DECISION STRUCTURES
CHAPTER 7 (EXCEPT FOR 7.4)

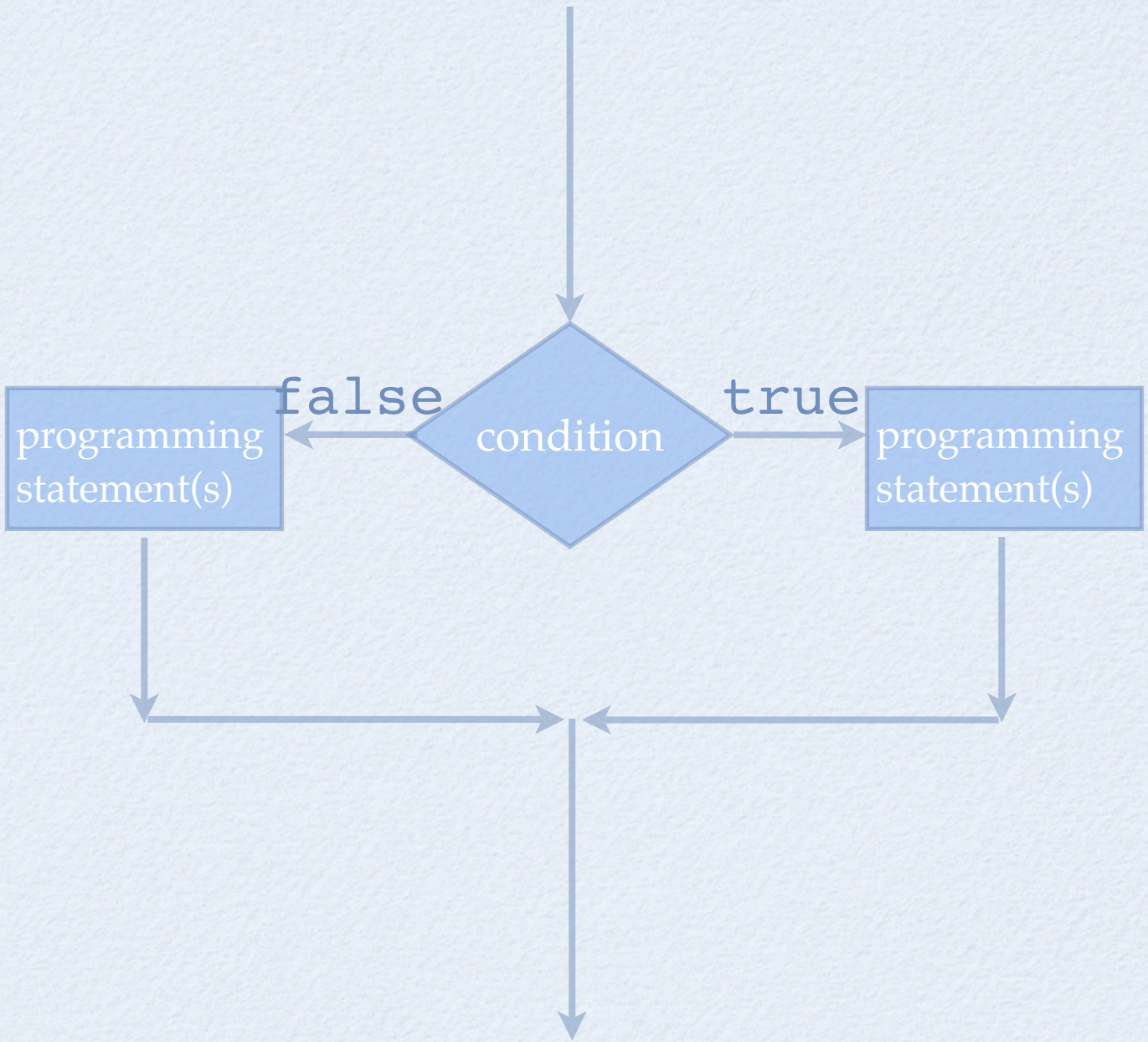
Learning Outcomes

- To understand the two-way decision programming pattern and its implementation using a Python if-else statement.
- To understand the multi-way programming pattern and its implementation using a Python if-elif-else statement.
- To understand the concept of Boolean expressions and the bool data type.
- To be able to read, write and implement algorithms that employ decision structures, including those that employ sequences of decisions and nested decision structures.

Two Way Decisions

- We have previously seen simple if statements.
- An "if" statement executes certain statements if some condition tests true.
- We can also have two-way decisions:
  - Execute one set of statements if the condition is true.
  - Execute another set of statements if the condition is not true.

Two-way decisions



IF-ELSE STATEMENTS

- Two-way decisions are called `if-else` statements.
- The syntax is similar to an `if` statement, followed by the keyword `else` and the "execute if `false`" block.

```
if <condition>:
   <body for true part>
else:
   <body for false part>
```

AN EXAMPLE

- Consider our quadratic equation program that we wrote earlier.
- Rewrite our program to calculate the roots to the equation only if solutions exist.
- Otherwise, print an error message.

MY ALGORITHM

1. Get the user to input the three coefficients (a, b, c)

QUICK EXERCISE

- Are the two program fragments equivalent?

```
if x < y:
   print("Hello")
else
   print("Bye")
```

```
if x > y:
   print("Bye")
else
   print("Hello")
```

My Answer:

EXERCISE

- Write a program that will ask the user for a number, and then print out whether it is even or odd.
  - Hint: use the modulo operator (%)

MY ALGORITHM

Multi-way decisions

- Consider the case of our quadratic equation solver again.
- When the discriminant is positive, we will get two solutions to the equation.
- When the discriminant (the value under the square root sign) is negative, there are no solutions.
- When the discriminant equals to zero, then we will have one solution.

Zero Discriminant

```
J:\> python quadratic.py

This program finds the solutions to a quadratic
equation, ax^2 + bx + c = 0

Please enter the coefficients (a, b, c):1, 2, 1



The solutions are: -1.0 -1.0

J:\>
```

That looks really odd!

Multi-way Decisions

- Wouldn't it make more sense if this were the output to our program?

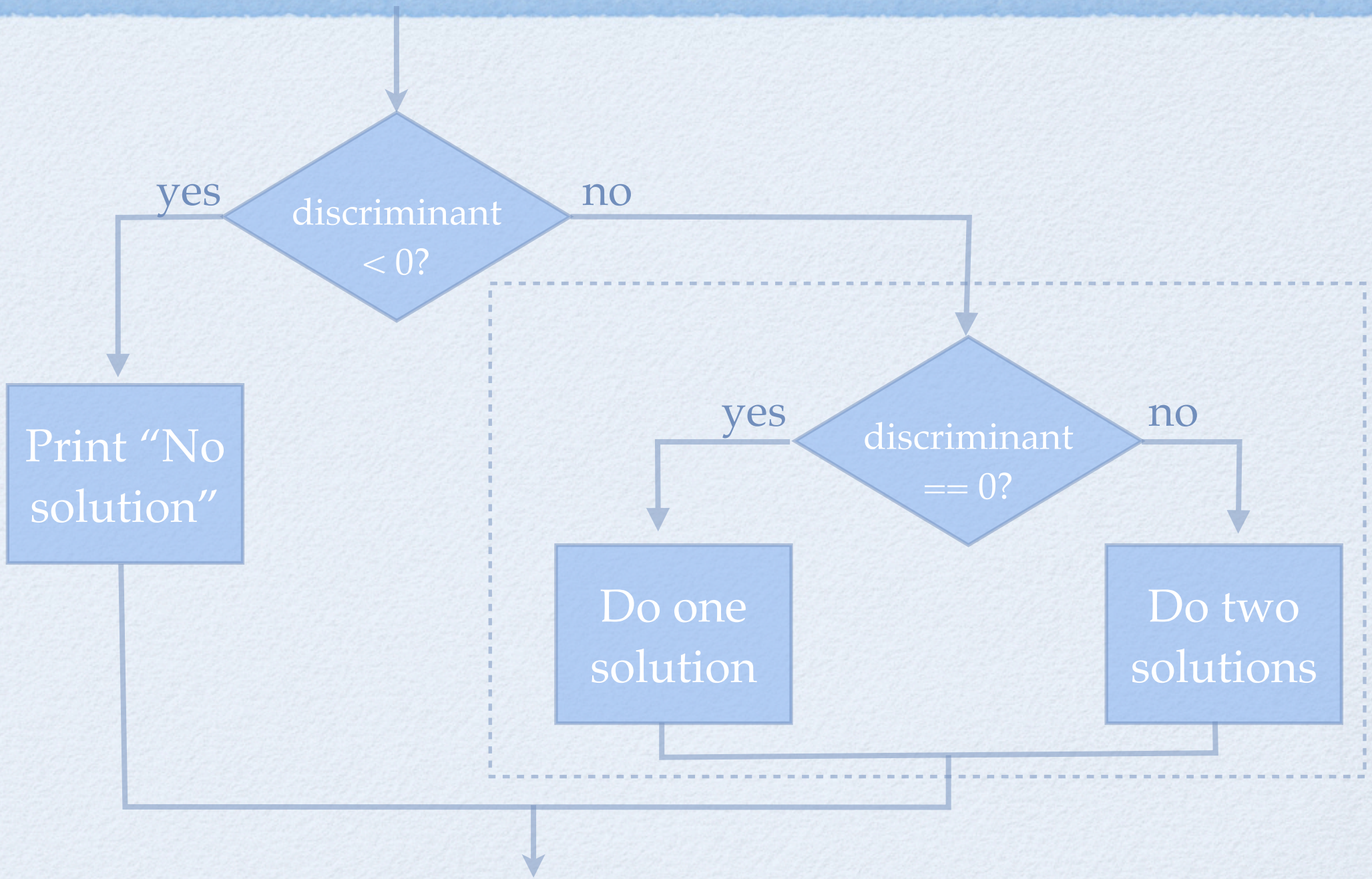
```
J:\> python quadratic3.py

This program finds the solutions to a quadratic
equation, ax^2 + bx + c = 0

Please enter the coefficients (a, b, c):1, 2, 1



The solution is: -1.0

J\:>
```

Looks much better!

Nested if statements

- Nesting (v): to fit inside one another.
- Idea:
  - We can put program statement(s) inside `if` statements.
  - We can even make function calls inside `if` statements.
  - Why can't we put an `if` statement inside another `if` statement?

NESTED IF FLOWCHART



WRITE THE CODE

```
discriminant = b*b - 4 * a * c

    print("There is no solution!")


    print("The solution is:", root)


    print("The solutions are:", root1, root2)
```

EXERCISE

```
score = eval(input("What is your score?"))
if score >= 60:
     if score >= 90:
       print("Congratulations! You got an A!")

print("Your score is", score)
```

- What will be printed if the score is:
  - 95?
  - 23?
  - 70?

EXERCISE

```
score = eval(input("What is your score?"))
if score >= 60:
     if score >= 90:
       print("Congratulations! You got an A!")
     else
       print("You passed the course!")
 else
      print("Sorry, you failed")
```

- What will be printed if the score is:
  - 95?
  - 23?
  - 70?

EXERCISE

- Suppose that x stores 7, and y stores 9, and z stores 11.
- What will be printed on the screen after executing the following program statements?

```
if x==1:
  if y==2:
    print(x)
  else
    print(y)
print(x+y)
```

elif statements

- The nested `if` statement was one way to deal with our problem.
- However, nested `if`s can get very tedious if we have more than 3 or 4 conditions

```
if <condition1>:
    ...
else
    if <condition2>:
      ...
    else
      if <condition3>:
        ...
      else
        ...
```

elif statements

- `elif` statements preserve the semantics (meaning) of nested `if` statements, but look more attractive.
- We combine an `else` that is followed immediately by an `if` into a single clause and call it `elif`:

```
if <condition1>:
    <case 1 statements>
elif <condition2>:
    <case 2 statements>
elif <condition3>:
    <case 3 statements>
elif <condition4>:
    <case 4 statements>
...
else:
  <default statement>
```

elif statements

- Conditions in `elif` statements are evaluated from the top down.
- The computer goes through each one until it finds the first one with a result of `true`.
- The rest of the conditions are then skipped and execution proceeds to the next statement.

What's wrong with this?

```
score = eval(input("What is your score?"))
if score > 60:
    print("You passed the course!")
elif score > 90:
    print("Congratulations! You got an A!")
else
    print("Sorry, you failed!")
```

Program Design: Case Study

- Decision structures mean that programs are not restricted to strictly sequential step-by-step execution of instructions.
- We can write more intelligent programs.
- But designing these programs is also harder.
- Case study: write a program that identifies the biggest of three numbers input by the user.

MAX OF THREE

- Here's the program outline:

```
def main():
    x1, x2, x3 = eval(input("I need three numbers"))



    # set the value of variable max to the largest number



    print("The largest number is", max)
```

- Obviously, there's a decision structure somewhere in there. But where? And how?

Strategy 1: Compare each to all

- We have three options for the maximum value: `x1`, `x2` or `x3`
- One straightforward way would be to test each of them in turn:
  - Start with `x1`, test if it is the biggest value.
  - If not, test if `x2` is the biggest.
  - If not, then `x3` has got to be the biggest.

Strategy 1

- x1 would be the biggest value if it is at least as big as x2 and at least as big as x3.
- Is this what we want?

```
if x1 >= x2 >= x3:
   max = x1
```

- First of all: is this *syntactically* correct? Does Python allow compound statements like this?
  - This is easily tested -- try it out in the IDLE interactive shell and see if it gives you an error!

Strategy 1: Testing

- We need to test and make sure that our algorithm works *under all conditions*.
- First: the positive condition.
  - When `x1 >= x2 >= x3`, does that mean that `x1` really is the largest?
- Then, the negative conditions:
  - When `x1 >= x2 >= x3` is not true, does that mean that `x1` definitely is NOT the largest?

Testing (cont'd)

- The second test fails because `x1 >= x2 >= x3` means that in order for the entire expression to test `true`,
  - `x1` has to be as least as big as `x2`, and
  - `x2` has to be as least as big as `x3`,
- This means that combinations such as `x1 = 5, x2 = 1, x3 = 3` won't work.
- Take-away point: when you are testing your programs, you have to test *all* cases, both positive and negative. Especially important, you need to test the *boundary cases*.

REFINEMENT

- What we actually need is for `x1` to be at least as big as `x2` and at least as big as `x3`.
  - We don't care about the ordering of `x2` and `x3`.
  - So we need two separate Boolean statements, combined together.

```
if x1 >= x2 and x1 >= x3:
    max = x1
```

- We will learn the details about Boolean conjunctions in the next chapter, but we can use the `and` for now.

Completing the Program

- Now we can complete the program by doing the same for `x2` and `x3`:

```
if x1 >= x2 and x1 >= x3:
   max = x1
elif x2 >= x1 and x2 >= x3:
   max = x2
else:
   max = x3
```

- In summary, our algorithm checks each possible value against all the others to determine if it is the max.

Imperfections with Strategy 1

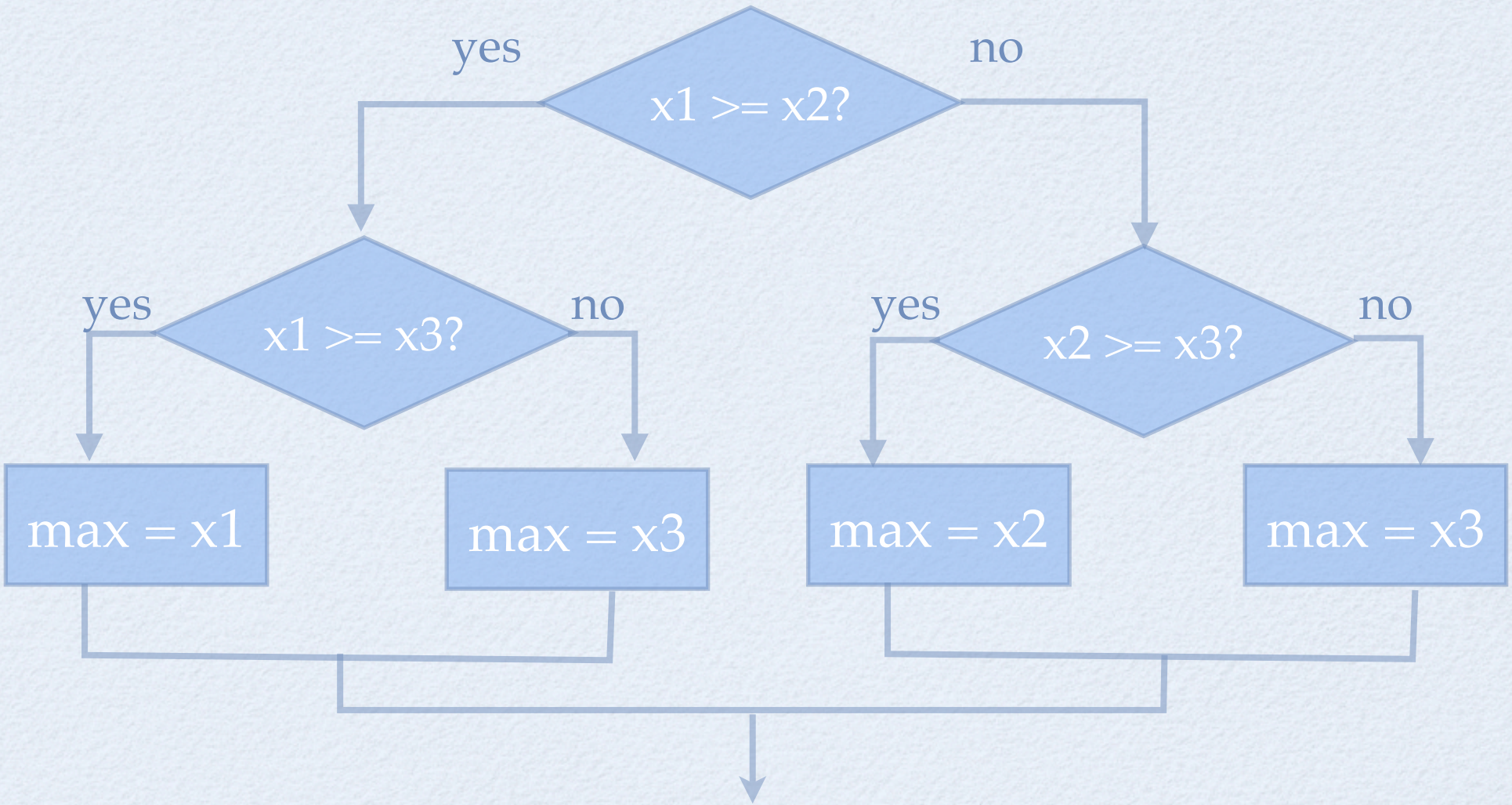
- Our strategy works, but there are some problems:
  - It is not very **scalable** -- if we want to find the maximum of 5 numbers, our program gets much longer and more complex.
  - It is not very **efficient**.
    - Consider the first condition. If x1 is greater than x2 but not greater than x3, then x3 must be the max, but our program doesn't know that.

Strategy 2: Decision Tree

- We can address the efficiency issue by sharing the comparison tests:
  - Suppose that x1, x2, x3 = 3, 1, 5
  - x1 >= x2 is true. Therefore, we know that x2 cannot possibly be the max.
  - Now we compare x1 >= x3, which is false, so we know the maximum is x3.
  - 2 comparisons instead of 3.

DECISION TREE

- Basically, we "branch" out into different possibilities



OUR PROGRAM

- Try to write up the programming statements that describe this decision tree.

```
if x1 >= x2:
```

Strategy 2: Problems

- The strength of strategy 2 is that it is efficient.
  - No matter what the ordering of the three values are, only two comparisons are made.
- However, it has the same scalability problem as the first strategy.
  - If we had to find the maximum of more than three values, the program gets much more complex very quickly (complexity explosion).
  - e.g. 4 values will need three levels of nested `if-else`s
- You will learn more about efficiency in COMP 305.

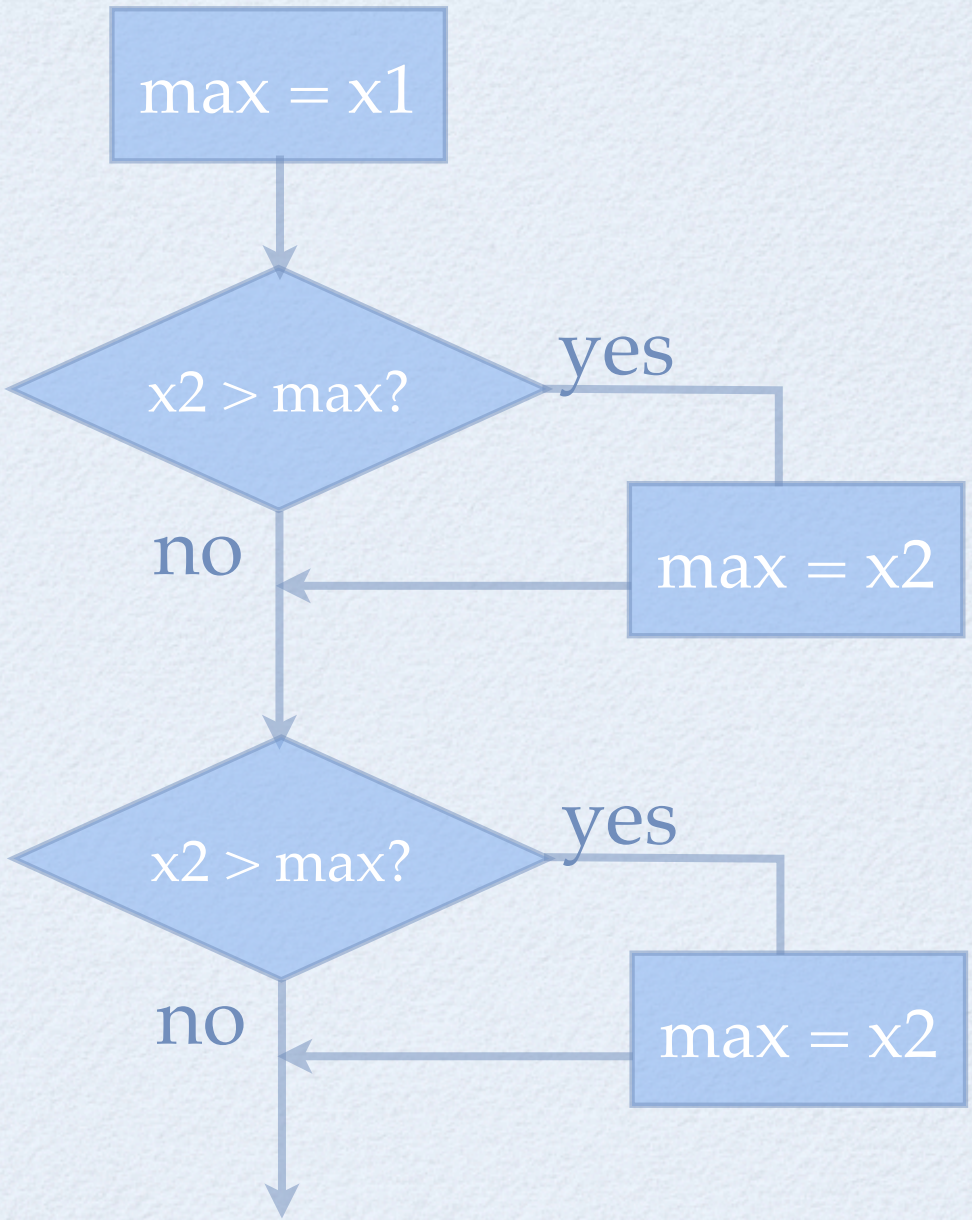
Strategy 3: Sequential Processing

- Sequential processing is also called **linear search**.
- Think: If I gave you a list of 1000 numbers, and asked you to find (manually) the biggest one.
- How would you do it?

Strategy 3

- Sequential processing looks through the list one number at a time.
- We keep track of the largest number seen so far.
  - Use a variable to "remember" this number.
  - We can actually just use the variable `max`!
- When we get to the end of the list, then we have the largest number.

Strategy 3 Flowchart



CONVERT TO PYTHON

```
max = x1
```

Strategy 3 Strengths

- The best thing about strategy 3 is its **scalability**.
- Imagine that we want to find the maximum of 5 numbers.
- We can easily add 4 more lines to the program.
- How about if we were to find the maximum of *n* values, input by the user?
  - Hint: Use a loop, put the `if` statement inside it.

N-WAY MAXIMUM

- My Program:

```
def main():
    n = eval(input("How many numbers?"))

    # set max to the first value


    # check the next n-1 values



    # print out max


main()
```

SUMMARY

- You now know:
  - What decision structures and Boolean conditions are.
  - How to control program flow via selection (executing certain program statements only if certain conditions are true).
  - How to design simple programs that consist of sequences of decision structures or nested decison structures.
  - How to test a program for boundary cases.