First Paradigm-Reducing to subproblem of smaller size

Prepared by *Pawan K. Mishra*

A general theme…

Given a problem P of input size n,

assume that someone gives you the solution of same problem P, but for size n-1.

Can you solve the problem P for input size n?

If it is possible to find the solution of bigger problem using the solution of smaller problem– you can build a recursive solution.

But, at the same time you can solve it iteratively.

Think backward but solve forward…

If it is possible to find the solution of bigger problem using the solution of smaller problem– you can build a recursive solution. (Thinking backward)

But, at the same time you can solve it iteratively. (Solving forward)

Think backward but solve forward…

Backward Thinking

Here f is some function which will take solution given by sub problem and it will solve the problem for size n

$$\text{Problem}(n)= \begin{cases} f(\text{Problem }(n-1)) & \text{--- You do something with the solution of sub problem of size n-1} \\ \text{Problem}(n_0) & \text{------Base case} \end{cases}$$

Forward solving—iterative code

```
Problem($n_o$)
for (i =$n_0$+1 to n )
    Problem(i) $\leftarrow$ f(Problem(i-1))
```

Backward Thinking

$$\text{Problem}(n) = \begin{cases} f(\text{Problem}(n-1)) \\ \\ \text{Problem}(n_0) \end{cases}$$

Forward Solving

```
Problem(n_o)
 for (i =n_0+1 to n )
     Problem(i) ← f(Problem(i-1))
```

For iterative code, you can find time complexity by counting the number of steps

T(n)= time to solve the problem of size n-1+ time to use the solution given by the sub problem of size n-1, i.e. f here.

$\Rightarrow$ T(n)= T(n-1)+ time to implement Function f

Prepared by *Pawan K. Mishra*

Problems Discussed in the class

1. Finding maximum/minimum.
2. Finding maximum and second maximum.
3. Finding maximum and minimum.
4. Searching an element in an array.
5. Sorting an array.
6. Finding maximum (contiguous) subarray sum.

Note that sometimes we need to look for the solution of problem with size n-2 or n-3…..

For finding maximum and second maximum, it is wise to look for the sub problem of size n-2

1. Maximum

$T(n)=T(n-1)+1$

2. Maximum and Second Maximum

Discussed two approaches

a. $T(n)=T(n-1)+2= 2n-3$

b. $T(n)=T(n-2)+3=1.5n-2$

3. Similarly, we can do for maximum and minimum

4. Searching

$T(n)=T(n-1)+1.$

This is linear search

5. Sorting

$T(n)=T(n-1)+n$

This is insertion sort.

Maximum Subarray Sum

Given an integer array A, find the subarray with maximum sum.

Number of sub arrays- nC2+n=O(n^2).

Naïve way--- which will find sum for each subarray and return the maximum—O(n^3)

```
$Currmax=0$
$for\ i=1\ to\ n$
   $for\ j=i\ to\ n$
        $for\ k= i\ to\ j$
            $sum\ +=A[k]$
   $Currmax=max(Currmax,sum)$
```

Prepared by *Pawan K. Mishra*

Maximum Subarray Sum

Given an integer array A, find the subarray with maximum sum.

Number of sub arrays- nC2+n=O(n^2).

Naïve way--- which will find sum for each subarray and return the maximum—O(n^3)

```
Currmax=0
for i=1 to n
   for j=i to n
         for k= i to j
              sum +=A[k]
   Currmax=max(Currmax,sum)
```

Prepared by *Pawan K. Mishra*

Maximum Subarray Sum

Given an integer array A, find the subarray with maximum sum.

Number of sub arrays- nC2+n=O(n^2).

Naïve way--- which will find sum for each subarray and return the maximum—O(n^3)

```
$Currmax=0$
$for\ i=1\ to\ n$
   $for\ j=i\ to\ n$
        $for\ k=i\ to\ j$
             $sum\ +=A[k]$
   $Currmax=max(Currmax,sum)$
```

Improvement- which will go for each subarray, but will not calculate sum for each array every time, but rather use the computed value—O(n^2)

Prepared by Pawan K. Mishra

Maximum Subarray Sum

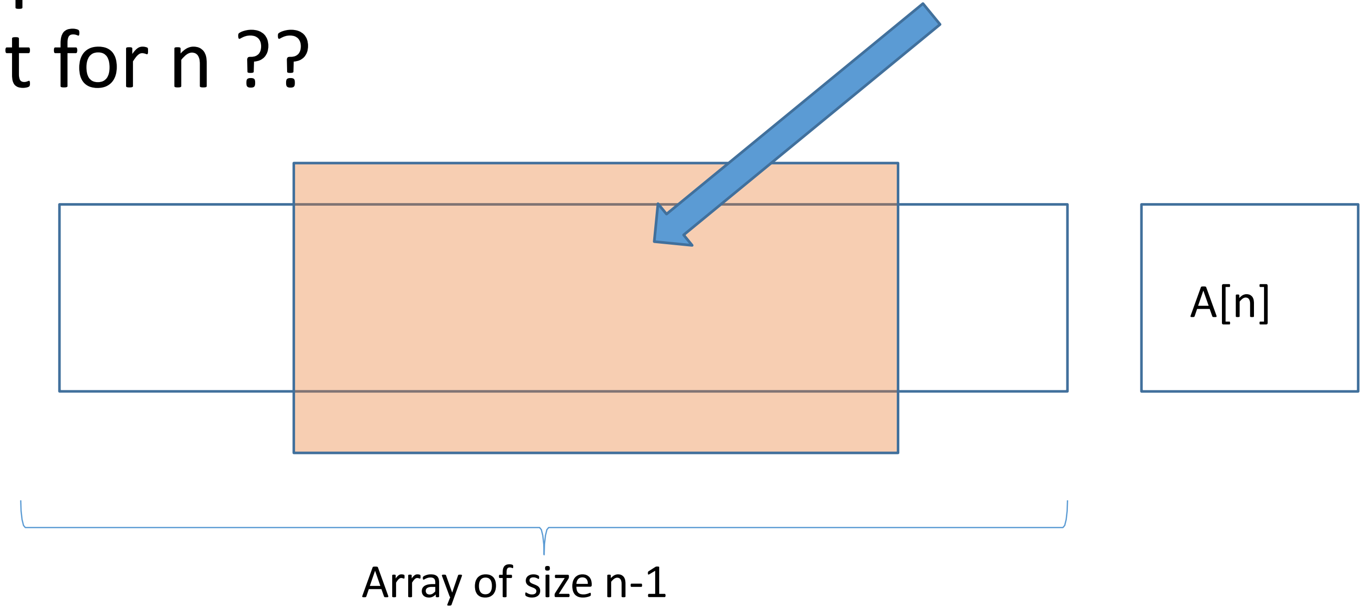
Using our learned paradigm

Same question– if we know the solution for size n-1, can we do it for n ??

Maximum Subarray Sum

Using our learned paradigm

Same question– if we know the solution for size n-1, can we do it for n ??



Prepared by *Pawan K. Mishra*

$$\text{MSA}(n) = \max \begin{cases} \text{MSA}(n-1) \\ \left.\begin{array}{l} A[n] \\ \text{Sum}(A[n-1], A[n]) \\ \text{Sum}(A[n-2], A[n-1], A[n]) \\ \quad\vdots \\ \text{Sum}(A[1], A[2], \ldots, A[n]) \end{array}\right\} \text{If max subarray contains } A[n], \text{ we need to check the following, which will take linear time, } O(n). \end{cases}$$

$$\mathrm{MSA}(n)= \max \left\{ \begin{array}{l} \mathrm{MSA}(n-1) \\ \left.\begin{array}{l} A[n] \\ \mathrm{Sum}(A[n-1], A[n]) \\ \mathrm{Sum}(A[n-2], A[n-1], A[n]) \\ \vdots \\ \mathrm{Sum}(A[1], A[2], \ldots, A[n]) \end{array}\right\} \end{array}\right.$$

If max subarray contains A[n], we need to check the following, which will take linear time, O(n).

$$T(n)=T(n-1)+n=O(n^2)$$

Prepared by *Pawan K. Mishra*

So, to get the overall time complexity O(n), we know that we cannot do extra work in linear time.

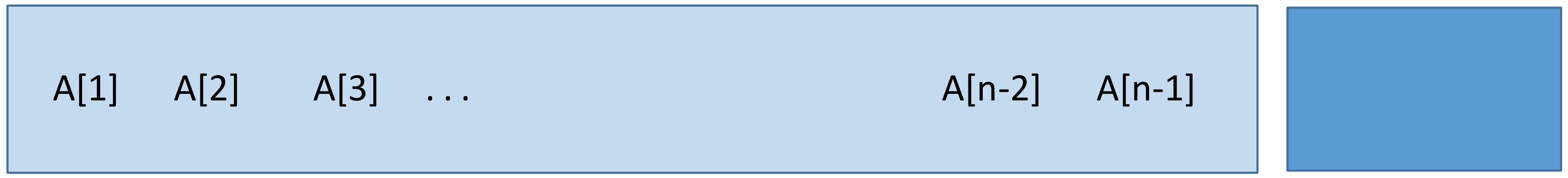
Thus, we will use our sub problem to give us some extra information, which will eventually leads to the overall time complexity in O(n)

$$MSA(n) = \max \begin{cases} MSA(n-1) \\ \boxed{\begin{array}{l} A[n] \\ Sum(A[n-1], A[n]) \\ Sum(A[n-2], A[n-1], A[n]) \\ \\ Sum(A[1], A[2], \ldots, A[n]) \end{array}} \end{cases}$$

To solve this
For problem size n, we
Want to invest constant
time.

max (Sum(A[n-1], A[n]), Sum(A[n-2], A[n-1], A[n]), …
,Sum(A[1], A[2], …, A[n]))

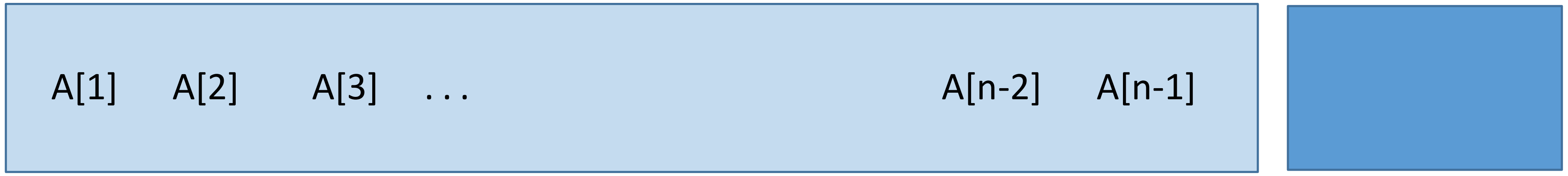
= A[n]+ max(Sum(A[n-1]), Sum(A[n-2], A[n-1]), …
,Sum(A[1], A[2], …,A[n-1])



Prepared by *Pawan K. Mishra*

max (Sum(A[n-1], A[n]), Sum(A[n-2], A[n-1], A[n]), …
,Sum(A[1], A[2], …, A[n]))

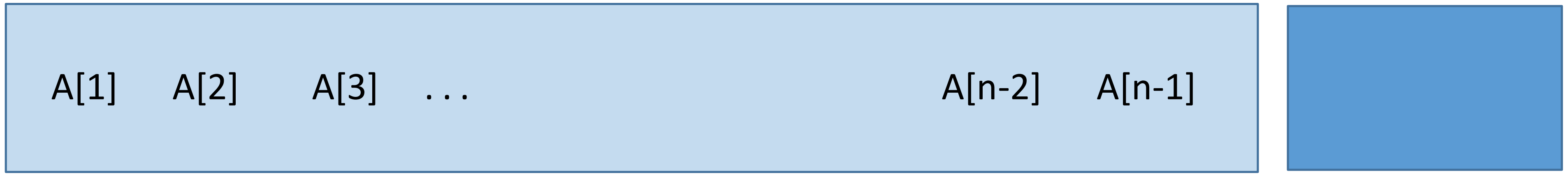
= A[n]+   max(Sum(A[n-1]), Sum(A[n-2], A[n-1]), …
,Sum(A[1], A[2], …,A[n-1])



Prepared by *Pawan K. Mishra*

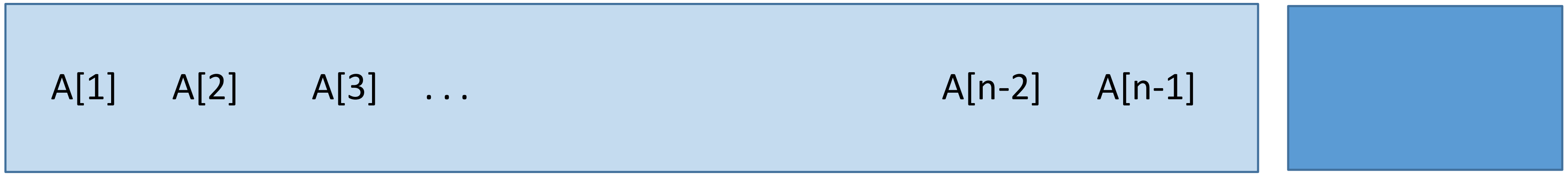
$$\max\left(\mathrm{Sum}(A[n-1], A[n]), \mathrm{Sum}(A[n-2], A[n-1], A[n]), \ldots, \mathrm{Sum}(A[1], A[2], \ldots, A[n])\right)$$

$$= A[n] + \max\left(\mathrm{Sum}(A[n-1]), \mathrm{Sum}(A[n-2], A[n-1]), \ldots, \mathrm{Sum}(A[1], A[2], \ldots, A[n-1]\right)$$



max (Sum(A[n-1], A[n]), Sum(A[n-2], A[n-1], A[n]), …
,Sum(A[1], A[2], …, A[n]))

= A[n]+ max(Sum(A[n-1]), Sum(A[n-2], A[n-1]), …
,Sum(A[1], A[2], …,A[n-1])



max (Sum(A[n-1], A[n]), Sum(A[n-2], A[n-1], A[n]), …
,Sum(A[1], A[2], …, A[n]))

= A[n]+ max(Sum(A[n-1]), Sum(A[n-2], A[n-1]), …
,Sum(A[1], A[2], …,A[n-1])

converted it to another problems on first
n-1 numbers

max (Sum(A[n-1], A[n]), Sum(A[n-2], A[n-1], A[n]), …
,Sum(A[1], A[2], …, A[n]))

= A[n]+ max(Sum(A[n-1]), Sum(A[n-2], A[n-1]), …
,Sum(A[1], A[2], …,A[n-1])

converted it to another problems on first
n-1 numbers

Include this it to the
subproblem

Formula for it

If we maintain Maximum Suffix according to this formula,

$$MS[i] = \max\{A[i], MS(i-1)+A[i]\}$$

We will get maximum sum of the array till i-th index.

Notion of max suffix sum

Array : 2, 3, -4, 3, -6, 7 -8

i : 1 2 3 4 5 6 7

i	max suffix sum till i-th position	elements in it
1	2	2
2	5	2, 3
3	1	2,3,-4
4	4	2,3,-4,3
5	-2	2,3,-4,3,-6
6	7	7
	-1	7, -8

Algorithm

```
MS=A[1]
MSA=A[1]

For i=2 to n
    MSA=max{ MSA ,   MS+A[i],   A[i]}
    MS= max{ A[i]   ,   MS+A[i]   }
Return MSA
```

Algorithm

```
MS=A[1]
MSA=A[1]

For i=2 to n
    MSA=max{ MSA ,   MS+A[i],   A[i]}
    MS= max{ A[i]   ,   MS+A[i]   }
Return MSA
```

The following
max(sum(A[n-1]), sum(A[n-2], A[n-1]), … ,sum(A[1], A[2], …,A[n-1])
Will be take care by this

Compute a^n

Given a and n, compute $\mathbf{a^n}$

$\mathbf{a^n = a^{n-1} * a}$

So, if we know solution for the subproblem of size n-1, we can solve it for the problem of size n using one extra multiplication. Assume that multiplication required constant operation.

$$T(n)=T(n-1)+1=O(n).$$

Assignment- USING THE LEARNED PARADIGM

Develop an algorithm to assist a trader operating under specific conditions for a particular commodity. The trader can only buy and sell the commodity once each during the season, with fluctuating daily prices. Additionally, the trader incurs a daily rent charge from the day of purchase until the day of sale. Naturally, the commodity can be sold only after it is bought.The objective is to create an O(n)-time algorithm that takes inputs including a list of commodity prices {p1, p2, ..., pn} for the n days in the past season and a rent rate r per day. The desired output is the maximum attainable profit. All basic arithmetic and comparison operations are considered to have a constant time complexity.

Sample input: Prices: 70, 100, 140, 40, 60, 90, 120, 30, 60. Rent per day: 15

Output: Max profit: 40

Buying at price 70, selling at 140, paying rent for the two days gives net profit of 40.