Rozděl a panuj

Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme řešit opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císaři: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídicí kuchařce“. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídicí kuchařce (hlavně proto, že se nám z ní pak budou snadno odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
{ budeme třídit takováto pole }
type Pole=array[1..MaxN] of Integer;

{ přerovnávací procedura pro úsek a[l..r] }
function prerovnej(a: Pole; l, r: integer): integer;
var i, j, x, q: integer;
begin
  { pivotem se stane poslední prvek úseku }
  x:=a[r];                { hodnota pivota }
  i:=l-1;                 { a[i] bude vždy poslední <= pivotovi }

  { samotné přerovnávání }
  for j:=l to r-1 do
    if a[j]<=x then       { právě probíraný prvek }
    begin                 { menší/rovný hodnotě pivota }
      i:=i+1;             { pak zvyš ukazatel }
      q:=a[j];            { a proveď přerovnání prvku }
      a[j]:=a[i];
      a[i]:=q;
    end;

  { nakonec přesuneme pivota za poslední <= }
  q:=a[r];
```

```
  a[r]:=a[i+1];
  a[i+1]:=q;
  prerovnej:=i+1;          { vrátíme novou pozici pivota }
end;

{ hlavní třídicí procedura, třídí a[l..r] }
procedure QuickSort(a: Pole; l, r: integer);
var m: integer;
begin
  if l<r then begin        { máme ještě co dělat? }
    m:=prerovnej(l,r);     { m pozice pivota }
    QuickSort(l,m-1);      { setřiď prvky napravo }
    QuickSort(m+1,r);      { setřiď prvky nalevo }
  end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které to nastane pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-2$ a 1 (pivot se nepočítá), načež pokračujeme s úsekem délky $N-2$, ten rozdělíme na $N-4$ a 1, atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N + (N-2) + (N-4) + \ldots + 1) = \mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N \log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2 \pm 1$); přerovnávání v obou částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty).

V hloubce $\lceil \log_2 N \rceil$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\lceil \log_2 N \rceil$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: Zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* Kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti „v prostřední polovině“ (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost

$\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4) \cdot N$, takže na k-té hladině budou úseky délek nejvýše $(3/4)^k \cdot N$, čili hladin bude maximálně $\log_{3/4} N = \mathcal{O}(\log N)$. Místo $1/4$ by fungovala i libovolná jiná konstanta mezi nulou a jedničkou, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.

- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. (Také se tak často QS implementuje.)
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností $1/2$ to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do seriálu o pravděpodobnostních algoritmech v 16. ročníku KSP). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián dostáváme pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídicí kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká *QuickSelect*). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších.

Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
function kty(var a: Pole; l, r, k: integer): integer;
var x, z: integer;
begin
  x:=prerovnej(a,l,r);   { x je pozice pivota }
  z:=x-l+1;              { pozice pivota vzhledem k [l..r] }
  if k=z then
    kty:=a[x]            { k-tý nejmenší je pivot }
  else if k<z then
    kty:=kty(a,l,x-1,k)  { k-tý nejmenší je nalevo }
  else
    kty:=kty(a,x+1,r,k-z);  { napravo }
end;
```

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku tříděním. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot, podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.
- Podobně jako u předchozího algoritmu, pokud je $k = z + 1$, pak je pivot m k-tým nejmenším prvkem posloupnosti. Není-li tomu tak a $k < z + 1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy, v opačném případě, kdy $k > z + 1$, vyhledáme $(k - z + 1)$-tého nejmenšího mezi posledními $n - z - 1$ prvky.

Řečeno s panem Pascalem:

```
{ potřebujeme přerovnávací funkci, která
  dostane hodnotu pivota jako parametr }
function prerovnejp(var a: Pole; l, r, m: integer): integer;
var q, p: integer;
begin
  { nalezneme pozici pivota }
  p:=l;
  while a[p]<>m do
    p:=p+1;
  { pivota prohodíme s posledním prvkem }
  q:=a[p]; a[p]:=a[r]; a[r]:=q;
  { a zavoláme původní přerovnávací fci }
  prerovnejp := prerovnej(a,l,r);
end;

{ hledání k-tého nejmenšího prvku z a[l..r] }
function kth(var a: Pole; l, r, k: integer): integer;
var medp:Pole;            { pole pro mediány pětic }
    i, j, q, x, pocet, m, z: integer;
begin
  pocet:=r-l+1;           { s kolika prvky pracujeme }
  if pocet<=1 then        { pouze jeden prvek? }
    kth:=a[l]             { výsledek nemůže být jiný }
  else if pocet<6 then begin { méně než 6 prvků }
    QuickSort(a,l,r);
    kth:=a[l+k-1];
    end
  else begin              { mnoho prvků, jde to tuhého }
    { rozdělíme prvky do pětic }
    q:=1;                 { zatím máme jednu pětici }
    i:=l;                 { levý okraj první pětice }
    j:=i+4;               { pravý okraj první pětice }
    while j<=r do begin   { procházíme celé pětice }
      QuickSort(a,i,j);
      medp[q]:=a[i+2];    { medián pětice }
      q:=q+1;             { zvyš počet pětic }
      i:=i+5;             { nastav levý okraj pětice }
      j:=j+5;             { nastav pravý okraj pětice }
    end;
    { případnou neúplnou pětici můžeme ignorovat }

    m:=kth(medp,1,q-1,q div 2); { najdeme medián mediánů pětic }

    x:=prerovnejp(a,l,r,m); { přerovnej a zjisti, kde skončil pivot }
    z:=x-l+1;             { pozice vzhledem k [l..r] }
```

```
    if k=z then
      kth:=m              { k-tý nejmenší je pivot }
    else if k<z then
      kth:=kth(a,l,x-1,k) { k-tý nejmenší nalevo }
    else
      kth:=kth(a,x+1,r,k-z); { napravo }
  end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze má slíbenou lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za pivotem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí např. pro $d = 10c$, takže opravdu $t(N) = \mathcal{O}(N)$. (Lépe už to nepůjde, jelikož na každé číslo se musíme podívat alespoň jednou.)

Cvičení

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny“ na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – teď zvolíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo můžeme zapsat jako $10^N A+B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N(AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B)\cdot(C+D) = AC+AD+BC+BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD+BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) = d$, kde d je nějaká konstanta. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem $3/2$, čili $((3/2)^k - 1)/(3/2 - 1) = \mathcal{O}((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k = 2^{k \log_2 3} = 2^{\log_2 N \cdot \log_2 3} = (2^{\log_2 N})^{\log_2 3} = N^{\log_2 3} \approx N^{1.58}$. Konstanta d se nám „schová do $\mathcal{O}$-čka“, takže algoritmus má časovou složitost přibližně $\mathcal{O}(N^{1.58})$. Existují i rychlejší algoritmy se složitostí až $\mathcal{O}(N)$, ale ty jsou mnohem ďábelštější a pro malá N se to sotva vyplatí.

Program si pro dnešek odpustíme, šetříme naše lesy (tedy alespoň trošku).

David Matoušek a Martin Mareš

Úloha 19-2-5: Hluboký les

Pomozte nám v hledání nejhlubšího lesa. Ten se nachází na místě, kde jsou dva stromy, které jsou u sebe nejblíže ze všech v lese. Váš program dostane na vstupu číslo N a dále N řádků s reálnými souřadnicemi jednotlivých stromů. V případě, že je dvojic nejbližších stromů víc, stačí vypsat libovolnou z nich.

Příklad: Pro $N = 4$ a stromy

1	3
2	1
3	1
4	3

by měl program vypsat: Stromy 2 a 3 jsou si k sobě nejblíže.

Úloha 19-5-5: Počet inverzí

Je dána posloupnost celých čísel $P_1, P_2, \ldots, P_N$. Čísla P_i a P_j jsou v *inverzi*, pokud $i < j$ a zároveň $P_i > P_j$. Inverze je tedy porucha ve vzestupném uspořádání posloupnosti. Vašim úkolem je zjistit, kolik inverzí posloupnost obsahuje.

Na prvním řádku vstupu je číslo N, na druhém řádku následuje N celých čísel v desítkovém zápisu oddělených mezerami. Počet inverzí vypište na standardní výstup. Čísel v posloupnosti je maximálně 100 000.

Příklad: Pro vstup:

```
5
4 5 3 1 2
```

vypište na standardní výstup 8.