Reflections 2: Array query

1) Maximum subarray problem

o(n2) solution:

```
max=0;
arr=[-1,2,3,4,5,6]
l1=len(arr)
sum=0
for i in range(l1):  #l1 is the array length
        for j in range(l1):
                sum= sum+arr[i] + arr[j]
                print(sum)
                if(sum>max):
                        max=sum
print(max)
# we can also have an another array to store cumulative results for every 'i'
```

This works with input size is 10^3 but when input size increases we need to find more optimal solution so

<table>
<tr>
<td>


Yessss!. Kadane's algorithm
</td>
<td>
K *is for* kindness, you always show.

A *is for* aptitudes, your special capabilities shine

D *is for* desire, your thoughts do aspire.

A *is for* adaptable, whenever things change

N *is for* notable, distinguished are your feats

E *is for* enamoured, forever *will your children be with you and wide spread your algorithm.*
</td>
</tr>
</table>

Okay Enough of it Now!.

Kadane's algorithm works in Time complexity **O(n)**

Dynamic programming approach:

Reflections 2: Array query

If the sub array would end at any particular index, what would be my maximum contiguous

subarray sum?

Steps:

- At a given index, i should choose between the max of two numbers max(n1,n2) where n1 is the value at previous index and n2 is the value at the given index plus the value at n1 (previous index).
- Whichever is the max its value is stored as value at that given index.

```
def max_subarray(A):
   max_ending_here = max_so_far = A[0]
   for x in A[1:]:
      max_ending_here = max(x, max_ending_here + x)
      max_so_far = max(max_so_far, max_ending_here)
   return max_so_far
```

This is O(n) solution for Maximum subarray problem

2) LookUp Table Technique:

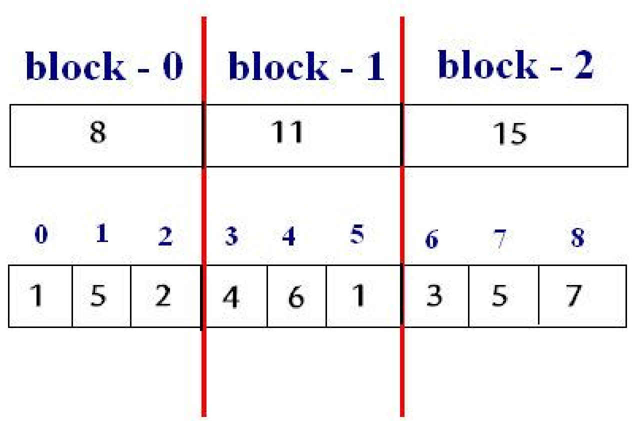
This technique is used when input is n^5 , we precompute the results in form a table and query this table for any input.

It uses memory space of n*n bytes and efficiency is O(n*n) and fetch time is O(1).

The memory utilization is more compared to others since 2D matrix is constructed for a given array.

3) Square-Root Decomposition:

It builds a 1D array of size=sqrt(size of original array) And query is fetched in O(sqrt(n)) time where n is the size of the original array.



Reflections 2: Array query

This original array is divided into sqrt(size(original array)) ie here len(arr)==8 and number of blocks formed by sqrt decomposition is Three and hence the search space is reduced
Eg if index 6 is given some query 1) we do 6/3 ie query index/number of blocks we get 2 and
2) To find its index in that block we do mod operation ie 6%3 we get 0 hence we query at index 0 of block 2
Demerits

When the search space is between these blocks we will have to combine two blocks and search for the query index.
This can be overcomed by Segment Trees.
PS: Explanation on segment trees will be added in the next commit.