Programming Practice

2018-10-18

Week 7

Notice

- TA Email : pp20182ta@gmail.com

  -> You can get an answer faster than personal TA email.

- TA's answer code

  -> We uploaded answer code for previous assignments on class website. (mrl.snu.ac.kr)

  -> Except few assignments which could be an answer for this week.

- Late Policy

  -> **1 day** : -25% **2 day** : -50% **3 or more** : No score

Errors & Warnings

When you compile C program with gcc you might see some errors and warnings.

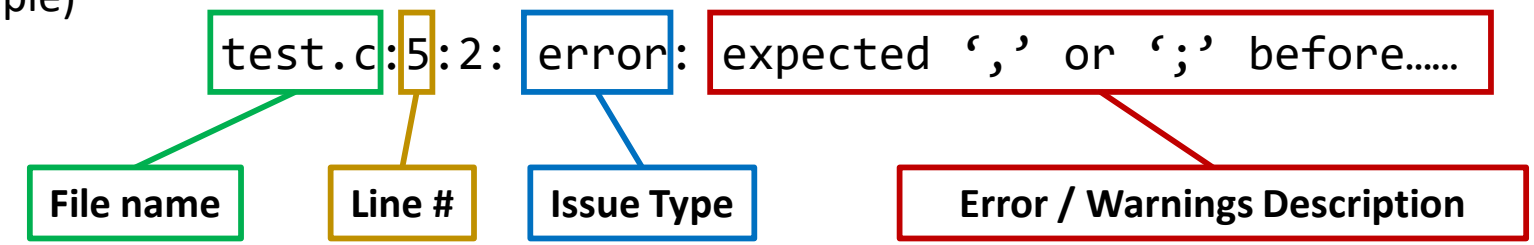
Errors -> Cannot compile. Check your code again.

-> Semi-colon, variable declaration, parenthesis, etc.

Warnings -> Compiled but your program could be run unintended way.

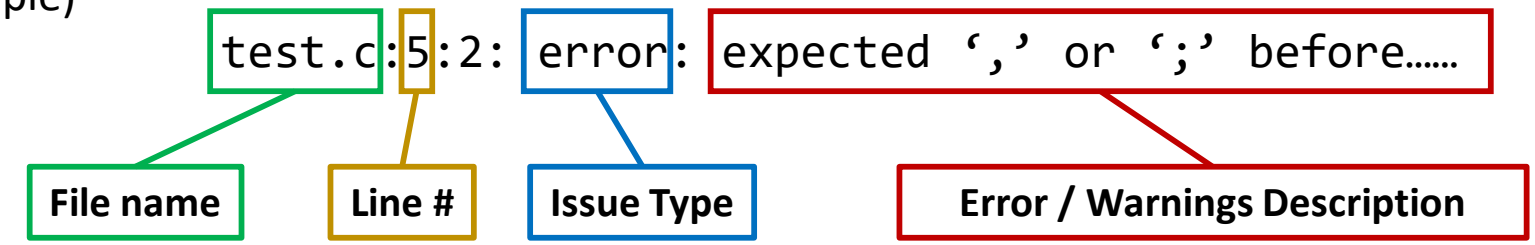
-> return value, type casting, unused variable, etc.

Example)



Errors & Warnings

Example)



How to debug yourself

1. Open **`test.c`** file
2. Go to Line **5**
3. Check your code whether the line was finished correctly by semi-colon.

4. If so, check 1 more line above / below. (In this case, check line 4 and 6 also)

Frequent ‘Warning’s

> test.c:8:1: **warning**: ‘n’ is used uninitialized in this function…

This warning means that you used uninitialized variable.

Programming safely -> Try to initialize variable every time you declare.

> test.c:6:2: **warning**: unused variable ‘k’…

You declared variable named ‘k’ but it’s not used. You can just delete this variable.

Frequent ‘Warning’s

> test.c:8:1: **warning**: ignoring return value of '**`scanf`**’…

It’s not necessary to use the return value of **`scanf`** function. You can ignore it.

> test.c:4:2: **warning**: ‘return’ with no value, in function returning…

Only main function can return without value even though it’s **`int`** type (**`int main()`…**)

But you should change your code to return correct value **`0`**

Because of the safety / stability issue.

```
1 int main(void) {
2     return;
3 }
```

```
1 int main(void) {
2     return 0;
3 }
```

Coding Convention

- Implicit rule for coding

  Indentation, variable name, parenthesis…

- Improve the readability
- Make maintenance easier

  - Some of you might be unfamiliar with it. But please keep trying to write code in-convention.

  - When you send email to us for questions,
    you can get an answer faster when your code is easy to read.

Example code

```c
#include <stdio.h>

int add(int arg1, int arg2)
{
    return arg1 + arg2;
}

int main(void)
{
    int start = 1, offset = 2, sum = 0;
    for (int i = start; i < 3; i++) {
        sum += add(i, offset);
        if (sum < 0) {
            printf("ERROR!\n");
        }
    }
    printf("%d\n", sum);
    return 0;
}
```

Use clear name for variable / function
Only exception is loop index

If you think this part is too dense,
You can put empty line between them

Write braces {,} in separate line
makes easier to identify
the range of function / loop

Array & Pointer

```
int v;

int *p;

int a[10];
```

Expression		Type
v		int
&v		pointer
p		pointer
*p		int
a		pointer
*a		int
a[0]		int
*(&v)		int

&(x) : Referencing
“return Location where x is stored”
“return y if MEMORY[y] = x”

*(x) : Dereferencing
“return Value stored in x”
“Look up MEMORY[x]”

Array & Pointer

%d means integer type
%p means pointer type (memory address)

```c
#include <stdio.h>

int main(void) {
    int n = 7;
    int *p; // Declare pointer variable
    int a[4] = {0, 1, 2 ,3};    // Declare array

    // Assign (address of n) into p
    p = &n;

    printf("n       : %d\n", n);
    printf("&n      : %p\n", &n);
    printf("*(&n)   : %d\n", *(&n));
    printf("p       : %p\n", p);
    printf("*p      : %d\n", *p);
    printf("----------------\n");
    printf("a[0]    : %d\n", a[0]);
    printf("&a[0]   : %p\n", &a[0]);
    printf("a       : %p\n", a);
    printf("*a      : %d\n", *a);
    printf("*(a+1)  : %d\n", *(a+1));
    printf("a[1]    : %d\n", a[1]);

    return 0;
}
```

Variable **n**
is stored in memory address
0x7fffc75937d4

< Result >

```
n       : 7
&n      : 0x7fffc75937d4
*(&n)   : 7
p       : 0x7fffc75937d4
*p      : 7
----------------
a[0]    : 0
&a[0]   : 0x7fffc75937e0
a       : 0x7fffc75937e0
*a      : 0
*(a+1)  : 1
a[1]    : 1
```

← Array itself is a pointer

You can access element
In both way

We've used **scanf** function without understanding what '&' was.

Array & Pointer

```
printf("%p", &n);
```

= "Print the address of variable n"

Let's assume that the result was…

0x7fffc74c

```
scanf("%d", &n);
```

= "Read 1 integer value and then store it into **0x7fffc74c** (address of **n**)"

Homework Problems

1. Swap
2. Array Comparison
3. Binary Search 2
4. Bubble Sort
5. Greatest Common Divisor
6. Efficient Maximum Subarray

Notice for Assignment 1, 2

- For assignment 1 and 2, all you need to do is implement function body.
- Do NOT modify given `main.c` file

main.c

```c
#include <stdio.h>

int function(int a, int b);

int main(void) {
        printf("%d\n", function(1, 2));
        return 0;
}
```

`main.c` uses your function

function.c

```c
#include <stdio.h>

int function(int a, int b) {
        /* Implement this function */
        int answer = 0;
        return answer;
}
```

Blue part is dummy.
It will be compiled successfully with wrong answer.
You need to fix it.

Notice for Assignment 1, 2

- How to test?

main.c

```
#include <stdio.h>
…
```

function.c

```
#include <stdio.h>
…
```

1. Put those 2 file in same folder. (copy & paste `main.c` file)
2. Compile with following command

```
gcc –o <filename> main.c function.c
```

3. Run

Problem. **1**

Swap

Description

Implement a ‘swap’ function.

The swap function takes two int pointer variables, and swaps the numbers in the two positions.

You must follow the function prototype:

```
void swap(int *a, int *b);
```

Submit the c code containing just your swap function. Do not include the main() function in your submission code.

- Notice for Submission -

Filename should be `swap.c`

Other filename will be rejected.

</> source code

```c
#include <stdio.h>

void swap(int *a, int *b){
    /* TODO */
}

```

Problem. **1**

Swap (cont.)

[swap.c]

</> source code

```c
#include <stdio.h>

void swap(int *a, int *b){
    /* TODO */
}

```

Following main() will be linked by server automatically

```c
int main(void) {
    int a, b;
    scanf("%d %d", &a, &b);
    swap(&a, &b);
    printf("%d %d", a, b);
    return 0;
}
```

Sample

[input]

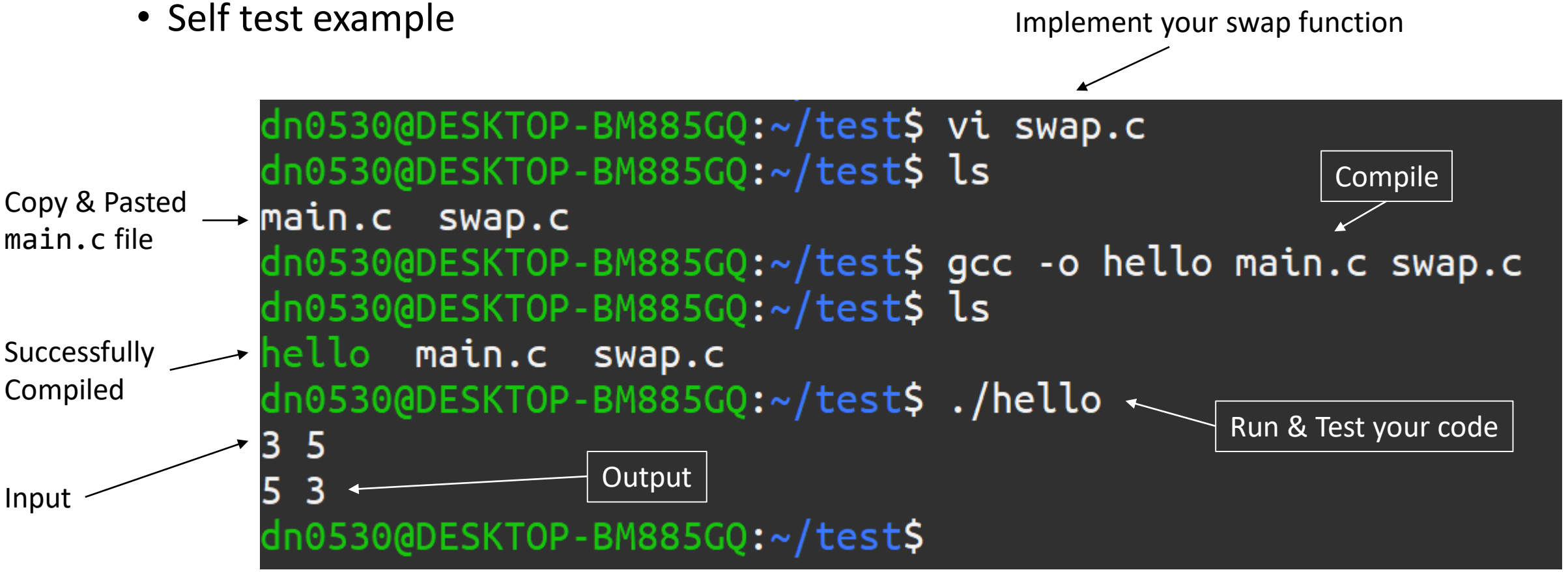
1 99

[output]

99 1

Notice for Assignment 1, 2

- Self test example



Problem. **2**

Array Comparison

Description

Implement `array_cmp()` function.

`array_cmp()` function takes length of array, array A, and array B. The second line contains N integers, all of which are single digit (from 0 to 9).

It returns -1 if decimal representation of A is smaller than B. Returns 1 if A is greater than B. Returns 0 if A and B are same. Decimal representation of array A means the number we can get by concatenating every elements of A.

Input

First line contains a single integer N $(1 \leq N \leq 10{,}000)$.

The second and third lines contain N single-digit integers.

Output

It returns -1 if decimal representation of A is smaller than B. Returns 1 if A is greater than B. Returns 0 if A and B are same.

Sample

```
[input]          [input]
5                5
1 2 3 4 5        5 4 3 2 1
1 3 3 3 3        3 3 3 3 3
[output]         [output]
-1               1
---------------  ---------------
[input]          [output]
5                0
1 2 3 4 5
1 2 3 4 5
```

Problem. **2**

Array Comparison (Cont.)

[main.c]

```c
#include <stdio.h>

int array_cmp(int, int *, int *);
int main() {
    int N;
    scanf("%d", &N);
    int A[N];
    int B[N];
    for (int i = 0; i < N; ++i) {
        scanf("%d", A + i);
    }
    for (int i = 0; i < N; ++i) {
        scanf("%d", B + i);
    }
    printf("%d\n", array_cmp(N, A, B));
    return 0;
}
```

[array_cmp.c]

```c
int array_cmp(int N, int *A, int *B) {
    /* TODO! */
}
```

- Notice for Submission -

Filename should be `array_cmp.c`

Other filename will be rejected.

Problem. **3**

Binary Search 2

Description

'Binary Search' searches for the target integer by continuously cutting the array in half until the target is found (or not found).

Let's redo the Binary Search. You will be given N pre-sorted integers without duplicates. This time, we'll search for not 1 but M target integers. (Use a for-loop to do M binary searches.)

Add up the results of all M binary searches, and print the sum. (As before, a search result equals to the **1-based index** of the target integer if found, or equals to `-1` if not found.)

Input

1st line: N
2nd line: [N integers]
3rd line: M
4th line: [M integers]

(Where $1 \leq N \leq 100000$ and $1 \leq M \leq 100000$, and the integers in 2nd, 4th line are all of `int` range.)

Output

Print the total sum of all M binary search results.

Sample

```
[input]
5
2 3 5 7 11
3
1 2 3

[output]
2

// because -1 + 1 + 2
```

It's guaranteed that there are no duplicates in 2nd line.

Notice that there could be an integer overflow while calculating sum

Efficient way to search value

….. Last week, there were few students who used linear search (not binary)

Look up whole array one by one

It's too slow!!! Linear search won't work this week.

Binary search algorithm

1. Calculate middle point

In C, integer type division $\frac{n}{2}$ equals to $\left\lfloor \frac{n}{2} \right\rfloor$ (when $n > 0$)
Thus, you don't have to check if n is even or odd.

2. Compare with target value
3. If target value is equal to middle point, return index.
4. If target value is bigger than middle point, find upper half array recursively. vice versa

Compare Efficiency (binary <-> linear)

Let's assume that initial array has $\mathbf{2^{10} = 1024}$ elements.

< Binary Search >

- Compare once (with midpoint)
- If missed, the size of array that we need to lookup shrinks into half each time.

$$2^{10} \rightarrow 2^{9} \rightarrow 2^{8} \rightarrow \cdots \rightarrow 2^{0}$$

- Maximum 10 step to find index.

< Linear Search >

- Compare once (with first element)
- If missed, the size of array that we need to lookup decreased by 1.

$$1024 \rightarrow 1023 \rightarrow 1022 \rightarrow \cdots \rightarrow 1$$

- Maximum 1024 step to find index.
- Average 512 step to find index.

In general, let's assume that initial array has N elements.

It needs $\log_2 N$ steps with **Binary Search Algorithm**.

It needs N steps with **Linear Search Algorithm**.

If N is big enough,

$$\mathbf{\log_2 N \ll N}$$

Problem. **4**

Bubble Sort

Description

Write a program that gets a number of integers as input, and sort the array in ascending order by using **bubble sort**.

The first line of the input will state the number of integers that will be given : N $(1 \leq N \leq 5000)$.

The second line contains N integers in ascending order without duplicates, all of which are within the range of `int`.

Print the ascending sorted array.

Input

First line contains a single integer N $(1 \leq N \leq 5000)$.

The second line contains N integers in ascending order.

Output

Print the ascending sorted array.

Sample

```
[input]
5
6 -5 -7 6 2

[output]
-7 -5 2 6 6
```

Problem. **5**

Greatest Common Divisor

Description

Let's find the Greatest Common Divisor of pairs of numbers!

N $(1 \leq N \leq 100)$ pairs of positive integers will be given, one pair in each line. All of the positive integers will be within `int` range.

For all N pairs, find each pair's GCD. Add all of the N GCDs and print the total sum.

(Hint1: write a recursive function and use it.)
(Hint2: use `long long` for the variable to save the total sum.)

Input

On the first line: an integer N $(1 \leq N \leq 100)$.
On each of the following N lines: a pair of positive integers x_i, y_i (all within `int` range, delimited by a single space).

Output

Print the sum of all N GCDs.

Sample

```
[input]
3
4 6
5 10
3 7

[output]
8

// because 2 + 5 + 1
```

Euclidean Algorithm

- Efficient method for computing the **greatest common divisor** (GCD)

Let's assume that a > b. Then,

gcd (a, b) = gcd (b, a-b)
= gcd (b, a-2b) (if a > 2b)
= gcd (b, a%b) (in general case)

gcd (x, 0) = x

- Example 1

gcd (7, 3) = gcd (3, 7%3) = gcd (3, 2)
= gcd (2, 3%2) = gcd (2, 1)
= gcd (1, 2%1) = gcd (1, 0) = 1

- Example 2

gcd (6, 4) = gcd (4, 6%4) = gcd (4, 2)
= gcd (2, 4%2) = gcd (2, 0) = 2

Proof of validity : https://en.wikipedia.org/wiki/Euclidean_algorithm#Proof_of_validity

Problem. **6**

Efficient Maximum Subarray

Description

Given a number of integers as input, find a subarray whose sum of its elements is the maximum. (See next slide for what is and is not a 'subarray'.)

N $(1 \leq N \leq 1{,}000{,}000)$ integers will be given.
Each integer will be in the range of [-100,000, 100,000].

Print the sum of the subarray with the maximum sum.

(Notice that there could be an integer overflow)

Input

First line contains a single integer N $(1 \leq N \leq 1{,}000{,}000)$.

Second line contains N integers, each in range [-10^5, 10^5].

Output

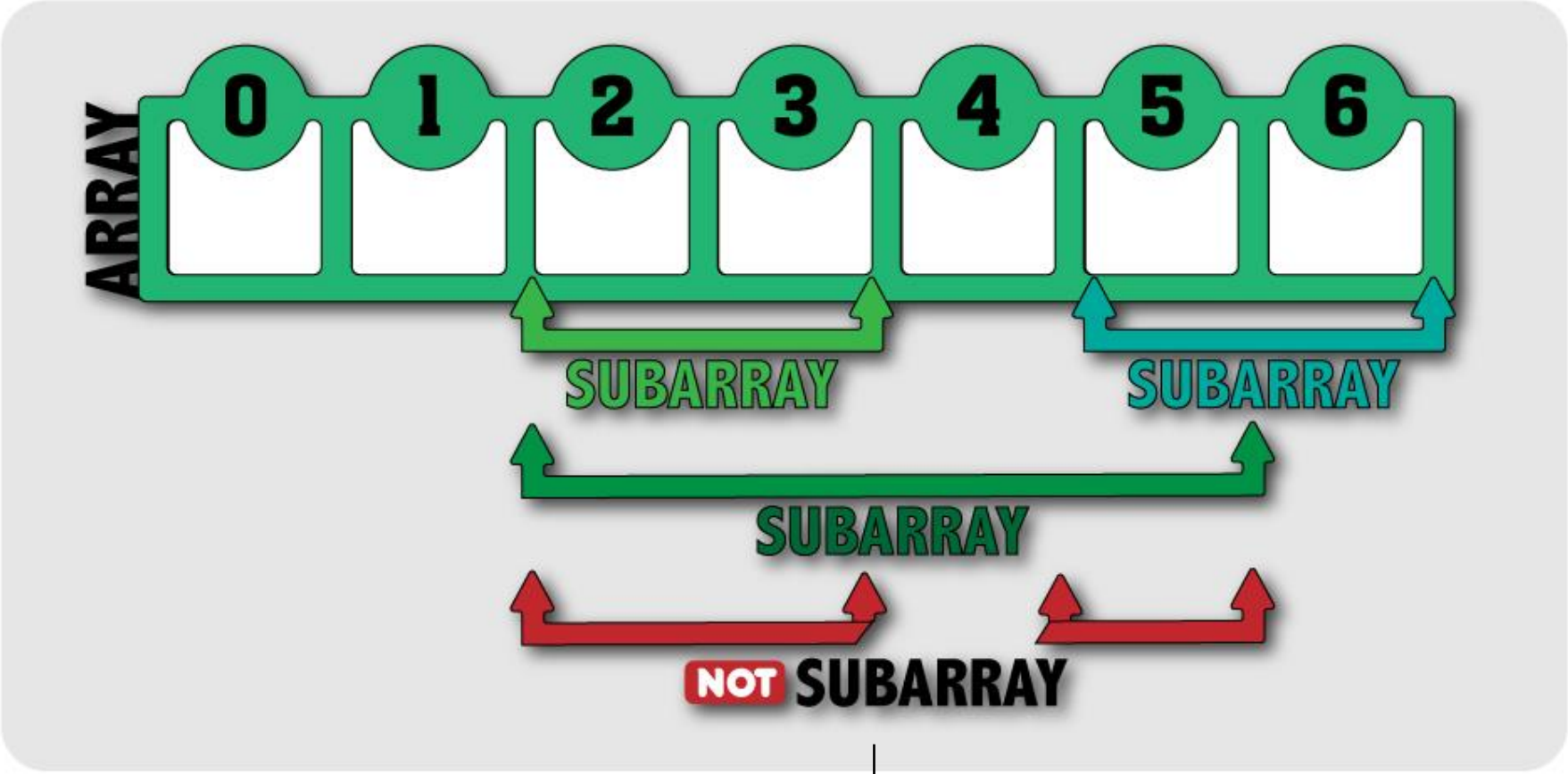
Print the sum of the maximum sum subarray.

Sample

[input]	[input]
4	5
2 -1 2 -1	0 0 0 -1 -1
[output]	[output]
3	0

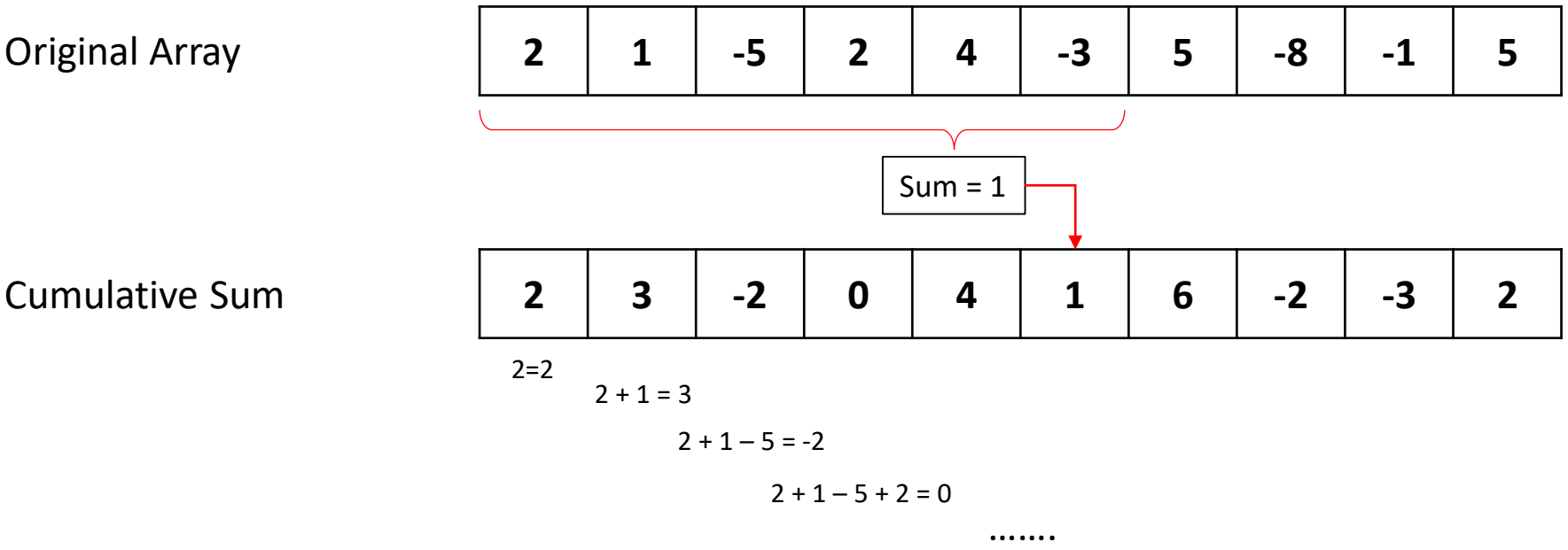
Problem. **3**

Efficient Maximum Subarray (cont.)

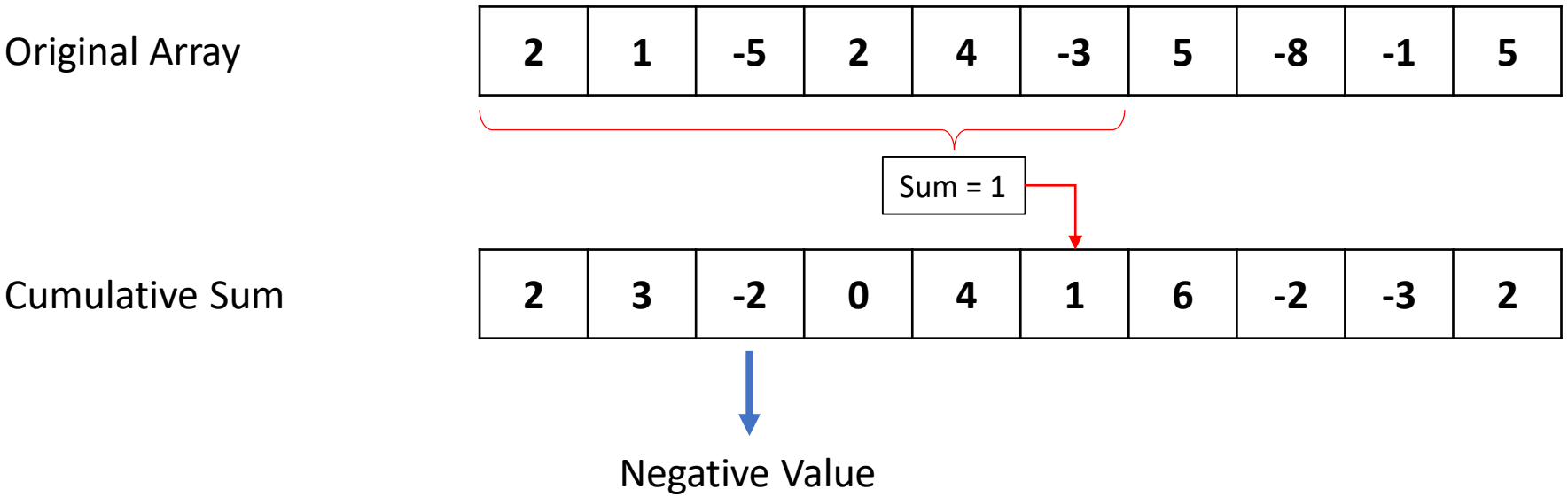


It should be continuous. Not separated

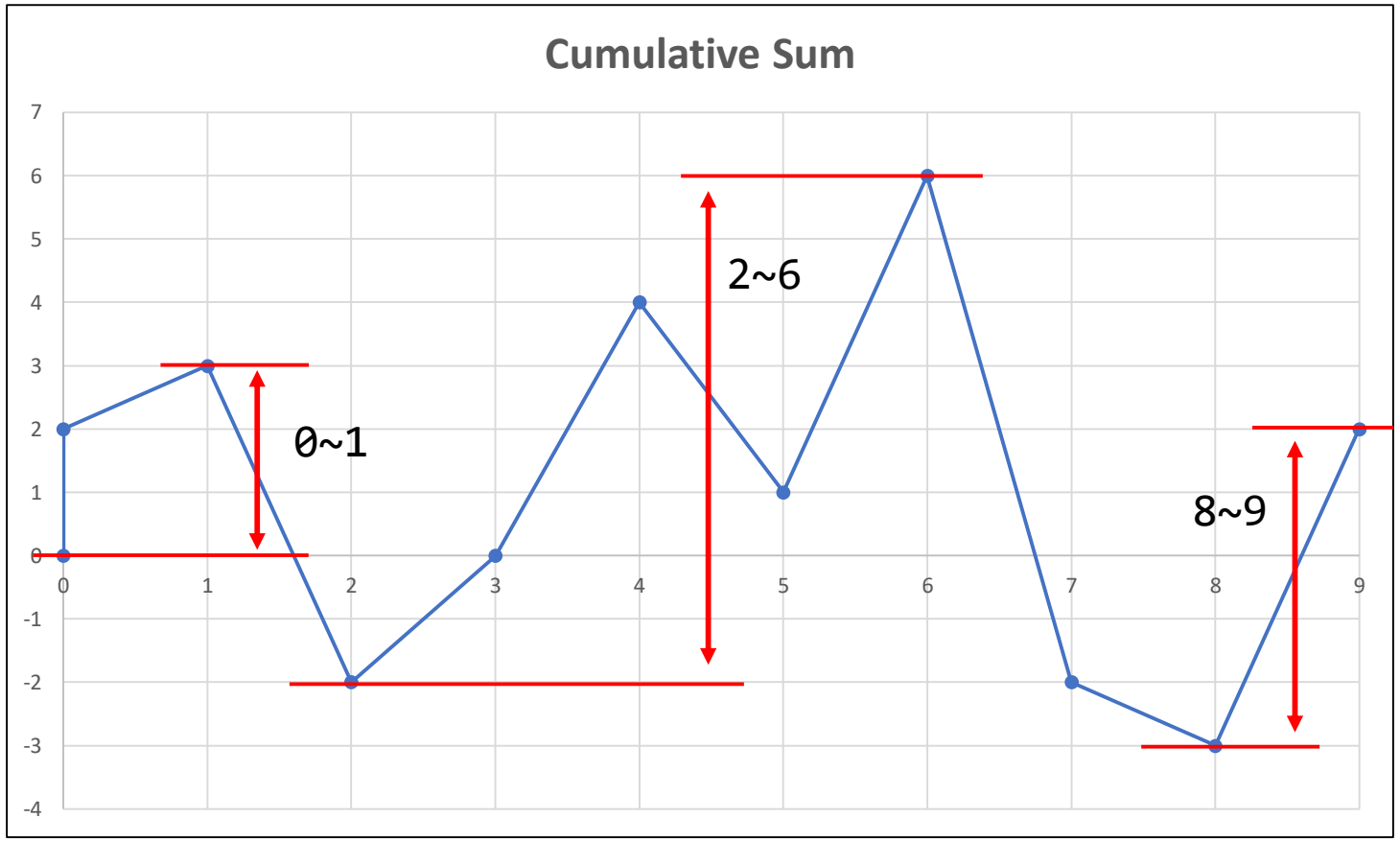
Efficient Maximum Subarray



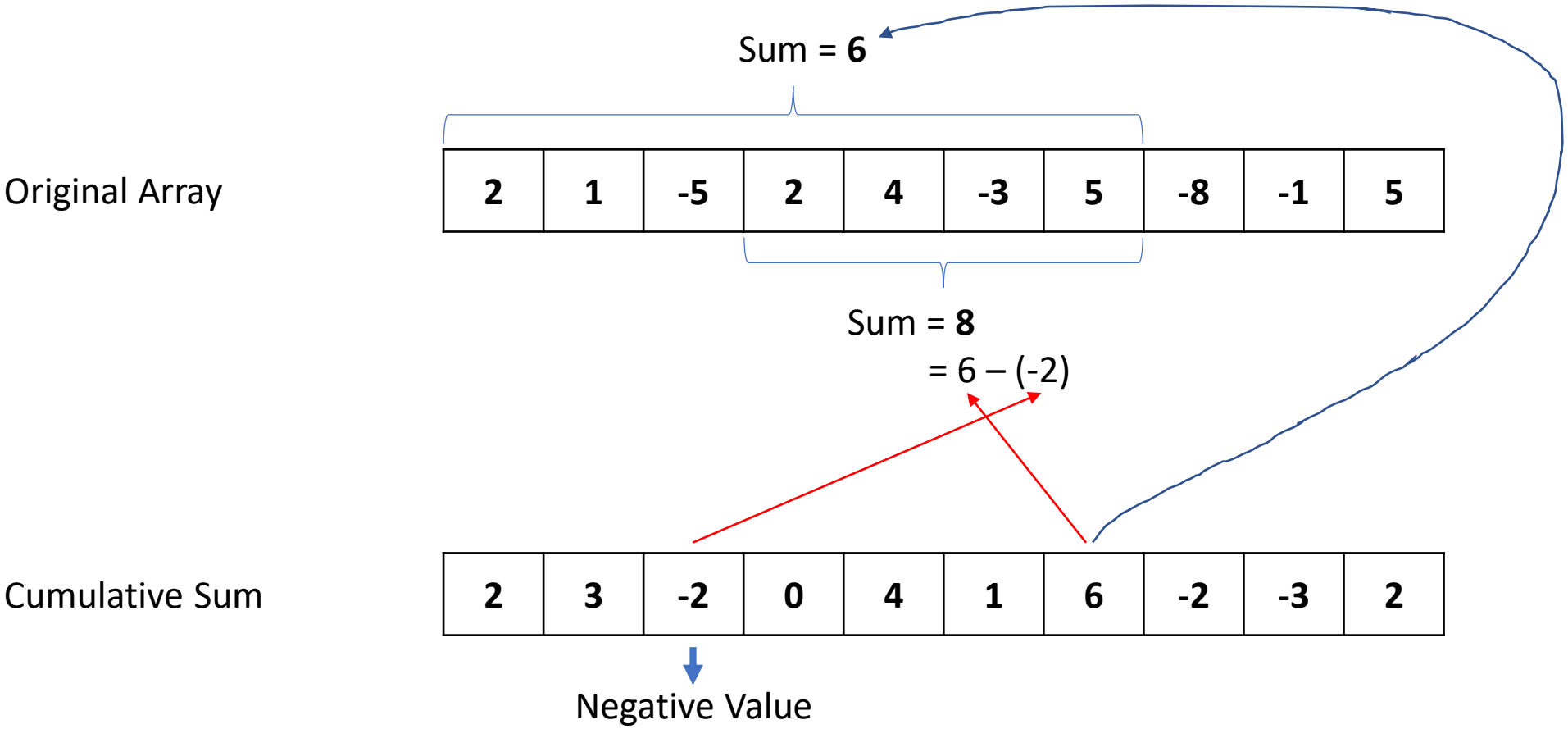
Efficient Maximum Subarray



Efficient Maximum Subarray



Efficient Maximum Subarray



-> If we meet negative value, we need to re-calculate cumulative sum from current point.

Efficient Maximum Subarray

Let’s do these calculations simultaneously!

	0	1	2	3	4	5	6	7	8	9
Original Input	**2**	**1**	**-5**	**2**	**4**	**-3**	**5**	**-8**	**-1**	**5**

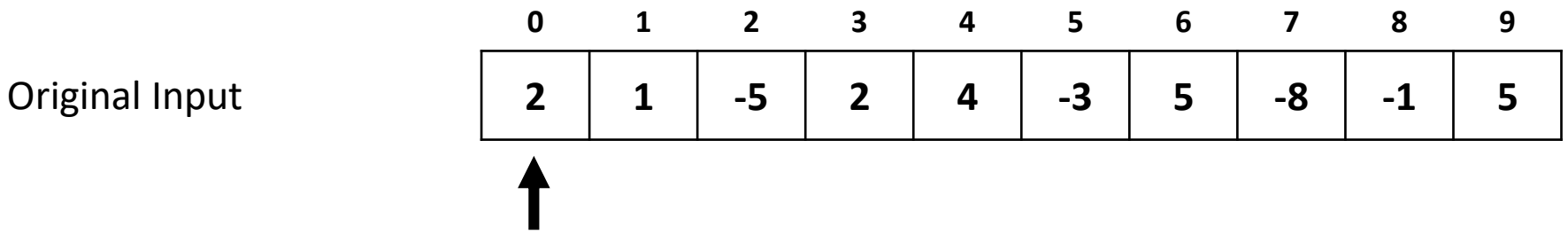
Current Index **-**

Current Value **-**

Cumulative Sum **0**

Max Sum (Result) **-10000000** → Before start, initialize max_sum small enough
If not, it would remain 0 when all elements are negative

Efficient Maximum Subarray



Current Index **0**

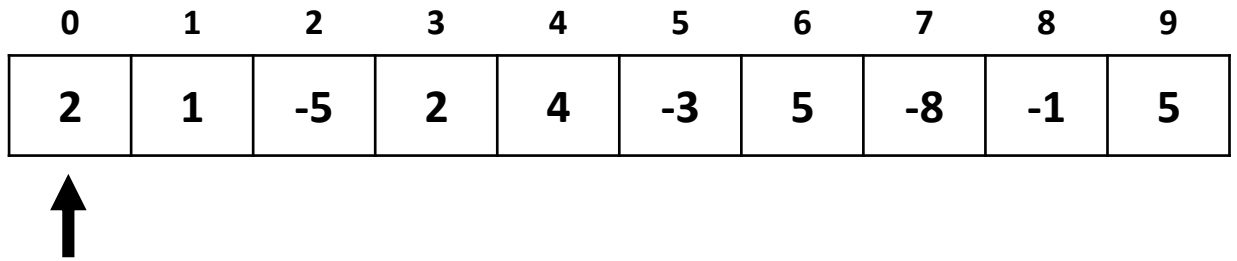
Current Value **2**

Cumulative Sum **0**

Max Sum (Result) **-10000000**

Efficient Maximum Subarray

Original Input



Current Index **0**

Current Value **2**

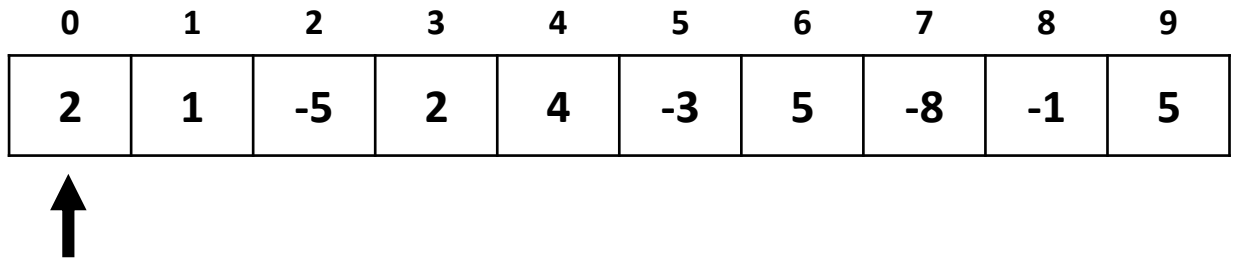
Cumulative Sum **2**

Max Sum (Result) **-10000000**

2 > -10000000 update!

Efficient Maximum Subarray

Original Input



Current Index **0**

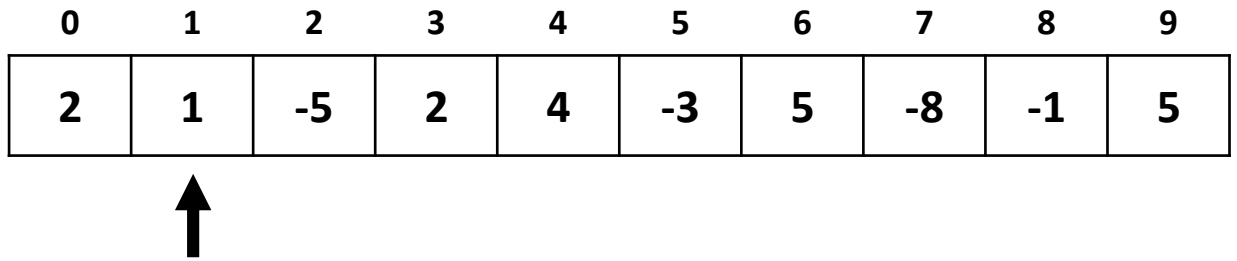
Current Value **2**

Cumulative Sum **2**

Max Sum (Result) **2**

Efficient Maximum Subarray

Original Input



Current Index **1**

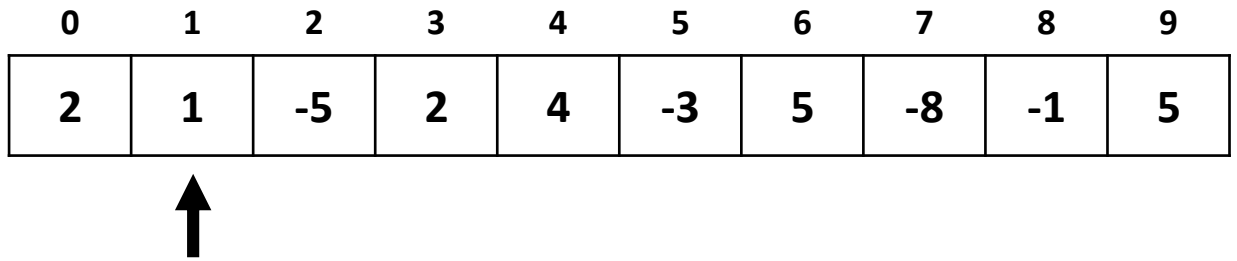
Current Value **1**

Cumulative Sum **2**

Max Sum (Result) **2**

Efficient Maximum Subarray

Original Input



Current Index **1**

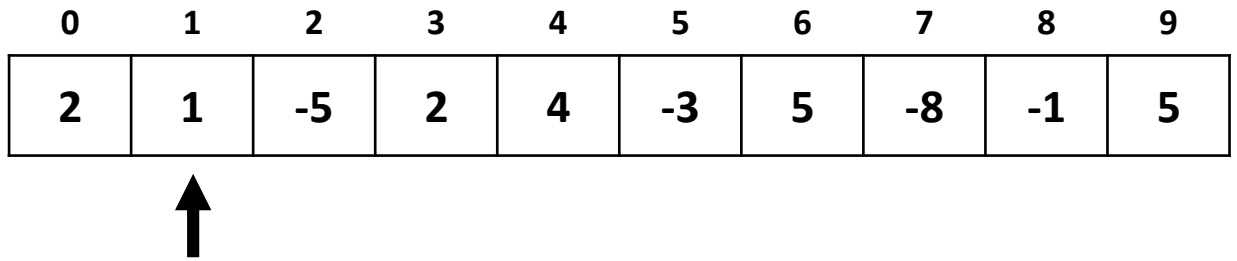
Current Value **1**

Cumulative Sum **3**

Max Sum (Result) **2**

3 > 2 update!

Efficient Maximum Subarray



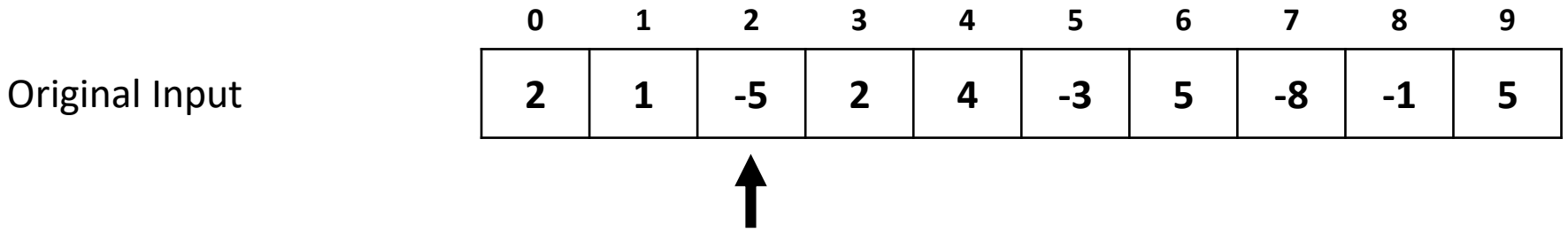
Current Index **1**

Current Value **1**

Cumulative Sum **3**

Max Sum (Result) **3**

Efficient Maximum Subarray



Current Index **2**

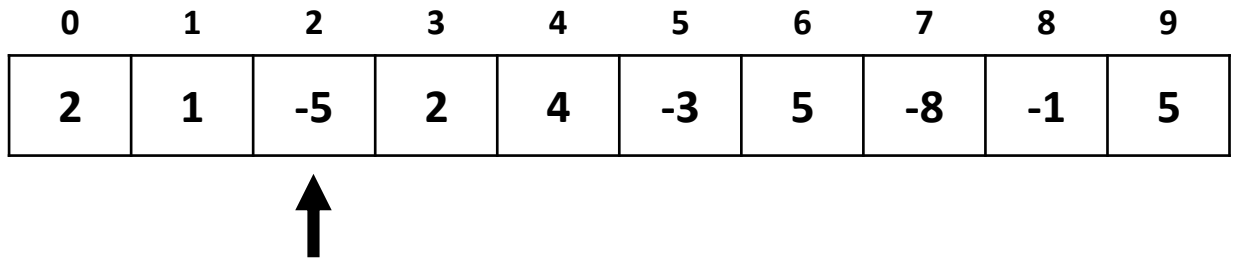
Current Value **-5**

Cumulative Sum **3**

Max Sum (Result) **3**

Efficient Maximum Subarray

Original Input



Current Index **2**

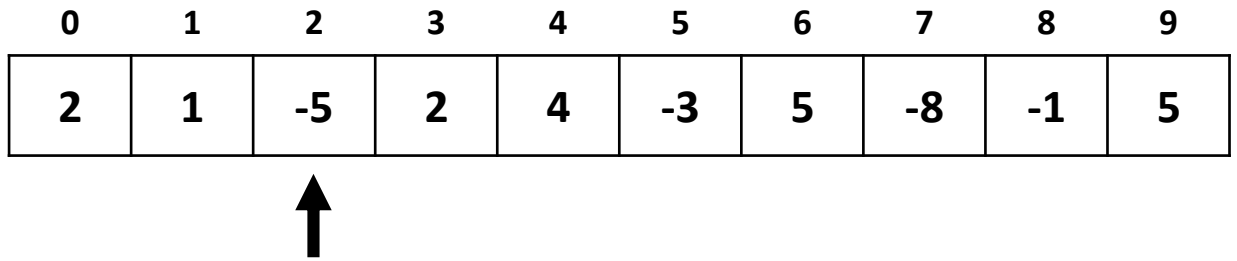
Current Value **-5**

Cumulative Sum **-2** → Negative!
Initialize cumulative sum to 0

Max Sum (Result) **3**

Efficient Maximum Subarray

Original Input



Current Index **2**

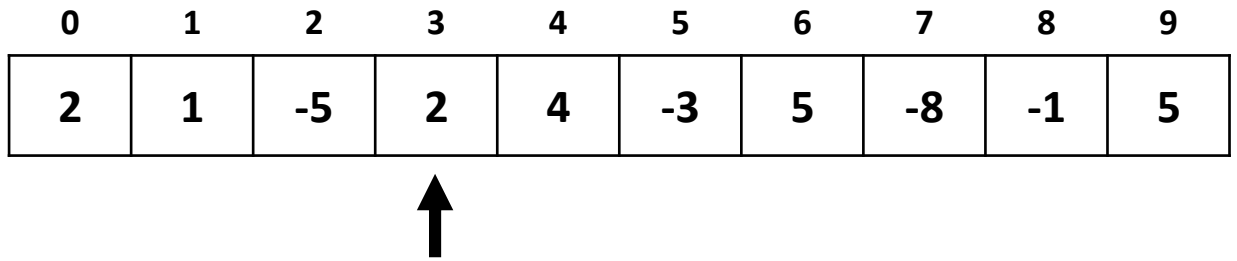
Current Value **-5**

Cumulative Sum **0**

Max Sum (Result) **3**

Efficient Maximum Subarray

Original Input



Current Index **3**

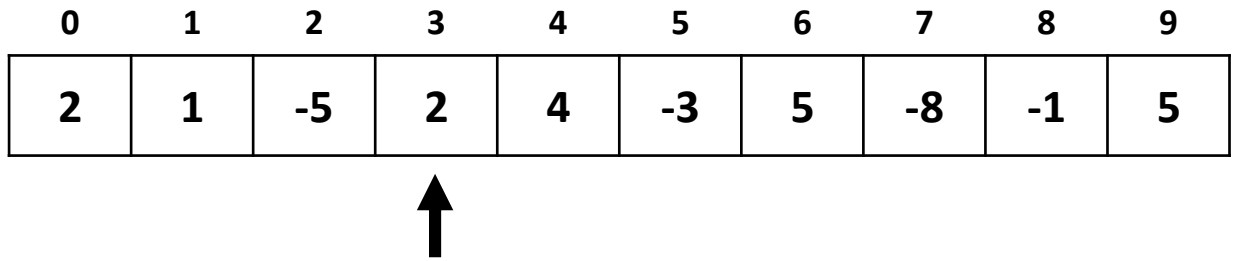
Current Value **2**

Cumulative Sum **0**

Max Sum (Result) **3**

Efficient Maximum Subarray

Original Input



Current Index **3**

Current Value **2**

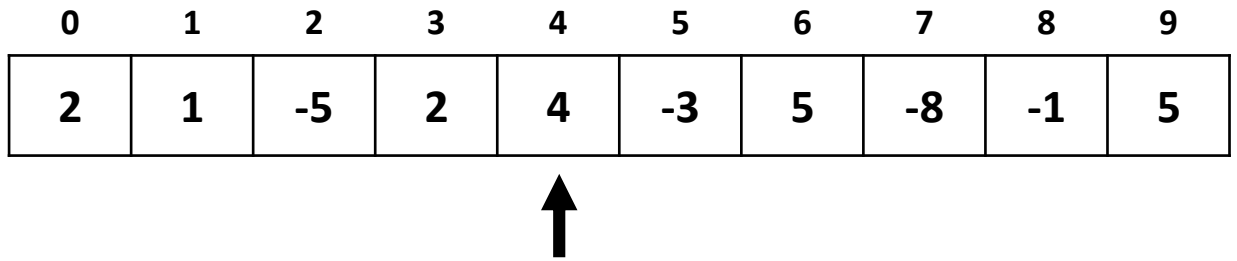
Cumulative Sum **2**

Max Sum (Result) **3**

2 < 3

Efficient Maximum Subarray

Original Input



Current Index **4**

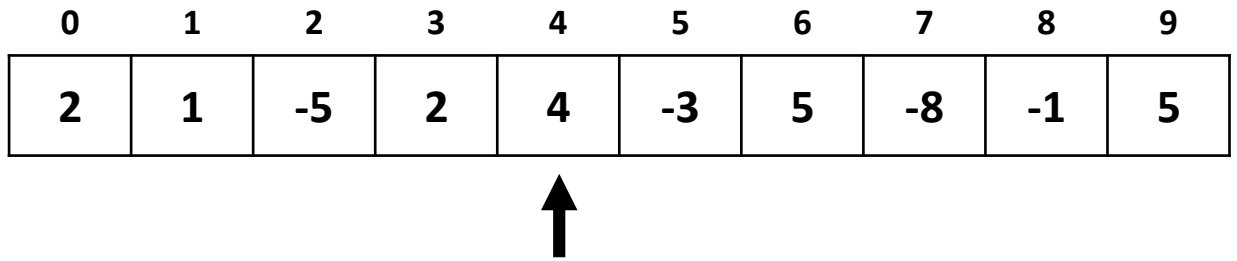
Current Value **4**

Cumulative Sum **2**

Max Sum (Result) **3**

Efficient Maximum Subarray

Original Input



Current Index **4**

Current Value **4**

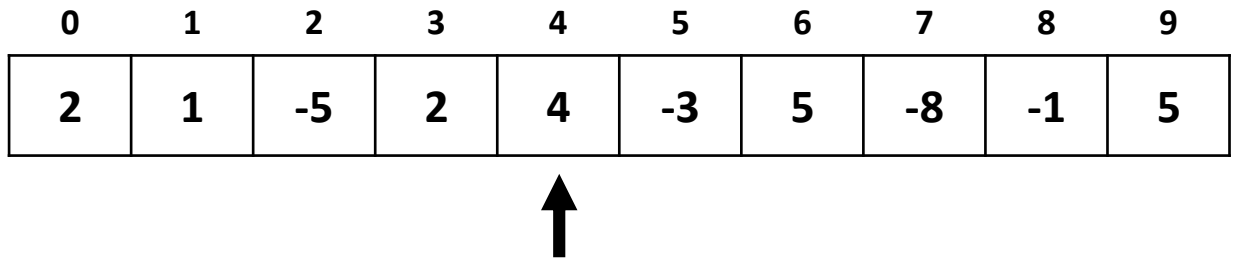
Cumulative Sum **6**

Max Sum (Result) **3**

6 > 3 update!

Efficient Maximum Subarray

Original Input



Current Index **4**

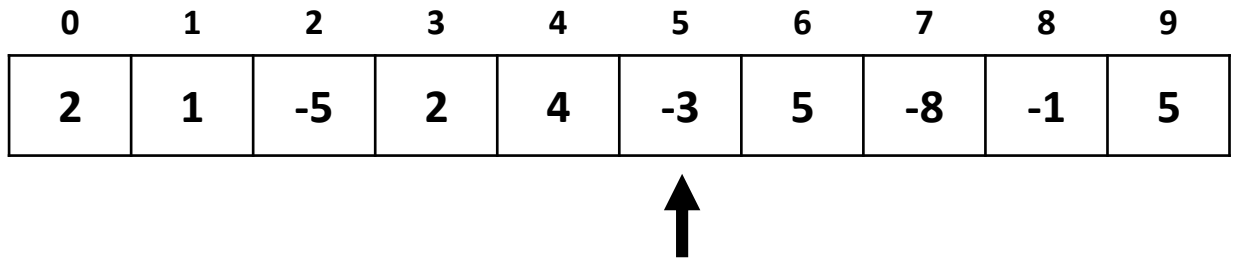
Current Value **4**

Cumulative Sum **6**

Max Sum (Result) **6**

Efficient Maximum Subarray

Original Input



Current Index **5**

Current Value **-3**

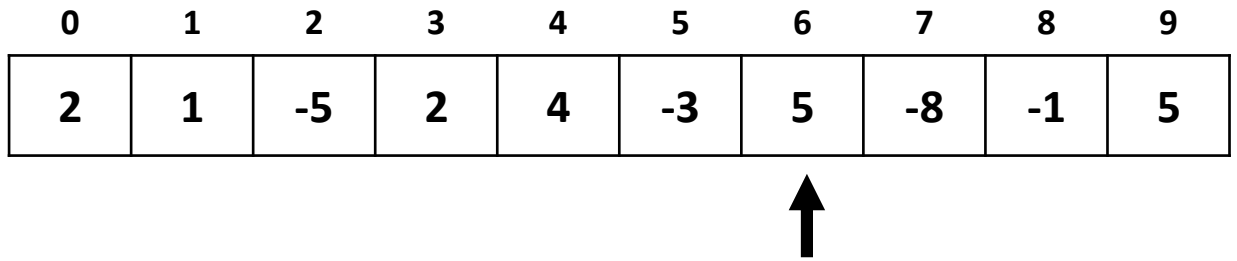
Cumulative Sum **3**

Max Sum (Result) **6**

3 < 6

Efficient Maximum Subarray

Original Input



Current Index **6**

Current Value **5**

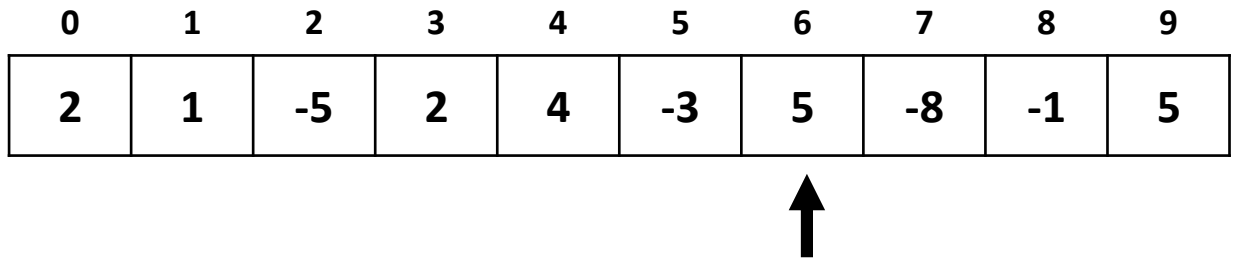
Cumulative Sum **8**

Max Sum (Result) **6**

8 > 6 update!

Efficient Maximum Subarray

Original Input



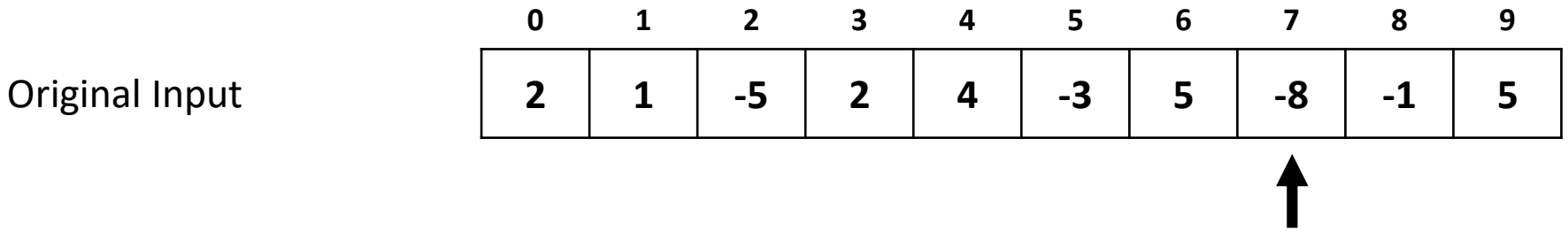
Current Index **6**

Current Value **5**

Cumulative Sum **8**

Max Sum (Result) **8**

Efficient Maximum Subarray



Current Index **7**

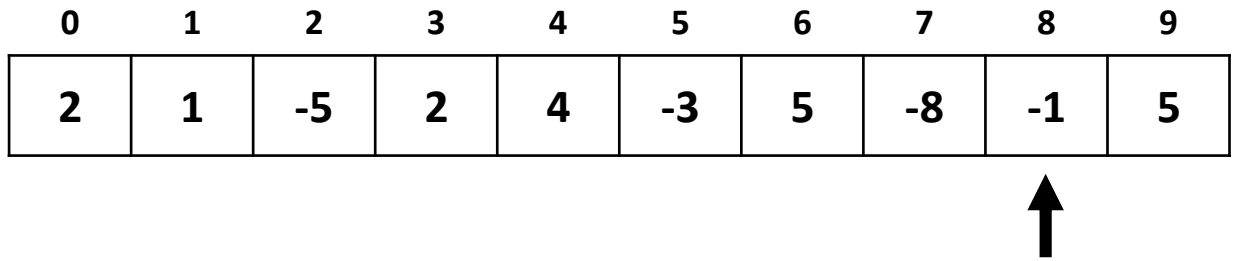
Current Value **-8**

Cumulative Sum **0**

Max Sum (Result) **8**

Efficient Maximum Subarray

Original Input



Current Index **8**

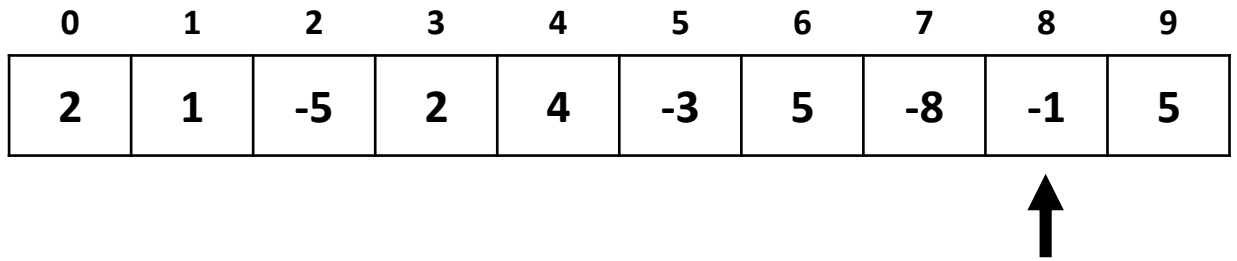
Current Value **-1**

Cumulative Sum **-1** → Negative!
Initialize cumulative sum to 0

Max Sum (Result) **8**

Efficient Maximum Subarray

Original Input



Current Index **8**

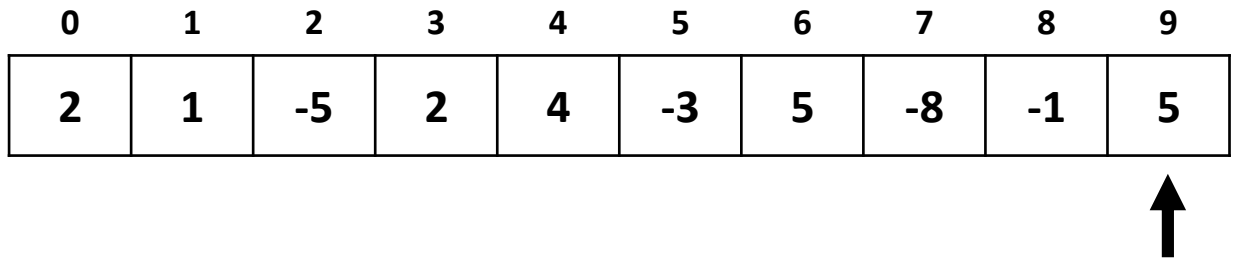
Current Value **-1**

Cumulative Sum **0**

Max Sum (Result) **8**

Efficient Maximum Subarray

Original Input



Current Index **9**

Current Value **5**

Cumulative Sum **5**

Max Sum (Result) **8**

5 < 8

Efficient Maximum Subarray

Original Input

0	1	2	3	4	5	6	7	8	9
2	1	-5	2	4	-3	5	-8	-1	5

Current Index **9**

Current Value **5**

Cumulative Sum **5**

Max Sum (Result) **8**

Finished !

The answer is **8**