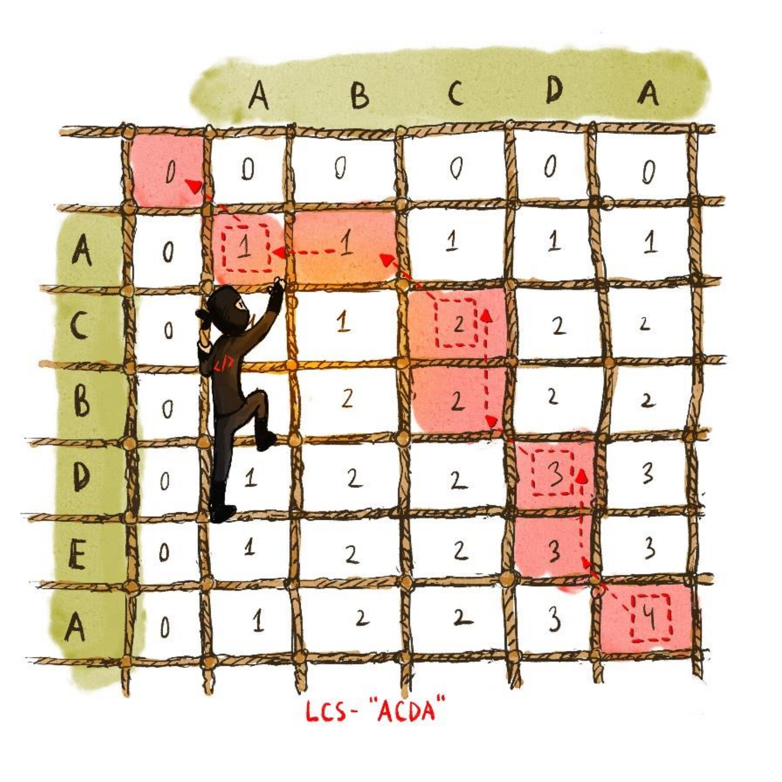
CS218: Design And Analysis Of Algorithms

Dynamic Programming II Memoization



Yan Gu

About Midterm Exam

- **Time: 11am-2pm on Feb 13**
- **Location: WCH 205/206**
- **Preparation: 2-page double-sided letter-size handwritten cheat-sheet**
- **Problems (tentatively):**
  - Multiple Choices
  - Fill-in-the-blank
  - Greedy proof
  - DP algorithm design
  - Tree algorithm design

Things to learn for dynamic programming

- Understand why dynamic programming makes an algorithm faster
- Understand the structure of dynamic programming
- Understand the classic DP algorithms and their variants
- Understand how to in general design DP algorithms
- Understand how to accelerate DP algorithms and apply to real-world applications

What is dynamic programming?

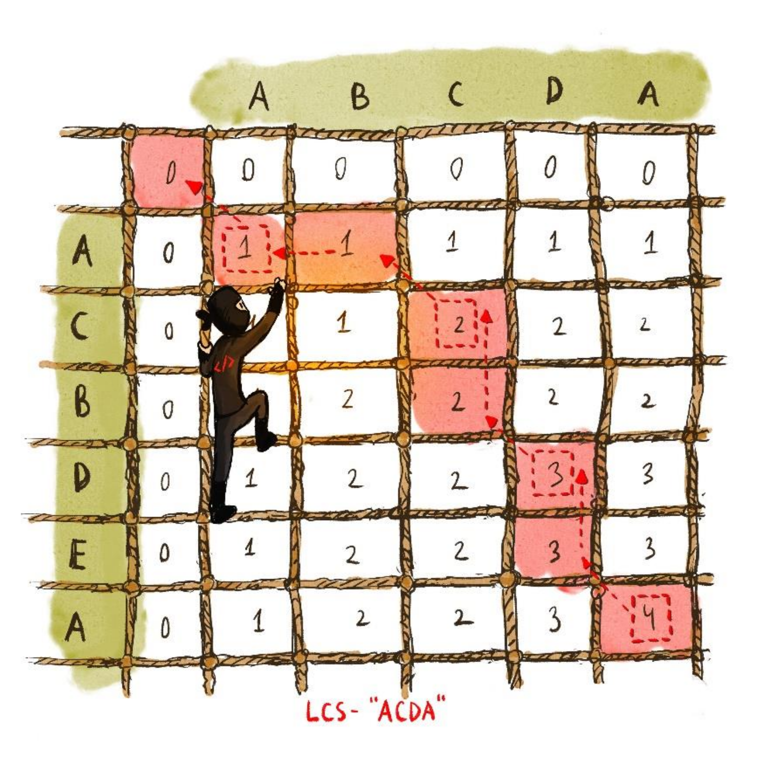
- Optimal substructure (**states**)
  - What defines a **subproblem**?
  - What should be **memoized** as the index/value of your array? What will you look up for later computations?
- The **decisions**
  - What are the possible “last move”?
  - Take max/min (or something else) for all decisions?
- **Boundary: What are the base cases?**
- **Answer: What to output?**
- **Recurrence**
  - Compute current state from previous states

Things to learn for dynamic programming

- Understand why dynamic programming makes an algorithm faster
- Understand the structure of dynamic programming
- Understand the classic DP algorithms and their variants
- Understand how to in general design DP algorithms
- Understand how to accelerate DP algorithms and apply to real-world applications

CS218: Design And Analysis Of Algorithms

Dynamic Programming II Memoization



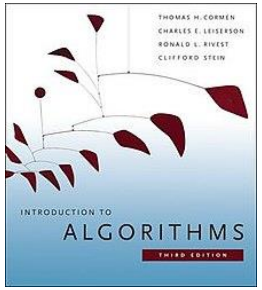
Yan Gu

Knapsack problem

- **A knapsack of weight limit W**
- **n items with value v_i and weight w_i**
- **How to use the knapsack to take the maximum total value?**
- **Variants:**
  - 0/1 knapsack (each item can be used at most once)
  - Unlimited knapsack (unlimited number of copies for each item)
  - k-knapsack (each item can be used k times, k can be different for different items)
  - Items conflict with each other
  - Items depend on each other
  - ……


$80, 2lb


$50, 1lb



$70, 5lb


$1500, 8lb

The DP implementation

```
int knapsack(int i, int j) {
    if (ans[i][j] != -1) return ans[i][j];
    if (i==0 or j == 0) return 0;
    int best = knapsack(i-1, j);
    if (j >= weight[i]) best = max(best, knapsack(i-1, j-weight[i])+value[i]);
    return ans[i][j] = best;
}

int ans[n][W] = {-1, … , -1};
answer = knapsack(n, W);
```

A non-recursive implementation

```
int ans[0][i] = {0, … , 0};
for i = 1 to n do
    for j = 0 to W do {
        ans[i][j] = ans[i-1][j];
        if (j >= weight[i])
            ans[i][j] = max(ans[i][j], ans[i-1][j-weight[i]]+value[j]);
    }
return ans[n][W];
```

- Generally, you need to be careful when using the non-recursive implementation — when computing a state, all the other states it depends on must be ready

Recursive vs. non-recursive version

- **Recursive version reflects the "memoization" part of DP algorithms**
  - If you need some "subproblem", call the function
  - If it's ready, read the result, if not, compute it and save it in the DP array
- **Non-recursive version directly computes the elements in the array**
  - Usually using for-loops to fill in the numbers in your DP table
  - More widely-used in many classic problems, slightly faster, and can be optimized easily
- **Sometimes it would be difficult to directly find a non-recursive solution**
- **But you'll find the recursive version using "memoization" is very straightforward!**

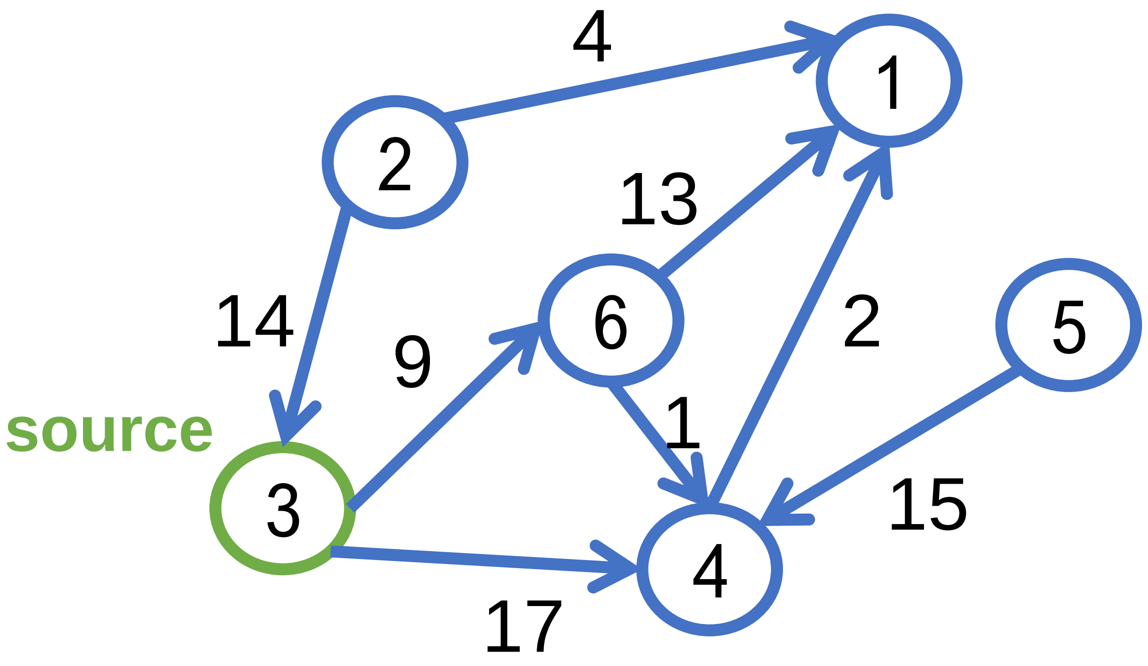
In this lecture

- Single source shortest path algorithm on DAGs
- Matrix multiplication chain

Single source shortest path algorithm on DAGs

Single source shortest path algorithm on DAGs

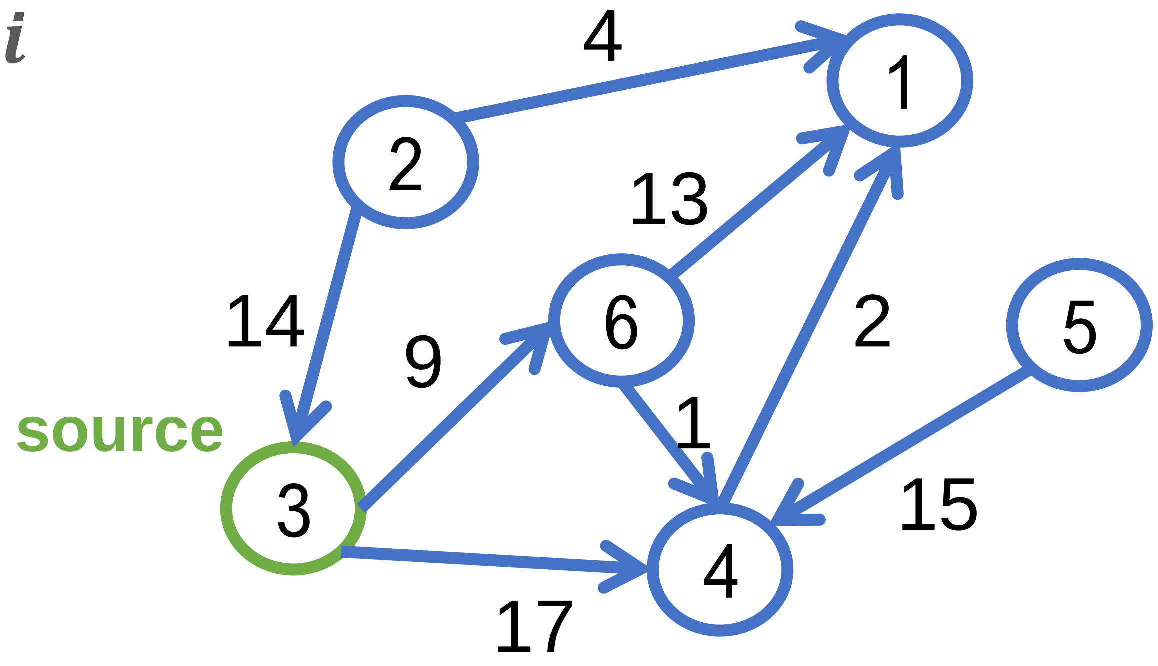
- **DAG: Directed acyclic graph**
  - Directed: every edge has a direction
  - Acyclic: no cycles formed
- **We want to find the shortest distance from s to all other vertices**
  - Some maybe unreachable, distance = ∞
  - Assume all weights are positive



Single source shortest path algorithm on DAGs

- **Consider the shortest distance from 3 to 1**
  - It can only be from 6 or 4
  - If it's from 6: how should we arrive at 6?
  - We should also take the shortest path to 6!!
  - Same for 4
- **Let $D[i]$ be the shortest distance to i**

$$D[i] = \min_{j\ is\ pred\ of\ i} (D[j] + dis[i,j])$$

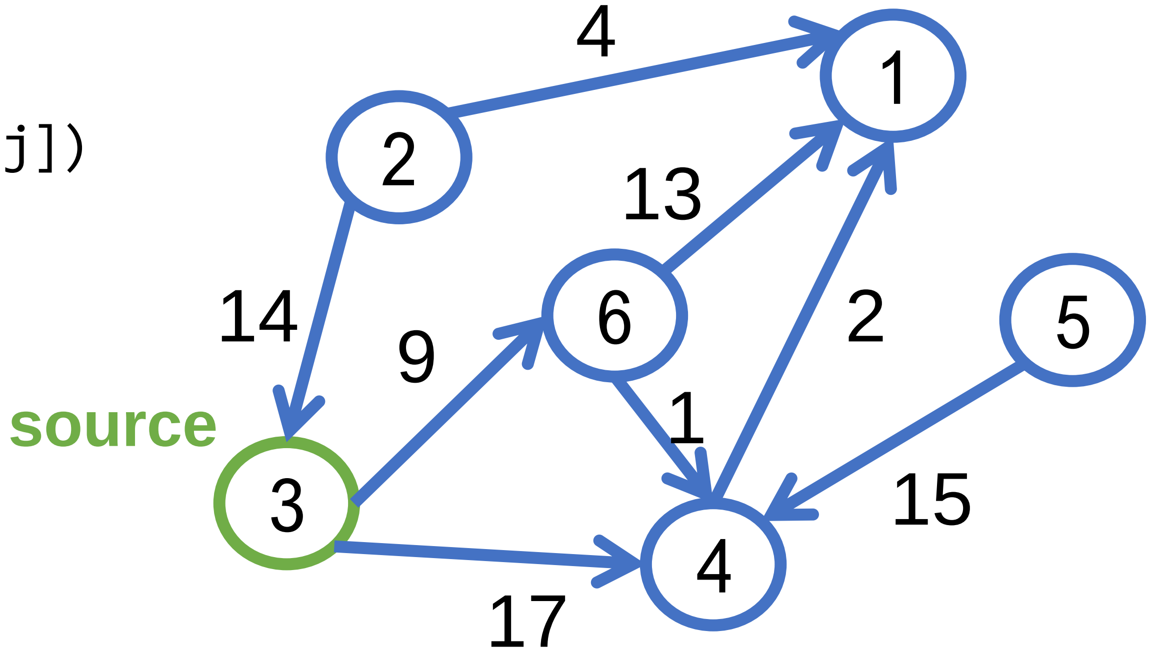


Single source shortest path algorithm on DAGs

- $D[i] = \min_{j \text{ is pred of } i} (D[j] + dis[i,j])$
- **OK we have a DP recurrence, but how to compute it in algorithms?**

i	1	2	3	4	5	6
D[i]	$+\infty$	$+\infty$	0	$+\infty$	$+\infty$	$+\infty$

```
Initialize D[ ] to be +infty
D[s] = 0;
for i = 1 to n
    foreach j as i's predecessor
        D[i] = min(D[i], D[j] + dis[i,j])
```



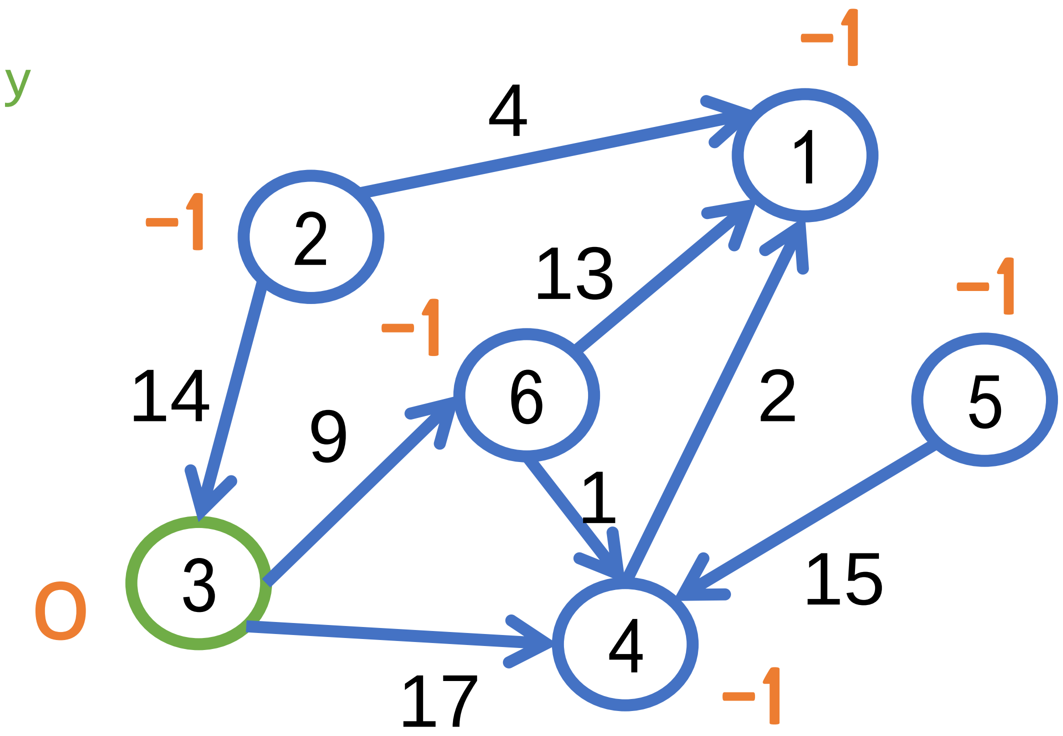
Will it compute the distance correctly?
When we try to compute D[i], are all relevant D[j] ready?

SSSP on DAGs: memoization

- Let’s go back to memoization!

i	1	2	3	4	5	6
D[i]	-1	-1	0	-1	-1	-1

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i’s predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

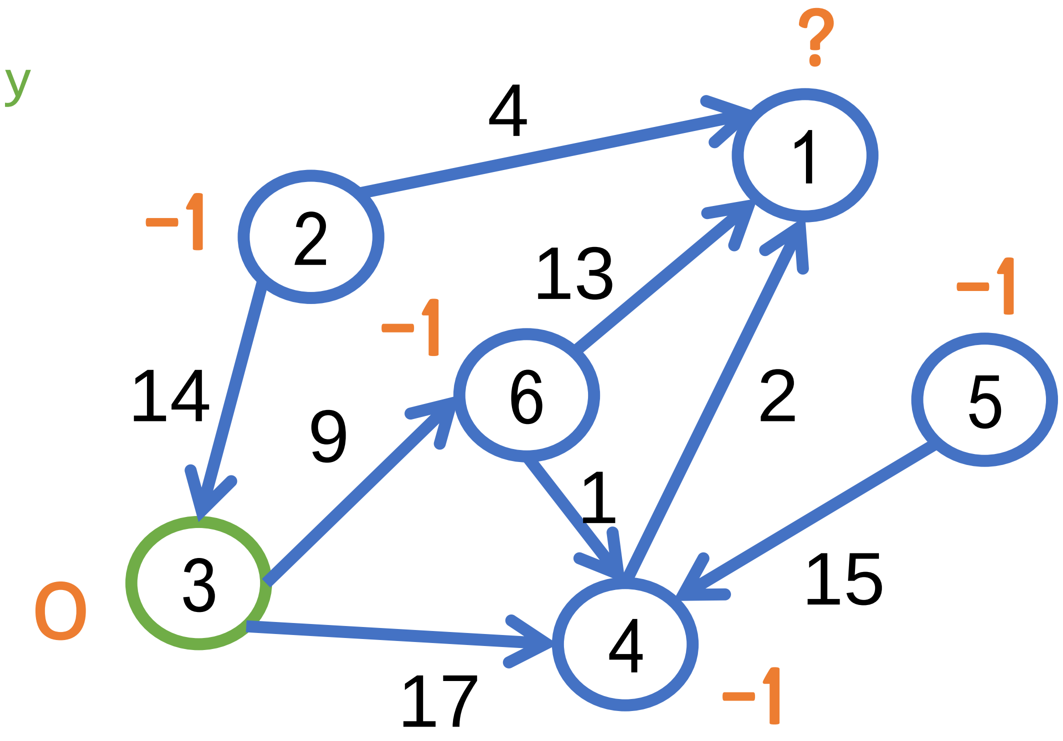


SSSP on DAGs: memoization

- Let's go back to memoization!

i	1	2	3	4	5	6
D[i]	-1	-1	0	-1	-1	-1

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

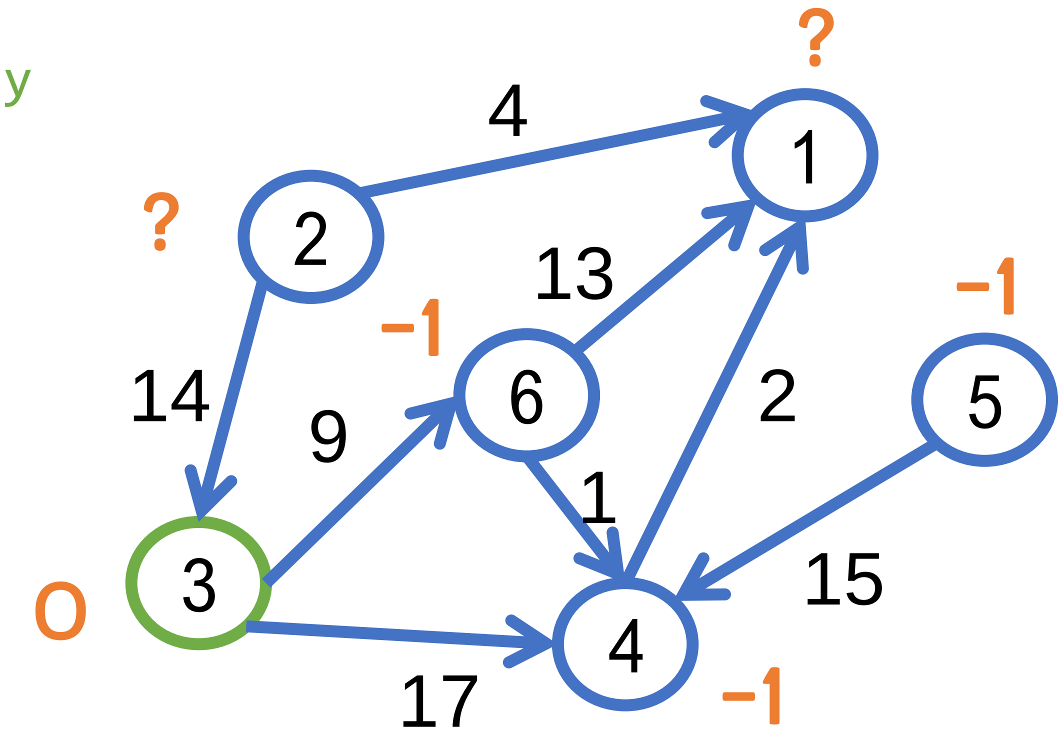


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	-1	0	-1	-1	-1

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

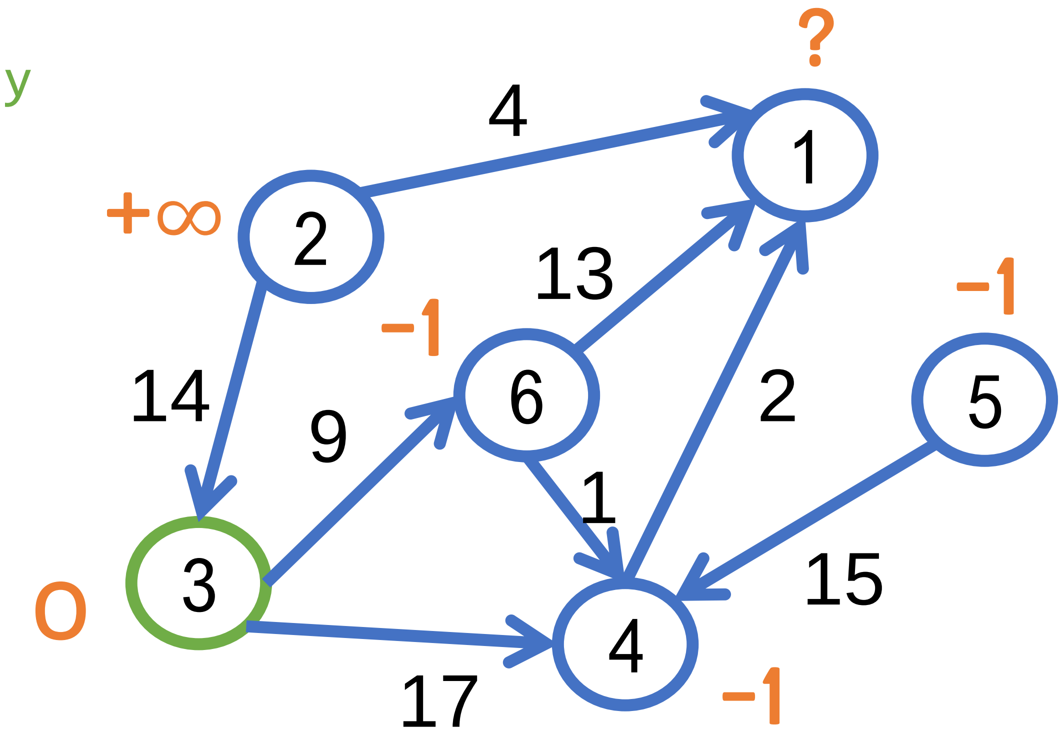


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	-1

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

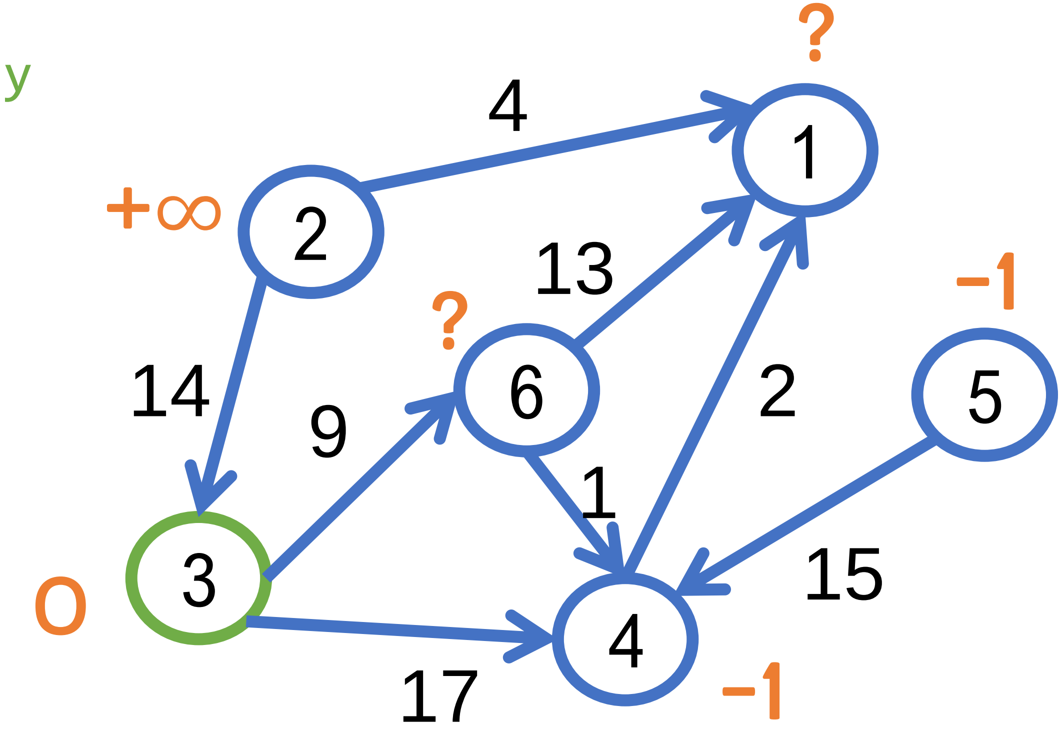


SSSP on DAGs: memoization

- Let's go back to memoization!

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	-1

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

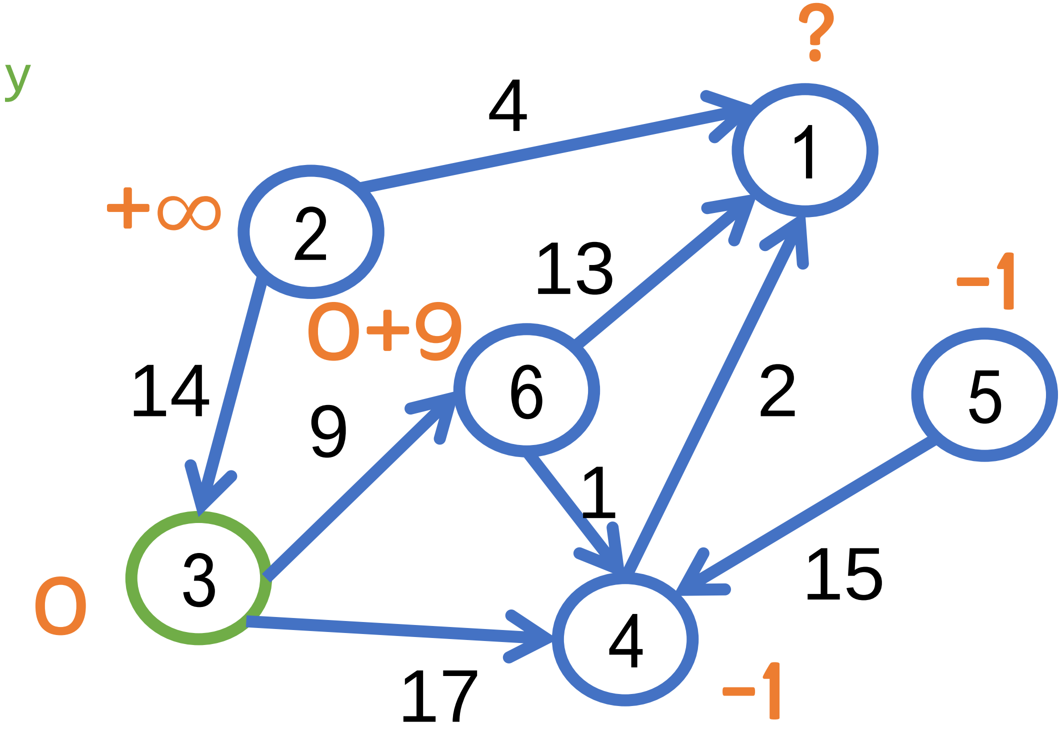


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	-1

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

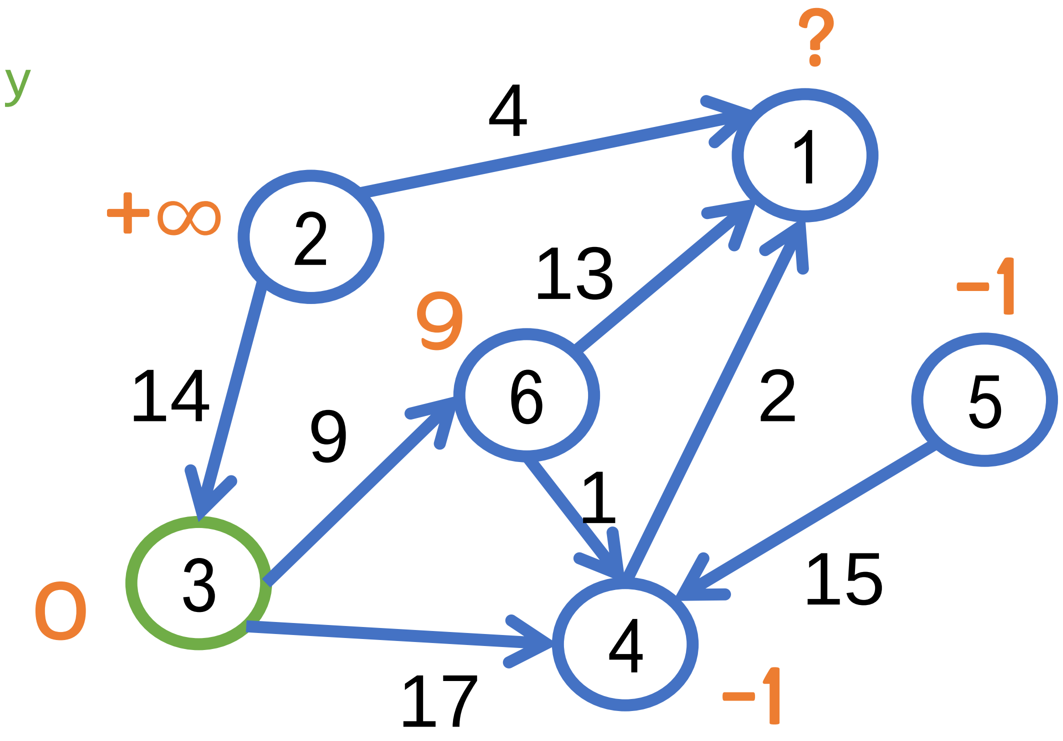


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

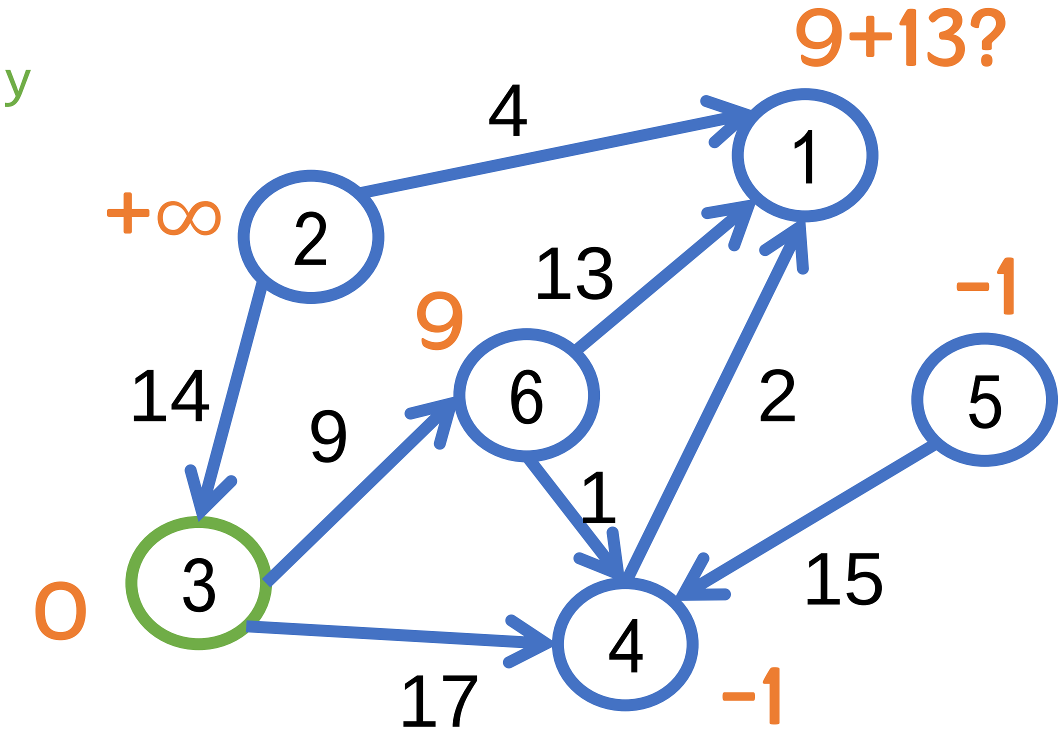


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

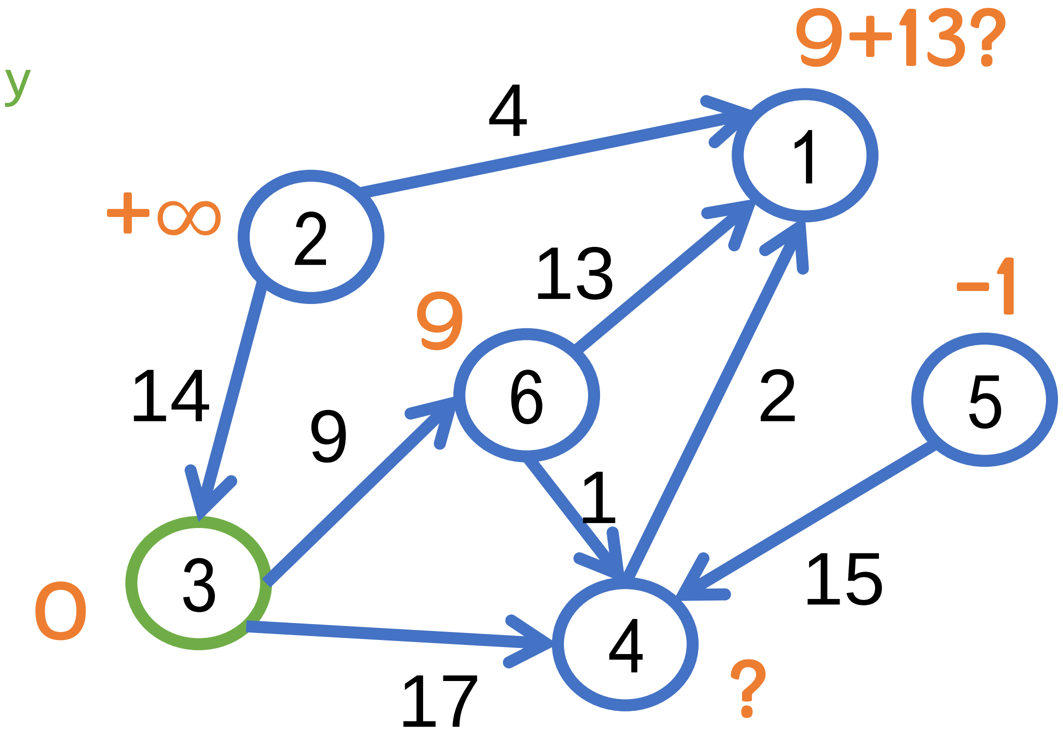


SSSP on DAGs: memoization

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	9

- **Let's go back to memoization!**

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

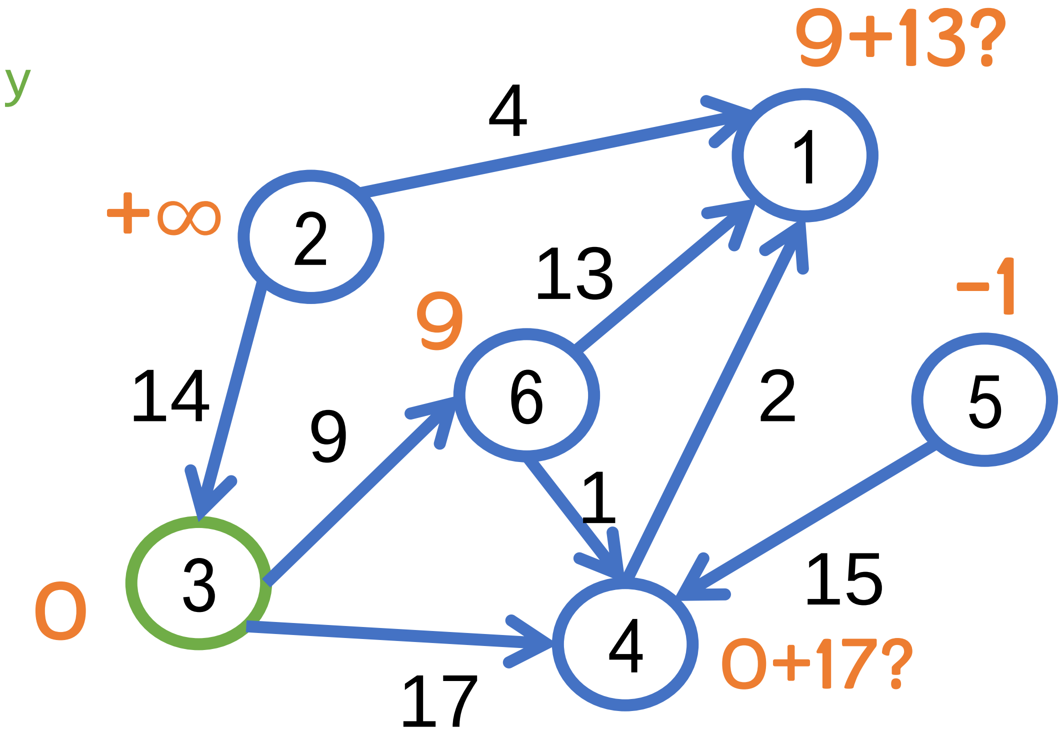


SSSP on DAGs: memoization

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	9

- Let's go back to memoization!

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

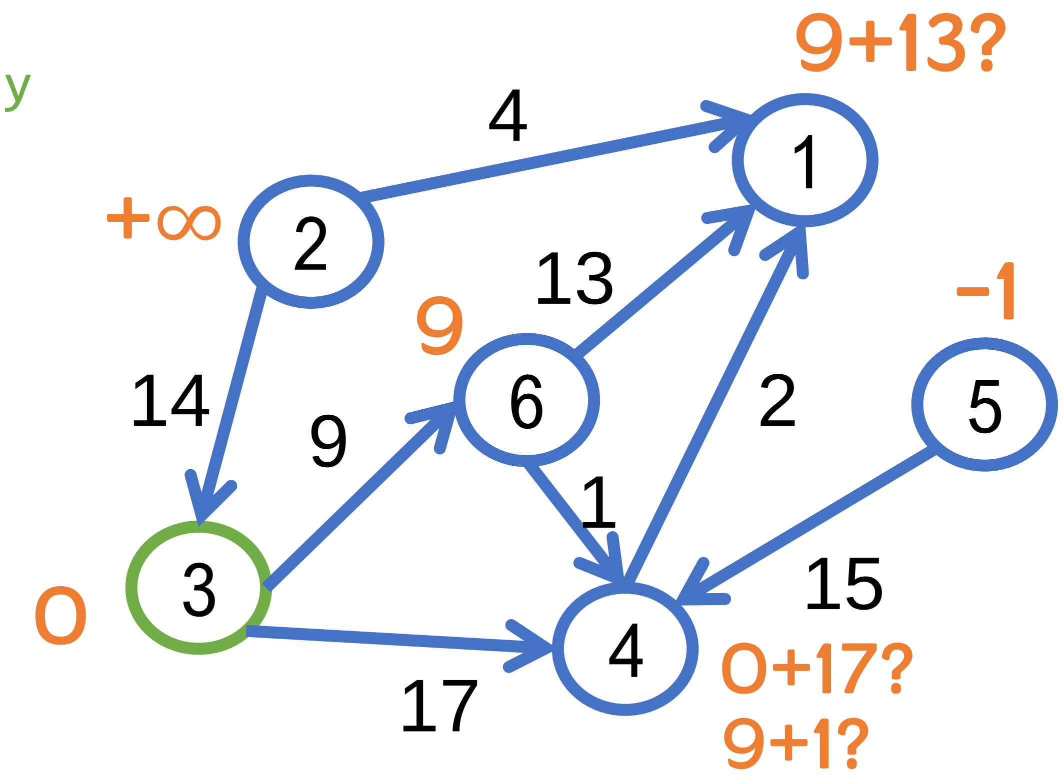


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

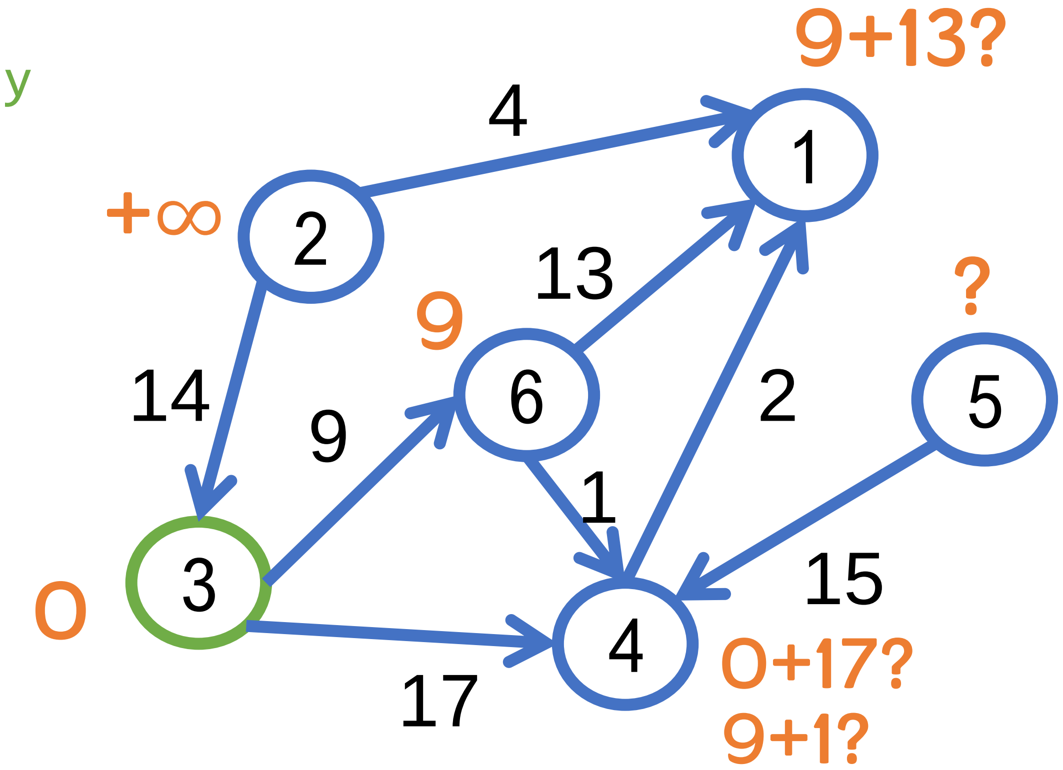


SSSP on DAGs: memoization

- Let's go back to memoization!

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	-1	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i’s predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

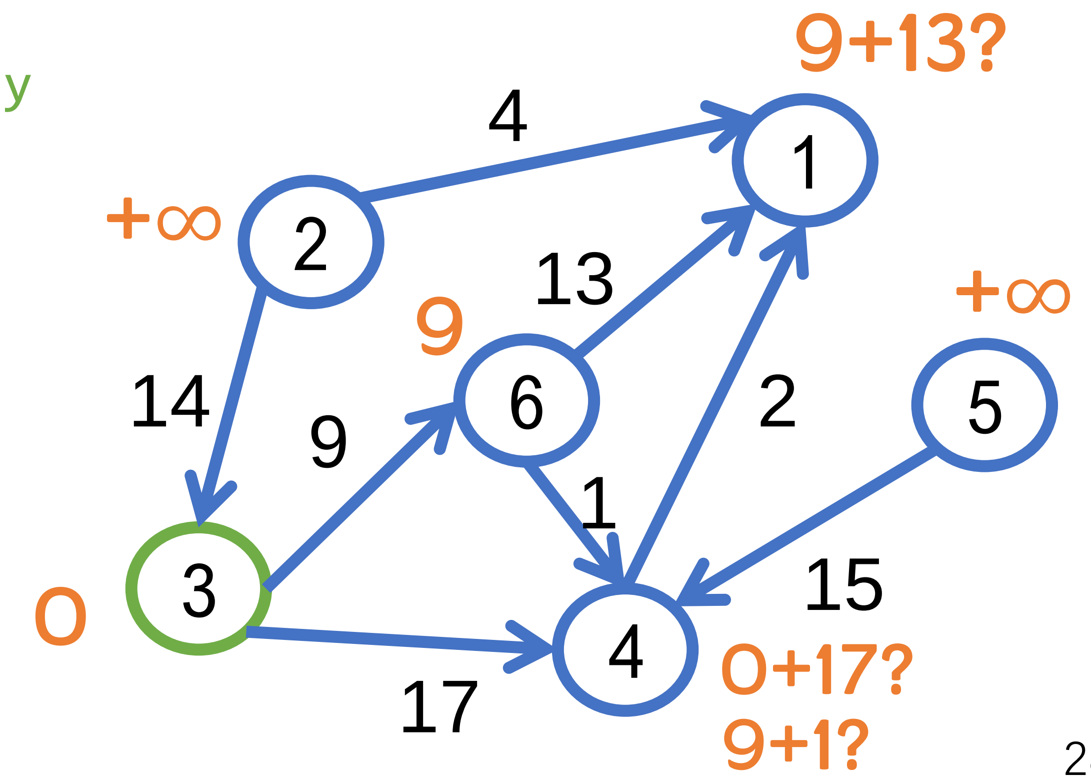


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	-1	+∞	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

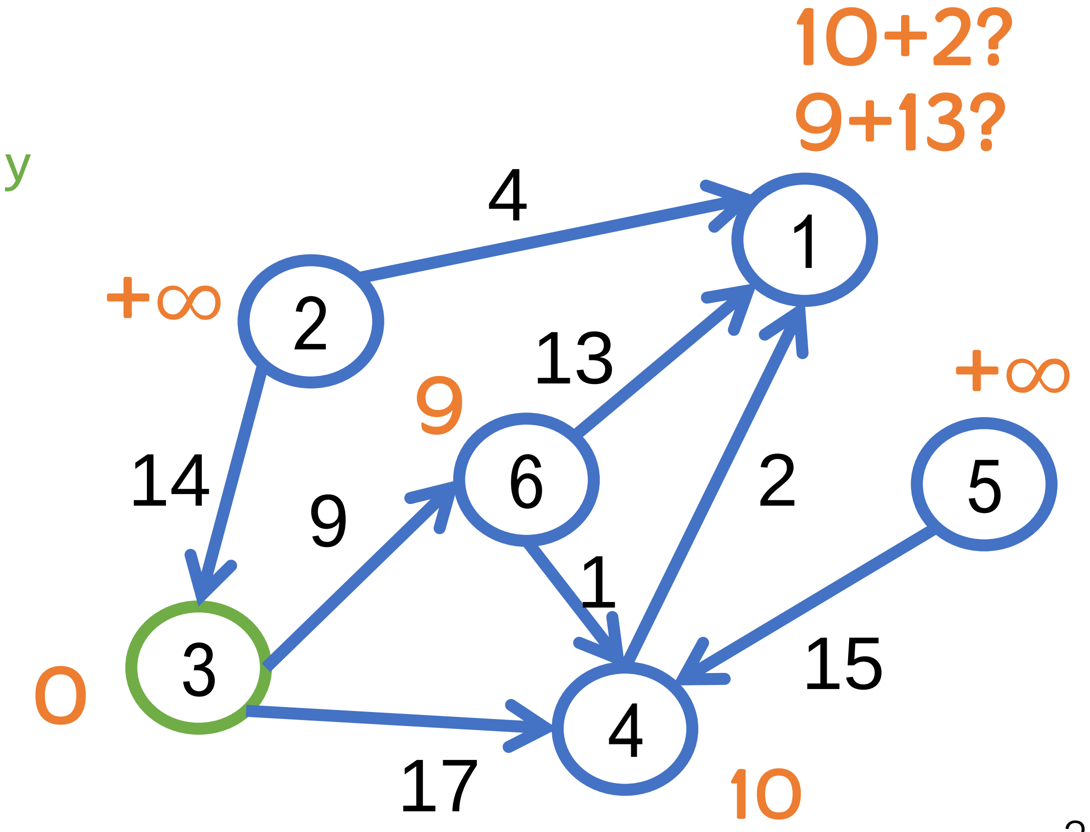


SSSP on DAGs: memoization

- **Let's go back to memoization!**

i	1	2	3	4	5	6
D[i]	-1	+∞	0	10	+∞	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

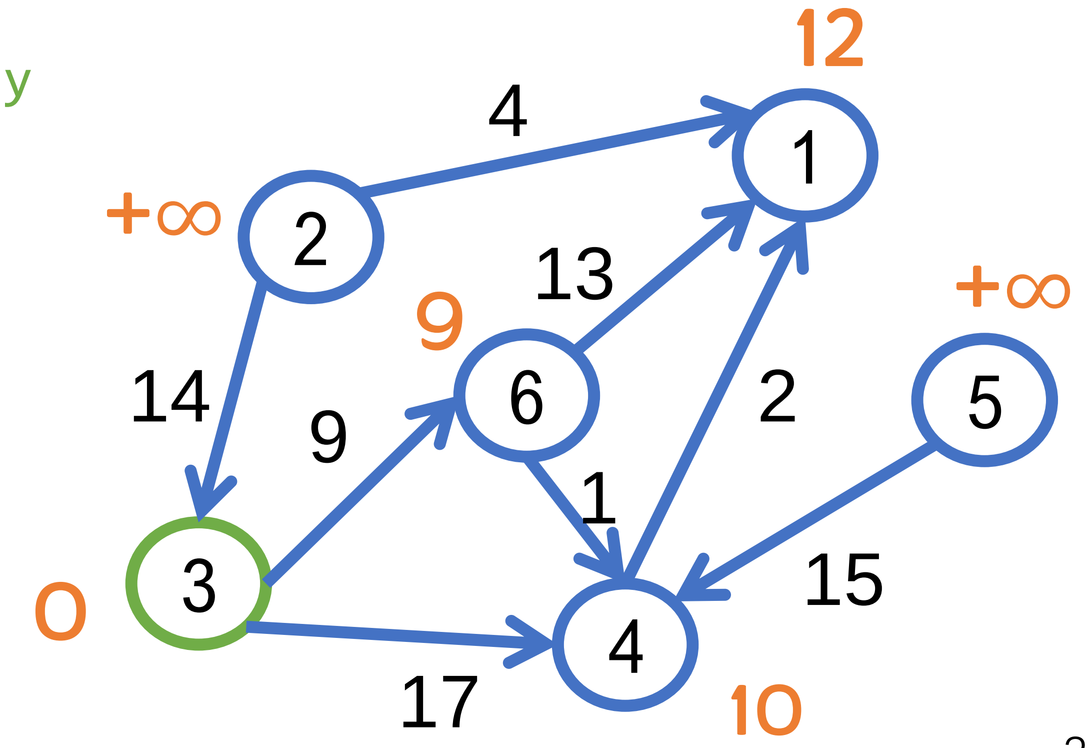


SSSP on DAGs: memoization

- Let's go back to memoization!

i	1	2	3	4	5	6
D[i]	12	+∞	0	10	+∞	9

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```



SSSP on DAGs: memoization

- **Let's go back to memoization!**

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```

$O(m+n)$ time

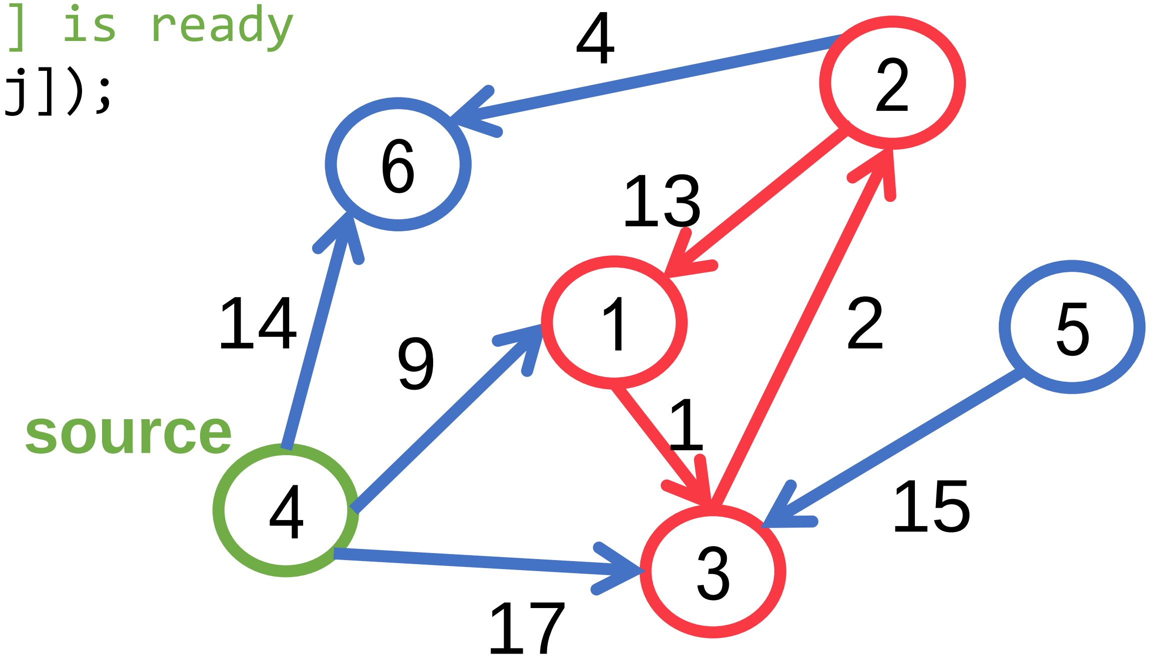
- Every vertex is computed once
- Every edge is used once

SSSP on DAGs

Compute(1) →Compute(2) →Compute(3) →Compute(1) → …

- **What happens if it's not a DAG?**

```
Function compute(node i) { //compute SSSP to i
  if (D[i] is not -1) return D[i]; // if memoized, directly return
  tmp = +infty;
  foreach j as i's predecessor {
    compute(j); // make sure D[j] is ready
    tmp = min(tmp, D[j] + dis[i,j]);
  }
  D[i] = tmp;
}

Initialize D[ ] to be -1
D[s] = 0;
for i = 1 to n
  compute(i);
```



Dynamic programming + memoization

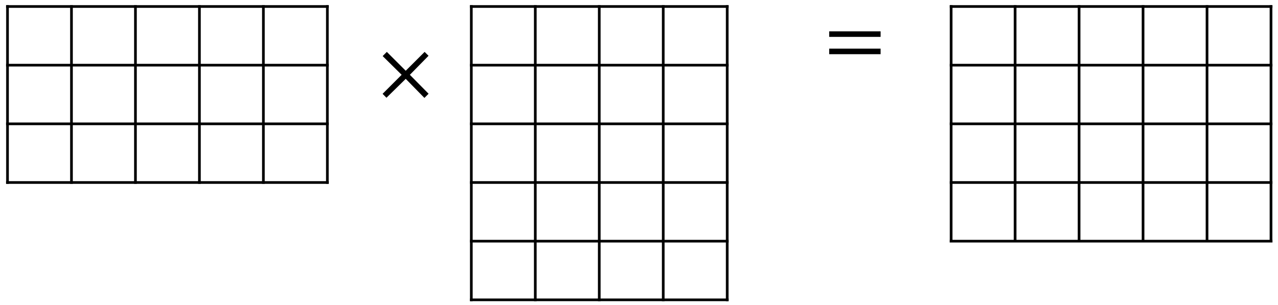
- **States must for a DAG (no cycle of dependency …)**

- **Then we can just try to compute each state**
- **If the state s has been computed, directly return! (use memorized results!)**
- **If the state s is not ready, compute it**
  - Look at all other states it depend on
  - Compute them (ready ones will be returned directly, otherwise we compute on the fly!)
  - Use the DP recurrence to compute the DP value for s

Matrix multiplication chain

Matrix multiplication chain

- **Matrix multiplication on two matrices $\boldsymbol{a \times b}$ and $\boldsymbol{c \times d}$**
  - b must equal to c
  - Getting a new matrix of $a \times d$
  - Total cost is $a \times b \times d$

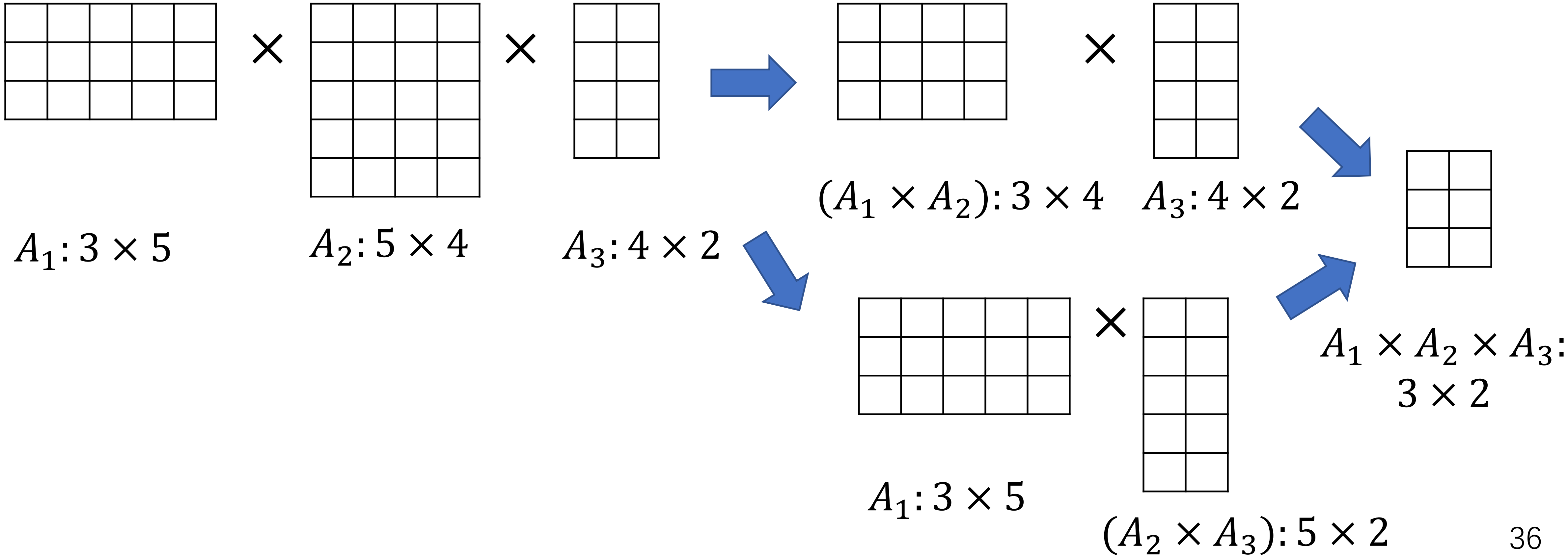


$A_1 : 3 \times 5$ $A_2 : 5 \times 4$ $B : 3 \times 4$

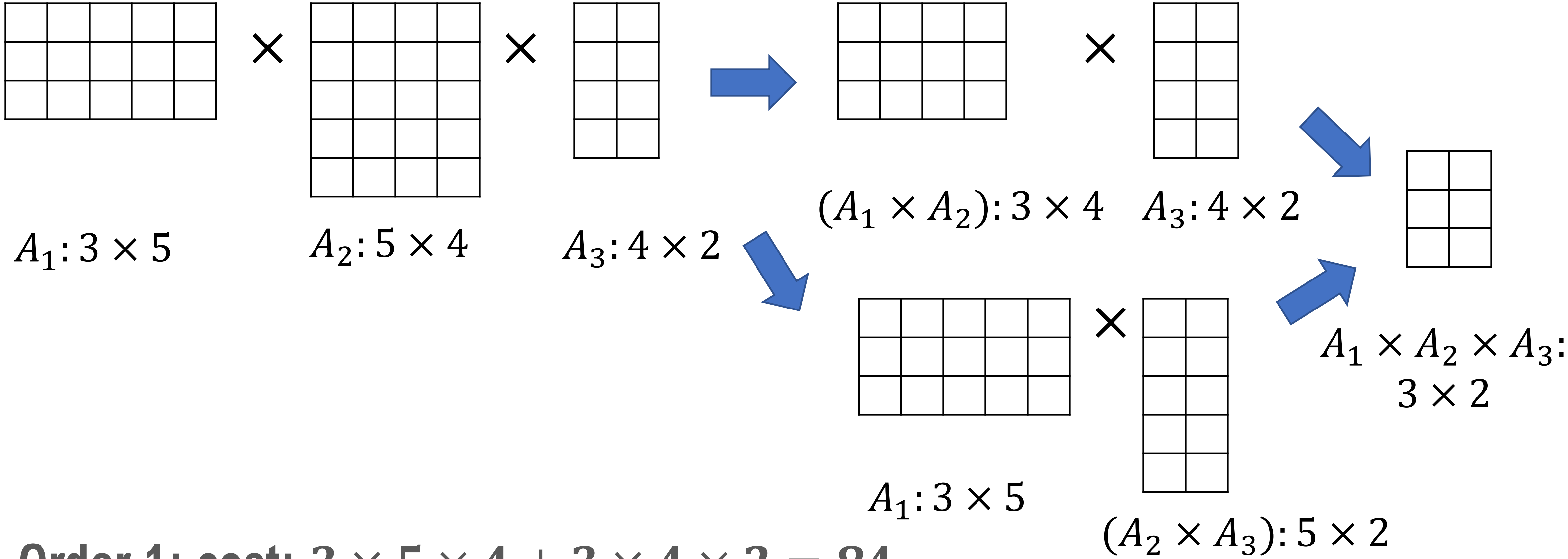
Need $3 \times 5 \times 4 = 60$ multiplications

Matrix multiplication chain

- What if we have three matrices?
- What is the cost?
- Associativity: $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$



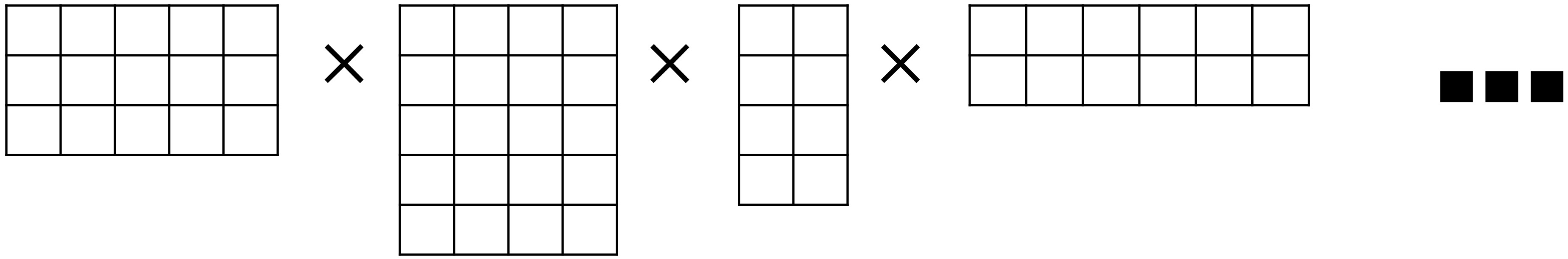
Matrix multiplication chain



- **Order 1: cost: $3 \times 5 \times 4 + 3 \times 4 \times 2 = 84$**
- **Order 2: cost: $5 \times 4 \times 2 + 3 \times 5 \times 2 = 70$**
- **What is the smallest cost?**

Matrix multiplication chain

- **What if we have multiple matrices?**
  - $A_1 \times A_2 \times A_3 \times \cdots A_n$
  - Assume matrix A_i is $a[i]$ by $a[i+1]$
  - ($a[1] \times a[2]$ matrix) $\cdot$ ($a[2] \times a[3]$ matrix) $\cdot$ ($a[3] \times a[4]$ matrix) ….



$A_1: 3 \times 5$ $A_2: 5 \times 4$ $A_3: 4 \times 2$ $A_4: 2 \times 6$

Matrix multiplication chain

- **Compute the product of a list of matrices**
  - Cost can be different because of order
  - Associativity: $(A_1 \times A_2) \times A_3 = A_1 \times (A_2 \times A_3)$
  - We can add paratheses arbitrarily
    - $(A_1 \times A_2) \times (A_3 \times A_4) \times A_5 \times \cdots$
    - $A_1 \times (A_2 \times (A_3 \times A_4)) \times A_5 \times \cdots$
    - $A_1 \times ((A_2 \times A_3) \times A_4) \times A_5 \times \cdots$
    - $A_1 \times ((A_2 \times A_3) \times (A_4 \times A_5)) \times \cdots$

What is the smallest cost?

$$\underbrace{A_1 \quad A_2 \quad A_3 \quad \cdots \quad A_k}_{\text{A matrix } a[1] \times a[k+1]} \quad \underbrace{A_{k+1} \quad \cdots \quad A_{n-1}}_{\text{A matrix } a[k+1] \times a[n]}$$

- **How many possibilities of different orders?**
- **Consider the last multiply, if its $A' = \prod_{i=1}^{k} A_i$ and $A'' = \prod_{i=k+1}^{n-1} A_i$**
- **Then the cost of the last multiplication must be $a[1] \times a[k+1] \times a[n]$**
- **Then what is the cost to get A' and A''?**

- **They are also matrix multiply chains!**

What is the smallest cost?

A_1 A_2 A_3 $\cdots$ A_k A_{k+1} $\cdots$ A_{n-1}

- Let $f[i,j]$ be the smallest cost of getting the product from A_i to A_j
- $f[i,j] = \min\limits_{k\in[i,j)} \left(f[i,k] + f[k+1,j] + a[i] \times a[k+1] \times a[j+1] \right)$

Matrix multiply from A_l to A_k

Matrix multiply from A_{k+1} to A_r

The cost of the last multiply

- f[i,i] = 0

DP recurrence

A_1 A_2 A_3 $\cdots$ A_k A_{k+1} $\cdots$ A_{n-1}

- **Let $f[i,j]$ be the smallest cost of getting the product from A_l to A_r**
- $f[i,j] = \min\limits_{k\in[i,j)} \left(f[i,k] + f[k+1,j] + a[i] \times a[k+1] \times a[j+1] \right)$

```
Initialize f[ ] to be +infty
For all i, f[i,i] = 0;
for i = 1 to n-1
  for j = i+1 to n-1
    for k = i to j-1
        f[i,j] = min(f[i,j],f[i,k]+f[k+1,j]+a[i]*a[k+1]*a[j+1])
Output f[1,n]
```

When compute f[1,5], we may need f[2,5], f[3,5], …

DP algorithm: memoization

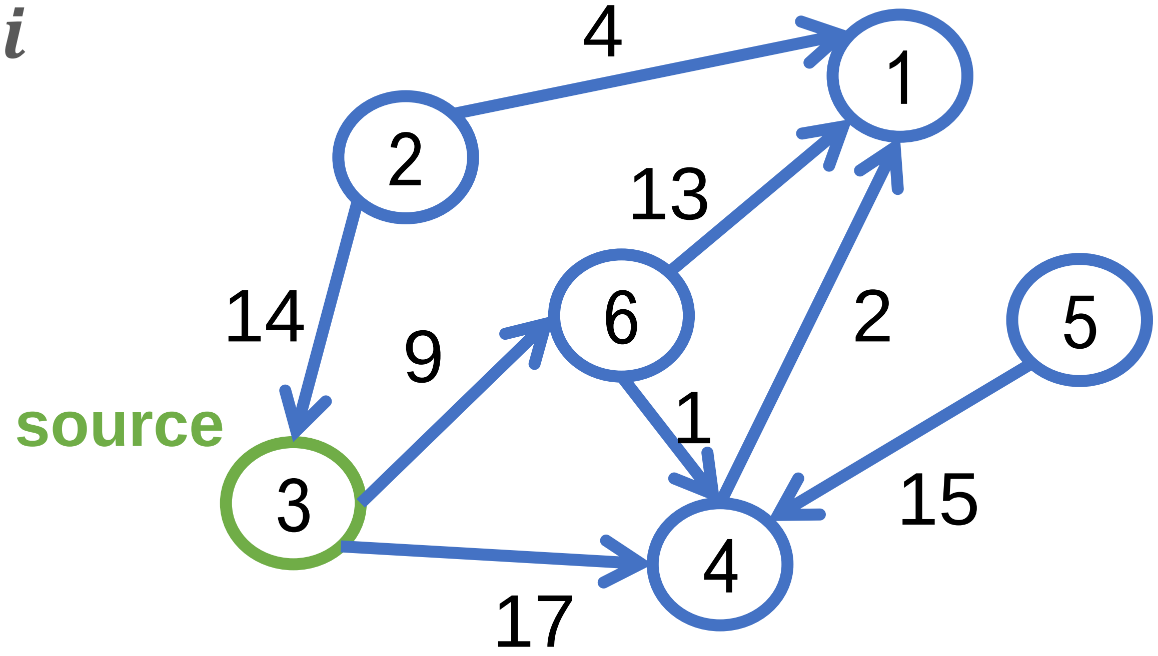
```
Function compute(int i, j) {
  if (f[i,j] is not -1) return f[i,j];
  f[i,j] = +infty;
  for k = i to j-1 {
    compute(i,k); // make sure we have f[i,k]
    compute(k+1,j);  // make sure we have f[k+1,j]
    f[i,j] = min(f[i,j], f[i,k] + f[k+1,j] + a[i]*a[k+1]*a[j+1]);
  }
  return f[i,j];
}

Initialize f[ ] to be -1
For all i, f[i,i] = 0;
for i = 1 to n-1
  for j = i+1 to n-1
    compute(i,j);
Output f[1,n]
```

Can we still design non-recursive algorithms?

Single source shortest path algorithm on DAGs

- **Consider the shortest distance from 3 to 1**
  - It can only be from 6 or 4
  - If it's from 6: how should we arrive at 6?
  - We should also take the shortest path to 6!!
  - Same for 4
- **Let $D[i]$ be the shortest distance to i**

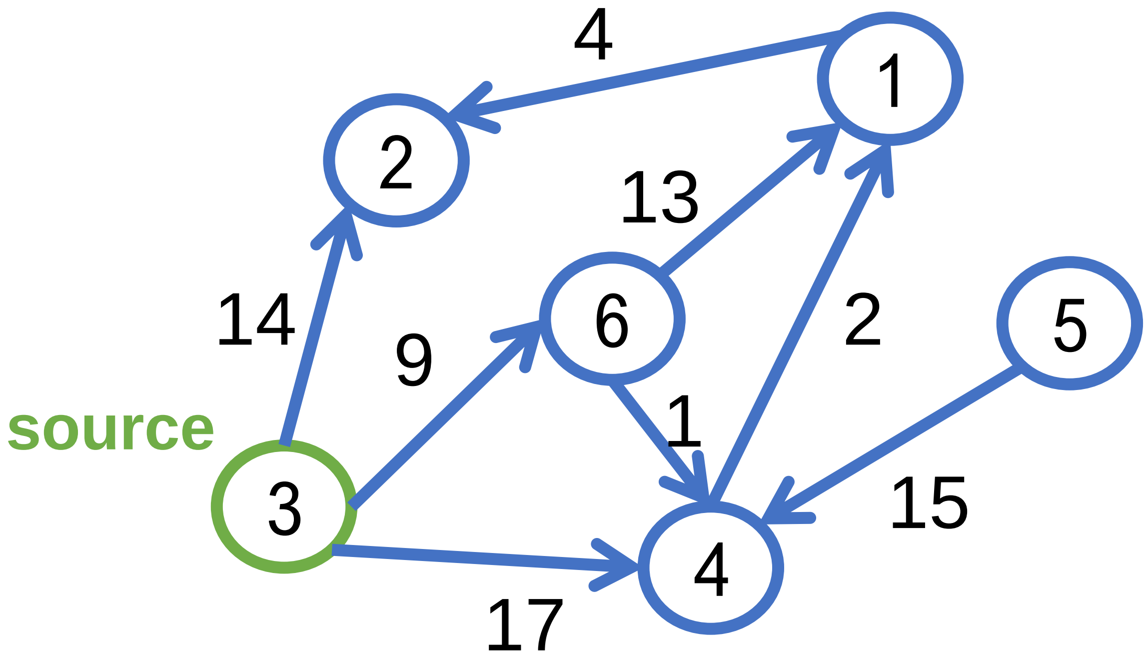
$$D[i] = \min_{j \text{ is pred of } i} (D[j] + dis[i, j])$$



Let's go back to the non-recursive version again…

- **What happens after the algorithm?**
  - Some D[i] are not the final answer: after we compute D[i], some of its predecessors j have D[j] updated to smaller values
  - But we didn't use it to update D[i]
- **Can we make it work?**
- **Method 1: compute the right order**

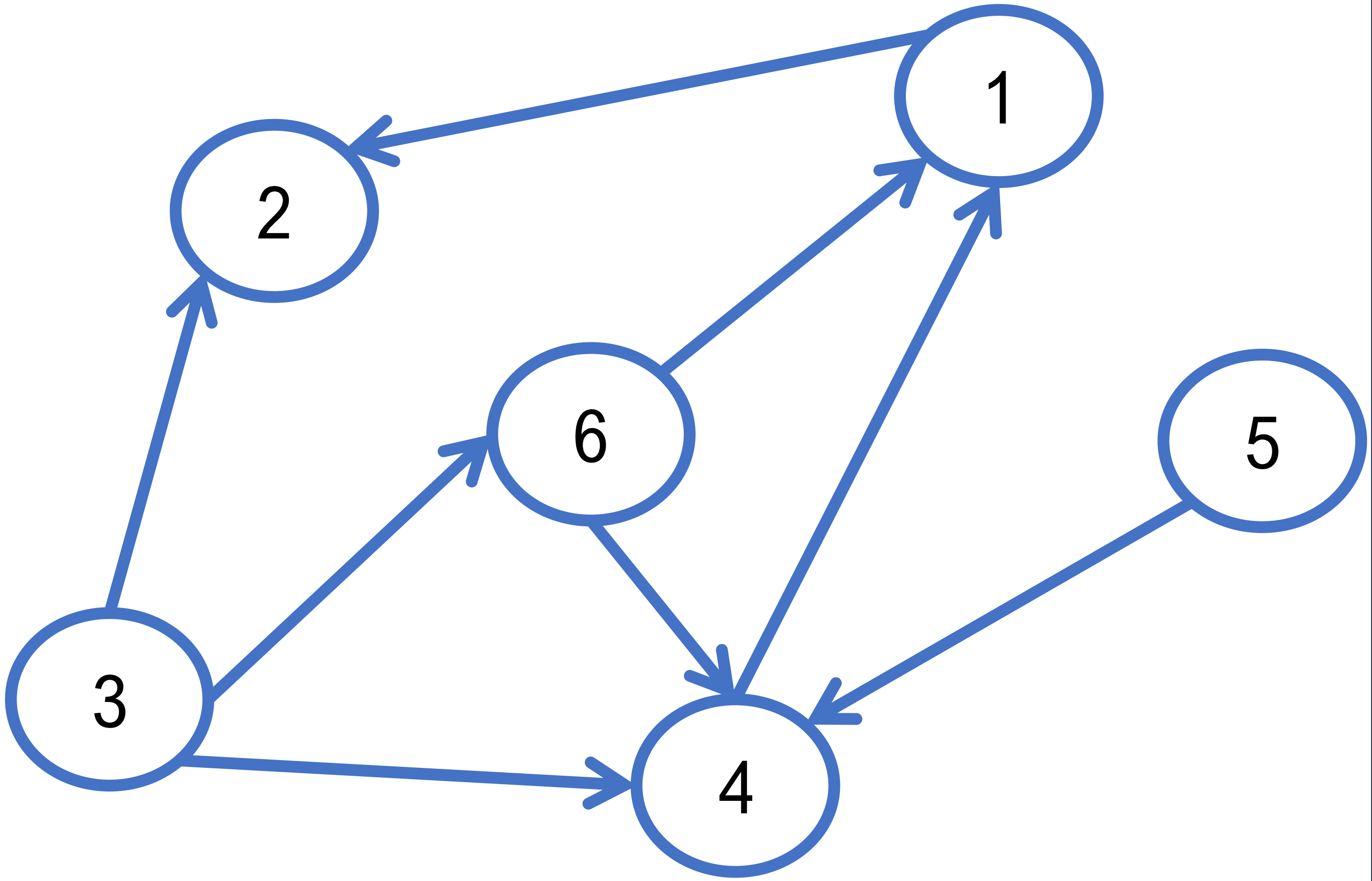
```
Initialize D[ ] to be +infty
D[s] = 0;

for i = 1 to n
    foreach j as i's predecessor
        D[i] = min(D[i], D[j] + dis[i,j])
```



Topological sort

Repeat until G is empty:

1. Choose a vertex v with in-degree 0
2. Output v
3. Remove v and all its edges

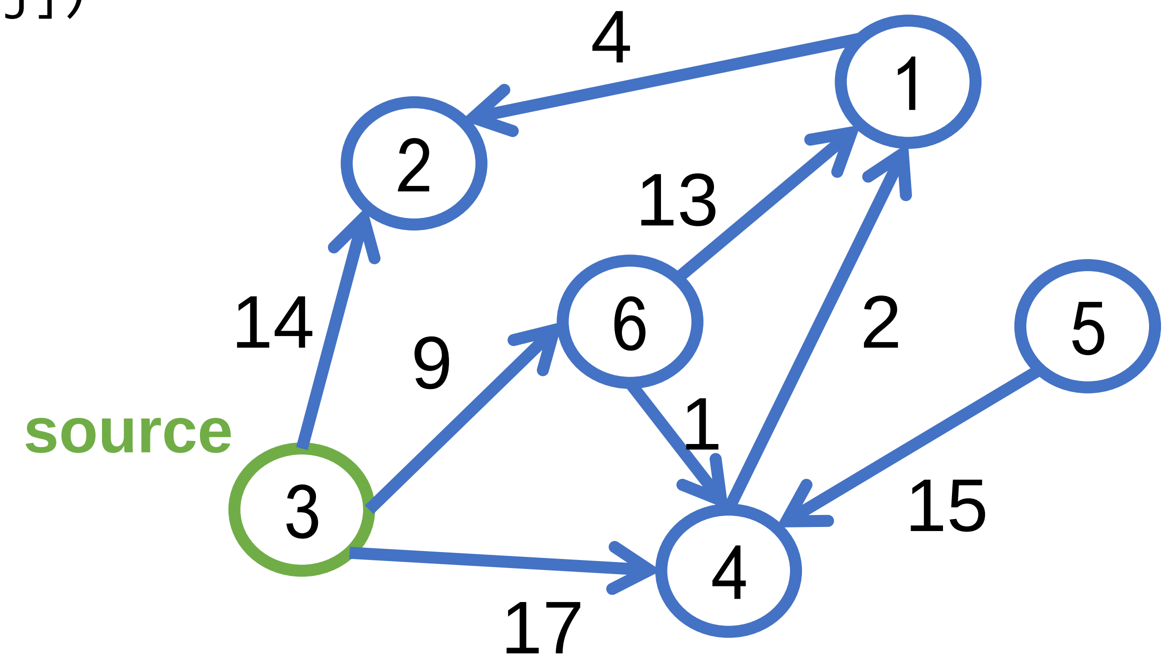


3 6 5 4 1 2

Let's go back to the non-recursive version again…

```
Initialize D[ ] to be +infty
D[s] = 0;

V' = topological_sort(V, E);
for k = 1 to n {
    i = V'[k];
    foreach j as i's predecessor
        D[i] = min(D[i], D[j] + dis[i,j])
}
```

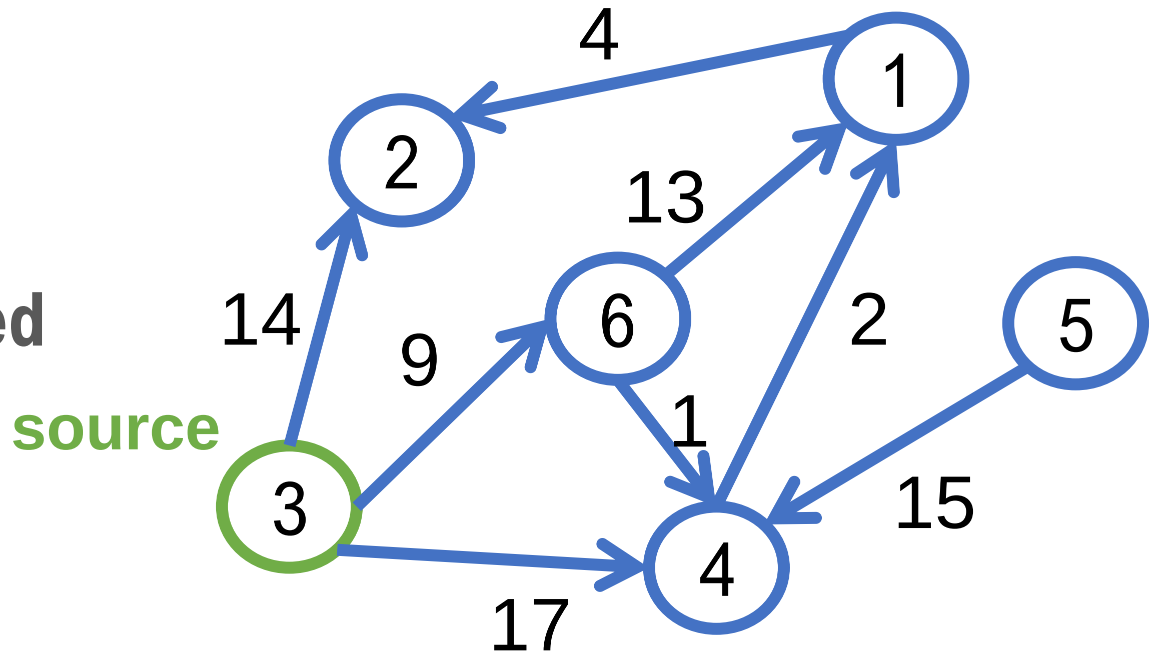
3 6 5 4 1 2



Let's go back to the non-recursive version again…

- **What happens after the algorithm?**
  - Some D[i] are not the final answer: after we compute D[i], some of its predecessors j have D[j] updated to smaller values
  - But we didn't use it to update D[i]
- **Can we make it work?**
- **Method 2: repeatedly do this!**
- **Bellman-Ford algorithm!**
- **Also OK for general graphs (no need to be a DAG)**

```
Initialize D[ ] to be +infty
D[s] = 0;

for i = 1 to n
    foreach j as i's predecessor
        D[i] = min(D[i], D[j] + dis[i,j])
```



Matrix Multiplication Chain

A_1 A_2 A_3 $\cdots$ A_k A_{k+1} $\cdots$ A_{n-1}

- Let $f[i,j]$ be the smallest cost of getting the product from A_i to A_j
- $f[i,j] = \min\limits_{k\in[i,j)} \left(f[i,k] + f[k+1,j] + a[i] \times a[k+1] \times a[j+1] \right)$

Matrix multiply from A_l to A_k

Matrix multiply from A_{k+1} to A_r

The cost of the last multiply

- f[i,i] = 0

Can we still use a non-recursive solution?

```
Initialize f[ ] to be +infty
For all i, f[i,i] = 0;
for i = 1 to n-1
  for j = i+1 to n-1
    for k = i to j-1
        f[i,j] = min(f[i,j],f[i,k]+f[k+1,j]+a[i]*a[k+1]*a[j+1])
Output f[1,n]
```

When compute f[1,5], we may need f[2,5], f[3,5], …

- **What is the right order to compute all elements?**

- Boundary f[i,i]=0
- Then we can compute all f[i, i+1]
- Then we can compute all f[i, i+2], since all f[i, j] with |j-i|<2 are ready
- Then we can compute all f[i, i+3], since all f[i, j] with |j-i|<3 are ready
- ……

Can we still use a non-recursive solution?

```
Initialize f[ ] to be +infty
For all i, f[i,i] = 0;
for delta = 1 to n-1
  for i = 1 to n-1-delta {
    j = i+delta;
    for k = i to j-1
        f[i,j] = min(f[i,j],f[i,k]+f[k+1,j]+a[i]*a[k+1]*a[j+1]);
  }
Output f[1,n]
```

- Boundary f[i,i]=0
- Then we can compute all f[i,i+1]
- Then we can compute all f[i,i+2], since all f[i,j] with |j-i|<2 are ready
- Then we can compute all f[i,i+3], since all f[i,j] with |j-i|<3 are ready
- ……

Fun fact for MM chain multiplication

- **A very similar problem: optimal binary search tree**
  - It can be solved in $O(n^2)$ due to monotonicity

- **MM-chain itself can be solved in $\boldsymbol{O(n \log n)}$ time**
  - Using some ideas in triangulating polygons

Summary: memoization

- **SSSP for DAGs**
  - Cannot directly compute all states one by one (not necessary sorted)
  - Memoization, or
  - First determine a right order (topological sort)
  - Do it multiple times: Bellman-Ford
- **Matrix multiplication chain**
  - State: f[i,j] to represent an interval from i to j
  - Cannot directly compute all states one by one
  - Memoization, or
  - Computer based on the order of the difference of j and i

DP on trees

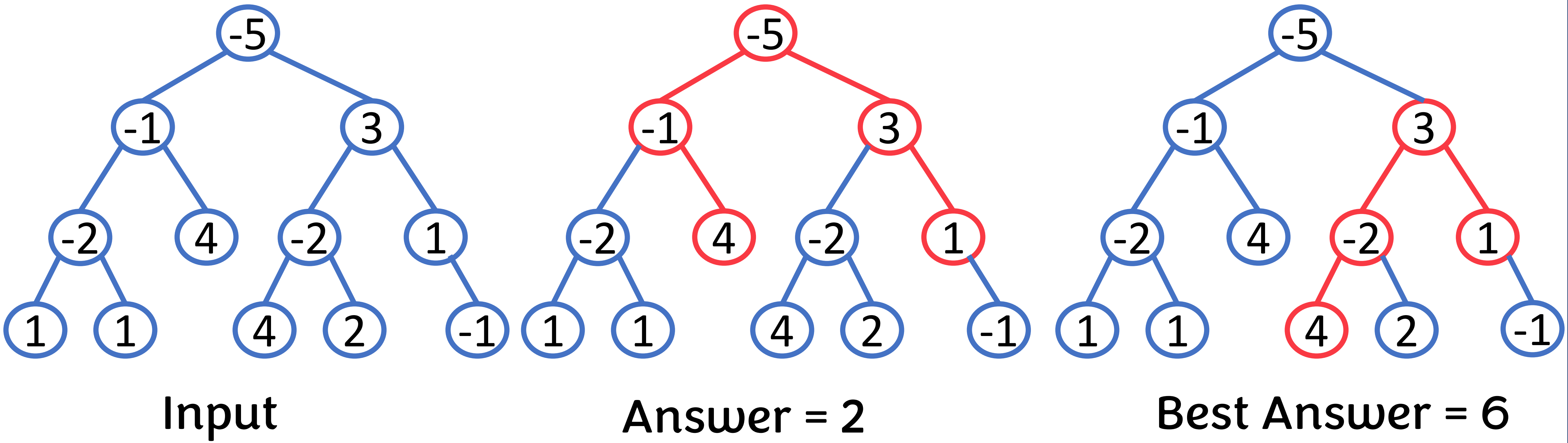
Sometimes we need to deal with a tree structure using dynamic programming

- Well, it’s still dynamic programming, but we can use some small tricks for this special case
- Recall that in the previous class, we said that the “dependency” between states cannot form cycles
- Tree structure is totally fine!

- Usually we can start from the top (root) of the tree
- Usually the state of a node can depend on all its children

Recall the interview problem in the first class…

- Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

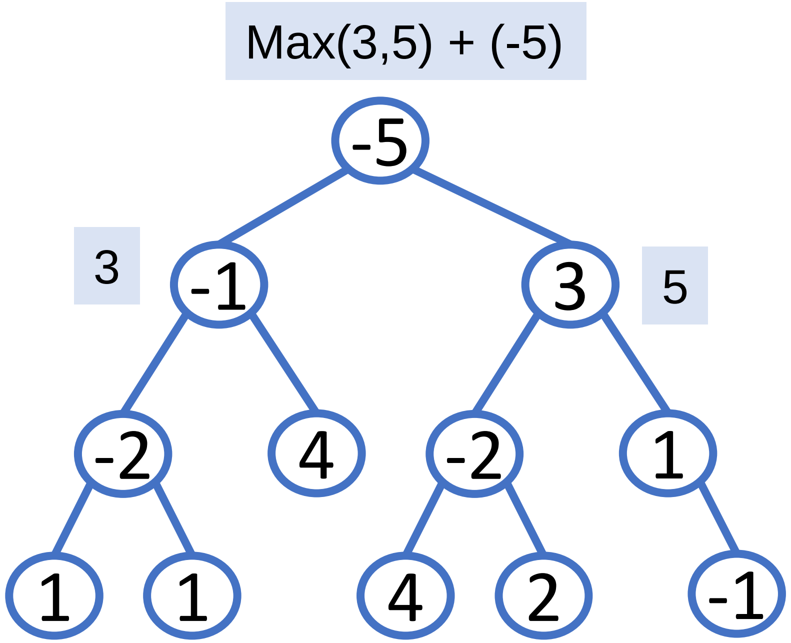


How can we design the state?

- Instead of directly working on the final output, let's define the state as something else…
- Observe: A path first goes up then down
- f[i] = the largest path sum with node i as the topmost node!

- Let j and k be i'th two children
- f[i] = max(f[j] + w[i], f[k] + w[i])

- Is it correct?

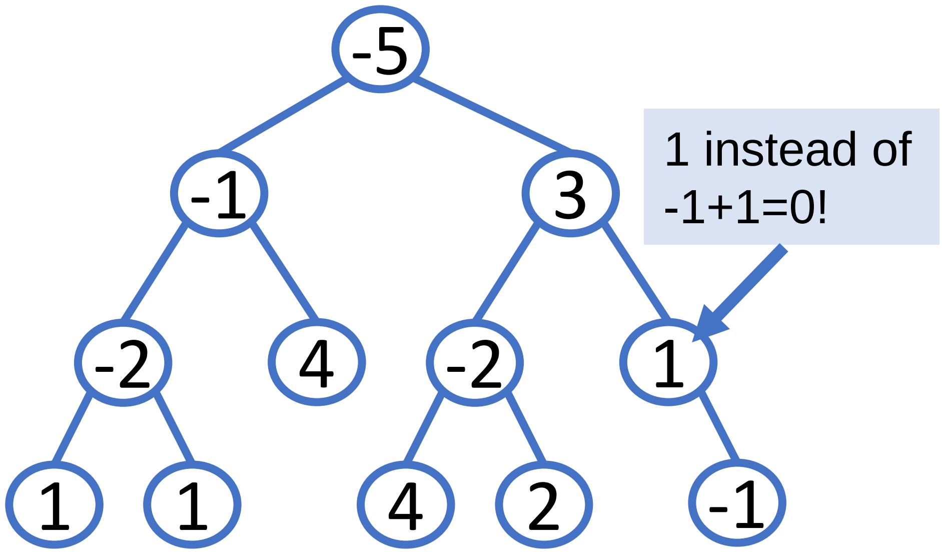


How can we design the state?

- Instead of directly working on the final output, let's define the state as something else…
- Observe: A path first goes up then down
- f[i] = the largest path sum with node i as the topmost node!

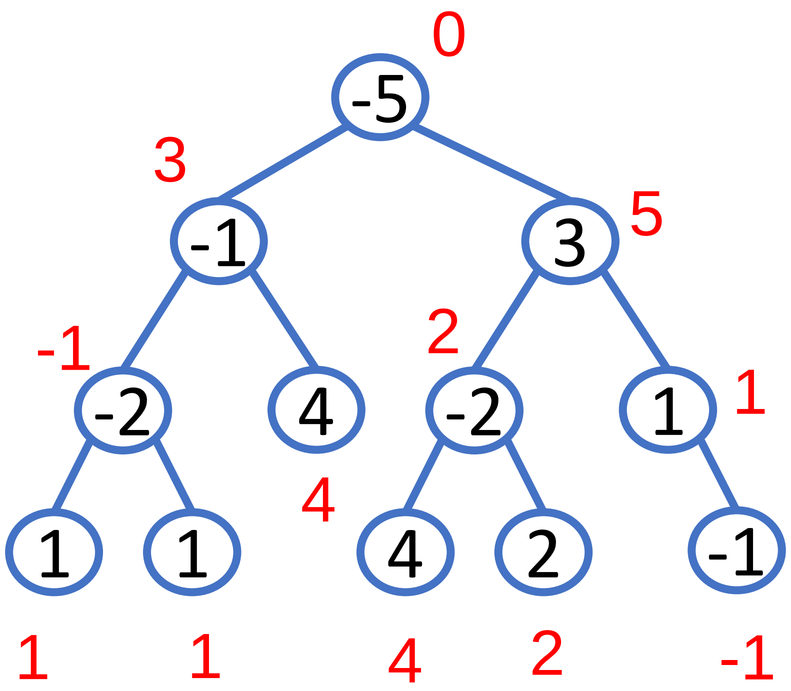
- Let j and k be i'th two children
- f[i] = max(f[j] + w[i], f[k] + w[i], **w[i]**)

- Must consider all cases:
  the path can be just i!



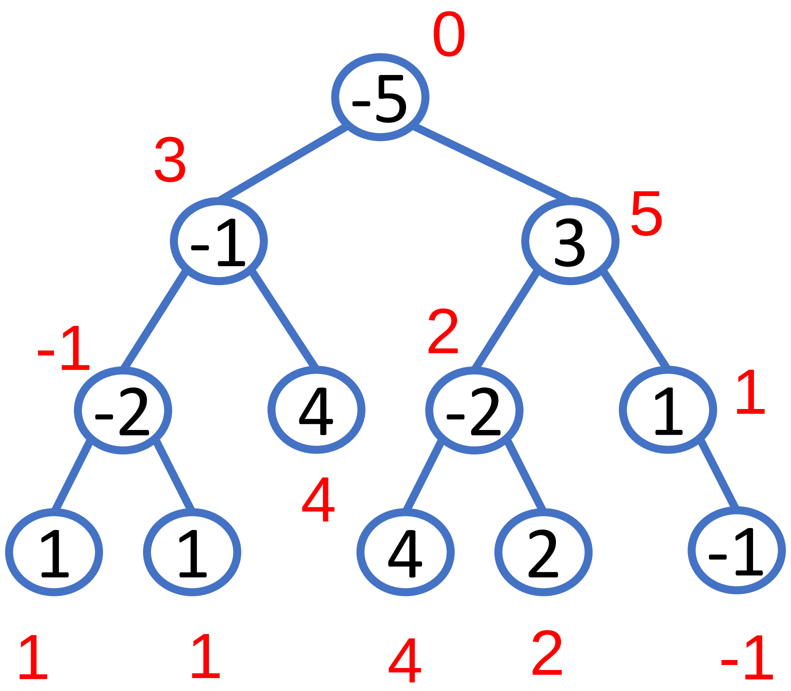
How can we design the state?

- f[i] = max(f[j] + w[i], f[k] + w[i], **w[i]**)



How can we design the state?

- With f[i], we can enumerate all nodes as the "shallowest" node




```
ans = -infty
foreach tree node i {
  let j and k be its two children;
  ans = max(ans, f[j]+f[k]+w[i]);
}
Output ans
```

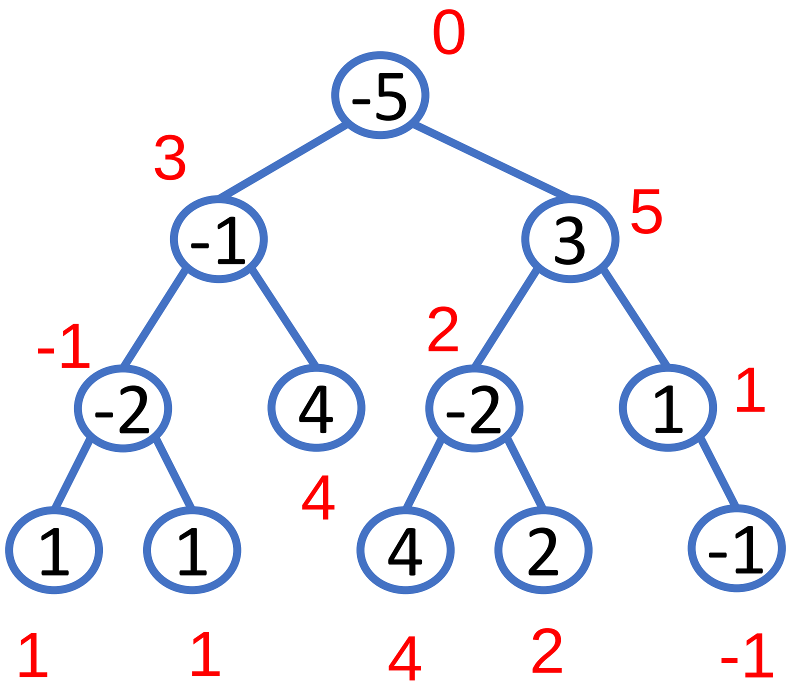
Is this correct?

Let j and k be the two children of i, the best path across node i is:

f[j] + f[k] + w[i]

How can we design the state?

- **Again, consider all cases! Maybe it only contains one side of the branch!**




```
ans = -infty
foreach tree node i {
  let j and k be its two children;
  ans = ……
}
Output ans
```

A simpler solution: allow f[i] to be max(f[i], 0) (you can think about how to do this, and potential issues of doing this)

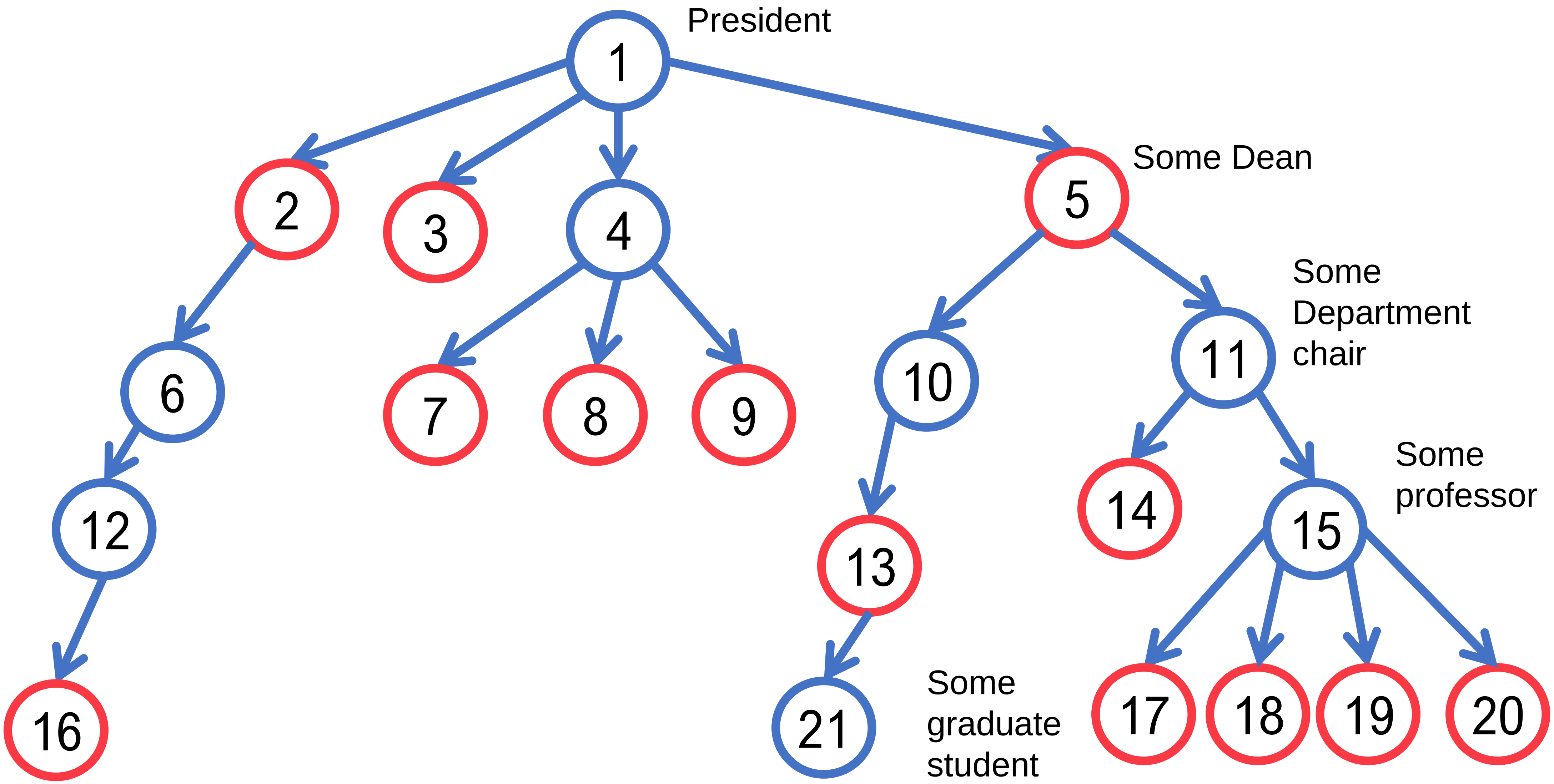
Let j and k be the two children of i, the best path across node i is:

Max(f[j] + f[k] + ω[i], ω[i], ω[i]+f[j], ω[i]+f[k])

Example: no-boss party

- In UCR, every employee has one direct boss
- All employees can be represented as a tree structure: every employee is represented as a tree node, and its parent is his/her direct boss
- Now we want to invite a subset of the employees to a party, but no one wants to join the party with his/her direct boss
- What is the maximum number of participants we can invite to the same party?

Example: no-boss party



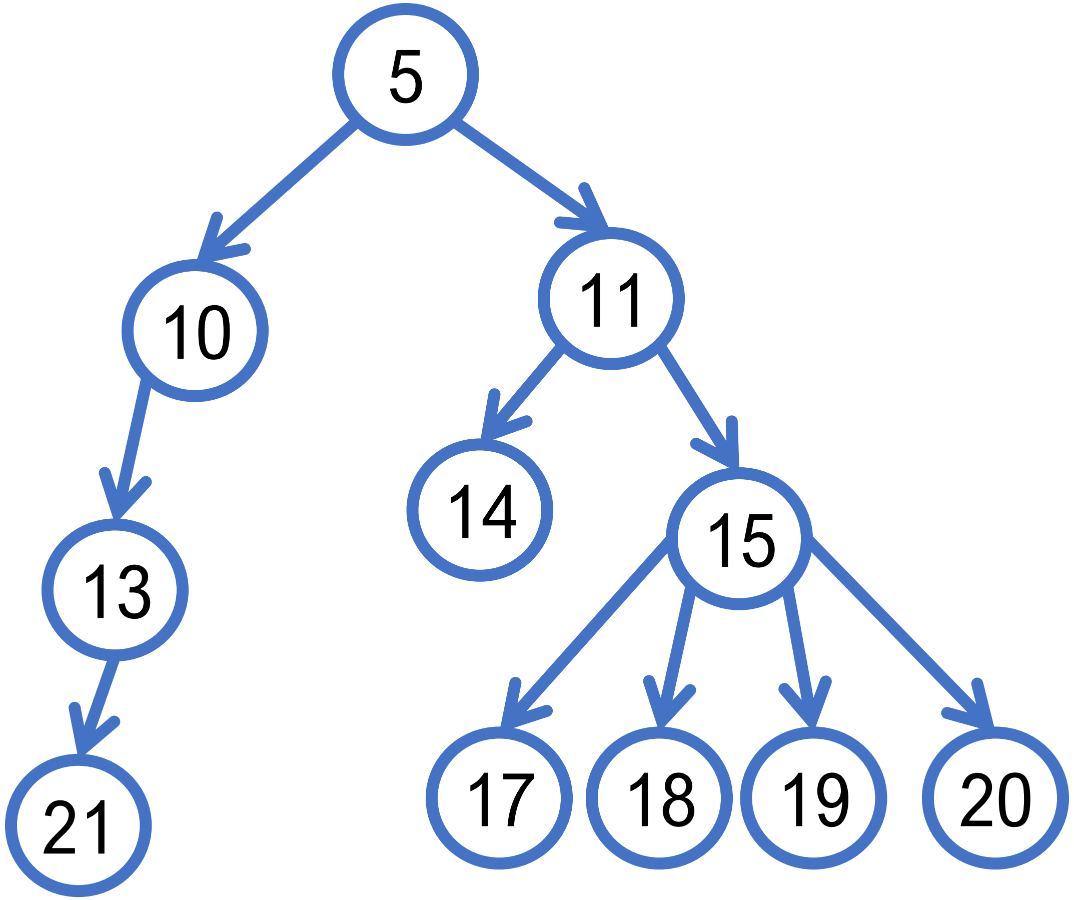
No-boss party

- **We can use $f[i]$ to denote the largest number of nodes we can choose from i's subtree**
  - f[i] should be computed using all f[j] for all its children j

- **But how can we make sure a node is never selected with its parent?**

- **Add! Another! Dimension!**

$f[5] = ?$
It should be computed from $f[10]$ and $f[11]$

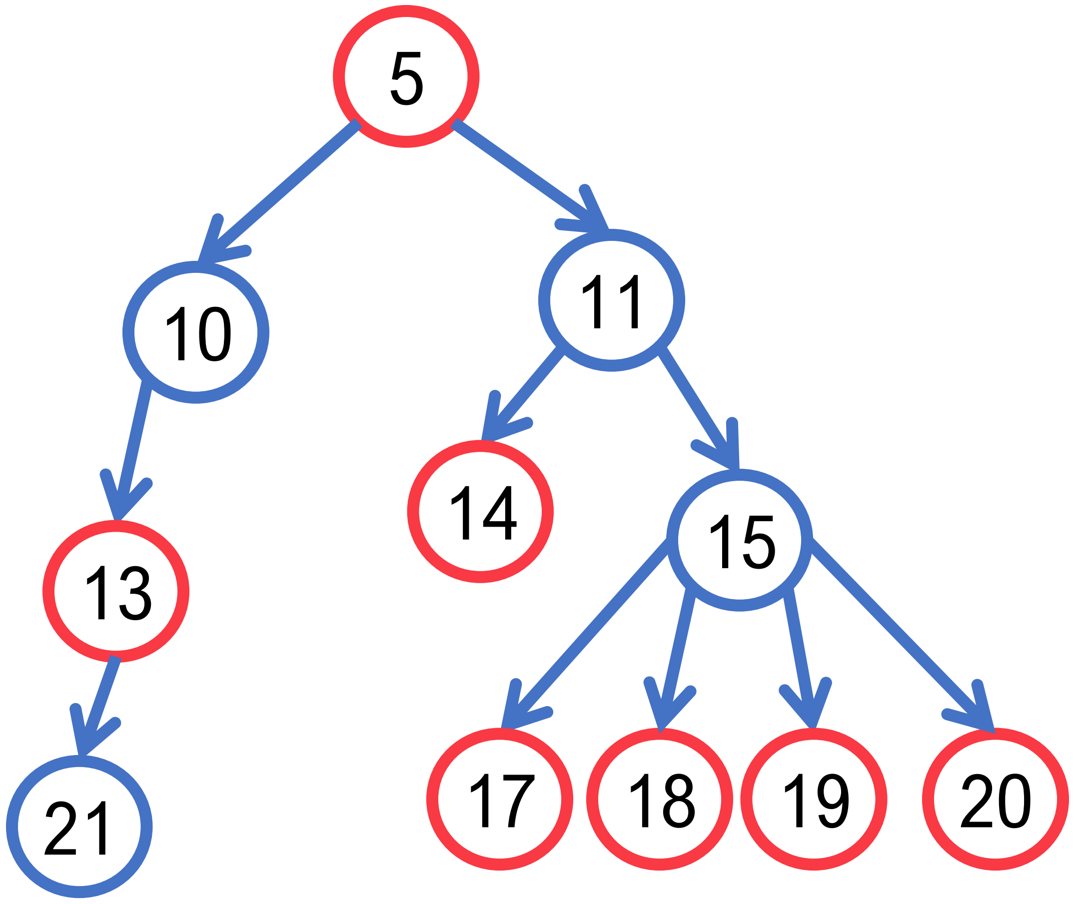


No-boss party

- $f[i,0]$ = the maximum number of people we can invite, if we don't invite i
- $f[i,1]$ = the maximum number of people we can invite, if we invite i

- $f[i,1] = 1 + \sum_{j \in child(i)} f[j,0]$
  - If we invite i, we cannot invite any of its children
  - For each subtree j, the best solution is of course the maximum number of participants in j's subtree without j

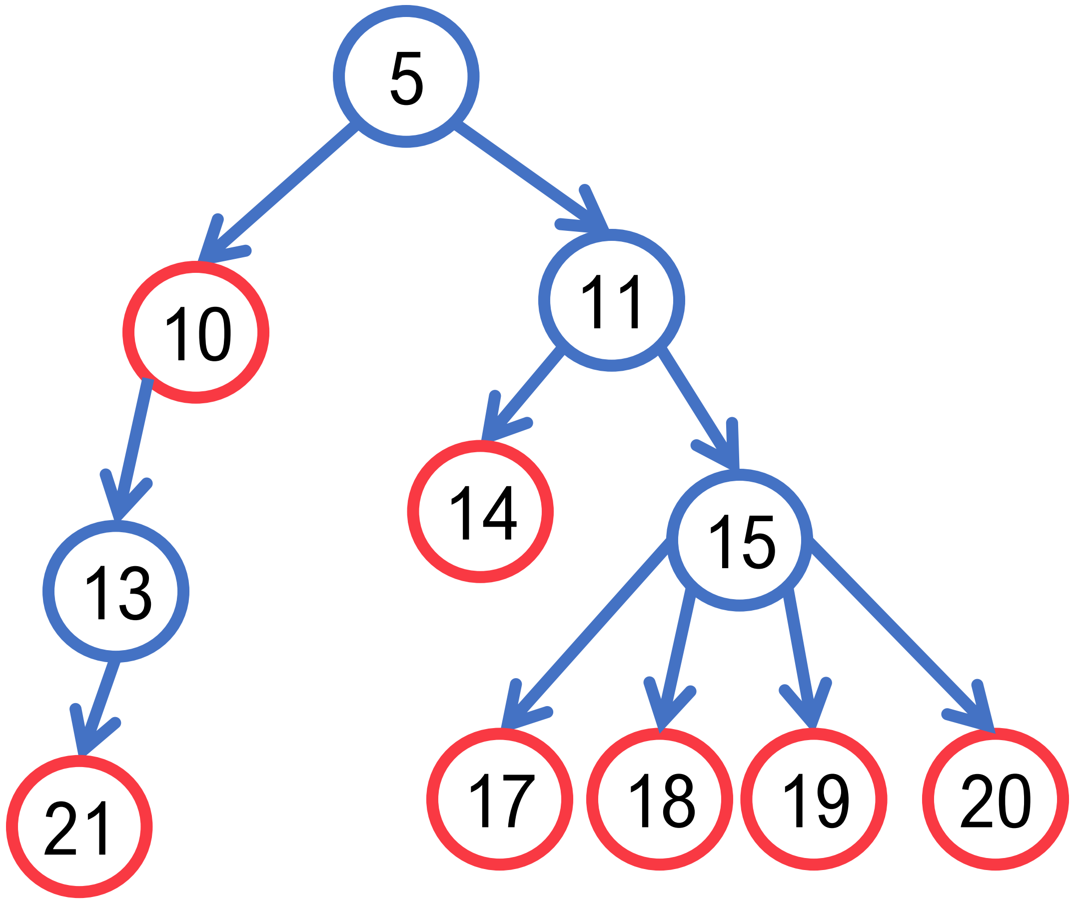
$f[5,1] = 1 + f[10,0] + f[11,0]$



No-boss party

$$f[5,0] = \max(f[10,0] + f[10,1]) + \max(f[11,0], f[11,1])$$

- $f[i,0]$ = the maximum number of people we can invite, if we don't invite i
- $f[i,1]$ = the maximum number of people we can invite, if we invite I
- $f[i,0] = \sum_{j \in child(i)} \max(f[j,0], f[j,1])$
  - If we don't invite i, we can either invite its children or not
  - For each subtree j, the best solution is of course the better solution between if we invite j or not



No-boss party: algorithm

- $\boldsymbol{f[i,1] = 1 + \sum_{j \in child(i)} f[j,0]}$
- $\boldsymbol{f[i,0]} = \sum_{j \in child(i)} \max(\boldsymbol{f[j,0], f[j,1]})$
- **Base case: $\boldsymbol{f[i,0] = 0}$ and $\boldsymbol{f[i,1] = 1}$**
- **An easy way: memorization**
  - Start from the root, traverse the tree until the leaves
- **A non-recursively way: decide the order based on the height**
  - First compute the f[ ] value for all leaves (height 1)
  - Then all nodes with height 2
  - Then height 3
  - …

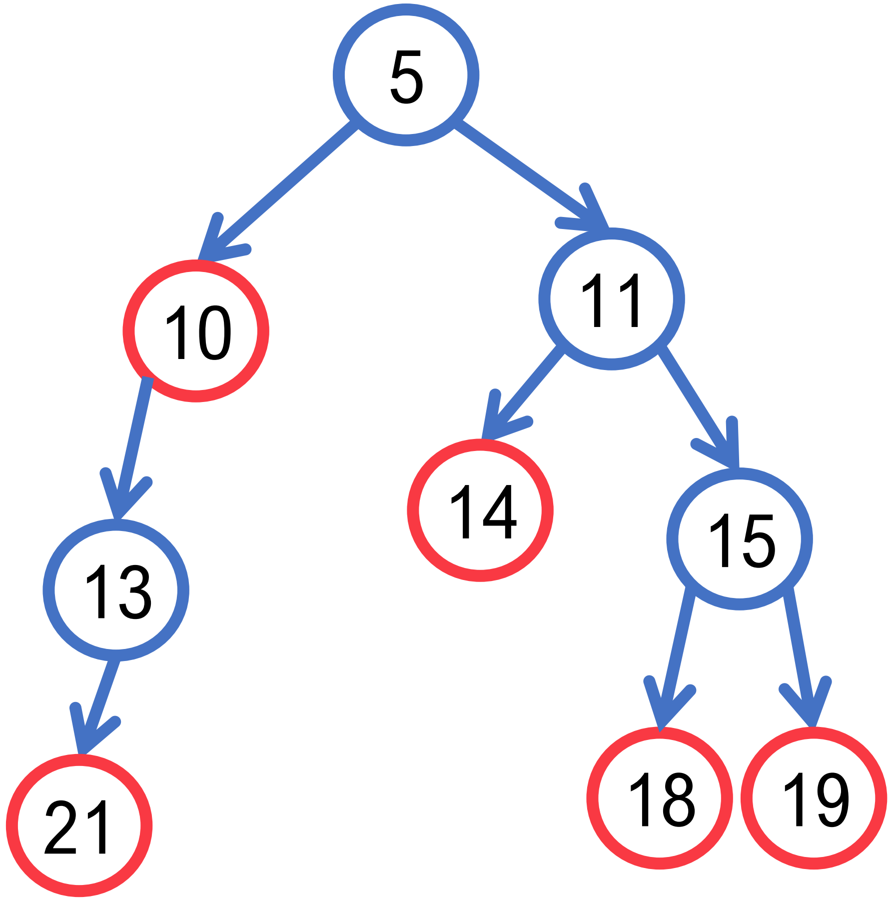
No-boss party: other variants

- If each node has a value $v[i]$, we want to maximize total value of selected people
- $f[i, 0]$ is max value of i's subtree with i, and $f[i, 1]$ is max value of i's subtree without i
- $f[i, 1] = \boldsymbol{v[i]} + \sum_{j \in child(i)} f[j, 0]$
- $f[i, 0] = \sum_{j \in child(i)} \max(f[j, 0], f[j, 1])$
- Base case: $f[i, 0] = 0$ and $\boldsymbol{f[i, 1] = v[i]}$

No-boss party: other variants

$$f[5,k,1] = v[5] + \max_{k_1+k_2=k-1} f[10,k_1,0] + f[11,k_2,0]$$

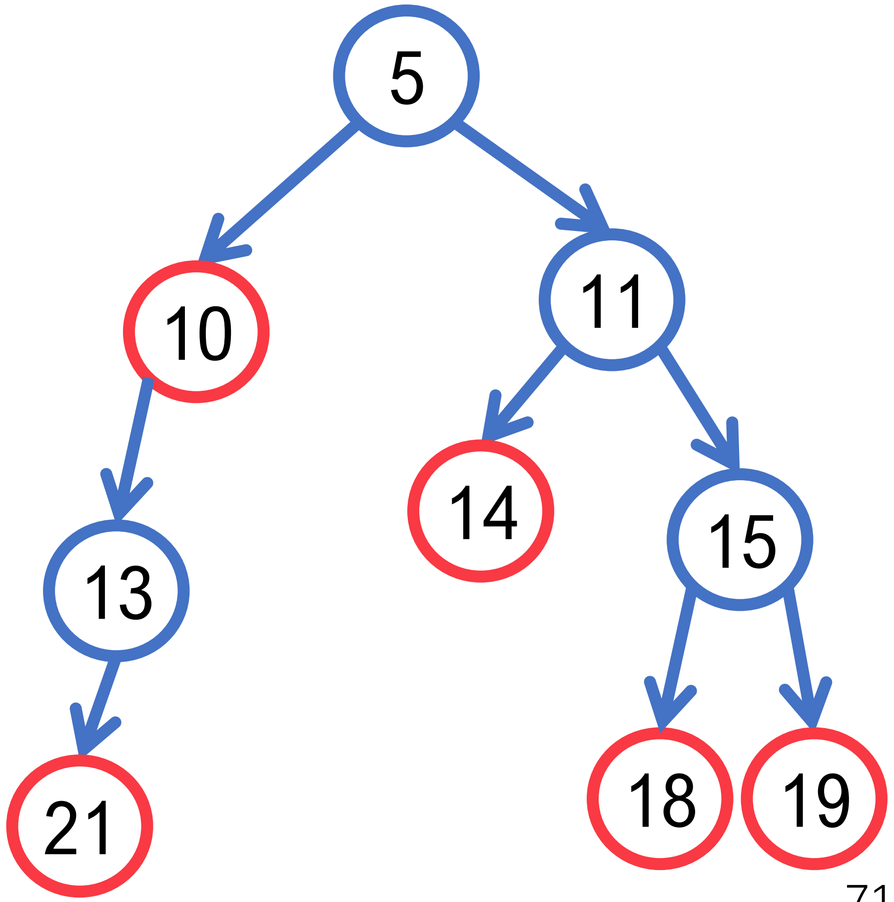
- If we can **only choose m people**
- If each node has a value $v[i]$, we want to maximize total value of selected people
- $f[i,k,1/0]$ is the max value of i's subtree if we select k people with/without selecting i
- $f[i,k,1] = v[i] +$ (select k-1 people from all its subtrees, but not choosing its children), i.e., transit from $f[j,*,0]$



No-boss party: other variants

$$f[5,k,0] = \max_{k_1+k_2=k}\left(\max(f[10,k_1,0], f[10,k_1,1]) + \max(f[11,k_2,0], f[11,k_2,1])\right)$$

- If we can **only choose m people**
- If each node has a value $v[i]$, we want to maximize total value of selected people
- $f[i,k,0]$ = select k people from all its subtrees
- How to compute "select k people of all its subtrees"?
  - This is a knapsack problem!
  - Try to figure out the details: see the homework problem (that's a must-have-a-boss party)



DP for trees

- **Usually we can start from the top (root) of the tree**
- **Usually the state of a node can depend on all its children**

- **Sometimes we can use another dimension for some additional state**
  - f[i, 0/1] for the i’s subtree with choosing/not choosing the current subtree root
  - f[i, k] for the i’s subtree with choosing k elements in this subtree