45. Jump Game II

Solved

Medium | Topics | Companies

You are given a **0-indexed** array of integers `nums` of length `n`. You are initially positioned at `nums[0]`.

Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at `nums[i]`, you can jump to any `nums[i + j]` where:

- `0 <= j <= nums[i]` and
- `i + j < n`

Return *the minimum number of jumps to reach* `nums[n - 1]`. The test cases are generated such that you can reach `nums[n - 1]`.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: nums = [2,3,0,1,4]
Output: 2
```

Constraints:

- `1 <= nums.length <= 10^4`
- `0 <= nums[i] <= 1000`
- It's guaranteed that you can reach `nums[n - 1]`.

Code:

```
int jump(vector<int>& nums) {
        int right=0,left=0,count=0,maxi=0;
        while(right<nums.size()-1){
            count++;
            for(int i=left;i<=right;i++){
                maxi=max(maxi,i+nums[i]);
            }
            left=right+1;
            right=maxi;
            //cout<<right<<endl;
        }
        return count;
}
```

TimeComplexity:O(n)
SpaceComplexity:O(1)

49. Group Anagrams

Solved

Medium | Topics | Companies

Given an array of strings `strs`, group the anagrams together. You can return the answer in **any order**.

Example 1:

> **Input:** `strs = ["eat","tea","tan","ate","nat","bat"]`
>
> **Output:** `[["bat"],["nat","tan"],["ate","eat","tea"]]`
>
> **Explanation:**
>
> - There is no string in strs that can be rearranged to form `"bat"`.
> - The strings `"nat"` and `"tan"` are anagrams as they can be rearranged to form each other.
> - The strings `"ate"`, `"eat"`, and `"tea"` are anagrams as they can be rearranged to form each other.

Example 2:

> **Input:** `strs = [""]`
>
> **Output:** `[[""]]`

Example 3:

> **Input:** `strs = ["a"]`
>
> **Output:** `[["a"]]`

Constraints:

- `1 <= strs.length <= 10^4`
- `0 <= strs[i].length <= 100`

Code:

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string,vector<string>> hash;
        for(int i=0;i<strs.size();i++){
            string a=strs[i];
            sort(a.begin(),a.end());
            hash[a].push_back(strs[i]);
        }
        vector<vector<string>> ans;
        for(auto a:hash){
            ans.push_back(a.second);
        }
        return ans;
}
```

TimeComplexity:O(nlogn)
SpaceComplexity:O(n)

122. Best Time to Buy and Sell Stock II

Solved

Medium | Topics | Companies

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the `i^th` day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the **maximum** profit you can achieve*.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
Total profit is 4 + 3 = 7.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Total profit is 4.
```

Example 3:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.
```

Constraints:

- `1 <= prices.length <= 3 * 10^4`

Code:

```
int rec(vector<int>& prices,int index,int can,vector<vector<int>>& dp){
        if(index==prices.size())return 0;
        if(dp[index][can]!=-1)return dp[index][can];
        int case1=0,case2=0;
        if(can==1){
            case1=prices[index]+rec(prices,index+1,0,dp);
            case2=0+rec(prices,index+1,1,dp);
        }
        else{
            case1=-prices[index]+rec(prices,index+1,1,dp);
            case2=0+rec(prices,index+1,0,dp);
        }

        return dp[index][can]=max(case1,case2);
}
int maxProfit(vector<int>& prices) {
        vector<vector<int>> dp(prices.size(),vector<int>(2,-1));
        return rec(prices,0,0,dp);
}
```

TimeComplexity:O(n)
SpaceComplexity:O(n)

16. 3Sum Closest

Solved

Medium | Topics | Companies

Given an integer array `nums` of length `n` and an integer `target`, find three integers in `nums` such that the sum is closest to `target`.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

Example 1:

```
Input: nums = [-1,2,1,-4], target = 1
Output: 2
Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

Example 2:

```
Input: nums = [0,0,0], target = 1
Output: 0
Explanation: The sum that is closest to the target is 0. (0 + 0 + 0 = 0).
```

Constraints:

- `3 <= nums.length <= 500`
- `-1000 <= nums[i] <= 1000`
- `-10^4 <= target <= 10^4`

Code:

```
int threeSumClosest(vector<int>& nums, int target) {
        int mini=INT_MAX,ans=0;
        sort(nums.begin(),nums.end());
        for(int i=0;i<nums.size()-2;i++){
            int left=i+1,right=nums.size()-1;
            while(left>i and right<nums.size() and left<right){
                int sum=nums[i]+nums[left]+nums[right];
                //cout<<nums[i]<<" "<<nums[left]<<" "<<nums[right]<<endl;
                if(abs(sum-target)<mini){
                    mini=abs(sum-target);
                    ans=sum;
                }
                if(sum==target){
                    return sum;
                }
                else if(sum<target){
                    left++;
                }
                else{
                    right--;
                }
            }
        }
        return ans;
}
```

TimeComplexity:O(n2)**
SpaceComplexity:O(1)

200. Number of Islands

Solved

Medium | Topics | Companies

Given an `m x n` 2D binary grid `grid` which represents a map of `'1'`s (land) and `'0'`s (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [
  ["1","1","1","1","0"],
  ["1","1","0","1","0"],
  ["1","1","0","0","0"],
  ["0","0","0","0","0"]
]
Output: 1
```

Example 2:

```
Input: grid = [
  ["1","1","0","0","0"],
  ["1","1","0","0","0"],
  ["0","0","1","0","0"],
  ["0","0","0","1","1"]
]
Output: 3
```

Constraints:

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 300`

Code:

```
void markzero(vector<vector<char>>& grid,int i,int j){
        if(i>=grid.size() || j>=grid[i].size() || j<0 || i<0 || grid[i][j]=='0')return;
        grid[i][j]='0';
        markzero(grid,i+1,j);
        markzero(grid,i-1,j);
        markzero(grid,i,j+1);
        markzero(grid,i,j-1);
}
int numIslands(vector<vector<char>>& grid) {
        int count=0;
        for(int i=0;i<grid.size();i++){
            for(int j=0;j<grid[i].size();j++){
                if(grid[i][j]=='1'){
                    count++;
                    markzero(grid,i,j);
                }
            }
        }
        return count;
}
```

TimeComplexity:

SpaceComplexity:

Quick Sort

Difficulty: **Medium** Accuracy: **55.23%** Submissions: **236K+** Points: **4**

Implement Quick Sort, a Divide and Conquer algorithm, to sort an array, **arr**[] in ascending order. Given an array, **arr**[], with starting index **low** and ending index **high**, complete the functions **partition()** and **quickSort()**. Use the last element as the pivot so that all elements less than or equal to the pivot come before it, and elements greater than the pivot follow it.

Note: The **low** and **high** are inclusive.

Examples:

Input: arr[] = [4, 1, 3, 9, 7]
Output: [1, 3, 4, 7, 9]
Explanation: After sorting, all elements are arranged in ascending order.

Input: arr[] = [2, 1, 6, 10, 4, 1, 3, 9, 7]
Output: [1, 1, 2, 3, 4, 6, 7, 9, 10]
Explanation: Duplicate elements (1) are retained in sorted order.

Input: arr[] = [5, 5, 5, 5]
Output: [5, 5, 5, 5]
Explanation: All elements are identical, so the array remains unchanged.

Constraints:
1 <= arr.size() <= 10^3
1 <= arr[i] <= 10^4

Code:

```
int partition(vector<int>& arr, int low, int high) {
        int pivot=arr[low],i=low,j=high;
        while(i<j){
            while(arr[i]<=pivot and i<=high-1)i++;
            while(arr[j]>pivot and j>=low+1)j--;
            if(i<j)swap(arr[i],arr[j]);
        }
        swap(arr[low],arr[j]);
        return j;
}
void quickSort(vector<int>& arr, int low, int high) {
        if(low<high){
            int p=partition(arr,low,high);
            quickSort(arr,low,p-1);
            quickSort(arr,p+1,high);
        }
        return ;
}
```

TimeComplexity:
SpaceComplexity:

Merge Sort

Difficulty: **Medium** Accuracy: **54.1%** Submissions: **205K+** Points: **4**

Given an array arr[], its starting position l and its ending position r. Sort the array using the merge sort algorithm.

Examples:

Input: arr[] = [4, 1, 3, 9, 7]
Output: [1, 3, 4, 7, 9]

Input: arr[] = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
Output: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Input: arr[] = [1, 3 , 2]
Output: [1, 2, 3]

Constraints:
1 <= arr.size() <= 10^5
1 <= arr[i] <= 10^5

Code:

```
void merge(int arr[], int l, int m, int r)
    {
        vector<int> temp;
        int right=m+1;
        int left=l;
        while(left<=m && right<=r){
            if(arr[left]<=arr[right]){
                temp.push_back(arr[left]);
                left++;
            }
            else{
                temp.push_back(arr[right]);
                right++;
            }
        }
        while(left<=m){
            temp.push_back(arr[left]);
            left++;
        }
        while(right<=r){
            temp.push_back(arr[right]);
            right++;
        }
        for(int i=l;i<=r;i++){
            arr[i]=temp[i-l];
        }
}
void mergeSort(int arr[], int l, int r){
        if(l==r){
            return;
        }
        int mid=(l+r)/2;
        mergeSort(arr,l,mid);
        mergeSort(arr,mid+1,r);
        merge(arr,l,mid,r);
}
```

TimeComplexity:
SpaceComplexity: