

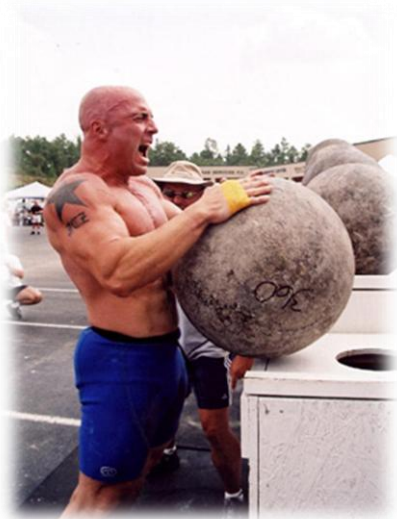


# 6. prednáška (20.3.2012)

## 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

```
for (int i = 1; i <= 3; i++)  
    for (int j = 1; j <= 3; j++)  
        for (int k = 1; k <= 3; k++)  
            System.out.println("[ "+i+" "+j+" "+k+" ]");
```

## ● Ďalšie výzvy:

- Ako zmeniť množinu z  $\{1, 2, 3\}$  na  $\{1, 2, \dots, 5\}$ ?
- 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 t'azký problém ...**

**... existuje jednoduchší podproblém?**



# Analýza trojíc...

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

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

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

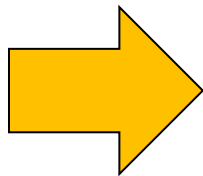
Každá skupina „trojíc“  
 sa skladá zo všetkých  
 dvojíc z čísel {1, 2, 3}



# Ako generovať trojice?

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

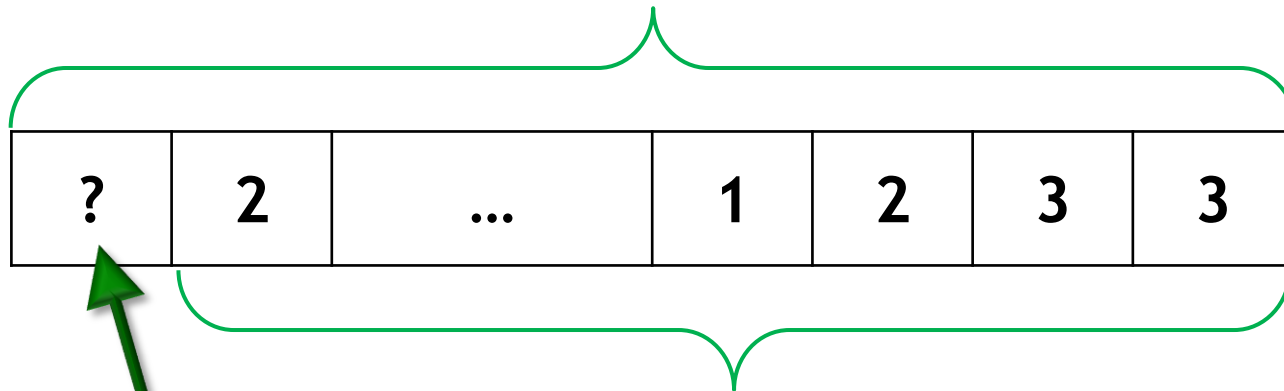
	1 + [...]	2 + [...]	3 + [...]
[1, 1]	[1, 1, 1]	[2, 1, 1]	[3, 1, 1]
[1, 2]	[1, 1, 2]	[2, 1, 2]	[3, 1, 2]
[1, 3]	[1, 1, 3]	[2, 1, 3]	[3, 1, 3]
[2, 1]	[1, 2, 1]	[2, 2, 1]	[3, 2, 1]
[2, 2]	[1, 2, 2]	[2, 2, 2]	[3, 2, 2]
[2, 3]	[1, 2, 3]	[2, 2, 3]	[3, 2, 3]
[3, 1]	[1, 3, 1]	[2, 3, 1]	[3, 3, 1]
[3, 2]	[1, 3,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





# Ako generovať $k$ -tice?

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



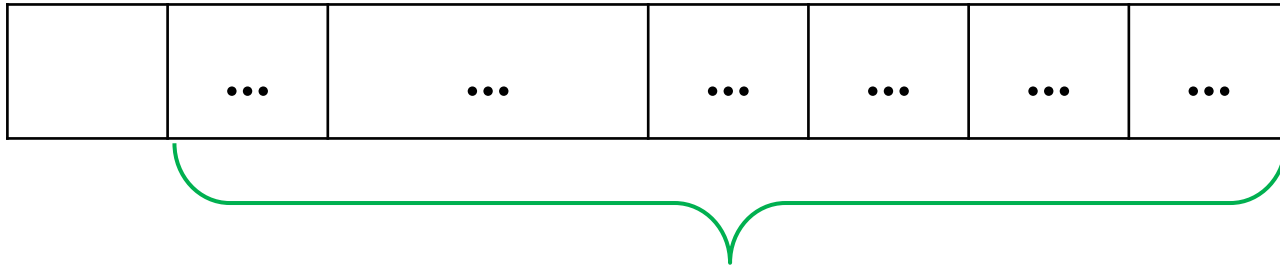
$(k-1)$ -prvková postupnosť z čísel  $\{1, 2, 3\}$

Postupným dosadením čísel 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  $(k-1)$ -prvkové  
postupnosti z čísel  $\{1, 2, 3\}$ ...

$\{1, 2, 3\}$



# Generovanie k-tíc

`private int[] pole;` generuj všetky postupnosti v podpoli určenom indexami: `odIndexu`, ..., `pole.length-1`

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

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

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

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.

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

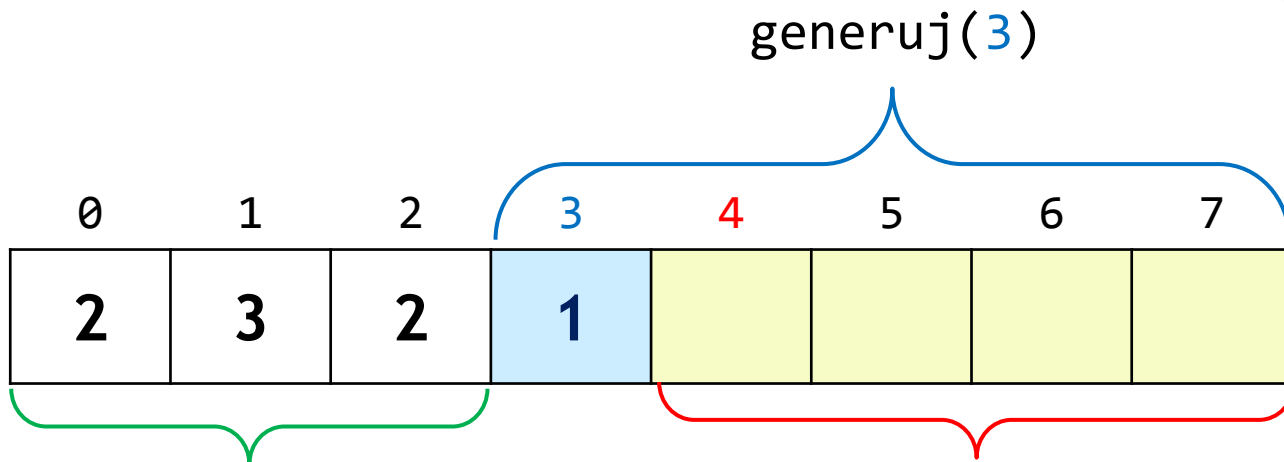
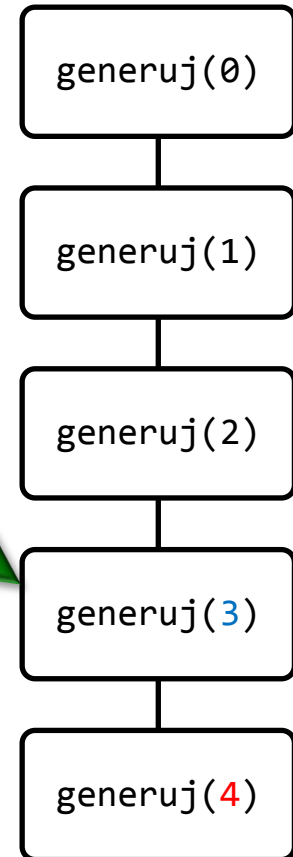
Začínáme generovanie na indexe 0, t.j. v celom poli.



# Ako to vlastne generuje?

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

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



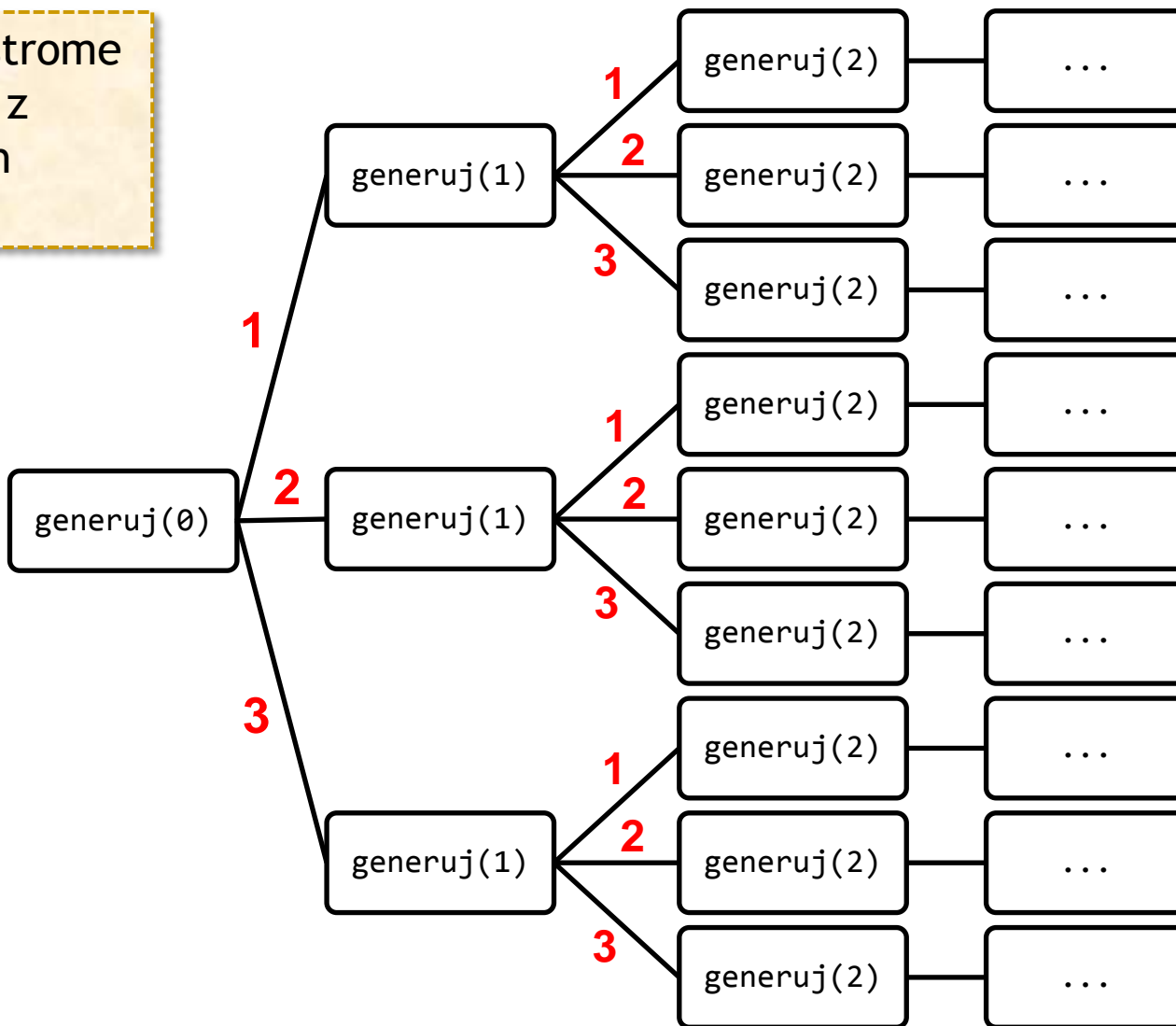
obsah pola určený metódami,  
ktoré viedli k volaniu  
generuj(3)

generuj(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?**



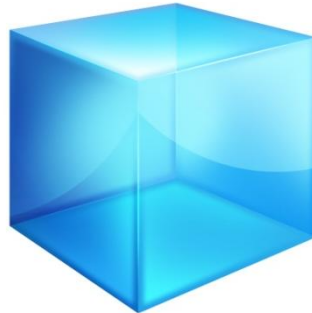
# Ako naplniť batoh?

Kapacita  
batoha: 4



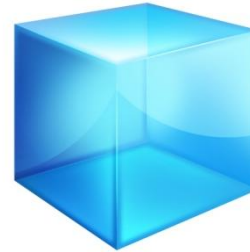
Veľkosť:

12 €



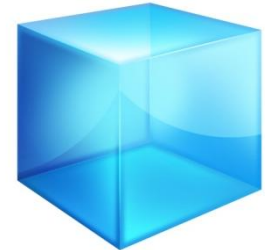
3

7 €



2

7 €



2

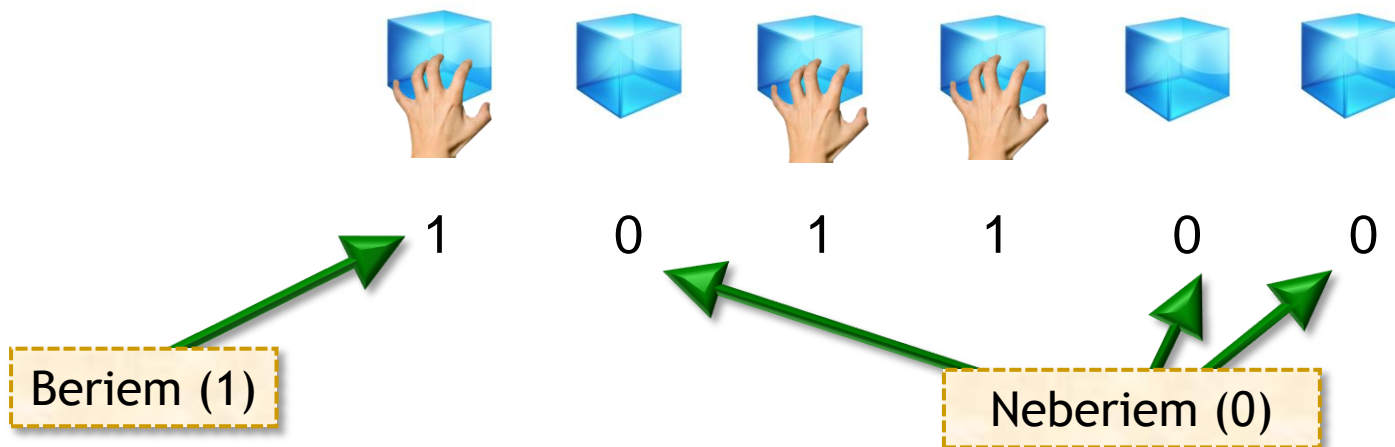
- Jednoduché stratégie nefungujú:

- 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?

Aká je **cena** výberu?



# Riešenie hrubou silou!

- Máme  $n$  predmetov:
  - vygenerujem všetky postupnosti dĺžky  $n$  z  $0$  a  $1$   
=  $n$ -prvkové **variácie s opakovaním** z prvkov množiny  $\{0 = \text{neberiem}, 1 = \text{beriem}\}$
  - 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:  $O(2^n)$

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



- 60 predmetov =  $2^{60}$  výberov  $> 10^{17}$  výberov
  - 1 GHz procesor ~  $10^9$  operácií 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 pod'me generovat' opät' nejaké číselné postupnosti...





# Variácie bez opakovania

- **Úloha:** Vygenerovať všetky postupnosti dĺžky  $k$  z prvkov množiny  $\{1, \dots, n\}$ , 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 < pole.length; i++)
        for (int j = i + 1; j < pole.length; j++)
            if (pole[i] == pole[j])
                return false;

    return true;
}
```

Overí, či sú všetky čísla v poli rôzne.

Pred výpisom overíme, že niet opakovaní.

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

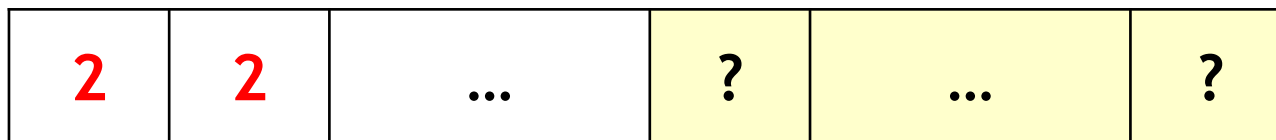
    for (int i = 1; i <= n; i++) {
        pole[odIndexu] = i;
        generuj(odIndexu + 1);
    }
}
```



# Ako znížiť straty?

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

generuj(...)

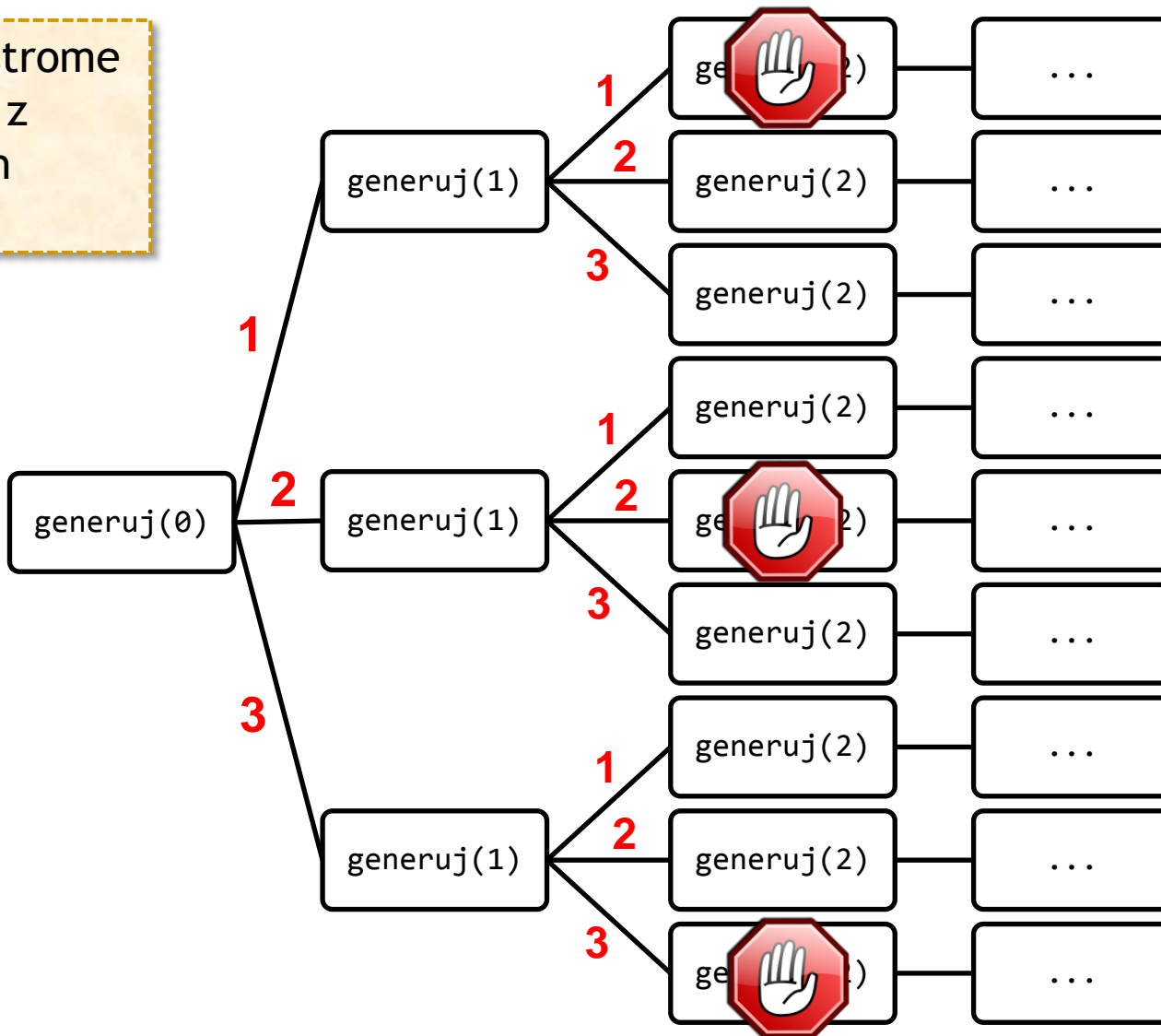


Načo generovať zvyšok, keď kvôli dvojkám na začiatku sa to aj tak nevypíše?



# 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 (pole[i] == hodnota)
            return false;

    return true;
}
```

Overí, či sa zadaná hodnota nenachádza v poli na indexoch 0,...,idx-1

```
private void generuj(int odIndexu) {
    ...
    for (int i = 1; i <= n; i++)
        if (moze(odIndexu, i)) {
            pole[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 zložitost'  $O(k)$
- **Zrýchlenie:** pamätajte si v poli, či sa číslo už nachádza v „ľavej“ fixovanej časti...

```

for (int i = 1; i <= n; i++) {
    if (!uzBolo[i]) {
        uzBolo[i] = true;
        pole[odIndexu] = i;
        generuj(odIndexu + 1);
        uzBolo[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  $\{1, \dots, n\}$ ?
  - prvky množiny dáme do nejakého poľa - napr. prvky
  - generujeme variácie indexov poľa prvky
  - výpis:

```
for (int i=0; i<pole.length; i++)
    variacia[i] = prvky[pole[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 uzBo1o
  - (rekurzívne) generovanie možností (riešení), ktoré vychádzajú z toho, čo máme ...
  - po návrate z rekurzie **vrátíme pôvodný stav** pred spravením jedného kroku „bližšie“
    - odevidovanie hodnoty v poli uzBo1o



# 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!**

