Mästarprov del 1

Christopher Lillthors 911005-3817

Kurs: ADK
Kurskod: DD1352

13 oktober 2015

Innehåll

1 Hantverkare **1**

2 Mängder av anagram **2**

.

Generated in LaTeX .

Mästarprov del 1

Kungliga Tekniska Högskolan 13 oktober 2015
Civilingenjör datateknik
ADK DD1352

1 Hantverkare

Genom att studera problemet inser man snabbt att ett delproblem är att avgöra vilken permutation av indatan som i slutändan lämpar sig bäst/i vilken ordning hantverkarna ska jobba (detta är ett exempel på "Shortest Job First"). Tiderna $\{t_1, t_2, t_3, \ldots, t_n\}$ kan permuteras på något sätt till $\{x_1, x_2, x_3, \ldots, x_n\}$

Med lite utforskande kommer man fram till att den totala kostnaden för alla hantverkare kan sammanfattas på följande vis:

$$C = c_h(x_1(n) + x_2(n-1) + x_3(n-2) + \ldots + x_n) \tag{1}$$

där c_h är den fasta timlönen

Med detta framför oss är det trivialt att inse att för att få en så liten summa som möjligt måste delsummorna vara så små som möjligt, ty n är potentiellt "stort" men blir senare mindre och mindre. För att kompensera för detta bör därför x_1 sättas till den kortaste tiden, x_2 till den näst kortaste tiden osv.

Algoritmen börjar med att skapa en $min-heap$ vilket i grund och botten är ett träd, men där löven med lägst värde ligger närmare roten. Det absolut lägsta lövet är således roten, vilket innebär att man kan få tag på det minsta elementet i $\mathcal{O}(1)$ tid men att göra en insättning/borttagning görs i $\mathcal{O}(log_2(n)$ (motsvarar trädets djup).

När denna heap är skapad plockar vi helt enkelt ut och tar bort de lägsta löven för varje iteration som går. Givet att det finns n stycken tider kommer slingan gå n gånger och tillhör således $\mathcal{O}(n)$. Inne i slingan utförs en hämtning/borttagning av det för tillfället största lövet vilket innebär $\mathcal{O}(1 + log_2(n)) \in \mathcal{O}(log_2(n))$ därefter beräknas delsumman för totalkostnaden av arbetet som beskrivet ovan och delresultatet sparas i en variabel för att kunna användas till nästkommande iteration.

Att algoritmen tillhör $\mathcal{O}(n * log_2(n))$ faller sig därmed naturligt med ovan beskrivning.

Algoritmen är girig och kommer alltid att returnera ett svar vilket förklarar sig i att det inte finns några sätt för programmet att ta en omväg kring sorteringen och alla beräkningar som utförs.

Procedure 1 Hantverkare
Input: $\{t_1, t_2, t_3, \ldots, t_n\} \in \mathbb{Q}_+$
Output: Minsta totalkostnaden för utfört arbete

```
1: function MinCost(times)
2:     q ← heap(times)                                   ▷ O(n * log2(n))
3:     minCost ← 0                                       ▷ O(1)
4:     while q.length() ≠ 0 do                           ▷ O(n)
5:         time ← q.poll()                               ▷ O(log2(n)
6:         minCost ← minCost + 100 * time * (1 + q.length())   ▷ O(1)
7:     end while
8:     return minCost
9: end function
```

2 Mängder av anagram

Algoritmen börjar med att ord skickas in i en lista, där n är listans längd och m är längden på listans längsta ord. Då vi inte har någon aning till en början om vilka ord som är anagram med varandra skapas en cache som kan hålla orden i väntan på att ett unikt hash för alla ord som är anagram till varandra beräknas under körningen. Sortering med radixSort(bucket sort) sker främst eftersom att utdatan skall vara sorterad och vara lexigrafiskt ordnat. Detta kostar i regel $\mathcal{O}(n * m)$.

Unik nyckel som sammanbinder anagram med varandra kan uppnås med en primtalsprodukt (alla bokstäver i alfabetet mappas mot ett primtal och multipliceras till ett enda stort tal), lexical sortering av strängarna samt ett hash över en *distributionsslista* dvs en lista av j element där j är antalet bokstäver i alfabetet. Med hjälp av detta skapas en *distributionslista* som en förteckning av hur många bokstäver av samma sort som finns i ordet. Ex: $ABBA \Rightarrow [2, 2, 0, ..., 0]$

Alla ovanstående sätt att beräkna en nyckel på är unika till just den mängden anagram, vilket jag nu tänker visa

Primtalsprodukt: Det tillhör grundläggande aritmetik vad definitionen av ett primtal är, men poängen är att detta tal vi får fram ej kan representeras på ett annat sätt än just en omkastning av samma bokstäver, dvs ett anagram.

Lexical sortering: Om 2 strängar bildar efter sortering samma sträng är dessa 2 ursprungliga strängar anagram.

Distributionslista: Om 2 strängar använder samma bokstäver exakt lika många gånger måste dessa två strängar vara anagram.

Eftersom att nyckeln vid multiplicering av primtal kan bli väldigt stora har

jag beslutat mig för att lägga in ett vilkor som avgör vilken metod som ska utföras för just det aktuella ordet. Just den siffran är baserad på antagandet att man skapat en map som börjar vid primtalet 2 och jobbar sig uppåt till närmaste primtal upp till ö. $113^9 \approx 3 * 10^{18}$ vilket är i närheten för vad ett 64-bitars tal klarar av att representera. Mitt värsta fall bygger alltså på den högst orimliga strängen "öööööööööö" vilket är långt ifrån ett giltigt ord.

Istället bygger jag då en nyckel genom en distributionslista och sedan ett hash (java eller dylikt) på det för att få ett unik tal som mappar till en unik lista i min cache.

Analys av algoritmen visar oss att vi måste loopa igenom alla ord exakt en gång men beroende på längden av det aktuella ordet kan vi hamna i 2 olika branches. Den "första" beräknar en nyckel med hjälp av en distributionslista och sedan insättning på rätt plats i cachen. Att skapa en distributionslista är trivialt och tvingar oss att loopa igenom det aktuella ordet för att veta exakt hur många instanser av en viss bokstav det finns. Det utförs för hela strängen och eftersom att den längsta strängen är n lång fastslår vi att operationen måste tillhöra $\mathcal{O}(n)$ medan beräkning av hash är ytterligare en linjär operation över listan måste även denna tillhöra $\mathcal{O}(n)$ och till slut insättning som förutsätter konstant lookup i en map och konstant insättning i en lista (länkad lista) Det totala arbetet som behöver genomföras är alltså $\mathcal{O}(n * m)$ i den branchen.

I den andra branchen däremot antas det att talen inte blir allt för stora och att konstant kostnad vid multipliktation gäller. När nyckeln för det aktuella ordet byggs upp vilar vi på primtalens egenskaper och dessa är mycket tydliga. Vi kan omöjligen få samma nyckel med ett ord som inte anagram med det aktuella ordet.

Även denna branch har samma tidskomplexitet som den förra, vilket också visar att övre och undre gräns sammanfaller, ty den måste in i någon av brancherna och utföra arbete. Som sista arbete loopar vi igenom cachen och plockar ut våra listor, cachens storlek är maximalt m stor och innehåller således maximalt m sorterade listor. Dessutom kommer den inte förbi sorteringen och saken är klar. Algoritmen är optimal.

Procedure 2 Mängder av anagram

Input: En lista av m ord där listans längsta ord är n lång
Output: En sorterad lista av anagram

```
1:  function FindAnagrams(words)
2:      cache ← map(Integer, List[m])
3:      sortedWords ← radixSort(words)                    ▷ O(n * m)
4:      for word in words do                              ▷ O(m)
5:          if length(word) ≥ 9 then
6:              key ← hash(distribution(word))            ▷ O(n)
7:              cache[key].insert(word)                   ▷ O(1)
8:          else
9:              key ← 1
10:             for character in word do                  ▷ O(n)
11:                 key ← key * getPrime(character)       ▷ O(1)
12:             end for
13:             cache[key].insert(word)                   ▷ O(1)
14:         end if
15:     end for
16:     return cache.values()                             ▷ O(m)
17: end function
```