# Report on Approximation Schemes for 0-1 Knapsack

**Seminar Advanced Algorithms and Data Structures**

Jan Wiegner

November 4, 2018

This is a report on the conference paper *Approximation Schemes for 0-1 Knapsack*[1] by Timothy M. Chan, and is heavily based on it. Additional explanations are mostly based on the Master's Thesis of Donguk Rhee[2].

## Problem Introduction

Knapsack problems are a class of optimization problems, where we are given a set of $n$ items. The items are indexed by $i \in \{1, 2, ..., n\}$ and an item $i$ has weight $w_i$ and profit $p_i$. We write our set as: $I = \{(w_1, p_1), \cdots, (w_n, p_n)\}$. The 0-1 Knapsack Problem, which we will be focusing on is to select a subset of these items that maximizes the total profit under the constraint that total weight is bounded by a given input value, the maximum Capacity $W$. More formally, we are searching for:

$$\max_{s \subseteq I} \left\{ \sum_{(w_i,p_i) \in s} p_i \ : \ \sum_{(w_i,p_i) \in s} w_i \leq W \right\}$$

We will focus in this report on how to find the optimal profit, which we will note $S$, not directly on how to recover the subset witch attains i, which we note $x^*$. We consider a small example of the 0-1 Knapsack problem with input:
$I = \{(5, 6), (4, 4), (4, 4)\}$ and $W = 8$.
A greedy algorithm would pick the first element $(5, 6)$ as it has the biggest profit-density, i.e. the most profit per weight. This would give us a profit of 6. The optimal solution however would be to pick the second and third element, where we get profit 8, i.e. $x^* = \{(4, 4), (4, 4)\}$ and $S = 8$.

Knapsack problems are NP-hard. However, their simplicity attracted many theoreticians and practitioners to research these problems, dating back to 1897[3]. This has resulted in many published exact algorithms and approximation algorithms.

## Dynamic Programming

Before looking at approximation algorithms, we will first revisit the standard dynamic programming approach to get the exact value to the problem.

We will assume from now on that the 0-1 Knapsack Problem does not contain items with weight greater than $W$. Such items can be removed in $\mathcal{O}(n)$ time and do not change the solution, so it is safe to make this assumption.
We then define the more general function

$$f_I(x) := \max_{s \subseteq I} \left\{ \sum_{(w_i,p_i) \in s} p_i \ : \ \sum_{(w_i,p_i) \in s} w_i \leq x \right\}$$

This function fully specifies our problem, as the optimal value we want to find $x^*$ is $f_I(W)$. The standard dynamic programming approach consists of calculating $f_I(W)$ by calculating all the other values of $f_I$ in a dynamic programming table. In the table we use the following abbreviations:
$f_{i:j}(x) := f_{\{(w_i,p_i),\cdots,(w_j,p_j)\}}(x)$, where $f_{i:i}(x) := f_{\{(w_i,p_i)\}}(x)$.

| $x$ | $f_{1:1}(x)$ | $f_{1:2}(x)$ | $\cdots$ | $f_{1:n}(x)$ |
|---|---|---|---|---|
| 0 | $f_{1:1}(0)$ | $f_{1:2}(0)$ | $\cdots$ | $f_{1:n}(0)$ |
| 1 | $f_{1:1}(1)$ | $f_{1:2}(1)$ | $\cdots$ | $f_{1:n}(1)$ |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ddots$ | $\ldots$ |
| $W$ | $f_{1:1}(W)$ | $f_{1:2}(W)$ | $\cdots$ | $f_{1:n}(W)$ |

We fill the table from the left and the top. Each table entry is calculated by deciding if we get a higher profit by including the element with the highest index or ignoring it. More formally:

$$f_{1:i}(x) = \max \{ f_{1:i-1}(x - w_i) + p_i, \ f_{1:i-1}(x) \}$$

The solution can directly be read in the bottom right corner.
Each field in the table is calculated in constant time by looking at two other table entries and a constant number of operations. We have $nW$ table entries so this approach gives us an algorithm that runs in $\mathcal{O}(nW)$ time. It isn't necessary in general have to compute the whole table, but even using recursion with memorization (to not recalculate recursive calls we have already made) the worst case time complexity doesn't change. This isn't very satisfying, as the runtime now depends on the capacity $W$, but as it is a NP-hard problem, we weren't going to get something fully polynomial in $n$ anyway.

We can however change the algorithm like this: rather than considering the maximum profit for a given weight, we can try to find the minimum weight for a given profit. For this we define :

$$g_I(x) := \min_{s \subseteq I} \left\{ \sum_{(w_i,p_i) \in s} w_i \ : \ \sum_{(w_i,p_i) \in s} p_i \geq x \right\}$$

This means that our optimal profit $x^*$ will be the biggest $x$ for witch $g_I(x) \leq W$
Let P be the maximal profits $p_i$ in an instance of the 0-1 Knapsack Problem. Then the possible total profits are in the range of $[0; nP]$. We can then fill out the following dynamic programming table to obtain our desired value. In the table we use the following abbreviations:
$g_{i:j}(x) := g_{\{(w_i,p_i),\cdots,(w_j,p_j)\}}(x)$ , with $g_{i:i}(x) := g_{\{(w_i,p_i)\}}(x)$

| $x$ | $g_{1:1}(x)$ | $g_{1:2}(x)$ | $\cdots$ | $g_{1:n}(x)$ |
|---|---|---|---|---|
| 0 | $g_{1:1}(0)$ | $g_{1:2}(0)$ | $\cdots$ | $g_{1:n}(0)$ |
| 1 | $g_{1:1}(1)$ | $g_{1:2}(1)$ | $\cdots$ | $g_{1:n}(1)$ |
| $\ldots$ | $\ldots$ | $\ldots$ | $\ddots$ | $\ldots$ |
| $nP$ | $g_{1:1}(nP)$ | $g_{1:2}(nP)$ | $\cdots$ | $g_{1:n}(nP)$ |

Again we fill the table from the left and the top. Each table entry is calculated by deciding if we get a lower weight by including the element with the highest index or ignoring it. More formally:

$$g_{1:i}(x) = \min \{g_{1:i-1}(x - p_i) + w_i, \ g_{1:i-1}(x)\}$$

To obtain the solution we simply scan the table for the highest profit that has our given capacity $W$ (or less). Again, each field in the table is calculated in constant time, so this approach gives us an algorithm that runs in $\mathcal{O}(n^2 P)$ time.

## Approximation Algorithms[2]

Since the 0-1 Knapsack Problem is NP-hard, there is no algorithm with polynomial time in just $n$ assuming $NP \neq P$. However both of our solutions are dependent in time complexity on factors $W$ or $P$ that can be arbitrarily large compared to $n$, and are out of our control. There may however be applications where we need to have a runtime bound independent from the values $W$ or $P$, maybe because they are too large. The way we will achieve this is that we abandon the requirement of finding a completely optimal solution. We aim to find a solution that has small amount of error in a time that depends on how much error we allow. That is, we want an algorithm where we can freely compromise between time it takes and error it produces. This is the idea behind approximation algorithms.

### Simple FPTAS for Knapsack

In our case we have a maximization problem, so we can define an approximate solution $x'$ with profit $S'$ such that:
$S'(1+\varepsilon) \geq S$ where $\varepsilon > 0$ is a parameter that denotes our error.
First notice that our second dynamic programming algorithm is a polynomial algorithm for 0-1 Knapsack Problem if the profits are polynomially bounded in $n$. This means it is a polynomial algorithm if $P$ is polynomial in size compared to $n$. We can therefore think about scaling down the profits to make it always polynomial as follows:
$\forall i \in \{1, 2, ..., n\}, p'_i = \lfloor p_i/\delta \rfloor$ for some $\delta$

This incurs some error due to the floor operation and our goal is that the relative error on total profit is less than $\varepsilon$. The error for an item $i$ is $p_i - \delta p'_i = p_i - \delta\lfloor p_i/\delta \rfloor < \delta$. Let $x^*$ be an optimal solution and $S$ its total profit. We can assume $S \geq P$ as there are no items with weight greater than $W$. The total error would then be less than $\frac{\delta n}{S}$ as there are at most $n$ items in $x^*$.

Because we want $\frac{\delta n}{S} \leq \frac{\delta n}{P} \leq \varepsilon$, we pick $\delta = \frac{\varepsilon P}{n}$.

Using our dynamic programming algorithm with complexity $\mathcal{O}(n^2 P)$, we can solve the scaled down problem in

$$\mathcal{O}(n^2 \lfloor P/\delta \rfloor) = \mathcal{O}\left(n^2 \frac{n}{\varepsilon}\right) = \mathcal{O}\left(\frac{1}{\varepsilon} n^3\right)$$

To obtain the solution to our original problem we simply rescale up by $\delta$ the solution to our downscaled problem. We get an algorithm that has polynomial run-time in both $n$ and $1/\varepsilon$. In general we call this type of algorithm a **fully polynomial time approximation scheme** (FPTAS).

### Fully Polynomial Time Approximation Scheme

More formally: Suppose we have a maximization problem $\max_x$ z(x) as is the case of the 0-1 Knapsack Problem. Suppose that the size of the instance of the problem is $n$ and that $x^*$ is an optimal solution.

We define a **fully polynomial time approximation scheme** (FPTAS) is as an algorithm that takes an instance of the maximization problem and a parameter $\varepsilon > 0$ and has the following properties:

- It is guaranteed to produce a solution $x'$ such that $z(x')(1+\varepsilon) > z(x^*)$
- It runs in polynomial run-time in both $n$ and $1/\varepsilon$.

We will be focusing on FPTAS in this report. The 0-1 Knapsack Problem is important for being one of the first NP-hard problems shown to possess FPTAS.

## Preliminaries for more advanced methods[1]

### Input Simplifications

Before we continue, we notice that we can do two simplifications on the input:

- We can discard all items with $p_i \leq \frac{\varepsilon}{n} P$ (we still have $P = \max_j p_j$).

We can do this, because it removes less than $n$ items (all $\leq \frac{\varepsilon}{n} P$) and thus changes the optimal value by at most $\varepsilon P$. Because the optimal value must be bigger than $P$ this is an error of at most $1 + \varepsilon$ of the optimal value. We also have that the minimal profit is now at least $\frac{\varepsilon}{n} P$. This means we can assume that $\frac{\max_j p_j}{\min_j p_j} \leq \frac{P}{\frac{\varepsilon}{n} P} = \frac{n}{\varepsilon}$.

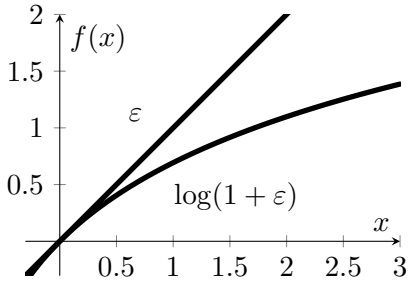
- We round all $p_i$ to a power of $1 + \varepsilon$.

This gives us a error of at most $1 + \varepsilon$ for each profit, so the total error is also at most $1 + \varepsilon$

Let $m$ be the number of different profits we have after this rounding. For the rounding we use powers of $1 + \varepsilon$ between $\log_{1+\varepsilon} \min_j p_j$ and $\log_{1+\varepsilon} \max_j p_j$, so we have:

$$m = \log_{1+\varepsilon} \max_j p_j - \log_{1+\varepsilon} \min_j p_j = \log_{1+\varepsilon} \frac{\max_j p_j}{\min_j p_j} \leq \log_{1+\varepsilon} \frac{n}{\varepsilon} = \frac{\log \frac{n}{\varepsilon}}{\log(1+\varepsilon)} \in \mathcal{O}\left(\frac{\log \frac{n}{\varepsilon}}{\varepsilon}\right)$$

(For $\varepsilon$ close to 0, $\varepsilon$ is the best linear approximation we can get for $\log(1 + \varepsilon)$.)



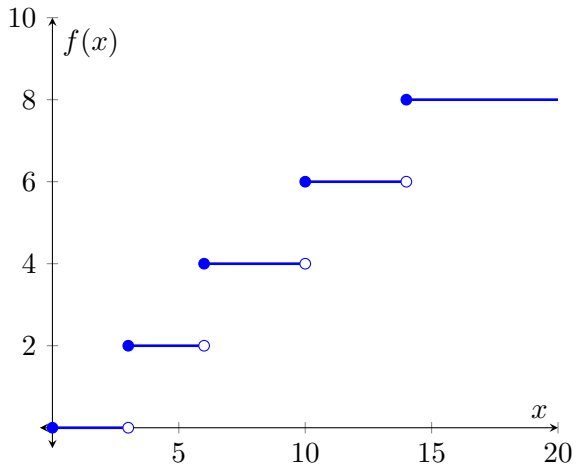

We therefore have at most $m \in \mathcal{O}(\frac{1}{\varepsilon} \log \frac{n}{\varepsilon})$ distinct profit values.

In the rest of our analysis we will use the $\widetilde{O}$ (read soft-O) notation, witch is like the $\mathcal{O}$ notation but hides polylogarithmic factors in $n$ and $\frac{1}{\varepsilon}$. That means $\mathcal{O}(n \log n + \frac{1}{\varepsilon} \log \frac{1}{\varepsilon}) = \widetilde{O}(n + \frac{1}{\varepsilon})$ and in our case $m \in \widetilde{O}(\frac{1}{\varepsilon})$.

## Pseudo-concave and p-uniform

In the previous section we saw a FPTAS that approximated $g_I(x)$. From now on will now look at (the outline of) how to make a FPTAS that approximates $f_I(x)$.

For this we adopt a "functional" approach in presenting our algorithms, which does not need explicit reference to dynamic programming, and makes analysis of approximation factors more elegant. Note that we can generalize $f_I$ so that it can accept *all* values for $x \in \mathbb{R}$ and that it is a monotone non-decreasing step function. This means it should look something like this:




Suppose we split our input set $I$ into two disjoint sets $I_a$ and $I_b$. Even if we had $f_{I_a}$ and $f_{I_b}$ we would have to combine them somehow to get $f_I$. To combine them we have to use $(max, +) - convolution$:

$$f_I(x) = (f_{I_a} \oplus f_{I_b})(x) = \max_{x' \in \mathbb{R}}(f_{I_a}(x') + f_{I_b}(x - x'))$$

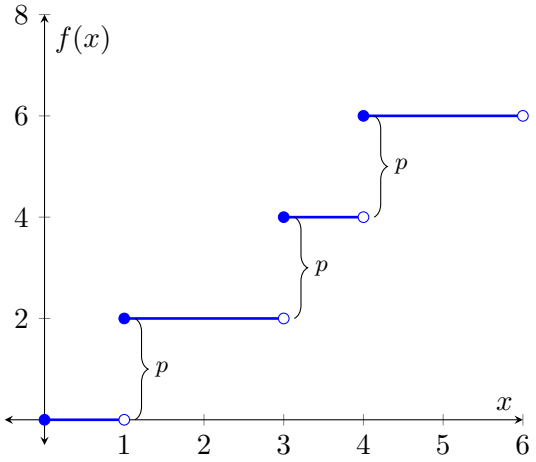
We have $f_I = f_{I_a} \oplus f_{I_b}$ (the operator $\oplus$ denotes the $(max, +) - convolution$)

We now look at the special case where all $p_i$'s in $I$ for a $f_I$ are equal to a common value $p$. If we first sort the items in non-decreasing order of $w_i$, the function $f_I$ is easy to compute with a greedy algorithm in time $\mathcal{O}(|I|) = \mathcal{O}(n)$. We have:

- The function values are $-\infty, 0, p, 2p, \cdots, np$
- The $x$-breakpoints are $0, w_1, w_1 + w_2, \cdots, w_1 + \cdots + w_n$

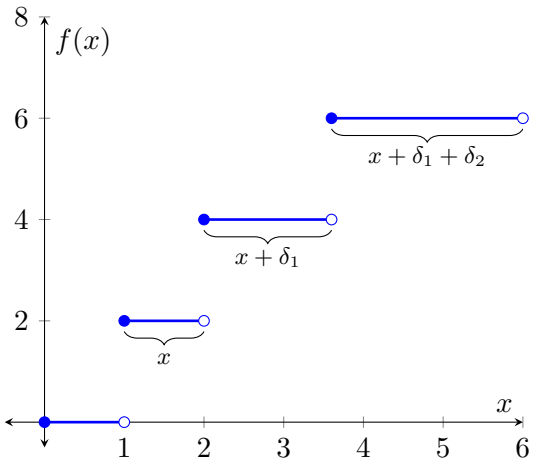
In general we say a step function is **p-uniform** if:

- The function values are of the form $-\infty, 0, p, 2p, \cdots, lp$ for some $l$



We say a **p-uniform** step function is **pseudo-concave** if:

- The sequence of differences of consecutive $x$-breakpoints is non-decreasing (i.e. the distance between breakpoints increases)




(in the graph $\delta_1, \delta_2 \geq 0$)

In the case where the $p_i$'s are all equal, $f_I$ is indeed **uniform** and **pseudo-concave**.

Coming back to our original problem, because we know that we have at most $m \in \widetilde{O}(\frac{1}{\varepsilon})$ distinct profits in our input set $I$. We can therefore split $I$ in $m$ disjoint sets where each only has one distinct $p_i$ value in the set. For each of these sets $I_k$ with index $k \in \{1, 2, ..., m\}$ we can compute $f_k = f_{I_k}$ using the greedy algorithm. (Here computing $f_k$ means calculating $f_k$ for all values.) The time this takes is :

- $\mathcal{O}(n \log n)$ to sort by profit and split into sets
- $\mathcal{O}(n \log n)$ to sort by weight and calculate all $f_i$'s greedily

The original problem is now reduced to calculating a $(1 + \mathcal{O}(\varepsilon))$ factor approximation of a monotone step function, which is the $\oplus$ of $m \in \widetilde{O}(\frac{1}{\varepsilon})$ uniform, pseudo-concave, monotone step functions.

## Outline of Algorithm with Exponent 5/2 [1]

We now want to calculate the $(max, +) - convolution$ of a set of $m$ functions (where each is a uniform, pseudo-concave, monotone step function). I won't talk here about how to efficiently calculate the $(max, +) - convolution$ of two functions, but will instead talk about a trick on how to compose these functions.

If $f$ and $h$ are monotone step functions with total complexity $l$ (i.e. sum of different function values is $l$), we assume that it is possible to compute a $(1+\mathcal{O}(\varepsilon))$ approximation of $f \oplus h$ in time:

(i) $\mathcal{O}(l) + \widetilde{O}((\frac{1}{\varepsilon})^2)$ in general

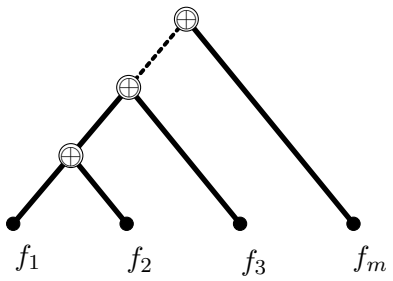
(ii) $\mathcal{O}(l) + \widetilde{O}(\frac{1}{\varepsilon})$ if $h$ is p-uniform and pseudo-concave

We need to compose the $m$ functions with the $(max, +) - convolution$.

## Sequential approach

Our first approach is to calculate the $(max, +) - convolutions$ sequentially. That is we calculate:

$$((((f_1 \oplus f_2) \oplus f_3) \oplus \cdots) \oplus f_m)$$




In each step we can use **(ii)** as at least one of the functions is p-uniform and pseudo-concave. The total time is $\widetilde{O}(n)+\widetilde{O}(m(\frac{1}{\varepsilon}))$. However, the approximation factor increases to $(1+O(\varepsilon))^m$. We can adjust $\varepsilon$ by a factor of $m$ to get back the error of $1+O(\varepsilon)$ because:

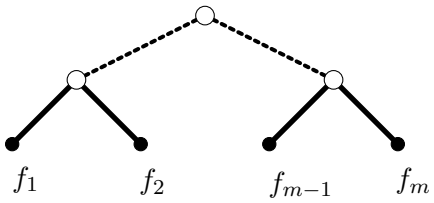
$$(1 + \frac{\varepsilon}{m})^m \leq e^{\varepsilon} \leq 1 + (e-1)\varepsilon \leq 1 + O(\varepsilon)$$

This increases the running time to:

$$\widetilde{O}(n) + \widetilde{O}\left(m \cdot \frac{1}{\varepsilon/m}\right) = \widetilde{O}(n) + \widetilde{O}\left(\frac{1}{\varepsilon}m^2\right) = \widetilde{O}\left(n + \left(\frac{1}{\varepsilon}\right)^3\right)$$

## Divide-and-conquer approach

We could think that using a divide-and-conquer approach might be better. To do this we split our set of functions into two and recursively calculate the convolution of each set and combine them with a final convolution.
That is, approximating $f_1 \oplus \cdots \oplus f_{m/2}$ and $f_{m/2+1} \oplus \cdots \oplus f_m$.



The recursion tree has $\mathcal{O}(m)$ nodes. According to **(i)** each has cost $\widetilde{O}((\frac{1}{\varepsilon})^2)$, and we have a total additional cost of $\widetilde{O}(n)$ for the leaves.

The approximation factor increases to $(1+O(\varepsilon))^{\log m}$. It is possible to argue that:

$$(1+O(\varepsilon))^{\log m} = 1 + O(\varepsilon \log m)$$

Therefore our error $1 + O(\varepsilon \log m)$ is too large by a factor of $\log m$. We can adjust $\varepsilon$ by a factor of $\log m$, which increases the running time only by polylogarithmic factors (hidden in $\widetilde{O}$). It gives us a time complexity:

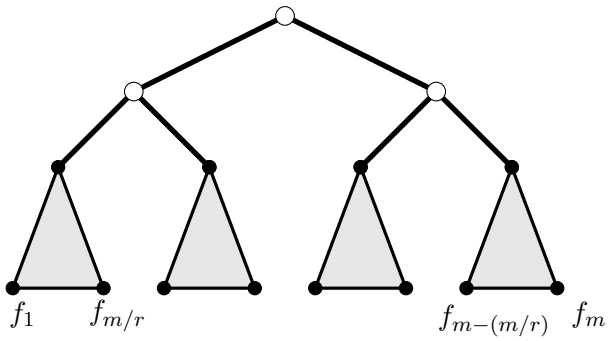
$$\widetilde{O}(n) + \widetilde{O}\left(\left(\frac{1}{\varepsilon}\right)^2 m\right) = \widetilde{O}\left(n + \left(\frac{1}{\varepsilon}\right)^3\right)$$

## Hybrid approach

Finally, we can also use a hybrid method. The intuition is to use the incremental method when $m$ is small and the divide and conquer method when $m$ is large.

Here is the precise algorithm:

1. Divide the set of given functions into $r$ subsets of $\frac{m}{r}$ functions, for a parameter $r$ to be specified later.
2. For each subset, approximate the $\oplus$ of its $\frac{m}{r}$ pseudo-concave functions using the sequential method (i.e. using **(ii)**).
3. Return an approximation of the $\oplus$ of the $r$ resulting functions, by using the divide-and-conquer method (i.e. using **(i)** ).




The total time is:

$$\widetilde{O}(n) + \widetilde{O}\left(r\frac{1}{\varepsilon}\left(\frac{m}{r}\right)^2 + (r-1)\left(\frac{1}{\varepsilon}\right)^2\right) = \widetilde{O}(n) + \widetilde{O}\left(\frac{1}{r}\left(\frac{1}{\varepsilon}\right)^3 + r\left(\frac{1}{\varepsilon}\right)^2\right)$$

We get the minimum if $r$ equal to a certain quantity $r = \lceil \sqrt{\varepsilon} m \rceil \in \widetilde{O}(\varepsilon^{-\frac{1}{2}})$. This is the case as the terms of the sum are linearly increasing and decreasing dependent on $r$, and both terms of the sum become equal with our value of $r$. Then we get:

$$\widetilde{O}(n) + \widetilde{O}\left(\varepsilon^{\frac{1}{2}}\left(\frac{1}{\varepsilon}\right)^3 + \varepsilon^{-\frac{1}{2}}\left(\frac{1}{\varepsilon}\right)^2\right) = \widetilde{O}\left(n + \left(\frac{1}{\varepsilon}\right)^{5/2}\right) = \widetilde{O}\left(n + \left(\frac{1}{\varepsilon}\right)^{2.5}\right)$$

## Additional insights

In overview in the best algorithm we:

- eliminiate too big and too small values
- roud the remaining inputs
- group them by same profit into $m$ groups
- calculate the problem for each group
- combine the groups cleverly using $(max, +) - convolution$

By loosing simplicity it is possible to obtain an algorithm in

$$\widetilde{O}\left(n + \left(\frac{1}{\varepsilon}\right)^{12/5}\right) = \widetilde{O}\left(n + \left(\frac{1}{\varepsilon}\right)^{2.4}\right)$$

This is achieved by avoiding some rounding operations and using a certain combinatorial/number-theoretic lemma. (It states that all numbers can be approximated well by integer multiples of a small set of values.)

We saw how to use $(max, +)-convolution$ to get a much better algorithm to compute a function that we before computed in a dynamic programming table. But this can be used in general for any function computed in a dynamic programming table witch computes the extremum of a sum.

The main open question remaining for 0-1 Knapsack FPTASs is whether the running time can be improved to near $\mathcal{O}(n + (\frac{1}{\varepsilon})^2)$.

I also want to mention that improved time bounds tell us how much accuracy we can guarantee while keeping near-linear running time: If there exists an algorithm with runtime in $\widetilde{O}(n + (\frac{1}{\varepsilon})^{12/5})$, it allows to give a $(1 + n^{-5/12})$-approximate solution in $\widetilde{O}(n)$ time.

$$\widetilde{O}\left(n + \left(n^{5/12}\right)^{12/5}\right) = \widetilde{O}(n)$$

As a final remark, we have described how to compute approximations of the optimal value, but not how to compute a corresponding subset of items. To output the subset, we can modify the algorithms to record extra information during execution, but this does not increase the runtime. All we really have to do is store the $argmax$ whenever we calculate a $max$.

## References

[1] Timothy M Chan. Approximation schemes for 0-1 knapsack. In *OASIcs-OpenAccess Series in Informatics*, volume 61. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[2] Donguk Rhee. Faster fully polynomial approximation schemes for knapsack problems, 2015.

[3] George B Mathews. On the partition of numbers. *Proceedings of the London Mathematical Society*, 1(1):486–490, 1896.