# Maximum Sum Subarray Problem:

The challenge of finding a contiguous subarray with the biggest sum using a given one-dimensional array is known as the subarray problem. For example, the contiguous subarray with the biggest sum in the array [-2, -3, 4, -1, -2, 1, 5, -3] is [4, 1, -2, 1, 5] and the sum is 7.s

The problem has the following cases:

- If the array includes only non-negative values, the problem is solved. the largest subarray will be the entire array.
- If the array includes only non-positive values, a solution is any subarray of size 1 holding the array's maximum value.

We will solve this problem using python with different algorithms like brute force, divide and conquer and dynamic programming then compares their results.

The algorithm will be analyzed using two methods (Analytical – Empirical). In the Empirical Method we will rely on two testing method:

- timeit  Library: to calculate the execution time from the start of function call to the end of it.
- Leetcode: problem solving platform provide test cases and analysis to your algorithm.

- **Brute-Force Algorithm:**

  We will start with brute-force, by getting every sub-array and return the one with maximum sum (using recursive approach).

- In this function, the flag is to indicate if we need to choose the current item or not. When the flag become true, we will choose the current item and recurse or return zero. If the flag become false, we have the option to picking the current one or proceeding to the next one.

```
def maxSubArraySum(nums):
    def solve(i, must_pick):
        if i >= len(nums):
           return 0 if must_pick else -2147483648
        return max(nums[i] + solve(i+1, True), 0 if must_pick else solve(i+1, False))
    return solve(0, False)

st= timeit.timeit()
print("Maximum contiguous sum is:",maxSubArraySum(arr))
end= timeit.timeit()
print ("The Execution Time:",np.abs((end-st)*1000),str("milliseconds"))
```

```
Maximum contiguous sum is: 7
The Execution Time: 4.548508999960177 milliseconds
```

- **Theoretical analysis:**

  For n=0 then O(0)=0

  So it will be $T(n) = T(n-1) + n \ when \ n > 0$

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

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

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

$$= T(n-i) + (n-i+1) + \dots$$

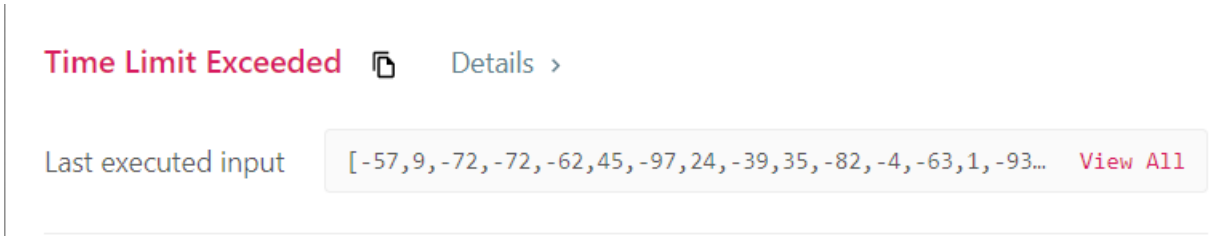
$$= \frac{n(n+1)}{2} + 1 = \frac{n^2+n+2}{2}$$

$$so \ it \ will \ be \ approximately \ O(n^2)$$

Time Complexity: $O(n^2)$ because the code tries every possible subarray sum.

Space Complexity: O(n), the recursive space for the code.

Timeit Library: shows that the execution time of the function is **4.548 ms.**




Leetcode: we got **Time Limit Exceeded** when test the code using large test case.

## ● Divide and Conquer:

- The recursive function will take 3 parameters (a: array, l is the left half of the array, r is the right half of the array). initialize the l with 0 and r with n-1.
- if left==right then it will be a one element only in the array and return it.
- round down the middle by math.floor() and it computed by (l+r)/2.
- initialize the leftmax and rightmax with -99999 and sum with 0.
- get the leftmax sum from the left half and rightmax sum from the right half.
- recursively get the maximum from both left and right (maxlr).
- return the maximum from 3 maximums (max of left, max of right and leftmax +rightmax).

```
def maximumsumsub(a,l=None,r=None ):

    if(l==None and r== None):
        l=0
        r=len(a)-1
    if(r==l):
        return a[l]
    m=math.floor((l+r)/2)
    lmax=-99999
    suum=0
    for i in range(m,l,-1):
        suum+=a[i]
        if(suum>lmax):
            lmax=suum
    rmax=-99999
    suum=0
    for i in range(m+1,r+1):
        suum+= a[i]
        if(suum>rmax):
            rmax=suum
    maxlr=max(maximumsumsub(a,l,m),maximumsumsub(a,m+1,r))
    return max(lmax+rmax,maxlr)

st= timeit.timeit()
print("Maximum contiguous sum is:",maximumsumsub(arr))
end= timeit.timeit()
print ("The Execution Time:",np.abs((end-st)*1000),str("milliseconds"))
```

```
Maximum contiguous sum is: 7
The Execution Time: 1.2580850000176724 milliseconds
```

- **Theoretical analysis:**

for n=1 then T(1)= O(1)

$T\left(\frac{n}{2}\right)$ *for each half part so it will be* $T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) = 2T(\frac{n}{2})$

Leftmax and rightmax combined will be $O(n)$

So it will be $T(n) = 2T\left(\frac{n}{2}\right) + O(n) \quad when\, n > 1$

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

$$= 2[2T\left(\frac{n}{4}\right) + \frac{n}{2}] + n$$
$$= 4T\left(\frac{n}{4}\right) + 2n$$
$$= 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$$
$$= 8T\left(\frac{n}{8}\right) + 3n$$

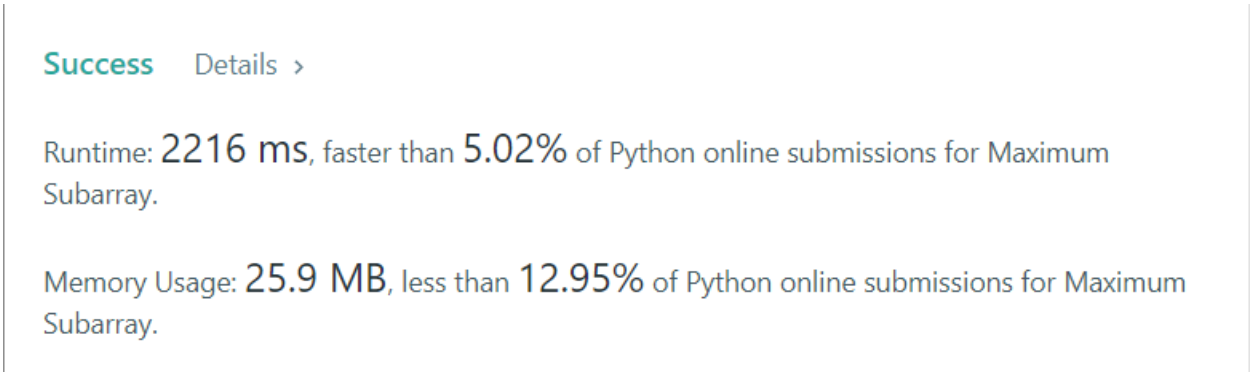
So the sequence will be $2^k T\left(\frac{n}{2^k}\right) + kn$

k=log2 n

Time Complexity: O(n log2 n).

Space Complexity: O(log n), the recursive space for the code.

Timeit Library: shows that the execution time of the function is **1.258 ms**.




Leetcode: after testing we got **Runtime = 2216 ms & Memory Usage = 25.9 MB**.

- **Kadane's Algorithm:**

```
max_so_far = arr[0]
curr_max = arr[0]

def maxSubArraySum(arr,i,n,max_so_far,curr_max):
    if i >= n:
        print("Maximum contiguous sum is:",max_so_far)
        return
    curr_max = max(arr[i], curr_max + arr[i])
    max_so_far = max(max_so_far,curr_max)
    maxSubArraySum(arr,i+1,n,max_so_far,curr_max)

st= timeit.timeit()
maxSubArraySum(arr,1,len(arr),max_so_far,curr_max)
end= timeit.timeit()
print ("The Execution Time:",np.abs((end-st)*1000),str("milliseconds"))
```

```
Maximum contiguous sum is: 7
The Execution Time: 2.63337399996999 milliseconds
```

The basic idea behind Kadane's algorithm is to search the array for all subarray curr_max. then record the greatest subarray sum max_so_far When we obtain a positive total, we compare it to max_so_far and update max_so_far if it is the maximum.

**Theoretical analysis:**

for n=1 then O(1)=1

So it will be $T(n) = T(n) + O(1) \quad when\, n > 1$

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

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

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

$$= (T(n-3) + 1) + 3$$

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

After substitution with 1 will be = $O(n)$

Time Complexity: O(n) for only one iteration.

Space Complexity: O(1) used space.

Timeit Library: shows that the execution time of the function is **2.633 ms.**




Leetcode: after testing we got **Runtime = 669 ms & Memory Usage = 25.5 MB**.

- **Dynamic Programming – Tabulation:**

```
def maxSubArray(nums):
        dp = [[0]*len(nums) for i in range(2)]
        dp[0][0], dp[1][0] = nums[0], nums[0]
        for i in range(1, len(nums)):
            dp[1][i] = max(nums[i], nums[i] + dp[1][i-1])
            dp[0][i] = max(dp[0][i-1], dp[1][i])
        return dp[0][-1]

st= timeit.timeit()
print("Maximum contiguous sum is:",maxSubArray(arr))
end= timeit.timeit()
print ("The Execution Time:",np.abs((end-st)*1000),str("milliseconds"))
```

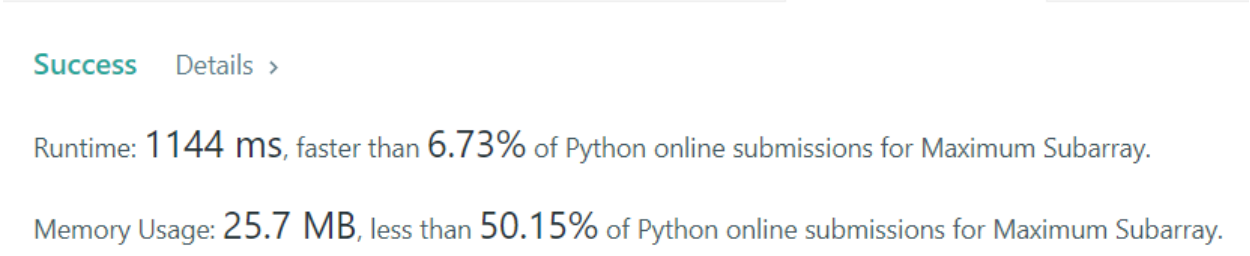
```
Maximum contiguous sum is: 7
The Execution Time: 2.8778549999515235 milliseconds
```

This solution is very similar to the last one. It's the same logic but using bottom-up approach. We can directly access previous values from the table.

Time Complexity: O(n), we use two iteration O(n) + O(n) = O(n).

Space Complexity: O(n) space for the table.

Timeit Library: shows that the execution time of the function is **2.877 ms.**




Leetcode: after testing we got **Runtime = 1144 ms & Memory Usage = 25.7 MB**.

## Discussion:

| | Time Complexity | Space Complexity | TimeIt | Leetcode |
|---|---|---|---|---|
| Brute-Force | $O(n^2)$ | O(n) | 4.548 ms | **Time Exceeded** |
| Divide&Conquer | O(n log2 n) | O(log n) | 1.258 ms | 2216 ms |
| DP - Kadane | O(n) | O(1) | 2.633 ms | **669 ms** |
| DP - Tabulation | O(n) | O(n) | 2.877 ms | 1144 ms |

After running the 4 algorithms and record their results as the following table:

After compare between the algorithms results, we found that:

- The memory usage in all algorithms is almost the same.
- the Brute-Force has the worst time complexity, and we cannot use it to handle a large data. In the other hand divide & conquer and Dynamic Programming techniques got a good time complexity.
- The Dynamic Programming techniques are more efficient than the Divide and Conquer with time complexity O(n).

- Kadane’s Algorithm take the shortest time according to leetcode tests.

## References:

*[1] Tadao, T."Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication",Electronic Notes in Theoretical Computer Science, Vol 61, 2002, https://doi.org/10.1016/S1571-0661(04)00313-5.*

*[2] "Maximum subarray problem", Wikipedia, Wikimedia Foundation, 2017. https://en.wikipedia.org/wiki/Maximum_subarray_problem.*

*[3] Malik, Hemant & Daescu, Ovidiu. (2018). k-Maximum Subarrays for Small k: Divide-and-Conquer made simpler.*

Colab Live Demo: https://colab.research.google.com/drive/1VaTmQw_9LYHatCtj-guSV4-QHu66q11n?usp=sharing