CS3230 Tutorial 11

Review

(AY 25/26 Semester 2)

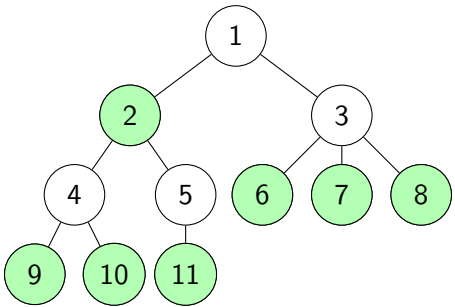
April 6, 2026

(Prepared by Benson and Suhail)

Contents

Q1(a). Max Independent Set of a Tree (DP)

Given a tree with n vertices and $n-1$ edges. Devise an algorithm to find its maximum independent set in $O(n)$ time.



Recall: A set of vertices I is called an **independent set** if no two vertices in I are adjacent to each other.

Q1(a). Max Independent Set of a Tree (DP)

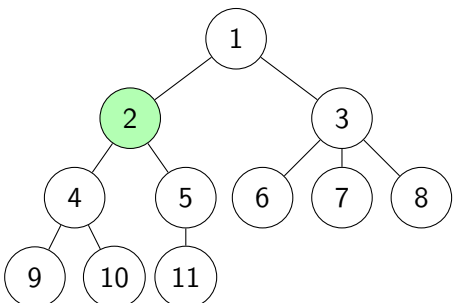
Thought process for a dynamic programming solution:

1. What are the decisions to be made? ($\Rightarrow$ Optimal Substructure)
2. In what order do we make these decisions?
3. What information do we need to make a decision / calculate the answer? ($\Rightarrow$ State)

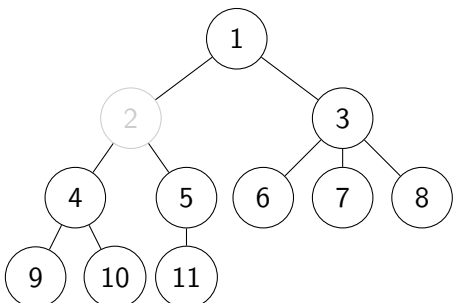
Q1(a). Max Independent Set of a Tree (DP)

What is the decision to be made?

To include a vertex



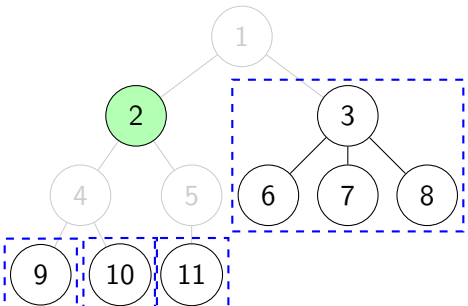
Not to include a vertex



Q1(a). Max Independent Set of a Tree (DP)

Optimal Substructure Property:

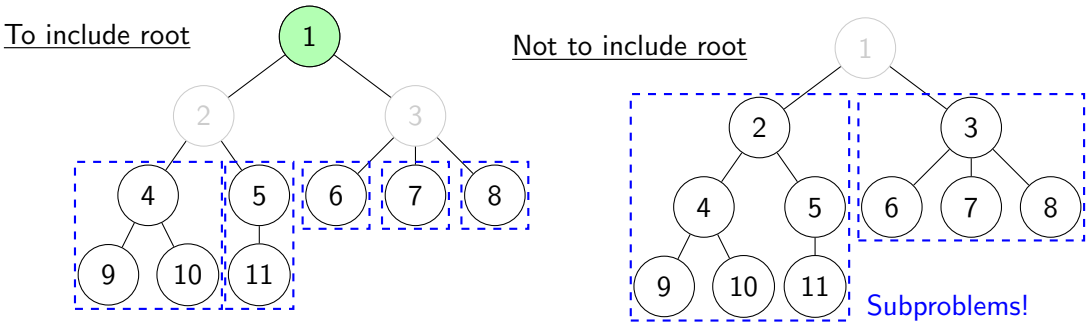
- Suppose we decided to pick a vertex v to be part of the independent set.
- What subproblem should we reduce to?
  - Remove v **and all its neighbours**. We end up with a forest of trees!
- Each tree in the forest (the subproblem) is optimally solved.



Q1(a). Max Independent Set of a Tree (DP)

In what reasonable order do we make the decisions?

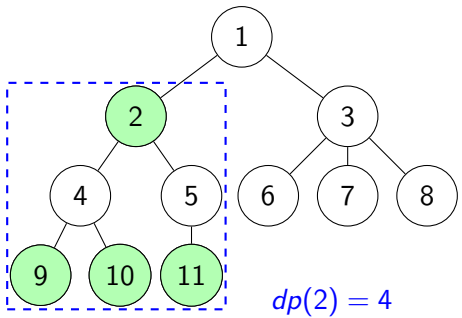
- From the root down to the leaves!



Q1(a). Max Independent Set of a Tree (DP)

State: A snapshot of the problem at a particular point during the computation.

- $dp(i)$: Size of the Maximum Independent Set for the subtree rooted at vertex i.

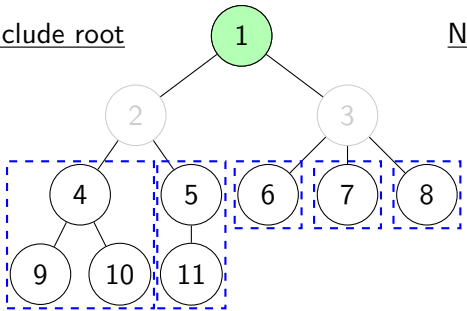


Q1(a). Max Independent Set of a Tree (DP)

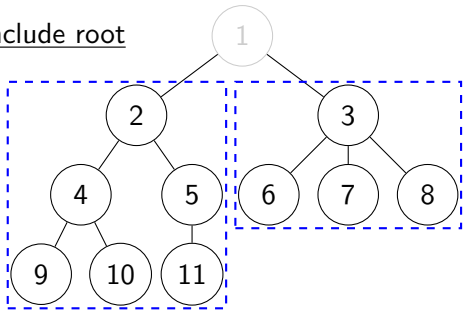
Recurrence: How you would calculate the value of a state from its subproblems.

- $dp(i)$: Size of the Maximum Independent Set for the subtree rooted at vertex i.
- $dp(i) = \max \left\{ 1 + \sum_{\text{grandchild } v \text{ of } i} dp(v), \sum_{\text{child } v \text{ of } i} dp(v) \right\}$

To include root



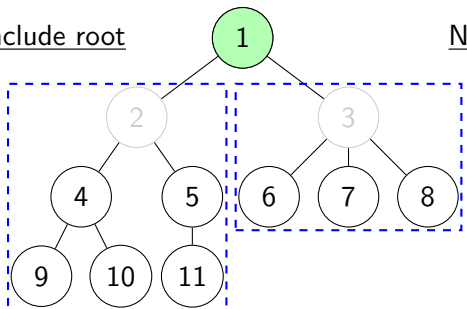
Not to include root



Q1(a). Max Independent Set of a Tree (DP)

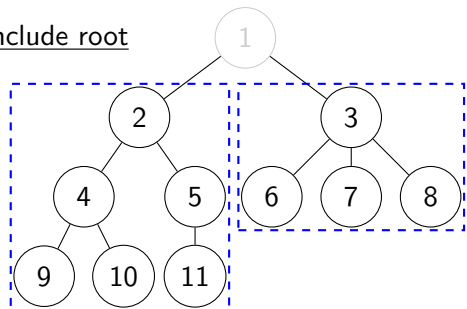
- This works but... can we avoid finding the grandchildren of the vertex i?

To include root



For the subproblems, the root of the subtree cannot be chosen.

Not to include root

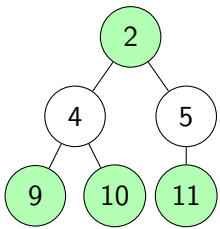


For the subproblems, the root of the subtree can be chosen.

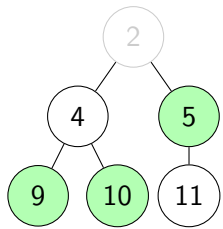
Q1(a). Max Independent Set of a Tree (DP)

State: A snapshot of the problem at a particular point during the computation.

- $dp(i, hasRoot)$: Size of the Maximum Independent Set for the subtree rooted at vertex i, where $hasRoot$ is **true** $\Leftrightarrow$ vertex i is included in the set.



$dp(2, \textbf{true}) = 4$



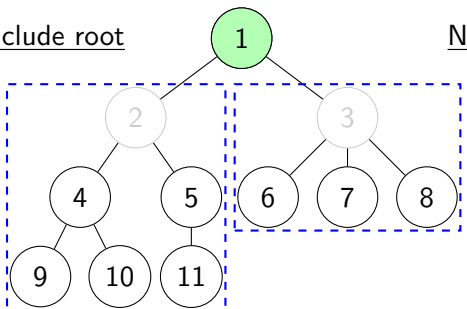
$dp(2, \textbf{false}) = 3$

Q1(a). Max Independent Set of a Tree (DP)

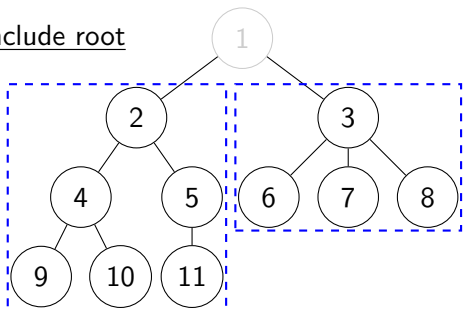
Recurrence: How you would calculate the value of a state from its subproblems.

- $dp(i, \textbf{true}) = 1 + \sum_{\text{child } v \text{ of } i} dp(v, \textbf{false})$
- $dp(i, \textbf{false}) = \sum_{\text{child } v \text{ of } i} max\{dp(v, \textbf{true}), dp(v, \textbf{false})\}$

To include root



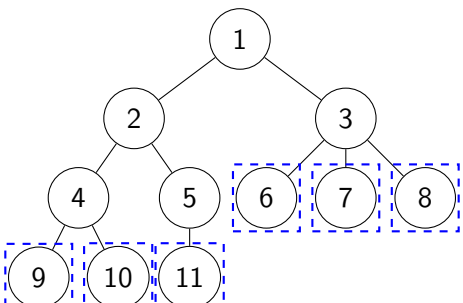
Not to include root



Q1(a). Max Independent Set of a Tree (DP)

Base case: States whose values are **trivial to calculate**.

- If vertex i is a leaf: $dp(i, \textbf{true}) = 1, dp(i, \textbf{false}) = 0$.



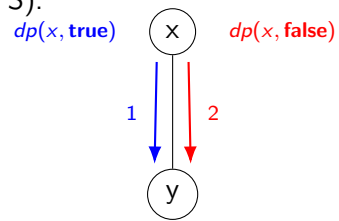
Q1(a). Max Independent Set of a Tree (DP)

We can compute the answer in a **top-down** manner. Since $dp(i, includeRoot)$ **only depends on its children** and the tree is **acyclic**, we will always reach base case (leaves).

The final answer will be $\max\{dp(root, \textbf{true}), dp(root, \textbf{false})\}$.

There are clearly $2n$ states. Each state processes all children of vertex i. Across all states, each edge is considered **at most twice** (similar to DFS).

Total Time Complexity: $O(n)$
Space Complexity: $O(n)$ (for DP + recursion stack)

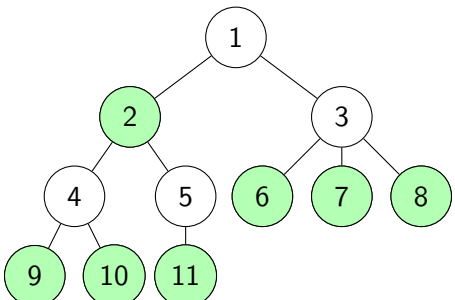


Q1(b). Max Independent Set of a Tree (Greedy)

Is a greedy solution possible?

Q1(b). Max Independent Set of a Tree (Greedy)

Given a tree with n vertices and $n-1$ edges. Devise an algorithm to find its maximum independent set in $O(n)$ time.

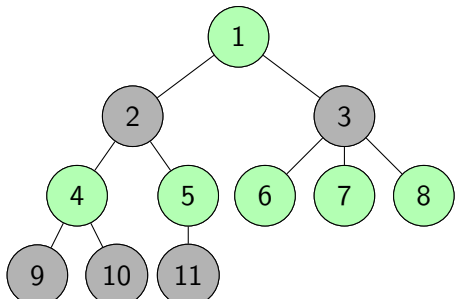


Q1(b). Max Independent Set of a Tree (Greedy)

What kind of greedy choice can we make?

Always pick root?

- Doesn't work :(



Q1(b). Max Independent Set of a Tree (Greedy)

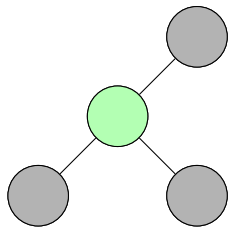
What kind of greedy choice can we make?

What happens when we choose a vertex?

- Blocks all neighbours!

Vertices with fewest neighbours?

- Leaves!



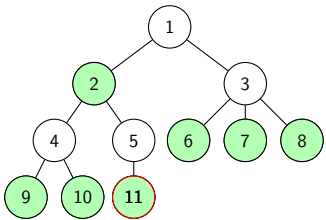
Q1(b). Max Independent Set of a Tree (Greedy)

Greedy Choice Property:

Let v be a leaf in the tree. There exists a maximum independent set containing v.

Proof (by exchange argument):

- Consider any optimal solution OPT.



- If OPT already contains v, we are done.

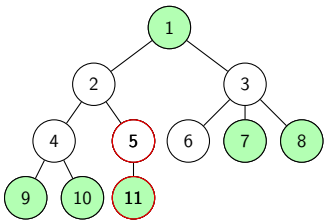
Q1(b). Max Independent Set of a Tree (Greedy)

Greedy Choice Property:

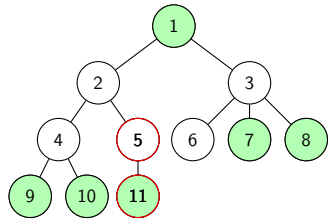
Let v be a leaf in the tree. There exists a maximum independent set containing v.

Proof (by exchange argument):

- If OPT doesn't contain v.
  - Case 1: v's parent is **chosen.**



  - Exchange v's parent with v – Equally optimal solution!
  - Case 2: v's parent is **not chosen**.

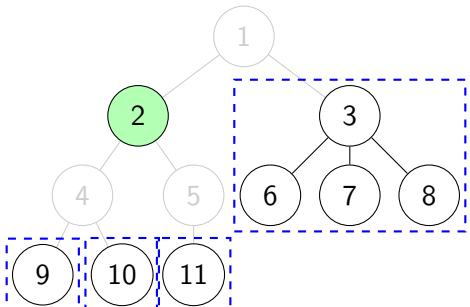


  - v can be chosen – Better solution! (Contradiction!)

Q1(b). Max Independent Set of a Tree (Greedy)

Optimal Substructure Property:

- Suppose we decided to pick a vertex v to be part of the independent set.
- What subproblem should we reduce to?
  - Remove v **and all its neighbours**. We end up with a forest of trees!
- Each tree in the forest (the subproblem) is optimally solved.

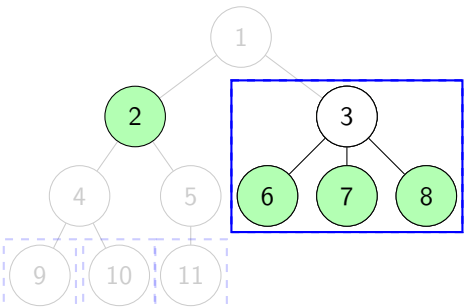


Q1(b). Max Independent Set of a Tree (Greedy)

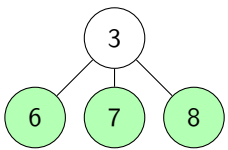
Optimal Substructure Property:

Let $v \in OPT$. Remove vertex v and all its neighbours. OPT contains a Maximum Independent Set in each tree in the resulting forest.

Proof Sketch (by cut-and-paste argument):



- Suppose not.

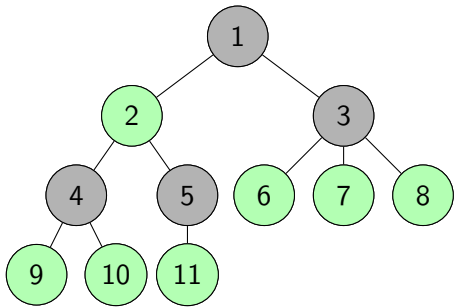


- Better solution! Contradiction!

Q1(b). Max Independent Set of a Tree (Greedy)

Algorithm Greedy Algorithm

```
1:  procedure FindIndependentSet(i)
2:      I ← ∅
3:      isRemoved ← false
4:      for each child v of i do
5:          I ← I ∪ FindIndependentSet(v)
6:          if v ∈ I then
7:              isRemoved ← true
8:      if not isRemoved then
9:          I ← I ∪ {i}
10:     return I
```



Q1(b). Max Independent Set of a Tree (Greedy)

Algorithm Greedy Algorithm

```
1:  procedure FindIndependentSet(i)
2:      I ← ∅
3:      isRemoved ← false
4:      for each child v of i do
5:          I ← I ∪ FindIndependentSet(v)
6:          if v ∈ I then
7:              isRemoved ← true
8:      if not isRemoved then
9:          I ← I ∪ {i}
10:     return I
```

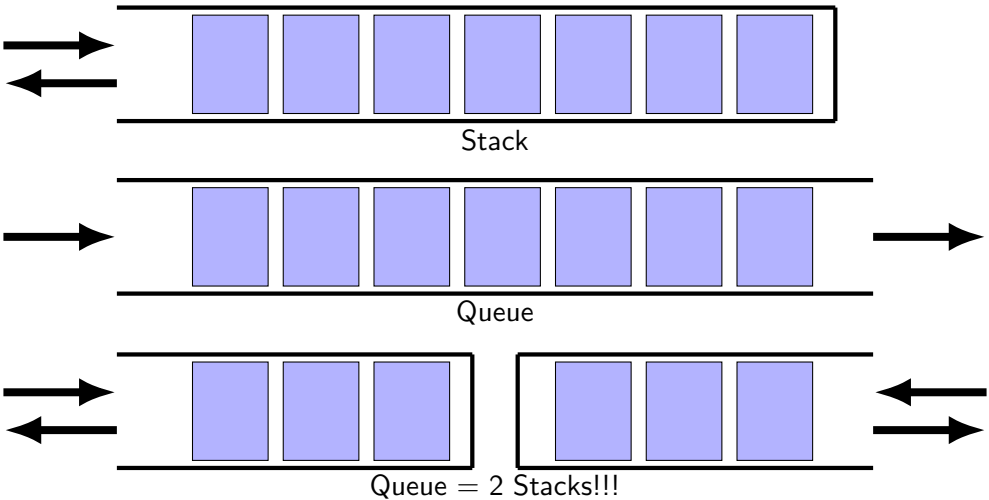
- **Time Complexity:** $O(n)$
- **Space Complexity:** $O(n)$

Q2. An efficient queue using two stacks

Suppose you have Python lists that supports the operations `L.append(x)`, `L.pop()` and `len(L)` (amortized) in $O(1)$. Explain how you can use *two* lists to implement a FIFO queue efficiently, i.e. $\textsc{Push}(x)$, $\textsc{Pull}()$, $\textsc{Size}()$ operations remains in (amortized) $O(1)$.

Q2. An efficient queue using two stacks

Idea: (Thanks to Hua Jun for drawing the figures in CS3233 slides!)



Q2. An efficient queue using two stacks

This animation is not available in the review version. Please use the animated version of the slides.

Q2. An efficient queue using two stacks

Algorithm:

Algorithm An efficient queue using two stacks

```
1: Initialize two empty Python lists s1 and s2.
2: procedure PUSH(x)
3:     s1.append(x)
4: procedure LEN()
5:     return len(s1) + len(s2)
6: procedure POP()
7:     if len(s2) = 0 then                ▷ s2 is empty, transfer everything from s1 to s2
8:         while len(s1) > 0 do
9:             s2.append(s1.pop())
10:    return s2.pop()
```

Q2. An efficient queue using two stacks

Let the lengths of `s1` and `s2` be ℓ_1 and ℓ_2 respectively.
Let the costs of `append`, `pop` and `len` be c_1, c_2 and c_3 respectively.

Operation	**Actual Cost**	$\phi(i) - \phi(i-1)$	**Amortized Cost**
PUSH	c_1		◎ $O(1)$
LEN	$2c_3$		◎ $O(1)$
POP (`len(s2)` $= 0$)	$c_1 \times \ell_1$ $+c_2 \times (\ell_1 + 1)$ $+c_3 \times (\ell_1 + 2)$		◎ $O(1)$
POP (`len(s2)` $\neq 0$)	$c_2 + c_3$		◎ $O(1)$

Q2. An efficient queue using two stacks

Main Idea: Find something which **decreases a lot** after performing the expensive operation!

Potential function: $\phi(i) = (c_1 + c_2 + c_3)\ell_1$.

- Since ℓ_1 is empty at the beginning, $\phi(0) = 0$.
- Since c_1, c_2, c_3 and ℓ_1 are all non-negative, $\phi(i) \geq 0$.

Operation	**Actual Cost**	$\phi(i) - \phi(i-1)$	**Amortized Cost**
PUSH	c_1	$c_1 + c_2 + c_3$	$2c_1 + c_2 + c_3$
LEN	$2c_3$	0	$2c_3$
POP (`len(s2)` $= 0$)	$c_1 \times \ell_1$ $+c_2 \times (\ell_1 + 1)$ $+c_3 \times (\ell_1 + 2)$	$-(c_1+c_2+c_3)\ell_1$	$c_2 + 2c_3$
POP (`len(s2)` $\neq 0$)	$c_2 + c_3$	0	$c_2 + c_3$

Q3. Subset-Sum $\leq_p$ Partition

Consider the following two problems.

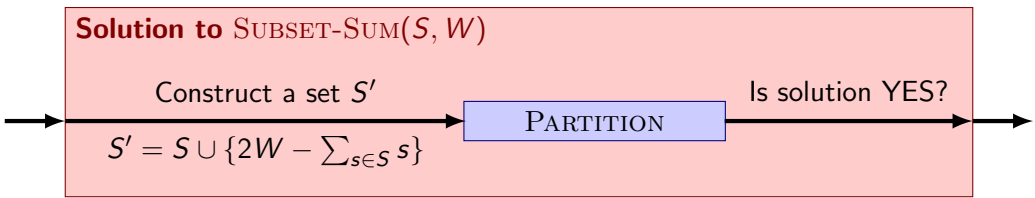
- **Subset-Sum**: Given a set of n non-negative integers $S = \{w_1, \ldots, w_n\}$ and a target W, decide whether there exists a subset $I \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in I} w_i = W$. (In other words, decide whether there exists a subset of S with sum W.)

YES-instance	NO-instance
$W = 0$ 	$W = 4$ 

- **Partition**: See Tutorial 11 slides.

Show that Subset-Sum $\leq_p$ Partition.

Q3. Subset-Sum $\leq_p$ Partition



Intuition:

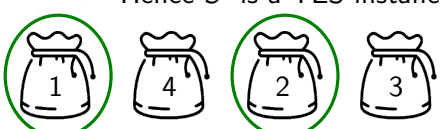
- We want the Partition blackbox to give us a set that sums to W.
  $\Rightarrow$ The set S' should sum to $2W$ so that each partition sums to W.
- How can we make that happen? By inserting an element into S!

Q3. Subset-Sum $\leq_p$ Partition

Theorem. (S, W) is a YES-instance to Subset-Sum if and only if S' is a YES-instance to Partition.

Proof:

- ($\Rightarrow$) Given that (S, W) is a YES-instance to Subset-Sum.
  - Then there exists $T \subseteq S$ such that T sums to W.
  - Take T as one component of the partition and $S' \setminus T$ as the other component. Then T sums to W and $S' \setminus T$ sums to $2W - W = W$.
  - Hence S' is a YES-instance to Partition.



$W = 3$

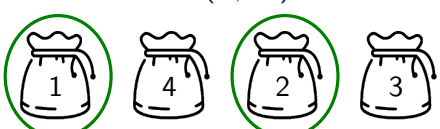
Sums to $2W = 6$

Q3. Subset-Sum $\leq_p$ Partition

Theorem. (S, W) is a YES-instance to Subset-Sum if and only if S' is a YES-instance to Partition.

Proof:

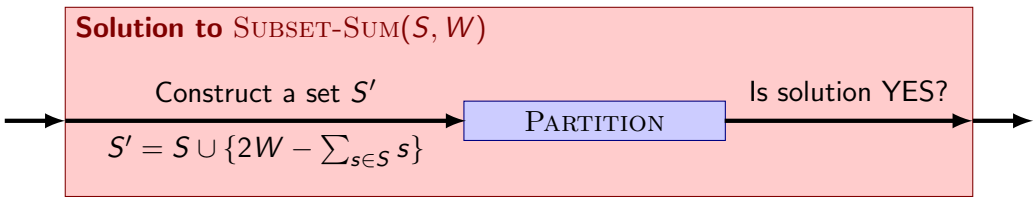
- ($\Leftarrow$) Given that S' is a YES-instance to Partition.
  - Then there exists a partition (P_1, P_2) of S', both P_1 and P_2 sums to W.
  - One of P_1 and P_2 does not contain the new element $2W - \sum_{s \in S} s$. That is a subset of S that sums to W.
  - Hence (S, W) is a YES-instance to Subset-Sum.



$W = 3$

Sums to $2W = 6$

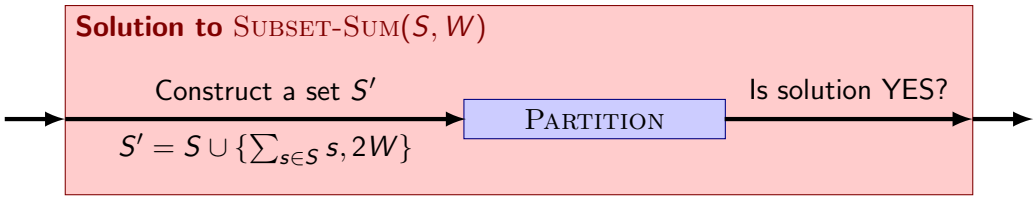
Q3. SUBSET-SUM $\leq_p$ PARTITION



Any issues?

- PARTITION requires all the input elements to be **non-negative**.
- Our input instance does not ensure this constraint: When $2W < \sum_{s \in S} s$.

Q3. Subset-Sum $\leq_p$ Partition



The correct solution: Add two elements instead!

- Insert the elements $\sum_{s \in S} s$ and $2W$.
  - Total sum: $2\sum_{s \in S} s + 2W$.
  - The partition gives us two (disjoint) sets of sum $\sum_{s \in S} s + W$.
  - **If W is positive**, these two new elements cannot be placed together.
- Proof is left as exercise!