ARRAYS

STATIC AND DYNAMIC ARRAYS

* What is an Array?
* When and where is an Array used?
* Complexity
* Static array usage example.
* Dynamic Array implementation details.
* Code Implementation.

* A static array is a fixed length container containing n elements indexable from the range [0, n-1].
* Indexable: This means that each slot/index in the array can be referenced with a number.

Array elements are stored in contiguous memory locations, meaning they are stored in a sequence.

When and where is an Array used?

1) Storing and accessing sequential data.
2) Temporarily storing objects.
3) Used by IO routines as buffers.
4) Lookup tables and inverse lookup tables.
5) Can be used to return multiple values from a function.
6) Used in Dynamic Programming to cache answers to subproblems.

Static-sized Arrays:-

Static-sized arrays have a fixed size, determined at compile time. Once declared, the size of a static array cannot be changed.

Characteristics:-

* Fixed size: The number of elements the array can hold is defined when the array is created and cannot be altered.
* Memory Allocation: Memory for the array is allocated on the stack, making allocation and deallocation fast.
* Performance: Accessing elements in a static array is fast because elements are stored contiguously in memory, enabling efficient indexing.

Dynamic-sized Arrays:-

Dynamic-sized arrays can change size during runtime. They can grow or shrink as needed, offering more flexibility.

Characteristics:-

* Resizable: The array can adjust its size at runtime to accommodate more (or fewer) elements than initially declared.
* Memory Allocation: Memory for dynamic arrays is typically allocated on the heap, which allows them to have a flexible size but also means that memory management (allocation and deallocation) is more complex and slightly slower.
* Efficiency Considerations: While dynamic arrays provide flexibility, resizing operations (like increasing the array's size) may require allocating new memory and copying existing elements to the new location, which can be costly in terms of performance.

Complexity :-

Operations	Static Array	Dynamic Array.
Access	O(1)	O(1)
Insert (at the end)	~~O(n)~~ N/A	O(1)
Insert (at a specific index)	N/A	O(n)
Delete (from the end)	N/A	O(1)
Delete (from a specific index)	N/A	O(n)
Search	O(n)	O(n)
Search (sorted array).	O(log n)	O(log n)
Updating	O(1)	O(1)
Appending.	N/A	O(1).

Properties of Arrays :-

A) Memory allocation :-

B - Base address
I - Index
S - Size of each element

Address A of the element at index I is computed as :

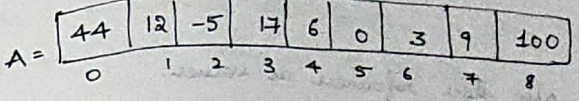
$A = B + (I \times S)$

B) Indexing :-

zero-based indexing.

Allows accessing element in O(1) time

Static Array usage :-



A[0] = 44

A[1] = 12

A[9] => Array index out-of-bounds.

⋮

A[8] = 100

Dynamic Array :-

can grow and shrink in size.

A = 34 | 4

A.add(-7) A = 34 | 4 | -7

A.add(34) A = 34 | 4 | -7 | 34

A.remove(4) A = 34 | 7 | 34

How can we implement a dynamic array?

Ans :- One way is to use a static array.

1) Create a static array with an initial capacity.
2) Add elements to the underlying static array, keeping track of the number of elements.
3) If adding another element will exceed the capacity, then create a new static array with twice the capacity and copy the original elements into it.

Dynamic Array implementation :-

-00-DynamicArray.java.

Two Dimensional Arrays :-

* Introduction :-

Arrays can also store references as values.

These references can point to anything -

- an object (or)
- any other data structure.

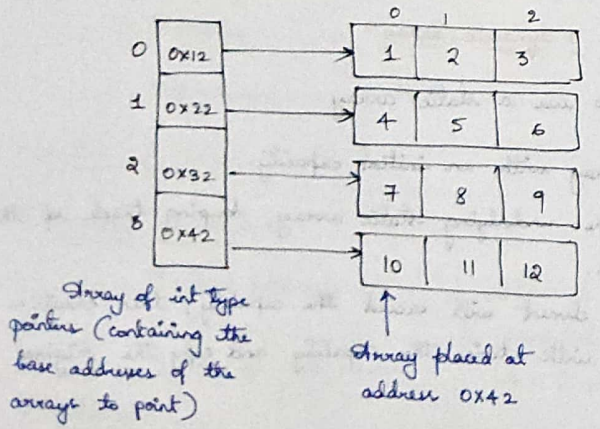
→ References are used to explicitly store memory locations that hold a value or an object.

→ Any time you build an object in java, you basically create a reference to that object.

* Two Dimensional Arrays :-

A two dimensional array is an array of references that holds references to other arrays.

These arrays are preferably used if you want to put together data items in a table or matrix-like structure.



Array of int type pointers (containing the base addresses of the arrays to point)

Array placed at address 0x42

Lists :-

Lists are a fundamental data structure in java that allow us to store and manipulate collections of elements.

* List interface :-

"List" interface — "java.util" package — extends "Collection" interface.

→ Ordered collection :

Elements in a list maintain the order in which they are inserted.

→ Allows duplicates :

Unlike sets, lists can contain duplicate elements.

→ Indexed access :

Elements can be accessed and modified using their index.

→ Dynamic sizing :

Lists automatically resize themselves as elements are added or removed.

```
List<Type> listName = new ArrayList<>();
//ArrayList is an implementation of List.
```

* ArrayList :-

ArrayList is one of the most commonly used implementations of the List interface in java.

It internally uses an array to store elements and provides fast access to elements by index.

* Dynamic sizing * Random access are key features

GCD and LCM :-

The GCD or HCF of two or more integers is the largest positive integer that divides each of the integers without leaving a remainder.

GCD/LCF of 12 and 18 :

Factors of 12: 1, 2, 3, 4, 6, 12

Factors of 18: 1, 2, 3, 6, 9, 18

Common factors are 1, 2, 3 and 6.

The largest among them is 6, so the GCD/HCF of 12 and 18 is 6.

LCM(a,b) * GCD(a,b) = a * b.

Euclid's algorithm :-

$$gcd(a, b) = gcd(b, r)$$

r is remainder when a is divided by b.

(ie) a % b

and keep on doing until b == 0.

```
int GCD (int A, int B) {
    if (B == 0) return A;
    else return GCD(B, A%B);
}
```

Catalan Number :-

Mathematical sequence that can consist of positive integers, which can be used to find the number of possibilities of various combinations.

$$\frac{2n!}{(n+1)!\,n!} \quad \text{or} \quad \frac{1}{n+1}\,{}^{2n}C_n.$$

Examples :-

① no. of expressions with n pairs of parenthesis

$n=3 \Rightarrow$ ()()(), (())(), ((())), (()()), ()(()) = 5.

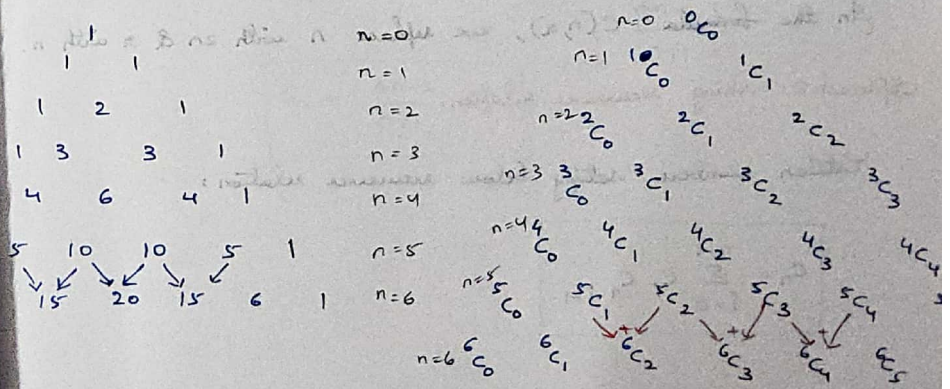
$$\frac{1}{3+1}\,{}^{(2\times3)}C_3 = \frac{1}{4}\,{}^{6}C_3 = \frac{1}{4}\,\frac{6!}{3!\,3!} = \frac{1}{4}\cdot\frac{6\times5\times4}{3\times2} = 5.$$

② no. of BSTs with n nodes.

Explanation :-

Approach 1 : Using Binomial Coefficient

${}^{n}C_r$ can be represented using Binomial Coefficient



$${}^{n}C_r = C(n,r) = C(n-1, r-1) + C(n-1, r)$$

$${}^{n}C_0 = 1 \qquad {}^{n}C_n = 1$$

The above representation is called Pascal's Triangle.

The above formula is to calculate Binomial coefficient $C(n,r)$.

$C(n,r) \Rightarrow$ Coefficient of x^{th} power in $(1+x)^n$.

Eg- $C(3,2)$

$\Rightarrow$ Coefficient of x^2 in $(1+x)^3$.

$(1+x)^3 = 1 + x^3 + 3x^2 + 3x$

ans: $\underline{3}$.

From the pascal's triangle, in row $n=3$, 2^{nd} term $= \underline{3}$.

Since catalan number is given by:

$$\frac{2n C_n}{n+1}$$

we can use Binomial Coefficient formula to find value of $2n C_n$ and divide the result with $(n+1)$.

In the formula $C(n,r)$, we replace n with $2n$ & r with n.

Approach 2: Using recurrence relation.

Catalan numbers satisfy below recurrence relation:

$$C_n = \sum_{i=0}^{n-1} C_i \, C_{n-i-1}$$

Approach 3: Calculate C_n using C_{n-1}

$$C_n = \frac{(2n)!}{(n+1)! \times n!}$$

$$C_{n-1} = \frac{[2(n-1)]!}{n! \times (n-1)!}$$

$$\frac{C_n}{C_{n-1}} = \frac{\frac{(2n)!}{(n+1)!\,n!}}{\frac{(2(n-1))!}{n!\,(n-1)!}} = \frac{(2n)!}{(2n-2)!} \frac{(n-1)!}{(n+1)!}$$

$$= \frac{2n \times (2n-1) \times (2n-2)!}{(2n-2)!} \times \frac{(n-1)!}{(n+1)(n)(n-1)!}$$

$$= \frac{2n \times (2n-1)}{n(n+1)}$$

$$= \frac{4n-2}{n+1}$$

$$\boxed{C_n = \frac{4n-2}{n+1} \cdot C_{n-1}}$$

Binomial Coefficient :-

A binomial coefficient C(n,k) can be defined as the coeffic
of x^k in the expansion of (1+x)^n.

A binomial coefficient C(n,k) also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k-element subsets (& k-combinations) of a n-element set.

Approach 1:- Recursion

$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

$$C(n,0) = C(n,n) = 1$$

Approach 2:- DP

```
C[i][j] = C[i-1][j-1] + C[i-1][j];
```

Approach 3:- Optimization.

```
C[j] = C[j] + C[j-1];
```

$$^nC_r = \frac{n!}{(n-r)!\,r!}$$

$$= \frac{n(n-1)\cdot(n-2)\cdots(n-r+1)\cdot(n-r)\cdots1}{(n-r)!\,r!}$$

$$= \frac{n(n-1)(n-2)\cdots(n-r+1)\cdot(n-r)!}{(n-r)!\,r!}$$

$$= \frac{n(n-1)(n-2)\cdots(n-r+1)}{r(r-1)(r-2)\cdots1}$$

Hashing

{1, 2, 1, 3, 2} Queries: {1, 3, 4, 2, 10}

How many times a number in the Queries appear in the given array?

1	2
3	1
4	0
2	2
10	0

We call count function for each query to count the number.

The count function implements for loop on array items to find the match and counts if there is match.

```
for(i in items)
    if (arr[i] == num)
        cnt++;
```

So the function takes O(N) time, N is size of array.

For each query, there is f^n call, so the total time is

$$O(q * N)$$, q is no. of queries.

Number Hashing:-

Instead of running loop for each query, we can initially run a loop to store the frequency of the number in an array, where the number itself is index of the array.

If maximum number in array is 12, we need to have max index of array as 12, so the array size should be 13.

```
//precompute
int[] hash = new int[13];
for(int i=0; i<n; i++) {
    hash[arr[i]] += 1;
}
int q = scn.nextInt();
while(q-- >0) {
    int number = sc.nextInt();
    print(hash[number]);
}
```

Limitation :-

If the maximum array element is very large like 10^9, this approach is not efficient as we can have array size limitations.

Array Declaration	Maximum size (Integer type)	Maximum size (Boolean type)
Inside main function	10^6	10^7
Globally	10^7	10^8

"Character Hashing" :-

string : "abcdabefc"

queries : {a, c, z}.

a	2
c	2
z	0

Similar to previous approach, for each query, a function call is made which iterates over string to mach the query character and counted.

This takes $O(Q \cdot N)$ time.

Optimised approach:

① Case 1 : 'a' to 'z'

Let's say we need to store 'a' in the string of character allowing 'a' to 'z'.

We need to store 'a' at index 0 & 'z' at index 25.

To identify the index to store a character, we take ASCII value of that character.

ASCII of 'a' is 97

'z' is 122

To make 97 to 0, we substract 97 (ASCII of 'a').

So corresponding index is given by:

given character - 'a'.

② Case 2 : 'A' to 'Z'

Similarly for 'A' to 'Z',

corresponding index is given by.

given character - 'A'.

A - 65

Z - 90

③ Case 3 All lowercase and uppercase

There are totally 256 characters.

0 - NULL
⋮
6 - ACK
⋮
31 - US
32 - sp
33 - !
34 - "
⋮
43 - +
44 - ,
45 - -
46 - .
47 - /
48 - 0
⋮
57 - 9
58 - :
59 - ;
⋮
63 - ?
64 - @
65 - A
⋮
90 - Z
91 - [
⋮
96 - `
97 - a
⋮
122 - z
123 - {
⋮
254 - ¦
255 -

Hash large numbers like 10^9 or higher :-

HashMap

Total Time complexity

= O(N * time taken by map data structure)

Storing & fetching = O(1).

worst case = O(N) because of internal collision.

What is collision & how the hashing works :-

* Hashing Techniques :-

1) Division method : This method is simple and uses the remainder of the key value divided by M as the hash value.

$$Hash(K) = K \% M$$

→ Choose M larger than number ⇒ M is ideally the size of hash table.

→ To minimize collision, M should be prime or number with small divisor.

Eg: ↳ 12345

M = 95

h(12345) = 12345 % 95 = 90.

↳ 1276

M = 11

h(1276) = 1276 % 11 = 0

Pros: very fast as it requires only single division operation.

Cons: poor performance as consecutive keys map to consecutive hash values in the hash table.

2) Mid Square Method :-

① Square the value of the key k i.e; k^2

② Extract the middle 'r' digits as the hash value.

$h(K) = h(K * K)$, K is the key value.

value of 'r' can be decided based on the size of the table.

Eg:

Suppose the hash table has 100 memory locations.

So r = 2 because two digits are required to map the key to the memory location.

$k = 60$

$k \times k = 60 \times 60$

$= 3600$

$h(60) = 60$

The hash value obtained is 60.

Pros :- The result is not dominated by the distribution of the top digit & bottom digit of the original key value.

Cons :- Size of key is one of the limitations.

Collisions.

3) Digit Folding Method :-

① Divide the key-value 'k' into a number of parts i.e, $k_1, k_2, \ldots k_n$ where each part has the same number of digits except for the last part that can have lesser digits than the other parts.

② Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Formula :

$k = k_1, k_2, k_3, k_4, \ldots, k_n$.

$s = k_1 + k_2 + k_3 + k_4 + \ldots + k_n$.

$h(k) = s$.

Here,

's' is obtained by adding the parts of the key 'k'.

Example :

$k = 12345$

$k_1 = 12 \quad k_2 = 34 \quad k_3 = 5$

$s = k_1 + k_2 + k_3$

$= 12 + 34 + 5$

$= 51$

$h(k) = 51$.

Note: The no. of digits in each part varies depending upon the size of the hash table.

Suppose for example the size of the hash table is 100, then each part must have two digits except for the last part which can have lesser number of digits.

④ Multiplication Method :-

① Choose a constant value A such that $0 < A < 1$
② Multiply the key value with A.
③ Extract the fractional part of kA.
④ Multiply the result of the above step by the size of the hash table i.e., M.
⑤ Resulting hash value is obtained by taking the floor of the result obtained in step ④.

Formula :-

$h(k) = \text{floor}(M(kA \bmod 1))$

M - size of the hash table.
k - key value.
A - constant value.

Example :-

$k = 12345$

$A = 0.357840$

$M = 100$

$h(12345) = \text{floor}[100(12345 \times 0.357840 \bmod 1)] =$

$= \text{floor}[100(4417.5348 \bmod 1)]$

$= \text{floor}[100(0.5348)]$

$= \text{floor}[53.48]$

$= 53.$

Commonly used hash functions :-

① SHA (Secure Hash Algorithm) :

SHA is a family of cryptographic hash functions designed by the National Security Agency (NSA) in the United States. SHA-1, SHA-2, SHA-3 are most widely used algorithms.

* SHA-1 : → 160-bit hash function
→ used for digital signatures and others
→ no longer secure due to known vulnerabilities.

* SHA-2 : → family of hash functions SHA-224, SHA-256, SHA-384, SHA-512.
→ These functions produce hash values of 224, 256, 384 and 512 bits respectively.
→ used in security protocols such as SSL/TLS and is considered secure.

* SHA-3 : → SHA-3 is the latest member of the SHA family
→ designed to be faster and more secure than SHA-2
→ produces hash values of 224, 256, 384, 512 bits.

② CRC (Cyclic Redundancy Check). for error detection in data transmission. It is non-cryptographic.

③ MurmurHash. non-crypto

④ BLAKE2. crypto.

⑤ Argon2. memory-hard password hashing function.

⑥ MD5 : (Message Digest 5) crypto.

Question: What if two or more array elements ... remainder?

array: { 2, 5, 16, 88, 139, 38, 48, 28, 18 }.

M = 10.

0 —
1
2 — 2
3
4
5 — 5
6 — 16
7
8 — 28 → 38 → 48 → 28 → 18. → collision (LINEAR CHAINING)
9 — 139

If "all" the elements go to same hash index, the worst case happens & we also call it "collision"

Collision Resolution :-

* Linear Probing
* Chaining.

* **Linear Probing :-**

Empty space is searched using Linear Search.

$H(k) = k \pmod M$

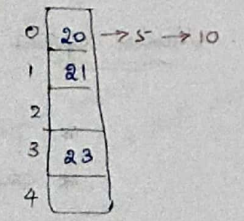
23, 21, 20, 5, 10. M=5

$H(23) = 23 \% 5 = 3$
$H(21) = 21 \% 5 = 1$
$H(20) = 20 \% 5 = 0$
$H(5) = 5 \% 5 = 0$ (collision) = 2
$H(10) = 10 \% 5 = 0$ (collision) = 4

0	20
1	21
2	5
3	23
4	10

* **Chaining :-**

Keeps with same hash address are linked together in the form of Linked List.



23, 21, 20, 5, 10.

$H(23) = 23 \pmod 5 = 3$
$H(21) = 21 \pmod 5 = 1$
$H(20) = 20 \pmod 5 = 0$
$H(5) = 5 \pmod 5 = 0$ collision
$H(10) = 10 \pmod 5 = 0$ collision

BINARY SEARCH

Number Systems :-

* Decimal To Any Base.
* Any Base To Decimal.
* Any Base To Any Base.
* Any Base Addition.
* Any Base Subtraction.
* Any Base Multiplication.

① Decimal to Any Base :- * $(248)_{10} \rightarrow (\ )_2$

Base : 2

Base
2 | 248
2 | 124 — 0 → remainder.
2 | 62 — 0 → quotient becomes dividend
2 | 31 — 0
2 | 15 — 1
2 | 7 — 1
2 | 3 — 1
2 | 1 — 1
0 — 1

$(11111000)_2$

2) 124 (62
124
0

2) 62 (31
62
0

divisor = Base = 2.

divd = 248, quotient = 124, rem = 0. — 0×10^0
divd = 124, quotient = 62, rem = 0 — 0×10^1
divd = 62, quotient = 31, rem = 0 — 0×10^2
divd = 31, quotient = 15, rem = 1 — 1×10^3
divd = 15, quotient = 7, rem = 1 — 1×10^4
divd = 7, quotient = 3, rem = 1 — 1×10^5
divd = 3, quotient = 1, rem = 1 — 1×10^6

0 +
0 +
0 +
1000 +
10000 +
100000 +
1000000 +
10000000

11111000 → answer.

Program :-

```
int base = 2; int ans = 0
int fact = 0;
while (divd > 0) {
int rem = divd % base;
ans += (int) [rem * Math.pow(10, fact)];
divd /= base;
fact++;
}
```

* Decimal — 0 to 9.
  Octal — 0 to 7
  Binary — 0 & 1
  ⋮

* $(764)_{10} \rightarrow (\ )_8$

8 | 764
8 | 95 — 4
8 | 11 — 7
8 | 1 — 3
0 — 1

$(1374)_8$

base	divd	quotient	rem	val
8	764	95	4	$4 \times 10^0 = 4$
8	95	11	7	$7 \times 10^1 = 70$
8	11	1	3	$3 \times 10^2 = 300$
8	1	0	1	$1 \times 10^3 = 1000$
				1374

⑧ Any Base to Decimal :-

$(11111000)_2 \rightarrow (?)_{10}$

$2^7\ 2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

↓

$1\times128 + 1\times64 + 1\times32 + 1\times16 + 1\times8 + 0\times4 + 0\times2 + 0\times1$

$128 + 64 + 32 + 16 + 8$

$= 248$

$(248)_{10}$ answer

Program :-

```
int base = 2; int num = 11111000;
int mul = 1;
int temp = num; int ans = 0;
while (temp > 0) {
    int rem = temp % 10;
    ans += (rem * mul);
    mul *= base;
    temp /= 10;
}
```

⑨ Any Base to Any Base :-

$(1374)_8 \rightarrow (\ )_2$

Step ①: any base to decimal

Step ②: decimal to any base.

① $(1374)_8 \rightarrow (?)_{10}$

$8^3\ 8^2\ 8^1\ 8^0$

$1\times512 + 3\times64 + 7\times8 + 4\times1$

$= 512 + 192 + 56 + 4$

$= (764)_{10}$

② $(764)_{10} \rightarrow (\ )_2$

2 | 764		
2 | 382	−0	
2 | 191	−0	
2 | 95	−1	
2 | 47	−1	
2 | 23	−1	
2 | 11	−1	
2 | 5	−1	
2 | 2	−1	
2 | 1	−0	
	0	−1

$(1011111100)_2$

Answer :- $(1011111100)_2$.

④ Any Base Addition :-

(1) (1) (1) → que/carry

3 4 6 (8)

7 7 7 (8)

1 3 4 5 → rem

7+6

= 13

= 8 + 5 → rem.

·1 → que

7+4+1

= 12

= 8·1 + 4

7+3+1

= 11

= 8·1 + 3

8 | 13
8 | 1 — 5
0 — 1

(15)$_8$

8 | 12
8 | 1 — 4
0 — 1

1 1 1

2 3 4 $_8$

3 4 3 $_8$

1 1 3 2

Program :-

```
int base = 8; num1 = 346, num2 = 777;
int carry = 0;   int res = 0;
int mul = 1;
while( num1 > 0 || num2 > 0 || carry > 0) {
    int dig1 = num1 % 10;
    int dig2 = num2 % 10;
    int sumOfDigs = dig1 + dig2 + carry;
    int rem = sumOfDigs % base;
    carry = sumOfDigs / base;
    res += (rem * mul);
    mul *= 10;
    num1 /= 10;
    num2 /= 10;
}
```

⑤ Any Base Subtraction :-

base = 8

(-1) (-1) (+1) +8 ✓ ← borrow

0 1 2 1 2 10

1 0 8

2 5 6 ✓

0 7 3 4

(+8) 2 − 6 ✗ borrow

10 − 6 = 4 × 10^0

(+8) 0 − 5 ✗ borrow.

8 − 5 = 3 × 10^1

(+8) 1 − 2 ✗ borrow

9 − 2 = 7 × 10^2

0 × 10^3

Program :-

```
int base = 8, num1 = 1212, num2 = 256;
int ans = 0;
int borr = 0;
int mul = 1;
while (borr > 0 || num1 > 0 || num2 > 0) {
    int dig1 = num1 % 10;
    int dig2 = num2 % 10;
    int diff = dig1 - dig2 - borr;
    if (diff < 0) {
        borr = 1;
        diff += 8;
    }
    ans += (diff * mul);
    mul *= 10;
    num1 /= 10;
    num2 /= 10;
}
```

⑥ By base multiplication :-

(2) (2) (3) → quo/carry
(1) (2) (3)
2 3 4

base = 8

7 6

1 6 5 0 (24) → can't be greater than 7.
(1) ↳ rem

(+) 2 1 0 4 X

2 2 7 1 0

↓

5 + 4 = 9 = 8 · 1 + 1 (quo, rem)

4 × 6 = 24 = 8 · 3 + 0 (carry/quo, rem)

6 × 3 + 3 = 21 = 8 · 2 + 5

6 × 2 + 2 = 14 = 8 · 1 + 6

7 × 4 = 28 = 8 · 3 + 4

7 × 3 + 3 = 24 = 8 · 3 + 0

7 × 2 + 3 = 17 = 8 · 2 + 1

Program :-

```
int base = 8, num1 = 234, num2 = 76;
int ans = 0, mul = 1;
while (num2 > 0) {
    int dig = num2 % 10;
    int productWithDigit = getProductWithDigit(num1, dig, base);
    ans = getSum(base, multiplier * productWithDigit, ans);
    num2 /= 10;
    multiplier *= 10;
}
```

```
getProductWithDigit (num1, dig, base) {
    int carry = 0, ans = 0, mul = 1;
    while (num1 > 0 || carry > 0) {
        int dig1 = num1 % 10;
        int prod = dig1 * dig + carry;
        carry = prod / base;
        ans += [mul * (prod % base)];
        mul *= 10;
        num1 /= 10;
    }
    return ans;
}

getSum (base, num1, num2) {
    int carry = 0, ans = 0, multiplier = 1;
    while (num1 > 0 || num2 > 0 || carry > 0) {
        int dig1 = num1 % 10;
        int dig2 = num2 % 10;
        int sum = dig1 + dig2 + carry;
        carry = sum / base;
        ans += [mul * (sum % base)];
        mul *= 10;
        num1 /= 10;
        num2 /= 10;
    }
    return ans;
}
```

Problem: SUM OF TWO ARRAYS

* 5
3 1 0 7 5 → arr1

6
1 1 1 1 1 1 → arr2.

o/p :- 1 4 2 1 8 6 → o/p array.

The output array size can be max (size of arr1, size of arr2) & max (size of arr1, size of arr2) + 1

9 9 9 9 9 — size = 5
3 3 2 1 — size = 4
1 0 3 3 2 0 — size = 6.

refer (_11_SumOfArrays.java) in GitHub.

Problem: Difference of Two Arrays

case 1:
arr1 → [2, 1]
arr2 → [2]
res → [1, 9]
print res
⇒ 19

case 2:
arr1 → [2]
arr2 → [2, 1]
res → [1, 9]
print -res
⇒ -19

In case 1: There is no borrow after subtraction of individual digits.

In case 2: diff = (arr1 - arr2)
= (i) 2 - 1 = 1
(ii) 0 - 2 = -2 ⇒ -2 + 10 = 8
∴ borrow = -1
ans: -81

(1) ← borrow
(10) 2
(-) 2 1
8 1

If borrow = -1, then the result need to be computed again as -81 is not expected result.

∴ If borrow = -1, then swap arr1 & arr2 positions and compute diff again, and append "-".

diff = arr2 - arr1
= [2, 1] - [2]
= [2, 1] - [0, 2]
= [1, 9]

∴ print -19.

refer (-12_DifferenceOfArrays2.java) in GitHub.

Problem: Rotate An Array (refer -14_RotateAnArray2.java)

(i) rotate: k = 2 [1, 2, 3, 4]
[2, 1, 4, 3]
↓ reverse
[3, 4, 1, 2]

reverse:
1, 2, 3, 4
ans: 4, 3, 2, 1

ans: [3, 4, 1, 2]

(ii) rotate: k = 2 [1, 4, 8, 12, 16, 20]

ans: [16, 20, 1, 4, 8, 12]

[12, 8, 4, 1, 20, 16]
↓ reverse
[16, 20, 1, 4, 8, 12]

From above two examples, to rotate an array by k positions by reversing, follow steps below:

1. reverse elements from positions 0 → n-k-1
2. reverse elements from positions n-k → n-1
3. reverse entire array 0 → n-1.

Problem

Print Subsets

[10, 20, 30].

- - -	0 0 0
- - 30	0 0 1
- 20 -	0 1 0
- 20 30	0 1 1
10 - -	1 0 0
10 - 30	1 0 1
10 20 -	1 1 0
10 20 30.	1 1 1

We can use n bits to manipulate the o/p.

There are 2^n subsets for n elements.

We will place bit 1 with the corresponding number at its position.

Suppose, in 0 0 1, (② ① ⓪)

in 2^{nd} position, we put 2^{nd} element (30)

in 1 1 0,

in 0^{th} & 1^{st} positions, we put 0^{th} & 1^{st} elements (10 20)

and in position 0, we simply put "-".

Point 1 :-

②	①	⓪		
0	0	0	— 0	
0	0	1	— 1	✓
0	1	0	— 2	
0	1	1	— 3	✓
1	0	0	— 4	
1	0	1	— 5	✓
1	1	0	— 6	
1	1	1	— 7	✓

observation:
all odd numbers have 1 in its last position. (2^{nd} pos).

Point ②:

0	0	0	— 0
0	0	1	— 1
0	1	0	— 2
0	1	1	— 3
1	0	0	— 4
1	0	1	— 5
1	1	0	— 6
1	1	1	— 7

/2 (divide by 2)

0	— 0 0
0	— 0 0
1	— 0 1
1	— 0 1
2	— 1 0
2	— 1 0
3	— 1 1
3	— 1 1

observation: dividing the numbers by 2 removes last bit.

Combining ① & ②, we can easily process positions in descending order (2^{nd}, 1^{st}, 0^{th}).

① if number is odd, then last position (2^{nd}) is 1, so replace with arr[2] = 30.

② Since last (2^{nd}) position is processed, divide the number by 2, to remove the processed bit and again repeat step ①.

③ Repeat for all k^{th} elements k: 0 → n.

Problem: Product of Array except self (LC 238 Med)

i/p: [1, 2, 3, 4] (0 1 2 3)

without using division

o/p: [24, 12, 8, 6].

product fits in 32-bit integer.

Approach:-

running product ⟶ prefix product

[1, 2, 6, 24] → [1, 1, 2, 6]

running product ⟵ suffix product.

[24, 24, 12, 4] → [24, 12, 4, 1]

say index 2, in [1, 2, 3, 4].

prefix prod = 2, suffix prod = 4

2 * 4 = 8.

Hence,

for index 2, take prefixprod[2] * suffixprod[2].

~~In case of 0th index, only take suffixprod[1]~~
~~In case of 3rd index, only take prefixprod[2].~~

[1, 1, 2, 6]
*
[24, 12, 4, 1]

∴ o/p : [24, 12, 8, 6]

Approach 2:- optimise space.

i/p: [1, 2, 3, 4]

ans: [1, 1, 2, 6] ⟶ left = [1, 1, 2, 6]

[1, 1, 2, 6] ⟵ * right = [24, 12, 4, 1]

ans: [24, 24, 12, 6]

[24, 12, 8, 6]

take product from left to right

& then from right to left. using explanation below.

APPROACH2 IS CALLED "BIDIRECTIONAL PRODUCT ACCUMULATION"

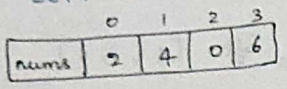
Explanation:-

The algorithm first starts by traversing the array from left to right, accumulating the product of all encountered numbers up to the current index while keeping track of the left product.

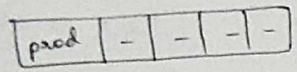
It then iterates through the array from right to left, computing the product of all elements to the right of the current index. At each index, this product is multiplied by the accumulated left product till that index to obtain the final result.

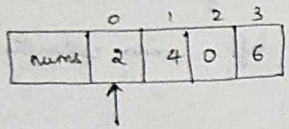
Illustration :-

MOVING LEFT TO RIGHT →

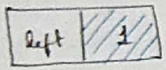


initialize empty "product" array to store the cumulative product





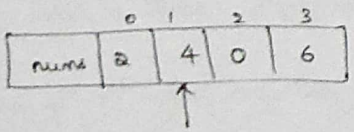
Compute the product of all left elements of "nums"



Since, there are no elements to left of nums[0], "left" is set to 1, and added to "prod" array.

"left" is updated by multiplying it with current number nums[0]

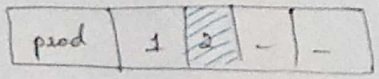
left = left * nums[0]
= left * 2
= 1 * 2
= 2.



"left" has been updated to 2.

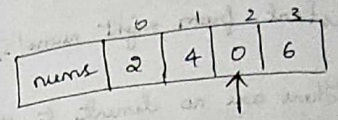
For the next element "nums[1]", this "left" = 2 holds product of all left elements of nums[1].

So, "left" is added to "prod" array

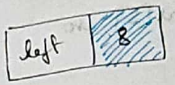


"left" is then updated by multiplying with current number nums[1]

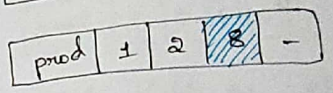
left = left * nums[1]
= 2 * 4
= 8.



"left" has been updated to 8.
Now, "left" holds product of all left elements of nums[2].

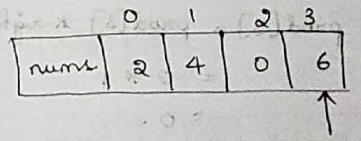


So, "left" is added to "prod" array.

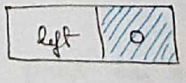


"left" is then updated by multiplying with current number nums[2]

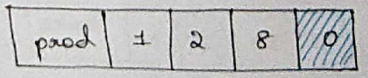
left = left * nums[2]
= 8 * 0
= 0



"left" has been updated to 0.
Now, "left" holds product of all left elements of nums[3].



So, "left" is added to "prod" array



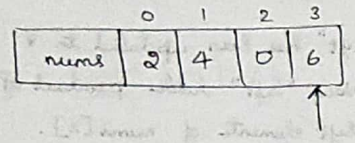
"left" is updated by multiplying with current number nums[3]

left = left * nums[3]
= 0 * 6
= 0

The product calculation for the left elements concludes as there are no more elements remaining.

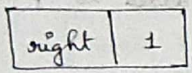
MOVING RIGHT TO LEFT ←

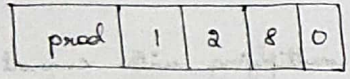
*



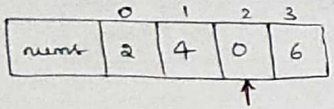
start from right nums[3].

There are no elements to the right of nums[3], so right = 1



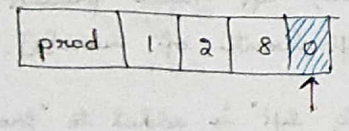


*

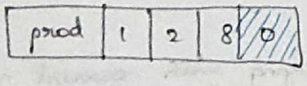


Multiply current element prod[3] with the product of all elements to the right.

⇒ prod[3] = prod[3] * right
= 0 * 1
= 0.



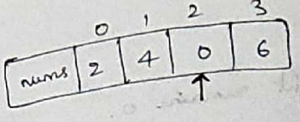
* Prods has been updated to 0.



Update right by multiplying current element nums[3].

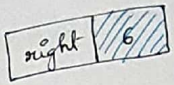
right = right * nums[3]
= 1 * 6
= 6.

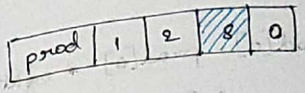
*



right has been updated to 6.

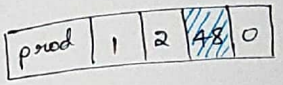
Next multiply prod[2] by "right" to compute prod[2]





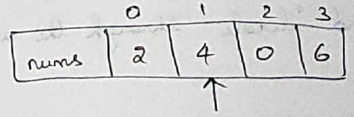
prod[2] = prod[2] * right
= 8 * 6
= 48.

prod has been updated to 48.



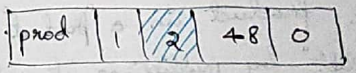
Update right as right = right * nums[2]
= 6 * 0
= 0.

*

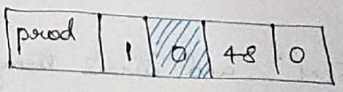


right has been updated to 0.

prod[1] = prod[1] * right
= 4 * 0
= 0.

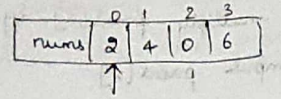


prod has been updated to 0.

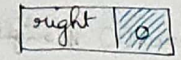


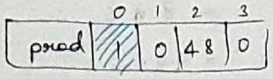
Update right as right = right * nums[1]
= 0 * 4
= 0.

*



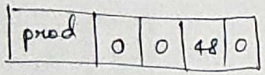
right remains 0.





prod[0] = prod[0] * right
= 0.

prod is updated with 0.



* Algorithm:-

1. Left product calculation:
   (i) We start with "left = 1" and iterate through the array from left to right.
   (ii) At each step, we accumulate the product of all encountered numbers upto the current index
   (iii) We store this cumulative product in a "product" array.
   (iv) Simultaneously, we update "left" by multiplying it with the current element in the array.
2. Right product calculation:
   (i) Similarly, start with "right = 1" and iterate through the array from right to left.
   (ii) We update the elements in the product array by multiplying them with "right", which represents the product of all elements to the right of the current index.
   (iii) Simultaneously, "right" is updated by multiplying it with the current element in the array.

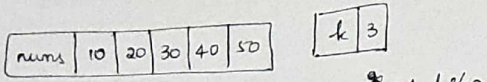
By completing both passes, each element in the product array represents the product of all elements except the one at the corresponding index in the input array.

Problem: Rotate Array

Slice Shift Rotation:-

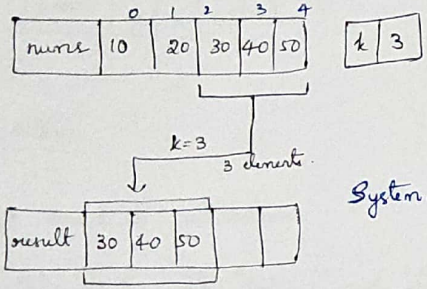
rotations : k

(i)



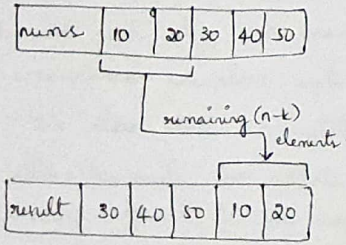
k = ~~k~~ k%n
= 3

(ii)



System.arraycopy(nums, n-k, result, 0, k);

(iii)



System.arraycopy(nums, 0, result, k, n-k);

refer - 14 - RotateArray.java, method "rotate3"

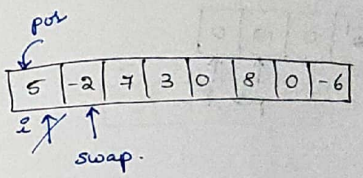
Problem: Rearrange Positive & Negative Values

Can use sorting techniques (revisit later). for in order maintenance.

Make all negative elements appear on the left and positive elements on the right.

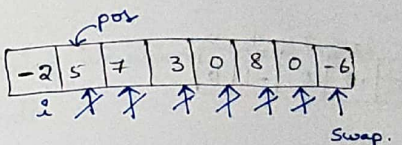
Maintaining original sorted order of the input array is not required for this task.

Eg:-

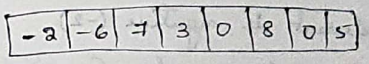


pos: tracks first positive position.

i: searches for negative integer.



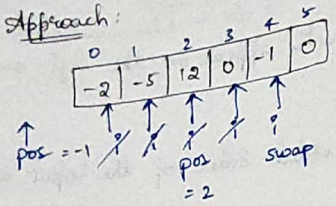
when negative found,
swap (arr[pos], arr[i])
then pos = pos+1.

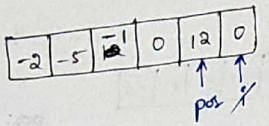


Program:-

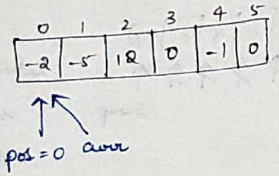
```
int pos = -1;
for (int i = 0; i < arr.length; i++) {
    if (pos < 0 && arr[i] >= 0) {
        pos = i;
    } else if (arr[i] < 0 && pos >= 0) {
        int temp = arr[i];
        arr[i] = arr[pos];
        arr[pos] = temp;
        pos = pos + 1;
    }
}
```

Eg:- Approach:

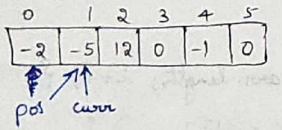




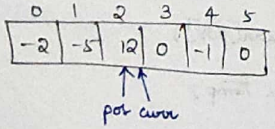
Approach 2:



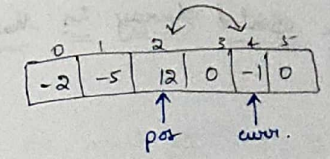
For curr = 0, element is negative (-2).
It's more recent negative number (curr = pos).
So don't swap. pos++



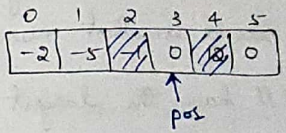
For curr = 1, element is negative (-5)
It's more recent negative number (curr = pos)
So don't swap. pos++



arr[curr] is positive, continue until negative is found.



For curr = 4, element is negative (-1).
It's not last negative number (curr ≠ pos), then
swap arr[curr] & arr[pos], pos++



Remaining iterations don't find any negative number to return result.

Program:

```
int pos = 0;
for (int curr = 0; curr < arr.length; curr++) {
    if (arr[curr] < 0) {
        if (curr != pos) {
            int temp = arr[curr];
            arr[curr] = arr[pos];
            arr[pos] = temp;
        }
        pos++;
    }
}
```

Problem: Rearrange Sorted Array in Max/Min Form

Statement

We're given a sorted array, "nums", containing positive integers only. We have to rearrange the array so that when returned, the 0th index of the array will have the largest number, the 1st index will have the smallest number, the 2nd index will have the second largest number, the 3rd index will have the second smallest number, and so on.

In the end, we'll have the largest remaining numbers in descending order and the smallest in ascending order at even and odd positions, respectively.

Constraints:

$0 \leq nums.length \leq 10^3$

$1 \leq nums[i] \leq 10^5$

Eg:-

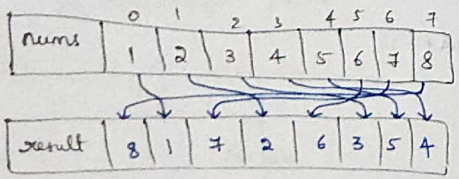
i/p:

nums	1	2	3	4	5	6	7	8

o/p:

nums	8	1	7	2	6	3	5	4

Approach 1: Create a new array.



Two pointers, p1 & p2 pointing at largest & smallest.
add element at p1 & then add element at p2

p1 --
p2 ++

until p1 & p2 meet at mid point.

Program:

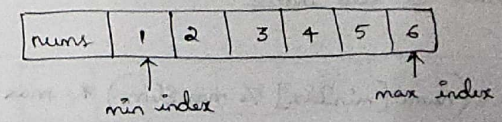
```
for(int i = 0; i < mid; i++) {
    p1 = n-1-i, p2 = i;
    result[index++] = nums[p1];
    result[index++] = nums[p2];
}
```

Approach 2: Use (%) and two pointers (without creating new array).

In this approach, we can store two elements at one index by utilizing the properties of the modulus operator.

We initialize two pointers and an additional variable as shown below:

* minIdx: Initialized to the start of "nums", representing the minimum values.
* maxIdx: Initialized to the end of "nums", representing the maximum values.
* maxElem: Initialized with one element greater than the maximum value in "nums", ensuring it is larger than any element currently in the array.



maxElem	6+1 = 7

This approach involves two phases:
encoding and decoding.

Phase 1: Encoding:

The maxElem value is larger than any other element in the array, allowing us to uniquely encode additional information in the array's indexes without loss.

If we take the modulus of the element at index 0 with 7, we get the same element back, i.e., 1.

Encoding values at even indexes:

$$nums[i] += (nums[maxIdx] \% maxElem) * maxElem \quad \text{— (1)}$$

$\Rightarrow$ nums[0] += (nums[5] % 7) * 7

+= (6 % 7) * 7

+= 42

nums[0] = 1 + 42 = 43.

43 is stored in nums[0].

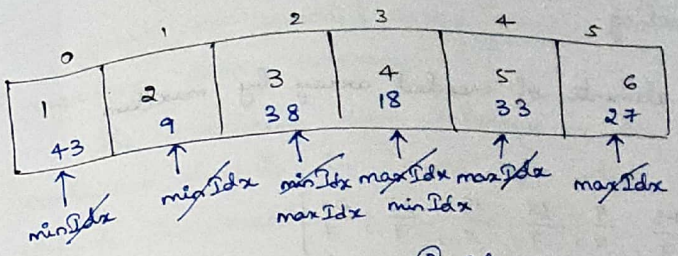
Encoding values at odd indexes:

$$nums[i] += (nums[minIdx] \% maxElem) * maxElem$$

$\Rightarrow$ nums[1] += (nums[2] % 7) * 7

nums[1] = 9

9 is stored in nums[1]



maxElem = 7

① idx = 0

1 + (6 % 7) * 7

= 1 + 6 * 7

= 43

② idx = 1

2 + (43 % 7) * 7

= 2 + 1 * 7

= 9

③ idx = 2

3 + (5 % 7) * 7

= 3 + 5 * 7

= 38

④ idx = 3

4 + (9 % 7) * 7

= 4 + 2 * 7

= 18

⑤ idx = 4

5 + (18 % 7) * 7

= 5 + 4 * 7

= 33

⑥ idx = 5

6 + (38 % 7) * 7

= 6 + 3 * 7

= 27

nums:

0	1	2	3	4	5
43	9	38	18	33	27

Phase 2: Decoding

Divide all elements of encoded array by maxElem.

$$\left[\frac{43}{7}, \frac{9}{7}, \frac{38}{7}, \frac{18}{7}, \frac{33}{7}, \frac{27}{7}\right]$$

$$[6, 1, 5, 2, 4, 3]$$

Note: This approach only works for non-negative numbers.

Problem: Missing Number

{3, 0, 1} → ans: 2 ⟶ for n = 3, elements should be {0, 1, 2, 3}
{0, 1} → ans: 2
{0, 1, 2, 3} → ans: 4. So missing is 2.

App ①: Bit Manipulation :-

wkt $0 \wedge 0 = 0$
$1 \wedge 1 = 0$.
$a \wedge a = 0$
$b \wedge a \wedge a = b$.

while i : 0 → n-1
val: 0 → n-1
int res = n;
if res ^ i ^ val ≠ 0,
then we get missing number.

App ②: Using sum :-

for 0 to n elements, sum = $\frac{n(n+1)}{2}$

```
for(int i=0; i<n; i++) {
    sum -= nums[i];
}
return sum;
```

App ③: Binary Search :- sort elements & perform B.S.

(ref: -30_LC268_Missing Number.java)

Challenge: MAXARRAY

Given an unsorted array "nums", find the sum of the maximum sum subarray.

The maximum sum subarray is an array of contiguous elements in "nums" for which the sum of the elements is maximum.

Constraints:

$1 \leq nums.length \leq 10^3$

$-10^4 \leq nums[i] \leq 10^4$

Eg:- ① Input

nums	-1	10	8

Output

nums	-1	10	8

10 + 8 = 18.

② Input

nums	-2	8	5	10	-9

Output 8 + 5 + 10 = 23.

nums	-2

③ Input

nums	3	7	-8	-1	-2	11	-2

Output: 11.

The maximum subarray sum problem inherently involves examining various subarrays to check if their sum is maximized, and the most convenient and efficient way to do this is using dynamic programming.

The key idea is to efficiently find the maximum subarray ending at any position based on the maximum subarray ending at the previous position.

Kadane's algorithm:

It uses the bottom-up approach of dynamic programming to solve subproblems iteratively, starting from the smallest subproblems and building up toward the larger problem.

The subproblem here is to find the maximum subarray sum that ends at a specific index i. We need to calculate this for every index "i" in the array.

The base case is the first element of the array, where both the current subarray sum and maximum subarray sum are initialized with the first element's value.

We reuse the previously computed maximum subarray sum at each step to find the solution for the current subproblem.

Algorithm -

Step ① Initialize a "currMax" variable to keep track of the maximum sum of the current array index and another "globalMax" variable to keep track of the largest sum seen so far. Both variables will be initialized with the first element of "nums".

Step ② Traverse the array from the second element until the end of the array is reached.

Step ③ While traversing, if "currMax" is less than 0, assign it the element at the current index. Otherwise, add the element at the current index to "currMax".

Step ④ If "globalMax" < "currMax", then "globalMax" = "currMax"

Eg:- -2 8 5 10 -9

currMax -2
globalMax -2

-2 ↓8 5 10 -9
currMax 8
globalMax 8

-2 8 ↓5 10 -9
currMax 8+5 = 13
globalMax 13

-2 8 5 ↓10 -9
currMax 13+10 = 23
globalMax 23

-2 8 5 10 ↓-9
currMax 23-9 = 14
globalMax = 23.

output = 23