# Bakalářské zkoušky (příklady otázek)

2024-02-09

## 1 Reprezentace bezkontextového jazyka (společné okruhy)

Uvažme následující jazyk nad abecedou $\{0, 1, \#\}$:

$$L = \{w\#s^R \mid w, s \in \{0,1\}^* \text{ a slovo } s \text{ je podslovem slova } w\}$$

(Poznámka: $s^R$ označuje slovo $s$ napsané pozpátku; podslovo je souvislý podřetězec, vč. prázdného a celého slova.)

1. Uveďte formální definici bezkontextové gramatiky a formální definici zásobníkového automatu.
2. Sestrojte nějakou bezkontextovou gramatiku generující jazyk $L$.
3. Sestrojte nějaký zásobníkový automat přijímající jazyk $L$.

**Nástin řešení**

1. *Bezkontextová gramatika* je $G = (V, T, \mathcal{P}, S)$, kde $V$ a $T$ jsou konečné neprázdné disjunktní množiny, $S \in V$, a $\mathcal{P}$ je konečná množina přepisovacích pravidel tvaru $H \to \beta$, kde $H \in V$ a $\beta \in (V \cup T)^*$.

   *Zásobníkový automat* je $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, kde $Q$, $\Sigma$ a $\Gamma$ jsou konečné neprázdné množiny, $\delta\colon Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \to P_{FIN}(Q \times \Gamma^*)$ , $q_0 \in Q$, $Z_0 \in \Gamma$, a $F \subseteq Q$ ($P_{FIN}$ značí konečné podmnožiny; $F$ lze vynechat, přijímáme-li prázdným zásobníkem).
2. Jazyk $L$ generuje například bezkontextová gramatika $G = (\{S, X, A\}, \{0, 1, \#\}, \mathcal{P}, S)$ s pravidly $\mathcal{P} = \{S \to AX, X \to 0X0 \mid 1X1 \mid A\#, A \to 0A \mid 1A \mid \lambda\}$ (kde proměnná $A$ generuje libovolné slovo nad $\{0, 1\}$).
3. Lze sestrojit převodem gramatiky $G$. Výsledkem je zásobníkový automat $(\{q_0\}, \{0, 1, \#\}, \{0, 1, \#, S, X, A\}, \delta, q_0, S)$ přijímající jazyk $L$ prázdným zásobníkem, kde přechodová funkce sestává z přechodů odpovídajících přepisovacím pravidlům a z přechodů pro čtení písmen:

   $$\delta = \{((q_0, \lambda, H), \beta) \mid H \to \beta \in \mathcal{P}\} \cup \{((q_0, a, a), \lambda) \mid a \in \{0, 1, \#\}\}$$

   Alternativně lze automat zkonstruovat přímo. Při čtení slova $w$ si jeho znaky ukládáme na zásobník, po přečtení # nejprve smažeme libovolný počet znaků ze zásobníku, poté čteme $s^R$ a kontrolujeme, zda souhlasí s písmeny na zásobníku, a následně vyprázdníme zásobník. Lze také sestrojit automat přijímající koncovým stavem.

## 2 Ovladač pro řadič disku (společné okruhy)

Napište implementaci funkce ovladače disku pro načtení jednoho bloku.

Uvažovaný disk je řízen pěti paměťově mapovanými registry, které slouží k zápisu příkazů (a jejich parametrů) a ke čtení aktuálního stavu zařízení (uvažujeme zařízení bez podpory přerušení, s obsluhou s aktivním čekáním).

<table>
<tr><th>Offset</th><th>Registr</th><th>Typ</th><th>Popis registru</th></tr>
<tr><td>0</td><td>Status</td><td>R</td><td>Bitové pole pro aktuální stav řadiče (pouze pro čtení, zápisy jsou ignorovány).<br><table><tr><td>31</td><td>29</td><td>...</td><td>2</td><td>1</td><td>0</td></tr><tr><td>0</td><td>0</td><td>...</td><td>0</td><td>BUSY</td><td>ERR</td></tr></table>BUSY – Bit je nastaven po zadání příkazu a vynulován po dokončení operace.<br>ERR – Bit signalizující chybový stav (1 znamená chybu).</td></tr>
<tr><td>4</td><td>Size</td><td>R</td><td>Velikost disku v blocích (pouze pro čtení, zápisy jsou ignorovány).</td></tr>
<tr><td>8</td><td>Command</td><td>W</td><td>Zápis do registru spustí zadaný příkaz (1 čtení, 2 zápis).</td></tr>
<tr><td>12</td><td>LBA</td><td>W</td><td>Logická adresa bloku pro příkaz zadaný v Command.</td></tr>
<tr><td>16</td><td>DMA</td><td>W</td><td>Fyzická adresa paměti pro uložení načtených dat z disku.</td></tr>
</table>

Bloky jsou na disku adresovány obvyklým způsobem pomocí LBA, jednotlivé bloky mají pevně danou velikost 512 bajtů. Přenos čtených či zapisovaných dat probíhá pomocí DMA (tedy z pohledu této otázky není třeba řešit nic víc než nastavení fyzické adresy pro uložení dat, vlastní čtení z disku a zápis do paměti je plně v režii řadiče), dokončení operace je signalizováno stavovým registrem (předpokládáme 32-bitový systém, velikost disku je omezena na 2 TB).

Dopište těla dvou níže uvedených funkcí a navrhněte strukturu pro uchování informací o disku (předpokládáme, že k systému může být připojeno více disků a ty jsou na úrovni jádra operačního systému rozlišeny různými instancemi struktury `disk_t`). Obě funkce vrací `true` pokud se daná operace zdařila, jinak `false` (bez dalších detailů).

Funkce `disk_init` je volána jednou při inicializaci daného zařízení, parametr `register_address` je virtuální adresa mapovaných registrů (tj. přímo adresa stavového registru). Funkce `disk_read_block_waiting` přečte jeden blok z disku. Načtený blok má být zapsán na fyzickou adresu `data_phys_addr` (váš kód může předpokládat její správnost). Návrat z funkce musí proběhnout až po dokončení čtení (je možné využít aktivní čekání).

Vaše implementace musí alespoň triviálním způsobem ošetřit možný současný přístup k disku z více procesů.

```
typedef struct {  ...  } disk_t;

bool disk_init(disk_t *disk, uint32_t register_address) {  ...  }

bool disk_read_block_waiting(disk_t *disk, size_t lba, uint32_t data_phys_addr) {  ...  }
```

**Nástin řešení** Nástin zdrojového kódu (bez symbolických konstant a komentářů):

```
typedef volatile struct {
    uint32_t status;
    uint32_t size;
    uint32_t command;
    uint32_t lba;
    uint32_t dma;
} disk_regs_t;

typedef struct {
    size_t max_lba;
    disk_regs_t *ctl;
    mutex_t mutex;
} disk_t;

static bool disk_is_ok(disk_t *disk) { return disk->ctl->status & 1 == 0; }
static bool disk_is_ready(disk_t *disk) { return disk->ctl->status & 2 == 0; }
static bool disk_wait_for_ready(disk_t *disk) {
    while (!disk_is_ready(disk)) {
        if (!disk_is_ok(disk)) {
            return false;
        }
   }
   return true;
```

```
}

bool disk_init(disk_t *disk, uint32_t register_address) {
    disk->ctl = (disk_regs_t *) register_address;
    disk->max_lba = disk->ctl->size;
    mutex_init(&disk->mutex);
    return disk_wait_for_ready(disk);
}

bool disk_read_block_waiting(disk_t *disk, size_t lba, uint32_t data_phys_addr) {
    if (lba >= disk->max_lba) {
        return false;
    }
    mutex_lock(&disk->mutex);
    bool ok = disk_wait_for_ready(disk);
    if (!ok) {
        mutex_unlock(&disk->mutex);
        return false;
    }

    disk->ctl->lba = lba;
    disk->ctl->dma = data_phys_addr;
    disk->ctl->command = 1;

    ok = disk_wait_for_ready(disk);
    mutex_unlock(&disk->mutex);

    return ok;
}
```

# 3 Skalání součin (společné okruhy)

1. Ověřte, že $\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \mathrm{trace}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B})$ je skalární součin na $\mathbb{R}^{m \times n}$, přičemž tzv. *stopa matice*, značená trace, je definována pro čtvercové matice řádu $n$ předpisem $\mathrm{trace}(\boldsymbol{C}) = \sum_{i=1}^{n} c_{ii}$.
2. Rozhodněte, zdali jsou matice $\boldsymbol{A}, \boldsymbol{B} \in \mathbb{R}^{4\times 6}$ na sebe ortogonální vzhledem k uvedenému skalárnímu součinu pro matici $\boldsymbol{A} = \begin{pmatrix} 1 & -2 & 3 & -4 & 5 & -6 \\ -2 & 3 & -4 & 5 & -6 & 7 \\ 3 & -4 & 5 & -6 & 7 & -8 \\ -4 & 5 & -6 & 7 & -8 & 9 \end{pmatrix}$ a matici $\boldsymbol{B}$, jejíž všechny prvky jsou rovny $-13$.
3. Zformulujete Cauchyho—Schwarzovu nerovnost a rozhodněte, zda pro čtvercovou matici $\boldsymbol{A}$ řádu $n$ platí: $(\mathrm{trace}(\boldsymbol{A}))^2 \leq n \cdot \mathrm{trace}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{A})$,

**Nástin řešení**

1. Po dosazení $\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \mathrm{trace}(\boldsymbol{A}^{\mathbf{T}}\boldsymbol{B}) = \sum_{i=1}^{n} (\boldsymbol{A}^{\mathrm{T}}\boldsymbol{B})_{ii} = \sum_{i=1}^{n} (\sum_{j=1}^{n} (\boldsymbol{A}^{\mathrm{T}}) ij b_{ji}) = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ji} b_{ji}$.

   Neboli po složkách mezi sebou vynásobíme matice $\boldsymbol{A}$ a $\boldsymbol{B}$ a výsledek sečteme.

   Toto zobrazení splňuje axiomy skalárního součinu:

   – $\langle \boldsymbol{A}, \boldsymbol{A} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ji})^2 \geq 0$

   – $\langle \boldsymbol{A}, \boldsymbol{A} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ji})^2 = 0 \Rightarrow \forall i, j : a_{ij} = 0 \Rightarrow \boldsymbol{A} = \mathbf{0}$

- $\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ji} b_{ji} = \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ji} b_{ji} = \langle \boldsymbol{B}, \boldsymbol{A} \rangle$
- $\langle \boldsymbol{A} + \boldsymbol{C}, \boldsymbol{B} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ji} + c_{ji}) b_{ji} = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ji} b_{ji} + \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ji} b_{ji} = \langle \boldsymbol{A}, \boldsymbol{B} \rangle + \langle \boldsymbol{C}, \boldsymbol{B} \rangle$
- $\langle t\boldsymbol{A}, \boldsymbol{B} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} t a_{ji} b_{ji} = t \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ji} b_{ji} = t \langle \boldsymbol{A}, \boldsymbol{B} \rangle$

Také vyplývá z toho, že jde o standardní skalární součin, pokud reálné matice typu $m \times n$ reprezentujeme jako reálné vektory o $mn$ složkách.

2. $\langle \boldsymbol{A}, \boldsymbol{B} \rangle = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ji} \cdot (-13) = -13 \sum_{i=1}^{n} \sum_{j=1}^{n} a_{ji} = 0$, protože $\boldsymbol{A}$ má řádkové součty $-3, 3, -3$ a 3.
3. Cauchyho—Schwarzova nerovnost: Pro každý vektorový prostor $V$ se skalárním součinem a normu odvozenou z tohoto skalárního součinu platí: $\forall \boldsymbol{u}, \boldsymbol{v} \in V : |\langle \boldsymbol{u}, \boldsymbol{v} \rangle| \leq ||\boldsymbol{u}|| \cdot ||\boldsymbol{v}||$.

   Dotazovaná nerovnost platí, umocněním obou stran Cauchyho—Schwarzovy nerovnosti dostaneme $\forall \boldsymbol{u}, \boldsymbol{v} \in V : \langle \boldsymbol{u}, \boldsymbol{v} \rangle^2 \leq \langle \boldsymbol{u}, \boldsymbol{u} \rangle \langle \boldsymbol{v}, \boldsymbol{v} \rangle$. Do této nerovnosti dosadíme uvedený skalární součin a vektory $\boldsymbol{u} = \mathbf{I}_n$ a $\boldsymbol{v} = \boldsymbol{A}$.

   Dostáváme: $(\mathrm{trace}(\boldsymbol{A}))^2 = (\mathrm{trace}(\mathbf{I}_n^{\mathrm{T}} \boldsymbol{A}))^2 = \langle \mathbf{I}_n, \boldsymbol{A} \rangle^2 \leq \langle \mathbf{I}_n, \mathbf{I}_n \rangle \langle \boldsymbol{A}, \boldsymbol{A} \rangle = n \cdot \mathrm{trace}(\boldsymbol{A}^{\mathrm{T}} \boldsymbol{A})$

# 4 Nespojitost funkce (společné okruhy)

Nechť $f : (0,1) \to \mathbb{R}$ je reálná funkce.

1. Definujte, co znamená, že $f$ je *nespojitá* v bodu $\frac{1}{2}$.
2. Je funkce $f$, když ji na tomto intervalu definujeme jako $f(x) = \frac{2x-1}{1-2x}$ pro $x \neq \frac{1}{2}$ a jako $f(\frac{1}{2}) = -1$, nespojitá v bodu $\frac{1}{2}$?
3. Pokud ji zadáme vzorcem $f(x) = \frac{2x-1}{1-2x}$ na celém intervalu $(0,1)$, je $f$ nespojitá v bodu $x = \frac{1}{2}$?

Odpovědi v částech 2 a 3 zdůvodněte.

**Nástin řešení**

1. Existuje $\varepsilon > 0$, že pro každé $\delta > 0$ existuje číslo $x \in (0,1)$, že $|x - \frac{1}{2}| < \delta$, ale $|f(x) - f(\frac{1}{2})| \geq \varepsilon$. Nebo: existuje posloupnost $(a_n) \subset (0,1)$, že $\lim a_n = \frac{1}{2}$, ale $(f(a_n))$ nemá limitu $f(\frac{1}{2})$.
2. Není. Pro tuto funkci je $\lim_{x \to \frac{1}{2}} f(x) = -1 = f(\frac{1}{2})$ (což je vidět třeba z algebraické úpravy $\frac{2x-1}{1-2x} = -1$ pro $x \neq -1$) takže (podle charakterizace spojitosti funkce v bodu pomocí limity) je tato $f$ v $\frac{1}{2}$ spojitá, a ne nespojitá.
3. Ne, není, pro $x = \frac{1}{2}$ není ani definovaná

# 5 Databáze SIS-Junior (specializace DW)

Na jedné základní škole v ČR nasadili nový informační systém SIS-Junior pro evidenci žáků a jejich studijní agendy. V seznamu níže jsou uvedeny vybrané tabulky a sloupce, které se týkají dat potřebných pro tisk vysvědčení (uvažujeme tradiční vysvědčení, na kterém jsou předměty a známky, nikoli slovní hodnocení).

```
STUDENT(id, first_name:CHAR, last_name:CHAR, born:DATE)
CLASS(id, label:CHAR, academic_year:INT, school_year:INT{1..9})
SUBJECT(id, name:CHAR)
REPORT_GRADE(student_id, class_id, subject_id, semester:INT{1..2}, grade:INT{1..5})
```

Sloupce pojmenované `id` jsou PK (s hodnotou UUID), cizí klíče mají vždy tvar název-tabulky_id. Datové typy jsou naznačeny za názvy položek, u čísel (kde je to podstatné) jsou uvedeny i očekávané rozsahy. Akademický rok se ukládá jako číslo roku, kdy začal (tj. 2023 odpovídá roku 2023/24).

1. Uvedené DB schéma zcela jistě není úplné. Doplňte sloupce a tabulky (včetně popisu jejich sloupců), které nejsou ve výpisu výše, abychom měli úplný popis části schématu odpovídající záměru použití (agenda vysvědčení), především aby bylo možné zcela splnit následující body.

2. Napište SQL dotaz typu `SELECT`, který vygeneruje podklady pro přípravu vysvědčení na druhé pololetí roku 2023/24 (tj. vypíše všechny platné trojice žák-třída-předmět, ke kterým pak učitelé budou ručně doplňovat známky a ukládat je do `REPORT_GRADE`).
3. Napište SQL dotaz typu `SELECT`, který vypíše průměrné známky pro každou unikátní kombinaci předmětu, ročníku studentů a školního roku (známky obou pololetí se průměrují dohromady).

**Nástin řešení** 1) Na první pohled chybí spojení student-class (je to běžný M:N vztah, takže tabulka) a subject-class (taky M:N, co se učí v které třídě). Také je možné uvést v subject-class tabulce i jestli se daný předmět učí jen v prvním, druhém, nebo obou pololetích.

```
STUDENT_IN_CLASS(student_id, class_id) ...PK jsou oba FK
SUBJECT_IN_CLASS(subject_id, class_id, semester) ...pokud je v obou semestrech, znamena to 2 zaznamy
```

2) Stačí začít s CLASS a spojit přes pomocné tabulky (protože nás zajímají jen IDčka studenta a předmětu).

```
SELECT student_in_class.student_id, class.id, subject_in_class.subject_id FROM class
JOIN student_in_class ON class.id = student_in_class.class_id
JOIN subject_in_class ON class.id = subject_in_class.class_id
WHERE class.academic_year = '2023' AND subject_in_class.semester = 2
```

3) Stačí jednoduchá kombinace JOIN a GROUP BY:

```
SELECT report_grade.subject_id, subject.name, class.academic_year, class.school_year,
  AVG(report_grade.grade)
FROM report_grade
JOIN class ON report_grade.class_id = class.id
JOIN subject ON report_grade.subject_id = subject.id
GROUP BY report_grade.subject_id, subject.name, class.academic_year, class.school_year
```

# 6 Databáze diplomových prací (specializace DW)

V rámci návrhu datového modelu aplikace byly na konceptuální úrovni identifikovány dvě datové třídy - *Osoba* a *Diplomová práce*. O osobě je nutné si pamatovat její unikátní číslo osoby, jméno, příjmení a typ (student/učitel). O diplomové práci je potřeba si pamatovat její název, studijní obor, název fakulty. Dále byly identifikovány následující vztahy:

- Diplomová práce má nejvýše jednoho *řešitele*. Řešitel může mít přidělenu nejvýše jednu práci
- Diplomová práce má právě jednoho *vedoucího*. Vedoucí může vést libovolný počet prací.

Pro výše popsanou situaci

1. Načrtněte pro popsanou situaci odpovídající UML model. Nezapomeňte vyznačit kardinality vztahů.
2. Převeďte Vámi navržený UML model na logický relační model. Nezapomeňte vyznačit všechny klíče a referenční integritu (cizí klíče). Převod proveďte tak, aby v cizích klíčích nemusela být nikdy vložena NULL hodnota.
3. Je logický relační model vzniklý převodem UML modelu vhodný z hlediska dosažené normální formy, pokud víte, že v modelované doméně atribut název fakulty funkčně závisí na studijním oboru (tedy *Studijní obor* → *Název fakulty*)? Pokud ne, jak by měl být logický relační model upraven, aby vhodný byl a proč?

**Nástin řešení**

1.
```
+----------------+                  +----------------+
| Osoba          |                  | DiplomovaPrace |
+----------------+ (0,1) Resi (0,1) +----------------+
| - Cislo        |------------------| - Nazev        |
| - Jmeno        |                  | - StudijniObor |
| - Prijmeni     | 1     Vede (0,*) | - NazevFakulty |
| - Typ          |------------------|                |
+----------------+                  +----------------+
```

2. – Osoba(<u>**OsobaID**</u>, <u>**Cislo**</u>, Jmeno, Prijmeni, Typ)

- DiplomovaPrace(**DiplomovaPraceID**, Nazev, StudijniObor, NazevFakulty, OsobaID)
  OsobaID $\subseteq$ Osoba.OsobaID
- Resi(**OsobaID**, **DiplomovaPraceID**),
  OsobaID $\subseteq$ Osoba.OsobaID,
  DiplomovaPraceID $\subseteq$ DiplomovaPrace.DiplomovaPraceID

Kardinalita vztahů se projeví v rozdílné definici klíčů vztahových tabulek.

3. Vzhledem k tranzitivní závislosti *DiplomovaPraceID* $\rightarrow$ *StudijniObor* $\rightarrow$ *NazevFakulty* není tabulka DiplomovaPrace ve 3NF. Pro nápravu je potřeba tabulku dekomponovat podle závislosti *StudijniObor* $\rightarrow$ *NazevFakulty*. Vzniknou dvě menší tabulky
   - Obor(**StudijniObor**, NazevFakulty)
   - DiplomovaPrace(**DiplomovaPraceID**,
     Nazev, StudijniObor), StudijniObor $\subseteq$ Obor.StudijniObor

# 7 Komprese (specializace DW)

Zpráva nad abecedou $\{a, b, c, d, e\}$ je náhodnou veličinou s rozložením pravděpodobností
$\{\frac{4}{20}, \frac{3}{20}, \frac{2}{20}, \frac{4}{20}, \frac{7}{20}\}$.

1. Vytvořte odpovídající kódovací strom pro Huffmanovo kódování do binární abecedy $\{0, 1\}$.
   Sourozence ve stromu řaďte dle (kumulativní) pravděpodobnosti.
   Následně zakódujte začátek zprávy *cdebea...*
2. Nadřízení by si přáli, aby byly zprávy kódovány pomocí následující tabulky:
   $a = 1$, $b = 10$, $c = 100$, $d = 1000$, $e = 0000$.
   Vysvětlete, proč použití tohoto způsobu kódování není úplně vhodné.
3. Jaké binární kódování by bylo potřeba použít, aby se délka zpráv co nejvíce blížila entropii zprávy?

**Nástin řešení**

1. Strom se konstruuje zdola nahoru spojováním dvojic symbolů s nejnižší pravděpodobností výskytu.
   Nejprve se tedy spojí symboly $(c, b)$ s celkovou pravděpodobností $\frac{5}{20}$.
   Poté symboly $(a, d)$ s celkovou pravděpodobností $\frac{8}{20}$.
   Poté symboly $((c, b), e)$ s celkovou pravděpodobností $\frac{12}{20}$.
   Nakonec symboly $((a, d), ((c, b), e))$ s celkovou pravděpodobností $\frac{20}{20}$.
   Kódovací strom tedy přidělí jednotlivým znakům kódy:
   $a = 00$, $d = 01$, $c = 100$, $b = 101$, $e = 11$.
   Začátek zprávy se v takovém případě zakóduje jako
   100|01|11|101|11|00| ...
2. Snazší argument je, že kódy svými délkami neodpovídají rozložení pravděpodobnosti (mj. dva poslední znaky se dvěma nejčastějšími výskyty mají nejdelší kódy) a kódové zprávy budou proto delší než v případě Huffmanova kódu. Bonusový postřeh je, že kód není prefixový, ale postfixový, a může proto nastat problém při jeho dekódování zleva doprava. Konkrétně například úseky 100000...00|1... bude potřeba nejprve celé přečíst a dekódovat zprava doleva, aby se zjistilo, jakým znakem zpráva začíná.
3. Aritmetické kódování.

# 8 Odeslání webového formuláře (specializace DW)

Mějme HTML formulář v tradiční (CGI-like, tedy bez skriptů na straně klienta) webové aplikaci na přidávání položek do databáze.

```
<form id="addForm" action="index.php?action=addItem" method="POST">...
```

Skript na straně serveru zpracuje data z formuláře a pokud jsou korektní, přidá nový záznam do databáze. Jako odpověď pak vygeneruje HTML stránku, na které je tabulka se všemi záznamy v databázi.

1. Uživatel vyplnil formulář, odeslal jej na server a zobrazila se mu stránka s aktuální tabulkou všech záznamů (včetně nově přidaného). Co přesně se stane, když uživatel klikne na tlačítko pro znovunačtení stránky (*Refresh*) v prohlížeči? Takové chování zřejmě není zcela žádoucí z pohledu uživatele, navrhněte příslušnou úpravu aplikace (bez použití skriptů na straně klienta).
2. Pokud data odesílaného formuláře nejsou korektní (nelze je uložit), je třeba zobrazit tento formulář znovu a zároveň jej předvyplnit daty, která uživatel zadal (aby je uživatel mohl jen upravit a nemusel zadávat znovu). Jak tuto funkcionalitu zajistit v kombinaci s vašim řešením problému z bodu 1?
3. Jak by bylo možné situaci popsanou v prvním bodě vyřešit, pokud bychom dovolili použít skripty na straně klienta? Načrtněte kostru takového řešení v JavaScriptu, pokud si nejste jisti názvy funkcí nebo událostí z DOM API, doplňte k nim komentář s vysvětlením.

Vaše řešení z bodů 1. a 2. popište pokud možno stručně a strukturovaně (např. formou odrážek), stačí popis pro běžnou situaci (není třeba pokrývat všechny myslitelné speciální případy). Naopak buďte konkrétní v technických detailech, zejména těch, které se týkají ukládání a přenosu dat mezi klientem a serverem (pokud je např. potřeba data od uživatele v bodu 2. dočasně někam uložit, napište kam/jak). Také můžete použít fragmenty PHP kódu, kde je považujete za vhodné.

**Nástin řešení** 1. Prohlížeč se zeptá, jestli má znovu odeslat data z formuláře, a pokud uživatel potvrdí, znovu se položka přidá do DB (tj. bude tam 2x). Vhodné řešení je místo HTML stránky vrátit HTTP response 302 nebo 303 (redirect) a hlavičku `Location` s novým URL (třeba i sám na sebe). Request na nové URL bude vykonán metodou GET a nové URL jen vrátí HTML (pak s refreshem nebude problém).

2. Rozdělení dotazu v bodě 1. na dva dotazy (POST uloží a udělá redirect, GET načte stránku) vede k tomu, že odeslaná data (POST) je potřeba někam uložit, aby si je následný GET mohl načíst. Typicky se na to používá user session (PHP na to má přímo API), jejíž ID je uloženo v cookies. Alternativně lze použít databázi nebo soubory. V obou případech je třeba, aby měla "ta data" nějaký unikátní identifikátor (např. náhodný nebo UUID), který se zároveň přenese v URL (GET na ten formulář bude obsahovat navíc query parametr s tím identifikátorem, tím se zároveň řekne obsluze toho GETu, že nemá zobrazit prázdný formulář, ale předvyplněný + zvýraznit chyby). HTML/JS testování formuláře před odesláním problém neřeší, testovat se musí na serveru.

3. Pokud dovolíme JS, můžeme si celou situaci usnadnit tak, že za odeslání formuláře je zodpovědný asynchronní HTTP request. Kostra takového JS by vypadala zhruba takto:

```
const form = document.getElementById('addForm');
form.onSubmit = function(ev) {
    ev.preventDefault();
    fetch(form.action, {
        body: new FormData(form),
        method: "POST",
    }).then(res => res.json())
    .then(res => {
        if (isResponseOk(res)) {
            location.assign('URL to page where list is displayed');
        } else {
            showErrors(form, res);
        }
    })
};
```

Funkce `isResponseOk()` a `showErrors()` není z pohledu otázky třeba rozepisovat, jejich semantika je jasná z názvu.

# 9 Kombinatorika (specializace OI-G-PADS, OI-G-PDM, OI-G-O, OI-O-PADS, OI-O-PDM, OI-PADS-PDM)

1. Kolik existuje dobře uzávorkovaných řetězců tvořených $n$ otevíracími a $n$ zavíracími závorkami? Řetězec závorek je dobře uzávorkovaný, jestliže v každém jeho počátečním úseku je alespoň tolik otevíracích závorek jako zavíracích. U tohoto vzorce nemusíte zdůvodňovat, proč platí.
2. Nechť $a_0 = 6$, $a_1 = 13$ a $a_i = 5a_{i-1} - 6a_{i-2}$ pro každé $i \geq 2$. Odvoďte vzorec pro $a_n$.

3. Který z počtů z předchozích dvou bodů je pro velké $n$ větší a proč?

**Nástin řešení**

1. Počet uzávorkování je jedním ze základních příkladů na Catalanova čísla, jejich počet je
$$\frac{1}{n+1}\binom{2n}{n}.$$
2. Rekurenci můžeme řešit například pomocí vytvořujících funkcí. Položíme-li $f(x) = \sum_{n\geq 0} a_n x^n$, rekurence nám dává
$$(1-5x+6x^2)f(x) = a_0 + (a_1 - 5a_0)x = 6 - 17x,$$
a tedy
$$\begin{aligned} f(x) &= \frac{6-17x}{1-5x+6x^2} = \frac{6-17x}{(1-2x)(1-3x)} \\ &= \frac{5}{1-2x} + \frac{1}{1-3x} = 5\sum_{n\geq 0}(2x)^n + \sum_{n\geq 0}(3x)^n. \end{aligned}$$
Porovnáním koeficientů u $x^n$ tedy dostáváme
$$a_n = 5 \cdot 2^n + 3^n.$$
3. Ze základních odhadů na kombinační čísla máme $\binom{2n}{n} = \Omega(4^n/n^{1/2})$, počet uzávorkování je tedy $\Omega(4^n/n^{3/2})$. Pro velké $n$ je proto větší než $a_n = O(3^n)$.

## 10 Geometrie (specializace OI-G-PADS, OI-G-PDM, OI-G-O)

1. Zformulujte Radonovu větu v $\mathbb{R}^d$.
2. Dokažte pomocí Radonovy věty: Je-li $P$ konvexní simpliciální (tj. každá faseta je simplex) mnohostěn v $\mathbb{R}^5$ takový, že konvexní obal každé trojice jeho vrcholů je stěna $P$, pak už je $P$ nutně simplex.

**Nástin řešení**

1. Jiří Matoušek, Introduction to Discrete Geometry, Theorem 1.3.1
2. Sporem: není-li $P$ simplex, má aspoň 7 vrcholů, a tedy podle Radonovy věty dostaneme Radonův rozklad, jehož menší část má nejvýše tři vrcholy; zároveň ale její konvexní obal má být stěna, a to je spor.

## 11 Pokročilá diskrétní matematika (specializace OI-G-PDM, OI-O-PDM, OI-PADS-PDM)

Porovnejte následující množiny dle mohutnosti:

1. $\mathbb{N}$
2. $\mathbb{N}^{10}$
3. Množina všech konečných posloupností přirozených čísel.
4. Množina všech posloupností čísel 0 a 1.
5. $\mathbb{R}$

**Nástin řešení** První tři množiny mají stejnou mohutnost; například, s využitím Cantor-Bernsteinovy věty:

- $\mathbb{N} \preceq \mathbb{N}^{10}$, jelikož můžeme každé $n \in \mathbb{N}$ injektivně zobrazit na 10-tici $(n, \ldots, n)$.

– $\mathbb{N}^{10} \preceq$ konečné posloupnosti, jelikož 10-tici celých čísel můžeme interpretovat jako posloupnost délky 10.

– konečné posloupnosti $\preceq \mathbb{N}$: Posloupnost $a_1, \ldots, a_n$ můžeme injektivně zobrazit na přirozené číslo

$$2^n \prod_{i=1}^{n} p_{i+1}^{a_i},$$

kde $p_i$ označuje $i$-té prvočíslo.

Čtvrtá a pátá množina mají stejnou mohutnost: Posloupnost $(a_i : i \in \mathbb{N})$, kde $a_i \in \{0,1\}$, můžeme injektivně zobrazit na reálné číslo $\sum_{i\in\mathbb{N}} \frac{a_i}{3^i}$. Naopak, reálné $r$ číslo si můžeme vyjádřit formou zápisu ve dvojkové soustavě jako $\pm d_m d_{m-1} \ldots d_1, c_1 c_2 \ldots$ a injektivně ho zobrazit na posloupnost $s, d_1, c_1, d_2, c_2, \ldots$, kde $d_i = 0$ pro $i > m$ a $s = 1$ právě když $r$ je kladné.

Mohutnost prvních tří množin je striktně menší než zbylých dvou. Zjevně $\mathbb{N} \preceq \mathbb{R}$, jelikož $\mathbb{N} \subset \mathbb{R}$. Že neplatí rovnost lze dokázat diagonální metodou: Řekněme například, že by existovala bijekce $f$ přiřazující každému přirozenému číslu posloupnost 0 a 1. Uvažme posloupnost $1-(f(1))_1$, $1-(f(2))_2$, .... Tato posloupnost 0 a 1 je různá od $f(i)$ pro každé $i \in \mathbb{N}$, jejikož její $i$-tý prvek je $1-(f(i))_i \neq (f(i))_i$. To je ve sporu se surjektivitou $f$.

## 12 Toky v sítích (specializace OI-G-PADS, OI-O-PADS, OI-PADS-PDM)

1. Formulujte základní myšlenky Dinicova algoritmu na hledání maximálního toku v síti. (Nemusíte psát detailní pseudokód celého algoritmu.)
2. Kolik fází (průchodů vnějším cyklem) může algoritmus udělat? Jaká je časová složitost jedné fáze? Co z toho plyne pro složitost celého algoritmu?
3. Ukažte lepší horní odhad na složitost fáze (a tím pádem na složitost celého algoritmu) pro sítě, v nichž všechny kapacity hran leží v množině $\{0,1,2,3\}$.

**Nástin řešení**

1. Viz Průvodce labyrintem algoritmů, kapitola 14.4.
2. Označme počet vrcholů $n$ a počet hran $m$; bez újmy na obecnosti je $n \in \mathcal{O}(m)$. Fází nikdy není více než $n$ (každá fáze prodlouží nejkratší nenasycenou cestu ze zdroje do spotřebiče alespoň o 1). Fáze se skládá z konstrukce sítě rezerv (běží v čase $\mathcal{O}(m)$), čištění sítě ($\mathcal{O}(m)$) a výpočtu blokujícího toku – ten najde nejvýše $m$ cest, každou v čase $\mathcal{O}(n)$. Celkem tedy fáze trvá $\mathcal{O}(nm)$ a celý algoritmus $\mathcal{O}(n^2 m)$.
3. Výpočet blokujícího toku za každou cestu zvýší velikost toku aspoň o 1. Velikost blokujícího toku je omezena velikostí maximálního toku v síti rezerv, a ten je omezen kapacitou jakéhokoliv řezu v síti rezerv. Řez kolem zdroje má kapacitu menší než $6n$, neboť rezervy hran leží mezi 0 a 6, takže celkem projdeme $\mathcal{O}(n)$ cest místo původního odhadu $m$. Fáze tedy běží v čase $\mathcal{O}(m)$ a celý algoritmus v $\mathcal{O}(nm)$.

## 13 Metrické prostory (specializace OI-G-PADS, OI-G-PDM, OI-G-O, OI-O-PADS, OI-O-PDM, OI-PADS-PDM)

1. Definujte metrický prostor $(M,d)$.
2. Kdy je množina $X \subset M$ uzavřená?
3. Nechť $(M,d)$ je jednotkový reálný interval $(0,1]$ (tj. bez 0 a včetně 1), s obvyklou metrikou $d(x,y) = |x-y|$. Je jeho podmnožina $X = (0, \frac{1}{2}]$ uzavřená?

Odpověď v části 3 zdůvodněte.

**Nástin řešení**

1. $M$ je množina (neprázdná) a metrika $d: M \times M \to \mathbb{R}$ splňuje, že (i) vždy $d(x,y) \geq 0$ a $d(x,y) = 0 \iff x = y$, (ii) vždy $d(x,y) = d(y,x)$ a (iii) vždy $d(x,z) \leq d(x,y) + d(y,z)$.

2. $X \subset M$ je uzavřená $\iff$ $M \setminus X$ je otevřená. Otevřenost $M \setminus X$ znamená, že

$$\forall x \in M \setminus X \, \exists r > 0 \, \big(y \in M \,\&\, d(x, y) < r \Rightarrow y \in M \setminus X\big).$$

   Nebo lze uzavřenost $X$ definovat pomocí limit: když $(a_n) \subset X$ má $\lim a_n = a \in M$, pak $a \in X$.

3. Ano, je, její doplněk $(\frac{1}{2}, 1]$ je otevřená množina.

# 14 Anti-aliasing (specializace PGVVH-PG)

Anti-aliasing je technika, která může významně vylepšit vzhled rastrových grafických výstupů.

1. Který z parametrů rastrového zobrazovacího zařízení je anti-aliasingem vylepšen (obrázek se nám jeví jako na „kvalitnějším“ zařízení – ale v jakém smyslu)? Jak se to pozná na výsledném obrázku? Jak byste se u výstupu do okna ve Windows přesvědčili, zda byl anti-aliasing použit?
2. Jak se může anti-aliasing implementovat v klasickém rastrovém vykreslování (rasterizace - např. při kreslení vektorové grafiky)?
3. Jak se anti-aliasing implementuje v prostředí paprskového zobrazovače (např. Ray-tracing)?

**Nástin řešení**

1. Anti-aliasing (vyhlazení) je vylepšení vzhledu nakresleného objektu do rastrového výstupního zařízení. Okraje vypadají více hladké, textury se v dálce „nezrní“ (potlačení interference, Moiré efektu). Je to vlastně imitace vyššího rozlišení displeje za pomoci barevných přechodových odstínů v okrajových pixelech. Virtuálně se zvyšuje rozlišení displeje.

   Matematický model pixelu: čtvereček (plocha!)

   Barva pixelu: integrální průměr barvy na ploše toho čtverečku. Tj. např. když se kreslí vybarvený kruh, tak na jeho okraji jsou pixely ne zcela pokryté kruhem, ty je potřeba vybarvit barvou tak sytou, jak je podíl zakryté plochy.

   Přesněji: je-li $\alpha$ podíl zakrytí pixelu objektem, je výsledná barva: $\alpha \cdot \mathit{barvaObjektu} + (1 - \alpha) \cdot \mathit{barvaPozadi}$.

   Jak se o A-A přesvědčit ve Windows: pořídit si screenshot daného okna a potom si tento rastrový obrázek prohlédnout se zvětšním (je lepší mít prohlížeč obrázků nastavený tak, aby sám neprováděl žádná vylepšení, žádné interpolace) - pokud budou na hranách vidět přechodové barevné odstíny, byl použit anti-aliasing.
2. Při rastrovém vykreslování (např. v interpretu SVG) se dá představit cílový pixel jako čtvereček $M \times M$ subpixelů (např. $4 \times 4$), kreslit vše původními algoritmy do obrázku s $4 \times 4$ vyšším rozlišením a potom výsledek zprůměrovat (filtrovat) do původního požadovaného rozlišení. Umí to dělat (nebo je to snadno implementovatelné na) GPU.
3. V Ray-tracingu se do jednoho pixelu posílá více paprsků místo jednoho. Ve 2D modelu pixelu (jednotkový čtvereček na ploše virtuálního senzoru) se musí použít některá z vzorkovacích metod (sampling), např. Jittering, Hammersley nebo “N-rooks”. V případě potřeby (vyšší efektivita) lze implementovat i adaptivní převzorkování, kde se více vzorků (paprsků) posílá jen tam, kde je to potřeba: kde je obrazová funkce “zajímavá”, to se zjistí primárním hrubším vzorkováním.

   Jittering: rozdělím čtvereček pixelu na $M \times M$ podčtverečků a do každého umístím jeden vzorek jako nezávislý náhodný pokus. Je to nestranný odhad požadovaného integrálu a potlačuje interference i vytváření shluků vzorků (šum).

# 15 Architektura programovatelných GPU (specializace PGVVH-PG)

Půjde nám o popis architektury moderních programovatelných grafických karet (GPU), budeme na ně nahlížet pouze z hlediska realtime zobrazování 3D grafiky. Pokud si nepamatujete, jak se některá komponenta přesně odborně nazývá, nevadí – opište její funkci svými slovy.

1. Popište, z jakých základních částí se GPU skládá. Uveďte je v pořadí, ve kterém se účastní zpracování vykreslovaných 3D dat (tzv. zobrazovací řetězec neboli „pipeline“)
2. Které části zobrazovacího řetězce můžete jako autoři aplikace programovat? Co jsou to „konstanty“ (*uniforms*) a data jakého typu se do nich ukládají?

3. Popište podrobněji poslední (z hlediska umístění v řetězci) programovatelný modul. Co má za úkol spočítat, co má na vstupu a co na výstupu?

**Nástin řešení**

1. Základní části GPU (jen ty nejrelevantnější pro 3D grafiku)
   (a) Zpracování vrcholů (vertex procesor) - běží v něm "uživatelský" Vertex shader, který má za úkol zpracovat data každého vrcholu. Běžně se provádí geometrické transformace (je povinností transformovat POSITION do tzv. ořezávacího systému souřadnic, tj. typicky provést tyto transformace: modelovací, pohledovou a projekční)

   (Nepovinné.) Další nepovinná část se může skládat z "Tesselation" a "Geometry" shaderů - slouží k přidávání dalších kreslených primitiv až na GPU (zjemňování trojúhelníkových sítí nebo nahrazení vrcholů jednoduchými objekty třeba pro částicový simulátor)

   (b) Sestavení primitiv (Primitive assembly) - organizační/logistický modul zodpovědný za to, že jsou ke každému primitivu přítomny všechny komponenty/vrcholy. Zde se spojují vrcholy např. do trojúhelníků

   (Nepovinné.) Ořezávání, projekce, culling - pevně zadrátovaná část, která provede všechny další výpočty potřebné k tomu, abychom měli ve vrcholech souřadnice pro vykreslení (rasterizaci). Odstraní části scény, které nebudou vidět (odvrácené, mimo zorné pole...)

   (c) Rasterizace - převedení vektorových dat primitiv (vrcholy) na fragmenty/pixely. De facto se zde hardwarově realizuje vyplnění trojúhelníků, čtverečků, kreslení čar... Důležitou součástí jsou interpolátory, které interpolují všechny ostatní veličiny přiřazené k vrcholům, obvykle se používá tzv. perspektivně korektní interpolace (jednou z těch veličin je i "hloubka" pro depth-buffer)

   (d) Zpracování fragmentů/pixelů (fragment/pixel processor) - běží zde Fragment/Pixel shader, který má jako hlavní úkol určit výslednou barvu pixelu. K tomu se používají textury (texturové souřadnice byly mezitím z vrcholů interpolovány do fragmentů), barvy vrcholů nebo další veličiny, které si programátoři zvolili. Například zde lze realizovat tzv. Phongovo stínování = interpolace normál.

   (e) Finální aplikace fragmentu a zápis do frame-bufferu. Zde se hlavně jedná o výpočet viditelnosti pomocí Depth-bufferu a někdy také o maskování pomocí tzv. šablony (stencil). Viditelné fragmenty se nakonec zapisují do viditelného frame-bufferu, při jejich aplikaci lze použít i např. poloprůhlednost (alpha blending)

2. Je povinné definovat dvě programovatelné komponenty: musí se dodat Vertex shader (přepočítávající data vrcholů, zejména jejich transformace) a Fragment/Pixel shader (pro výslednou barvu výsledných pixelů). Nepovinně se mohou použít též Tesselation shaders (Tesselation Control Shader a Tesselation Evaluation Shader) nebo Geometry shader či nejnovější Mesh shaders.

   "Uniforms" jsou globální hodnoty, které aplikace posílá shaderům. Z hlediska shaderů se jedná o Read-only data (proto jim Microsoft říká "Constants"), nemění se v rámci jedné kreslicí dávky (např. jednoho vykreslovacího příkazu OpenGL). Mezi kreslicími dávkami se naopak často mění, a protože se jedná obvykle o data malého rozsahu, jde o celmi efektivní přístup. Typicky jsou takto předávány transformační matice (Model-View-Projection), odkazy na textury, údaje o světelných zdrojích, křivky pro barevné tónování... - vše, co se nehodí přidávat ke všem vrcholům scény.

3. Poslední je Fragment/Pixel Shader. Má za úkol určit barvu (volitelně i průhlednost) fragmentu, který se pak bude zapisovat do frame-bufferu. K tomu používá jednak veličiny předané z vrcholů (a interpolované v rasterizéru), dále textury (barevná mapa, normálová mapa, apod.) a obvykle i globální údaje o světelných zdrojích.

   Na vstupu jsou veličiny interpolované z vrcholů (poloha ve světovém systému souřadnic, normálový vektor, vlastní barva plochy...) a uniforms (textury, poloha a parametry zdroje světla...).

   Na výstupu je finální barva a průhlednost fragmentu (RGBA). Fragment se pak zkombinuje s již existujícím pixelem ve frame-bufferu a výsledkem je změna toho pixelu (toto vše samozřejmě jen pokud fragment projde testem šablony a hloubkovým testem).

   (Nepovinné.) Úplně zřídka se využívá možnosti, aby fragment shader modifikoval i hloubku (souřadnici Z) fragmentu. Tím se sice může ovlivnit jeho viditelnost, ale zároveň se degraduje HW optimalizační mechanismus GPU, tzv. Early Fragment Test (Early Depth Test).

# 16 Rekurzivní sledování paprsku (specializace PGVVH-PG)

Budeme se zabývat algoritmem zobrazování 3D scény, který se nazývá „Ray tracing“ (RT, česky by to bylo „Rekurzivní sledování paprsku“). Vaše slovní odpovědi můžete doprovodit obrázky a schématy, ale nezapomeňte je opatřit vysvětlivkami.

1. Popište princip algoritmu rekurzivního sledování paprsku, co je na vstupu a co je výsledkem výpočtu?
2. Z jakých složek se počítá světlo v rekurzivní funkci `shade()`? U jednotlivých složek uveďte, do jaké míry jsou fyzikálně správné (škála může být „přesně takto to v přírodě funguje“ – „ok, ale vzorec je trochu zjednodušen“ – „neodpovídá to fyzikální realitě, je tam jen kvalitativní shoda“).
3. Jaké hlavní nedostatky vykazuje základní algoritmus RT popsaný v prvním bodě, pokud bychom ho chtěli použít jako fotorealistickou zobrazovací metodu? Zkuste uvést aspoň rámcově, jak se dají jednotlivé nedostatky potlačit či omezit.

**Nástin řešení** Každým pixelem virtuálního rastrového obrázku (obrazovky) se pošle paprsek proti směru šíření světla do 3D scény (to navrhl již 1984 Whitted). Paprsek se protne se scénou a pokud na něco narazí, přenese se do pixelu barva toho průsečíku na povrchu tělesa. Pokud není zasaženo žádné těleso, vezme se barva pozadí. Sondovací paprsek se nejčastěji implementuje jako rekurzivní funkce

```
RGBcolor shade(Point3D origin, Vector3D direction, int recursionDepth)
```

Uvnitř funkce `shade()` se zkombinují tyto složky:

1. Přímé osvětlení povrchu tělesa ze všech definovaných světelných zdrojů. U každého zdroje světla se nejprve zkontroluje, zda má přímou viditelnost na průsečík (pokud ne, zdroj se ignoruje). Pak se aplikuje některý z lokálních modelů odrazu světla, například Phong nebo Torrance-Sparrow. Přesnost: závisí na přesnosti modelu odrazu světla, ten může být hodně věrný. Ostré stíny se přepočítávající přesně.
2. Pokud inkrementovaná hloubka rekurze dosáhla limitu, žádně další složky se nepočítají a dosavadní hodnota se rovnou vrátí. Přesnost: zde se dopouštíme chyby, protože redukujeme množství světla. Přesně to dělají až Monte-Carlo metody renderingu.
3. U potenciálně lesklého povrchu se spočítá zrcadlově odražený vektor a funkce `shade()` se zavolá rekurzivně. Výsledek se přičte přenásobený vhodnou konstantou ("lesklost" povrchu). Přesnost: nejnovější chybou je předpoklad zrcadlového směru odrazu paprsku. Reálné materály takhle nefungují.
4. U průhledného materiálu se podle indexů lomu určí zalomený směr paprsku a také se funkce `shade()` zavolá rekurzivně. Výsledek se přičte přenásobený vhodnou konstantou ("průhlednost" povrchu). Přesnost: chybou je opět předpoklad dokonale rovného povrchu, na kterém se světlo láme.

Akumulovaná barva se z funkce shade() vrátí jako výsledek. Má mít sémantiku "co by vidět pozorovatel z budou 'origin', kdyby se díval směrem 'direction'".

Hlavní nedostatky:

1. Rekurze je omezována, obrázek je tedy tmavší. Řešením by bylo použít nějaký Monte-Carlo algoritmus, který by omezenost výpočtu numericky korektně kompenzoval (viz Path-tracing).
2. Zrcadlový směr odrazu/lomu. Měl by se použít distribuovaný ray-tracing a pomocí většího množství odražených/zalomených paprsků lépe aproximovat složitější fyziku odrazu/lomu světla na reálném materiálu.
3. Nepřímé osvětlení - RT umí jen přímé osvětlení od viditelných zdrojů světla, to hrubě neodpovídá realitě. Měly by se započítat i delší cesty světla od zdrojů k průsečíku, to ale umí až nestranné Monte-Carlo metody (Path-tracing, Light-tracing).
4. Bodové nebo směrové světelné zdroje (u kterých lze jednoduše spočítat viditelnost z průsečíku) jsou nerealistické. V přírodě existují jen plošné zdroje světla, ty by se měly zohlednit tím, že by se zavedla částečná viditelnost pomocí Monte-Carlo metody.
5. Jeden paprsek na pixel je málo, je třeba použít anti-aliasing (supersampling) a vzorkovat barvu pixelu mnoha primárními paprsky.

## 17 Distribuované sledování paprsku (specializace PGVVH-PG)

Tyto techniky se také nazývají „Monte-Carlo ray tracing“ a obecně vylepšují základní princip algoritmu rekurzivního sledování paprsku (Whitted 1984).

1. Vyjmenujte několik příkladů, kde je užitečné do rekurzivního sledování paprsku zapojit stochastický výpočet (Monte-Carlo integraci).
2. Vyberte si jedno konkrétní použití Monte-Carlo a popište ho detailně. Které veličina se integruje? Který nedostatek původního přístupu se odstraňuje nebo potlačuje? Celý postup (vylepšení) popište dostatečně podrobně, do detailu se nemusíte zabývat jen vzorkováním – to bude předmětem další podotázky.
3. Navrhněte vzorkování (sampling), které by se dalo dobře použít v předchozím příkladu. Stačí popsat metodu vzorkování rámcově nebo nakreslit obrázek s dostatečnými vysvětlivkami. Šlo by toto vzorkování implementovat adaptivně (zamyslete se nad tím, jak bychom tu adaptivitu mohli řídit)?

**Nástin řešení**

1. Měkké stíny, anti-aliasing, hloubka ostrosti objektivu, rozmazání pohybem, měkké odrazy světla, disperze světla.
2. Měkké stíny - plošný zdroj světla, integruje se osvětlení přes plochu zdroje. V každém průsečíku se odhaduje, kolik procent světla dopadá ze zdroje. Prakticky se odhady provádí vzorkováním plochy zdroje, tj. Monte-Carlo integrací. Původní velmi ostré hranice světla a stínu se nahradí příjemnějšími a realističtějšími měkkými přechody, současně se zohlední vzdálenost světelného zdroje, překážky a příjemce stínu.
3. Světelný zdroj ve tvaru obdélníka by se dobře dal vzorkovat metodou jittering (rozdělení plochy na MxN stejně velkých dílků a náhodný výběr jednoho vzorku v každém dílku). Některé jiné tvary zdroje jsou též pohodlně a korektně vzorkovatelné, jen je potřeba mít důraz na nestrannost (konstantní hustotu pravděpodobnosti na ploše zdroje).

   Adaptivní jittering je možný, lépe se implementuje v omezených podmínkách (tj. ne neomezený), např. systémem “quadtree” nebo zjemňováním faktorem 2 (rozdělením již existující oblasti na poloviny).

## 18 Procesy a vlákna (specializace PVS)

Pro všechny body předpokládejte kontext nějakého běžného desktopového operačního systému.

1. Vysvětlete, co to je vlákno (thread). Co v procesu tvoří součást kontextu vlákna (tedy co má každé vlákno vlastní)? Co z procesu naopak součástí kontextu vlákna není (tedy co všechna vlákna sdílejí)?

   **Nástin řešení** Vlákno je běžící výpočet (posloupnost operací) v procesu.

   Hlavní součásti stavu jsou zásobník obsahující posloupnost rámců funkcí volaných v kontextu vlákna a registry procesoru, který vlákno vykonává (pokud vlákno neběží, tak bude obsah registrů někde uložen, třeba na zásobníku). Součástí kontextu vlákna je také jeho stav z pohledu systému, tedy zda vlákno právě běží na nějakém procesoru, zda čeká na naplánování, zda čeká na nějakou podmínku.

   Součástí kontextu vlákna není halda ani globální proměnné.

2. Vysvětlete, co to je preemptivní přepínání vláken, a popište, jaké operace je při přeplánování nutné provést. V jakých situacích může v systému dojít k přeplánování vlákna na jiné? Je možné jiné než preemptivní přepínání (odpověď na tuto otázku rozveďte, nestačí pouze ano/ne)?

   **Nástin řešení** Systém, kde plánovač může donutit vlákno opustit procesor (přeplánovat ho) i bez spolupráce tohoto vlákna (bez toho, aby se vlákno muselo samo vzdát procesoru voláním nějaké funkce). Typicky má v takovém systému vlákno přiděleno nějaký časový úsek (kvantum), jak dlouho může bez přerušení běžet na procesoru. K detekci vyčerpání kvanta a tedy k přeplánování bude docházet typicky při obsluze přerušení časovače. Dále k přeplánování vlákna dochází při jeho vstupu do pasivního čekání.

3. Vysvětlete, co to přesně znamená, že vlákno přejde do pasivního čekání. Jak pasivní čekání probíhá a jak probíhá jeho ukončení? Uveďte příklad alespoň dvou funkcí, jejíž volání může typicky vést k pasivnímu čekání.

**Nástin řešení** Vlákno čeká na splnění nějaké podmínky probuzení, aniž by spotřebovávalo procesorový čas. Vstup do pasivního čekání znamená, že se vláknu nastaví stav „čekající“ a dojde k přeplánování na jiné vlákno. Čekající vlákna pak plánovač neuvažuje při výběru dalšího vlákna pro běh na procesoru. Po splnění podmínky probuzení vlákno pouze přejde do stavu „připravené“ (tedy nezačne nutně okamžitě běžet), což značí, že ho plánovač může vybrat pro běh na procesoru při nějakém budoucím přeplánovávání vláken.

Pasivní čekání typicky způsobí např. volání Join (čekání na ukončení nějakého jiného vlákna), Sleep (čekání na uplynutí nějakého času), čekání na vstup do kritické sekce při volání Lock na zámku zamčeném jiným vláknem, volání nějaké I/O funkce jako čtení dat se souboru (pokud nemůže být dokončeno synchronně).

## 19 Vyhodnocovací strom (specializace PVS)

Navrhněte vhodnou objektovou strukturu (třídy, rozhraní) pro stromovou reprezentaci matematických výrazů s celočíselnými konstantami, proměnnými, unárními a binárními operátory. Implementujte základní operace pro vytištění výrazu v prefixové notaci a vyhodnocení výrazu pro zadané hodnoty proměnných (stačí kód pro jeden unární a jeden binární operátor). Respektujte pravidla objektového návrhu a umožněte snadnou rozšiřitelnost o další operátory.

Dále stručně slovně popište (není nutné psát žádný kód, ale je to možné), jak byste kód upravili tak, aby mohl obsahovat i další běžné číselné typy (float, double, ...) a operace nad nimi.

Pro vypracování otázky si zvolte jeden z jazyků C#, C++, Java. Hodnotí se zejména použití vhodných nástrojů jazyka, drobné syntaktické detaily nejsou podstatné.

```
typedef std::map< std::string, int> Variables;

class AbstractNode {
public:
    virtual ~AbstractNode() {}
    virtual void print_prefix(std::ostream &) const = 0;
    virtual int evaluate(const Variables &) const = 0;
};

typedef std::unique_ptr< AbstractNode> NodePtr;

class BinaryNode : public AbstractNode {
protected:
    BinaryNode(NodePtr a, NodePtr b) : a_(std::move(a)), b_(std::move(b)) {}
private:
    NodePtr a_, b_;
    virtual void print_prefix(std::ostream & o) const
    {
        o << name() << " ";
        a_->print_prefix(o);
        o << " ";
        b_->print_prefix(o);
    }
    virtual int evaluate(const Variables & v) const
    {
        return fnc( a_->evaluate(v), b_->evaluate(v));
    }
    virtual std::string name() const = 0;
    virtual int fnc( int, int) const = 0;
};

class MinusNode : public BinaryNode {
public:
    MinusNode(NodePtr a, NodePtr b) : BinaryNode(std::move(a), std::move(b)) {}
```

```
    virtual std::string name() const
    { return "-"; }
    virtual int fnc( int x, int y) const
    { return x - y; }
};

class VariableNode : public AbstractNode {
public:
    VariableNode(std::string name) : name_(std::move(name)) {}
    virtual void print_prefix(std::ostream & o) const
    {
        o << name_;
    }
    virtual int evaluate(const Variables & v) const
    {
        auto it = v.find(name_);
        if (it == v.end()) throw std::domain_error("Unknown variable");
        return it->second;
    }
private:
    std::string name_;
};
```

Rozšíření na další číselné typy je vhodné implementovat pomocí šablon nebo generik, nikoliv okopírováním kódu.

## 20 Databáze SIS-Junior (specializace PVS)

Na jedné základní škole v ČR nasadili nový informační systém SIS-Junior pro evidenci žáků a jejich studijní agendy. V seznamu níže jsou uvedeny vybrané tabulky a sloupce, které se týkají dat potřebných pro tisk vysvědčení (uvažujeme tradiční vysvědčení, na kterém jsou předměty a známky, nikoli slovní hodnocení).

```
STUDENT(id, first_name:CHAR, last_name:CHAR, born:DATE)
CLASS(id, label:CHAR, academic_year:INT, school_year:INT{1..9})
SUBJECT(id, name:CHAR)
REPORT_GRADE(student_id, class_id, subject_id, semester:INT{1..2}, grade:INT{1..5})
```

Sloupce pojmenované `id` jsou PK (s hodnotou UUID), cizí klíče mají vždy tvar název-tabulky_id. Datové typy jsou naznačeny za názvy položek, u čísel (kde je to podstatné) jsou uvedeny i očekávané rozsahy. Školní rok se ukládá jako číslo roku, kdy začal (tj. 2023 odpovídá roku 2023/24).

1. Uvedené DB schéma zcela jistě není úplné. Doplňte sloupce a tabulky (včetně popisu jejich sloupců), které nejsou ve výpisu výše, abychom měli úplný popis části schématu odpovídající záměru použití (agenda vysvědčení), především aby bylo možné zcela splnit následující body.
2. Napište SQL dotaz typu `SELECT`, který vygeneruje podklady pro přípravu vysvědčení na druhé pololetí roku 2023/24 (tj. vypíše všechny platné trojice žák-třída-předmět, ke kterým pak učitelé budou ručně doplňovat známky a ukládat je do `REPORT_GRADE`).
3. Napište SQL dotaz typu `SELECT`, který vypíše průměrné známky pro každou unikátní kombinaci předmětu, ročníku studentů a školního roku (známky obou pololetí se průměrují dohromady).

**Nástin řešení** 1) Na první pohled chybí spojení student-class (je to běžný M:N vztah, takže tabulka) a subject-class (taky M:N, co se učí v které třídě). Také je možné uvést v subject-class tabulce i jestli se daný předmět učí jen v prvním, druhém, nebo obou pololetích.

```
STUDENT_IN_CLASS(student_id, class_id) ...PK jsou oba FK
SUBJECT_IN_CLASS(subject_id, class_id, semester) ...pokud je v obou semestrech, znamena to 2 zaznamy
```

2) Stačí začít s CLASS a spojit přes pomocné tabulky (protože nás zajímají jen IDčka studenta a předmětu).

```
SELECT student_in_class.student_id, class.id, subject_in_class.subject_id FROM class
JOIN student_in_class ON class.id = student_in_class.class_id
JOIN subject_in_class ON class.id = subject_in_class.class_id
WHERE class.academic_year = '2023' AND subject_in_class.semester = 2
```

3) Stačí jednoduchá kombinace JOIN a GROUP BY:

```
SELECT report_grade.subject_id, subject.name, class.academic_year, class.school_year,
  AVG(report_grade.grade)
FROM report_grade
JOIN class ON report_grade.class_id = class.id
JOIN subject ON report_grade.subject_id = subject.id
GROUP BY report_grade.subject_id, subject.name, class.academic_year, class.school_year
```

# 21 Robotický manipulátor (specializace PVS)

Mějme robotický manipulátor pro přesouvání předmětů na pracovním stole. Hlava manipulátoru se pohubuje ve 3D nad rovinou stolu a má na sobě připevněné chapadlo (grip), kterým může libovolný předmět uchopit. Manipulátor se ovládá jednoduchým textovým protokolem přes sériovou linku a rozlišuje následující příkazy:

- `X=123 Y=456 Z=789` — zahájí přesun manipulátoru na souřadnice $[123, 456, 789]$
- `STATUS` — vrátí informace o aktuální poloze a stavu manipulátoru
- `OPEN` — otevře robotické chapadlo (uvolní předmět)
- `CLOSE` — zavře robotické chapadlo (uchopí předmět)
- `STOP` — zastaví probíhající pohyb manipulátoru
- `HOME` — provede kalibraci manipulátoru na počáteční souřadnice

Na každý příkaz manipulátor odpoví `OK`, jakmile je dokončen, nebo `ERROR` (následovaný číslem a textovým popisem chyby) pokud dojde k chybě. Příkaz `STATUS` za indikátorem `OK` vrací navíc textově zakódovaný aktuální stav zařízení (poloha, stav chapadla). S manipulátorem je možné komunikovat po sériové lince, která je reprezentována třídou implementující následující interface:

```
interface ISerialLink {
    // sends one command over the link
    void writeCommand(string command);
    // returns last response (waits if no response is available)
    string readResponse();
}
```

Metoda `writeCommand` není blokující, pouze zapíše daný příkaz do fronty. Metoda `readResponse` může blokovat, pokud není žádná odpověď ve vstupním bufferu sériové linky. Je zaručeno, že příkazy (a tedy i odpovědi) budou zpracovány v pořadí, ve kterém byly zadány.

Ve vašem oblíbeném mainstreamovém objektovém jazyce (C++, C#, nebo Java) vyřešte následující (výše uvedený interface si případně upravte, aby odpovídal vybranému jazyku):

1. Navrhněte rozhraní pro třídu, která bude zapouzdřovat práci s robotickým manipulátorem. Návrh musí být rozumně efektivní, tj. metody by neměly blokovat volajícího, uživatel by měl mít lepší mechanismus než polling (periodické dotazování na stav) pro zjištění, že došlo k ukončení nějaké operace.
2. Načrtněte fragment kódu, který přesune věc pomocí robotického manipulátoru z místa A do místa B s použitím rozhraní z předchozího bodu. Obě místa jsou dána souřadnicemi, v místě A se věc uchopí, v místě B se věc uvolní z úchopu.
3. Napište alespoň jeden netriviální test pro třídu implementující rozhraní z prvního bodu (tj. test, který provede alespoň jeden přesun). Stručně popište, jak byste implementaci vašeho rozhraní mohli testovat bez samotného robotického manipulátoru (aby bylo možné testovat váš kód pomocí např. GitHub Actions, které běží ve virtuálním kontejneru a není možné k nim připojit náš hardware).

**Nástin řešení**

1. Každá operace bude mít samostatnou metodu, metody (alespoň přesun, OPEN, CLOSE, a HOME) by měly vracet něco, na co je možné se později dotázat (promise, future, task... dle vybraného jazyka), případně je možné použít async metody. Alternativa je mít nějaký event-emitting mechanismus, kam si může volající zaregistrovat callback/delegáta, který je zavolán v okamžiku, kdy nějaká probíhající operace skončí. Metoda pro STATUS operaci může blokovat a vrátit nějakou status strukturu. Metoda STOP nemusí vracet nic.
2. Tady závisí na rozhraní z předchozího bodu. Např. v moderním C# s použitím async metod lze psát normální sekvenční kód doplněný o await-y. V některých jazycích-knihovnách lze promises/futures řetězit (nechat zavolat další funkci, když se promise resolvuje).
3. Test musí mít hlavně smysluplný scénář (co se provede a co má být na výstupu) a verifikovat koncový stav (asserty). Druhou část otázky řeší mockup, tedy vlastní implementace `ISerialLink`, která emuluje činnost manipulátoru a kterou v testu použije implementace našeho rozhraní. Zde je užitečná dependency injection.

# 22 Obsluha zařízení a synchronizace (specializace SP)

Tato otázka navazuje na otázku o ovladači pro řadič disku ze společné části zkoušky.

1. Popište, zda a jakým způsobem by se muselo změnit rozhraní nabízené řadičem disku operačním systému tak, aby ovladač mohl místo aktivního čekání používat pasivní čekání a obsluhu přerušení. Popište krok po kroku, jak by v takovém případě probíhalo čtení bloku.
2. Upravte či nově načrtněte kód funkce pro čtení bloku a kód funkce pro obsluhu přerušení tak, aby se požadavky na čtení bloku evidovaly ve společné frontě a vlákna čekala na jejich vyřízení pasivně, podle popisu z předchozího bodu.
3. Vysvětlete, jak je ve vaší implementaci z předchozího bodu řešena synchronizace přístupu ke sdíleným proměnným ovladače, stačí řešení funkční na systému s jedním procesorem. Pokud vaše řešení předpokládá, že nějakou část synchronizace vyřeší externí funkce, popište také jak.

V odpovědi neřešte prolog a epilog obsluhy přerušení zodpovědný za uložení a obnovení registrů procesoru a případné potvrzení obsluhy přerušení, funkce pro obsluhu přerušení je volána se zakázaným přerušením a uschovanými registry procesoru. Předpokládejte existenci běžných funkcí pro synchronizaci, manipulaci stavu vlákna či manipulaci fronty požadavků, v řešení stačí uvést jejich signaturu a sémantiku (pokud není zřejmá z názvu).

**Nástin řešení**

1. Rozhraní se v principu změnit nemusí, stačí doplnit informaci o tom, že řadič při dokončení příkazu požádá o přerušení. Čtení bloku se pak zahájí nastavením čísla bloku na disku a adresy bloku v paměti a odesláním příkazu pro čtení. Vlákno se pak uspí a bude čekat na probuzení. To vykoná funkce pro obsluhu přerušení, která v minimálním provedení nemusí již dělat nic dalšího, ale typicky ještě vzbuzenému vláknu předá informaci o výsledku operace ze stavového registru řadiče.
2. 
```
typedef struct {
    size_t lba;
    uint32_t dma_phys_addr;
    thread_t sleeper;
    int status;
} request_t;

typedef struct {
    size_t max_lba;
    disk_regs_t *ctl;
    request_queue_t queue;
} disk_t;

bool disk_issue_request (disk_t *disk, request_t *request) {
    if (request->lba >= disk->max_lba) {
```

```
            return false;
        }
        bool ok = disk_wait_for_ready (disk);
        if (!ok) return false;

        disk->ctl->lba = request->lba;
        disk->ctl->dma = request->dma_phys_addr;
        disk->ctl->command = CMD_READ;

        return true;
    }

    bool disk_read_block_blocking (disk_t *disk, size_t lba, uint32_t dma_phys_addr) {
        request_t *request = (request_t *) malloc (sizeof (request_t));
        assert (request != NULL);
        request->lba = lba;
        request->dma_phys_addr = dma_phys_addr;
        request->sleeper = get_current_thread ();
        request->status = STATUS_WAIT;

        bool interrupts = push_disable_interrupts ();
        queue_append (&disk->queue, request);
        if (queue_head (&disk->queue) == request) {
            disk_issue_request (disk, request);
        }
        while (request->status == STATUS_WAIT) {
            thread_sleep ();
        }
        pop_interrupts (interrupts);
        bool result = (request->status == STATUS_OK);

        free (request);
        return result;
    }

    void disk_interrupt (disk_t *disk) {
        request_t *this_request = queue_pop (&disk->queue);
        this_request->status = disk_is_ok (disk) ? STATUS_OK : STATUS_FAIL;
        thread_wake (this_request->sleeper);
        request_t *next_request = queue_head (&disk->queue);
        if (next_request) {
            disk_issue_request (disk, next_request);
        }
    }
```

3. Řešení používá zákaz přerušení v těch částech kódu, kde by přepnutí kontextu nebo přerušení mohlo ohrozit sekvenci zápisů do registrů řadiče disku nebo přístup ke sdílené frontě požadavků. Funkce pro alokaci paměti jsou volané v běžném kontextu vlákna a tak mohou použít zámek.

## 23 Interní struktury systémů souborů (specializace SP)

Uvažujte soubor délky čtyř bloků (clusterů). Nakreslete jeho uložení v systému souborů FAT a ext (přesná varianta systému souborů není důležitá, zvolte libovolnou jednoduchou). V nákresech vyznačte relevantní obsah FAT tabulek a inode struktur a popište, jak se v každém systému souborů tento obsah najde podle jména souboru (předpokládejte, že máte k dispozici příslušnou položku adresáře, a napište, jak se z obsahu této položky vypočítá pozice relevantního obsahu FAT tabulky nebo inode struktury na disku). V nákresech si zvolte a používejte konkrétní čísla bloků.

Ve vašem řešení se můžete opřít o tuto (zjednodušenou) definici inode struktury ze zdrojových souborů jádra Linuxu (FAT

tabulka je prostě pole celých čísel):

```
struct ext2_inode {
    __le16  i_mode;             /* File mode */
    __le16  i_uid;              /* Low 16 bits of Owner Uid */
    __le32  i_size;             /* Size in bytes */
    __le32  i_atime;            /* Access time */
    ...
    __le16  i_gid;              /* Low 16 bits of Group Id */
    __le16  i_links_count;      /* Links count */
    __le32  i_blocks;           /* Blocks count */
    __le32  i_flags;            /* File flags */
    __le32  i_dir_block[12];    /* Pointers to direct blocks */
    __le32  i_ind_block;        /* Pointer to indirect block */
    __le32  i_dind_block;       /* Pointer to double indirect block */
    __le32  i_tind_block;       /* Pointer to triple indirect block */
};
```

**Nástin řešení** V systému souborů FAT stačí čtyři položky FAT tabulky. Pokud bude soubor (například) v clusterech 2, 3, 4 a 5, pak FAT bude na pozicích 2-5 obsahovat [3, 4, 5, EOF]. První relevantní položka FAT tabulky je uvedena v adresářové položce, její pozice na disku je `FAT_start_offset + FAT_first_block_index * FAT_entry_size // disk_block_size`.

V systému souborů ext stači jeden inode. Pokud bude soubor (například) v blocích 100, 101, 102 a 103, pak inode bude v seznamu bloků na pozicích 0-3 obsahovat [100, 101, 102, 103]. Číslo inode je uvedeno v adresářové položce, jako její pozice na disku stačí `inode_table_start_offset + inode_index * inode_size // disk_block_size`, případně o něco složitější `inode_table_start_offset + inode_index // inode_table_group_size * group_size + inode_index mod inode_table_g` pokud chceme uvažovat rozdělení systému souborů na groups.

# 24 Sockety (specializace SP)

Napište implementaci webového serveru, který na každé příchozí spojení (bez čekání na HTTP GET request a obecně bez kontroly korektnosti HTTP požadavku ze strany klienta) reaguje odesláním HTML stránky s obsahem <HTML><BODY>Hello !</BODY></HTML>.

Použijte TCP sockety, signatury relevantních systémových funkcí (abecedně) jsou:

```
int accept (int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int bind (int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int close (int fd);
int listen (int sockfd, int backlog);
ssize_t read (int fd, void buf, size_t count);
ssize_t recv (int sockfd, void buf, size_t len, int flags);
ssize_t send (int sockfd, const void buf, size_t len, int flags);
int shutdown (int sockfd, int how);
int socket (int domain, int type, int protocol);
ssize_t write (int fd, const void buf, size_t count);
```

V řešení se zaměřte hlavně na celkovou strukturu implementace a správné pořadí volání jednotlivých funkcí. Přesné hodnoty parametrů jako `flags` nejsou nutné, použijte vhodné symbolické konstanty. Podobně nemusíte řešit chybové stavy.

```
int main ()
{
    int server_socket = socket (AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in server_address;
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = htonl (INADDR_ANY);
    server_address.sin_port = htons (HTTP_PORT);
```

```
    bind (server_socket, (struct sockaddr *) &server_address, sizeof (server_address));
    listen (server_socket, 0);

    while (true) {
        struct sockaddr_in client_address;
        socklen_t client_address_size = sizeof (client_address);
        int client_socket = accept (server_socket, (struct sockaddr *) &client_address, &client_address_size)
        char *page = "<HTML><BODY>Hello␣!</BODY></HTML>";
        write (client_socket, page, strlen (page));
        shutdown (client_socket, SHUT_RDWR);
        close (client_socket);
    }
}
```

## 25 Jednotkové testování software (specializace SP)

Následující fragment kódu je zjednodušený jednotkový test používaný v knihovně Apache Commons Collections na ověření fungování metody `add` kolekcí:

```
public abstract class AbstractCollectionTest<E> {
    @Test
    public void testCollectionAdd () {
        final E[] elements = getSomeElements ();
        for (final E element : elements) {
            resetCollection ();
            getCollection ().add (element);
            assertEquals (1, getCollection ().size (), "...");
            assertTrue (getCollection ().contains (element), "...");
        }
        resetCollection ();
        int size = 0;
        for (final E element : elements) {
            getCollection ().add (element);
            size++;
            assertEquals (size, getCollection ().size (), "...");
            assertTrue (getCollection ().contains (element), "...");
        }
    }
    ...
}
```

1. Identifikujte, jaké vlastnosti metody `add` test ověřuje, a napište vhodné chybové hlášky u jednotlivých volání `assert`.
2. Uvažujte kolekci reprezentující seznam prvků typu `E` a rozhodněte, zda by daná metoda ověřila všechny typicky deklarované vlastnosti metody `add`, nebo zda by jí bylo vhodné či nutné opravit (a v takovém případě napište kód ilustrující takovou úpravu).
3. Uvažujte kolekci reprezentující množinu prvků typu `E` a rozhodněte, zda by daná metoda ověřila všechny typicky deklarované vlastnosti metody `add`, nebo zda by jí bylo vhodné či nutné opravit (a v takovém případě napište kód ilustrující takovou úpravu).

**Nástin řešení**

1. Test ověřuje, že přidáním do prázdné kolekce vznikne jednoprvková kolekce, která obsahuje přidávaný prvek, a že postupným přidáváním prvků do kolekce příslušně roste velikost kolekce, která obsahuje všechny dosud přidávané prvky. Chybové hlášky by měly odrážet podstatu testu, tedy "adding one element to empty collection should yield a collection of size 1" je vhodnější než "collection size is not 1".
2. U seznamu je důležité pořadí prvků, tedy by bylo vhodné doplnit test o ověření, že přidávaný prvek je na konci seznamu.
3. Množina nemůže obsahovat shodné prvky, tedy by bylo nutné upravit test o ověření, že při opakovaném vložení stejného prvku se nezmění velikost.

# 26 Statistické testy – t-test (specializace UI-SU)

Výrobce sušenek vyrábí sušenky s deklarovanou hmotností 100 gramů. Při kontrole kvality bylo náhodně vybráno a převáženo 100 sušenek, které měly průměrnou hmotnost 102 gramů se směrodatnou odchylkou 2 gramy. Pomocí t-testu chceme statisticky vyhodnotit, jestli se průměrná hmotnost sušenek liší od deklarované hmotnosti.

1. Jaká je nulová a alternativní hypotéza jednovýběrového t-testu?
2. Spočítejte hodnotu testové statistiky pro data zadaná výše. Jaké má tato testová statistika pravděpodobnostní rozdělení?
3. Liší se statisticky významně hmotnost sušenek od deklarované hmotnosti? *Nápověda:* Kritická hodnota testové statistiky na hladině významnosti $\alpha = 0.05$ je cca 2.

**Nástin řešení**

1. Nulová hypotéza je, $H_0 : \bar{X} = \mu$, kde $\bar{X}$ je průměrná hmotnost sušenek a $\mu$ je testovaná střední hodnota. Alternativní hypotéza je $H_1 : \bar{X} \neq \mu$.
2. $t = \frac{102-100}{2/\sqrt{100}} = 10$, má t rozdělení s 99 stupni volnosti
3. Spočítaná $t$-hodnota je větší než kritická hodnota, zamítáme tedy nulovou hypotézu. Průměrná hmotnost sušenek se liší od deklarované hmotnosti.

# 27 Lineární regrese a regularizace (specializace UI-SU)

Chceme natrénovat lineární model tvaru $y = \beta_1 x_1 + \beta_2 x_2 + \beta_0$ na datech níže. Při trénování chceme použít gradientní metodu a L2 regularizaci. Použijeme učící konstantu $\alpha = 0.1$ a regularizační konstantu $\lambda = 0.5$.

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 2 | 3 |
| 2 | 3 | 9 |
| 1 | 1 | 2 |
| 2 | 0 | 2 |

1. Definujte chybovou funkci pro model zadaný podle parametrů nahoře (součet čtverců s L2 regularizačním členem).
2. Předpokládejme, že aktuální odhad koeficientů modelu je $\beta_1 = 1$, $\beta_2 = 3$, $\beta_0 = -1$. Proveďte jeden krok gradientní metody pro nastavení koeficientů. Jaké jsou jejich nové hodnoty?

**Nástin řešení**

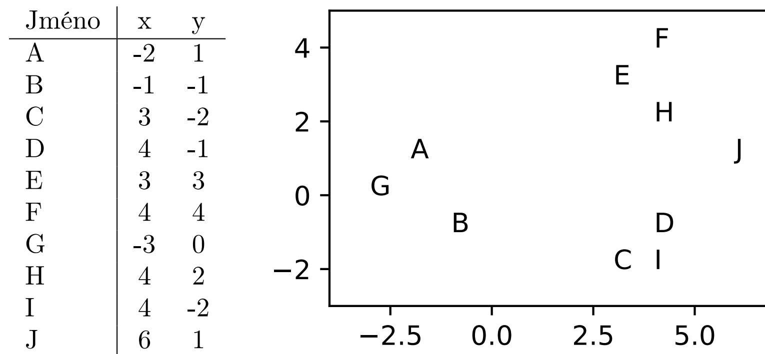
1. Chybová funkce je typicky RSS $= \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$, pro L2 regularizaci přičteme ještě člen $\lambda \sum_{i=1}^{m} \beta_i^2$ ($N$ je počet trénovacích vzorů, $m+1$ je počet koeficientů modelu, $\hat{y}_i$ je výstup modelu pro $i$-tý vstup). Koeficient $\beta_0$ se většinou v regularizaci neuvažuje.
2. Je potřeba spočítat gradient chybové funkce podle jednotlivých parametrů. Spočítáme ho pro jeden trénovací vzor, pro všechny vzory to bude jen součet. Derivace $\frac{dRSS}{d\beta_0} = -2(y_i - \hat{y}_i)$, derivace $\frac{d\text{RSS}}{d\beta_i} = -2(y_i - \hat{y}_i)x_i$. Nesmíme zapomenout také na derivaci regularizačního členu podle $\beta_i$ – ta je $\lambda 2\beta_i = \beta_i$. Dosazením dostaneme, že derivace RSS podle $\beta_0$ je

4 + 2 + 2 + (-2) = 6, derivace podle $\beta_1$ je 2 a derivace podle $\beta_2$ je 16. Přičtením regularizačního členu dostaneme, že derivace chybové funkce podle $\beta_1$ je 3 a podle $\beta_2$ je 19. Hodnoty v dalším kroku tedy jsou $\beta_0 = \beta_0 - \alpha \cdot 6 = -1.6$, $\beta_1 = \beta_1 - \alpha \cdot 3 = 0.7$ a $\beta_2 = \beta_2 - \alpha \cdot 19 = 1.1$.

# 28 Shlukování (specializace UI-SU)

Máme data zadaná v tabulce níže a graficky znázorněná v obrázku. Chtěli bychom je rozdělit do třech shluků pomocí algoritmu $k$-means.

*Poznámka:* Jména nejsou součástí dat – slouží jen pro vizualizaci a pro zjednodušení formulace odpovědí.



| Jméno | x | y |
|---|---|---|
| A | -2 | 1 |
| B | -1 | -1 |
| C | 3 | -2 |
| D | 4 | -1 |
| E | 3 | 3 |
| F | 4 | 4 |
| G | -3 | 0 |
| H | 4 | 2 |
| I | 4 | -2 |
| J | 6 | 1 |



1. Popište stručně hlavní kroky algoritmu $k$-means.
2. Předpokládejme, že v první iteraci jsme náhodně vybrali body $A$, $F$ a $J$ jako počáteční středy shluků. Jaké bude rozdělení bodů do shluků po dokončení první iterace? Jaké budou středy těchto shluků? Při výpočtech používejte Euklidovskou vzdálenost.
3. Změní se nějak středy shluků v dalších iteracích algoritmu?

**Nástin řešení**

1. Algoritmus napřed náhodně zvolí počáteční středy shluků. Následně opakuje dva kroky: přiřazení každého bodu k nejbližšímu středu shluku a přepočítání polohy středu (jako průměr bodů patřících do daného shluku).
2. V prvním shluku budou body $\{A, G, B\}$, ve druhém shluku budou body $\{E, F, H\}$, ve třetím budou body $\{J, D, I, C\}$. Nové středy budou průměry těchto bodů $(-2, 0)$, $(\frac{11}{3}, 3)$ a $(\frac{17}{4}, -1)$.
3. Přiřazení do shluků už se v další iteraci nezmění, středy shluků tedy také ne.

# 29 Hluboké učení v NLP (specializace UI-ZPJ)

1. Popište základní techniky pro trénování word embeddings (skipgram, CBOW).
2. Popište základní modely rekurentních sítí používaných při zpracovnání přirozeného jazyka (standardní rekurentní sítě, LSTM). Jaké problémy vznikají při trénování standardních rekurentních sítí? Jak je LSTM sítě řeší?

**Nástin řešení**

1. Principem CBOW je předpovědět prostřední slovo na základě okolního kontextu. Principem skipgram je opak – předpovědět okolní slova na základě prostředního slova.
2. Základním problémem s trénováním standardních rekurentních sítí jsou tzv. explodující a mizející gradienty. LSTM sítě tento problém řeší tím, že jednotlivé neurony nahrazují LSTM buňkami, které explicitně pracují s vnitřním stavem.

3. Přesné vzorce pro obě podotázky lze najít například v 9. přednášce z předmětu NPFL124 (Zpracování přirozeného jazyka).

# 30 Lineární regrese a regularizace (specializace UI-ZPJ)

Chceme natrénovat lineární model tvaru $y = \beta_1 x_1 + \beta_2 x_2 + \beta_0$ na datech níže. Při trénování chceme použít gradientní metodu a L2 regularizaci. Použijeme učící konstantu $\alpha = 0.1$ a regularizační konstantu $\lambda = 0.5$.

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 2 | 3 |
| 2 | 3 | 9 |
| 1 | 1 | 2 |
| 2 | 0 | 2 |

1. Definujte chybovou funkci pro model zadaný podle parametrů nahoře (součet čtverců s L2 regularizačním členem).
2. Předpokládejme, že aktuální odhad koeficientů modelu je $\beta_1 = 1$, $\beta_2 = 3$, $\beta_0 = -1$. Proveďte jeden krok gradientní metody pro nastavení koeficientů. Jaké jsou jejich nové hodnoty?

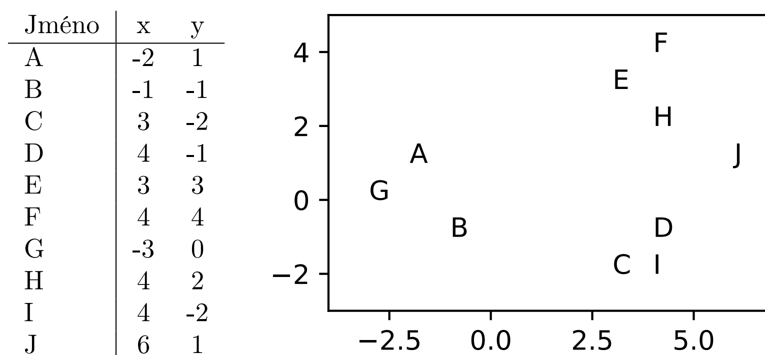
**Nástin řešení**

1. Chybová funkce je typicky RSS $= \sum_{i=1}^{N}(y_i - \hat{y}_i)^2$, pro L2 regularizaci přičteme ještě člen $\lambda \sum_{i=1}^{m} \beta_i^2$ ($N$ je počet trénovacích vzorů, $m+1$ je počet koeficientů modelu, $\hat{y}_i$ je výstup modelu pro $i$-tý vstup). Koeficient $\beta_0$ se většinou v regularizaci neuvažuje.
2. Je potřeba spočítat gradient chybové funkce podle jednotlivých parametrů. Spočítáme ho pro jeden trénovací vzor, pro všechny vzory to bude jen součet. Derivace $\frac{dRSS}{d\beta_0} = -2(y_i - \hat{y}_i)$, derivace $\frac{d\text{RSS}}{d\beta_i} = -2(y_i - \hat{y}_i)x_i$. Nesmíme zapomenout také na derivaci regularizačního členu podle $\beta_i$ – ta je $\lambda 2\beta_i = \beta_i$. Dosazením dostaneme, že derivace RSS podle $\beta_0$ je 4 + 2 + 2 + (-2) = 6, derivace podle $\beta_1$ je 2 a derivace podle $\beta_2$ je 16. Přičtením regularizačního členu dostaneme, že derivace chybové funkce podle $\beta_1$ je 3 a podle $\beta_2$ je 19. Hodnoty v dalším kroku tedy jsou $\beta_0 = \beta_0 - \alpha \cdot 6 = -1.6$, $\beta_1 = \beta_1 - \alpha \cdot 3 = 0.7$ a $\beta_2 = \beta_2 - \alpha \cdot 19 = 1.1$.

# 31 Shlukování (specializace UI-ZPJ)

Máme data zadaná v tabulce níže a graficky znázorněná v obrázku. Chtěli bychom je rozdělit do třech shluků pomocí algoritmu $k$-means.

*Poznámka:* Jména nejsou součástí dat – slouží jen pro vizualizaci a pro zjednodušení formulace odpovědí.



| Jméno | x | y |
|---|---|---|
| A | -2 | 1 |
| B | -1 | -1 |
| C | 3 | -2 |
| D | 4 | -1 |
| E | 3 | 3 |
| F | 4 | 4 |
| G | -3 | 0 |
| H | 4 | 2 |
| I | 4 | -2 |
| J | 6 | 1 |



1. Popište stručně hlavní kroky algoritmu $k$-means.
2. Předpokládejme, že v první iteraci jsme náhodně vybrali body $A$, $F$ a $J$ jako počáteční středy shluků. Jaké bude rozdělení bodů do shluků po dokončení první iterace? Jaké budou středy těchto shluků? Při výpočtech používejte Euklidovskou vzdálenost.

3. Změní se nějak středy shluků v dalších iteracích algoritmu?

**Nástin řešení**

1. Algoritmus napřed náhodně zvolí počáteční středy shluků. Následně opakuje dva kroky: přiřazení každého bodu k nejbližšímu středu shluku a přepočítání polohy středu (jako průměr bodů patřících do daného shluku).
2. V prvním shluku budou body $\{A, G, B\}$, ve druhém shluku budou body $\{E, F, H\}$, ve třetím budou body $\{J, D, I, C\}$. Nové středy budou průměry těchto bodů $(-2, 0)$, $(\frac{11}{3}, 3)$ a $(\frac{17}{4}, -1)$.
3. Přiřazení do shluků už se v další iteraci nezmění, středy shluků tedy také ne.

## 32 Informované prohledávání (specializace UI-SU, UI-ZPJ)

Uvažme hlavolam na obrázku níže. Jedná se o mřížku $3 \times 3$ s 8 označenými kostičkami a jedním volným místem. Naším cílem je ze zadaného počátečního stavu uspořádat kostičky do předem daného cílového stavu, ve kterém jsou kostičky uspořádány od 1 do 8 zleva doprava, shora dolů. V každém kroku můžeme posunout právě jednu kostičku na sousední volné místo. Navíc chceme provést minimální možný počet kroků.

1. Formalizujte zadaný problém jako informovaný prohledávací problém. Speciálně popište, jak vypadá stav, jaký je přechodový model a jaký je test cílového stavu.
2. Pro vyřešení definovaného problému navrhněte optimální informovaný prohledávací algoritmus. Popište jeho průběh (například pomocí pseudokódu).
3. Navrhněte heuristickou funkci pro výše zmíněný hlavolam. Jaké vlastnosti má vaše heuristická funkce? Zaručuje použití této heuristiky s algoritmem z předchozího bodu nalezení optimálního řešení? *Nápověda: Napsali jste algoritmus jako tree search nebo graph search?*

**Nástin řešení**

1. Stav je rozložení kostiček, přechodový model je pohnutí jednou kostičkou, čímž dostanu jiné rozložení (nezapomenout na cenu akce = 1), kontrola cílového stavu zkontroluje, zda se rozložení shoduje se zadaným cílem.
2. A*, je jedno jestli graph search nebo tree search.
3. Například taxicab distance. U graph search musí být heuristika monotonní/konzistentní. U tree search stačí přípustnost/spodní odhad. Všechny rozumné heuristiky budou splňovat oboje.