CS102: Introduction to Algorithms – FPTAS for 0/1 Knapsack

Lecture Notes

Contents

1 Introduction

In many real-world problems, we want to choose the best option among many possibilities. Some examples include:

- Packing items in a backpack to maximize value (Knapsack)
- Selecting a subset of numbers whose sum is closest to a target (Subset Sum)
- Dividing tasks among machines efficiently (Scheduling)

Some of these problems are **NP-hard**, meaning that finding the exact optimal solution may take *exponentially long time* for large inputs. But in many cases, an **approximate solution** that is "good enough" is acceptable.

—

2 Approximation Algorithms and FPTAS

2.1 What is an Approximation Algorithm?

An **approximation algorithm** provides a solution close to optimal:

$$(1 - \varepsilon) \cdot OPT \leq S \leq (1 + \varepsilon) \cdot OPT$$

where:

- S = solution returned by the algorithm
- OPT = optimal solution
- ε = small number representing allowed error (e.g., 0.1 for 10% error)

2.2 Fully Polynomial-Time Approximation Scheme (FPTAS)

A **Fully Polynomial-Time Approximation Scheme (FPTAS)** is:

- An algorithm that can produce a $(1 - \varepsilon)$-approximation for *any* $\varepsilon > 0$
- Runs in polynomial time in both:
  - n = number of items
  - $1/\varepsilon$ = precision parameter

Key idea: Convert a *pseudo-polynomial algorithm* (fast if numbers are small) into a *truly polynomial* approximation algorithm by scaling.

—

3 0/1 Knapsack Problem – Step by Step

3.1 Problem Definition

Given:

- n items, each with value v_i and weight w_i
- Knapsack capacity W

Goal: Pick a subset of items to maximize total value without exceeding W:

$$\max \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W, \quad x_i \in \{0, 1\}$$

3.2 Weight-based Dynamic Programming

We can define a DP table as:

$DP[i][w]$ = maximum value achievable using first i items with total weight $\leq w$

Recurrence:

$$DP[i][w] = \begin{cases} DP[i-1][w], & \text{if } w_i > w \\ \max(DP[i-1][w], v_i + DP[i-1][w-w_i]), & \text{otherwise} \end{cases}$$

Time Complexity: $O(nW)$ – called **pseudo-polynomial** because it depends on W, which may be very large.

Example:

$$(v, w) = \{(60, 10), (100, 20), (120, 30)\}, \quad W = 50$$

DP table size = 3×50 = manageable. But if $W = 10^6$, runtime = $O(3 \cdot 10^6)$, too slow.

—

3.3 Value-based Dynamic Programming

Alternative DP:

$DP[i][v]$ = minimum weight needed to achieve total value v using first i items

Recurrence:

$$DP[i][v] = \begin{cases} DP[i-1][v], & \text{if } v_i > v \\ \min(DP[i-1][v], w_i + DP[i-1][v-v_i]), & \text{otherwise} \end{cases}$$

We then choose the largest v s.t. $DP[n][v] \leq W$.

Time Complexity: $O(nV_{\max})$, where $V_{\max} = \sum_i v_i$

—

4 Why DP is Pseudo-Polynomial

The time depends on $V_{\max}$ (sum of item values) or W (sum of weights), not on the *number of bits needed to represent input*:

$$\text{Input size } \sim O(n \log W)$$

Hence, it's not truly polynomial.

—

Why the Standard (Weight-based) DP Cannot Be Turned into an FPTAS

The standard dynamic programming (DP) approach for the knapsack problem has runtime

$$O(nW),$$

where W is the numeric weight limit.

This makes it **pseudo-polynomial**, because W can be exponentially large compared to its bit-length $\log W$.

We cannot simply scale or round weights, because the constraint is a *hard limit*. A small rounding error could make the total weight exceed W, thus invalidating feasibility.

$$\text{Rounding weights } \Rightarrow \text{ breaks feasibility.}$$

Hence, there is **no known FPTAS** based on the weight-based DP.

Step 3: Why the Value-Based DP Works for FPTAS

The value-based DP computes:

$$dp[i][v] = \text{minimum weight required to achieve value } v,$$

and runs in

$$O(nV_{\max}),$$

where $V_{\max}$ is the maximum total value.

The idea is that instead of iterating over all possible values, we can *scale down the values* so that $V_{\max}$ becomes small and the runtime becomes polynomial.

The Scaling Trick (Core of FPTAS)

Let K be a scaling factor. We define new scaled values:

$$v_i' = \left\lfloor \frac{v_i}{K} \right\rfloor.$$

Now, solve the scaled instance using the value-based DP. Its runtime becomes

$$O(nV'_{\max}),$$

where

$$V'_{\max} = \sum_i v'_i \approx \frac{V_{\max}}{K}.$$

We choose K cleverly so that:

$$K = \frac{\varepsilon\, v_{\max}}{n},$$

which ensures runtime polynomial in n and $1/\varepsilon$, and also guarantees that the total loss due to rounding is within the ε margin.

5 Developing FPTAS for Knapsack

5.1 Step 1: Scaling Item Values

Choose:

$$K = \frac{\varepsilon \cdot v_{\max}}{n}$$

and define scaled values:

$$v'_i = \left\lfloor \frac{v_i}{K} \right\rfloor$$

We then run value-based DP on v'_i.

5.2 Step 2: Why This Scaling Works

- Each item loses at most K due to flooring:

$$0 \le v_i - K \cdot \lfloor v_i/K \rfloor < K$$

- Total loss over n items $\le nK = \varepsilon v_{\max} \le \varepsilon OPT$

Hence, final solution $\ge (1-\varepsilon)OPT$.

—

FPTAS for Knapsack: Worked Example

1) Problem Instance (Small, Exact)

Items (v_i, w_i):

$$(12,4), \quad (10,6), \quad (8,5), \quad (11,7)$$

Knapsack capacity: $W = 15$.

$$V_{\max} = 12 + 10 + 8 + 11 = 41, \quad v_{\max} = 12$$

We will run an FPTAS with error parameter $\varepsilon = 0.20$ (20% allowed loss).

2) Choose Scaling Factor K

A common and practical choice is:

$$K = \frac{\varepsilon\, v_{\max}}{n}.$$

Reason: $v_{\max}$ is the largest single-item value, useful to bound rounding error. If an item with value $v_{\max}$ is infeasible (i.e., $w_i > W$), ignore such items for computing $v_{\max}$.

Given $n = 4$, $v_{\max} = 12$, $\varepsilon = 0.2$:

$$K = \frac{0.2 \times 12}{4} = \frac{2.4}{4} = 0.6.$$

(Alternative choice often used: $K = \varepsilon V_{\max}/n$.)

3) Scale Values (Round Down)

We compute:

$$v_i' = \left\lfloor \frac{v_i}{K} \right\rfloor.$$

$$\begin{aligned} v_1' &= \lfloor 12/0.6 \rfloor = 20, \\ v_2' &= \lfloor 10/0.6 \rfloor = 16, \\ v_3' &= \lfloor 8/0.6 \rfloor = 13, \\ v_4' &= \lfloor 11/0.6 \rfloor = 18. \end{aligned}$$

Each original value satisfies:

$$K v_i' \le v_i < K(v_i' + 1),$$

so the rounding error per item is less than K.

4) Run Value-Based DP on Scaled Instance

We run the DP:

$$dp[v'] = \text{minimum total weight needed to achieve scaled value } v'.$$

Initialize:

$$dp[0] = 0, \quad dp[v'] = +\infty \text{ for } v' > 0.$$

Process items sequentially (iterating v' high to low):

- After item 1 ($v_1' = 20, w_1 = 4$): $dp[20] = 4$
- After item 2 ($v_2' = 16, w_2 = 6$): $dp[16] = 6,\ dp[36] = dp[20] + 6 = 10$
- After item 3 ($v_3' = 13, w_3 = 5$):

$$dp[33] = dp[20] + 5 = 9, \quad dp[29] = dp[16] + 5 = 11$$

- After item 4 ($v_4' = 18, w_4 = 7$): more combinations are formed.

The total possible scaled sum:

$$V'_{\max} = 20 + 16 + 13 + 18 = 67.$$

We now find the maximum v^* such that:

$$dp[v^*] \leq W(= 15).$$

The best feasible combination is items $\{1, 2, 3\}$ (weights $4 + 6 + 5 = 15$) giving scaled sum:

$$v^* = 20 + 16 + 13 = 49.$$

The corresponding real total value:

$$A = 12 + 10 + 8 = 30.$$

The true optimal value (OPT) is also 30, so the FPTAS found the optimal solution here.

5) Mapping Back and Approximation Guarantee

The DP returned scaled value v^* with $dp[v^*] \leq W$. Each chosen real value satisfies $v_i \geq Kv'_i$, so:

$$A = \sum_{i \in S} v_i \geq K \sum_{i \in S} v'_i = Kv^*.$$

Hence,

$$A \geq (1 - \varepsilon)OPT.$$

In our example:

$$A = 30, \quad OPT = 30, \quad A/OPT = 1.0 \geq 1 - \varepsilon = 0.8.$$

6) Proof Sketch for $(1 - \varepsilon)$-Approximation

For each item i:

$$v_i < K(v'_i + 1) \implies v_i - Kv'_i < K.$$

Summing over items in the optimal set:

$$OPT = \sum_{i \in OPT} v_i < K \sum_{i \in OPT} v'_i + nK.$$

Thus,

$$OPT' > \frac{OPT}{K} - n.$$

Since the DP finds the optimal scaled solution ($A' \geq OPT'$):

$$A' \geq \frac{OPT}{K} - n.$$

Multiplying by K:

$$A \geq KA' \geq OPT - nK.$$

Choose $K = \frac{\varepsilon v_{\max}}{n}$. Since $v_{\max} \leq OPT$,

$$nK = n \cdot \frac{\varepsilon v_{\max}}{n} = \varepsilon v_{\max} \leq \varepsilon OPT.$$

Therefore:

$$A \geq OPT - \varepsilon OPT = (1 - \varepsilon)OPT.$$

7) Runtime Analysis

After scaling:

$$v_i' \le \frac{v_i}{K} \le \frac{v_{\max}}{K} = \frac{v_{\max}}{\varepsilon v_{\max}/n} = \frac{n}{\varepsilon}.$$

Hence:

$$V_{\max}' = \sum_i v_i' \le n \cdot \frac{n}{\varepsilon} = \frac{n^2}{\varepsilon}.$$

Runtime of the value-based DP:

$$T(n, \varepsilon) = O(n \cdot V_{\max}') = O\left(\frac{n^3}{\varepsilon}\right).$$

With optimization, it can be improved to:

$$T(n, \varepsilon) = O\left(\frac{n^2}{\varepsilon}\right),$$

which is polynomial in both n and $1/\varepsilon$.

8) Recap and Intuition

- The exact DP is pseudo-polynomial (depends on $V_{\max}$).
- FPTAS scales values down so DP size is polynomial in $n, 1/\varepsilon$.
- Total rounding loss $\le nK$.
- Choosing $K = \Theta(\varepsilon v_{\max}/n)$ ensures loss $\le \varepsilon OPT$.

Hence, the returned solution satisfies:

$$A \ge (1 - \varepsilon)OPT, \quad T(n, \varepsilon) = \text{poly}(n, 1/\varepsilon).$$

9) Numeric Check on Our Instance

With $\varepsilon = 0.2,\ K = 0.6$, scaled values are $20, 16, 13, 18$. DP yields items $\{1, 2, 3\}$, scaled sum 49, real value 30, weight 15.

$$OPT = 30, \quad A = 30 \ge (1 - 0.2) \cdot 30 = 24.$$

Perfect — the algorithm achieved the exact optimum.

10) Practical Notes

- K may also be chosen as $K = \varepsilon V_{\max}/n$.
- Always ensure $v_{\max}$ refers to an item with $w_i \le W$.
- The method scales easily; for large n, memory/time may still grow, but remains polynomial.

6 Intuition Behind $K = \varepsilon v_{\max}/n$

Let K be a scaling factor.

We define new scaled values:

$$v'_i = \left\lfloor \frac{v_i}{K} \right\rfloor .$$

Now, solve the scaled instance using the value-based DP. Its runtime becomes:

$$O(nV'_{\max}),$$

where

$$V'_{\max} = \sum_i v'_i \approx \frac{V_{\max}}{K}.$$

We choose K cleverly so that:

$$K = \frac{\varepsilon\, v_{\max}}{n},$$

which ensures that the runtime is polynomial in n and $1/\varepsilon$, and also guarantees that the total loss due to rounding is within the ε-margin.

—

7 Other Problems Admitting FPTAS

- **Subset Sum:** Scale numbers; DP tracks sums target
- **Partition Problem:** Special case of Subset Sum
- **0/1 Knapsack:** Values scaled as shown

Not applicable: Problems without pseudo-polynomial DP (e.g., general TSP, Bin Packing).

Reason: The Core Idea Behind FPTAS Scaling

A **Fully Polynomial Time Approximation Scheme (FPTAS)** works by scaling down large numerical parameters (typically values or costs) to make a pseudo-polynomial dynamic programming (DP) algorithm run in true polynomial time.

This works only for problems that:

- Already have pseudo-polynomial algorithms (e.g., DP with numeric limits), and
- Are optimization problems with bounded numeric inputs.

Thus, the scaling trick converts a pseudo-polynomial algorithm $\rightarrow$ true polynomial time, at the cost of a small approximation error.

2. Where the Trick Works

Let's check a few famous problems and whether the scaling trick (FPTAS) applies:

booktabs, array, graphicx

cc

Problem	Has pseudo-poly DP?	FPTAS possible?	What we scale?	Remarks
0/1 Knapsack	Yes ($O(nW)$ or $O(nV_{\max})$)	Yes	Values v_i	Canonical example
Subset Sum	Yes ($O(n \cdot T)$)	Yes	Target sum / subset sums	Essentially same as knapsack
Partition Problem	Yes (special case of Subset Sum)	Yes	Subset sums	FPTAS via same scaling trick
Scheduling ($1\|\|L_{\max}$)	Exact poly algo	Not needed	—	Already polynomial
Scheduling ($1\|\|\sum T_i$)	NP-hard, no pseudo-poly DP	No	—	No FPTAS possible
Travelling Salesman (metric)	No pseudo-poly DP	No	—	Only PTAS for special cases (e.g., Euclidean)
Bin Packing	No pseudo-poly DP	No	—	Only Asymptotic PTAS (APTAS)

3. Why It Works for Some and Not Others

The key structure enabling FPTAS is the *pseudo-polynomial DP form*:

$$DP[i][x] = \text{optimum value using first } i \text{ items and parameter } x,$$

where x runs up to a numeric limit such as W or $V_{\max}$.

This dependence on numeric magnitude (not bit-length) is what makes the algorithm pseudo-polynomial.

If we can scale those numeric parameters (like v_i or weights) by a factor related to ε, we can make the DP's range polynomial in n and $1/\varepsilon$, yielding a truly polynomial-time approximation algorithm.

4. Example: Subset Sum Problem

Problem: Given numbers $a_1, a_2, \ldots, a_n$ and a target T, find a subset whose sum is $\leq T$ and as close to T as possible.

DP (Pseudo-polynomial): We maintain all possible subset sums up to T. Time complexity $= O(nT)$.

Scaling Trick for FPTAS:

$$a_i' = \left\lfloor \frac{a_i}{K} \right\rfloor, \qquad K = \frac{\varepsilon\, a_{\max}}{n}.$$

Then the maximum sum becomes approximately $O(n/\varepsilon)$, so the runtime is $O(n^2/\varepsilon)$ — polynomial in n and $1/\varepsilon$.

The recovered subset sum gives a $(1-\varepsilon)$-approximation to the true optimum.

$$\Rightarrow \text{Hence, Subset Sum admits an FPTAS (and so does Partition).}$$

5. When It Fails

For problems like Job Scheduling, TSP, or Vertex Cover, the input cannot be expressed as a DP bounded by a numeric sum. Thus, there's no pseudo-polynomial structure to "scale down."

Hence:

$$\text{FPTAS} \neq \text{general-purpose trick;}$$

it applies only to problems already having pseudo-polynomial DPs.

6. Quick tips to check for FPTAS

c

Property	Needed for FPTAS?
Has pseudo-polynomial DP	Required
Numeric objective (value/weight)	Helps define scaling
NP-hardness due to numeric encoding	Usually true
Additive / combinatorial structure	Required
Continuous optimization	Not suitable

—

8 Summary

- Pseudo-polynomial algorithms are polynomial in numeric values, not input size
- FPTAS converts pseudo-poly DP into a true polynomial-time approximate algorithm
- Scaling factor $K = \varepsilon v_{\max}/n$ balances precision and speed
- Approximation error is controlled: solution $\geq (1-\varepsilon)OPT$
- Works for Knapsack, Subset Sum, Partition