Algorithm Design and Analysis

Approximation Algorithms

Goals for today

- Understand the **motivation** and **definition** of approximation algorithms
- Demonstrate three common techniques for approximation algorithms:
  - **Greedy** (Job Scheduling)
  - **LP rounding** (Vertex Cover)
  - **Scaling** (Knapsack)

Approximation algorithms: what & why

- Some problems are hard to solve (e.g., NP-Hard problems)
- What can we do when faced with such problems?
  - Give up?
  - Implement **heuristics/pruning** that speed up "common" inputs. Algorithm is still worst-case exponential time but fast enough for many "real life" inputs.

***Idea (approximation algorithms)*: Try to find a solution that is not necessarily optimal but is **provably close to optimal**, with an efficient (polynomial time) algorithm.**

Formal definition

Definition (c-approximation algorithm):

- Consider an optimization problem (minimize or maximize)
- Say the value of the optimal solution is **OPT**
- Say that our algorithm outputs a solution with value **ALG**
- Our algorithm is a ***c*-approximation** if

Minimization Problems

$$ALG \leq c \cdot OPT$$

The solution is at most $c > 1$ times **too big**

Maximization Problems

$$ALG \geq c \cdot OPT$$

The solution is at most $c < 1$ times **too small**

Technique #1

Greedy algorithms

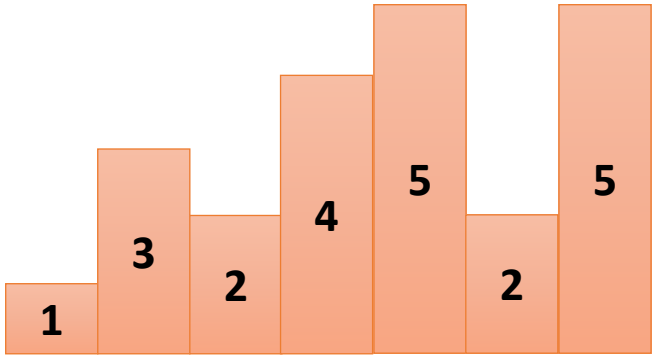
Job Scheduling

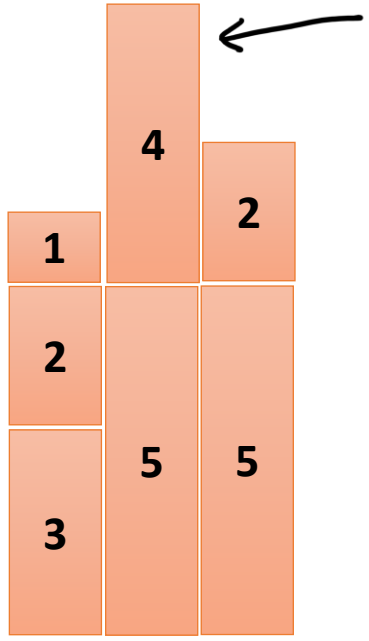
***Problem*: Given m identical "machines" and n "jobs", where job i takes p_i processing time to run, assign jobs to machines to minimize the *makespan*, the time at which the last job finishes**

***Alternative interpretation*: Given n blocks where block i has height p_i, we want to make m stacks of blocks, with the goal of minimizing the height of the tallest stack**

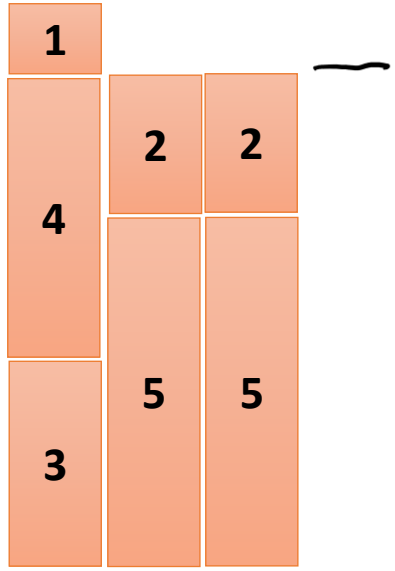
Job Scheduling

Example: $p = \{1, 3, 2, 4, 5, 2, 5\}, m = 3$





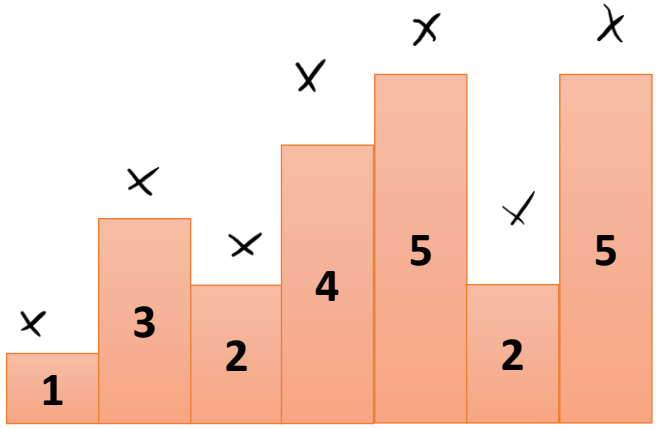
Makespan = 9

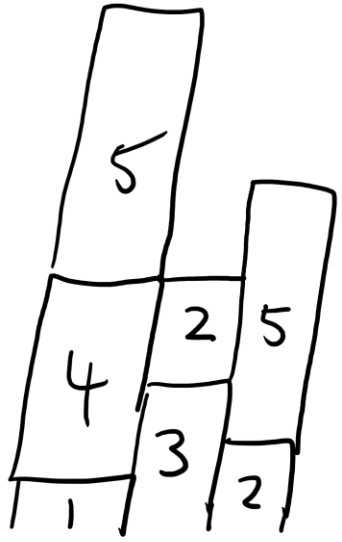


Makespan = 8

Approximation algorithm for job scheduling

<u>Algorithm: Greedy job scheduling</u>
start with m empty stacks
for each block
    add the block to the shortest current stack





ALG = 10

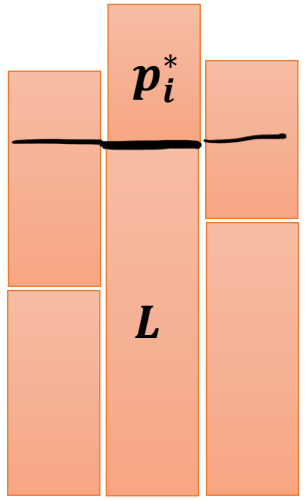
Analysis of greedy job scheduling

Claim: Greedy job scheduling is a **2-approximation** algorithm

Proof: WTS ALG ≤ 2·OPT

- Let p_i^* be the height of the *last block* on the tallest stack
- Let L be the remaining height of the tallest stack
  - OPT ≥ $\Sigma p_i / m$
  - OPT ≥ p_i^*
  - OPT ≥ L

$ALG = p_i^* + L \leq OPT + OPT = 2\,OPT$



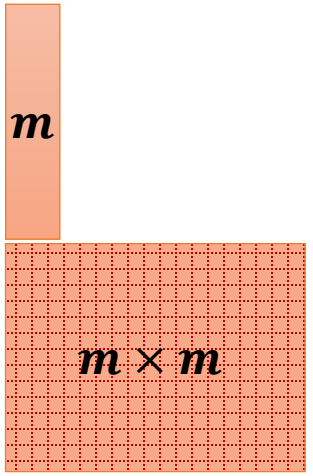
$ALG = p_i^* + L$

Can we do better than 2?

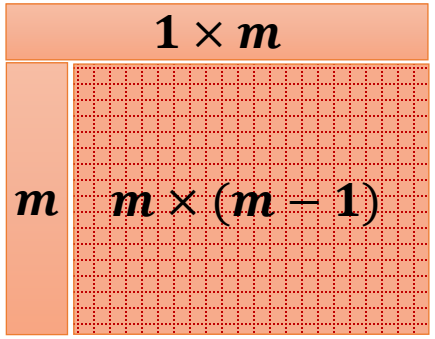
Question: What is a worst-case input for greedy scheduling?

Idea: Small stuff first, big stuff at the end

Example: m^2 blocks of size 1, then one block of size m



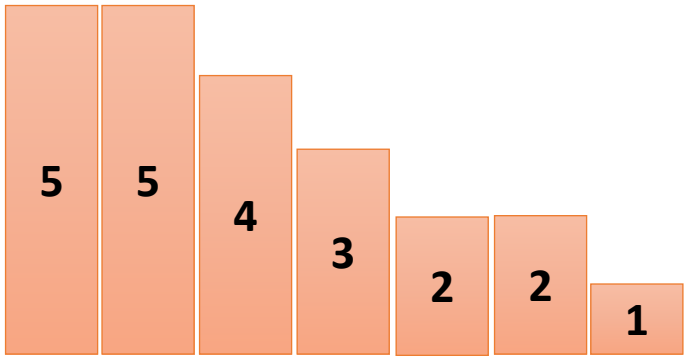
ALG = $2m$

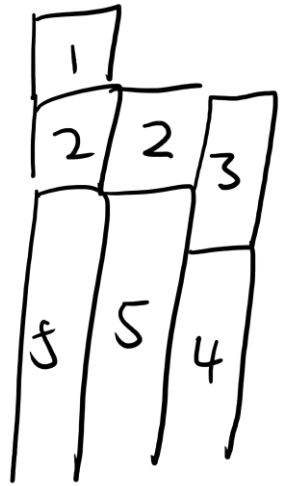


OPT = $m + 1$

Better algorithm for job scheduling

<u>Algorithm: Sorted greedy job scheduling</u>
start with m empty stacks
for each block i **in order from big to small**
    add block i to the shortest current stack





ALG = 8

Analysis of sorted greedy job scheduling

Claim: Sorted greedy job scheduling is a **1.5**-approximation algorithm

Proof: ALG ≤ 1.5 OPT

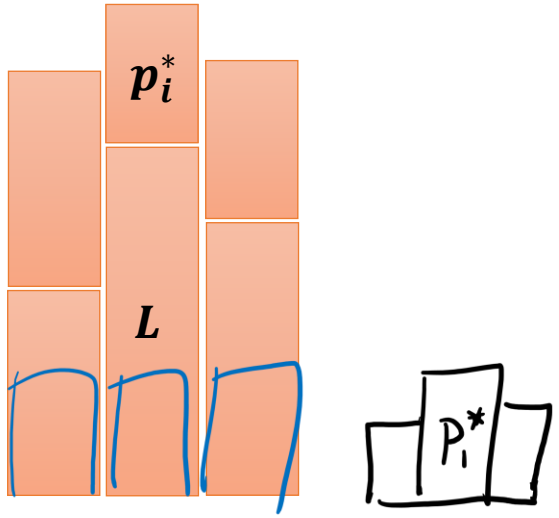
- $OPT \geq p_i^*$ and $OPT \geq L$ still true

- At least (m+1) blocks of size p_i^* (if $L > 0$)
- One stack has at least two
- $OPT \geq 2p_i^*$

$\Rightarrow$ $ALG = p_i^* + L \leq \frac{1}{2}OPT + OPT = 1.5\,OPT$

(If $L = 0$) $ALG = p_i^* = OPT$

□



Summary of Greedy

Take-home messages:

- Greedy algorithms are often good approximations
- Hardest part is the proof
  - Need to find a way to connect OPT to ALG
  - Often achieved by lower bounding OPT and relating this to ALG

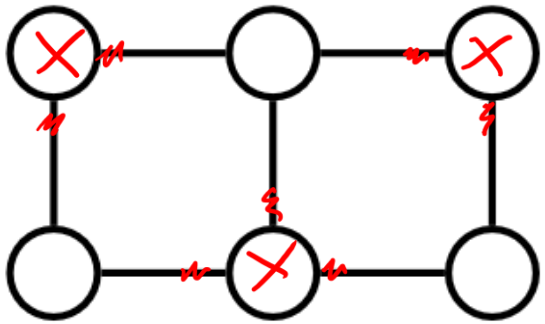
Technique #2

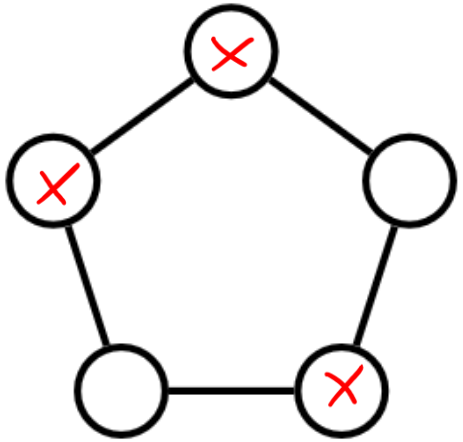
LP Rounding

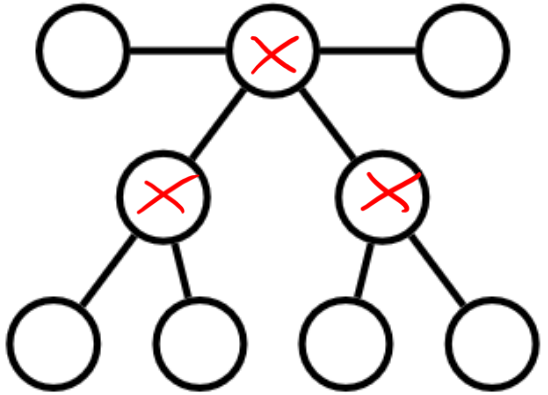
Problem: Vertex Cover

Problem*: Given an undirected graph $G = (V, E)$, a ***vertex cover is a subset of the vertices $C \subseteq V$ such that every edge is adjacent to at least one $v \in C$.**

A ***minimum vertex cover*** is a smallest possible vertex cover



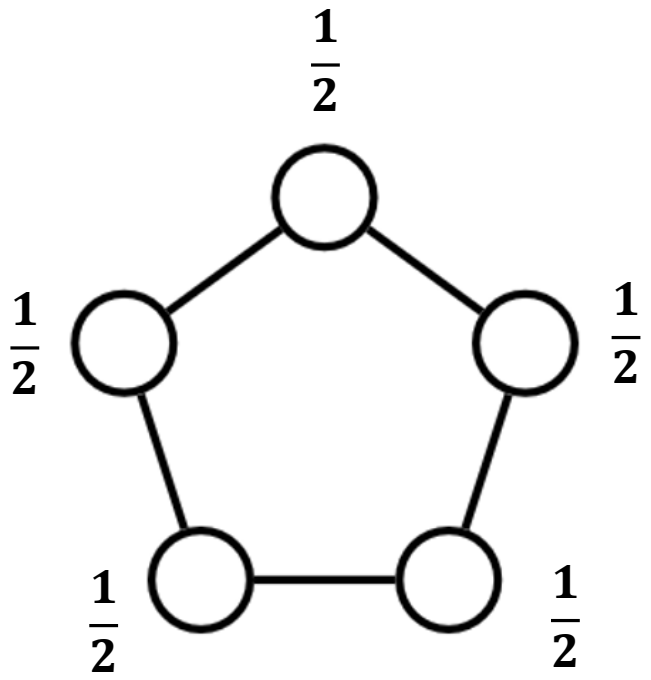




Linear program (relaxation) for vertex cover

Variables x_v for each vertex v

$$\begin{aligned} \text{minimize} \quad & \sum_{v \in V} x_v \\ \text{s.t.} \quad & x_u + x_v \geq 1 \quad \text{for all } (u,v) \in E \\ & x_v \geq 0 \quad \text{for all } v \in V \end{aligned}$$



Remember: Can give fractional solutions

Approximation algorithm for vertex cover

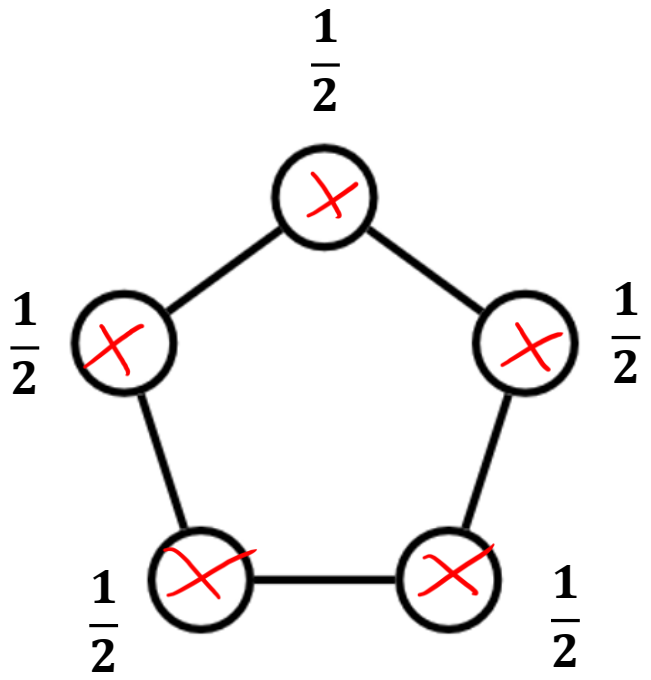
<u>Algorithm: Rounding vertex cover</u>

Solve the LP relaxation for x_v

for each vertex v

  if $x_v \geq 1/2$ **then**

    add v to the vertex cover



Analysis of LP rounding for vertex cover

Claim 1: The LP rounding algorithm outputs a valid vertex cover

Proof: AFSOC (u, v) is not covered

$x_u < 1/2$ $x_v < 1/2$

$x_u + x_v < 1$ (Infeasible)

Contradiction!

Analysis of LP rounding for vertex cover

***Claim 2*: The LP rounding algorithm is a 2-approximation algorithm**

Proof: Rounding x_v from $1/2$ to 1 doubles (at most)

Objective = $\sum x_v$ at most doubles

$$ALG \leq 2 \cdot LP \leq 2 \cdot OPT$$

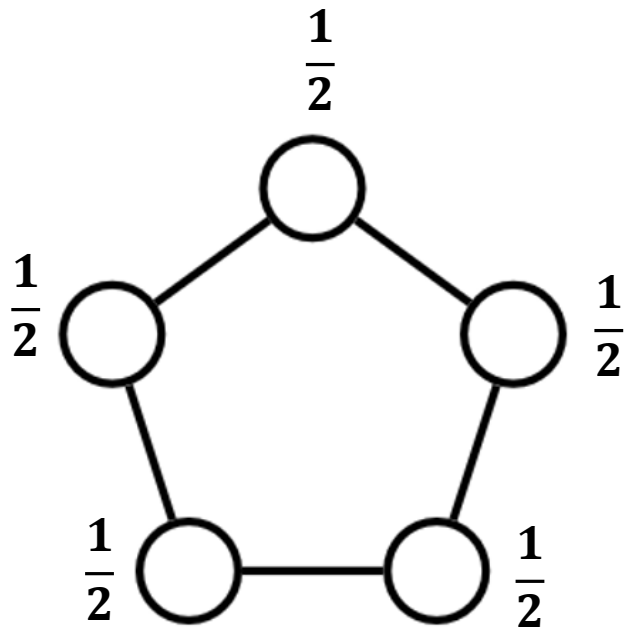
$LP \leq OPT$: relaxation of a minimization problem

□

Check your understanding

Question: Can we apply this algorithm to *any* LP relaxation and get a 2-approximation? Why or why not

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in E} x_e \\ \text{s.t.} \quad & \sum_{\substack{e \text{ s.t.} \\ v \in e}} x_e \leq 1 \qquad \text{for all } v \in V \\ & x_e \geq 0 \qquad \text{for all } e \in E \end{aligned}$$



- Rounding up would violate constraints
- Rounding down would give a low value (zero)

Technique #3

Scaling

Problem: Knapsack

Definition (Knapsack): Given a set of n **items**, the i^{th} of which has **size** s_i and **value** v_i. The goal is to find a subset of the items whose **total size is at most** S, with **maximum possible value**.

- In Lecture 10, we devised a DP solution that runs in $O(nS)$ time where S is the size of the knapsack.
- This is **pseudopolynomial** time, i.e., polynomial in the input numbers but not in the input size
- Efficient only if S is small

Alternative DP formulation

V = max value of any item

$O(n^2 V)$

- We can alternatively make the runtime depend polynomially on the **values** rather than **size/weight**

$G(k, P) =$ Minimum weight of a subset of items $\{1, \ldots, k\}$ with value $\geq P$

$$G(k,P) = \begin{cases} 0 & \text{if } k = 0 \text{ and } P \leq 0 \\ \infty & \text{if } k = 0 \text{ and } P > 0 \\ \min\{G(k-1, P), G(k-1, P - V_k) + S_k\} & \text{otherwise} \end{cases}$$

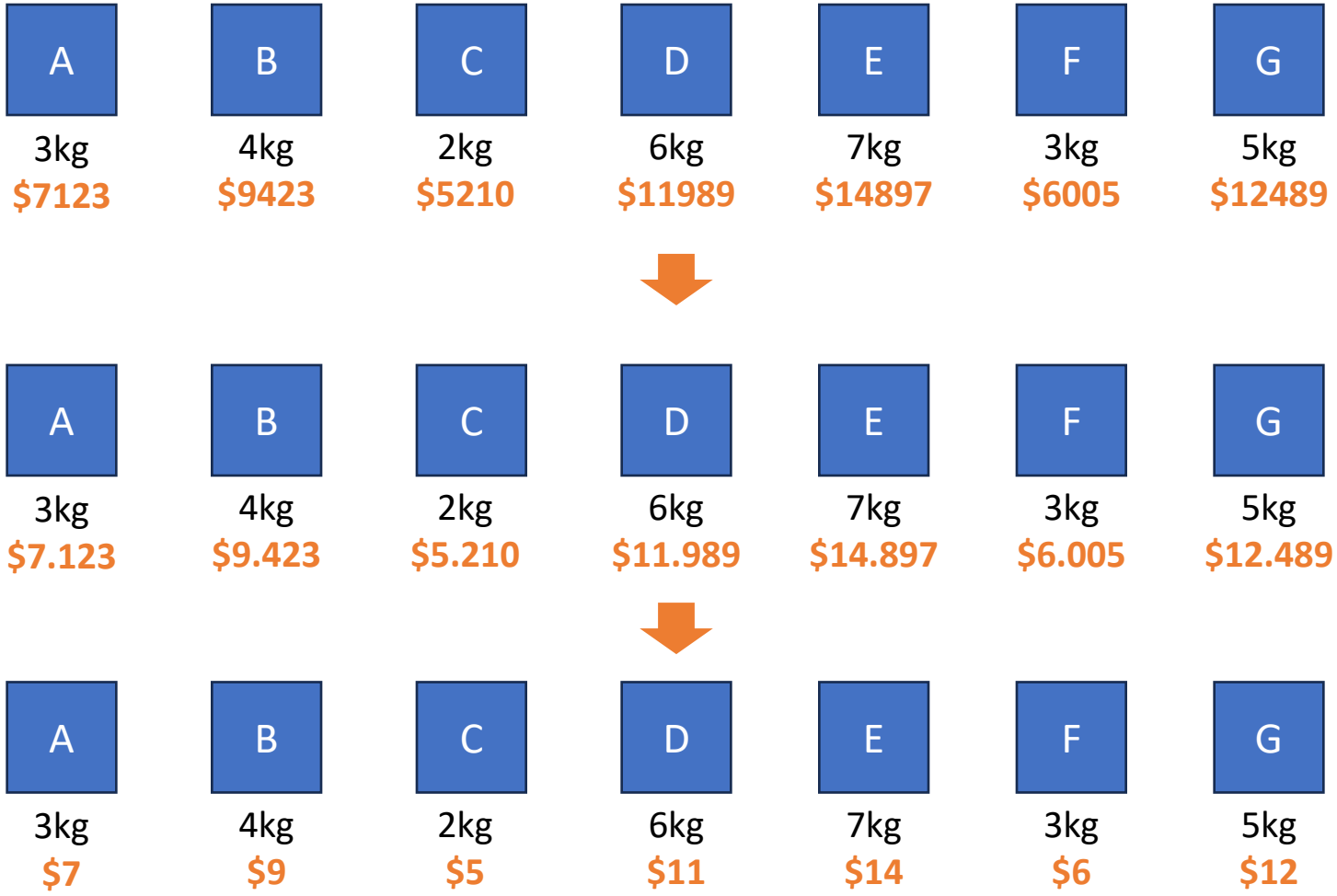
Scaling: The Idea

- We have a **pseudopolynomial-time** algorithm running $O(n^2V)$ where V is the maximum value of any item.
- This is efficient if V is small
- So, let's just make it small?

***Idea (scaling)*: When the runtime depends on a number in the input, scale those numbers down and round them, introducing small error.**

Scaling: The Idea

$O(n^2 W)$



Question: What should the scaling factor be?

The scaling algorithm

all values are now in $\{0, \ldots, 10n\}$

- Scale all values down by a factor of $\boldsymbol{k = \frac{V}{10n}}$, i.e., set $v_i' = \left\lfloor \frac{v_i}{k} \right\rfloor$
- Solve the scaled problem and output the optimal set of items

Claim: This algorithm runs in $O(n^3)$ *time*

Proof: $O(n^2 V) = O(n^3)$ because max value $\leq 10n$

□

The scaling algorithm

- Scale all values down by a factor of $k = \frac{V}{10n}$, i.e., set $v_i' = \left\lfloor \frac{v_i}{k} \right\rfloor$
- Solve the scaled problem and output the optimal set of items

Claim: This algorithm is a 0.9-approximation

Proof: "Loss" per item $v_i - v_i' \cdot k = v_i - \left\lfloor \frac{v_i}{k} \right\rfloor \cdot k \leq k$

"Loss" for whole subset $\leq nk = \frac{V}{10}$

$ALG \geq OPT - \frac{V}{10} \geq OPT - \frac{OPT}{10} = 0.9\ OPT$

Rounding error

Scaling more generally

- The constant of 10 was arbitrary and gave us a 0.9 approximation
- Can scale by $\frac{\varepsilon V}{n}$ to get a $(1-\varepsilon)$ approximation!
- This is called a ***polynomial-time approximation scheme*** (PTAS). We can get any constant factor we want!
- Works for other dynamic programming algorithms that run in pseudopolynomial time

Summary

- We **defined** the concept of ***approximation algorithms***
- We practiced **three techniques** for building approximation algorithms:
  - **Greedy** (Job Scheduling)
  - **LP rounding** (Vertex Cover)
  - **Scaling** (Knapsack)