Introduction to
Approximation Algorithms

Mong-Jen Kao (高孟駿)

Friday 13:20 – 15:10

Outline

- Approximation Scheme
  - PTAS & FPTAS
- The Knapsack Problem
  - An FPTAS for Knapsack
  - Strongly NP-hardness & Non-existence of FPTAS

Approximation Schemes

– To Approximate to any Desirable Degrees

Use more computation time for ***arbitrarily-good*** approximation guarantees.

Not every problem has approximation schemes.

Approximation Schemes

- An algorithm $\mathcal{A}$ is called an ***approximation scheme***
  for an optimization problem Π if,

  on *any input instance* I and *any error parameter* $\epsilon > 0$,
  the algorithm $\mathcal{A}$ always produces

  - a $(1+\epsilon)$-approximate solution for I, if Π is a *minimization* problem,
  - a $(1-\epsilon)$-approximate solution for I, if Π is a *maximization* problem.

  - That is, $|\mathcal{A}(I) - OPT_I| \leq \epsilon \cdot OPT_I$ always holds.

The *relative error* between $\mathcal{A}(I)$ and OPT_I can be arbitrarily small!

Approximation Schemes

- An approximation scheme $\mathcal{A}$ is said to be
  - A ***polynomial-time approximation scheme*** **(PTAS)** if its *running time* is bounded by a *polynomial in* $|I|$, i.e., $g\left(\frac{1}{\epsilon}\right) \cdot poly(|I|)$ for some function g.
  - A ***fully polynomial-time approximation scheme*** (**FPTAS or Fully-PTAS**), if its *running time* is bounded by a ***polynomial in*** $|\boldsymbol{I}|$ ***and*** $\boldsymbol{1/\epsilon}$, i.e., $poly\left(|I|, \frac{1}{\epsilon}\right)$

A systematic way to exchange computation time for *arbitrarily close* approximation guarantees.

Approximation Schemes

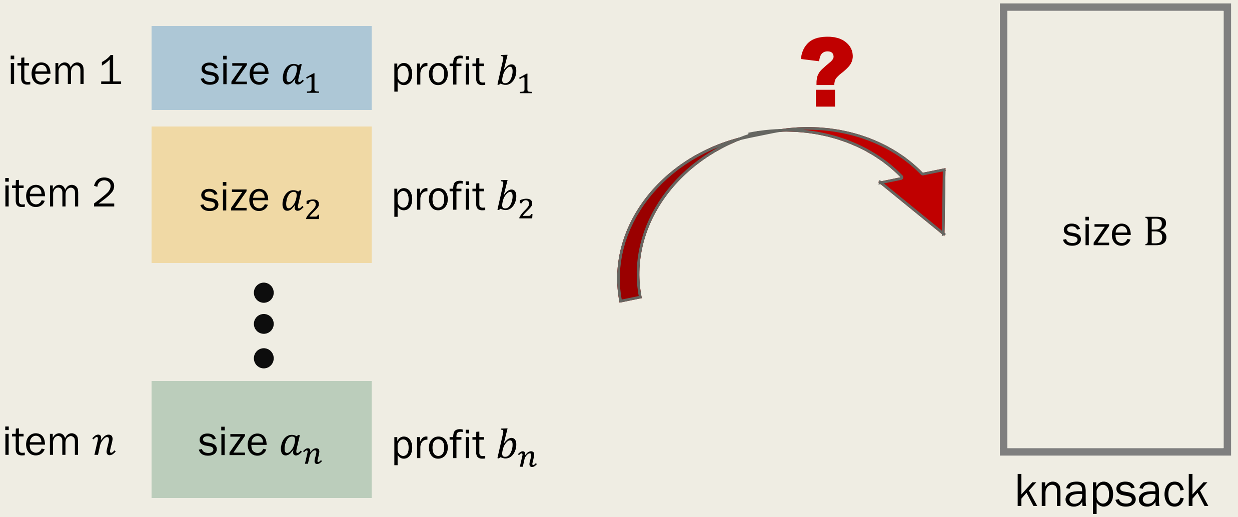
- In this course, we will see
  - An *FPTAS* for the Knapsack problem.
  - A necessary condition for FPTAS to exist for a problem.

The Knapsack Problem

The Knapsack Problem

- Given a set of n items with **size a_i** and **profit b_i**, where $1 \leq i \leq n$, and a **knapsack size B**,

  the Knapsack problem is to compute a **subset $A \subseteq [1, n]$** with $\sum_{i \in A} a_i \leq B$ such that $\sum_{i \in A} b_i$ **is maximized**.



To maximize the **total profit** put in the knapsack

The Knapsack Problem

- The Knapsack problem is a classic *NP-complete* problem.
  - *Can be reduced from the Partition problem*, one of the 6 basic NP-complete problems.

- The Knapsack problem can be solved by *dynamic programming* in *pseudo-polynomial time*.

- We will see this problem can be approximated *efficiently to any desirable degree*.

Dynamic Programming

for the Knapsack Problem

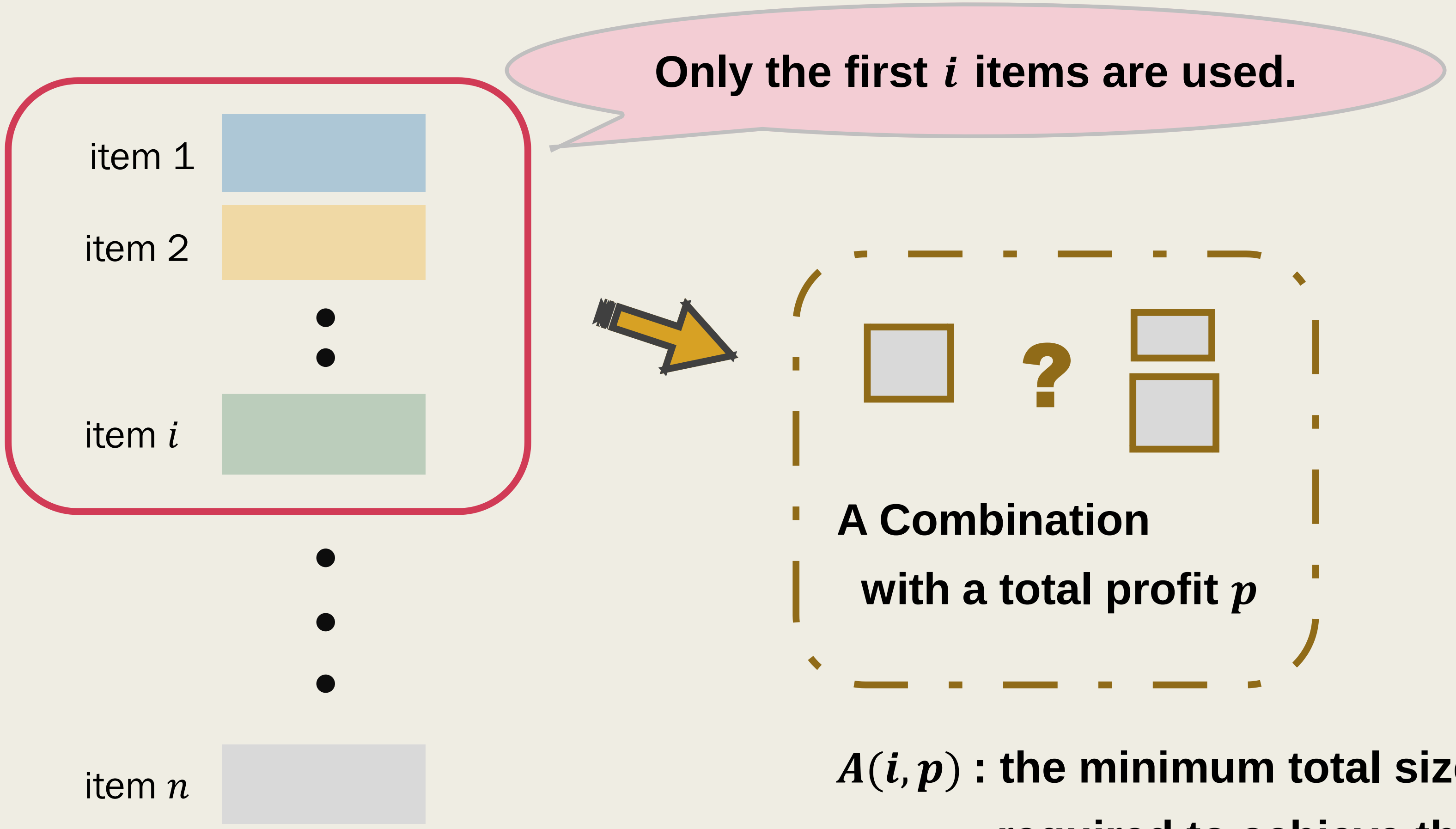
Dynamic Programming for Knapsack

- The Knapsack problem can be solved by standard dynamic programming technique in pseudo-polynomial time.

  – For any $0 \leq i \leq n$ and $p \geq 0$,
    let $A(i, p)$ denote the **minimum total size** it requires to get a **total profit of p *using only the first i items***.

    $A(i, p)$ is defined to be ∞ if no such combination exists.

- For any $0 \le i \le n$ and $p \ge 0$,
  let $A(i, p)$ denote the **minimum total size** it requires to get a **total profit of p using only the first i items**.



$A(i, p)$: the minimum total size required to achieve this setting.

- For any $0 \leq i \leq n$ and $p \geq 0$, let $A(i, p)$ denote the **minimum total size** it requires to get a **total profit of p using *only the first i items***.

  $A(i, p)$ is defined to be ∞ if no such combination exists.

- Let $P = max_{1 \leq i \leq n} b_i$.

  Clearly, the answer to the Knapsack problem is

  the maximum p, where $0 \leq p \leq n \cdot P$, that makes $A(n, p) \leq B$.

The Recurrence Formula for $A(i,p)$

- By our definition, when $i > 0$, we have

When $p < 0$,
we have no valid combination at all.

$$A(i,p) = \begin{cases} \begin{cases} \infty, & \text{if } p < 0 \\ \min\{A(i\text{-}1,p),\ A(i\text{-}1,p\text{-}b_i) + a_i\}, & \text{if } p \geq 0 \end{cases}, & \text{for } i > 0, \end{cases}$$

For $p \geq 0$, the optimal combination either **contains the i^{th}-item** or **does not contain it**.

has **size** $A(i-1, p-b_i) + a_i$

has **size** $A(i-1, p)$

A Combination with a total profit p

The Recurrence Formula for $A(i,p)$

- For $i = 0$, we have

$$A(i,p) = \begin{cases} \quad \\ \begin{cases} 0, & \text{if } p = 0 \\ \infty, & \text{if } p \neq 0 \end{cases}, & \text{for } i = 0. \end{cases}$$

When $i = 0$, no item is available for use.

The only valid combination is an empty set with a zero size.

The Recurrence Formula for $A(i, p)$

- We have the recurrence for $A(i, p)$

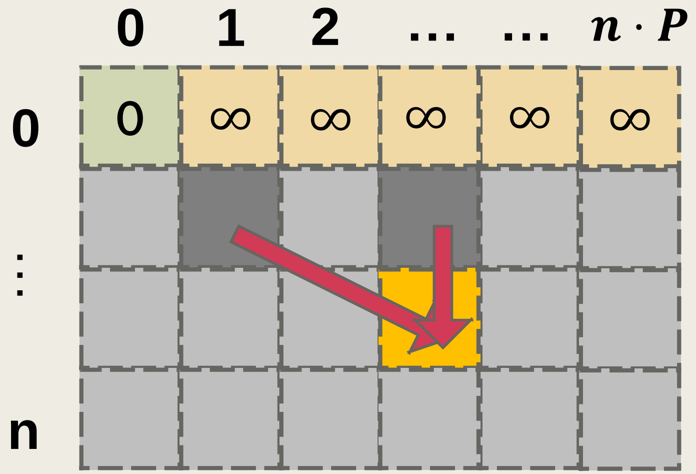
$$A(i,p) = \begin{cases} \begin{cases} \infty, & \text{if } p < 0 \\ \min\{ A(i\text{-}1,p),\ A(i\text{-}1, p\text{-}b_i) + a_i \}, & \text{if } p \geq 0 \end{cases}, & \text{for } i > 0, \\ \\ \begin{cases} 0, & \text{if } p = 0 \\ \infty, & \text{if } p \neq 0 \end{cases}, & \text{for } i = 0. \end{cases}$$

- Using the formula, we can compute $A(i, p)$ for all $0 \leq i \leq n$ and $0 \leq p \leq n \cdot P$, where $\mathrm{P} = \max_{1 \leq i \leq n} b_i$ is the maximum profit of the items.
  - The time complexity is $O(n^2 \cdot P)$.

Dynamic Programming for Knapsack

$$A(i,p)=\begin{cases}\begin{cases}\infty, & \text{if } p<0\\ \min\{A(i\text{-}1,p),\ A(i\text{-}1,p\text{-}b_i)+a_i\}, & \text{if } p\geq 0\end{cases}, & \text{for } i>0,\\[2ex] \begin{cases}0, & \text{if } p=0\\ \infty, & \text{if } p\neq 0\end{cases}, & \text{for } i=0.\end{cases}$$

The time complexity is $O(n^2 \cdot P)$.



A Pseudo-Polynomial Time Algorithm

- The Knapsack problem can be solved by standard dynamic programming technique.
  - The time complexity is $O(n^2 \cdot P)$,

    which is not polynomial in the input length n but ***grows with** the* ***value of the input numbers***.
  - It is a ***pseudo-polynomial time*** algorithm.

It can be ***very slow***
when the value of P is large.

Inefficiency of Pseudo-Polynomial Time Algorithms

- For example,
  - $n = 2,\ \max b_i = 10^{18}$,

    DP takes $\Theta(10^{18})$ time to execute.

  - In contrast to the sorting algorithm,
    whose running time does not depend on the value of the inputs,
    DP for Knapsack can be very inefficient.

  - This is inevitable, if the optimal solution must be computed.

One Natural Question to Ask

- The computation for the Knapsack problem is *time-consuming* because it requires ***absolute precision*** **in the resulting size and profit**.

- If only *near-optimal solutions* are sought, can we compute a good solution ***efficiently*** for the Knapsack problem?

Approximating
the Optimal Solution for Knapsack

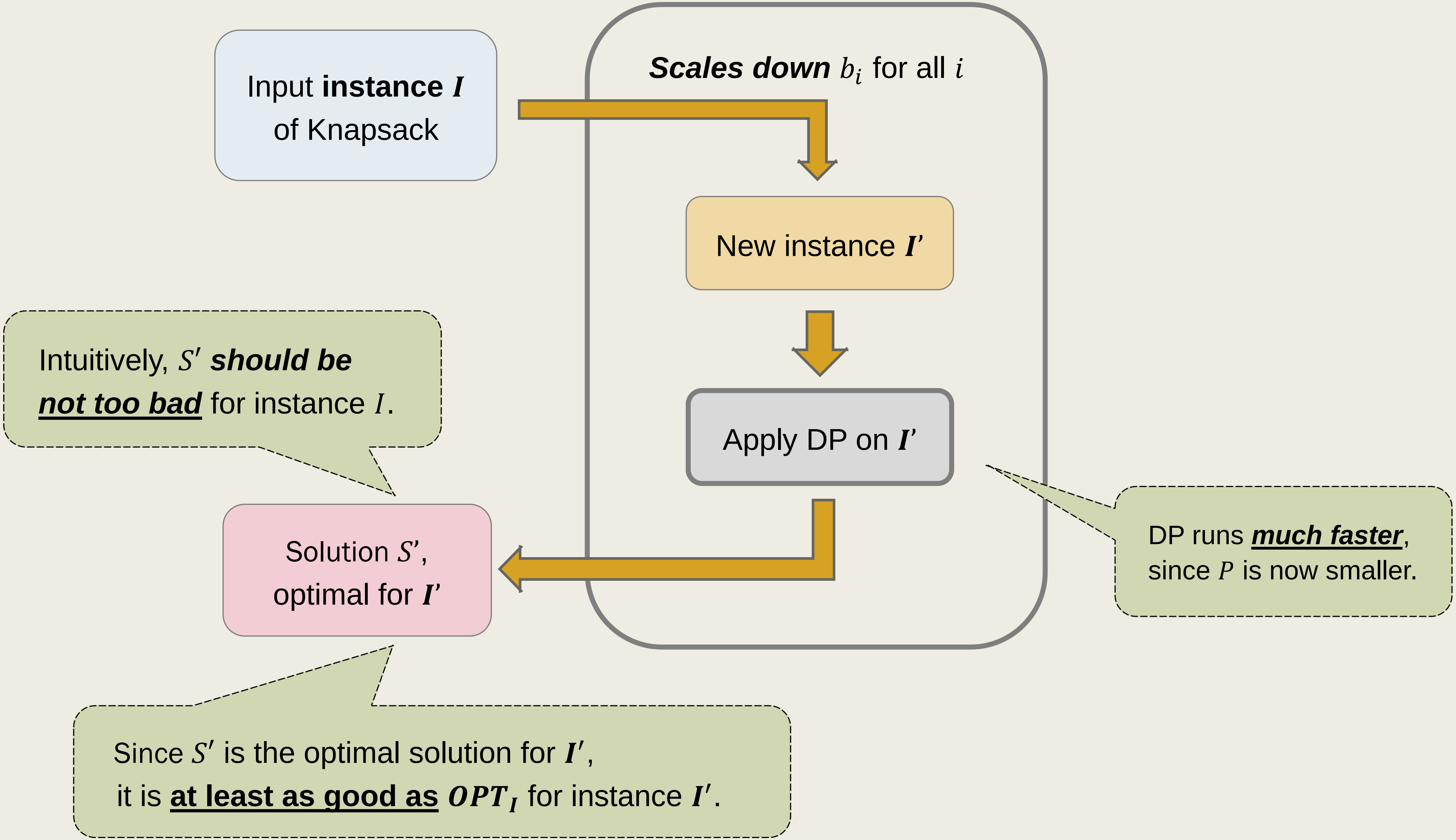
With ***a little bit*** *(?) of compromise* on the solution quality,
we can compute a good solution ***a lot faster***!

Observation and Idea

- The computation for the Knapsack problem is *time-consuming* because it aims for an *absolute precision in the resulting value*.

- By ***scaling down** the profits of the items*, we can *reduce the range of possible profit*.
  - The range of profits becomes smaller.
  - Dynamic programming becomes much faster, and the *solution computed* is still ***reasonably good***.

Simple idea, works great.


Input instance I of Knapsack
Scales down b_i for all i
New instance I'
Apply DP on I'
Solution S', optimal for I'
Intuitively, S' should be not too bad for instance I.
DP runs much faster, since P is now smaller.
Since S' is the optimal solution for $\boldsymbol{I'}$, it is at least as good as $\boldsymbol{OPT_I}$ for instance $\boldsymbol{I'}$.

Observation and Idea

- Let K be the *scaling factor* for the profits,
  i.e., we are to set $b_i' := \lfloor b_i/K \rfloor$ for all $1 \le i \le n$.
  - For dynamic programming to run *in time polynomial in n*,
    K must be $\Omega(P/n)$.

So that, the new maximum profit will be

$$\max_{1 \le i \le n} b_i' = \max_{1 \le i \le n} \lfloor b_i/K \rfloor = O\big(poly(n)\big).$$

Algorithm Description

Approximation Algorithm $\mathcal{A}$ for Knapsack

- Let
  - $\boldsymbol{I} = \{(a_1, b_1), (a_2, b_2), \dots, (a_n, b_n), B\}$ *denote the input instance*
  - $\epsilon > 0$ be the input *error parameter*

- W.L.O.G., we assume
  - $B \geq \max_{1 \leq i \leq n} a_i$, and hence $OPT_I \geq P$.
    - If $a_i > B$ for some item i, then this item can be dropped.

Description of the Algorithm $\mathcal{A}$

1. Let $K = \frac{\epsilon P}{n}$, where $P := \max_{1 \le i \le n} b_i$.

2. For each $1 \le i \le n$, define $b_i' := \left\lfloor \frac{b_i}{K} \right\rfloor$.

3. Apply *dynamic programming* on $\boldsymbol{I}' = \{(a_1, b_1'), (a_2, b_2'), \ldots, (a_n, b_n'), B\}$.
   Let S' be the combination computed.

4. Output S' as the approximate solution for $\boldsymbol{I}$.

Analysis of Algorithm $\mathcal{A}$

The Analysis

- To show that $\mathcal{A}$ is a $(1-\epsilon)$-approximation for Knapsack, we need to prove the following.

  - The ***feasibility*** of the algorithm.

    $\boldsymbol{S'}$ is *indeed a feasible solution* for the input instance $\boldsymbol{I}$.

  - The ***approximation guarantee*** of the algorithm.

    The *value of* S' *with respect to* I is at least $(1-\epsilon)$ times the profit of the (unknown) optimal combination OPT_I, i.e.,

$$\sum_{i \in S'} b_i \geq (1-\epsilon) \cdot \sum_{i \in OPT_I} b_i \ .$$

The Feasibility of $\mathcal{A}$

- The dynamic programming returns a feasible solution for I'.

  So, we have $\sum_{i \in S'} a_i \leq B$.

- Since I and I' have the same Knapsack size,

  S' is also feasible for I.

The Approximation Guarantee of $\mathcal{A}$

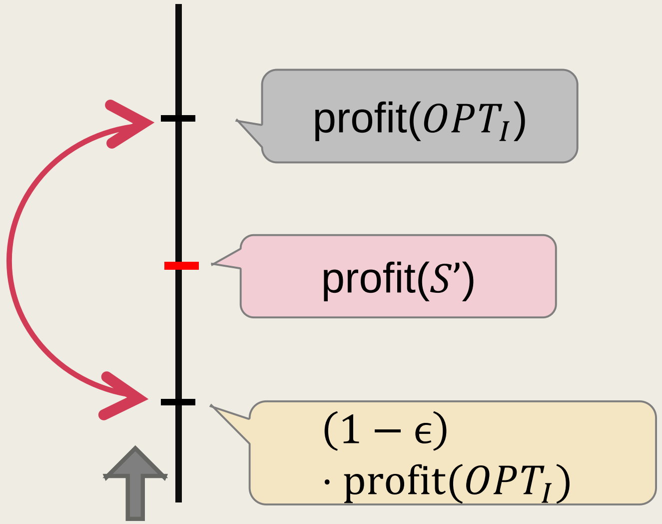
- For any $A \subseteq [1, \ldots, n]$, let $\mathrm{profit}(A)$ denote the profit of A under I and $\mathrm{profit}'(A)$ denote the profit of A under I', i.e.,

$$\mathrm{profit}(A) \coloneqq \sum_{i \in A} b_i \quad \text{and} \quad \mathrm{profit}'(A) \coloneqq \sum_{i \in A} b_i' \ .$$

- We will prove following lemma.

Lemma.

We have $\mathrm{profit}(S') \geq (1 - \epsilon) \cdot \mathrm{profit}(OPT_I)$.



Lemma.

We have $\text{profit}(S') \geq (1-\epsilon) \cdot \text{profit}(OPT_I)$.

- By the setting of b_i' for any item i, we have

$$b_i \geq K \cdot b_i' \geq b_i - K .$$

- Then, we have

$$\text{profit}(S') \geq K \cdot \text{profit}'(S') \geq K \cdot \text{profit}'(OPT) \geq \text{profit}(OPT) - n \cdot K .$$

S' is optimal for I'.

At most n items are selected in OPT_I.

- By the definition of K, we have

$$\text{profit}(OPT) - n \cdot K = \text{profit}(OPT) - \varepsilon \cdot P \geq (1-\varepsilon) \cdot \text{profit}(OPT) .$$

$P \leq \text{profit}(OPT_I)$.

$(1-\epsilon)$-Approximation for Knapsack

- In conclusion, we obtain the following theorem.

Theorem.

Algorithm $\mathcal{A}$ computes a $(1-\epsilon)$-approximation solution for the Knapsack problem in $O(n^3/\epsilon)$ time.

- The time required by DP is $O\left(n^2 \cdot \left\lfloor \frac{P}{K} \right\rfloor\right) = O(n^3/\epsilon)$.

Not many problems have FPTAS.

Strongly NP-hardness &

Non-existence of FPTAS

The Existence of FPTAS

- In theory, FPTAS seems to be the most desirable algorithm for combinatorial optimization problems.
  - It approximates the problem to any desirable degree.
  - It may *not always* *be* *practically useful*, since the *desirable solution quality often requires undesirable running time*.

- Nevertheless, only a small portion of problems has FPTAS, which we will see in the following.

The Existence of FPTAS

- In the following,
  we derive a necessary condition for the existence of FPTAS.

- When the objective function is
  - *Integer-valued*, and
  - *Polynomially-bounded* by the sum of ***input numbers***, i.e.,

$$OPT_I < poly\left(\sum_{a \in I} |a|\right),$$

FPTAS leads to a *pseudo-polynomial time algorithm*.

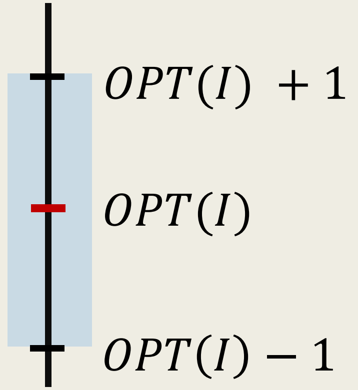
The Existence of FPTAS

- When the objective function of the problem is <u>*integer-valued*</u> and polynomially-bounded by the sum of ***the input numbers***, i.e., $OPT_I < poly(\sum_{a \in I} |a|)$, **FPTAS leads to *pseudo-polynomial time algorithms***.
  - The idea is simple:

To force FPTAS to return an optimal solution.

Set ϵ properly, so that FPTAS has to return a solution that is within $OPT(I) \pm 1$.

Since the objective is integer-valued, FPTAS **must** return the optimal solution.



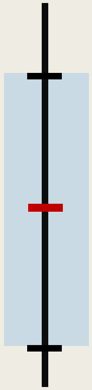
- When the objective function of the problem is *integer-valued* and polynomially-bounded by the sum of ***the input numbers***, i.e., $OPT_I < poly(\sum_{a \in I} |a|)$, **FPTAS leads to *pseudo-polynomial time algorithms***.

  – Assume the above conditions.
  We will derive a pseudo-polynomial time algorithm for this problem.

  – Let $\epsilon = 1/poly(\sum_{a \in I} |a|)$ and apply the FPTAS.
  Then the value of the solution computed is within

  $$(1 \pm \epsilon) \cdot OPT_I < OPT_I \pm \epsilon \cdot p(|I_u|) = OPT_I \pm 1,$$

  which means that it must be OPT_I.



  – The running time of the above is polynomial in $|I|$ and $1/\epsilon$, which is $poly(\sum_{a \in I} |a|)$, i.e., polynomial in the input numbers.

The Existence of FPTAS

- We have derived a necessary condition for the existence of FPTAS for a large category of optimization problems, i.e., problems with integer-valued & polynomially-bounded objective.
  - When such a problem has an FPTAS, it must have a pseudo-polynomial time algorithm as well.
  - Conversely, if such a problem has no pseudo-polynomial time algorithm, it cannot have an FPTAS.

Strongly NP-hardness

- An NP-hard problem is said to be ***strongly NP-hard***,

  if the problem *remains NP-hard* even when *all of its input numbers* are *bounded* by a *polynomial in its input length*.

  - Most NP-hard problems are in fact strongly NP-hard.
  - By definition, *strongly NP-hard problems* have *no pseudo-polynomial time algorithms*, unless P=NP.

Most of the problems we consider in this course are in this category.

An Alternative Definition

- An NP-hard problem is said to be ***strongly NP-hard***,

  if the problem *remains NP-hard* even when *all of its input numbers* are written in *unary* representation.

  – That is, instead of writing a number in its binary representation, we use the unary representation.

  – For example, for the number 10,
  we use 1111111111 instead of 1010.

Strongly NP-hardness

- An NP-hard problem is called ***strongly NP-hard***, if it *remains NP-hard* even when *all of its input numbers* are *bounded* by a *polynomial in its input length*.
  - Most NP-hard problems are in fact strongly NP-hard.
  - By definition, *strongly NP-hard problems* have *no pseudo-polynomial time algorithms*, unless P=NP.

- Hence, we conclude that, strongly NP-hard problems with integer-valued & polynomially-bounded objective cannot have FPTAS, unless P=NP.

Most of the problems we consider in this course are in this category.

Let’s proceed to our next problem.