STU FIIT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES

Dátové štruktúry a algoritmy

Aula magna
utorok 11:00

zimný semester
2017/2018

Čo je to algoritmus?

- Na počiatku bolo slovo ...
- Ľudia mali hovorený jazyk, ale prečo vzniklo písmo?
- V starovekej Mezopotámii s miliónom obyvateľov potrebovali registrovať majetok, daňové povinnosti a ich spĺňanie / nespĺňanie
- Informácie zapamätať, vyhľadávať a aktualizovať

- Základným prvkom písma je **abeceda**
  - každá konečná neprázdna množina symbolov

Čísla ako postupnosti symbolov

- V Mezopotámii to bola kombinácia čísel 10 a 6.
- Základné jednotky: 1, 10, 60, 600, 3600, …
- Prečo tento systém neprežil?
- Prečo používame desiatkovú sústavu?
- Čo očakávame od zápisu čísla?
  - Rýchlo z neho rozumieme, čo tým zapísaným číslom myslíme
  - Efektívne aritmetické operácie
- Efektívne sčitovanie:
  - spojiť karty (symboly) dvoch čísel a potom minimalizácia „kariet“
  - použiť minimálny počet symbolov … Rímske čísla to mali pôvodne tiež, až neskôr sa začali dávať menšie pred väčšie (napr. IX=9).
- Násobenie ťažko realizovateľné…

Po písme prišla matematika

- Písmo: vedeli sme vytvárať a zapisovať poznatky
- Hľadali sme objektivitu: jazyk, v ktorom, všetko čo napíšeme, sa dá jednoznačne interpretovať…
- A tiež chceme vytvoriť jazyk, v ktorom je každá argumentácia verifikovateľná …
- **Úlohou matematiky je automatizovať**

  Vymyslieť Pytagorovu vetu bolo intelektualne náročné, ale keď stavební robotníci potrebovali vyrábať 90° uhly, tak napínali trojuholníky so stranami 3,4 a 5 a vedeli, že majú pravú uhol …

Úlohou matematiky je automatizovať

- Problém je množina inštancii (prípadov) problému … nekonečná množina.
- Napr. Problém sústavy lineárnych rovníc:
existuje nekonečne veľa sústav lineárnych rovníc
- Chcem algoritmus, ktorý konečným počtom krokov opíše algoritmus ako to vyriešiť …
a týmto redukujeme nekonečno na konečno.
- Algoritmus je jednoznačným popisom problému … mám konečný opis problému.
- Častokrát však mám problém (zadanie úlohy) a nemám (zatiaľ neviem) algoritmus ako ju riešiť...

David Hilbert a Kurt Gödel

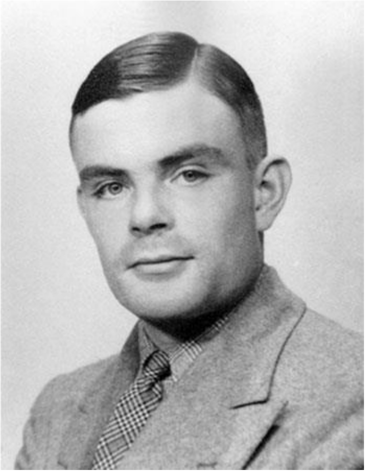
- David Hilbert mal sen, že pre každý matematicky formulovateľný problém existuje algoritmus (konečný popis tohto problému) a ak ho nemáme je to naša hlúposť ... (a musíme len hľadať)
- Hilbert veril, že práca matematika (dokazovať) sa dá automatizovať

- Gödel, 1930 dokázal, že existujú matematické tvrdenia, ktoré sa nedajú dokázať ani vyvrátiť.
- Jazyk matematiky je silnejší na popisovanie ako na argumentovanie (dokážem v ňom viac zapísať ako dokázať)

Vzniká teoretická informatika (computer science)

- Dokážem popísať algoritmy jednoznačne, a na vykonanie nepotrebujem človeka (jeho improvizáciu a vedomosti)
- Vznikla technológia, ktorá umožňuje mechanicky vykonávať postupnosti krokov: matematický výpočet môžem delegovať na stroje …
- Predtým než boli prvé počítače … chceme vedieť rozhodnúť, ktoré problémy sa dajú riešiť, a ktoré sa nedajú riešiť
- Aby sme to mohli matematicky dokázať, tak musíme vedieť matematicky zapísať algoritmus …

Alan Turing



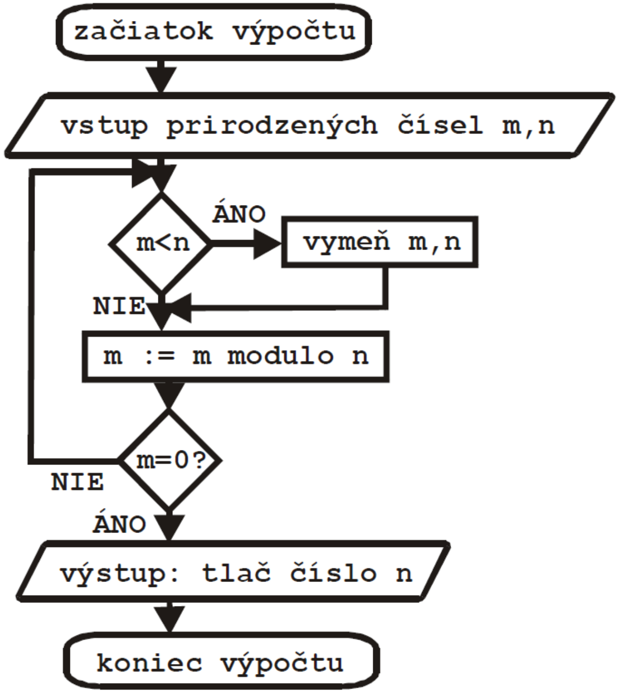
- Turing vymyslel **Stroj**, a všetko čo sa na ňom dá vykonať je algoritmus, ostatné nie…
- Prečo mu ľudia verili, že TOTO je ten stroj, ktorý definuje všetky algoritmy?
- Turing si uvedomil, že všetko (všetky informácie) sú postupnosti symbolov; matematik upravuje postupnosti symbolov …
- Čo to znamená „vykonať operáciu“?
  - Vyberiem si postupnost symbolov a vykonám nad ňou operáciu …
  - dostanem inú postupnosť symbolov
- Dĺžka postupnosti nemôže byť obmedzená (dĺžka pásky je neobmedzená).

Alan Turing

- Turing si uvedomil, že náš mozog je konečne veľký a teda z každého „nekonečna" vidíme len konečne veľkú časť… čiže zmena, ktorá môže vo výpočtovom stroji nastať, je najviac konštantne veľká (ako aj kapacita mozgu)
- Ktoré operácie teda umožním vykonávať, aby to boli všetky možné? **stačí operácie, ktoré môžu zmeniť najviac jeden symbol**. <u>To sú všetky operácie, iné neexistujú</u> (zložitejšie viem rozdeliť na jednotlivé kroky úprav po jednom symbole)
- Týmto sme definovali **Turingov stroj**
  - Je to to isté, ako keď dáme matematikovi ceruzku a gumu, tak potom mu sta písať a gumovať na 1D páske
- Aj iné výpočtové formalizmy sa ukázali, že sú ekvivalentné s **Turingovym strojom**
- **Kvantový počítač (fyzikálna realita) je tiež ekvivalentný s Turingovým strojom**
  - **vypočítam na ňom to isté (aj keď možno pomalšie)**

Algoritmus (opakovanie)

- Postup na vyriešenie úloh určitého typu
  vstup → výstup
- Jednoznačnosť krokov
- Všeobecnosť použitia
- Vlastnosti
  - Konečnosť
  - Efektívnosť



Inými slovami

- **Algoritmus** v informatike je **jednoznačná**, **presná** a **konečná** postupnosť operácií, ktoré sú aplikovateľné na množinu objektov alebo symbolov (čísiel, šachových figúrok, ingrediencií na bábovku).
- Počiatočný stav týchto objektov je **vstup**om, ich koncový stav je **výstup**om.
- Počet operácií, vstupy a výstupy sú **konečné** (aj keď bežne počítame napr. s iracionálnym číslom π, vždy jeho číselnú reprezentáciu obmedzíme pri numerických výpočtoch na konečnú presnosť, napr. π=3.14).
- Jeden problém môže byť riešení viacerými algoritmami

Vlastnosti algoritmov

- **Jednoznačnosť** znamená, že každý krok algoritmu musí byť presne definovaný. Nesmie dovoľovať viac výkladov, jednoznačne je určený krok za ním nasledujúci.
- **Univerzálnosť** algoritmu znamená, že je použiteľný pre riešenie veľkej skupiny úloh toho istého typu, líšiacich sa vstupnými údajmi (tak napr. algoritmus pre hľadanie koreňov kvadratickej rovnice je použiteľný pre každú kvadratickú rovnicu).

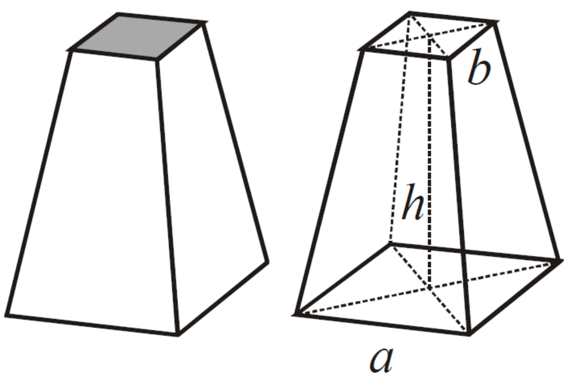
Vlastnosti algoritmov (2)

- **Rezultatívnosť** znamená, že algoritmus vždy musí po konečnom počte krokov dojsť k nejakému riešeniu.
- **Správnosť (korektnosť)** algoritmu znamená, že odpoveď pre každý vstup je správna a korektná – teda vyhovuje špecifikácii problému, ktorý algoritmus o sebe tvrdí, že rieši.
- **Efektívnosť** algoritmus zahŕňa zdroje potrebné na vykonanie výpočtu – výpočtový čas, kapacita pamäte, počet správ pri komunikácií, a pod.

Prvé algoritmy v Starom Egypte

- Návod na výpočet objemu zrezaného ihlanu (ihlan so štvorcovou podstavou, ktorého vrchol je zrezaný rovinou rovnobežnou s podstavou) podľa tzv. Moskovského papyrusu (Egypt, 1850 p.n.l.):
  - Je daná orezaná pyramída, ktorej výška je 6, strana podstavy je 4 a strana hornej základne je 2. Vypočítaj objem tejto pyramídy:

    1. Umocni číslo 4 na druhú, dostaneš 16.
    2. Číslo 4 zdvojnásob, dostaneš 8.
    3. Umocni na druhou číslo 2, dostaneš 4.
    4. Tieto čísla 16, 8 a 4 sčítaj, dostaneš 28.
    5. Urči tretinu z čísla 6, dostaneš 2.
    6. Zdvojnásob číslo 28, dostaneš 56.
    7. Celkový výsledok je 56, počítal si dobre.



- Použitie symbolov až u starých Grékov, V=1/3 h (a^2 + ab + b^2)
- **Formalizácia konceptu algoritmu v Euklidových *Základoch*** (3. st. p. n. l.)

Najväčší spoločný deliteľ prirodzených čísel m a n

- Euklidov algoritmus:
  1. Keď *m* je menšie ako *n*, vymeň ich hodnoty
  2. Do *m* daj hodnotu zvyšku po delení *m*/*n* (v modernej terminológii sa to zapisuje ako *m* := *m* modulo *n*)
  3. Keď sa *m* nerovná nule, choď na krok 1 s novými hodnotami *m* a *n*
  4. Vráť *n* ako výsledok

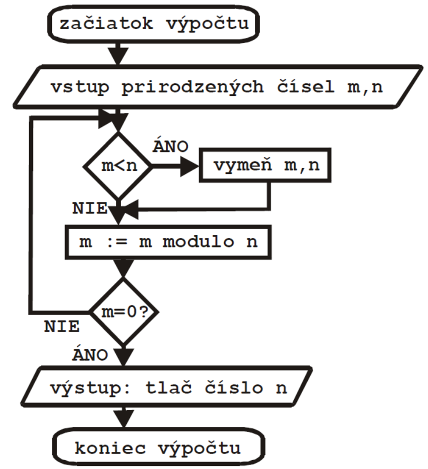
Eratosthenovo sito (3. st. p.n.l.)

- Nájdenie všetkých prvočísiel menších ako *n* (prvočíslo je celé číslo väčšie ako 1, ktoré je bezo zvyšku deliteľné iba sebou samým a číslom 1, v súčasnej dobe sa prvočísla často využívajú napr. pri šifrovaní správ). Eratosthenov predpis znie:
  1. Zapíš do zoznamu všetky čísla od 2 do *n*
  2. Nech *k*=2
  3. Pre každé číslo *m* medzi *k*+1 a *n* skontroluj, či je presným násobkom *k*, keď áno, vyškrtni *m* zo zoznamu.
  4. Do *k* daj najmenšie ešte nevyškrtnuté číslo zo zoznamu
  5. Keď *k* je menšie ako *n*, opakuj od kroku 3
  6. Akékoľvek číslo zo zoznamu, ktoré nebolo vyškrtnuté, je prvočíslo

	~~1~~	2	3	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~
deliteľné	def.			2		2		2	3	2
	11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
deliteľné		2		2	3	2		2		2
	~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
deliteľné		2		2		2		2		2
	31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
deliteľné		2	3	2	5	2		2	3	2
	41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
deliteľné		2		2	3	2		2	7	2

Zápis algoritmus

- Zrozumiteľnosť a stručnosť zápisu
- Slovný opis
- Vývojový diagram
- Pseudokód




```
Vstup dvoch celých čísel: m, n
  rob
    keď m < n vymeň m,n
    m := m modulo n
  dokial n≠0
  vytlač n
```

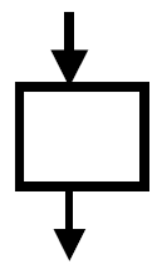
- Kód v programovacom jazyku
  - Vyšší programovací jazyk
  - Medzikód
  - Strojový jazyk
- Logický obvod

```
int nsd(int m, int n)
{
  if (n == 0)
    return m;
  return nsd(n, m%n);
}
```

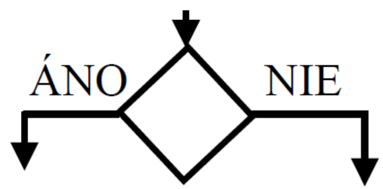
Jazyk vývojových diagramov

- Popisuje tok riadenia od jedného k ďalšiemu kroku algoritmu (bežne zhora dole) pomocou orientovaných hrán (šípok) spájajúcich činnosti opísané v blokoch.

- Začiatok, resp. koniec 

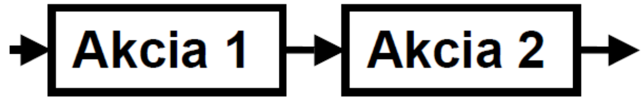
- Operačný blok, obsahuje akcie 

- Rozhodovací blok, splnenie/nesplnenie podmienky určí nasledujúci krok riadenia

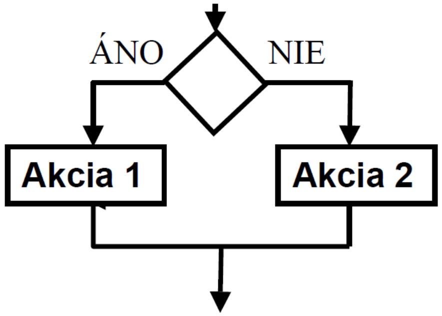


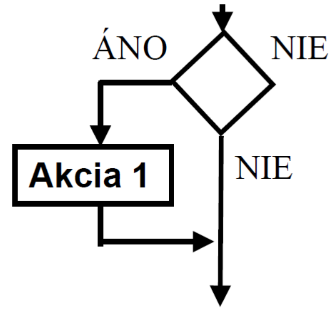
Jazyk vývojových diagramov (2)

- Sekvencia

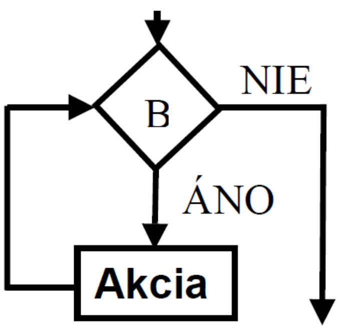


- Vetvenie

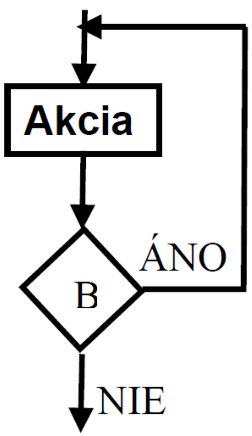




- Cyklus s testom na začiatku



Cyklus s testom na konci

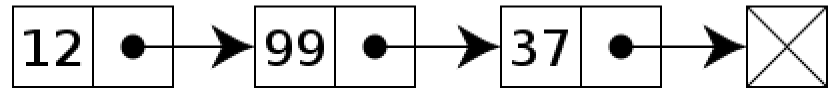


Dátová štruktúra (opakovanie)

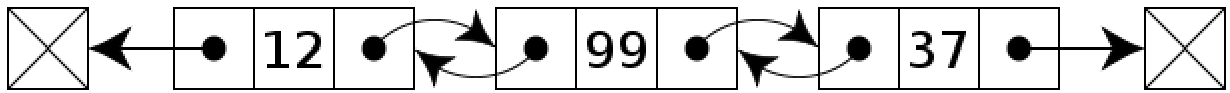
- Spôsob organizácie a uchovania dát, ktorý umožňuje efektívne využitie / spracovanie dát
- Operácie – **čo** umožňuje s dátami robiť
- Konzistentnosť
- Vlastnosti
  - Súbežnosť (concurrency)
  - Perzistentnosť (persistency)
  - Dynamickosť (online/offline)
  - Efektívnosť

Spájaný zoznam (linked list)

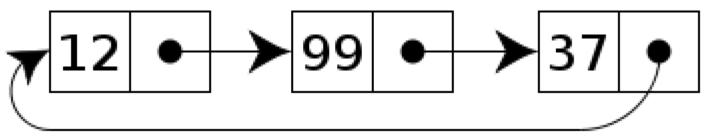
- Jednosmerne zreťazený



- Obojsmerne zreťazený



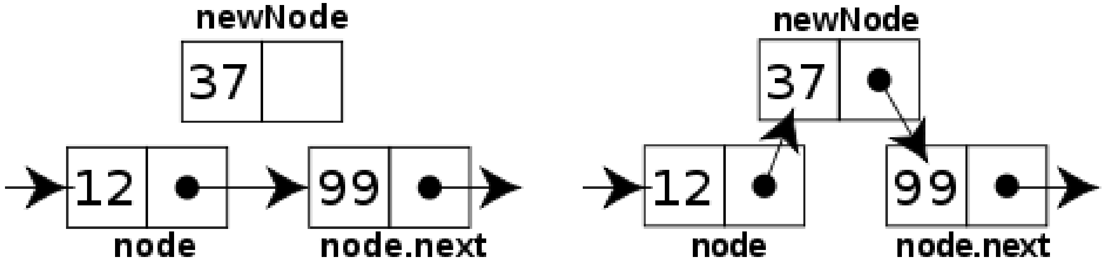
- Cyklický



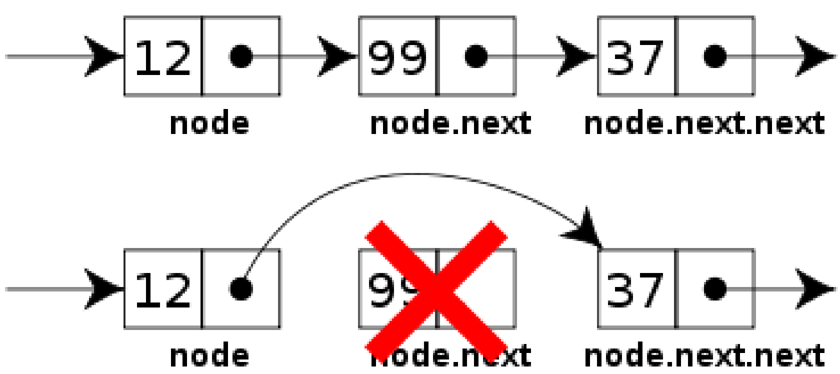
- Pomocné prvky (tzv. sentinel / dummy nodes)
- Ako vyzerá prázdny zoznam?
- Ako zistíme, či je zoznam cyklický?

Spájaný zoznam – operácie

- Vloženie (operácia insert)



- Odstránenie (operácie remove)



Reťazec / pole (vektor)

40	30		10	20	-5	30		
0	1	2	3	4	5	6	7	8

- Operácie:
  - Nastaviť i-ty prvok – **Set(i, value)**
  - Zistiť hodnotu i-teho prvku – **value** ← **Get(i)** resp. **getCharAt(i)**
  - Nájsť pozíciu prvku – **index** ← **find(value)**
  - Určenie dĺžky – **uint** ← **getLength()**
  - Je prázdny? – **bool** ← **isEmpty()**

Ako sa pracuje na tomto predmete?

- **Prednášky (utorok 11:00):**
  zapájať sa, pýtať sa, počúvať, ...

- **Cvičenia:**
  písanie programov (riešenie úloh)
  malé úlohy a veľké zadania
  konzultácie k úlohám, odovzdanie zadaní
  www.turing.sk/fiit/dsa

- **Doma:**
  vypracovanie veľkých zadaní
  dokončenie malých úloh

TURING

STU FIIT

Fakulta informatiky a informačných technológií, Slovenskej technickej univerzity v Bratislave

STU FIIT Datové štruktúry a algoritmy

Zimný semester 2017/2018

Získať hlbšie znalosti o metódach navrhovania efektívnych algoritmov a dátových štruktúr a osvojiť si príslušné zručnosti. Pochopiť princípy špecifikovania údajových typov. Zvládnuť postupy, metódy, štruktúry údajov pre usporadúvanie a vyhľadávanie. Získať praktické skúsenosti v oblasti implementovania algoritmov a údajových typov.

Prednášky: **Mgr. Jozef Tvarožek, PhD.**

Cvičenia: **Ing. Róbert Cuprik**, **Ing. Michal Farkaš**, **Ing. Tomáš Farkaš**, **Ing. Ivan Kapustík**, **Ing. Samuel Pecár**, **Ing. Jakub Ševcech, PhD.**, **Ing. Petra Vrablecová**

Použite prihlasovacie údaje z AIS.

AIS login

Heslo

PRIHLÁSIŤ

Vyučujúci

- Prednášky (utorok 11:00):
  Jozef Tvarožek

- Cvičiaci:
  Róbert Cuprík
  Michal Farkaš
  Tomáš Farkaš
  Ivan Kapustík
  Samuel Pecár
  Jakub Ševcech
  Petra Vrablecová

Podmienky absolvovania

- Môžete získať až 100 bodov
- priebežne riešené úlohy (zadania)
  (max. 50 bodov: na cvičeniach 20 a doma 30):
  - na cvičení sa budú riešiť **malé úlohy**; (môže ich byť viac, každé najviac za 2 body), do konečného hodnotenia sa započítava najviac 20 bodov za všetky malé úlohy;
  - doma sa budú riešiť 3 **zadania**
    každé max. 10 bodov, min. 4.
- priebežný test (max. 15 bodov, treba získať min. 10)
- Podmienky udelenia zápočtu:
  - minimálne 25 bodov z priebežne úloh (vrátane zadaní)
  - minimálne 5 bodov z priebežného testu
- záverečná skúška (max. 35 bodov, treba získať min. 15)

Plagiátorstvo

- Všetko, čo sa predkladá na hodnotenie, **musí byť vlastná samostatná práca študenta** alebo musí byť označené ako prevzaté.
- Samozrejme, **body možno získať len za vlastnú prácu.** Opisovanie sa netoleruje.
- Pokiaľ sa študent pokúša absolvovať tento predmet nie vlastnou prácou, kvalifikuje sa na FX.

Čo chcem, aby ste si odniesli z tohto predmetu

- Prehľad nástrojov (algoritmov a dátových štruktúr) pre riešenie rozličných problémov
- Porozumieť vlastnostiam týchto nástrojov, najmä:
  - časovej efektívnosť (výpočtovej zložitosti)
  - pamäťovej efektívnosti (priestorovej zložitosti)
- Schopnosť aplikovať a prispôsobiť-upraviť tieto nástroje pre špecifické problémy
(za účelom dosiahnutia čo najlepšej efektívnosti)
- Otvorenosť pre ďalšie štúdium nových algoritmov a použitie efektívnych algoritmov a dátových štruktúr vo vašej ďalšej práci

Jeden problém, viac riešení

- Problém je zaujímavý vtedy, keď je zadaný všeobecne, aby pokrýval rozmanitosť problémov reálneho sveta
- Všeobecne zadaný problém má zvyčajne viacero možných riešení
- Chceme vedieť rozlíšiť rôzne tieto riešenia, a zvoliť to, ktoré vyhovuje požiadavkám našej situácie

Pre dané N určiť súčet čísel 1, 2, ..., N

- Prvé možné riešenie:
  čísla postupne pripočítavame k výsledku

```
int i, sucet = 0;
for (i = 1; i <= N; i++)
  sucet += i;
```

- Vykonáme rádovo N operácií
  - Pre väčšie vstupy (zvyšujúce sa N) sa úmerne **zvyšuje** aj výpočtová zložitosť

Pre dané N určiť súčet čísel 1, 2, ..., N

- Druhé možné riešenie:
  výpočet použitím vzorca v uzavretom tvare

```
int sucet = N*(N+1)/2;
```

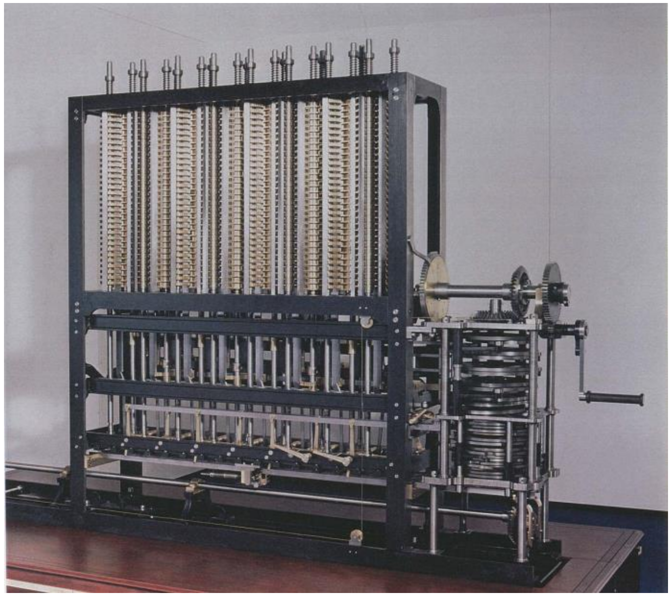
- Vykonáme vždy konštantný počet operácií
  - Pre väčšie vstupy (zvyšujúce sa N) sa **NEZVYŠUJE** výpočtová zložitosť

Prečo má zmysel analyzovať algoritmy?

- Klasifikovať problémy a algoritmy podľa náročnosti
- Predvídať výkon, porovnávať algoritmy a dátové štruktúry, vylaďovať parametre
- Lepšie porozumieť a vylepšovať implementácie algoritmov a dátových štruktúr
- **Intelektuálna výzva**

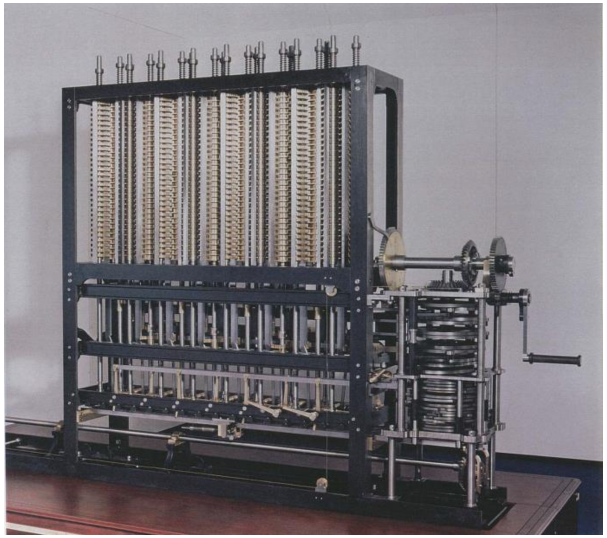
Charles Babbage (1791-1871)

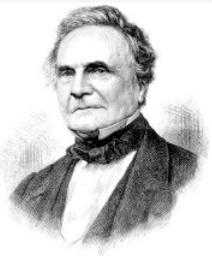
- anglický matematik navrhol v roku 1830 stroj **Differential engine** poháňaný parou na výpočet a tlač matematických tabuliek, využívajúci metódu diferencií. Táto metóda prevádzala výpočet polynómov na sčítanie.



Charles Babbage (1791-1871)

- Návrh univerzálneho stroja **Analytical Engine**
  - Výpočet plánoval riadiť pomocou diernych štítkov, kedy jeden druh štítkov obsahoval určenie operácie a druhý adresu čísla; stroj mal mať mechanickú adresovateľnú pamäť (až 1000 čísel po 50 cifrách) a "mlynček" = centrálnu aritmetickú jednotku, v ktorej sa mali vykonávať základné operácie.

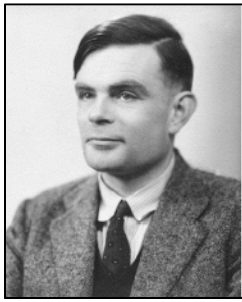




"As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time?"

— Charles Babbage (1864)

Alan Turing (1947) – Donald E. Knuth (1960)



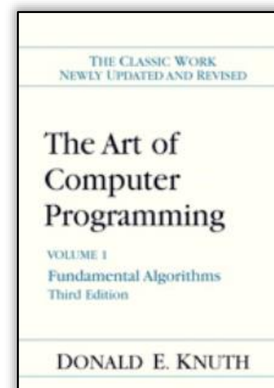
"It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process . . ."

— Alan Turing (1947)

- Donald E. Knuth:

To analyze an algorithm:

- Develop a good implementation.
- Identify unknown quantities representing the basic operations.
- Determine the cost of each basic operation.
- Develop a realistic model for the input.
- Analyze the frequency of execution of the unknown quantities.
- Calculate the total running time: $\sum_{q} \mathrm{frequency}(q) \times \mathrm{cost}(q)$



Súčasnosť

- Aho, Hopcroft, Ullman (1970)
- Cormen, Lieserson, Rivest, Stein (1990-)
- Analýza najhoršieho prípadu
- Použitie O-notácie pre asymptotický horný odhad
- Klasifikujeme algoritmy podľa týchto zložitostí
- Nevýhoda tohto prístupu: **Nemôžeme použiť na predvídanie výkonu alebo porovnanie algoritmov!**
  - Quicksort – počet porovnaní v najhoršom prípade $O(N^2)$
  - Mergesort – počet porovnaní v najhoršom prípade O(N log N)
  - V praxi je však Quicksort zvyčajne dva krát rýchlejší a používa polovičné množstvo pamäti...

Ako merať zložitosť algoritmov?

- Analýza zložitosti algoritmu je **výpočet-odhad** požiadaviek na výpočtové prostriedky, ktoré bude vykonanie algoritmu vyžadovať **v závislosti na veľkosti vstupu**
- Veľkosť vstupu
  - Počet bitov vstupného čísla
  - Dĺžka postupnosti čísel na vstupe
  - Rozmery vstupnej matice
  - Počet znakov textu na vstupe
  - ...

Výpočtové prostriedky

- Výpočtová (časová) zložitosť
  - Cykly procesora
  - Doba výpočtu
  - Počet vykonaných inštrukcií
  - Počet transakcií nad databázou
- Pamäťová (priestorová) zložitosť
  - Pamäťové bunky
- Komunikačná zložitosť
  - Sieťová kapacita
  - Počet spojení
- Algoritmus môže byť tzv. **citlivý na vstup** (na hodnoty vstupu, nielen množstvo vstupu)

Schopnosti potrebné pre analýzu zložitosti

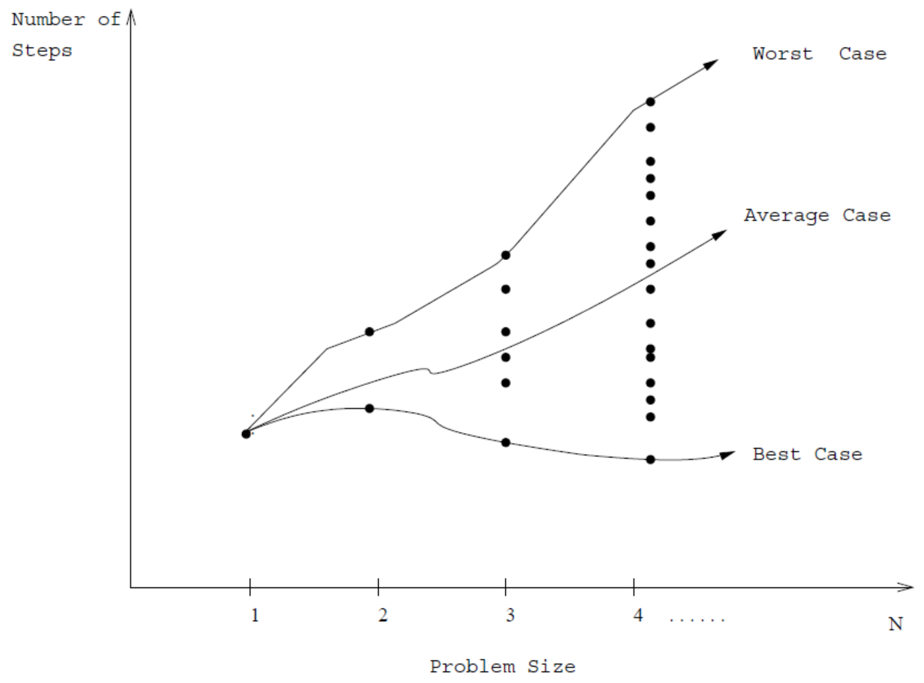
- Porozumenie algoritmu
- Odhad rádov rýchlosti rastu funkcií
- Schopnosť abstrahovať výpočtový model
- Matematické znalosti
  - diskrétna matematika, kombinatorika
  - matematickej analýzy – rady, rekurentné rovnice
  - základy pravdepodobnosti
- ...

Model výpočtového systému

- Jednoprocesorový Random Access Machine (RAM)
- Dáta sú uložené v adresovateľnej pamäti
- Vykonáva aritmeticko-logické, riadiace a pamäťové inštrukcie
- Časová zložitosť inštrukcií je jednotková (konštantná)
  - Rozšírenie: môžeme uvažovať aj váhu (cenu) inštrukcií
- Cykly a volanie funkcií nie sú jednoduché operácie a ich zložitosť závisí na veľkosti dát a obsahu funkcie
- Inštrukcie sú vykonávané postupne-sekvenčne
  - Neuvažujeme súbežné vykonávanie vlákien
    rozšírenie: Parallel Random Access Machine (PRAM)

Analýza prípadov

- **Najhorší prípad** (worst case)
  Zložitosť algoritmu v najhoršom prípade je funkcia definovaná maximálnym počtom krokov, ktoré algoritmus vykoná pri (ľubovoľnom) vstupe veľkosti N.



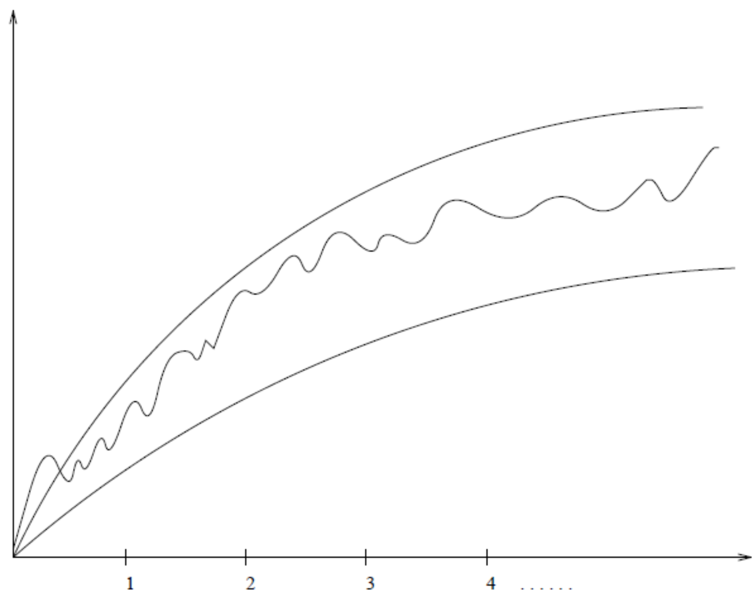
Analýza prípadov (2)

- **Najlepší prípad** (best case)
  Zložitosť algoritmu v najlepšom prípade je funkcia definovaná minimálnym počtom krokov, ktoré algoritmus vykoná pri (nejakom) vstupe veľkosti N.

- **Priemerný prípad** (average case)
  Zložitosť algoritmu v priemernom prípade je funkcia definovaná priemerným počtom krokov, ktoré algoritmus vykoná pri vstupoch veľkosti N.

Asymptotická zložitosť

- Presné určenie počtu vykonaných operácií:
  - veľmi náročné v prípade zložitých algoritmov
  - skoro zbytočné pre jednoduché algoritmy
- Jednoduchšie je analyzovať horné a dolné ohraničenia počtu vykonaných krokov

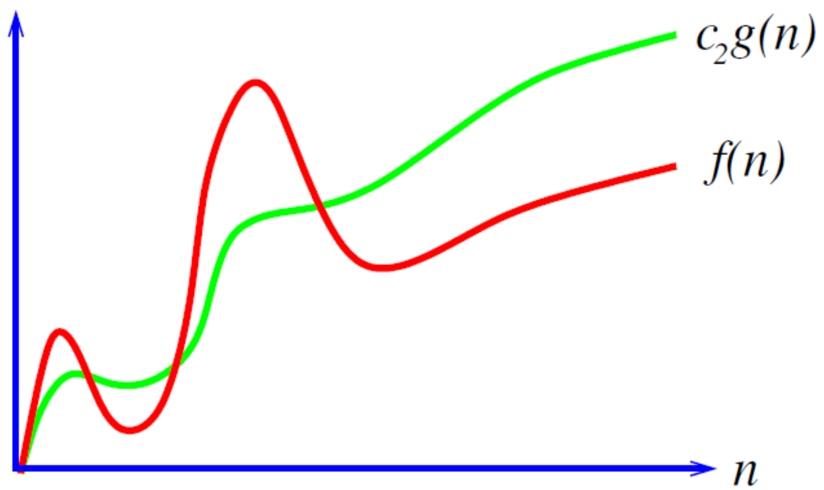


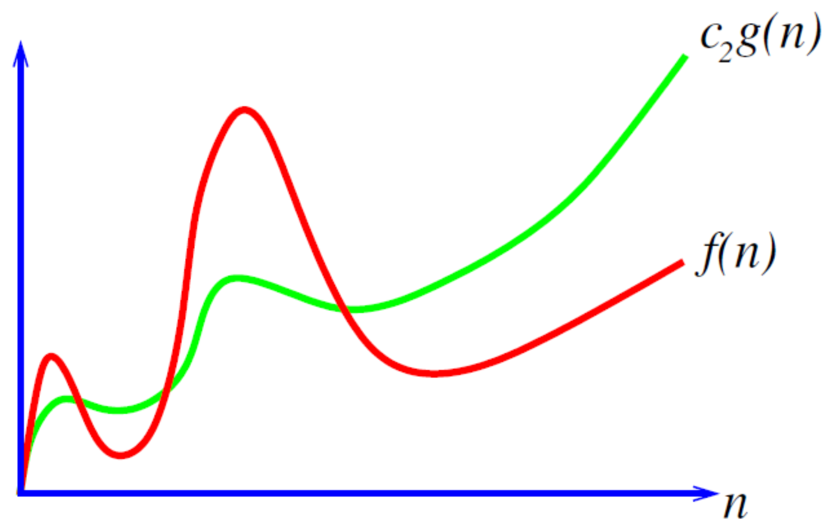
Asymptotická zložitosť (2)

- Prakticky nás zaujíma ako sa bude algoritmus správať pre veľké vstupy idúce do **nekonečna**
- Vyjadrujeme rád rastu funkcie, **zanedbávame príspevok nižších rádov**
- Asymptotická zložitosť je vyjadrenie pre takú (veľkú) veľkosť problému, aby sa prejavil rád rastu funkcie zložitosti v závislosti na veľkosti vstupu
- **Asymptoticky lepší algoritmus bude lepší pre všetky vstupy okrem konečného počtu malých vstupov**

Ohraničujúce funkcie

- **Horný odhad: $f(n) = O(g(n))$**
  znamená, že $c_2 \times g(n)$ je horné ohraničenie $f(n)$.

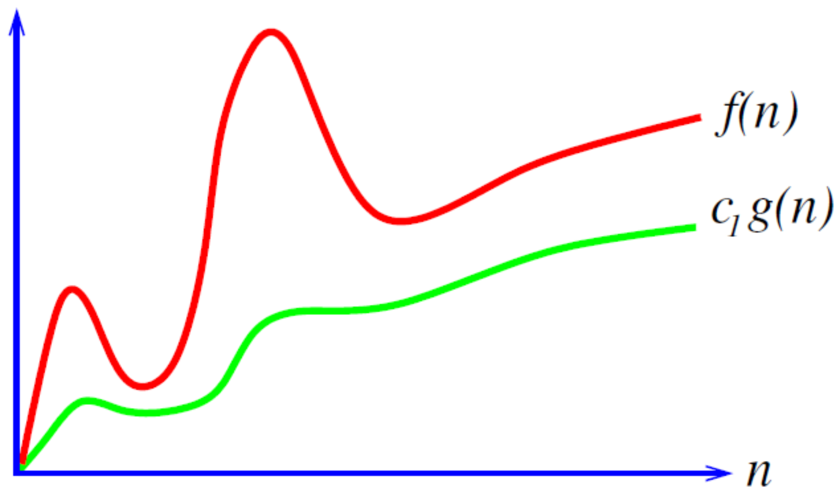


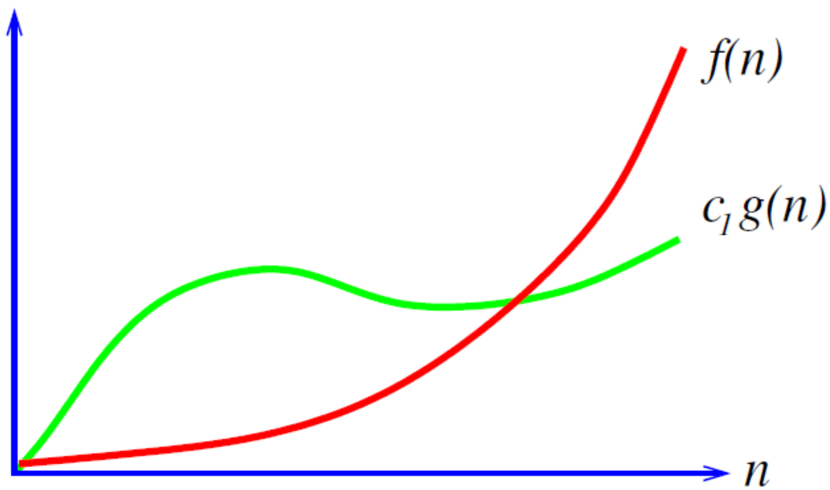


- $6n^2 - 4n = O(n^2)$, tiež $6n^2 - 4n = O(n^3)$
- $O(g(n))$ je množina, a píšeme aj $6n^2 - 4n \in O(n^2)$

Ohraničujúce funkcie

- **Dolný odhad: f(n) = Ω(g(n))**
  znamená, že c_1 x g(n) je dolné ohraničenie f(n).

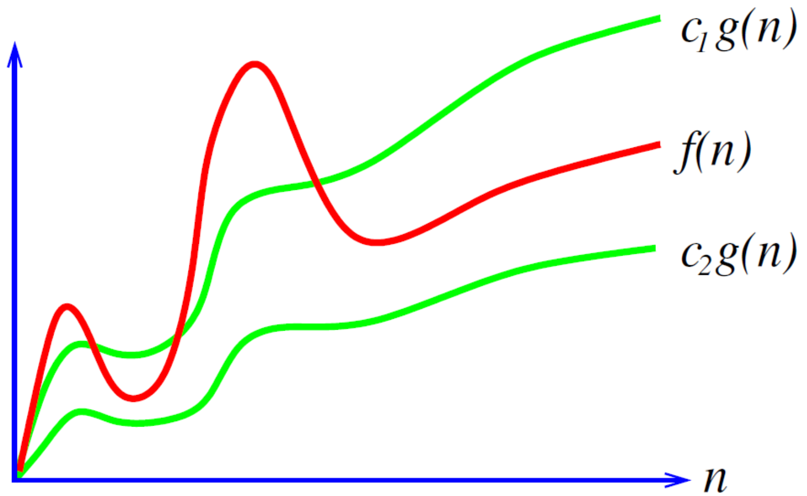




- 5n log n + 3 = Ω(n log n), ale tiež 5n log n + 3 = Ω(n)
- Ω(g(n)) je množina, píšeme 5n log n + 3 ∈ Ω(n)

Ohraničujúce funkcie

- **Tesný odhad: f(n) = Θ(g(n))**
  znamená, že c_1 x g(n) je horné ohraničenie f(n) a c_2 x g(n) je dolné ohraničenie f(n). c_1 a c_2 sú konštantne nezávislé od n.



- $5n^2 - 6n + 7 = \Theta(n^2)$

Asymptotická dominancia

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Asymptotická dominancia (2)

- n^a dominuje n^b akk $a > b$, lebo

$$\lim_{(\mathrm{n}\to\infty)} \frac{n^b}{n^a} = n^{b-a} \to 0$$

- Nejaké základné funkcie:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \sqrt{n} > \log n \gg 1$$

Triedenie priamym vkladaním (Insert sort)

- Insert sort spracúva vstupnú množinu postupne tak, že pojednom pridáva prvky na správne miesto do výslednej usporiadanej postupnosti (ktorá je najskôr prázdna a postupne sa rozširuje).

```
int* insert_sort(int *input, int n)
{
  int i, result[n];
  for (i = 0; i < n; i++)
    insert(input[i], result);
  return result;
}
```

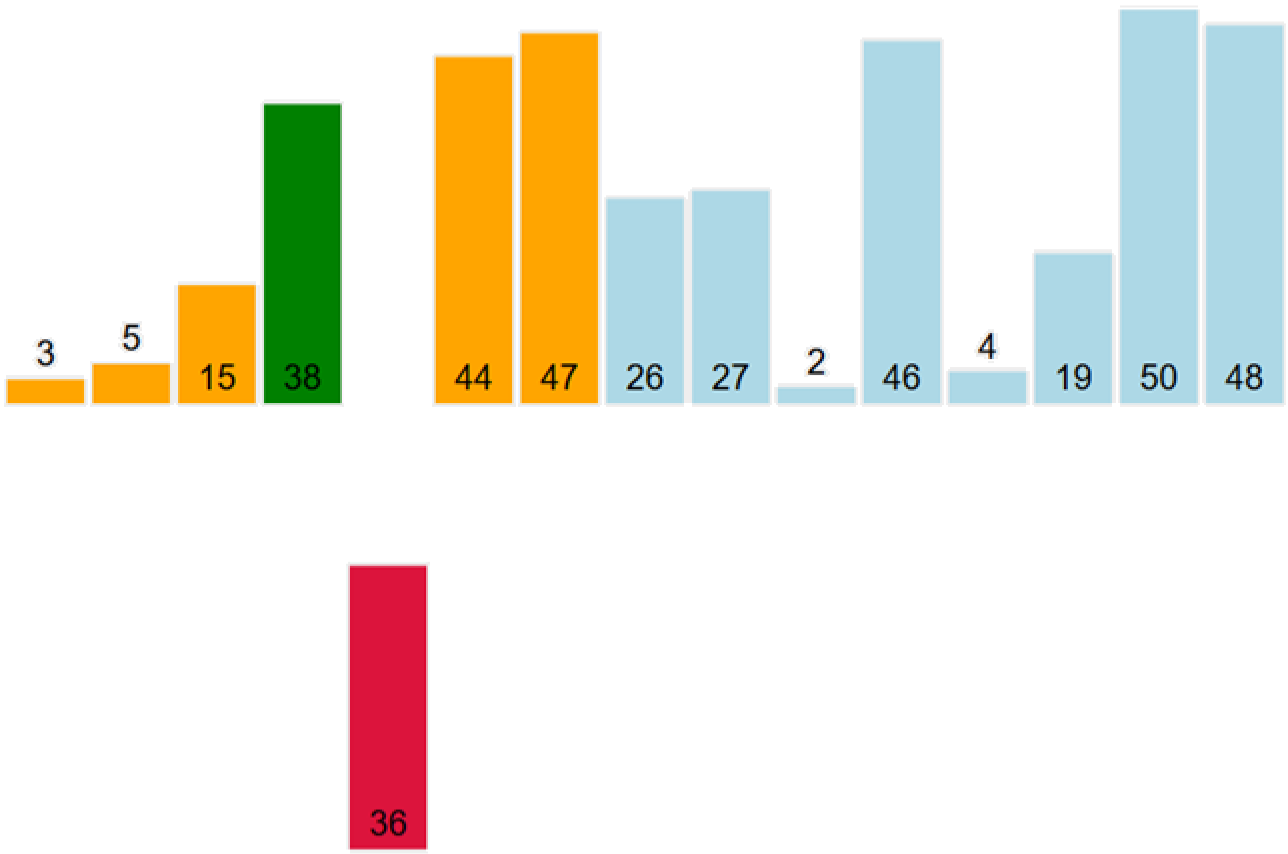
Triedenie priamym vkladaním (Insert sort)

```
int* insert_sort(int *input, int n)
{
  int i, result[n];
  for (i = 0; i < n; i++)
    insert(input[i], result);
  return result;
}
```

- Procedúra **insert**(prvok, pole) pomocou jednoduchého cyklu vloží prvok do poľa (v ktorom sú prvky v usporiadanom poradí) na správne miesto; vyžaduje rádovo ***L*** operácií, kde ***L*** je dĺžka poľa.

Triedenie priamym vkladaním (Insert sort)

- Animovaná ukážka: https://visualgo.net/bn/sorting



Triedenie zlučovaním (Merge sort)

- Merge sort vstupnú množinu rozdelí na dve polovice, každú rekurzívne utriedi, no a výslednú usporiadanú postupnosť všetkých prvkov určí zlúčením týchto menších usporiadaných postupností.

```
int* merge_sort(int *input, int left, int right)
{
  int mid = (left+right)/2;
  merge_sort(input,left,mid);
  merge_sort(input,mid+1,right);
  return merge(input,left,mid,right);
}
```

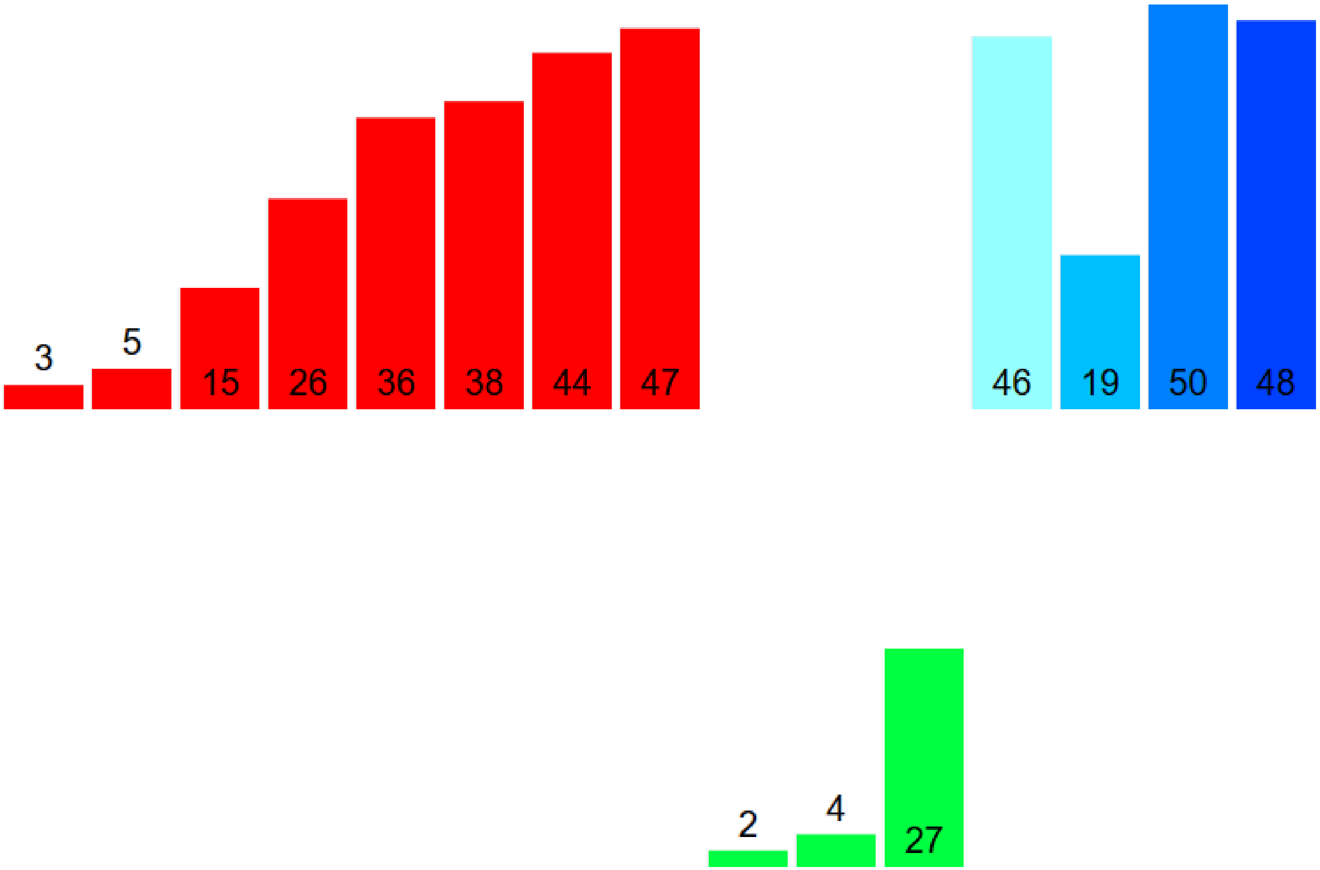
Triedenie zlučovaním (Merge sort)

```
int* merge_sort(int *input, int left, int right)
{
  int mid = (left+right)/2;
  merge_sort(input,left,mid);
  merge_sort(input,mid+1,right);
  return merge(input,left,mid,right);
}
```

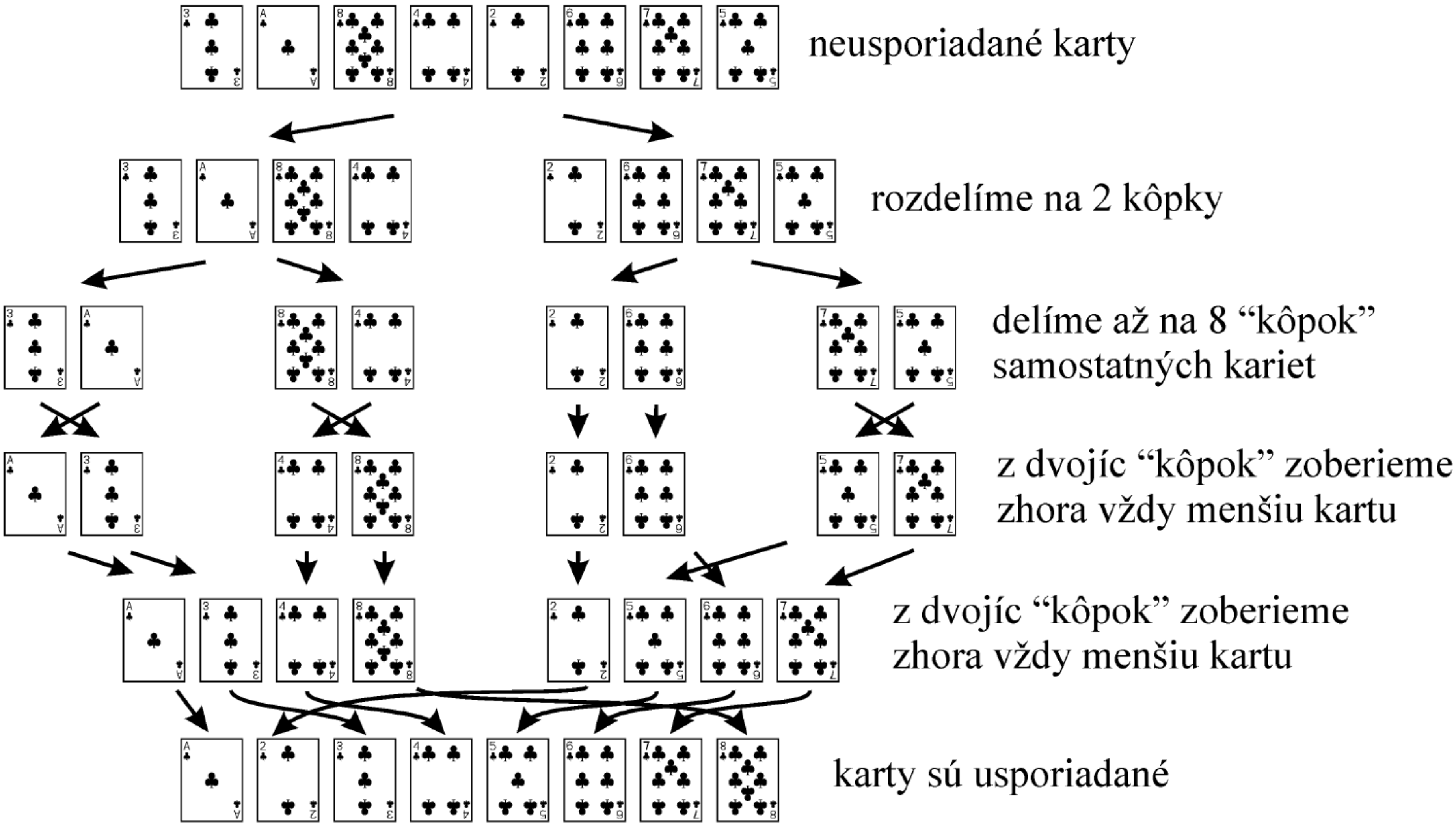
- Procedúra **merge**(input, left, middle, right) pomocou jednoduchého cyklu spojí usporiadané postupnosti prvkov input[left, …, middle] a input[middle+1, …, right] do jednej usporiadanej postupnosti; vyžaduje rádovo right-left (dĺžka poľa vstupujúceho do operácie) operácií.

Triedenie zlučovaním (Merge sort)

- Animovaná ukážka: https://visualgo.net/bn/sorting

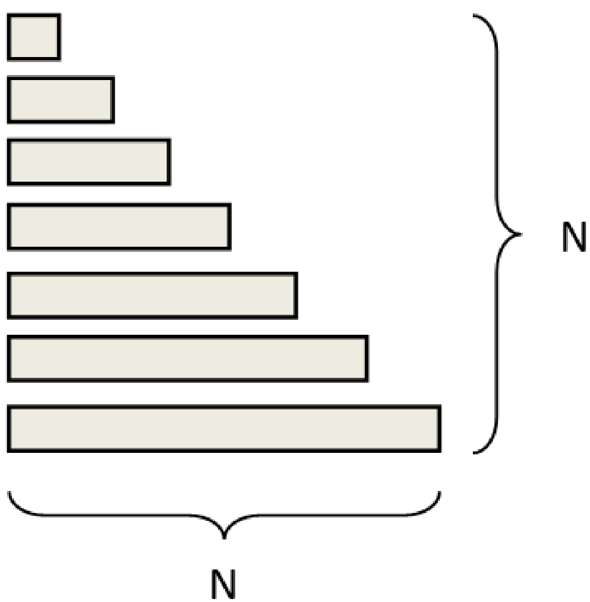


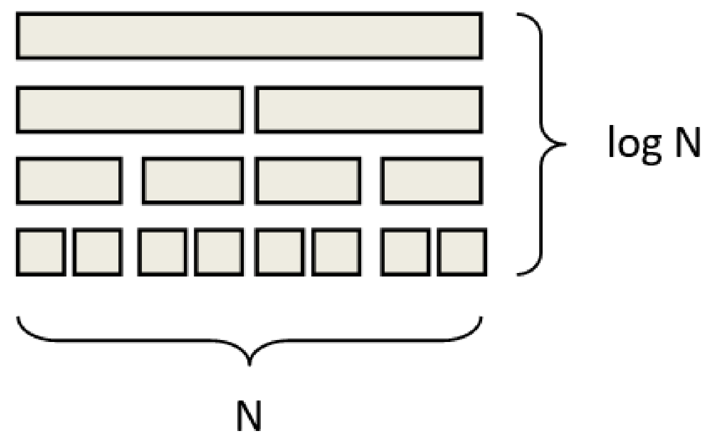
Ukážka triedenia zlučovaním hracích kariet



Zložitosť Insert sort vs. Merge sort

- Obe operácie **insert** a **merge** vykonajú počet operácií lineárne závislý od veľkosti poľa na vstupe. Algoritmy ale tieto procedúry volajú principiálne odlišným spôsobom, a preto celkový počet operácií, ktoré tieto algoritmy vykonajú je odlišný.





Zložitosť Insert sort vs. Merge sort

- Vzhľadom na vzájomnú komplikovanosť operácií môžeme určiť konštanty pre asymptotický odhad:

$\mathbf{c_{insert} = 4}$

$\mathbf{T_{insert}(N) = c_{insert} * N^2}$

$\mathbf{T_{insert}(10) = 4 * 10^2 = 400}$

$\mathbf{T_{insert}(10^7) = 4 * 10^7 * 10^7 =}$

$\mathbf{= 4 * 10^{14}}$

$\mathbf{c_{merge} = 50}$

$T_{merge}(N) = c_{merge} * N \log N$

$T_{merge}(10) = 50 * 10 * 3 = 1500$

$T_{merge}(10^7) = 50 * 10^7 * 20 =$

$= 10^{10}$

Abstraktné dátové typy (ADT)

- Všeobecný model pre dátový typ (dátovú štruktúru) vyjadrený pomocou abstrakcie:
  - Určíme operácie s dátovým typom a ich vlastnosti
  - Abstrahujeme od konkrétnej implementácie
- ADT môžeme implementovať rôznymi spôsobmi bez toho, aby to ovplyvnilo správnosť behu programu-algoritmu, ktorý ADT používa

Dátový typ

- Každá hodnota v programe má dátový typ
- Množina použiteľných dátových typov je určená použitým programovacím jazykom
- Dátový typ premennej určuje
  - **Množinu hodnôt**, ktoré možno dátovým typom reprezentovať
  - **Vnútornú reprezentácie v počítači** (využitie-kódovanie v pamäti)
  - **Prípustné operácie**, ktoré možno nad hodnotami daného typu vykonávať

Jednoduché dátové typy

- Boolean
  - Množina hodnôt {**true**, **false**}
  - Reprezentácia v pamäti ako 1 byte
  - Prípustné operácie: NOT, AND, OR
- Štandardne v jazyku C:
  - char, int, float, double
  - ukazovateľ (smerník)
  - Zložené: pole, struct, union

Abstraktný dátový typ vs. dátová štruktúra

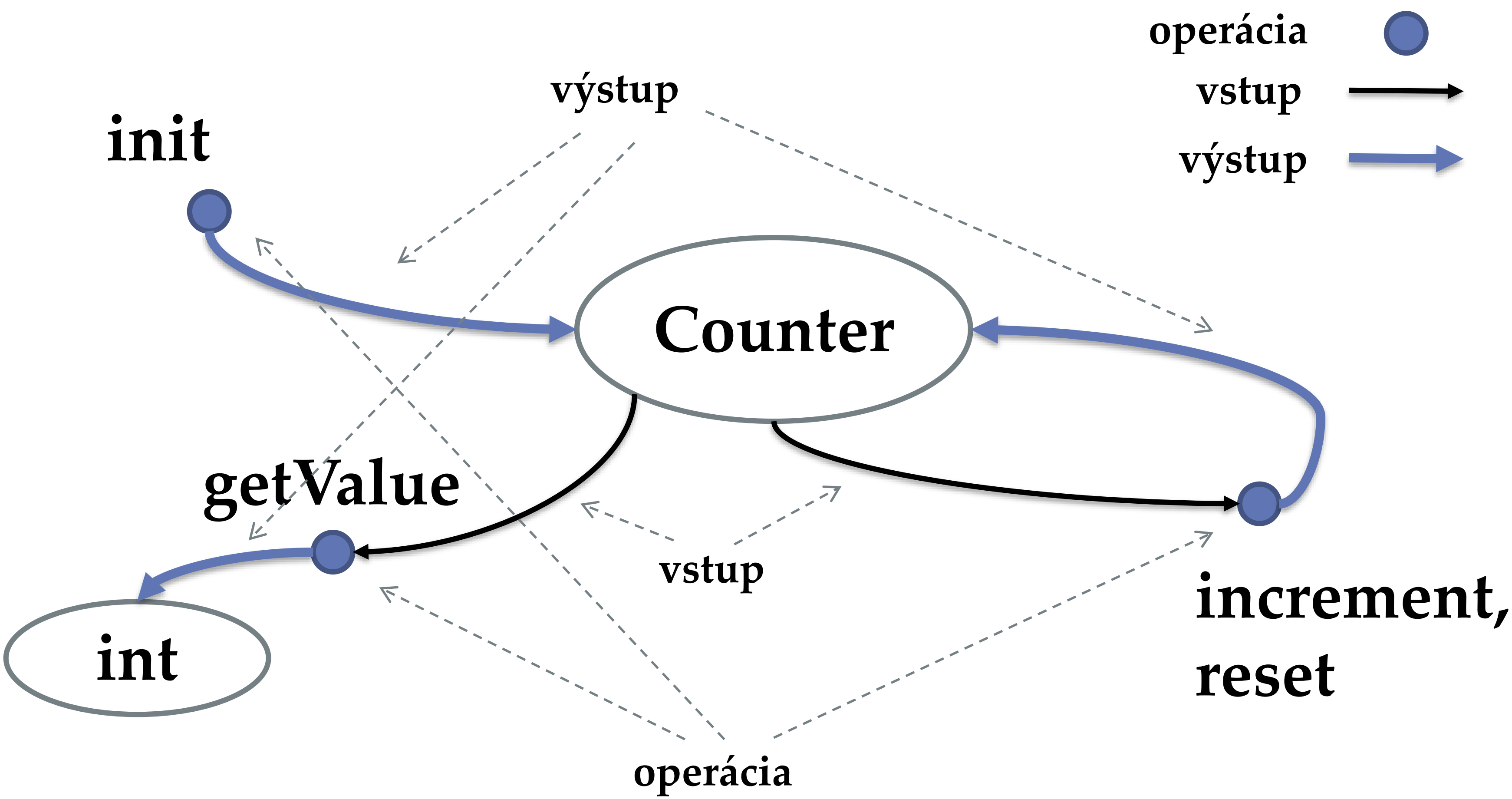
- Abstraktný dátový typ
  - Množina typov údajov a operácií, ktoré sú špecifikované nezávisle od konkrétnej implementácie
  - Reprezentuje model zložitejšieho dátového typu
  - Abstraktný model
- Dátová štruktúra
  - Implementácia ADT v programovacom jazyku
  - Reprezentácia typov údajov v ADT
  - Voľba algoritmov pre implementáciu operácií ADT

Definícia ADT

- Formálne
  - Signatúra a axiómy
- Programátorsky
  - Definícia rozhranie s operáciami

- Ukážka: **Počítadlo – ADT**
  - Stevard v lietadle počíta cestujúcich ...

Počítadlo (Counter) – signatúra



Počítadlo (Counter) – axiómy

- Opisujú vlastnosti – význam (sémantiku) operácií prostredníctvom ekvivalencie výrazov

  Pre všetky C ∈ Counter platí:

  getValue (init) = 0
  getValue (increment(C)) = getValue(C) + 1
  reset (C) = init

Počítadlo – Programátorské rozhranie

- Definícia rozhrania s operáciami

```
int getValue();
void increment();
void reset();
```

- Možná implementácia:

```
int value = 0;
int getValue() { return value; }
void increment() { value++; }
void reset() { value = 0; }
```

Počítadlo – Programátorské rozhranie

- Definícia rozhrania v jazyku C

```c
int getValue(struct Counter *c);
void increment(struct Counter *c);
void reset(struct Counter *c);
```

- Možná implementácia v jazyku C:

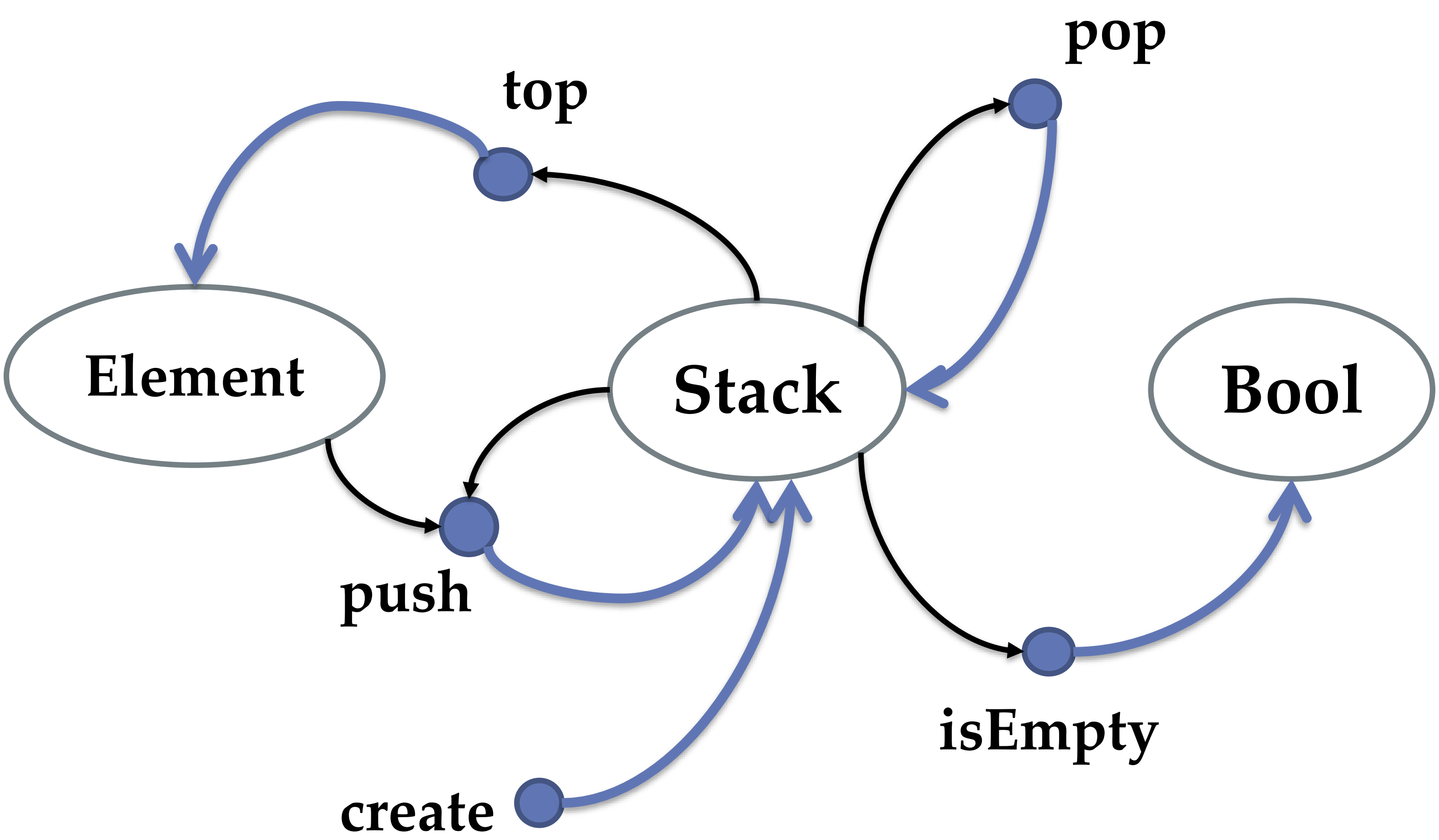
```c
struct Counter
{
  int value;
};

int getValue(struct Counter *c) { return c->value; }
void increment(struct Counter *c) { c->value++; }
void reset(struct Counter *c) { c->value = 0; }
```

Ďalšie ADT (s obmedzeným prístupom)

- **Zásobník (Stack)**
  - Push (vlož na vrch zásobníka)
  - Pop (vyber z vrchu zásobníka)
  - LIFO – Last In, First Out
- **Rad-Front (Queue)**
  - Enqueue (pridaj nakoniec do radu-fronty)
  - Dequeue (odober zo začiatku radu-fronty)
  - FIFO – First In, First Out

Zásobník (Stack) – Signatúra



Zásobník (Stack) – axiómy

Pre všetky $S \in$ Stack, $e \in$ Element platí:

isEmpty(create) = true

isEmpty(push(S,e)) = false

pop(create) = error

pop(push(S,e)) = S

top(create) = error

top(push(S,e)) = e

Zásobník (Stack) – implementácia poľom

```
Stack S:
create(S)
   top(S) ← 0

push(S,x)
   top(S) ← top(S) + 1
   S[top(S)] ← x

pop(S)
   if isEmpty(S)
        then error"underflow"
        else top(S) ← top(S) - 1
              return  S
```

Zásobník – Implementácia poľom v jazyku C

- Lenivá implementácia, globálna premenná

```
int stack[MAX], head;

void push(int v) { stack[head++]=v; }
int pop(void) { return stack[--head]; }
void create(void) { head = 0; }
int isEmpty(void) { return !head; }
```

Čo chcem, aby ste si odniesli z tohto predmetu

- Prehľad nástrojov (algoritmov a dátových štruktúr) pre riešenie rozličných problémov
- Porozumieť vlastnostiam týchto nástrojov, najmä:
  - časovej efektívnosť (výpočtovej zložitosti)
  - pamäťovej efektívnosti (priestorovej zložitosti)
- Schopnosť aplikovať a prispôsobiť-upraviť tieto nástroje pre špecifické problémy
  (za účelom dosiahnutia čo najlepšej efektívnosti)
- Otvorenosť pre ďalšie štúdium nových algoritmov a použitie efektívnych algoritmov a dátových štruktúr vo vašej ďalšej práci

Podmienky absolvovania

- Môžete získať až 100 bodov
- priebežne riešené úlohy (zadania)
  (max. 50 bodov: na cvičeniach 20 a doma 30):
  - na cvičení sa budú riešiť **malé úlohy**; (môže ich byť viac, každé najviac za 2 body), do konečného hodnotenia sa započítava najviac 20 bodov za všetky malé úlohy;
  - doma sa budú riešiť 3 **zadania**
    každé max. 10 bodov, min. 4.
- priebežný test (max. 15 bodov, treba získať min. 10)
- Podmienky udelenia zápočtu:
  - minimálne 25 bodov z priebežne úloh (vrátane zadaní)
  - minimálne 5 bodov z priebežného testu
- záverečná skúška (max. 35 bodov, treba získať min. 15)