Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

6 November 2023

Algorithms & Data Structures Exercise sheet 7 HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 13 November 2023.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Exercise 7.1 *1-3 subset sums* (1 point).

Let $A[1, \dots, n]$ be an array containing n positive integers, and let $b \in \mathbb{N}$. We want to know if there exists a subset $I \subseteq \{1, 2, \dots, n\}$, together with multipliers $c_i \in \{1, 3\}$, $i \in I$ such that:

$$b = \sum_{i \in I} c_i \cdot A[i].$$

If this is possible, we say b is a 1-3 subset sum of A. For example, if $A = [16, 4, 2, 7, 11, 1]$ and $b = 61$, we could write $b = 3 \cdot 16 + 4 + 3 \cdot 2 + 3 \cdot 1$.

Describe a DP algorithm that, given an array $A[1, \dots, n]$ of positive integers, and a positive integer $b \in \mathbb{N}$ returns True if and only if b is a 1-3 subset sum of A. Your algorithm should have asymptotic runtime complexity at most $O(b \cdot n)$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps?
5. *Extracting the solution*: How can the solution be extracted once the table has been filled?
6. *Running time*: What is the running time of your solution?

Exercise 7.2 *Road trip.*

You are planning a road trip for your summer holidays. You want to start from city C_0, and follow the only road that goes to city C_n from there. On this road from C_0 to C_n, there are $n - 1$ other cities $C_1, \dots, C_{n-1}$ that you would be interested in visiting (all cities $C_1, \dots, C_{n-1}$ are on the road from C_0 to C_n). For each $0 \le i \le n$, the city C_i is at kilometer k_i of the road for some given $0 = k_0 < k_1 < \dots < k_{n-1} < k_n$.

You want to decide in which cities among $C_1, \dots, C_{n-1}$ you will make an additional stop (you will stop in C_0 and C_n anyway). However, you do not want to drive more than d kilometers without making a stop in some city, for some given value $d > 0$ (we assume that $k_i < k_{i-1} + d$ for all $i \in [n]$ so that this is satisfiable), and you also don't want to travel backwards (so from some city C_i you can only go forward to cities C_j with $j > i$).

(a) Provide a *dynamic programming* algorithm that computes the number of possible routes from C_0 to C_n that satisfy these conditions, i.e., the number of allowed subsets of stop-cities. Your algorithm should have $O(n^2)$ runtime.

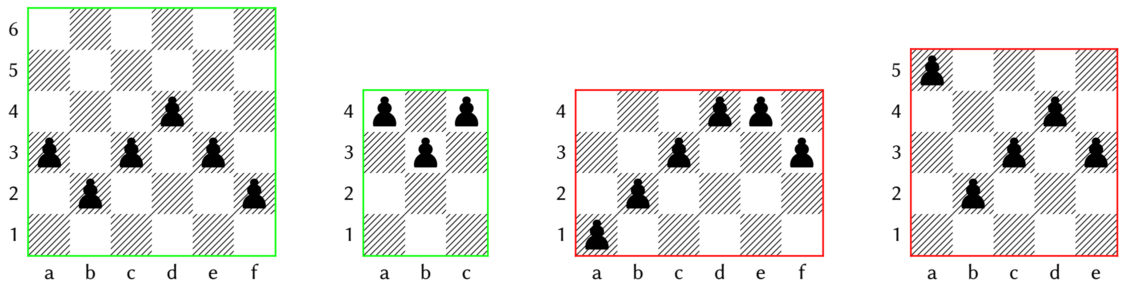
    Address the same six aspects as in Exercise 7.1 in your solution.

(b) If you know that $k_i > k_{i-1} + d/10$ for every $i \in [n]$, how can you turn the above algorithm into a linear time algorithm (i.e., an algorithm that has $O(n)$ runtime) ?

Exercise 7.3 *Safe pawn lines.*

On an $N \times M$ chessboard (N being the number of rows and M the number of columns), a *safe pawn line* is a set of M pawns with exactly one pawn per column of the chessboard, and such that every two pawns from adjacent columns are located diagonally to each other. When a pawn line is not safe, it is called *unsafe*.

The first two chessboards below show safe pawn lines, the latter two unsafe ones. The line on the third chessboard is unsafe because pawns d4 and e4 are located on the same row (rather than diagonally); the line on the fourth chessboard is unsafe because pawn a5 has no diagonal neighbor at all.



Describe a DP algorithm that, given $N, M > 0$, counts the number of safe pawn lines on an $N \times M$ chessboard. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(NM)$.

Exercise 7.4 *String counting* **(1 point)**.

Given a binary string $S \in \{0,1\}^n$ of length n, let $f(S)$ be the number of times "11" occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{“}\underline{11}10\underline{1}1\text{”}) = 3$ because the string contains three 1 that are followed by another 1 (underlined). Given n and k, the goal is to count the number of binary strings S of length n with $f(S) = k$.

Describe a DP algorithm that, given positive integers n and k with $k < n$, reports the required number. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(nk)$.

Hint: *Define a three dimensional DP table* $DP[1 \dots n][0 \dots k][0 \dots 1]$.

Hint: *The entry* $DP[i][j][l]$ *counts the number of strings of length* i *with* j *occurrences of* "11" *that end in* l *(where* $1 \leq i \leq n$, $0 \leq j \leq k$ *and* $0 \leq l \leq 1$*).*

Exercise 7.5 *Approximately solving knapsack* **(1 point)**.

Consider a knapsack problem with n items with values $v_i \in \mathbb{N}$ and weights $w_i \in \mathbb{N}$ for $i \in \{1, 2, \dots, n\}$, and weight limit $W \in \mathbb{N}$. Assume[1] that $w_{\max} := \max_{1 \leq i \leq n} w_i \leq W$ and also that $W \leq \sum_{i=1}^{n} w_i$.

For a set of items $I \subseteq \{1, 2, \dots, n\}$, we write $v(I) = \sum_{i \in I} v_i$ and $w(I) = \sum_{i \in I} w_i$ for the total value (resp. weight) of I. So, the solution to a knapsack problem is given by

$$\text{opt} := \max \{v(I) : I \subseteq \{1, 2, \dots, n\}, \quad w(I) \leq W\}.$$

Let $\varepsilon > 0$. We say a set of items $I \subseteq \{1, 2, \dots, n\}$ is *ε-feasible* if it violates the weight limit by at most a factor $1 + \varepsilon$, that is, if

$$w(I) \leq (1 + \varepsilon) \cdot W.$$

In this exercise, we construct an algorithm that finds an ε-feasible set I with $v(I) \geq \text{opt}$ in polynomial time in n and $1/\varepsilon$.

(a) Let $k \in \mathbb{N}$. Consider a 'rounded version' of the knapsack problem above obtained by replacing the weights w_i by:

$$\widetilde{w_i} := k \cdot \left\lfloor \frac{w_i}{k} \right\rfloor.$$

Recall the dynamical programming algorithm you have seen in the lecture which solves the knapsack problem in time $O(n \cdot W)$. Explain how to modify this algorithm to solve the 'rounded version' in time $O\big((W/k) \cdot n\big)$. For simplicity, you may assume that W is an integer multiple of k for this part.

Hint: *After rounding, all the* $\widetilde{w_i}$ *are integer multiples of* k*. Use this to reduce the number of table entries you have to compute in the dynamical programming algorithm.*

We write opt_k for the optimum solution value of the rounded problem in part (a). From now on, you may assume that your modified algorithm also returns a set I_k with $v(I_k) = \text{opt}_k$ and $\sum_{i \in I_k} \widetilde{w_i} \leq W$.

(b) Explain why $\text{opt}_k \geq \text{opt}$.

(c) Set $\varepsilon := (nk)/w_{\max}$. Show that $w(I_k) \leq (1 + \varepsilon) \cdot W$, that is, show that I_k is ε-feasible.

Hint: *Show that* $w_i \leq \widetilde{w_i} + k$ *for each* $i \in \{1, 2, \dots, n\}$*. We know that* $\sum_{i \in I_k} \widetilde{w_i} \leq W$*. Combine these facts to show that* $w(I_k) := \sum_{i \in I_k} w_i \leq W + n \cdot k$*. Finally, use the fact that* $W/w_{\max} \geq 1$ *to conclude your proof.*

(d) Now let $\varepsilon > 0$ be arbitrary. Describe an algorithm that finds an ε-feasible set I of items with $v(I) \geq \text{opt}$ in time $O(n^3/\varepsilon)$. Prove the runtime guarantee and correctness of your algorithm.

Hint: *Apply the algorithm of part (a) to the rounded problem with* $k = (w_{\max} \cdot \varepsilon)/n$*. For simplicity, you may assume that this* k *is an integer in your proof. Then, use the assumption that* $W \leq \sum_{i=1}^{n} w_i$ *(*$\leq n \cdot w_{\max}$*) to bound the runtime of the algorithm in terms of* n *and* $1/\varepsilon$*. Finally, use part (b) and part (c) to show correctness.*

[1] Why are these assumptions reasonable?

(e)* Let $\varepsilon = 1/100$. Give an example of a knapsack problem which has an ε-feasible solution I with value

$$v(I) = 2 \cdot \mathrm{opt},$$

Your example should satisfy $w_{\max} \leq W$ and $W \leq \sum_{i=1}^{n} w_i$.

Exercise 7.1 *1-3 subset sums* **(1 point).**

Let $A[1,\dots,n]$ be an array containing n positive integers, and let $b \in \mathbb{N}$. We want to know if there exists a subset $I \subseteq \{1,2,\dots,n\}$, together with multipliers $c_i \in \{1,3\}$, $i \in I$ such that:

$$b = \sum_{i\in I} c_i \cdot A[i].$$

If this is possible, we say b is a 1-3 subset sum of A. For example, if $A = [16,4,2,7,11,1]$ and $b = 61$, we could write $b = 3\cdot 16 + 4 + 3\cdot 2 + 3\cdot 1$.

Describe a DP algorithm that, given an array $A[1,\dots,n]$ of positive integers, and a positive integer $b \in \mathbb{N}$ returns True if and only if b is a 1-3 subset sum of A. Your algorithm should have asymptotic runtime complexity at most $\underline{O(b\cdot n)}$.

Very similar to subset sum.

1. *Dimensions of the DP table*: $DP[0\dots n][0\dots b]$ (Hint)
2. *Subproblems*: $DP[a][s]$ is True if, and only if, s can be written as a sum $\sum_{i\in I} c_i \cdot A[i]$ where $I \subseteq \{i : 1 \le i \le a\}$, and $c_i \in \{1,3\}$ for each $i \in I$.

$DP[a][s]$ is true $\iff \exists\ I \subseteq \{i : 1 \le i \le a\},\ c_i \in \{1,3\}\ \forall i \in I$

s.t. $s = \sum_{i\in I} c_i \cdot A[i]$

3. *Recursion*: DP can be computed recursively as follows:

$$\begin{aligned}
&DP[0][s] = \texttt{False} && 1 \le s \le b && (1) && (1)\\
&DP[a][0] = \texttt{True} && 0 \le a \le b && (2) && (2)\\
&DP[a][s] = \underbrace{DP[a-1][s]}_{(3.1)} \texttt{ or } \underbrace{DP[a-1][s-A[a]]}_{(3.2)} \texttt{ or } \underbrace{DP[a-1][s-3\cdot A[a]]}_{(3.3)} && 1 \le a \le n, && (3) && (3)\\
& && 1 \le s \le b.
\end{aligned}$$

Note that in equation (3), the entries '$DP[a-1][s-A[a]]$' and '$DP[a-1][s-3\cdot A[a]]$' might fall outside the range of the table, in which case we treat them as `False`.

→ These remarks are often made instead of doing a cumbersome case distinction.

(1) empty sum is defined as zero in AnD.

(2) 0 can always be written for $I = \emptyset$ (empty) sum

(3) very similar to subset sum

(3.1) If it was possible before, we can simply not take the element $c_i \cdot A[a]$ and still end up with s.

(3.2) We take the element $1 \cdot A[a]$.

(3.3) We take the element $3 \cdot A[a]$.

4. *Calculation order*: Following the recurrence relations above, we compute first by order of increasing a, and then by increasing order of s.
5. *Extracting the solution*: The solution can be found in $DP[n][b]$, by part 2.
6. *Running time*: The running time of the solution is $O(nb)$ as there are $(n+1) \cdot (b+1) = O(nb)$ entries in the table, each entry requires $O(1)$ time to compute, and we extract the solution in $O(1)$ time.

Exercise 7.4 *String counting* **(1 point).**

Given a binary string $S \in \{0,1\}^n$ of length n, let $f(S)$ be the number of times "11" occurs in the string, i.e. the number of times a 1 is followed by another 1. In particular, the occurrences do not need to be disjoint. For example $f(\text{"}\underline{1}\underline{1}1 0\underline{1}1\text{"}) = 3$ because the string contains three 1 that are followed by another 1 (underlined). Given n and k, the goal is to count the number of binary strings S of length n with $f(S) = k$.

Describe a DP algorithm that, given positive integers n and k with $k < n$, reports the required number. In your solution, address the same six aspects as in Exercise 7.1. Your solution should have complexity at most $O(nk)$.

(1) ***Hint:*** *Define a three dimensional DP table* $DP[1\dots n][0\dots k][0\dots 1]$.

Hint: *The entry* $DP[i][j][l]$ *counts the number of strings of length* i *with* j *occurrences of "11" that end in* l *(where* $1 \le i \le n, 0 \le j \le k$ *and* $0 \le l \le 1$*).*

Hints are <u>very</u> useful here.

(1) Why three dimensional ?

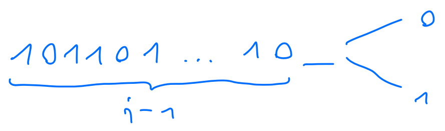
$\Longrightarrow$ how much information do we need ?

1. *Dimensions of the DP table*: $DP[1\dots n][0\dots k][0\dots 1]$.
2. *Subproblems*: The entry $DP[i][j][l]$ describes the number of strings of length i with j occurrences of "11" that end in l.

$DP[i][j][l] :=$ " # of strings of length i with j occurrences of "11" that end in $l \in \{0,1\}$ "

3. *Recursion*: The base cases for $i = 1$ are given by $DP[1][0][0] = 1$ (the string "0"), $DP[1][0][1] = 1$ (the string "1"), $DP[1][j][0] = 0$ and $DP[1][j][1] = 0$ for $1 \le j \le k$. The update rule is as follows: For $1 < i \le n$ and $0 \le j \le k$, to get a string of length i with j occurrences of "11" ending in 0, we can append "0" to a string of length $i-1$ with j occurrences of "11" ending in 0 or 1, which gives

$$DP[i][j][0] = DP[i-1][j][0] + DP[i-1][j][1].$$



To get a string of length i with j occurrences of "11" ending in 1, we can append "1" to a string of length $i-1$ with j occurrences of "11" ending in 0 or to a string of length $i-1$ with $j-1$ occurrences of "11" ending in 1 (if $j > 0$). Thus,

out of bounds check

$$DP[i][j][1] = \begin{cases} DP[i-1][j][0], & \text{if } j = 0 \\ DP[i-1][j][0] + DP[i-1][\underline{j-1}][1], & \text{if } j > 0. \end{cases}$$

$$\underbrace{101101 \ldots 11}_{i-1}\ \_ \quad \begin{cases} 0, & \text{then "11" occurrences stay the same} \\ 1, & \text{then } +1 \text{ "11" occurrence} \end{cases}$$

4. *Calculation order*: The entries can be calculated in order of increasing i. There is no interaction between entries with the same i, hence the order within the same value of i can be arbitrary (e.g. increasing in j and l).
5. *Extracting the solution*: The solution is $DP[n][k][0] + DP[n][k][1]$.
6. *Running time*: The running time of the solution is $O(nk)$ as there are $O(nk)$ entries in the table, each of which is processed in $O(1)$ time, and the solution is extracted in $O(1)$.

Exercise 7.5 *Approximately solving knapsack* **(1 point).**

Consider a knapsack problem with n items with values $v_i \in \mathbb{N}$ and weights $w_i \in \mathbb{N}$ for $i \in \{1, 2, \ldots, n\}$, and weight limit $W \in \mathbb{N}$. Assume[1] that $w_{\max} := \max_{1 \leq i \leq n} w_i \leq W$ and also that $W \leq \sum_{i=1}^{n} w_i$.

For a set of items $I \subseteq \{1, 2, \ldots, n\}$, we write $v(I) = \sum_{i \in I} v_i$ and $w(I) = \sum_{i \in I} w_i$ for the total value (resp. weight) of I. So, the solution to a knapsack problem is given by

$$\text{opt} := \max \{v(I) : I \subseteq \{1, 2, \ldots, n\}, \quad w(I) \leq W\}.$$

Let $\varepsilon > 0$. We say a set of items $I \subseteq \{1, 2, \ldots, n\}$ is ε-*feasible* if it violates the weight limit by at most a factor $1 + \varepsilon$, that is, if

$$w(I) \leq (1 + \varepsilon) \cdot W.$$

In this exercise, we construct an algorithm that finds an ε-feasible set I with $v(I) \geq \text{opt}$ in polynomial time in n and $1/\varepsilon$.

(a) Let $k \in \mathbb{N}$. Consider a 'rounded version' of the knapsack problem above obtained by replacing the weights w_i by:

$$\widetilde{w_i} := k \cdot \left\lfloor \frac{w_i}{k} \right\rfloor.$$

Recall the dynamical programming algorithm you have seen in the lecture which solves the knapsack problem in time $O(n \cdot W)$. Explain how to modify this algorithm to solve the 'rounded version' in time $O\big((W/k) \cdot n\big)$. For simplicity, you may assume that W is an integer multiple of k for this part.

Hint: *After rounding, all the $\widetilde{w_i}$ are integer multiples of k. Use this to reduce the number of table entries you have to compute in the dynamical programming algorithm.*

We use the Knapsack algorithm as shown in the lecture, with the difference being that

$\forall \tilde{w}_i,\ i \in [n]$: $\tilde{w}_i$ is a multiple of k

Now notice that we fill the table $MW[0 \ldots n][0 \ldots W]$, but in the range $0 \ldots W$ we only consider multiples of k, thus $MW[0 \ldots n][0 \ldots \lfloor \frac{W}{k} \rfloor]$. Since $\lfloor \frac{W}{k} \rfloor \leq \frac{W}{k}$ and every cell is in constant time we get $O((W/k)\, n)$ as the runtime.

The dynamical programming algorithm from the lecture fills out a table $\mathrm{MW}[0,\dots,n][0,\dots,W]$ with entries given by

$$\mathrm{MW}(i,w) := \text{‘largest value } v(I) \text{ one can obtain with } I \subseteq \{1,2,\dots,i\} \text{ and } w(I) \le w\text{’}.$$

This table is filled using the recursion:

$$\mathrm{MW}(i,w) = \max\{\mathrm{MW}(i-1,w),\ v_i + \mathrm{MW}(\overset{i-1}{\cancel{i}}, w - w_i)\}$$

Note that after rounding, all the $\widetilde{w_i}$ are integer multiples of k. Therefore, in the rounded problem, it suffices to only compute the table entries where the index w is an integer multiple of k. This leads to $(W/k+1)\cdot(n+1)$ table entries in total, each of which takes constant time to compute, meaning runtime $O\big((W/k)\cdot n\big)$ in total.

We write opt_k for the optimum solution value of the rounded problem in part (a). From now on, you may assume that your modified algorithm also returns a set I_k with $v(I_k) = \mathrm{opt}_k$ and $\sum_{i\in I_k} \widetilde{w_i} \le W$.

(b) Explain why $\mathrm{opt}_k \ge \mathrm{opt}$.

Since $\tilde{w}_i = k \cdot \left\lfloor \frac{w_i}{k} \right\rfloor \le k \cdot \frac{w_i}{k} = w_i$, for any set I s.t. $w(I) \le W$ we also have $\sum_{i \in I} \tilde{w}_i \le W$.

We have $\widetilde{w_i} \le w_i$ for each $i \in \{1,2\dots,n\}$. Therefore, any set I of items with $w(I) \le W$ also satisfies $\sum_{i\in I} \widetilde{w_i} \le W$. Since the values of each item are the same in the rounded problem, this implies that $\mathrm{opt}_k \ge \mathrm{opt}$.

(c) Set $\varepsilon := (nk)/w_{\max}$. Show that $w(I_k) \leq (1+\varepsilon)\cdot W$, that is, show that I_k is ε-feasible.

Hint: *Show that $w_i \leq \widetilde{w_i} + k$ for each $i \in \{1,2,\dots,n\}$. We know that $\sum_{i\in I_k} \widetilde{w_i} \leq W$. Combine these facts to show that $w(I_k) := \sum_{i\in I_k} w_i \leq W + n\cdot k$. Finally, use the fact that $W/w_{\max} \geq 1$ to conclude your proof.*

We have

$$\left(\frac{w_i}{k}\right) - 1 \leq \left\lfloor\frac{w_i}{k}\right\rfloor \iff w_i - k \leq \tilde{w}_i \qquad (1)$$

Now notice that

$$w(I_k) := \sum_{i\in I_k} w_i \overset{(1)}{\leq} \sum_{i\in I_k} (\tilde{w}_i + k)$$

$$= \sum_{i\in I_k} \tilde{w}_i + \sum_{i\in I_k} k$$

$$\overset{(i)}{\leq} W + nk$$

$$\overset{(ii)}{\leq} W + \frac{W}{w_{max}} nk$$

$$= W\left(1 + \underbrace{\frac{nk}{w_{max}}}_{\varepsilon}\right)$$

(i) $\sum_{i\in I_k} \tilde{w}_i \leq W$ follows from (b) and $I_k \subseteq \{1,\dots,n\}$ $\Rightarrow |I_k| \leq n$

(ii) we used $W/w_{max} \geq 1$

(d) Now let $\varepsilon > 0$ be arbitrary. Describe an algorithm that finds an ε-feasible set I of items with $v(I) \geq \text{opt}$ in time $O(n^3/\varepsilon)$. Prove the runtime guarantee and correctness of your algorithm.

Hint: *Apply the algorithm of part (a) to the rounded problem with $k = (w_{\max} \cdot \varepsilon)/n$. For simplicity, you may assume that this k is an integer in your proof. Then, use the assumption that $W \leq \sum_{i=1}^n w_i$ $(\leq n \cdot w_{\max})$ to bound the runtime of the algorithm in terms of n and $1/\varepsilon$. Finally, use part (b) and part (c) to show correctness.*

Let $k = (w_{max} \cdot \varepsilon) / n \overset{(*)}{\in} \mathbb{N}$. $\quad (*)$ assumption hint

We execute the modified knapsack algorithm from (a) with $\tilde{w}_i = k \lfloor \frac{w_i}{k} \rfloor \quad \forall i \in [n]$.

For runtime:

$$\overset{(a)}{\Longrightarrow} (W/k) \cdot n = \frac{W \cdot n}{w_{max} \cdot \varepsilon} n = \frac{n^2}{\varepsilon} \cdot \frac{W}{w_{max}}$$

$$\overset{(i)}{\leq} \frac{n^2}{\varepsilon} \cdot \frac{n \cdot w_{max}}{w_{max}} = \frac{n^3}{\varepsilon} \leq O\left(\frac{n^3}{\varepsilon}\right)$$

where we used $W \leq n \cdot w_{max}$ at (i).

For correctness, we show that the I_k returned from (a) for our k is indeed ε-feasible, i.e. $w(I_k) \leq (1+\varepsilon)W$ and $v(I_k) \geq opt$.

By part (b) we have $v(I_k) \geq opt$.

Since $k = (w_{max} \cdot \varepsilon)/n \Longrightarrow \varepsilon = (nk)/w_{max}$, by part (c) we have $w(I_k) \leq (1+\varepsilon) W$

(e)* Let $\varepsilon = 1/100$. Give an example of a knapsack problem which has an ε-feasible solution I with value

$$v(I) = 2 \cdot \text{opt},$$

Your example should satisfy $w_{\max} \leq W$ and $W \leq \sum_{i=1}^{n} w_i$.

Solution:

Consider the knapsack problem with two items, $v_1 = v_2 = w_1 = w_2 = 101$ and $W = 200$. Then, $\text{opt} = 101$ as we can only take one item. However, taking both items yields a solution which violates the weight limit by only a factor $1/100$ (as $(202 - 200)/200 = 1/100$). Thus there is an ε-feasible solution with value $202 = 2 \cdot \text{opt}$.