Cvičení 11

Příklad 1: Vezměme si následující Algoritmus 1. Vstupem tohoto algoritmu může být libovolné přirozené číslo n.

Algoritmus 1:

```
PrintSeq(n):
    print n
    while n > 1 do
        if n mod 2 = 0 then
            n := n / 2
        else
            n := 3 * n + 1
        print n
```

a) Nakreslete graf řídícího toku tohoto algoritmu.

b) Popište výpočet, který tento algoritmus provede, pokud jako vstup dostane číslo 5. Vypište posloupnost jednotlivých konfigurací při tomto výpočtu.

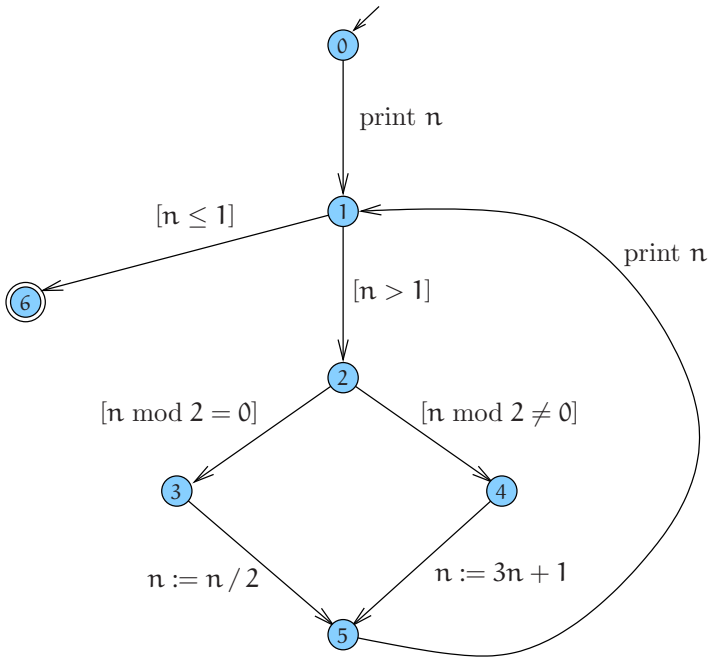
c) Kolik kroků provede tento algoritmus, když jako vstup dostane číslo 7 ? Co bude výstupem?

Poznámka: Předpokládejte, že hodnoty proměnné n mohou být libovolná (neomezeně velká) přirozená čísla.

Řešení:

a) Graf řídícího toku je uveden na Obrázku 1.

b) Výpočet vypadá následovně



Obrázek 1: Graf řídícího toku

Krok	Stav	n
0	0	5
1	1	5
2	2	5
3	4	5
4	5	16
5	1	16
6	2	16
7	3	16
8	5	8
9	1	8
10	2	8
11	3	8
12	5	4
13	1	4
14	2	4
15	3	4
16	5	2
17	1	2
18	2	2
19	3	2
20	5	1
21	1	1
22	6	1

c) Výstupem bude posloupnost $7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

Celkový počet kroků se určí následovně: Cyklem se projde celkem 16 krát. Při každém průchodu cyklem se provedou 4 kroky. Na začátku se provede mimo cyklus jeden krok a konci také jeden krok. Celkem se tedy provede $1 + 16 \cdot 4 + 1 = 66$ kroků.

Příklad 2:

a) Navrhněte a popište pseudokódem algoritmus pro řešení následujícího problému:

VSTUP: Přirozené číslo n.
OTÁZKA: Je n prvočíslo?

- Nakreslete graf řídícího toku vámi navrženého algoritmu.
- Odsimulujte činnost tohoto algoritmu pro některé vstupy (např. $n = 0$, $n = 1$, $n = 2$, $n = 3$, $n = 4$, $n = 15$, $n = 16$, apod.). Určete, kolik kroků váš algoritmus pro tyto vstupy provede.

b) Navrhněte a popište pseudokódem algoritmus pro řešení následujícího problému (jedná se o problém rozkladu přirozeného čísla na prvočísla):

VSTUP: Přirozené číslo n, kde $n > 1$.
VÝSTUP: Prvočísla $p_1, p_2, \ldots, p_k$ taková, že $p_1 \cdot p_2 \cdot \ldots \cdot p_k = n$.

Poznámka: Algoritmus může vzniknout vhodnou modifikací a rozšířením algoritmu navrženého v předchozím bodě nebo případně můžete tento algoritmus použít jako podprogram.

Řešení:

a) Přímočaré řešení je Algoritmus 2, o něco efektivnější řešení je Algoritmus 3.

Algoritmus 2: Testování prvočíselnosti — verze 1

```
PRIME(n):
    if n ≤ 1 then
        return FALSE
    for i := 2 to n − 1 do
        if n mod i = 0 then
            return FALSE
    return TRUE
```

b) Například to jde udělat pomocí Algoritmu 4.

Příklad 3: Níže uvedený Algoritmus 5 by měl řešit následující problém:

Algoritmus 3: Testování prvočíselnosti — verze 2

```
PRIME(n):
    if n ≤ 1 then
        return FALSE
    else if n = 2 then
        return TRUE
    else if n mod 2 = 0 then
        return FALSE
    else
        i := 3
        while i * i ≤ n do
            if n mod i = 0 then
                return FALSE
            i := i + 2
        return TRUE
```

Algoritmus 4: Rozklad na prvočísla

```
PRIME-FACTORIZATION(n):
    i := 2
    while i ≤ n do
        while n mod i = 0 do
            print i
            n := n / i
        i := i + 1
```

Vstup: Přirozené číslo n.

Výstup: Hodnota $n!$ (tj. faktoriál čísla n).

Připomeňme, že funkce faktoriál je definována následovně:

- $0! = 1$,
- $n! = (n-1)! \cdot n$ pro $n \geq 1$.

Pro jednoduchost předpokládejte, že hodnotami proměnných mohou být libovolná (neomezeně velká) přirozená čísla.

Algoritmus 5: Výpočet faktoriálu

```
FACTORIAL(n):
    x := 1
    for i := 2 to n do
        x := x * i
    return x
```

a) Nakreslete graf řídícího toku tohoto algoritmu.

b) Popište výpočet, který tento algoritmus provede, pokud jako vstup dostane číslo 5. Vypište posloupnost jednotlivých konfigurací při tomto výpočtu.

c) Nyní je cílem ukázat, že daný algoritmus je korektní, tj. pro každý vstup se po konečném počtu kroků zastaví a vydá správný výsledek.

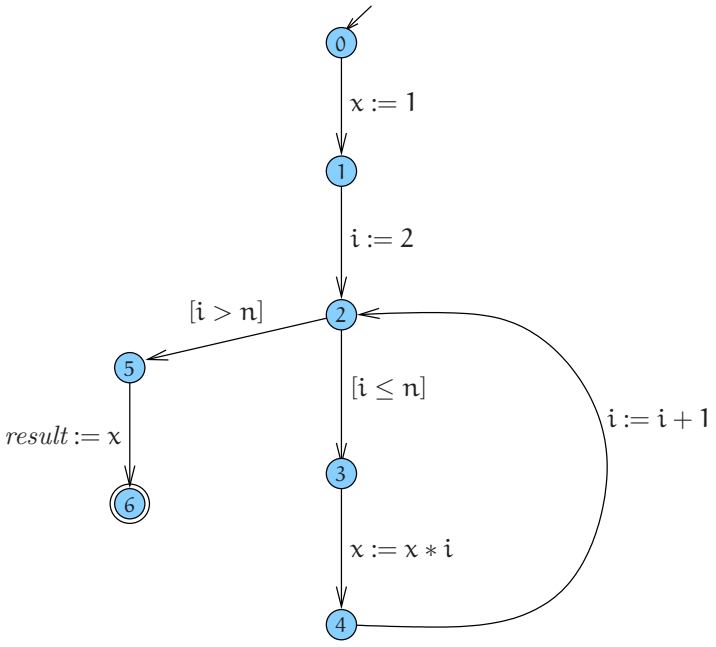
V případě Algoritmu 5 je výhodné analýzu korektnosti rozdělit na dvě části — na analýzu případu, kdy $n = 0$, a na analýzu případů, kdy $n \geq 1$.

- Ukažte, že algoritmus korektně pracuje pro vstup, kde $n = 0$. Zde stačí odsimulovat výpočet daného algoritmu pro tento vstup (a zkontrolovat, že se zastaví a jeho výstup odpovídá očekávanému výsledku).

Dále tedy předpokládejte, že pro hodnotu na vstupu platí $n \geq 1$ (tj. následující body řešte s tímto dodatečným předpokladem):

- Zformulujte hypotézy ohledně toho, jaké invarianty platí v jednotlivých místech v kódu (tj. v jednotlivých vrcholech grafu řídícího toku). Snažte se navrhnout takové invarianty, aby se pomocí nich dala zdůvodnit korektnost výše uvedeného algoritmu.
- Ověřte, že invarianty navržené v předchozím bodě opravdu platí.
- S využitím těchto invariantů zdůvodněte, že platí, že pokud výpočet algoritmu skončí, tak algoritmus vrátí správný výsledek.
- Ukažte, že pro libovolný vstup platí, že výpočet výše uvedeného algoritmu skončí po konečném počtu kroků.

Řešení:



Obrázek 2: Graf řídícího toku funkce FACTORIAL

a) Graf řídícího toku je na Obrázku 2.

b) Výpočet bude vypadat takto:

Krok	Stav	n	i	x	*result*
0	0	5	?	?	?
1	1	5	?	1	?
2	2	5	2	1	?
3	3	5	2	1	?
4	4	5	2	2	?
5	2	5	3	2	?
6	3	5	3	2	?
7	4	5	3	6	?
8	2	5	4	6	?
9	3	5	4	6	?
10	4	5	4	24	?
11	2	5	5	24	?
12	3	5	5	24	?
13	4	5	5	120	?
14	2	5	6	120	?
15	5	5	6	120	?
16	6	5	6	120	120

c) Případ, kdy $n = 0$, je triviální — algoritmus vrací výsledek 1, což je v pořádku. Dále se tedy předpokládá, že $n \geq 1$.

Invarianty, které platí v jednotlivých stavech:

- Stav 0: —
- Stav 1: $x = 1$
- Stav 2: $2 \leq i \leq n+1$, $x = (i-1)!$
- Stav 3: $2 \leq i \leq n$, $x = (i-1)!$
- Stav 4: $2 \leq i \leq n$, $x = i!$
- Stav 5: $i = n+1$, $x = n!$
- Stav 6: $i = n+1$, $x = n!$

To, že se algoritmus zastaví, plyne z toho, že se s každou iterací cyklu snižuje hodnota $(n+1)-i$. Tato hodnota nikdy neklesne pod nulu, takže se nemůže snižovat donekonečna.

Poznámka — formální důkaz by mohl vypadat tak, že se řídícím stavům přiřadí následující vektory:

- Stav 0: (4)
- Stav 1: (3)
- Stav 2: $(2, n+1-i, 2)$
- Stav 3: $(2, n+1-i, 1)$
- Stav 4: $(2, n+1-i, 0)$
- Stav 5: (1)
- Stav 6: (0)

Snadno se ověří, že se s každým krokem jde do konfigurace, které je přiřazen lexikograficky menší vektor.

Příklad 4: Níže uvedený Algoritmus 6 by měl sloužit k nalezení prvku v setříděném poli. Jedná se o jednu možnou variantu algoritmu pro binární vyhledávání (metodou půlení intervalu). Z hlediska této úlohy není podstatné, jakého konkrétního typu jsou prvky tohoto pole, pro jednoduchost můžeme předpokládat, že jsou to celá čísla. Prvky pole jsou indexovány od nuly, tj. pokud pole A má n prvků, jedná se o prvky $A[0], A[1], \ldots, A[n-1]$.
Algoritmus by měl řešit následující problém:

> Vstup: Hledaná hodnota x, pole A o n prvcích (kde $n \geq 0$), jehož prvky jsou setříděny od nejmenšího po největší, tj. pro všechna přirozená čísla i a j taková, že $0 \leq i < j < n$, platí $A[i] \leq A[j]$.
>
> Výstup: Přirozené číslo i udávající index prvního výskytu hodnoty x v poli A nebo speciální hodnota NotFound v případě, kdy se hodnota x v poli A nenachází.

Poznámka: Pro jednoduchost předpokládejte, že hodnoty proměnných mohou být libovolně velká celá čísla.

a) Nakreslete graf řídícího toku tohoto algoritmu.

b) Vypište posloupnost konfigurací ve výpočtu, kde vstupem jsou hodnoty $x = 6$, $A = [1, 3, 3, 4, 6, 6, 8, 9, 10, 10, 12, 13]$ a $n = 12$.

Algoritmus 6: Binární vyhledávání

```
1  BSearch(x, A, n):
2      ℓ := 0
3      r := n
4      while ℓ < r do
5          k := ⌊(ℓ + r) / 2⌋
6          if A[k] < x then
7              ℓ := k + 1
8          else
9              r := k
10     if ℓ < n and A[ℓ] = x then
11         return ℓ
12     return NotFound
```

c) Řekněme, že bychom v algoritmu provedli následující změny (vždy jen jednu z těchto změn). Pro každou z těchto změn najděte příklad vstupu, při kterém algoritmus (s touto změnou) nepracuje korektně (např. se nezastaví, přistupuje k prvkům pole mimo povolený rozsah, vrací chybný výstup, apod.).

   (a) Na řádku 4 změnit podmínku $\ell < r$ na $\ell \leq r$.

   (b) Na řádku 7 změnit přiřazení $\ell := k + 1$ na $\ell := k$.

   (c) Na řádku 9 změnit přiřazení $r := k$ na $r := k - 1$.

   (d) Na řádku 9 změnit přiřazení $r := k$ na $r := k + 1$.

   (e) Na řádku 5 změnit přiřazení $k := \lfloor (\ell + r) / 2 \rfloor$ na $k := \lceil (\ell + r) / 2 \rceil$ (resp. na $k := \lfloor (\ell + r + 1) / 2 \rfloor$).

d) Navrhněte vhodné invarianty, které podle vás platí v jednotlivých vrcholech grafu řídícího toku.

   Nápověda: Před provedením testu $\ell < r$ na řádku 4 by mělo platit následující:

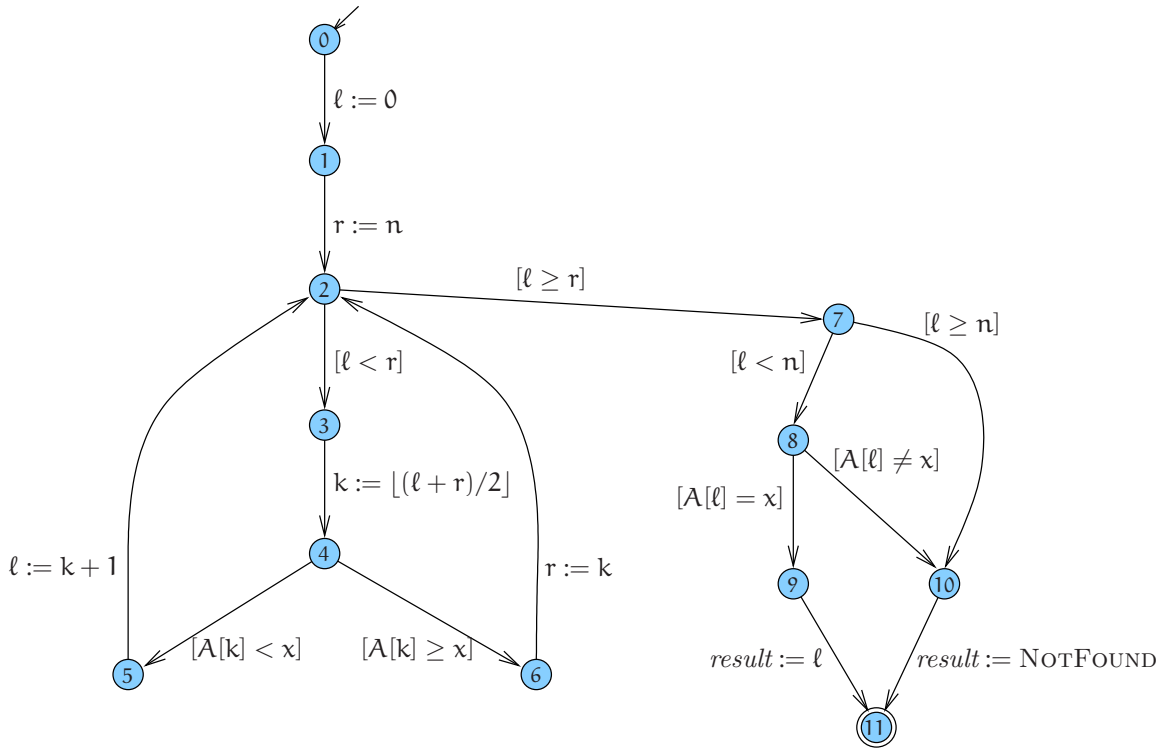
   - $0 \leq \ell \leq r < n$,
   - pro každé i takové, že $0 \leq i < \ell$, je $A[i] < x$,
   - pro každé i takové, že $r \leq i < n$, je $A[i] \geq x$.

e) Ověřte, že invarianty navržené v předchozím bodě opravdu platí.

f) Zjistěte, zda se algoritmus pro každý vstup zastaví. Pokud ano, dokažte to, pokud ne, uveďte příklad vstupu, pro který se výpočet algoritmu nikdy nezastaví.

g) Na základě předchozí analýzy buď zdůvodněte, že je výše uvedený algoritmus korektní, nebo uveďte příklad vstupu, pro který se nechová korektně.

h) Řekněme, že bychom měli implementaci tohoto algoritmu, kde by pro hodnoty proměnných n, ℓ, r a k byla použita 32-bitová celá čísla se znaménkem (tj. čísla, jejichž hodnoty mohou



Obrázek 3: Graf řídícího toku funkce BSearch

být v rozsahu $-2^{31}, \ldots, 2^{31} - 1$) a i veškeré aritmetické operace s těmito proměnnými by byly prováděny na tomto datovém typu. Bude algoritmus, tak jak byl popsán, správně fungovat pro všechny vstupy, kde $n < 2^{31}$?

Řešení:

a) Graf řídícího toku je na Obrázku 3.

b) Výpočet vypadá následovně (ve všech konfiguracích je $A = [1, 3, 3, 4, 6, 6, 8, 9, 10, 10, 12, 13]$, $n = 12$ a $x = 6$, takže tyto hodnoty jsou pro přehlednost z tabulky vypuštěny):

Krok	Stav	ℓ	r	k	*result*
0	0	?	?	?	?
1	1	0	?	?	?
2	2	0	12	?	?
3	3	0	12	?	?
4	4	0	12	6	?
5	6	0	12	6	?
6	2	0	6	6	?
7	3	0	6	6	?
8	4	0	6	3	?
9	5	0	6	3	?
10	2	4	6	3	?
11	3	4	6	3	?
12	4	4	6	5	?
13	6	4	6	5	?
14	2	4	5	5	?
15	3	4	5	5	?
16	4	4	5	4	?
17	6	4	5	4	?
18	2	4	4	4	?
19	7	4	4	4	?
20	8	4	4	4	?
21	9	4	4	4	?
22	11	4	4	4	4

c) (a) Na řádku 4 změnit podmínku $\ell < r$ na $\ell \leq r$:
- $x = 2$, $A = [0, 1]$, $n = 2$ — bude přistupovat k prvku $A[2]$
- $x = 1$, $A = [0, 1]$, $n = 2$ — nezastaví se

(b) Na řádku 7 změnit přiřazení $\ell := k + 1$ na $\ell := k$.
- $x = 2$, $A = [0, 1]$, $n = 2$ — nezastaví se
- $x = 1$, $A = [0, 1]$, $n = 2$ — nezastaví se

(c) Na řádku 9 změnit přiřazení $r := k$ na $r := k - 1$.
- $x = 1$, $A = [0, 1]$, $n = 2$ — vrátí NotFound místo 1

(d) Na řádku 9 změnit přiřazení $r := k$ na $r := k + 1$.
- $x = 1$, $A = [0, 1]$, $n = 2$ — nezastaví se

(e) Na řádku 5 změnit přiřazení $k := \lfloor (\ell + r) / 2 \rfloor$ na $k := \lceil (\ell + r) / 2 \rceil$ (resp. na $k := \lfloor (\ell + r + 1) / 2 \rfloor$).
- $x = 1$, $A = [0, 1]$, $n = 2$ — nezastaví se
- $x = 0$, $A = [0, 1]$, $n = 2$ — nezastaví se

d) V následujícím popisu nabývají proměnné ℓ, r, k a i jako hodnoty celá čísla, tj. hodnoty z množiny $\mathbb{Z}$. Při použití kvantifikátorů se vždy kvantifikuje přes množinu $\mathbb{Z}$, pro přehlednost to ale není uvedeno, aby byl zápis stručnější. Invarianty $n \geq 0$ a $\forall i \forall j (0 \leq i < j < n \rightarrow A[i] \leq A[j])$, které platí ve všech stavech, jsou rovněž u jednotlivých stavů pro přehlednost vypuštěny.

Invarianty, které platí v jednotlivých stavech:

- Stav 0: —
- Stav 1: $\ell = 0$
- Stav 2: $0 \le \ell \le r \le n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(r \le i < n \ \to\ A[i] \ge x)$
- Stav 3: $0 \le \ell < r \le n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(r \le i < n \ \to\ A[i] \ge x)$
- Stav 4: $0 \le \ell < r \le n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(r \le i < n \ \to\ A[i] \ge x),$ $k = \lfloor(\ell + r)/2\rfloor$
  Z $\ell < r$ a $k = \lfloor(\ell + r)/2\rfloor$ navíc vyplývá $\ell \le k < r$. (Stačí si uvědomit, že pokud $k = \lfloor(\ell + r)/2\rfloor$, tak $2k \le \ell + r < 2k + 2$. Protože $\ell < r$, tak $r - \ell > 0$.
  Protože $2k \le \ell + r$, platí $2k < (\ell + r) + (r - \ell) = 2r$, z čehož vyplývá, že $k < r$.
  Podobně z $\ell + r < 2k + 2$ plyne $2\ell = (\ell + r) - (r - \ell) < 2k + 2$, a tedy $\ell < k + 1$, což platí právě tehdy, když $\ell \le k$.)
- Stav 5: $0 \le \ell < r \le n,\ \forall i(0 \le i \le k \ \to\ A[i] < x),\ \forall i(r \le i < n \ \to\ A[i] \ge x),$ $k = \lfloor(\ell + r)/2\rfloor$, $\ell \le k < r$
- Stav 6: $0 \le \ell < r \le n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(k \le i < n \ \to\ A[i] \ge x),$ $k = \lfloor(\ell + r)/2\rfloor$, $\ell \le k < r$
- Stav 7: $0 \le \ell \le n,\ \ell = r,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell \le i < n \ \to\ A[i] \ge x)$
- Stav 8: $0 \le \ell < n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell \le i < n \ \to\ A[i] \ge x)$
- Stav 9: $0 \le \ell < n,\ A[\ell] = x,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell < i < n \ \to\ A[i] \ge x)$
- Stav 10: Platí jedna z následujících dvou možností:
  - $\ell = n,\ \forall i(0 \le i < n \ \to\ A[i] < x)$,
  - $0 \le \ell < n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell \le i < n \ \to\ A[i] > x)$

  Tyto dvě možnosti se dají zhrnout do jediného případu:
  - $0 \le \ell \le n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell \le i < n \ \to\ A[i] > x)$
- Stav 11: Platí jedna z následujících dvou možností:
  - *result* $= \ell,\ 0 \le \ell < n,\ A[\ell] = x,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell < i < n \ \to\ A[i] \ge x)$
  - *result* = NotFound, $0 \le \ell \le n,\ \forall i(0 \le i < \ell \ \to\ A[i] < x),\ \forall i(\ell \le i < n \ \to\ A[i] > x)$

e) Stačí probrat všechny hrany a pro každou z nich zkontrolovat, že pokud platí příslušný invariant před provedením dané instrukce, pak bude odpovídající invariant platit i po jejím provedení.

f) To, že se algoritmus zastaví, plyne z toho, že se s každou iterací cyklu snižuje hodnota $r - \ell$. Protože platí invariant $\ell \le r$, tato hodnota nikdy neklesne pod nulu, takže se nemůže snižovat donekonečna.

To, že se hodnota $r - \ell$ skutečně s každou iterací cyklu snižuje, je vidět z toho, že při každé iteraci cyklu musí nastat jeden ze dvou případů:

- Provede se přiřazení $\ell := k+1$: Vzhledem k tomu, že před tímto přiřazením platí $\ell \le k$, tímto přiřazením se zvětší hodnota proměnné ℓ. Hodnota proměnné r se v tomto případě nemění, takže hodnota $r - \ell$ se sníží.
- Provede se přiřazení $r := k$: Vzhledem k tomu, že před tímto přiřazením platí $k < r$, tímto přiřazením se sníží hodnota proměnné r. Hodnota proměnné ℓ se v tomto případě nemění, takže hodnota $r - \ell$ se sníží.

Formální důkaz toho, že se algoritmus zastaví by mohl vypadat tak, že se konfiguracím přiřadí následující vektory na základě řídícího stavu a ověří se, že při provedení každé instrukce se z aktuální konfigurace přejde do konfigurace, které je přiřazen lexikograficky menší vektor.

- Stav 0: (6)
- Stav 1: (5)
- Stav 2: $(4, r - \ell, 3)$
- Stav 3: $(4, r - \ell, 2)$
- Stav 4: $(4, r - \ell, 1)$
- Stav 5: $(4, r - \ell, 0)$
- Stav 6: $(4, r - \ell, 0)$
- Stav 7: (3)
- Stav 8: (2)
- Stav 9: (1)
- Stav 10: (1)
- Stav 11: (0)

g) Z předchozí analýzy vyplývá, že se algoritmus pro každý vstup zastaví a vrátí očekávaný výsledek. Pokud tedy není ve vyše uvedené analýze nějaká chyba, tak by měl být algoritmus korektní.

h) Označme MaxInt maximální hodnotu, kterou mohou nabývat proměnné daného celočíselného typu. V případě 32-bitových celých čísel se znaménkem bude $\text{MaxInt} = 2^{31} - 1 = 2147483647$.

Z výše popsaných invariantů vyplývá, že proměnné ℓ, r a k nabývají během výpočtu hodnoty z intervalu od 0 do n (včetně). Pokud tedy $0 \leq n \leq \text{MaxInt}$, tak i pro hodnoty těchto tří proměnných bude platit, že jsou vždy v intervalu $0, 1, \ldots, \text{MaxInt}$.

Mohlo by se tedy zdát, že je vše v pořádku. Ve skutečnosti je ale problém s výpočtem hodnoty výrazu na pravé straně v přiřazení $k := \lfloor (\ell + r)/2 \rfloor$, kde se hodnota $\ell + r$ do tohoto intervalu nemusí vejít. Co se v takovém případě stane, závisí na sémantice daného programovacího jazyka, detailech implementace apod. V některých jazycích (např. C, C++) je chování v případě přetečení znaménkového celého čísla nedefinované, v jiných (např. Java) se výpočet provede v modulární aritmetice modulo 2^{32}, apod. Při tomto chování se například pro hodnoty $\ell = 2147483646$ a $r = 2147483647$ chybně spočítá $l + r = -3$ a $k = -1$. Nejen, že pak neplatí invariant $\ell \leq k < r$, ale přistupuje se pak k prvku $A[-1]$, tedy mimo povolený rozsah indexů.

Příklad 5: Navrhněte algoritmus pro řešení následujícího problému. Jedná se o problém přiřadit vrcholům grafu barvy z dané množiny barev tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou. Barvy jsou označeny čísly $1, 2, \ldots, k$, kde k je celkový počet barev, které máme k dispozici. Pokud máme dán graf $G = (V, E)$, kde V je množina jeho vrcholů a E množina jeho hran, ***obarvením*** grafu G pomocí k barev budeme rozumět libovolnou takovou funkci $f : V \to \{1, 2, \ldots, k\}$, kde pro každou hranu $\{u, v\} \in E$ (kde $u, v \in V$) platí $f(u) \neq f(v)$.

Vstup: Neorientovaný graf $G = (V, E)$ a přirozené číslo k.

Výstup: Nějaké obarvení grafu G pomocí k barev nebo informace, že žádné takové obarvení neexistuje.

Poznámky:

- Předpokládejte, že vrcholy grafu G jsou označeny čísly $1, 2, \ldots, n$ (kde n je celkový počet vrcholů), a že graf G je zadán na vstupu ve formě, kdy je dáno toto číslo n a seznam hran, kde je každá hrana reprezentována jako dvojice čísel udávajících čísla vrcholů spojených touto hranou.
- Může být rozumné celé řešení rozložit na několik podprogramů (funkcí, procedur, metod, ...), řešících jednotlivé podúlohy. U podprogramů řešících jednoduché dílčí podúlohy, kde je jasné, jak by se daný podprogram dal implementovat, není třeba detailně (např. pomocí pseudokódu) popisovat činnost tohoto podprogramu, ale stačí stručně slovně popsat, *co* má tento podprogram dělat (není třeba popisovat, *jak* to bude dělat).

  Oproti tomu klíčové části algoritmu by měly být popsány přesně a podrobně, nejlépe pomocí pseudokódu, tak, aby jejich případná implementace v nějakém programovacím jazyce spočívala jen v rutinním přepsání tohoto pseudokódu do daného programovacího jazyka.
- Při řešení může být výhodné použít rekurzi.
- Alespoň neformálně pak zdůvodněte, proč je vámi navržený algoritmus korektní, tj. čím je zaručeno, že se pro každý vstup zastaví po konečném počtu kroků a že vydá správný výsledek.

Řešení: Příkladem přímočarého (ale nepříliš efektivního) řešení je Algoritmus 7. V tomto algoritmu se předpokládá, že n, k, f a *neighbours* jsou globální proměnné s následujícím významem (všechny ostatní proměnné jsou lokální):

- n — počet vrcholů grafu
- k — počet barev, které jsou k dispozici
- f — pole reprezentující aktuální přiřazení barev vrcholům. Barvy jsou reprezentovány čísly $1, 2, \ldots, k$. Číslo 0 znamená, že danému vrcholu není zatím přiřazena žádná barva.
- *neighbours* — pole přiřazující každému vrcholu množinu jeho sousedů, tj. $i' \in$ *neighbours*$[i]$ právě tehdy, když vrcholy i a i' jsou spojeny hranou.

Pole f a *neighbours* jsou indexována od jedné.

Procedura Read-Input načte graf a číslo k, procedura Print-Solution pak vypisuje nalezené obarvení grafu.

Jedná se o řešení hrubou silou, které systematicky vyzkouší všechna možná obarvení. Pokud tedy existuje alespoň obarvení daného grafu k barvami, tento algoritmus nějaké takové obarvení najde.

Algoritmus 7: Barvení grafu

```
MAIN():
    READ-INPUT()
    for i := 1 to n do
        f[i] := 0
    if SEARCH(1) then
        PRINT-SOLUTION(f)
    else
        print "No solution"

SEARCH(i):
    if i > n then
        return TRUE
    for c := 1 to k do
        if ∀i′ ∈ neighbours[i] : f[i′] ≠ c then
            f[i] := c
            if SEARCH(i + 1) then
                return TRUE
            f[i] := 0
    return FALSE
```