UNIVERSITY OF TORONTO

Refresher on

Algorithms

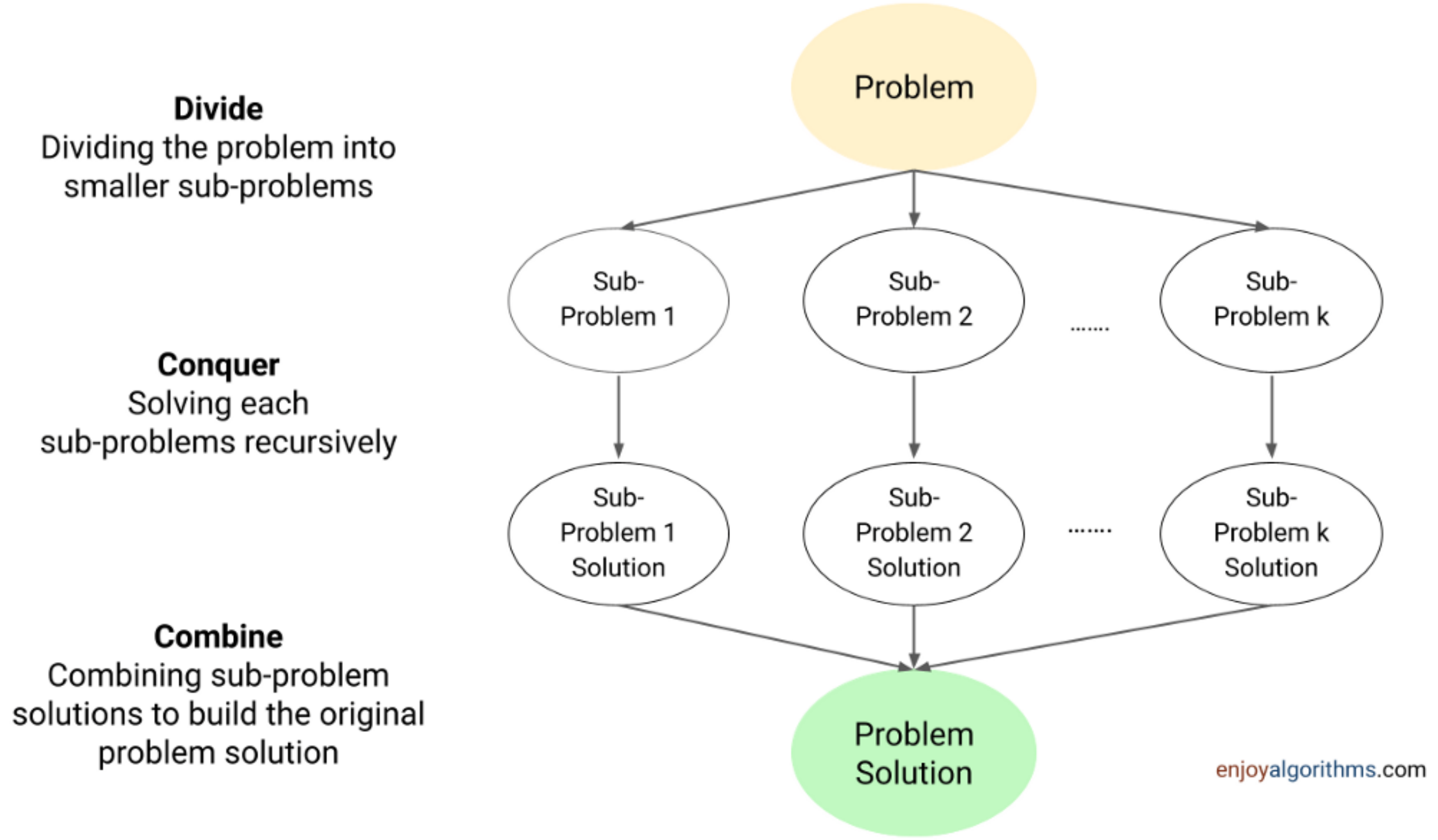
Keerat Kaur Guliani

ML Operations, TUSK

Topics to cover today…

+ Divide and Conquer
+ Dynamic Programming
+ Greedy Approach

Divide and Conquer



Divide and Conquer - Types

+ DAC using one subproblem – Decrease and Conquer

  Eg: Binary Search

+ DAC using two subproblems

  Eg: Merge Sort

Illustration: Maximum Subarray Sum

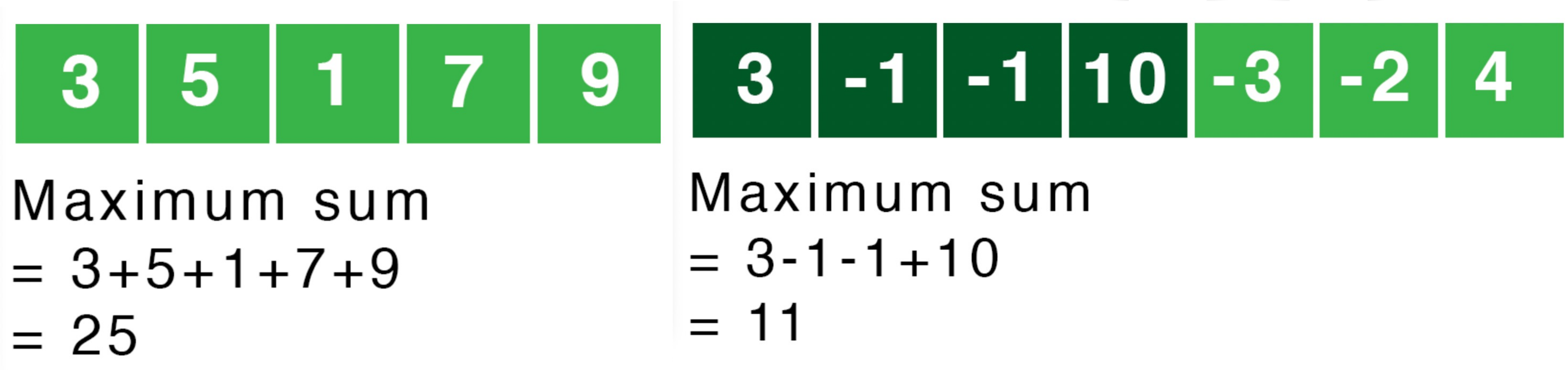
+ Given an integer array, find the subarray with the largest sum/return its sum.

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
```

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

+ Not the same as maximum subsequence sum!

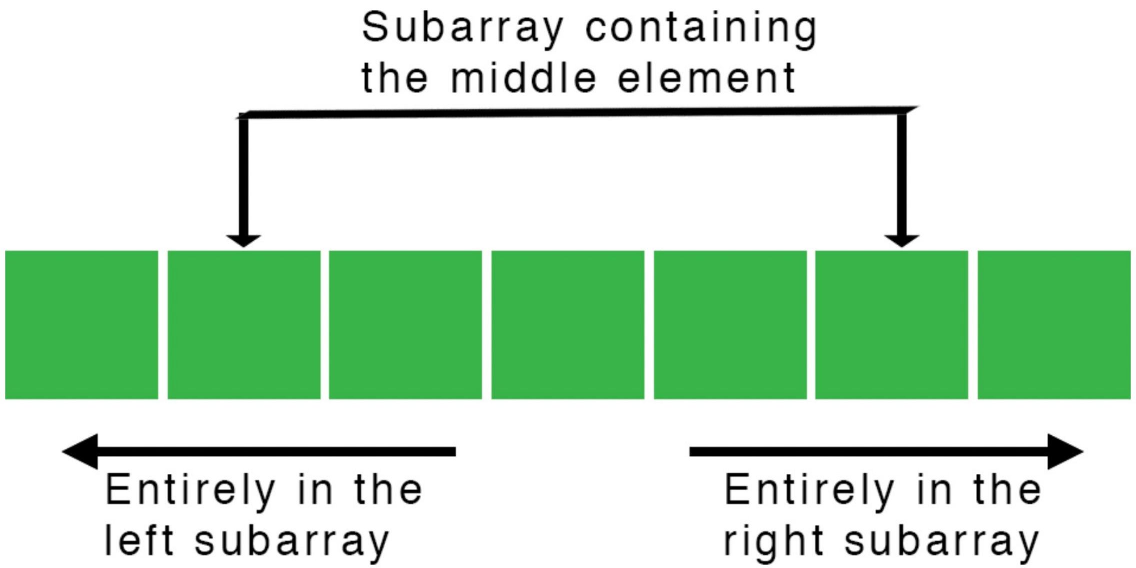
Maximum Subarray

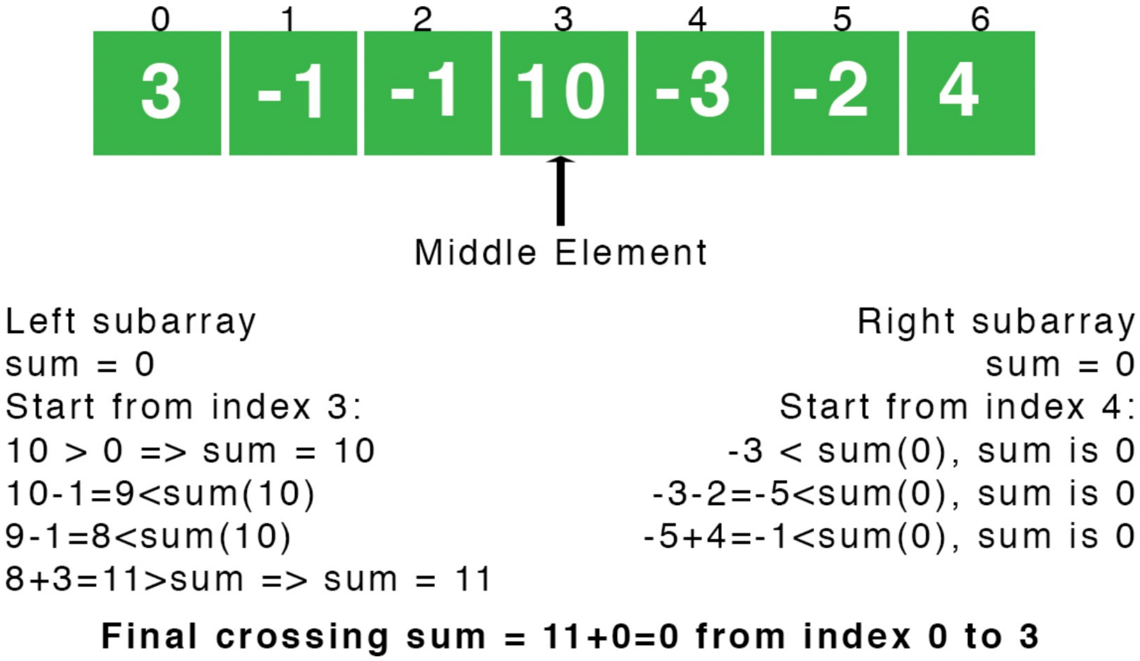


Naïve Method: Consider every possible subarray using 2 loops and return the overall maximum.

Time complexity: O(n^2)

Maximum Subarray with DAC





Code

$T(n) = 2T(n/2) + \Theta(n)$

Time: $O(n\log n)$

Space: $O(1)$

Dynamic Programming

What is DP?

+ Useful technique for optimization problems
+ Looks through all possible sub-problems and never recomputes the solution to any sub-problem
+ Guarantees correctness and efficiency

When can DP be applied?

+ Breaking down an optimization problem into simpler sub-problems (*optimal substructure*)
+ Storing the solution to each sub-problem so that each sub-problem is solved only once (*memoization*)

What exactly is a sub-problem?

+ Smaller version of the original problem
+ Sub-problems build on each other to obtain the solution to the original problem

Illustration 1: Longest Common Subsequence (LCS)

+ Given two strings s1 and s2 return the length of the longest common subsequence of characters between the two strings (need not be contiguous).

```
Input:
s1 = "ADC"
s2 = "ABCD"

Output: 2
Explanation:
"ADC"
"ABCD"
Both strings share the subsequence "A", "D".
```

```
Input:
s1 = "DBC"
s2 = "CBD"

Output: 1
Explanation:
"DBC"
"CBD"
          or
"DBC"
"CBD"
          or
"DBC"
"CBD"
```

```
Input:
s1 = "ABCD"
s2 = "ABCD"

Output: 4
Explanation:
"ABCD"
"ABCD"
Both strings share the subsequence "A", "B", "C", "D".
```

For example,
lcs('azb', 'aab') = 1 + lcs('az', 'aa') = 1 + max(lcs('az', 'a'), lcs('a', 'aa'))
= 1 + max(max(lcs('az', ''), lcs('a', 'a')) , 1 + lcs(' ', 'a'))
= 1 + max(1,1) = 2

Subproblem decomposition using the DP Table

Find the LCS between AGGTAB and GXTXAYB.

	""	A	G	G2	T	A3	B
" "	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Code

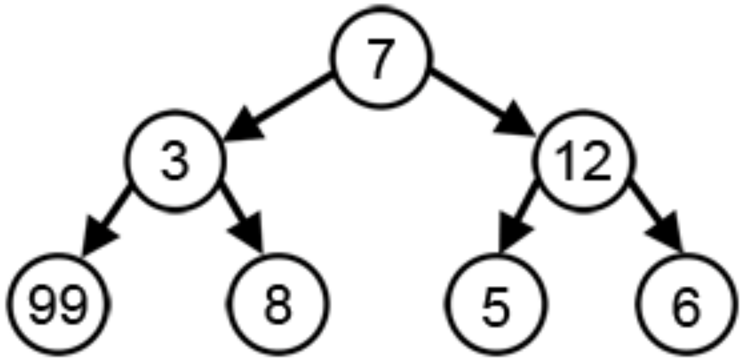
Time: O(mn)

Space: O(mn)

Where m = len(s1), n = len(s2)

Greedy Algorithm

+ An optimization algorithm that prioritizes the optimal choice at each step as it attempts to find the overall optimum to solve the entire problem.
+ An example where it works: Shortest path through a graph
+ An example where it does not work: To reach the largest sum in the graph



Greedy Algorithm

+ *When can a greedy algorithm be used?*

  Greedy choice property: A global (overall) optimal solution can be reached by choosing the optimal choice at each step.

  Optimal substructure: A problem has an optimal substructure if an optimal solution to the entire problem contains the optimal solutions to the sub-problems.

+ *Limitations of Greedy Algorithms:*
  - May not be able to find the global optimum because only limited data is considered.
  - Decision at each time step is made based on choices made so far, which may not include all possible future choices.

Illustration: Activity Selection Problem

+ Given N activities with their start time and end time. The task is to find the solution set having a maximum number of non-conflicting activities that can be executed within the given time, assuming only a single activity can be performed at a given time.

Input: start[] = [10, 12, 20}]
end[] = [20, 25, 30]
Output: [0, 2]
Explanation: A maximum of two activities can be performed, i.e. Activity 0 and Activity 2[0-based indexing].

Input: start[] = [1, 3, 0, 5, 8, 5]
finish[] = [2, 4, 6, 7, 9, 9]
Output: [0, 1, 3, 4]
Explanation: A maximum of four activities can be performed, i.e. Activity 0, Activity 1, Activity 3, and Activity 4[0-based indexing].

The Greedy Choice

Choose activities with the earliest finish time. The earlier you finish, the more time you have for other activities!

- Steps:
  - Sort the activities according to their finishing time
  - Select the first activity from the sorted array and print it
  - Do the following for the remaining activities in the sorted array
    - If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it

Code

Time: O(nlogn)

Space: O(1)