



# **Programovanie v jazyku Python**

Optimalizácia, dynamické programovanie  
prednáška 5

Katedra kybernetiky a umelej inteligencie  
Technická univerzita v Košiciach  
Ing. Ján Magyar

# Optimalizácia kódu

- cieľom je zvýšiť výkon existujúceho programu
- súčasť ladenia kódu
- eliminácia zbytočných a viacnásobných operácií

# Viacnásobné prechádzanie zoznamom

```
my_list = [1240, -25, 37.24, -12, 0, 35000, 24, 17.23]
```

```
for x in range(len(my_list)):
    if my_list[x] < 0:
        my_list[x] = 0
```

```
for x in range(len(my_list)):
    my_list[x] *= 1.05
```

```
print(my_list)
```

# Zbytočné operácie

```
def is_prime(number):  
    for x in range(2, number):  
        if number % x == 0:  
            return False  
    return True
```

```
is_prime(123475862311)
```

# Dynamické programovanie

- slúži pre optimalizáciu problémov s exponenciálnou zložitosťou
- aplikácia
  - pretínajúce sa podproblémy - Fibonacciho čísla
  - optimálna štruktúra - knapsack problem

# Fibonacciho čísla

```
steps = 0
```

```
def fib(a):  
    global steps  
    steps += 1  
    print("Calculating fib for", a)  
    if a == 0 or a == 1:  
        return 1  
    else:  
        return fib(a - 1) + fib(a - 2)
```

# Fibonacciho čísla - jednoduchšie

```
def fib_smart(a, memo):  
    global num_calls  
  
    num_calls += 1  
    print("fib_smart called with", a)  
    if a not in memo:  
        memo[a] = fib_smart(a - 2, memo) + fib_smart(a - 1, memo)  
    return memo[a]  
  
n = 10  
memo = {0: 1, 1: 1}  
num_calls = 0  
fib_smart(n, memo)
```

# Memoizácia

- čiastočné výsledky ukladáme do tabuľky
- ak sme ešte nevypočítali výsledok pre daný vstup, vypočítame a pridáme ho do tabuľky
- pri opätovnom volaní funkcie iba načítame výsledok z tabuľky (table lookup)
- všetky volania funkcie musia pracovať s tou istou tabuľkou



# Teoréma optimálnej štruktúry

Ak riešenia podproblémov sú lokálne optimálne, potom celkové riešenie bude globálne optimálne.

# Knapsack problem (problém batohu)

- klasický problém optimalizačných algoritmov
- zlodej sa vlámал do bytu a chce ukradnúť cennosti v najvyššej možnej hodnote
- zlodej má iba jeden batoh s maximálnou nosnosťou  $W$
- každý objekt v byte je popísaný pomocou dvojice  $(m, w)$  kde  $m$  je hodnota objektu a  $w$  je jeho hmotnosť
- úlohou je nájsť množinu najcennejších objektov, ktorých celková hmotnosť nepresiahne  $W$

# Greedy riešenie knapsack problému

- greedy algoritmy hľadajú vždy najlepšie okamžité riešenie
- riešenie: zober najcennejšie veci až dovtedy, kým nie je tvoj batoh plný
- nie vždy nájdeme optimálne riešenie, ale riešenie je zvyčajne dosť dobré
- jednoducho sa implementuje, je výpočtovo lacné

príklad:

$\mathbf{w} = [5, 3, 2]$

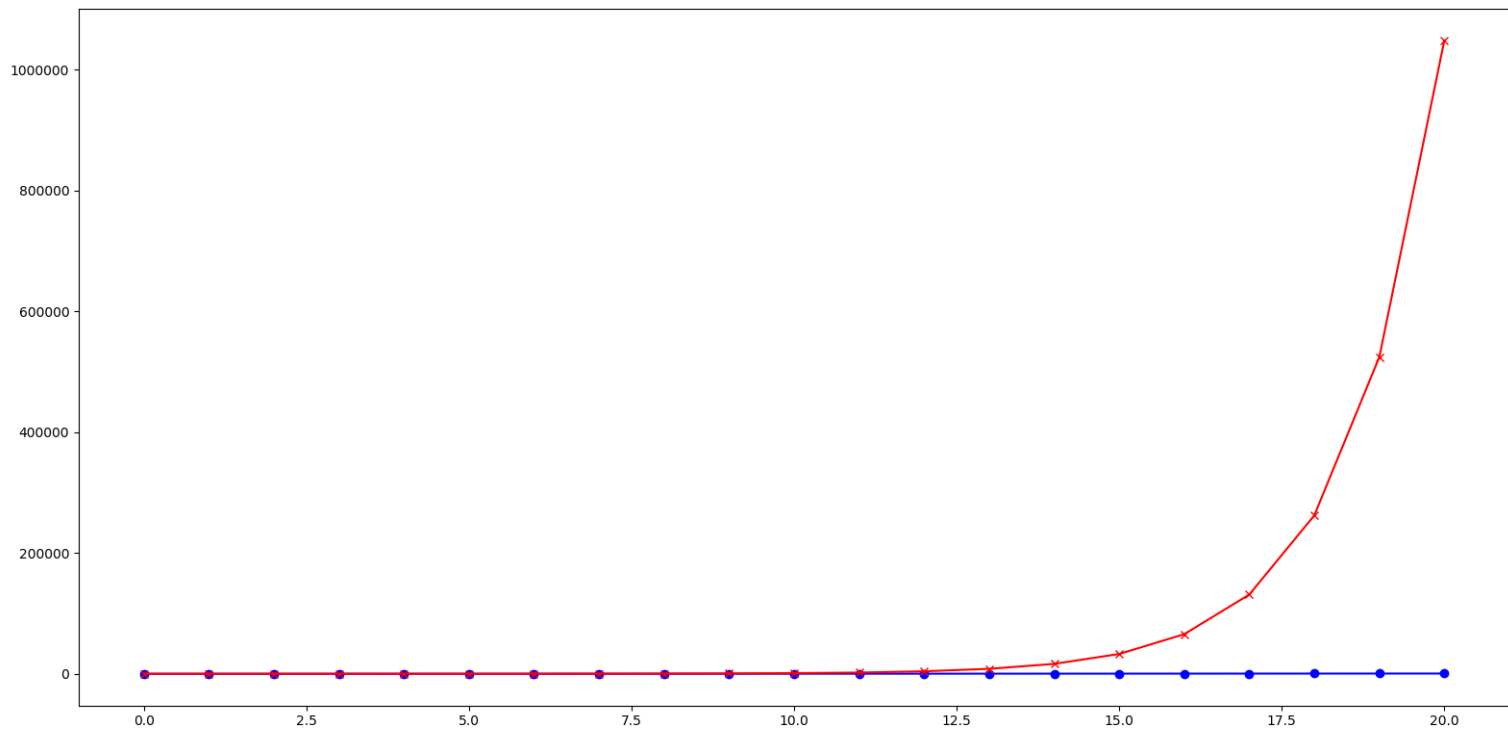
$\mathbf{m} = [9, 7, 8]$

$W = 5$

# Riešenie knapsack problému

- intuitívne – brute force
- vyskúšame všetky možnosti
- možné riešenia reprezentujeme ako vektor  $n$  hodnôt
- každá hodnota bude 0 alebo 1 (nezoberieme/zoberieme prvok)
- pre každý vektor skontrolujeme hmotnosť prvkov
- koľko máme možností?

# Prečo nepoužívame algoritmy s exponenciálnym rastom?



# Riešenie knapsack problému

- vhodná reprezentácia - rozhodovací strom
  - každý uzol je reprezentovaný ako trojica  $(i, w, m)$
  - ľavá vetva obsahuje prípady kde sme nezobrali objekt s indexom  $i$
  - pravá vetva obsahuje prípady kde sme zobrali objekt s indexom  $i$
  - optimálne riešenie je list s najväčšou hodnotou  $m$

príklad:

$$\mathbf{w} = [5, 3, 2]$$

$$\mathbf{m} = [9, 7, 8]$$

$$W = 5$$

```
def max_val(w, m, i, aW):
    global num_calls
    num_calls += 1
    if i == 0:
        if w[i] <= aW:
            return m[i]
        else:
            return 0
    without_i = max_val(w, m, i - 1, aW)
    if w[i] > aW:
        return without_i
    else:
        with_i = m[i] + max_val(w, m, i - 1, aW - w[i])
    return max(with_i, without_i)
```

```

def fast_max_val(w, v, i, aW, m):
    global num_calls
    num_calls += 1
    try:
        return m[(i, aW)]
    except KeyError:
        if i == 0:
            if w[i] <= aW:
                m[(i, aW)] = v[i]
                return v[i]
            else:
                m[(i, aW)] = 0
                return 0
        without_i = fast_max_val(w, v, i - 1, aW, m)
        if w[i] > aW:
            m[(i, aW)] = without_i
            return without_i
        else:
            with_i = v[i] + fast_max_val(w, v, i - 1, aW - w[i], m)
            res = max(with_i, without_i)
            m[(i, aW)] = res
            return res

```



# Zhrnutie

- optimalizácia kódu a jej ciele
- dynamické programovanie
- memoizácia
- teoréma optimálnej štruktúry
- rozhodovací strom v riešení algoritmizačných úloh