ARRAY

Arrays

Characteristics

- **Memory layout**: hold values in a **contiguous** block of memory.

- **Fixed Size**: the size of an array is defined when it is created and cannot be changed. However, high-level languages have different implementations, making it dynamic.

- **Homogeneous elements:** all elements are of the same data type (int, float, char…)

- **Efficiency**: accessing elements by index is very efficient *O(1)*, since each index maps directly to a memory location. Also, range scans benefit from CPU cache lines since arrays are stored in contiguous blocks of memory.

Arrays – Kadane's algorithm

- Kadane's algorithm is a dynamic programming algorithm to solve **maximum subarray sum**

- At every **index i**:

  start a new subarray at **i**

  extend the previous subarray to include **array[i]**

- **Algorithm**

  1. Initialize:

```
    int maxSoFar = array[0];

    int maxEndingHere = array[0];
```

  2. Loop through the array

```
    for (int i = 1; i < array.size(); ++i) {

        maxEndingHere = max(array[i], maxEndingHere + array[i]);

        maxSoFar = max(maxSoFar, maxEndingHere);

    }
```

  3. Return maxSoFar;

Problem – 217. Contains Duplicate

leetcode.com/problems/contains-duplicate

Problem

- You are given an array of numbers

- Return any value that appears at least twice

Solution – Contains Duplicate

LeetCode leetcode.com/problems/contains-duplicate

Solution

- Loop through the array

- Check if the value is in a hash table

- Return **true** if the value exist

- The problem requires at least twice, but one modification may be having a specific count

Code – Contains Duplicate

LeetCode leetcode.com/problems/contains-duplicate/

Code   Time: **O(n)**   Space: **O(n)**

```cpp
bool containsDuplicate(vector<int>& nums) {
    unordered_map<int, int> seen;
    for (int i = 0; i < nums.size(); ++i) {
        seen[nums[i]]++;
        if (seen[nums[i]] == 2) {
            return true;
        }
    }
    return false;
}
```

LeetCode leetcode.com/problems/two-sum

Problem

- Given an **array** of numbers and a **target**, example: **array [2,7,11,15]** and **target 9**

- Return indices of two numbers where they add up to **target**

- **Output**: [0,1]

  ```
  array[0] + array[1] = 2 + 7 = 9
  ```

Solution – Two Sum

LeetCode leetcode.com/problems/two-sum

Solution

- Iterative over each number in the array

- Calculate the difference between target and each number, example:

  `array[0] = 2,` `target 9,` `then 9 – 2 = 7`

- Now we know we need the number **7** to sum up to **9**

- Check in a *hashmap* if we have 7 in some part of the array

  `hash[7] exists?`

- If yes, return the current index and the index of 7

- If not, store the index of the current number in the hashmap for future evaluation

  `hash[2] = 0`

Code – Two Sum

LeetCode leetcode.com/problems/two-sum

Code Time: **O(n)**   Space: **O(n)**

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    std::unordered_map<int, int> numMap;
    // n being the size of nums
    for (int i = 0; i < nums.size(); i++) {
        // current number of the array
        int number = nums[i];
        int diff = target - number;

        // check if the difference is in some part of the array
        // by using a hashmap
        if (numMap.find(diff) != numMap.end()) {
            return { numMap[diff], i};
        }

        // register the current number index
        numMap[number] = i;
    }
    // no matches
    return {};
}
```

Problem – 238. Product of Array Except Self

leetcode.com/problems/product-of-array-except-self

Problem Statement

- …

Solution

- ...

Code – 238. Product of Array Except Self

 leetcode.com/problems/product-of-array-except-self

Code   Time: O(n)   Space: O(1)

- ...

Problem Statement / Solution / Code   Time: O(-)   Space: O(-)

- …

Problem – 152. Maximum Product Subarray

Medium

Problem Statement / Solution / Code   Time: **O()**   Space: **O()**

- ...

 leetcode.com/problems/find-minimum-in-rotated-sorted-array

Problem Statement / Solution / Code   Time: **O(n)**   Space: **O(n)**

- ...

Problem Statement

- You are given an integer **array** of stock prices

- Choose a **price[i]** to buy and **price[i]** to sell where you achieve maximum profits

- **Example:**

  prices = [9, 1, 3, 4]

- **Output**: [1,3]

  array[3] - array[1] = 4 + 1 = 3

Solution - Best Time to Buy and Sell Stock

LeetCode leetcode.com/problems/best-time-to-buy-and-sell-stock

Solution

- Initialize **profit = 0**

- Initialize **lowestBuyPrice = prices[0]**

- Loop through the prices

- Track the lowest buy price → **min(lowestBuyPrice, prices[i])**

- Check if selling "today" will make the maximum profit and update profit:

  max(prices[i] – buy > profit, profit)

- Update profit

  max(prices[i] - buy

Code - Best Time to Buy and Sell Stock

 leetcode.com/problems/best-time-to-buy-and-sell-stock

Code (simplified)   Time: O(n)   Space: O(n)

```cpp
int maxProfit(vector<int>& prices) {
    int profit = 0;
    int buy = prices[0];
    for (auto i = 1; i < prices.size(); i++) {
        buy = min(buy, prices[i]);
        profit = max(profit, prices[i] - buy)
    }
    return profit;
}
```

Code - Best Time to Buy and Sell Stock

 leetcode.com/problems/best-time-to-buy-and-sell-stock

Code (optimized)   Time: **O(n)**   Space: **O(n)**

- Same logic, but with better branch prediction and less computation

```cpp
int maxProfit(vector<int>& prices) {
    int profit = 0;
    int buy = prices[0];
    for (auto i = 1; i < prices.size(); i++) {
        if (prices[i] < buy) {
            buy = prices[i];
        } else if (prices[i] - buy > profit) {
            profit = prices[i] - buy;
        }
    }
    return profit;
}
```

Problem Statement

- You are given an integer **array** of stock prices

- Choose a **price[i]** to buy and **price[i]** to sell where you achieve maximum profits

- You can buy/sell **multiple times**, but only hold **at most one** transaction at a time

- Output is the **maximum profits**

- **Example:**

```
prices = [9, 1, 3, 4]
```

Output: 2 + 1 = 3

buy (price = 1), **sell** (price = 3), **profit** = 2

buy (price = 3), **sell** (price = 4), **profit** = 1

LeetCode leetcode.com/problems/best-time-to-buy-and-sell-stock-ii

Solution

- Loop through the array starting from index 1

- If current **price[i]** is lower than previous **price[i - 1]**, buy and sell

- **Example:**

    prices = [1, 8, 4]   prices[0] = 1, prices[1] = 8, prices[2] = 4

    prices[0] < prices[1] → *true, profit = 8 - 1 = 7*

    prices[2] < prices[1] → false, do nothing

Code   Time: **O(n)**   Space: **O(n)**

```cpp
int maxProfit(vector<int>& prices) {
    int profit = 0;
    for (int i = 1; i < prices.size(); ++i) {
        if (prices[i] > prices[i-1]) {
            profit += prices[i] - prices[i -1];
        }
    }
    return profit;
}
```

Problem - Best Time to Buy and Sell Stock IV

leetcode.com/problems/best-time-to-buy-and-sell-stock-iv

Problem Statement

- ...

LeetCode leetcode.com/problems/best-time-to-buy-and-sell-stock-iv

Solution

- ...

Code - Best Time to Buy and Sell Stock IV

leetcode.com/problems/best-time-to-buy-and-sell-stock-iv

Code (simplified)   Time: **O(n)**   Space: **O(n)**

```cpp
int maxProfit(vector<int>& prices) {
    int profit = 0;
    int buy = prices[0];
    for (auto i = 1; i < prices.size(); i++) {
        buy = min(buy, prices[i]);
        profit = max(profit, prices[i] - buy)
    }
    return profit;
}
```