1. Série domácích úkolů

Odevzdávejte do 27. 10. 14:00

1. *Naivní algoritmus může být pomalý:* [5b]
   Naivní algoritmus, který zkouší všechny možné začátky jehly v seně a vždy porovnává řetězce, má časovou složitost $\mathcal{O}(JS)$. Může být opravdu tak pomalý, uvážíme-li, že porovnávání řetězců skončí, jakmile najde první neshodu? Sestrojte vstup, na kterém algoritmus poběží $\Theta(JS)$ kroků, přestože nic nenajde.

2. *Nejdelší vlastní prefix, který je sufixem:* [5b]
   Je dáno slovo. Chceme nalézt jeho nejdelší vlastní prefix, který je současně suffixem.

3. *Příliš mnoho jehel:* [5b]
   Nalezněte příklad jehel a sena, v němž je asymptoticky více než lineární počet výskytů. Přesněji řečeno ukažte, že pro každé n existuje vstup, v němž je součet délek jehel a sena $\Theta(n)$ a počet výskytů není $\mathcal{O}(n)$.

4. *Počet výskytů:* [5b]
   Mějme seno a jehly. Popište algoritmus, který v lineárním čase pro každou jehlu spočítá, kolikrát se v seně vyskytuje. Časová složitost by neměla záviset na počtu výskytů – ten, jak už víme, může být superlineární.

1) I když porovnání dvou řetězců ukončím po první neshodě: cabc / bbc
tak bude výsledná složitost následující:

Délka prefixu jehly — počet porovnání

- 1 -> S
- 2 -> $\frac{S}{2}$
- 3 -> $\frac{S}{3}$
- ⋮
- $|J|$ -> $\frac{S}{J}$

-> uvažujeme-li, že vždy dojde k neshodě hned v prvním písmenu

$$\sum_{i=1}^{J} \frac{S}{i} = S \cdot \sum_{i=1}^{J} \frac{1}{i} = S \cdot H_J \simeq S \cdot \log J$$

Což je sice hrubé ale přece $\mathcal{O}(SJ)$.

Za předpokladu, že $H_J :=$ J-tí harmonické číslo, které hrubě odhadujeme $\log$

Vstup, pro který by naivní alg. vyšel časově $\Theta(|S|)$:

Seno, které bude k-násobkem $J[:-1]$, tedy $S = k \cdot J[:-1]$

Např.:

$J = abc$

$S = \underbrace{abab\ldots ab}_{k \text{ krát}}$

$O(J \cdot S)$

Pak:

Délka prefixu jehly	# porovnání
1	1. $\lvert S\rvert$
2	~~2.~~ ~~$\frac{\lvert S\rvert}{2}$~~
3	~~3.~~ ~~$\frac{\lvert S\rvert}{3}$~~
⋮	⋮
J	~~J.~~ ~~$\frac{\lvert S\rvert}{J}$~~

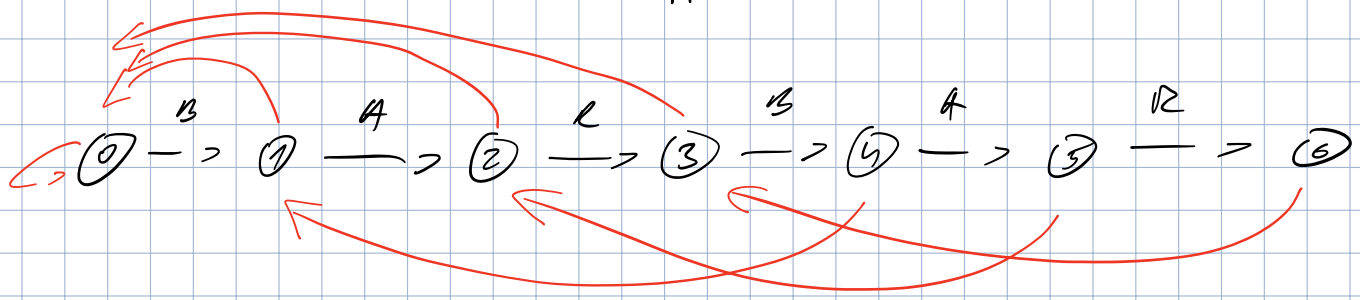
2) Nejdelší vlastní prefix, co je suffix:

ind. exc.

BARBAR -> LPS = BAR

prefix := S[:i+1]

suffix := S[i:]



0-0 0-0 0-0 0-0 1-0 2-0 3-0

Zpětná hrana z posledního stavu KMP automatu povede na nejbližší další výskyt takového prefixu. Pokud bude chtít hledat po zpětných hranách z poslední pozice KMP automatu a bude sčítat rozdíly indexů stavů, dostane index nejdelšího prefixu, co je zároveň suffix.

Složitost algoritmu je $O(|J|+|S|)$, jelikož $O(|J|)$ trvá vytvořit KMP automat a průchod po zpětných hranách z posledního stavu musí být také nanejvýš $O(|S|)$.

Níže přiložený kód:

```
def Krok(i: int, x: str, input_string: str, z: array):
    while input_string[i] != x:
        if (i == 0):
            return 0
        i = z[i]
    return i+1

def VytvorKMP(input_string: str):
    z = [0]*(len(input_string)+1)
    z[0] = 0
    z[1] = 0
    i = 0
    for j in range(2, len(input_string)+1):
        i = Krok(i, input_string[j-1], input_string, z)
        z[j] = i
    return z

def NajdiNejdelsiPrefixCojeSuffix(z: array, input_string: str):
    t = z[len(z)-1]
    final_len = 0
    while (t != 0):
        final_len += t - z[t]
        t = z[t]
    return input_string[:final_len]

retezec = input("Zadejte retezec: \n")
print(NajdiNejdelsiPrefixCojeSuffix(VytvorKMP(retezec),retezec))
```

→ Skákání zpět po zpětných hranách

→ sčítání si rozdílů indexů pro finální délku prefixu.

3) Příliš mnoho jehel

$S = \text{„}\underbrace{AAA\ldots AA}_{n}\text{“}$

$J = \{ \underbrace{\text{„}AA\ldots AA\text{“}}_{n/2}, \underbrace{\text{„}A\ldots A\text{“}}_{n/4}, \underbrace{\text{„}A\ldots\text{“}}_{n/8}, \ldots \}$ → dokud $n/2^i > 1$

$|S| = n$

$|J| = \sum_{i=1}^{\sqrt{n}} n/2^i \in n \cdot \sum_{i=1}^{\infty} 1/2^i = n \cdot 1$

$S, J \in O(n)$, ale výskyty vostan rychlejí:

Délka prefixu sama	# výskytů jehel
1	1
2	3
3	6
4	10
5	15
6	21

→ Avšak počet výskytů pro takovéto jehly roste očividně rychleji jak $O(n)$

4) Počet výskytů jehel —> lineární přístup

Pokud jde o lineární přístup vzhledem k senu a jednotlivé jehle, lze využít KMP alg., který pracuje v čase $O(J+S)$.

Ten pracuje tak, že v čase $O(J)$ vytvoří vyhledávací automat pro danou jehlu:

> to dvěma průchody, kdy nejdříve sestrojím dopředné hrany a následně v druhém průchodu pustím na dejný automat substringy jehly $J[1:]$ a budu si pamatovat stavy, do kterých jsem vstoupil. Tím efektivně vytvořím zpětné hrany a tudíž i v čase $O(J)$ celý vyhledávací automat pro jednotlivou jehlu.

Následně pro zadané seno stačí automat projít a vždy, kdykoli dojdu do koncového stavu automatu, inkrementuji #výskytů, který po celém průchodu senem vrátím jako finální počet výskytů. Celkem to pro každou jehlu bude práce $O(J+S)$.