Dynamic Programming

Maximum Subarray Sum Problem

Dynamic Programming

- Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- Dynamic programming applies when the subproblems overlap—that is, when subproblems share subsubproblems. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems. A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

Dynamic Programming

- We typically apply dynamic programming to ***optimization problems***. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.

Dynamic Programming

- When developing a dynamic-programming algorithm, we follow a sequence of four steps:
    1. Characterize the structure of an optimal solution.
    2. Recursively define the value of an optimal solution.
    3. Compute the value of an optimal solution, typically in a bottom-up fashion.
    4. Construct an optimal solution from computed information.

Dynamic Programming

- We examine the two key ingredients that an optimization problem must have in order for dynamic programming to apply:

  1. optimal substructure and
  2. overlapping subproblems

Optimal substructure

- The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. A problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems.
- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems. Consequently, we must take care to ensure that the range of subproblems we consider includes those used in an optimal solution.
- Dynamic programming often uses optimal substructure in a bottom-up fashion. That is, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem.

Overlapping subproblems

- The second ingredient that an optimization problem must have for dynamic programming to apply is that the space of subproblems must be “small” in the sense that a recursive algorithm for the problem solves the same subproblems over and over, rather than always generating new subproblems. Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems***.

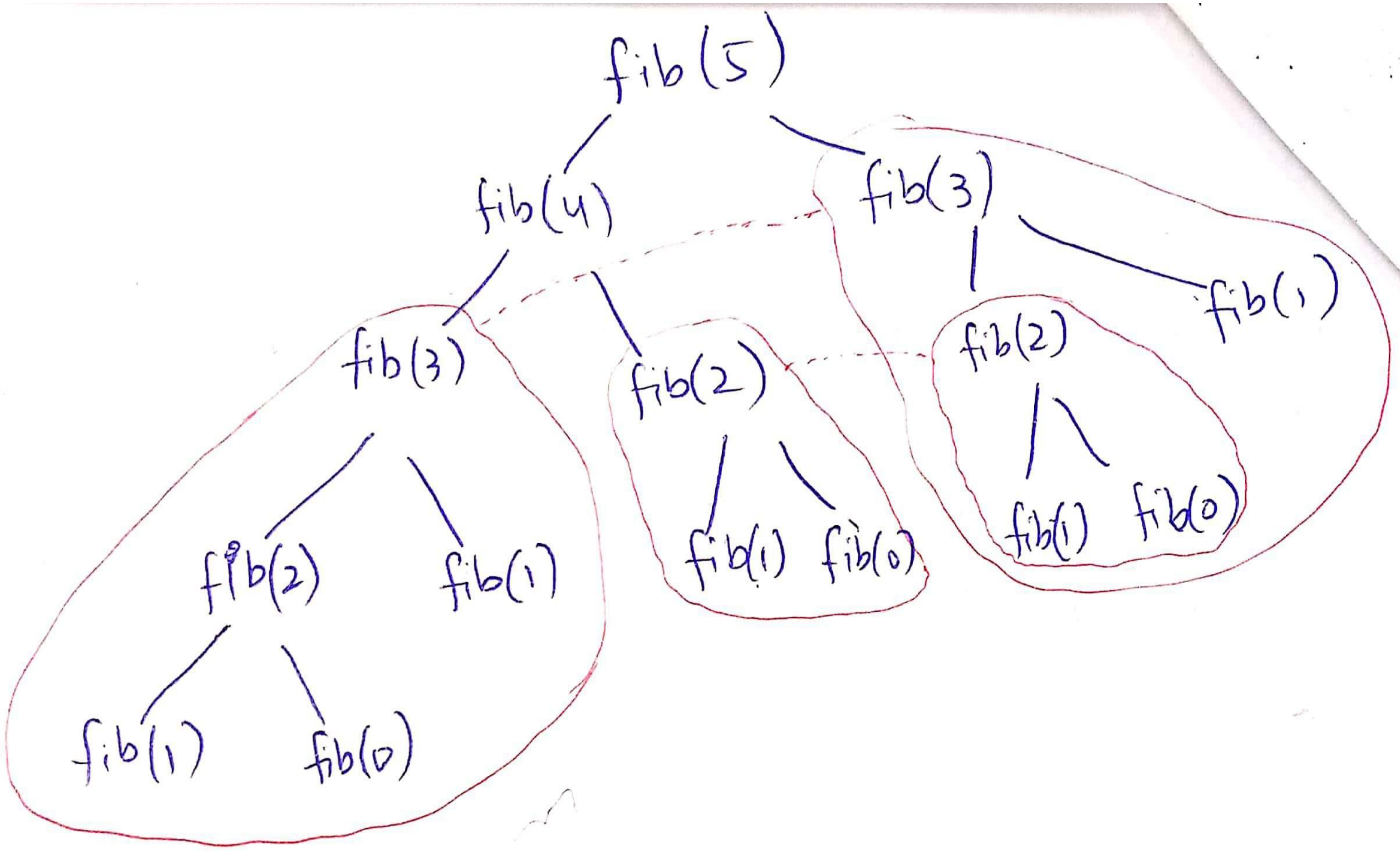
- Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup

Fibonacci Numbers

- f(n) = 0, 1, 1, 2, 3, 5, 8, …
- f(n) = f(n-1) + f(n-2) if n>=2
- f(1) = 1, f(0) = 0 BASE CASES

Solution by Recursion

```
int fib(int n)
{
if (n<=1)
    return n;
rlse
    return fib(n-1) + fib(n-2)
}
```


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)
fib(1)
fib(0)

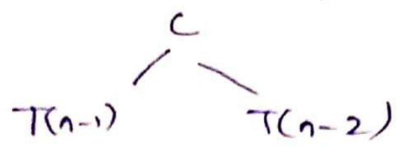
Recurrence Relation:

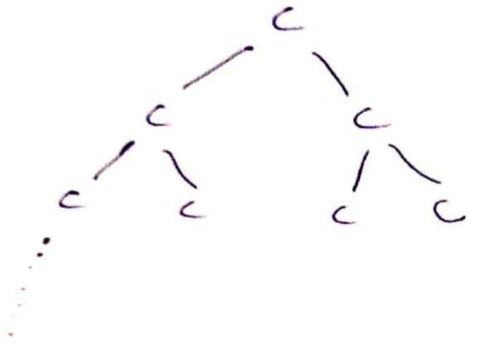
$$T(n) = T(n-1) + T(n-2) + c$$

$T(1) = c$

$T(0) = c.$

$T(n)$





c

2c

4c

8c.

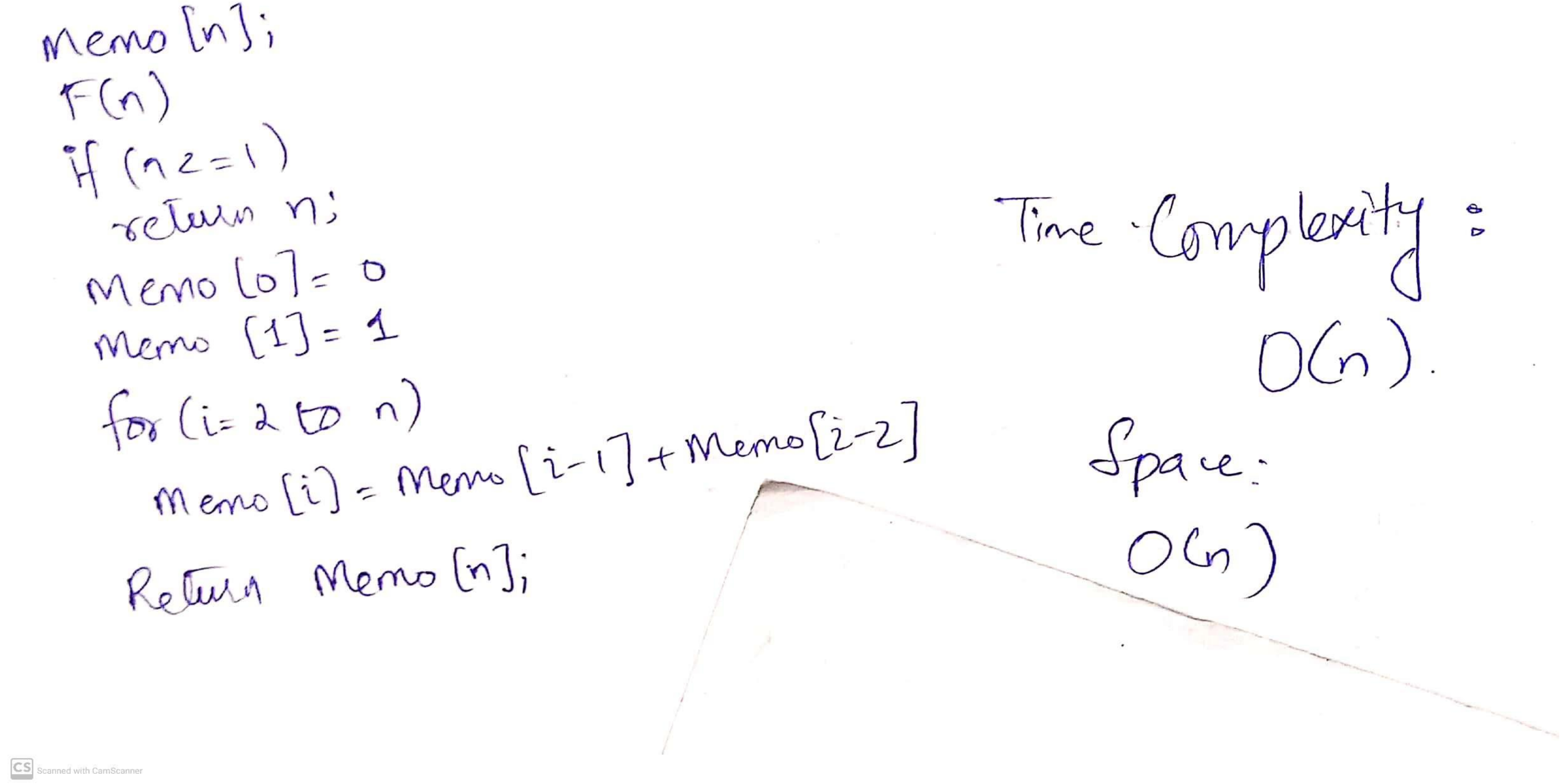
n levels

$$c + 2c + 4c + 8c + \dots + nc$$
$$c(1 + 2 + 4 + \dots + n)$$ ~~[illegible]~~

Geometric series of n numbers.

$$\frac{1(2^n - 1)}{2 - 1} = 2^n - 1.$$
$$= O(2^n).$$

Solution Using Dynamic Programming



```
memo [n];
F(n)
if (n <= 1)
  return n;
Memo [0] = 0
Memo [1] = 1
for (i = 2 to n)
  Memo [i] = Memo [i-1] + Memo [i-2]
Return Memo [n];
```

Time Complexity:

$O(n)$.

Space:

$O(n)$

Maximum Subarray Problem

- The maximum subarray problem is the task of finding the contiguous subarray within a one-dimensional array of numbers which has the largest sum.

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4

Brute Force Approach

- Find all possible contiguous subarrays.
- Calculate their sums.
- Return the subarray which has maximum sum value.

Brute Force Approach

- Find all possible contiguous subarrays.
- Calculate their sums.
- Return the subarray which has maximum sum value.

- The maximum subarray can start and end at any index from 1 to n.
- You can fix the starting point and find all sub-arrays starting at that index.
- Or you can fix the ending point and find all sub-arrays ending at that index.

Brute Force Algorithm - $O(n^3)$

- Given an array ‘a’ of size ‘n’, find all sub-arrays starting at a particular index.

```
max = -infinity
for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      {
      int sum = 0;
      for (int k = i; k <= j; k++)
         sum += a[k];
      if (sum > max)
         max = sum;
      }
```

Brute Force Algorithm - $O(n^2)$

- Given an array ‘a’ of size ‘n’, find all sub-arrays starting at a particular index.

```
max = -infinity
for (int i = 0; i < n; i++)
    int sum = 0;
    for (int j = i; j < n; j++)
      {
       sum += a[j];
       if (sum > max)
         max = sum;
      }
```

Brute Force Approach

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4

- Lets find all possible sub-arrays by fixing the end point.
- Then find all sub-arrays ending at that index.
- End index could have values from 1 to n.

Brute Force Approach

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4

All subarrays ending at index 1

- Sub-arrays ending at index 1:
  There's only one such sub-array

Brute Force Approach

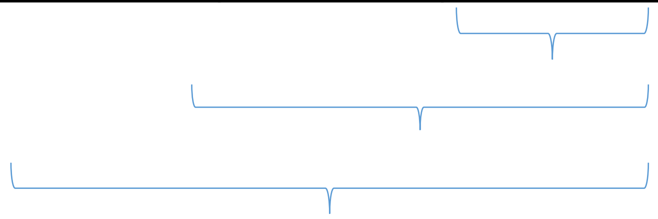
i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4

All subarrays ending at index 2

- Sub-arrays ending at index 2:
  There are two such sub-arrays

Brute Force Approach

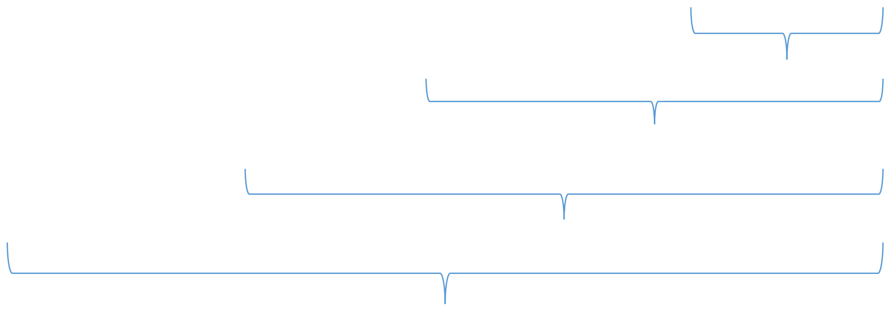
i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4


All subarrays ending at index 3

- Sub-arrays ending at index 3:
  There are three such sub-arrays

Brute Force Approach

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4



All subarrays ending at index 4

- Sub-arrays ending at index 4:
  There are four such sub-arrays

Brute Force Algorithm – O(n^2)

Given an array ‘a’ of size ‘n’, find all sub-arrays ending at a particular index.

```
max = -infinity
for (end = 1 to n)
    int sum = 0;
    for (start = end to 1)
      {
       sum += a[start];
       if (sum > max)
         max = sum;
      }
```

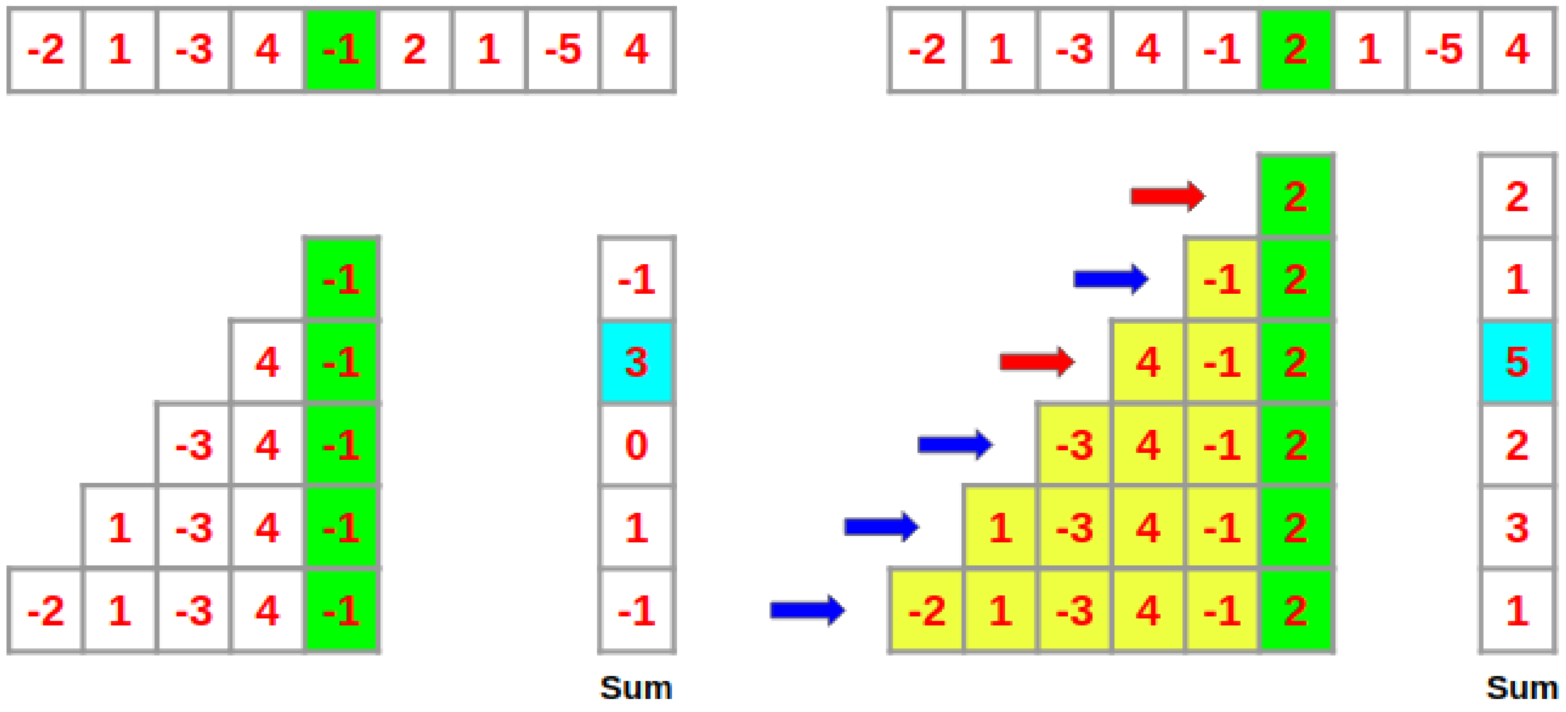
Brute Force Algorithm – O(n^2)

Given an array ‘a’ of size ‘n’, find all sub-arrays ending at a particular index.

```
max = -infinity
for (end = 1 to n)
    int sum = 0;
    for (start = end to 1)
      {
       sum += a[start];
       if (sum > max)
         max = sum;
         start_index = start
         end_index = end
      }
```

Dry Run of Brute Force Approach

- Following figure shows all sub-arrays ending at indices 5 & 6.



i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4

$\underbrace{\qquad\qquad}$

All subarrays ending at index 1

LocalMax for subarrays ending at index 1

LocalMax[1] = A[1]

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4

All subarrays ending at index 1

LocalMax for subarrays ending at index 1

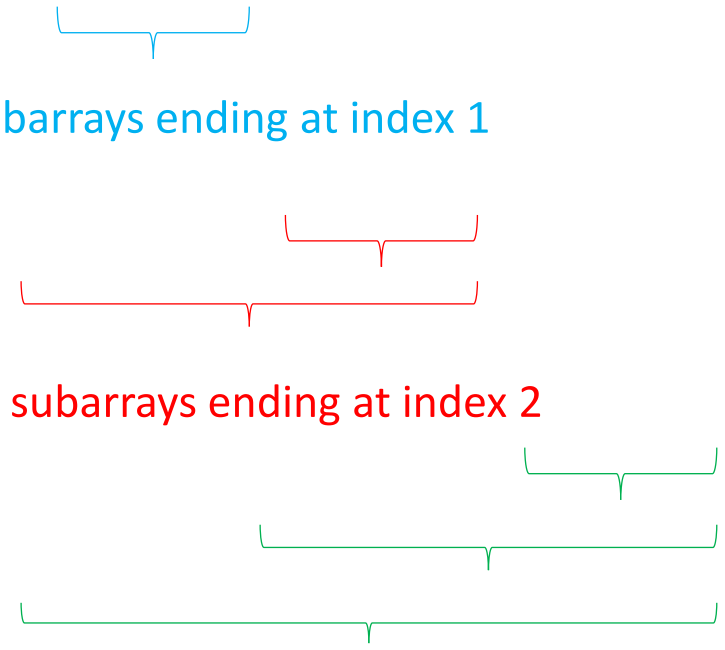
LocalMax[1] = A[1]

All subarrays ending at index 2

LocalMax for subarrays ending at index 2

$$\text{LocalMax}[2] = \max \begin{cases} A[2] \\ A[1]+A[2] \end{cases}$$

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4



All subarrays ending at index 1

All subarrays ending at index 2

All subarrays ending at index 3

LocalMax for subarrays ending at index 1

LocalMax[1] = A[1]

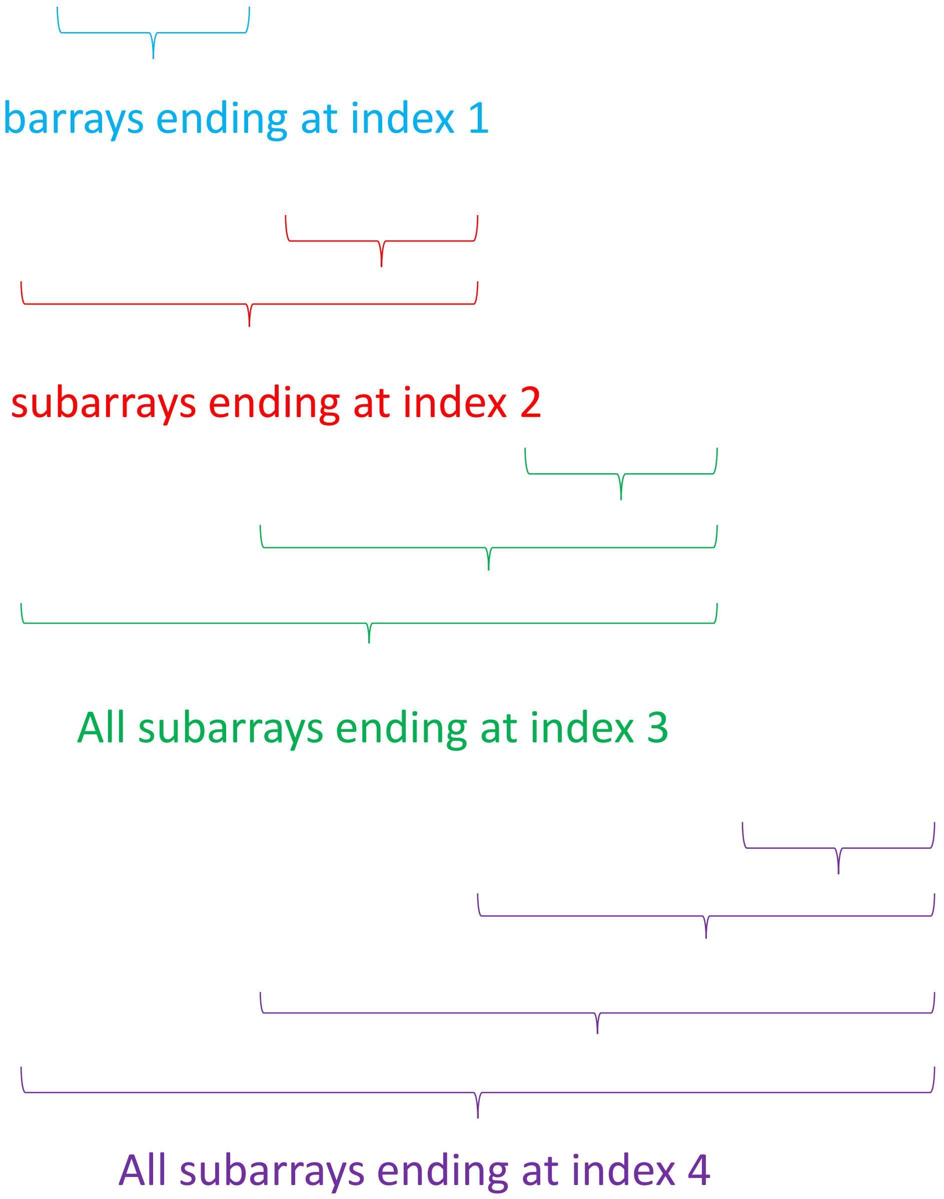
LocalMax for subarrays ending at index 2

$$\text{LocalMax[2]} = \max \begin{cases} A[2] \\ A[1]+A[2] \end{cases}$$

LocalMax for subarrays ending at index 3

$$\text{LocalMax[3]} = \max \begin{cases} A[3] \\ A[2]+A[3] \\ A[1]+A[2]+A[3] \end{cases}$$

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4


All subarrays ending at index 1
All subarrays ending at index 2
All subarrays ending at index 3
All subarrays ending at index 4

LocalMax for subarrays ending at index 1

LocalMax[1] = A[1]

LocalMax for subarrays ending at index 2

$$\text{LocalMax[2]} = \max \begin{cases} A[2] \\ A[1]+A[2] \end{cases}$$

LocalMax for subarrays ending at index 3

$$\text{LocalMax[3]} = \max \begin{cases} A[3] \\ A[2]+A[3] \\ A[1]+A[2]+A[3] \end{cases}$$

LocalMax for subarrays ending at index 4

$$\text{LocalMax[4]} = \max \begin{cases} A[4] \\ A[3]+A[4] \\ A[2]+ A[3]+A[4] \\ A[1]+ A[2]+ A[3]+A[4] \end{cases}$$

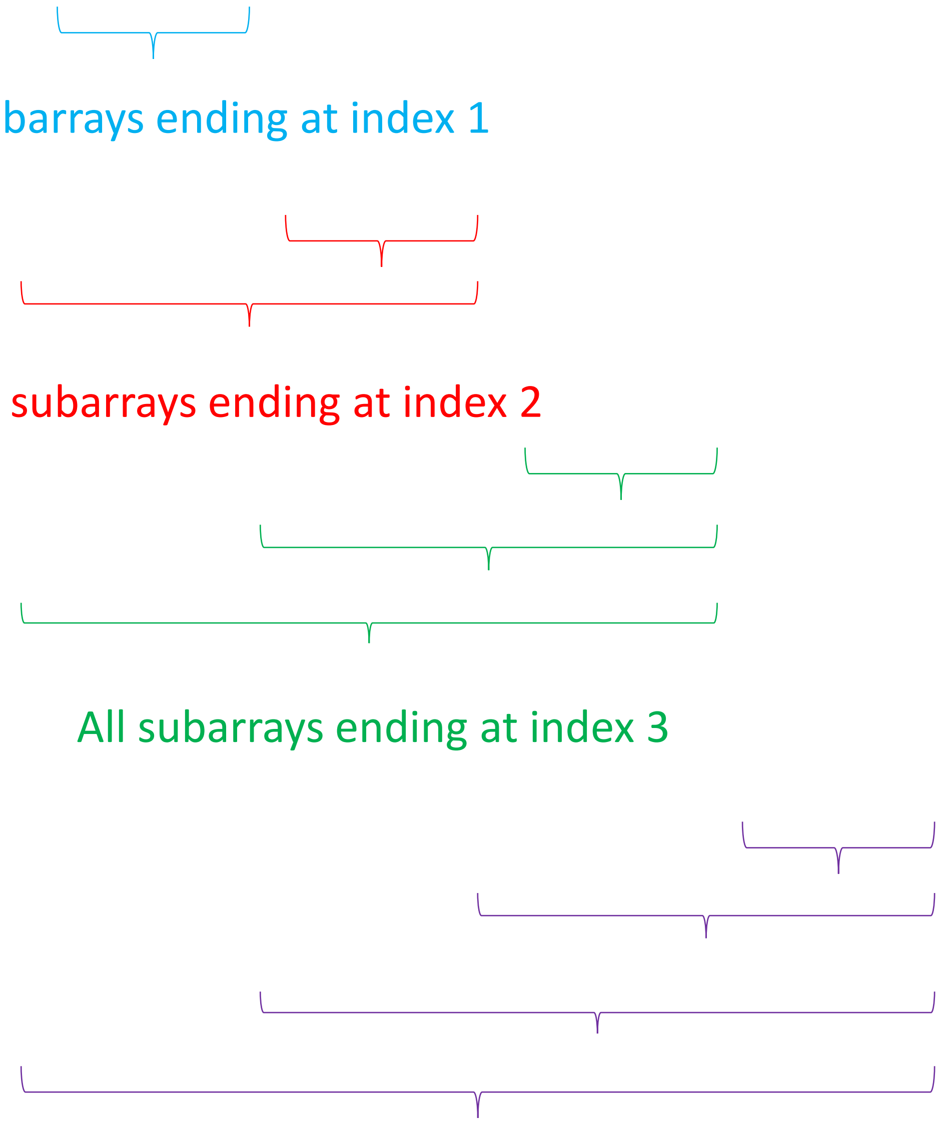
DP Solution (Kadane’s Algorithm)

- Identify overlapping Sub-problems.
- See if we can find optimal solution of the problem using optimal solutions of sub-problems.

DP Solution (Kadane’s Algorithm)

- Identify overlapping Sub-problems.
- See if we can find optimal solution of the problem using optimal solutions of sub-problems.
- Devise Recursive formula for DP solution.

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4



All subarrays ending at index 1

LocalMax for subarrays ending at index 1

LocalMax[1] = A[1]

All subarrays ending at index 2

LocalMax for subarrays ending at index 2

$$LocalMax[2] = max \left\{ \begin{array}{r} A[2] \\ A[1]+A[2] \end{array} \right.$$

All subarrays ending at index 3

LocalMax for subarrays ending at index 3

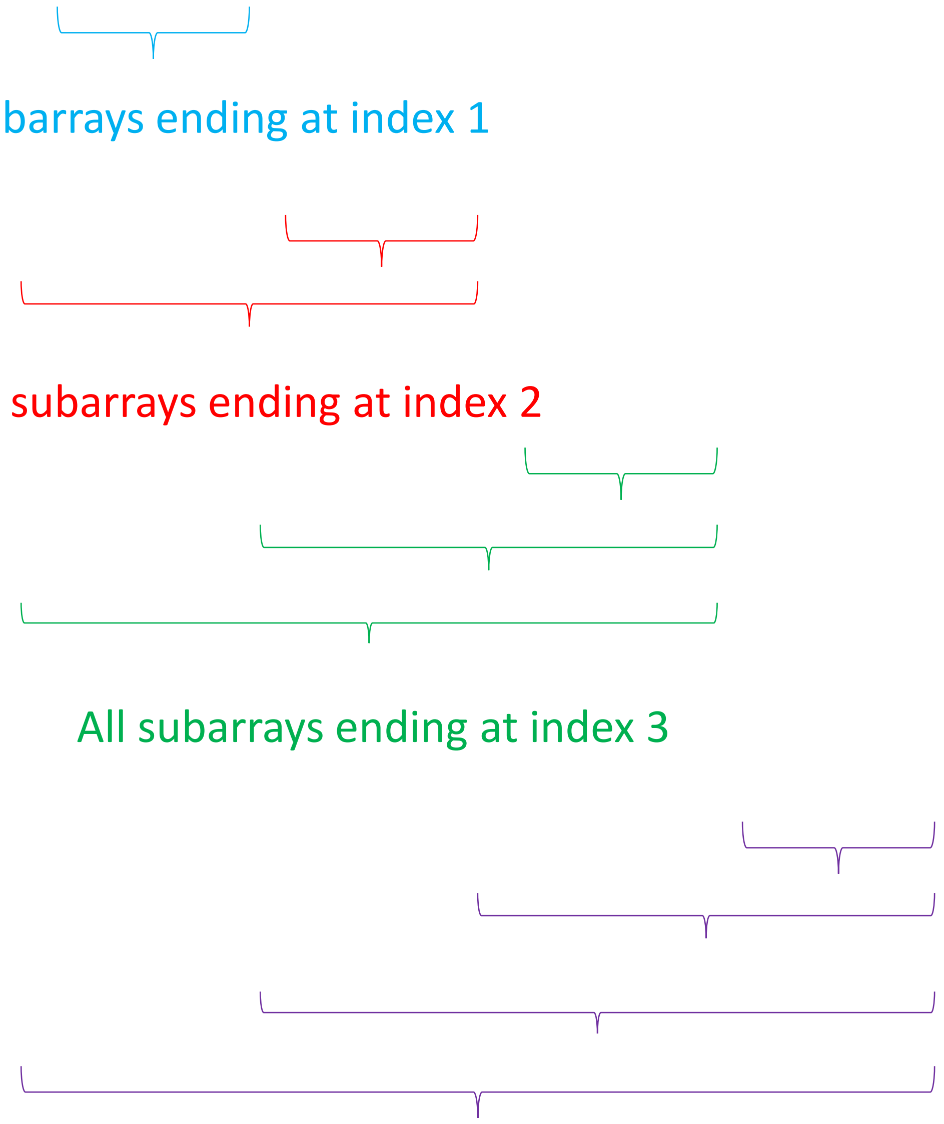
$$LocalMax[3] = max \left\{ \begin{array}{r} A[3] \\ A[2]+A[3] \\ A[1]+ A[2]+A[3] \end{array} \right.$$

All subarrays ending at index 4

LocalMax for subarrays ending at index 4

$$LocalMax[4] = max \left\{ \begin{array}{r} A[4] \\ A[3]+A[4] \\ A[2]+ A[3]+A[4] \\ A[1]+ A[2]+ A[3]+A[4] \end{array} \right.$$

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4


All subarrays ending at index 1

All subarrays ending at index 2

All subarrays ending at index 3

All subarrays ending at index 4

LocalMax for subarrays ending at index 1

LocalMax[1] = A[1]

LocalMax for subarrays ending at index 2

$$\text{LocalMax[2]} = \max \begin{cases} \text{A[2]} \\ \text{LocalMax[1]} + \text{A[2]} \end{cases}$$

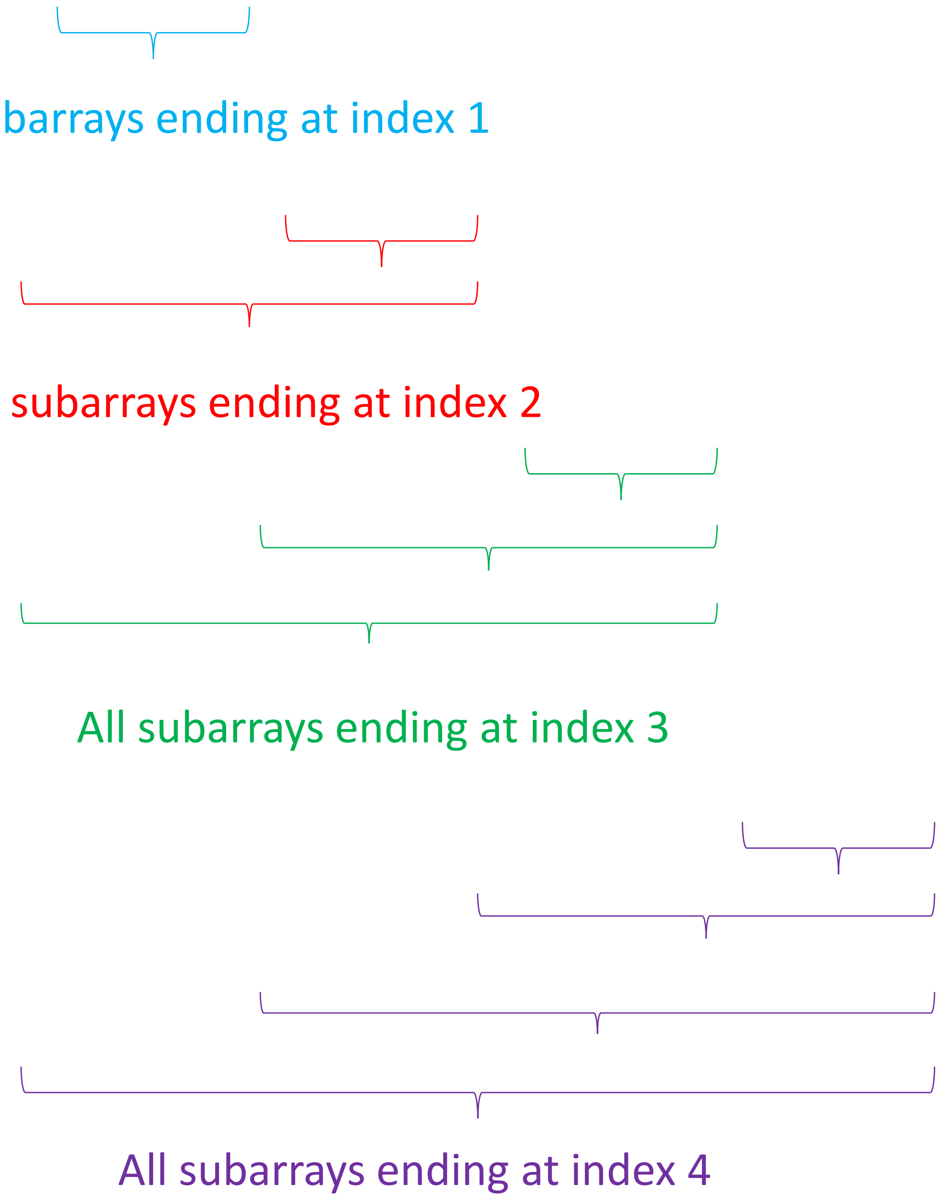
LocalMax for subarrays ending at index 3

$$\text{LocalMax[3]} = \max \begin{cases} \text{A[3]} \\ \text{LocalMax[2]} + \text{A[3]} \end{cases}$$

LocalMax for subarrays ending at index 4

$$\text{LocalMax[4]} = \max \begin{cases} \text{A[4]} \\ \text{LocalMax[3]} + \text{A[4]} \end{cases}$$

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4



All subarrays ending at index 1
All subarrays ending at index 2
All subarrays ending at index 3
All subarrays ending at index 4

LocalMax for subarrays ending at index 1

LocalMax[1] = A[1]

LocalMax for subarrays ending at index 2

$$\text{LocalMax[2]} = \max \begin{cases} A[2] \\ \text{LocalMax[1]} + A[2] \end{cases}$$

LocalMax for subarrays ending at index 3

$$\text{LocalMax[3]} = \max \begin{cases} A[3] \\ \text{LocalMax[2]} + A[3] \end{cases}$$

LocalMax for subarrays ending at index 4

$$\text{LocalMax[4]} = \max \begin{cases} A[4] \\ \text{LocalMax[3]} + A[4] \end{cases}$$

LocalMax[i] = Max (A[i] + LocalMax[i-1] , A[i])

DP Solution (Kadane’s Algorithm)

- Identify overlapping Sub-problems.
- See if we can find optimal solution of the problem using optimal solutions of sub-problems.
- Devise Recursive formula for DP solution.

LocalMax[i] = Max (A[i] + LocalMax[i-1] , A[i])

DP Solution

Given: An array ‘a’ of size ‘n.

```
LocalMax[1] = a[1]
for (i = 2 to n)
      {
      LocalMax[i] = Max (A[i] + LocalMax[i-1] , A[i])
      }
globalMax = max (LocalMax)
```

DP Solution

Given: An array ‘a’ of size ‘n.

```
LocalMax[1] = a[1]
for (i = 2 to n)
      {
      LocalMax[i] = Max (A[i] + LocalMax[i-1] , A[i])
      }
globalMax = max (LocalMax)
```

O(n)

DP Solution - Tasks

- The algorithm on previous slide gives the value of the optimal solution.
- Modify it to return the optimal solution i.e save and return start and end index of the sub-array which gives maximum sum.
- The algorithm on previous slide uses extra space to save localMax. In this algorithm we are looking at only one previous value so we can save it in a variable instead of saving localMax for all indices. Modify the code so that it doesn’t use extra space.

DP Solution - Tasks

- Dry-run the DP algo on following example and compute localMax of every index and then find globalMax.

i	1	2	3	4	5	6	7	8	9
A[i]	-2	1	-3	4	-1	2	1	-5	4
LocalMax[i]									

- GlobalMax = ?