3. Počítačové cvičení

Rekurze a iterace

BIOLOGICKÉ vs. POČÍTAČOVÉ ALGORITMY

Příkladem biologického algoritmu může být proces replikace DNA. Na začátku procesu je jedno vlákno dvoušroubovice DNA, po skončení procesu existují dvě naprosto identická vlákna dvoušroubovice DNA. Replikace vlákna DNA v prokaryotické buňce obsahuje tyto kroky (zjednodušeno):

1) Nalezení replikačního počátku
2) DNA helikáza rozplétá rodičovskou dvoušroubovici
3) DNA-topoizomeráza odstraňuje nadšroubovicové smyčky, které vznikají při rozplétání
4) Vytváří se replikační vidlice
5) Na separovaná vlákna se váže protein SSB pro stabilizaci, aby nedošlo k opětovnému spojení s komplementárním vláknem

Stažení podkladů z GitHub

1. Na vlastním GitHub účtu vytvořte kopii (**fork**) zdrojového repozitáře. Otevřete v prohlížeči adresu zdrojového repozitáře. Vpravo nahoře najdete tlačítko **Fork** a klikněte na něj.
2. Naklonujte si repozitář ze **svého GitHub účtu** do složky s dnešním cvičením.
   ```
   git clone <repository address>
   ```
3. V lokálním repozitáři nastavte pomocí terminálu novou vzdálenou adresu (**remote**) na **původní** (vut-mpc-prg) adresu repozitáře (trojúhelníková spolupráce):
   ```
   git remote add upstream <repository_address>
   ```

Problém s vracením mincí

Představte si, že jste v knihkupectví a kupujete si knihu za 423 Kč. U pokladny platíte 500 Kč bankovkou. Pokladní vám má vrátit 77 Kč. Kdyby vám vrátila po jedné minci hodnot 50, 20, 5 a 2 Kč, budete spokojení, protože vám v peněžence bude chrastit nejmenší možný počet mincí, které vám pokladní mohla dát. Kdyby vám však chtěla těch 77 Kč vrátit v 77 jednokorunových mincích, asi byste měli pochyby o jejím duševním zdraví.

Jak rozhodnout, které mince a v jakém množství vrátit, aby jich zákazník dostal co nejméně? Toto je příklad problému, který by šlo algoritmizovat. Před tím však musíme provést analýzu problému a uvědomit si, co máme na vstupu, co chceme mít na výstupu a jaké prostředky máme k dispozici.

Vstup:	množství peněz ***M***, které se má vrátit
Výstup:	nejmenší množství mincí, jejichž součet hodnot se bude rovnat ***M***
Prostředek:	mince v hodnotách 50 (***pd***), 20 (***dc***), 10 (***ds***), 5 (***p***), 2 (***d***) a 1 (***j***) Kč
Podmínka:	50****pd*** + 20****dc*** + 10****ds*** + 5****p*** + 2****d*** + 1****j*** = ***M***
	pd, ***dc***, ***ds***, ***p***, ***d*** a ***j*** musí mít co nejmenší hodnotu

Nástin algoritmu v pseudokódu:

VrátitMince(M)

1 ***while*** *M > 0*

2 c ← mince s největší hodnotou, ale menší nebo rovnou M

3 dej minci zákazníkovi

4 M ← M – c

Trochu podrobněji slovně:

VrátitMince(M)

1 dej zákazníkovi celočíselný výsledek po dělení M/50 v 50 Kč mincích

2 zbytek po dělení se má ještě vrátit

3 dej zákazníkovi celočíselný výsledek po dělení zbytek/20 v 20 Kč mincích

4 zbytek po dělení se má ještě vrátit

5 dej zákazníkovi celočíselný výsledek po dělení zbytek/10 v 10 Kč mincích

6 zbytek po dělení se má ještě vrátit

7 dej zákazníkovi celočíselný výsledek po dělení zbytek/5 v 5 Kč mincích

8 zbytek po dělení se má ještě vrátit

9 dej zákazníkovi celočíselný výsledek po dělení zbytek/2 v 2 Kč mincích

10 dej zákazníkovi zbytek po dělení v 1 Kč mincích

Úkol 1: Zapište do pseudokódu podrobnější popis vracení mincí a implementujte v R. Vytvořte novou revizi (**commit**) a změny nahrajte na svůj vzdálený repozitář (**push**).

1. Nově vytvořený soubor přidejte do revize
   `git add <file_name>`
2. Vytvořte revizi, zadejte zprávu k revizi, uložte a zavřete.
   `git commit`
3. Vytvořenou revizi odešlete do svého repozitáře na GitHub.
   `git push origin master`

Řešený problém vracení mincí není univerzální, zabývali jsme se pouze případem pro mince v české měně. Co když budeme chtít problém zobecnit pro použití v jakékoli měně, či budeme mít omezený počet mincí některých hodnot?

Vezměme nejprve obecný případ pro jakoukoli měnu:

Vstup: množství peněz M, které se má vrátit, a mince o hodnotách $c=(c_1, c_2,\dots,c_d)$ v sestupných hodnotách $(c_1 > c_2 > \dots > c_d)$, d *je počet druhů mincí*

Výstup: celočíselné hodnoty $i_1, i_2,\dots, i_d$

Podmínka: $c_1*i_1 + c_2*i_2 + \dots + c_d*i_d = M$, kde $i_1 + i_2 + \dots + i_d$ je co nejmenší

Úkol 2: Zapište do pseudokódu vracení mincí pro jakoukoli měnu. Nápověda: použijte indexování polí.

Úkol 3: Pseudokód pro vracení mincí pro jakoukoli měnu implementujte v R jako samostatnou funkci, kde vstupem budou hodnoty mincí dané měny a částka kolik se má vrátit.
Vytvořte novou revizi (**commit**) a změny nahrajte na svůj vzdálený repozitář (**push**).

Úkol 4: Najděte takové vstupní hodnoty, pro které nebude algoritmus fungovat správně, tj. na výstupu bude špatný výsledek.

Aby byl algoritmus správný, nesmí existovat žádná varianta vstupů, při které by byl výstup špatný! Korektní algoritmus řeší všechny možnosti.

Správným řešením problému vracení mincí v jakékoli měně, resp. hodnot mincí by mohlo být toto:

*VraceníMincí(M, **c**, d)*

1 nejmenšíPočetMincí ← ∞

*2 **for each** ($i_1, i_2,\ldots, i_d$) **from** (0,…,0) **to** ($M/c_1,\ldots, M/c_d$)*

3 hodnotaMince ← $\sum^d_{k=1} i_k * c_k$

*4 **if** hodnotaMince = M*

5 početMincí ← $\sum^d_{k=1} i_k$

*6 **if** početMincí < nejmenšíPočetMincí*

7 nejmenšíPočetMincí ← PočetMincí

*8 **nejlepšíVrácení**← (i1, i2,…, id)*

9 return nejlepšíVrácení

Rekurzivní vs. iterativní algoritmy

Problém může být řešen rekurzivně či iterativně pouze v případě, že se v něm stále opakuje ta samá sekvence úkolů. Rekurzivní algoritmus volá sám sebe, tj. vnořuje stále hlouběji do sebe, až se dostane k nejmenší části problému, kterou lze okamžitě vyřešit a po jejím vyřešení dochází k postupnému vynořování, přičemž k řešení aktuální části algoritmu se používá výsledek řešení vnořené části. Iterativní algoritmus řeší jednotlivé opakující se části problému samostatně a sekvenčně za sebou (použití cyklů jako ***while*** a ***for*** je nejjednodušším základem iterativního algoritmu).

Jak vytvořit rekurzivní algoritmus: Nejčokoládovější cesta.

Představte si myš v labyrintu s místnůstkama, v kterých je umístěno různé množství čokolády, viz obrázek níže. Množství čokolády je znázorněno čísly. Myš se může pohybovat pouze směrem dolů a šikmo doprava. Jejím cílem je sežrat co nejvíc čokolády. Která cesta bude ta správná?

Algoritmus chceme řešit rekurzivně, takže musíme najít základ problému a jeho nejmenší okrajovou část, jejíž řešení je potřeba k vyřešení další nadřazené části. Cesta začíná nahoře a myš má dvě možnosti, kam jít, tj. jen dolů nebo šikmo doprava. Labyrint si představíme jako matici ***M*** s označením řádků ***r*** a sloupců ***s***. Počáteční pozice myši je ***M(1,1)***, obecně ***M(r,s)***. Myš může změnit pozici na ***M(r+1,s)*** nebo ***M(r+1,s+1)***.

Matice může být obecně libovolně velká a jednotlivá políčka můžou mít libovolnou hodnotu. Nejtriviálnější případ k řešení je případ, že matice bude mít jen tři prvky (1 horní a 2 spodní). Dalším (nadřazeným) problémem je přidání dalšího řádku do matice. Vyřešením cesty pro matici o 3 prvcích dostaneme pozici, odkud můžeme řešit následující řádek. Máme tedy nalezeno jádro rekurze, čímž je řešení problému 3-prvkové matice.

myš začíná zde nahoře u trojky

3			
1	4		
5	3	0	
1	2	6	7

a může skončit kdekoli dole

jádro rekurze

x	
y	z

Jádro rekurze řešíme následovně. Jsou jen dvě možnosti pohybu myši s množstvím získané čokolády:

1) z pozice ***M(1,1)*** se přesune na pozici ***M(2,1)***, čímž získá čokolády 3 + 1 = 4,
2) z pozice ***M(1,1)*** se přesune na pozici ***M(2,2)***, čímž získá čokolády 3 + 4 = 7.

Jelikož myš hledá cestu s maximálním množstvím čokolády, řešením jádra je cesta, při které myš získá celkem 7 kousků čokolády. Řešení konkrétního příkladu můžeme zobecnit: hledáme maximum z ***[M(r,s) + M(r+1,s), M(r,s) + M(r+1,s+1)]***.

Řešení jádra v pseudokódu:

Jadro(M,r,s)

```
1  C ← M(r,s)
2  Cdolu ← M(r+1,s)
3  Csikmo ← M(r+1,s+1)
4  return max(Cdolu, Csikmo) + C
```

Obecné řešení jádra nám dá novou počáteční pozici (tj. indexy ***r*** a ***s***) pro řešení rozšířeného problému. Současně se redukuje původní matice, protože některé prvky již dále uvažovat nemůžeme, když se lze pohybovat pouze směrem dolů a šikmo doprava. (Redukce samozřejmě je pouze myšlenková, ve skutečnosti se nic neredukuje.)

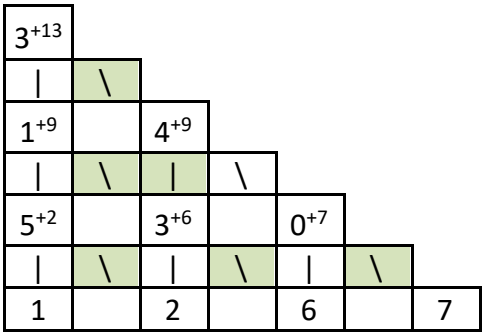
Varianta pro max. cesta do ***M(r+1,s)***

1			
5	3	0	
1	2	6	7

Varianta pro max. cesta do ***M(r+1,s+1)***

4		
3	0	
2	6	7

Řešit se musí všechny varianty, protože nevíme, která nakonec povede k cestě s maximálním množstvím čokolády. Když se však podíváte, obě varianty obsahují na začátku jádro. Stejně tak všechny ostatní varianty, které by vzešly z některého řešení. Tímto se dostáváme k úplně stejnému podproblému (až na hodnoty) jako dříve. Opět bychom použili funkci ***Jadro(M,r,s)***, ale indexy ***r*** a ***s*** nebudou rovny 1, jako na začátku, ale budou rovny indexům vzešlým z předcházejícího volání funkce. Jelikož však nevíme, jaké je předcházející řešení, noříme se stále více do matice. Skutečné řešení problému začíná od konce, tzn. jde se zpátky od nejnižších prvků matice až do pozice (1,1), přičemž se díky rekurzi řeší všechny možnosti a postupně se vybírá ta nejlepší.



Řešíme algoritmus rekurzivně, tj. voláme jádro rekurze. Když budeme předpokládat, že máme jádro správně, tak to obecně pak vypadá takto:

1 *C ← M(r,s)*
2 *Cdolu ← Jadro(M,r+1,s)*
3 *Csikmo ← Jadro(M,r+1,s+1)*
4 *return max(Cdolu, Csikmo) + C*

Před zkonstruování celého rekurzivního algoritmu si ještě musíme ujasnit, kdy algoritmus skončí. V tomto případě, až ***r*** se bude rovnat počtu řádků matice, tj. myš se dostane do políčka v posledním řádku a my potřebujeme vědět, kolik je tam čokolády. U psaní kódu pro rekurzivní algoritmus řešte ukončení jako první!

A takto vypadá celý rekurzivní algoritmus pro nalezení nejčokoládovější cesty (zatím nám dá jen kolik čokolády myška zbaští) a ***Jadro()*** jsme vyměnili za rekurzivní volání výsledné funkce:

Cokolada(M,r,s)

```
1   if r = počet řádků M
2           return M(r,s)
3   else
5           C ← M(r,s)
6           Cdolu ← Cokolada(M,r+1,s)
7           Csikmo ← Cokolada(M,r+1,s+1)
4           return max(Cdolu, Csikmo) + C
```

Úkol 5: Implementujte rekurzivní algoritmus v R.
Vytvořte novou revizi (**commit**) a změny nahrajte na svůj vzdálený repozitář (**push**).

Úkol 6: Vyřešte stejný problém iterativně.
Vytvořte novou revizi (**commit**) a změny nahrajte na svůj vzdálený repozitář (**push**).

Další příklad na rekurzi: Hanojská věž

Máme tři kolíky. Na levém kolíku jsou na sobě položené disky tak, že menší je vždy položený na větším. Cílem je přesunou všechny disky na pravý kolík. Větší disk se nikdy nesmí položit na menší. Na prázdný kolík je možné dát jakýkoli disk.

Vstup: disky o počtu *n*

Výstup: pořadí kroků, které vyřeší Hanojskou věž o *n* discích

Nejzákladnějším problémem Hanojské věže je přesun 1 disku z jednoho kolíku na jiný. To je víceméně poslední krok řešení problému. Stejně jako v předcházejícím příkladu s čokoládou musí kód algoritmu nejprve vyřešit ukončení. V tomto případě tedy, když počet disků je ***n***, ale zbývá už jen jeden, tak ho z aktuálního kolíku (***zKolík***) přesuneme na požadovaný kolík (***naKolík***) a průběh ukončíme.

V opačném případě se problém rekurzivního algoritmu zmenší o 1 (tedy ***n - 1***) a disky se přesouvají z aktuálního kolíku (***zKolík***) na jiný volný kolík (***volnýKolík***), než kam chceme přesunout všechny ostatní. Pak můžeme přesunout ten největší disk, co zbyl, na kolík, kam jsme původně chtěli (***naKolík***). A jako poslední krok, hromádku ***n-1*** disků přesuneme z jakože volného kolíku (***volnýKolík***) na největší disk, který je na požadovaném kolíku (***naKolík***). Nyní můžeme tuto myšlenku převést na rekurzivní algoritmus:

```
HanojskáVěž(n,zKolíku,naKolík)
1  if n = 1
2        output „Přesuň disk z kolíku zKolíku na kolík naKolík“
3        return
4  else
5        volnýKolík ← 6 – zKolíku – naKolík
6        HanojskáVěž(n-1,zKolíku,volnýKolík)
7        output „Přesuň disk z kolíku zKolíku na kolík naKolík“
8        HanojskáVěž(n-1,volnýKolík,naKolík)
```

? Proč je na řádku 5 *„volnýKolík ← 6 – zKolíku – naKolík“* ta šestka? Když máme kolíky označeny čísly 1, 2 a 3, tak 6 je jejich součet a tím můžeme spočítat, který kolík je volný.

Úkol 7: Implementujte tento algoritmus v R a nechte vyřešit úlohu pro 5 disků.
Vytvořte novou revizi (**commit**) a změny nahrajte na svůj vzdálený repozitář (**push**).

Rekurzivní algoritmy je často možné přepsat do iterativních a naopak. Rozhodnutí, který typ algoritmu použít, závisí na přehlednosti a pochopitelnosti výsledného algoritmu a především jeho efektivitě.

Úkol 8: Dle str. 33 v Knize 1 implementujte v R iterativní i rekurzivní řešení Fibonacciho posloupnosti.
Vytvořte novou revizi (**commit**) a změny nahrajte na svůj vzdálený repozitář (**push**).
Zjistěte, které řešení je časově efektivnější pro větší hodnoty vstupu (n = 10, 100, 1000).