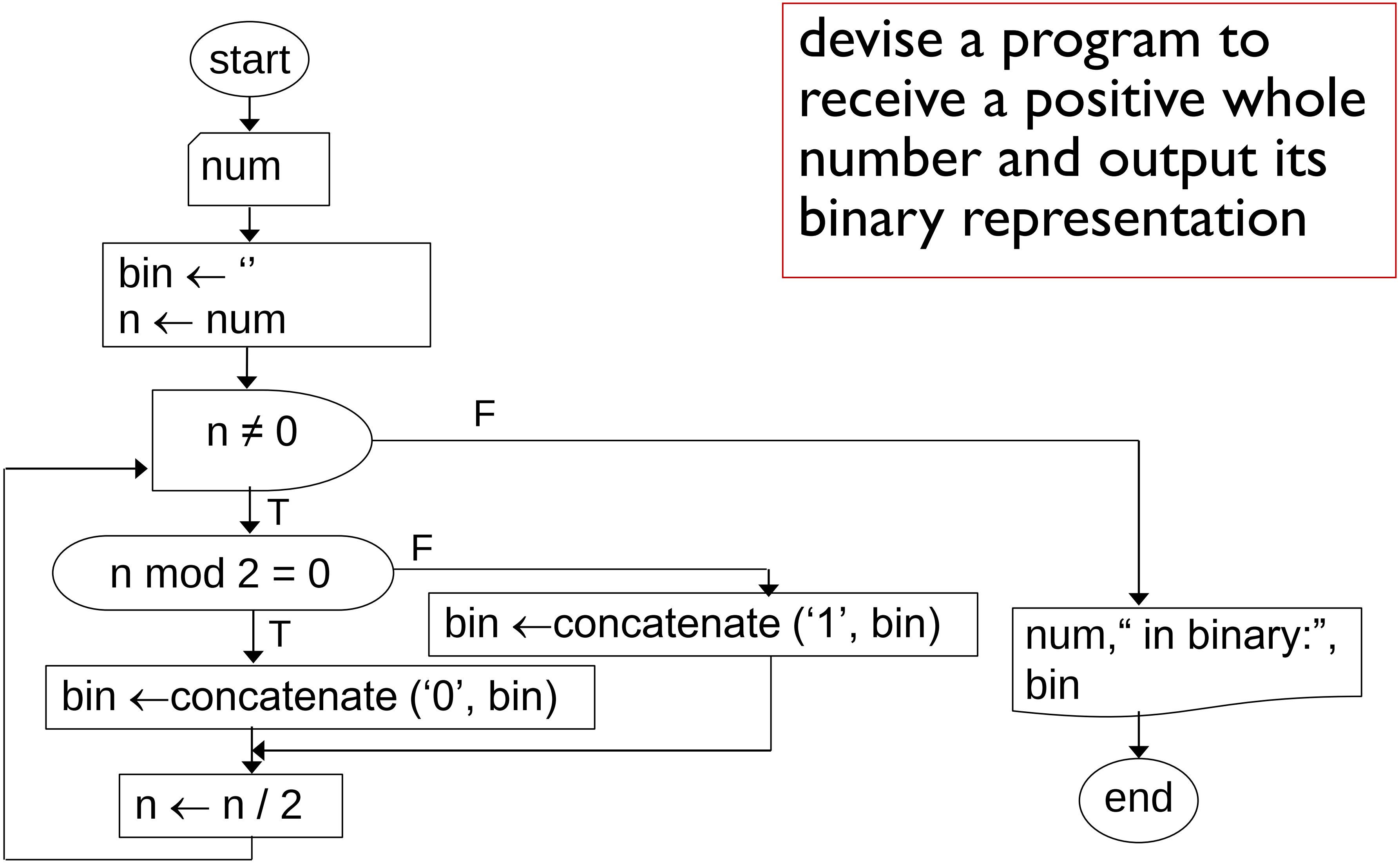
Do it yourself. First, you may want to review the Euclid's method for finding GCD, here.

trace it

trace your solution for the following 3 cases: when the two inputs are (12,30), (13,8) and (0, 15)

advanced example 9

devise a program to receive a positive whole number and output its binary representation



trace it

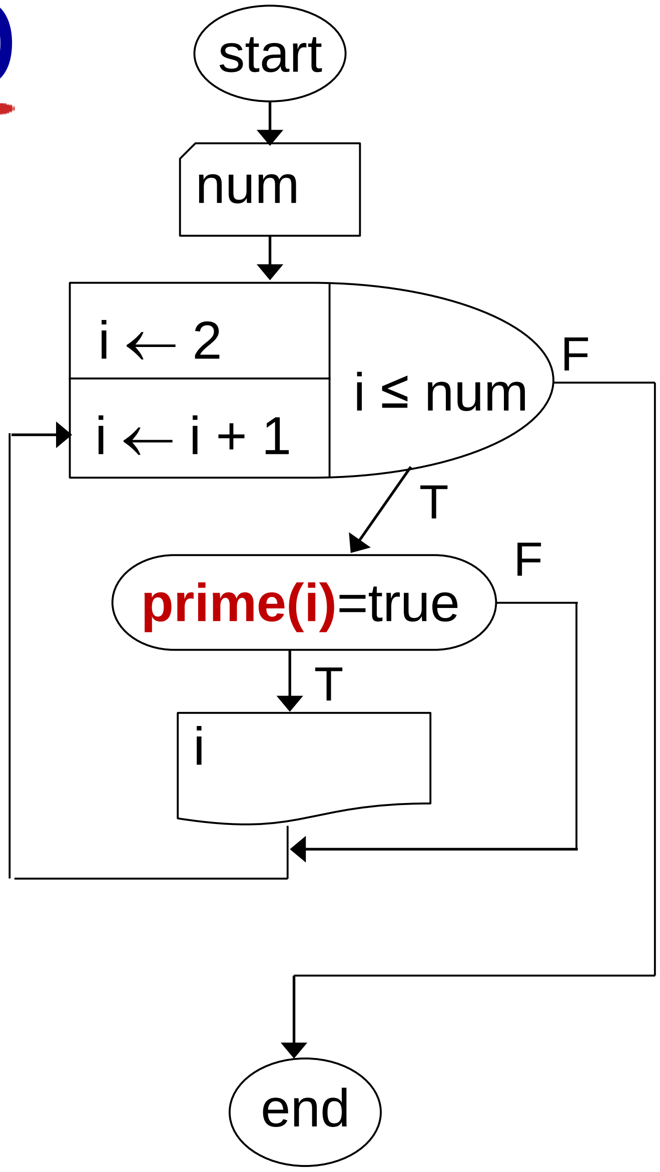
let’s trace it for when the input is 13:

num	bin	n	output
13			
	‘’		
		13	
	‘1’		
		6	
	‘01’		
		3	
	‘101’		
		1	
	‘1101		
		0	
			13 in binary: 1101

advanced example 10

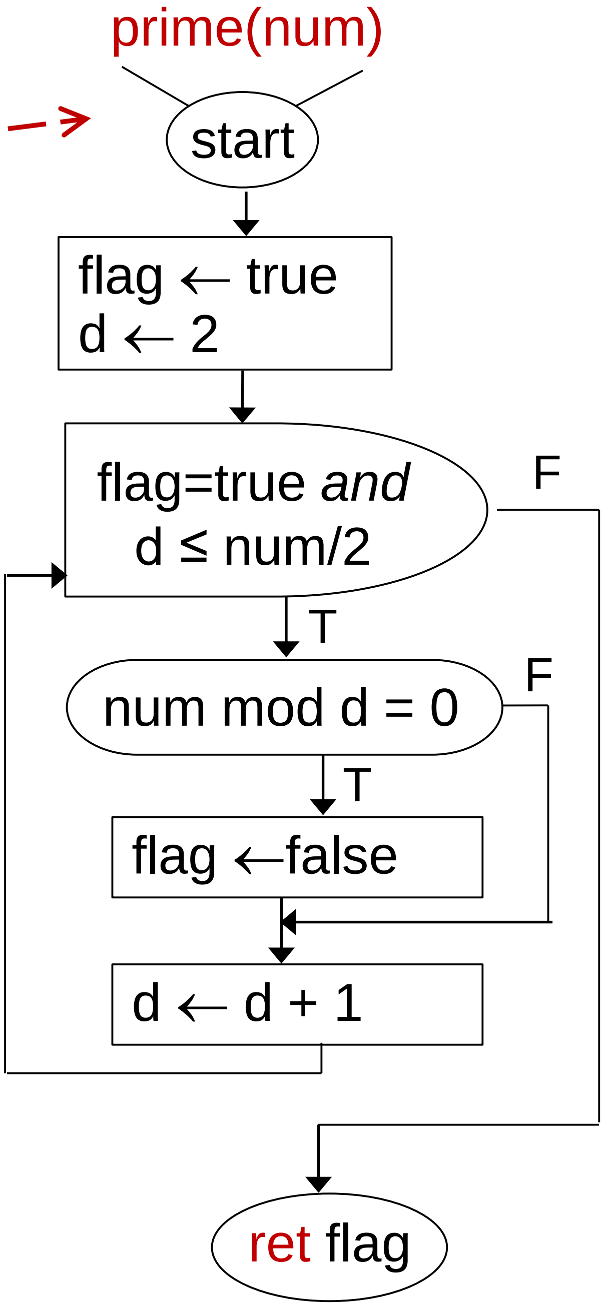
devise a program to receive a positive number, and print all prime numbers less than or equal to that number

we are going to reuse our solution to adv'd example 4 (to check if a number is **prime** or not). we need to modify it a bit to return a True or False answer though.



continued

- this is called **sub-algorithm**.
- a sub-algorithm is called from other algorithms
- like we called the prime sub-algorithm from inside our adv'd example 10
- when we trace an algorithm, we normally treat the sub-algorithms as a black box (and, we assume they are correct)
- **exercise:** trace adv'd example 10, for when input is 8



sotware dev life cycle (revisit)

- **requirements**
  - understand/analyze what the problem/project is
- **design**
  - find/define a solution and present it in a schematic way
- **implementation**
  - code it in a programming language
- **verification**
  - verify the solution for test cases and fix possible flaws

wrap up

- tips
  - understanding the project is a crucial phase
  - never miss the design phase
    - this is a common weakness/mistake of many
    - no matter how tiny/easy the problem/project is
  - use meaningful names for memory spaces
    - by the way, in programming languages we refer to a "memory space" by a "variable"; review all slides
  - nest properly
    - avoid use of goto, continue, or alike statements
  - verify the design before the implementation