Data Structures and Algorithms

(ESO207)

Lecture 1:

- An **overview** and **motivation** for the course
- some **concrete** examples.

Acknowledgment

Thanks to Prof Surender Baswana for allowing me to use and modify his lecture slides.

The website of the course

moodle.cse.iitk.ac.in

- CSE

  ESO207: Data Structures and Algorithms

Prerequisite of this course

- A good command on Programming in C
  - Programs involving arrays
  - Recursion
  - Linked lists (preferred)

- **Fascination for solving Puzzles**

Salient features of the course

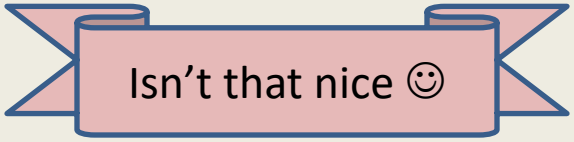
- **Every concept** — **We** shall **re-invent** in the class itself.
- **Solving each problem** — Through discussion **in the class**.

solution will emerge naturally if we ask

right set of questions

and then try to find their **answers**.

… so that finally it is a concept/solution derived by you and not a concept from some scientist/book/teacher.



Let us open a desktop/laptop

A processor (CPU)

speed = few GHz
(a few **nanoseconds** to execute an instruction)

Internal memory (RAM)

size = a few GB (Stores a billion bytes/words)
speed = a few GHz(a few **nanoseconds** to read a byte/word)

External Memory (Hard Disk Drive)

size = a few tera bytes
speed : seek time = **miliseconds**
transfer rate= around **billion** bits per second

A simplifying assumption
(for the rest of the lecture)

It takes around a few **nanoseconds** to execute an instruction.

*(This assumption is **well supported** by the modern day computers)*

EFFICIENT ALGORITHMS

What is an algorithm ?

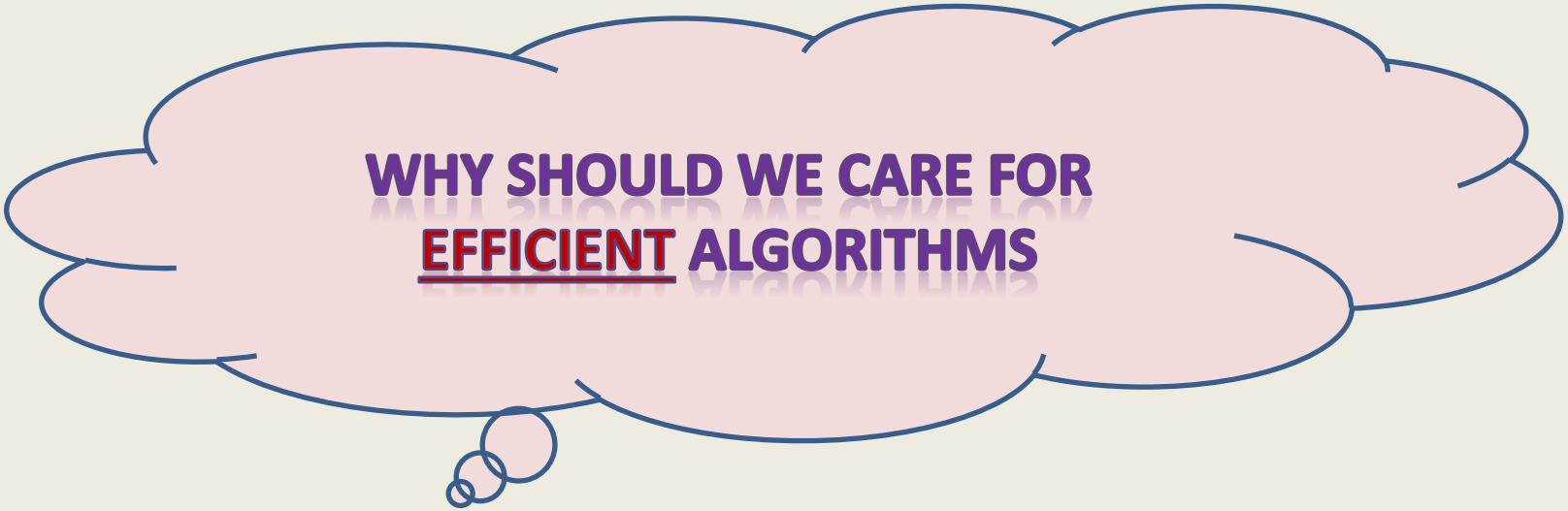
Definition:

A **finite sequence** of **well defined instructions**

required to solve a given computational problem.

A prime objective of the course:

Design of **efficient** algorithms



WE HAVE PROCESSORS RUNNING AT GIGAHERTZ?

Revisiting problems from ESC101

Problem 1:
Fibonacci numbers

Fibonacci numbers

$F(0) = 0;$
$F(1) = 1;$
$F(n) = F(n-1) + F(n-2)$ for all $n > 1;$

$$F(n) \approx a \cdot b^n$$

An easy exercise : Using induction or otherwise, show that

$$F(n) > 2^{\frac{n-2}{2}}$$

Algorithms you must have implemented for computing $F(n)$:

- **Iterative**
- **recursive**

Iterative Algorithm for $F(n)$

```
IFib(n)
if n=0 return 0;
    else if n=1 return 1;
        else {
                    a ← 0;   b← 1;
                    For(i=2 to n) do
                    {     temp ← b;
                          b← a+b;
                          a← temp;
                     }
              }
  return b;
```

Recursive algorithm for $F(n)$

```
Rfib(n)
{   if n=0 return 0;
        else if n=1 return 1;
                else return(Rfib(n-1) + Rfib(n-2))
}
```

Homework 1

(compulsory)

Write a **C** program for the following problem:

Input: a number n

n : **long long int** (64 bit integer).

Output: **F(n) mod 2014**

Time Taken	Largest n for Rfib	Largest n for IFib
1 minute		
10 minutes		
60 minutes		

Problem 2:
Subset-sum problem

Input: An array **A** storing n numbers, and a number s

A	12	3	46	34	19	101	208	120	219	115

220

Output: Determine if there is a subset of numbers from **A** whose sum is s.

The fastest existing algorithm till date : $2^{n/2}$ instructions

- Time for $n = 100$ **At least an year**
- Time for $n = 120$ **At least 1000 years**

on the fastest existing computer.

Problem 3:
Sorting

Input: An array **A** storing **n** numbers.

Output: Sorted **A**

A fact:

A significant fraction of the code of all the software is for sorting or searching only.

To sort **10 million** numbers on the present day computers

- **Selection sort** will take at least a few hours.
- **Merge sort** will take only a few seconds.
- **Quick sort** will take ??? .

How to design efficient algorithm for a problem ?

Design of algorithms and data structures is also an Art

Requires:

- **Creativity**
- **Hard work**
- **Practice**
- **Perseverance** (most important)

Summary of Algorithms

- There are many practically relevant problems for which there does not exist any efficient algorithm till date ☹. (How to deal with them ?)
- Efficient algorithms are important for theoretical as well as practical purposes.
- Algorithm design is an art which demands a lot of creativity, intuition, and perseverance.
- More and more applications in real life require efficient algorithms
  - Search engines like **Google** exploits many clever algorithms.

THE DATA STRUCTURES

An Example

Given: a telephone directory storing telephone no. of **hundred million** persons.

Aim: to answer a sequence of **queries** of the form

"what is the phone number of a given person ?".

Solution 1 :

Keep the directory in an array.

do **sequential search** for each query.

Time per query: around **1/10th** of a **second**

Solution 2:

Keep the directory in an array, and **sort it** according to names,

do **binary search** for each query.

Time per query: less than **100 nanoseconds**

Aim of a data structure ?

To store/organize a given data in the memory of computer so that each subsequent operation (query/update) can be performed quickly ?

Range-Minima Problem

A Motivating example

to realize the importance of data structures

Range-Minima Problem

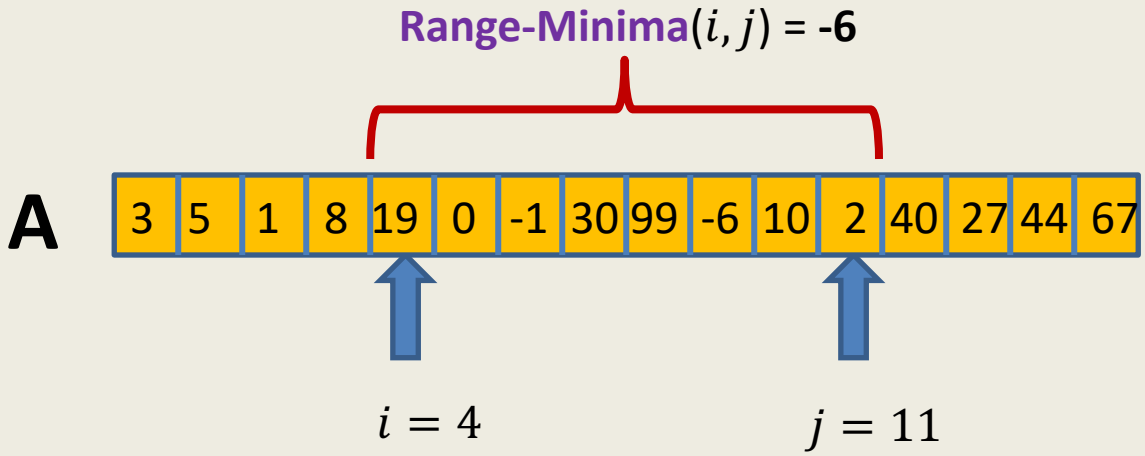
Given: an array **A** storing n numbers,

Aim: a data structure to answer a sequence of queries of the following type

Range-minima(i, j) : report the smallest element from **A**[i],…,**A**[j]

Let n = one **million**.

No. of queries = **10 millions**



Range-Minima Problem

Applications:

- **Computational geometry**
- **String matching**
- **As an efficient subroutine in a variety of algorithms**

(we shall discuss these problems sometime in this course or the next level course CS345)

Range-Minima Problem

Solution 1:

Answer each query in a brute force manner using **A** itself.

```
Range-minima-trivial(i,j)
{    temp ← i+1;
     min ← A[i];
      While(temp <= j)
      {   if (min > A[temp])
              min ← A[temp];
          temp← temp+1;
      }
      return min
}
```

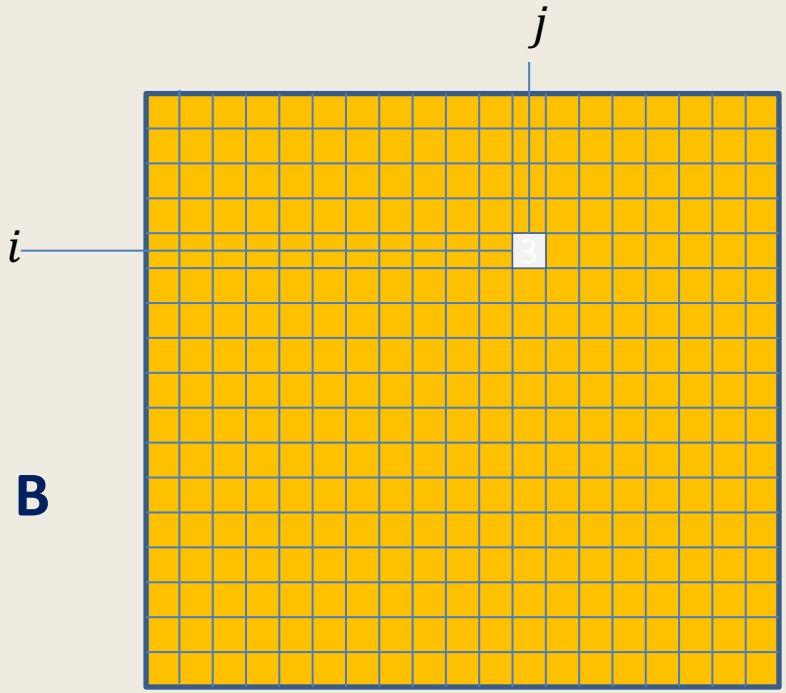
Time for answering **all** queries:
a few hours

Time taken to answer a query: few **milliseconds**

Range-Minima Problem

Solution 2:

Compute and store answer for each possible query in a $n \times n$ matrix **B**.



Solution 2 is Theoretically efficient but practically impossible

Size of **B** is **too large** to be kept in RAM. So we shall have to keep most of it in the **Hard disk drive.** Hence it will take a few **milliseconds per query**.

B$[i][j]$ stores the smallest element from **A**$[i]$,…,**A**$[j]$

Space : roughly n^2 words.

Range-Minima Problem

Question: Does there exist a data structure for Range-minima which is

- **Compact**

  (nearly **the same size** as the input array A)
- **Can answer each query efficiently ?**

  (a few **nanoseconds** per query)

Homework 2: Ponder over the above question.

(we shall solve it soon)

Data structures to be covered in this course

Elementary Data Structures

- Array
- List
- Stack
- Queue

Hierarchical Data Structures

- Binary Heap
- Binary Search Trees

Augmented Data Structures

- Look forward to working with all of you to make this course enjoyable.

- This course will be light in contents (no formulas)

  But it will be very demanding too.

- In case of any difficulty during the course,

  just drop me an email without any delay.

  I shall be happy to help ☺

Data Structures and Algorithms

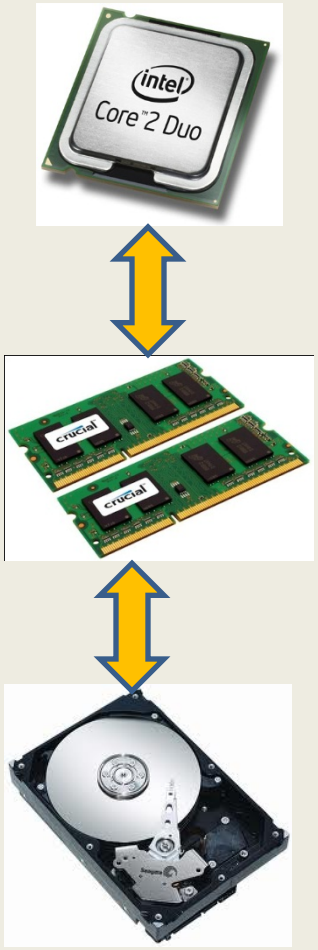
(ESO207)

Lecture 2:

- Model of computation
- Efficient algorithm for **F**(n) **mod** m.

RECAP OF THE 1ST LECTURE

Current-state-of-the-art computer



A processor (CPU)

speed = few GHz

(a few **nanoseconds** to execute an instruction)

Internal memory (RAM)

size = a few GB (Stores few million bytes/words)

speed = a few GHz(a few **nanoseconds** to read a byte/word)

External Memory (Hard Disk Drive)

size = a few tera bytes

speed : seek time = **miliseconds**

transfer rate= a **billion** bytes per second

Motivation

- for Efficient algorithms
  - Subset sum problem
  - Sorting
- for Efficient Data Structures
  - Telephone Directory
  - Range Minima

Homework 1

(compulsory)

Write a **C** program for the following problem:

Input: a number n

n : **long long int** (64 bit integer).

Output: **F(n) mod 2014**

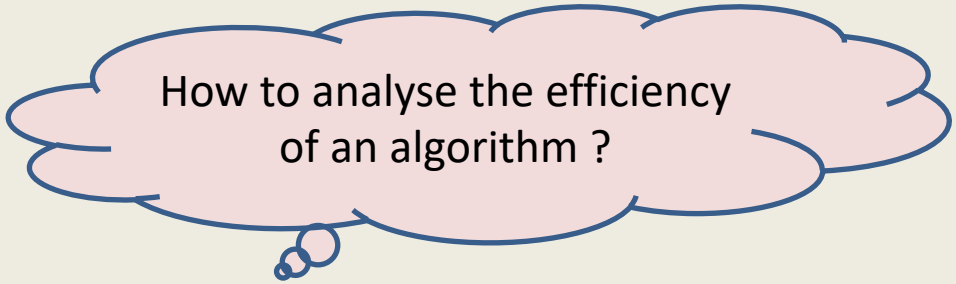
Time Taken	Largest n for Rfib	Largest n for IFib
1 minute	48	4.5×10^9
10 minutes	53	4.5×10^{10}
60 minutes	58	2.6×10^{11}

Processor: 2.7 GHz

Here are the values obtained by executing the program on a typical computer.

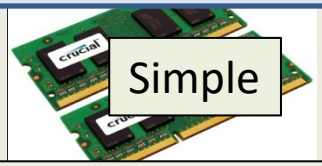
Inferences
from the Homework

- Inference for **RFib** algorithm
  - Too slow ☹
- Inference for **Ifib** algorithm :
  - Faster than **Rfib**
  - But not a solution for the problem ☹
- Efficiency of an algorithm does matter
- Time taken to solve a problem may vary depending upon the algorithm used



Current-state-of-the-art Computer

A model of computation



Simple

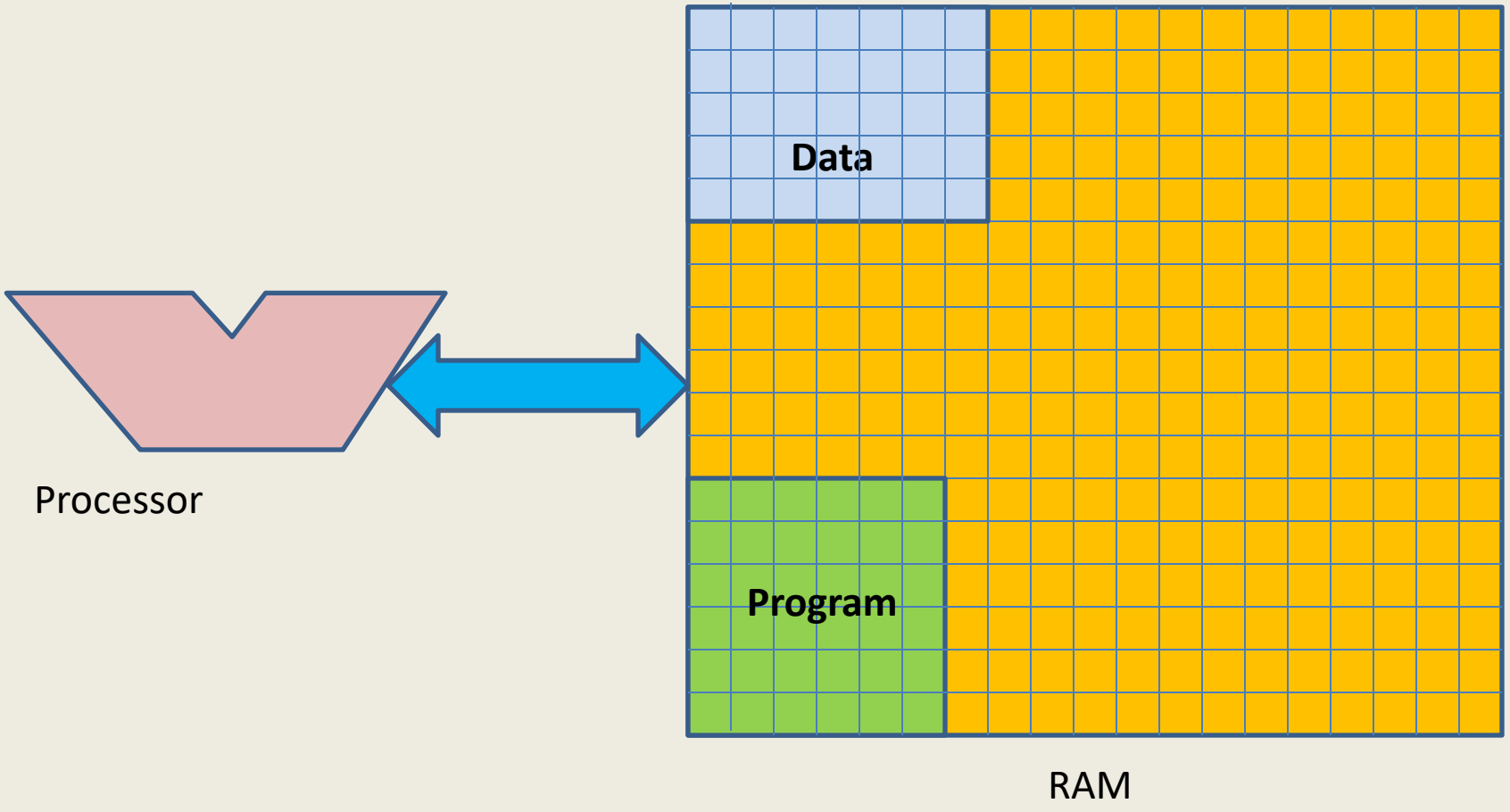
Close to real world computer

word RAM :

a model of computation

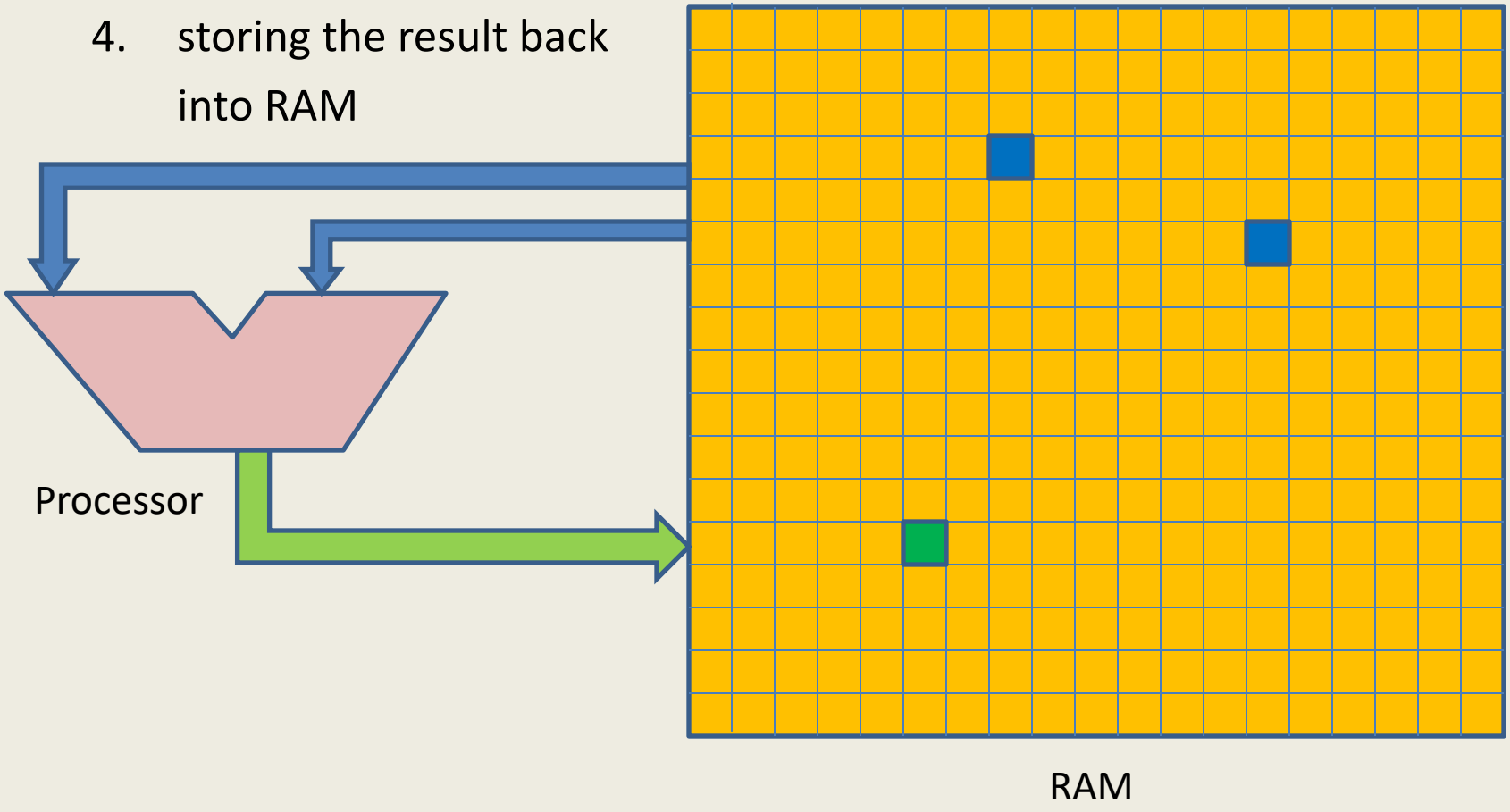
word RAM :

a model of computation



How is an instruction executed?

1. Decoding instruction
2. fetching the operands
3. performing arithmetic/logical operation
4. storing the result back into RAM



➔ Each instruction takes a few cycles (click ticks) to get executed.

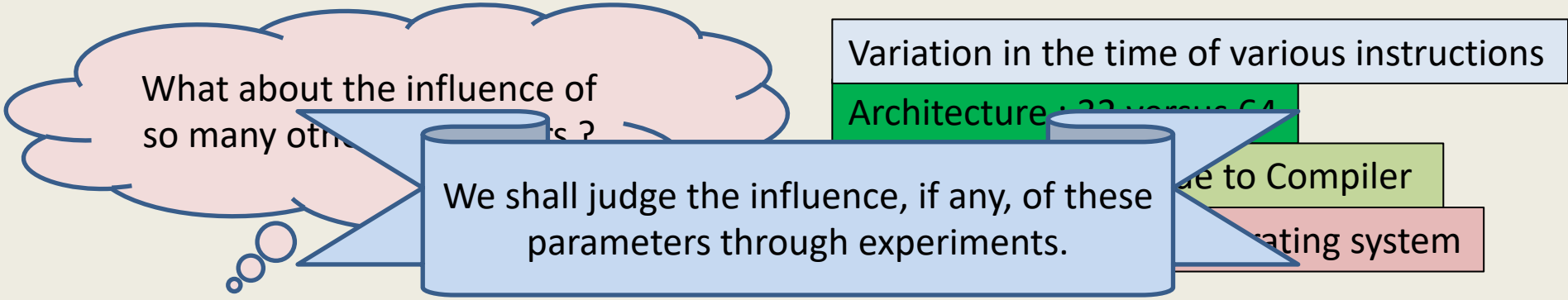
word RAM model of computation: Characteristics

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

Efficiency of an algorithm

Question: How to measure time taken by an algorithm ?

- Number of instructions taken in **word RAM** model.



Who knows, these factors might have little or negligible impact on most of algorithms. ☺

Homework 1 from Lecture 1

Computing F(n) mod m

Iterative Algorithm for F(n) mod m

Total number of instructions=
$4+3(n-1)+1 \approx 3n$

IFib(n,m)

```
if n = 0  return  0;
    else if n =1 return 1;
        else {      a ← 0;  b← 1;
                    For(i =2 to n) do
                    {    temp ← b;
                         b← a + b  mod m ;
                         a ← temp;
                    }
             }
  return b;
```

4 instructions

n-1 iterations

3 instructions per iteration

the final instruction

Let us calculate the number of instructions executed by IFib(n,m)

Recursive algorithm for F(n) mod m

RFib(n,m)

{ if n =0 return 0;

else if n =1 return 1;

else return((**RFib**$(n-1,m)$ + **RFib**$(n-2,m)$) mod m)

}

Let G(n) denote the number of instructions executed by RFib(n,m)

- **G(0) = 1;**
- **G(1) = 2;**
- For n **>1**

$$\mathbf{G}(n) = \mathbf{G}(n-1) + \mathbf{G}(n-2) + 4$$

Observation 1: **G**(n) >**F**(n) for all n;

➔**G**(n)> $2^{(n-2)/2}$**!!!**

Algorithms for F(n)mod m

- # instructions by **Recursive** algorithm **RFib**(n): $> 2^{\frac{n-2}{2}}$

  (exponential in n)

- # instructions by **Iterative** algorithm **IFib**(n): $3n$

  (linear in n)

None of them works for entire range of **long long int** n and **int** m

Question: Can we compute **F**(n)**mod** m quickly ?

How to compute F(n)mod m quickly ?

… need some better insight …

A warm-up example

How good are your programming skills ?

Compute $x^n \bmod m$

Problem: Given three integers x, n, and m, compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{otherwise} \end{cases}$$

Power(x, n, m)

```
If (n = 0) return 1;
 else {
        temp ← Power(x, n − 1, m);
        temp ← (temp × x) mod m ;
        return temp;
      }
```

4 instructions excluding the **Recursive** call

Compute $x^n \bmod m$

Problem: Given three integers x, n, and m, compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{otherwise} \end{cases}$$

Power(x, n, m)

Power$(x, n-1, m)$;

Power$(x, n-2, m)$;

Power$(x, 0, m)$

No. of instructions executed by **Power**(x, n, m) = $4n$

Compute $x^n \bmod m$

Problem: Given three integers x, n, and m, compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \times x^{n/2} & \text{if } n \text{ is even} \\ x^{n/2} \times x^{n/2} \times x & \text{if } n \text{ is odd} \end{cases}$$

Power(x, n, m)

```
If (n = 0) return 1;
else {
        temp ← Power(x, n/2, m);
        temp ← (temp × temp) mod m ;
        if (n mod 2 = 1 ) temp ←(temp × x)mod m ;
       return temp;
      }
```

5 instructions excluding the **Recursive** call

Compute $x^n \bmod m$

Problem: Given three integers x, n, and m, compute $x^n \bmod m$.

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x^{n/2} \times x^{n/2} & \text{if } n \text{ is } \textbf{even} \\ x^{n/2} \times x^{n/2} \times x & \text{if } n \text{ is } \textbf{odd} \end{cases}$$

Power(x, n, m)

Power$(x, n/2, m)$

Power$(x, n/4, m)$

Power$(x, 0, m)$

No. of instructions executed by **Power**(x, n, m) = $5 \log_2 n$

Efficient Algorithm for F(n)mod m

Idea 1

Question: Can we express **F**(n) as a^n for some constant a ?

Unfortunately **no**.

Idea 2

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & ? & 1 \\ 1 & & 0 \end{pmatrix} \times \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix}$$

Unfolding the RHS of this equation, we get …

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \times \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

A clever algorithm for F(n)mod m

Clever-algo-Fib(n, m)

{ $A \leftarrow \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$;

$B \leftarrow A^{n-1} \bmod m$;

$C \leftarrow B \times \begin{pmatrix} 1 \\ 0 \end{pmatrix}$;

return $C[1]$; // the first element of vector C stores F(n)**mod** m

}

4 instructions

6 instructions

Question: How to compute $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ efficiently ?

Answer :

Inspiration from Algorithm for x^n **mod** m

A clever algorithm for F(n)mod m

Let A be a 2×2 matrix.

- If n is even, $A^n = A^{n/2} \times A^{n/2}$
- If n is odd, $A^n = A^{n/2} \times A^{n/2} \times A$

Question: How many instructions are required to multiply two 2×2 matrices ?

Answer: 12

Question: Number of instructions for computing $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \bmod m$?

Answer : 35 $\log_2 (n-1)$

Question: Number of instructions in **New-algo-Fib**(n, m)

Answer: 35 $\log_2(n-1)$+ 11

Three algorithms

Algorithm for F(n)mod m	No. of Instructions
RFib(n,m)	$> 2^{(n-2)/2}$
IterFib(n,m)	$3n$
Clever_Algo_Fib(n,m)	$35 \log_2 (n-1) + 11$

?

- Which algorithm is the best ?
- What is the exact base of the exponent in the running time of **RFib** ?
- Are we justified in ignoring the influence of so many other parameters ?
  (Variation in the time of instructions/Architecture/Code optimization/…)
- How close to **reality** is the RAM model of computation ?

Find out yourself !

Data Structures and Algorithms

(ESO207)

Lecture 3:

- **Time complexity**, **Big "O" notation**
- Designing Efficient Algorithm
  - **Maximum sum subarray Problem**

Three algorithms

Algorithm for F(n)mod m	No. Instructions in RAM model
RFib(n,m)	$> 2^{(n-2)/2}$
IterFib(n,m)	$3n$
Clever_Algo_Fib(n,m)	$35 \log_2 (n-1) + 11$

- Which algorithm turned out to be the best **experimentally ?**

Lesson 1
from Assignment 1 ?

No. of instructions executed by algorithm in **RAM** model

Proportional to ?

Time taken by algorithm in real life

May be different for different input ?

Dependence on input

Time complexity of an algorithm

Definition:

the **worst case** number of instructions executed

as a **function** of the **input size** (or a parameter defining the input size)

The number of bits/bytes/words

Examples to illustrate input size :

Problem	Input size
Computing **F**(n)**mod** m for any positive integers n and m	$\log_2 n + \log_2 m$ bits
Whether an array storing j numbers (each stored in a **word**) is sorted ?	j words
Whether a $n \times m$ matrix of numbers (each stored in a **word**) contains "14" ?	nm words

Homework: What is the time complexity of **Rfib**, **IterFib**, **Clever-Algo-Fib** **?**

Example:

Whether an array **A** storing j numbers (<u>each stored in a **word**</u>) is sorted ?

IsSorted(A)

```
{   i←1;                                   1 time
    flag← true;                            1 time
     while(i < j  and flag==true)
    {
          If (A[i]< A[i − 1])    flag←false ;     } j − 1 times in the worst case
         i ← i + 1;
    }
    Return flag ;                          1 time
}
```

Time complexity = $2j + 1$

Example:

Time complexity of matrix multiplication

Each element of the matrices occupies one **word** of RAM.

Matrix-mult(**C**$[n, n]$,**D**$[n, n]$)

```
{   for i = 0 to n − 1                          n times
    {     for j = 0 to n − 1                    n times
          {       M[i, j] ← 0;
                  for k = 0 to n − 1
                  {       M[i, j]← M[i, j] + C[i, k]*D[k, j];   n + 1 instructions
                  }
           }
    }
    Return M                                    1 time
}
```

Time complexity = $n^3 + n^2 + 1$

Lesson 2 learnt from Assignment 1 ?

Algorithm for F(n)mod m	No. of Instructions
RFib(n,m)	$> 2^{(n-2)/2}$
IterFib(n,m)	$3n$
Clever_Algo_Fib(n,m)	$100 \log_2 (n-1) + 1000$

Question: What would have been the outcome if

No. of instructions of Clever_Algo_Fib(n,m) = $100 \log_2 (n-1) + 1000$

Answer: Clever_Algo_Fib would still be the fastest algorithm for large value of n.

COMPARING EFFICIENCY OF ALGORITHMS

Comparing efficiency of two algorithms

Let **A** and **B** be two algorithms to solve a given problem.

Algorithm **A** has time complexity : $2\,n^2 + 125$

Algorithm **B** has time complexity : $5\,n^2 + 67\,n + 400$

Question: Which algorithm is more efficient ?

Obviously **A** is more efficient than **B**

Comparing efficiency of two algorithms

Let **A** and **B** be two algorithms to solve a given problem.

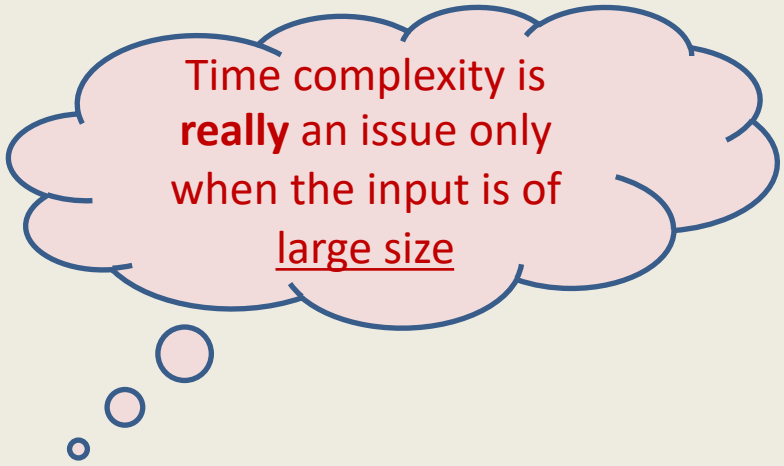
Algorithm **A** has time complexity : $2\,n^2 + 125$

Algorithm **B** has time complexity : $50\,n + 125$

Question: Which one would you prefer based on the **efficiency** criteria ?

Answer : A is more efficient than **B** for $n < 25$

B is more efficient than **A** for $n > 25$



Rule 1

Compare the **time complexities** of two algorithms for **asymptotically large value** of input size only

Comparing efficiency of two algorithms

Algorithm **B** with time complexity $50\,n + 125$

is certainly more efficient than

Algorithm **A** with time complexity : $2n^2 + 125$

A judgment question for you !

Algorithm **A** has time complexity **f**(n)= $5\,n^2 + n + 1250$

Researchers have designed two new algorithms B and C

- Algorithm **B** has time complexity **g**(n) = $n^2 + 10$
- Algorithm **C** has time complexity **h**(n) = $10\,n^{1.5} + 20\,n + 2000$

Which of **B** and **C** is an improvement over **A** in the **true sense** ?

$$\lim_{n\to\infty} \frac{\mathbf{g}(n)}{\mathbf{f}(n)} = 1/5$$

$$\lim_{n\to\infty} \frac{\mathbf{h}(n)}{\mathbf{f}(n)} = 0$$

C is an improvement over **A** in the true sense.

Rule 2

An algorithm **X** is superior to another algorithm **Y** if the **ratio** of time complexity of **X** and time complexity of **Y** **approaches 0** for asymptotically large input size.

Some Observations

Algorithm **A** has time complexity **f**(n)= $5n^2 + n + 1250$

Researchers have designed two new algorithms B and C

- Algorithm **B** has time complexity **g**(n) = $n^2 + 10$
- Algorithm **C** has time complexity **h**(n) = $10n^{1.5} + 20n + 2000$

Algorithm **C** is **the most efficient** of all.

Observation 1:

multiplicative or additive **Constants** do **not** play any role**.**

Observation 2:

The highest order term governs the time complexity asymptotically.

ORDER NOTATIONS

A **mathematical** **way**
to capture the **intuitions** developed till now.
(reflect upon it yourself)

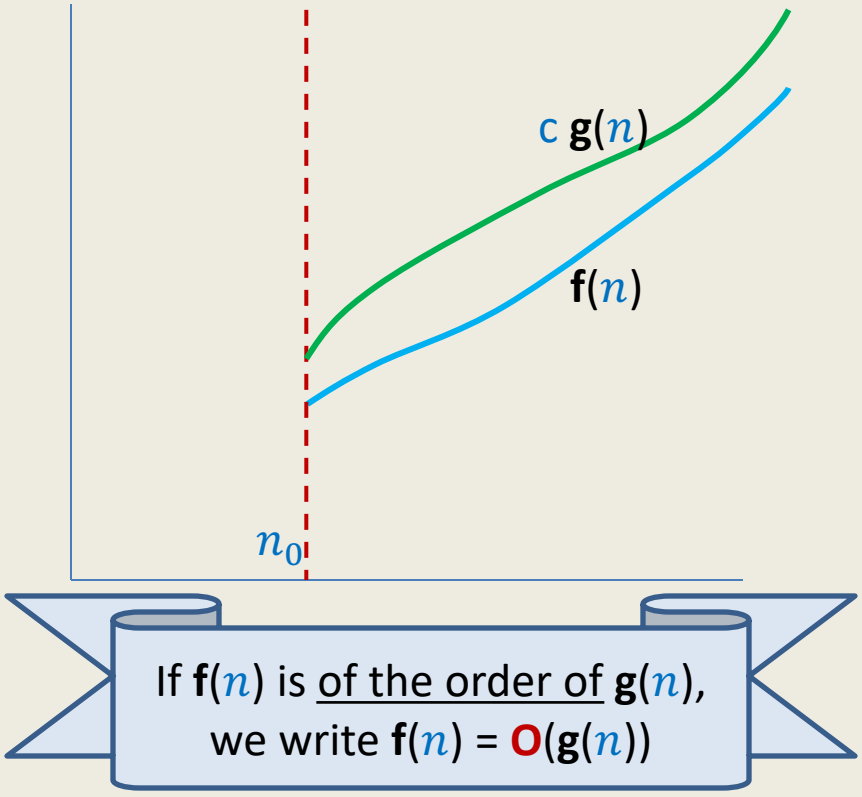
Order notation

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

f(n) is said to be <u>of the order of</u> **g**(n)

if there exist constants c and n_0 such that

f(n) ≤ c **g**(n) for all n > n_0



Order notation :
Examples

- $20\,n^2$ = **O**(n^2)
- ~~$100\,n + 60$ = **O**(n^2)~~ **Loose**
- $100\,n + 60$ = **O**(n)
- ~~$10\,n^2$ = **O**($n^{2.5}$)~~ **Loose**
- 2000 = **O**(1)

c =20, $n_0 = 1$

c =1, $n_0 = 160$

c =160, $n_0 = 1$

While analyzing time complexity of an algorithm accurately, our aim should be to choose the **g**(n) which is not **loose**.
Later in the course, we shall refine & extend this notion suitably.

Simple observations:

- If **f**(n) = **O**(**g**(n)) and **g**(n) = **O**(**h**(n)), then
  f(n) = **O**(**h**(n))
- If **f**(n) = **O**(**h**(n)) and **g**(n) = **O**(**h**(n)), then **f**(n) + **g**(n) = **?**(**h**(n))

These observations can be helpful for simplifying time complexity.

Prove these observation as Homeworks

A neat description of time complexity

- Algorithm **B** has time complexity **g**(n) = $n^2 + 10$

  Hence **g**(n) = **O**(n^2)

- Algorithm **C** has time complexity **h**(n) = $10\ n^{1.5} + 20\ n + 2000$

  Hence **h**(n) = **O**($n^{1.5}$)

- Algorithm for multiplying two $n \times n$ matrices has time complexity

  $n^3 + n^2 + 1$ = **O**(n^3)

Homeworks:

- **g**(n) = 2^n , **f**(n) = 3^n . Is **f**(n) = **O**(**g**(n)) ? Give proof.
- What is the time complexity of **selection sort** on an array storing n elements ?
- What is the time complexity of **Binary search** in a sorted array of n elements ?

HOW TO DESIGN EFFICIENT ALGORITHM ?

(This sentence captures precisely the goal of theoretical computer science)

Designing an efficient algorithm

Facts from the world of algorithms:

1. **No formula** for designing efficient algorithms.
2. Every new problem demands a **fresh** approach.

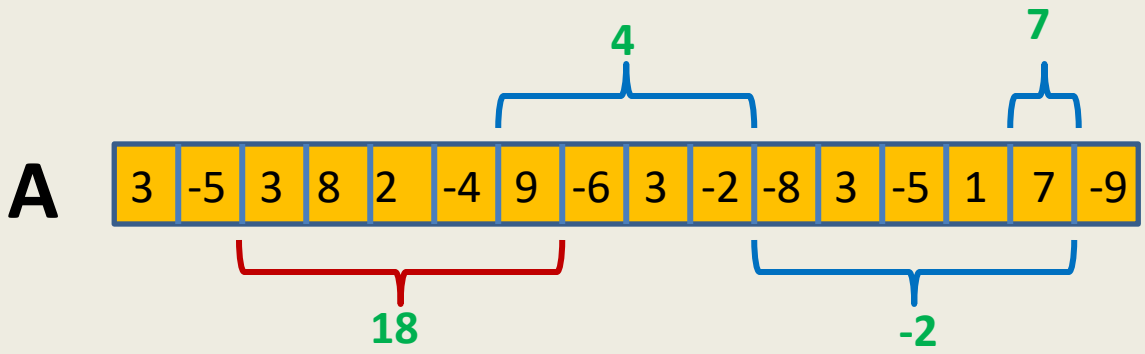
3. Designing an efficient algorithm or data structure requires
    1. Ability to make **key observations**.
    2. Ability to ask **right kind of questions**.
    3. A **positive attitude** and …
    4. a lot of **perseverance**.

We shall demonstrate the above facts during this course many times.

Max-sum subarray problem

Given an array **A** storing n numbers,

find its **subarray** the sum of whose elements is maximum.



Max-sum subarray problem: A trivial algorithm

```
A_trivial_algo(A)
{ max← A[0];
  For i=0 to n-1
     For j=i to n-1
          {    temp ← compute_sum(A,i,j);
               if max< temp  then max← temp;
          }
  return max;
}

compute_sum(A, i,j)
{  sum←A[i];
   For k=i+1 to j     sum← sum+A[k];
   return sum;
}
```

Homework: Prove that its time complexity is **O**(n^3)

Max-sum subarray problem:

Question: Can we design **O**(n) time algorithm for Max-sum subarray problem ?

Answer: Yes.

Think over it with a fresh mind ….
We shall design it together in the next class…☺

Data Structures and Algorithms

(ESO207)

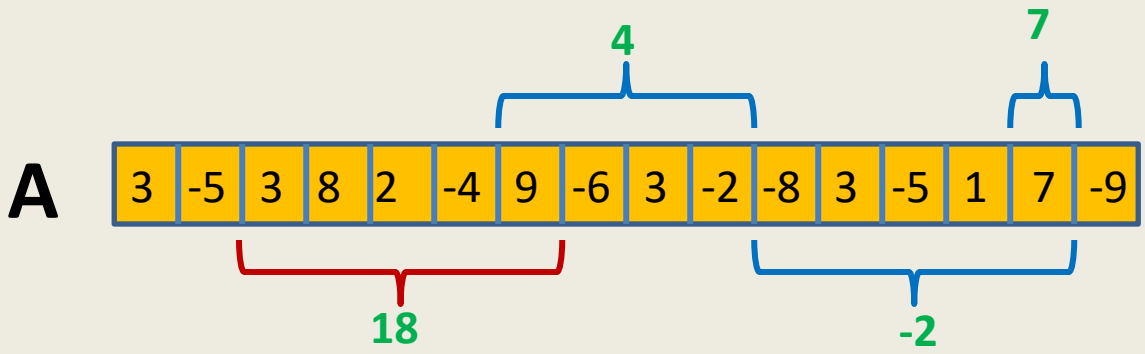
Lecture 4:

- Design of **O**(n) time algorithm for **Maximum sum subarray**
- Proof of **correctness** of an algorithm
- A new problem : **Local Minima in a grid**

Max-sum subarray problem

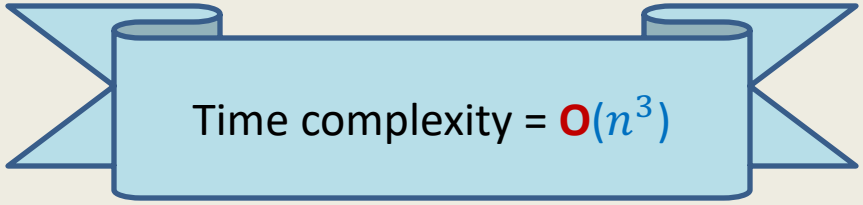
Given an array **A** storing n numbers,

find its **subarray** the sum of whose elements is maximum.



Max-sum subarray problem: A trivial algorithm

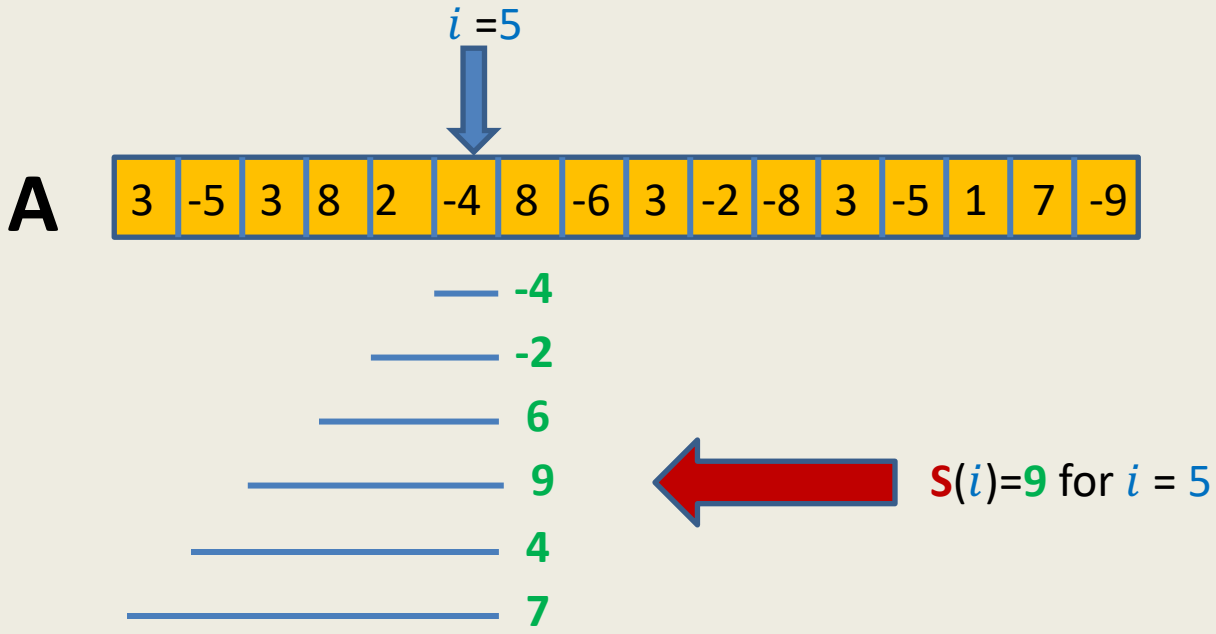
```
A_trivial_algo(A)
{ max← A[0];
  For i=0 to n-1
     For j=i to n-1
          {    temp ← compute_sum(A,i,j);
               if max< temp  then max← temp;
          }
 return max;
}

compute_sum(A, i,j)
{  sum←A[i];
   For k=i+1 to j     sum← sum+A[k];
   return sum;
}
```



DESIGNING AN O(n) TIME ALGORITHM

Focusing on any particular index i

Let **S**(i): the sum of the **maximum-sum subarray ending at index** i.



Observation:

In order to solve the problem, it suffices to compute **S**(i) for each $0 \leq i < n$.

Focusing on any particular index i

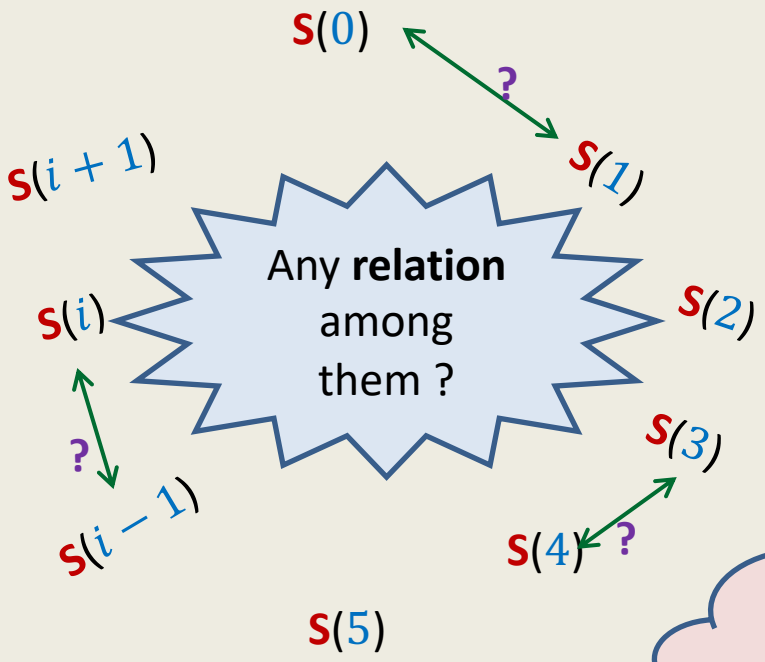
Observation:

In order to solve the problem, it suffices to compute **S**(i) for each $0 \leq i < n$.

Question: If we wish to achieve O(n) time to solve the problem,
how quickly should we be able to compute **S**(i) for a given index i ?

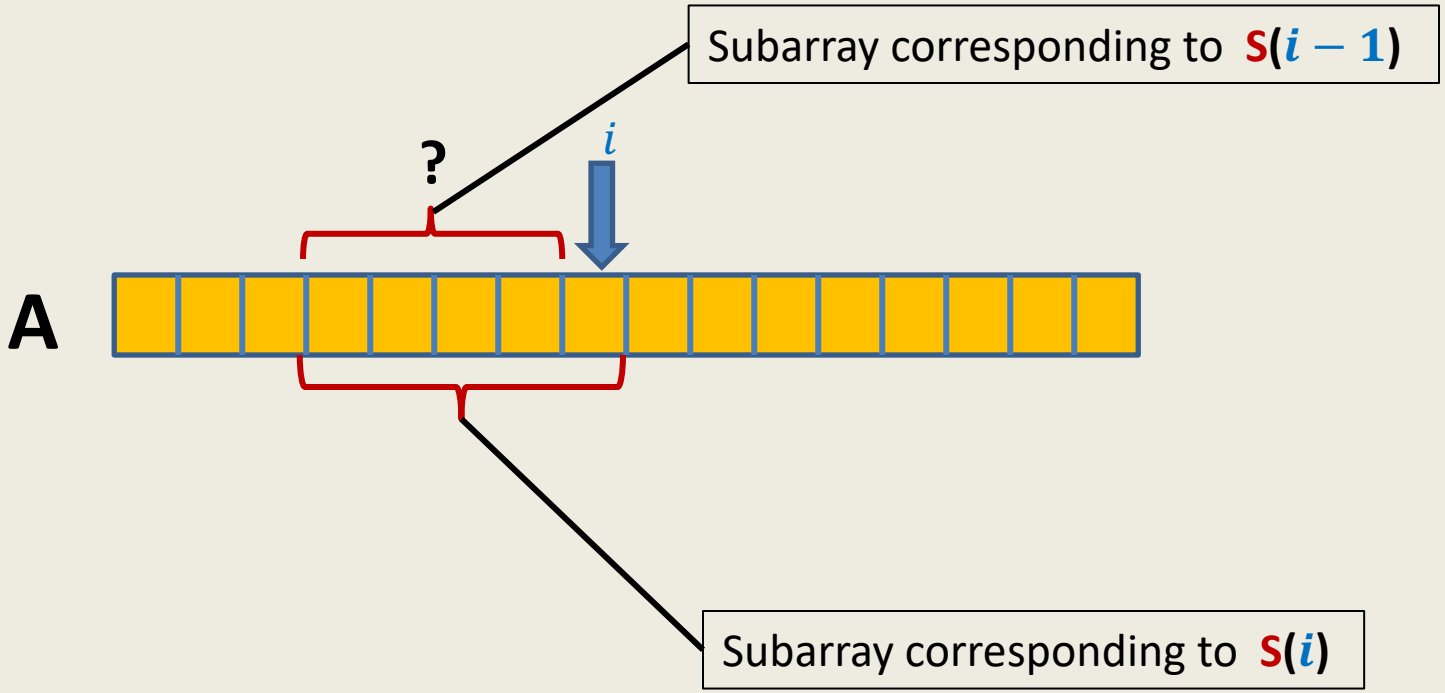
Answer: **O**(1) time.

How to compute S(i) in O(1) time ?



What is the **relation** between S(i) and S($i-1$) ?

Relation between S(i) and S($i-1$)



Theorem 1:

If S($i-1$) > 0 **then** S(i) = S($i-1$) + A[i]

else S(i) = A[i]

An O(n) time Algorithm for Max-sum subarray

Max-sum-subarray-algo(**A**[0 ... $n-1$])

{ **S**[0] ← **A**[0];

for i = **1** to $n-1$

{ **If S**[$i-1$] > 0 **then S**[i] ← **S**[$i-1$] + **A**[i]

else S[i] ← **A**[i]

}

"Scan **S** to return the maximum entry"

}

O(**1**) time

$n-1$ **repetitions**

O(**1**) time

O(n) time

Time complexity of the algorithm = **O**(n)

Homework:

- Refine the algorithm so that it uses only **O**(1) extra space.

An O(n) time Algorithm for Max-sum subarray

Max-sum-subarray-algo(**A**[$0 \ \ldots \ n-1$])

{ **S**[0] ← **A**[0]

for i = **1** to $n-1$

{ **If** **S**[$i-1$] > 0 **then** **S**[i] ← **S**[$i-1$] + **A**[i]

else **S**[i] ← **A**[i]

}

"Scan **S** to return the maximum entry"

}

What is **the proof of correctness** of the algorithm ?

What does correctness of an algorithm mean ?

For every possible **valid input**, the algorithm must output **correct** answer.

An O(n) time Algorithm for Max-sum subarray

Max-sum-subarray-algo(**A**[$0 \ \ldots \ n-1$])

```
{   S[0] ← A[0]
    for i = 1 to n − 1
    {     If S[i − 1] > 0   then S[i] ← S[i − 1] + A[i]
                            else S[i] ← A[i]
    }
    "Scan S to return the maximum entry"
}
```

Question:

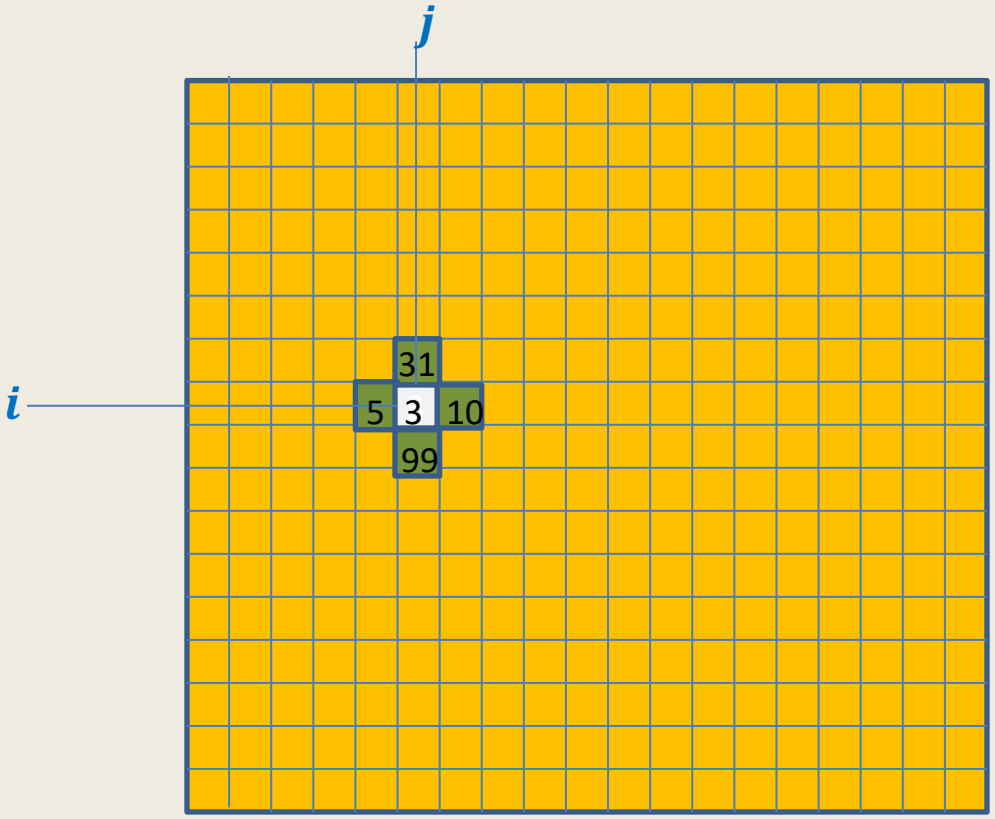
What needs to be proved in order to establish the correctness of this algorithm ?

Ponder over this question before coming to the next class…

NEW PROBLEM:
LOCAL MINIMA IN A GRID

Local minima in a grid

Definition: Given a $n \times n$ grid storing distinct numbers,
an entry is local minima if it is smaller than each of its neighbors.

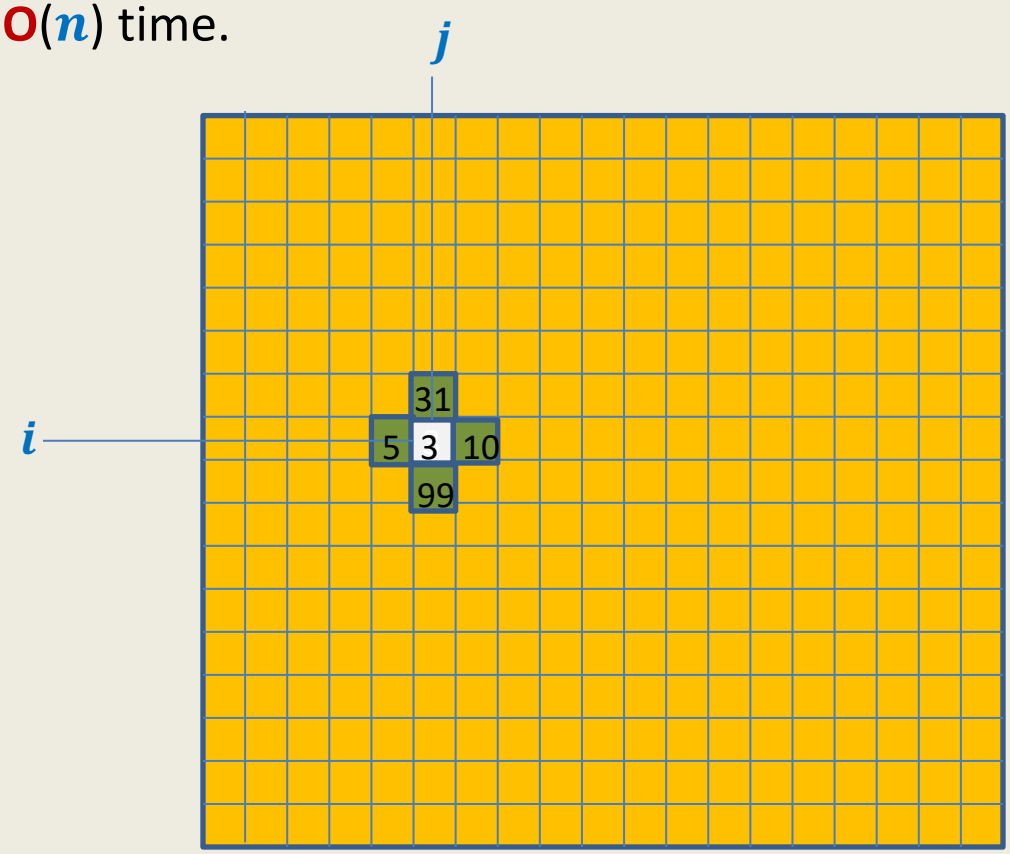


Does a **local minima** exist always ?

Yes. After all, **global minima** is also a **local minima**.

Local minima in a grid

Problem: Given a $n \times n$ grid storing distinct numbers, output any local minima in $O(n)$ time.



Using common sense principles

- There are some simple but very fundamental principles
  which are not restricted/confined to a specific stream of science/philosophy.

- These principles, which we usually learn as common sense,
  can be used in so many diverse areas of human life.

- For the current problem of local minima,
  we shall use two such simple principles.

This should convince you that designing algorithm does not require any thing **magical** ☺!

Two simple principles

1. Respect every new idea even if it does not solve a problem finally.

2. Principle of simplification:

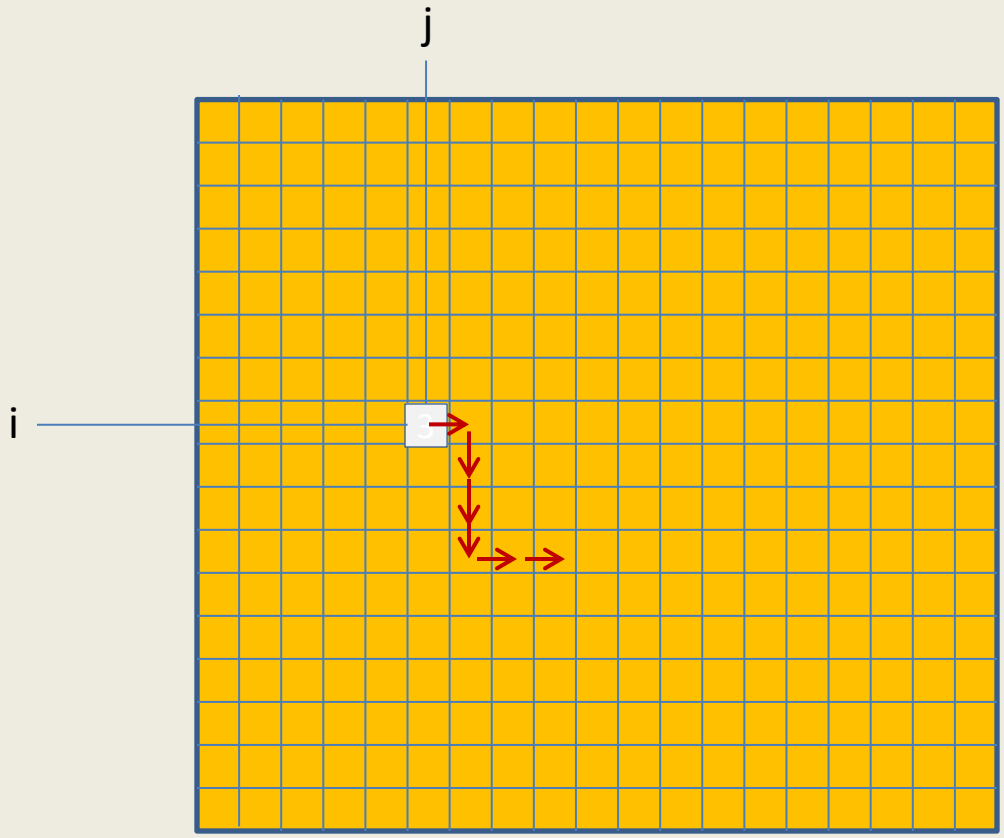
If you find a problem difficult,

➔ Try to solve its simpler version, and then …

➔ Try to extend this solution to the original (difficult) version.

A new approach

Repeat : *if current entry is not local minima, explore the neighbor storing smaller value.*



A new approach

```
Explore()
{    Let c be any entry to start with;
     While(c is not a local minima)
     {
        c ← a neighbor of c storing smaller value
     }
     return c;
}
```

Question: What is the proof of correctness of **Explore** ?

Answer:

➔It suffices if we can prove that **While** loop eventually terminates.

➔Indeed, the loop terminates since **we never visit a cell twice**.

A new approach

```
Explore()
{    Let c be any entry to start with;
     While(c is not a local minima)
     {
        c ← a neighbor of c storing smaller value
     }
     return c;
}
```

Worst case time complexity : **O**(n^2)

How to apply this principle ?

First principle:
Do not discard **Explore**()

Second principle:
Simplify the problem

Local minima in an array

A 

Theorem 2: A local minima in an array storing n distinct elements can be found in **O**(**log** n) time.

Homework:

- Design the algorithm stated in **Theorem 2**.
- Spend some time to extend this algorithm to grid with running time= **O**(n).

Please come prepared in the next class ☺

Data Structures and Algorithms

(ESO207)

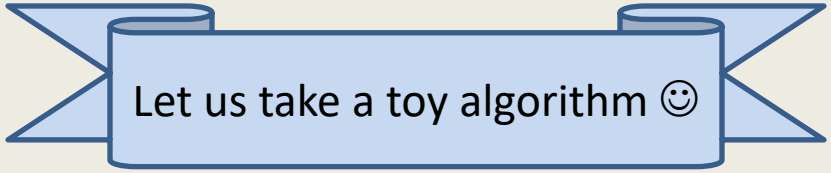
Lecture 5:

- More on **Proof of correctness** of an algorithm
- Design of **O**(n) time algorithm for **Local Minima in a grid**

PROOF OF CORRECTNESS

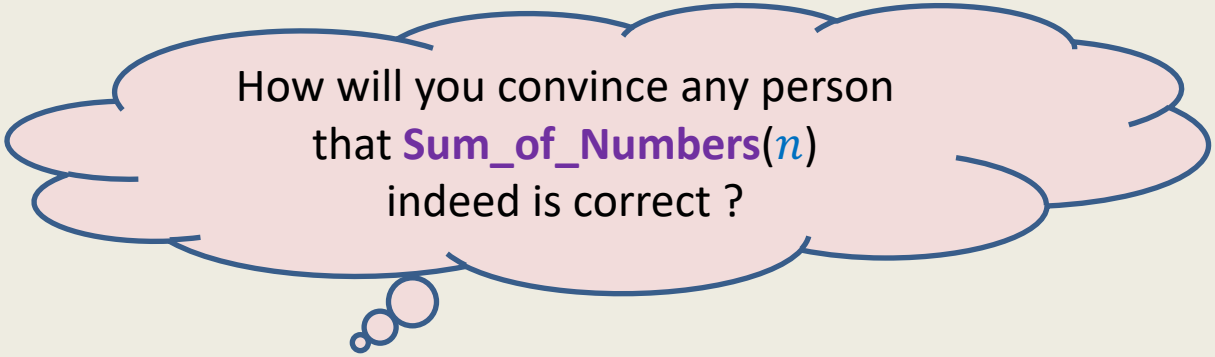
What does correctness of an algorithm mean ?

For every possible **valid input**, the algorithm must output **correct** answer.



Algorithm for computing sum of numbers from 0 to n

```
Sum_of_Numbers(n)
{   Sum←0;
    for i = 1 to n
    {
        Sum← Sum  + i;
    }
    return Sum;
}
```



Natural responses:

- It is obvious !
- Compile it and run it for some random values of n.
- Go over first few iterations explaining what happens to **Sum**.

How will you respond

if you have to do it for the following code ?

```
void dij(int n,int v,int cost[10][10],int dist[])
{
 int i,u,count,w,flag[10],min;
 for(i=1;i<=n;i++)
  flag[i]=0,dist[i]=cost[v][i];
 count=2;
 while(count<=n)
 {
  min=99;
 for(w=1;w<=n;w++)
   if(dist[w]<min && !flag[w])
    min=dist[w],u=w;
  flag[u]=1;
  count++;
 for(w=1;w<=n;w++)
   if((dist[u]+cost[u][w]<dist[w]) && !flag[w])
    dist[w]=dist[u]+cost[u][w];
 }
}
```

Think for some time to realize

- the non-triviality
- the Importance

of proof of correctness of an iterative algorithm.

In the following slide, we present an overview of the proof of correctness.

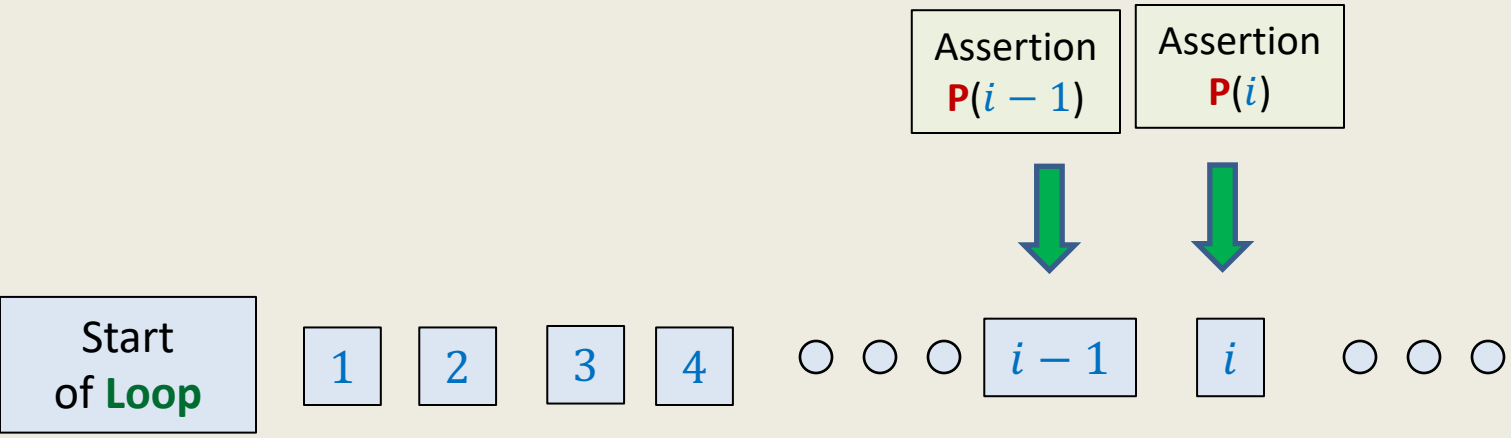
Interestingly, such a proof will be just

Expressing our intuition/insight of the algorithm in a **formal** way ☺.

Proof of correctness

For an iterative algorithm

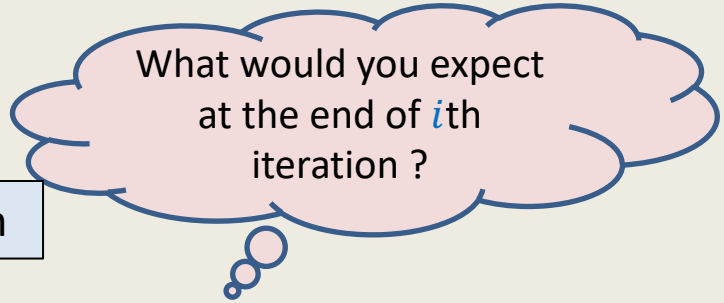
Insight of the algorithm ⟶ **Theorem**



Prove **P**(i) by

1. Assuming **P**($i-1$)
2. **Theorem**
3. Body of the **Loop**

} Proof by induction



The most difficult/creative part of proof : To come up with the right assertion **P**(i)

Algorithm for computing sum of numbers from 0 to n

```
{   Sum←0;
    for i = 1 to n
    {
        Sum← Sum  + i;
    }
    return Sum;
}
```

Assertion P(i) : At the end of ith iteration **Sum** stores the sum of numbers from 0 **to** i.

Base case: **P**(0) holds.

Assuming **P**($i-1$), assertion **P**(i) also holds.

P(n) holds.

An O(n) time Algorithm for Max-sum subarray

Let **S**(i): the sum of the **maximum-sum subarray ending at index** i.

Theorem 1 : If S($i-1$) > 0 **then S**(i) = **S**($i-1$) + **A**[i]

else S(i) = **A**[i]

Max-sum-subarray-algo(**A**[$0 \ \ldots \ n-1$])

```
{   S[0] ← A[0]
    for i = 1 to n − 1
    {     If S[i − 1] > 0  then S[i] ← S[i − 1] + A[i]
                           else S[i] ← A[i]
    }
    "Scan S to return the maximum entry"
}
```

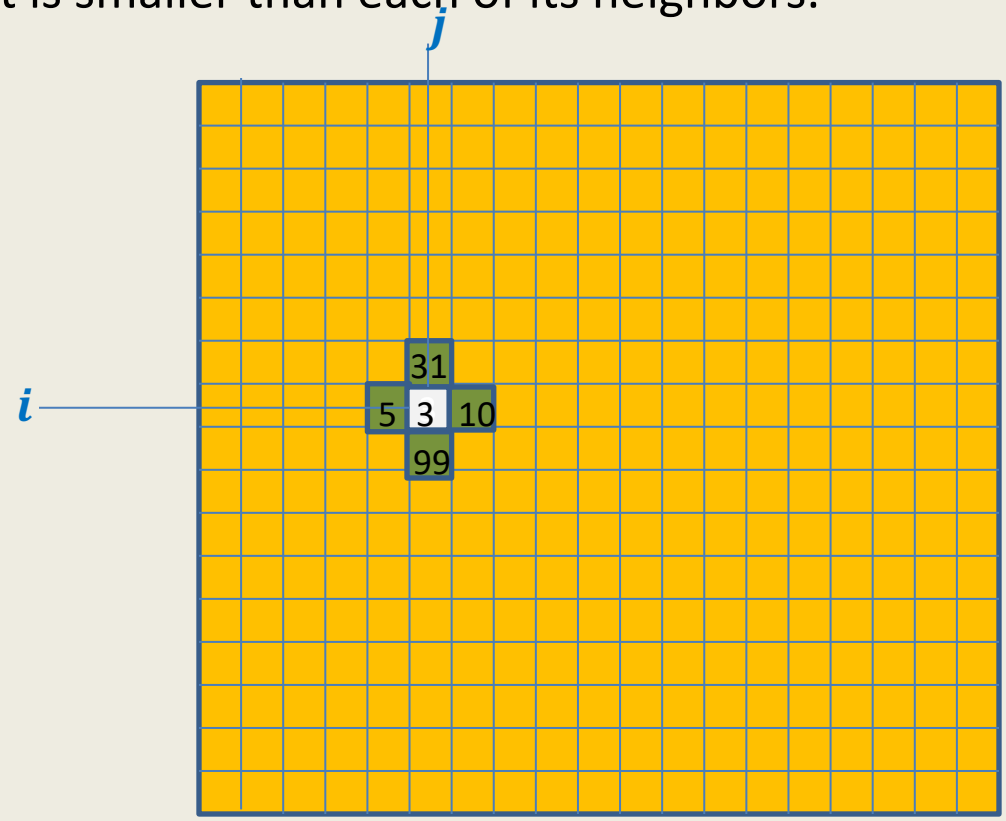
Assertion P(i) : **S**[i] stores the sum of maximum sum subarray ending at **A**[i].

Homework: Prove that **P**(i) holds for all $i \leq n-1$

LOCAL MINIMA IN A GRID

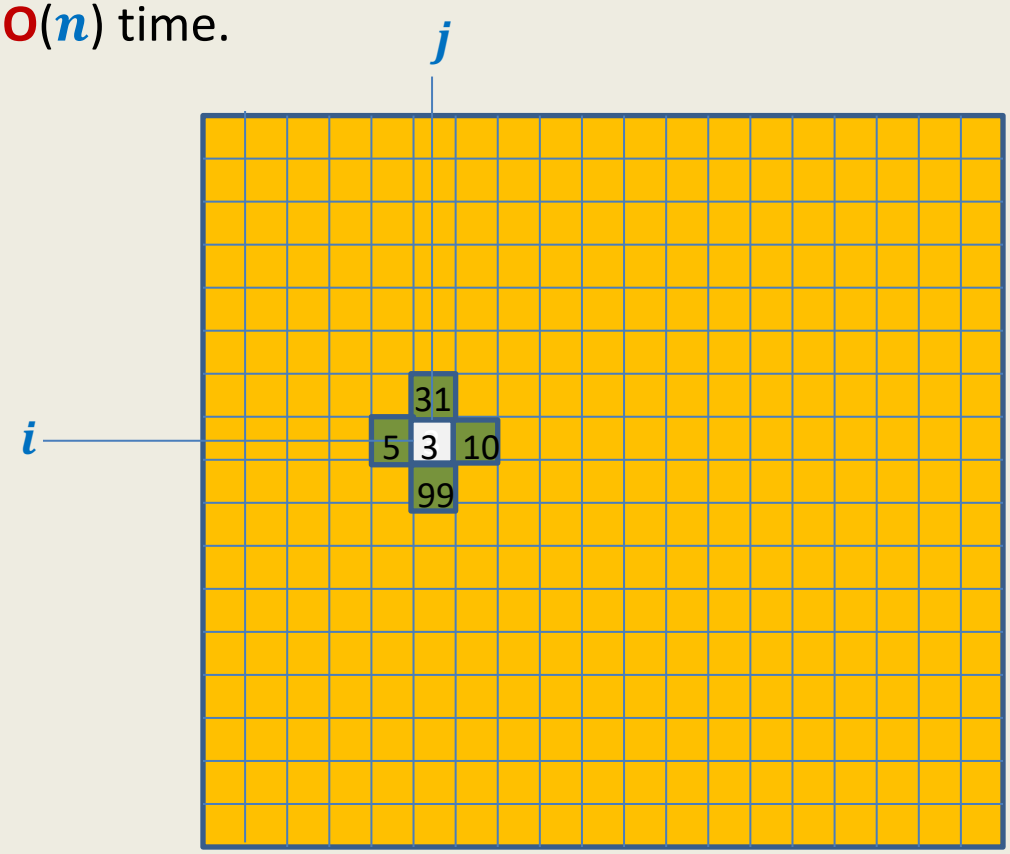
Local minima in a grid

Definition: Given a $n \times n$ grid storing distinct numbers, an entry is local minima if it is smaller than each of its neighbors.



Local minima in a grid

Problem: Given a $n \times n$ grid storing distinct numbers, output any local minima in $O(n)$ time.



Two simple principles

1. Respect every new idea which solves a problem even partially.

2. Principle of simplification:

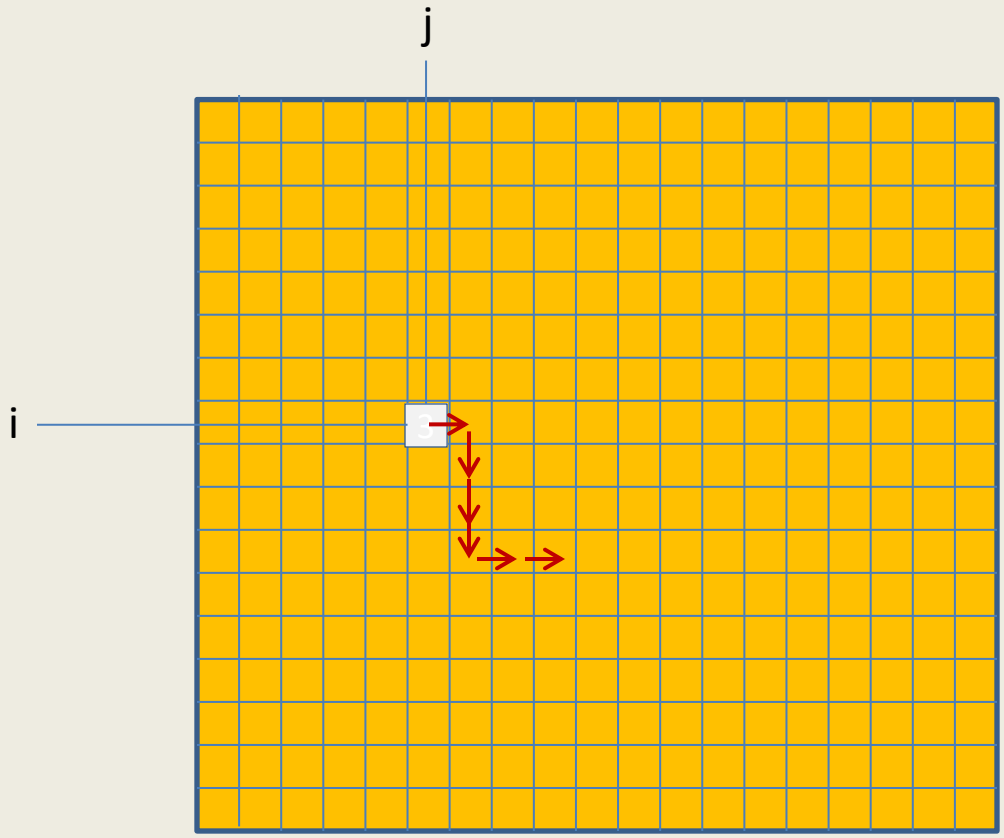
If you find a problem difficult,

➔ try to solve its simpler version, and then

➔ extend this solution to the original (difficult) version.

A new approach

Repeat : *if current entry is not local minima, explore the neighbor storing smaller value.*



A new approach

```
Explore()
{    Let c be any entry to start with;
     While(c is not a local minima)
     {
        c ← a neighbor of c storing smaller value
     }
     return c;
}
```

A new approach

```
Explore()
{    Let c be any entry to start with;
     While(c is not a local minima)
     {
        c ← a neighbor of c storing smaller value
     }
     return c;
}
```

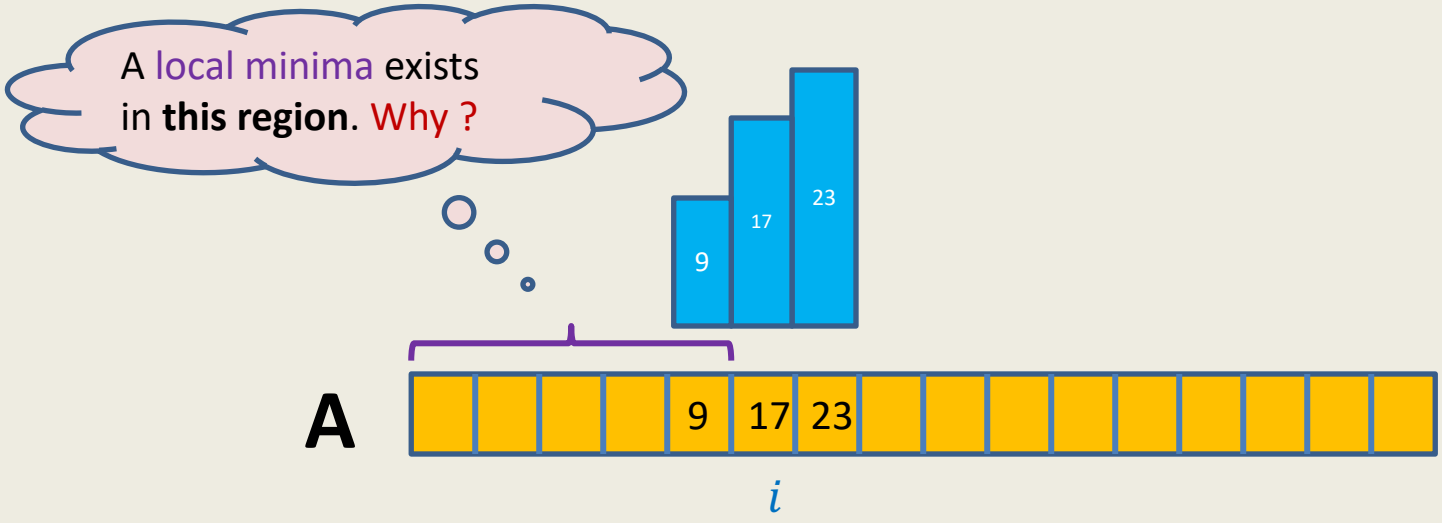
Worst case time complexity : **O**(n^2)

How to apply this principle ?

First principle:
Do not discard **Explore**()

Second principle:
Simplify the problem

Local minima in an array



Theorem: There is a local minima in **A**[0,…, $i-1$].

Proof: Suppose we execute **Explore()** from **A**[$i-1$].

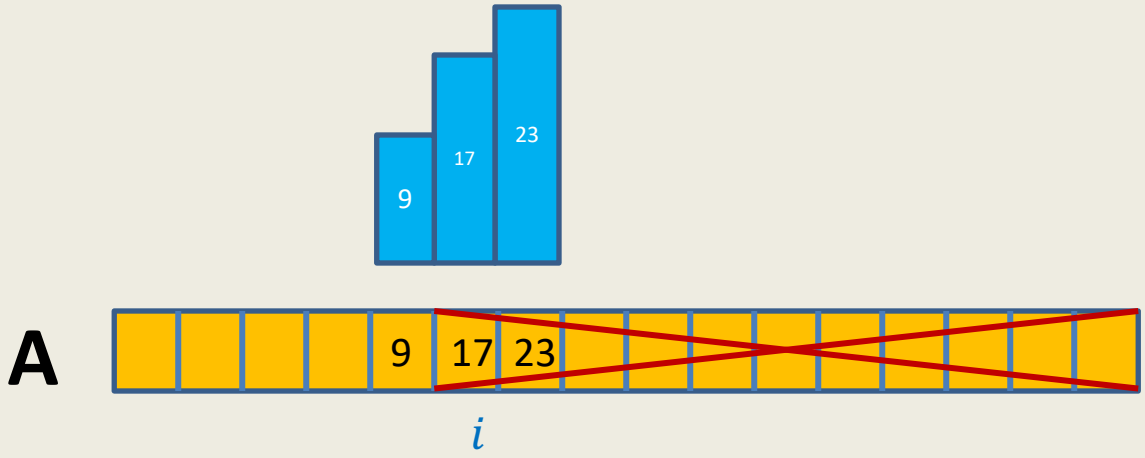
Explore(), if terminates, will return local minima.

It will terminate without ever entering **A**[i,…, $n-1$].

Hence there is a local minima in **A**[0,…, $i-1$].

Algorithmic proof

Local minima in an array



Theorem: There is a local minima in **A**[0,…, $i-1$].

➔ We can confine our search for local minima to only **A**[0,…, $i-1$].

➔Our problem size has reduced.

Question: Which i should we select so as to reduce problem size significantly ?

Answer: $middle$ point of array **A.**

Local minima in an array

(Similar to binary search)

```
int Local-minima-in-array(A) {
    L ← 0;
    R ← n − 1;
    found ← FALSE;
    while( not found )
    {
        mid ← (L + R)/2;
        If (mid is a local minima)
            found ← TRUE;
        else if(A[mid + 1] < A[mid])   L ← mid + 1
            else                       R ← mid − 1
    }
    return mid; }
```

How many iterations ?

O(log n)

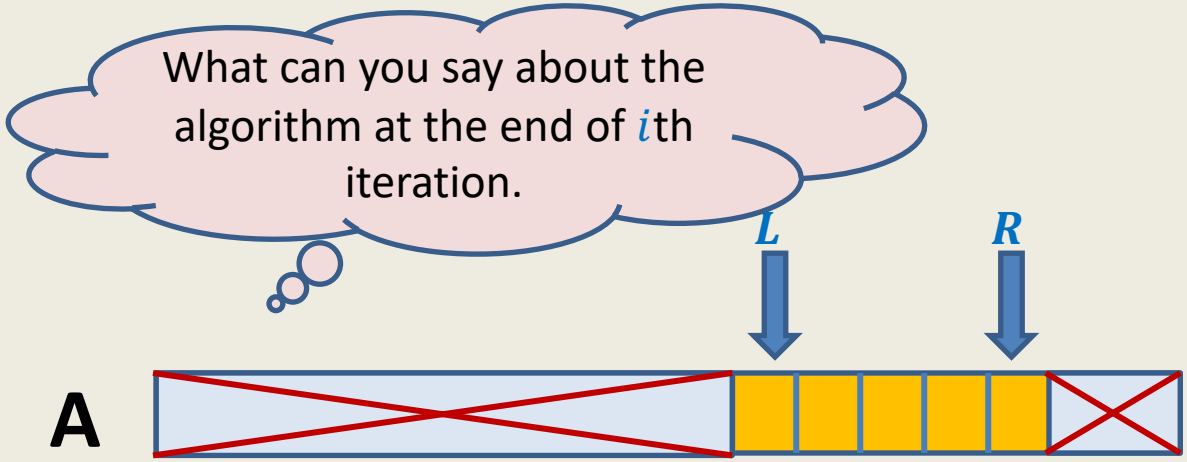
O(1) time in one iteration

➔ Running time of the algorithm = O(log n)

Proof of correctness ?

Local minima in an array

(Proof of correctness)

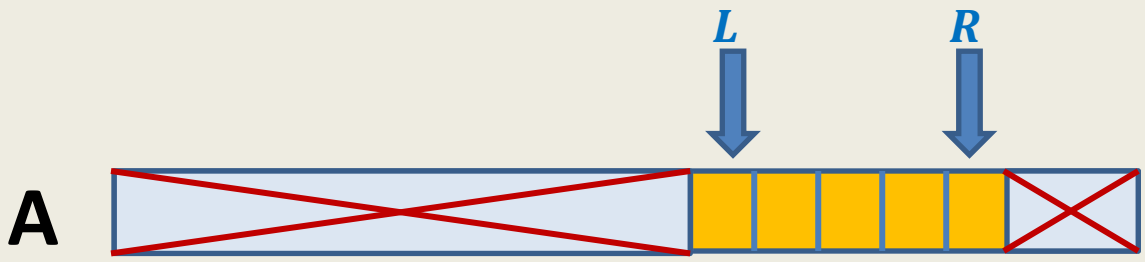


P(i) : At the end of ith iteration,

"A local minima of array **A** exists in **A**[L,…, R]."

Local minima in an array

(Proof of correctness)



P(i) : At the end of ith iteration,

"A local minima of array **A** exists in **A**[L,…, R]."

=

"**A**[L]< **A**[$L-1$]" and "**A**[R]< **A**[$R+1$]".

Homework:

- Make sincere attempts to prove the assertion **P**(i).
- How will you use it to prove that **Local-minima-in-array(A)** outputs a local minima ?

Local minima in an array

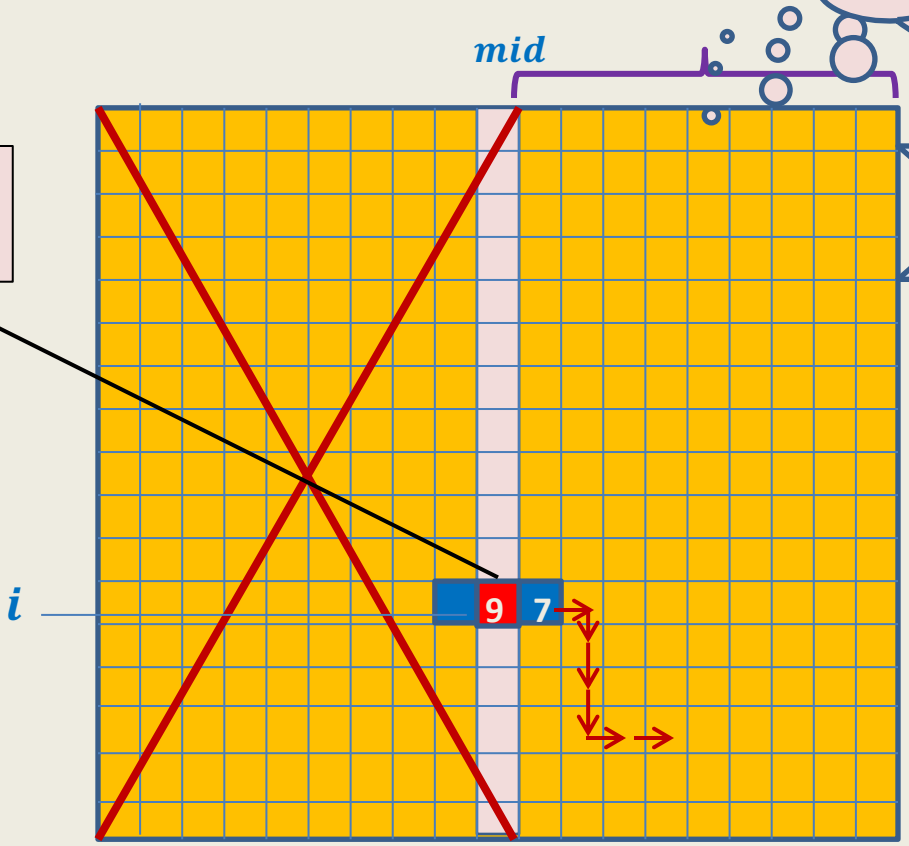
Theorem: A local minima in an array storing n distinct elements can be found in **O**(**log** n) time.

Local minima in a grid

(extending the solution from 1-D to 2-D)

Search for a local minima in the column **M**[$*$, mid]

Under what circumstances even this smallest element is not a local minima ?



Execute **Explore()** from **M**[$i, mid + 1$]

Homework:

Use this idea to design an **O**(n **log** n) time algorithm for this problem.

… and do not forget to prove its correctness ☺.

Make sincere attempts to
answer all questions raised in this lecture.

Data Structures and Algorithms

(ESO207)

Lecture 6:

- Design of **O**(n) time algorithm for **Local Minima in a grid**
- Data structure gem: **Range minima Problem**

Local minima in an array

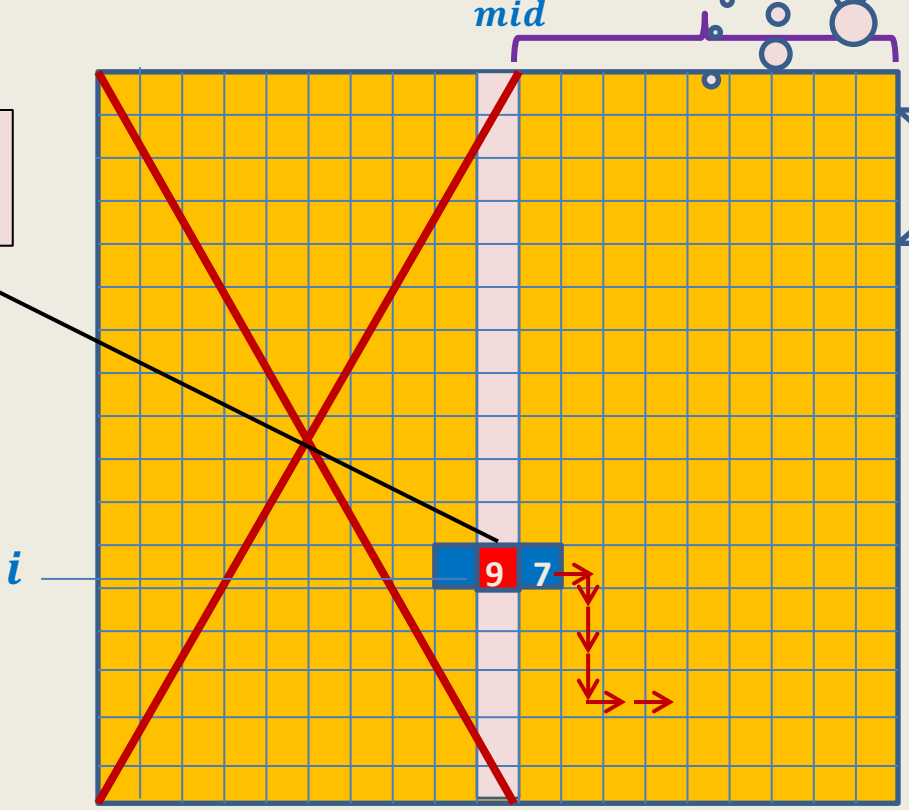
Theorem: A local minima in an array storing n distinct elements can be found in **O**(**log** n) time.

Local minima in a grid

(extending the solution from 1-D to 2-D)

Search for a local minima in the column **M**[$*$, mid]

Under what circumstances even this smallest element is not a local minima ?



Execute **Explore()** from **M**[i, $mid + 1$]

Homework:

Use this idea to design an **O**(n **log** n) time algorithm for this problem.

… and do not forget to prove its correctness ☺.

Local minima in a grid

Int Local-minima-in-grid(M) // returns the column containing a local minima

```
{       L ← 0;
        R ← n − 1;
       found ← FALSE;
       while(not found)
       {       mid ← (L + R)/2;
               If (M[*, mid] has a local minima)    found ← TRUE;
                  else {
                          let M[k, mid] be the smallest element in M[*, mid]
                          if(M[k, mid + 1] < M[k, mid] L← mid + 1    ;
                          else  R← mid − 1
                       }
        }
        return mid;
}
```

O(n) time

O(n) time

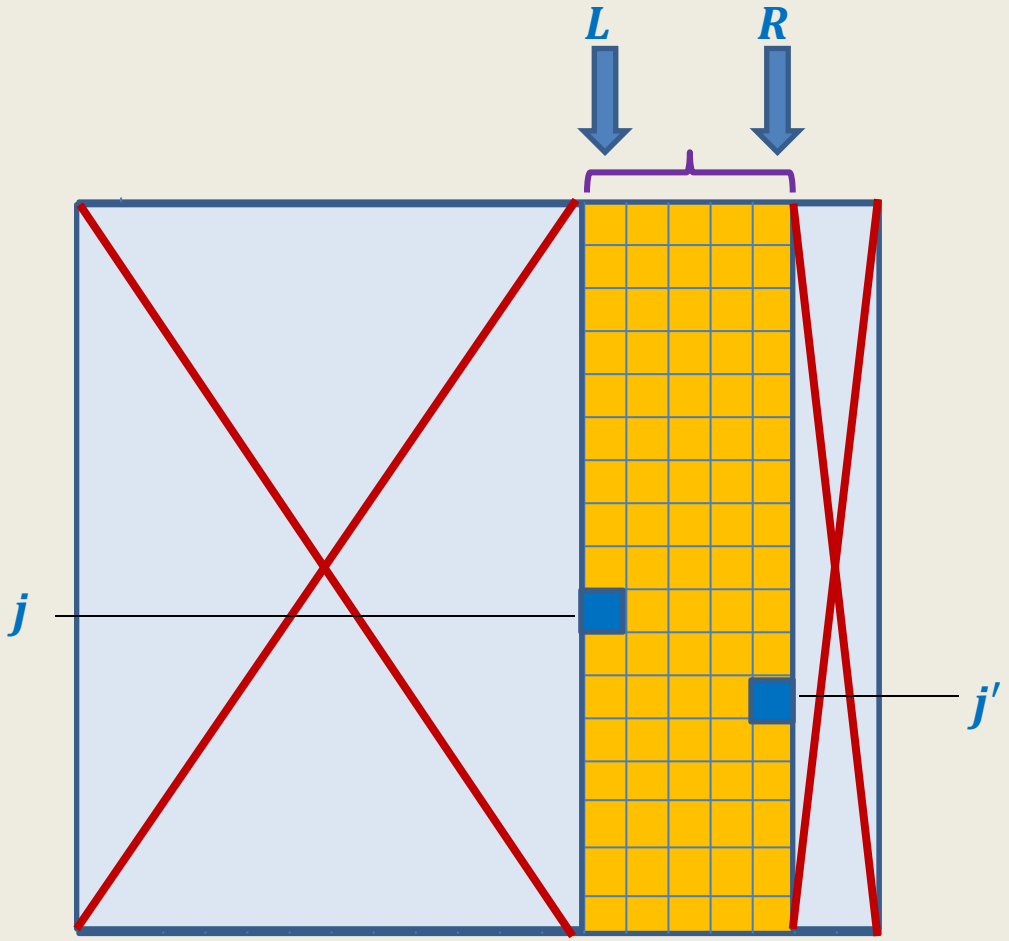
O(n) time

➔ Running time of the algorithm = O(n log n)

Proof of **correctness** ?

Local minima in a grid

(Proof of Correctness)



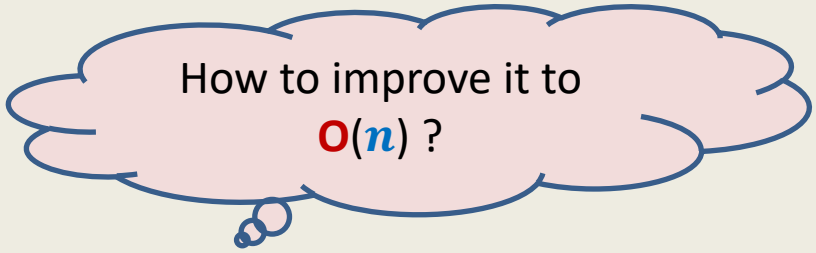
P(i) :

"A local minima of grid **M** exists in **M**[L,…, R]."

$\exists\ j$ such that **M**[j, L] < **M**[$*, L-1$]" **and** $\exists\ j'$ such that **M**[j', R] < **M**[$*, R+1$]"

Local minima in a grid

Theorem: A local minima in an $n \times n$ grid storing distinct elements can be found in **O**(n **log** n) time.



Local minima in a grid in O(n) time

Let us carefully look at the calculations of the running time of the current algo.

$cn + cn + cn +$ … (**log** n terms) … $+ cn$ = O(n **log** n)

What about the following series

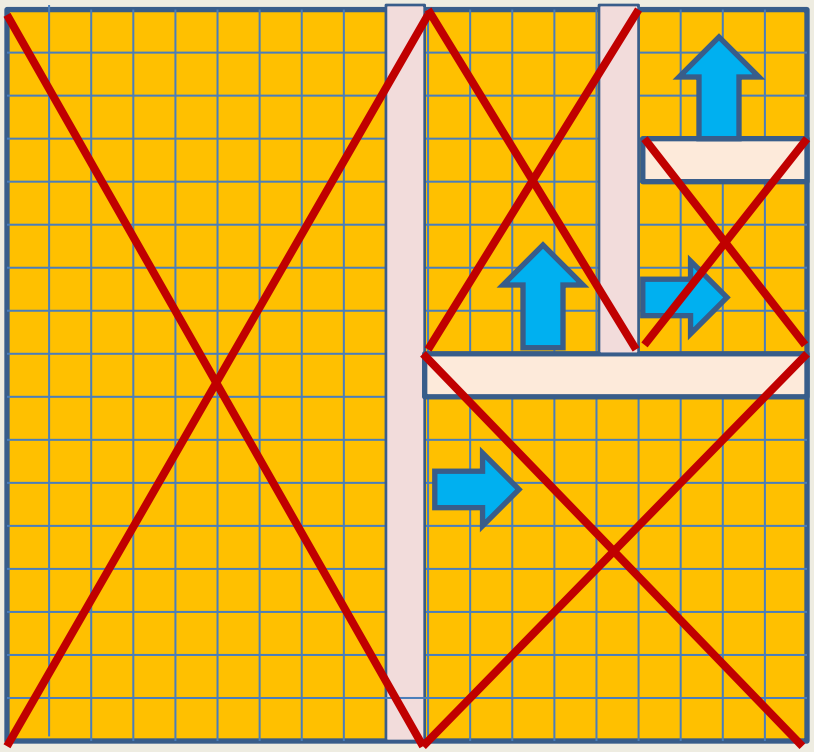
$c\frac{n}{2} + c\frac{n}{4} + c\frac{n}{8} +$ … (**log** n terms) … $+ cn$ = ?

It is $2cn$ = O(n) .

Get an !DEA from this series to modify our current algorithm

Local minima in a grid in $O(n)$ time

Bisect alternatively along rows and column



INCORRECT

Exercise: Think of a counterexample!

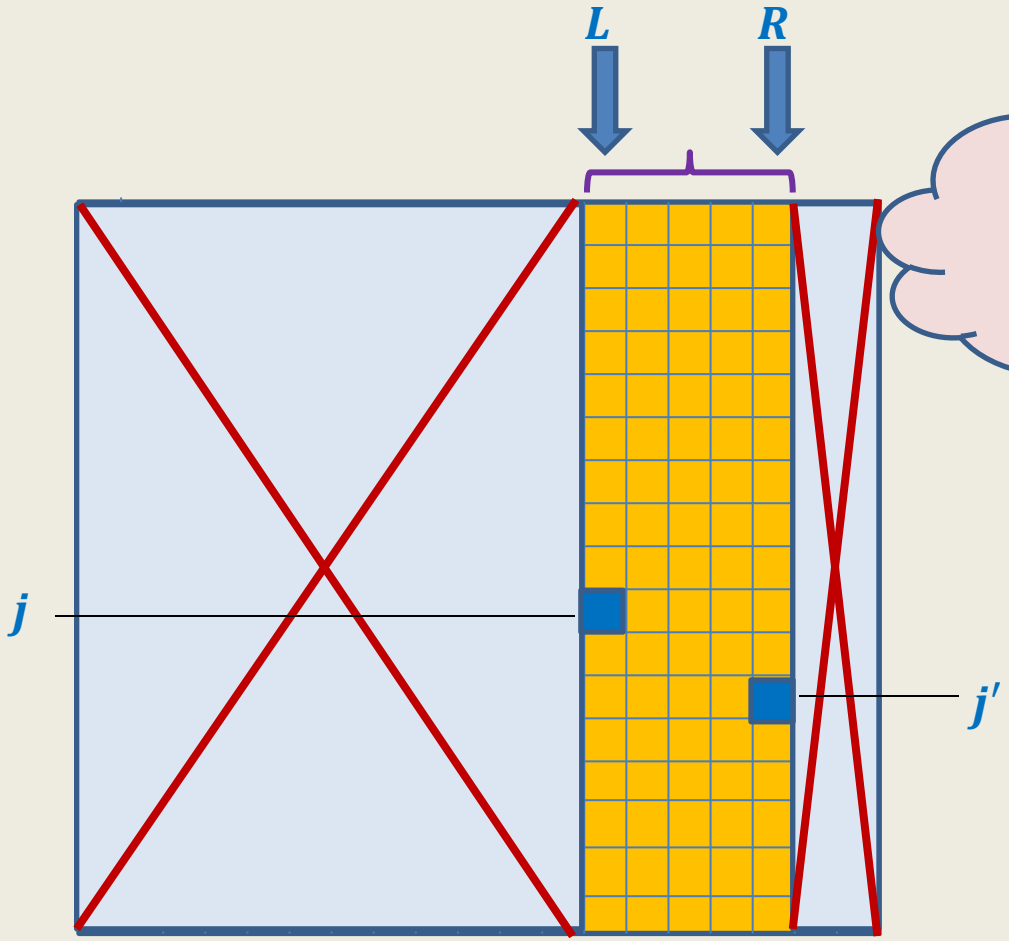
Lessons learnt

- No hand-waving works for iterative algorithms ☺
- We must be sure about
  - What is **P**(i)
  - Proof of **P**(i).

Let us revisit the ($n \log n$) algorithm

Local minima in a grid

(Proof of Correctness)



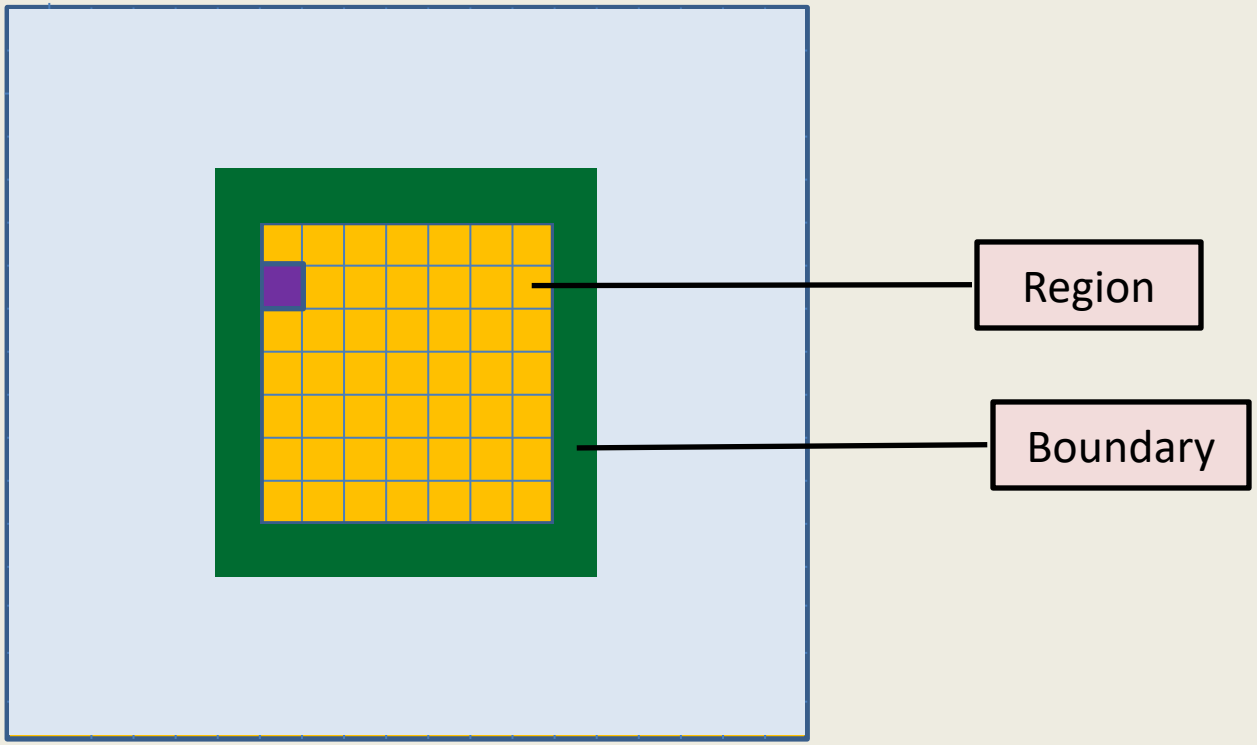
Is there an alternate and simpler **P**(i) ?

P(i) :

"A local minima of grid **M** exists in **M**[L,..., R]."

$\exists\ j$ such that **M**[j, L] < **M**[$*, L-1$]" **and** $\exists\ j'$ such that **M**[j', R] < **M**[$*, R+1$]"

At any stage, what guarantees the existence of local minima in the region ?

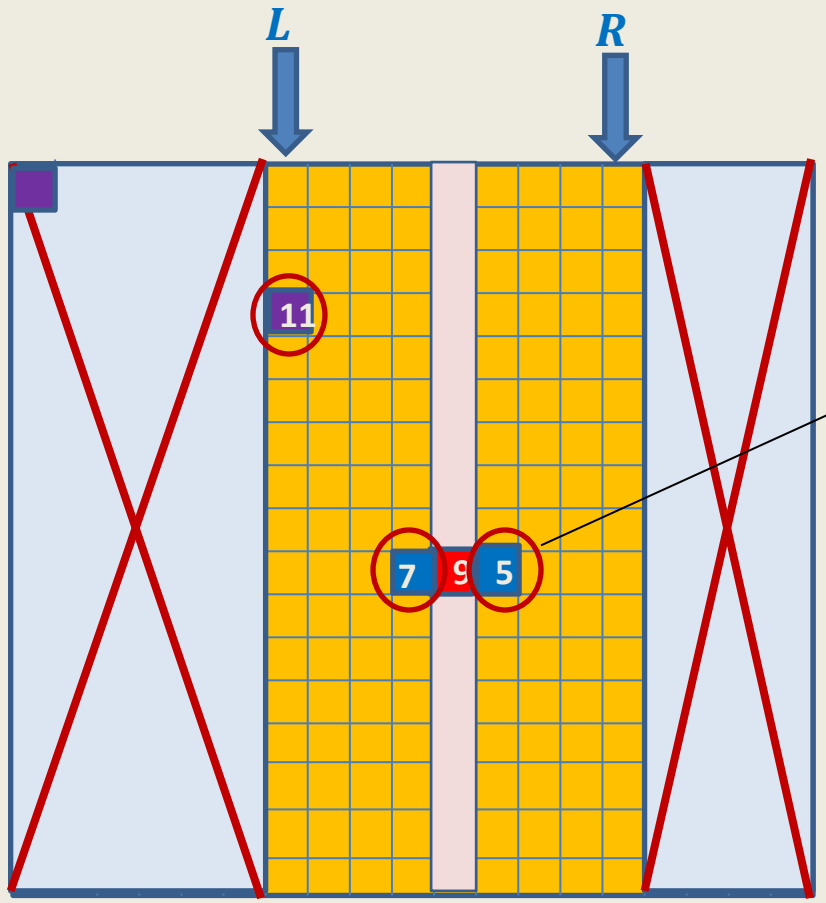


- At each stage, our algorithm may maintain a cell in the region whose value is smaller than all elements lying at the **boundary** of the **region** ?

(Note the boundary lies outside the region)

Local minima in a grid

Alternate O(n log n) algorithm)

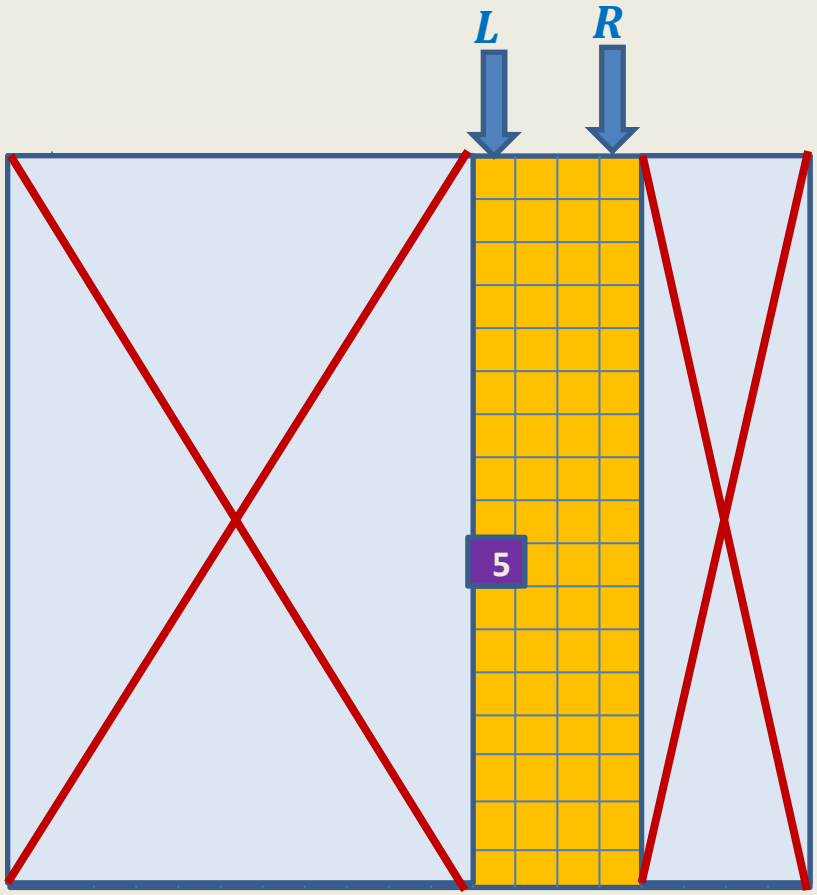


How will we maintain it after each iteration ?

smallest

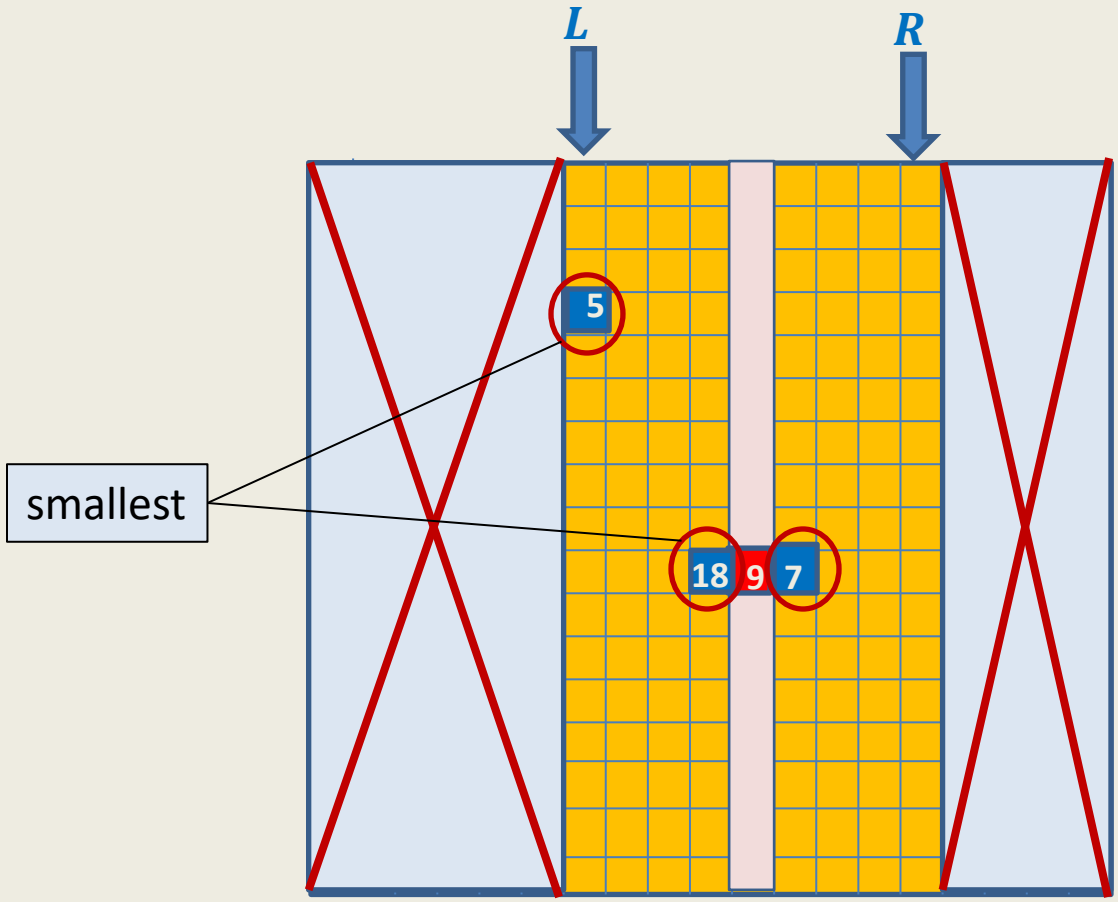
Local minima in a grid

Alternate O($n \log n$) algorithm



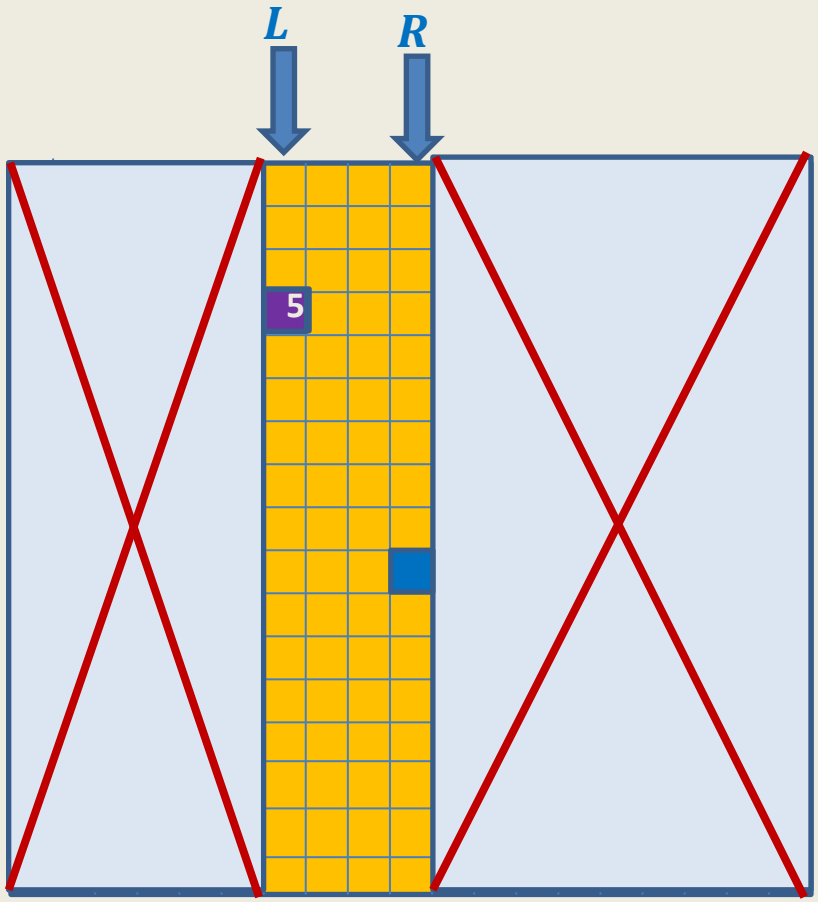
Local minima in a grid

Alternate O($n \log n$) algorithm



Local minima in a grid

Alternate O($n \log n$) algorithm



- A neat pseudocode of this algorithm is given on the following slide.

Local minima in a grid

Alternate O($n \log n$) algorithm

Int Local-minima-in-grid(M) // returns the column containing a local minima

```
{       L ← 0;
        R ← n − 1;
        found ← FALSE;
        C ←(0,0);
        while(not found)
        {       mid ← (L + R)/2;
                If (M[*, mid] has a local minima)    found ← TRUE;
                   else {
                                let M[k, mid] be the smallest element in M[*, mid]
                                C' ←(k,mid − 1);
                                C'' ←(k,mid + 1);
                                C_min ← the cell containing the minimum value among {C, C', C''};
                                If (column of C_min is present in (mid + 1, R))    L ← mid + 1;
                                  else R ← mid − 1;
                                C ← C_min ;
                        }
        }
        return mid;
}
```

Extending it to O(n) is easy now.
Do it as a homework.

Homework: Prove the correctness of this algorithm.

Theorem:

Given an $n \times n$ grid storing n^2 distinct elements, a local minima can be found in **O**(n) time.

Question:

On which algorithm paradigm, was this algorithm based on ?

- **Greedy**
- **Divide and Conquer**
- **Dynamic Programming**

Proof of correctness of algorithms

Worked out examples:

- **GCD**
- **Binary Search**

GCD

GCD(a,b) // $a \geq b$

{

  while (b <> 0)

  { $t \leftarrow b$;

    $b \leftarrow a$ mod b ;

    $a \leftarrow t$

  }

  return a;

}

Lemma (Euclid):

If $n \geq m > 0$, then

gcd(n,m) = **gcd**(m,n mod m)

Proof of correctness of GCD(a,b) :

Let a_i : the value of variable a after ith iteration.

b_i : the value of variable b after ith iteration.

Assertion $P(i)$: **gcd**(a_i, b_i) = **gcd**(a,b)

Theorem : $P(i)$ holds for each iteration $i \geq 0$.

Proof: (By induction on i).

Base case: ($i = 0$) hold trivially.

Induction step:

(Assume $P(j)$ holds, show that $P(j+1)$ holds too)

$P(j)$ ➔ **gcd**(a_j, b_j) = **gcd**(a,b). ----(1)

$(j+1)$ iteration ➔ a_{j+1}= b_j **and** b_{j+1}= a_j mod b_j ---(2)

Using **Euclid**'s Lemma and (2),

gcd(a_j, b_j) = **gcd**(a_{j+1}, b_{j+1}) ------(3).

Using (1) and (3),assertion $P(j+1)$ holds too.

Binary Search

```
Binary-Search(A[0...n − 1], x)
L ← 0;
R ← n − 1;
Found ← false;
While ( L ≤ R and Found = false )
{    mid ← (L+R)/2;
     If (A[mid] = x) Found ← true;
     else if (A[mid] < x)   L ← mid + 1 ;
          else              R ← mid - 1
}
if Found return true;
else return false;
```

Observation: If the code returns true, then indeed **output** is correct.

So all we need to prove is that whenever code returns false , then indeed x is not present in **A**[].

This is because Found is set to true only when x is indeed found.

Binary Search

Binary-Search(A[$0 \ldots n-1$], x)

L ← 0;

R ← $n-1$;

Found ← false;

While (L ≤ R and Found = false)

{ mid ← (L+R)/2;

If (A[mid] $= x$) Found ← true;

else if (A[mid] $< x$) L ← mid + 1 ;

else R ← mid - 1

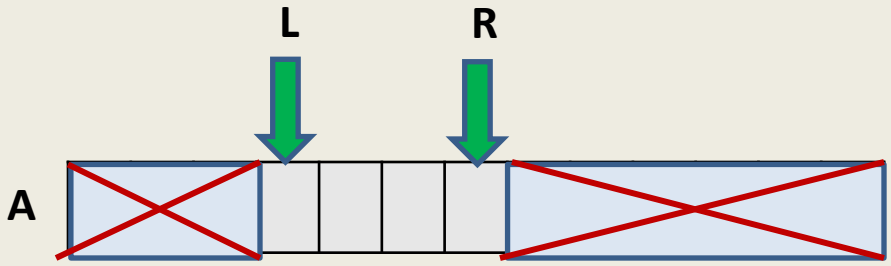
}

if Found return true;

else return false;

Assertion $P(i)$: $x \notin$ { A[0],…,A[L−1] }

and $x \notin$ { A[R+1],…,A[$n-1$] }



Range-Minima Problem

A Motivating example

to realize the importance of data structures

Range-Minima Problem

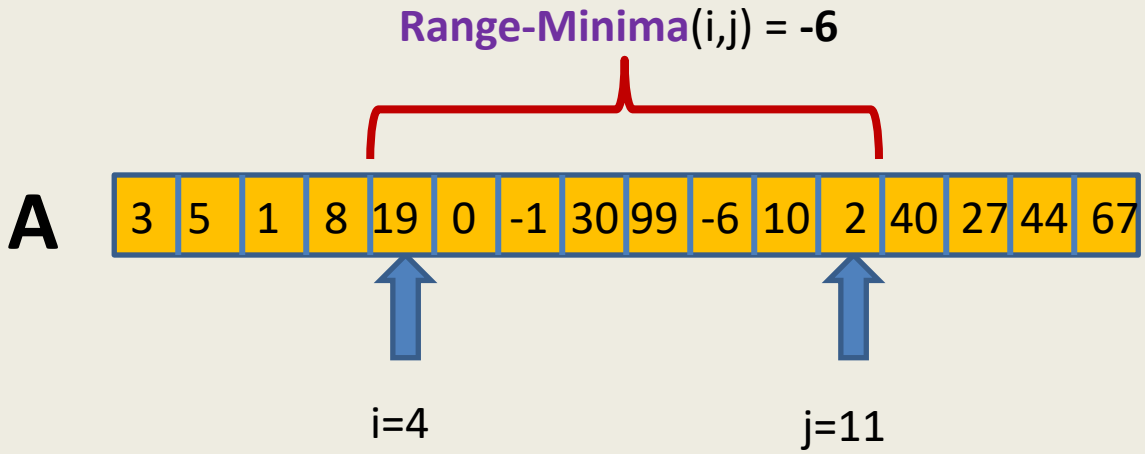
Given: an array **A** storing n numbers,

Aim: a data structure to answer a sequence of queries of the following type

Range-minima(i,j) : report the smallest element from **A**[i],…,**A**[j]

Let **A** store one **million** numbers

Let the number of queries be **10 millions**



Range-Minima Problem

Question: Does there exist a data structure which is

- **Compact**
  (**O**(n log n) **size**)
- **Can answer each query efficiently ?**
  (**O**(1) time)

Homework : Ponder over the above question.

(we shall solve it in the next class)

Data Structures and Algorithms

(ESO207)

Lecture 7:

- **Data structure for Range-minima problem**

  Compact and fast

Data structures

AIM:

To organize a data in the memory

so that any query can be answered efficiently.

Example:

Data: A set **S** of n numbers

Query: "Is a number x present in **S** ?"

A trivial solution: **sequential search**

O(n) time per query

A Data structure solution:

- Sort **S**

O(n **log** n) time to build sorted array.

- Use **binary search** for answering query

O(**log** n) time per query

Data structures

AIM:

To organize a data in the memory

so that any query can be answered efficiently.

Important assumption:

No. of queries to be answered will be **many**.

Parameters of Efficiency

- Query time
- Space
- Preprocessing time

RANGE-MINIMA Problem

Range-Minima Problem

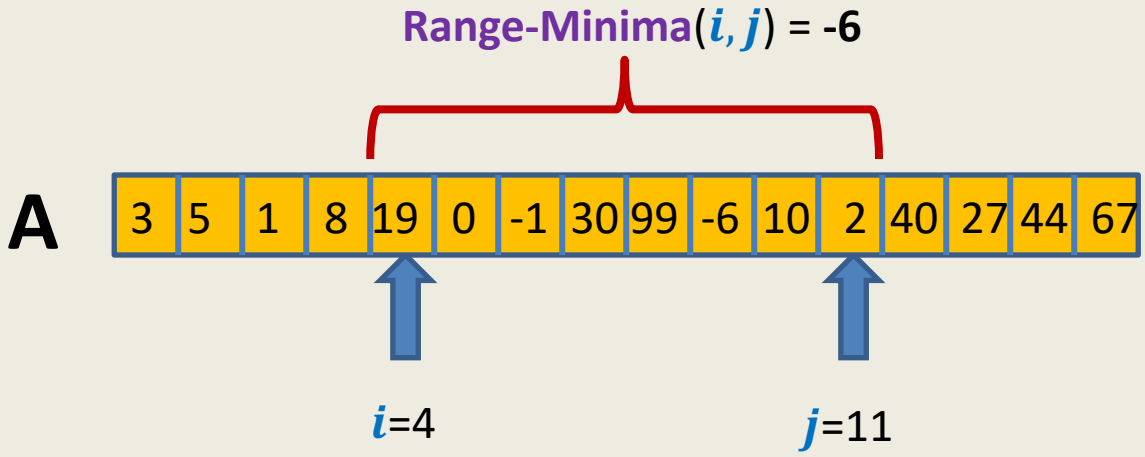
Given: an array **A** storing n numbers,

Aim: a data structure to answer a sequence of queries of the following type

Range-minima(i, j) : report the smallest element from **A**[i],…,**A**[j]

Let **A** store one **million** numbers

Let the number of queries be **10 millions**



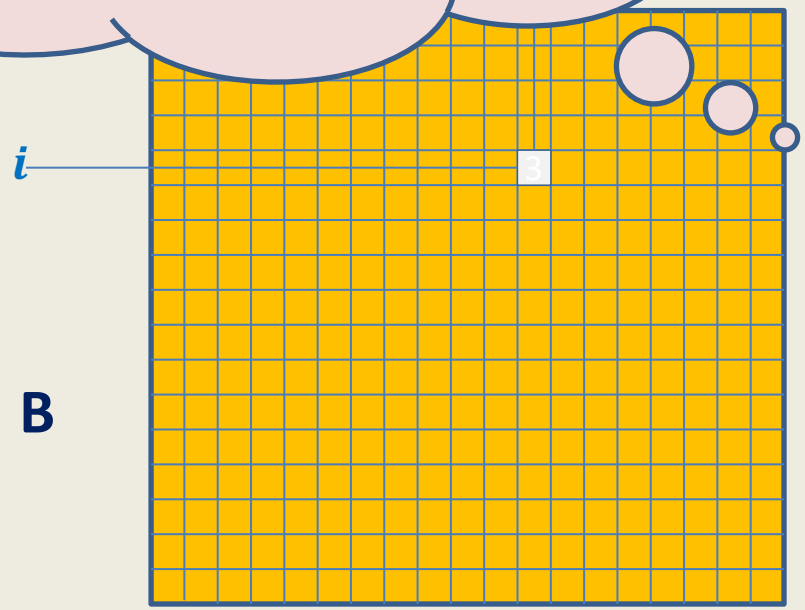
Range-Minima Problem

Solution 1

(brute force)

Size of **B** is **too large** to be kept in RAM.
So we shall have to keep most of it in the **Hard disk drive.**
Hence it will take a few **milliseconds per query**.

```
Range-minima-trivial(i, j)
{   temp ← i + 1;
    min ← A[i];
     While(temp <= j)
     {   if (min > A[temp])
             min ← A[temp];
         temp← temp+1;
    }
    return min
}
```



Time complexity for one query: O(n)

(a few **hours** for 10 million queries)

Space : O(n^2)

Impractical

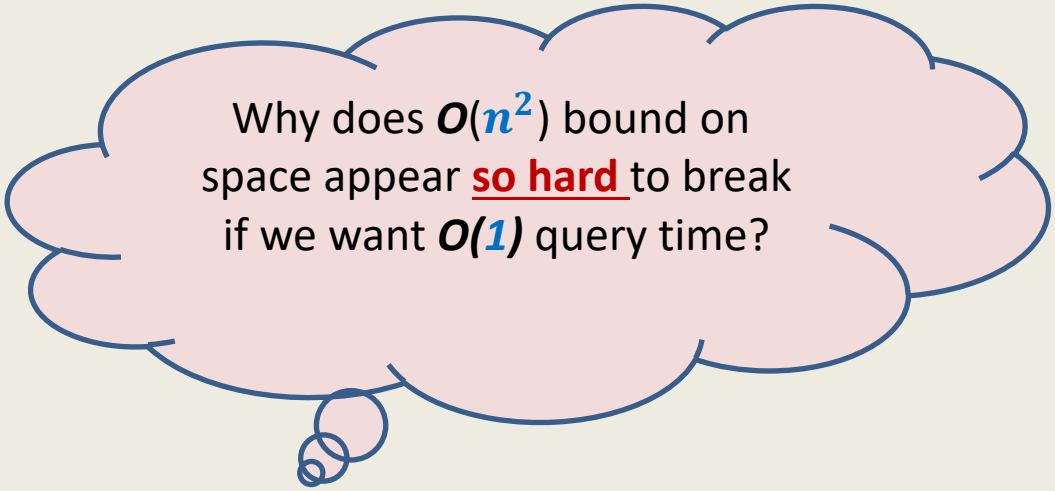
Range-Minima Problem

Query:

Report_min(**A**,i, j) : report smallest element from {**A**[i],…,**A**[j]}

Aim :

- **compact** data structure
- **O(1) Query time** for any $1 \le i < j \le n$.



... Because of **artificial hurdles**

Artificial hurdle

If we want to answer each query in O(1) time,

➔ we must store its answer explicitly.

➔ Since there are around $O(n^2)$ queries,
so $O(n^2)$ space is needed.

Spend some time to find the origin of this hurdle….

Artificial hurdle

… If we fix the first parameter i for all queries, we need $O(n)$ space.

True Fact

➔

for all i, we need $O(n^2)$ space.

A wrong inference

because it assumes that data structure for an index i will work in **total isolation** of others.

Collaboration (team effort)
works in real life

Why not try
collaboration for the
given problem ?

Range-minima problem:
Breaking the $O(n^2)$ barrier using collaboration

An Overview:

- Keep n tiny data structures:

  Each index i stores minimum **only for a few** $j > i$.

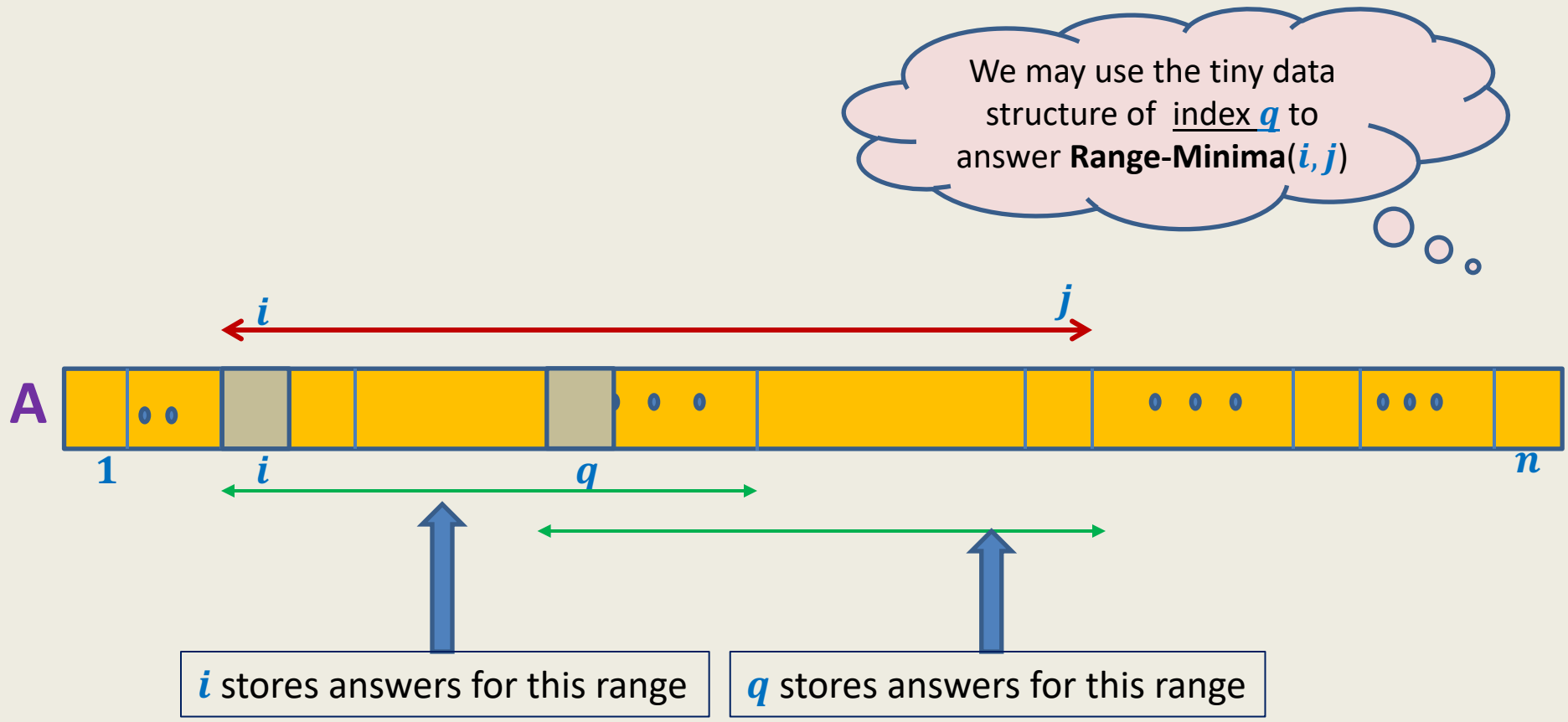
- For a query **Range-minima**(i, j),

  if the answer is not stored in the tiny data structure of i,

  look up tiny data structure of some index q (chosen carefully).

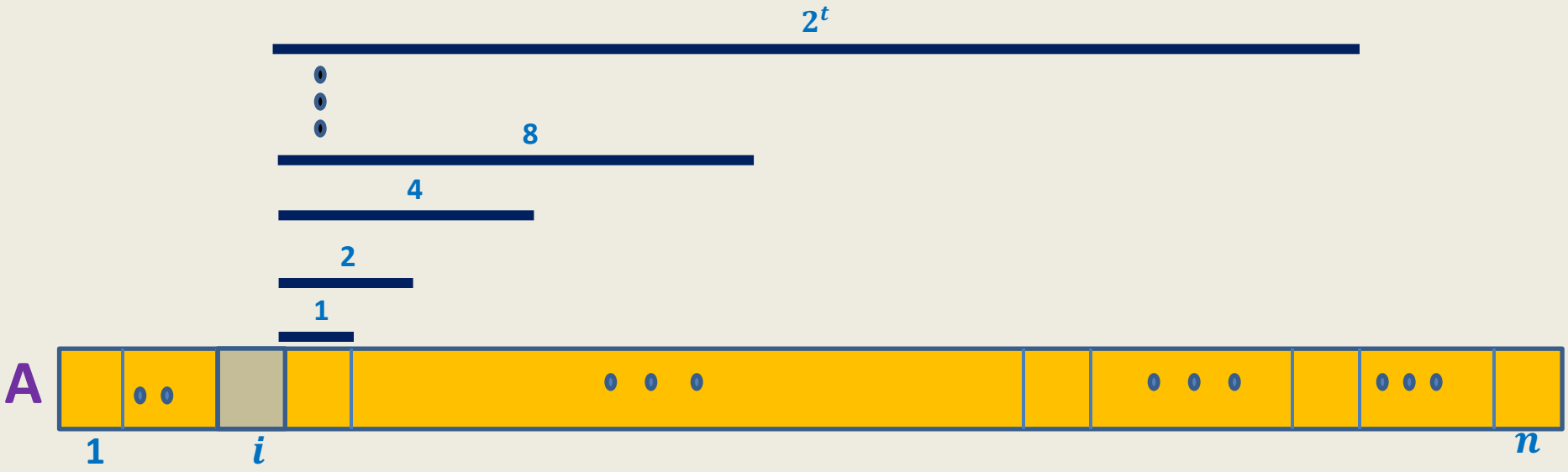
HOW DOES COLLABORATION WORK IN THIS PROBLEM ?

Range-minima problem:
Breaking the $O(n^2)$ barrier using collaboration



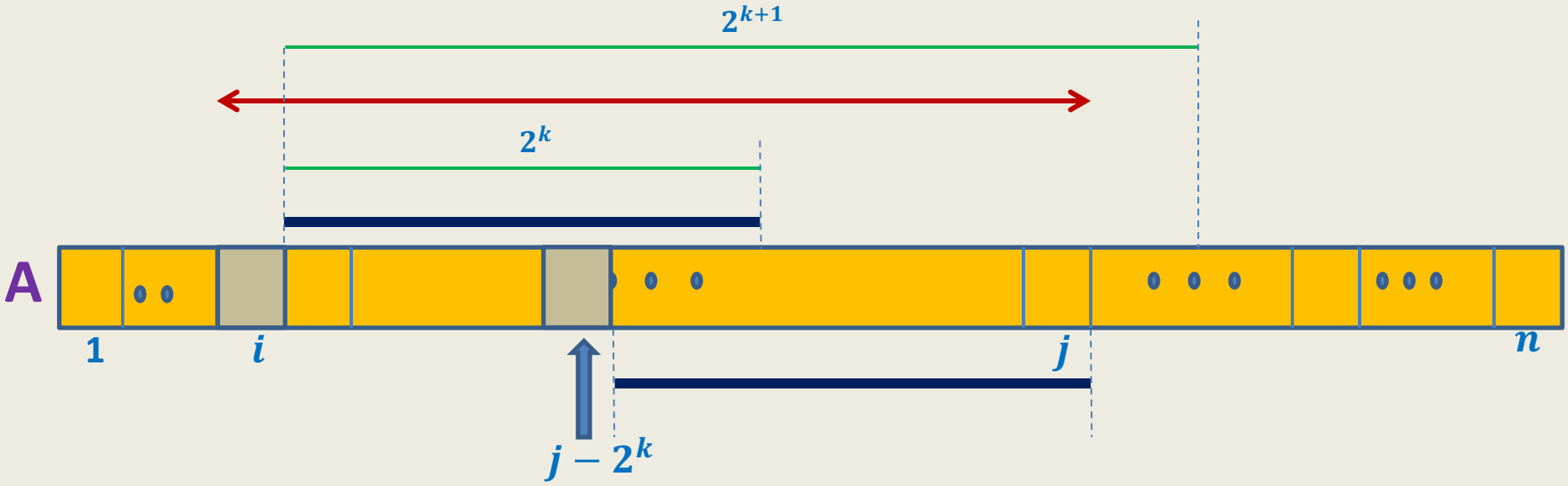
DETAILS OF TINY DATA STRUCTURES

Range-minima problem :
Details of tiny data structure stored at each i



Tiny data structure of Index i stores minimum element for {A[i],…,A[$i + 2^k$]} for each $k \leq \log_2 n$

Answering Range-minima query for index i :
Collaboration works



We shall use two additional arrays

Definition :

Power-of-2[m] : the greatest number of the form 2^k such that $2^k \leq m$.

Examples: **Power-of-2**[5] = 4,

Power-of-2[19]= 16,

Power-of-2[32]=32.

Definition :

Log[m] : the greatest integer k such that $2^k \leq m$.

Examples: **Log**[5] = 2,

Log[19]= 4,

Log[32]=5.

Homework: Design **O**(n) time algorithm to compute arrays **Power-of-2**[] and **Log**[] of size n.

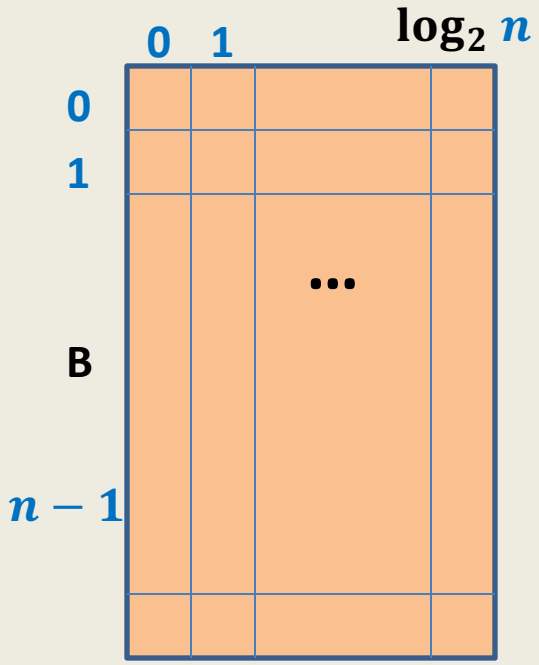
FINAL SOLUTION FOR
RANGE MINIMA PROBLEM

Range-Minima Problem:

Data structure with O(n log n) space and O(1) query time

Data Structure:

- n × log n matrix **B** where **B[i][k]** stores minimum of {**A**[i],**A**[i +1],…, **A**[i +2^k]}
- Array **Power-of-2[]**
- Array **Log[]**



Range-minima-(i, j)

```
{   L ← j − i;
    t ← Power-of-2[L];
    k ← Log[L];
    If (t = L) return B[i][k];
    else       return min( B[i][k] , B[j − t][k]; );
}
```

Theorem:

There is a data structure for range-minima problem that takes

O$(n$ **log** $n)$ **space** and **O**(1) **query time.**

Preprocessing time:

O$(n^2$ **log** $n)$: Trivial

O$(n$ **log** $n)$: Doable with little hints

Homework:

Design an **O**(n **log** n) time algorithm

to build the n **×** **log** n matrix **B** used in data structure of Range-Minima problem.

Hint: (Inspiration from iterative algorithm for Fibonacci numbers).

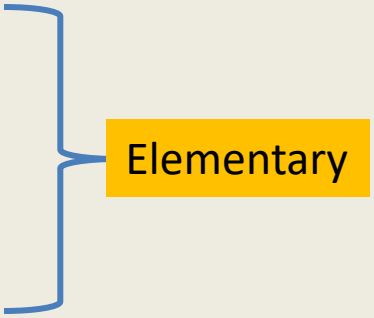
Spend some time before looking at the *more explicit hint* below. (it is just a click away)…You can do it…

To compute **B**[i][k], you need to know only two entries from column $**$.

Data structures

(To be discussed in the course)

- Arrays
- Linked Lists
- Stacks
- Queues



Tree Data Structures:

- Binary heap
- Binary Search Trees
- Augmented Data structures

Data Structures for integers:

- Hash Tables
- Searching in **O**(**log log** n) time (if time permits)

Data Structures and Algorithms

(ESO207)

Lecture 8:

Data structures:

- **Modeling** versus **Implementation**
- Abstract data type "**List**" and its implementation

Data Structure

Definition:

A collection of data elements *arranged* and *connected* in a way that can facilitate efficient executions of a (potentially long) sequence of operations.

Two steps process for designing a Data Structure

Step 1: Mathematical Modeling

A **Formal** description of the possible operations of a data structure.

Operations can be classified into two categories:

Query Operations: Retrieving some information from the data structure

Update operations: Making a change in the data structure

Outcome of Mathematical Modeling: an **Abstract Data Type**

Step 2: Implementation

Explore the ways of organizing the data that facilitates performing each operation efficiently using the exis

Since we don't specify here the way how each operation of the data structure will be implemented

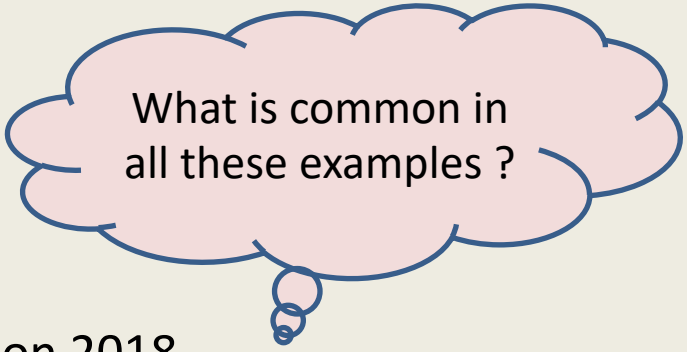
MODELING OF LIST

OUTCOME WILL BE:

ABSTRACT DATA TYPE “LIST”

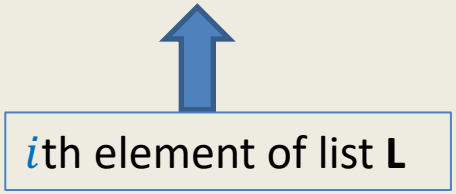
Mathematical Modeling of a List

- List of Roll numbers passing a course.
- List of Criminal cases pending in High Court.
- List of Rooms reserved in a hotel.
- List of Students getting award in IITK convocation 2018.



Inference: List is a **sequence** of elements.

L: $a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$



Query Operations on a List

- **IsEmpty(L)**: determine if **L** is an empty list.
- **Search(x,L)**: determine if **x** appears in list **L**.
- **Successor(p,L)**:

  return the element of list **L** which succeeds/follows the element at *location* **p**.

  Example:

  If **L** is $a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$ and **p** is location of element a_i,

  then **Successor(p,L)** returns a_{i+1}.
- **Predecessor(p,L)**:

  Return the element of list **L** which precedes (appears before) the element at location **p**.

The type of this parameter will depend on the implementation

Other possible operations:

- **First(L)**: return the first element of list **L**.
- **Enumerate(L)**: Enumerate/print all elements of list **L** in the order they appear.

Update Operations on a List

- **CreateEmptyList**(**L**): Create an empty list.
- **Insert**(**x,p,L**): Insert **x** at a given location **p** in list **L**.

  Example: If **L** is $a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$ and **p** is location of element a_i, then after **Insert**(**x,p,L**), **L** becomes

  $$a_1, \ldots, a_{i-1}, \mathbf{x}, a_i, a_{i+1}, \ldots, a_n$$

- **Delete**(**p,L**): Delete element at location **p** in **L**

  Example: If **L** is $a_1, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$ and **p** is location of element a_i, then after **Delete**(**p,L**), **L** becomes

  $$a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n$$

- **MakeListEmpty(L):** Make the List **L** empty.

IMPLEMENTATION OF ABSTRACT DATA TYPE "LIST"

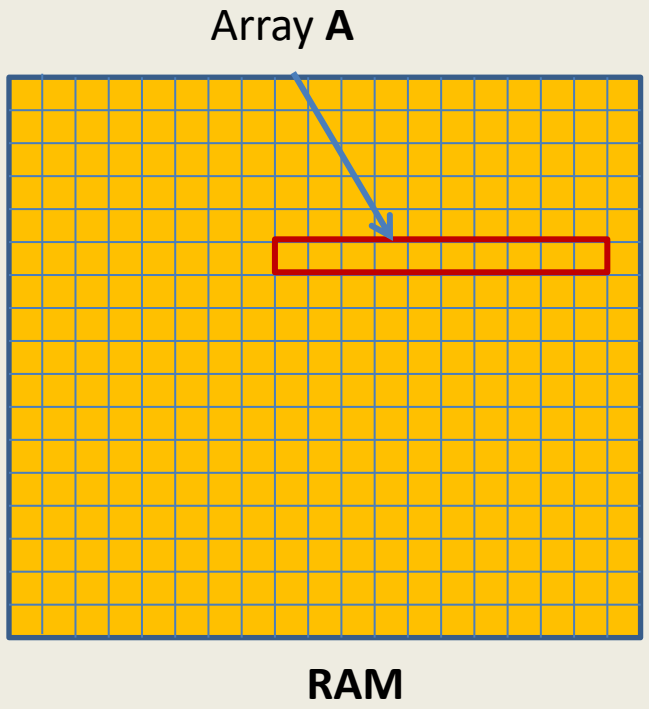
Array based Implementation

Taught in a Computer Hardware course.

- **RAM** allows **O**(1) time to access any memory location.
- Array is a contiguous chunk of memory kept in RAM.
- For an array **A**[] storing n words,

  the address of element **A**[i] =

  "start address of array **A**" + i



How can we use array for implementing a list?

Array based Implementation

- Store the elements of List in array **A** such that

  A[i] denotes ($i + 1$)th element of the list at each stage

  (since index starts from 0).

  (Assumption: The maximum size of list is known in advance.)
- Keep a integer variable **Length** to denote the number of elements in the list at each stage.

Example: If at any moment of time List is 3,5,1,8,0,2,40,27,44,67,

then the array **A** looks like:

Question: How to describe location of an element of the list ?

Answer: by the corresponding array index. Location of 5th element of List is 4.

Time Complexity of each List operation using Array based implementation

Operation	Time Complexity per operation
IsEmpty(L)	O(1)
Search(x,L)	O(n)
Successor(i,L)	O(1)
Predecessor(i,L)	O(1)
CreateEmptyList(L)	O(1)
Insert(x,i,L)	O(n)
Delete(i,L)	O(n)
MakeListEmpty(L)	O(1)

n: number of elements in list at present

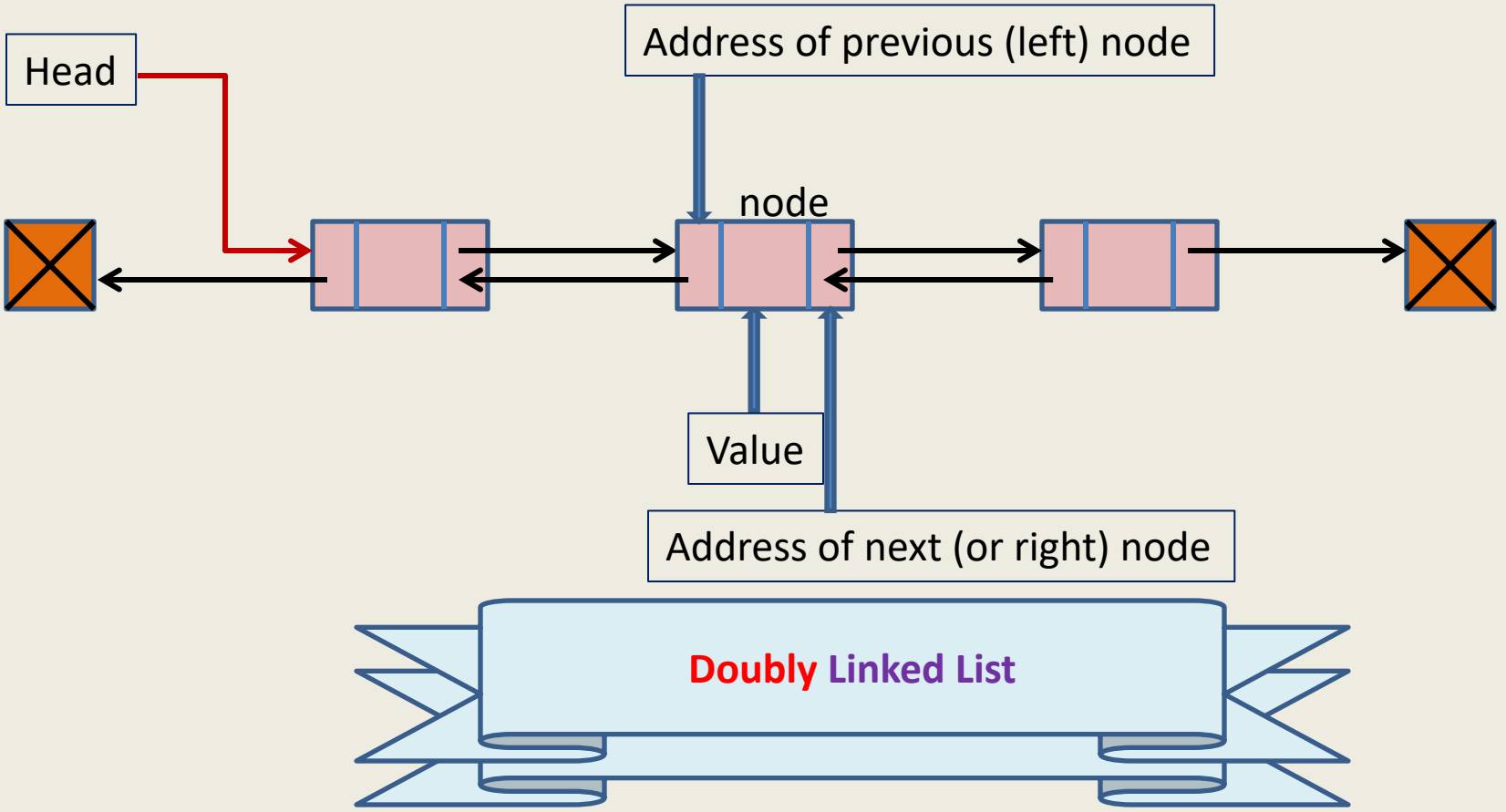
Arrays are very **rigid**

All elements from A[i] to A[$n-1$] have to be shifted to the **right** by one place.

All elements from A[$i+1$] to A[$n-1$] have to be shifted to the **left** by one place.

operation with matching complexity.

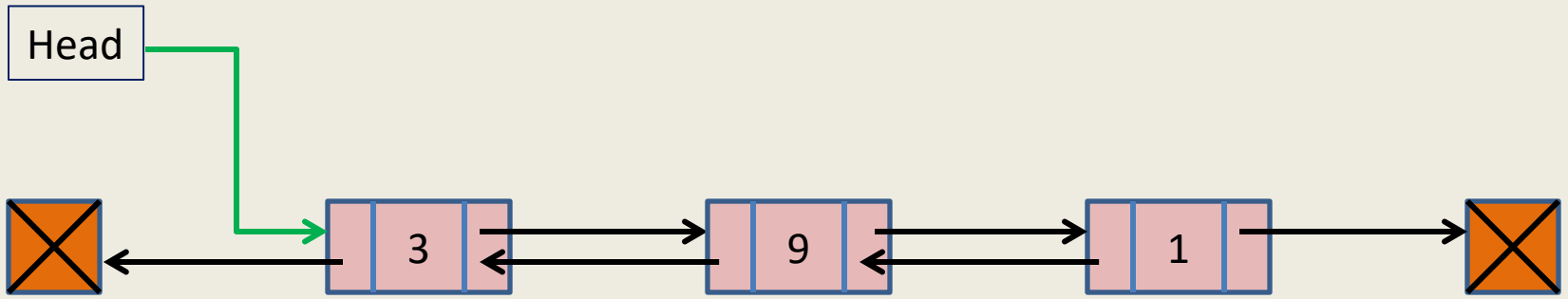
Link based Implementation:



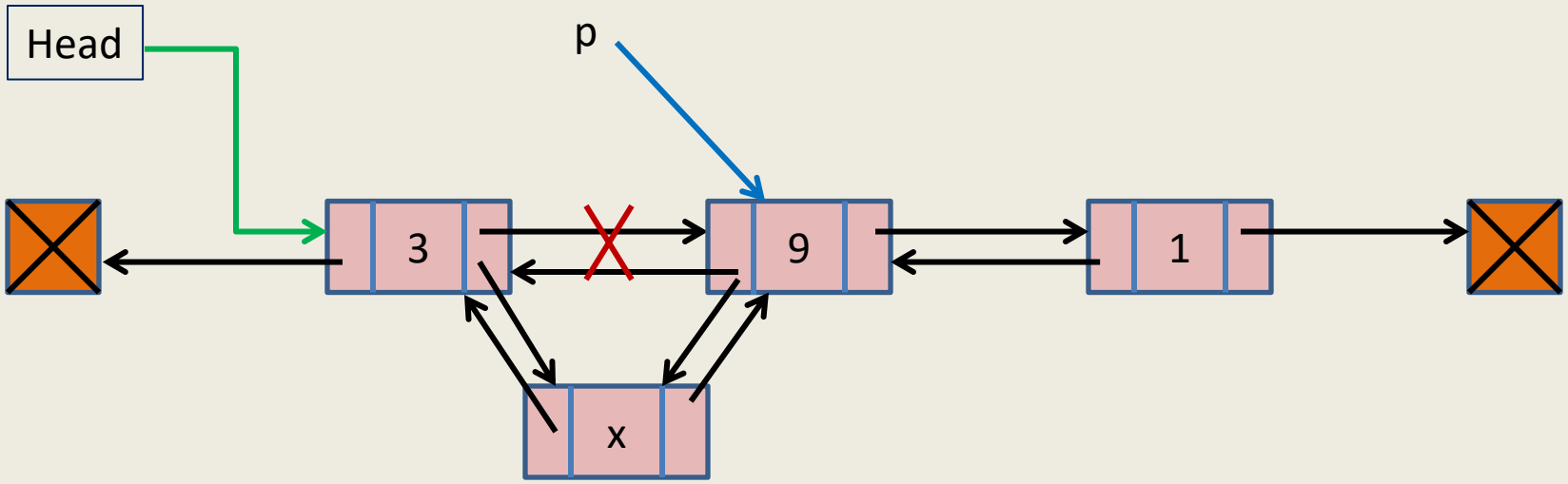
Doubly Linked List based Implementation

- Keep a doubly linked list
  where elements appear in the order we follow while traversing the list.
- The location of an element : the **address** of the node containing it.

Example: List 3,9,1 appears as

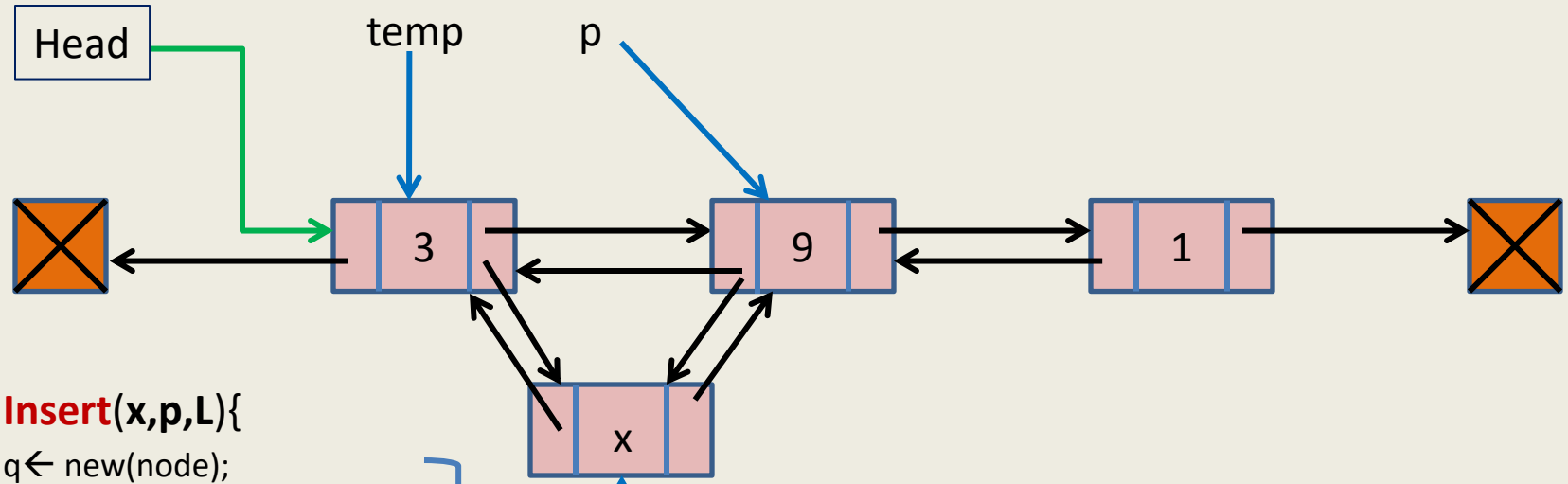


How to perform **Insert**(**x,p,L**) ?



How is it done actually ?

How to perform **Insert**(x,p,L) ?



Insert(x,p,L){
q← new(node);
q.value← x;
temp← p.left;
q.right ← p;
p.left← q;
q.left ← temp;
temp.right← q;
}

O(1) time

Lists are very **flexible**

Homework: How to perform **Delete**(x,p,L) ?

A Common Mistake

Often students interpret the parameter **p in Insert**(**x,p,L**) with the integer signifying the order (1st, 2nd,…) at which element **x** is to be inserted in list **L**.

Based on this interpretation, they think that **Insert**(**x,p,L**) will require scanning the list and hence will take time of the order of **p** and not **O**(1).

Here is my advice:

Please refer to the **modeling** of the list for exact interpretation of **p**.

The implementation in the lecture is for that modeling and indeed takes **O**(1) time.

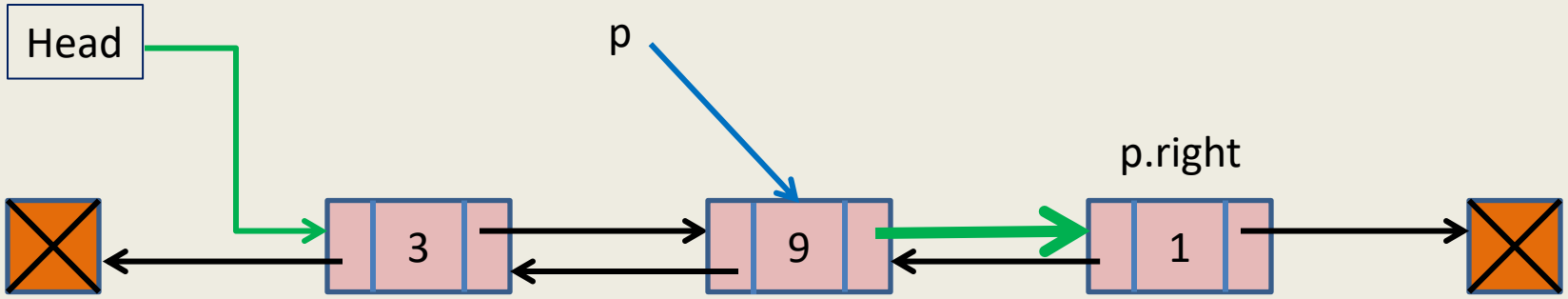
However, if you wish to model list *differently* such that the parameter **p** is an integer parameter as defined above, then you are right.

Lesson learnt:

Implementation of a data structure

depends upon its mathematical modeling (interpretation of various operations).

How to perform **successor**(**p,L**) ?



Successor(**p,L**){
q← p.right;
return q.value;
}

Time Complexity of each List operation using Doubly Linked List based implementation

Operation	Time Complexity per operation
IsEmpty(L)	O(1)
Search(x,L)	O(n)
Successor(p,L)	O(1)
Predecessor(p,L)	O(1)
CreateEmptyList(L)	O(1)
Insert(x,p,L)	O(1)
Delete(p,L)	O(1)
MakeListEmpty(L)	O(1)

It takes O(1) time if we implement it by setting the **head** pointer of list to NULL. However, if one has to free the memory used by the list, then it will require traversal of the entire list and hence O(n) time. You might learn more about it in Operating System course.

Homework: Write **C** Function for each operation with matching complexity.

Doubly Linked List based implementation versus array based implementation of "List"

Operation	Time Complexity per operation for array based implementation	Time Complexity per operation for doubly linked list based implementation
IsEmpty(L)	O(1)	O(1)
Search(x,L)	O(n)	O(n)
Successor(p,L)	O(1)	O(1)
Predecessor(p,L)	O(1)	O(1)
CreateEmptyList(L)	O(1)	O(1)
Insert(x,p,L)	O(n)	O(1)
Delete(p,L)	O(n)	O(1)
MakeListEmpty(L)	O(1)	O(1)

A CONCRETE PROBLEM

Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with name **x**
- Insert a new record (name, phone #,…)

Array based solution	Linked list based solution
Log n	$O(n)$
$O(n)$	$O(1)$

Yes. Keep the array sorted according to the **names** and do Binary search for **x**.

ver it …

Can we achieve the best of the two data structure simultaneously ?

We shall together invent such **a novel data structure** in the next class

Data Structures and Algorithms

(ESO207)

Lecture 9:

Inventing a new Data Structure with

- Flexibility of lists for updates
- Efficiency of arrays for search

Important Notice

There are basically two ways of introducing a new/innovative solution of a problem.

1. One way is to just explain it without giving any clue as to how the person who invented the concept came up with this solution.
2. Another way is to start from scratch and take a journey of the route which the inventor might have followed to arrive at the solution.

   This journey goes through various hurdles and questions,

   each hinting towards a better insight into the problem

   if we have patience and open mind.

Which of these two ways is better ?

I believe that the second way is better and more effective.

The current lecture is based on this way. The data structure we shall invent is called a Binary Search Tree. This is the most fundamental and versatile data structure. We shall realize this fact many times during the course …

Doubly Linked List based implementation versus array based implementation of "List"

Operation	Time Complexity per operation for array based implementation	Time Complexity per operation for doubly linked list based implementation
IsEmpty(L)	O(1)	O(1)
Search(x,L)	O(n)	O(n)
Successor(p,L)	O(1)	O(1)
Predecessor(p,L)	O(1)	O(1)
CreateEmptyList(L)	O(1)	O(1)
Insert(x,p,L)	O(n)	O(1)
Delete(p,L)	O(n)	O(1)
MakeListEmpty(L)	O(1)	O(1)

Arrays are very **rigid**

Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with ID no. x
- Insert a new record (ID no., phone #,…)

Array based solution	Linked list based solution
Log n	O(n)
O(n)	Log n

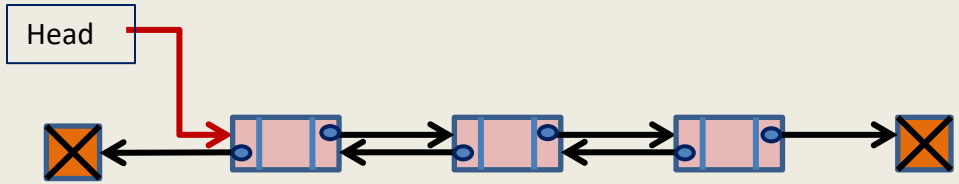
Can we achieve **the best of** the two data structure simultaneously ?

We shall together invent such **a novel data structure** today

Inventing a new data structure

New Data structure

Lists are flexible, so let us try modifying the linked list structure to achieve fast **search** time.

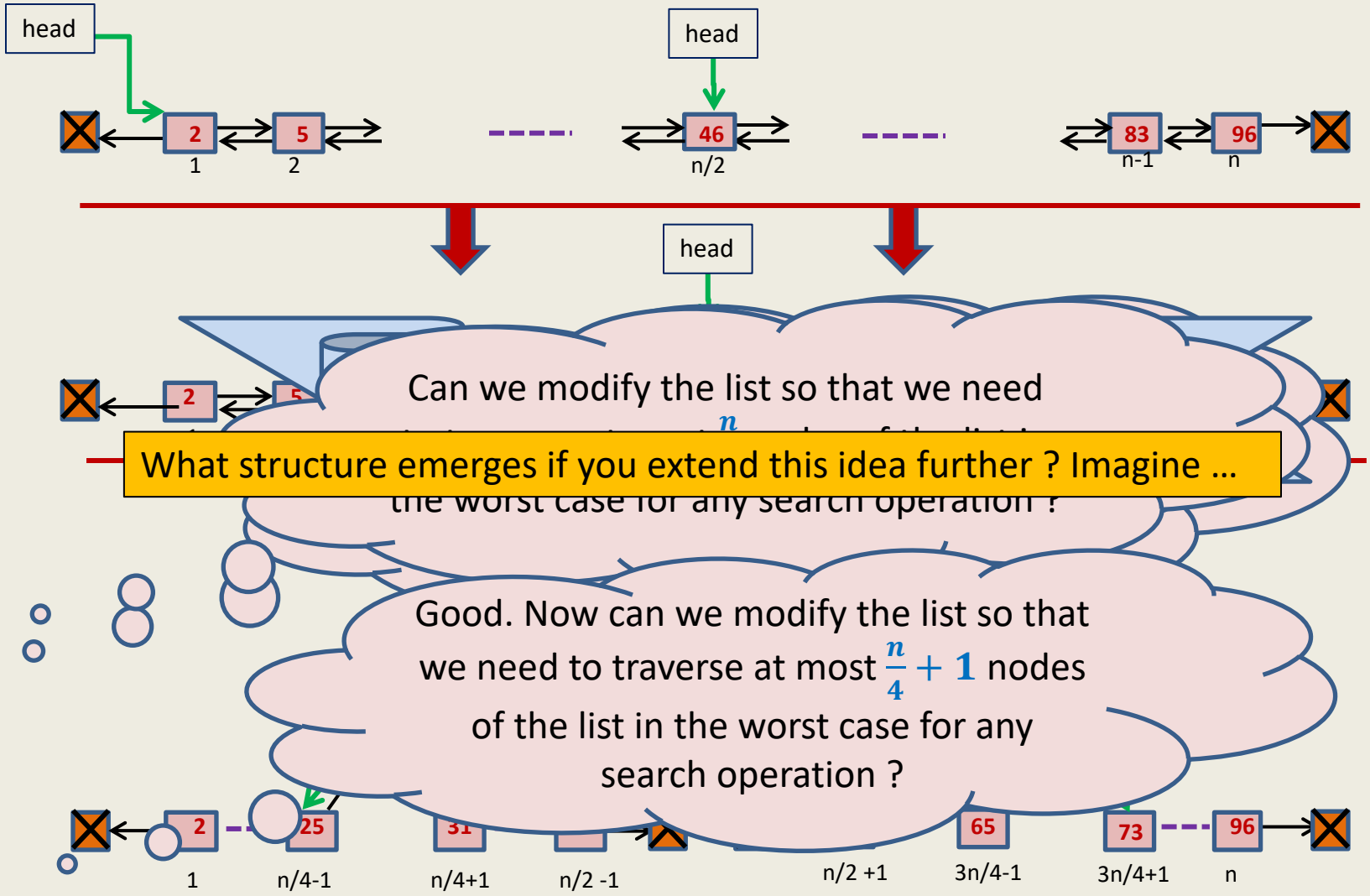


Lists

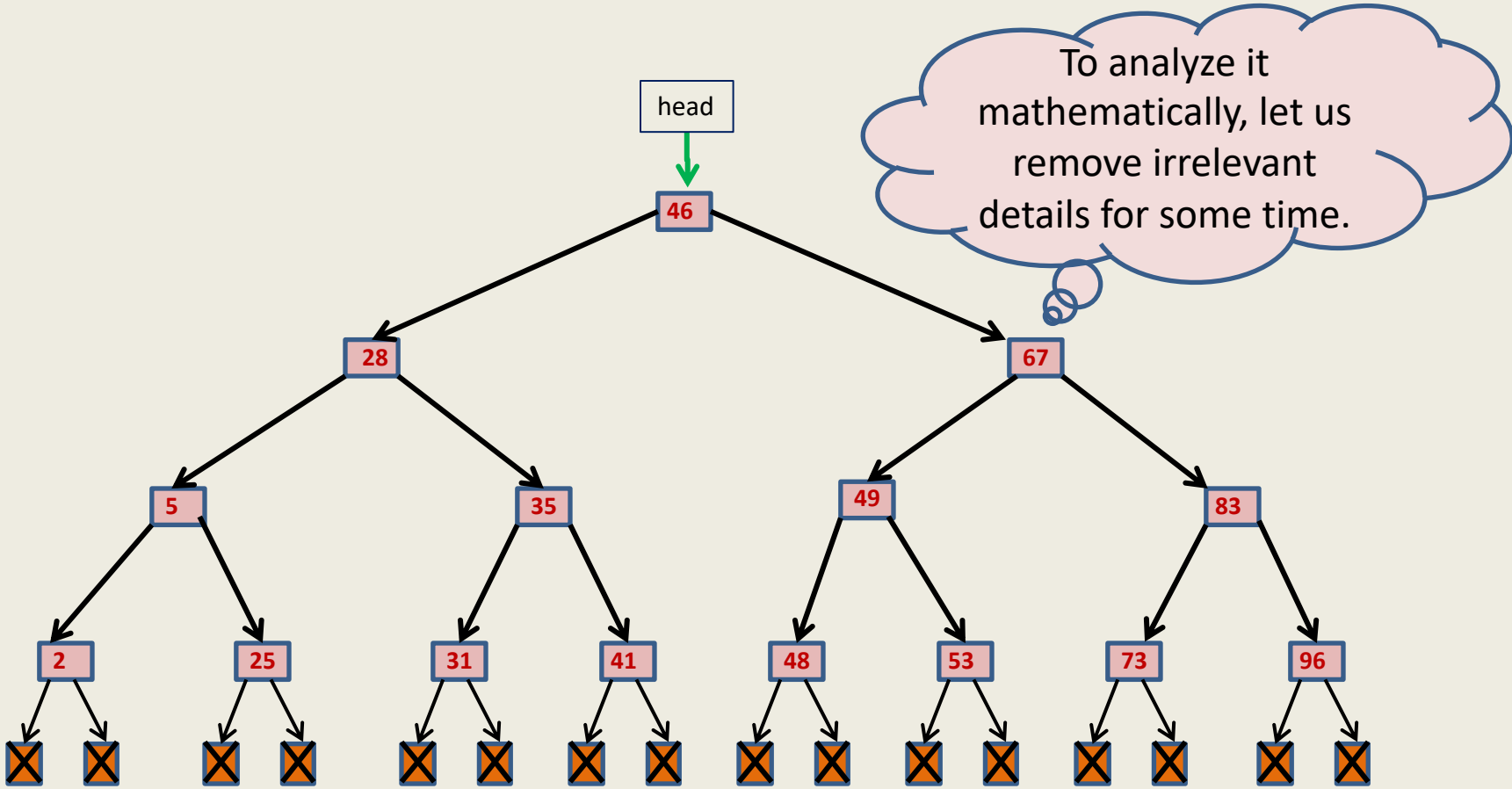
Too Rigid for updates

Array

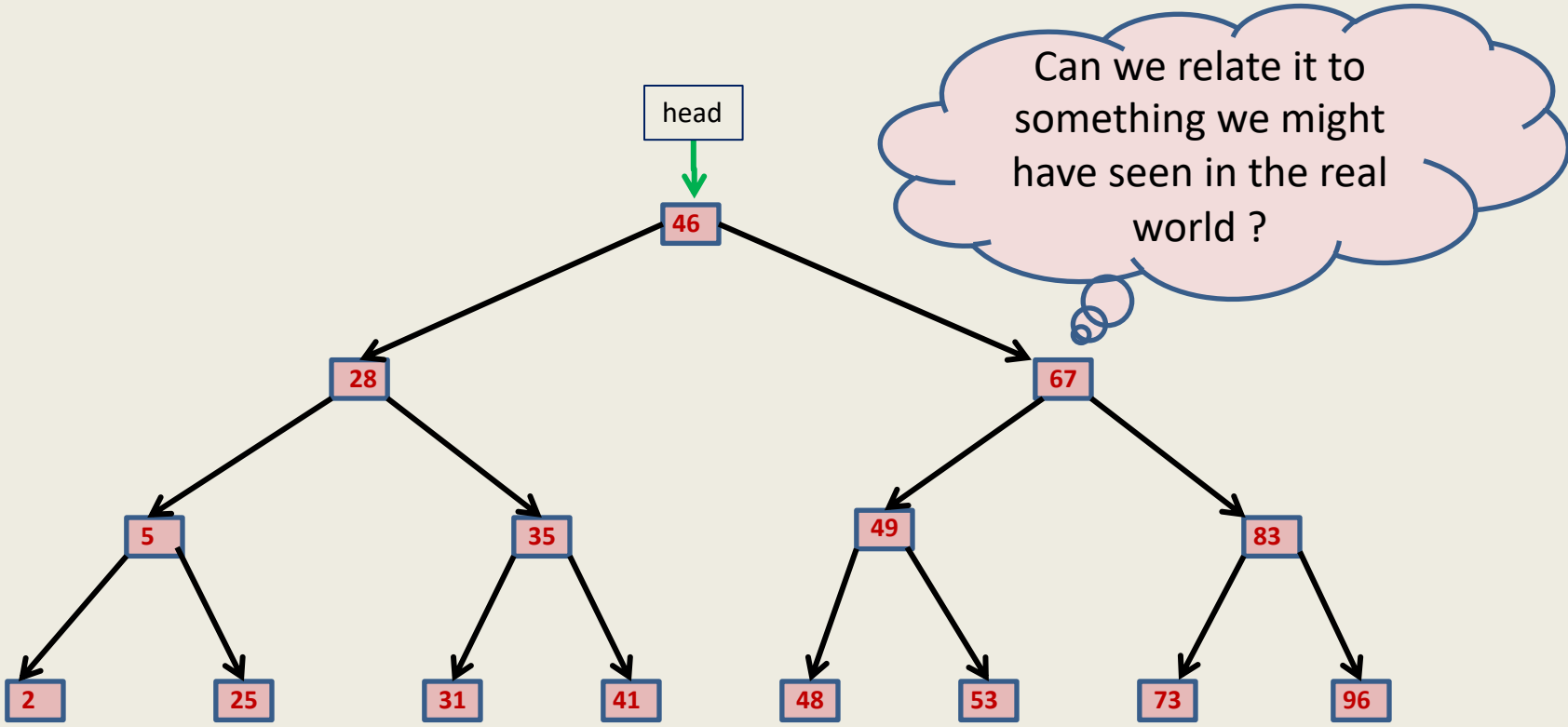
Restructuring doubly linked list



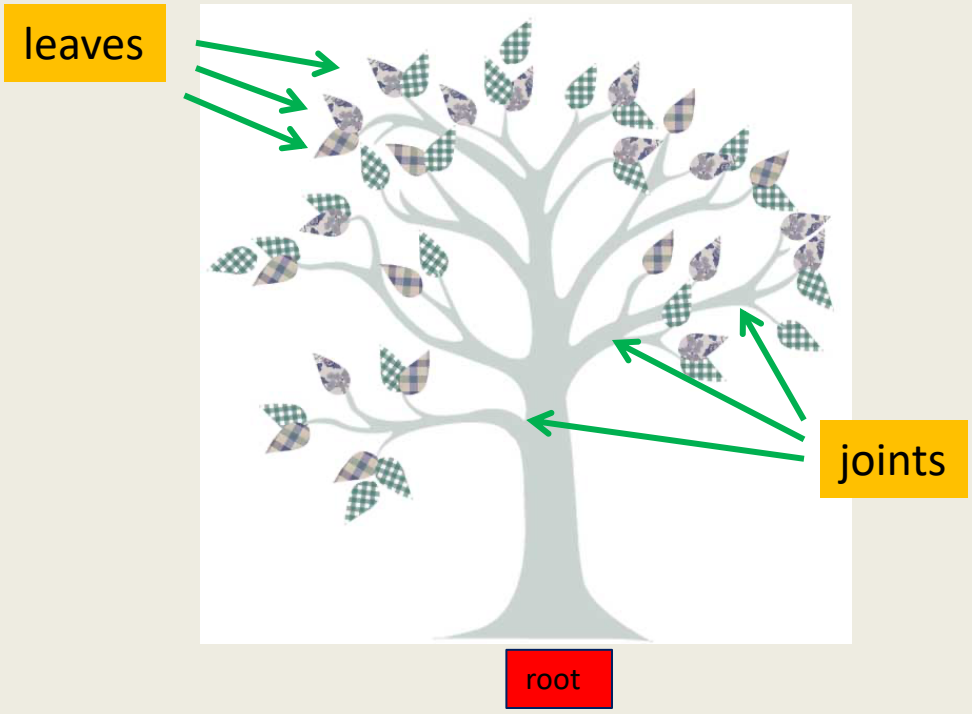
A new data structure emerges



A new data structure emerges

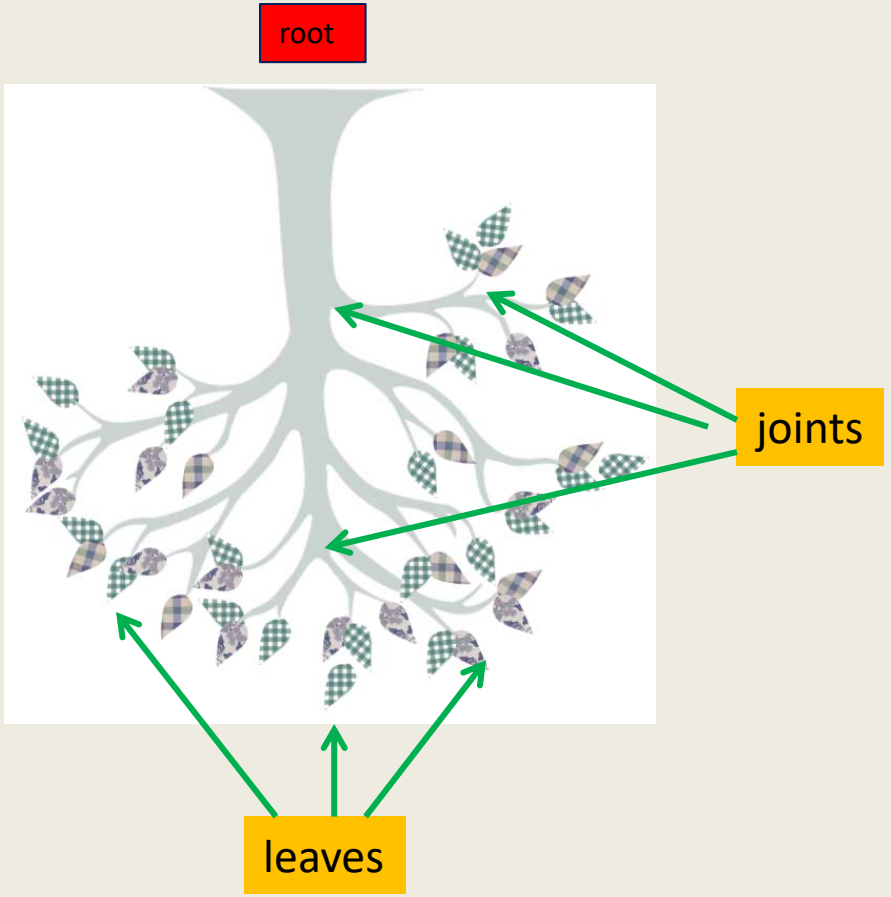


Nature :
a great source of inspiration

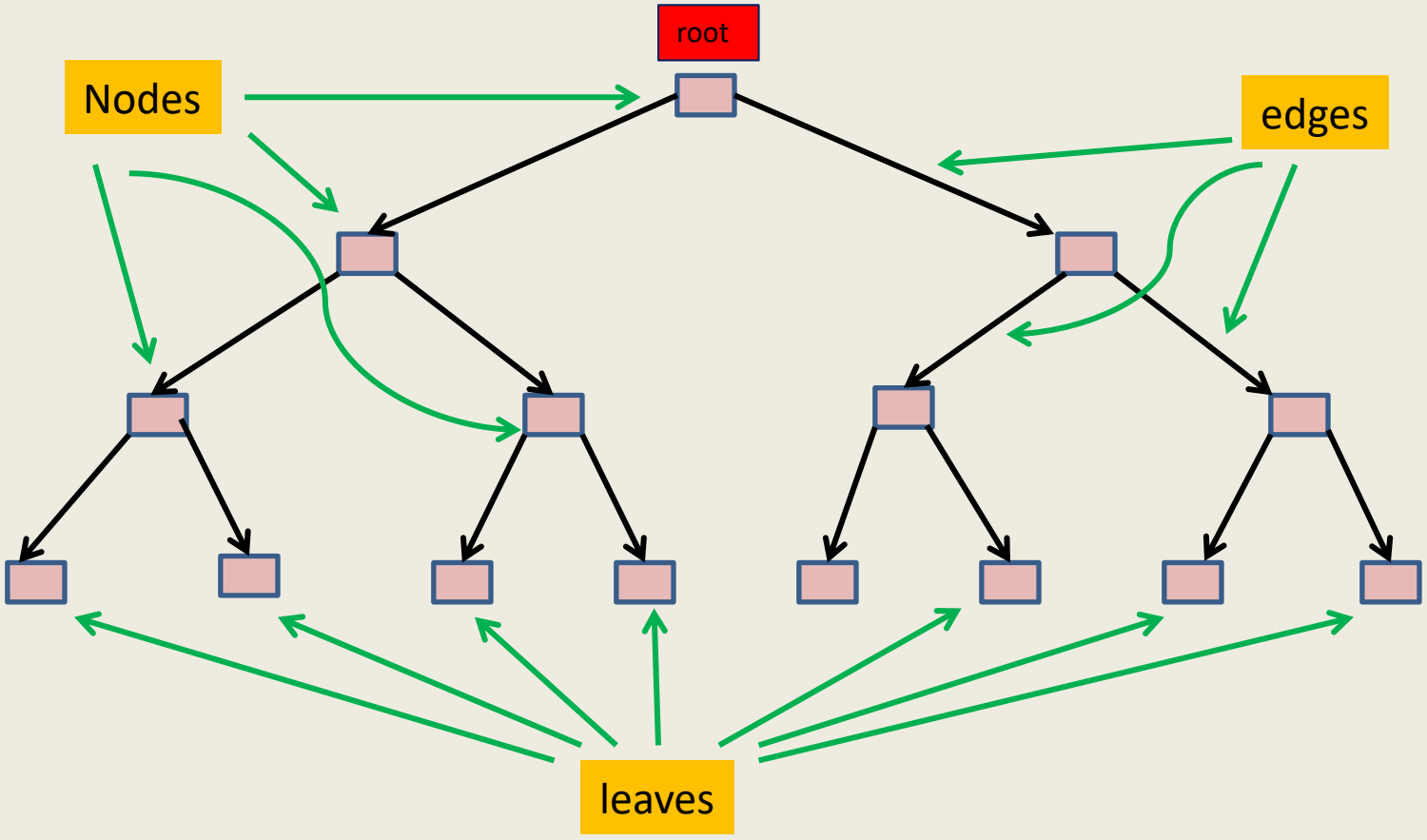


Nature :
a great source of inspiration

Nature :
a great source of inspiration



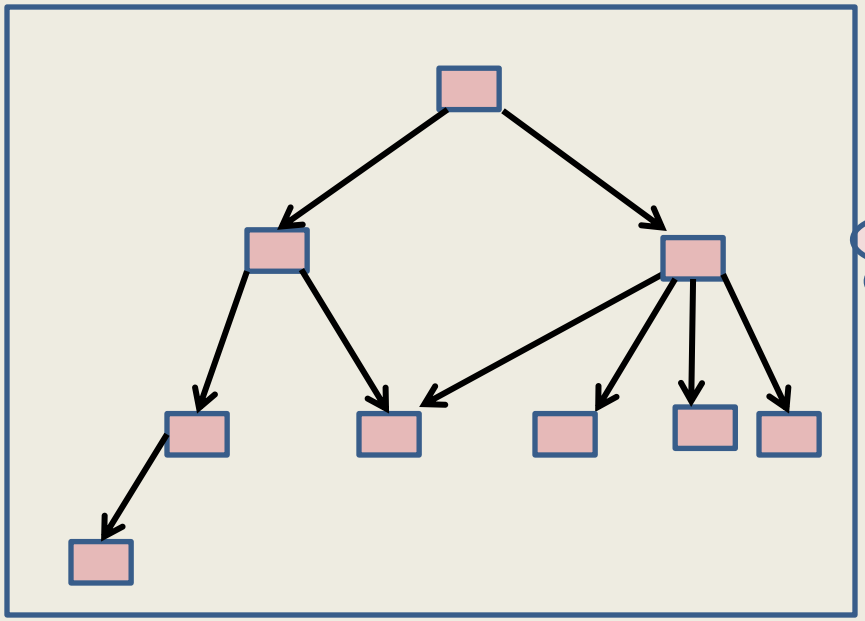
Nature :
a great source of inspiration

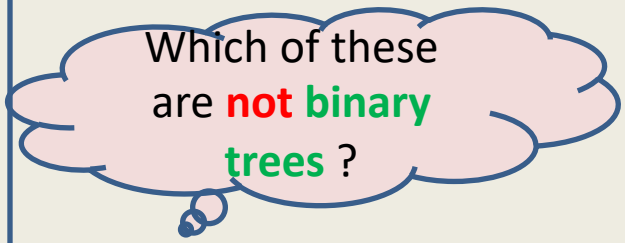


Binary Tree: A mathematical model

Definition: A collection of nodes is said to form a binary tree if

1. There is exactly one node with no incoming edge.
   This node is called the **root** of the tree.
2. Every node other than root node has **exactly one incoming edge**.
3. Each node has **at most two outgoing edges**.





Binary Tree: some terminologies

- If there is an edge from node u to node v,

  then u is called **parent** of v ,and v is called **child** of u.
- The **Height** of a Binary tree **T** is the maximum number of edges from the root to any leaf node in the tree **T**.

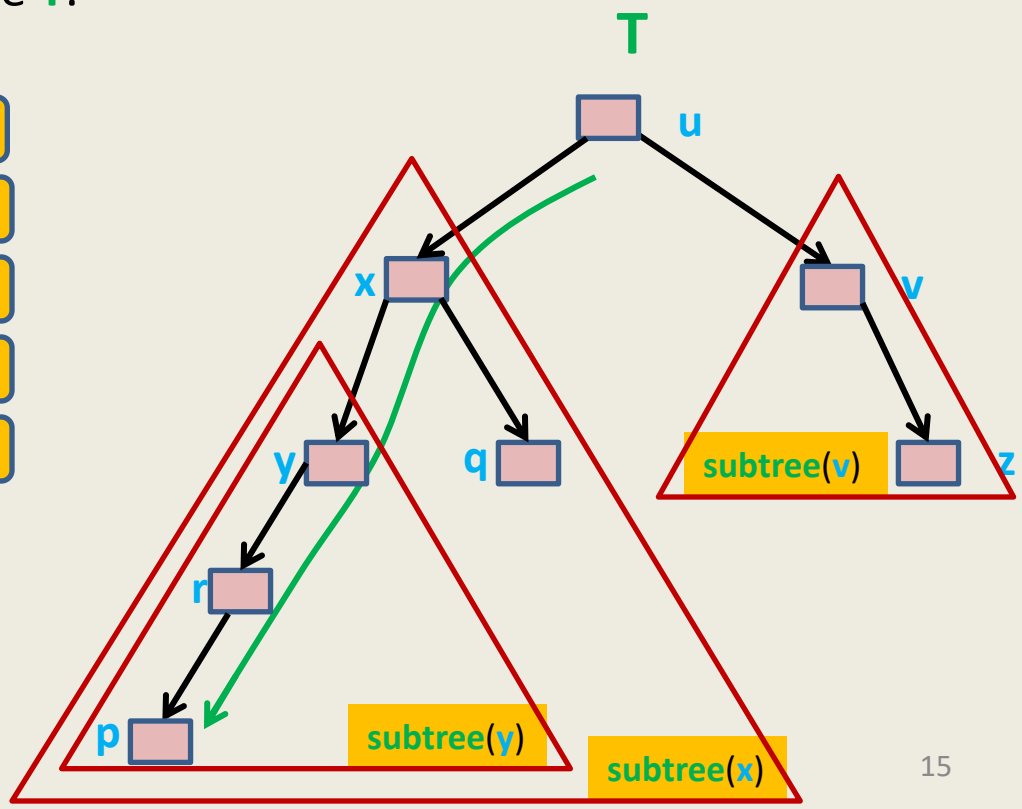
parent(y) = x

parent(v) = u

children(y) = {r}

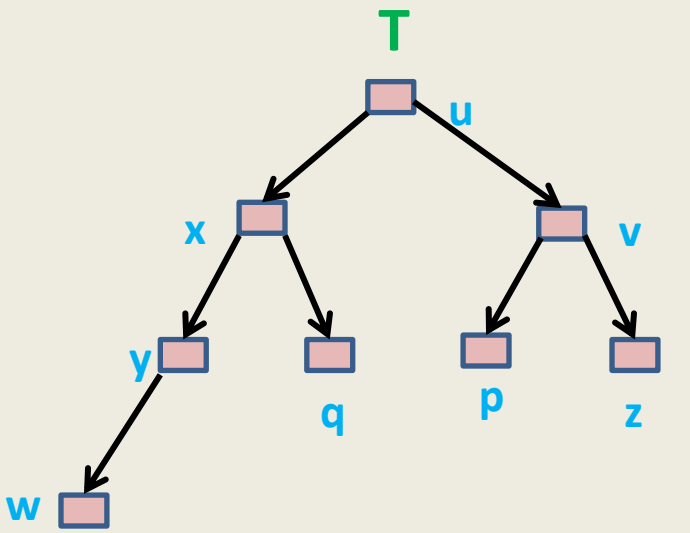
children(x) = {y,q}

height(T) = 4



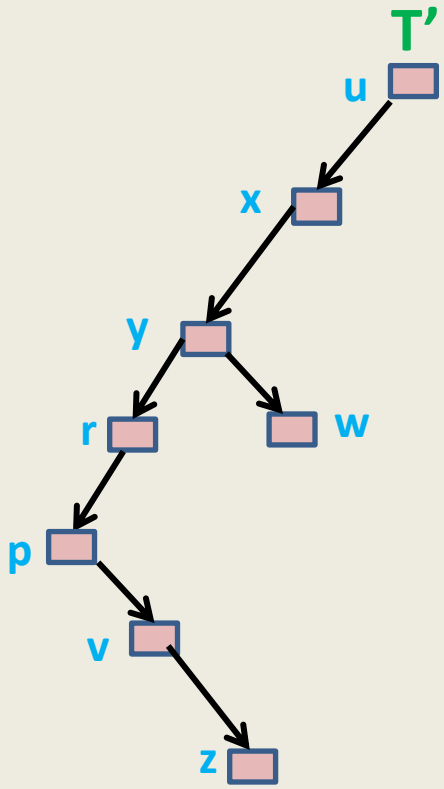
Varieties of Binary trees

We call it Perfectly balanced

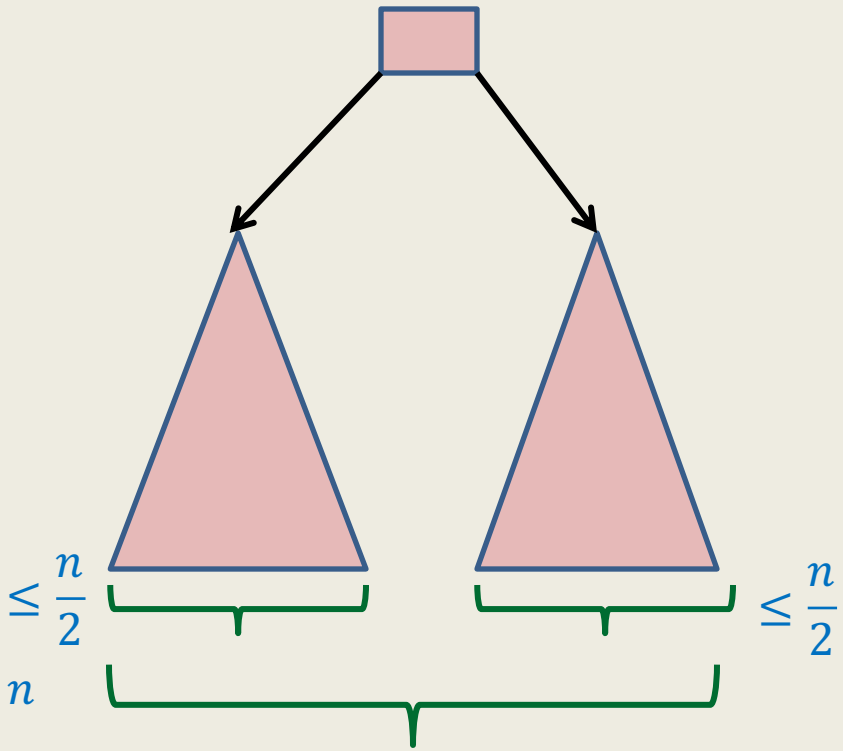


For every node, the number of nodes in the **subtrees of its two children** differ at **atmost** by 1.

skewed



Height of a perfectly balanced Binary tree

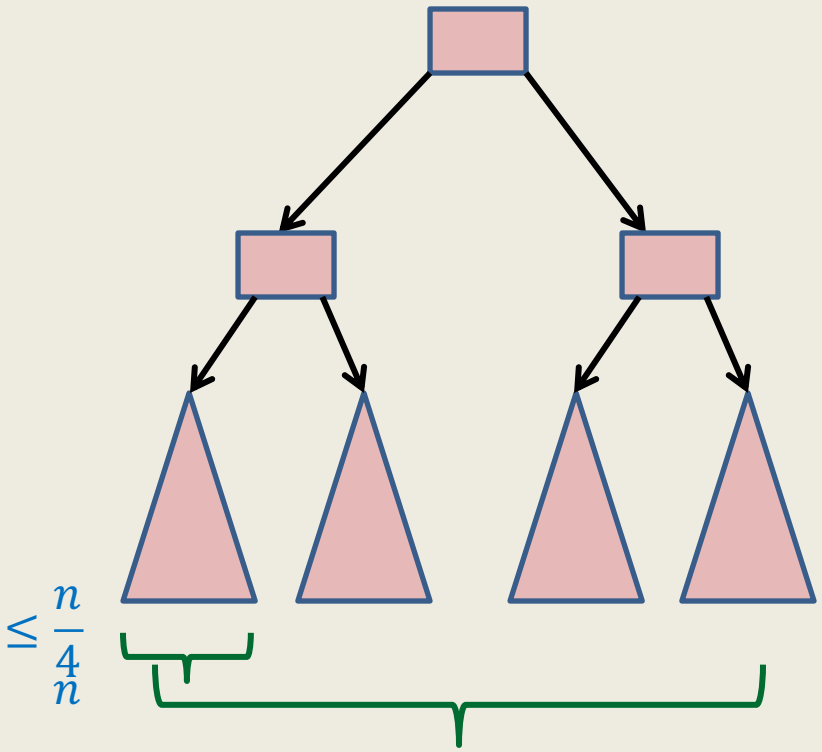


$\boldsymbol{H}(n)$: Height of a perfectly balanced binary tree on n nodes.

$\boldsymbol{H}(1) = 0$

$\boldsymbol{H}(n) \leq \mathbf{1} + \boldsymbol{H}\left(\frac{n}{2}\right)$

Height of a perfectly balanced Binary tree

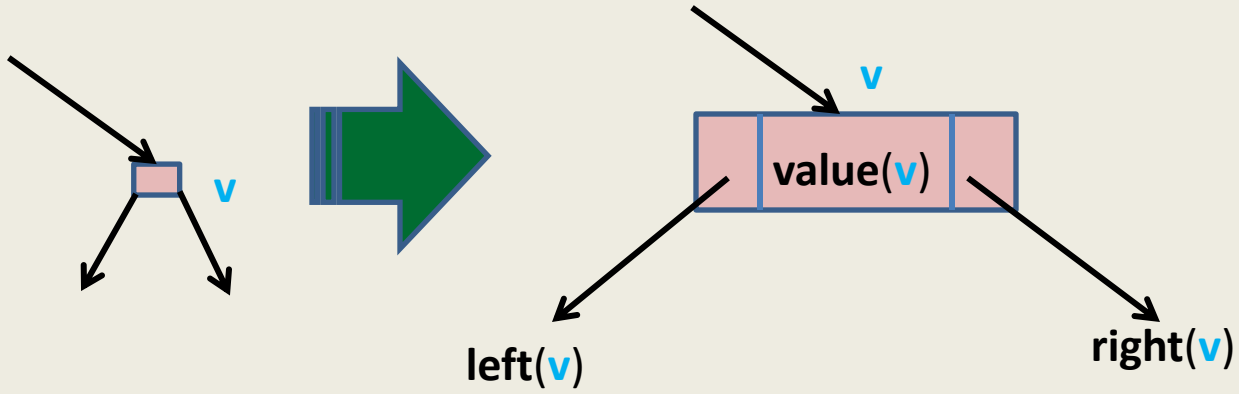


$\boldsymbol{H}(n)$: Height of a perfectly balanced binary tree on n nodes.

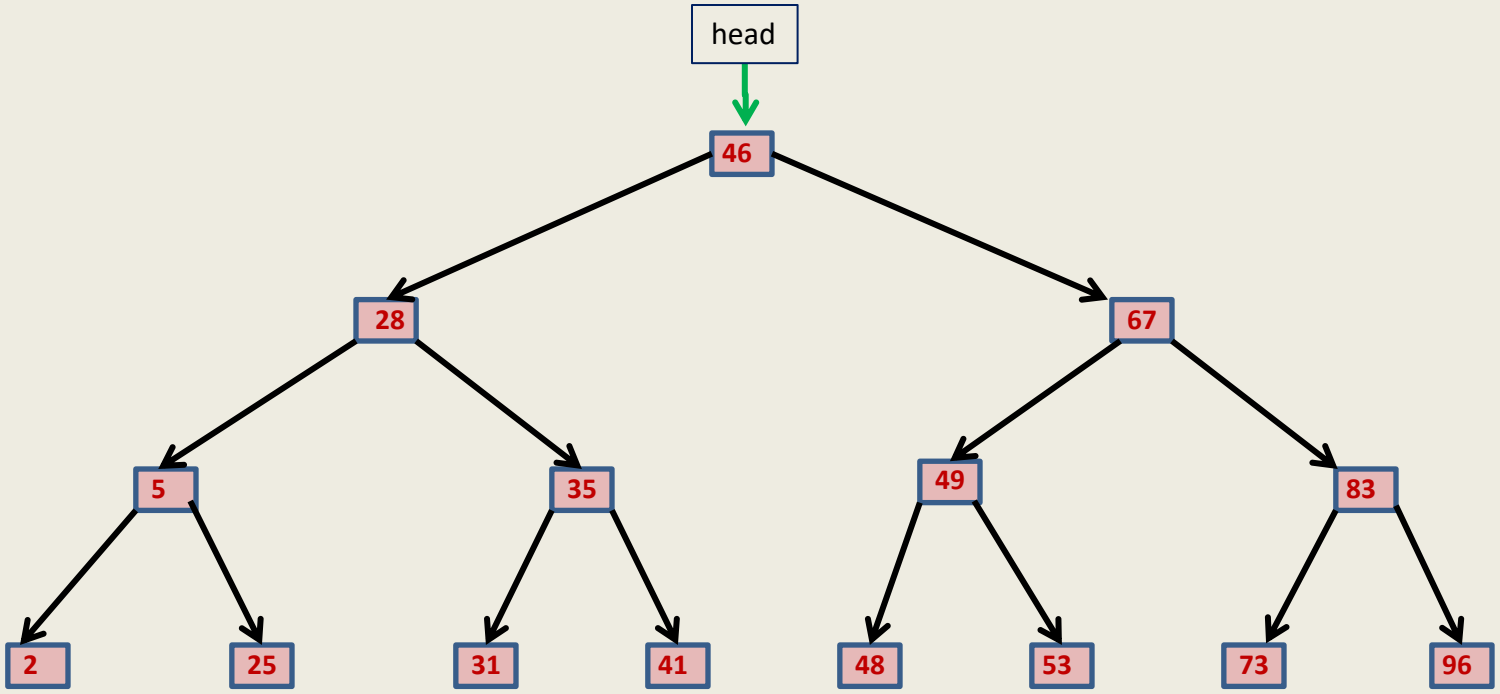
$\boldsymbol{H}(1) = 0$

$$\begin{aligned}\boldsymbol{H}(n) &\leq \mathbf{1} + \boldsymbol{H}\left(\frac{n}{2}\right)\\ &\leq \mathbf{1} + \mathbf{1} + \boldsymbol{H}\left(\frac{n}{4}\right)\\ &\leq \underbrace{\mathbf{1} + \mathbf{1} + \cdots +}_{i} \boldsymbol{H}\left(\frac{n}{2^i}\right)\\ &\leq \mathbf{log}_2\, n\end{aligned}$$

Implementing a Binary tree



Binary Search Tree (BST)



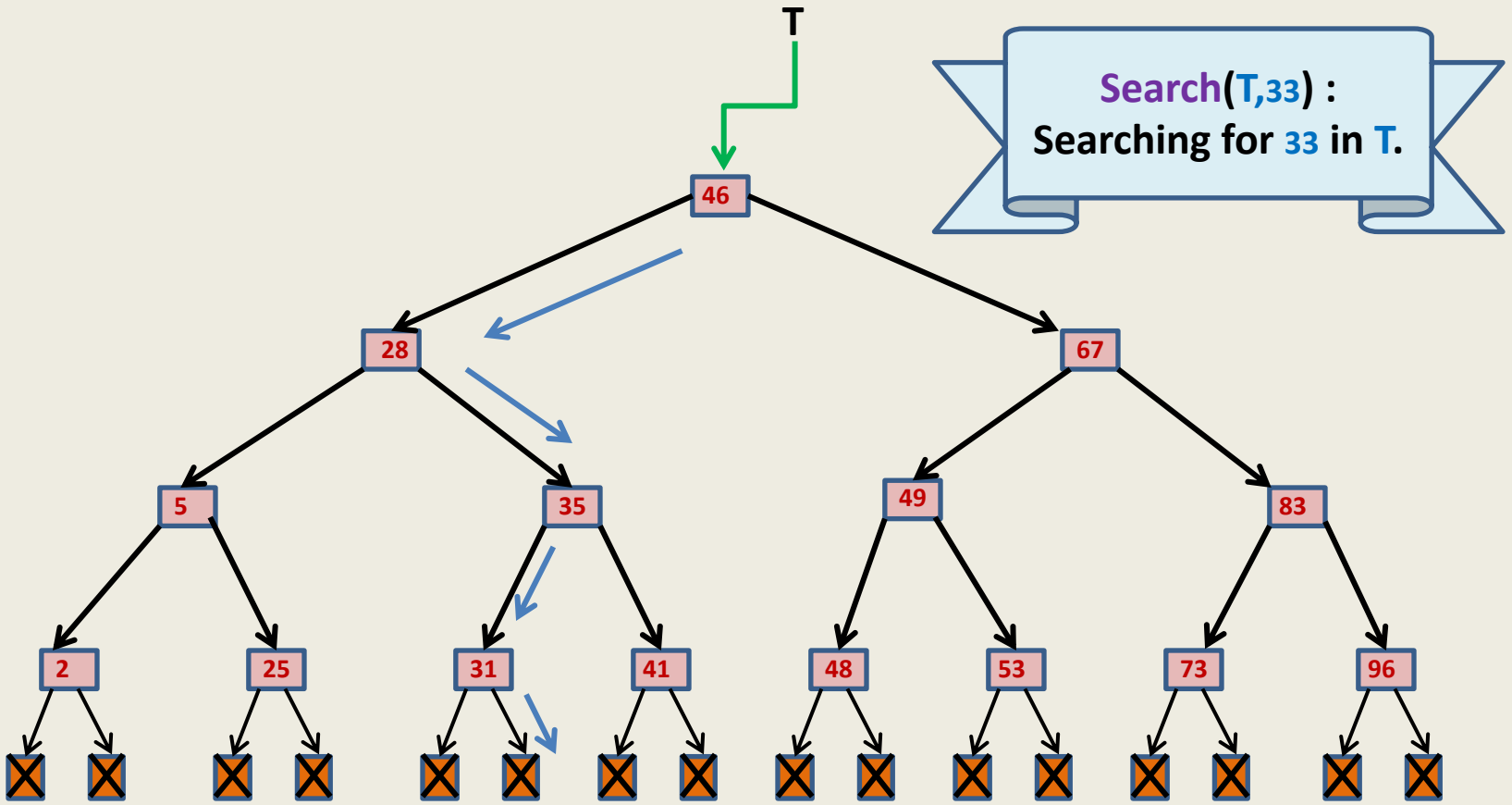
Definition: A Binary Tree **T** storing values is said to be **Binary Search Tree**

if for each node **v** in **T**

- If **left**(**v**) <> NULL, then **value**(**v**) > **value** of every node in **subtree**(**left**(**v**)).
- If **right**(**v**)<>NULL, then **value**(**v**) < **value** of every node in **subtree**(**right**(**v**)).

Search(T,x)

Searching in a Binary Search Tree

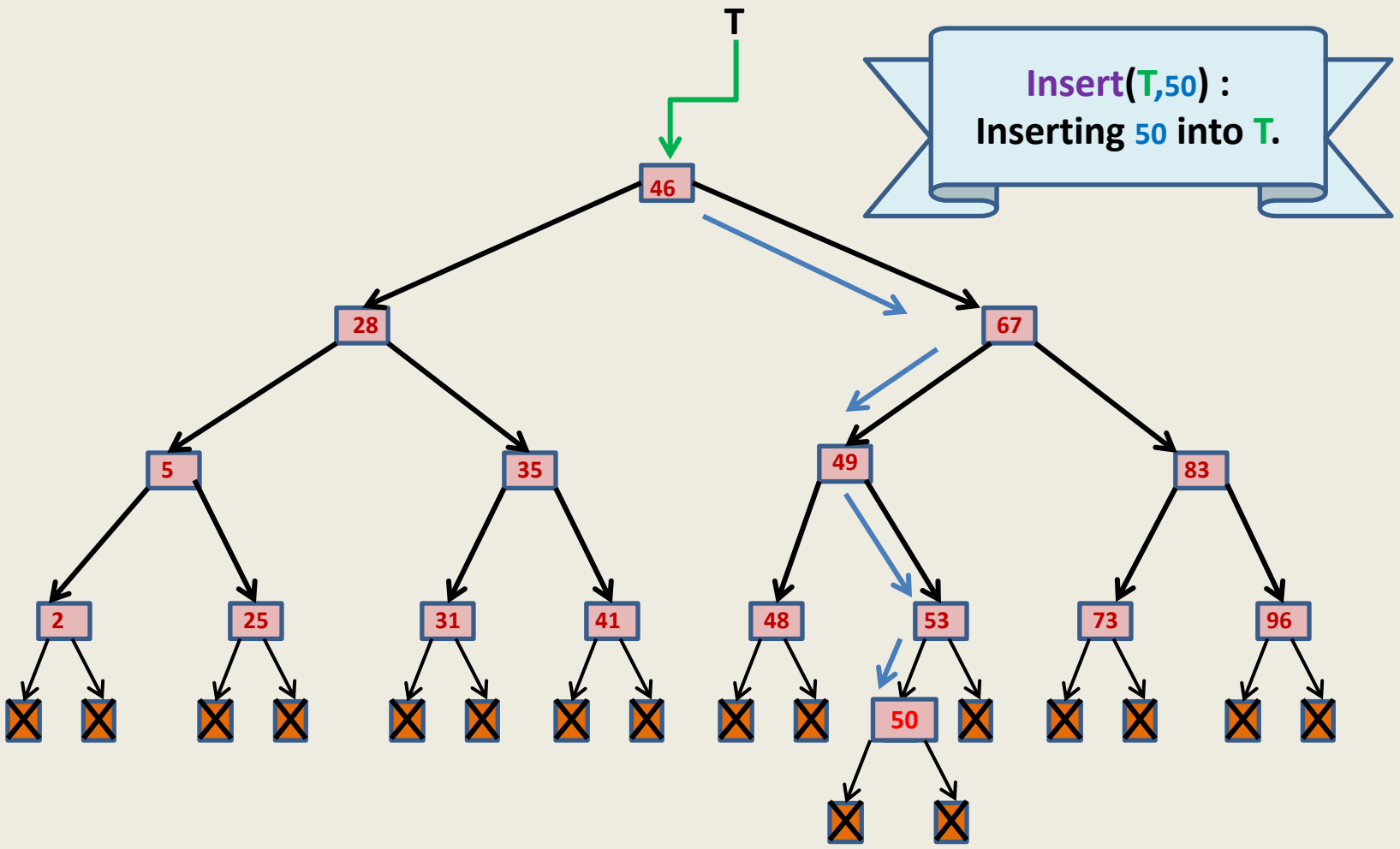


Search(T,x)
Searching in a Binary Search Tree

```
Search(T,x)
{       p← T;
        Found← FALSE ;
        while( Found= FALSE & p<> NULL )
        {       if(value(p) = x)   Found← TRUE   ;
                else if (value(p) < x)  p← right(p)  ;
                        else    p← left(p)  ;
        }
        return p;
}
```

Insert(T,x)

Insertion in a Binary Search Tree



A question

Time complexity of

Search(**T**,x) and **Insert**(**T**,x) in a Binary Search Tree **T** = **O(Height(T))**

Homeworks

1. Write pseudocode for **Insert**(T,x) operation similar to the pseudocode we wrote for **Search**(T,x).

2. Design an algorithm for the following problem:

Given a sorted array **A** storing n elements,

build a "perfectly balanced" BST storing all elements of **A**

in **O**(n) time.

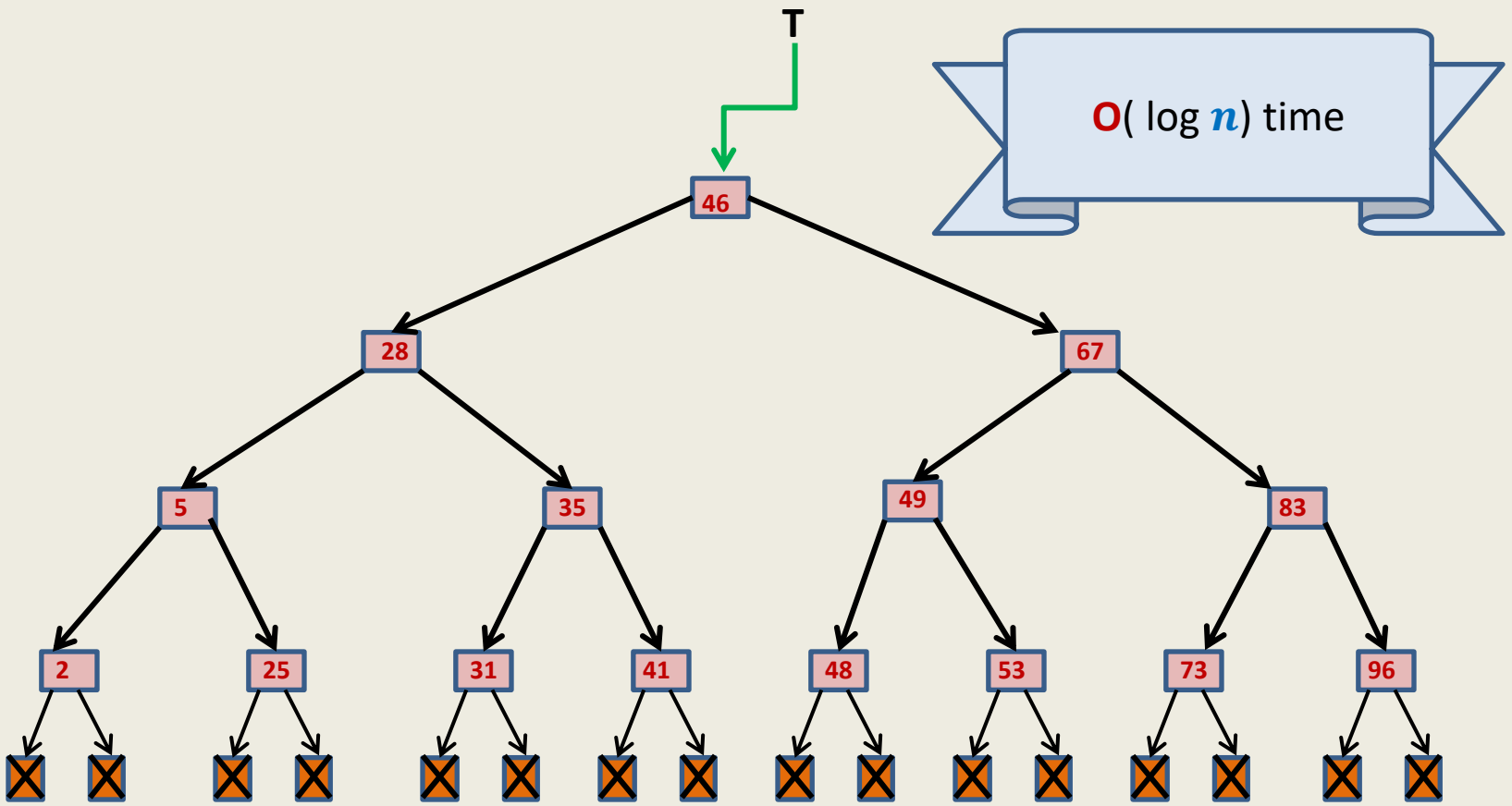
Homework 3

What does the following algorithm accomplish ?

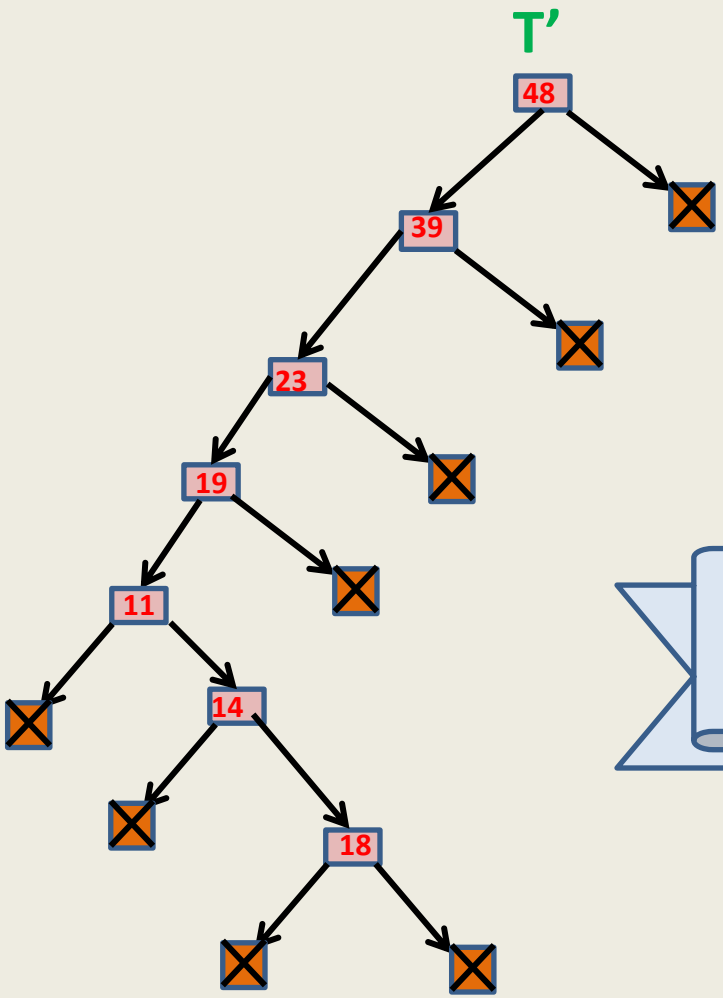
```
Traversal(T)
{     p← T;
      if(p=NULL) return;
      else{     if(left(p) <> NULL)    Traversal(left(p));
                print(value(p));
                if(right(p) <> NULL)    Traversal(right(p));
           }
}
```

Ponder over this algorithm for a few minutes to know what it is doing. You might like to try it out on some example of BST.

Time complexity of any search and any single insertion in a perfectly balanced Binary Search Tree on n nodes



Time complexity of any search and any single insertion in a sqewed Binary Search Tree on n nodes



O(n) time !☹

Our Original Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with ID no. x
- Insert a new record (ID no., phone #,…)

Array based solution	Linked list based solution
Log n	O(n)
O(n)	Log n

Solution : We may keep **perfectly balanced** BST.

Hurdle: What if we insert records in increasing order of ID ?

➔ BST will be skewed ☹

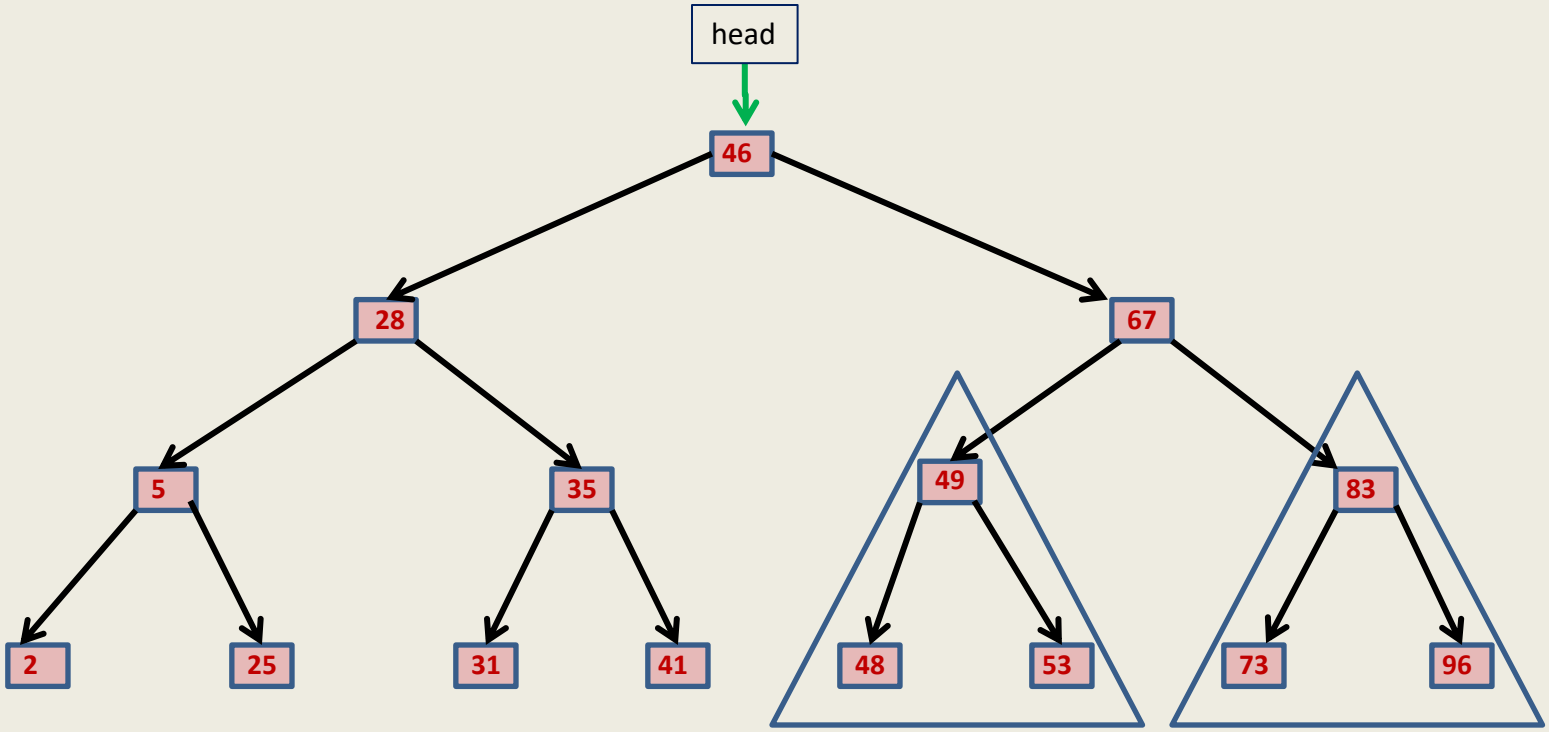
Data Structures and Algorithms

(ESO207)

Lecture 10:

- Exploring nearly balanced BST for the directory problem
- Stack: a new data structure

Binary Search Tree (BST)



Definition: A Binary Tree **T** storing values is said to be **Binary Search Tree**

if for each node **v** in **T**

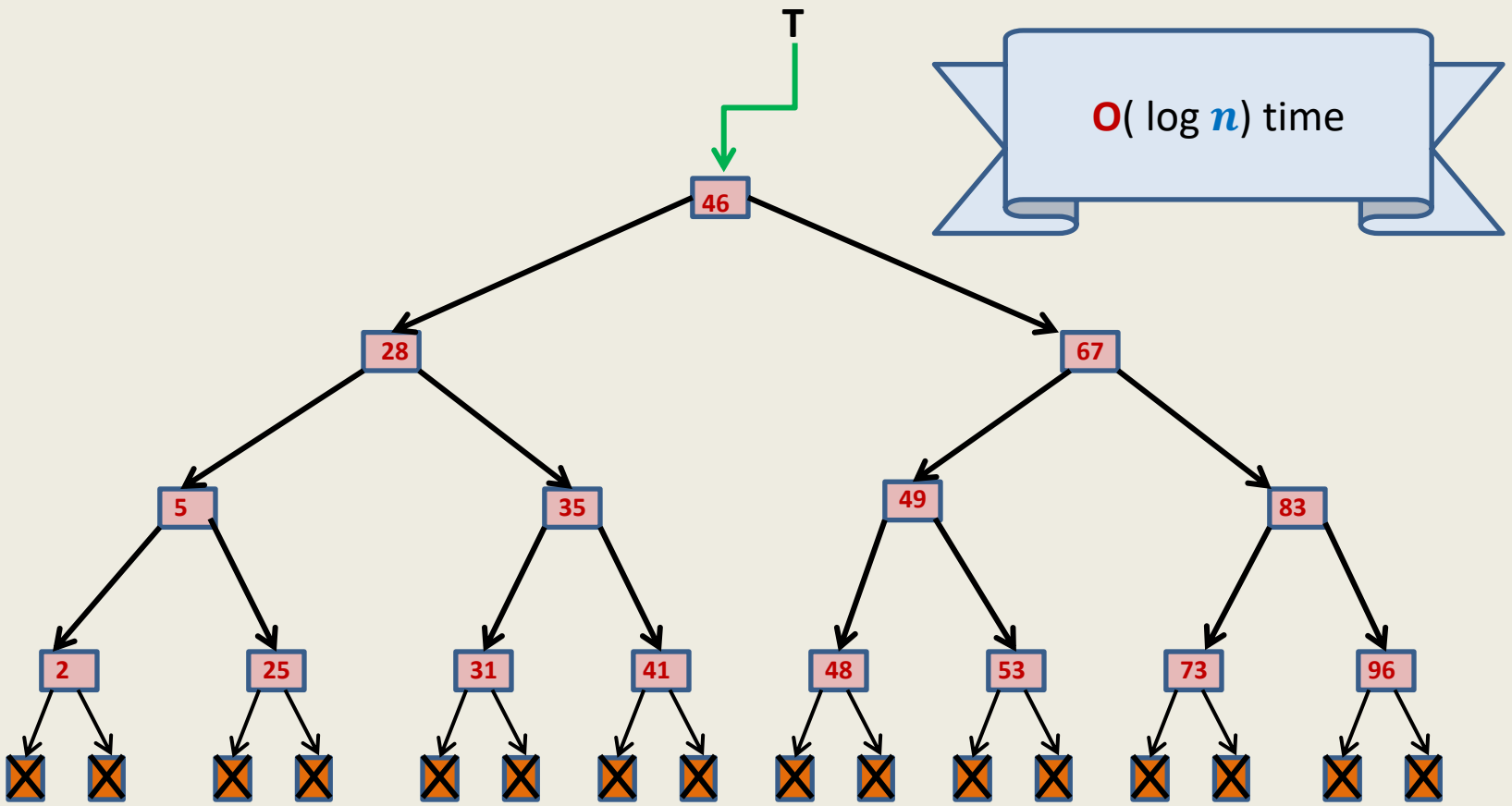
- If **left**(**v**) <> NULL, then **value**(**v**) > **value** of every node in **subtree**(**left**(**v**)).
- If **right**(**v**)<>NULL, then **value**(**v**) < **value** of every node in **subtree**(**right**(**v**)).

A question

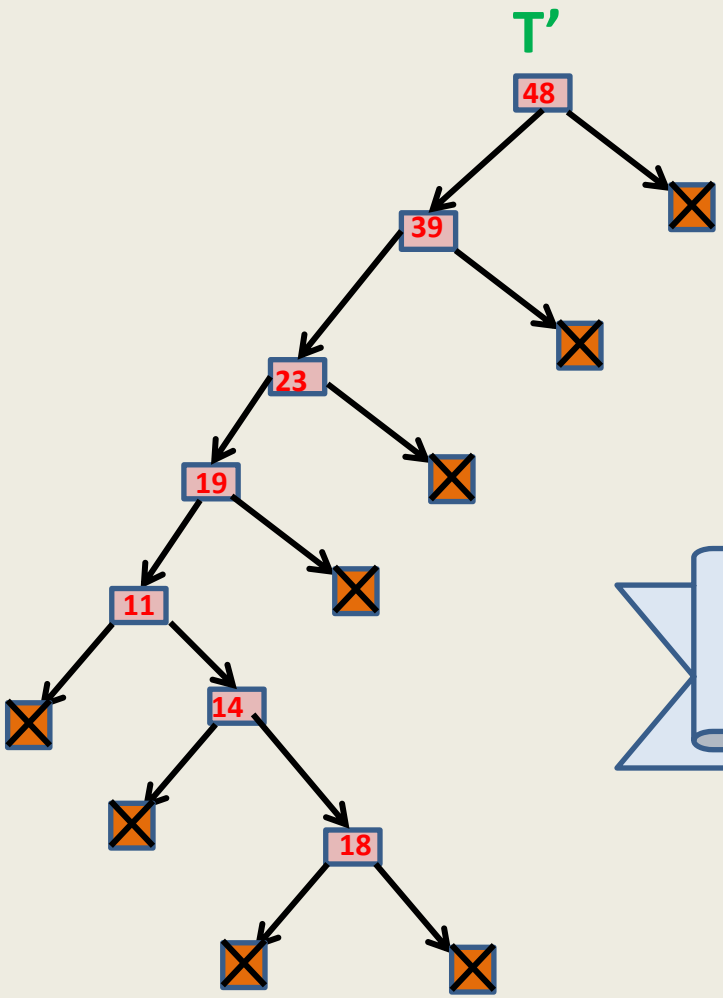
Time complexity of

Search(**T**,x) and **Insert**(**T**,x) in a Binary Search Tree **T** = **O(Height(T))**

Time complexity of any search and any single insertion in a perfectly balanced Binary Search Tree on n nodes



Time complexity of any search and any single insertion in a sqewed Binary Search Tree on n nodes



O(n) time !☹

Our Original Problem

Maintain a telephone directory

Operations:

- Search the phone # of a person with ID no. x
- Insert a new record (ID no., phone #,…)

Array based solution	Linked list based solution
Log n	O(n)
O(n)	Log n

Solution : We may keep **perfectly balanced** BST.

Hurdle: What if we insert records in increasing order of ID ?

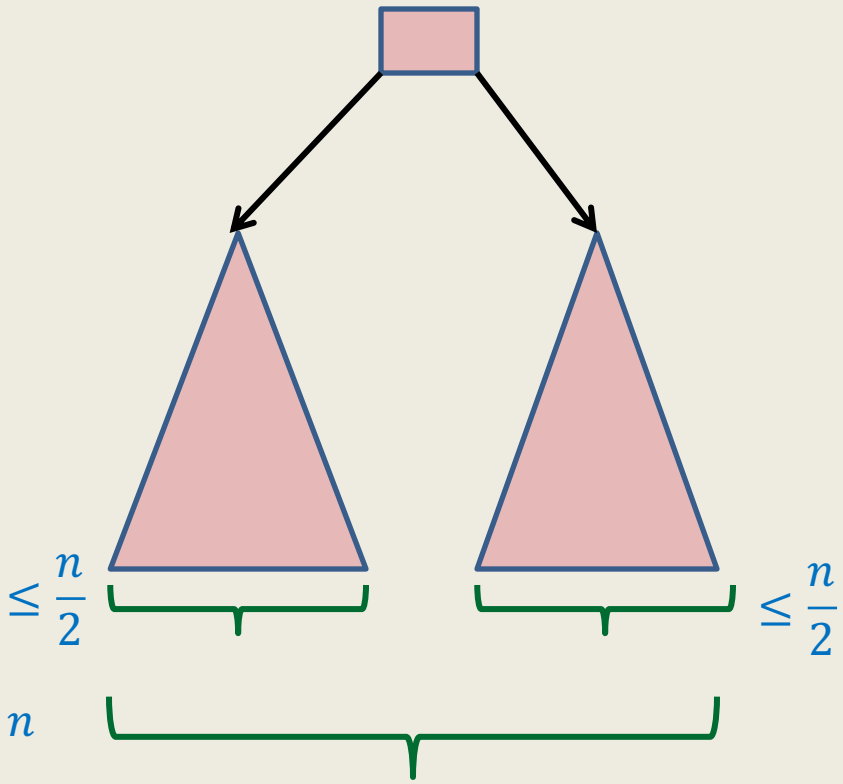
➔ BST will be skewed ☹

BST data structure that we invented looks very elegant,

let us try to find a way to overcome the hurdle.

- Let us try to find a way of achieving **Log n** search time.
- Perfectly balanced BST achieve **Log n** search time.
- But the definition of Perfectly balanced BST looks **too restrictive**.
- Let us investigate : How crucial is perfect balance of a BST ?

How crucial is perfect balance of a BST ?

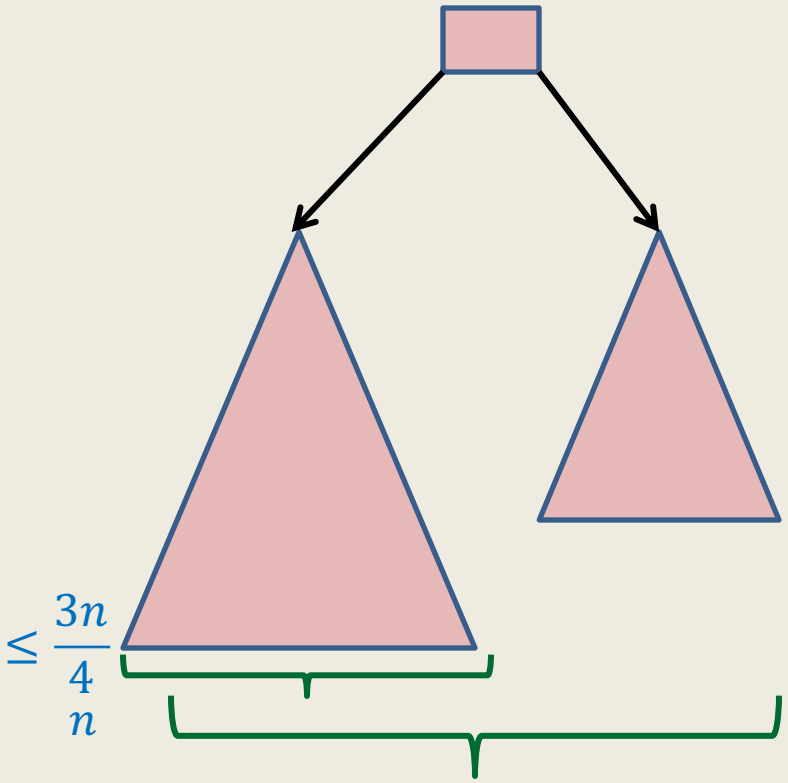


$$H(1) = 0$$

$$H(n) = \leq 1 + H\left(\frac{n}{2}\right)$$

Let us change this recurrence slightly.

How crucial is perfect balance of a BST ?



$$H(1) = 0$$

$$H(n) \leq 1 + H\left(\frac{3n}{4}\right)$$

$$\leq 1 + 1 + H\left(\left(\frac{3}{4}\right)^2 n\right)$$

$$\leq 1 + 1 + \cdots + H\left(\left(\frac{3}{4}\right)^i n\right)$$

$$\leq \log_{4/3} n$$

What lesson did you get from this recurrence ? Think for a while before going further …

Lesson learnt :
We may as well work with **nearly** balanced BST

Nearly balanced Binary Search Tree

Terminology:

size of a binary tree is the number of nodes present in it.

Definition: A binary search tree **T** is said to be **nearly balanced** at node **v**, if

$\text{size}(\text{left}(v)) \leq \frac{3}{4} \text{size}(v)$

and

$\text{size}(\text{right}(v)) \leq \frac{3}{4} \text{size}(v)$

Definition: A binary search tree **T** is said to be **nearly balanced** if it is **nearly balanced** at each node.

Nearly balanced Binary Search Tree

Think of ways of using **nearly balanced** **BST** for solving our dictionary problem.

You might find the following **observations/tools** helpful :

- If a node **v** is **perfectly balanced**, it requires **many insertions** till **v** ceases to remain **nearly balanced**.

- Any arbitrary **BST** of size n can be converted into a **perfectly balanced** **BST** in **O**(n) time.

Solving our dictionary problem

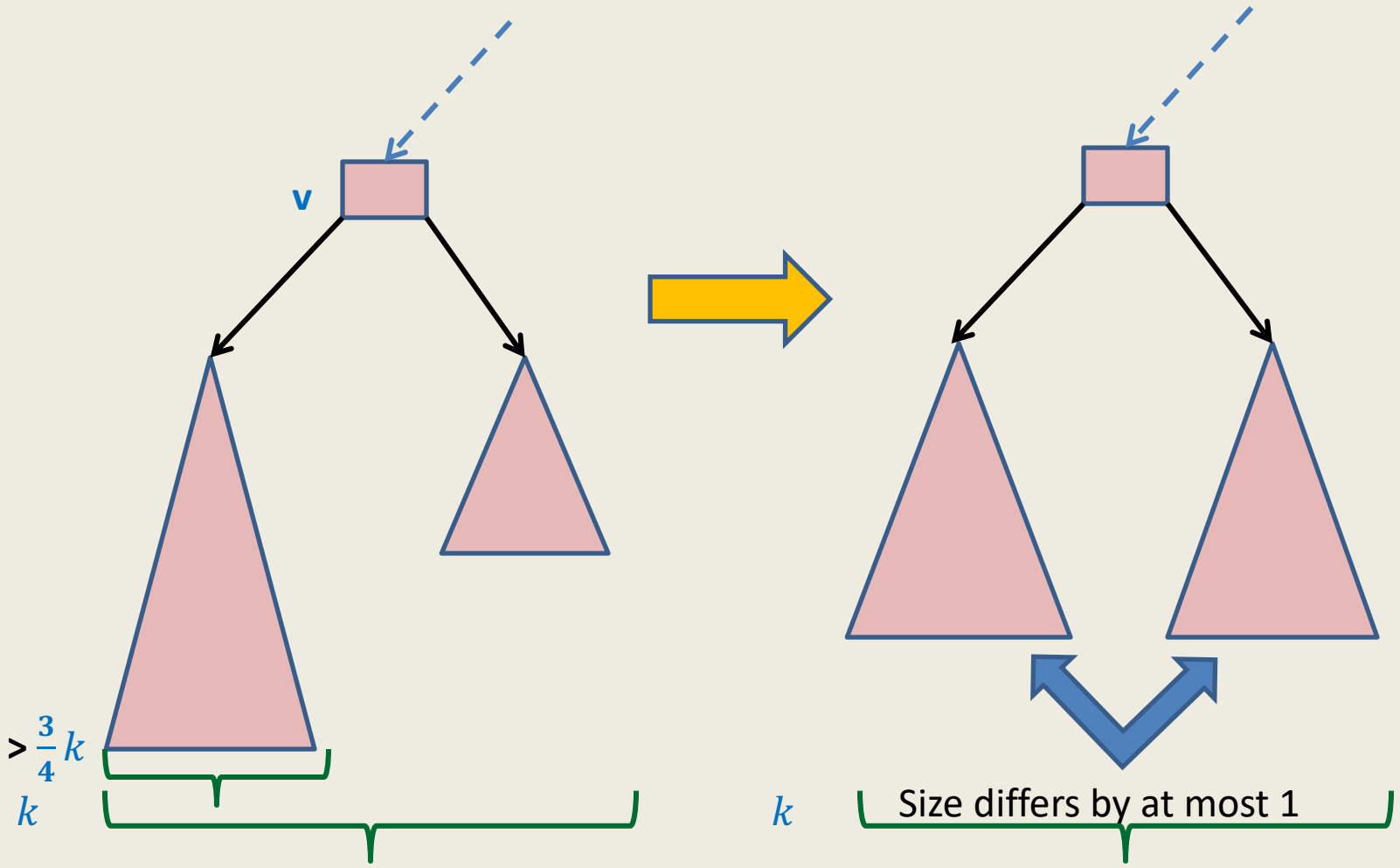
Preserving O(log n) height after each operation

- Each node v in **T** maintains an additional field **size**(v) which is the number of nodes in the **subtree**(v).

- Keep **Search**(**T**,x) operation unchanged.

- Modify **Insert**(**T**,x) operation as follows:
  - Carry out normal insert and update the **size** fields of nodes traversed.
  - If BST **T** is ceases to be **nearly balanced** at any node **v**, transform **subtree(v)** into **perfectly balanced BST**.

"Perfectly Balancing" subtree at a node v



What can we say about this data structure ?

It is elegant and reasonably simple to implement.

Yes, there will be huge computation for *some* insertion operations.

But the number of such operations will be rare.

So, at least intuitively, the data structure appears to be efficient.

Indeed, this data structure achieve the following goals:

- For any arbitrary sequence of n **operations**, total time will be **O(n log n)**.
- Worst case search time: **O(log n)**

How can we justify these claims ?

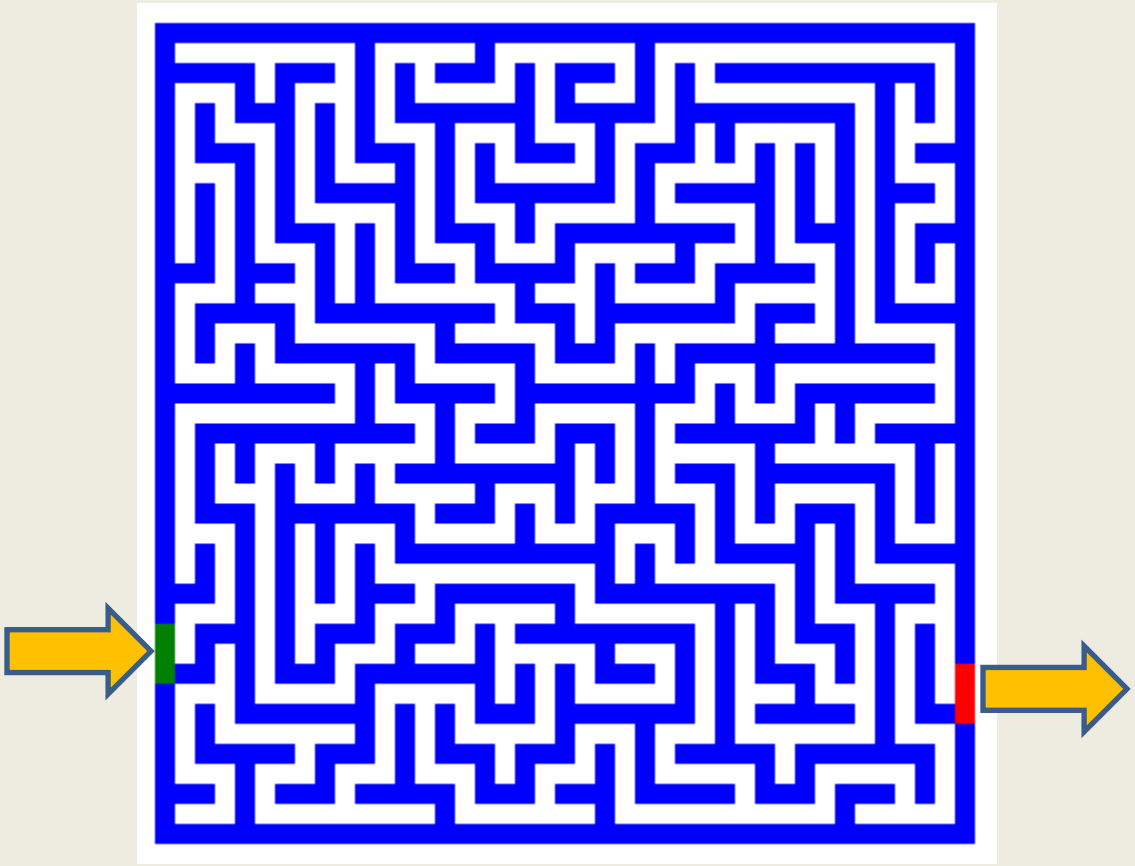
Keep thinking till we do it in a few weeks ☺.

Stack: a data structure

A few motivating examples

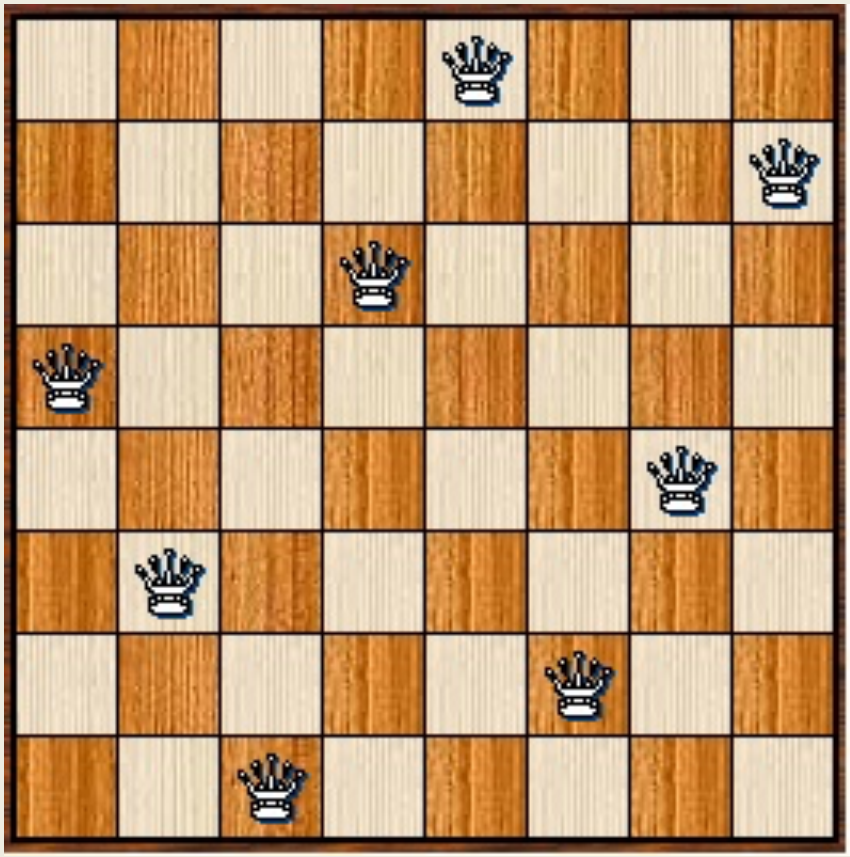
Finding path in a maze

Problem : How to design an algorithm for finding a path in a maze ?



8-Queens Problem

Problem: How to place **8 queens on a chess board** so that no two of them attack each other ?



Expression Evaluation

- $x = 3 + 4 * (5 - 6 * (8 + 9\text{^}2) + 3)$

Problem:

Can you write a program to evaluate any arithmetic expression ?

Stack: a data structure

Stack

Data Structure Stack:

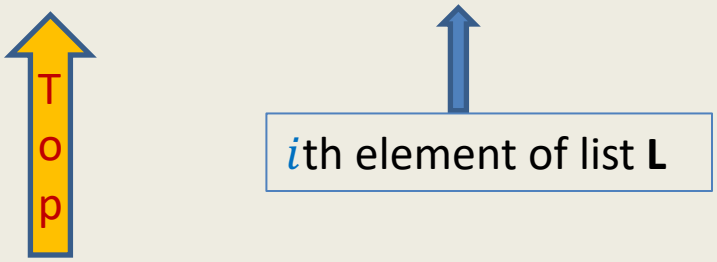
- **Mathematical Modeling of Stack**
- **Implementation of Stack**

Revisiting List

List is modeled as a sequence of elements.

we can insert/delete/query element at any arbitrary position in the list.

L : $a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n$

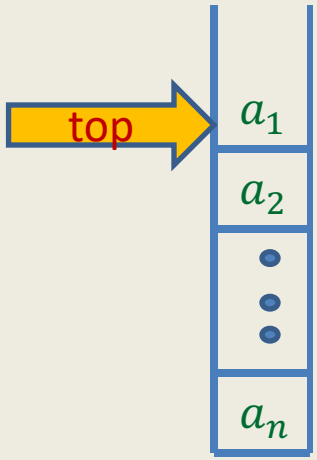


What if we restrict all these operations to take place only at one end of the list ?

Stack: a new data structure

A special kind of list

where all operations (insertion, deletion, query) take place at one end only, called the **top**.



Operations on a Stack

Query Operations

- **IsEmpty(S)**: determine if **S** is an empty stack
- **Top(S)**: returns the element at the top of the stack

  Example: If **S** is $a_1, a_2, \ldots, a_n$, **then Top(S)** returns a_1 .

Update Operations

- **CreateEmptyStack(S)**: Create an empty stack
- **Push(x,S)**: push **x** at the top of the stack **S**

  Example: If **S** is $a_1, a_2, \ldots, a_n$, then after **Push(x,S)**, stack **S** becomes

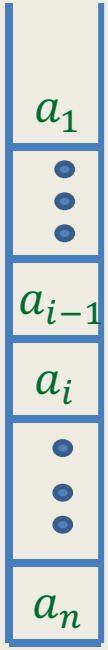
  $\mathbf{x}, a_1, a_2, \ldots, a_n$
- **Pop(S)**: Delete element from top of the stack **S**

  Example: If **S** is $a_1, a_2, \ldots, a_n$, then after **Pop(S)**, stack **S** becomes

  $a_2, \ldots, a_n$

An Important point about stack

How to access ith element from the top ?



- To access ith element, we must
  pop (hence delete) **one by one** the top $i-1$ elements from the stack.

A puzzling question/confusion

- **Why do we restrict** the functionality of a list ?
- **What will be the use** of such restriction ?

How to evaluate an arithmetic expression

Evaluation of an arithmetic expression

Question: How does a computer/calculator

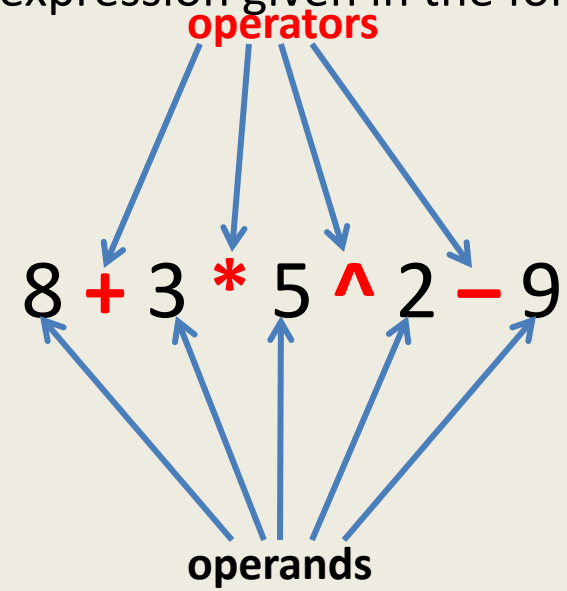
evaluate an arithmetic expression given in the form of a string of symbols ?

$$8 + 3 * 5 \wedge 2 - 9$$

Evaluation of an arithmetic expression

Question: How does a computer/calculator

evaluate an arithmetic expression given in the form of a string of symbols ?



First it splits the string into **tokens** which are operators or operands (numbers). This is not difficult. But how does it evaluate it finally ???

Precedence of operators

Precedence: "priority" among different operators

- Operator **+** has same precedence as **–**.
- Operator ***** (as well as **/**) has higher precedence than **+**.
- Operator ***** has same precedence as **/**.
- Operator **^** has higher precedence than ***** and **/**.

Associativity of operators

What is 2^3^2 ?

What is 3-4-2 ?

What is 4/2/2 ?

Associativity:

"How to group operators of same type ?"

A ● B ● C = ??

(A ● B) ● C **or** A ● (B ● C)

Left associative **Right** associative

A trivial way to evaluate an arithmetic expression

8 + 3 * 5^ 2 - 9

- First perform all ^ operations.
- Then perform all * and / operations.
- Then perform all + and - operations.

Disadvantages:

1. An ugly and case analysis based algorithm
2. Multiple scans of the expression (one for each operator).
3. What about expressions involving parentheses: 3+4*(5-6/(8+9^2)+33)
4. What about associativity of the operators:
   - 2^3^2 = 512 **and** not 64
   - 16/4/2 = 2 **and** not 8.

Overview of our solution

1. **Focusing on a simpler version of the problem:**
    1. Expressions without parentheses
    2. Every operator is left associative

2. **Solving the simpler version**

3. **Transforming the solution of simpler version to generic**

Step 1

Focusing on a simpler version of the problem

Incorporating precedence of operators through priority number

Operator	Priority
+ , -	1
* , /	2
^	3

Insight into the problem

Let o_i : the operator at position *i* in the expression.

Aim: To determine an order in which to execute the operators

8 + 3 * 5 ^ 2 – 9 * 67

Position of an operator **does** matter

Question: Under what conditions can we execute operator o_i immediately?

Answer: if

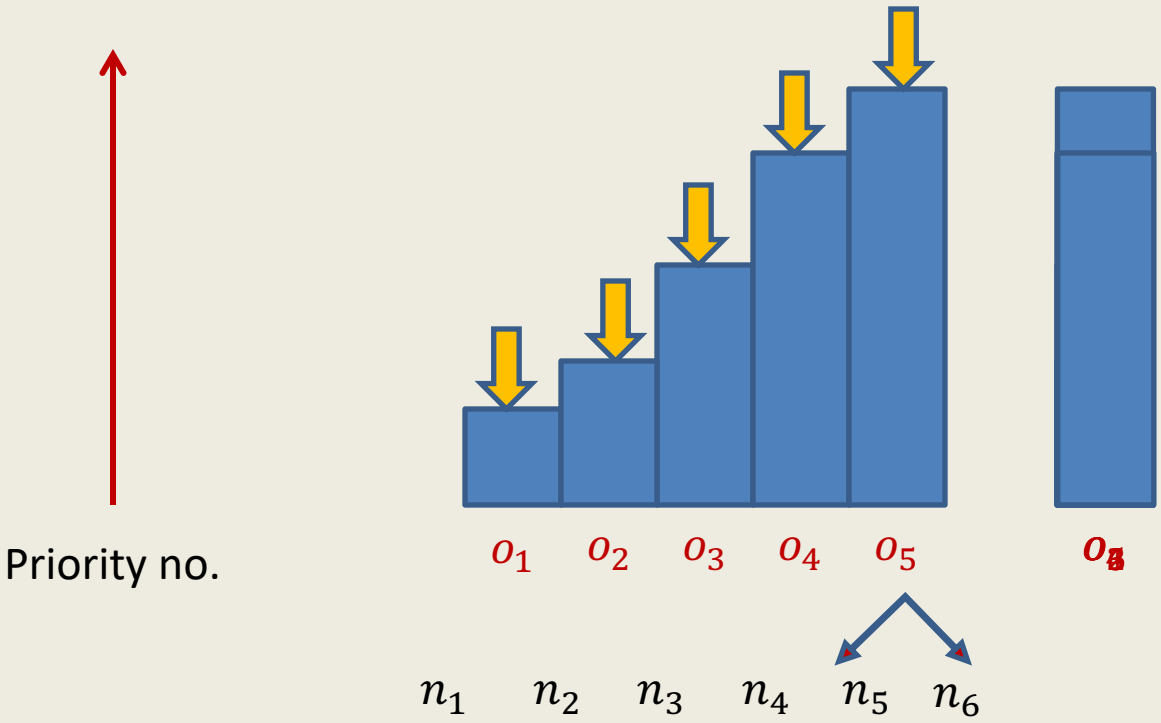
- priority(o_i) > priority(o_{i-1})
- priority(o_i) ≥ priority(o_{i+1})

Give reasons for ≥ instead of >

Question:

How to evaluate expression in a single scan ?

Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ n_6\ o_6$ …

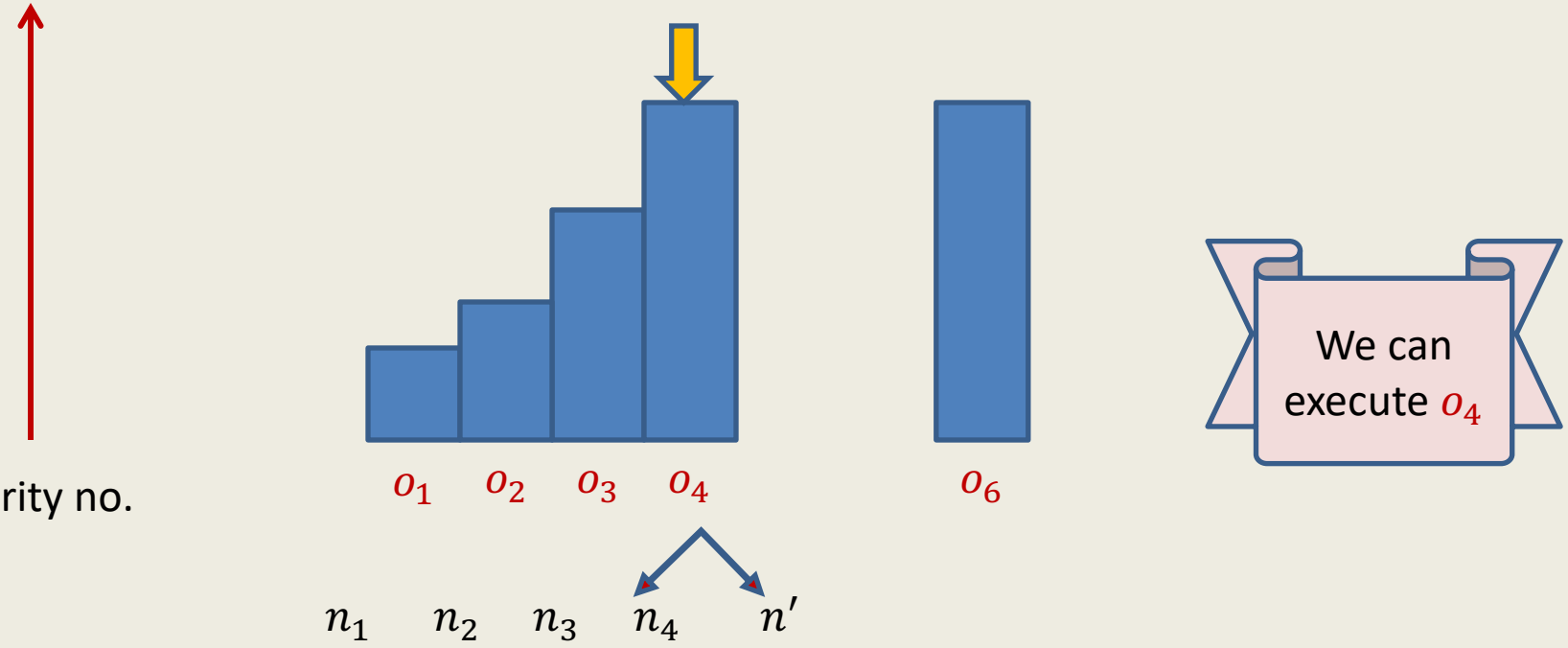


We can execute o_5

Question:

How to evaluate expression in a single scan ?

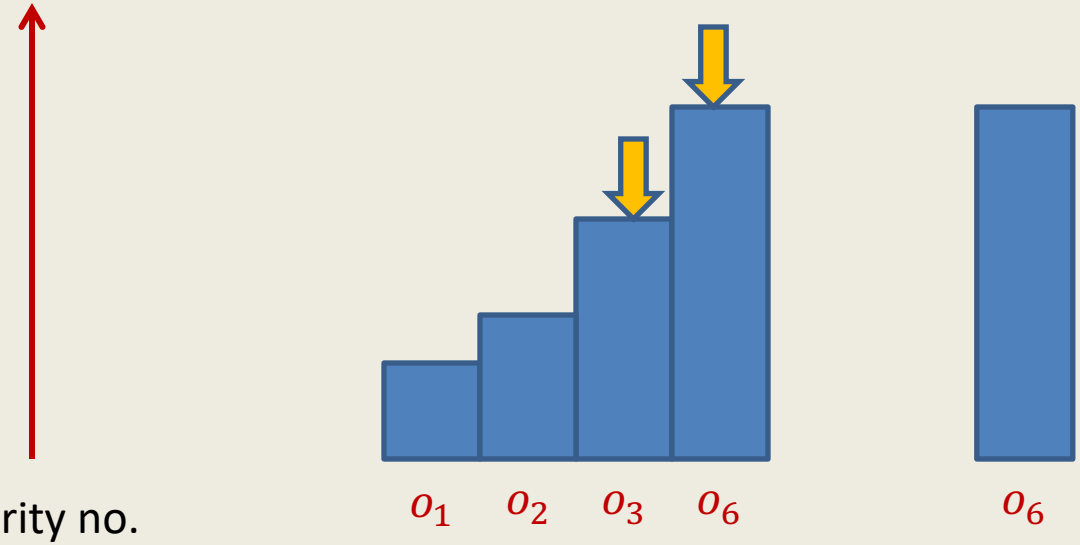
Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ n_6\ o_6$ …



Question:

How to evaluate expression in a single scan ?

Expression: $n_1 o_1\ n_2 o_2\ n_3\ o_3\ n_4\ o_4\ n_5\ o_5\ n_6\ o_6$ …



Homework:

Spend sometime to design an algorithm for evaluation of arithmetic expression based on the insight we developed in the last slides.

(hint: use 2 stacks.)