Programování 2

13. cvičení, 11-05-2023

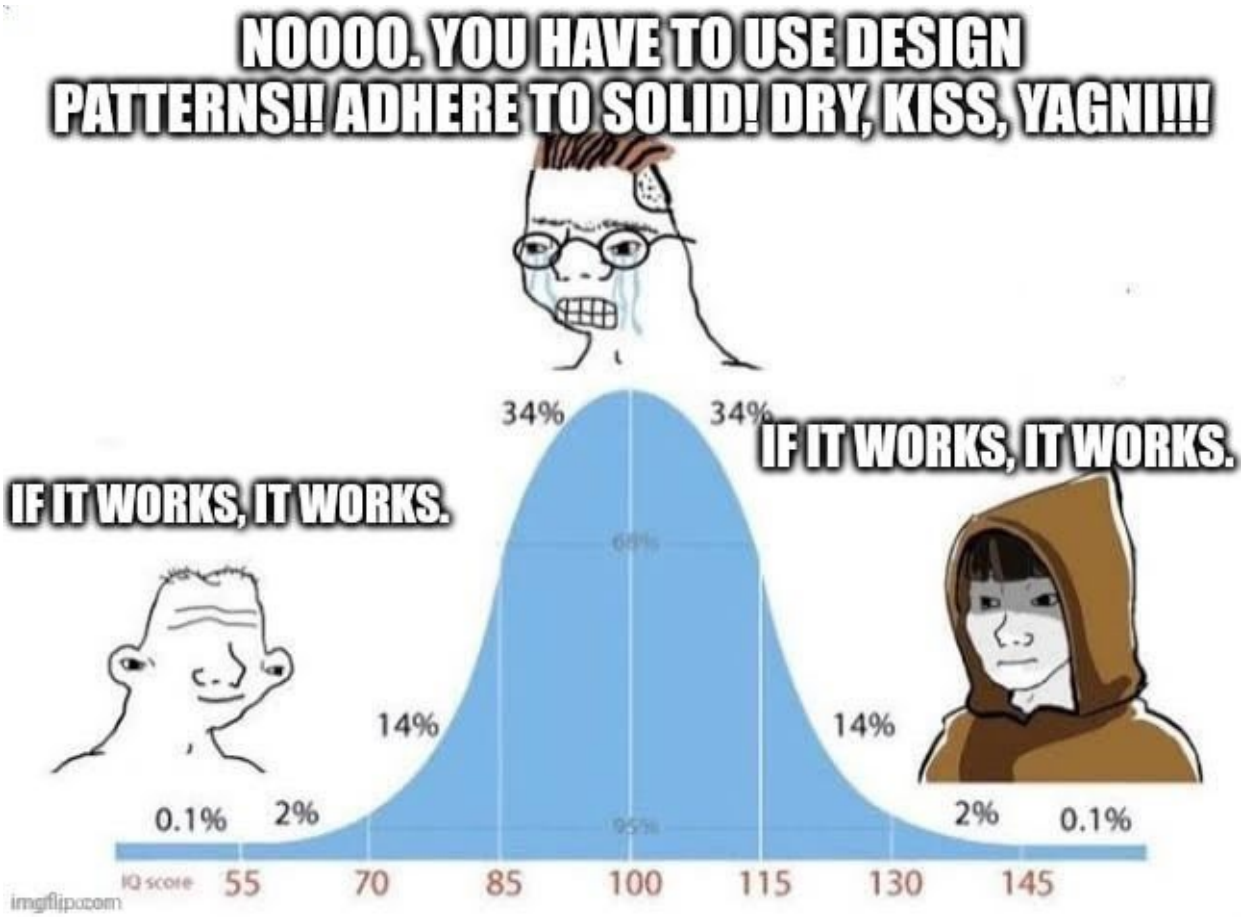
tags: `Programovani 2`, `ctvrtek 1` `ctvrtek 2`

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 18. května si napíšeme zápočtový test:
    - Přijdete na cvičení v obvyklém termínu 10:40, resp. 15:40
    - Dostanete jedinou programovací úlohu, kterou vyřešíte přímo na cvičení ve vymezeném čase 75 minut.
    - Řešení nahrajete do ReCodExu a tam najdete i hodnocení.
    - Následující den najdete v repozitáři řešení.
3. **Domácí úkoly**
    - Cesta věže
    - Barvení hran grafu (*bipartite graph*)
    - Následující permutace
4. - 18. 5.

Dnešní program:

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



Hlavně je dobře psát jasný kód.

Práce v souborovém systému: `pathlib`

Třída `Path`: adresa objektu v souborovém systému.

```
from pathlib import Path

Path("main.py").exists()
Out[3]: True

Path("img").mkdir()          # Nový adresář

Path("img").mkdir()          # Už existuje - chyba
Traceback (most recent call last):
  File "C:\ProgramData\Anaconda3\lib\site-packages\IPython\core\interactiveshell.py", line 3369, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-5-c6fb86db84a7>", line 1, in <cell line: 1>
    Path("img").mkdir()
  File "C:\ProgramData\Anaconda3\lib\pathlib.py", line 1323, in mkdir
    self._accessor.mkdir(self, mode)
FileExistsError: [WinError 183] Cannot create a file when that file already exists: 'img'

Path("img").mkdir(exist_ok=True) # nedojde k chybě
```

```
19
```

Soubory můžeme také přesouvat. `pathlib` rozumí, v jakém operačním sytému pracuje:

```
file = Path("sk.py").replace("img/sk.py")
file
Out[8]: WindowsPath('img/sk.py')

```

`.parent`, `.name` a další:

```
Path("img").parent
Out[10]: WindowsPath('.')
Path("img").parent.parent
Out[11]: WindowsPath('.')
```

V jakém adresáři běží můj skript?

```
from pathlib import Path

folder = Path("__file__").parent
print(folder)
```

Domácí úkoly

Následující permutace

Máme permutaci a máme najít lexikograficky následující permutaci.

Asi by to nebylo moc efektivní, ale takto můžeme v principu generovat všechny permutace.

- vypsat si permutace malých množin
- najít a zobecnit pravidlo

```
def main() -> None:
    n = int(input())
    permutation = [int(s) for s in input().split()]
    if len(permutation) == 1:
        print("NEEXISTUJE")
        return
    for i in range(len(permutation) - 1, 0, -1):
        if permutation[i] < permutation[i-1]:
            continue
        else:
            break
    else:
        print("NEEXISTUJE")
        return
```

```
    pivot = i-1
    first = i
    i = i + 1
    while i < len(permutation):
        if permutation[pivot] < permutation[i] < permutation[first]:
            first = i
        i += 1
    permutation[pivot], permutation[first] = permutation[first],
permutation[pivot]
    permutation = [*permutation[:pivot+1], *sorted(permutation[pivot+1:])]
    print(*permutation)


if __name__ == "__main__":
    main()
```

Bipartite graph

Toto je celkem lehká úloha na procházení grafu: Pro každý uzel děláme toto:

- natřeme ho aktuální barvou a přepneme aktuální barvu
- pro všechny děti:
  - zkontrolujeme, zda je dítě již zabarveno. Pokud ano, zda barva sedí.
  - Pokud barva nesedí, graf nelze vymalovat.
  - Pokud uzel zatím není zabarvený, zabarvíme a šup s ním do zásobníku nebo pro něj rekurzivně voláme barvící funkci.

```
import sys
from collections import defaultdict


def red_green():
    n_vertices = int(input())
    n_roads = int(input())
    graph = defaultdict(list)
    colors = [-1] * (n_vertices+1)
    for _ in range(n_roads):
        start, end = [int(s) for s in input().split()]
        graph[start].append(end)
        graph[end].append(start)

    stack = [1]
    colors[1] = 1
    while stack:
        node = stack.pop()
        color = colors[node]
        for neighbour in graph[node]:
            if colors[neighbour] == -1:
```

```
                colors[neighbour] = (not color)
                stack.append(neighbour)
            elif colors[neighbour] == color:
                print("Nelze")
                return
    if -1 in colors[1:]:
        print("Nelze")
        return
    print(*[i for i in range(1, n_vertices+1) if colors[i] == 1])
    print(*[i for i in range(1, n_vertices+1) if colors[i] == 0])
    return


if __name__ == "__main__":
    red_green()
```

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.

- Používáme prioritní frontu: nejdřív se zabýváme poli, které vidíme poprvé.
- Jinak máme dost podobnou implementaci jako u problému 8 dam.

```
# 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 queen 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()
```

Dynamické programování

Příklad 1: Baťoh

n položek s váhou w a cenou v. Najít položky s největší cenou, kterých váha nepřesahuje W.

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))

```

```
# A Dynamic Programming based Python
# 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)))

```

```
30
31 if __name__ == '__main__':
32     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.

```
1  datasets = [
2      ["AGGTAB", 'GXTXAYB'],
3      ['ABCDGH', 'AEDFHR'],
4      ["GAATTCAGTTA","GGATCGA"]
5  ]
6
7
8  def print_matrix(d: list[list[int]]) -> None:
9      for row in range(len(d)):
10         print(*d[row])
11     print()
12
13
14 def lcs(s1:str, s2:str) -> str:
15     d = [[0 for _ in range(len(s2)+1)] for _ in range(len(s1)+1)]
16     for row in range(1,len(s1)+1):
17         c = s1[row-1]
18         for col in range(1, len(s2)+1):
19             if s2[col-1] != c:
20                     d[row][col] = max(d[row][col-1], d[row-1][col])
21             else:
22                 d[row][col] = d[row-1][col-1] + 1
23         print_matrix(d)
24     return
25
26
27 def main() -> None:
28     global datasets
29     s1, s2 = datasets[2]
30     subs = lcs(s1, s2)
31     print("".join(subs))
32
33
34 if __name__ == "__main__":
35     main()
36
```