



EKSI



KORESPONDENČNÍ SEMINÁŘ Z INFORMATIKY

KSI, neboli Korespondenční Seminář z Informatiky, je celoroční soutěž pro středoškoláky organizovaná vysokoškolskými studenty informatiky (většinou z Fakulty informatiky Masarykovy univerzity). Cílem semináře je seznámit řešitele se zajímavými oblastmi informatiky a procvičit programátorské a matematické myšlení. Chcete se naučit nové věci, potrénovat svůj mozek, zasoutěžit si nebo jet na zážitkové soustředění? Začněte řešit KSI!

Každý ročník semináře se skládá z pěti tematických sad. Sada vždy začíná úvodním povídáním, ve kterém vás, řešitele, uvedeme do tématu, vysvětlíme základní myšlenky, pojmy a triky. Následují soutěžní úlohy různého charakteru, na jejichž vyřešení a sepsání řešení máte zhruba jeden měsíc. Zatímco nejjednodušší úlohy by po přečtení úvodního povídání neměly dělat problémy ani úplným začátečníkům, nad tou nejtěžší se může zapotit i ostrý řešitel programátorských soutěží.

Zadání zveřejňujeme na našich internetových stránkách a řešitelům je zasíláme i poštou. Odevzdávání úloh a další komunikace pak probíhá přes náš webový systém. Po termínu odevzdání vaše řešení opravíme, obodujeme a okomentujeme. Po vyhodnocení poslední sady úloh jsou nejúspěšnější řešitelé pozváni na týdenní soustředění. Úspěšní řešitelé semináře (ti, kteří dosáhnou alespoň 60 % celkového počtu bodů) budou zároveň přijati na Fakultu informatiky MU bez přijímacích zkoušek¹. Více informací o KSI hledejte na

<http://ksi.fi.muni.cz>

Jaké budou úlohy? Jak psát a odevzdat řešení?

Ve většině úloh bude vaším úkolem řešit algoritmický či programátorský problém. Některé z našich úloh mohou být složité nebo pracné, ale s dobrým nápadem se většinou dají vyřešit rychle a elegantně. Není třeba vyřešit všechny úlohy sady – pošlete, co vyřešíte, zbytek si po uzavření sady můžete přečíst ve vzorovém řešení. Také není nutné, abyste měli úlohu kompletně dořešenou. I pokud máte jen nějaké nápady co s ní napište nám je a my vaši snahu určitě oceníme.

Pokud sepisujete řešení úloh poprvé, určitě se podívejte na článek „Jak psát řešení“ na našem webu. Zde zdůrazníme, že při opravování hodnotíme především správnost postupu (funkčnost) a efektivitu řešení, ale v neposlední řadě také kvalitu popisu a zdůvodnění správnosti. Důležitou součástí řešení je také vyhodnocení časové a případně i prostorové složitosti vašich algoritmů (více o časové složitosti najdete ve článku „O prostorové a časové složitosti“).

Řešení se odevzdávají elektronicky na našich stránkách <http://ksi.fi.muni.cz>. Preferujeme texty ve formátu **PDF** (.pdf), dále přijímáme i formáty .doc, .odt, .txt. V textu řešení prosím vždy **uvedte své jméno a školu**.

Co vás čeká v první sadě letošního KSI? Letošní ročník jsme pro vás připravili novinku. Kromě vymýšlení a popisu algoritmů si tyto algoritmy můžete i naprogramovat. Ke každému příkladu dostanete nachystané programky, kterým chybí části kódu. Náš kód se postará o načítání dat a jejich

¹S jakýmikoliv dotazy ohledně přijetí na Fakultu informatiky MU můžete obrátit na studijní oddělení studijni@fi.muni.cz.

přípravu. Vaším úkolem bude napsat kód, který s daty pracuje a upravuje je podle zadání příkladu. Náš program vám potom pomůže zkontrolovat, jestli váš kód pracuje správně. K programování budeme používat velice populární jazyk Python.

Abyste se do toho lépe dostali, nachystali jsme pro vás malou ochutnávku základních konstrukcí, které Python umí. Velmi praktickou součástí Pythonu je jeho Shell (interpret), s jehož pomocí si můžete Python snadno a rychle vyzkoušet.

Python

Abyste mohli řešit příklady, budete potřebovat Python 2.

Pokud používáte Linux, tak doporučujeme stáhnout si balík *idle*. Příkazem `idle` pak spustíte vývojové prostředí a jste připraveni na následující část úvodníku.

Pro Windows si stáhněte odpovídající verzi ze stránky <https://www.python.org/download/releases/2.7.8/>. Společně s pythonem se vám nainstaluje i vývojové prostředí IDLE. To spustíte a můžete pokračovat ve čtení úvodníku a zároveň si zkusit jednotlivé ukázky.



Čísla

Abychom v programu mohli něco vypočítat, potřebujeme vědět jak zacházet s daty. Nejprve se ve zkratce podíváme na celá čísla. V Pythonu prostě napíšete číslo a je to. Základní aritmetické operace se také zapisují tak, jak byste čekali. Zkuste si sami v interpretu provést příkazy:

```
1 | print 1
2 | print 42
3 | print 1+1
4 | print 2*2
5 | print 13/4
6 | print 13%4
```

Všimněte si posledních dvou operací: `/` a `%`. Jedná se o celočíselné dělení se zbytkem na celých číslech. Pomocí lomítka zjistíme celou část výsledku, procento nám naopak vrátí zbytek po dělení.

Kromě počítání s čísly je také chceme ukládat, abychom je mohli použít později. K tomu slouží proměnné. Proměnnou zavedeme tak, že napíšeme její název a pomocí `=` do ní přiřadíme nějakou hodnotu. Pomocí tohoto názvu se na ni pak můžeme odkazovat:

```
1 | a = 3
2 | b = 4
3 | print a
4 | print a+b
```

Výsledky výpočtů také můžeme ukládat do proměnných:

```
1 |   vysledek = a*b
2 |   print vysledek
3 |   print a, "krat", b, "se rovna", vysledek
```

Jak název napovídá, tak proměnné můžeme i přepisovat na jiné hodnoty:

```
1 |   vysledek = a + b
2 |   print vysledek
3 |   vysledek = a * b
4 |   print vysledek
5 |   vysledek = a + (6 * b)
6 |   print vysledek
```

Názvy proměnných mohou být skoro libovolné. Je ale velmi dobrým programátorským zvykem pro názvy proměnných používat pouze písmena, podtržítko a spíše výjimečně i cifry. Je také potřeba si dávat pozor na velikost písmen – v Pythonu totiž velké písmeno není to stejné jako malá písmena. Pomůže vám při psaní a nám při opravování, pokud proměnné pojmenujete podle jejich významu. Vyhýbejte se pojmenováním a, b, x, a_1 a preferujte označení, z kterých vám obsah proměnné bude zřejmý (`delka_seznamu`, `mezisoucet`, `seznam_slov`, ...). Věřte, že tak zabráníte spoustě chyb, ke kterým by mohlo dojít, kdybyste si např. oblíbenou proměnnou i někde omylem přepsali. Též je zvykem v jazyce Python psát proměnné malými písmeny a pro oddělení slov používat podtržítko.

```
1 |   promenna = 7
2 |   Promenna = 2
3 |   print promenna
4 |   print Promenna
5 |   PROMENNA = 3
6 |   print (promenna + Promenna) * PROMENNA
7 |   moje_specialni_promenna_s_velice_dlouhym_nazvem = 42
8 |   pr0m3nn4 = 1337
9 |   print pr0m3nn4 % moje_specialni_promenna_s_velice_dlouhym_nazvem
```

Jak jste si jistě všimli, kód s posledními dvěma proměnnými není příliš dobře čitelný. Až budete psát vaše programy, nazývejte proměnné krátkými jmény. Kód se potom bude lépe číst vám i opravujícím!

6 5 ? 4 7

Vstup a výstup

Velmi užitečným nástrojem jsou vstupní a výstupní příkazy `input` a `print`. S druhým z nich jsme se už potkali, takže jen ve zkratce; pomocí tohoto příkazu může program tisknout různé informace, například o průběhu výpočtu nebo vypisovat jeho mezivýsledky. Použití příkazu je:

```
1 |   print(mezera)<argumenty>
```

Jako argumenty lze použít nejen čísla, ale třeba i text. Abychom v programu odlišili text od kódu, musíme ho uvést pomocí uvozovek

```
1 | print "Textovy retezec"
```

Takto uvozený text se potom nazývá textový řetězec. Kromě tisku lze řetězec také ukládat do proměnných a dále s ním pracovat. Více si o řetězcích řekneme později, zatím si ale můžete vyzkoušet třeba

```
1 | retezec = "textovy retezec"
2 | print retezec
```

Pokud chceme vytisknout více hodnot zároveň, pak jednotlivé hodnoty oddělujeme čárkami. Pokud budete chtít, aby se za výpisem automaticky neukončil řádek, pak musíte za poslední argument přidat čárku.

```
1 | print "Ahoj", "Karliku"
2 | print "Jedna je ", 1
3 | print "Jedna plus dva je ", 1+2
4 | a = 3
5 | b = 5
6 | print a
7 | print "a"
8 | print "a + b =", a+b
9 | print a, "+", b, "=", a+b
10 | print "Prvni print"; print "Druhy print"
11 | print "Prvni print",; print "Druhy print"
```

Pomocí příkazu `print` už umí s námi náš program komunikovat. Abychom mu mohli odpovídat, použijeme příkaz `input()`. S jeho pomocí si od nás program může vyžádat nějakou reakci. Při tomto dotazu se program zastaví a počká, dokud mu něco nenapišeme. Text, který mu napíšeme, pak interpretuje jako zápis proměnné a vrátí.

```
1 | vstup = input() # napiste: 3
2 | print "vstup je ", vstup
```

Pokud napíšeme číslo, program si ho jako číslo uloží. Pokud napíšeme nějaký výraz, program si ho nejprve spočítá a uloží pouze výsledek (zkuste zadat například `1+1`). Pomocí příkazu `input` lze načítat i složitější struktury. Ty ale nebudeme při řešení této sady potřebovat, takže si je teď už vysvětlovat nebudeme.

Programy

Zatím jsme pracovali pouze s interpretem. Nyní si vyzkoušíme psát celé programy, aby se nám kód při vypnutí interpretu nesmazal a také abychom mohli psát i složitější kód.

V menu interpretu zvolte možnost vytvořit nový soubor a někam ho uložte jako `0-prvni.py`. **POZOR:** Je důležité, aby programy měly příponu `.py`, jinak vám nebudou fungovat! Do souboru uložte následující program:

```
1 | print "Gratulujeme! Uspesne jste spustili svuj prvni program!"
2 | zprava = "Tato zprava byla vytvorena programem 0-prvni.py"
```

V editoru pak zvolte možnost **Run > Run module**. Otevře se vám zpět interpret a v něm se spustí váš kód. V případě programu `0-prvni.py` vás tedy uvítá nadšená zpráva a dále můžete používat interpret. Všimněte si, že program kromě vypsání textu také uložil zprávu do proměnné `zprava`. Zkuste si tedy v interpretu tuto proměnnou vypsát:

```
1 | print zprava
```

Všechno, co program vytvořil, je vám tedy dostupné i potom, co skončil. Toho můžete později využít například při zkoušení jednotlivých částí vašich programů, hledání chyb a podobně.

Booleovské proměnné

Důležitou součástí programu jsou i pravdivostní, tzv. booleovské proměnné. Ty mohou mít hodnotu `True` (pravda) nebo `False` (nepravda). Můžeme je ukládat do proměnných stejně jako čísla nebo textové řetězce. V programu lze booleovské proměnné vytvářet pomocí porovnávání dvou jiných proměnných

```
1 | boo = 10 < 15
```

K dispozici jsou v Pythonu operátory `<`, `>` (ostře menší/větší), `<=`, `>=` (menší nebo rovno/větší nebo rovno). Dále lze použít operátory rovná se `==`, nebo jsou různé `!=`. Pro rovnost je velmi důležité používat dvě rovnítka. Jedno rovnítko se totiž v Pythonu používá k přiřazení hodnoty do proměnné. Když zapomenete druhé rovnítko, nebude vám program fungovat.

Booleovské proměnné lze taky kombinovat. K tomu slouží příkazy `and` (a zároveň) a `or` (alespoň jeden). Oba se používají úplně stejně, viz následující kód:

```
1 | boo = True or False
2 | print "True or False = ", boo
```

Další možností, jak pracovat s booleovskými proměnnými, je příkaz `not`. Ten se vyhodnotí na přesně opačnou hodnotu, než má jeho argument:

```
1 | foo = not True
2 | print "not True = ", foo
```

Více příkladů, jak se dají používat booleovské proměnné najdete v programu `1-boolean.py`.

Podmínky

Abychom mohli vytvářet i složitější programy, musíme umět běh programu větvit. Nejjednodušším způsobem, jak to udělat jsou podmíněné příkazy. Jejich funkcionality je jednoduchá: napíšeme nějakou podmínku, a pokud platí, tak se provede nějaký kód. Pokud neplatí, tak se provede jiný.

```
1 | cislo = input()
2 | if cislo > 0:
3 |     print "Cislo je vetsi nez 0"
4 | else:
5 |     print "Cislo neni vetsi nez 0"
```

V programu tedy napíšeme klíčové slovo `if`, za něj podmínku a řádek ukončíme dvojtečkou. Další řádek pak odsadíme pomocí mezerů. Příkaz na tomto řádku se provede pouze v případě, že podmínka byla splněna (v našem případě tedy pouze pokud jste napsali číslo větší než 0). Pokud

napišeme víc řádků za sebou, které budou mít stejné odsazení, tak všechny tyto řádky se provedou pouze v případě, že podmínka platila.

Další částí podmíněného příkazu je řádek `else:`. Ten se píše opět bez odsazení. Všimněte si, že zase končí dvojtečkou. To znamená, že následující řádek musí zase být odsazený. Tento (a případně i další odsazené řádky) se naopak provedou pouze v případě, že podmínka splněna nebyla. Po několika odsazených řádcích pokračujeme řádky bez odsazení, které se opět provedou vždy, ať už podmínka dopadla jakkoliv.

```
1  if cislo > 10:
2      print "Cislo je vetsi nez 10, zmensuji ho na 10"
3      cislo = 10
4  if cislo < 0:
5      cislo = 0
6      print "Zadali jste cislo mensi nez 0! Nahrazeno nulou."
7  print "Upravene cislo je ", cislo
```

Další příklady použití podmínek najdete v programu `2-podminky.py`

Funkce

Další velmi užitečnou částí programů jsou funkce. S jejich pomocí dělíme program na menší části, aby se nám v něm lépe orientovalo. Zároveň také zabraňujeme tomu, abychom stejnou funkcionalitu psali několikrát za sebou na čtyřech různých místech v programu.

```
1  def secti(prvni, druhe):
2      soucet = prvni + druhe
3      return soucet
4
5  vysledek = secti(4, 2)
6  print "Vysledek je ", vysledek
```

Pro definici funkce použijeme klíčové slovo `def`. Za ním pak název funkce, kterou chceme definovat (v našem případě `secti`). Do závorčky pak napíšeme seznam proměnných, které musíme zadat (v našem případě jsou to dvě). První řádek definice pak ukončíme znovu dvojtečkou. Stejně jako u podmínek, i pro funkce musí být každý další řádek, který patří k funkci, odsazený pomocí mezer. Na posledním řádku pak bude klíčové slovo `return` a nějaká hodnota (číslo, booleovská hodnota, text, ...).

Pokud budeme chtít tuto funkci v programu použít, pak jednoduše zavoláme funkci pomocí názvu a do závorčky zadáme požadované hodnoty (v našem případě dvě čísla). Funkce něco udělá a potom nám vrátí hodnotu, kterou vypočítala. Tuto hodnotu si můžeme uložit do jiné proměnné a dále s ní pracovat.

Pokud příklad nahoře napíšete do souboru a soubor spustíte, pak vám vypíše `Výsledek je 6`. Kromě toho ale také definuje funkci `secti(prvni, druhe)`, kterou můžete i nadále používat. Zkuste si do interpretu zadat příkaz

```
1  | print secti(3, 4)
```

Pokud chceme, aby funkce pouze něco udělala a nezajímá nás výsledek (funkce například jen vypíše text na obrazovku, nastaví nějakou hodnotu apod.), tak můžeme vynechat řádek `return`. Takováto funkce potom nevrací nic a nazýváme ji také procedura.

Příklady na to, jak vytvářet a používat funkce, najdete v programu `3-funkce.py`.

Seznamy

Zatím jsme se naučili ukládat jednotlivé hodnoty do proměnných. Často ale v programech potřebujeme uložit hodnot hodně a nebo dokonce ani dopředu nevíme, kolik jich vlastně bude. Pro takové případy jsou v Pythonu k dispozici seznamy.

```
1 seznam = [1, 2, 3, 4]
2 print "seznam =", seznam
3 print "seznam[1] =", seznam[1]
```

Seznam je schránka, která v sobě obsahuje několik dalších hodnot. Vytvoříme ji pomocí hranatých závorek, do kterých napíšeme jednotlivé hodnoty oddělené čárkami. Se seznamem můžeme pracovat buď jako s celkem, nebo se můžeme odkazovat na jednotlivé položky ze seznamu pomocí hranatých závorek. Tyto položky jsou v seznamu seřazeny postupně tak, jak jsou napsány v definici seznamu. Je také důležité vědět, že prvky se číslují od nuly. V našem příkladu tak volání `seznam[1]` vrátí druhý prvek ze seznamu, tedy dvojku.

Do seznamů můžeme i přidávat a odebírat prvky. Přidáváme pomocí příkazu `append(prvek)`. Všimněte si, že `append` je pomocí tečky připojeno za název seznamu, ke kterému chceme prvek připojit. Naproti tomu odstranění prvku se dělá pomocí klíčového slova `del`. Za něj pak přidáme název seznamu, ze kterého odstraňujeme a do hranatých závorek pak index prvku, který odstraňujeme (v následujícím příkazu tak indexem 2 odstraníme třetí prvek ze seznamu, tedy trojku).

```
1 seznam.append(5)
2 print "Seznam po pridani prvku =", seznam
3 del seznam[2]
4 print "Seznam po smazani prvku 2 =", seznam
```

Kromě čísel do seznamu můžeme ukládat i všechny ostatní hodnoty a to včetně dalších seznamů. Můžete si to sami vyzkoušet nebo se podívejte do programu `4-seznamy.py`.



Cykly

Posledním příkazem, který si v tomto úvodníku popíšeme, budou cykly. Nejjednodušším z nich je `while`. Zapisuje se velmi podobně jako příkaz `if`, a to

```
1 i = 0
2 while i < 10:
3     print "i =", i
4     i = i+1
5 print "Cyklus while ukoncen, vysledna hodnota i je", i
```

Za klíčové slovo `while` napíšeme nějakou podmínku a řádek ukončíme dvojtečkou. Následující odsazené řádky se provedou pouze tehdy, pokud podmínka platila. Na rozdíl od podmíněných příkazů se po provedení celého těla cyklu program vrátí zpátky na začátek cyklu. Potom znovu vyhodnotí, jestli podmínka stále platí. Pokud ano, tak tělo cyklu provede znovu. To opakuje, dokud podmínka nepřestane platit. Potom pokračuje kódem, který následuje za odsazenými řádky.


```

1 seznam = ["a", "bb", "ccc", "dddd"]
2 for prvek in seznam:
3     print "Dalsi prvek seznamu je", prvek
4     print "Cyklus for ukoncen"

```

Dalším druhem cyklu je `for`. Tomu zadáme nějaký seznam a cyklus pro každý prvek tohoto seznamu provede jednou kód, který k němu přísluší. Zapisujeme jako `for`, za kterým následuje nějaký název proměnné (v našem případě `prvek`). Dále klíčové slovo `in` a seznam, kterým chceme procházet. Následuje už dobře známá dvojtečka a několik řádků s odsazením.

Další příklady cyklů najdete v programu `5-cykly.py`.

Zadání úloh

Ted' když už víte, jak se pracuje s Pythonem, tak se můžeme vrhnout na samotné zadání úloh. Na stránkách `ksi.fi.muni.cz` si k aktuální sadě můžete stáhnout zadání programů jednotlivých úloh. Vaším úkolem bude naprogramovat jednu nebo více funkcí, jejichž funkcionalita bude popsána v zadání. Váš kód budete psát do souboru `reseni.py`, který zatím obsahuje pouze hlavičky jednotlivých funkcí. Každá úloha obsahuje soubor `run.py`, pomocí kterého se daná úloha spouští, abyste si mohli ověřit správnost vašeho řešení.

Stejně jako v předchozích ročnících, hlavní částí vašeho řešení je popis toho, co jste vlastně udělali, jak jste postupovali, proč je váš postup správný. Ke každé úloze tedy odevzdáváte nejen soubor `reseni.py`, ale i text s vysvětlením řešení (více na stránce <http://ksi.fi.muni.cz/clanek?id=4>).

Příklad 0: Ukázkový příklad (0 bodů)

Karlík se rozhodl, že si procvičí svou zručnost a postavil si stroj pro reprezentaci čísel. Tento stroj je jednoduchý – číslo je reprezentováno jako sloupec, kde hodnota čísla určuje výšku sloupce.

Když se Karlík na svůj stroj koukal, rozhodl se ho pořádně vyzkoušet a začal různě přehazovat čísla. Vymyslel si různé úkoly a hledal k nim co nejvíce postupů, které potom zkoumal. Jeden z nich se mu ale pořád nedařil: uspořádání náhodně rozházených čísel. A potřeboval by proto vaši pomoc.

Vaším úkolem je doplnit funkci `serad(pole, delka)` tak, aby pro pole `pole` o délce `delka` dokázala pomocí metody `pole.prohod(index1, index2)` pole vzestupně seřadit. Vzestupně seřazené pole čísel je takové pole, kde pro každá dvě po sobě jdoucí čísla platí, že to levé je menší.

Karlíkův stroj je navíc poměrně jednoduchý. Nelze v něm vytvářet žádná pole. I vy proto ve funkci `serad` nesmíte vytvářet žádná nová pole a můžete používat pouze to, které jste dostali na začátku. Kód algoritmu nezapomeňte dobře okomentovat!

Řešení

Pokud není pole správně seřazené, tak to znamená, že některé dva sousední prvky jsou ve špatném pořadí. Nejjednodušší proto bude, když zkontrolujeme všechny sousedící dvojice.

Kontrolu dvojic budeme provádět zleva doprava. Pokud najdeme nějakou špatnou, tak ji prohodíme a pokračujeme s kontrolou. Po prohození se ale pole změnilo, takže bude potřeba provést další kontrolu celého pole. Proto si potřebu další kontroly vždy poznačíme do booleovské proměnné `zmena`. Pokud mají prvky stejnou hodnotu, tak je nevyměňujeme.

Kontrolu budeme opakovat tak dlouho, dokud nebude pole správně seřazené.

Nyní potřebujeme dokázat, že algoritmus funguje správně. Z návrhu algoritmu je zřejmé, že pokud se program nezacyklí a ukončí se, tak bude pole seřazené správně. Podle zadání totiž každá sousedící dvojice musí být správně seřazená. Náš algoritmus skončí pouze tehdy, když neexistuje žádná dvojice, která by byla seřazená špatně. Nyní tedy stačí ukázat, že se program nezacyklí.

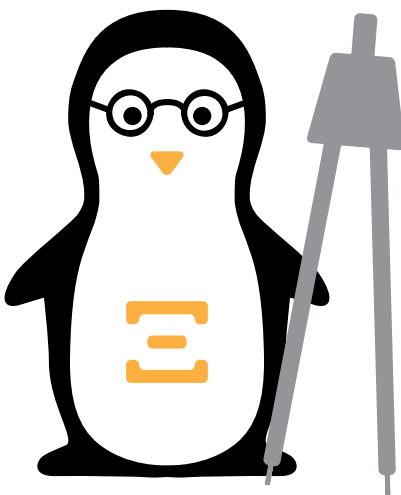
Víme, že některý z prvků v poli je největší (případně je takových prvků víc, v tom případě vybereme ten nejvíce vpravo). Tento prvek se po prvním průchodu kontroly dostane na konec pole. Všechny prvky vpravo od něj jsou totiž menší než on sám, takže pokud ho srovnáme s jeho pravým sousedem, tak ho posuneme o jedna doprava. Následující kontrola ale kontroluje dvojici o jedna doprava, takže ho zase srovná s menším prvkem a prohodí. Takhle se dostane až na konec pole.

Pro druhé kolo kontroly si představme, že poslední (tedy ten největší) prvek už v poli není. Kontrola ho bude zkoumat až při úplně posledním pokusu, takže do té doby nemá na výpočet vliv. Stejně jako při předchozím průchodu se bude druhý největší prvek posouvat až na konec pole. Až bude na předposledním místě, srovná ho kontrola s posledním prvkem, který je ale větší nebo stejně velký, takže zůstanou oba na místě.

Podobně pak probíhá každá další kontrola, takže po k -tém kole kontrol bude posledních k prvků na svém místě. Nejvíce tedy proběhne n kol, kde n je počet prvků pole.

Složitost jedné kontroly je konstantní, složitost prohození také. V každém kole kontrol proběhne n kontrol, kol bude nanejvýš n . Z toho nám vyplývá, že složitost algoritmu je $\mathcal{O}(n^2)$. Protože jediné pole které používáme je to vstupní, je prostorová složitost algoritmu $\mathcal{O}(n)$, takže jsme hotovi².

Poznámka pro náročné řešitele: Zajisté jste poznali, že se jedná o algoritmus bubble sort. Pokud máte pocit, že úlohu dokážete vyřešit efektivněji, můžete si váš algoritmus zkusit naimplementovat a řešení – včetně precizního popisu algoritmu a komentářů v kódu – nám poslat na email `maara@mail.muni.cz`. Úloha není bodovaná, ale autory nejzajímavějších řešení čeká odměna!



²Více o časové a prostorové složitosti najdete na našem webu v článku „O prostorové a časové složitosti“.

E Zadání 1. sady úloh KSI (termín odevzdání: 3. 11. 2014)

Řešení zasílejte pomocí internetového systému na adrese <http://ksi.fi.muni.cz>.

Příklad 1: Karlík si chce hrát! (10 bodů)

Náš Karlík by si pořád jenom chtěl hrát. Nám však začíná semestr a už nemáme tolik času, abychom si s ním celé dny mohli hrát. Pomoz nám!

Jeho oblíbenou hrou je hra „Uhádni mé číslo“. Tuto hru jistě znáš i ty. Karlík si myslí číslo z určitého rozsahu. Tvým úkolem je jeho číslo uhodnout. Pokud se netrefíš, tak ti řekne, jestli bylo tvoje číslo větší či menší než jeho hádané. Karlík je však perfekcionista a rád hraje jenom s hráčem, který hádá jeho čísla efektivně.

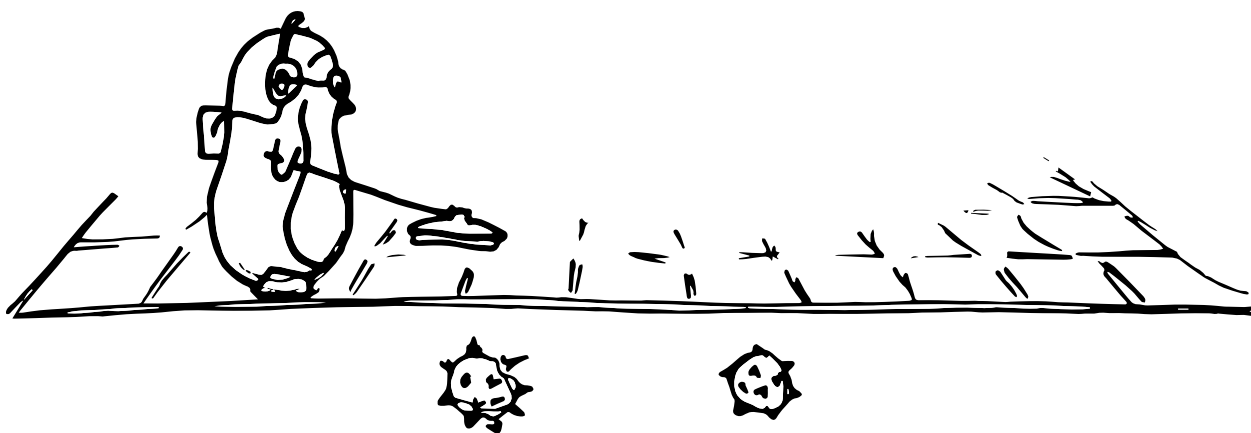
Tvým úkolem bude vymyslet a implementovat algoritmus pro hádání těchto čísel. To provedeš tak, že naimplementuješ funkci `hadej`. Až budeš hotov, můžeš svou funkci nechat otestovat pomocí souboru `run.py`.

Můžeš si s Karlíkem zahrát buď jenom jednu hru, kde ti ukáže průběh hry, nebo ho nechat funkci pořádně otestovat – poté ti Karlík napíše statistiku vypovídající a efektivitě tvého algoritmu.

Odevzdávej svůj soubor `reseni.py` a stručný komentář k algoritmu, kde vysvětlíš jak pracuje a jakou má časovou složitost.

Příklad 2: Karlík na minách (10 bodů)

Jelikož Karlík je velký fanoušek starých počítačových her, rozhodl se jednoho dne hrát miny. Aby se však Karlík zabavil pořádně, chce uhrát co nejvíce her, a to se mu podaří tak, že v každé hře klikne co nejméněkrát to jen jde.



Zde přichází vaše chvíle, kdy Karlíkovi pomůžete se spočítáním, kam má kliknout tak, aby hru dohrál na co nejméně kliků.

Pravidla min jsou následovná. Hrací plán je velikosti $N \times M$, kde na každém políčku je buď číslo, nebo mina. Číslo určuje, kolik min se nachází v sousedství daného políčka, tedy čísla jsou mezi 0 a 8 včetně. Dvě políčka sousedí tehdy, pokud sdílejí hranu nebo roh. Při kliknutí se odkrývají jednotlivá políčka takovým způsobem, že pokud políčko obsahuje 0, odkrývají se i sousední políčka, dokud se nenarazí na políčko obsahující jinou hodnotu než 0. Pokud byla hodnota políčka jiná než 0, odkrývá se pouze toto políčko. V případě, že hráč klikne na minu, tak hru prohrává. Hra končí, pokud byla odhalena všechna políčka, kde se nenachází miny.

Příklad jednoho kliknutí na plánu, kde X jsou miny a K je pozice kliknutí:

X	.	.	X	.	.	.	X	X	.
.	.	.	.	X
.	.	K	.	.	X
.	X	.
.

Na plánu nejsou žádné miny sousedící s místem kliknutí, tedy se na pozici objeví 0 a překlopí se i sousední políčka. Výsledek bude následující:

X	.	.	X	.	.	.	X	X	.
1	1	1	2	X
0	0	0	1	2	X
0	0	0	0	1	1	1	1	X	.
0	0	0	0	0	0	0	1	.	.

Vaším úkolem bude vymyslet a naprogramovat algoritmus, který spočítá, kde je třeba klikat do herního pole tak, aby Karlík vyhrál hru v co nejméně krocích. Váš algoritmus dostane na vstup pole nul a jedniček, kde 0 značí prázdné políčko a 1 políčko s minou. To, jestli políčko sousedí s nějakou minou, si už budete muset dopočítat sami. Vráťte seznam pozic ve tvaru `[klik0, klik1, ..., klikn]`, kde jednotlivé kliky jsou dvouprvkové seznamy `[x, y]`, tj. kliknete na pole`[x][y]` (dejte si pozor na číslování od 0). Pro lepší reprezentaci hry vám bude sloužit simulátor, který odsimuluje vaše kliky na herním plánu. Bližší popis zadání najdete v komentářích kódu.

Příklad 3: Sirky (10 bodů)

Najděte nejbližší osobu a nejbližší krabičku zápalek a zahrajte si následující hru: Vysypte sirky na hromádku a střídavě z ní odebírejte. V každém tahu může hráč odebrat 1, 2, nebo 3 sirky. Prohrává ten, kdo už nemůže provést žádný tah, protože na něj sirky nezbyly.

Možná vás hra brzy přestane bavit, protože odhalíte, že v závislosti na počtu sirek, se kterými se začíná, má buď začínající, nebo nezačínající hráč jednoduchou vyhrávající strategii, tedy takovou strategii, která mu zajistí vítězství, ať hraje protihráč jakkoliv.

Hru můžeme zobecnit na jiné povolené tahy (tahy rozumíme možné počty odebraných sirek). Pokud mezi povolenými tahy není 1, je možné prohrát, i když na hromádce ještě nějaké sirky zbývají. Například pokud jsou povolené tahy 3, 5, 7 a zbývají 2 sirky, nemůžeme provést žádný tah a tedy prohráváme.

První částí úlohy je nalézt, zdůvodnit optimálnost a implementovat strategii pro zmíněnou *konkrétní variantu*, v níž jsou povolené tahy 1, 2 a 3. Optimální strategie je taková, která když dostane příležitost zvítězit, tak ji využije.³

Druhou částí úlohy je vyřešit *obecnou variantu*. Zde už ale není možné nalézt obecně fungující strategii, kterou jenom zapíšete do programu, jako v konkrétní variantě. Musíte tedy vymyslet algoritmus, který podle počáteční velikosti hromádky a seznamu povolených tahů optimální strategii vypočítá.

Vášim úkolem tedy bude dopsat funkce `vymysliTahKonkretni` a `vymysliTahObecna` v souboru `reseni.py`.

I pokud vyřešíte pouze jednu variantu, nebojte se částečné řešení odevzdat (za každou z variant můžete získat polovinu celkového počtu bodů). Kompletní řešení by mělo obsahovat:

- popis konkrétní strategie včetně zdůvodnění optimality
- popis algoritmu pro výpočet obecné strategie (včetně zdůvodnění optimálnosti)
- soubor `reseni.py` s implementovanými a okomentovanými strategiemi

Funkčnost vaší strategie si můžete vyzkoušet pomocí přiloženého programu:

```
$ python sirky.py
Vyber podulohu:
  - (0) konkretni
  - (1) obecna
Poduloha: 1
Vyber strategii:
  - (0) clovek
  - (1) hloupa
  - (2) chytra
Strategie zacinajiciho hrace: 1
Strategie nezacinajiciho hrace: 1
Pocet sirek na hromadce: 5
Povolene tahy: 2, 3, 4

### kolo 1, hrac 1
Pocet sirek: 5
Odebrano sirek: 2

### kolo 2, hrac 2
Pocet sirek: 3
Odebrano sirek: 3

### kolo 3, hrac 1
Pocet sirek: 0
Nelze provest zadny tah!

### Vyhral nezacinajici hrac ###
```

³Pokud zvítězit nemůže (za předpokladu, že soupeř bude hrát optimálně), tak může vaše strategie hrát jakkoliv, třeba náhodně. Jen pozor na to, aby se vždy jednalo o jeden z povolených tahů a nepokusili jste se sebrat více sirek, než je na hromádce.

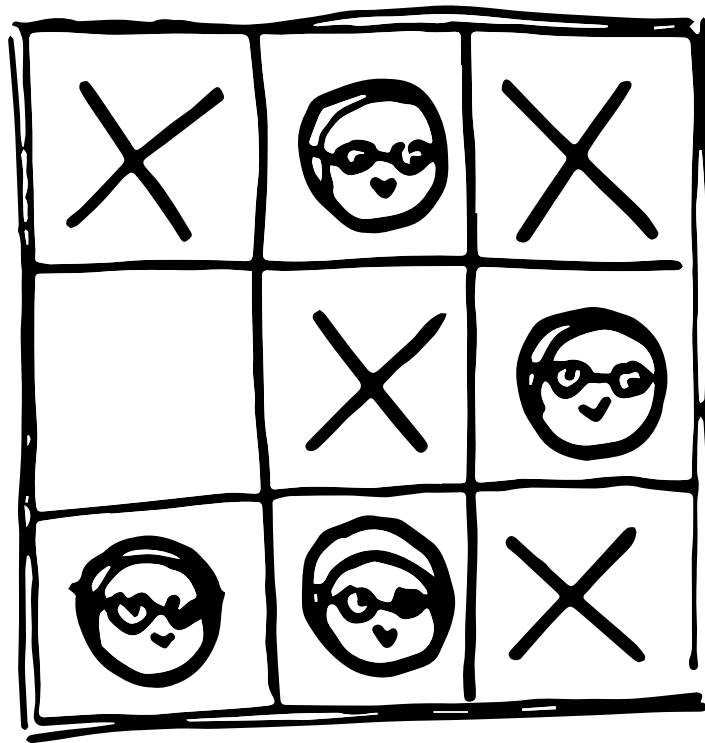
Příklad 4: Karlík hraje piškvorky (10 bodů)

Karlík si už dlouho chtěl zahrát piškvorky, nikdo z organizátorů ale na něj neměl čas. Rozhodl se tedy, že si zahraje proti algoritmu. Karlík ovšem není zrovna zdatný hráč piškvorek. Dokonce si všiml, že algoritmus ani nehraje férově.

Hrají se základní piškvorky na ploše 3×3 . Každý tah je reprezentován jako uspořádaná dvojice souřadnic (x, y) , kde $(0, 0)$ značí horní levý roh našeho herního pole. Hráči jsou reprezentováni znakem O [velké o] (Karlík) a X , prázdné pole znakem $_$. Hráči se postupně střídají ve svém tahu. Ve svém tahu hráč může vyhodnotit svůj tah a provede jej právě oznámením výše uvedené uspořádané dvojicí souřadnic. Správný tah je takový, kdy souřadnice mění políčko z $_$ na X nebo O . První tah v našich piškvorkách vždy náleží Karlíkovi.

Vášim úkolem je pomoci Karlíkovi s touto hrou. Máte za úkol napsat funkce `platnyTah` a `opravTah` takové, že `platnyTah` vrací hodnotu `True` v případě, že tah byl platný (podle výše uvedených pravidel) nebo `False`, pokud platný nebyl. Funkce `opravTah` má za úkol bez ohledu na platnost tahu najít tah, který by byl v daný moment nejvýhodnější (v případě více takových tahů je jedno, který vrátíte).

Váš postup při řešení problémů zformulujte a alespoň intuitivně naznačte, proč je správný a adekvátně náročný příp. napište, jak by jste ho ještě zdokonalili.



Příklad 5: Složité výpočty (10 bodů)

Karlík si chtěl vytvořit prográmeček, který by mu počítal mocniny dvojky. Moc se mu to ale nepovedlo, zkusil to pětkrát a žádná z funkcí mu nevrací to, co by chtěl. Vy byste vypisování mocnin dvojky jistě hravě zvládli, proto po vás budeme chtít něco jiného. Karlíkovy funkce jsou si na první pohled dost podobné, a přesto se od sebe některé výrazně liší i v něčem jiném než ve vypočítaných hodnotách, a to v časové složitosti. Když už se s nimi Karlík tak nadřel, tak

poslouží alespoň jako zadání úlohy – u každé z pěti přiložených funkcí `karlikuvP#` určete jejich asymptotickou časovou složitost a své rozhodnutí zdůvodněte.

A malá rada na závěr – pokud si budete funkce spouštět, nelekejte se, když se vám občas zasekne výpočet. Karlík se opravdu moc nesnažil a paměťová složitost některých funkcí je fakt velká (vy však určujete časovou).

A to je z této sady KSI vše. Přejeme ti hodně úspěchů při řešení, a když budeš mít jakékoliv otázky, neváhej se na nás obrátit e-mailem na adresu `ksi@fi.muni.cz` nebo v diskuzním fóru na našich webových stránkách.

Termín odevzdání 1. sady úloh KSI: 3.11.2014

`http://ksi.fi.muni.cz`

