Korespondenční Seminář z Programování

34. ročník **KSP** **Březen 2022**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám čtvrté číslo hlavní kategorie 34. ročníku KSP.

Opět se můžete těsit na dvě praktické a dvě teoretické úlohy. Tentokrát nepřidáváme žádnou těžší úlohu kategorie X, protože na **úlohu 34-3-X1 z minulé série** nám nepřišlo žádné správné řešení. Tak jsme její **termín prodloužili do konce této série** a **přidali nápovědu do jejího zadání**. Nakonec je zde tradičně pokračování Manimového seriálu, tentokrát s podtitulem 3D.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek.** Úspěšným řešitelem se stává ten, kdo získá za celý ročník (této kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepku na notebook a možná i další překvapení.

Termín série: **neděle 10. dubna ve 32:00 (tedy další ráno v 8:00)**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/h/odevzdavatko/*

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha

Odměna série: Každému, kdo z této série **získá alespoň 42 bodů**, pošleme sladkou odměnu.

Čtvrtá série třicátého čtvrtého ročníku KSP

34-4-1 No pun indented! — 9 bodů

Kristýna právě napsala řešení úložky z KSP-X v Pythonu. Ovšem stalo se nemyslitelné. Její oblíbený editor (který si sama napsala předchozí noc) jí při ukládání souboru smazal všechny mezery na začátku řádků. *Co teď s tím?* pomyslela si Kristýna. *Psát znovu se mi to nechce, tak přeci nemůže být tolik možností, jak doplnit mezery, aby se jednalo o validní Python. Projdu všechny a vyzkouším, která z nich na zadaných vstupech funguje.*

Ukažte, že se Kristýna mýlí, a zjistěte, že pro její kód je více validních odsazení, než zvládne vyzkoušet do termínu úlohy. Spočítejte, kolik takových odsazení je.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Na vstupu je nejprve číslo N, následuje N řádků programu. Řádky na svém začátku ani konci nemají mezery (uprostřed však být mohou) a každý obsahuje alespoň jeden znak. Na každém řádku je buď *řídící příkaz* nebo *normální příkaz*. Řádky řídících příkazů (jako například cykly, podmínky, definice funkcí, ...) se poznají tak, že končí dvojtečkou.

První příkaz v programu nesmí mít žádné odsazení. Ve validním programu musí za každým řídícím příkazem následovat řádek s o jedna vyšším odsazením. Za normálním příkazem se nesmí velikost odsazení zvýšit. Může ovšem zůstat stejná nebo se snížit, a to o libovolně mnoho úrovní. Je zaručeno, že poslední řádek na vstupu není řídící příkaz.

Za validní programy považujeme všechny takové, které splňují výše popsané podmínky. Neřešíme jiná pravidla platná v Pythonu, například jestli jsou splněny vztahy mezi řídícími příkazy, nebo jestli jsou v daném odsazení definované všechny identifikátory. Kristýna totiž i takovéto programy vyzkouší spustit a až pak zjistí, že program spadne.

Jako výstup odevzdejte počet validních odsazení. Protože toto číslo může snadno narůst, tak na výstup vypište jeho zbytek po dělení $10^9 + 7$ (což je šikovně velké prvočíslo). Zbytek po dělení doporučujeme aplikovat už v průběhu výpočtu při sčítání a násobení, výsledek to nezmění a zabrání to přetečení datového typu.

Ukázkový vstup:

```
12
#!/usr/bin/env python3
from random import *
seed(42)
print(100)
for i in range(100):
try:
s = "".join(choice("abc") for i in range(10))
finally:
if random()<0.4:
print(s+":")
elif True:
print(s)
```

Ukázkový výstup:

```
12
```

I když jen jeden způsob odsazení vytvoří skutečně validní Python, naší definici vyhoví 12 různých odsazení. Kdybychom si vzali onen validní Pythoní kód a spustili jej, všimneme si, že by opět vytvořil zadání pro tuto úlohu. Prozradíme, že na něj by správná odpověď byla 792 306 071.

34-4-2 Písemka z analýzy — 10 bodů

Profesor Nilpferd učí matematickou analýzu a neustále udržuje své studenty ve střehu nenadálými písemkami. Studenti vymýšlejí čím dál šílenější hypotézy (hippotézy?) o tom,

jak se rozhoduje, kdy písemku zadá a kdy ne. Alice si je jistá, že je to podle fáze měsíce. Bob má podezření na večerní červánky. Podle Cyrila se zaručeně řídí ligou ve famfrpálu.

Každý večer se zeptáte svých spolužáků. Každý řekne svůj názor na to, zda bude zítra písemka. Vy si na základě vyslechnutých tipů vytvoříte vlastní názor a podle něj jdete buďto klidně spát, nebo se začnete na zítřek zuřivě učit. Dopoledne se dozvíte, zda písemka nastala nebo ne. Podle toho můžete upravit, jak se budete další večer chovat. Toto se opakuje každý den v semestru.

Svou předpověď můžete popsat algoritmem. Ten každý den dostane všech n tipů spolužáků, vydá předpověď, a pak se dozví, jestli se vyplnila. Podle toho si upraví svůj vnitřní stav pro další den (například názor na to, jak dobrý je tip kterého spolužáka).

Myslíte si, že mezi spolužáky je alespoň jeden, který uhodl, jak se profesor rozhoduje, a tím pádem tipuje vždy správně. Jen vědět, kdo to je...

Vymyslete co nejlepší předpovědní algoritmus, tedy takový, který za těchto předpokladů udělá co nejméně chyb v závislosti na počtu spolužáků n a počtu dní semestru d. Snažte se minimalizovat počet chyb v nejhorším případě. Můžete předpokládat, že d je mnohem větší než n. Časová složitost algoritmu nás nezajímá.

34-4-3 Korelace nejsou tranzitivní — 11 bodů

Ondra si nedávno přečetl dvě studie – že čím více lidé pijí mléko, tím vyšší v průměru jsou, a že vyšší lidé jsou v průměru úspěšnější. Nyní přemýšlí, jestli by neměl začít pít více mléka.

Je na vás, abyste mu vysvětlili, že takové uvažování nedává smysl. Nejen že korelace neimplikuje kauzalitu, ale to, že spolu něco koreluje, vůbec není tranzitivní. Je klidně možné, že čím více lidé pijí mléko, tím jsou v průměru méně úspěšní!

Máte k dispozici m studií, kde každá ukazuje silnou korelaci mezi dvěma veličinami (např. čím více lidé pijí mléko, tím jsou v průměru úspěšnější), a vaším cílem je najít posloupnost korelujících veličin takovou, že skončíte u negace první veličiny, abyste demonstrovali, že korelace nejsou tranzitivní.

Například posloupnost:

- čím jsou lidé bohatší, tím více peněz utrácí za víno,
- čím více peněz utrácí za víno, tím více vína pijí,
- čím více vína pijí, tím více alkoholu konzumují,
- čím více alkoholu konzumují, tím jsou méně bohatí,

by mohla být korektním řešením.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku dostanete počet veličin n, kterých se studie týkají, a počet studií m. Na každém z následujících m řádků bude trojice x_i, y_i a k_i, kde x_i a y_i udává, kterých dvou veličin se studie týká, a k_i je vždy buď 1 nebo -1, což udává, kterým směrem byla korelace naměřena. Číslo 1 odpovídá kladné korelaci (například čím jsou lidé bohatší, tím více peněz utrácí za víno) a -1 záporné korelaci (například čím lidé více konzumují alkohol, tím jsou méně bohatí).

Uvedené korelace jsou obousměrné. Pokud x koreluje s y, koreluje také y s x, a to se stejným znaménkem.

Platí, že $0 \leq x_i, y_i < n$ (veličiny číslujeme od nuly), $\forall i : x_i \neq y_i$ a neexistují dvě různé studie, které by se týkaly stejné dvojice veličin.

Formát výstupu: První řádek výstupu bude obsahovat počet veličin ve vašem cyklu. Na dalším řádku pak budou tyto veličiny. Musí platit, že mezi každými dvěma sousedními veličinami existuje studie a poslední veličina v je rovná té první. Navíc musí posloupnost korelací vyjadřovat, že čím více v, tím méně v.

Pokud existuje více řešení, vypište libovolné.

Ukázkový vstup:

```
4 5
0 3 -1
2 3 -1
1 2 1
1 0 -1
2 0 1
```

Ukázkový výstup:

```
5
0 3 2 1 0
```

34-4-4 Geometrie — 15 bodů

Jirka upekl dort a chtěl se o něj podělit se svými kamarády. Jelikož to není žádný cukrář, tak jeho dort měl poměrně neobvyklý nepravidelný tvar. Dokonce ani nebyl konvexní. Říkal si, že že by této vlastnosti dortu mohl využít. Rád by dort rozdělil na co nejvíce částí (neví totiž, kolik jeho kamarádů ještě dorazí) za použití pouze jednoho řezu.

Na vstupu jsou souřadnice N bodů nekonvexního N-úhelníku. Můžete předpokládat, že žádné tři body nejsou kolineární, tedy neleží na jedné přímce, a žádné dva body nemají shodnou ani jednu ze souřadnic.

Popište algoritmus, který spočítá, na kolik částí lze rozdělit N-úhelník pouze za pomoci jednoho řezu přímkou.

Lehčí varianta (za 8 bodů): Pro zisk části bodů můžete předpokládat, že optimální řez je rovnoběžný s osou x.

34-4-S Manimujeme – 3D — 15 bodů

Seriál je tento ročník zaměřen na generování animací pomocí Pythoní knihovny Manim a obsahuje tedy řadu animací, které se do formátu PDF nehodí. Proto je dostupný pouze na webu.[1]

Tomáš Sláma

[1] http://ksp.mff.cuni.cz/viz/34-4-S

Recepty z programátorské kuchařky: Geometrie

Geometrické algoritmy

V dnešním díle našeho kuchařkového speciálu se budeme učit vařit geometrické problémy. A co že si představujeme pod pojmem geometrický problém? Trochu analytické geometrie, například zjištění, na které straně orientované přímky bod leží, trocha plotů, neboli konvexních obalů, a obecně mnoho zametání.

V celé kuchařce se omezíme pouze na dvourozměrné problémy, tedy na algoritmy v rovině. Některé postupy se dají zobecnit pro trojrozměrné, a většinou i pro n-rozměrné problémy, ale to je již nad rámec této kuchařky.

Geometrické základy

Nejdříve trocha středoškolské analytické geometrie pro ty, kdo ji ještě neměli. Ostatní mohou tuto sekci přeskočit.

Každý bod v rovině můžeme určit jeho souřadnicemi vůči osám. Nejběžněji se používá takzvaný *kartézský souřadný systém,* tedy dvě na sebe kolmé osy označované jako x-ová osa (vodorovná) a y-ová osa (svislá). Obvykle se uvažuje, že hodnoty na osách rostou směrem doprava (osa x) a směrem nahoru (osa y), my se toho budeme v naší kuchařce držet.

Místo, kde se obě osy protínají, se označuje jako *počátek* soustavy souřadnic. Samotné *souřadnice* bodu zapisujeme jako dvojici čísel, která udávají, o kolik jednotek se musíme posunout ve směru které z os, abychom z počátku dorazili do bodu, kterému souřadnice patří. Počátek má souřadnice $[0, 0]$. Bod se souřadnicemi $[a, b]$ leží na pozici, kterou získáme tak, že se od počátku posuneme o a jednotek ve směru první osy (x-ové) a o b jednotek ve směru druhé osy (y-ové).

Vše ostatní funguje tak, jak jsme se učili při geometrii na základní škole, tedy úsečka je určena dvěma krajními body, obdélník čtyřmi a podobně. Ještě si ale řekneme, co je to vektor, a zavedeme některé další pojmy.

Často potřebujeme popsat vzájemnou polohu dvou bodů. Můžeme například udat jejich vzdálenost a směr (třeba jako úhel vzhledem k ose x). Praktičtější ale bývá říci, o kolik se liší jejich x-ové a y-ové souřadnice. To nám dá dvojici čísel, které říkáme *vektor.*

Pokud například k bodu $[1, 1]$ přičteme vektor $a = (2, -1)$, dostaneme se do bodu $[3, 0]$. Stejně tak, pokud odečteme například bod $[4, 2]$ od bodu $[1, 3]$, tak dostaneme vektor $b = (-3, 1)$ udávající jejich vzájemnou polohu.

Pomocí vektoru a bodu tedy lze určit přímku. Bod nám určí, kam umístit vektor, a vektor nám určí směr přímky z daného bodu. Tomuto vektoru se říká *směrový vektor,* nebo také někdy *směrnice,* dané přímky nebo úsečky.

Samotné vyjádření přímky nebo úsečky poté může být ve dvou tvarech. Prvním z nich je *parametrický tvar.* Základem je nějaký bod $A = [a_x, a_y]$. Od toho se ve směru směrového vektoru $u = (u_x, u_y)$ můžeme pohybovat libovolně a stále budeme na přímce. To nám vede na následující tvar, kde t je libovolný reálný parametr, neboli proměnná, za kterou si můžeme dosadit jakékoliv reálné číslo a vždy nám vyjde bod na přímce. Parametrický tvar vypadá takto:

$$x = a_x + tu_x$$

$$y = a_y + tu_y$$

To samé můžeme vyjádřit i vektorově, tedy $X = A + tu$.

Pro ilustrování funkce parametru, když bude $t = 0$, tak dostaneme výchozí bod přímky. Pokud poté budeme s parametrem hýbat od $-\infty$ do $+\infty$, dostaneme postupně všechny body na přímce.

Druhým způsobem zápisu je *obecný tvar přímky.* K jeho vyjádření budeme potřebovat kolmý vektor ke směrovému vektoru, tomu se také říká *normálový vektor.* V rovině ho získáme jednoduše. Pokud je $v = (v_x, v_y)$ směrnice přímky, tak vektor na něj kolmý má tvar $n = (v_y, -v_x)$. Jako poznámku pro zvídavé můžeme uvést, že *skalární součin* těchto vektorů, tedy součin po složkách ($v \cdot n = ab + b(-a)$), je roven 0, což je také jedna z definic kolmosti.

A jak tedy vypadá slibovaný obecný tvar přímky? Pokud je $n = (a, b)$ normálový vektor přímky, tak obecný tvar přímky je rovnice $ax + by + c = 0$. Dobře, a a b máme, jak ale zjistit c? Normálový vektor určuje směr, kterým přímka povede, ale stále ji můžeme libovolně posouvat. Potřebujeme ještě znát jeden bod, který na naší přímce leží, aby byla určená jednoznačně.

Když dosadíme souřadnice takového bodu do rovnice přímky s neznámou c, získáme tak rovnici pro c, kterou vyřešíme. A máme hotovo, známe hodnoty všech koeficientů v rovnici. Ještě si můžeme všimnout, že pro $c = 0$ prochází přímka počátkem.

Takovéto tvary se hodí nejen pro nějaké zapsání přímek, ale také pro *zjištění jejich průsečíku.* Když hledáme průsečík, hledáme vlastně místo, kde mají obě přímky navzájem stejné x-ové a y-ové souřadnice. A to vede na jednoduché soustavy lineárních rovnic, které jistě již vyřešit umíte.

Ještě si ale zdůrazněme rozdíl úseček oproti přímkám. V případě parametrického tvaru omezujeme velikost parametru t (například $t \in \langle 0, 1 \rangle$) a v případě obecného tvaru omezujeme rozsah jedné ze souřadnic (například $x \in \langle -2, 2 \rangle$). V případě, že bychom chtěli vyjádřit polopřímku, si parametr nebo souřadnici omezíme pouze z jedné strany.

Nakonec si ukážeme jednu základní aplikaci parametru a parametrického vyjádření úsečky. Jak snadno spočítat střed nějaké úsečky AB? V takovém případě není nic jednodušší, než si vzít vektor $B - A$, přenásobit ho parametrem $1/2$ (střed úsečky je v polovině její délky) a přičíst k bodu A. Triviální úpravou pak zjistíme, že střed úsečky můžeme spočítat jako aritmetický průměr jejích krajních bodů:

$$A + \frac{1}{2} \cdot (B - A) = \frac{A + B}{2}$$

Jako příklad na rozkoukání si ukážeme, jak zjistit, na které straně přímky leží bod.

Zjištění polohy bodu vůči přímce

Nejdříve si zavedeme pojem orientovaná přímka. Když budeme mít přímku určenou dvojicí bodů A a B, budeme se na ni dívat, jako kdybychom stáli v prvním bodě (bod A) a dívali se směrem ke druhému (bod B). Pak již máme jasně definovanou pravou a levou stranu a můžeme říci, kde vůči přímce bod leží.

Vezměme si tedy přímku určenou body A a B a bod X. Určíme si vektory $u = X - A$ a $v = B - A$ (s prvky u_x, u_y, respektive v_x, v_y) a porovnáme úhel mezi nimi.

Pokud jste už měli analytickou geometrii, určitě znáte vzoreček na výpočet úhlu mezi dvěma vektory. Vzoreček má tvar:

$$\cos\alpha = \frac{u \cdot v}{|u||v|}$$

Jeho nevýhodou je, že výpočet inverzní funkce $\cos^{-1}$ trvá dlouho. Je proto lepší použít jiný způsob výpočtu, kde si vystačíme pouze s násobením.

Tím jiným způsobem je výpočet determinantu matice určené těmito vektory. *Matice* je pouze tabulka, kde jsou vektory poskládány pod sebe (ta naše tedy bude velká 2 na 2 políčka).

Determinant matice této velikosti nám udává obsah rovnoběžníku určeného zadanými vektory. A navíc znaménko determinantu nám říká, jestli je úhel mezi vektory (měřený v kladném směru, tedy proti směru hodinových ručiček) menší než π, nebo větší než π.

Kdo se ještě s determinanty nesetkal, může brát následující vzorec pro výpočet determinantu matice dva krát dva jako kouzelnou formuli. Kdo přesto chce zdůvodnění, může si zkusit udělat rozbor všech vzájemných poloh dvou přímek (a jejich směrových vektorů), které mohou nastat. Po chvíli dojdete ke vztahu přesně odpovídajícímu následujícímu vzorečku:

$$d = u_x v_y - u_y v_x.$$

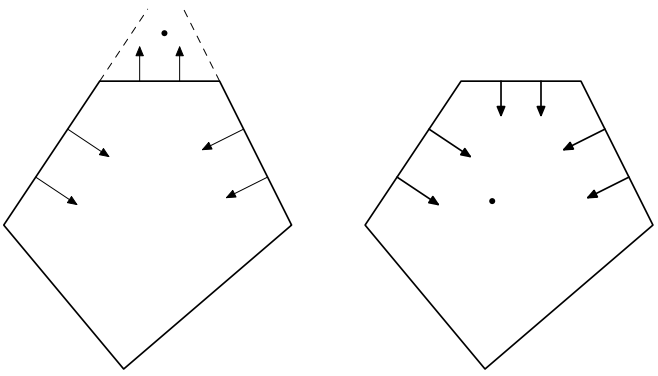
Pokud vyjde d kladné, je bod napravo od přímky, pokud vyjde d záporné, je bod nalevo od přímky, a konečně, pokud vyjde $d = 0$, tak bod leží na přímce.

Bod a konvexní mnohoúhelník

Konvexní mnohoúhelník je takový, který nemá žádný vnitřní úhel větší než 180°. Jinou definicí je, že pokud si zvolíme libovolné dva body v mnohoúhelníku a natáhneme úsečku mezi nimi, nikdy nám nevyleze z mnohoúhelníku ven.

Když už víme, co konvexní mnohoúhelník je, jak zjistíme, jestli nějaký bod leží v něm, nebo ne? Využijeme vlastnosti konvexnosti. Stačí nám jít po hranách na obvodu a zjišťovat, jestli hledaný bod leží na stejné straně všech hran (tedy přímek určených koncovými body hran), nebo neleží.

Pokud bod leží na stejné straně všech hran, nachází se uvnitř mnohoúhelníku. Pokud se ale vůči jen jediné hraně nachází na jiné straně než vůči ostatním, leží bod vně mnohoúhelníku. Nejlépe to vysvětlí obrázek:



Tomuto postupu se také někdy říká test polorovinami. Každá kontrola nám zabere konstantně mnoho času. Časová složitost tohoto postupu je tedy lineární vzhledem k počtu hran, neboli $\mathcal{O}(N)$.

Bod a nekonvexní mnohoúhelník

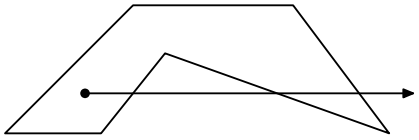
Pro nekonvexní útvary je již postup o něco těžší, protože jak si můžeme všimnout, postup s kontrolováním polohy bodu vůči všem hranám fungovat nebude.

Můžeme si ale na chvíli zahrát na Robina Hooda a ze zkoumaného bodu vystřelit šíp, respektive vést polopřímku. Pěkně se nám bude počítat, pokud polopřímku povedeme rovnoběžně s nějakou z os (třeba ve směru $(1, 0)$). Celé řešení pak spočívá v počítání, kolikrát polopřímka protne hranici mnohoúhelníku.

Můžeme si totiž všimnout, že finálně polopřímka skončí venku a nikdy více již do mnohoúhelníku nevstoupí. A pokaždé, když do mnohoúhelníku vstoupí, musí z něj zase někdy vystoupit. Pokud tedy bod leží venku, začali jsme polopřímku vést zvenku, a tedy bude počet protnutí sudý, pokud bod leží uvnitř, tak bude počet protnutí hranice lichý.

Jediné, na co je potřeba dát pozor, je situace, kdy polopřímka povede přesně skrz nějaký vrchol. V takovém případě se musíme podívat na opačné krajní body hran, které se v tomto vrcholu stýkají. Pokud se obě nachází ve stejné polorovině určené polopřímkou, jen jsme se vrcholu dotkli, ale neprošli jsme skrz (a tedy nepočítáme žádný průsečík). Pokud se ale krajní body hran nachází v opačných polorovinách, znamená to, že jsme ve vrcholu hranici proťali a musíme započítat jeden průsečík.

Jako cvičení na rozmyšlenou necháme situaci, kdy se druhý krajní bod jedné z hran nachází na polopřímce.

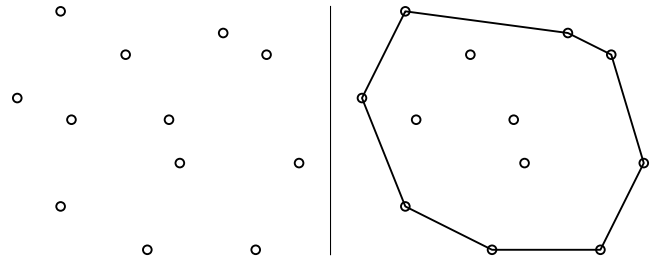


Opět musíme zkontrolovat polopřímku vůči všem hranám, takže časová složitost je znovu $\mathcal{O}(N)$ (i když s o něco vyšší konstantou, protože spočítání průsečíku je více početních operací než jeden test polorovinou).

Konvexní obal a zametání roviny

Podíváme se na jeden z nejznámějších geometrických problémů, totiž hledání konvexního obalu množiny bodů v rovině. *Konvexní obal* je nejmenší konvexní mnohoúhelník, který obsahuje všechny zadané body. Můžeme si všimnout, že všechny vrcholy výsledného mnohoúhelníka musí být nějaké body ze zadané množiny, jinak bychom mohli mnohoúhelník ještě zmenšit (a nebyl by to konvexní obal).

Jako motivaci si představte třeba situaci, že máte sad ovocných stromů a chcete je oplotit co nejkratším plotem. Jak takový plot, nebo obecně obal, nalézt?



Vlevo neobalené body, vpravo obalené.

Ukážeme si postup, kterému se říká *zametání roviny.* Je to trik, který najde uplatnění u mnoha různých geometrických problémů a vyplatí se ho umět.

Základní myšlenka spočívá v tom, že nějakou přímkou, říkejme jí *zametací přímka,* přejedeme přes celou rovinu (od minus nekonečna do plus nekonečna, zleva doprava nebo shora dolů) a vždy, když zametací přímka protne nějaký pro nás zajímavý bod, zpracujeme příslušnou událost. *Událost* je něco významného, co souvisí s příslušným bodem (průsečík přímek, vrchol mnohoúhelníka apod.)

Ale jak jet přímkou postupně od minus nekonečna do plus nekonečna? To není vůbec nutné. Pohyb přímky můžeme začít v nějakém startovním bodě (většinou první událost v setříděné posloupnosti událostí) a ukončit ho po zpracování všech událostí. Navíc nebudeme přímkou pohybovat plynule, ale budeme jí vždy skákat z události na událost (protože mezi událostmi se nic zajímavého neděje).

Vraťme se k našemu problému s konvexním obalem. Jako události budeme brát všechny body, které dostaneme na vstupu. V tomto případě nám žádné nové události v průběhu výpočtu vznikat nebudou, takže frontu událostí můžeme implementovat jako lineární spojový seznam.

Na začátku si body setřídíme podle jejich x-ové souřadnice (zatím budeme pro jednoduchost předpokládat, že žádné dva body nemají stejnou x-ovou souřadnici), začneme je zametací přímkou postupně procházet zleva doprava a budeme si udržovat konvexní obal bodů, které jsme už zpracovali.

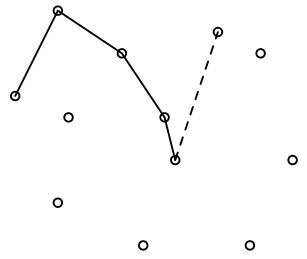
V průběhu výpočtu si budeme konstruovat horní a dolní *obálku.* Obě obálky budou určitě začínat v nejlevějším a končit v nejpravějším bodě (jednoduchým pozorováním lze nahlédnout, že tyto body do obalu určitě patří). A jak už název napovídá, horní obálka půjde vrchem a bude se zatáčet stále doprava, a dolní obálka naopak půjde spodem a bude se stále zatáčet doleva.

Můžeme se pro zjednodušení dohodnout, že nejlevější i nejpravější bod patří do obou obálek. Když pak horní a dolní obálku spojíme, dostaneme konvexní obal.

Horní (respektive dolní) obálku si budeme udržovat jako lineární seznam vrcholů.

Teď si ukážeme, jak bude probíhat jeden krok zpracování. Výpočet se bude provádět samostatně pro horní a dolní obálku, my si ho ukážeme jen pro horní (pro dolní je až na zrcadlení stejný).

Uvažujme, že už máme nějakou část horní obálky, skočili jsme zametací přímkou na další bod a ten teď chceme přidat. Podíváme se na poslední bod v horní obálce a zkontrolujeme úhel poslední hrany v obálce a úsečky mezi posledním bodem obalu a novým bodem.



K tomu můžeme využít například test polorovinami z úvodu kuchařky (pokud nový bod leží vůči poslední hraně horní obálky napravo, je vnitřní úhel konvexní, pokud nalevo, je úhel konkávní). Jestliže se horní obálka zatáčí doprava, máme vyhráno, přidáme nový bod do obálky a můžeme se posunout na další bod. Zajímavější je ale situace, kdy se nám obálka stočí doleva a vznikne konkávní úhel.

Pokud se podíváme na obrázek výše, jasně vidíme, že je potřeba dosavadní poslední bod obálky odebrat a zkusit spojit nově přidávaný bod s předposledním. Odstraníme tedy poslední bod obálky a budeme test opakovat s předposledním bodem.

Takto budeme pokračovat (a případně vyhazovat další body), buď než bude úhel hran konvexní, nebo dokud nám v obálce nezůstane pouze jeden bod (počáteční). Pak nový bod přidáme do obálky a pokračujeme s dalším.

Výše popsaný postup je nejvýhodnější provádět najednou pro obě dvě obálky. Tedy každý bod se pokusíme připojit k horní i dolní obálce a podle toho obě obálky příslušně upravíme.

Proč tento postup funguje? Postupně projdeme všechny body a každý z nich se alespoň na chvíli stane posledním bodem obálky. Při změně obálky se obsažená plocha v konvexním obalu vždy pouze zvětší a žádný bod nám tedy nemůže zůstat mimo konvexní obal.

Ještě jsme zapomněli na případ, kdy úhel není ani konvexní, ani konkávní. V takovém případě se rozhodneme, jestli budeme vrchol tohoto úhlu započítávat mezi vrcholy konvexního obalu. Obvykle se takový vrchol z konvexního obalu vyhazuje, ale nakonec vždycky záleží, k čemu ten konvexní obal vlastně potřebujeme.

Skončíme, až zametací přímkou skočíme na poslední bod a zpracujeme ho. V tomto bodě se nám obálky spojí a dostaneme celý konvexní obal. Teď ale přichází otázka, kolik času nám tento postup zabere?

Může se zdát, že hodně, protože při vyhazování bodů z obálky můžeme postupně vyhodit skoro všechny body. Označme si velikost zadané množiny (počet bodů na vstupu programu) N. Musíme si uvědomit, že každý bod do obálky přidáme pouze jednou a vyhodíme ho také maximálně jednou, tedy časová složitost je lineární k velikosti množiny, čili $\mathcal{O}(N)$, v případě, že již máme setříděný vstup. Pokud ne, musíme ještě přičíst čas potřebný k setřídění bodů, tedy $\mathcal{O}(N \log N)$ při použití nějakého rychlého třídicího algoritmu.[2]

Nakonec ještě zbývá dořešit více bodů se stejnou x-ovou souřadnicí. Pokud to nejsou krajní body, tak nám to v postupu nevadí. Menším problémem je, když to jsou počáteční, nebo koncové body. Problém ale snadno vyřešíme tím, když body seřadíme lexikograficky, tedy nejdříve podle x, a pokud je stejné, pak podle y. To nám jednoznačně určí pořadí bodů a počáteční i koncový bod.

Také si to můžeme představit tak, že rovinu nepatrně natočíme. Tím se určitě konvexní obal (až na natočení) nezmění, nikde nebudou dva body nad sebou a z pohledu algoritmu je to vlastně totéž, jako bychom prošli body v lexikografickém pořadí.

Hledání průsečíků úseček

Nakonec si ukážeme ještě jeden typický zametací problém, který principu zametání využívá o trochu více než konvexní obal. Představte si, že máte v rovině N úseček a chcete najít všechny jejich průsečíky.

Hledáme samozřejmě co nejrychlejší algoritmus vzhledem k N a počtu průsečíků P.

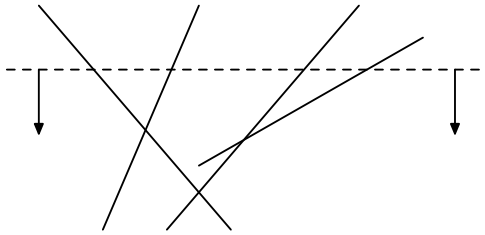
[2] `http://ksp.mff.cuni.cz/viz/kucharky/trideni`

Bystří si již jistě spočítali, že průsečíků může být v extrémním případě až N^2, a tedy nic rychlejšího než zkontrolovat každou úsečku se všemi dalšími v tomto případě není.

Ale takové případy se moc často nestávají, spíše naopak. Uvažujme tedy, že průsečíků je řádově tolik, kolik je úseček a v tom případě je výše popsaný algoritmus již pomalý.

Předpokládejme pro zjednodušení, že v žádném bodě se neprotínají tři a více úseček, žádné dvě úsečky nemají více než jeden společný bod (neleží přes sebe) a žádná úsečka není ani přesně svislá, ani přesně vodorovná. Vyřešení takovýchto případů spočívá v snadných úpravách uvedeného řešení.

Použijeme opět zametací přímku (pro lepší představu teď jdoucí shora dolů, obecně ale nemá směr zametání význam), kterou budeme skákat přes události, a na ní si budeme udržovat aktuální stav. Nazvěme ji třeba *průřezem.* Jak už název napovídá, bude udržovat pořadí úseček, které aktuálně protínají zametací přímku. Jelikož se průřez bude po každé události měnit, budeme pro něj potřebovat šikovnou datovou strukturu. Ale na to se podíváme až potom, co si rozebereme události, ať víme, co od průřezu budeme chtít.



Stejně jako v minulém případě budou mezi událostmi všechny body na vstupu (tedy počáteční i koncové body úseček), vyskytnou se tam ale i další. Pojďme si tedy trochu lépe rozebrat události a akce, které se při nich mají stát:

- *Začátek úsečky:* Přidáme úsečku na správné místo do průřezu, spočítáme případné průsečíky s okolními úsečkami a přidáme je do seznamu událostí.
- *Konec úsečky:* Smažeme úsečku z průřezu, a jelikož se nám dvě okolní úsečky dostanou smazáním této k sobě, musíme ještě spočítat jejich případný průsečík a přidat ho do seznamu událostí.
- *Průsečík:* Započítáme a zapíšeme si průsečík úseček, prohodíme pořadí těchto dvou úseček na průřezu, a jelikož se nám k sobě na průřezu dostaly nové úsečky, musíme spočítat, jestli se někde protínají, a případně průsečíky přidat do seznamu událostí.

Spočítání průsečíků úseček je jednoduchá analytická geometrie. Nejdříve porovnáme jejich směrnice. Pokud jdou od sebe, nemusíme se o nic starat, pokud jdou k sobě, spočítáme, ve kterém bodě se protnou. A když máme tento bod, jenom ověříme, jestli leží na obou úsečkách (neboli že úsečky nekončí ještě před spočítaným průsečíkem).

Když se podíváme na požadavky, hodilo by se nám umět v průřezu rychle vyhledávat, přidávat a mazat, k čemuž nám nejlépe poslouží vyhledávací strom. Ale co za informace si budeme o úsečkách ve vrcholech stromu pamatovat? Jejich aktuální x-ovou pozici (tedy přesněji x-ovou souřadnici bodu této úsečky na úrovni zametací přímky)? Tu bychom museli po každé události u všech úseček přepočítat, budeme na to tedy muset jít chytřeji.

Ve vrcholech stromu si budeme ukládat pouze nějaký rovnicový tvar úsečky (například její obecnou rovnici, nebo směrnici a bod) a vždy, když budeme vyhledávat ve stromu, tak si na základě aktuální y-ové pozice zametací přímky spočítáme v konstantním čase aktuální x-ovou pozici úsečky (jednoduchým doplněním do obecné rovnice) a podle toho se budeme ve vyhledávacím stromu pohybovat.

Máme tedy datovou strukturu pro průřez, ale jak dlouho budou trvat operace s ní? Jelikož v každou chvíli bude ve vyhledávacím stromu maximálně N vrcholů (tedy maximálně tolik, kolik je úseček), budou všechny operace se stromem trvat $\mathcal{O}(\log N)$.

Do seznamu událostí budeme potřebovat také přidávat prvky, takže tentokrát se nám mnohem více hodí použití nějaké haldy. Opět si můžeme uvědomit, že v haldě bude najednou pouze $\mathcal{O}(N)$ prvků (za každou úsečku její začátek a konec a průsečíky úseček vedle sebe na průřezu, tedy maximálně $N-1$ průsečíků), takže operace v ní bude trvat $\mathcal{O}(\log N)$.

Když už máme vybudované datové struktury, podívejme se na to, jak algoritmus poběží. Na začátku přidáme do průřezu první úsečku a do seznamu událostí všechny začátky i konce úseček. Pak již jen postupujeme po událostech, každou událost zpracujeme podle postupu výše a skončíme ve chvíli, kdy nám dojdou všechny události.

Algoritmus funguje správně, jelikož postupně projde přes všechny průsečíky (když jedna úsečka protíná více dalších, tak postupným prohazováním v průřezu se dostanou všechny tyto dvojice vedle sebe a všechny průsečíky přidáme do událostí) a žádný průsečík neprojdeme vícekrát.

Zpracování jakékoliv události nás stojí konstantní množství operací s datovými strukturami, a protože každá z těchto operací stojí maximálně $\mathcal{O}(\log N)$, tak nás zpracování jedné události stojí $\mathcal{O}(\log N)$. Počet událostí je $2N + P$ kde N je počet úseček a P počet průsečíků na výstupu, tedy celková časová složitost je $\mathcal{O}((N+P)\log N)$. Pro pořádek ještě uveďme paměťovou složitost, které je díky použitým datovým strukturám $\mathcal{O}(N)$.

Můžeme si všimnout, že pokud by průsečíků bylo řádově N^2, tak jsme si vlastně pohoršili. Předpokládali jsme ale situaci, kdy je průsečíků řádově stejně jako úseček. V tomto případě je náš algoritmus výrazně rychlejší.

Závěr

Prošli jsme si základní geometrické algoritmy pro rovinné problémy a ukázali jejich základní myšlenky. Různou aplikací a kombinací těchto postupů můžeme řešit většinu lehčích geometrických problémů v rovině, které potkáme.

Jen jako ochutnávku si ještě uvedeme například *Voroného diagramy,* což je rozklad roviny na oblasti, které jsou vždy nejblíž danému bodu (motivací může být například přiřazení obcí na mapě k nejbližšímu krajskému městu). Při jejich konstrukci se také uplatní zametání roviny, ovšem tentokrát již ne přímkou, ale pomocí zametacích parabol.

A jak jsme si uvedli na začátku, mnohé z uvedených postupů lze zobecnit z roviny i do prostoru, ale o tom někdy jindy. Pokud máte zájem o další informace o geometrických algoritmech, tak vás můžeme odkázat na *Průvodce labyrintem algoritmů* stáhnutelný ze stránek Martina Mareše.[3]

Pokud stále nemáte geometrie dost, můžete si ještě zkusit vyhledat pojmy *kombinatorická a výpočetní geometrie.* Dostanete se tak ke spoustě dalších zajímavých materiálů.

Jirka Setnička

[3] `http://pruvodce.ucw.cz/`