

If ... then ... else ...
(4. přednáška)



FF UK

Department of Logic

Jonathan L. Verner

- ▶ v předchozích přednáškách byly programy v podstatě lineární

- ▶ v předchozích přednáškách byly programy v podstatě lineární
- ▶ “lineární” algoritmy jsou schopné vyřešit relativně velkou třídu problémů

- ▶ v předchozích přednáškách byly programy v podstatě lineární
- ▶ “lineární” algoritmy jsou schopné vyřešit relativně velkou třídu problémů
- ▶ někdy to ale není ono

```
from math import sqrt

def solutions(a,b,c):
    disk = b**2 - 4*a*c
    x1 = (-b + sqrt(disk))/(2.0*a)
    x2 = (-b - sqrt(disk))/(2.0*a)
    return (x1, x2)

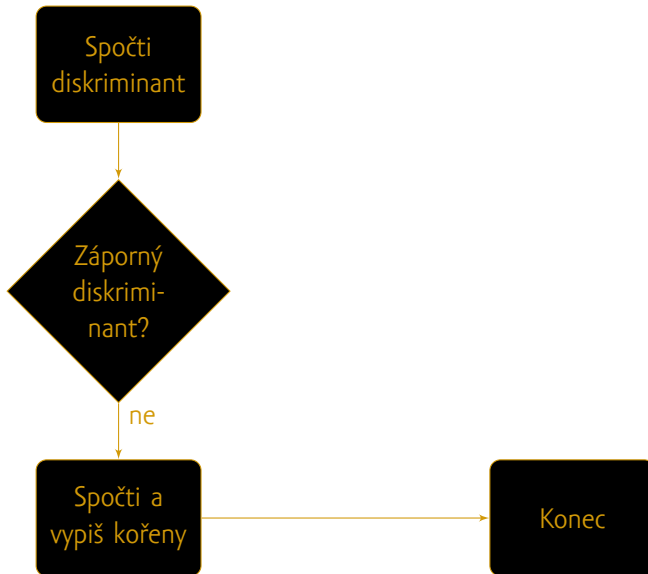
def solvequad():
    print("Zadejte koeficienty a,b,c")
    a,b,c=int(input("a:")),int(input("b:")),int(input("c:"))
    print("Koreny jsou:", solutions(a,b,c))
```

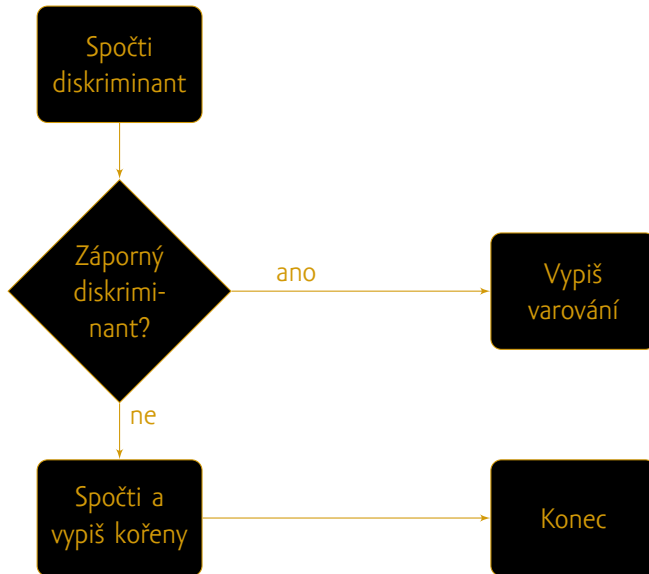
```
>>> solvequad()
Zadejte koeficienty a,b,c
a:10
b:10
c:10
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    solvequad()
  File "/tmp/quad.py", line 12, in solvequad
    print("Koreny jsou:", solutions(a,b,c))
  File "/tmp/quad.py", line 5, in solutions
    x1 = (-b + sqrt(diskr))/(2.0*a)
ValueError: math domain error
```

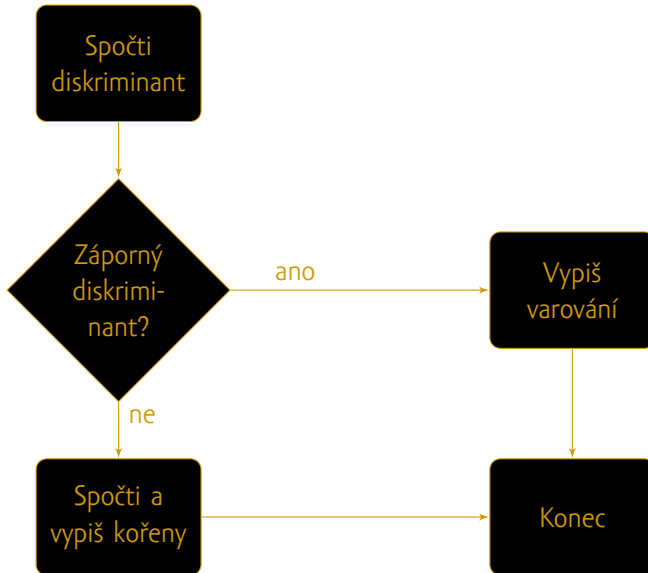
Spočti
diskriminant











```
from math import sqrt

def solutions(a,b,c):
    disk = b**2 - 4*a*c
    if disk >= 0:
        x1 = (-b + sqrt(disk))/(2.0*a)
        x2 = (-b - sqrt(disk))/(2.0*a)
        return (x1, x2)
    else:
        return "Rovnice nema (realne) reseni"

def solvequad():
    print("Zadejte koeficienty a,b,c")
    a,b,c=int(input("a:")),int(input("b:")),int(input("c:"))
    print("Koreny jsou:", solutions(a,b,c))
```

```
>>> solvequad()
```

```
Zadejte koeficienty a,b,c
```

```
a:10
```

```
b:10
```

```
c:10
```

```
Koreny jsou: Rovnice nema (realne) reseni
```

If...else statement

```
if <podminka>:  
    blokA  
else:  
    blokB
```

```
if <podminka>:  
    blokA  
else:  
    blokB
```

- ▶ <podminka> musi byt vyraz, ktery ma hodnotu typu Boolean


```
if <podminka>:  
    blokA  
else:  
    blokB
```

- ▶ <podminka> musi byt vyraz, ktery ma hodnotu typu **Boolean**
- ▶ pokud ma <podminka> hodnotu **True**, provede se **blokA**

```
if <podminka>:  
    blokA  
else:  
    blokB
```

- ▶ <podminka> musí být výraz, který má hodnotu typu **Boolean**
- ▶ pokud má <podminka> hodnotu **True**, provede se **blokA**
- ▶ jinak se provede **blokB**

```
if <podminka>:  
    blokA  
else:  
    blokB
```

- ▶ <podminka> musí být výraz, který má hodnotu typu **Boolean**
- ▶ pokud má <podminka> hodnotu **True**, provede se **blokA**
- ▶ jinak se provede **blokB**
- ▶ **else:** ... nemusí být přítomno

Matematický zápis	Zápis v Pythonu
$<$	<code><</code>
$>$	<code>></code>
\leq	<code><=</code>
\geq	<code>>=</code>
$=$	<code>==</code>
\neq	<code>!=</code>

```
>>> x=5
```

```
>>> x > 2 or x < 6
```

```
True
```

```
>>> x > 2 and x > 6
```

```
False
```

```
>>> not x < 3
```

```
True
```

Booleovské výrazy

not, and, or

```
>>> x=5
>>> x > 2 or x < 6
True
>>> x > 2 and x > 6
False
>>> not x < 3
True
```

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Napište program, který se uživatele zeptá na booleovský výraz obsahující čtyři proměnné a, b, c, d a vypíše tabulku obdobnou té předchozí

Hint: Použijte funkci `eval` a cyklama projdete všechny možnosti.

```
>>> tabulka()
```

```
Zadej bool vyraz: (a or b) and (c or d)
```

```
  A | B | C | D | (a or b)and(c or d)
```

```
-----
```

True		True		True		True		True
True		True		True		False		True
True		True		False		True		True
True		True		False		False		False
True		False		True		True		True
True		False		True		False		True
True		False		False		True		True
True		False		False		False		False
False		True		True		True		True
False		True		True		False		True
False		True		False		True		True
False		True		False		False		False
False		False		True		True		False
False		False		True		False		False
False		False		False		True		False
False		False		False		False		False


```
>>> solvequad()  
Zadejte koeficienty  
a:1  
b:2  
c:1  
Koreny jsou: (-1.0, -1.0)
```

```
>>> solvequad()
```

Zadejte koeficienty

a:1

b:2

c:1

Koreny jsou: (-1.0, -1.0)

- ▶ to může být trochu matoucí
- ▶ chtěli bychom, aby pro dvounásobné kořeny vypsali hlášku

```
>>> solvequad()
```

Zadejte koeficienty

a:1

b:2

c:1

Koreny jsou: (-1.0, -1.0)

- ▶ to může být trochu matoucí
- ▶ chtěli bychom, aby pro dvounásobné kořeny vypsal hlášku

spočti diskriminant

pokud:

$\text{diskr} < 0$ vypiš chybovou hlášku o imaginárních kořenech

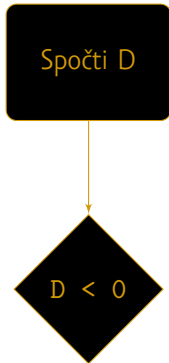
$\text{diskr} == 0$ vypiš hlášku o dvojitém kořeni

$\text{diskr} > 0$ vypiš oba kořeny

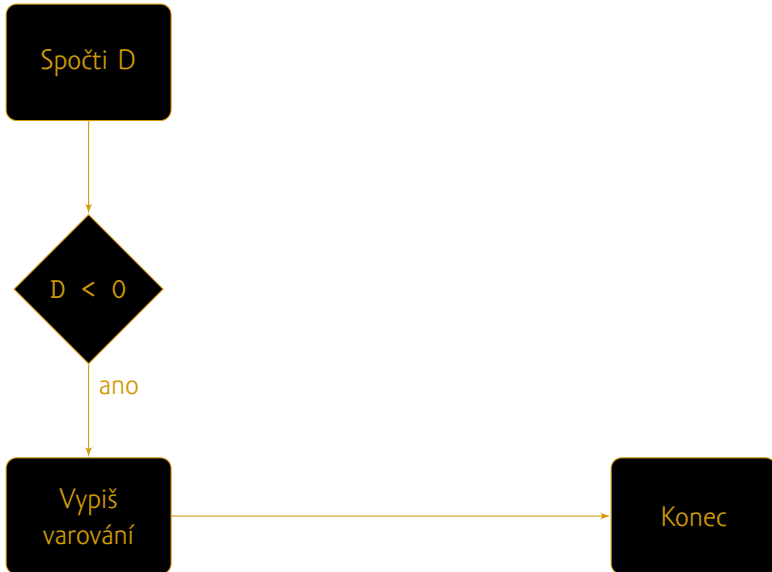
```
from math import sqrt

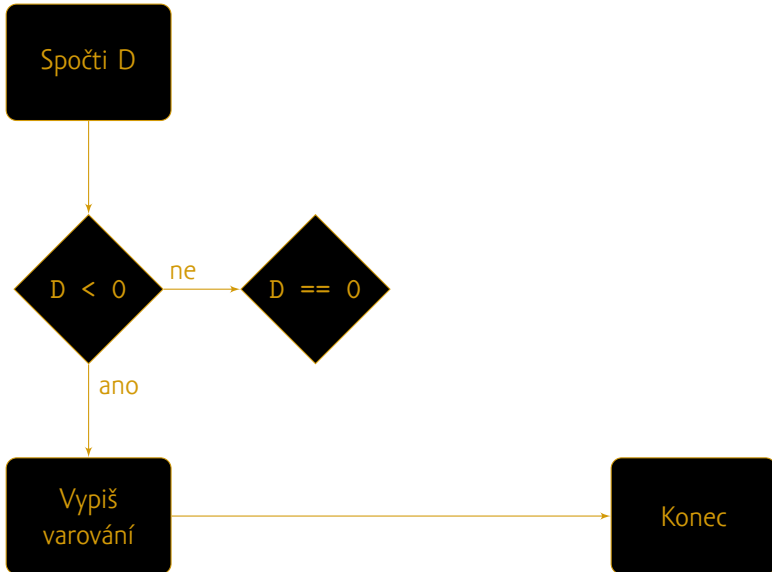
def solutions(a,b,c):
    D = b**2 - 4*a*c
    if D >= 0:
        x1 = (-b + sqrt(D))/(2.0*a)
        x2 = (-b - sqrt(D))/(2.0*a)
        if D == 0:
            return x1
        else:
            return x1, x2
    else:
        return "Rovnice nema (realne) reseni"
```

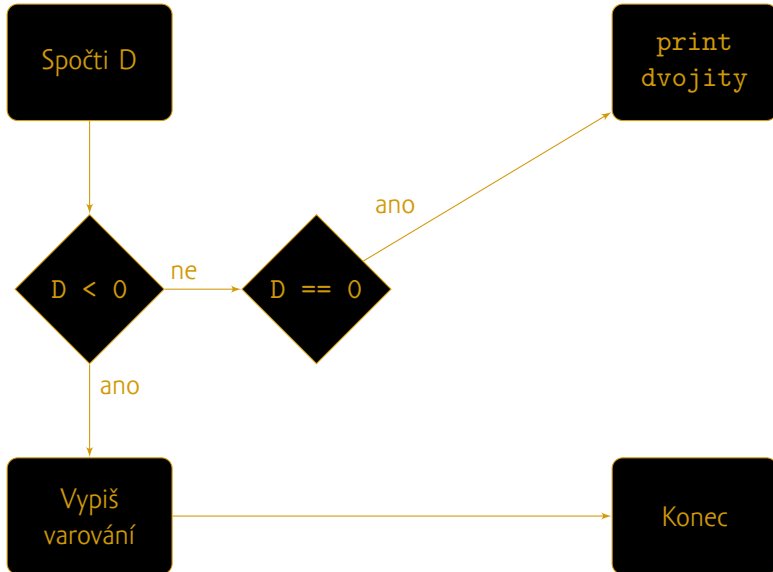
Spočti D

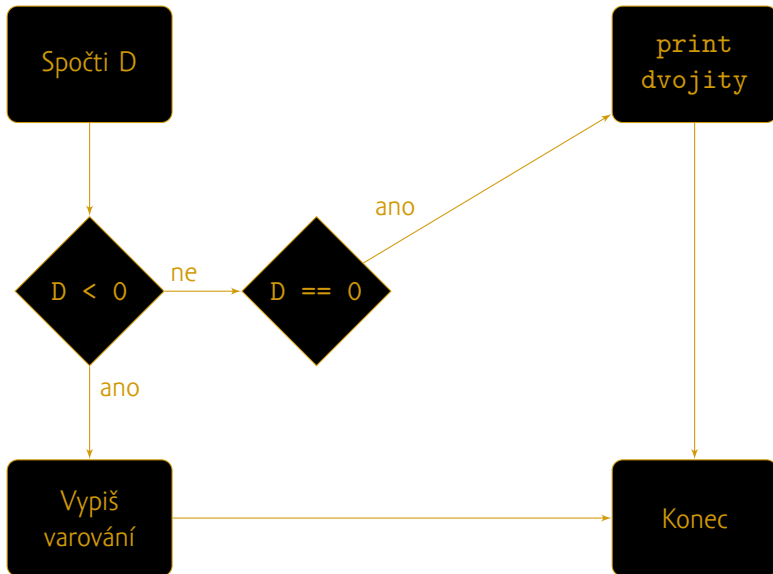


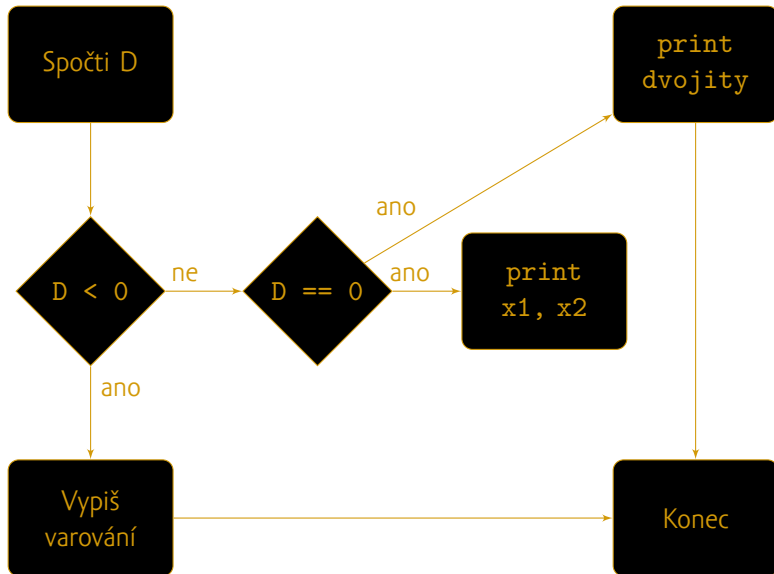


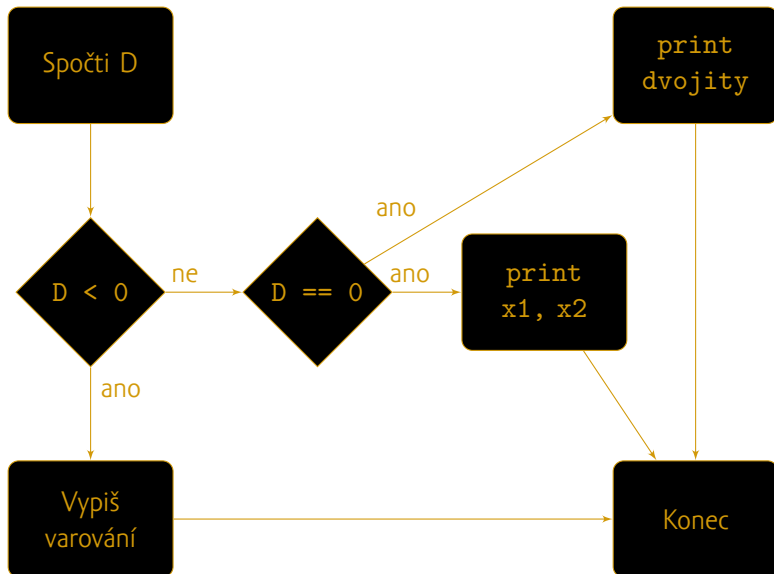












```
from math import sqrt

def solutions(a,b,c):
    D = b**2 - 4*a*c
    if D > 0:
        x1 = (-b + sqrt(D))/(2.0*a)
        x2 = (-b - sqrt(D))/(2.0*a)
    elif D == 0:
        return (-b + sqrt(D))/(2.0*a)
    else:
        return "Rovnice nema (realne) reseni"
```

```
if <podminka1>:  
    blok1  
elif <podminka2>:  
    blok2  
elif <podminka3>:  
    blok3  
...  
else:  
    blokE
```

- ▶ provede se vždy první blok, jehož podmínka je splněna
- ▶ provede se vždy **nejvýše** jeden blok!!

- ▶ jedno z typických použití 'iř'-ů: vyhnout se chybám

- ▶ jedno z typických použití 'if'-ů: vyhnout se chybám
- ▶ když je 'if'-ů moc, hlavní algoritmus se může ztratit

- ▶ jedno z typických použití 'if'-ů: vyhnout se chybám
- ▶ když je 'if'-ů moc, hlavní algoritmus se může ztratit
- ▶ jeden ze způsobů jak to řešit jsou tzv. **výjimky**

- ▶ jedno z typických použití 'if'-ů: vyhnout se chybám
- ▶ když je 'if'-ů moc, hlavní algoritmus se může ztratit
- ▶ jeden ze způsobů jak to řešit jsou tzv. **výjimky**

Dělej A a pokud dojde k chybě, proved' B

- ▶ jedno z typických použití 'if'-ů: vyhnout se chybám
- ▶ když je 'if'-ů moc, hlavní algoritmus se může ztratit
- ▶ jeden ze způsobů jak to řešit jsou tzv. **výjimky**

Dělej A a pokud dojde k chybě, proved' B

```
def solutions(a,b,c):  
    try:  
        D = b**2 - 4*a*c  
        x1 = (-b + sqrt(diskr))/(2.0*a)  
        x2 = (-b - sqrt(diskr))/(2.0*a)  
        return (x1, x2)  
    except ValueError:  
        return "Rovnice nema (realne) reseni"
```

try ... except

```
try:  
    <blokA>  
except <TypChyby1>[ as <identifikator1>]:  
    <blok1>  
except <TypChyby2>[ as <identifikator2>]:  
    <blok2>  
...  
except:  
    <blokE>
```

try ... except

```
try:
    <blokA>
except <TypChyby1>[ as <identifikator1>]:
    <blok1>
except <TypChyby2>[ as <identifikator2>]:
    <blok2>
...
except:
    <blokE>
```

- ▶ python prevede <blokA>

try ... except

```
try:  
    <blokA>  
except <TypChyby1>[ as <identifikator1>]:  
    <blok1>  
except <TypChyby2>[ as <identifikator2>]:  
    <blok2>  
...  
except:  
    <blokE>
```

- ▶ python provede <blokA>
- ▶ pokud dojde k chybe, vyhleda radku, ktera odpovida typu chyby, ke ktere doslo a vykona odpovidajici blok

```
try:  
    <blokA>  
except <TypChyby1>[ as <identifikator1>]:  
    <blok1>  
except <TypChyby2>[ as <identifikator2>]:  
    <blok2>  
...  
except:  
    <blokE>
```

- ▶ python provede <blokA>
- ▶ pokud dojde k chybe, vyhleda radku, ktera odpovida typu chyby, ke ktere doslo a vykona odpovidajici blok
- ▶ pokud je volitelne pritomny nazev promenne (<identifikatorn>), pak se do teto promenne vlozi informace o


```
try:  
    <blokA>  
except <TypChyby1>[ as <identifikator1>]:  
    <blok1>  
except <TypChyby2>[ as <identifikator2>]:  
    <blok2>  
...  
except:  
    <blokE>
```

- ▶ python provede <blokA>
- ▶ pokud dojde k chybe, vyhleda radku, ktera odpovida typu chyby, ke ktere doslo a vykona odpovidajici blok
- ▶ pokud je volitelne pritomny nazev promenne (<identifikatorn>), pak se do teto promenne vlozi informace o

verze 3.2 - exceptions

```
def solutions(a,b,c):
    try:
        D = b**2 - 4*a*c
        x1 = (-b + sqrt(diskr))/(2.0*a)
        x2 = (-b - sqrt(diskr))/(2.0*a)
        return (x1, x2)
    except ValueError as chyba:
        msg = unicode(chyba)
        if msg == 'math domain error':
            return "Rovnice nema (realne) reseni."
        else:
            print("Chyba: Neco se pokazilo.")
            return "Rovnice nema (realne) reseni"
    except TypeError:
        print("Chyba: Nektery koeficient nebyl ciselny")
        return "Rovnice nema (realne) reseni"
```

Napište program, který vypíše největší ze tří čísel

```
x1,x2,x3=eval(input("Zadej tri cisla oddelena carkou: "))
```

```
...
```

```
print "Maximum je ", max
```

1. verze

Procházení všech možností

- ▶ je třeba provést jedno z následujících přiřazení:

`max = x1`

`max = x2`

`max = x3`

1. verze

Procházení všech možností

- ▶ je třeba provést jedno z následujících přiřazení:

```
max = x1
```

```
max = x2
```

```
max = x3
```

- ▶ zdá se, že stačí akorát vložit odpovídající 'if'-y:

1. verze

Procházení všech možností

- ▶ je třeba provést jedno z následujících přiřazení:

```
max = x1
```

```
max = x2
```

```
max = x3
```

- ▶ zdá se, že stačí akorát vložit odpovídající 'if'-y:

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

```
    max = x1
```

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

```
    max = x2
```

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

```
    max = x3
```

1. verze

Procházení všech možností

- ▶ je třeba provést jedno z následujících přiřazení:

```
max = x1
```

```
max = x2
```

```
max = x3
```

- ▶ zdá se, že stačí akorát vložit odpovídající 'if'-y:

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

```
    max = x1
```

```
elif x2 >= x1 and x2 >= x3:
```

```
    max = x2
```

```
elif x3 >= x2 and x3 >= x1:
```

```
    max = x3
```

Co by se stalo, kdybychom chtěli počítat maximum ze 4?

- ▶ složitost podmínek by narostla

Co by se stalo, kdybychom chtěli počítat maximum ze 4?

- ▶ složitost podmínek by narostla
- ▶ pokud bychom chtěli zjistit maximum z n -čísel, potřebovali bychom n -podmínek, každá by měla

Co by se stalo, kdybychom chtěli počítat maximum ze 4?

- ▶ složitost podmínek by narostla
- ▶ pokud bychom chtěli zjistit maximum z n -čísels, potřebovali bychom n -podmínek, každá by měla $n - 1$ členů, t.j. $n(n - 1)$ porovnání!

Co by se stalo, kdybychom chtěli počítat maximum ze 4?

- ▶ složitost podmínek by narostla
- ▶ pokud bychom chtěli zjistit maximum z n -čísel, potřebovali bychom n -podmínek, každá by měla $n - 1$ členů, t.j. $n(n - 1)$ porovnání!
- ▶ podmínky neberou v potaz, co už víme z předchozích

Co by se stalo, kdybychom chtěli počítat maximum ze 4?

- ▶ složitost podmínek by narostla
- ▶ pokud bychom chtěli zjistit maximum z n -čísel, potřebovali bychom n -podmínek, každá by měla $n - 1$ členů, t.j. $n(n - 1)$ porovnání!
- ▶ podmínky neberou v potaz, co už víme z předchozích

Rozhodovací strom

Pokud už vím, že

$$x1 \geq x2$$

Rozhodovací strom

Pokud už vím, že

$$x1 \geq x2$$

pak mi stačí testovat jen

$$x1 \geq x3$$

Rozhodovací strom

Pokud už vím, že

$$x1 \geq x2$$

pak mi stačí testovat jen

$$x1 \geq x3$$

Pokud naopak není pravda, že

$$x1 \geq x2$$

Rozhodovací strom

Pokud už vím, že

$x1 \geq x2$

pak mi stačí testovat jen

$x1 \geq x3$

Pokud naopak není pravda, že

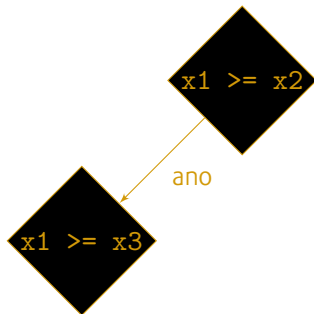
$x1 \geq x2$

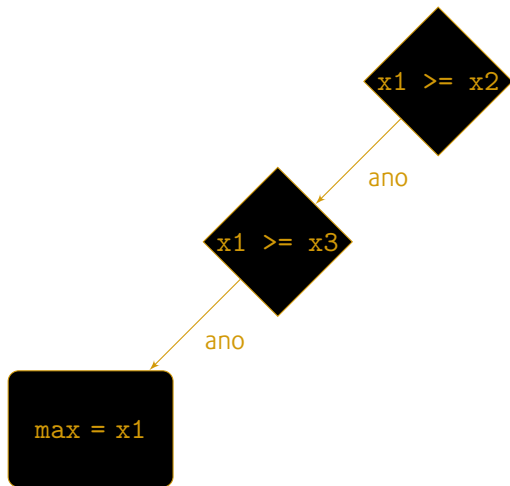
pak mi stačí testovat

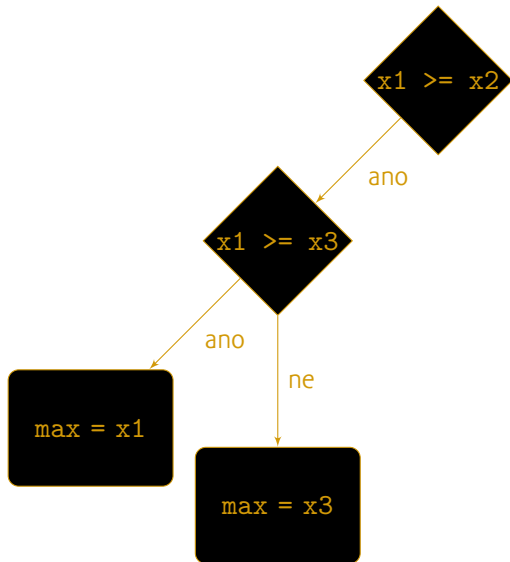
$x2 \geq x3$
if $x1 \geq x2$:

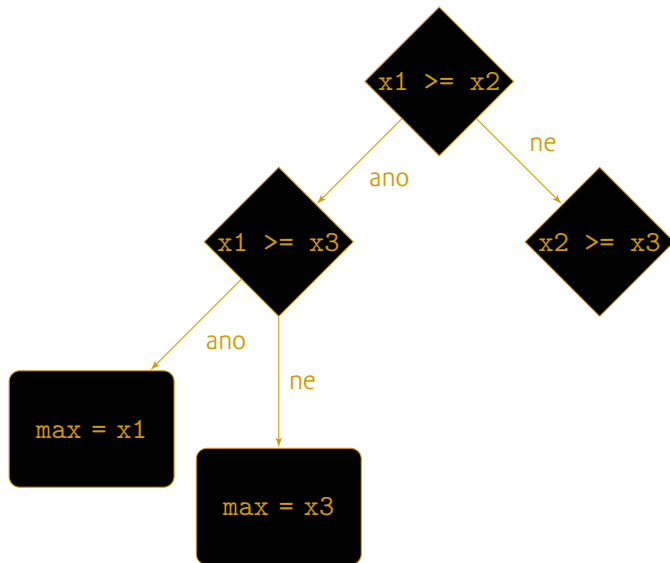
if $x1 \geq x3$:

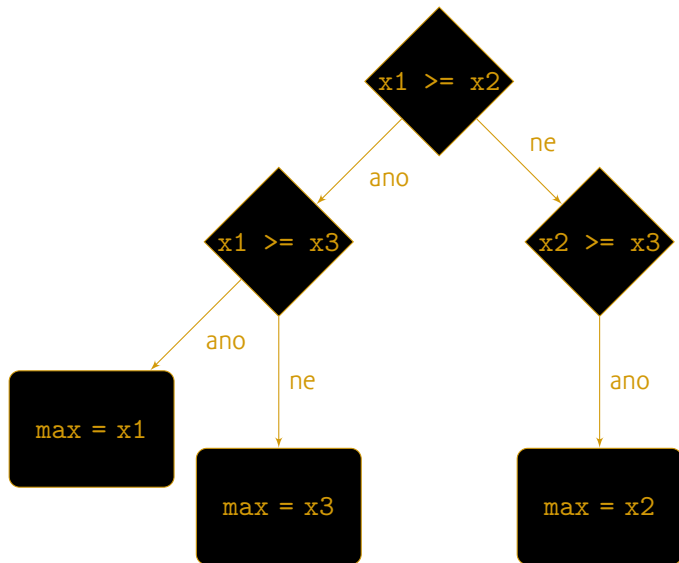


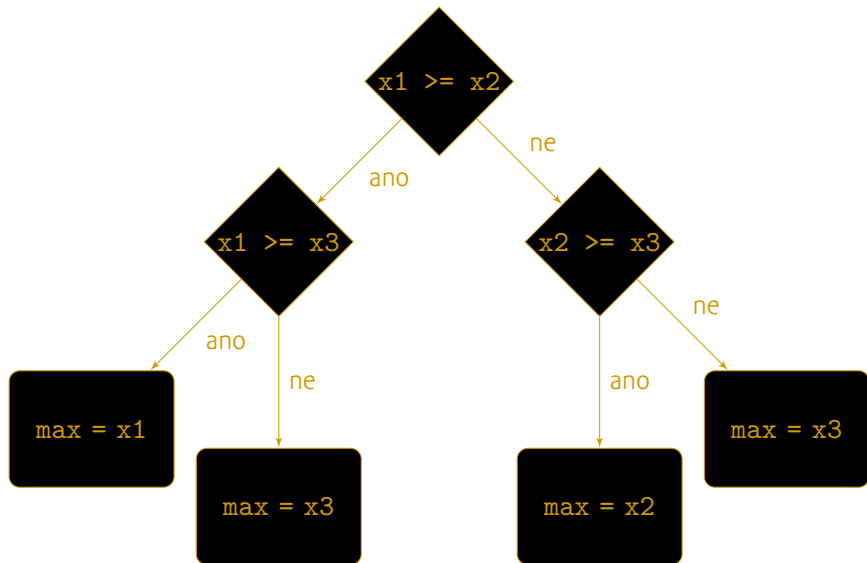


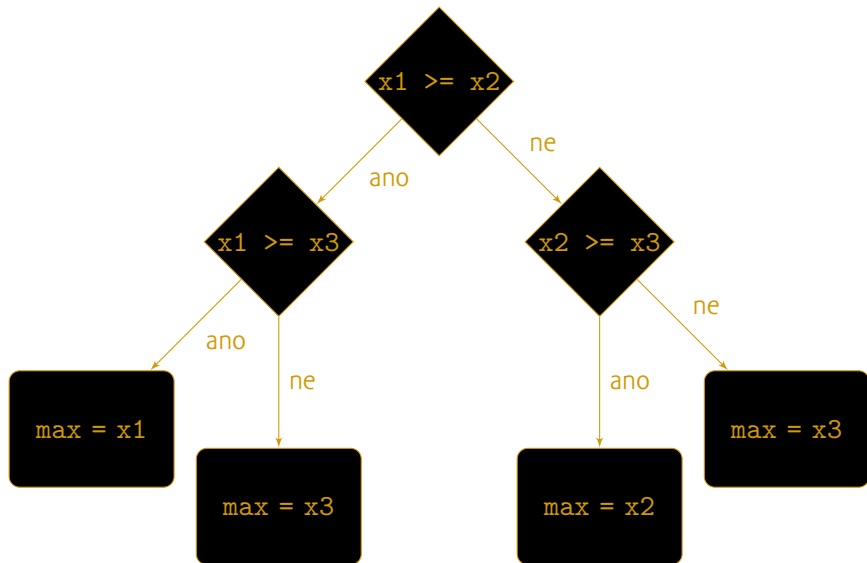












- ▶ výpočetně efektivní: minimum nutných porovnáání

- ▶ výpočetně efektivní: minimum nutných porovnáání
- ▶ neefektivní zápis !!!

3. verze

- ▶ jak by to řešil člověk?

3. verze

- ▶ jak by to řešil člověk?
- ▶ pro 3 čísla nejspíše metodou kouknu a vidím

3. verze

- ▶ jak by to řešil člověk?
- ▶ pro 3 čísla nejspíše metodou kouknu a vidím
- ▶ pokud ale čísel bude 1000?

3. verze

- ▶ jak by to řešil člověk?
- ▶ pro 3 čísla nejspíše metodou kouknu a vidím
- ▶ pokud ale čísel bude 1000?
- ▶ sekvenciální průchod

3. verze

- ▶ jak by to řešil člověk?
- ▶ pro 3 čísla nejspíše metodou kouknu a vidím
- ▶ pokud ale čísel bude 1000?
- ▶ sekvenciální průchod

3. verze - sekvenciální průchod

```
nastav max na prvni clen posloupnosti
prejdi k dalsimu clenu
je-li aktualni clen vetsi nez max:
    nastav max na aktualni clen
prejdi k dalsimu clenu
```


3. verze - sekvenciální průchod

```
nastav max na prvni clen posloupnosti
prejdi k dalsimu clenu
je-li aktualni clen vetsi nez max:
    nastav max na aktualni clen
prejdi k dalsimu clenu
```

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

verze 3.1 - obecné řešení

```
# program: maxofn.py
# Najde maximum z posloupnosti cisel
# autor: Student Logiky

def maximum():
    n = int(input('Delka posloupnosti:'))

    max = float(input('Zadej prvni clen:'))

    for i in range(n-1):
        x = float(input('Zadej dalsi clen:'))
        if x > max:
            max = x

    print("Maximum je", max)
```

4. verze - "podvodné řešení"

```
def maximum():  
    x1, x2, x3 = eval(input("Zadej tri cisla (a,b,c):"))  
    print("Nejvetsi cislo je", max(x1,x2,x3))
```

- ▶ většinou je více cest, jak dojít k řešení

- ▶ většinou je více cest, jak dojít k řešení
- ▶ ne vždy je první nápad nejlepší

- ▶ většinou je více cest, jak dojít k řešení
- ▶ ne vždy je první nápad nejlepší
- ▶ často (ne vždy) je dobré se zamyslet nad obecným případem

- ▶ většinou je více cest, jak dojít k řešení
- ▶ ne vždy je první nápad nejlepší
- ▶ často (ne vždy) je dobré se zamyslet nad obecným případem
- ▶ "do not reinvent the wheel"