

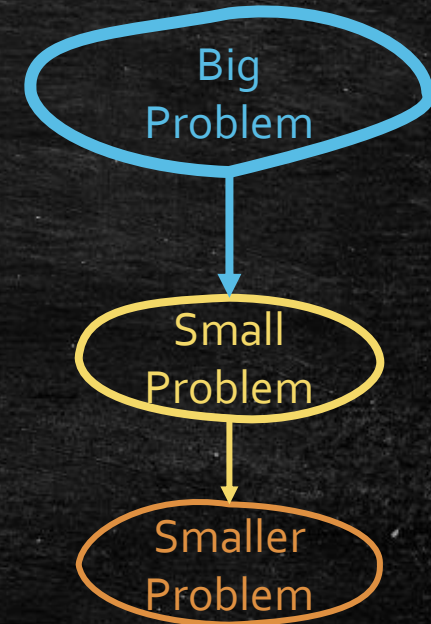
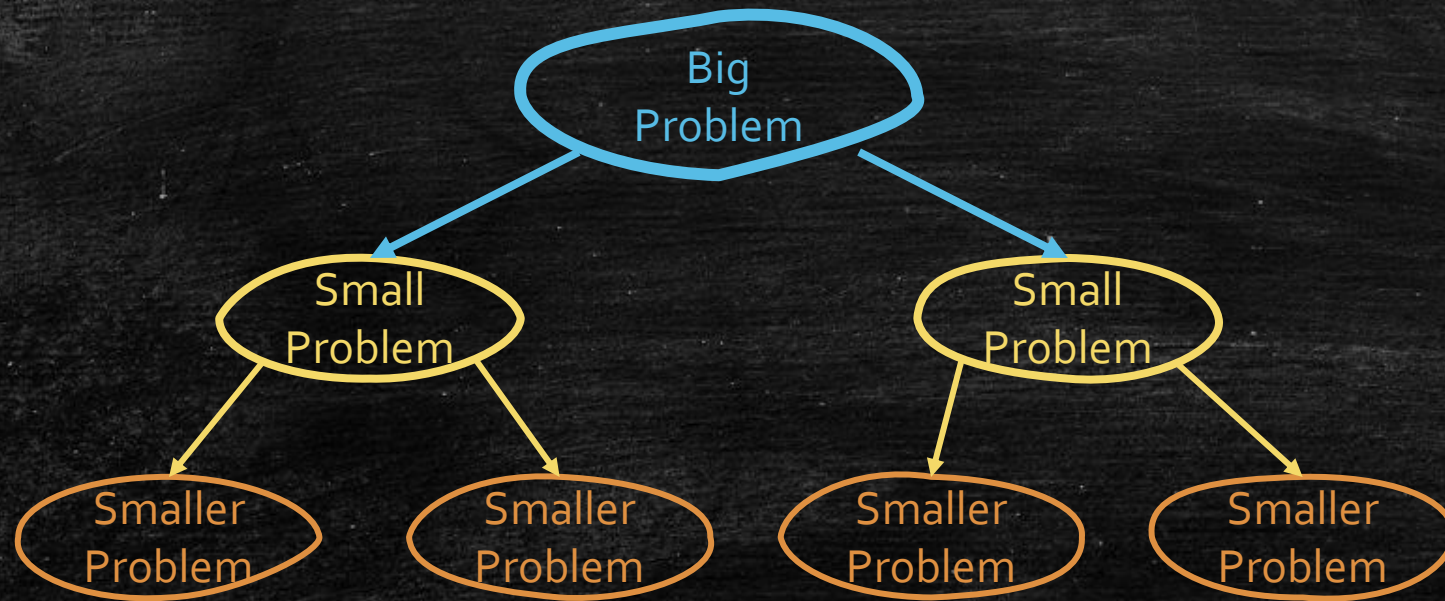
Dynamic Programming

Recall Divide and Conquer vs. Greedy

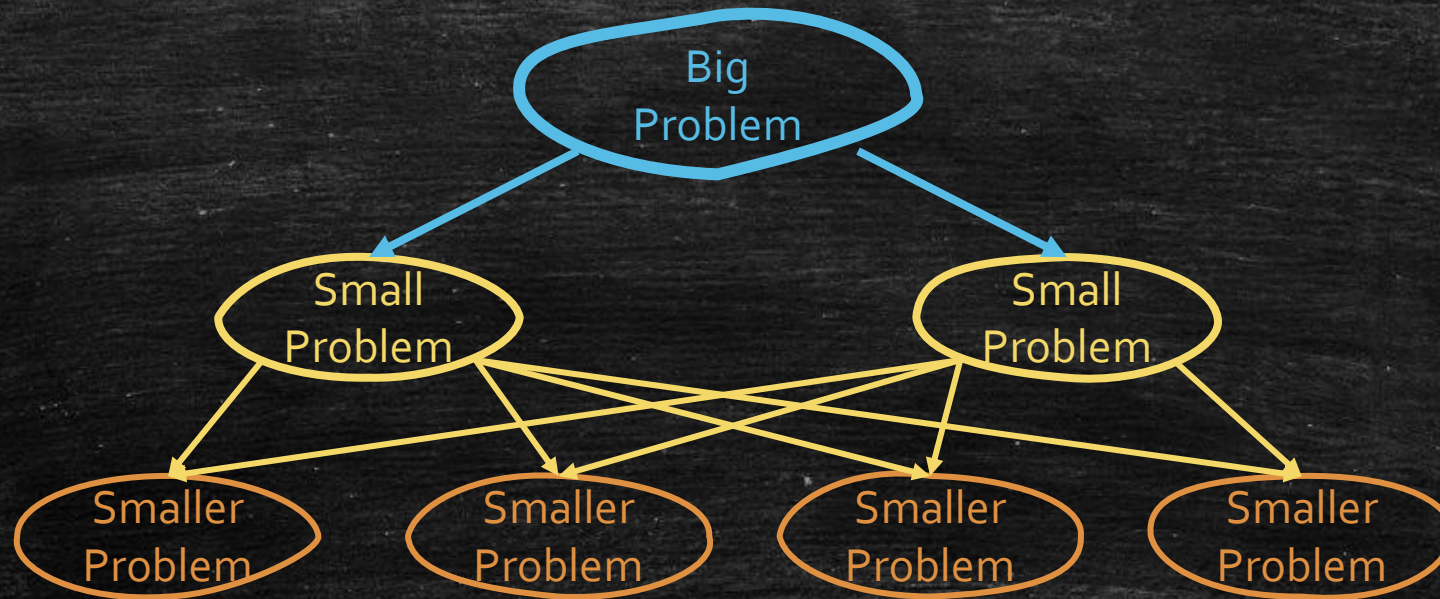
Divide and Conquer

vs.

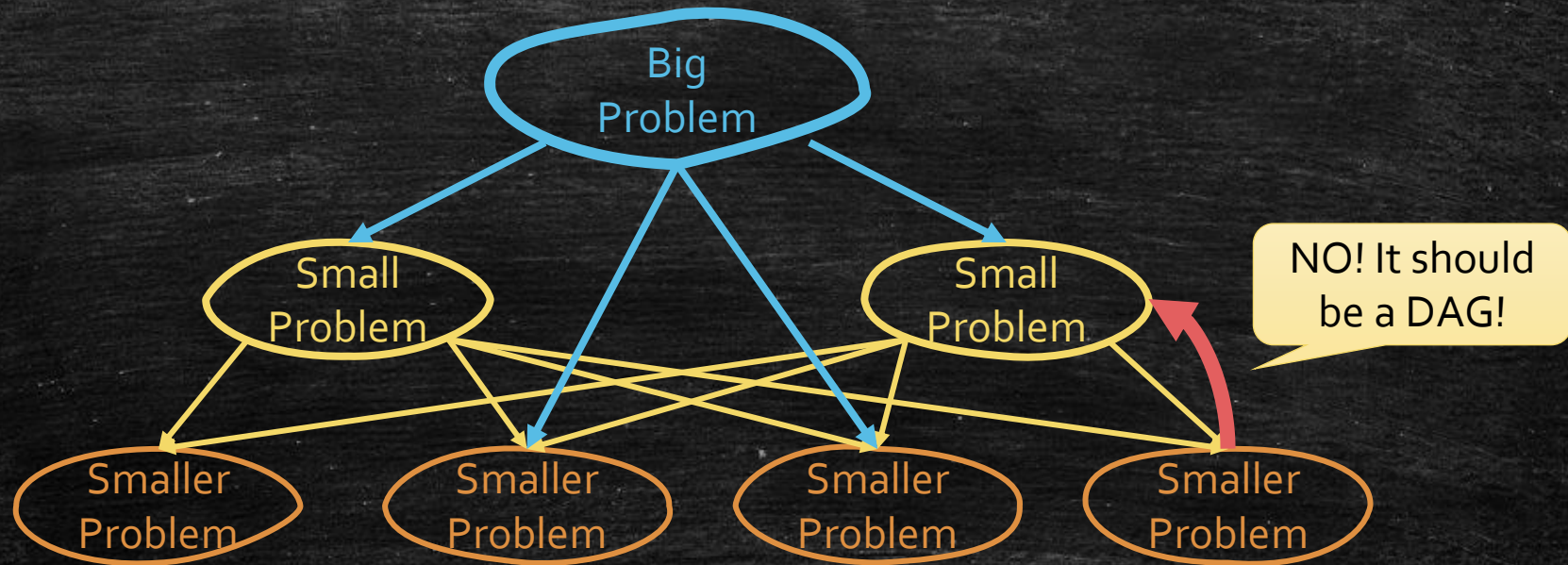
Greedy



Dynamic Programming vs. Divide and Conquer



Dynamic Programming



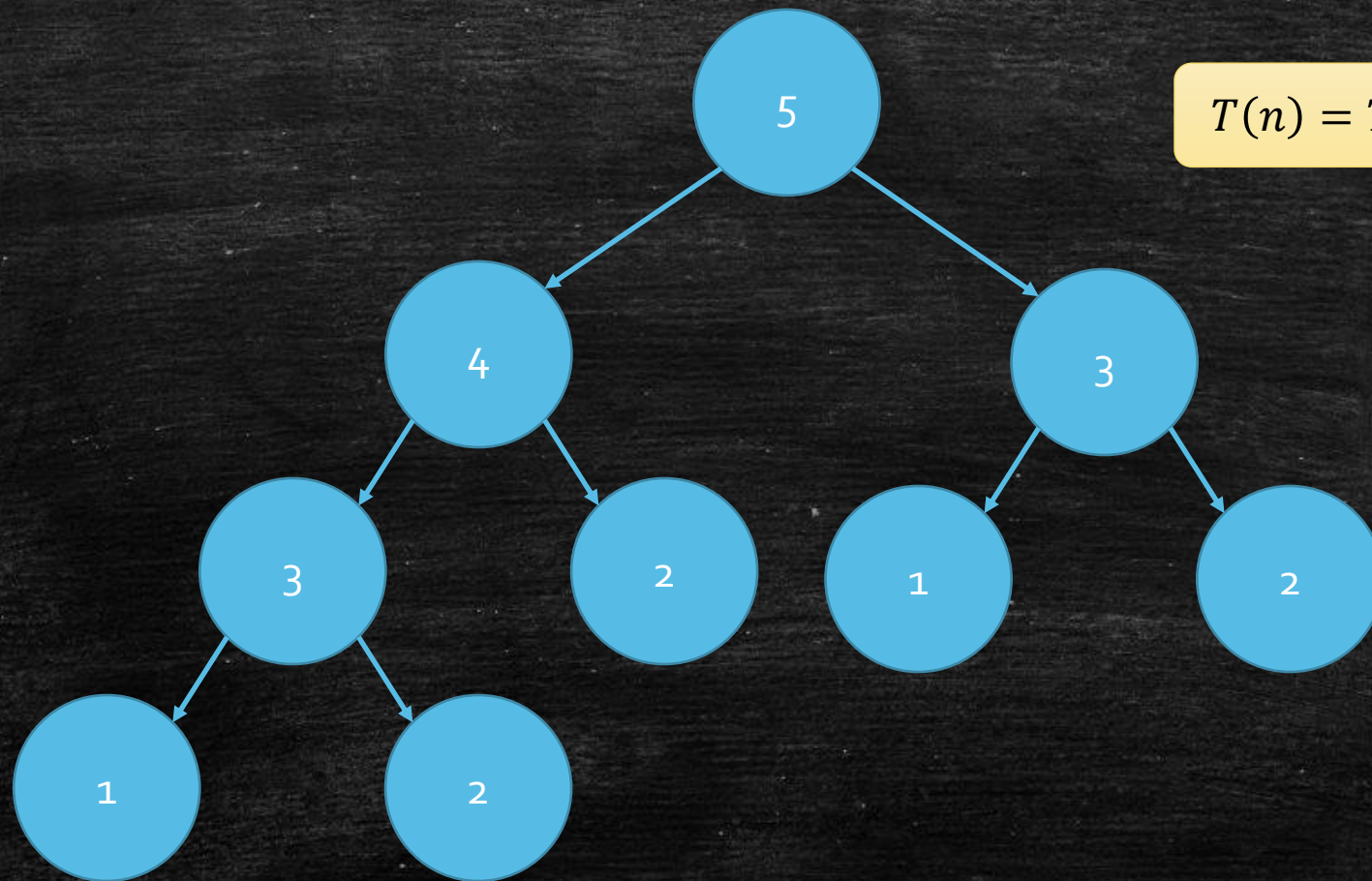
An Easy Example

- Fibonacci
- $Fib(n) = Fib(n - 1) + Fib(n - 2)$
- Solve Recursively

Fibonacci

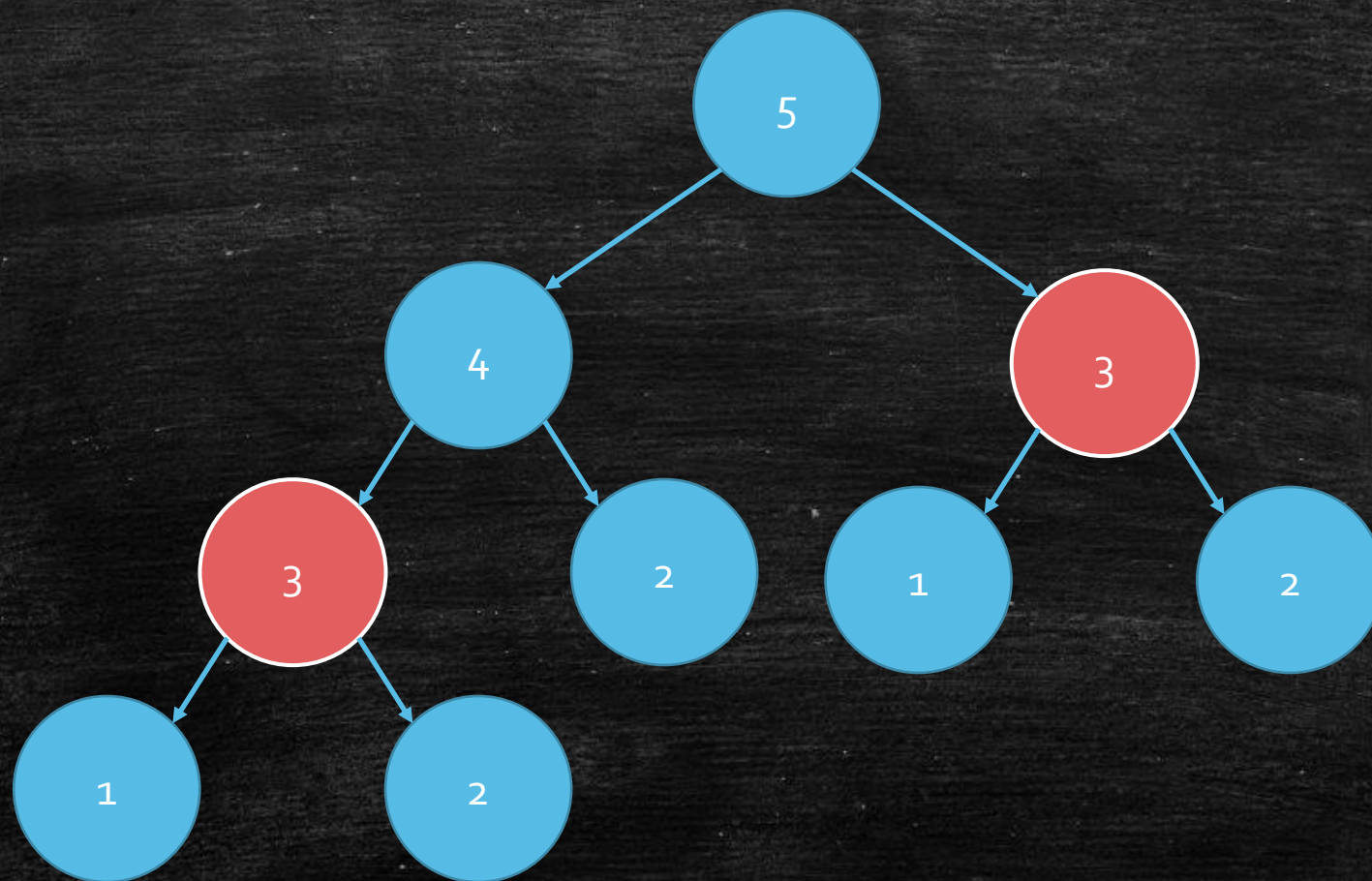
```
function fib(n)
  if n > 1
    return fib(n - 1) + fib(n - 2)
  else
    return 1
```


Recursive Tree

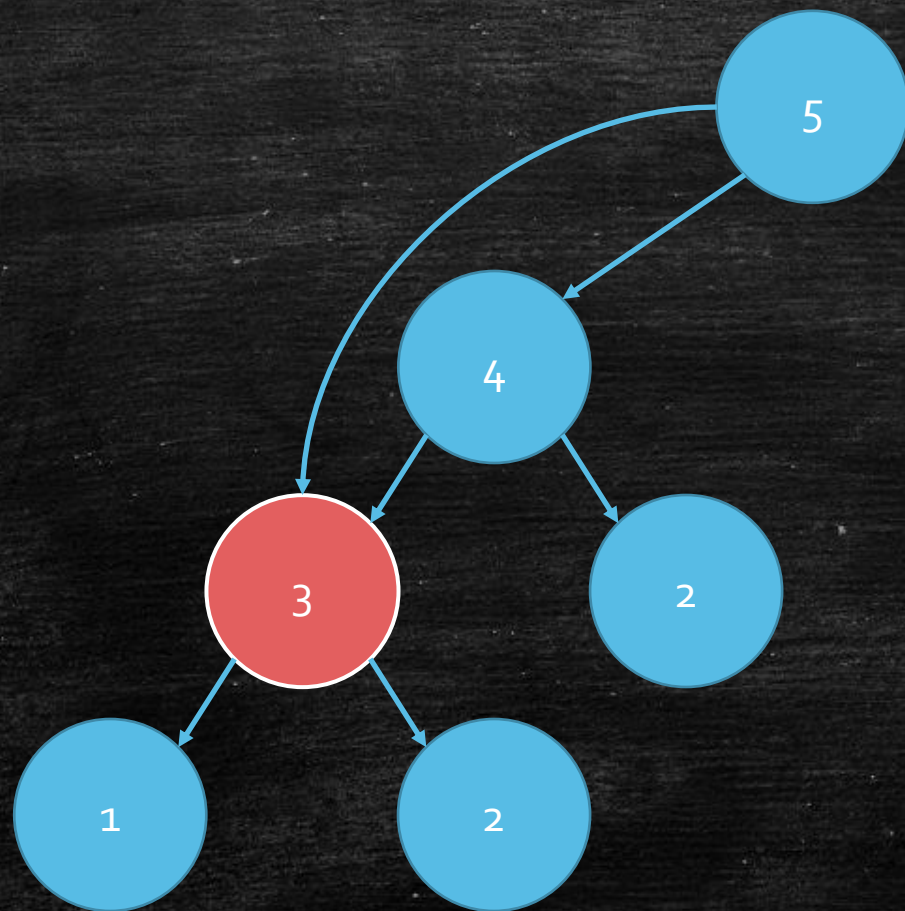


$$T(n) = T(n - 1) + T(n - 2)$$

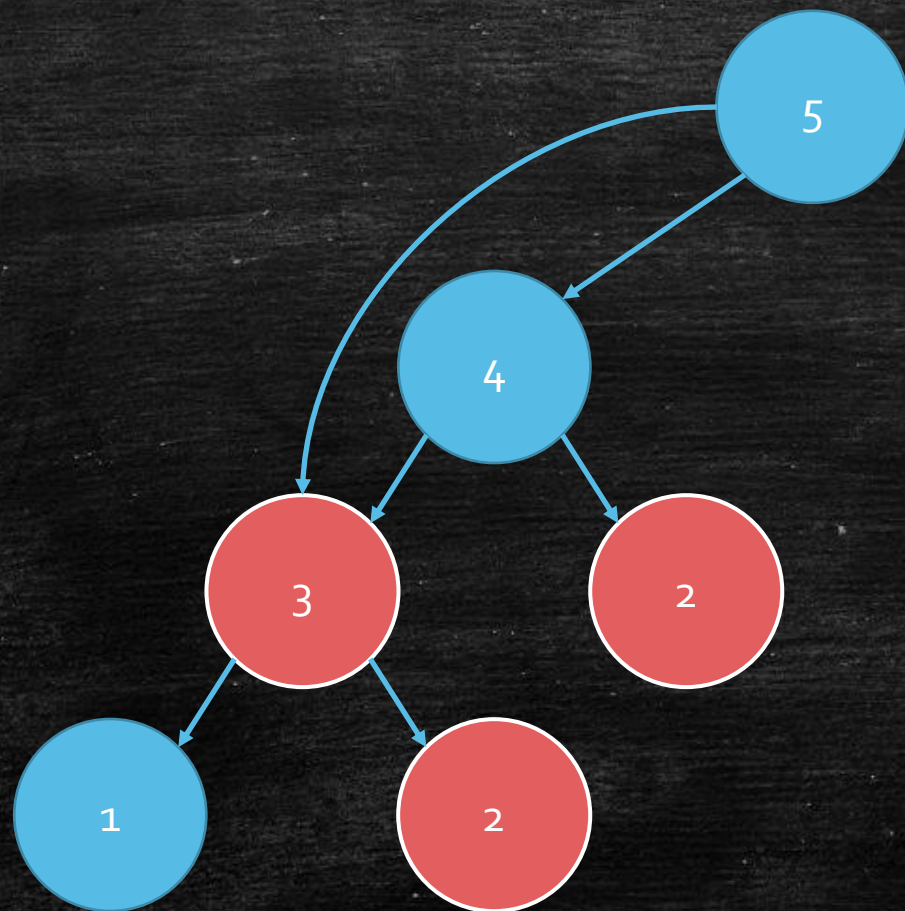
Improvement



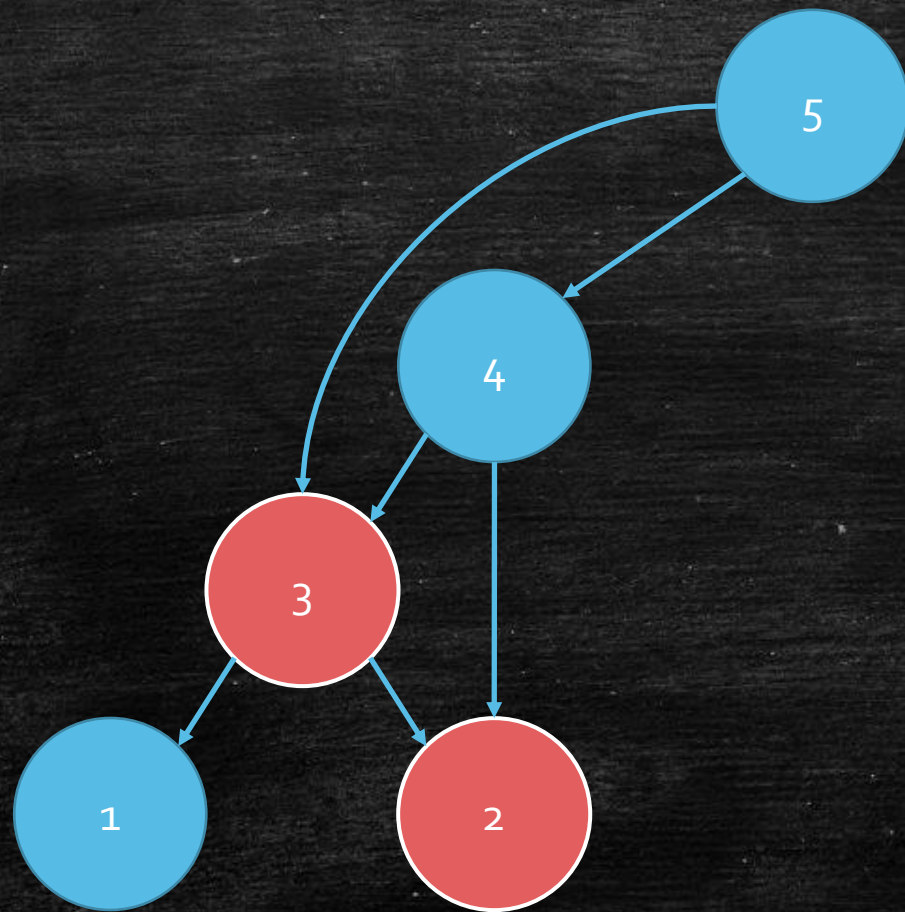
Improvement



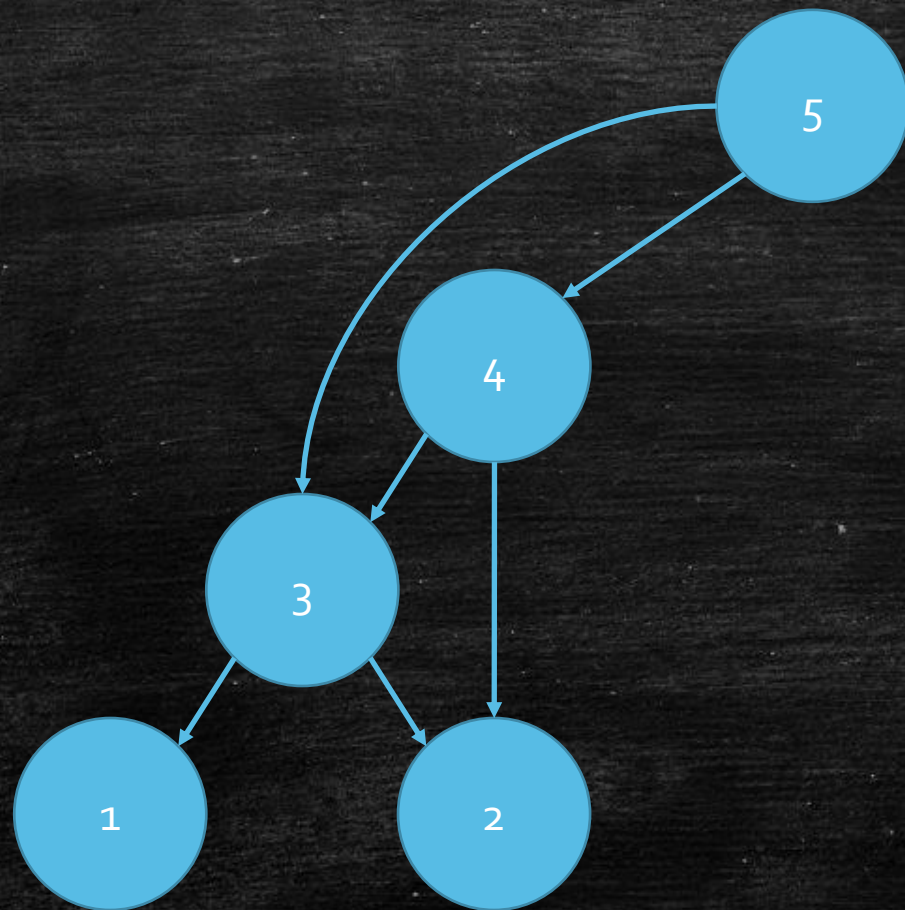
Improvement



Improvement



Improvement



It becomes a DAG!

Implement: memoization

Fibonacci

```
function fib(n)
```

```
    Check whether n is stored, if yes then directly return.
```

```
    if n > 1
```

```
        return & store  $fib(n - 1) + fib(n - 2)$ 
```

```
    else
```

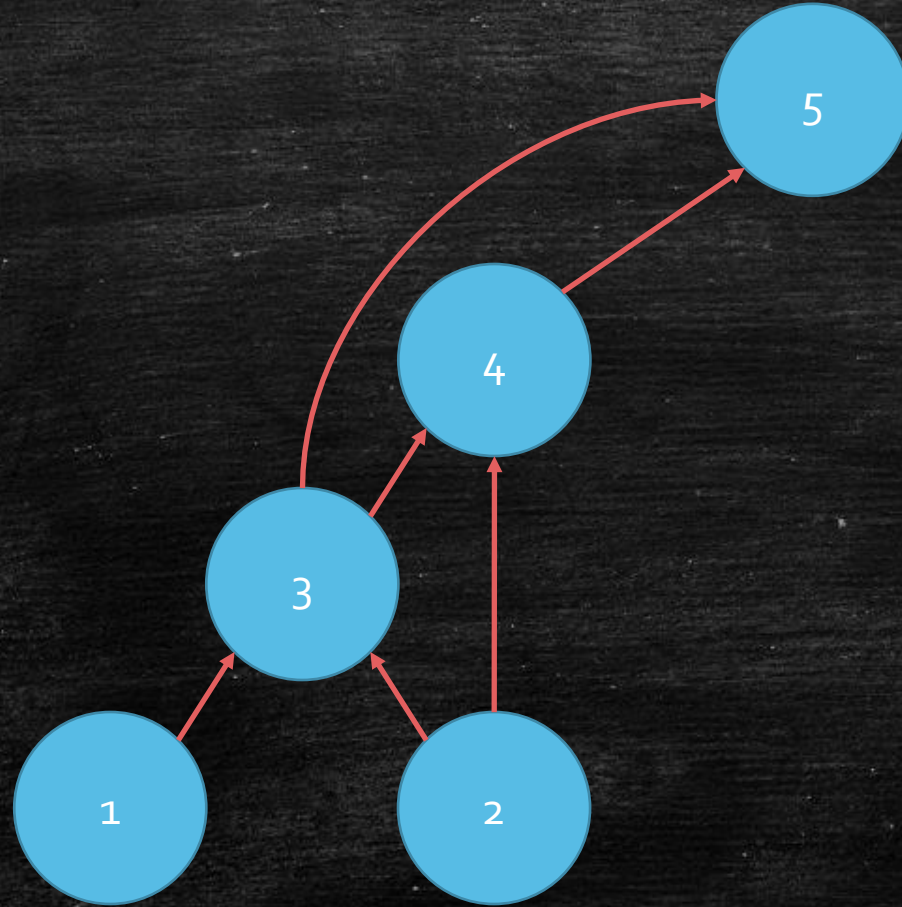
```
        return & store 1
```

Each i

- Calculate once
- Checked twice

Totally: $O(n)$

Improvement



Reverse the graph.
Solve them by the
topological order.

Implement: DP

- Observation
 - If we know $fib(1) \dots fib(i - 1)$.
 - $fib(i)$ can be calculated in constant time.
- DP: calculate all status by a topological order.

Fibonacci

$O(n)$

```
function fib(n)
    fib[0] = fib[1] = 1
    for i = 2 to n
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```


Guideline for DP design

- Design a **recursive** Algorithm.
- Merge the **common** subproblems.
- Check whether we are in a **DAG** and find the **topological order** of this DAG. (usually, by hand.)
- Solve & store the subproblems by the topological order.

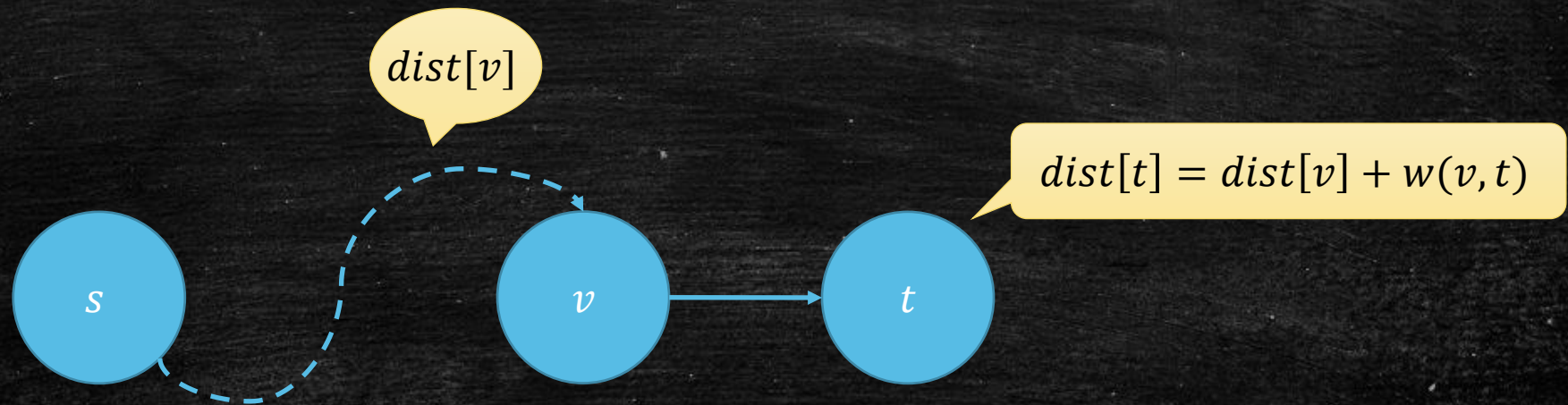
Let us use the guideline

Shortest Path in DAGs

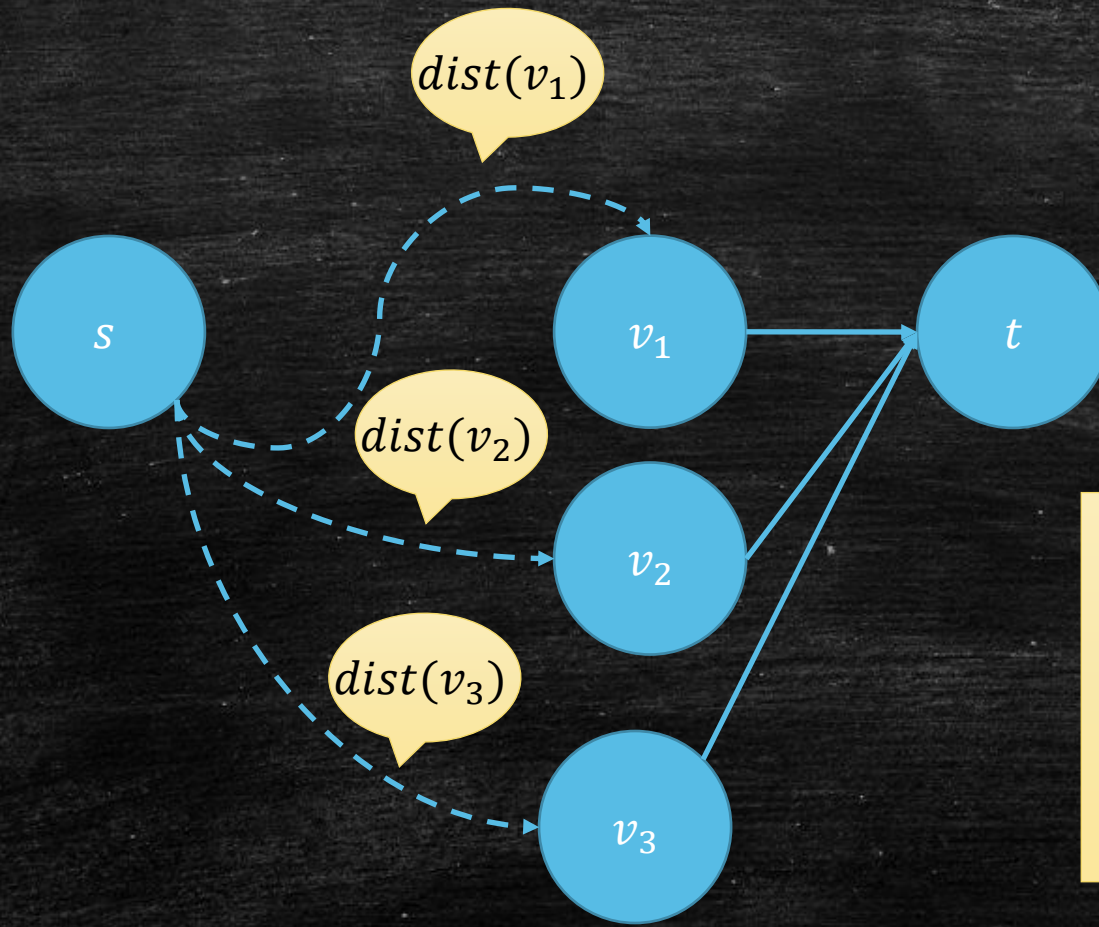
- **Input:** A Directed Acyclic Graph (DAG) $G = (V, E)$, a start vertex $s \in V$, and a weight function $w(e)$ for all $e \in E$. (possible non-negative)
- **Output:** the distance from s to every $v \in V$.

Important Fact

- Not restricted in DAG!
- Used in Dijkstra, BFS, Bellman-Ford.

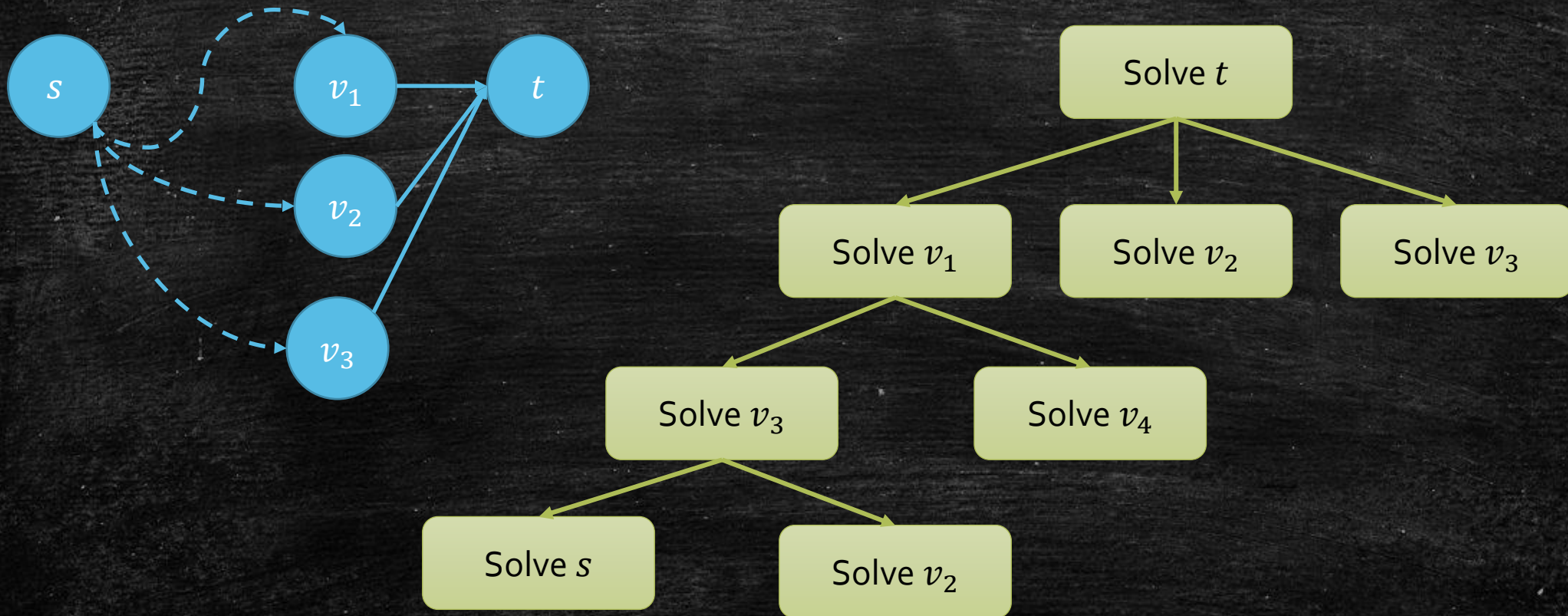


Solve $s \rightarrow t$ distance recursively!

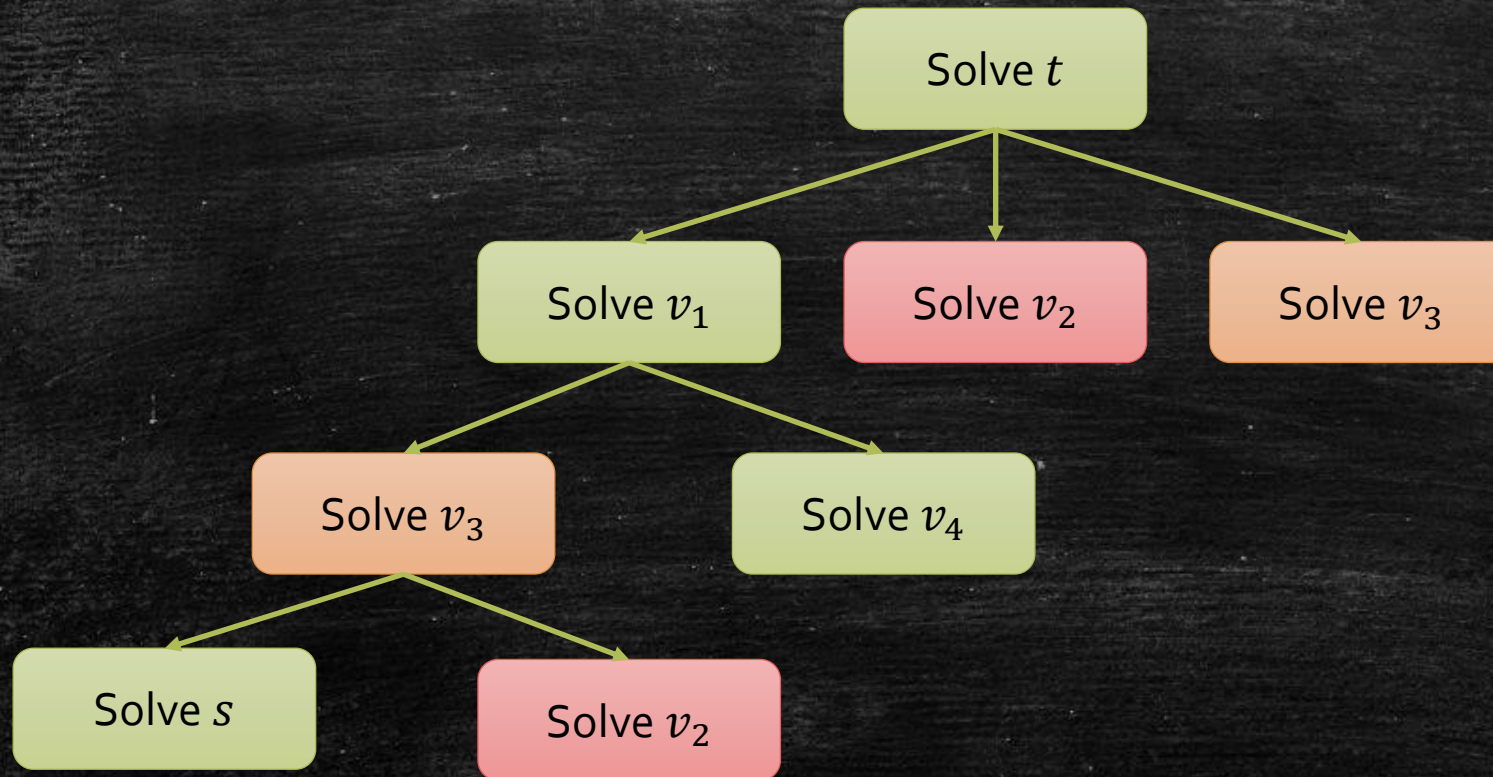


$$dist(t) = \min \begin{cases} dist(v_1) + w(v_1, t) \\ dist(v_2) + w(v_2, t) \\ dist(v_3) + w(v_3, t) \end{cases}$$

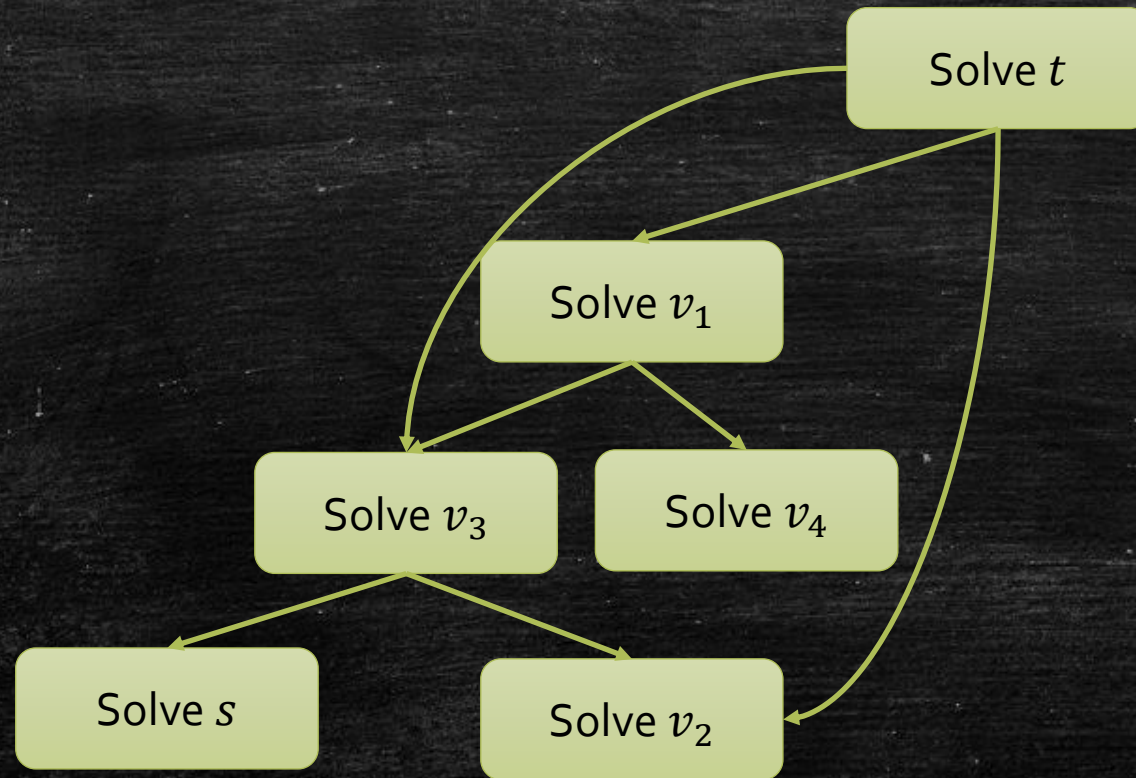
A recursive method to solve $\text{dist}[t]$.



Merge common subproblems

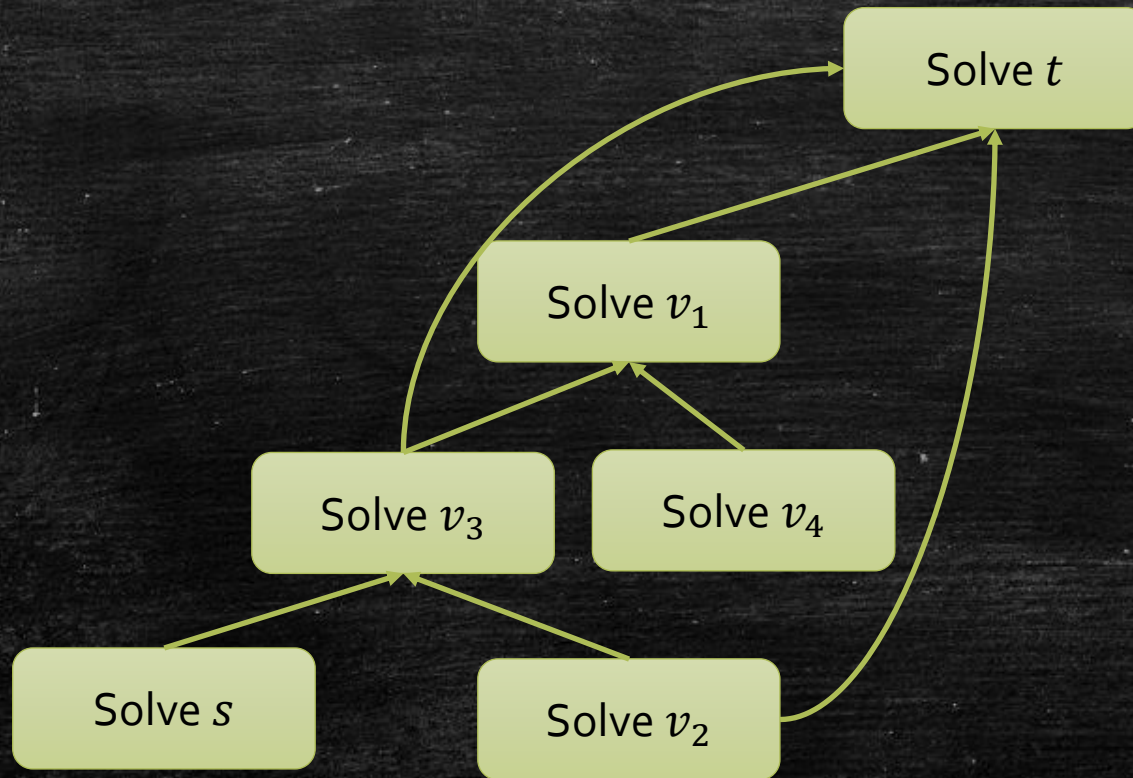


Merge common subproblems



We have at most n subproblems!

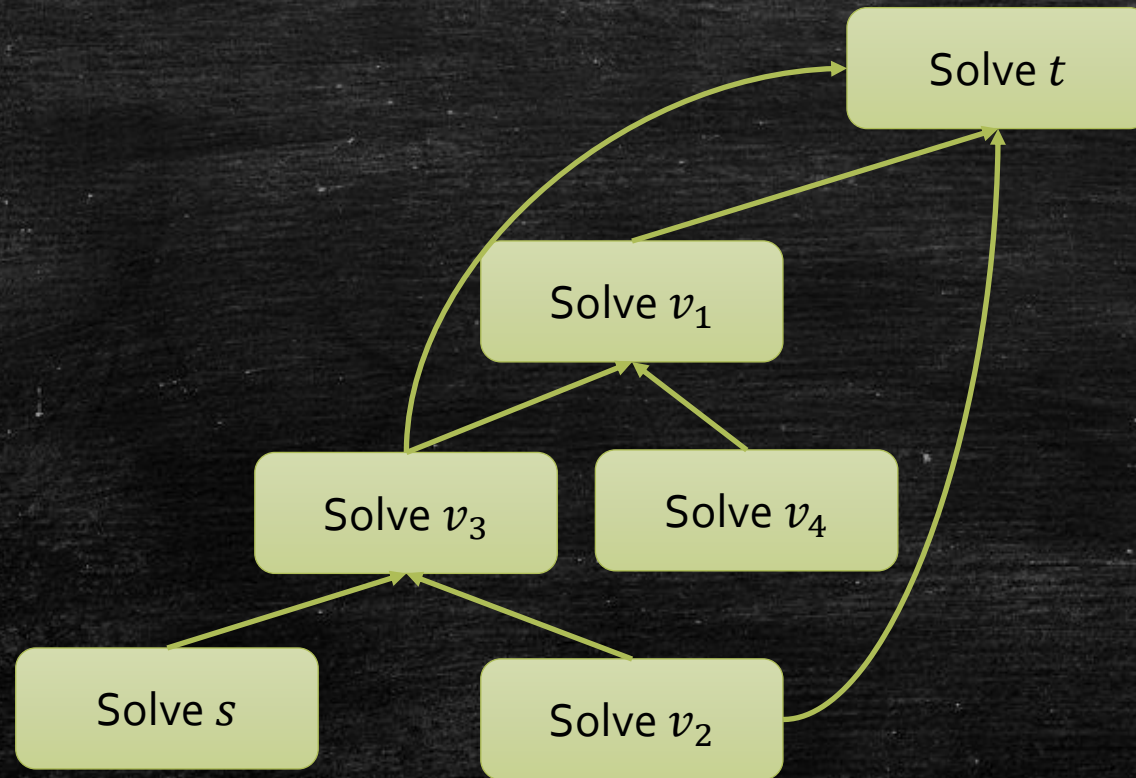
Are we in a DAG?



We have at most n subproblems!

It is exactly a DAG, because it is G !

Solve it by topological order!



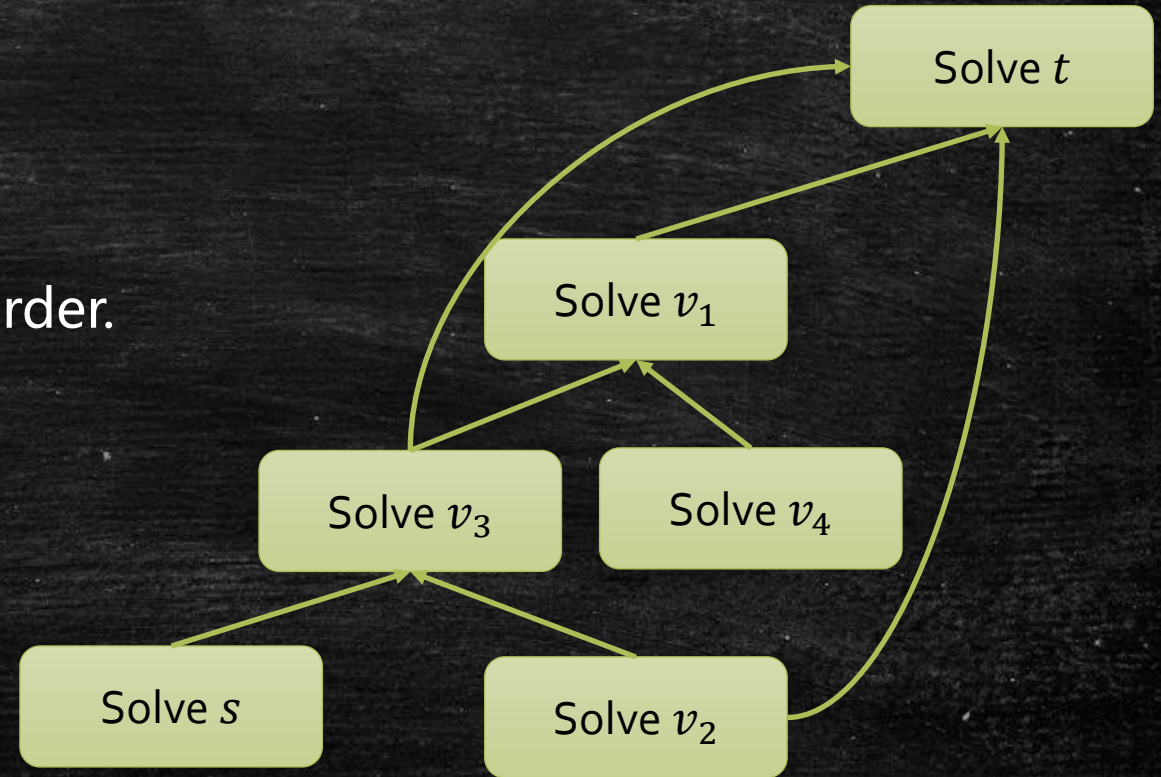
We have at most n subproblems!

It is exactly a DAG, because it is G !

Solve it by topological order!

- Plan

- Find a Topological Order of V .
 - $O(|V| + |E|)$
- $dist[s] = 0$.
- Solve & record $dist[u]$ by the order.
- Solve $dist[u] = \min\{\begin{array}{l} \text{▪ } dist[v_1] + w(v_1, u) \\ \text{▪ } dist[v_2] + w(v_2, u) \\ \text{▪ } dist[v_3] + w(v_3, u) \\ \text{▪ } \dots \end{array}\}$
- $O(|V| + |E|)$



What about the correctness?

- We can easily check the correctness of DP Algorithms by induction.
- Base case:
 - Check our initialization: $dist[s] = 0$.
- Induction:
 - Assume $dist[u_i]$ is correct for all $i < k$.
 - $dist[u_k]$ can be solved correctly by the min of
 - $dist[v_1] + w(u_k, v_1)$
 - $dist[v_2] + w(u_k, v_2)$
 - $dist[v_3] + w(u_k, v_3)$
 - ...

The topological order make
a feasible induction order!

A simpler guideline

- Find subproblems.
- Check whether we are in a **DAG** and find the **topological order** of this DAG. (usually, by hand.)
- Solve & store the subproblems by the topological order.

More DP algorithms!

Longest Increasing Subsequence

- **Input:** A sequence a_1, a_2, \dots, a_n .
- **Output:** the Longest Increasing Subsequence (LIS)
 - $a_{i_1} < a_{i_2} < a_{i_3} \dots < a_{i_k}$
 - $i_1 < i_2 < i_3 \dots < i_k$

1	5	13	2	6	24	15	23	2	16
---	---	----	---	---	----	----	----	---	----

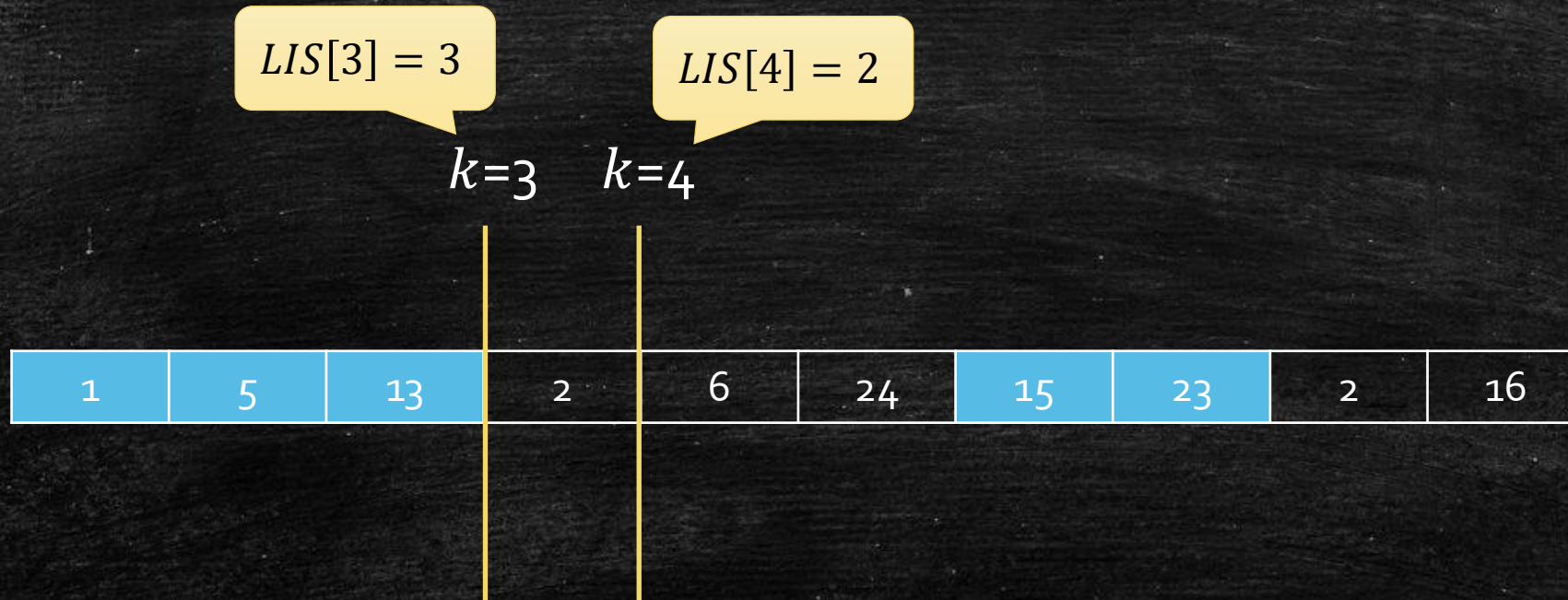
Solve LIS recursively!

- Enumerate the last number a_i .
 - $LIS = \max_{i \leq n} LIS(a_i)$

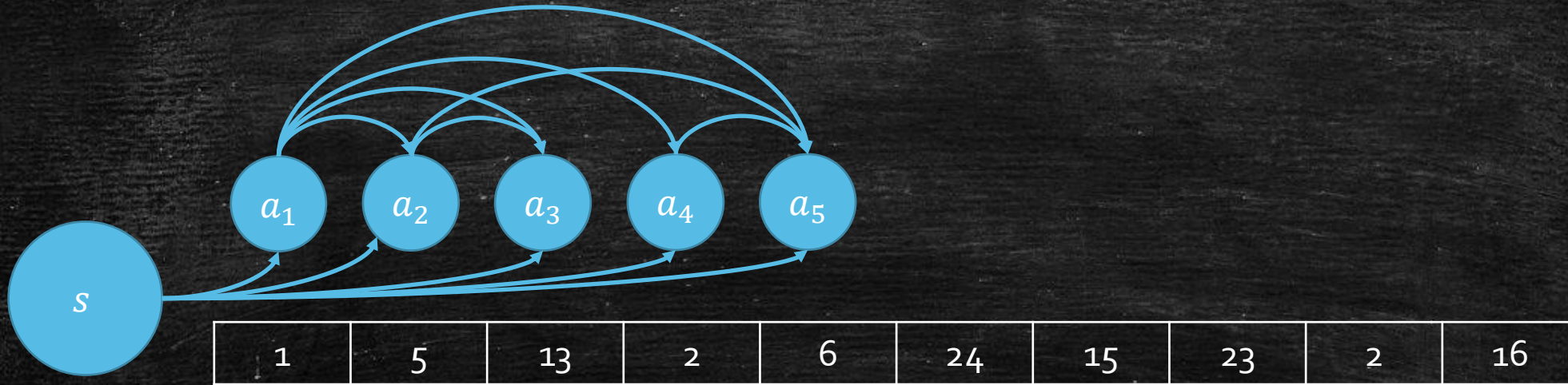
1	5	13	2	6	24	15	23	2	16
---	---	----	---	---	----	----	----	---	----

Define subproblems

- $LIS[k]$: the Longest Increasing Subsequence ended by a_k .



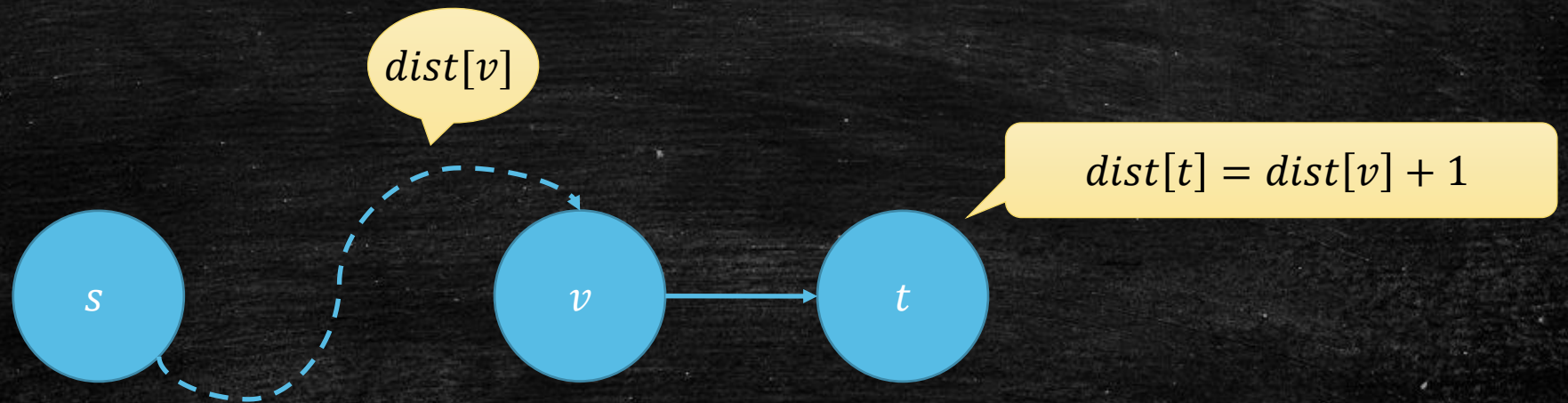
Another view of the problem.



$LIS[k]$ can be viewed as the longest path from s to a_k .

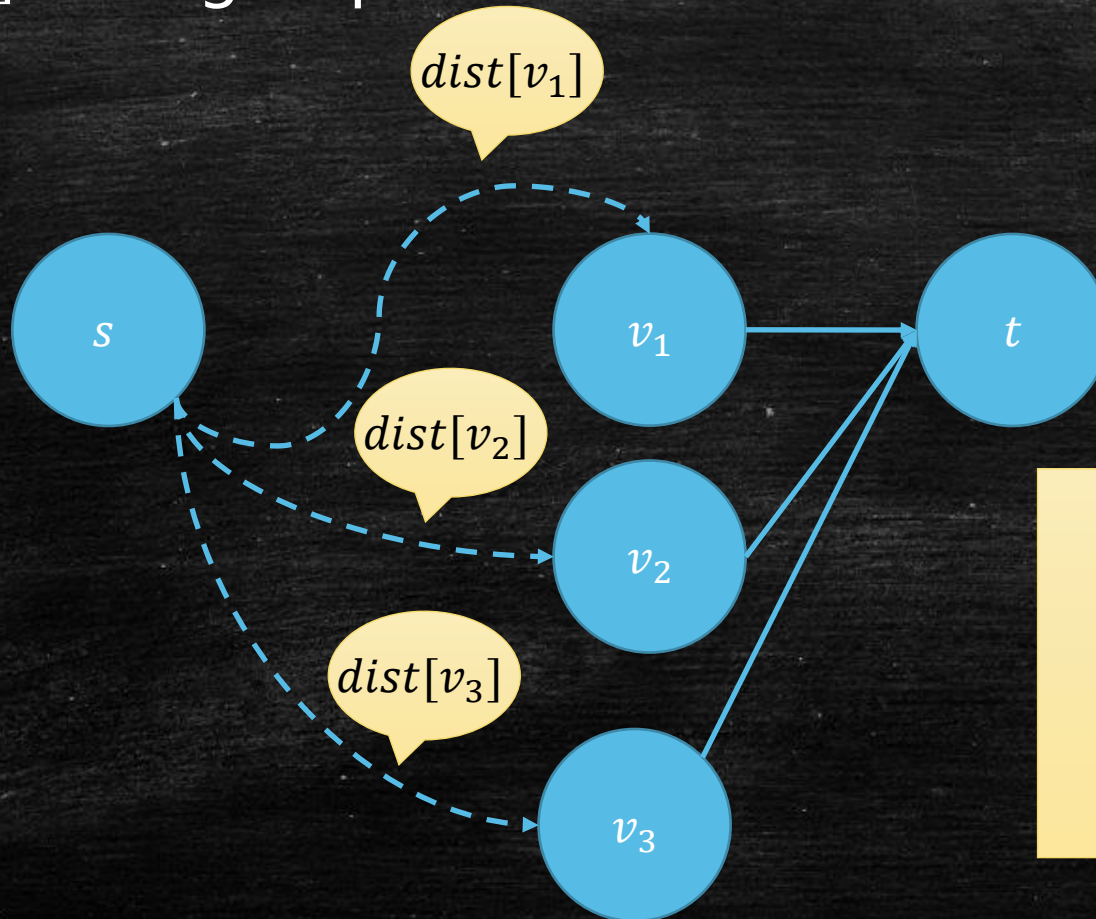
Important Fact

- $dist[v] \rightarrow$ longest path from s to v .



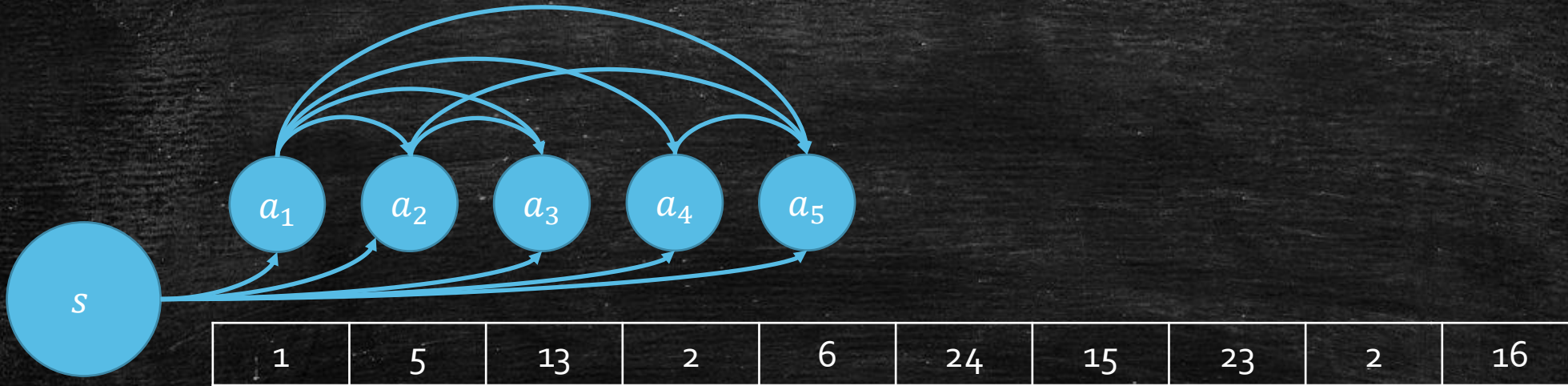
Important Fact

- $dist[v] \rightarrow$ longest path from s to v .



$$dist[t] = \max \begin{cases} dist[v_1] + 1 \\ dist[v_2] + 1 \\ dist[v_3] + 1 \end{cases}$$

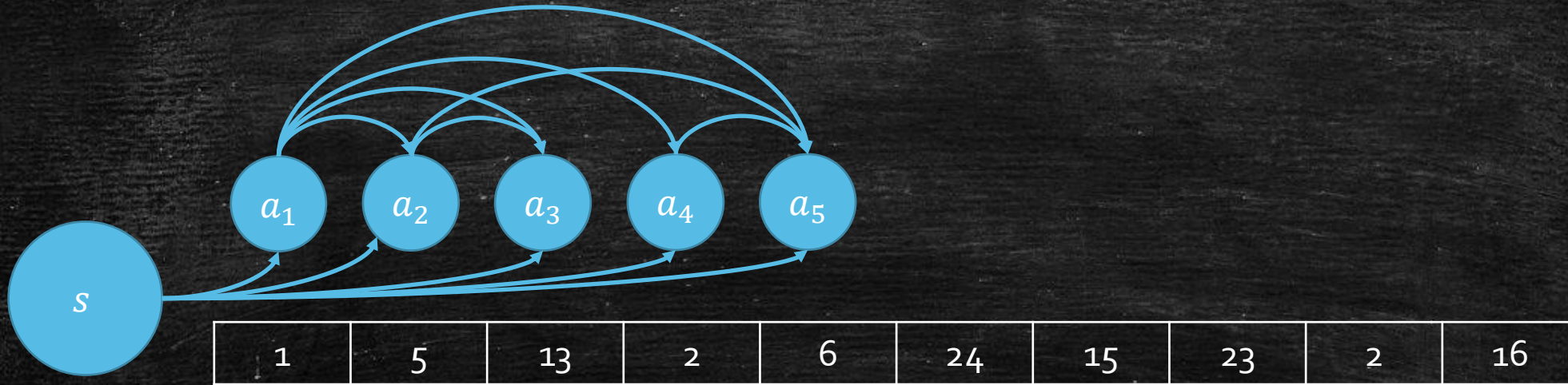
Another view of the problem.



$LIS[k]$ can be viewed as the longest path from s to a_k .

$1 \dots n$ is a topological order!

Another view of the problem.



Longest Increasing Subsequence

function $LIS(n)$

$lis[0] = 0$

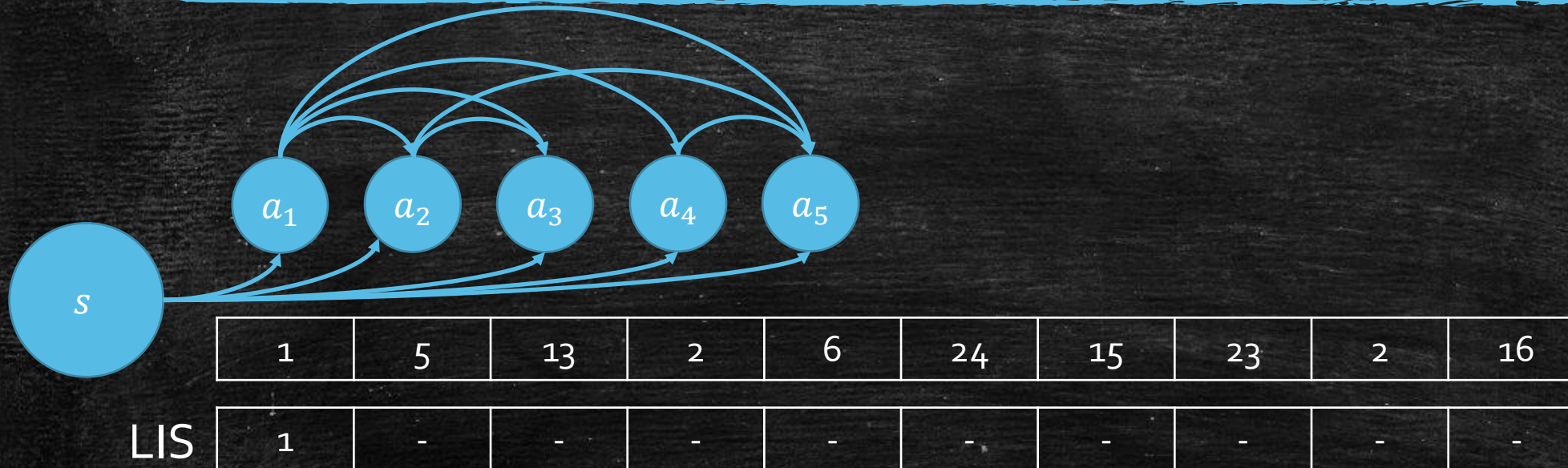
for $i = 1$ to n

$lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$

$s = a_0 = -\infty$

return $\max_{1 \leq i \leq n} lis[i]$

Another view of the problem.



Longest Increasing Subsequence

```
function LIS( $n$ )  
   $lis[0] = 0$   
  for  $i = 1$  to  $n$   
     $lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$   
  return  $\max_{1 \leq i \leq n} lis[i]$ 
```

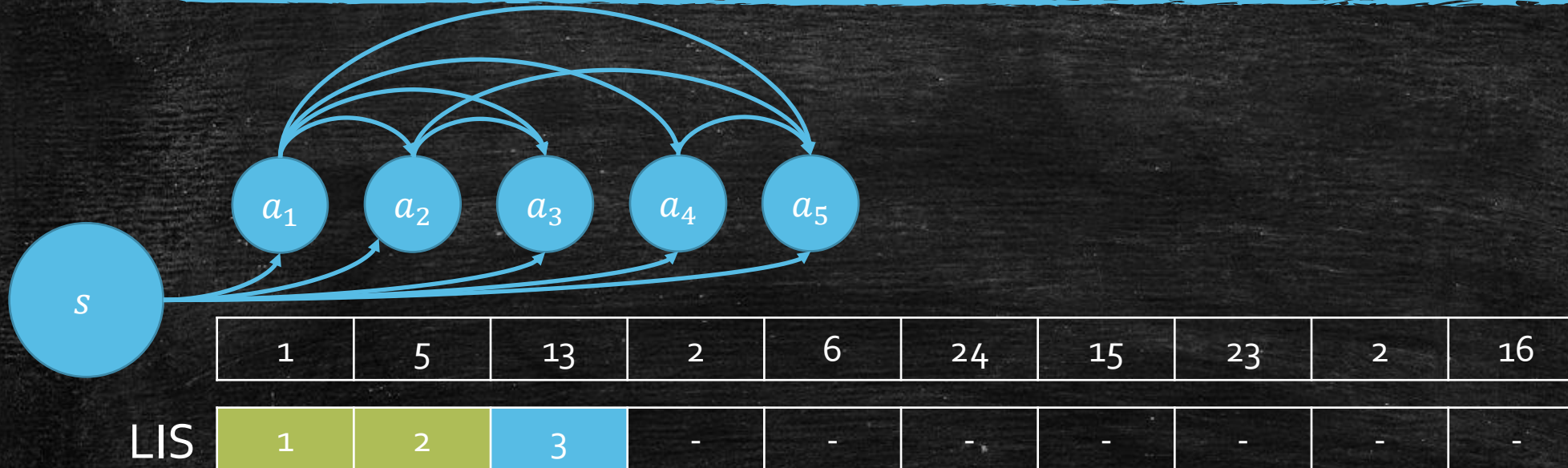

Another view of the problem.



Longest Increasing Subsequence

```
function LIS( $n$ )  
   $lis[0] = 0$   
  for  $i = 1$  to  $n$   
     $lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$   
  return  $\max_{1 \leq i \leq n} lis[i]$ 
```

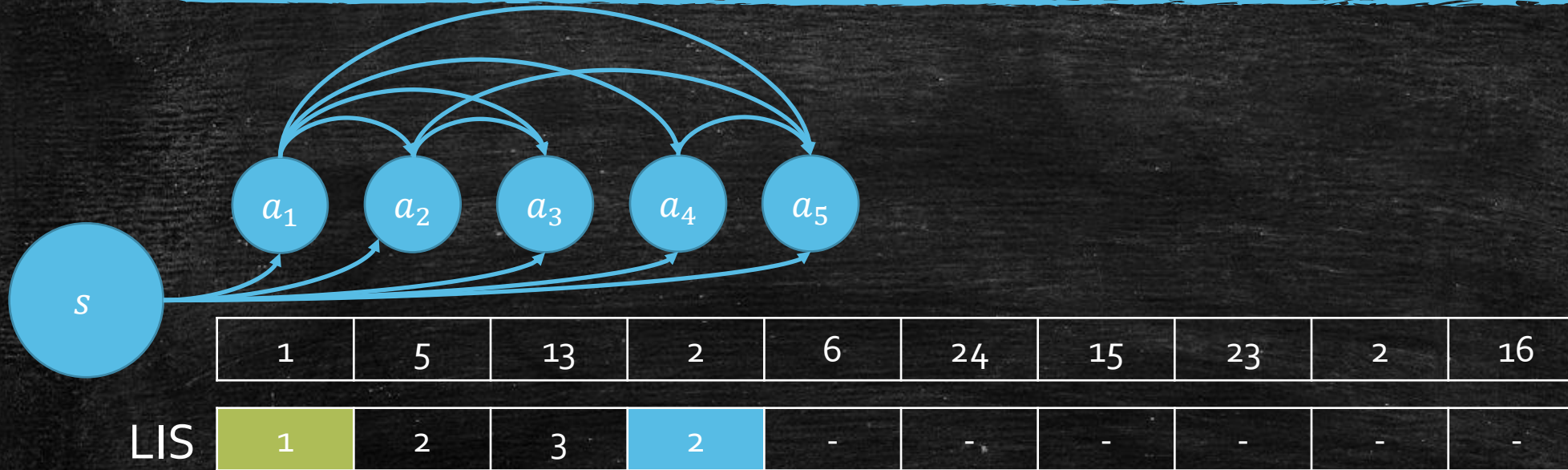

Another view of the problem.



Longest Increasing Subsequence

```
function LIS( $n$ )  
   $lis[0] = 0$   
  for  $i = 1$  to  $n$   
     $lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$   
  return  $\max_{1 \leq i \leq n} lis[i]$ 
```

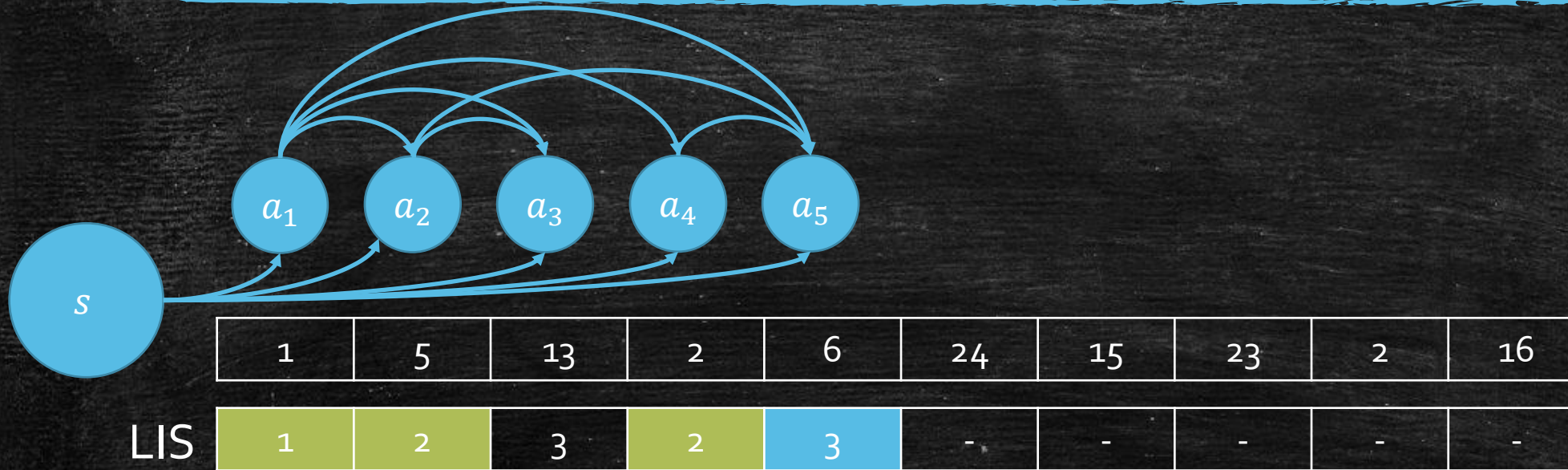

Another view of the problem.



Longest Increasing Subsequence

```
function LIS( $n$ )  
   $lis[0] = 0$   
  for  $i = 1$  to  $n$   
     $lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$   
  return  $\max_{1 \leq i \leq n} lis[i]$ 
```


Another view of the problem.



Longest Increasing Subsequence $O(n^2)$

function $LIS(n)$

$lis[0] = 0$

for $i = 1$ to n

$lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$

$s = a_0 = -\infty$

return $\max_{1 \leq i \leq n} lis[i]$

Edit Distance

- Motivation: How to change from one string to another?
- Allowed operations
 - **Insertion**: insert a character to a specific location.
 - **Deletion**: delete a character from a specific location.
 - **Replacement**: rewrite a character at a specific location.
- Change SNOWY to SUNNY?
 - SNNWY
 - SNNY
 - SUNNY

Another View

- Allowed operations
 - Alignment: Insert space with 0 cost.
 - Insertion: **rewrite** a character from a space at a specific location.
 - Deletion: **rewrite** a character to a space at a specific location.
 - Replacement: **rewrite** a character at a specific location.

S	N	O	W	Y
---	---	---	---	---

S	U	N	N	Y
---	---	---	---	---

Another View

- Allowed operations
 - Alignment: Insert space with 0 cost.
 - Insertion: **rewrite** a character from a space at a specific location.
 - Deletion: **rewrite** a character to a space at a specific location.
 - Replacement: **rewrite** a character at a specific location.

S	N	O	W	Y
---	---	---	---	---

S	U	N	N	Y
---	---	---	---	---

Another View

- Allowed operations

- Alignment: Insert space with 0 cost.
- Insertion: **rewrite** a character from a space at a specific location.
- Deletion: **rewrite** a character to a space at a specific location.
- Replacement: **rewrite** a character at a specific location.

Change alignment

S	N	O	W	_	Y
---	---	---	---	---	---

S	U	N	N	Y	_
---	---	---	---	---	---

Another View

- Allowed operations
 - Alignment: Insert space with 0 cost.
 - Insertion: **rewrite** a character from a space at a specific location.
 - Deletion: **rewrite** a character to a space at a specific location.
 - Replacement: **rewrite** a character at a specific location.

The same
as before.

S	_	N	O	W	Y
---	---	---	---	---	---

S	U	N	N	_	Y
---	---	---	---	---	---

Optimization

- What is the minimized cost to change from a string to another? (it is symmetric)
- We call it the **Edit Distance** of the two string.
- Usage
 - Quantifying how dissimilar two strings are.

Edit Distance Calculation

- **Input:** two strings
 - $X: x_1, x_2, \dots, x_m$
 - $Y: y_1, y_2, \dots, y_n$
- **Output:** the edit distance between x and y .
- Another view
 - Find the best alignment!

Find out the subproblems

Imagine the best alignment from the tail.

X	?	?	?	?	?
Y	?	?	?	?	?

Find out the subproblems

Case 1

X	?	?	?	?	x_m
-----	---	---	---	---	-------

Y	?	?	?	?	y_n
-----	---	---	---	---	-------

Find out the subproblems

Case 2

X	?	?	?	?	—
-----	---	---	---	---	---

Y	?	?	?	?	y_n
-----	---	---	---	---	-------

Find out the subproblems

Case 3

X	?	?	?	?	x_m
Y	?	?	?	?	—

Find out the subproblems

Case 1

The best alignment for
 $X[1 \sim m - 1]$ and
 $Y[1 \sim n - 1]$.



Plus one cost if
 $x_m \neq y_n$

Find out the subproblems

Case 2

The best alignment for
 $X[1 \sim m]$ and
 $Y[1 \sim n - 1]$.

X	?	?	?	?	—
Y	?	?	?	?	y_n

Plus one cost

Find out the subproblems

Case 3

The best alignment for
 $X[1 \sim m - 1]$ and
 $Y[1 \sim n]$.

X	?	?	?	?	x_m
Y	?	?	?	?	—

Plus one cost

Do you find out the
subproblems?








Subproblems

- $ED[i, j]$: The edit distance between $X[1..i]$ and $Y[1..j]$.
- $ED[n, m]$: The edit distance between X and Y .
- How to solve $ED[i, j]$: min of three cases
 - $ED[i - 1, j - 1] + \mathbf{1}_{x_i \neq y_j}$
 - $ED[i, j - 1] + 1$
 - $ED[i - 1, j] + 1$

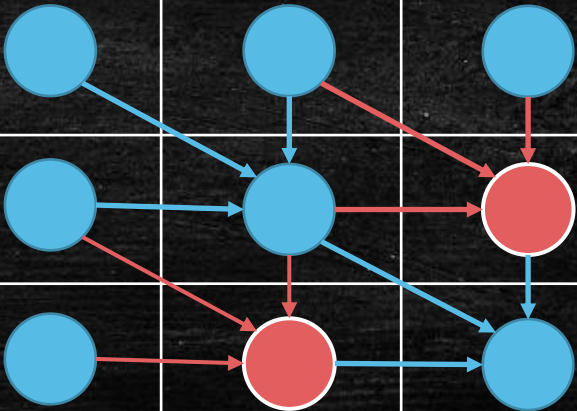
Is it DAG?

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$							
$i = 1$							
$i = 2$							
$i = 3$							
$i = \dots$							
$i = m$							

Is it DAG?

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$							
$i = 1$							
$i = 2$							
$i = 3$							
$i = \dots$							
$i = m$							

Is it DAG?

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$							
$i = 1$							
$i = 2$							
$i = 3$							
$i = \dots$							
$i = m$							

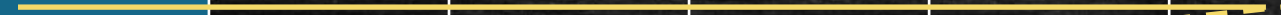
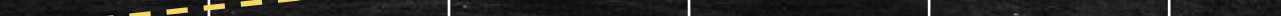

A topological order

$ED[i, j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$							
$i = 1$							
$i = 2$							
$i = 3$							
$i = \dots$							
$i = m$							




Start!

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$	0	1	2	3	4	5	6
$i = 1$	1						
$i = 2$	2						
$i = 3$	3						
$i = \dots$	4						
$i = m$	5						




Start!

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$	0	1	2	3	4	5	6
$i = 1$	1						
$i = 2$	2						
$i = 3$	3						
$i = \dots$	4						
$i = m$	5						





Start!

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$	0	1	2	3	4	5	6
$i = 1$	1						
$i = 2$	2						
$i = 3$	3						
$i = \dots$	4						
$i = m$	5						

Start!





$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$	0	1	2	3	4	5	6
$i = 1$	1						
$i = 2$	2						
$i = 3$	3						
$i = \dots$	4						
$i = m$	5						

Start!

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$	0	1	2	3	4	5	6
$i = 1$	1						
$i = 2$	2						
$i = 3$	3						
$i = \dots$	4						
$i = m$	5						

Running Time?

$$O(nm) \cdot O(1)$$

$ED[i,j]$	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$j = 4$	$j = \dots$	$j = n$
$i = 0$	0	1	2	3	4	5	6
$i = 1$	1						
$i = 2$	2						
$i = 3$	3						
$i = \dots$	4						
$i = m$	5						

Knapsack Problems

- **Input:** n items with cost c_i and value v_i , and a capacity W .
- **Output:** Select a subset of items, with total cost at most W . The goal is to maximize the total value.

A nice greedy approach.

- Select the item with from larger **value-cost** ratio.

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

A nice greedy approach.

- Select the item with from larger **value-cost** ratio.

$$W = 10000$$

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

A nice greedy approach.

- Select the item with from larger **value-cost** ratio.

$$W = 10000$$

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

It looks quite intuitive, and it is correct now!

A nice greedy approach.

- Select the item with from larger **value-cost** ratio.

$W = 100000$

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

But when we
become rich...

Problem: items are
not divisible!

A nice greedy approach.

- Select the item with from larger **value-cost** ratio.

$W = 100000$

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Qie Gao	90000	100000

But when we
become rich...

Problem: items are
not indivisible!



Buy 0.82112 portion of
the "Qie Gao"

What if items are indivisible?

- The Knapsack Problem is NP-Hard!
- Are we going to talk about approximation algorithms?
- No!
- Let's make a DP algorithm with reasonable running time!

Find out subproblems!

- What we always do before:
- $f[i]$: the maximum value we can get by using the first i items.

$f[i]$	5	10	13	16	21	30	?
--------	---	----	----	----	----	----	---

Find out subproblems!

- What we always do before:
- $f[i]$: the maximum value we can get by using the first i items.

How to solve
 $f[i]$ by $f[j < i]$?

$f[i]$	5	10	13	16	21	30	?
--------	---	----	----	----	----	----	---

Find out subproblems!

- What we always do before:
- $f[i]$: the maximum value we can get by using the first i items.

$f[i]$	5	10	13	16	21	30	?
--------	---	----	----	----	----	----	---

How to solve $f[i]$ by $f[j < i]$?

We know $f[j]$ but we do not know how much budget it uses!

The space of subproblems is not large enough!

Find out subproblems!

- What we always do before:
- $f[i]$: the maximum value we can get by using the first i items.
- Use $g[i]$ to store how much budget $f[i]$ uses.

$f[i]$	5	10	13	16	21	30	?
--------	---	----	----	----	----	----	---

How to solve $f[i]$ by $f[j < i]$?

We know $f[j]$ but we do not know how much budget it uses!

Find out subproblems!

- What we always do before:
- $f[i]$: the maximum value we can get by using the first i items.
- Use $g[i]$ to store how much budget $f[i]$ uses.

$f[i]$	13	16	21	30	?
--------	----	----	----	----	---

It is greedy, and it is not optimal!

How to solve $f[i]$ by $f[j < i]$?

We know $f[j]$ but we do not know how much budget it uses!

Find out subproblems!

- What we always do before:
- $f[i]$: the maximum value we can get by using the first i items.
- ~~Use $g[i]$ to store how much budget $f[i]$ uses.~~

$f[i]$	13	16	21	30	?
--------	----	----	----	----	---

It is greedy, and it is not optimal!

How to solve $f[i]$ by $f[j < i]$?

We know $f[j]$ but we do not know how much budget it uses!

Move back to recursion to check how much we need.

- Buy Hermès or not?
- Buy: Earn 90000, continue to buy "iPhone, Book, and Laptop" by $W - 100000$ budget.
- Not Buy: Earn 0, continue to buy "iPhone, Book, and Laptop" by W budget.

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

Find out subproblems!

- What we always do before:
- $f[i, w]$: the maximum value we can get by using the first i items, and with w budget.
- ~~Use $g[i]$ to store how much budget $f[i]$ uses.~~

$f[i]$	5	10	13	16	21	30	?
--------	---	----	----	----	----	----	---

How to solve $f[i]$ by $f[j < i]$?

We know $f[j]$ but we do not know how much budget it uses!

Find out subproblems!

- What we always do before:
- $f[i, w]$: the maximum value we can get by using the first i items, and with w budget.

$f[i, w]$	0	1	2	3	4	5	6	...	W
0									
1									
2									
3						$f[i, w]$			
...									
n									$f[n, W]$

How to solve
 $f[i, w]$.

Solve $f[i, w]$

- What we always do before:
- $f[i, w]$: the maximum value we can get by using the first i items, and with w budget.
- Two options for item i
 - **Buy it:** We can at most use $w - c_i$ budget before i .
 - **Not Buy it:** We can at most use w budget before i .
 - Solve $f[i, w] = \max\{f[i - 1, w], f[i - 1, w - c_i] + v_i\}$.

Check the topological order

- $f[i, w]$: the maximum value we can get by using the first i items, and with w budget.
- $f[i, w] = \max\{f[i - 1, w], f[i - 1, w - c_i] + v_i\}$

$O(nW) \cdot O(1)$

$f[i, w]$	0	1	2	3	4	5	6	...	W
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0					$f[i, w]$			
...	0								
n	0								$f[n, W]$

Diagram details: A yellow bracket above the table header labels the columns from index 2 to 5 as c_i . A blue arrow points from the cell at row 2, column 3 to the cell at row 3, column 5, which is labeled $f[i, w]$.

Knapsack has many
variants!

Surplus Supply

- **Input:** n items with cost c_i and value v_i , and a capacity W .
- **Output:** Select some items (each items can be selected more than once), with total cost at most W . The goal is to maximize the total value.

	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

How to transfer subproblems now?

- Two options for item i
 - **Buy it:** We can at most use $w - c_i$ budget before i .
 - **Not Buy it:** We can at most use w budget before i .
 - Solve $f[i, w] = \max\{f[i - 1, w], f[i - 1, w - c_i] + v_i\}$.
- Problem!
 - We can buy multiple times!

A new subproblem transfer!

- Problem!
 - We can buy multiple times!
- New transfer
 - **Not Buy it anymore:** We can at most use w budget before i .
 - **Buy it:** We can at most use $w - c_i$ budget before i and i .
 - Solve $f[i, w] = \max\{f[i - 1, w], f[i, w - c_i] + v_i\}$.

Check the topological order

- $f[i, w]$: the maximum value we can get by using the first i items, and with w budget.
- $f[i, w] = \max\{f[i - 1, w], f[i, w - c_i] + v_i\}$

[illegible]

Let us program it!

- $f[i, w]$: the maximum value we can get by using the first i items, and with w budget.
- $f[i, w] = \max\{f[i - 1, w], f[i, w - c_i] + v_i\}$

Knapsack with Surplus Supply $O(nW)$

```
function knapsack(n)
   $f[0,0] = f[0,1] = f[0,2] = \dots = f[0,W] = 0$ 
   $f[0,0] = f[1,0] = f[2,0] = \dots = f[n,0] = 0$ 
  for  $w = 0$  to  $W$ 
    for  $i = 1$  to  $n$ 
       $f[i, w] = \max\{f[i - 1, w], f[i, w - c_i]\}$ 
  return  $f[n, W]$ 
```


We can make it simple

- $f[w]$: the maximum value we can with w budget.
- $f[w] = \max_{i=1 \sim n} \{f[w], f[w - c_i] + v_i\}$

Knapsack with Surplus Supply

```
function knapsack(n)
  f[0] = f[1] = ... f[n] = 0
  for w = 0 to W
    for i = 1 to n
      f[w] = max{f[w], f[w - c_i] + v_i}
  return f[n, W]
```

$O(nW)$ but with less space.

They are not polynomial
on the input size!

$O(nW)$ is not polynomial!

- Input: n items with cost c_i and value v_i , and a capacity W .
- Input size: the unit of bits to represent the input.
- $W = 2^N$ by using N bits
 - time complexity becomes $O(2^N)$.

Input of The Knapsack Problem

- **Input** of Knapsack
 - n, W
 - n values v_i and n costs c_i .
- Assume we have $O(N)$ bits, (**input size** = N)
- To present n values and costs, we need to use at least $O(n)$ bits. Hence, we at most present $n = O(N)$ with $O(N)$ bits.
- W can be $O(2^N)$
- So, the running time $O(nW)$ can become $O(N2^N)$, which is not polynomial!

Today's goal

- Learn what is DP.
- Learn how to **prove** DP's correctness.
- Learn the general **guideline** for designing DP Algorithms.
- Learn to apply the **guideline** on:
 - Fibonacci
 - Shortest Path on DAGs
 - Edit Distance
 - Knapsack