Recursie en dynamisch programmeren

Bij puur recursieve functies kun je het risico lopen dat je veel werk dubbel doet. Dynamisch programmeren kan dan een alternatief zijn.

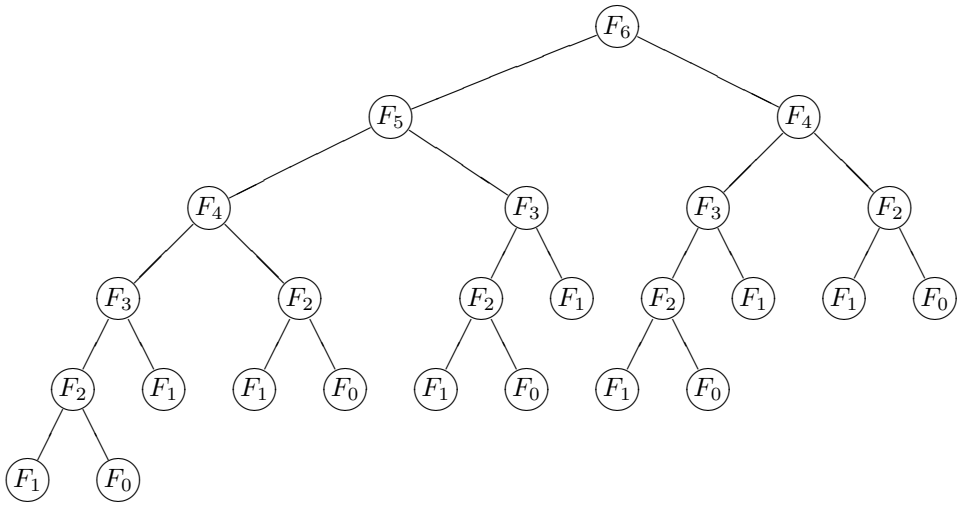
Fibonacci getallen

De Fibonacci getallen (gebaseerd op het fokgedrag van konijnen) zijn als volgt gedefinieerd: $F_0 = F_1 = 1$, en $F_n = F_{n-1} + F_{n-2}$ voor $n \geq 2$.

De volgende functie berekent op recursieve wijze de Fibonacci getallen:

```
int Fibonacci (N)
{
     if (N==0 or N==1)
     then return 1;
     else  return (Fibonacci(N-1) + Fibonacci(N-2));
     fi
}
```

Als we met de deze functie bijvoorbeeld F_6 willen uitrekenen, geeft de volgende boom de verschillende (recursieve) aanroepen van de functie weer:



Het is duidelijk dat veel berekeningen dubbel worden uitgevoerd: F_4 wordt twee keer uitgerekend, F_3 drie keer, F_2 vijf keer. Dynamisch programmeren is efficiciënter. We berekenen 'van onderop' alle Fibonacci getallen die we nodig hebben en slaan die op in een array `Fib` (dat groot genoeg is). Bijvoorbeeld met de volgende code:

```
int Fibonacci (N)
{
     Fib[0] = 1;
     Fib[1] = 1;
     for m=2 to N
     do   Fib[m] = Fib[m-1] + Fib[m-2];
     od
     return Fib[N];
}
```

Rijtjes corresponderende haakjes

Een rijtje van N corresponderende haakjesparen bevat N openhaakjes en N sluithaakjes, waarbij ieder openhaakje eerder komt dan het corresponderende sluithaakje.

Als $N = 1$, is de enige mogelijkheid: `()`.

Als $N = 2$, zijn de enige mogelijkheden: `()()`, `(())`.

Als $N = 3$, zijn de enige mogelijkheden: `()()()`, `()(())`, `(())()`, `(()())`, `((()))`.

Laat R_N het aantal mogelijke rijtjes van N corresponderende haakjesparen zijn. Dan is $R_1 = 1$, $R_2 = 2$ en $R_3 = 5$. Verder is $R_0 = 1$: er is precies één rijtje van 0 haakjesparen: het lege rijtje.

Om R_N voor algemene N te berekenen, bedenken we dat een rijtje altijd met een openhaakje begint. Tussen dit openhaakje en het corresponderende sluithaakje staat een aantal andere haakjes. Om precies te zijn: er staat een rijtje van i corresponderende haakjesparen, waarbij i gelijk kan zijn aan $0, 1, 2, \ldots, N-1$. Als er i haakjesparen tussen dit openhaakje en sluithaakje in staan, staan er na het sluithaakje nog $N-1-i$ haakjesparen:

$$(\underbrace{\ldots\ldots\ldots\ldots\ldots}_{i \text{ haakjesparen}}) \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots\ldots}_{N-1-i \text{ haakjesparen}}$$

Voor de i haakjesparen zijn R_i mogelijkheden, en voor de $N-1-i$ haakjesparen zijn R_{N-1-i} mogelijkheden. Omdat al deze mogelijkheden gecombineerd kunnen worden zijn er voor een bepaalde i in totaal $R_i \times R_{N-1-i}$ mogelijkheden. Nu kunnen we dus ook nog kiezen wat de waarde i wordt: $0, 1, 2, \ldots, N-1$. Daarmee wordt het totaal aantal mogelijkheden:

$$R_N = R_0 \times R_{N-1} + R_1 \times R_{N-2} + R_2 \times R_{N-3} + \cdots + R_{N-2} \times R_1 + R_{N-1} \times R_0.$$

Deze formule kunnen we gebruiken om R_N recursief te berekenen, bijvoorbeeld met de volgende functie:

```
int Rijtjes (N)
{    int Som;
     if (N==0)
     then return 1;
     else  Som = 0;
          for i=0 to N-1
          do   Som += Rijtjes(i) × Rijtjes(N-1-i);
          od
          return Som;
     fi
}
```

Met deze recursieve functie worden veel waardes meerdere keren berekend. Teken zelf maar eens de boom van (recursieve) aanroepen voor het berekenen van R_5. Ook hier kunnen we het dubbele werk voorkomen met dynamisch programmeren. We berekenen 'van onderop' alle waardes van R_i die we nodig hebben en slaan die op in een array R (dat groot genoeg is). Bijvoorbeeld met de volgende code:

```
int Rijtjes (N)
{
     R[0] = 1;
     for m=1 to N
     do   Som = 0;
          for i=0 to m-1
          do   Som += R[i] × R[m-1-i];
          od
          R[m] = Som;
     od
     return R[N];
}
```

Op deze wijze *berekenen* we iedere R_i precies één keer, ook al *gebruiken* we dezelfde waarde meerdere keren.

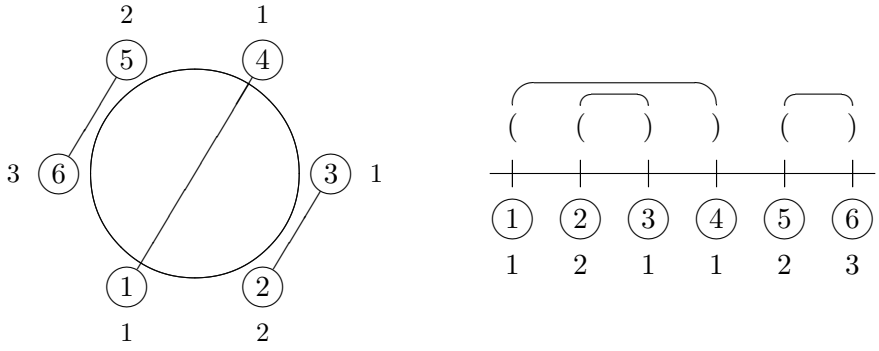
Beiers bierfestijn

Voor de situatieschets, zie de betreffende programmeeropgave.

Sleutel tot de oplossing van de opgave is dat we inzien dat we de ronde tafel op een willekeurige plek mogen 'openvouwen'. We maken er dan een *rij* van P professoren van, met nummers $1, 2, \ldots, P$.

Twee professoren i en j (met $i < j$ kunnen alleen met elkaar toasten als er tussen hen in een even aantal professoren zit. Immers: anders kunnen de tussenliggende professoren niet allemaal toasten zonder de armen van i en j te kruisen. Als er $2k$ professoren (met $k \geq 0$) tussen i en j in zitten, is $j = i+1+2k$.

Twee professoren i en j (met $i < j$) kunnen nu worden opgevat worden als twee corresponderende haakjes. Een geldige toast (zonder kruisende armen) komt dan overeen met een rijtje van $N = \frac{1}{2}P$ corresponderende haakjesparen. In het volgende voorbeeld met $P = 6$ professoren en een rijtje met $N = \frac{1}{2}P = 3$ corresponderende haakjesparen, is dit idee uitgewerkt. Kleine cirkels met cijfer 1,2,...,6 staan voor de zes professoren. Het cijfer bij elke professor staat voor zijn biermerk.



Bij ieder rijtje haakjesparen kunnen we nu de score bepalen, d.w.z.: hoeveel toasts er plaatsvinden met gelijke biermerken. In het bovenstaande voorbeeld is dat 1. Het gaat er nu om dat we de maximale score bepalen over alle mogelijke haakjesparen. In ons voorbeeld is dat maximum 2.

Laat nu i en j met $j = i + 1 + 2k$ voor zekere $k \geq 0$ twee professoren zijn, met een even aantal professoren ($2k$ om precies te zijn) ertussenin, en laat

$$S(i,j) = \begin{cases} 1 & \text{als } i \text{ en } j \text{ hetzelfde biermerk drinken} \\ 0 & \text{als } i \text{ en } j \text{ niet hetzelfde biermerk drinken} \end{cases}$$

Laat verder $\texttt{BestScore}(i,j)$ de best mogelijke score zijn die de professoren $i, i+1, \ldots, j$ kunnen bereiken als zij met elkaar moeten toasten. Als $j = i + 1$ (d.w.z.: $k = 0$, dan is er maar één mogelijkheid: i en j moeten met elkaar toasten, en $\texttt{BestScore}(i,j) = S(i,j)$. Uiteindelijk gaat het ons natuurlijk om $\texttt{BestScore}(1,P)$: om de beste score als alle P professoren een toast uitbrengen.

We gaan nu $\texttt{BestScore}(i,j)$ uitrekenen voor algemene i en j met $j = i + 1 + 2k$ en $k \geq 0$. Als de professoren $i, i+1, \ldots, j$ met elkaar moeten toasten, zal professor i met een professor m moeten toasten, waarbij m gelijk is aan een van de professoren $i+1, i+3, i+5, \ldots, i+1+2k = j$. Als we nu de beste m vinden, en vervolgens ook de professoren tussen i en m op de beste manier laten toasten, en de professoren tussen m en j op de beste manier laten toasten, dan hebben we $\texttt{BestScore}(i,j)$ te pakken:

$$\texttt{BestScore}(i,j) = \max_{m=i+1,i+3,\ldots,j} \Big(S(i,m) + \texttt{BestScore}(i+1, m-1) + \texttt{BestScore}(m+1, j) \Big). \quad (1)$$

Hierin definiëren we $\texttt{BestScore}(r,s) = 0$ als $r > s$ (als we 0 professoren hebben, hebben we tenslotte 0 toasts met gelijk biermerk).

We kunnen nu formule (1) gebruiken om $\texttt{BestScore}(1,P)$ recursief te berekenen. Hierbij zullen we echter weer veel werk dubbel gaan doen. Dat kunnen we weer voorkomen met dynamisch programmeren. We berekenen eerst $\texttt{BestScore}(i,j)$ voor i en j die dicht bij elkaar liggen, en bouwen dat geleidelijk op tot grotere afstanden tussen i en j, en uiteindelijk tot $\texttt{BestScore}(1,P)$.

Voor het geval dat $P = 8$ berekenen we dan achtereenvolgens (waarbij `BS`(i, j) gewoon `BestScore`(i, j) betekent):

`BS`$(1, 2)$, `BS`$(2, 3)$, `BS`$(3, 4)$, `BS`$(4, 5)$, `BS`$(5, 6)$, `BS`$(6, 7)$, `BS`$(7, 8)$,
`BS`$(1, 4)$, `BS`$(2, 5)$, `BS`$(3, 6)$, `BS`$(4, 7)$, `BS`$(5, 8)$,
`BS`$(1, 6)$, `BS`$(2, 7)$, `BS`$(3, 8)$,
`BS`$(1, 8)$.

Om bijvoorbeeld `BestScore`$(1, 6)$ te berekenen met behulp van formule (1), maken we gebruik van `BestScore`$(3, 6)$ (als $m = 2$) `BestScore`$(2, 3)$ en `BestScore`$(5, 6)$ (als $m = 4$) en `BestScore`$(2, 5)$ (als $m = 6$).

Probeer dit nu eens helemaal te implementeren!

Combinatie recursie-dynamisch programmeren

Bij puur dynamisch programmeren los je het oorspronkelijke probleem op door 'van onderop' alle mogelijke deelproblemen op te lossen. Dit kan als nadeel hebben dat je meer rekenwerk doet dan nodig is. Sommige deelproblemen heb je misschien helemaal niet nodig voor de oplossing van het oorspronkelijke probleem.

Wat je dan ook kunt doen, is alsnog een recursieve implementatie gebruiken, waarbij je iedere waarde die je berekent, opslaat. Hierdoor hoef je een waarde geen tweede keer te berekenen.