Korespondenční Seminář z Programování

38. ročník **KSP** **Listopad 2025**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám druhé číslo hlavní kategorie 38. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Dále na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál**, jehož díly mohou vycházet samostatně.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.

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 (hlavní 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 21. prosince 2025 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; Experimentální (neobvyklá) úloha

Odměna série: **Odznáček do profilu na webu** si vysloužíí ten, kdo **získá alespoň 1 bod z každé úlohy druhého dílu seriálu**.

Druhá série třicátého osmého ročníku KSP

38-2-1 Všechny cesty vedou přes Řím — 12 bodů

Uvažme svět, kde celou společnost řídí skrytá organizace bohatých slonů (hippoteticky). Tahle organizace se doslechla, že se slézají hroši v Brně na Velké Programovací Konferenci. Toho by se jistě dalo využít pro zbohatnutí! Bohatý slon = šťastný slon!

Víme, že nejvíce hrochů jede do Brna z Kolína (tam se asi potkali, kdo ví...). Chceme najít město, kam umístit závoru se slonem, který bude vybírat peníze za vstup dovnitř. Jenže hroši jsou mazaní jako liška, rychlí jako jaguár a chytří jako Medvěd, tedy se vyhnou podporování sloní organizace jakkoliv to jen půjde.

Proto musíme závoru umístit do města, přes které hroši určitě projedou, ať chtějí, nebo ne. Jaká města připadají v úvahu? Jinými slovy, vaším úkolem je najít všechna města, která jsou na *všech cestách* z Kolína do Brna, aby se jim nedalo vyhnout. Nezapomeňte svoje řešení pořádně zdůvodnit.

Mapa je zadaná jako orientovaný graf, kde města reprezentují vrcholy.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

38-2-2 Stop STOPkám — 11 bodů

Kevin jel autem do Brna se ubytovat do hotelu Hippo-Sleep na Velkou Programovací Konferenci, avšak před odjezdem zjistil, že si bohužel zapomněl prodloužit dálniční známku. Musel proto jet jenom po hlavních a vedlejších silnicích mimo dálnice. To je pro Kevina obzvlášť nepříjemné, protože musí často čekat na STOPce na křižovatkách, což opravdu nemá rád. Naštěstí má Kevin přístup k seznamu silnic v celém Česku a ví, že se v celé republice nachází N křižovatek, které jsou propojené M obousměrnými silnicemi, o každé z nich ještě zná její prioritu a délku.

Kdy musí Kevin čekat? Každá křižovatka má hlavní cestu, složenou z až dvou silnic, které do ni vedou s nejvyšší prioritou. Zbytek silnic potom tvoří cesty vedlejší. Kevin potom nemusí čekat na STOPce právě tehdy, když přijede na křižovatku po hlavní cestě. Naopak, přijede-li z vedlejší silnice, čekat musí.

Kevinův dům se nachází na křižovatce s číslem 1, a jeho cíl má číslo N. U startu Kevin musí čekat vždy a u cíle Kevin čekat nemusí. Pomozte Kevinovi s navigací, a najděte mu takovou trasu, u které co nejméněkrát zastaví na STOPce. Pokud je takových tras více, tak vypište tu s nejkratší délkou. Pokud je jich i tak více, vypište libovolnou z nich.

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

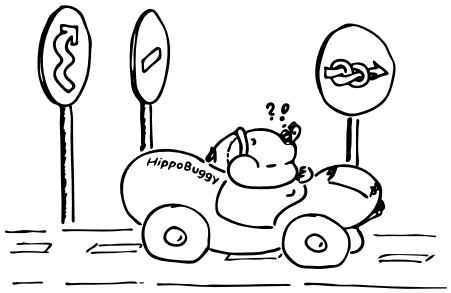
Formát vstupu: Na prvním řádku dostanete N, počet křižovatek, a M, počet silnic v republice. Na každém z dalších M řádků se nachází popis silnice:

- Číslo p, priorita silnice. Žádné dvě silnice nemají stejnou prioritu.
- Čísla a, b, indexy koncových křižovatek silnice.
- Číslo ℓ, délka silnice mezi a a b.

Formát výstupu: Na jediný řádek výstupu vypište indexy křižovatek v pořadí od začátku do konce, tak jak se vyskytují ve vaší trase.

Ukázkový vstup:	*Ukázkový výstup:*

`4 3`
`10 1 2 1`
`9 1 3 1`
`80 1 4 1`
	`1 4`



38-2-3 Opisování — 11 bodů

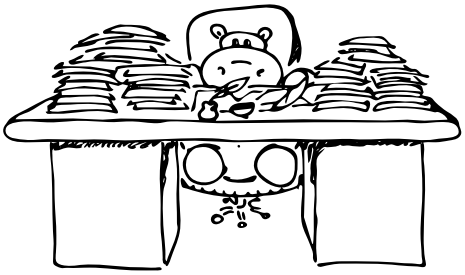
Petr má poslední dobou plné ruce práce. Spousta lidí si chce narychlo zařídit řidičský průkaz, aby mohli jet na Velkou Programovací Konferenci v dalekém Brně, takže má jeho autoškola rekordní počet N studentů. Všichni jeho studenti ale posledních pár týdnu veškerý volný čas věnovali přípravě na konferenci, takže se na rychle se blížící závěrečný písemný test skoro vůbec neučili. Rozhodli se tedy, že budou podvádět.

Petr je toho názoru, že by školní prostředí mělo být co nejpodobnější praxi. Proto své studenty nechává při zkouškách sedět na silnici. Všichni studenti sedí v jedné řadě a dívají se stejným směrem – po směru silnice. Každý student (až na prvního) vidí do testu studenta sedícího před ním. Pokud na nějakou otázku student nezná odpověď, tak ji opíše od studenta sedícího před ním, pokud ten student odpověď zná, nebo ji od někoho také opsal.

Závěrečný test se skládá z M otázek. Petr je přísný učitel, nechá proto projít jen studenty, které zodpoví správně všechny otázky. Také chce své studenty za jejich opisování potrestat, rád by tedy určil zasedací pořádek tak, aby jich prošlo co nejméně. Využil tedy svou schopnost čtení mysli, aby zjistil, kteří studenti znají odpověď na kterou otázku. Pomůžete mu?

Na vstupu dostanete čísla N a M a tabulku jedniček a nul o velikosti $N \times M$, která pro každou dvojici studenta a otázky obsahuje informaci o tom, jestli na danou otázku daný student zná odpověď. Vaším úkolem je vypsat takové pořadí studentů, aby po opisovaní mělo plný počet bodů co nejméně studentů. Petr si chce být jistý, že studentů projde opravdu co nejméně, je tedy **nezbytné**, abyste o vašem algoritmu důsledně dokázali, že opravdu vždy vyprodukuje optimální řešení.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`



38-2-4 Zákeřná parkovací místa — 11 bodů

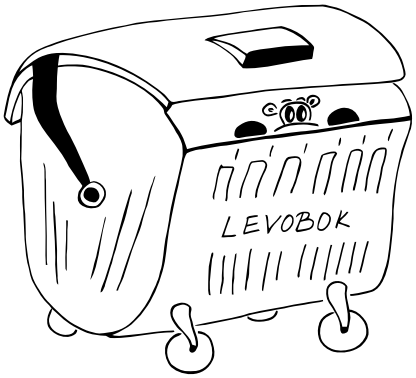
Sára jela do Brna na Velkou Programovací Konferenci. Našla si tam hezké ubytování v hotelu HippoSleep, ale bohužel bylo parkoviště plné, takže musela zaparkovat na vedlejším parkovišti nedaleko hotelu. Tato parkoviště bylo poměrně zvláštní. Všechna parkovací místa byla v jedné řadě za sebou a každé místo bylo označené nějakým přirozeným číslem (včetně nuly). Dokonce bylo i možné, že některá místa měla stejná čísla!

Sára byla po své dlouhé cestě tak unavená, že si těchto zvláštností všimla až další den ráno, když si uvědomila, že si nepamatuje, kde zaparkovala své auto. Navíc jsou teď všechna místa obsazená auty, které vypadají totožně jako její a také všechna překrývají čísla na parkovacích místech. Sára chvíli panikařila, ale pak si vzpomněla, že číslo označující její parkovacího místa je větší nebo rovno než čísla na obou sousedních parkovacích místech. Pro parkovací místa na okrajích předpokládáme, že vedle nich se nachází pomyslné místo označené nulou.

Aby našla své parkovací místo, Sára prochází řadou parkovacích míst a kouká se pod auta. Přitom ale nechce pod auta koukat příliš dlouho, protože má na konferenci spoustu zajímavých přednášek. Abyste Sáře pomohli, musíte splnit dvě úlohy:

- Rozmyslete si a ukažte, zda Sářino parkovací místo vždy v dané řadě parkovacích míst existuje. Pokud neexistuje, ukažte, kdy k tomu dojde.
- Najděte Sáře její parkovací místo v asymptoticky co nejlepší složitosti, vzhledem k počtu pohledů pod auta. Složitost očekáváme, že bude lepší než lineární. Pokud je potřeba, ošetřete případ, kdy takové místo neexistuje.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`



38-2-S Hledá se pivot — 15 bodů

V dnešním dílu seriálu o pravděpodobnostních algoritmech se podíváme na zoubek algoritmům Quicksort a Quickselect založeným na rozdělování prvků podle pivota.

Hledá se k-tý nejmenší prvek

Představme si, že jsme dostali nějakou dlouhou posloupnost čísel $x_0, \ldots, x_{n-1}$ a chceme zjistit, které číslo je k-té nejmenší (počítáno od 0). Půjdeme na to následovně: Vybereme si z posloupnosti jeden prvek, kterému budeme říkat *pivot.* Rozdělíme posloupnosti na tři části:

- L (levé) jsou prvky menší než pivot,
- S (střední) jsou prvky rovné pivotovi,
- P (pravé) jsou prvky větší než pivot.

Když si posloupnost představíme seřazenou, budou všechny prvky části L (nějak zamíchané) ležet před všemi prvky části S, a ty zase před všemi prvky P (nějak zamíchanými). Proto:

- Je-li $k < |L|$, musí se hledaný prvek nacházet v L a být tam zase k-tý nejmenší.
- Je-li $|L| \leq k < |L| + |S|$, musí hledaný prvek ležet v S, a tedy být roven pivotovi.
- Zbývá případ $k > |L| + |S|$. Tehdy hledaný prvek leží v P a je tam $(k - |L| - |S|)$-tý nejmenší.

V každém případě jsme buďto rovnou odpověděli, nebo problém převedli na jiný, menší problém stejného druhu. To vede na jednoduchý rekurzivní algoritmus, kterému se říká *Quickselect:*

```
def quickselect(x: list[int], k: int) -> int:
    # Jako pivota zvolíme prostřední z prvků
    pivot = x[len(x) // 2]
    leve = [a for a in x if a < pivot]
    stred = [a for a in x if a == pivot]
    prave = [a for a in x if a > pivot]
    if k < len(leve):
        return quickselect(leve, k)
    elif k < len(leve) + len(stred):
        return pivot
    else:
        j = k - len(leve) - len(stred)
        return quickselect(prave, j)
```

Zbývá spočítat časovou složitost. Samotné rozdělování prvků na tři části v jednom zavolání funkce trvá $\Theta(n)$. Další volání jsou rychlejší, protože vstup se postupně zmenšuje. To, jak rychle se zmenšuje, je záleží na volbě pivota.

- Nejhorší případ nastane například tehdy, když jako pivota pokaždé vybereme *minimum.* Pak bude část L prázdná, část S bude samotný pivot a zbývajících $n-1$ prvků bude v P. Hledáme-li největší prvek ($k = n-1$), pokračujeme vždy v P. V jednom kroku rekurze jsme tedy strávili $\mathcal{O}(n)$ času a vstup se zmenšil pouze o 1. Celkový čas tedy činí $\mathcal{O}(n) + \mathcal{O}(n-1) + \ldots + \mathcal{O}(1) = \mathcal{O}(n^2)$.
- Velmi dobře to naopak dopadne, pokud jako pivota budeme volit *medián,* tedy prvek, který by v setříděném pořadí ležel uprostřed. (Nenechme se zviklat tím, že zatím nevíme, jak medián rychle najít.) Tehdy jak L, tak P budou mít méně než $n/2$ prvků. V každém kroku rekurze tedy vstup zmenšíme alespoň dvakrát. Celková časová složitost bude činit $\mathcal{O}(n)+\mathcal{O}(n/2)+\mathcal{O}(n/4)+\ldots+\mathcal{O}(1) = \mathcal{O}(n \cdot (1 + 1/2 + 1/4 + \ldots))$.

  Výraz v závorkách je geometrická řada a s těmi se hodí umět zacházet obecně. Součet geometrické řady $q^0 + q^1 + \ldots + q^{m-1}$ je pro $q \neq 1$ roven $(q^m - 1)/(q - 1)$. To je pro $0 < q < 1$ menší než $1/(1-q)$. V našem výpočtu složitosti je tedy $q = 1/2$, takže součet v závorce je menší než 2. Celý Quickselect proto běží v lineárním čase.
- Co kdybychom nevybírali přesně medián, ale *pseudomedián?* Tak budeme říkat prvkům, které by v setříděném pořadí ležely v prostředních dvou čtvrtinách. Prvky v první čtvrtině jsou pak menší nebo rovné pivotovi (takže jsou v L nebo S, tedy nikoliv v P), prvky v poslední čtvrtině jsou naopak větší nebo rovné pivotovi (jsou v S nebo P, tedy nikoliv v L). Jak L, tak P tedy obsahuje nejvýše $(3/4) \cdot n$ prvků.

  Složitost celého algoritmu proto činí $\mathcal{O}(n \cdot (1 + (3/4) + (3/4)^2 + (3/4)^3 + \ldots))$. Zde máme geometrickou řadu s $q = 3/4$, tedy se součtem nejvýše 4. I pro pseudomediány proto vyjde lineární složitost.

Radost nám kazí, že nevíme, jak medián, nebo aspoň pseudomedián najít. K tomu nám pomůže randomizace… (Dodejme, že existuje i mnohem složitější deterministický algoritmus, který najdete v Průvodci labyrintem algoritmů, oddíl 10.8.)

Náhodný výběr pivota

Nejprve uvažujme následující algoritmus na výběr pseudomediánu: Vybereme libovolný prvek. Spočítáme, kolik prvků je větších a kolik menších. Z toho poznáme, jestli jsme našli pseudomedián. A pokud nenašli, spustíme algoritmus znovu.

Jedno spuštění algoritmu trvá $\mathcal{O}(n)$. Jelikož aspoň polovina prvků jsou pseudomediány (může jich být víc, pokud se hodnoty opakují), algoritmus uspěje s pravděpodobností aspoň 1/2. Podle lemmatu o džbánu (to znáte z pravděpodobnostní kuchařky) tedy průměrný počet pokusů do prvního úspěchu je nejvýše 2. Hledání pseudomediánu proto trvá *průměrně* lineární čas. Zdůrazněme, že se jedná o průměr vzhledem k náhodným číslům generovaným našim algoritmem, nikoliv vzhledem k možným vstupům.

Daleko jednodušší ovšem je vzít původní Quickselect a prostě v něm jako pivota vybrat rovnoměrně náhodný prvek posloupnosti (rovnoměrně náhodný znamená, že všechny prvky mají stejnou pravděpodobnost, že budou vybrány). Ukážeme, že i toto vede na průměrně lineární složitost.

Výpočet algoritmu rozdělíme na *epochy.* Epochu ukončíme, kdykoliv si jako pivota vybereme pseudomedián. Když epocha začne se vstupem velikosti m, každé rekurzivní volání během epochy trvá $\mathcal{O}(m)$. Stejným použitím lemmatu

o džbánu jako v minulém odstavci vyjde, že průměrný počet rekurzivních volání na epochu je maximálně 2. Průměrná složitost celé jedné epochy tedy činí $\mathcal{O}(m)$. Současně ale za celou epochu klesne velikost posloupnosti na maximálně $(3/4) \cdot m$. Součet průměrných složitostí epoch tedy činí $\mathcal{O}(n \cdot (1 + (3/4) + (3/4)^2 + \ldots)) = \mathcal{O}(n)$. (Tady jsme použili linearitu střední hodnoty, kterou známe z kuchařky.)

Rozdělujeme na místě

Náš algoritmus má lineární časovou složitost (alespoň v průměru), ale reálná implementace nebude moc rychlá. Tráví totiž spoustu času vytvářením nových polí a jejich rušením. Lepší je udržovat celou dobu prvky v jednom poli a při rozdělování na skupiny je jenom přehazovat uvnitř pole.

Práci si trochu zjednodušíme pozorováním, že u prvků rovných pivotovi si můžeme vybrat, zda je zařadíme do L, S, nebo P. Správnost algoritmu tím nenarušíme. Jen si musíme dávat pozor na to, abychom v degenerovaných případech (pivot je minimum nebo maximum, všechny prvky jsou stejné apod.) vstup zmenšili o aspoň jeden prvek.

```
def rozdel(x: list[int], l: int, p: int) \
        -> tuple[int, int]:
    pivot = x[(l + p) // 2]
    while l <= p:
        while x[l] < pivot:
            l += 1
        while x[p] > pivot:
            p -= 1
        if l <= p:
            x[l], x[p] = x[p], x[l]
            l += 1
            p -= 1
    return l, p
```

Tato funkce dostane pole x, jehož prvky na pozicích ℓ až p má rozdělit na skupiny. Prvky v lineárním čase přehází, a pak vrátí dvojici indexů ℓ' a p', přičemž platí:

- $p' < \ell'$
- Na pozicích $\ell, \ldots, p'$ jsou prvky z L a část S.
- Na pozicích $\ell', \ldots, p$ jsou prvky z P a část S.
- Mezi p' a ℓ' může být jeden prvek, který patří do S.
- Žádná z těchto tří částí tedy neobsahuje všechny prvky.

Rozmysleme si, proč jsou tyto vlastnosti splněné. Označíme ℓ a p původní okraje úseku, ℓ' a p' aktuální hodnoty proměnných `l` a `p`.

1. Dokud hlavní cyklus neskončí, mezi ℓ a ℓ' vždy leží prvky menší rovné pivotovi, mezi p' a p prvky větší rovné pivotovi.
2. V prvním průchodu vnějšího cyklu leží mezi ℓ' a p' aspoň jeden výskyt pivota. Posouvání ℓ' doprava se tedy zarazí nejpozději o něj, stejně tak posouvání p' doleva. V dalších průchodech cyklu se posun ℓ' zarazí nejpozději o prvky větší než pivot napravo od p' a posun p' o prvky nalevo od ℓ'.
3. Po vnitřních cyklech je tedy $x[\ell'] \geq \mathit{pivot}$ a $x[p'] \leq \mathit{pivot}$. Po prohození $x[\ell']$ a $x[p']$ tedy můžeme posunout ℓ' doprava a p' doleva. Pak se buďto indexy překřížily a jsme hotovi, nebo pokračujeme, přičemž stále platí vlastnost 1.
4. Ovšem pozor, mohlo se stát, že $\ell' = p'$. Tehdy musí prvek $x[\ell'] = x[p']$ být současně menší roven i větší roven pivotovi. Takže je to pivot a algoritmus se může zastavit s jedním pivotem mezi p' a ℓ'.

S rozdělováním na místě můžeme naprogramovat Quickselect na místě, dokonce bez rekurze:

```
def quickselect(x: list[int], k: int) -> int:
    levy = 0
    pravy = len(x) - 1
    while levy < pravy:
        l, p = rozdel(x, levy, pravy)
        if levy + k <= p:
            pravy = p
        elif levy + k < l:
            return x[l - 1]
        else:
            k -= l - levy
            levy = l
    return x[levy]
```

Zde `levy` a `pravy` jsou okraje úseku v poli `x`, kde hledáme `k`-tý nejmenší prvek. Časová složitost bude nadále $\mathcal{O}(n)$, ovšem s mnohem lepšími konstantami.

Quicksort

Na podobném principu je postavený také třídicí algoritmus *Quicksort.* Ten nejprve rozdělí vstup podle pivota na levou, střední a pravou část. Pak levou a pravou rekurzivně setřídí. A nakonec všechny tři části slepí za sebe:

```
def quicksort(x: list[int]) -> list[int]:
    if len(x) <= 1:
        return x

    pivot = x[len(x) // 2]
    leve = [a for a in x if a < pivot]
    stred = [a for a in x if a == pivot]
    prave = [a for a in x if a > pivot]
    return (quicksort(leve)
            + stred
            + quicksort(prave))
```

Pojďme stanovit složitost. Opět nejprve nejhorší případ, kdy si jako pivota vybereme (řekněme) minimum. Tehdy bude levá část prázdná, střední jednoprvková a zbylých $n-1$ prvků zbude napravo. Rozdělit problém na podproblémy trvá $\Theta(n)$ času, složit z výsledků podproblémů výsledek celého problému také $\Theta(n)$ času. Celkem tedy trávíme čas $\Theta(n) + \Theta(n-1) + \ldots + \Theta(1)$, což se sečte na $\Theta(n^2)$.

Nejlepší případ zase nastane, když se nám bude dařit jako pivota vybírat medián. Tehdy problém setřídění n prvků převedeme na dva problémy setřídění nejvýše $n/2$ prvků. Časovou složitost tedy můžeme popsat rovnicí

$$T(n) = 2T(n/2) + \Theta(n).$$

Zkušený informatik zajásá, že takhle se přeci chová Mergesort, takže musí vyjít $T(n) = \Theta(n \log n)$.

Pokud se nechceme odvolávat na jiné algoritmu, můžeme složitost odvodit jednoduchou úvahou o stromu rekurze. Představme si strom, který má na nulté hladině kořen s původním problémem velikosti n. Pod něj na první hladinu zavěsíme 2 podproblémy velikosti $n/2$, které si kořen zavolal. Každý z nich si zavolal další 2 podproblémy velikosti $n/4$, takže druhá hladina obsahuje celkem 4 podproblémy velikosti $n/4$. Obecně i-tá hladina obsahuje 2^i problémů velikosti $n/2^i$. V každém z nich trávíme čas $\Theta(n/2^i)$, v součtu přes celou hladinu tedy $\Theta(n)$.

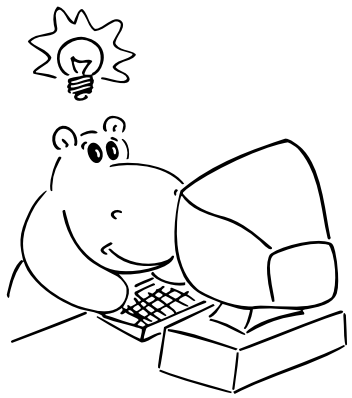
To vynásobíme počtem hladin. Kolik jich strom má? Velikost podproblémů se zmenšuje exponenciálně, takže podproblémy na $(\log n)$-té hladině už budou nejvýše jednoprvkové, čímž se rekurze zastaví. Celková složitost tedy vyjde $\Theta(n \log n)$.

Nakonec uvážíme variantu Quicksortu, v níž pivota volíme rovnoměrně náhodně. Spočítáme její průměrnou časovou složitost. Všimneme si, že složitost každého podproblému je lineární v počtu prvků, které se ho účastní. Proto stačí pro každý prvek spočítat, kolika podproblémů se účastní, a toto pak sečíst přes všechny prvky. Díky linearitě střední hodnoty tato úvaha platí i pro průměrnou složitost a průměrný počet podproblémů.

Vyberme si tedy nějaký prvek a sledujme ve stromu rekurze, jakých podproblémů se účastní. Kořene určitě ano. Pokud nebyl vybrán jako pivot, padne do právě jednoho podproblému. Pak zase do jednoho z jeho podproblémů atd. Všechny tyto podproblémy tvoří cestu ve stromu z kořene dolů, která končí buď listem, nebo vrcholem, kde se náš prvek stal pivotem.

Tuto cestu si opět rozdělíme na epochy tak, že epochu ukončíme podproblémem, kde jsme jako pivota vybrali pseudomedián. Stejně jako u Quickselectu je epocha tvořena průměrně dvěma podproblémy. Mezi dvěma po sobě jdoucími epochami se velikost podproblému zmenší aspoň $(3/4)$-krát. Velikosti tedy klesají exponenciálně, pročež počet epoch musí být logaritmický.

Jeden prvek se tedy účastní průměrně $\mathcal{O}(\log n)$ podproblémů. To platí pro všechny prvek stejně, takže vynásobením počtem prvků získáme, že průměrná časová složitost Quicksortu činí $\mathcal{O}(n \log n)$.



Triky v implementaci Quicksortu

Quicksort je velice populární algoritmus. Najdeme ho ve standardní knihovně mnoha programovacích jazyků, protože je v praxi velice rychlý. Ve srovnání s jinými algoritmy, jako je třeba Mergesort nebo Heapsort, sice dosahuje složitosti $\mathcal{O}(n \log n)$ jenom v průměru, ale zato se chová velmi přátelsky k moderním procesorům a jejich keším.

Vyzkoušejme si několik triků, které se v implementacích Quicksortu hodí. Především můžeme opět využít rozdělování podle pivota na místě:

```
def quicksort(x: list[int]) -> None:
    def qs(levy: int, pravy: int) -> None:
        if levy < pravy:
            l, p = rozdel(x, levy, pravy)
            qs(levy, p)
            qs(l, pravy)

    qs(0, len(x) - 1)
```

Také si můžeme všimnout, že pro malé vstupy je rozdělování a spojování daleko dražší než vlastní třídění, tedy porovnávání a prohazování prvků. Na takových vstupech budou kvadratické třídicí algoritmy (jako třeba Insertsort) rychlejší než Quicksort. Upravíme tedy Quicksort, aby v *malých podproblémech* (menších než nějaká hranice) ukončil rekurzi a přepnul na kvadratický algoritmus.

Nakonec se můžeme zbavit rekurze. Pořídíme si vlastní zásobník (třeba v poli), na který si budeme odkládat úseky, které ještě zbývá dotřídit. Hlavní smyčka algoritmu si vždy vezme jeden úsek ze zásobníku a rozdělí ho na podúseky. Malé podúseky rovnou setřídí kvadratickým algoritmem. Velké podúseky odloží na zásobník, aby se jimi zabývala v nějaké další iteraci. Dokonce může jeden z podúseků odložit na zásobník, zatímco druhým bude rovnou pokračovat.

Úkol 1 [1b]: Dokažte, že odložíme-li na zásobník *větší* z obou úseků, zatímco tím menším budeme pokračovat, bude na zásobníku v každém okamžiku ležet nejvýše logaritmický počet úseků.

Vaše implementace

Už víme, že Quicksort běží v průměrném čase $\mathcal{O}(n \log n)$. Jak to ale vypovídá o jeho rychlosti na skutečném počítači? Pojďme s tím experimentovat. K experimentům se bude hodit implementace Quicksortu v nějakém kompilovaném programovací jazyku, ideálně C, C++ nebo Rustu. V nouzi poslouží i Python, byť měření bude dost zatížené režií jazyka. Kdybyste si nebyli jistí vhodností nějakého jazyka, ozvěte se nám emailem nebo na Discordu.

Úkol 2 [3b]: Naprogramujte Quicksort tak, aby třídil na místě a používal všechny triky z této kapitoly. Jako pivota vybírejte medián z prvniho, prostředního a posledního prvku úseku, Rekurzi zastavujte na 10 prvcích.

Úkol 3 [2b]: Naprogramujte generátor testovacích dat, který vyrobí vstupy následujících druhů složené z 32-bitových celých čísel:

- rovnoměrně náhodná čísla,
- rostoucí posloupnost,
- co nejhorší případ pro váš výběr pivota mediánem ze tří prvků,
- Fibonacciho čísla modulo 2^{32} (posloupnost začíná 0, 1 a každé další číslo je součtem dvou předchozích modulo 2^{32}).

Od každého druhu vyrobte vstupy velikostí přibližně 1.4^k prvků pro k od 15 do 54.

Experimenty

Napište program, který bude vaši implementaci Quicksortu postupně spouštět na testovacích datech jednotlivých veli-

kostí a pokaždé změří, jak dlouho třídění trvalo (nepočítejte do toho přípravu vstupu, jen samotné třídění).

Pokus by měl být opakovatelný. Pokud generujete náhodná čísla, nastavujte konkrétní random seed, aby se program při každém spuštění choval stejně.

Abyste omezili šum v naměřených datech (způsobený například tím, že operační systém vašeho počítače se občas zamyslí nad něčím jiným), měřte pro každou velikost vstupu pětkrát (pokaždé s jiným rnadom seedem). Nejmenší a největší hodnotu zahoďte, ze zbylých tří spočítejte průměr.

Pokud je pro velké vstupy program příliš pomalý, tak tyto vstupy z experimentu vyřaďte.

Naměřené hodnoty zakreslete do grafu. Na vodorovné ose budiž velikost vstupu, na svislé čas. (Na kreslení grafů se může hodit třeba pythoní knihovna Matplotlib.) Nezapomeňte uvést, na jakém hardwaru jste měřili (typ procesoru, taktovací frekvenci, množství paměti a operační systém).

Úkol 4 [2b]: Vyzkoušejte několik různých hranic pro ukončení rekurze. Měřte na náhodných vstupech. Nakreslete do jednoho grafu křivky pro jednotlivé hranice. Na základě měření si vyberte nejlepší hodnotu hranice a tu pak používejte v dalších úkolech.

Úkol 5 [2b]: Vyzkoušejte vliv volby pivota. Srovnejte:

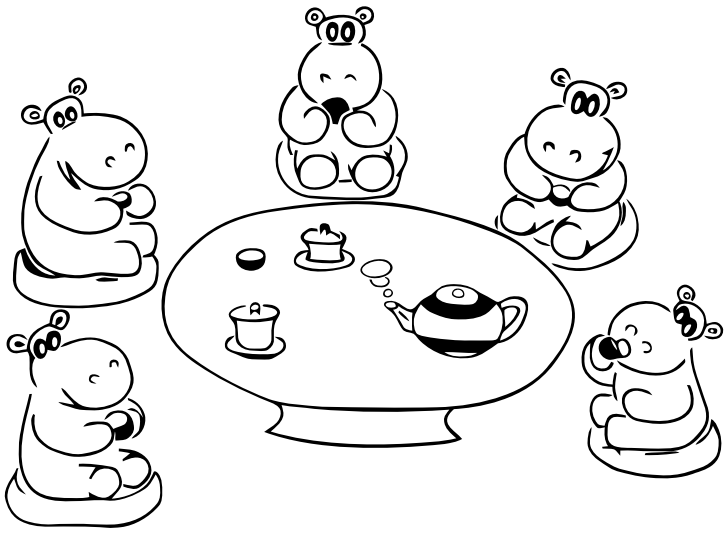
- prostřední prvek úseku,
- medián z prvního, prostředního a posledního prvku,
- rovnoměrně náhodně vybraný prvek úseku,
- volitelně: prostřední prvek, ovšem než začneme třídit, náhodně prvky zamícháme.

Pro každý z těchto způsobů vytvořte jeden graf a do něj zakreslete křivky naměřených časů pro jednotlivé druhy vstupů (rostoucí/náhodný/...). Rozmyslete si a popište, jaká volba pivota vám přijde nejlepší.

Úkol 6 [2b]: U každé verze Quicksortu riskujeme, že budeme mít smůlu (nebo dostaneme škodolibý vstup) a program poběží kvadraticky pomalu. Tomu se dá bránit: budeme sledovat hloubku zanoření (v rekurzivním algoritmu by to byla hloubka rekurze, v nerekurzivním si můžeme u intervalů udržovat, kolikerým rozdělením vstupu vznikly). Pokud hloubka překročí $\alpha \cdot \log n$ pro vhodně zvolenou konstantu α, úsek dotřídíme nějakým algoritmem se složitostí $\mathcal{O}(n \log n)$ v nejhorším případě. Tomuto algoritmu se říká *Hybridsort.* Zkuste ho implementovat a experimentálně stanovit optimální hodnotu α. Vyzkoušejte všechny druhy vstupů. Z vysledků měření opět vytvořte graf.

Úkol 7 [3b]: Implementujte *dvoupivotový Quicksort.* Ten v každém kroku vybere dva pivoty a podle nich rozdělí vstupu na tři části (menší než první pivot, mezi pivoty, větší než druhý pivot). Dvoupivotové rozdělování se ještě dá hezky udělat na místě. Ubezpečte se, že vaše rozdělování funguje i v případech, kdy si jako pivota vyberete minimum nebo maximum, případně kdy jsou všechny prvky v úseku stejné. Měřením porovnejte rychlost klasického Quicksortu s dvoupivotovým. Opět vyzkoušejte všechny druhy vstupů a výsledky zakreslete do grafu.

Odevzdejte ZIP, v němž bude zdrojový kód všech programů a PDF obsahující: popis všech programů, grafy s naměřenými hodnotami a interpretaci výsledků měření.



matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*