MAXIMUM SUB-ARRAY SUM

ALGORITHMS PROJECT

UNDERSTANDING THE PROBLEM

You are given an Array[] with n elements. You need to find the Maximum sum of Subarray.

Among all Subarrays of that array. A subarray of array A[] of length n is a contiguous segment from A[i] through A[j] where 0<=i<=j<=n

- if the array contains all non-negative numbers the maximum subarray is the entire array
- Several different sub. array may have the same maximum
- fun fact: this problem asked in Amazon, Facebook Microsoft

EXAMPLES:

Example 1:

Input: A[] = (-5, 8, 9, -6, 10, -15, 3}

Output: 21, the subarray {8, 9, -6, 10} has the maximum sum among all subarrays

Example 2:

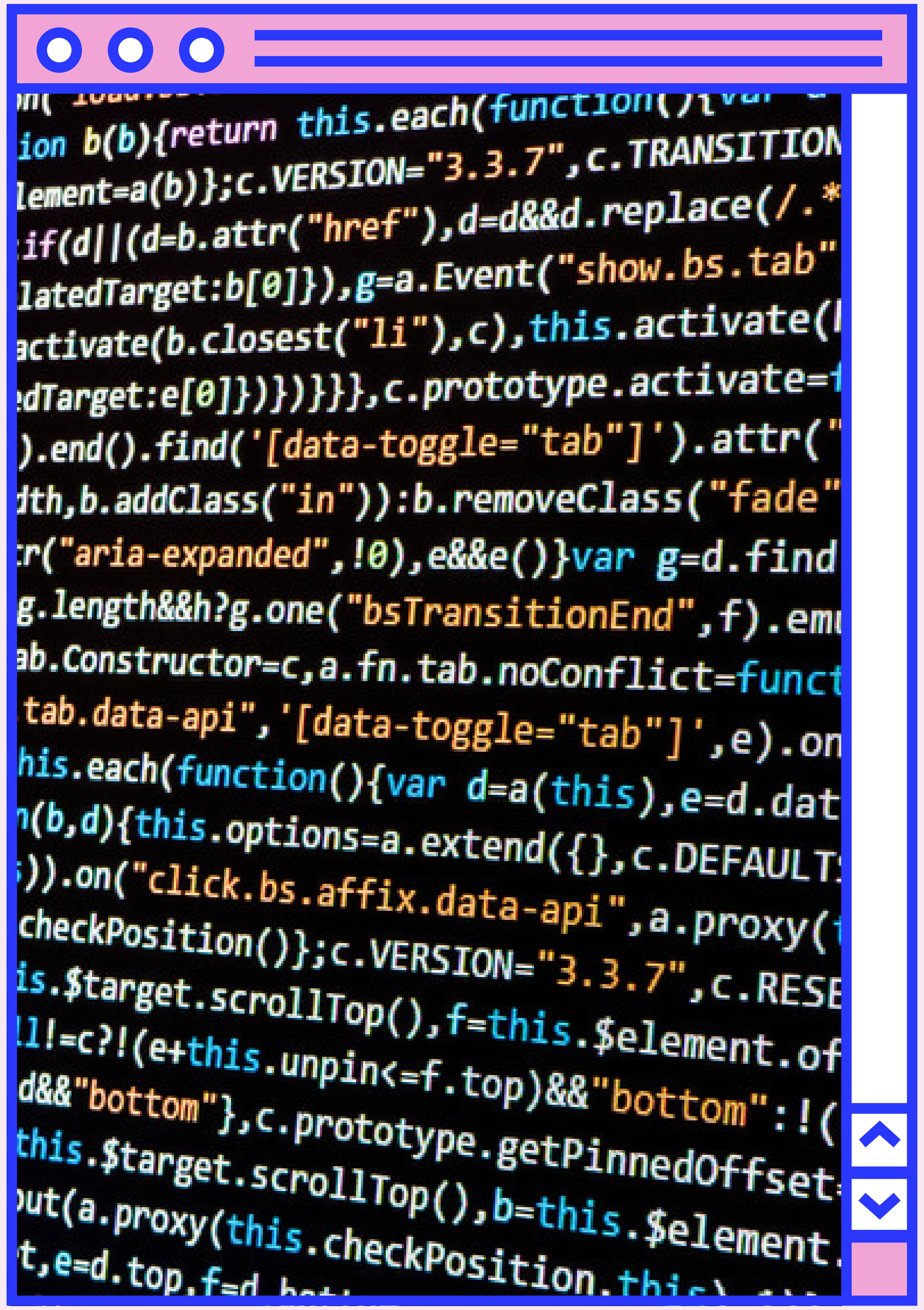
Input: A[] = {-4, -7,-1.5.-27

Output: 4, the subarray {-1, 5} has the maximum sum

Example 3:

Input: X[] = [-4, 5, 7, -6, 10, -15, 3], Output: 16

Explanation: The subarray [5. 7, -6, 10] has the maximum sum



BRUTE FORCE APPROACHE I

```
int max_sum = 0

for (i = 0 to n-1){
 for ( j = i to n-1) {
    int sum = 0
    for ( k = i to j) {
       Sum = sum + Array[k]

    if ( sum > max_sum)
      max_sum = sum
}
         }
return max_sum
}
```

algorithm that uses 3 nested loops, the first for loop to determine the starting point of the sub-array, the second for the for loop to determine the end point of the sub-array, and the last for loop to compute the sum between the starting point and ending point and compare the outputs to get the largest value.

The problem with this algorithm is that it cannot deal with an array of large size, a direct or positive relationship between the size of the array and the time it will take to show the result (an increase in the size of an array causes an increase in execution time).

TIME COMPLEXITY

int max_sum = 0	O(1)
for (*i* = 0 to n-1)	O(n)
for (*j* = i to n-1)	O(n)
int sum = 0	O(1)
for (k = i to j)	O(n)
Sum = sum + Array[k]	O(1)
if (sum > max_sum)	O(1)
max_sum = sum	O(1)
return max_sum	O(1)
worst case complexity of the code	O(n^3)

Brute force approach II

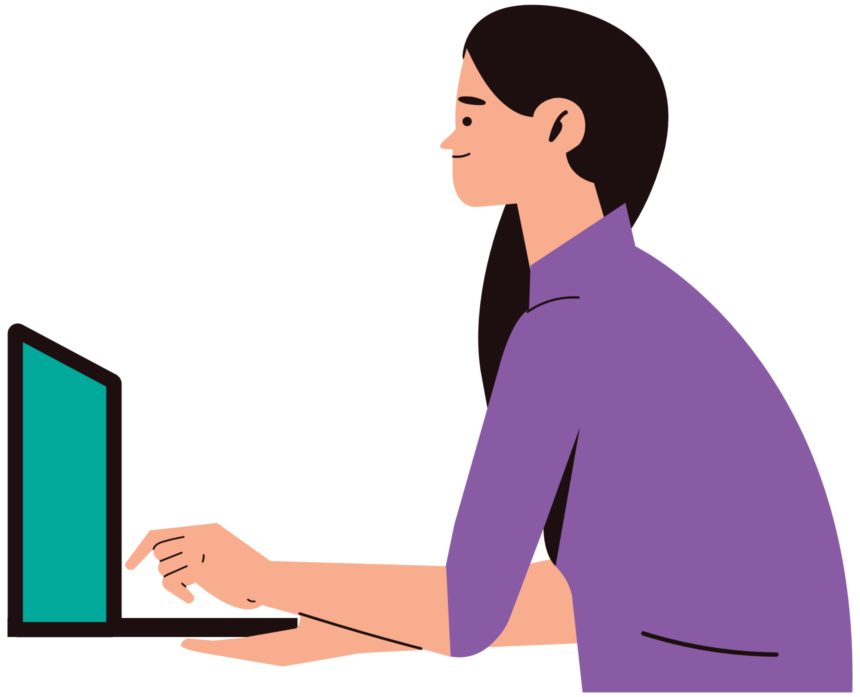
So The idea is to start at all
positions in the
array and calculate running sums.
The outer loop picks the beginning
element,
the inner loop finds the maximum
possible sum with the first element
picked by
the outer loop and compares this
maximum with

```
brute_forceII (arr, n)
max sum = 0
for (i= 0 to n-1)
sum=0
for(i=iton-1)
sum= sum+ arr[j]
if (sum > max_sum)
max sum = sum
retun max sum
```

TIME COMPLEXITY

`max_sum = 0`	O(1)
`for I in range(0,len(arr)):`	O(n)
`sum = 0`	O(1)
`for j in range(i,len(arr)):`	O(n)
`sum += arr[j]`	O(1)
`max_sum = max(sum,max_sum)`	O(1)
`return max_sum`	O(1)
worst case complexity of the code	O(n^2)

DYNAMIC PROGRAMMING APPROACH USING AN AUXILIARY ARRAY



Auxiliary

data structure is a fancy way and logical replication

of data from one or more columns of a table.

Auxiliary

data structures contain copies of, references to, or data

computed from base table data, deleting data from an auxiliary data structure

does not result in the loss of base table data. They are therefore derived

data.

Auxiliary array (aka, prefix sum) : is a simple yet powerful technique that allows for the

fast calculation of sums of elements in given contiguous segments of array, is a very vital tool in

competitive programming. thus

reduces the time complexity of our program

Applications:

- Evaluation of Arithmetic Expressions
- Sorting data
- Reverse a Data

```
MAXSUBARRAYSUM( A, N)
MAX_ENDING[N]
MAX_ENDING[0]= A[0]
FOR(I-1 TO N-1 ){
IF (A[I] + MAX_ENDING[I-1] >0)
 MAX_ENDING[I]= A[I] + MAX_ENDING[I-
1]
ELSE
 MAX_ENDING[I]= A[I]}
ANS = 0
FOR(I = 0 TO N-1 )
 ANS = MAX(ANS, MAX_ENDINGL[I])
```

TIME COMPLEXITY

max_ending[n]	O(1)
max ending[o] - A[0]	O(1)
for (i=1 to n-1)	O(n)
if (A[i] + max_ending[i-1] >0)	O(1)
max_ending[i] =A[i] + max_ending[i-1]	O(1)
max_ending[i] = A[i]	O(1)
ans=0	O(1)
for(i=0 to n-1)	O(n)
ans = max(ans, max_ending[i])	O(1)
worst case complexity of the code	O(n)

KADANE'S ALGORITHM

an Efficient Approach that attempt to improve the previous algorithms by merge the advantages of the Dynamic Programming using auxiliary array and the brute Force approach and Dynamic Programming approach Using an auxiliary array

```
maxSubarraySum(A, n)
max_so_far =0, max_ending_here = 0
  for (i=1 to n)
  max_ending_here += A[i]
  if( max_ending_here <0)
    max_ending_here =0
return max_so_far
```

TIME COMPLEXITY

max_so_far = 0	O(1)
max_ending_here = 0	O(1)
for (i=0 to n-1)	O(n)
max_ending_here += A[i]	O(1)
if(max_ending_here <0)	O(1)
max_ending_here= 0	O(1)
return max_so_far	O(1)
worst case complexity of the code	O(n)

Comparison of Solutions

ALGORITHM	TIME COMPLEXITY	SPACE COMPLEXITY
BRUTE FORCE APPROACH I	O(N^3)	O(1)
BRUTE FORCE APPROACH II	O(N^2)	O(1)
DYNAMIC PROGRAMMING USING AUXILIARY ARRAY	O(N)	O(N)
KADANE	O(N)	O(1)

REVAN ALQAHMI

NORAH ALSALEM

RINAD NASSER

RANIYA MASSOUD