Algoritmy 1

část 1

Radim BĚLOHLÁVEK
Katedra informatiky
Univerzita Palackého v Olomouci

Základní informace

- **webové stránky**
  - http://belohlavek.inf.upol.cz/vyuka/alm1-2018-19.html (předmět)
  - http://belohlavek.inf.upol.cz/ (přednášející)
  - další (cvičící, STAG, katedra informatiky)
- **studium**
  - přednášky (vede přednášející)
  - cvičení (vede a podmínky určuje cvičící)
  - konzultační hodiny (využívat)
  - samostanté studium (studium literatury, promýšlení problémů, praktické u počítače)
- **absolvování předmětu**
  - zápočet (udělí cvičící, získat alespoň 60% bodů za zadaných úkolů: 1 písemný test, 1 program)
  - zkouška (vypsané předtermíny a termíny)

Charakteristika předmětu, doporučení

- co je obsahem předmětu
  - základní algoritmy a datové struktury v informatice
  - návrh, analýza a implementace
- charakter předmětu
  - jeden z nejdůležitějších předmětů pro informatiky
  - vyžaduje schopnost kombinatorického uvažování
  - není primárně předmětem o programování, ale schopnost programovat je potřebná k procvičování
  - není typickým matematický předmětem, ale přesnost matematického uvažování je zde typická
  - související předměty: Paradigmata programování 1, Základy programování 1
- rady pro studium
  - chodit na předášky a cvičení (šetří čas studia, rozpozná důležité od méně důležitého)
  - implementovat probírané algoritmy (programovat, $>$ 10 hod./týden)
  - snažit se věci pochopit do hloubky, neučit se zpaměti (během studia se témata v obměnách opakují, pochopení na začátku studium usnadní)
  - příprava na zkoušku (individuální, 2 dny je málo, spíš týden)

Další informace

- na katedře
  - možnost zapojit se do studentských soutěží
  - možnost zapojit se do výzkumu
  - možnost studentských zahraničních pobytů
- chyby v tomto studijním textu
  - hlaste mi prosím osobně nebo e-mailem
- bonus
  - zkoušku se známkou "výborně" získá ten, kdo prokazatelně přispěje originálním výsledkem k rozvoji oblasti algoritmů (posuzuje přednášející)
  - např. nový algoritmus přinášející zlepšení oproti současnému stavu, nový výsledek o složitosti algoritmu (teoretický nebo experimentální)

Algoritmy
základní úvahy

Co je to algoritmus?

První přiblížení ("definice"):
Algoritmus je posloupnost instrukcí pro řešení problému.

Tato "definice" je nepřesná (tedy to vlastně není definice):

– Co je to problém?
– Co je to instrukce?
– Co to znamená řešit problém?

Názorně ale vystihuje podstatu pojmu algoritmus.

Existuje řada podobných "definic" pojmu algoritmus, více či méně rozsáhlých a příp. přidávajících další omezení. Např.

"an algorithm is a finite procedure, written in a fixed symbolic vocabulary, governed by precise instructions, moving in discrete steps, 1, 2, 3, . . . , whose execution requires no insight, cleverness, intuition, intelligence, or perspicuity, and that sooner or later comes to an end."
–D. Berlinsky, The Advent of the Algorithm

Tedy, co je to problém, instrukce, řešit problém?

Problém

V běžném životě hovoříme např. o následujících problémech:

1. Problém zaparkování auta.
2. Problém výpočtu řešení lineární rovnice (tj. rovnice tvaru $ax + b = 0$).
3. Problém zjištění, zda dané přirozené číslo n je prvočíslo.
4. Izraelsko-palestinský problém.
5. Problém jak žít smysluplný život.

1, 2, 3 jsou příklady problémů, o kterých se hovoří v definici algoritmu uvedené dříve a které nás budou v dalším zajímat. Problémy 4 a 5 ne.

Problém (např. problém 1, 2 nebo 3) je popsán (zadán, specifikován):

– množinou přípustných zadání (vstupů),

– přiřazením (předpisem, zobrazením), který pro každé přípustné zadání (vstup) říká, jaké je odpovídající (tj. správné) řešení (výstup).

Problém se někdy přímo chápe jako přiřazení, které každému přípustnému vstupu přiřazuje odpovídající výstup.

Takové pojetí (problém = množina přípustných vstupů a přiřazení) dává poměrně přesné vymezení pojmu problém, které nám zatím stačí.

Příklady:

1. Problém zaparkování auta.
   Přípustným vstupem je popis S počáteční dopravní situace (zahrnuje polohu auta, které je třeba zaparkovat, popis okolní situace včetně místa, kam je třeba zaparkovat).
   Přiřazení specifikuje pro každý S popis správné koncové situace T (tj. ve které je auto správně zaparkováno).
   (Popisy S a T mohou být značně komplikované.)
2. Problém výpočtu řešení lineární rovnice.
   Přípustné vstupy jsou dvojice $\langle a, b\rangle$ racionálních čísel a a b.
   Přiřazení říká, že výstupem odpovídajícím vstupu $\langle a, b\rangle$ je
   - číslo c, pro které $ac + b = 0$, pokud $a \neq 0$,
   - text "Každé číslo je řešením", pokud $a = 0$ a $b = 0$.
   - text "Nemá řešení", pokud $a = 0$ a $b \neq 0$.

3. Problém zjištění, zda dané přirozené číslo n je prvočíslo.
Přípustné vstupy jsou přirozená čísla n.
Přiřazení říká, že výstupem odpovídajícím vstupu n je
"Ano", pokud n je prvočíslo,
"Ne", pokud n není prvočíslo.

Problémy 2 a 3 jsou typické příklady problémů, kterými se budeme zabývat. Zkrácený popis problému:

problém (*název problému*)
vstup: *popis přípustného vstupu*
výstup: *popis odpovídajícího výstupu*

Příklad:

problém (test prvočíselnosti)
vstup: přirozené číslo n
výstup: "Ano", pokud n je prvočíslo; "Ne", pokud n není prvočíslo.

Instrukce

Instrukce je jednoznačný srozumitelný pokyn jako např.:

- Sešlápni brzdový pedál.
- Sečti čísla a a b. (aritmetická instrukce)
- Do proměnné x ulož číslo 5. (instrukce přiřazení)
- Pokud je $C > 0$, pak zvyš hodnotu b o 1. (podmíněná instrukce)
- Přečti číslo, které je na vstupu. (vstupně/výstupní instrukce)
- Vytiskni hodnotu proměné x. (vstupně/výstupní instrukce)
- Pro každé $i = 1, 2, 3, 4, 5$ postupně proveď: vytiskni hodnotu i^2 (instrukce cyklu; v těle cyklu je vstupně/výstupní instrukce)

Pojem instrukce (opět nepřesně definovaný) vychází z představy, že existuje někdo, např. člověk (nebo něco, např. počítač), kdo (co) instrukcím rozumí a je schopen je mechanicky, tj. bez dalšího přemýšlení, vykonávat. Tento vykonavatel instrukcí je schopen vykonávat instrukce tak, jak jsou předepsány algoritmem (tj. ve správném pořadí).

Řešení problému algoritmem — co to znamená?

Algoritmus řeší daný problém, pokud pro každý přípustný vstup I daného problému, jemuž odpovídá výstup O (tj. O je správný výstup pro vstup I) platí:
Vykonávání instrukcí podle algoritmu se vstupem I se po určité době (po konečném počtu kroků) zastaví a na výstupu je O.

Tj. vykonáváním instrukcí podle algoritmu se od vstupu I po konečném počtu kroků dobereme k O.

Říkáme, že algoritmus se pro vstup I zastaví s výstupem O.

Přitom se předpokládá, že vstup I je na začátku vykonávání instrukcí zapsán na dohodnutém vstupním zařízení (soubor na disku, je zadán z klávesnice apod.) a že výstup se objeví na dohodnutém výstupním zařízení (soubor na disku, obrazovka apod.).

Zpět k problému výpočtu řešení rovnice $ax + b = 0$

Posloupnost instrukcí 1:
- Pokud $a \neq 0$, zapiš na výstup číslo $-b/a$.
- Pokud $a = 0$ a $b = 0$, zapiš na výstup "Každé číslo je řešením".
- Pokud $a = 0$ a $b \neq 0$, zapiš na výstup "Nemá řešení".

Posloupnost instrukcí 2:
- Pokud $a \neq 0$, zapiš na výstup číslo $-b/a$, jinak zapiš číslo b/a.

Posloupnost instrukcí 3:
- Dokud $a \neq 0$, prováděj: zvyš hodnotu b o 1.
- Pokud $a = 0$ a $b \neq 0$, zapiš na výstup "Nemá řešení".

Posloupnost 1 je algoritmus pro řešení daného problému. (Velmi jednoduchý algoritmus pro velmi jednoduchý problém. Další budou složitější.)
Posloupnosti 2 a 3 ne (2 nedává správné výstupy, 3 se pro některé vstupy nezastaví).

Za algoritmy nelze považovat ani následující posloupnosti:

Posloupnost 4:
Zkus odhadnout řešení.
Vyzkoušej, zda je správné.
Pokud ano, zapiš ho na výstup.
Pokud ne, zpřesni odhad a jdi dál.

Není to algoritmus, protože není jasné, jak postupovat.

Posloupnost 5:
Spusť na PC program Mathematica.
Vyřeš v něm rovnici.
Výsledek zapiš na výstup.

Není to algoritmus, odvolává se na zařízení (jeho existence a dostupnost je časově podmíněna), které problém vyřeší.

Existuje přesná definice pojmu algoritmus a dalších pojmů?

Ano. Zabývá se tím informatická disciplína zvaná teorie vyčíslitelnosti (computability theory).

Poznamenejme, že existují různé názory na to, co pojem algoritmus přesně znamená, a tedy různé definice pojmu algoritmus. Téměř všechny definice lze však chápat jako zpřesnění výše uvedené nepřesné definice.

Ještě jednou. Naše nepřesná definice, která říká

"Algoritmus je posloupnost instrukcí pro řešení problému."

není vlastně definicí. Popisuje intuitivní představu o tom, co to je algoritmus.

Co řekli o algoritmech

Algorithmics is more than a branch of computer science. It **is the core of computer science**, and, in all fairness, can be said to be relevant to most of science, business, and technology.
—D. Harel, Algorithmics: The Spirit of Computing, 1992

Two ideas lie gleaming on the jeweler's velvet. The first is the calculus, the second the algorithm. The calculus and the rich body of mathematical analysis to which it gave rise made modern science possible; but **it has been the algorithm that has made possible the modern world**.
—D. Berlinski, The Advent of the Algorithm, 2000

Co řekli o pojmu algoritmus

A person well-trained in computer science knows how to deal with algorithms: how to construct them, manipulate them, understand them, analyze them. This knowledge is preparation for much more than writing good computer programs; it is a general-purpose mental tool that will be a definite aid to understanding other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm. . . . An attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.
—D. E. Knuth, Selected Papers on Computer Science, 1996

Donald E. Knuth

Professor Emeritus of The Art of Computer Programming
Stanford University
`http://www-cs-faculty.stanford.edu/~uno/`
jeden z nejvýznamnějších informatiků
autor několikasvazkového díla
"The Art of Computer Programming"
autor systému TeX

Několik zásadních fakt o algoritmech a problémech

... ke kterým se budeme v tomto předmětu i v dalších předmětech vracet.

– Je každý problém řešitelný algoritmem?
Ne, existují přirozené problémy, které nelze řešit žádným algoritmem (algoritmicky neřešitelené).

– Má smysl říkat, že problémy jsou lehké a těžké, že některé problémy jsou lehčí než jiné?
Ano, zabývá se tím teorie vyčíslitelnosti.

– Má klasifikace problémů podle obtížnosti nějaký praktický význam?
Ano, základní je dělení algoritmicky řešitelných problémů na rychle řešitelné a ty, pro které rychlý algoritmus není znám. Prakticky významné jsou oba typy problémů.

– Lze o algoritmech říkat, že jeden je lepší (efektivnější) než druhý?
Ano, zabývá se tím teorie složitosti a je to prakticky velmi důležité. Budeme se tím zabývat i v tomto předmětu.

– To vypadá zajímavě, ale asi to bude náročné nastudovat. Dá se to a budu všemu rozumět?
  Ano, dá se to. Prošly tím stovky jiných. Ne všemu je třeba rozumět do nejmenších detailů. Něco je třeba pochopit hlouběji, někde stačí pochopit základní principy.

Jak popsat (specifikovat) algoritmus?

Budeme se zabývat zejm. algoritmy, které jsou určeny pro počítač (tj. instrukce vykonává počítač). Základní způsoby popisu takových algoritmů jsou:

– Přirozeným jazykem. To jsme už viděli.
  +: Snadno srozumitelné i laikům.
  −: Může být nejednoznačné. Zdlouhavé.
– Programovacím jazykem. Budeme používat jazyk C (ale je možné použít jakýkoli programovací jazyk).
  +: Jednoznačné.
  +: Vytvořit počítačový program je pak snadné (téměř ho máme). Rozumí tomu programátoři.
  −: Obsahuje i zbytečné (nepodstatné) detaily. Dlouhé.

- Pseudokódem. Pseudokód je jazyk blízký programovacímu jazyku, ale je úspornější, protože neobsahuje tolik podrobností. Budeme používat pseudokód z knihy Cormen et al.: Introduction to Algorithms, 2nd Ed. MIT Press, 2001.
  - +: Je snadno pochopitelný i pro ty, kteří nemají zkušenosti s programováním a programovacími jazyky. Je vytvořený přímo pro srozumitelný a úsporný popis algortmů. Umožňuje snadný přepis do programovacích jazyků.
  - −: Snad to, že při implementaci algoritmu je třeba ho přepsat do programovacího jazyka.
- Dalšími (polo)formálními prostředky (vývojové diagramy, ...).

Algoritmus sčítání v desítkové soustavě

sčítání přirozených čísel v desítkové soustavě (základní škola)

$$\begin{array}{rcccc} & 1 & 0 & 9 & 3 \\ + & 2 & 3 & 4 & 5 \\ \hline & 3 & 4 & 3 & 8 \end{array}$$

protože:

postupujeme zprava (od poslední cifry) doleva

$3+5=8 \Rightarrow$ napíšeme 8 a přenášíme 0

$9+4=13 \Rightarrow$ napíšeme 3 a přenášíme 1

$1+0+3=4 \Rightarrow$ napíšeme 4 a přenášíme 0

$1+2=3 \Rightarrow$ napíšeme 3 a přenášíme 0

podobně

$$\begin{array}{rcccc} & 9 & 7 & 2 & 8 \\ + & 2 & 3 & 9 & 9 \\ \hline 1 & 2 & 1 & 2 & 7 \end{array}, \quad \begin{array}{rcc} & 1 & 0 \\ + & 2 & 5 \\ \hline & 3 & 5 \end{array}, \quad \begin{array}{rcccc} & 0 & 0 & 1 & 3 \\ + & 8 & 6 & 8 & 2 \\ \hline & 8 & 6 & 9 & 5 \end{array}$$

Popišme přesně řešený problém.

problém (sčítání v desítkové soustavě):
vstup: $a_{n-1}a_{n-2}\cdots a_1a_0$, $b_{n-1}b_{n-2}\cdots b_1b_0$,
kde a_i, b_i jsou číslice z množiny $\{0, 1, \dots, 9\}$
výstup: $c_nc_{n-1}\cdots c_1c_0$,
(posloupnost), která je zápisem v desítkové soustavě čísla $A + B$, kde A a B jsou čísla jejichž zápisy jsou posloupnosti $a_{n-1}\cdots a_0$ a $b_{n-1}\cdots b_0$

Poznámky:

– Prvky n-prvkové posloupnosti indexujeme pomocí $0, 1, \dots, n-1$, a ne $1, 2, \dots, n$ (bývá to výhodnější, uvidíme).
– Místo a_i také píšeme $a[i]$ (tak se to píše v programovacích jazycích).

Popis v přirozeném jazyku

1. Je-li $a[0] + b[0] < 10$, nastav hodnoty $c[0]$ na $a[0] + b[0]$ a t na 0;
   jinak nastav $c[0]$ na $a[0] + b[0] - 10$ a t na 1;
2. Je-li $a[1] + b[1] + t < 10$, nastav $c[1]$ na $a[1] + b[1] + t$ a t na 0;
   jinak nastav $c[1]$ na $a[1] + b[1] + t - 10$ a t na 1;

   ...

n. Je-li $a[n-1] + b[n-1] + t < 10$, nastav $c[n-1]$ na $a[n-1] + b[n-1] + t$ a t na 0;
   jinak nastav $c[n-1]$ na $a[n-1] + b[n-1] + t - 10$ a t na 1;

n+1. nastav $c[n]$ na t.

Vidíme, že popis v přirozeném jazyku je těžkopádný a nepřehledný.

Jiný popis v přirozeném jazyku

1. Nastav t na 0.
2. Pro hodnoty i od 0 do $n-1$ prováděj postupně:

   Je-li $a[i]+b[i]+t<10$, nastav $c[i]$ na $a[i]+b[i]+t$ a t na 0;
   jinak nastav $c[i]$ na $a[i]+b[i]+t-10$ a t na 1;

3. nastav $c[n]$ na t.

Zařadili jsme konstrukci zvanou cyklus ("Pro hodnoty i od ..."). Stále to není ono.

Popis v pseudokódu

```
Scitani-Cisel-Desitkove(a[0..n − 1], b[0..n − 1])
t ← 0
for i ← 0 to n − 1
  do c[i] ← a[i] + b[i] + t mod 10
     t ← ⌊(a[i] + b[i] + t)/10⌋
c[n] ← t
```

Poznamenejme:

– $a \bmod n$ je zbytek po dělení čísla a číslem n
  př.: $5 \bmod 2 = 1$, $12 \bmod 10 = 2$, ...

– $\lfloor a \rfloor$ je největší celé číslo, které je $\leq a$
  $\lfloor 0.5 \rfloor = 0$, $\lfloor 1.2 \rfloor = 1$, $\lfloor 1.0 \rfloor = 1$, ...

Popis v programovací jazyku C

```c
int ScitaniCiselDesitkove (int *a, int *b, int *c)
{
  int i,t;

  t=0;
  for (i = 2; i < n; i++){
    c[i] = (a[i] + b[i] + t) % 10;
    t = (a[i] + b[i] + t) / 10;
  }
  c[n]=t;
  return 0;
}
```

Není tak přehledné (a srozumitelné) jako pseudokód.

Shrnutí

Probrali jsme:

- pojem algoritmus
- pojmy problém, instrukce, řešení problému algoritmem
- příklady problémů
- příklady algoritmů
- jak popsat algoritmus

Základní instrukce používané v algoritmech

- přiřazení
- aritmetické instrukce a výrazy
- logické instrukce a výrazy
- podmíněný příkaz
- cykly
- další

přiřazení:

$i \leftarrow 1$
$j \leftarrow i + 1$

$i \leftarrow 1$
$i \leftarrow i * i$

$i \leftarrow 1$
$i \leftarrow i * i$

$i \leftarrow 1$
$i \leftarrow i * i$
$i \leftarrow 5$

aritmetické instrukce a výrazy:

$+$ (sčítání), $-$ (odčítání), $*$ (násobení), $/$ (dělení),

$x \bmod y$ (zbytek po dělení čísla x číslem y):
$5 \bmod 2 = 1$, $5 \bmod 3 = 2$, $6 \bmod 2 = 0$

případně další

příklady:

```
1  p ← 7
2  q ← p * p + 2 * p − 8
```


logické instrukce a výrazy:

and (konjunkce, logické "a"),
or (disjunkce, logické "nebo")

$=$ (test na rovnost hodnot),
$<>$ nebo $!=$ (test na nerovnost hodnot),
$<$ (porovnání dle velikosti, např. čísel),

$i = 5$ (test na rovnost hodnoty proměnné i a hodnoty 5, výsledkem je logická hodnota pravda, nebo nepravda)

$2 < j$, $2 * j <> 6$, apod.

$(i = 5)$ and $(j < 10)$
(má hodnotu pravda, právě když mají oba výrazy $i = 5$ i $j < 10$ hodnotu pravda)

$(i = 5)$ or $(j < 10)$
(má hodnotu pravda, právě když má aspoň jeden z výrazů $i = 5$ a $j < 10$ hodnotu pravda)

podmíněný příkaz:

if-then: příkaz se vykoná, pokud je splněna podmínka:

```
if i = 0 then print(Hodnota je 0.)
```

if-then-else: pokud podmínka není splněna, vykoná se příkaz za else

```
if i = 0 then print(Hodnota je 0.)
          else print(Hodnota není 0.)
```

```
if i = 0 then print(Hodnota je 0.)
else if i = 1 then print(Hodnota je 1.)
     else print(Hodnota není ani 0, ani 1.)
```

Odsazením textu vyjadřujeme logickou strukturu programu:

```
if i = 0 then print(Hodnota je 0.)
                print(Ještě jednou: hodnota je 0.)
          else print(Hodnota není 0.)
```


cyklus for:
cyklus s řídicí proměnnou (čítačem)

```
for i ← 0 to n do
  print(i)
```

```
for i ← 1 to 9 do
  for i ← 1 to 9 do
    print(i)
    print(∗)
    print(j)
    print(=)
    print(i ∗ j)
```

cyklus while-do:
vykonává se, dokud platí podmínka za slovem while

```
i ← 1
while i < 10 do
  print(i * i)
  i ← i + 1
```

```
nactiPole(a[0..9])
i ← 0
while (a[i] <> 0 and i < 10) do
  print(i)
  i ← i + 1
```


cyklus repeat-until:
vykonává se, dokud není splněna podmínka za slovem until
(vykoná se aspoň jednou)

```
i ← 0
repeat
  i ← i + 1
  print(i * i)
until i = 10
```

```
i ← 1
repeat
  read(i)
  if i mod 2 = 0 then print(Zadal jsi sudé číslo.)
  else print(Zadal jsi liché číslo.)
until i = 0
```

Základní otázky o algoritmech

– **Správnost algoritmu**:
  Je dán problém. Skončí navržený algoritmus pro každý přípustný vstup problému po konečném počtu kroků a se správným výstupem? Tj. jde vůbec o algoritmus řešící daný problém?
  Pro každý navržený algoritmus je třeba ověřit (dokázat), zda je správný. Jinak může hrozit vážné riziko (řízení složitých systémů—elektrárna, ABS systém u aut, . . .).
  Zda je algoritmus správný, je někdy snadno vidět, může to ale být náročné.
– **Složitost algoritmu**:
  Jak dlouho trvá vykonávání algoritmu (tj. jaká je jeho časová složitost)?
  Kolik paměti algoritmus potřebuje (tj. jaká je jeho paměťová složitost)?
  Složitost nám dává informaci o tom, jak je algoritmus kvalitní a umožňuje ho z tohoto pohledu porovnat s jinými algoritmy.

– **Optimalita algoritmu**:
  Existuje lepší algoritmus?
  Pro některé algoritmy lze dokázat, že lepší neexistují, tedy nemá smysl je hledat. (Co to ale znamená "lepší"? Je třeba rozebrat.)

Nyní vysvětlíme intuitivně a na příkladech.
Později se k tomu vrátíme a vysvětlíme přesně.

Správnost algoritmu – příklad

Scitani-Cisel-Desitkove($a[0..n-1], b[0..n-1]$)

$t \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$
  do $c[i] \leftarrow a[i] + b[i] + t \bmod 10$
    $t \leftarrow \lfloor (a[i] + b[i] + t)/10 \rfloor$
$c[n] \leftarrow t$

Jak dokážeme správnost našeho algoritmu?
V tomto případě je to snadné (někdo by řekl zbytečné). Pro úplnost to dokažme (když se to na základí škole neudělá, děti se učí algoritmus nazpamět' bez pochopení).

Tvrzení Algoritmus Scitani-Cisel-Desitkove je správný.

Důkaz Číslo A si představíme jako A korálků. Zápisu $a[n-1]a[n-2]\cdots a[0]$ čísla A v desítkové soustavě odpovídá následující uspořádání korálků (uspořádání se snadněji namaluje než slovně popisuje).

Krok 1: Korálky navlékáme na nit po 10 (nit s 10 korálky svážeme a říkáme jí svazek). Tak získáme několik svazků (každý má 10 korálků) a zbyde právě $a[0]$ nenavlečených korálků. (Udělejte si příklad, pro $A = 123$ získáme 12 svazků a 3 korálky zbydou). Zbylé korálky dáme stranou.

Krok 2: Postupujeme dál tak, že k sobě svazujeme vždy 10 svazků získaných v předchozím kroku. Tak získáme několik větších svazků (každý má 100 korálků) a zbyde právě $a[1]$ nesvázaných svazků o 10 korálcích. (Pro $A = 123$ získáme 1 svazek o 100 korálcích a zbydou 2 svazky o 10 korálcích). Zbylé svazky dáme stranou.

Postupujeme dál, až dojdeme do kroku n, ve kterém zbývá méně než 10 svazků s 10^{n-1} korálky. Počet těchto svazků je právě $a[n-1]$. (Pro $A = 123$ tak dojdeme do kroku 3, ve kterém zbývá 1 svazek o 100 korálcích.)

Tak si představíme i číslo B.

Svazky korálků a korálky, které jsme získali z A korálků představujících číslo A a B korálků představujících číslo B dáme k sobě. Získáme tak $C = A + B$ korálků.

Abychom získali uspořádání těchto korálků, které odpovídá zápisu C v desítkové soustavě, musíme je správně uspořádat. Protože korálky již jsou uspořádané ve svazcích, dojdeme k požadovanému uspořádání následovně.

Dáme dohromady jednotlivé korálky (je jich $a[0] + b[0]$). Je-li jich alespoň 10, vytvoříme nový svazek o 10 korálcích. Korálky, které nejsou v novém svazku, dáme stranou, nový svazek ponecháme. Je jasné, že stranou dáme $a[0] + b[0] \bmod 10$ korálků a že vznikne $\lfloor (a[0] + b[0])/10 \rfloor$ nových svazků (žádný nebo jeden). To jsou právě čísla přiřazená $c[0]$ a t na řádcích 3 a 4 v kroku pro $i = 0$.

V obecném kroku pro i dáme dohromady svazky s 10^i korálky (je jich $a[i] + b[i] + t$, kde t je počet nových svazků vytvořených v předchozím kroku). Je-li jich alespoň 10, vytvoříme nový svazek o 10^{i+1} korálcích. Svazky o 10^i korálcích, které nejsou v novém svazku, dáme stranou, nový svazek ponecháme. Je jasné, že stranou dáme $a[i] + b[i] + t \bmod 10$ svazků a že vznikne $\lfloor (a[i] + b[i] + t)/10 \rfloor$ nových svazků (žádný nebo jeden). To jsou právě čísla přiřazená $c[i]$ a t na řádcích 3 a 4 v kroku pro i.

Tak dojdeme až k $i = n$, kdy opustíme cyklus a kdy zbývá $\lfloor (a[n-1] + b[n-1] + t)/10 \rfloor$ svazků o 10^n korálcích. To je správná hodnota $c[n]$ je to také hodnota přiřazená $c[n]$ na řádku 5.

Důkaz je hotov.

Je důkaz správnosti nutný?

Ano. U mnoha algoritmů je ale správnost zřejmá, takže důkaz odbydeme slovy "Je zřejmý".

Důkazem správnosti není, když ukážeme, že algoritmus správně funguje pro několik zvolených vstupů (nemusí totiž správně fungovat pro další vstupy).

U jiných algoritmů je důkaz správnosti komplikovaný. Když chceme algoritmus "jen" použít, nemusíme důkaz správnosti číst, pokud algoritmus převezmeme z důvěryhodného zdroje (např. kniha o algoritmech).

V tomto předmětu se ale správností algoritmů budeme zabývat.

Důkaz správnosti může mít mnoho podob

Jiná podoba důkazu správnosti Scitani-Cisel-Desitkove (hutná verze, můžete přeskočit):

Je $A = \sum_{i=0}^{n-1} a[i] * 10^i$, $B = \sum_{i=0}^{n-1} b[i] * 10^i$, tedy pro $C = A + B$ a jeho zápis $C = \sum_{i=0}^{n-1} c[i] * 10^i$ musí platit:

$$
\begin{aligned}
c[0] &= (a[0] + b[0]) \bmod 10, \\
c[1] &= (a[1] + b[1] + \lfloor (a[0] + b[0])/10 \rfloor) \bmod 10, \\
&\dots \\
c[n-1] &= (a[n-1] + b[n-1] + \lfloor (a[n-2] + b[n-2] + \\
&\quad \lfloor (a[n-3] + b[n-3] + \cdots)/10 \rfloor))/10 \rfloor) \bmod 10, \\
c[n] &= \lfloor (a[n-1] + b[n-1] + \lfloor (a[n-2] + b[n-2] + \cdots)/10 \rfloor))/10 \rfloor
\end{aligned}
$$

což jsou právě hodnoty obdržené naším algoritmem.

Vždy je třeba najít vhodný kompromis mezi stručností a srozumitelností.

Když navržený algoritmus není správný

Jak to prokážu?

Musíme najít příklad, pro který algoritmus skončí s nesprávným výstupem (tzv. protipříklad).

Tj. najít alespoň jeden přípustný vstup I, pro který je správným výstupem O, ale pro který algoritmus skončí s jiným výstupem, $O' \neq O$, nebo neskončí vůbec.

Příklad nesprávného algoritmu (pro hledání nejkratší cesty)

Je dána síť měst, některá jsou spojena silnicí, jsou dána města Z (začátek) a K (konec). Jak najít nejkratší cestu z Z do K?

Přípustným vstupem je tedy popis sítě měst (např. ve formě použité níže) a dvojice měst Z a K; odpovídajícím výstupem je nejkratší cesta ze Z do K.

Návrh algoritmu: Začnu v Z a jdu do nejbližšího města M_1, ve kterém jsem ještě nebyl; v M_1 postupuji stejně (jdu do nejbližšího města M_2, ve kterém jsem ještě nebyl), až dojdu do K.

Tento algoritmus je nesprávný.

Proč?

Vstup 1: Síť
$\{(\{A, B\}, 1), (\{A, D\}, 5), (\{B, C\}, 2), (\{C, D\}, 1), (\{D, E\}, 2)\}$,
$Z = A, K = E$.
$(\{A, B\}, 1)$ znamená, že mezi A a B je silnice délky 2, atd. Algoritmus najde cestu A, B, C, D, E délky 6 $(1 + 2 + 1 + 2)$. To je skutečně nejkratší cesta.

ALE
Vstup 2: Síť
$\{(\{A, B\}, 1), (\{A, D\}, 5), (\{B, C\}, 4), (\{C, D\}, 1), (\{D, E\}, 2)\}$,
$Z = A, K = E$.
Algoritmus najde cestu A, B, C, D, E délky 8 $(1 + 4 + 1 + 2)$. To není nejkratší cesta! Nejkratší je A, D, E, má délku 7.

Vstup 2 je protipříklad (jeden z mnoha možných), který ukazuje, že navržený algoritmus není správný.

Největší společný dělitel a Euklidův algoritmus

problém (největší společný dělitel, greatest common divisor)
vstup: nezáporná celá čísla m, n, alespoň jedno z nich > 0
výstup: $\gcd(m, n)$ (největší společný dělitel m a n)

Poznámky:

$\gcd(m, n)$ je největší přirozené číslo, které dělí m i n (se zbytem 0).

k dělí m, což značíme $k|m$, právě když existuje přirozené číslo l tak, že $m = kl$.

Tedy $\gcd(3, 5) = 1$, $\gcd(3, 6) = 3$, $\gcd(12, 18) = 6$, atd.

První algoritmus (naivní):

```
gcd-naive(m, n)
    t ← min{m, n}
    while m mod t ≠ 0 or n mod t ≠ 0
      do t ← t − 1
    return t
```

Důkaz správnosti gcd-naive(m, n): Začíná s $t = \min\{m, n\}$, což je číslo $\geq \gcd(m, n)$. Dokud t není dělitelem obou m i n, hodnota t se sníží o 1. Vrácená hodnota t tedy je $\gcd(m, n)$. Vždy se zastaví, nejpozději pro $t = 1$, protože 1 dělí každé m a n.

Existuje ale lepší algoritmus (později vysvětlíme, v čem "lepší").

```
gcd-Euclid(m, n)
    while n ≠ 0
      do r ← m mod n
         m ← n
         n ← r
    return m
```

Algoritmus uvedl Eukleidés (cca 325–260 př. Kr) ve své knize Základy.

Jeden z nejstarších algoritmů vůbec.

Na první pohled není jasné, proč skutečně počítá největšího společného dělitele.

```
gcd-Euclid(m, n)
1   while n ≠ 0
2     do r ← m mod n
3        m ← n
4        n ← r
5   return m
```

Jak algoritmus pracuje? Podívejme se, jak pracuje pro $(m, n) = (84, 24)$.

Při prvním testu podmínky na ř. 1 je $(m, n) = (84, 24)$, instrukce cyklu 2–4 se provedou ($r \leftarrow 12$, neboť $12 = 84 \bmod 24$; $m \leftarrow 24$; $n \leftarrow 12$).

Následuje druhý test podmínky na ř. 1 s $(m, n) = (24, 12)$, instrukce 2–4 se provedou ($r \leftarrow 0$, neboť $0 = 24 \bmod 12$; $m \leftarrow 12$; $n \leftarrow 0$).

Následuje třetí test podmínky na ř. 1 s $(m, n) = (12, 0)$, podmínka $n \neq 0$ není splněna, přejde se k instrukci na ř. 5 a výstupem je 12, což je skutečně $\gcd(84, 24)$.

Rozmyslete, co se stane, je-li na vstupu (m, n) a $m < n$.

Důkaz správnosti gcd-Euclid:
Uvědomme si, že algoritmus prochází dvojice
(m, n), $(n, m \bmod n)$, $(m \bmod n, n \bmod (m \bmod n))$, $\ldots$,
dokud není vytvořena dvojice $(p, 0)$. Číslo p je pak výstupem algoritmu.

Je třeba ukázat, že
1. Každé dvě po sobě následující dvojice v posloupnosti mají stejný gcd.
2. $\gcd(p, 0) = p$.
3. Algoritmus skončí, a to s dvojicí $(p, 0)$.

Ad 1: Zřejmě stačí ukázat, že pro každá m, n je $\gcd(m, n) = \gcd(n, m \bmod n)$.

Připomeňme: k dělí m, právě když existuje l tak, že $m = kl$. Dále $m \bmod n$ lze psát jako $m \bmod n = m - qn$ pro vhodné přirozené č. q. (zkuste na příkladech a pak zdůvodněte).

$\gcd(m, n) = \gcd(n, m \bmod n)$ dokážeme ověřením, že společný dělitel k čísel m a n je i společným dělitelem čísel n a $m \bmod n$ a naopak, tj. společný dělitel čísel n a $m \bmod n$ je i společným dělitelem čísel m a n. (uvědomte si, proč to stačí)

Je-li k společným dělitelem m a n, pak $m = kl_1$ a $n = kl_2$ pro nějaká l_1, l_2. Pak tedy $m \bmod n = m - qn = kl_1 - qkl_2 = k(l_1 - ql_2)$, tj. k je dělitelem čísla $m \bmod n$, a je tedy společným dělitelem čísel n a $m \bmod n$.

Je-li k společným dělitelem n a $m \bmod n$, pak $n = kl_1$ a $m \bmod n = m - qn = kl_2$, pro nějaká q, l_1, l_2. Pak tedy $m = qn + kl_2 = qkl_1 + kl_2 = k(ql_1 + l_2)$, tedy k dělí m a je společným dělitelem čísel m a n.

Ad 2: Plyne přímo z definice gcd.

Ad 3: Vždy je $m \bmod n < n$ (zdůvodněte). Když algoritmus začne s dvojicí (m, n), kde $m > n$, splňuje případná druhá dvojice $(n, m \bmod n)$ podmínku $n > m \bmod n$. První prvek druhé a každé další dvojice je tedy menší než první prvek předchozí dvojice a druhý prvek každé dvojice je menší než její první prvek. Je zřejmé, že taková posloupnost dvojic musí být konečná a její poslední dvojice má tvar $(k, 0)$.

Když začne s dvojicí (m, n), kde $m = n$, je další dvojicí $(n, 0)$ a skončí.

Když algoritmus začne s dvojicí (m, n), kde $m < n$, . . . (dokončete sami).

Důkaz je hotov.

Proč je gcd-Euclid lepší než gcd-naive?

Protože má menší časovou složitost.
Co to znamená? Uvidíme v dalším.

Zatím zkusme ručně spočítat gcd(84,24).

1. Jak jsme viděli, testuje gcd-Euclid(84, 24) dvojice (84, 24), (24, 12), (12, 0), pak skončí (zhruba řečeno po 3 krocích).

2. gcd-naive(84, 24) však musí snížit hodnotu t 12krát (zhruba řečeno tedy skončí až po 12 krocích).

gcd-Euclid rekurzivně a počet rekurzívních volání

Všimněme si, že gcd-Euclid lze vyjádřit rekurzívně takto:

gcd-Euclid-r(m, n)
1 **if** $n = 0$
2 **then return** m
3 **else** gcd-Euclid-r$(n, m \bmod n)$

Rekurzívně = algoritmus „volá sám sebe“

Trvání algoritmu je úměrné počtu rekurzívních volání. Úzká souvislost s tzv. Fibonacciho čísly F_i, která jsou definována rekurzívně takto:

$$F_0 = 0, \quad F_1 = 1, \quad F_i = F_{i-1} + F_{i-2} \text{ pro } i \geq 2.$$

tedy Fibonacciho čísla tvoří posloupnost

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Lamého věta Pokud $m > n \geq 1$ a $n < F_{k+1}$, pak gcd-Euclid-r(m, n) provede méně než k rekurzívních volání (zdůvodníme později).

Složitost algoritmu

Popisuje efektivitu algoritmu z hlediska zdrojů, které potřebuje.

Časová složitost

- Popisuje, jak je algoritmus “rychlý”, tj. kolik času trvá výpočet podle algoritmu (od zahájení po ukončení).
- Tedy popisuje efektivitu algoritmu z hlediska času jakožto využívaného zdroje.
- Touto složitostí se budeme zejména zabývat.

Paměťová (prostorová) složitost

- Popisuje, kolik paměti počítače je třeba pro výpočet podle algoritmu.
- Tedy popisuje efektivitu algoritmu z hlediska paměti jakožto využívaného zdroje.
- Touto složitostí se budeme zabývat okrajově. Více bude v předmětu Vyčíslitelnost a složitost.

Časová složitost algoritmu

vyjadřuje závislost trvání výpočtu daného algoritmu na velikosti vstupních dat

je to tedy funkce (zobrazení), která velikosti vstupních dat (tj. číslu) přiřadí trvání výpočtu (tj. číslo), tedy funkce $T : \mathbb{N} \to \mathbb{N}$ s významem:

$$\begin{array}{ccc} n & \mapsto & T(n) \\ \vdots & & \vdots \\ \text{velikost vstupu} & & \text{trvání výpočtu} \end{array}$$

Jak ale chápat $T(n)$? Dvě základní možnosti:

– časová složitost v nejhorším případě:
  $T(n)$ je největší trvání výpočtu podle algoritmu pro vstupy délky n.
– časová složitost v průměrném případě:
  $T(n)$ je průměrné trvání výpočtu podle algoritmu pro vstupy délky n.

Přesně: problém P, jeho vstupy označujeme I, algoritmus A. Označme:

– $t_A(I)$ ... trvání výpočtu podle algoritmu A pro vstup I,
– $|I|$ ... velikost vstupu I.

Definice Časová složitost algoritmu A v nejhorším případě je funkce $T : \mathbb{N} \to \mathbb{N}$ definovaná takto:

$$T(n) = \max\{t_A(I) \mid I \text{ je vstup problému } P \text{ a } |I| = n\}.$$

Časová složitost algoritmu A v průměrném případě je funkce $T : \mathbb{N} \to \mathbb{N}$ definovaná takto:

$$T(n) = \frac{t_A(I_1) + \cdots + t_A(I_m)}{m},$$

kde $I_1, \ldots, I_m$ jsou všechny vstupy problému P, které mají délku n.

Co je $t_A(I)$, tedy trvání algoritmu A pro konkrétní vstup I?

Definice $t_A(I)$ je počet elementárních výpočetních kroků vykonaných od zahájení do skončení výpočtu algoritmem A pro vstup I

trvání výpočtu tedy měříme v počtech provedených výpočetních kroků (instrukcí), nikoli v počtech sekund (viz později).

elementárním výpočetním krokem je obvykle instrukce (instrukce pseudokódu nebo instrukce programovacího jazyka ve kterém je algoritmus zapsán)

trvání výpočtu měřené počtem výpočetních kroků (instrukcí) je výhodnější než trvání měřené skutečným časem výpočtu (není závislé na implementaci a zejm. na počítači)

Uvědomme si:

- Časová složitost není trvání výpočtu (je to funkce popisující závislost trvání na velikosti vstupu).
- Je třeba upřesnit, zda myslím časovou složitost v nejhorším případě nebo v průměrném případě, jinak to nemá smysl (výpočet pro jeden vstup dané velikosti může trvat jinou dobu než výpočet pro jiný vstup stejné velikosti).
- Měřit trvání v počtech instrukcí je výhodné (nezávisí na počítači a implementaci). Jistou nevýhodou je skutečnosti, že nedává zcela konkrétní představu o čase, který výpočet trvá.

Další poznámky:

- Pro zjednodušení někdy uvažujeme jen počet "důležitých" (pro algoritmus základních) instrukcí, tj. těch, které se při výpočtu vykonávají často. Jde typicky o instrukce vykonávané opakovaně, např. uvnitř cyklu.
- Co je velikost vstupu, tj. $|I|$?
  Např. počet cifer vstupních čísel u problému sčítání čísel v desítkové soustavě; počet měst u problému hledání nejkratší cesty mezi městy v síti měst; počet prvků pole, které je třeba setřídit (později detailně) atd.
  Obecně $|I|$ vhodným způsobem vyjadřuje "rozsáhlost" vstupu I. Může existovat více přirozených způsobů, jak měřit rozsáhlost vstupu (např. u problému hledání v síti měst to může být počet měst, nebo jinak: počet spojnic mezi městy). Záměr: chceme, aby $t_A(I)$ přirozeně záviselo na $|I|$ (např. $t_A(I) = 2 * |I| + 3$).
  Tedy velikost vstupu je funkce $|\ | : Vst \to \mathbb{N}$ (Vst je množina přípustných vstupů problému P).

Příklad: složitost algoritmu sčítání v des. soustavě

Scitani-Cisel-Desitkove($a[0..n-1], b[0..n-1]$)

$t \leftarrow 0$
for $i \leftarrow 0$ **to** $n-1$
  do $c[i] \leftarrow a[i] + b[i] + t \bmod 10$
    $t \leftarrow \lfloor (a[i] + b[i] + t)/10 \rfloor$
$c[n] \leftarrow t$

Připomeňme: vstupem I je dvojice čísel zapsaných v desítkové soustavě, zápis každého z nich obsahuje n pozic.

1. Co je velikost vstupu $|I|$, tj. $|a[0..n-1], b[0..n-1]|$?
Položme $|a[0..n-1], b[0..n-1]| = n$.
(Mohli bychom ale vzít i $|a[0..n-1], b[0..n-1]| = 2n$, zkuste také.)

2. Co je $t_A(I)$, tj. $t_A(a[0..n-1], b[0..n-1])$?
Předpokládejme, že za elementární instrukce považujeme instrukce našeho pseudokódu a že za trvání výpočtu považujeme počet vykonaných elementárních instrukcí. Během výpočtu pro vstup $a[0..n-1], b[0..n-1]$ se provede:

$1\times$ řádek 1: 1 instrukce přiřazení,
$n\times$ řádky 2–4: 1 instrukce na ř. 2 (přiřazení nové hodnoty proměnné i), 4 instrukce na ř. 3 (2x instrukce $+$, 1x mod, 1x přiřazení), 5 instrukcí na ř. 4 (2x instrukce $+$, 1x $/$, 1x $\lfloor\ \rfloor$, 1x přiřazení), celkem $10n$ instrukcí,
$1\times$ řádek 5: 1 instrukce přiřazení.

Celkem se tedy provede $10n+2$ instrukcí. Tedy
$t_A(I) = t_A(a[0..n-1], b[0..n-1]) = 10n+2$.

Je zřejmé, že $t_A(I) = 10n+2$ pro každý vstup I velikosti n.
(U jiných algoritmů ale může pro dva vstupy I_1, I_2 se stejnou velikostí být $t_A(I_1) \neq t_A(I_2)$. Pomyslete např. na gcd-Euclid.)

Časová složitost v nejhorším případě je tedy $T(n) = 10n+2$.
Časová složitost v průměrném případě je také $T(n) = 10n+2$.

Co se změní, pokud provedení ř. 3 budeme považovat za provedení 1 instrukce?
Co se změní, pokud i provedení ř. 4 budeme považovat za provedení 1 instrukce?

Poznámka: velikost čísla coby vstupu

U problému sčítání čísel v desítkové soustavě je vstup tvořen dvěma n-ticemi čísel. První je $a_{n-1}\dots a_0$ reprezentovaná v poli $a[0..n-1]$ (tj. $a[0]=a_0, \dots, a[n-1]=a_{n-1}$), druhá je $b_{n-1}\dots b_0$ reprezentovaná v poli $b[0..n-1]$ (tj. $b[0]=b_0, \dots, b[n-1]=b_{n-1}$). Vstupem tedy nejsou čísla $A = \sum_{i=0}^{n} -1 a_i * 10^i$ a $B = \sum_{i=0}^{n} -1 b_i * 10^i$ reprezentovaná čísly a_i a b_i. Proto jsme za velikost vstupu volili n (ale mohli bychom zvolit i $2n$).

Často se objevuje situace, kdy vstupem problému je přirozené číslo N. Přirozený způsob jak měřit velikost vstupu pak je:

$$\begin{aligned} |N| &= \text{počet bitů potřebných pro zápis čísla } N\text{, tedy} \\ |N| &= \text{počet míst zápisu čísla } N \text{ ve dvojkové soustavě, tedy} \\ |N| &= \lfloor log_2 N \rfloor + 1 \end{aligned}$$

Příklady: $|1| = 1$ (protože $(1)_2 = 1$), $|2| = 2$ ($(2)_2 = 10$), $|3| = 2$ ($(3)_2 = 11$), $|4| = 3$ ($(4)_2 = 100$), $|5| = 3$ ($(5)_2 = 101$), ...

Zdůvodnění:

Každé číslo N je některým z čísel 2^{k-1}, $2^{k-1}+1$, ..., 2^k-1 pro vhodné k.

To jsou právě čísla, jejichž binární zápis má k míst.

Právě pro tato čísla platí
$\lfloor \log_2 2^{k-1} \rfloor = k-1$, $\lfloor \log_2 2^{k-1}+1 \rfloor = k-1$, ..., $\lfloor \log_2 2^k - 1 \rfloor = k-1$.
(Pro $N < 2^{k-1}$ je $\lfloor \log_2 N \rfloor < k-1$, pro $N > 2^k - 1$ je $\lfloor \log_2 N \rfloor > k-1$.)

Tedy pro každé $M \in \{2^{k-1}, 2^{k-1}+1, \dots, 2^k-1\}$ platí $\lfloor \log_2 M \rfloor + 1 = k$.

Tedy speciálně i pro N platí

$$\lfloor \log_2 N \rfloor + 1 = k$$

Příklady k procvičení:
1. Jaký je počet míst zápisu čísla N v desítkové soustavě (obecněji, v soustavě o základu b)?
2. Označíme-li $|N|_b$ počet míst zápisu čísla N v soustavě o základu b, jaký je vztah mezi $|N|_{10}$ a $|N|_2$? (Použijte $\log_a N = \log_a b \cdot \log_b N$.)

Často se vyskytující složitosti

Analýza složitosti a odvození vztahu $T(n) = 10n + 2$ bylo v tomto případě velmi jednoduché. U jiných algoritmů to je obtížnější, ale v principu stejné.

Při analýze složitosti algoritmů se často vyskytují některé funkce. Jednou z nich je $10n + 2$. Dalšími jsou.

c ... konstanta
$\log_b n$ ... logaritmus o základu b (místo $\log_2 n$ píšeme také $\lg n$)
n ... lineární (obecněji $an + b$)
$n \log n$ ... logaritmicko lineární (log-lineární)
n^2 ... kvadratická
n^3 ... kubická (obecněji $an^3 + bn^2 + cn + d$ nebo polynomy vyšších řádů)
2^n ... exponenciální
$n!$ ... faktoriál

Proč jsou uvedeny v tomto pořadí? Prozkoumejte jejich chování (např. jak rychle rostou). Vrátíme se k tomu.

Složitost $T(n)$ jako nositel důležité informace

O čem nás informuje zpráva, že $T(n) = 10n + 2$, popř. $T(n) = n^2$, popř. $T(n) = 2^n$?

Stručně řečeno, algoritmus se složitostí $T(n) = 10n + 2$, popř. $T(n) = n^2$ je prakticky použitelný.
Algoritmus se složitostí $T(n) = 2^n$ je prakticky nepoužitelný.

Proveďme zatím jednoduchou úvahu.

Úvaha 1 Uvažujme dva různé algoritmy, A_1 a A_2, pro řešení problému (např. problém setřídit n-prvkovou posloupnost čísel). Předpokládejme, že jejich složtosti jsou
$T_1(n) = n^2$ a $T_2(n) = 20n \log_2 n$.

Algoritmy jsou vykonávány na počítačích C_1 a C_2. C_1 je rychlý, vykoná 10^{10} instrukcí/sekundu, C_2 jen 10^7 instrukcí/sekundu.

Chceme vyřešit pro vstup I velikosti 10,000,000 (10 milionů), tj. $|I| = 10^7$.
S pomocí A_1 na poč. C_1 tvá výpočet

$$\frac{(10^7)^2}{10^{10}} = 10^4 sec = 166\ min\ 40\ sec = 2\ hod\ 46\ min\ 40\ sec.$$

S pomocí A_2 na poč. C_2 tvá výpočet

$$\frac{20 \cdot 10^7 \cdot \log 10^7}{10^7} \approx 465\ sec = 7\ min\ 45\ sec.$$

S A_2 na 1000krát pomalejším počítači je výpočet cca 21.5 krát rychlejší.

Závěr: Složitost algoritmu je skutečně důležitá. Zvýšením rychlosti počítače ji neobejdeme.

Úvaha 2 Jak dlouho trvá výpočet? (Přibližné) hodnoty některých funkcí.

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	33	100	1000	1024	3628800
100	6.6	100	660	10^4	10^6	$1.27 \cdot 10^{30}$	$9.33 \cdot 10^{157}$
10^3	10	10^3	10^4	10^6	10^9	$1.07 \cdot 10^{301}$	$4.02 \cdot 10^{2567}$
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}	$1.99 \cdot 10^{3010}$	$2.85 \cdot 10^{35659}$
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2 \cdot 10^7$	10^{12}	10^{18}		

Probíhá-li výpočet na velmi rychlém počítači, který provádí 10^{15} operací za sekundu (rychlý asi jako nejrychlejší počítač v současné době), jsou doby výpočtů v sekundách následující (zaokrouhleno).

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	0	0	0	0	0	0	0
100	0	0	0	0	0	$1.27 \cdot 10^{15}$	$9.33 \cdot 10^{142}$
10^3	0	0	0	0	0	$1.07 \cdot 10^{286}$	$4.02 \cdot 10^{2552}$
10^4	0	0	0	0	0.001	$1.99 \cdot 10^{2995}$	$2.85 \cdot 10^{35644}$
10^5	0	0	0	10^{-5}	1		
10^6	0	0	0	0.001	1000		

Je to pro algoritmy se složitostí 2^n a $n!$ tak hrozné?

Jak dlouho je např. $2.85 \cdot 10^{35644}$ sec? (trvání pro $n = 10^4$ při složitosti $n!$)
Rok má průměrně 31,556,926 sekund. Je to tedy
$2.85 \cdot 10^{35644} / 31556926 \approx 9 \cdot 10^{35636}$ let!

Doba existence planety Země $4.54 \cdot 10^9$ let. Tedy výpočet je cca $2 \cdot 10^{35627}$krát delší než doba existence naší planety!

Výsledku výpočtu se nikdy nedočkáme!

Co první významné (nenulové) trvání pro složitosti 2^n a $n!$?

Pro $n = 100$ a $T(n) = 2^n$ je doba výpočtu $4 \cdot 10^7$ let, tj. 40 miliónů let (před 65 mil. lety vyhynuli dinosauři).

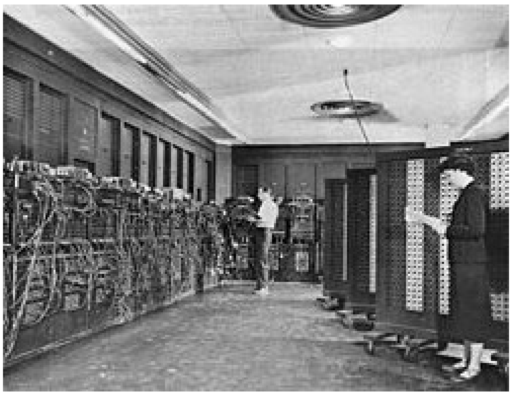
Ani v tomto případě se výsledku nikdy nedočkáme.

Tomuto jevu se říká "curse of exponential complexity" (prokletí exponenciální složitosti, R. Bellman použil podobný pojem "curse of dimensionality").

Počítače včera, dnes a zítra

VČERA – **první počítač**:
ENIAC (Electronic Numerical Integrator And Computer)

- 1946
- první obecně programovatelný počítač výpočetně ekvivalentní tzv. Turingovu stroji (obecně programovatelný)
- Univ. Pennsylvania, USA (US Army projekt, $6 mil. v dnešní hodnotě)
- 27 tun, plocha 63 m^2, příkon 150kW
- 380 operací násobení za sekundu



DNES - **nejrychlejší počítač na světě**
Jak se měří rychlost?

- v jednotkách FLOPS (FLoating point Operations Per Second, tj. počet operací s plovoucí řádovou čárkou za sekundu)
- měří se speciálním testem zahrnujícím výpočet tzv. LU rozkladu velké matice
- TF TFLOPS (teraFLOPS) = 10^{12} FLOPS, PFLOPS (petaFLOPS) = 10^{15} FLOPS, EFLOPS (exaFLOPS) = 10^{18} FLOPS
- od r. 1993 existuje seznam TOP500 (aktualizován 2x ročně, při Int. Supercomputing Conference a ACM/IEEE Supercomputing Conference)

nejrychlejší počítač (červen 2017):
Sunway TaihuLight, National Supercomputing Center, Wuxi (Čína), 93 PFLOPS

ZÍTRA – **Moorův zákon**

G. E. Moore, *1929, spoluzakladatel Intelu
zákon popisující trend ve vývoji hardware:
"počet součástí integrovaných obvodů se zdvojjnásobí každé dva roky";
platí přibližně i pro rychlost počítačů, paměť apod.

Tedy: Parametry hardware se zlepšují velmi rychle.

Pozor na předpovědi. Ukazují naše permanentní podceňování vývoje.

"It would appear that we have reached the limits of what it's possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in five years."
—John von Neumann, 1949

"I think there is a world market for maybe five computers."
—Thomas J Watson, President of IBM, 1943

"Computers in the future will weigh no more than 1.5 tons."
—Popular Mechanics, 1949

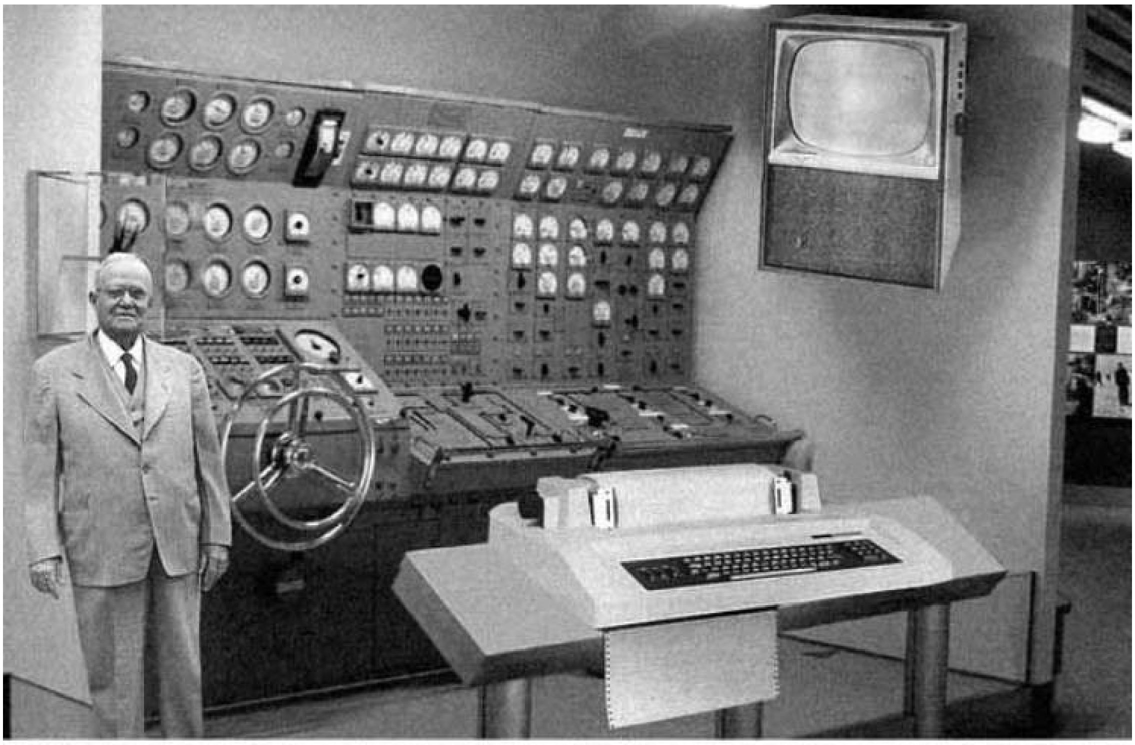
"I have travelled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year."
—Editor in charge of business books, Prentice Hall, 1957

"Transmission of documents via telephone wires is possible in principle, but the apparatus required is so expensive that it will never become a practical proposition."
—Dennis Gabor, 1962 (laureát Nobelovy ceny za holografii)

"There is no reason for any individual to have a computer in his home."
—Ken Olsen, co-founder of Digital Equipment Corporation, 1977

"640kB should be enough for anyone."
—Bill Gates, 1981

Jak by mohl vypadat domácí počítač v roce 2004 (představa v roce 1954)?



Scientists from the RAND Corporation have created this model to illustrate how a "home computer" could look like in the year 2004. However the needed technology will not be economically feasible for the average home. Also the scientists readily admit that the computer will require not yet invented technology to actually work, but 50 years from now scientific progress is expected to solve these problems. With teletype interface and the Fortran language, the computer will be easy to use.

Cvičení Údaje v řádcích udávají čas t, který máme k dispozici. Funkce $f(n)$ ve sloupcích udávají dobu výpočtu v milisekundách potřebnou pro zpracování vstupu o velikosti n. Do každého políčka (daného časem t a funkcí $f(n)$) doplňte údaj, který udává maximální velikost n vstupu, který je možné zpracovat v čase t při době výpočtu dané $f(n)$.
Např. pro t =1 sec a $f(n) = n$ je maximální velikost vstupu 1000, protože takový výpošet trvá $f(1000) = 1000$ msec $= 1$ sec.

čas	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1 sec		1000					
1 min							
1 hod							
1 den							
1 měs.							
1 rok							
1 století							

Úvaha 3 Pomůže technologický pokrok?
Jak se zvětší velikost vstupu, který jsme za danou dobu (kterou můžeme čekat) schopni algoritmem zpracovat, zvyšuje-li se rychlost počítačů každým rokem dvojnásobně?

Že se rychlost počítačů zvyšuje každým rokem dvojnásobně (to odpovídá Mooreovu zákonu), znamená, že Čas potřebný k provedení jedné instrukce v roce $r+1$ je 2krát menší než v roce r (jinými slovy, za danou časovou jednotku počítač vykoná v roce $r+1$ je 2krát více instrukcí než v roce r).

Označme
$t_{\max}$ ... doba, kterou můžeme čekat (např. 5 min, 2 hod, 10 ms)
$n_{\max}(r)$ ... max. velikost vstupu, který jsme schopni zpracovat (v roce r)
$f(n)$ ... časová složitost algoritmu (počet instrukcí).

Jak velký vstup jsme schopni zpracovat v roce $r+1$, tj. jaký je $n_{\max}(r)$?
Uvažme pro $f(n) = n$ (lineární) a $f(n) = 2^n$ (exponenciální).

Doba výpočtu pro vstup velikosti n je $f(n) \cdot c_r$, kde c_r je čas potřebný k provedení jedné instrukce v roce r.

$\mathbf{f(n) = n}$: $n_{\max}(r+1) = 2n_{\max}(r)$, tedy po roce budeme schopni zpracovat dvakrát větší vstup ("dvakrát více dat").

Proč? $n_{\max}(r)$ je největší n, pro které $n \cdot c_r \leq t_{\max}$. Hledáme $n_{\max}(r+1)$, tj. největší n, pro které $n \cdot c_{r+1} \leq t_{\max}$, tedy (protože $c_{r+1} = \frac{1}{2}c_r$) pro které $n \cdot \frac{1}{2}c_r \leq t_{\max}$. Je jasné, že tím je $2n_{\max}(r)$, tj. $n_{\max}(r+1) = 2n_{\max}(r)$.

Snadno také vidíme, že $n_{\max}(r+k) = 2^k n_{\max}(r)$, tj. po k letech se velikost zpracovatelného vstupu zvýší 2^kkrát.

$\mathbf{f(n) = 2^n}$: $n_{\max}(r+1) = n_{\max}(r) + 1$, tedy po roce budeme schopni zpracovat jen o jednotku velikosti větší vstup.

Proč? Stejnou úvahou dojdeme k tomu, že $n_{\max}(r+1)$ je největší n, pro které je $2^n \cdot \frac{1}{2}c_r \leq t_{\max}$, a tím je $n_{\max}(r) + 1$.

Snadno také vidíme, že $n_{\max}(r+k) = n_{\max}(r) + k$, tj. po k letech se velikost zpracovatelného vstupu zvýší o k, každý rok jsme schopni zpracovat data větší jen o jednu položku!

$\Rightarrow$ U algoritmů s exponenciální složitostí technologický pokrok nepomůže.

Cvičení (předchozí úvaha obecněji) Jak se zvětší velikost vstupu, který jsme za danou dobu $t_{\max}$ schopni algoritmem zpracovat, zvětší-li se rychlost počítačů za období od roku r do roku r' d-násobně?

V roce r je největší velikost zpracovatelného vstupu $n_{\max}(r)$, což je tedy největší n, pro které $f(n) \cdot c_r \leq t_{\max}$. Dále je $c_{r+1} = \frac{1}{d} \cdot c_r$.
$n_{\max}(r')$ je největší n, pro které $f(n) \cdot c_{r'} \leq t_{\max}$, tj. $f(n) \cdot \frac{1}{d} \cdot c_r \leq t_{\max}$.

Předpokládejme pro jednoduchost, že $f(n_{\max}(r)) \cdot c_r = t_{\max}$ a že f má inverzní funkci f^{-1}.
Pak $n_{\max}(r')$ je největší n, pro které $f(n) \leq d \cdot f(n_{\max}(r))$, tedy $n \leq f^{-1}(d \cdot f(n_{\max}(r)))$.

Pro $f(n) = 2^n$:
$n \leq \lg(d \cdot 2^{n_{\max}(r)}) = \lg d + n_{\max}(r)$, tj. $n_{\max}(r') = \lfloor \lg d + n_{\max}(r) \rfloor$.

Pro $f(n) = n^p$:
$n \leq \sqrt[p]{d \cdot n_{\max}(r)^p} = \sqrt[p]{d} \cdot n_{\max}(r)$, tj. $n_{\max}(r') = \lfloor \sqrt[p]{d} \cdot n_{\max}(r) \rfloor$.

Předpokládejme opět $f(n_{\max}(r)) \cdot c_r = t_{\max}$. Doplňte hodnoty $n_{\max}(r')$ do následující tabulky.

d	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1							
2							
4							
10							
100							
1000							
10^6							
10^p							
2^p							