W12: Reductions and Intractability

CS3230 AY21/22 Sem 2

Table of Contents

Click on the link to jump to the relevant sections!

Reductions

Discrete Math: Contrapositive

Recall that:

- *If p, then q*
- Contrapositive: *If not q, then not p*

Discrete Math: Contrapositive

Recall that:

- *If p, then q*
- Contrapositive: *If not q, then not p*

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Discrete Math: Contrapositive

Recall that:

- *If p, then q*
- Contrapositive: *If not q, then not p*

p	q	$p \rightarrow q$
T	*T*	*T*
T	*F*	*F*
F	*T*	*T*
F	*F*	*T*

$\sim q$	$\sim p$	$\sim q \rightarrow \sim p$
F	*F*	*T*
T	*F*	*F*
F	*T*	*T*
T	*T*	*T*

Discrete Math: Decimals to Binary

$O(log(x))$ bits to represent!

An integer $x >= 1$, needs $n = \lfloor log_2(x) \rfloor + 1$ bits to represent it

x	Binary Repr.	n
1	1	1
2	10	2
3	11	2
4	100	3
5	101	3
10	1010	4
23	10111	5
63	111111	6
64	1000000	7

Solving problems

- Given a certain Problem A, how do you go about solving it?

Solving problems

- Given a certain Problem A, how do you go about solving it?

- Come up with an algorithm for it normally

Solving problems

- Given a certain Problem A, how do you go about solving it?

- Come up with an algorithm for it normally, or
- Remember that you know how to solve Problem B, so you use that to help you

Solving problems

- Given a certain Problem A, how do you go about solving it?

- Come up with an algorithm for it normally, or
- Remember that you know how to solve Problem B, so you use that to help you

Reductions: set in a context where we think of the relationship between 2 problems!

Reductions: Algorithm Design perspective

- To solve problem A, you think of Problem B that **you already know how to solve**

Reductions: Algorithm Design perspective

- To solve problem A, you think of Problem B that **you already know how to solve**
- Reduction: **transform** type of input for A to type of input for B

Reductions: Algorithm Design perspective

- To solve problem A, you think of Problem B that **you already know how to solve**
- Reduction: **transform** type of input for A to type of input for B

- **Important:** the problem B is **black-boxed**. You must **not** modify it!
    - What is legal is for you to prepare an input for B, but you must not modify the algorithm

Reductions: Algorithm Design perspective

Pseudocode:

```
def solve_A(input_A):
```

Reductions: Algorithm Design perspective

Pseudocode:

Reduction: **transform** type of input for A to type of input for B

```
def solve_A(input_A):
1.  input_B = reduction_from_A_to_B(input_A)
```

Reductions: Algorithm Design perspective

Pseudocode:

```
def solve_A(input_A):
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B = solve_B(input_B) # magically given
```

Reduction: **transform** type of input for A to type of input for B

Remember: you cannot modify the solution to B!

Reductions: Algorithm Design perspective

Pseudocode:

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A)
    output_B = solve_B(input_B) # magically given

    output_A = transform_output_B_to_A(output_B)
    return output_A
```

Reduction: **transform** type of input for A to type of input for B

Remember: you cannot modify the solution to B!

Post-processing so that we get what we actually want

Reductions: Algorithm Design perspective

Pseudocode:

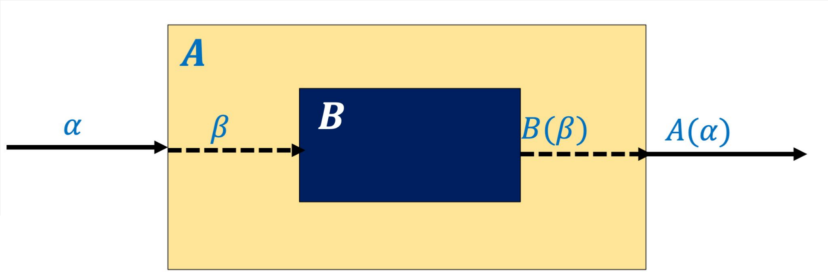
```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A)
    output_B = solve_B(input_B) # magically given

    output_A = transform_output_B_to_A(output_B)
    return output_A
```

Reduction: **transform** type of input for A to type of input for B

Remember: you cannot modify the solution to B!

Post-processing so that we get what we actually want



Toy Example: Min in array

Problem A: Finding minimum of element in an array

Problem B (know how to solve): Finding maximum of element in an array

Toy Example: Min in array

Problem A: Finding minimum of element in an array

Problem B (know how to solve): Finding maximum of element in an array

Reduction from A to B: negate all the elements in the array

(Intuition: the larger values in original array becomes “more negative” in new array, so the old min is the new max)

Toy Example: Min in array

```
def min_in_arr(input_A):
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B = max_in_arr(input_B) # magically given
3.   output_A = transform_output_B_to_A(output_B)
4.   return output_A

def reduction_from_A_to_B(input_A):
1.   return [-1 * num for num in input_A]

def transform_output_B_to_A(output_B):
1.   return -1 * output_B
```

Toy Example: Min in array

```
def min_in_arr(input_A):
    input_B = reduction_from_A_to_B(input_A)
    output_B = max_in_arr(input_B) # magically given
    output_A = transform_output_B_to_A(output_B)
    return output_A

def reduction_from_A_to_B(input_A):
    return [-1 * num for num in input_A]

def transform_output_B_to_A(output_B):
    return -1 * output_B
```

```
input_A = [-3, -1, 0, 2, 5]
```

Toy Example: Min in array

```
def min_in_arr(input_A):
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B = max_in_arr(input_B) # magically given
3.   output_A = transform_output_B_to_A(output_B)
4.   return output_A
                                                    input_A = [-3, -1, 0, 2, 5]
                                                    input_B = [3, 1, 0, -2, -5]

def reduction_from_A_to_B(input_A):
1.   return [-1 * num for num in input_A]

def transform_output_B_to_A(output_B):
1.   return -1 * output_B
```

Toy Example: Min in array

```
def min_in_arr(input_A):
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B = max_in_arr(input_B) # magically given
3.   output_A = transform_output_B_to_A(output_B)
4.   return output_A

def reduction_from_A_to_B(input_A):
1.   return [-1 * num for num in input_A]

def transform_output_B_to_A(output_B):
1.   return -1 * output_B
```

```
input_A = [-3, -1, 0, 2, 5]
input_B = [3, 1, 0, -2, -5]
output_B = 3
```

Toy Example: Min in array

```
def min_in_arr(input_A):
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B = max_in_arr(input_B) # magically given
3.   output_A = transform_output_B_to_A(output_B)
4.   return output_A

def reduction_from_A_to_B(input_A):
1.   return [-1 * num for num in input_A]

def transform_output_B_to_A(output_B):
1.   return -1 * output_B
```

```
input_A = [-3, -1, 0, 2, 5]
input_B = [3, 1, 0, -2, -5]
output_B = 3
output_A = -3
```

Reductions: Algorithm Analysis perspective

- For the purpose of the NP-Completeness chapters, we consider reductions as tools for us to compare “hardness” between problems

Reductions: Algorithm Analysis perspective

- For the purpose of the NP-Completeness chapters, we consider reductions as tools for us to compare "hardness" between problems

- "Hardness" usually refers to a class of time for the problem
  - "Easily solvable" - we have a polynomial time algorithm - $O(n^c)$, *eg* n^2, n^{100}, n^{100000}
  - "Hard" - we only have exponential time algorithm

Reductions: Algorithm Analysis perspective

- For the purpose of the NP-Completeness chapters, we consider reductions as tools for us to compare "hardness" between problems

- "Hardness" usually refers to a class of time for the problem
  - "Easily solvable" - we have a polynomial time algorithm - $O(n^c)$, *eg* n^2, n^{100}, n^{100000}
  - "Hard" - we only have exponential time algorithm

- We will show that if Problem A has a polynomial-time $p(n)$ reduction to Problem B:
  - then B is "at least as hard" as A

A note on polynomial time

In the context of the NP-Completeness chapter:

- We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time

In the context of the NP-Completeness chapter:

- We usually discuss polynomial time **with respect to encoding of input**
- Examples on encoding of input:
  - If input is a number n, then the encoding is $O(logn)$ bits

A note on polynomial time

In the context of the NP-Completeness chapter:

- We usually discuss polynomial time **with respect to encoding of input**
- Examples on encoding of input:
  - If input is a number n, then the encoding is $O(\log n)$ bits
  - Array of n elements, with max value M: $O(n \log M)$ bits

A note on polynomial time

In the context of the NP-Completeness chapter:

- We usually discuss polynomial time **with respect to encoding of input**
- Examples on encoding of input:
  - If input is a number n, then the encoding is $O(\log n)$ bits
  - Array of n elements, with max value M: $O(n \log M)$ bits

- You **aren't** required to know the Turing Machine part, but you **should know** that 'polynomial time' is computed with respect to the encoding of the input.

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 1

```
def just_loop(n):
for i in range(n):
  print(i)
return True
```

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 1

```
def just_loop(n):
for i in range(n):
    print(i)
return True
```

- Let *ENC* denote input encoding
- ENC = $O(logn)$

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 1

```python
def just_loop(n):
    for i in range(n):
        print(i)
    return True
```

- Let *ENC* denote input encoding
- ENC = $O(\log n)$
- Time: $O(n) = O(2^{\log n})$

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 1

```
def just_loop(n):
1. for i in range(n):
2.     print(i)
3. return True
```

- Let *ENC* denote input encoding
- ENC = $O(\log n)$

- Time: $O(n) = O(2^{\log n}) = O(2^{ENC})$

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 1

```python
def just_loop(n):
    for i in range(n):
        print(i)
    return True
```

- Let *ENC* denote input encoding
- ENC = $O(\log n)$

- Time: $O(n) = O(2^{\log n}) = O(2^{ENC})$
- Algorithm is **exponential** time **with respect to length of input encoding**

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 2

```
# array size n, max value: n
def arr_loop(nums):
  n = len(nums)
  for i in range(n):
    print(nums[i])
  return True
```

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 2

```
# array size n, max value: n
def arr_loop(nums):
1. n = len(nums)
2. for i in range(n):
3.   print(nums[i])
4. return True
```

- Let *ENC* denote input encoding
- ENC = $O(n\log n)$

We usually discuss polynomial time **with respect to encoding of input**

A note on polynomial time: Example 2

```
# array size n, max value: n
def arr_loop(nums):
1. n = len(nums)
2. for i in range(n):
3.   print(nums[i])
4. return True
```

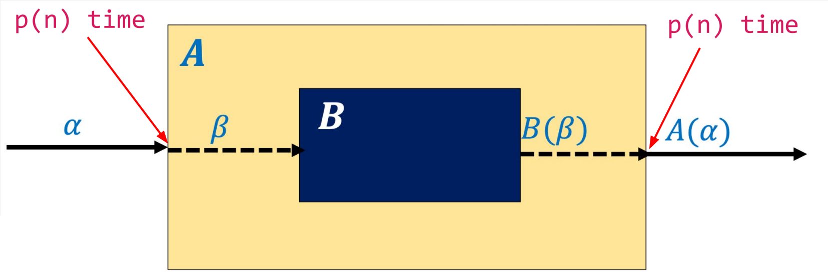
- Let *ENC* denote input encoding
- ENC = $O(n\log n)$

- Time: $O(n)$
- Algorithm is **polynomial** time **with respect to length of input encoding**

Reductions: Algorithm Analysis perspective

p(n)-time reduction

```
def solve_A(input_A):
1.   input_B = reduction_from_A_to_B(input_A) # p(n) time
2.   output_B = solve_B(input_B)
3.
4.   output_A = transform_output_B_to_A(output_B) # p(n) time
5.   return output_A
```



Reductions: Algorithm Analysis perspective

p(n)-time reduction. Assume `solve_B` takes T(n) time

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B)

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

The `input_B` must be of size at most p(n) (assuming that we take time to read from `input_A`)

Reductions: Algorithm Analysis perspective

p(n)-time reduction. Assume `solve_B` takes T(n) time

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

The `input_B` must be of size at most p(n) (assuming that we take time to read from `input_A`)

Reductions: Algorithm Analysis perspective

Time to solve A: $T(p(n)) + 2p(n)$

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Reductions: Algorithm Analysis perspective

Time to solve A: $\underbrace{T(p(n))}_{\text{Time to solve B:}} + 2p(n)$

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Reductions: Algorithm Analysis perspective

Time to solve A: $\underbrace{T(p(n))}$ $+ 2p(n)$

Time to solve B:
If polytime is possible...

Then the time to solve A is also polytime!

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A

Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is “easily solvable”, then so is A (a way to interpret previous statement)

Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A (a way to interpret previous statement)
- If A is "hard", then so is B (contrapositive)

Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A (a way to interpret previous statement)
- If A is "hard", then so is B (contrapositive)

- The last one is also the reason for the notation for reduction (the less than equal to is in terms of hardness) -- B is "at least as hard" as A

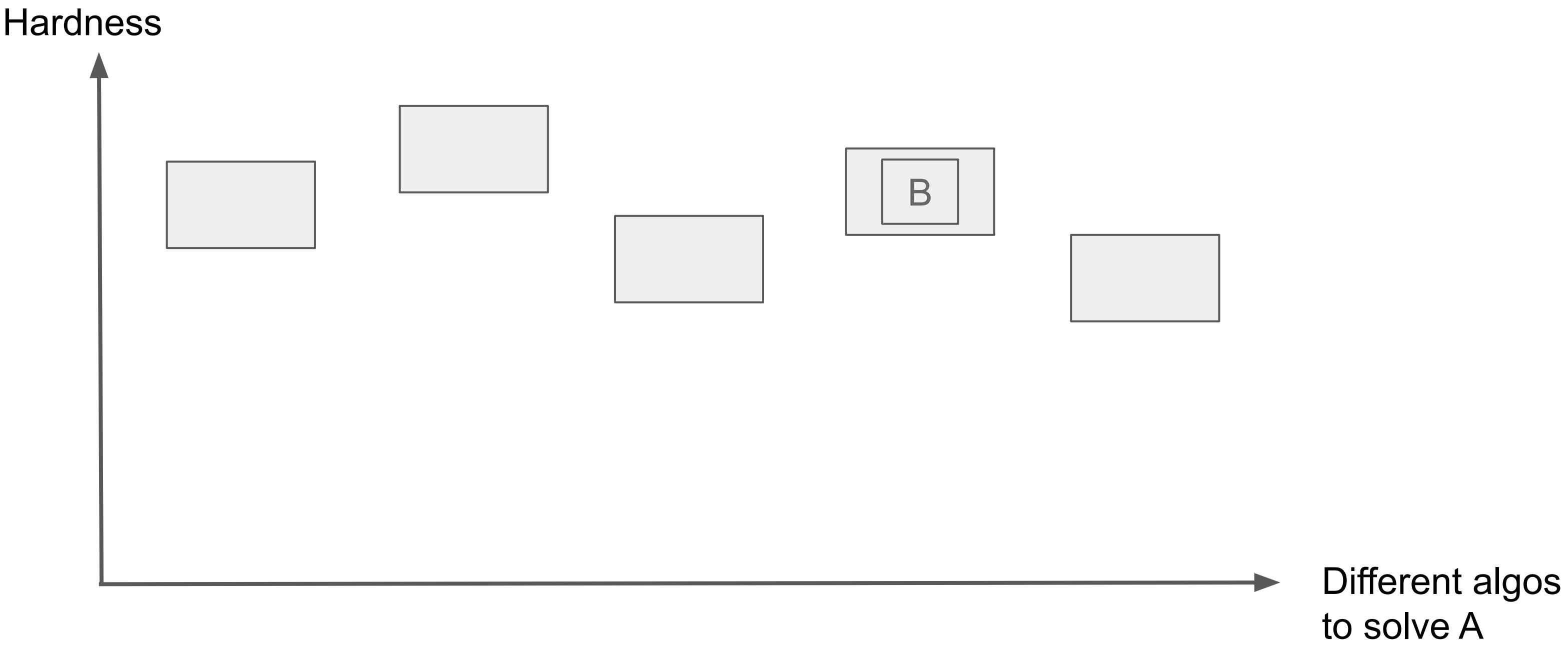
$A \leq_p B$

Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

We have a lot of possible ways to solve A! Using B might **not** be the best way

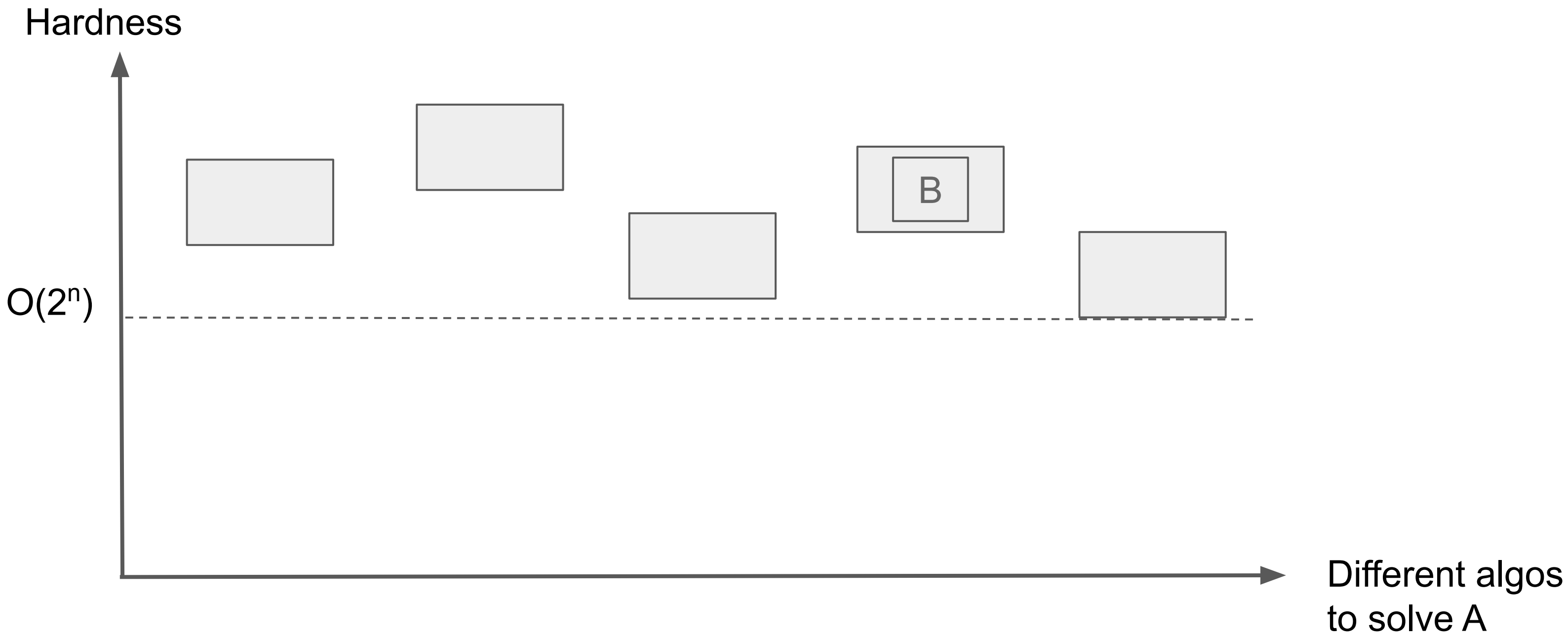


$A \leq_p B$

Intuition for comparing hardness

```
def solve_A(input_A):
1.  input_B = reduction_from_A_to_B(input_A) # p(n) time
2.  output_B = solve_B(input_B) # T(p(n)) time
3.
4.  output_A = transform_output_B_to_A(output_B) # p(n) time
5.  return output_A
```

Want to show: If A is “hard”, then B is “at least as hard” as A



$A \leq_p B$

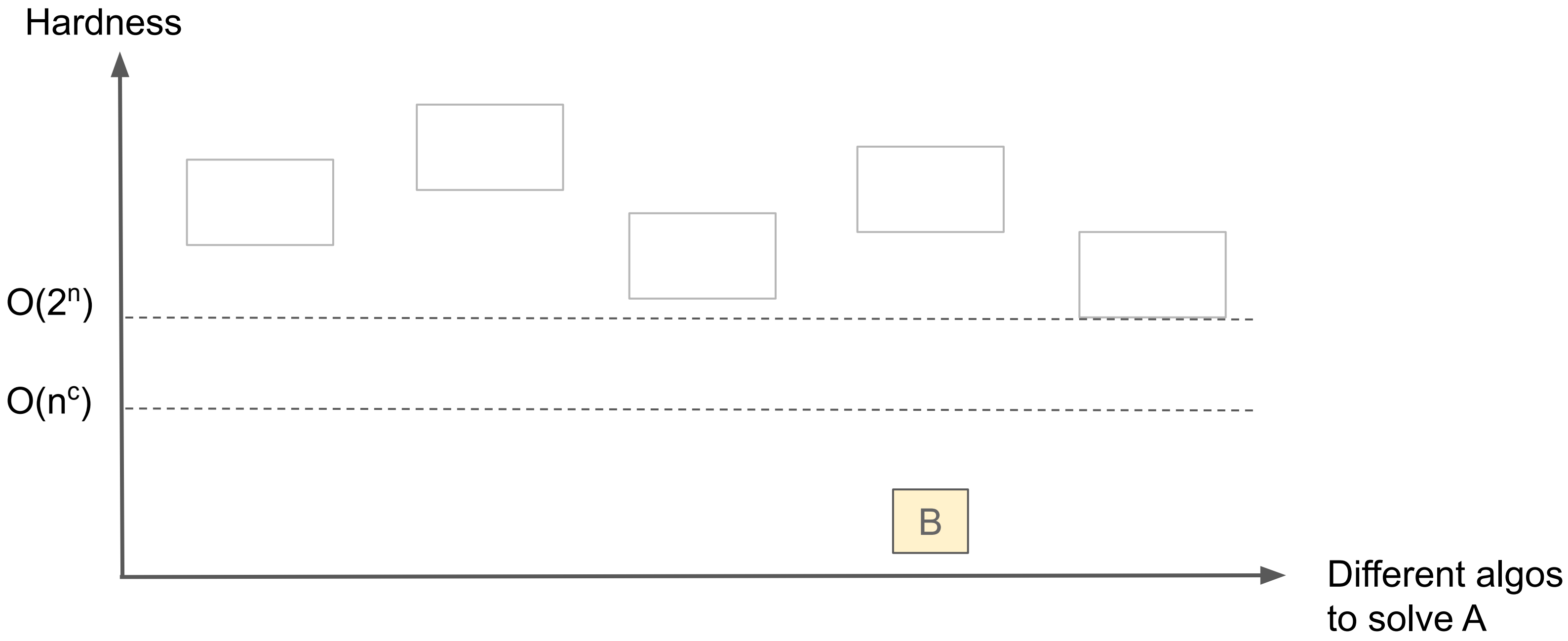
Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Want to show: If A is “hard”, then B is “at least as hard” as A

Suppose not: B is actually “easy”!



$A \leq_p B$

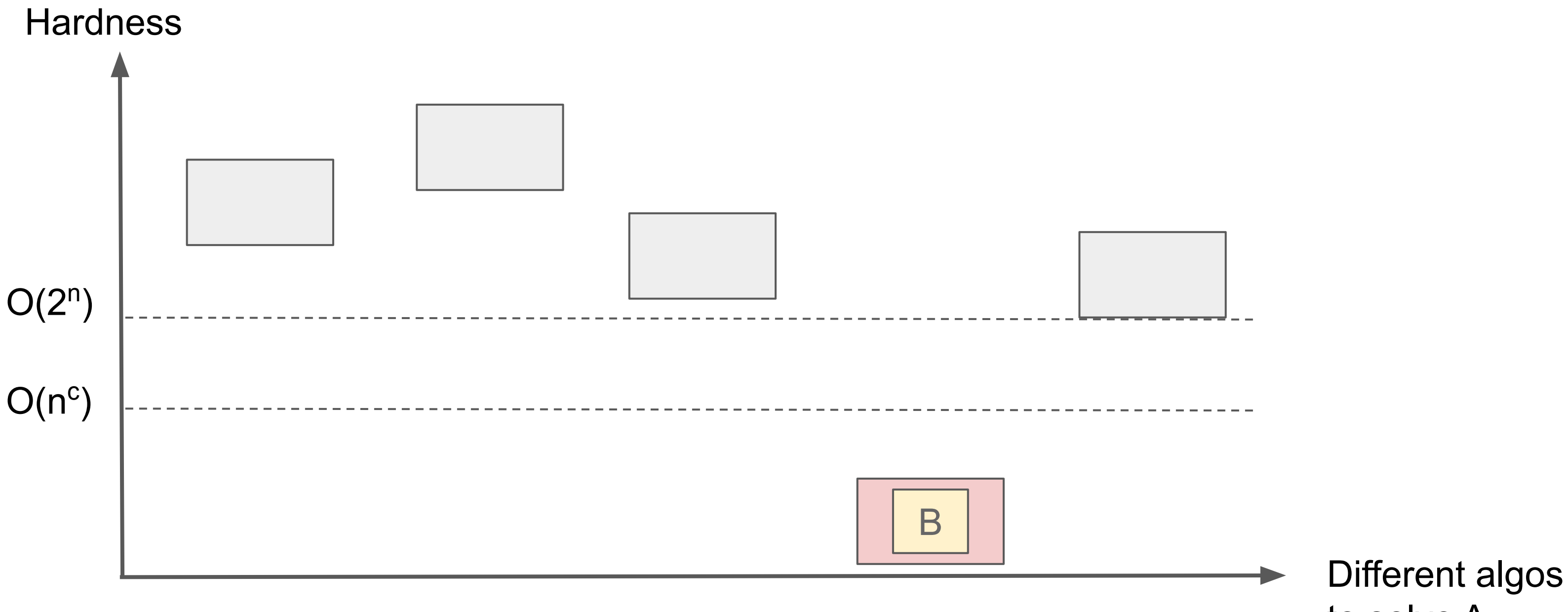
Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Want to show: If A is “hard”, then B is “at least as hard” as A

Suppose not: B is actually “easy” → The algo for A that uses B must be polytime (A is “easy” also)



Technicality: reduction must be in polytime

$A \leq_p B$

Intuition for comparing hardness

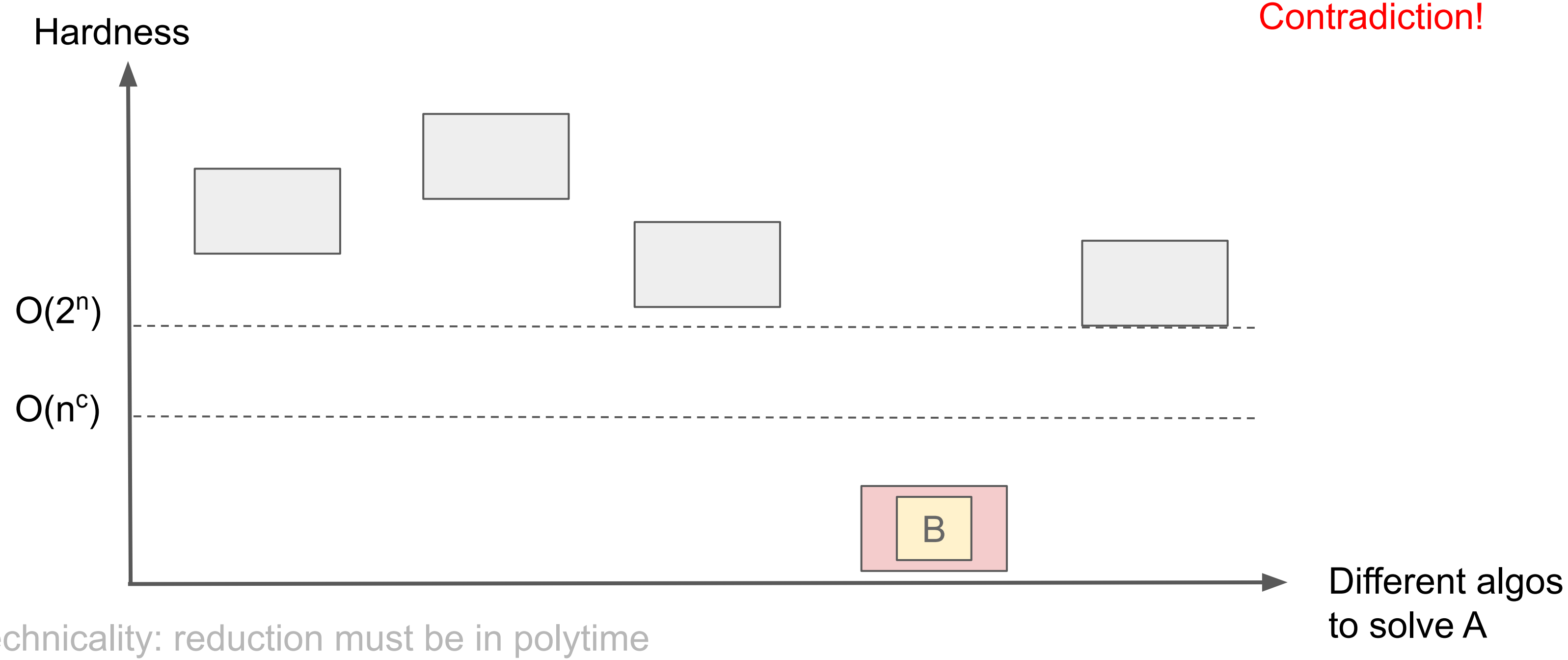
```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Want to show: If A is "hard", then B is "at least as hard" as A

Suppose not: B is actually "easy" → The algo for A that uses B must be polytime (A is "easy" also)

Contradiction!



Technicality: reduction must be in polytime

$A \leq_p B$

Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

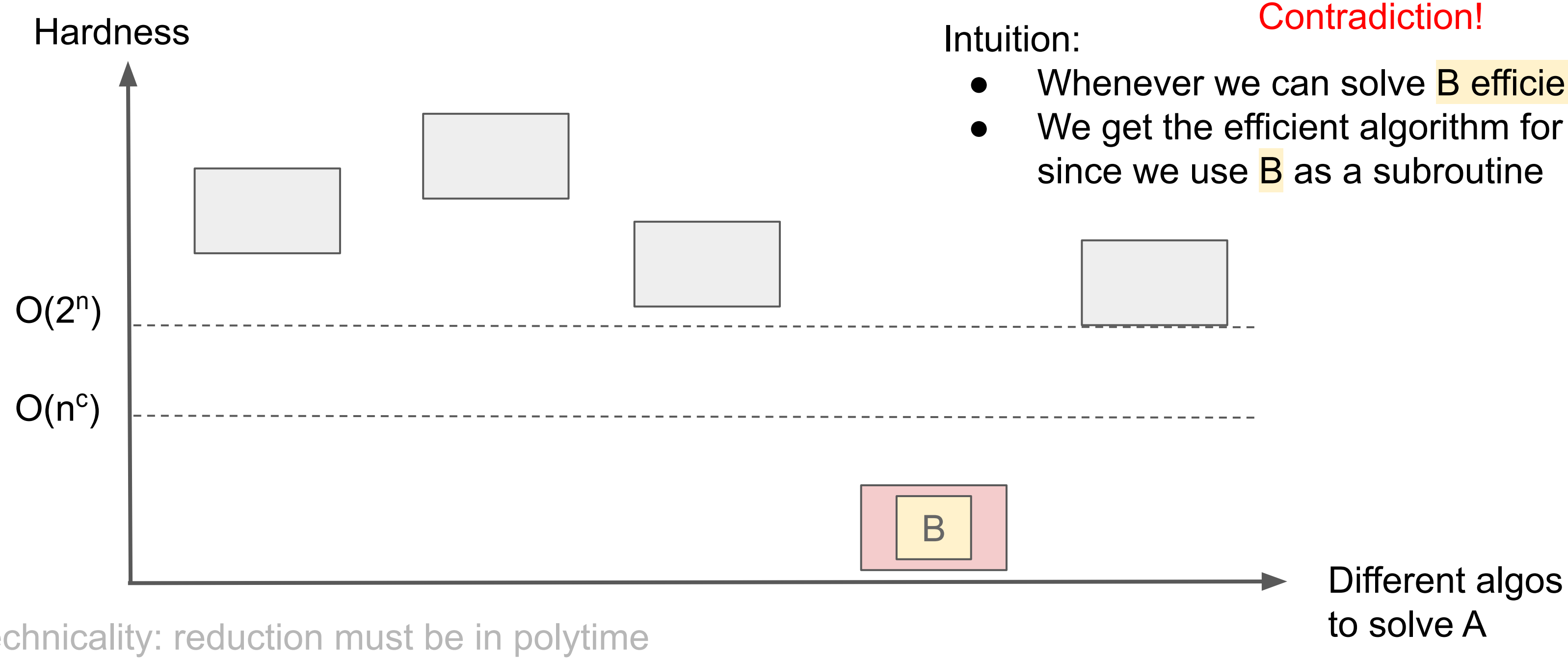
Want to show: If A is “hard”, then B is “at least as hard” as A

Suppose not: B is actually “easy” → The algo for A that uses B must be polytime (A is “easy” also)

Contradiction!

Intuition:

- Whenever we can solve B efficiently
- We get the efficient algorithm for A “for free” since we use B as a subroutine



Technicality: reduction must be in polytime

$A \leq_p B$

Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Note: If A is “easy”, it does **not** say anything about B.

$A \leq_p B$

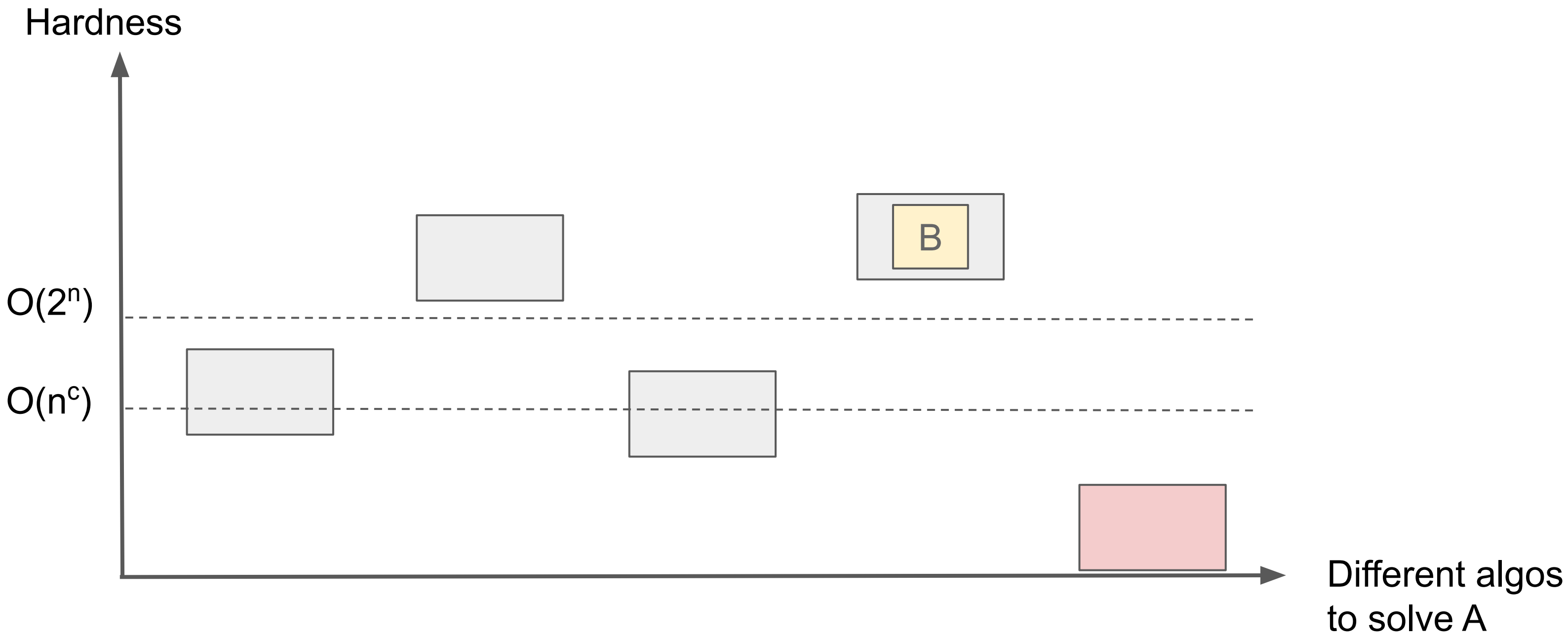
Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Note: If A is “easy”, it does **not** say anything about B.

A could have been “easy” because of another algorithm



Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is “easily solvable”, then so is A
- If A is “hard”, then so is B

IMPORTANT

Please get the direction of reduction right!

To show a problem is “hard”, you need to reduce **FROM** a “hard” problem

Doing it in the wrong direction means you are proving the wrong thing →
likely to get 0 marks for the question if in an exam

Decision Problems

Decision Problems

- Essence of decision problem:
  - Takes in an input
  - Output YES or NO (you can think of it as boolean values)

- YES-instance: Input to the problem that will output YES
- NO-instance: Input to the problem that will output NO

Reductions between Decision Problems

```
def solve_A(input_A) -> bool:
    input_B = reduction_from_A_to_B(input_A)
    output_B: bool = solve_B(input_B) # magically given
    return output_B
```

Reductions between Decision Problems

```
def solve_A(input_A) -> bool:
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B: bool = solve_B(input_B) # magically given
3.   return output_B
```

When is this algorithm correct?

Reductions between Decision Problems

```
def solve_A(input_A) -> bool:
    input_B = reduction_from_A_to_B(input_A)
    output_B: bool = solve_B(input_B) # magically given
    return output_B
```

When is this algorithm correct?

- `input_A` is YES-instance → `input_B` is also a YES-instance

Reductions between Decision Problems

```
def solve_A(input_A) -> bool:
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B: bool = solve_B(input_B) # magically given
3.   return output_B
```

When is this algorithm correct?

- input_A is YES-instance → input_B is also a YES-instance
- input_A is NO-instance → input_B is also a NO-instance

Reductions between Decision Problems

```
def solve_A(input_A) -> bool:
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B: bool = solve_B(input_B) # magically given
3.   return output_B
```

When is this algorithm correct?

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_A` is NO-instance → `input_B` is also a NO-instance

Basically: the reduction must do the “right thing” and cleverly transform. **You have to prove this**!

Reductions between Decision Problems

```
def solve_A(input_A) -> bool:
1.   input_B = reduction_from_A_to_B(input_A)
2.   output_B: bool = solve_B(input_B) # magically given
3.   return output_B
```

When is this algorithm correct?

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance

(Take the contrapositive, usually this makes proving easier)

Basically: the reduction must do the “right thing” and cleverly transform. **You have to prove this**!

Polynomial Time Reduction

$A \leq_p B$ is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

Polynomial Time Reduction

$A \leq_p B$ is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance

Polynomial Time Reduction

$A \leq_p B$ is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`

Polynomial Time Reduction

$A \leq_p B$ is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`

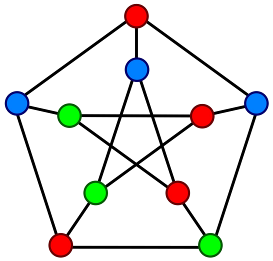
Coming up with the reduction will involve:

- Proposing a scheme on how to prepare `input_B`
- Proving these 3 properties above

Question 1:
Reducibility between Optimisation and Decision Problems

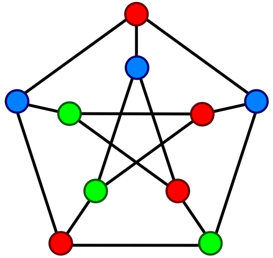
Question 1

Graph Colouring: Given a graph, colour a vertex with set of colours, such that the colours of v_1 and v_2 must be different if (v_1, v_2) is an edge



Question 1

Graph Colouring: Given a graph, colour a vertex with set of colours, such that the colours of v_1 and v_2 must be different if (v_1, v_2) is an edge



Optimisation Problem: Minimum number of colours to colour the graph

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Question 1

NUS
National University of Singapore

In the graph coloring problem, you are given an undirected graph G = (V,E) and asked to color the vertex with a set of colors such that the colors of v_1 and v_2 must be different if $(v_1,v_2) \in E$. In the optimization version of the problem, the task is to find the minimum number of colors required to color the graph. In the decision version, the question is whether there is a coloring of the graph using k or fewer colors. Select all true statements.

1. If we can solve the optimization problem for graph coloring in polynomial time, we would be able to solve the decision problem for graph coloring in polynomial time.
2. If we can solve the decision problem for graph coloring in polynomial time, we would be able to solve the optimization problem for graph coloring in polynomial time.
3. If the decision problem for graph coloring cannot be solved in polynomial time, the optimization problem for graph coloring cannot be solved in polynomial time.
4. If the optimization problem for graph coloring cannot be solved in polynomial time, the decision problem for graph coloring cannot be solved in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Question 1 Solution

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 1 - Solve optimisation → Solve decision:

1. If we can solve the optimization problem for graph coloring in polynomial time, we would be able to solve the decision problem for graph coloring in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Question 1 Solution

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 1 - Solve optimisation → Solve decision:

1. Use the optimisation solution to obtain x, the minimum colours
2. Check if $x \leq k$

1. If we can solve the optimization problem for graph coloring in polynomial time, we would be able to solve the decision problem for graph coloring in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Question 1 Solution

Option 1 - Solve optimisation → Solve decision:

1. Use the optimisation solution to obtain x, the minimum colours
2. Check if $x \leq k$

Reduction from what to what?

1. If we can solve the optimization problem for graph coloring in polynomial time, we would be able to solve the decision problem for graph coloring in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Question 1 Solution

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 1 - Solve optimisation → Solve decision:

1. Use the optimisation solution to obtain x, the minimum colours
2. Check if $x \leq k$

Reduction from what to what?

decision $\leq_p$ *optimisation*

```
def solve_decision(input_A, k) -> bool:
1.  input_B = input_A
2.  output_B = solve_optimisation(input_B) # magically given
3.  return output_B ≤ k
```

1. If we can solve the optimization problem for graph coloring in polynomial time, we would be able to solve the decision problem for graph coloring in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Question 1 Solution

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 2 - Solve decision → Solve optimisation:

2. If we can solve the decision problem for graph coloring in polynomial time, we would be able to solve the optimization problem for graph coloring in polynomial time.

Question 1 Solution

Optimisation Problem: Minimum number of colours to colour the graph

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 2 - Solve decision → Solve optimisation:

1. Linear search (or binary search) on the value k

2. If we can solve the decision problem for graph coloring in polynomial time, we would be able to solve the optimization problem for graph coloring in polynomial time.

Question 1 Solution

Optimisation Problem: Minimum number of colours to colour the graph

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 2 - Solve decision → Solve optimisation:

1. Linear search (or binary search) on the value k
2. Idea: Check if 1 is min, check if 2 is min, check if 3 is min, and so on until |V|..

2. If we can solve the decision problem for graph coloring in polynomial time, we would be able to solve the optimization problem for graph coloring in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Question 1 Solution

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 2 - Solve decision → Solve optimisation:

1. Linear search (or binary search) on the value k
2. Idea: Check if 1 is min, check if 2 is min, check if 3 is min, and so on until |V|..

You are going to check at most the number of nodes -- polynomial in size of input!

So it still takes polynomial time

2. If we can solve the decision problem for graph coloring in polynomial time, we would be able to solve the optimization problem for graph coloring in polynomial time.

Optimisation Problem: Minimum number of colours to colour the graph

Question 1 Solution

Decision Problem: Whether there is a colouring of the graph using k or fewer colours

Option 3:

- Contrapositive of Option 1

Option 4:

- Contrapositive of Option 2

1. If we can solve the optimization problem for graph coloring in polynomial time, we would be able to solve the decision problem for graph coloring in polynomial time.
2. If we can solve the decision problem for graph coloring in polynomial time, we would be able to solve the optimization problem for graph coloring in polynomial time.
3. If the decision problem for graph coloring cannot be solved in polynomial time, the optimization problem for graph coloring cannot be solved in polynomial time.
4. If the optimization problem for graph coloring cannot be solved in polynomial time, the decision problem for graph coloring cannot be solved in polynomial time.

Question 2:
PARTITION and BALL-PARTITION

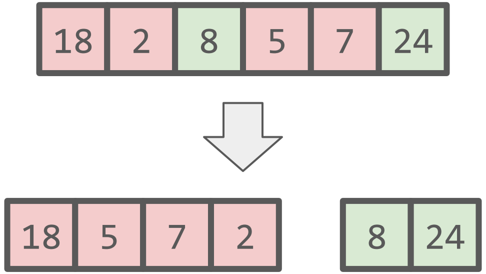
Question 2: PARTITION

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

18	2	8	5	7	24

Question 2: PARTITION

Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?



Question 2: BALL-PARTITION

Given k balls, can we divide the balls into two boxes with an equal number of balls?

Question 2: BALL-PARTITION

Given k balls, can we divide the balls into two boxes with an equal number of balls?

Clearly, YES if and only if k is even

Proposal for Reduction

1. From the problem in PARTITION, we have S, a set of positive integers
2. k = total sum of elements in S
3. Use this k for BALL-PARTITION

18	2	8	5	7	24

k = 18 + 2 + 8 + 5 + 7 + 24 = 64

```
def PARTITION(array) -> bool:
    k = reduction_from_A_to_B(array)
    output_B: bool = BALL_PARTITION(k)
    return output_B
```

Question 2

Consider the two problems:

PARTITION: Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?

BALL-PARTITION: Given k balls, can we divide the balls into two boxes with an equal number of balls?

We try to show that PARTITION $\leq_P$ BALL-PARTITION using the following transformation A:

1) From the problem PARTITION, we are given S, a set of positive integers.

2) We define k as the total sum of all the elements in S.

3) We use this number k for the BALL-PARTITION problem.

What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Additionally, IF the reduction is correct, what does it mean for *PARTITION $\leq_p$ BALL-PARTITION*?

What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Question 2 Solution

Option 2 is wrong with the transformation A

```
def PARTITION(array) -> bool:
    k = reduction_from_A_to_B(array)
    output_B: bool = BALL_PARTITION(k)
    return output_B
```

What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Question 2 Solution

Option 2 is wrong with the transformation A

A(S): BALL-PARTITION, S: PARTITION

S = {1, 3} → A(S) = 4

4 balls can be "Ball-Partitioned"

But {1, 3} cannot be Partitioned

```
def PARTITION(array) -> bool:
1.   k = reduction_from_A_to_B(array)
2.   output_B: bool = BALL_PARTITION(k)
3.   return output_B
```

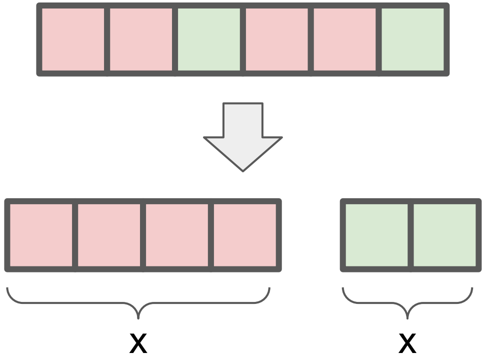
What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Question 2 Solution

Option 3:

If answer to PARTITION is YES, then let the sum of value in **single** partition be *x*.



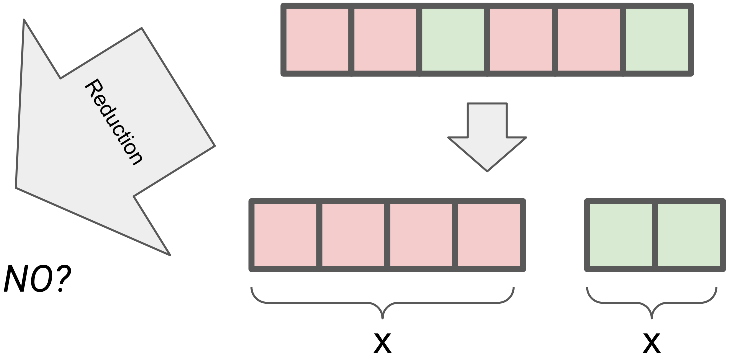
What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Question 2 Solution

Option 3:

If answer to PARTITION is YES, then let the sum of value in **single** partition be *x*.
The total sum $k = 2x$



BALL-PARTITION(2x) - YES or NO?

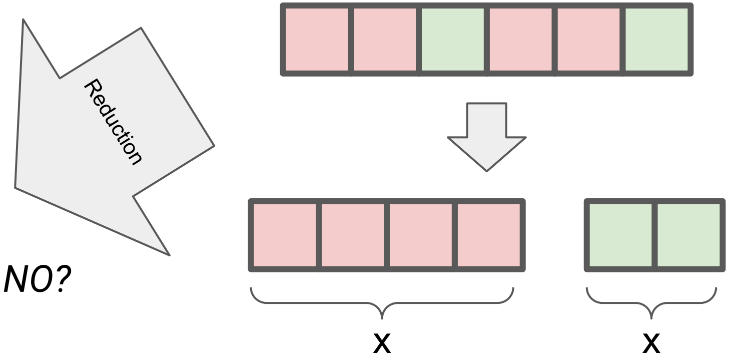
Question 2 Solution

What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Option 3:

If answer to PARTITION is YES, then let the sum of value in **single** partition be *x*. The total sum *k = 2x*, which is an even number → returns YES for BALL-PARTITION



BALL-PARTITION(2x) - ***YES*** *or NO?*

Question 2 Solution

What is wrong with this transformation?

- The transformation is correct.
- A YES solution to $A(S)$ does not imply a YES solution to S.
- A YES solution to S does not imply a YES solution to $A(S)$
- The transformation does not run in polynomial time.

Option 3:

If answer to PARTITION is YES, then let the sum of value in **single** partition be *x*. The total sum $k = 2x$, which is an even number → returns YES for BALL-PARTITION

Option 4:

Adding things up runs in polynomial time

Question 2 Solution

IF the reduction is correct, what does it mean for

PARTITION $\leq_p$ *BALL-PARTITION*?

- BALL-PARTITION is “at least as hard” as PARTITION

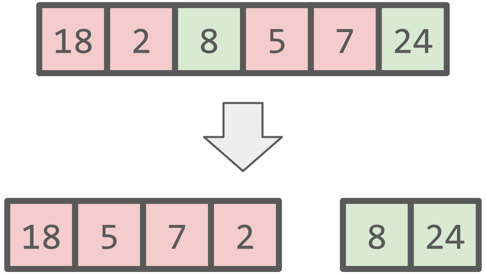
- If there is a polynomial time solution for BALL-PARTITION, then there is a polynomial time solution for PARTITION

But take note, the reduction was NOT correct. This is a hypothetical scenario

Question 3:
PARTITION and KNAPSACK

Question 3: PARTITION

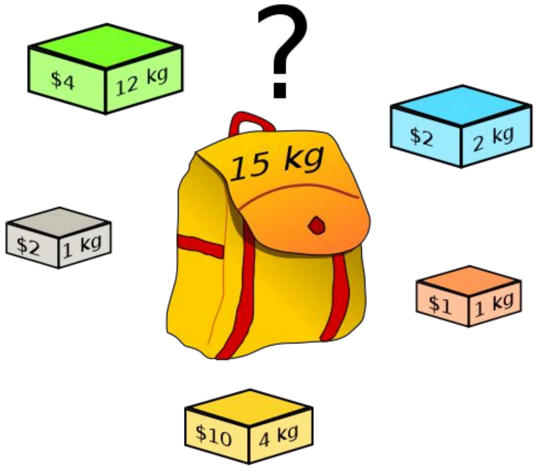
Given a set of positive integers S, can the set be partitioned into two sets of equal total sum?



Question 3: KNAPSACK

Given n items described by non-negative integer pairs $(w_1, v_1), \ldots (w_n, v_n)$, capacity W and threshold V.

Is there a subset of item with total weight at most W and at least V?

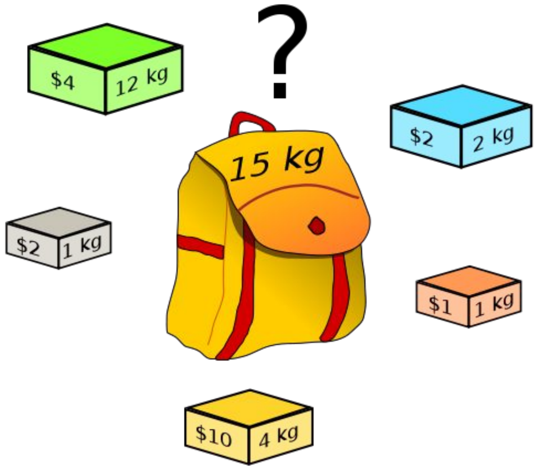


Question 3: KNAPSACK

Given *n* items described by non-negative integer pairs $(w_1, v_1), \ldots (w_n, v_n)$, capacity W and threshold V.

Is there a subset of item with total weight at most W and at least V?

e.g. W = 15, V = 14, and specified items



Question 3: KNAPSACK

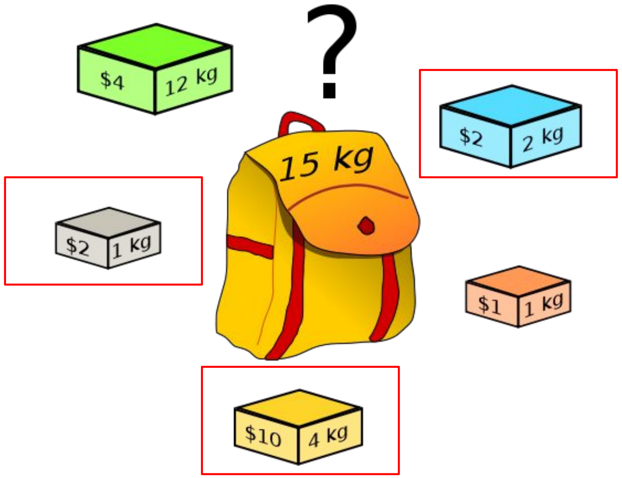
Given n items described by non-negative integer pairs (w_1, v_1), … (w_n, v_n), capacity W and threshold V.

Is there a subset of item with total weight at most W and at least V?

e.g. W = 15, V = 14, and specified items

Total weight: 1 + 4 + 2 = 7 ≤ 15

Total value: 2 + 10 + 2 = 14 ≥ 14

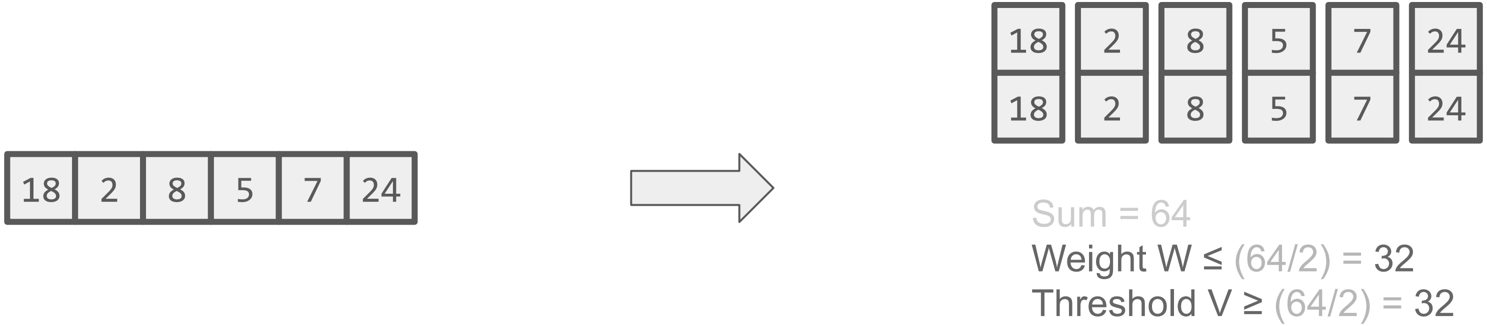


Question 3: Proposed reduction

Given a PARTITION instance $\{w_1, \dots, w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1, w_1), ... (w_n, w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Question 3: Proposed reduction

Given a PARTITION instance $\{w_1, \ldots, w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1, w_1), … (w_n, w_n) with capacity $W = S/2$ and threshold $V = S/2$.



Input to PARTITION

Input to KNAPSACK

Question 3

PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

PARTITION instances with total weights that are odd cannot be partitioned into two equal weight sets, hence can immediately be answered with a NO answer. Consider the following transformation of PARTITION instances with total weights that are even numbers into instances of KNAPSACK:

Given a PARTITION instance $\{w_1, ..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1,w_1), ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Select all true statements.

1. A YES answer to the PARTITION instance implies a YES answer to the KNAPSACK instance.
2. A YES answer to the KNAPSACK instance implies a YES answer to the PARTITION instance
3. The transformation runs in polynomial time

IF the reduction is correct, what does it mean for *PARTITION* $\leq_p$ *KNAPSACK*?

Question 3 Solution

PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1 ,..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1,w_1), ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 1: YES to PARTITION → YES to KNAPSACK

Simply use the same subset in PARTITION for knapsack → weight will be S/2, and value S/2

Question 3 Solution

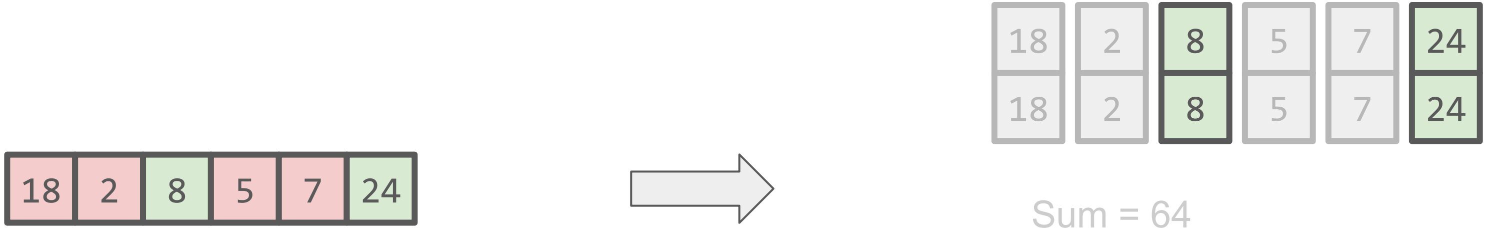
PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1, ..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1,w_1), ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 1: YES to PARTITION → YES to KNAPSACK

Simply use the same subset in PARTITION for knapsack → weight will be S/2, and value S/2



Sum = 64
Weight W ≤ (64/2) = 32
Threshold V ≥ (64/2) = 32

Question 3 Solution

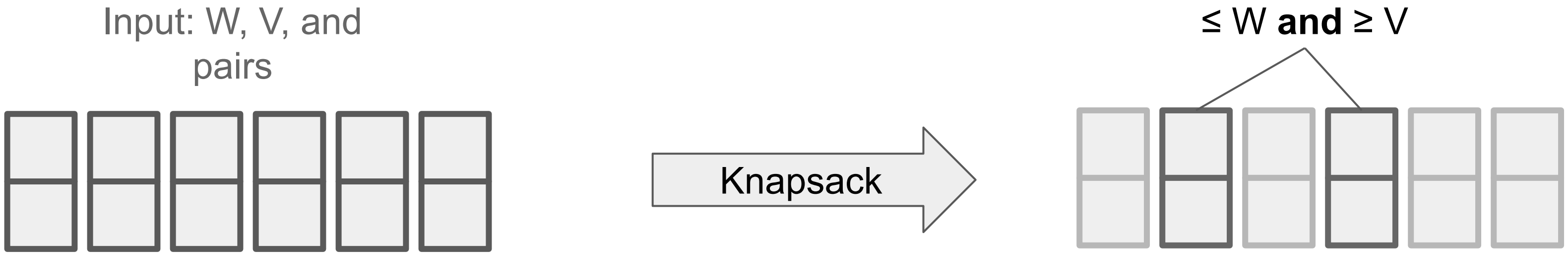
PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1, ..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1,w_1), ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 2: YES to KNAPSACK → YES to PARTITION

We have a subset with total weight ≤ S/2 and total value ≥ S/2



Question 3 Solution

PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

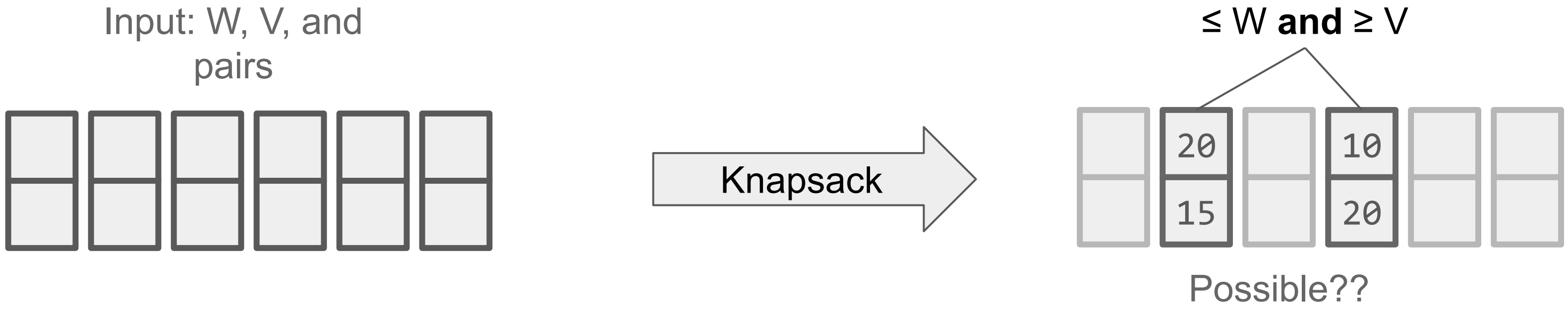
KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1 ,..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1,w_1), ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 2: YES to KNAPSACK → YES to PARTITION

We have a subset with weight total weight ≤ S/2 and total value ≥ S/2

- Let's say S/2 = 32. Is it possible that total weight = 30 (bcs ≤) and total value = 35 (bcs ≥)?



Question 3 Solution

PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

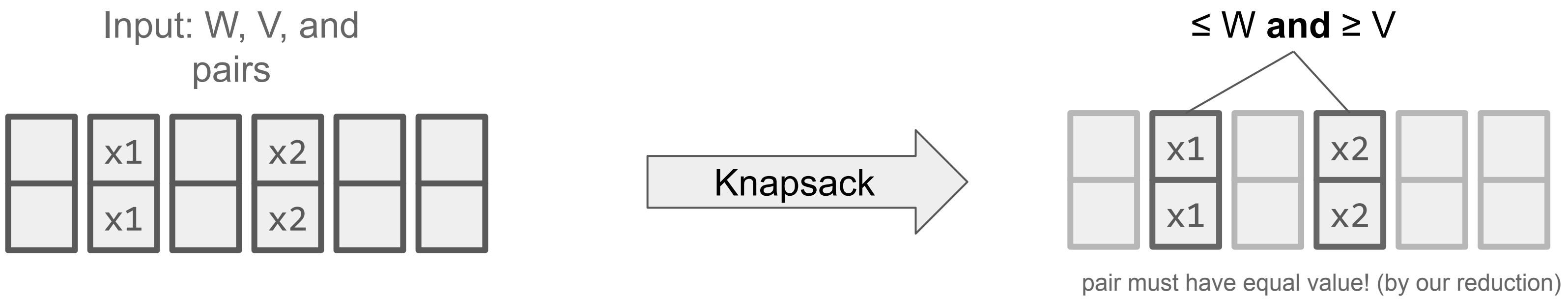
KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1, ..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance $\boxed{(w_1,w_1)}$, ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 2: YES to KNAPSACK → YES to PARTITION

We have a subset with weight total weight ≤ S/2 and total value ≥ S/2

- ~~Let's say S/2 = 32. Is it possible that total weight = 30 (bcs ≤) and total value = 35 (bcs ≥)?~~ **NO!**
- Weight equals value in the transformed instance → total weight = value = S/2



Question 3 Solution

PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

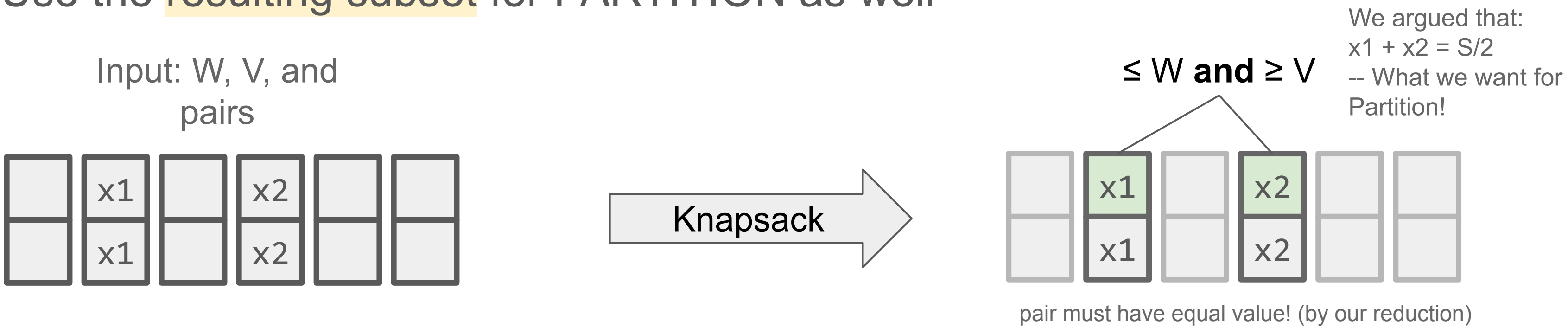
KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1 ,..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance $\boxed{(w_1,w_1),}$ ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 2: YES to KNAPSACK → YES to PARTITION

We have a subset with weight total weight ≤ S/2 and total value ≥ S/2

- ~~Let's say S/2 = 32. Is it possible that total weight = 30 (bcs ≤) and total value = 35 (bcs ≥)?~~ **NO!**
- Weight equals value in the transformed instance → total weight = value = S/2
- Use the resulting subset for PARTITION as well



Question 3 Solution

PARTITION problem: Given a set T of nonnegative integers, can we partition T into two sets of equal total sum?

KNAPSACK problem: Given n items described by non-negative integer pairs (w_1,v_1), ... (w_n,v_n), capacity W and threshold V, is there a subset of item with total weight at most W and total value at least V?

Given a PARTITION instance $\{w_1, ..., w_n\}$ with total sum $S = \sum_{i=1}^{n} w_i$, construct a KNAPSACK instance (w_1,w_1), ... (w_n,w_n) with capacity $W = S/2$ and threshold $V = S/2$.

Option 3:

Transformation simply copies weight to value, so this runs in polynomial time

Question 3 Solution

A ≤ₚ B is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`

We satisfied all the criteria for a polynomial time reduction!

$A \leq_p B$ is a **polynomial time reduction** between decision problems A and B, when it transforms `input_A` of problem A to `input_B` of problem B such that:

- `input_A` is YES-instance → `input_B` is also a YES-instance
- `input_B` is YES-instance → `input_A` is also a YES-instance
- The transformation takes polynomial time in the size of `input_A`

Question 3 Solution

We satisfied all the criteria for a polynomial time reduction!

We have:

$$PARTITION \leq_p KNAPSACK$$

- KNAPSACK is “at least as hard as” PARTITION
- If there is a polynomial time solution for KNAPSACK, then there is a polynomial time solution for PARTITION

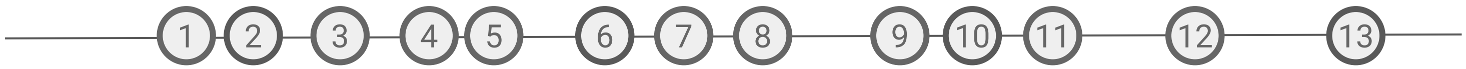
Question 4: WacDonalds

Question 4: WacDonalds

- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible

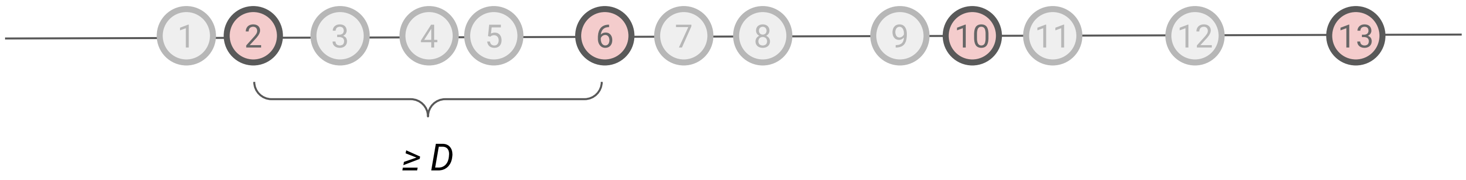
Question 4: WacDonalds

- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found n suitable locations: $a_1, \dots, a_n$



Question 4: WacDonalds

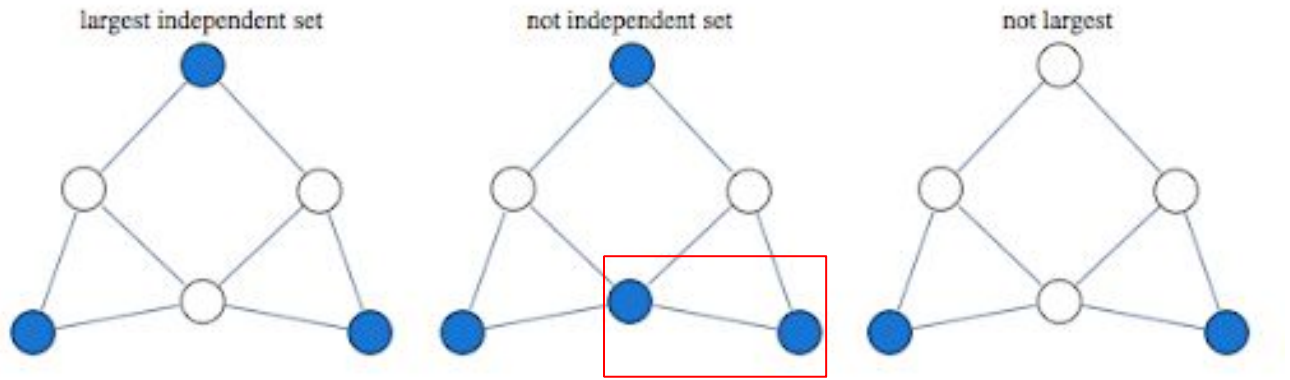
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found n suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least D distance apart to minimise competition



Question 4: Independent Set

Given a graph *G = (V, E)*, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of *V*



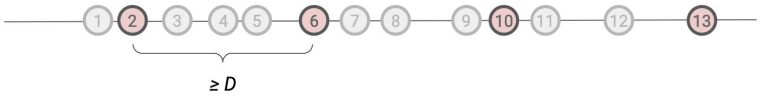
Question 4a: Reduction

Describe how to model WacDonalds as a maximum independent set problem!

i.e. Design an input graph for the maximum independent set problem, so that you can solve WacDonalds

Question 4: WacDonalds

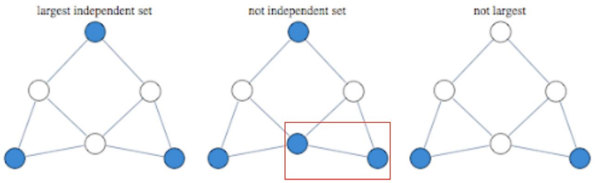
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found n suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least D distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices $\boldsymbol{V}$ such that no two vertices in the graph is connected by an edge

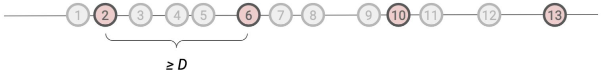
Maximum Independent Set: Largest subset of V



Question 4a: Solution

Question 4: WacDonalds

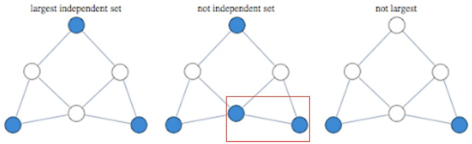
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found *n* suitable locations: $a_1, \dots, a_n$
- The restaurants should be at least *D* distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of *V*

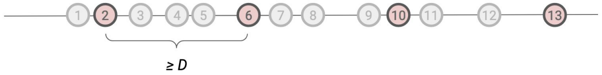


Study the key property of WacDonalds (problem you are reducing from):

Question 4a: Solution

Question 4: WacDonalds

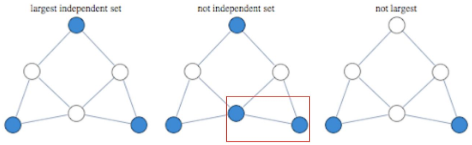
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found *n* suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least *D* distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of *V*



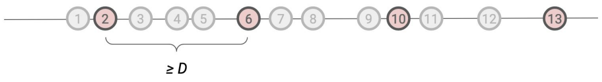
Study the key property of WacDonalds (problem you are reducing from):

- The restaurants opened must be $\geq D$ apart. Everything less not included

Question 4a: Solution

Question 4: WacDonalds

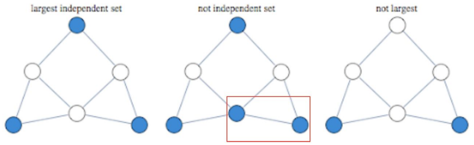
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found *n* suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least *D* distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of *V*



Study the key property of WacDonalds (problem you are reducing from):

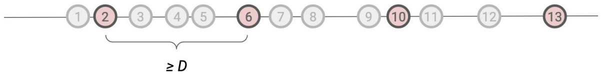
- The restaurants opened must be ≥ *D* apart. Everything less not included

How can Independent set help? What's the key property?

Question 4a: Solution

Question 4: WacDonalds

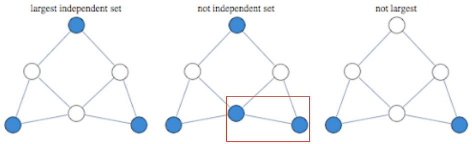
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found n suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least D distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices $\boldsymbol{V}$ such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of V



Study the key property of WacDonalds (problem you are reducing from):

- The restaurants opened must be ≥ D apart. Everything less not included

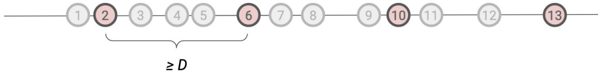
How can Independent set help? What's the key property?

- 2 neighbouring nodes are not included in the independent set

Question 4a: Solution

Question 4: WacDonalds

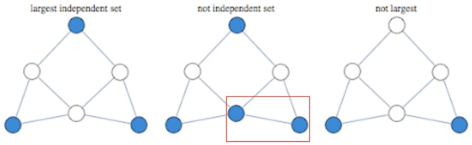
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found n suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least D distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices $\boldsymbol{V}$ such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of V



Study the key property of WacDonalds (problem you are reducing from):

- The restaurants opened must be ≥ D apart. Everything less not included

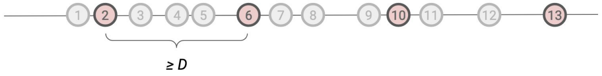
How can Independent set help? What's the key property?

- 2 neighbouring nodes are not included in the independent set

Question 4a: Solution

Question 4: WacDonalds

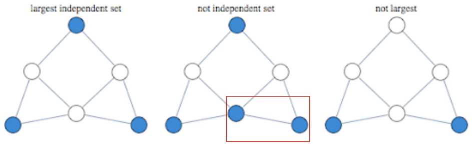
- WacDonalds want to open as many of its chain restaurant on Orchard Road as possible
- It found n suitable locations: $a_1, \ldots, a_n$
- The restaurants should be at least D distance apart to minimise competition



Question 4: Independent Set

Given a graph $G = (V, E)$, **independent set** is a subset of vertices ***V*** such that no two vertices in the graph is connected by an edge

Maximum Independent Set: Largest subset of V



Study the key property of WacDonalds (problem you are reducing from):

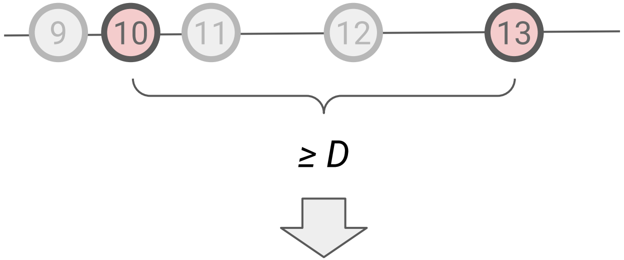
- The restaurants opened must be ≥ D apart. Everything less not included

How can Independent set help? What's the key property?

- 2 neighbouring nodes are not included in the independent set
- Node = location, edge = given to pair of nodes w/ dist less than D

Question 4a: Solution

- Node represents location. Two nodes v_i and v_j have an edge if distance between the two locations are less than D

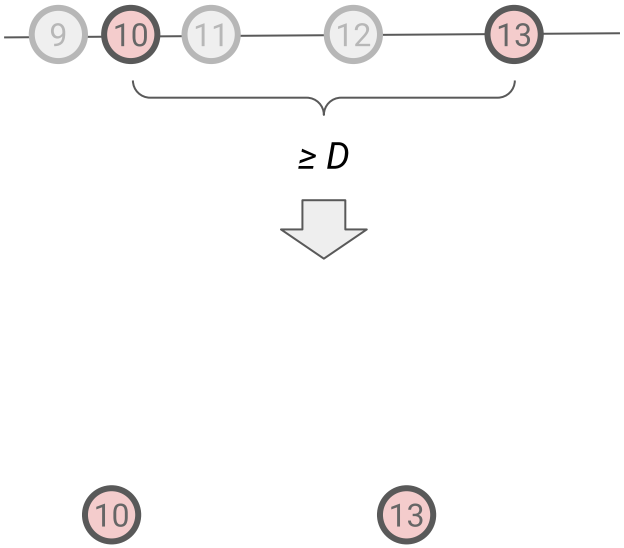


Q4 Reduction

Start off with this restaurant locations

Question 4a: Solution

- Node represents location. Two nodes v_i and v_j have an edge if distance between the two locations are less than D

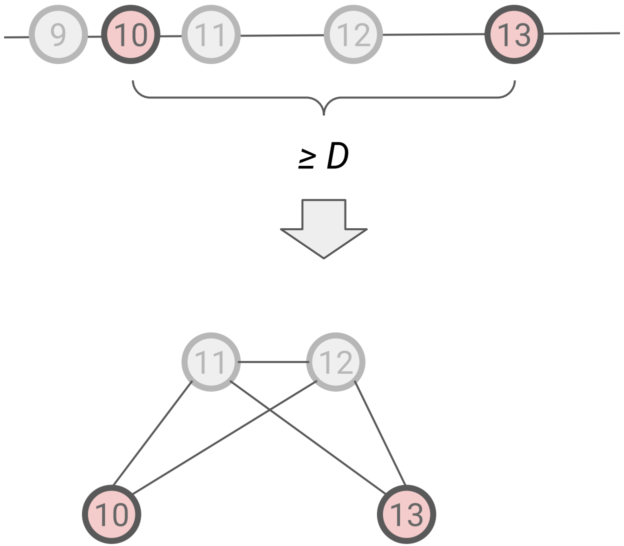


Q4 Reduction

```
10 to 13 is safe, because they are
at least D apart
```

Question 4a: Solution

- Node represents location. Two nodes v_i and v_j have an edge if distance between the two locations are less than D

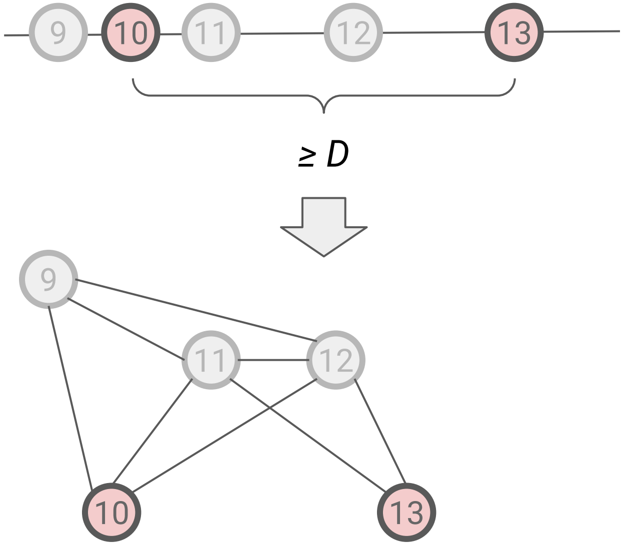


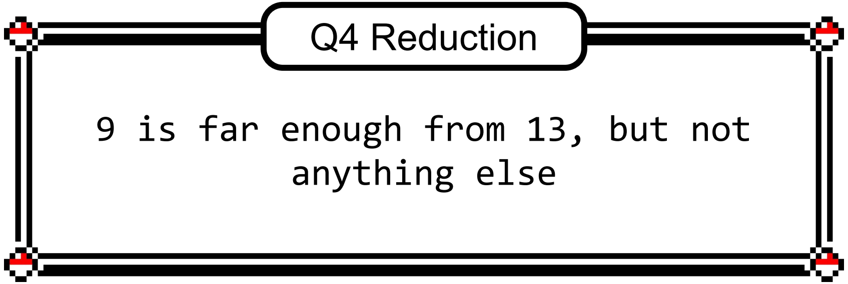
Q4 Reduction

11 and 12 are too close to both 10 and 13

Question 4a: Solution

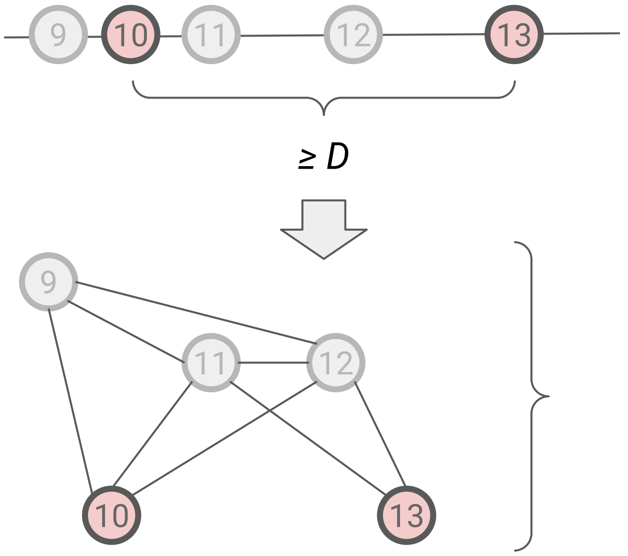
- Node represents location. Two nodes v_i and v_j have an edge if distance between the two locations are less than D





Question 4a: Solution

- Node represents location. Two nodes v_i and v_j have an edge if distance between the two locations are less than D



Q4 Reduction

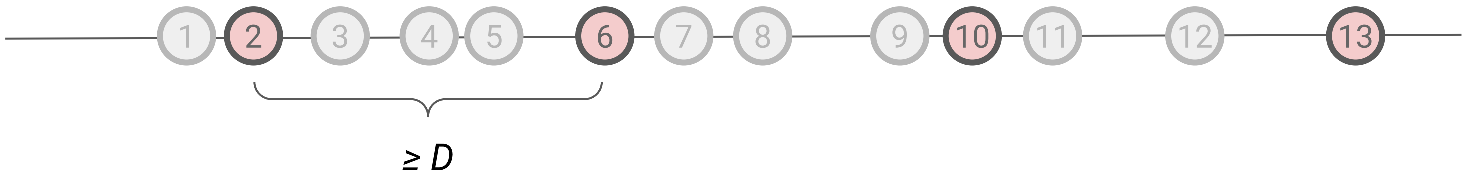
```
This is now input to Maximum
Independent Set!
```

Question 4b: Optimal Substructure

Look for optimal substructure in YuckDonalds problem and argue that

$$M(V) = 1 + M(V - \{a_j \in V : d(a_i, a_j) < D\})$$

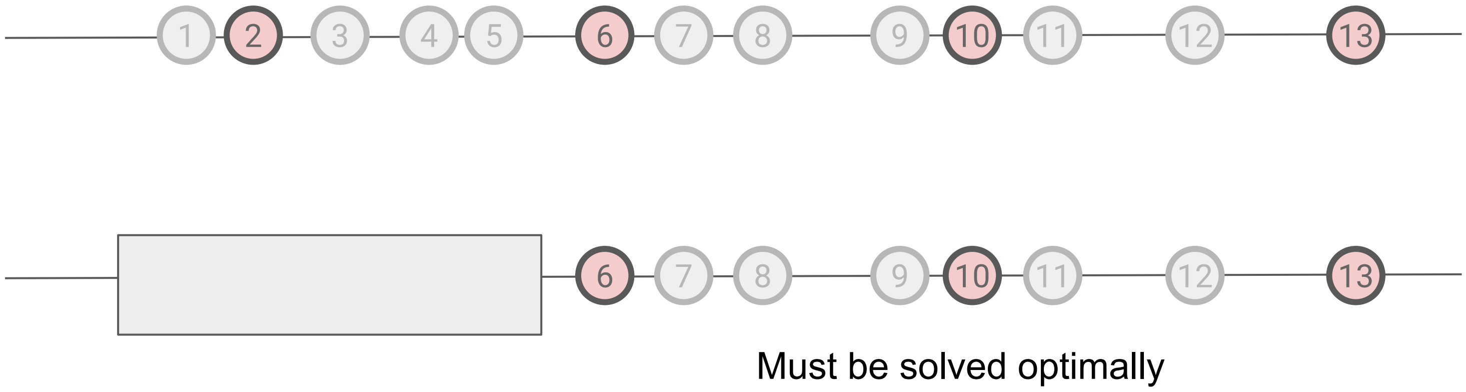
where $M(V)$ is the size of the largest set of restaurants that can fit in V, a_i is an element in an optimal solution and $d(a_i, a_j)$ is the distance between a_i and a_j.



Question 4b: Solution

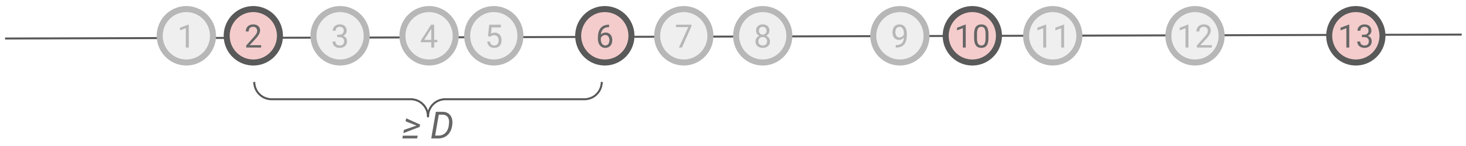
Consider an optimal solution OPT:

- Remove a location in OPT and everything that has distance < *D*
- The remaining problem must be solved optimally as well
- (Do the proof like usual…)



Question 4c: Greedy Choice

Assume that the locations for a set of restaurant $a_1, \ldots, a_n$ is sorted and lie on a straight line. Show that a_1 is part of some optimal solution, i.e. selecting the smallest element satisfies the greedy choice property.

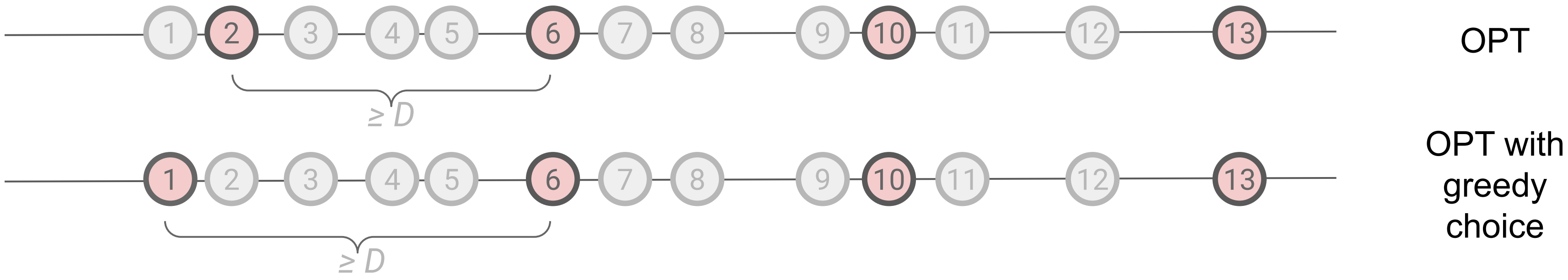


Question 4c: Solution

Consider an optimal solution OPT:

- If a_1 is in OPT -- done
- Otherwise, take the smallest a_j in OPT, and swap it with a_1

The solution is still optimal



Question 4d: Algorithm and Correctness

Complete your greedy algorithm for evaluating $M(V)$ and argue its correctness by mathematical induction.

Question 4d: Solution

Complete your greedy algorithm for evaluating $M(V)$ and argue its correctness by mathematical induction.

Algorithm:

1. Add the leftmost element
2. Remove all elements with distance less than D from the added element
3. Repeat until no element left

Algorithm:

1. Add the leftmost element
2. Remove all elements with distance less than D from the added element
3. Repeat until no element left

Question 4d: Solution

Correctness:

1. Base case: empty set of locations → output nothing

Algorithm:

1. Add the leftmost element
2. Remove all elements with distance less than D from the added element
3. Repeat until no element left

Question 4d: Solution

Correctness:

1. Base case: empty set of locations → output nothing
2. Inductive hypothesis: Assume algorithm correct when $|V| < k$ for some k

Algorithm:

1. Add the leftmost element
2. Remove all elements with distance less than D from the added element
3. Repeat until no element left

Question 4d: Solution

Correctness:

1. Base case: empty set of locations → output nothing
2. Inductive hypothesis: Assume algorithm correct when $|V| < k$ for some k
3. Consider set V with $|V| = k$
    a. Greedy choice: leftmost element part of optimal independent set

Algorithm:

1. Add the leftmost element
2. Remove all elements with distance less than D from the added element
3. Repeat until no element left

Question 4d: Solution

Correctness:

1. Base case: empty set of locations → output nothing
2. Inductive hypothesis: Assume algorithm correct when $|V| < k$ for some k
3. Consider set V with $|V| = k$
    a. Greedy choice: leftmost element part of optimal independent set
    b. Optimal substructure: after removal, we need to solve the remaining set optimally

Algorithm:

1. Add the leftmost element
2. Remove all elements with distance less than D from the added element
3. Repeat until no element left

Question 4d: Solution

Correctness:

1. Base case: empty set of locations → output nothing
2. Inductive hypothesis: Assume algorithm correct when $|V| < k$ for some k
3. Consider set V with $|V| = k$
   a. Greedy choice: leftmost element part of optimal independent set
   b. Optimal substructure: after removal, we need to solve the remaining set optimally
   c. Inductive Hypothesis: we can solve the remaining set optimally

Question 4e: Reduction Implications

- In the previous parts, we will show that WacDonalds can be solved greedily in polynomial time
- We have a reduction based on 4a

Question 4e: Reduction Implications

- In the previous parts, we will show that WacDonalds can be solved greedily in polynomial time
- We have a reduction based on 4a

- Since we have a polynomial time algorithm for WacDonalds, we also get a polynomial time algorithm for maximum independent set problem.
  True or False?

Question 4e: Solution

False

Reduction in 4a) was made from WacDonalds to Maximum Independent Set

Question 4e: Solution

False

Reduction in 4a) was made from WacDonalds to Maximum Independent Set

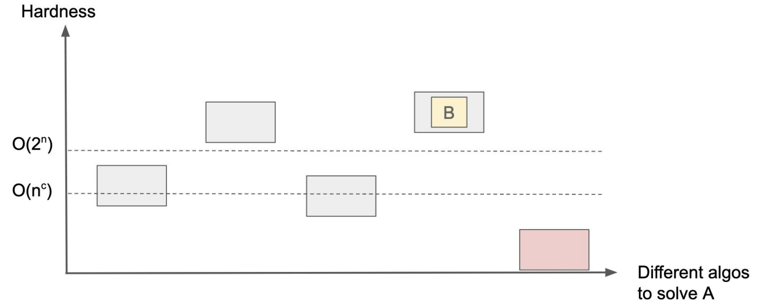
$A \leq_p B$

Intuition for comparing hardness

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A) # p(n) time
    output_B = solve_B(input_B) # T(p(n)) time

    output_A = transform_output_B_to_A(output_B) # p(n) time
    return output_A
```

Note: If A is "easy", it does **not** say anything about B.
A could have been "easy" because of another algorithm



Reductions: Algorithm Analysis perspective

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is "easily solvable", then so is A
- If A is "hard", then so is B

IMPORTANT
Please get the direction of reduction right!

To show a problem is "hard", you need to reduce **FROM** a "hard" problem

Doing it in the wrong direction means you are proving the wrong thing →
likely to get 0 marks for the question if in an exam

Summary

- Reductions allows us to compare “hardness” of problems

- Further usefulness:
  - If suppose problem A has no algorithm to solve it (such a class exists!)
  - Reducing A to B means that you can say for “free” that no algorithm solves B

Summary

```
def solve_A(input_A):
    input_B = reduction_from_A_to_B(input_A)
    output_B = solve_B(input_B) # magically given

    output_A = transform_output_B_to_A(output_B)
    return output_A
```

Notation for poly-time reduction from A to B:

$$A \leq_p B$$

- If B has poly time algorithm, then so does A
- If B is “easily solvable”, then so is A
- If A is “hard”, then so is B

IMPORTANT

Please get the direction of reduction right!

To show a problem is “hard”, you need to reduce **FROM** a “hard” problem

Doing it in the wrong direction means you are proving the wrong thing → likely to get 0 marks for the question if in an exam