6. prednáška (21.3.2022)

Backtracking

alebo

hrubou silou
na (skoro) všetky problémy

Začnime zľahka...

- **Problém:**
  - vygenerovať všetky postupnosti dĺžky 3 z čísel 1, 2, 3
    - matematicky: 3-prvkové variácie s opakovaním z prvkov množiny $\{1, 2, 3\}$
- **Očakávaný výstup:**

```
[1, 1, 1]    [2, 1, 1]    [3, 1, 1]
[1, 1, 2]    [2, 1, 2]    [3, 1, 2]
[1, 1, 3]    [2, 1, 3]    [3, 1, 3]
[1, 2, 1]    [2, 2, 1]    [3, 2, 1]
[1, 2, 2]    [2, 2, 2]    [3, 2, 2]
[1, 2, 3]    [2, 2, 3]    [3, 2, 3]
[1, 3, 1]    [2, 3, 1]    [3, 3, 1]
[1, 3, 2]    [2, 3, 2]    [3, 3, 2]
[1, 3, 3]    [2, 3, 3]    [3, 3, 3]
```

Generovanie trojíc

```
int[] p = new int[3];
for (int i = 1; i <= 3; i++) {
  p[0] = i;
  for (int j = 1; j <= 3; j++) {
    p[1] = j;
    for (int k = 1; k <= 3; k++) {
      p[2] = k;
      System.out.println(Arrays.toString(p));
    }
  }
}
```

Ďalšie výzvy:

- Ako zmeniť množinu z $\{1, 2, 3\}$ na $\{1, 2, \ldots, 5\}$?
- Ako zmeniť množinu z $\{1, 2, 3\}$ na $\{0, 1, 2\}$?
- Čo ak by sme chceli 4-prvkové postupnosti čísel?
- Čo ak by sme chceli 5-prvkové postupnosti čísel?

Generovanie k-tíc

- Čo ak by sme chceli k-prvkové postupnosti z čísel $\{1, 2, 3\}$, t.j. k-prvkové variácie s opakovaním, kde k dopredu nepoznáme?
  - pridávanie vnorených cyklov nepomôže, pretože k nepoznáme dopredu...

Generovanie k-tíc je ťažký problém ...

... existuje jednoduchší podproblém?

Analýza trojíc...

[1, 1, 1]	[2, 1, 1]	[3, 1, 1]
[1, 1, 2]	[2, 1, 2]	[3, 1, 2]
[1, 1, 3]	[2, 1, 3]	[3, 1, 3]
[1, 2, 1]	[2, 2, 1]	[3, 2, 1]
[1, 2, 2]	[2, 2, 2]	[3, 2, 2]
[1, 2, 3]	[2, 2, 3]	[3, 2, 3]
[1, 3, 1]	[2, 3, 1]	[3, 3, 1]
[1, 3, 2]	[2, 3, 2]	[3, 3, 2]
[1, 3, 3]	[2, 3, 3]	[3, 3, 3]

Každá skupina „trojíc" sa skladá zo všetkých dvojíc z čísel {1, 2, 3}

Ako generovať trojice?

- Vygenerujme najprv všetky **dvojice**...
  - a pred každú dvojicu najprv pridáme 1, potom 2 a potom 3...

	1 + [...]	2 + [...]	3 + [...]
[1, 1]	[1, 1, 1]	[2, 1, 1]	[3, 1, 1]
[1, 2]	[1, 1, 2]	[2, 1, 2]	[3, 1, 2]
[1, 3]	[1, 1, 3]	[2, 1, 3]	[3, 1, 3]
[2, 1]	[1, 2, 1]	[2, 2, 1]	[3, 2, 1]
[2, 2]	[1, 2, 2]	[2, 2, 2]	[3, 2, 2]
[2, 3]	[1, 2, 3]	[2, 2, 3]	[3, 2, 3]
[3, 1]	[1, 3, 1]	[2, 3, 1]	[3, 3, 1]
[3, 2]	[1, 3, 2]	[2, 3, 2]	[3, 3, 2]
[3, 3]	[1, 3, 3]	[2, 3, 3]	[3, 3, 3]

Generovanie v praxi

Ako generovať k-tice?

k-prvková postupnosť z čísel $\{1, 2, 3\}$

?	2	…	1	2	3	3

$(k\text{-}1)$-prvková postupnosť z čísel $\{1, 2, 3\}$

Postupným dosadením čísiel 1, 2 a 3 dostaneme 3 rôzne k-prvkové postupnosti z čísel $\{1, 2, 3\}$

Ako generovať k-tice?

Ako vygenerovať všetky
k-prvkové postupnosti z čísel $\{1, 2, 3\}$?

	…	…	…	…	…	…

Vygeneruj všetky $(k\text{-}1)$-prvkové
postupnosti z čísel $\{1, 2, 3\}$…

{1, 2, 3}

Generovanie trojíc

```java
int[] p = new int[3];

for (int i = 1; i <= 3; i++) {                generuj(0)
  p[0] = i;

  for (int j = 1; j <= 3; j++) {              generuj(1)
    p[1] = j;

    for (int k = 1; k <= 3; k++) {            generuj(2)
      p[2] = k;

      System.out.println(Arrays.toString(p));
    }
  }
}
                                        báza rekurzie
```

Generovanie k-tíc

generuj všetky postupnosti v podpoli určenom indexami: `odIndexu, …, p.length-1`

```
private int[] p;

private void generuj(int odIndexu) {
    if (odIndexu == p.length) {
        vypis();
        return;
    }

    for (int i = 1; i <= 3; i++) {
        p[odIndexu] = i;
        generuj(odIndexu + 1);
    }
}

public void generuj() {
    generuj(0);
}
```

Ak máme generovať postupnosť v podpoli dĺžky 0, znamená to, že pole je naplnené...

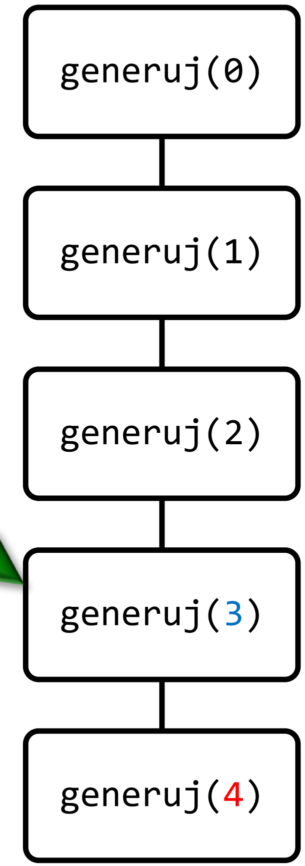
Na prvú pozíciu podpoľa postupne dáme hodnoty *1*, *2* a *3*. Po dosadení každej z nich spustíme generovanie všetkých podpostupností v o 1 menšom podpoli.

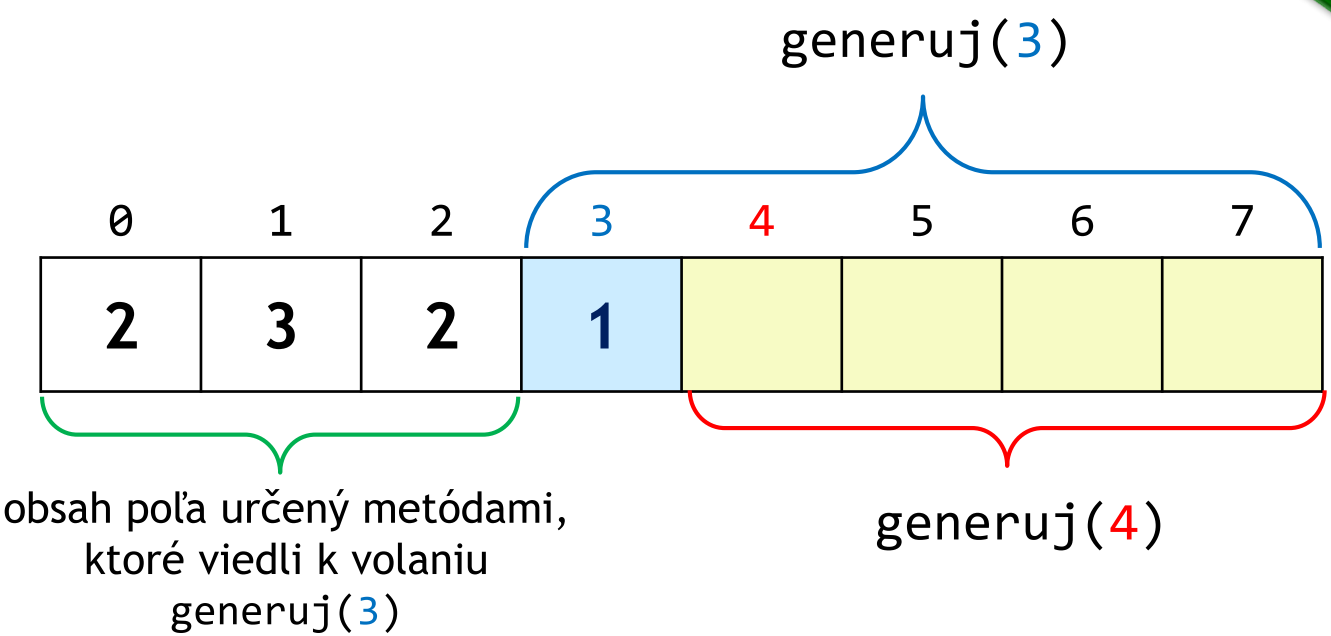
Začíname generovanie na indexe 0, t.j. v celom poli.

Ako to vlastne generuje?

```
private void generuj(int odIndexu) {
    ...
    for (int i = 1; i <= 3; i++) {
        p[odIndexu] = i;
        generuj(odIndexu + 1);
    }
}
```

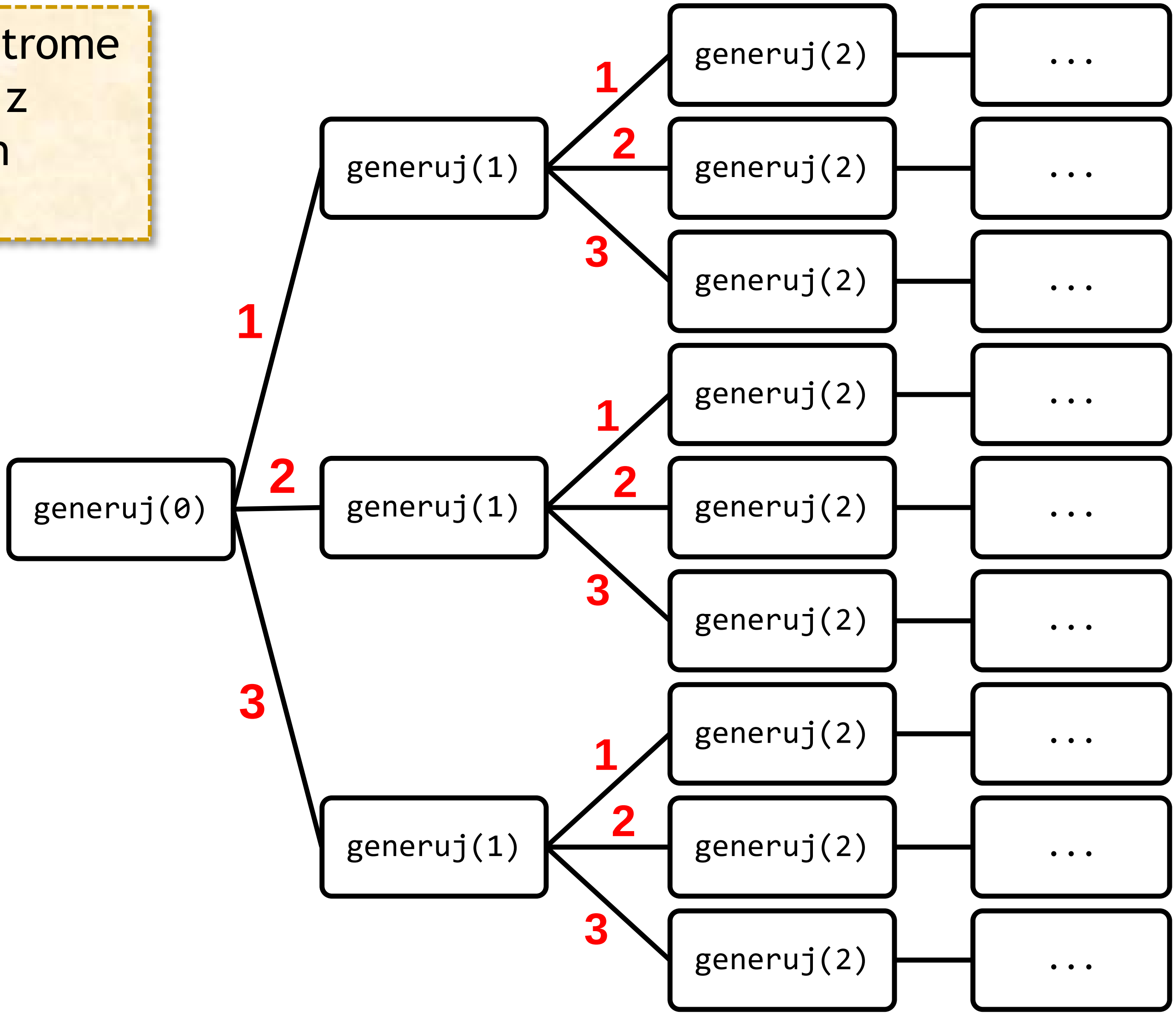
určuje obsah na indexe 3 a nechá generovať obsah od indexu 4





Strom volaní

Každá vetva v strome volaní je jedna z vygenerovaných postupností...



Na čo je to dobré?

Koho už len zaujíma generovanie číselných postupností?!

Problém batoha

Trezor s cennosťami

Každá vec v trezore má svoju **veľkosť** a **cenu**...

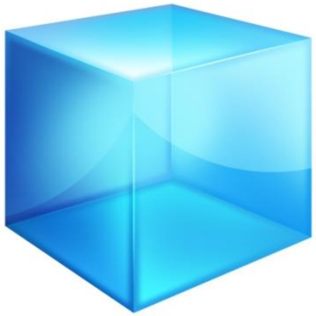
Zlodej s batohom

Kapacita batoha je **obmedzená!**

Ktoré veci zobrať, aby cena lupu bola čo najväčšia a zároveň sa to pomestilo do batoha?

Ako naplniť batoh?

Kapacita batoha: 4

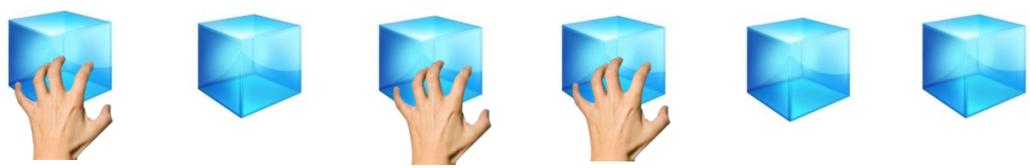
	12 €	7 €	7 €
			
Veľkosť:	3	2	2

- Jednoduché stratégie nefungujú:
  - kým je miesto, ber najdrahšiu vec, ktorú ešte nemáš
  - kým je miesto, ber vec s najlepším pomerom cena/veľkosť (jednotková cena), ktorú ešte nemáš

Riešenie hrubou silou!

- Máme n predmetov
- S každým predmetom sú dve možnosti:
  - predmet **zoberiem**
  - predmet **nezoberiem**
- Každý **výber** možno charakterizovať postupnosťou z núl a jednotiek...

Zmestí sa výber **do batoha?**



1 0 1 1 0 0

Aká je **cena výberu**?

Beriem (1)

Neberiem (0)

Riešenie hrubou silou!

- Máme n predmetov:
  - vygenerujem všetky postupnosti dĺžky n z 0 a 1 = n-prvkové **variácie s opakovaním** z prvkov množiny $\{0 = neberiem, 1 = beriem\}$
  - každá **postupnosť zodpovedá výberu**, pre ktorý spočítam:
    - či sa zmestí do batoha (súčet veľkostí vybraných predmetov je menší ako kapacita batoha)
    - aká je cena výberu (súčet cien vybraných predmetov)
- **Riešenie:**
  - zoberiem výber, ktorý sa zmestí do batoha a má najlepšiu cenu

Programujeme...

Aká je efektivita riešenia?

- n predmetov = 2^n možných výberov
- spracovanie jedného výberu $O(n)$
- celkový čas: $2^{O(n)}$

Verí sa (P ?= NP), že túto úlohu nejde riešiť rýchlejšie

- 60 predmetov = 2^{60} výberov > 10^{17} výberov
  - 1 GHz procesor ~ 10^9 operácii za sekundy
  - $10^{17}/10^9 = 10^8$ sekúnd > 3 roky

Riešenie hrubou silou!

- Problémy riešime **preskúmaním všetkých možností riešenia**

- Často aplikujeme **schému**:
  - `generuj` – generuje všetky postupnosti zadanej dĺžky, ktoré zodpovedajú nejakej možnosti riešenia
  - `spracuj` – overí, či vygenerovaná možnosť je **prípustná** (*napr. celková veľkosť je menšia ako kapacita*) a ak áno, možnosť sa nejako **dodatočne spracuje** (*napr. overí sa, či nie je lepšia ako doposiaľ najlepšie nájdené riešenie*)

A čo ďalej?

A poďme generovať opäť nejaké číselné postupnosti...

Variácie bez opakovania

- **Úloha:** Vygenerovať všetky postupnosti dĺžky k z prvkov množiny $\{1, \ldots, n\}$, v ktorých sa ale žiadne číslo neopakuje.
  - matematicky: generujeme ***k*-prvkové variácie bez opakovania** z n-prvkovej množiny

- **Riešenie:**
  - vygenerujeme všetky postupnosti dĺžky k
  - pre každú postupnosť overíme, či tam náhodou nie sú dve rovnaké čísla – ak nie, tak ju „vypíšeme“

Generujeme bez opakovania

```
private boolean vyhovuje() {
  for (int i = 0; i < p.length; i++)
    for (int j = i + 1; j < p.length; j++)
      if (p[i] == p[j])
        return false;

  return true;
}
```

Overí, či sú všetky čísla v poli rôzne.

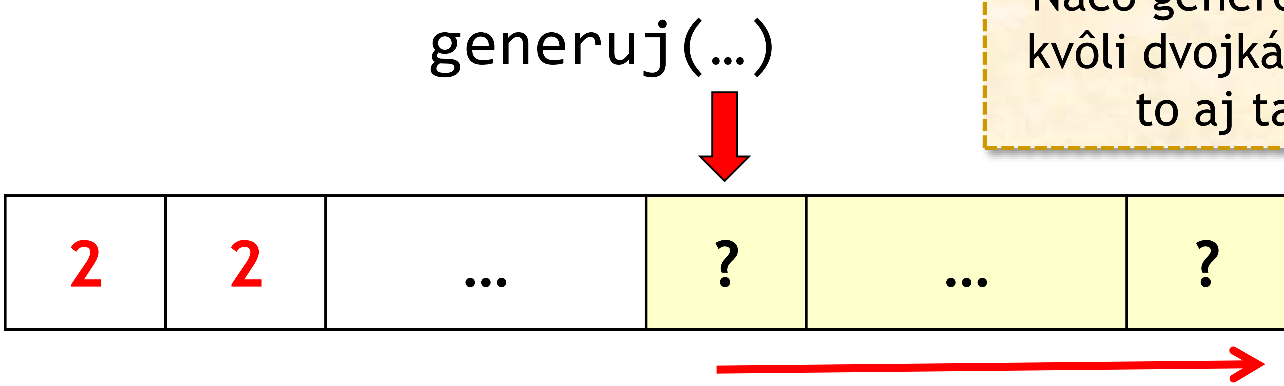
```
private void generuj(int odIndexu) {
  if (odIndexu == p.length) {
    if (vyhovuje())
      vypis();
    return;
  }

  for (int i = 1; i <= n; i++) {
    p[odIndexu] = i;
    generuj(odIndexu + 1);
  }
}
```

Pred výpisom overíme, že niet opakovaní.

Ako znížiť straty?

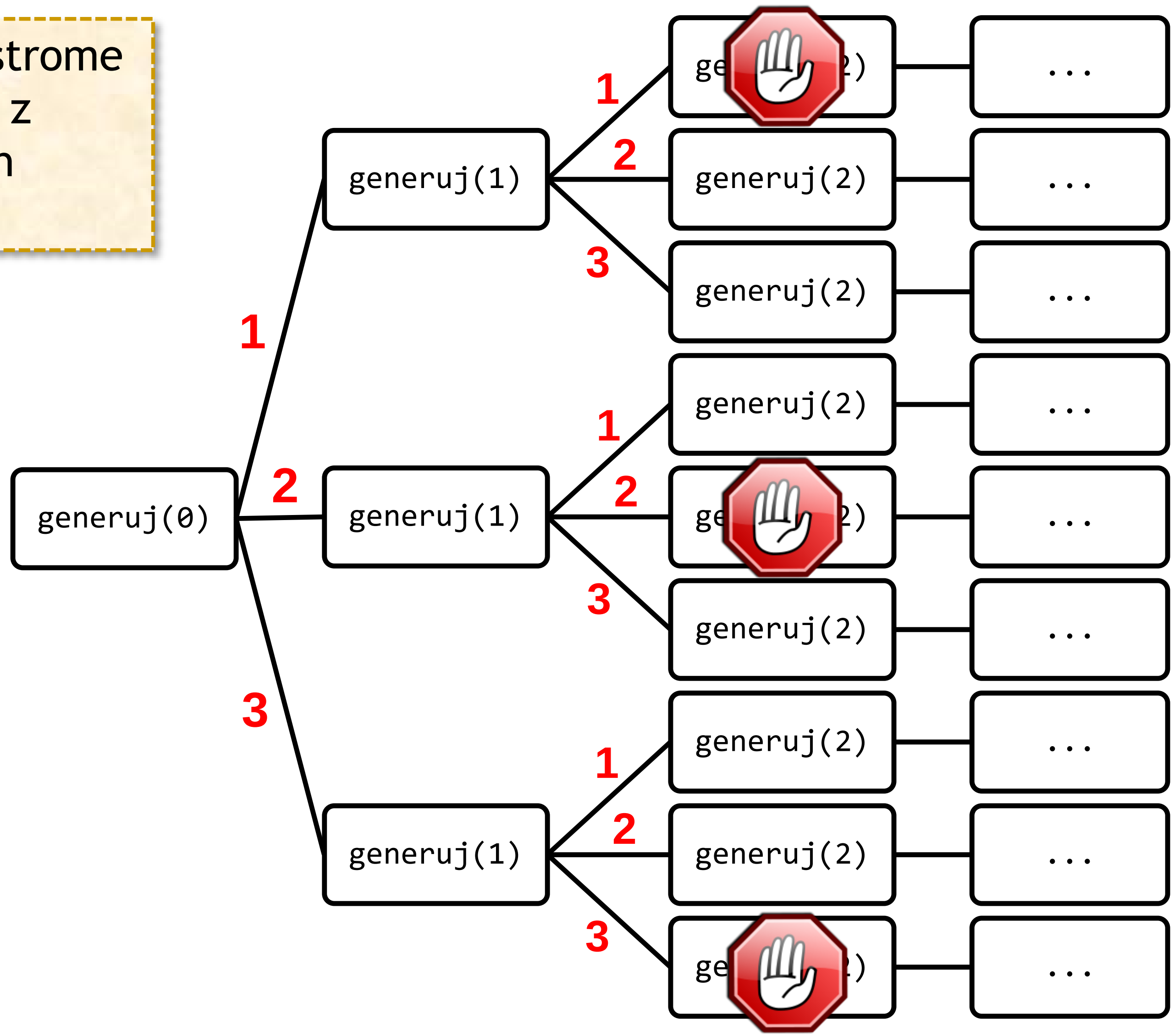
- Metóda `vyhovuje()` filtruje, čo sa „vypíše“...
- Postupnosti bez opakovania dĺžky 3 z prvkov množiny $\{1, 2, 3, 4\}$
  - počet „vypísaných“: $4.3.2 = 24$ postupností
  - počet vygenerovaných: $4^3 = 64$ postupností
  - … generujeme 40 postupností zbytočne („nevypíšu“ sa)

Način generovať zvyšok, keď kvôli dvojkám na začiatku sa to aj tak nevypíše?



Strom volaní

Každá vetva v strome volaní je jedna z vygenerovaných postupností...



Opatrné generovanie...

```
private boolean moze(int idx, int hodnota) {
  for (int i=0; i<idx; i++)
    if (p[i] == hodnota)
      return false;

    return true;
}
```

Overí, či sa zadaná hodnota nachádza v poli na indexoch `0,…,idx-1`

```
private void generuj(int odIndexu) {

  ...

for (int i = 1; i <= n; i++)
  if (moze(odIndexu, i)) {
    p[odIndexu] = i;
    generuj(odIndexu + 1);
  }
}
```

Hodnotu **i** umiestnime do poľa iba vtedy, keď sa ešte v poli „naľavo“ nenachádza.

Opatrné generovanie rýchlo...

- Overenie, či číslo môžeme umiestniť do poľa (metóda `moze`), má časovú zložitosť $O(k)$
- **Zrýchlenie**: pamätajme si v poli, či sa číslo už nachádza v „ľavej“ fixovanej časti...

```
private Set<Integer> pouzite = new ???Set<Integer>()

for (int i = 1; i <= n; i++) {
  if (!pouzite.contains(i)) {
    pouzite.add(i);
    p[odIndexu] = i;
    generuj(odIndexu + 1);
    p[odIndexu] = -1;
    pouzite.remove(i);
  }
}
```

Poznačíme, že sme pridali do poľa číslo `i` ...

Poznačíme, že číslo `i` už viac nie je v platnej časti poľa ...

Opatrné generovanie rýchlo...

- Overenie, či číslo môžeme umiestniť do poľa (metóda `moze`), má časovú zložitosť $O(k)$
- **Zrýchlenie**: pamätajme si v poli, či sa číslo už nachádza v „ľavej" fixovanej časti...

```
for (int i = 1; i <= n; i++) {
  if (!pouzite[i]) {
    pouzite[i] = true;
    p[odIndexu] = i;
    generuj(odIndexu + 1);
    p[odIndexu] = -1;
    pouzite[i] = false;
  }
}
```

Poznačíme, že sme pridali do poľa číslo i ...

Poznačíme, že číslo i už viac nie je v platnej časti poľa ...

Variácie bez opakovania - finty

- **Permutácie** = n-prvkové variácie bez opakovania prvkov z n-prvkovej množiny...
- Ako generovať **variácie z inej množiny** ako sú prvky množiny $\{1, \ldots, n\}$?
  - prvky množiny dáme do nejakého poľa - napr. `prvky`
  - generujeme variácie indexov poľa `prvky`
  - výpis:

```
for (int i=0; i<p.length; i++)
  variacia[i] = prvky[p[i]]
```

Generujeme postupnosť indexov v poli `prvky`.

Backtracking

- Backtracking = **prehľadávanie s návratom**
- Spôsob na hľadanie/konštrukciu riešenia tak, že skúmame (generujeme) všetky možnosti
- Zvyčajná **schéma**:
  - sprav jeden krok „bližšie“ ku skonštruovaniu riešenia
    - pridanie hodnoty do poľa a jej zaevidovanie v poli `pouzite`
  - (rekurzívne) generovanie možností (riešení), ktoré vychádzajú z toho, čo máme ...
  - po návrate z rekurzie vrátime pôvodný stav pred spravením jedného kroku „bližšie“
    - odevidovanie hodnoty v poli `pouzite`

Backtracking a efektívnosť

- Zvyčajne veľmi neefektívny prístup s exponenciálnou časovou zložitosťou...
- **Typy problémov:**
  - backtracking je **jediná možná stratégia** na ich vyriešenie (resp. iné stratégie majú rovnakú časovú zložitosť)
  - backtracking je jediná možná stratégia na ich vyriešenie, no pre isté typy vstupov ich ide riešiť **„prakticky efektívnymi“ algoritmami** (napr. pseudopolynomiálne algoritmy)
  - riešenie vieme nájsť **bez skúmania všetkých možností** (napr. polynomiálne algoritmy)

ak nie sú otázky...

Ďakujem za pozornosť!