Advanced Algorithms (Fall 2023)

Rounding Data and Dynamic Programming

Lecturers: 尹一通，刘景铖，栗师

Nanjing University

Outline

Outline

Knapsack Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a subset S of items that

$$\text{maximizes} \sum_{i \in S} v_i \qquad \text{s.t.} \sum_{i \in S} w_i \leq W.$$

Knapsack Problem

Input: an integer bound $W > 0$

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a subset S of items that

$$\text{maximizes } \sum_{i \in S} v_i \qquad \text{s.t.} \sum_{i \in S} w_i \leq W.$$

- Motivation: you have budget W, and want to buy a subset of items of maximum total value

Greedy Algorithm

sort items according to non-increasing order of v_i/w_i
for each item in the ordering **do**
    take the item if we have enough budget

Greedy Algorithm

```
sort items according to non-increasing order of v_i/w_i
for each item in the ordering do
    take the item if we have enough budget
```

- Bad example: $W = 100, n = 2, w = (1, 100), v = (1.1, 100)$.

Greedy Algorithm

1: sort items according to non-increasing order of v_i/w_i
2: **for** each item in the ordering **do**
3: take the item if we have enough budget

- Bad example: $W = 100, n = 2, w = (1, 100), v = (1.1, 100)$.
- Optimum takes item 2 and greedy takes item 1.

Fractional Knapsack Problem

Input: integer bound $W > 0$,

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a vector $(\alpha_1, \alpha_2, \cdots, \alpha_n) \in [0,1]^n$ that

$$\text{maximizes } \sum_{i=1}^{n} \alpha_i v_i \qquad \text{s.t.} \sum_{i=1}^{n} \alpha_i w_i \le W.$$

Fractional Knapsack Problem

Input: integer bound $W > 0$,

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a vector $(\alpha_1, \alpha_2, \cdots, \alpha_n) \in [0,1]^n$ that

$$\text{maximizes} \sum_{i=1}^{n} \alpha_i v_i \qquad \text{s.t.} \sum_{i=1}^{n} \alpha_i w_i \le W.$$

Greedy Algorithm for Fractional Knapsack

1: sort items according to non-increasing order of v_i/w_i,

2: for each item according to the ordering, take as much fraction of the item as possible.

Fractional Knapsack Problem

Input: integer bound $W > 0$,

a set of n items, each with an integer weight $w_i > 0$

a value $v_i > 0$ for each item i

Output: a vector $(\alpha_1, \alpha_2, \cdots, \alpha_n) \in [0, 1]^n$ that

$$\text{maximizes } \sum_{i=1}^{n} \alpha_i v_i \qquad \text{s.t.} \sum_{i=1}^{n} \alpha_i w_i \le W.$$

Greedy Algorithm for Fractional Knapsack

1: sort items according to non-increasing order of v_i/w_i,

2: for each item according to the ordering, take as much fraction of the item as possible.

Theorem Greedy algorithm gives the optimum solution for fractional knapsack.

DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \leq W' \end{cases}$$

DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \le W' \end{cases}$$

- Running time of the algorithm is $O(nW)$.

DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{matrix} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{matrix} \right\} & i > 0, w_i \le W' \end{cases}$$

- Running time of the algorithm is $O(nW)$.

Q: Is this a polynomial time?

DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \le W' \end{cases}$$

- Running time of the algorithm is $O(nW)$.

Q: Is this a polynomial time?

A: No.

DP for Knapsack Problem

- $opt[i, W']$: the optimum value when budget is W' and items are $\{1, 2, 3, \cdots, i\}$.

$$opt[i, W'] = \begin{cases} 0 & i = 0 \\ opt[i-1, W'] & i > 0, w_i > W' \\ \max \left\{ \begin{array}{c} opt[i-1, W'] \\ opt[i-1, W' - w_i] + v_i \end{array} \right\} & i > 0, w_i \le W' \end{cases}$$

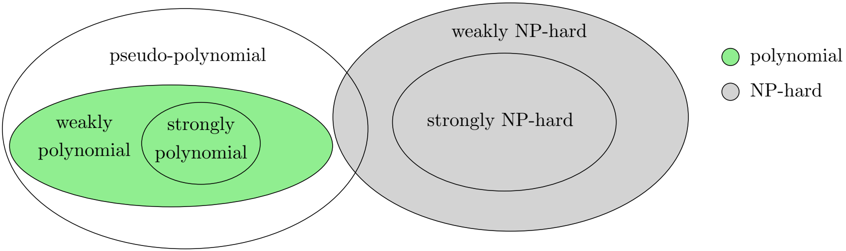
- Running time of the algorithm is $O(nW)$.

Q: Is this a polynomial time?

A: No.

- The input size is polynomial in n and $\log W$; running time is polynomial in n and W.
- The running time is pseudo-polynomial.

- n: number of integers $\quad W$: maximum value of all integers
- pseudo-polynomial time: $\mathrm{poly}(n, W)$ (e.g., DP for Knapsack)
- weakly polynomial time: $\mathrm{poly}(n, \log W)$ (e.g., Euclidean Algorithm for Greatest Common Divisor)
- strongly polynomial time: $\mathrm{poly}(n)$ time, assuming basic operations on integers taking $O(1)$ time (e.g., Kruskal's)
- weakly NP-hard: NP-hard to solve in time $\mathrm{poly}(n, \log W)$
- strongly NP-hard: NP-hard even if $W = \mathrm{poly}(n)$



Outline

Idea for improving the running time to polynomial

- If we make weights upper bounded by $\mathrm{poly}(n)$, then pseudo-polynomial time becomes polynomial time
- Coarsening the weights: $w_i' = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer A.

Idea for improving the running time to polynomial

- If we make weights upper bounded by $\mathrm{poly}(n)$, then pseudo-polynomial time becomes polynomial time
- Coarsening the weights: $w_i' = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer A.
- However, coarsening weights will change the problem.

Idea for improving the running time to polynomial

- If we make weights upper bounded by $\mathrm{poly}(n)$, then pseudo-polynomial time becomes polynomial time
- Coarsening the weights: $w_i' = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer A.
- However, coarsening weights will change the problem.
- $\dfrac{\text{weight budget constraint} \quad : \quad \text{hard}}{\text{maximum value requirement} \quad : \quad \text{soft}}$

Idea for improving the running time to polynomial

- If we make weights upper bounded by $\mathrm{poly}(n)$, then pseudo-polynomial time becomes polynomial time
- Coarsening the weights: $w'_i = \left\lfloor \frac{w_i}{A} \right\rfloor$ for some appropriately defined integer A.
- However, coarsening weights will change the problem.
- $\frac{\text{weight budget constraint} \quad : \quad \text{hard}}{\text{maximum value requirement} \quad : \quad \text{soft}}$
- We coarsen the values instead
- In the DP, we use values as parameters

- Let A be some integer to be defined later

- Let A be some integer to be defined later
- $v_i' := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item i

- Let A be some integer to be defined later
- $v_i' := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item i
- Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i] : v'(S) \geq V'} w(S)$

- Let A be some integer to be defined later
- $v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item i
- Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i]: v'(S) \geq V'} w(S)$

$$f[i, V'] = \begin{cases} 0 & V' \leq 0 \\ \infty & i = 0, V' > 0 \\ \min \left\{ \begin{array}{c} f[i-1, V'] \\ f[i-1, V' - v'_i] + w_i \end{array} \right\} & i > 0, V' > 0 \end{cases}$$

- Let A be some integer to be defined later
- $v'_i := \left\lfloor \frac{v_i}{A} \right\rfloor$ be the scaled value of item i
- Definition of DP cells: $f[i, V'] = \min_{S \subseteq [i] : v'(S) \geq V'} w(S)$

$$f[i, V'] = \begin{cases} 0 & V' \leq 0 \\ \infty & i = 0, V' > 0 \\ \min \left\{ \begin{array}{c} f[i-1, V'] \\ f[i-1, V' - v'_i] + w_i \end{array} \right\} & i > 0, V' > 0 \end{cases}$$

- Output A times the largest V' such that $f[n, V'] \leq W$.

- Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ $\quad$ opt: optimum value of $\mathcal{I}$
- Instance $\mathcal{I}'$: $(Av'_1, \cdots, AV'_n)$ $\quad$ opt': optimum value of $\mathcal{I}'$

- Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ $\qquad$ opt: optimum value of $\mathcal{I}$
- Instance $\mathcal{I}'$: $(Av'_1, \cdots, AV'_n)$ $\qquad$ opt': optimum value of $\mathcal{I}'$

$$\begin{aligned} & v_i - A < Av'_i \le v_i, & \forall i \in [n] \\ \Longrightarrow \quad & \mathrm{opt} - nA < \mathrm{opt}' \le \mathrm{opt} & \end{aligned}$$

- $\mathrm{opt} \ge v_{\max} := \max_{i\in[n]} v_i$ (assuming $w_i \le W, \forall i$)

- Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ $\quad$ opt: optimum value of $\mathcal{I}$
- Instance $\mathcal{I}'$: $(Av_1', \cdots, AV_n')$ $\quad$ opt': optimum value of $\mathcal{I}'$

$$
\begin{aligned}
& v_i - A < Av_i' \leq v_i, \qquad \forall i \in [n] \\
\implies \quad & \mathrm{opt} - nA < \mathrm{opt}' \leq \mathrm{opt}
\end{aligned}
$$

- $\mathrm{opt} \geq v_{\max} := \max_{i \in [n]} v_i$ (assuming $w_i \leq W, \forall i$)
- setting $A := \left\lfloor \frac{\epsilon \cdot v_{\max}}{n} \right\rfloor$: $(1-\epsilon)\mathrm{opt} \leq \mathrm{opt}' \leq \mathrm{opt}$

- Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ opt: optimum value of $\mathcal{I}$
- Instance $\mathcal{I}'$: $(Av_1', \cdots, AV_n')$ opt': optimum value of $\mathcal{I}'$

$$
\begin{aligned}
& v_i - A < Av_i' \leq v_i, \qquad \forall i \in [n] \\
\implies \quad & \mathrm{opt} - nA < \mathrm{opt}' \leq \mathrm{opt}
\end{aligned}
$$

- $\mathrm{opt} \geq v_{\max} := \max_{i\in[n]} v_i$ (assuming $w_i \leq W, \forall i$)
- setting $A := \left\lfloor \frac{\epsilon \cdot v_{\max}}{n} \right\rfloor$: $(1-\epsilon)\mathrm{opt} \leq \mathrm{opt}' \leq \mathrm{opt}$

- $\forall i, v_i' = O(\frac{n}{\epsilon})$ $\implies$ running time $= O(\frac{n^3}{\epsilon})$

- Instance $\mathcal{I}$: $(v_1, v_2, \cdots, v_n)$ $\quad$ opt: optimum value of $\mathcal{I}$
- Instance $\mathcal{I}'$: $(Av_1', \cdots, AV_n')$ $\quad$ opt': optimum value of $\mathcal{I}'$

$$\begin{aligned} & v_i - A < Av_i' \leq v_i, && \forall i \in [n] \\ \implies \quad & \mathrm{opt} - nA < \mathrm{opt}' \leq \mathrm{opt} \end{aligned}$$

- $\mathrm{opt} \geq v_{\max} := \max_{i \in [n]} v_i$ (assuming $w_i \leq W, \forall i$)
- setting $A := \left\lfloor \frac{\epsilon \cdot v_{\max}}{n} \right\rfloor$: $(1-\epsilon)\mathrm{opt} \leq \mathrm{opt}' \leq \mathrm{opt}$

- $\forall i, v_i' = O(\frac{n}{\epsilon})$ $\quad \implies \quad$ running time $= O(\frac{n^3}{\epsilon})$

Theorem There is a $(1+\epsilon)$-approximation for the knapsack problem in time $O(\frac{n^3}{\epsilon})$.

Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms A_ϵ, where A_ϵ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem

Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms A_ϵ, where A_ϵ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme A_ϵ such that the running time of A_ϵ is $\mathrm{poly}(n, \frac{1}{\epsilon})$ for input instances of n.

Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms A_ϵ, where A_ϵ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme A_ϵ such that the running time of A_ϵ is $\mathrm{poly}(n, \frac{1}{\epsilon})$ for input instances of n.

- So, Knapsack admits an FPTAS.

Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms A_ϵ, where A_ϵ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1 + \epsilon$ or $1 - \epsilon$, depending on whether the problem is a minimization/maximization problem

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme A_ϵ such that the running time of A_ϵ is $\mathrm{poly}(n, \frac{1}{\epsilon})$ for input instances of n.

- So, Knapsack admits an FPTAS.

Q: Assume P $\neq$ NP. What is a neccesary condition for a NP-hard problem to admit an FPTAS?

Def. A polynomial-time approximation scheme (PTAS) is a family of algorithms A_ϵ, where A_ϵ for every $\epsilon > 0$ is a (polynomial-time) $(1 \pm \epsilon)$-approximation algorithm.

- Remark: the approximation ratio is $1+\epsilon$ or $1-\epsilon$, depending on whether the problem is a minimization/maximization problem

Def. A fully polynomial-time approximation scheme (FPTAS) is an approximation scheme A_ϵ such that the running time of A_ϵ is $\mathrm{poly}(n, \frac{1}{\epsilon})$ for input instances of n.

- So, Knapsack admits an FPTAS.

Q: Assume P $\neq$ NP. What is a neccesary condition for a NP-hard problem to admit an FPTAS?

- Vertex cover? Maximum independent set?

Outline

Outline

Makespan Minimization on Identical Machines

Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Makespan Minimization on Identical Machines

Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Output: schedule of jobs on machines with minimum makespan

Makespan Minimization on Identical Machines

Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Output: schedule of jobs on machines with minimum makespan

$\sigma : [n] \to [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$

Makespan Minimization on Identical Machines

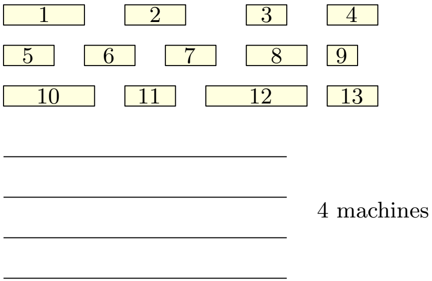
Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Output: schedule of jobs on machines with minimum makespan

$\sigma : [n] \to [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$



Makespan Minimization on Identical Machines

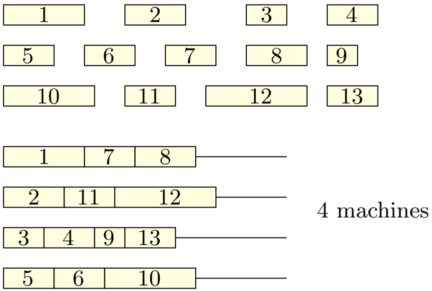
Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Output: schedule of jobs on machines with minimum makespan

$\sigma : [n] \to [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$



Makespan Minimization on Identical Machines

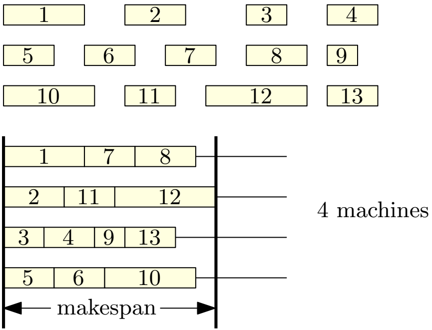
Input: n jobs index as $[n]$

each job $j \in [n]$ has a processing time $p_j \in \mathbb{Z}_{>0}$

m machines

Output: schedule of jobs on machines with minimum makespan

$\sigma : [n] \to [m]$ with minimum $\max_{i \in [m]} \sum_{j \in \sigma^{-1}(i)} p_j$



Greedy Algorithm

```
start from an empty schedule
for j = 1 to n do
    put job j on the machine with the smallest load
```

Greedy Algorithm

1: start from an empty schedule
2: **for** $j = 1$ to n **do**
3: put job j on the machine with the smallest load

Analysis of $(2 - \frac{1}{m})$-Approximation for Greedy Algorithm

Greedy Algorithm

1: start from an empty schedule
2: **for** $j = 1$ to n **do**
3: put job j on the machine with the smallest load

Analysis of $(2 - \frac{1}{m})$-Approximation for Greedy Algorithm

$$p_{\max} := \max_{j \in [n]} p_j$$

$$\text{alg} \leq p_{\max} + \frac{1}{m} \cdot \Big(\sum_{j \in [n]} p_j - p_{\max}\Big) = \Big(1 - \frac{1}{m}\Big) p_{\max} + \frac{1}{m} \sum_{j \in [n]} p_j$$

Greedy Algorithm

1: start from an empty schedule
2: **for** $j = 1$ to n **do**
3: put job j on the machine with the smallest load

Analysis of $(2 - \frac{1}{m})$-Approximation for Greedy Algorithm

$$p_{\max} := \max_{j \in [n]} p_j$$

$$\mathrm{alg} \leq p_{\max} + \frac{1}{m} \cdot \Big(\sum_{j \in [n]} p_j - p_{\max}\Big) = \Big(1 - \frac{1}{m}\Big) p_{\max} + \frac{1}{m} \sum_{j \in [n]} p_j$$

$$\left.\begin{array}{lcl} \mathrm{opt} & \geq & p_{\max} \\ \mathrm{opt} & \geq & \frac{1}{m}\sum_{j \in [n]} p_j \end{array}\right\} \Longrightarrow \quad \mathrm{alg} \leq \Big(2 - \frac{1}{m}\Big)\mathrm{opt}$$

Q: What happens if all items have size at most $\epsilon \cdot \mathrm{opt}$?

Q: What happens if all items have size at most $\epsilon \cdot \mathrm{opt}$?

A: $\mathrm{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\max} \leq \mathrm{opt} + \epsilon \cdot \mathrm{opt} = (1 + \epsilon)\mathrm{opt}.$

Q: What happens if all items have size at most $\epsilon \cdot \mathrm{opt}$?

A: $\mathrm{alg} \le \frac{1}{m} \sum_{j \in [n]} p_j + p_{\max} \le \mathrm{opt} + \epsilon \cdot \mathrm{opt} = (1+\epsilon)\mathrm{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \mathrm{opt}$?

Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\max} \leq \text{opt} + \epsilon \cdot \text{opt} = (1+\epsilon)\text{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that #(distinct sizes) is small

Q: What happens if all items have size at most $\epsilon \cdot \mathrm{opt}$?

A: $\mathrm{alg} \leq \frac{1}{m}\sum_{j\in[n]} p_j + p_{\max} \leq \mathrm{opt} + \epsilon \cdot \mathrm{opt} = (1+\epsilon)\mathrm{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \mathrm{opt}$?

A: We can round the sizes, so that #(distinct sizes) is small

Overview of Algorithm

1: declare j small if $p_j < \epsilon \cdot p_{\max}$ and big otherwise

Q: What happens if all items have size at most $\epsilon \cdot \mathrm{opt}$?

A: $\mathrm{alg} \leq \frac{1}{m}\sum_{j\in[n]} p_j + p_{\max} \leq \mathrm{opt} + \epsilon \cdot \mathrm{opt} = (1+\epsilon)\mathrm{opt}$.

Q: What can we do if all items have size at least $\epsilon \cdot \mathrm{opt}$?

A: We can round the sizes, so that #(distinct sizes) is small

Overview of Algorithm

1: declare j small if $p_j < \epsilon \cdot p_{\max}$ and big otherwise
2: use trunction + DP to solve the instance defined by big jobs

Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\max} \leq \text{opt} + \epsilon \cdot \text{opt} = (1+\epsilon)\text{opt}.$

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that #(distinct sizes) is small

Overview of Algorithm

1: declare j small if $p_j < \epsilon \cdot p_{\max}$ and big otherwise
2: use trunction + DP to solve the instance defined by big jobs
3: use DP for instance $(p'_j)_{j \text{ big}}$ to schedule big jobs

Q: What happens if all items have size at most $\epsilon \cdot \text{opt}$?

A: $\text{alg} \leq \frac{1}{m} \sum_{j \in [n]} p_j + p_{\max} \leq \text{opt} + \epsilon \cdot \text{opt} = (1+\epsilon)\text{opt}.$

Q: What can we do if all items have size at least $\epsilon \cdot \text{opt}$?

A: We can round the sizes, so that #(distinct sizes) is small

Overview of Algorithm

1: declare j small if $p_j < \epsilon \cdot p_{\max}$ and big otherwise
2: use trunction + DP to solve the instance defined by big jobs
3: use DP for instance $(p'_j)_{j \text{ big}}$ to schedule big jobs
4: add small jobs to schedule greedily

Outline

Dynamic Programming for Big Jobs

- $B := \{j \in [n] : p_j \geq \epsilon p_{\max}\}$: set of big jobs

Dynamic Programming for Big Jobs

- $B := \{j \in [n] : p_j \geq \epsilon p_{\max}\}$: set of big jobs
- $p'_j := \max\{p_{\max}(1+\epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$

  p'_j is the rounded size of j

Dynamic Programming for Big Jobs

- $B := \{j \in [n] : p_j \geq \epsilon p_{\max}\}$: set of big jobs
- $p'_j := \max\{p_{\max}(1+\epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$

  p'_j is the rounded size of j
- $k := |\{p'_j : j \in B\}|$: #(distinct rounded sizes)

  $k \leq 1 + \log_{1+\epsilon} \frac{p_{\max}}{\epsilon p_{\max}} = O\left(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon}\right)$

Dynamic Programming for Big Jobs

- $B := \{j \in [n] : p_j \geq \epsilon p_{\max}\}$: set of big jobs
- $p'_j := \max\{p_{\max}(1+\epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$

  p'_j is the rounded size of j
- $k := |\{p'_j : j \in B\}|$: #(distinct rounded sizes)

  $k \leq 1 + \log_{1+\epsilon} \frac{p_{\max}}{\epsilon p_{\max}} = O\left(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon}\right)$
- $\{q_1, q_2, \cdots, q_k\} := \{p'_j : j \in B\}$: the k distinct rounded sizes

Dynamic Programming for Big Jobs

- $B := \{j \in [n] : p_j \geq \epsilon p_{\max}\}$: set of big jobs
- $p'_j := \max\{p_{\max}(1+\epsilon)^t \leq p_j : t \in \mathbb{Z}\}, \forall j \in B$

  p'_j is the rounded size of j
- $k := |\{p'_j : j \in B\}|$: #(distinct rounded sizes)

  $k \leq 1 + \log_{1+\epsilon} \frac{p_{\max}}{\epsilon p_{\max}} = O\left(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon}\right)$
- $\{q_1, q_2, \cdots, q_k\} := \{p'_j : j \in B\}$: the k distinct rounded sizes
- $n_1, \cdots, n_k$: #(big jobs) with rounded sizes being $q_1, \cdots, q_k$

Constructing a Directed Acyclic Graph $G = (V, E)$

Constructing a Directed Acyclic Graph $G = (V, E)$

- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i], \forall i \in [k]$
  - denotes the instance with a_1 jobs of size q_1, a_2 jobs of size q_2, $\cdots$, a_k jobs of size q_k

Constructing a Directed Acyclic Graph $G = (V, E)$

- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i], \forall i \in [k]$
  - denotes the instance with a_1 jobs of size q_1, a_2 jobs of size q_2, $\cdots$, a_k jobs of size q_k
- an arc $(a_1, \cdots, a_k) \to (b_1, \cdots b_k)$ of weight $\sum_{i=1}^{k}(b_i - a_i)q_i$, if $a_i \leq b_i, \forall i \in [k]$, and $a_i < b_i$ for some $i \in [k]$
  - reducing instance $(b_1, \cdots b_k)$ to $(a_1, \cdots, a_k)$ requires 1 machine of load $\sum_{i=1}^{k}(b_i - a_i)q_i$

Constructing a Directed Acyclic Graph $G = (V, E)$

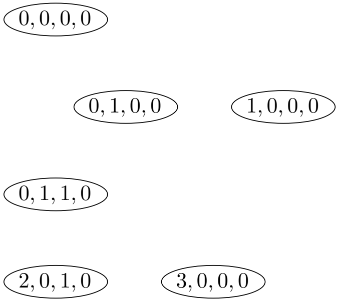
- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i], \forall i \in [k]$
  - denotes the instance with a_1 jobs of size q_1, a_2 jobs of size q_2, $\cdots$, a_k jobs of size q_k
- an arc $(a_1, \cdots, a_k) \to (b_1, \cdots b_k)$ of weight $\sum_{i=1}^{k}(b_i - a_i)q_i$, if $a_i \leq b_i, \forall i \in [k]$, and $a_i < b_i$ for some $i \in [k]$
  - reducing instance $(b_1, \cdots b_k)$ to $(a_1, \cdots, a_k)$ requires 1 machine of load $\sum_{i=1}^{k}(b_i - a_i)q_i$

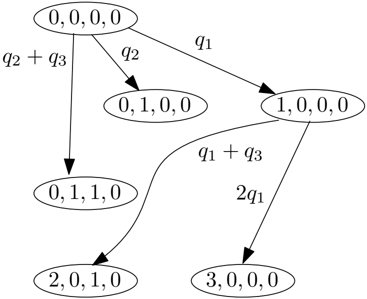
- Goal: find a path from $(0, \cdots, 0)$ to $(n_1, \cdots, n_k)$ of at most m edges, so as to minimize the maximum weight on the path.

Constructing a Directed Acyclic Graph $G = (V, E)$

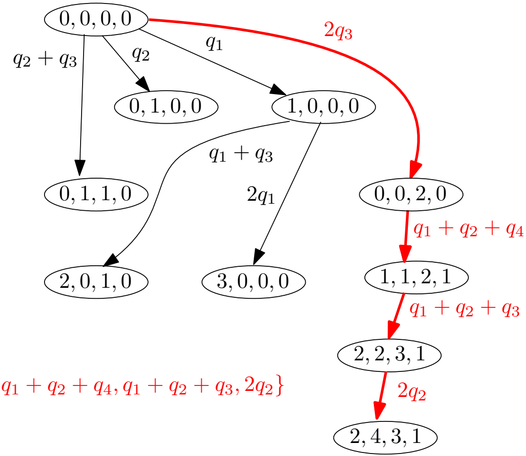
- a vertex $(a_1, \cdots, a_k)$, $a_i \in [0, n_i], \forall i \in [k]$
  - denotes the instance with a_1 jobs of size q_1, a_2 jobs of size q_2, $\cdots$, a_k jobs of size q_k
- an arc $(a_1, \cdots, a_k) \to (b_1, \cdots b_k)$ of weight $\sum_{i=1}^{k}(b_i - a_i)q_i$, if $a_i \leq b_i, \forall i \in [k]$, and $a_i < b_i$ for some $i \in [k]$
  - reducing instance $(b_1, \cdots b_k)$ to $(a_1, \cdots, a_k)$ requires 1 machine of load $\sum_{i=1}^{k}(b_i - a_i)q_i$

- Goal: find a path from $(0, \cdots, 0)$ to $(n_1, \cdots, n_k)$ of at most m edges, so as to minimize the maximum weight on the path.
- problem can be solved in $O(m \cdot |E|)$ time using DP
- $O(m \cdot |E|) = O(m \cdot n^{2k}) = n^{O\left(\frac{1}{\epsilon} \cdot \log \frac{1}{\epsilon}\right)}$.


0, 0, 0, 0
0, 1, 0, 0
1, 0, 0, 0
0, 1, 1, 0
2, 0, 1, 0
3, 0, 0, 0



0, 0, 0, 0
q2 + q3
q2
q1
0, 1, 0, 0
1, 0, 0, 0
q1 + q3
0, 1, 1, 0
2q1
2, 0, 1, 0
3, 0, 0, 0



$\text{cost} = \max\{2q_3, q_1 + q_2 + q_4, q_1 + q_2 + q_3, 2q_2\}$

Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j\in B}$ $\quad \mathrm{opt}_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j\in B}$ $\quad \mathrm{opt}'_B$: its optimum makespan

Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j \in B}$ $\quad \text{opt}_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j \in B}$ $\quad \text{opt}'_B$: its optimum makespan
- $\text{opt}'_B \leq \text{opt}_B$

Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j\in B}$ $\quad \mathrm{opt}_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j\in B}$ $\quad \mathrm{opt}'_B$: its optimum makespan
- $\mathrm{opt}'_B \leq \mathrm{opt}_B$
- schedule for $\mathcal{I}'_B \Rightarrow$ schedule for $\mathcal{I}_B$:
  $(1+\epsilon)$-blowup in makespan

Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j\in B}$ $\quad \mathrm{opt}_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j\in B}$ $\quad \mathrm{opt}'_B$: its optimum makespan
- $\mathrm{opt}'_B \leq \mathrm{opt}_B$
- schedule for $\mathcal{I}'_B \Rightarrow$ schedule for $\mathcal{I}_B$:

  $(1+\epsilon)$-blowup in makespan

Theorem The dynamic programming algorithm gives a schedule of makespan at most $(1+\epsilon)\mathrm{opt}_B$ in time $n^{O\left(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\right)}$.

Analysis of Algorithm for Big Jobs

- $\mathcal{I}_B$: instance $(p_j)_{j \in B}$ $\quad \mathrm{opt}_B$: its optimum makespan
- $\mathcal{I}'_B$: instance $(p'_j)_{j \in B}$ $\quad \mathrm{opt}'_B$: its optimum makespan
- $\mathrm{opt}'_B \leq \mathrm{opt}_B$
- schedule for $\mathcal{I}'_B \Rightarrow$ schedule for $\mathcal{I}_B$:
  $(1+\epsilon)$-blowup in makespan

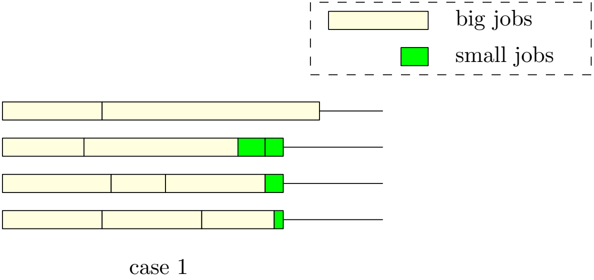
Theorem The dynamic programming algorithm gives a schedule of makespan at most $(1+\epsilon)\mathrm{opt}_B$ in time $n^{O\left(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\right)}$.

Adding small jobs to schedule

```
starting from the schedule for big jobs
for every small job j do
    add j to the machine with the smallest load
```

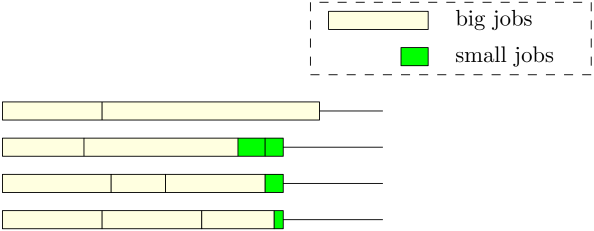
Outline

Analysis of the Final Algorithm



- Case 1: makespan is not increased by small jobs

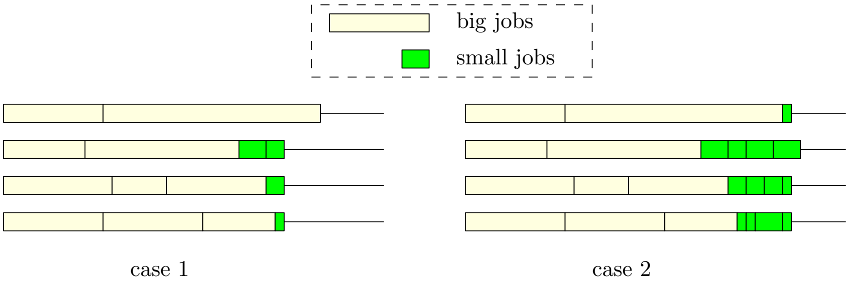
Analysis of the Final Algorithm



case 1

- Case 1: makespan is not increased by small jobs

$$\text{alg} \leq (1+\epsilon)\text{opt}_B \leq (1+\epsilon)\text{opt}.$$

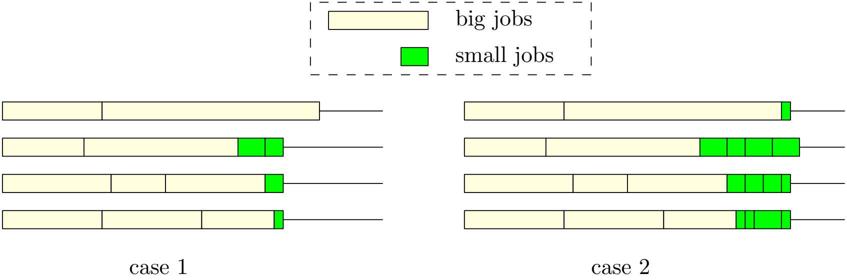


- Case 1: makespan is not increased by small jobs

$$\text{alg} \leq (1+\epsilon)\text{opt}_B \leq (1+\epsilon)\text{opt}.$$

- Case 2: makespan is increased by small jobs

Analysis of the Final Algorithm

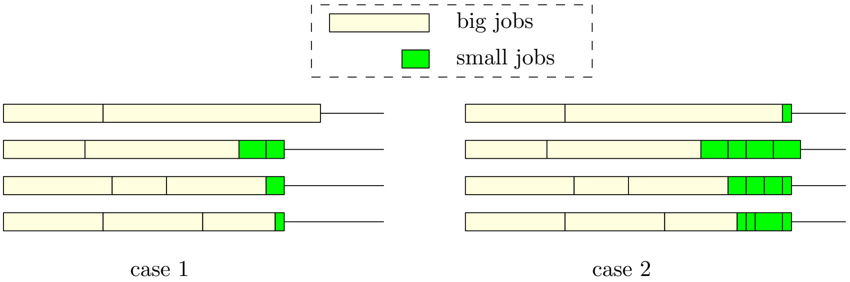


- Case 1: makespan is not increased by small jobs

$$\text{alg} \leq (1+\epsilon)\text{opt}_B \leq (1+\epsilon)\text{opt}.$$

- Case 2: makespan is increased by small jobs
  - loads between any two machines differ by at most size of a small job, which is at most $\epsilon \cdot p_{\max}$

Analysis of the Final Algorithm



- Case 1: makespan is not increased by small jobs

$$\text{alg} \leq (1+\epsilon)\text{opt}_B \leq (1+\epsilon)\text{opt}.$$

- Case 2: makespan is increased by small jobs
  - loads between any two machines differ by at most size of a small job, which is at most $\epsilon \cdot p_{\max}$

$$\text{alg} \leq \epsilon \cdot p_{\max} + \frac{1}{m}\sum_{j\in[n]} p_j \leq \epsilon \cdot \text{opt} + \text{opt} = (1+\epsilon)\cdot \text{opt}.$$

Outline

Outline

Bin Packing

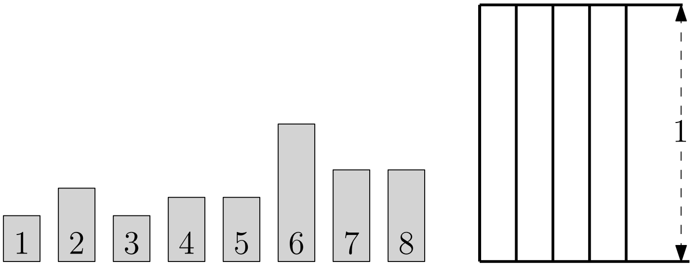
Input: n items indexed by $[n]$, with sizes $s_1, s_2, \cdots, s_n \in (0, 1]$

Output: a packing of items into smallest number of bins of capacity 1.

Bin Packing

Input: n items indexed by $[n]$, with sizes $s_1, s_2, \cdots, s_n \in (0, 1]$

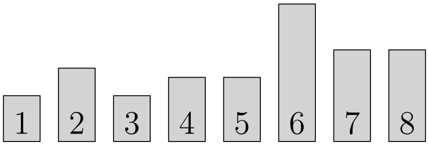
Output: a packing of items into smallest number of bins of capacity 1.

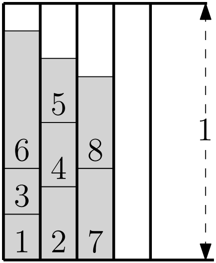


Bin Packing

Input: n items indexed by $[n]$, with sizes $s_1, s_2, \cdots, s_n \in (0, 1]$

Output: a packing of items into smallest number of bins of capacity 1.

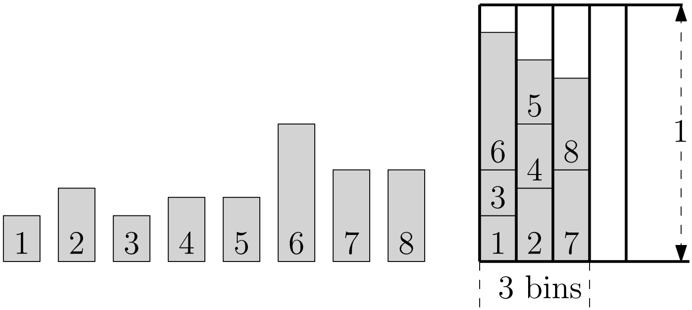




Bin Packing

Input: n items indexed by $[n]$, with sizes $s_1, s_2, \cdots, s_n \in (0, 1]$

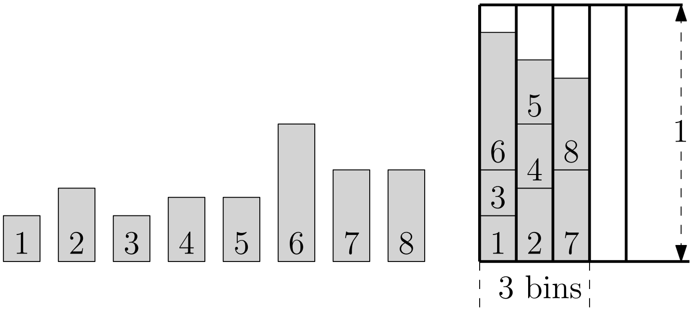
Output: a packing of items into smallest number of bins of capacity 1.



Bin Packing

Input: n items indexed by $[n]$, with sizes $s_1, s_2, \cdots, s_n \in (0, 1]$

Output: a packing of items into smallest number of bins of capacity 1.



	#containers	container capacity
bin packing	objective	fixed
scheduling	fixed	objective

First-Fit

```
initially there are 0 bins
for i ← 1 to n do
    if item i fits into an existing bin then put i into the bin
    else open a new bin and put i into the bin
```

First-Fit

```
initially there are 0 bins
for i ← 1 to n do
    if item i fits into an existing bin then put i into the bin
    else open a new bin and put i into the bin
```

Obs. In the output, at most one bin has total size $\le 1/2$.

First-Fit

initially there are 0 bins
for $i \leftarrow 1$ to n **do**
    if item i fits into an existing bin **then** put i into the bin
    else open a new bin and put i into the bin

Obs. In the output, at most one bin has total size $\leq 1/2$.

- If our algorithm uses t bins, then $\mathrm{opt} > \frac{t-1}{2}$ and $\mathrm{opt} \in \mathbb{Z}_{>0}$
- t is even: $\mathrm{opt} \geq \frac{t}{2}$ t is odd: $\mathrm{opt} \geq \frac{t+1}{2}$.

First-Fit

initially there are 0 bins
for $i \leftarrow 1$ to n **do**
    if item i fits into an existing bin **then** put i into the bin
    else open a new bin and put i into the bin

Obs. In the output, at most one bin has total size $\leq 1/2$.

- If our algorithm uses t bins, then $\mathrm{opt} > \frac{t-1}{2}$ and $\mathrm{opt} \in \mathbb{Z}_{>0}$
- t is even: $\mathrm{opt} \geq \frac{t}{2}$ t is odd: $\mathrm{opt} \geq \frac{t+1}{2}$.

Lemma The greedy algorithm gives a 2-approximation.

Theorem Unless P=NP, there is no poly-time approximation algorithm for bin packing with approximation ratio $< 3/2$.

Theorem Unless P=NP, there is no poly-time approximation algorithm for bin packing with approximation ratio $< 3/2$.

Proof.

- It is NP-hard to decide if whether the items can be packed into 2 bins or not, using the reduction from equal partition. □

Theorem Unless P=NP, there is no poly-time approximation algorithm for bin packing with approximation ratio $< 3/2$.

Proof.

- It is NP-hard to decide if whether the items can be packed into 2 bins or not, using the reduction from equal partition. □

Equal Partition

Input: n numbers $x_1, x_2, \cdots, x_n \in \mathbb{Z}_{>0}$

Output: decide if there is a partition of $[n]$ into A and B such that $\sum_{i \in A} x_i = \sum_{i \in B} x_i$

Theorem Equal Partition is (weakly) NP-hard.

- The approximation ratio is bad only when opt is small
- NP-hard to decide between $\mathrm{opt} \leq 2$ and $\mathrm{opt} \geq 3$.
- Open: NP-hard to decide between $\mathrm{opt} \leq 100$ and $\mathrm{opt} \geq 102$?

- The approximation ratio is bad only when opt is small
- NP-hard to decide between $\mathrm{opt} \leq 2$ and $\mathrm{opt} \geq 3$.
- Open: NP-hard to decide between $\mathrm{opt} \leq 100$ and $\mathrm{opt} \geq 102$?

- The conjecture has not been disproved (assuming P $\neq$ NP):

Conjecture: There is an efficient algorithm that outputs a solution with $\mathrm{opt} + 1$ bins.

- The approximation ratio is bad only when opt is small
- NP-hard to decide between $\text{opt} \leq 2$ and $\text{opt} \geq 3$.
- Open: NP-hard to decide between $\text{opt} \leq 100$ and $\text{opt} \geq 102$?
- The conjecture has not been disproved (assuming P $\neq$ NP):

Conjecture: There is an efficient algorithm that outputs a solution with $\text{opt} + 1$ bins.

- asymptotic α-approximation: an efficient algorithm that finds solution with $\alpha \cdot \text{opt} + c$ bins, with $c = O(1)$.

- The approximation ratio is bad only when opt is small
- NP-hard to decide between $\text{opt} \leq 2$ and $\text{opt} \geq 3$.
- Open: NP-hard to decide between $\text{opt} \leq 100$ and $\text{opt} \geq 102$?

- The conjecture has not been disproved (assuming P $\neq$ NP):

Conjecture: There is an efficient algorithm that outputs a solution with $\text{opt} + 1$ bins.

- asymptotic α-approximation: an efficient algorithm that finds solution with $\alpha \cdot \text{opt} + c$ bins, with $c = O(1)$.

Theorem First-Fit-Decreasing algorithm outputs a solution using at most $(11/9) \cdot \text{opt} + 4$ bins. That is, it is an asymptotic $11/9$-approximation.

Def. An asymptotic polynomial-time approximation scheme (APTAS) for minimization problems is a family of algorithms A_ϵ along with a constant $c \geq 0$, where algorithm A_ϵ for every $\epsilon > 0$ returns a solution of value at most $(1+\epsilon)\mathrm{opt} + c$ in polynomial time.

Def. An asymptotic polynomial-time approximation scheme (APTAS) for minimization problems is a family of algorithms A_ϵ along with a constant $c \geq 0$, where algorithm A_ϵ for every $\epsilon > 0$ returns a solution of value at most $(1+\epsilon)\mathrm{opt} + c$ in polynomial time.

Theorem For any fixed $\epsilon > 0$, there is a polynomial time algorithm that, given a bin-packing instance $\mathcal{I}$, outputs a solution with at most $(1+\epsilon)\mathrm{opt} + 1$ bins.

- That is, there is an APTAS for bin-packing.

- $\gamma > 0$ a small constant: item i is $\begin{cases} \text{small} & \text{if } s_i < \gamma \\ \text{big} & \text{if } s_i \geq \gamma \end{cases}$

- $\gamma > 0$ a small constant: item i is $\begin{cases} \text{small} & \text{if } s_i < \gamma \\ \text{big} & \text{if } s_i \geq \gamma \end{cases}$

What to do if all items are small?

- $\gamma > 0$ a small constant: item i is $\begin{cases} \text{small} & \text{if } s_i < \gamma \\ \text{big} & \text{if } s_i \geq \gamma \end{cases}$

What to do if all items are small?

- First-Fit: all but at most 1 bin has total size $\leq 1 - \gamma$
- $\mathrm{alg} \leq \left\lceil \frac{\mathrm{opt}}{1-\gamma} \right\rceil < \frac{1}{1-\gamma} \cdot \mathrm{opt} + 1, \quad \gamma := \epsilon/2 \quad \Rightarrow \quad \frac{1}{1-\gamma} < 1 + \epsilon$

- $\gamma > 0$ a small constant: item i is $\begin{cases} \text{small} & \text{if } s_i < \gamma \\ \text{big} & \text{if } s_i \geq \gamma \end{cases}$

What to do if all items are small?

- First-Fit: all but at most 1 bin has total size $\leq 1 - \gamma$
- $\mathrm{alg} \leq \left\lceil \frac{\mathrm{opt}}{1-\gamma} \right\rceil < \frac{1}{1-\gamma} \cdot \mathrm{opt} + 1, \quad \gamma := \epsilon/2 \quad \Rightarrow \quad \frac{1}{1-\gamma} < 1 + \epsilon$

What to do if all items are big?

- $\gamma > 0$ a small constant: item i is $\begin{cases} \text{small} & \text{if } s_i < \gamma \\ \text{big} & \text{if } s_i \geq \gamma \end{cases}$

What to do if all items are small?

- First-Fit: all but at most 1 bin has total size $\leq 1 - \gamma$
- $\mathrm{alg} \leq \left\lceil \frac{\mathrm{opt}}{1-\gamma} \right\rceil < \frac{1}{1-\gamma} \cdot \mathrm{opt} + 1, \quad \gamma := \epsilon/2 \quad \Rightarrow \quad \frac{1}{1-\gamma} < 1 + \epsilon$

What to do if all items are big?

- truncate item sizes to obtain $\mathcal{I}'$, using DP to solve $\mathcal{I}'$
- two essential properties:

  $\mathrm{opt}(\mathcal{I}') \approx \mathrm{opt}(\mathcal{I})$ #(item sizes in $\mathcal{I}'$) is small

- $\gamma > 0$ a small constant: item i is $\begin{cases} \text{small} & \text{if } s_i < \gamma \\ \text{big} & \text{if } s_i \geq \gamma \end{cases}$

What to do if all items are small?

- First-Fit: all but at most 1 bin has total size $\leq 1 - \gamma$
- $\text{alg} \leq \left\lceil \frac{\text{opt}}{1-\gamma} \right\rceil < \frac{1}{1-\gamma} \cdot \text{opt} + 1, \quad \gamma := \epsilon/2 \quad \Rightarrow \quad \frac{1}{1-\gamma} < 1 + \epsilon$

What to do if all items are big?

- truncate item sizes to obtain $\mathcal{I}'$, using DP to solve $\mathcal{I}'$
- two essential properties:

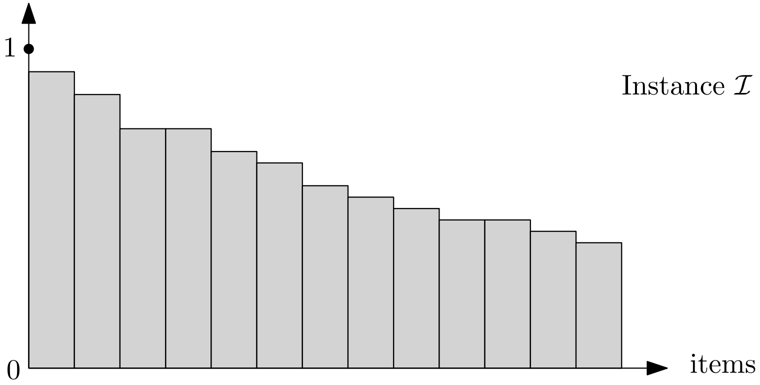
  $\text{opt}(\mathcal{I}') \approx \text{opt}(\mathcal{I})$ #(item sizes in $\mathcal{I}'$) is small

- general instance: pack big items using truncation + DP, then use First-Fit to pack small items

Outline

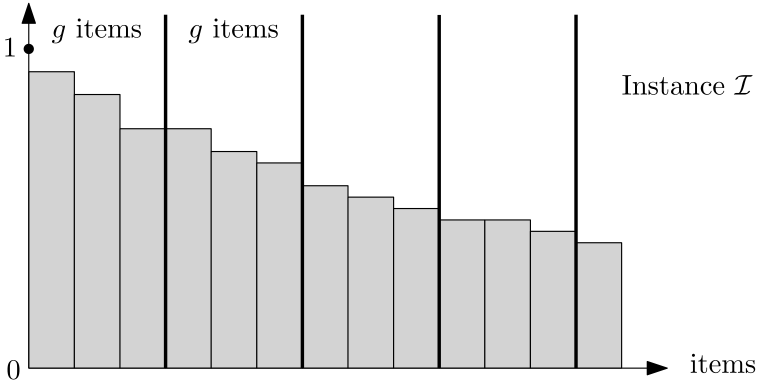
Construction of Instance $\mathcal{I}'$

1: sort items in non-increasing sizes



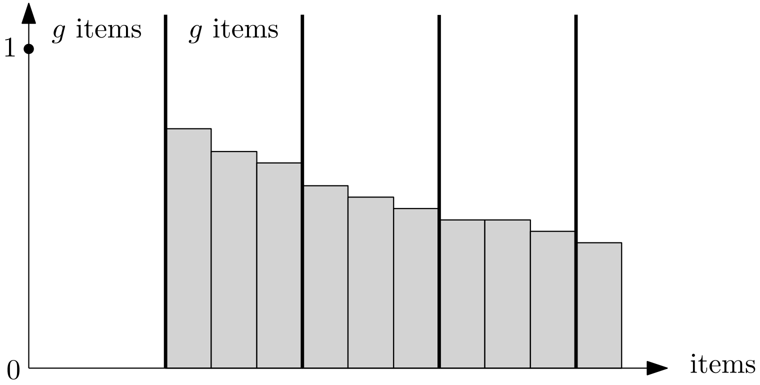
Construction of Instance $\mathcal{I}'$

1: sort items in non-increasing sizes
2: partition items into groups of size g



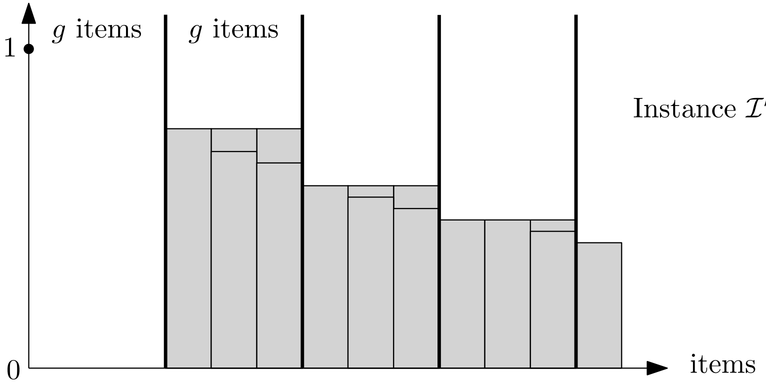
Construction of Instance $\mathcal{I}'$

1: sort items in non-increasing sizes
2: partition items into groups of size g
3: discard the first group



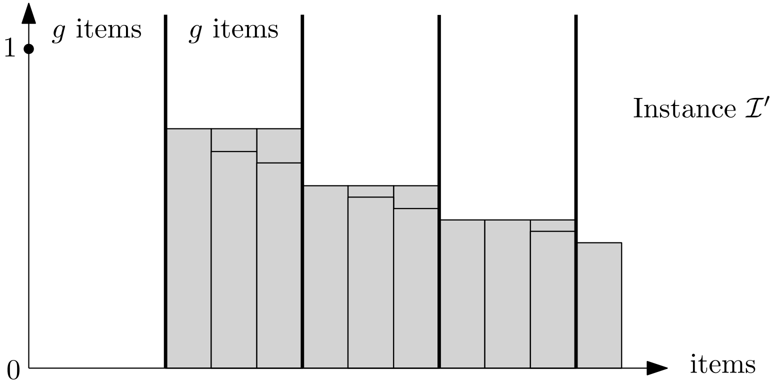
Construction of Instance $\mathcal{I}'$

1: sort items in non-increasing sizes
2: partition items into groups of size g
3: discard the first group
4: **for** each of the other groups **do**
5: change item size to the biggest size in group



Construction of Instance $\mathcal{I}'$

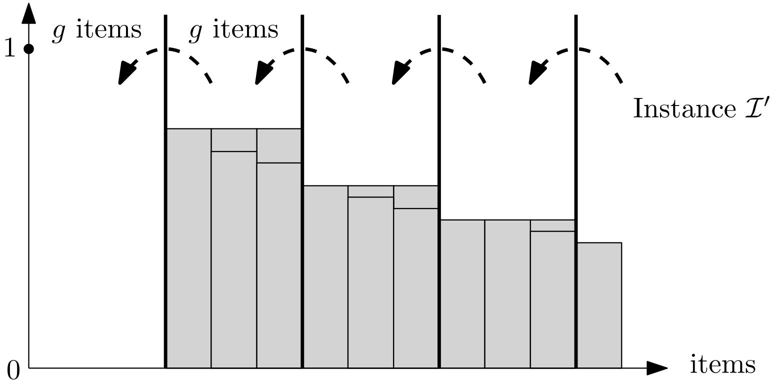
sort items in non-increasing sizes
partition items into groups of size g
discard the first group
for each of the other groups **do**
    change item size to the biggest size in group



$\mathrm{opt}(\mathcal{I}) - g \leq \mathrm{opt}(\mathcal{I}')$

Construction of Instance $\mathcal{I}'$

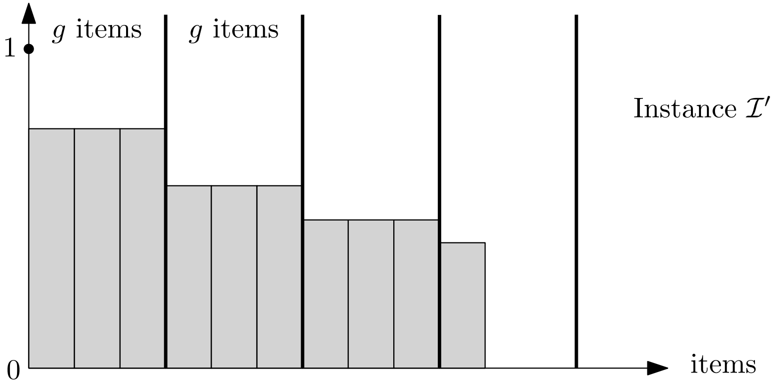
1: sort items in non-increasing sizes
2: partition items into groups of size g
3: discard the first group
4: **for** each of the other groups **do**
5: change item size to the biggest size in group



$\text{opt}(\mathcal{I}) - g \le \text{opt}(\mathcal{I}')$

Construction of Instance $\mathcal{I}'$

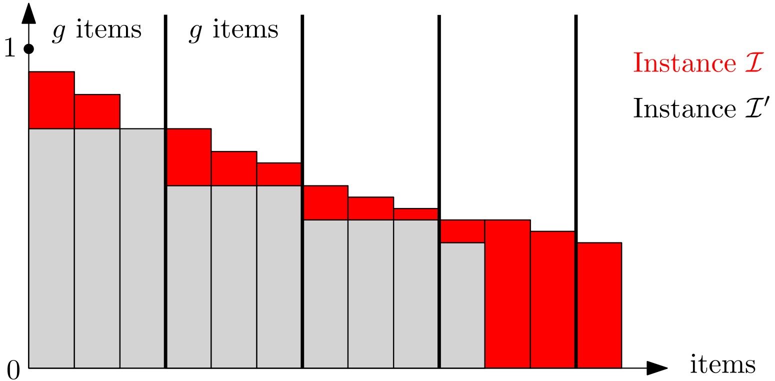
```
sort items in non-increasing sizes
partition items into groups of size g
discard the first group
for each of the other groups do
    change item size to the biggest size in group
```



$$\mathrm{opt}(\mathcal{I}) - g \leq \mathrm{opt}(\mathcal{I}')$$

Construction of Instance $\mathcal{I}'$

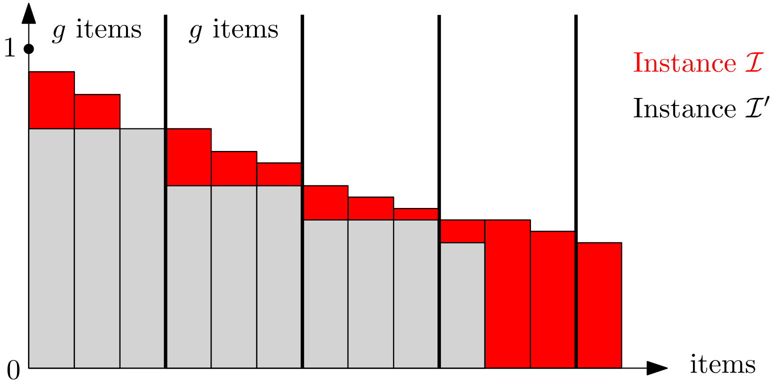
```
sort items in non-increasing sizes
partition items into groups of size g
discard the first group
for each of the other groups do
    change item size to the biggest size in group
```



$$\text{opt}(\mathcal{I}) - g \leq \text{opt}(\mathcal{I}')$$

Construction of Instance $\mathcal{I}'$

```
sort items in non-increasing sizes
partition items into groups of size g
discard the first group
for each of the other groups do
    change item size to the biggest size in group
```



$$\text{opt}(\mathcal{I}) - g \leq \text{opt}(\mathcal{I}') \leq \text{opt}(\mathcal{I})$$

- every group in $\mathcal{I}'$ has the same size.

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$
- $\mathcal{I}'$ can be solved exactly by DP in $O(n^{2k})$-time

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$
- $\mathcal{I}'$ can be solved exactly by DP in $O(n^{2k})$-time

Dynamic Programming for $\mathcal{I}'$ in $O(n^{2k})$-time

- let $s^{(1)} \geq s^{(2)} \geq \cdots \geq s^{(k)}$ be the k distinct sizes

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$
- $\mathcal{I}'$ can be solved exactly by DP in $O(n^{2k})$-time

Dynamic Programming for $\mathcal{I}'$ in $O(n^{2k})$-time

- let $s^{(1)} \geq s^{(2)} \geq \cdots \geq s^{(k)}$ be the k distinct sizes
- let $n_1, n_2, \cdots, n_k$ be the number of items of each size

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$
- $\mathcal{I}'$ can be solved exactly by DP in $O(n^{2k})$-time

Dynamic Programming for $\mathcal{I}'$ in $O(n^{2k})$-time

- let $s^{(1)} \geq s^{(2)} \geq \cdots \geq s^{(k)}$ be the k distinct sizes
- let $n_1, n_2, \cdots, n_k$ be the number of items of each size
- vertex $(a_1, a_2, \cdots, a_k)$: the instance with a_1 items of size $s^{(1)}$, a_2 items of size $s^{(2)}$, $\cdots$, and a_k items of size $s^{(k)}$

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$
- $\mathcal{I}'$ can be solved exactly by DP in $O(n^{2k})$-time

Dynamic Programming for $\mathcal{I}'$ in $O(n^{2k})$-time

- let $s^{(1)} \geq s^{(2)} \geq \cdots \geq s^{(k)}$ be the k distinct sizes
- let $n_1, n_2, \cdots, n_k$ be the number of items of each size
- vertex $(a_1, a_2, \cdots, a_k)$: the instance with a_1 items of size $s^{(1)}$, a_2 items of size $s^{(2)}$, $\cdots$, and a_k items of size $s^{(k)}$
- an arc $(a_1, a_2, \cdots, a_k) \to (b_1, b_2, \cdots, b_k)$ if
  - $a_i \geq b_i$ for every $i \in [k]$ and,
  - $s^{(1)}(b_1 - a_1) + s^{(2)}(b_2 - a_2) + \cdots + s^{(k)}(b_k - a_k) \leq 1$

- every group in $\mathcal{I}'$ has the same size.
- $k :=$ the number of distinct sizes in $\mathcal{I}'$, $k \leq \left\lfloor \frac{n}{g} \right\rfloor$
- $\mathcal{I}'$ can be solved exactly by DP in $O(n^{2k})$-time

Dynamic Programming for $\mathcal{I}'$ in $O(n^{2k})$-time

- let $s^{(1)} \geq s^{(2)} \geq \cdots \geq s^{(k)}$ be the k distinct sizes
- let $n_1, n_2, \cdots, n_k$ be the number of items of each size
- vertex $(a_1, a_2, \cdots, a_k)$: the instance with a_1 items of size $s^{(1)}$, a_2 items of size $s^{(2)}$, $\cdots$, and a_k items of size $s^{(k)}$
- an arc $(a_1, a_2, \cdots, a_k) \to (b_1, b_2, \cdots, b_k)$ if
  - $a_i \geq b_i$ for every $i \in [k]$ and,
  - $s^{(1)}(b_1 - a_1) + s^{(2)}(b_2 - a_2) + \cdots + s^{(k)}(b_k - a_k) \leq 1$
- DP: computing the shortest path from $(0, 0, \cdots, 0)$ to $(n_1, n_2, \cdots, n_k)$

$$\mathrm{opt}(\mathcal{I}) - g \leq \mathrm{opt}(\mathcal{I}') \leq \mathrm{opt}(\mathcal{I}).$$

$$\mathrm{opt}(\mathcal{I}) - g \le \mathrm{opt}(\mathcal{I}') \le \mathrm{opt}(\mathcal{I}).$$

- solving $\mathcal{I}'$ $\quad\Rightarrow\quad$ packing for $\mathcal{I}$ with $\le \mathrm{opt}(\mathcal{I}) + g$ bins

$$\mathrm{opt}(\mathcal{I}) - g \le \mathrm{opt}(\mathcal{I}') \le \mathrm{opt}(\mathcal{I}).$$

- solving $\mathcal{I}'$ $\quad \Rightarrow \quad$ packing for $\mathcal{I}$ with $\le \mathrm{opt}(\mathcal{I}) + g$ bins
- $s_i \ge \gamma, \forall i \in [n] \qquad \Longrightarrow \qquad \mathrm{opt}(\mathcal{I}) \ge \gamma n$.

$$\mathrm{opt}(\mathcal{I}) - g \leq \mathrm{opt}(\mathcal{I}') \leq \mathrm{opt}(\mathcal{I}).$$

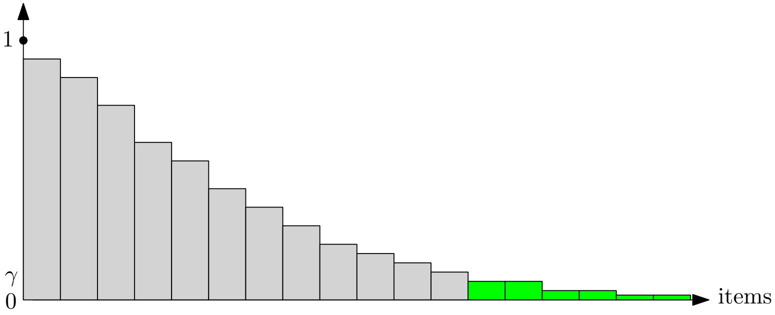
- solving $\mathcal{I}'$ $\quad\Rightarrow\quad$ packing for $\mathcal{I}$ with $\leq \mathrm{opt}(\mathcal{I}) + g$ bins
- $s_i \geq \gamma, \forall i \in [n] \qquad \Longrightarrow \qquad \mathrm{opt}(\mathcal{I}) \geq \gamma n$.
- setting $g := \epsilon\gamma n \qquad \Longrightarrow \qquad g \leq \epsilon \cdot \mathrm{opt}(\mathcal{I})$ and $k \leq \frac{n}{g} \leq \frac{1}{\epsilon\gamma}$

$$\mathrm{opt}(\mathcal{I}) - g \leq \mathrm{opt}(\mathcal{I}') \leq \mathrm{opt}(\mathcal{I}).$$

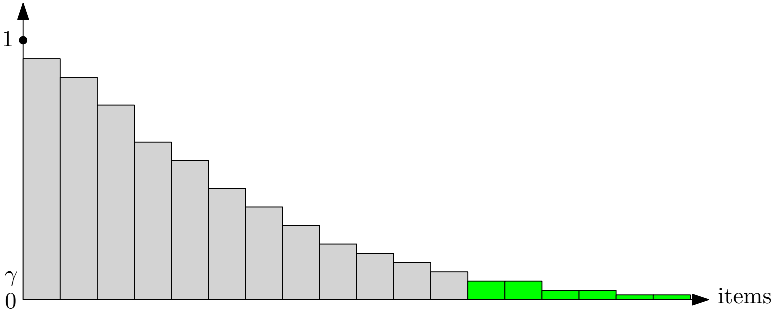
- solving $\mathcal{I}'$ $\quad\Rightarrow\quad$ packing for $\mathcal{I}$ with $\leq \mathrm{opt}(\mathcal{I}) + g$ bins
- $s_i \geq \gamma, \forall i \in [n] \quad\Longrightarrow\quad \mathrm{opt}(\mathcal{I}) \geq \gamma n$.
- setting $g := \epsilon\gamma n \quad\Longrightarrow\quad g \leq \epsilon \cdot \mathrm{opt}(\mathcal{I})$ and $k \leq \frac{n}{g} \leq \frac{1}{\epsilon\gamma}$

Theorem There is an $O(n^{2/(\epsilon\gamma)})$-time $(1+\epsilon)$-approximation algorithm for the bin-packing problem when all items have size at least γ,

Outline

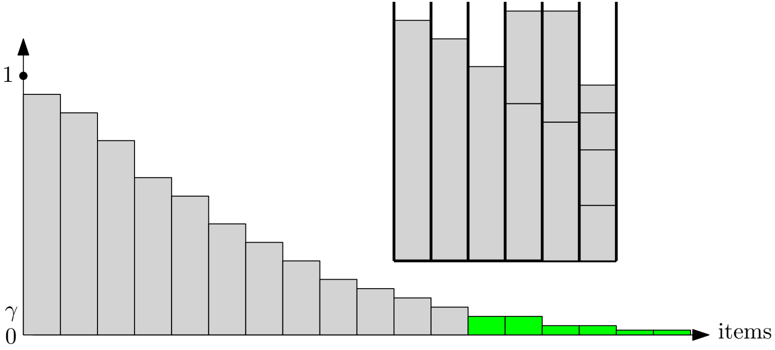


Combining Algorithms for Small and Big Items



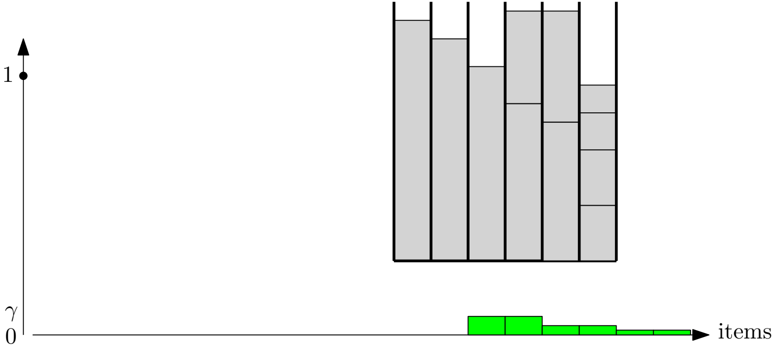
Combining Algorithms for Small and Big Items

1: Use truncation + DP to obtain solution $\mathcal{S}$ for big items



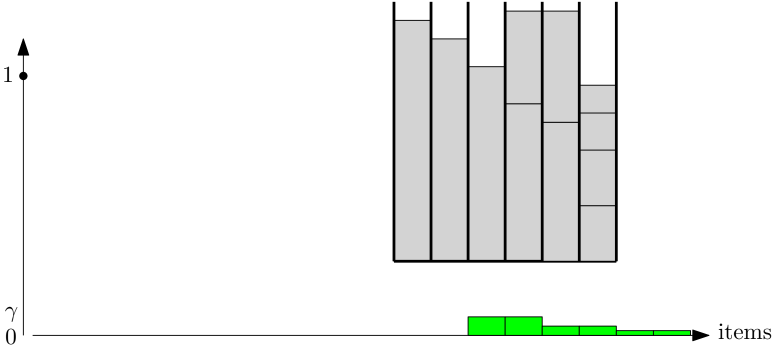
Combining Algorithms for Small and Big Items

1: Use truncation + DP to obtain solution $\mathcal{S}$ for big items



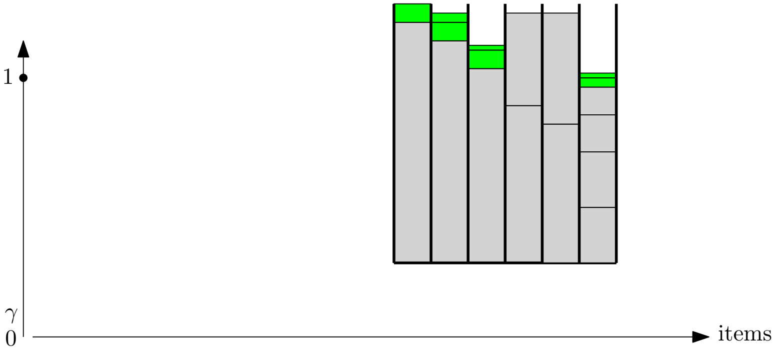
Combining Algorithms for Small and Big Items

1: Use truncation + DP to obtain solution $\mathcal{S}$ for big items



Combining Algorithms for Small and Big Items

1: Use truncation + DP to obtain solution $\mathcal{S}$ for big items
2: Starting from $\mathcal{S}$, use First-Fit to pack small items

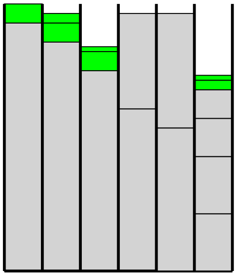


Combining Algorithms for Small and Big Items

1: Use truncation + DP to obtain solution $\mathcal{S}$ for big items
2: Starting from $\mathcal{S}$, use First-Fit to pack small items

Analysis of the Combined Algorithm

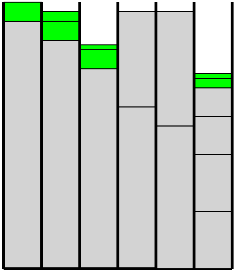
Analysis of the Combined Algorithm



case 1

- Case 1: no new bins are used to pack small items

Analysis of the Combined Algorithm

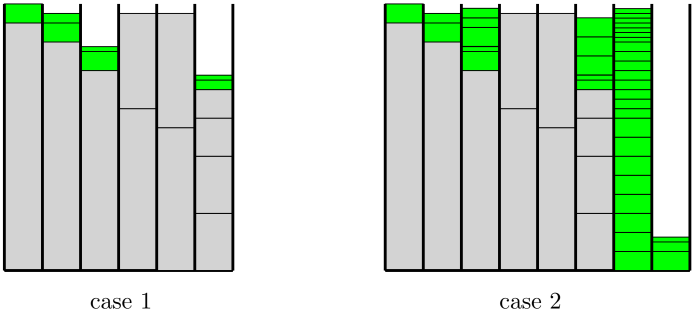


case 1

- Case 1: no new bins are used to pack small items

$$\#(\text{bins used}) \leq (1+\epsilon)\cdot \mathrm{opt}(\mathcal{I}_{\mathrm{big}}) \leq (1+\epsilon)\cdot \mathrm{opt}(\mathcal{I})$$

Analysis of the Combined Algorithm

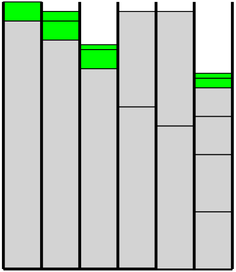


- Case 1: no new bins are used to pack small items

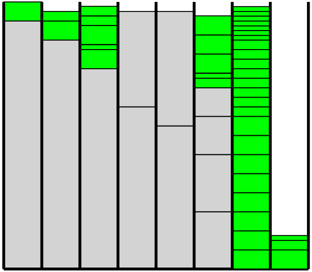
$$\#(\text{bins used}) \leq (1+\epsilon) \cdot \mathrm{opt}(\mathcal{I}_{\mathrm{big}}) \leq (1+\epsilon) \cdot \mathrm{opt}(\mathcal{I})$$

- Case 2: new bins are used

Analysis of the Combined Algorithm



case 1



case 2

- Case 1: no new bins are used to pack small items

$$\#(\text{bins used}) \leq (1+\epsilon)\cdot \mathrm{opt}(\mathcal{I}_{\mathrm{big}}) \leq (1+\epsilon)\cdot \mathrm{opt}(\mathcal{I})$$

- Case 2: new bins are used
  at most one bin has total size $\leq 1-\gamma$

$$\#(\text{bins used}) < \frac{\mathrm{opt}(\mathcal{I})}{1-\gamma} + 1$$

- Setting $\gamma = \epsilon/2 \quad \implies$
  #(bins used) $< \frac{\mathrm{opt}(\mathcal{I})}{1-\epsilon/2} + 1 \leq (1+\epsilon)\mathrm{opt}(\mathcal{I}) + 1$

Theorem There is an $O(n^{2/(\epsilon^2)})$-time algorithmn that outputs a solution with at most $(1+\epsilon)\mathrm{opt}(\mathcal{I}) + 1$ bins.

Theorem There is an APTAS for bin-packing.