

Några svar till TDDD86 Datastrukturer, algoritmer och programmeringsparadigm

2015-08-24

Följande är lösningsskisser och svar till uppgifterna på tentan. Lösningarna som ges här ska bara ses som vägledning och är oftast inte tillräckliga som svar på tentan.

- 2 (a) $O(n^2)$ resp. $O(n \log n)$.
(b) Använd ett splayträd istället.
- 3 (a) En hashtabell **originals** som avbildar strängar på mängder av strängar.
Huvudidén är att undvika att generera alla permutationer av varje ord (vilket kräver $L!$ tid).
 - Läs varje ord **word** från indata, sortera det med insertion sort för att få **dorw** och lägg till **word** i mängden **originals[dorw]** (skapa mängden om den inte finns). Håll reda på största mängdstorleken. $O(NL^2)$.
 - Iterera över **originals**. Stanna när en mängd av maximal storlek hittas. Skriv ut orden i denna mängd. $O(N)$.

Man kan tänka sig att sortera varje ord med räknesortering, vilket skulle ge tidskomplexitet $O(NLR)$, där R är storleken på alfabetet, men insertion sort ($O(NL^2)$) är antagligen snabbare i praktiken för ord från något naturligt språk.

- (b) Vi behöver undvika en exponentiell sökning efter alla möjliga ordstegar. Här är en sådan lösning.

Vi behöver en metod **neighbours(dorw)** som antar att **originals** redan finns. Givet ett ord w av längd k genererar metoden de k ord av längd $k - 1$ som w ger upphov till och returnerar de som är giltiga (dvs de som är nycklar i **originals**). $O(L^2)$.

- Initialisera en hashtabell **rung_height** från sorterade ord till **int**:ar med alla värden satta till 0. **rung_height** för varje **dorw** är maximala stegpinnehöjden för **dorw** över alla anagramstegar **dorw** kan finnas med i. Lägsta stegpinnehöjden räknas som 0.
- Skapa en sorterad array av sorterade ord i ökande längdordning. $O(N)$ tid med räknesortering.
- För varje sorterat ord **dorw** i arrayen:

```
rung_height[dorw] = max(rung_height[nbr] for nbr in neighbours[dorw]) + 1
// Om neighbours[dorw] är tom, gör ingenting, eftersom dorw måste vara
// lägsta stegpinne i alla möjliga stegar. Notera att grannarna redan
// behandlats pga sorteringen.
```

 N anrop till **neighbours**, $O(NL^2)$.
- Hitta det sorterade ordet med störst **rung_height** och sök iterativt efter en sekvens av grannar med **rung_height** ett mindre än föregående. $O(N + \text{poly}(L))$.
- Slå upp de sorterade orden i denna steg (baklänges) i **originals** och skriv ut en sekvens av ord i språket.

- 4
- A. Om vi vill sortera en mängd av slumpvis ordnade element så att vi får bäst prestanda (utan att bry oss om stabilitet) bör vi använda quicksort.
 - A eller C. Även i det här fallet vill vi ha hög prestanda, men bryr oss inte om stabilitet. Om observationerna är slumpmässigt ordnade vinner quicksort. Ett rimligt antagande är att den osorterade **Observation**-arrayen fyllts i efter någorlunda stigande tidsstämpel. I så fall vill vi använda insertionsort för att dra nytta av att arrayen innehåller partiellt ordnat data.
 - B. I det fallet vill vi ha effektivitet och stabilitet och objekten är ordnade slumpmässigt efter **importance**. Den vinnande algoritmen blir mergesort.
 - C. Här har vi en array som är nästan helt ordnad, så vi vill använda insertionsort.