Repetition:
Maximum Sub-array

Problem

- Given a sequence of positive and negative integers
  - Find the contiguous subarray with the maximum sum
  -

2	-5	3	7	-5	4	2	-4	3	-2	1

- Proposed in 1977 by Ulf Grenander
  - In a two-dimensional version for image recognition

Brute Force Solution

- Given $a[0,\ldots,n-1]$
  - Maximize
  - `{sum(a[i:j]) for i in range(n) for j in range(i+1,n)}`
  - Costs:
    - To calculate `sum(a[i:j])` need $j-i-1$ additions
    - $\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}(k-i-1) = \frac{1}{6}(n^3-n^2+2n)$

Preprocessing

- We can preprocess the array
  - Let `s[j] = a[0]+a[1]+a[2]+... a[j-1]`
  - `= sum(a[:j])`
  - Calculating *s* costs *n* additions
  - Then `sum(a[i:j]) = a[i]+...+a[j-1]`
  - `= sum(a[:j])-sum(a[:i])`
  - `= s[j]-s[i]`

Preprocessing

- Costs are now:
  - Creating `s`, which costs time *n*
  - Forming $1 + 2 + \ldots + (n - 1)$ = $n(n-1)/2$ elements

Divide and Conquer

- If the array is divided, what can happen to the maximum sum sub-array?
  - Three cases:
    - Maximum sub-array in the left half
    - Maximum sub-array in the right half
    - Maximum sub-array straggles the divider

Divide and Conquer

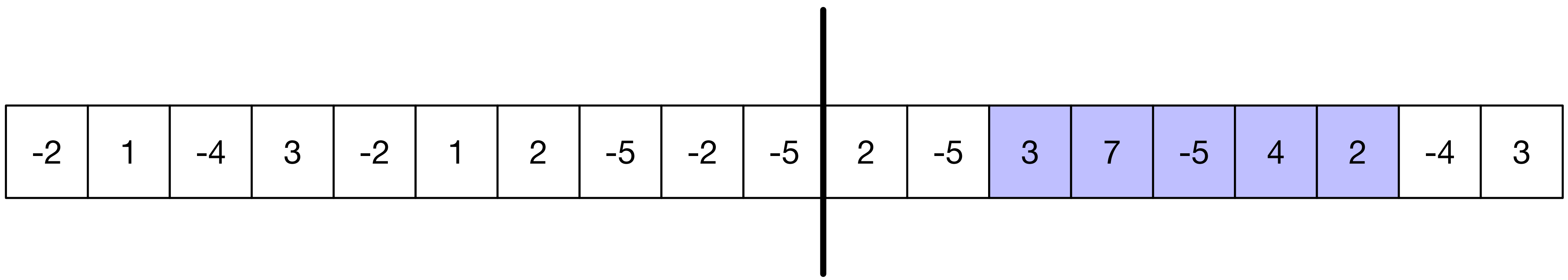
- Case 1:

2	-5	3	7	-5	4	2	-4	3	-2	1	-4	3	-2	1	2	-5	-2	-5

  - Compare the two maximum sum sub-arrays of each half and select the bigger one

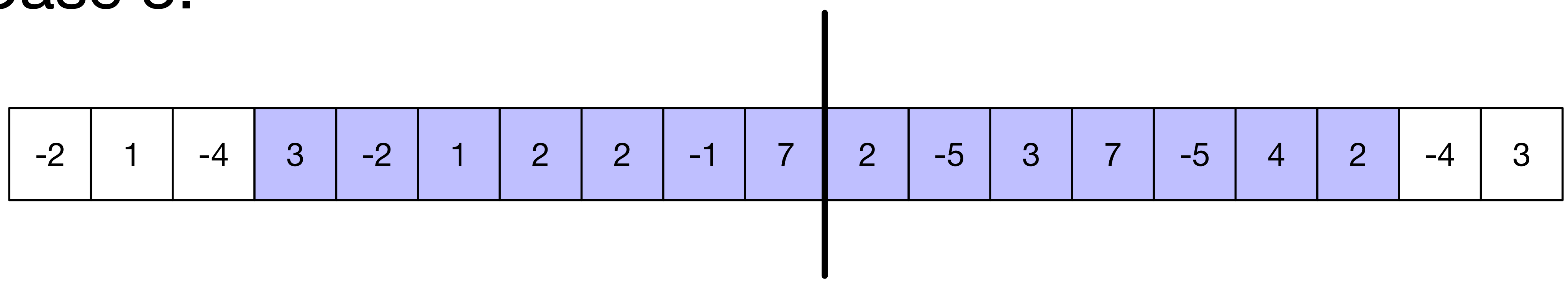
Divide and Conquer

- Case 2:



Divide and Conquer

- Case 3:



  - This means we need to **also** consider maximum sub-arrays that end at the left and that start at the right
  - These should be determined also in the algorithm

Divide and Conquer

- Ending on the right edge:
    - Case 1: Subarray part of the right half
    - Case 2: Subarray straddles division
        - In this case: Need to know the sum of the elements in the left half

Divide and Conquer

- Divide and conquer algorithm:
  - Divide the array into halves
  - Out of two halves:
    - Calculate four different values:
      - Total maximum sum sub-array
      - Total maximum sum sub-array starting on left
      - Total maximum sum sub-array ending at right
      - Total sum

Divide and Conquer

- For simplicity: just calculate the maxima and not the indices
  -

```
def max_sub_array(lista):
    #divide
    left = lista[:len(lista)//2]
    right = lista[len(lista)//2:]
    #calculate and then return four values

    return total, ttl_from_left, ttl_from_right, suma
```

Divide and Conquer

- For the calculation, we get the four values for the left and right half

```
def max_sub_array(lista):
    #divide
    left = lista[:len(lista)//2]
    right = lista[len(lista)//2:]
    #recursive step
    ltotal, lttl_from_left, lttl_from_right, lsuma = 
max_sub_array(left)
    rtotal, rttl_from_left, rttl_from_right, rsuma = 
max_sub_array(right)

    suma = lsuma+rsuma

    #Calculate the other three return values as well

return total, ttl_from_left, ttl_from_right, suma
```

Divide and Conquer

- Getting the sum is easy:
    - Just add up the sums of the left and right

Divide and Conquer

- How do we calculate the maximum sum sub-array from the information in the left and right halves:
  - Case 1:
    - The total maximum sub-array is the maximum of the total maximum sub-arrays of both sides

1	-5	2	-3	4	5	18	2	-2	1	-1	-1	1	-8	6	-2	1	3	5	2	1

Divide and Conquer

- Case 2:
  - The best choice is composed of the maximal one on the left ending at the end and the one on the right starting at the beginning

3	-5	2	1	-7	2	-4	4	5	2	2	1	1	1	1	1	-9	5	3	-1	1

Divide and Conquer

- How about starting on the left?
  - Case 1: Best case is the one starting on the left

4	5	2	2	-3	-3	2	1	-7	2	-4	1	1	1	1	1	-9	5	3	-1	1

  - Case 2: Best case is all of left plus the one subarray starting on the right

4	5	2	2	-3	-3	2	1	-1	2	-2	1	2	1	1	1	-9	5	3	-1	1

  - All of left gives you 9, violet part gives you 6, total is 15
    - This is why we also calculate the sum of each part

Divide and Conquer

- Similarly, maximum sum sub-array ending at the end could be:
  - Best sub-array ending at the end of the left sub-array plus all of the right half
  - Just the best sub-array ending at the end of the right half

Divide and Conquer

- Time analysis:
  - At each divide step:
    - We just make the recursive call
  - At each conquer step:
    - We calculate the four values (and the bounds of the corresponding sub-array)
  - Recurrence is $T(n) = 2T(n/2) + \Theta(1)$
  - MT: Compare $\Theta(1)$ with $n^{\log_2(2)} = n^1$
    - $T(n) = \Theta(n)$

Implementation

- In Python, you can use tuples and tuple extraction in order to pass several values

```
def maxsub(lista):
    if len(lista)==1:
        return max(0,lista[0]), max(0,lista[0]),  max(0,lista[0]), 
lista[0]
    else:
        left = lista[:len(lista)//2]
        right = lista[len(lista)//2:]

        ltot, lbeg, lend, lsum = maxsub(left)
        rtot, rbeg, rend, rsum = maxsub(right)


        return mytot, mybeg, myend, mysum
```

Divide and Conquer

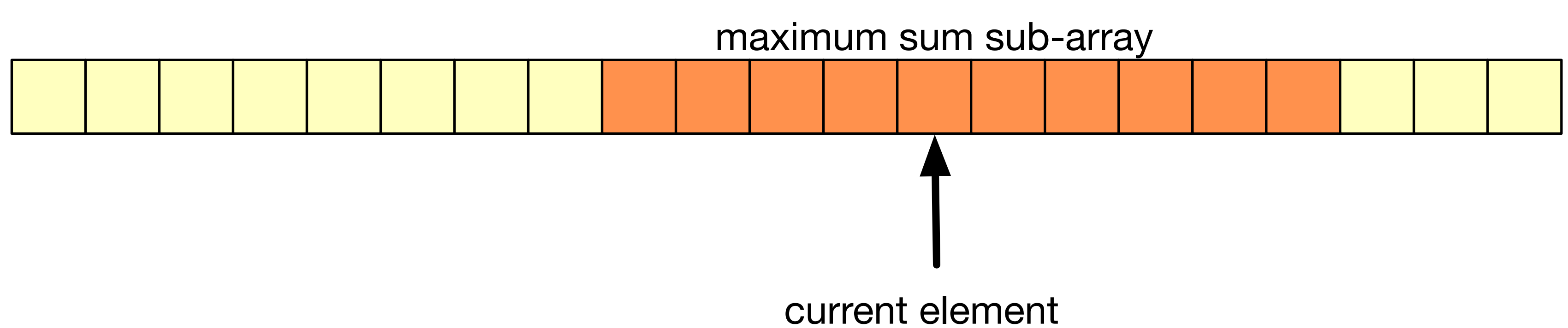
- What are the loop invariants?
  - We calculate four values.
    - Their correctness gives a conjunction of four elements

Dynamic Programming

- A dynamic programming approach:
    - What happens if the array has only one element?
        - In this case, the solution is :
            - Empty array if element is negative
            - The complete array if element is positive

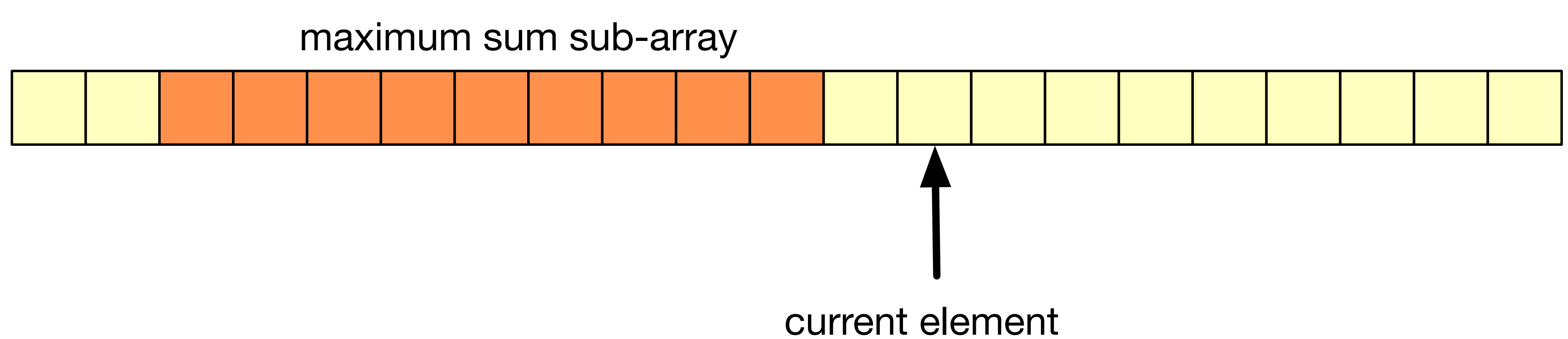
Dynamic Programming

- What happens if one element is added?
  - We need a loop invariant for this:
    - In order to design one, we need to see what might happen:
      - The new element is part of the maximum sum sub-array, but we cannot tell yet because we have not scanned everything



Dynamic Programming

- What happens if one element is added?
  - We need a loop invariant for this:
    - In order to design one, we need to see what might happen:
      - The new element is **not** part of the maximum sum sub-array, but we cannot tell yet because we have not scanned everything



Dynamic Programming

- Because we have to keep both cases in mind:
  - Consider two arrays:
    - A1: Best sum sub-array seen so far
    - A2: Best sum sub-array ending in the new element

Dynamic Programming

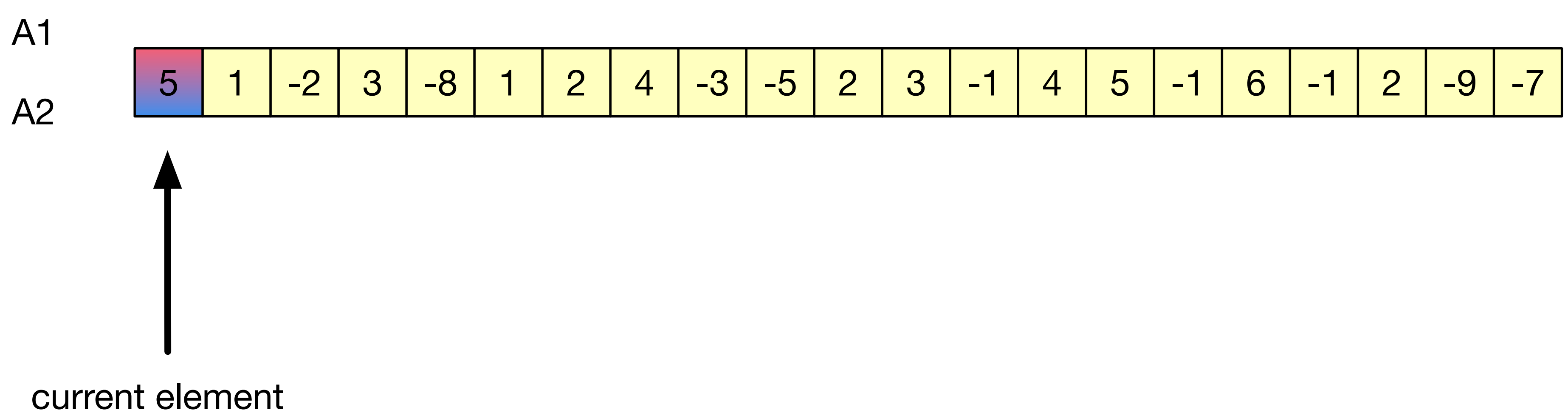
- To maintain A1:
  - A1 stays the same
  - By adding the new element, the current A2 plus the new element can become the new A1
    - This can only happen if the new element is positive (or zero)

Dynamic Programming

- To maintain A2:
  - If the new element is positive, we add it to A2
  - If the new element is negative, we try adding it to A2
    - Only if the new sum is negative, we A2 becomes empty

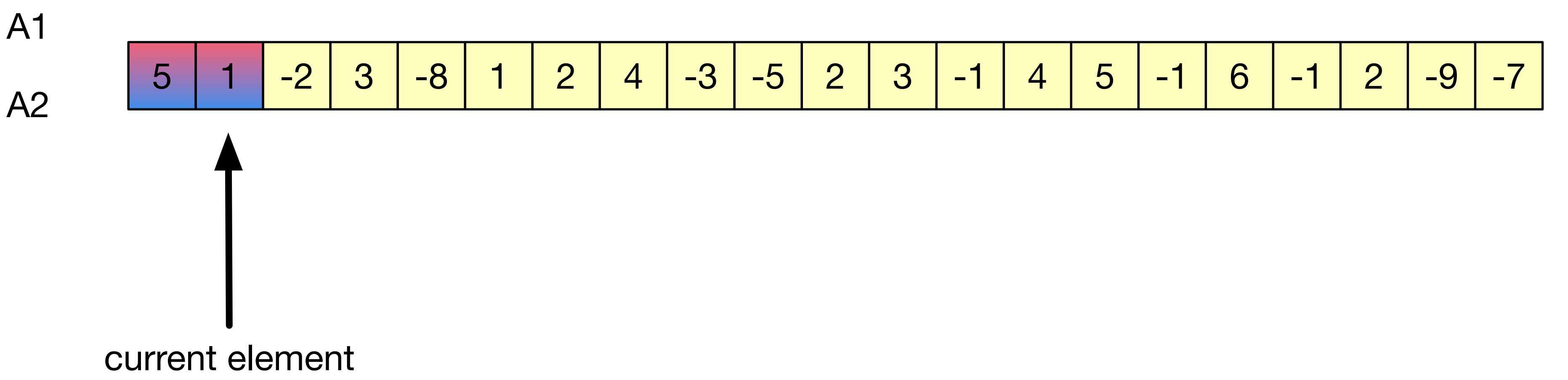
Dynamic Programming

- Example:
-



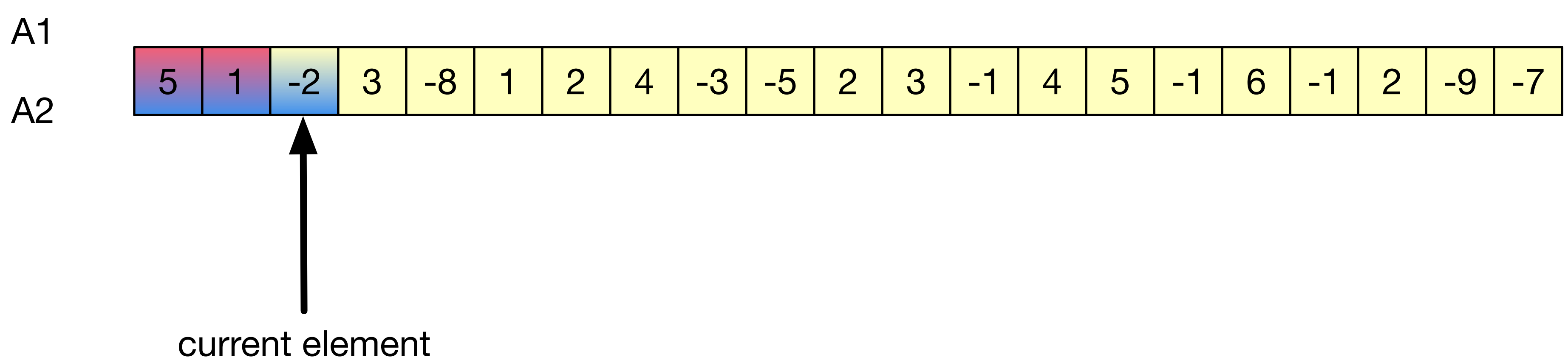
Dynamic Programming

- Example
  - sum(A1)=sum(A2)=6



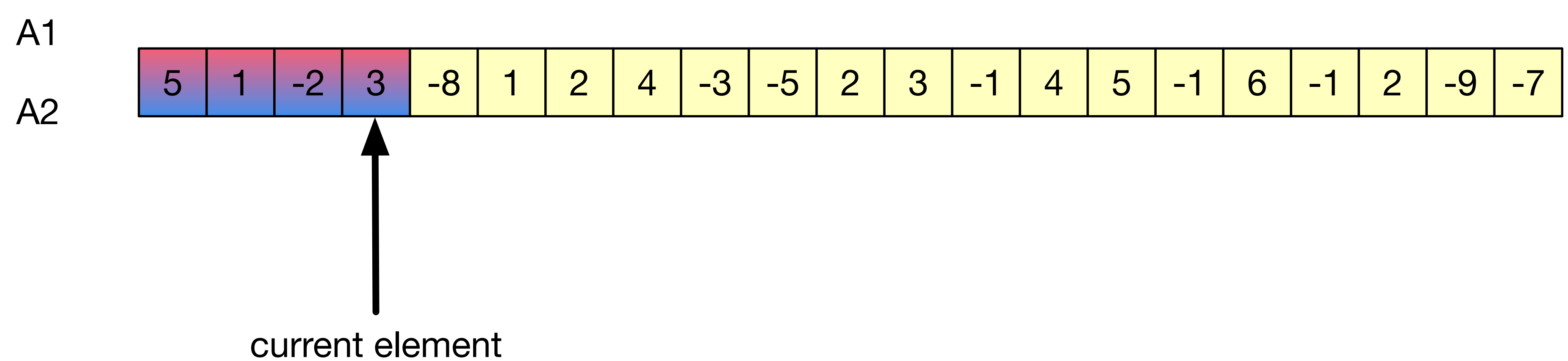
Dynamic Programming

- Example
  - sum(A1)=6, sum(A2)=4



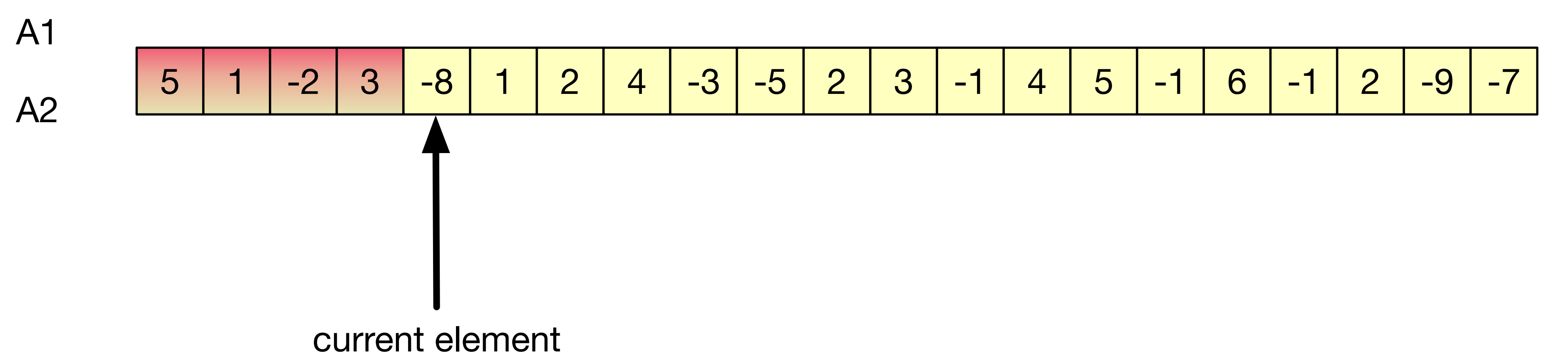
Dynamic Programming

- Example
    - sum(A2)=7, sum(A1) = max(6,7)=7



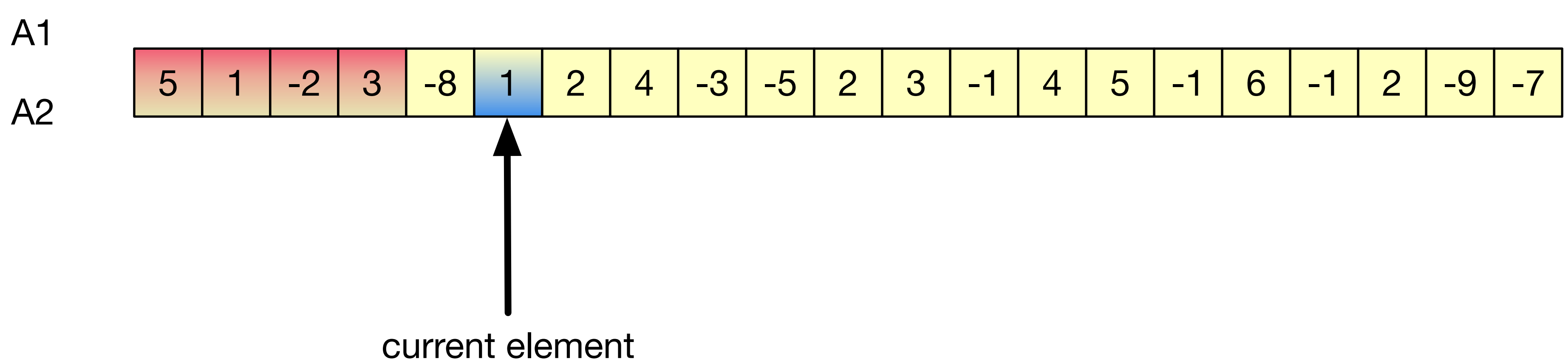
Dynamic Programming

- Example
  - A2 becomes empty, sum(A1)=7



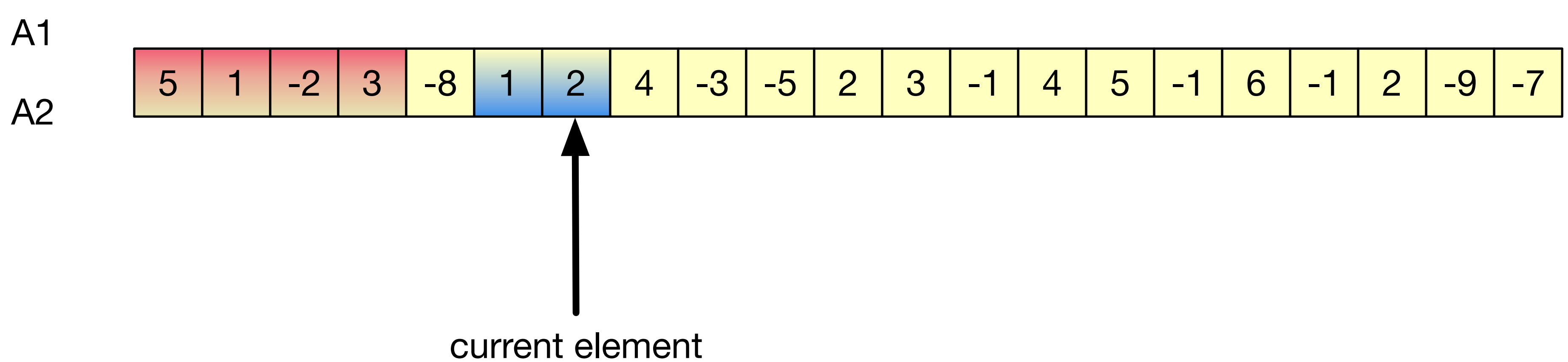
Dynamic Programming

- Example
    - sum(A2) = 1, sum(A1) = 7



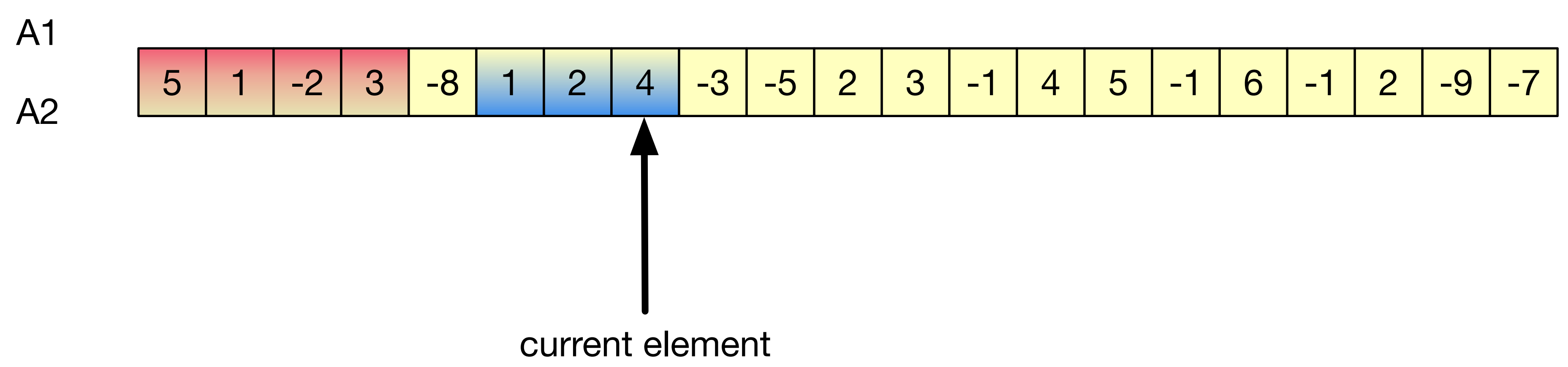
Dynamic Programming

- Example
  - sum(A1) = 7, sum(A2) = 3



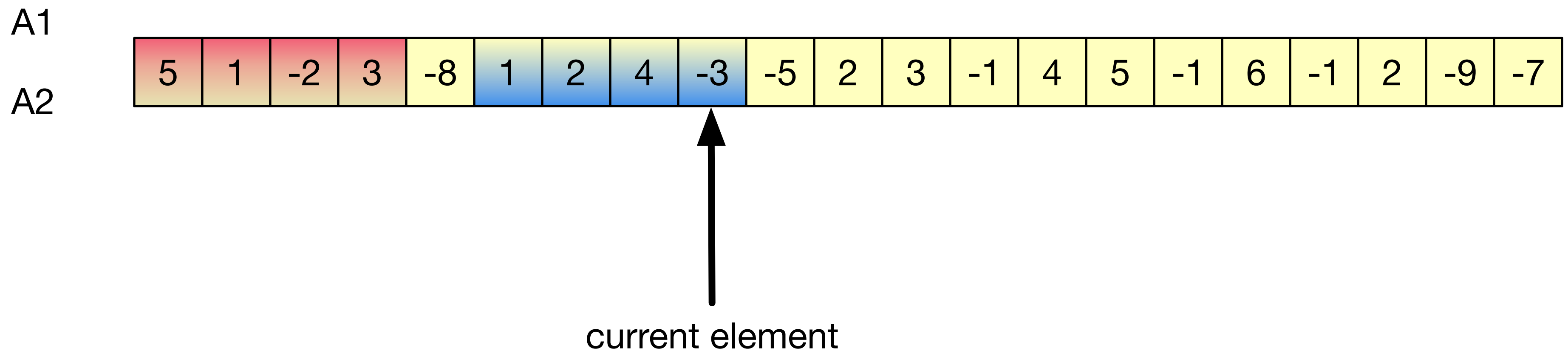
Dynamic Programming

- Example
  - sum(A1) = 7, sum(A2) = 7



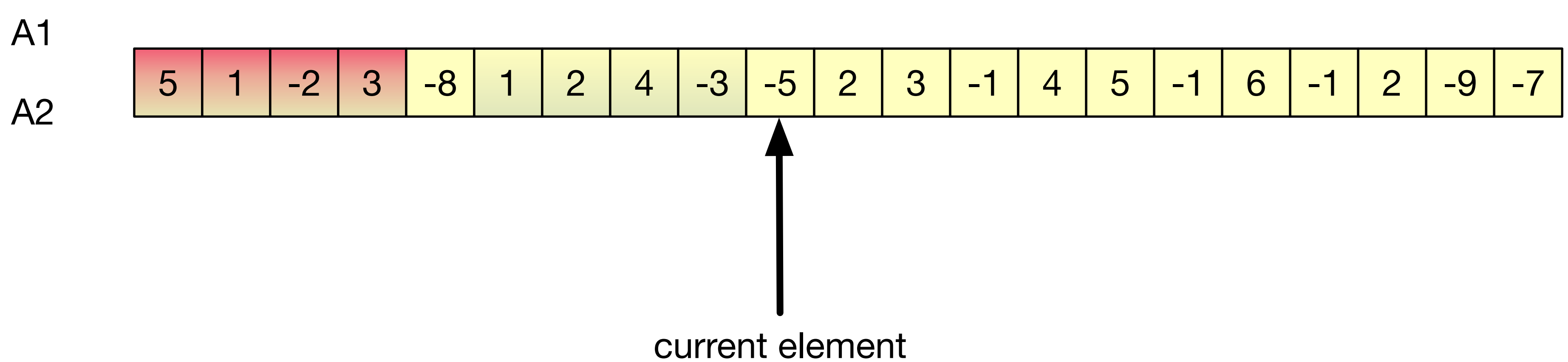
Dynamic Programming

- Example:
  - sum(A1)=7, sum(A2)=4



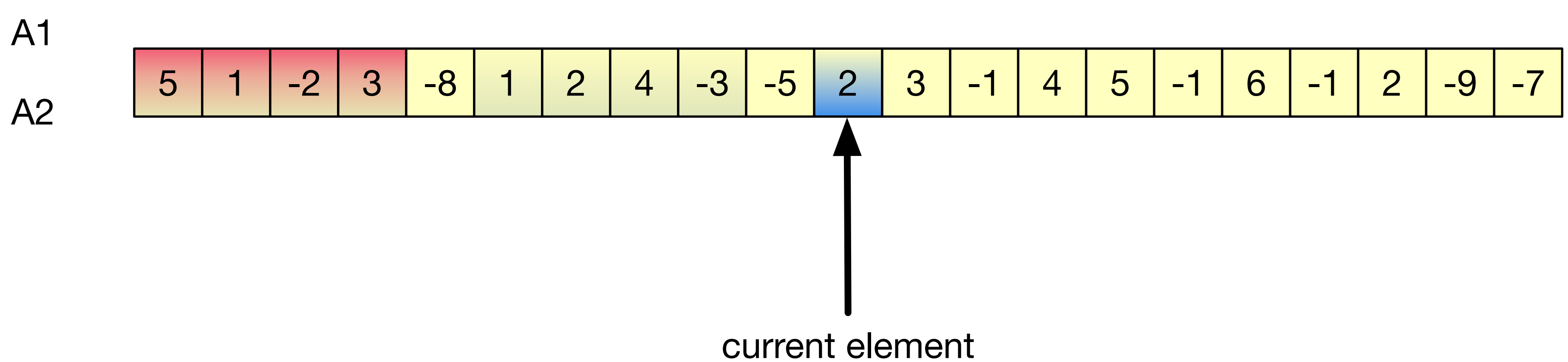
Dynamic Programming

- Example



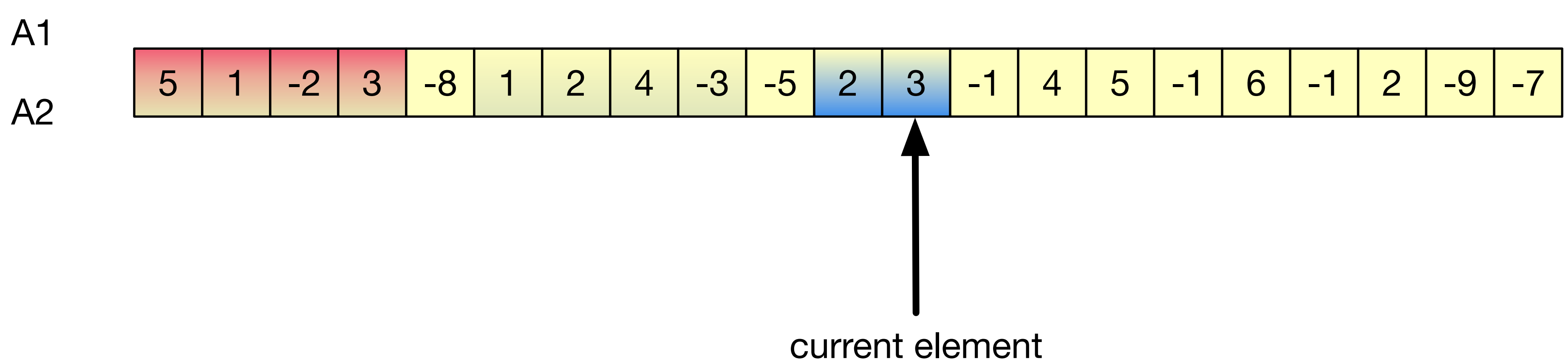
Dynamic Programming

- Example
  - sum(A1)=7, sum(A2)=2



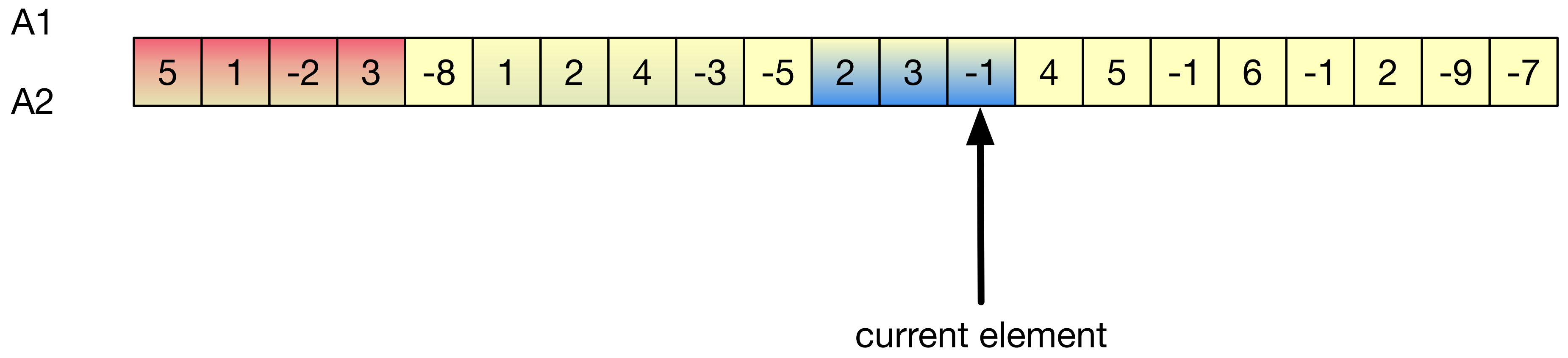
Dynamic Programming

- Example
  - sum(A1) = 7, sum(A2)=5



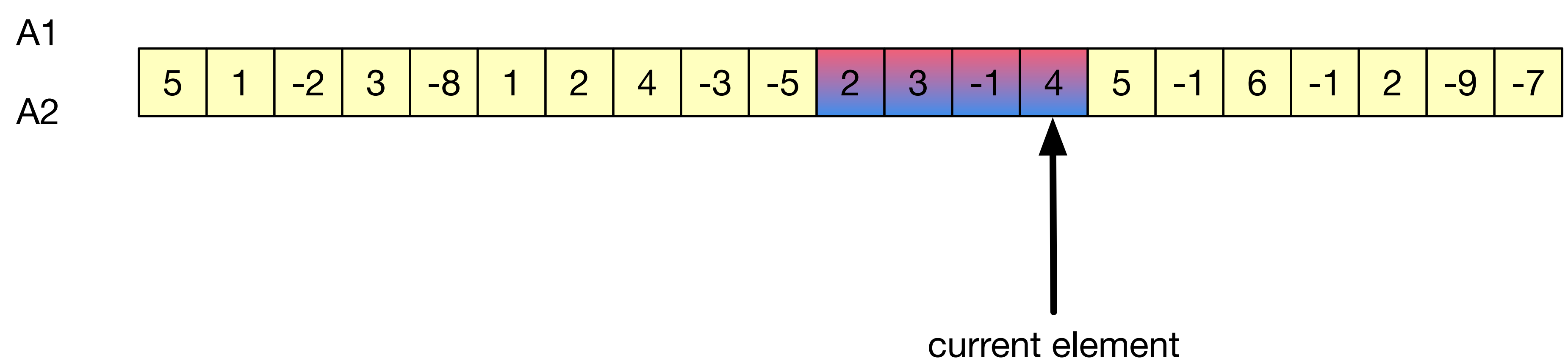
Dynamic Programming

- Example:
  - sum(A1)=7, sum(A2)=4



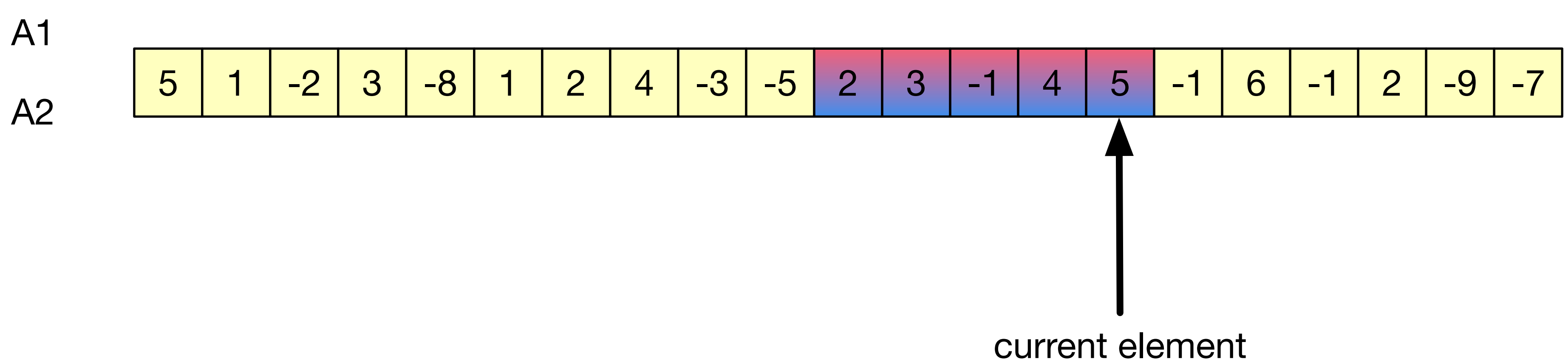
Dynamic Programming

- Example:
  - sum(A1) = sum(A2) = 8



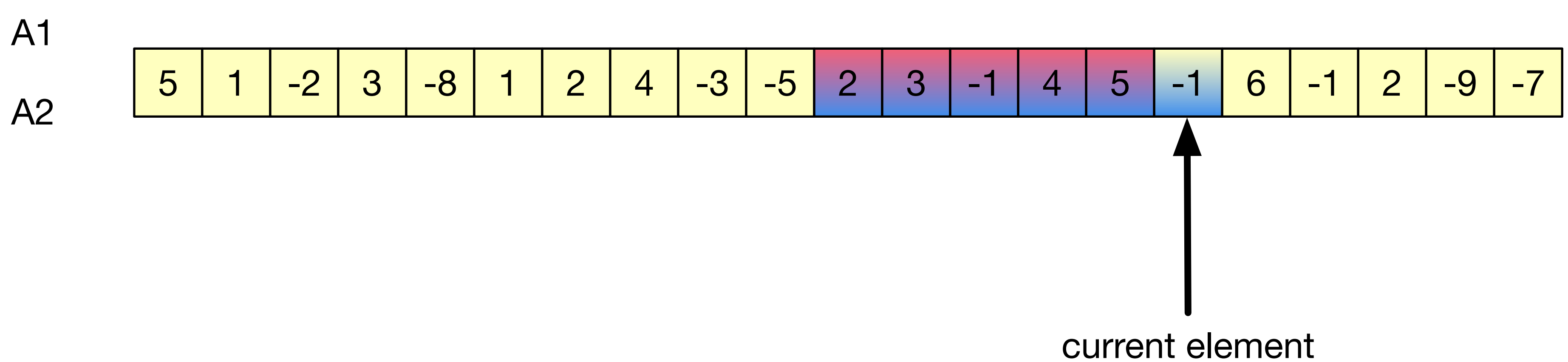
Dynamic Programming

- Example:
    - sum(A1)=sum(A2)=13



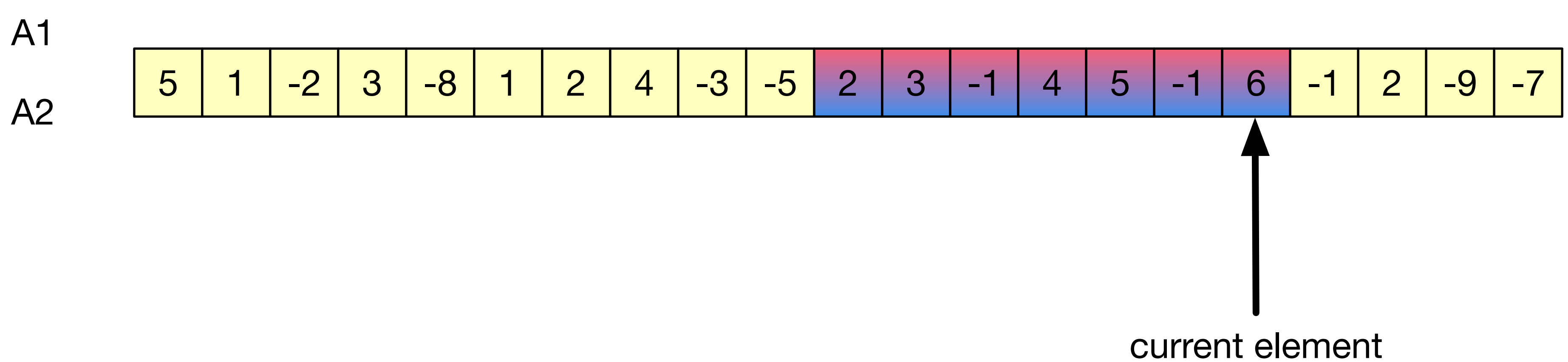
Dynamic Programming

- Example:
  - sum(A1)=13, sum(A2)=12



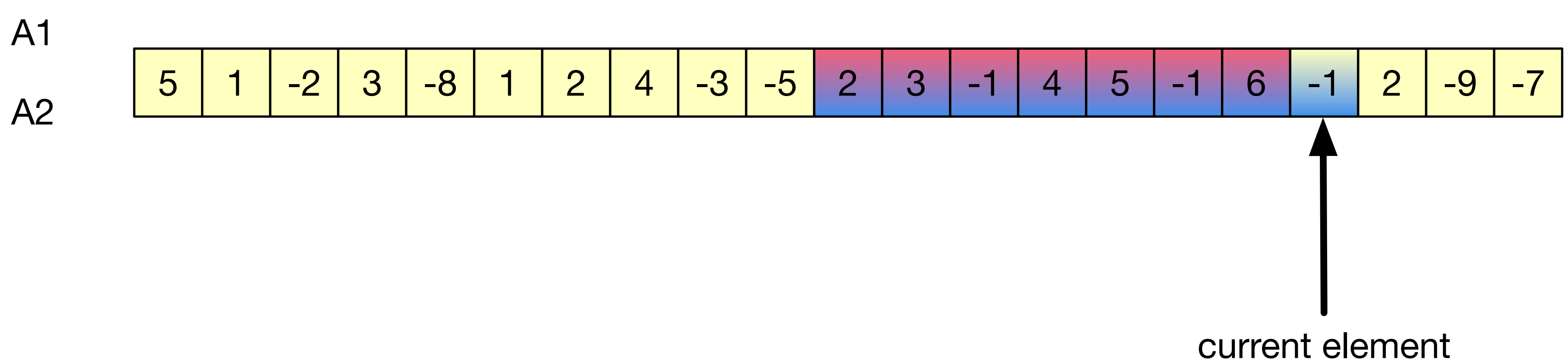
Dynamic Programming

- Example:
  - sum(A1)=sum(A2)=18



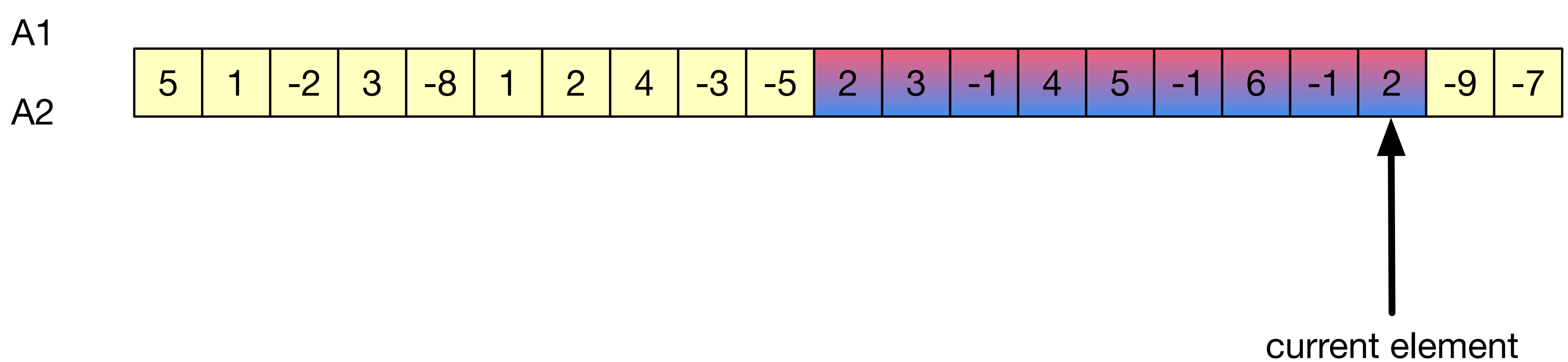
Dynamic Programming

- Example:
  - sum(A1)=18, sum(A2)=17



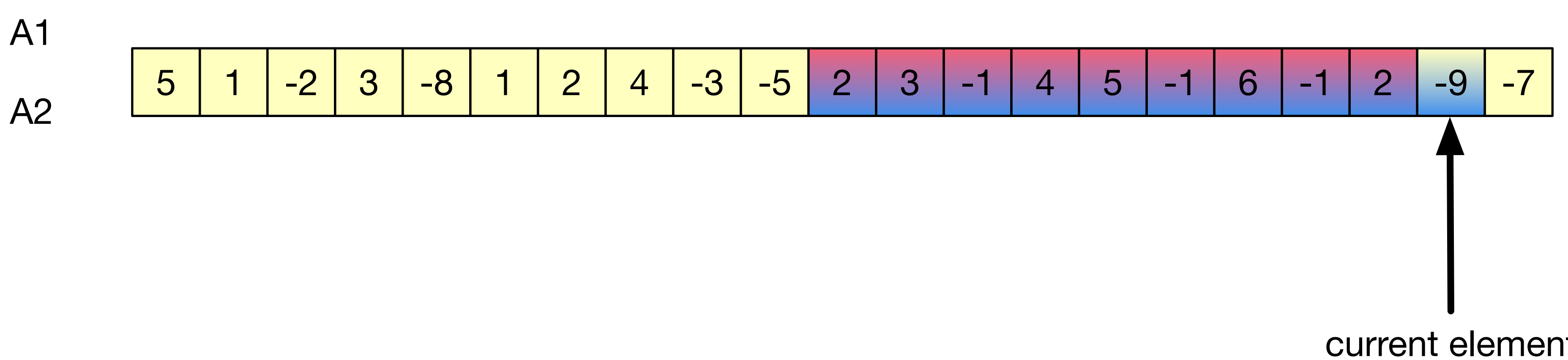
Dynamic Programming

- Example:
  - sum(A1)=sum(A2)=19



Dynamic Programming

- Example:
  - sum(A1)=19, sum(A2)=10



Dynamic Programming

- Example:
  - sum(A1)=19, sum(A2)=3