INFORMATIKA

Jedničkový obdélník
(Úlohy z MO kategorie P, 48. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

Při našem putování po zajímavých úlohách Matematické olympiády kategorie P (programování) se tentokrát vydáme do velmi daleké historie a ukážeme si jednu soutěžní úlohu z krajského kola 38. ročníku MO. Tento ročník olympiády probíhal ve školním roce 1988/89, tedy ještě za socialistického Československa a navíc v době, kdy kategorie P teprve čerstvě vznikala. V té době u nás nebyly k dispozici osobní počítače, takže zadání soutěžních úloh se nezachovalo v elektronické podobě a nenajdete ho proto ani v archívu úloh na webu olympiády – tento archív začíná až 42. ročníkem MO ve školním roce 1992/93. Museli jsme proto sáhnout do starého archívu „papírového“. Zadání úlohy uvádíme bez úprav v původní historické podobě.

* * * * * * * * * * * * *

Je dáno dvojrozměrné pole A (matice) velikosti $n \times m$, jehož prvky jsou pouze čísla 0 nebo 1. Navrhněte algoritmus, který v daném poli A nalezne maximální „obdélník“ obsahující samé jedničky (maximální ve smyslu „obsahující co nejvíc jedniček“). Výsledkem práce algoritmu bude čtveřice čísel i, j, k, l takových, že $A_{i,j}$ je prvek v levém horním rohu a $A_{k,l}$ prvek v pravém dolním rohu nalezeného maximálního obdélníku.

Naším úkolem tedy je nalézt v zadané matici nul a jedniček co největší souvislý obdélníkový výřez (podmatici), který bude tvořen pouze jedničkami. Takovému obdélníkovému výřezu budeme nadále zkráceně říkat *jedničkový obdélník*. Je to úloha poměrně snadná a nezáludná, která

nevyžaduje znalost žádných pokročilých algoritmů nebo datových struktur. Přesto ale i při řešení takto jednoduché úlohy si můžeme ukázat celou řadu užitečných postupů a programovacích technik, které nám umožní snížit časovou složitost výsledného programu. Při návrhu efektivního algoritmu bývá často výhodné využít vhodný předvýpočet – tedy nepracovat přímo se zadanými vstupními daty, ale nejprve si je nějak vhodně upravit, aby pak následný vlastní výpočet probíhal rychleji. Ukážeme si postupně několik různých způsobů, jaký předvýpočet můžeme zvolit.

V zadání úlohy není uvedeno, zda máme počítat s číslováním řádků a sloupců matice A od 0 nebo od 1. Pro postup řešení úlohy to není vůbec podstatné, naše volba se projeví pouze ve výsledku (posunutí výsledných hodnot indexů i, j, k, l o 1). V matematice jsme zvyklí na číslování řádků matice od 1 do n a sloupců od 1 do m, proto se toho budeme držet i v řešení naší úlohy. V současných programovacích jazycích rodiny C (C, C++, Java, C#) nebo ve stále užívanějším Pythonu se ovšem všechna pole indexují automaticky od 0. To nám nebude vadit, v programu si jednoduše vytvoříme pole s $n+1$ řádky a $m+1$ sloupci a obsah zkoumané matice A do něj uložíme až od řádku 1 a od sloupce 1. V řádku a ve sloupci s číslem 0 budou v našem pracovním poli uloženy samé nuly, což se nám při řešení bude dokonce občas velmi hodit.

Původní zadání úlohy se také nijak nezmiňuje o situaci, kdy úloha nemá žádné řešení. To nastane v případě, že zadaná matice A obsahuje samé nuly. Správný program nesmí v takové situaci zhavarovat, musí ji spolehlivě detekovat a vypsat vhodný výsledek. V našich programových ukázkách budeme v takovém případě místo souřadnic levého horního a pravého dolního rohu nalezeného jedničkového obdélníku vypisovat čtyři nuly.

Po úvodním rozboru situace se můžeme pustit do řešení úlohy. Začneme jako obvykle tím nejjednodušším postupem – mechanickým vyzkoušením všech možností. Budeme postupně zkoumat všechny možné obdélníkové výřezy matice A, zda jsou tvořeny samými jedničkami. Kdykoliv najdeme obdélník tvořený samými jedničkami, porovnáme jeho velikost s velikostí dosud největšího nalezeného jedničkového obdélníku a pokud je větší, uložíme si jeho velikost a souřadnice. Každý obdélníkový výřez je určen polohou svého levého horního rohu a pravého dolního rohu. Levý horní roh můžeme zvolit $n \cdot m$ způsoby, pro každou takovou volbu zvolíme pravý dolní roh $\mathrm{O}(n \cdot m)$ způsoby, takže celkem projdeme $\mathrm{O}(n^2 \cdot m^2)$ podmatic. Každý zvolený obdélník projdeme a zkontrolujeme v čase $\mathrm{O}(n \cdot m)$. Celková časová složitost tohoto primitivního řešení je tudíž $\mathrm{O}(n^3 \cdot m^3)$.

```
n, m = (int(_) for _ in input("Rozměry matice: ").split())
print("Matice 0/1 po řádcích:")
a = [[0] * (m+1)]
for r in range(n):
    a.append([0] + [int(_) for _ in input().split()])

def test(a, i, j, k, l):
    # kontrola obdélníka a[i][j] - a[k][l] na samé 1
    for x in range(i, k+1):
        for y in range(j, l+1):
            if a[x][y] == 0:
                return False
    return True

def obdelnik(a):
    # zkoušení všech obdélníkových výřezů matice a
    max_p = 0
    max_i, max_j, max_k, max_l = 0, 0, 0, 0
    for i in range(1, n+1):
       for j in range(1, m+1):
          for k in range(i, n+1):
             for l in range(j, m+1):
                 plocha = (k-i+1)*(l-j+1)
                 if plocha > max_p and test(a, i, j, k, l):
                     max_p = plocha
                     max_i, max_j, max_k, max_l = i, j, k, l
    return max_i, max_j, max_k, max_l

print("Maximální jedničkový obdélník:", *obdelnik(a))
```

Výrazná neefektivita uvedeného řešení je způsobena zejména tím, že mnohé obdélníkové výřezy se různě překrývají. My ovšem každý z nich zvlášť procházíme a testujeme jeho prvky, zda se rovnají nule, i když jsme stejnou kontrolu již provedli v rámci testování jiné podmatice. Pokusíme se tedy odstranit tento problém s opakovaným testováním prvků zadané matice.

Jednou možností zrychlení výpočtu je použít tzv. prefixové součty. S prefixovými součty se v programování setkáváme zejména u úloh pracujících s číselnými posloupnostmi, kde i-tý prefixový součet posloupnosti předsta-

vuje součet prvních i členů posloupnosti. V úlohách s maticemi obvykle využíváme dvourozměrnou variantu prefixových součtů. K původní matici A si vytvoříme druhou matici B stejné velikosti, v níž prvek $B_{x,y}$ bude udávat součet všech čísel ležících v matici A v obdélníku s levým horním rohem na pozici $1, 1$ a pravým dolním rohem na pozici x, y. Ukážeme si to na jednoduchém příkladu:

matice A

1	1	0	0	1	1
0	1	1	0	1	1
1	1	1	1	1	0
1	0	0	1	1	1
1	1	1	1	1	0

matice B

1	2	2	2	3	4
1	3	4	4	6	8
2	5	7	8	11	13
3	6	8	10	14	17
4	8	11	14	19	22

Nejprve popíšeme, jak lze popsanou matici dvourozměrných prefixových součtů B efektivně sestrojit. Všechny prvky v řádku 0 a ve sloupci 0 budou z technických důvodů opět nulové. Prvky matice B budeme počítat postupně po řádcích od 1 do n, každý řádek procházíme vždy zleva doprava po sloupcích od 1 do m. Hodnotu $B_{x,y}$ určíme podle vztahu

$$B_{x,y} = A_{x,y} + B_{x-1,y} + B_{x,y-1} - B_{x-1,y-1}.$$

Vzhledem ke zvolenému pořadí výpočtu se tak v každém okamžiku odkazujeme pouze na hodnoty matice B, které jsme již spočítali dříve. Pokud některý z uvedených prvků matice B neexistuje, nahradíme jeho hodnotu ve vzorci nulou. V programu k tomu využijeme skutečnost, že prvky matice ležící v řádku 0 a ve sloupci 0 mají hodnotu 0. Nakreslete si sami schéma matice a vyznačte si v něm oblasti, s nimiž ve vzorci pracujeme. Uvedený vztah je založen na tzv. principu inkluze a exkluze, který známe z diskrétní matematiky. Po sečtení

$$A_{x,y} + B_{x-1,y} + B_{x,y-1}$$

jsme oblast $B_{x-1,y-1}$ započítali do součtu $B_{x,y}$ dvakrát, a proto ji musíme zase jednou odečíst. Každý prvek matice B jsme spočítali v konstantním čase, takže sestrojení celé matice B má asymptotickou časovou složitost $\mathrm{O}(n \cdot m)$.

Budeme-li mít k dispozici popsanou matici B, s její pomocí dokážeme zkontrolovat každý obdélníkový výřez matice A v konstantním čase. Využijeme při tom skutečnost, že součet všech prvků obdélníkového výřezu

matice s hodnotami 0 nebo 1 je roven počtu jedniček obsažených v tomto výřezu. Obdélník proto obsahuje samé jedničky právě tehdy, když je součet všech jeho prvků roven jeho velikosti. Uvažujme obdélníkový výřez s levým horním rohem $A_{i,j}$ a pravým dolním rohem $A_{k,l}$. Součet všech jeho prvků snadno spočítáme jako

$$B_{k,l} - B_{k,j-1} - B_{i-1,l} + B_{i-1,j-1}.$$

Opět jsme zde využili již zmíněný princip inkluze a exkluze a dvakrát odečtenou oblast $B_{i-1,j-1}$ jsme zase zpátky jednou přičetli. Obdélník je tvořen samými jedničkami, pokud je součet jeho prvků roven jeho velikosti, tedy hodnotě $(k-i+1)\cdot(l-j+1)$.

Stejně jako v prvním řešení postupně projdeme všech $\mathrm{O}(n^2 \cdot m^2)$ obdélníkových výřezů zadané matice A. Každý z nich tentokrát ale zkontrolujeme v konstantním čase, takže celková časová složitost takto vylepšeného řešení je $\mathrm{O}(n^2 \cdot m^2)$. Do výsledné asymptotické časové složitosti je třeba započítat ještě složitost výpočtu pomocné matice B, ale ta je pouze $\mathrm{O}(n \cdot m)$ a výslednou složitost tedy nijak neovlivní.

Ukázkový program je přímou implementací popsaného postupu. Pro zjednodušení kódu a paměťovou úsporu zde dvourozměrné prefixové součty prvků zadané matice A ukládáme přímo do matice A, protože původní zadané hodnoty matice A nebudeme již v dalším výpočtu potřebovat.

```
n, m = (int(_) for _ in input("Rozměry matice: ").split())
print("Matice 0/1 po řádcích:")
a = [[0] * (m+1)]
for r in range(n):
    a.append([0] + [int(_) for _ in input().split()])

def soucet(a):
    # dvourozměrné prefixové součty matice a
    for x in range(1, n+1):
        for y in range(1, m+1):
            a[x][y] += a[x-1][y]+a[x][y-1]-a[x-1][y-1]

def obdelnik(a):
    # zkoušení všech obdélníkových výřezů matice a
    max_p = 0
    max_i, max_j, max_k, max_l = 0, 0, 0, 0
```

```
    for i in range(1, n+1):
        for j in range(1, m+1):
            for k in range(i, n+1):
                for l in range(j, m+1):
                    plocha = (k-i+1) * (l-j+1)
                    pocet1 = a[k][l] - a[k][j-1] - a[i-1][l] \
                             + a[i-1][j-1]
                    if plocha > max_p and pocet1 == plocha:
                        max_p = plocha
                        max_i, max_j, max_k, max_l = i, j, k, l
    return max_i, max_j, max_k, max_l

soucet(a)
print("Maximální jedničkový obdélník:", *obdelnik(a))
```

Nyní si předvedeme jiný postup řešení s odlišným předvýpočtem, který nás dovede k ještě rychlejšímu programu. V úvodním pomocném výpočtu si spočítáme délky souvislých sloupců jedniček v matici A a uložíme si je do matice B. Jestliže $A_{x,y} = 0$, položíme také $B_{x,y} = 0$. V opačném případě bude hodnota $B_{x,y} = t$ znamenat, že prvek $A_{x,y}$ a dalších přesně $t - 1$ prvků matice A pod ním mají hodnotu 1. Uvedený předvýpočet můžeme demonstrovat na následujícím příkladu:

matice A

1	1	0	0	1	1
0	1	1	0	1	1
1	1	1	1	1	0
1	0	0	1	1	1
1	1	1	1	1	0

matice B

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

Matici B sestrojíme po jednotlivých sloupcích. Každý sloupec matice A procházíme zdola nahoru, nula se do matice B jednoduše opíše, zatímco jednička se při přenosu do matice B zvýší o hodnotu, která leží v matici B bezprostředně pod ní. Každý prvek matice B takto spočítáme v konstantním čase, takže sestrojení celé matice B má asymptotickou časovou složitost $\mathrm{O}(n \cdot m)$.

Zbývá ukázat, jak nyní vyřešíme zadanou úlohu pomocí hodnot uložených v matici B. V matici B je nenulový prvek právě na těch místech,

kde je v matici A jednička. Chceme tedy projít všechny obdélníkové výřezy matice B tvořené pouze nenulovými prvky a vybrat z nich ten největší. Podobně jako v předchozích dvou řešeních budeme zkoumat všechny možné pozice levého horního rohu obdélníku, kterých je $n \cdot m$. Pro každý levý horní roh ale nebudeme zkoušet všechny možné pravé dolní rohy, nýbrž pravé horní rohy. Těch je jenom $\mathrm{O}(m)$, protože řádková souřadnice pravého horního rohu je již určena volbou levého horního rohu. Prvek $B_{i,j}$ může být levým horním rohem jedničkového obdélníku pouze tehdy, když $B_{i,j} > 0$. Prvek $B_{i,l}$ může být pravým horním rohem jedničkového obdélníku s levým horním rohem $B_{i,j}$, jsou-li kladná všechna čísla $B_{i,y}$ pro $y = j, j+1, \ldots, l$.

Velikost maximálního obdélníku, který je v původní matici A tvořen samými jedničkami a který má levý horní roh $A_{i,j}$ a pravý horní roh $A_{i,l}$, nyní již snadno určíme pomocí hodnot uložených v matici B. Tento obdélník má šířku $l - j + 1$ určenou polohou obou jeho horních rohů. Jeho výška je určena minimem z hodnot $B_{i,y}$ pro $y = j, j+1, \ldots, l$, neboť přesně tak byly hodnoty matice B definovány.

Již jsme ukázali, že počet způsobů volby obou horních roků jedničkového obdélníku je $\mathrm{O}(n \cdot m^2)$. Kdybychom pro každou takovou volbu hledali minimum z hodnot $B_{i,y}$ zvlášť, museli bychom k tomu provést až $\mathrm{O}(m)$ operací a celková časová složitost by se tím zhoršila na $\mathrm{O}(n \cdot m^3)$. Potřebný výpočet minima, neboli určování výšky maximálního jedničkového obdélníku, můžeme ovšem provádět průběžně zároveň s volbou indexu l. Tím dosáhneme výsledné asymptotické časové složitosti celého řešení $\mathrm{O}(n \cdot m^2)$.

V ukázkové programové implementaci právě popsaného řešení jsme opět provedli předvýpočet přímo v zadané matici A. Ušetříme tak dodatečnou paměť potřebnou na pomocnou druhou matici B. Původní hodnoty matice A nebudeme již v dalším výpočtu potřebovat a navíc by ani nebylo těžké obnovit je, kdybychom snad chtěli (stačilo by nahradit všechny nenulové prvky matice jedničkami).

```
n, m = (int(_) for _ in input("Rozměry matice: ").split())
print("Matice 0/1 po řádcích:")
a = [[0] * (m+1)]
for r in range(n):
    a.append([0] + [int(_) for _ in input().split()])

def soucet(a):
    # sloupcové počty jedniček pod prvkem v matici a
```

```
    for y in range(1, m+1):
        for x in range(n-1, 0, -1):
            if a[x][y] == 1:
                a[x][y] += a[x+1][y]

def obdelnik(a):
    # zkoušení všech obdélníkových výřezů matice a
    max_p = 0
    max_i, max_j, max_k, max_l = 0, 0, 0, 0
    for i in range(1, n+1):
        for j in range(1, m+1):
            vyska = a[i][j]
            for l in range(j, m+1):
                if a[i][l] == 0: break
                vyska = min(vyska, a[i][l])
                plocha = (l-j+1) * vyska
                if plocha > max_p:
                    max_p = plocha
                    k = i + vyska -1
                    max_i, max_j, max_k, max_l = i, j, k, l
    return max_i, max_j, max_k, max_l

soucet(a)
print("Maximální jedničkový obdélník:", *obdelnik(a))
```

Právě popsaný předvýpočet délek souvislých sloupců jedniček v matici A bychom mohli samozřejmě provést také symetricky na řádky matice. Pokud v celém řešení zaměníme řádky a sloupce, dostaneme analogický postup s asymptotickou časovou složitostí $\mathrm{O}(n^2 \cdot m)$. Z obou variant si můžeme vybrat tu časově výhodnější podle toho, zda je větší n nebo m, tj. zda má zadaná matice A více řádků nebo sloupců. Tímto doplněním získáme ještě o něco lepší řešení s časovou složitostí $\mathrm{O}(n \cdot m \cdot \min(n, m))$.

S postupným vylepšováním řešení ale stále ještě nejsme u konce. Možná vás překvapí, že naše úloha má ještě rychlejší řešení. K vyhledávání jedničkových obdélníků v matici ovšem musíme přistoupit trochu jiným způsobem – místo zkoušení jejich rohů budeme teď vyhledávat jejich strany.

Maximální obdélník tvořený samými jedničkami se vyznačuje tím, že každá jeho strana sousedí s nějakou nulou (nebo je na okraji celé matice). V opačném případě by totiž takový jedničkový obdélník nebyl maximální,

jeho stranu by bylo možné posunout směrem „ven“ a tím obdélník zvětšit. Chceme-li tedy nalézt levou stranu maximálního jedničkového obdélníku, stačí vyhledávat v matici A jedničky, které zleva sousedí s nulou.

Než si popíšeme celý algoritmus, připravíme si opět vhodný předvýpočet. Bude podobný jako v přechozím řešení, kde jsme si ke každé jedničce v matici A spočítali, kolik dalších jedniček leží bezprostředně pod ní. Tyto údaje jsme si uložili do matice B. Tentokrát použijeme úplně stejnou matici B a kromě ní si zavedeme ještě druhou pomocnou matici C, v níž si obdobným způsobem pro každou jedničku spočítáme, kolik jedniček leží ve sloupci matice A souvisle nad ní. Opět si to předvedeme na stejném konkrétním příkladu:

matice A						matice B						matice C					
1	1	0	0	1	1	1	3	0	0	5	2	1	1	0	0	1	1
0	1	1	0	1	1	0	2	2	0	4	1	0	2	1	0	2	2
1	1	1	1	1	0	3	1	1	3	3	0	1	3	2	1	3	0
1	0	0	1	1	1	2	0	0	2	2	1	2	0	0	2	4	1
1	1	1	1	1	0	1	1	1	1	1	0	3	1	1	3	5	0

Matici C sestrojíme po jednotlivých sloupcích analogicky, jako jsme to dělali v předchozím řešení s maticí B. Každý sloupec matice A procházíme shora dolů, nula se do matice C jednoduše opíše, zatímco jednička se při přenosu do matice C zvýší o hodnotu, která leží v matici C bezprostředně nad ní. Každý prvek matice C tak spočítáme v konstantním čase, takže sestrojení obou matic B a C má asymptotickou časovou složitost $\mathrm{O}(n \cdot m)$.

Zbývá popsat vlastní algoritmus řešení úlohy. Budeme procházet matici A postupně po řádcích zleva doprava a budeme hledat situaci, kde se nacházejí vedle sebe na řádku 0 a hned za ní 1. Na levém okraji maximální jedničkové podmatice nutně musí ležet prvek $A_{i,j} = 1$ s takovouto vlastností. Budeme od něj postupovat po řádku matice směrem doprava, dokud nenarazíme na nulu nebo na pravý okraj matice. Pro každý prvek $A_{i,l} = 1$, přes který budeme takto procházet, určíme maximální jedničkový obdélník s levým okrajem ve sloupci j a pravým okrajem ve sloupci l. Výšku takového obdélníku směrem dolů i nahoru spočítáme pomocí hodnot připravených v maticích B a C. Při tomto výpočtu postupujeme obdobně jako v předchozím řešení.

Asymptotická časová složitost tohoto našeho řešení je překvapivě pouze $\mathrm{O}(n \cdot m)$. Již jsme ukázali, že počáteční výpočet pomocných matic B a C

má složitost $O(n \cdot m)$. Následné hledání maximální jedničkové podmatice je pak sice tvořeno třemi do sebe vnořenými cykly, ale oba vnitřní cykly společně provádějí pouze jeden průchod řádkem matice délky m, zatímco vnější cyklus prochází přes řádky matice a má tedy délku n. Když na řádku procházíme souvislý úsek sousedících jedniček, nemůžeme při tom minout žádnou jedničku sousedící zleva s nulou. Až po skončení tohoto úseku budeme na řádku hledat další jedničku následující po nule.

Závěrečná programová ukázka je přímou implementací posledního uvedeného algoritmu. Stejně jako v předchozím programu ani zde matici B ve skutečnosti nepotřebujeme a její hodnoty si můžeme uložit přímo do původní matice A. Pro údaje z druhé pomocné matice, kterou jsme v rozboru algoritmu označili jako C, již ale druhou datovou strukturu potřebovat budeme.

```
n, m = (int(_) for _ in input("Rozměry matice: ").split())
print("Matice 0/1 po řádcích:")
a = [[0] * (m+1)]
for r in range(n):
    a.append([0] + [int(_) for _ in input().split()])

def soucet_pod(a):
    # sloupcové počty jedniček pod prvkem v matici a
    for y in range(1, m+1):
        for x in range(n-1, 0, -1):
            if a[x][y] == 1:
                a[x][y] += a[x+1][y]

def soucet_nad(a):
    # sloupcové počty jedniček nad prvkem v matici a
    c = [[0] * (m+1)]
    for x in range(1, n+1):
        c.append([a[x][y] for y in range(m+1)])
    for y in range(1, m+1):
        for x in range(2, n+1):
            if c[x][y] == 1:
                c[x][y] += c[x-1][y]
    return c
```

```
def obdelnik(a, c):
    # zkoušení všech obdélníkových výřezů matice a
    max_p = 0
    max_i, max_j, max_k, max_l = 0, 0, 0, 0
    for i in range(1, n+1):
        j = 1
        while j <= m:
            if a[i][j-1] == 0 and a[i][j] > 0:
                vyska_pod = a[i][j]
                vyska_nad = c[i][j]
                l = j
                while l <= m and a[i][l] > 0:
                    vyska_pod = min(vyska_pod, a[i][l])
                    vyska_nad = min(vyska_nad, c[i][l])
                    plocha = (l-j+1) * \
                             (vyska_pod + vyska_nad - 1)
                    if plocha > max_p:
                        max_p = plocha
                        max_i = i - vyska_nad + 1
                        max_j = j
                        max_k = i + vyska_pod - 1
                        max_l = l
                    l += 1
                j = l
            j += 1
    return max_i, max_j, max_k, max_l

c = soucet_nad(a)
soucet_pod(a)
print("Maximální jedničkový obdélník:", *obdelnik(a, c))
```