# Main algorithm design strategies

- ✓ ***Exhaustive Computation*.** Generate every possible candidate solution and select an optimal solution.
- ***Greedy*.** Create next candidate solution one step at a time by using some greedy choice.
- ***Divide and Conquer*.** Divide the problem into non-overlapping subproblems of the same type, solve each subproblem with the same algorithm, and combine sub-solutions into a solution to the entire problem.
- ***Dynamic Programming*.** Start with the smallest subproblem and combine optimal solutions to smaller subproblems into optimal solution for larger subproblems, until the optimal solution for the entire problem is constructed.
- ***Iterative Improvement.*** Perform multiple iterations of the algorithm, at each iteration moving closer to the optimal solution, until no further improvement is possible.

*The point is, ladies and gentlemen, greed is good.*
*Greed works, greed is right.*
*Greed clarifies, cuts through, and captures the essence of the evolutionary spirit.*
*Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind.*
*And greed—mark my words—will save not only Teldar Paper but the other malfunctioning corporation called the USA.*

*Gordon Gekko [Michael Douglas], Wall Street (1987)*

https://www.youtube.com/watch?v=VVxYOQS6ggk

*There is always an easy solution to every human problem - neat, plausible, and wrong.*

*H. L. Mencken, "The Divine Afflatus",*
*New York Evening Mail (November 16, 1917)*

# Greedy algorithms: a gentle introduction

Lecture 05.01

by *Marina Barsky*

# **Greedy** Algorithms

- *Optimization problem*: search through all candidate configurations to find a solution with some *min/max* value of an *objective function*

- The goal: to find this optimal solution (configuration)
- We can do it with Exhaustive Search, but we hope that a Greedy Algorithm can lead us to the goal faster

- A *greedy algorithm* repeatedly takes the next step that is 'best' according to some “greedy choice”

- It ***hopes*** that with each choice it will move closer to the goal

# **Greedy** Algorithms

Advantages:

- They are **easy** to discover
- They are **easy** to describe
- They are often **easy** to implement
- They are often **efficient**

Disadvantages:

- They are often **incorrect**
- Since the intuitive idea rarely works in practice, **you have to prove** that your greedy algorithm produces an optimal solution!

PROBLEM 1

# The Change Making Problem

Cashier algorithm

# Change Making Problem

- When a customer pays for an item there is often a surplus that must be returned as change

- The goal of the ***Change Making Problem*** is to refund a target surplus value using **as few coins as possible**


US Currency includes the following coins:

1 ¢ , 5 ¢ , 10 ¢ , 25 ¢ , 50 ¢ , 100 ¢

# Change Making: example

- For example, if the target is 45 ¢ , then an optimal solution is 25 ¢ ,10 ¢ ,10 ¢
- Another solution that is not optimal is 10 ¢ ,10 ¢ ,10 ¢ ,10 ¢ ,5 ¢
- Notice that we count the total number of coins and not the number of distinct coins


US Currency includes the following coins:

1 ¢ , 5 ¢ , 10 ¢ , 25 ¢ , 50 ¢ , 100 ¢

Link to play

# Change Problem: formalized

*Convert some amount of money into given denominations, using the smallest possible number of coins*

## Change Problem

***Input:*** An integer *target* and an array of $d$ denominations $c = (c_1, c_2, \ldots, c_d)$, in decreasing order of value $(c_1 > c_2 > \cdots > c_d)$.

***Output:*** A list of $d$ integers $i_1, i_2, \ldots, i_d$ such that

$c_1 \cdot i_1 + c_2 \cdot i_2 + \cdots + c_d \cdot i_d = target$,

and $i_1 + i_2 + \cdots + i_d$ is as small as possible.

configuration →

objective function

# Designing a greedy algorithm

When designing a greedy algorithm we think what could be the 'best' next step in the current situation

We call this single step a ***greedy move***

What could the greedy step be for the change-making problem?

- We are choosing one coin at a time
- The goal is to use the fewest number of coins

# Designing a greedy algorithm

When designing a greedy algorithm we think what could be the 'best' next step in the current situation

We call this single step a ***greedy move***

What could the greedy step be for the change-making problem?

- We are choosing one coin at a time
- The goal is to use the fewest number of coins

- It seems that we make the most progress by **choosing the largest valued coin** that doesn't exceed the surplus that currently remains

# Greedy Algorithm in English

- At each step, add to the solution the largest-valued coin that doesn't exceed the surplus that currently remains

- Until the surplus is 0

# Example of Greedy Change Making

The greedy algorithm on target value 48 ¢ with coin set [100 ¢ , 50 ¢ , 25 ¢ , 10 ¢ , 5 ¢ , 1 ¢ ]

| Surplus | Coin |
|---|---|
| 48 ¢ | |
| 23 ¢ | 25 ¢ |
| 13 ¢ | 10 ¢ |
| 3 ¢ | 10 ¢ |
| 2 ¢ | 1 ¢ |
| 1 ¢ | 1 ¢ |
| 0 | 1 ¢ |
| | |

25 ¢ , 10 ¢ , 10 ¢ , 1 ¢ , 1 ¢ , 1 ¢ - 6 coins for 48 ¢

This is an optimal solution!

# Cashier's Algorithm: "pseudocode"

**Algorithm *change* (target, c, d)**

***surplus* ← *target***
**while *surplus* > 0:**
    ***coin* ← coin with the largest denomination**
        **that does not exceed *surplus***
    **give coin with denomination *coin* to customer**
    ***surplus* ← *surplus* – *coin***

# Cashier's Algorithm: pseudocode

**Algorithm *change* (target, array c of d denominations)**

$c_1 > c_2 > \cdots > c_d$

Initialize array $(i_1, i_2, \ldots, i_d)$ with all zeros

$surplus \leftarrow target$

**for** $k$ **from** 1 **to** $d$:

$i_k \leftarrow \lfloor \frac{surplus}{c_k} \rfloor$

$surplus \leftarrow surplus - c_k \cdot i_k$

**return** $(i_1, i_2, \ldots, i_d)$

Running time O(d)

(Assuming the denominations are sorted descending)

Does choosing coins with the largest denomination always lead to the optimal solution?

# You are in India

Consider the set of coins {1,5,10,20,25,50} rupees and the target value 40 rupees.

- The greedy change algorithm gives:

25, 10, 5 3 coins

What is an optimal solution?

# You are in India

Consider the set of coins {1,5,10,20,25,50} rupees and the target value 40 rupees.

- The greedy change algorithm gives:

25, 10, 5 — 3 coins

- Optimal solution:

20, 20 — 2 coins

*Note*: in 1875 a twenty-cent coin existed in the United States

# US postage

***change* is an incorrect algorithm!**

- Consider U.S. postage: 1¢, 10¢, 21¢, 34¢, 70¢, 100¢, 350¢, 1225¢, 1500¢.
- ***change*** algorithm: 140¢ = 100 + 34 + 1 + 1 + 1 + 1 + 1 + 1.
- ***optimal***: 140¢ = 70 + 70.

# Good News / Bad News

**Theorem**: The greedy change algorithm always gives an optimal solution when the $i^{th}$ greatest denomination is divisible by the $(i+1)^{st}$ greatest denomination

However, it also works in many other cases including US coins {100 ¢ , 50 ¢ , 25 ¢ , 10 ¢ , 5 ¢ , 1 ¢ } and Euro coins {100 ¢ , 50 ¢ , 20 ¢ , 10 ¢ , 5 ¢ , 2 ¢ , 1 ¢ }

# When to use the "cashier" algorithm

**Theorem**: The greedy change algorithm always gives an optimal solution when the $i^{th}$ greatest denomination is divisible by the $(i+1)^{st}$ greatest denomination

In other words, it works when taking 1 coin with denomination $c_i = qc_{i+1}$ is always preferred than taking $q$ coins of denomination $c_{i+1}$

Example:
Denominations: $c_1 = \mathbf{25}$, $c_2 = \mathbf{20}$, $c_3 = 10$, $c_4 = 5$, and $c_5 = 1$.
Target: 40

**This set does not work:** when we add $c_1 = 25$ to our solution, the next denomination $c_2 = 20$ taken twice would fill more surplus and may lead to an overall better solution, but by taking $c_1 = 25$ we miss this opportunity because there is no surplus left to fit another $c_2 = 20$

# So what is the correct algorithm?

- Exhaustive search?
- We could consider every possible combination of coins with denominations $c_1, c_2, \ldots, c_d$ that adds to *target*, and return the combination with the fewest number of coins.
- We only need to consider combinations with

$i_1 \leq target/c_1$ , $i_2 \leq target/c_2$ ...

(in general, $i_k$ should not exceed $target/c_k$ , because we would otherwise be returning an amount of money larger than *target*)

Consider all possible vectors $[i_1, i_2, \cdots i_d]$ such that $c_1 \cdot i_1 + c_2 \cdot i_2 + \cdots + c_d \cdot i_d = target$

*Each $i_k$ can take values from 0 to target/$i_k$*

**Algorithm exhaustive_change(*target*, *c*, *d*):**

*min_num_coins* ← ∞
for each ($i_1$, . . . , $i_d$) from (0, . . . , 0) to (*target*/$c_1$,..., *target*/$c_d$)
    *value* ← $\sum_{k=1}^{d} i_k \cdot c_k$
    if *value* = *target*
        *num_coins* ← $\sum_{k=1}^{d} i_k$
        if *num_coins* < *min_num_coins*
            *min_num_coins* ← *num_coins*

return *min_num_coins*

# Running time of *exhaustive_change*

- To calculate the total number of operations in the for loop, we can take the approximate number of operations performed in each iteration and multiply this by the total number of iterations.
- Since there are roughly

$$\frac{target}{c_1} \times \frac{target}{c_2} \times \ldots \times \frac{target}{c_d}$$

possible vectors of size d, the algorithm performs about $d \times \frac{target^d}{c_1 c_2 \cdots c_d}$ operations, and thus the running time is exponential in the number of different coin denominations *d.*

We will revisit this problem in the Dynamic Programming Unit

# General Greedy Strategy

- Make some greedy choice
- Reduce to a smaller problem
- Iterate (or Recur) on a smaller problem

## Definition

A greedy choice is called a *safe move* if there is an optimal solution consistent with this first move

PROBLEM 2

# Maximum Loot Problem

## Fractional Knapsack

# Maximum Value of the Loot

- A thief breaks into a spice shop and finds
  - **4** pounds of saffron with total cost **$20,000**
  - **3** pounds of vanilla with total cost **$600**
  - **5** pounds of cinnamon with total cost **$50**
- His backpack fits at most **9** pounds, therefore he cannot take everything
- The thief would like to **maximize** the total value of spices in his backpack

# Problem: Fractional knapsack

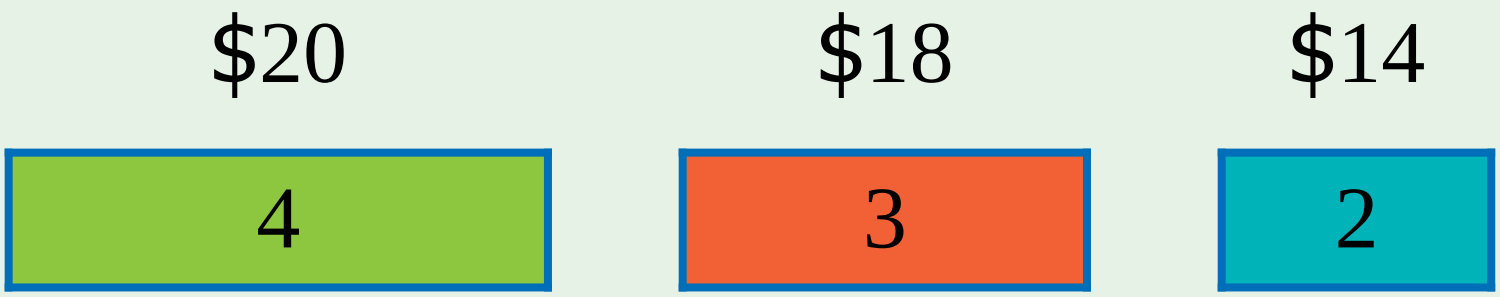
**Input:** The capacity of a backpack $W$ as well as the weights $(w_1,\ldots, w_n)$ and total benefit values $(v_1,\ldots, v_n)$ for $n$ different items.

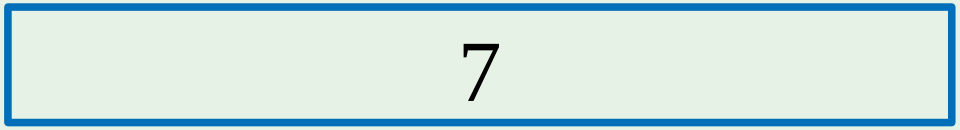
**Output:** The set of items (or their fractions) with the **maximum** benefit that fit into the backpack.

# Possible greedy moves

- Fill with the most expensive item first
- First take an item that fills least space
- First take an item that fills most space
- ?

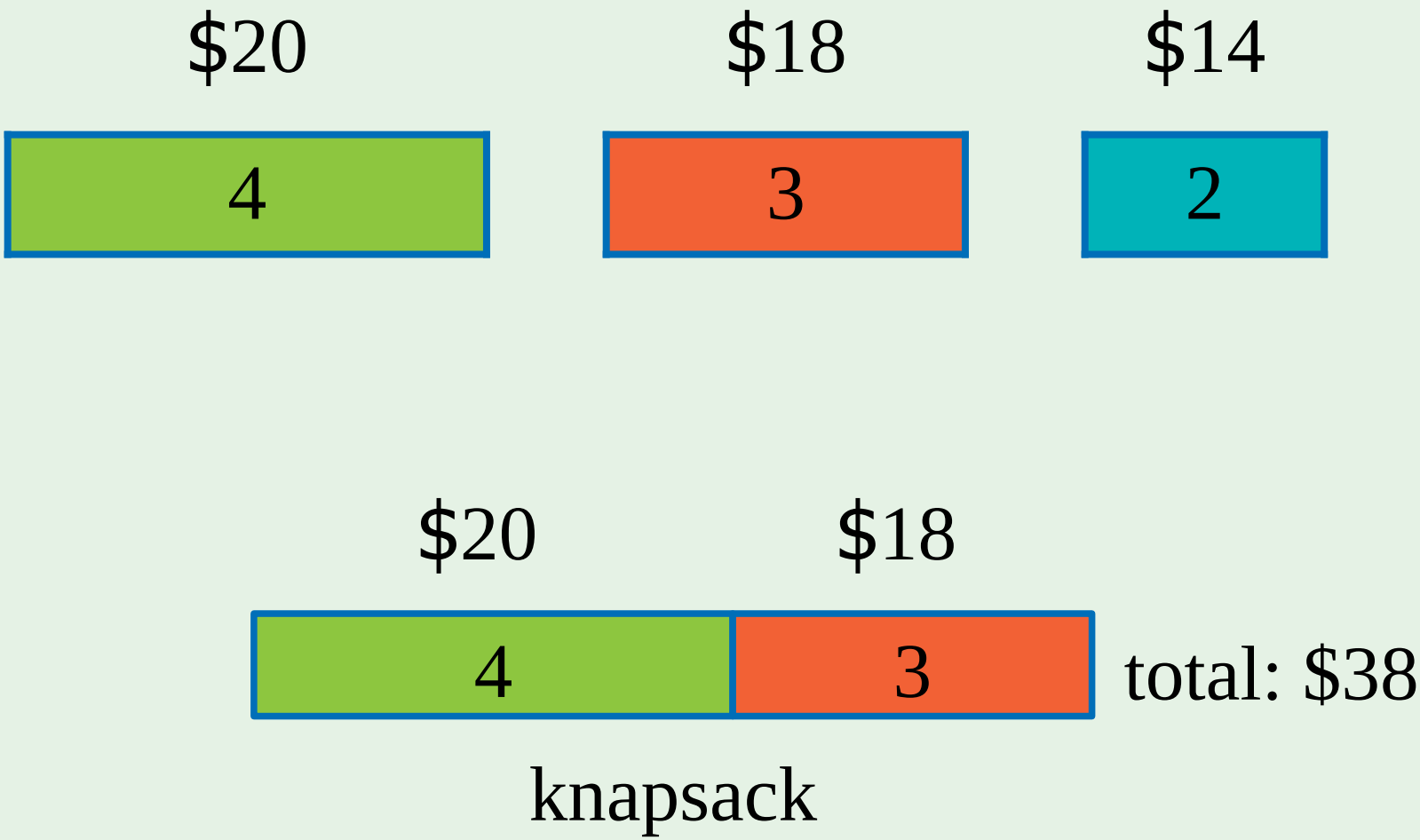
# Move 1: fill with the **most expensive** first



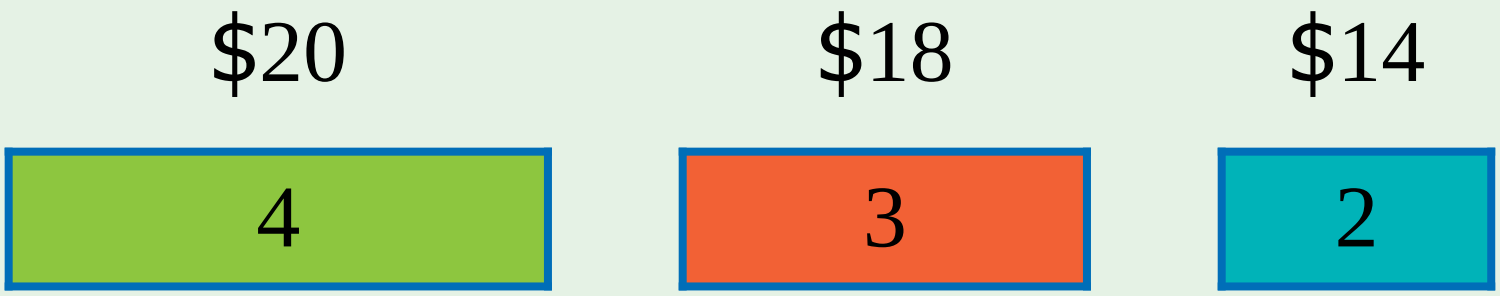



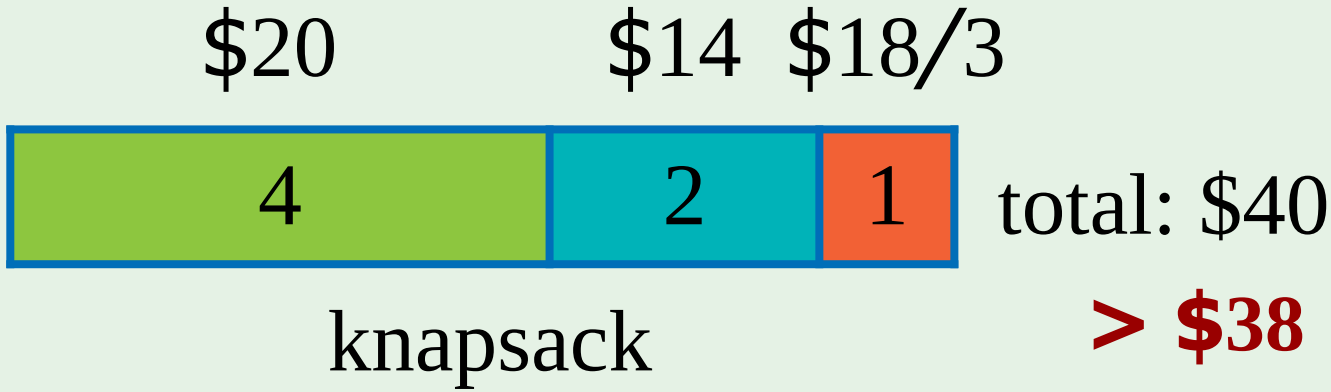

knapsack

# Move 1: fill with the **most expensive** first



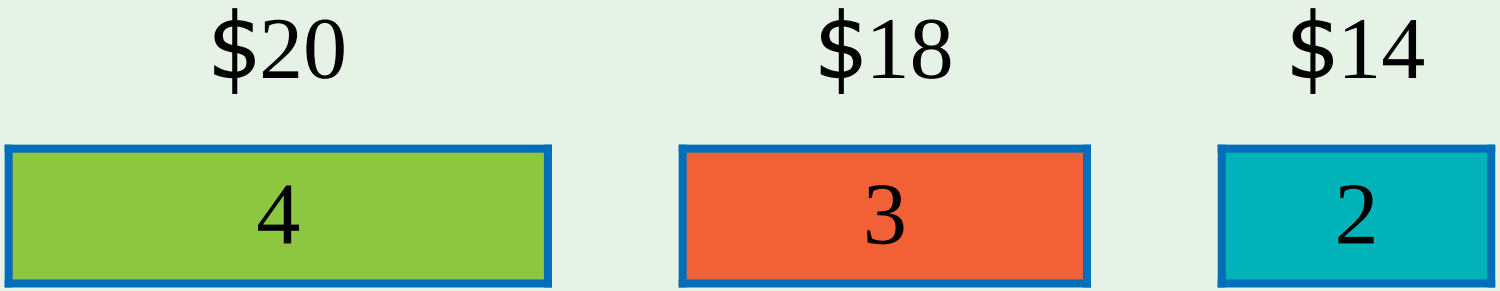
# Move 1 is not safe: counterexample

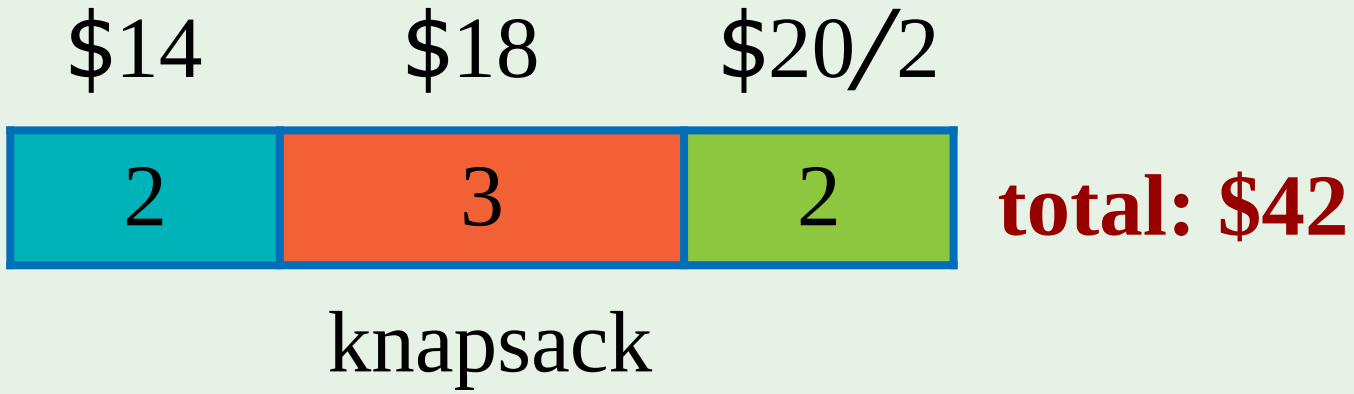


# Optimal solution

## What is the correct greedy move?

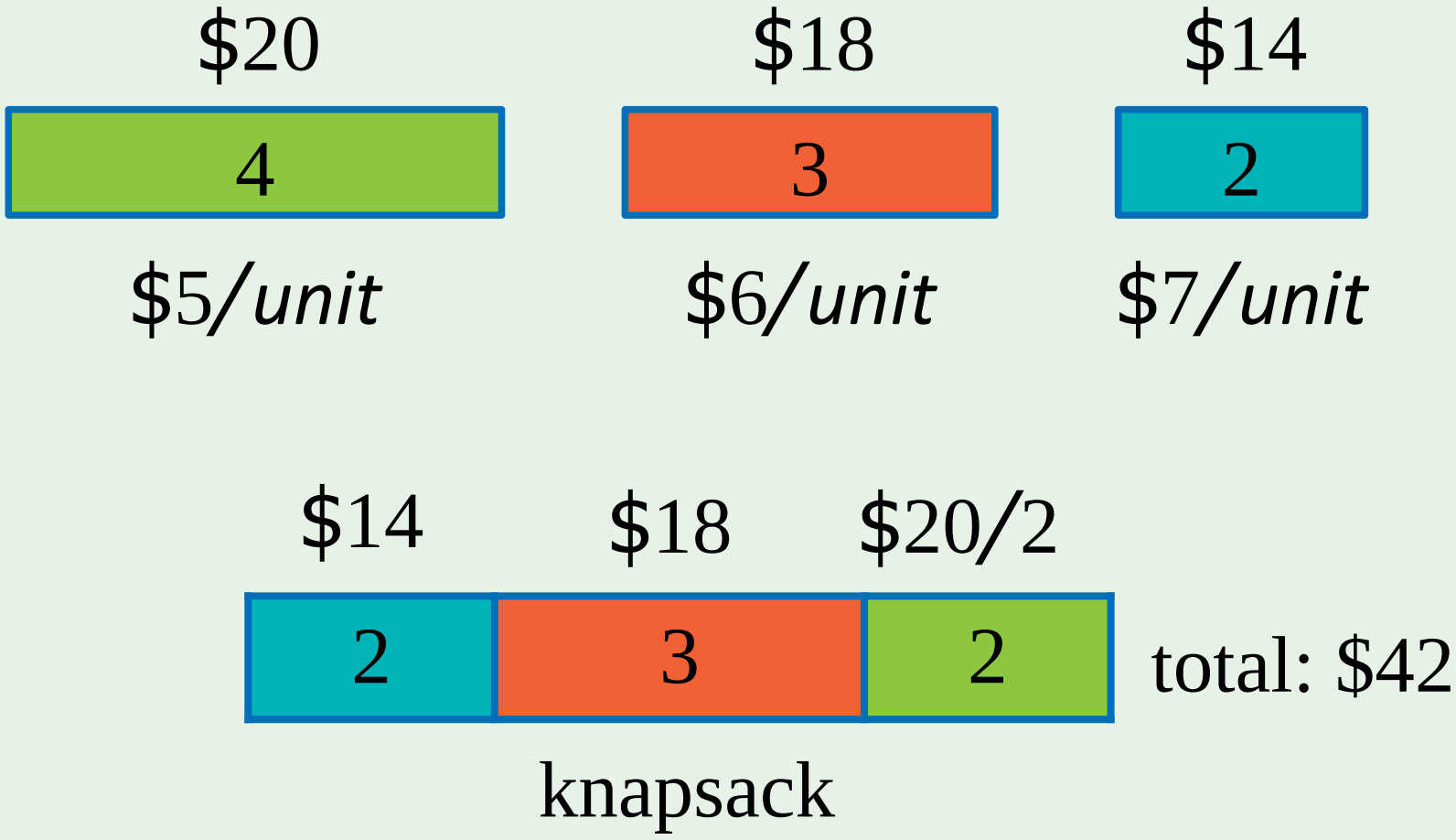



**total: $42**

# Optimal solution

## Greedy choice: fill with ***max price per unit***



# Greedy choice for Fractional Knapsack

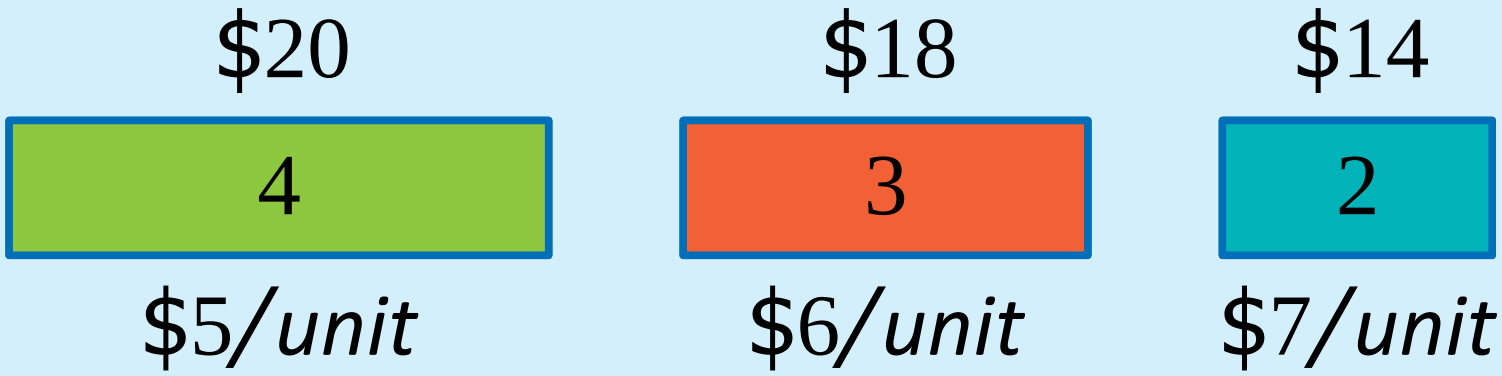
## Theorem

There exists an **optimal solution** that uses as much as possible of an item with the **maximum value per unit of weight**

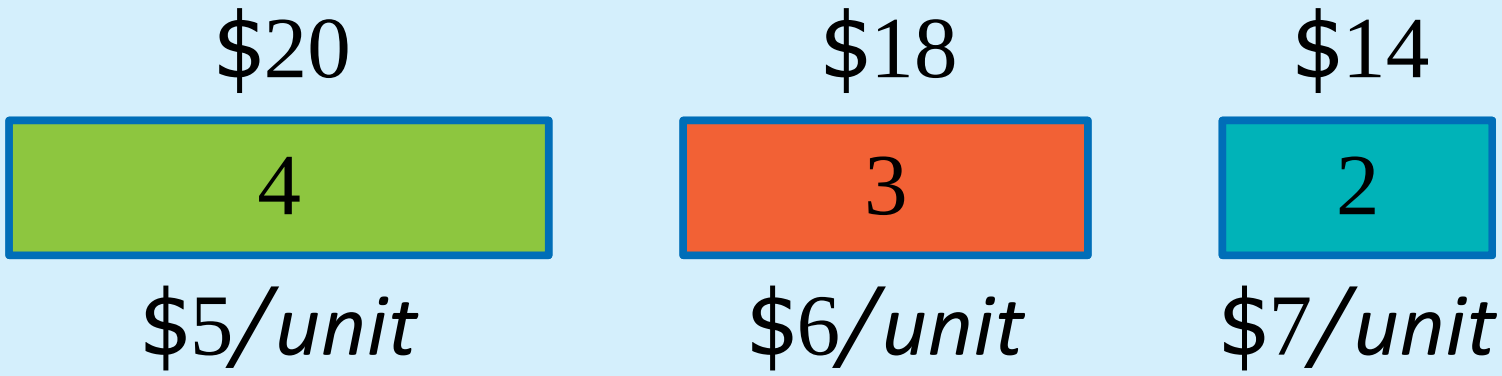
Note the following:

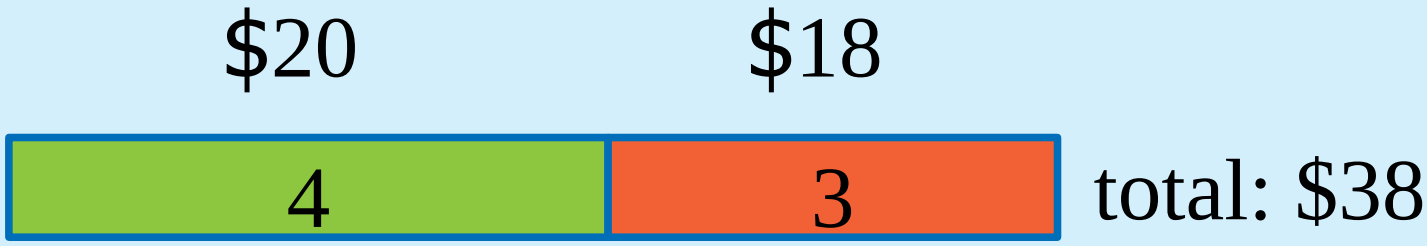
- We do not claim that every optimal solution must contain item A with this property
- There might be another optimal solution, but this other optimal solution will not be better than the one containing A

# Proof



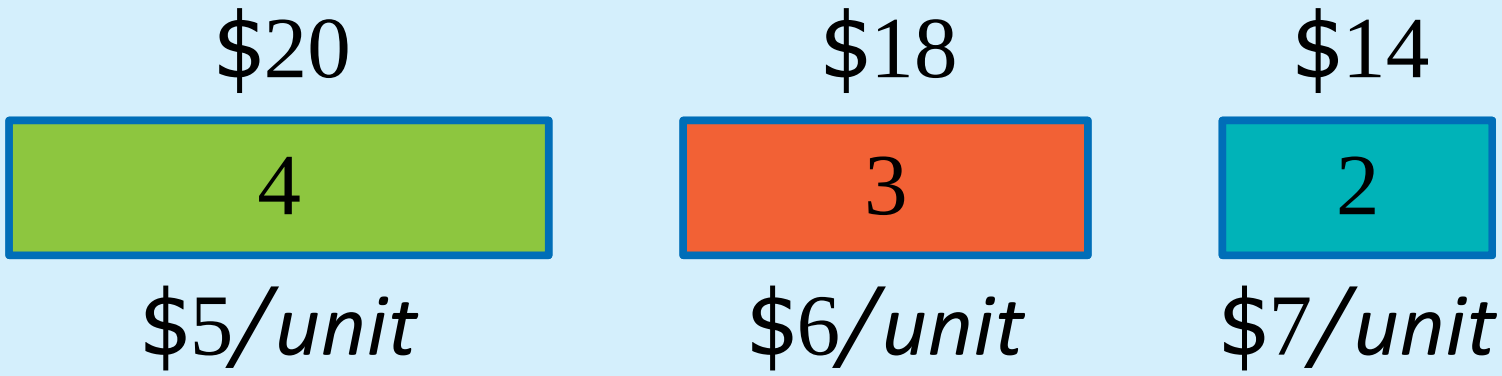
# Proof

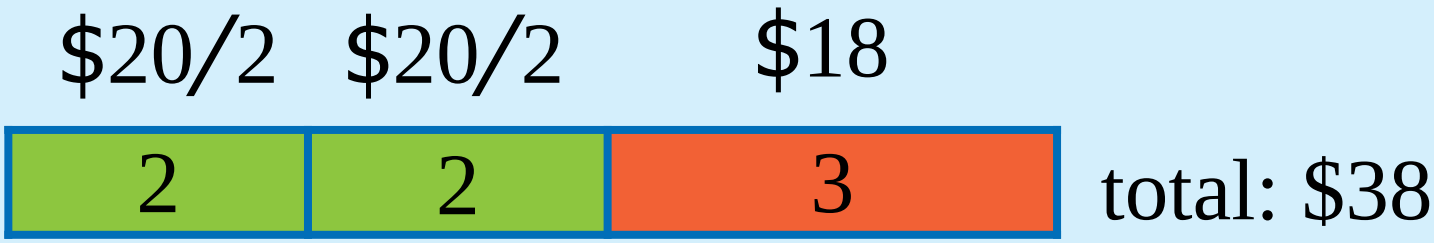



The proof is by **contradiction**:
Let's assume that there exists an **optimal solution** which **does not include** the item with the highest value per unit (blue item).

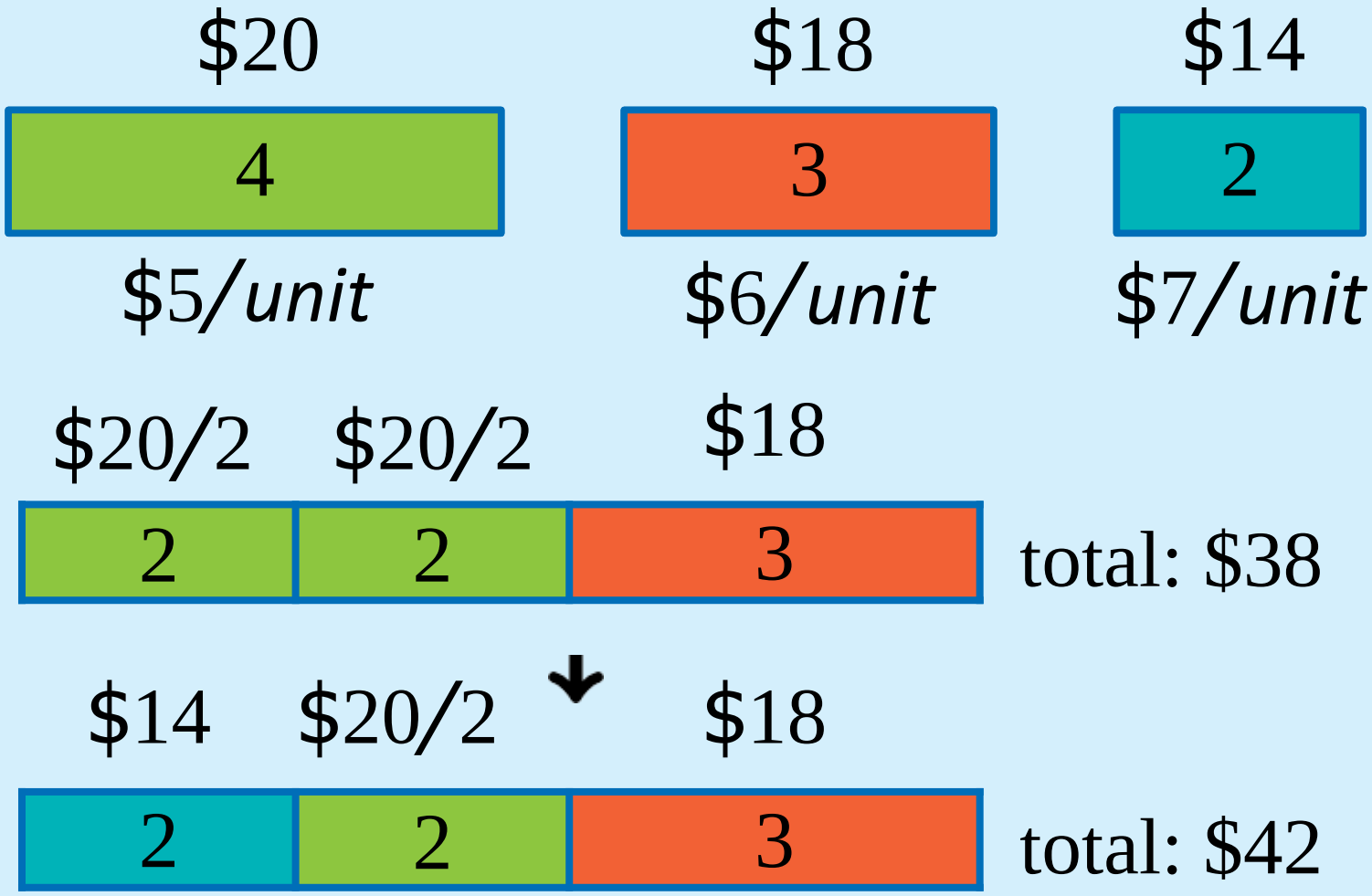
# Proof




However, because we can take fractions, we can split the green item into fractions…

# Proof




And we can improve the “optimal solution” with the blue item fractions – so that was not an optimal solution - contradiction

# Exchange argument

- The proof uses an ***exchange argument*** to show that the greedy move will lead to an optimal solution
- The general structure of such an argument is a proof by contradiction, where we assume, for the sake of reaching a contradiction, that there is a better solution than one found by the greedy algorithm
- We then argue that there is an *exchange* that we could make among the components of this solution that would lead to a better solution

# What is the max loot?

Input

| | |
|---|---|
| W: | 50 |
| C1: | w=60<br>v=200 |
| C2: | w=100<br>v=500 |
| C3: | w=120<br>v=300 |

Output

?

Input

| | |
|---|---|
| W: | 10 |
| C1: | w=2<br>v=20 |
| C2: | w=500<br>v=3000 |

Output

?

# Fractional Knapsack Algorithm in English

While knapsack is not full

Choose item *i* with maximum $\frac{v_i}{w_i}$

If item fits into knapsack, take all of it

Otherwise take so much as to fill the knapsack

Return total value and amounts taken

# Algorithm frac_knapsack ($W$ , $w_1$, $v_1$, . . . , $w_n$, $v_n$)

$A \leftarrow [0, 0, \ldots, 0]$ #amount for each item

$V \leftarrow 0$ #total value in knapsack

**repeat** $n$ **times:**

  **if** $W = 0$:

    **return** $(V , A)$

  **select** $i$ **with** $w_i > 0$ **and max** $\frac{v_i}{w_i}$

  $a \leftarrow \min(w_i, W)$

  $V \leftarrow V + a \frac{v_i}{w_i}$

  $w_i \leftarrow w_i - a$

  $A[i] \leftarrow A[i] + a$

  $W \leftarrow W - a$

**return** $(V , A)$

## Lemma

The running time of ***Knapsack*** is $O(n^2)$.

## Proof

- Select best item on each step is $O(n)$
- Main loop is executed $n$ times
- Overall, $O(n^2)$

■

Of course, we can do better

If we know: $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \cdots \geq \frac{v_n}{w_n}$

## Frac_knapsack$(W, w_1, v_1, \ldots, w_n, v_n)$

**$A \leftarrow [0, 0, \ldots, 0]$**
**$V \leftarrow 0$**

**for *i* from 1 to *n*:**
    **if $W = 0$:**
        **return $(V, A)$**
    **$a \leftarrow \min(w_i, W)$**  # take next item in order
    **$V \leftarrow V + a\frac{v_i}{w_i}$**
    **$w_i \leftarrow w_i - a$**
    **$A[i] \leftarrow A[i] + a$**
    **$W \leftarrow W - a$**
**return $(V, A)$**

# Improved Knapsack

- Now each iteration is $O(1)$
- Knapsack after sorting is $O(n)$
- Sort + Knapsack is $O(n \log n)$

Assuming that we know how to sort in O(n log n)

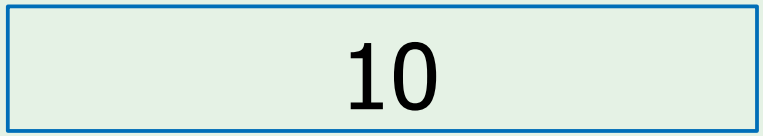
PROBLEM 2A

# Maximum loot (discrete items)

Discrete Knapsack, Knapsack 01

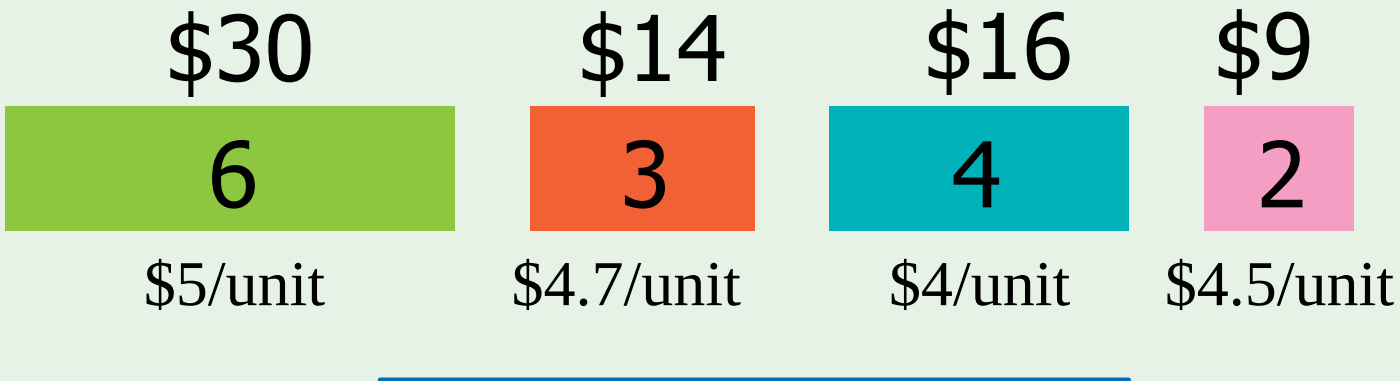
**No fractions allowed!**

# Example

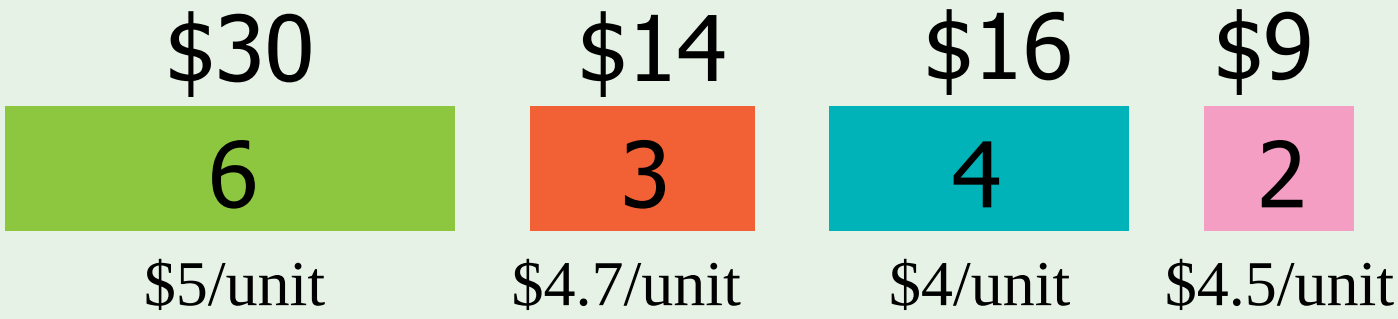




knapsack

# Example



# Example



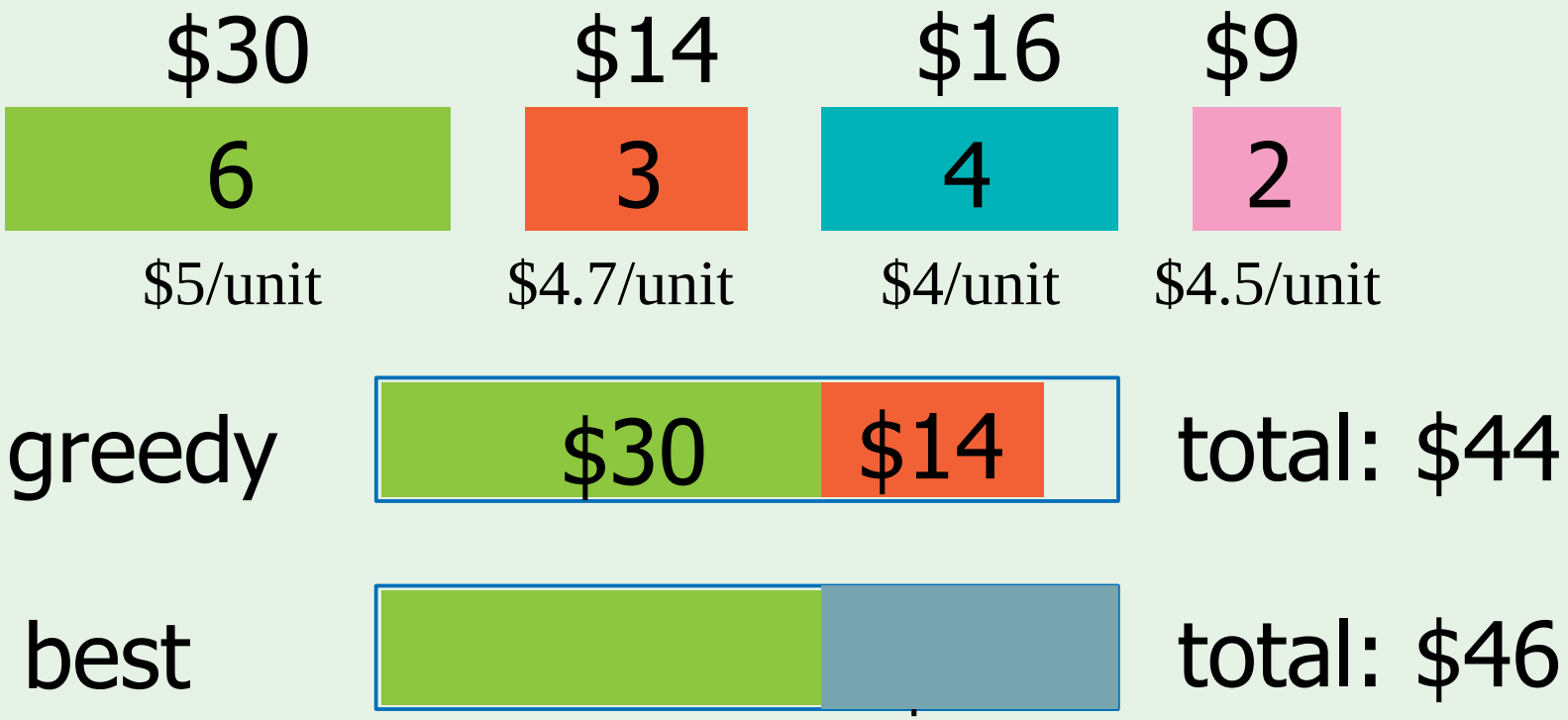


total: $44

10

# Why does greedy fail for the discrete knapsack?

## Example




taking an element of maximum value per unit of weight is not a safe move!

PROBLEM 3

# Car fueling

Greedy travels

# Car fueling problem

**Input:** A car which can travel at most $L$ kilometers with full tank,
a source point $A$, a destination point $B$ and $n$ gas stations at distances

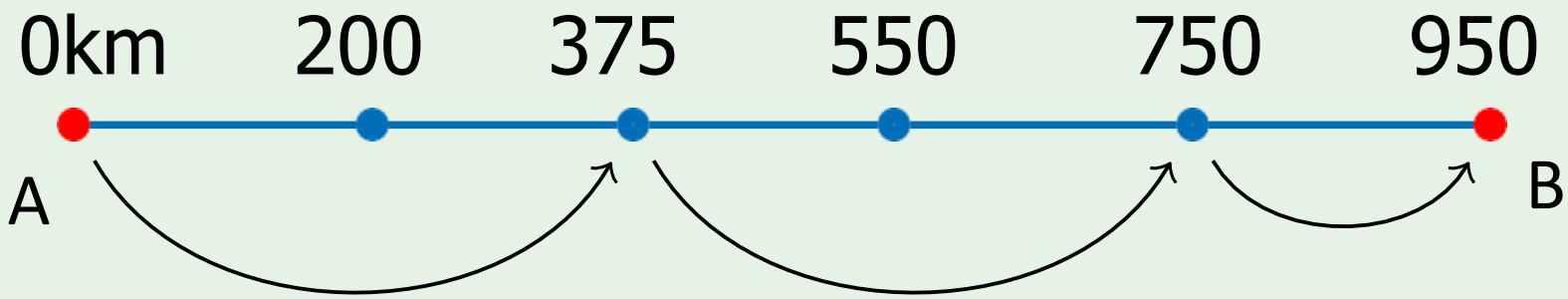
$$x_1 \le x_2 \le x_3 \le \cdots \le x_n$$

in kilometers from $A$ along the path from $A$ to $B$.

**Output:** The minimum number of refills to get from $A$ to $B$, besides refill at $A$.

# Car Fueling: problem instance

Distance with full tank = 400km




Minimum number of refills = 2

# General Greedy Strategy

- Make some greedy choice
- Reduce to a smaller problem (subproblem)
- Iterate

# Possible greedy choices:

- Go until there is no fuel
- Refill at the the *closest* gas station
- Refill at the *farthest reachable* gas station

# Greedy Algorithm

- Start at *A*
- Refill at the *farthest reachable* gas station *G*
- Make *G* the new *A*
- Get from new *A* to *B* with minimum number of refills

## Definition

*Subproblem* is a problem of the same type but of smaller size

## Example

Min number of refills from *A* to *B* = first refill at *G* + min number of refills from *G* to *B*

## Formal Definition

A greedy choice is called a *safe move* if there is an optimal solution consistent with this first move

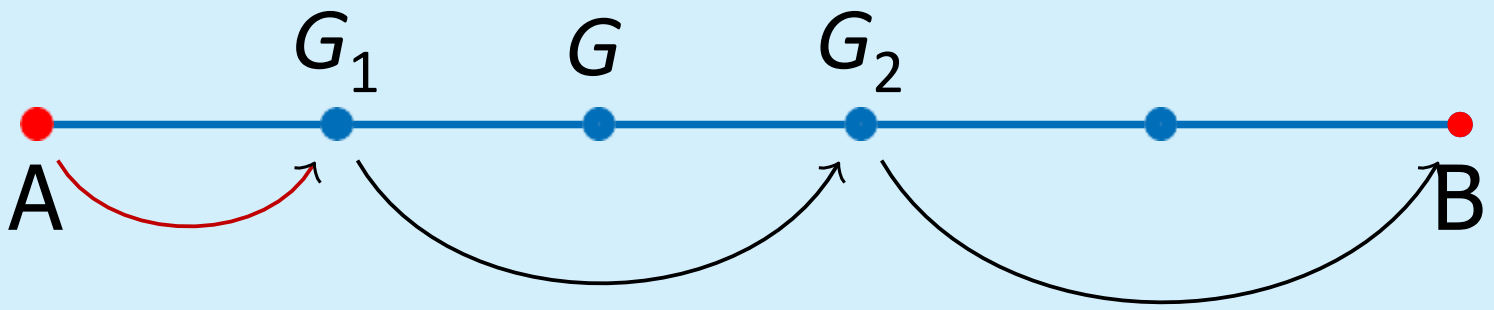
## Theorem

To refill at the farthest reachable gas station is a safe move.

# Proof: setup

- Route $R$ with the minimum number of refills
- $G_1$ - position of first refill in $R$
- $G_2$ - next stop in $R$ (refill or $B$)
- $G$ - farthest refill reachable from $A$

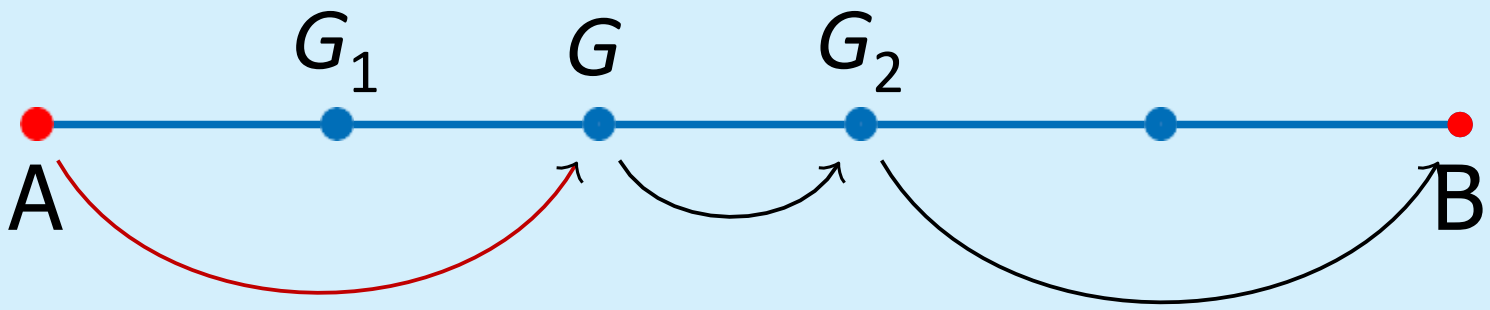
# Could we have done better?




First case: $G$ is closer than $G_2$

Refill at $G$ instead of $G_1$

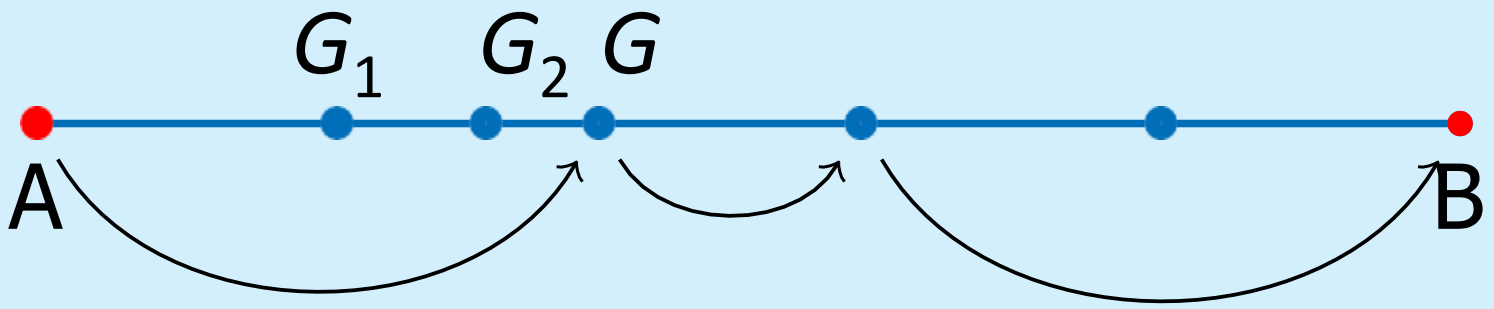
# Could we have done better?




First case: $G$ is closer than $G_2$

Refill at $G$ instead of $G_1$

# Could we have done better?




Second case: $G_2$ is closer than $G$

Avoid refill at $G_1$, $G_2$

# Proof: refill at G is a safe move

- Route $R$ with the minimum number of refills
- $G_1$ - position of first refill in $R$
- $G_2$ - next stop in $R$ (refill or $B$)
- $G$ - farthest refill reachable from $A$

Only 2 cases to consider:

1. If $G$ is closer than $G_2$, refill at $G$ instead of $G_1$ (same total number of refills)
2. Otherwise, avoid refill at $G_1$, $G_2$ (less refills) ■

$A = x_0 \le x_1 \le x_2 \le \cdots \le x_n \le x_{n+1} = B$

**Algorithm min_refills(*A*, *n*, *L*)**

```
num_refills ← 0, current_refill ← 0
while current_refill ≤ n:
    last_refill ← current_refill
    while A [current_refill + 1] − A [last_refill ] ≤ L:
        current_refill ← current_refill + 1
    if current_refill = last_refill :
        return IMPOSSIBLE
    if current_refill  ≤ n:
        num_refills ← num_refills + 1
return num_refills
```

## Lemma

The running time of `min_refills`($A$, $n$, $L$) is $O(n)$.

## Proof

- During the entire algorithm *current_refill* changes from 0 to $n + 1$, one-by-one
- During some of these changes *num_refills* also increases by 1
- Thus, $O(n)$ operations in total

Dramatization :) by Emma Waters

PROBLEM 3A

# More traveling

Traveling Salesman

# Sample problem: Soldering

In manufacturing it is often necessary to solder components onto electronic circuit boards.

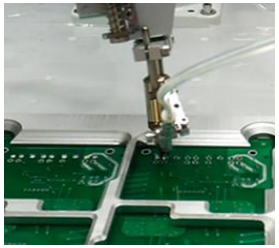
In this context we want to minimize the amount of time it takes a robotic arm to do the job.

## Soldering problem

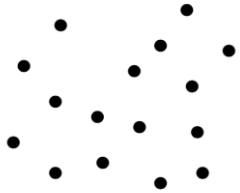
**Input**: A set of *n* points in the plane.

**Output**: A shortest route that travels to each point exactly once and returns to the initial point.

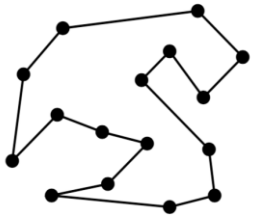
Circuit board



Robotic arm



Set of points



A candidate solution

# Greedy idea: Nearest Neighbor Tour

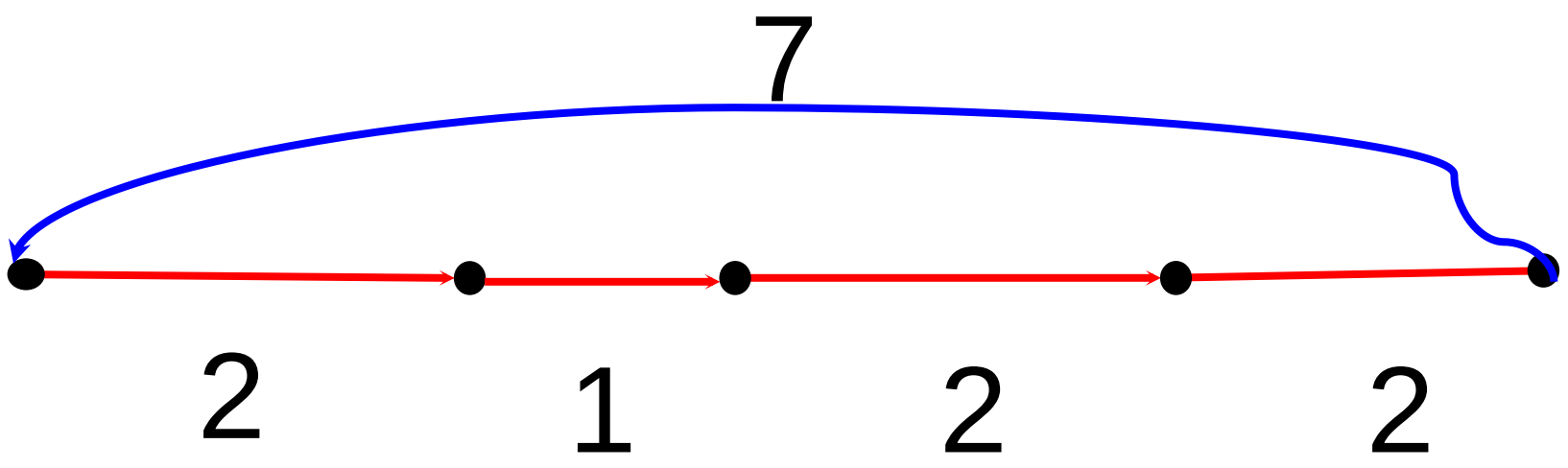
A greedy solution starts at some point $p_0$ and then walks to its nearest neighbor $p_1$ first, then repeats from $p_1$, etc. until done.

**Algorithm NearestNeighbor (set of *n* points in 2D)**

Pick and visit initial point $p_0$
$p = p_0$
$i = 0$
while there are still unvisited points
    $i = i + 1$
    let $p_i$ be the closest unvisited point to $p_{i-1}$
    visit $p_i$

return to $p_0$ from $p_i$

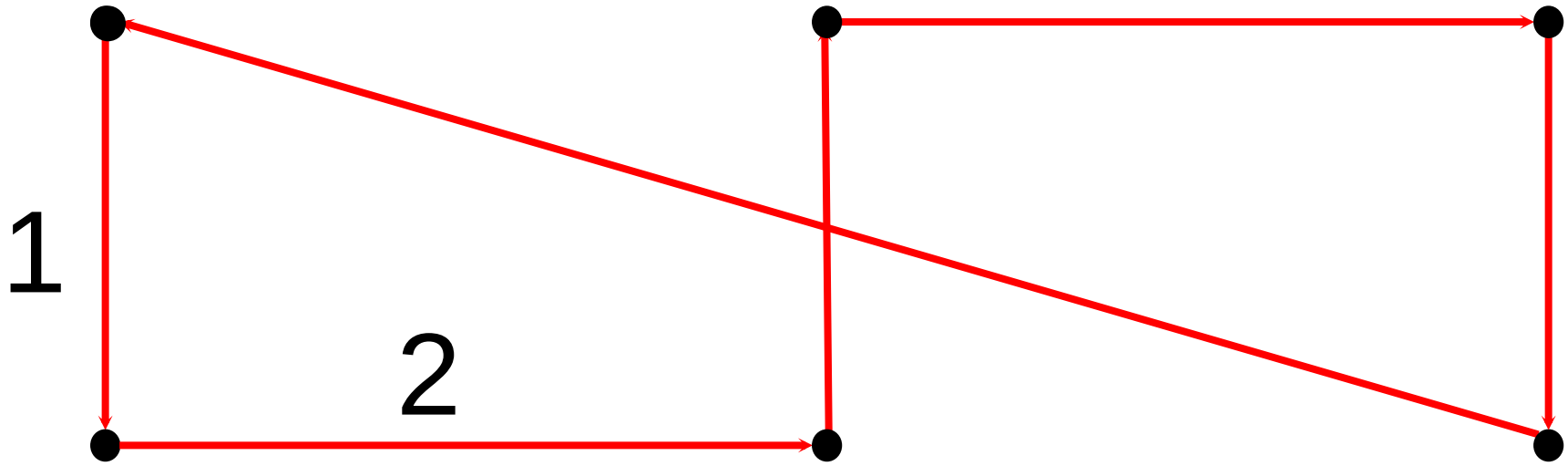
# It works!




Nearest neighbors gives 2 + 1 +2 + 2 + 7 = 14

Correct algorithm should produce desired result for **any** instance of the problem!

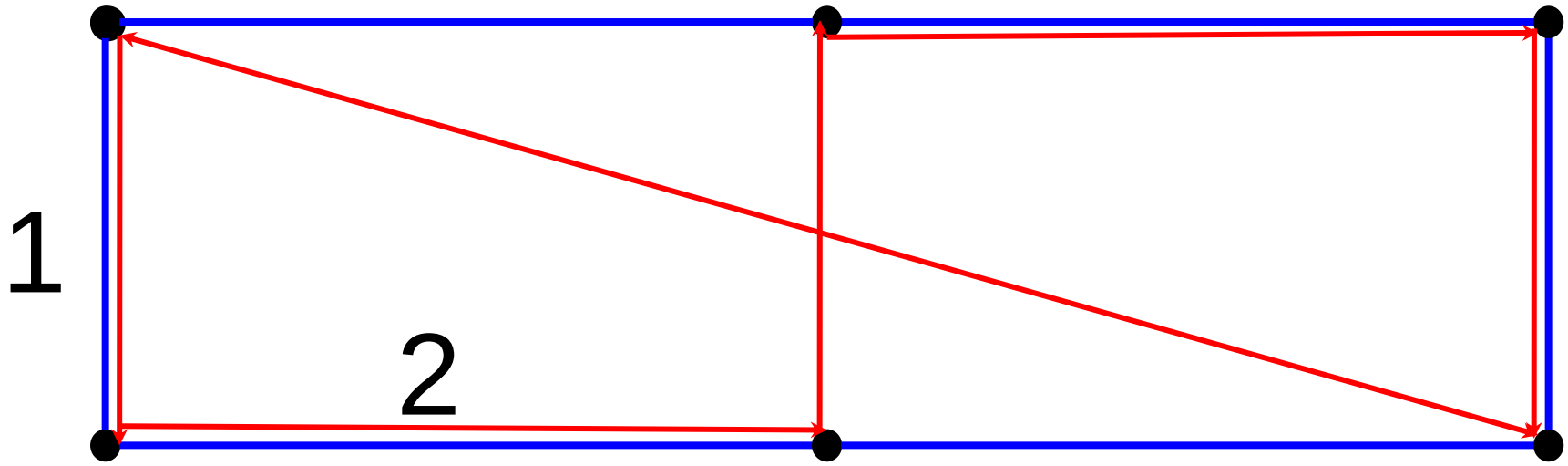
# It works...




Correct algorithm should produce desired result for **any** instance of the problem!

It does not work.

Enough to show a counterexample




Problem instance

The optimal solution has length 1+2+2+1+2+2 = 10.

Nearest neighbors gives $1+2+1+2+1+\sqrt{17} = 11.123$.

# Correct algorithm: exhaustive search

We could try all possible orderings of the points, then select the one which minimizes the total path length:

## Algorithm OptimalPath (set of n points in 2D)

$d := \infty$

$P_{min} :=$ Null

For each of the $n!$ paths $\Pi_i$ through $n$ points

    If $(cost(\Pi_i) \leq d)$ then

        $d = cost(\Pi_i)$ and $P_{min} = \Pi_i$

Return $P_{min}$

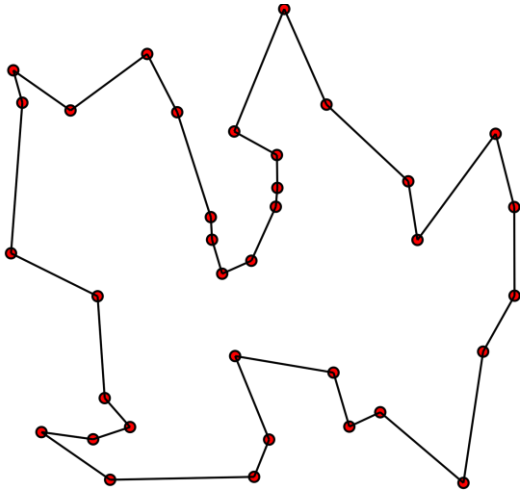
Since all possible orderings are considered,
we are guaranteed to end up with the shortest possible tour.

# Exhaustive Search is correct but too slow!

Because it tries all *n*! permutations, it is much too slow to use when there are more than 10-20 points.

**What grows faster: $2^n$, n! or $n^n$ ?**

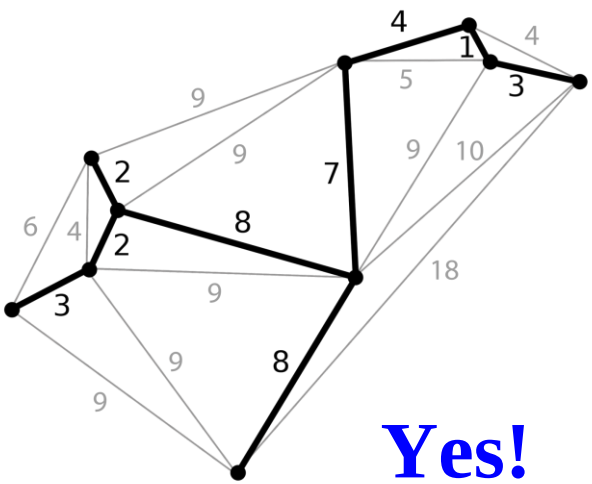
*Traveling Salesman Path:*
The shortest possible route that visits each vertex and returns to the origin vertex



# Exhaustive Search is correct but too slow!

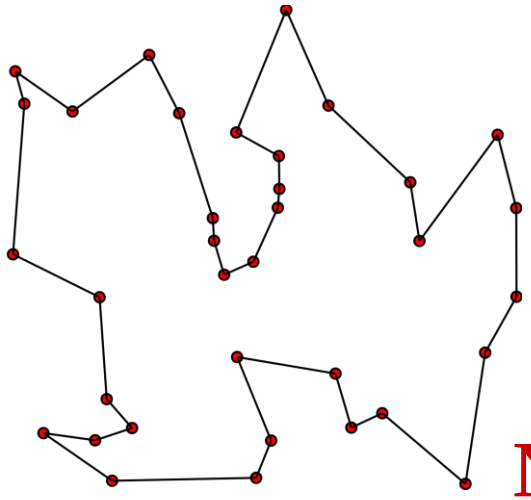
**No efficient, correct algorithm exists for the *traveling salesman problem*!**

*Minimum spanning tree*:
Subgraph connecting each pair of vertices with min overall edge cost




**Yes!**

*Traveling Salesman Path:*
The shortest possible route that visits each vertex and returns to the origin vertex



**No!**

# TSP applications

- Drilling printed circuit boards
- Transportation and logistics (school buses, meals on wheels, airplane schedules, etc.)
- Analyzing crystal structure
- Clustering data