

## 15 Rounding Data + Dynamic Programming

### Knapsack:

Given a set of items  $\{1, \dots, n\}$ , where the  $i$ -th item has weight  $w_i \in \mathbb{N}$  and profit  $p_i \in \mathbb{N}$ , and given a threshold  $W$ . Find a subset  $I \subseteq \{1, \dots, n\}$  of items of total weight at most  $W$  such that the profit is maximized (we can assume each  $w_i \leq W$ ).

$$\begin{array}{ll} \max & \sum_{i=1}^n p_i x_i \\ \text{s.t.} & \sum_{i=1}^n w_i x_i \leq W \\ & \forall i \in \{1, \dots, n\} \quad x_i \in \{0, 1\} \end{array}$$

# 15 Rounding Data + Dynamic Programming

## Knapsack:

Given a set of items  $\{1, \dots, n\}$ , where the  $i$ -th item has weight  $w_i \in \mathbb{N}$  and profit  $p_i \in \mathbb{N}$ , and given a threshold  $W$ . Find a subset  $I \subseteq \{1, \dots, n\}$  of items of total weight at most  $W$  such that the profit is maximized (we can assume each  $w_i \leq W$ ).

$$\begin{array}{ll} \max & \sum_{i=1}^n p_i x_i \\ \text{s.t.} & \sum_{i=1}^n w_i x_i \leq W \\ & \forall i \in \{1, \dots, n\} \quad x_i \in \{0, 1\} \end{array}$$

# 15 Rounding Data + Dynamic Programming

## Algorithm 1 Knapsack

```
1:  $A(1) \leftarrow [(0, 0), (p_1, w_1)]$ 
2: for  $j \leftarrow 2$  to  $n$  do
3:    $A(j) \leftarrow A(j - 1)$ 
4:   for each  $(p, w) \in A(j - 1)$  do
5:     if  $w + w_j \leq W$  then
6:       add  $(p + p_j, w + w_j)$  to  $A(j)$ 
7:       remove dominated pairs from  $A(j)$ 
8: return  $\max_{(p, w) \in A(n)} p$ 
```

The running time is  $\mathcal{O}(n \cdot \min\{W, P\})$ , where  $P = \sum_i p_i$  is the total profit of all items. This is only **pseudo-polynomial**.

# 15 Rounding Data + Dynamic Programming

## Definition 74

An algorithm is said to have pseudo-polynomial running time if the running time is polynomial when the numerical part of the input is encoded in unary.

# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.

# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .

# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .
- ▶ Set  $p'_i := \lfloor p_i/\mu \rfloor$  for all  $i$ .

## 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .
- ▶ Set  $p'_i := \lfloor p_i/\mu \rfloor$  for all  $i$ .
- ▶ Run the dynamic programming algorithm on this revised instance.



# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .
- ▶ Set  $p'_i := \lfloor p_i/\mu \rfloor$  for all  $i$ .
- ▶ Run the dynamic programming algorithm on this revised instance.

Running time is at most

$$\mathcal{O}(nP')$$

# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .
- ▶ Set  $p'_i := \lfloor p_i/\mu \rfloor$  for all  $i$ .
- ▶ Run the dynamic programming algorithm on this revised instance.

Running time is at most

$$\mathcal{O}(nP') = \mathcal{O}\left(n \sum_i p'_i\right)$$

# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .
- ▶ Set  $p'_i := \lfloor p_i/\mu \rfloor$  for all  $i$ .
- ▶ Run the dynamic programming algorithm on this revised instance.

Running time is at most

$$\mathcal{O}(nP') = \mathcal{O}\left(n \sum_i p'_i\right) = \mathcal{O}\left(n \sum_i \left\lfloor \frac{p_i}{\epsilon M/n} \right\rfloor\right)$$

# 15 Rounding Data + Dynamic Programming

- ▶ Let  $M$  be the maximum profit of an element.
- ▶ Set  $\mu := \epsilon M/n$ .
- ▶ Set  $p'_i := \lfloor p_i/\mu \rfloor$  for all  $i$ .
- ▶ Run the dynamic programming algorithm on this revised instance.

Running time is at most

$$\mathcal{O}(nP') = \mathcal{O}\left(n \sum_i p'_i\right) = \mathcal{O}\left(n \sum_i \left\lfloor \frac{p_i}{\epsilon M/n} \right\rfloor\right) \leq \mathcal{O}\left(\frac{n^3}{\epsilon}\right).$$

## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\sum_{i \in S} p_i$$

## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\sum_{i \in S} p_i \geq \mu \sum_{i \in O} p'_i$$

## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\begin{aligned}\sum_{i \in S} p_i &\geq \mu \sum_{i \in S} p'_i \\ &\geq \mu \sum_{i \in O} p'_i\end{aligned}$$

## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\begin{aligned}\sum_{i \in S} p_i &\geq \mu \sum_{i \in S} p'_i \\ &\geq \mu \sum_{i \in O} p'_i \\ &\geq \sum_{i \in O} p_i - |O|\mu\end{aligned}$$



## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\begin{aligned}\sum_{i \in S} p_i &\geq \mu \sum_{i \in S} p'_i \\ &\geq \mu \sum_{i \in O} p'_i \\ &\geq \sum_{i \in O} p_i - |O|\mu \\ &\geq \sum_{i \in O} p_i - n\mu\end{aligned}$$

## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\begin{aligned}\sum_{i \in S} p_i &\geq \mu \sum_{i \in S} p'_i \\ &\geq \mu \sum_{i \in O} p'_i \\ &\geq \sum_{i \in O} p_i - |O|\mu \\ &\geq \sum_{i \in O} p_i - n\mu \\ &= \sum_{i \in O} p_i - \epsilon M\end{aligned}$$

## 15 Rounding Data + Dynamic Programming

Let  $S$  be the set of items returned by the algorithm, and let  $O$  be an optimum set of items.

$$\begin{aligned}\sum_{i \in S} p_i &\geq \mu \sum_{i \in S} p'_i \\ &\geq \mu \sum_{i \in O} p'_i \\ &\geq \sum_{i \in O} p_i - |O|\mu \\ &\geq \sum_{i \in O} p_i - n\mu \\ &= \sum_{i \in O} p_i - \epsilon M \\ &\geq (1 - \epsilon)\text{OPT} .\end{aligned}$$

# Scheduling Revisited

The previous analysis of the scheduling algorithm gave a makespan of

$$\frac{1}{m} \sum_{j \neq \ell} p_j + p_\ell$$

where  $\ell$  is the last job to complete.

# Scheduling Revisited

The previous analysis of the scheduling algorithm gave a makespan of

$$\frac{1}{m} \sum_{j \neq \ell} p_j + p_\ell$$

where  $\ell$  is the last job to complete.

Together with the observation that if each  $p_i \geq \frac{1}{3} C_{\max}^*$  then LPT is optimal this gave a  $4/3$ -approximation.

## 15.2 Scheduling Revisited

Partition the input into **long** jobs and **short** jobs.

## 15.2 Scheduling Revisited

Partition the input into **long** jobs and **short** jobs.

A job  $j$  is called short if

$$p_j \leq \frac{1}{km} \sum_i p_i$$

## 15.2 Scheduling Revisited

Partition the input into **long** jobs and **short** jobs.

A job  $j$  is called short if

$$p_j \leq \frac{1}{km} \sum_i p_i$$

**Idea:**

1. Find the optimum Makespan for the long jobs by brute force.



## 15.2 Scheduling Revisited

Partition the input into **long** jobs and **short** jobs.

A job  $j$  is called short if

$$p_j \leq \frac{1}{km} \sum_i p_i$$

**Idea:**

1. Find the optimum Makespan for the long jobs by brute force.
2. Then use the list scheduling algorithm for the short jobs, always assigning the next job to the least loaded machine.

We still have a cost of

$$\frac{1}{m} \sum_{j \neq \ell} p_j + p_\ell$$

where  $\ell$  is the last job (this only requires that all machines are busy before time  $S_\ell$ ).

We still have a cost of

$$\frac{1}{m} \sum_{j \neq \ell} p_j + p_\ell$$

where  $\ell$  is the last job (this only requires that all machines are busy before time  $S_\ell$ ).

If  $\ell$  is a long job, then the schedule must be optimal, as it consists of an optimal schedule of long jobs plus a schedule for short jobs.

We still have a cost of

$$\frac{1}{m} \sum_{j \neq \ell} p_j + p_\ell$$

where  $\ell$  is the last job (this only requires that all machines are busy before time  $S_\ell$ ).

If  $\ell$  is a long job, then the schedule must be optimal, as it consists of an optimal schedule of long jobs plus a schedule for short jobs.

If  $\ell$  is a short job its length is at most

$$p_\ell \leq \sum_j p_j / (mk)$$

which is at most  $C_{\max}^* / k$ .

Hence we get a schedule of length at most

$$\left(1 + \frac{1}{k}\right) C_{\max}^*$$

There are at most  $km$  long jobs. Hence, the number of possibilities of scheduling these jobs on  $m$  machines is at most  $m^{km}$ , which is constant if  $m$  is constant. Hence, it is easy to implement the algorithm in polynomial time.

### Theorem 75

*The above algorithm gives a polynomial time approximation scheme (PTAS) for the problem of scheduling  $n$  jobs on  $m$  identical machines if  $m$  is constant.*

We choose  $k = \lceil \frac{1}{\epsilon} \rceil$ .

Hence we get a schedule of length at most

$$\left(1 + \frac{1}{k}\right) C_{\max}^*$$

There are at most  $km$  long jobs. Hence, the number of possibilities of scheduling these jobs on  $m$  machines is at most  $m^{km}$ , which is constant if  $m$  is constant. Hence, it is easy to implement the algorithm in polynomial time.

### Theorem 75

*The above algorithm gives a polynomial time approximation scheme (PTAS) for the problem of scheduling  $n$  jobs on  $m$  identical machines if  $m$  is constant.*

We choose  $k = \lceil \frac{1}{\epsilon} \rceil$ .

Hence we get a schedule of length at most

$$\left(1 + \frac{1}{k}\right) C_{\max}^*$$

There are at most  $km$  long jobs. Hence, the number of possibilities of scheduling these jobs on  $m$  machines is at most  $m^{km}$ , which is constant if  $m$  is constant. Hence, it is easy to implement the algorithm in polynomial time.

### Theorem 75

*The above algorithm gives a polynomial time approximation scheme (PTAS) for the problem of scheduling  $n$  jobs on  $m$  identical machines if  $m$  is constant.*

We choose  $k = \lceil \frac{1}{\epsilon} \rceil$ .

## How to get rid of the requirement that $m$ is constant?

We first design an algorithm that works as follows:

On input of  $T$  it either finds a schedule of length  $(1 + \frac{1}{k})T$  or certifies that no schedule of length at most  $T$  exists (assume  $T \geq \frac{1}{m} \sum_j p_j$ ).

We partition the jobs into long jobs and short jobs:

- ▶ A job is long if its size is larger than  $T/k$ .
- ▶ Otw. it is a short job.



How to get rid of the requirement that  $m$  is constant?

We first design an algorithm that works as follows:

On input of  $T$  it either finds a schedule of length  $(1 + \frac{1}{k})T$  or certifies that no schedule of length at most  $T$  exists (assume  $T \geq \frac{1}{m} \sum_j p_j$ ).

We partition the jobs into long jobs and short jobs:

- ▶ A job is long if its size is larger than  $T/k$ .
- ▶ Otw. it is a short job.

How to get rid of the requirement that  $m$  is constant?

We first design an algorithm that works as follows:

On input of  $T$  it either finds a schedule of length  $(1 + \frac{1}{k})T$  or certifies that no schedule of length at most  $T$  exists (assume  $T \geq \frac{1}{m} \sum_j p_j$ ).

We partition the jobs into long jobs and short jobs:

- ▶ A job is long if its size is larger than  $T/k$ .
- ▶ Otw. it is a short job.

How to get rid of the requirement that  $m$  is constant?

We first design an algorithm that works as follows:

On input of  $T$  it either finds a schedule of length  $(1 + \frac{1}{k})T$  or certifies that no schedule of length at most  $T$  exists (assume  $T \geq \frac{1}{m} \sum_j p_j$ ).

We partition the jobs into **long** jobs and **short** jobs:

- ▶ A job is long if its size is larger than  $T/k$ .
- ▶ Otw. it is a short job.

- ▶ We round all **long jobs** down to multiples of  $T/k^2$ .
- ▶ For these rounded sizes we first find an optimal schedule.
- ▶ If this schedule does not have length at most  $T$  we conclude that also the original sizes don't allow such a schedule.
- ▶ If we have a good schedule we extend it by adding the short jobs according to the LPT rule.

- ▶ We round all **long jobs** down to multiples of  $T/k^2$ .
- ▶ For these rounded sizes we first find an optimal schedule.
  - ▶ If this schedule does not have length at most  $T$  we conclude that also the original sizes don't allow such a schedule.
  - ▶ If we have a good schedule we extend it by adding the short jobs according to the LPT rule.

- ▶ We round all **long jobs** down to multiples of  $T/k^2$ .
- ▶ For these rounded sizes we first find an optimal schedule.
- ▶ If this schedule does not have length at most  $T$  we conclude that also the original sizes don't allow such a schedule.
- ▶ If we have a good schedule we extend it by adding the short jobs according to the LPT rule.

- ▶ We round all **long jobs** down to multiples of  $T/k^2$ .
- ▶ For these rounded sizes we first find an optimal schedule.
- ▶ If this schedule does not have length at most  $T$  we conclude that also the original sizes don't allow such a schedule.
- ▶ If we have a good schedule we extend it by adding the short jobs according to the LPT rule.

After the first phase the rounded sizes of the long jobs assigned to a machine add up to at most  $T$ .

There can be at most  $k$  (long) jobs assigned to a machine as otherwise their rounded sizes would add up to more than  $T$  (note that the rounded size of a long job is at least  $T/k$ ).

Since, jobs had been rounded to multiples of  $T/k^2$  going from rounded sizes to original sizes gives that the Makespan is at most

$$\left(1 + \frac{1}{k}\right)T .$$



After the first phase the rounded sizes of the long jobs assigned to a machine add up to at most  $T$ .

There can be at most  $k$  (long) jobs assigned to a machine as otherwise their rounded sizes would add up to more than  $T$  (note that the rounded size of a long job is at least  $T/k$ ).

Since, jobs had been rounded to multiples of  $T/k^2$  going from rounded sizes to original sizes gives that the Makespan is at most

$$\left(1 + \frac{1}{k}\right)T .$$

After the first phase the rounded sizes of the long jobs assigned to a machine add up to at most  $T$ .

There can be at most  $k$  (long) jobs assigned to a machine as otherwise their rounded sizes would add up to more than  $T$  (note that the rounded size of a long job is at least  $T/k$ ).

Since, jobs had been rounded to multiples of  $T/k^2$  going from rounded sizes to original sizes gives that the Makespan is at most

$$\left(1 + \frac{1}{k}\right)T .$$

During the second phase there always must exist a machine with load at most  $T$ , since  $T$  is larger than the average load.

Assigning the current (short) job to such a machine gives that the new load is at most

$$T + \frac{T}{k} \leq \left(1 + \frac{1}{k}\right)T .$$

During the second phase there always must exist a machine with load at most  $T$ , since  $T$  is larger than the average load. Assigning the current (short) job to such a machine gives that the new load is at most

$$T + \frac{T}{k} \leq \left(1 + \frac{1}{k}\right)T .$$

**Running Time for scheduling large jobs:** There should not be a job with rounded size more than  $T$  as otherwise the problem becomes trivial.

Hence, any large job has rounded size of  $\frac{i}{k^2}T$  for  $i \in \{1, \dots, k^2\}$ . Therefore the number of different inputs is at most  $n^{k^2}$  (described by a vector of length  $k^2$  where, the  $i$ -th entry describes the number of jobs of size  $\frac{i}{k^2}T$ ). This is polynomial.

The schedule/configuration of a particular machine  $x$  can be described by a vector of length  $k^2$  where the  $i$ -th entry describes the number of jobs of rounded size  $\frac{i}{k^2}T$  assigned to  $x$ . There are only  $(k + 1)^{k^2}$  different vectors.

This means there are a constant number of different machine configurations.

**Running Time for scheduling large jobs:** There should not be a job with rounded size more than  $T$  as otw. the problem becomes trivial.

Hence, any large job has rounded size of  $\frac{i}{k^2}T$  for  $i \in \{k, \dots, k^2\}$ . Therefore the number of different inputs is at most  $n^{k^2}$  (described by a vector of length  $k^2$  where, the  $i$ -th entry describes the number of jobs of size  $\frac{i}{k^2}T$ ). **This is polynomial.**

The schedule/configuration of a particular machine  $x$  can be described by a vector of length  $k^2$  where the  $i$ -th entry describes the number of jobs of rounded size  $\frac{i}{k^2}T$  assigned to  $x$ . There are only  $(k+1)^{k^2}$  different vectors.

This means there are a constant number of different machine configurations.

**Running Time for scheduling large jobs:** There should not be a job with rounded size more than  $T$  as otherwise the problem becomes trivial.

Hence, any large job has rounded size of  $\frac{i}{k^2}T$  for  $i \in \{k, \dots, k^2\}$ . Therefore the number of different inputs is at most  $n^{k^2}$  (described by a vector of length  $k^2$  where, the  $i$ -th entry describes the number of jobs of size  $\frac{i}{k^2}T$ ). **This is polynomial.**

The schedule/configuration of a particular machine  $x$  can be described by a vector of length  $k^2$  where the  $i$ -th entry describes the number of jobs of rounded size  $\frac{i}{k^2}T$  assigned to  $x$ . There are only  $(k + 1)^{k^2}$  different vectors.

This means there are a constant number of different machine configurations.

**Running Time for scheduling large jobs:** There should not be a job with rounded size more than  $T$  as otherwise the problem becomes trivial.

Hence, any large job has rounded size of  $\frac{i}{k^2}T$  for  $i \in \{k, \dots, k^2\}$ . Therefore the number of different inputs is at most  $n^{k^2}$  (described by a vector of length  $k^2$  where, the  $i$ -th entry describes the number of jobs of size  $\frac{i}{k^2}T$ ). **This is polynomial.**

The schedule/configuration of a particular machine  $x$  can be described by a vector of length  $k^2$  where the  $i$ -th entry describes the number of jobs of rounded size  $\frac{i}{k^2}T$  assigned to  $x$ . There are only  $(k + 1)^{k^2}$  different vectors.

This means there are a **constant** number of different machine configurations.



Let  $\text{OPT}(n_1, \dots, n_{k^2})$  be the **number of machines** that are required to schedule input vector  $(n_1, \dots, n_{k^2})$  with Makespan at most  $T$ .

If  $\text{OPT}(n_1, \dots, n_{k^2}) \leq m$  we can schedule the input.

We have

$$\text{OPT}(n_1, \dots, n_{k^2}) = \begin{cases} 0 & (n_1, \dots, n_{k^2}) = 0 \\ 1 + \min_{(s_1, \dots, s_{k^2}) \in C} \text{OPT}(n_1 - s_1, \dots, n_{k^2} - s_{k^2}) & (n_1, \dots, n_{k^2}) \geq 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

Hence, the running time is roughly  $(k+1)^{k^2} n^{k^2} \approx (nk)^{k^2}$ .

Let  $\text{OPT}(n_1, \dots, n_{k^2})$  be the **number of machines** that are required to schedule input vector  $(n_1, \dots, n_{k^2})$  with Makespan at most  $T$ .

**If  $\text{OPT}(n_1, \dots, n_{k^2}) \leq m$  we can schedule the input.**

We have

$$\text{OPT}(n_1, \dots, n_{k^2}) = \begin{cases} 0 & (n_1, \dots, n_{k^2}) = 0 \\ 1 + \min_{(s_1, \dots, s_{k^2}) \in C} \text{OPT}(n_1 - s_1, \dots, n_{k^2} - s_{k^2}) & (n_1, \dots, n_{k^2}) \geq 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

Hence, the running time is roughly  $(k+1)^{k^2} n^{k^2} \approx (nk)^{k^2}$ .

Let  $\text{OPT}(n_1, \dots, n_{k^2})$  be the **number of machines** that are required to schedule input vector  $(n_1, \dots, n_{k^2})$  with Makespan at most  $T$ .

**If  $\text{OPT}(n_1, \dots, n_{k^2}) \leq m$  we can schedule the input.**

We have

$$\text{OPT}(n_1, \dots, n_{k^2}) = \begin{cases} 0 & (n_1, \dots, n_{k^2}) = 0 \\ 1 + \min_{(s_1, \dots, s_{k^2}) \in C} \text{OPT}(n_1 - s_1, \dots, n_{k^2} - s_{k^2}) & (n_1, \dots, n_{k^2}) \not\geq 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

Hence, the running time is roughly  $(k+1)^{k^2} n^{k^2} \approx (nk)^{k^2}$ .

Let  $\text{OPT}(n_1, \dots, n_{k^2})$  be the **number of machines** that are required to schedule input vector  $(n_1, \dots, n_{k^2})$  with Makespan at most  $T$ .

**If  $\text{OPT}(n_1, \dots, n_{k^2}) \leq m$  we can schedule the input.**

We have

$$\text{OPT}(n_1, \dots, n_{k^2}) = \begin{cases} 0 & (n_1, \dots, n_{k^2}) = 0 \\ 1 + \min_{(s_1, \dots, s_{k^2}) \in C} \text{OPT}(n_1 - s_1, \dots, n_{k^2} - s_{k^2}) & (n_1, \dots, n_{k^2}) \not\equiv 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

Hence, the running time is roughly  $(k+1)^{k^2} n^{k^2} \approx (nk)^{k^2}$ .

We can turn this into a PTAS by choosing  $k = \lceil 1/\epsilon \rceil$  and using binary search. This gives a running time that is exponential in  $1/\epsilon$ .

Can we do better?

Scheduling on identical machines with the goal of minimizing Makespan is a strongly NP-complete problem.

**Theorem 76**

*There is no FPTAS for problems that are strongly NP-hard.*

We can turn this into a PTAS by choosing  $k = \lceil 1/\epsilon \rceil$  and using binary search. This gives a running time that is exponential in  $1/\epsilon$ .

### Can we do better?

Scheduling on identical machines with the goal of minimizing Makespan is a strongly NP-complete problem.

### Theorem 76

*There is no FPTAS for problems that are strongly NP-hard.*

We can turn this into a PTAS by choosing  $k = \lceil 1/\epsilon \rceil$  and using binary search. This gives a running time that is exponential in  $1/\epsilon$ .

### Can we do better?

Scheduling on identical machines with the goal of minimizing Makespan is a **strongly NP-complete** problem.

### Theorem 76

*There is no FPTAS for problems that are strongly NP-hard.*

We can turn this into a PTAS by choosing  $k = \lceil 1/\epsilon \rceil$  and using binary search. This gives a running time that is exponential in  $1/\epsilon$ .

Can we do better?

Scheduling on identical machines with the goal of minimizing Makespan is a **strongly NP-complete** problem.

### Theorem 76

*There is no FPTAS for problems that are strongly NP-hard.*



- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$

- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$

- ▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
- ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
- ▶ For strongly NP-complete problems this is not possible unless P=NP

- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$
- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$

▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
- ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
- ▶ For strongly NP-complete problems this is not possible unless P=NP

- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$
- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$
- ▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
- ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
- ▶ For strongly NP-complete problems this is not possible unless P=NP

- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$
- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$
- ▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
- ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
- ▶ For strongly NP-complete problems this is not possible unless P=NP

- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$
- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$
- ▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
  - ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
  - ▶ For strongly NP-complete problems this is not possible unless P=NP

- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$
- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$
- ▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
- ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
- ▶ For strongly NP-complete problems this is not possible unless P=NP

- ▶ Suppose we have an instance with polynomially bounded processing times  $p_i \leq q(n)$
- ▶ We set  $k := \lceil 2nq(n) \rceil \geq 2 \text{OPT}$
- ▶ Then

$$\text{ALG} \leq \left(1 + \frac{1}{k}\right) \text{OPT} \leq \text{OPT} + \frac{1}{2}$$

- ▶ But this means that the algorithm computes the optimal solution as the optimum is integral.
- ▶ This means we can solve problem instances if processing times are polynomially bounded
- ▶ Running time is  $\mathcal{O}(\text{poly}(n, k)) = \mathcal{O}(\text{poly}(n))$
- ▶ For strongly NP-complete problems this is not possible unless P=NP

## More General

Let  $\text{OPT}(n_1, \dots, n_A)$  be the number of machines that are required to schedule input vector  $(n_1, \dots, n_A)$  with Makespan at most  $T$  ( $A$ : number of different sizes).

If  $\text{OPT}(n_1, \dots, n_A) \leq m$  we can schedule the input.

$\text{OPT}(n_1, \dots, n_A)$

$$= \begin{cases} 0 & (n_1, \dots, n_A) = 0 \\ 1 + \min_{(s_1, \dots, s_A) \in C} \text{OPT}(n_1 - s_1, \dots, n_A - s_A) & (n_1, \dots, n_A) \geq 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

$|C| \leq (B + 1)^A$ , where  $B$  is the number of jobs that possibly can fit on the same machine.

The running time is then  $O((B + 1)^A n^A)$  because the dynamic programming table has just  $n^A$  entries.



## More General

Let  $OPT(n_1, \dots, n_A)$  be the number of machines that are required to schedule input vector  $(n_1, \dots, n_A)$  with Makespan at most  $T$  ( $A$ : number of different sizes).

If  $OPT(n_1, \dots, n_A) \leq m$  we can schedule the input.

$$OPT(n_1, \dots, n_A) = \begin{cases} 0 & (n_1, \dots, n_A) = 0 \\ 1 + \min_{(s_1, \dots, s_A) \in C} OPT(n_1 - s_1, \dots, n_A - s_A) & (n_1, \dots, n_A) \geq 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

$|C| \leq (B + 1)^A$ , where  $B$  is the number of jobs that possibly can fit on the same machine.

The running time is then  $O((B + 1)^A n^A)$  because the dynamic programming table has just  $n^A$  entries.

## More General

Let  $\text{OPT}(n_1, \dots, n_A)$  be the number of machines that are required to schedule input vector  $(n_1, \dots, n_A)$  with Makespan at most  $T$  ( $A$ : number of different sizes).

If  $\text{OPT}(n_1, \dots, n_A) \leq m$  we can schedule the input.

$$\text{OPT}(n_1, \dots, n_A) = \begin{cases} 0 & (n_1, \dots, n_A) = 0 \\ 1 + \min_{(s_1, \dots, s_A) \in C} \text{OPT}(n_1 - s_1, \dots, n_A - s_A) & (n_1, \dots, n_A) \not\geq 0 \\ \infty & \text{otw.} \end{cases}$$

where  $C$  is the set of all configurations.

$|C| \leq (B + 1)^A$ , where  $B$  is the number of jobs that possibly can fit on the same machine.

The running time is then  $O((B + 1)^A n^A)$  because the dynamic programming table has just  $n^A$  entries.

# Bin Packing

Given  $n$  items with sizes  $s_1, \dots, s_n$  where

$$1 > s_1 \geq \dots \geq s_n > 0 .$$

Pack items into a minimum number of bins where each bin can hold items of total size at most 1.

## Theorem 77

*There is no  $\rho$ -approximation for Bin Packing with  $\rho < 3/2$  unless  $P = NP$ .*

# Bin Packing

Given  $n$  items with sizes  $s_1, \dots, s_n$  where

$$1 > s_1 \geq \dots \geq s_n > 0 .$$

Pack items into a minimum number of bins where each bin can hold items of total size at most 1.

## Theorem 77

*There is no  $\rho$ -approximation for Bin Packing with  $\rho < 3/2$  unless  $P = NP$ .*

# Bin Packing

## Proof

- ▶ In the partition problem we are given positive integers  $b_1, \dots, b_n$  with  $B = \sum_i b_i$  even. Can we partition the integers into two sets  $S$  and  $T$  s.t.

$$\sum_{i \in S} b_i = \sum_{i \in T} b_i \quad ?$$

- ▶ We can solve this problem by setting  $s_i := 2b_i/B$  and asking whether we can pack the resulting items into 2 bins or not.
- ▶ A  $\rho$ -approximation algorithm with  $\rho < 3/2$  cannot output 3 or more bins when 2 are optimal.
- ▶ Hence, such an algorithm can solve Partition.

# Bin Packing

## Proof

- ▶ In the partition problem we are given positive integers  $b_1, \dots, b_n$  with  $B = \sum_i b_i$  even. Can we partition the integers into two sets  $S$  and  $T$  s.t.

$$\sum_{i \in S} b_i = \sum_{i \in T} b_i \quad ?$$

- ▶ We can solve this problem by setting  $s_i := 2b_i/B$  and asking whether we can pack the resulting items into 2 bins or not.
- ▶ A  $\rho$ -approximation algorithm with  $\rho < 3/2$  cannot output 3 or more bins when 2 are optimal.
- ▶ Hence, such an algorithm can solve Partition.

# Bin Packing

## Proof

- ▶ In the partition problem we are given positive integers  $b_1, \dots, b_n$  with  $B = \sum_i b_i$  even. Can we partition the integers into two sets  $S$  and  $T$  s.t.

$$\sum_{i \in S} b_i = \sum_{i \in T} b_i \quad ?$$

- ▶ We can solve this problem by setting  $s_i := 2b_i/B$  and asking whether we can pack the resulting items into 2 bins or not.
- ▶ A  $\rho$ -approximation algorithm with  $\rho < 3/2$  cannot output 3 or more bins when 2 are optimal.
- ▶ Hence, such an algorithm can solve Partition.

# Bin Packing

## Proof

- ▶ In the partition problem we are given positive integers  $b_1, \dots, b_n$  with  $B = \sum_i b_i$  even. Can we partition the integers into two sets  $S$  and  $T$  s.t.

$$\sum_{i \in S} b_i = \sum_{i \in T} b_i \quad ?$$

- ▶ We can solve this problem by setting  $s_i := 2b_i/B$  and asking whether we can pack the resulting items into 2 bins or not.
- ▶ A  $\rho$ -approximation algorithm with  $\rho < 3/2$  cannot output 3 or more bins when 2 are optimal.
- ▶ Hence, such an algorithm can solve Partition.



## Definition 78

An asymptotic polynomial-time approximation scheme (APTAS) is a family of algorithms  $\{A_\epsilon\}$  along with a constant  $c$  such that  $A_\epsilon$  returns a solution of value at most  $(1 + \epsilon)\text{OPT} + c$  for minimization problems.

## Definition 78

An asymptotic polynomial-time approximation scheme (APTAS) is a family of algorithms  $\{A_\epsilon\}$  along with a constant  $c$  such that  $A_\epsilon$  returns a solution of value at most  $(1 + \epsilon)\text{OPT} + c$  for minimization problems.

- ▶ Note that for Set Cover or for Knapsack it makes no sense to differentiate between the notion of a PTAS or an APTAS because of scaling.
- ▶ However, we will develop an APTAS for Bin Packing.

## Definition 78

An asymptotic polynomial-time approximation scheme (APTAS) is a family of algorithms  $\{A_\epsilon\}$  along with a constant  $c$  such that  $A_\epsilon$  returns a solution of value at most  $(1 + \epsilon)\text{OPT} + c$  for minimization problems.

- ▶ Note that for Set Cover or for Knapsack it makes no sense to differentiate between the notion of a PTAS or an APTAS because of scaling.
- ▶ However, we will develop an APTAS for Bin Packing.

# Bin Packing

Again we can differentiate between small and large items.

## Lemma 79

Any packing of items into  $\ell$  bins can be extended with items of size at most  $\gamma$  s.t. we use only  $\max\{\ell, \frac{1}{1-\gamma}\text{SIZE}(I) + 1\}$  bins, where  $\text{SIZE}(I) = \sum_i s_i$  is the sum of all item sizes.

# Bin Packing

Again we can differentiate between small and large items.

## Lemma 79

Any packing of items into  $\ell$  bins can be extended with items of size at most  $\gamma$  s.t. we use only  $\max\{\ell, \frac{1}{1-\gamma}\text{SIZE}(I) + 1\}$  bins, where  $\text{SIZE}(I) = \sum_i s_i$  is the sum of all item sizes.

- ▶ If after Greedy we use more than  $\ell$  bins, all bins (apart from the last) must be full to at least  $1 - \gamma$ .
- ▶ Hence,  $r(1 - \gamma) \leq \text{SIZE}(I)$  where  $r$  is the number of nearly-full bins.
- ▶ This gives the lemma.

# Bin Packing

Again we can differentiate between small and large items.

## Lemma 79

Any packing of items into  $\ell$  bins can be extended with items of size at most  $\gamma$  s.t. we use only  $\max\{\ell, \frac{1}{1-\gamma}\text{SIZE}(I) + 1\}$  bins, where  $\text{SIZE}(I) = \sum_i s_i$  is the sum of all item sizes.

- ▶ If after Greedy we use more than  $\ell$  bins, all bins (apart from the last) must be full to at least  $1 - \gamma$ .
- ▶ Hence,  $r(1 - \gamma) \leq \text{SIZE}(I)$  where  $r$  is the number of nearly-full bins.
- ▶ This gives the lemma.

# Bin Packing

Again we can differentiate between small and large items.

## Lemma 79

Any packing of items into  $\ell$  bins can be extended with items of size at most  $\gamma$  s.t. we use only  $\max\{\ell, \frac{1}{1-\gamma}\text{SIZE}(I) + 1\}$  bins, where  $\text{SIZE}(I) = \sum_i s_i$  is the sum of all item sizes.

- ▶ If after Greedy we use more than  $\ell$  bins, all bins (apart from the last) must be full to at least  $1 - \gamma$ .
- ▶ Hence,  $r(1 - \gamma) \leq \text{SIZE}(I)$  where  $r$  is the number of nearly-full bins.
- ▶ This gives the lemma.

Choose  $\gamma = \epsilon/2$ . Then we either use  $\ell$  bins or at most

$$\frac{1}{1 - \epsilon/2} \cdot \text{OPT} + 1 \leq (1 + \epsilon) \cdot \text{OPT} + 1$$

bins.

It remains to find an algorithm for the large items.



## Linear Grouping:

Generate an instance  $I'$  (for large items) as follows.

- ▶ Order large items according to size.
- ▶ Let the first  $k$  items belong to group 1; the following  $k$  items belong to group 2; etc.
- ▶ Delete items in the first group;
- ▶ Round items in the remaining groups to the size of the largest item in the group.

## Linear Grouping:

Generate an instance  $I'$  (for large items) as follows.

- ▶ Order large items according to size.
- ▶ Let the first  $k$  items belong to group 1; the following  $k$  items belong to group 2; etc.
- ▶ Delete items in the first group;
- ▶ Round items in the remaining groups to the size of the largest item in the group.

## Linear Grouping:

Generate an instance  $I'$  (for large items) as follows.

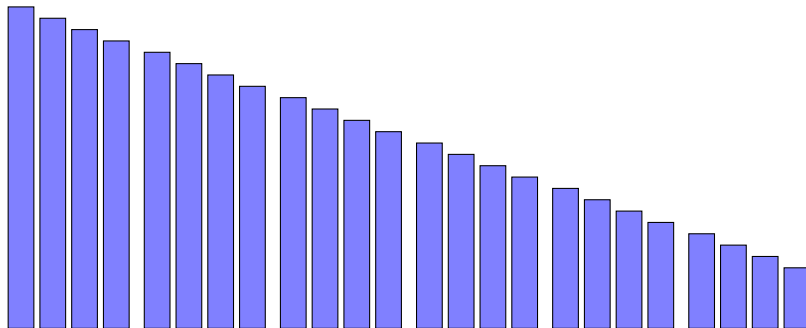
- ▶ Order large items according to size.
- ▶ Let the first  $k$  items belong to group 1; the following  $k$  items belong to group 2; etc.
- ▶ Delete items in the first group;
- ▶ Round items in the remaining groups to the size of the largest item in the group.

## Linear Grouping:

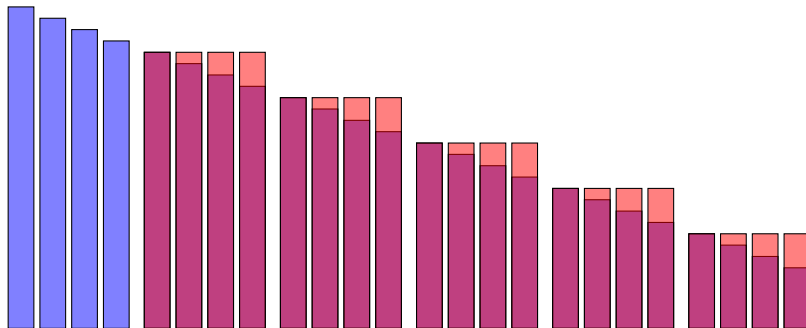
Generate an instance  $I'$  (for large items) as follows.

- ▶ Order large items according to size.
- ▶ Let the first  $k$  items belong to group 1; the following  $k$  items belong to group 2; etc.
- ▶ Delete items in the first group;
- ▶ Round items in the remaining groups to the size of the largest item in the group.

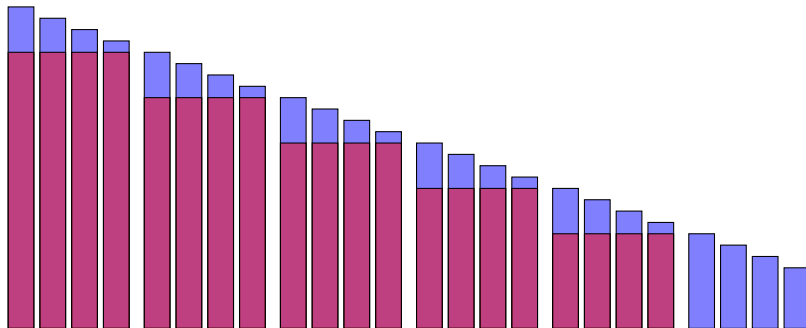
# Linear Grouping



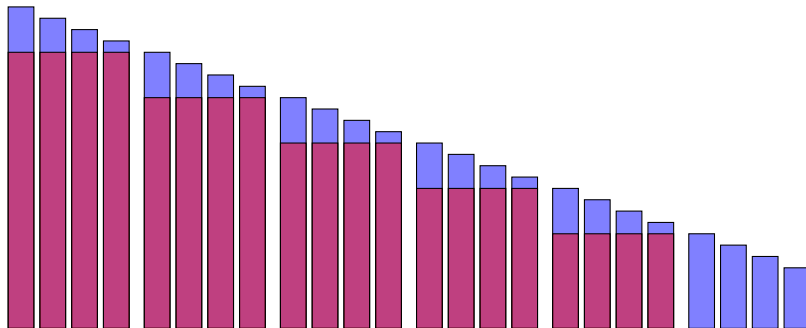
# Linear Grouping



# Linear Grouping



# Linear Grouping





## Lemma 80

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

Proof 1:

Any bin packing for  $I$  gives a bin packing for  $I'$  as follows:

pack the items of group 1 into the bins for  $I'$  (the items for group 1 have been packed).

pack the items of groups 2, where in the packing for  $I$  the items for group 2 have been packed.

## Lemma 80

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 1:

- ▶ Any bin packing for  $I$  gives a bin packing for  $I'$  as follows.
- ▶ Pack the items of group 2, where in the packing for  $I$  the items for group 1 have been packed;
- ▶ Pack the items of groups 3, where in the packing for  $I$  the items for group 2 have been packed;
- ▶ ...

## Lemma 80

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 1:

- ▶ Any bin packing for  $I$  gives a bin packing for  $I'$  as follows.
- ▶ Pack the items of group 2, where in the packing for  $I$  the items for group 1 have been packed;
- ▶ Pack the items of groups 3, where in the packing for  $I$  the items for group 2 have been packed;
- ▶ ...

## Lemma 80

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 1:

- ▶ Any bin packing for  $I$  gives a bin packing for  $I'$  as follows.
- ▶ Pack the items of group 2, where in the packing for  $I$  the items for group 1 have been packed;
- ▶ Pack the items of groups 3, where in the packing for  $I$  the items for group 2 have been packed;
- ▶ ....

## Lemma 80

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 1:

- ▶ Any bin packing for  $I$  gives a bin packing for  $I'$  as follows.
- ▶ Pack the items of group 2, where in the packing for  $I$  the items for group 1 have been packed;
- ▶ Pack the items of groups 3, where in the packing for  $I$  the items for group 2 have been packed;
- ▶ ...

## Lemma 81

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 2:

- ▶ Any bin packing for  $I'$  gives a bin packing for  $I$  as follows.
- ▶ Pack the items of group 1 into  $k$  new bins;
- ▶ Pack the items of groups 2, where in the packing for  $I'$  the items for group 2 have been packed;
- ▶ ...

## Lemma 81

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 2:

- ▶ Any bin packing for  $I'$  gives a bin packing for  $I$  as follows.
- ▶ Pack the items of group 1 into  $k$  new bins;
- ▶ Pack the items of groups 2, where in the packing for  $I'$  the items for group 2 have been packed;
- ▶ ...

## Lemma 81

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 2:

- ▶ Any bin packing for  $I'$  gives a bin packing for  $I$  as follows.
- ▶ Pack the items of group 1 into  $k$  new bins;
- ▶ Pack the items of groups 2, where in the packing for  $I'$  the items for group 2 have been packed;
- ▶ ....



## Lemma 81

$$\text{OPT}(I') \leq \text{OPT}(I) \leq \text{OPT}(I') + k$$

### Proof 2:

- ▶ Any bin packing for  $I'$  gives a bin packing for  $I$  as follows.
- ▶ Pack the items of group 1 into  $k$  new bins;
- ▶ Pack the items of groups 2, where in the packing for  $I'$  the items for group 2 have been packed;
- ▶ ...

Assume that our instance does not contain pieces smaller than  $\epsilon/2$ . Then  $\text{SIZE}(I) \geq \epsilon n/2$ .

We set  $k = \lfloor \epsilon \text{SIZE}(I) \rfloor$ .

Then  $n/k \leq n/\lfloor \epsilon^2 n/2 \rfloor \leq 4/\epsilon^2$  (note that  $\lfloor \alpha \rfloor \geq \alpha/2$  for  $\alpha \geq 1$ ).

Hence, after grouping we have a constant number of piece sizes ( $4/\epsilon^2$ ) and at most a constant number ( $2/\epsilon$ ) can fit into any bin.

We can find an optimal packing for such instances by the previous Dynamic Programming approach.

- ▶ cost (for large items) at most

$$\text{OPT}(I') + k \leq \text{OPT}(I) + \epsilon \text{SIZE}(I) \leq (1 + \epsilon) \text{OPT}(I)$$

- ▶ running time  $\mathcal{O}((\frac{2}{\epsilon}n)^{4/\epsilon^2})$ .

Assume that our instance does not contain pieces smaller than  $\epsilon/2$ . Then  $\text{SIZE}(I) \geq \epsilon n/2$ .

We set  $k = \lfloor \epsilon \text{SIZE}(I) \rfloor$ .

Then  $n/k \leq n/\lfloor \epsilon^2 n/2 \rfloor \leq 4/\epsilon^2$  (note that  $\lfloor \alpha \rfloor \geq \alpha/2$  for  $\alpha \geq 1$ ).

Hence, after grouping we have a constant number of piece sizes ( $4/\epsilon^2$ ) and at most a constant number ( $2/\epsilon$ ) can fit into any bin.

We can find an optimal packing for such instances by the previous Dynamic Programming approach.

- ▶ cost (for large items) at most

$$\text{OPT}(I') + k \leq \text{OPT}(I) + \epsilon \text{SIZE}(I) \leq (1 + \epsilon) \text{OPT}(I)$$

- ▶ running time  $\mathcal{O}((\frac{2}{\epsilon}n)^{4/\epsilon^2})$ .

Assume that our instance does not contain pieces smaller than  $\epsilon/2$ . Then  $\text{SIZE}(I) \geq \epsilon n/2$ .

We set  $k = \lfloor \epsilon \text{SIZE}(I) \rfloor$ .

Then  $n/k \leq n/\lfloor \epsilon^2 n/2 \rfloor \leq 4/\epsilon^2$  (note that  $\lfloor \alpha \rfloor \geq \alpha/2$  for  $\alpha \geq 1$ ).

Hence, after grouping we have a constant number of piece sizes ( $4/\epsilon^2$ ) and at most a constant number ( $2/\epsilon$ ) can fit into any bin.

We can find an optimal packing for such instances by the previous Dynamic Programming approach.

- ▶ cost (for large items) at most

$$\text{OPT}(I') + k \leq \text{OPT}(I) + \epsilon \text{SIZE}(I) \leq (1 + \epsilon) \text{OPT}(I)$$

- ▶ running time  $\mathcal{O}((\frac{2}{\epsilon}n)^{4/\epsilon^2})$ .

Assume that our instance does not contain pieces smaller than  $\epsilon/2$ . Then  $\text{SIZE}(I) \geq \epsilon n/2$ .

We set  $k = \lfloor \epsilon \text{SIZE}(I) \rfloor$ .

Then  $n/k \leq n/\lfloor \epsilon^2 n/2 \rfloor \leq 4/\epsilon^2$  (note that  $\lfloor \alpha \rfloor \geq \alpha/2$  for  $\alpha \geq 1$ ).

Hence, after grouping we have a constant number of piece sizes ( $4/\epsilon^2$ ) and at most a constant number ( $2/\epsilon$ ) can fit into any bin.

We can find an optimal packing for such instances by the previous Dynamic Programming approach.

- ▶ cost (for large items) at most

$$\text{OPT}(I') + k \leq \text{OPT}(I) + \epsilon \text{SIZE}(I) \leq (1 + \epsilon) \text{OPT}(I)$$

- ▶ running time  $\mathcal{O}((\frac{2}{\epsilon} n)^{4/\epsilon^2})$ .

Assume that our instance does not contain pieces smaller than  $\epsilon/2$ . Then  $\text{SIZE}(I) \geq \epsilon n/2$ .

We set  $k = \lfloor \epsilon \text{SIZE}(I) \rfloor$ .

Then  $n/k \leq n/\lfloor \epsilon^2 n/2 \rfloor \leq 4/\epsilon^2$  (note that  $\lfloor \alpha \rfloor \geq \alpha/2$  for  $\alpha \geq 1$ ).

Hence, after grouping we have a constant number of piece sizes ( $4/\epsilon^2$ ) and at most a constant number ( $2/\epsilon$ ) can fit into any bin.

We can find an optimal packing for such instances by the previous Dynamic Programming approach.

- ▶ cost (for large items) at most

$$\text{OPT}(I') + k \leq \text{OPT}(I) + \epsilon \text{SIZE}(I) \leq (1 + \epsilon) \text{OPT}(I)$$

- ▶ running time  $\mathcal{O}((\frac{2}{\epsilon} n)^{4/\epsilon^2})$ .

Assume that our instance does not contain pieces smaller than  $\epsilon/2$ . Then  $\text{SIZE}(I) \geq \epsilon n/2$ .

We set  $k = \lfloor \epsilon \text{SIZE}(I) \rfloor$ .

Then  $n/k \leq n/\lfloor \epsilon^2 n/2 \rfloor \leq 4/\epsilon^2$  (note that  $\lfloor \alpha \rfloor \geq \alpha/2$  for  $\alpha \geq 1$ ).

Hence, after grouping we have a constant number of piece sizes ( $4/\epsilon^2$ ) and at most a constant number ( $2/\epsilon$ ) can fit into any bin.

We can find an optimal packing for such instances by the previous Dynamic Programming approach.

- ▶ cost (for large items) at most

$$\text{OPT}(I') + k \leq \text{OPT}(I) + \epsilon \text{SIZE}(I) \leq (1 + \epsilon) \text{OPT}(I)$$

- ▶ running time  $\mathcal{O}((\frac{2}{\epsilon}n)^{4/\epsilon^2})$ .

## Can we do better?

In the following we show how to obtain a solution where the number of bins is only

$$\text{OPT}(I) + \mathcal{O}(\log^2(\text{SIZE}(I))) .$$

Note that this is usually better than a guarantee of

$$(1 + \epsilon)\text{OPT}(I) + 1 .$$



## Can we do better?

In the following we show how to obtain a solution where the number of bins is only

$$\text{OPT}(I) + \mathcal{O}(\log^2(\text{SIZE}(I))) .$$

Note that this is usually better than a guarantee of

$$(1 + \epsilon)\text{OPT}(I) + 1 .$$

## Can we do better?

In the following we show how to obtain a solution where the number of bins is only

$$\text{OPT}(I) + \mathcal{O}(\log^2(\text{SIZE}(I))) .$$

Note that this is usually better than a guarantee of

$$(1 + \epsilon)\text{OPT}(I) + 1 .$$

# Configuration LP

## Change of Notation:

- ▶ Group pieces of identical size.
- ▶ Let  $s_1$  denote the largest size, and let  $b_1$  denote the number of pieces of size  $s_1$ .
- ▶  $s_2$  is second largest size and  $b_2$  number of pieces of size  $s_2$ ;
- ▶ ...
- ▶  $s_m$  smallest size and  $b_m$  number of pieces of size  $s_m$ .

# Configuration LP

## Change of Notation:

- ▶ Group pieces of identical size.
- ▶ Let  $s_1$  denote the largest size, and let  $b_1$  denote the number of pieces of size  $s_1$ .
- ▶  $s_2$  is second largest size and  $b_2$  number of pieces of size  $s_2$ ;
- ▶ ...
- ▶  $s_m$  smallest size and  $b_m$  number of pieces of size  $s_m$ .

# Configuration LP

## Change of Notation:

- ▶ Group pieces of identical size.
- ▶ Let  $s_1$  denote the largest size, and let  $b_1$  denote the number of pieces of size  $s_1$ .
- ▶  $s_2$  is second largest size and  $b_2$  number of pieces of size  $s_2$ ;
- ▶ ...
- ▶  $s_m$  smallest size and  $b_m$  number of pieces of size  $s_m$ .

# Configuration LP

## Change of Notation:

- ▶ Group pieces of identical size.
- ▶ Let  $s_1$  denote the largest size, and let  $b_1$  denote the number of pieces of size  $s_1$ .
- ▶  $s_2$  is second largest size and  $b_2$  number of pieces of size  $s_2$ ;
- ▶ ...
- ▶  $s_m$  smallest size and  $b_m$  number of pieces of size  $s_m$ .

## Change of Notation:

- ▶ Group pieces of identical size.
- ▶ Let  $s_1$  denote the largest size, and let  $b_1$  denote the number of pieces of size  $s_1$ .
- ▶  $s_2$  is second largest size and  $b_2$  number of pieces of size  $s_2$ ;
- ▶ ...
- ▶  $s_m$  smallest size and  $b_m$  number of pieces of size  $s_m$ .

# Configuration LP

A possible packing of a bin can be described by an  $m$ -tuple  $(t_1, \dots, t_m)$ , where  $t_i$  describes the number of pieces of size  $s_i$ .

Clearly,

$$\sum_i t_i \cdot s_i \leq 1.$$

We call a vector that fulfills the above constraint a **configuration**.



# Configuration LP

A possible packing of a bin can be described by an  $m$ -tuple  $(t_1, \dots, t_m)$ , where  $t_i$  describes the number of pieces of size  $s_i$ . Clearly,

$$\sum_i t_i \cdot s_i \leq 1 .$$

We call a vector that fulfills the above constraint a **configuration**.

# Configuration LP

A possible packing of a bin can be described by an  $m$ -tuple  $(t_1, \dots, t_m)$ , where  $t_i$  describes the number of pieces of size  $s_i$ . Clearly,

$$\sum_i t_i \cdot s_i \leq 1 .$$

We call a vector that fulfills the above constraint a **configuration**.

# Configuration LP

Let  $N$  be the number of configurations (exponential).

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \\ & \forall j \in \{1, \dots, N\} \quad x_j \text{ integral} \end{array}$$

# Configuration LP

Let  $N$  be the number of configurations (**exponential**).

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \\ & \forall j \in \{1, \dots, N\} \quad x_j \text{ integral} \end{array}$$

# Configuration LP

Let  $N$  be the number of configurations (**exponential**).

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \\ & \forall j \in \{1, \dots, N\} \quad x_j \text{ integral} \end{array}$$

# Configuration LP

Let  $N$  be the number of configurations (**exponential**).

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \\ & \forall j \in \{1, \dots, N\} \quad x_j \text{ integral} \end{array}$$

**How to solve this LP?**

later...

We can assume that each item has size at least  $1/\text{SIZE}(I)$ .



# Harmonic Grouping

- ▶ Sort items according to size (monotonically decreasing).
- ▶ Process items in this order; close the current group if size of items in the group is at least 2 (or larger). Then open new group.
- ▶ I.e.,  $G_1$  is the smallest cardinality set of largest items s.t. total size sums up to at least 2. Similarly, for  $G_2, \dots, G_{r-1}$ .
- ▶ Only the size of items in the last group  $G_r$  may sum up to less than 2.

# Harmonic Grouping

- ▶ Sort items according to size (monotonically decreasing).
- ▶ Process items in this order; close the current group if size of items in the group is at least 2 (or larger). Then open new group.
- ▶ I.e.,  $G_1$  is the smallest cardinality set of largest items s.t. total size sums up to at least 2. Similarly, for  $G_2, \dots, G_{r-1}$ .
- ▶ Only the size of items in the last group  $G_r$  may sum up to less than 2.

# Harmonic Grouping

- ▶ Sort items according to size (monotonically decreasing).
- ▶ Process items in this order; close the current group if size of items in the group is at least 2 (or larger). Then open new group.
- ▶ I.e.,  $G_1$  is the smallest cardinality set of largest items s.t. total size sums up to at least 2. Similarly, for  $G_2, \dots, G_{r-1}$ .
- ▶ Only the size of items in the last group  $G_r$  may sum up to less than 2.

# Harmonic Grouping

- ▶ Sort items according to size (monotonically decreasing).
- ▶ Process items in this order; close the current group if size of items in the group is at least 2 (or larger). Then open new group.
- ▶ I.e.,  $G_1$  is the smallest cardinality set of largest items s.t. total size sums up to at least 2. Similarly, for  $G_2, \dots, G_{r-1}$ .
- ▶ Only the size of items in the last group  $G_r$  may sum up to less than 2.

# Harmonic Grouping

From the grouping we obtain instance  $I'$  as follows:

- ▶ Round all items in a group to the size of the largest group member.
- ▶ Delete all items from group  $G_1$  and  $G_r$ .
- ▶ For groups  $G_2, \dots, G_{r-1}$  delete  $n_i - n_{i-1}$  items.
- ▶ Observe that  $n_i \geq n_{i-1}$ .

# Harmonic Grouping

From the grouping we obtain instance  $I'$  as follows:

- ▶ Round all items in a group to the size of the largest group member.
- ▶ Delete all items from group  $G_1$  and  $G_r$ .
- ▶ For groups  $G_2, \dots, G_{r-1}$  delete  $n_i - n_{i-1}$  items.
- ▶ Observe that  $n_i \geq n_{i-1}$ .

# Harmonic Grouping

From the grouping we obtain instance  $I'$  as follows:

- ▶ Round all items in a group to the size of the largest group member.
- ▶ Delete all items from group  $G_1$  and  $G_r$ .
- ▶ For groups  $G_2, \dots, G_{r-1}$  delete  $n_i - n_{i-1}$  items.
- ▶ Observe that  $n_i \geq n_{i-1}$ .

# Harmonic Grouping

From the grouping we obtain instance  $I'$  as follows:

- ▶ Round all items in a group to the size of the largest group member.
- ▶ Delete all items from group  $G_1$  and  $G_r$ .
- ▶ For groups  $G_2, \dots, G_{r-1}$  delete  $n_i - n_{i-1}$  items.
- ▶ Observe that  $n_i \geq n_{i-1}$ .



## Lemma 82

*The number of different sizes in  $I'$  is at most  $\text{SIZE}(I)/2$ .*

## Lemma 82

*The number of different sizes in  $I'$  is at most  $\text{SIZE}(I)/2$ .*

- ▶ Each group that survives (recall that  $G_1$  and  $G_r$  are deleted) has total size at least 2.
- ▶ Hence, the number of surviving groups is at most  $\text{SIZE}(I)/2$ .
- ▶ All items in a group have the same size in  $I'$ .

## Lemma 82

*The number of different sizes in  $I'$  is at most  $\text{SIZE}(I)/2$ .*

- ▶ Each group that survives (recall that  $G_1$  and  $G_r$  are deleted) has total size at least 2.
- ▶ Hence, the number of surviving groups is at most  $\text{SIZE}(I)/2$ .
- ▶ All items in a group have the same size in  $I'$ .

## Lemma 82

*The number of different sizes in  $I'$  is at most  $\text{SIZE}(I)/2$ .*

- ▶ Each group that survives (recall that  $G_1$  and  $G_r$  are deleted) has total size at least 2.
- ▶ Hence, the number of surviving groups is at most  $\text{SIZE}(I)/2$ .
- ▶ All items in a group have the same size in  $I'$ .

## Lemma 83

*The total size of deleted items is at most  $\mathcal{O}(\log(\text{SIZE}(I)))$ .*

## Lemma 83

The total size of deleted items is at most  $\mathcal{O}(\log(\text{SIZE}(I)))$ .

- ▶ The total size of items in  $G_1$  and  $G_r$  is at most 6 as a group has total size at most 3.
- ▶ Consider a group  $G_i$  that has strictly more items than  $G_{i-1}$ .
- ▶ It discards  $n_i - n_{i-1}$  pieces of total size at most

$$3 \frac{n_i - n_{i-1}}{n_i} \leq \sum_{j=n_{i-1}+1}^{n_i} \frac{3}{j}$$

since the average piece size is only  $3/n_i$ .

- ▶ Summing over all  $i$  that have  $n_i > n_{i-1}$  gives a bound of at most

$$\sum_{j=1}^{n_{r-1}} \frac{3}{j} \leq \mathcal{O}(\log(\text{SIZE}(I))) .$$

(note that  $n_r \leq \text{SIZE}(I)$  since we assume that the size of each item is at least  $1/\text{SIZE}(I)$ ).

## Lemma 83

The total size of deleted items is at most  $\mathcal{O}(\log(\text{SIZE}(I)))$ .

- ▶ The total size of items in  $G_1$  and  $G_r$  is at most 6 as a group has total size at most 3.
- ▶ Consider a group  $G_i$  that has strictly more items than  $G_{i-1}$ .
- ▶ It discards  $n_i - n_{i-1}$  pieces of total size at most

$$3 \frac{n_i - n_{i-1}}{n_i} \leq \sum_{j=n_{i-1}+1}^{n_i} \frac{3}{j}$$

since the average piece size is only  $3/n_i$ .

- ▶ Summing over all  $i$  that have  $n_i > n_{i-1}$  gives a bound of at most

$$\sum_{j=1}^{n_{r-1}} \frac{3}{j} \leq \mathcal{O}(\log(\text{SIZE}(I))) .$$

(note that  $n_r \leq \text{SIZE}(I)$  since we assume that the size of each item is at least  $1/\text{SIZE}(I)$ ).

### Lemma 83

The total size of deleted items is at most  $\mathcal{O}(\log(\text{SIZE}(I)))$ .

- ▶ The total size of items in  $G_1$  and  $G_r$  is at most 6 as a group has total size at most 3.
- ▶ Consider a group  $G_i$  that has strictly more items than  $G_{i-1}$ .
- ▶ It discards  $n_i - n_{i-1}$  pieces of total size at most

$$3 \frac{n_i - n_{i-1}}{n_i} \leq \sum_{j=n_{i-1}+1}^{n_i} \frac{3}{j}$$

since the average piece size is only  $3/n_i$ .

- ▶ Summing over all  $i$  that have  $n_i > n_{i-1}$  gives a bound of at most

$$\sum_{j=1}^{n_r-1} \frac{3}{j} \leq \mathcal{O}(\log(\text{SIZE}(I))) .$$

(note that  $n_r \leq \text{SIZE}(I)$  since we assume that the size of each item is at least  $1/\text{SIZE}(I)$ ).



## Lemma 83

The total size of deleted items is at most  $\mathcal{O}(\log(\text{SIZE}(I)))$ .

- ▶ The total size of items in  $G_1$  and  $G_r$  is at most 6 as a group has total size at most 3.
- ▶ Consider a group  $G_i$  that has strictly more items than  $G_{i-1}$ .
- ▶ It discards  $n_i - n_{i-1}$  pieces of total size at most

$$3 \frac{n_i - n_{i-1}}{n_i} \leq \sum_{j=n_{i-1}+1}^{n_i} \frac{3}{j}$$

since the average piece size is only  $3/n_i$ .

- ▶ Summing over all  $i$  that have  $n_i > n_{i-1}$  gives a bound of at most

$$\sum_{j=1}^{n_{r-1}} \frac{3}{j} \leq \mathcal{O}(\log(\text{SIZE}(I))) .$$

(note that  $n_r \leq \text{SIZE}(I)$  since we assume that the size of each item is at least  $1/\text{SIZE}(I)$ ).

### Algorithm 1 BinPack

- 1: **if**  $\text{SIZE}(I) < 10$  **then**
- 2:     pack remaining items greedily
- 3: Apply harmonic grouping to create instance  $I'$ ; pack discarded items in at most  $\mathcal{O}(\log(\text{SIZE}(I)))$  bins.
- 4: Let  $x$  be optimal solution to configuration LP
- 5: Pack  $\lfloor x_j \rfloor$  bins in configuration  $T_j$  for all  $j$ ; call the packed instance  $I_1$ .
- 6: Let  $I_2$  be remaining pieces from  $I'$
- 7: Pack  $I_2$  via  $\text{BinPack}(I_2)$

$$\text{OPT}_{\text{LP}}(I_1) + \text{OPT}_{\text{LP}}(I_2) \leq \text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$$

Proof:

Each piece appearing in  $I'$  can be traced to a piece in  $I_1$  or  $I_2$  of no larger size. Hence,

$\text{OPT}_{\text{LP}}(I')$  is feasible solution for  $I_1$  and  $I_2$ .

$\text{OPT}_{\text{LP}}(I) \geq \text{OPT}_{\text{LP}}(I')$

$$\text{OPT}_{\text{LP}}(I_1) + \text{OPT}_{\text{LP}}(I_2) \leq \text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$$

## Proof:

- ▶ Each piece surviving in  $I'$  can be mapped to a piece in  $I$  of no lesser size. Hence,  $\text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$
- ▶  $\lfloor x_j \rfloor$  is feasible solution for  $I_1$  (even integral).
- ▶  $x_j - \lfloor x_j \rfloor$  is feasible solution for  $I_2$ .

$$\text{OPT}_{\text{LP}}(I_1) + \text{OPT}_{\text{LP}}(I_2) \leq \text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$$

## Proof:

- ▶ Each piece surviving in  $I'$  can be mapped to a piece in  $I$  of no lesser size. Hence,  $\text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$
- ▶  $\lfloor x_j \rfloor$  is feasible solution for  $I_1$  (even integral).
- ▶  $x_j - \lfloor x_j \rfloor$  is feasible solution for  $I_2$ .

$$\text{OPT}_{\text{LP}}(I_1) + \text{OPT}_{\text{LP}}(I_2) \leq \text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$$

## Proof:

- ▶ Each piece surviving in  $I'$  can be mapped to a piece in  $I$  of no lesser size. Hence,  $\text{OPT}_{\text{LP}}(I') \leq \text{OPT}_{\text{LP}}(I)$
- ▶  $\lfloor x_j \rfloor$  is feasible solution for  $I_1$  (even integral).
- ▶  $x_j - \lfloor x_j \rfloor$  is feasible solution for  $I_2$ .

# Analysis

Each level of the recursion partitions pieces into three types

1. Pieces discarded at this level.
2. Pieces scheduled because they are in  $I_1$ .
3. Pieces in  $I_2$  are handed down to the next level.

Pieces of type 2 summed over all recursion levels are packed into at most  $\text{OPT}_{1D}$  many bins.

Pieces of type 1 are packed into at most

$$\mathcal{O}(\log(\text{SIZE}(I))) \cdot L$$

many bins where  $L$  is the number of recursion levels.

# Analysis

Each level of the recursion partitions pieces into three types

1. Pieces discarded at this level.
2. Pieces scheduled because they are in  $I_1$ .
3. Pieces in  $I_2$  are handed down to the next level.

Pieces of type 2 summed over all recursion levels are packed into at most  $\text{OPT}_{1D}$  many bins.

Pieces of type 1 are packed into at most

$$\mathcal{O}(\log(\text{SIZE}(I))) \cdot L$$

many bins where  $L$  is the number of recursion levels.



# Analysis

Each level of the recursion partitions pieces into three types

1. Pieces discarded at this level.
2. Pieces scheduled because they are in  $I_1$ .
3. Pieces in  $I_2$  are handed down to the next level.

Pieces of type 2 summed over all recursion levels are packed into at most  $\text{OPT}_{\text{LP}}$  many bins.

Pieces of type 1 are packed into at most

$$\mathcal{O}(\log(\text{SIZE}(I))) \cdot L$$

many bins where  $L$  is the number of recursion levels.

# Analysis

Each level of the recursion partitions pieces into three types

1. Pieces discarded at this level.
2. Pieces scheduled because they are in  $I_1$ .
3. Pieces in  $I_2$  are handed down to the next level.

Pieces of type 2 summed over all recursion levels are packed into at most  $\text{OPT}_{\text{LP}}$  many bins.

Pieces of type 1 are packed into at most

$$\mathcal{O}(\log(\text{SIZE}(I))) \cdot L$$

many bins where  $L$  is the number of recursion levels.

# Analysis

Each level of the recursion partitions pieces into three types

1. Pieces discarded at this level.
2. Pieces scheduled because they are in  $I_1$ .
3. Pieces in  $I_2$  are handed down to the next level.

Pieces of type 2 summed over all recursion levels are packed into at most  $\text{OPT}_{\text{LP}}$  many bins.

Pieces of type 1 are packed into at most

$$\mathcal{O}(\log(\text{SIZE}(I))) \cdot L$$

many bins where  $L$  is the number of recursion levels.

# Analysis

We can show that  $\text{SIZE}(I_2) \leq \text{SIZE}(I)/2$ . Hence, the number of recursion levels is only  $\mathcal{O}(\log(\text{SIZE}(I_{\text{original}})))$  in total.

# Analysis

We can show that  $\text{SIZE}(I_2) \leq \text{SIZE}(I)/2$ . Hence, the number of recursion levels is only  $\mathcal{O}(\log(\text{SIZE}(I_{\text{original}})))$  in total.

- ▶ The number of non-zero entries in the solution to the configuration LP for  $I'$  is at most the number of constraints, which is the number of different sizes ( $\leq \text{SIZE}(I)/2$ ).
- ▶ The total size of items in  $I_2$  can be at most  $\sum_{j=1}^N x_j - \lfloor x_j \rfloor$  which is at most the number of non-zero entries in the solution to the configuration LP.

# Analysis

We can show that  $\text{SIZE}(I_2) \leq \text{SIZE}(I)/2$ . Hence, the number of recursion levels is only  $\mathcal{O}(\log(\text{SIZE}(I_{\text{original}})))$  in total.

- ▶ The number of non-zero entries in the solution to the configuration LP for  $I'$  is at most the number of constraints, which is the number of different sizes ( $\leq \text{SIZE}(I)/2$ ).
- ▶ The total size of items in  $I_2$  can be at most  $\sum_{j=1}^N x_j - \lfloor x_j \rfloor$  which is at most the number of non-zero entries in the solution to the configuration LP.

# How to solve the LP?

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

In total we have  $b_i$  pieces of size  $s_i$ .

## Primal

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1, \dots, m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$

## Dual

$$\begin{array}{ll} \max & \sum_{i=1}^m \gamma_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} \gamma_i \leq 1 \\ & \forall i \in \{1, \dots, m\} \quad \gamma_i \geq 0 \end{array}$$

# How to solve the LP?

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

In total we have  $b_i$  pieces of size  $s_i$ .

## Primal

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$

## Dual

$$\begin{array}{ll} \max & \sum_{i=1}^m y_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} y_i \leq 1 \\ & \forall i \in \{1, \dots, m\} \quad y_i \geq 0 \end{array}$$



# How to solve the LP?

Let  $T_1, \dots, T_N$  be the sequence of all possible configurations (a configuration  $T_j$  has  $T_{ji}$  pieces of size  $s_i$ ).

In total we have  $b_i$  pieces of size  $s_i$ .

## Primal

$$\begin{array}{ll} \min & \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$

## Dual

$$\begin{array}{ll} \max & \sum_{i=1}^m y_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} y_i \leq 1 \\ & \forall i \in \{1, \dots, m\} \quad y_i \geq 0 \end{array}$$

## Separation Oracle

Suppose that I am given variable assignment  $y$  for the dual.

**How do I find a violated constraint?**

I have to find a configuration  $T_j = (T_{j1}, \dots, T_{jm})$  that

is feasible for  $P_j$ .

and has a large profit.

But this is the Knapsack problem.

## Separation Oracle

Suppose that I am given variable assignment  $\gamma$  for the dual.

**How do I find a violated constraint?**

I have to find a configuration  $T_j = (T_{j1}, \dots, T_{jm})$  that

- ▶ is feasible, i.e.,

$$\sum_{i=1}^m T_{ji} \cdot \gamma_i \leq 1 ,$$

- ▶ and has a large profit

$$\sum_{i=1}^m T_{ji} \gamma_i > 1$$

But this is the Knapsack problem.

## Separation Oracle

Suppose that I am given variable assignment  $\gamma$  for the dual.

**How do I find a violated constraint?**

I have to find a configuration  $T_j = (T_{j1}, \dots, T_{jm})$  that

- ▶ is feasible, i.e.,

$$\sum_{i=1}^m T_{ji} \cdot \gamma_i \leq 1 ,$$

- ▶ and has a large profit

$$\sum_{i=1}^m T_{ji} \gamma_i > 1$$

But this is the Knapsack problem.

## Separation Oracle

Suppose that I am given variable assignment  $\gamma$  for the dual.

**How do I find a violated constraint?**

I have to find a configuration  $T_j = (T_{j1}, \dots, T_{jm})$  that

- ▶ is feasible, i.e.,

$$\sum_{i=1}^m T_{ji} \cdot \gamma_i \leq 1 ,$$

- ▶ and has a large profit

$$\sum_{i=1}^m T_{ji} \gamma_i > 1$$

But this is the Knapsack problem.

## Separation Oracle

We have FPTAS for Knapsack. This means if a constraint is violated with  $1 + \epsilon' = 1 + \frac{\epsilon}{1-\epsilon}$  we find it, since we can obtain at least  $(1 - \epsilon)$  of the optimal profit.

The solution we get is feasible for:

Dual'

$$\begin{array}{ll} \max & \sum_{i=1}^m y_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} y_i \leq 1 + \epsilon' \\ & \forall i \in \{1, \dots, m\} \quad y_i \geq 0 \end{array}$$

Primal'

$$\begin{array}{ll} \min & (1 + \epsilon') \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1, \dots, m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$

## Separation Oracle

We have FPTAS for Knapsack. This means if a constraint is violated with  $1 + \epsilon' = 1 + \frac{\epsilon}{1-\epsilon}$  we find it, since we can obtain at least  $(1 - \epsilon)$  of the optimal profit.

The solution we get is feasible for:

Dual'

$$\begin{array}{ll} \max & \sum_{i=1}^m y_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} y_i \leq 1 + \epsilon' \\ & \forall i \in \{1, \dots, m\} \quad y_i \geq 0 \end{array}$$

Primal'

$$\begin{array}{ll} \min & (1 + \epsilon') \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1, \dots, m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$

## Separation Oracle

We have FPTAS for Knapsack. This means if a constraint is violated with  $1 + \epsilon' = 1 + \frac{\epsilon}{1-\epsilon}$  we find it, since we can obtain at least  $(1 - \epsilon)$  of the optimal profit.

The solution we get is feasible for:

### Dual'

$$\begin{array}{ll} \max & \sum_{i=1}^m y_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} y_i \leq 1 + \epsilon' \\ & \forall i \in \{1, \dots, m\} \quad y_i \geq 0 \end{array}$$

### Primal'

$$\begin{array}{ll} \min & (1 + \epsilon') \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1, \dots, m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$



## Separation Oracle

We have FPTAS for Knapsack. This means if a constraint is violated with  $1 + \epsilon' = 1 + \frac{\epsilon}{1-\epsilon}$  we find it, since we can obtain at least  $(1 - \epsilon)$  of the optimal profit.

The solution we get is feasible for:

### Dual'

$$\begin{array}{ll} \max & \sum_{i=1}^m y_i b_i \\ \text{s.t.} & \forall j \in \{1, \dots, N\} \quad \sum_{i=1}^m T_{ji} y_i \leq 1 + \epsilon' \\ & \forall i \in \{1, \dots, m\} \quad y_i \geq 0 \end{array}$$

### Primal'

$$\begin{array}{ll} \min & (1 + \epsilon') \sum_{j=1}^N x_j \\ \text{s.t.} & \forall i \in \{1 \dots m\} \quad \sum_{j=1}^N T_{ji} x_j \geq b_i \\ & \forall j \in \{1, \dots, N\} \quad x_j \geq 0 \end{array}$$

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

How do we get good primal solution (not just the value)?

The constraints used when computing  $z$  imply that the solution is feasible for

the LP where we drop all unused constraints. We can then compute the corresponding feasible primal solution by setting the primal variables for unused constraints to zero. The primal value is  $z - \epsilon' \text{OPT}$ .

The dual is feasible, so we can increase variables for which the corresponding dual constraint has not been used.

The primal value is  $z - \epsilon' \text{OPT}$ .

The algorithm for computing the primal solution is:

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

**How do we get good primal solution (not just the value)?**

- ▶ The constraints used when computing  $z$  **certify** that the solution is feasible for **DUAL'**.
- ▶ Suppose that we drop all unused constraints in **DUAL**. We will compute the same solution feasible for **DUAL'**.
- ▶ Let **DUAL''** be **DUAL** without unused constraints.
- ▶ The dual to **DUAL''** is **PRIMAL** where we ignore variables for which the corresponding dual constraint has not been used.
- ▶ The optimum value for **PRIMAL''** is at most  $(1 + \epsilon')\text{OPT}$ .
- ▶ We can compute the corresponding solution in polytime.

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

**How do we get good primal solution (not just the value)?**

- ▶ The constraints used when computing  $z$  certify that the solution is feasible for  $\text{DUAL}'$ .
- ▶ Suppose that we drop all unused constraints in  $\text{DUAL}$ . We will compute the same solution feasible for  $\text{DUAL}'$ .
  - ▶ Let  $\text{DUAL}''$  be  $\text{DUAL}$  without unused constraints.
  - ▶ The dual to  $\text{DUAL}''$  is  $\text{PRIMAL}$  where we ignore variables for which the corresponding dual constraint has not been used.
  - ▶ The optimum value for  $\text{PRIMAL}''$  is at most  $(1 + \epsilon')\text{OPT}$ .
  - ▶ We can compute the corresponding solution in polytime.

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

**How do we get good primal solution (not just the value)?**

- ▶ The constraints used when computing  $z$  **certify** that the solution is feasible for **DUAL'**.
- ▶ Suppose that we drop all unused constraints in **DUAL**. We will compute the same solution feasible for **DUAL'**.
- ▶ Let **DUAL''** be **DUAL** without unused constraints.
  - ▶ The dual to **DUAL''** is **PRIMAL** where we ignore variables for which the corresponding dual constraint has not been used.
  - ▶ The optimum value for **PRIMAL''** is at most  $(1 + \epsilon')\text{OPT}$ .
  - ▶ We can compute the corresponding solution in polytime.

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

**How do we get good primal solution (not just the value)?**

- ▶ The constraints used when computing  $z$  certify that the solution is feasible for  $\text{DUAL}'$ .
- ▶ Suppose that we drop all unused constraints in  $\text{DUAL}$ . We will compute the same solution feasible for  $\text{DUAL}'$ .
- ▶ Let  $\text{DUAL}''$  be  $\text{DUAL}$  without unused constraints.
- ▶ The dual to  $\text{DUAL}''$  is  $\text{PRIMAL}$  where we ignore variables for which the corresponding dual constraint has not been used.
- ▶ The optimum value for  $\text{PRIMAL}''$  is at most  $(1 + \epsilon')\text{OPT}$ .
- ▶ We can compute the corresponding solution in polytime.

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

**How do we get good primal solution (not just the value)?**

- ▶ The constraints used when computing  $z$  **certify** that the solution is feasible for **DUAL'**.
- ▶ Suppose that we drop all unused constraints in **DUAL**. We will compute the same solution feasible for **DUAL'**.
- ▶ Let **DUAL''** be **DUAL** without unused constraints.
- ▶ The dual to **DUAL''** is **PRIMAL** where we ignore variables for which the corresponding dual constraint has not been used.
- ▶ The optimum value for **PRIMAL''** is at most  $(1 + \epsilon')\text{OPT}$ .
- ▶ We can compute the corresponding solution in polytime.

## Separation Oracle

If the value of the computed dual solution (which may be infeasible) is  $z$  then

$$\text{OPT} \leq z \leq (1 + \epsilon')\text{OPT}$$

**How do we get good primal solution (not just the value)?**

- ▶ The constraints used when computing  $z$  **certify** that the solution is feasible for **DUAL'**.
- ▶ Suppose that we drop all unused constraints in **DUAL**. We will compute the same solution feasible for **DUAL'**.
- ▶ Let **DUAL''** be **DUAL** without unused constraints.
- ▶ The dual to **DUAL''** is **PRIMAL** where we ignore variables for which the corresponding dual constraint has not been used.
- ▶ The optimum value for **PRIMAL''** is at most  $(1 + \epsilon')\text{OPT}$ .
- ▶ We can compute the corresponding solution in polytime.



This gives that overall we need at most

$$(1 + \epsilon') \text{OPT}_{\text{LP}}(I) + \mathcal{O}(\log^2(\text{SIZE}(I)))$$

bins.

We can choose  $\epsilon' = \frac{1}{\text{OPT}}$  as  $\text{OPT} \leq \#\text{items}$  and since we have a fully polynomial time approximation scheme (FPTAS) for knapsack.

This gives that overall we need at most

$$(1 + \epsilon') \text{OPT}_{\text{LP}}(I) + \mathcal{O}(\log^2(\text{SIZE}(I)))$$

bins.

We can choose  $\epsilon' = \frac{1}{\text{OPT}}$  as  $\text{OPT} \leq \# \text{items}$  and since we have a **fully polynomial time approximation scheme (FPTAS)** for knapsack.