Data Structures and Algorithms

(ESO207)

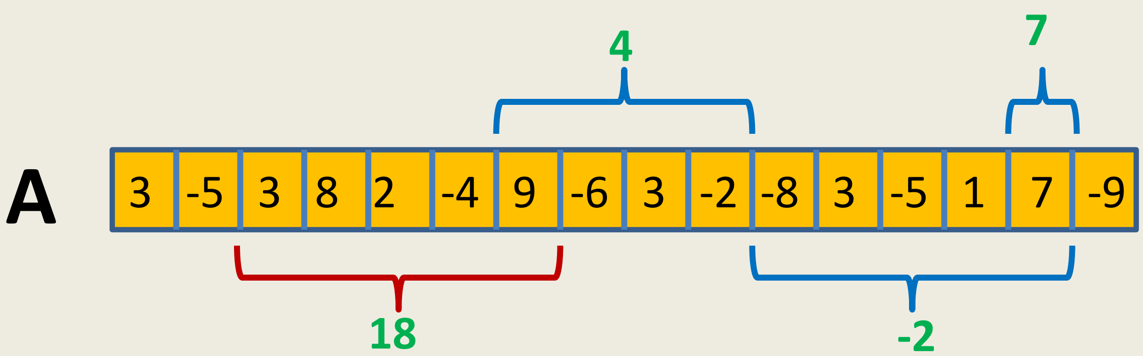
Lecture 4:

- 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;
}
```

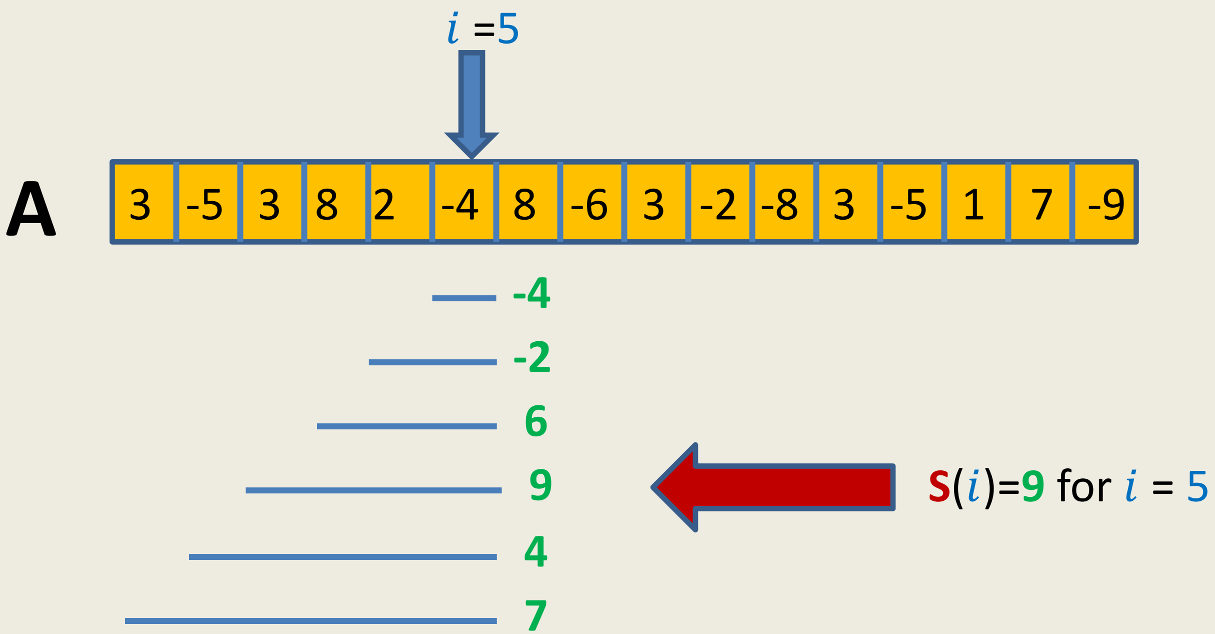
Time complexity = O(n^3)

```
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

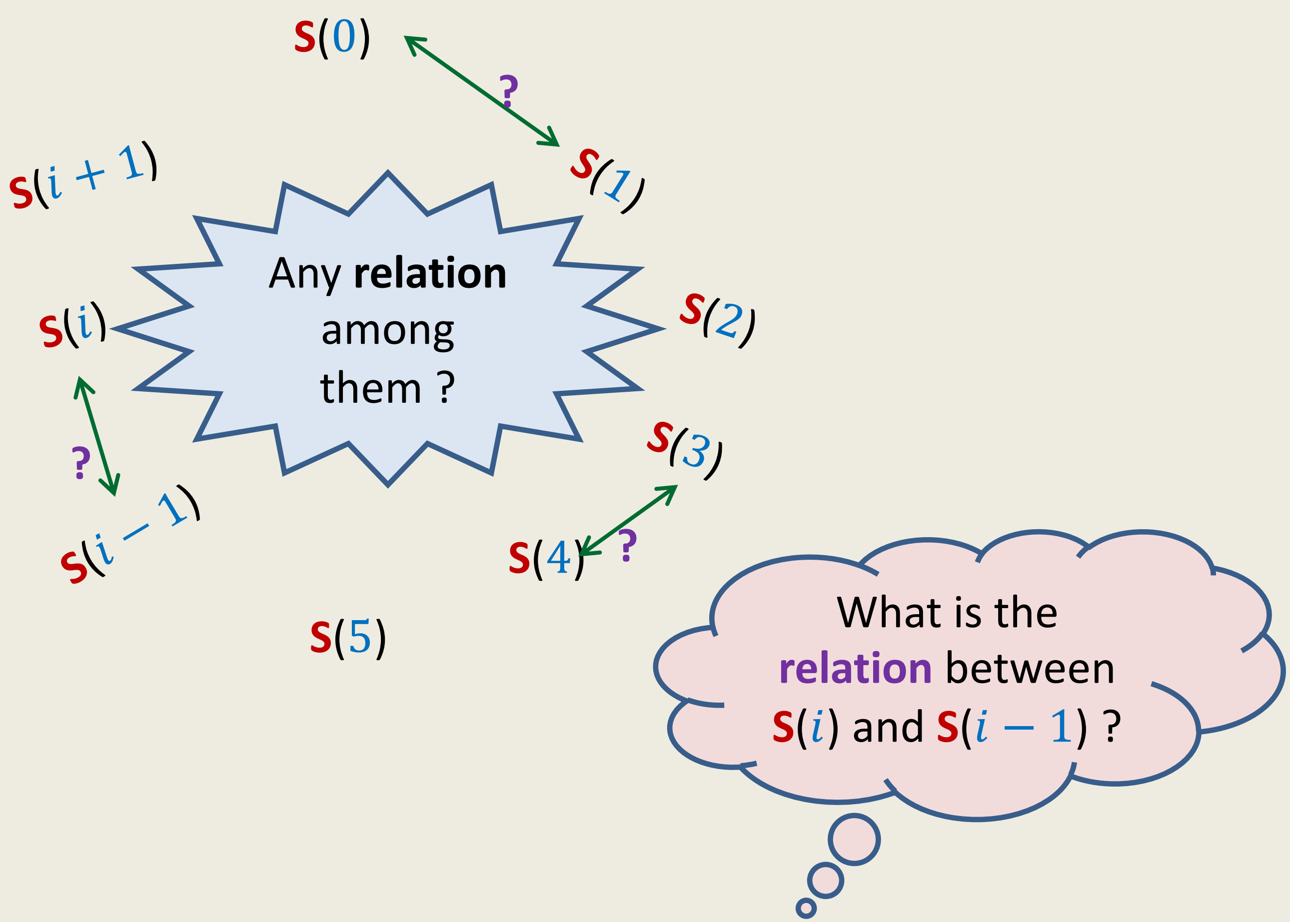
Observation:

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

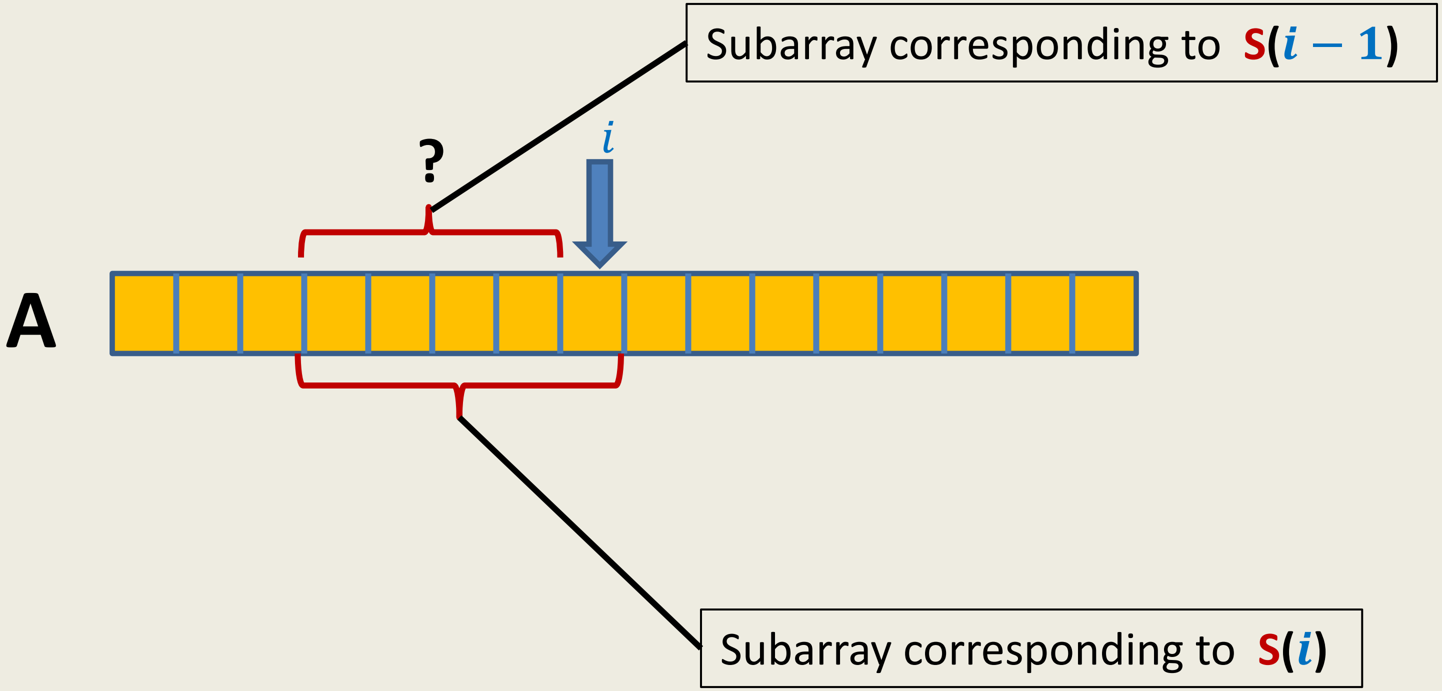
Question: If we wish to achieve O(n) time to solve the problem, how quickly should we be able to compute **S**(i) for a given index i ?

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

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



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[$0 \ \dots \ n-1$]**)**

{ **S**[0] ← **A**[0]; O(1) time

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

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

else S[i] ← **A**[i] O(1) time

}

"Scan **S** to return the maximum entry" **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**[$0 \; \ldots \; \boldsymbol{n-1}$])

{   **S**[0] ← **A**[0]

     for i = **1** to $\boldsymbol{n-1}$

     {   **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**[$0 \ \dots \ n-1$])

{ **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 ?

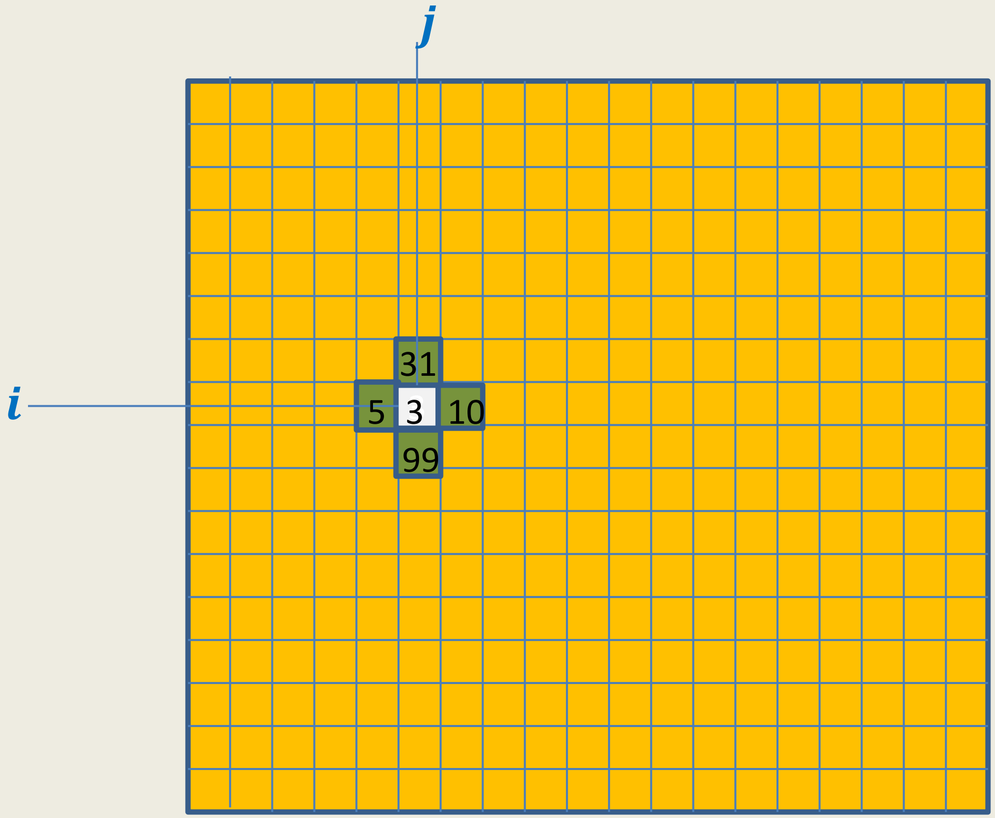
Ponder over this question before coming to the next class…

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.

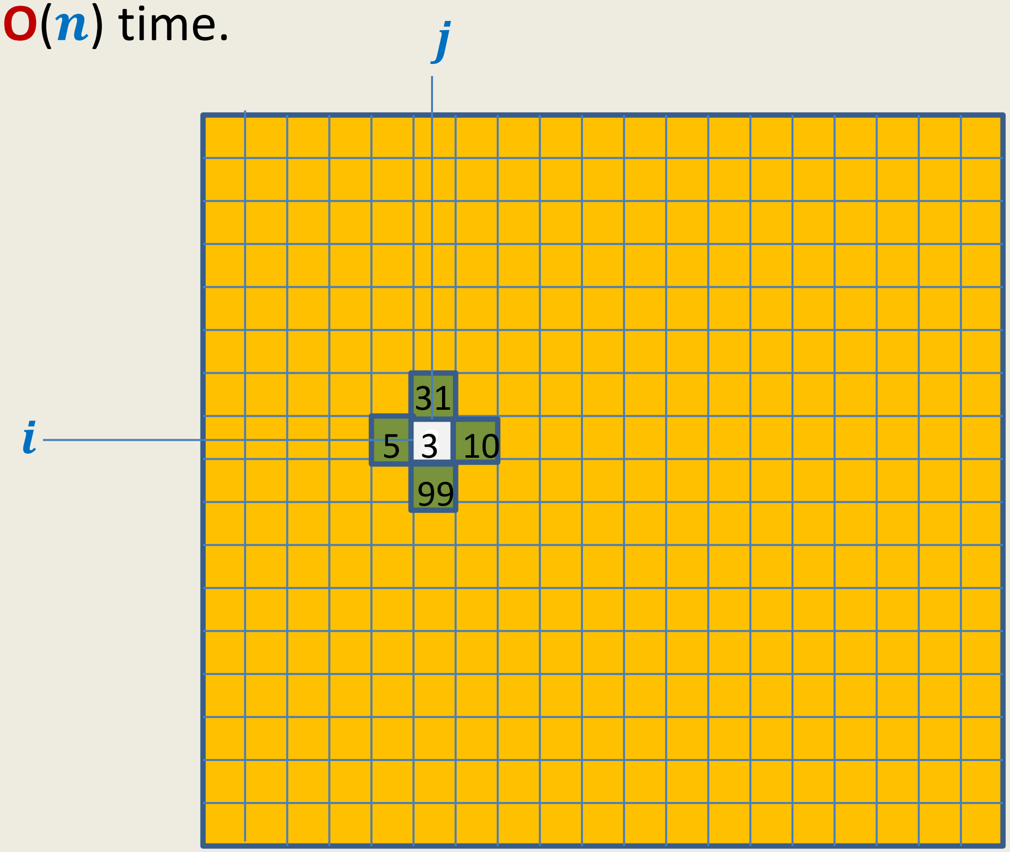


Does a **local minima** exist always ?

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 <u>does not require</u> any thing **magical** ☺!

Two simple principles

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

2. Principle of simplification:

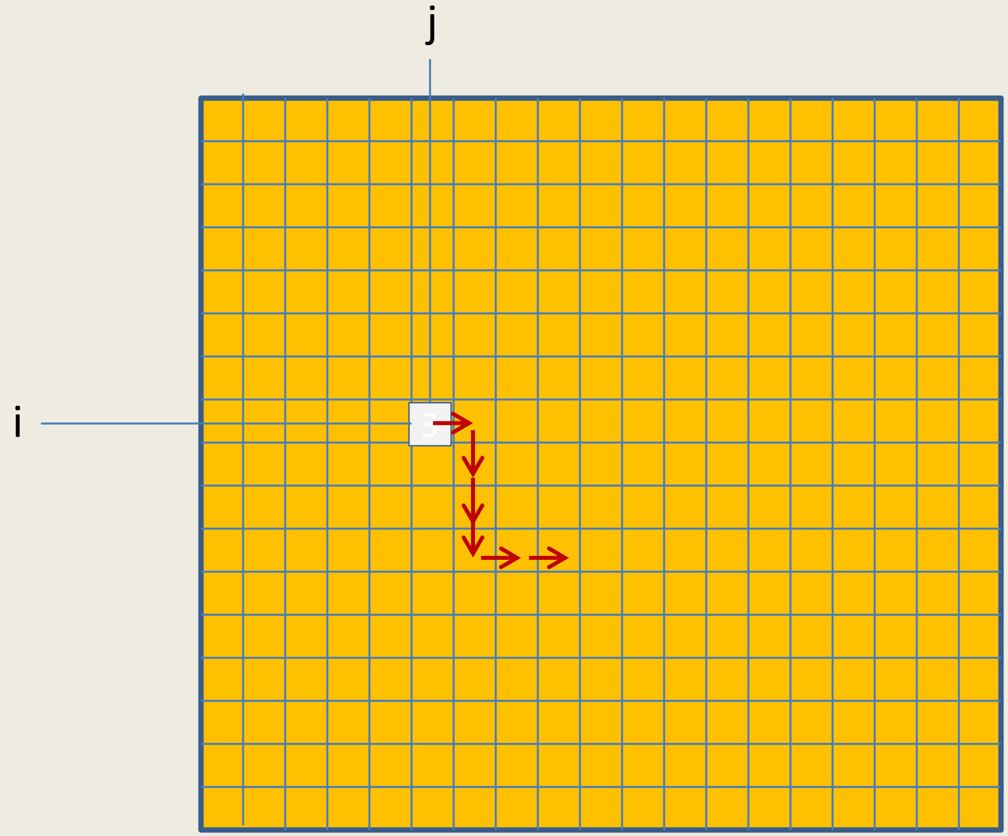
If you find a problem difficult,

→ 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 <u>smaller value</u>

}

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 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 ☺