4b) Uvažme orientovaný graf $G = (V, E)$, kde V je množina vrcholů (číslovaných přirozenými čísly od 0) a E je množina hran. Dále uvažme konečnou množinu barev K a zobrazení $c : E \to K$.

Pro libovolné $v_s, v_e \in V$ definujme predikát P: $P(v_s, v_e) = \texttt{true} \iff \exists k \in K :$ existuje cesta

$$v_0, e_0, v_1, \ldots, v_{n-1}, e_{n-1}, v_n : v_s = v_0 \wedge v_e = v_n \wedge \forall\, 0 \leq i < n : e_i = (v_i, v_{i+1}) \wedge e_i \in E \wedge c(e_i) = k.$$

Uvažme algoritmus $cPath(G, v_s, v_e)$, který má na vstupu graf G a jeho dva libovolné vrcholy v_s a v_e. $cPath(G, v_s, v_e)$ vrací $\texttt{true} \iff P(v_s, v_e) = \texttt{true}$

i) Analyzujte asymptotickou časovou složitost algoritmu $cPath(G, v_s, v_e)$ v nejhorším případě.

ii) Analyzujte asymptotickou časovou složitost algoritmu $cPath(G, v_s, v_e)$ v nejlepším případě. O(1)

iii) Navrhněte algoritmus $cPath^+(G, v_s, v_e)$, který řeší stejný problém a má lepší asymptotickou časovou složitost v nejhorším případě. Analyzujte tuto složitost.

```
1  Function cPath(G, v_s, v_e)
2      for int n := 1; n < |V|; n := n + 1 do
3          foreach cestu v_0, e_0, v_1, ..., v_{n-1}, e_{n-1}, v_n v G do
4              if v_0 ≠ v_s ∨ v_n ≠ v_e then continue
5              color := c(e_0)
6              res := 1
7              for int i := 1; i < n; i := i + 1 do
8                  if c(e_i) ≠ color then res := 0; break
9              if res = 1 then return true
10     return false
```

pro každou barvu DFS/BFS... O(|E|)

$\sum_{n=1}^{|V|-1} |V|^{n+1} \cdot O(n) \leq |V| \cdot |V|^{|V|+1} \cdot O(|V|) = O(|V|^{|V|+3})$

(60 bodů)

Poznámka 1: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1.

Poznámka 2: Časovou složitost analyzujte vzhledem k velikosti množin V a E.

Poznámka 3: V bodě iii) můžete předpokládat, že G je efektivně representován pomocí seznamu následníků Adj, kde $Adj(v, c)$ vrací v konstantním čase množinu hran s barvou c, které vycházejí z vrcholu v.

Uvažte datový typ `digit` $= \{0, \ldots, 9\}$ a operaci `inc(digit x[], int size)`, která inkrementuje číslo zapsané pomocí pole prvků typu `digit` o velikosti `size` (indexované od 0 do `size` -1) s implementací danou níže.

Algorithm 1: `inc(digit x[], int size)`

```
1 assert(size > 0);
2 i ← size;
3 do
4 |   i ← i − 1;
5 |   x[i] ← (x[i] + 1) % 10;
6 while x[i] = 0 ∧ i > 0;
```

Uvažte posloupnost n volání operace `inc` počínaje polem `x` $= [0, \ldots, 0]$.

1. Jaká je asymptotická složitost této posloupnosti operací při použití analýzy nejhoršího případu?
2. Jaká je amortizovaná složitost této posloupnosti operací?

Poznámka: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1 (řádek 3 má složitost 0).

V nejhorším případě: $O(size)$ ← jedna operace

n - operací : $O(n \cdot size)$

Průměrně potřebujeme: $5 + \frac{1}{10} \cdot 3 + \frac{1}{100} \cdot 3 + \frac{1}{1000} \cdot 3 + \ldots$

≤ 1

~~$5 + \frac{1}{2} 3 \cdot \frac{1}{x}$~~

6 kroků

inc | 6 kreditů

Průměrná složitost je $\frac{6 \cdot n}{1} \in O(n)$

Příklad 4

0 bodů

min. 0 bodů

Nechť $G = (V, E)$ je orientovaný graf, kde V je množina vrcholů a E je množina hran. E je reprezentovaná pomocí seznamu následníků *Succ* a seznamu předchůdců *Pred*, tj. $Succ(v)$ vrací v konstantním čase množinu následníků vrcholu v a $Pred(v)$ vrací v konstantním čase množinu předchůdců vrcholu v.

Dále uvažme níže popsaný algoritmus $allReach(G, v_t)$, který pro $v_t \in G$ vrací množinu vrcholů $V' \subseteq V$, která obsahuje všechny vrcholy v', pro které existuje v G cesta z v' do v_t.

```
1  Function allReach(G, v_t)
2      reach := ∅
3      foreach v' ∈ V do
4          foreach v ∈ V do
5              v.visited := false
6          v'.visited := true
7          Q := emptyQueue()
8          Q.enqueue(v')
9          while Q.notEmpty() do
10             v := Q.dequeue()
11             if v == v_t then
12                 reach := reach ∪ {v'}
13                 break
14             foreach u ∈ Succ(v) do
15                 if u.visited = false then
16                     u.visited := true
17                     Q.enqueue(u)
18     return reach
```

i) Analyzujte asymptotickou časovou složitost algoritmu $allReach(G, v_t)$ v nejhorším případě.

ii) Analyzujte asymptotickou časovou složitost algoritmu $allReach(G, v_t)$ v nejlepším případě.

iii) Navrhněte algoritmus $allReach^+(G, v_t)$, který řeší stejný problém a má lepší asymptotickou časovou složitost v nejhorším případě.

iii) Analyzujte asymptotickou časovou složitost algoritmu $allReach^+(G, v_t)$ v nejhorším případě.

Poznámka 1: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1.

Poznámka 2: Časovou složitost analyzujte vzhledem k velikosti množin V a E.

Cyklus 9 - 17 běží $|V|$ krát

14 - 17 běží $|V|$ krát

$$O(|V|^2 + |V|) \cdot |V| = O(|V|^3)$$

LEPŠÍ ODHAD

9 - 17 se dá odhadnout lépe $|E|$ krát bez ohledu na počet uzlů

$$O(|V| + |E| + |V|) \cdot |V| \leq O((|E| + |V|) \cdot |V|)$$

↑ pro souvislý graf, kde $|E| \geq |V|$

V nejlepším případě:

3-17: $|V|$ krát
4-6: $|V|$ krát
7.-17: $O(1)$ krát
14.-17: $O(1)$ krát

$$O(|V| \cdot (|V| + 1)) = O(|V|^2)$$

iii) Prohledávání do hloubky s využitím „pred" namísto „succ":
spuštění z v_t. $O(|E|)$ v nejhorším případě.

Uvažme datovou strukturu, která implementuje frontu (tj. níže popsané operace *enqueue*(x) a *dequeue*()) pomocí dvou zásobníků *head* a *tail*.

Analyzujte a zdůvodněte amortizovanou časovou složitost libovolné posloupnosti n operací *enqueue*(x) a *dequeue*().

Poznámka: Předpokládejte uniformní cenové kritérium, kde složitost každého řádku programu je 1.

```
1  global(stack head, tail)
2  Function enqueue(x)
3  |  tail.push(x)
4  Function dequeue()
5  |  if head.empty() then
6  |  |  while ¬tail.empty() do
7  |  |  |  x := tail.pop()
8  |  |  |  head.push(x)
9  |  x := head.pop()
10 |  return x
```

Předpokládejme, že začínáme s prázdnou frontou.

	cena	kredity
enqueue	1	1+3
dequeue	3+n·3	3

Cyklus 6.–8. proběhne maximálně 1× pro každý prvek vložený enqueue

Příklad 4

150 bodů

min. 40 bodů

$$P(A, B, n) = true \Leftrightarrow \exists i \in \{1, \ldots, n\} : \forall j \in \{1, \ldots, n\} : A[i] \geq B[j]$$

Níže uvedený algoritmus *dominate*(A, B, n) řeší tento problém.

(a) Analyzujte asymptotickou časovou složitost algoritmu *dominate* v nejhorším případě.

(b) Analyzujte asymptotickou časovou složitost algoritmu *dominate* v nejlepším případě.

(c) Navrhněte algoritmus *dominate*$^+(A, B, n)$, který řeší tento problém a má lepší asymptotickou časovou složitost v nejhorším případě než algoritmu *dominate*. Analyzujte asymptotickou časovou složitost algoritmu *dominate*$^+$ v nejhorším případě.

```
1  Function dominate(A, B, n)
2      found := false
3      i := 1
4      while i ≤ n ∧ found = false do
5          found := true
6          j := 1
7          while j ≤ n ∧ found = true do
8              if A[i] < B[j] then
9                  found := false
10             j := j + 1
11         i := i + 1
12     return found
```

a) V nejhorším případě $O(n^2)$

b) V nejlepším případě $O(n)$

c) najít max pole A v $O(n)$ krocích ~ porovnat toto max s prvky pole B. $O(n)$ krocích. Celkově $O(n+n) = O(n)$