INFORMATIKA

Veletrh dortů
(Úlohy z MO kategorie P, 44. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha

V Matematické olympiádě kategorie P (programování) se setkáváme s mnoha velmi zajímavými úlohami. Některé z nich jsou snadné, jiné naopak značně náročné. V dnešním pokračování dlouhodobého seriálu věnovaného vybraným problémům z MO kategorie P (programování) si ukážeme jednu z těch velmi snadných. Jedná se o teoretickou úlohu z domácího kola nedávno skončeného 70. ročníku MO (školní rok 2020/21). Patří do kategorie často zadávaných úloh typu „práce s posloupnostmi čísel“. Úlohám tohoto typu jsme se věnovali již několikrát, obvykle jsme ale k optimálnímu vyřešení problému potřebovali použít nějakou netriviální programátorskou techniku typu dynamického programování nebo třeba předvýpočet prefixových součtů posloupnosti. Jak sami uvidíte, v optimálním řešení dnešní úlohy nic takového potřebovat nebudeme, vystačíme jenom s jednoduchou logickou úvahou. Ukážeme si ovšem, že i takto jednoduchý problém lze řešit různými způsoby a s různou časovou složitostí.

Nejprve se jako obvykle seznámíme s přesným zadáním úlohy. Některé jeho formulace jsme pro potřeby článku maličko upravili.

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

Radek se letos jako každý rok vypravil na veletrh dortů. Ten tvoří řada n stánků očíslovaných od 1 do n. Pravý důvod Radkovy přítomnosti je, že za malý poplatek a_i Kč může Radek v i-tém stánku ochutnat představovaný dort. Radek by samozřejmě rád ochutnal co nejvíce dortů, jenže má to dva háčky. Za prvé, Radek disponuje pouze omezenou částkou k Kč. Za druhé moc dobře ví, že jakmile poprvé ochutná dort v nějakém stánku i,

nedá mu to a ochutná poté dort ve vedlejším stánku $i + 1$, následně ve stánku $i + 2$ a tak dále, než mu dojdou peníze nebo než dojde na konec řady (tehdy skončí, i kdyby mu ještě peníze zbývaly). Ve kterém stánku má Radek začít, aby ochutnal co nejvíce dortů?

Soutěžní úloha

Je zadána posloupnost n kladných celých čísel $a_1, a_2, \ldots, a_n$, kde a_i je cena i-tého dortu, a kladné celé číslo k, tedy počet korun, které má Radek k dispozici. Zjistěte, kolik nejvíce dortů může Radek sníst a u kterého stánku má začít, aby toho dosáhl. Vaším úkolem je vypsat dvě čísla d a p taková, že platí následující podmínky:

1. $1 \leq d \leq n$ a $1 \leq p \leq n - d + 1$.
2. $a_p + a_{p+1} + \ldots + a_{p+d-1} \leq k$, tedy lze ochutnat d dortů, začne-li Radek na pozici p.
3. pro všechna $1 \leq q \leq n - d$ platí $a_q + a_{q+1} + \ldots + a_{q+d} > k$, tedy více než d dortů ochutnat nelze.

Pokud existuje více vyhovujících pozic p, můžete vypsat kteroukoliv z nich.

Formát vstupu

První řádek obsahuje dvě kladná celá čísla n a k. Druhý řádek obsahuje n kladných celých čísel $a_1, a_2, \ldots, a_n$.

Formát výstupu

Výstup je tvořen dvěma čísly d a p, kde d je největší počet dortů, které Radek může ochutnat, a p je číslo stánku, u něhož má začít, aby d dortů snědl.

Příklady

Vstup:	Výstup:
7 6 2 1 2 3 2 2 1	3 2

Radek může ochutnat tři dorty, pokud začne ochutnávkou druhého dortu: za dorty zaplatí $1 + 2 + 3 = 6$ Kč. Jiné správné řešení by bylo začít na první nebo páté pozici.

Vstup:	Výstup:
3 10 100 100 100	0 1

V tomto případě nemůže Radek ochutnat ani jeden dort, jeho prázdná řada dortů pak může začínat na pozici 1, 2, nebo 3.

Bodování

Plných 10 bodů získá řešení s optimální časovou složitostí.

Za řešení, které na běžném počítači vyřeší úlohu pro $n = 10^6$ během několika sekund, dostanete až 7 bodů.

Za řešení dostatečně efektivní pro $n = 5000$ dostanete až 4 body.

Za jakékoliv funkční řešení dostanete až 2 body.

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

Zadání soutěžních úloh MO kategorie P bývá obvykle zabalené do nějaké „pohádky“, jejímž cílem je buď ukázat praktickou využitelnost řešeného problému, nebo jenom jednoduše řešitele trochu pobavit. V tomto konkrétním případě se zjevně jedná o ten druhý případ. Podobnou formu ovšem mají i úlohy zadávané na mezinárodních olympiádách středoškoláků v informatice a dokonce i na obdobných soutěžích univerzitních, takže pro naše soutěžící je to i užitečná příprava na to, co je čeká v případě postupu do mezinárodních soutěží. Řešení každé úlohy proto začíná tím, že soutěžící musí pozorně přečíst celý text zadání, oddělit z něj to nepodstatné a z dlouhého „pohádkového“ povídání vyloupnout jádro řešeného problému. Úlohu je pak zpravidla možné přetransformovat do mnohem kratší podoby a tu vyjádřit strohou matematickou formulací. Náš dnešní problém takto můžeme shrnout jedinou větou: v zadané posloupnosti n kladných celých čísel nalezněte nejdelší souvislý úsek se součtem rovným nejvýše k.

Úloha má primitivní řešení spočívající v postupném otestování všech souvislých úseků dané posloupnosti. Přitom si průběžně pamatujeme nejdelší úsek, jehož součet nepřesáhl k; ukládáme si jeho délku d a počáteční index p. V posloupnosti délky n existuje $\mathrm{O}(n^2)$ souvislých úseků, neboť n způsoby můžeme zvolit začátek úseku i a pro každou volbu začátku můžeme $\mathrm{O}(n)$ způsoby zvolit koncový index úseku j. Projít úsek $a_i, \ldots, a_j$ a spočítat jeho součet představuje práci $\mathrm{O}(n)$, takže celková časová složitost tohoto triviálního algoritmu je $\mathrm{O}(n^3)$. Popsaný algoritmus vede ke správnému výsledku, ale je velmi neefektivní, takže v soutěži za něj bylo možné získat pouze 2 body z celkových 10 možných.

Naprogramovat uvedený postup je snadné. V Pythonu a také třeba ve všech programovacích jazycích z rodiny C se všechna pole a seznamy indexují od 0, zatímco v naší úloze jsou stánky na veletrhu označeny pořadovými čísly počínaje od 1. Tento drobný nesoulad vyřešíme v programu nejsnáze tím, že na začátek seznamu a s načtenými vstupními daty přidáme jeden fiktivní prvek, který tak dostane index 0 a s nímž poté v programu

už nebudeme vůbec pracovat.

```
n, k = map(int, input().split())
a = [None] + list(map(int, input().split()))

d, p = 0, 0
for i in range(1, n+1):
    for j in range(i, n+1):
        if j - i + 1 > d and sum(a[i:j+1]) <= k:
            d = j - i + 1
            p = i

print(d, p)
```

Výpočet můžeme značně urychlit lepší organizací práce. V předchozím řešení jsme počítali součet každého souvislého úseku zvlášť. Známe-li ovšem součet nějakého úseku $a_i, \ldots, a_j$ a chceme určit součet úseku o jeden prvek delšího $a_i, \ldots, a_{j+1}$, je zbytečné procházet tento nový úsek celý od začátku. Jednodušší a rychlejší je využít předchozí známou hodnotu součtu a pouze ji v konstantním čase upravit přičtením posledního prvku a_{j+1}. V programu tedy budeme postupně n způsoby volit začátek úseku i a pro každou volbu začátku zvolíme $\mathrm{O}(n)$ způsoby konec úseku j. Součet prvků v úseku $a_i, \ldots, a_j$ počítáme průběžně vždy s využitím známého součtu předchozího úseku, tedy v konstantním čase. Celková časová složitost takto upraveného algoritmu se tím sníží na $\mathrm{O}(n^2)$. Za takovéto řešení s kvadratickou časovou složitostí mohli řešitelé olympiády dostat nejvýše 4 body.

```
n, k = map(int, input().split())
a = [None] + list(map(int, input().split()))

d, p = 0, 0
for i in range(1, n+1):
    soucet = 0
    for j in range(i, n+1):
        soucet += a[j]
        if j - i + 1 > d and soucet <= k:
            d = j - i + 1
            p = i
        if soucet > k:
            break

print(d, p)
```

Při řešení různých úloh na zpracování posloupnosti čísel lze často úspěšně využít techniku tzv. prefixových součtů. Zamyslíme se, zda by nám něco

přinesla i v případě této úlohy. Řešení zahájíme předvýpočtem, při kterém si postupně pro každý koncový index j určíme součet prvních j členů posloupnosti, tzn. hodnotu $s_j = a_1 + a_2 + \ldots + a_j$. Všechny tyto součty lze postupně vypočítat v lineárním čase. Nejprve položíme $s_0 = 0$ a dále pak když už známe hodnotu s_i, snadno z ní spočítáme v konstantním čase hodnotu s_{i+1}. Po ukončení předvýpočtu můžeme přikročit k vlastnímu řešení úlohy. Tak jako v našem prvním řešení budeme postupně zkoumat všechny souvislé úseky zadané posloupnosti. Pro každou volbu počátečního indexu i a koncového indexu j nyní dokážeme určit součet tohoto úseku v konstantním čase jako $s_j - s_{i-1}$. Časová složitost předvýpočtu je $\mathrm{O}(n)$, časová složitost následného vlastního výpočtu je určena počtem souvislých úseků, což je $\mathrm{O}(n^2)$. Celý algoritmus využívající prefixových součtů má tedy stejnou časovou složitost $\mathrm{O}(n^2)$ jako naše předchozí řešení.

```
n, k = map(int, input().split())
a = [None] + list(map(int, input().split()))

s = [0] * (n+1)
for i in range(1, n+1):
    s[i] = s[i-1] + a[i]

d, p = 0, 0
for i in range(1, n+1):
    for j in range(i, n+1):
        if j - i + 1 > d and s[j] - s[i-1] <= k:
            d = j - i + 1
            p = i

print(d, p)
```

Zatím se zdá, že nám použití prefixových součtů nepřineslo nic nového. Již přece víme, že k vyřešení úlohy s kvadratickou časovou složitostí prefixové součty vůbec nepotřebujeme. Popsané řešení s prefixovými součty ale dokážeme ještě vylepšit pomocí binárního vyhledávání a dosáhneme tak podstatně lepší časové složitosti $\mathrm{O}(n \log n)$. Za takové řešení se již v soutěži udělovalo až 7 bodů.

Nejprve si opět v čase $\mathrm{O}(n)$ spočítáme prefixové součty zadané posloupnosti. Dále budeme procházet všechny počáteční indexy i a pro každý z nich nás zajímá, kterým indexem j bude končit co nejdelší souvislý úsek se součtem nepřevyšujícím hodnotu k. Již z minulého řešení víme, že tento úsek má součet rovný $s_j - s_{i-1}$. Všechny prvky zadané posloupnosti jsou kladná čísla, takže posloupnost jejich prefixových součtů je rostoucí. Ke zvolenému indexu i chceme tedy nalézt co největší j takové, aby platilo

$s_j - s_{i-1} \leq k$. K určení takového indexu j můžeme použít binární vyhledávání (metodu půlení intervalů), které má logaritmickou časovou složitost vzhledem k délce prohledávaného intervalu. Tímto intervalem, v němž hledáme index j, je rozmezí od i do n, takže časová složitost našeho binárního vyhledávání je $\mathrm{O}(\log n)$. Binární vyhledávání indexu j provádíme zvlášť pro každou volbu počátečního indexu i, celková časová složitost tohoto algoritmu je proto skutečně $\mathrm{O}(n \log n)$.

```
n, k = map(int, input().split())
a = [None] + list(map(int, input().split()))

s = [0] * (n+1)
for i in range(1, n+1):
    s[i] = s[i-1] + a[i]

d, p = 0, 0
for i in range(1, n+1):
    levy, pravy = i, n
    while levy <= pravy:
        j = (levy + pravy) // 2
        if s[j] - s[i-1] == k:
            break
        if s[j] - s[i-1] < k:
            levy = j+1
        else:
            pravy = j-1
    if levy > pravy:
        j = pravy
    if j - i + 1 > d:
        d = j - i + 1
        p = i

print(d, p)
```

Toto ale stále ještě není nejlepší možné řešení, naši úlohu lze řešit dokonce s lineární časovou složitostí vzhledem k délce zadané posloupnosti. Řešení s časovou složitostí $\mathrm{O}(n)$ již bude jistě časově optimální, neboť k nalezení správného výsledku jistě musíme zadanou posloupnost délky n celou projít.

Řešení s lineární časovou složitostí nepoužívá předvýpočet prefixových součtů, jako tomu bylo v předchozích uvedených řešeních. Algoritmus je založen na technice dvou postupujících ukazatelů, která se často využívá v podobných úlohách s posloupnostmi kladných čísel. Využijeme skutečnosti, že jestliže nejdelší souvislý úsek se součtem nejvýše k začínající v posloupnosti na indexu i končí na indexu j, potom nejdelší souvislý úsek

se součtem nejvýše k začínající na indexu $i + 1$ musí končit buď také na indexu j, nebo na nějakém indexu vyšším než j.

Během výpočtu si budeme udržovat dvě proměnné i, j a součet všech prvků zadané posloupnosti s indexy z rozmezí od i do j (včetně obou krajních mezí), který označíme *soucet*. Proměnné i, j v každém okamžiku vymezují právě zkoumaný úsek posloupnosti. Na začátku výpočtu bude $i = 1$, $j = 1$ a v průběhu algoritmu budou obě tyto proměnné pouze růst. Pro každou hodnotu proměnné i budeme postupně zvyšovat j a současně přepočítávat hodnotu *soucet* tak dlouho, dokud $j \leq n$ a zároveň *soucet* $\leq k$. Tím najdeme nejdelší souvislý úsek se součtem nejvýše k začínající na indexu i. Jeho délku si zaznamenáme a pokud jsme s j došli až na konec posloupnosti, výpočet ukončíme. V opačném případě proměnnou i zvýšíme o 1, čímž se patřičně sníží součet úseku uložený v proměnné *soucet*. Proměnnou j přitom ponecháme na poslední hodnotě z předchozího kroku výpočtu a popsaný výpočet zopakujeme pro novou hodnotu i. Výsledkem je nejdelší ze zaznamenaných délek.

```
n, k = map(int, input().split())
a = [None] + list(map(int, input().split())) + [0]

d, p = 0, 0
i, j, soucet = 1, 1, a[1]
while j <= n:
    if soucet <= k:
        if j - i + 1 > d:
            d = j - i + 1
            p = i
        j += 1
        soucet += a[j]
    else: #soucet > k
        soucet -= a[i]
        i += 1

print(d, p)
```

Uvedený algoritmus pro každé i najde největší možné j takové, aby součet úseku posloupnosti uložený v proměnné *soucet* nepřevýšil hodnotu k. Nakonec proto správně určí délku nejdelšího souvislého úseku posloupnosti se součtem nejvýše k. Časová složitost popsaného algoritmu je skutečně $O(n)$, neboť v každém kroku výpočtu buď zvýšíme hodnotu i, nebo hodnotu j; obě tyto proměnné jsou však na začátku rovny 1 a nikdy nepřesáhnou hodnotu n.