Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 38

- **An interesting problem:**

  shortest path from a source to destination
- **Sorting Integers**

SHORTEST PATHS IN A GRAPH

A fundamental problem

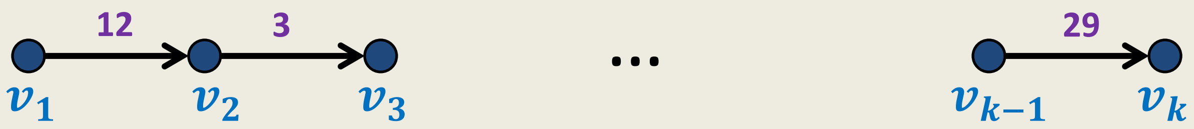
Notations and Terminologies

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 $1 \leq i < k$.



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 $\omega : E \to R^{+}$ 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 $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with $\boldsymbol{\omega}: \boldsymbol{E} \to \boldsymbol{R}^{+}$ and a source vertex $\boldsymbol{s} \in \boldsymbol{V}$

Aim:

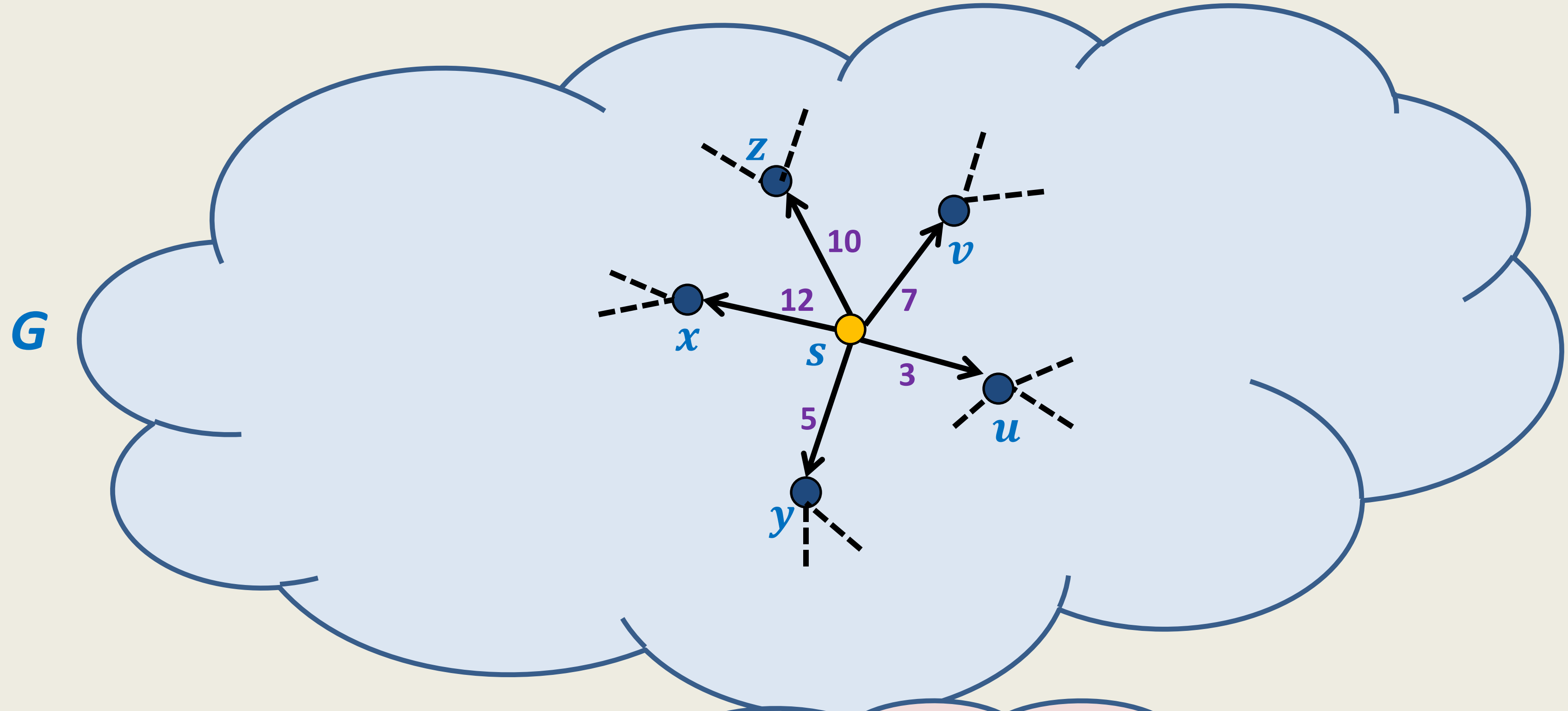
- Compute $\boldsymbol{\delta}(\boldsymbol{s}, \boldsymbol{v})$ for all $\boldsymbol{v} \in \boldsymbol{V} \backslash \{\boldsymbol{s}\}$
- Compute $\boldsymbol{P}(\boldsymbol{s}, \boldsymbol{v})$ for all $\boldsymbol{v} \in \boldsymbol{V} \backslash \{\boldsymbol{s}\}$

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

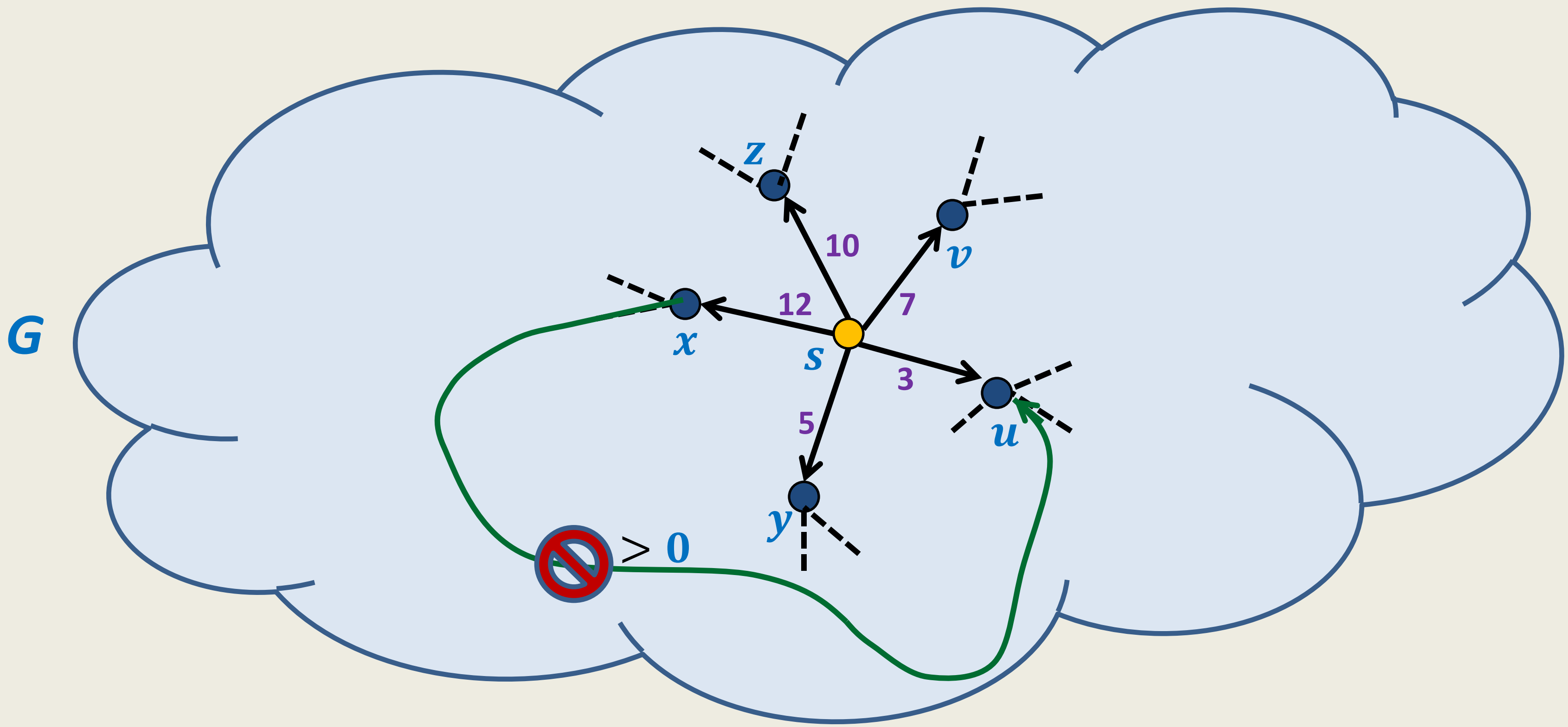


Can you spot any vertex for which you are certain about its distance from s ?

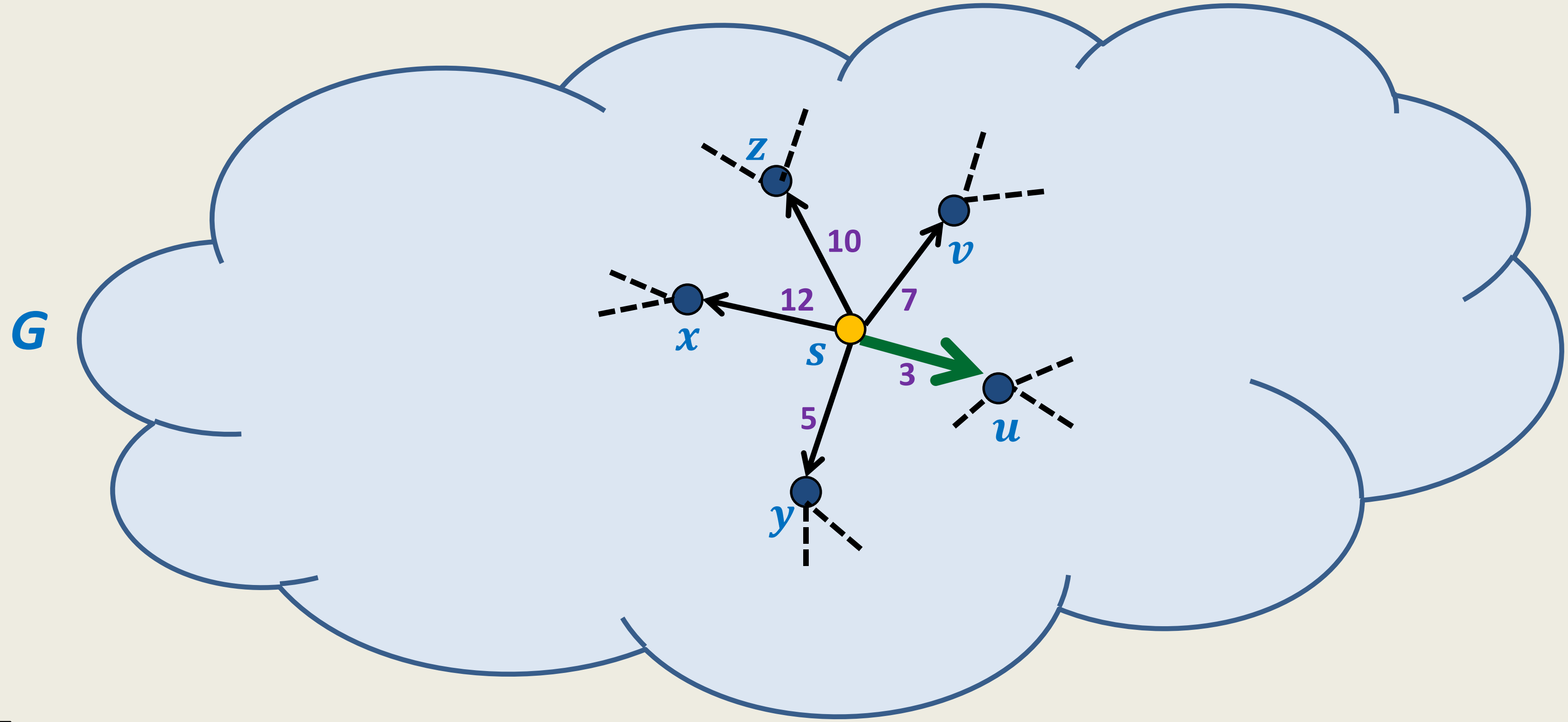
Answer: vertex u.

Give reasons.

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 u since we have computed distance to u. & so its job is done.

So now there will be $n - 1$ vertices.

The new graph **will preserve** those shortest paths from s in which u is not present.

But what about those shortest paths from s that pass through u ?

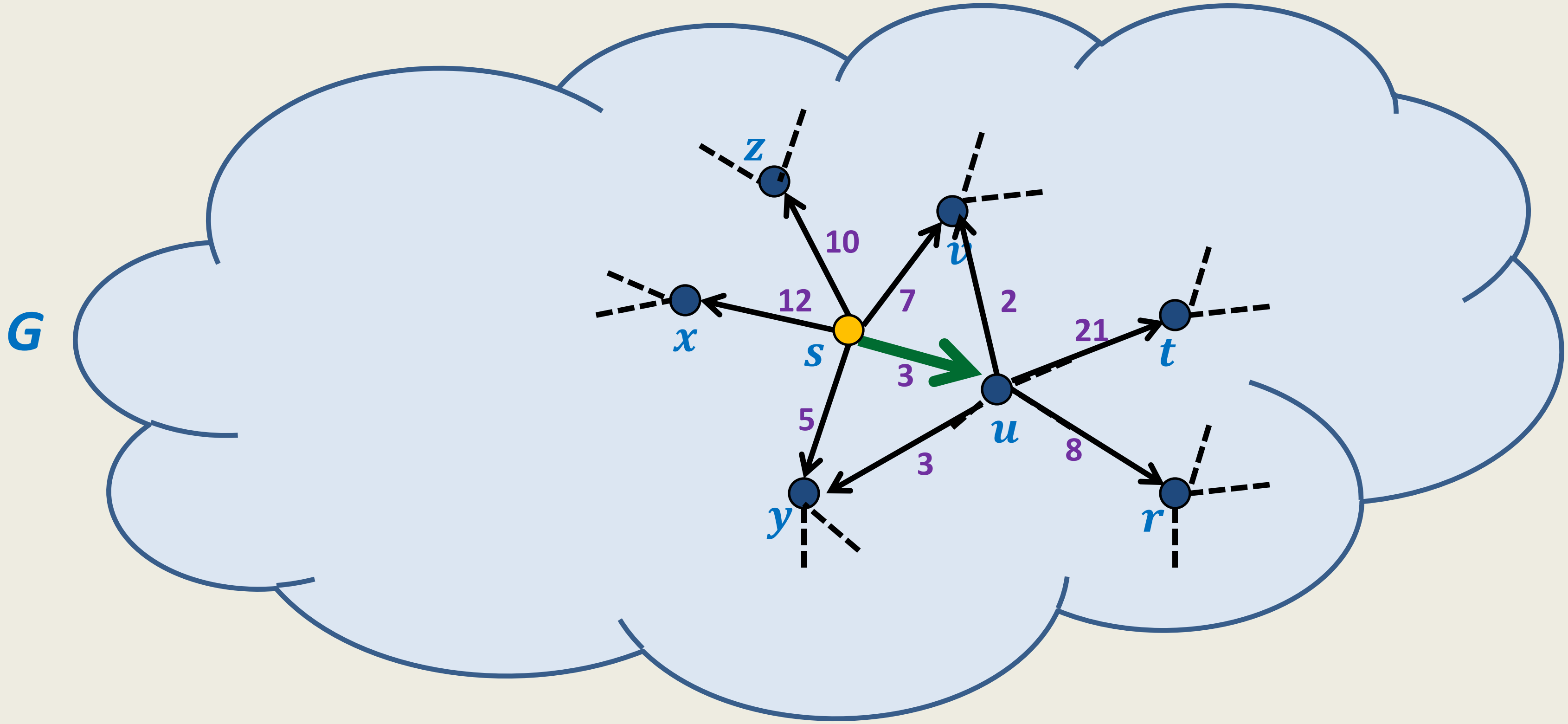
We lost them with the removal of u. ☹

So we can't afford to remove u.

How can then we get a smaller instance ?

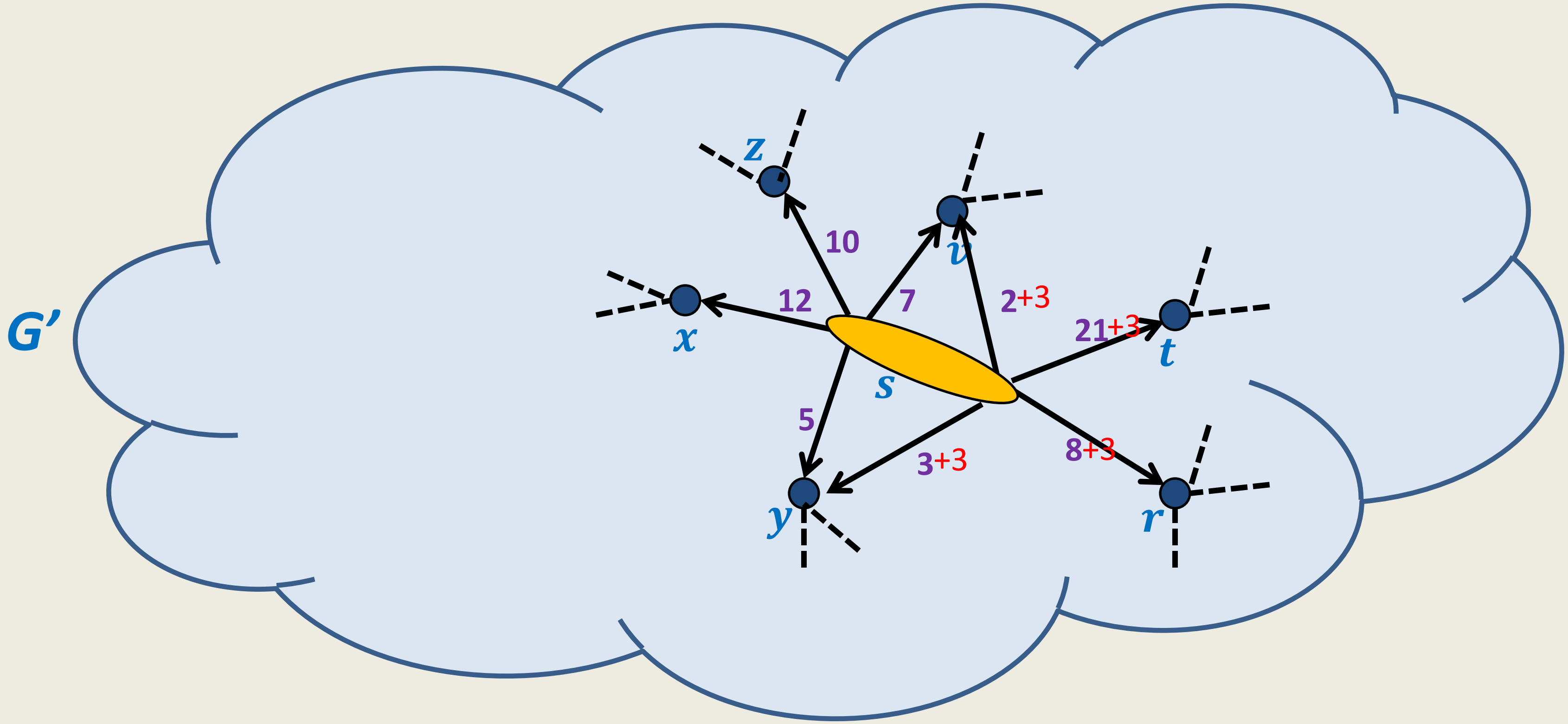
Merging s and u ☺
But how ?

An Example

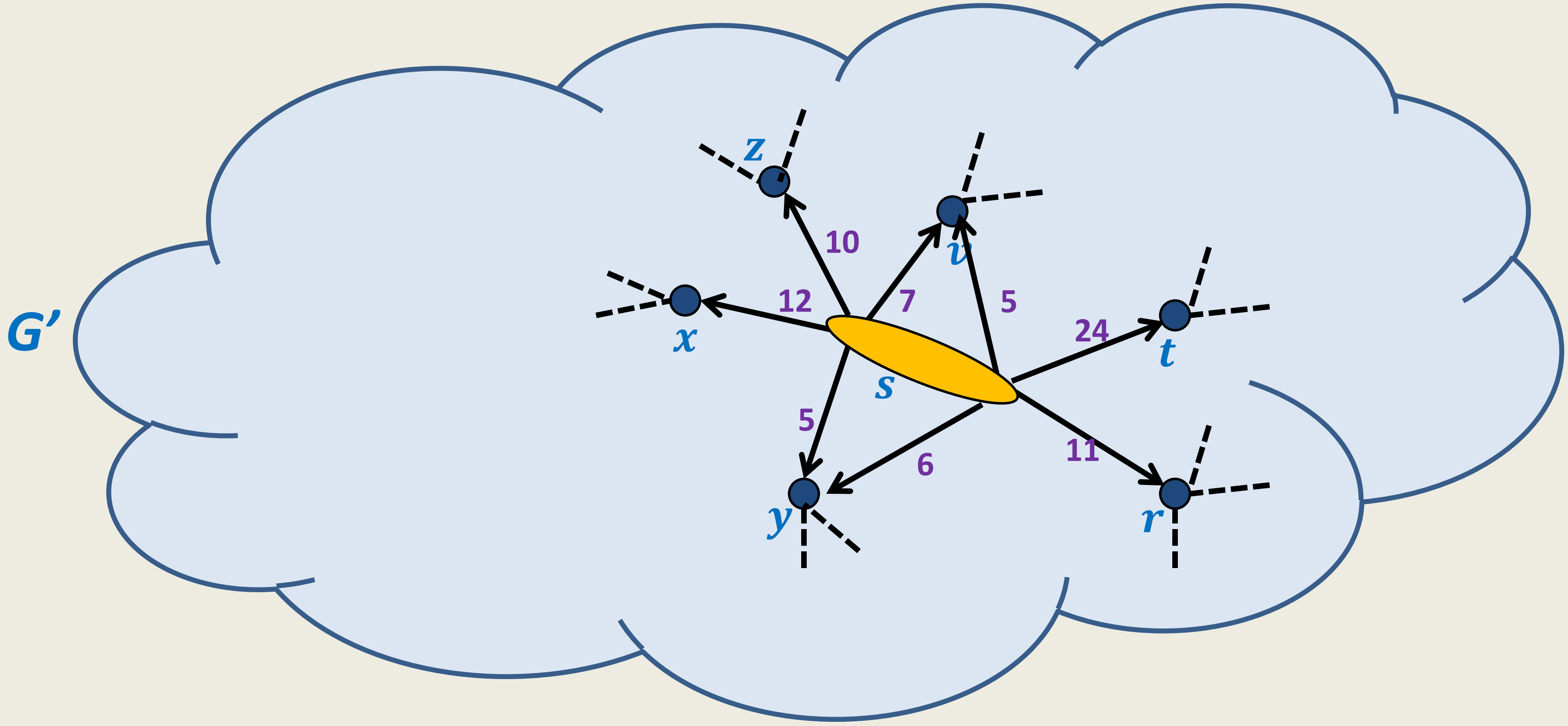


Let us look carefully around u ?

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 $(u,x) \in E$,

   add edge (s,x);

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

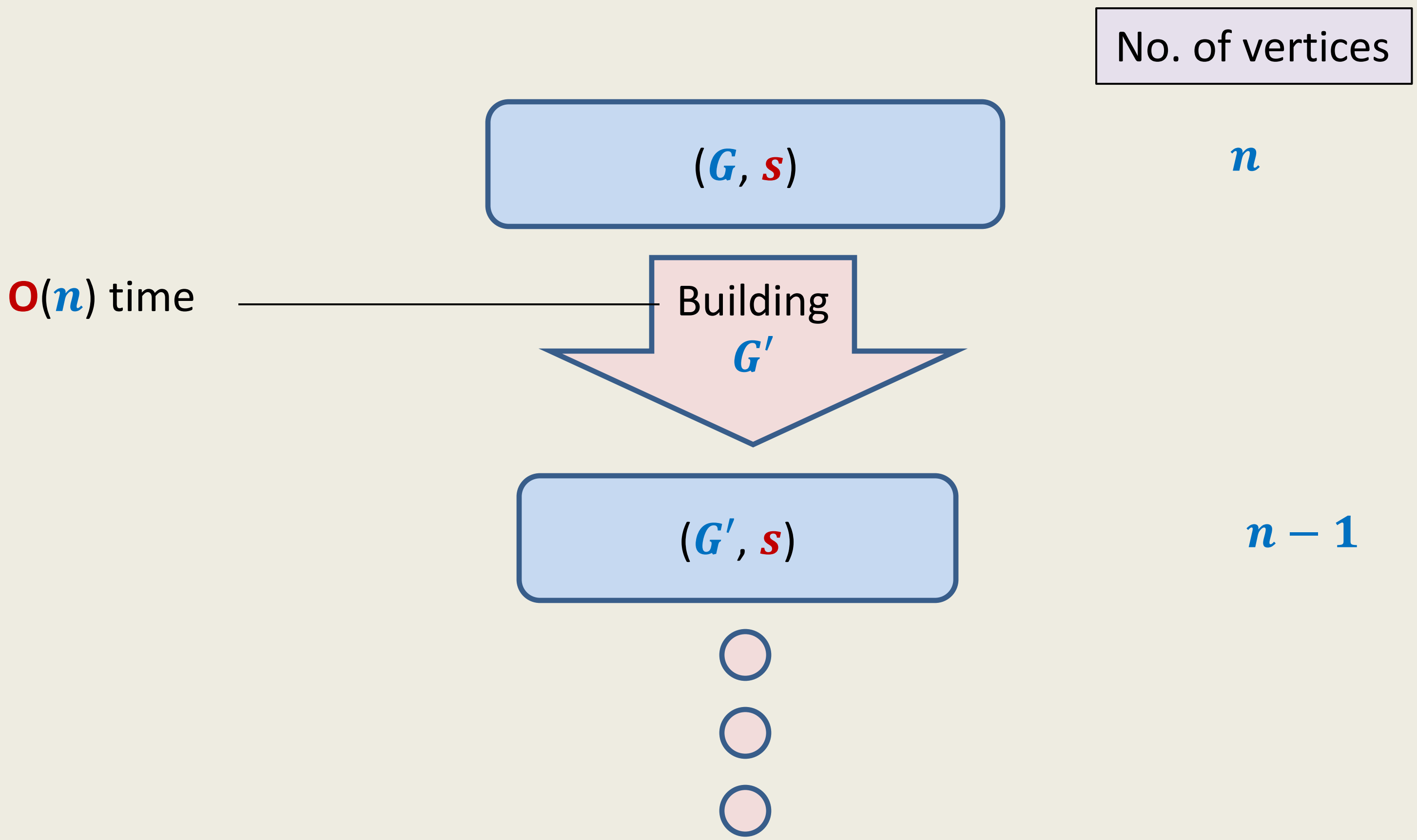
2. In case of two edges from s to any vertex x, keep only the **lighter** edge.

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), average case **O**$(n \log n)$
- **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 **O**$(n \log n)$ 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.

word RAM model of computation: Characteristics

- Word is the **basic storage** unit of RAM.
- Each input item (number, name) is stored in **binary format**.
- RAM can be viewed as a huge array of words. Any arbitrary location of RAM can be **accessed** in the same time **irrespective** of the location.
- Data as well as Program **reside fully** in RAM.
- Each arithmetic or logical operation (+,-,*,/,or, xor,…) involving **O(log n) bits** take **a constant number of steps** by the CPU, where **n** is the number of bits of input instance.

Integer sorting

Counting sort: algorithm for sorting integers

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

Output: Sorted array **A**.

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

Extra space: **O**(k)

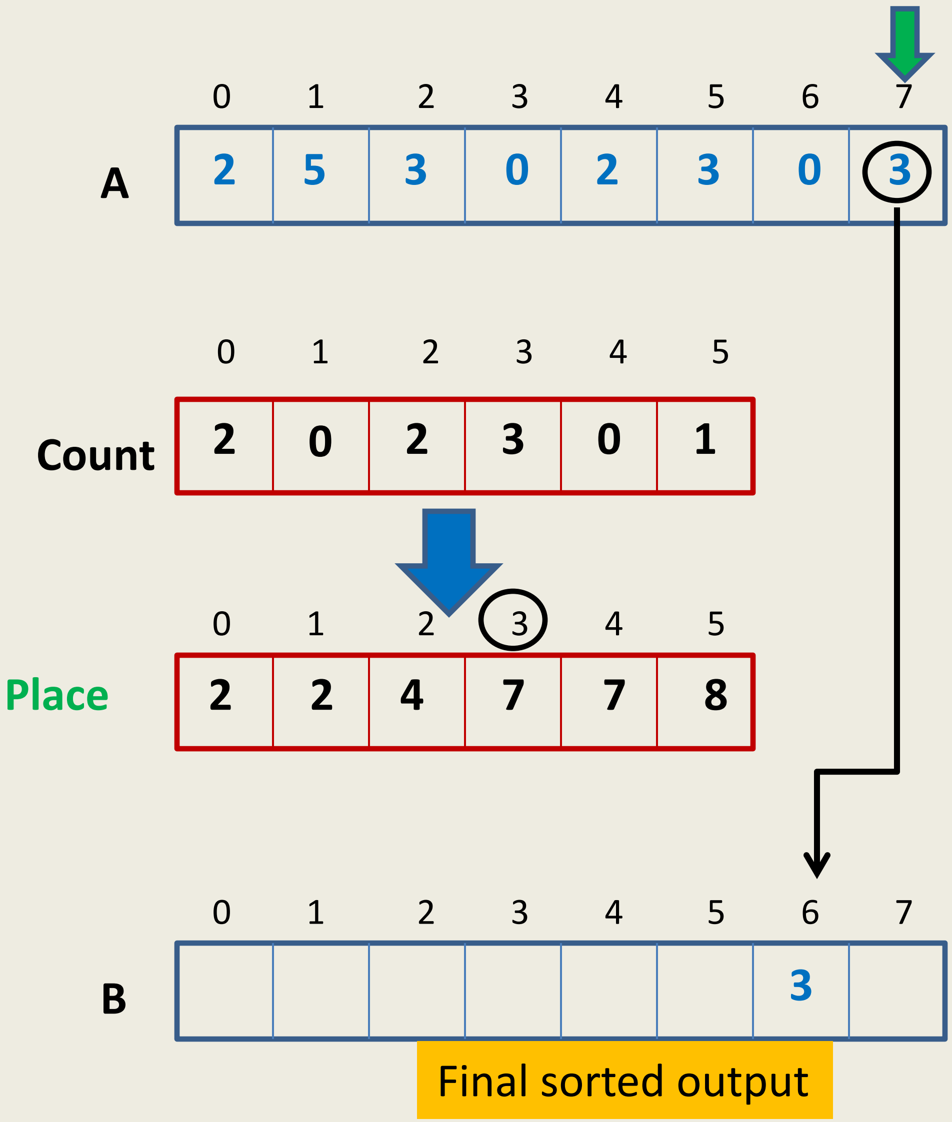
Motivating example: Indian railways

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.

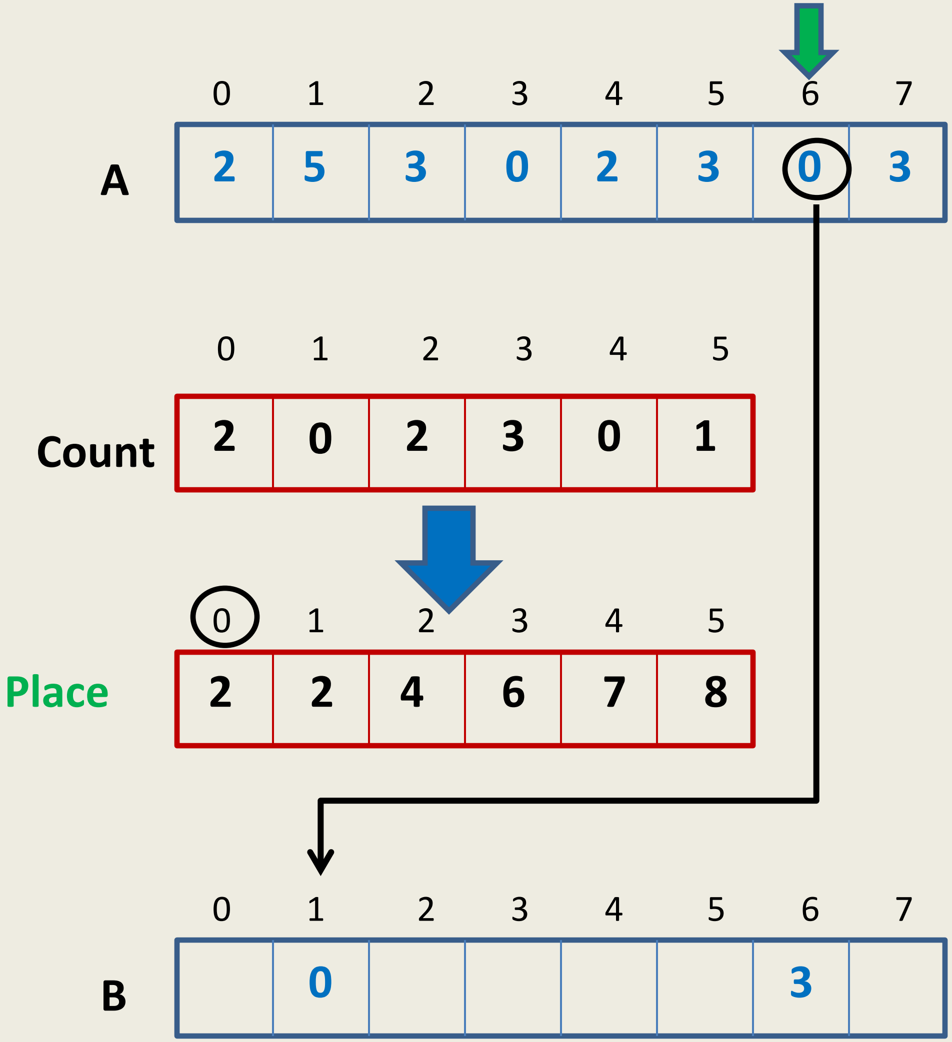
Counting sort: algorithm for sorting integers



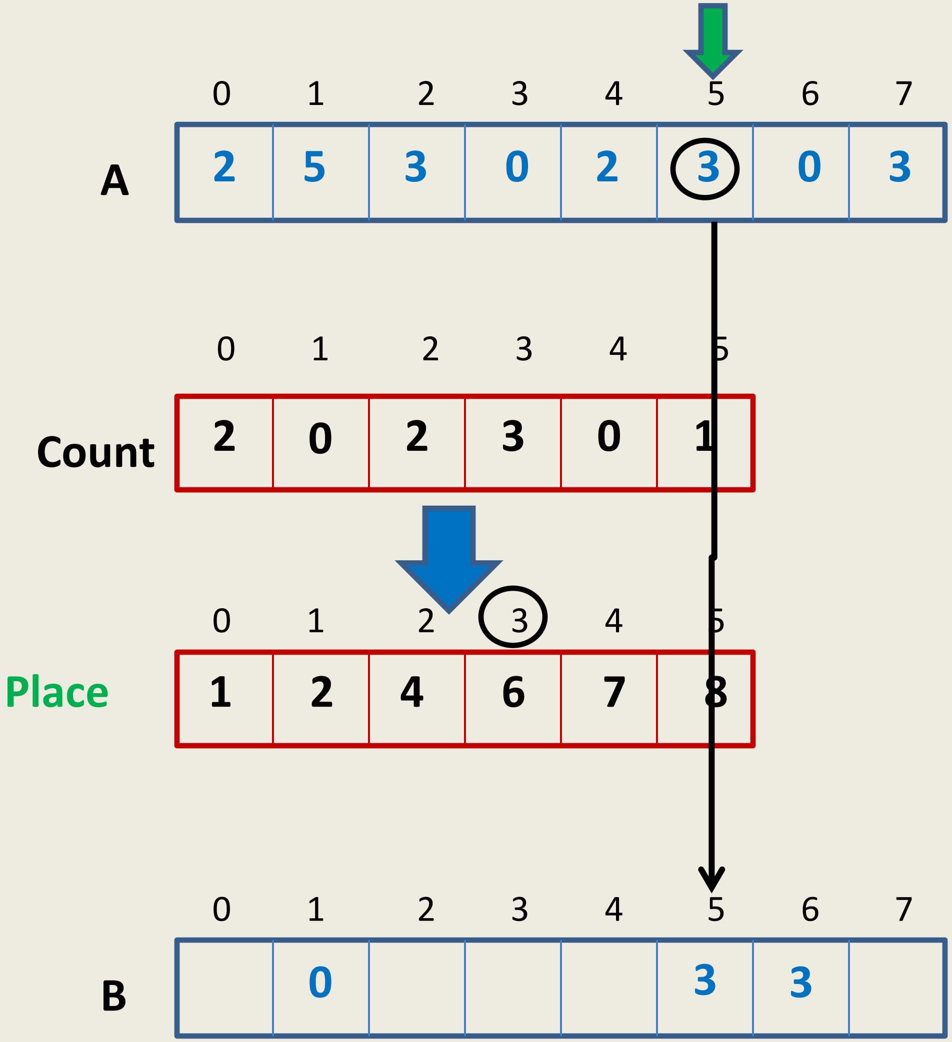
If **A**[*i*]=*j*,
where should A[*i*] be
placed in **B** ?

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

Counting sort: algorithm for sorting integers



Counting sort: algorithm for sorting integers



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;

Each arithmetic operations

involves O(log n + log k) bits

Counting sort: algorithm for sorting integers

Note: The algorithm performs arithmetic operations involving **O**(**log** n + **log** k) bits.
In **word RAM** model, it takes **O**(1) time for such an operation.

Theorem: An array storing n integers in the range $[\mathbf{0}..k-\mathbf{1}]$can be sorted in **O**($n$+$k$) time and using total **O**(n+k) space in **word RAM** model.

➔ **For** k **=** **O**(n), we get an optimal algorithm for sorting. But what if k is large ?
➔ **For** $k = n^t$, time and space complexity is **O**(n^t).

(too bad for $t > 1$. ☹)

Question:
How to sort n integers in the range $[\mathbf{0}..n^t]$ in **O**(tn) time and using **O**(n) space?

Next class