Data Structures and Algorithms

(ESO207)

Lecture 31

- Magical applications of Binary trees -II

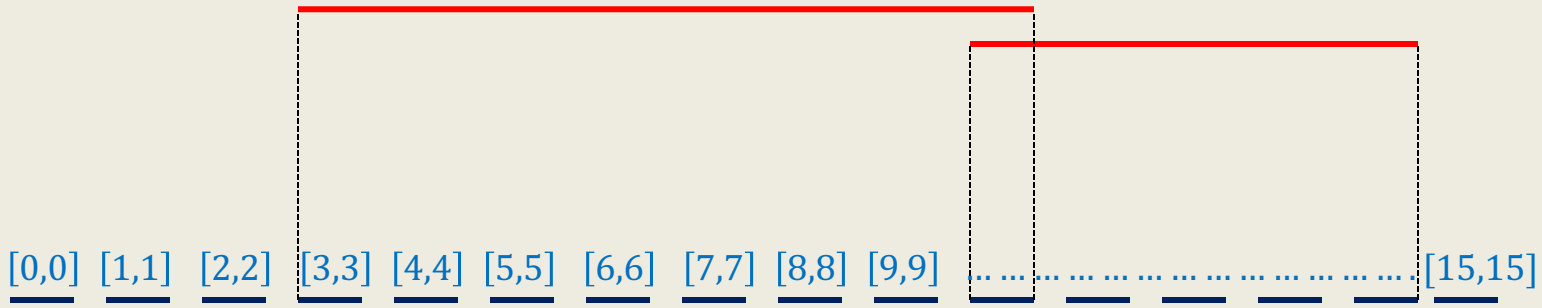
RECAP OF LAST LECTURE

Intervals

S = {[***i***, ***j***], $0 \leq i \leq j < n$}

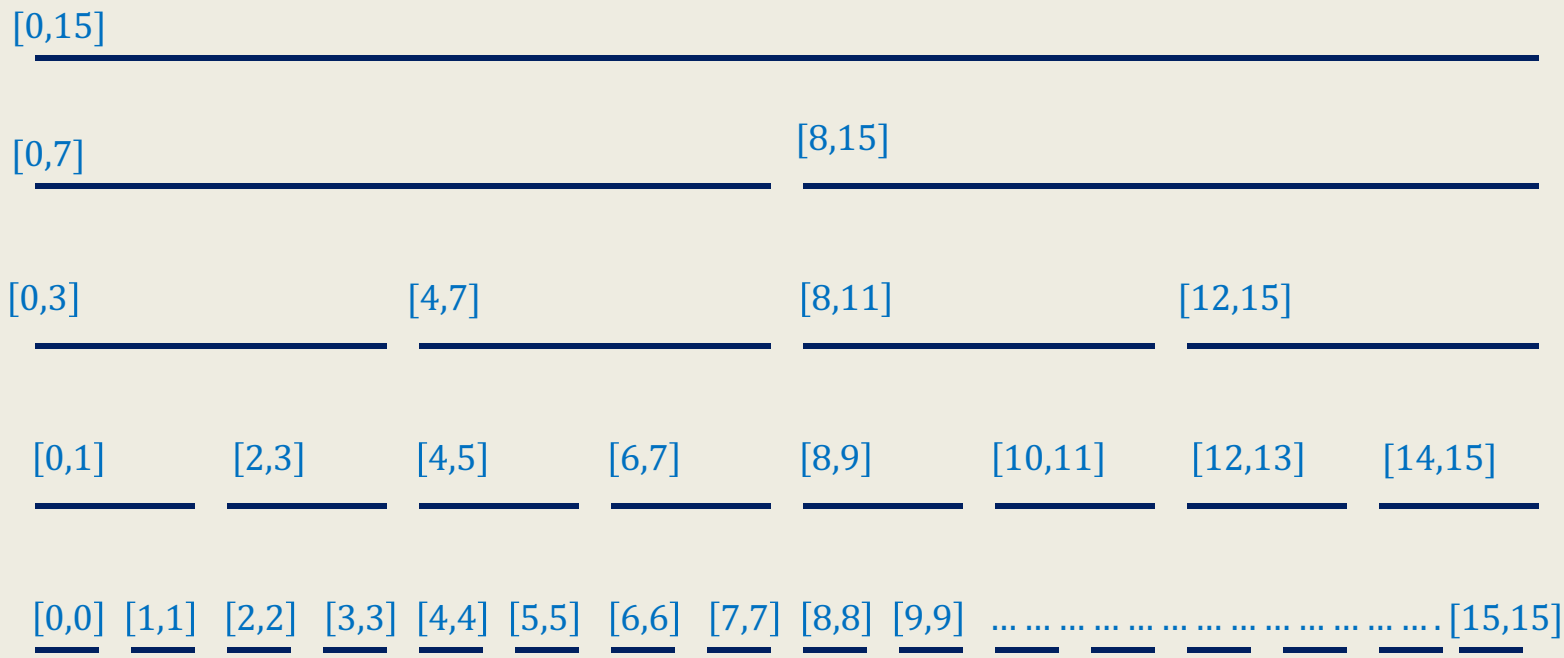
Question: Can we have a small set **X**⊂**S** of **intervals** s.t.

every interval in **S** can be expressed as a union of a few **intervals** from **X** ?

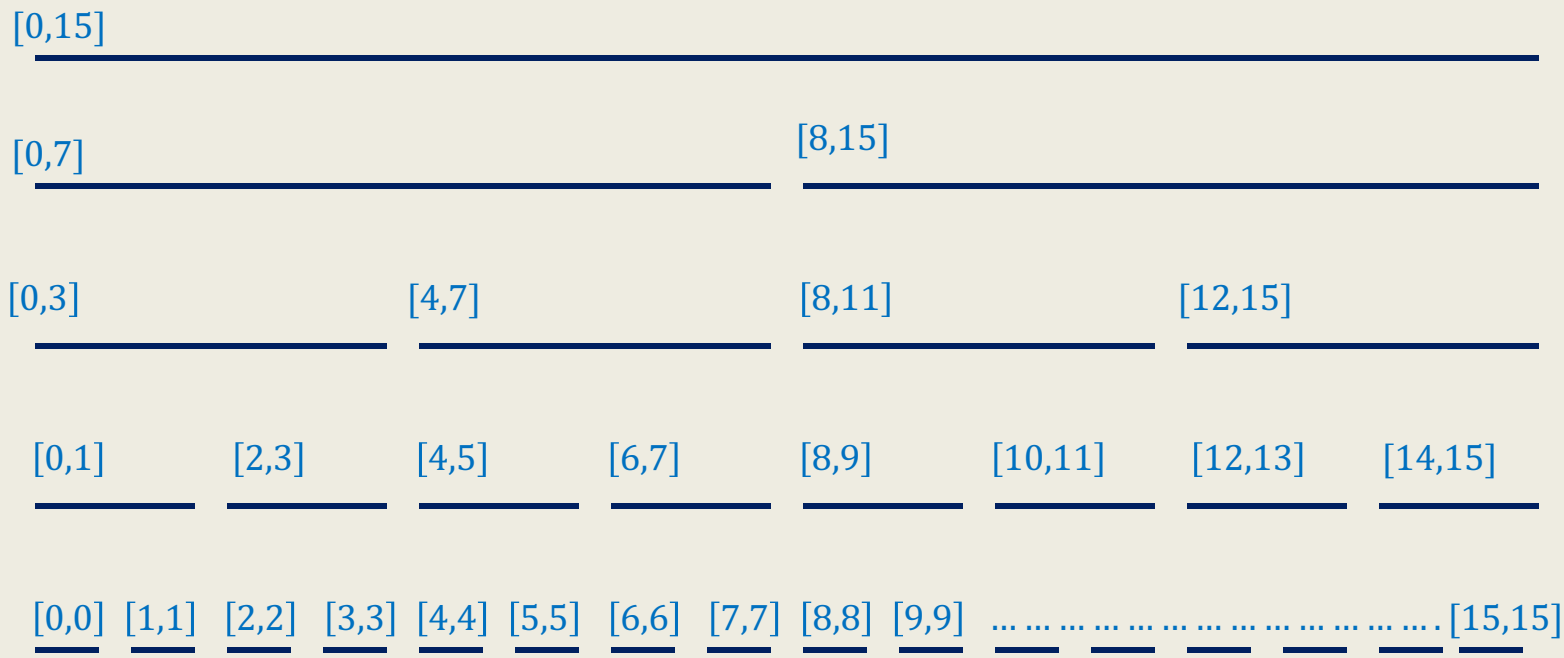


Answer: yes☺

Hierarchy of intervals



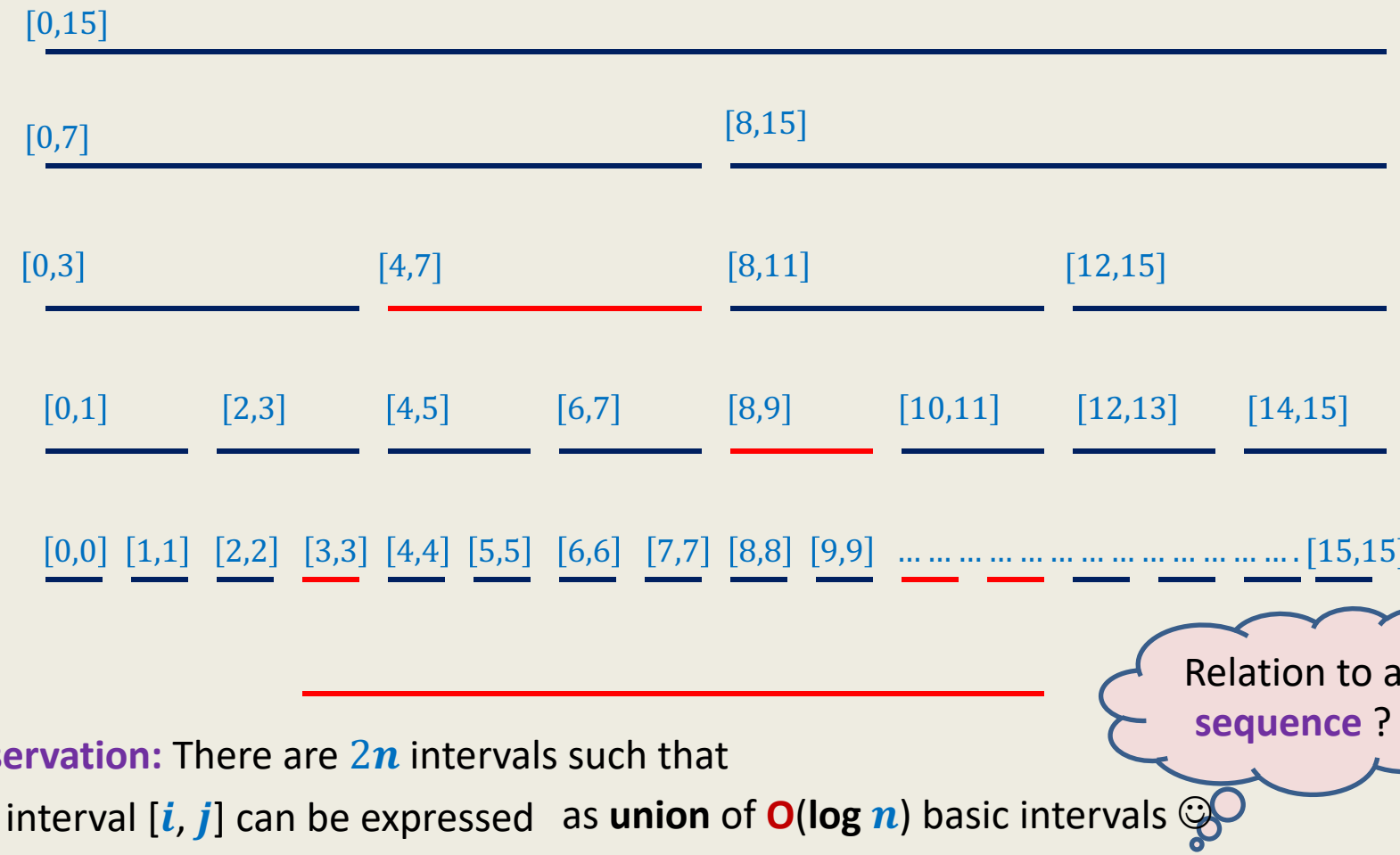
Hierarchy of intervals



Observation: There are $2n$ intervals such that

any interval [i, j] can be expressed as **union** of **O**(**log** n) basic intervals ☺

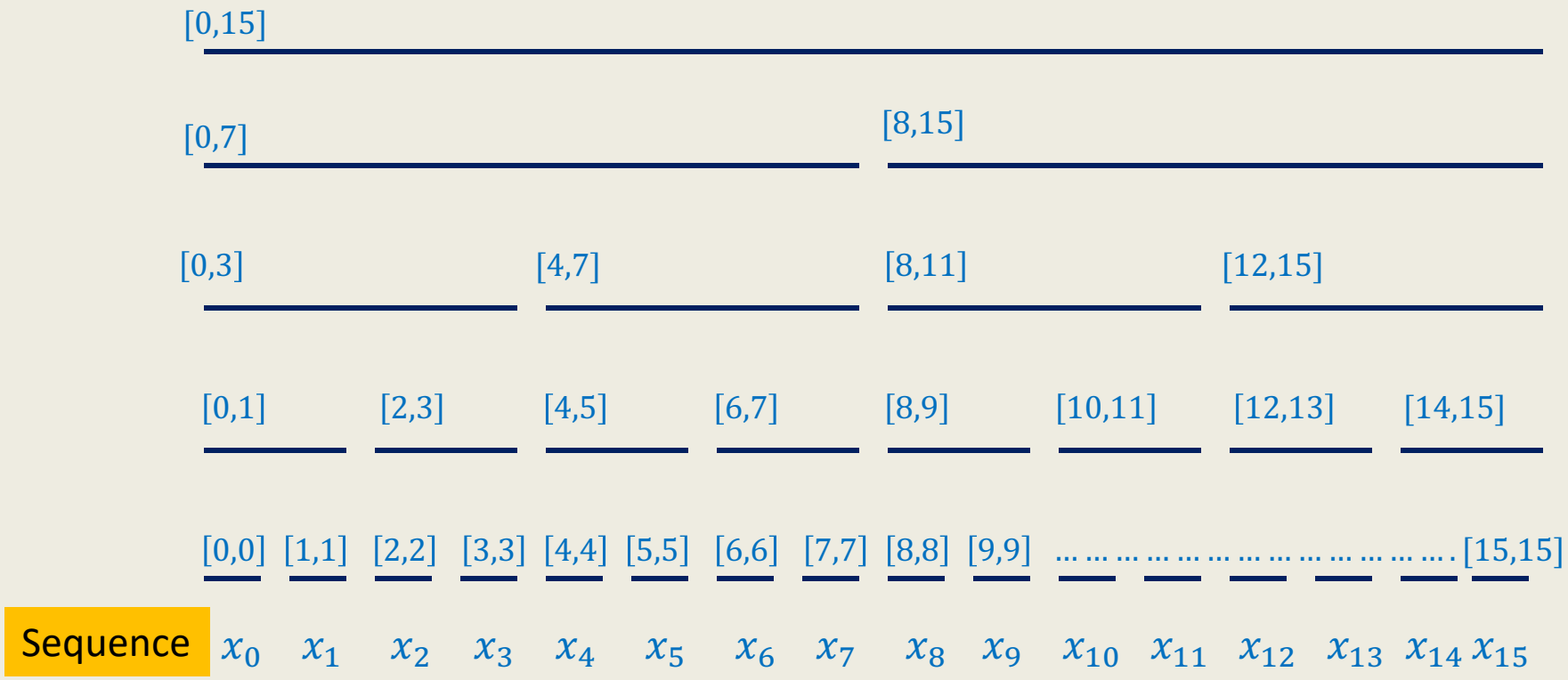
Hierarchy of intervals



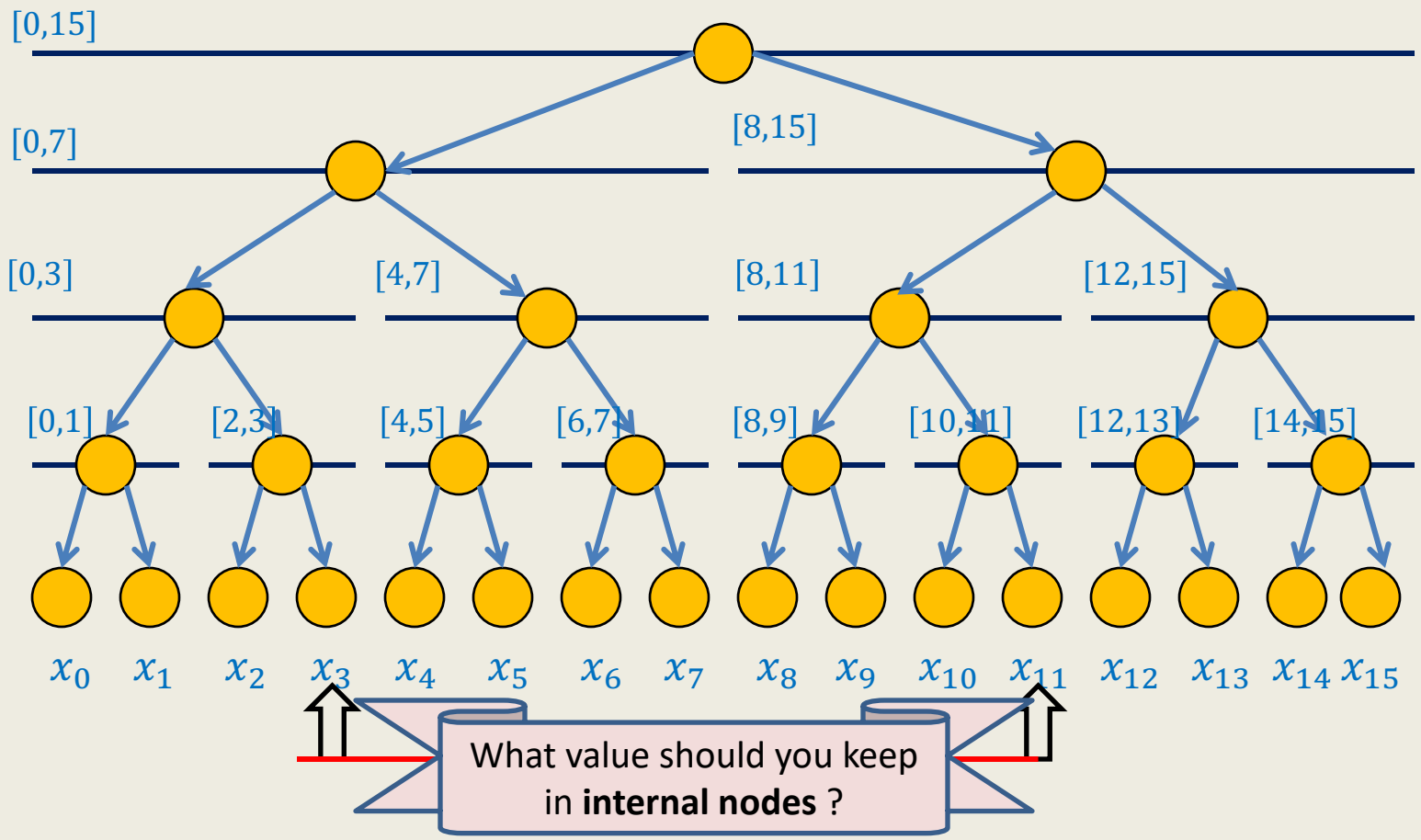
Observation: There are $2n$ intervals such that

any interval [i, j] can be expressed as **union** of **O**(**log** n) basic intervals ☺

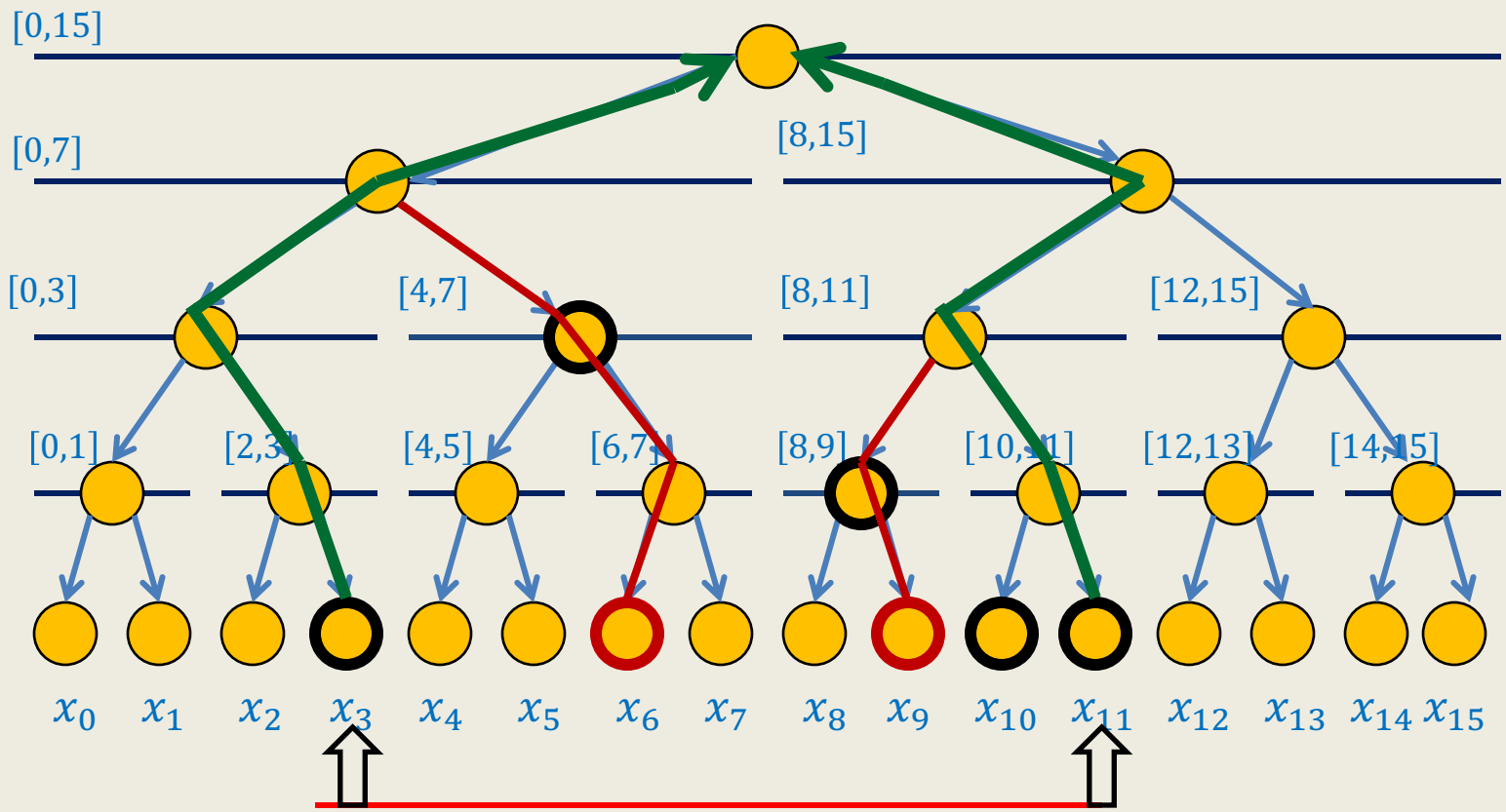
Which data structure emerges ?



A Binary tree



How to perform Operation on an interval ?



How to perform Operation on an interval ?

Problem 2

Dynamic Range-minima

Dynamic Range Minima Problem

Given an initial sequence **S =** $\prec x_0, \ldots, x_{n-1} \succ$ of numbers, maintain a compact data structure to perform the following operations efficiently for any $0 \leq i < j < n$.

- **ReportMin**(i, j):

  Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$

- **Update**(i, **a**):

  a becomes the new value of x_i.

Example:

Let the initial sequence be **S =** $\prec$ 14, 12, 3, 49, 4, 21, 322, -40 $\succ$

ReportMin(1, 5) returns **3**

ReportMin(0, 3) returns **3**

Update(2, **19**) update **S** to $\prec$ 14, 12, **19**, 49, 4, 21, 322, -40 $\succ$

ReportMin(1, 5) returns **4**

ReportMin(0, 3) returns **12**

Dynamic Range Minima Problem

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ of numbers, maintain a compact data structure to perform the following operations efficiently for any $0 \leq i < j < n$.

- **ReportMin**(i, j):

  Report the minimum element from $\{x_k \mid \text{for each } i \leq k \leq j\}$

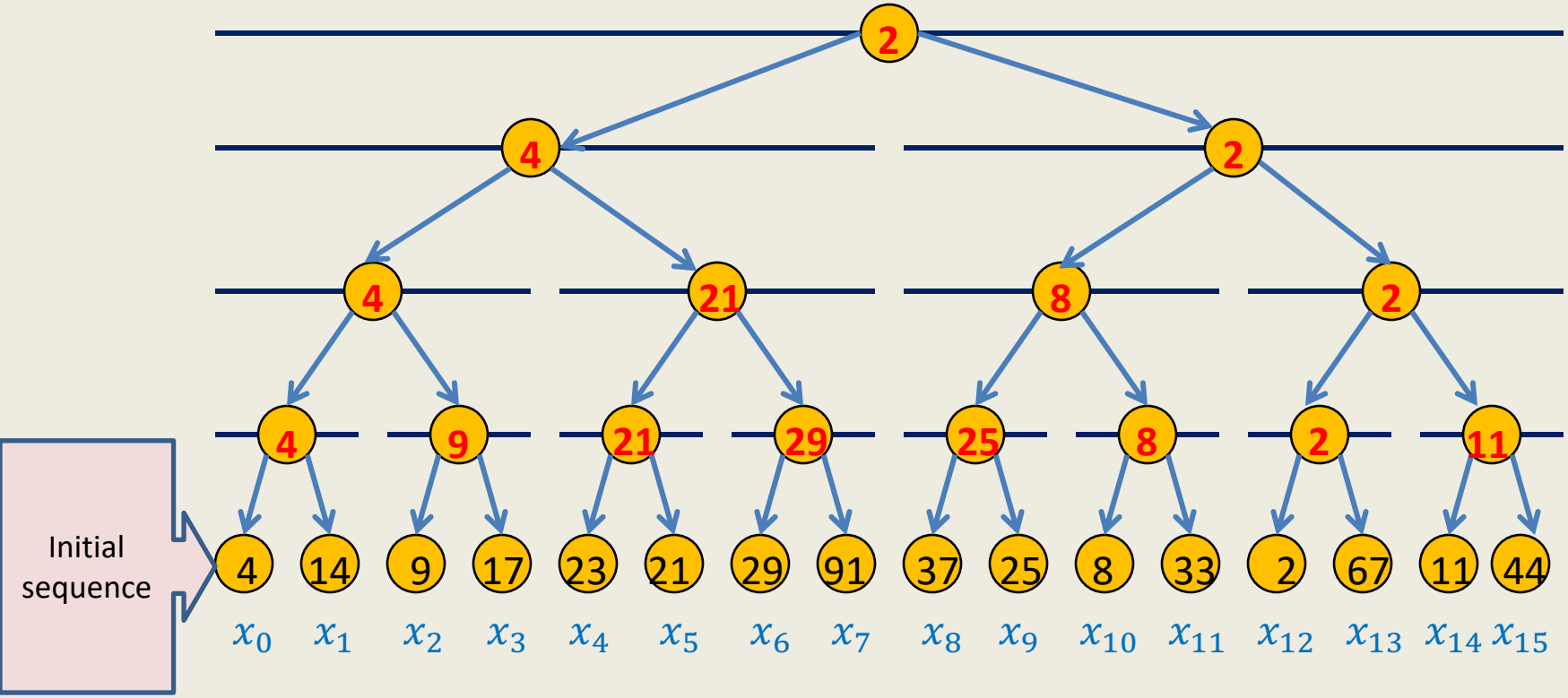
- **Update**(i, **a**):

  a becomes the new value of x_i.

AIM:

- **O**(n) size data structure.
- **ReportMin**(i, j) in **O**(log n) time.
- **Update**(i, **a**) in **O**(log n) time.

Data structure for dynamic range minima

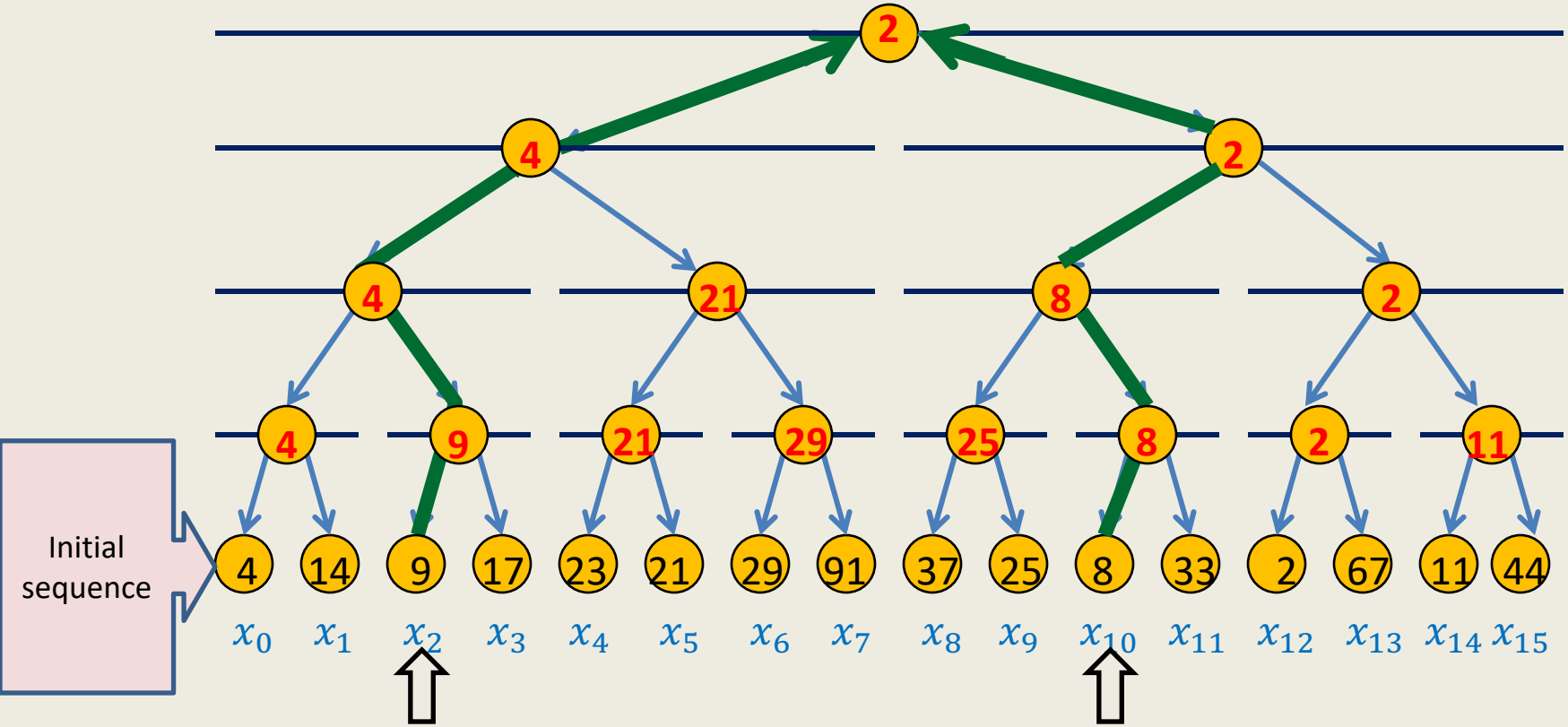


Question: What should be stored in an internal node **v** ?

Answer:

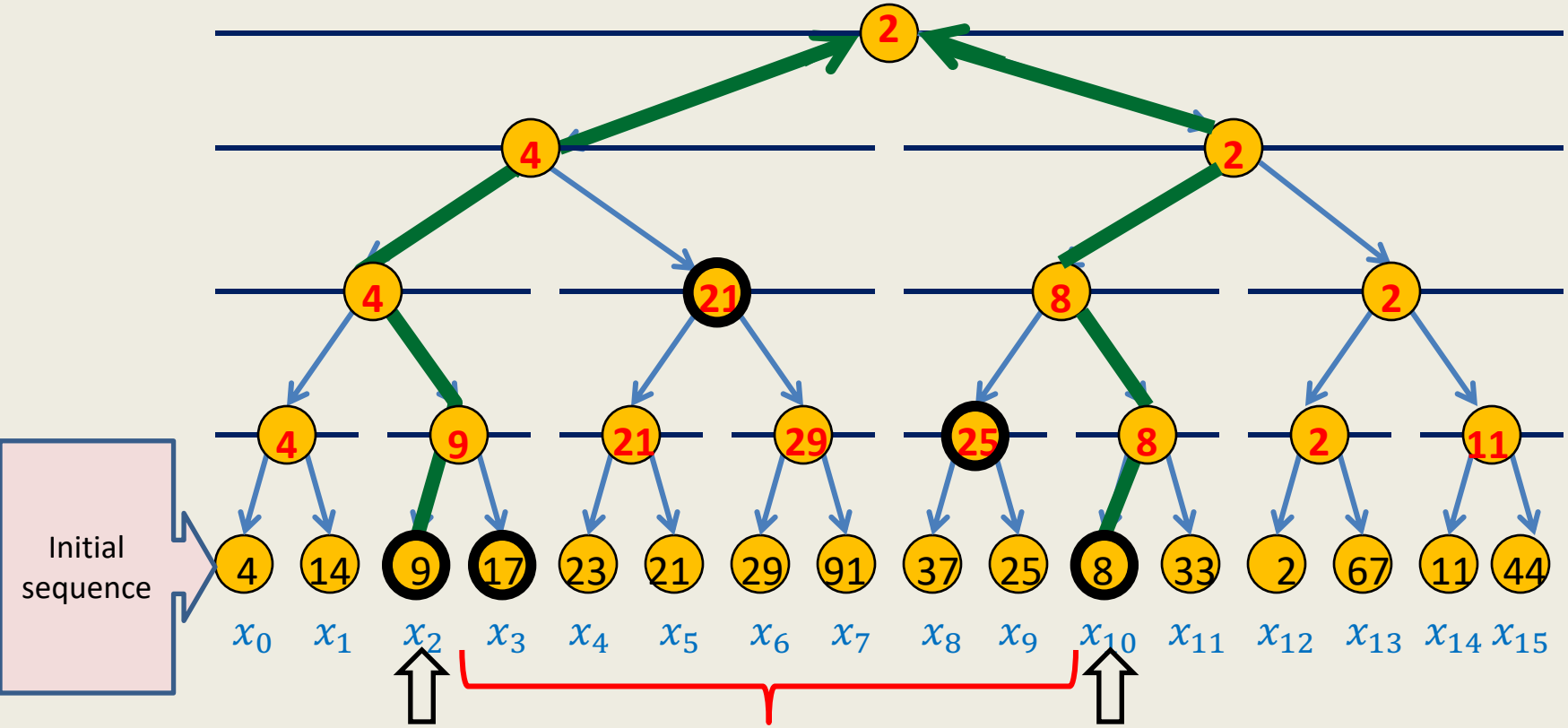
minimum value

Data structure for dynamic range minima



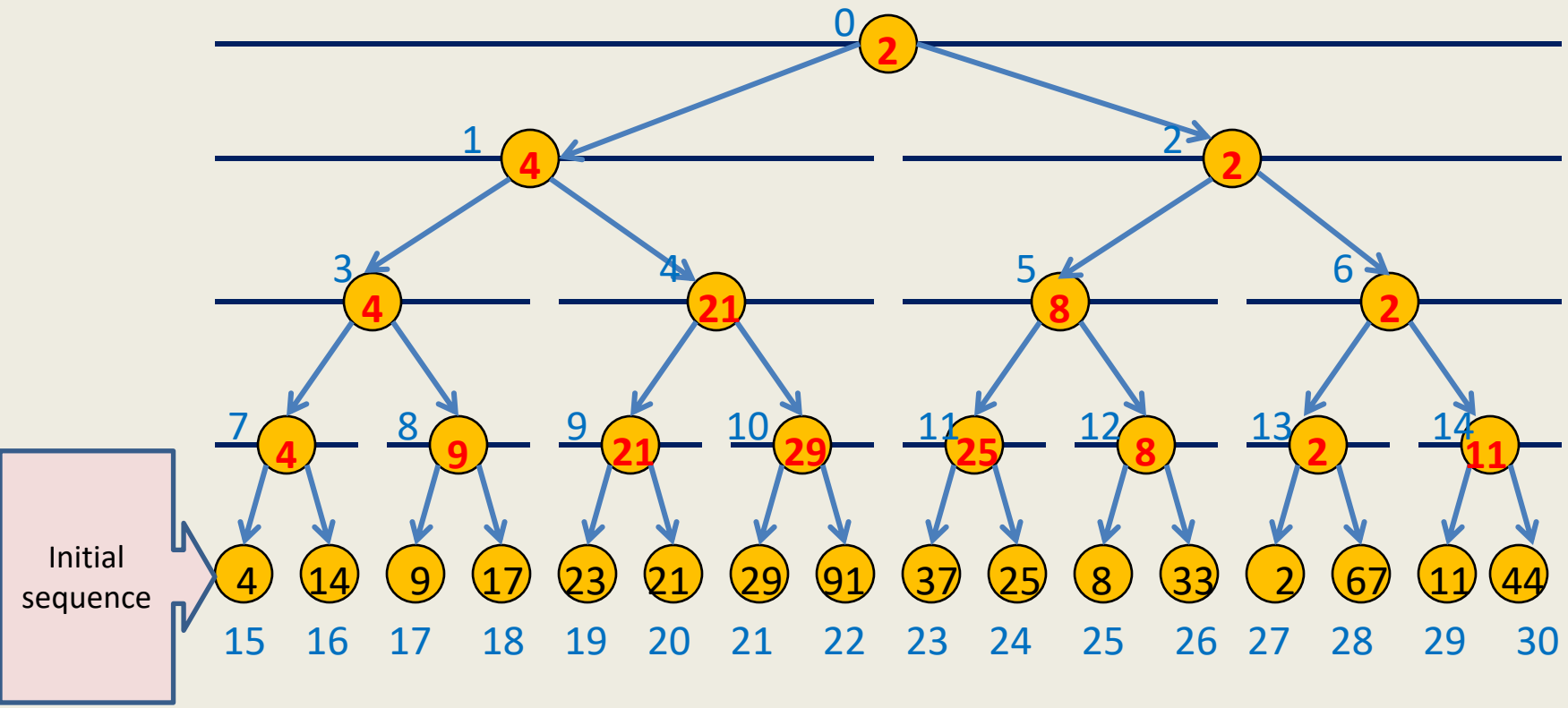
How to do Report-Min(2,10) ?

Data structure for dynamic range minima



How to do Report-Min(2,10) ?

Data structure for dynamic range minima



Data structure: An array **A** of size 2**n**-1.

Copy the sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ into **A**$[n-1]$…**A**$[2n-2]$

Leaf node corresponding to x_i = **A**$[(n-1)+i]$

How to check if a node is left child or right child of its parent ?

(if index of the node is odd, then the node is left child, else the node is right child)

Update(i, a)

Update(i, a)

$i \leftarrow (n-1) + i$;

A[i**]**$\leftarrow a$;

$i \leftarrow \lfloor (i-1)/2 \rfloor$;

While(?? **)**

{

}

Homework

At the end of the lecture slides, an **incorrect** pseudocode is given for **Update(i, a)**.

It is followed by a **correct** one.

Ponder over it ☺

Report-Min(i,j)

Report-Min(i,j)

$i \leftarrow (n-1)+i$;

$j \leftarrow (n-1)+j$;

min ← A(i);

If (j > i)

{ If (A(j)< min) min ← A(j);

While($\lfloor (i-1)/2 \rfloor <> \lfloor (j-1)/2 \rfloor$)

{

If(i%2=1 and A($i+1$)< min) min ← ;

If(j%2=0 and A($j-1$)< min) min ← ;

i ← ;

j ← ;

}

}

return min;

Another interesting problem on sequences

Practice Problem

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ of n numbers,

maintain a compact data structure to perform the following operations efficiently :

- **Report_min**(i , j):

  Report the minimum element from $\{x_i, \ldots x\_j\}$.

- **Multi-Increment**(i, j, **Δ**):

  Add **Δ** to each x_k for each $i \leq k \leq j$

Example:

Let the initial sequence be **S** = $\prec$ 14, 12, 3, 12, 111, 51, 321, -40 $\succ$

Report_min(1 , 4):

returns 3

Multi-Increment(2,6,10):

S becomes $\prec$ 14, 12, **13**, **22**, **121**, **61**, **331**, -40 $\succ$

Report_min(1 , 4):

returns 12

An challenging problem on sequences

For summer vacation

(not for the exam)

∗ Problem

Given an initial sequence **S** = $\prec x_0, \ldots, x_{n-1} \succ$ of n numbers,

maintain a compact data structure to perform the following operations efficiently :

- **Report_min**(i , j):

  Report the minimum element from $\{x_i, \ldots x\_j\}$.

- **Multi-Increment**(i, j, **Δ**):

  Add **Δ** to each x_k for each $i \leq k \leq j$

- **Rotate**(i , j):

  $x_i \leftrightarrow x_j$, $x_{i+1} \leftrightarrow x_{j-1}$, ...

Example:

Let the initial sequence be **S** = $\prec$ 14, 12, 23, 19, 111, 51, 321, -40 $\succ$

After **Rotate**(1,6), **S** becomes

$\prec$ 14, **321**, **51**, **111**, **19**, **23**, **12**, -40 $\succ$

The solution of this problem will require additional tools. But all these tools have already been discussed in the course.

Problem 4

A data structure for sets

Sets under operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \ldots \{n-1\}$

Aim: a compact data structure to perform

- **Union**(i, j):

  Unite the two sets containing i and j.

- **Same_sets**(i, j):

  Determine if i and j belong to the same set.

Trivial Solution 1

Keep an array **Label[]** such that

Label[i]=**Label**[j] if and only if i and j belong to the same set.

➔ **Same_sets**(i, j):

check if **Label**[i]=**Label**[j] ?

O(**1**) time

➔ **Union**(i, j):

For each $0 \leq k < n$

if (**Label**[k]= **Label**[i]) **Label**[k] ← **Label**[j])

O(n) time

Sets under operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \ldots \{n-1\}$

Aim: a compact data structure to perform

- **Union**(i, j):
  Unite the two sets containing i and j.
- **Same_sets**(i, j):
  Determine if i and j belong to the same set.

Trivial Solution 2

Treat the problem as a graph problem: ??

Connected component

- **V** = {0,…, $n-1$}, **E** = empty set initially.
- A set ⇔
- Keep array **Label[]** such that **Label**[i]=**Label**[j] iff i and j belong to the same component.

➔

Union(i, j) :

O(n) time

add an edge (i, j) and

recompute connected components using **BFS/DFS**.

Sets under operations

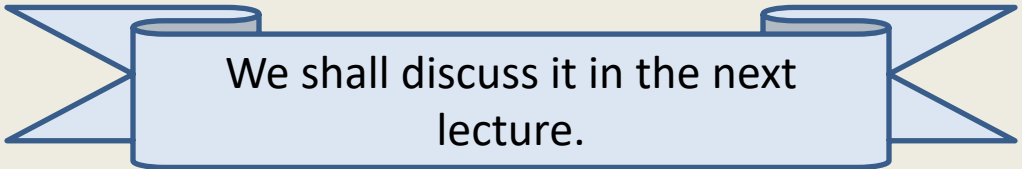
Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \ldots \{n-1\}$

Aim: a compact data structure to perform

- **Union**(i, j):
  Unite the two sets containing i and j.
- **Same_sets**(i, j):
  Determine if i and j belong to the same set.

Efficient solution:

- A data structure which supports each operation in **O(log n)** time.
- **An additional heuristic**
  ➔ time complexity of an operation :



Homework

For **Dynamic Range-minima** problem

Update(i, a)

Update(i, a)

$i \leftarrow (n-1) + i$;

A[i]$\leftarrow a$;

$i \leftarrow \lfloor (i-1)/2 \rfloor$;

While($i \geq 0$ **)**

{

If(a**<** A[i]) A[i]$\leftarrow a$**;**

$i \leftarrow \lfloor (i-1)/2 \rfloor$;

}

There is an **error** in the above pseudocode. Try spotting it.

Update(i, a)

Update(i, a)

$i \leftarrow (n-1) + i$;

A[i] $\leftarrow a$;

$i \leftarrow \lfloor (i-1)/2 \rfloor$;

While($i \geq 0$ **)**

{

If(A[$2i+1$**] < A[**$2i+2$**])**

A[i**] $\leftarrow$ A[**$2i+1$**]**

else

A[i**] $\leftarrow$ A[**$2i+2$**] ;**

$i \leftarrow \lfloor (i-1)/2 \rfloor$;

}

Here is the correct pseudocode.

Data Structures and Algorithms

(ESO207)

Lecture 32

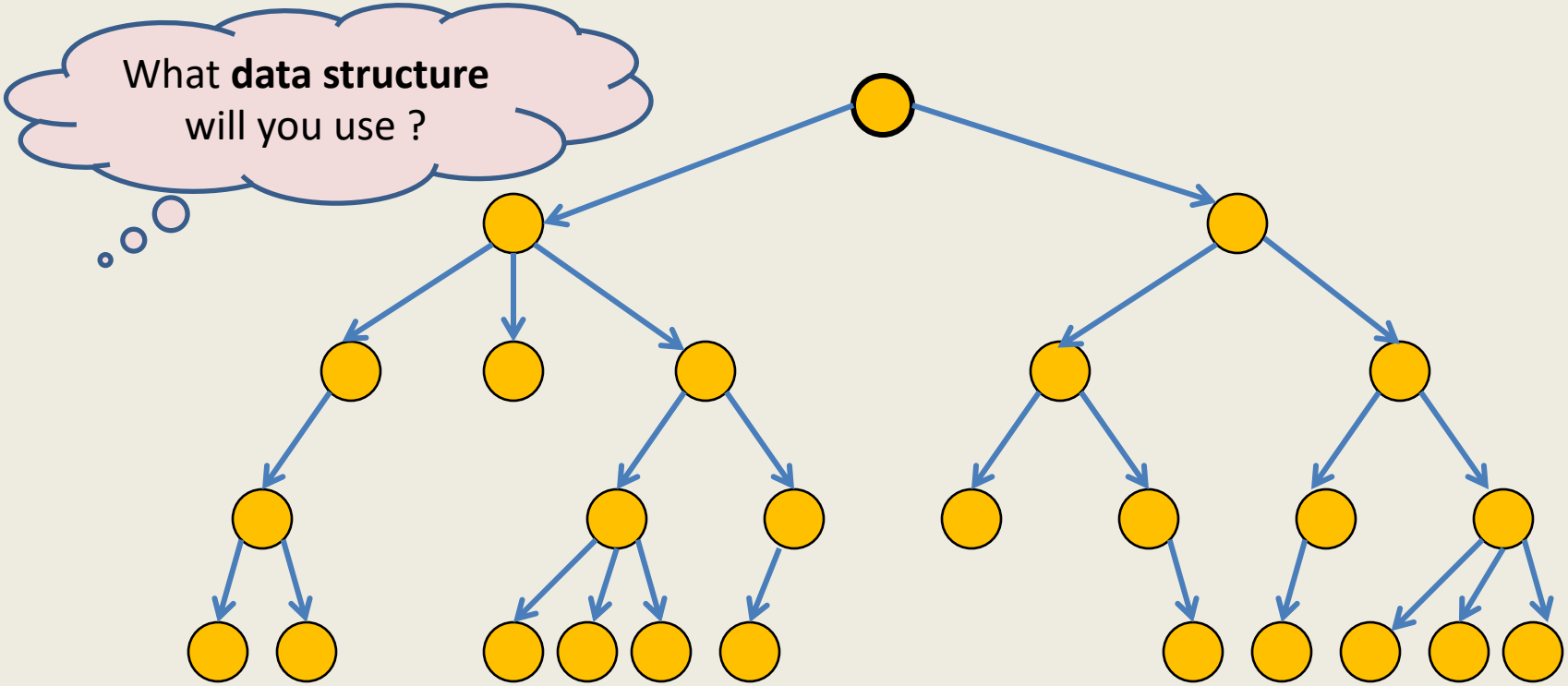
- Magical application of binary trees – III

Data structure for sets

Rooted tree

Revisiting and extending

A typical rooted tree we studied



Definition we gave:

Every vertex, except **root**, has exactly one **incoming** edge and has a path **from** the root.

Examples:

Binary search trees,

DFS tree,

BFS tree.

A typical rooted tree we studied

Question: what data structure can be used for representing a rooted tree ?

Answer:

Data structure 1:

- Each node stores a list of its children.
- To access the tree, we keep a pointer to the root node.

  (there is no way to access any node (other than root) directly in this data structure)

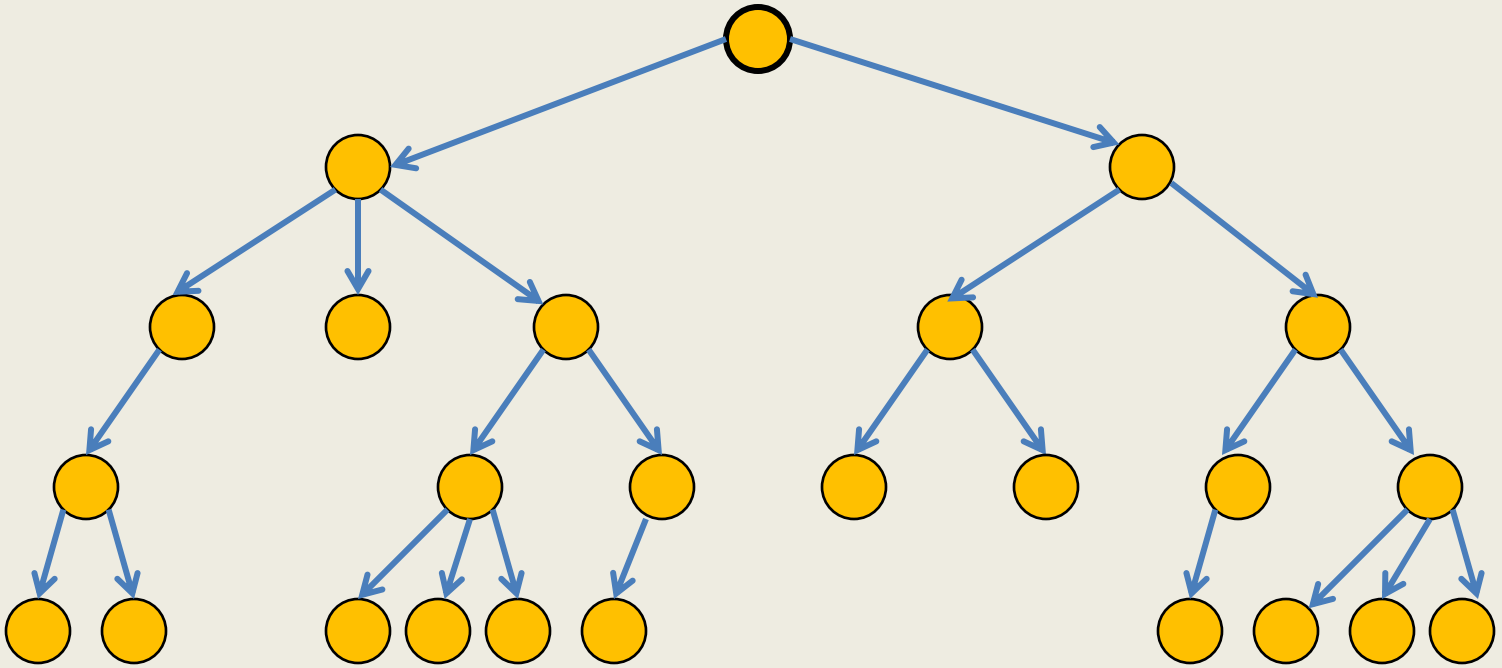
Data structure 2: (If nodes are labeled in a contiguous range [0..n-1])

rooted tree becomes an instance of a **directed graph**.

So we may use **adjacency list** representation.

Advantage: We can access each node directly.

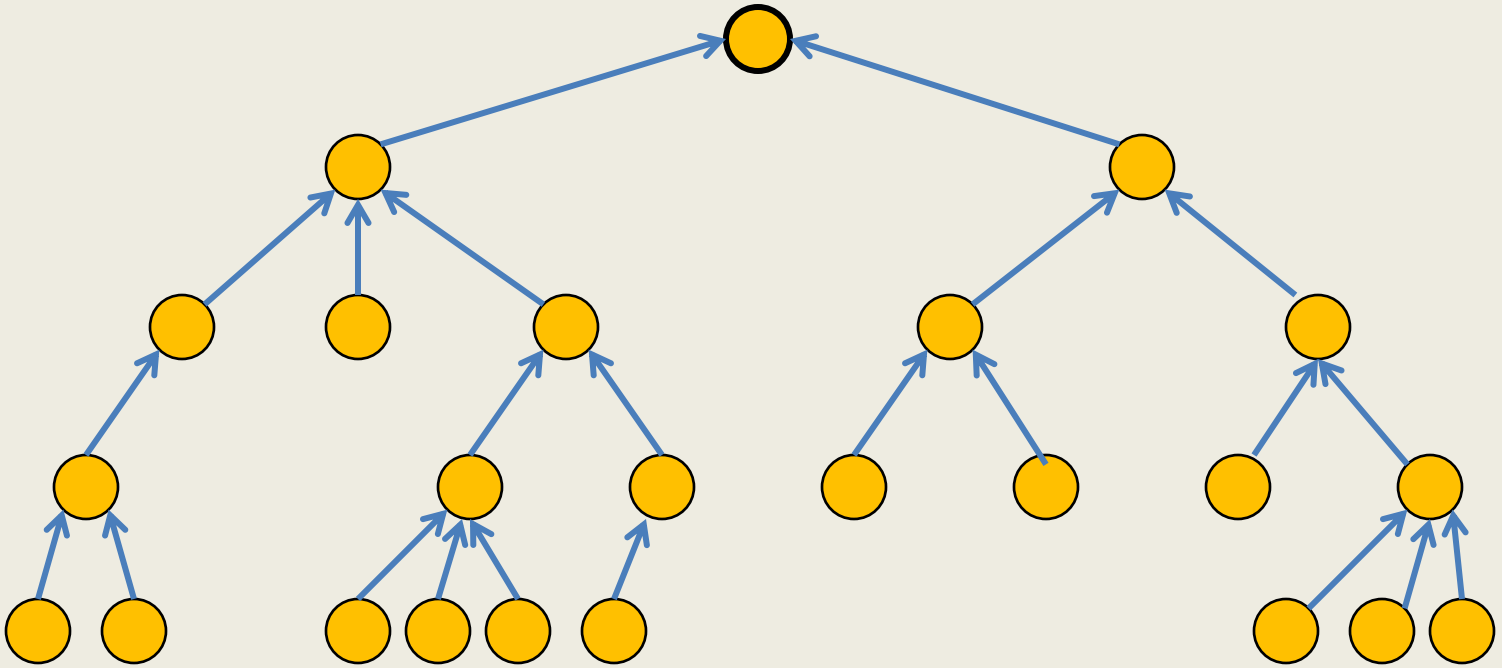
Extending the definition of rooted tree



Extended Definition:

Type 1: Every vertex, except **root**, has exactly one **incoming** edge and has a path **from** the root.

Extending the definition of rooted tree



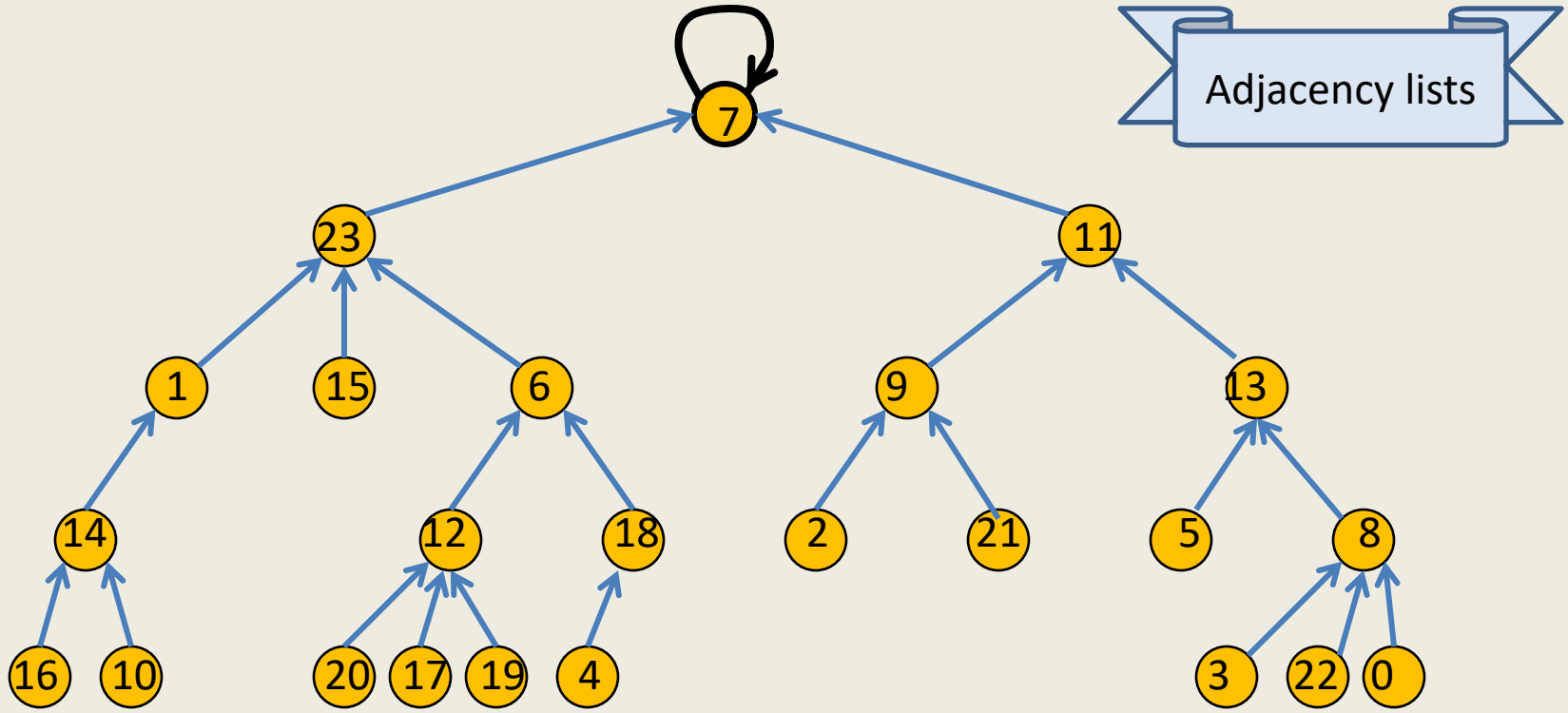
Extended Definition:

Type 1: Every vertex, except **root**, has exactly one **incoming** edge and has a path **from** the root.

OR

Type 2: Every vertex, except root, has exactly one **outgoing** edge and has a path **to** the root.

Data structure for rooted tree of type 2



If nodes are labeled in a **contiguous** range $[0..n-1]$,

there is even simpler and more compact data structure

Guess ??

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Parent	8	23	9	8	18	13	23	7	13	11	14	7	6	11	1	23	14	12	6	12	12	9	8	7

Application of rooted tree of type 2

Maintaining sets

Sets under two operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \ldots \{n-1\}$

Aim: a compact data structure to perform

- **Union**(i, j):
  Unite the two sets containing i and j.
- **Same_sets**(i, j):
  Determine if i and j belong to the same set.

Trivial Solution

Treat the problem as a graph problem: Connected component

- **V** = $\{0, \ldots, n-1\}$, **E** = empty set initially.
- A set ⇔
- Keep array **Label[]**

.

➔

Union(i, j) :
  if (**Same_sets**(i, j) = **false**)
  add an edge (i, j)

O(n) time

Sets under two operations

Given: a collection of n singleton sets $\{0\}, \{1\}, \{2\}, \ldots \{n-1\}$

Aim: a compact data structure to perform

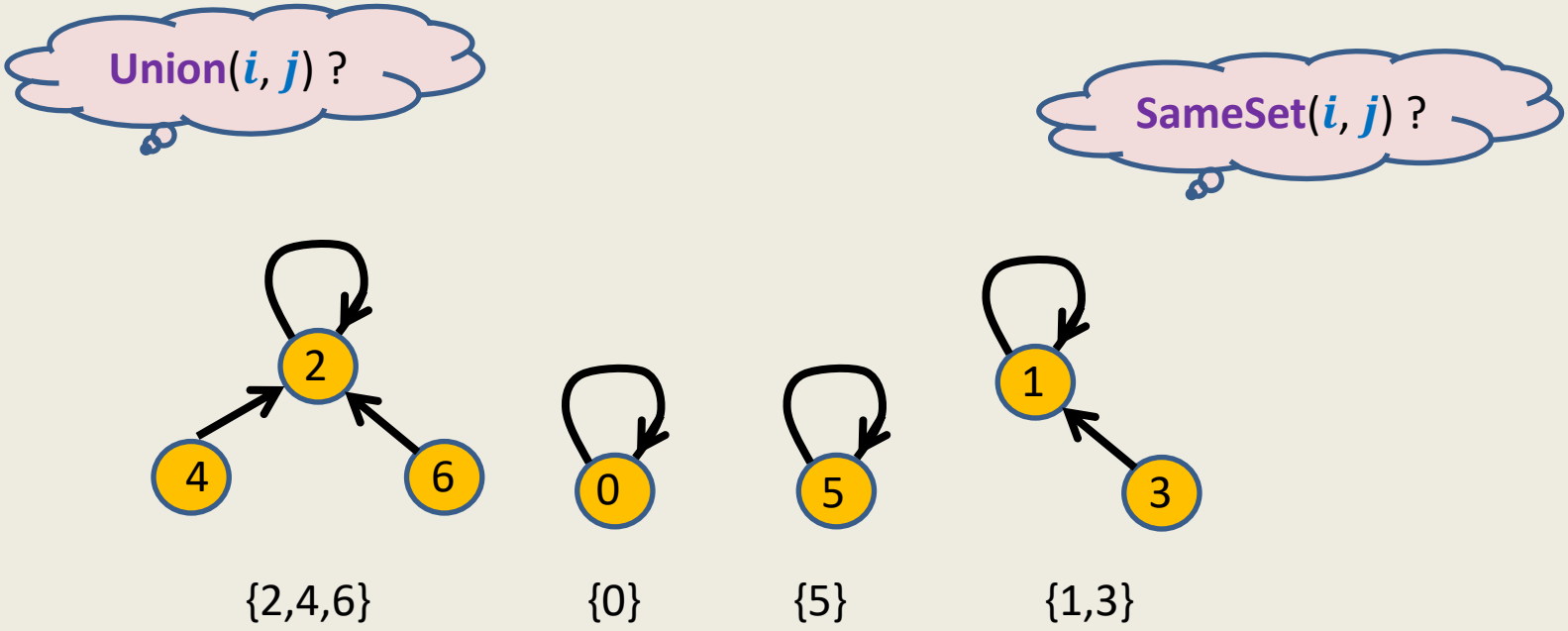
- **Union**(i, j):
  Unite the two sets containing i and j.
- **Same_sets**(i, j):
  Determine if i and j belong to the same set.

Efficient solution:

- A data structure which supports each operation in **O(log n)** time.
- **An additional heuristic**
  ➔ time complexity of an operation : practically **O(1)**.

Data structure for sets

Maintain each set as a rooted tree.



Data structure for sets

Maintain each set as a rooted tree.

Question: How to perform operation **Same_sets**(i, j) ?

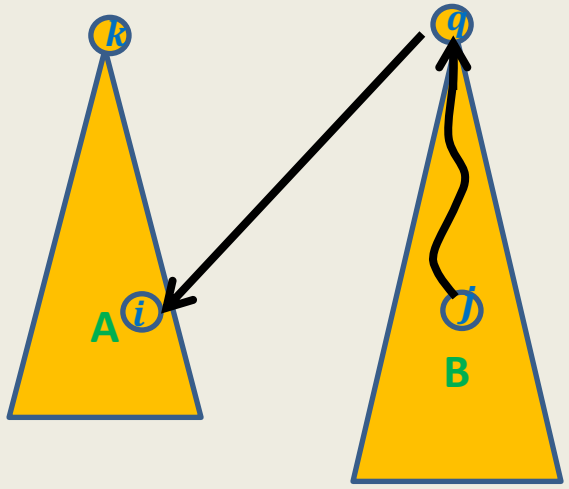
Answer: Determine if i and j belong to the same tree.

➔ **find** root of i

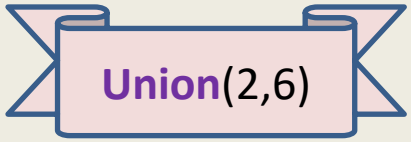
Question: How to perform **Union**(i, j) ?

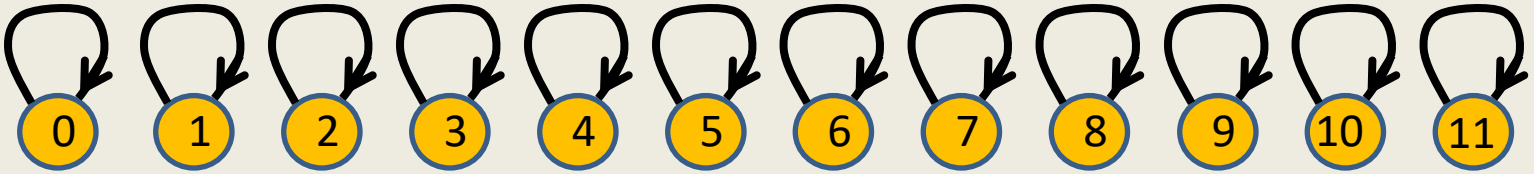
Answer:

- **find** root of j; let it be q.
- **Parent**(q) ← i.



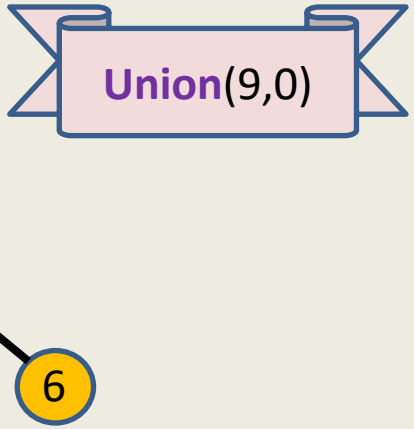
A rooted tree as a data structure for sets





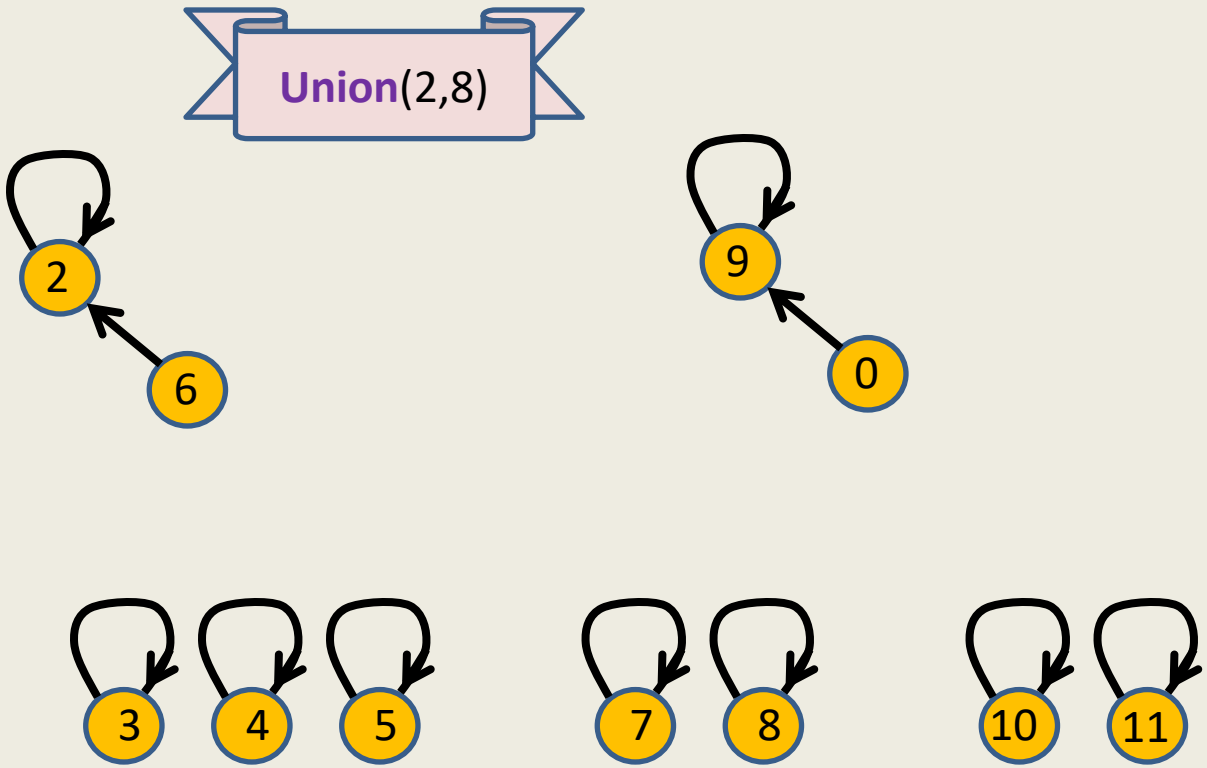
Parent	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

A rooted tree as a data structure for sets



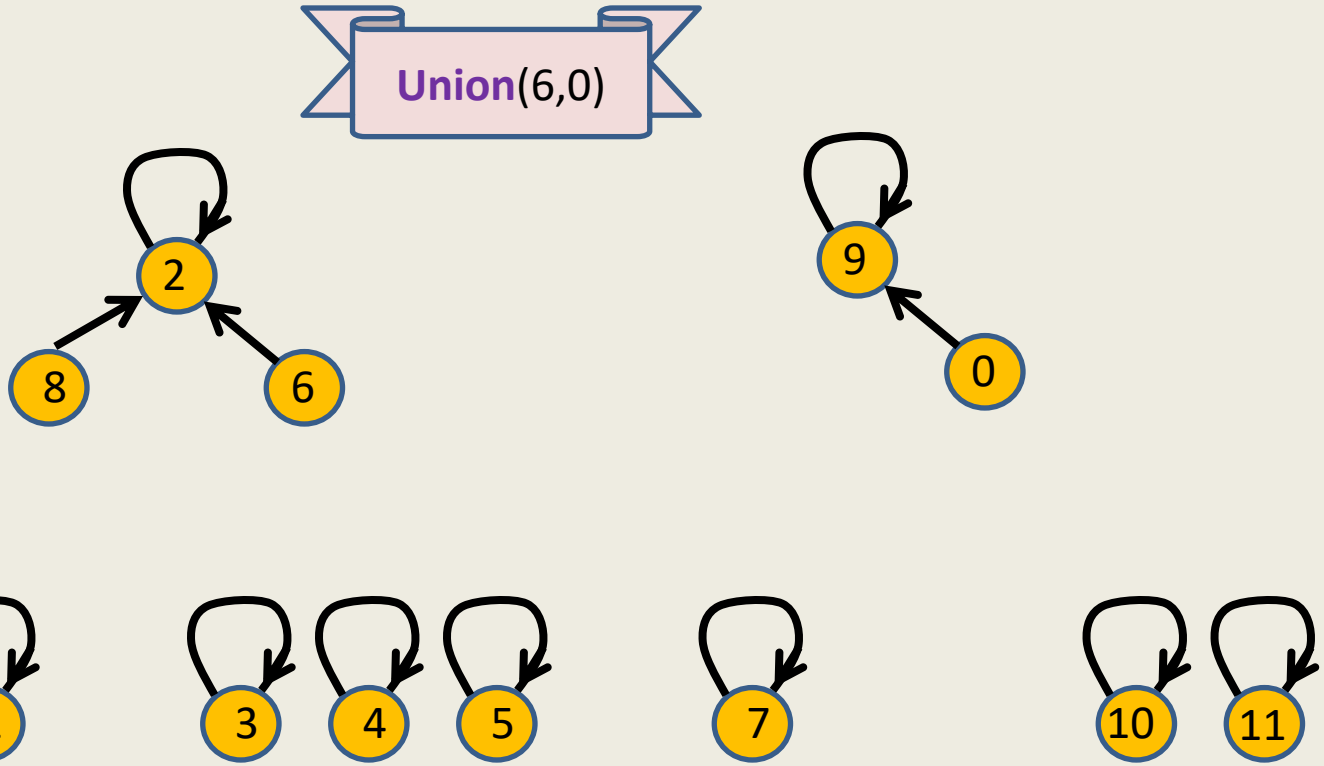
Parent	0	1	2	3	4	5	2	7	8	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

A rooted tree as a data structure for sets



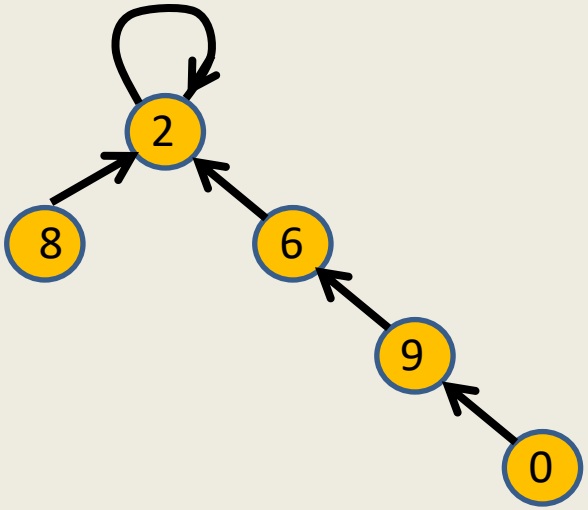
Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

A rooted tree as a data structure for sets



Parent	9	1	2	3	4	5	2	7	2	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

A rooted tree as a data structure for sets



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**7**	**2**	**6**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

Pseudocode for Union and SameSet()

```
Find(i)    // subroutine for finding the root of the tree containing i
      If (Parent(i) = i)     return i ;
      else  return Find(Parent(i));
```

```
SameSet(i, j)
      k ← Find(i);
      l ← Find(j);
      If (k = l)     return true else return false

Union(i, j)
      k ← Find(j);
      Parent(k)← i;
```

Observation: Time complexity of Union(i, j) as well as Same_sets(i, j) is governed by the time complexity of Find(i) and Find(j).

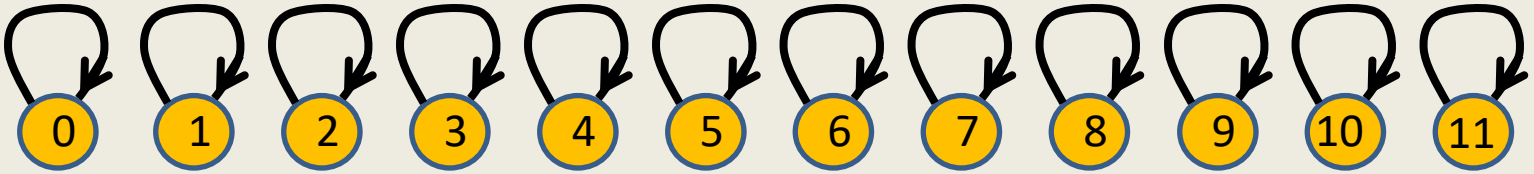
Question: What is time complexity of Find(i) ?

Answer: depth of the node i in the tree containing i.

Time complexity of Find(i)

Union(0,1)
Union(1,2)
Union(2,3)
…
Union(9,10)
Union(10,11)

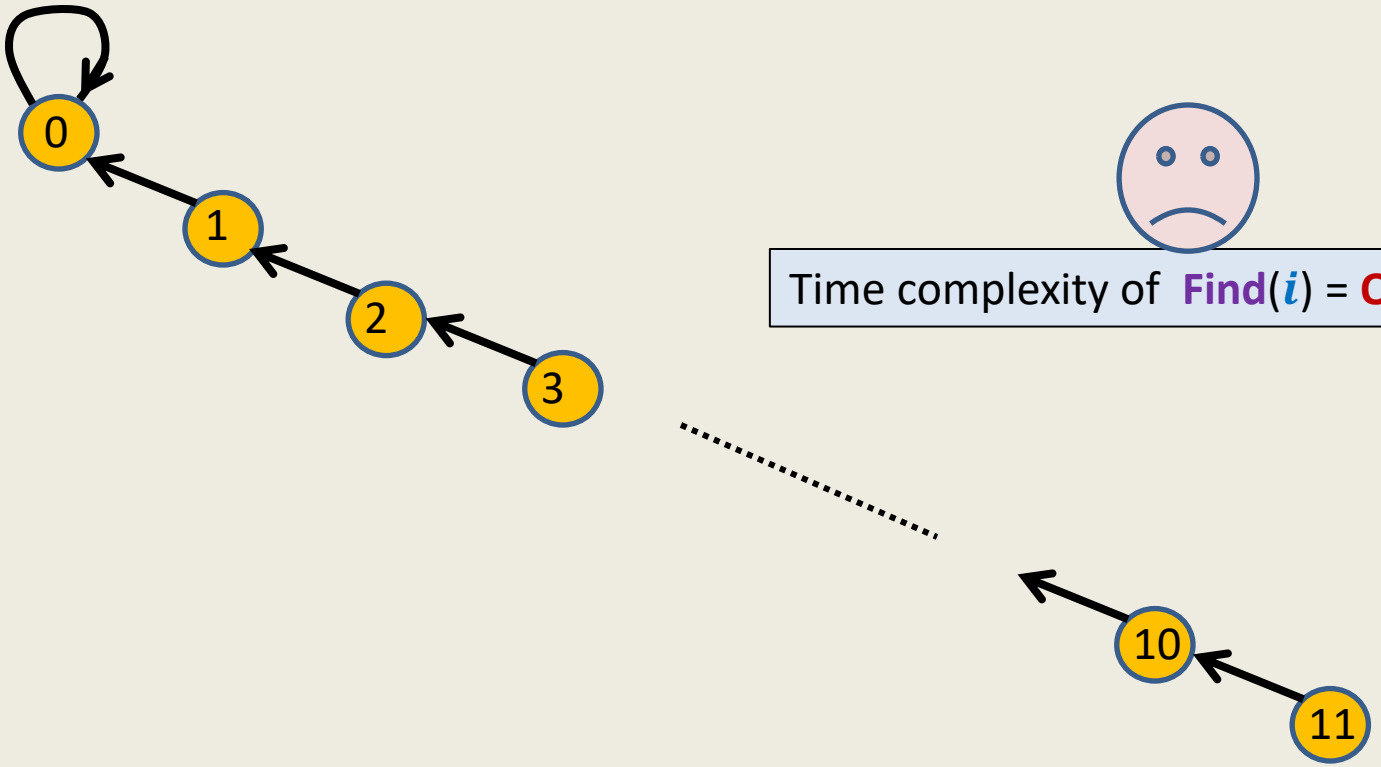
What will be the rooted tree structures after these union operations ?



Parent	0	1	2	3	4	5	6	7	8	9	10	11
	0	1	2	3	4	5	6	7	8	9	10	11

Time complexity of Find(i)

Union(0,1)
Union(1,2)
Union(2,3)
…
Union(9,10)
Union(10,11)



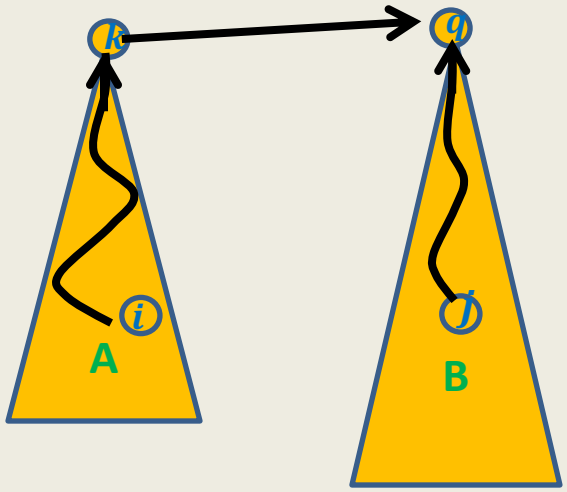
Time complexity of **Find**(i) = **O**(n)

Parent	**0**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	0	1	2	3	4	5	6	7	8	9	10	11

Improving the time complexity of Find(i)

Heuristic 1: Union by size

Improving the Time complexity



Key idea: Change the **union**(i,j) .

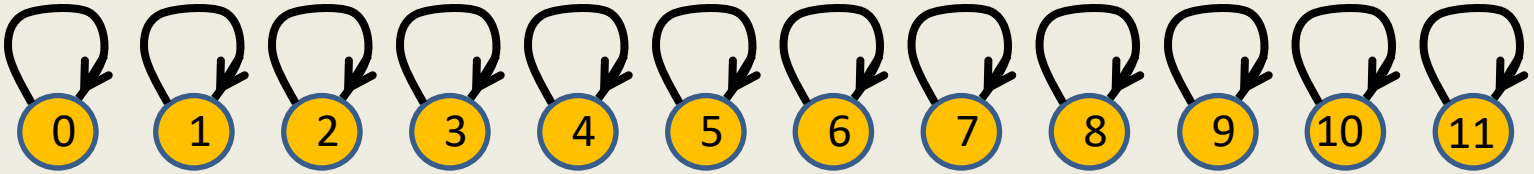
While doing **union**(i,j), hook the **smaller size** tree to the **root of the bigger size tree**.

For this purpose, keep an array **size[0,..,n-1]**

size[i] = number of nodes in the tree containing i

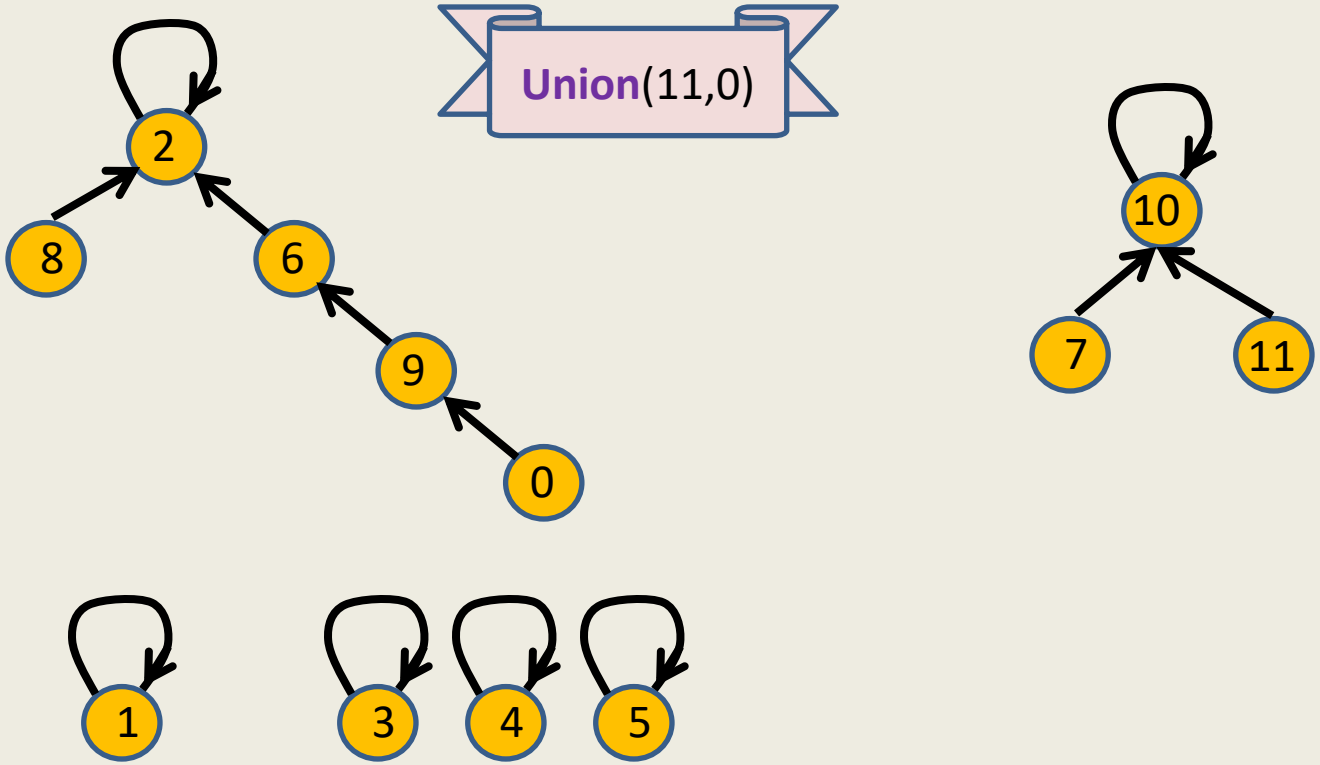
(**if** i is a **root** and zero otherwise)

Efficient data structure for sets



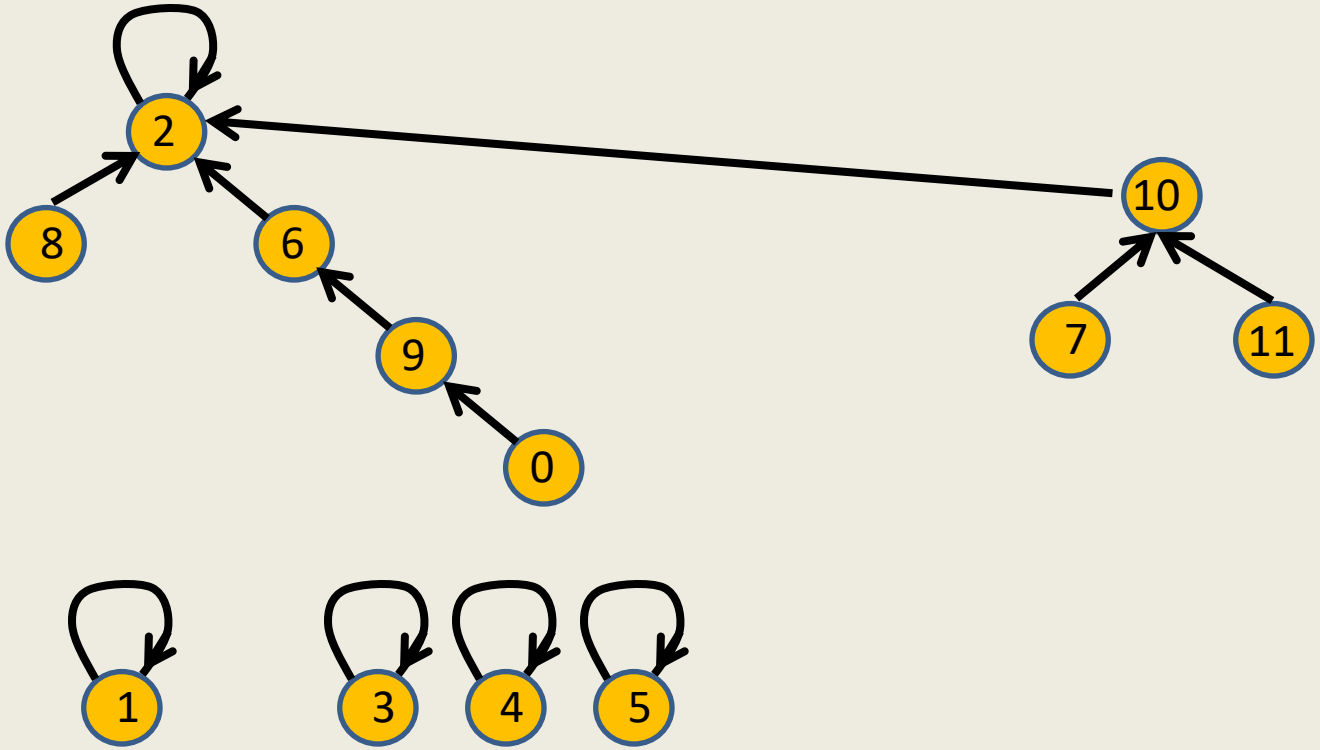
Parent	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11
size	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Efficient data structure for sets



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**10**	**2**	**6**	**10**	**10**
	0	1	2	3	4	5	6	7	8	9	10	11
size	**0**	**1**	**5**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**3**	**0**

Efficient data structure for sets



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**10**	**2**	**6**	**10**	**10**
	0	1	2	3	4	5	6	7	8	9	10	11
size	**0**	**1**	**5**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**3**	**0**

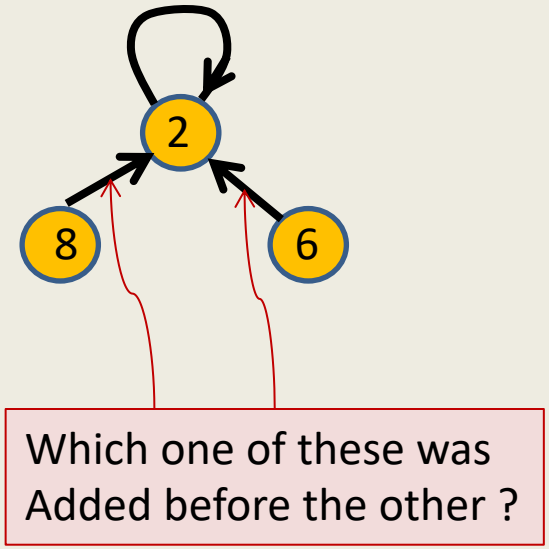
Pseudocode for modified Union

```
Union(i, j)
        k ← Find(i);
        l ← Find(j);
        If(size(k) < size(l) )
                l ← Parent(k);
                size(l) ← size(k) + size(l);
                size(k) ← 0;
        Else
                k ← Parent(l);
                size(k) ← size(k) + size(l);
                size(l) ← 0;
```

Question: How to show that Find(i) for any i will now take O(log n) time only ?

Answer: It suffices if we can show that **Depth(i)** is O(log n).

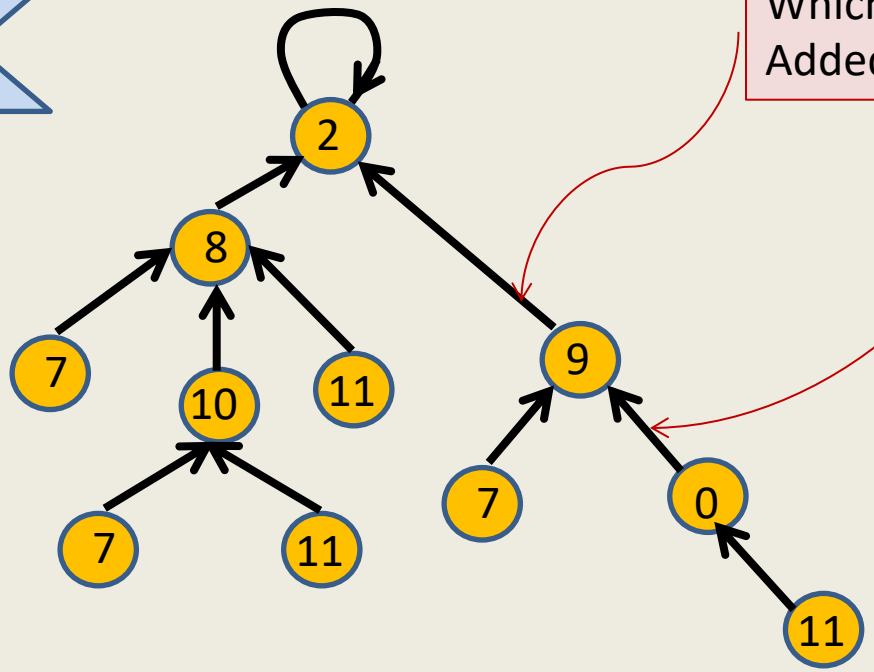
Can we infer history of a tree?



Answer: Can not be inferred with any certainty ☹.

Can we infer history of a tree?

During union, we join **roots** of two trees.

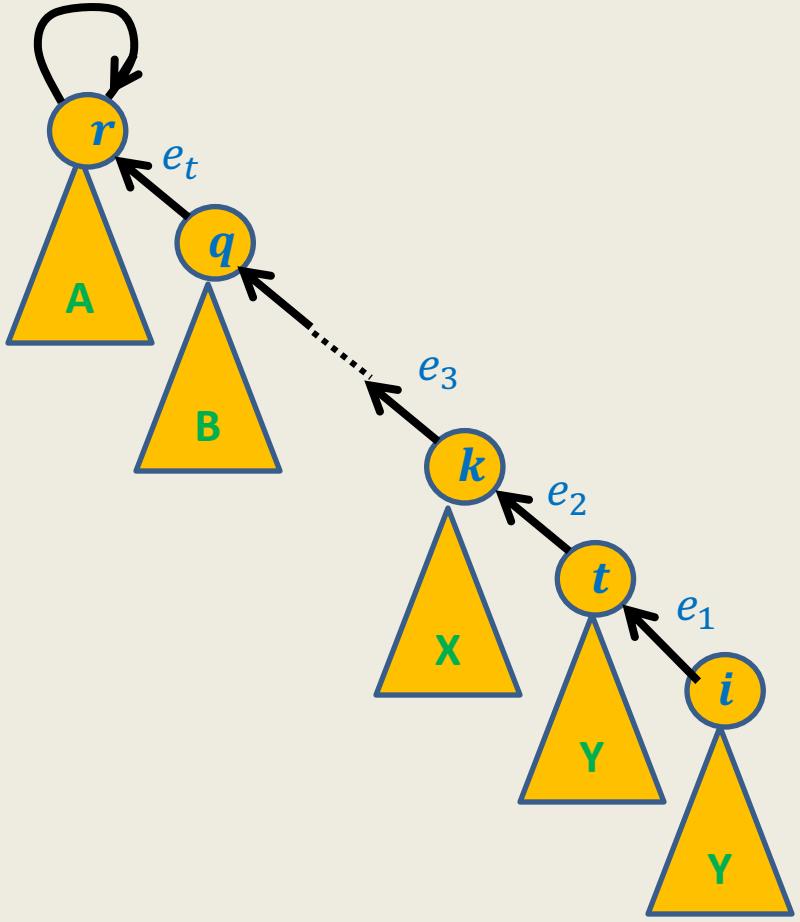


Which one of these was Added before the other ?

➔ (0→9) was added **before** (9→2).

Theorem: The edges on a **path** from node v to root were inserted in the order they appear on the **path**.

How to show that depth of any element = O(log n) ?



Let e_1, e_2, …, e_t be the edges on the path from i to the **root.**

Let us visit the history.
(how this tree came into being ?).

Edges e_1, e_2, …, e_t would have been added in the order:

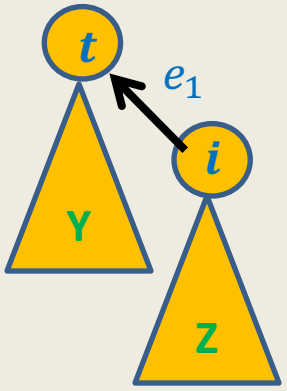
e_1
e_2
…
e_t

How to show that depth of any element = O(log n) ?

Let e_1, e_2, …, e_t be the edges on the path from i to the **root.**

Consider the moment just before edge e_1 is inserted.

Let no. of elements in subtree $T(i)$ at that moment be n_i.



We added edge $i \rightarrow t$ (and **not** $t \rightarrow i$).

➔ no. of elements in $T(t) \geq n_i$.

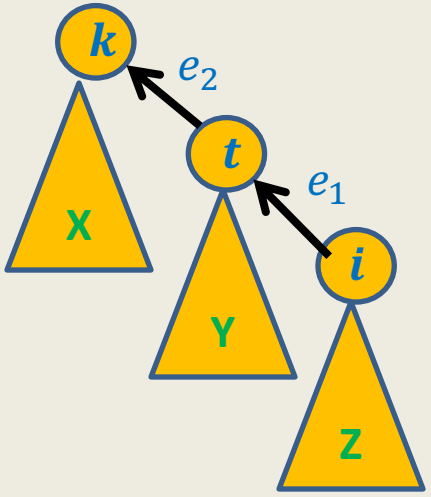
➔ After the edge $i \rightarrow t$ is inserted,

no. of element in $T(t) \geq 2n_i$

How to show that depth of any element = O(log n) ?

Let e_1, e_2, …, e_t be the edges on the path from i to the **root.**

Consider the moment just before edge e_2 is inserted.



no. of element in $T(t) \geq 2n_i$

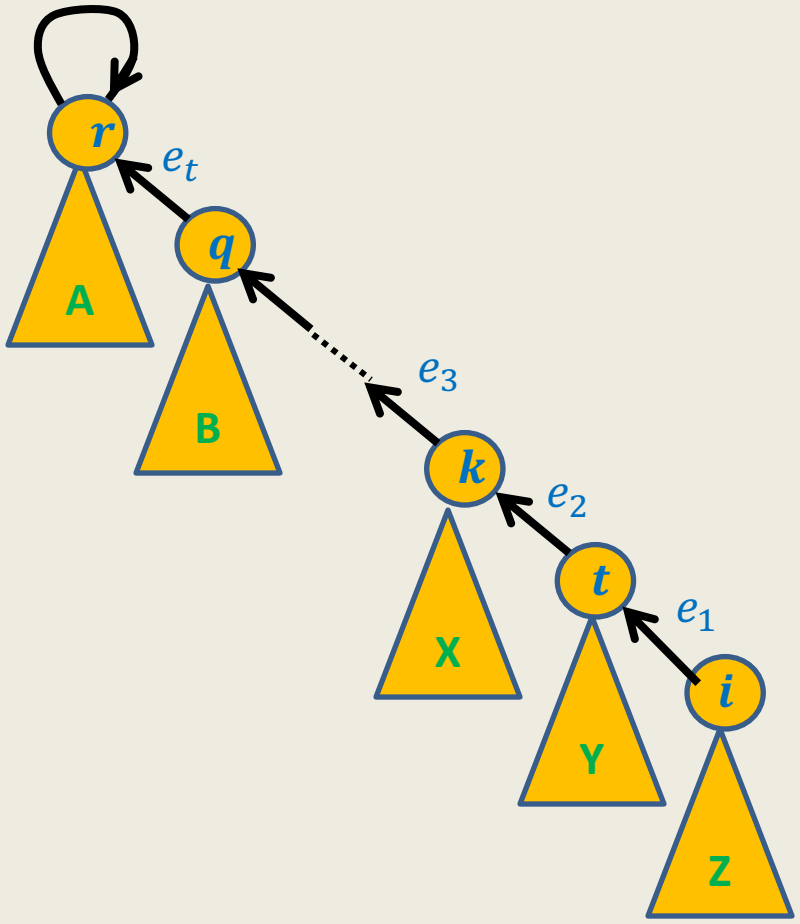
We added edge $t \rightarrow k$ (and **not** $k \rightarrow t$).

➔ # elements in $T(k) \geq 2n_i$.

➔ After the edge $t \rightarrow k$ is inserted,

no. of element in $T(k) \geq 4n_i$

How to show that depth of any element = O(log n) ?



Let e_1, e_2, …, e_t be the edges on the path from i to the **root.**

Arguing in a similar manner for edge e_3, …, e_t ➔

elements in $T(r)$ after insertion of $e_t \geq 2^t n_i$

Obviously $2^t n_i \leq n$

➔

Theorem: $t \leq \log_2 n$

Theorem: Given a collection of **n** singleton sets
followed by a sequence of **union** and **find** operations,
there is a data structure based that achieves **O**(log **n**) time per operation.

Question: Can we achieve even better bounds ?

Answer: Yes.

A new heuristic for better time complexity

Heuristic 2: Path compression

This is how this heuristic got invented

- The time complexity of a **Find(i)** operation is proportional to the depth of the node i in its rooted tree.
- If the elements are stored closer to the root, faster will the **Find()** be and hence faster will be the overall algorithm.

The algorithm for **Union** and **Find** was used in some application of **data-bases**.

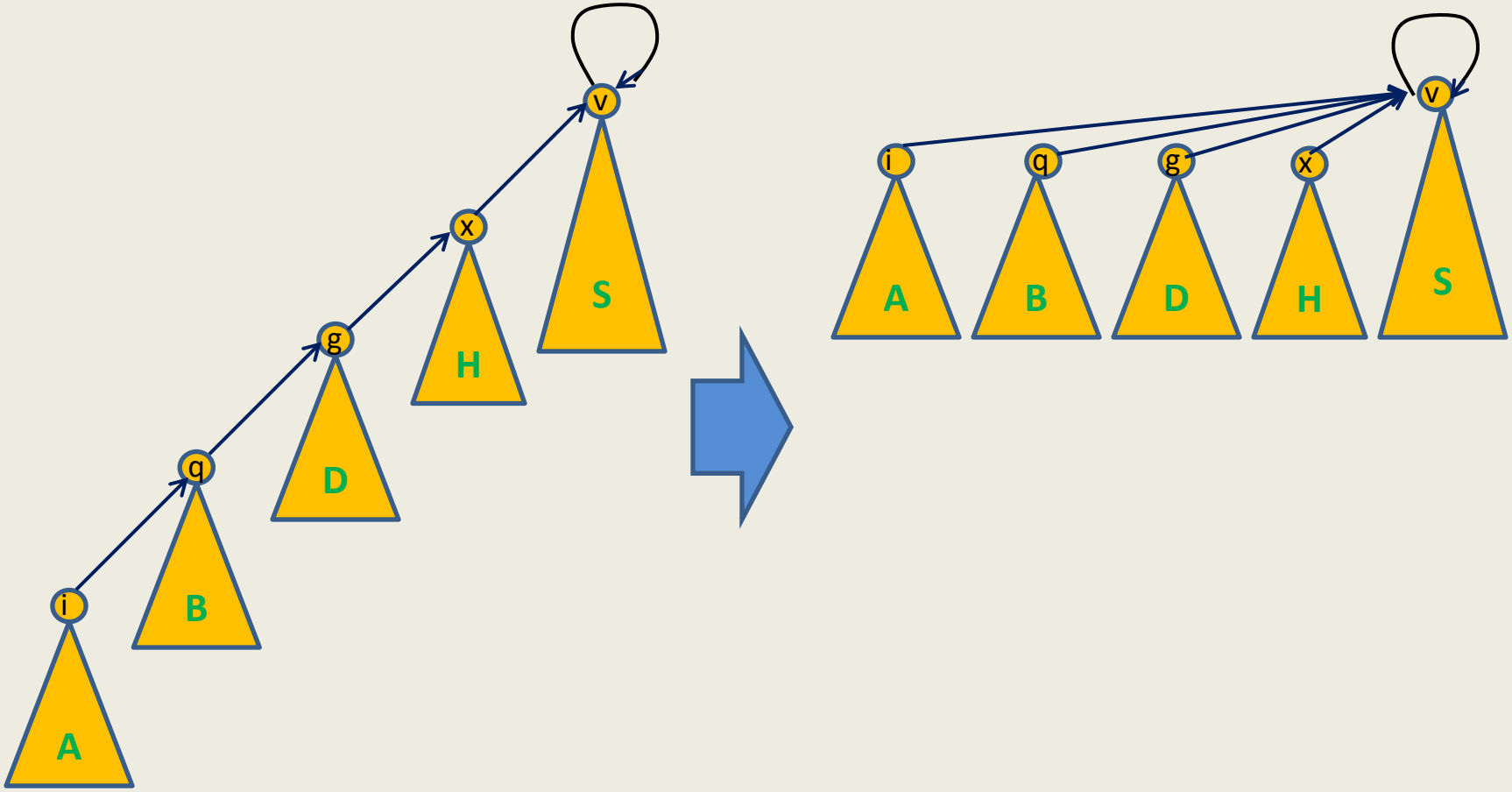
A clever programmer did the following modification to the code of **Find(i)**.

> While executing **Find(i)**, we traverse the path from node i to the root. Let $v_1, v_2, \ldots, v_t$, be the nodes traversed with v_t being the root node. At the end of **Find(i)**, if we update parent of each v_k, $1 \leq k < t$, to v_t , we achieve a reduction in depth of many nodes. This modification increases the time complexity of **Find(i)** by at most a constant factor. But this little modification increased the overall speed of the application very significantly.

The heuristic is called **path compression**. It is shown pictorially on the following slide.

It remained a mystery for many years to provide a theoretical explanation for its practical success.

Path compression during Find(i)



Pseudocode for the modified Find

```
Find(i)
    If (Parent(i) = i)    return i ;
    else
        j ← Find(Parent(i));
        Parent(i) ← j;
    return j
```

Data Structures and Algorithms

(ESO207)

Lecture 33

- Algorithm for ith order statistic of a set S.

Problem definition

Given a set S of n elements

compute ith **smallest** element from S.

Applications:

Trivial algorithm:

But sorting takes **O**(**n** log **n**) time and appears to be an overkill for this simple problem.

AIM: To design an algorithm with **O**(n) time complexity.

Assumption (For the sake of **neat description** and **analysis** of algorithms of this lecture):

- All elements of S are assumed to be **distinct**.

A motivational background

Though it was **intuitively appealing** to believe that there exists an **O**(n) time algorithm to compute ith smallest element, it remained a challenge for many years to design such an algorithm…

In **1972**, five well known researchers: **Blum, Floyd, Pratt, Rivest**, and **Tarjan** designed the **O**(n) time algorithm. It was designed during a **lunch break** of a conference when these five researchers sat together for the first time to solve the problem.

In this way, the problem which remained unsolved for many years got solved in less than an hour. But one should not ignore the efforts these researchers spent for years before arriving at the solution … It was their effort whose fruit got ripened in that hour ☺.

Notations

We shall now introduce some notations which will help in a neat description of the algorithm.

Notations

- S :

  the given set of n elements.

- e_i :

  ith **smallest** element of S.

- $S_{<x}$:

  subset of S consisting of all elements **smaller than** x.

- $S_{>x}$:

  subset of S consisting of all elements **greater than** x.

- **rank(S,x) :**

  1 + number of elements in S that are smaller than x.

- **Partition(S,x):**

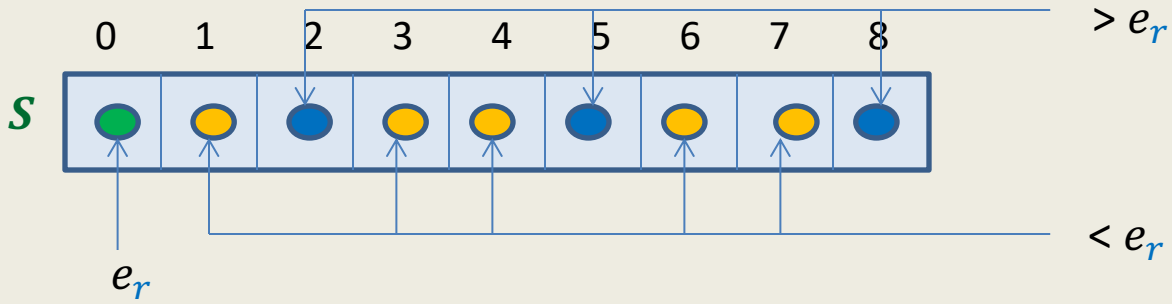
  algorithm to partition S into $S_{<x}$ and $S_{>x}$;

  this algorithm returns **($S_{<x}$, $S_{>x}$, r)** where r**=rank(S,x).**

Why should such an algorithm exist ?

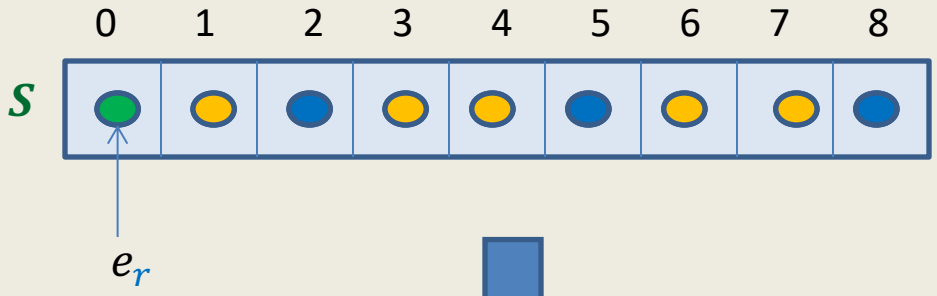
(inspiration from **QuickSort**)

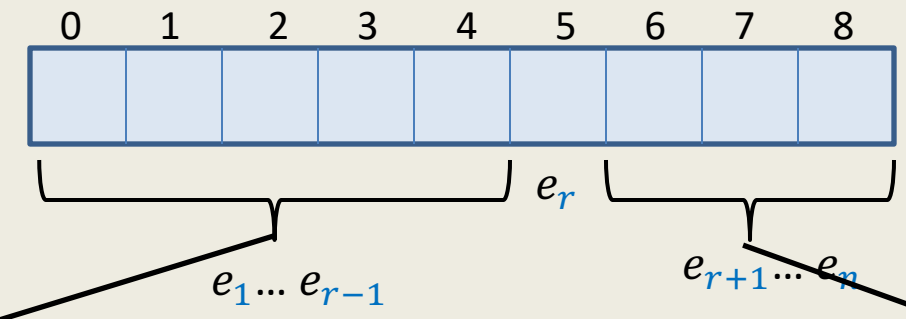
QuickSelect(S,i)



What happens during
Partition(S,0,8)

QuickSelect(S,i)





If $i > r$ we can discard these element.

If $i < r$ we can discard these elements.

Pseudocode for QuickSelect(S,i)

```
QuickSelect(S,i)
{       Pick an element x from S;
        (S_<x, S_>x, r)← Partition(S,x);
         If(i = r) return x;
         Else If (i< r)
                        QuickSelect(S_<x,i)
                Else
                        QuickSelect( S_>x,i − r );
}
```

Average case time complexity: O(n)

Worst case time complexity : O(n^2)

Analysis is simpler than **Quick Sort**.

Towards worst case O(n) time algorithm …

Key ideas

- **Inspiration** from some recurrences.

isn't is surprising that knowledge of recurrence can help in the design an efficient algorithm) ☺

- Concept of **approximate median**

This is the usual trick:
When a problem appears difficult, weaken the problem and try to solve it.

- Learning from **QuickSelect(S,i)**

This is also a natural choice.
Can we fine tune this algorithm to achieve our goal ?

Learning from recurrences

Question: what is the solution of recurrence **T**(n) = cn + **T**($9n/10$) ?

Answer: O(n)**.**

Sketch (by gradual unfolding):

T(n) = cn + c $9n/10$ + c $81\ n/100$ + …

= $cn[1 + 9/10 + 81/100 + \ldots]$

= **O**(n)

Lesson 1 :

Solution for **T**(n) = cn + **T**(an) is **O**(n) **if** $0 < a < 1$.

Learning from recurrences

Question: what is the solution of recurrence

$$T(n) = cn + T(n/6) + T(5n/7) \text{ ?}$$

Answer: O(n).

Sketch: (by induction)

Assertion: $T(n) \le c_1 n$.

Induction step: $T(n) = cn + T(n/6) + T(5n/7)$

$$\le cn + \frac{37}{42} c_1 n$$

$$\le c_1 n \quad \text{if } c_1 \ge \frac{42}{5} c$$

Lesson 2 :

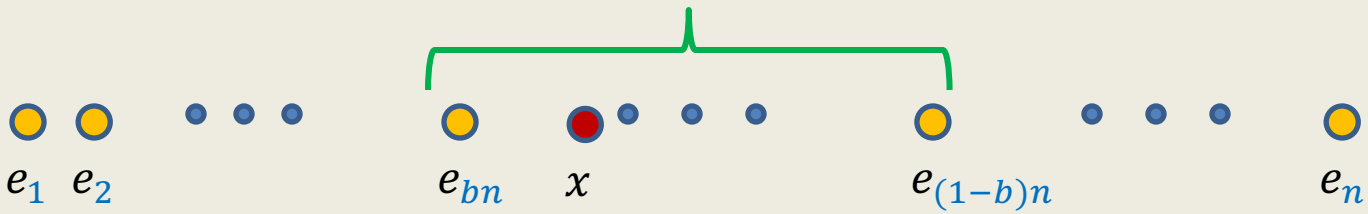
Solution for $T(n) = cn + T(an) + T(dn)$ is $O(n)$ if $a + d < 1$.

Concept of **approximate median**

Definition: Given a constant $0 < b \leq 1/2$,

an element $x \in S$

if **rank(x, S)**



Learning from QuickSelect(S,i)

QuickSelect(S,i)
{ Pick an element x from S;
($S_{<x}$, $S_{>x}$, r)← Partition(S,x);
If(i = r) return x;
Else If (i< r)
QuickSelect($S_{<x}$,i)
Else
QuickSelect($S_{>x}$,$i - r$);
}

O(n)

T($(1-b)n$)

Lesson 1

Answer: T(n) = cn + T($(1-b)n$)
= O(n)

What is time complexity of the algorithm

Algorithm 2

Select(S,i)

(A **linear time** algorithm)

Overview of the algorithm

Select(S,i)

{ Compute a b-approximate median, say x, of S; } **O(n)**

($S_{<x}$, $S_{>x}$, r)← **Partition(S,x);** } **O(n)**

If(i = r) return x;

Else If (i< r)

Select($S_{<x}$,i)

Else

Select($S_{>x}$,$i - r$); } **T($(1-b)n$)**

}

Observation: If we can compute b-approximate median in **O(n)** time, we get **O(n)** time algo.

But that appears too much to expect from us. Isn't it ?

So what to do ☹ ?

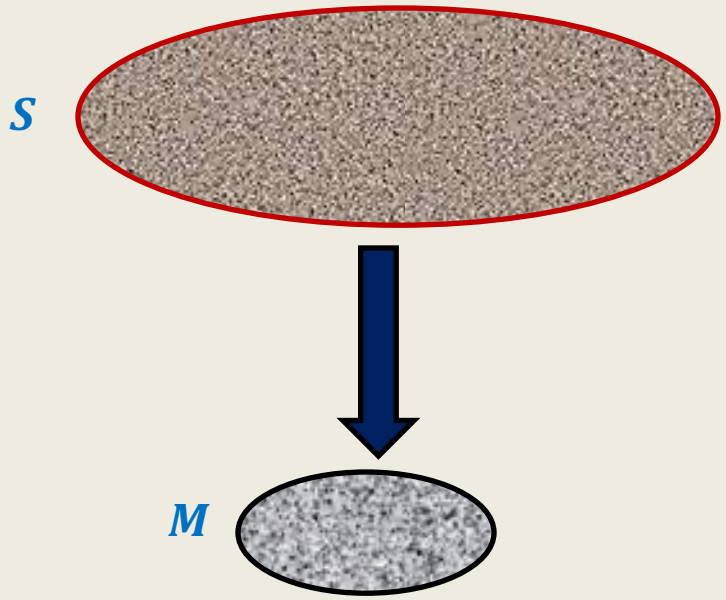
Hint: use **Lesson 2**

Observation: If we can compute b-approximate median

time complexity of the algorithm will still be **O(n).**

Spend some time on this observation to infer what it hints at.

AIM: How to compute a b-approximate median of S
in **O(**n**)+ T(**dn**)** time



Question: Can we form a set M of size dn such that
exact median of M is b-**approximate** median of S?

Forming the subset M with desired parameters

This step forms the core of the algorithm and is indeed a brilliant stroke of inspiration. The student is strongly recommended to ponder over this idea from various angles.

- Divide S into **groups** of **5** elements;
- Compute median of each group by sorting;
- Let M be the set of medians;

} **O(n)**

- Compute median of M, let it be x;

} **T($n/5$)**

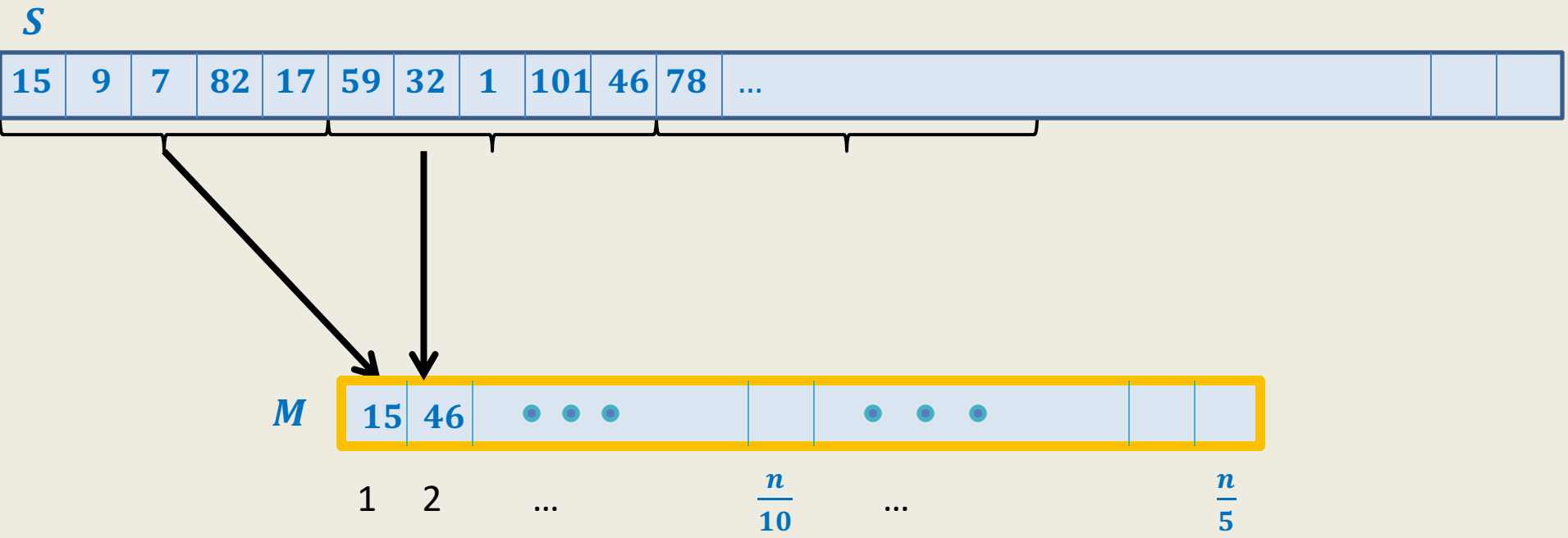
Question: Is x an approximate median of S ?

Answer: indeed.

The rank of x in M is $n/10$. Each element in M has two elements smaller than itself in its respective group. Hence there are at least $\frac{3n}{10} - 1$ elements in S which are **smaller** than x. In a similar way, there are at least $\frac{3n}{10} - 1$ elements in S which are **greater** than x. Hence, x is $\frac{3n}{10}$ -approximate median of S.

(See the animation on the following slide to get a better understanding of this explanation.)

Forming the subset M

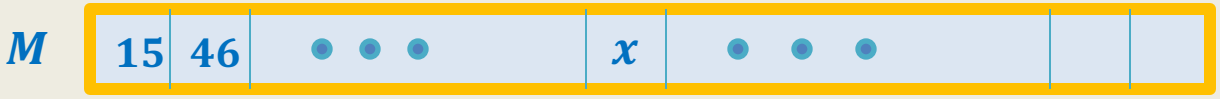


- Divide S into **groups** of **5** elements;
- Compute median of each group by sorting;

} O(n)

Forming the subset M

S



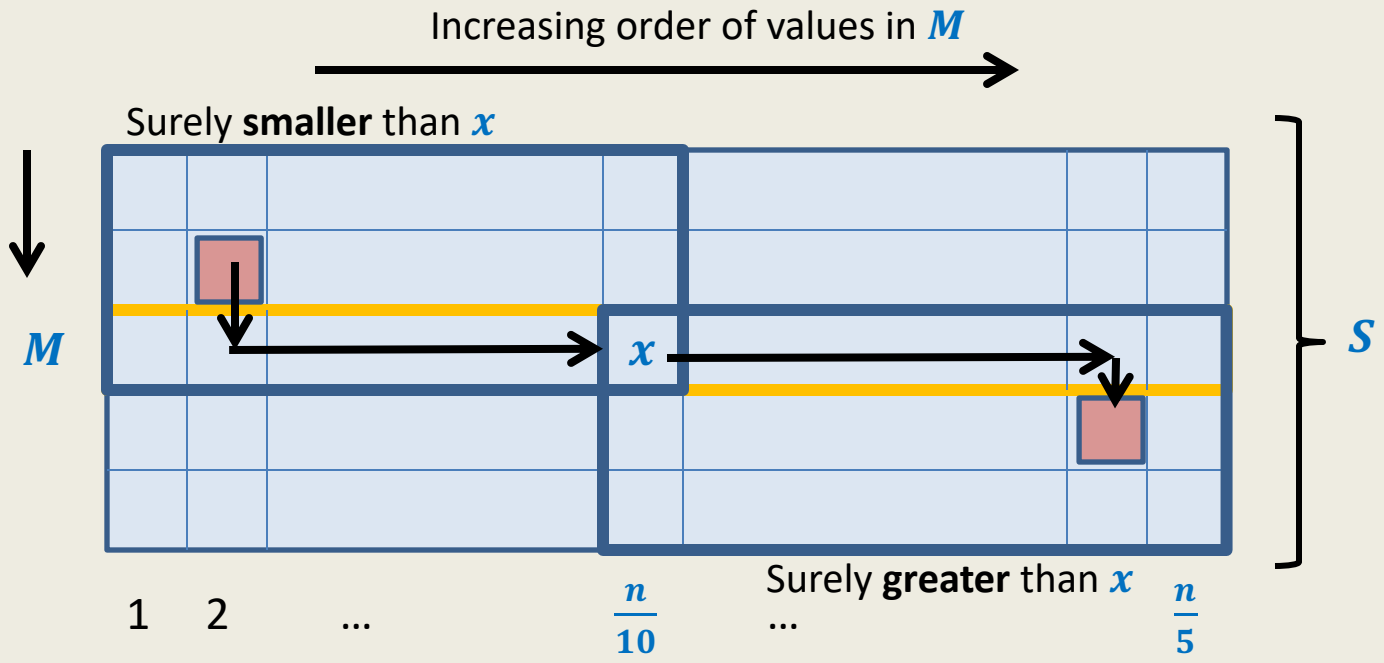
- Divide S into **groups** of **5** elements;
- Compute median of each group by sorting;
- Let M be the set of medians;
- Let x be median of M.

What can we say about rank of x in S ?

Spend some time to answer this question before moving ahead.

Forming the subset M

Bring back the remaining 4 elements associated with each element of M



➔ x is $\left(\frac{3n}{10}\right)$ −approximate median of S.

Time required to form M : **O**(n)

Pseudocode for Select(S,i)

```
Select(S,i)
    M ← ∅;
    Divide S into groups of 5 elements;
    Sort each group and add its median to M;
    x ← Select(M,|M|/2);
    (S_<x, S_>x, r)← Partition(S,x);
    If(i = r) return x;
    Else If (i< r)
            Select(S_<x,i)
        Else
            Select(S_>x,i − r);
```

O(n)

T(n/5)

O(n)

T(7n/10)

Analysis

T(n) = cn + T(n/5) + T(7n/10)

= O(n) **[Learning from Recurrence of type II]**

Theorem: Given any S of **n** elements, we can compute ith smallest element from S in O(**n**) worst case time.

Exercises

(Attempting these exercises will give you a better insight into the algorithm.)

- What is magical about number 5 in the algorithm ?
- What if we divide the set S into groups of size 3 ?
- What if we divide the set S into groups of size 7 ?
- What if we divide the set S into groups of even size (e.g. 4 or 6) ?

Data Structures and Algorithms

(ESO207)

Lecture 34

- A new algorithm design paradigm: Greedy strategy

  part I

Path to the solution of a problem

No formula

But there is a **systematic approach** which usually works ☺

Today's lecture will demonstrate this approach ☺

Problem : JOB Scheduling

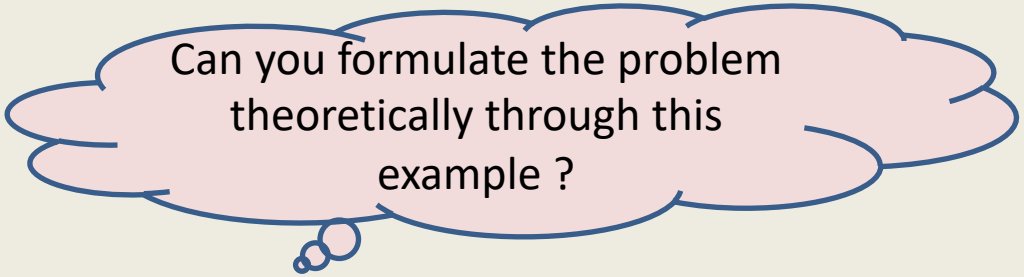
Largest subset of non-overlapping job

A motivating example

Antaragni 2021

- There are n large-scale activities to be performed in Auditorium.
- Each large scale activity has a **start time** and **finish time**.
- There is **overlap** among various activities.

Aim: What is the **largest subset** of activities that can be performed ?



Formal Description

A job scheduling problem

INPUT:

- A set J of n jobs $\{j_1, j_2, \ldots, j_n\}$
- job j_i is specified by two real numbers

  s(i): start time of job j_i

  f(i): finish time of job j_i
- A single server

Constraints:

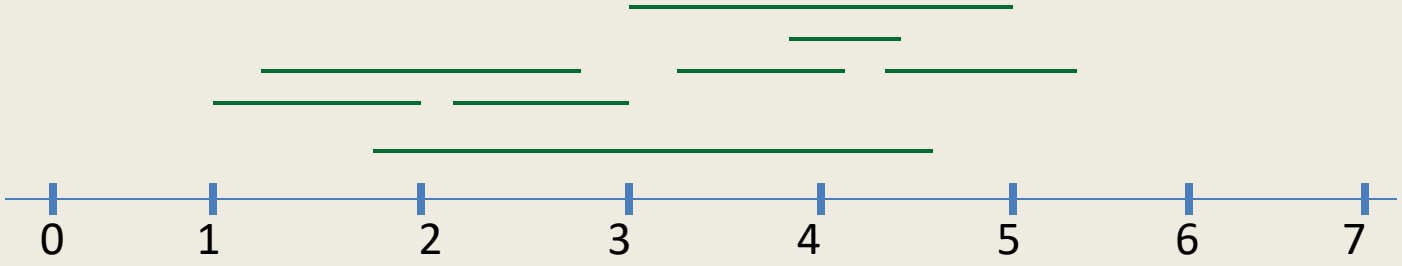
- Server can execute at most one job at any moment of time.
- **Job j_i**, if scheduled,

Aim:

To select the **largest** subset of **non-overlapping** jobs which can be executed by the server.

Example

INPUT: $(1, 2), (1.2, 2.8), (1.8, 4.6), (2.1, 3), (3, 5), (3.3, 4.2), (3.9, 4.4), (4.3, 5.4)$



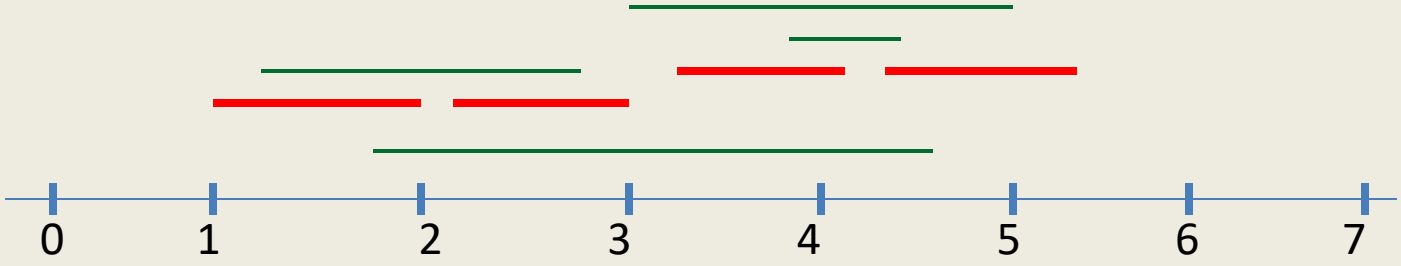
It makes sense to work with pictures than these numbers☺

job i is said to be **non-overlapping** with job k if

Try to find solution for the above example.

Example

INPUT: $(1, 2), (1.2, 2.8), (1.8, 4.6), (2.1, 3), (3, 5), (3.3, 4.2), (3.9, 4.4), (4.3, 5.4)$

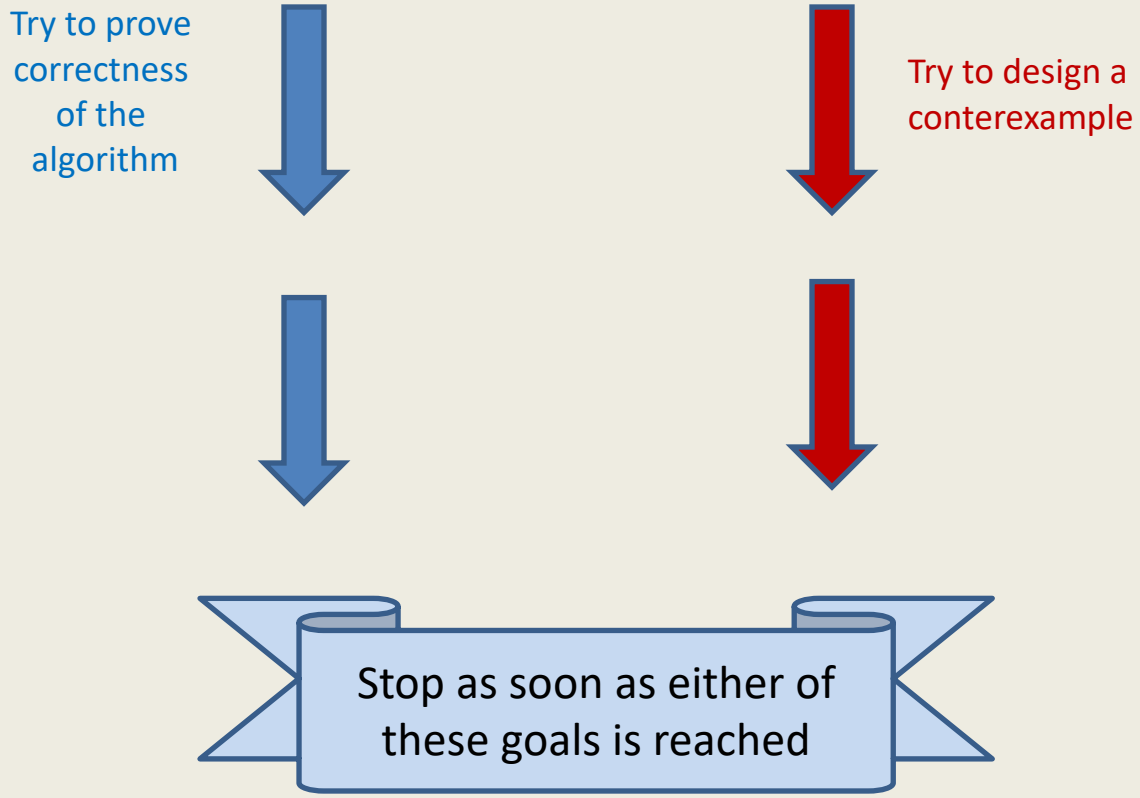


job i is said to be **non-overlappi** $[s(k), f(k)] = \emptyset$

What strategy come to your mind?

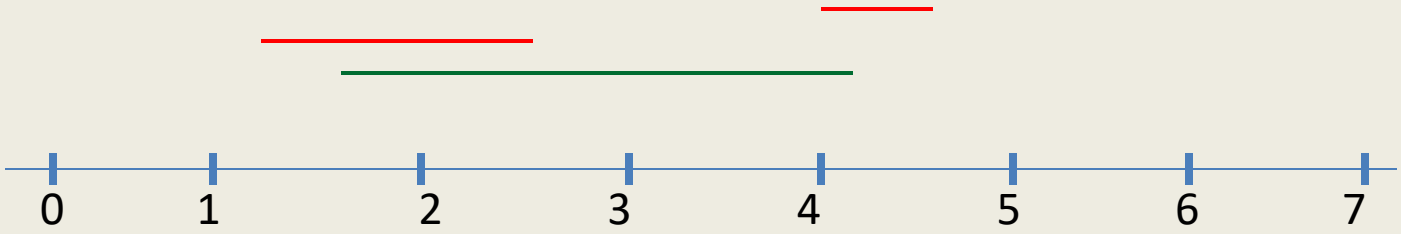
Designing algorithm for any problem

1. Choose a strategy based on some intuition
2. Transform the strategy into an algorithm.



Designing algorithm for the problem

Strategy 1: Select the earliest start time job



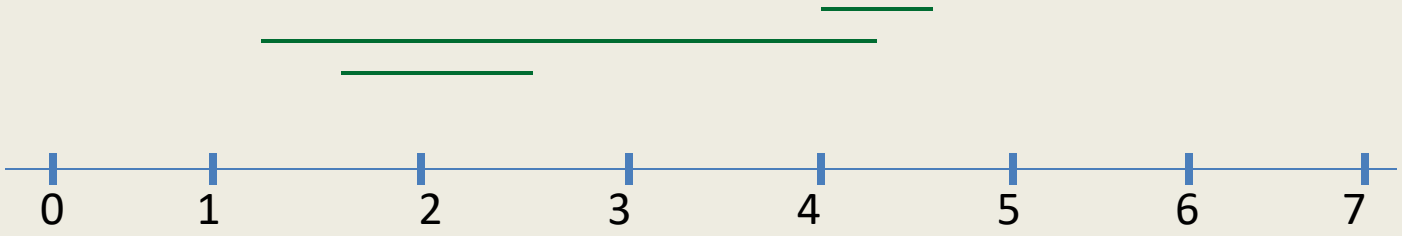
When one fails to prove the correctness of this strategy, one should search for a counterexample. Can you transform the above example into a counterexample ?

Intuition:

It might be better to assign jobs as early as possible so as **to make optimum use of server**.

Designing algorithm for the problem

Strategy 1: Select the earliest start time job

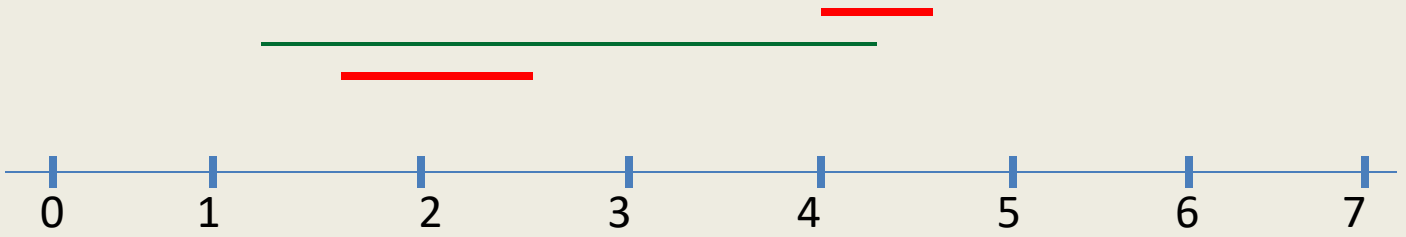


Intuition:

It might be better to assign jobs as early as possible so as **to make optimum use of server.**

Designing algorithm for the problem

Strategy 1: Select the earliest start time job



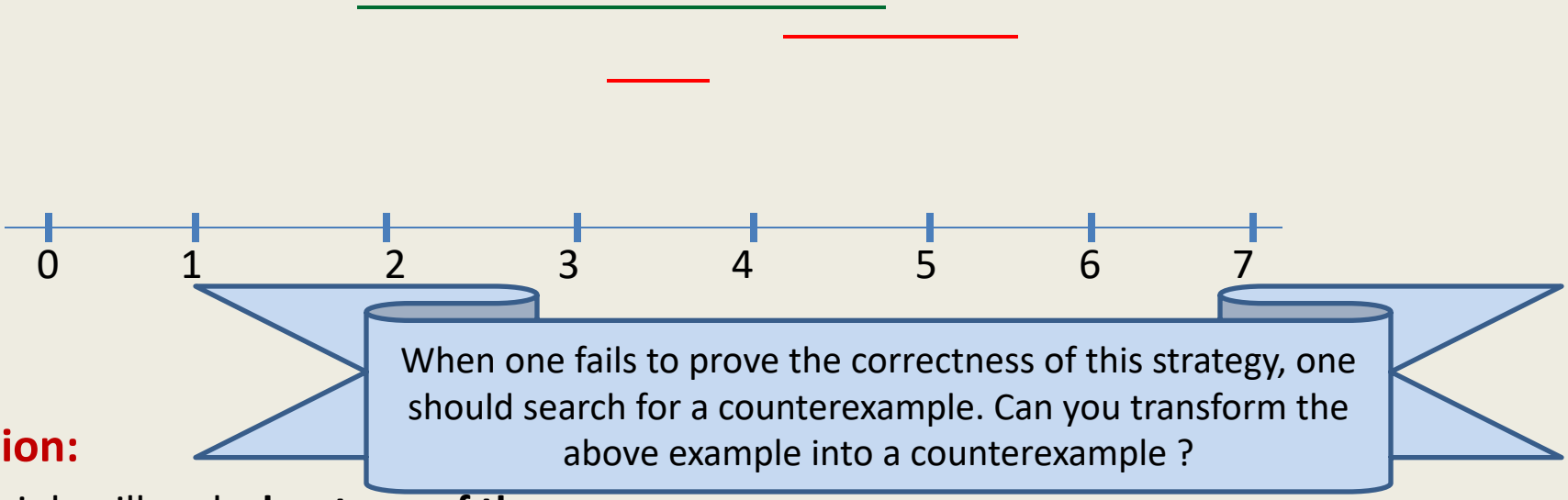
Instead of getting disappointed, try to realize that this counterexample points towards some other strategy which might work.

Intuition:

It might be better to assign jobs as early as possible so as to make optimum use of server.

Designing algorithm for the problem

Strategy 2: Select the job with smallest duration



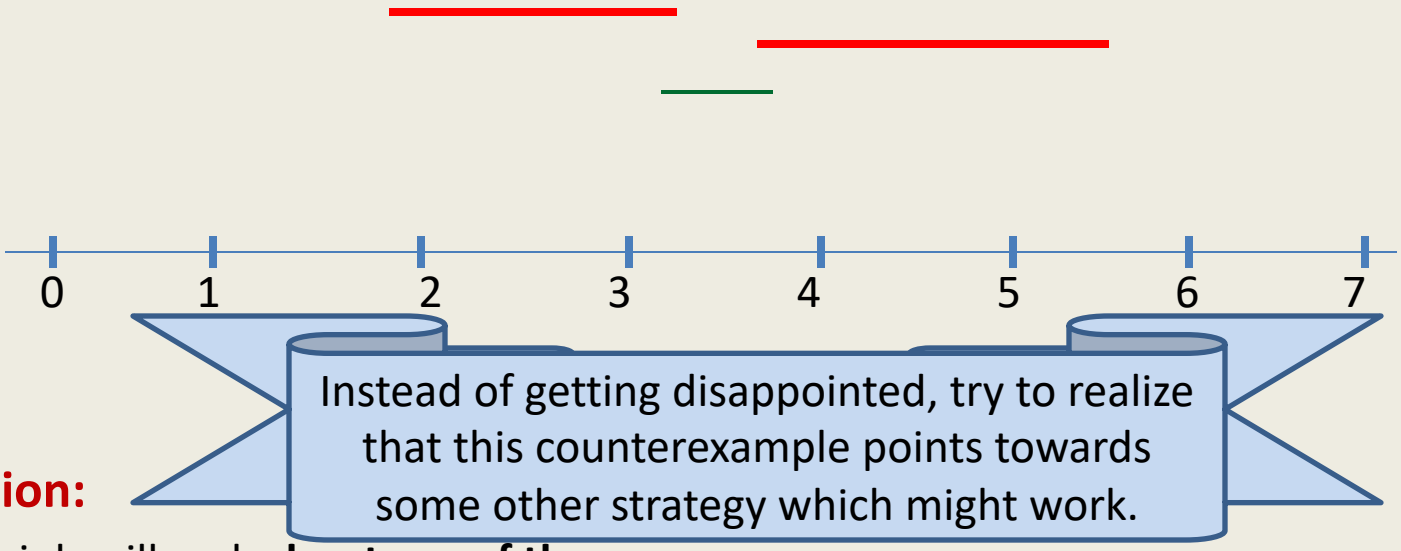
Intuition:

Such a job will make **least use of the server**

➔ this might lead to larger number of jobs to be executed

Designing algorithm for the problem

Strategy 2: Select the job with smallest duration



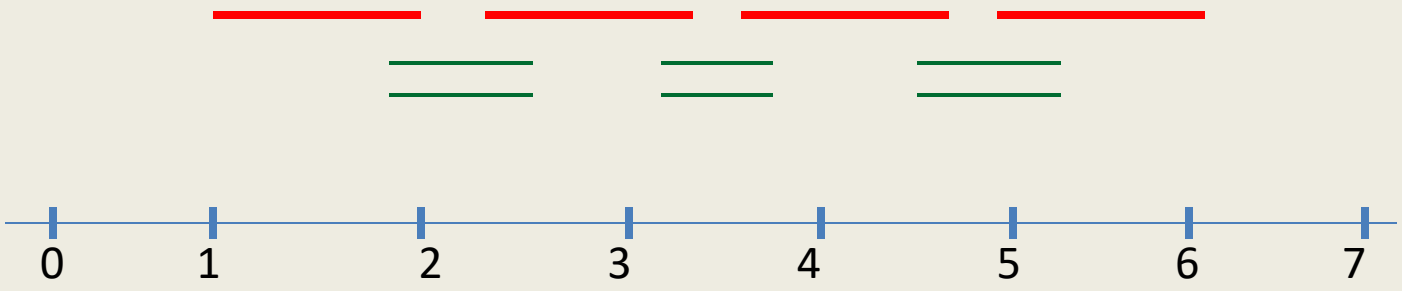
Intuition:

Such a job will make **least use of the server**

➔ this might lead to larger number of jobs to be executed

Designing algorithm for the problem

Strategy 3: Select the job with smallest no. of overlaps



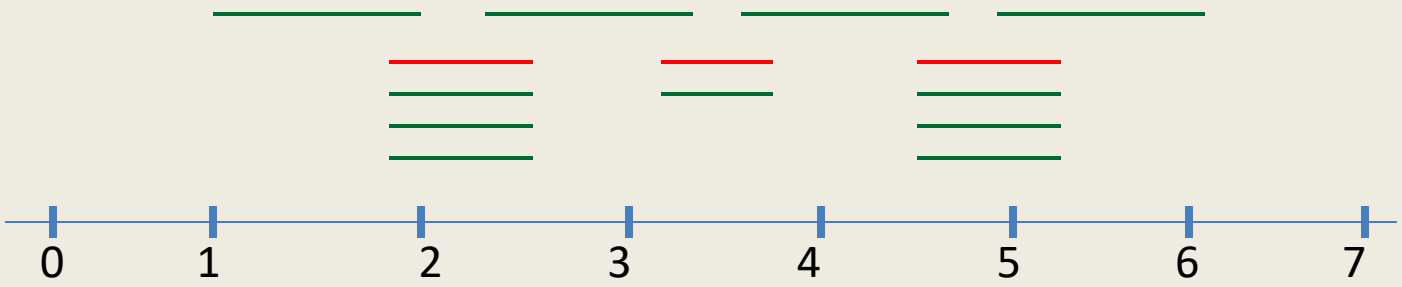
When one fails to prove the correctness of this strategy, one should search for a counterexample. Can you transform the above example into a counterexample ?

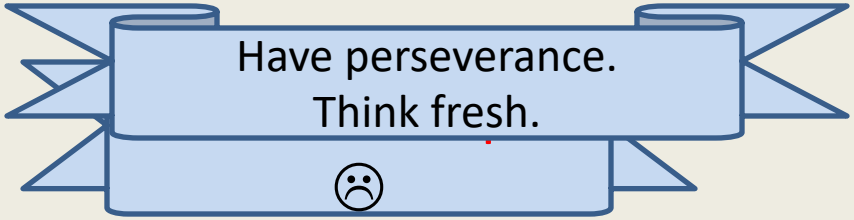
Intuition:

Selecting such a job will result in **least number of other jobs to be discarded**.

Designing algorithm for the problem

Strategy 3: Select the job with smallest no. of overlaps



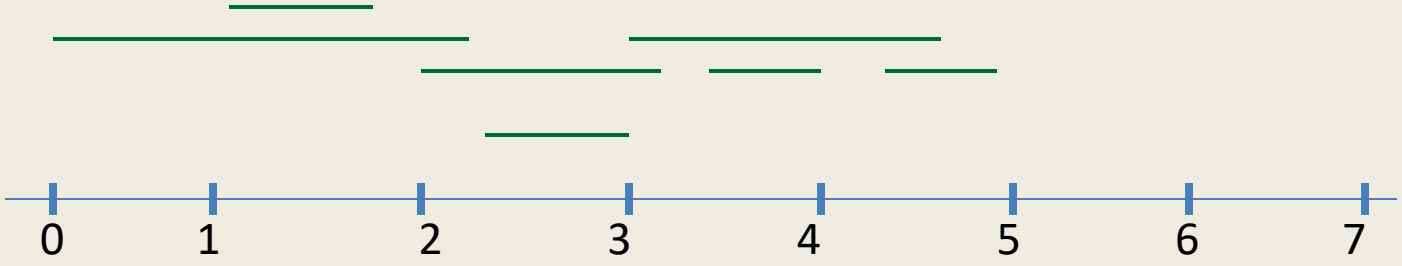


Intuition:

Selecting such a job will result in **least number of other jobs to be discarded.**

Designing algorithm for the problem

Strategy 4: Select the job with earliest finish time



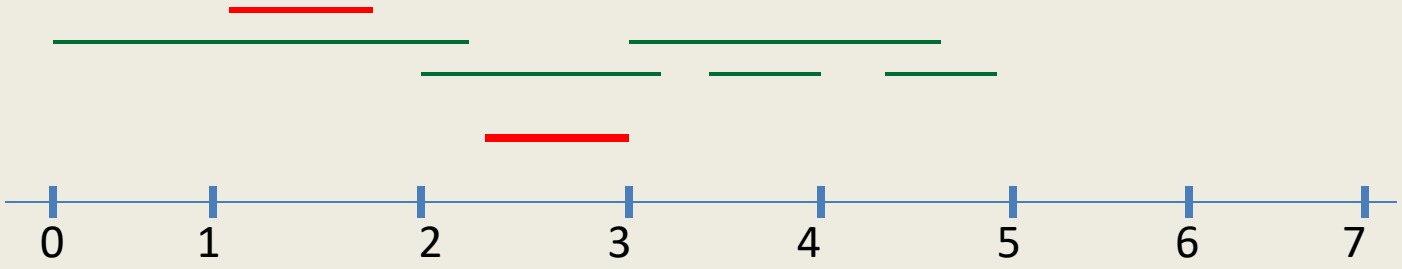
Intuition:

Selecting such a job will **free** the server **earliest**

➔ hence more no. of jobs might get scheduled.

Designing algorithm for the problem

Strategy 4: Select the job with earliest finish time



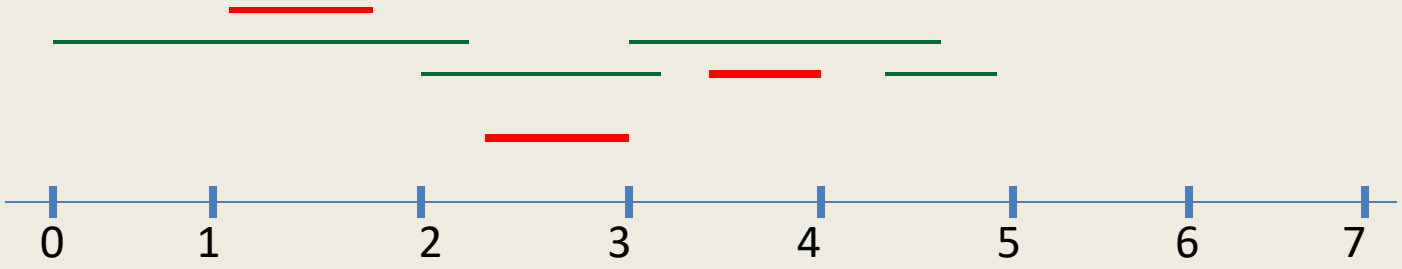
Intuition:

Selecting such a job will **free** the server **earliest**

➔ hence more no. of jobs might get scheduled.

Designing algorithm for the problem

Strategy 4: Select the job with earliest finish time



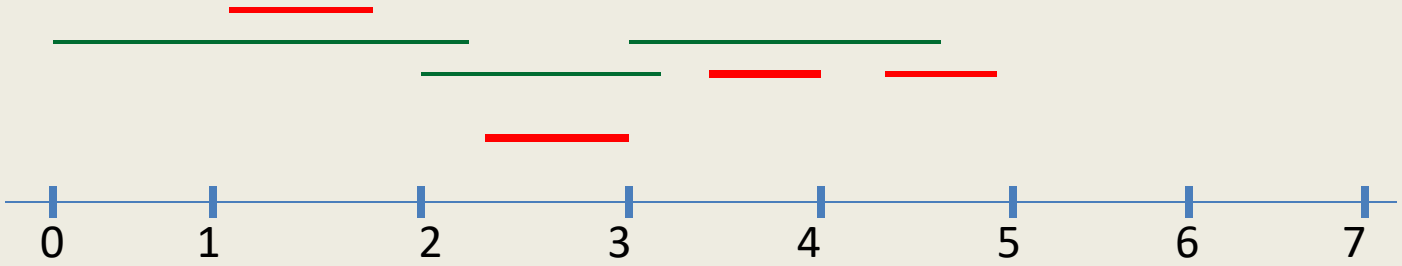
Intuition:

Selecting such a job will **free** the server **earliest**

➔ hence more no. of jobs might get scheduled.

Designing algorithm for the problem

Strategy 4: Select the job with earliest finish time



It is indeed a correct solution for this example. We should try to prove the correctness of the algorithm.☺But first we give a full description of the algorithm based on this strategy.

Intuition:

Selecting such a job will **free** the server **earliest**

➔ hence more no. of jobs might get scheduled.

Algorithm "earliest finish time"

Description

Algorithm (**Input :** set J of n jobs.)

1. Define $A \leftarrow \emptyset$;
2. **While** $J <> \emptyset$ **do**

   { Let x be the job from J with earliest finish time;

   $A \leftarrow A \cup \{x\}$;

   Remove x and all jobs that overlap with x from set J;

   }
3. Return A;

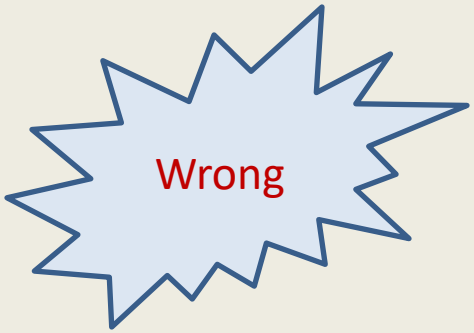
Running time for a trivial implementation of the above algorithm: O(n^2)

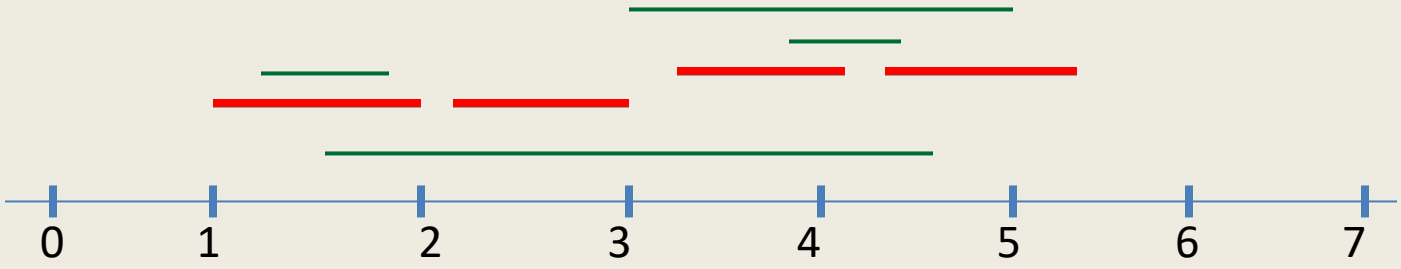
Algorithm "earliest finish time"

Correctness

Let x be the job with earliest finish time.

Lemma1: x is present in the optimal solution for J.





Too strong a claim !

Algorithm "earliest finish time"

Correctness

Let x be the job with earliest finish time.

Lemma1: There exists an optimal solution for J in which x is present.

Proof: Consider any optimal solution O for J.

Let y be the job from O with earliest finish time.

Let $O' \leftarrow O \setminus \{y\}$

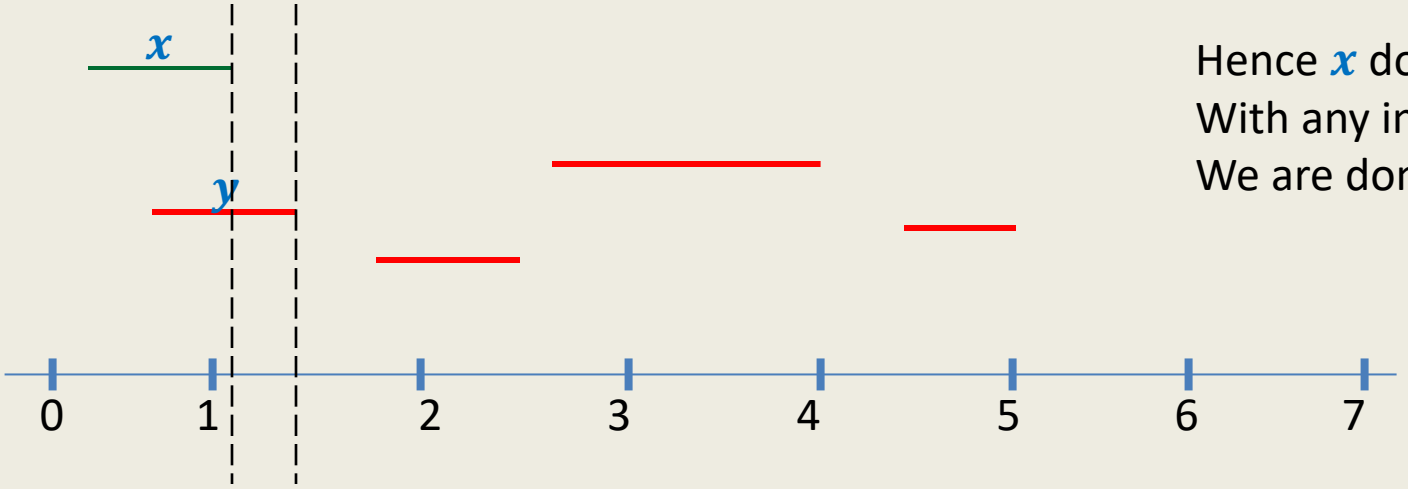
$\rightarrow f(y) < s(z)\ \forall\ z \in O'$

$\rightarrow f(x) \leq f(y)$

$\rightarrow f(x) < s(z)\ \forall\ z \in O'$

$O' \cup \{x\}$ is also an optimal solution.

Reason: $O' \cup \{x\}$ has no overlapping intervals. Give arguments.



Hence x does not overlap
With any interval of O'.
We are done ☺

Homework

Spend 30 minutes today on the following problems.

1. Use **Lemma1** to complete the proof of correctness of the algorithm.

2. Design an **O**($n$log n) implementation of the algorithm.

Data Structures and Algorithms

(ESO207)

Lecture 35

- A new algorithm design paradigm: Greedy strategy

  part II

Continuing Problem from last class

JOB Scheduling

Largest subset of non-overlapping job

A job scheduling problem

Formal Description

INPUT:

- A set J of n jobs $\{j_1, j_2, \ldots, j_n\}$
- job j_i is specified by two real numbers

  s(i): start time of job j_i

  f(i): finish time of job j_i
- A single server

Constraints:

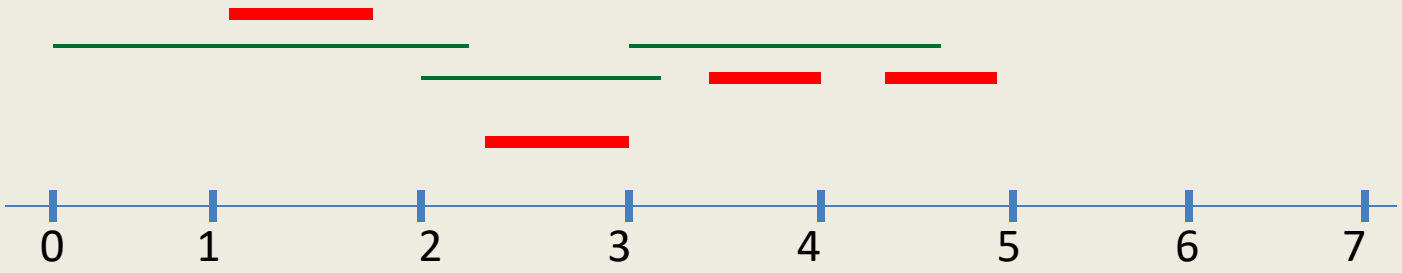
- Server can execute at most one job at any moment of time and a job.
- **Job** j_i, if scheduled, has to be scheduled during[**s**(i), **f**(i)] only.

Aim:

To select the **largest** subset

Designing algorithm for the problem

Strategy 4: Select the job with earliest finish time



Intuition:

Selecting such a job will **free** the server **earliest**

➔ hence more no. of jobs might get scheduled.

Algorithm "earliest finish time"

Algorithm (**Input :** set J of n jobs.)

1. Define $A \leftarrow \emptyset$;
2. **While** $J <> \emptyset$ **do**

   { Let $x \in J$ has earliest finish time;

   $A \leftarrow A \cup \{x\}$;

   $J \leftarrow J \backslash \textbf{Overlap}(x)$;

   }
3. Return A;

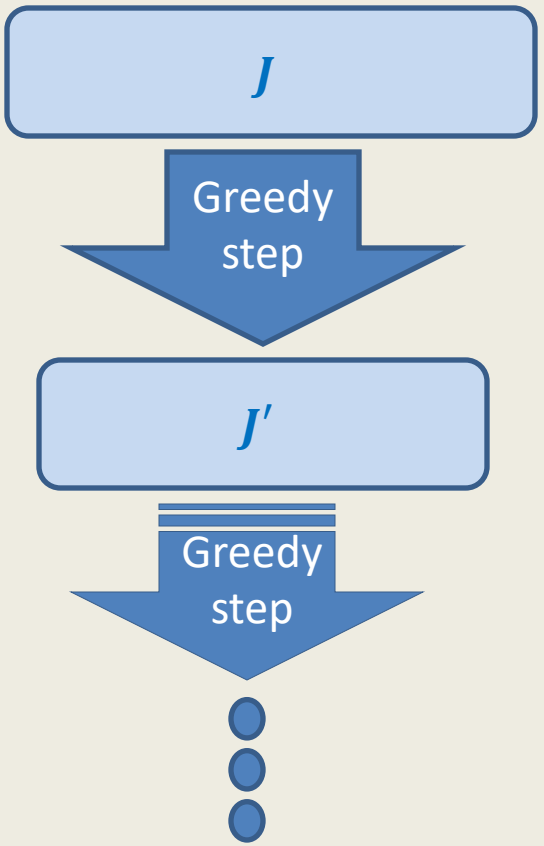
Lemma1 (last class):

There exists an optimal solution for J containing the **earliest finish time** job.

Proof of correctness ?

Let $x \in J$ be the job with earliest finish time.

Let $J' = J \backslash \textbf{Overlap}(x)$



Algorithm "earliest finish time"

How to prove it ?

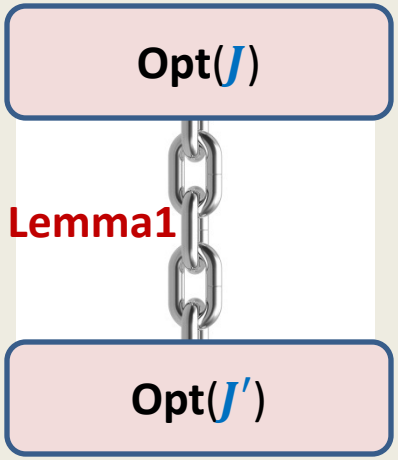
proof correct

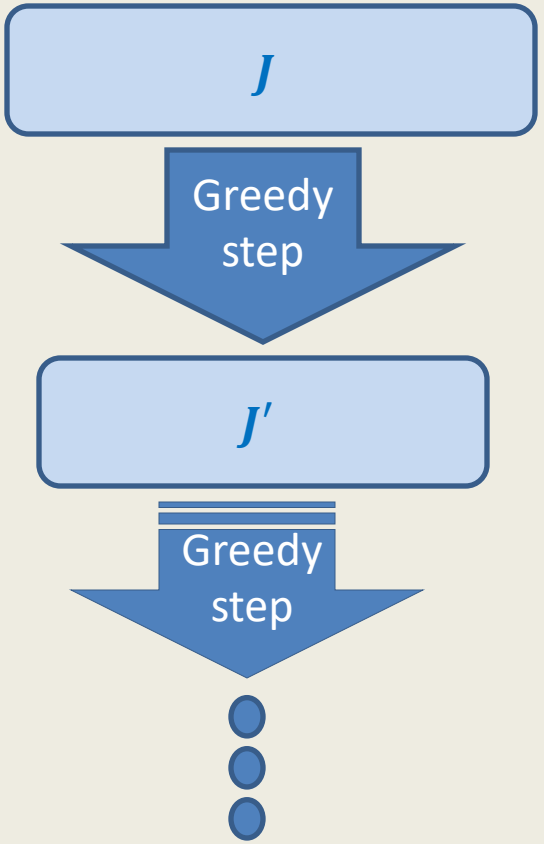
Proof of correctness ?

Let $x \in J$ be the job with earliest finish time.

Let $J' = J \backslash \textbf{Overlap}(x)$

Opt(J)= Opt(J') + 1





Notation:

Opt(J):

Theorem: $\mathbf{Opt}(J) = \mathbf{Opt}(J') + 1$.

- Proof has two parts

  $\mathbf{Opt}(J) \geq \mathbf{Opt}(J') + 1$

  $\mathbf{Opt}(J') \geq \mathbf{Opt}(J) - 1$

  Try to give a physical interpretation to these inequalities.

- Proof for each part is a proof by **construction**

Algorithm "earliest finish time"

Proving $\mathbf{Opt}(J) \geq \mathbf{Opt}(J') + 1$

Observation: start time of every job in J' is greater than finish time of x.

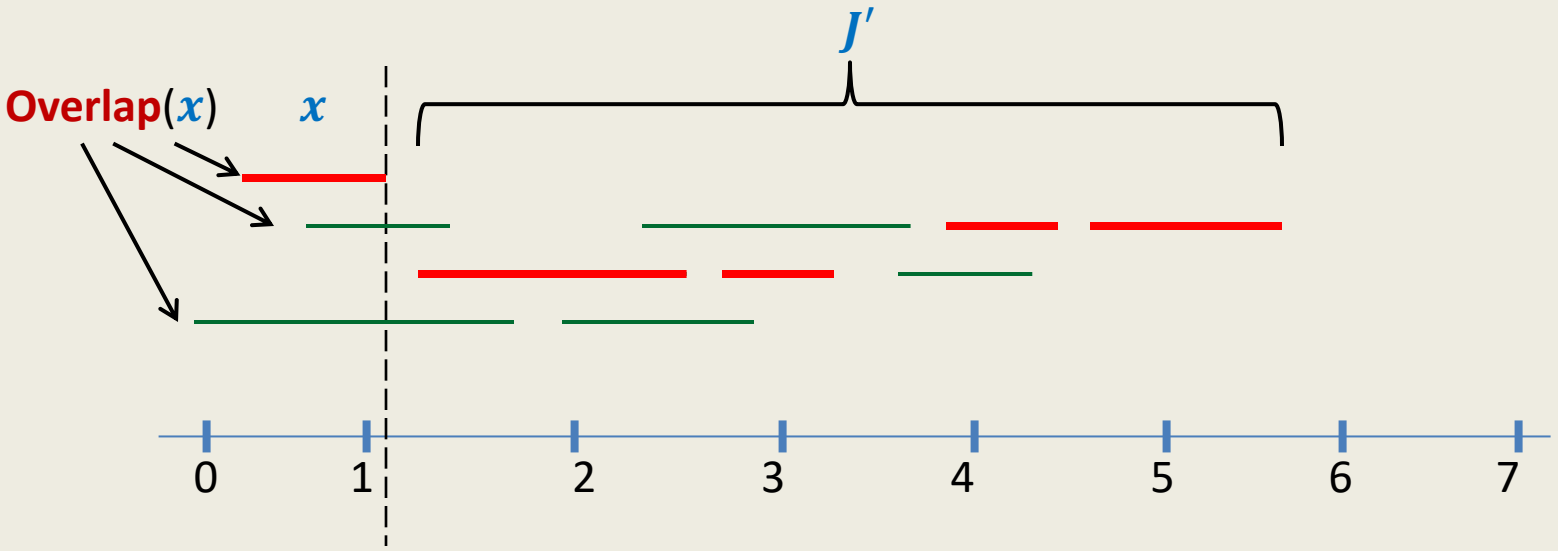
Let O' be any optimal solution for J'.

None of the jobs in

Hence O' U$\{x\}$ is a

Therefore **Opt**$(J) \geq |O'| + 1$

From an **optimal solution** of J'
can you derive a **solution** for J with one extra job?



Algorithm "earliest finish time"

Lemma1 (last class): There exists an optimal solution for J in which x is present.

Let O be an optimal solution for J containing x.

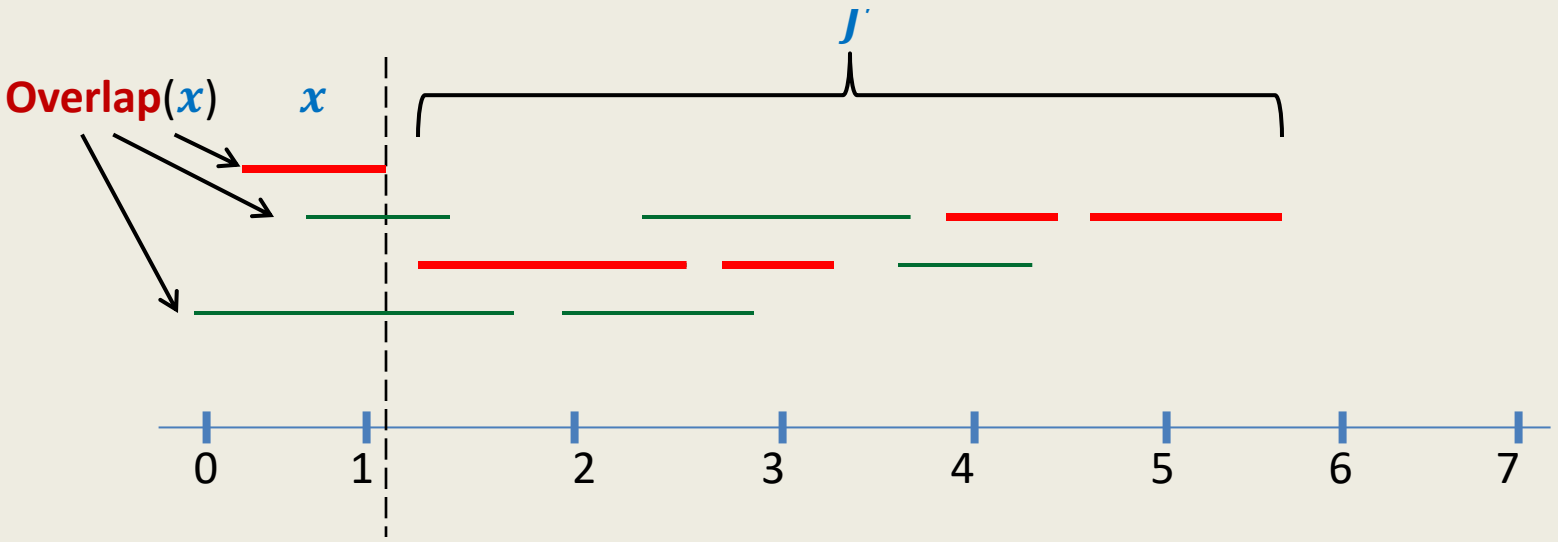
None of the jobs in O overlaps with x.

➔ Every job from O

Hence $O \setminus \{x\}$ is a subset of non-overlapping jobs from J'.

Therefore **Opt**(J') ≥ $|O|$ - 1

From an **optimal solution** of J
can you derive a **solution** for J' with one job less?



Theorem:

Given any set J of n jobs,

the algorithm based on "**earliest finish time**" approach

computes the largest subset of non-overlapping job.

O(n log n) implementation of the Algorithm

This is not the only way to achieve **O(n log n)** time. It can be done other ways as well.

Algorithm (**Input :** set J of n jobs.)

1. Define $A \leftarrow \emptyset$;
2. **While** $J <> \emptyset$ **do**
   {
   Let $x \in J$ have earliest finish time;
   $A \leftarrow A \cup \{x\}$;
   $J \leftarrow J \backslash$ **Overlap**(x);
   }
3. Return A;

Maintain a **binary min-heap** for J based on **finish time** as the key.

Sort J in increasing order of **start time**.

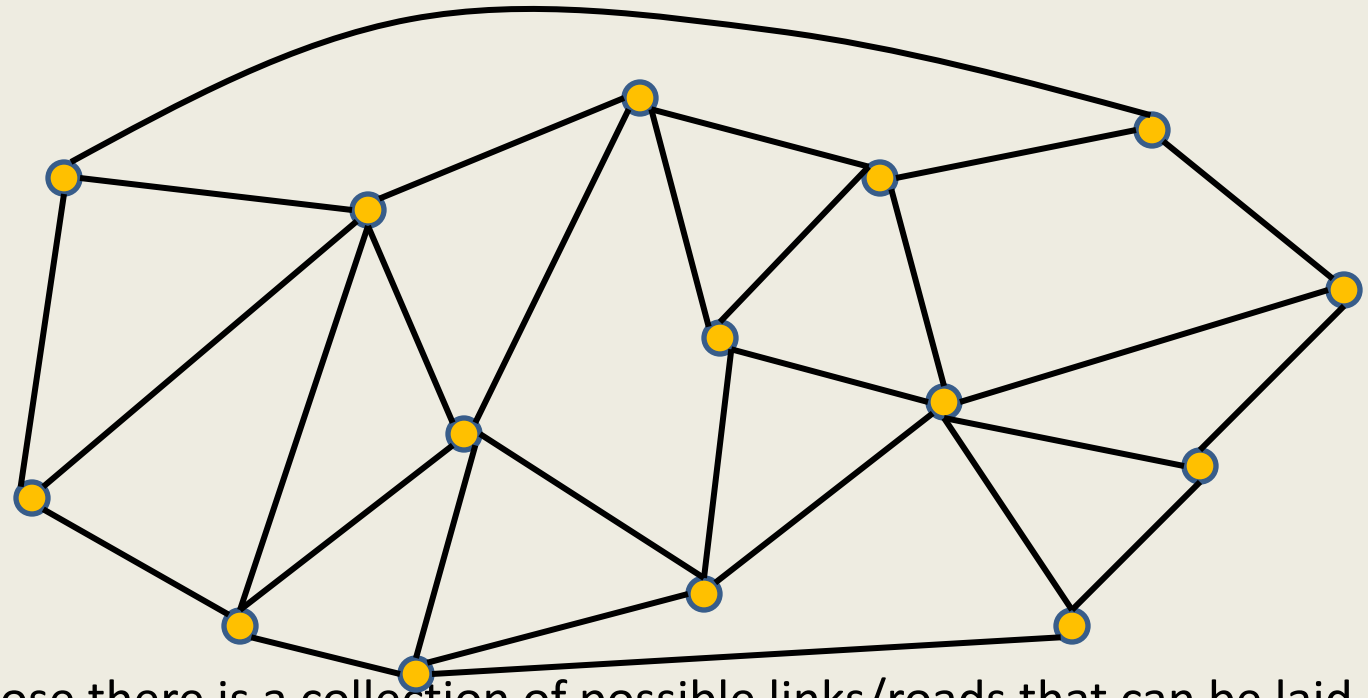
➔ **O(n^2) time complexity is obvious**

Problem 2

First we shall give motivation.

Motivation:

A road or telecommunication network



Suppose there is a collection of possible links/roads that can be laid.

But laying down each possible link/road is costly.

Aim: To lay down **least number** of links/roads to ensure **connectivity** between each pair of nodes/cities.

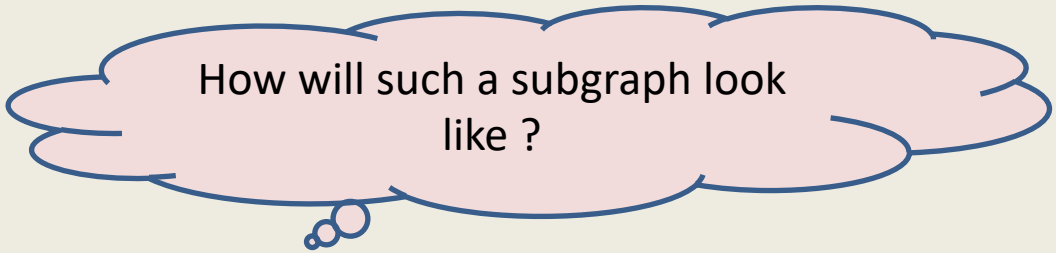
Motivation

Formal description of the problem

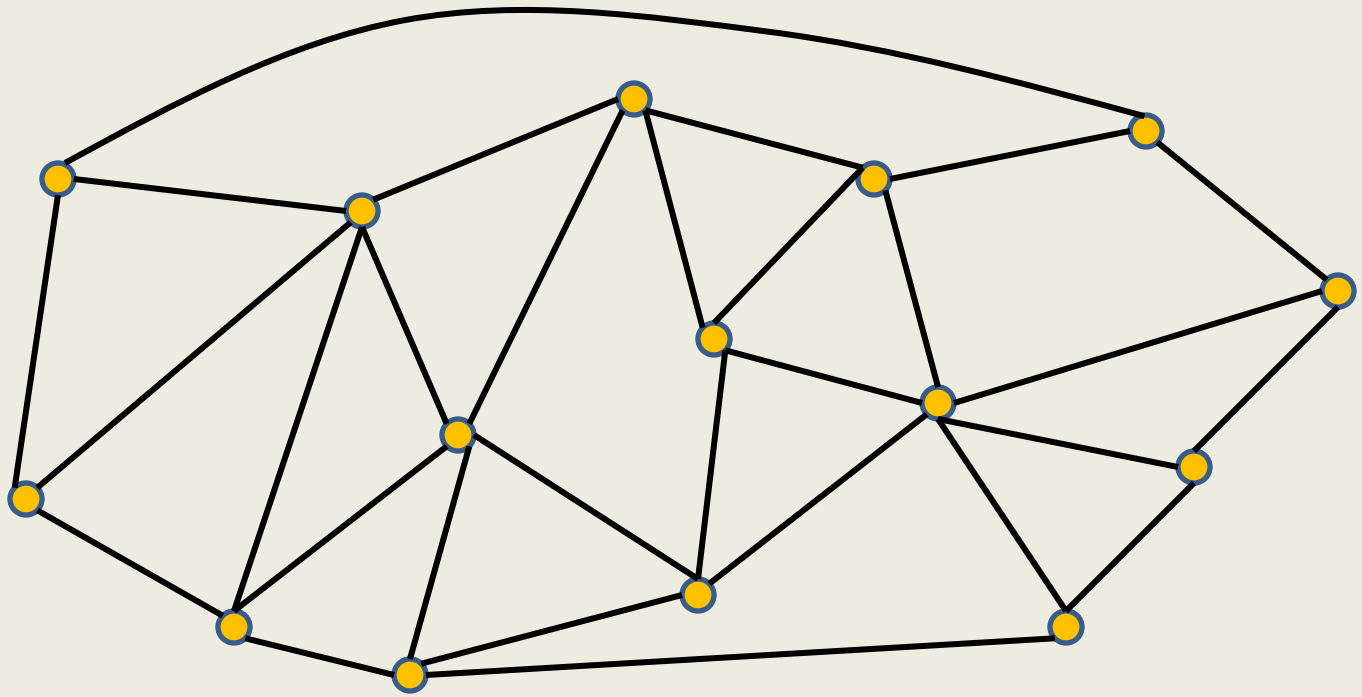
Input: an undirected graph **G**=(**V**,**E**).

Aim: compute a **subgraph** (**V**,**E'**), **E' ⊆ E** such that

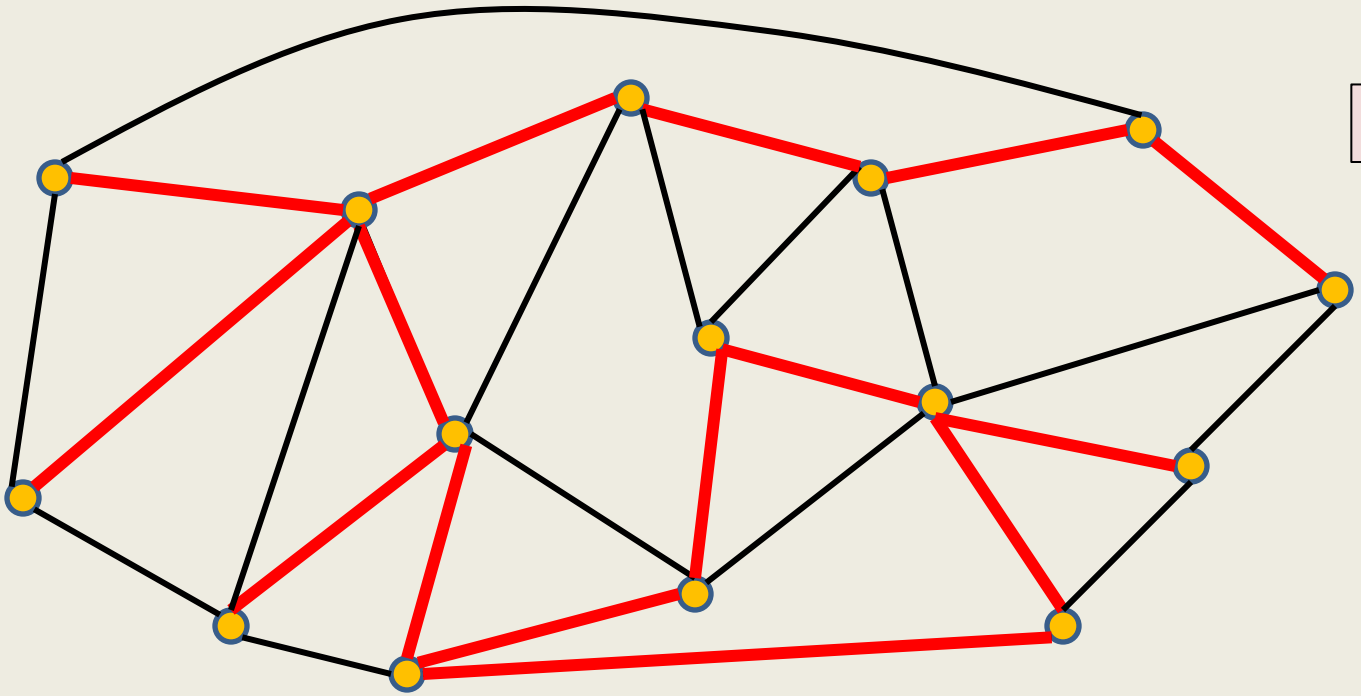
- **Connectivity** among all **V** is guaranteed in the **subgraph**.
- **|E'|** is **minimum.**



A road or telecommunication network



A road or telecommunication network



No | Yes | No

Is this subgraph meeting our requirement ?

A tree

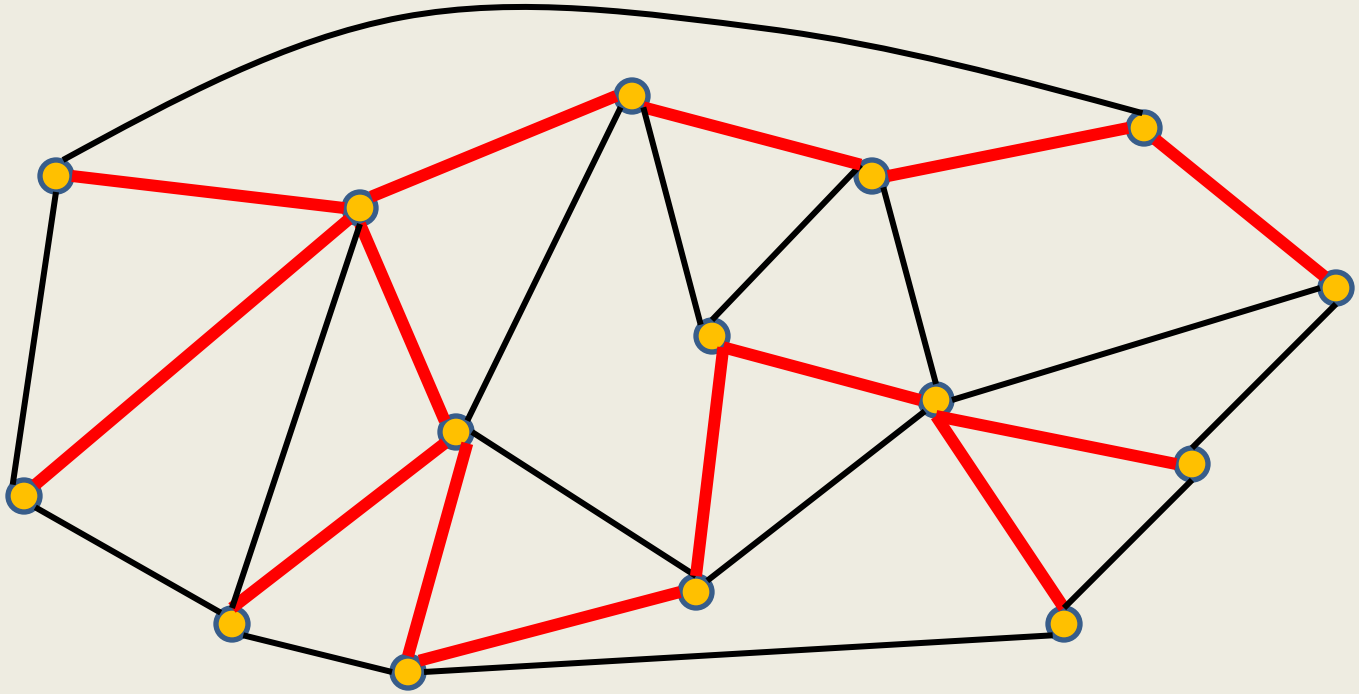
The following definitions are **equivalent**.

- An undirected graph which is **connected**
- An undirected graph where each pair of vertices has
- An undirected **connected** graph on n vertices and
- An undirected graph on n vertices and $n-1$ **edges** and

A Spanning tree

Definition: For an undirected graph (V,E),

A spanning tree is a **subgraph** (V,E'), E' ⊆ E which is a tree.

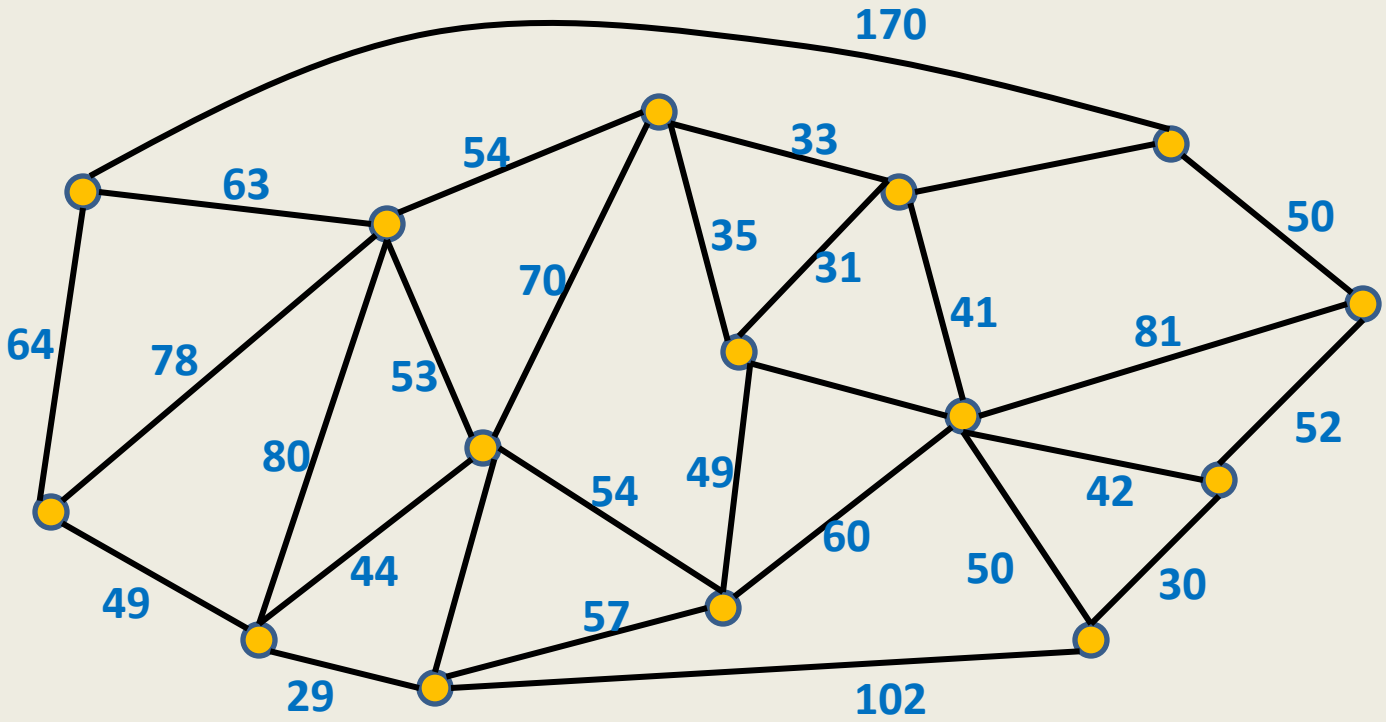


Observation: Given a spanning tree T of a graph G, adding a nontree edge e to T creates a unique cycle.

There will be total $m - n + 1$ such cycles. These are called **fundamental cycles** in G induced by the spanning tree T.

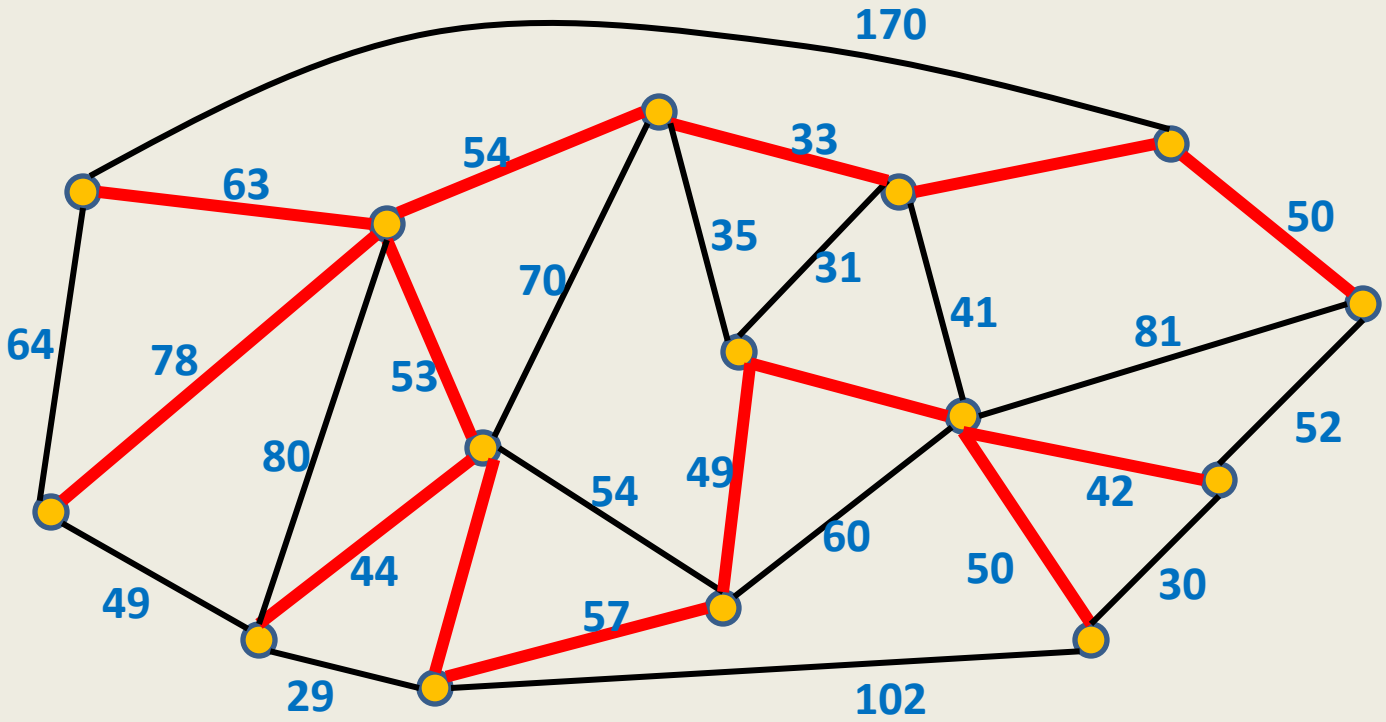
A road or telecommunication network

Assign each edge a **weight/cost**.



Adding more reality to the problem

A road or telecommunication network



Any arbitrary spanning tree (like the one shown above) will not serve our goal☹.

We need to select the spanning tree with **least weight/cost**.

Problem 2

Minimum spanning tree

Problem Description

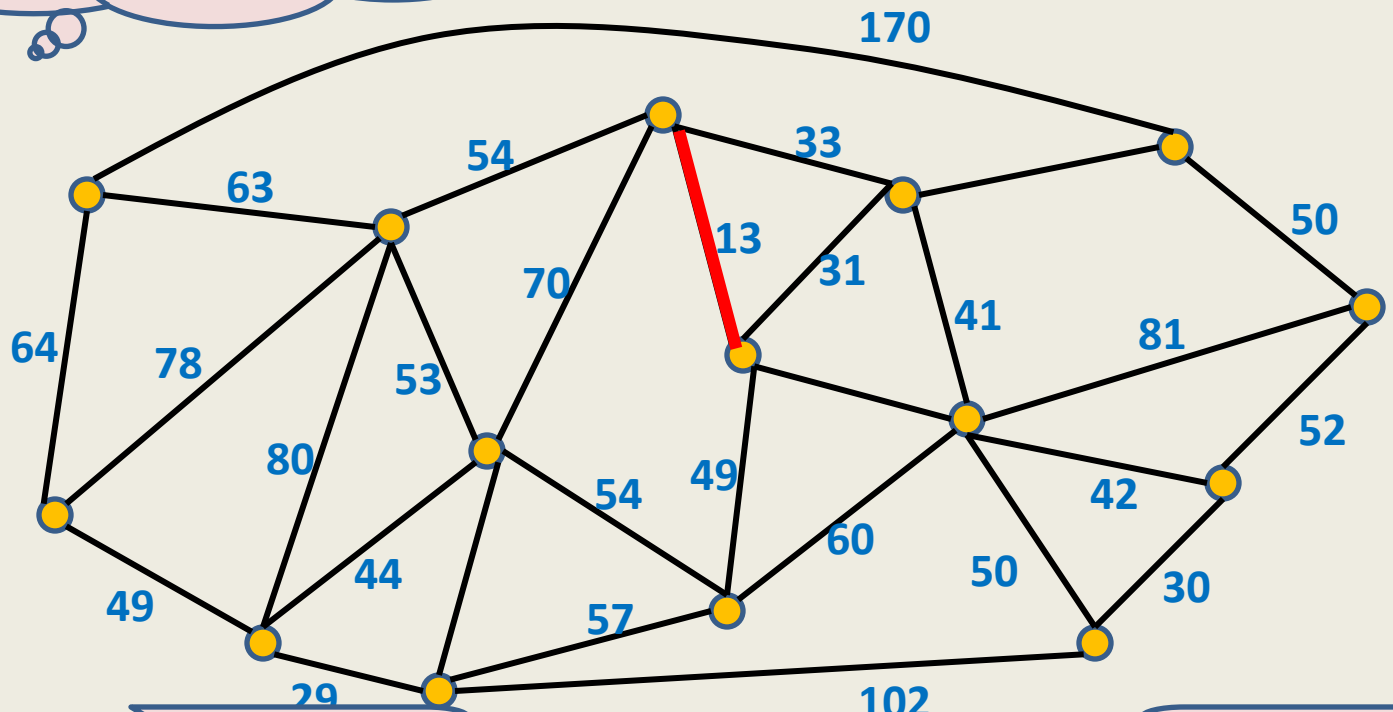
Input: an undirected graph **G**=(**V**,**E**) with **w**: **E** → $\mathbb{R}$,

Aim: compute a **spanning tree** (**V**,**E'**), **E'** ⊆ **E** such that

$\sum_{e \in E'} w(e)$ is **minimum**.

How to compute a MST ?

The least weight edge should be in MST.
But why ?



Look a nd.
Ca

Is there any edge for which you feel strongly to be present in MST ?

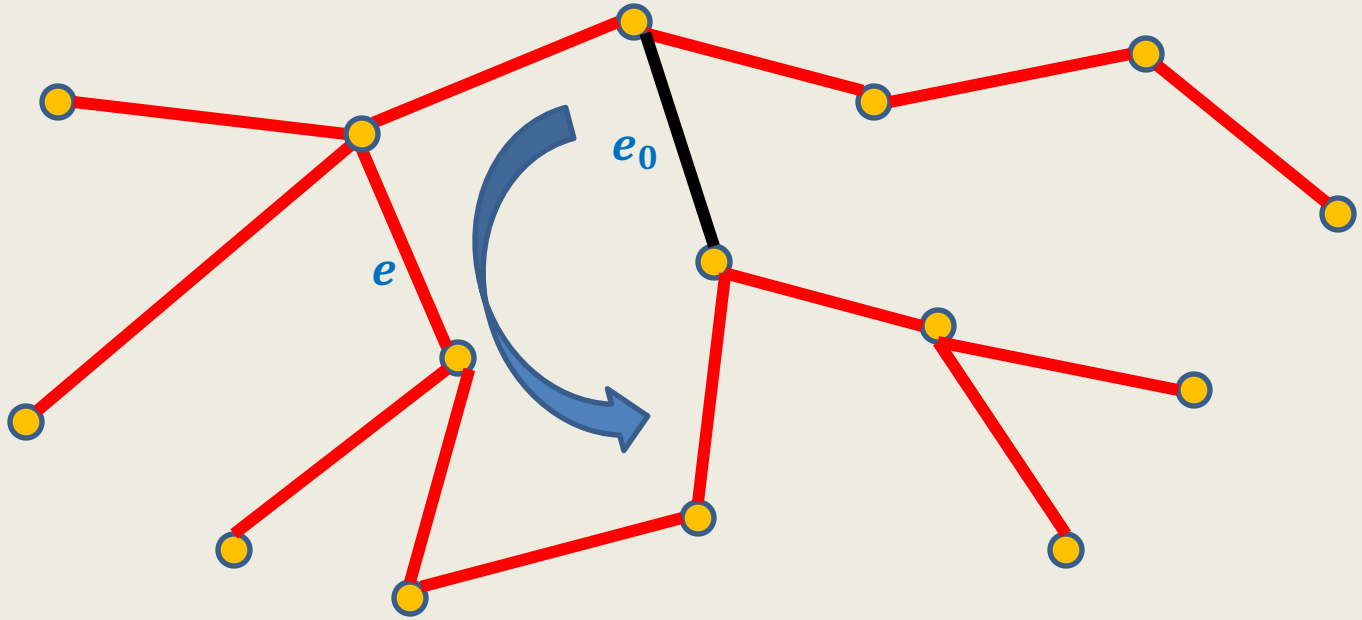
Let $e_0 \in E$ be the edge of least weight in the given graph.

Lemma2: There is a **MST** T containing e_0.

Proof: Consider any **MST** T. Let $e_0 \notin T$.

Consider the fundamental cycle C defined by e_0 in T.

Swap e_0 with any edge $e \in T$ present in C.



Let $e_0 \in E$ be the edge of least weight in the given graph.

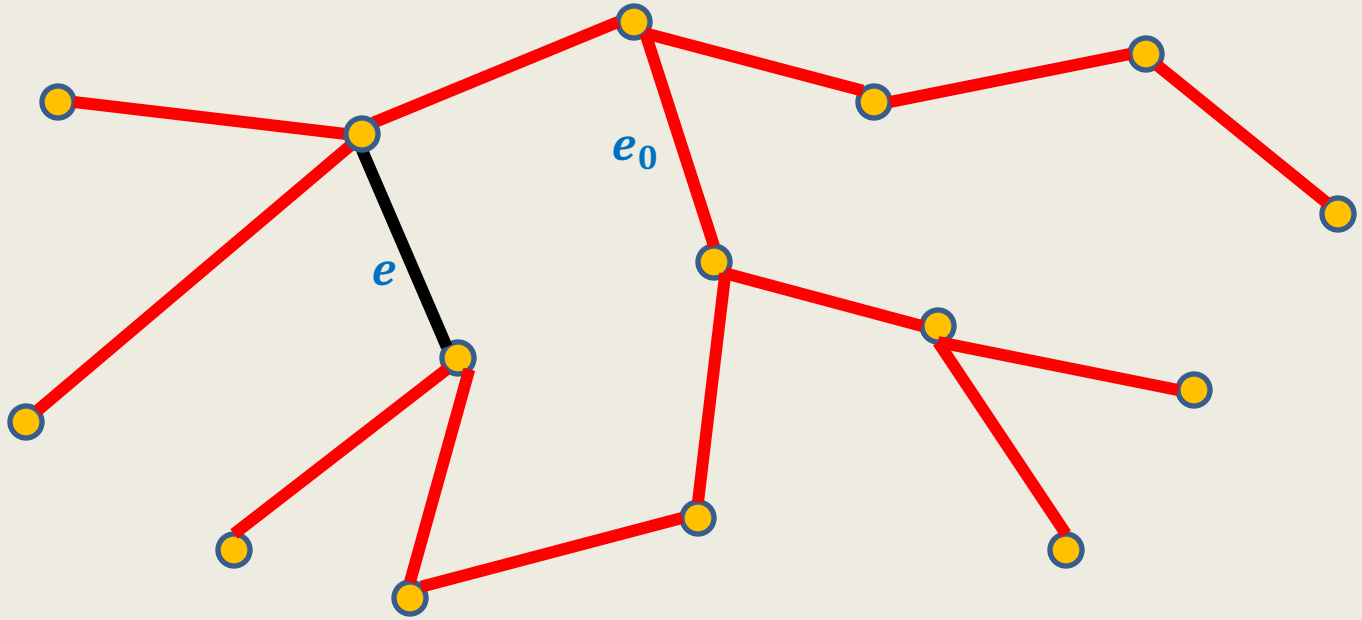
Lemma2: There is a **MST** T containing e_0.

Proof: Consider any **MST** T. Let $e_0 \notin T$.

Consider the fundamental cycle C defined by e_0 in T.

Swap e_0 with any edge $e \in T$ present in C.

We get a spanning tree of weight $\leq w(T)$.



Try to translate Lemma2 to an algorithm for MST ?

with **inspiration** from the job scheduling problem ☺

Data Structures and Algorithms

(ESO207)

Lecture 36

- A new algorithm design paradigm: Greedy strategy

  part III

Continuing Problem from last class

Minimum spanning tree

Problem Description

Input: an undirected graph $G=(V,E)$

Aim: compute a **spanning tree** (V, E')

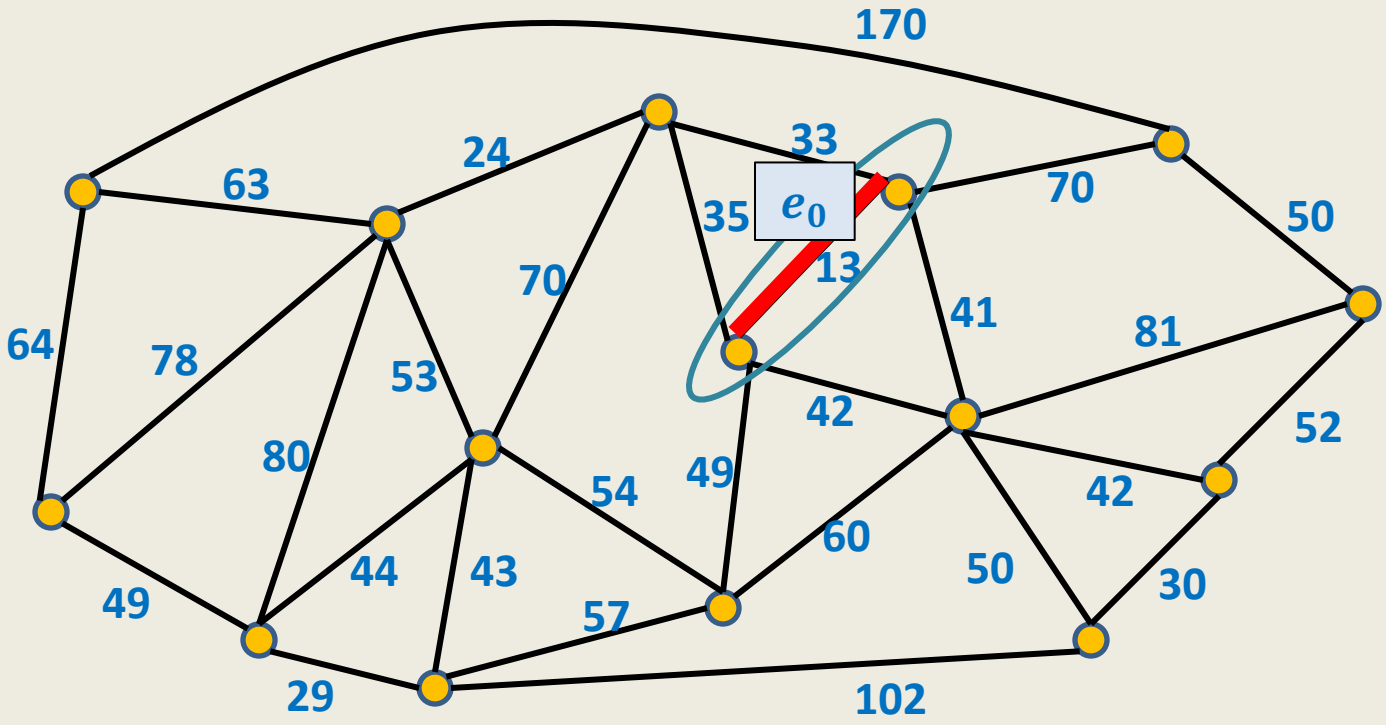
Lemma (proved in last class):

If $e_0 \in E$ is the edge of **least weight** in G,

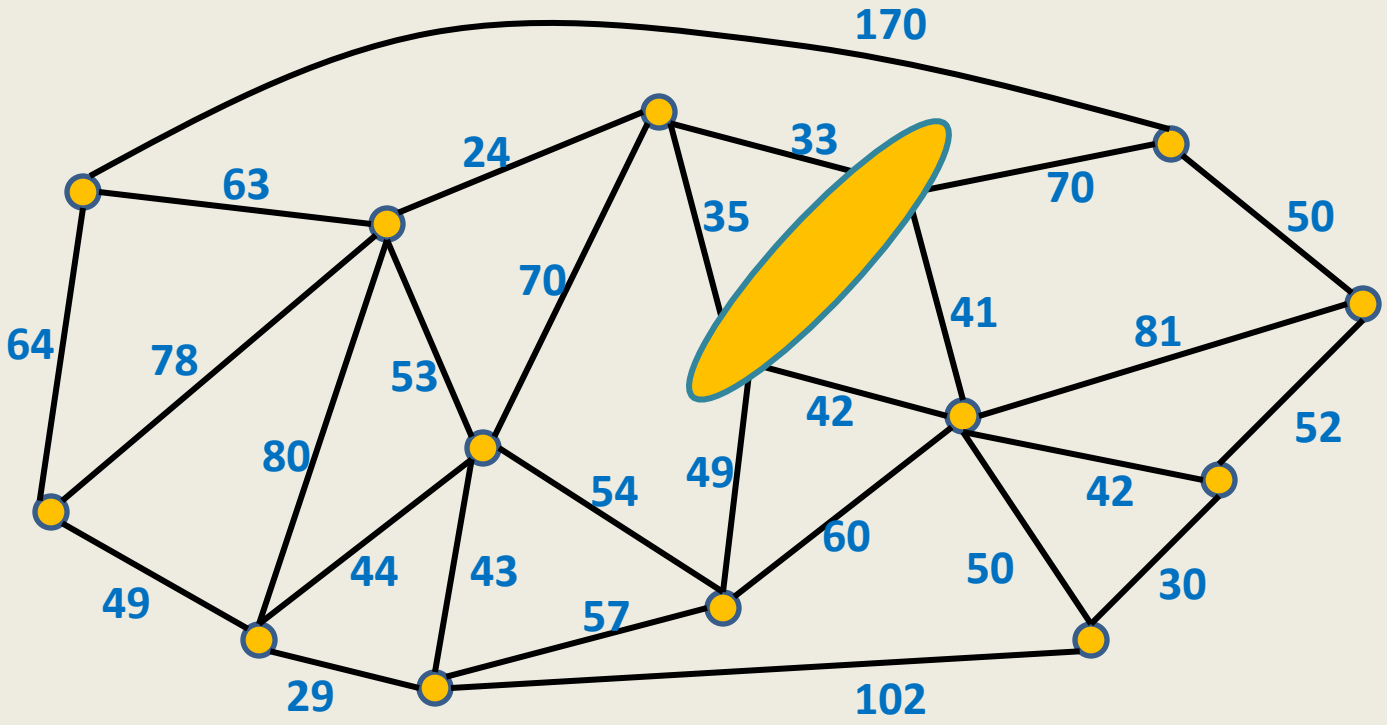
then there is a **MST** T containing e_0.

How to use this **Lemma** to design an algorithm for **MST** ?

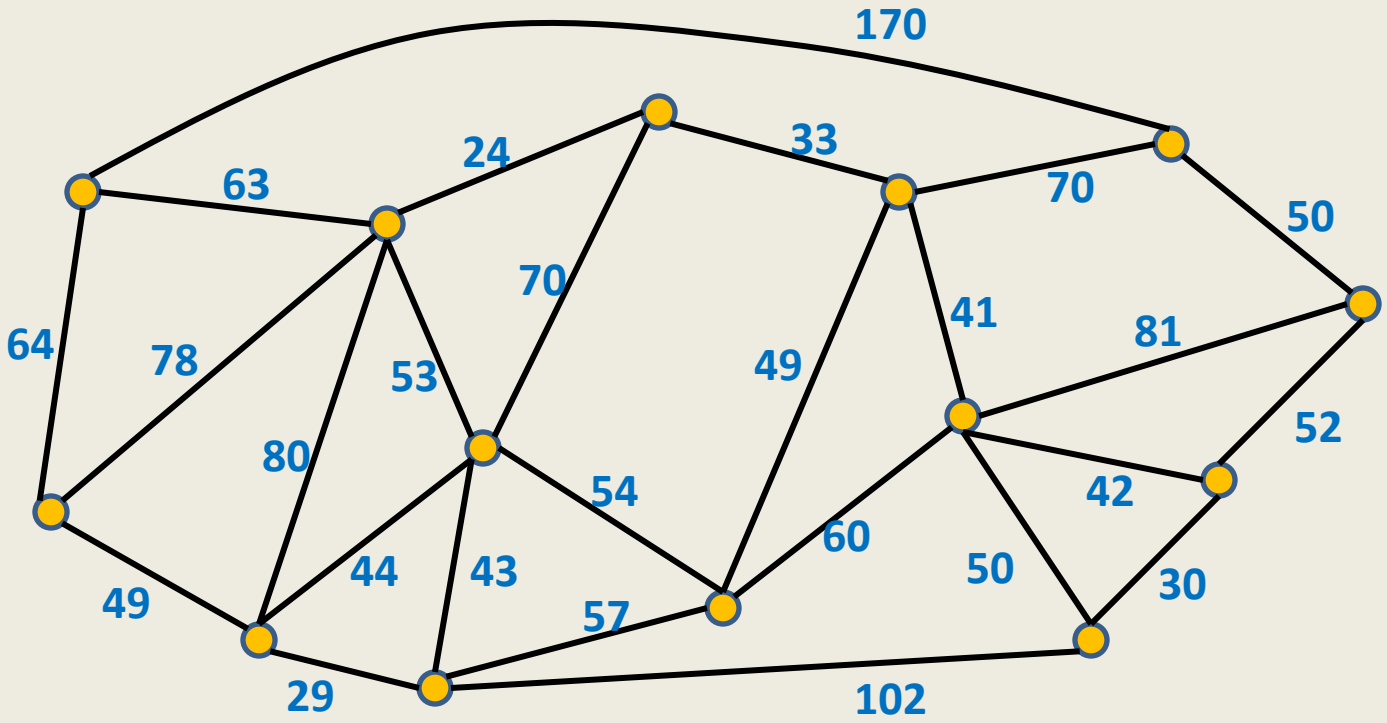
Minimum Spanning Tree (MST)



Minimum Spanning Tree (MST)



Minimum Spanning Tree (MST)

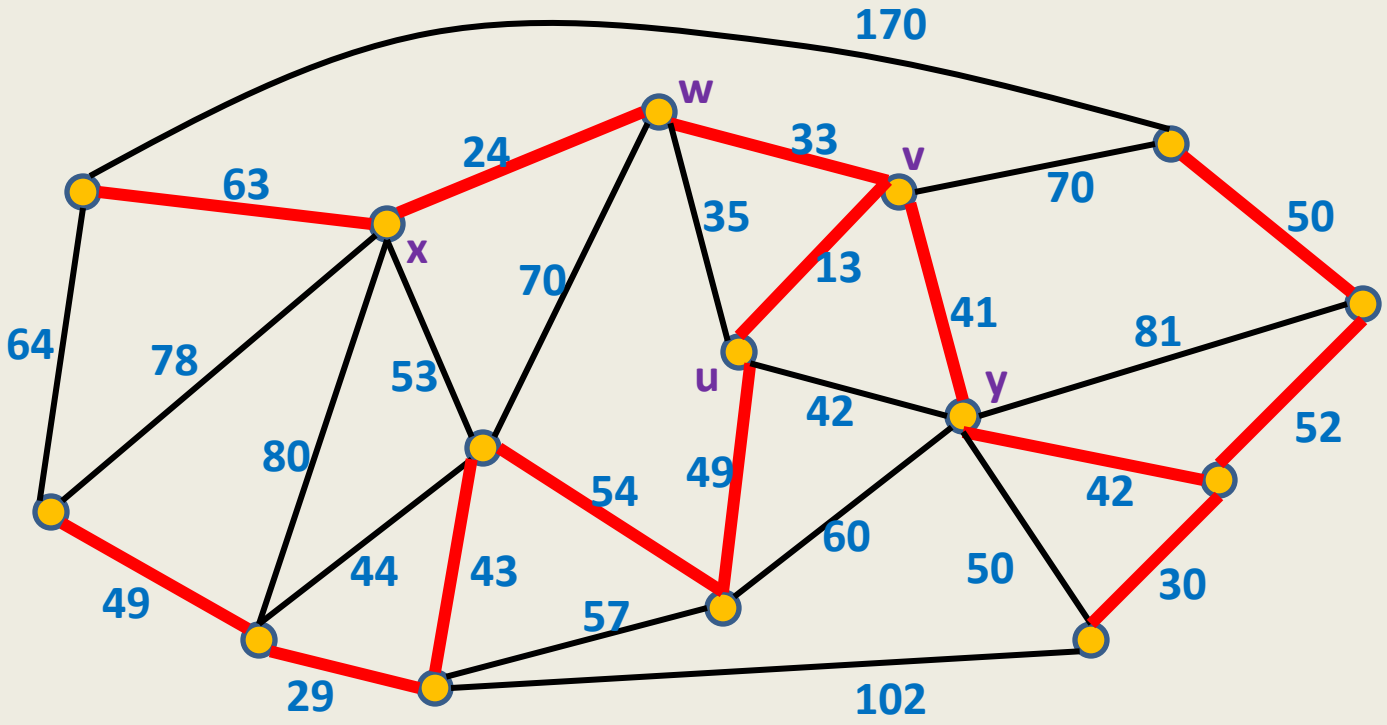


G'

Theorem:

w(MST(G)) =

Minimum Spanning Tree (MST)



A useful lesson for design of a graph algorithm

If you have a complicated algorithm for a graph problem, …

- search for **some graph theoretic property**

to design **simpler** and **more efficient** algorithm

Two graph theoretic properties of MST

- **Cut property**
- **Cycle property**

Every algorithm till date is **based on** one of **these properties**!

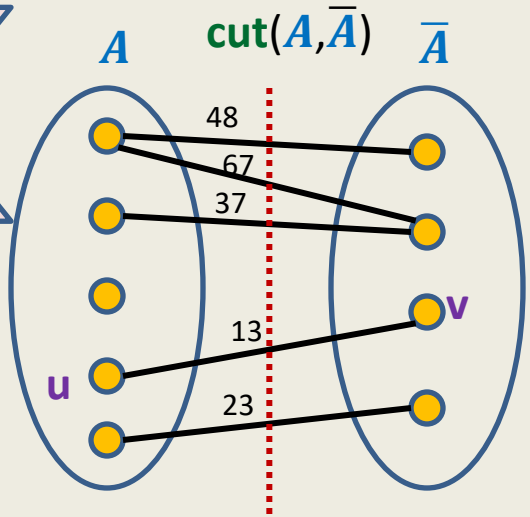
Cut Property

Cut Property

Definition: For any subset $A \subseteq V$

cut$(A, \overline{A})$ = { $(u,v) \in E$ |

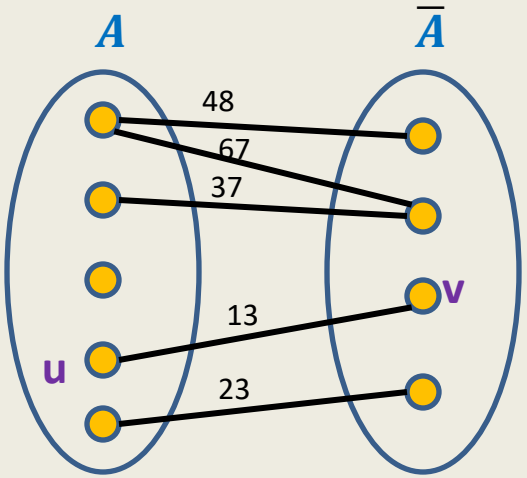
Pursuing **greedy strategy** to minimize weight of MST, what can we say about the edges of cut$(A, \overline{A})$?



Cut-property:

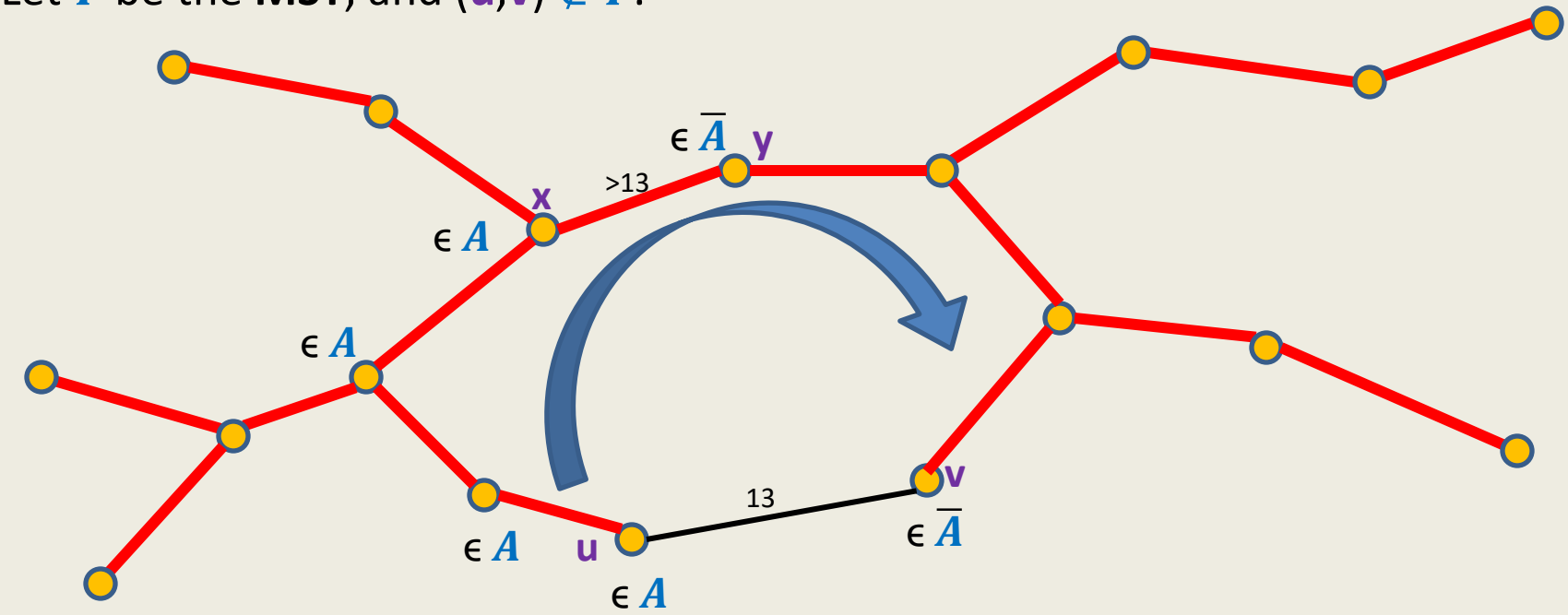
The **least weight edge** of a cut$(A, \overline{A})$ must be in **MST.**

Proof of cut-property



Proof of cut-property

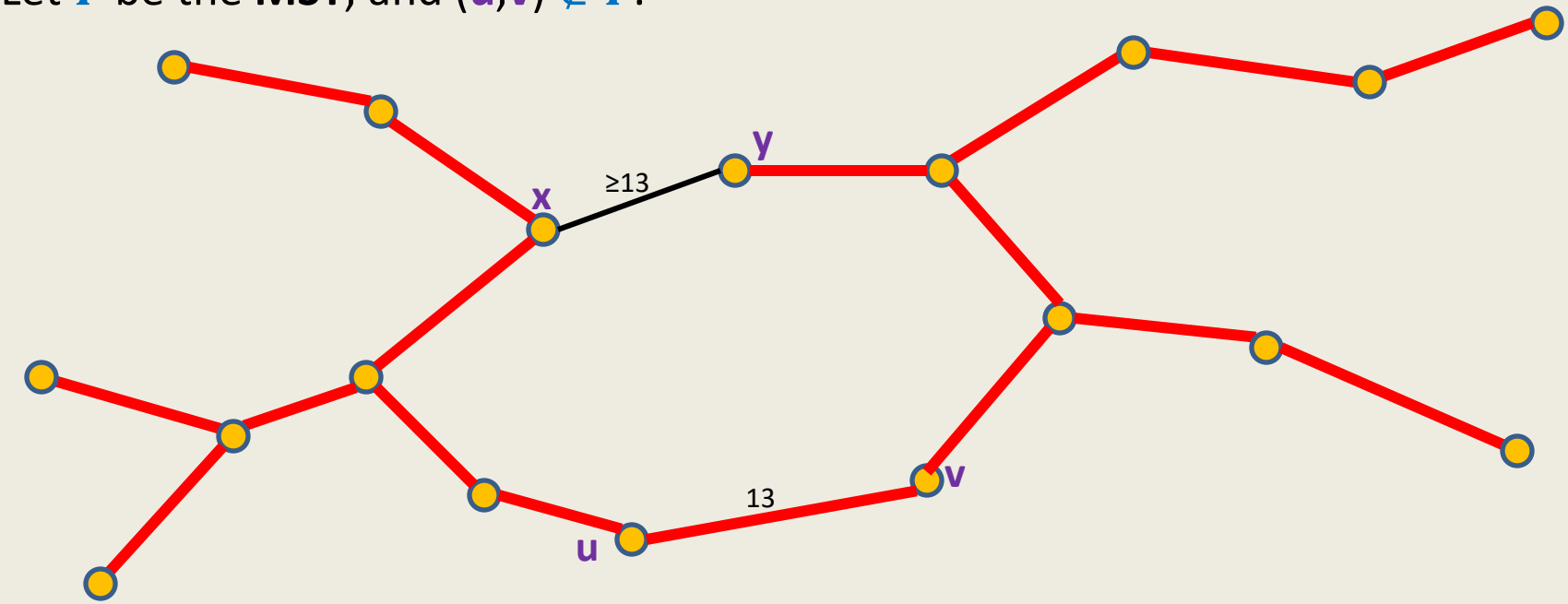
Let T be the **MST**, and (u,v) $\notin T$.



Question: What happens if we **remove** (x,y) from T, and **add** (u,v) to T.

Proof of cut-property

Let T be the **MST**, and (u,v) $\notin T$.



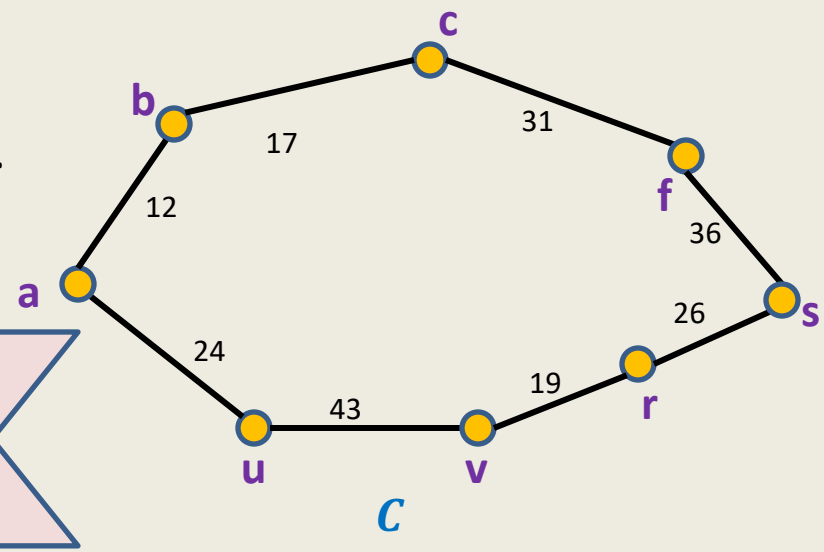
Question: What happens if we **remove** (x,y) from T, and **add** (u,v) to T.

We get a spanning tree T' with weight < **weight**(T)
A contradiction !

Cycle Property

Cycle Property

Let C be any cycle in G.

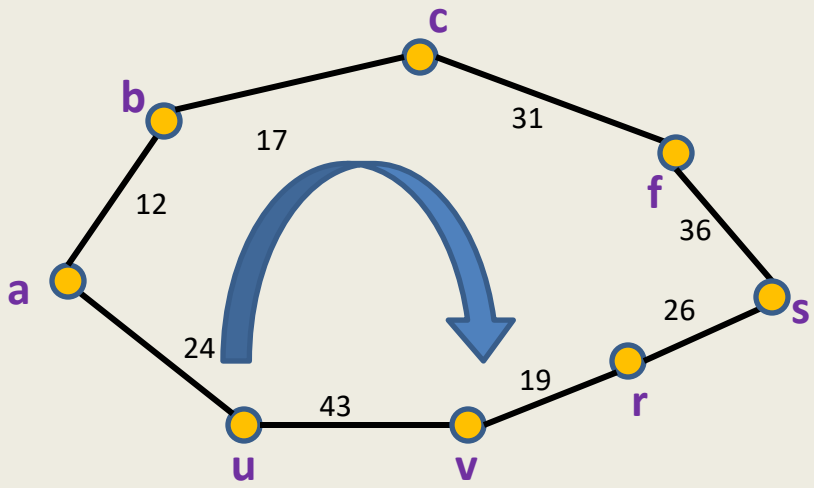


Pursuing **greedy strategy** to minimize weight of MST, what can we say about the edges of cycle C ?

Cycle-property:

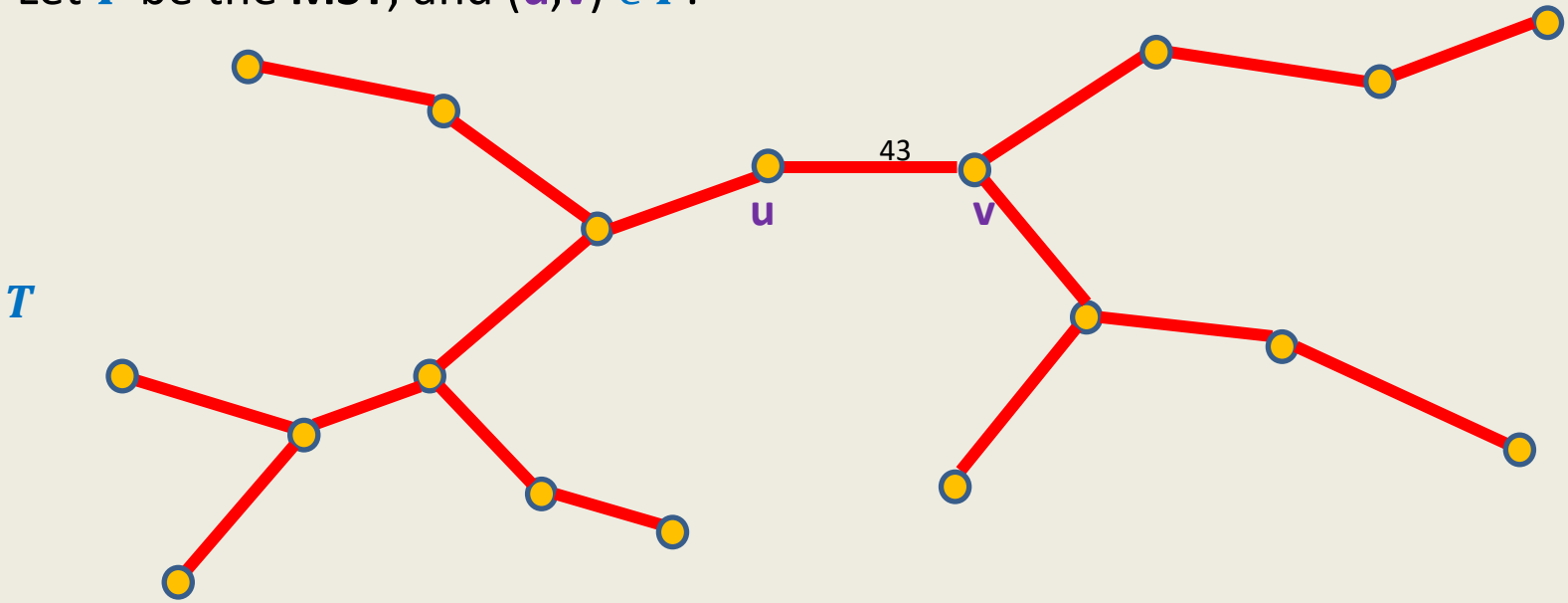
Maximum weight edge of any cycle C **can not** be present in **MST**.

Proof of Cycle property



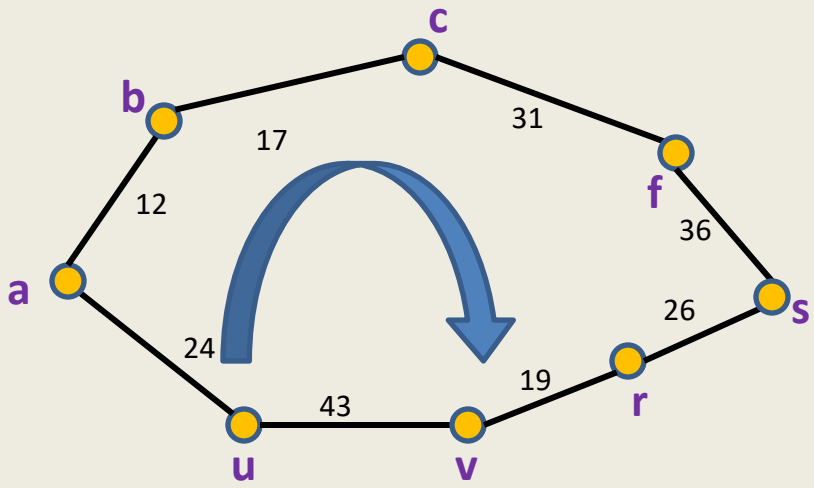
C

Let T be the **MST**, and (u,v) $\in T$.

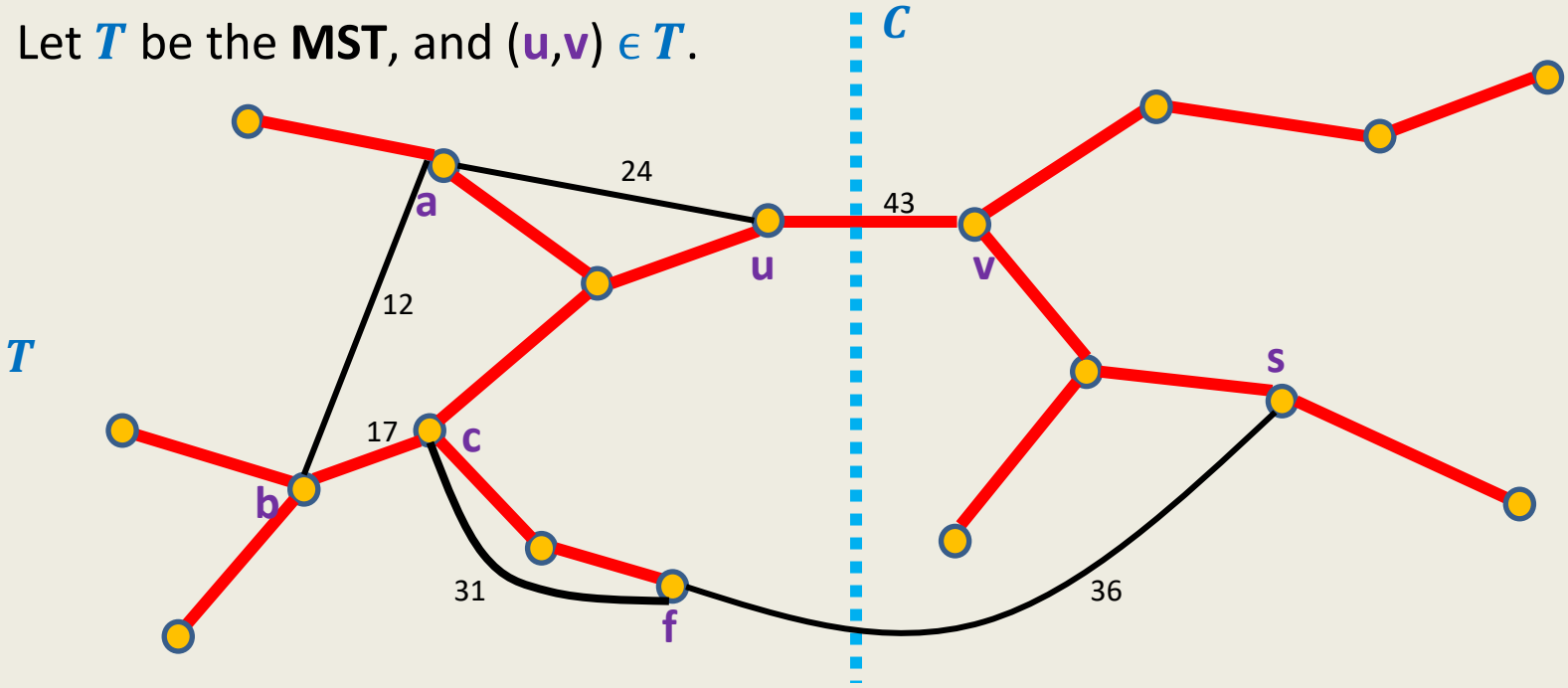


T

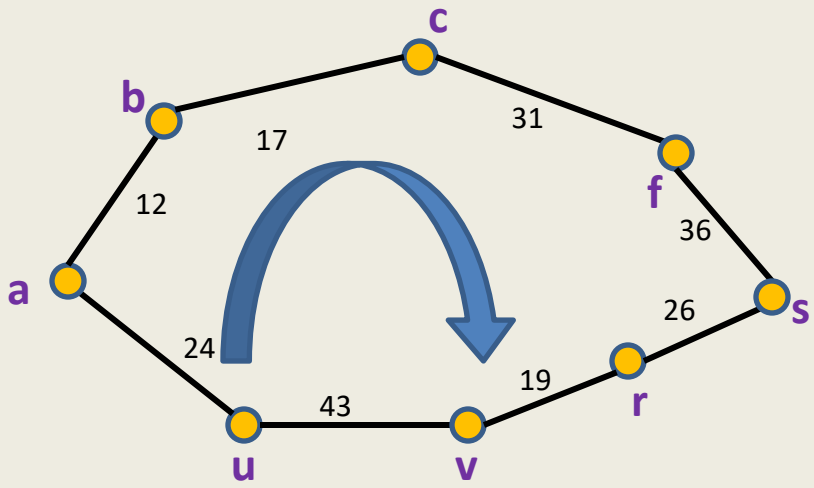
Proof of Cycle property



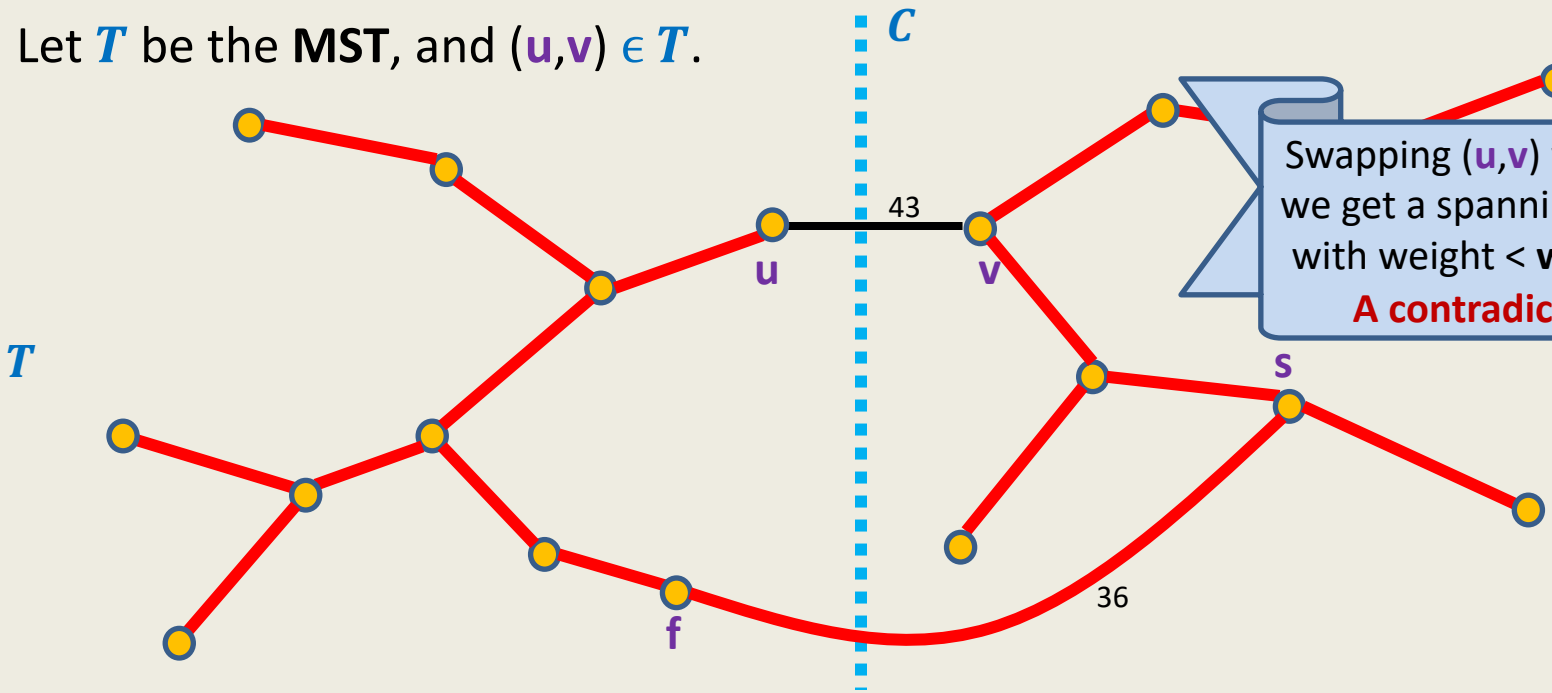
Let T be the **MST**, and (u,v) $\in T$.



Proof of Cycle property



Let T be the **MST**, and (u,v) $\in T$.

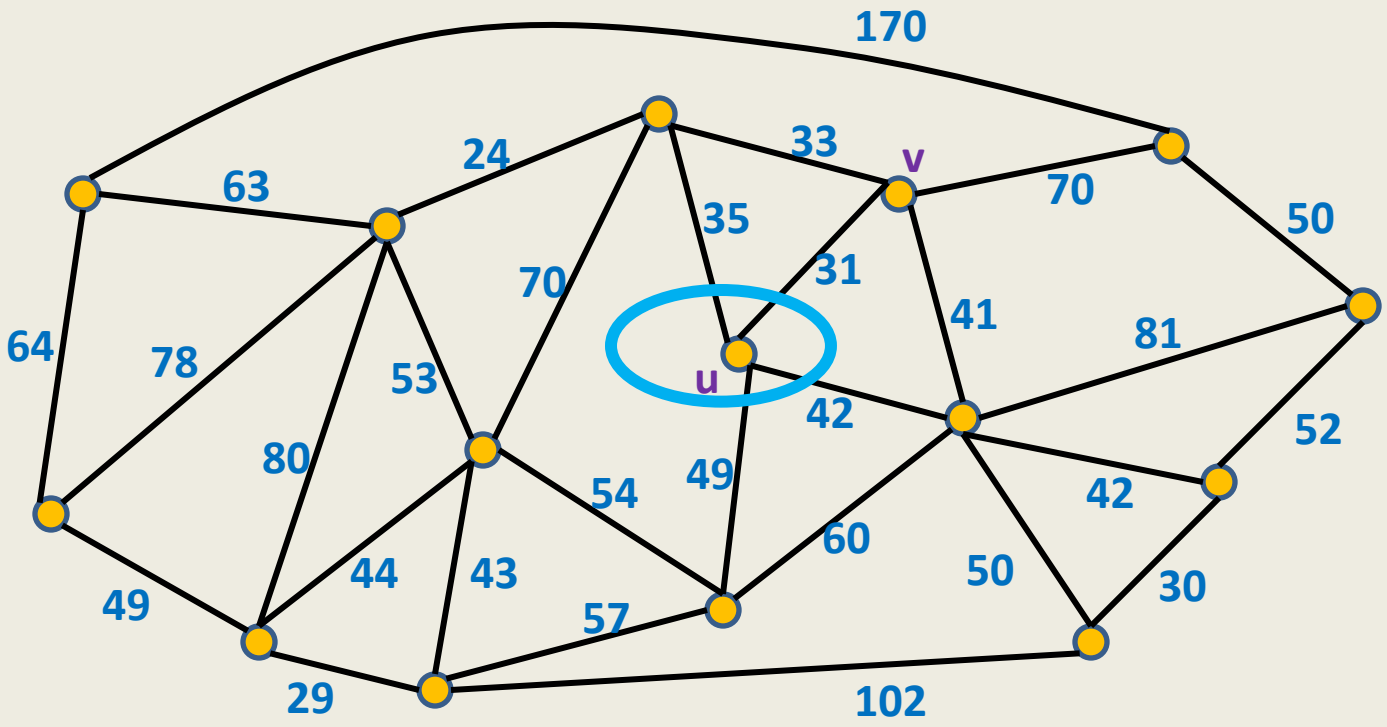


Swapping (u,v) with (f,s), we get a spanning tree T' with weight < **weight**(T)

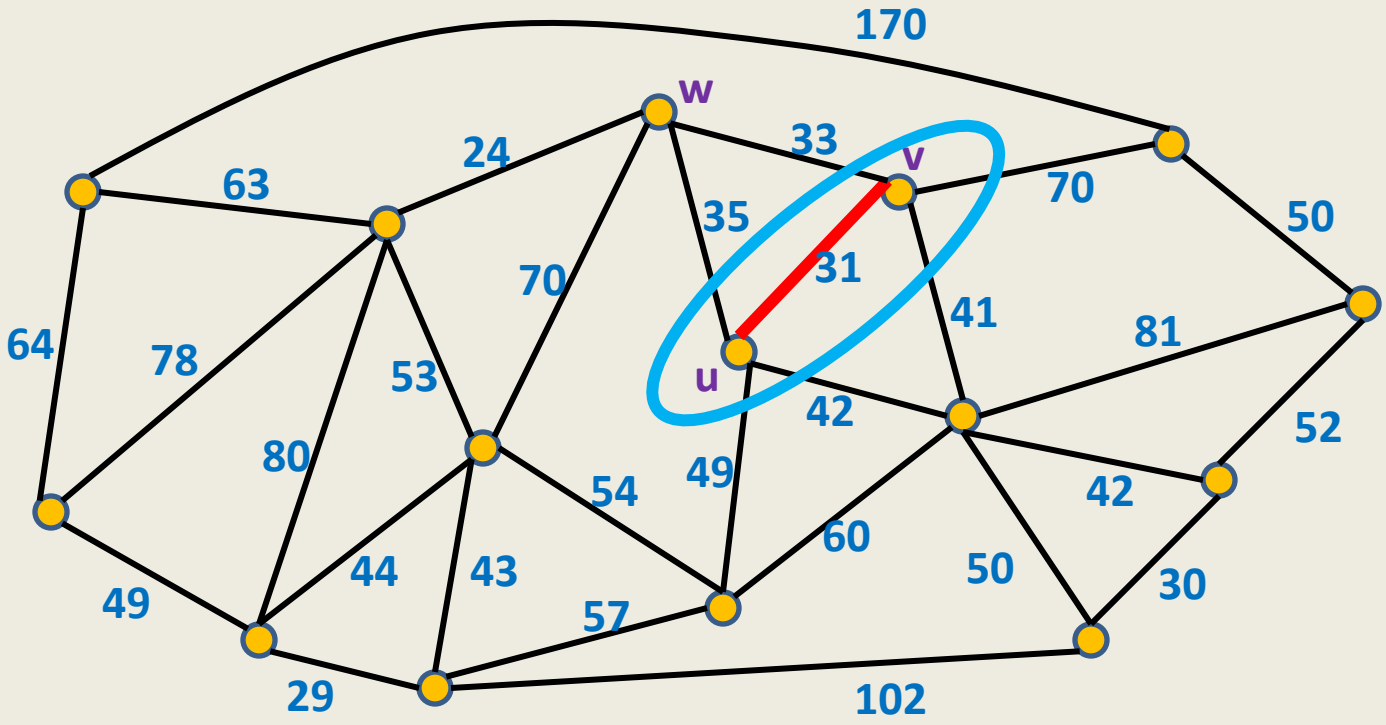
A contradiction !

Algorithms based on cut Property

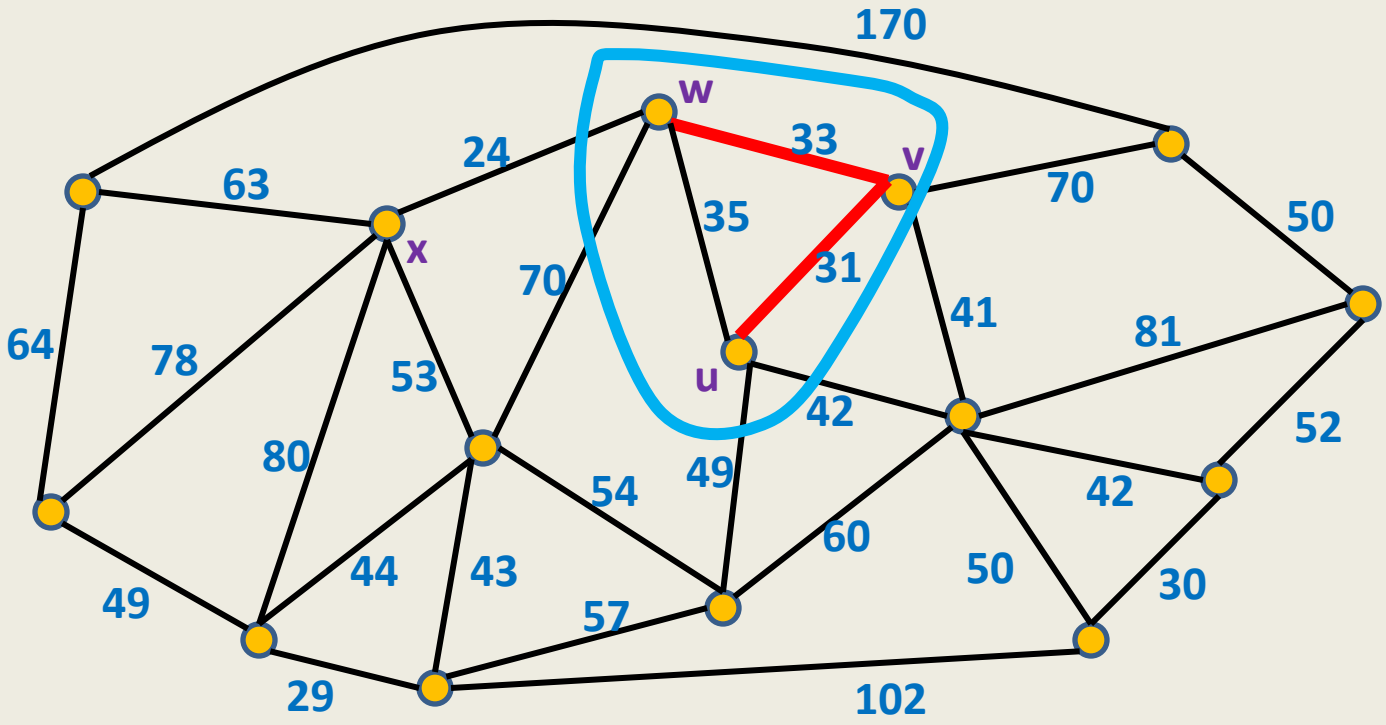
How to use cut property to compute a MST ?



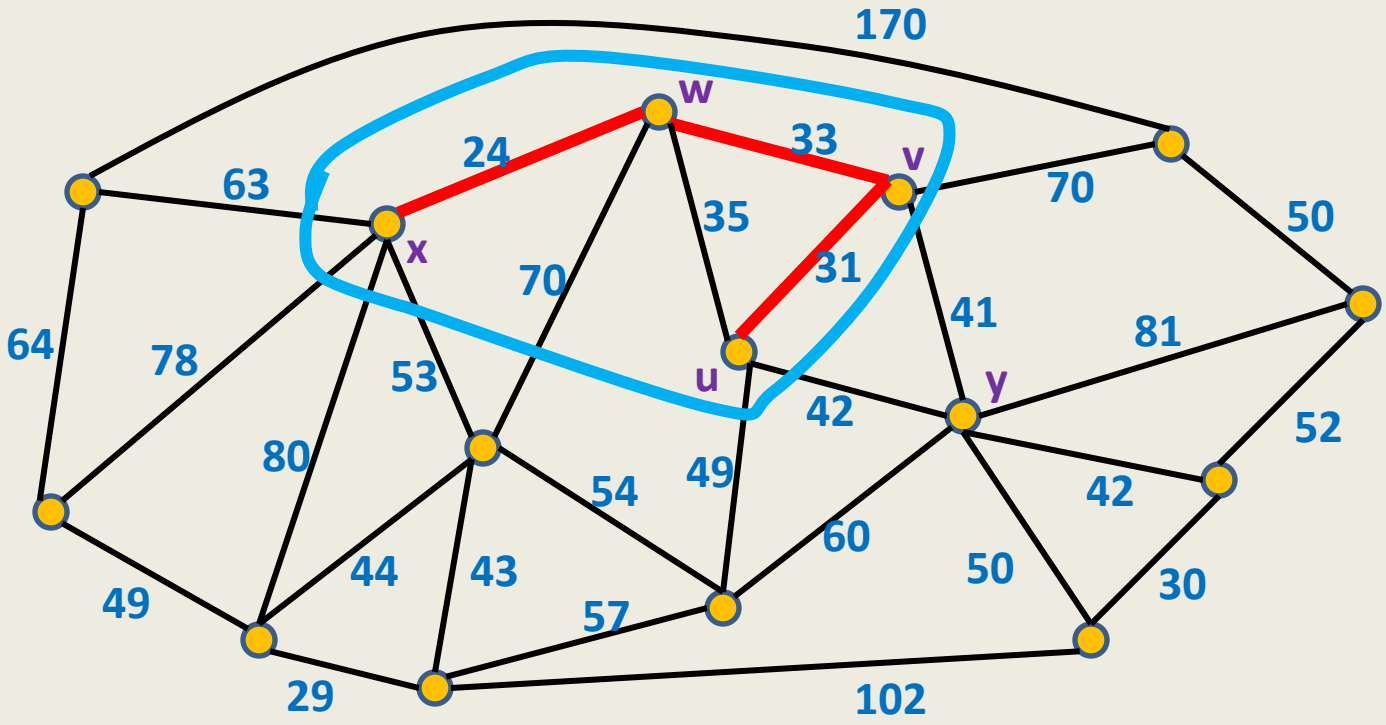
How to use cut property to compute a MST ?



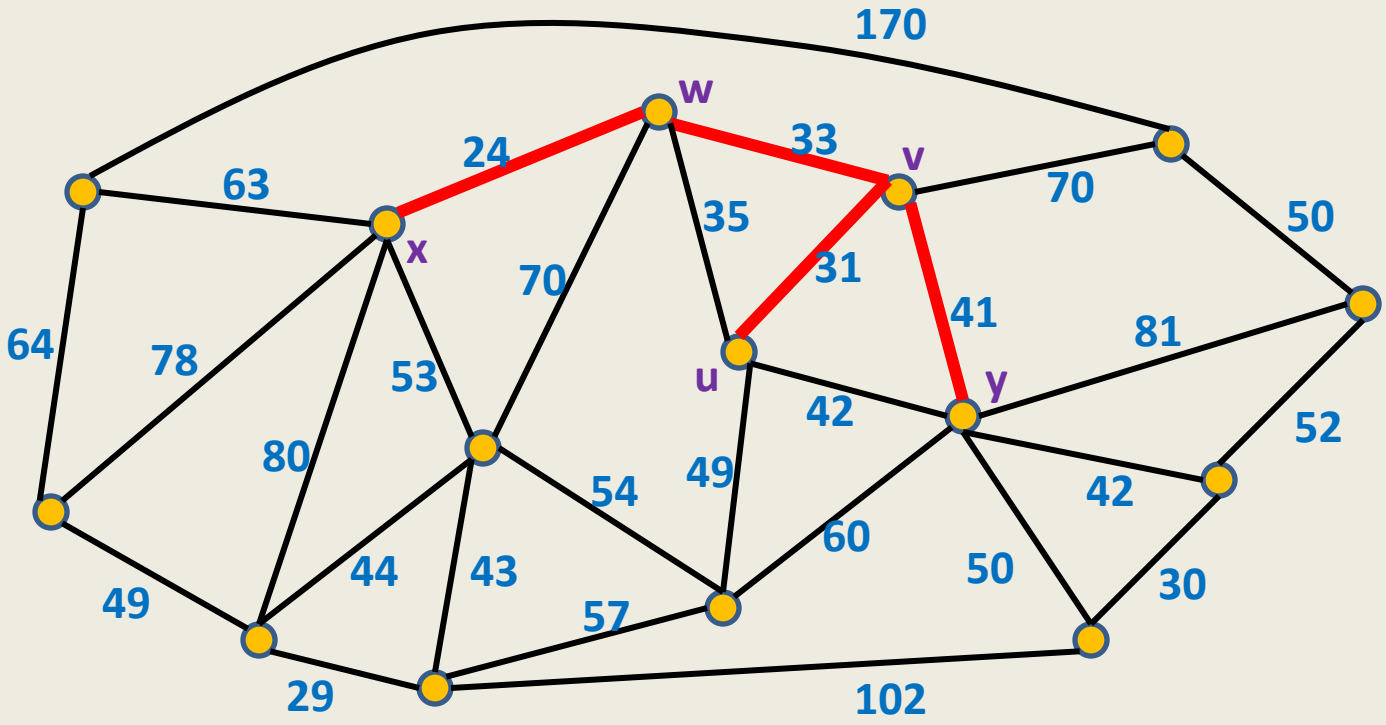
How to use cut property to compute a MST ?



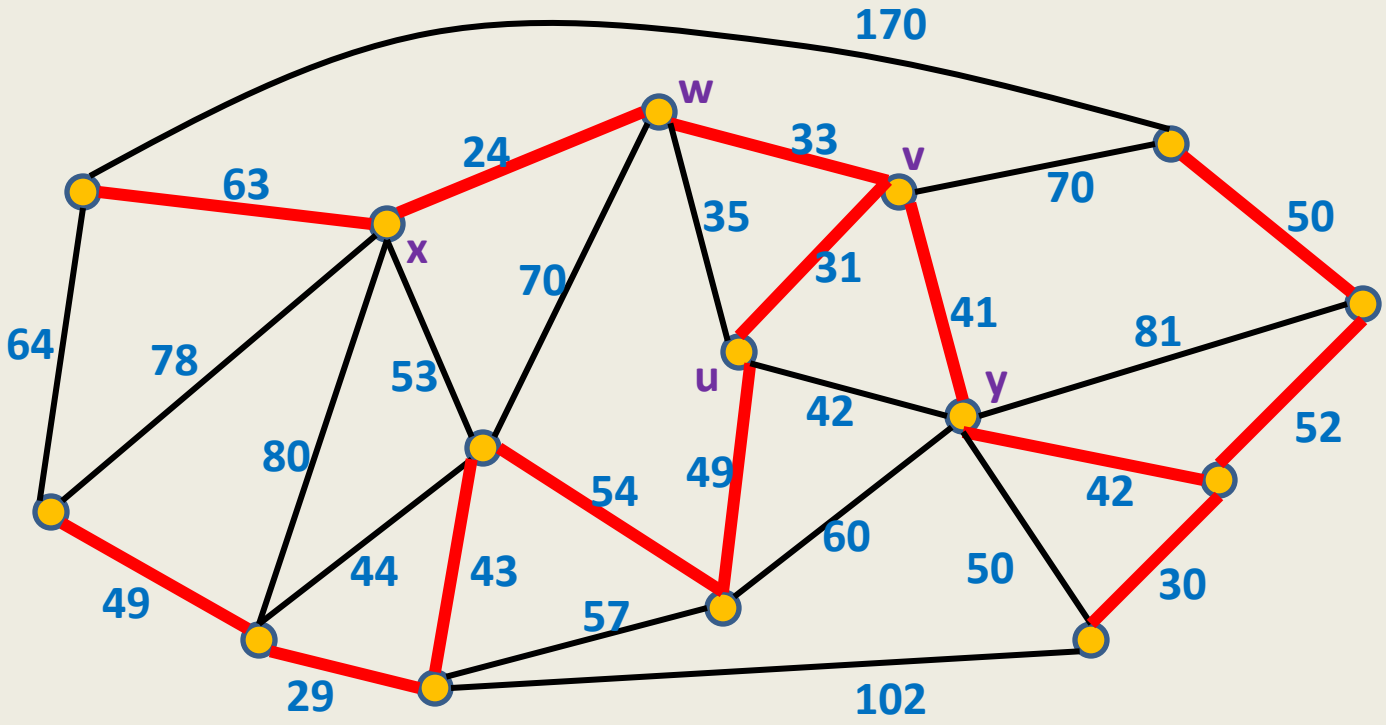
How to use cut property to compute a MST ?



How to use cut property to compute a MST ?



How to use cut property to compute a MST ?



An Algorithm based on cut property

Algorithm (Input: **graph** G =(V,E) with **weights** on edges)

$T \leftarrow \emptyset$;

$A \leftarrow$ {u};

While ($A <> V$ **) do**

{ Compute the least weight edge from **cut**(A,$\overline{A}$);

Let this edge be (x,y), with x$\in A$, y$\in \overline{A}$;

$T \leftarrow T \cup$ {(x, y)};

$A \leftarrow A \cup$ {y};

}

Return T;

Number of iterations of the **While** loop **:** $n-1$

Time spent in one iteration of While loop: **O**(m)

➔ **Running time** of the algorithm: **O**(mn)

Algorithm based on cycle Property

An Algorithm based on cycle property

Description

Algorithm (Input: **graph** G =(V,E) with **weights** on edges)

While (E **has any cycle** **) do**

{ Compute any cycle C;

Let (**u**,**v**) be the **maximum weight** edge of the cycle C;

Remove (**u**,**v**) from E;

}

Return E;

Number of iterations of the **While** loop : $m - n + 1$

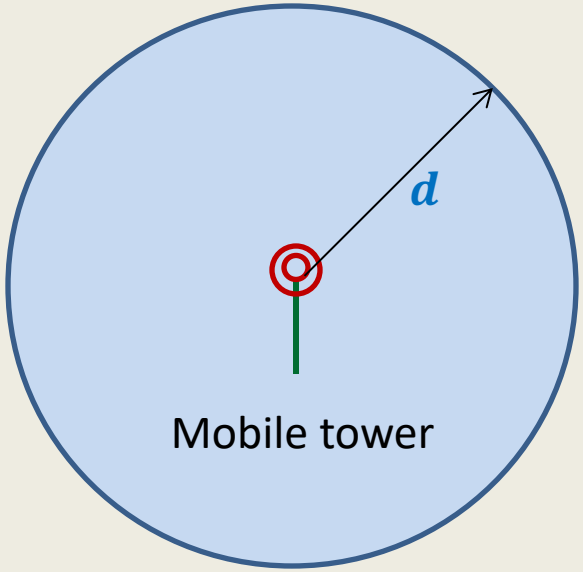
Time spent in one iteration of While loop: $O(n)$

➔ **Running time** of the algorithm: $O(mn)$

Problem 3

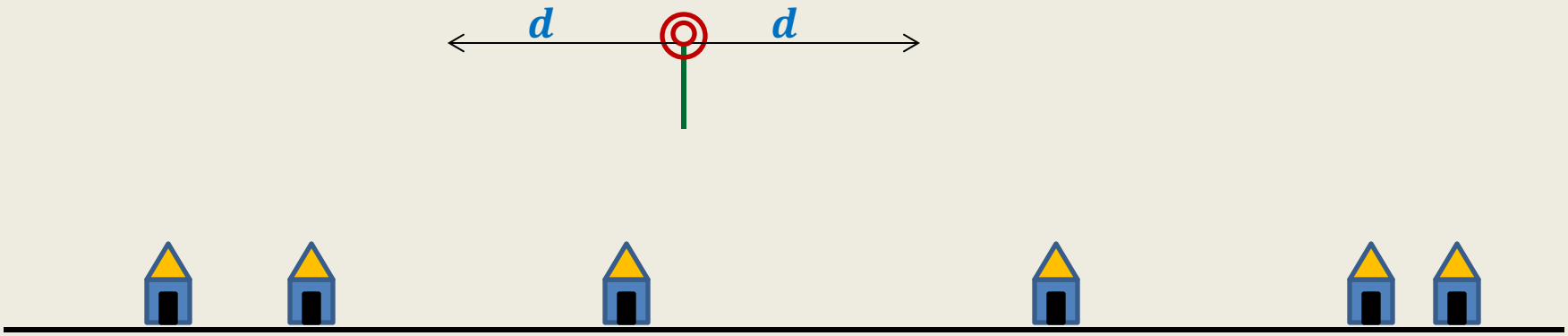
Mobile towers on a road

Mobile towers on a road



A mobile tower can cover any cell phone within radius d.

Mobile towers on a road



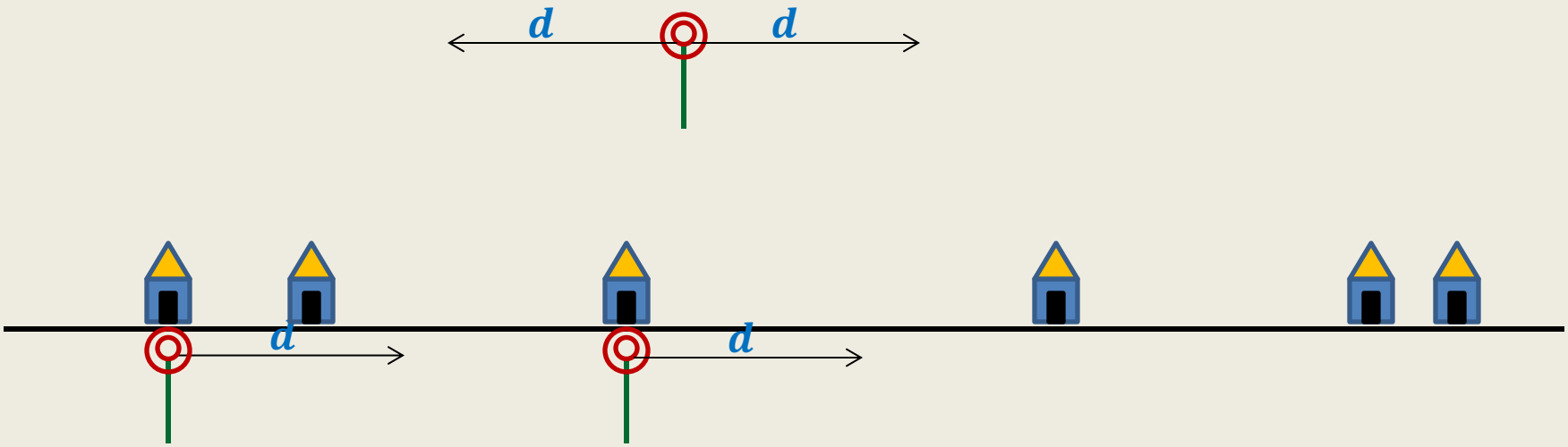
Problem statement:

There are n houses located along a road.

We want to place mobile towers such that

- Each house is **covered** by at least one mobile tower.
- The number of mobile towers used is **least** possible.

Mobile towers on a road



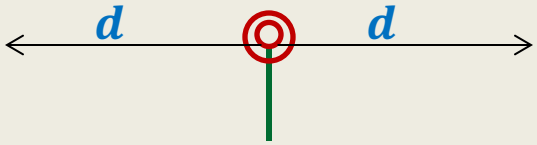
Strategy 1:

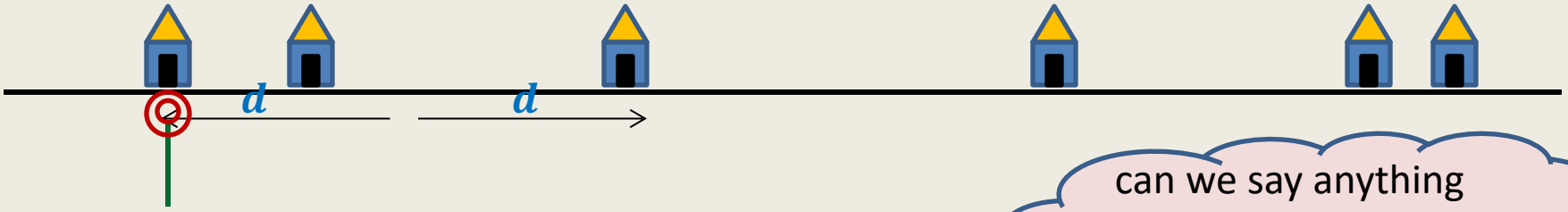
Place tower at first house,

Remove all houses covered by this tower.

Proceed to the next uncovered house …

Mobile towers on a road





can we say anything about the optimal solution ?

Strategy 2:

Place tower at distance d to the right of the first house;

Remove all houses covered by this tower;

Proceed to the next uncovered house along the road…

Lemma: There is an optimal solution for the problem in which the leftmost tower is placed at distance d to the right of the first house

Homework …

Ponder over the following questions before coming for the next class

- Use **cycle property** and/or **cut property** to design a **new algorithm for MST**
- Use some data structure to improve the running time of the algorithms discussed in this class to **O**($m \log n$)

Data Structures and Algorithms

(ESO207)

Lecture 37

- A new algorithm design paradigm: Greedy strategy

part IV

Problems solved till now

1. Job Scheduling Problem
2. Mobile Tower Problem
3. MST

Ponder over this question before moving ahead

Problem 1

Job scheduling Problem

INPUT:

- A set J of n jobs $\{j_1, j_2, \ldots, j_n\}$
- job j_i is specified by two real numbers

  s(i): start time of job j_i

  f(i): finish time of job j_i
- A single server

We could not say anything about the **complete solution** of this problem.

Constraints:

- Server can execute at most one job at any moment of time and a job.
- **Job j_i**, if scheduled, has to be scheduled during[**s**(i), **f**(i)] only.

Aim:

To select the **largest** subset of **non-overlapping** jobs which can be executed by the server.

Problem 1
Job scheduling Problem

INPUT:

- A set J of n jobs $\{j_1, j_2, \ldots, j_n\}$
- job j_i is specified by two real numbers

  s(i): start time of job j_i

  f(i): finish time of job j_i
- A single server

Constraints:

- Server can execute at most one job at any moment of time and a job.
- **Job j_i**, if scheduled, has to be scheduled during[**s**(i), **f**(i)] only.

Aim:

To select the **largest** subset of **non-overlapping** jobs which can be executed by the server.

All that we could do was to make a local observation

Let $x \in J$ be the job with earliest finish time.

Lemma1 : There exists an optimal solution for J in which x is present.

Let $J' = J \setminus \text{Overlap}(x)$

Lemma 1 gives very small information about the optimal solution ☹
How to use it to compute this solution ?

J(original instance)

Greedy step

J' (**smaller** instance)

Opt(J)

Lemma1

Opt(J')

Opt(J)= Opt(J') + 1 -- (i)

Equation (i) hints at recursive solution of the problem ☺

Theorem: $\mathbf{Opt}(J) = \mathbf{Opt}(J') + 1$.

- Proof has two parts

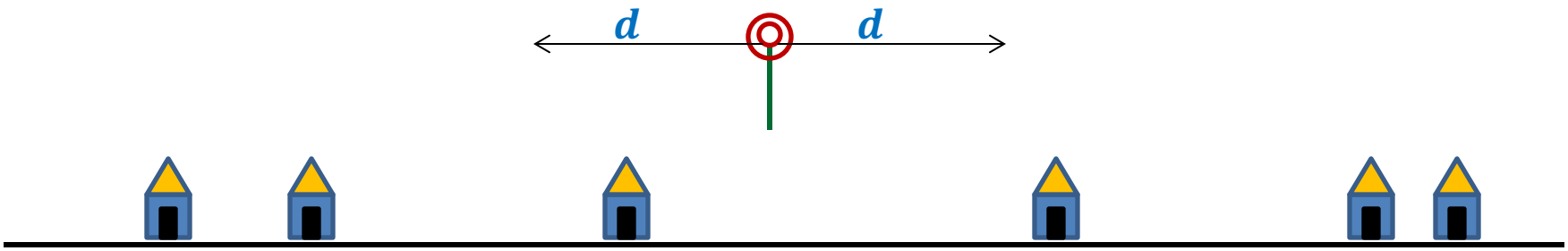
  $\mathbf{Opt}(J) \geq \mathbf{Opt}(J') + 1$

  $\mathbf{Opt}(J') \geq \mathbf{Opt}(J) - 1$

- Proof for each part is a proof by **construction**

Problem 2

Mobile Tower Problem



We could not say anything about the **complete solution** of this problem.

Problem statement:

There is a set H of n houses located along a roa

We want to place mobile towers such that

- Each house is **covered** by at least one mobile tower.
- The number of mobile towers used is **least** possible.

All that we could do was to make a local observation

Lemma 2: There is an optimal solution for the problem in which the leftmost tower is placed at distance d to the right of the first house.

Let x be the tower loca

Let H' = $H \setminus$ {all house

Lemma 1 gives very small information about the optimal solution ☹
How to use it to compute this solution ?

H(original instance)

Opt(H)

Opt(H)= **Opt**(H') + **1**

Greedy step

Lemma 2

H'(**smaller** instance)

Opt(H')

Equation **(i)** hints at recursive solution of the problem ☺

What is a greedy strategy ?

A strategy that is

- Based on some **local** approach
- With the **objective to optimize** some function.

Note:

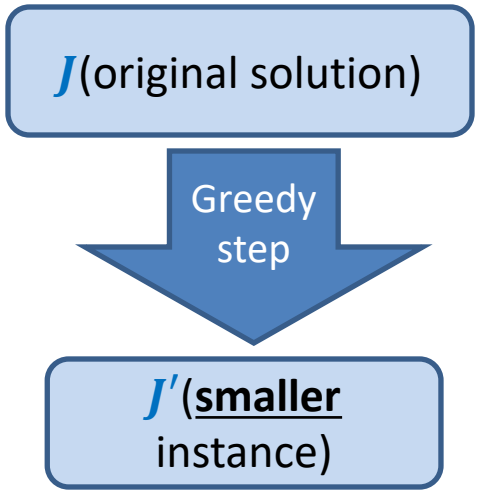
Recall that the divide and conquer strategy takes a **global approach**.

Design of a greedy algorithm

Let **A** be an instance of an optimization problem.

1. Make **a local observation** about the solution.

2. Use this observation to express optimal solution of **A** in terms of
   - Optimal solution of a smaller instance **A'**
   - Local step



3. This gives a recursive solution.
4. Transform it into iterative one.

MST

Input: an undirected graph $G=(V,E)$ with $w: E \rightarrow \mathbb{R}$,

Aim: compute a **spanning tree** (V, E'), $E' \subseteq E$ such that $\sum_{e \in E'} w(e)$ is **minimum.**

If you have understood a generic way to design a greedy algorithm, then try to solve the MST problem.

Lemma

If $e_0 \in E$ is the edge of **least weight** in G, then there is a **MST** T containing e_0.

How to use this **Lemma** to design an algorithm for **MST** ?

Problem 4
Overlapping Intervals

The aim of this problem is to make you realize that

it is sometime very nontrivial to design a greedy algorithm.
In particular, it is quite challenging to design the smaller instance.

Problem 4
Overlapping Intervals

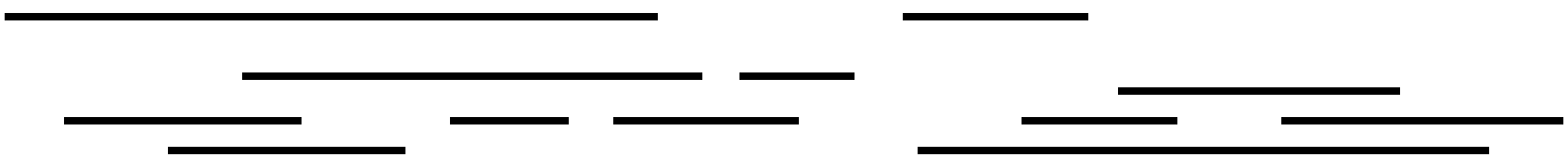
Overlapping Intervals

Problem statement:

Given a set **A** of n intervals, compute smallest set **B** of intervals so that

for every interval **I** in **A\B**, there is some interval in **B** which overlaps/intersects with **I**.

A



Overlapping Intervals

Problem statement:

Given a set **A** of n intervals, compute smallest set **B** of intervals so that

for every interval **I** in **A\B**, there is some interval in **B** which overlaps/intersects with **I**.

another optimal solution ☺

Overlapping Intervals

Strategy 1

Interval with **maximum length** should be there in optimal solution

Intuition:

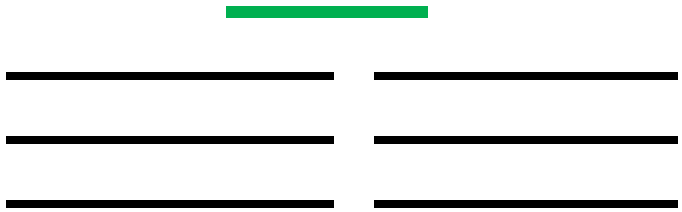
Selecting such an interval will **cover maximum** no. of other intervals

There is a counter example ☹

Overlapping Intervals

Strategy 1

Interval with **maximum length** should be there in optimal solution



Intuition:

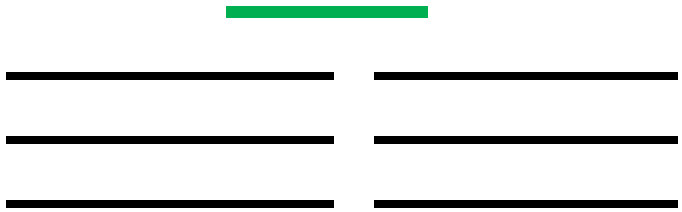
Selecting such an interval will **cover maximum** no. of other intervals

There is a counter example ☹

Overlapping Intervals

Strategy 2

Interval that **overlaps maximum no.** of intervals should be there in optimal solution



Intuition:

Selecting such an interval will **cover maximum** no. of other intervals

There is a counter example ☹

Overlapping Intervals

Strategy 2

Interval that **<u>overlaps maximum no.</u>** of intervals should be there in optimal solution

Overlapping Intervals

Strategy 2

Interval that **overlaps maximum no.** of intervals should be there in optimal solution

Not an optimal solution ☹

Overlapping Intervals

Strategy 2

Interval that **overlaps maximum no.** of intervals should be there in optimal solution

An optimal solution has size 2.

Think for a while :

After failure of two strategies, how to proceed to design the algorithm.

Overlapping Intervals

Let I* be the interval with earliest finish time.
Let I' be the interval with **maximum** finish time overlapping I*.

Lemma1: There is an optimal solution for set **A** that contains **I'.**

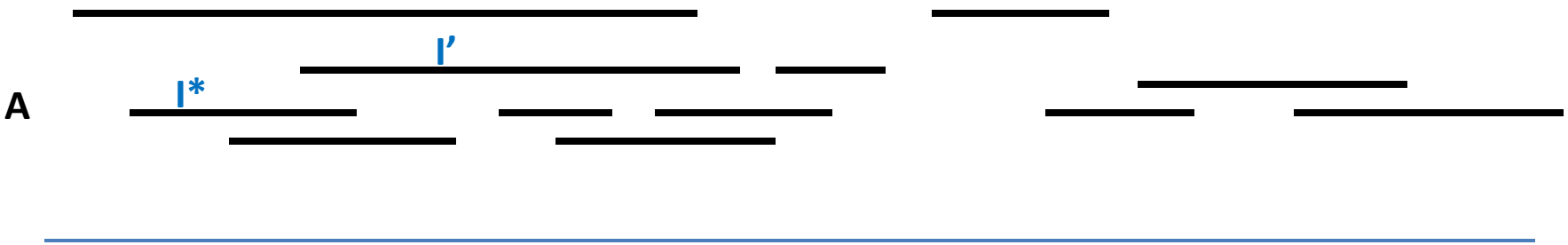
Proof:(sketch) :

If I* is overlapped by any other interval in the optimal solution, say I^,
I' will surely overlap all intervals that are overlapped by I^.
➔ Swapping I^ by I' will still give an optimal solution.

Exploit the fact that I* has earliest finish time for this claim.

Overlapping Intervals

Question: How to obtain smaller instance **A'** using **Lemma 1** ?

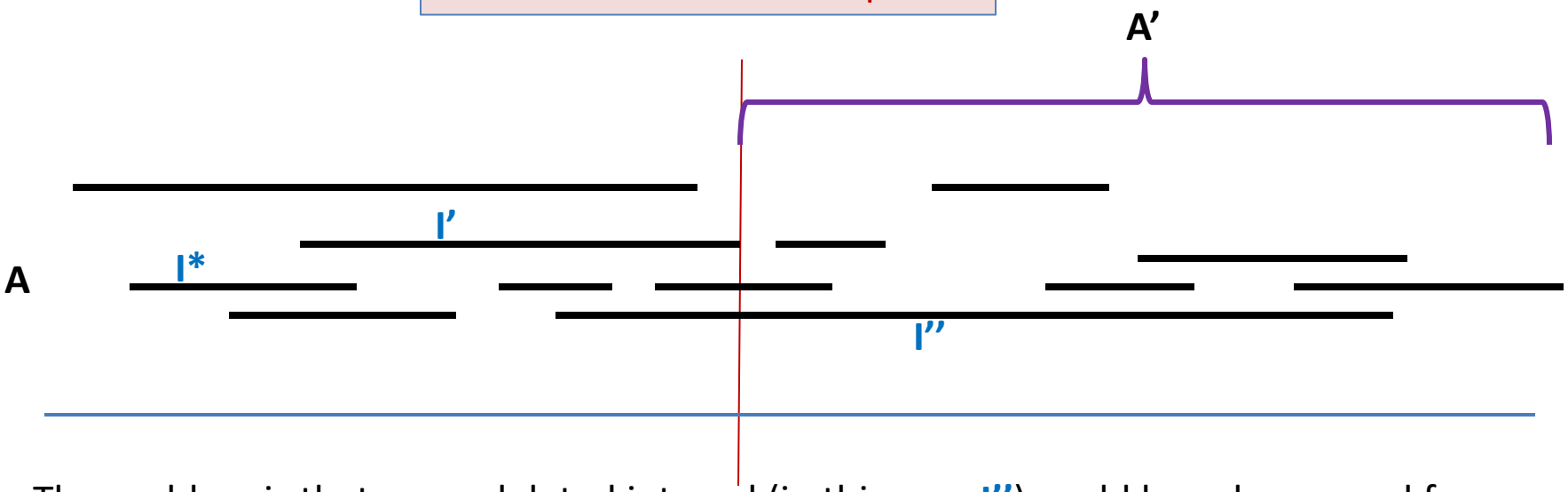


Overlapping Intervals

Question: How to obtain smaller instance **A'** using **Lemma 1** ?

Naive approach : remove from **A** all intervals which overlap with **I'**. This is **A'**.

There is a counter example ☹



The problem is that some deleted interval (in this case **I''**) could have been used for intersecting many intervals if it were not deleted. But deleting it from the instance disallows it to be selected in the solution.

Homework

- How will you form the smaller instance ?
- Design an algorithm for the problem.
- Give a neat, concise, and formal proof of correctness of the algorithm.

Data Structures and Algorithms

(ESO207)

Lecture 38

- An interesting problem:

shortest path from a source to destination

SHORTEST PATHS IN A GRAPH

A fundamental problem

Notations and Terminologies

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

- ω: $E \rightarrow R^+$
- 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 u to v of minimum length is called the **shortest path** from u to v

Definition: **Distance** from u to v is the **length** of the shortest path from u to v.

Notations:

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

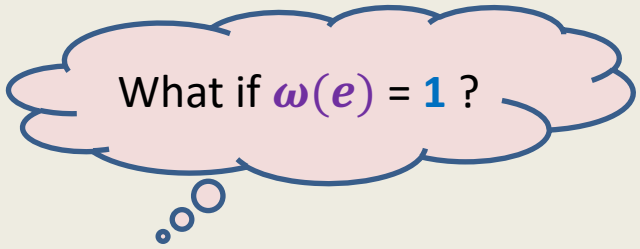
$P(u, v)$: The shortest path from u to v.

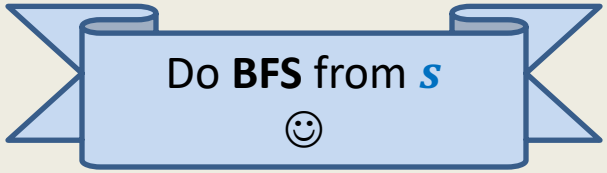
Problem Definition

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 $v \in V \backslash \{s\}$





Problem Definition

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

Aim:

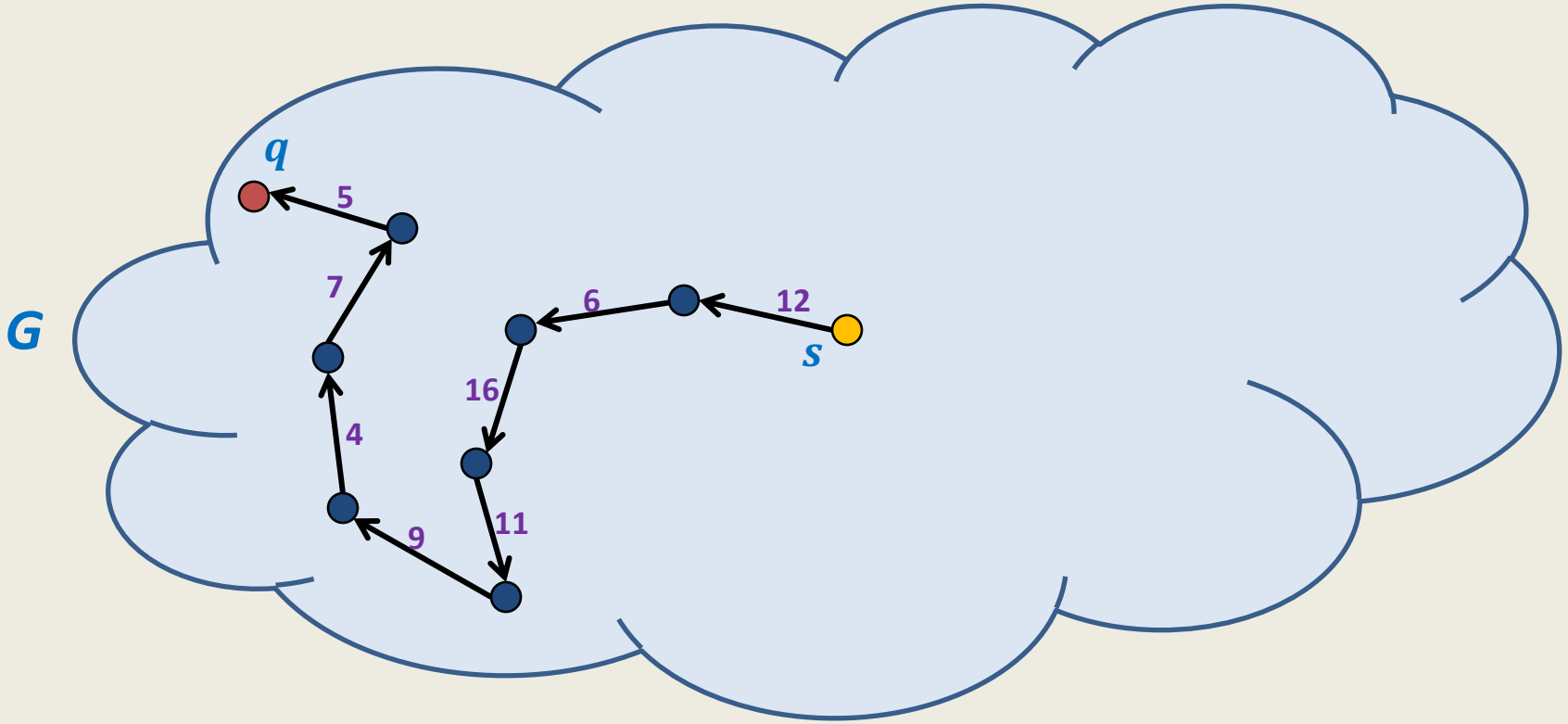
- Compute $\delta(s, v)$ for all $v \in V \backslash \{s\}$
- Compute $P(s, v)$ for all $v \in V \backslash \{s\}$

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 !

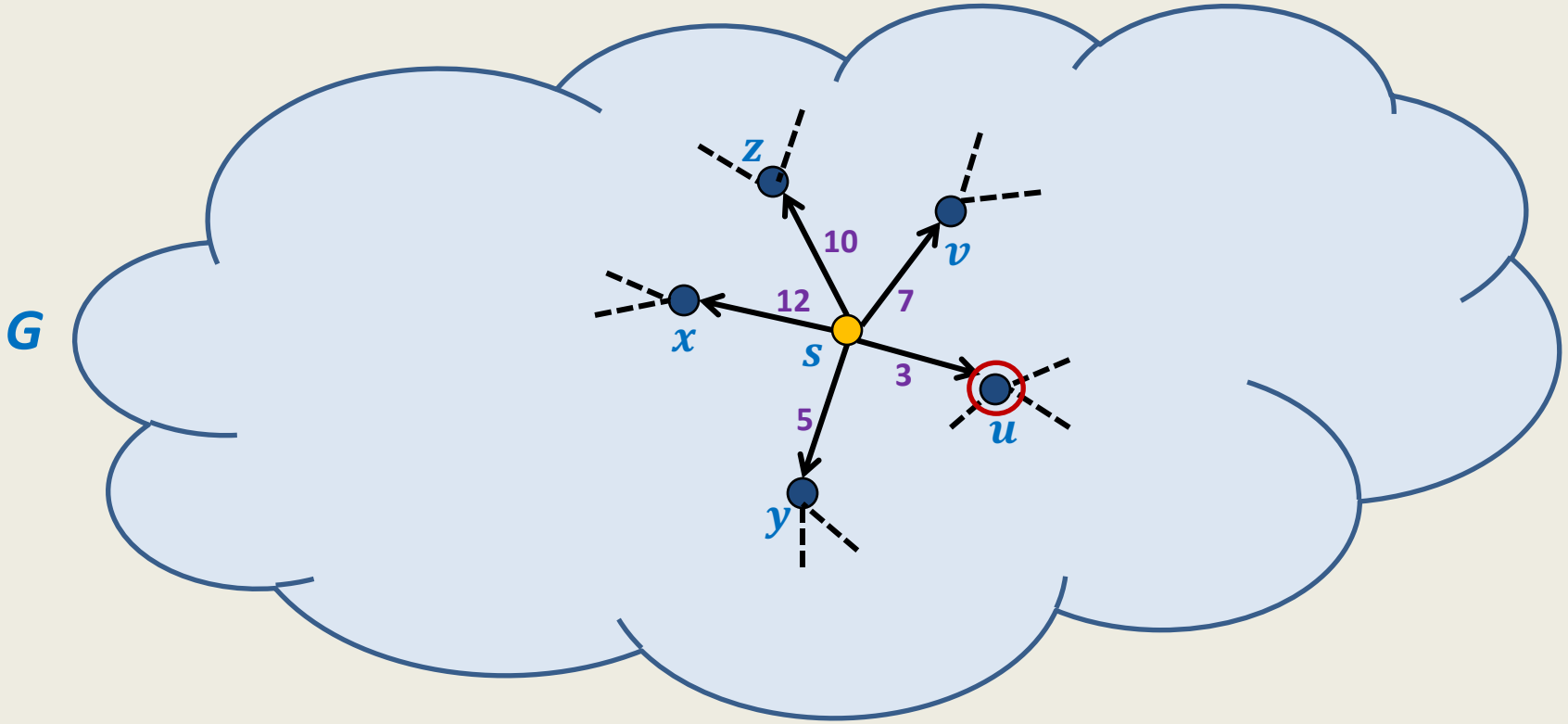
An example to get an insight into this problem



Inference:

The distance to any vertex depends upon global parameters.

An example to get an insight into this problem

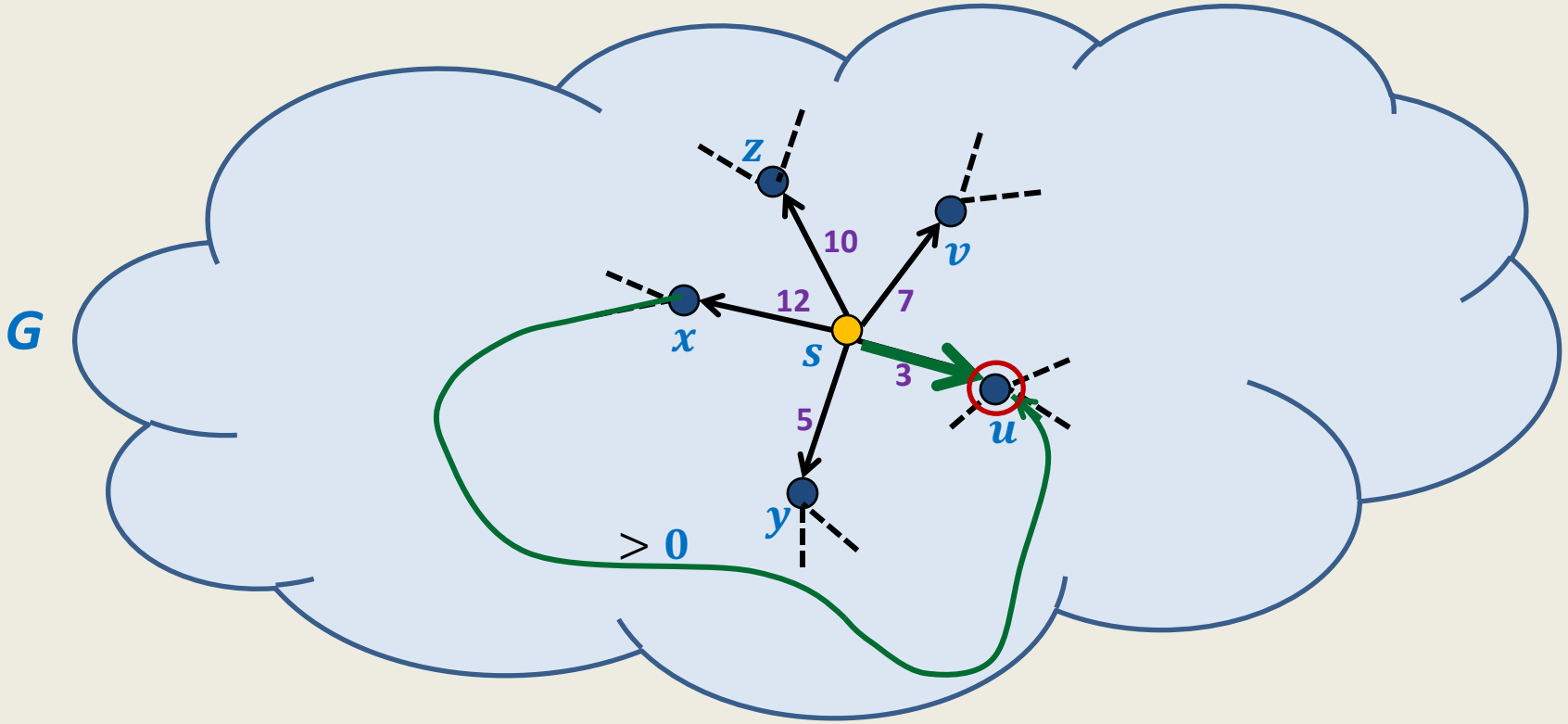


Question: Is there any vertex in this picture for which you are certain about the distance from s ?

Answer: vertex u.

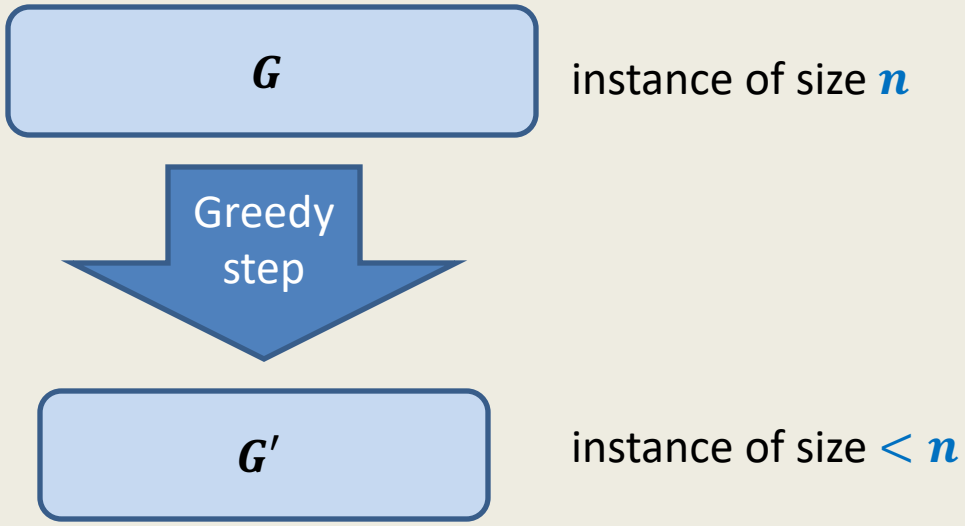
Give reasons.

An example to get an insight into this problem



➔ The shortest path to vertex u is edge (s,u).

Designing a greedy algorithm for shortest paths

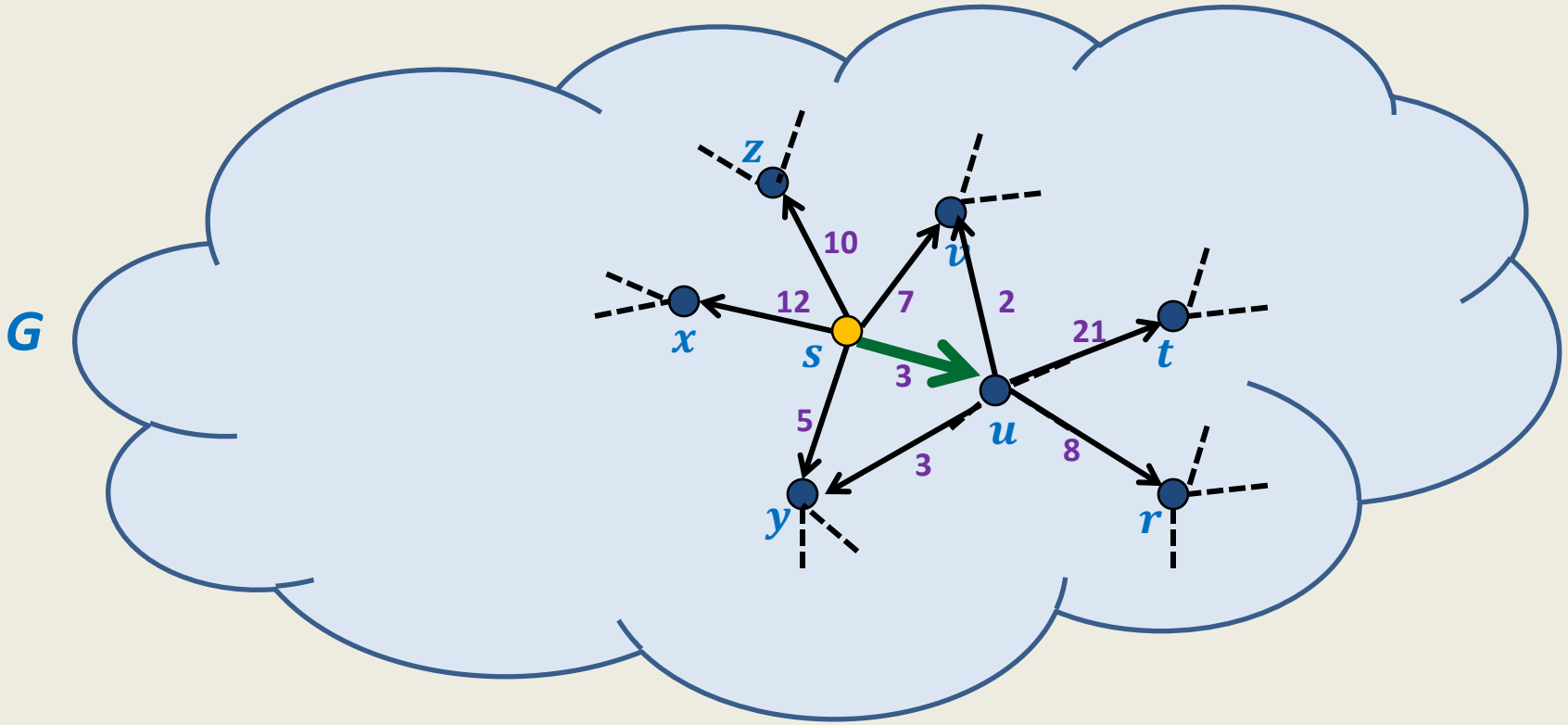


1. **Establish** a relation between

and

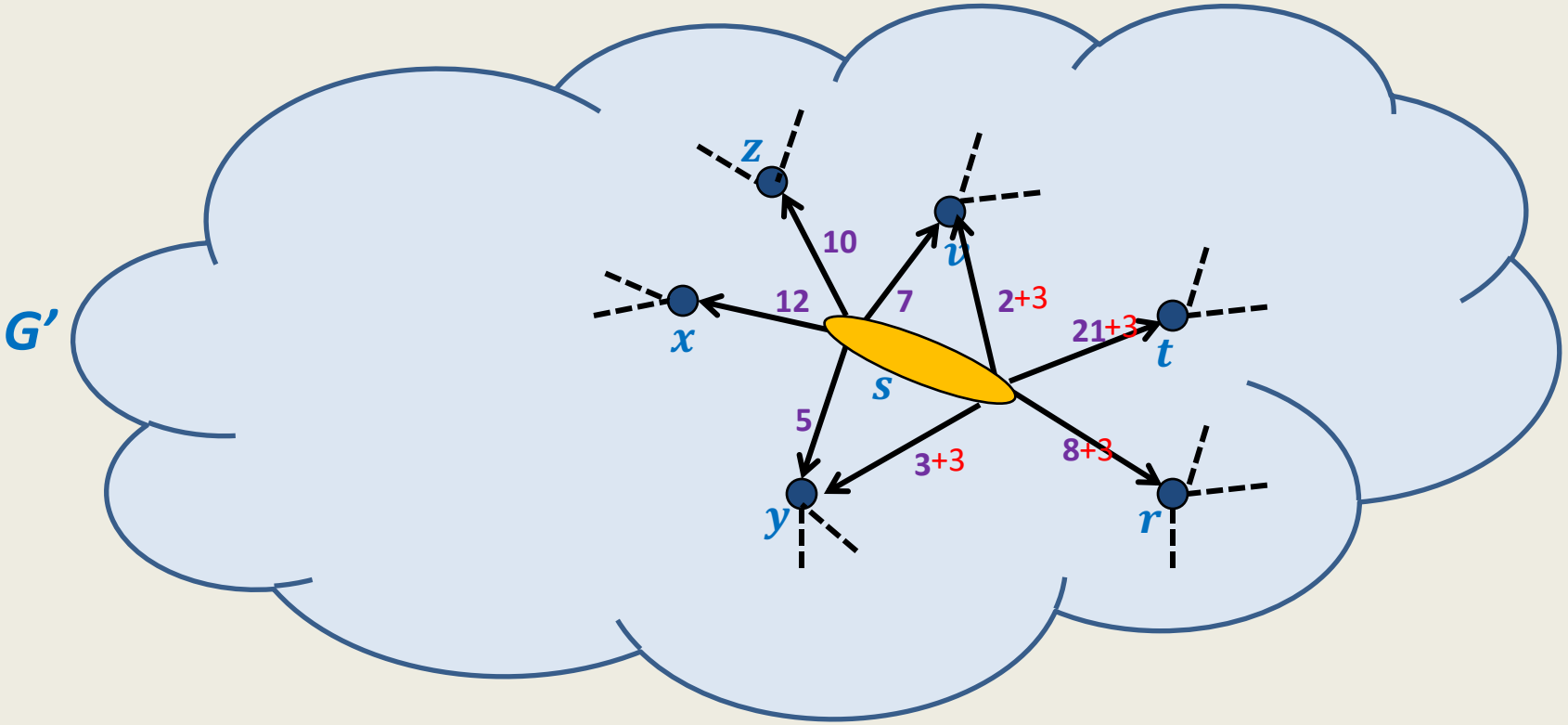
shortest paths in G'

An example to get an insight into this problem

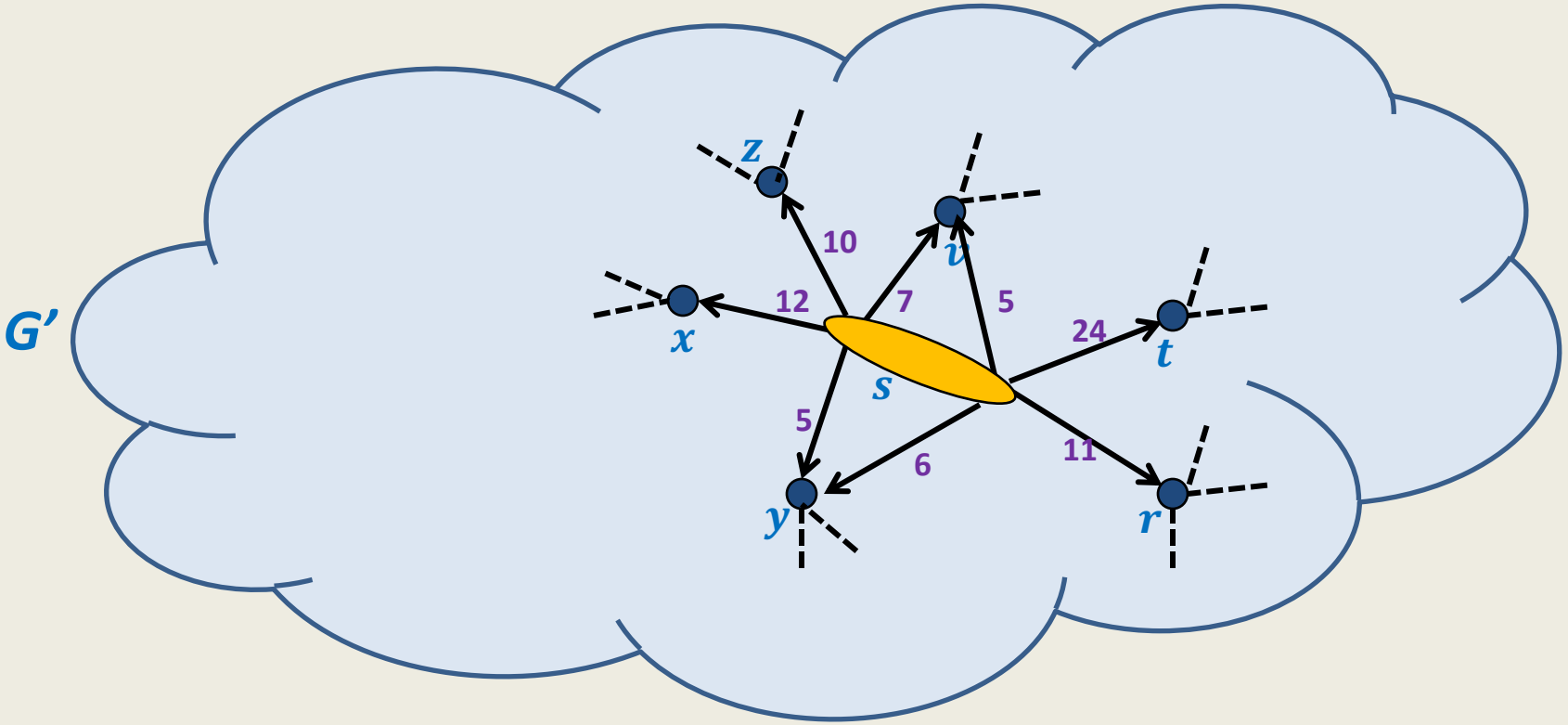


Question: Can you remove vertex u without affecting the distance from s ?

An example to get an insight into this problem



An example to get an insight into this problem



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

3. Remove vertex u.

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

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

➔ an algorithm for **distances** from s

Can you see some negative points of this algorithm ?

Shortcomings of the algorithm

- **No insight** into the (beautiful) structure of shortest paths.
- **Just convinces** that we can solve the shortest paths problem in polynomial time.
- **Very few options** to improve the time complexity.

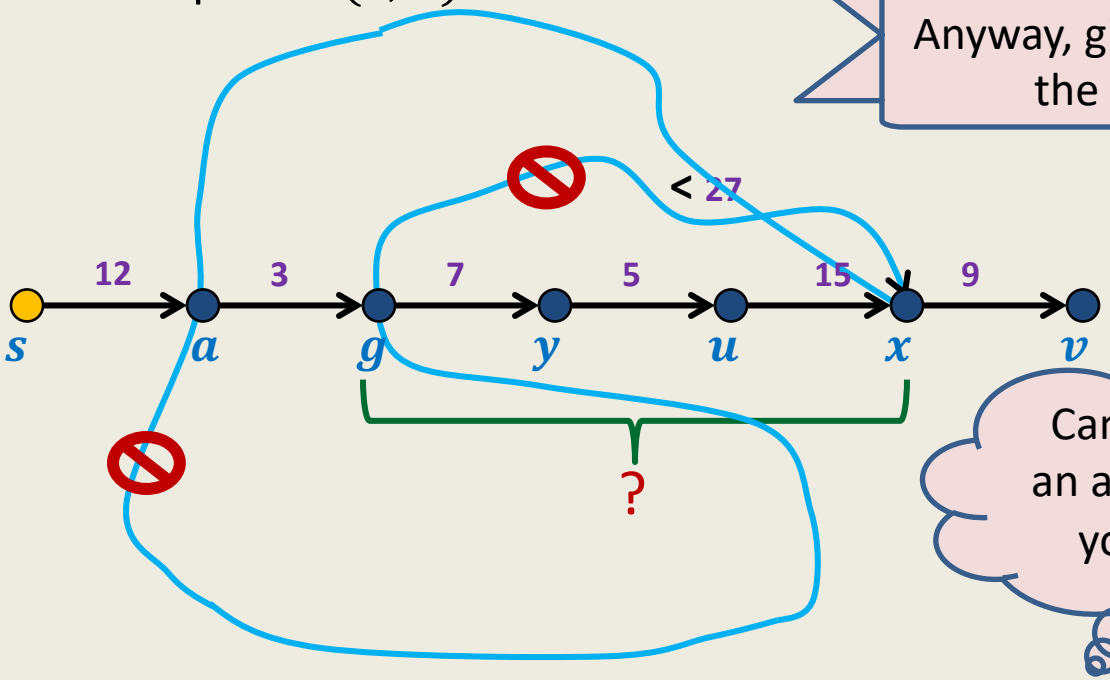
We shall now design a very **insightful** algorithm based on **properties** of shortest paths.

PROPERTY OF A SHORTEST PATH

Optimal subpath property

Consider any shortest path $P(s, v)$.

$\delta(s, v) = 51$



Anyway, give a proof of the lemma

Can you realize an assumption in your proof ?

Lemma 1: Every **subpath** of a shortest path is also a shortest path.

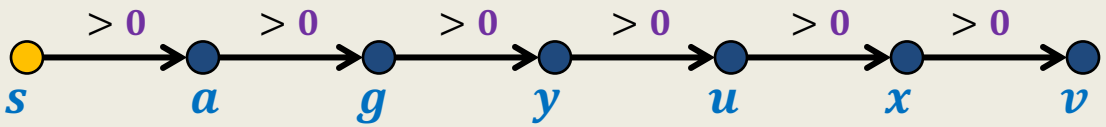
NOTE: Does the lemma use the fact that the edge weights are positive?

If yes, can you locate the exact place where it used it?

Homework: Write a complete and formal proof for **Lemma 1**

Exploiting the positive weight on edges

Consider once again a shortest path $P(s, v)$.



➔ $\delta(s,a) < \delta(s,g) < \delta(s,y) < \delta(s,u) < \delta(s,x) < \delta(s,v)$ (1)

➔ The first nearest vertex of s must be its **neighbor**.

More insights …

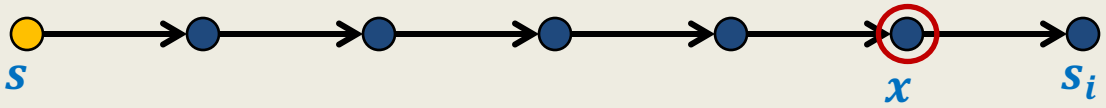
Let s_i :

$s_0 = s$.

What can we say about x ?

x must be s_j for some $j < i$.

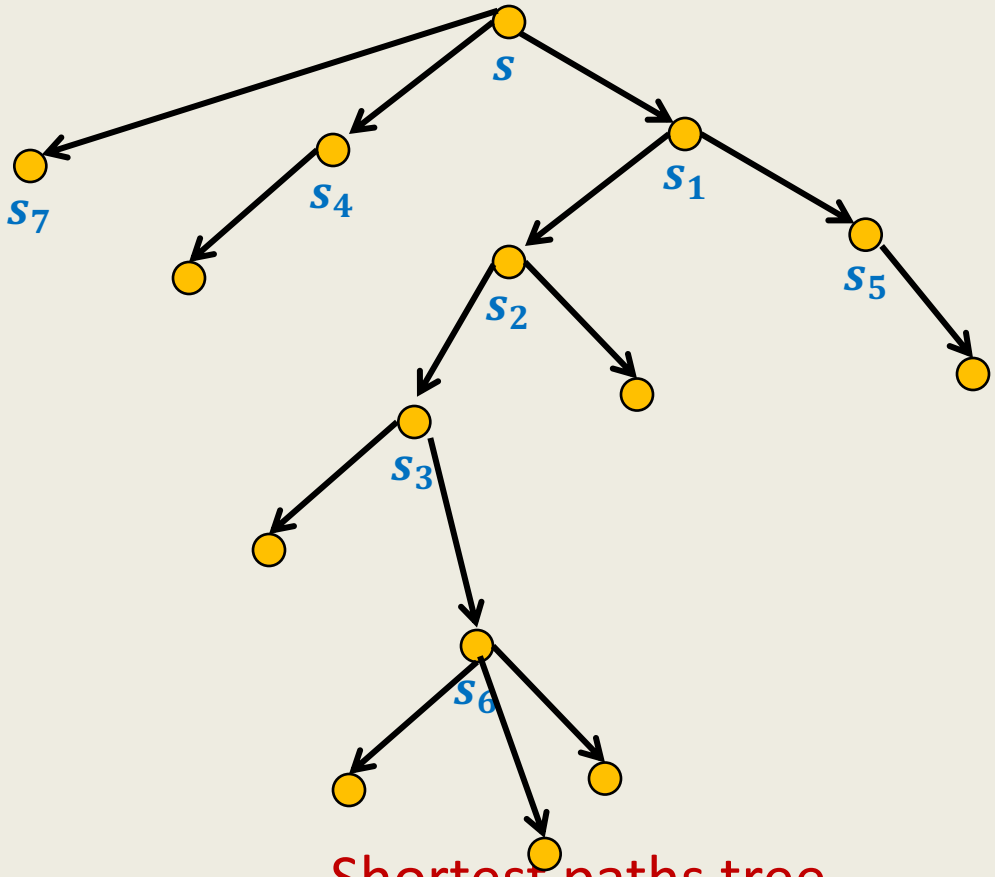
Consider the shortest path $P(s, s_i)$.



Lemma 2: $P(s, s_i)$ must be of the form

What picture captures all shortest paths from s?

Complete picture of all shortest paths ?



Shortest paths tree

Designing the algorithm …

Lemma 2: $P(s, s_i)$ must be of the form $s \rightsquigarrow s_j \rightarrow s_i$ for some $j < i$.

shortest

Question: Can we use **Lemma 2** to design an algorithm ?

Incremental way to compute shortest paths.

Ponder over it before going to the next slide ☺

Designing the algorithm …

Lemma 2: $P(s, s_i)$ must be of the form $s \rightsquigarrow s_j \rightarrow s_i$ for some $j < i$.

shortest

Question: Can we use **Lemma 2** to design an algorithm ?

Incremental way to compute shortest paths.

The next slide explains it precisely.

If shortest paths to s_j, $j < i$ is known, we can compute s_i.

But how ?

All we know is that

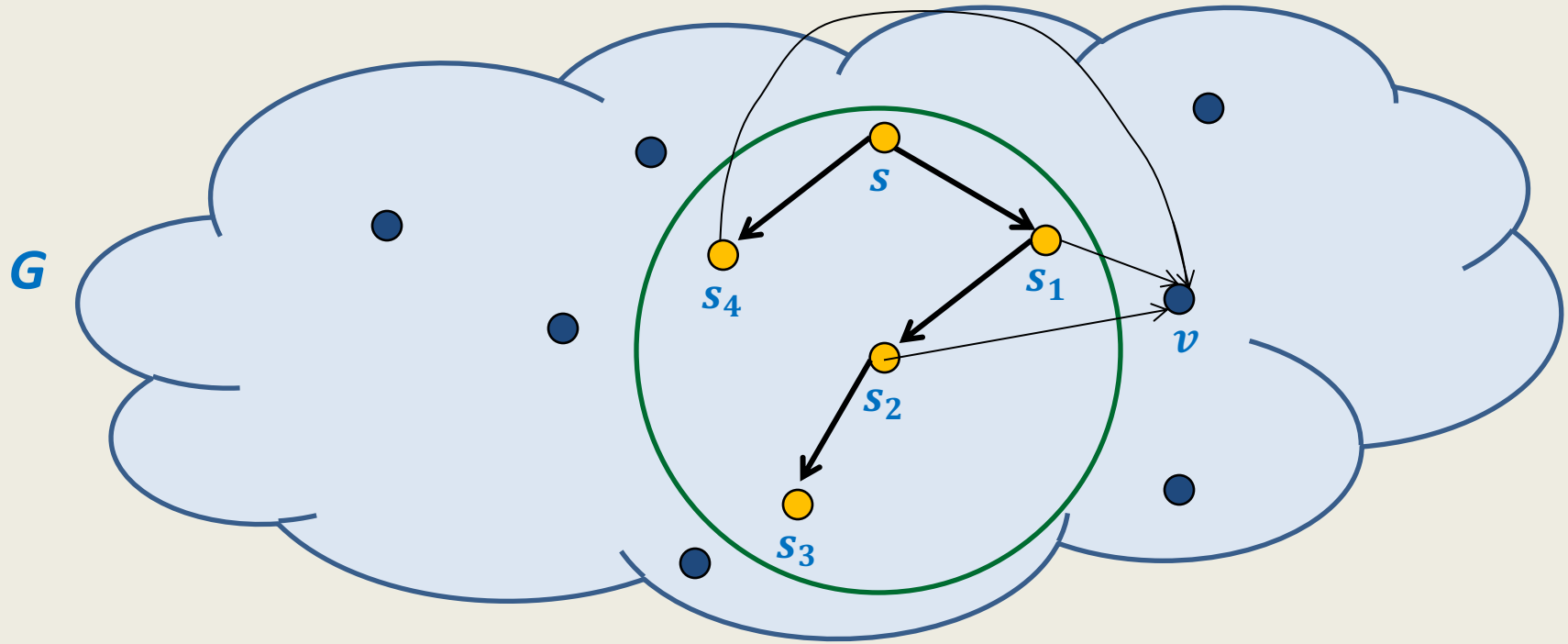
But which neighbor ?

Hint:

For each neighbor, compute some *label* based on **Lemma 2** s.t. s_i is the neighbour with least label.

Suppose we have computed $s_1, s_2, \ldots, s_{i-1}$.

We can compute s_i as follows.



For each $v \in V \setminus \{s_1, s_2, \ldots, s_{i-1}\}$

$$L(v) = \min_{(s_j, v) \in E)} \left(\delta(s, s_j) + \omega(s_j, v)\right)$$

s_i is the vertex with **minimum** value of L.

Dijkstra's algorithm

Dijkstra-algo(s,G)

{ $U \leftarrow V \setminus \{s\}$;

$S \leftarrow \{s\}$;

For i = 1 to $n-1$ do

{ **For each** $v \in U$ do

{ $L(v) \leftarrow \infty$;

For each $(x,v) \in E$ with $x \in S$ do

$L(v) \leftarrow$ **min**($L(v)$, $\delta(s,x) + \omega(x,v)$)

}

$y \leftarrow$ vertex from U with **minimum** value of L;

$\delta(s,y) \leftarrow L(y)$;

move y from U to S;

}

In this algorithm, we first compute $L(v)$ for each $v \in U$ and then find the vertex with the least L value.

Try to rearrange its statements so that in the beginning of each iteration, we have L values **computed already**.

This rearrangement will be helpful for improving the running time.

So please try it on your own first before viewing the next slide.

Dijkstra's algorithm

Dijkstra-algo(s,G)

{ $U \leftarrow V$; $L(v) \leftarrow \infty$ for all $v \in U$;

$L(s) \leftarrow 0$;

For i = 0 to $n-1$ do

{ $y \leftarrow$ vertex from U with **minimum** value of L;

$\delta(s,y) \leftarrow L(y)$;

move y from U to S;

For each $v \in U$ do

{ $L(v) \leftarrow \infty$;

For each $(x,v) \in E$ with $x \in S$ do

$L(v) \leftarrow$ **min**($L(v)$, $\delta(s,x) + \omega(x,v)$)

}

}

a lot of **re-computation**

Dijkstra's algorithm

Dijkstra-algo(s,G)

{ $U \leftarrow V$; $L(v) \leftarrow \infty$ for all $v \in U$;

$L(s) \leftarrow 0$;

For i = 0 to $n-1$ do

{ $y \leftarrow$ vertex from U with **minimum** value of L;

$\delta(s,y) \leftarrow L(y)$;

move y from U to S;

For each $v \in U$ do

{

For each $(x,v) \in E$ with $x \in S$ do

$L(v) \leftarrow$ **min**($L(v)$, $\delta(s,x) + \omega(x,v)$)

}

}

Only neighbors of y

What are the vertices whose L value may change in this iteration ?

Dijkstra's algorithm

Dijkstra-algo(s,G)

{ $U \leftarrow V$; $L(v) \leftarrow \infty$ for all $v \in U$;

$L(s) \leftarrow 0$;

For i = 0 to $n-1$ do

{ $y \leftarrow$ vertex from U with **minimum** value of L;

$\delta(s,y) \leftarrow L(y)$;

move y from U to S;

For each $(y,v) \in E$ with $v \in U$ do

{

$L(v) \leftarrow$ **min**($L(v)$, $\delta(s,y) + \omega(y,v)$)

}

}

1 **extract-min** operation

deg(y) **Decrease-key** operations

Time complexity of Dijkstra's algorithm

Total number of **extract-min** operation : n

Total **Decrease-key** operations : m

Using **Binary heap** to maintain the set U, the time complexity: O(m log n)

Theorem: Given a directed graph with positive weights on edges, we can compute all shortest paths from a given vertex in O(m log n) time.

Fibonacci heap supports **Decrease-key** in O(1) time and **extract-min** in O(log n).

➔ Total time complexity using Fibonacci heap: O(m + n log n)

Data Structures and Algorithms

(ESO207)

Lecture 39

- Integer sorting
- Counting Sort and Radix Sort

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 (we will not prove it in this course):

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)

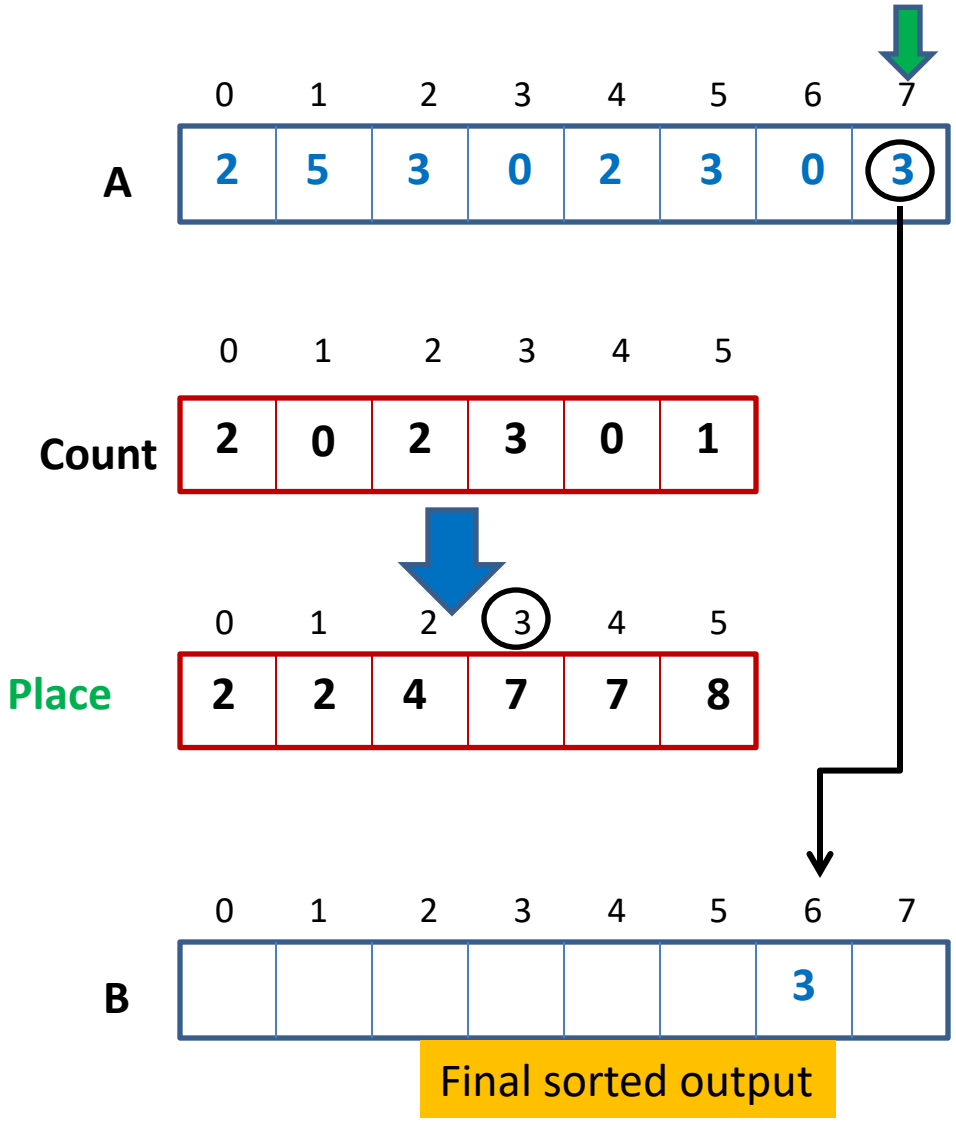
Motivating example: Indian railways

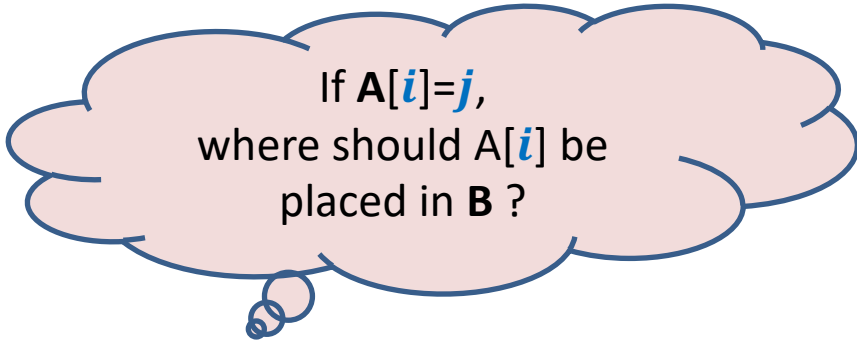
There are **13 lacs** employees.

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

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

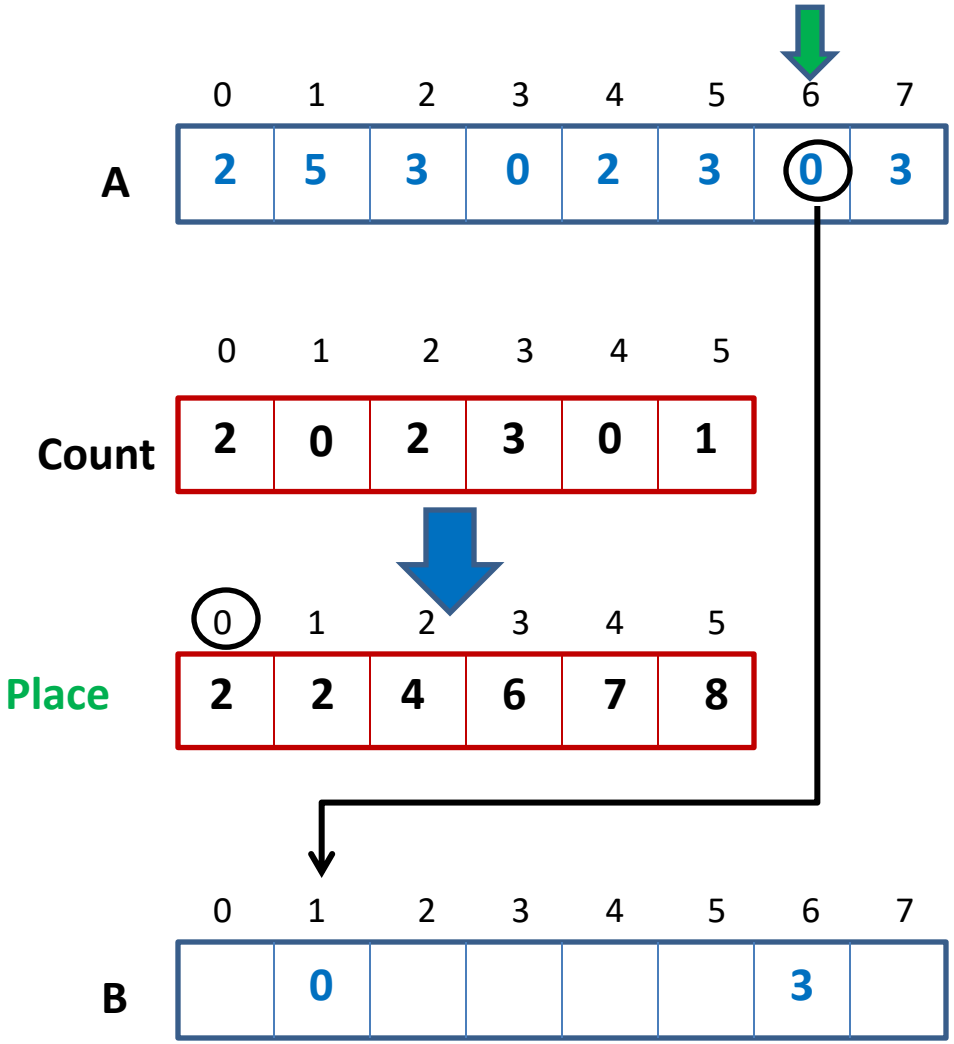
Counting sort: algorithm for sorting integers



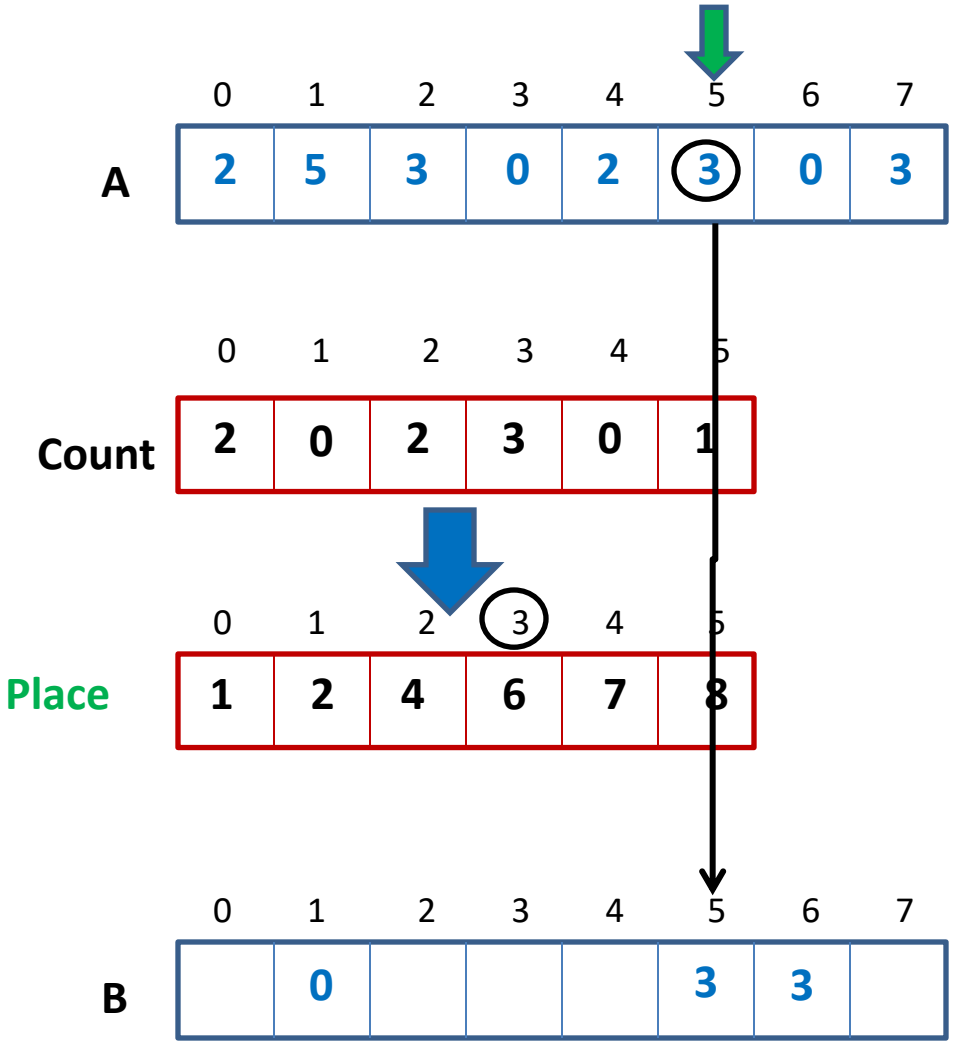


Certainly after all those elements in **A** which are **smaller** than j

Counting sort: algorithm for sorting integers



Counting sort: algorithm for sorting integers



Types of sorting algorithms

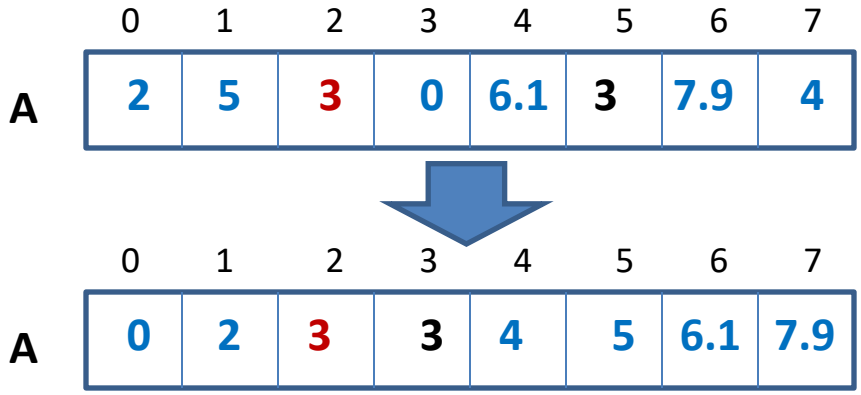
In Place Sorting algorithm:

A sorting algorithm which uses

Example: Heap sort, Quick sort.

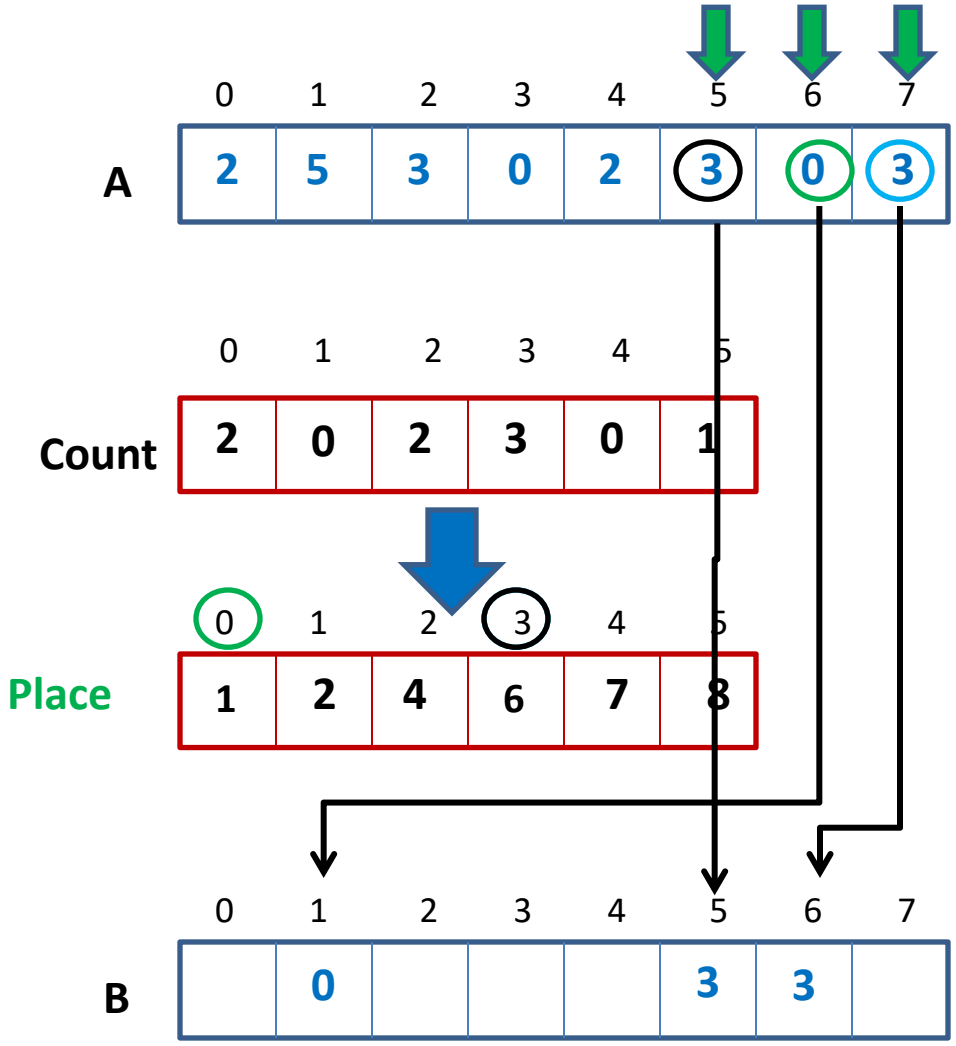
Stable Sorting algorithm:

A sorting algorithm which preserves



Example: Merge sort.

Counting sort: a visual description



Why did we scan elements of **A** in reverse order (from index $n-1$ to 0) while placing them in the final sorted array **B** ?

Answer:

1 To ensure that Counting sort is **stable**.

t The reason why stability is required will

t become clear soon ☺

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;

Place[0]← Count[0];

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

Key points of Counting sort:

- It performs arithmetic operations involving **O**(**log** n + **log** k) bits
  (**O**(1) time in **word RAM**).
- It is a **stable** sorting algorithm.

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

➔ **For** $k \leq n$,

➔ For $k = n^t$,

(too bad for $t > 1$. ☹)

Question:

How to sort n integers in the range $[0..n^t]$ in

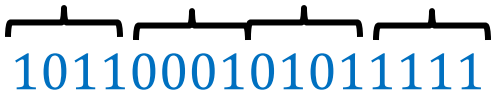
Radix Sort

Digits of an integer

507266

No. of **digits** = 6

value of **digit** $\in \{0, \dots, 9\}$



No. of **digits** = 4

value of **digit** $\in \{0, \dots, 15\}$

It is up to us how we define digit ?

Radix Sort

Input: An array **A** storing n integers, where

(i) each integer has exactly d **digits**.

(ii) each **digit** has **value** $< k$

(iii) $k < n$.

Output: Sorted array **A**.

Running time:

O(dn) in **word RAM** model of computation.

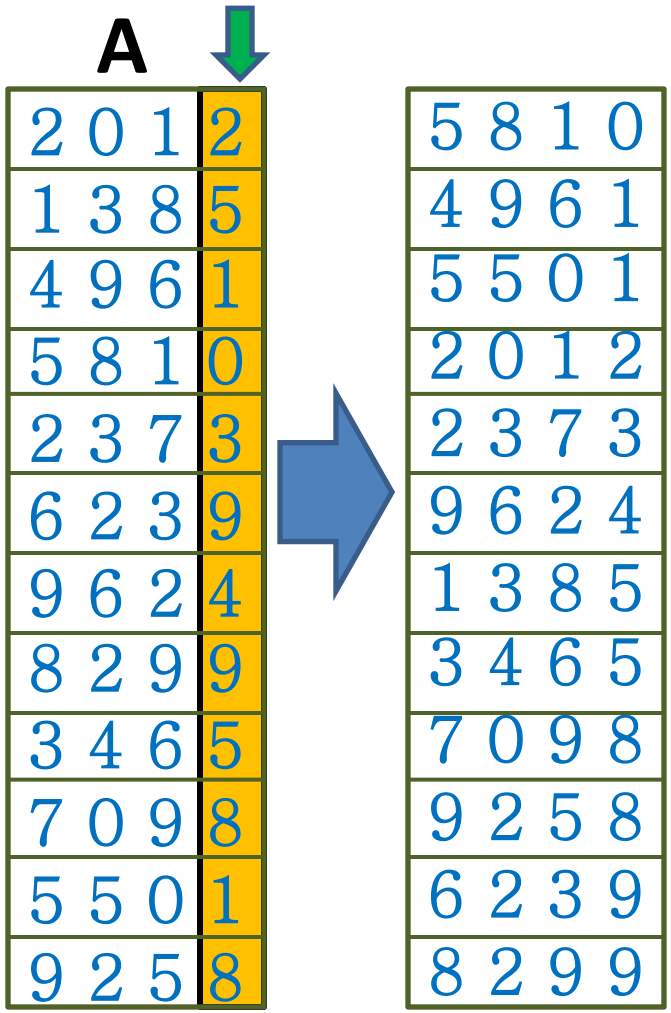
Extra space:

O($n + k$)

Important points:

- makes use of a **count sort.**
- Heavily relies on the fact that **count sort** is a **stable sort** algorithm.

Demonstration of Radix Sort through example

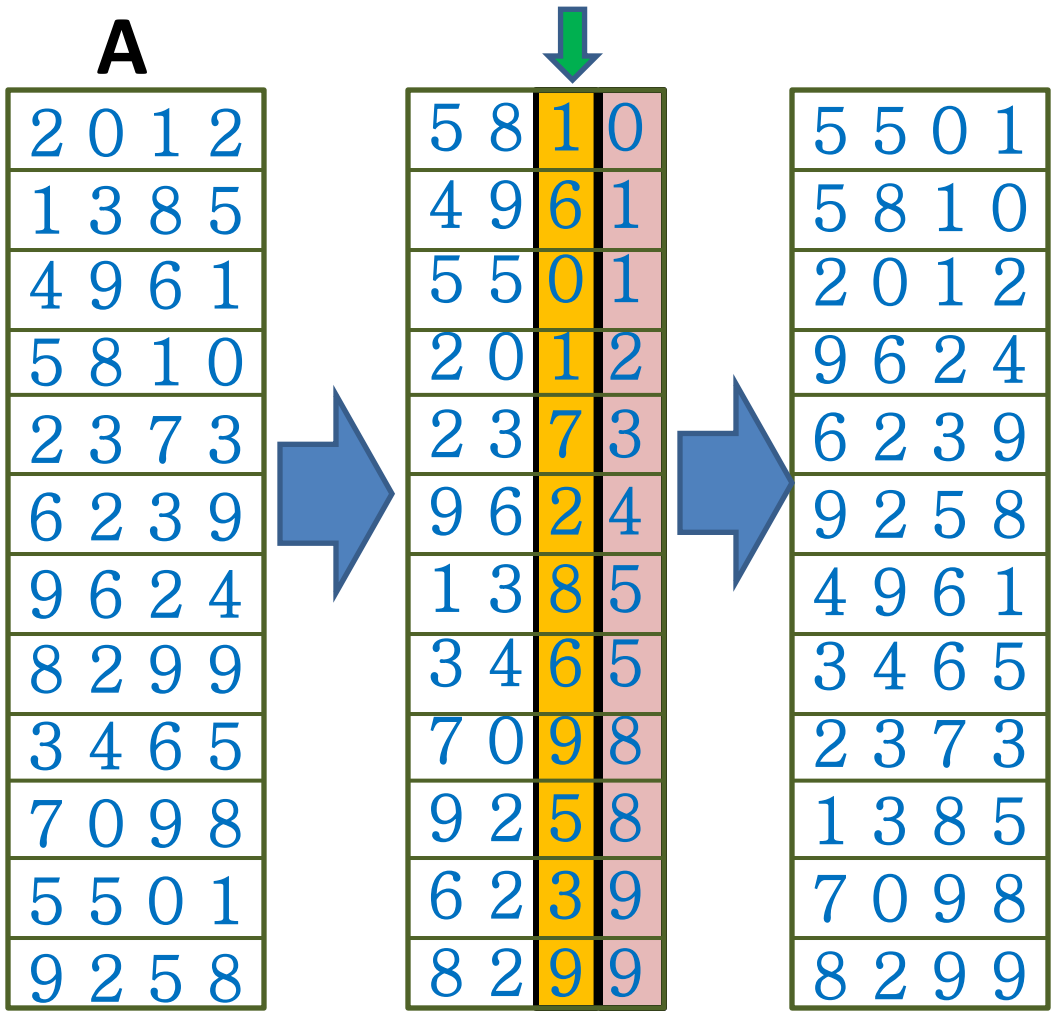


d =4

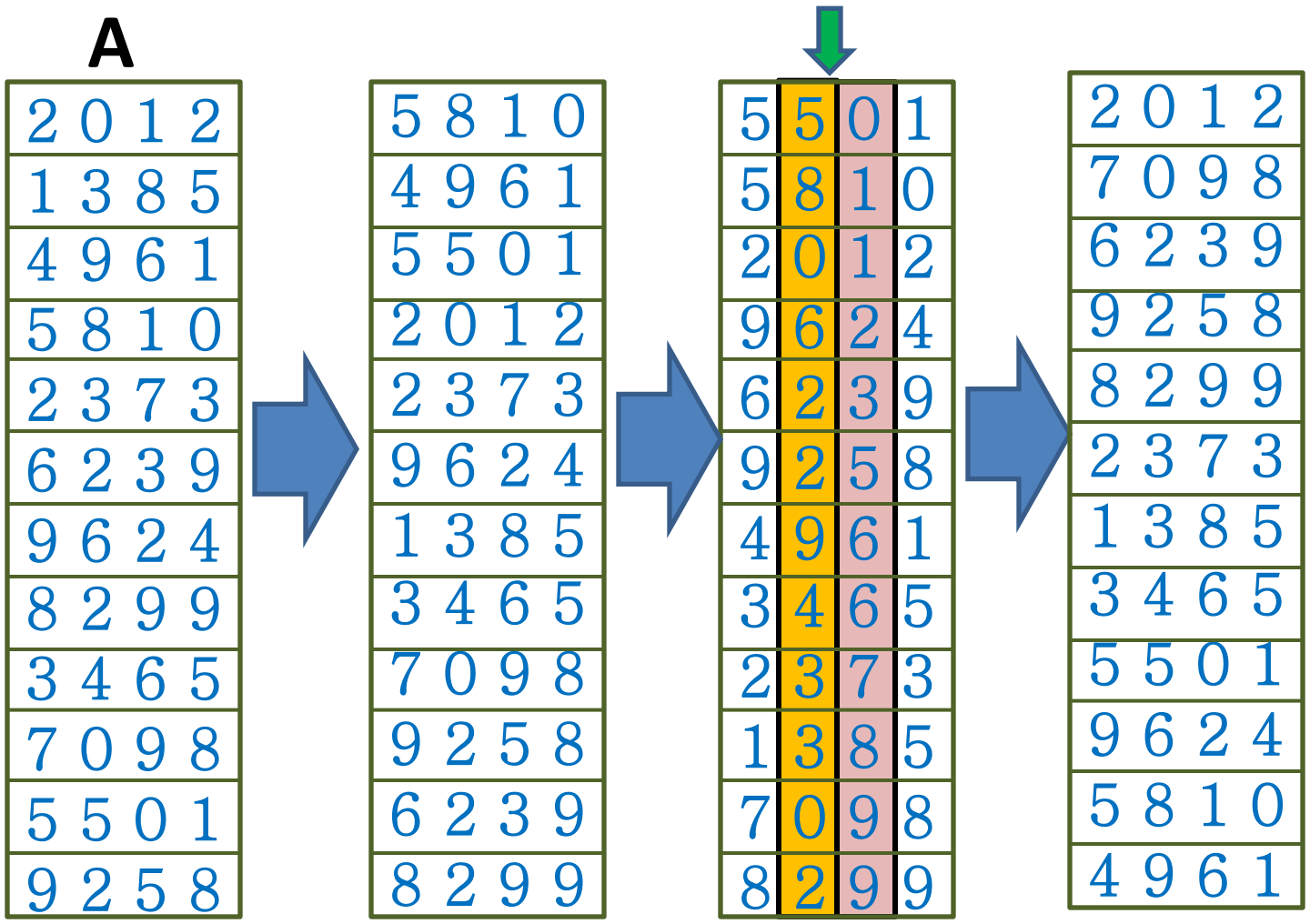
n =12

k =10

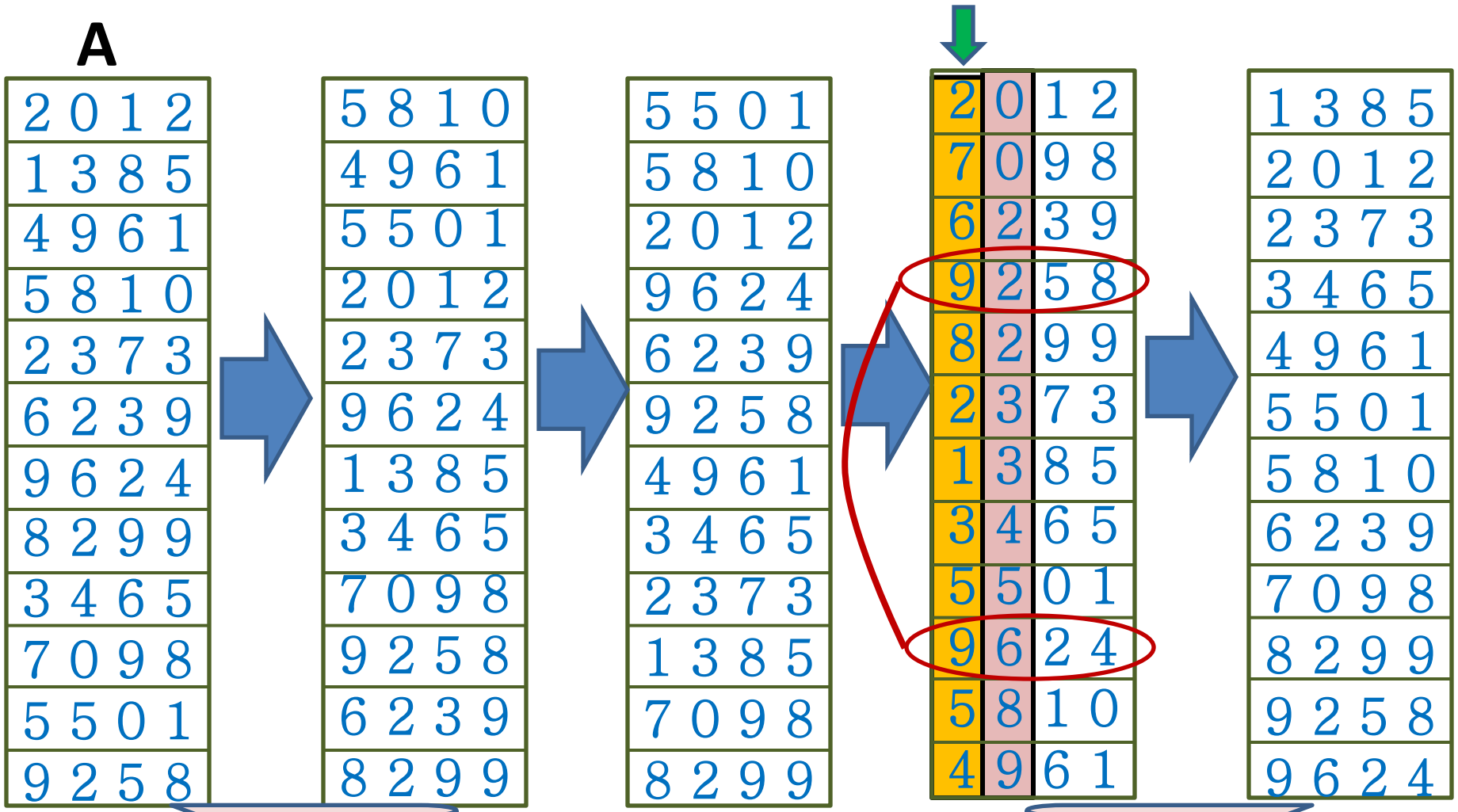
Demonstration of Radix Sort through example



Demonstration of Radix Sort through example



Demonstration of Radix Sort through example



Can you see where we are exploiting the fact that **Countsort** is a **stable** sorting algorithm ?

Radix Sort

RadixSort(A[$0 \ldots n-1$], d, k)

{ For j=1 to d do

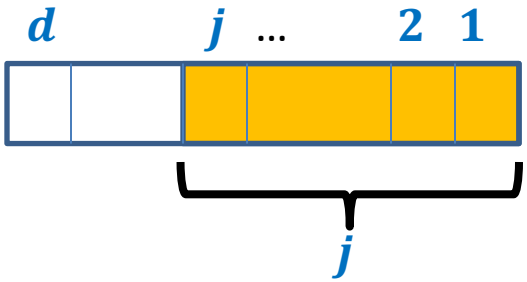
Execute CountSort(A,k) with **jth digit** as the **key;**

return **A;**

}

Correctness:

A number stored in **A** ➔



Inductive assertion:

At the end of **jth** iteration, array **A** is sorted according to the last j digits.

During the induction step, you will have to use the fact that **Countsort** is a **stable** sorting algorithm.

Radix Sort

RadixSort(A[$0 \ldots n-1$], d, k)

{ For j=1 to d do

Execute CountSort(A,k) with jth digit as the key;

return **A;**

}

Time complexity:

- A single execution of **CountSort(A,k)** runs in **O($n+k$) time** and **O($n+k$) space**.
- For $k < n$,

  ➔ a single execution of **CountSort(A,k)** runs in **O(n)** time.

  ➔ Time complexity of radix sort = **O(dn)**.
- ➔Extra space used = **O(n)**

Question: How to use Radix sort to sort n integers in range $[0..n^t]$ in **O(tn)** time and **O(n)** space ?

Answer:

1 bit

log n bits

d	k	Time complexity
$t \log n$	2	O($tn \log n$)
t	n	O(tn)

What digit to use ?

Power of the word RAM model

- **Very fast** algorithms for **sorting integers:**

  Example: n integers in range $[0..n^{10}]$ in **O**(n) time and **O**(n) space ?

- **Lesson:**

  Do not always go after **Merge sort** and **Quick sort** when input is integers.

- **Interesting programming exercise:**

  Compare Quick sort with **Radix sort** for sorting **long** integers.

Data Structures and Algorithms

(ESO207)

Lecture 40

- Search data structure for integers : Hashing

Data structures for searching

in O(1) time

Motivating Example

Input: a given set S of 1009 positive integers

Aim: Data structure for **searching**

Example

{

123, 579236, 1072664, 770832456778, 61784523, 100004503210023, 19,

762354723763099, 579, 72664, 97708324567700123 8, 84, 100004503210023,

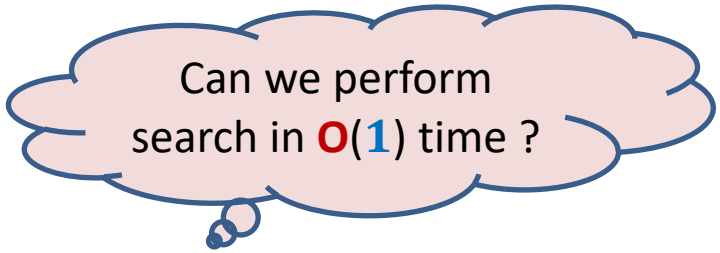
…

}

Data structure : Array storing S in sorted order

Searching : Binary search

O(log $|S|$) time



Problem Description

U :

$S \subseteq U$,

$n = |S|$, $n \ll m$

A search query:

Aim: A data structure for a given set S

that can facilitate search in **O**(**1**) time

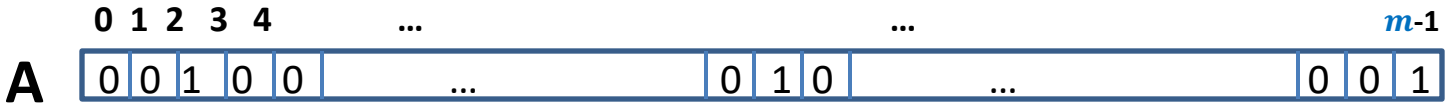
A trivial data structure for O(1) search time

Build a 0-1 array **A** of size m such that

A[i] = 1 **if** $i \in S$.

A[i] = 0 **if** $i \notin S$.

Time complexity for searching an element in set S : **O(1).**



This is a totally Impractical data structure because $n \ll m$!

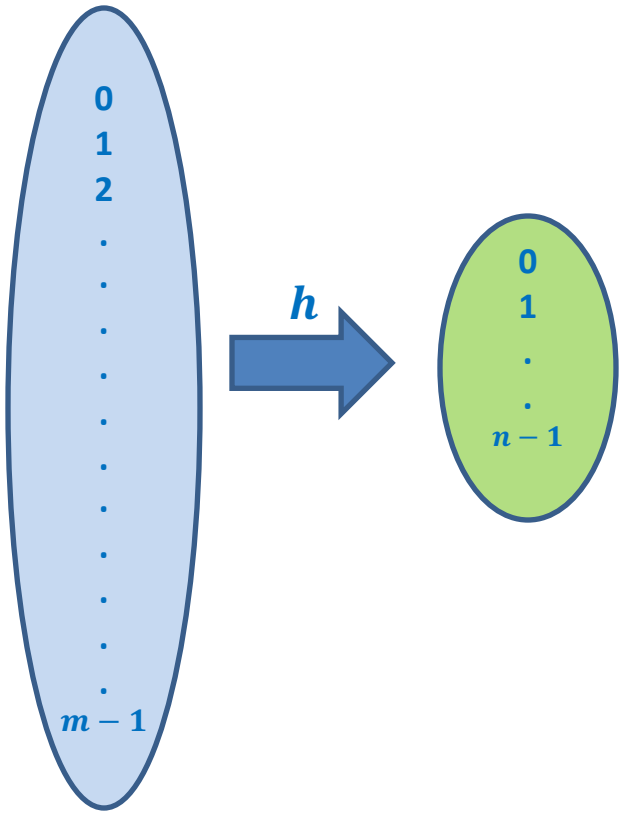
Example: n = few thousands, m = few trillions.

Question:

Can we have a data structure of **O**(n) size that can answer a search query in **O**(1) time ?

Answer: Hashing

Hash function



Hash function:

h is a mapping from U to $\{0,1,\ldots,n-1\}$ with the following characteristics.

- **Space** required for h
- $h(i)$ computable in

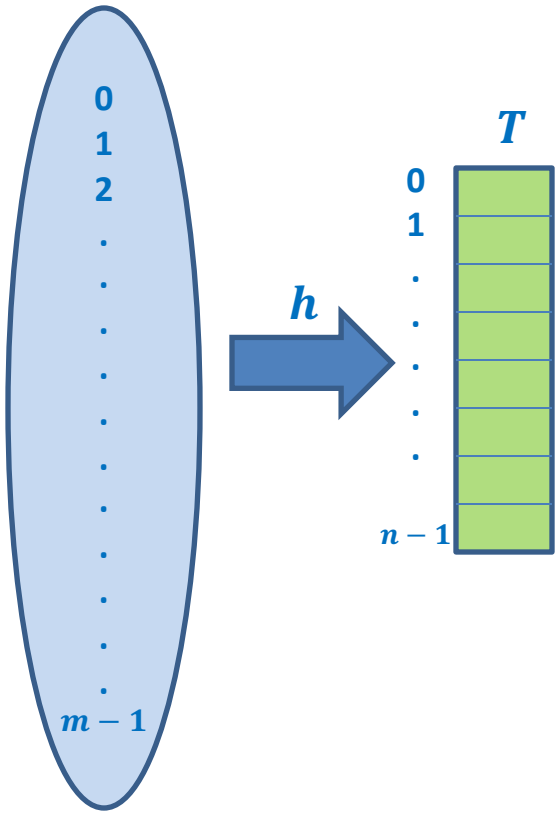
Example:

Hash value:

For a given hash function h, and $i \in U$.

$h(i)$ is called hash value of i

Hash function, hash value



Hash function:

h is a mapping from U to $\{0,1, \dots, n-1\}$

with the following characteristics.

- **Space** required for h : a few **words**.
- $h(i)$ computable in **O(1) time** in **word RAM.**

Example: $h(i) = i \bmod n$

Hash value:

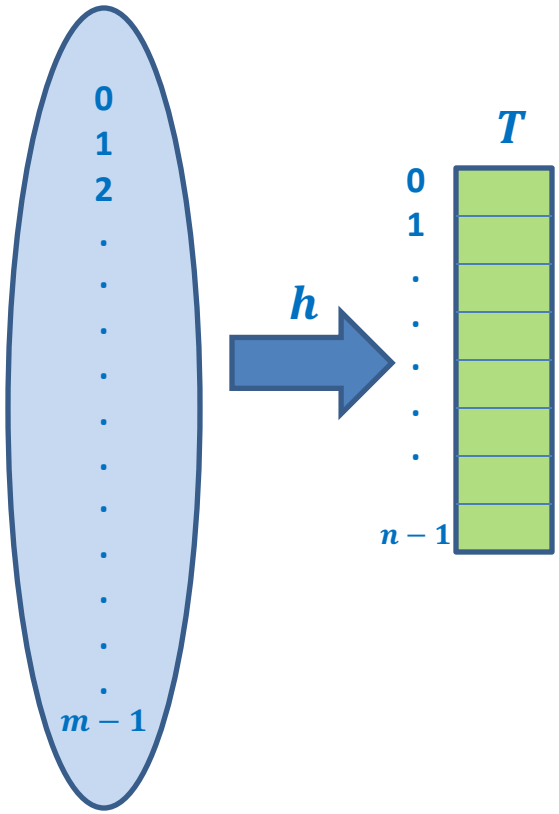
For a given hash function h, and $i \in U$.

$h(i)$ is called hash value of i

Hash Table:

An array $T[0 \dots n-1]$

Hash function, hash value, hash table



Hash function:

h is a mapping from U to $\{0,1,\ldots,n-1\}$

with the following characteristics.

- **Space** required for h : a few **words**.
- $h(i)$ computable in **O(1) time** in **word RAM.**

Example: $h(i) = i \bmod n$

Hash value:

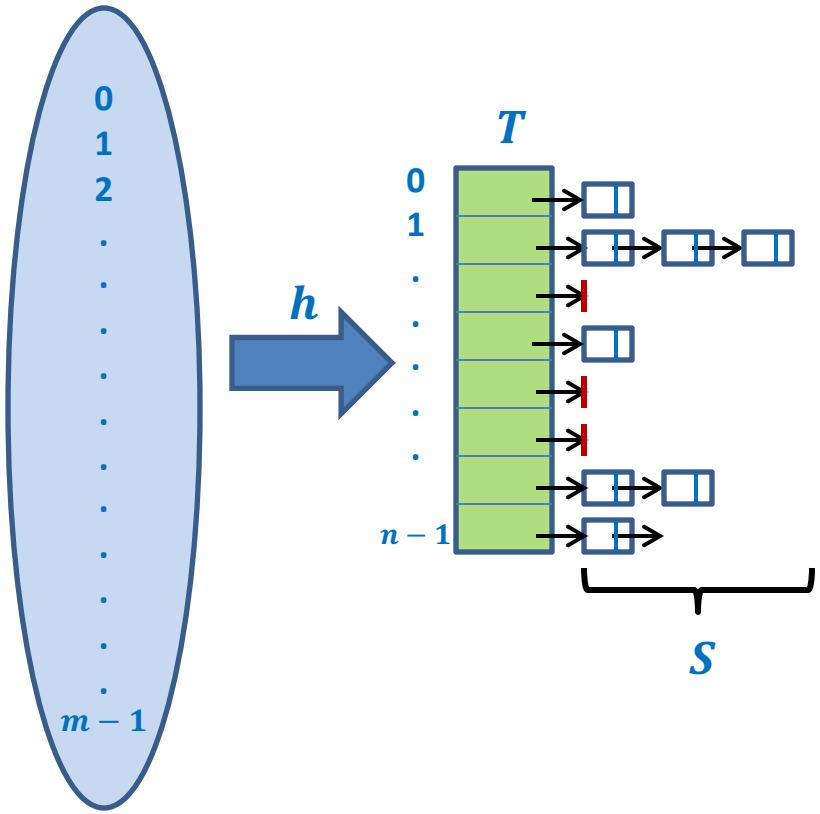
For a given hash function h, and $i \in U$.

$h(i)$ is called hash value of i

Hash Table:

An array $T[0 \ldots n-1]$

Hash function, hash value, hash table



Hash function:

h is a mapping from U to $\{0,1,\ldots,n-1\}$ with the following characteristics.

- **Space** required for h : a few **words**.
- $h(i)$ computable in **O(1) time** in **word RAM.**

Example: $h(i) = i \bmod n$

Hash value:

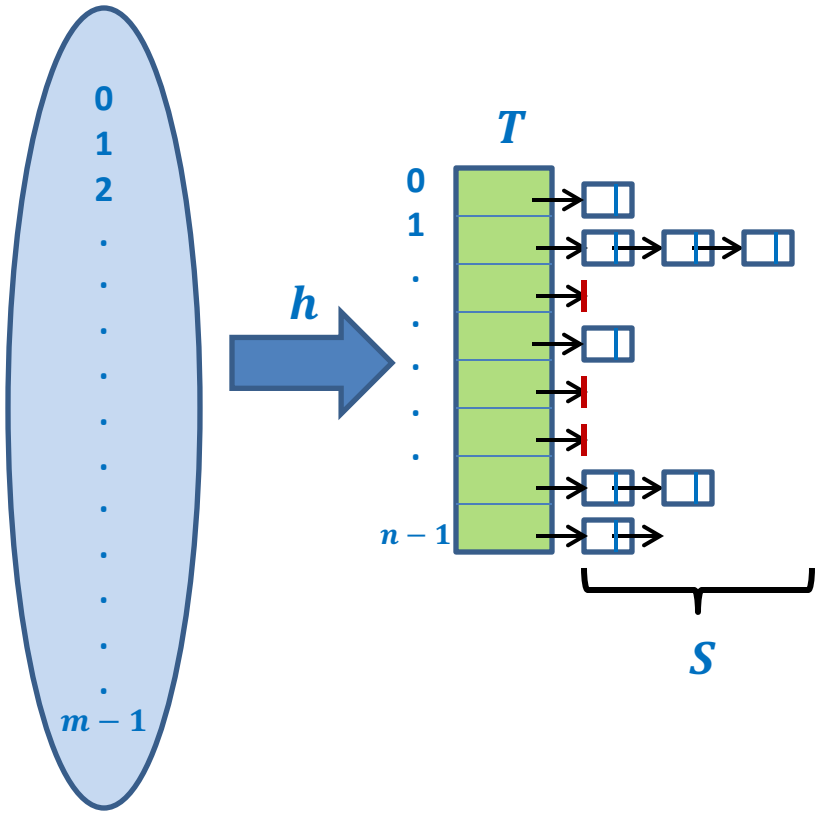
For a given hash function h, and $i \in U$.

$h(i)$ is called hash value of i

Hash Table:

An array $T[0 \ldots n-1]$ of pointers storing S.

Hash function, hash value, hash table



Question:

How to use (h,T) for searching an element $i \in U$?

Answer:

$k \leftarrow h(i)$;

Search element i in the list $T[k]$.

Time complexity for searching:

O(length of the **longest** list in T).

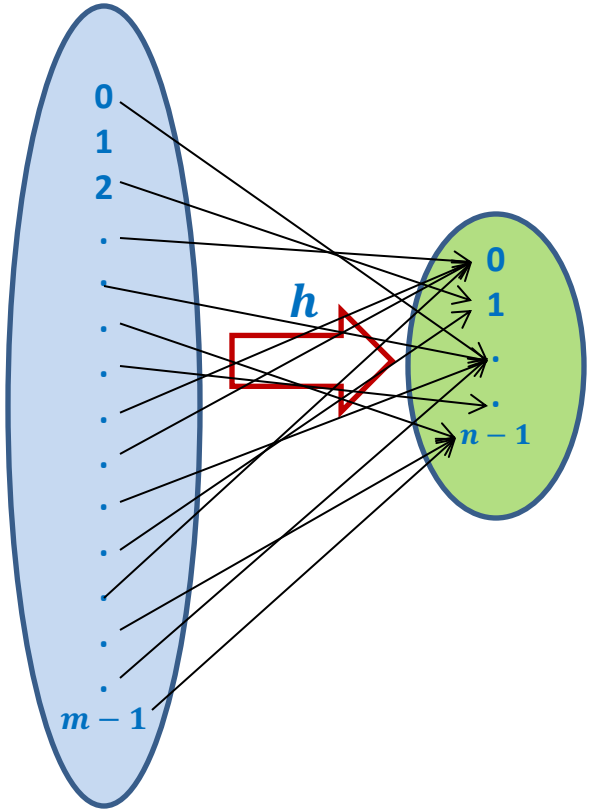
Efficiency of Hashing depends upon hash function

A hash function h is <u>good</u> if it can **evenly** distributes S.

Aim: To search for a good hash function for a given set S.

There **can not be** any hash function h which is good for **<u>every</u>** S**.**

Hash function, hash value, hash table



For every h, there exists a subset of $\left\lceil \frac{m}{n} \right\rceil$ elements from U which are hashed to same value under h.

So we can always construct a subset S for which all elements have same hash value

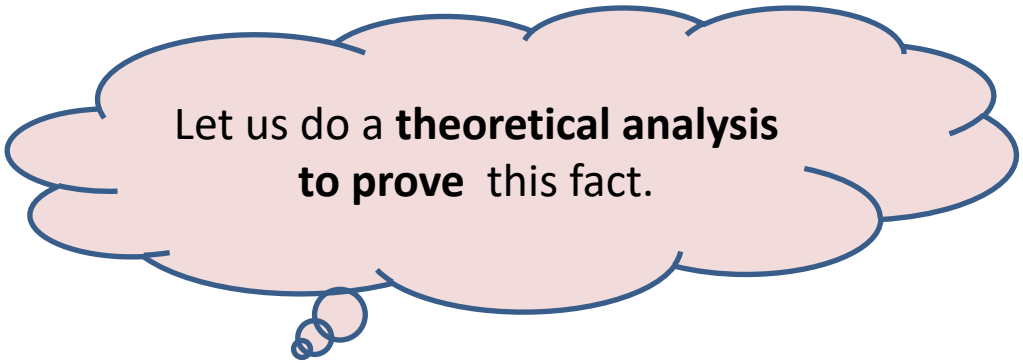
➔ All elements of this set S are present in a single list of the hah table T associated with h.

➔ **O**(n) worst case search time.

Why does hashing work so well in Practice ?

$$h(i) = i \bmod n$$

Because the set S is usually a **uniformly random** subset of U.



Why does hashing work so well in Practice ?

$$h(x) = x \bmod n$$

Let $y_1, y_2, \ldots, y_n$ denote n elements selected randomly uniformly from U to form S.

Question:

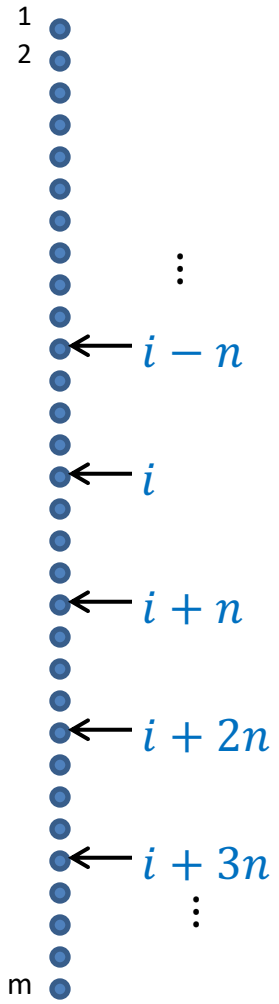
What is expected number of elements of S colliding with y_1?

Answer: Let y_1 takes value i.

P(y_j collides with y_1) = ??

How many possible values can y_j take ? $m-1$

How many possible values can collide with i ?



Why does hashing work so well in Practice ?

$$h(x) = x \bmod n$$

Let $y_1, y_2, \ldots, y_n$ denote n elements selected randomly uniformly from U to form S.

Question:

What is expected number of elements of S colliding with y_1?

Answer: Let y_1 takes value i.

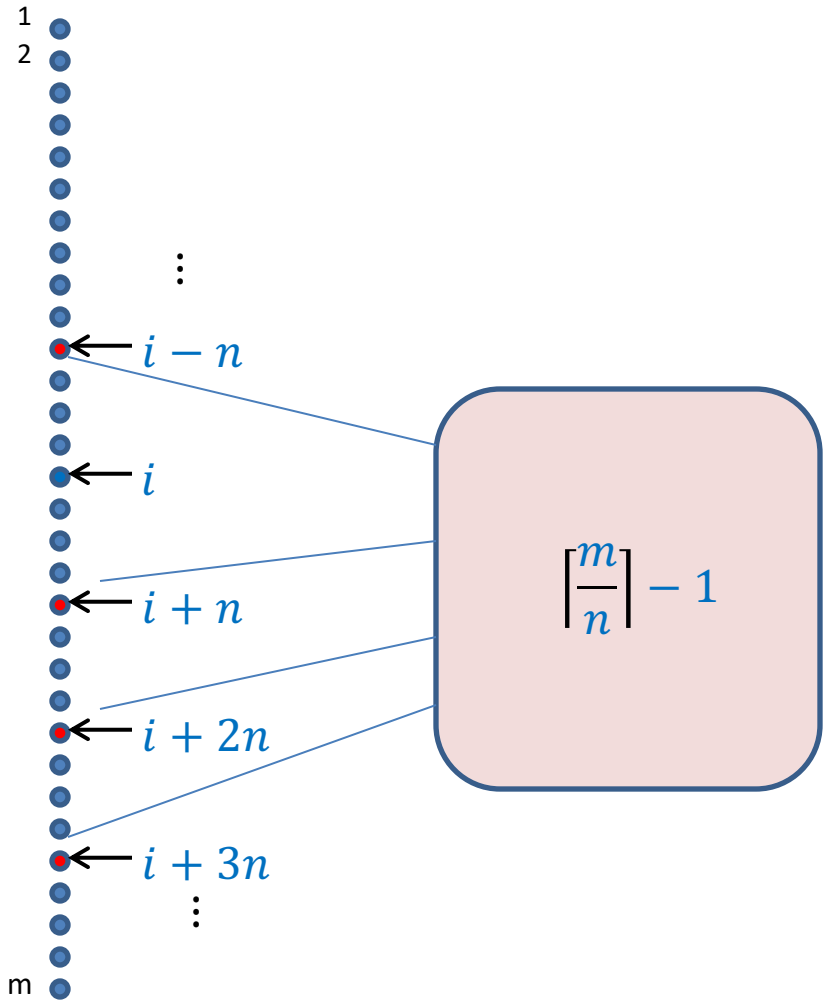
P(y_j collides with y_1) =

$$\frac{\lceil \frac{m}{n} \rceil - 1}{m-1}$$

Expected number of elements of S colliding with y_1 =

$$= \frac{\lceil \frac{m}{n} \rceil - 1}{m-1}(n-1)$$

$$= O(1)$$



Why does hashing work so well in Practice ?

Conclusion

1. $\boldsymbol{h}(i) = i \bmod n$ works so well because
   for a uniformly random subset of $\boldsymbol{U}$,
   the **expected** number of collision at an index of $\boldsymbol{T}$ is **O**(1).

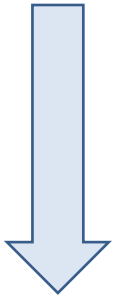
It is easy to fool this hash function such that it achieves **O**(s) search time.
(do it as a simple exercise).

This makes us think:

"How can we achieve worst case **O**(1) search time for a given set $\boldsymbol{S}$."

Hashing: theory

1953



1984

$U : \{0,1, \dots, m-1\}$

$S \subseteq U$,

$n = |S|$,

Theorem [FKS, 1984]:

A hash table and hash function can be computed in <u>average</u> **O(n) time** for a **<u>given</u>** S s.t.

Space : **O(n)**

Query time: worst case **O(1)**

Ingredients :

- elementary knowledge of **prime numbers**.
- The algorithms use **simple randomization**.

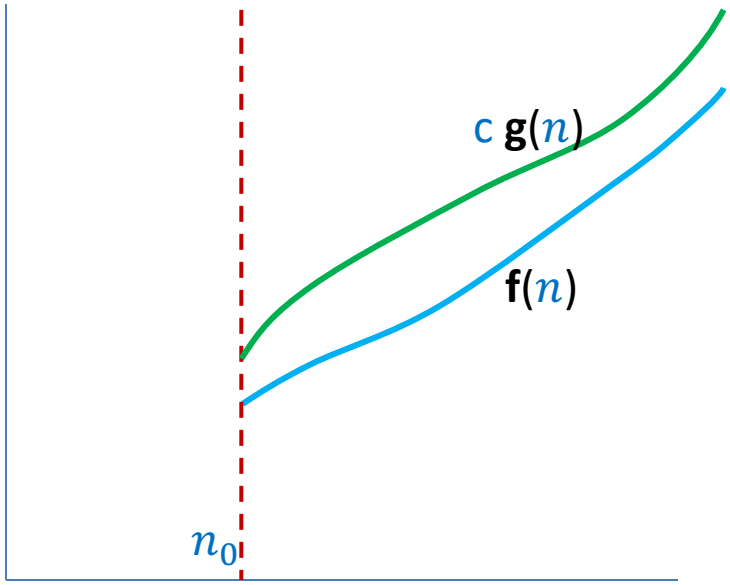
Order notation

Definition: Let **f**(n) and **g**(n) be any two increasing functions of n.

f(n) is said to be

if there exist constants c and n_0 such that

f(n) ≤ c **g**(n) for all n > n_0



f(n) = **O**(**g**(n))

Order notation extended

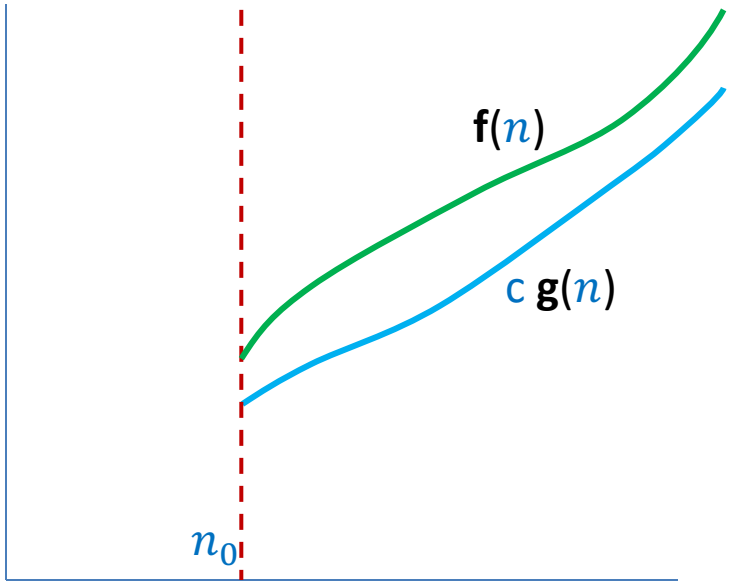
Definition: Let **f**(n) and **g**(n) be any two increasing functions of n.

f(n) is said to be

if there exist constants c and n_0 such that

f(n) ≥ c **g**(n) for all n > n_0

f(n) = **Ω**(**g**(n))



$$\frac{n^2}{100} = \Omega(10000\, n \log n)$$

Order notation extended

Observations:

- **f**(n) = **O**(**g**(n)) **if and only if** **g**(n) = **Ω**(**f**(n))

One more Notation:

If **f**(n) = **O**(**g**(n)) **and** **g**(n) = **O**(**f**(n)), then

g(n) = **Θ**(**f**(n))

Examples:

- $\frac{n^2}{100} =$ **Θ**($10000\ n^2$)

- Time complexity of Quick Sort is $\boldsymbol{\Omega}(n \log n)$

- Time complexity of Merge sort is $\boldsymbol{\Theta}(n \log n)$