# Data Structures and Algorithms

(CS210A)

**Lecture 38**

- An interesting problem:

  shortest path from a source to destination

- Sorting Integers

# SHORTEST PATHS IN A GRAPH

## A fundamental problem

# Notations and Terminologies

A directed graph $G = (V, E)$

- $\omega: E \rightarrow R^+$
- Represented as **Adjacency** lists or **Adjacency** matrix
- $n = |V|$ , $m = |E|$

Question: what is a path in $G$?

Answer: A sequence $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for all $1 \le i < k$.


Length of a path $P = \sum_{e \in P} \omega(e)$

# Notations and Terminologies

**Definition**:

The path from $\boldsymbol{u}$ to $\boldsymbol{v}$ of minimum length is called the **shortest path** from $\boldsymbol{u}$ to $\boldsymbol{v}$

**Definition**: **Distance** from $\boldsymbol{u}$ to $\boldsymbol{v}$ is the **length** of the shortest path from $\boldsymbol{u}$ to $\boldsymbol{v}$.

**Notations**:

$\boldsymbol{\delta}(\boldsymbol{u}, \boldsymbol{v})$ : distance from $\boldsymbol{u}$ to $\boldsymbol{v}$.

$\boldsymbol{P}(\boldsymbol{u}, \boldsymbol{v})$ : The shortest path from $\boldsymbol{u}$ to $\boldsymbol{v}$.

# Problem Definition

**Input**: A directed graph $G = (V, E)$ with $\omega: E \rightarrow R^+$ and a source vertex $s \in V$

**Aim**:

- Compute $\delta(s, v)$ for all $v \in V \backslash \{s\}$
- Compute $P(s, v)$ for all $v \in V \backslash \{s\}$

What if $\omega(e) = 1$ ?

Do **BFS** from $s$
☺

# Problem Definition

**Input**: A directed graph $G = (V, E)$ with $\omega: E \rightarrow R^{+}$ and a source vertex $s \in V$

**Aim**:

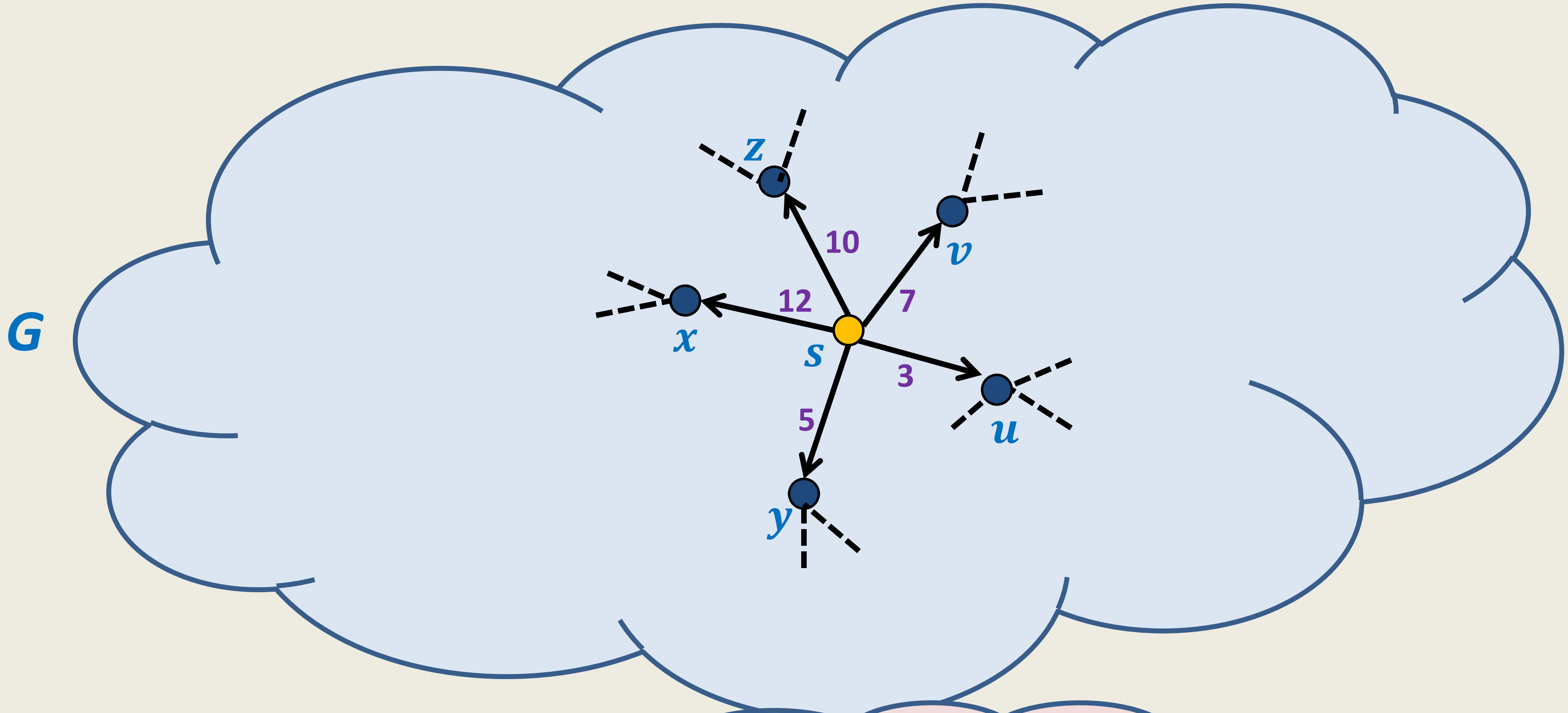
- Compute $\delta(s, v)$ for all $v \in V \backslash \{s\}$
- Compute $P(s, v)$ for all $v \in V \backslash \{s\}$

**First algorithm** : by **Edsger Dijkstra** in **1956**

And still the best …

I am sure you will be able to re-invent it yourself if you are asked right questions ☺ So get ready !

# An Example

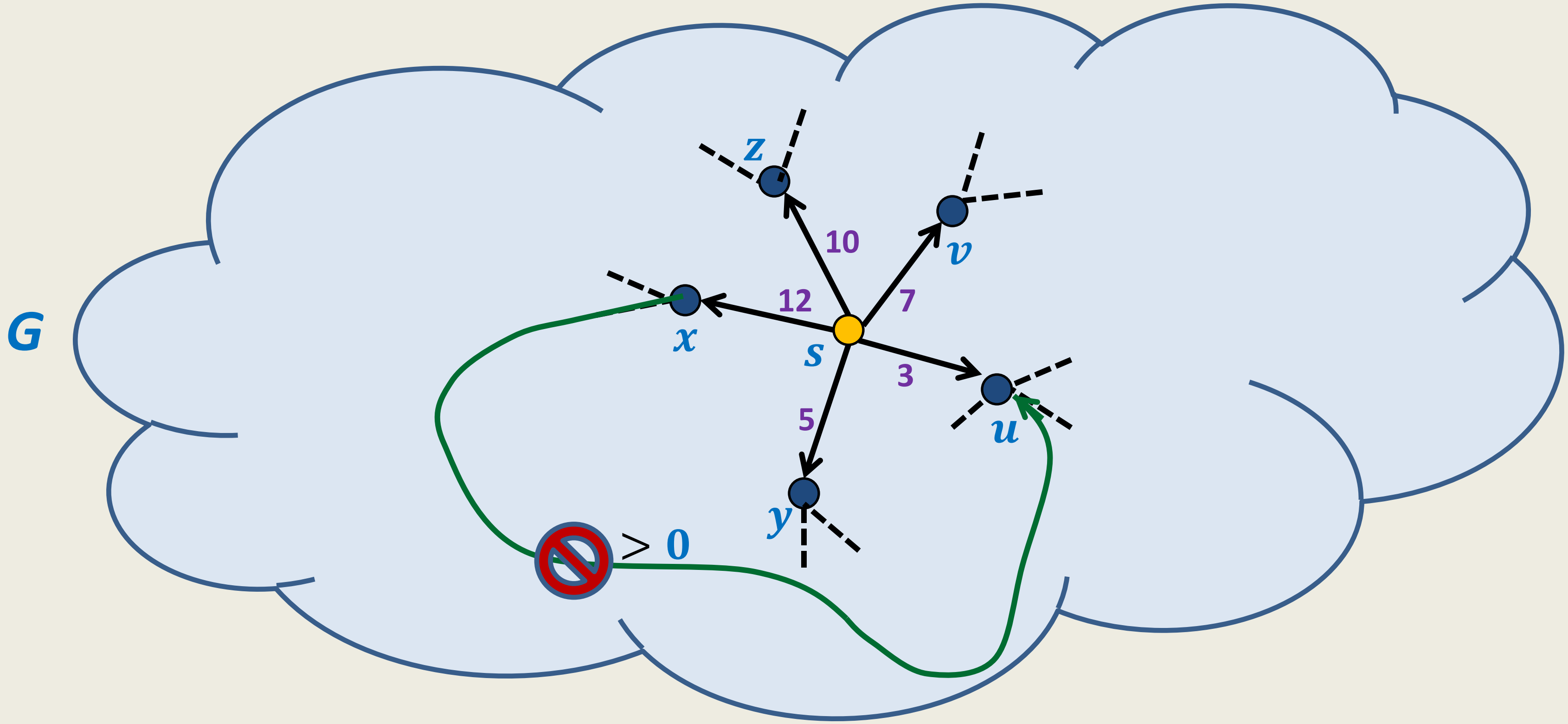



Can you spot any vertex for which you are certain about its distance from $s$ ?

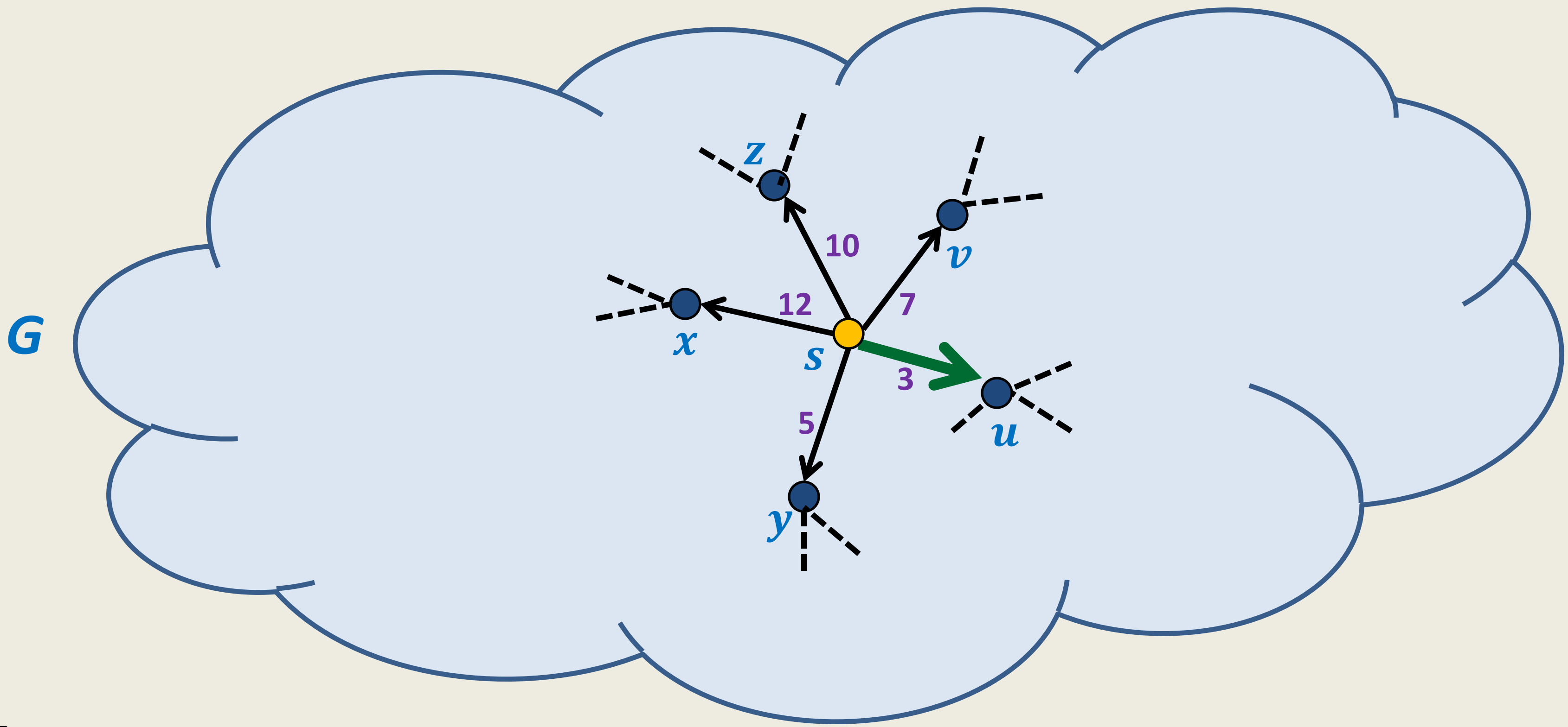
Give reasons.

**Answer**: vertex $u$.

# An Example



# An Example




➔ Yes, the edge $(s, u)$ is indeed the shortest path to $u$.

To form a **smaller instance** of the problem.
But how ?

next step ?

How to use it to design an algorithm for shortest paths ?

# **Pondering over the problem**

**Idea 1** :

Remove $\boldsymbol{u}$ since we have computed distance to $\boldsymbol{u}$. & so its job is done.

So now there will be $\boldsymbol{n-1}$ vertices.

The new graph **will preserve** those shortest paths from $\boldsymbol{s}$ in which $\boldsymbol{u}$ is not present.

But what about those shortest paths from $\boldsymbol{s}$ that pass through $\boldsymbol{u}$ ?

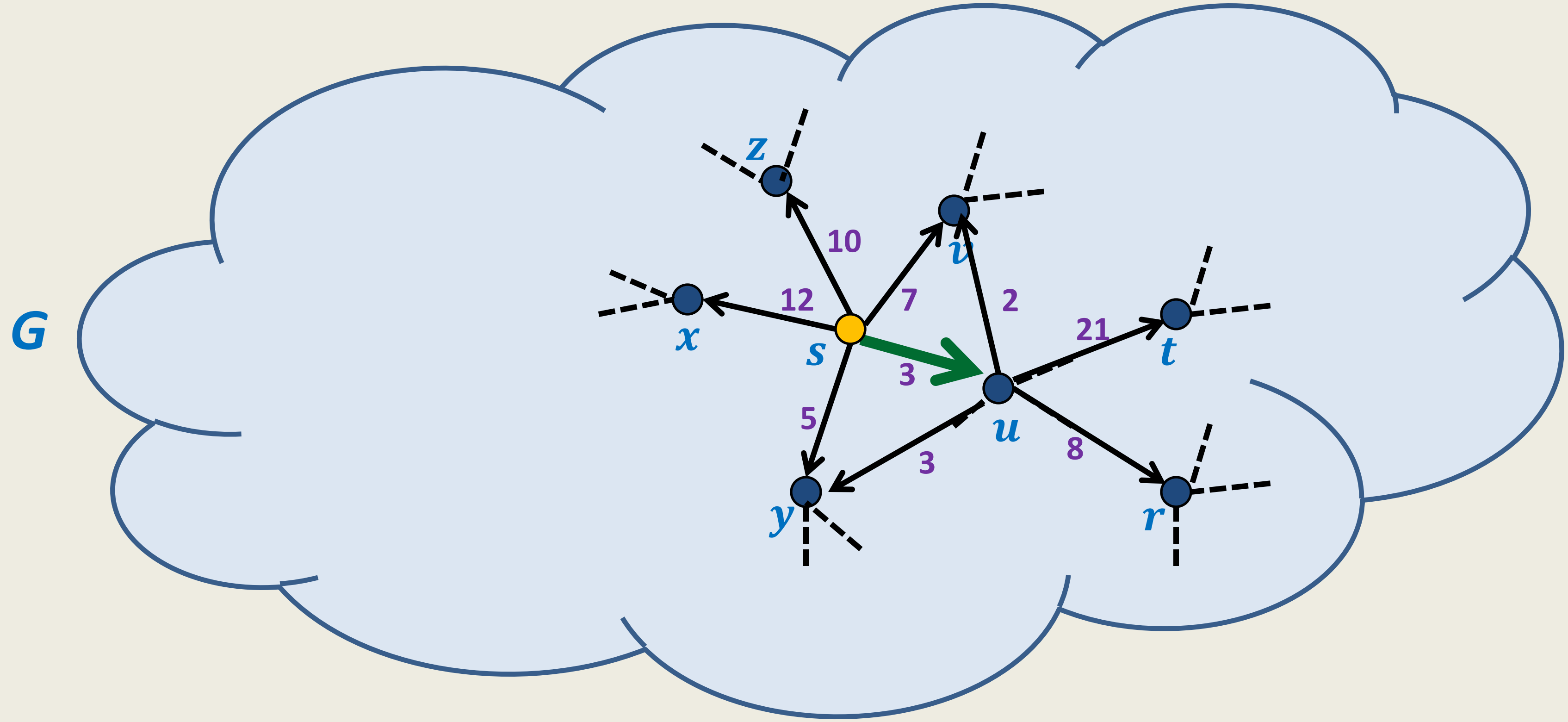
We lost them with the removal of $\boldsymbol{u}$. ☹

So we can't afford to remove $\boldsymbol{u}$.

How can then we get a smaller instance ?

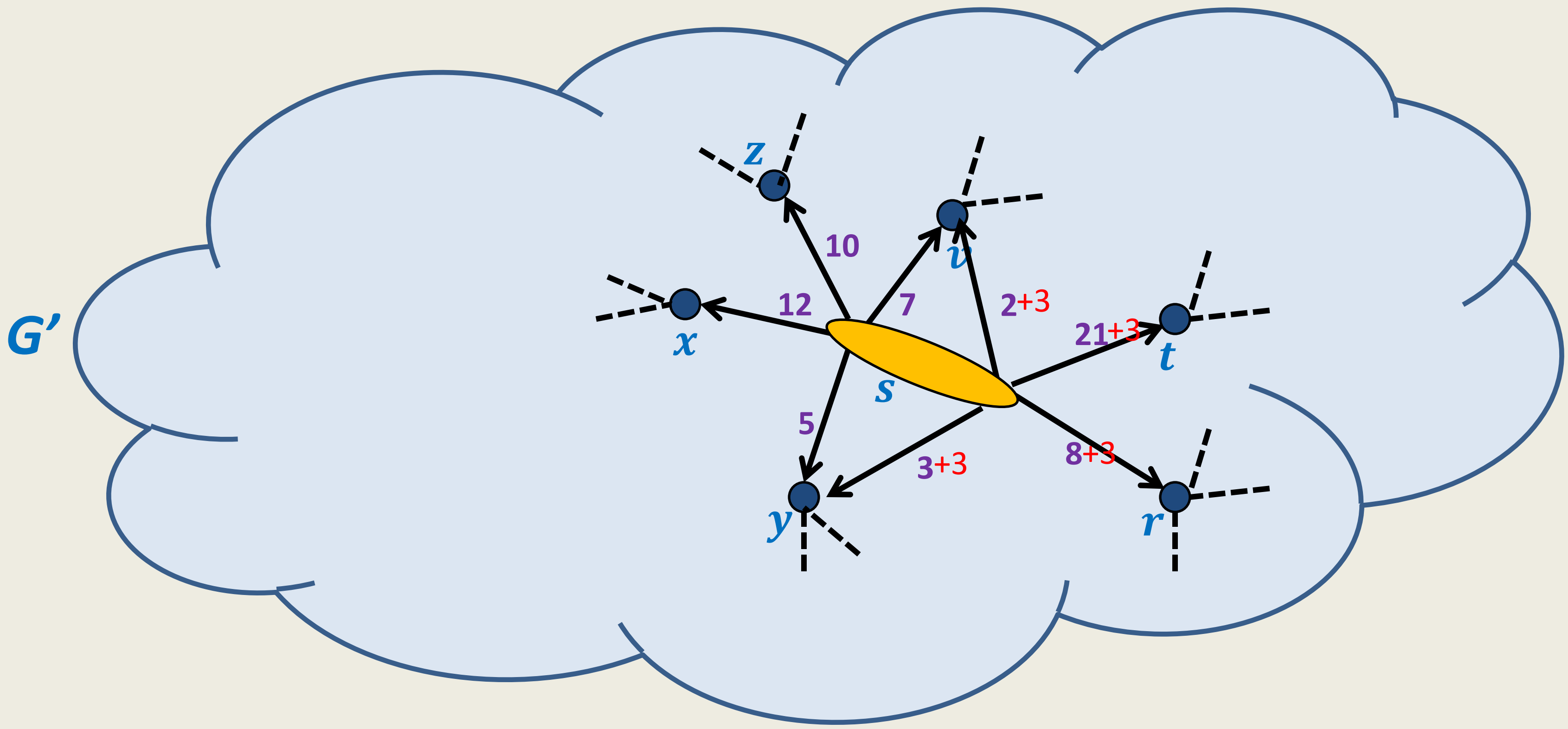
Merging $\boldsymbol{s}$ and $\boldsymbol{u}$ ☺
But how ?

# An Example

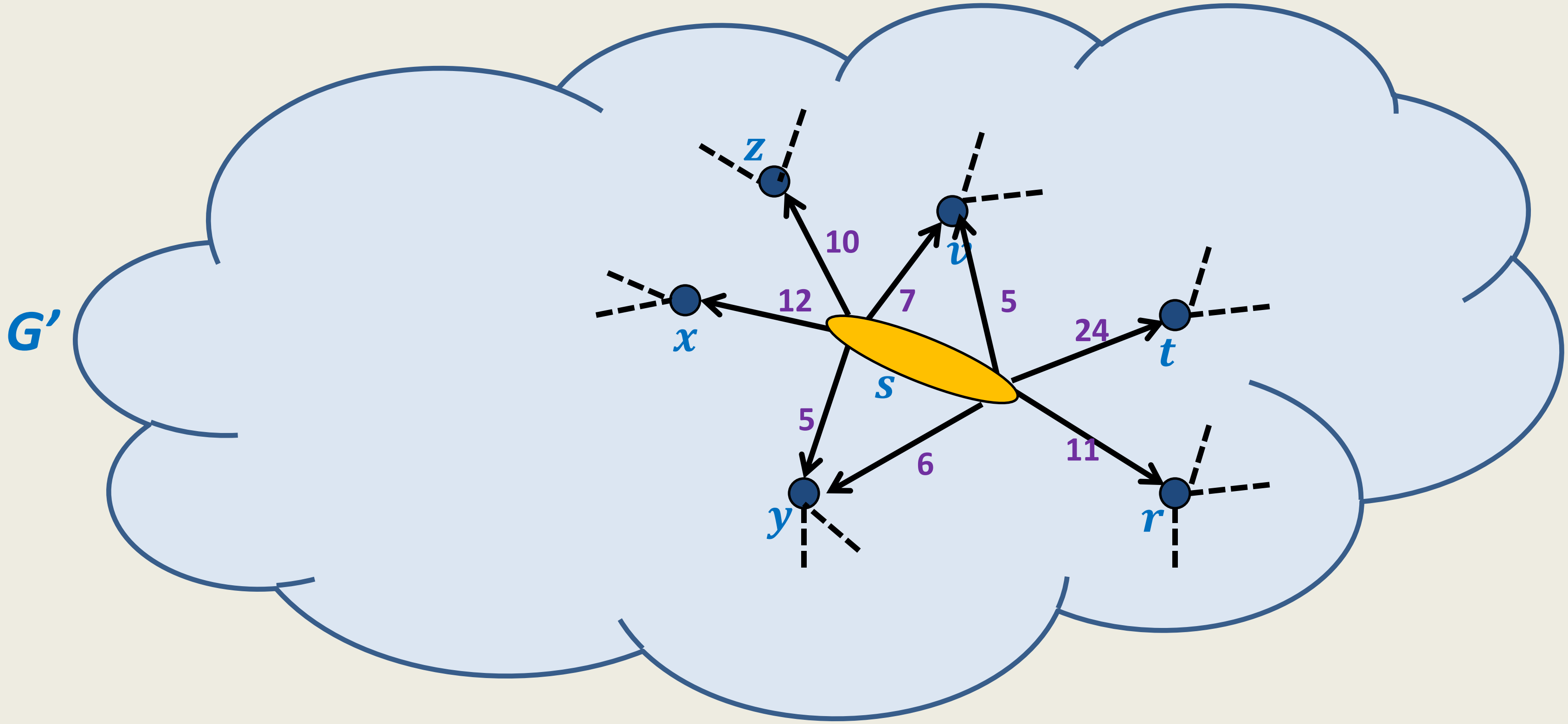



Let us look carefully around $u$ ?

# An Example



# An Example



# How to compute instance $G'$

Let ($s$,$u$) be the least weight edge from $s$ in $G$=($V$, $E$).

Transform $G$ into $G'$ as follows.

1. For each edge ($u$,$x$)∊ $E$,

   add edge ($s$,$x$);

   $\omega$($s$,$x$)← $\omega$($s$,$u$) + $\omega$($u$,$x$);

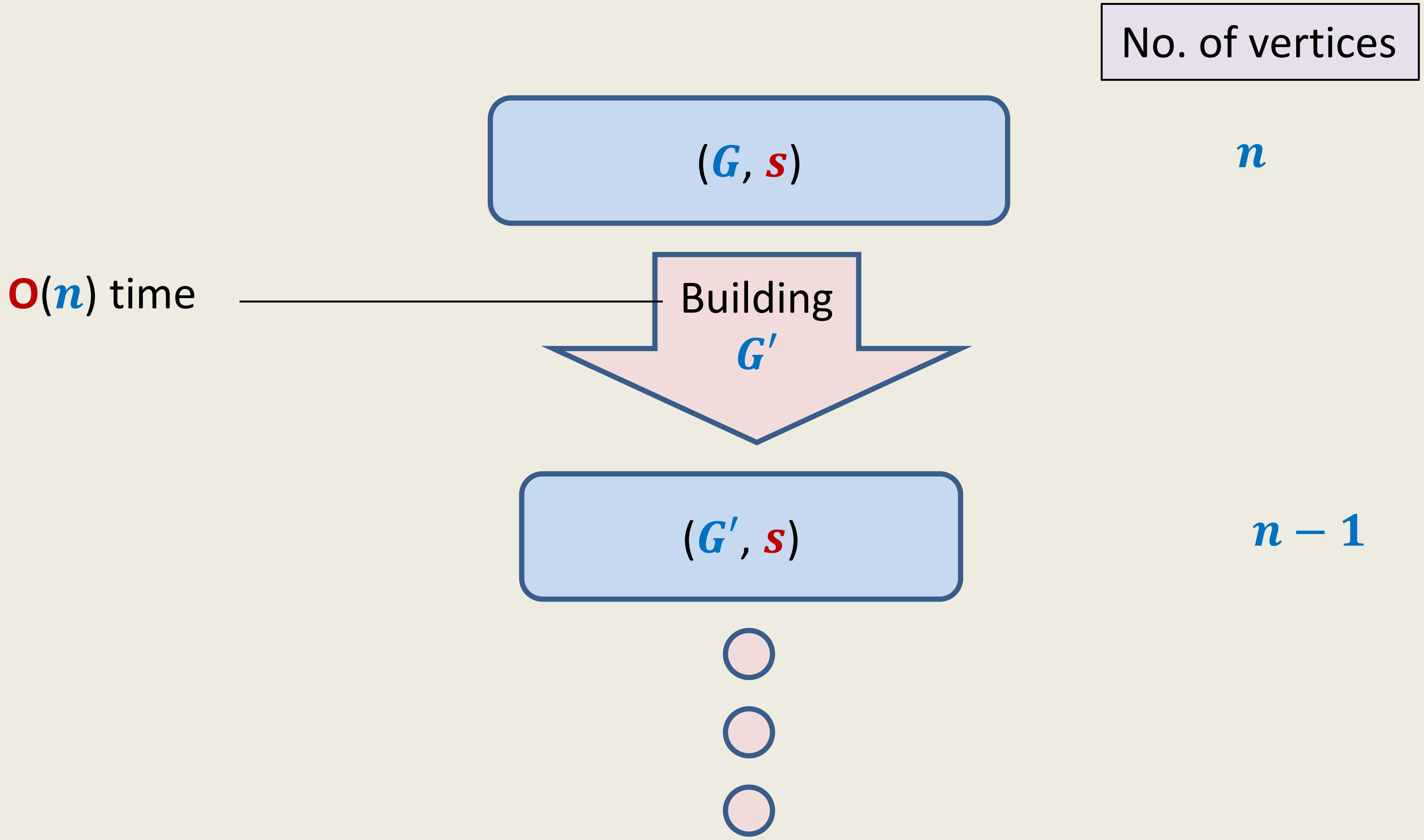
2. In case of two edges from $s$ to any vertex $x$

3. Remove vertex $u$.

**Theorem:** For each $v \in V \backslash \{s, u\}$,

$$\delta_G(s, v) = \delta_{G'}(s, v)$$

How efficient an algorithm for shortest paths can you design ?




➔ an algorithm for **distances** from $s$ with $O(n^2)$ time complexity.

# Integer sorting

# Algorithms for Sorting $n$ elements

- **Insertion** sort: O($n^2$)
- **Selection** sort: O($n^2$)
- **Bubble** sort: O($n^2$)
- **Merge** sort: O($n$ **log** $n$)
- **Quick** sort: worst case **O**($n^2$),
- **Heap** sort: O($n$ **log** $n$)

**Question:** What is common among these algorithms ?

**Answer:** All of them use only **comparison** operation to perform sorting.

## Theorem (to be proved in CS345):

Every comparison based sorting algorithm

must perform at least $n$ **log** $n$ comparisons in the worst case.

# **Question:** Can we sort in **O**($n$) time ?

**The answer** depends upon

- the **model of computation**.
- the **domain** of input.

# Integer sorting

# **Counting sort:** algorithm for sorting integers

**Input:** An array **A** storing $n$ integers in the range [$0 \ldots k-1$].

$k$ = **O**($n$)

**Output:** Sorted array **A**.

**Running time: O**($n+k$) in **word RAM** model of computation.

**Extra space: O**($k$)

**Motivating example: Indian railways**

There are **13 lacs** employees.

**Aim :** To **sort** them list according to **DOB** (date of birth)

**Observation:** There are only **14600** different date of births possible.

# Counting sort: algorithm for sorting integers

**A**

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

**Count**

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 0 | 2 | 3 | 0 | 1 |

**Place**

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 2 | 2 | 4 | 7 | 7 | 8 |

**B**

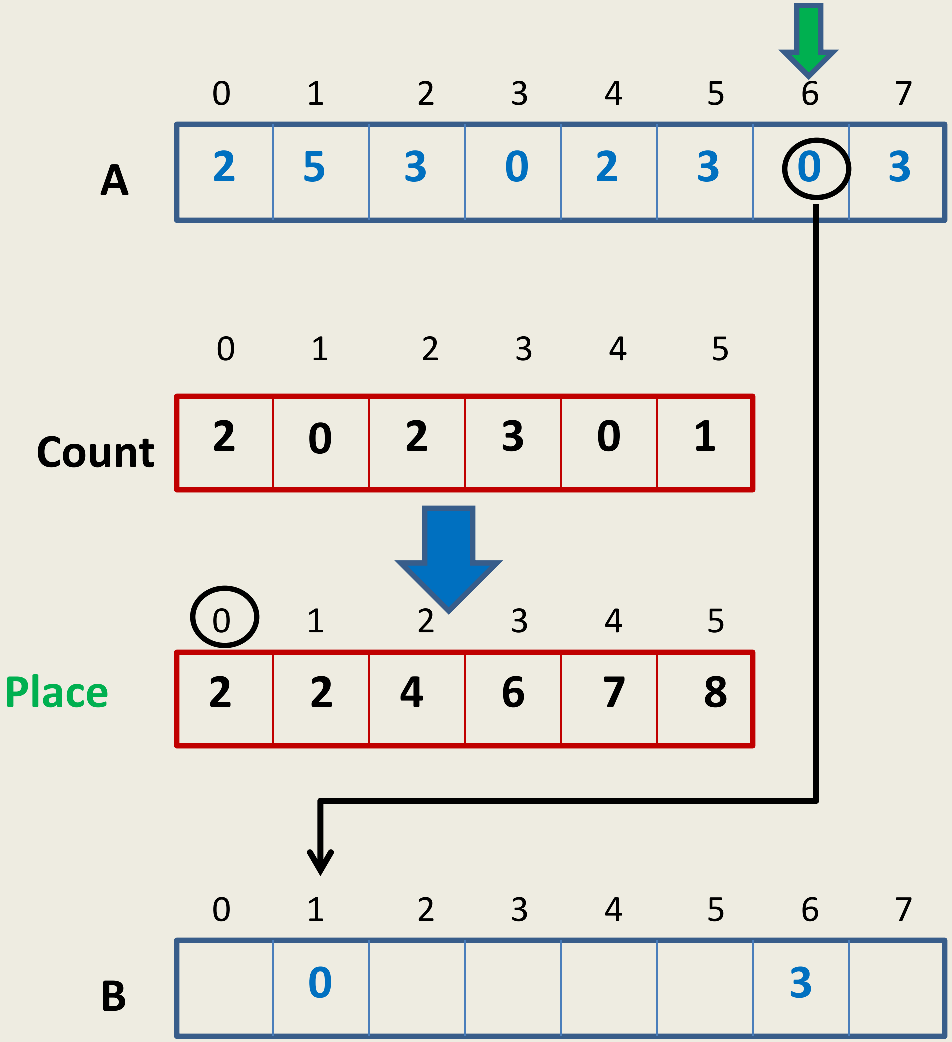
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| | | | | | | 3 | |

Final sorted output

If **A**[*i*]=*j*,
where should A[*i*] be
placed in **B** ?

Certainly after all those elements in **A** which are **smaller** than *j*

# Counting sort: algorithm for sorting integers



# **Counting sort:** algorithm for sorting integers

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **A** | 2 | 5 | 3 | 0 | 2 | 3 | 0 | 3 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Count** | 2 | 0 | 2 | 3 | 0 | 1 |

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Place** | 1 | 2 | 4 | 6 | 7 | 8 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| **B** | | 0 | | | | 3 | 3 | |

# **Counting sort:** algorithm for sorting integers

**Algorithm** (A[$0 \ldots n-1$], $k$)

For $j$=0 to $k-1$ do Count[$j$]← 0;

For $i$=0 to $n-1$ do Count[ A[$i$] ]← Count[ A[$i$] ] +1;

For $j$=0 to $k-1$ do Place[$j$]← Count[$j$];

For $j$=1 to $k-1$ do Place[$j$]← Place[$j-1$] + Count[$j$] ;

For $i$=$n-1$ to $0$ do

{ B[ Place[A[$i$]]-1 ]← A[$i$];

Place[A[$i$]] ← Place[A[$i$]]-1;

}

return B;

What is the time complexity of this algorithm in **word RAM** model ?