Seminář z programování

Zdeněk Sawa

Katedra informatiky, FEI
Vysoká škola báňská – Technická universita Ostrava
17. listopadu 15, Ostrava-Poruba 708 33
Česká republika

16. září 2025

Vyučující

Jméno: Zdeněk Sawa

E-mail: zdenek.sawa@vsb.cz

Místnost: EA413

WWW: http://www.cs.vsb.cz/sawa/spr

Seminář:

- přednáška: úterý 9:00–10:30, místnost UA4
- cvičení: úterý 10:45–12:15, místnost UA4

Požadavky na zápočet

- V průběhu semestru budou na následující adrese zveřejňovány problémy, za jejichž (úspěšná) řešení budete dostávat body:

  http://www.cs.vsb.cz/sawa/spr/problems.html

- U každého problému bude uveden počet bodů, který za jeho vyřešení můžete získat, a termín, do kterého je možné řešení daného problému odevzdávat.
- Řešení posílejte e-mailem na adresu zdenek.sawa@vsb.cz.
- O vyřešených problémech stručně poreferujete vyučujícímu na konci semestru.
- Pro získání zápočtu je třeba získat alespoň 51 bodů.
- Body je možné též získat za problémy vyřešené v rámci soutěže v programování CTU Open 2025
  (20 bodů za každý problém vyřešený v rámci soutěže).

Obsah předmětu

Cílem je seznámit se podrobněji s problematikou návrhu a implementace algoritmů.

Konkrétně:

- základní obecné metody používané při tvorbě algoritmů (rekurzivní algoritmy, dynamické programování, greedy algoritmy)
- některé grafové algoritmy
- algoritmy pro práci s (velkými) čísly
- řešení některých kombinatorických problémů
- řešení některých problémů z oblasti výpočetní geometrie
- ...

Algoritmy a problémy

Algoritmy slouží k řešení **problémů**.

Při formulaci problému by mělo být uvedeno:

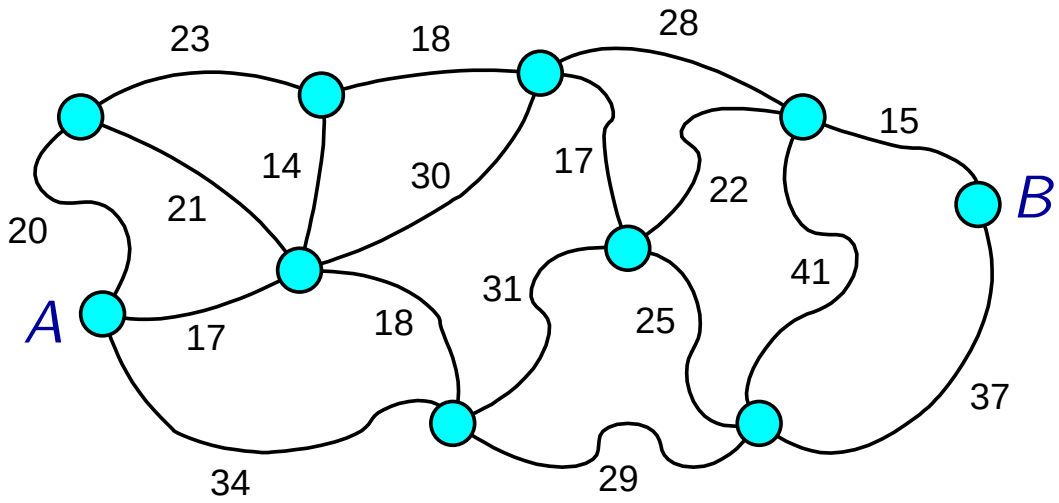
- Co je **vstupem**.
- Co je **výstupem**.
- Jak výstup závisí na vstupu.

Poznámka: Konkrétnímu vstupu problému se říká **instance problému**.

Příklad problému

Vstup: Seznam měst a silnic spojujících tato města.
U každé silnice je uvedeno, z kterého města do kterého vede, a jaká je její délka (v km).
Dále pak dvě města ze seznamu měst – označme je město A a město B.

Výstup: Nejkratší cesta z města A do města B.



Proč též vznikl tento předmět

- Organizace **ACM (Association for Computing Machinery)** pořádá už od roku 1977 každoročně soutěž v programování **ACM International Collegiate Programming Contest (ICPC)**.
- Soutěže se účastní tříčlenné týmy studentů z univerzit z celého světa.
- Celosvětovému finále předchází regionální kola (např. středoevropské regionální kolo), kterým mohou předcházet národní předkola.
- Několik let je pořádáno česko-slovenské předkolo pod názvem CTU Open. Bývá pořádáno distribuovaně (Praha – Brno – Ostrava – Plzeň – Bratislava – Žilina – Banská Bystrica – Košice).
- Jedna z motivací pro vznik tohoto předmětu: Připravit naše studenty na řešení úloh typu, jaké se objevují v této soutěži.

Server Online Judge

- Při soutěžích v programování ACM bývají řešení vyhodnocována automaticky – systém pro vyhodnocování pošle jako vstup programu testovací data a vyhodnotí jeho výstup.
- Podobné systémy jako se používají při těchto soutěžích jsou k dispozici i na Internetu. Asi největší a nejznámější z nich je na adrese

  https://onlinejudge.org/

- Na tomto serveru je k dispozici několik tisíc problémů.
- V rámci SPR budou zadávány na adrese

  http://www.cs.vsb.cz/sawa/spr/problems.html

  problémy z výše uvedeného serveru a za úspěšné řešení problému bude považován váš program, který mi pošlete do stanoveného termínu pro daný problém, a který bude přijat tímto serverem.

Vyhodnocování problémů na serveru Online Judge

- Pro odesílání problémů na server je třeba se zaregistrovat.
- Problémy jsou číslovány.
- K odesílání řešení slouží webový formulář, na který se dostanete přes tlačítko „Submit“ u zadání daného problému.
- Systému se posílá zdrojový kód, nikoliv přeložený binární soubor.
- Celý program musí být v jediném souboru.
- Je možné použít následující programovací jazyky:
  - C
  - C++
  - C++11
  - Java
  - Pascal
  - Python

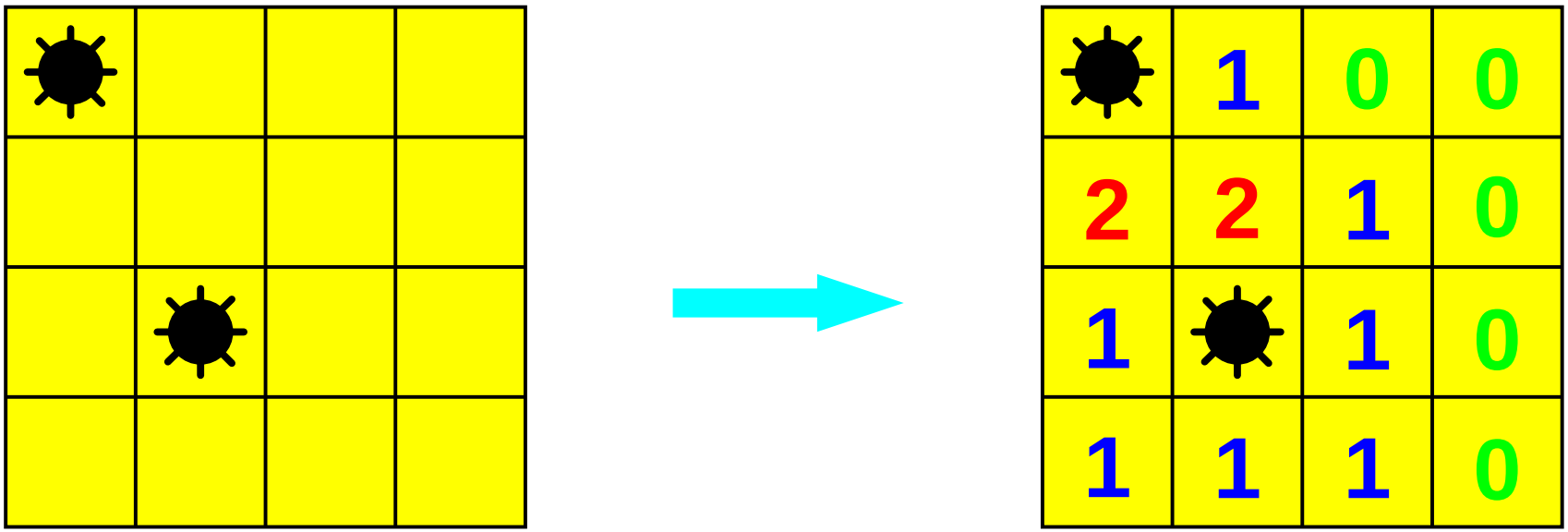
Vyhodnocování problémů na serveru Online Judge

- Všechna zadání jsou toho typu, že:
  - program čte data ze standardního vstupu
  - program zapisuje data na standardní výstup
- Součástí zadání je přesný popis toho, jak bude vstup a výstup programu vypadat.
- Ve všech případech je vstup i výstup programu čistě textový.
- Na serveru se testuje pouze to, zda pro dané vstupy dává váš program správný výstup.
- Součástí zadání bývá vždy ukázkový vstup a odpovídající ukázkový výstup.

  Poznámka: To, že vám váš program funguje na ukázkovém vstupu v zadání ještě neznamená, že je vaše řešení správné a že bude serverem přijato.

Příklad problému: Minesweeper (10189)

Úkol: zjistit pro každé prázdné políčko počet sousedních min



Příklad vstupu a výstupu

Příklad vstupu

```
4 4
*...
....
.*..
....
3 5
**...
.....
.*...
0 0
```

Příklad výstupu

```
Field #1:
*100
2210
1*10
1110

Field #2:
**100
33200
1*100
```

Vyhodnocování problémů na serveru Online Judge

Průběh vyhodnocování na serveru Online Judge:

- Pomocí webového formuláře odešlete váš program na server.
- Program je přeložen.
- Program je spuštěn a na jeho standardní vstup jsou poslána testovací data.
- Pokud program neskončí ve stanoveném časovém limitu, je násilně ukončen.
- Pokud program úspěšně skončí, je výstup, který program vyprodukoval, porovnán s požadovaným výstupem (pozn. někdy je k tomuto účelu vytvořen speciální program, pokud může existovat více různých správných výstupů).
- Výsledek (tj. zda bylo vaše řešení přijato nebo ne) zjistíte na stránce přístupné z menu pomocí položky „My submissions“, na které se zobrazují odpovědi serveru.

Vyhodnocování problémů na serveru Online Judge

Možné odpovědi serveru (Online Judge):

- **Accepted** – problém byl úspěšně vyřešen
- **Compile Error** – program se nepodařilo přeložit
- **Restricted Function** – program používá nějakou nepovolenou funkci
- **Runtime Error** – program skončil chybou za běhu (např. Segmentation Fault)
- **Time Limit Exceeded** – program běžel příliš dlouho a byl násilně ukončen
- **Wrong Answer** – program vydal chybný výstup
- **Presentation Error** – výstup vypadá správně, ale neodpovídá plně požadovanému formátu výstupu

Vyhodnocování problémů na serveru Online Judge

Pro programy posílané na server platí určitá omezení ohledně toho, jaké typy operací program může provádět.

Program nesmí:

- pracovat se soubory (kromě standardního vstupu a výstupu)
- komunikovat po síti
- spouštět jiné procesy
- komunikovat s jinými procesy
- používat jakákoliv jiná volání operačního systému

Naopak následující činnosti program provádět může:

- číst ze standardního vstupu a zapisovat na standardní výstup
- alokovat paměť (maximální množství paměti, které má program k dispozici, je však omezeno (řádově 10–20 MB))
- používat funkce ze standardních knihoven (matematické funkce, práce s řetězci, datové struktury, . . .)

Vyhodnocování problémů na serveru Online Judge

Několik poznámek:

- V zadání je přesně popsán formát vstupu. Program **nemusí** řešit situace, kdy data na vstupu neodpovídají uvedenému formátu.
- Program by neměl o vstupních datech předpokládat nic, co není uvedeno v zadání.
- V zadání je přesně popsán formát výstupu. Výstup musí **přesně** odpovídat tomuto formátu.
- Systém neposkytuje žádné informace o konkrétních testovacích datech. Není možné získat data, která způsobují, že dostávám odpověď **Wrong Answer**.
- Počet pokusů o vyřešení problému (tj. počet odeslání programu na server) není omezen.

Zadání je třeba číst pozorně!

Vyhodnocování problémů na serveru Online Judge

Ještě několik dalších poznámek:

- Vstupní data bývají toho typu, že umožňují otestovat program pro mnoho různých instancí problému. Zadání specifikuje, jak jsou jednotlivé instance od sebe ve vstupu odděleny (např. prázdným řádkem).
- Testovací data bývají značně velká (řádově i MB dat).
- Je třeba dávat pozor na okrajové případy, které jsou podle zadání přípustné.

Vyhodnocování problémů na serveru Online Judge

Několik poznámek k programům v Javě:

- Program se posílá jako jediný soubor.
- Program může obsahovat více tříd, žádná z nich však nesmí být `public`.
- Na začátku souboru nesmí být uveden příkaz `package`, tj. všechny třídy definované v programu patří do defaultního bezejmenného package.
- Program musí obsahovat třídu nazvanou `Main`.
- Třída `Main` musí obsahovat statickou metodu

  ```
  public static void main(String[] args)
  ```

  jejímž zavoláním bude program spuštěn.

Posílání řešení problémů

- Řešení posílejte e-mailem na adresu `zdenek.sawa@vsb.cz`.
- Zdrojový kód přikládejte k mailu jako přílohu (attachment).
- Soubor se zdrojovým kódem pojmenujte $\langle cislo\_problemu \rangle . \langle pripona \rangle$, kde:
  - $\langle cislo\_problemu \rangle$ – číslo problému,
  - $\langle pripona \rangle$ –přípona určující použitý programovací jazyk (`.c`, `.cpp`, `.c11`, `.java`, `.pas`, `.py`)

  Například: `10189.c`, `10189.cpp`, `10189.c11`, `10189.java`, `10189.pas`, `10189.py`
- Do textu mailu napište Vaše jméno, příjmení, login a čísla a názvy problémů, jejichž řešení posíláte.
- Nebalte posílané soubory do archivu ani je nekomprimujte.
- Kromě zdrojových kódů **neposílejte** žádné další soubory (např. přeložené programy).

Bodování

- Body uvedené u zadaných problémů jsou maximem bodů, které za vyřešení daného problému můžete získat.
- Definitivní počet získaných bodů bude určen až po referování.
- Strhávání bodů při zjištění, že řešení bylo zkopírováno:
  - první případ: **–20 bodů**
  - druhý případ: **–50 bodů**
  - třetí případ: neudělení zápočtu

Poznámka: Za zkopírované je považováno i řešení vzniklé úpravami nějakého existujícího řešení (přejmenování identifikátorů, změna formátování, přeházení funkcí nebo metod ve zdrojovém kódu, apod.).

Soutěž v programování
CTU Open Contest 2025

7. a 8. listopadu 2025

Info pro soutěžící z VŠB-TU Ostrava:
http://acm.vsb.cz

Oficiální stránky soutěže, registrační formulář:
http://contest.felk.cvut.cz/25prg

Soutěž CTU Open

- Soutěží tříčlenné týmy, které řeší zadané úlohy (bývá řádově 8–12 úloh).
- Na řešení úloh je omezený čas (5 hodin).
- Bude probíhat distribuovaně (současně v Praze, Brně, Ostravě, Plzni, Bratislavě, Žilině, Banské Bystrici a Košicích).
- Nejlepší tým (resp. týmy) z každé školy postoupí do regionálního středoevropského kola, které se bude konat ve Wroclawi v Polsku ve dnech 5.–7. prosince 2025 .

Poznámky k implementaci

Standardní vstup a výstup

V běžně používaných operačních systémech (MS Windows, všechny možné odnože Unixu) je možné v příkazové řádce při spouštění **přesměrovat** standardní vstup a výstup.

- Standardní vstup je možné číst ze souboru místo z klávesnice:

```
./program < input.txt
```

```
program.exe < input.txt
```

- Standardní výstup je možné zapisovat do souboru místo na obrazovku:

```
./program > output.txt
```

- Obojí je možné zároveň:

```
./program < input.txt > output.txt
```

Čtení ze standardního vstupu v C

- Čtení po jednotlivých znacích (bytech):

```
int getchar(void);
```

- Naposledy přečtený znak lze vrátit zpět:

```
int ungetc(int c, FILE *stream);
```

Použití:

```
ungetc(c, stdin );
```

Čtení ze standardního vstupu v C

Čtení po řádcích:

```
char* gets(char *s);
```

Silně se nedoporučuje používat tuto funkci!

Lepší varianta:

```
char* fgets(char *s, int size, FILE *stream);
```

Použití:

```
#define BUF_SIZE 1024
char buffer[BUF_SIZE];
char *s = fgets(buffer, BUF_SIZE, stdin);
```

Čtení ze standardního vstupu v C

Formátovaný vstup:

```
int scanf(const char *format, ...);
```

Příklad použití:

```
int m, n;
while (scanf("%d %d", &m, &n) == 2) {
    ...
```

Funkce scanf() vrací počet úspěšně přečtených parametrů. Při chybě vrací EOF.

Čtení ze standardního vstupu v C

Čtení po blocích:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Použití:

```
#define BUF_SIZE 1024
char buffer[BUF_SIZE];
size_t ret = fread(buffer, 1, BUF_SIZE, stdin);
```

Zápis na standardní výstup v C

Jeden znak:

```
int putchar(int c);
```

Jeden řádek (automaticky přidává na konec '\n'):

```
int puts(const char *s);
```

Formátovaný výstup:

```
int printf(const char *format, ...);
```

Po blocích:

```
size_t fwrite(const void *p, size_t sz, size_t nmemb, FILE *stream);
```

Funkce qsort

Deklarace

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
           int(*compare)(const void *, const void *));
```

Návratová hodnota funkce compare:

- < 0 – první argument je menší než druhý
- $= 0$ – první a druhý argument se rovnají
- > 0 – první argument je větší než druhý

Funkce qsort

Použití

```
int a[LEN];
   ...
int compare(const void *xv, const void *yv) {
    const int *x = (int *)xv;
    const int *y = (int *)yv;
    if (*x < *y) return -1;
    else if (*x > *y) return 1;
    return 0;
}
   ...
qsort(a, n, sizeof(int), compare);
```

Funkce qsort

Použití

```
int a[LEN];
   . . .
int compare(const void *x, const void *y) {
     return *(const int *)x - *(const int *)y;
}
   . . .
qsort(a, n, sizeof(int), compare);
```

Pozn.: Pro porovnávání řetězců se hodí funkce strcmp.

ASCII znaky

- V C, C++ ani v Javě se nedělá rozdíl mezi znakem a jeho ASCII (resp. Unicode) hodnotou. Se znaky je možné dělat stejné operace jako s čísly.

```
for (int i = 'A'; i <= 'Z'; i++) {
    a[i] = . . .
}
```

```
for (int i = 65; i <= 90; i++) {
    a[i] = . . .
}
```

ASCII tabulka

0	NUL	16	DLE	32		48	0	64	@	80	P	96	‘	112	p
1	SOH	17	DC1	33	!	49	1	65	A	81	Q	97	a	113	q
2	STX	18	DC2	34	"	50	2	66	B	82	R	98	b	114	r
3	ETX	19	DC3	35	#	51	3	67	C	83	S	99	c	115	s
4	EOT	20	DC4	36	$	52	4	68	D	84	T	100	d	116	t
5	ENQ	21	NAK	37	%	53	5	69	E	85	U	101	e	117	u
6	ACK	22	SYN	38	&	54	6	70	F	86	V	102	f	118	v
7	BEL	23	ETB	39	’	55	7	71	G	87	W	103	g	119	w
8	BS	24	CAN	40	(	56	8	72	H	88	X	104	h	120	x
9	HT	25	EM	41	)	57	9	73	I	89	Y	105	i	121	y
10	LF	26	SUB	42	*	58	:	74	J	90	Z	106	j	122	z
11	VT	27	ESC	43	+	59	;	75	K	91	[	107	k	123	{
12	FF	28	FS	44	,	60	<	76	L	92	\	108	l	124	|
13	CR	29	GS	45	-	61	=	77	M	93	]	109	m	125	}
14	SO	30	RS	46	.	62	>	78	N	94	^	110	n	126	~
15	SI	31	US	47	/	63	?	79	O	95	_	111	o	127	DEL

Příklady využití vlastností ASCII tabulky

- Převod číslice na odpovídající hodnotu:

```
char c; // c obsahuje znaky '0', '1', '2', ..., '9'
int x = c - '0';
```

- Převod hodnoty x v intervalu 0–9 na znak:

```
char c = x + '0';
```

- Převod písmen 'A', 'B', ..., 'Z' na čísla 0, 1, ... , 25:

```
int x = c - 'A';
```

- Zjištění, zda proměnná c obsahuje malé písmeno a pokud ano, tak jeho převedení na velké písmeno:

```
if (c >= 'a' && c <= 'z') {
    c = c - 'a' + 'A';
}
```

Příklady využití vlastností ASCII tabulky

- Převod hexadecimální číslice na odpovídající hodnotu:

```
int hex2num(char c) {
    if (c >= '0' && c <= '9') return c - '0';
    if (c >= 'A' && c <= 'F') return c - 'A' + 10;
    if (c >= 'a' && c <= 'f') return c - 'a' + 10;
    return -1;
}
  ...
int x = hex2num(c);
```

Příklady využití vlastností ASCII tabulky

Varianta 2: připravit si tabulku

```
int hex_table[256];
int init() {
    for (int i = 0; i < 256; i++) {
        if (i >= '0' && i <= '9') hex_table[i] = i - '0';
        else if (i >= 'A' && i <= 'F') hex_table[i] = i - 'A' + 10;
        else if (i >= 'a' && i <= 'f') hex_table[i] = i - 'a' + 10;
        else hex_table[i] = -1;
    }
}
  ...
int x = hex_table[c];
```

Poznámka: V tomto případě tabulka příliš velký efekt nemá. Má smysl například u Unicodu při převodech mezi malými a velkými písmeny apod.

Funkce strlen v C

Je něco divného na následující konstrukci?

```
char *s;
    ...
for (int i = 0; i < strlen(s); i++) {
      // něco udělej s s[i]
    ...
}
```

Funkce strlen v C

Je něco divného na následující konstrukci?

```
char *s;
    ...
for (int i = 0; i < strlen(s); i++) {
      // něco udělej s s[i]
    ...
}
```

Funkce strlen vyžaduje čas úměrný délce řetězce!
Pokud n je délka řetězce, pak má výše uvedená smyčka časovou složitost $O(n^2)$, nikoliv $O(n)$.

Funkce strlen v C

Jednoduché řešení:

```
char *s;
    ...
int l = strlen(s);
for (int i = 0; i < l; i++) {
        // něco udělej s s[i]
      ...
}
```

Použití třídy String v Javě

Je něco špatně na následujícím cyklu?

```
String[] a;
    . . .
String s = "";
for (int i = 0; i < a.length; i++) {
     s += a[i];
}
return s;
```

Použití třídy String v Javě

Je něco špatně na následujícím cyklu?

```
String[] a;
    . . .
String s = "";
for (int i = 0; i < a.length; i++) {
    s += a[i];
}
return s;
```

Pokud pole a má délku n, zbytečně se vytváří n instancí třídy String. Pokun m je součet délek všech řetězců v poli a, kopírování obsahu mezi instancemi třídy String může vyžadovat až čas $O(m \cdot n)$.

Použití třídy String v Javě

Lepší řešení:

```
String[] a;
    . . .
StringBuilder s = new StringBuilder();
for (int i = 0; i < a.length; i++) {
    s.append(a[i]);
}
return s.toString();
```

Celkový čas je v tomto případě $O(m)$, kde m je součet délek všech řetězců v poli a.

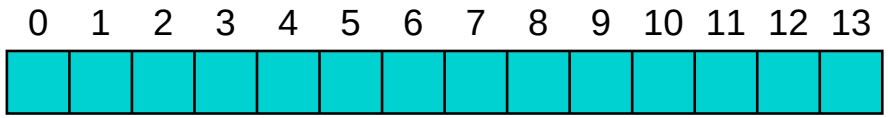
Datové struktury

Připomeňme si nejdůležitější **datové struktury** a speciálně pak časovou složitost jednotlivých operací prováděných na těchto datových strukturách (vkládání, odstraňování, vyhledávání prvků):

Datové struktury

Připomeňme si nejdůležitější **datové struktury** a speciálně pak časovou složitost jednotlivých operací prováděných na těchto datových strukturách (vkládání, odstraňování, vyhledávání prvků):

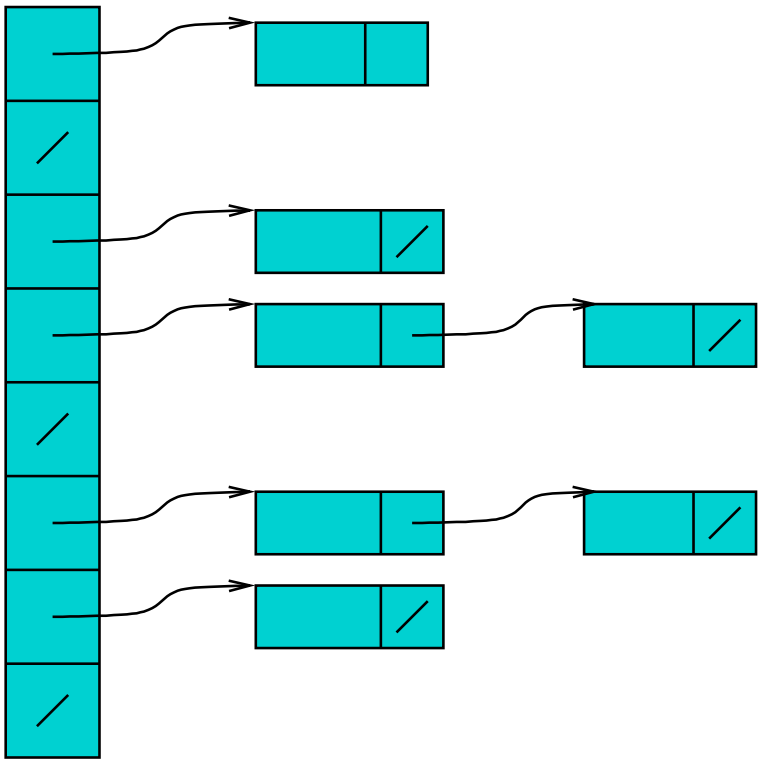
Pole:



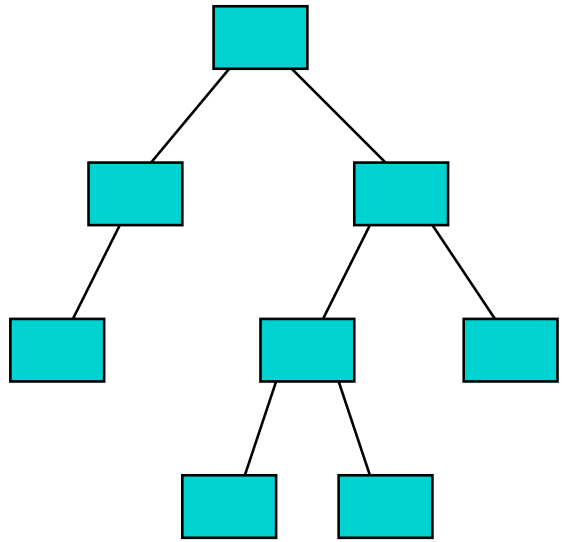
Seznam:

Datové struktury

Hashovací tabulka:



Strom:



Abstraktní datové typy

Datové struktury slouží k implementaci **abstraktních datových typů** (ADT), jako jsou například:

Abstraktní datové typy

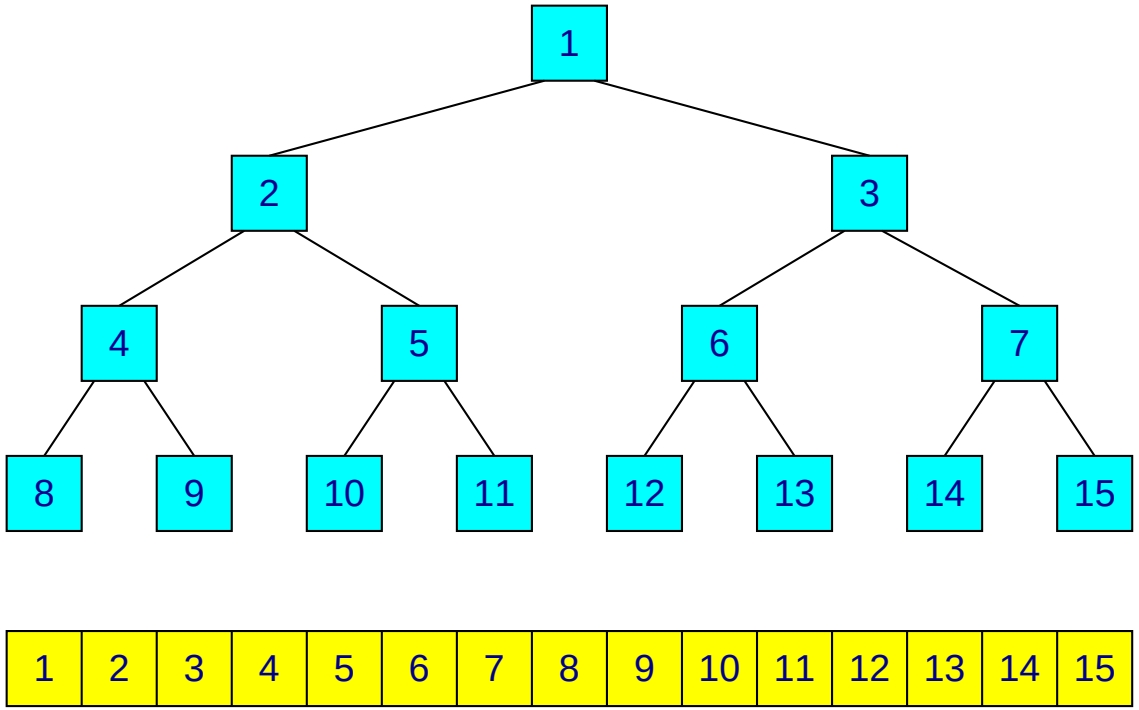
Datové struktury slouží k implementaci **abstraktních datových typů** (ADT), jako jsou například:

- množina (set)
- sekvence (list, array, vector)
- asociativní pole (associative array, map, mapping, hash, dictionary)
- zásobník (stack)
- fronta (queue, deque)
- prioritní fronta (priority queue)

ADT je dán svým **rozhraním** (tj. jaké operace s ním můžeme provádět). Jeden a tentýž ADT může být implementován pomocí různých datových struktur.

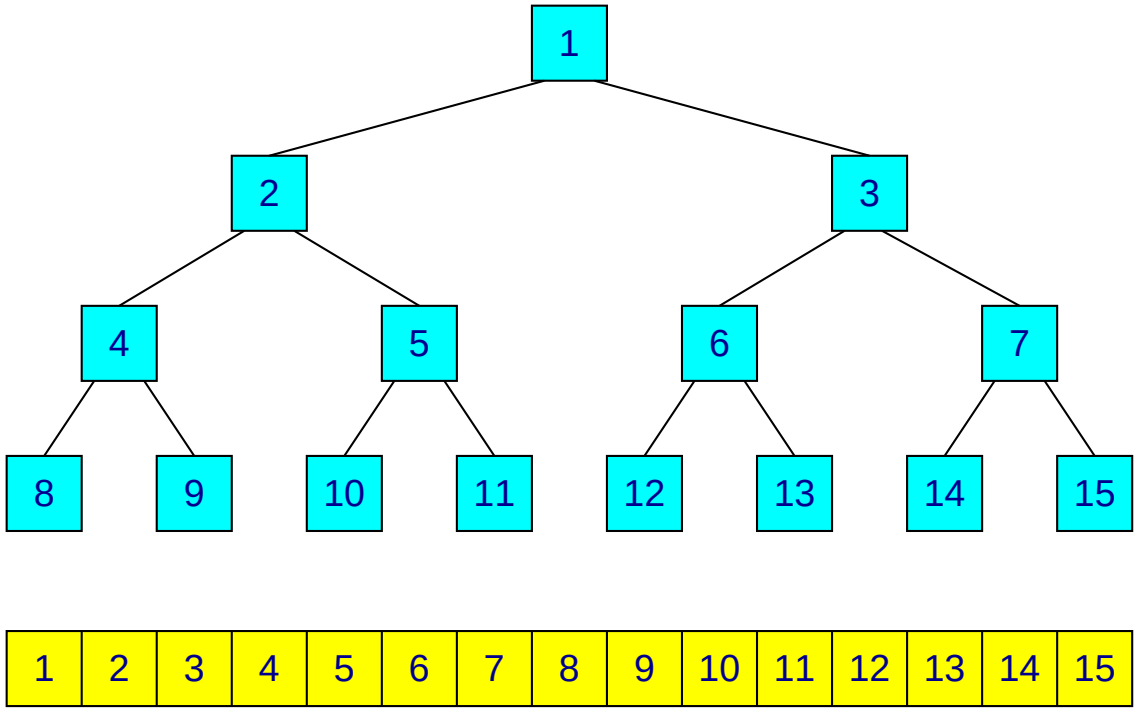
Úplný binární strom

Efektivní uložení úplného binárního stromu v paměti:



Úplný binární strom

Efektivní uložení úplného binárního stromu v paměti:



Potomci vrcholu s indexem i mají indexy $2i$ a $2i + 1$.
Rodič vrcholu s indexem i má index $\lfloor i/2 \rfloor$.

Bitová pole

Místo booleovského pole délky n jako například

```
bool b[1000];
```

je někdy vhodnější použít **bitové pole** délky $\lceil n/k \rceil$ tvořené hodnotami typu **int**, kde k je počet bitů v čísle typu **int**:

```
int a[32];
```

Bitová pole

Místo booleovského pole délky n jako například

```
bool b[1000];
```

je někdy vhodnější použít **bitové pole** délky $\lceil n/k \rceil$ tvořené hodnotami typu **int**, kde k je počet bitů v čísle typu **int**:

```
int a[32];
```

Poznámka: Hodnotu $\lceil n/k \rceil$ lze spočítat jako $(n + k - 1)/k$, například $\lceil n/32 \rceil$ lze spočítat jako $(n + 31)/32$.

Bitová pole

- Přečtení hodnoty prvku $b[i]$:

Bitová pole

- Přečtení hodnoty prvku $b[i]$:

  a[i / 32] & (1 << (i % 32))

  nebo trochu efektivněji

  a[i >> 5] & (1 << (i & 0x1F))

- Nastavení prvku $b[i]$ na 1:

Bitová pole

- Přečtení hodnoty prvku $b[i]$:

  a[i / 32] & (1 << (i % 32))

  nebo trochu efektivněji

  a[i >> 5] & (1 << (i & 0x1F))

- Nastavení prvku $b[i]$ na 1:

  a[i >> 5] |= (1 << (i & 0x1F))

- Nastavení prvku $b[i]$ na 0:

Bitová pole

- Přečtení hodnoty prvku $b[i]$:

  a[i / 32] & (1 << (i % 32))

  nebo trochu efektivněji

  a[i >> 5] & (1 << (i & 0x1F))

- Nastavení prvku $b[i]$ na 1:

  a[i >> 5] |= (1 << (i & 0x1F))

- Nastavení prvku $b[i]$ na 0:

  a[i >> 5] &= ~(1 << (i & 0x1F))

Reprezentace čísel v počítači

Typické velikosti celočíselných datových typů v jazycích C a C++ na 32-bitových systémech:

Typ	Min	Max	Bitů
signed char	−128	127	8
unsigned char	0	255	8
short	−32768	32767	16
unsigned short	0	65535	16
int	−2147483648	2147483647	32
unsigned int	0	4294967295	32
long	−2147483648	2147483647	32
unsigned long	0	4294967295	32
long long	−9223372036854775808	9223372036854775807	64
unsigned long long	0	18446744073709551615	64

Datový typ long long

Datový typ **long long** reprezentující 64-bitový integer je součástí ISO C99, ale ne ISO C89 (ANSI C) ani C++, ale překladače ho často podporují.

Příklad použití

```
#include <stdio.h>

int main()
{
    long long a, b, c;
    scanf("%lld %lld", &a, &b);
    c = a + b * 16LL;
    printf("%lld\n", c);
    return 0;
}
```

Práce s velkými čísly

Pokud ani 64-bitový integer nepostačuje, musíme si jednotlivé operace pro práci s čísly implentovat sami nebo použít vhodnou knihovnu.

Jednotlivé číslice čísla uložíme do pole.

Například číslo 856238190829347293482738 4 při použití základu 10:

24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8	5	6	2	3	8	1	9	0	8	2	9	3	4	7	2	9	3	4	8	2	7	3	8	4

Nebo při použití základu 10000:

6	5	4	3	2	1	0
8	5623	8190	8293	4729	3482	7384

Práce s velkými čísly

Struktura reprezentující velké číslo:

```
#define NUM_LEN 1000
#define NUM_BASE 10000

struct Number {
    int len;
    short a[NUM_LEN];
};
```

Práce s velkými čísly

Inicializace struktury (malým) celým číslem:

```
void num_init(Number *a, int x)
{
    int len = 0;
    while (x > 0) {
        a->a[len++] = x % NUM_BASE;
        x /= NUM_BASE;
    }
    a->len = len;
}
```

Práce s velkými čísly

Součet dvou čísel

```
void num_add(Number *src1, Number *src2, Number *dest) {
    int l1 = src1->len;
    int l2 = src2->len;
    int l;
    short *pa = src1->a; short *pb = src2->a;
    short *pc = dest->a;
    if (l1 < l2) {
        l = l2;
        memset(pa+l1, 0, (l-l1)*sizeof(short));
    } else {
        l = l1;
        memset(pb+l2, 0, (l-l2)*sizeof(short));
    }
        ...
```

Práce s velkými čísly

```
        ...
    int carry = 0;
    for (int i = 0; i < l; i++) {
        carry += *pa++ + *pb++;
        *pc++ = carry % NUM_BASE;
        carry /= NUM_BASE;
    }
    if (carry > 0) {
        if (l >= NUM_LEN) overflow();
        *pc = carry;
        l++;
    }
    dest->len = l;
}
```

Práce s velkými čísly

Pokud m a n jsou počty číslic čísel se kterými pracujeme, pak:

- **Součet** a **rozdíl** mají složitost $O(m+n)$
  (velikost výsledku je nanejvýš $\max\{m,n\}+1$ číslic).
- **Součin** má složitost $O(mn)$
  (velikost výsledku je nanejvýš $m+n$ číslic).
- **Podíl** (a výpočet **zbytku** po dělení) mají složitost $O(mn)$.

Poznámka: Například pro násobení a dělení existují i algoritmy, které mají asymptotickou složitost menší než $O(mn)$. Pro praktické účely je však většinou jednoduchý algoritmus se složitostí $O(mn)$ nejrychlejší.

Poznámka: Pokud v některých případech bude vždy jeden z operandů „malé“ číslo (vejde se do jedné číslice), může se vyplatit implementovat speciální jednodušší funkci pro tento případ místo použití obecného algoritmu.

Výpočet odmocniny

Následující funkce ilustruje, jakým způsobem se dá vypočítat pro dané celé kladné číslo x hodnota $\lfloor \sqrt{x} \rfloor$. Složitost algoritmu je $O(n^2)$, kde n je počet bitů čísla x.

SQRT(x)

```
1   n ← počet bitů čísla x        ▷ n musí být sudé
2   y ← 0
3   b ← 1 ≪ (n − 2)
4   while b > 0
5         do if x ≥ y + b
6               then x ← x − (y + b)
7                    y ← (y ≫ 1) | b
8               else  y ← y ≫ 1
9            b ← b ≫ 2
10  return y
```

Knihovny pro práci s velkými čísly

V C a C++ například knihovna GMP (GNU MP Bignum Library), která podporuje:

- práci s libovolně velkými celými čísly
- práci s libovolně velkými racionálními čísly (zlomky)
- práci s floting point čísly s libovolnou přesností

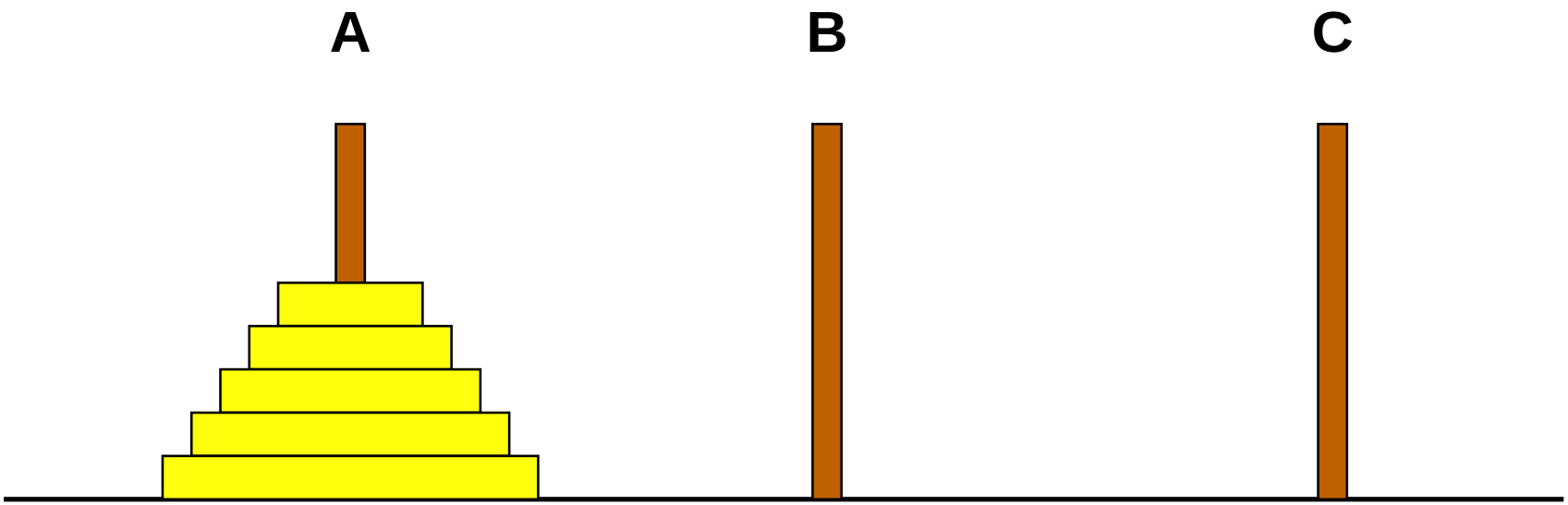
Poznámka: Knihovna GMP není součástí standardních knihoven jazyka C.

V Javě jsou součástí standardních knihoven třídy:

- java.math.BigDecimal – decimálně reprezentovaná desetinná čísla
- java.math.BigInteger – binárně reprezentovaná libovolně velká celá čísla

Rekurzivní algoritmy

Hanojské věže



Úkol: Přemístit disky z A na B, přičemž:

- V jednom okamžiku je možné přesouvat jen jeden disk.
- Není dovoleno položit větší disk na menší.

Hanojské věže

$n = 1$:

$A \to B$

Hanojské věže

$n = 1$:

$$A \to B$$

$n = 2$:

$$A \to C$$
$$A \to B$$
$$C \to B$$

Hanojské věže

$n = 1$:

$A \to B$

$n = 2$:

$A \to C$
$A \to B$
$C \to B$

$n = 3$:

$A \to B$
$A \to C$
$B \to C$
$A \to B$
$C \to A$
$C \to B$
$A \to B$

Hanojské věže

$n = 1$:

$A \to B$

$n = 2$:

$A \to C$
$A \to B$
$C \to B$

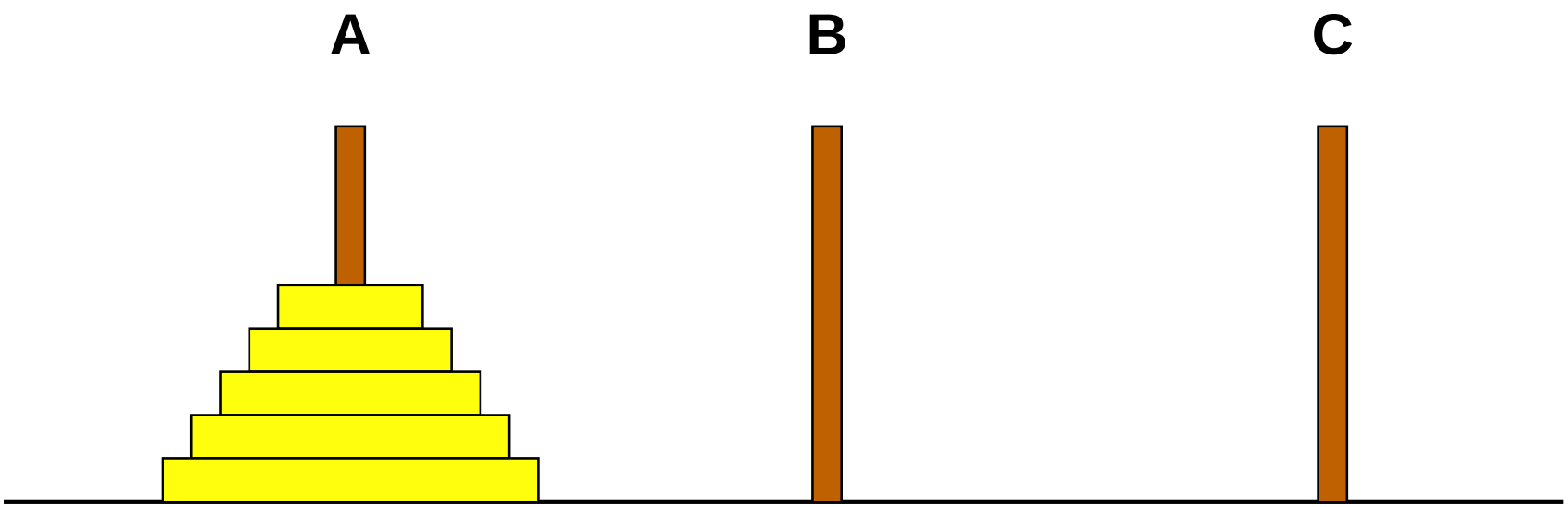
$n = 3$:

$A \to B$
$A \to C$
$B \to C$
$A \to B$
$C \to A$
$C \to B$
$A \to B$

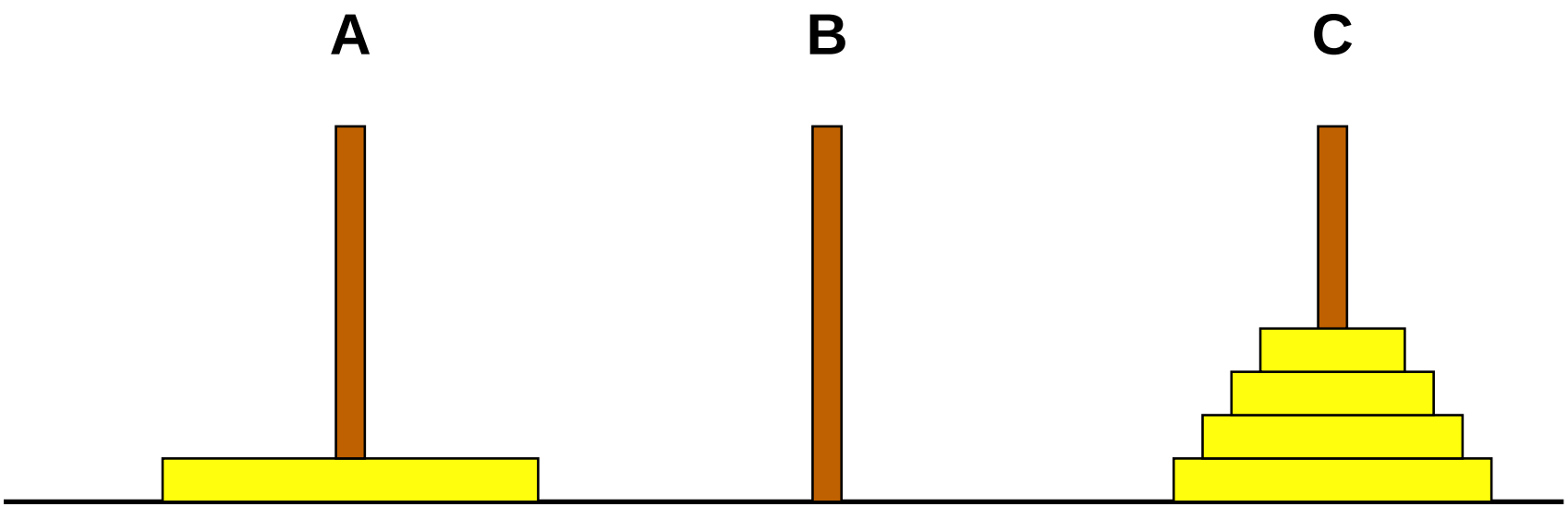
$n = 4$:

$A \to C$
$A \to B$
$C \to B$
$A \to C$
$B \to A$
$B \to C$
$A \to C$
$A \to B$
$C \to B$
$C \to A$
$B \to A$
$C \to B$
$A \to C$
$A \to B$
$C \to B$

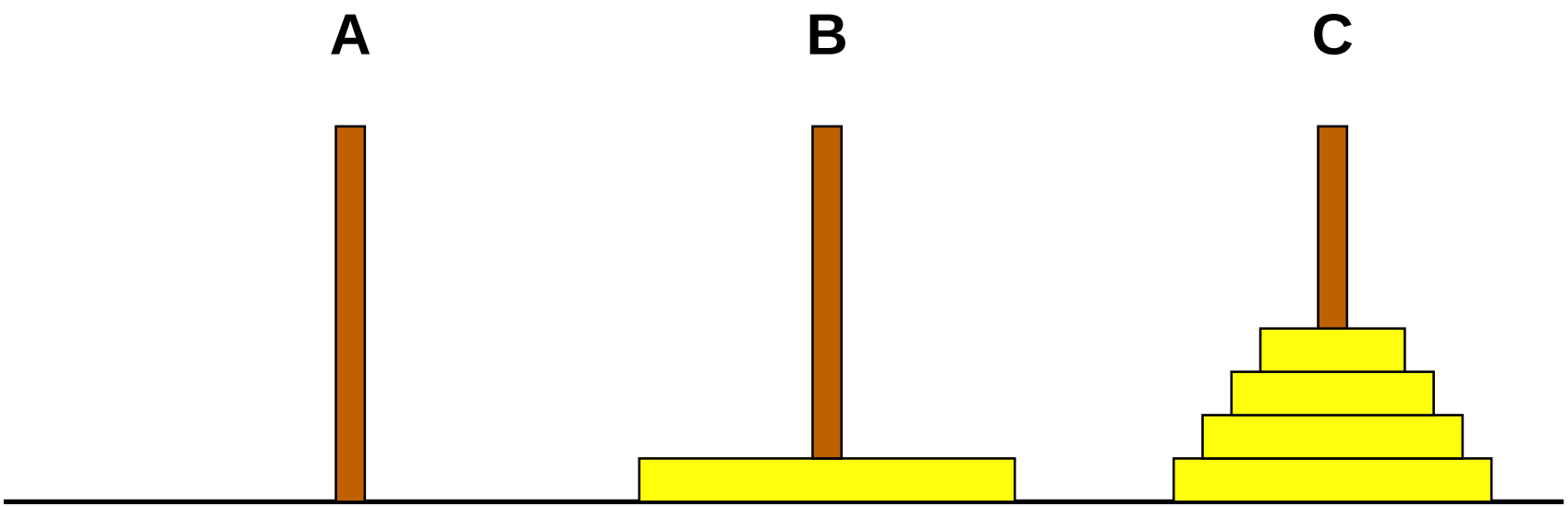
Hanojské věže



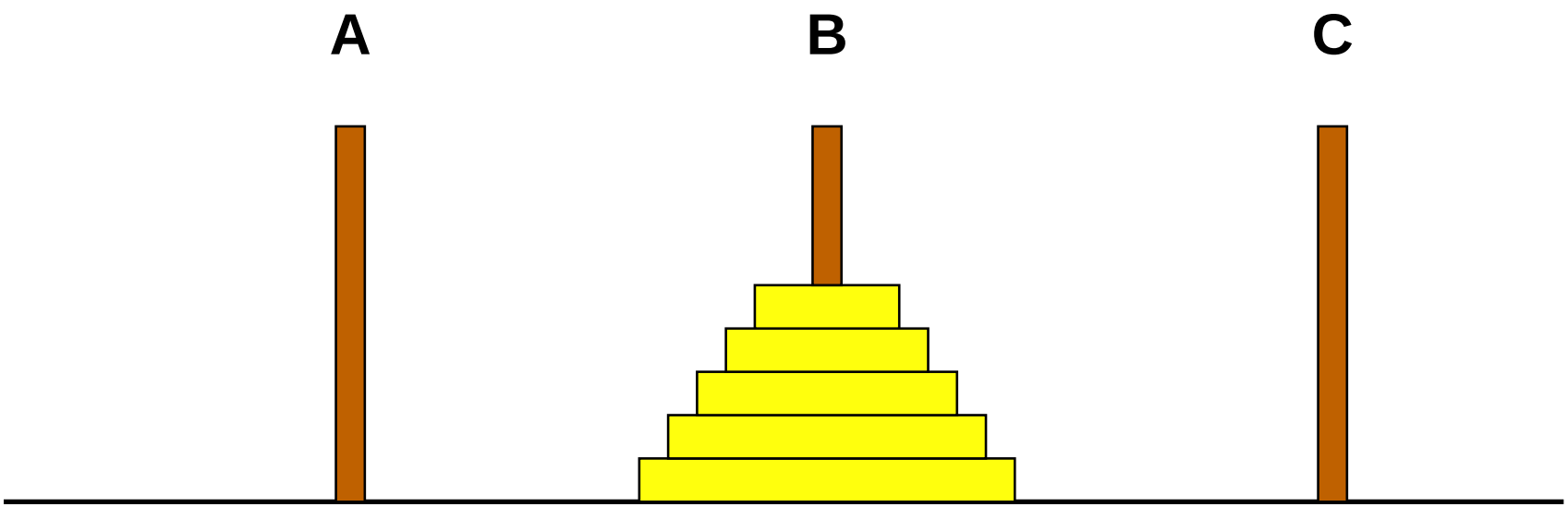
Hanojské věže



Hanojské věže



Hanojské věže



Hanojské věže

```
void hanoi(int n, char src, char dst, char tmp)
{
    if (n == 0) return;
    hanoi(n-1, src, tmp, dst);
    printf("%c -> %c\n", src, dst);
    hanoi(n-1, tmp, dst, src);
}

int main()
{
    int n;
    scanf("%d", &n);
    hanoi(n, 'A', 'B', 'C');
    return 0;
}
```

Hanojské věže

Označme $P(n)$ počet tahů, které provede algoritmus pro n disků.

Tvrzení

$P(n) = 2^n - 1$.

Důkaz:

- Pro $n = 0$: $P(n) = 0 = 2^0 - 1$
- Pro $n > 0$: Předpokládáme, že $P(n-1) = 2^{n-1} - 1$.

$$\begin{aligned} P(n) &= 2P(n-1) + 1 = \\ &= 2(2^{n-1} - 1) + 1 = \\ &= 2 \cdot 2^{n-1} - 2 + 1 = \\ &= 2^n - 1 \end{aligned}$$

Hanojské věže

Tvrzení

Pro přesun n disků je třeba minimálně $2^n - 1$ tahů.

Důkaz:

Indukcí.

Uvedený algoritmus nalezne tedy optimální řešení.

Poznámka

Otázka: Jak dlouho by trvalo přesunutí 64 disků, pokud by přesunutí jednoho disku trvalo 1 s ?

Hanojské věže

Tvrzení

Pro přesun n disků je třeba minimálně $2^n - 1$ tahů.

Důkaz:

Indukcí.

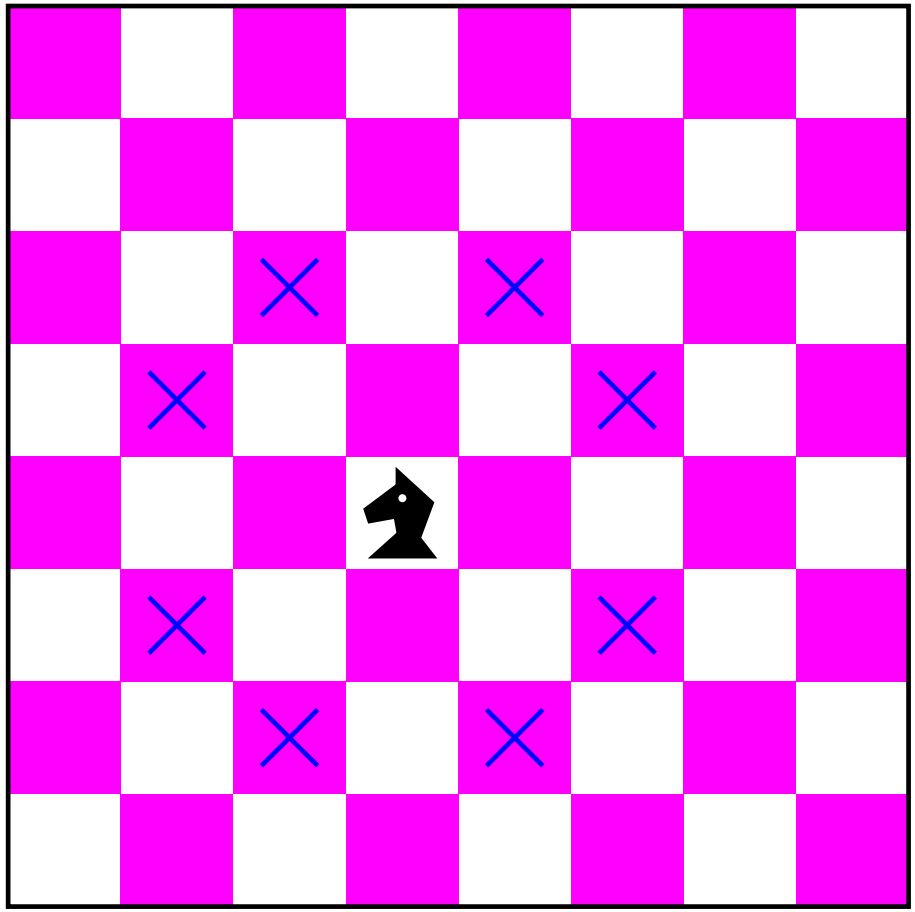
Uvedený algoritmus nalezne tedy optimální řešení.

Poznámka

Otázka: Jak dlouho by trvalo přesunutí 64 disků, pokud by přesunutí jednoho disku trvalo 1 s ?

Odpověď: 18446744073709551615 s, tj. asi 585 miliard let.

Jezdec na šachovnici



Problém: Najít cestu, na které jezdec navštíví každé políčko na šachovnici právě jednou.

Jezdec na šachovnici

Jedno z možných řešení:

1	38	55	34	3	36	19	22
54	47	2	37	20	23	4	17
39	56	33	46	35	18	21	10
48	53	40	57	24	11	16	5
59	32	45	52	41	26	9	12
44	49	58	25	62	15	6	27
31	60	51	42	29	8	13	64
50	43	30	61	14	63	28	7

```
int h[MAX][MAX];
int n, m;
bool found;
int moves[8][2] = {
    { 1, -2 }, { 2, -1 }, { 2, 1 }, { 1, 2 },
    { -1, 2 }, { -2, 1 }, { -2, -1 }, { -1, -2 }
};

void init()
{
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            h[i][j] = 0;
        }
    }
    m = n * n;
}
```

Jezdec na šachovnici

```
int main()
{
    scanf("%d", &n);
    init();
    if (search(1, 0, 0)) {
        print_solution();
    } else {
        printf("No solution\n");
    }
    return 0;
}
```

Jezdec na šachovnici

```
bool search(int k, int x, int y)
{
    h[y][x] = k;
    if (k == m) return true;
    for (int i = 0; i < 8; i++) {
        int x2 = x + moves[i][0];
        int y2 = y + moves[i][1];
        if (x2 < 0 || x2 >= n || y2 < 0 || y2 >= n) continue;
        if (h[y2][x2] != 0) continue;
        if (search(k+1, x2, y2)) return true;
    }
    h[y][x] = 0;
    return false;
}
```

Rekurzivní algoritmy

Rekurzivní algoritmus je algoritmus, který převede řešení původního problému na řešení několika podobných problémů pro menší instance.

Obecné schéma rekurzivních algoritmů:

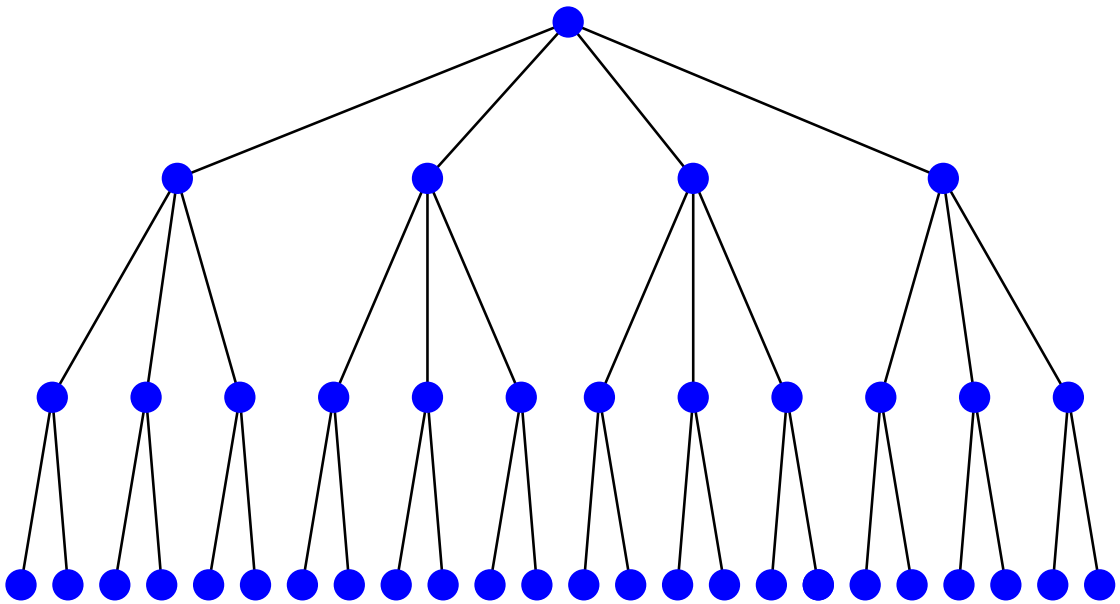
- Pokud se jedná o elementární případ, vyřeš ho přímo a vrať výsledek.
- V opačném případě vytvoř instance podproblémů.
- Zavolej sám sebe pro každou z těchto instancí.
- Z výsledků pro jednotlivé podproblémy slož řešení původního problému a vrať ho jako výsledek.

Poznámka: Instance podproblémů musí vždy být v nějakém smyslu menší než instance původního problému. Často (ne však vždy) se zmenšuje velikost instance.

Rekurzivní algoritmy

Výpočet rekurzivního algoritmu je možné znázornit jako strom:

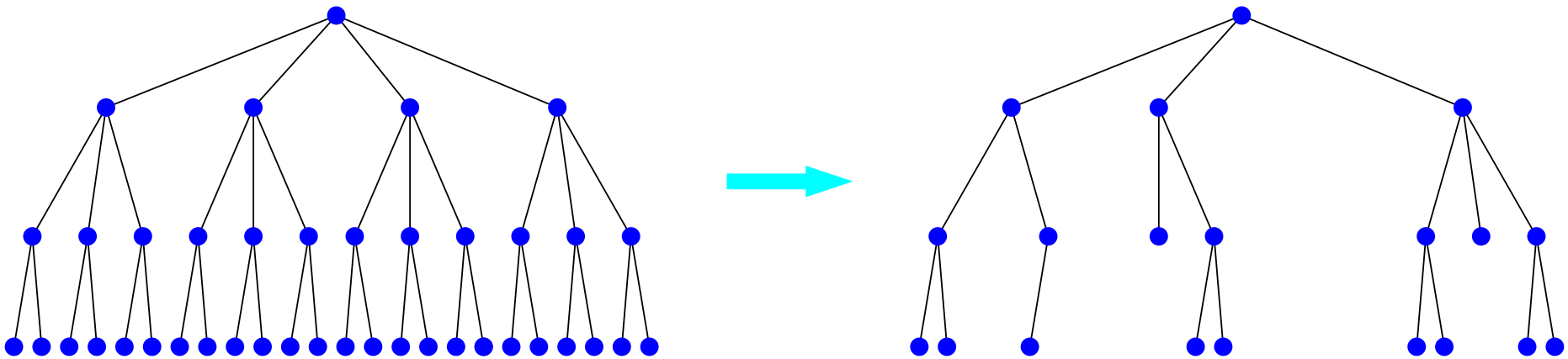
- vrcholy stromu odpovídají jednotlivým podproblémům
- kořen je původní problém
- potomci vrcholu odpovídají podproblémům daného problému



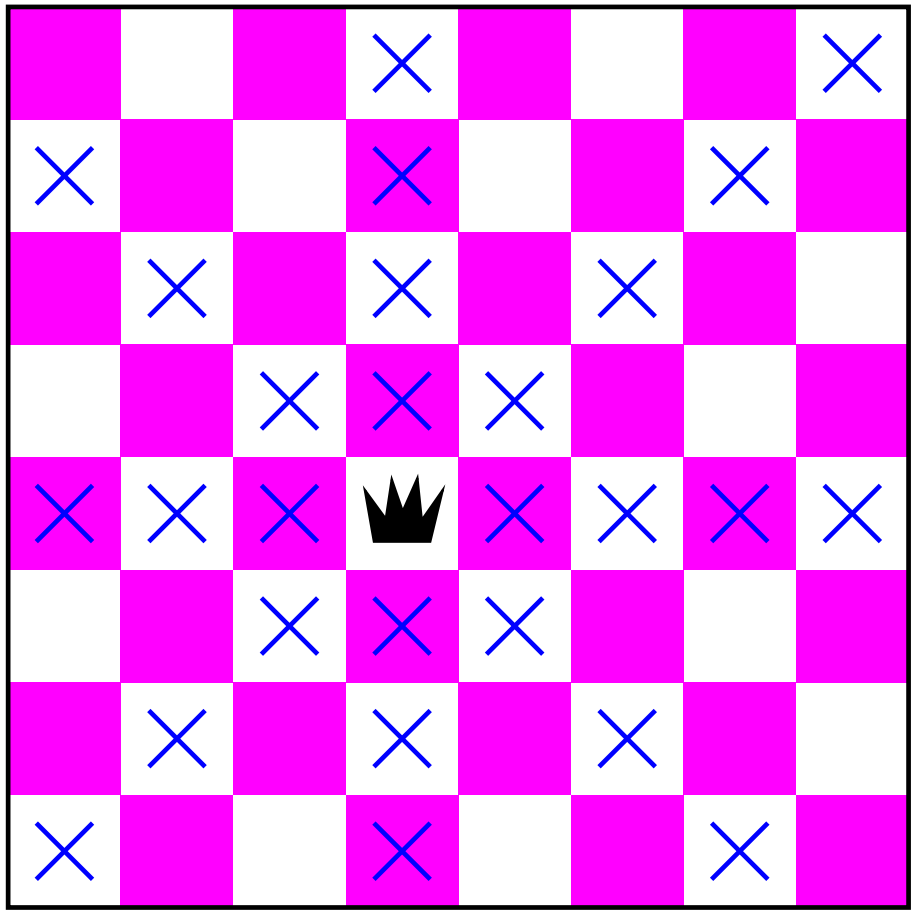
Pruning

Přístup, kdy do rekurzivním algoritmu doplníme nějaké vhodné testy, které způsobí, že se rekurzivní procedura nebude volat pro některé podproblémy, u kterých je jisté, že jejich vyřešení nepovede k řešení celého problému, se nazývá **pruning**.

Čím více větví stromu tímto způsobem odstraníme, tím lépe.



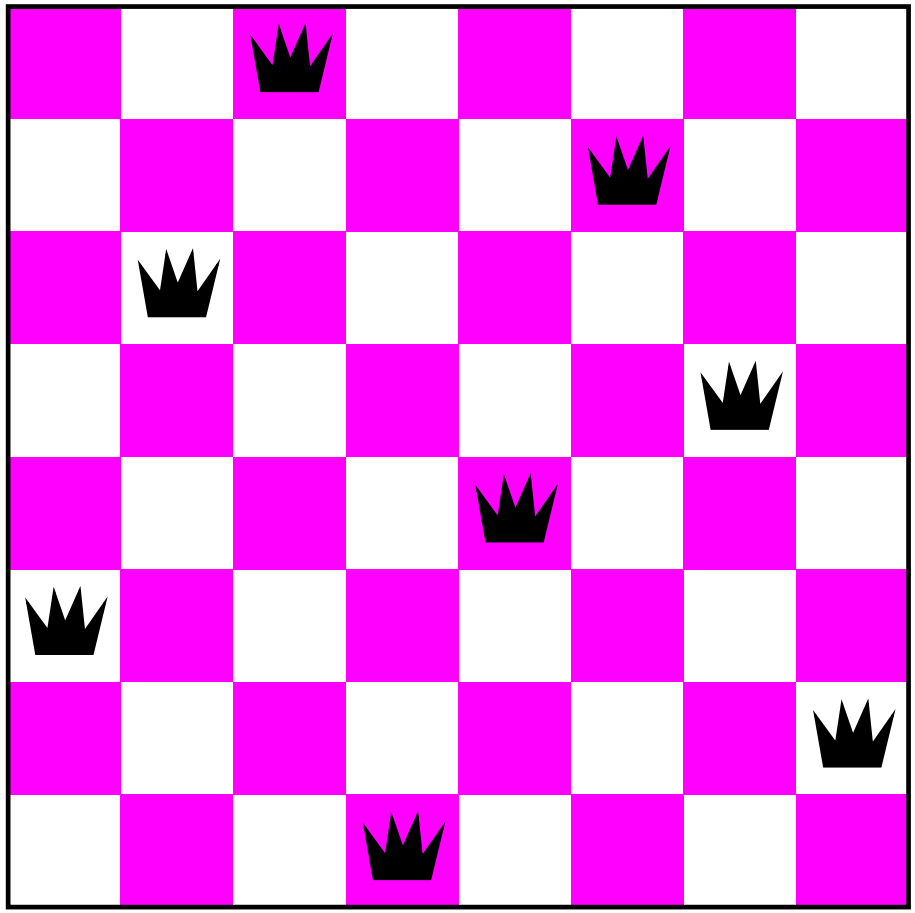
Problém osmi dam



Problém: Najít všechny možnosti, jak rozmístit n dam na šachovnici velikosti $n \times n$ tak, aby se žádné dvě dámy navzájem neohrožovaly.

Problém osmi dam

Jedno z možných řešení.



Problém osmi dam

```
int y[MAX];
bool a[MAX]; // a[i] – radek i je obsazeny
bool b[2*MAX]; // b[i] – diagonala 1
bool c[2*MAX]; // c[i] – diagonala 2
int n;

int main()
{
    ...
    search(0);
    ...
}
```

Problém osmi dam

```
void search(int k)
{
    if (k == n) {
        print_solution();
        return;
    }
    for (int i = 0; i < n; i++) {
        int i2 = k+i; int i3 = k-i+n;
        if (a[i] || b[i2] || c[i3]) continue;
        y[k] = i;
        a[i] = 1; b[i2] = 1; c[i3] = 1;
        search(k+1);
        a[i] = 0; b[i2] = 0; c[i3] = 0;
    }
}
```

Problém osmi dam

	0	1	2	3
0	0	1	2	3
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6

x+y

	0	1	2	3
0	0	1	2	3
1	−1	0	1	2
2	−2	−1	0	1
3	−3	−2	−1	0

x−y

Generování permutací

Permutace prvků určité množiny je jeden možný způsob, jak mohou být tyto prvky uspořádány do posloupnosti.

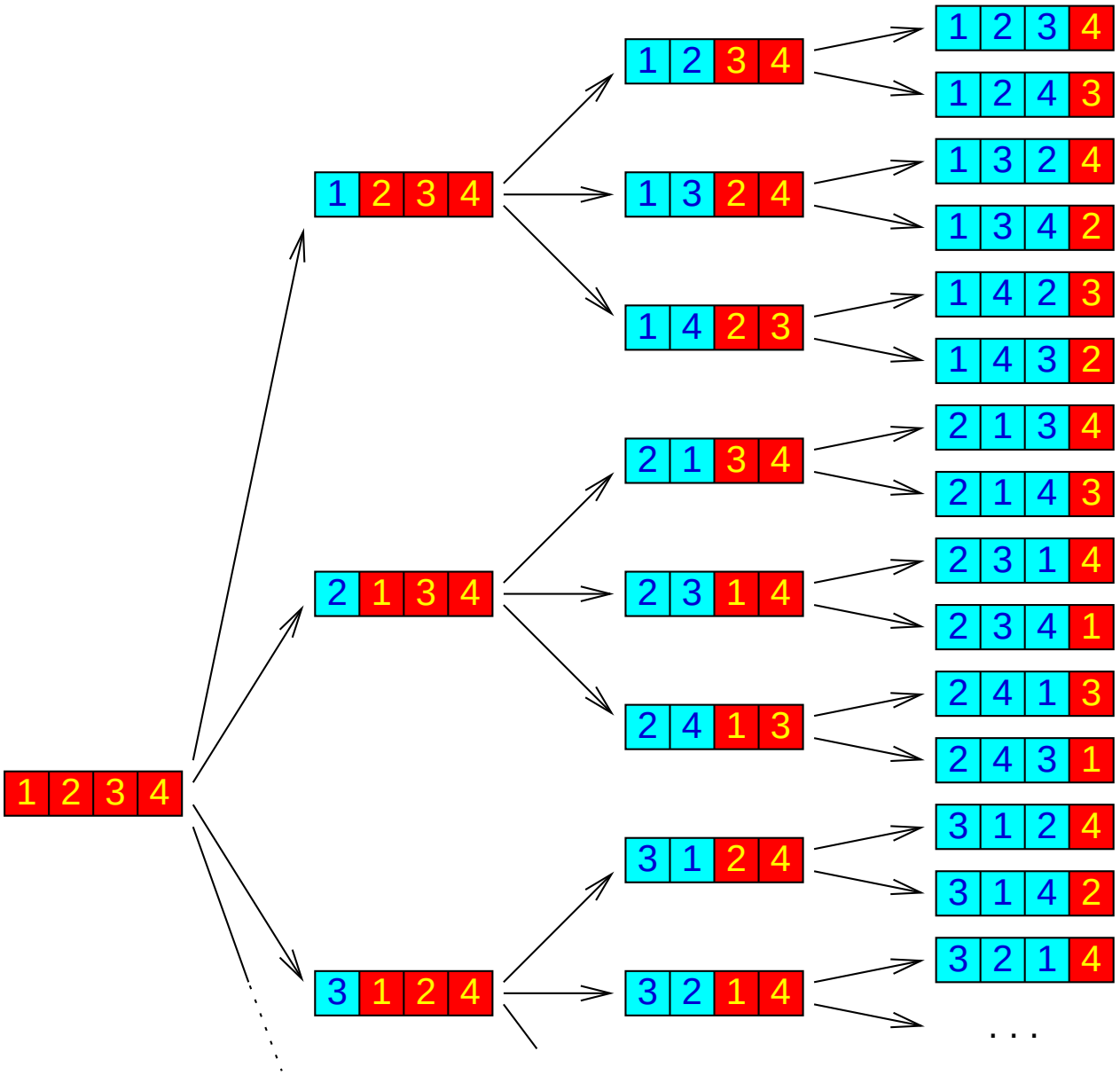
Například pro prvky $1, 2, 3, 4$ jsou všechny jejich permutace:

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

Celkový počet všech permutací n prvků je $n!$.

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot (n-1) \cdot n$$

Generování permutací



```
void gen(int k)
{
    if (k == n) {
        print_perm();
        return;
    }
    gen(k+1);
    for (int i = k+1; i < n; i++) {
        int tmp = a[i]; a[i] = a[k]; a[k] = tmp;
        gen(k+1);
    }
    int tmp = a[k];
    for (int i = k+1; i < n; i++) {
        a[i-1] = a[i];
    }
    a[n-1] = tmp;
}
```

Generování permutací

Označme $S(n)$ dobu, kterou stráví algoritmus prací na podproblému velikosti n bez započtení doby strávené řešením jeho podproblémů. Označme $T(n)$ dobu, kterou stráví algoritmus prací na podproblému velikosti n včetně práce na jeho podproblémech.

Zjevně platí, že $S(n) \leq an + b$ pro nějaké konstanty a a b.

Tvrzení

Pokud $S(n) \leq an + b$ pro nějaké konstanty a a b, pak $T(N) \in O(n! \cdot n)$.

Tvrzení

Pokud $S(n) \leq an + b$ pro nějaké konstanty a a b, pak $T(N) \in O(n! \cdot n)$.

Důkaz: Indukcí ukážeme, že $T(n) \leq (a+b) \cdot n! \cdot n$:

- Pro $n = 1$: $T(1) = S(1) \leq a \cdot 1 + b = (a+b) \cdot 1! \cdot 1$
- Pro $n > 1$:

$$\begin{aligned} T(n) &= n \cdot T(n-1) + S(n) \leq \\ &\leq n \cdot (a+b) \cdot (n-1)! \cdot (n-1) + an + b \leq \\ &\leq (a+b) \cdot n! \cdot (n-1) + (a+b) \cdot n = \\ &= (a+b)(n! \cdot (n-1) + n) \leq \\ &\leq (a+b)(n! \cdot (n-1) + n!) = \\ &= (a+b) \cdot n! \cdot n \end{aligned}$$

Generování permutací

Permutace je možné generovat i nerekurzivně (n – počet prvků):

```
int* a = new int[n];
for (int i = 0; i < n; i++) {
    a[i] = i+1;
}
print_perm(a, n);
while (next_perm(a, n)) {
    print_perm(a, n);
}
delete a;
```

```
bool next_perm(int a[], int n) {
    int i = n-2;
    while (i >= 0) {
        if (a[i] < a[i+1]) break;
        i--;
    }
    if (i < 0) return false;
    int j = n-1;
    while (a[j] < a[i]) j--;
    int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
    i++;
    j = n-1;
    while (i < j) {
        tmp = a[i]; a[i] = a[j]; a[j] = tmp;
        i++; j--;
    }
    return true;
}
```

Odstranění rekurze

Původní funkce používající rekurzi:

```
void f(Node *v)
{
    if (v == NULL) return;
    f(v->left);
    print(v->value);
    f(v->right);
}
```

Odstranění rekurze

Odstranění tail rekurze:

```
void f(Node *v)
{
L1: if (v == NULL) return;
    f(v->left);
    print(v->value);
    v = v->right;
    goto L1;
}
```

Odstranění rekurze

Nahrazení goto cyklem while:

```
void f(Node *v)
{
    while (v != NULL) {
        f(v->left);
        print(v->value);
        v = v->right;
    }
}
```

Odstranění rekurze

Odstranění „obyčejné“ rekurze použitím zásobníku:

```
void f(Node *v)
{
    Stack s;
L1: while (v != NULL) {
        s.push(v);
        v = v->left;
        goto L1;
    L2: v = s.pop();
        print(v->value);
        v = v->right;
    }
    if (!s.is_empty()) goto L2;
}
```

Odstranění rekurze

Přeuspořádání kódu, nahrazení goto pomocí cyklů:

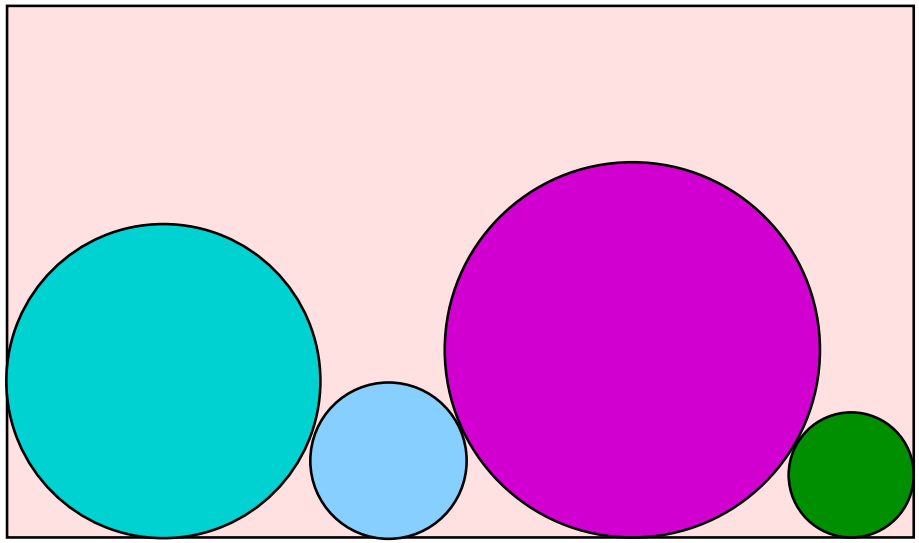
```
void f(Node *v)
{
    Stack s;
    while (true) {
        while (v != NULL) {
            s.push(v);
            v = v->left;
        }
        if (s.is_empty()) return;
        v = s.pop();
        print(v->value);
        v = v->right;
    }
}
```

Problém „How Big Is It?“ (10012)

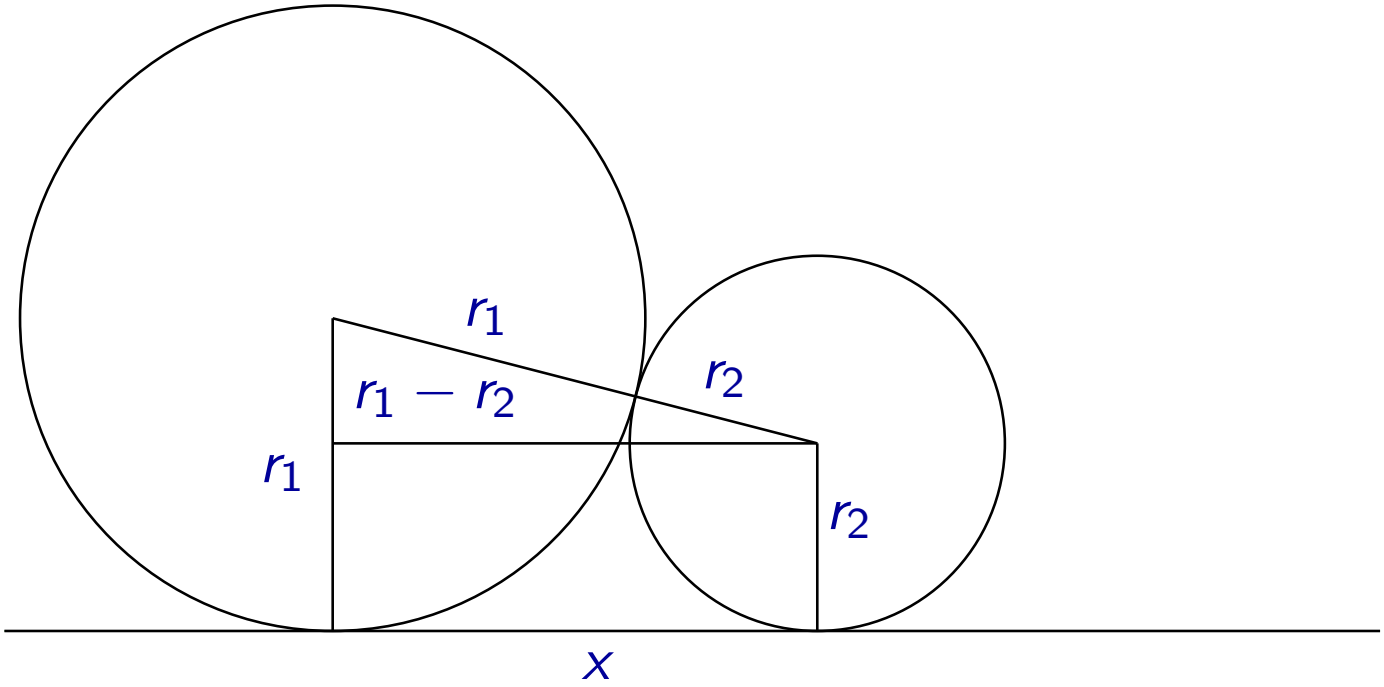
Problém

Vstup: Množina kruhů (jejich poloměrů).

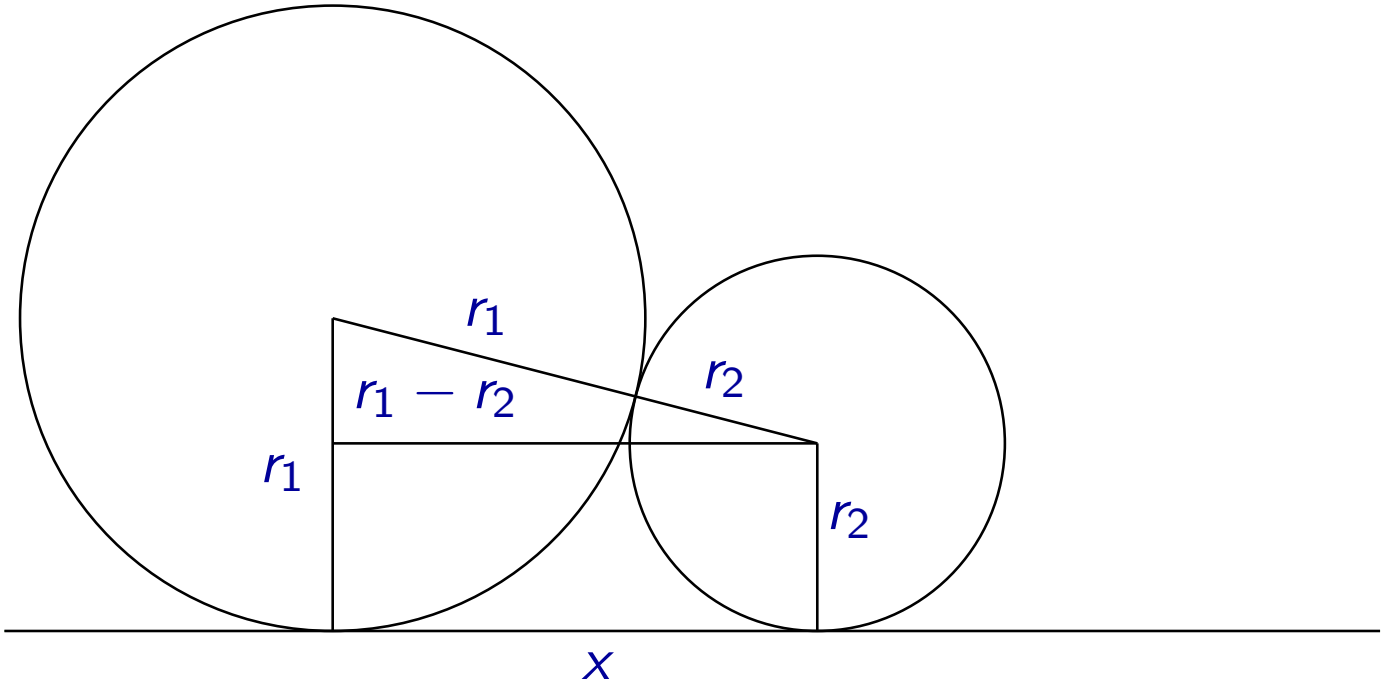
Výstup: Minimální šířka obdélníka, do kterého se vejdou všechny kruhy tak, aby se všechny dotýkaly spodního okraje obdélníka.



Problém „How Big Is It?“ (10012)

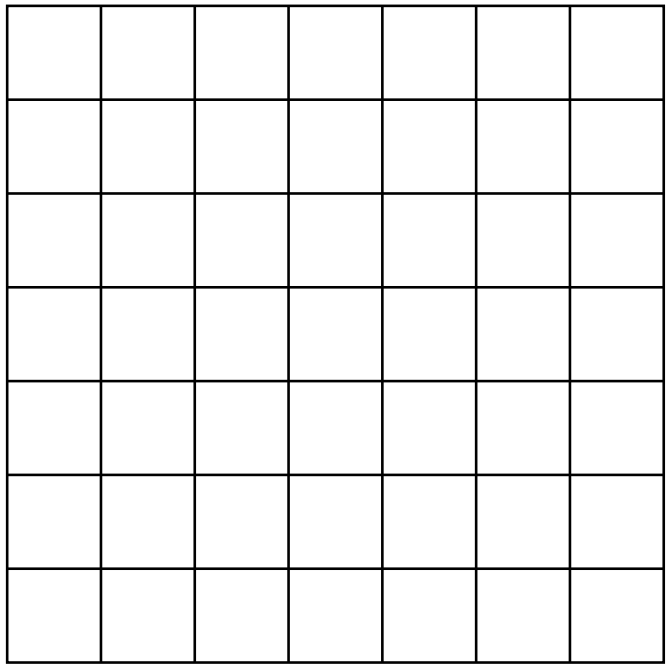


Problém „How Big Is It?“ (10012)



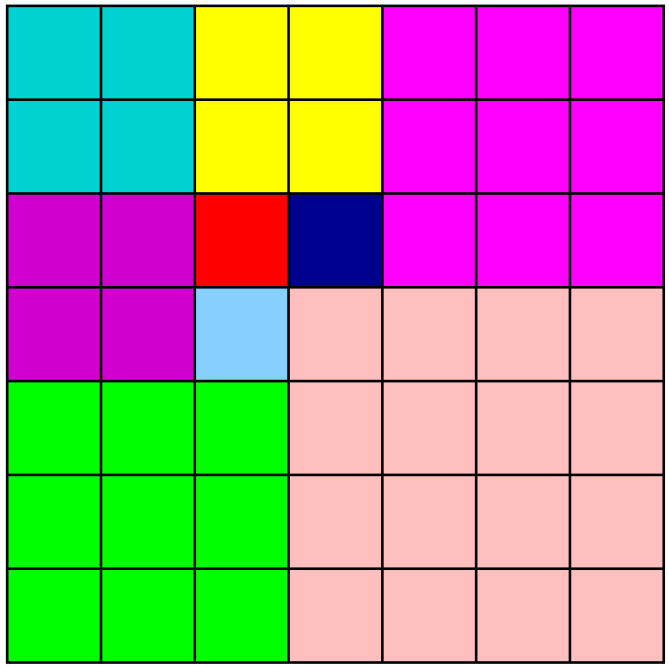
$$
\begin{aligned}
x^2 &= (r_1 + r_2)^2 - (r_1 - r_2)^2 = \\
&= r_1^2 + 2r_1r_2 + r_2^2 - (r_1^2 - 2r_1r_2 + r_2^2) = \\
&= 4r_1r_2 \\
x &= \sqrt{4r_1r_2}
\end{aligned}
$$

Problém „Bigger Square Please . . . “ (10270)



Úkol: Najít způsob, jak rozdělit čtverec velikosti $n \times n$ ($2 \leq n \leq 50$) na co nejmenší počet čtverců, jejichž velikosti mohou být 1×1, 2×2, . . . , $(n-1) \times (n-1)$.

Problém „Bigger Square Please . . . “ (10270)



Úkol: Najít způsob, jak rozdělit čtverec velikosti $n \times n$ ($2 \leq n \leq 50$) na co nejmenší počet čtverců, jejichž velikosti mohou být 1×1, 2×2, . . . , $(n-1) \times (n-1)$.

Greedy algoritmy

Shoemaker's Problem (10026)

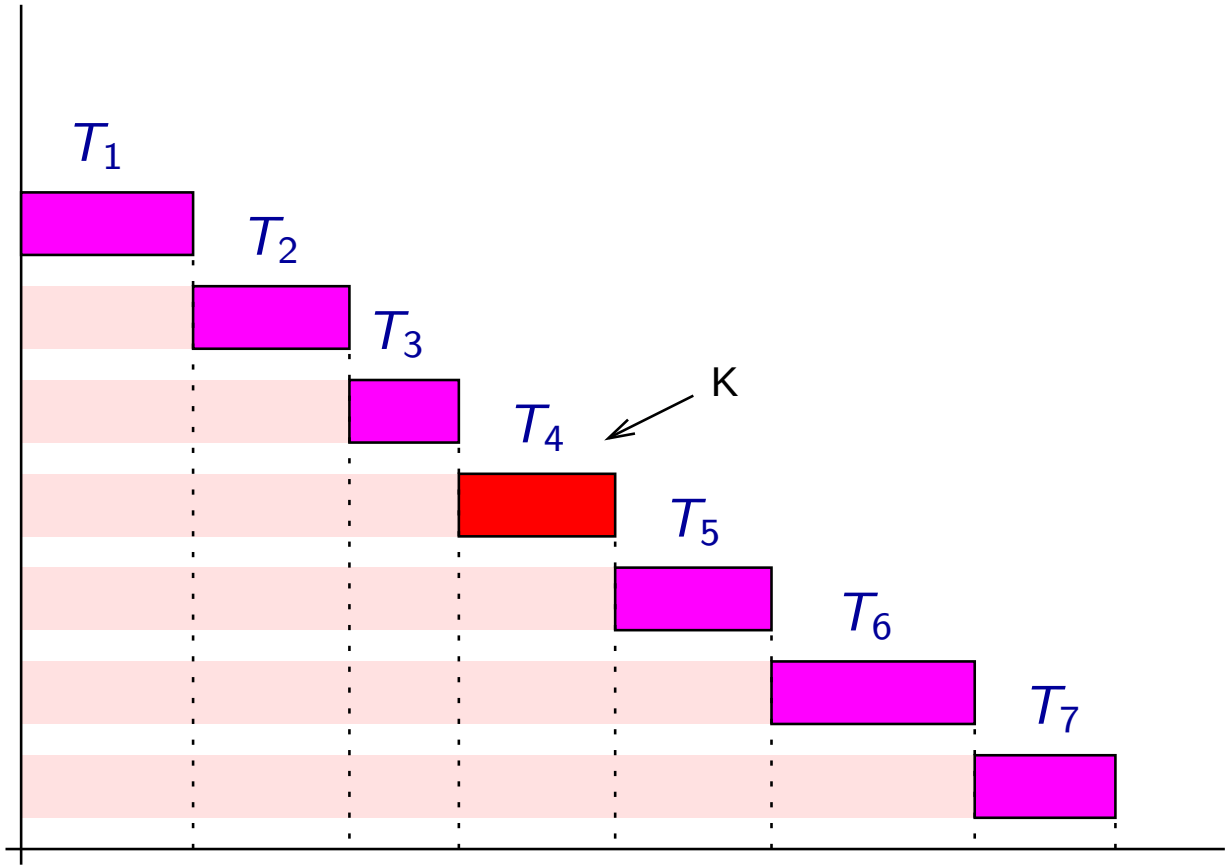
- Obuvník nabral od zákazníků mnoho zakázek.
- U každé zakázky je předem známo, jak dlouho bude trvat práce na této zakázce (čas T_i pro i-tou zakázku).
- Obuvník je schopen pracovat najednou jen na jedné zakázce.
- U každé zakázky platí obuvník dohodnuté penále za každý den zpoždění, kdy na zakázce nepracuje (částku S_i pro i-tou zakázku).

Problém:

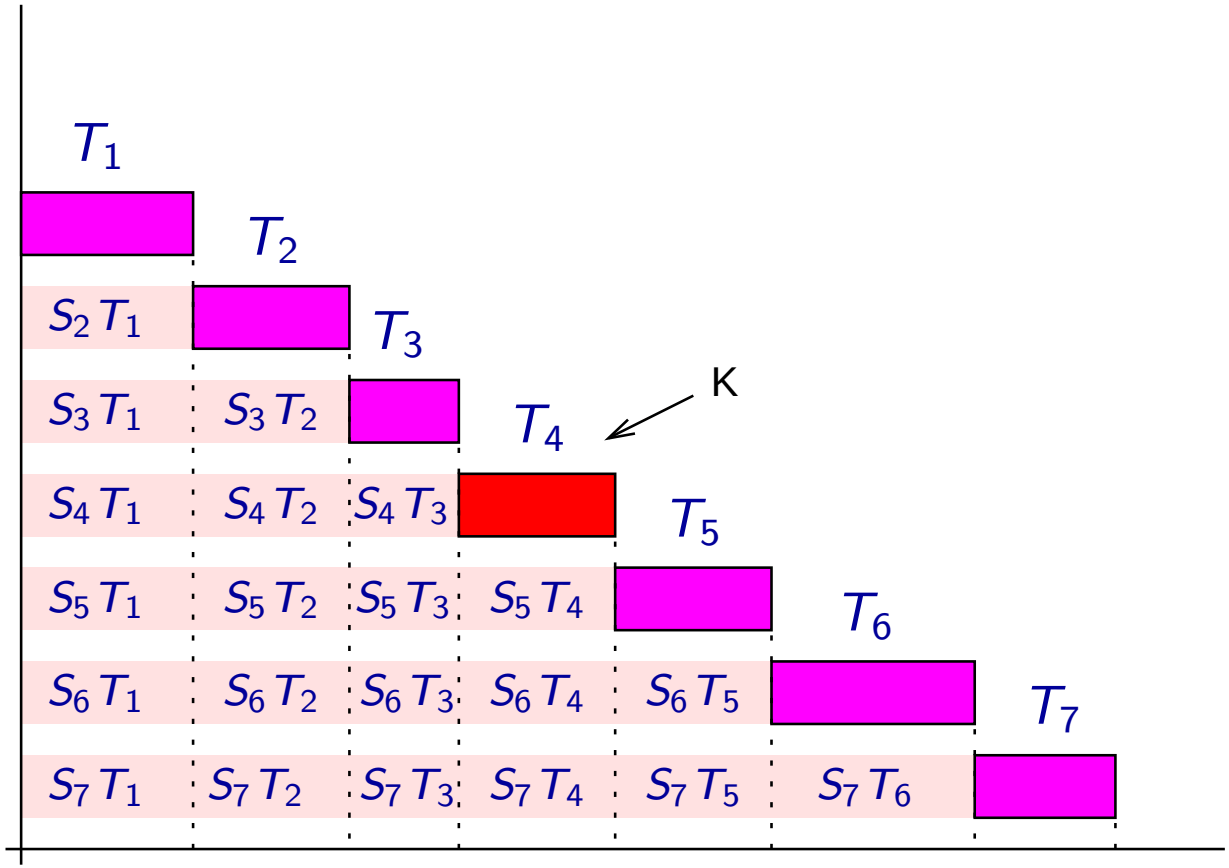
Vstup: Počet zakázek N a hodnoty T_i a S_i pro zakázky 1 až N.

Výstup: Optimální pořadí zakázek, kdy obuvník zaplatí na penále minimální částku.
(Pokud existuje více optimálních řešení, zvolit z nich lexikograficky nejmenší.)

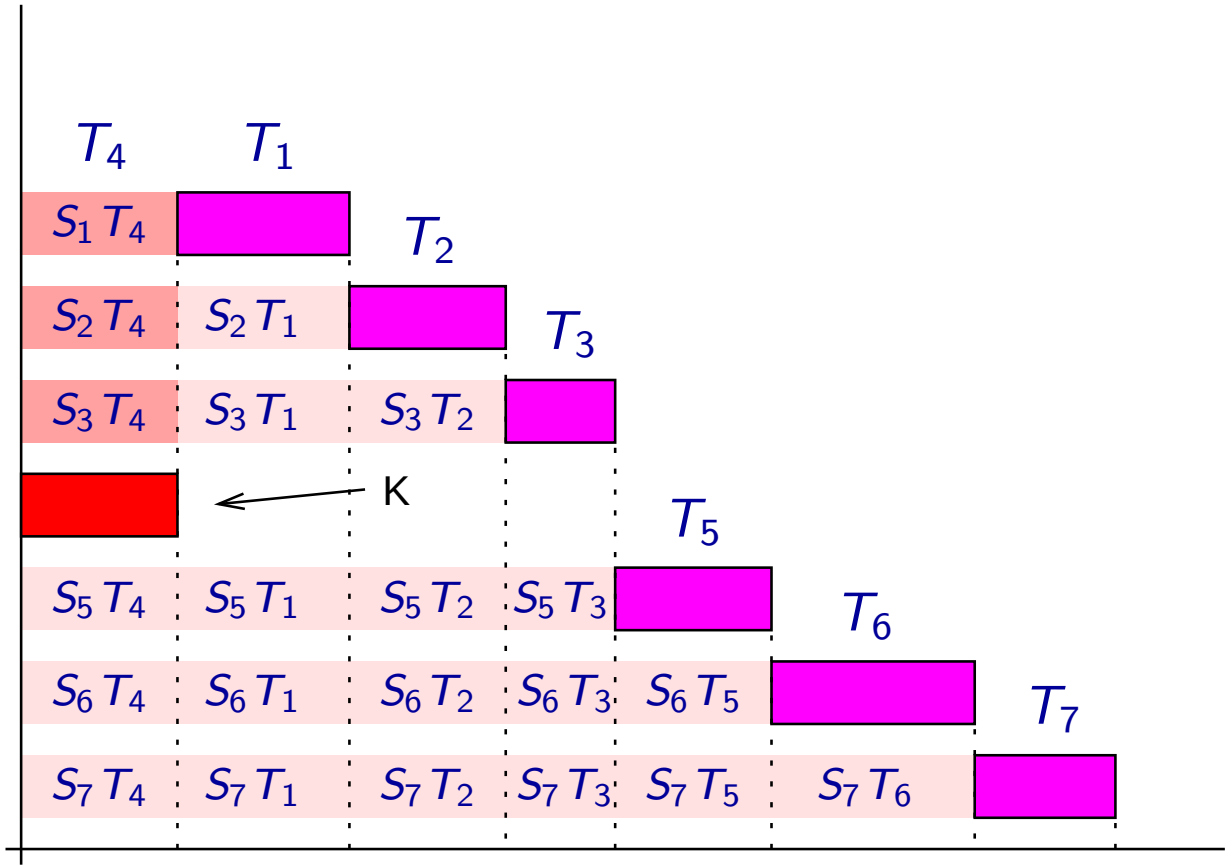
Shoemaker's Problem (10026)



Shoemaker's Problem (10026)



Shoemaker's Problem (10026)



Shoemaker's Problem (10026)

Zvolme K takové, že

$$\forall i : \frac{S_K}{T_K} \geq \frac{S_i}{T_i}$$

z čehož plyne, že

$$\forall i : S_K T_i \geq S_i T_K$$

neboli

$$\forall i : S_K T_i - S_i T_K \geq 0$$

Označme S původní částku. Pokud zakázku K přesuneme na první místo, zaplatíme částku S':

$$S' = S - \sum_{i<K} S_K T_i + \sum_{i<K} S_i T_K = S - \sum_{i<K} (S_K T_i - S_i T_K) \leq S$$

Greedy algoritmy

Definice

Greedy algoritmy jsou algoritmy, které hledají globálně optimální řešení pomocí posloupnosti lokálně optimálních voleb.

Návrh greedy algoritmu:

1. Určit strukturu problému, určit co jsou podproblémy.
2. Navrhnout rekurzivní řešení.
3. Důkaz, že v každém kroku je možné určit optimální volbu.
4. Iterativní implementace.

Poznámka: Na greedy algoritmy se lze dívat jako na do extrému dotažený pruning, kde nám v každém kroku zbývá jediná volba.

The Grand Dinner (10249)

Na závěr soutěže se koná slavnostní večeře, které se účastní členové všech týmů.
Chceme je rozesadit tak, aby u žádní dva členové jednoho týmu neseděli u stejného stolu.

Problém:

Vstup: Počet týmů, počet stolů, počty členů jednotlivých týmů a počty míst u jednotlivých stolů.

Výstup: Rozesazení všech členů všech týmů, tak aby žádní dva členové jednoho týmu neseděli u téhož stolu, nebo informace o tom, že řešení neexistuje.

Dynamické programování

Problém

Vstup: Čísla $a_1, a_2, \ldots, a_n$ a číslo m.

Otázka: Existuje nějaká podmnožina čísel $a_1, a_2, \ldots, a_n$ taková, že součet čísel v této podmnožině je m?

Poznámka: Správnější by bylo mluvit o **multimnožině**.

Příklad: $\{8, 5, 12, 9, 5\}$, $m = 34$

Problém

Vstup: Čísla $a_1, a_2, \ldots, a_n$ a číslo m.

Otázka: Existuje nějaká podmnožina čísel $a_1, a_2, \ldots, a_n$ taková, že součet čísel v této podmnožině je m?

Poznámka: Správnější by bylo mluvit o **multimnožině**.

Příklad: $\{8, 5, 12, 9, 5\}$, $m = 34$

$$8 + 12 + 9 + 5 = 34$$

Problém

Vstup: Čísla $a_1, a_2, \ldots, a_n$ a číslo m.

Otázka: Existuje nějaká podmnožina čísel $a_1, a_2, \ldots, a_n$ taková, že součet čísel v této podmnožině je m?

Poznámka: Správnější by bylo mluvit o **multimnožině**.

Příklad: $\{8, 5, 12, 9, 5\}$, $m = 34$

$$8 + 12 + 9 + 5 = 34$$

Problém budeme chtít řešit v krátkém čase (řekněme do 1 sekundy) pro vstupy, kde $n \leq 1000$ a $m \leq 32000$.

Dynamické programování

Správné, ale neefektivní rekurzivní řešení (hrubou silou):

```
bool solve(int i, int j) {
    bool result;
    if (i == 0) {
        result = (j == 0) ? 1 : 0;
    } else {
        result = solve(i - 1, j);
        if (!result && a[i - 1] <= j) {
            result = solve(i - 1, j - a[i - 1]);
        }
    }
    return result;
}
```

Dynamické programování

Idea: Ukládat si řešení podproblémů, které jsem už vyřešil, do tabulky.

```
bool solve(int i, int j) {
    if (table[i][j] >= 0) return table[i][j];
    bool result;
    if (i == 0) {
        result = (j == 0) ? 1 : 0;
    } else {
        ...
    }
    table[i][j] = result;
    return result;
}
```

Poznámka: Prvky pole table jsou inicializovány na -1.

Nerekurzivní řešení – systematicky řešíme všechny možné podproblémy od nemenších po největší.

```
bool solve(int n, int m) {
    table[0][0] = 1
    for (int k = 1; k <= m) table[0][k] = 0;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            bool result = table[i − 1][j];
            if (!result && a[i − 1] <= j) {
                result = table[i − 1][j − a[i − 1]];
            }
            table[i][j] = result;
        }
    }
    return table[n][m];
}
```

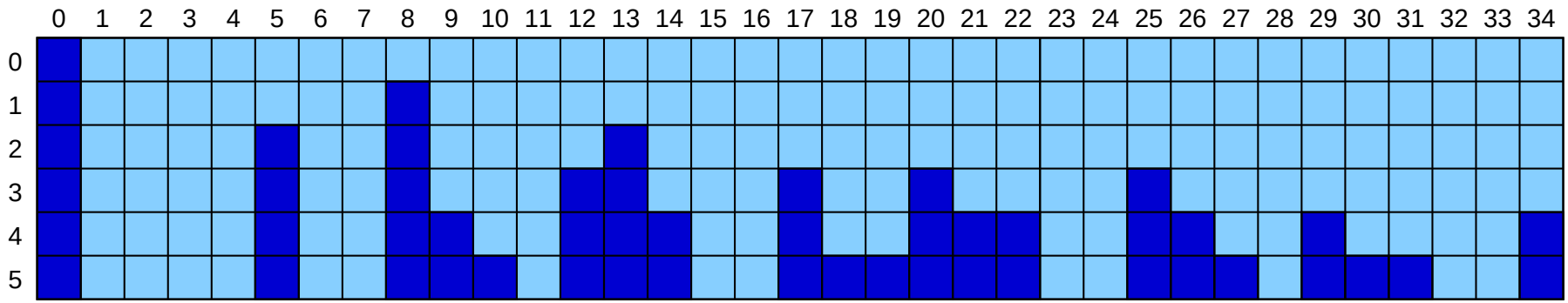
Dynamické programování

Možná vylepšení předchozího programu:

- Program si nemusí pamatovat celé pole table, stačí, když si pamatuje aktuální a jeden předchozí řádek – šetří paměť.
- Vzhledem k tomu, že prvky pole table jsou jen 0 a 1, použít bitová pole – šetří paměť i čas.

Dynamické programování

$$a = \{8, 5, 12, 9, 5\}, \quad m = 34$$



Dynamické programování - memoization

Pro **dynamické programování** je charakteristické použití nějaké vhodné datové struktury (nejčastěji pole), kam si ukládáme řešení již vyřešených problémů.

Memoization – postup shora dolů (od větších podproblémů k menším) jako u rekurzivních algoritmů spolu s ukládáním výsledků pro jednotlivé podproblémy.

- Je nutné nějak evidovat, které podproblémy jsou už vyřešené.
- Řeší se jen ty podproblémy, jejichž řešení jsou třeba.

Dynamické programování

Dynamické programování – postup zdola nahoru (od menších podproblémů k větším):

- Místo rekurzivního volání se jen rovnou sahá pro výsledek podproblému do tabulky.
- Podproblémy je třeba řešit ve vhodném pořadí, aby bylo zaručeno, že již byly vyřešeny menší podproblémy, jejichž řešení je třeba k vyřešení podproblému, který se aktuálně řeší.
- Řeší se i podproblémy, které nejsou pro spočítání celkového výsledku nutné.
- Někdy umožňuje efektivnější implementaci než memoization.

The 3n+1 problem (100)

Uvažujme následující program:

```
print n
while n > 1:
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3 * n + 1
    print n
```

Například pro $n = 22$ vypíše:

22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1

The 3n+1 problem (100)

Definujeme $f(n)$ jako počet čísel, které program vypíše pro vstup n.
Například $f(22) = 16$.

Problém „The 3n+1 problem“:

Vstup: Dvě celá kladná čísla i a j ($0 < i, j < 1000000$).

Výstup: Maximum z hodnot $f(i)$, $f(i+1)$, ..., $f(j)$

The 3n+1 problem (100)

Neefektivní rekurzivní řešení:

```
int f(int n)
{
    if (n == 1) return 1;
    return (n % 2 == 0) ? f(n/2) + 1 : f(3*n+1) + 1;
}
```

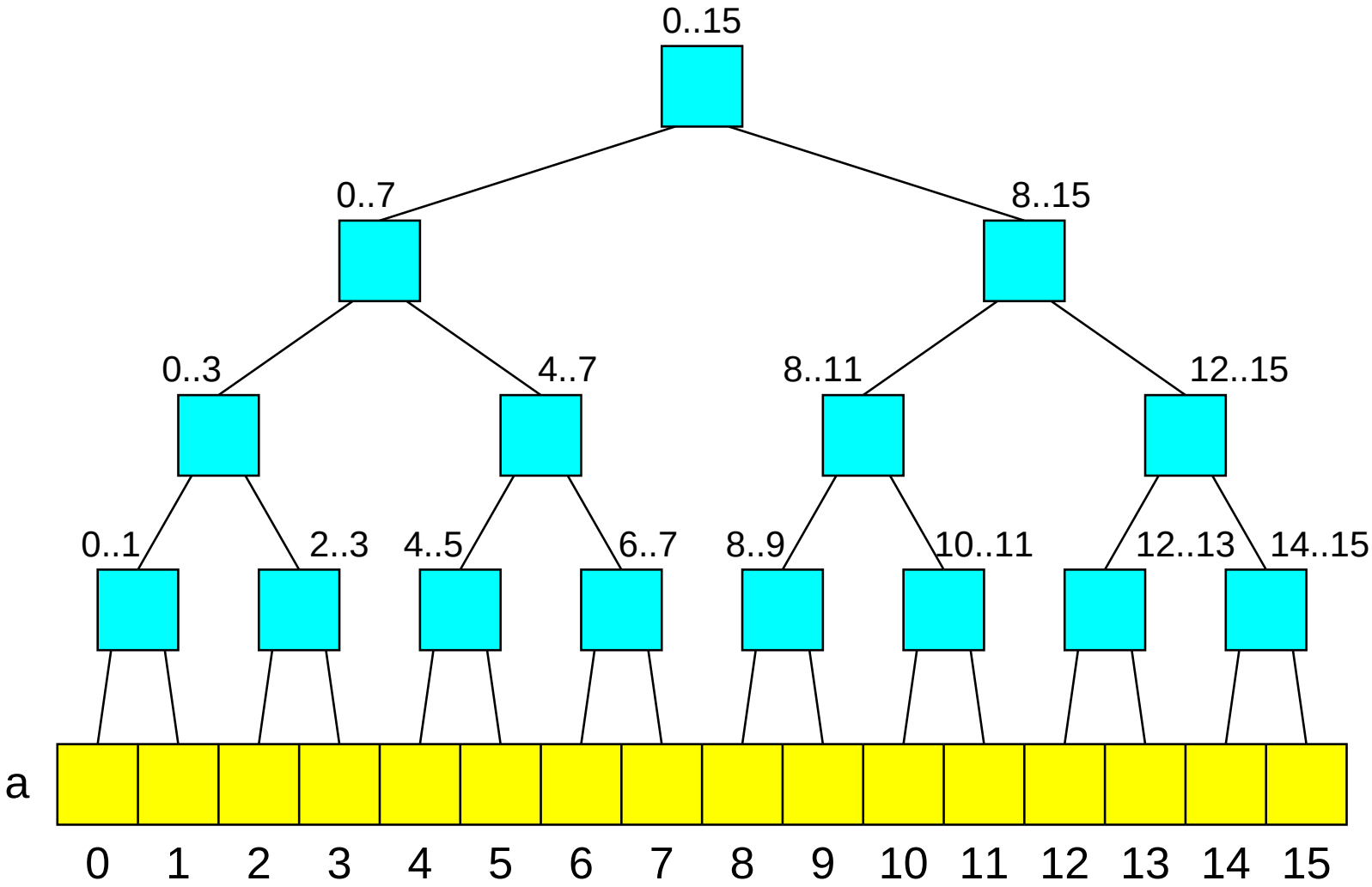
The 3n+1 problem (100)

Řešení s použitím memoizace:

```
int f(int n)
{
    if (n < MAX && a[n] != 0) return a[n];
    int x = (n % 2 == 0) ? f(n/2) + 1 : f(3*n+1) + 1;
    if (n < MAX) a[n] = x;
    return x;
}
```

Poznámka: Pole a je inicializováno na hodnoty 0, pouze $a[1] = 1$.

The 3n+1 problem (100)



The 3n+1 problem (100)

Pozorování

Binární strom, který má n listů a kde ostatní vrcholy mají vždy dva potomky, má celkem $2n - 1$ vrcholů.

Důkaz: Indukcí.

Problém „Expressions“ (10157)

Označme X množinu všech dobře utvořených výrazů tvořených pouze znaky "(" a ")":

- $\varepsilon \in X$
- Pokud $A \in X$, pak $(A) \in X$.
- Pokud $A, B \in X$, pak $AB \in X$.

Například "()(())()" $\in X$, "((()(()))" $\in X$, ale "())(" $\notin X$.

Délka výrazu E je počet znaků v E.

Hloubka výrazu E, označená $d(E)$, je definována takto:

- $d(\varepsilon) = 0$
- $d(E) = d(A) + 1$, jestliže $E = (A)$, kde $A \in X$.
- $d(E) = \max\{d(A), d(B)\}$, jestliže $E = AB$, kde $A, B \in X$.

Problém „Expressions“ (10157)

Problém

Vstup: Délka výrazu n a hloubka výrazu d (kde $2 \leq n \leq 300$ a $1 \leq d \leq 150$).

Výstup: Počet všech dobře utvořených výrazů délky n a hloubky d.

Naříklad pro $n = 6$ a $d = 2$ je odpověď 3:

$$(())() \qquad ()(()) \qquad (()())$$

Problém „Piggy-Bank“ (CERC 1999)

Máme prasátko (pokladničku), které je plné peněz. Chceme zjistit (minimální) částku, kterou obsahuje, ale nechceme ho rozbít. Známe váhu plného i prázdného prasátka a víme, jaké jsou různé typy mincí – známe jejich váhu a cenu.

Problém

Vstup: Váha prázdného prasátka E a váha plného prasátka F ($1 \leq E \leq F \leq 10000$).
Popis N typů mincí, pro každý typ mince její váha a cena ($1 \leq N \leq 500$).

Výstup: Minimální částka, kterou prasátko obsahuje, nebo informace, že řešení neexistuje.

Problém „Is Bigger Smarter?“ (10131)

Chceme vyvrátit hypotézu, že čím je slon těžší, tím vyšší je jeho IQ. Máme informace o nějaké množině slonů, kde u každého slona známe jeho váhu a IQ.
Úkolem je najít co nejdelší posloupnost slonů, kde postupně roste váha a klesá IQ.

Problém

Vstup: Hodnoty $W[i]$ a $S[i]$ pro $i \in \{1, \ldots, n\}$ (kde $n \leq 1000$).

Výstup: Posloupnost $a[1], a[2], \ldots, a[m]$ taková, že

$$W[a[1]] < W[a[2]] < \cdots < W[a[m]]$$

$$S[a[1]] > S[a[2]] > \cdots > S[a[m]]$$

a kde m je maximální.

Problém „Weights and Measures“ (10154)

Vršíme na sebe želvy. U každé želvy známe její váhu a víme kolik každá želva unese. Počet želv je maximálně 5607.

Otázka zní, jaký je maximální počet želv, které můžeme na sebe navršit tak, aby žádná želva nenesla větší váhu než unese.

Cutting Sticks (10003)

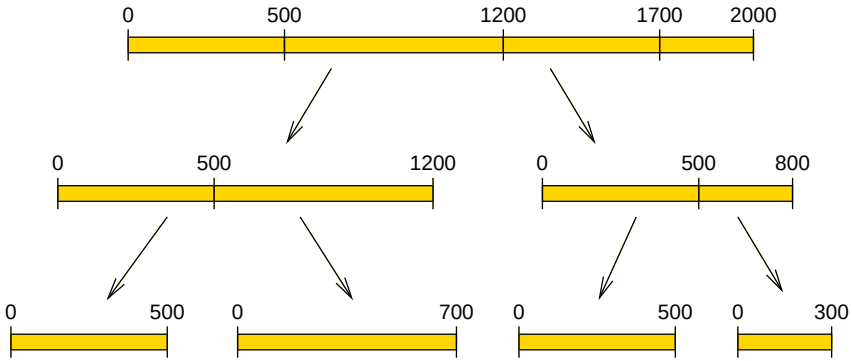
Máme za úkol rozřezat tyč v určených místech.
Za každý řez platíme cenu, která je úměrná délce kusu tyče, který právě rozřezáváme.
Řezy můžeme provádět v libovolném pořadí.

Problém „Cutting Sticks“

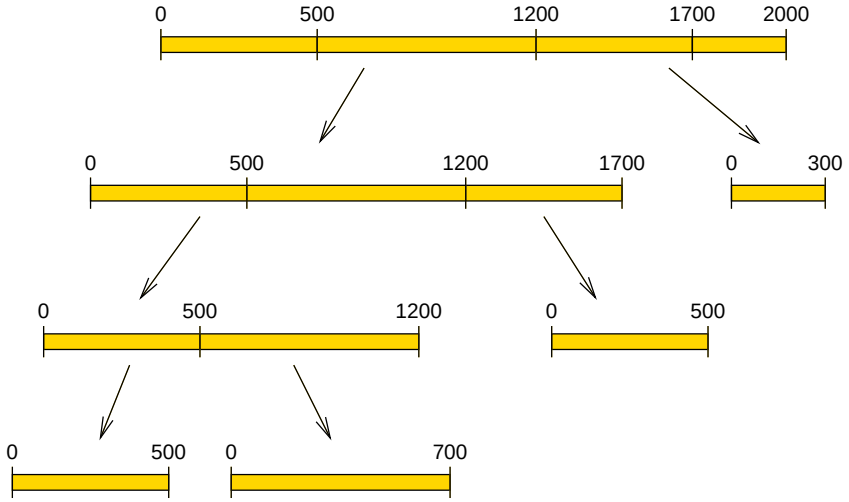
Vstup: Délka tyče l, počet řezů n a pozice jednotlivých řezů. ($n < 50$)

Výstup: Minimální částka, kterou je nutné zaplatit za rozřezání tyče.

Cutting Sticks (10003)



$$2000 + 1200 + 800 = 4000$$



$$2000 + 1700 + 1200 = 4900$$

Adventures in Moving – Part IV (10201)

Nákladní auto se potřebuje dostat z města A do města B, přičemž chceme utratit co nejméně za naftu.
Auto má 200 litrovou nádrž a spotřebu 1 litr nafty na kilometr. Ve městě A obsahuje nádrž 100 litrů nafty, při dojetí do města B musí obsahovat alespoň 100 litrů.

Problém „Adventures in Moving – Part IV“

Vstup:
- Vzdálenost mezi městy A a B v km (max. 10000).
- Informace o n čerpacích stanicích ($n \leq 100$): jejich pozice na cestě a cena za 1 litr nafty.

Výstup: Minimální částka, kterou je třeba zaplatit, nebo informace o tom, že za daných podmínek nelze z města A do města B dojet.

Ferry Loading (10261)

Na trajekt najíždějí auta. Cílem je dostat na trajekt co největší počet aut.

- Auta najíždějí na trajekt v pořadí, ve kterém přijela (tj. pořadí najíždění aut je pevně dáno).
- Auta se řadí do dvou řad. V každé řadě se řadí těsně za sebou (mezi auty nejsou mezery).
- U každého auta můžeme určit, zda má jet do jedné nebo do druhé řady.

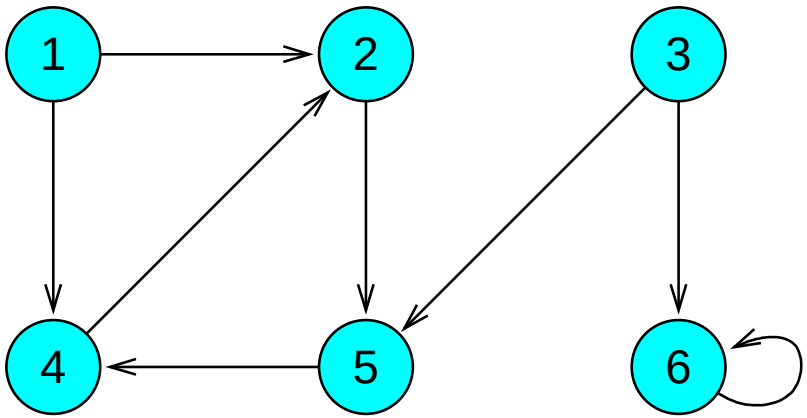
Problém „Ferry Loading“

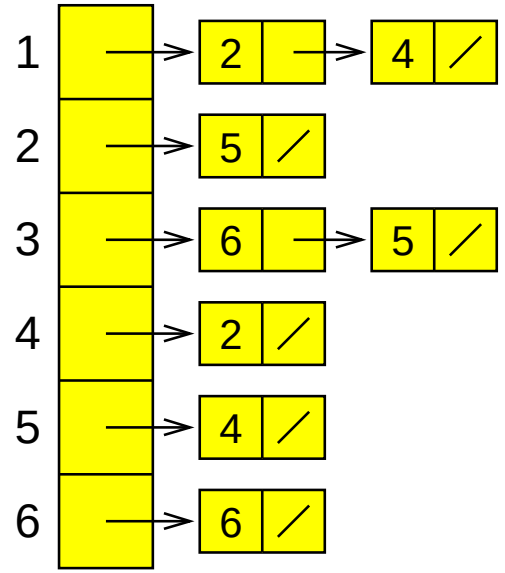
Vstup: Délka lodě v metrech (maximálně 100 m) a délky jednotlivých aut v centimetrech (v rozmezí 100–3000 cm).

Výstup: Instrukce pro jednotlivá auta, zda mají jet vlevo nebo vpravo, které zajistí, že se na trajekt vejde co největší počet aut.

Grafové algoritmy

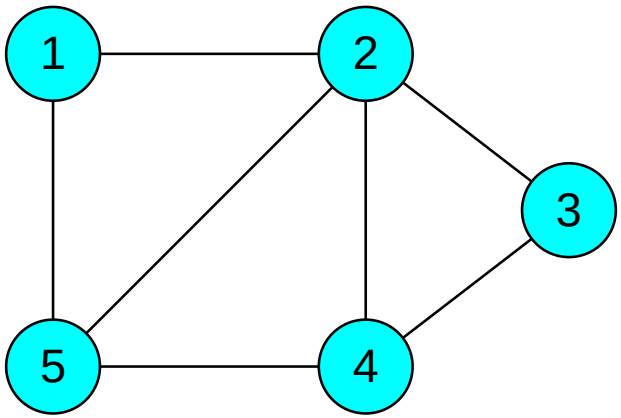
Reprezentace grafu

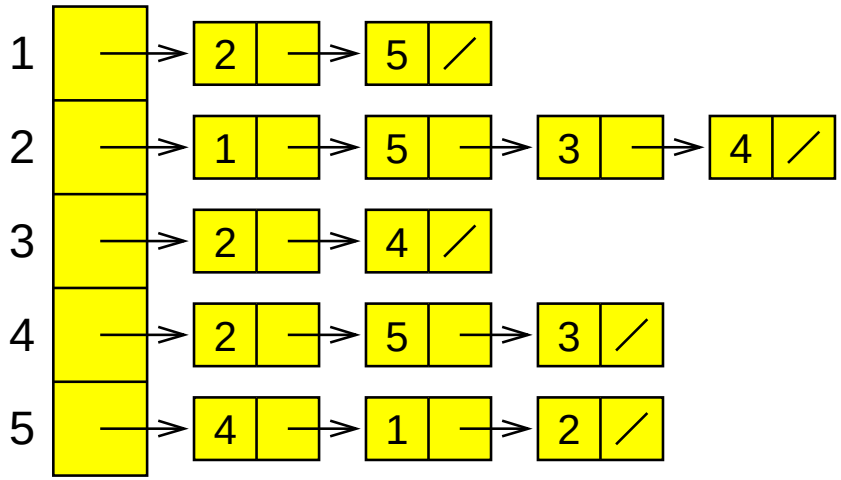




	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Reprezentace grafu





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Prohledávání do šířky

Nalezení nejkratší cesty v grafu, kde hrany nejsou ohodnoceny:

- Algoritmus pro **prohledávání grafu do šířky**
- Vstupem je graf G (s množinou vrcholů V) a počáteční vrchol s.
- Algoritmus pro všechny vrcholy najde nejkratší cestu z vrcholu s.
- Pro graf, který má n vrcholů a m hran je doba výpočtu tohoto algoritmu $\Theta(n + m)$.

Prohledávání do šířky

Algoritmus **prohledávání do šířky** (breadth-first search):

BFS(G, s)

```
1  for each vertex u ∈ V[G] − {s}
2        do color[u] ← WHITE
3           d[u] ← ∞
4           π[u] ← NIL
5  color[s] ← GRAY
6  d[s] ← 0
7  π[s] ← NIL
8  Q ← ∅
9  ENQUEUE(Q, s)
```

Prohledávání do šířky

```
H1 10  while Q ≠ ∅
   11        do u ← DEQUEUE(Q)
   12           for each v ∈ Adj[u]
   13                 do if color[v] = WHITE
   14                       then color[v] ← GRAY
   15                            d[v] ← d[u] + 1
   16                            π[v] ← u
   17                            ENQUEUE(Q, v)
   18           color[u] ← BLACK
```

Prohledávání do šířky

Implementace fronty:

- Operace $Q \leftarrow \emptyset$:

  qstart $= 0$;
  qend $= 0$;

- Operace $\textsc{Enqueue}(Q, u)$:

  queue[qend++] $=$ u;

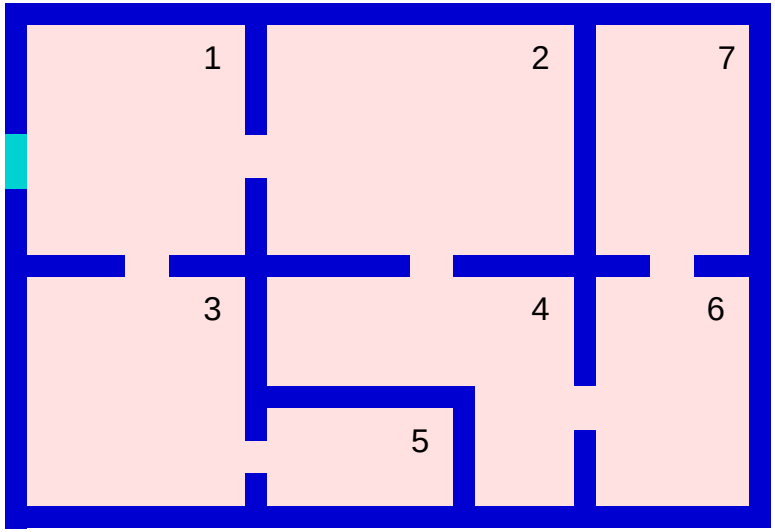
- Operace $u \leftarrow \textsc{Dequeue}(Q)$:

  u $=$ queue[qstart++];

- Test, zda je fronta neprázdná ($Q \neq \emptyset$):

  qstart $<$ qend

The New Villa (321)



- Pan Black se vrací do své vily pozdě v noci, a chce se dostat ze vstupní haly do své ložnice.
- Pan Black se velice bojí tmy a nikdy nevstoupí do místnosti, kde není rožnuté světlo.
- V domě jsou podivně rozmístěny vypínače. Vypínač v jedné místnosti slouží k ovládání světla v nějaké úplně jiné místnosti.

The New Villa (321)

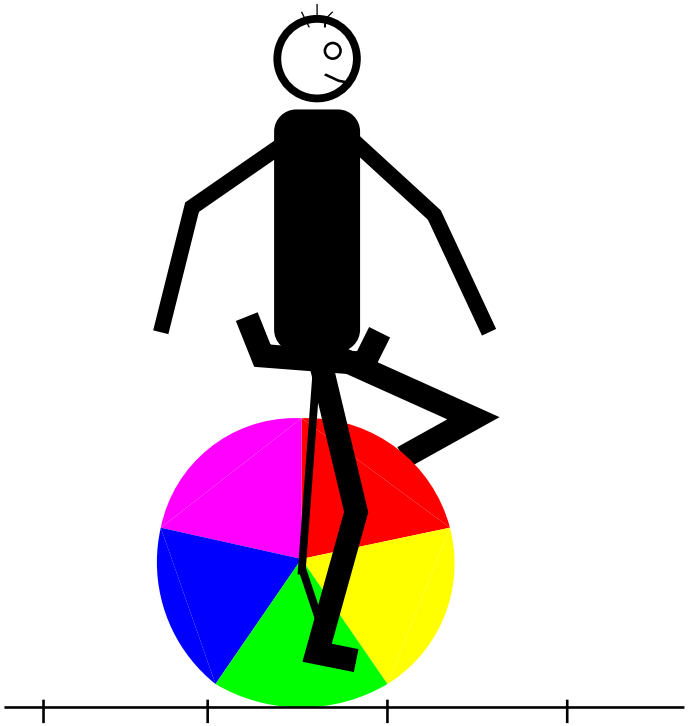
- Na začátku se svítí jen ve vstupní hale a všude jinde je zhasnuto.
- Na konci musí být světlo rožnuto v ložnici a všude jinde musí být zhasnuto.
- Zajímá nás kolik kroků má nejkratší řešení (jeden krok je jeden přechod z jedné místnosti do druhé nebo jedno stisknutí vypínače).

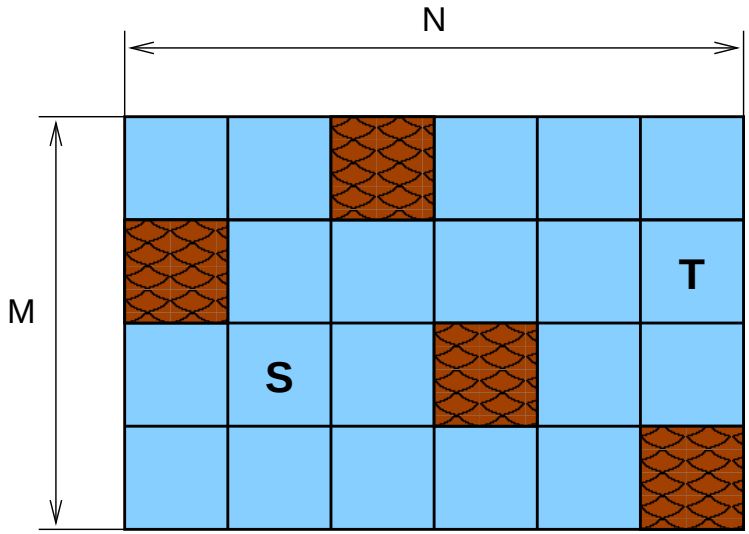
Problém:

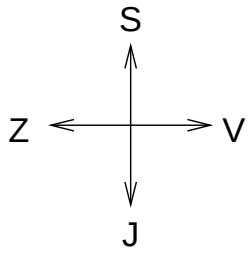
Vstup: Počet místností N ($1 \leq N \leq 10$, místnost 1 je vstupní hala, místnost N je ložnice),
seznam všech dveří (informace, které místnosti spojují),
seznam všech vypínačů (informace, ve které místnosti se vypínač nachází a svělo které místnosti ovládá).

Výstup: Počet kroků nejkratšího řešení nebo informace, že řešení neexistuje.

The Monocycle (10047)







Artista jezdí na jednokolce po ploše tvořené kachličkami.
Velikost jedné kachličky přesně odpovídá $1/5$ obvodu kola.

The Monocycle (10047)

- Přejetí z jedné kachličky na druhou trvá 1 sekundu.
- Otočení se o 90° trvá 1 sekundu.
- Artista začíná na kachličce S, které se dotýká zelenou částí kola. Na začátku je natočený směrem na sever.
- Cílem je dostat se na kachličku T tak, aby se jí dotýkal zelenou částí kola, přičemž na natočení nezáleží.

Problém

Vstup: Popis plochy, počáteční pozice S a cílová pozice T.

Výstup: Minimální čas, za který se může artista dostat z pozice S na pozici T při dodržení výše uvedených podmínek, nebo informace, že řešení neexistuje.

Hang or not to hang (CEPC 2003)

Máme program skládající se z n instrukcí ($1 \leq n \leq 16$) a pracující s pamětí tvořenou 32 jednobitovými buňkami označenými $\mathrm{MEM}[0] \,..\, \mathrm{MEM}[31]$:

Instrukce	Sémantika
AND a b	$\mathrm{MEM}[a] := \mathrm{MEM}[a]$ and $\mathrm{MEM}[b]$
OR a b	$\mathrm{MEM}[a] := \mathrm{MEM}[a]$ or $\mathrm{MEM}[b]$
XOR a b	$\mathrm{MEM}[a] := \mathrm{MEM}[a]$ xor $\mathrm{MEM}[b]$
NOT a	$\mathrm{MEM}[a] :=$ not $\mathrm{MEM}[a]$
MOV a b	$\mathrm{MEM}[a] := \mathrm{MEM}[b]$
SET a c	$\mathrm{MEM}[a] := c$
RANDOM a	$\mathrm{MEM}[a] :=$ random value (0 nebo 1)
JMP x	skok na instrukci číslo x
JZ x a	skok na instrukci číslo x, pokud $\mathrm{MEM}[a] = 0$
STOP	ukončení programu

$0 \leq a, b < 32$, $0 \leq x < n$, $c \in \{0, 1\}$

Hang or not to hang (CEPC 2003)

Problém

Vstup: Kód programu

Výstup: Nejmenší počet instrukcí, které program provede než se zastaví nebo informace, že program se nikdy nezastaví.

Poznámka: Poslední instrukce programu je vždy instrukce STOP.

Problém „Bicoloring“ (10004)

Problém

Vstup: Neorientovaný graf $G = (V, E)$ (souvislý).

Otázka: Je možné graf G obarvit dvěma barvami?
(Je možné každému vrcholu přiřadit jednu z dvojice barev tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou?)

Chceme najít řešení s časovou složitostí $O(|V| + |E|)$.

Nalezení dvou nejvzdálenějších vrcholů ve stromu

Problém

Vstup: Neorientovaný souvislý acyklický graf (strom).

Výstup: Dvojice vrcholů u, v taková, že vzdálenost z u do v je maximální.

Chceme najít algoritmus s časovou složitostí $O(n)$, kde n je počet vrcholů grafu.

Nalezení dvou nejvzdálenějších vrcholů ve stromu

Řešení:

1. Zvol libovolný list A.
2. Najdi nejvzdálenější vrchol od A, označme ho B.
3. Najdi nejvzdálenější vrchol od B, označme ho C.
4. Výstupem je dvojice B, C.

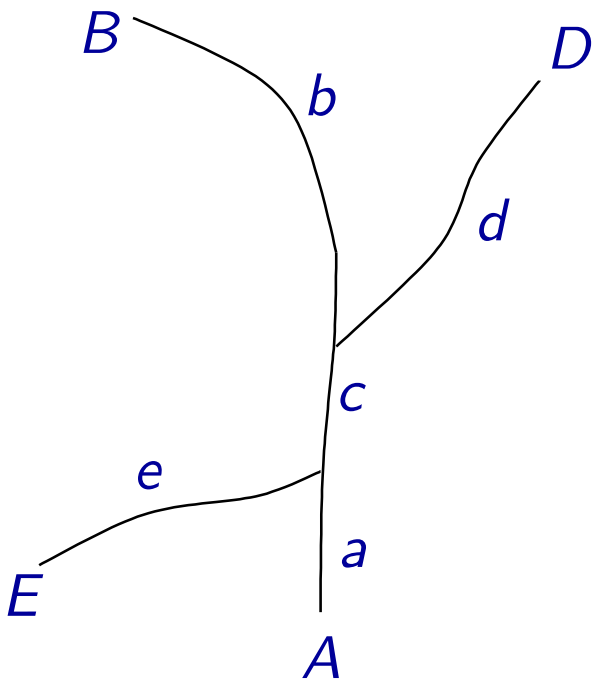
Poznámka: B i C jsou určitě listy.

Nalezení dvou nejvzdálenějších vrcholů ve stromu

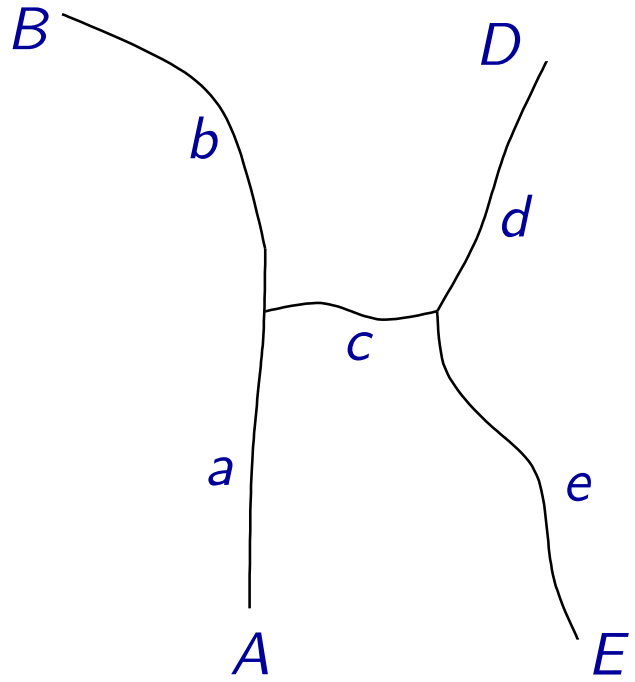
Korektnost řešení:

Předpokládejme, že D, E jsou dva nejvzdálenější vrcholy.

Stačí ukázat, že existuje vrchol F takový, že $d(B, F) \geq d(D, E)$.



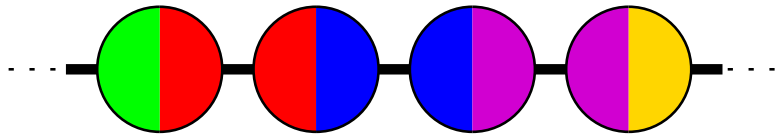
$b + c + e \geq d + c + e$



$b + c + e > d + e$

$(b \geq c + d)$

Problém „The Necklace“ (10054)



- Máme hromadu dvoubarevných korálků.
- Chceme z nich udělat náhrdelník tak, že sousední korálky se budou navzájem dotýkat stejnou barvou.
- Musíme použít všechny korálky, žádný nesmí zbýt.

Problém

Vstup: Seznam korálků (barvy jsou označeny čísly 1 až 50).

Výstup: Pořadí, ve kterém máme navléct korálky na nit, nebo informace, že řešení neexistuje.

Prohledávání do hloubky

Prohledávání do hloubky je základem celé řady grafových algoritmů, které zjišťují nějaké informace o struktuře grafu.
Časová složitost těchto algoritmů je většinou stejná jako u prohledávání do hloubky, tj. $O(|V| + |E|)$.

Využitím prohledávání do hloubky lze v čase $O(|V| + |E|)$ řešit například následující problémy:

- zjištění, zda graf obsahuje cyklus
- topologické uspořádání orientovaného grafu
- nalezení silně souvislých komponent orientovaného grafu
- nalezení artikulací v neorientovaném grafu
- nalezení mostů v neorientovaném grafu
- nalezení 2-souvislých komponent v neorientovaném grafu

Prohledávání do hloubky

Algoritmus **prohledávání do hloubky** (depth-first search):

```
DFS(G)
1  for each vertex u ∈ V[G]
2        do color[u] ← WHITE
3           π[u] ← NIL
4  time ← 0
5  for each vertex u ∈ V[G]
6        do if color[u] = WHITE
7              then DFS-Visit(u)
```

Prohledávání do hloubky

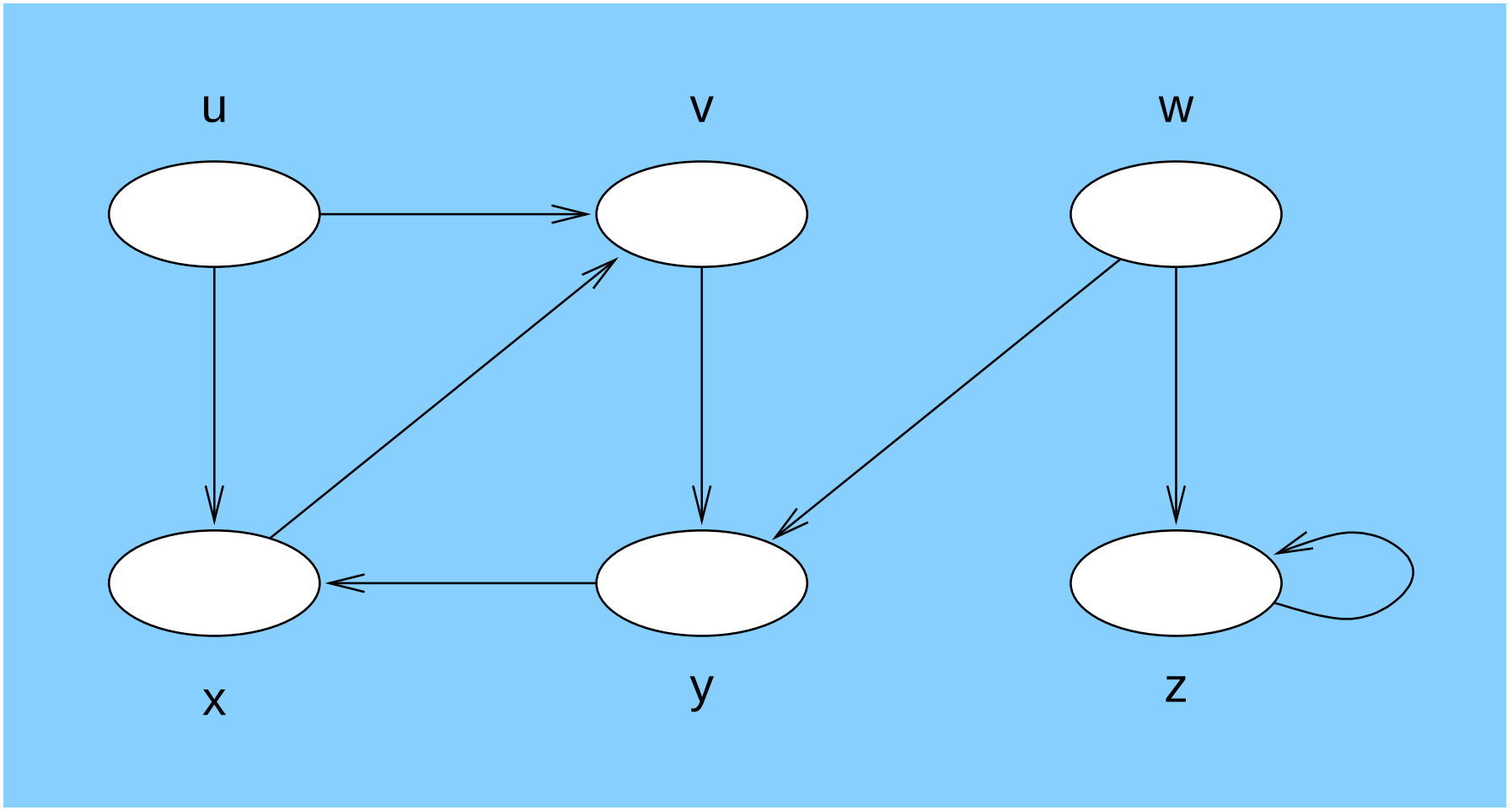
DFS-Visit(u)

```
1  color[u] ← GRAY                ▷ Bílý vrchol u byl právě objeven.
2  d[u] ← time ← time +1
3  for each v ∈ Adj[u]            ▷ Prozkoumání hrany (u, v).
4        do if color[v] = WHITE
5              then π[v] ← u
6                   DFS-Visit(v)
7  color[u] ← BLACK               ▷ Obarvení u na černo; byl dokončen.
8  f[u] ← time ← time +1
```

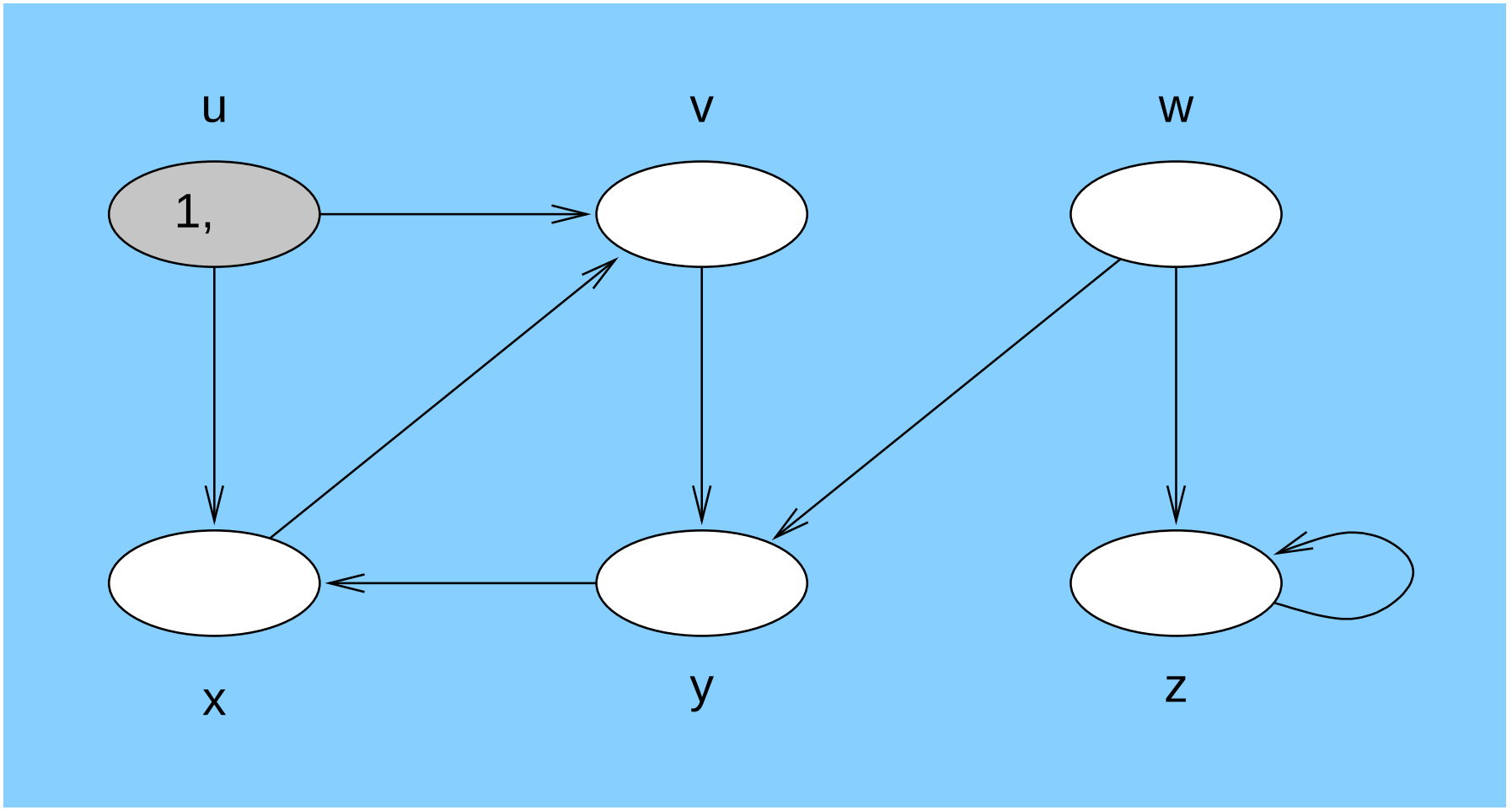
$d[u]$ – čas, kdy byl vrchol u poprvé objeven (a obarven šedě)

$f[u]$ – čas, kdy byl vrchol u dokončen (a obarven černě)

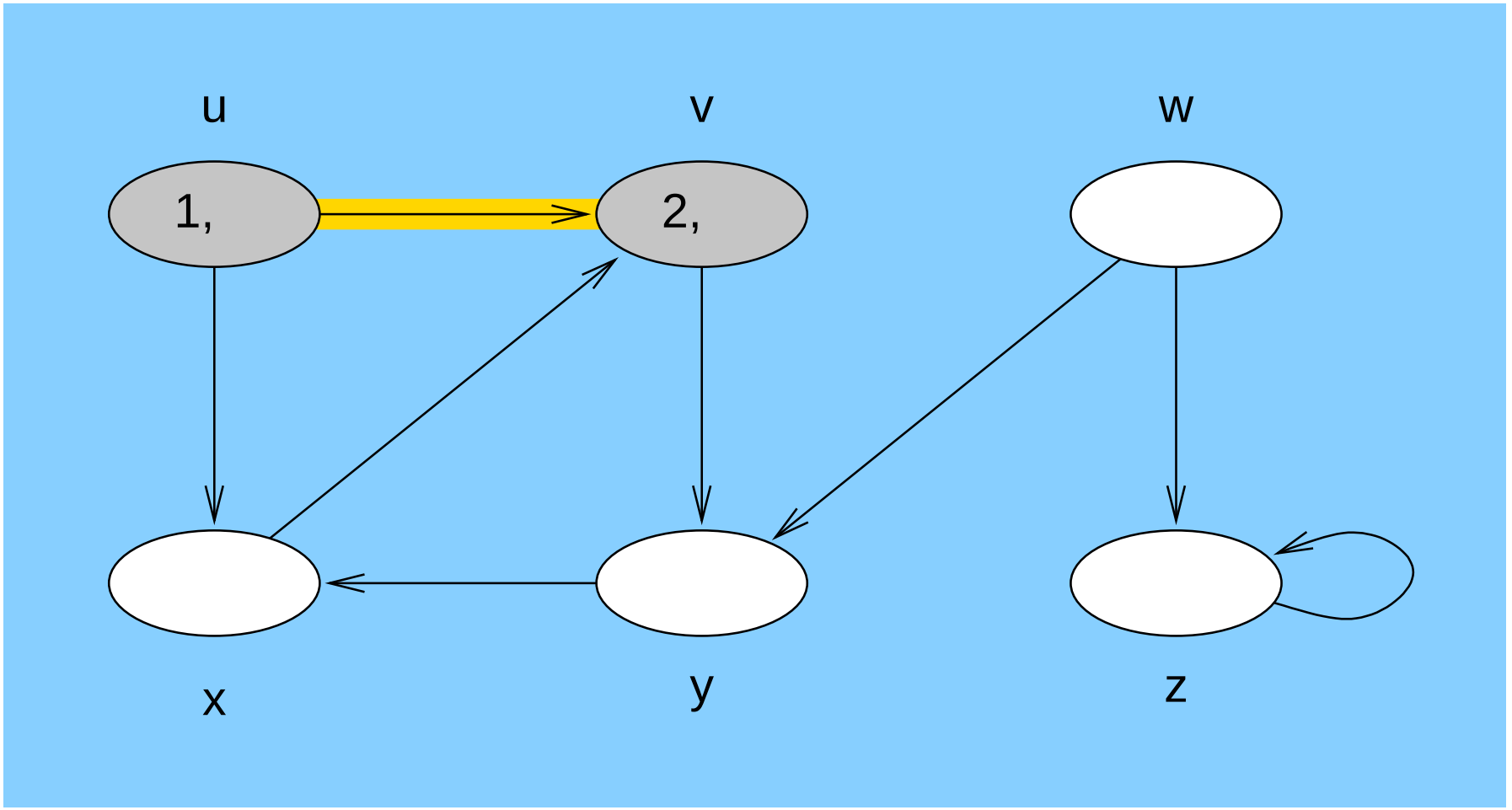
Prohledávání do hloubky



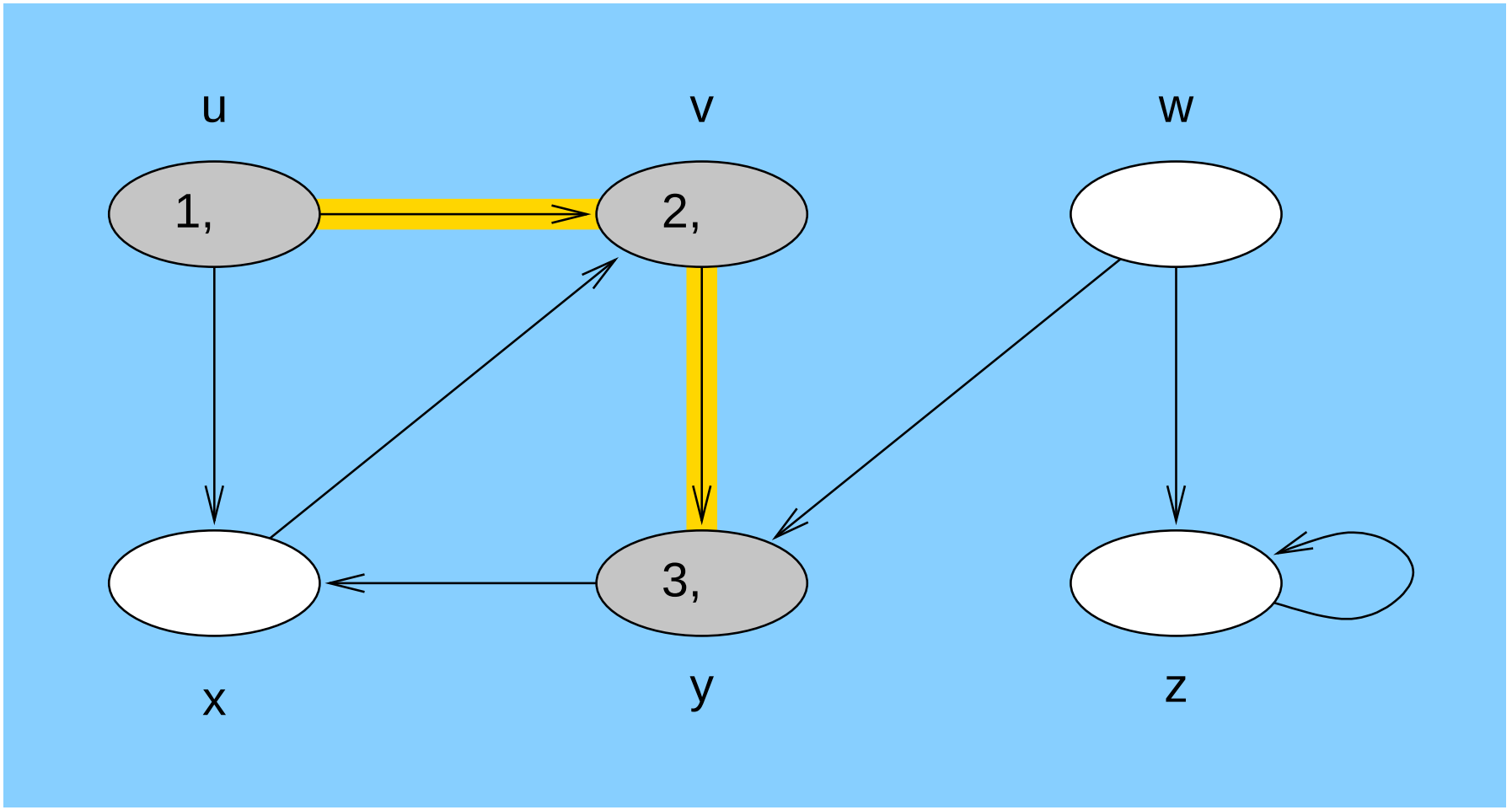
Prohledávání do hloubky



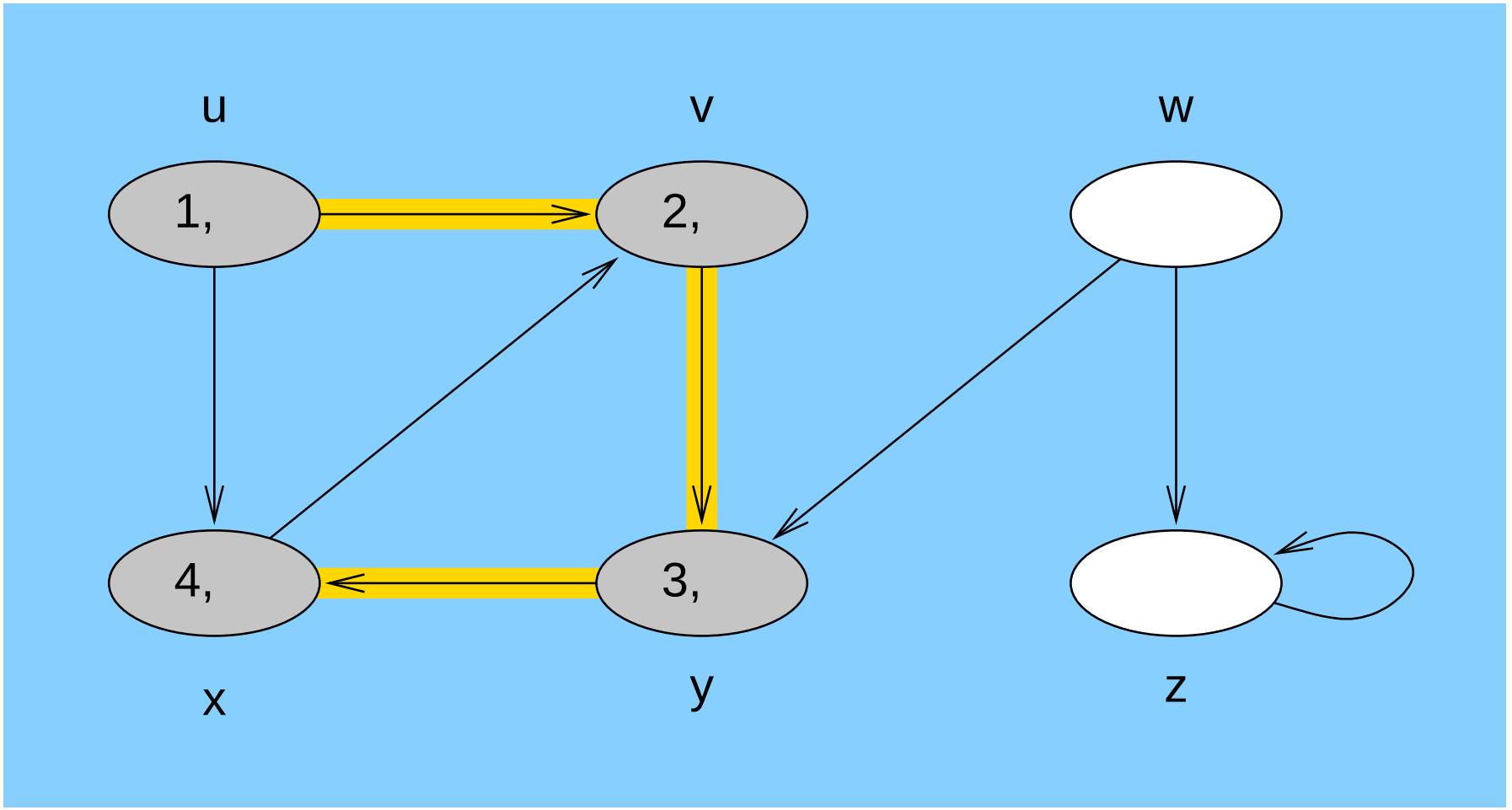
Prohledávání do hloubky



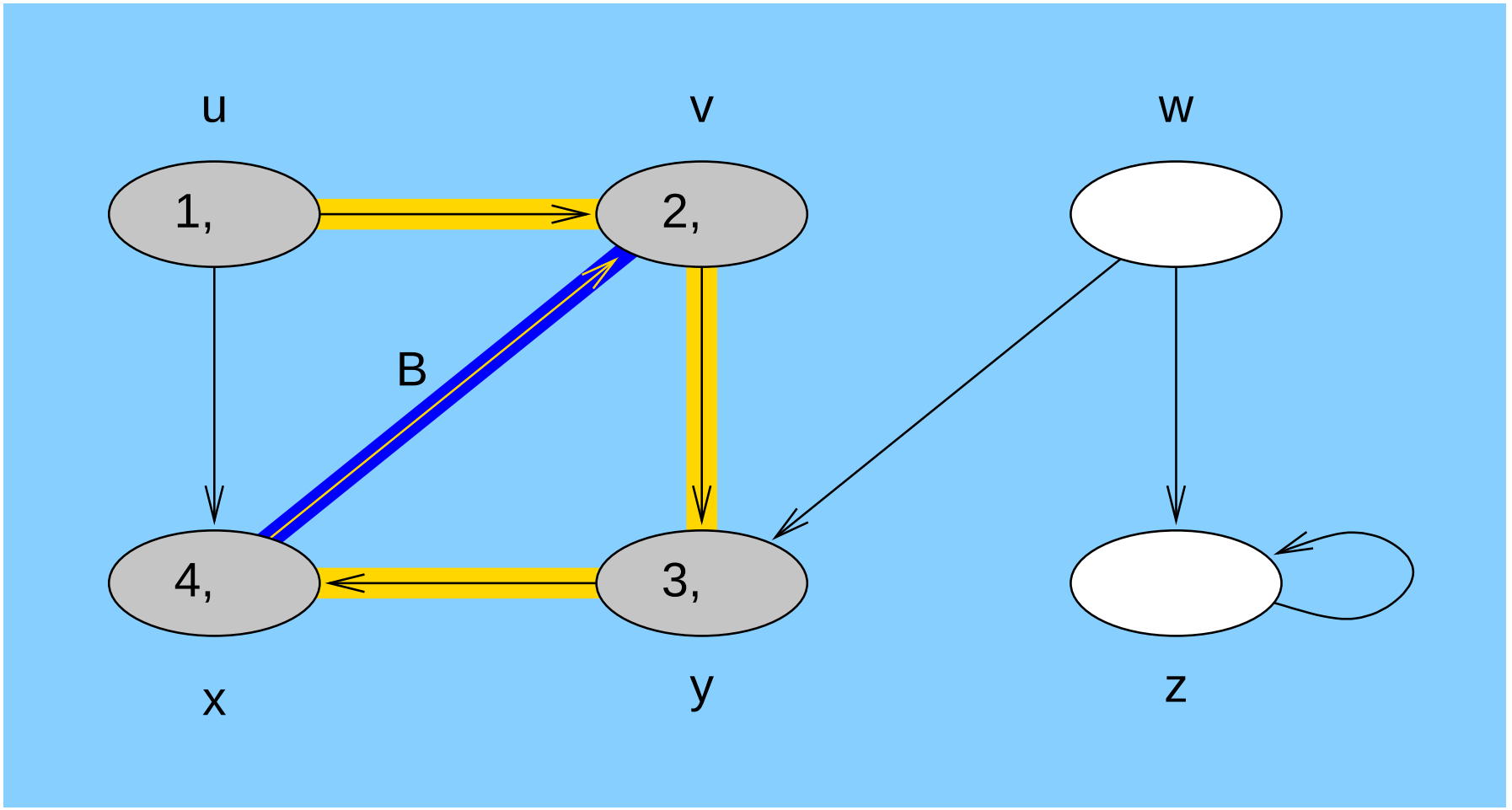
Prohledávání do hloubky



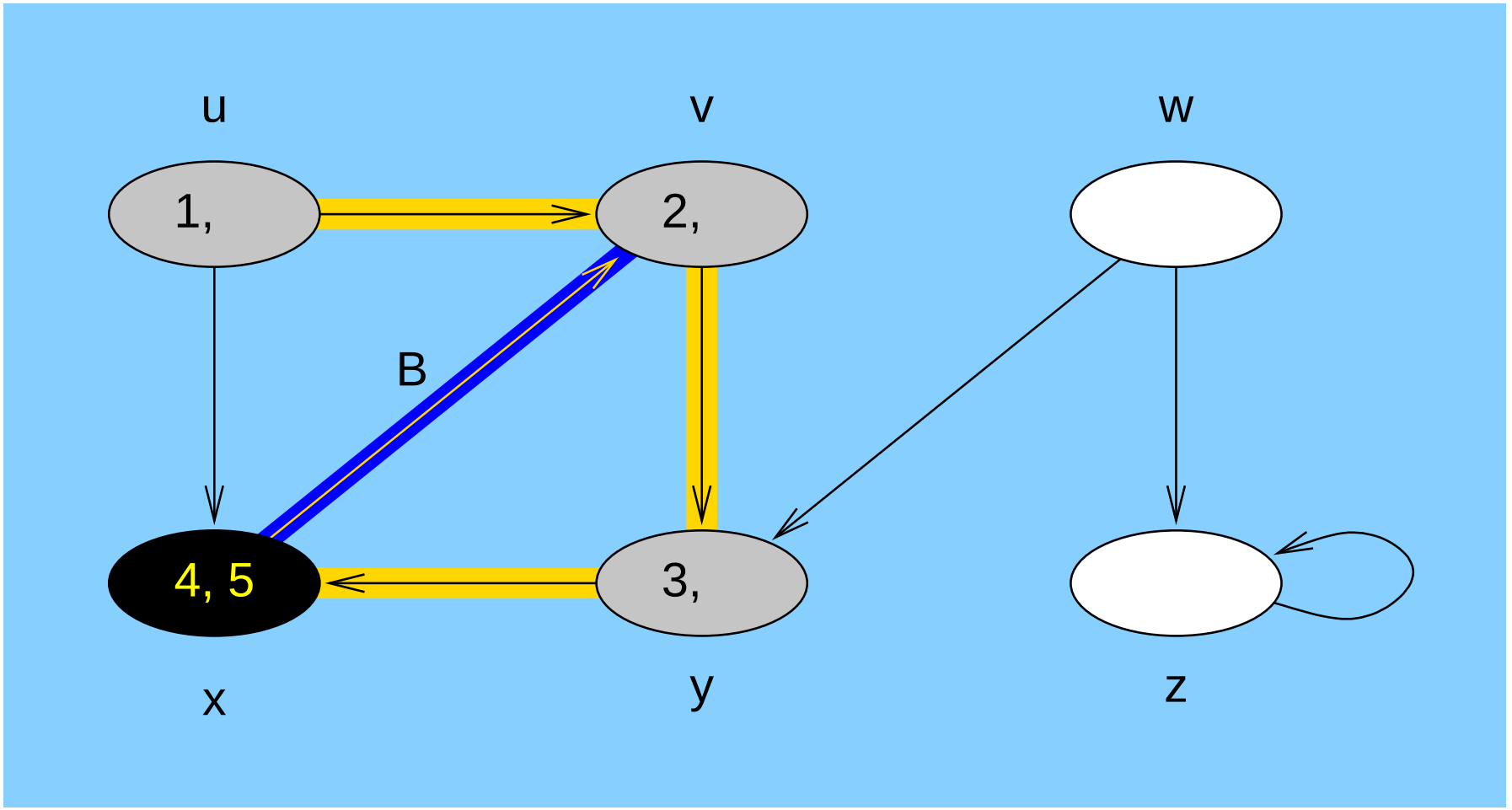
Prohledávání do hloubky



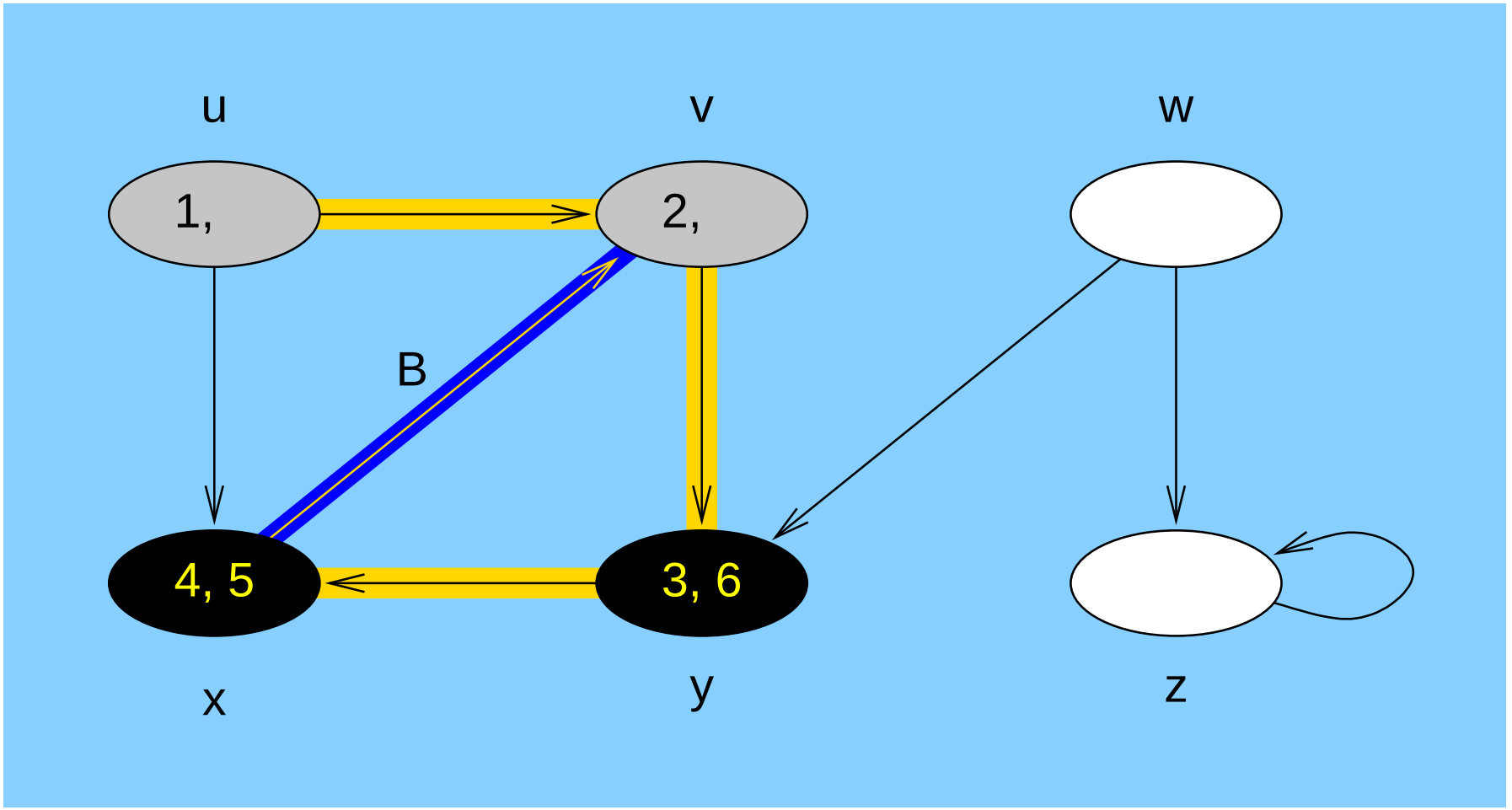
Prohledávání do hloubky



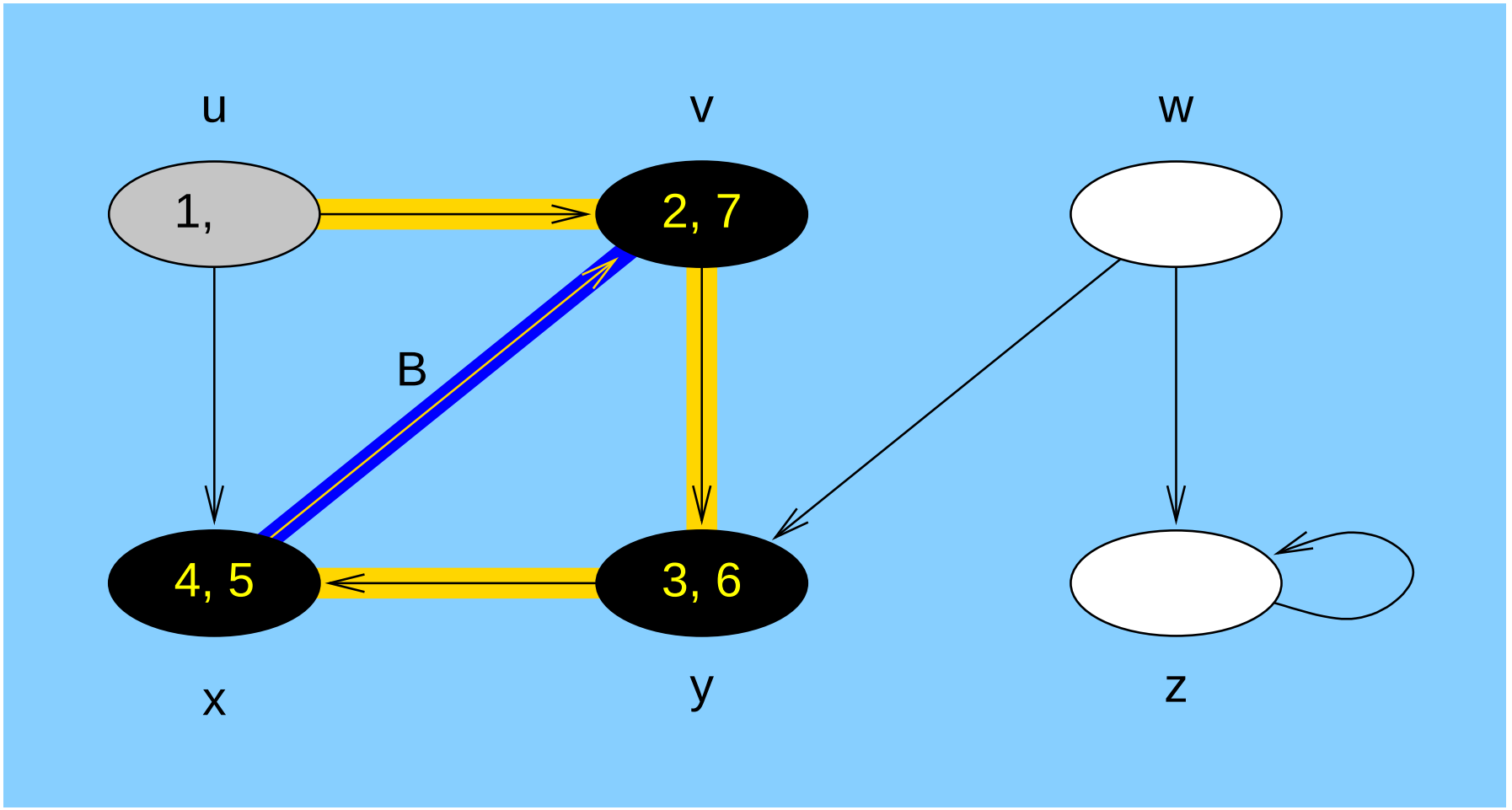
Prohledávání do hloubky



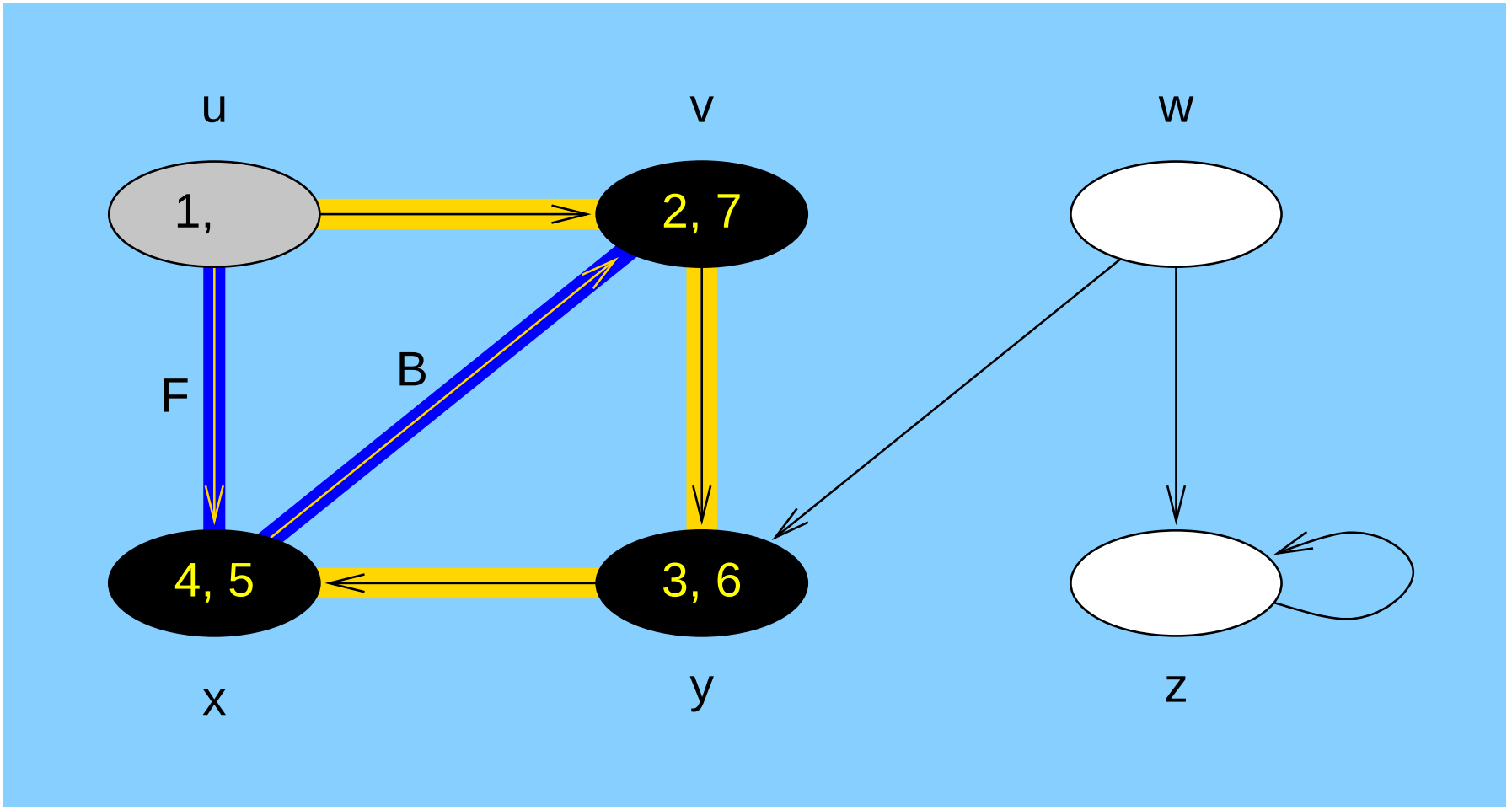
Prohledávání do hloubky



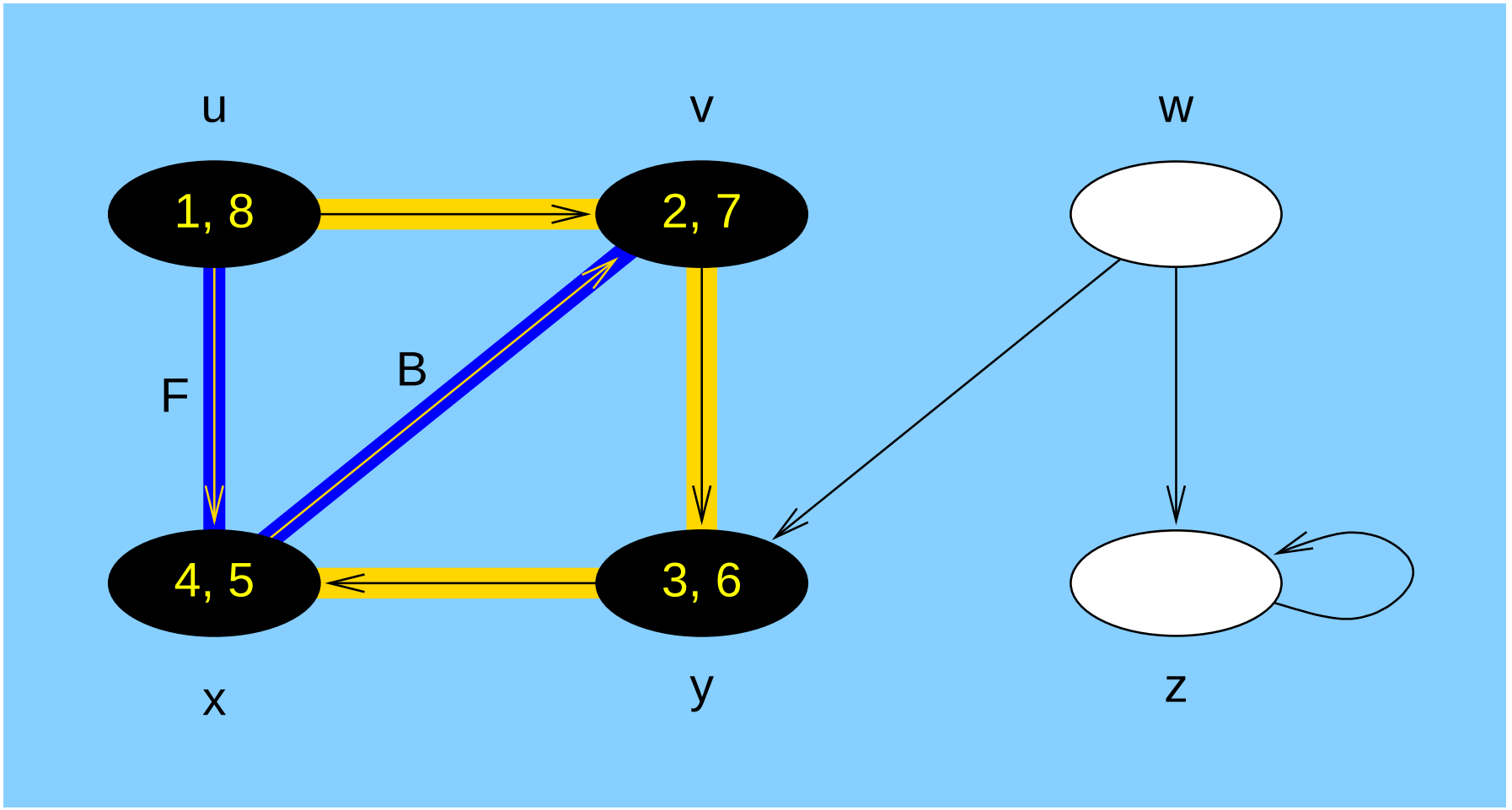
Prohledávání do hloubky



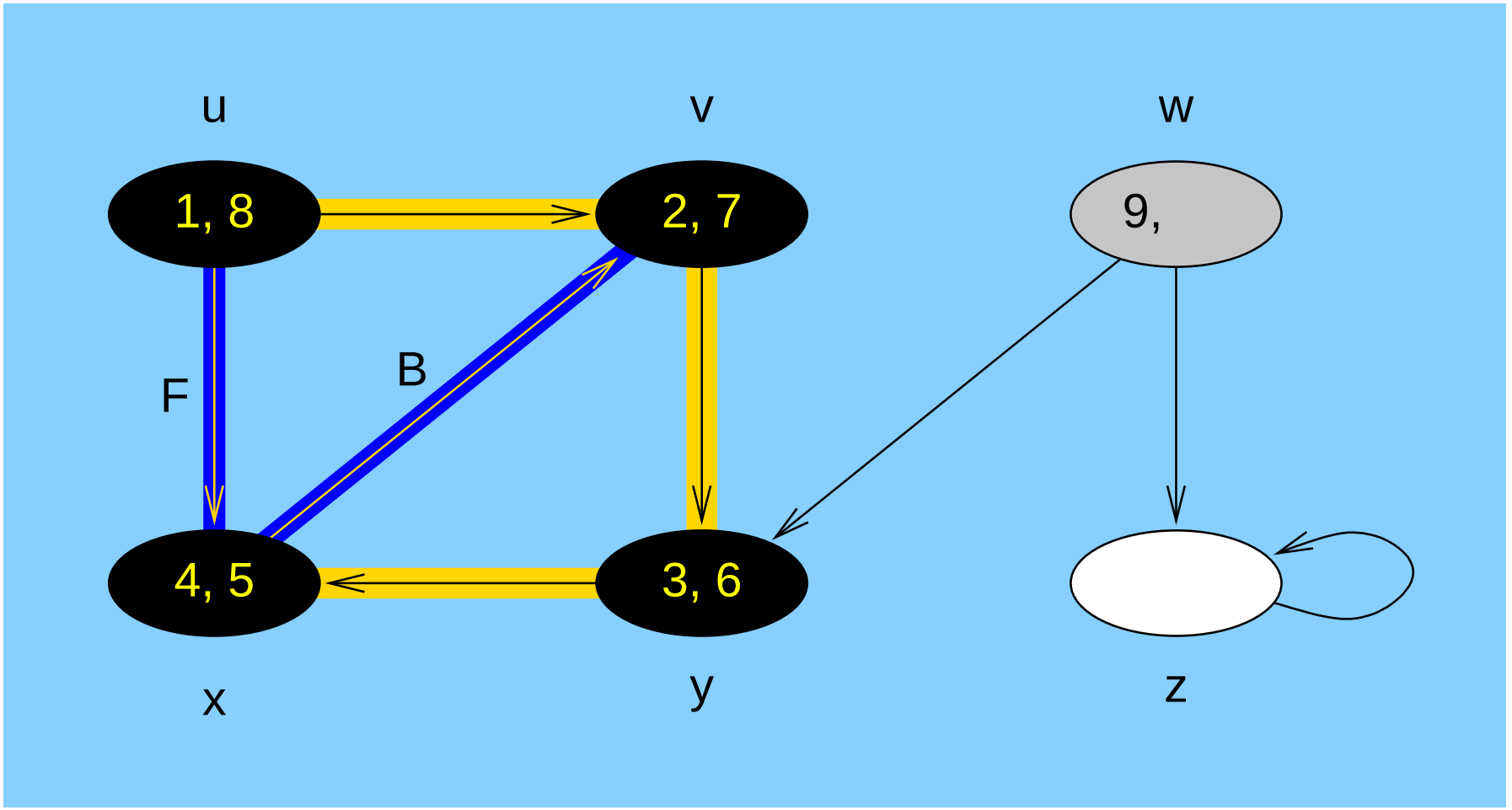
Prohledávání do hloubky



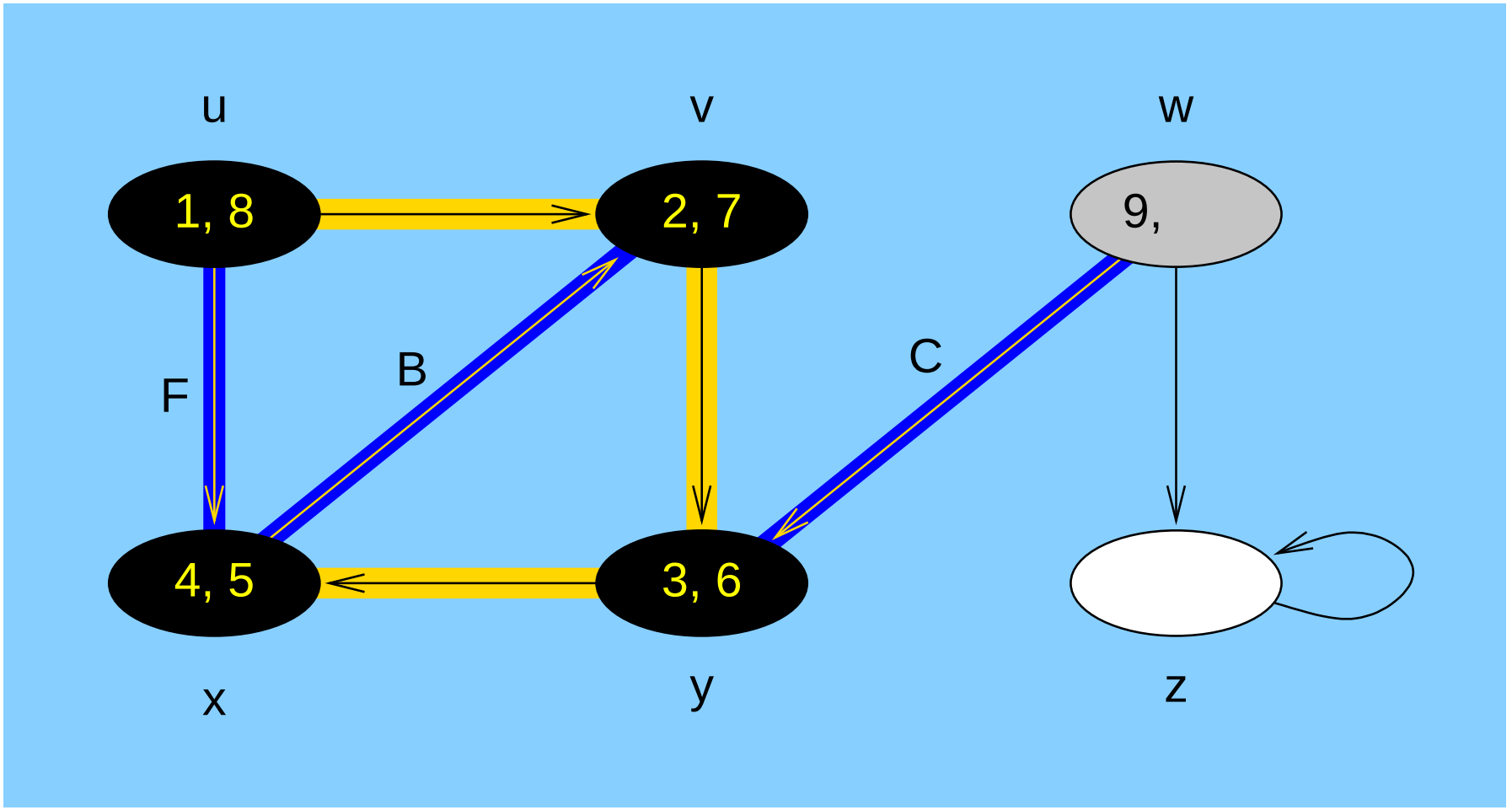
Prohledávání do hloubky



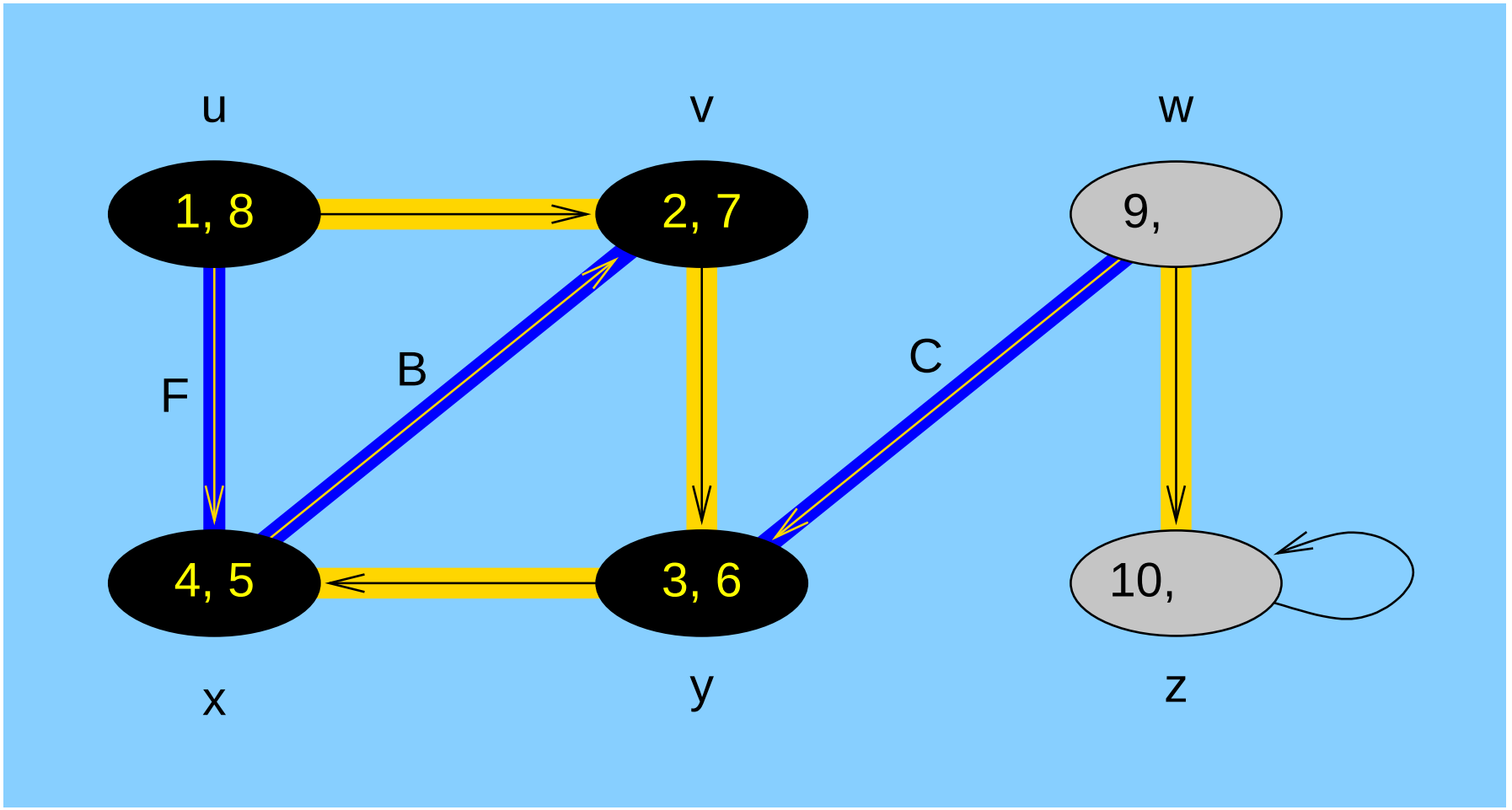
Prohledávání do hloubky



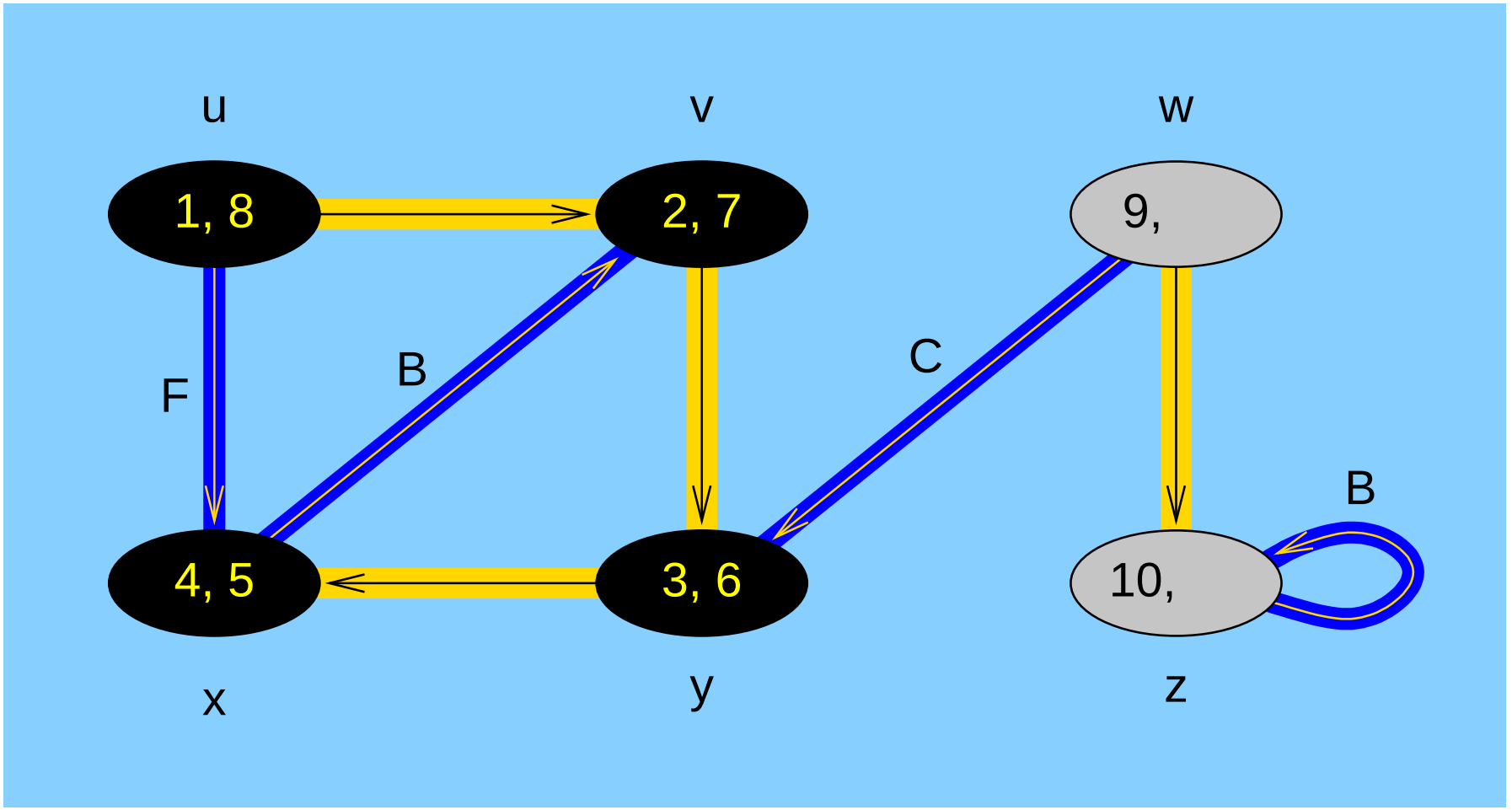
Prohledávání do hloubky



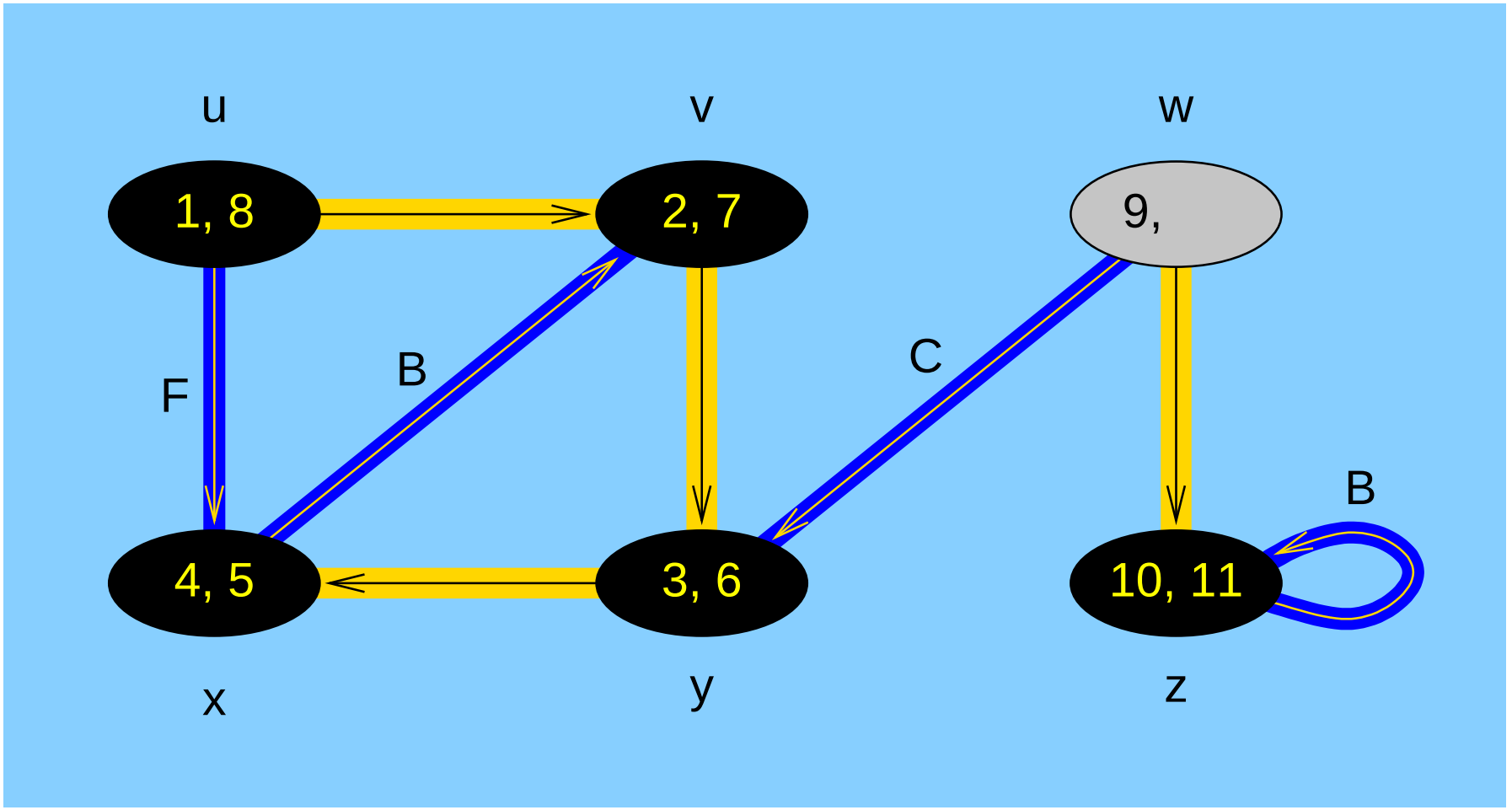
Prohledávání do hloubky



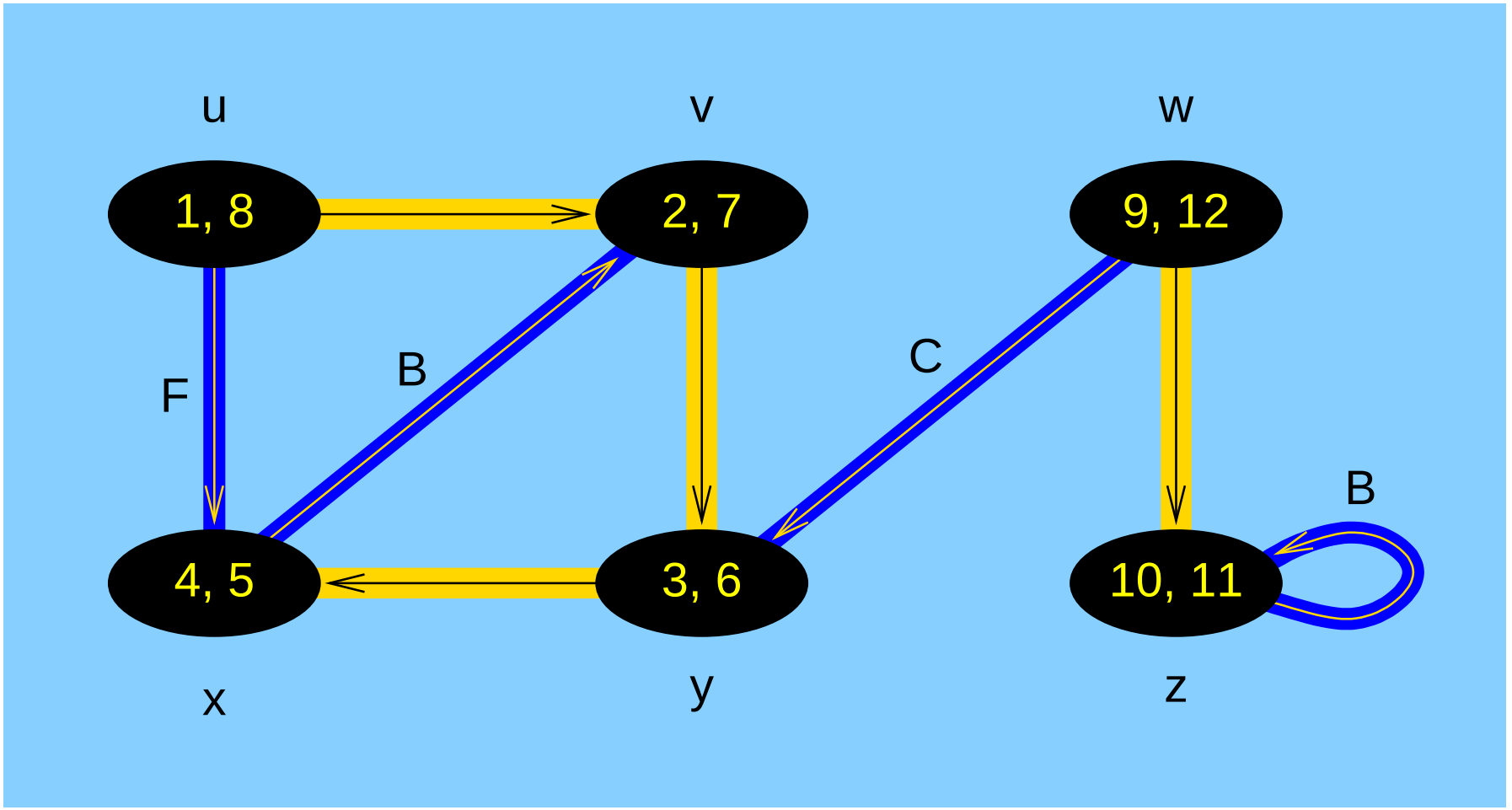
Prohledávání do hloubky



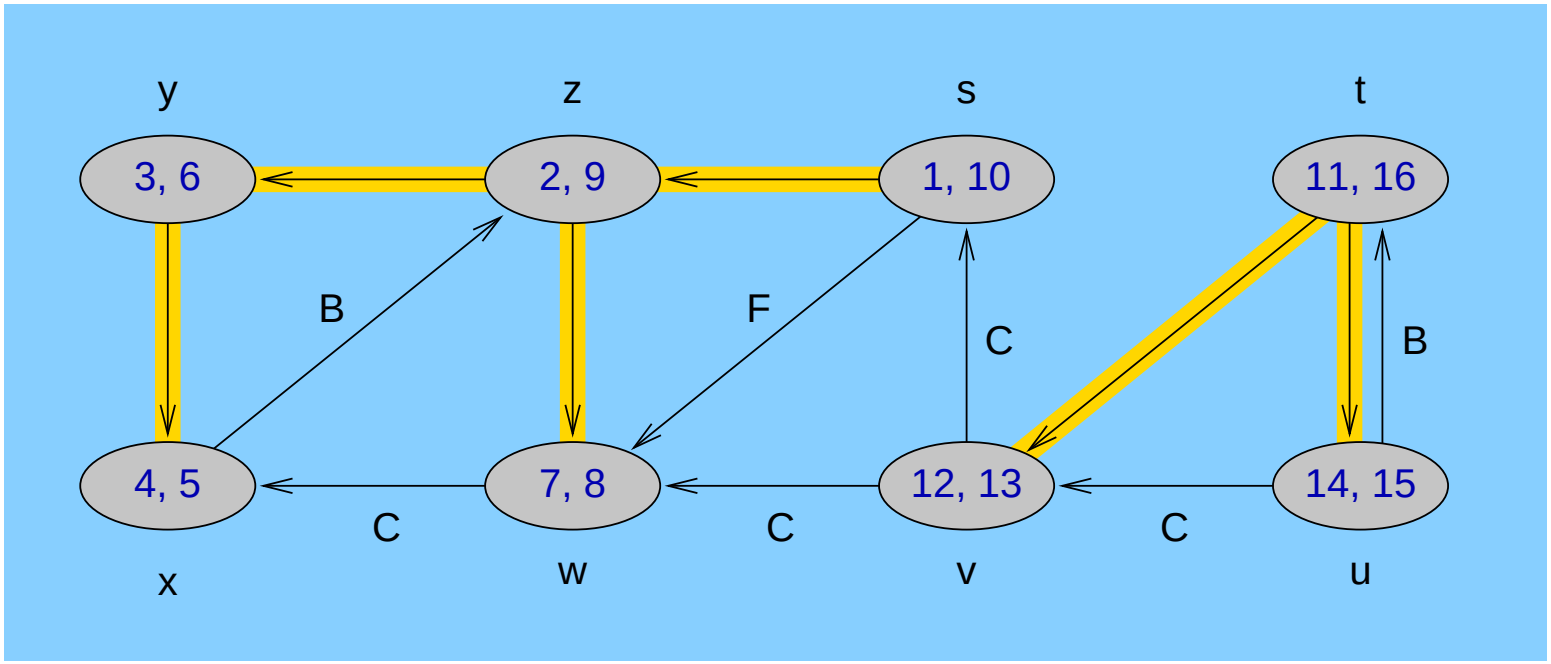
Prohledávání do hloubky



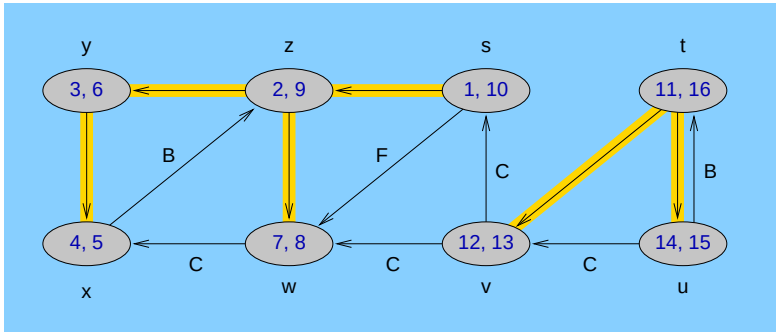
Prohledávání do hloubky

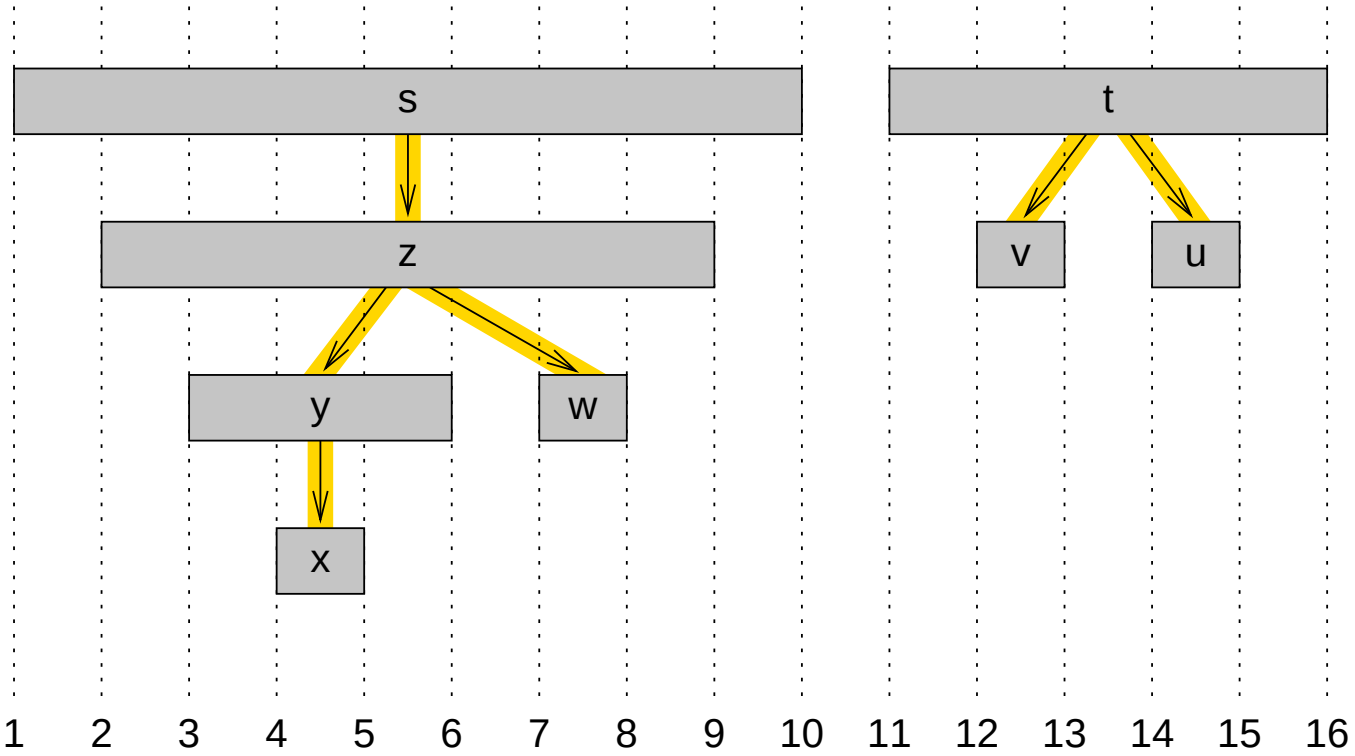


Prohledávání do hloubky

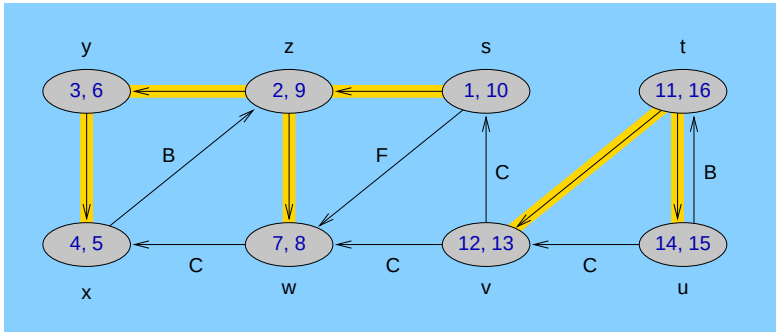


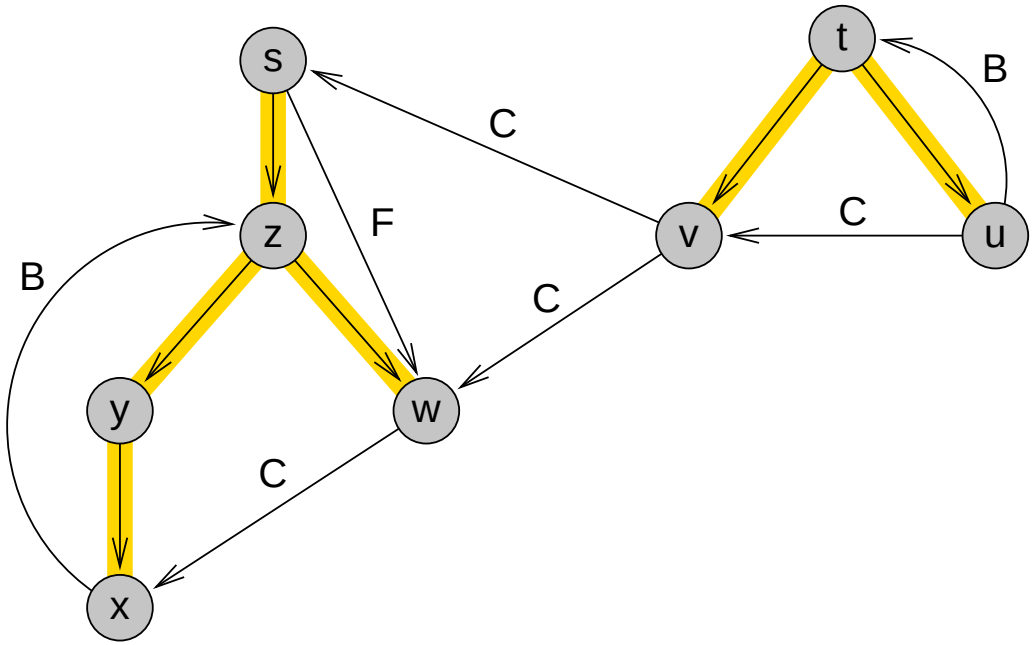
Prohledávání do hloubky





Prohledávání do hloubky





Prohledávání do hloubky

Označme $I(u)$ interval příslušející vrcholu u, tj. od $d[u]$ do $f[u]$.

Pozorování

Pro libovolné dva vrcholy u a v vždy platí právě jedná z podmínek:

- intervaly $I(u)$ a $I(v)$ jsou disjunktní a u není potovkem v ani v není potomkem u, nebo
- interval $I(u)$ je podintervalem intervalu $I(v)$ a u je potomkem v, nebo
- interval $I(v)$ je podintervalem intervalu $I(u)$ a v je potomkem u.

Důsledek

Vrchol v je potomkem vrcholu u právě když $d[u] < d[v] < f[v] < f[u]$.

Prohledávání do hloubky

Pozorování

Šedé vrcholy tvoří cestu $v_1, v_2, \ldots, v_k$, kde pro $i < j$ platí, že

$$d[v_i] < d[v_j] < f[v_j] < f[v_i]$$

Věta (White-path theorem)

Vrchol v je potomkem vrcholu u právě tehdy, když v čase $d[u]$ platí, že vrchol v je dosažitelný z u po cestě tvořené pouze bílými vrcholy.

Prohledávání do hloubky

Klasifikace hran:

- **Stromové hrany (tree edges)** – při průchodu těmito hranami byl objeven nový vrchol
- **Zpětné hrany (back edges)** – hrany z potomků do předchůdců
- **Dopředné hrany (forward edges)** – hrany z předchůdců do potomků, které nejsou stromovými hranami
- **Příčné hrany (cross edges)** – všechny zbývající hrany

Prohledávání do hloubky

Při průchodu hranou (u, v) lze její typ určit podle barvy, kterou je obarven vrchol v:

- WHITE – stromová hrana
- GRAY – zpětná hrana
- BLACK – dopředná (pokud $d[u] < d[v]$) nebo příčná (pokud $d[u] > d[v]$)

Prohledávání do hloubky

Při průchodu hranou (u, v) lze její typ určit podle barvy, kterou je obarven vrchol v:

- WHITE – stromová hrana
- GRAY – zpětná hrana
- BLACK – dopředná (pokud $d[u] < d[v]$) nebo příčná (pokud $d[u] > d[v]$)

Poznámka

V neorientovaném grafu je typ hrany určen při **prvním** průchodu hranou.

Pozorování

Neorientovaný graf obsahuje pouze stromové a zpětné hrany.

Prohledávání do hloubky

Problém

Vstup: Orientovaný graf G.

Otázka: Obsahuje graf G cyklus?

Prohledávání do hloubky

Problém

Vstup: Orientovaný graf G.

Otázka: Obsahuje graf G cyklus?

Věta

Graf obsahuje cyklus právě tehdy, když algoritmus prohledávání do hloubky najde alespoň jednu zpětnou hranu.

Prohledávání do hloubky

Problém

Vstup: Orientovaný graf G.

Otázka: Obsahuje graf G cyklus?

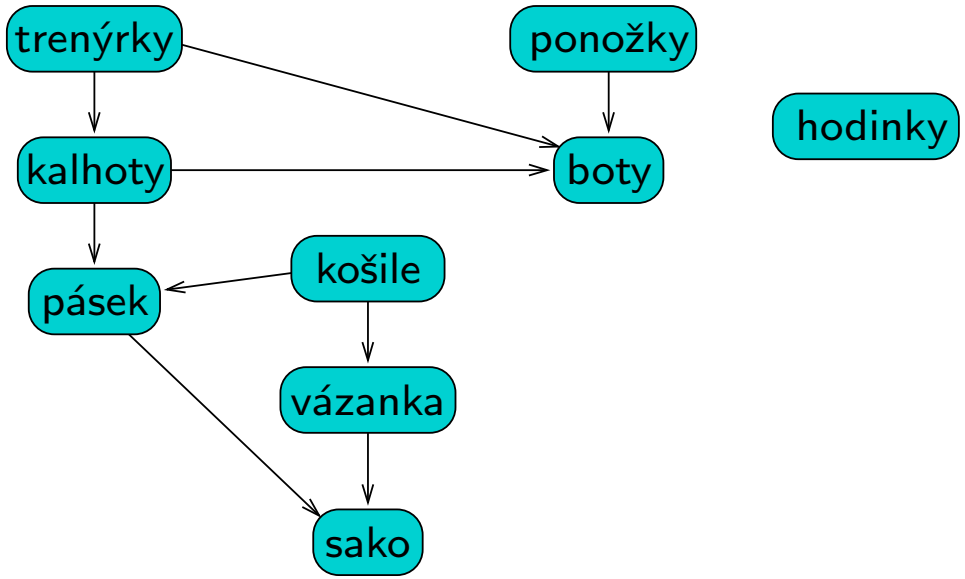
Věta

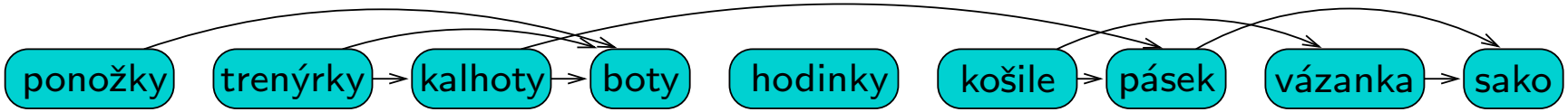
Graf obsahuje cyklus právě tehdy, když algoritmus prohledávání do hloubky najde alespoň jednu zpětnou hranu.

Důkaz: Je zřejmé, že pokud graf obsahuje zpětnou hranu, pak určitě obsahuje cyklus.

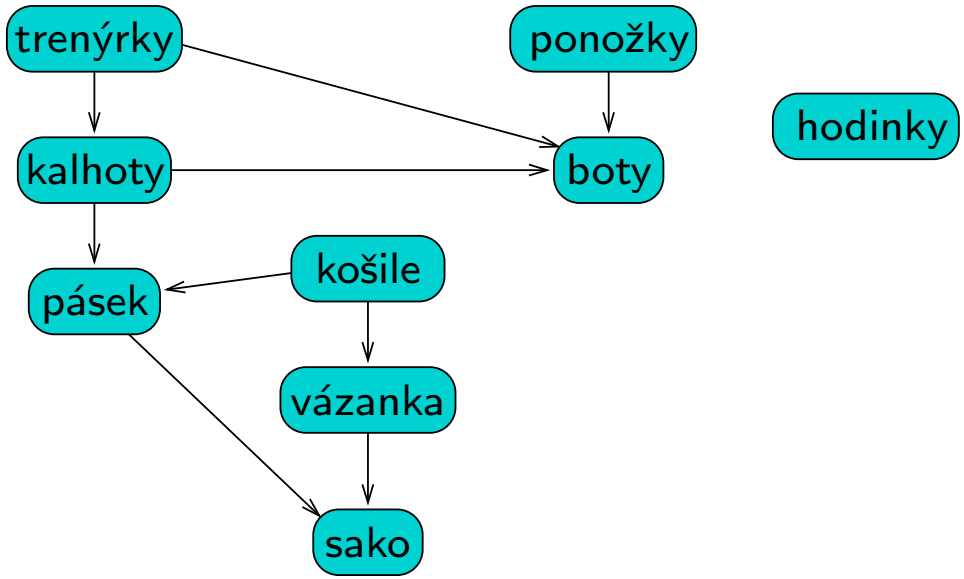
Pro libovolnou hranu (u, v), která není zpětná (tj. je stromová, dopředná nebo příčná), platí $f[u] > f[v]$ (naopak pro zpětnou platí $f[u] < f[v]$). Pokud by v grafu, ve kterém neexistuje žádná zpětná hrana, existoval cyklus, pro všechny hrany (u, v) na tomto cyklu by muselo platit $f[u] > f[v]$, to ale není možné.

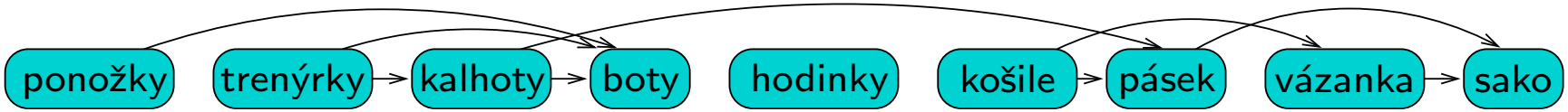
Topologické uspořádání grafu





Topologické uspořádání grafu





Postup: Při prohledávání do hloubky přidat na začátek seznamu vrchol v okamžiku, kdy je obarven na černo.

Topologické uspořádání grafu

Problém

Vstup: Acyklický orientovaný graf a dva jeho vrcholy s a t.

Výstup: Počet všech cest z s do t.

Silně souvislé komponenty

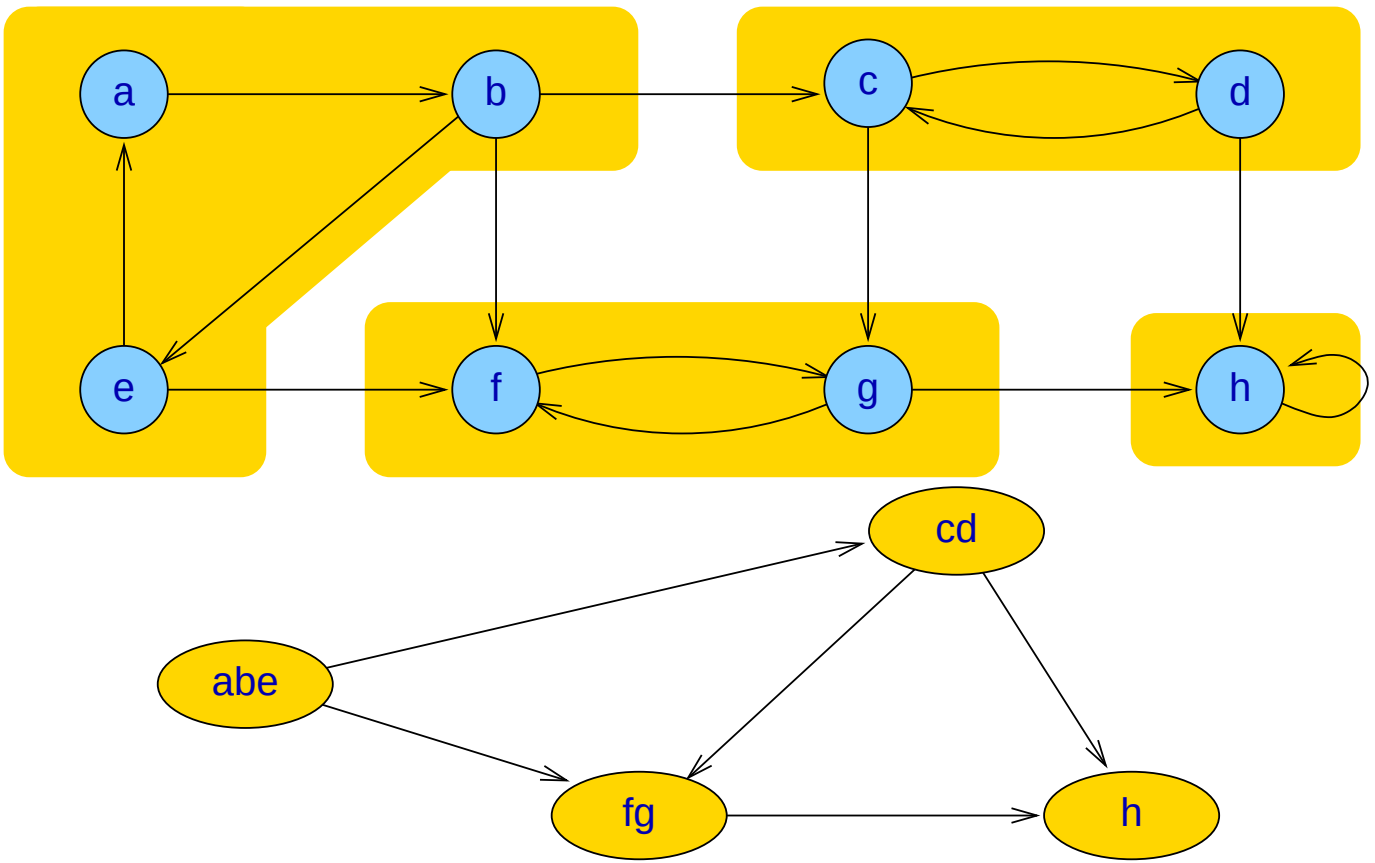
Definice

Silně souvislá komponenta grafu $G = (V, E)$ je maximální množina vrcholů $C \subseteq V$ taková, že pro libovolné dva vrcholy $u, v \in C$ existuje cesta z u do v i cesta z v do u.

Poznámka: Silně souvislé komponenty tvoří rozklad na množině vrcholů grafu G.

Ke grafu G můžeme sestrojit **graf komponent** $G^{\mathrm{SCC}} = (V^{\mathrm{SCC}}, E^{\mathrm{SCC}})$, jehož vrcholy jsou silně souvislé komponenty grafu G (označme je $C_1, C_2, \ldots, C_k$), a kde $(C_i, C_j) \in E^{\mathrm{SCC}}$ právě když G obsahuje nějakou hranu (x, y) takovou, že $x \in C_i$ a $y \in C_j$.

Silně souvislé komponenty



Silně souvislé komponenty

Algoritmus

1. zavolat $\mathrm{DFS}(G)$ a pro každý vrchol určit $f[u]$
2. vytvořit G^T
3. zavolat $\mathrm{DFS}(G^T)$, přičemž vrcholy v hlavní smyčce jsou probírány v sestupném pořadí podle $f[u]$ (spočítaném v kroku 1)
4. vrcholy každého stromu vytvořeného v kroku 3 tvoří jednu silně souvislou komponentu grafu G

Poznámka: Graf $G^T = (V, E^T)$ vznikne z grafu G otočením směru hran, tj. $E^T = \{(u, v) \mid (v, u) \in E\}$.

Tarjanův algoritmus

Tarjanův algoritmus pro nalezení silně souvislých komponent:

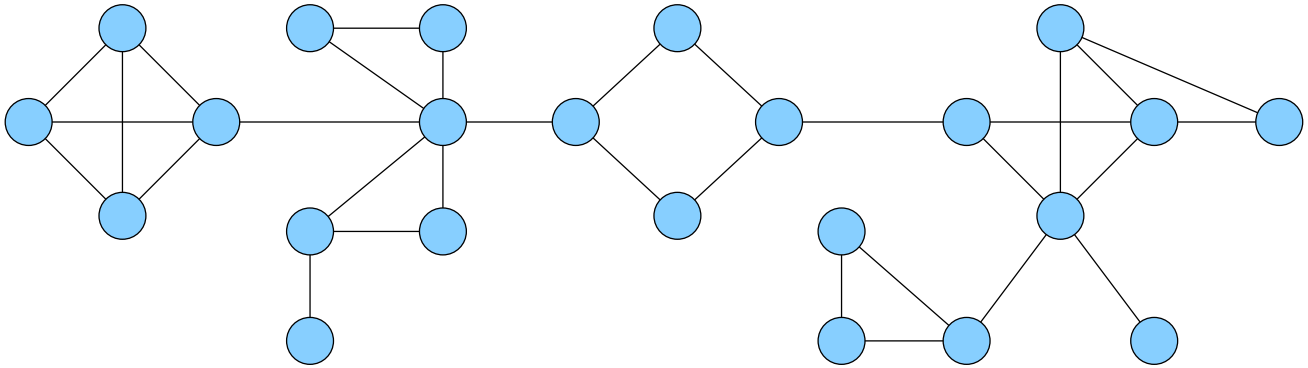
DFS(G)

```
1   for each vertex u ∈ V[G]
2         do visited[u] ← FALSE
3   time ← 0
4   stack ← ∅
5   SCC ← ∅
6   for each vertex u ∈ V[G]
7         do if not visited[u]
8               then DFS-Visit(u)
```

DFS-Visit(u)

```
 1  visited[u] ← TRUE;
 2  r[u] ← d[u] ← time ← time +1
 3  inComponent[u] ← FALSE
 4  PUSH(stack, u)
 5  for each v ∈ Adj[u]
 6        do if not visited[v]   then DFS-Visit(v)
 7           if not inComponent[v]    then r[u] ← MIN(r[u], r[v])
 8  if r[u] = d[u]
 9     then C ← ∅
10           repeat v = POP(stack)
11                  inComponent[v] ← true
12                  C ← C ∪ {v}
13             until v = u
14           SCC ← SCC ∪{C}
```

Problém „Tourist Guide“ (10199)



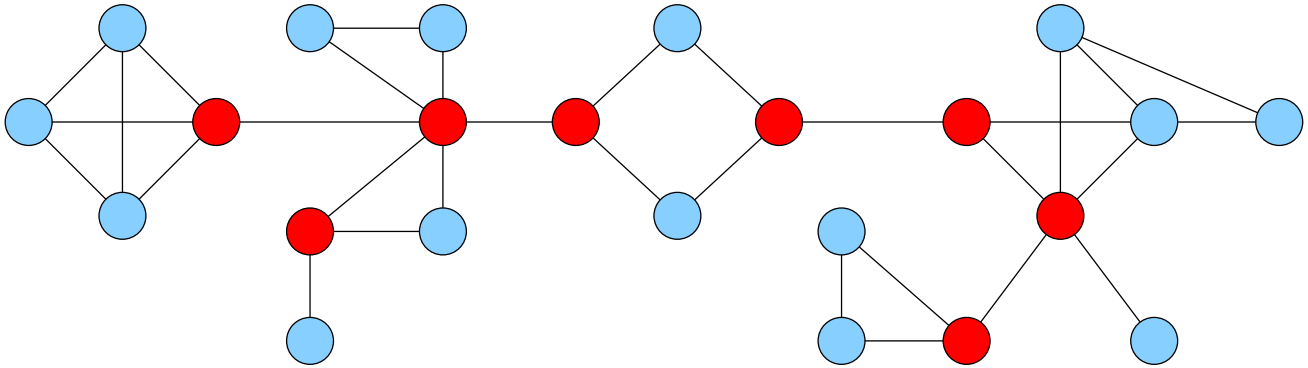
Problém

Vstup: Popis města – seznam křižovatek a cest mezi nimi.

Výstup: Seznam křižovatek, na kterých jsou umístěny kamery.

Víme, že na křižovatce C je kamera umístěna právě když existují nějaké dvě křižovatky A a B takové, že při cestě z A do B (nebo z B do A) musíme projet křižovatkou C.

Problém „Tourist Guide“ (10199)



Problém

Vstup: Popis města – seznam křižovatek a cest mezi nimi.

Výstup: Seznam křižovatek, na kterých jsou umístěny kamery.

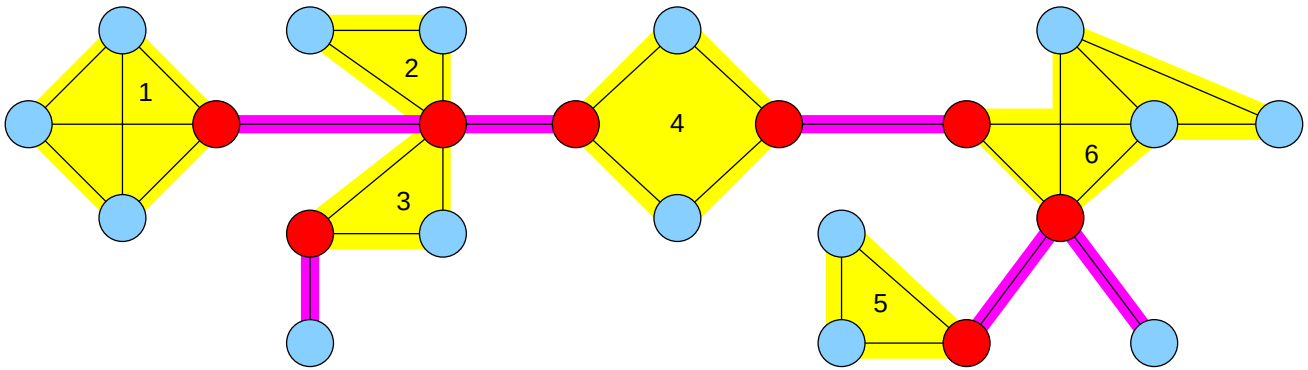
Víme, že na křižovatce C je kamera umístěna právě když existují nějaké dvě křižovatky A a B takové, že při cestě z A do B (nebo z B do A) musíme projet křižovatkou C.

Artikulace, mosty a 2-souvislé komponenty

Nechť G je souvislý neorientovaný graf.

Definice

Vrchol v je **artikulací** grafu G, jestliže jeho odstraněním se graf G rozpadne na více komponent.
Hrana e je **mostem**, jestliže jejím odstraněním se graf G rozpadne na dvě komponenty.
2-souvislá komponenta grafu G je maximální podgraf grafu G takový, že libovolné dvě jeho hrany leží na společném cyklu.



Hledání artikulací

Nechť G je souvislý neorientovaný graf a nechť G_π je strom, který vznikne při prohledávání grafu G do hloubky.

Tvrzení

Kořen stromu G_π je v G artikulací právě když má v G_π alespoň dva přímé potomky.

Hledání artikulací

Tvrzení

Pro každý vrchol v, který není v G_π kořenem, platí, že v je artikulací právě když má nějakého (přímého) potomka s takového, že z s ani ze žádného jeho potomka nevede zpětná hrana do žádného z předchůdců vrcholu v.

Definujeme

$$low[v] = \min \begin{cases} d[v] & \\ d[w] & : \text{ existuje zpětná hrana } (u, w), \\ & \text{kde } u \text{ nějaký potomek } v \end{cases}$$

Hledání artikulací

```
void dfs()
{
    time = 0;
    for (int i = 0; i < n; i++) {
        Node *u = &nodes[i];
        if (u->color == WHITE) {
            dfs_visit(u);
            u->articulation = (u->children > 1);
        }
    }
}
```

```
int dfs_visit(Node *u) {
    u->color = GRAY; u->d = ++time;
    int ret = u->d;
    for (Edge *e = u->edges; e; e = e->next) {
        Node *v = e->node;
        if (v->color == WHITE) {
            v->parent = u; u->children++;
            int r = dfs_visit(v);
            if (r < ret) ret = r;
            if (u->d <= r) u->articulation = true;
        } else if (v->color == GRAY && u->parent != v) {
            if (v->d < ret) ret = v->d;
        }
    }
    u->color = BLACK;
    return ret;
}
```

Problém „The Problem with the Problem Setter“ (10092)

Máme dánu zásobu problémů a naším úkolem je vybrat některé z nich.

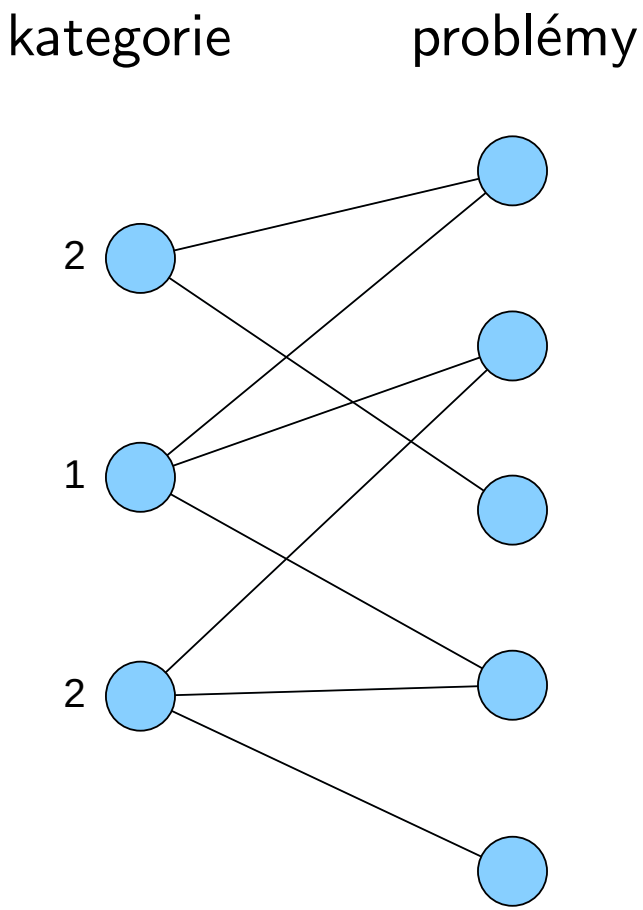
- Každý problém patří do jedné nebo více kategorií.
- Pro každou kategorii je určeno, kolik problémů z dané kategorie máme vybrat.
- Vybrané problémy musí být přiřazeny do jednotlivých kategorií tak, že žádný problém není přiřazen současně do více kategorií.

Problém

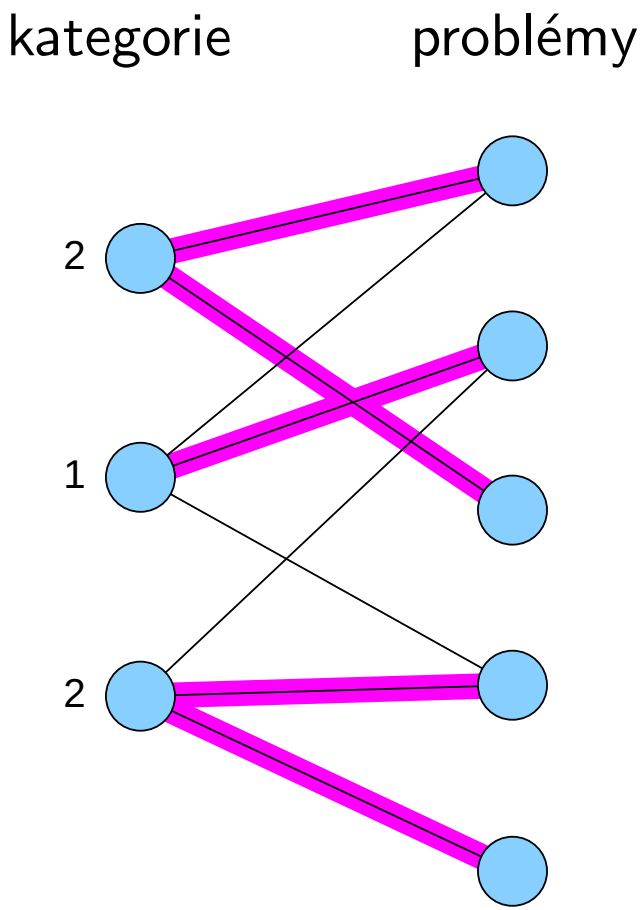
Vstup: Seznam kategorií, přičemž u každé kategorie je uvedeno, kolik problémů z dané kategorie musíme vybrat, a seznam problémů, přičemž u každého problému je uvedeno, do kterých kategorií patří.

Výstup: Seznamy problémů vybraných do jednotlivých kategorií.

Problém „The Problem with the Problem Setter“ (10092)

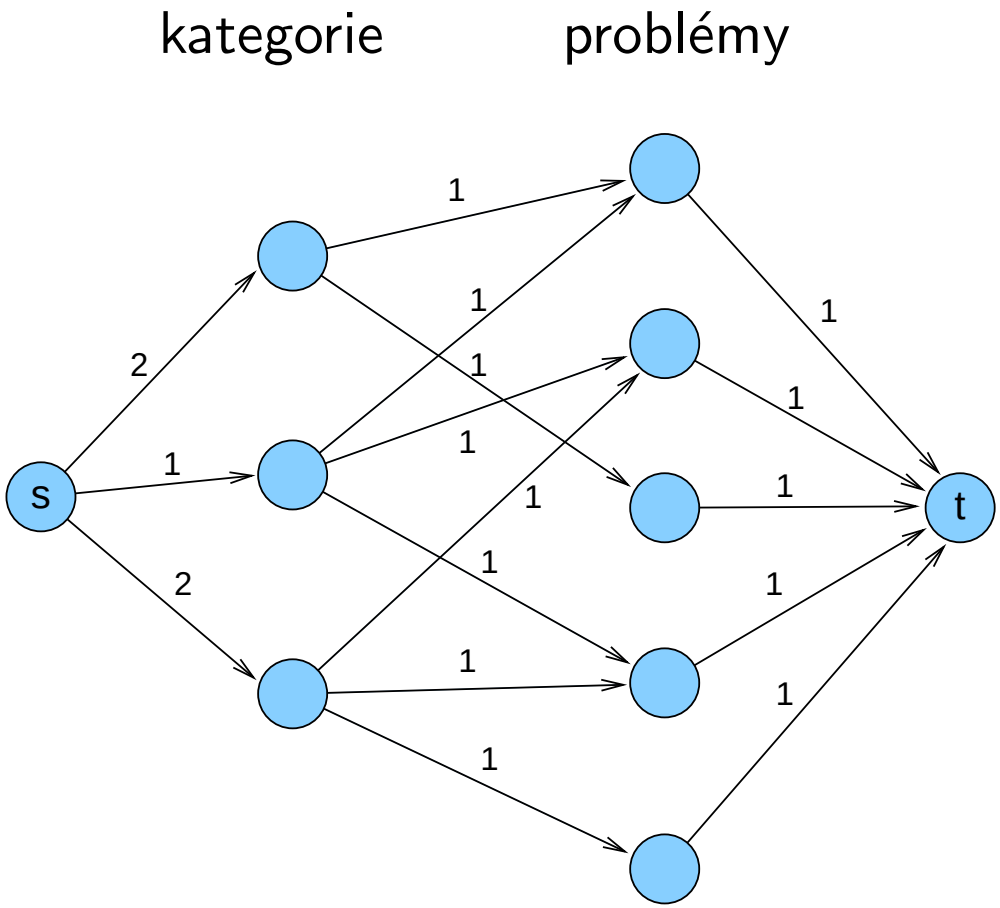


Problém „The Problem with the Problem Setter“ (10092)



Problém „The Problem with the Problem Setter“ (10092)

Problém je možné formulovat jako problém hledání **maximálního toku** v síti:



Hledání maximálního toku v síti

Definice

Síť je orientovaný graf $G = (V, E)$, kde je každé hraně $(u, v) \in E$ přiřazena kapacita $c(u, v) \geq 0$. (Pro $(u, v) \notin E$ je $c(u, v) = 0$.)
V síti se vyskytují dva speciální vrcholy: **zdroj** s a **stok** t.

Definice

Tok je funkce $f : V \times V \to \mathbb{R}$ splňující:

- Pro všechny $u, v \in V$ je $f(u, v) \leq c(u, v)$.
- Pro všechny $u, v \in V$ je $f(u, v) = -f(v, u)$.
- Pro všechny $u \in V - \{s, t\}$ je $\sum_{v \in V} f(u, v) = 0$.

Hledání maximálního toku v síti

Definice

Pro tok f definujeme **velikost toku** $|f|$ jako

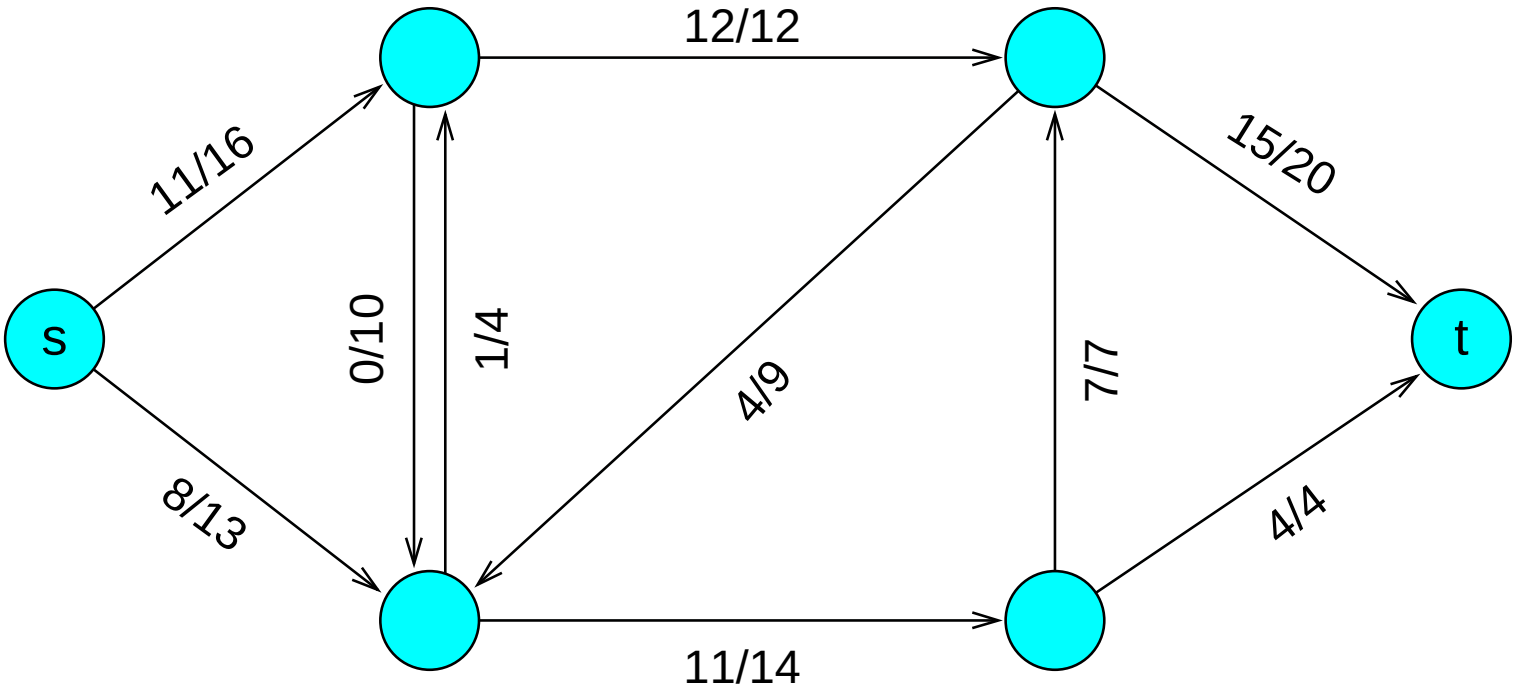
$$|f| = \sum_{v \in V} f(s, v)$$

Problém

Vstup: Síť G.

Výstup: Tok f takový, že $|f|$ je maximální.

Hledání maximálního toku v síti



Hledání maximálního toku v síti

Fordova-Fulkersonova metoda

FORDOVA-FULKERSONOVA-METODA(G, s, t)

1 inicializuj tok f na 0
2 **while** existuje zlepšující cesta p
3 **do** zvyš tok f podél cesty p

Hledání maximálního toku v síti

Mějme síť G a tok f.

Definice

Pro dvojici vrcholů u a v definujeme **residuální kapacitu**

$$c_f(u, v) = c(u, v) - f(u, v)$$

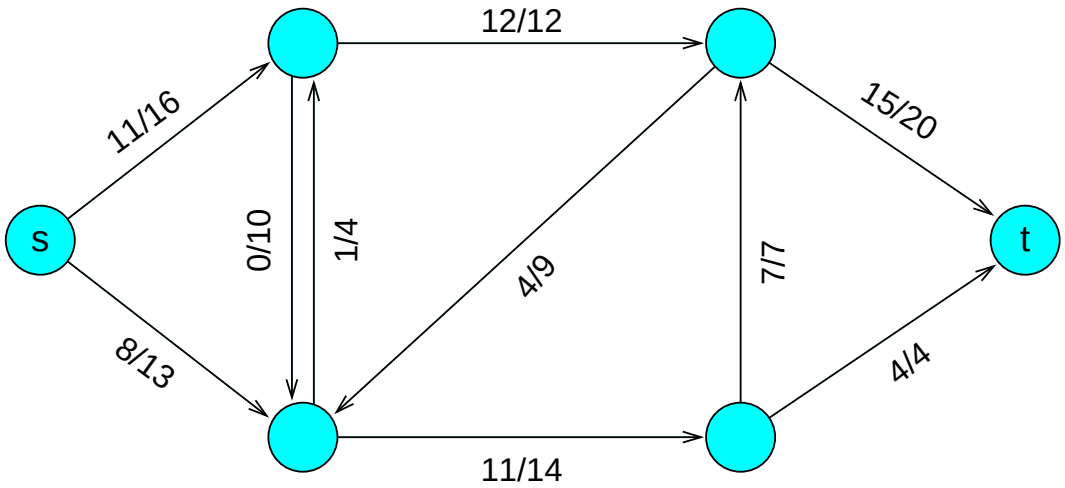
Poznámka: Pokud v síti G neexistuje hrana (u, v) ani (v, u), pak je vždy $c_f(u, v) = 0$.

Definice

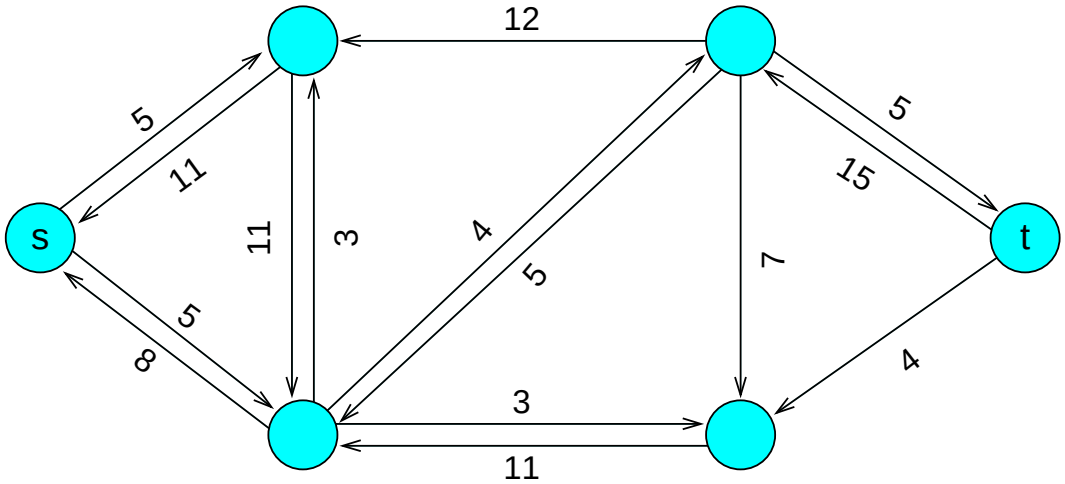
Pro síť $G = (V, E)$ a tok f definujeme **residuální síť** $G_f = (V, E_f)$, kde

$$E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$$

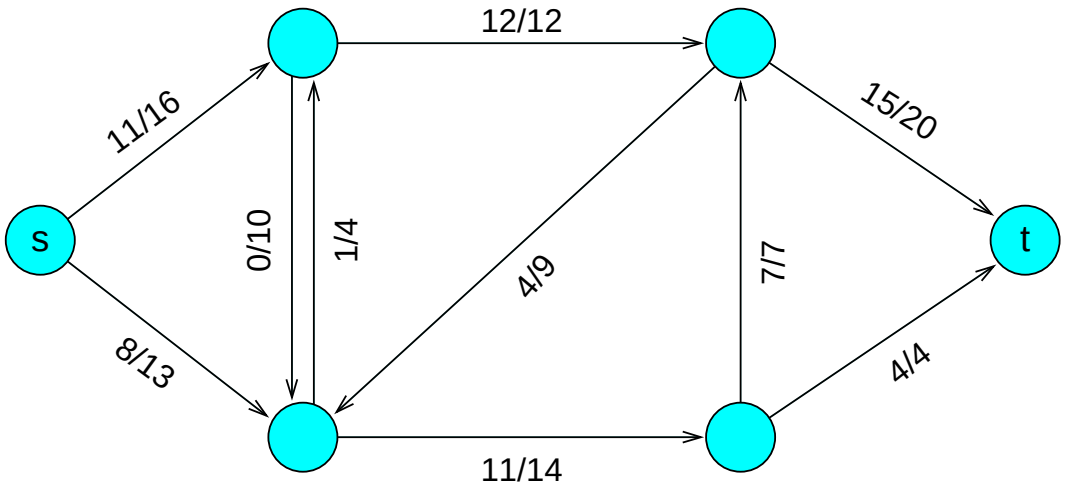
Hledání maximálního toku v síti



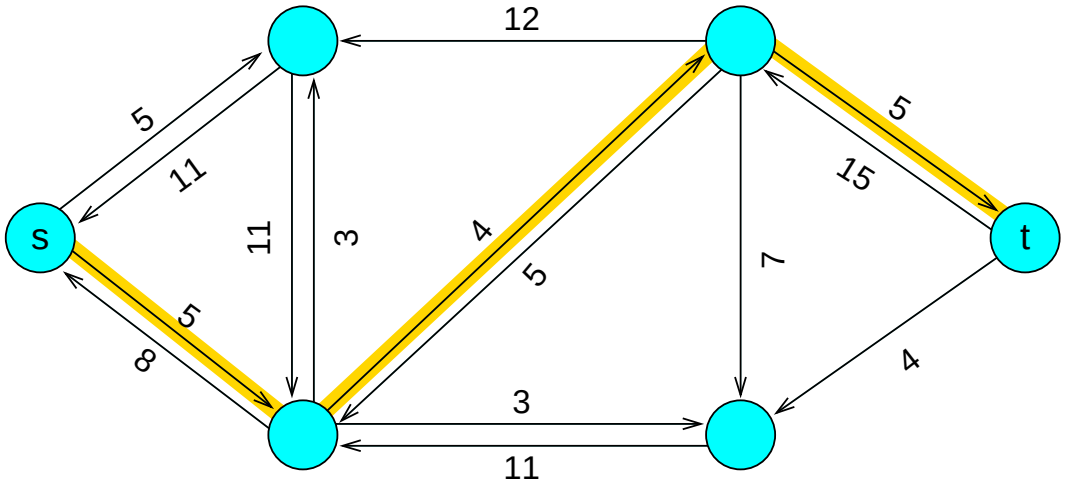
Pro uvedenou síť a tok dostáváme následující residuální síť:



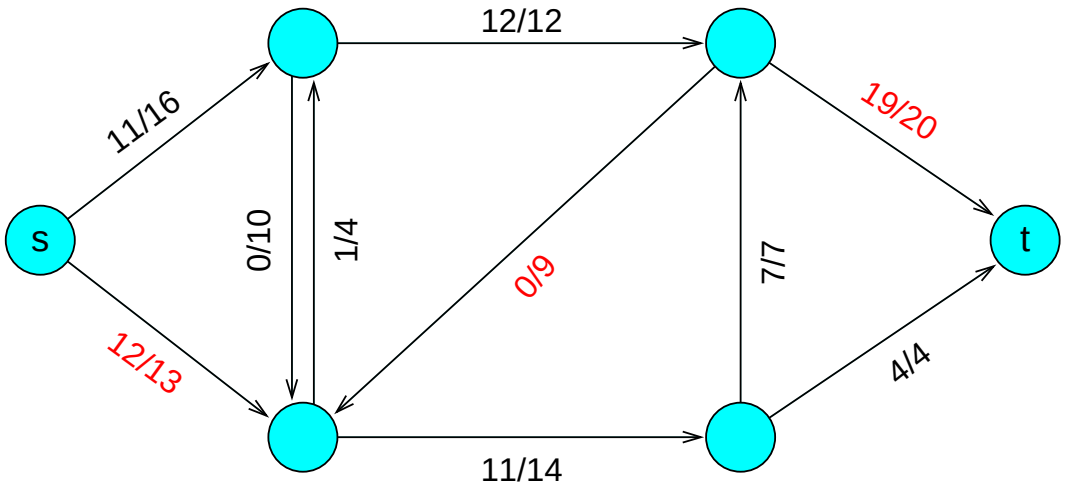
Hledání maximálního toku v síti



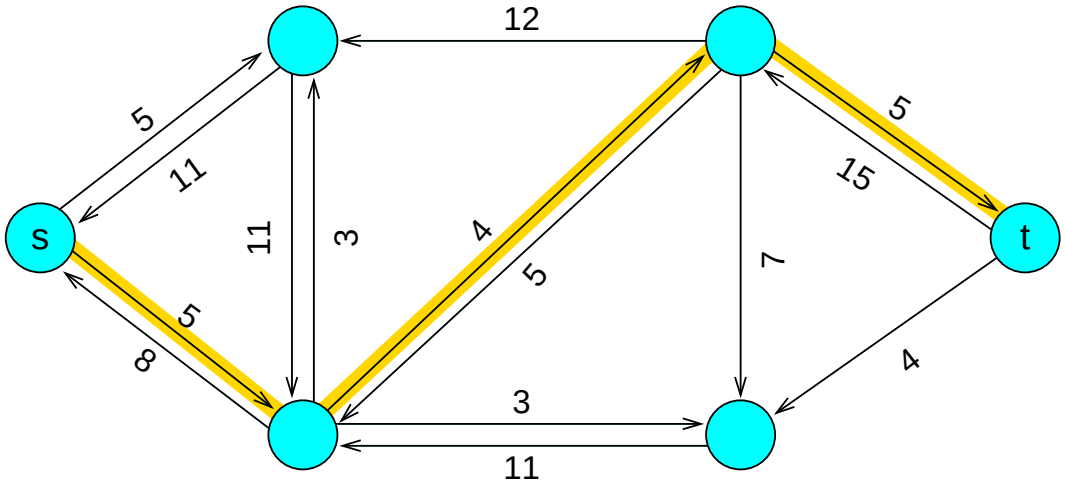
Pro uvedenou síť a tok dostáváme následující residuální síť:



Hledání maximálního toku v síti



Pro uvedenou síť a tok dostáváme následující residuální síť:



Hledání maximálního toku v síti

Věta

Tok f je maximální právě když v G_f neexistuje zlepšující cesta.

Pokud blíže nespecifikujeme, jakým způsobem hledáme zlepšující cesty, je časová složitost Fordovy-Fulkersonovy metody $O(|E||f^*|)$, kde f^* je maximální tok nalezený algoritmem.

Pokud pro hledání zlepšující cesty použijeme prohledávání do šířky a nalezneme tedy v každém kroku vždy **nejkratší** zlepšující cestu z s do t, bude časová složitost algoritmu $O(|V||E|^2)$.

Poznámka: Existují i efektivnější (ale komplikovanější) algoritmy.

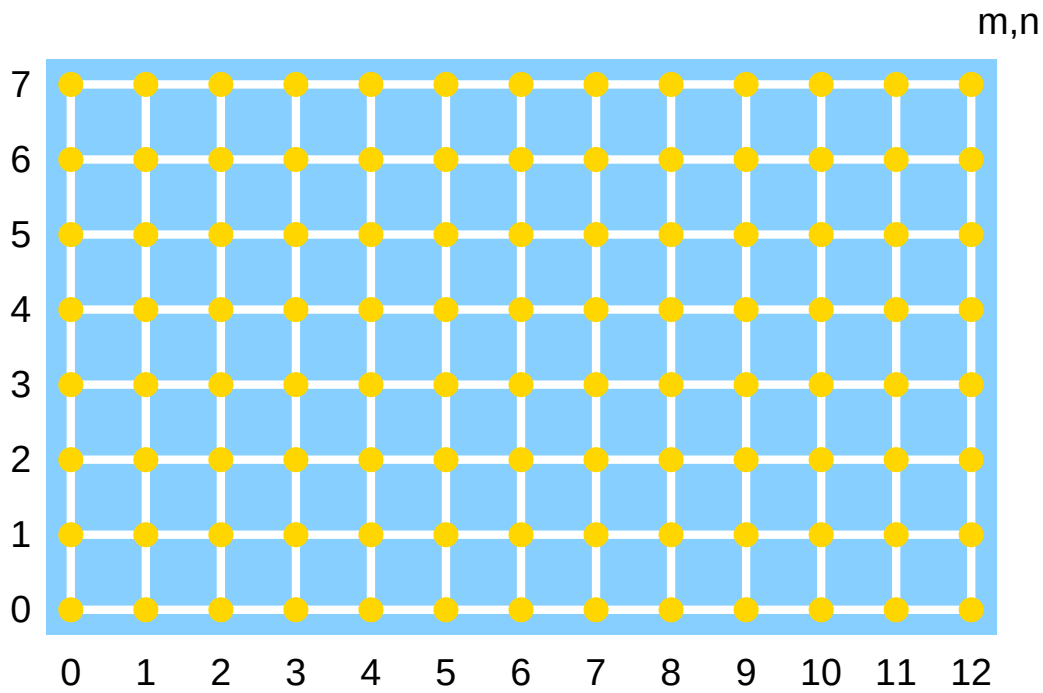
Kombinatorické problémy

Kombinatorické problémy

Problém

Vstup: Čísla m a n.

Výstup: Počet různých cest z bodu $(0, 0)$ do bodu (m, n), jestliže se můžeme pohybovat jen doprava a nahoru.



Kombinační čísla

Definice

Kombinační číslo udává, kolika způsoby lze vybrat k prvků z n prvkové množiny:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Řešení předchozího problému:

$$\binom{m+n}{n}$$

Příklad: Uvažujme výraz vzniklý roznásobením výrazu $(a+b)^n$. Koeficient u členu $a^k b^{n-k}$ bude $\binom{n}{k}$.

Kombinační čísla

Platí

$$\binom{n}{k} = \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!}$$

$$\binom{n}{k} = \binom{n}{n-k} \qquad \binom{n}{0} = 1 \qquad \binom{n}{n} = 1$$

Pro $k > n$ platí

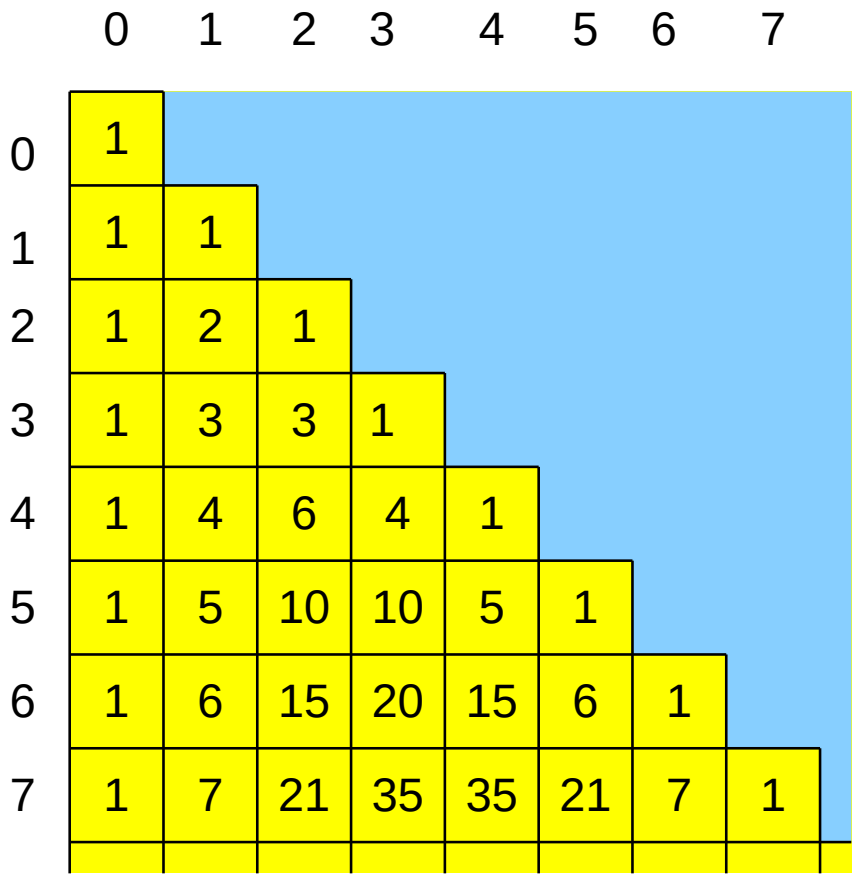
$$\binom{n}{k} = 0$$

Pro výpočet lze použít též následující vztah:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Kombinační čísla

Pascalův trojúhelník:



Kombinační čísla

Rychlejší metoda výpočtu je založena na následujícím vztahu:

$$\binom{n}{k+1} = \binom{n}{k} \cdot \frac{n-k}{k+1}$$

přičemž

$$\binom{n}{0} = 1$$

Důkaz:

$$\binom{n}{k+1} = \frac{n!}{(k+1)!(n-k-1)!} = \frac{n!}{k!(n-k)!} \cdot \frac{n-k}{k+1} = \binom{n}{k} \cdot \frac{n-k}{k+1}$$

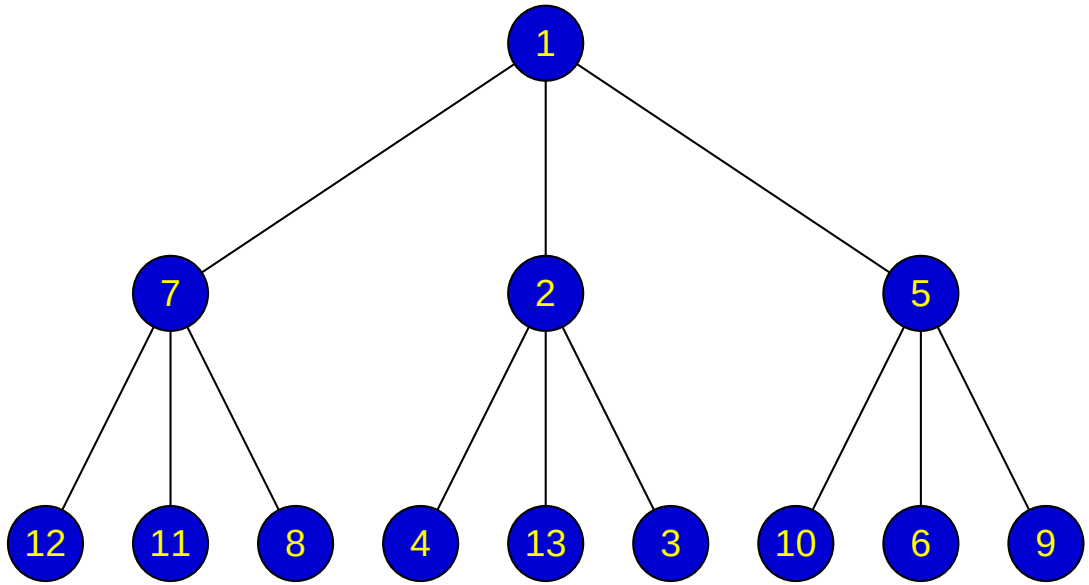
Problém „Complete Tree Labeling“ (10247)

Problém

Vstup: Čísla k, d taková, že $k \cdot d \leq 21$.

Výstup: Kolika způsoby můžeme přiřadit vrcholům k-árního stromu hloubky d čísla $\{1, 2, \ldots, n\}$, kde n je počet vrcholů stromu tak, aby číslo přiřazené vrcholu bylo vždy menší než čísla přiřazená jeho potomkům.

$k = 3$
$d = 2$



Problém „Pairsumonious Numbers“ (10202)

Pro daných n čísel $a_1, a_2, \ldots, a_n$ můžeme snadno spočítat pro všech $n(n-1)/2$ dvojic a_i, a_j, kde $i < j$, součty $a_i + a_j$.

Problém

Vstup: Číslo n a $m = n(n-1)/2$ čísel $s_1, s_2, \ldots, s_m$.

Výstup: Čísla $a_1, a_2, \ldots, a_n$ taková, že součty jejich dvojic jsou $s_1, s_2, \ldots, s_m$, případně informace, že řešení neexistuje.

Poznámka: U daného s_k víme, že $s_k = a_i + a_j$ pro nějaká a_i a a_j, ale nevíme o která a_i a a_j se jedná.

Problém „Queue“ (10128)

Ve frontě stojí N lidí, přičemž každý je jinak vysoký.

Člověk vidí doleva jen pokud nalevo od něj nestojí někdo vyšší než on a doprava jen pokud napravo od něj nestojí někdo vyšší než on.

Problém

Vstup: Čísla N, L, R.

Otázka: Kolika způsoby lze frontu N lidí seřadit tak, aby právě L lidí vidělo doleva a právě R lidí vidělo doprava?

Problém „Queue“ (10128)

Ve frontě stojí N lidí, přičemž každý je jinak vysoký.

Člověk vidí doleva jen pokud nalevo od něj nestojí někdo vyšší než on a doprava jen pokud napravo od něj nestojí někdo vyšší než on.

Problém

Vstup: Čísla N, L, R.

Otázka: Kolika způsoby lze frontu N lidí seřadit tak, aby právě L lidí vidělo doleva a právě R lidí vidělo doprava?

$$\begin{aligned} a[n][l][r] \quad = \quad & a[n-1][l-1][r] \quad + \\ & a[n-1][l][r-1] \quad + \\ & (n-2) \cdot a[n-1][l][r] \end{aligned}$$

Problém „Polynomial Coefficients“ (10105)

Uvažujme rozvoj polynomu $(x_1 + x_2 + \ldots + x_k)^n$.

Problém

Vstup: Čísla n,k a $n_1, n_2, \ldots, n_k$ taková, že $0 < n, k < 13$ a $n_1 + n_2 + \ldots + n_k = n$.

Výstup: Koeficient u členu $x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k}$.

Problém „Polynomial Coefficients“ (10105)

Uvažujme rozvoj polynomu $(x_1 + x_2 + \ldots + x_k)^n$.

Problém

Vstup: Čísla n,k a $n_1, n_2, \ldots, n_k$ taková, že $0 < n, k < 13$ a $n_1 + n_2 + \ldots + n_k = n$.

Výstup: Koeficient u členu $x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k}$.

$$\binom{n}{n_1}\binom{n-n_1}{n_2}\binom{n-n_1-n_2}{n_3}\cdots\binom{n_k}{n_k}$$

Problém „War“ (10158)

Tajný agent sleduje agenty dvou navzájem znepřátelených států, přičemž ovšem neví, kdo patří ke které straně.

Z občasných schůzek dvojic agentů je schopen vypozorovat, kdy daná dvojice patří ke stejné straně (jsou přátelé) a kdy k opačným stranám (jsou nepřátelé).

Chceme vyhodnocovat vztahy mezi agenty.

Uvažujme posloupnost následujících operací:

- $setFriends(x, y)$ – zjistili jsme, že x a y patří ke stejné straně
- $setEnemies(x, y)$ – zjistili jsme, že x a y patří každý k jiné straně
- $areFriends(x, y)$ – dotaz, zda je jisté, že x a y patří ke stejné straně
- $areEnemies(x, y)$ – dotaz, zda je jisté, že x a y patří každý k jiné straně

Problém „War“ (10158)

Operace *setFriends* a *setEnemies* musí být ignorovány, jestliže jsou v rozporu s doposud získanými informacemi, přičemž musí být vypsáno odpovídající chybové hlášení.

Operace *areFriends* a *areEnemies* musí vypsat odpověď na danou otázku.

Relace „být přáteli“, označená $\sim$, a „být nepřáteli“, označená $*$, mají následující vlastnosti:

- $\sim$ je ekvivalence:
  - $x \sim x$
  - Pokud $x \sim y$, pak i $y \sim x$.
  - Pokud $x \sim y$ a $y \sim z$, pak i $x \sim z$.
- $*$ je symetrická a ireflexivní:
  - Pokud $x * y$, pak $y * x$.
  - Nikdy neplatí $x * x$.
- Navíc:
  - Pokud $x * y$ a $y * z$, pak $x \sim z$.
  - Pokud $x \sim y$ a $y * z$, pak $x * z$.

Problém „War“ (10158)

Problém

Vstup: Počet agentů n a posloupnost operací *setFriends*, *setEnemies*, *areFriends* a *areEnemies*.

Výstup: Odpovídající výstupy pro každou z těchto operací.

Poznámka: Agenti jsou označeni čísly $0, 1, \ldots, n-1$.

Datová struktura pro disjunktní množiny

Chceme udržovat informace o prvcích $x_1, x_2, \ldots, x_n$, které jsou rozděleny do disjunktních množin $S_1, S_2, \ldots, S_k$.

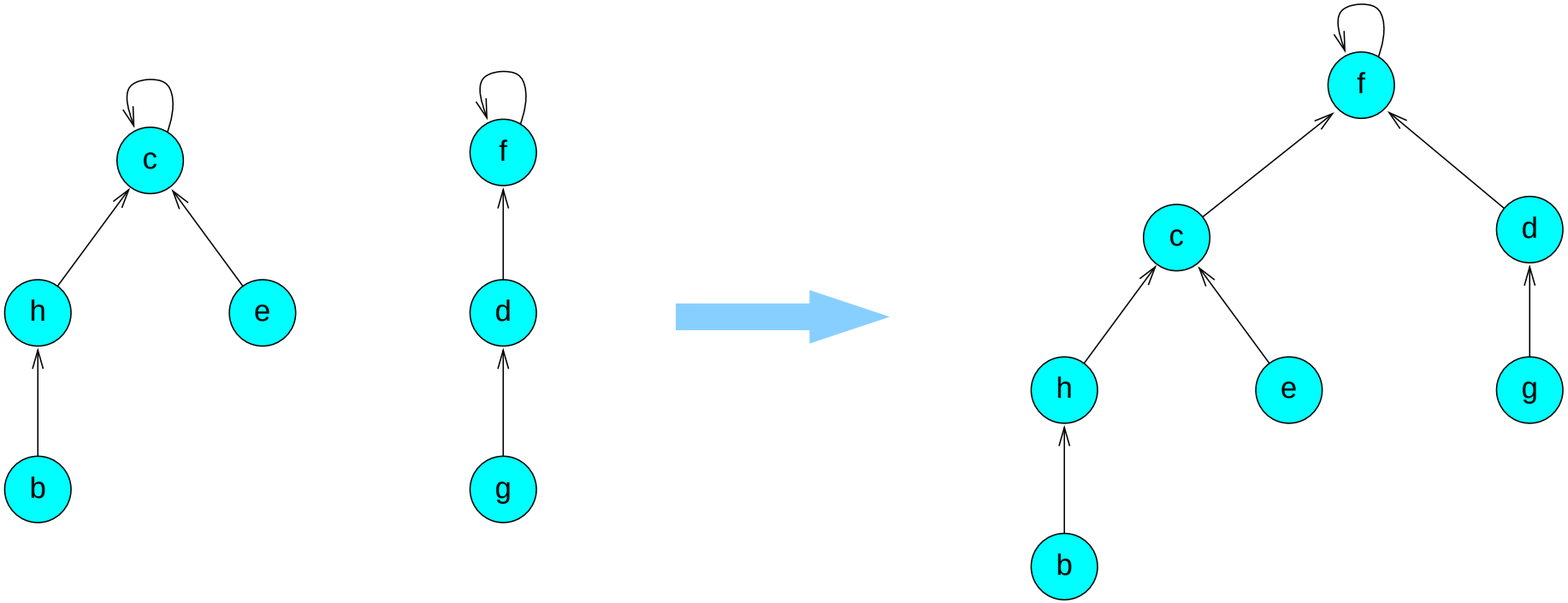
Každá množina má jednoho reprezentanta.

Chceme provádět následující operace:

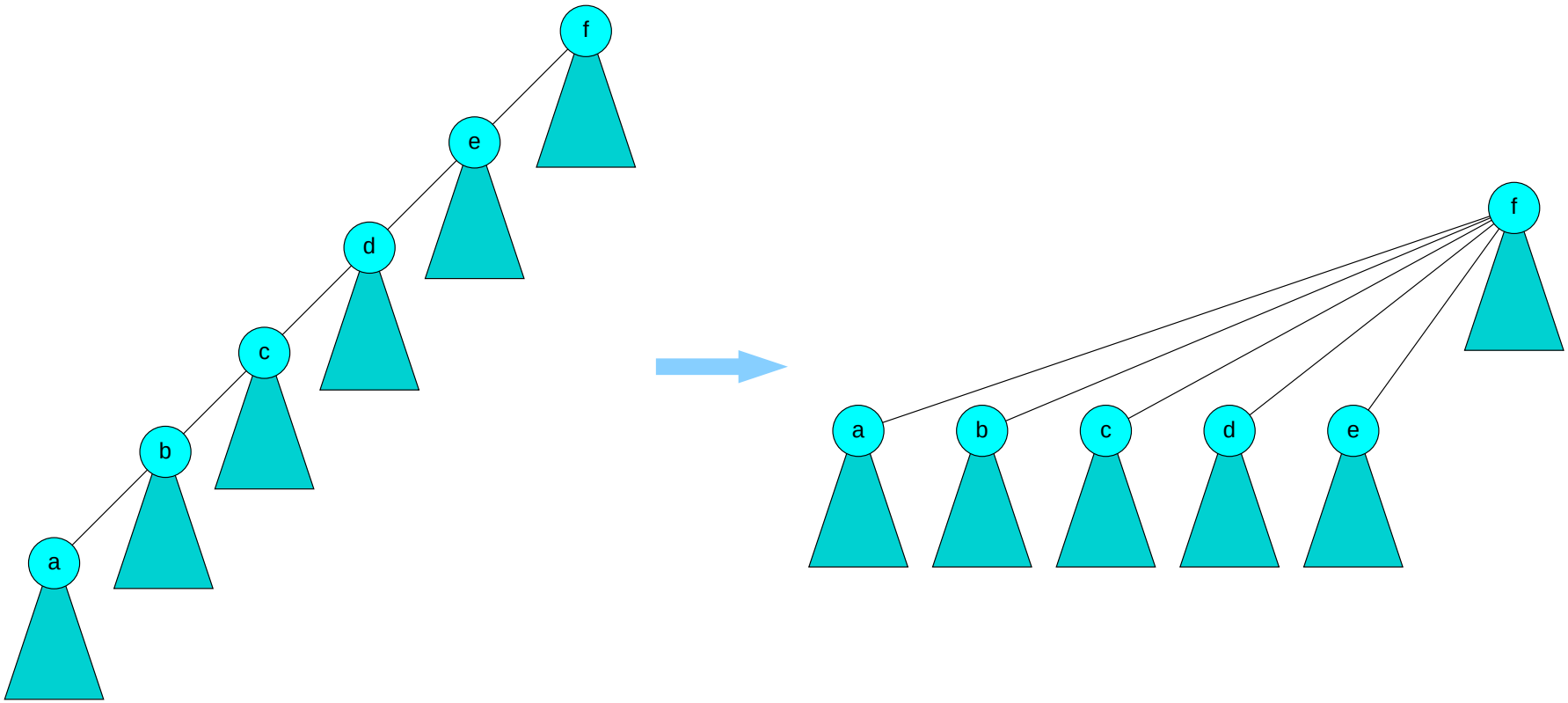
- MAKE-SET(x) – vytvoření nové množiny obsahující pouze prvek x (x nesmí být prvkem žádné již vytvořené množiny).
- UNION(x, y) – sjednocení množin obsahujících prvky x a y.
- FIND-SET(x) – nalezení reprezentanta množiny obsahující prvek x.

Datová struktura pro disjunktní množiny

Jednotlivé množiny mohou být reprezentovány jako stromy, přičemž kořen stromu je reprezentantem dané množiny:



Datová struktura pro disjunktní množiny



Datová struktura pro disjunktní množiny

MAKE-SET(x)

```
1  p[x] ← x
2  rank[x] ← 0
```

FIND-SET(x)

```
1  if x ≠ p[x]
2      then p[x] ← FIND-SET(p[x])
3  return p[x]
```

Datová struktura pro disjunktní množiny

```
UNION(x, y)
1   v ← FIND-SET(x)
2   w ← FIND-SET(y)
3   if rank[v] > rank[w]
4       then p[w] ← v
5       else p[v] ← w
6            if rank[v] = rank[w]
7               then rank[w] ← rank[w] + 1
```

Datová struktura pro disjunktní množiny

Tvrzení

Sekvence m operací MAKE-SET, UNION a FIND-SET, z nichž n operací je MAKE-SET, vyžaduje v nejhorším případě čas $O(m\,\alpha(n))$.

Poznámka: $\alpha(n)$ je extrémně pomalu rostoucí funkce, která se pro jakékoliv „rozumné“ hodnoty n chová jako konstantní funkce (například pro $n < 10^{80}$ je $\alpha(n) \leq 4$).

Přesná definice funkce $\alpha(n)$: $\alpha(n) = \min\{k : A_k(1) \geq n\}$

$$A_k(j) = \begin{cases} j+1 & \text{pokud } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{pokud } k \geq 1 \end{cases}$$

kde $A_k^{(0)}(j) = j$ a $A_k^{(i+1)}(j) = A_k(A_k^{(i)}(j))$.

Teorie čísel

Základní pojmy

Celá čísla $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
Přirozená čísla $\mathbb{N} = \{0, 1, 2, 3, 4, \ldots\}$

Notace $d \mid a$ označuje, že d **dělí** a, což znamená, že existuje nějaké $k \in \mathbb{Z}$ takové, že $a = kd$.
Pokud $d \mid a$, říkáme také, že a je **násobkem** d.
Pokud d nedělí a, píšeme $d \nmid a$.

Pokud $d \mid a$ a $d \geq 0$, říkáme, že d je **dělitelem** a.
Například dělitelé čísla 24 jsou 1, 2, 3, 4, 6, 8, 12 a 24.

Každé číslo a je dělitelné **triviálními děliteli** 1 a a.

Prvočísla

Definice

Prvočíslo je takové celé číslo $a > 1$, které je dělitelné pouze triviálními děliteli 1 a a.

Prvních 20 prvočísel:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71$$

Celé číslo $a > 1$, které není prvočíslem, se nazývá číslo **složené**.

Poznámka: Čísla 0 a 1 nejsou ani prvočísla ani čísla složená.

Eratosthenovo síto

Eratosthenovo síto (Sieve of Eratosthenes) je algoritmus umožňující poměrně rychle vygenerovat všechna „malá“ prvočísla, případně pro „malá“ složená čísla najít jejich dělitele:

```
void init_sieve() {
    for (int i = 2; i < MAX; i++)
        if (a[i] == 0)
            for (int j = i; j < MAX; j += i)
                a[j] = i;
}
```

Poznámka: Prvek $a[i]$ obsahuje největší prvočíselný dělitel čísla i. Číslo i je prvočíslo, právě když $a[i] = i$.

Eratosthenovo síto

Počet provedených operací pro pole velikosti n je menší než

$$n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots + \frac{n}{n} = n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right)$$

Pro libovolné $n > 0$ je n-té **harmonické číslo** H_n definováno jako

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Věta

Pro libovolné H_n platí: $\ln n + \frac{1}{n} < H_n < \ln n + 1$

Počet provedených operací je tedy maximálně $O(n \log n)$.

Prvočíslelnost a faktorizace

Problém **prvočíselnosti** je problém zjistit, zda je dané x prvočíslem.

Problém **faktorizace** je problém najít pro dané x rozklad na prvočísla (v zádadě stačí umět najít nějaký netriviální dělitel).

Pro „malá“ x se dají oba problémy snadno řešit:
Stačí vyzkoušet jako potenciální dělitele všechna čísla od 2 do $\lfloor\sqrt{x}\rfloor$.
(Můžeme vynechat sudá čísla větší než 2.)

Pokud n je počet bitů čísla x, provede tento algoritmus zhruba $2^{n/2}$ operací dělení.

Prvočíslelnost a faktorizace

Problém prvočíselnosti se dá řešit v čase polynomiálním vzhledem k n:

- Deterministický algoritmus (se složitostí $O(n^{12+\varepsilon})$) byl nalezen teprve v roce 2002 (Agrawal-Kayal-Saxena).
  (Později se podařilo složitost o něco snížit.)
- Randomizované algoritmy byly známy dříve: Solovay-Strassen (1977), Miller-Rabin (1978)

Pro problém faktorizace není znám polynomiální algoritmus.

Problém „Factovisors“ (10139)

Problém

Vstup: Čísla m, n ($0 \leq m, n < 2^{31}$).

Otázka: Dělí m hodnotu $n!$?

Poznámka: $0! = 1$, $n! = n \cdot (n-1)!$ pro $n > 0$

Problém „Factovisors“ (10139)

Problém

Vstup: Čísla m, n ($0 \leq m, n < 2^{31}$).

Otázka: Dělí m hodnotu $n!$?

Poznámka: $0! = 1$, $n! = n \cdot (n-1)!$ pro $n > 0$

Stačí využít následující větu:

Věta (Legendre)

Počet výskytů prvočísla p v rozkladu čísla $n!$ na prvočísla je

$$\sum_{k \geq 1} \left\lfloor \frac{n}{p^k} \right\rfloor .$$

Operace dělení a modulo

Pro libovolné celé číslo a a libovolné kladné celé číslo n existuje právě jedno celé číslo q a právě jedno celé číslo r takové, že $0 \leq r < n$ a $a = qn + r$.

Hodnota $q = \lfloor a/n \rfloor$ se nazývá **podíl**.
Hodnota $r = a \bmod n$ se nazývá **zbytek** po dělení.

Zbytková třída modulo n obsahující celé číslo a je

$$[a]_n = \{a + kn : k \in \mathbb{Z}\}.$$

Naříklad $[3]_7 = \{\ldots, -11, -4, 3, 10, 17, \ldots\}$.

Poznámka: Tutéž třídu můžeme označit například též $[-4]_7$ nebo $[10]_7$.

Zbytkové třídy

Místo $a \in [b]_n$ píšeme též $a \equiv b \pmod{n}$.

Množina všech těchto tříd je

$$\mathbb{Z}_n = \{[a]_n : 0 \leq a < n\}$$

Často píšeme jen

$$\mathbb{Z}_n = \{0, 1, \ldots, n-1\}$$

kde 0 reprezentuje $[0]_n$, 1 reprezentuje $[1]_n$ atd.

Poznámka: Musíme však mít stále na paměti, že se jedná o zbytkové třídy. Naříklad -1 jako prvek $\mathbb{Z}_n$ reprezentuje $[n-1]_n$, protože

$$-1 \equiv n - 1 \pmod{n}$$

Modulární aritmetika

Neformálně se dá říct, že modulární aritmetika je stejná jako aritmetika na celých číslech s tím rozdílem, že pracujeme modulo n:

- Pracujeme pouze s čísly $\{0, 1, 2, \ldots, n-1\}$.
- Pokud je výsledek nějaké aritmetické operace x (v oboru celých čísel), nahradíme ho hodnotou $(x \bmod n)$.

Například pokud pracujeme modulo 7:

$$5 + 4 = 2 \qquad 3 - 4 = 6 \qquad 3 \cdot 5 = 1 \qquad 0 \cdot 2 = 0$$

Modulární aritmetika

Protože platí, že pokud $a \equiv a' \pmod{n}$ a $b \equiv b' \pmod{n}$, pak

$$\begin{aligned} a+b &\equiv a'+b' && \pmod{n} \\ ab &\equiv a'b' && \pmod{n} \end{aligned}$$

můžeme definovat na $\mathbb{Z}_n$ sčítání a násobení modulo n, označené $+_n$ a $\cdot_n$, jako

$$\begin{aligned} [a]_n +_n [b]_n &= [a+b]_n \\ [a]_n \cdot_n [b]_n &= [ab]_n \end{aligned}$$

Poznámka: Podobně můžeme zavést i odčítání.

Umocňování v modulární aritmetice

Problém

Vstup: Přirozená čísla a, b, n.

Výstup: Hodnota $a^b \bmod n$.

Využijeme toho, že $a^{2i} = a^i \cdot a^i$ a $a^{2i+1} = a^i \cdot a^i \cdot a$.

MODULAR-EXPONENTIATION(a, b, n)

```
1  d ← 1
2  ⟨b_k, b_{k−1}, ..., b_0⟩ je binární reprezentace b
3  for i ← k downto 0
4        do d ← (d · d) mod n
5           if b_i = 1
6              then d ← (d · a) mod n
7  return d
```

Umocňování v modulární aritmetice

Alternativně můžeme postupovat zprava doleva:

```
int modular_exponentiation(int a, int b, int n) {
    int d = 1;
    while (b > 0) {
        if (b & 1) {
            d = (d * a) % n;
        }
        a = (a * a) % n;
        b >>= 1;
    }
    return d;
}
```

Umocňování

Algoritmus pro umocňování se dá využít pro rychlé nalezení n-tého prvku posloupnosti $x_0, x_1, x_2, \ldots$, kde máme zadány hodnoty $a_0, a_1, \ldots, a_{k-1}$ a $b_0, b_1, \ldots, b_{k-1}$, a kde:

$$x_i = \begin{cases} a_i & \text{pro } i < k \\ b_{k-1}x_{i-1} + b_{k-2}x_{i-2} + \cdots + b_0 x_{i-k} & \text{pro } i \geq k \end{cases}$$

Příklad: Pro $k = 2$, $a_0 = 0$, $a_1 = 1$, $b_0 = 1$ a $b_1 = 1$ dostáváme Fibonacciho posloupnost.

Umocňování

Všimněme si, že

$$\begin{pmatrix} b_{k-1} & b_{k-2} & \cdots & b_1 & b_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ x_{i-3} \\ \vdots \\ x_{i-k} \end{pmatrix} = \begin{pmatrix} x_i \\ x_{i-1} \\ x_{i-2} \\ \vdots \\ x_{i-k+1} \end{pmatrix}$$

a tedy

$$\begin{pmatrix} b_{k-1} & b_{k-2} & \cdots & b_1 & b_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}^{n-k+1} \begin{pmatrix} x_{k-1} \\ x_{k-2} \\ x_{k-3} \\ \vdots \\ x_0 \end{pmatrix} = \begin{pmatrix} x_n \\ x_{n-1} \\ x_{n-2} \\ \vdots \\ x_{n-k+1} \end{pmatrix}$$

Společní dělitelé

Pokud $d \mid a$ a $d \mid b$, pak d je **společný dělitel** a a b.

Například společní dělitelé 24 a 30 jsou: 1, 2, 3, 6

Z $d \mid a$ a $d \mid b$ vyplývá $d \mid (a+b)$ a $d \mid (a-b)$.

Obecně platí:

Pokud $d \mid a$ a $d \mid b$, pak $d \mid (ax+by)$ pro libovolné $x, y \in \mathbb{Z}$.

Pokud $a \mid b$, pak buď $|a| \leq |b|$ nebo $b = 0$.

Pokud $a \mid b$ a $b \mid a$, pak $a = \pm b$.

Největší společný dělitel

Největší společný dělitel (greatest common divisor) dvou celých čísel a a b takových, že alespoň jedno z nich není 0, je největší číslo z množiny společných dělitelů a a b. Označujeme ho $\gcd(a, b)$.

Například: $\gcd(24, 30) = 6$ $\gcd(5, 7) = 1$ $\gcd(0, 9) = 9$

Pokud $a \neq 0$ a $b \neq 0$, pak $\gcd(a, b)$ je číslo mezi 1 a $\min(|a|, |b|)$.
Definujeme $\gcd(0, 0) = 0$.

Některé jednoduché vlastnosti:

$$\begin{aligned} \gcd(a, b) &= \gcd(b, a) \\ \gcd(a, b) &= \gcd(-a, b) \\ \gcd(a, b) &= \gcd(|a|, |b|) \\ \gcd(a, 0) &= |a| \\ \gcd(a, ka) &= |a| \end{aligned}$$

Eukleidův algoritmus

```
EUCLID(a, b)
1  if b = 0
2      then return a
3      else return EUCLID(b, a mod b)
```

Poznámka: Zřejmě asi nejstarší popsaný algoritmus (zhruba 300 let před n.l.)

Příklad:
$\textsc{Euclid}(30, 21) = \textsc{Euclid}(21, 9) = \textsc{Euclid}(9, 3) = \textsc{Euclid}(3, 0) = 3$

Eukleidův algoritmus

Poznámka: S výjikou prvního volání vždy platí $a > b \geq 0$. Pokud $a < b$, zavolá se $\textsc{Euclid}(b, a)$, a pokud $a = b$, zavolá se $\textsc{Euclid}(b, 0)$.

Nerekurzivní implementace:

$\textsc{Euclid}(a, b)$

```
1  while b > 0
2        do c ← a mod b
3           a ← b
4           b ← c
5  return a
```

Eukleidův algoritmus

Fibonacciho čísla: $F_0 = 0$, $F_1 = 1$, $F_i = F_{i-1} + F_{i-2}$ pro $i \geq 2$
Posloupnost 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Fibonacciho čísla souvisí s tzv. **zlatým řezem** ϕ:

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\ldots \qquad \widehat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\ldots$$

Indukcí se dá ukázat, že

$$F_i = \frac{\phi^i - \widehat{\phi}^i}{\sqrt{5}}$$

Vzhledem k tomu, že $|\widehat{\phi}| < 1$, je $|\widehat{\phi}^i|/\sqrt{5} < 1/\sqrt{5} < 1/2$, takže F_i je rovno $\phi^i/\sqrt{5}$ zaokrouhleno na nejbližší celé číslo. Fibonacciho čísla tedy rostou exponenciálně.

Eukleidův algoritmus

Tvrzení

Pokud $a > b \geq 1$ a $\textsc{Euclid}(a, b)$ vykoná $k \geq 1$ rekurzivních volání, pak $a \geq F_{k+2}$ a $b \geq F_{k+1}$.

Důkaz: Indukcí podle k.

Věta (Lamé)

Pro libovolné $k \geq 1$ platí, že pokud $a > b \geq 1$ a $b < F_{k+1}$, pak $\textsc{Euclid}(a, b)$ provede méně než k rekurzivních volání.

Navíc můžeme indukcí podle k ukázat, že $\textsc{Euclid}(F_{k+1}, F_k)$ provede právě $k - 1$ rekurzivních volání.

Eukleidův algoritmus

Počet rekurzivních volání pro libovolné a, b, kde $a > b$, je $O(\log b)$.

Pokud oba operandy mají n bitů, a předpokládáme, že časová složitost operace dělení je $O(n^2)$, dostáváme celkovou časovou složitost $O(n^3)$.

Podrobnější analýzou se dá ukázat, že celková časová složitost je $O(n^2)$.

Rozšířený Eukleidův algoritmus

Problém

Vstup: Nazáporná celá kladná čísla a, b.

Výstup: Celá čísla d, x, y taková, že $d = \gcd(a, b)$ a $d = ax + by$.

EXTENDED-EUCLID(a, b)

```
1  if b = 0
2      then return (a, 1, 0)
3  (d', x', y') ← EXTENDED-EUCLID(b, a mod b)
4  (d, x, y) ← (d', y', x' − ⌊a/b⌋y')
5  return (d, x, y)
```

Rozšířený Eukleidův algoritmus

- Pokud $b = 0$: Funkce vrátí $(a, 1, 0)$. Zjevně platí $a = \gcd(a, 0)$ a $a = a \cdot 1 + 0 \cdot 0$.
- Pokud $b > 0$: Předpokládáme, že funkce vrátí (d', x', y') takové, že $d' = \gcd(b, a \bmod b)$ a $d' = bx' + (a \bmod b)y'$.

  Zjevně je $d = \gcd(a, b) = d' = \gcd(b, a \bmod b)$.

  Dosazením za $(a \bmod b)$ dostáváme

$$d = bx' + (a - \lfloor a/b \rfloor b)y' = ay' + b(x' - \lfloor a/b \rfloor y')$$

  takže $x = y'$ a $y = x' - \lfloor a/b \rfloor y'$.

Rozšířený Eukleidův algoritmus

Tvrzení

Pokud $a > 0$ a $b > 0$, pak EXTENDED-EUCLID(a, b) vrátí d, x, y taková, že $|x| < b/d$ a $|y| < a/d$.

Důkaz: Indukcí podle počtu rekurzivních volání.

Modulární lineární rovnice

Problém

Vstup: Celá čísla a, b, n, kde $a > 0$ a $n > 0$.

Výstup: Všechna x taková, že $ax \equiv b \pmod{n}$.

Poznámka: Řešení nemusí existovat nebo může existovat jedno nebo více řešení.

Protože v $\mathbb{Z}_n$ je $\langle a \rangle = \{a^{(x)} : x > 0\} = \{ax \bmod n : x > 0\}$, řešení existuje právě když $b \in \langle a \rangle$.

Modulární lineární rovnice

Věta

Pro libovolná celá kladná čísla a a n platí v $\mathbb{Z}_n$, že $\langle a \rangle = \langle d \rangle = \{0, d, 2d, \ldots, ((n/d) - 1)d\}$, kde $d = \gcd(a, n)$, a tedy $|\langle a \rangle| = n/d$.

Důkaz: Platí $d \in \langle a \rangle$, protože $\textsc{Extended-Euclid}(a, n)$ vrátí x', y' taková, že $ax' + ny' = d$, takže $ax' \equiv d \pmod{n}$.

Z $d \in \langle a \rangle$ plyne, že $\langle a \rangle$ obsahuje $\{0, d, 2d, \ldots, ((n/d) - 1)d\}$, takže $\langle d \rangle \subseteq \langle a \rangle$.

Ukážeme, že $\langle a \rangle \subseteq \langle d \rangle$. Jestliže $m \in \langle a \rangle$, pak $m = ax \bmod n$ pro nějaké x, neboli $m = ax + ny$ pro nějaké y. Ovšem z $d \mid a$ a $d \mid n$ plyne $d \mid m$, takže $m \in \langle d \rangle$.

Modulární lineární rovnice

Důsledek

Rovnice $ax \equiv b \pmod{n}$ má řešení právě když $\gcd(a, n) \mid b$.

Důsledek

Pokud má rovnice $ax \equiv b \pmod{n}$ řešení, pak má d různých řešení modulo n.

Důkaz: Sekvence $ai \bmod n$, kde $i = 0, 1, 2, \ldots$ je periodická s periodou $\mathrm{ord}(a) = |\langle a \rangle| = n/d$.

Jestliže $b \in \langle a \rangle$, pak se b vyskytne v posloupnosti $ai \bmod n$, kde $i = 0, 1, 2, \ldots, n-1$ právě d krát.
(Indexy i, kde $ai \bmod n = b$ jsou řešeními.)

Modulární lineární rovnice

Tvrzení

Pokud $d = \gcd(a, n)$, $d = ax' + ny'$ a $d \mid b$, pak jedno z řešení rovnice $ax \equiv b \pmod{n}$ je $x_0 = x'(b/d) \bmod n$.

Důkaz:

$$\begin{aligned} ax_0 &\equiv ax'(b/d) && \pmod{n} \\ &\equiv d(b/d) && \pmod{n} \qquad (\text{protože } ax' \equiv d \pmod{n}) \\ &\equiv b && \pmod{n} \end{aligned}$$

Modulární lineární rovnice

Tvrzení

Pokud nějaké x_0 je řešením rovnice $ax \equiv b \pmod{n}$, pak $x_i = x_0 + i(n/d)$, kde $i = 0, 1, \ldots, d-1$ a $d = \gcd(a, n)$, je d různých řešení této rovnice.

Důkaz: Je zřejmé, že hodnoty $i(n/d)$, a tedy i hodnoty x_i jsou navzájem různé modulo n pro $i = 0, 1, \ldots, d-1$.

Navíc platí

$$\begin{aligned} ax_i \bmod n &= a(x_0 + i(n/d)) \bmod n \\ &= (ax_0 + ain/d) \bmod n \\ &= ax_0 \bmod n \qquad (\text{protože } d \mid a) \\ &= b \end{aligned}$$

Modulární lineární rovnice

Algoritmus

MODULAR-LINEAR-EQUATION-SOLVER(a, b, n)

```
1  (d, x', y') ← EXTENDED-EUCLID(a, n)
2  if d | b
3      then x_0 ← x'(b/d) mod n
4           for i ← 0 to d − 1
5               do print (x_0 + i(n/d)) mod n
6      else  print "no solutions"
```

Modulární lineární rovnice

Důsledek

Pokud $\gcd(a, n) = 1$, pak má rovnice $ax \equiv b \pmod{n}$ právě jedno řešení modulo n.

Důsledek

Pokud $\gcd(a, n) = 1$, pak má rovnice $ax \equiv 1 \pmod{n}$ právě jedno řešení modulo n, a v opačném případě nemá žádné řešení.

Poznámka: Řešení rovnice $ax \equiv 1 \pmod{n}$ označujeme $(a^{-1} \bmod n)$.

Můžeme ho spočítat pomocí EXTENDED-EUCLID, protože $\gcd(a, n) = 1 = ax + ny$ znamená, že $ax \equiv 1 \pmod{n}$.

Problém „Marbles“ (10090)

Máme n skleněnek a chceme nakoupit krabice, do kterých je uložíme.

Krabice jsou dvou typů:

- jeden typ stojí c_1 Kč a vejde se do něj n_1 skleněnek,
- druhý typ stojí c_2 Kč a vejde se do něj n_2 skleněnek.

Nakoupené krabice musí být zcela zaplněny.

Problém

Vstup: Přirozená čísla n, n_1, n_2, c_1, c_2 (všechna menší než 2^{31}).

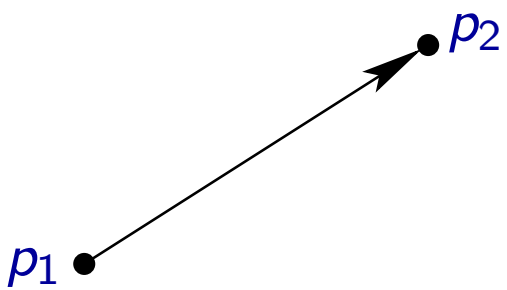
Výstup: Počet krabic prvého a druhého typu, tak aby zaplacená částka byla minimální, případně odpověď, že řešení neexistuje.

Výpočetní geometrie

Bod v rovině (2-D) je zadán dvojicí souřadnic (x, y), bod v prostoru (3-D) trojicí souřadnic (x, y, z).

Dvojice bodů $p_1 = (x_1, y_1, z_1)$ a $p_2 = (x_2, y_2, z_2)$ tvoří vektor $\overrightarrow{p_1 p_2}$, jehož souřadnice jsou (x, y, z), kde $x = x_2 - x_1$, $y = y_2 - y_1$ a $z = z_2 - z_1$.

Poznámka: Stučněji zapisujeme $\overrightarrow{p_1 p_2} = p_2 - p_1$.

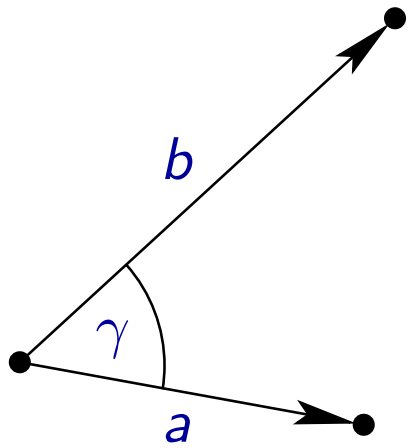


Délka vektoru $p = (x, y, z)$ je $|p| = \sqrt{x^2 + y^2 + z^2}$.

Skalární součin

Skalární součin $a \cdot b$ vektorů $a = (a_x, a_y, a_z)$ a $b = (b_x, b_y, b_z)$ je hodnota

$$a \cdot b = a_x b_x + a_y b_y + a_z b_z$$



Platí $a \cdot b = |a| \cdot |b| \cdot \cos\gamma$, z čehož plyne $\cos\gamma = \frac{a \cdot b}{|a| \cdot |b|}$.

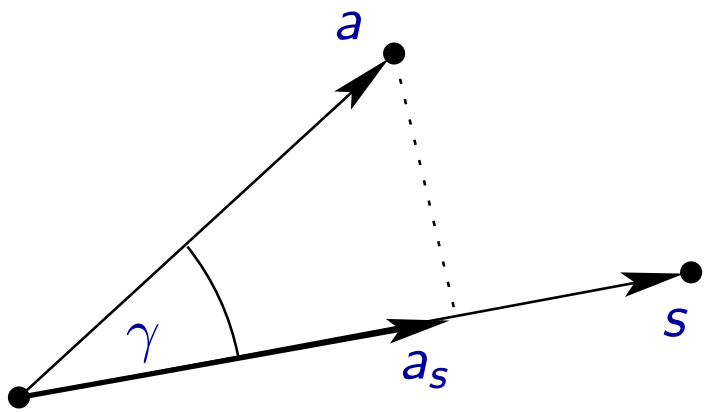
Vektory a a b jsou navzájem kolmé právě když $a \cdot b = 0$.

Platí $a \cdot a = a^2 = |a|^2$.

Skalární součin

Průmět vektoru a do vektoru s

$$|a_s| = a \cdot \cos\gamma = \frac{a \cdot s}{|s|}$$



Platí

$$a_s = s \cdot \frac{|a_s|}{|s|} = s \cdot \frac{a \cdot s}{|s|^2} = s \cdot \frac{a \cdot s}{s^2}$$

Vektorový součin

Vektorový součin $a \times b$ vektorů $a = (a_x, a_y, a_z)$ a $b = (b_x, b_y, b_z)$ je vektor $c = (c_x, c_y, c_z)$, kde

$$\begin{aligned} c_x &= a_y b_z - b_y a_z \\ c_y &= a_z b_x - b_z a_x \\ c_z &= a_x b_y - b_x a_y \end{aligned}$$

což lze také zapsat jako

$$a \times b = \det \begin{pmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{pmatrix}$$

kde $i = (1, 0, 0)$, $j = (0, 1, 0)$ a $k = (0, 0, 1)$.

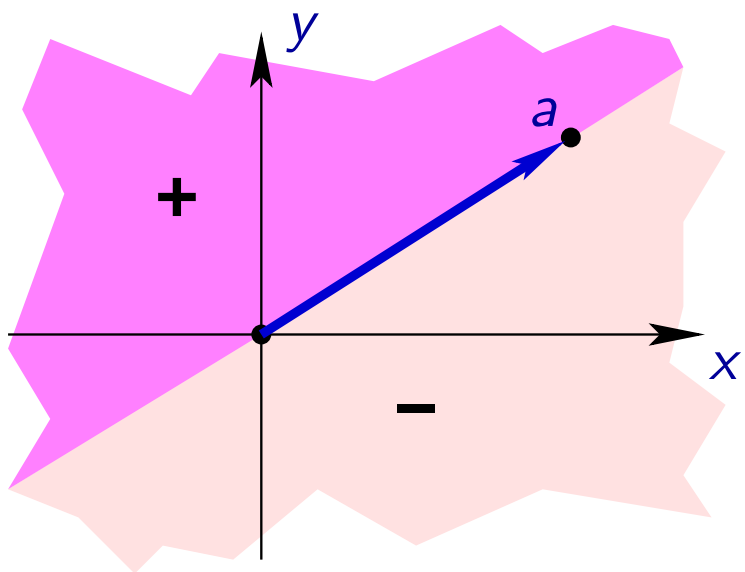
Vektor c je kolmý k vektorům a i b, a platí $|c| = |a| \cdot |b| \cdot \sin \gamma$.

Vektorový součin

Speciálně v rovině, tj. pokud $a_z = 0$ a $b_z = 0$, platí $c_x = 0$, $c_y = 0$ a

$$c_z = \det \begin{pmatrix} a_x & a_y \\ b_x & b_y \end{pmatrix} = a_x b_y - b_x a_y$$

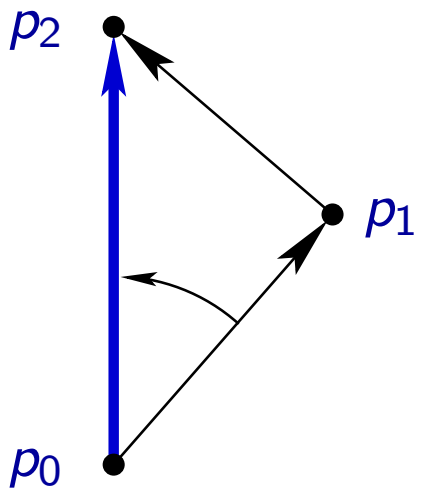
- Pokud $c_z > 0$, pak se b nachází proti směru hodinových ručiček od a.
- Pokud $c_z < 0$, pak se b nachází ve směru hodinových ručiček od a.
- Pokud $c_z = 0$, pak a a b směřují buď stejným nebo opačným směrem.

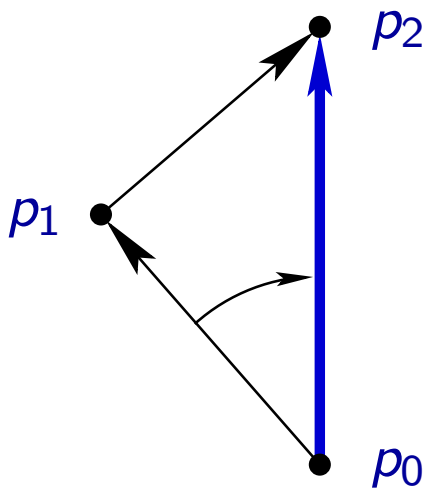


Vektorový součin

Pro dvojici po sobě jdoucích vektorů $\overrightarrow{p_0p_1}$ a $\overrightarrow{p_1p_2}$ můžeme snadno určit, zda druhý z nich zatáčí doleva nebo doprava:

Stačí spočítat vektorový součin $(p_2 - p_0) \times (p_1 - p_0)$.



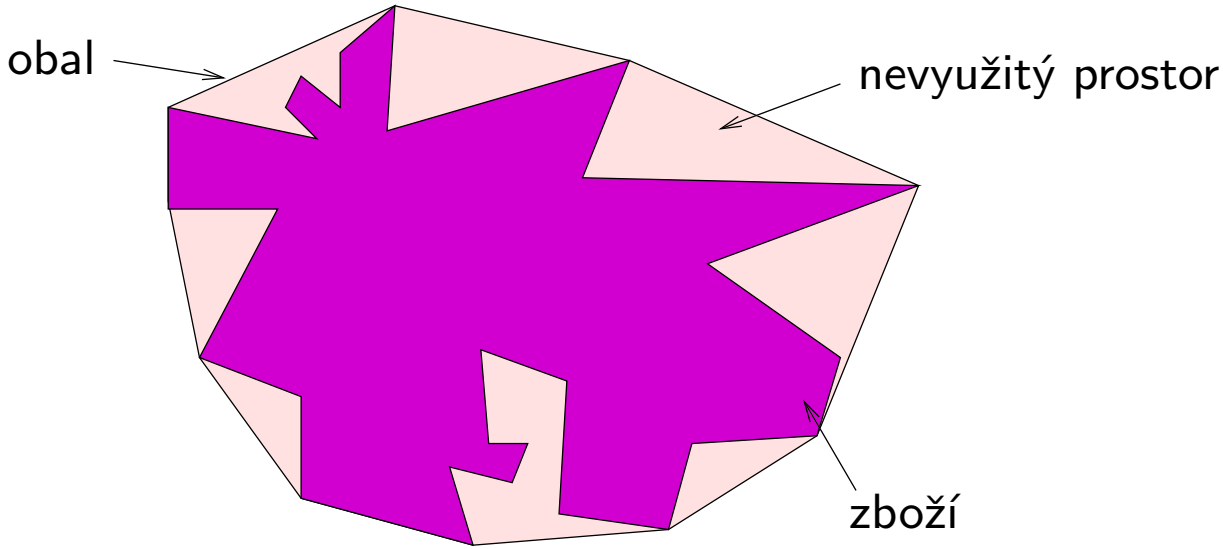


Výpočet vzdálenosti d bodu p_2 od přímky p_0p_1:

$$d = \frac{|\overrightarrow{p_0p_1} \times \overrightarrow{p_0p_2}|}{|\overrightarrow{p_0p_1}|}$$

Problém „Useless Tile Packers“ (10065)

Balíme nějaké zboží do obalu a chceme spočítat kolik procent tvoří nevyužitý prostor.

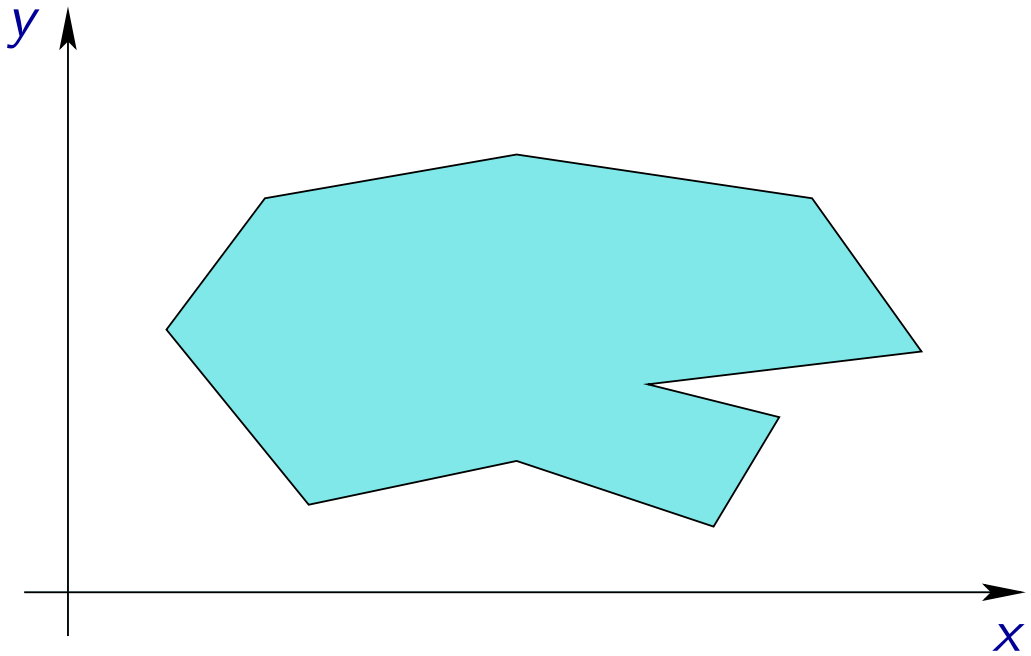


Problém

Vstup: Souřadnice krajních bodů zboží.

Výstup: Množtví nevyužitého prostoru v procentech.

Výpočet obsahu mnohoúhelníka

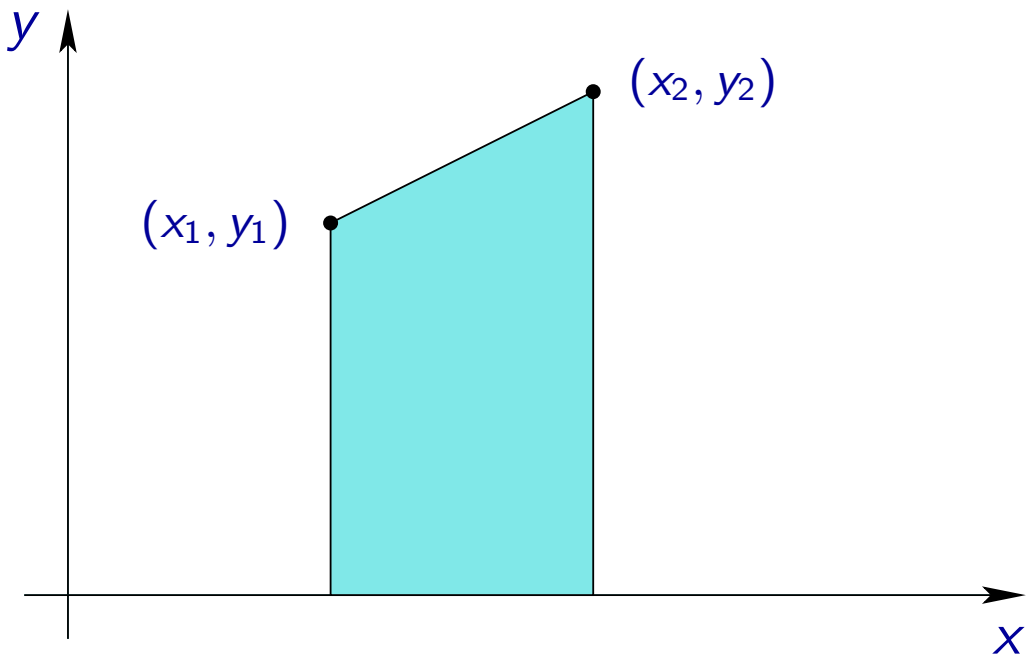


Problém

Vstup: Souřadnice krajních bodů mnohoúhelníka.

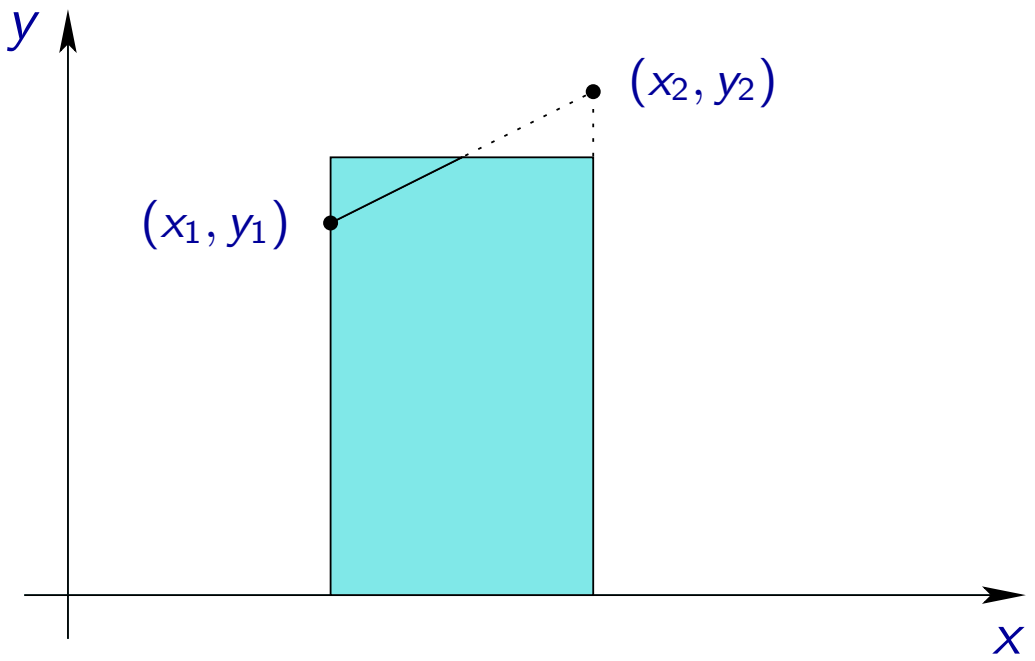
Výstup: Obsah mnohoúhelníka.

Výpočet obsahu mnohoúhelníka



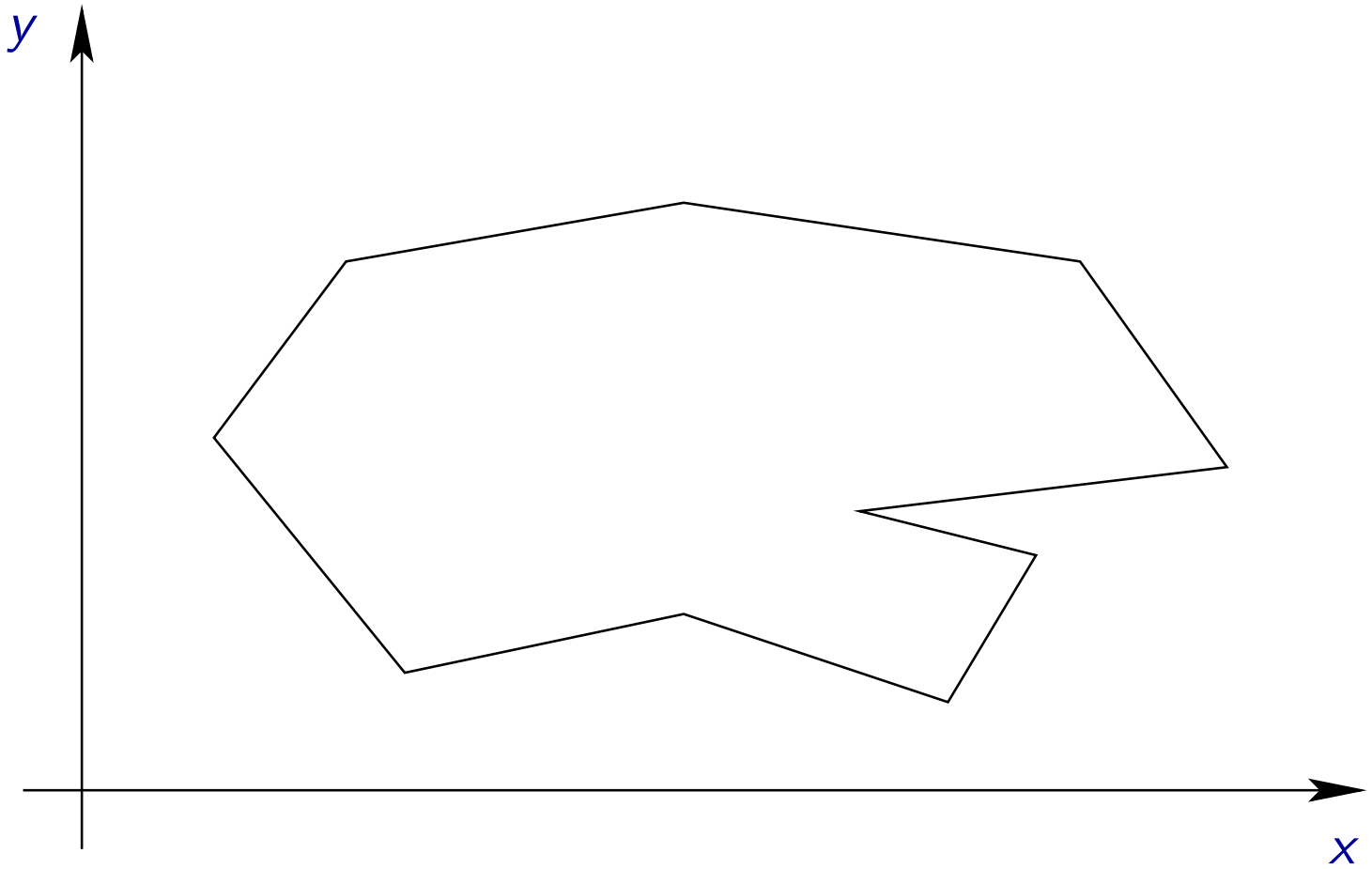
$$S = (x_2 - x_1)\frac{y_1 + y_2}{2}$$

Výpočet obsahu mnohoúhelníka

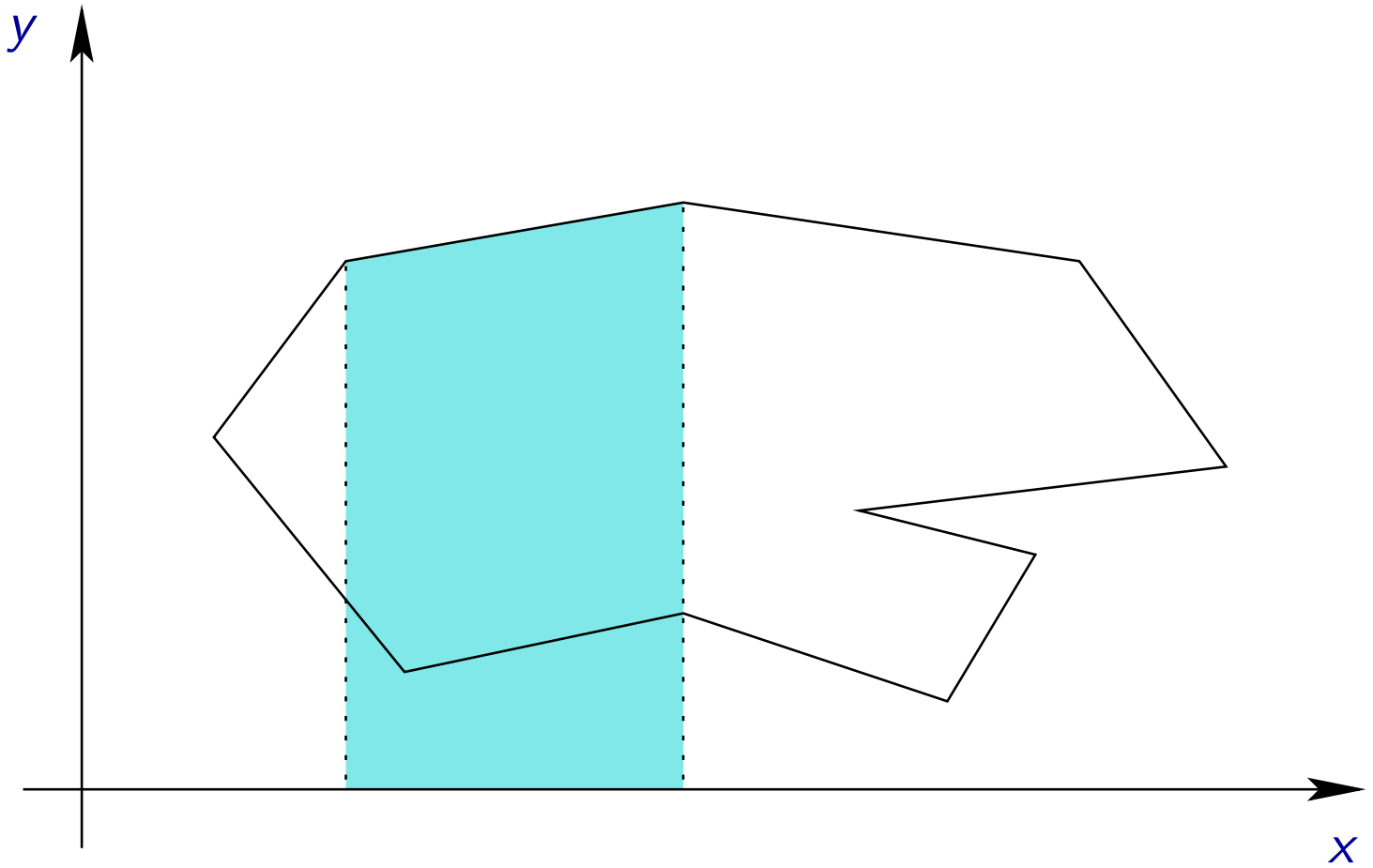


$$S = (x_2 - x_1)\frac{y_1 + y_2}{2}$$

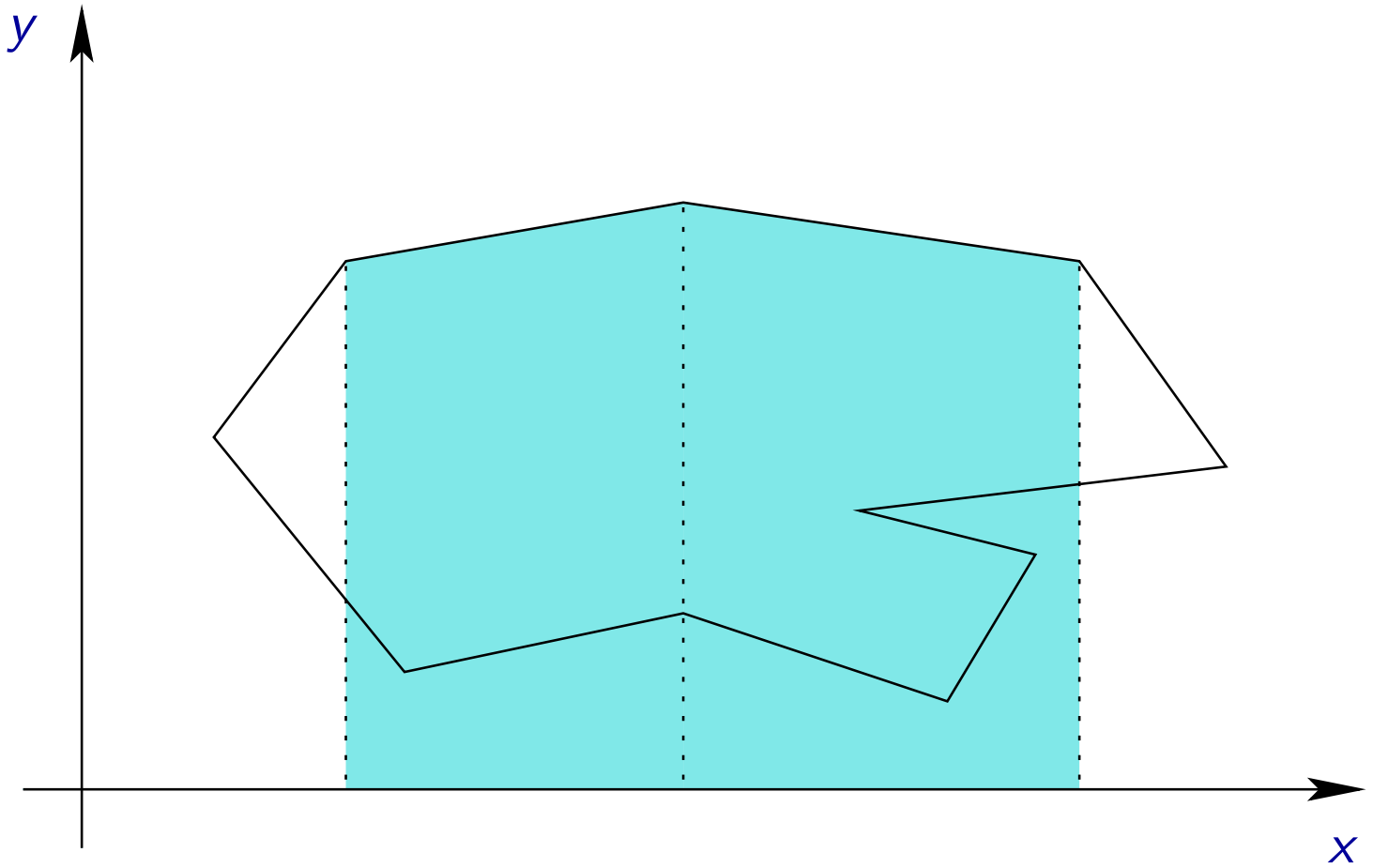
Výpočet obsahu mnohoúhelníka



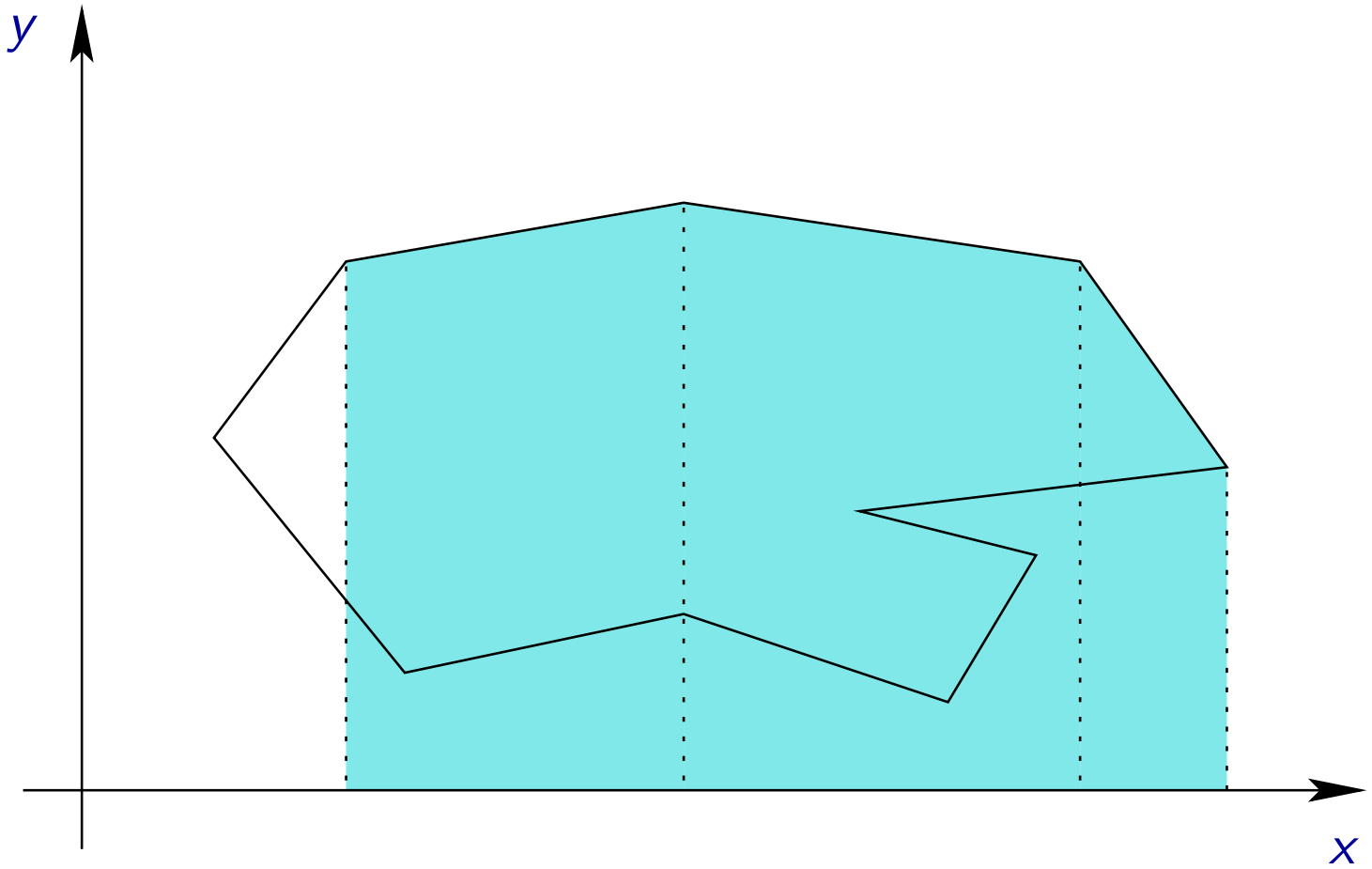
Výpočet obsahu mnohoúhelníka



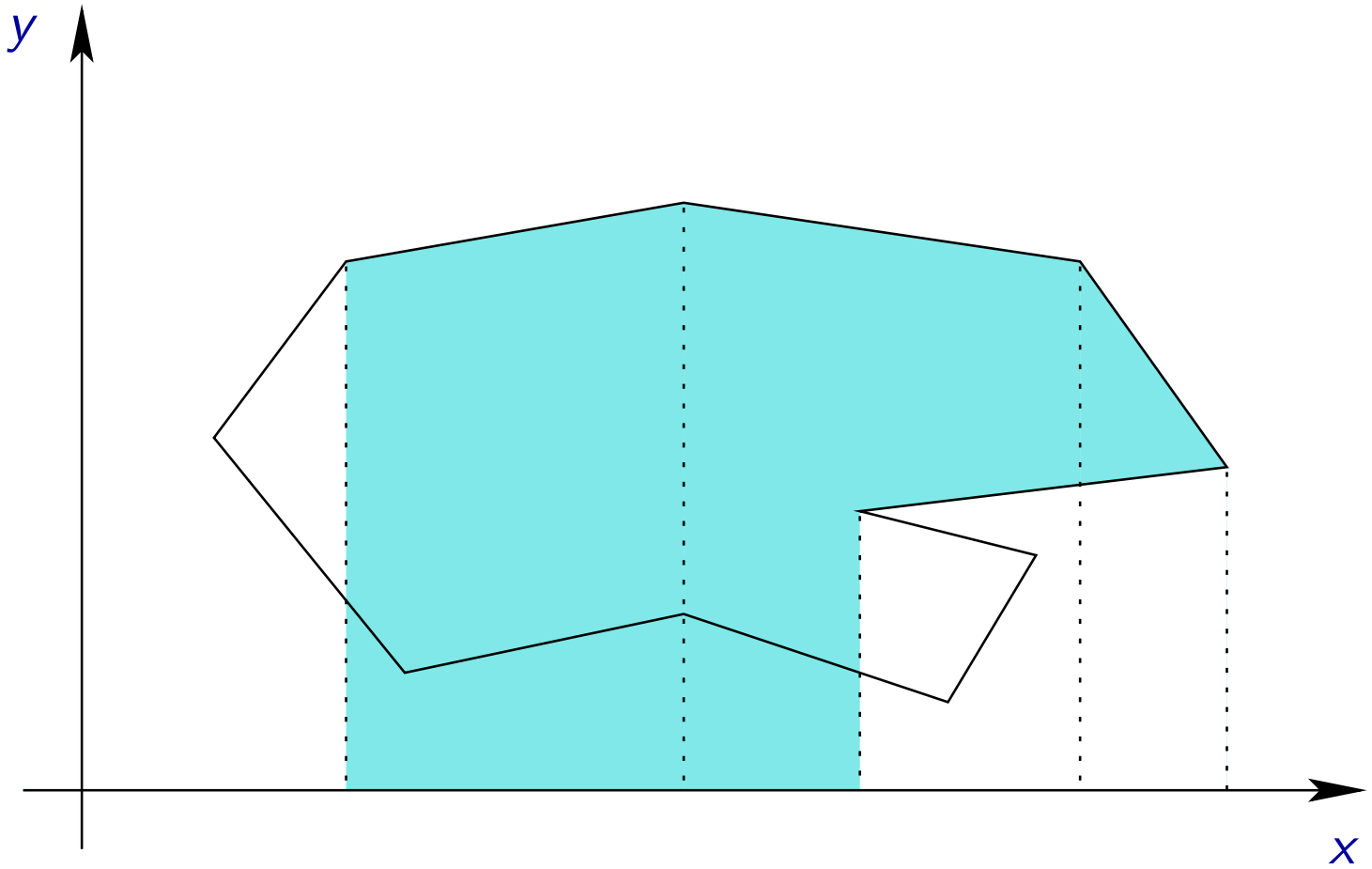
Výpočet obsahu mnohoúhelníka



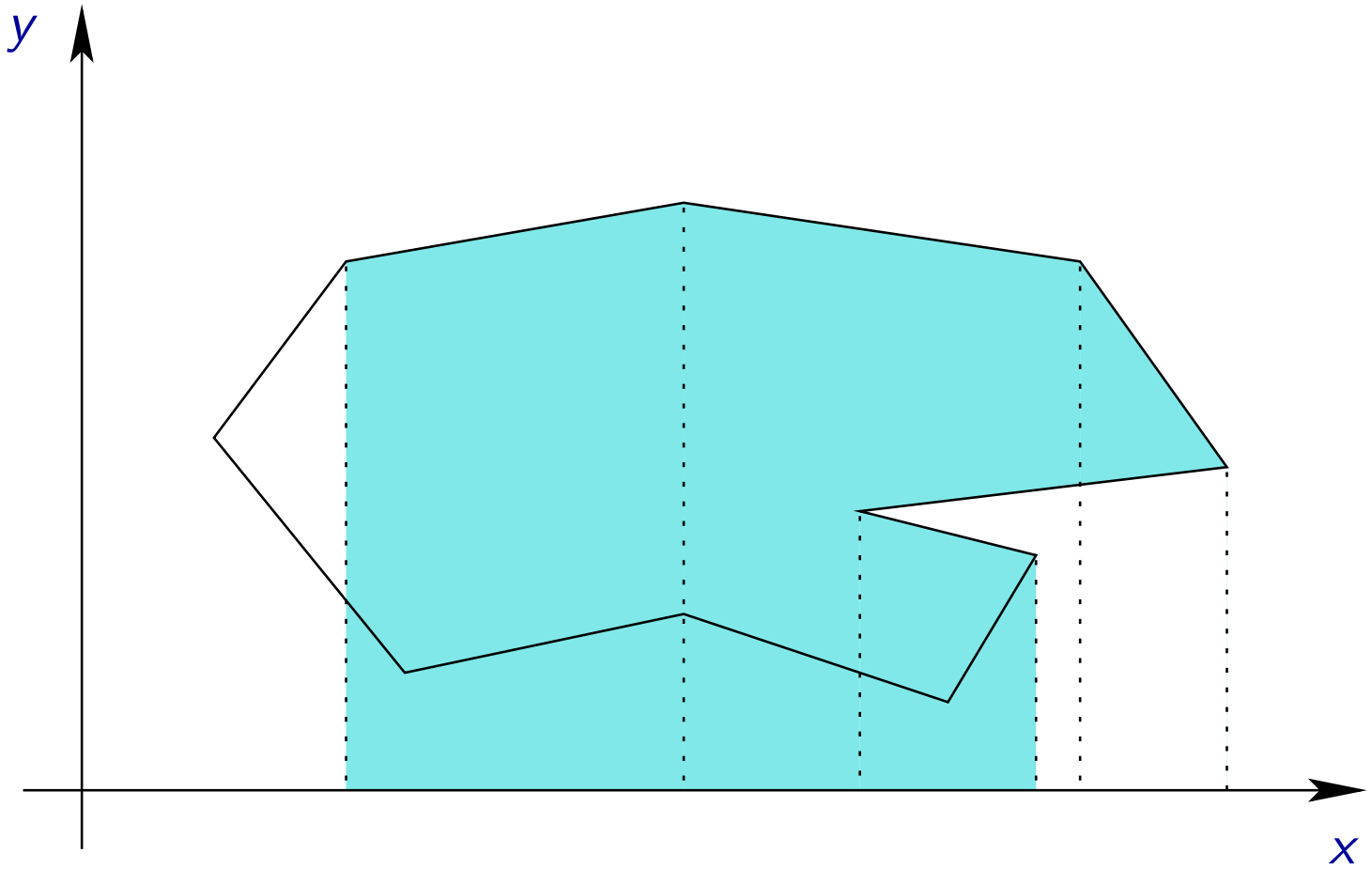
Výpočet obsahu mnohoúhelníka



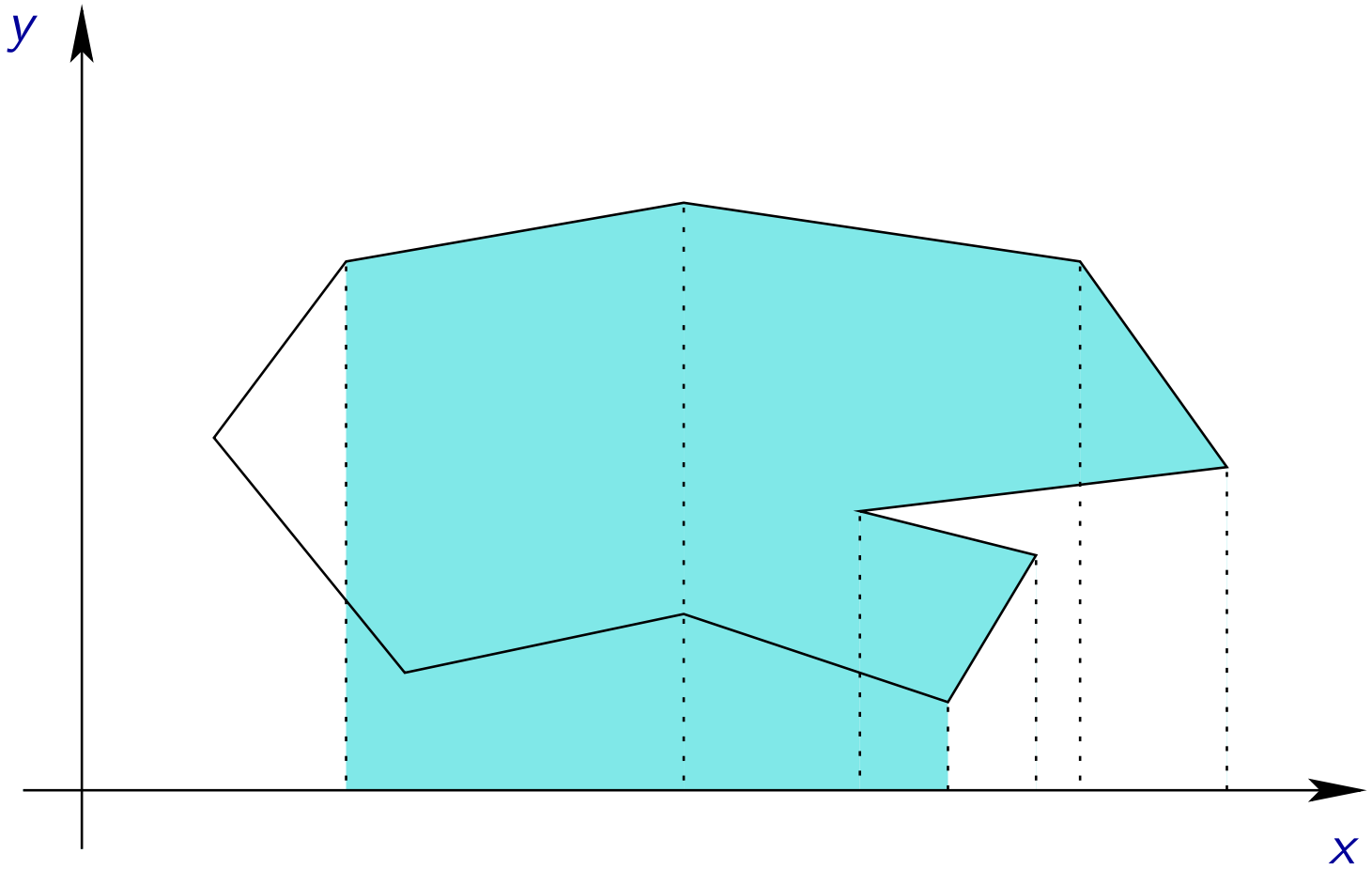
Výpočet obsahu mnohoúhelníka



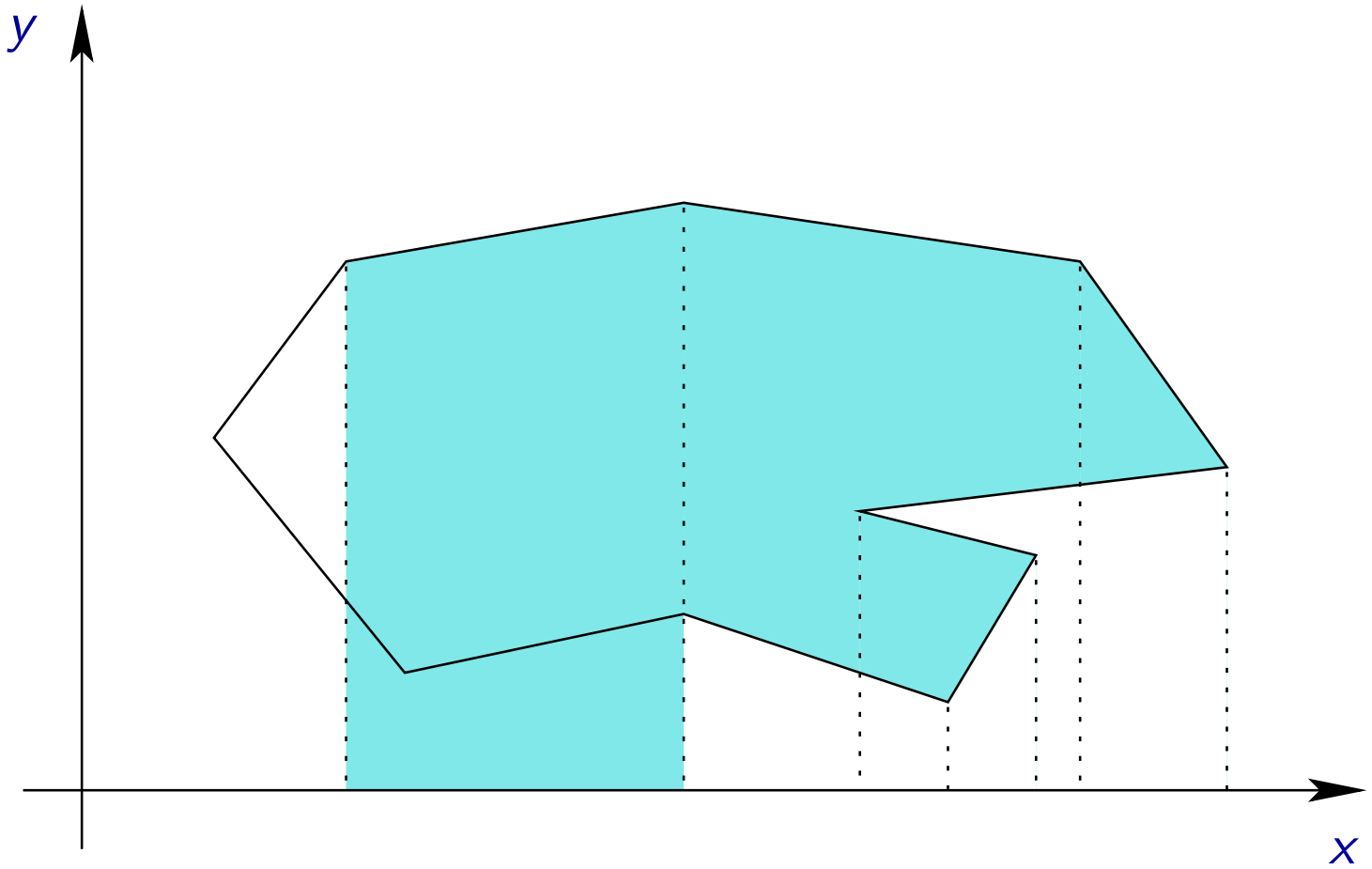
Výpočet obsahu mnohoúhelníka



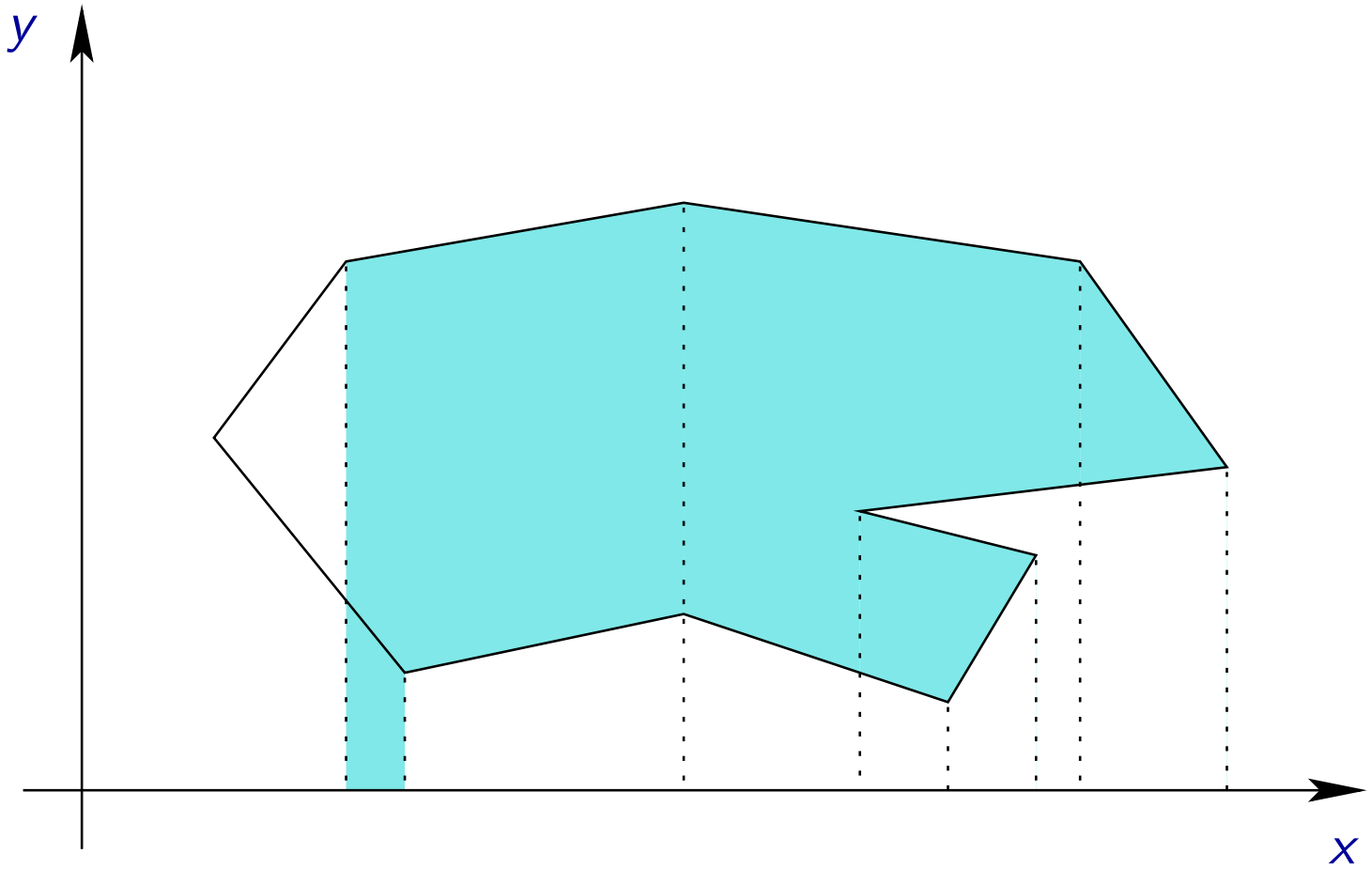
Výpočet obsahu mnohoúhelníka



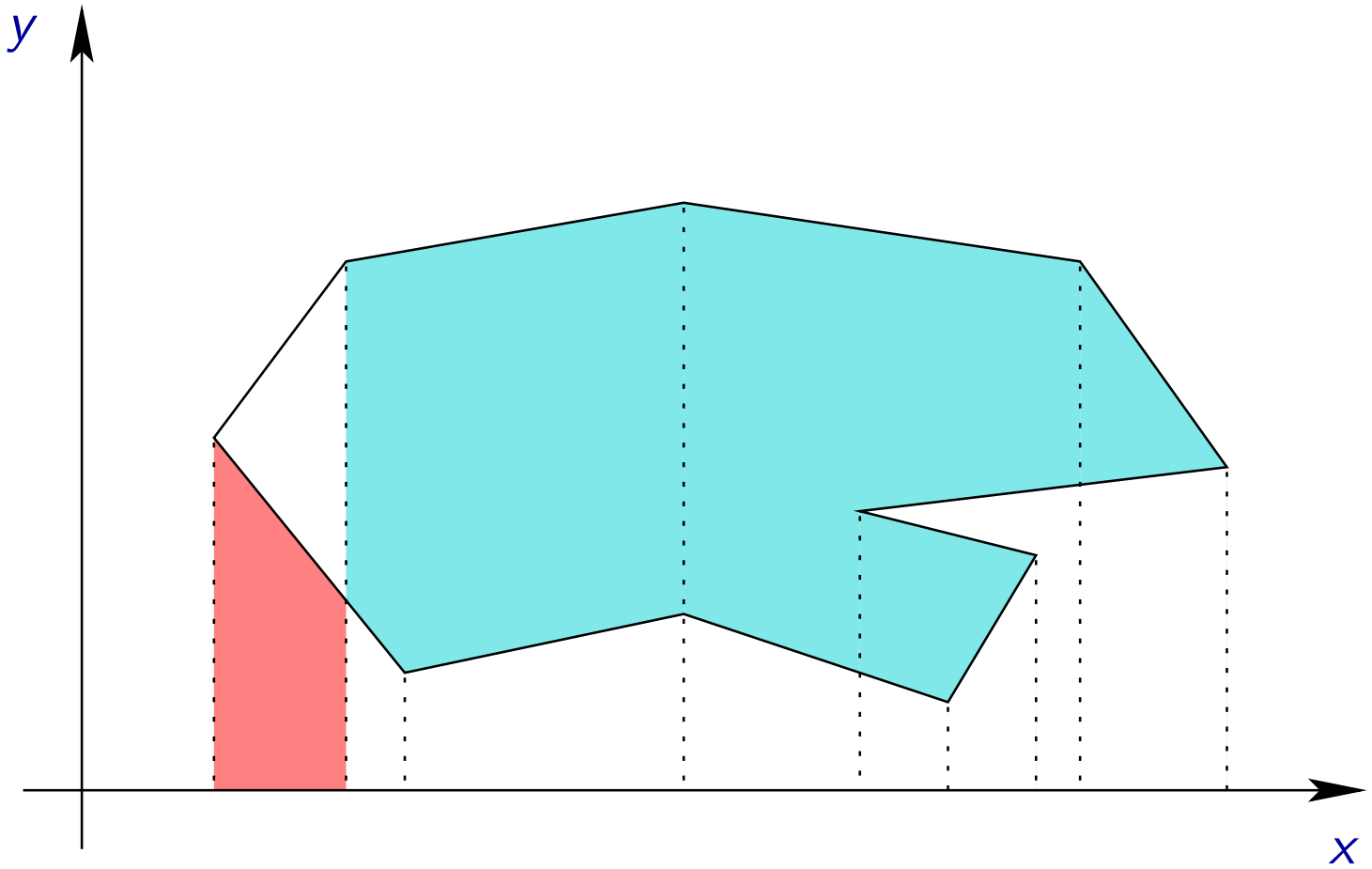
Výpočet obsahu mnohoúhelníka



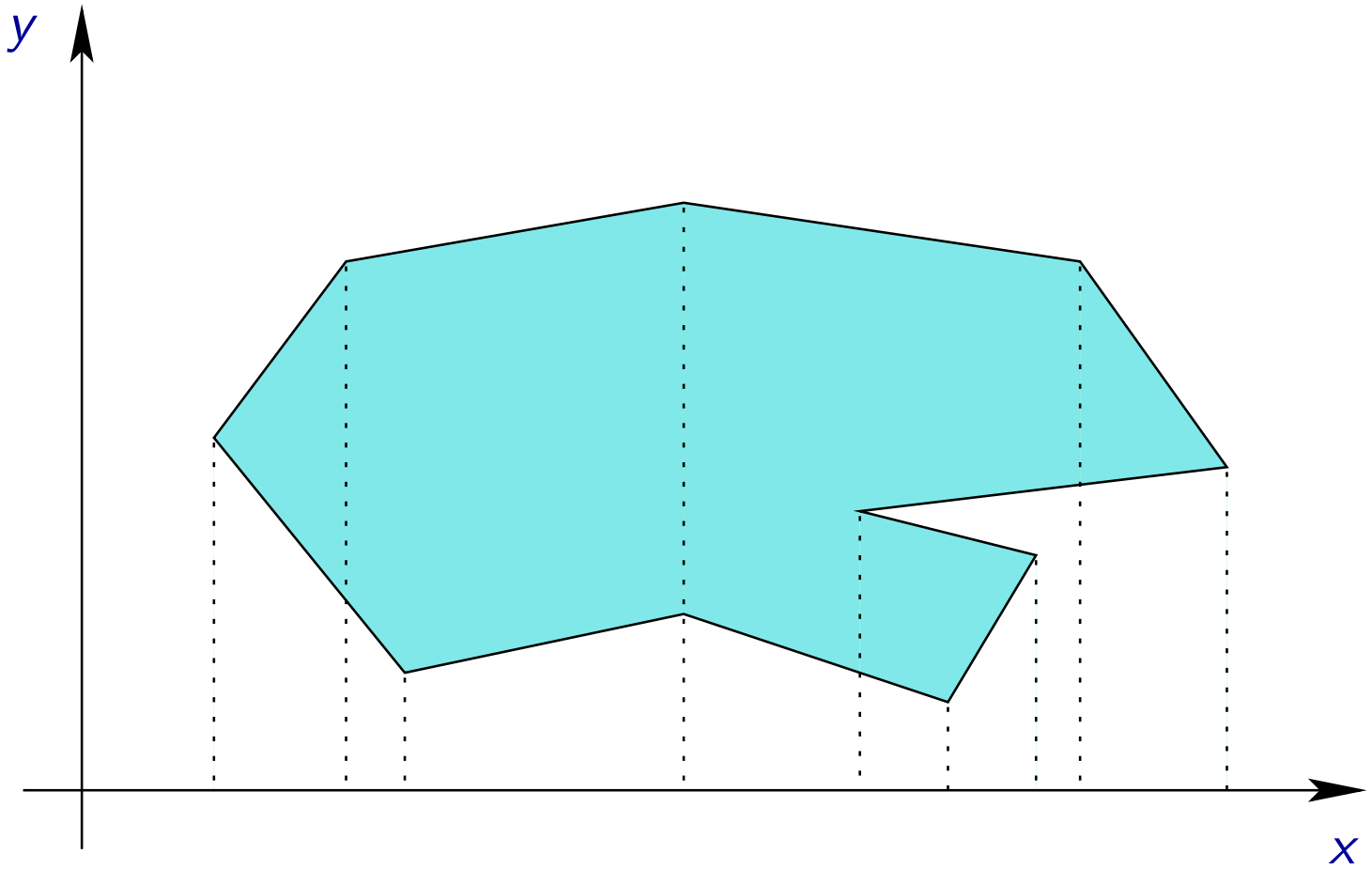
Výpočet obsahu mnohoúhelníka



Výpočet obsahu mnohoúhelníka



Výpočet obsahu mnohoúhelníka



Výpočet obsahu mnohoúhelníka

Algoritmus pro výpočet obsahu mnohoúhelníka:

n – počet bodů

x – pole x-ových souřadnic bodů $1 \ldots n$

y – pole y-ových souřadnic bodů $1 \ldots n$

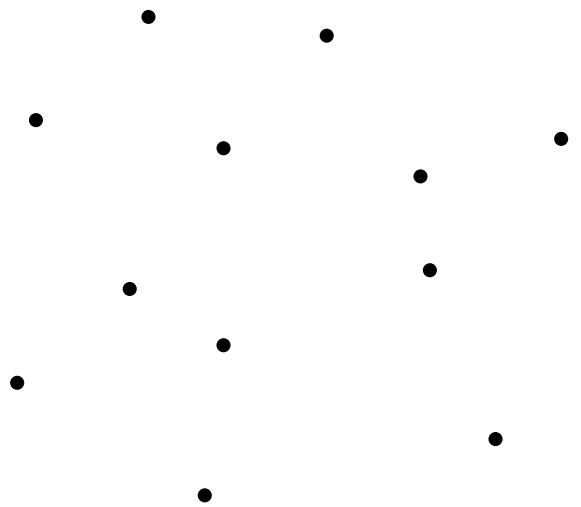
POLYGON-AREA(n, x, y)

1 $S \leftarrow 0$
2 **for** $i = 1$ **to** $n - 1$
3 **do** $S \leftarrow S + (x[i + 1] - x[i]) \cdot (y[i] + y[i + 1])$
4 $S \leftarrow S + (x[1] - x[n]) \cdot (y[n] + y[1])$
5 $S \leftarrow S/2$
6 **return** S

Doba běhu algoritmu je v $O(n)$.

Konvexní obal

Konvexní obal (convex hull) dané množiny bodů Q je nejmenší konvexní mnohoúhelník P takový, že všechny body z Q leží buď na hranici P nebo uvnitř P.



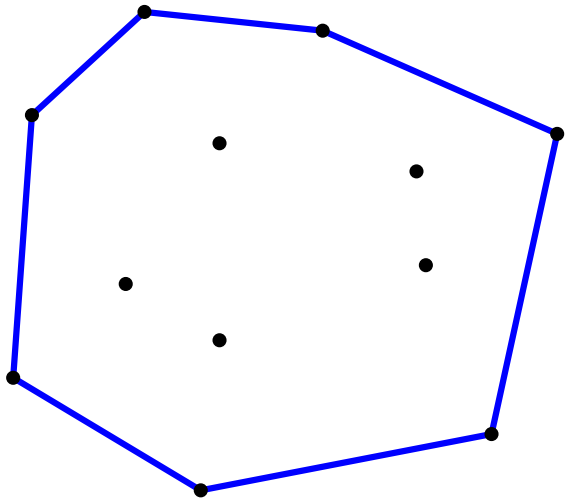
Problém

Vstup: Množina bodů $P = \{p_1, p_2, \ldots, p_n\}$.

Výstup: Konvexní obal množiny bodů P.

Konvexní obal

Konvexní obal (convex hull) dané množiny bodů Q je nejmenší konvexní mnohoúhelník P takový, že všechny body z Q leží buď na hranici P nebo uvnitř P.

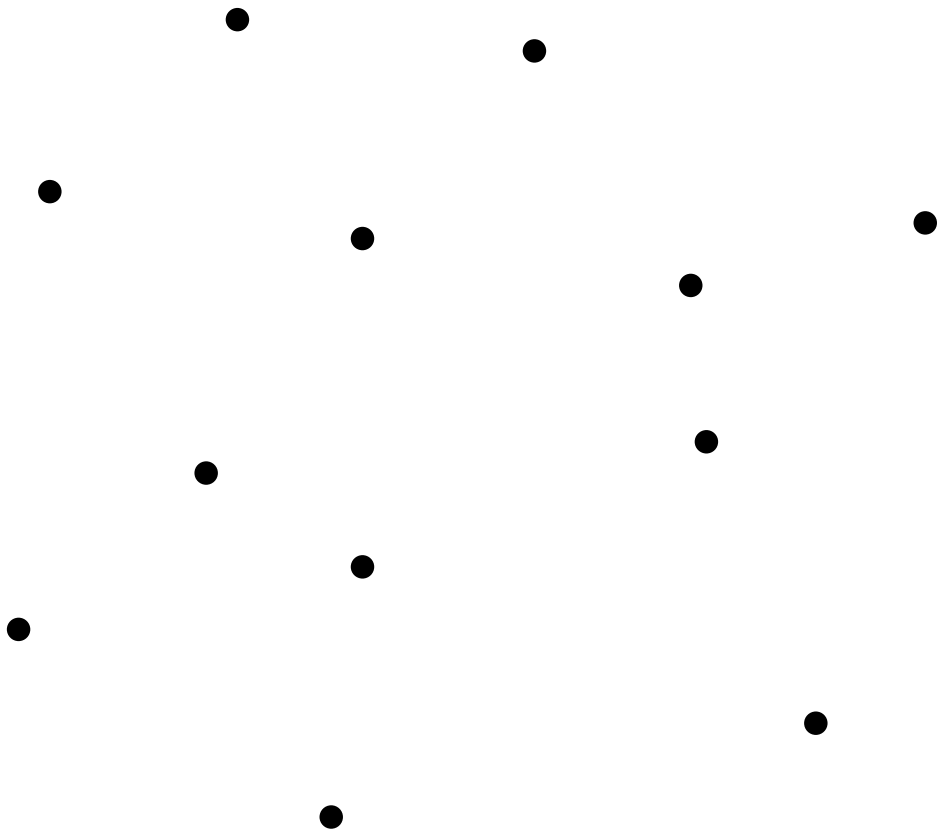


Problém

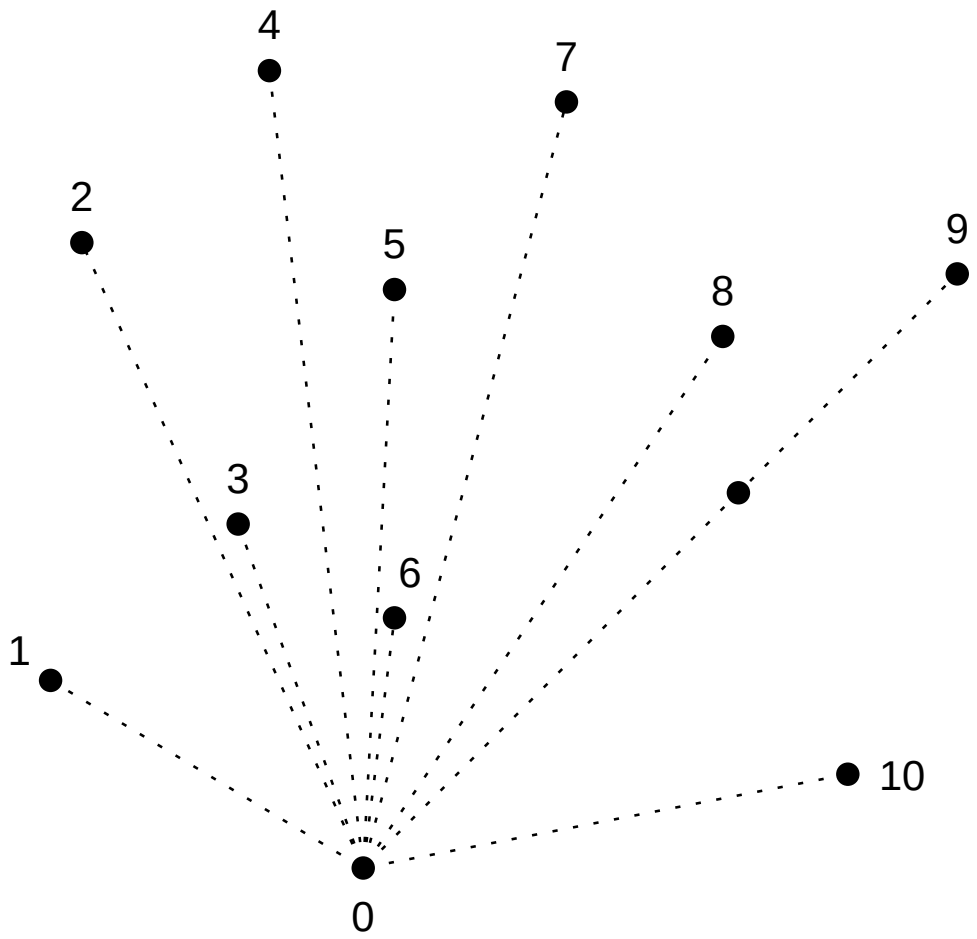
Vstup: Množina bodů $P = \{p_1, p_2, \ldots, p_n\}$.

Výstup: Konvexní obal množiny bodů P.

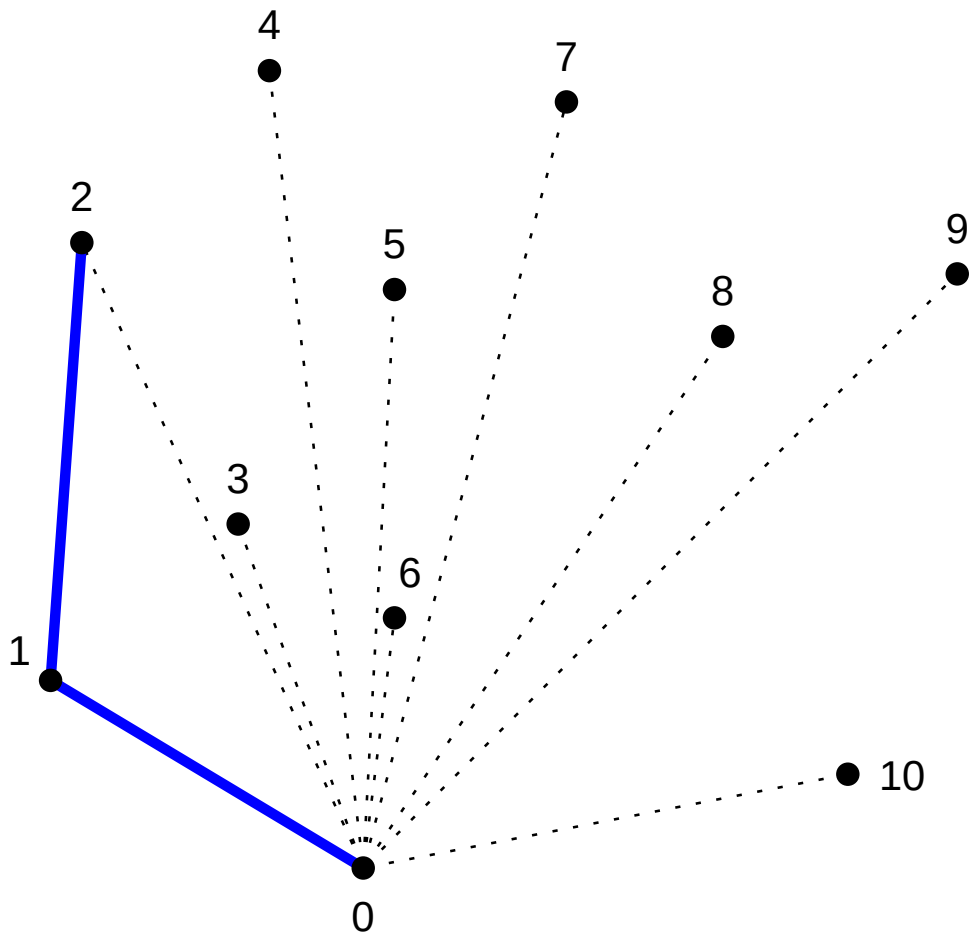
Konvexní obal



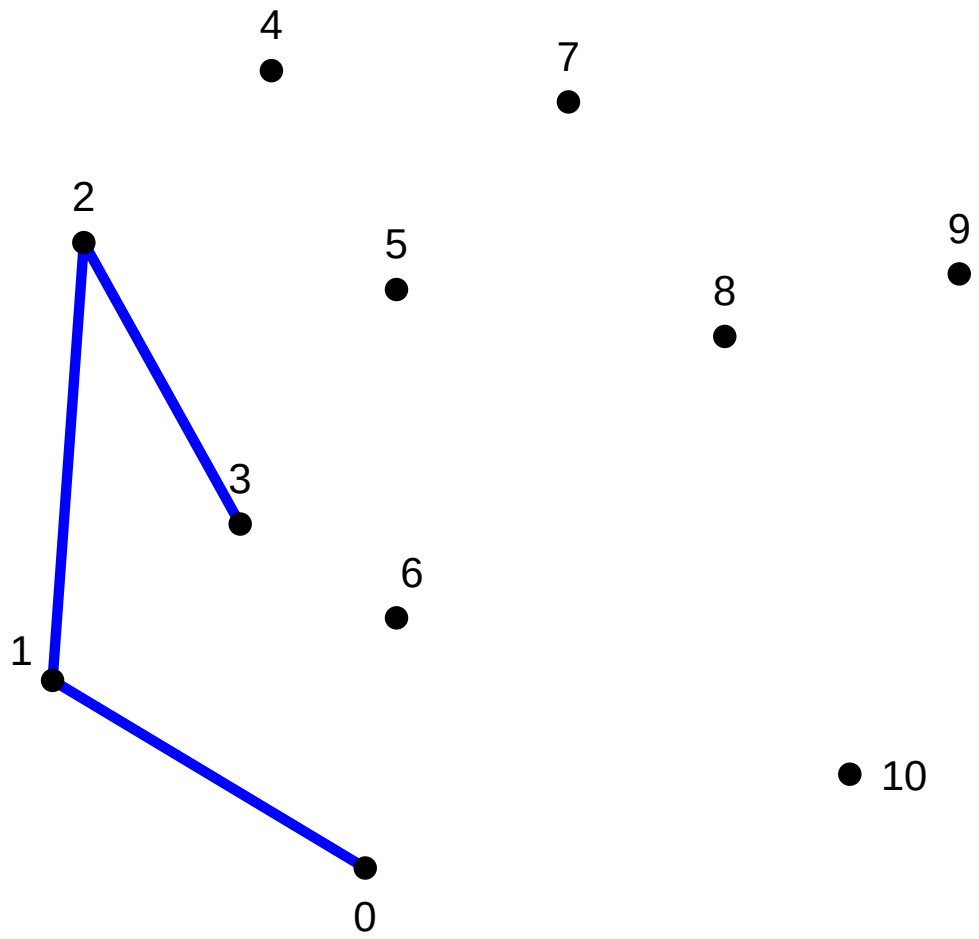
Konvexní obal



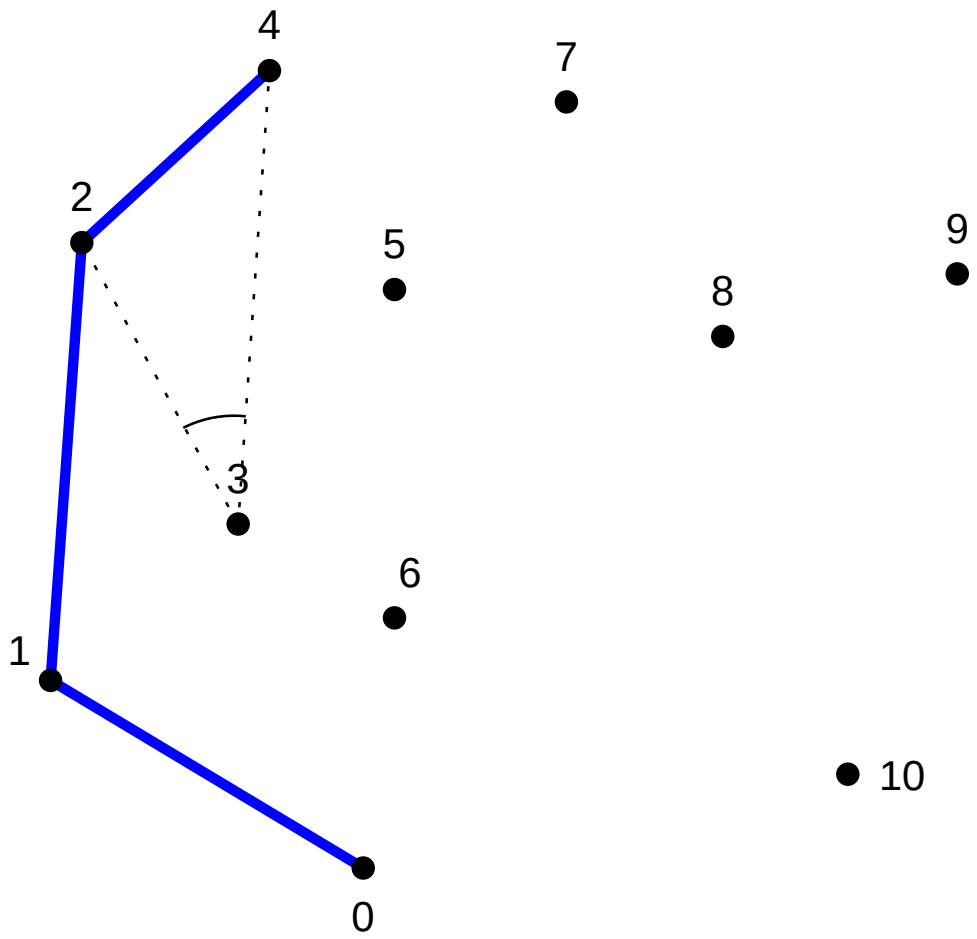
Konvexní obal



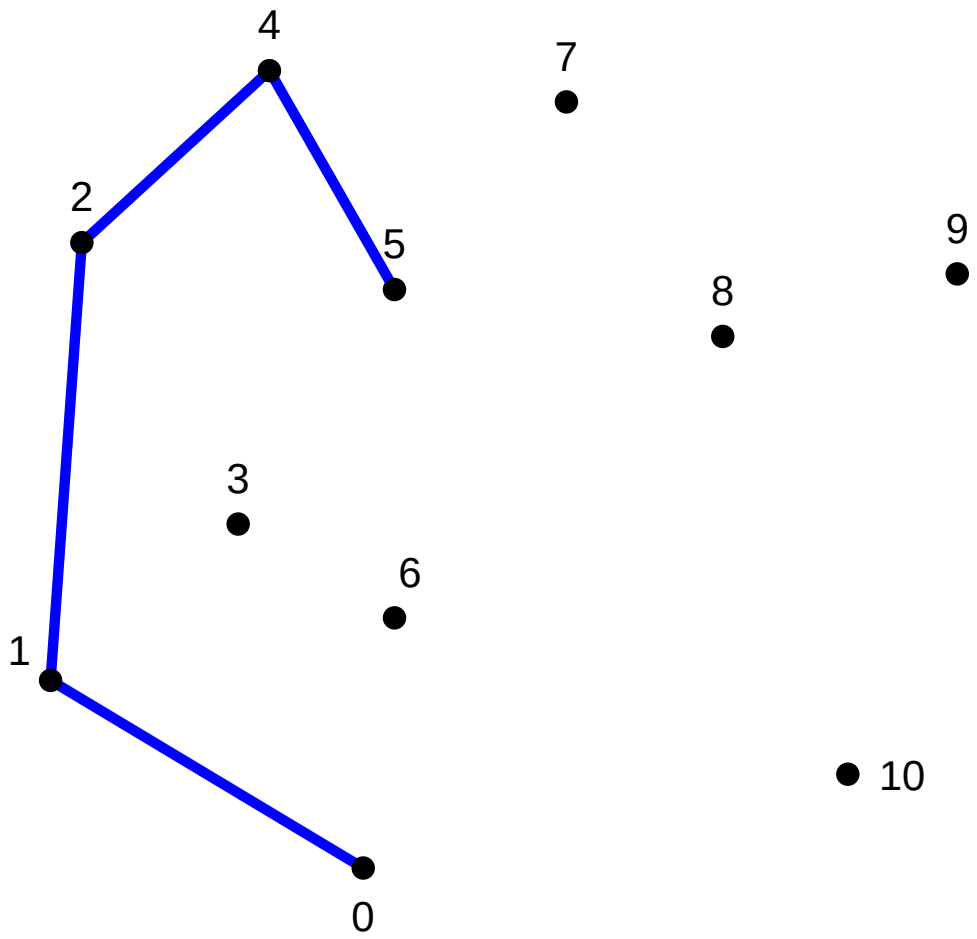
Konvexní obal



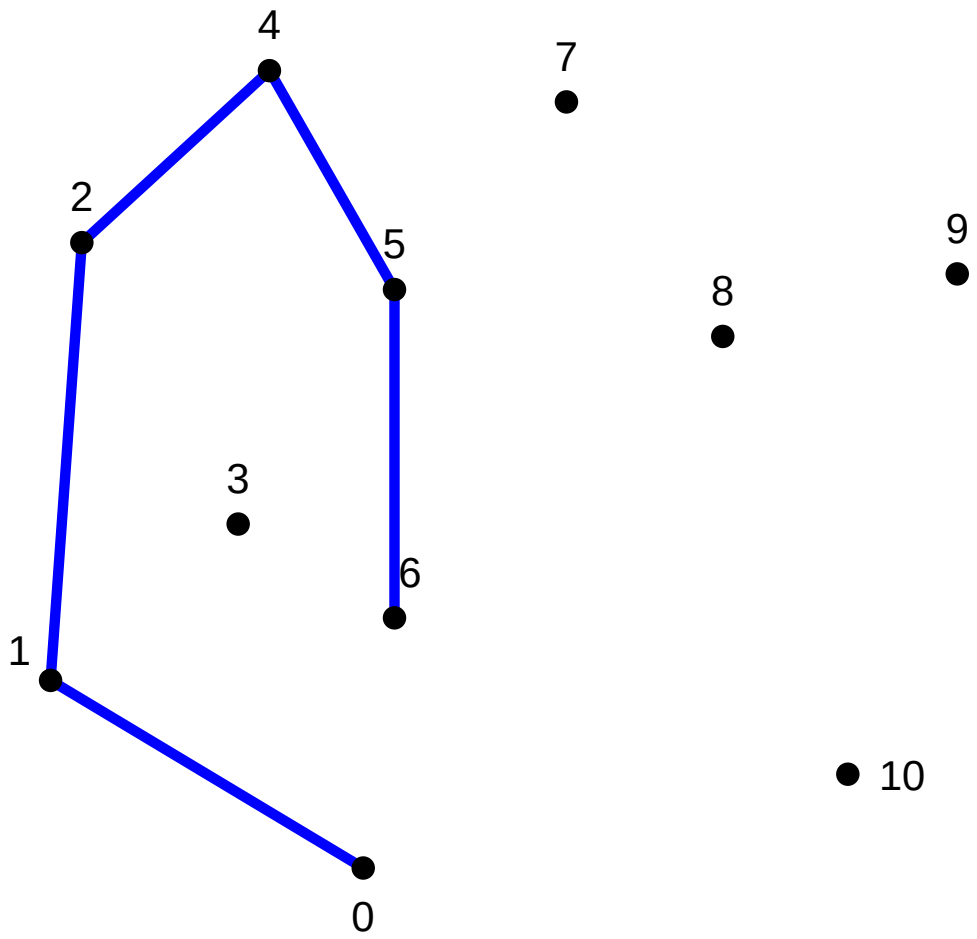
Konvexní obal



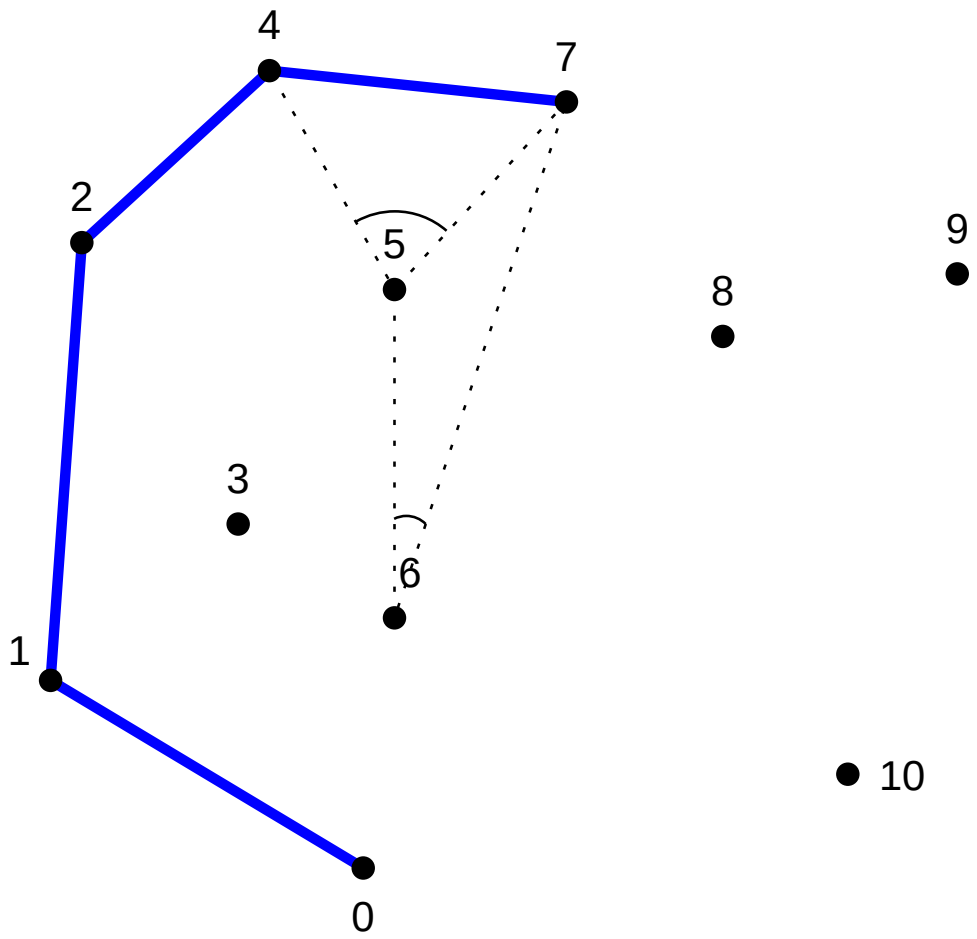
Konvexní obal



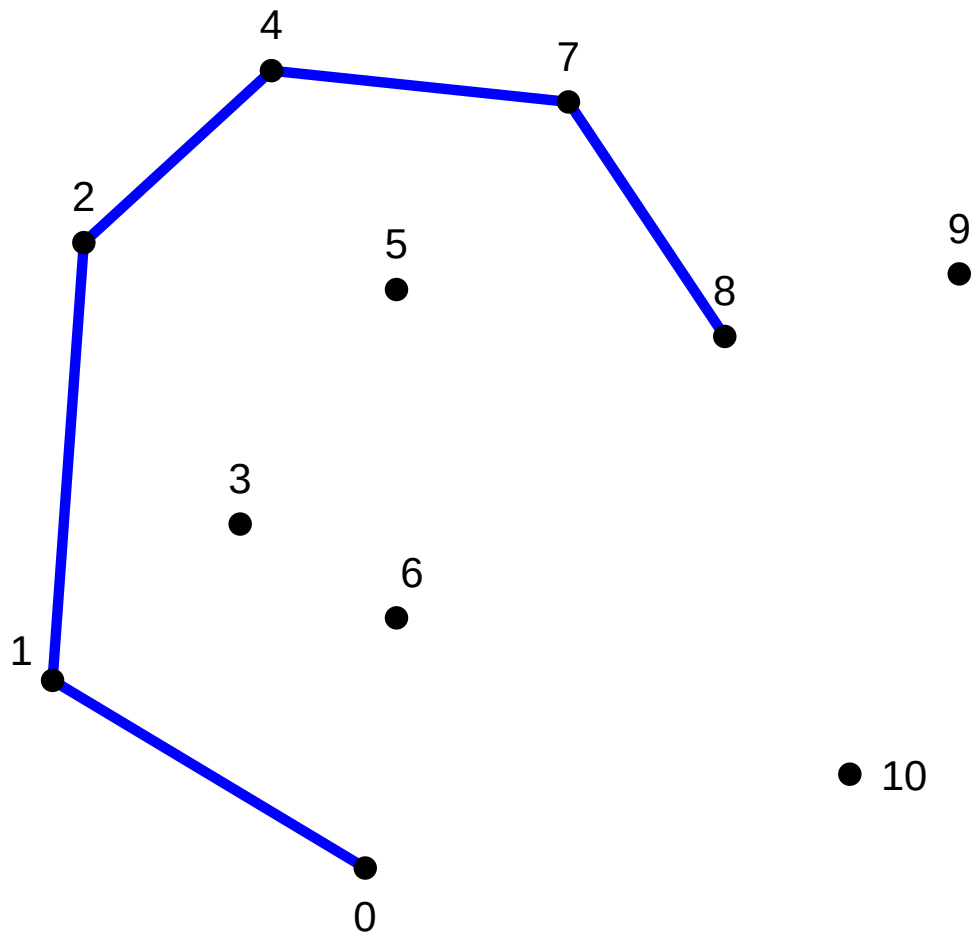
Konvexní obal



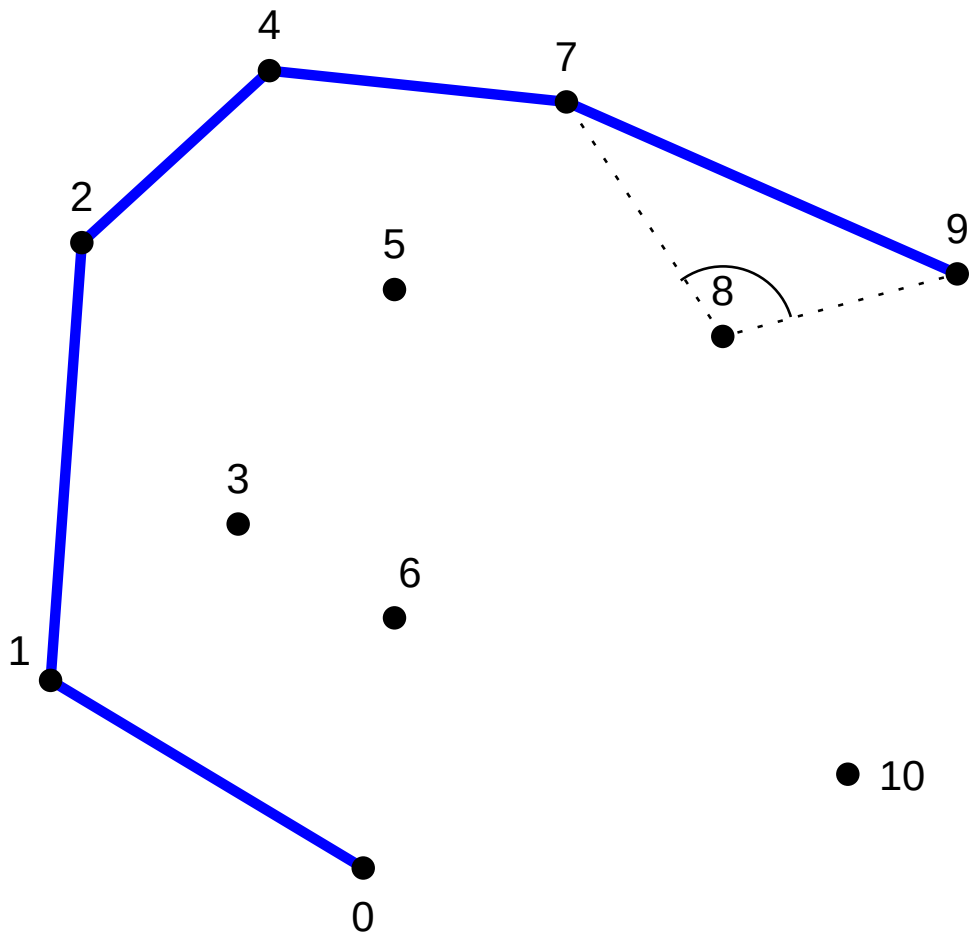
Konvexní obal



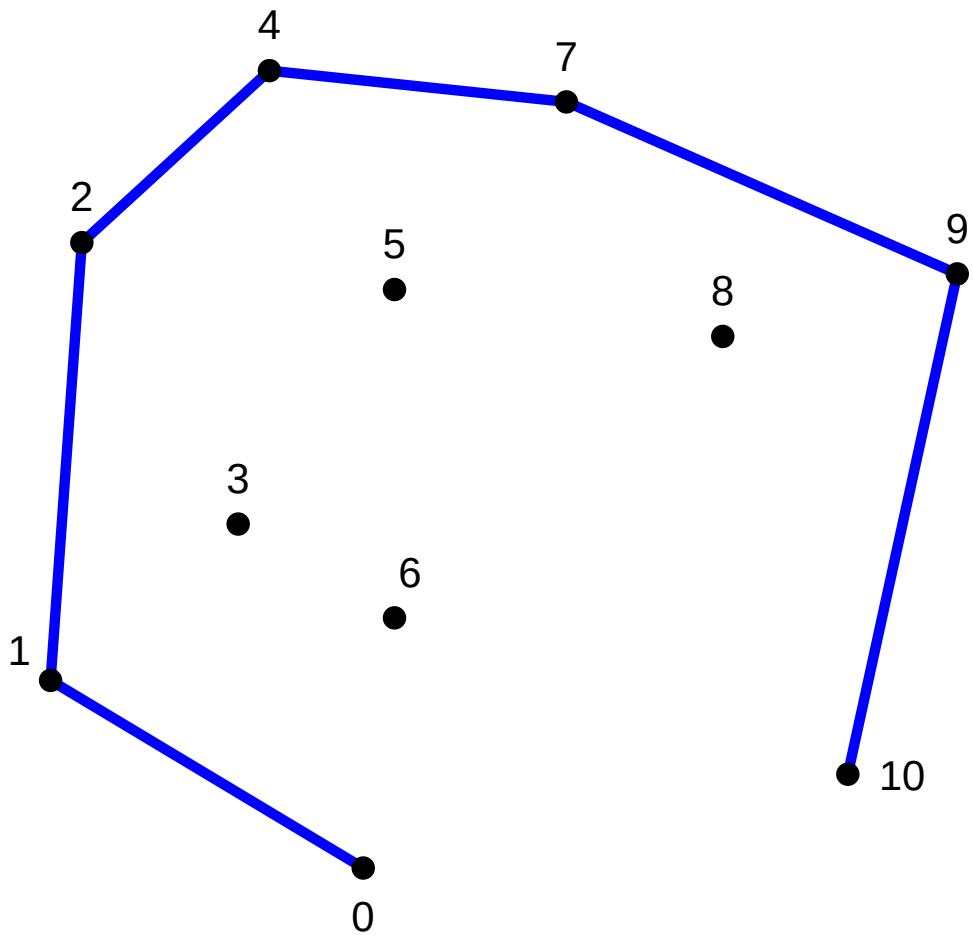
Konvexní obal



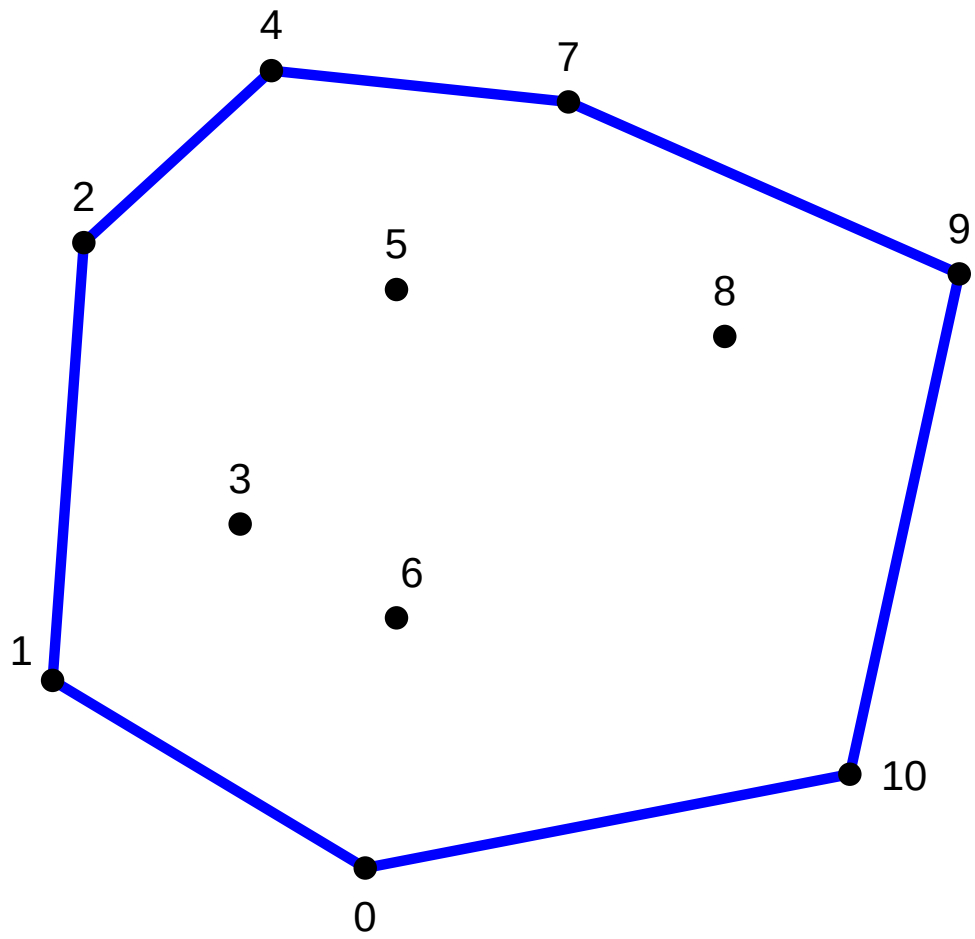
Konvexní obal



Konvexní obal



Konvexní obal



Konvexní obal

GRAHAM-SCAN(Q)

1 nechť p_0 je bod s minimální y-ovou souřadnicí
(a s minimální x-ovou souřadnicí, jestliže existuje víc takových bodů)
2 nechť $\langle p_1, p_2, \ldots, p_m \rangle$ jsou zbývající body z Q seřazené ve směru hodinových ručiček podle úhlů, které svírají vzhledem k bodu p_0
(pokud více bodů svírá stejný úhel, odstraníme všechny kromě toho, který je od p_0 nejvzdálenější)
3 PUSH(S, p_0); PUSH(S, p_1); PUSH(S, p_2)
4 **for** $i \leftarrow 3$ **to** m
5 **do while** úhel tvořený body NEXT-TO-TOP(S), TOP(S) a p_i nezatáčí doprava
6 **do** POP(S)
7 PUSH(S, p_i)
8 **return** S

Konvexní obal

Doba běhu algoritmu GRAHAM-SCAN(Q) je $O(n \log n)$, kde $n = |Q|$.

- Nalezení bodu p_0 vyžaduje čas $O(n)$.
- Seřazení bodů podle úhlů vyžaduje čas $O(n \log n)$.
- Vyřazení bodů, které svírají stejný úhel, vyžaduje čas $O(n)$.
- Provedení hlavní smyčky vyžaduje čas $O(n)$.

Poznámka: Operace PUSH je provedena nejvýše n krát.
Operace POP je provedena nejvýše tolikrát, kolikrát byla provedena operace PUSH.

Problém „Chocolate Chip Cookies“

Vykrajujeme kulaté zákusky o průměru $5\,\mathrm{cm}$ z napečeného odelníkového plechu.

Celá plocha, ze které vykrajujeme, je nepravidelně posypána kousky čokolády (jeden kousek je jeden bod). Chceme vykrojit takový zákusek, na kterém bude co nejvíce kousků čokolády.

Problém

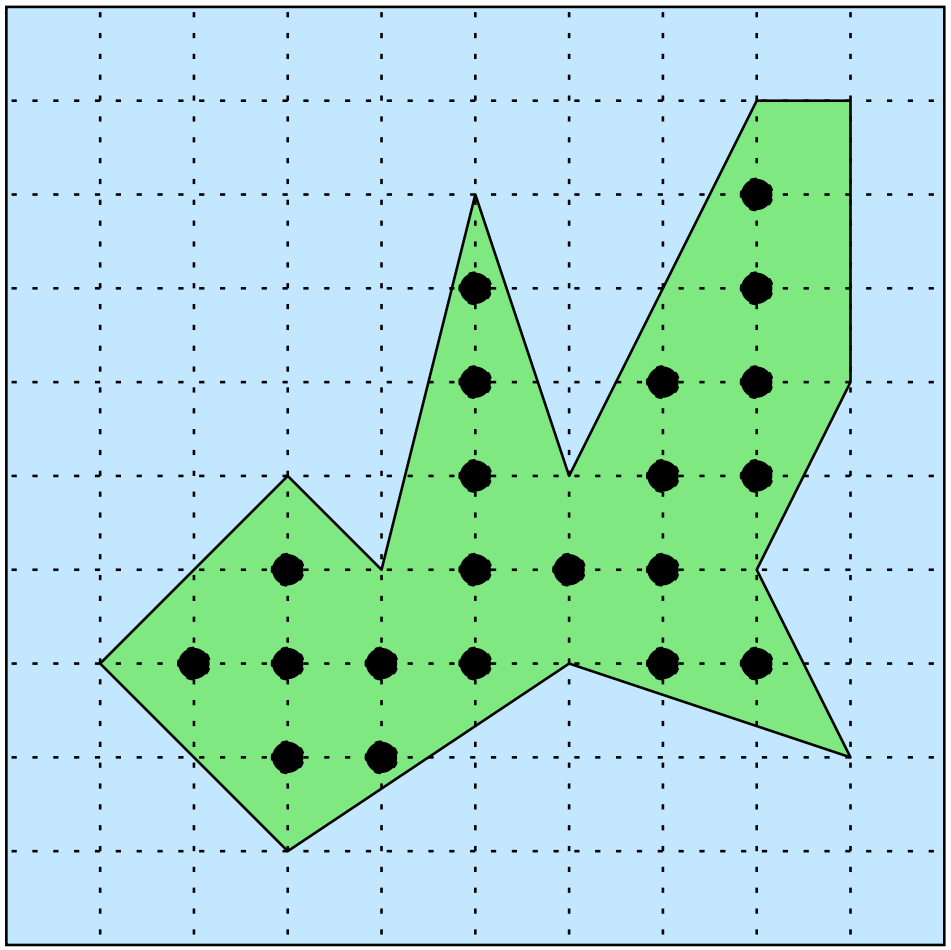
Vstup: Souřadnice jednotlivých kousků čokolády.

Výstup: Maximální počet kousků čokolády, které se mohou nacházet na jednom zákusku.

Poznámka: Problém je možné vyřešit v čase $O(n^2 \log n)$, kde n je počet kousků čokolády.

Problém „Trees on My Island“ (10088)

Chceme na ostrově vysázet stromy.



Problém „Trees on My Island“ (10088)

Ostrov je zadán jako mnohoúhelník, jehož vrcholy mají celočíselné souřadnice.

Poznámka: Stromy se nesmí nacházet na hranicích mnohoúhelníka.

Problém

Vstup: Souřadnice vrcholů mnohoúhelníka, který popisuje tvar ostrova.

Výstup: Počet stromů, které se vejdou na ostrov.

Chceme algoritmus, jehož časová složitost je $O(n \log m)$, kde n je počet vrcholů mnohoúhelníka a m je maximální hodnota souřadnic.

Problém „The Closest Pair Problem“ (10245)

Problém

Vstup: Množina n bodů v rovině.

Výstup: Vzdalenost dvou nejbližších bodů z této množiny.

Jednoduchý algoritmus, který vyzkouší všechny dvojice bodů, má časovou složitost $O(n^2)$.

Existuje algoritmus s časovou složitostí $O(n \log n)$.

Tento algoritmus je rekurzivní a jeho vstupem je množina bodů P a dvojice polí X a Y, kde:

- X obsahuje body z množiny P seřazené podle x-ové souřadnice.
- Y obsahuje body z množiny P seřazené podle y-ové souřadnice.

Problém „The Closest Pair Problem“ (10245)

Pokud je $|P| \leq 3$, vyzkouší se všechny dvojice bodů z P.

Pokud je $|P| > 3$:

- Najdeme svislou přímku l, která rozdělí množinu P na dvojici podmnožin P_L a P_R, kde $|P_L| = \lceil |P|/2 \rceil$ a $|P_R| = \lfloor |P|/2 \rfloor$, přičemž:
  - P_L obsahuje body ležící nalevo od l (případně na l).
  - P_R obsahuje body ležící napravo od l (případně na l).

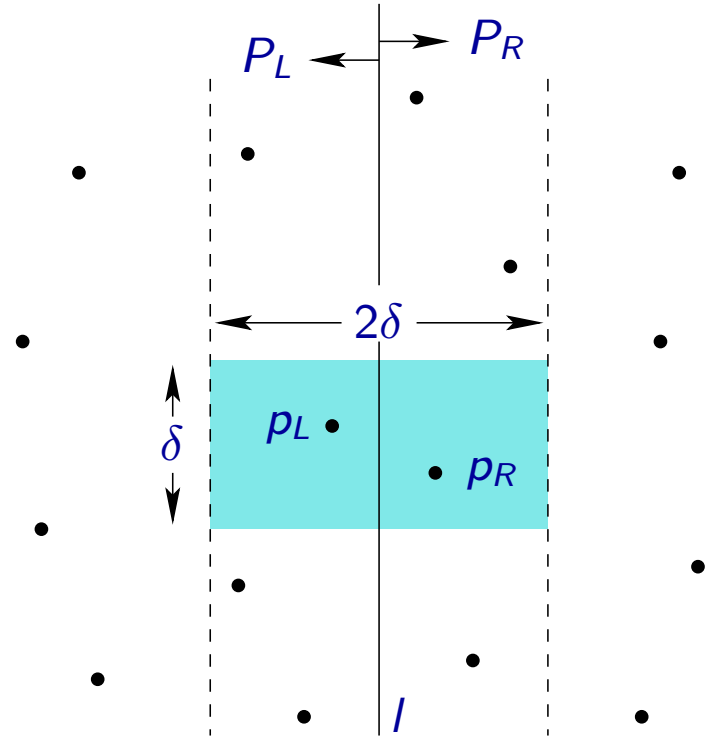
  Odpovídající způsobem rozdělíme pole X na X_L a X_R, a pole Y na Y_L a Y_R.

- Zavoláme proceduru rekurzivně nejprve s argumenty P_L, X_L, Y_L, a pak s argumenty P_R, X_R, Y_R.

  Jesliže tato dvě volání vrátí výsledky δ_L a δ_R, položíme $\delta = \min(\delta_L, \delta_R)$.

Problém „The Closest Pair Problem“ (10245)

- Prozkoumáme body v pruhu šířky 2δ kolem přímky l:



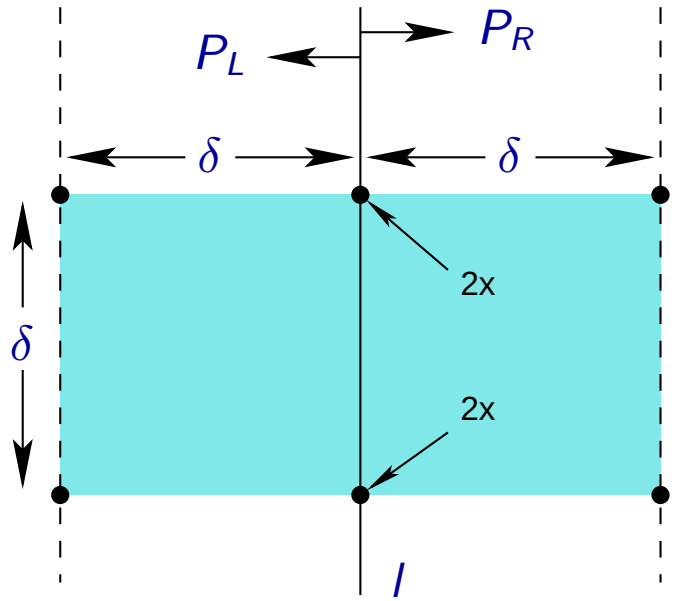
Pokud je vzdálenost mezi body $p_L \in P_L$ a $p_R \in P_R$ menší než δ, pak se musí oba nacházet v nějakém obdélníku velikosti $\delta \times 2\delta$ vycentrovaném podél přímky l.

Problém „The Closest Pair Problem“ (10245)

1. Vytvoříme pole Y', obsahující všechny body z Y nacházející se v pruhu šířky 2δ kolem l, seřazené podle y-ových souřadnic.
2. Pro každý bod p z pole Y' zkontrolujeme jeho vzdálenost k následujícím 7 bodům v tomto poli. Minimální vzdálenost udržujeme v proměnné δ'.
3. Vrátíme hodnotu $\min(\delta, \delta')$.

Poznámka: Pokud žádné body nekoincidují, stačí prověřovat následujících 5 bodů.

Problém „The Closest Pair Problem“ (10245)



V každém z obou čtverců velikosti $\delta \times \delta$ se mohou nacházet nanejvýš 4 body (v rozích čtverce).

V nejhorším případě se tedy v obdelníku velikosti $\delta \times 2\delta$ nachází 8 bodů (resp. 6 bodů, pokud žádné body nekoincidují).

Problém „The Closest Pair Problem“ (10245)

Algoritmus je třeba implementovat tak, aby doba běhu byla $T(n) = 2T(n/2) + O(n)$.

Je tedy třeba zajistit, aby jedno rekurzivní volání vyžadovalo čas $O(n)$, kde n je velikost daného podproblému.

Asi nejsložitější je rozdělení pole Y na Y_L a Y_R:

```
1  length[Y_L] ← length[Y_R] ← 0
2  for i ← 1 to length[Y]
3        do if Y[i] ∈ P_L
4              then length[Y_L] ← length[Y_L] + 1
5                   Y_L[length[Y_L]] ← Y[i]
6              else length[Y_R] ← length[Y_R] + 1
7                   Y_R[length[Y_R]] ← Y[i]
```

Permutace

Problém „Pixel Shuffle“ (CEPC 2005)

Uvažujeme bitmapu velikosti $n \times n$ (kde n je sudé).
Máme dánu určitou transformaci ϕ, která nějaký zadaným způsobem prohodí pixely bitmapy (např. otočí bitmapu o 90° proti směru hodinových ručiček).

Kolikrát musíme transformaci ϕ zopakovat, abychom dostali původní obrázek?

Problém „Pixel Shuffle“ (CEPC 2005)

Tranformace ϕ je zadána posloupností následujících klíčových slov, která popisují elementární transformace, ze kterých je transormace ϕ složena:

- id – identita, bitmapu nezmění
- rot – otočí bitmapu o 90° stupňů prosti směru hodinových ručiček
- sym – přetočí bitmapu podél svislé osy
- bhsym – přetočí dolní polovinu bitmapy podél svislé osy
- bvsym – otočí dolní polovinu bitmapy vzhůru nohama
- div – řádky $0, 2, \ldots, n-2$ se stanou řádky $0, 1, \ldots, n/2-1$ a řádky $1, 3, \ldots, n-1$ se stanou řádky $n/2, n/2+1, \ldots, n-1$
- mix – prolnutí všech dvojic řádků k a $k+1$: nejprve vytvoříme řádek délky $2n$, přičemž do něj střídavě zařazujeme pixely z řádků k a $k+1$, a tento řádek pak v polovině rozdělíme na řádky k a $k+1$

Problém „Pixel Shuffle“ (CEPC 2005)

Poznámka: Pokud je klíčové slovo následováno znakem ‘$-$’, označuje transformaci inverzní k dané operaci (např. $\mathrm{rot}-$ – otočení o $90°$ ve směru hodinových ručiček).

Problém

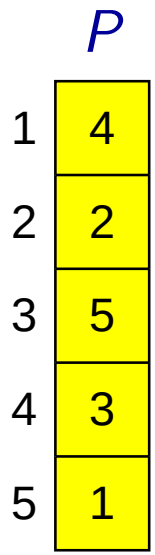
Vstup: Sudé číslo n ($2 \leq n \leq 1024$) a posloupnost klíčových slov (max. 32) popisující transformaci ϕ.

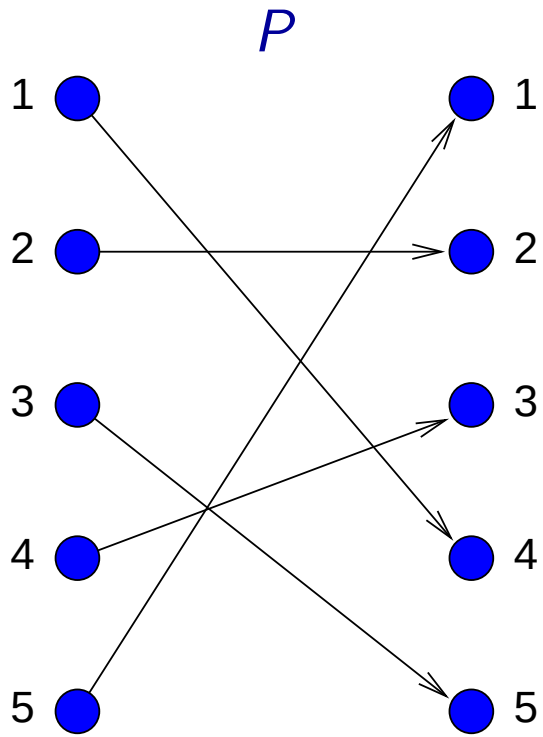
Výstup: Minimální číslo $m > 0$ takové, že pokud na bitmapu velikosti $n \times n$ aplikujeme m-krát transformaci ϕ, dostaneme původní bitmapu.

Permutace

Permutace na množině M je libovolné bijektivní zobrazení $P : M \to M$.

Například na množině $M = \{1, 2, 3, 4, 5\}$:

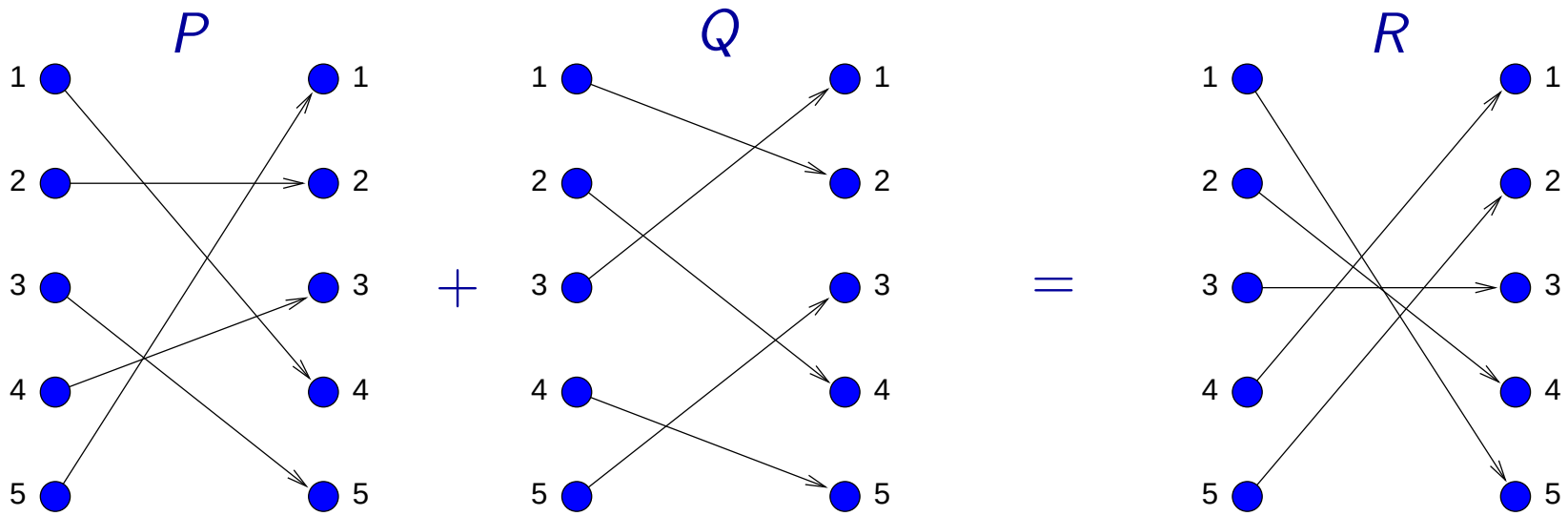




Poznámka: Budeme uvažovat pouze permutace na konečných množinách.

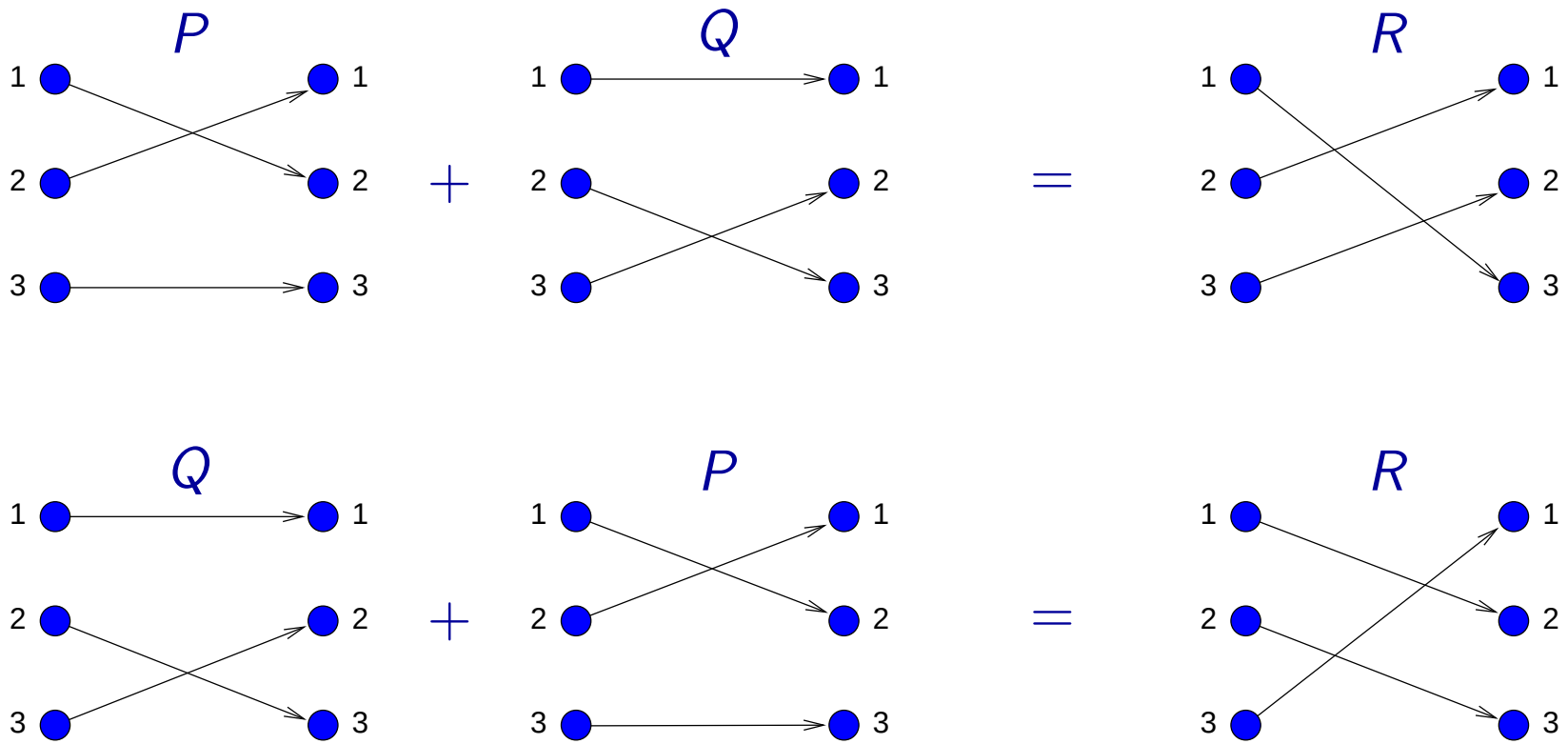
Skládání permutací

Permutace je možné skládat: $R = P \circ Q$, kde $\forall x : R(x) = Q(P(x))$



Skládání permutací

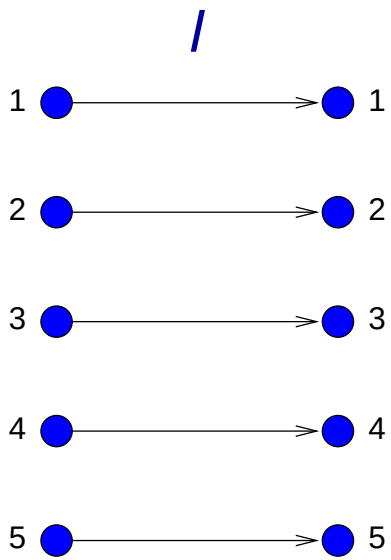
Skládání permutací je asociativní, ale není komutativní:



Permutace

Množina všech permutací na dané množině M spolu s operací skládání permutací tvoří grupu.

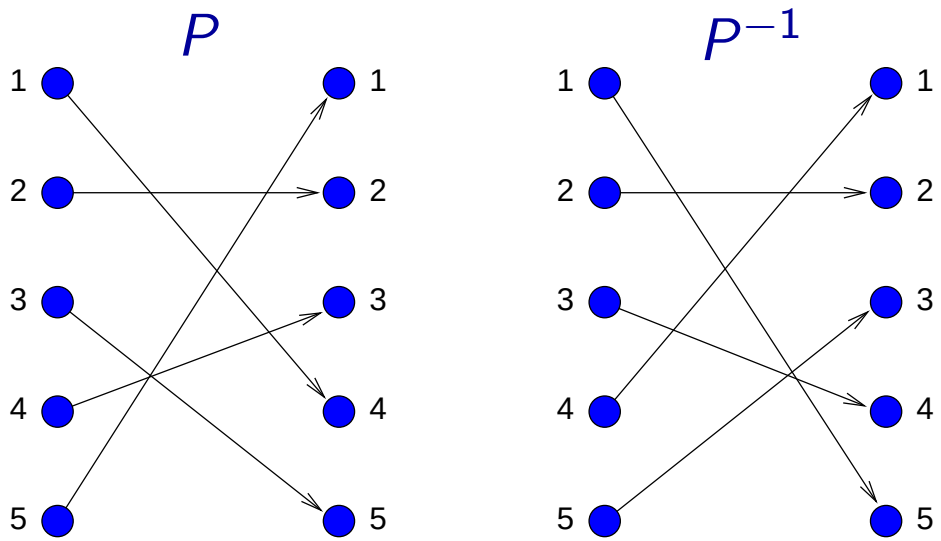
Jednotkovým prvkem této grupy je **identická** permutace I ($\forall x : I(x) = x$):



Zjevně pro libovolnou permutaci P platí $P \circ I = I \circ P = P$.

Permutace

Inverzním prvkem k permutaci P je **inverzní** permutace P^{-1} taková, že $P^{-1}(x) = y$ právě když $P(y) = x$:



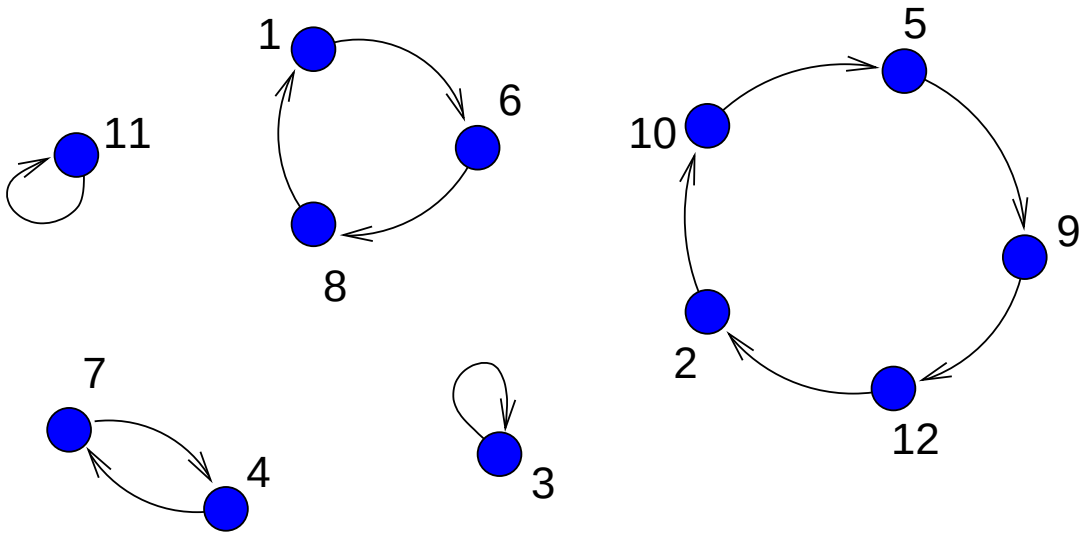
Zjevně platí $P \circ P^{-1} = P^{-1} \circ P = I$.

Také je zjevné, že $(P_1 \circ P_2 \circ \cdots \circ P_n)^{-1} = P_n^{-1} \circ P_{n-1}^{-1} \circ \cdots \circ P_1^{-1}$.

Permutace

Permutaci P na množině M lze znázornit jako orientovaný graf $G = (V, E)$, kde $V = M$, a kde $(x, y) \in E$ právě když $f(x) = y$.

	1	2	3	4	5	6	7	8	9	10	11	12
P	6	10	3	7	9	8	4	1	12	5	11	2



Permutace

FIND-CYCLES(P, n)

```
 1  Cycles ← ∅
 2  for i ← 1 to n
 3        do mark[i] ← FALSE
 4  for i ← 1 to n
 5        do if not mark[i]
 6              then C ← ∅
 7                   j ← i
 8                   repeat mark[j] ← TRUE
 9                          C ← C ∪ {j}
10                          j ← P[j]
11                     until mark[j]
12                   Cycles ← Cycles ∪ {C}
13  return Cycles
```

Permutace

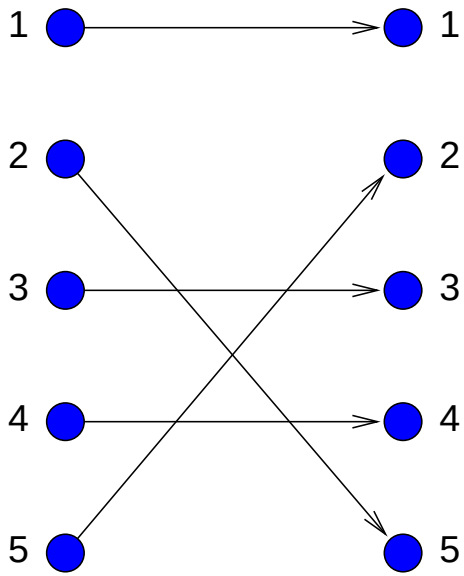
Pro danou permutaci P na množině M je možné najít všechny její cykly v čase $O(n)$, kde $n = |M|$.

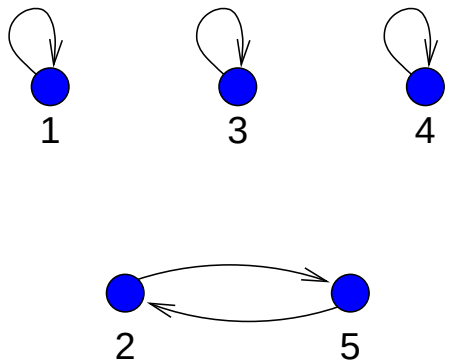
Řešení problému „Pixel Shuffle“:
Spočítat nejmenší společný násobek délek všech cyklů.

Transpozice

Transpozice je permutace, která prohodí právě dva nějaké prvky x a y (přičemž $x \neq y$).

Příklad: Transpozice na množině $M = \{1, 2, 3, 4, 5\}$, která prohodí prvky 2 a 5:





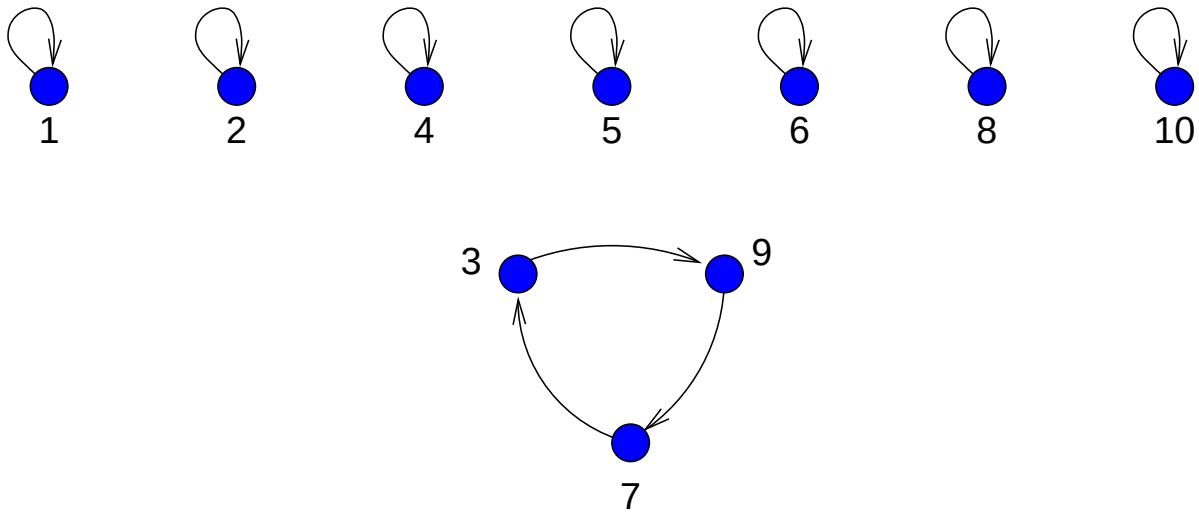
Libovolnou permutaci P je možné složit z transpozic.

Pro libovolnou transpozici T zjevně platí $T^{-1} = T$.

Trojcykly

Trojcyklus je permutace, která prohodí právě tři nějaké prvky x, y a z (přičemž $x \neq y$, $x \neq z$ a $y \neq z$).

Příklad: Trojcyklus na množině $M = \{1, 2, \ldots, 10\}$, který prohodí prvky 3, 7 a 9:



Problém „Algebra“ (CTU Open 2001)

Problém

Vstup: Číslo n a permutace čísel $1, 2, \ldots, n$, kde $3 \leq n \leq 1000000$.

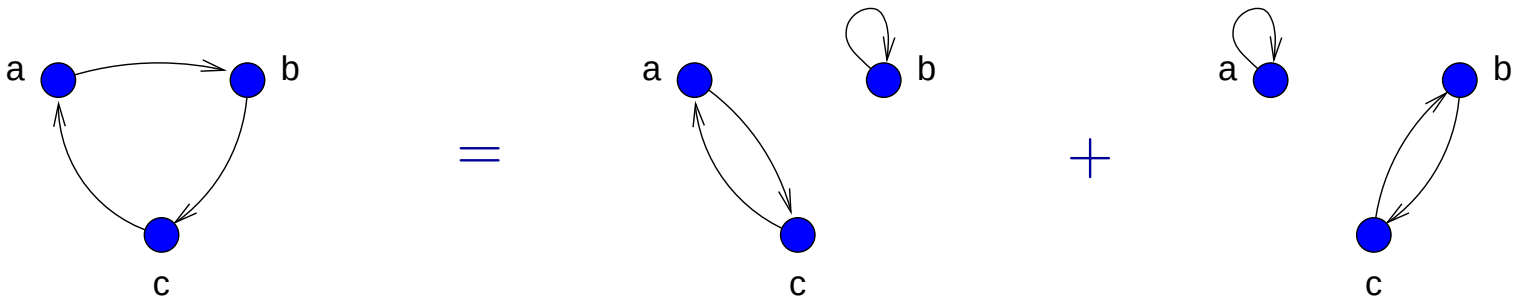
Otázka: Je možné zadanou permutaci složit z posloupnosti trojcyklů?

Trojcykly a transpozice

Tvrzení

Nechť P je permutace na konečné množině M takové, že $|M| \geq 3$. Permutaci P je možné složit z trojcyklů právě když je možné složit P ze sudého počtu transpozic.

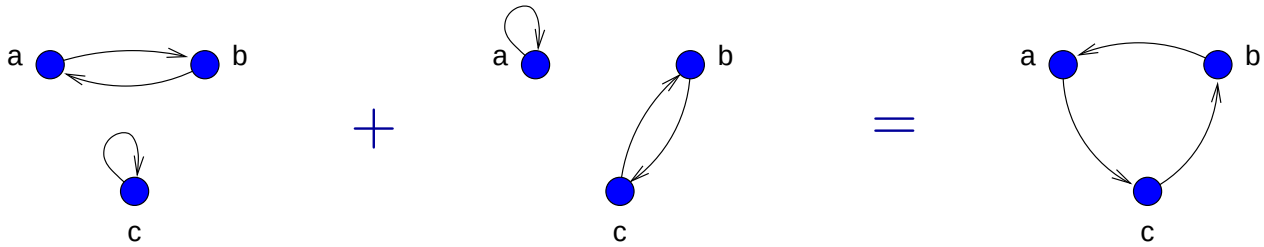
Důkaz: ($\Rightarrow$) Každý trojcyklus je možné nahradit dvojicí transpozic:



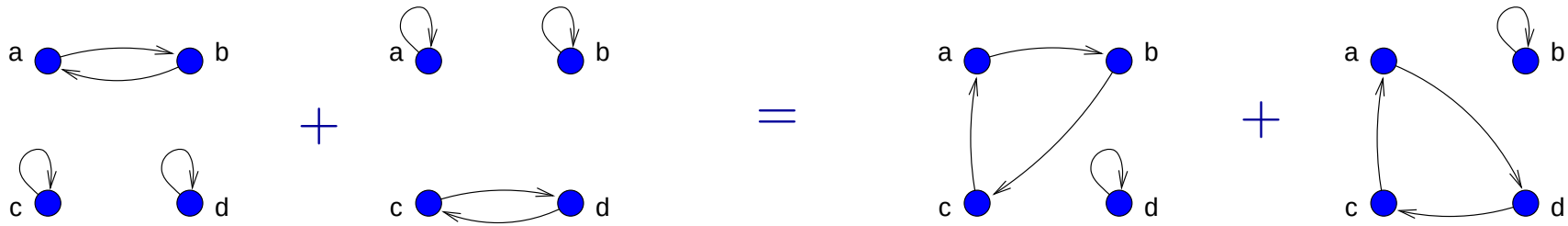
Trojcykly a transpozice

($\Leftarrow$) Každou dvojici transpozic lze nahradit 0, 1 nebo 2 trojcykly.

- Dvojici transpozic, kde první z nich prohodí prvky a a b a druhá také a a b, lze vypustit (nahradit 0 trojcykly).
- Dvojici transpozic, kde první z nich prohodí prvky a a b a druhá prohodí prvky b a c, lze nahradit 1 trojcyklem:



- Dvojici transpozic, kde první z nich prohodí prvky a a b a druhá prohodí prvky c a d, lze nahradit 2 trojcykly:



Sudé a liché permutace

Tvrzení

Libovolnou permutaci P je možné složit buď pouze ze sudého počtu transpozic nebo pouze z lichého počtu transpozic.

Permutace, které je možné složit pouze ze sudého počtu transpozic, se nazývají **sudé** permutace.
Permutace, které je možné složit pouze z lichého počtu transpozic, se nazývají **liché** permutace.

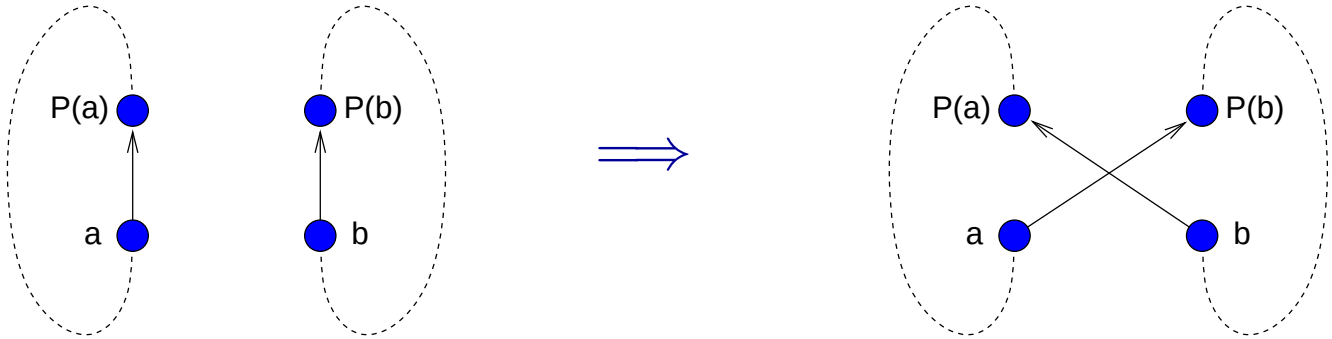
Důkaz tvrzení:
Uvažujme libovolnou permutaci P a libovolnou transpozici T na nějaké konečné množině M:

- Jak se liší počty cyklů permutací P a $P \circ T$?
- Jak se liší počty cyklů sudé délky?
- Jak se liší počty cyklů liché délky?

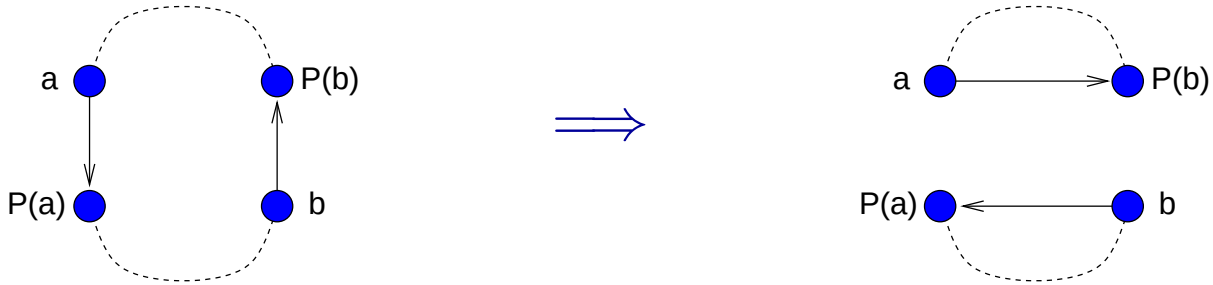
Sudé a liché permutace

Předpokládejme, že transpozice T prohodí prvky a a b. Mohou nastat dva případy:

- Prvky a a b leží v permutaci P každý na jiném cyklu:



- Prvky a a b leží v permutaci P na společném cyklu:



Sudé a liché permutace

- Pokud prvky a a b neležely na společném cyklu ($C_1 + C_2 \Rightarrow C$):

$\lvert C_1 \rvert$	$\lvert C_2 \rvert$	$\lvert C \rvert$	ΔN	ΔS	ΔL
S	S	S	-1	-1	0
S	L	L	-1	-1	0
L	S	L	-1	-1	0
L	L	S	-1	$+1$	-2

- Pokud prvky a a b ležely na společném cyklu ($C \Rightarrow C_1 + C_2$):

$\lvert C \rvert$	$\lvert C_1 \rvert$	$\lvert C_2 \rvert$	ΔN	ΔS	ΔL
S	S	S	$+1$	$+1$	0
S	L	L	$+1$	-1	$+2$
L	S	L	$+1$	$+1$	0
L	L	S	$+1$	$+1$	0

Sudé a liché permutace

Provedením jedné transpozice se:

- celkový počet cyklů změní o -1 nebo $+1$,
- počet cyklů sudé délky změní o -1 nebo $+1$,
- počet cyklů liché délky změní o -2, 0 nebo $+2$.

Provedením sudého počtu transpozic se celkový počet cyklů i počet cyklů sudé délky změní o sudé číslo.

Provedením lichého počtu transpozic se celkový počet cyklů i počet cyklů sudé délky změní o liché číslo.

Sudost nebo lichost počtu cyklů liché délky se nikdy nemění.

Sudé a liché permutace

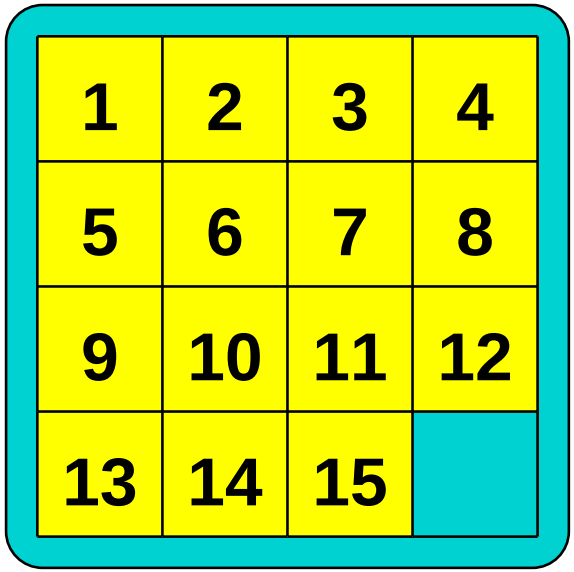
Uvažujme permutaci P nad množinou M. Označme n celkový počet cyklů v P.

Permutace P je **sudá** právě když je hodnota $|M| - n$ sudá.
Permutace P je **lichá** právě když je tato hodnota lichá.

Permutace P je **sudá** právě když obsahuje sudý počet cyklů sudé délky.
Permutace P je **lichá** právě když obsahuje lichý počet cyklů sudé délky.

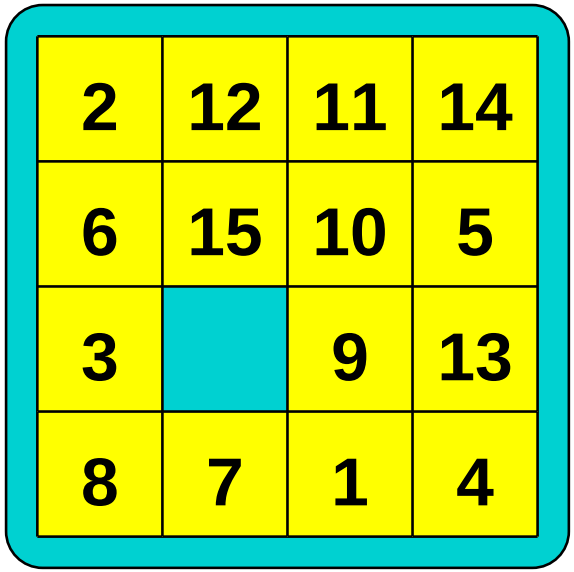
Problém „15-Puzzle Problem“ (10181)

Cílová pozice:



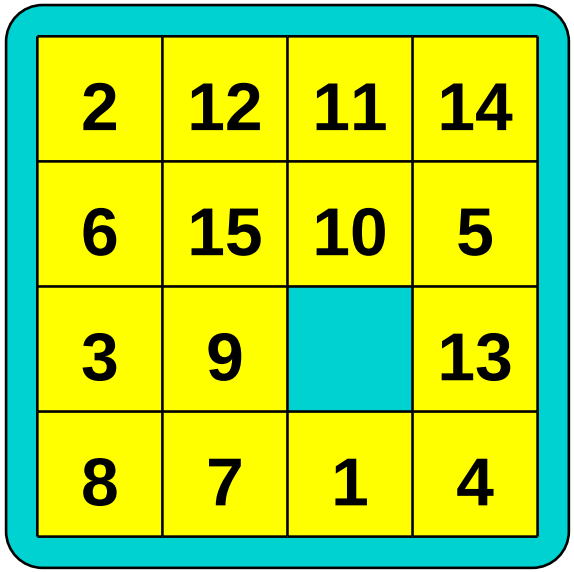
Problém „15-Puzzle Problem“ (10181)

Náhodná pozice:



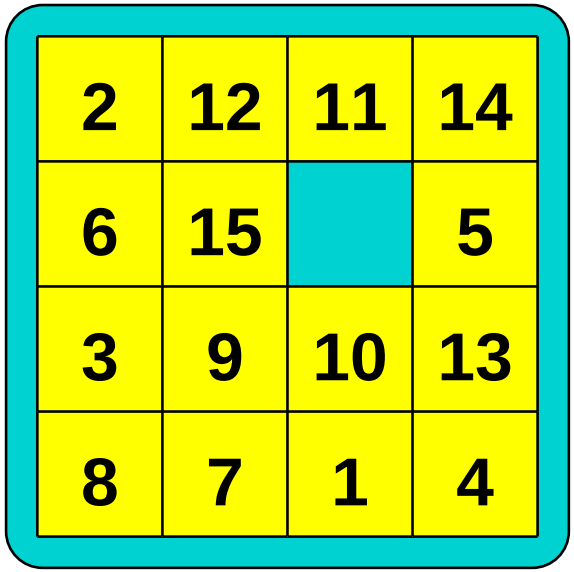
Problém „15-Puzzle Problem“ (10181)

Náhodná pozice:



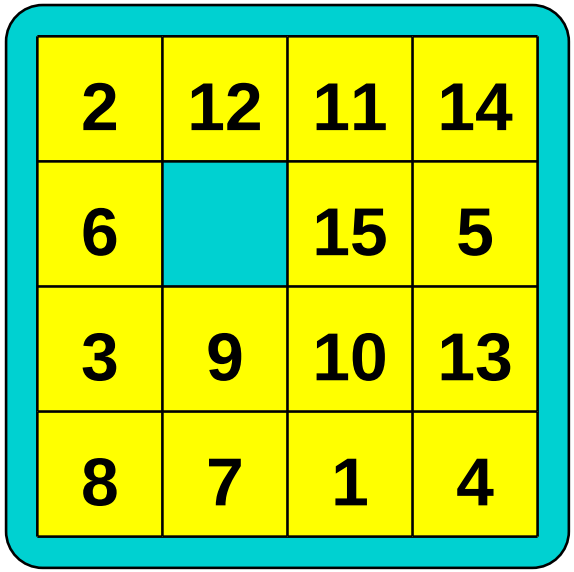
Problém „15-Puzzle Problem“ (10181)

Náhodná pozice:



Problém „15-Puzzle Problem“ (10181)

Náhodná pozice:



Problém „15-Puzzle Problem“ (10181)

Problém

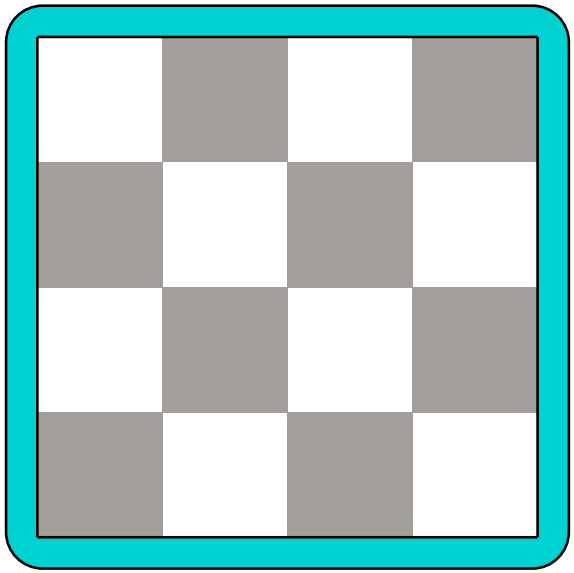
Vstup: Nějaká náhodná pozice.

Výstup: Co nejkratší posloupnost tahů taková, že po jejím provedení dostaneme cílouvou pozici, nebo informace, že řešení neexistuje.

Poznámka: Problém můžeme zobecnit na hlavolam velikosti $m \times n$, případně na hlavolam ve více rozměrech (např. třírozměrný).

Můžeme také uvažovat libovolnou počáteční i cílovou pozici.

Problém „15-Puzzle Problem“ (10181)



Pozorování: Mezera se přesouvá v každém tahu buď z bílého pole na černé nebo z černého pole na bílé.

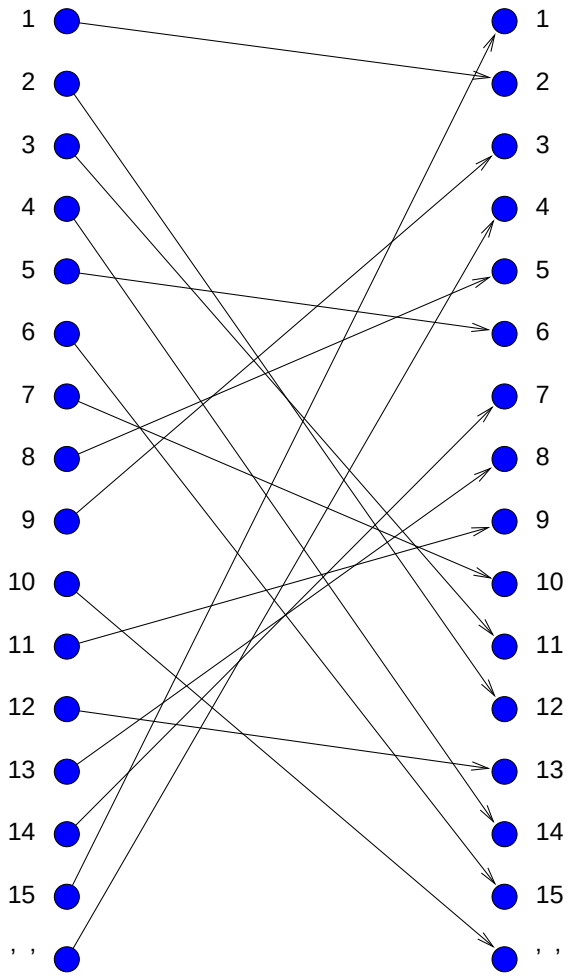
Po sudém počtu tahů tedy musí být mezera na poli stejné barvy jako na začátku, a naopak po lichém počtu tahů na poli opačné barvy.

Můžeme tedy snadno určit, zda řešení (pokud existuje) musí sestávat z lichého nebo ze sudého počtu tahů.

Problém „15-Puzzle Problem“ (10181)

Na jednotlivé pozice se můžeme dívat jako na permutace:

2	12	11	14
6	15	10	5
3		9	13
8	7	1	4



Problém „15-Puzzle Problem“ (10181)

Každý tah je transpozicí.

V závislosti na tom, zda jsou permutace reprezentující počáteční a cílovou pozici **sudé** nebo **liché**, můžeme určit, zda řešení musí sestávat z lichého nebo sudého počtu tahů.

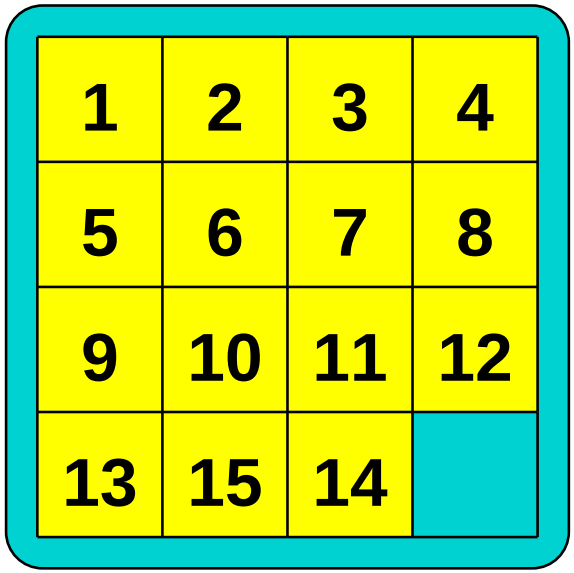
Máme dvě kritéria, která určují, zda počet tahů v řešení musí být sudý nebo lichý:

1. kritérium založené na střídání bílých a černých polí
2. kritérium založené na sudosti a lichosti permutací

Pokud jsou tato dvě kritéria v rozporu, řešení určitě neexistuje.

Problém „15-Puzzle Problem“ (10181)

Příklad pozice, která nemá řešení:



Problém „15-Puzzle Problem“ (10181)

Chceme ukázat následující tvrzení:

Pokud obě kritéria nejsou v rozporu, pak řešení existuje.

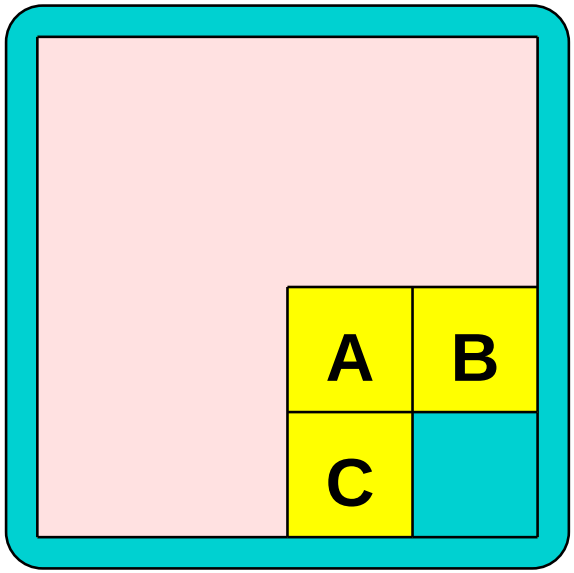
Předpokládejme, že mezera se v počáteční i cílové pozici nachází na tomtéž místě.
(Pokud ne, nejprve s ní z počáteční pozice najedeme na místo, kde se má nacházet v cílové pozici.)

Zjevně stačí ukázat, že každý trojcyklus, který nemění pozici mezery, lze realizovat nějakou posloupností tahů.

Řešení (ne nutně optimální) lze pak složit z jednotlivých trojcyklů.

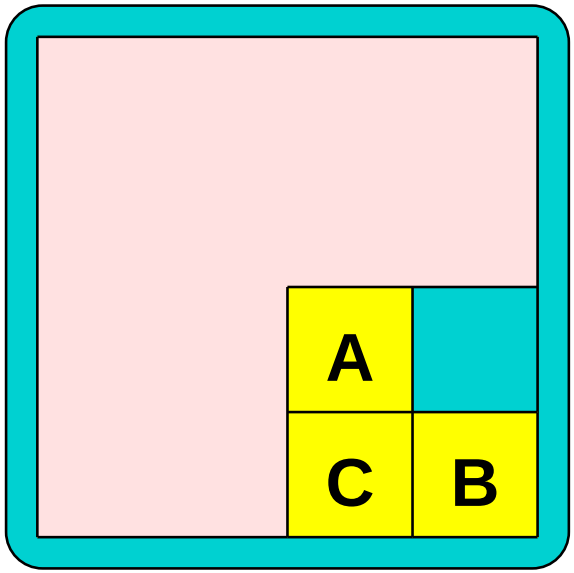
Problém „15-Puzzle Problem“ (10181)

Nejprve ukážeme, že alespoň jeden nějaký trojcyklus je možné realizovat nějakou posloupností tahů:



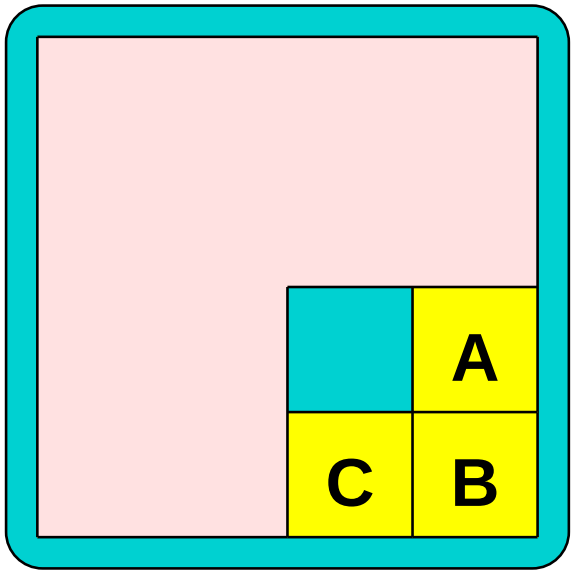
Problém „15-Puzzle Problem“ (10181)

Nejprve ukážeme, že alespoň jeden nějaký trojcyklus je možné realizovat nějakou posloupností tahů:



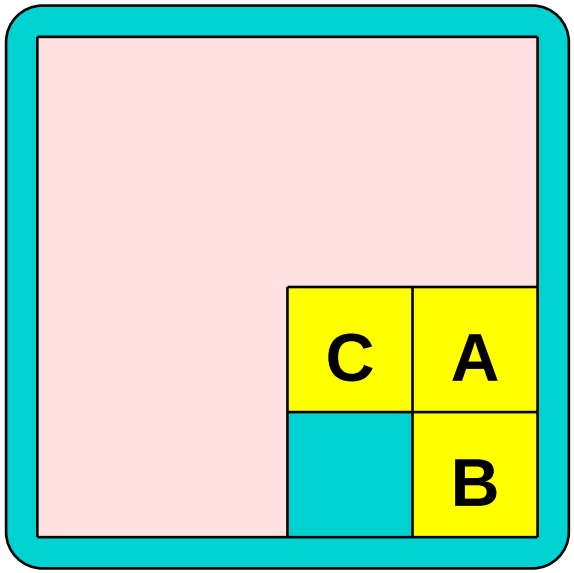
Problém „15-Puzzle Problem“ (10181)

Nejprve ukážeme, že alespoň jeden nějaký trojcyklus je možné realizovat nějakou posloupností tahů:



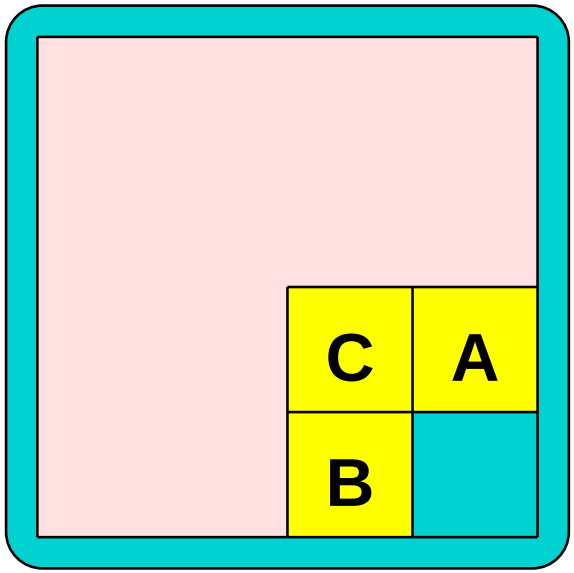
Problém „15-Puzzle Problem“ (10181)

Nejprve ukážeme, že alespoň jeden nějaký trojcyklus je možné realizovat nějakou posloupností tahů:



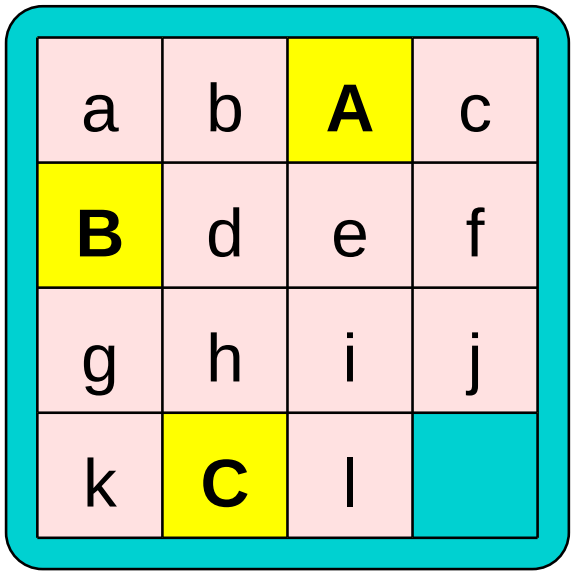
Problém „15-Puzzle Problem“ (10181)

Nejprve ukážeme, že alespoň jeden nějaký trojcyklus je možné realizovat nějakou posloupností tahů:



Problém „15-Puzzle Problem“ (10181)

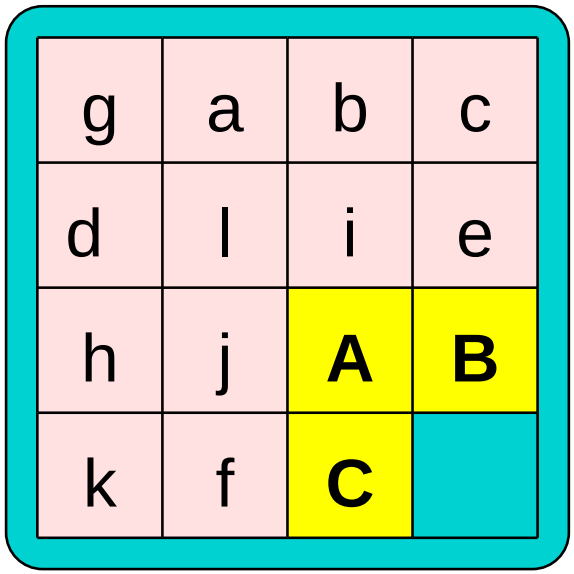
Nyní ukážeme, jak realizovat libovolný trojcyklus:



- Nejprve nějakou posloupností tahů P přemístíme A, B a C na místa, na kterých už trojcyklus realizovat umíme . . .

Problém „15-Puzzle Problem“ (10181)

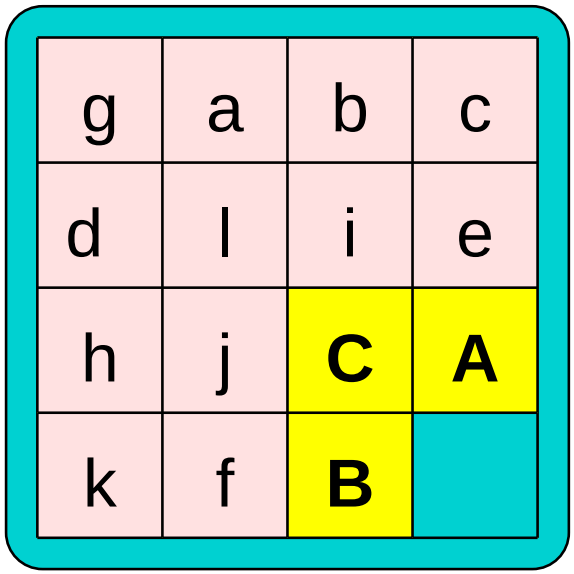
Nyní ukážeme, jak realizovat libovolný trojcyklus:



- Posloupností tahů T realizujeme trojcyklus . . .

Problém „15-Puzzle Problem“ (10181)

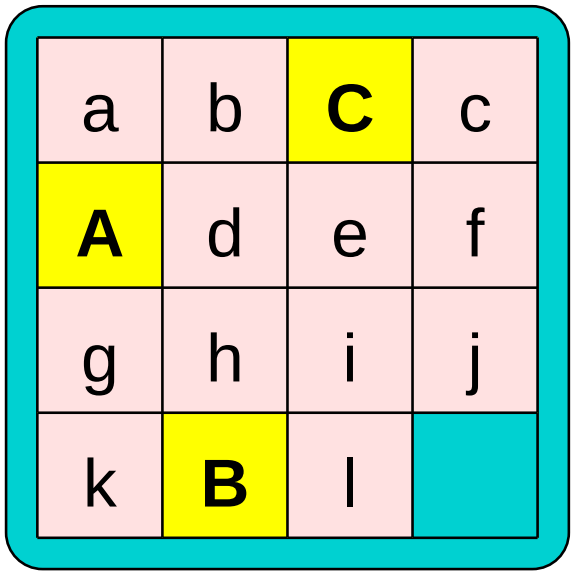
Nyní ukážeme, jak realizovat libovolný trojcyklus:



- Posloupností P^{-1} (tj. posloupností P „pozpátku“) vrátíme vše (kromě A, B a C) na původní místo ...

Problém „15-Puzzle Problem“ (10181)

Nyní ukážeme, jak realizovat libovolný trojcyklus:



- . . . a jsme hotovi.

Problém „15-Puzzle Problem“ (10181)

Pro nalezení nejkratšího řešení lze použít například rekurzivní algoritmus, ovšem jen pro hlavolamy malé velikosti, například 4×4.

Je možné použít pruning založený na následujícím pozorování:

Každé pozici P přiřaďme hodnotu $D(P)$ definovanou jako

$$D(P) = \sum_{i \in M} d_i(P)$$

kde M je množina všech kostek, a $d_i(P)$ je vzdálenost kostky i v permutaci P do místa, kam i patří v cílové permutaci C:

$$d_i(P) = |x_i(P) - x_i(C)| + |y_i(P) - y_i(C)|$$

Pozorování: Abychom se z P dostali do C, je zjevně třeba provést alespoň $D(P)$ tahů.

Hry

Hra „Kočka a myš“

- Hraje se na hrací ploše reprezentované orientovaným grafem $G = (V, E)$, kde jeden z vrcholů je označen jako myší díra.
- Jeden hráč hraje s figurkou představující kočku a druhý s figurkou představující myš.
- Na začátku se figurky nachází každá na nějakém vrcholu grafu G.
- Hráči se střídají v tazích. V jednom tahu musí hráč přesunout svou figurku nacházející se na vrcholu u na některý vrchol v takový, že $(u, v) \in E$.
- Myš vyhraje, pokud se jí podaří dosáhnout myší díry. (Kočka nesmí do myší díry vstupovat.)
- Kočka vyhraje, pokud se jí podaří dosáhnout situace, že se kočka i myš nachází na stejném vrcholu.
- Pokud se během hry nějaká situace zopakuje, přičemž na tahu je tentýž hráč, výsledkem je remíza.

Hra „Kočka a myš“

Poznámka: Předpokládáme, že každý vrchol kromě myší díry má alespoň jednoho následníka.

Problém

Vstup: Graf $G = (V, E)$, vrchol $v_d \in V$, kde se nachází myší díra, vrcholy v_k a v_m, na kterých se nachází figurky kočky a myši, a informace, který hráč je momentálně na tahu.

Otázka: Který hráč má v dané situaci vítěznou strategii?

Poznámka: Možné odpovědi jsou *myš*, *kočka* a *žádný*.

Hry

Uvažujme hry, jako jsou například šachy, dáma, NIM (odebírání zápalek), piškvorky nebo dříve uvedená hra s kočkou a myší, kde:

- hrají proti sobě dva hráči, kteří se střídají v tazích,
- nehraje zde žádnou roli náhoda,
- oba hráči mají úplné informace o aktuální situaci ve hře,
- jediné změny ve hře jsou ty, které provedou hráči při provádění tahů,
- některé situace jsou koncové a je v nich určeno, který z hráčů vyhrál (případně, zda nastala remíza), přičemž nelze vyhrát jinak než dosažením takovéto koncové situace.

Hry

Na každou takovouto hru se můžeme dívat jako na čtveřici $A = (G, player, F, win)$, kde:

- $G = (V, E)$ je orientovaný graf, kde V je množina všechn možných situací ve hře, a kde $(u, v) \in E$ právě když se situace u je možný tah do situace v.
- Funkce $player : V \to \{I, II\}$ určuje, který hráč je v dané situaci na tahu.
- $F \subseteq V$ je množina koncových situací.
- Funkce $win : F \to \{I, II\}$ určuje, který hráč v dané koncové situaci vyhrál.

Poznámka: Čtveřice A se někdy nazývá **aréna**.

Hry

Hra (game) je dvojice (A, v_s), kde $A = (G, player, F, win)$ je aréna a $v_s \in V$ je počáteční situace.

Předpokládejme, že $v_0, v_1, \ldots, v_k$ je dosavadní průběh hry, tj. platí pro něj $v_0 = v_s$, dále $(v_i, v_{i+1}) \in E$ pro všechna $0 \leq i < k$, a také $v_i \notin F$, pokud $i < k$.

Strategie hráče I je funkce, která každému takovému dosavadnímu průběhu hry, pro který platí $player(v_k) = I$ a $v_k \notin F$ přiřazuje nějaké $v \in V$ takové, že $(v_k, v) \in E$ (tj. tah, který má hráč I v dané situaci provést).

U daného typu her se můžeme omezit na strategie nezávisející na historii, tj. na strategie, kde zvolený tah závisí pouze na v_k.

Strategií hráče p, kde $p \in \{I, II\}$, budeme rozumět funkci $f : V_p \to V$, kde $V_p = \{v \in V \mid player(v) = p,\ v \notin F\}$.

Hry

Strategie hráče p je **vítězná** (**vyhrávající**) pokud hráč p při použití této strategie vždy vyhraje, bez ohledu na to, jak hraje jeho protihráč.

Definujme množinu $W_I \subseteq V$ jako množinu všech vrcholů v, kde hráč I má vyhrávající strategii ve hře (A, v). (Podobně definujeme i W_{II}.)

Poznámka: Zjevně platí $W_I \cap W_{II} = \emptyset$.

Problém

Vstup: Aréna $A = (G, player, F, win)$.

Výstup: Množina W_I.

Hry

Označme W_i množinu vrcholů, ve kterých má hráč I strategii, která mu zaručuje, že vyhraje do i tahů.

Zjevně platí $W_0 = \{v \in F \mid win(v) = I\}$.

Dále je zjevné, že $W_0 \subseteq W_1 \subseteq W_2 \subseteq \cdots$.

Pokud má každý vrchol z V jen konečný počet následníků (a speciálně, pokud je množina V konečná), platí

$$W_I = \bigcup_{i \geq 0} W_i$$

.

Hry

Pro libovolný vrchol $u \in W_i$, kde $i > 0$, musí platit:

- Pokud $player(u) = I$, pak **existuje** alespoň jeden vrchol $v \in Succ(u)$ takový, že $v \in W_j$ pro nějaké $j < i$.
- Pokud $player(u) = II$, pak pro **každý** vrchol $v \in Succ(u)$ platí, že $v \in W_j$ pro nějaké $j < i$.

Poznámka: $Succ(u)$ označuje množinu následníků vrcholu u, tj. $Succ(u) = \{v \in V \mid (u, v) \in E\}$.

Výše popsané vztahy nám dávají návod jak množinu W_I (a případně odpovídající vyhrávající strategii) spočítat.

Hry

Množinu W_I je možné spočítat v čase $O(|V| + |E|)$:

WINNING-SET$(G, player, F, win)$

```
1   for each vertex u ∈ V[G]
2         do count[u] ← |Succ(u)|
3   Q ← ∅
4   W ← ∅
5   for each vertex u ∈ F such that win(u) = I
6         do W ← W ∪ {u}
7            ENQUEUE(Q, u)
        ...
```

Poznámka: $Succ(u)$ a $Pred(u)$ označují množiny následníků a předchůdců vrcholu u v grafu G.

Hry

```
    . . . li:game-end1
 8  while Q ≠ ∅
 9        do v ← DEQUEUE(Q)
10           for each u ∈ Pred(v)
11                do if player[u] = I
12                      then if u ∉ W
13                              then W ← W ∪ {u}
14                                   ENQUEUE(Q, u)
15                      else     ▷ Tj. player[u] = II
16                           count[u] ← count[u] − 1
17                           if count[u] = 0
18                              then W ← W ∪ {u}
19                                   ENQUEUE(Q, u)
20  return W
```

Problém „A Multiplication Game“ (847)

Dva hráči hrají následující hru:

Nejprve zvolí (náhodně) nějaké celé číslo n větší než 1.

Hra začíná hodnotou $p = 1$. Hráči se střídají v tazích. Jeden tah vypadá tak, že hráč, který je momentálně na tahu, vynásobí číslo p nějakým celým číslem v intervalu 2 až 9 (včetně).

Vyhrává hráč, který jako první dosáhne hodnoty p takové, že $p \geq n$.

Problém

Vstup: Číslo n (kde $1 < n < 2^{32}$).

Otázka: Který z hráčů má v této hře vyhrávající strategii?

Problém „A Multiplication Game“ (847)

Řešení:

```
char* solve(unsigned int n)
{
    while (1) {
        n = (n + 8) / 9;
        if (n <= 1) return "Prvni";
        n = (n + 1) / 2;
        if (n <= 1) return "Druhy";
    }
}
```

Problém „A Multiplication Game“ (847)

Důkaz korektnosti je jednoduchý.

Jediný problematický případ je, když víme, že na intervalu $\lceil x/2 \rceil, \ldots, x-1$ prohrává hráč, který je právě na tahu, a chceme ukázat, že na intervalu $\lceil \lceil x/2 \rceil / 9 \rceil, \ldots, \lceil x/2 \rceil - 1$ vyhrává hráč, který je právě na tahu.

Stačí ukázat, že pro každé y z intervalu $\lceil \lceil x/2 \rceil / 9 \rceil, \ldots, \lceil x/2 \rceil - 1$ existuje nějaké $k \in \{2, 3, \ldots, 9\}$ takové, že $\lceil x/2 \rceil \leq y \cdot k < x$.

Pokud by tomu tak nebylo, muselo by existovat nějaké celé číslo $k \geq 1$ takové, že $y \cdot k < \lceil x/2 \rceil$ a současně $x \leq y \cdot (k+1)$.

Z $y \cdot k < \lceil x/2 \rceil$ plyne $y \cdot k < x/2$, neboli $2yk < x$.

Protože $2yk < x \leq y \cdot (k+1)$, musí platit $2k < k+1$, neboli $k < 1$, což je spor.

Problém „Stone Game“ (10165)

Dva hráči hrají hru:

- Na začátku mají N hromádek kamenů, které obsahují $P_1, P_2, \ldots, P_N$ kamenů.
- Hráč, který je na tahu, zvolí libovolnou hromádku a z ní odebere libovolný (ale nenulový) počet kamenů.
- Hráči se v tazích střídají.
- Vyhrává hráč, který odebere poslení kámen (resp. kameny).

Problém

Vstup: Počet hromádek N a počty kamenů $P_1, P_2, \ldots, P_N$.

Otázka: Má hráč, který táhne jako první, vyhrávající strategii?

Problém „Stone Game“ (10165)

Poznámka: Hra je známa pod názvem NIM.

Úloha má překvapivě jednoduché řešení:

Hráč, který je právě na tahu, má vyhrávající strategii, právě když

$$P_1 \text{ xor } P_2 \text{ xor } \cdots \text{ xor } P_N \neq 0$$