

Programování 2

12. cvičení, 6-05-2025

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na <https://github.com/PKvasnick/Programovani-2>.
2. Toto je **poslední** cvičení v tomto semestru, příští týden 13. května je rektorský den a 21. května si napíšeme zápočtový test.
 - Přijdete na cvičení v obvyklém termínu 17:20.
 - Dostanete jedinou programovací úlohu, kterou vyřešíte přímo na cvičení ve vymezeném čase cca. 75 minut.
 - Řešení nahrajete do ReCodExu a tam najdete i hodnocení.
 - Následující den najdete v repozitáři řešení.

3. Domácí úkoly

Dnes dostanete poslední domácí úkoly v tomto semestru.

K úkolům na tento týden promluvím podrobněji.

Dnešní program:

- Domácí úkoly
- Dynamické programování:
 - Baťoh
 - Nejdelší společný podřetězec

Domácí úkoly

Při prohlížení domácích úkolů na tento týden jsem dokola psal dvě věci:

- nevhodné datové struktury
- zbytečná duplicita kódu

Ukážeme si, jak se oběma vadám kódu vyhnout.

Cesta věže

Toto je malý problém, takže není vůbec potřebné moc se zamýšlet. Prozkoumáme všechny možné tahy věže, a na každém navštíveném poli necháme informaci o tom, kolika nejméně tahy se k němu lze dostat.

Jde o prohledávání do šířky: končíme, když máme vyplněna všechna dostupná pole, a vrátíme údaj ze zadaného cílového pole.

Abychom předešli zmnožování kódu pro různé směry pohybu věže, uděláte si funkci, která vrátí všechna pole, dostupná z dané pozice věže.

Používáme prioritní frontu: nejdřív se zabýváme poli, které vidíme poprvé.

```
# Find the shortest path of the rook on chessboard
from itertools import product
import heapq

SIZE = 8
INF = 10_000

@lambda cls: cls() # Create class instance immediately
class Chessboard:
    def __init__(self):
        """Just create chessboard"""
        self.chessboard = dict([(i,j),INF) for i, j in product(range(SIZE), range(SIZE))]
        self.start = None
        self.end = None

    def is_in_range(self, k, l):
        return (k,l) in self.chessboard.keys()

    def set_obstacle(self, i, j):
        self.chessboard[(i,j)] = -1

    def set_start(self, i, j):
        self.start = (i, j)
        self.chessboard[(i,j)] = 0

    def set_end(self, i, j):
        self.end = (i, j)

    def get_steps(self, i, j):
        return self.chessboard[(i,j)]

    def set_steps(self, i, j, steps):
        self.chessboard[(i,j)] = steps

    def rook_fields(self, i, j):
        """Return a list of fields controlled by a rook at (i, j)"""
        steps = [(1, 0), (0, 1), (-1, 0), (0, -1)]
        fields = []
        for s, t in steps:
            k = i + s
            l = j + t
            while self.is_in_range(k, l) and self.chessboard[k,l] != -1:
                fields.append((k, l))
                k = k + s
                l = l + t
        return fields
```

```

def print(self):
    chart = [["_"] for _ in range(SIZE)] for _ in range(SIZE)]
    for pos, steps in self.chessboard.items():
        i, j = pos
        if steps == -1:
            chart[i][j] = "x"
        elif steps == INF:
            chart[i][j] = "?"
        else:
            chart[i][j] = str(steps)
    for i in range(SIZE):
        print(*chart[i])
    print()

```

```

def read_chessboard():
    global Chessboard
    for i in range(SIZE):
        row = input().strip()
        for j in range(SIZE):
            if row[j] == ".":
                continue
            elif row[j] == "x":
                Chessboard.set_obstacle(i, j)
            elif row[j] == "v":
                Chessboard.set_start(i, j)
            elif row[j] == "c":
                Chessboard.set_end(i, j)
    return

```

```

def grade_chessboard():
    global Chessboard
    stack = []
    heapq.heappush(stack, (0, *Chessboard.start))
    while stack:
        steps, row, col = heapq.heappop(stack)
        for field in Chessboard.rook_fields(row, col):
            field_steps = Chessboard.get_steps(*field)
            if field_steps <= steps + 1:
                continue
            else:
                Chessboard.set_steps(*field, steps+1)
                heapq.heappush(stack, (steps+1, *field))
    Chessboard.print()
    return

```

```

def main():
    global Chessboard
    read_chessboard()
    grade_chessboard()

```

```

steps_to_end = Chessboard.get_steps(*Chessboard.end)
if steps_to_end == INF:
    print(-1)
else:
    print(steps_to_end)
return

if __name__ == '__main__':
    main()

```

Domino

Tady máme dvě osobitosti:

- symetrii kostek - kostku AB můžeme použít jako AB i BA. Je potřeba s ní rozumně naložit tak, abychom kód zbytečně nevětvali.
- datovou strukturu: Jaká struktura bude nejlépe vyhovovat pro získání všech kostek, které můžeme přidat na konec stávajícího řetězce?

Algoritmicky jde o prohledávání do hloubky.

```

sequence = []
max_sequence = []
box = [[0 for _ in range(7)] for _ in range(7)]

def print_box() -> None:
    for row in box:
        print(*row, sep = " ")
    print()

def parse_dominoes() -> tuple[int, int]:
    global box
    n, start = [int(s) for s in input().split()]
    dominoes = input().split()
    assert len(dominoes) == n, "Inconsistent parsing of dominoes"
    for d in dominoes:
        assert len(d) == 2, "Incorrect domino " + d
        i, j = [int(c) for c in d]
        box[i][j] += 1
        if i != j:
            box[j][i] += 1
    return n, start

def solve(start: int) -> None:
    global box, sequence, max_sequence
    for i in range(7):
        if box[start][i] > 0:
            box[start][i] -= 1
            if start != i:

```

```

        box[i][start] -= 1
        sequence.append(str(start) + str(i))
        if len(sequence) > len(max_sequence):
            max_sequence = sequence.copy()
        solve(i)
        sequence.pop()
        box[start][i] += 1
        if start != i:
            box[i][start] += 1
    return

def main() -> None:
    n, start = parse_dominoes()
    solve(start)
    print(len(max_sequence))
    print(*max_sequence)

if __name__ == "__main__":
    main()

```

Dynamické programování

Příklad 1: Baťoh

n položek s_i , $i = 1, 2, \dots, n$ s váhou w_i a cenou v_i . Najít podmnožinu položek tak, že

- jejich váha nepřesahuje W
- jejich celková cena je největší ze všech podmnožin splňujících předchozí podmínku.

Naivní řešení je najít mezi přípustnými podmnožinami tu s největší celkovou cenou.

Lze dobře řešit rekurzivně:

Maximum pro n -tou hodnotu:

- Maximum pro váhu W a $N-1$ položek (s vyloučením této hodnoty)
- Maximum pro váhu $W - w[n]$ a $N-1$ položek (se zařazením této hodnoty)

```

def knapSack(W, wt, val, n):

    # Base Case
    if n == 0 or W == 0:
        return 0











    # If weight of the nth item is more than Knapsack of capacity W,

```

```
# then this item cannot be included in the optimal solution
if (wt[n-1] > w):
    return knapSack(w, wt, val, n-1)

# return the maximum of two cases:
# (1) nth item included
# (2) not included
else:
    return max(
        val[n-1] + knapSack(
            w-wt[n-1], wt, val, n-1),
        knapSack(w, wt, val, n-1))
```

Dynamické programování: budujeme tabulku: bereme maximální hodnotu z konfigurace bez zařazeného objektu a s ním.

Weights (kg)				Knapsack capacities (kg)										
2	1	5	3	0	1	2	3	4	5	6	7	8	9	10
				0	0	0	0	0	0	0	0	0	0	0
				0	0	300	300	300	300	300	300	300	300	300
 				0	200	300	500	500	500	500	500	500	500	500
  				0	200	300	500	500	500	600	+ 700	900	900	900
   				0	200	300	500	700	800	1000	1000	1000	1100	=1200
300	200	400	500											
Values (\$)														

```
# Program for 0-1 Knapsack problem
# Returns the maximum value that can
# be put in a knapsack of capacity w
```

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]

    # Build table K[][] in bottom up manner
    for i in range(n + 1):
        for w in range(W + 1):
            if i == 0 or w == 0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]
                               + K[i-1][w-wt[i-1]],
                               K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
```

```

return K[n][w]

def main() -> None:
    profit = [60, 100, 120]
    weight = [10, 20, 30]
    w = 50
    print(knapsack(w, weight, profit, len(profit)))

if __name__ == '__main__':
    main()

```

Příklad 2: Nejdelší společný podřetězec

Důležitá úloha (diff, neukloetidové sekvence a pod.).

Vzpomeňte si na Levensteinovu vzdálenost. Opět můžeme řešit rekurzivně, ale my si ukážeme DP řešení - budeme vyplňovat tabulku.

- pokud jsou písmena stejná, přidáme 1
- pokud se liší, použijeme maximum z řešení pro vynechání znaku z 1. posloupnosti a pro vynechání z 2. posloupnosti.

		<div>A Y Z X</div>				
		0	1	2	3	4
A X Y T	0	0	0	0	0	0
	1	0	1	1	1	1
	2	0	1	1	1	2
	3	0	1	2	2	2
	4	0	1	2	2	2

```

datasets = [
    ["AGGTAB", "GXTXAYB"],
    ["ABCDGH", "AEDFHR"],
    ["GAATTCAGTTA", "GGATCGA"]
]

def print_matrix(d: list[list[int]]) -> None:
    for row in range(len(d)):

```

```

        print(*d[row])
    print()

def lcs(s1:str, s2:str) -> str:
    d = [[0 for _ in range(len(s2)+1)] for _ in range(len(s1)+1)]
    for row in range(1, len(s1)+1):
        c = s1[row-1]
        for col in range(1, len(s2)+1):
            if s2[col-1] != c:
                d[row][col] = max(d[row][col-1], d[row-1][col])
            else:
                d[row][col] = d[row-1][col-1] + 1
    print_matrix(d)
    return

def main() -> None:
    global datasets
    s1, s2 = datasets[2]
    subs = lcs(s1, s2)
    print("".join(subs))

if __name__ == "__main__":
    main()

```

Domácí úkoly

1. **Bipartitní graf:** zjistěte, zda je možné obarvit vrcholy grafu dvěma barvami tak, aby žádné dva sousedící vrcholy neměli stejnou barvu.
2. **Lidské kruhy :** N lidí se náhodně pochyťá za ruce. Máte zjistit, kolik kruhů vytvoří.