Achtste college algoritmiek

5 april 2012

Restant Decrease and conquer
Dynamisch Programmeren

Opgave 10: Dutch Flag Problem

Gegeven een array gevuld met R, W, en B.
Reorganiseer dit array zo dat van links naar rechts eerst alle 'R', dan de 'W' en dan de 'B' komen te staan. Het algoritme moet lineair zijn en in situ (alleen interne verwisselingen). Het mag maar één doorgang door het array maken.


```
// invoer: array A[0...n-1] waarin A[i] = 'R', 'W' of 'B'
// uitvoer: array A[0...n-1] waarin eerst alle 'R' komen,
//          dan alle 'W', en ten slotte alle 'B'
r := 0; w := 0; b := n-1;
while w <= b do
   if A[w] = 'R' then
      wissel(A[r],A[w]); r := r+1; w := w+1;
   else
      if A[w] = 'W' then
         w := w+1;
      else  // A[w] = 'B'
         wissel(A[w],A[b]); b := b-1;
      fi
   fi
od
```


Variable-size decrease: de reductie in grootte is variabel, dus kan in elke stap anders zijn

Voorbeelden:

- Algoritme van Euclides
  Dit is gebaseerd op: ggd(m, n) = ggd(n, m **mod** n)
- Flipping pancakes (Levitin, excercise 4.5.12 **(zie tweede editie, 5.6.11)**)

- Binaire zoekbomen

Probleem: *Gegeven* twee niet-negatieve gehele getallen m en n (niet beide nul). *Vraag*: wat is de grootste gemeenschappelijke deler, genoteerd als ggd(m,n), van m en n?

Voorbeeld: ggd(60,24) = 12; ggd(25,0) = 25; ggd(200, 441) = 1; ggd(588,495) = 3.

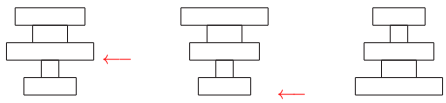
Het algoritme van Euclides is gebaseerd op het herhaald gebruiken van de gelijkheid

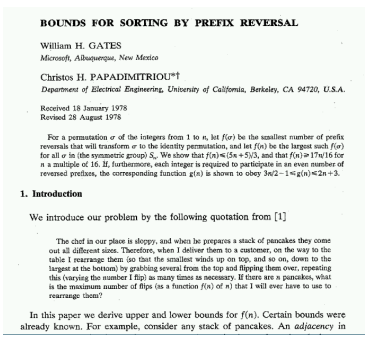
$$\text{ggd}(m, n) = \text{ggd}(n, m \bmod n),$$

totdat de tweede parameter nul wordt.

Voorbeeld: ggd(60,24) = ggd(24, 12) = ggd(12, 0) = 12

Probleem: Gegeven een stapel van n pannenkoeken, allemaal verschillend in grootte. Verder is alleen een spatel beschikbaar, die je onder een pannenkoek kan schuiven, waarna je de hele stapel daarbovenop in één keer kan omdraaien. De bedoeling is om uiteindelijk alle pannenkoeken bovenop elkaar te krijgen in volgorde van grootte (de grootste onderop).





BOUNDS FOR SORTING BY PREFIX REVERSAL

William H. GATES
Microsoft, Albuquerque, New Mexico

Christos H. PAPADIMITRIOU*†
Department of Electrical Engineering, University of California, Berkeley, CA 94720, U.S.A.

Received 18 January 1978
Revised 28 August 1978

For a permutation σ of the integers from 1 to n, let $f(\sigma)$ be the smallest number of prefix reversals that will transform σ to the identity permutation, and let $f(n)$ be the largest such $f(\sigma)$ for all σ in (the symmetric group) S_n. We show that $f(n) \leq (5n+5)/3$, and that $f(n) \geq 17n/16$ for n a multiple of 16. If, furthermore, each integer is required to participate in an even number of reversed prefixes, the corresponding function $g(n)$ is shown to obey $3n/2 - 1 \leq g(n) \leq 2n+3$.

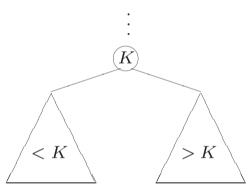
1. Introduction

We introduce our problem by the following quotation from [1]

> The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are n pancakes, what is the maximum number of flips (as a function $f(n)$ of n) that I will ever have to use to rearrange them?

In this paper we derive upper and lower bounds for $f(n)$. Certain bounds were already known. For example, consider any stack of pancakes. An *adjacency* in

Een binaire zoekboom is een binaire boom waarbij voor elke knoop geldt dat de waarde in die knoop groter is dan alle waarden in zijn linkersubboom, en kleiner dan alle waarden in zijn rechtersubboom.



Bij het zoeken naar een waarde in een gewone binaire boom (bijv. WLR) moeten in het slechtste geval alle n knopen bekeken worden. Zoeken in een binaire zoekboom is i.h.a. efficiënter: in het slechtste geval worden $h+1$ knopen bekeken, met h de hoogte van de boom.

```
knoop* zoeken(knoop* root, int getal) {
  if ( root == null )      // lege boom
    return null;
  else
     if ( root->info == getal )       // gevonden!
        return root;
     else
        if ( getal < root->info )
           return zoeken(root->links, getal);
        else
           return zoeken(root->rechts, getal);
}  // zoeken
```


Bekijk en vergelijk vier verschillende oplossingsmethoden voor het berekenen van a^n:

1. Brute force: gebaseerd op de definitie, $a^n = \overbrace{a * \cdots * a}^{n\times}$
2. Divide and conquer: gebaseerd op $a^n = a^{\lfloor \frac{n}{2} \rfloor} * a^{\lceil \frac{n}{2} \rceil}$
3. Decrease by one: gebaseerd op $a^n = a^{n-1} * a$
4. Decrease by a constant factor: gebaseerd op

$$a^n = \begin{cases} (a^{\frac{n}{2}})^2 & \text{als } n \text{ even is} \\ (a^{\frac{n-1}{2}})^2 * a & \text{als } n \text{ oneven is} \end{cases}$$

Definitie Fibonacci-getallen:

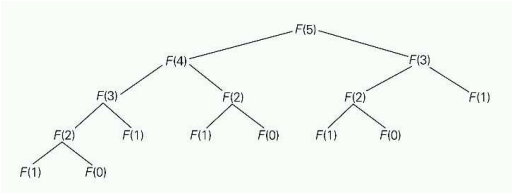
$$\text{fib}(n) = \begin{cases} 0 & \text{als } n = 0 \\ 1 & \text{als } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{als } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, ...

Recursieve C++-functie:

```
long fib1 (int n) {
   if ( ( n==0 ) || ( n == 1) )
      return n;  // gaat goed!
   else
      return ( fib1 (n-1) + fib1 (n-2) );
} // fib1
```

Watervaleffect



Voor $n = 5$ worden sommige recursieven aanroepen meerdere malen gedaan. Voor grotere waarden van n wordt dit watervaleffect steeds groter. Dit komt doordat deelproblemen elkaar overlappen.

Oplossing: gebruik een array om tussenresultaten op te slaan, en los op die manier elk deelprobleem precies één keer op.

Dit kan op twee manieren:

1. Top down: memory function
   Combineert recursie met het gebruik van een array
2. Bottom up: het klassieke dynamisch programmeren (DP)
   Vult het array van klein naar groot (for-loop)


```
const int MAX = 45;
long fib2 (int n) {  // recursie met array !
   static long memo[MAX] = {-1};  // eenmalig op -1
   if ( memo[n] > -1 ) // al eerder berekend
      return memo[n];
   else {
      if ( ( n==0 ) || ( n == 1) )
         memo[n] = n;  // gaat goed!
      else
         memo[n] = fib2 (n-1) + fib2 (n-2);
      return memo[n];
   } // else
} // fib2
```


Dynamisch programmeren: gebruikt ook een array voor het opslaan van tussenresultaten, maar werkt bottom up. Gebruikt de recurrente betrekking waaraan de Fibonacci-getallen voldoen.

```
fibo[0] = 0;
fibo[1] = 1;
for (i=2; i<=n; i++) {
    fibo[i] = fibo[i-1] + fibo[i-2];
}
return fibo[n];
```

Je hebt overigens niet het hele array nodig, maar je kunt volstaan met 3 (of zelfs 2) variabelen. Zo krijg je de bekende iteratieve oplossing (zie ook Programmeermethoden).

- nuttig bij problemen met *overlappende deelproblemen*
- druk een oplossing van het probleem uit in oplossingen van deelproblemen (*recursieve formulering*)
- deeloplossingen worden opgeslagen in een *tabel* zodra ze berekend zijn, waardoor elk deelprobleem maar *één keer* hoeft te worden opgelost
- na afloop bevat (of is) de tabel de oplossing van het oorspronkelijke probleem
- DP is van oorsprong een *bottom up* methode: start met de kleine gevallen en combineer hun oplossingen tot oplossingen van steeds grotere gevallen
- er is ook een *top down* variant (*memory function*)

- de bottom up methode is *iteratief*, de top down variant is recursief
- bottom up lost *alle* deelproblemen op, top down alleen degene die echt nodig zijn voor het oplossen van het oorspronkelijke probleem
- bij beide varianten wordt eenzelfde soort tabel gebruikt
- bij bottom up wordt de tabel in een *bepaalde volgorde* gevuld, bij top down gebeurt dat meer willekeurig
- bij de bottom up manier is vaak een qua geheugengebruik *efficiënter* algoritme af te leiden

We willen een busreis maken langs steden $0, 1, 2, \ldots, n$, in die volgorde. Aangezien meerdere busmaatschappijen op de verschillende (deel)trajecten rijden, zijn de prijzen voor een rit van plaats i naar plaats j (via alle tussenliggende steden) per bus verschillend. Het kan dus voordeliger zijn om in plaats van rechtstreeks met de goedkoopste bus van plaats 0 naar n te reizen, (een paar keer) over te stappen en met een andere bus verder te gaan.

Laat prijs$[i][j]$, de prijs van het goedkoopste buskaartje rechtstreeks van i naar j, gegeven zijn voor alle $i \leq j$. Het probleem is nu om de prijs van de goedkoopste reis van 0 naar n te vinden.

Laat kosten(n) de prijs van de goedkoopste busreis van 0 naar n voorstellen, langs alle tussenliggende steden (in oplopende volgorde). Dan geldt:

$$\text{kosten}(n) = \begin{cases} 0 & \text{als } n = 0 \\ \min_{0 \leq k < n}(\text{kosten}(k) + \text{prijs}[k][n]) & \text{als } n \geq 1 \end{cases}$$

Voorbeeld:

$$\text{prijs} = \begin{pmatrix} 0 & 5 & 10 & 15 \\ - & 0 & 7 & 13 \\ - & - & 0 & 4 \\ - & - & - & 0 \end{pmatrix}$$

De prijs van de goedkoopste busreis van 0 naar 3 is hier 14 (met tussenstop in plaats 2).

Een recursief algoritme:

```
kosten(n)::
    if n=0 then
       return 0;
    else
       temp := prijs[0][n];   // k = 0
       for k := 1 to n-1 do
           hulp := kosten(k) + prijs[k][n];
           if hulp < temp then
               temp := hulp;
           fi
       od
       return temp;
    fi .
```


De recursieve oplossing doet exponentieel veel aanroepen, en er is heel veel overlap tussen de deelproblemen. Oplossing: deeloplossingen opslaan in een geschikt array.

Laat kosten$[i]$ de prijs van de goedkoopste busreis van 0 naar i voorstellen, langs alle tussenliggende steden (in oplopende volgorde). We zoeken dus kosten$[n]$.
Dan geldt:

$$\text{kosten}[i] = \begin{cases} 0 & \text{als } i = 0 \\ \min_{0 \leq k < i}(\text{kosten}[k] + \text{prijs}[k][i]) & \text{als } i \geq 1 \end{cases}$$

We gaan het array nu bottom up vullen. Merk op dat om kosten$[i]$ te berekenen, *alle* kleinere waarden kosten$[k]$ met $k < i$ nodig zijn. Die moeten dus al eerder berekend zijn. We moeten het array derhalve van links naar rechts vullen.


```
kosten[0] := 0;
for i := 1 to n do
      temp := prijs[0][i]; // met 1 bus, zonder overstappen
      for k := 1 to i − 1 do
          hulp := kosten[k]+prijs[k][i];
          if hulp < temp then
              temp := hulp; // goedkoopste tot dusver
          fi
      od
      kosten[i] := temp;
od
return kosten[n];
```

Het algoritme is eenvoudig zo aan te passen dat ook de tussenstops van de goedkoopste reis worden gevonden.


```
kosten[0] := 0; stop[0] := 0;
for i := 1 to n do
      temp := prijs[0][i]; tempstop:= 0; // met 1 bus
      for k := 1 to i − 1 do
          hulp := kosten[k] + prijs[k][i];
          if hulp < temp then
              temp := hulp; // goedkoopste tot dusver
              tempstop := k; // bijbehorende tussenstop
          fi
      od
      kosten[i] := temp; stop[i] := tempstop;
od
return kosten[n];
```

Hierin is stop$[i]$ steeds de laatste tussenstop op de goedkoopste reis van 0 naar i.

- **Lezen/leren bij dit college:**

  Sheets; inleiding hoofdstuk 8

- **Werkcollege:**

  donderdag 5 april 2012, 13:45–15:30, in zaal Archipel

- **Opgaven:**

  www.liacs.nl/home/rvvliet/algoritmiek/

- **Volgend college:** donderdag 12 april 2012