Leetcode The Hard Way

Arrays

Authors: @heiheihang@wingkwong

Overview

Arrays are a common data structure used in many programming languages such as Python, C++, Java, and Javascript. They are used to store a collection of items and can be one-dimensional or multi-dimensional.

In the context of LeetCode, the term "array" can refer to different data structures in different languages, such as a List in Python, Array or Vector in C++, Array or ArrayList in Java, and Array in Javascript.

Let's take a look at some examples of array:

- Python
- C++
- Java

Written by @heiheihang

```
scores_of_students = [86, 76, 67, 98, 95]
boys_and_girls_of_classes = [[10,23], [20,20], [15,12], [13,16]]
basketball_matches = [[0, 76, 86, 100],
[56, 0, 87, 65],
[65, 34, 0, 86],
[72, 65, 78, 0]]
```

We have 3 types of arrays.

The syntax that we are using is Python, please refer to your own language of preference if needed.

scores_of_students: This array is an 1-d array containing the scores of each student. We can perform the following operations to obtain different information of the scores:

- max(scores_of_students): returns the highest score in the array, which is 98 in this case
- min(scores_of_students): returns the highest score in the array, which is 67 in this case
- sum(scores_of_students): returns the sum of score in the array, which is 422 in this case
- scores_of_students.sort(): sort the scores in order, which is [67, 76, 86, 95, 98] in this case. Note that it is preferred to use the pre-built sorting function when you are solving a problem NOT TARGETED to teach you sorting. It will speed up your learning.

boys_and_girls_of_classes: This array is a 2-d array containing the number of boys and girls of each class. For example, the first class has 10 boys and 23 girls. We can access the number of girls in the 3rd class with boys_and_girls_of_classes[2][1]. This is useful to obtain specific information. Let's take a look at several operations on this 2-d array:

- boys_and_girls_of_classes.sort(): We have two elements in each entry this time. In pythonthe pre-built sort()sorts by the first element, then the second element, then the third (if it exists) etc...
- boys_and_girls_of_classes.sort(key = lambda Class : Class[1]): We can use the keyparameter to change the sort()behaviour. We declare that we want to look at the number of girls FIRSTin each Classin this case. There are more advanced application of sort, and we will learn them in harder problems.

- list(map(lambda Class: Class[0] + Class[1], boys_and_girls_of_classes)): This returns the list of the class size of each size. It is useful if we want to know the total number of students in each class.

basketball_matches: This array is a 2-d array, but it is special that its dimension is n x n. These arrays (or better, matrices) usually have a special meaning. In this case, we have the scores of each team competing with each other. For example, team 1 vs team 2 has the score of 76 - 56. We will use for-loops to iterate these arrays.

- Python
- C++
- Java

Written by @heiheihang

```
for i in range(len(basketball_matches)):
for j in range(len(basketball_matches[0])):
print("Team " + str(i) + " " + basketball_matches[i][j])
print("Against")
print("Team " + str(j) + " " + basketball_matches[j][i])
```

Complexity

Operation	Complexity	Explanation
Look-up (Access)	$$O(1)$$	When we do array[1], the program can instantly find the value stored at the first location.
Add	$$O(1)$$	More accurately this is amortised O(1). When we add to the end of the array, it only takes constant time.
Pop	$$O(1)$$	When we remove the last element of the array, it takes constant time.

Insert	$$O(N)$$	When we insert an element to the middle of the array, it takes O(N) time. The whole array needs to be restructured to accommodate the new element.
Remove	$$O(N)$$	When we remove an element in the middle of the array, it takes O(N) time. The whole array needs to be restructured to replace the missing gap of the replaced element.
Len	$$O(1)$$	This may seem like to be O(N) as we have to go through the whole array to check its length. However, checking the length of an array in many languages should be

pre-computed in their data structures, so it only takes constant time.

Suggested Problems

Problem Name	Difficulty	Solution Link
1480 - Running Sum of 1d Array	Easy	View Solutions
1929 - Concatenation of Array	Easy	View Solutions
1431 - Kids With the Greatest Number of Candies	Easy	View Solutions
1572 - Matrix Diagonal Sum	Easy	View Solutions
0036 - Valid Sudoku	Medium	N/A

Backtracking

Author: @wingkwong

Overview

Backtracking is a general algorithmic technique that involves exploring all possible solutions to a problem by incrementally building a solution and then undoing (or "backtracking" on) the choices that lead to dead ends. It is a form of depth-first search and is particularly useful for solving problems that involve searching through a large number of possibilities, such as finding all possible solutions to a problem or finding the one solution that satisfies a set of constraints.

The steps for using backtracking to solve a problem are as follows:

1. Understand the problem and its requirements by reading the problem statement and examples.
2. Develop a recursive algorithm that incrementally builds a solution and backtracks when a dead end is reached.
3. Define a base case for the recursion that indicates when a complete solution has been found, and a terminating condition that indicates when to stop the recursion.
4. Test the solution on the provided test cases to check if it works correctly and returns the expected output.

Backtracking can be used for various of problems such as:

- Generating all possible combinations of a set of items.
- Finding all possible solutions to a problem
- Finding a specific solution that satisfies a set of constraints.
- Solving puzzles or other combinatorial problems
- And many more

Backtracking can be very inefficient, especially when the number of possible solutions or the size of the input is large. Therefore, it is important to carefully analyze the problem and develop an efficient backtracking algorithm.

Example: 0046 - Permutations (Medium)

If we have an array $nums$ of distinct integers, what are all the possible permutations? If the input is $[1,2,3]$, then the permutations would be $[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]$. In C++, it is easy to solve this problem by using the built-in STL next_permutation. However, we can also solve it using backtracking.

The general steps are as follows.

1. Sort the input array if necessary. However, in this example, sorting is not necessary.

```
sort(nums.begin(), nums.end());
```

2. Define ansand tmpwhere ansis the array storing all final permutations and tmpis used to store possible permutations at some point.

```
vector<vector<int>> ans;
vector<int> tmp;
```

3. Call backtrack()function in main

```
backtrack(nums, ans, tmp);
```

4. Let's add logic in backtrack()function. First we need to define the exit criteria. When should we push tmpto ans? If tmpalready got enough candidates, then we can push tmpto ans.

```
if ((int) tmp.size() == (int) nums.size()) {
ans.push_back(tmp);
return;
}
```

5. Iterate each number, check If the candidate has been used or not, skip it if it is already in tmp. Otherwise, push it to tmpand call backtrack()again. After that, remove the previous candidate from tmpand move to the next candidate.

```
for (auto x : nums) {
if (find(tmp.begin(), tmp.end(), x) != tmp.end()) continue;
tmp.push_back(x);
backtrack(nums, ans, tmp);
tmp.pop_back();
}
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0039 - Combination Sum	Medium	View Solutions
0040 - Combination Sum II	Medium	View Solutions
0046 - Permutations	Medium	View Solutions
0078 - Subsets	Medium	View Solutions

Binary Search

Authors: @heiheihang@wingkwong

Overview

Binary search is a widely used algorithm for searching an element in a sorted array or list. The basic idea of binary search is to divide the search space in half with each iteration and compare the middle element with the target element. If the middle

element is greater than the target element, the search space is reduced to the left half of the array, otherwise, it is reduced to the right half. This process is repeated until the target element is found or the search space is exhausted.
The time complexity of binary search is $O(log n)$, which is more efficient than the linear search algorithm $O(n)$, which checks all elements one by one. However, for binary search to work, the array or list must be sorted.
Binary search can be implemented using a while loop, recursion, or a combination of both. The implementation details may vary, but the basic idea remains the same. The basic steps for a binary search algorithm are:

1. Initialize two pointers, one pointing to the start of the array and the other pointing to the end.
2. Find the middle element of the array by calculating the average of the two pointers.
3. Compare the middle element with the target element.
4. If the middle element is equal to the target element, the search is complete and the index of the target element is returned.
5. If the middle element is less than the target element, move the left pointer to the middle element + 1 and repeat step 2.
6. If the middle element is greater than the target element, move the right pointer to the middle element - 1 and repeat step 2.
7. If the left pointer is greater than the right pointer, the target element is not found and the function returns -1.

In conclusion, binary search is a fast and efficient algorithm for searching an element in a sorted array or list. Its time complexity is $O(log n)$, which is much better than the linear search algorithm. However, it requires the array or list to be sorted for it to work.
Let's look at the most basic form of binary search:

Example: 0704. Binary Search

Given an array of integers numswhich is sorted in ascending order, and an integer target, write a function to search targetin nums. If targetexists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n)runtime complexity.

For example, given the sorted input:

```
nums = [-1,0,3,5,9,12], target = 9
```

The index of the element 9 is 4. We can use the following template to find the target

- Python
- C++
- Java

Written by @heiheihang

```
def binarySearch(nums, target):
lp, rp = 0, len(nums) - 1
while (lp <= rp):
mid = (lp + rp) // 2
if (nums[mid] == target):
return mid
elif (nums[mid] < target):
lp = mid + 1
else:
rp = mid - 1
return -1
```

There can be very challenging questions using binary search, but we should focus on the basic application first.

Tips

1. When the input is sorted, it's probably a hint to use binary search.
2. When you need to find the first / last index of something, then it may be another hint to use binary search (as index is sorted).
3. When initialising the boundary, we may think about the possible range (sometimes r may not necessarily be the max of the constraint
4. In while condition, you may see some people write $while (l < r)$, $while (r > l)$, $while (l <= r)$, or $while (l >= r)$. either one works depending on how to write the logic - just personal preference. See herefor more.
5. Think about the mid - if there are even elements, should we pick the left or the right mid? e.g. [1,2,3,4] like choosing 2 or 3?
6. Think about how to shrink - if mid is never be the answer, then we can exclude it ($l = m + 1$ / $r = m - 1$).
7. If you are using C++, most people use $l + (r - l) / 2$ to avoid integer overflow problem. Let's say l & r are large enough like ~INT_MAX, when u sum them up, it causes overflow.
   We know that m is actually somewhere in between l and r so let $m = l + x$ where x is an arbitrary value. Since we know that $m = (l + r) / 2$, we can substitute to have the following $$$ l + x = (l + r) / 2 \ x = (l + r) / 2 - l \ x *2 = (l + r) - 2* l \ x * 2 = (r - l) \ x = (r - l) / 2 \ $$$ so putting x back to the first equation, we would have $$$ m = l + x \ m = l + (r - l) / 2 $$$

Suggested Problems

Problem Name	Difficulty	Solution Link
0704 - Binary Search	Easy	View Solutions
0153 - Find Minimum in Rotated Sorted Array	Medium	N/A
0162 - Find Peak Element	Medium	N/A
0154 - Find Minimum in Rotated Sorted Array II	Hard	N/A

Brute Force

Author: @wingkwong

Overview

The brute force method for solving a problem involves using a simple and straightforward approach to solve the problem, without worrying about optimization. This method is often used as a starting point for solving a problem, as it can help to understand the problem better and develop a basic understanding of its requirements. The steps for using the brute force method are as follows:

1. Understand the problem and its requirements by reading the problem statement and examples.
2. Develop a simple and straightforward solution that solves the problem by using a nested loop (or loops) to iterate through all possible combinations of the input.
3. Test the solution on the provided test cases to check if it works correctly and returns the expected output.
4. Optimize the solution if necessary, by analyzing its time and space complexity and identifying areas where it can be improved.

Note that the brute force approach often results in a solution that has a high time complexity, it is important to analyze the solution and optimize it, or come up with a more efficient algorithm that solves the problem, because the brute force method is not a good solution for problems that have a large input size or a large number of test cases.

Example 1: 1480 -Running Sum of 1d Array

Given an array nums. We define a running sum of an array as $runningSum[i] = sum(nums[0] ... nums[i])$. Return the running sum of nums.

Input: nums = $[1,2,3,4]$

Output: $[1,3,6,10]$

Explanation: Running sum is obtained as follows: $[1, 1 + 2, 1 + 2 + 3, 1 + 2 + 3 + 4]$

For a brute force solution, we iterate each element $a[i]$ and we iterate from $j = [0 .. i]$ to add $a[j]$ to sum. This solution is acceptable but it is slow as we have two for-loops here. A better approach would be using Prefix Sumto reduce time complexity.

- C++

Written by @wingkwong

```
class Solution {
public:
vector<int> runningSum(vector<int>& nums) {
int n = nums.size();
vector<int> ans(n);
for (int i = 0; i < n; i++) {
int sum = 0;
for (int j = 0; j <= i; j++) {
sum += nums[j];
}
ans[i] = sum;
}
return ans;
}
};
```

Example 2: 2006 - Count Number of Pairs With Absolute Difference K

Given an integer array nums and an integer k, return the number of pairs (i, j) where $i < j$ such that $|nums[i] - nums[j]| == k$.

The value of $|x|$ is defined as:

x if $x >= 0$

$-x$ if $x < 0$

Similar to Example 1, we iterate each element and iterate the elements after that to search for each pair to see if the condition can be met or not. Some better approaches would be using Sliding Windowor Counting Sort to reduce time complexity.

- C++

Written by @wingkwong

```
class Solution {
public:
int countKDifference(vector<int>& nums, int k) {
int n = nums.size(), ans = 0;
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (abs(nums[i] - nums[j]) == k) {
ans += 1;
}
}
}
return ans;
}
};
```

Greedy

Authors: @abhishek-sultaniya@wingkwong

Overview

A greedy algorithm is a type of algorithmic approach that follows the problem-solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In other words, at each step, it chooses the option that looks the best at that moment without considering the potential impact of the decision on future steps.
This algorithm is mainly used for optimization problems where the goal is to find the best solution among a set of possibilities. The solutions are constructed incrementally, with the algorithm making the locally optimal choice at each stage.
A Greedy algorithm is simple and easy to implement, but it doesn't always give an optimal solution. It can be used to solve problems such as scheduling, Huffman coding, and finding the shortest path in a graph.
Overall, the Greedy algorithm is a useful approach for solving optimization problems, but it should be used with caution, as it may not always lead to the best global solution.

Example 1: 0605 - Can Place Flowers

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots.
Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule.
For a greedy solution, we would solve in such a way that we will always have the best choice at every max. Our task is to calculate maximum flowers we can plant. Its simple that if there are three consecutive zeroes then the middle one will be planted. But if we have to calculate maximum then we will miss 2 side case this way.
Case 1: 001....

Here intially we have just 2 consecutive zeroes but we can plant at first place. So we will consider this case too.
Case 2: ...100
Here at the end we have just 2 consecutive zeroes but we can plant at last place. So we will consider this case too.

- C++

Written by @abhishek-sultaniya

```
class Solution {
public:
bool canPlaceFlowers(vector<int>& flowerbed, int p) {
int n = flowerbed.size();
// count variable will calculate max flowers we can plant
int count = 0;
if (flowerbed[0] == 0 && n == 1) {
count++;
}
// The following will cover case 1
if (n > 1 && flowerbed[0] == 0 && flowerbed[1] == 0) {
count++;
flowerbed[0] = 1;
}
// Mid approach to check 3 consecutive zeroes
for (int i = 1; i < n - 1; i++) {
if (flowerbed[i] == 0 && flowerbed[i - 1] == 0 && flowerbed[i + 1] == 0) {
flowerbed[i] = 1;
count++;
}
}
// The following will cover case 2
if (n > 2 && flowerbed[n - 2] == 0 && flowerbed[n - 1] == 0) {
count++;
}
return count >= p;
}
};
```

Example 2: 455 - Assign Cookies

Assume you are an awesome parent and want to give your children some cookies. But, you should give each child at most one cookie.
Each child i has a greed factor g[i], which is the minimum size of a cookie that the child will be content with; and each cookie j has a size s[j]. If s[j] >= g[i], we can assign the cookie j to the child i, and the child i will be content. Your goal is to maximize the number of your content children and output the maximum number
This problem uses the concept of greedy. Our aim is to just assign the cookies starting from the child with less greediness to maximize the number of happy children. So we will sort greediness of child and size of cookies too. Then as soon as a child gets the cookie, we move to next child. We are using greedy in such a way that if child will less greediness cannot get a cookie, then all children with higher greediness will also not get that cookie.

- C++

Written by @abhishek-sultaniya

```
class Solution {
public:
int findContentChildren(vector<int>& g, vector<int>& s) {
sort(s.begin(), s.end());
sort(g.begin(), g.end());
int ans = 0;
for (int j = 0; j < s.size() && ans < g.size(); ++j) {
```

```
if (g[ans] <= s[j]) {
ans++;
}
}
return ans;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0561 - Array Partition	Easy	N/A
0860 - Lemonade Change	Easy	N/A
1005 - Maximize Sum Of Array After K Negations	Easy	N/A
0045 - Jump Game II	Medium	N/A
0402 - Remove K Digits	Medium	N/A

Hash Map

Authors: @heiheihang@wingkwong

Overview

A Hash map, also known as a hash table, is a data structure that stores key-value pairs and uses a hash function to map keys to their corresponding values. The hash function takes a key as input, performs some calculations on it, and returns an index (also known as a "hash code") where the value corresponding to that key can be found.

The basic idea behind a hash map is to use the key to quickly locate the corresponding value in the table, without having to search through all the elements one by one, like in a linear search. The time complexity of a hash map is $O(1)$ on average, which is much faster than a linear search $O(n)$.

A hash map is implemented as an array of "buckets", where each bucket can store one or more key-value pairs. When a key is hashed, the hash function calculates the index of the bucket where the key-value pair should be stored. If two keys hash to the same index, it is called a collision, and there are different strategies to handle collisions, such as chaining or open addressing.

Hash maps are widely used in many applications, such as databases, caches, and programming languages. They are also used as an efficient data structure for implementing other data structures such as sets, dictionaries, and associative arrays.

Hash maps have a few advantages over other data structures such as arrays and linked lists:

- Hash maps have constant time complexity for inserting, retrieving and removing elements, which makes them more efficient than arrays and linked lists, which have linear time complexity.
- Hash maps can store any type of data as a key, while arrays can only store integers as keys.
- Hash maps can be resized dynamically, which allows them to adapt to changing data and usage patterns.

In conclusion, a hash map is a data structure that stores key-value pairs and uses a hash function to quickly locate values based on keys. It has an average time complexity of O(1), which makes it more efficient than other data structures such as arrays and linked lists. Hash maps are widely used in many applications and are an efficient data structure for implementing other data structures such as sets, dictionaries, and associative arrays.

Example: 0001 - Two Sum (Easy)

Given an array of integers numsand an integer target, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have *exactly*one solution, and you may not use the *same*element twice.

You can return the answer in any order.

For example, given the following input:

```
nums = [2,7,11,15], target = 9
```

We can see that the first two elements (2and 7) add up to the target (9). So we need to return [0,1], as these two indices refer to 2and 7.

The naive way to solve this problem is to use a nested for-loop:

- Python
- C++
- Java

Written by @heiheihang

```
class Solution:
def twoSum(self, nums: List[int], target: int) -> List[int]:
# locate the first element
for i in range(len(nums)):
# search the second element from i + 1
for j in range(i+1, len(nums)):
# check if they add up to target
if(nums[i] + nums[j] == target):
# return the two indices if they do
return [i, j]
return -1
```

We observe that with a nested for-loop, the runtime complexity is $$O(n^2)$$. Let us look at how hash map can help us here.

Hash Map basically is a label. For example, if we want to store the (value, index) pair from the example above in a Hash Map.

- Python
- C++
- Java

Written by @heiheihang

```
# we use {} to initialize a hash map
hash_map = {}
# we want to map the (value, index) pair in this list
input_1 = [2,7,11,15]
```

```
# we put the (value, index) pair to the hash map
hash_map[2] = 0
hash_map[7] = 1
hash_map[11] = 2
hash_map[15] = 3
```

The special feature of hash map is that, from now on, if we want to know a value exists in input_1or not, we can just perform:

- Python
- C++
- Java

Written by @heiheihang

```
if( 7 in hash_map):
print("7 is in input_1")
else:
print("7 is not in input_1")
```

This operation only takes $$O(1)$$ time! Without hash map, we would need to iterate the input to search for a specific element.
After understanding Hash Map, are you able to solve Two Sumin $$O(N)$$ time?

Suggested Problems

Problem Name	Difficulty	Solution Link
0217 - Contains Duplicate	Easy	View Solutions
0219 - Contains Duplicate II	Easy	View Solutions
0003 - Longest Substring Without Repeating Characters	Medium	View Solutions
0706 - Design HashMap	Medium	View Solutions

Heap (Priority Queue)

Authors: @heiheihang@potatochick@SkollRyu

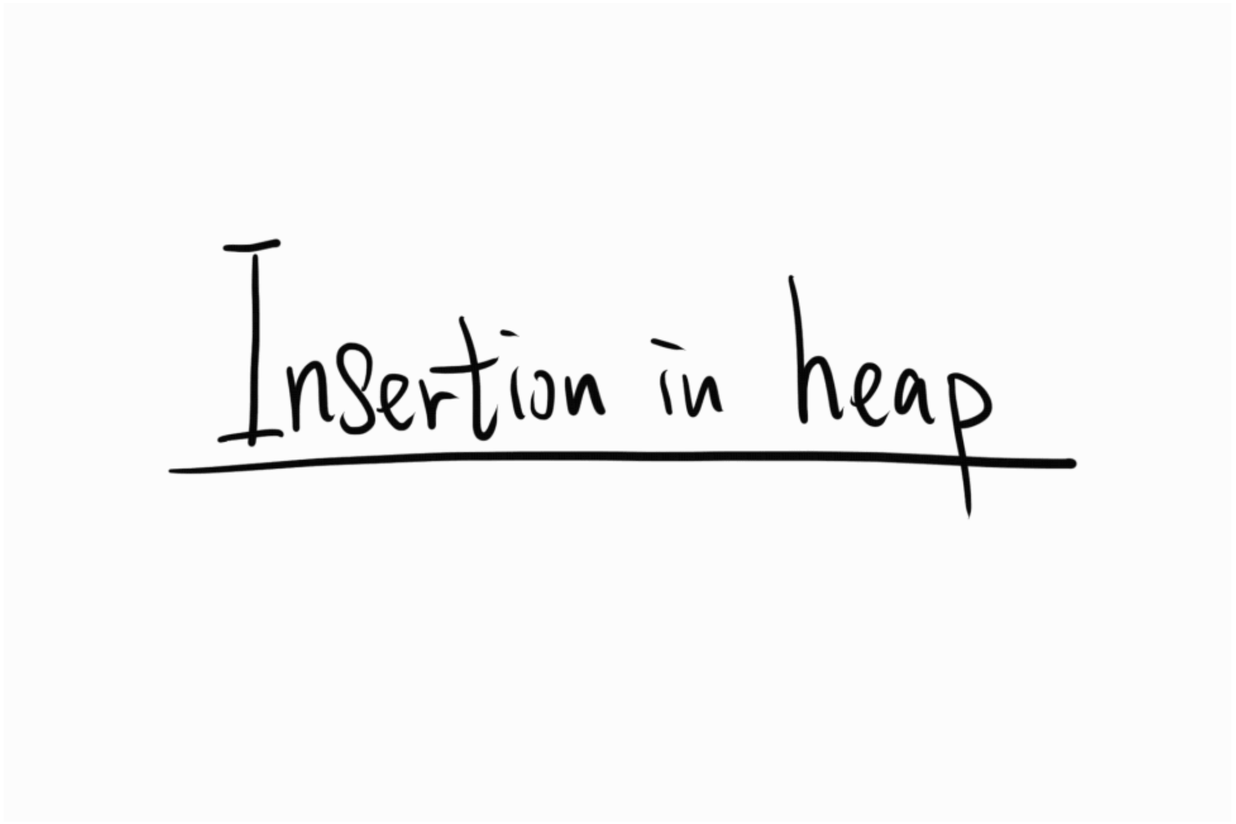
Overview

A heap, or a priority queue, is a data structure that efficiently stores elements in a particular order. It is very efficient in inserting an element to the heap ($$O(logN)$$), and very efficient in removing the first element of the heap ($$O(logN)$$). To know the details of heap, we recommend you to look at this.

Operations

Insertion

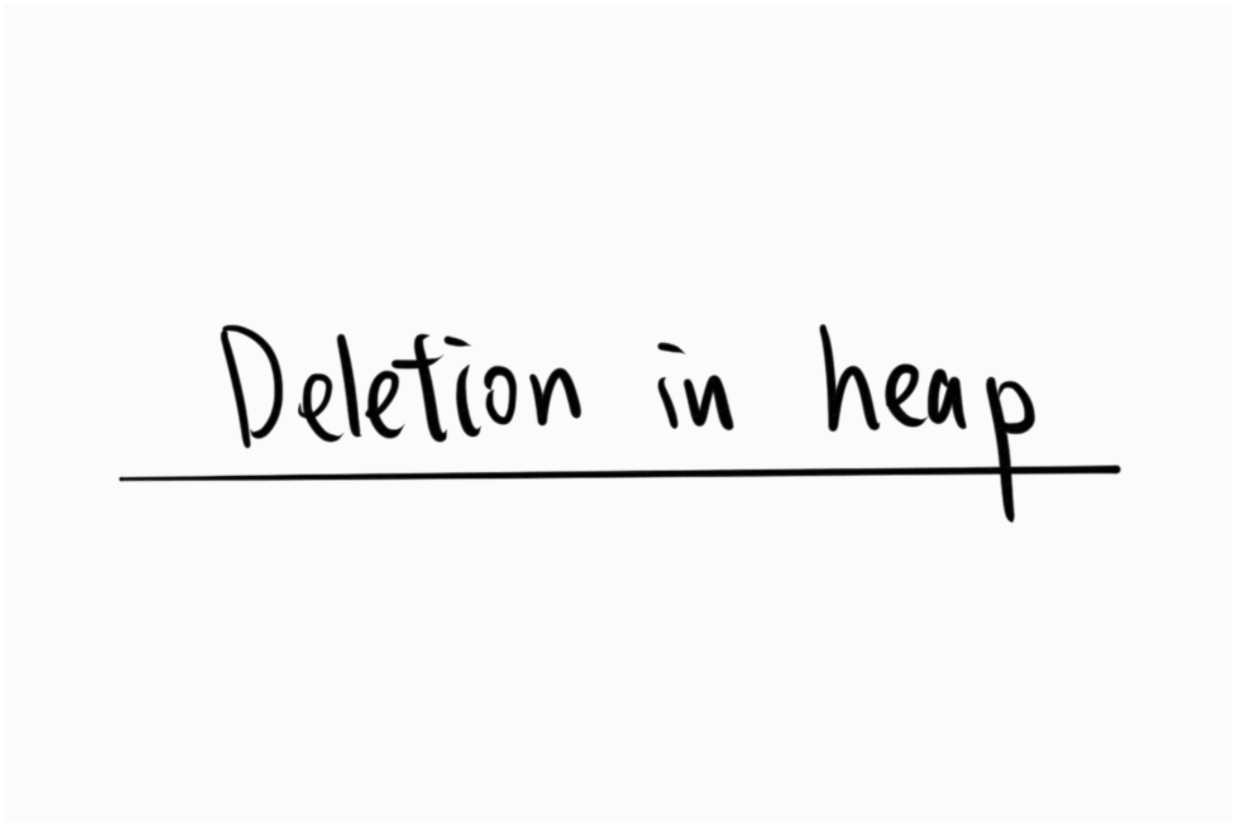
To do insetion in heap, we would add the new element to the end of the heap. If the position of the new element violates the heap properity, the new elements will be sifted up until it reaches the correct position.



Insertion in heap

Deletion

In heap, the deletion refers to pop the root in the heap. After the root of the heap has been popped out, the last element in the heap will be inserted to the root position. If this violates the heap porperity, the new root would be sifted down until it reaches the correct position.



Python

By default, when we refer to heap, most implementations are min-heaps. This means the first element is always the smallest element.

In Python, you can use the following functions to interact with a heap:

```
heap = [] #to initialize a heap, it is just a list
heappush(heap, 3) # to add an element to the heap, we can use heappush
#heap = [3]
heappush(heap, 5)
```

```
#heap = [3, 5] , the heap always keeps the smallest element in front!
smallest_element_from_heap = heappop(heap) #we can remove the first element from heap with
heappop
#heap = [5] , 3 is removed
#smallest_element_from_heap = 3 #after heappop, it is stored in the variable
```

That's it! These are the operations you need for using heap in LeetCode.
There is one more trick to learn for using heap. How do we tweak the heap to make it a max-heap?

```
max_heap = []
#we want to store 10, but change it to -10 for max-heap
heappush(max_heap, -10)
#max_heap = [-10]
#we want to store 7, but change it to -7 for max-heap
heappush(max_heap, -7)
#max_heap = [-10, -7]
max_element_from_heap = -1 * heappop(heap)
#heap = [-7], -10 is removed
#max_element_from_heap = 10, we have retrieved the largest element from the heap
```

Let's work on a problem (LeetCode Link)
You are given an array of integers stoneswhere stones[i]is the weight of the ithstone.
We are playing a game with the stones. On each turn, we choose the heaviest two stonesand smash them together. Suppose the heaviest two stones have weights xand ywith x <= y. The result of this smash is:

- If x == y, both stones are destroyed, and
- If x != y, the stone of weight xis destroyed, and the stone of weight yhas new weight y - x.

At the end of the game, there is at most onestone left.
Return *the smallest possible weight of the left stone*. If there are no stones left, return 0.
I want you to think about these questions before working on it:

- Should we use a min-heap or a max-heap?
- If it is a max-heap, how to we "store" the numbers?
- What do we have to check before retrieving the two heaviest stones?

As we need to get the two heaviest stones in every iteration, we should use a max-heap for quick access of the largest elements. To use a max-heap, we can store the negative of the integer. We have to check if there are at least two more stones in the heap before retrieving the two heaviest stones.

```
def lastStoneWeight(self, stones: List[int]) -> int:
#initialize the max_heap
max_heap = []
#add elements to max_heap
for stone in stones:
#store its negative to keep the most positive stone in front
heappush(max_heap, -stone)
#we have to check if there are at least two stones in the heap
while(len(max_heap) > 1):
#pop the largest element from max_heap, multiplied by -1
largest_stone = -1 * heappop(max_heap)
#pop the second largest element (now largest) from max_heap, multiplied by -1
second_largest_stone = -1 * heappop(max_heap)
#push the new stone if they are not equal
if(largest_stone != second_largest_stone):
new_stone = largest_stone - second_largest_stone
#remember to store its negative
heappush(max_heap, -new_stone)
#if there is a stone left, return the stone, its positive value
if(max_heap):
```

```
return -max_heap[0]
#if no stone left, return 0
return 0
```

C++

In C++, when we are refer to heap, we mostly refer to priority queue. By default, priority queue is a max heap in c++.

Create a max heap:

```
priority_queue<int> max_heap; // max heap
```

Create a min heap:

```
priority_queue<int,vector<int>,greater<int>> min_heap; // min heap
```

Other related function:

```
priority_queue<int> max_heap; //max heap
//To push element into a priority queue
max_heap.push(1);
max_heap.push(2);
max_heap.push(3);
//max_heap now contains: {3,2,1}
//To push element from a vector into a priority queue
vector<int> vc = {6,5,4};
for (auto x:vc){
max_heap.push(x);
}
//max_heap now contains: {6,5,4,3,2,1}
//To get element from the priority queue
```

```
int top_element = max_heap.top();max_heap.pop();
cout<<top_element; //output: 6
//As we want to access the second largest element later, we need to remove the max element
after we access it.
//To get all element from the priority queue
while(!max_heap.empty()){
int element = max_heap.top();max_heap.pop();
cout<<element<<" "; //output: 5 4 3 2 1
}
```

Advance usage:Use heap to sort the element by value while containing the index of the elements.

Let's work on a problem (LeetCode Link)

you are given an m x n binary matrix mat of 1's (representing soldiers) and 0's (representing civilians). The soldiers are positioned in front of >the civilians. That is, all the 1's will appear to the left of all the 0's in each row.

A row i is weaker than a row j if one of the following is true:

The number of soldiers in row i is less than the number of soldiers in row j. Both rows have the same number of soldiers and i < j. Return the indices of the k weakest rows in the matrix ordered from weakest to strongest.

The idea of this question is

- count the number of soilders in each row
- sort it
- return the 1st - kth weakest indexof row

We will use a min heap as we want the result rank from weakest to strongest.

Create a min heap which will contains pair of {number of soldiers in the row, row index}.

By default, c++ will rank the order of element by the first element in the heap. In this case, it will be number of soldiers

```
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> pq;
//priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq; // this line will also
work
```

To access the pair of {number of soldiers in the row, row index}

```
pair<int,int> top_element;
top_element=pq.top();pq.pop();
int number_of_soldiers = top_element.first;
int index = top_element.second;
```

My solution:

```
vector<int> kWeakestRows(vector<vector<int>>& mat, int k) {
priority_queue<pair<int,int>,vector<pair<int,int>>,greater<>> pq; //min heap
//push elements to min heap
for (int i =0;i<mat.size();i++){
int count= 0;
for (int j=0;j<mat[0].size();j++){
if (mat[i][j] == 1) count++;
}
pq.push({count,i}); //push pair of {number of soldiers in the row, row index} to the min
heap
//pq.push(make_pair(count,i)); can replace with this line of syntax
}
vector<int> result;
int count = 0;
//get the index only from the heap and put it in the array
while(!pq.empty() && count<k){
count++;
int ans = pq.top().second;
pq.pop();
result.push_back(ans);
}
return result;
}
```

Additional knowledge:You can create a max heap with pair<int,int>with following syntax

```
priority_queue<pair<int,int>> pq;
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0703 - Kth Largest Element in a Stream	Easy	View Solutions
0215 - Kth Largest Element in an Array	Medium	View Solutions
0973 - K Closest Points to Origin	Medium	View Solutions

Kadane Algorithm

Authors: @ShivaRapolu01@wingkwong

Overview

The Kadane's algorithm is a well-known method for solving the problem of finding the maximum sum of a contiguous subarray of a given array of numbers. The basic idea behind the algorithm is to iterate through the array, keeping track of the maximum sum seen so far and the current sum, and updating the maximum sum whenever a new maximum is found. The algorithm has a time complexity of $O(n)$.

Algorithm

1. Initialize variables to keep track of the current sum and maximum sum, setting them both to the first element of the array.
2. Starting from the second element, iterate through the rest of the array.
3. At each element, calculate the current sum by adding the current element to the previous current sum. If the current sum is less than zero, set the current sum to zero.
4. Compare the current sum to the maximum sum and update the maximum sum if the current sum is greater.
5. Return the maximum sum as the result of the algorithm.

Note: Algorithm will work for an array of integers where all numbers in the array are non-negative. If the array contains negative numbers, a variant of this algorithm called "Maximum subarray problem" should be used.

Example 1: [0053 - Maximum Subarray]

```
Given an integer array nums, find the contiguous subarray (containing at least one number)
which has the largest sum and return its sum.
A subarray is a contiguous part of an array.
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

A subarray is a contiguous part of an array maintaining the order of elements. $[1,2,3]$ is a subarray of $[1,2,3,4,5]$ but $[1,3,2]$ & $[1,3,5]$ are not.
Consider an array of positive integers only, the maximum sum subarray will be the entire array itself. If the array contains negative integers only, then the maximum sum subarray will be the maximum element of the array. It gets tricker when we have both positive and negative numbers.
If the array consists of positive integers(need not be all positive integers). Then there definitely exists a subarray which has positive sum, because I can choose one positive element in worst case and it would be the subarray with positive sum.
The main idea of Kadane's algorithm is to neglect the negative sum subarrays and take maximum among the positive sum subarrays.

- C++

Written by @ShivaRapolu01

```
class Solution {
public:
int maxSubArray(vector<int>& nums) {
int n = nums.size();
// globalSum is where the maximum sum of subarray is stored
// localSum is where the sum of current subarray is stored
int globalSum = INT_MIN, localSum = 0;
for (int i = 0; i < n; i++) {
// add current element to current sum
localSum = localSum + nums[i];
// if current sum is greater than globalSum, update globalSum
if (globalSum < localSum) {
globalSum = localSum;
}
// if upon adding ith element current sum is becoming less than 0
// it cannot contribute to the maximum sum subarray so we neglect it
// and reset our current sum to 0 to start another subarray freshly
if (localSum < 0) {
localSum = 0;
}
}
return globalSum;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
121 - Best Time to Buy and Sell Stock	Easy	N/A
152 - Maximum Product Subarray	Medium	N/A
918 - Maximum-sum-circular-subarray	Medium	N/A

Kadane's 2D Algorithm (Variation)

Overview

Kadane's 2D Algorithm is a variation of the original Kadane's algorithm that is used to find the maximum sum of a submatrix in a given 2D array. It is a powerful tool for solving problems related to image processing, such as finding the maximum sum of a sub-image in a larger image. The basic idea behind the algorithm is to first find the maximum sum of each row of the submatrix by using the original Kadane's algorithm, then find the maximum sum of all the rows by using the same algorithm again. The algorithm has a time complexity of $O(n^4)$ and is often used in conjunction with other techniques such as dynamic programming to improve the performance.

Algorithm

1. Any submatrix has 4 sides so we need 4 variables to uniquely identify and store the boundaries of the maximum sum submatrix.
2. Using the 1D kadane's algorithm we can find the maximum sum subarray in a 1D array and with some modifications we can retrieve the boundaries(starting index and ending index) of this maximum sum subarray.
3. We need to convert the submatrix into 1D array in such a way that we can identify the boundaries of maximum sum submatrix. For this we can try fixing the left and right boundaries (finalLeft and finalRight) of the submatrix and then we calculate the cummulative sum in each row and store it as 1D array.
4. Now we can apply 1D Kadane's algorithm on this 1D array to find the maximum sum subarray and we retrieve the boundaries of this maximum sum subarray.
5. The boundaries retrieved from this 1D Kadane's algorithm are the final Top and FinalBottom boundaries of the maximum sum submatrix in the original 2D matrix.

Consider the below problem statement:

Example 1: [0085 - Maximal Rectangle]

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

```
Input: matrix = [
["1","0","1","0","0"],
["1","0","1","1","1"],
["1","1","1","1","1"],
["1","0","0","1","0"]
]
Output: 6
```

If noticed properly we can see that maximum sum submatrix is the 6. Which is the submatrix enclosed by zero-indexed vertices $(1,2),(1,4),(2,2),(2,4)$.
Hint: The problem statement resonates with Kadane's algorithm. The main thing is how to extend the 1D Kadane's algorithm to 2D. General Kadane's algorithm works on a 1D array, so first we need to convert the submatrix into 1D array in such a way that we can uniquely identify the boundaries of maximum sum submatrix. Then we can apply Kadane's algorithm on this 1D array to find the maximum sum subarray.

- C++

Written by @ShivaRapolu01

```
class Solution {
public:
int Kadane(vector<int> arr, int &cursumLeft, int &cursumRight, int n) {
// function returns maxiumum sum of subarray and also updates
// the left and right indices of the subarray in cursumLeft and cursumRight respectively
int localSum = 0, globalSum = 0;
// variable to store the right index of current subarray
int localSumRight = 0;
// variable to store starting index of intermediate subarrays
int localCurStart = 0;
for (int i = 0; i < n; ++i) {
localSum += arr[i];
if (localSum < 0) {
localSum = 0;
localCurStart = i + 1;
} else if (localSum > globalSum) {
```

```
globalSum = localSum;
cursumLeft = localCurStart;
cursumRight = i;
}
}
return globalSum;
}
int maximalRectangle(vector<vector<char>>& matrix) {
int globalMaxSum = 0;
// variables to indicate maximum submatrix boundaries
int finalLeft, finalRight, finalTop, finalBottom;
int rows = matrix.size();
int cols = matrix[0].size();
const int INF = -(rows * cols);
// since there can be maximum of rows * cols number of 1's in the matrix
// if we encounter 0 in a row in the current submatrix,
// our maximum sum submatrix can't contain this row as it should contain only 1's.
// as we need to neglect the whole row,
// we need erase contributions of 1's in the same row which are before 0
// hence we add INF to temp array before passing it to Kadane's algorithm
// set the left column
for (int left = 0; left < cols; ++left) {
vector<int> temp(rows, 0);
// temp is used to store sum between current left and right boundaries for every row.
// set the right column corresponding to left
for (int right = left; right < cols; ++right) {
// calculate sum between current left and right for each row
for (int i = 0; i < rows; ++i) {
if (matrix[i][right]=='1') {
temp[i]+=1;
} else {
temp[i]+=INF;
}
}
```

```
// Find the maximum sum subarray in this created temp array using Kadane's 1D algorithm.
int localSum, localSumLeft, localSumRight;
localSum = Kadane(temp, localSumLeft, localSumRight, rows);
// compare sum with maximum sum so far.
// if sum is more, then update globalMaxSum
// and also update boundaries of maximum sum submatrix
if (localSum > globalMaxSum) {
globalMaxSum = localSum;
finalLeft = left;
finalRight = right;
finalTop = localSumLeft;
finalBottom = localSumRight;
}
}
}
return globalMaxSum;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
363 - Max Sum of Rectangle No Larger Than K	Hard	N/A

Linear Search

Author: @siddoinghisjob

Overview

Lets say we have a linear data structure - array, linked list - and we need to search for a certain element. We can use linear search here. In linear search we traverse the whole array and then while traversing we check for the particular item. If there's a match then we print that position(s).

To elaborate there are three main steps in performing linear search

1. Traverse the data structure, e.g. $a = [1, 2, 3, 4, 5]$
2. Check for the required element while traversing, e.g. $a[i] == 3$
3. Do something with the value at position i or $a[i]$

Here we can have three cases :

- Case I - Best Case*When the element we are looking for is at index-0 i.e., first position:*
  In this case we can break the loop as soon as we find the element and as here that is the 0 index, we can break the loop on the 0 index itself. This results in $O(1)$ time complexity.
- Case II - Average Case*When the element we are looking for is at middle position i.e., length/2 index:*
  In this case we will have to traverse the data structure for the half of length. This means that the time complexity is $O(n)$.
- Case III - Worst Case*When the element we are looking for is at last index:*
  Here we have traverse the whole data structure. In this case the time complexity is $O(n)$.

Example #1: 1295 - Find Numbers with Even Number of Digits

Here we are given an array and we are required to find out the numbers that have even number of digits.
We can think this problem as a linear search problem, where we are supposed to search for all the numbers that have evennumber of digits. Thats' it. Now to do so we will traverse the array and find the length of each number and check it for being even. To find the number of digits of a number, we will use some basic mathematical logic - we will take the number's log base 10 and add one to it i.e., $log10 + 1$. And if even length numbers are found then we will simply increase the count of even digits by 1.

- C++

Written by @siddoinghisjob

```
class Solution {
public:
int findNumbers(vector<int>& nums) {
// initially the answer is zero
int ans = 0;
for (int i = 0; i < nums.size(); i++) {
// calculate the length of the number using log10 function
int len = log10(nums[i]) + 1;
// check whether len is an even number
if (len % 2 == 0) {
ans++;
}
}
return ans;
}
};
```

Example: 2089 - Find target Indices after sorting array

You are given a 0-indexed integer array nums and a target element target.A target index is an index i such that nums[i] == target. Return a list of the target indices of nums after sorting nums in non-decreasing order. If there are no target indices, return an empty list. The returned list must be sorted in increasing order.
If we sort this array using library sorting functions then this problem is reduced to a simple linear search question. Where we are supposed to search for the target. Thats' it. Now to do so we will create a vetor for storing the answers, and the run a for loop and traverse every element and check it. If found then we will add that index to the vector. Finally we will return the vector and its' done.

This will take $O(nlogn)$ time complexity as we will be using sort function which takes $O(nlogn)$.

- C++

Written by @siddoinghisjob

```
class Solution {
public:
vector<int> targetIndices(vector<int>& nums, int target) {
// sorting the vector using stl function
sort(nums.begin(),nums.end());
// vector to store the required indices
vector<int> ans;
// linear searching to find the target
for (int i = 0; i < nums.size(); i++) {
if (nums[i] == target) {
ans.push_back(i);
}
}
return ans;
}
};
```

Suggested problems

Problem Name	Difficulty	Solution Link
1539 - Kth Missing Positive Number	Easy	View Solutions
1672 - Richest Customer Wealth	Easy	View Solutions
0540 - Single Element in a Sorted Array	Medium	View Solutions
0275 - H-Index II	Medium	N/A

Linked List

Author: @itsmenikhill

Overview

In this tutorial you will learn about Linked Lists, and its implementation using Java.

Problem with using Arrays was that we have to have some idea about the size of the array that we require. To counter this we learnt about dynamic arrays. Linked list is another approach to tackle this problem. In linked lists we do not have to worry about the size at all.
A linked list is a linear data structure that has a series of connected nodes. Each node has two fields, $data$ and an $address$. We call the start of a linked list, $head$. We can all it anything but by convention, we'll call it $head$.

Representation of a Linked List

Each node in a linked list contains:

- A data item
- Address of next node

Both of these items are wrapped together in a class:

- Java

Written by @itsmenikhill

```
class Node {
int data;
Node next;

public Node(){
this.data = data;
this.next = null;
}
}
```

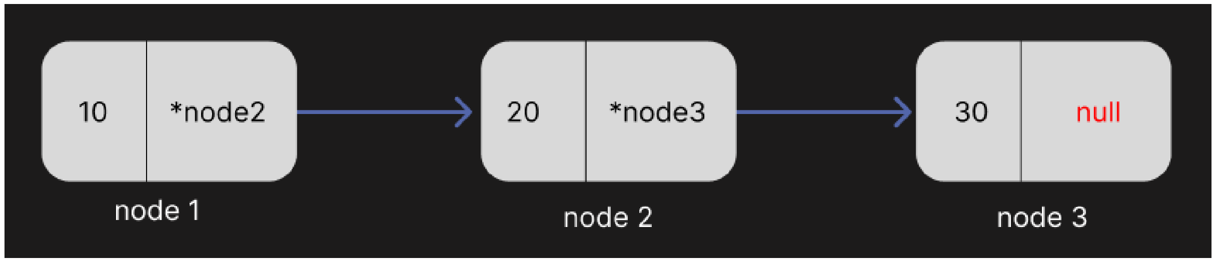
Now we will create a Linked list using this Node class:

- Java

```
class LinkedList {
Node node1 = new Node();
Node node2 = new Node();
Node node3 = new Node();
// set 10 as the value for 1st node
node1.data = 10;
// set node2 as the next node for node1
node1.next = node2;
// set 20 as the value for 2nd node
node2.data = 20;
// set node3 as the next node for node2
node2.next = node3;
// set 30 as the value for 3rd node
node3.data = 30;
// this is not required. By default next of a node is NULL
node3.next = null;

}
```

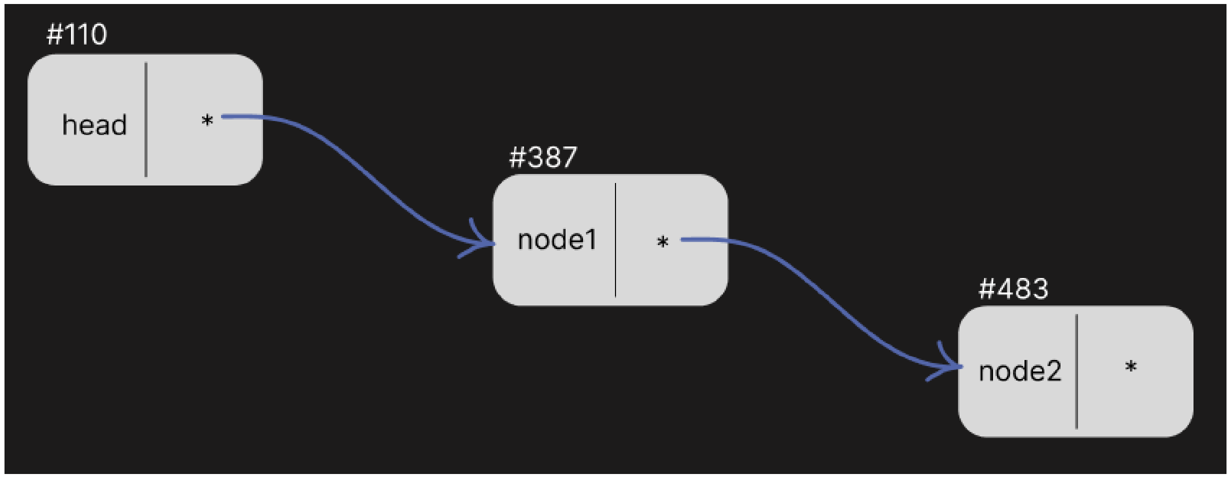
So now we have our first linked list.



The asterisk (*) signifies the address of the node, and not the value of that node.

Linked List in Memory

In arrays the elements are contiguous, which means they are placed one after the other in the memory. However, in linked lists, the elements are scattered in the memory.
When we declare an integer array of size 15, we would require 60 bytes of contiguous space in the memory to store the elements of that array. But in linked list we would only have to find 4 bytes to store the first element and from there, we look for another 4 bytes to store the next element. So, we see that the linked list items are scattered in memory unlike arrays.
That is the advantage linked lists have over array when it comes to memory. We have to look for smaller spaces to store the items, rather than chunks of contiguous space.



Numbers with # behind them are the addresses in memory where these nodes are stored. Here, $head$ stores the address for $node1$ and $node1$ stores the address for $node2$. Notice that the address are not sequential.

Basic Linked list operations

- Traversal
- Insertion

- Deletion
- Search

Let's see their implementation in a linked list.

Traversal

We have to access each element of the linked list. Remember, the $head$ points to the first node, and the $next$ pointer of the last node points to $null$.
Traversing through the linked list is fairly simple. We keep moving from the head towards the end of the list. We would know we have reached the last node when $next$ points to $null$.

- Java

Written by @itsmenikhill

```
// temp pointer that points to head initially
Node temp = head;
// check if next points to null
while (temp.next != null) {
// print the data stored in the current node
System.out.println(temp.data);
// move temp to the next node
temp = temp.next;
}
```

You are given the head of a linked list and a number. Check if the given number is present in the linked list or not. Return true if present, else return false.
We will traverse the list and at each node we will check if we have the required element in the current node or not. If we found the element, return true. If we have iterated over the list and not not found the number, we will return false.

- Java

Written by @itsmenikhill

```
public static boolean findElement(Node head, int target){
Node ptr = head;
while (ptr != null) {
// compare the node element and the target number
if (ptr.data == target) {
// number found
return true;
}
// move pointer to the next node
ptr = ptr.next;
}
// number not found
return false;
}
```

Inserting an element to the Linked list

We can add element to the beginning, middle or at the end of the linked list.

Inserting at the beginning

You are given a linked list $[3]$ -> $[4]$ -> $[5]$ -> $null$ and we have to add another node $[1]$ to the front. How would you do this?

Approach:

- Allocate memory for a new node
- Store the data
- Point $next$ of new node to $head$
- Point $head$ to the newnode

- Java

Written by @itsmenikhill

```
// allocate memory
Node newNode = new Node();
// store data
newNode.data = 1;
// point next of new node to head
newNode.next = head;
// make head point to new node
head = newNode;
```

Inserting at the middle

You are given a linked list $[3]$ -> $[4]$ -> $[5]$ -> $[6]$ -> $null$. Insert a new node $[7]$ at the 3rd place to make the final list as $[3]$ -> $[4]$ -> $[7]$ -> $[5]$ -> $[6]$ -> $null$.

Approach:

- Allocate memory for the new node
- Store the data in the new node
- Traverse to the node just before the required index
- Change the pointers
- Java

Written by @itsmenikhill

```
Node newNode = new Node();
newNode.data = 7;
Node temp = head;

// position at which we want to insert the new element
```

```
int pos = 3;

// loop runs only for i=2. when i=3, pos=3 as well, hence loop terminates.
for (int i = 2; i < pos; i++) {
if(temp.next!=null) {
temp = temp.next;
}
}
// here temp points to the 2nd node
newNode.next = temp.next;
// new node inserted after temp.
temp.next = newNode;
```

One important thing to note here is that before breaking any connection in the linked list, we must first make the new connections. Make first, break later.

Inserting at the end

You are given a linked list $[3]$ -> $[4]$ -> $[5]$ -> $[6]$ -> $null$. Insert a new node $[7]$ at the end to make the final list as $[3]$ -> $[4]$ -> $[5]$ -> $[6]$ -> $[7]$ -> $null$

Approach:

- Allocate memory for the new node
- Store the data in the new node
- Traverse till the end of the list
- Make the $next$ of the last node point to the new node
- Java

Written by @itsmenikhill

```
Node newNode = new Node();
newNode.data = 7;
```

```
Node temp = head;

// when temp.next is null, we'll know we are at the last node
while (temp.next != null) {
// move pointer to the next node
temp = temp.next;
}

// temp is now at the last node.
// point $next$ of the last node to the new node
temp.next = newNode;
```

Deleting an element from the Linked list

We can delete the first node, or the last node, or some other position in the middle.

Deleting from the beginning

To delete from the beginning we just have to move the $head$ to its next, so that no pointer would be pointing to the first node.

- Java

Written by @itsmenikhill

```
// change the pointer from the head node, to the next node.
head = head.next;
```

Deleting from the end

- Move the pointer to the second last node
- Set the $next$ pointer of second last node to point to $null$
- Java

Written by @itsmenikhill

```
Node temp = head;
// move the pointer to the second last node
while (temp.next.next != null) {
// move the current pointer to the next node
temp = temp.next;
}
temp.next = null;
```

Deleting from any position in the middle

- Traverse to the element just before the node to delete
- Change the next of this node to point to next of the node to delete
- Java

Written by @itsmenikhill

```
Node temp = head;
// we are running the loop from 2 node because we have to move to the node
// just before the node we want to remove
for (int i = 2; i < pos; i++) {
// change pointer from current node to the next node
temp = temp.next;
}
temp.next = temp.next.next;
```

Search for an item in the list

- Make a temporary pointer ptr point to $head$
- Move ptr to next node until ptr is $null$

- At each iteration, check if the data in ptr is same as the number we want. If yes, then return $true$
- Return $false$ if number not found
- Java

Written by @itsmenikhill

```
Node ptr = head;
while (ptr != null) {
// check if data in current node matches the number we are looking for
if (ptr.data == num) {
// return true if number found
return true;
}
// move ptr to the next node
ptr = ptr.next;
}
// return false if number not found
return false;
```

Complexity Analysis

Operation		
Complexity		
Explanation		
Look up	$$O(N)$$	We will have to iterate from head till the element we want
Insertion at beginning	$$O(1)$$	Simply change the head pointer
Insertion at the end	$$O(N)$$	Move from the head to the last item then change the pointers
Deletion from the beginning	$$O(1)$$	Simply change the head pointer
Deletion from the end	$$O(N)$$	Move from the head to the node just before the item you want to delete

Example: [0237 - Delete Node in a Linked List]

There is a singly linked list. You have to delete a $node$ from the list. You are given the $node$ to delete but not the $head$ of the > list. Delete the given $node$. Note that by deleting the $node$, we $do not$ mean removing it from memory. We mean:

- The value of the given node should not exist in the linked list.
- The number of nodes in the linked list should decrease by one.
- All the values before node should be in the same order.
- All the values after node should be in the same order.

Change the value of the $current node$ to the value of the $next$ node. Do this until the last node.

- Java

Written by @itsmenikhill

```
class Solution {
// we are given a node to delete
public void deleteNode(ListNode node) {
// check if the current node is second last node
while(node.next.next!=null){
// change the value of current node to the value of next node
node.val = node.next.val;
// move node to the next node
node=node.next;
}

// we are at the second last node
// change value of the last node to the value of next node
node.val = node.next.val;
// change the next of last node to null
node.next = null;
}
}
```

Example: 0019 - Remove Nth Node From End of List

Given the head of a linked list, remove the nth node from the end of the list and return its head. Given Linked list: $[1]$ -> $[2]$ -> $[3]$ -> $[4]$ -> $[5]$ for $n = 2$ change the list to $[1]$ -> $[2]$ -> $[3]$ -> $[5]$.

Approach

- First we find the size of the list
- nth node from the end is the $size - (n + 1)th$ node from the $front$
- Once we have the size, we can iterate over the list till the node just before the node to remove
- Change the pointers to remove the node
- Java

Written by @itsmenikhill

```
class Solution {
public ListNode removeNthFromEnd(ListNode head, int n) {
ListNode ptr = head;
// find the size
int size = findSize(head);
ptr = head;

// if size is equal to the n, remove node at head
if (size == n) {
ptr = ptr.next;
head = ptr;
return head;
}
// move ptr to the node just before the node to remove
for (int i = 0; i< size - n - 1; i++) {
ptr = ptr.next;
}

// check if the node to remove is the last node
if (ptr.next.next != null){
ptr.next = ptr.next.next;
} else {
ptr.next=null;
}
return head;
}
// method to find the size of the list
public int findSize(ListNode head) {
// temporary pointer at head
ListNode ptr = head;
int size = 0;
// increase the size till we reach the end of the list
while (ptr != null) {
size += 1;
ptr = ptr.next;
}
return size;
}
}
```

Example: 0234 - Palindrome Linked List

Given the $head$ of a singly linked list, return $true$ if it is a $palindrome$ or $false$ otherwise.

Approach

- We use $two pointer$ method and $recursion$ to solve this problem
- We keep a $global$ left pointer, that points to the current left node
- As we go deep in recursion we move our $right$ pointer forward towards the end of the list
- When $right$ pointer is at the end we compare its value to the $left$ node
- If they are same we move $left$ pointer forward and come out of the recursion
- As we come out of the recursive call, out $right$ pointer would move towards $left$
- If at any point the values are not same, we return $false$
- Java

Written by @itsmenikhill

```
class Solution {
// global left pointer
static ListNode left;
public boolean isPalindrome(ListNode head) {
// point the left to head
left = head;
boolean b = check(head);
return b;
}
// recursive method that moves pointer towards right and compares values
private boolean check(ListNode right){
// if right is null, we have reached the end of the list
if(right == null){
return true;
}
boolean b = check(right.next);
// if at any point b is false, we return false. If this happens even once,
// no further comparisons would happen and each recursive call would return false
if(b == false){
```

```
return false;
} else {
// else compare the values
// if values are equal, move left towards right and return true to the previous call
if (left.val == right.val) {
left = left.next;
return true;
} else {
// if values not same, return false
return false;
}
}
}
}
```

Example: 0328 - Odd Even Linked List

Given the $head$ of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.
The $first$ node is considered odd, and the $second$ node is $even$, and so on.
Note that the relative order inside both the even and odd groups should remain as it was in the input.
You must solve the problem in O(1) extra space complexity and O(n) time complexity.

Approach

- We maintain three pointers, odd at the first node, $even$ at the second node and $evenhead$ also at the second node.
- $evenhead$ will not be changed. It will point to the starting of the list of even nodes.
- The node after every even node, is an odd node.

- So, $next$ node for current odd node, would be $next$ node of the current $even$ node.
- And $next$ node for the current $even$ node, would be $next$ node of the just changed odd node.
- Java

Written by @itsmenikhill

```java
class Solution {
public ListNode oddEvenList(ListNode head) {
// we check if there are at least three nodes in the list
// if there are only two nodes, then first node is even and second node is odd
// if only one node, then first node is odd
// so just return head
if (head == null || head.next == null || head.next.next == null) {
return head;
}
// odd is at head
ListNode odd = head;
// make second node as even
ListNode even = head.next;
// keep a pointer evenhead at even. This will not be changed.
ListNode evenHead = even;
// we have to move odd to next of even
// so we check is even.next is null of even is null
while (even != null && even.next != null) {
// if not, then make odd.next point to even.next
odd.next = even.next;
// move odd to odd.next
odd = odd.next;
// even.next to odd.next
even.next = odd.next;
// move even to even.next
}

// at this point we have connected all the even nodes together, and all odd nodes together
// now we connect the odd nodes, and even nodes
odd.next = evenHead;
return head;
}
}
```

Suggested Problems

Problem Name	Difficulty	Solution Link
206. Reverse Linked List	Easy	N/A
86. Partition List	Medium	N/A
148. Sort List	Medium	N/A

MOD (1e9 + 7)

Author: @tannudaral

Overview

When the answer to a problem is a very large number, problem setters expect you to output it "modulo m", that is, the remainder after dividing the answer by m (for example, "modulo $1e9 + 7$"). So if the actual answer is very large, with the use of modulo m, it would be sufficient to use the data types int and long long. Since many languages do not support large-integer arithmetic, this method avoids integer overflow. The task of modulo operator $\%$, also know as the remainder operator, is to give the remainder. We denote it by $x\,mod\,m$, the remainder when x is divided by m. For example, 17 mod 5 $=$ 2 because 17 $=$ $3*5 + 2$.

Modular Arithmetic

An important property of the modulo is that in addition, subtraction and multiplication, the remainder can be taken before the operation:

Addition

$(a + b)\,mod\,m$ $=$ $(a\,mod\,m + b\,mod\,m)\,mod\,m$

Subtraction

The remainder should usually fall between $0....m−1$. However, in C++ and other languages, a negative number's remainder is either zero or negative. An easy way to make sure there are no negative remainders is to add m to the result. It is only needed when there are subtractions in the code and the remainder may become negative.
$(a\,−\,b)\,mod\,m$ $=$ $(a\,mod\,m\, − \,b\,mod\,m\,+\,m)\,mod\,m$

Multiplication

$(a *b)\,mod\,m$ $=$ $(a\,mod\,m* b\,mod\,m)\,mod\,m$

Division

The modular division is completely different from modular addition, subtraction and multiplication. It also does not always exist. It requires a concept called the "Modular Multiplicative Inverse". The modular multiplicative inverse of a number a is the number $a^{−1}$ such that $a · a^{−1} \,mod\, m = 1$. You may notice that this is similar to the concept of a reciprocal of a number, but here we don't want a fraction; we want an integer, specifically an integer between 0 and $m−1$ inclusive.
There are twofaster ways to calculate the inverse:

- Extended GCD algorithm
- Fermat's little theorem

The extended GCD algorithm may be more versatile and sometimes faster, but Fermat's little theorem method is more popular, since it's almost free once you implement exponentiation, which we will cover here.
Fermat's little theorem says that provided the modulus m is a prime number (10^9+7 is prime) then $a^{m}\,mod\,m=a\,mod\,m$. Working backwards, $a^{m−1}\,mod\,m = 1 = a · a^{m−2}\, mod\, m$, therefore the number we need is $a^{m−2}\, mod\, m$. Hence, we can calculate the modular multiplicative inverse $a^{−1}$ using $a^{−1} = a^{m−2}\, mod\, m$ when m is prime. We can now define the division operator as:
$$ (a\, /\, b) \,mod\, m = (a\, mod\, m\, * b^{-1} \,mod \,m)\, mod\, m $$

Example

Let's understand this with the factorial of a number program. The following code calculates $n!$, the factorial of n, modulo m:

- C++

```
int factorial(int n) {
int M = 1e9 + 7;
long long fact = 1;
for (int i = 2; i <= n; i++) {
// WRONG APPROACH
// Here, fact may exceed 2 ^ 64 - 1
fact = fact * i;
}
return fact % M;
}
```

Thus, we can take the remainder after every operation and the numbers will never become too large.

- C++

```
int factorial(int n) {
int M = 1e9 + 7;
long long fact = 1;
for (int i = 2; i <= n; i++) {
// Here, fact never exceeds 10 ^ 9 + 7
fact = (fact * i) % M;
}
return fact;
}
```

Why 1e9 + 7?

The number $1e9 + 7$ fits nicely into a signed 32-bit integer. It is also the first 10-digit prime number. In some problems we need to compute the Modular Multiplicative Inverse and it helps very much that this number is prime.

In fact any prime number less then $2^{30}$ will be fine in order to prevent possible overflows. But this one can be easily written as $1e9 + 7$. This reasoning almost uniquely determined this number.

References

1. Competitive Programmer's Handbook
2. Fermat's Little Theorem

Prefix Sum

Author: @wingkwong

Overview

The prefix sum is a technique used to efficiently calculate the sum of all elements in an array up to a certain index. It is also known as cumulative sum, and it is often used in various computational problems such as range sum queries or dynamic programming.
The basic idea behind the prefix sum is to pre-compute the sum of all elements up to each index in the array and then use these pre-computed sums to quickly calculate the sum of any sub-array in the array.
The steps for implementing the prefix sum technique are as follows:

1. Create a new array of the same length as the original array, and initialize the first element to the value of the first element of the original array.

2. Starting from the second element, iterate through the rest of the original array, and at each element, calculate the prefix sum by adding the current element to the previous prefix sum, and store this value in the corresponding element of the new array.
3. To find the sum of any sub-array, we can use the pre-computed prefix sum array, by subtracting the prefix sum of the starting index of the sub-array from the prefix sum of the ending index + 1.

The prefix sum has a time complexity of O(n) and a space complexity of O(n), it is efficient and widely used in various computational problems such as range sum queries, dynamic programming and more.
Let's say the input a is $[1, 2, 3, 4, 5]$. The prefix sum array $pref$ would be $[1, 3, 6, 10, 15]$ which can be calculated as follows:
$$$ pref[0] = a[0] \ pref[1] = a[0] + a[1] \ pref[2] = a[0] + a[1] + a[2] \ ... $$$
We can notice that $pref[i]$ is the previous value $pref[i - 1]$ plus the input $a[i]$ starting from $i = 1$, which can be illrustrated as follows:
$$$ pref[0] = a[0] \ pref[1] = pref[0] + a[1] \ pref[2] = pref[1] + a[2] \ ... $$$
To generalise, we have
$$$ pref[i] = \begin{cases} a[0], & \text{if i is 0} \ pref[i - 1] + a[i], & \text{if i >= 1} \end{cases} $$$

- C++

Written by @wingkwong

```
vector<int> generatePrefixSum(vector<int>& a) {
int n = a.size();
vector<int> pref(n);
pref[0] = a[0];
for (int i = 1; i < n; i++) pref[i] = pref[i - 1] + a[i];
}
```

Example : 1480 - Running Sum of 1d Array

```
Given an array nums. We define a running sum of an array as runningSum[i] = sum(nums[0] ...
nums[i]).
Return the running sum of nums.
Input: nums = [1,2,3,4]
Output: [1,3,6,10]
Explanation: Running sum is obtained as follows: [1, 1 + 2, 1 + 2 + 3, 1 + 2 + 3 + 4]
```

Let's start with a brute force solution, we iterate each element $a[i]$ and we iterate from $j = [0 .. i]$ to add $a[j]$ to sum. This solution is acceptable but it is slow as we have two for-loops here.

- C++

Written by @wingkwong

```
class Solution {
public:
vector<int> runningSum(vector<int>& nums) {
int n = nums.size();
vector<int> ans(n);
for (int i = 0; i < n; i++) {
int sum = 0;
for (int j = 0; j <= i; j++) {
sum += nums[j];
}
ans[i] = sum;
}
return ans;
}
};
```

However, if we utilise the idea of Prefix sum, we know the result at some point has been calculated. Therefore, we can just do it in a $O(n)$ way.

- C++

Written by @wingkwong

```
class Solution {
public:
vector<int> generatePrefixSum(vector<int>& a) {
int n = a.size();
vector<int> pref(n);
pref[0] = a[0];
for (int i = 1; i < n; i++) pref[i] = pref[i - 1] + a[i];
return pref;
}
vector<int> runningSum(vector<int>& nums) {
return generatePrefixSum(nums);
}
};
```

As we don't actually need $pref$ for further process in this question, we can just write it inline instead.

- C++

Written by @wingkwong

```
class Solution {
public:
vector<int> runningSum(vector<int>& nums) {
for (int i = 1; i < nums.size(); i++) {
nums[i] += nums[i - 1];
}
return nums;
}
};
```

Prefix Sum is useful when we want to find the sum of all elements in a given range or something related to subarray problems. Besides, it doesn't have to be sum. We can make it like product ($pref[i] = pref[i - 1] * a[i]$) or even XOR ($pref[i] = pref[i - 1] \oplus a[i]$).

Example: 0303 - Range Sum Query - Immutable

```
Given an integer array nums, handle multiple queries of the following type:
Calculate the sum of the elements of nums between indices left and right inclusive where
left <= right.
Implement the NumArray class:
NumArray(int[] nums) Initializes the object with the integer array nums.
int sumRange(int left, int right) Returns the sum of the elements of nums between indices
left and right inclusive (i.e. nums[left] + nums[left + 1] + ... + nums[right]).
Input
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
Output
[null, 1, -1, -3]
Explanation
NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3
```

Sometimes we may pad a zero as the first element in prefix sum as we want to exclude the first element. For example, let's say we have an input $[1, 2, 3, 4, 5]$, the prefix sum array would be $[0, 1, 3, 6, 10, 15]$. In this case, we can write as follows:

- C++

Written by @wingkwong

```
vector<int> generatePrefixSum(vector<int>& a) {
int n = a.size();
vector<int> pref(n + 1);
for (int i = 0; i < n; i++) pref[i + 1] = pref[i] + a[i];
return pref;
}
```

Given l and r, if we want to calculate the sum of the elements of $nums$ between l and r inclusive. The answer is simply $pref[r + 1] - pref[l]$.
Let's say we have an input $[a,b,c,d]$ and $pref$ would be $[0, a, a+b, a+b+c, a+b+c+d]$. Supposing we want to calculate the sum for the last three elements (i.e. $l = 1, r = 3$), it is easy to see the answer is $b + c + d$.
If we use $pref$ to calculate, that would be
$$$ rangeSum(l, r) = pref[r + 1] - pref[l] \ rangeSum(1, 3) = pref[4] - pref[1] \ = (a + b + c + d) - (a) \ = b + c + d $$$

- C++

Written by @wingkwong

```
class NumArray {
public:
vector<int> pref;
vector<int> generatePrefixSum(vector<int>& a) {
int n = a.size();
vector<int> pref(n + 1);
for (int i = 0; i < n; i++) pref[i + 1] = pref[i] + a[i];
return pref;
}
NumArray(vector<int>& nums) {
pref.resize(nums.size() + 1);
pref = generatePrefixSum(nums);
}
int sumRange(int left, int right) {
return pref[right + 1] - pref[left];
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
1480 - Running Sum of 1d Array	Easy	View Solutions
0303 - Range Sum Query - Immutable	Easy	N/A
1004 - Max Consecutive Ones III	Medium	View Solutions
0974 - Subarray Sums Divisible by K	Medium	View Solutions

Queue & Stack

Author: @heiheihang

Stack

Stack is the data structure The first item that comes in will be the first to go out. Let's look at this question (LeetCode Link)
Given a string scontaining just the characters '(', ')', '{', '}', '['and ']', determine if the input string is valid.
An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

To validate a string of parentheses is valid, we must have a corresponding opening bracket when we see a closing bracket. Lets look at some examples:

```
valid_string_1 = "([])"
```

We can process the string in the following way:

```
#1st character from valid_string_1
valid_string_1_step_1 = ["("]
```

We have the first character first. We do not need to do anything if it is an opening bracket.

```
#2nd character from valid_string_1
valid_string_1_step_2 = ["(", "["]
```

We have the second character now. Again, we do not need to do anything if it is an opening bracket.

```
#3rd character from valid_string_1
valid_string_1_step_3a = ["(", "[", "]"]
```

Here, we see a matching pair "["and "]", so we can cancel them out.

```
#3rd character from valid_string_1 (removing valid pair)
valid_string_1_step_3b = ["("]
```

We have removed the two last seen elements form the stack.

```
#4th character from valid_string_1 (after removing 2nd and 3rd)
valid_string_1_step_4a = ["(", ")"]
```

After adding the fourth character, we see a matching pair between the 1st character and the 4th character. We can remove it now.

```
#4th character from valid_string_1 (after removing 1st and 4th)
valid_string_1_step_4a = []
```

With no more character to process and with an empty stack, we know that there are no remaining opening and closing brackets. Hence, we can validate this string.
Now let's look at a counter example:

```
invalid_string_1 = "[()}"
```

Similar to the scenario above, we can skip the first two characters and have:

```
invalid_string_1_step3a = ["[", "(", ")"]
```

We see a matching pair in the 2nd character and the 3rd character, so we can remove them.

```
invalid_string_1_step3b = ["["]
```

Now we look at the 4th character:

```
invalid_string_1_step4 = ["[", "}"]
```

We have a closing bracket, and it is not matching its corresponding opening bracket ("{"), so we know that this string is invalid.

Are you able to code the solution out after looking at these two examples?

```
class Solution:
def isValid(self, s: str) -> bool:
#we use a stack to keep track of brackets
stack = []
#iterate the characters in the string
for c in s:
#we store the character in the stack if it is an opening bracket
if(c == "(" or c == "[" or c == "{"):
stack.append(c)
#we check if there is a matching opening bracket
#when we see a closing bracket
elif(c == ")"):
if(len(stack) == 0 or stack[-1] != "("):
return False
else:
stack.pop()
#we check if there is a matching opening bracket
#when we see a closing bracket
elif(c == "]"):
```

```
if(len(stack) == 0 or stack[-1] != "["):
return False
else:
stack.pop()
#we check if there is a matching opening bracket
#when we see a closing bracket
elif(c == "}"):
if(len(stack) == 0 or stack[-1] != "{"):
return False
else:
stack.pop()
#if the stack is not empty, there are some unmatched opening brackets
#this suggests it is not valid
if(len(stack) != 0):
return False
return True
```

Queue

Queue is the data structure that is First-In-First-Out. The first person who enters the queue should be the first person to leave the queue.
We can look at the following problem (LeetCode Link)
As the problem statement and examples are quite long, we kindly ask you to read them on LeetCode.
To implement a Queue, we must have a data structure that handles adding element on the left in $$O(1)$$ __ time. These are the options in different languages:

- Python: queue = deque([]) # queue.appendleft(x) , queue.pop()
- C++: queue<int> q; // q.push(x), q.pop()

We can use these data structures to simulate this problem

```
class Solution:
```

```
def timeRequiredToBuy(self, tickets: List[int], k: int) -> int:
queue = deque([])
#we initialize the queue first
#as we need to keep track of the kth person, we add the index as well
for i in range(len(tickets)):
ticket_needed = tickets[i]
queue.appendleft([i,ticket_needed])
#keep track of overall time
time = 0
#we continue until queue is empty
while(queue):
#each person spends one second for purchase
time += 1
#access the first person and her ticket needed from the queue
first_person_in_queue, tickets_remaining = queue[-1]
#the first person buys one ticket , so she needs one less ticket
tickets_remaining -= 1
#we remove the first person from the queue
queue.pop()
#if the first person still needs to buy more tickets, we move her back to the end of the
queue
if(tickets_remaining != 0):
queue.appendleft([first_person_in_queue, tickets_remaining])
#if the first person is the target and she has bought all the tickets, we return the time
else:
if(first_person_in_queue == k):
return time
return time
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0155 - Min Stack	Easy	View Solutions
0496 - Next Greater Element I	Easy	View Solutions
1475 - Final Prices With a Special Discount in a Shop	Easy	View Solutions

Sliding Window

Authors: @heiheihang@wingkwong

Overview

Sliding Window is a technique used for iterating through a finite data set, typically an array, in a specific and controlled way. It involves creating a window, which is a subset of

the data, that "slides" through the larger data set, typically one element at a time, while performing a specific operation on each subset of the data.
The size of the window and the number of elements it moves at each step can be adjusted to suit the needs of the specific problem being solved. The technique is commonly used in algorithms that involve finding patterns or trends in data, such as finding the maximum/minimum value in a set of data, or counting the number of occurrences of a specific element.
Sliding window can be applied in various problems such as:

- Finding the maximum/minimum value in a set of data.
- Counting the number of occurrences of a specific element.
- Finding the longest substring without repeating characters.
- Finding the maximum sum of a sub-array of size k.

Overall, the sliding window technique is a useful approach for solving specific types of problems that involve iterating through a data set in a controlled way, such as in pattern matching, data analysis, and statistics. It allows for an efficient and controlled iteration of a data set, which can lead to improved performance and more accurate results.

Example 1: 1876 - Substrings of Size Three with Distinct Characters

A string is goodif there are no repeated characters.
Given a string s, return *the number of good substringsof length threein*s.
Note that if there are multiple occurrences of the same substring, every occurrence should be counted.
A substringis a contiguous sequence of characters in a string.
For example, with this input:

```
s = "xyzzaz"
```

The substrings with length of 3 are:

```
s1 = "xyz" #index 0-2
s2 = "yzz" #index 1-3
s3 = "zza" #index 2-4
s4 = "zaz" #index 3-5
```

Among these substrings, the only substring with distinct characters is "xyz".

In this problem, we need to keep a *window*of substrings of length 3.

We can use the following strategy:

- left_pointerto keep track of the left character of the substring length of $3$
- right_pointerto keep track of the right character of the substring length of $3$
- We check if the following characters are unique:
    - s[left_pointer]
    - s[left_pointer + 1]
    - s[right_pointer]

Let's take a look at the following solution:

- Python
- C++
- Java

Written by @heiheihang

```
def countGoodSubstrings(self, s: str) -> int:
# two pointers to keep track of sliding window
left_pointer = 0
right_pointer = 2
```

```
unique_substring_count = 0
# when the sliding window is within s
while (right_pointer < len(s)):
# we declare the 3 characters in the sliding window
first_char = s[left_pointer]
second_char = s[left_pointer + 1]
third_char = s[right_pointer]
# if all characters are unique, add 1
if (first_char != second_char and first_char != third_char and second_char != third_char):
unique_substring_count += 1
# shift the sliding window right
left_pointer += 1
right_pointer += 1
# return result
return unique_substring_count
```

In this problem, the size of the sliding window is constant. There are harder problems with varying sliding window size, but you need to learn Hash Mapfirst.

Suggested Problems

Problem Name	Difficulty	Solution Link
1852 - Distinct Numbers in Each Subarray	Medium	View Solutions
1004 - Max Consecutive Ones III	Medium	View Solutions
1876 - Substrings of Size Three with Distinct Characters	Medium	N/A

Sorting Introduction

Author: @wingkwong

Overview

Sorting refers to rearranging elements in a specific order. The most common order is either ascending or descending. There are a lot of algorithms to sort the array with different time complexity.
In C++, if define a static array of N elements of type int such as $$a[4]$$ you can sort like as below where $$N$$ is the number of elements to be sorted.

```
sort(a, a + N);
```

If you want to sort for a specific range $$[x, y)$$, then use

```
sort(a + x, a + y);
```

For dynamic array, we do in such way

```
sort(a.begin(), a.end());
```

If you want to sort for a specific range $$[x, y)$$, then use

```
sort(a.begin() + x, a.begin() + y);
```

To sort in an decreasing order,

```
sort(a.begin(), a.end(), greater<int>());
```

By default, sort()sorts the elements in the range $$[x, y)$$ into ascending order. If the container includes pairs, tuples or array\<int, N>, it first sorts the first element, then the second element if there is a tie and so on. For example, the following comparator is same as sort(a.begin(), a.end());.

```
sort(a.begin(), a.end(), [&](const array<int, 3>& x, const array<int, 3>& y) {
return (
(x[0] < y[0]) ||
(x[0] == y[0] && x[1] < y[1]) ||
(x[0] == y[0] && x[1] == y[1] || x[2] < y[2])
);
});
```

Suggested Problems

Problem Name	Difficulty	Solution Link
[0921 - Sort an Array]()	Medium	[View Solutions]()

Bubble Sort

Author: @RadhikaChhabra17

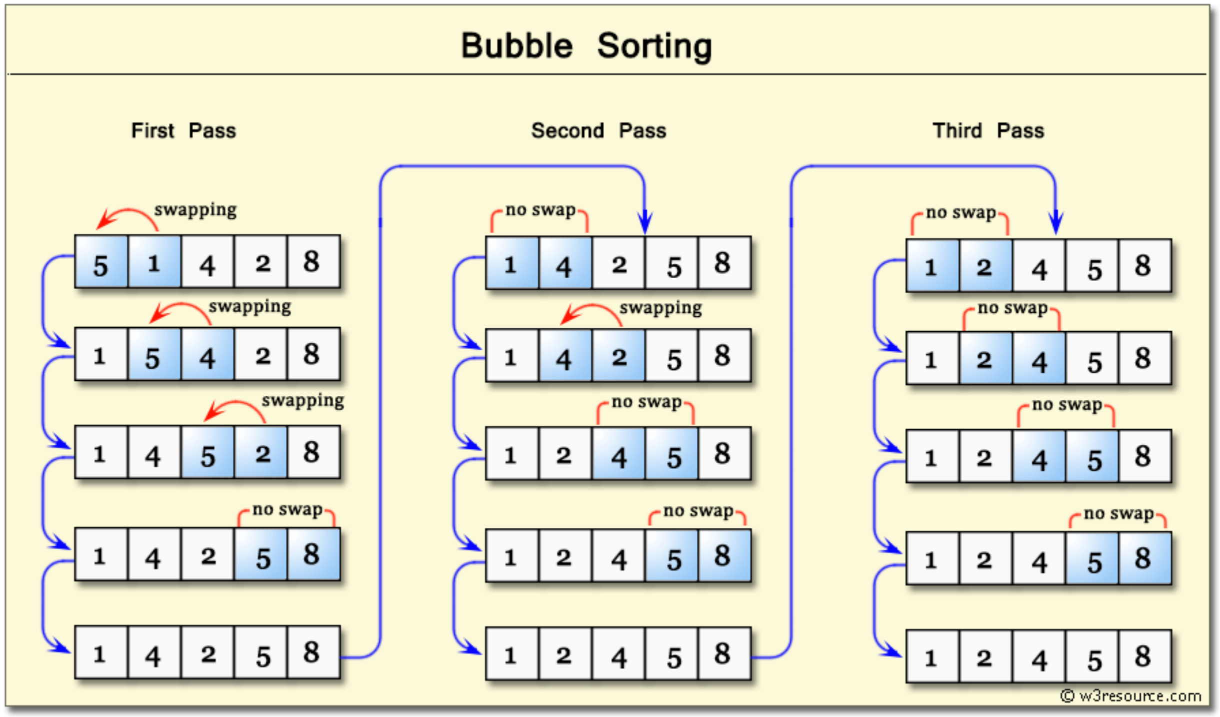
Contributor: @wingkwong

Overview

Bubble Sort is a sorting algorithm which compares the adjacent elements and swap their positions if they are placed in wrong order. At max, we need to compare adjacent elements for $$(n - 1)$$ iterations where n is the size of array to be sorted. At the end of each iteration, larger (or smaller, as required) value is sorted and placed at correct positions.
The language that we are using is C++, please refer to your own language of preference if needed.

Algorithm

Make two nested loop, the 1st loop would be for number of pass the algorithm woud run, total $n-1$ passes and second would be for comparison in that pass. In each pass, one element is sorted (largest or smallest, as required) and placed in correct position and rest are compared in further passes.
If one element is bigger (or smaller in decreasing order case) than its next element, then both should be swapped.
Consider the example of unsorted list and see how the algorithm works.
$$ arr = \{5, 1, 4, 2, 8\} $$



We can use a variable $check$ to see if there is swap in one pass or not. If there is no swapping in one pass, they we don't have to check for other pass.

- C++

Written by @RadhikaChhabra17

```
void bubblesort(vector<int> &arr) {
int n = arr.size();
bool check = true;
for (int i = 0; i < n - 1 && check; i++) {
check = false;
for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
swap(arr[j], arr[j + 1]);
check = true;
}
}
}
}
```

Complexity Analysis

Time Complexity

For first iteration will run $$(n-1)$$ times. For the second one, it will run $$(n-2)$$ times and so on.
Therefore, the Time Complexity for the worst case $= (n - 1) + (n - 2) + (n - 3) + ... + 1 = O(n^2)$
Use of variable $check$ will reduce the time complexity further as if there is no change in any iterations, next iterations will not occurs, this reduces the time complexity.

Space Complexity

Since there is no extra space, Space Complexity = $$O(1)$$. This shows that it is an inline sorting.

Example: 2164. Sort Even and Odd Indices Independently

You are given a 0-indexed integer array nums. Rearrange the values of nums according to the following rules:

1. Sort the values at odd indices of nums in non-increasing order. For example, if nums = [4,1,2,3] before this step, it becomes [4,3,2,1] after. The values at odd indices 1 and 3 are sorted in non-increasing order.

2. Sort the values at even indices of nums in non-decreasing order. For example, if nums = [4,1,2,3] before this step, it becomes [2,1,4,3] after. The values at even indices 0 and 2 are sorted in non-decreasing order.

Return the array formed after rearranging the values of nums.

In this problem, we can run two bubble sort. One for even indices with non decreasing order and one for odd indices with non increasing order.

- C++

Written by @RadhikaChhabra17

```
class Solution {
public:
vector<int> sortEvenOdd(vector<int>& nums) {
// work for even indices
for (int i = 0; i < nums.size(); i += 2){
for (int j = i + 2; j < nums.size(); j += 2){
// sort in non-decreasing order
if (nums[i] > nums[j]){
swap(nums[i], nums[j]);
}
}
}
// work for odd indicies
for (int i = 1; i < nums.size(); i += 2){
for (int j = i + 2; j < nums.size(); j += 2){
// sort in non-increasing order
if (nums[i] < nums[j]){
swap(nums[i], nums[j]);
}
}
}
return nums;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0075 - Sort Colors	Medium	View Solutions
0148 - Sort List	Medium	View Solutions

Cyclic Sort

Author: @prishit55

Contributor: @wingkwong

Overview

Cyclic Sort is an in-place, unstable-sorting algorithm.

- in-place: An in-place algorithm transforms input without using any other auxiliary data structure. As the algorithm executes the input is overwritten by the output.

- unstable: Unstable sorting algorithm don't preserve the relative order of equal elements while sorting.

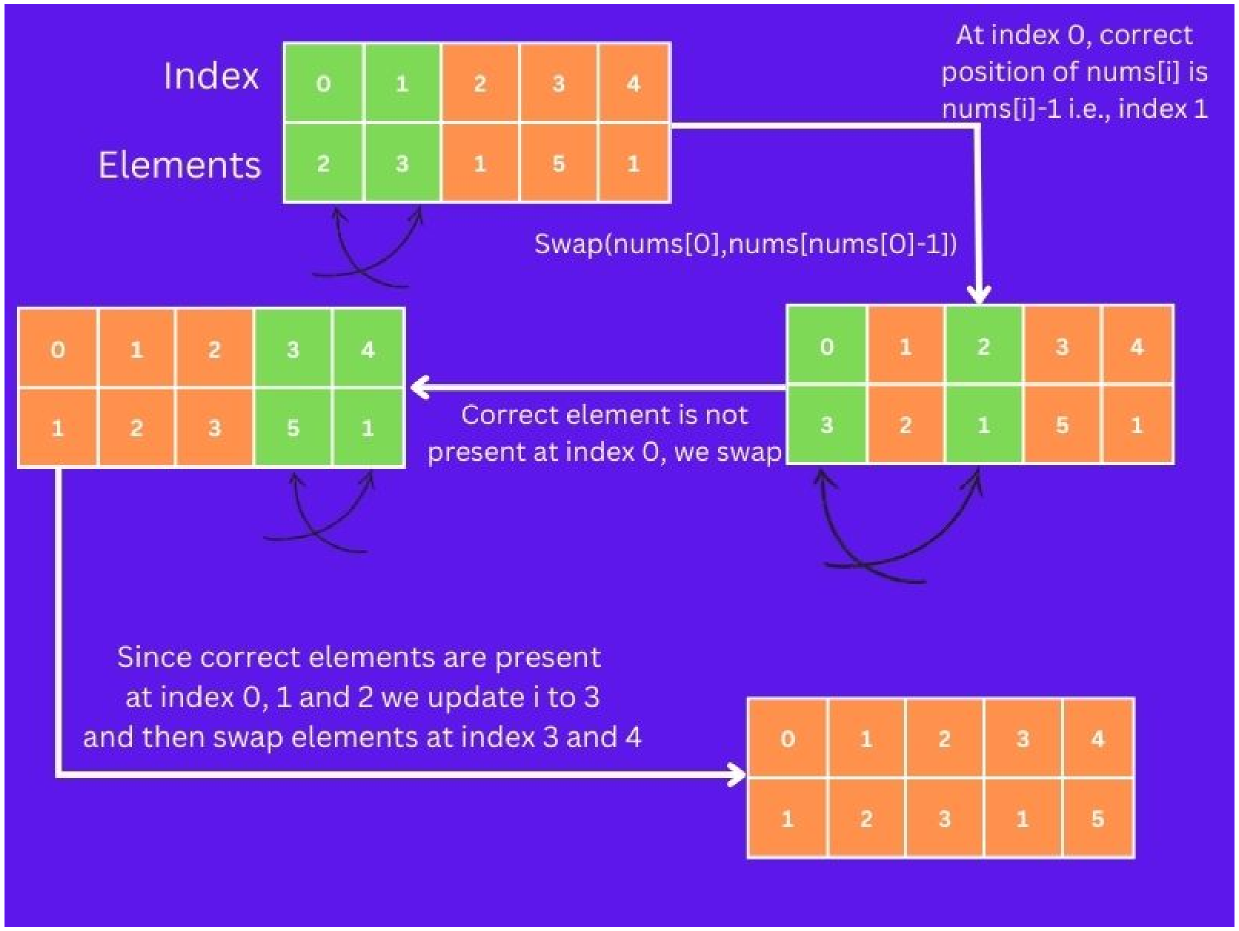
Cyclic Sort algorithm factors the permutation to be sorted into number of cycles, which are individually rotated to give desired sorted result.

Algorithm

Consider an array with n distinct elements. For any element x we can find the index at which it will occur in the sorted array by counting the number of elements smaller than x in the entire array. Now,

- if the element is at the correct position then do nothing
- else, write the element to its intended position. That position must be inhabited by a different number y which has to be moved to its correct position. The process continues until we get an element at the original position of x.

This completes one cycle. Repeating the cycle for every element will generate a sorted array.


Index
Elements
0 1 2 3 4
2 3 1 5 1
At index 0, correct position of nums[i] is nums[i]-1 i.e., index 1
Swap(nums[0],nums[nums[0]-1])
0 1 2 3 4
3 2 1 5 1
Correct element is not present at index 0, we swap
0 1 2 3 4
1 2 3 5 1
Since correct elements are present at index 0, 1 and 2 we update i to 3 and then swap elements at index 3 and 4
0 1 2 3 4
1 2 3 1 5

In the above example until and unless the correct element reaches its correct position, variable i will not get updated. This depicts one cycle. For any element $nums[i]$ its correct position will be $nums[nums[i]-1]$, and if at any index correct element is not present that means its a duplicate element.

Example: 442. Find All Duplicates in an Array

An array of integers in the range [1,n] is given, each integer appears once or twice. We have to find all the integers that appear twice in the array.
*Naive Approach*would be to simply use a map or a frequency array to store the frequency of each element and return an array of elements appearing twice. But we require constant space complexity.
*Efficient Approach*is to put each element at its each correct index in the array as we have all integers in the range $[1,n]$, and check if any of the element is not at its correct position then it is a duplicate element. Hence we can use Cyclic Sort algorithm for this problem.

- C++

Written by @prishit55

```
class Solution {
public:
vector<int> findDuplicates(vector<int>& nums) {
vector<int> duplicates;
int n = nums.size();
// cyclc sort
int i = 0;
while (i < n) {
//correct index
int index = nums[i] - 1;
// if correct element is not present at the index
if (nums[i] != nums[index]) {
// we swap the elements
swap(nums[i], nums[index]);
} else {
//do nothing if element is present at its correct position
i++;
}
}
for(int i = 0; i < n; i++) {
// extract all those elements which are not present at their correct position
if (nums[i] != i + 1) {
duplicates.push_back(nums[i]);
}
}
return duplicates;
}
};
```

Time Complexity : $$O(N)$$

Space Complexity : $$O(1)$$

Cyclic Sort pattern is very useful to solve problems involving arrays containing numbers in a given range, finding the missing or duplicate numbers.

Suggested Problems

Problem Name	Difficulty	Solution Link
0268 - Missing Number	Easy	N/A
0448 - Find All Numbers Disappeared In An Array	Easy	N/A
0645 - Set Mismatch	Easy	N/A
0041 - First Missing Positive	Hard	N/A

Insertion Sort

Author: @Shivashish-rwt

Contributor: @wingkwong

Overview

Insertion sort is one of the sorting algorithms that sort the elements by placing an unsorted element in correct sorted order one at a time.

A sorted arrayis an array in which elements are sorted either in ascending or descending order, e.g $[1, 2, 3, 4, 5, 6, 7]$.

In a sorted array, all the elements to the left of any element are smaller than that. To make the array sorted, we have to place an element in such a position, so that every element to the left are smaller than that. That is pretty much what we do in a insertion sort.

Algorithm

1. Pick an element from the array, and store it in a variable $nums$.
2. Now, start comparing the $nums$ with all the elements to the left of it.
3. If the $nums$ is less than the current element shift the current element to right until you find an element smaller than $nums$.
4. Place the $nums$ in current position.
5. Repeat all the above steps until the array is sorted.

For the array $[3,2,5,10,9]$, the steps would be

- First of all, we will pick the first element which is 3 in our case.
- Now, we will compare with all the elements left to it, in this case there is nothing left to 3 so, we will do nothing.
- Now, pick the next element which is 2 and start comparing with all the elements left to it.

- First element left to 2 is 3 , which is greater than selected element 2. So, we will move 3 to right. If the element left to selected element is less than selected element then, our selected element have reached it's correct position so, we will place selected element there only.
- Now, there is nothing no more element further left to our selected element 2. So, we will place 2 at the beginning of array.
- We will proceed with the same way for all the elements. Then, at last we will get our array as sorted.

Example: 1464. Maximum Product of Two Elements in an Array

Given the array of integers nums, you will choose two different indices i and j of that array. Return the maximum value of $(nums[i] - 1) * (nums[j] - 1)$.
The problem wants us to find the maximum value of $(nums[i] - 1) * (nums[j] - 1)$, where i and j are two different indices of the given array.
By looking at the expression, we can observe that its just a product of two numbers of the given array.
We have to maximize the product. Now, in order to maximize the product of two numbers we have to choose the two largest number possible. So, in order to find the maximum value of $(nums[i] - 1) * (nums[j] - 1)$ we just have to take the largest two number in the given array.
Now, we know how can we maximize the value of given expression. But we have to also figure out the how can we get the two largest number present in the given array. Here comes the sorting method, if we sort our array then, the largest number would be

present at the last index and second largest number would be present at the second last index of the sorted array. For sorting, we are going to use Insertion Sort Algorithm.
We have figured out the solution of the problem:

- Sort the given array using insertion sort (Refer to the algorithm section).
- Take out the last two elements because those are the largest two elements in our array.
- Put the values in the expression and return it.
- C++
- Java
- Python

Written by @Shivashish-rwt

```
class Solution {
public:
void insertionSort(vector<int> &arr, int n) {
for (int i = 1; i < n; i++) {
// Picking element from array
int nums = arr[i];
int j = i - 1;
// Comparing nums with all the elements left to it
while (j >= 0 and arr[j] > nums) {
// Shifting the greater element to the right
arr[j + 1] = arr[j];
j--;
}
// Placing selected element at correct position
arr[j + 1] = nums;
}
}
int maxProduct(vector<int>& nums) {
int n = nums.size();
insertionSort(nums, n);
return (nums[n - 1] - 1) * (nums[n - 2] - 1);
}
};
```

Time Complexity

Apart from sorting the the given array we are not doing anything in the solution. The insertion sort would take $O(n ^ 2)$ to sort the array. Therefore, the Time complexity is $O(n ^ 2)$

Space Complexity

We are not using any extra space apart from the array we have to sort. So, the space complexity is $O(1)$.

Suggested Problems

Problem Name	Difficulty	Solution Link
1365 - How Many Numbers Are Smaller Than the Current Number	Easy	View Solutions
2037 - Minimum Number of Moves to Seat Everyone	Easy	N/A
1913 - Maximum Product Difference Between Two Pairs	Easy	N/A
2089 - Find Target Indices After Sorting Array	Easy	N/A

Merge Sort

Author: @Sreetama2001

Contributor: @wingkwong

Overview

Merge Sort works by recursively breaking down an array into multiple subarrays and then after comparing each of the subarrays. It arranges them into ascending or descending order by value and merges them into a single sorted array.
Suppose we have an array of integers $[6, 5, 3, 1, 8, 7, 2, 4]$.
In Python it is called listand in C++ it can be called either array or vectorand in Java it is called a ArrayList.
Then we can see that merge sort is performed in this way.

Image by Brian Hans via [Medium]

Algorithm

Divide

- Calculate the midpoint by checking if the left index is less than the right index, if yes divide the array.
- Now continue dividing the array until $index*{left} < *index*{right}$ becomes false, that is until the division is not possible.

Conquer

- After dividing the array into the smallest units, start merging the elements again by comparing them.
- We need to compare and merge starting from the last splits or last smallest units. So Recursionneeds to be done here.

Merge

- Since each half is already sorted so we need to just sort between 2 halves to combine / mergethan to make a bigger sorted array.

Example: 0912 - Sort an Array

Given an array of integers nums, sort the array in ascending order and return it. You must solve the problem without using any built-in functions in O(nlog(n)) time complexity and with the smallest space complexity possible.

Top Down Approach

Algorithm

- if $left == right$
  - array has only one element, hence return.
- if $right > left$
  - find the middle point to divide the array into two halves: $middle = left + (right - left) / 2$

    - call mergeSort again for first half for further dividing: call $mergeSort(array, left, middle)$
    - call mergeSort again for second half for further dividing: call $mergeSort(array, middle + 1, right)$
    - merge the two halves sorted: call $merge(array, left, middle, right)$
    - merge function is called to compare and merge the elements into an array
- C++
- Python3
- Java

```
class Solution {
public:
void merge(vector<int>& nums, int l, int m, int r) {
// create a temporary array
vector<int> tmp(r - l + 1);
// index for left subarray
int i = l;
// index for right subarray
int j = m + 1;
// index for temporary array
int k = 0;
while (i <= m && j <= r) {
// increment the left pointer
// if the right pointer element is bigger
// Since we are sorting in ascending order,left(smaller element) goes first
if(nums[i] <= nums[j]) tmp[k++] = nums[i++];
else tmp[k++] = nums[j++];
}
// Since in the above while loop if one condition stop satisfying loop breaks
// Then we need to take care of next / remaining elements
// Hence adding remaining elements of left half
// adding remaining elements of right half
while (j <= r) tmp[k++] = nums[j++];
// Copy data to nums
for (i = 0; i < k; i++) nums[l + i] = tmp[i];
}
void mergeSort(vector<int>& nums, int l, int r) {
if (l >= r) return;
// middle index, same as (l + r) / 2
int m = l + (r - l) / 2;
mergeSort(nums, l, m);
mergeSort(nums, m + 1, r);
merge(nums, l, m, r);
}
// function to return sorted array in leetcode
vector<int> sortArray(vector<int>& nums) {
mergeSort(nums, 0, nums.size() - 1);
return nums;
}
};
```

Bottom Up Approach / Iterative technique

Algorithm

- It starts with an element in the array. It is an iterative approach and because one item array is always sorted
- Compares two nearby elements to merge into a sorted subarray. Similarly, we then merge the sorted subarrays like we have done in top-down recursive approach (two-pointer approach)
- Continues until we have a sorted array
- Java

```
class Solution {
public List<Integer> sortArray(int[] nums) {
List<Integer> res = new ArrayList<>();
if (nums == null || nums.length == 0) return res;
mergeSort(nums);
for (int i : nums) res.add(i);
return res;
}
// iterative only
private void mergeSort(int[] nums) {
// here the size is doubled by 2
// Since we are taking 2 elements at a time
// That is the size of elements to be merged are becoming 2, 4, 8, 16 ...
for (int size = 1; size < nums.length; size *= 2) {
for (int i = 0; i < nums.length - size; i += 2 * size) {
int mid = i + size - 1;
int end = Math.min(i + 2 * size - 1, nums.length - 1);
merge(nums, i, mid, end);
}
}
}
// Same as the merge function of the top down approach
private void merge(int[] nums, int l, int mid, int r) {
int[] tmp = new int[r - l + 1];
int i = l, j = mid + 1, k = 0;
while (i <= mid || j <= r) {
if (i > mid || j <= r && nums[i] > nums[j]) {
tmp[k++] = nums[j++];
} else {
tmp[k++] = nums[i++];
}
}
// merging rest of the elements
System.arraycopy(tmp, 0, nums, l, r - l + 1);
}
}
```

Merging of \$n\$ elements takes \$n\$ time and since each time the array is cut into half it takes \$\log*{2}n\$ time to reach the top. So total time complexity is\$O(n\log*{2} n)\$.

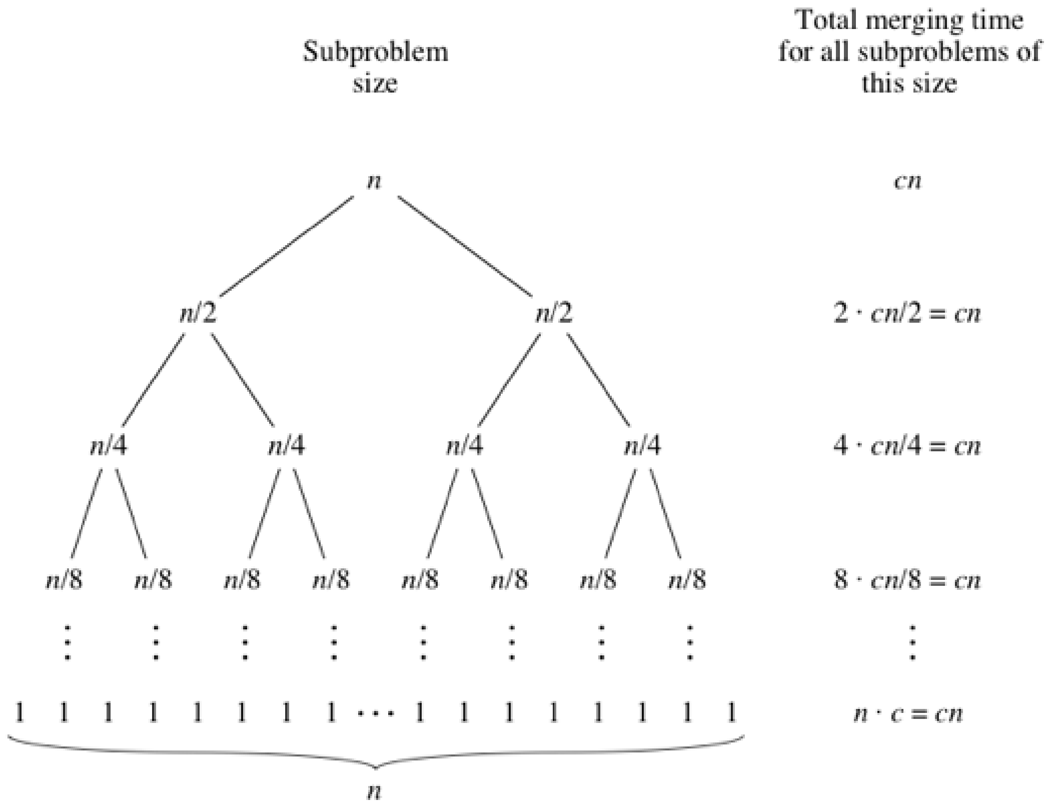


Image taken from Khan Academy

Merge Sort is a stable sortbecause the same element in an array maintain their original positions with respect to each other that means the original order of elements of input

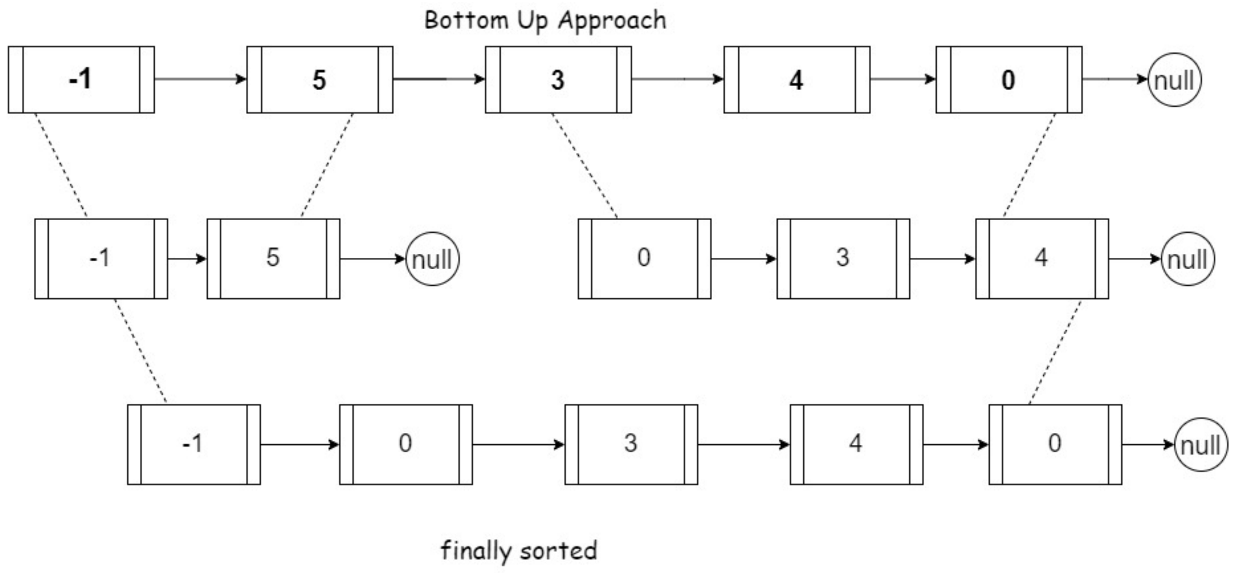
set is preserved.
Merge sort copies of more than a constant number of array elements. Hence it requires additional space which depends upon the input size of the array elements. So is an out of place algorithm.
Time Complexity: Best & Worst & Average is $O(n \log_{2} n)$
Space Complexity: $O(n)$
For very large arrays Merge sort is in effienct as it allocates an extra space of $O(n)$ so we should go for Quick sort.

Example: 0148 - Sort List

Given the head of a linked list, return the list after sorting it in ascending order.
Here we can follow both top-down and bottom-up merge sort.

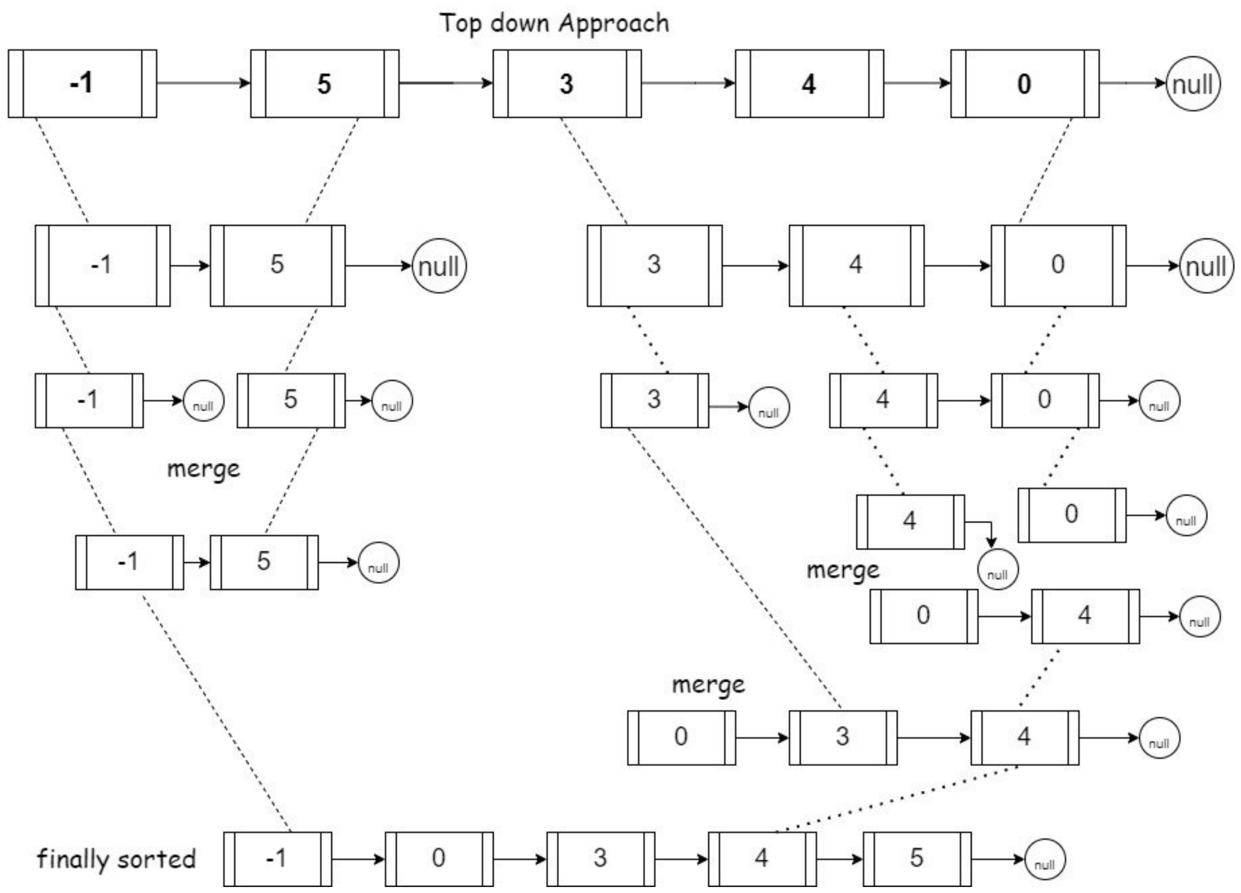
Approach for Bottom-Up Merge Sort of Linked list

1. The two lists to be merged must be ordered respectively.
2. We can only start to merge two lists that only have one element.
3. Then we get an ordered list that has two elements
4. Do this again (that is "recursion").



Approach for Top-down Merge Sort of Linked list

1. Keep recursively dividing the list until there is only one node in the linked list.
2. Sort each sublist and merge each sorted sublist in a new array.
3. The two lists to be merged must be ordered respectively.
4. Here to order we must follow two pointer approach discussed above.



Steps to apply merge sort to a Linked list

Note: Implementation is based on Merge Sort on an array as discussed above.

1. Divide the linked list into two equal parts until when only one element is left.
2. To Divide, we need to find mid in the linked list using slow and fast pointersmethod.
3. Then merge the left and the right nodes of the linked list.

- C++

```
/*
* Definition for singly-linked list.
* struct ListNode {
* int val;
* ListNode *next;
* ListNode() : val(0), next(nullptr) {}
* ListNode(int x) : val(x), next(nullptr) {}
* ListNode(int x, ListNode *next) : val(x), next(next) {}
* };
*/
class Solution {
public:
void mergeSort(ListNode **head) {
// create a current pointer
ListNode *curr = *head ;
ListNode * left;
ListNode * right;
// if the linked list is null
// or the size is one return back the same
if(curr == NULL || curr->next == NULL) return;
// call to find the middle node between left and right
findMiddle(curr, &left, &right);
// call merge_sort again to divide left half
mergeSort(&left);
// call merge_sort again to divide right half
mergeSort(&right);
// call to merge left and right by sorting them
*head = merge(left, right);
}
void findMiddle(ListNode *curr, ListNode **left, ListNode **right) {
// make a slow pointer
ListNode* slow = curr;
// make a fast pointer
ListNode* fast = curr-> next;
// then we move our fast up to it not become null
while(fast != NULL) {
fast = fast-> next;
if(fast != NULL) {
fast = fast-> next;
slow = slow-> next;
}
}
*left = curr;
// right to slow next
*right = slow-> next;
// and slow next to null
slow-> next = NULL;
}
ListNode* merge(ListNode* left, ListNode* right) {
ListNode* res = NULL;
// Check if left is null, nothing to merge
if(left == NULL) return right;
if(right == NULL) return left;
// if value of the left <= value of right
// then res = left
if(left-> val <= right-> val) {
res = left;
// and again call merge for res's next
res-> next = merge(left-> next, right);
} else {
res = right;
// and again call merge for res's next
res-> next = merge(left, right-> next);
}
return res;
}
```

```
ListNode* sortList(ListNode* head) {
mergeSort(&head);
return head;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
912-Sort an Array	Medium	View Solutions
56-Merge Intervals	Medium	N/A
148-Sort List	Medium	View Solutions
327-Count of Range Sum	Hard	View Solutions
23-Merge k Sorted Lists	Hard	View Solutions

Selection Sort

Author: @Bobliuuu

Contributor: @wingkwong

Overview

Selection sort is a commonly used comparison-based sorting algorithm. It's very simple to implement and works on the premise that two subarrays are maintained: one which is sorted, and one which is unsorted. In each step, one more element of the array gets sorted, until the entire array is sorted.

Algorithm

The concept of selection sort is that each time, we find the minimum element in the unsorted subarray, and we put it at the end of the sorted subarray.
Consider an array with n distinct elements. Looping n times, we

- find the minimum element in the unsorted subarray
- move that element to the end of the sorted subarray

The reason the algorithm works is that each time the smallest value in the unsorted array is the largest value in the sorted array.

Example: [0075 - Sort Colors]

An array of integers $nums$ is given, with n values that are either 0, 1, or 2 (representing red, white, or blue). We have to sort them in place so that adjacent elements are together.
In order to achieve this, we can use selection sort.

Pseudocode

- We loop through the array once.
- For each iteration, we keep the index of seperation of the sorted and unsorted subarrays.
- Then, we loop through the array, keeping the index of the minimum element.
- Finally, we move that element to the end of the sorted subarray, and increment the index of seperation.
- We keep doing this until we reach the end of the array, and print out the resulting array.

Dry Run

Let's see the algorithm work on the first test case: $nums = [2, 0, 2, 1, 1, 0]$ and $n = 6$.
In the first iteration, we find the minimum value from index 0 to 5, which is 0.
This value is moved to the 0-th index.
The array is now $nums = [0, 2, 2, 1, 1, 0]$.
In the second iteration, we find the minimum value from index 1 to 5, which is 0. This value is moved to the 1st index.
The array is now $nums = [0, 0, 2, 1, 1, 2]$.
In the third iteration, we find the minimum value from index 2 to 5, which is 1. This value is moved to the 2nd index.
The array is now $nums = [0, 0, 1, 2, 1, 2]$.
In the fourth iteration, we find the minimum value from index 3 to 5, which is 1. This value is moved to the 3th index.
The array is now $nums = [0, 0, 1, 1, 2, 2]$.
In the fifth iteration, we find the minimum value from index 4 to 5, which is 2. This value is moved to the 4th index.

The array is still $nums = [0, 0, 1, 1, 2, 2]$.
In the last iteration, we find the minimum value from index 5 to 5, which is 2. This value is moved to the 5th index.
The array is still $nums = [0, 0, 1, 1, 2, 2]$.
Now, let's look at the solution to the example question.

- C++

Written by @Bobliuuu

```
class Solution {
public:
void sortColors(vector<int>& nums) {
int n = nums.size();
// Loop once through the array, keeping the seperation index of the sorted and unsorted
subarray
for (int i = 0; i < n - 1; i++) {
// index of mini element
int midx = i;
// Loop through array again to find the index of the minimum value in the unsorted subarray
for (int j = i + 1; j < n; j++) {
// If the current element is smaller than the current minimum element
if (nums[j] < nums[midx]) {
// Keep the index of the actual minimum element
midx = j;
}
}
// Swap the minimum value with the current array element (the end of the sorted subarray)
swap(nums[i], nums[midx]);
}
}
};
```

Complexity

The time complexity of this program is $$O(n ^ 2)$$, since we run two for loops to iterate through the array.
The space complexity of this program is $$O(1)$$, since we don't need any extra space other than our original array.

Suggested Problems

Problem Name	Difficulty	Solution Link
0075 - Sort Colors	Easy	View Solutions
0268 - Missing Number	Easy	View Solutions
0448 - Find All Numbers Disappeared In An Array	Easy	N/A

Tim Sort

Author: @Bobliuuu

Contributor: @wingkwong

Overview

Timsort is a fast stable sorting algorithm based upon both insertion sortand merge sort. The algorithm was first created by Tim Peters in 2002, and is now being used in Python's sort()and Java's Arrays.sort(). The reason this algorithm is so fast is because it leverages the benefits of both merge sort and insertion sort. Let's see how it works!

Algorithm

Timsort works by first splitting an array into runs. A runis a subarray of data spliced from the original array. These runs are generated using merge sort (each run has a standard size of 32-64, to split the array into small enough pieces for insertion sort to be fast on each one), and insertion sort is used to sort each run. Finally, merge sort combines these sorted arrays together recursively.
Basically, to run timsort:

- We split the array into runs.
- For each run, run insertion sort to sort that section.
- Merge runs together one by one using merge sort, by comparing values in each sorted list and combining them.

This algorithm works because each run is sorted using insertion sort, and merge sort makes sure that each subarray is merged to the original array in the correct position.

Example: 0287 - Find The Duplicate Number

An array of integers in the range [1, n] is given, where one integer is repeated. We have to find this repeated number.
Naive Approach: Using a Hashmap or a frequency array, we can store the number of times each element comes up. We then return an array by looping through the frequency array finding the value that appears twice. However, this requires $$O(n)$$ space complexity, and the problem requires us to have $O(1)$ space complexity.
For this sort of problem, we can use timsort to lower our space complexity!

Pseudocode

- Sort the array using timsort
- Loop through the array
- If two values are the same, then that value must be repeated. Return that value.

Dry Run

Let's do a dry run of timsort with the array $[5, 4, 3, 1, 2, 6, 7, 4]$, and a run size of $2$.

- Each run is sorted using insertion sort. The array becomes $[4, 5, 1, 3, 2, 6, 4, 7]$.
- The merges happen using recursion. We first attempt to split the array into two parts, down the middle.
- First, the left part is merged, meaning the first two runs are merged. Then the array becomes $[1, 3, 4, 5, 2, 6, 4, 7]$.
- Then, the right part is merged, meaning the next two runs are merged. The the array becomes $[1, 3, 4, 5, 2, 4, 6, 7]$.
- Finally, the entire array is merged. The array is finally sorted: $[1, 2, 3, 4, 4, 5, 6, 7]$.
- C++

Written by @Bobliuuu

```
class Solution {
```

```
// initalize the size of each run
const int RUN = 32;
void insertionSort(vector<int>& nums, int left, int right) {
for (int i = left; i <= right; i++) {
int tmp = nums[i];
int j = i - 1;
while (j >= left && nums[j] > tmp) {
nums[j + 1] = nums[j];
j--;
}
nums[j + 1] = tmp;
}
}
void merge(vector<int>& nums, int left, int mid, int right) {
// maintain the two previous lists
vector<int> lt, rt;
int lenlt = mid - left + 1, lenrt = right - mid;
for (int i = 0; i < lenlt; i++) {
lt.push_back(nums[left + i]);
}
for (int i = 0; i < lenrt; i++) {
rt.push_back(nums[mid + 1 + i]);
}
// start recreating the correct list, putting the smaller one each time
int i = 0, j = 0, k = left;
while (i < lenlt && j < lenrt) {
if (lt[i] <= rt[j]) {
nums[k] = lt[i];
i++;
} else {
nums[k] = rt[j];
j++;
}
k++;
```

```
}
while (i < lenlt) {
nums[k] = lt[i];
k++; i++;
}
while (j < lenrt) {
nums[k] = rt[j];
k++; j++;
}
}
void timSort(vector<int>& nums) {
int n = nums.size();
// insertion sort on each run
for (int i = 0; i < n; i += RUN) {
insertionSort(nums, i, min((i + RUN-1), (n - 1)));
}
for (int size = RUN; size < n; size = 2 * size) {
for (int left = 0; left < n; left += 2 * size) {
// determine indices for each run for merging
int mid = left + size - 1, right = min((left + 2 * size - 1), (n - 1));
// merge the two runs if needed
if (mid < right) {
// use recursion to merge the array
merge(nums, left, mid, right);
}
}
}
}
int findDuplicate(vector<int>& nums) {
// use timsort to sort the array
timSort(nums);
for (int i = 0; i < nums.size() - 1; i++) {
// return the duplicate if found
if (nums[i] == nums[i + 1]) {
```

```
return nums[i];
}
}
return 0;
}
};
```

Complexity

The time complexity of this program is $$O(n log n)$$, since the merging takes $logn$ steps, and merges n values each time.
The space complexity of this program is $$O(1)$$, since we don't need any extra space other than our original array.

Suggested Problems

Problem Name	Difficulty	Solution Link
0075 - Sort Colors	Easy	View Solutions
0268 - Missing Number	Easy	View Solutions
0448 - Find All Numbers Disappeared In An Array	Easy	N/A

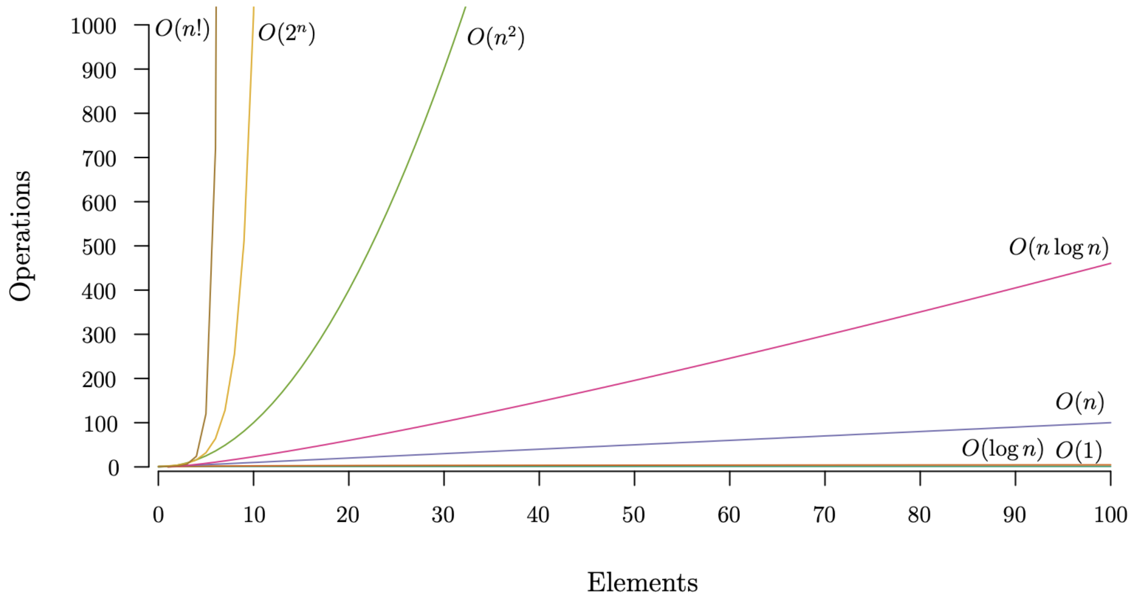
Time Complexity

Author: @wingkwong

Overview

Time Complexity is one of the important measurements when it comes to writing an efficient solution. It estimates how much time your solution needs based on some input. If your solution is too slow, even it passes some test cases, it will still consider it as a wrong answer.

The time complexity is denoted by Big O notation. Normally, $$n$$ means the input size. For example, if the input is a string $$s$$, then $$n$$ would be the length of $$s$$.



Estimating Time Complexity

Input size	Time Complexity
$$n <= 10$$	$$O(n!), O(n^7),O(n^6)$$
$$n <= 20$$	$$O(2^n)$$
$$n <= 80$$	$$O(n^4)$$
$$n <= 400$$	$$O(n^3)$$
$$n <= 7500$$	$$O(n^2)$$
$$n <= 10^6$$	$$O(nlogn)$$, $$O(n)$$
$$large$$	$$O(1)$$, $$O(logn)$$

Example 1:

In the following case, the time complexity depends on $$n$$. Therefore, it is $$O(n)$$.

```
for (int i = 0; i < n; i++) {
// do something
}
```

Example 2:

In the following case, the time complexity depends on $$n$$ and $$m$$. Therefore, it is $$O(n*m)$$.

```
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
// do something
}
}
```

Example 3:

In the following case, the time complexity is $$O(\sqrt n)$$. You can see $$i * i <= n$$ as $$i <= \sqrt n$$.
As sqrt() returns double, it would be safe to use i * i <= n to check the condition instead of using i <= sqrt(n).

```
for (int i = 2; i * i <= n; i++) {
// do something
}
```

Useful Resources

- Big-O_Cheat_Sheet-Letter.pdf
- BIG-O.pdf

Trie

Author: @wingkwong

Overview

208. Implement Trie (Prefix Tree)will be used as an example.

A trie (pronounced as "try") or prefix tree is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

- Trie() Initializes the trie object.
- void insert(String word) Inserts the string word into the trie.
- boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
- boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

Example 1:

```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]
Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple"); // return True
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app"); // return True
```

Constraints:

- 1 <= word.length, prefix.length <= 2000
- word and prefix consist only of lowercase English letters.
- At most 3 * 10^4 calls in total will be made to insert, search, and startsWith.

Trie

We can see Trie containing a number of Trie nodes. Each node contains a value and links to other nodes. We start from the root, we traverse till e so that we have gee. At this node, we have three different nodes to traverse so that we have $geek$, $geer$, and $geez$. We can also further to have $geeks$ and $geekt$ and so on.

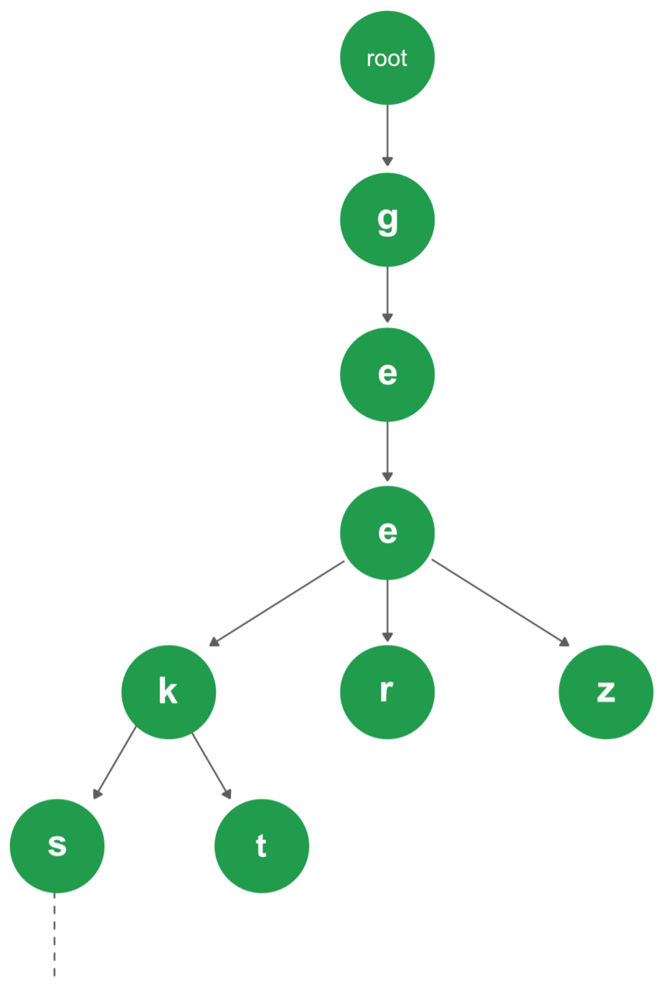


Diagram Source: https://www.geeksforgeeks.org/

Trie Node

Each Trie Node should have a children array with the size of 26 for character a to z. Also it has a boolean variable $$isEndOfWord$$ to indicate if a word is ended at this node.

```
class TrieNode {
public:
// is a word ended at this node
bool isEndOfWord;
// children for 26 characters
TrieNode* children[26];
// constructor - setting initial values
TrieNode() {
// no word is ended here
isEndOfWord = false;
// not linking to other Trie node
for (int i = 0; i < 26; i++) {
children[i] = NULL;
}
}
};
```

Initializing

```
Trie() {
// init Trie - define the very first node
root = new TrieNode();
}
```

Searching

Given a word, searchreturns if the word is in the trie.

```
bool search(string word) {
// start from the root node
TrieNode* node = root;
// iterate the word
for (int i = 0; i < (int) word.size(); i++) {
// get the index of the character
// a -> 0
// b -> 1
// ...
// z -> 25
int idx = word[i] - 'a';
// if there is no such node, that means the word doesn't exist
if (!node->children[idx]) return false;
// otherwise, traverse the next node
node = node->children[idx];
}
// check if this node is marked with isEndOfWord = true
return node->isEndOfWord;
}
```

Insertion

Given a word, insertinserts it into the trie.

```
void insert(string word) {
// start from the root node
TrieNode* node = root;
for (int i = 0; i < (int) word.size(); i++) {
// get the index of the character
// a -> 0
// b -> 1
// ...
// z -> 25
int idx = word[i] - 'a';
// traverse each node,
if (!node->children[idx]) {
// if the node doesn't exist,
// create a new node
node->children[idx] = new TrieNode();
}
// traverse the next one
node = node->children[idx];
}
// mark this node with isEndOfWord = true
node->isEndOfWord = true;
}
```

startsWith

Given a prefix, startsWithchecks if there is any word in the trie that starts with the given prefix.

```
bool startsWith(string prefix) {
// start from the root node
TrieNode* node = root;
// iterate each character in prefix
for (int i = 0; i < (int) prefix.size(); i++) {
// get the index of the character
// a -> 0
// b -> 1
// ...
// z -> 25
int idx = prefix[i] - 'a';
// if there is no such node, that means the word doesn't exist
if (!node->children[idx]) return false;
// otherwise, traverse the next node
node = node->children[idx];
}
// all target nodes have been traversed, return true here
return true;
}
```

Two Pointers

Authors: @heiheihang@wingkwong

Overview

The two pointers technique is a technique used to iterate through a data set, typically an array or a list, in a controlled way. It involves using two pointers, one pointing to the beginning of the data set and the other pointing to the end, and moving them towards each other based on specific conditions. This technique is commonly used to solve problems that involve searching for a specific condition or pattern within a data set, or that require a comparison between different elements in the data set.

The two pointers technique is mainly used for solving problems that have a linear time complexity, it can lead to substantial performance improvements over a brute-force approach. Some common examples of problems that can be solved using this technique include:

- Finding the maximum / minimum value in a set of data.
- Counting the number of occurrences of a specific element.

- Finding the longest substring without repeating characters.
- Finding the maximum sum of a sub-array of size $k$.

Overall, the two pointers technique is a useful approach for solving specific types of problems that involve iterating through a data set in a controlled way, such as in pattern matching, data analysis, and statistics. It allows for an efficient and controlled iteration of a data set, which can lead to improved performance and more accurate results.

Example 1: 977. Squares of a Sorted Array

Given an integer array numssorted in non-decreasingorder, return *an array of the squares of each numbersorted in non-decreasing order*.

Let's look at this example

```
# input
nums = [-4,-1,0,3,10]
```

From this input, we can generate the following square numbers:

```
squares = [16, 1, 0, 9, 100]
```

We want to return the following sorted squares:

```
answer = [0, 1, 9, 16, 100]
```

You may be thinking, why can't we generate the squares and then sort the result? This approach would take $$O(NlogN)$$, and we want to do better than this.

We can sequentially add the next biggest elements with the two pointer approach. We first set a left_pointerat the left of the list and a right_pointerat the right of the list. The left_pointershould be pointing at the largestnegative number (most negative), and the right_pointershould be pointing at the largestpositive number. We can move the pointers accordingly to find the next largest squared number.

- Python
- C++
- Java

Written by @heiheihang

```
class Solution:
def sortedSquares(self, nums: List[int]) -> List[int]:
# initialize two pointers
left_pointer, right_pointer = 0, len(nums) - 1
# initialize result
res = []
# while left_pointer does not meet right_pointer
while(left_pointer <= right_pointer):
# if the square of left_pointer and right_pointer
if(abs(nums[left_pointer]) > abs(nums[right_pointer])):
res.append(nums[left_pointer] ** 2)
# we move the left to the right
left_pointer += 1
else:
res.append(nums[right_pointer] ** 2)
# we move the right pointer to the left
right_pointer -= 1
# we need to reverse the result list
res.reverse()
return res
```

Unfortunately, there is no fixed way to perform two pointers. However, generally, we have a pointer at the start of the list and another pointer at the end of the list. We have to carefully analyze the question and choose the most appropriate approach to operate the two pointers.

Suggested Problems

Problem Name	Difficulty	Solution Link
1768 - Merge Strings Alternately	Easy	View Solutions
2108 - Find First Palindromic String in the Array	Easy	View Solutions
0283 - Move Zeroes	Easy	View Solutions

Graph Theory

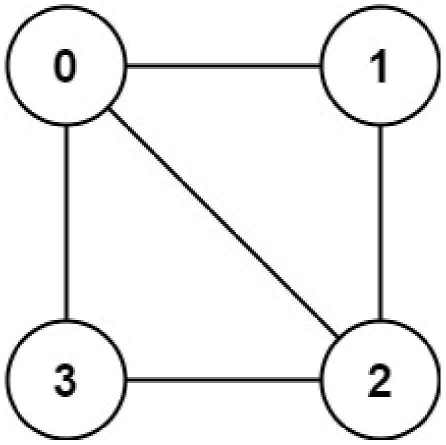
Introduction

Authors: @heiheihang@wingkwong

Overview

A graph is made up of a collection of points or objects called vertices and connections between some of those vertices called edges. The edges can be either one-way (can only be traversed in one direction), two-way, have a numerical value associated with traversing them, or without any value. We can use graphs to solve a plethora of interesting problems!

Here is a undirected graph with 4 vertices (or nodes) and 5 edges.



Example

In real life, we may use graphs. Let's say we have six people:

- Alice
- Bob
- Cathy
- Danny
- Ethan
- Fiona

and we are also given a list of friends.

```
[["Alice", Bob"], ["Cathy", "Danny"], ["Alice", "Cathy"], ["Ethan", "Fiona"]]
```

Here, we know that:

- Alice and Bob are friends
- Cathy and Danny are friends
- Alice and Cathy are friends
- Ethan and Fiona are friends

We say that Alice, Bob, Cathy, and Danny are in Friend Group 1(they are friends or have common friends). Ethan and Fiona are in Friend Group 2(they are friends or have common friends).
In this task, we can easily tell the number of friend groups(there are 2 friend groups), as well as the size of the largest friend group(the largest group - Friend Group 1 - has 4 members).
This seems easy at first glance! We just need to "group them up". However, this is more complicated than you think. There are three potential solutions to this problem :

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Union Find

We will learn different strategies for similar problems, and hopefully you know which one to use after learning the key concepts in graph theory.

Bellman Ford Algorithm

Author: @wingkwong

Overview

Bellman Ford Algorithm computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. Similar to Dijkstra's algorithm, it proceeds by relaxation. However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not been processed, which all of its outgoing edges will be processed. On the other hand, Bellman Ford Algorithm relaxes all the edges and does the relaxation only $|V| - 1$ times where $|V|$ is the number of vertices in the graph. This is because given a graph with no negative weight cycles with V vertices, the shortest path between any two vertices has at most $|V| - 1$ edges.

Implementation

- C++
- Python

Written by @wingkwong

```
template<typename T_a3, typename T_vector>
void bellman_ford(T_a3 &g, T_vector &dist, int src, int mx_edges) {
dist[src] = 0;
for (int i = 0; i <= mx_edges; i++) {
T_vector ndist = dist;
for (auto x : g) {
auto [from, to, cost] = x;
ndist[to] = min(ndist[to], dist[from] + cost);
}
dist = ndist;
}
}
```

Binary Tree

Author: @wingkwong

Overview

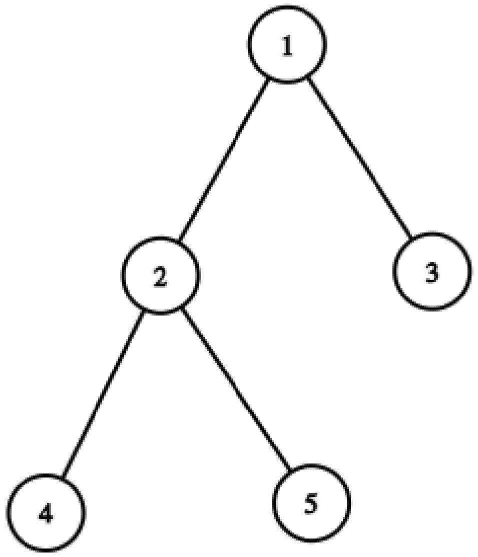
A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. The root node is the topmost node in the tree and the leaf nodes are the nodes at the bottom with no children. Binary trees are commonly used to implement data structures such as binary search trees, which are used for efficient searching and sorting. The height of the binary tree is the number of edges from the root to the deepest leaf node. The depth of a node is the number of edges from the root to that node.

Properties

- The number of nodes on level $l$ is equal to the $2^l$, like on level $0$ (root node) we got $2 ^ 0 >= 1$ node only.
- The Maximum number of nodes in a binary tree of height $h$ is $2^h - 1$.

Traversal

There are different ways to traverse trees - In-order, Pre-order, and Post-order. Supposing we have a binary tree with $5$ nodes,



Pre-order

- Visit the root
- Traverse the left sub-tree
- Traverse the right sub-tree
- C++
- Python

```
void preorder(TreeNode* node) {
if (node == NULL) return;
s.push_back(node->val);
preorder(node->left);
preorder(node->right);
}
```

In-order

- Traverse the left sub-tree
- Visit the root
- Traverse the right sub-tree
- C++
- Python

```
void inorder(TreeNode* node) {
if (node == NULL) return;
inorder(node->left);
s.push_back(node->val);
inorder(node->right);
}
```

Post-order

- Traverse the left sub-tree
- Traverse the right sub-tree
- Visit the root
- C++
- Python

```
void postorder(TreeNode* node) {
if (node == NULL) return;
postorder(node->left);
postorder(node->right);
s.push_back(node->val);
}
```

Summary

Traversal		
Path		
Order		
Pre-order	1 -> 2 -> 4 -> 5 -> 3	Root -> Left -> Right
In-order	4 -> 2 -> 5 -> 1 -> 3	Left -> Root -> Right
Post-order	4 -> 5 -> 2 -> 3 -> 1	Left -> Right -> Root

Breadth First Search (BFS)

Author: @heiheihang

Overview

In Breadth-First Search (BFS), we explore allthe closest nodes first before going one step further. A good example would be:
Given a binary tree, find the closest nodefrom rootthat has the value 3

Of course, you may use DFS to find the solution by iterating all nodes. However, as you can imagine, if the target node is the right child of the root, we have wasted so much time iterating the entire left branch of the root!
BFS would immediately locate the closest target node without wasting time iterating deeper nodes.
We will introduce the following template for BFS:

```
def findTargetNode(root, targetValue):
if(root is None):
return None
#currentLevel contains the nodes with the same distance to root (closest so far)
currentLevel = [root]
#we increase our depth one by one as long as there is still node
while(len(level) > 0):
#we store the current level node's children in nextLevel
nextLevel = []
for node in currentLevel:
#skip if the node is None
if(node is None):
continue
#we can be sure the target node is the CLOSEST so we can return
#because we are traversing the tree level by level
if(node.val == targetValue):
return node
#add the children to nextLevel
nextLevel.append(node.left)
nextLevel.append(node.right)
#change the currentLevel to nextLevel (no target node in this level, go next)
currentLevel = nextLevel
#if no target node has been returned
return None
```

In general, we use a queueto model BFS.As the head of the queue represents the closest nodes, and the tail of the queue represents the furthest nodes. We look at the head of the queue, and add new nodes to the end of the queue.
We can start applying this template to the following problem (LeetCode Link).
Given the rootof a binary tree, return *the average value of the nodes on each level in the form of an array*. Answers within 10-5of the actual answer will be accepted.
In this problem, our primary goal is to separate the tree into different levels. For example, we have these following levels:

1. [1]
2. [2,3]
3. [4, _, 3, 5]

When we can separate the tree into different levels, we are just one step before obtaining the solution (which is just getting the averages of each list)
The challenge here is how can we separate the tree into different levels. We can use the template above with currentLeveland nextLevel.

```
def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:
answer = []
#saves the nodes in the currentLevel
currentLevel = [root]
#continue traversing as long as there is still unexplored nodes
while(len(currentLevel) > 0):
#stores the children of the nodes in the currentLevel
nextLevel = []
#stores the total sum of the currentLevel nodes
currentLevelNodeCount = 0
currentLevelSum = 0
for node in currentLevel:
if(node is None):
```

```
currentLevelSum += node.val
nextLevel.append(node.left)
nextLevel.append(node.right)
currentLevelNodeCount += 1
#calculate the level average
if currentLevelNodeCount:
currentLevelAverage = currentLevelSum / currentLevelNodeCount
answer.append(currentLevelAverage)
#explore the nextLevel
currentLevel = nextLevel
return answer
```

We should keep practising the this template of BFS in these similar problems.

Suggested Problems

Problem Name	Difficulty	Solution Link
0101 - Symmetric Tree	Easy	View Solutions
0199 - Binary Tree Right Side View	Medium	View Solutions
0103 - Binary Tree Zigzag Level Order Traversal	Medium	View Solutions

Depth First Search (DFS)

Author: @heiheihang

Overview

In Depth-First Search (DFS), we aim to finish one branch before looking at other branches.
A good example of DFS is the following problem (LeetCode Link):

Given the rootof a binary tree, return *its maximum depth*.
A binary tree's maximum depthis the number of nodes along the longest path from the root node down to the farthest leaf node.
We want to know how farwe can travel from the root, so we try one path at a time. (Of course, this problem can be solved by Breadth-First-Search , but DFS is more intuitive)
DFS can be implemented in the following way

```
def dfs(node):
if(node == None):
# we stop when node is invalid
return
# explore left branch first
dfs(node.left)
# evalute current node
print("I just visited the left branch!")
print("I am number: " + str(node.val))
print("I am visiting the right branch now!")
# explore right branch
dfs(node.right)
```

With this template of DFS, we can modify the function above to obtain the depth of each branch

```
def findMaximumDepth(root):
def dfs(node):
if(node == None):
# we stop when node is invalid
return 0
# explore left branch first
left_branch_depth = dfs(node.left)
print("I just visited the left branch!")
```

```
# explore right branch
right_branch_depth = dfs(node.right)
# return the larger depth of the two branches
return max(left_branch_depth, right_branch_depth) + 1
return dfs(root)
```

There we go! This is a simple DFS problem. We are going to work through a few more DFS problems together.

Let's look at another problem (LeetCode Link)

Given the rootof a binary tree, return *the length of the diameterof the tree*.

The diameterof a binary tree is the lengthof the longest path between any two nodes in a tree. This path may or may not pass through the root.

The lengthof a path between two nodes is represented by the number of edges between them.

This problem may seem difficult at first glance. However it is just a minor tweak from the previous problem. The longest path between two nodes would be the sum of the maximum depth of the left branch and that of the right branch. Modify the code above before you look at the solution below.

```
def findTreeDiameter(root):
diameter = 0
def dfs(node):
if(node == None):
# we stop when node is invalid
return 0
# explore left branch first
left_branch_depth = dfs(node.left)
print("I just visited the left branch!")
print("I am visiting the right branch now!")
# explore right branch
```

```
right_branch_depth = dfs(node.right)
#the longest path at the current node is the maximum depth of left and right
local_diameter = left_brach_depth + right_branch_depth + 1
#update the global variable
nonlocal diameter
diameter = max(diameter, local_diameter)
# return the larger depth of the two branches
return max(left_branch_depth, right_branch_depth) + 1
dfs(root)
return diameter
```

Here are some similar problems in which you can tweak the template above to obtain a solution.

Suggested Problems

Problem Name	Difficulty	Solution Link
0404 - Sum of Left Leaves	Easy	View Solutions
0110 - Balanced Binary Tree	Easy	View Solutions
0559 - Maximum Depth of N-ary Tree	Easy	View Solutions

Dijkstra's Algorithm

Author: @wingkwong

Overview

Dijkstra's algorithm is a popular graph search algorithm that is used to find the shortest path between two nodes in a graph. It is a greedy algorithm that uses a priority queue to prioritize the nodes that have the shortest distance from the starting node. The algorithm starts with the starting node and visits the neighboring nodes, updating their

distances and adding them to the priority queue. The process is repeated until the destination node is reached.
The algorithm works by maintaining a priority queue of unvisited nodes, where the priority of a node is determined by the shortest distance from the starting node. The algorithm initializes the starting node with a distance of zero and all other nodes with a distance of infinity. It then repeatedly selects the node with the smallest distance that has not yet been visited and updates the distances of its unvisited neighbors, adding them to the priority queue if they are not already in it. The algorithm terminates when the destination node is dequeued from the priority queue.

Implementation

Let $$dist[u]$$ be the distance / cost / weight to reach node $$u$$. Initially, we use a priority queue to maintain the pair $$p$$ where $$p.first$$ is the node and $$p.second$$ is the cost. We set the distance from source to source is $$0$$ with $$0$$ cost and push the starting point to the priority queue.
The first run, the vertex is the source node. We remove it and check its neighbors. If the distance to the neighbor is greater than the current distance plus the cost, then it means a shorter path is found. Hence, we update it and push it to the priority queue for further process.

- C++

Written by @wingkwong

```
template<typename T_pair, typename T_vector>
void dijkstra(T_pair &g, T_vector &dist, int start) {
priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
dist[start] = 0;
pq.push({start, 0});
while (!pq.empty()) {
```

```
auto [u_node, u_cost] = pq.top(); pq.pop();
if (u_cost > dist[u_node]) continue;
for (auto [v_node, v_cost] : g[u_node]) {
if (dist[v_node] > dist[u_node] + v_cost) {
dist[v_node] = dist[u_node] + v_cost;
pq.push({v_node, dist[v_node]});
}
}
}
}
```

Dijkstra's algorithm is guaranteed to find the shortest path between two nodes in a graph if all the edge weights are non-negative. If the graph contains negative edge weights, Bellman-Ford algorithm should be used instead.

Suggested Problems

Problem Name	Difficulty	Solution Link
0743 - Network Delay Time	Medium	View Solutions
1334 - Find the City With the Smallest Number of Neighbors at a Threshold Distance	Medium	View Solutions

Disjoint Set Union (DSU)

Author: @wingkwong

Overview

A set is a collection of elements. If two sets have no common elements, then they are called disjoint sets. For example, {1, 2} and {3, 4} are disjoint sets while {1, 2} and {1, 3} are not because they have a common element $1$.
Disjoint Set Union (or DSU or Union Find) is a data structure that allows us to combine any two sets into one. Let's say we have $10$ elements and we initialise an array $root$

with a size of 10. Here we have 10 sets and each individual element in the set is the parent.

```
vector<int> root(10);
for(int i = 0; i < 10; i++) root[i] = i;
```

If we join the first element 1 and 2 together, we first check if they belong to the same parent. If they do, it means they have already in the same set. Otherwises, we can point one to another and update $root$ like $root[2] = 1$ which means the root of element 2 is 1. We can make it more flexible to check if they are already in the same set or not simply by returning a boolean value.

```
bool unite(int x, int y) {
x = get(x);
y = get(y);
if (x != y) {
if (x < y) root[y] = x;
else root[x] = y;
return true;
}
return false;
}
```

If we need to check whether two elements have the same parent, then we need a function get to check it. To implement that, we simply check if the target element x is $root[x]$, otherwise we can call the same function recursively until we have the root. In other word, the parent would be

```
int get(int x) {
return x == root[x] ? x : get(root[x]);
}
```

However, the above implementation is not efficient as each call depends on n while we need to optimize it nearly constant time.
One way to optimize it is compress the path. For example, if the root element is 1 and we have the chain like 1 -> 2 -> 3 -> 4. If we write it vertically, element 1 is on the top level, element 2 is on the second level, element 3 is on the third level and so on. We can compress these into the same level, i.e. element 2, 3 and 4 would be on the second level only so that we don't need to talk all the nodes between the root and the source. This would achieve $O(log n)$ per call on average.

```
int get(int x) {
return (x == root[x] ? x : (root[x] = get(root[x])));
}
```

We can futher optimize using union by rank. In the previous implementation, we always join the second one to the first one. However, we can choose the best side to make it faster. We can base on the depth of the trees to determine which side we would like to attach.

```
bool unite(int x, int y) {
x = get(x);
y = get(y);
if (x != y) {
if (rank[x] > rank[y]) {
root[y] = x;
} else if (rank[x] < rank[y]) {
root[x] = y;
} else {
```

```
root[y] = x;
rank[x] += 1;
}
cnt--;
return true;
}
return false;
}
```

Here's the final templatized version.

```
class dsu {
public:
vector<int> root, rank;
int n;
int cnt;
dsu(int _n) : n(_n) {
root.resize(n);
rank.resize(n);
for(int i = 0; i < n; i++) {
root[i] = i;
rank[i] = 1;
}
cnt = n;
}
inline int getCount() { return cnt; }
inline int get(int x) { return (x == root[x] ? x : (root[x] = get(root[x]))); }
inline bool unite(int x, int y) {
x = get(x);
y = get(y);
if (x != y) {
if (rank[x] > rank[y]) {
root[y] = x;
} else if (rank[x] < rank[y]) {
root[x] = y;
} else {
root[y] = x;
rank[x] += 1;
}
cnt--;
return true;
}
return false;
}
};
```

Here's some basic usages.

```
int main() {
int n = 10;
// init
dsu d = dsu(n);
// unite
d.unite(1, 2);
d.unite(3, 4);
d.unite(1, 4);
// get the parent
int p = d.get(1);
return 0;
}
```

Suggested Problems

Problem Name	Difficulty	Solution Link
1061. Lexicographically Smallest Equivalent String	Medium	View Solutions
2421. Number of Good Paths	Hard	View Solutions
2382. Maximum Segment Sum After Removals	Hard	N/A

Kruskal's Algorithm

Author: @wingkwong

Overview

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree of a connected, undirected graph. The algorithm starts with each vertex in its own separate connected component, and iteratively adds edges to the MST in increasing order of weight, while ensuring that adding the edge does not form a cycle.
Here are the steps to find the MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Initialize the MST as an empty set.
3. For each edge in the sorted list of edges:
   - If adding the edge does not form a cycle in the MST, add the edge to the MST.
   - Otherwise, discard the edge.
4. Repeat steps 3 until all the vertices are included in the MST.

The time complexity of Kruskal's algorithm is $O(E log E)$ where E is the number of edges in the graph. It is more efficient than Prim's algorithm when the number of edges is much larger than the number of vertices.

Implementation

PREREQUISITE

- Disjoint Set Union
- C++

Written by @wingkwong

```
class dsu {
public:
vector<int> root, rank;
int n;
int cnt;
dsu(int _n) : n(_n) {
```

```
root.resize(n);
rank.resize(n);
for(int i = 0; i < n; i++) {
root[i] = i;
rank[i] = 1;
}
cnt = n;
}
inline int getCount() { return cnt; }
inline int get(int x) { return (x == root[x] ? x : (root[x] = get(root[x]))); }
inline bool unite(int x, int y) {
x = get(x);
y = get(y);
if (x != y) {
if (rank[x] > rank[y]) {
root[y] = x;
} else if (rank[x] < rank[y]) {
root[x] = y;
} else {
root[y] = x;
rank[x] += 1;
}
cnt--;
return true;
}
return false;
}
};
int mst(vector<vector<int>>& g) {
int n = (int) g.size();
vector<array<int, 3>> edges;
// g[i] = {from, to, weight}
for (auto x : g) edges.push_back({x[2], x[0], x[1]});
sort(edges.begin(), edges.end());
```

```
dsu d(n + 1);
int minimum_weight = 0;
for (auto x : edges) {
if (d.unite(x[1], x[2])) {
minimum_weight += x[0];
}
}
return minimum_weight;
}
```

Suggested Problems

Problem Name	Difficulty	Solution Link
1135 - Connecting Cities With Minimum Cost	Medium	View Solutions
1168 - Optimize Water Distribution in a Village	Hard	View Solutions

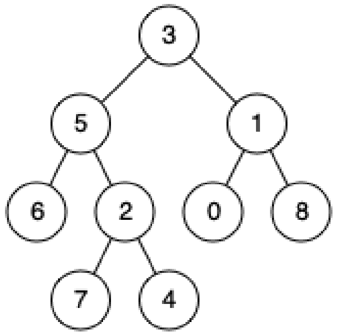
Lowest Common Ancestor(LCA)

Author: @RohitTaparia

Contributor: @wingkwong

Overview

Lowest common ancestor (LCA) of two nodes x and y in a tree or directed acyclic graph (DAG) is the deepest(lowest) node that has both x and y as descendants. Hence, LCA is the ancestor of x and y which is the farthest from the root node in a tree. In most cases, we also consider a node to be a descendant of itself. We have assumed this fact for this article,i.e, $$LCA(x,x)=x$$. Also, the $$LCA(x,y)$$ is a node that surely lies on the shortest path between $$x$$ and $$y$$, since if there was a smaller path, there would surely be a node at a lower depth which is their mutual ancestor, and this cannot be possible as if it was, this node would have been the LCA.



In this example, for nodes 7 and 4, the LCA is 2.
For nodes 6 and 4, the LCA is 5.
For nodes 4 and 8, the LCA is the root itself, i.e. 3.

$$NOTE:$$ The LCA in a binary tree for the root with any other node will be the root itself.
One of the most common applications of LCA is to determine the distance between pairs of nodes in a binary tree(or any other tree for that matter).In the above example, the distance between 6 and 4 can be computed as can be computed as the distance from 6 to $root(3)$, plus the distance from the 4 to $root(3)$, minus twice the distance from the root to their lowest common ancestor(LCA), that is,
$$ dist(x,y) = dist(x, root) + dist(y, root)- 2 * dist(LCA, root) $$

Implementation

We can notice from the definition of LCA that the LCA of two nodes $$x$$ and $$y$$ is nothing but the node of the intersection of the paths from $$x$$ and $$y$$ to the root node. In the tree above, the paths from 7 and 6 to the root node have their first intersection at 5. Hence, $$LCA(7, 6) = 5$$. We can calculate the paths using DFS and find the intersection using a stack based approach, or using a recursive approach. This is the general(naive) solution, and takes $$O(N)$$ time and $$O(N)$$ space. Below is the code for the iterative approach using stacks.

- C++

Written by @RohitTaparia

```
int findingLCA(int x, int y, vector<int>& adj) {
// adj[i] represents parent node of i
int root = 0;
stack<int> x_path, y_path;
// find first path
while (x != root) {
x_path.push(x);
x = adj[x];
}
x_path.push(x);
// find second path
while (y != root) {
y_path.push(y);
y = adj[y];
}
y_path.push(y);
int lca = -1;
// find the last common node
while ((!x_path.empty() && !y_path.empty()) &&
(x_path.top() == y_path.top())) {
lca = x_path.top();
x_path.pop();
y_path.pop();
}
return lca;
}
```

The same logic can be implemented using recursion, so that we do not need to use stacks explicitly. We store paths from root to node $$x$$ and from root to node $$y$$ and then check iterate to the last common node, which is the LCA. Explicitly, what we are trying to do here is to find which is the last common node while traversing both the paths. Obviously the root will be common in both paths, since we assume that both nodes are present. Then we need to go to the common node which is the farthest from the root node. This we can do if we traverse both the paths. The last common node gives us the LCA.

- C++

Written by @RohitTaparia

```
// findLCA will return LCA only if both node x, y are present, else -1
int findLCA(Node* root, int x, int y) {
vector<int> path_root_to_x, path_root_to_y;
// if either x or y is not present return -1
if (!findPath(root, path_root_to_x, x) || !findPath(root, path_root_to_y, y))
return -1;
// check for LCA now, which is farthest common node from root in both paths
for (int i = 0; i < path_root_to_x.size() && i < path_root_to_y.size(); i++)
if (path_root_to_x[i] != path_root_to_y[i]) break;
return path_root_to_x[i - 1];
}
bool findPath(Node* root, vector<int>& current_path, int value) {
// if root is NULL, then no paths
if (root == NULL) return false;
current_path.push_back(root->key);
if (root->key == value) return true;
// check if value is found in left or right sub-tree
if ((root->left && findPath(root->left, current_path, value)) ||
(root->right && findPath(root->right, current_path, value)))
return true;
// remove root since not found in subtree
current_path.pop_back();
return false;
}
```

Example: 0235 -Lowest Common Ancestor of a Binary Tree

```
Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Let's start with a recursive solution. The idea is simple. We start from the root and start checking in the left and right subtree of every node(basically DFS). If the current subtree contains both p and q, i.e, neither of them is $$NULL$$, then the function will reuurn the root of this subtree which will be the LCA. If any one of them is $$NULL$$, then the function returns the other one. If both are $$NULL$$, then the result will also be $$NULL$$. The time complexity will be $$O(N)$$ and space somplexity would be $$O(N)$$, since maximum height for a binary tree(skewed) will be $$N$$.

- C++

Written by @RohitTaparia

```
/**
* definition for a binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
// base case to check if the root is null or
// one of the required nodes is the root itself
// used the recursive implementation discussed earlier
if (root == NULL) {
return root;
}
if (root == p || root == q) {
return root;
}
// recurse for the left subtree, basically dfs
TreeNode* left = lowestCommonAncestor(root->left, p, q);
// recurse for the right subtree
TreeNode* right = lowestCommonAncestor(root->right, p, q);
// if one of them is NULL means we need to return the other one
if (left == NULL) {
return right;
} else if (right == NULL) {
return left;
} else {
// when both left and right are not null, we can say that this is the LCA
return root;
}
}
};
```

Example: 2096. Step-By-Step Directions From a Binary Tree Node to Another

```
You are given the root of a binary tree with n nodes. Each node is uniquely assigned a
value from 1 to n. You are also given an integer startValue representing the value of the
start node s, and a different integer destValue representing the value of the destination
node t.
Find the shortest path starting from node s and ending at node t. Generate step-by-step
directions of such path as a string consisting of only the uppercase letters 'L', 'R', and
'U'. Each letter indicates a specific direction:
'L' means to go from a node to its left child node.
'R' means to go from a node to its right child node.
'U' means to go from a node to its parent node.
Return the step-by-step directions of the shortest path from node s to node t.
Input: root = [5,1,2,3,null,6,4], startValue = 3, destValue = 6
```

```
Output: "UURL"
Explanation: The shortest path is: 3 → 1 → 5 → 2 → 6.
```

In this problem, we need to find the closest point(from nodes), where path from root to nodes intersect, which is LCA of both the nodes. Hence, we first find the LCA node of start and destination. Then we need to get path from LCA to the starting node($$lca_s$$) and from LCA to destination($$lca_d$$). This method has also been explained above. In this we simply do a simple DFS and first explore the left path, and then the right path. Whenever we find the node, we return true, otherwise we backtrack and explore the right path. Now that we have both paths, we will convert all chars in $$lca_s$$ to $$U$$, since we need to move upward.
At last, we concatenate both strings and return the combined path.

- C++

Written by @RohitTaparia

```
/**
* definition for a binary tree node.
* struct TreeNode {
* int val;
* TreeNode *left;
* TreeNode *right;
* TreeNode(int x) : val(x), left(NULL), right(NULL) {}
* };
*/
class Solution {
public:
// function to get LCA of given two nodes
// used the recursive implementation discussed earlier
TreeNode* getLCA(TreeNode* root, int start, int dest) {
if (!root) return NULL;
if (root->val == start || root->val == dest) return root;
// recurse for left subtree
TreeNode* left = getLCA(root->left, start, dest);
// recurse for right subtree
TreeNode* right = getLCA(root->right, start, dest);
// if both are not null, this node is LCA
if (left && right) return root;
// else return the node which is not NULL
else if (left) {
return left;
}
return right;
}
bool findPath(TreeNode* root, string& path, int val) {
if (!root) return false;
// if node is found, we can return true
if (root->val == val) return true;
// try to find node for left
path.push_back('L');
if (findPath(root->left, path, val)) return true;
path.pop_back();
// try to find node for right
path.push_back('R');
if (findPath(root->right, path, val)) return true;
path.pop_back();
return false;
}
string getDirections(TreeNode* root, int initialValue, int finalValue) {
// get LCA of start and destination node
TreeNode* lca = getLCA(root, initialValue, finalValue);
string lcaS = "", lcaD = "";
// find both paths
findPath(lca, lcaS, initialValue);
findPath(lca, lcaD, finalValue);
```

```
for (auto& c : lcaS) c = 'U';
// merge both paths,
// i.e. start node -> destination node
return lcaS + lcaD;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
1123. Lowest Common Ancestor of Deepest Leaves	Medium	N/A
235. Lowest Common Ancestor of a Binary Search Tree	Medium	N/A

Minimum Spanning Tree

Author: @wingkwong

Overview

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. There are different algorithms that can be used to find the MST of a graph, such as Kruskal's algorithm, Prim's algorithm and Boruvka's algorithm.

Kruskal's Algorithm

See Here

Prim's Algorithm

Not Available Yet

Boruvka's Algorithm

Not Available Yet

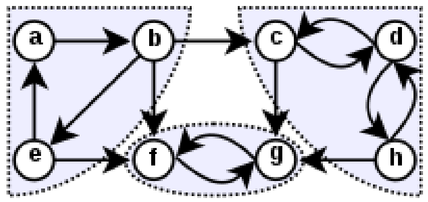
Tarjan's Algorithm

Author: @BlackPanther112358

Contributor: @wingkwong

Overview

Tarjan's Algorithm is popular algorithm for finding the Strongly Connected Components (SCC) of a directed graph. In the below graph, the nodes in a blue envelope constitute a single connected component as any node u as a path to another node v and vice versa. Note that while we can reach the node f from the node e, the opposite is not true. Hence, f and e are not part of the same component. Thus, the following graph has 3 strongly connected components as highlighted. A single strongly connected component can be formally described as maximal set of vertices such that for any 2 vertices belonging to the set, say u and v, there exists a path from u to v and vice versa.



$\$ Source: Strongly connected component from Wikipedia

In this tutorial we will discuss the Tarjan's Algorithm to find SCC.

Implementation

The algorithm works by using a single DFSin the graph. To understand its working first let's define a few standard terms :

$in-time(u) =$ time at which node u was reached for the first time or the in time of DFS for the node.

$low-link(u) =$ smallest time reachable of a node reachable from the DFS subtree of node u

We also need to ensure that we don't mix 2 different SCC's due to a cross-edge between them. To counter this issue, we will use a stack to store the nodes which have not been assigned to an SCC and have been visited so far.
Thus, whenever we find an SCC, we will pop the corresponding nodes from the stack.
We will call the first node we discovered of an SCC the route node for sake of simplicity.
Thus, we will identify the SCC by its root node. To check when we have reached a root node, we just check if $low(u)$ and $in_time(u)$ are equal.
The pseudo-code for the algorithm can be found here.
The Time and Space Complexity of the algorithm is $O(V + E)$, where V represents the number of vertices and E represents the number of edges, same as that os DFS.
The implementation of above can be as follows (along with above graph as example) :

- C++

```
#include <bits/stdc++.h>
using namespace std;
struct find_SCC {
int timer = 0;
// store the in-time for DFS search
vector<int> in_time;
// stores the low-link value for every node
vector<int> low_link;
// checks whether a node is on the stack or not
vector<bool> on_stack;
// stack to store the currently available nodes
stack<int> stk;
// to store the final answer
vector<vector<int>> res;
// recursive function to do the DFS search on the graph
void dfs(int u, vector<vector<int>> &graph) {
// set the values for in_time and low_link, and put the node on stack
```

```
timer++;
stk.push(u);
on_stack[u] = true;
// DFS to neighbours of current node
for (int v : graph[u]) {
// if the node is unvisited
if (in_time[v] == -1) {
dfs(v, graph);
// update the low-link value for node u
low_link[u] = min(low_link[u], low_link[v]);
// else if the node was visited before, and is still on the stack
} else if (on_stack[v]) {
// update the low-link for node u
low_link[u] = min(low_link[u], in_time[v]);
}
}
// check if u is the root node for a SCC
if (low_link[u] == in_time[u]) {
vector<int> SCC;
// all the nodes above u in the stack are in SCC of u
while (stk.top() != u) {
int v = stk.top();
stk.pop();
SCC.push_back(v);
on_stack[v] = false;
}
// now removing u from stack and adding it to the SCC
stk.pop();
SCC.push_back(u);
on_stack[u] = false;
// adding the SCC to the answer
res.push_back(SCC);
}
return;
```

```
}
// takes input of graph as adjacency list and returns the SCC of graph as
// vectors
vector<vector<int>> tarjans(vector<vector<int>> &adjacencyList) {
int n = adjacencyList.size();
in_time.resize(n, -1);
low_link.resize(n, -1);
on_stack.resize(n, false);
for (int u = 0; u < n; u++) {
if (in_time[u] == -1) dfs(u, adjacencyList);
}
return res;
}
};
int main() {
// constructing adjacency list for example graph shown, mapping node a to 0, b
// to 1 and so on...
vector<vector<int>> graph(8);
graph[0].push_back(1);
graph[1].push_back(2);
graph[1].push_back(4);
graph[1].push_back(5);
graph[2].push_back(3);
graph[2].push_back(6);
graph[3].push_back(2);
graph[3].push_back(7);
graph[4].push_back(0);
graph[4].push_back(5);
graph[5].push_back(6);
graph[6].push_back(5);
graph[7].push_back(3);
graph[7].push_back(6);
// using the tarjan's algo
find_SCC t = find_SCC();
```

```
vector<vector<int>> res = t.tarjans(graph);
// output the final result
cout << "The Strongly Connected Components for the graph are:" << endl;
for (vector<int> i : res) {
for (int j : i) {
cout << (char)('a' + j) << " ";
}
cout << endl;
}
/*
output of the above code is:
The Strongly Connected Components for the graph are:
f g
h d c
e b a
*/
return 0;
}
```

We can further cluster all the nodes belonging to an SCC into one node, and represent all the edges from and to a constituent node of the SCC to this new node. The resulting graph is called the Condensation Graph

Using Tarjan's algorithm to find bridges in a Undirected Graph

Here, we are searching in an undirected graph, and also we don't need to remove the cross-edges. Thus, the stack is no longer required. An edge between node u to v will be considered a bridge if there is no other way to reach v from u if the edge is removed. We can check for this by using the properties of the low-link time. If the low-link time of v is less than that of u, we can conclude that we can reach v from some other path, otherwise the edge between them is a bridge. Here is an example for same.

Example #1: 1192 - Critical Connections in a Network
This problem directly asks us to find bridges in the given graph. The input is provided in form of pairs of nodes with an edge between them. Thus, we will first convert this to an adjacency list and apply the same algorithm discussed above.

- C++

```
class Solution {
public:
// initializing the variables
int timer = 0;
vector<int> in_time, low_link;
vector<vector<int>> graph;
vector<vector<int>> res;
// recursive function to perform DFS
void dfs(int u, int p = -1) {
in_time[u] = low_link[u] = timer;
timer++;
for (int v : graph[u]) {
// we ignore the parent node
if (v == p) continue;
// if we discover a new node
if (in_time[v] == -1) {
dfs(v, u);
low_link[u] = min(low_link[u], low_link[v]);
// check if the edge is a bridge
if (low_link[v] > in_time[u]) {
res.push_back({u, v});
}
} else {
```

```
low_link[u] = min(low_link[u], in_time[v]);
}
}
return;
}
vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
in_time.resize(n, -1);
low_link.resize(n, -1);
graph.resize(n);
// make an adjacency list from the input
for (auto connection : connections) {
graph[connection[0]].push_back(connection[1]);
graph[connection[1]].push_back(connection[0]);
}
// as the entire graph is connected, just call dfs on 0
dfs(0);
return res;
}
};
```

Using Tarjan's algorithm to find articulation points in a Undirected Graph

Similar to bridges, articulation points are vertices which if removed will disconnect the graph. Once again, we can modify the Tarjan's algorithm to find such vertices in a given undirected graph.

Example #2: 1568 - Minimum Number of Days to Disconnect Island

Notice that the answer cannot be more than 2. Any rectangular shape will have atleast 4 corner cells, and if we remove the vertical neighbour and horizontal neighbour cells of

anyone one of these, we have disconnected the graph. Thus, we need to check when the answer can 0 or 1, and we are done.
The answer will be 0 if the islands are initially disconnected. This can be checked using basic DFS. The answer will be 1 when the graph has atleast 1 articulation point. The condition for articulation points is similar to that for finding bridges, except we also check for presence of parents.

- C++

```
class Solution {
public:
// initializing the variables
int n, m, cnt_islands = 0;
int timer = 0;
vector<vector<int>> in_time, low_link;
// recursive function to perform DFS, return true if an articulation point is
// detected
bool dfs(pair<int, int> u, vector<vector<int>>& grid,
pair<int, int> p = {-1, -1}) {
int i = u.first, j = u.second;
in_time[i][j] = low_link[i][j] = timer++;
cnt_islands++;
// variable to check for articulation point in DFS subtree of the node
bool has_articulation_point = false;
// variable to count number of children visited first by the node
int cnt_children = 0;
// find all neighbours from the grid
vector<pair<int, int>> neighbours;
if ((i > 0) && (grid[i - 1][j])) neighbours.push_back({i - 1, j});
if ((i < (n - 1)) && (grid[i + 1][j])) neighbours.push_back({i + 1, j});
if ((j > 0) && (grid[i][j - 1])) neighbours.push_back({i, j - 1});
if ((j < (m - 1)) && (grid[i][j + 1])) neighbours.push_back({i, j + 1});
for (auto v : neighbours) {
```

```
// if the neighbour is a parent, ignore it
if (v == p) continue;
// if the neighbour was already visited
else if (in_time[v.first][v.second] != -1)
low_link[i][j] = min(low_link[i][j], in_time[v.first][v.second]);
// if the neighbour is a new node
else {
// if the subtree has an articulation point
if (dfs(v, grid, u)) has_articulation_point = true;
// update the low-link value
low_link[i][j] = min(low_link[i][j], low_link[v.first][v.second]);
// if the point itself is an articulation point
if ((low_link[v.first][v.second] >= in_time[i][j]) &&
(p != (pair<int, int>){-1, -1}))
has_articulation_point = true;
cnt_children++;
}
}
// if it has an articulation point, return true
if (((p == (pair<int, int>){-1, -1}) && (cnt_children > 1)) ||
has_articulation_point)
return true;
// else return false
return false;
}
int minDays(vector<vector<int>>& grid) {
n = grid.size();
m = grid[0].size();
int connected_comp = 0;
bool has_articulation_point = false;
in_time.resize(n, vector<int>(m, -1));
low_link.resize(n, vector<int>(m, -1));
for (int i = 0; i < n; i++) {
for (int j = 0; j < m; j++) {
```

```
// ignore the cells of grid with water, or who have been visited
if ((grid[i][j] == 0) || (in_time[i][j] != -1)) continue;
// we already have found a connected component, and this will be a new
// one, thus we directly return 0
if (connected_comp > 0) return 0;
if (dfs({i, j}, grid)) has_articulation_point = true;
connected_comp++;
}
}
// if there are no cells with land
if (cnt_islands == 0) return 0;
// if there is only one cell with land
else if (cnt_islands == 1)
return 1;
if (has_articulation_point) return 1;
return 2;
}
};
```

Suggested Problems

Problem Name	Difficulty	Solution Link
1489 - Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree	Hard	N/A

References

1. Tarjan's Algorithm For Strongly Connected Components
2. Tarjan's strongly connected components algorithm

Topological Sorting

Author: @wingkwong

Overview

Topological Sorting is a linear ordering of its vertices such that for every directed edge $(u, v)$ from vertex $u$ to vertex $v$, $u$ comes before $v$ in the ordering.

In order to find the order, we start from those nodes which do not have any prerequisites / dependencies. In other word, those nodes with indegree 0. Then we incrementally add the nodes to the order following the given prerequisites. For each node with an edge, we remove the edge from the graph. By doing so, there would be more nodes without dependency. At the end, there is no edges that can be removed, which gives two possible results. The first one is a cycle is form which cannot remove in above steps. The second one is all the edges from the graph have been removed and we got the topological order of the graph.
The time complexity would be $O(|E| + |V|)$.

Implementation

The following implementation is using BFS.

- Gis the graph built with the dependencies
- indegreeis used to record the indegree of given node
- ordersis the topologically sorted order
- isTopologicalSortedis used to determine if the graph can be topologically sorted or not
- C++

Written by @wingkwong

```
struct TopologicalSort {
int n;
vector<int> indegree;
vector<int> orders;
vector<vector<int>> G;
bool isTopologicalSorted = false;
TopologicalSort(vector<vector<int>>& g, vector<int>& in) {
G = g; vector<vector<int>>
n = (int) G.size();
indegree = in;
int res = 0;
queue<int> q;
for(int i = 0; i < n; i++) {
if(indegree[i] == 0) {
q.push(i);
}
}
while(!q.empty()) {
auto u = q.front(); q.pop();
orders.push_back(u);
for(auto v : G[u]) {
if(--indegree[v] == 0) {
q.push(v);
}
}
res++;
}
isTopologicalSorted = res == n;
}
};
```

Example 1: [0207 - Course Schedule]

- C++

Written by @wingkwong

```
// ...
// TopologicalSort implementation here
// ...
class Solution {
public:
```

```
bool canFinish(int n, vector<vector<int>>& prerequisites) {
// define the graph
vector<vector<int>> g(n);
// define indegree
vector<int> indegree(n);
// build the graph
for(auto p : prerequisites) {
// we have to take p[1] in order to take p[0]
g[p[1]].push_back(p[0]);
// increase indegree by 1 for p[0]
indegree[p[0]]++;
}
// init topological sort
TopologicalSort ts(g, indegree);
// we can finish all courses only if we can topologically sort
return ts.isTopologicalSorted;
}
};
```

Example 2: 0210 - Course Schedule II

- C++

Written by @wingkwong

```
// ...
// TopologicalSort implementation here
// ...
class Solution {
public:
vector<int> findOrder(int n, vector<vector<int>>& prerequisites) {
// define the graph
vector<vector<int>> g(n);
// define indegree
vector<int> indegree(n);
// build the graph
for(auto p : prerequisites) {
// we have to take p[1] in order to take p[0]
g[p[1]].push_back(p[0]);
// increase indegree by 1 for p[0]
indegree[p[0]]++;
}
// init topological sort
TopologicalSort ts(g, indegree);
// we can finish all courses only if we can topologically sort
// hence, return an empty array if it cannot be sorted
if (!ts.isTopologicalSorted) return {};
// else return the order
return ts.orders;
}
};
```

Math

Number Theory

Binary Exponentiation

Author: @wingkwong

Overview

Binary Exponentiation is a method for efficiently calculating large powers of a number, such as a^n. Instead of using the naive approach of repeatedly multiplying the base number by itself, which has a time complexity of $O(n)$, binary exponentiation uses a technique called "exponentiation by squaring" to accomplish the same task in $O(log n)$ time complexity.
The basic idea behind binary exponentiation is that we can express $$a ^ n$$ as $$a *a ... a$$ but it is not efficient for a large $$a$$ and $$n$$. If we display the exponent in binary representation, says $13 = 1101_2$, then we have $$3 ^{13} = 3^83^4*3^1.$$*
Supposing we have a sequence $$a ^ 1, a ^ 2, a ^4, ..., a^{\lfloor log_2 n\rfloor}$$, we can see the an element in the sequence is the square of previous element, i.e. $$3 ^ 4 = (3^2)^2$$. Therefore, to calculate $$3 ^ {13}$$, we just need to calculate $${\lfloor log_2 13\rfloor} = 3$$ times, i.e. (1 -> 4 -> 8). We skip 2 here because the bit is not set. This approach gives us $$O(log n)$$ complexity.
To generalise it, for a positive integer $$n$$, we have

$$x^n = \begin{cases} x\,(x^2)^{\frac{n-1}{2}}, & \text{if } n \text{ is odd} \\ (x^2)^{\frac{n}{2}}, & \text{if } n \text{ is even.} \end{cases}$$

Implementation

- C++

Written by @wingkwong

```
long long fastpow(long long base, long long exp) {
long long res = 1;
while (exp > 0) {
// if n is odd, a ^ n can be seen as a ^ (n / 2) * a ^ (n / 2) * a
if (exp & 1) res *= base;
// if n is even, a ^ n can be seen as a ^ (n / 2) * a ^ (n / 2)
base *= base;
// shift 1 bit to the right
exp >>= 1;
}
return res;
}
```

In case you need to take mod during the calculation, we can do as follows.

- C++

Written by @wingkwong

```
long long modpow(long long base, long long exp, long long mod) {
base %= mod;
long long res = 1;
while (exp > 0) {
if (exp & 1) res = (res * base) % mod;
base = (base * base) % mod;
exp >>= 1;
}
return res;
}
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0050 - Pow(x, n)	Medium	View Solutions

Sieve of Eratosthenes

Author: @wingkwong

Overview

The Sieve of Eratosthenes is an algorithm used to find all prime numbers up to a given limit. It works by iteratively marking as composite (i.e., not prime) the multiples of each prime, starting with 2. The algorithm starts by creating a list of all integers from 2 to the limit. It then marks the first number, 2, as prime and removes all multiples of 2 from the list. The next unmarked number in the list is 3, which is also prime, so it marks it and removes all multiples of 3 from the list. This process continues until all numbers in the list have been marked as prime or composite. The remaining unmarked numbers are the prime numbers up to the given limit.

Implementation

- C++

Written by @wingkwong

```
vector<bool> sieveOfEratosthenes(const int n) {
vector<bool> isPrime(n + 1, true);
isPrime[0] = isPrime[1] = false;
for (int i = 2; i * i <= n; i++) {
if (isPrime[i]) {
for (int j = i * i; j <= n; j += i) {
isPrime[j] = false;
}
}
}
return isPrime;
}
```

Suggested Problems

Problem Name	Difficulty	Solution Link
0204 - Count Primes	Medium	View Solutions

Bit Manipulation

Author: @wingkwong

Overview

We all know that information is stored in the form of bits in computer memory, which means directly manipulating these bits will yield faster results than performing computation on ordinary data.

Binary uses only 0 and 1 to represent a number in a base-2 number system. The series of 0 and 1 are called bits. If the bit is 1, then this bit is set. We read binary number from right to left. For example, the binary representation of number 9 is $1001*2$ which can be calculated by summing up all the set bit: $2^3 + 2^0 = 9*{10}$. Bit Manipulation utilises different bitwise operations to manipulate bits.

Bitwise Operators

X	Y	X & Y	X | Y	X ^ Y	~ X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

AND (&) Operator

& takes two bit integers to compare. If the bits are both 1, then the resulting bit is 1, else 0.
For example, 0010_2 & $0011_2 = 0010_2$ because only the second bits from the right are both 1.

Usage #1: Check if the rightmost bit is set

Let's say n is 5_{10} which is 0101_2. If we execute n & 1, i.e. 0101_2 & 0001_2, the result is 0001_2 because only the rightmost bits are both 1 and other bits would return 0.

Usage #2: Check if the i-th bit is set

Let's say n is 5_{10} which is 0101_2. How to check if the 2-nd, 3-rd, or 4-th bit is set? Using the same idea, the mask would be 0010_2, 0100_2, 1000_2 respectively. And you may notice that the mask is always a power of 2. A common way to do it is to use left shift operator (which will be discussed below), i.e. n & $(1 << i)$ where n is the i-th bit to be checked.

Usage #3: Remove the rightmost set bit

We can use n & $(n - 1)$ to remove the rightmost set bit.

```
n n n - 1 n & (n - 1)
-- ---- ---- -------
0 0000 0111 0000
1 0001 0000 0000
2 0010 0001 0000
3 0011 0010 0010
4 0100 0011 0000
5 0101 0100 0100
6 0110 0101 0100
7 0111 0110 0110
8 1000 0111 0000
9 1001 1000 1000
10 1010 1001 1000
11 1011 1010 1010
12 1100 1011 1000
13 1101 1100 1100
14 1110 1101 1100
```

Example #1: [0231 - Power of Two (Easy)]

We know that a power of 2 is a positive number and only has one bit set. We can use n & $(n - 1)$ to see the result is 0 or not to determine if the target value is a power of 2 or not.

In short, n & $(n - 1)$ never have a 1 bit in the same place.

For example, 1000 (8), and 0111(7) - never have 1 bit in the same place for Power of Two

```
class Solution {
public:
bool isPowerOfTwo(int n) {
// 1. check if it is a positive number
// 2. check the value is 0 after removing the rightmost bit
return n > 0 && !(n & (n - 1));
}
};
```

Example #2: [0191 - Number of 1 Bits]

Instead of checking the bits one by one, we can use n & $(n - 1)$ to jump to the next set bit.

```
class Solution {
public:
int hammingWeight(uint32_t n) {
int ans = 0;
for (; n; n = n & (n - 1)) ans++;
return ans;
}
};
```

OR (|) Operator

$\vert$ takes two bit integers to compare. If either bits are 1, then the resulting bit is 1, else 0.
For example, $0010_2 | 0011_2 = 0011_2$ because only the first and the second bits from the right are 1 in either value.

Usage #1: Set the rightmost bit

Let's say n is 4_{10} which is 0100_2. If we execute $n \vert 1$, i.e. $0100_2 \vert 0001_2$, the result is 0101_2 because 0001_2 would turn the rightmost bit of 0100_2 to 1 since that bit is set.

Usage #2: Set the i-th bit

Let's say n is 4_{10} which is 0100_2. How to set other bits? Similar to above example, we can use left shift operator (which will be discussed below), i.e. $n \vert (1 << i)$ where n is the i-th bit to be set.

Example: 0421 - Maximum XOR of Two Numbers in an Array

```
class Solution {
public:
int findMaximumXOR(vector<int>& nums) {
int ans = 0, mask = 0;
for(int i = 31; i >= 0; i--){
unordered_set<int> s;
// set i-th in mask
mask |= (1 << i);
for (auto x : nums) s.insert(mask & x);
// set i-th in ans
int best = ans | (1 << i);
for(auto pref : s){
if(s.find(pref ^ best) != s.end()){
ans = best;
break;
}
}
}
return ans;
}
};
```

XOR (^)

^ takes two bit intergers to compare. If one bit is zero and another bit is one, then the resulting bit is 1, else 0.
For example, 0010_2 ^ $0011_2 = 0001_2$ because the first bit got 0 and 1 while the second bit got both 1.

XOR Properties

- X ^ 0 = X
- X ^ X = 0
- X ^ 1 = ~X (Compliment of that number)
- X ^ Y = Y ^ X (Commutativity)
- (X ^ Y) ^ Z = X ^ (Y ^ Z) (Associativity)

Example #1: 0268 - Missing Number

Given the fact that we know n distinct numbers in the range $[0, n]$, we can find the missing number using the above XOR properties.

For example, let's say the input is $[0, 1, 3]$ and we know the the missing number is 2. We can compare the index (0, 1, 2) and the value (0, 1, 3) and write 3 ^ $(0$ ^ $0)$ ^ $(1$ ^ $1)$ ^ $(2$ ^ $3)$.
Based on property #2, we know 0 ^ 0 and $(1$ ^ $1)$ would be 0.
Based on property #1, we know that 0 ^ 1 would be 1. Therefore, we got 3 ^ $(2$ ^ $3)$.
Based on property #4, we can rewrite as 2 ^ $(3$ ^ $3)$ and use property #2 again to get 2 ^ 0.
Based on property #1, we have our final answer which is 2.

```
class Solution {
public:
int missingNumber(vector<int>& nums) {
// index = 0 1 2
// value = 0 1 3
// n ^ (0 ^ 0) ^ (1 ^ 1) ^ (2 ^ 3)
// 3 ^ 0 ^ 0 ^ 2 ^ 3
// 0 ^ 0 ^ 2 ^ 3 ^ 3
// 2 ^ 0
// 2
int n = nums.size(), ans = n;
for(int i = 0; i < n; i++) ans ^= (i ^ nums[i]);
return ans;
}
};
```

Example #2: 0136 - Single Number

As every element appears twice except for one. We can use property #2 to make all elements which appear twice become 0. At the end, there would be 0 and that element which appears once. Then we use property #1 to get the final answer.

```
class Solution {
public:
int singleNumber(vector<int>& nums) {
// nums: [4,1,2,1,2]
// 0 ^ 4 ^ 1 ^ 2 ^ 1 ^ 2
// 0 ^ (1 ^ 1) ^ (2 ^ 2) ^ 4
// 0 ^ 0 ^ 0 ^ 4
// 4
int ans = 0;
for(auto x : nums) ans ^= x;
return ans;
}
};
```

Example #3: Swap 2 numbers

XOR (^) can be used to swap 2 numbers by changing the bits and reversing it.
Let's say, $a = 4$ (0100_2), $b = 6$ (0110_2), we want $a = 6$ and $b = 4$ as our answer.

```
a = a ^ b => (4 ^ 6) => (0100 ^ 0110) => 2 (0010)
b = b ^ a => (6 ^ 2) => (0110 ^ 0010) => 4 (0100)
a = a ^ b => (2 ^ 4) => (0010 ^ 0100) => 6 (0110)
```

NOT (~)

~ inverts (flip) all the bits of a bit intergers, which means 1 would become 0 and vice versa. If we apply it on a positive integer x, then it is simply $-x-1$.
For example, if we apply NOT on $~0010_2$, then the resulting value is 1101_2.

The below code snippets are actually equivalent

```
for (int i = n; i >= 0; i--) {
// i = n, n - 1, n - 2, ..., 2, 1, 0
}
```

```
for (int i = n; ~i; i--) {
// i = n, n - 1, n - 2, ..., 2, 1, 0
}
```

Left-Shift (<<)

$<<$ shifts the bits to the left. For example, $1 << 1 = 2$ because we shift the 1 (0001_2) to the left to become 2 (0010_2).
Similarily, $1 << 2 = 4$ because we shift the 1 (0001_2) to the left twice to become 4 (0100_2).
And you may find that $1 << n$ is actually $2 ^ n$. Also $n << m$ means multiplying n by 2 power m. i.e, $n = n * (2^m)$.
In simple, $n << m$ *shifting each bit of n to left m times*. Let's say, $n = 8$ and $m = 2$. n can be represented as 1000_2 in binary, Therefore, $8 << 2$ = 1000_2 << 2 = 100000_2 (32), which is same as $(8 * 2^2)$ = 32

Example #1: 0078 - Subsets

```
class Solution {
public:
vector<vector<int>> subsets(vector<int>& nums) {
int n = nums.size();
// number of subsets for n elements would be 2 ^ n
// because for each element, you can choose to take it or not
// if take = 1, don't take = 0, then we can use bit manipulation
int p = 1 << n; // 1 * 2 ^ n
vector<vector<int>> ans;
for(int i = 0; i < p; i++){
vector<int> t;
for(int j = 0; j < n; j++){
if((1 << j) & i) t.emplace_back(nums[j]);
}
ans.emplace_back(t);
}
return ans;
}
};
```

Right-Shift (>>)

$>>$ shifts the bits to the right. For example 3_{10} (0011_2) $>> 1$ would become 1 (0010_2).
$4 >> 1$ dividing 4 by 2. Also $n >> m$ means dividing n by 2 power m. i.e, $n = n / (2^m)$
In simple, $n >> m$ *shifting each bit of n to right m times*. Let's say, $n = 8$ and $m = 2$. n can be represented as 1000_2 in binary, Therefore, $8 >> 2$ = 1000_2 >> $2$$ = 0010_2 (4), which is same as $(8 / 2^2)$ = 4

Example: Check the bits one by one

```
while (n > 0) {
int bit = n & 1;
// do something with bit
// ...
// shift bits to the right
// which is same as n /= 2
n >>= 1;
}
```

Two's Compliment and Negative Numbers

Computers typically store integers in two’s complement representation. A positive number is represented as itself while a negative number is represented as the two’s complement of it’s absolute value (with a 1 in its sign bit to indicate that a negative value).

Let’s look at the 4-bitinteger -3 as an example. *If it’s a 4-bit number, we have one bit for the sign and three bits for the value*. We want the complement with respect to 2^3, which is 8.

The complement of 3 (the absolute value of -3) with respect to 8 is 5. Binary value of 5 is 0101_2. Therefore, -3 in binary as a 4-bit number is 1101_2, with the first bit being the sign bit.

In other words, the binary representation of $-K$ (negative K) as a N-bit number is $concat(1, 2^N-1 - K)$ otherwise *prepend (prefix) the sign bit*(1) with the value for the calculated complement.

Another way to look at this is that we invert the bits in the positive representation and then add 1. 3 is 0011_2 in binary.

*Flip the bits*to get 100, add 1 to get 101_2, then prepend the sign bit 1 to get 1101_2.

```
Positive Values Negative Values
-------- ------ -------- ------
7 0 111 -1 1 111
```

```
6 0 110 -2 1 110
5 0 101 -3 1 101
4 0 100 -4 1 100
3 0 011 -5 1 011
2 0 010 -6 1 010
1 0 001 -7 1 001
0 0 000
```

Common Bit Tasks

Get Bit

It shifts 1 over by *i*bits, creating a value that looks like $0100*2$ (2^i). By performing and AND (&) with num, we clear all bits other than the bit at bit _i*.
Finally, compare that value to 0, if that new value is not *zero*, then bit *i*must have a 1, otherwise, bit *i*is a 0.

```
int getBit(int num, int i) {
return num & (1 << i);
}
```

Set Bit

It shits 1 over by *i*bits, creating a value that looks like $0100*2$ (2 to the power of i). By performing an OR with num, only the value at bit _i*will change.
All other bits of the mask are *zero*and will not affect num.

```
int setBit(int num, int i) {
return num | (1 << i);
}
```

Combinatorics

Author: @BlackPanther112358

Overview

Combinatorics is primarily involved with art of counting. This includes counting number of ways for a certain position to occur, to arrange some objects according to given rules or choosing some objects from a collection. Quite often, the required number takes a very large value, thus it is a very common practice to return the answer by taking modulo with some prime number p (which is $1e9 + 7$ quite often, and quite recently 998244353 also has become prominent). You can read more about modulo here.

Finding $n \choose r$

Pronounced "n choose r", this is the mathematical notation to represent number of ways to choose r objects out of a collection of n objects. The analytical formula can be written as $n \choose r$ $=$ $\frac{n!}{r!\,(n - r)!}$.
This leads to a neat recurrence relation: $n \choose r$ $=$ $n - 1 \choose r$ + $n - 1 \choose r - 1$ $\$
We can precompute all the required values using the above formula in $O(n^2)$ and then perform lookup in $O(1)$ time. The resulting values also form Pascal's Triangle.

Example #1: 2221 - Find Triangular Sum of an Array

The important insight here is that the figure provided is nothing but an inverted Pascal's Triangle and contribution of each cell in the final sum is the value of cell multiplied by the binomial coefficient at the particular position in Pascal's Triangle.
Thus for the cell at i^{th} index in the topmost row, it's value is multiplied by $n - 1 \choose i$ and added to the final sum $modulo\,10$. Time Complexity of the program is $O(n^2)$ for computing the binomial coefficient and $O(n)$ Space complexity.

- C++

```
class Solution {
public:
int triangularSum(vector<int>& nums) {
int n = nums.size();
vector<int> pascalTriangleRow = {1};
// calculate the ith row using (i - 1)th row
for (int i = 0; i < n; i++) {
vector<int> nextRow = {1};
for(int j = 1; j < i; j++){
nextRow.push_back((pascalTriangleRow[j] + pascalTriangleRow[j - 1]) % 10);
}
nextRow.push_back(1);
pascalTriangleRow = nextRow;
}
// calculate the final answer as discussed above
int ans = 0;
for (int i = 0; i < n; i++) {
ans += (nums[i] * pascalTriangleRow[i]) % 10;
}
return ans%10;
}
};
```

Sometimes it is not possible to calculate the entirety of Pascal's Triangle due to larger values of n. In this case, we begin by precomputing $x!$$\,$ $\forall$$\,$$x \in [{0, n}]$. Similarly, we will also precomputethe modular inverses. This can be achieved in $O(n)$ time. Thus we can now compute $n \choose r$ using the analytical equation presented earlier. You can read about modular inverses here
The implementation of above can be as follows:

- C++

```
struct comb{
int mod;
// make arrays to store the factorial and inverse factorial modulo m
vector<long long int> factorial;
vector<long long int> inverse_factorial;
// N is the maximum value possible of input
comb (int N, int mod_in = 1e9 + 7) {
// calculate values for factorial
mod = mod_in;
factorial.push_back(1);
for(int i = 1; i <= N; i++) factorial.push_back((factorial.back() * i) % mod);
// calculate values for inverse factorial
vector<long long int> inverse;
inverse.push_back(1);
inverse.push_back(1);
inverse_factorial.push_back(1);
inverse_factorial.push_back(1);
for (int i = 2; i <= N; i++) {
inverse.push_back((mod - ((mod/i) * inverse[mod%i]) % mod) % mod);
inverse_factorial.push_back((inverse_factorial[i - 1] * inverse[i]) % mod);
}
}
// function to calculate nCr(n, r)
long long int nCr(int n, int r){
```

```
if ((r < 0) || (r > n)) return 0;
return ((factorial[n] * inverse_factorial[r]) % mod * inverse_factorial[n - r]) % mod;
}
};
```

For further reading, you can visit cp-algorithms.

Finding the n^{th} catalan number

This is a very famous sequence of natural numbers and has a variety of applications.

- Number of ways to make balanced bracket sequences using n left and n right brackets.
- Number of ways to make binary trees
- Number of ways to form a mountain range with n upstrokes and downstrokes. $\$

Hereis a more exhastive list.

The n^{th} Catalan number can be found using the formula: C_n $=$ $\frac{1}{n + 1}$$2n \choose n$

Example #2: 1863 - Sum of All Subset XOR Totals

This is an example of a very tricky problem which heavily simplifies after using some Combinatorics and Bit Manipulation

Here we will consider the i^{th} bit from the right. Let's say that the i^{th} bit is set in k out of n numbers in some given subset. If k is odd, then i^{th} bit is set in the XOR of all numbers of the subset, otherwise, it is not set.

Hence if there are m numbers out of n with i^{th} bit set, then the contribution of the bit is $Place\,value\,of\,the\,bit$ $*$ $number\,of\,ways\,to\,get\,odd\,k$

Thus we can find $\sum_{k = 1}^{k <= m}$ $m \choose k$ for all odd values of k, which comes out to $2^{m - 1}$. Furthermore, we can choose the remaining elements in the subset in $2^{n - m}$ ways by similar logic. Hence total ways to get odd values of k are $2^{n - 1}$, which is independent of both m and k.
Hence all we need to do is find bits which are set atleast once (by computing OR) and then multiply the final answer with $2^{n - 1}$. Time Complexity of the program is $O(n)$ with $O(1)$ Space Complexity.

- C++

```
class Solution {
public:
int subsetXORSum(vector<int>& nums) {
int arrayOR = 0;
// do OR of whole array to obtain bits which are set atleast once
for (int num : nums) arrayOR |= num;
// compute the final answer using the formula discussed
return arrayOR * (1 << (nums.size() - 1));
}
};
```

Example #3: 0062 - Unique Paths

Here our robot always goes either down or right. We know that we have to go down $m - 1$ times and go left $n - 1$ times. Thus we need to find the number of ways to arrange these. One way to visualize this is if we have $m + n - 2$ blank spaces, and we have to fill $n - 1$ of them using R (representing going right) and remaining using D (representing going down). Then we can just choose the number of spaces to fill with L from total number of spaces. The the final solution is simply $m + n - 2 \choose n - 1$.

Notice that we are not required to return the value after taking modulo and the constraints allow for a $O(n^2)$ precomputation. Thus, we will simply construct the entire Pascal's Triangle and query it everytime to calculate the answer.

- C++

```
class Solution {
public:
int uniquePaths(int m, int n) {
// the upper limit is m + n - 2 = 198
vector<vector<long long int>> PascalTriangle(199, vector<long long int> ());
PascalTriangle[0] = {1};
// calculating every row of the triangle
for (int i = 1; i <= 198; i++) {
PascalTriangle[i].push_back(1);
// using the recurrence relation.
for (int j = 1; j < i; j++) {
// take mod with INT_MAX as otherwise some values may overflow.
PascalTriangle[i].push_back((PascalTriangle[i - 1][j] + PascalTriangle[i - 1][j - 1]) %
INT_MAX);
}
PascalTriangle[i].push_back(1);
}
// query the final answer
return PascalTriangle[m + n - 2][n - 1];
}
};
```

NOTE: Since every testcase only asks us to find $n \choose r$ for particular values of n and r, we can instead of precomputing the entire Pascal's Triangle, just compute the paricular value of $n \choose r$ using the recurrence relation and memoization. This will lead to less time and space complexity, as we only calculate the values we need. Also,

then we no longer need to take modulo with INT_MAX as all the values will fit in the "int" type as mentioned in the question.
You can check the complete solution for this problem here

Example #4: 2400 - Number of Ways to Reach a Position After Exactly k Steps

Let's represent going left as -1 and going right as $+1$. Thus, following the same idea as before, we have k blanks to fill with $+1$ and -1 such that there sum is equal to $endPos - startPos$.
Here we can immediately see that such will be impossible in only 2 cases:

- The parity of k and $endPos - startPos$ is different.
- The magnitude of k is less than magnitude of $endPos - startPos$.

After checking for above 2 cases, we know for sure that there exists a solution. Now we can just find the number of $1's$ and $-1's$ required to sum to $endPos - startPos$.
Expressing this as an equation:
$(1) *a* + *(-1)* b = endPos - startPos$, such that $a + b = k$
Here a represents the number of 1, i.e., the right steps and similarly b represents number of -1, i.e., the number of left steps. We are now interested in finding the number of possible values of a and b such that the above equations are satisfied.
Adding both equations,
$2a$ $=$ $endPos - startPos + k$
Thus,
a $=$ $\frac{k \, + \, endPos \, - \, startPos}{2}$
Similarly, by subtracting the equations and simplifying,
b $=$ $\frac{k \, - \, endPos \, + \, startPos}{2}$

Then the solution is $k \choose a$ $=$ $k \choose b$ as we need to find number of ways to choose a or b moves, out of k moves.
Thus, we need to find $nCr$$(k, \frac{k - endPos + startPos}{2})$.
To implement this, you can both precompute the entire Pascal's Triangle, or use concept of mudular inverses to find the required value.
You can check the complete solution for this problem here

Suggested Problems

Problem Name	Difficulty	Solution Link
920 - Number of Music Playlists	Hard	N/A
1916 - Count Ways to Build Rooms in an Ant Colony	Hard	View Solutions
1467 - Probability of a Two Boxes Having The Same Number of Distinct Balls	Hard	N/A

References

1. cp-algorithms

String

Palindrome

Author: @wingkwong

Overview

A palindrome reads the same forward and backward. amanaplanacanalpanamaand 10101are the examples of palindrome.
There are multiple ways to check if a string is a palindrome or not.

Iteration

As we know it reads the same forward and backward, which means $s[0]$ is same as $s[n - 1]$, $s[1]$ is same as $s[n - 2]$ and so on. Therefore, we can iterate $n / 2$ times to check if the left side is same as the right side.

- C++

Written by @wingkwong

```
bool isPalindrome(const string& s) {
for (int i = 0; i < s.size() / 2; i++) {
```

```
if (s[i] != s[s.size() - i - 1]) {
return false;
}
}
return true;
}
```

Built-in Functions

We can directly use built-in function to reverse a string and check if it is same as the target one.

- C++

Written by @wingkwong

```
bool isPalindrome(const string& s) {
string t = s;
reverse(t.begin(), t.end());
return s == t;
}
```

- C++

Written by @wingkwong

```
bool isPalindrome(const string& s) {
return s == string(s.rbegin(), s.rend());
}
```

- C++

Written by @wingkwong

```
bool isPalindrome(const string &s) {
```

```
return equal(s.begin(), s.begin() + s.size() / 2, s.rbegin());
}
```

Palindrome with Range

For a given range, we can follow the same idea to use two pointers to check if a sub-string is a palindrome .

- C++

Written by @wingkwong

```
bool palindromeWithRange(string s, int i, int j) {
while (i < j) {
if (s[i] != s[j]) return false;
i++, j--;
}
return true;
}
```