Maximum sum/product subarray

problem: Given a Array, find the subarray with largest sum/prduct
Note: This problem is different from **maximum sum subarray of size k** . Here we need to find all the subarray of size k and find that subarray whose elements sum is maximum.
Here we just need to find subarray with maximum sum.

```
// subarray with largest sum
class Solution {
    public int maxSumSubArray(int[] nums) {
        if(nums.length == 0)
            return 0;
        int s = 0;
        int max = Integer.MIN_VALUE;
        for(int i=0; i < nums.length; i++){
            for(int j=i; j < nums.length ; j++){
              s += nums[j];
              max = Math.max(max,s);
            }
            s = 0;
        }
        return max;
    }
Time complexity : O(n^2)
```

1. **Dynamic Programming, Kadane's Algorithm**

The simple idea of Kadane’s algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this). Each time we get a positive-sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

```
public int maxSubArray(int[] nums) {
        if(nums.length == 0)
            return 0;
       int curr_max_sum = Integer.MIN_VALUE, max_ending_here = 0;
        for(int i = 0; i < nums.length; i++){
            max_ending_here  += nums[i];
            if(curr_max_sum < max_ending_here){
                curr_max_sum = max_ending_here;
            }
```

```
            if(max_ending_here < 0){
                max_ending_here = 0;
            }
        }

        return curr_max_sum;
    }


// If we have to print subarray with maximum sum, keep track of starting and
ending index of subarray.

  public int[] printMaxSumSubArray(int[] arr){
   if(arr.length == 0)
      return new int[1];
    int currMax = Integer.MIN_VALUE, maxEndHere = 0, start = 0, end = 0, big = 0;
    for(int i=0; i < arr.length; i++){
       maxEndHere += arr[i];
       if(maxEndHere < 0){
           maxEndHere = 0;
           big = i+1;
       }else if(maxEndHere > currMax){
         currMax = maxEndHere;
          start = big;
           end = i;
       }
    }
   int[] res = new int[end-start];
   int l = 0;
   for(int i= start; i < end; i++){
      arr[i] = res[l++];
    }
    return res;
  }
```

Time complexity : $O(n^2)$

- **max product subarray**

```
public int maxProduct(int[] nums) {
        if(nums.length == 0)
            return 0;
        int P = 1;
        int max = Integer.MIN_VALUE;
        for(int i=0; i < nums.length; i++){
            for(int j=i; j < nums.length ; j++){
              P *= nums[j];
              max = Math.max(max,P);
            }
```

```
            P = 1;
        }
        return max;
    }
}
```

Time Complexity: $O(n^2)$

Max product can be obtained by product of two positive number and two negative number.
so we will maintain maxval represents max product till current index and minval will represent product of negative integer till current index
and maxProduct will hold overall max product.

```
public int maxProduct(int[] nums) {
        if(nums.length == 0)
            return 0;
        int minval = nums[0], maxval = nums[0], maxProduct = nums[0];


        for(int i=1; i < nums.length; i++){

        // if current index is negative then max will become min and min will
become
        max after product, so swap  max and min values
            if(nums[i] < 0){
                int temp = maxval;
                maxval = minval;
                minval = temp;
            }

            maxval = Math.max(nums[i],nums[i]*maxval);
            minval = Math.min(nums[i],nums[i]*minval);

            maxProduct = Math.max(maxProduct, maxval);
        }
        return maxProduct;
    }
```