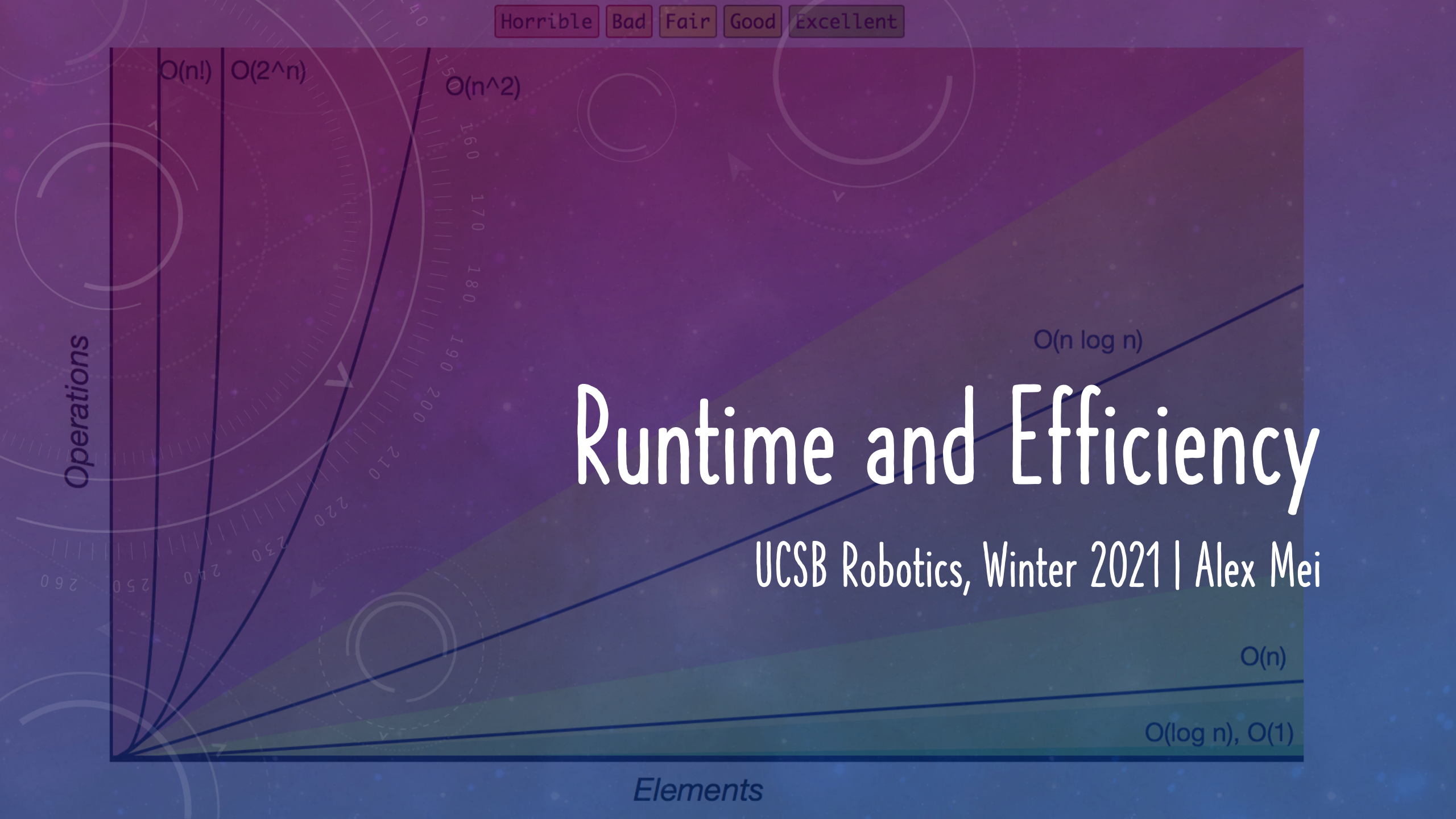


Runtime and Efficiency

UCSB Robotics, Winter 2021 | Alex Mei

ANNOUNCEMENTS

- CodePath iOS Development Course - Spring, 2021: https://discord.gg/BzqjEPv38x
  - Info Session February 20th @ 7 PM: https://tinyurl.com/dqrfabbf
- Robotics T-Shirt Design Competition Deadline Extended
- $15 per person prize for 1st place team at Project Showcase (February 27th 2 PM)

DAILY RUNDOWN

- Runtime Analysis
- Design Techniques
- Project Sprint 2 Work Time

ANALYSIS GOALS

- Determine growth of running time vs input size
- Focus on asymptotic behavior
- Ignore the noise produced by the function
- Estimate worst-case scenario

BIG O NOTATION

- Ignore Coefficients
- Focus on the Largest Term
- Focus on the number of computed steps
- Primitive operation = one step

EXAMPLE 1: ACCUMULATOR

- Initialize result, i (2 steps)
- Comparison i < N (n steps)
- i++ (2n steps)
- Accumulate result (3n steps)
- Return result (1 step)
- Overall: 3 + 6N steps = O(N)

```
/* N is the length of the array*/
int sumArray(int arr[], int N)
{
        int result=0;
        for(int i=0; i < N; i++)
                result+=arr[i];
        return result;
}
```

EXAMPLE 2

- O(log N)
- Intuition:
  - Log N recursive calls
  - Each call does constant operations

```
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}
```

TWO RULES

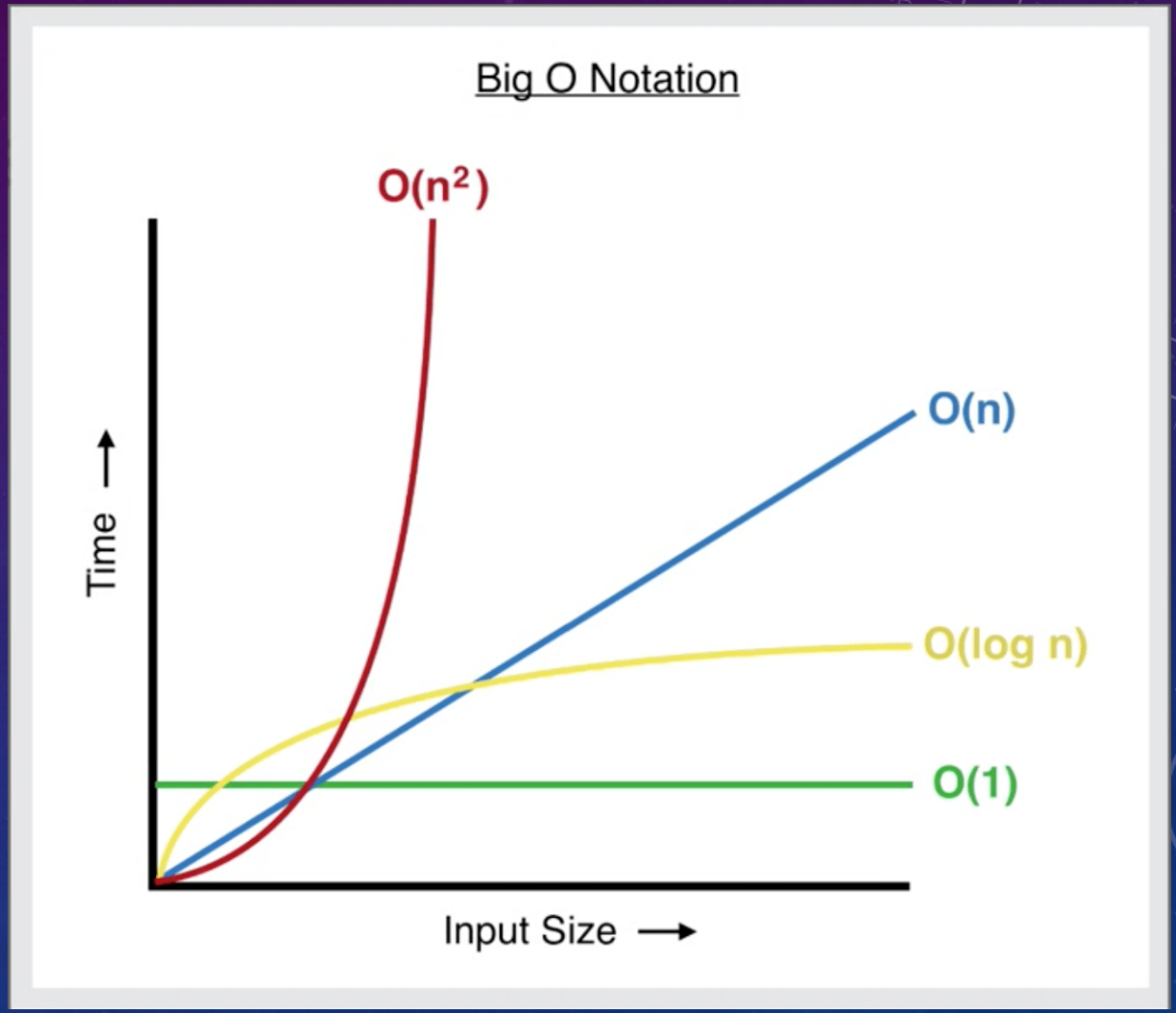
- Rule of Sum: O(a) + O(b) = O(max(a, b))
  - Example: sumArray, O(3) + O(6N) = O(6N) = O(N)
- Rule of Product: O(a) * O(b) = O(a * b)
  - Example: iterate through square array, O(N) outer * O(N) inner = O(N^2)

```
for (int i = 0; i<array.length; i++){
     for (int j = 0; j<array[i].length; j++){
        string += array[i][j];
     } System.out.println(string)

}
```

COMMON RUNTIMES

- Constant: O(1)
- Logarithmic: O(Log N)
- Linear: O(N)
- Linearithmic: O(N * log N)
- Power: O(N^constant)
- Exponential: O(constant^N)



MAXIMUM SUBSEQUENCE PROBLEM

- Given a sequence of integers, find the maximum possible value of a subsequence.
- Negative numbers can be included.
- The subsequence must be continuous.
- Example: 5, 10, 15, -20, 25, 30, 34, -1, 23, 18, 5, -3, 19, 20, 21, -3

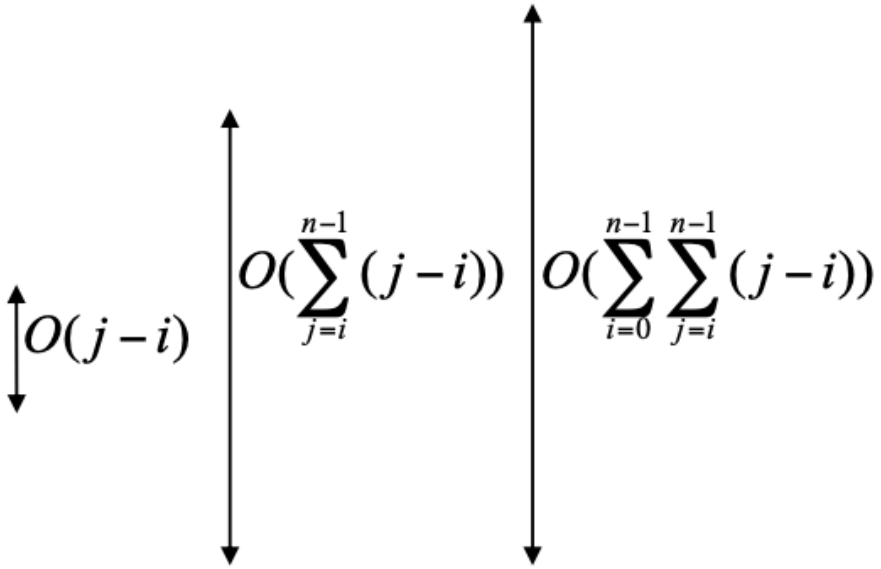
TRY IT YOURSELF!

- Come up with an algorithm that solves the Maximum Subsequence Problem
- Four solutions: O(N^3), O(N^2), O(N log N), O(N)

BRUTE FORCE SOLUTION: O(N^3)

```
int maxSum = 0;                                          O(1)

for( int i = 0; i < a.size( ); i++ )
for( int j = i; j < a.size( ); j++ )
{                                                        O(1)
        int thisSum = 0;
        for( int k = i; k <= j; k++ )                    O(1)
                thisSum += a[ k ];
        if( thisSum > maxSum )                           O(1)
                maxSum   = thisSum;
}
return maxSum;
```



BRUTE FORCE SOLUTION OPTIMIZED: O(N^2)

```
into maxSum = 0;

for( int i = 0; i < a.size( ); i++ )
        int thisSum = 0;
        for( int j = i; j < a.size( ); j++ )
        {
            thisSum += a[ j ];
            if( thisSum > maxSum )
                maxSum   = thisSum;
        }
return maxSum;
```

DIVIDE AND CONQUER SOLUTION: O(N LOG N)

```
// Find the maximum possible sum in arr[] auch that arr[m] is part of it
int maxCrossingSum(int arr[], int l, int m, int h)
{
    // Include elements on left of mid.
    int sum = 0;
    int left_sum = INT_MIN;
    for (int i = m; i >= l; i--)
    {
        sum = sum + arr[i];
        if (sum > left_sum)
          left_sum = sum;
    }

    // Include elements on right of mid
    sum = 0;
    int right_sum = INT_MIN;
    for (int i = m+1; i <= h; i++)
    {
        sum = sum + arr[i];
        if (sum > right_sum)
          right_sum = sum;
    }

    // Return sum of elements on left and right of mid
    // returning only left_sum + right_sum will fail for [-2, 1]
    return max(left_sum + right_sum, left_sum, right_sum);
}
```

```
// Returns sum of maxium sum subarray in aa[l..h]
int maxSubArraySum(int arr[], int l, int h)
{
   // Base Case: Only one element
   if (l == h)
     return arr[l];

   // Find middle point
   int m = (l + h)/2;

   /* Return maximum of following three possible cases
      a) Maximum subarray sum in left half
      b) Maximum subarray sum in right half
      c) Maximum subarray sum such that the subarray crosses the midpoint */
   return max(maxSubArraySum(arr, l, m),
              maxSubArraySum(arr, m+1, h),
              maxCrossingSum(arr, l, m, h));
}
```

CLEVER SOLUTION: O(N)

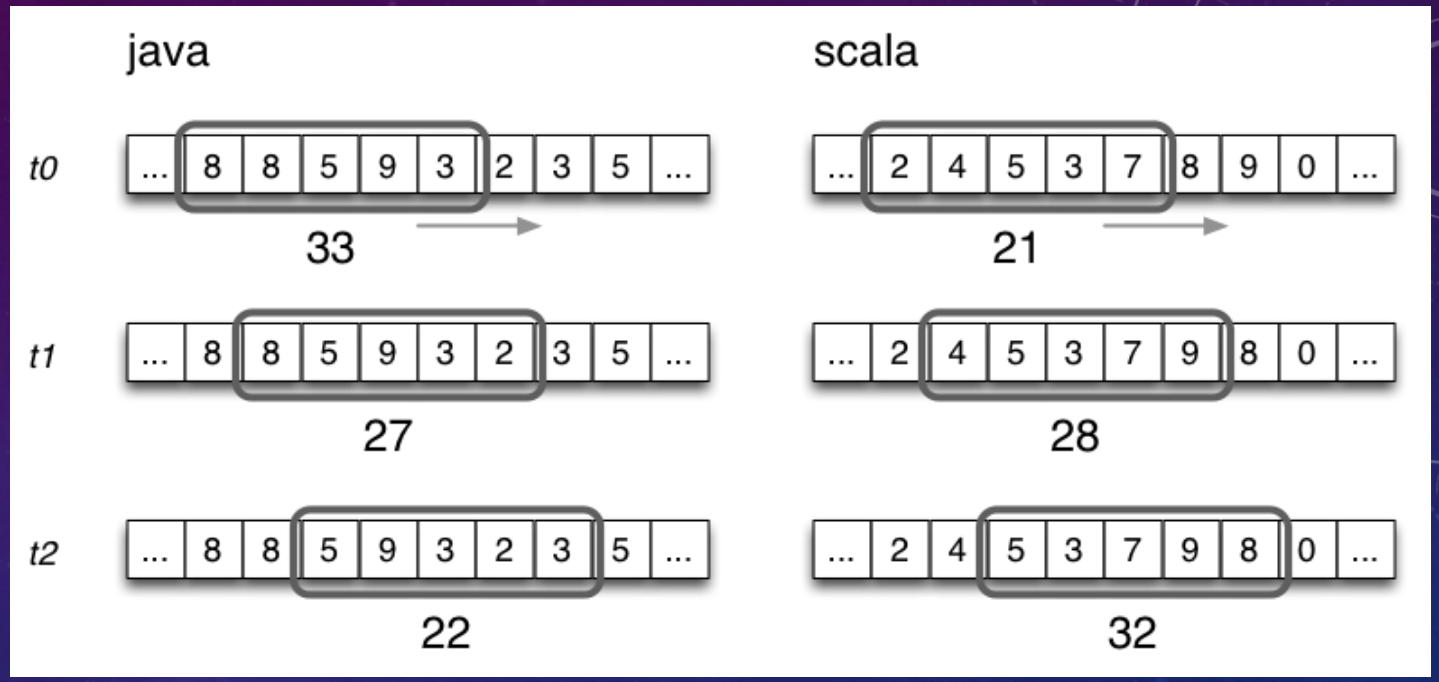
```
    int maxSum = 0, thisSum = 0;

    for( int j = 0; j < a.size( ); j++ )
    {
        thisSum += a[ j ];

        if ( thisSum > maxSum )
            maxSum = thisSum;
        else if ( thisSum < 0 )
            thisSum = 0;
    }
    return maxSum;
}
```

LAST TIPS...

- Accumulator Pattern
- Sliding Window Method
- Uniqueness: Sets and Dictionaries
- Dictionaries for Constant Access



PRACTICE PROBLEM:

You are given a number n and an array of integers A, where each integer can be anywhere from 1 to n (inclusive). Presume that in A, every integer from 1 to n appears at least once. Describe an algorithm that finds the minimum length of the contiguous subsequence that contains every integers from 1 to n at least once. Can your algorithm run in O(m) time (where m is the size of array A)?

PROJECT SPRINT 2 WEEK 2

- Perform Daily Standup in groups (ideally, document this)
- Project Time! :)