CS11313 - **Fall** 2023

Design & Analysis of Algorithms

Reductions

Ibrahim Albluwi

Reductions

A *reduction* from problem X to problem Y:
An algorithm for solving problem X that includes a solver of problem Y as a subroutine.

Reductions

A *reduction* from problem X to problem Y:
An algorithm for solving problem X that includes a solver of problem Y as a subroutine.

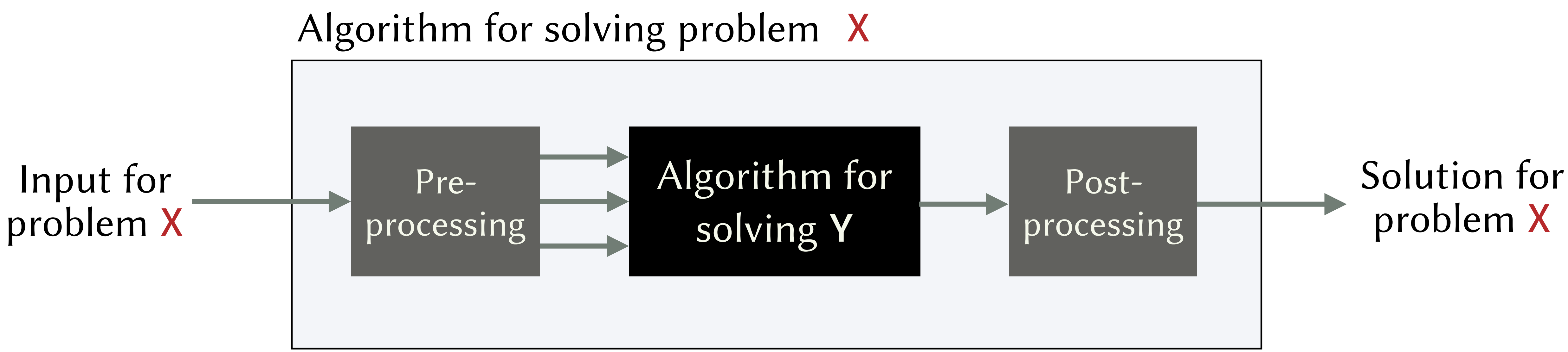
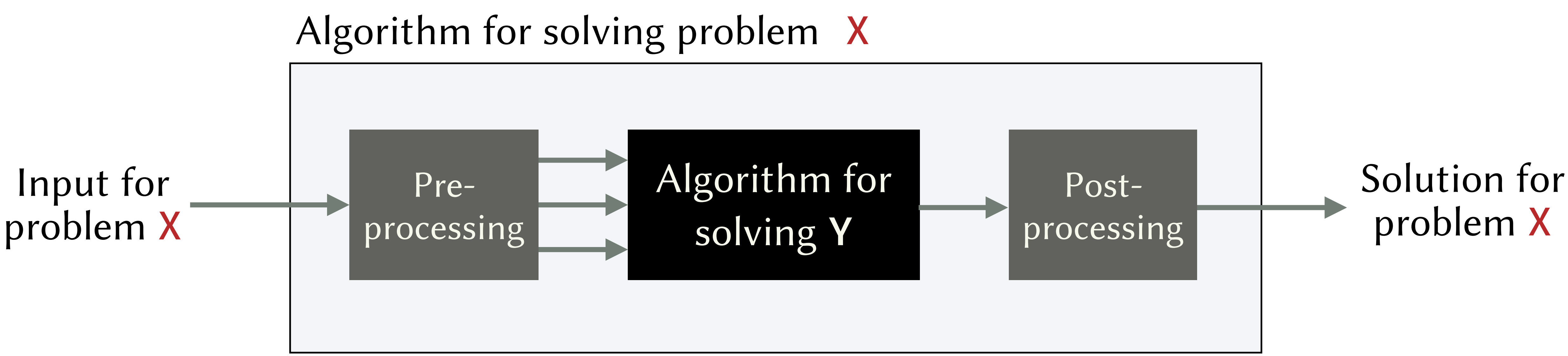


Figure adapted from a slide by Kevin Wayne

Reductions

A *reduction* from problem X to problem Y:
An algorithm for solving problem X that includes a solver of problem Y as a subroutine.



Total cost for solving X = Cost of solving Y + Cost of reduction

Y might be called multiple time (typically 1 call)

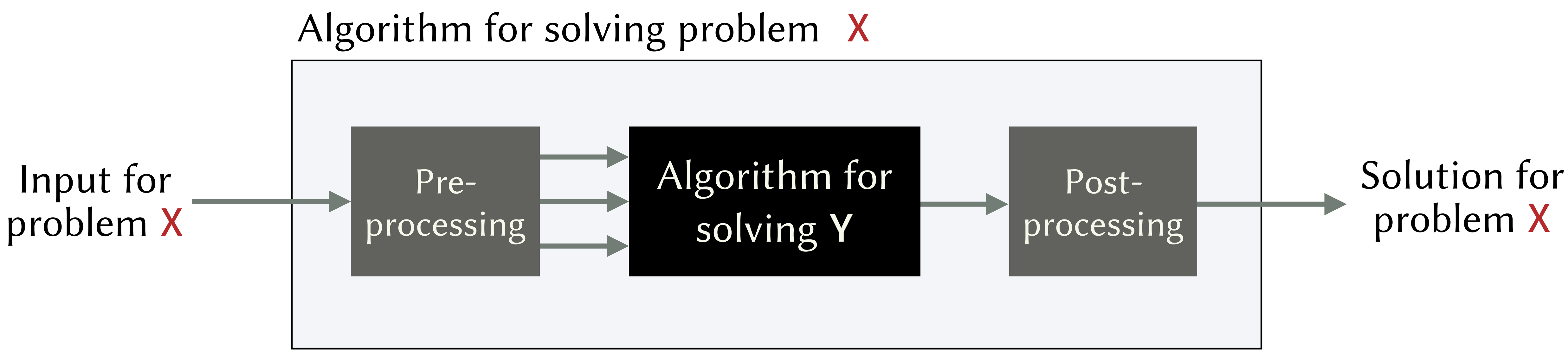
Typically less than the cost of solving Y

Figure adapted from a slide by Kevin Wayne

Reductions

A *reduction* from problem X to problem Y:
An algorithm for solving problem X that includes a solver of problem Y as a subroutine.

Problem X *reduces* to problem Y (denoted as $X \leqslant Y$):
An algorithm for solving Y can be used to solve X.



Total cost for solving X = Cost of solving Y + Cost of reduction

Y might be called multiple time (typically 1 call)

Typically less than the cost of solving Y

Reductions (Examples)

LINEAR
Given b and c, solve $bx + c = 0$

QUADRATIC
Given a, b and c, solve $ax^2 + bx + c = 0$

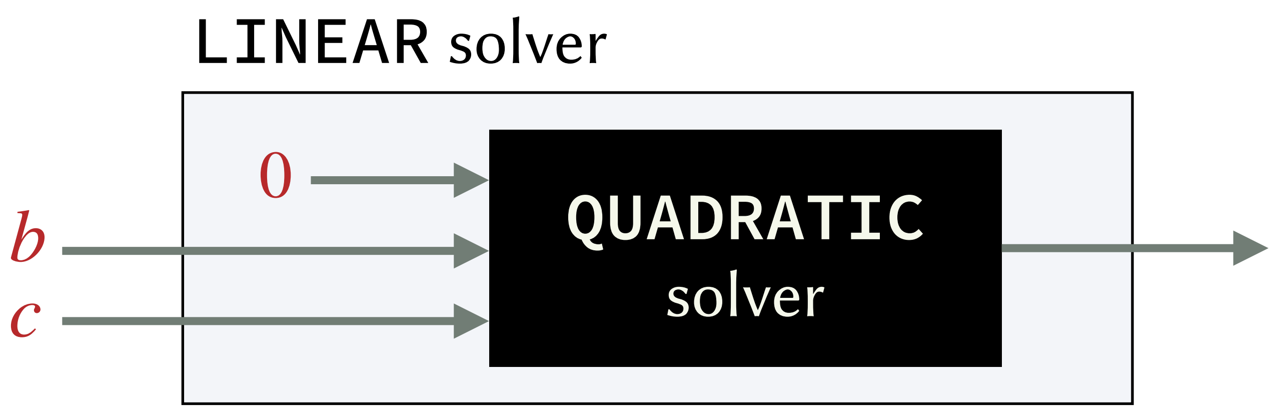
Given a solver for `QUADRATIC` can we solve `LINEAR`?

Reductions (Examples)

LINEAR
Given b and c, solve $bx + c = 0$

QUADRATIC
Given a, b and c, solve $ax^2 + bx + c = 0$

LINEAR reduces to QUADRATIC

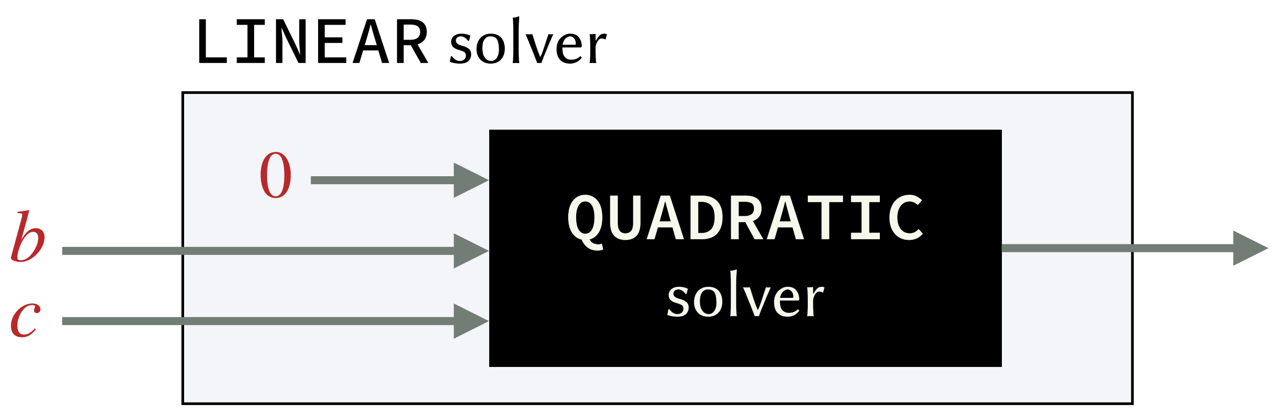


Reductions (Examples)

LINEAR
Given b and c, solve $bx + c = 0$

QUADRATIC
Given a, b and c, solve $ax^2 + bx + c = 0$

LINEAR reduces to QUADRATIC



SELECT
Given a list of elements, find the k^{th} largest element.

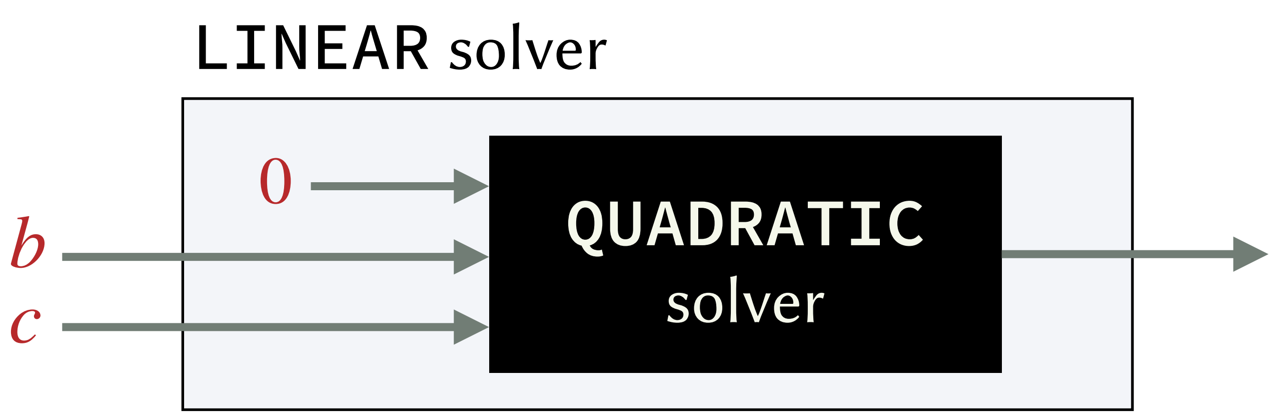
SORT
Given a list of elements, order the elements in non-decreasing order.

Reductions (Examples)

LINEAR
Given b and c, solve $bx + c = 0$

QUADRATIC
Given a, b and c, solve $ax^2 + bx + c = 0$

LINEAR reduces to QUADRATIC



SELECT
Given a list of elements, find the k^{th} largest element.

SORT
Given a list of elements, order the elements in non-decreasing order.

SELECT reduces to SORT

Use SORT to sort the elements and then report the element of rank k.

SORT reduces to SELECT

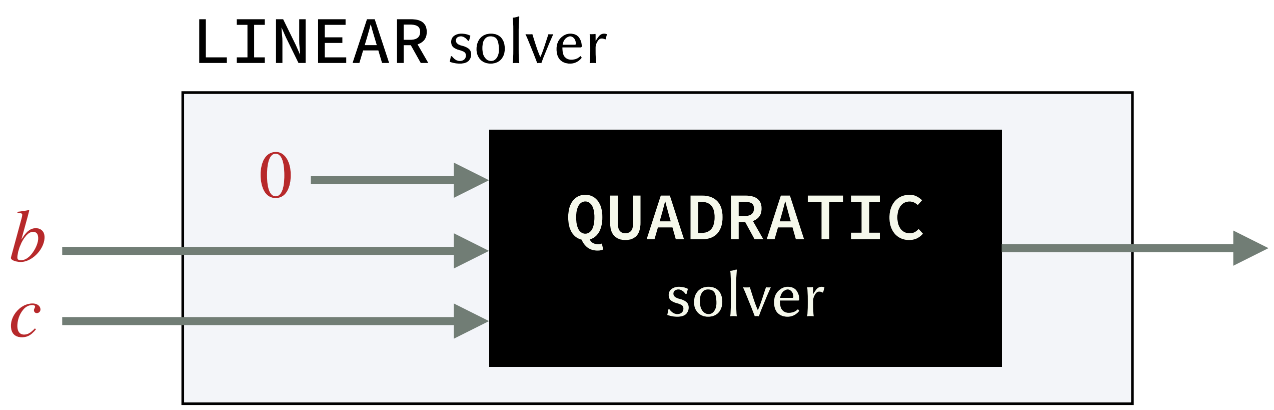
Sort the elements by repeatedly using SELECT to find the next largest element.

Reductions (Examples)

LINEAR
Given b and c, solve $bx + c = 0$

QUADRATIC
Given a, b and c, solve $ax^2 + bx + c = 0$

LINEAR reduces to QUADRATIC



SELECT
Given a list of elements, find the k^{th} largest element.

SORT
Given a list of elements, order the elements in non-decreasing order.

SELECT reduces to SORT

Use SORT to sort the elements and then report the element of rank k.

Running Time. $\underbrace{O(N \log N)}_{\text{SORT}} + \underbrace{O(1)}_{\text{reduction}}$

SORT reduces to SELECT

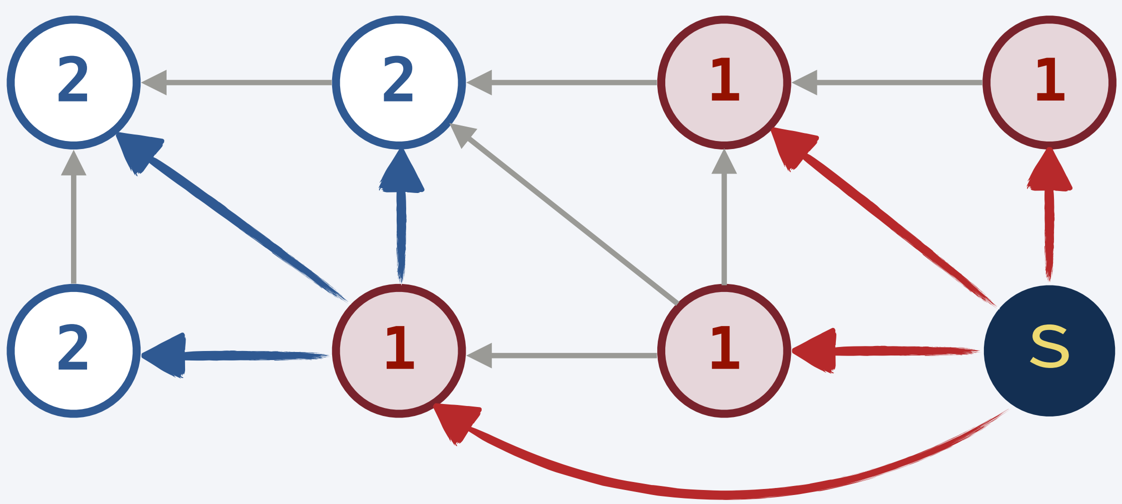
Sort the elements by repeatedly using SELECT to find the next largest element.

Running Time. $\underbrace{O(N)}_{\text{SELECT}} \times \underbrace{O(N)}_{\text{reduction}}$

Reductions (Examples)

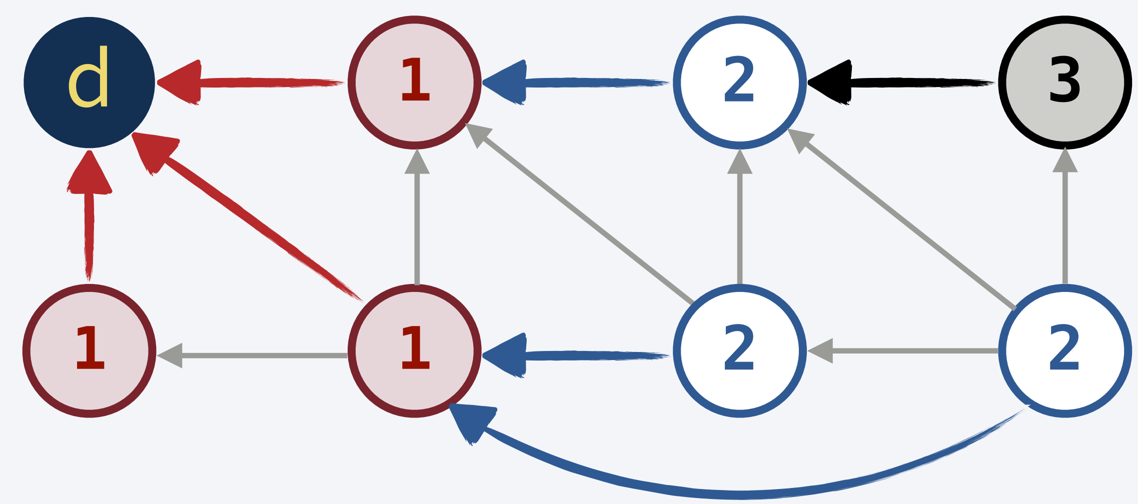
SSSP (Single Source Shortest Paths)

Given a graph G and a source vertex s, find the shortest path from s to every vertex in G.



SDSP (Single Destination Shortest Paths)

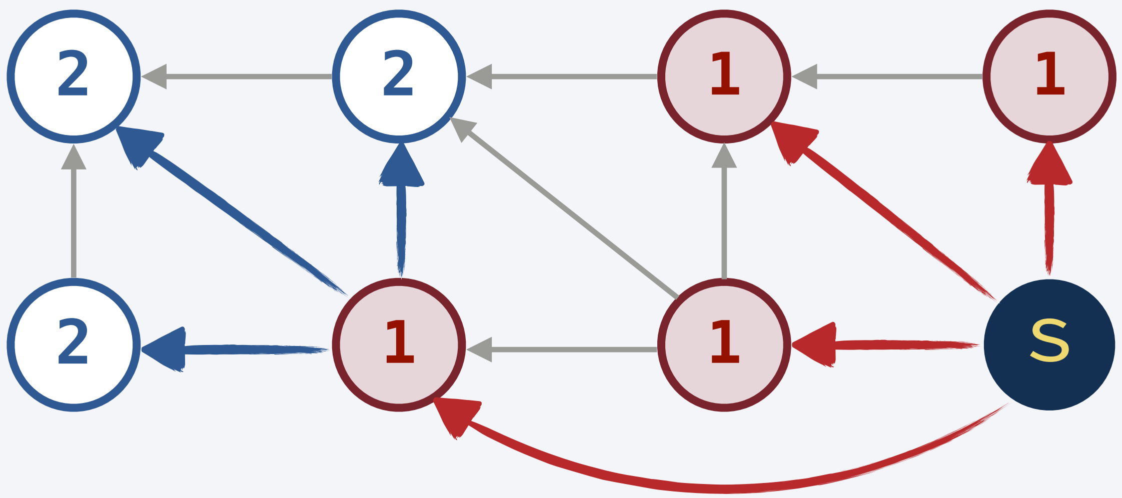
Given a graph G and a destination vertex d, find the shortest path from every vertex in G to d.



Reductions (Examples)

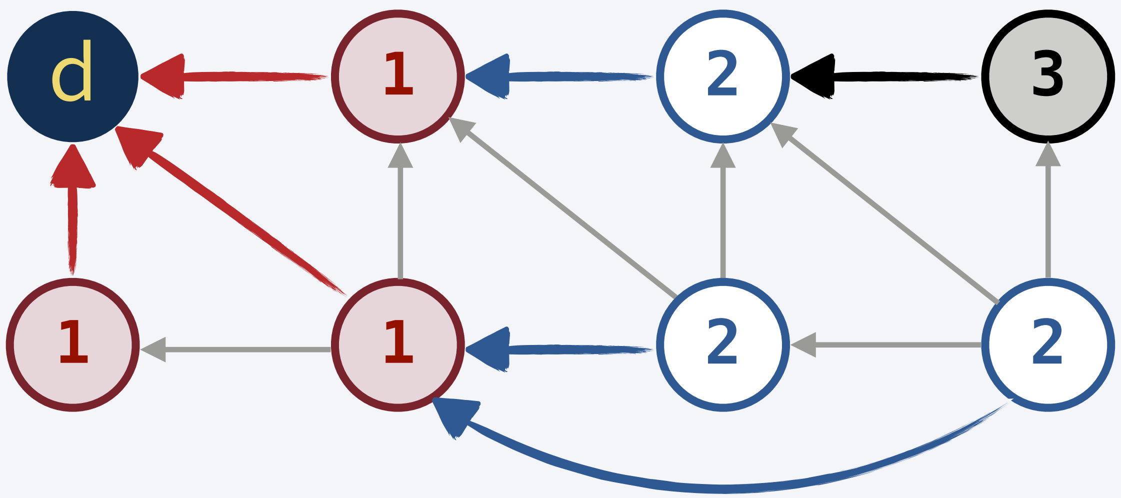
SSSP (Single Source Shortest Paths)

Given a graph G and a source vertex s, find the shortest path from s to every vertex in G.

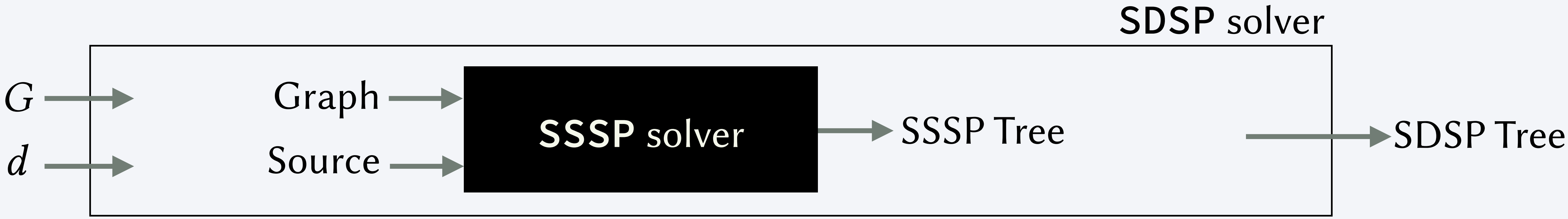


SDSP (Single Destination Shortest Paths)

Given a graph G and a destination vertex d, find the shortest path from every vertex in G to d.



SDSP reduces to SSSP



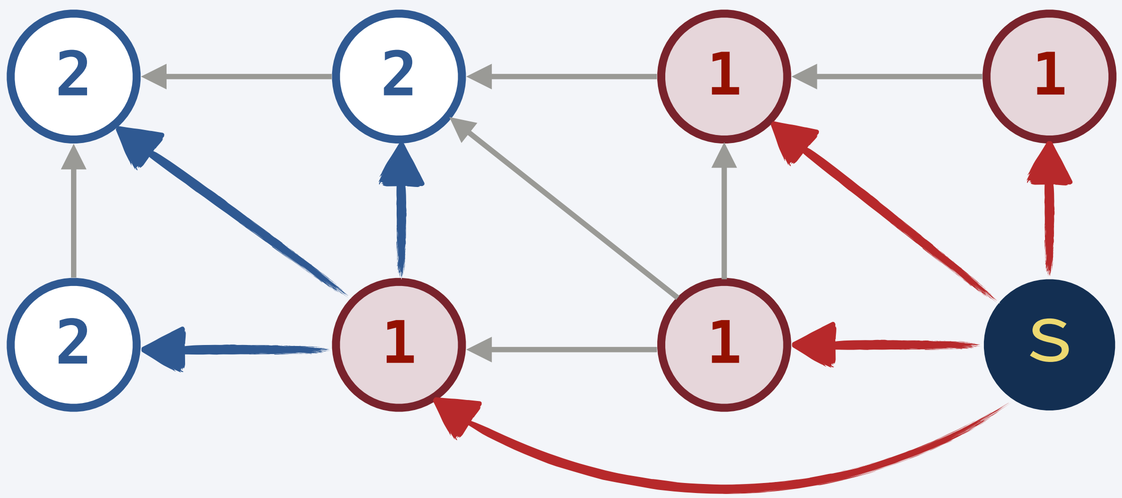
Reductions (Examples)

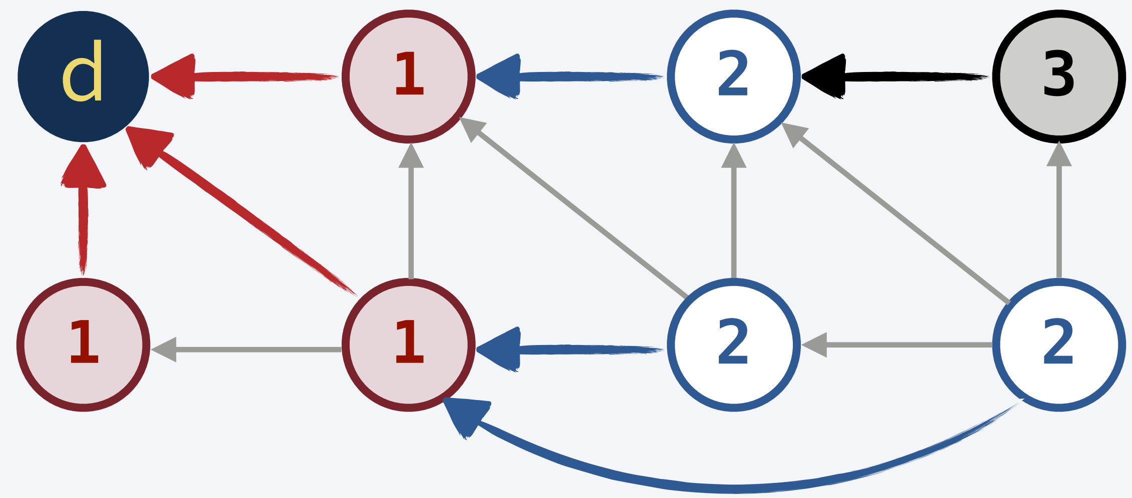
SSSP (Single Source Shortest Paths)

Given a graph G and a source vertex s, find the shortest path from s to every vertex in G.

SDSP (Single Destination Shortest Paths)

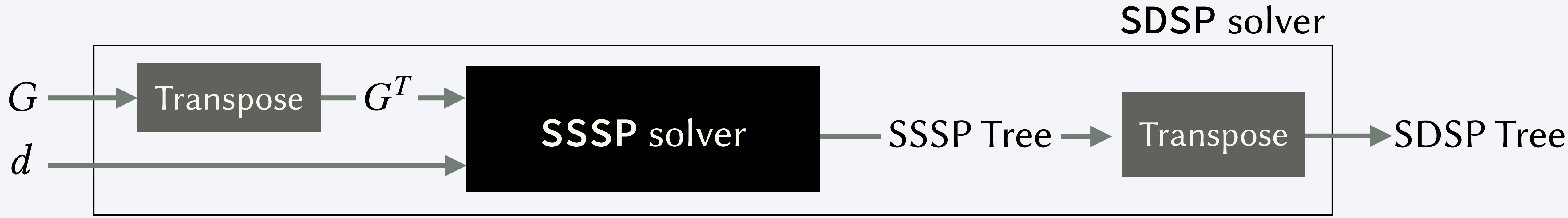
Given a graph G and a destination vertex d, find the shortest path from every vertex in G to d.





SDSP reduces to SSSP

- Create G^T, a transpose of G.
- Set s to d and run **SSSP** on G^T.
- Transpose the shortest paths tree.



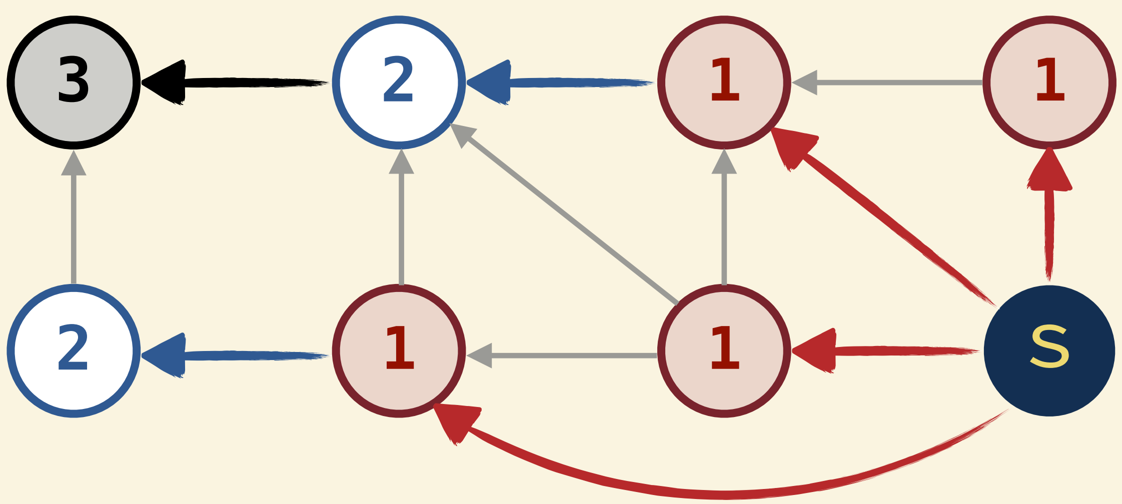
Reductions (Examples)

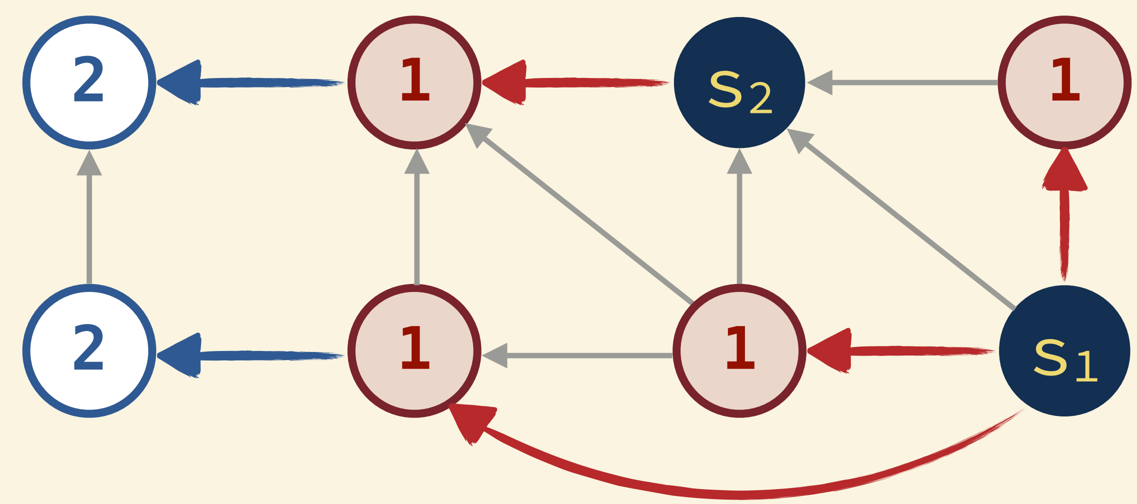
SSSP (Single Source Shortest Paths)

Given a graph G and a source vertex s, find the shortest path from s to every vertex in G.

MSSP (Multi-Source Shortest Paths)

Given a graph G and a set $S \subseteq G$ of source vertices, find the shortest path from S every vertex in G .

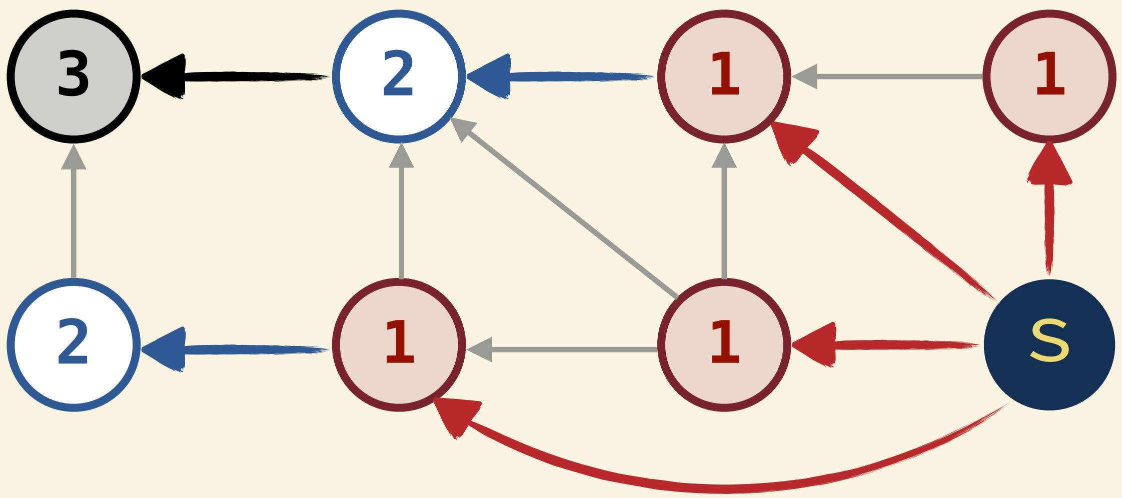




Reductions (Examples)

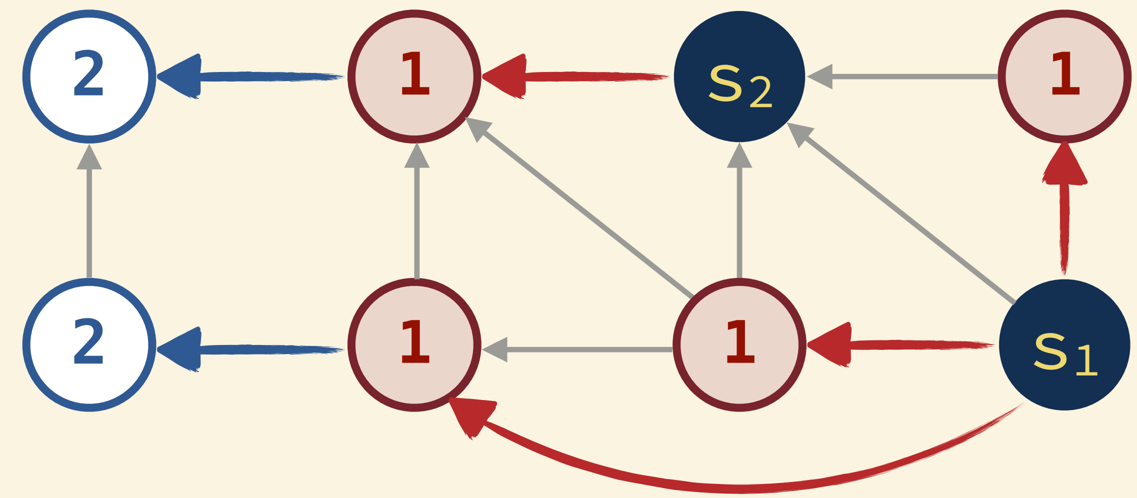
SSSP (Single Source Shortest Paths)

Given a graph G and a source vertex s, find the shortest path from s to every vertex in G.



MSSP (Multi-Source Shortest Paths)

Given a graph G and a set $S \subseteq G$ of source vertices, find the shortest path from S every vertex in G.

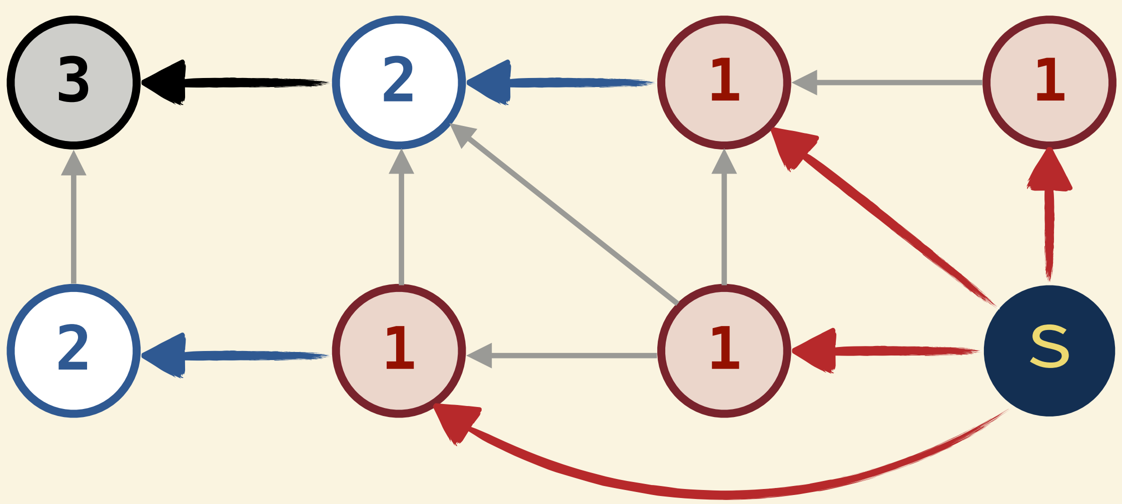


MSSP reduces to SSSP

Reductions (Examples)

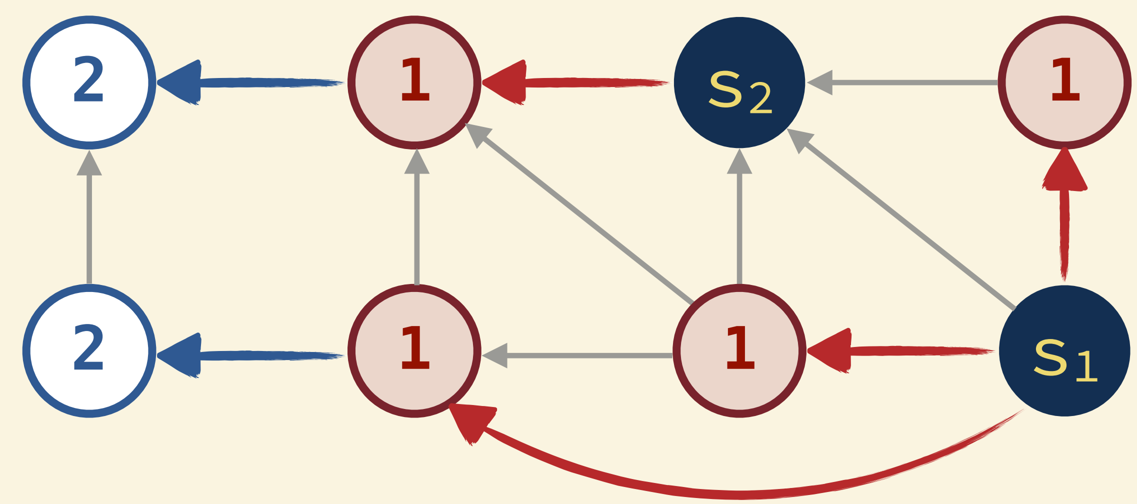
SSSP (Single Source Shortest Paths)

Given a graph G and a source vertex s, find the shortest path from s to every vertex in G.



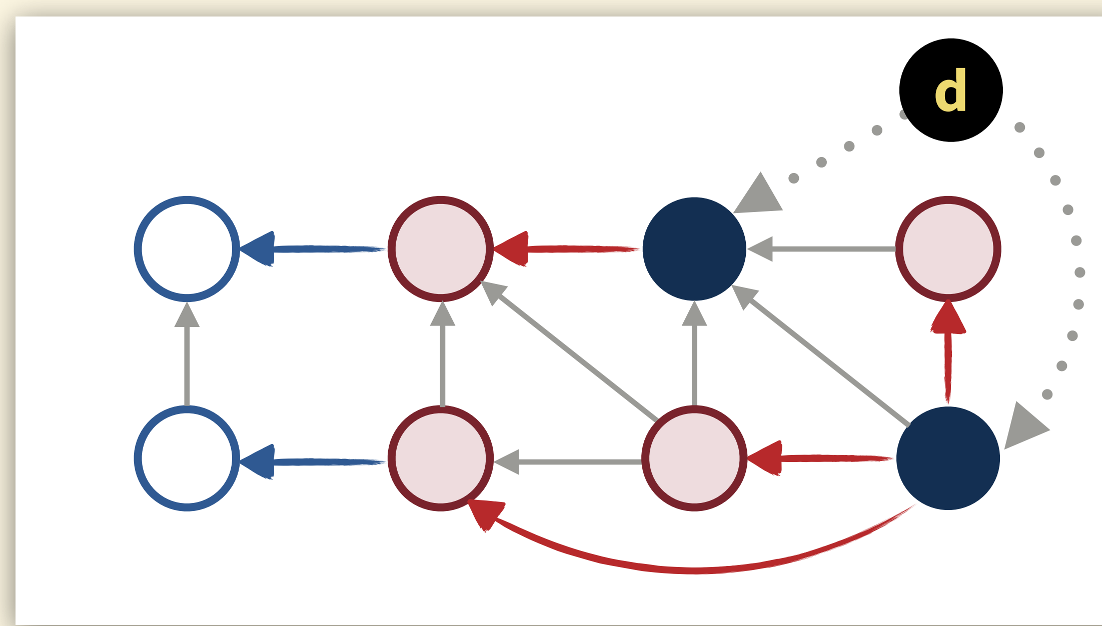
MSSP (Multi-Source Shortest Paths)

Given a graph G and a set $S \subseteq G$ of source vertices, find the shortest path from S every vertex in G .



MSSP reduces to SSSP

- Create G' by adding a vertex d to G. Add an edge of zero weight from d to every vertex $v \in S$
- Set d as the source and solve **SSSP** on G'.
- Remove from the resulting shortest paths tree the edges from d to S.



PITFALL

Saying that algorithm A reduces to algorithm B.

PITFALL

Saying that algorithm A reduces to algorithm B.

Example.

Selection Sort repeatedly selects the next minimum element in the array (using a *linear search* in the array) and places it in its position.

Heap Sort repeatedly selects the next minimum element in the array (using a *heap data structure*) and places it in its position.

It is **WRONG** to say that Selection Sort reduces to Heap Sort or that Heap Sort reduces to Selection Sort.

! *Reductions are between Problems NOT Algorithms*

Exercise # 1

Show that **3SUM-B** reduces to **3SUM-0** in linear time.

3SUM-0 Input: N integers: $x_1, x_2, x_3, \ldots, x_N$.
Output: **TRUE** iff there are three distinct indices i, j and k such that $x_i + x_j + x_k = 0$.

3SUM-B Input: An integer b and N integers: $x_1, x_2, x_3, \ldots, x_N$.
Output: **True** iff there are three distinct indices i, j and k such that: $x_i + x_j + x_k = b$.

Hint: The idea is in the preprocessing of the input!

Exercise # 1

Show that **3SUM-B** reduces to **3SUM-0** in linear time.

3SUM-0 Input: N integers: $x_1, x_2, x_3, \ldots, x_N$.
Output: **TRUE** iff there are three distinct indices i, j and k such that $x_i + x_j + x_k = 0$.

3SUM-B Input: An integer b and N integers: $x_1, x_2, x_3, \ldots, x_N$.
Output: **True** iff there are three distinct indices i, j and k such that: $x_i + x_j + x_k = b$.

Solution: Change every x in the input of **3SUM-B** to $3x - b$ and feed it to **3SUM-0**.

If $(3x_i - b) + (3x_j - b) + (3x_k - b) = 0$ Then:

$$3x_i + 3x_j + 3x_k = 3b$$

Divide by 3: $x_i + x_j + x_k = b$

Exercise # 2

Suppose there is a proof that no computer can solve problem X.

How can we prove that a problem Y is also impossible to solve?

A. Show that X reduces to Y.

B. Show that Y reduces to X.

C. Computers can solve any problem. It is only that *we* might not be clever enough to come up with an algorithm!

D. Reductions have nothing to do with this question.

Exercise # 2

Suppose there is a proof that no computer can solve problem X.

How can we prove that a problem Y is also impossible to solve?

- Show that X reduces to Y.

B. Show that Y reduces to X.

C. Computers can solve any problem. It is only that *we* might not be clever enough to come up with an algorithm!

D. Reductions have nothing to do with this question.

Exercise # 2

Suppose there is a proof that no computer can solve problem X.

How can we prove that a problem Y is also impossible to solve?

😁 Show that X reduces to Y.

B. Show that Y reduces to X.

C. Computers can solve any problem. It is only that *we* might not be clever enough to come up with an algorithm!

D. Reductions have nothing to do with this question.

X reduces to Y

We can solve X using Y.

If Y is solvable:
X is also solvable (contradiction!)

Y reduces to X

We can solve Y using X.

While X is unsolvable, there might be another way for solving Y not using X.

Undecidability

WIKIPEDIA
The Free Encyclopedia

Undecidable problem

13 languages

Article Talk Tools

From Wikipedia, the free encyclopedia

In computability theory and computational complexity theory, an **undecidable problem** is a decision problem for which it is proved to be impossible to construct an algorithm that always leads to a correct yes-or-no answer.[1] The halting problem is an example: it can be proven that there is no algorithm that correctly determines whether an arbitrary program eventually halts when run.[2]

Reductions (Showing Undecidability)

HALT

Given a program P and an input d, does $P(d)$ terminate?
(i.e. will not enter an infinite loop)

DEAD-CODE

Given a program P, an input d, and a line number x, will $P(d)$ execute line x?

! **HALT** is known to be undecidable.

Reductions (Showing Undecidability)

HALT

Given a program P and an input d, does $P(d)$ terminate?
(i.e. will not enter an infinite loop)

DEAD-CODE

Given a program P, an input d, and a line number x, will $P(d)$ execute line x?

! **HALT** is known to be undecidable.

🤔 How can we show that **DEAD-CODE** is also undecidable?

Reductions (Showing Undecidability)

HALT

Given a program P and an input d, does $P(d)$ terminate?
(i.e. will not enter an infinite loop)

DEAD-CODE

Given a program P, an input d, and a line number x, will $P(d)$ execute line x?

! **HALT** is known to be undecidable.

How can we show that **DEAD-CODE** is also undecidable?

Answer. Show that `HALT` reduces to `DEAD-CODE`.

Reductions (Showing Undecidability)

HALT

Given a program P and an input d, does $P(d)$ terminate?
(i.e. will not enter an infinite loop)

DEAD-CODE

Given a program P, an input d, and a line number x, will $P(d)$ execute line x?

! **HALT** is known to be undecidable.

How can we show that **DEAD-CODE** is also undecidable?

Answer. Show that HALT reduces to DEAD-CODE.

HALT reduces to DEAD-CODE

Since HALT can be solved using DEAD-CODE and HALT is known to be impossible to solve, DEAD-CODE must also be impossible to solve.

Reductions (Showing Undecidability)

HALT

Given a program P and an input d, does $P(d)$ terminate?
(i.e. will not enter an infinite loop)

DEAD-CODE

Given a program P, an input d, and a line number x, will $P(d)$ execute line x?

! **HALT** is known to be undecidable.

How can we show that **DEAD-CODE** is also undecidable?

Answer. Show that `HALT` reduces to `DEAD-CODE`.

HALT reduces to DEAD-CODE

- Assume that line K is at the end of program P.
  Replace every `halt` instruction in P with `goto K`.
- Feed P, d, and K into a **DEAD-CODE** solver. If the result is **TRUE**, then $P(d)$ halts.
  If the result is **FALSE**, then $P(d)$ does not halt.

Since HALT can be solved using DEAD-CODE and HALT is known to be impossible to solve, DEAD-CODE must also be impossible to solve.

Reductions (Showing Undecidability)

TOTALITY

Does a given program P terminate on all possible inputs?
(never enters an infinite loop!)

EQUIVALENCE

Given two programs P_1 and P_2. Do these two programs produce the same output for every input?
(i.e. are they equivalent?)

! **TOTALITY** is known to be undecidable.

Reductions (Showing Undecidability)

TOTALITY

Does a given program P terminate on all possible inputs?
(never enters an infinite loop!)

EQUIVALENCE

Given two programs P_1 and P_2. Do these two programs produce the same output for every input?
(i.e. are they equivalent?)

! **TOTALITY** is known to be undecidable.

How can we show that **EQUIVALENCE** is also undecidable?

Reductions (Showing Undecidability)

TOTALITY

Does a given program P terminate on all possible inputs?
(never enters an infinite loop!)

EQUIVALENCE

Given two programs P_1 and P_2. Do these two programs produce the same output for every input?
(i.e. are they equivalent?)

! **TOTALITY** is known to be undecidable.

How can we show that **EQUIVALENCE** is also undecidable?

Answer. Show that TOTALITY reduces to EQUIVALENCE.

TOTALITY reduces to EQUIVALENCE

Since TOTALITY can be solved using EQUIVALENCE and TOTALITY is known to be impossible to solve, EQUIVALENCE must also be impossible to solve.

Reductions (Showing Undecidability)

TOTALITY

Does a given program P terminate on all possible inputs?
(never enters an infinite loop!)

EQUIVALENCE

Given two programs P_1 and P_2. Do these two programs produce the same output for every input?
(i.e. are they equivalent?)

! **TOTALITY** is known to be undecidable.

🤔 How can we show that **EQUIVALENCE** is also undecidable?

Answer. Show that TOTALITY reduces to EQUIVALENCE.

TOTALITY reduces to EQUIVALENCE

- Create P_1 as a copy of P, except that it outputs **TRUE** instead of its original output.

Since TOTALITY can be solved using EQUIVALENCE and TOTALITY is known to be impossible to solve, EQUIVALENCE must also be impossible to solve.

Reductions (Showing Undecidability)

TOTALITY

Does a given program P terminate on all possible inputs? (never enters an infinite loop!)

EQUIVALENCE

Given two programs P_1 and P_2. Do these two programs produce the same output for every input? (i.e. are they equivalent?)

! **TOTALITY** is known to be undecidable.

How can we show that **EQUIVALENCE** is also undecidable?

Answer. Show that TOTALITY reduces to EQUIVALENCE.

TOTALITY reduces to EQUIVALENCE

- Create P_1 as a copy of P, except that it outputs **TRUE** instead of its original output.
- Create a program P_2 that outputs **TRUE** and does nothing else.

Since TOTALITY can be solved using EQUIVALENCE and TOTALITY is known to be impossible to solve, EQUIVALENCE must also be impossible to solve.

Reductions (Showing Undecidability)

TOTALITY

Does a given program P terminate on all possible inputs?
(never enters an infinite loop!)

EQUIVALENCE

Given two programs P_1 and P_2. Do these two programs produce the same output for every input?
(i.e. are they equivalent?)

! **TOTALITY** is known to be undecidable.

🤔 How can we show that **EQUIVALENCE** is also undecidable?

Answer. Show that `TOTALITY` reduces to `EQUIVALENCE`.

`TOTALITY` reduces to `EQUIVALENCE`

- Create P_1 as a copy of P, except that it outputs **TRUE** instead of its original output.
- Create a program P_2 that outputs **TRUE** and does nothing else.
- Use `EQUIVALENCE` to check if P_1 and P_2 are equivalent.
  If they are equivalent, P terminates on all input. If they are not, the only possibility is that P does not terminate on some input (since the output of P_1 and P_2 is always the same).

Since `TOTALITY` can be solved using `EQUIVALENCE` and `TOTALITY` is known to be impossible to solve, `EQUIVALENCE` must also be impossible to solve.

PITFALL

Confusing the direction of the reduction.

PITFALL

Confusing the direction of the reduction.

Remember. X reduces to Y (denoted as $X \leqslant Y$) means that X can be solved using a solver for Y.

Implication. If X reduces to Y with an easy transformation, then X is not harder than Y.

Example. **DEAD-CODE** $\leqslant$ **HALT** means that **DEAD-CODE** is not harder to solve than **HALT**. This is not interesting because we already know that **HALT** is impossible to solve.

Example. **HALT** $\leqslant$ **DEAD-CODE** means that **HALT** is not harder to solve than **DEAD-CODE**. Since **HALT** is impossible to solve, **DEAD-CODE** must also be impossible (because **HALT** is not harder!)

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Sorting a list of n elements using comparisons only.

- A trivial **upper bound**. $O(n!)$
  We don't need more time than what is needed to check all the permutations.

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Sorting a list of n elements using comparisons only.

- A trivial **upper bound**. $O(n!)$
  We don't need more time than what is needed to check all the permutations.
- Another trivial **upper bound**. $O(n^2)$
  We don't need more time than what naive sorting algorithms like Bubble Sort need.

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Sorting a list of n elements using comparisons only.

- A trivial **upper bound**. $O(n!)$
  We don't need more time than what is needed to check all the permutations.
- Another trivial **upper bound**. $O(n^2)$
  We don't need more time than what naive sorting algorithms like Bubble Sort need.
- A better **upper bound**. $O(n \log n)$
  Merge Sort and Heap Sort perform $\Theta(n \log n)$ comparisons.

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Sorting a list of n elements using comparisons only.

- A trivial **upper bound**. $O(n!)$
  We don't need more time than what is needed to check all the permutations.
- Another trivial **upper bound**. $O(n^2)$
  We don't need more time than what naive sorting algorithms like Bubble Sort need.
- A better **upper bound**. $O(n \log n)$
  Merge Sort and Heap Sort perform $\Theta(n \log n)$ comparisons.
- A trivial **Lower Bound**. $\Omega(n)$
  We can't sort all the elements unless we see all the elements!

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Sorting a list of n elements using comparisons only.

- A trivial **upper bound.** $O(n!)$
  We don't need more time than what is needed to check all the permutations.
- Another trivial **upper bound.** $O(n^2)$
  We don't need more time than what naive sorting algorithms like Bubble Sort need.
- A better **upper bound.** $O(n \log n)$
  Merge Sort and Heap Sort perform $\Theta(n \log n)$ comparisons.
- A trivial **Lower Bound.** $\Omega(n)$
  We can't sort all the elements unless we see all the elements!
- A better **lower Bound.** $\Omega(n \log n)$
  There is a famous proof for that!

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Multiplying two integers of length n digits each.

- A trivial **upper bound.** $O(n^2)$
  We can use Long Multiplication.
- A better **upper bound.** $O(n^{1.585})$
  Karatsuba's Algorithm runs in $\Theta(n^{\log_2 3} \approx n^{1.5849})$ time.

Upper & Lower Bounds

Upper Bound. An upper bound T for a problem shows that the problem can be solved in $O(T)$.

Lower Bound. A lower bound T for a problem means that there is no hope of finding an algorithm that runs in time better than $\Omega(T)$ in the worst case.

Example. Multiplying two integers of length n digits each.

- A trivial **upper bound**. $O(n^2)$
  We can use Long Multiplication.
- A better **upper bound**. $O(n^{1.585})$
  Karatsuba's Algorithm runs in $\Theta(n^{\log_2 3} \approx n^{1.5849})$ time.

- A trivial **Lower Bound**. $\Omega(n)$
  We can't multiply the two numbers unless we see all the digits!
- A **conjectured** better **lower Bound**. $\Omega(n \log n)$
  There is no proof for that yet!

Reductions (Lower Bounds)

PAIR

Given lists L_1 and L_2 of size N, pair the min in L_1 with the min in L_2, the next min in L_1 with the next min in L_2, etc.

SORT

Given a list of elements, sort them in non-decreasing order.

Example.

```
L1   = [13, 7, 3, 1, 11, 2]
L2   = [2, 8, 6, 4, 10, 0]
PAIR = [1-0, 2-2, 3-4, 7-6, 11-8, 13-10]
```

Reductions (Lower Bounds)

PAIR

Given lists L_1 and L_2 of size N, pair the min in L_1 with the min in L_2, the next min in L_1 with the next min in L_2, etc.

SORT

Given a list of elements, sort them in non-decreasing order.

Example.

```
L1   = [13, 7, 3, 1, 11, 2]
L2   = [2, 8, 6, 4, 10, 0]
PAIR = [1-0, 2-2, 3-4, 7-6, 11-8, 13-10]
```

PAIR reduces to **SORT**

- Use **SORT** to sort L_1 and L_2.
- Pair L_1`[0]` with L_2`[0]`,
  L_1`[1]` with L_2`[1]`,
  etc.

Reductions (Lower Bounds)

PAIR

Given lists L_1 and L_2 of size N, pair the min in L_1 with the min in L_2, the next min in L_1 with the next min in L_2, etc.

SORT

Given a list of elements, sort them in non-decreasing order.

Example.

```
L1   = [13, 7, 3, 1, 11, 2]
L2   = [2, 8, 6, 4, 10, 0]
PAIR = [1-0, 2-2, 3-4, 7-6, 11-8, 13-10]
```

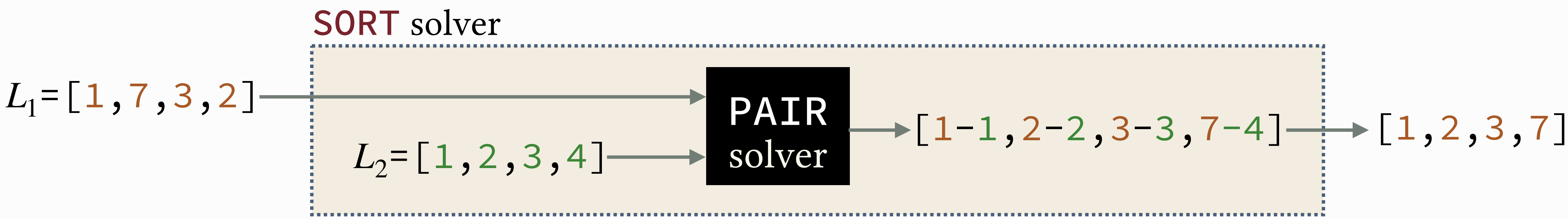
PAIR reduces to **SORT**

- Use **SORT** to sort L_1 and L_2.
- Pair L_1`[0]` with L_2`[0]`,
  L_1`[1]` with L_2`[1]`,
  etc.

SORT reduces to **PAIR**

- Let L_1 be the list to be sorted.
- Create L_2 containing the numbers 1 to N.
- Extract the sorted version of L_1 from the result of applying **PAIR** on L_1 and L_2.

Reductions (Lower Bounds)



Implication.

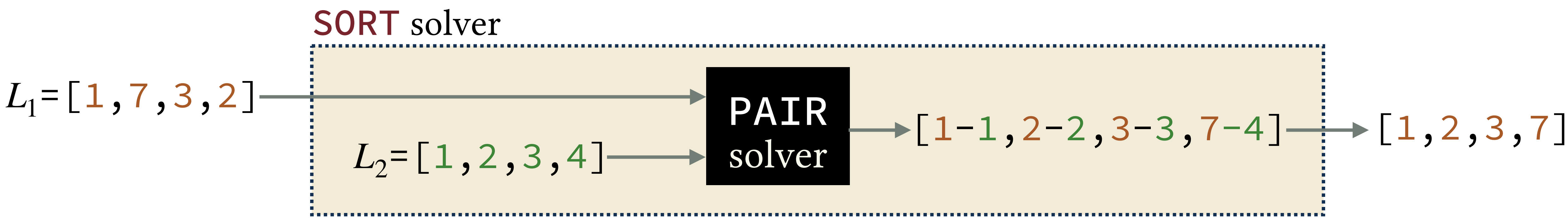
PAIR reduces to SORT

- Use SORT to sort L_1 and L_2.
- Pair L_1`[0]` with L_2`[0]`,
  L_1`[1]` with L_2`[1]`,
  etc.

SORT reduces to PAIR

- Let L_1 be the list to be sorted.
- Create L_2 containing the numbers 1 to N.
- Extract the sorted version of L_1 from the result of applying **PAIR** on L_1 and L_2.

Reductions (Lower Bounds)



Implication.

- We already know that any comparison based algorithm for **SORT** performs $\Omega(N \log N)$ compares in the worst case.

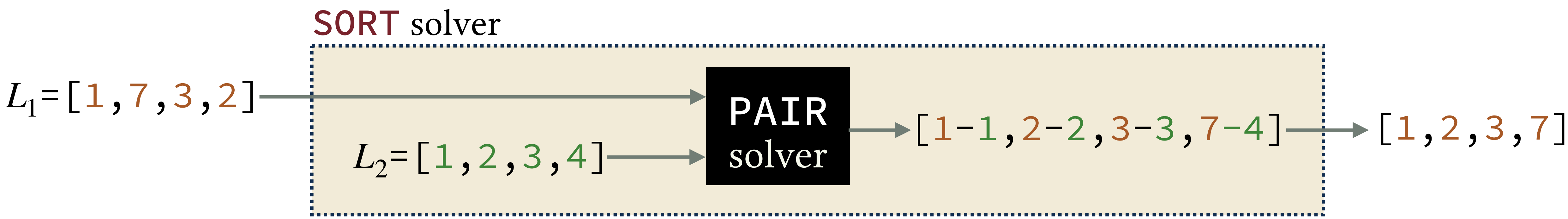
PAIR reduces to SORT

- Use SORT to sort L_1 and L_2.
- Pair L_1`[0]` with L_2`[0]`,
  L_1`[1]` with L_2`[1]`,
  etc.

SORT reduces to PAIR

- Let L_1 be the list to be sorted.
- Create L_2 containing the numbers 1 to N.
- Extract the sorted version of L_1 from the result of applying **PAIR** on L_1 and L_2.

Reductions (Lower Bounds)



Implication.

- We already know that any comparison based algorithm for **SORT** performs $\Omega(N \log N)$ compares in the worst case.
- The reduction from **SORT** to **PAIR** requires only $\Theta(N)$ amount of work (creating L_2 and extracting the result)

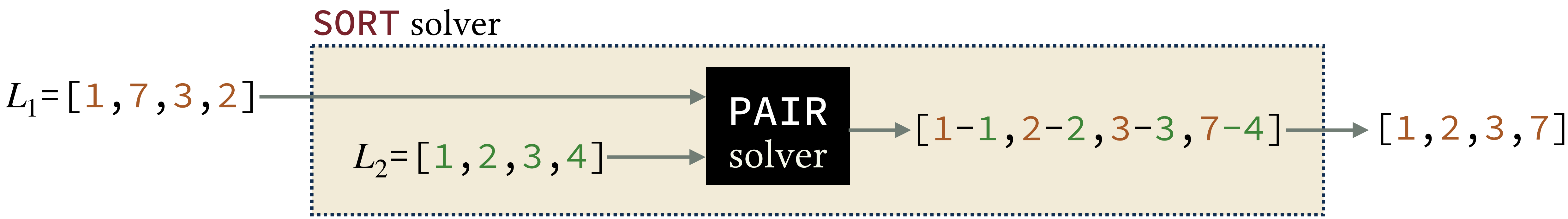
PAIR reduces to SORT

- Use SORT to sort L_1 and L_2.
- Pair L_1[0] with L_2[0],
  L_1[1] with L_2[1],
  etc.

SORT reduces to PAIR

- Let L_1 be the list to be sorted.
- Create L_2 containing the numbers 1 to N.
- Extract the sorted version of L_1 from the result of applying **PAIR** on L_1 and L_2.

Reductions (Lower Bounds)



Implication.

- We already know that any comparison based algorithm for **SORT** performs $\Omega(N \log N)$ compares in the worst case.
- The reduction from **SORT** to **PAIR** requires only $\Theta(N)$ amount of work (creating L_2 and extracting the result)
- **PAIR** must require $\Omega(N \log N)$ compares in the worst case.
  Otherwise, the $\Omega(N \log N)$ lower bound for **SORT** is not correct (contradiction!)

PAIR reduces to SORT

- Use SORT to sort L_1 and L_2.
- Pair L_1`[0]` with L_2`[0]`,
  L_1`[1]` with L_2`[1]`,
  etc.

SORT reduces to PAIR

- Let L_1 be the list to be sorted.
- Create L_2 containing the numbers 1 to N.
- Extract the sorted version of L_1 from the result of applying **PAIR** on L_1 and L_2.

Exercise # 3

LONGEST-PATH

Given an undirected graph G and two distinct vertices s and t, find the longest simple path (no repeated vertices) between s and t.

LONGEST-CYCLE

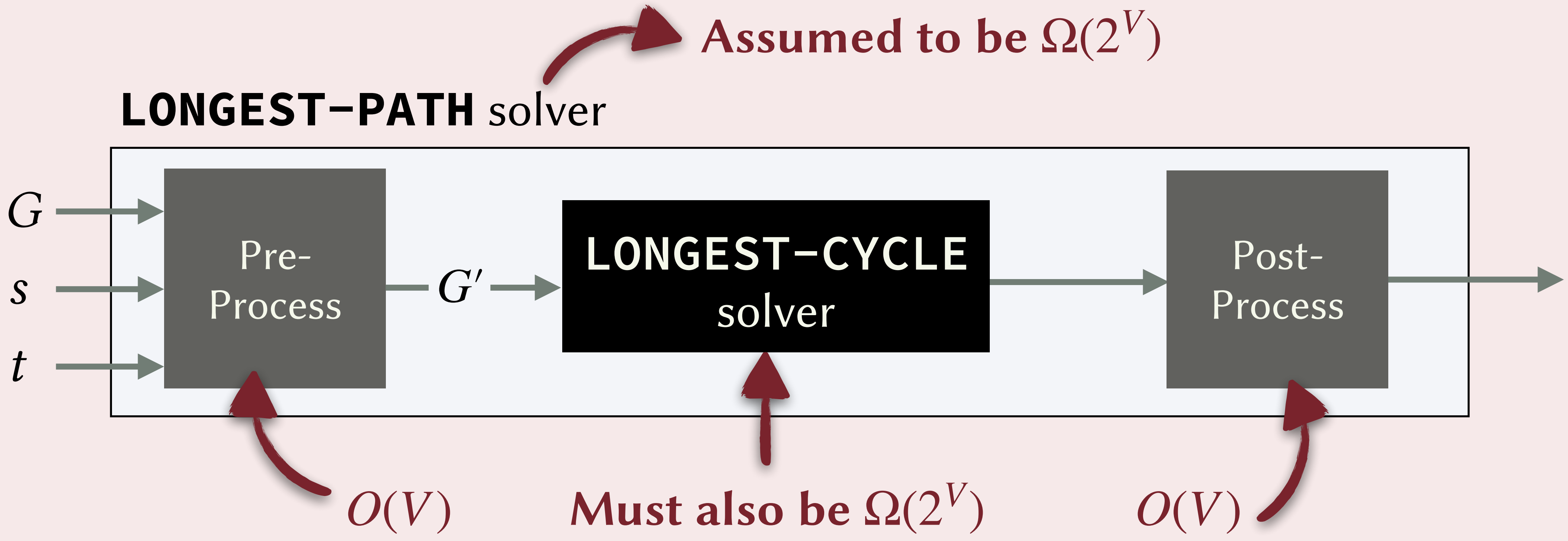
Given an undirected graph G, find the longest simple cycle (no repeated vertices or edges except the first and last vertex).

Assume that there is a proof that the lower bound for the **LONGEST-PATH** problem is $\Omega(2^V)$, where V is the number vertices in the graph*.

Use a *reduction* to prove that the lower bound for **LONGEST-CYCLE** is also $\Omega(2^V)$.

* *Note that this is just an assumption and that no such proof currently exists.*

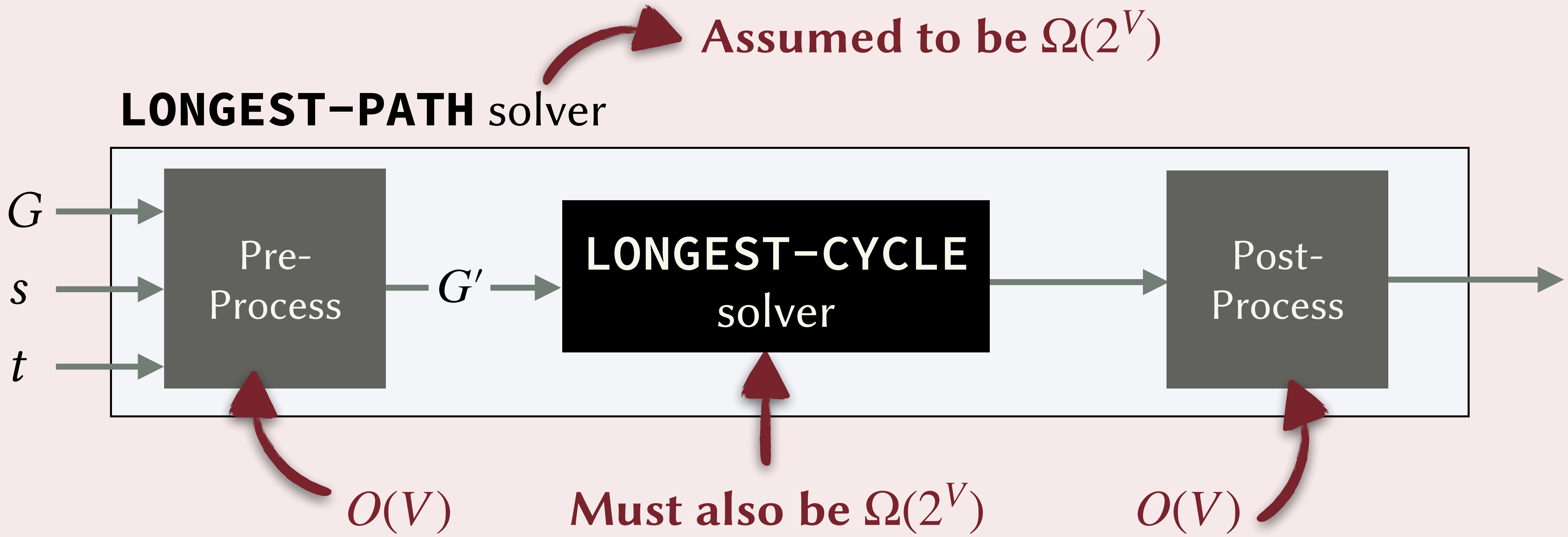
Exercise # 3 (solution)



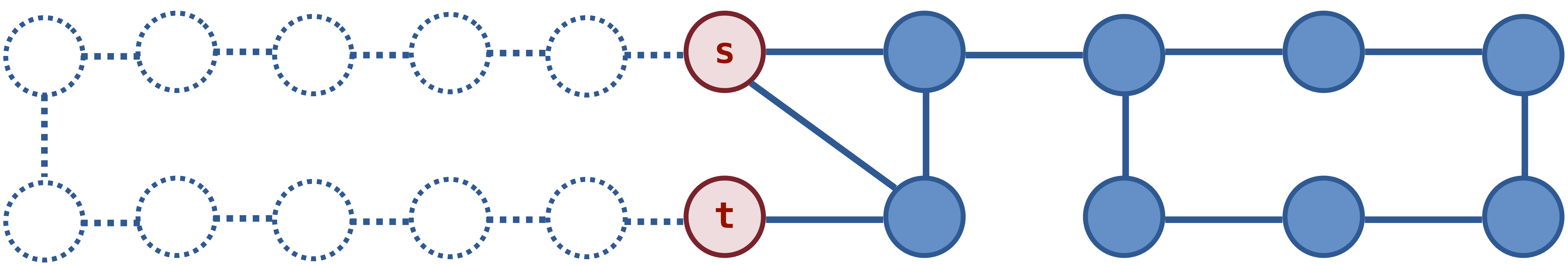
LONGEST-PATH reduces to **LONGEST-CYCLE**

* *Note that this is just an assumption and that no such proof currently exists.*

Exercise # 3 (solution continued)



LONGEST–PATH reduces to **LONGEST–CYCLE**



Add a cycle from s to t that is has $> V$ vertices. Finding the longest cycle in the modified graph will lead to newly added cycle + the longest path from s to t.

* *Note that this is just an assumption and that no such proof currently exists.*

Exercise # 4

`MIN`

Given a list of N elements, find the minimum element (using comparisons only)

`SORT`

Given a list of N elements, sort them in non-decreasing order (using comparisons only)

Use a *reduction* to prove that $\Omega(\log N)$ is a lower bound for **`MIN`**.

Exercise # 4

MIN

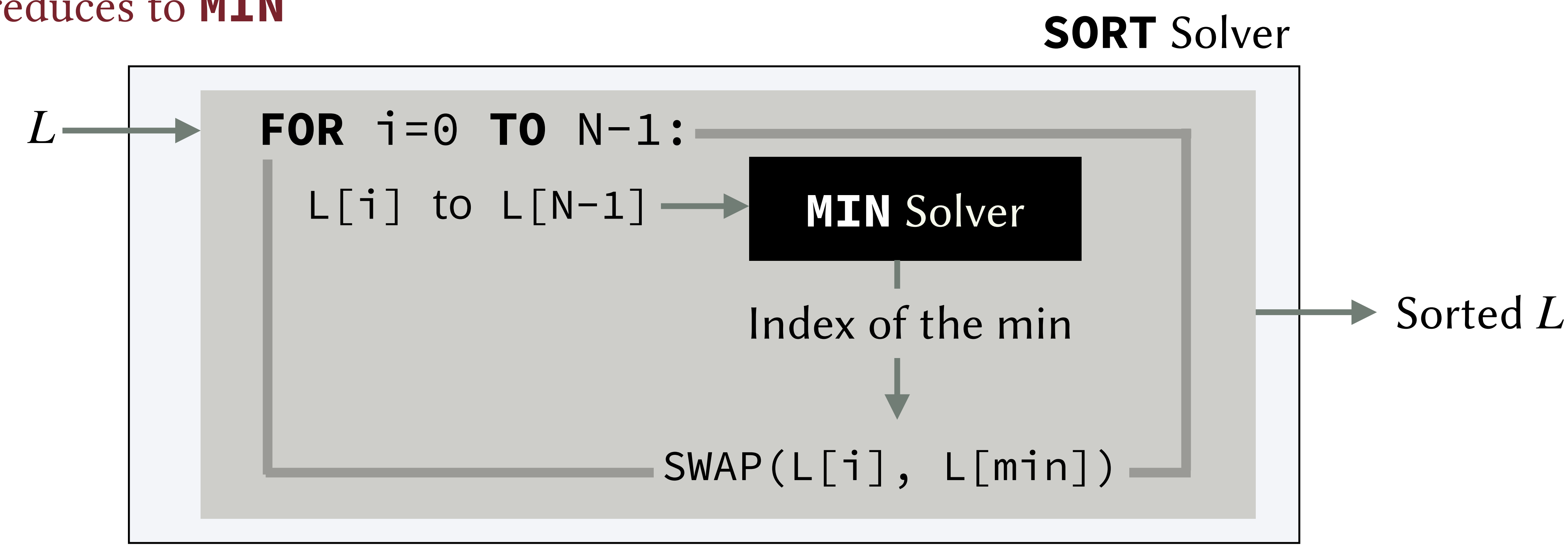
Given a list of N elements, find the minimum element (using comparisons only)

SORT

Given a list of N elements, sort them in non-decreasing order (using comparisons only)

Use a *reduction* to prove that $\Omega(\log N)$ is a lower bound for **MIN**.

SORT reduces to **MIN**



Sorting time = $N \times$ (time for **MIN** Solver + swapping time)

If the complexity of **MIN** Solver is less than $\log N$, the sorting complexity becomes less than $N \log N$, which is impossible (the sorting lower bound is $\Omega(N \log N)$).