DYNAMISCH PROGRAMMEREN 2
OPGAVE 15.4-5

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a set of n nembers.

ANTWOORD

Daar hoeven we niet moeilijk over te doen: Voer LCS (Longest Common Subsequence) uit op de rij en een gesorteerde kopie van die rij. Het kopiëren en sorteren kost $O(n \log n)$-tijd, maar de looptijd wordt gedomineerd door het LCS algoritme: $O(n^2)$.

SPECIAAL DP-ALGORITME

Voor de gein zullen we echter een speciaal dynamisch-programmeren-algoritme ontwerpen voor dit probleem. We gaan er in het begin even van uit dat we alleen de *lengte* van de langste monotoon stijgende deelrij[1] (LMSD) willen weten. Vaak is het makkelijk om achteraf het algoritme uit te breiden zodat die deelrij ook daadwerkelijk wordt opgeleverd. De eerste stap is het zoeken van een recurrente betrekking die het probleem beantwoordt met behulp van het antwoord op kleinere versies van het probleem:

Recurrente betrekking

Wat zien we als een kleinere versie van het probleem? Een voor de hand liggende keuze is 'de lengte van de LMSD van de eerste $n-1$ getallen uit de lijst'. Helaas loopt deze poging in de soep: Stel dat we een lijst van 10 getallen hebben, en we weten dat de langste LMSD van de eerste 9 getallen 5 lang is. Nu hebben we nog geen idee of we het 10e getal hier aan kunnen toevoegen en het antwoord voor de hele invoer dus 5 of 6 is! Het probleem is namelijk dat we niet weten wat het laatste getal in die rij van 5 was, en dus of het 10e getal hoger is, want alleen dan mogen wie die toevoegen.

Gelukkig is dit probleem meteen een mooie voorzet voor de oplossing: We nemen als kleinere versie van het probleem 'de lengte van de LMSD van de eerste $n-1$ getallen uit de lijst, *die eindigt met getal* $n-1$'. We leggen dus als extra eis op dat deze deelrij eindigt met het laatste getal in de kleinere rij. Definieer $f(i)$ als de lengte van de LMSD van de eerste i getallen die eindigt met het ie getal.

Dit levert een mooie oplossing op: voor $f(i)$ nemen we dus zeker getal i aan het eind, en dan de langste LMSD van de eerdere getallen die eindigen met een getal dat kleiner-of-gelijk aan het ie getal. Als recurrente betrekking ziet dat er zo uit (a_i is het ie getal in de invoer):

(We doen even weer alsof het maximum van een lege verzameling 0 is)

[1] Voor de duidelijkheid: een monotoon stijgende rij is een rij waarvoor ieder element groter-of-gelijk aan het vorige is (dit in tegenstelling tot een *strikt* monotoon stijgende rij, waarbij ieder element strikt groter is dan het vorige). Een deelrij is een deel van de elementen van een rij, maar wel in dezelfde volgorde, bijvoorbeeld $\langle 1,4,8 \rangle$ is een deelrij van $\langle 1,8,4,3 \rangle$, maar $\langle 8,1 \rangle$ is dat niet.

$$f: \{0, \ldots, n\} \to \mathbb{N}$$
$$f(i) = \begin{cases} 0 & \text{als } i = 0 \text{ (de LMSD van een lege rij is natuurlijk leeg)} \\ 1 + \max_{j \in \{1,\ldots,i-1\} \wedge a_j \leq a_i} f(j) & \text{anders} \end{cases}$$

Uiteraard is in het oorspronkelijke probleem niet gezegd dat het laatste element er bij moet zitten, dus het antwoord op het probleem is niet zomaar $f(n)$. We weten niet welk element het laatste zal zijn van de LMSD, maar we kunnen ze allemaal proberen en de beste nemen: het antwoord op het hele probleem is $\max_{i \in \{0,\ldots,n\}} f(i)$.

Recursie vs. Dynamisch Programmeren

In principe hebben we het probleem hiermee opgelost! We kunnen namelijk gewoon deze recurrente betrekking recursief gaan uitrekenen, alle informatie die we nodig hebben is aanwezig. Maar overtreden we niet de tijdsgrens als we dit doen?

Als we domweg $f(10)$ gaan uitrekenen, gaat die – om het maximum te vinden – alle $f(0)$ t/m $f(9)$ uitrekenen. En $f(9)$ gaat op zijn beurt rustig $f(0)$ t/m $f(8)$ uitrekenen (die elk allemaal weer in recursie gaan)… Dit is een exponentieel algoritme!

Maar – en hier komt de clou van Dynamisch Programmeren – de recursieboom mag dan wel heel veel knopen bevatten, het zijn er maar heel weinig verschillende. Stel dat we de instanties van f van klein naar groot uitrekenen en alle uitkomsten in een tabel onthouden. Dat kan prima, want om $f(i)$ uit te rekenen heb je alleen maar $f(j)$ nodig voor $j < i$, en omdat we die uitkomsten onthouden hebben doen we geen werk dubbel.

Het algoritme

Het is tijd om een concreet algoritme te gaan maken. (De invoer array ‘a’ begint voor het gemak op index 1 met het eerste getal.)

```
int[] f = new int[n+1];      // de tabel met uitkomsten van f, n is de lengte van de invoer
f[0] = 0;                    // initialisatie van het randgeval
for (int i=1; i<=n; i++) {   // de rest van f[] gaan vullen
    int max = 1;
    for (int j=1; j<i; j++) {
        if (a[j] <= a[i])
            max = Math.max(max, f[j] + 1);
    }
    f[i] = max;
}

int max = 0;                 // tweede fase: het maximum van de f[] waarden opleveren
for (int i=0; i<=n; i++) {
       max = Math.max(max, f[i]);
}
return max;
```

De eerste fase heeft 2 geneste for-lussen, die duurt $O(n^2)$, het maximum opleveren in de tweede fase wordt daarmee ruim overschaduwd, dat duurt slechts $O(n)$-tijd.

Opleveren van de LMSD zelf

Het algoritme geeft nu de lengte van de LMSD terug, wat leuk is, maar we wilden eigenlijk ook weten welke getallen er dan precies inzitten.

Ik zal het hier niet meer voordoen, maar we kunnen het algoritme eenvoudig uitbreiden op een manier die heel typisch is voor constructieversies van DP's : Hou bij iedere keuze die je maakt bij welke keuze dat was, en reconstrueer het antwoord dan achterstevoren.

In dit geval komt dat er op neer dat je een extra array ter grootte van f aanmaakt: bijvoorbeeld $b(i)$ die aangeeft welke $f(j)$ het maximum was dat je voor $f(i)$ koos. Als je dan aan het eind de grootste waarde van f hebt gevonden: $f(m)$, dan weet je ten eerste dat a_m het laatste getal van de LMSD is, en dat $m' = b(m)$ het op-een-na-laatste getal is, en $b(m')$ het getal daarvoor, enz. totdat je hele deelrij af is.