



Uitwerkingen

1. We passen het dynamic programming algoritme toe op de situatie uit de slides met $n = 3$. De prijslijst:

lengte i	1	2	3	4
prijs p_i	1	5	8	9

Toepassen van het algoritme levert een array b met op rank i de beste prijs voor een stuk hout ter lengte i . Dan: $b[0] = 0$, een initialisatie- of start-stap. Vervolgens wordt in de buitenste j -loop de waarde $b[j]$ berekend.

$$\begin{array}{l} j = 1 \quad i = 1 \quad p[1] + b[0] = 1 + 0 = 1 \\ \qquad \qquad \qquad b[1] := 1 \\ j = 2 \quad i = 1 \quad p[1] + b[1] = 1 + 1 = 2 \\ \qquad \qquad \qquad i = 2 \quad p[2] + b[0] = 5 + 0 = 5 \\ \qquad \qquad \qquad \qquad \qquad b[2] := 5 \\ j = 3 \quad i = 1 \quad p[1] + b[2] = 1 + 5 = 6 \\ \qquad \qquad \qquad i = 2 \quad p[2] + b[1] = 5 + 1 = 6 \\ \qquad \qquad \qquad i = 3 \quad p[3] + b[0] = 8 + 0 = 8 \\ \qquad \qquad \qquad \qquad \qquad b[3] := 8 \end{array}$$

Dus: de beste oplossing voor lengte $n = 3$ is 8.

2. Stel we hebben de volgende prijstabel:

lengte i	1	2	3
prijs p_i	1	5	7
$\frac{p_i}{i}$	1	$\frac{5}{2}$	$\frac{7}{3}$

Een greedy keuze voor steeds de hoogste dichtheid levert bij een stuk van lengte 3: stuk van lengte 2 met waarde 5, en dan een stuk van lengte 1 met waarde 1, dus totaal 6. Maar de beste keus is duidelijk een stuk van lengte 3 met waarde 7. Dus een greedy keuze voor een stuk met de hoogste dichtheid levert hier geen optimale oplossing.

3. Idee: we houden behalve het array b met de optimale suboplossingen nog een array c bij met het aantal hakken voor die suboplossing. Ook het array c moet steeds geupdated worden. Bij het berekenen van de nieuwe benefit moeten de hakkosten ook in rekening gebracht worden.

4. We passen het algoritme van Kadane toe op het array $[-3, 10, -8, 9]$.

$$B[0] = \max(-3, 0) = 0.$$

$$\begin{aligned} r = 1 \quad B[1] &= \max(0, B[0] + A[1]) = 10 \\ r = 2 \quad B[2] &= \max(0, B[1] + A[2]) = 2 \\ r = 3 \quad B[3] &= \max(0, B[2] + A[3]) = 11 \end{aligned}$$

De maximumwaarde van het beste subarray is 11. Het subarray dat deze waarde levert is $[10, 8, 9]$.

5. Bij het fractional knapsack probleem met

	b	w	$\frac{b}{w}$
s_1	3	3	1
s_2	2	1	2
s_3	2	1	2

en maximaal totaalgewicht $W = 3$ leidt een greedy keuze voor de grootste waarde tot het kiezen van s_1 met benefit 3, en niet tot de optimale oplossing $\{s_2, s_3\}$ met benefit 4.

6. Bij het fractional knapsack probleem met

	b	w	$\frac{b}{w}$
s_1	1	1	1
s_2	4	2	2

en maximaal totaalgewicht $W = 2$ leidt een greedy keuze voor het laagste gewicht tot het kiezen van s_1 met benefit 1 en niet tot de optimale oplossing $\{s_2\}$ met benefit 4.

7. We bekijken de verzameling $S = \{s_1, \dots, s_7\}$ met de volgende informatie over benefit, gewicht, en het quotient benefit/gewicht:

	b	w	$\frac{b}{w}$
s_1	12	4	3
s_2	10	6	$1\frac{2}{3}$
s_3	8	5	$1\frac{3}{5}$
s_4	11	7	$1\frac{4}{7}$
s_5	14	3	$4\frac{2}{3}$
s_6	7	1	7
s_7	9	6	$1\frac{1}{2}$

De greedy volgorde is: $s_6, s_5, s_1, s_2, s_3, s_4, s_7$. Het maximale totaalgewicht is gegeven als $W = 18$. We kiezen in de greedy volgorde, dit levert de volgende totale benefit:

$$7 \cdot \frac{1}{1} + 14 \cdot \frac{3}{3} + 12 \cdot \frac{4}{4} + 10 \cdot \frac{6}{6} + 8 \cdot \frac{4}{5} = 47\frac{2}{5}$$

8. We passen het algoritme voor knapsack-0-1 toe op de gegevens uit de vorige opgave.

De verzameling S bevat:

$$(12, 4), (10, 6), (8, 5), (11, 7), (14, 3), (7, 1), (9, 6)$$

en we hanteren die volgorde. Dus bijvoorbeeld $S_1 = \{(12, 4)\}$. De volgende array geeft $B[k, w]$ voor $k = 0, \dots, 5$ en $w = 0, \dots, 18$.

$k \setminus w$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
2	0	0	0	0	12	12	12	12	12	12	22	22	22	22	22	22	22	22	22
3	0	0	0	0	12	12	12	12	12	20	22	22	22	22	22	30	30	30	30
4	0	0	0	0	12	12	12	12	12	20	22	23	23	23	23	30	30	33	33
5	0	0	0	14	14	14	14	26	26	26	26	26	34	36	37	37	37	37	44
6	0	7	7	14	21	21	21	26	33	33	33	33	34	41	43	44	44	44	44
7	0	7	7	14	21	21	21	26	33	33	33	33	34	41	43	44	44	44	44

(Waarom werkt het algoritme niet correct als je w laat stijgen ipv dalen?)

9. We schrijven dit optimalisatie-probleem als knapsack-probleem. We representeren de verzameling biedingen als $S = \{s_1, \dots, s_m\}$ waarbij een bod s_i een benefit e_i heeft, en een gewicht k_i . Dus in tabel-vorm:

	b	w	$\frac{b}{w}$
s_1	e_1	k_1	$\frac{e_1}{k_1}$
s_2	e_2	k_2	$\frac{e_2}{k_2}$
\vdots	\vdots	\vdots	\vdots
s_m	e_m	k_m	$\frac{e_m}{k_m}$

Het maximale totaalgewicht is $W = n$.

Het probleem is een fractional knapsack probleem als we ervan uitgaan dat voor elk bod s_i de bidder ook geïnteresseerd is in minder dan k_i dingen.

Het probleem is een knapsack01 probleem als we ervan uitgaan dat voor elk bod s_i de bidder òf precies k_i dingen wil of anders niets.

10. Opmerking: De tijdscomplexiteit van een implementatie van het algoritme zoals gegeven in het boek hangt af van de manier waarop de verzameling S wordt gerepresenteerd. Stel we representeren S als ongeordende lijst, dan is het n keer een element met de hoogste relatieve waarde zoeken in $\mathcal{O}(n^2)$. Maar met een handigere implementatie kan het beter, namelijk in $\mathcal{O}(n \log n)$, en daarom wordt gezegd dat dit algoritme ‘in $\mathcal{O}(n \log n)$ is’.

We representeren de verzameling S met een priority queue, waarbij de key van een item s_i zijn relatieve waarde $\frac{b_i}{w_i}$ is, en een hogere waarde grotere prioriteit heeft. In de initialiseer-fase van het algoritme wordt voor elk element s_i zijn relatieve waarde $\frac{b_i}{w_i}$ in de priority queue opgeslagen. Dat is n (aantal elementen) keer $\log n$ (toevoegen aan priority queue) dus in $\mathcal{O}(n \log n)$. In de while-loop wordt n (aantal elementen) keer een item uit de priority queue gehaald (in $\log n$), en verder zijn er alleen elementaire acties. De while-loop zit dus ook in $\mathcal{O}(n \log n)$.

Het hele algoritme zit dus in $\mathcal{O}(n \log n)$.

11. Een aangepast algoritme voor knapsack01:

Algorithm 01Knapsack(S, W):

Input: Set S of n items, such that item i has a positive benefit b_i and positive integer weight w_i ; positive integer maximum total weight W

Output: For $w = 0, \dots, W$, maximum benefit $B[w]$ of a subset of S with total weight at most w

for $w := 0, \dots, W$ **do**

$B[w] := 0$

$S[w] := \emptyset$

for $k := 1, \dots, n$ **do**

for $w := W, \dots, w_k$ **do**

if $B[w - w_k] + b_k > B[w]$ **then**

$B[w] := B[w - w_k] + b_k$

$S[w] := S[w - w_k] \cup \{s_k\}$

We passen dit algoritme toe op de volgende verzameling S , met items s_i met benefit b_i en gewicht w_i :

	b	w
s_1	2	1
s_2	6	4
s_3	10	5
s_4	3	2

met maximaal totaalgewicht $W = 8$.

$k \setminus w$	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset	\emptyset
1	0	2	2	2	2	2	2	2	2
	\emptyset	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$
2	0	2	2	2	6	8	8	8	8
	\emptyset	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_2\}$	$\{s_1, s_2\}$	$\{s_1, s_2\}$	$\{s_1, s_2\}$	$\{s_1, s_2\}$
3	0	2	2	2	6	10	12	12	12
	\emptyset	$\{s_1\}$	$\{s_1\}$	$\{s_1\}$	$\{s_2\}$	$\{s_3\}$	$\{s_1, s_3\}$	$\{s_1, s_3\}$	$\{s_1, s_3\}$
4	0	2	3	5	6	10	12	13	15
	\emptyset	$\{s_1\}$	$\{s_4\}$	$\{s_1, s_4\}$	$\{s_2\}$	$\{s_3\}$	$\{s_1, s_3\}$	$\{s_3, s_4\}$	$\{s_1, s_3, s_4\}$

12. We gaan uit van een input-verzameling S met elementen s_1, \dots, s_n waarbij we steeds die volgorde van de items hanteren, en waarbij elk item s_i een benefit b_i en een gewicht w_i heeft.

Het fractional knapsack algoritme geeft als output x_1, \dots, x_n met elke x_i een geheel getal $0 \leq x_i \leq w_i$. Als het algoritme ‘geen fractions gebruikt’ dan geldt dus voor alle $i \in \{1, \dots, n\}$ dat of $x_i = 0$ of $x_i = w_i$.

Het knapsack01 algoritme geeft als output $B[0], \dots, B[W]$, met $B[w]$ de maximale benefit bij een keuze uit een subset van S met totaalgewicht ten hoogste w .

De algoritmes geven dus om te beginnen verschillende outputs.

Vind je in de twee gevallen wel dezelfde totale benefit? We schrijven B_{01} voor de totale benefit gevonden door het knapsack01 algoritme, en B_f voor de totale benefit gevonde door het fractional knapsack algoritme. Er geldt $B_{01} \geq B_f$ omdat B_{01} de beste waarde vindt voor een deelverzameling van S , en B_f komt *in dit geval* voort uit de keuze van een deelverzameling (we gebruiken geen fractions). Er geldt $B_f \geq B_{01}$ voor elke gevonden totale benefit bij gegeven verzameling S en totaalgewicht W , dus ook in dit specifiekere geval waarin B_f geen fractions nodig heeft.

NB we kunnen niet concluderen dat de deelverzamelingen van S die tot B_{01} en B_f leiden gelijk zijn.

13. We passen het machine-scheduling algoritme toe op de volgende verzameling taken, met begin- en eind-tijd gegeven:

$$\{(1, 2), (1, 3), (1, 4), (2, 5), (3, 7), (4, 9), (5, 6), (6, 8), (7, 9)\}$$

We hanteren de volgorde als boven, en schrijven de taken als t_1, \dots, t_9 . Dit leidt tot het volgende schema:

Op machine 1: $t_1 : [1, 2], t_4 : [2, 5], t_7 : [5, 6], t_8 : [6, 8]$.

Op machine 2: $t_2 : [1, 3], t_5 : [3, 7], t_9 : [7, 9]$.

Op machine 3: $t_3 : [1, 4], t_6 : [4, 9]$.

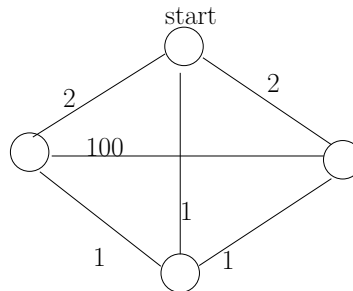
14. We gaan uit van n taken, t_1, \dots, t_n , waarbij elke t_i een start-tijd s_i en een eind-tijd f_i heeft. Het idee: op tijdstip t kies je uit de taken met $s_i \geq t$ een taak met kleinste f_i (dit is een greedy choice).

- init:
 - $t := 0$
 - sorteer de taken met de lexicografische ordening op start- en eind-tijd
- kies uit de taken met $s_i \geq t$ een taak t_j met kleinste eind-tijd f_j .
 - $t := f_j$

NB: je moet mogelijk alle taken met $s_i \geq t$ bekijken om er een met kleinste eind-tijd te vinden. Bijvoorbeeld bij $t = 0$, en schedule-bare taken $t_1 = (1, 4)$ en $t_2 = (2, 3)$ kies je voor t_2 .

Voorbeeld: van de taken $(1, 5), (2, 3), (3, 7), (4, 6), (6, 7)$ wordt gekozen de tweede, vierde, vijfde.

15. Het volgende voorbeeld laat zien dat de greedy methode bij het Traveling Salesman Probleem niet werkt.



16. Een algoritme voor het berekenen van de Fibonacci-getallen:

```

Algorithm fib( $n$ ):
  if  $n \leq 2$  then
    return 1
  else
    return (fib( $n - 1$ ) + fib( $n - 2$ ))
  
```

Wat is de tijdscomplexiteit van je algoritme?

17. Een dynamic programming algoritme voor het berekenen van de Fibonacci-getallen: ($F[0]$ is alleen een place-holder omdat arrays nu eenmaal bij index 0 beginnen.)

Algorithm fib(n):

```
new array  $F$ 
 $F[0] := 0$ 
 $F[1] := 1$ 
 $F[2] := 1$ 
for  $i := 3$  to  $n$  do
     $F[i] := F[i - 1] + F[i - 2]$ 
return  $F[n]$ 
```

Dit algoritme is in $\mathcal{O}(n)$; ruimtecomplexiteit ook in $\mathcal{O}(n)$.

Een alternatief:

Algorithm fib(n):

```
 $a := 1$ 
 $b := 1$ 
if  $n > 2$  then
     $c := a + b$ 
     $a := b$ 
     $b := c$ 
return  $b$ 
```

Dit algoritme is in $\mathcal{O}(n)$; ruimtecomplexiteit in $\mathcal{O}(1)$.