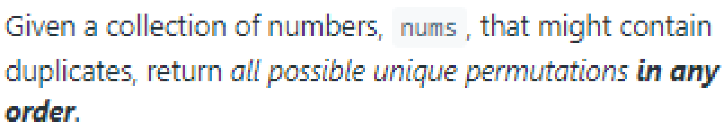
Leetcode

Thing’s I have learned from leetcode

▼ Leetcode 47 - Permutation II

Backtracking + Hash Map



Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations* ***in any order***.

Hash Map: Turn the list into the number of occurence of a list (I got this)

Neetcode

```
count = { n:0 for n in nums }
for n in nums:
    count[n]+=1
```

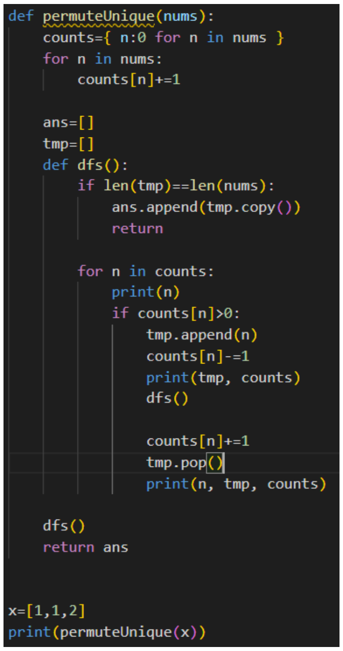
Me

```
elements=[]
while nums:
    s=nums[0]
    elements.append([s, nums.count(s)])
    while s in nums:
        nums.remove(s)
```

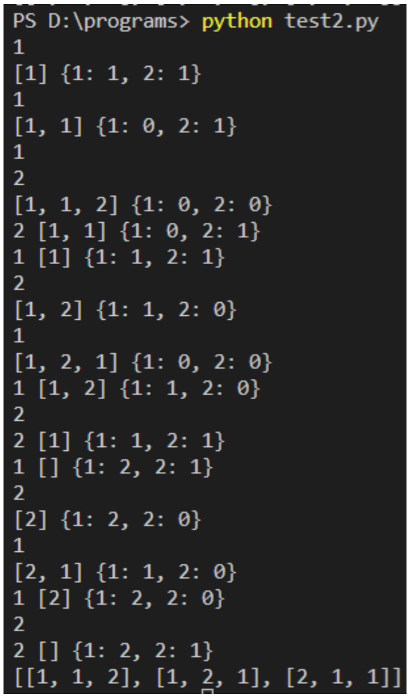
Backtracking: In order to not get duplicates. (Neetcode)

```
ans=[]
tmp=[]
def dfs():
    if len(tmp)==len(nums):
        ans.append(tmp.copy())
        return

    for n in counts:
        if counts[n]>0:
            tmp.append(n)
            counts[n]-=1
            dfs()

            counts[n]+=1
            tmp.pop()

dfs()
return ans
```

Explain:

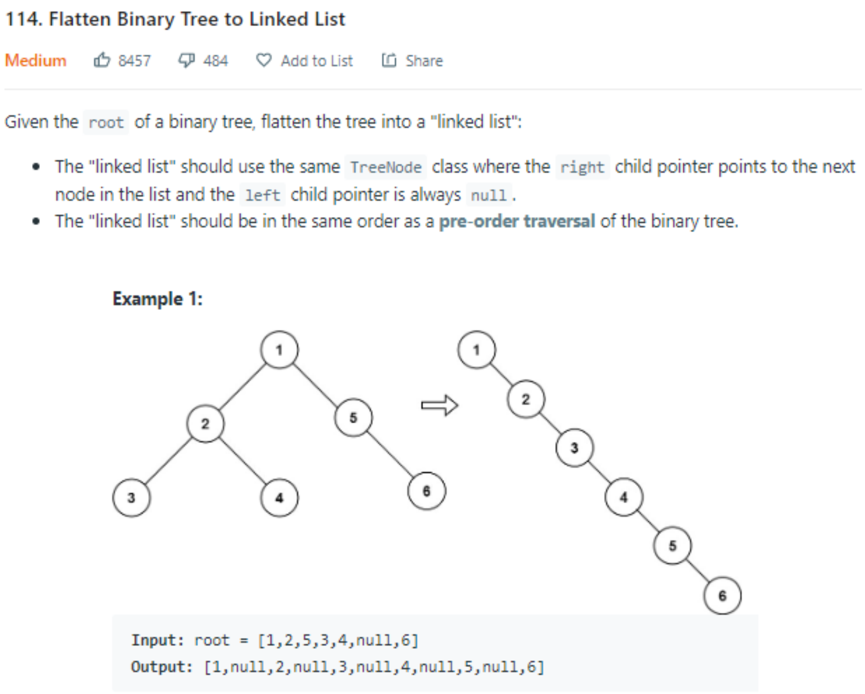


```
def permuteUnique(nums):
    counts={ n:0 for n in nums }
    for n in nums:
        counts[n]+=1

    ans=[]
    tmp=[]
    def dfs():
        if len(tmp)==len(nums):
            ans.append(tmp.copy())
            return

        for n in counts:
            print(n)
            if counts[n]>0:
                tmp.append(n)
                counts[n]-=1
                print(tmp, counts)
                dfs()

                counts[n]+=1
                tmp.pop()
                print(n, tmp, counts)

    dfs()
    return ans


x=[1,1,2]
print(permuteUnique(x))
```



```
PS D:\programs> python test2.py
1
[1] {1: 1, 2: 1}
1
[1, 1] {1: 0, 2: 1}
1
2
[1, 1, 2] {1: 0, 2: 0}
2 [1, 1] {1: 0, 2: 1}
1 [1] {1: 1, 2: 1}
2
[1, 2] {1: 1, 2: 0}
1
[1, 2, 1] {1: 0, 2: 0}
1 [1, 2] {1: 1, 2: 0}
2
2 [1] {1: 1, 2: 1}
1 [] {1: 2, 2: 1}
2
[2] {1: 2, 2: 0}
1
[2, 1] {1: 1, 2: 0}
1 [2] {1: 2, 2: 0}
2
2 [] {1: 2, 2: 1}
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

▼ Leetcode 114 - Flatten Binary Tree to Linked List



114. Flatten Binary Tree to Linked List

Medium 8457 484 Add to List Share

Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a **pre-order traversal** of the binary tree.

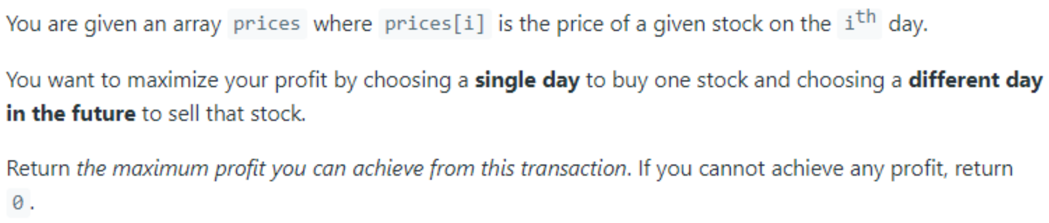
Example 1:

```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

Not understood

```
def dfs(root):
    if not root:
        return None
    leftTail = dfs(root.left)
    rightTail = dfs(root.right)

    if root.left:
        leftTail.right = root.right
        root.right = root.left
        root.right = None

    last = rightTail or leftTail or root
    return last

dfs(root)
```

▼ Leetcode 121 - Best Time to Buy and Sell Stock



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

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return `0`.

Best solution: Use two pointers to prevent negative profit. Use a left pointer to be the day to buy (min price), and use a right pointer to be the day to sell (max price).

Both of mine and neetcode’s code has memory O(1) and time O(n) complexity.

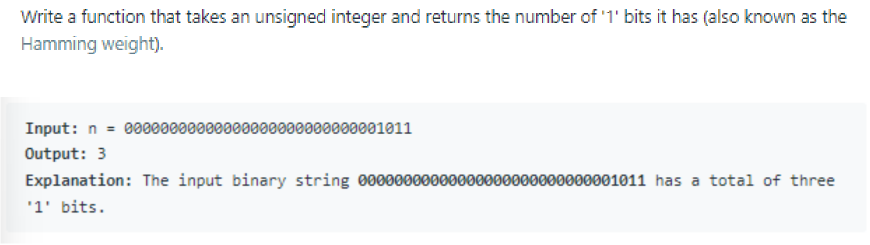
Mine

```
tmp = prices[0]
ans = 0
for x in prices[1::]:
    if x<tmp:
        tmp=x
    else:
        if x-tmp>ans:
            ans = x-tmp
return ans
```

Neetcode

```
res = 0
l = 0
for r in range(1,len(prices)):
    #if the price ahead is lowwer than the price behind
    if prices[r]<prices[l]:
        l = r
    res = max(res, prices[r]-prices[l])
return res
```

▼ Leetcode 191 - Number of 1 Bits



Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

```
Input: n = 00000000000000000000000000001011
Output: 3
Explanation: The input binary string 00000000000000000000000000001011 has a total of three
'1' bits.
```

We can solve this problem by:

Preseudocode from neetcode

First: Check the last number of the bit if it’s 1 or not.

Second: We shift the remaining bits to the right. (This is how a CPU works)

After seeing the Preseudocode

Neetcode O(1) / O(1)

```
sum=0
while n:
    sum+=n%2
    n=n>>1
return sum
```

Me

```
sum=0
while n!=0:
    if n%2==1:
        sum+=1
    n=int(n/2)
return sum
```

What I wrote before O(2n)

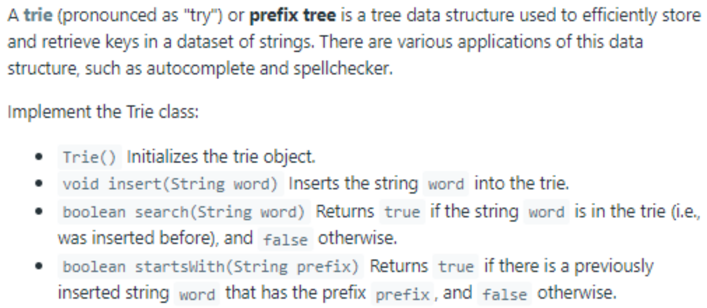
```
sum=0
if n%2:
    sum+=1
    n-=1
tmp=2
while tmp<=n:
    tmp*=2
tmp/=2
while n!=0:
    if tmp<=n:
        n-=tmp
        sum+=1
    tmp/=2
return sum
```

A different way: Let n = n & n-1(We can count how many 1's are there, instead of going through the whole n.)

When we are doing n-1, we are actually subtracting 1 bit from n, then when we n & n-1 together, we are actually removing the first bit, and the remainings stayed the same

```
sum=0
while n:
    sum+=1
    n = n & (n-1)
return sum
```

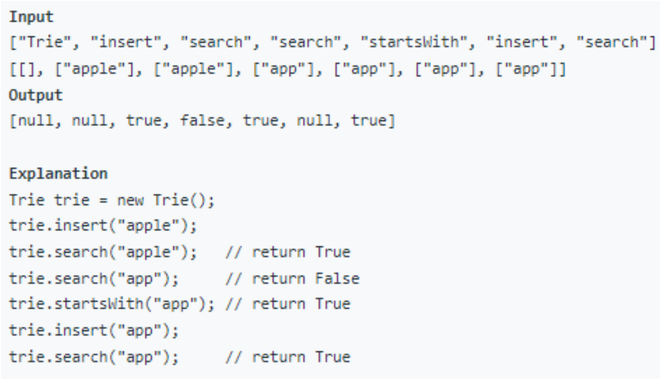
Leetcode 208 - Implement Trie (Prefix Tree)



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:

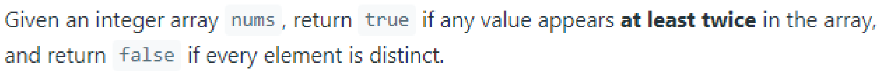
- `Trie()` Initializes the trie object.
- `void insert(String word)` Inserts the string `word` into the trie.
- `boolean search(String word)` Returns `true` if the string `word` is in the trie (i.e., was inserted before), and `false` otherwise.
- `boolean startsWith(String prefix)` Returns `true` if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



```
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.search("app");     // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");     // return True
```

```
class TrieNode:
    def __init__(self):
        #set the children of the word as a hashmap
        self.children={}
        self.endOfword=False

class Trie:
    def __init__(self):
        self.root=TrieNode()
    def insert(self, word):
        cur=self.root
        for x in word:
            #if x is not in the children of the hashmap add it to the hashmap
            if x not in cur.children:
                cur.children[x]=TrieNode()
            #change the link towards the current hashmap to continue
            cur=cur.children[x]
        #mark the end of word
        cur.endOfword=True
    def search(self, word):
        cur=self.root
        for x in word:
            if x not in cur.children:
                return False
            cur=cur.children[x]
        #the end of the word might not be the end of a word
        return cur.endOfword
    def startsWith(self, prefix):
        cur=self.root
        for x in prefix:
            if x not in cur.children:
                return False
            cur=cur.children[x]
        return True
```

Leetcode 217 - Contains Duplicates



Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

	Brutal Way	Sorted	Hashset
Time Complexity	O(n^2)	O(nlogn)	O(n)
Mem Complexity	O(1)	O(1)	O(n)

The brutal way is to check every element in the array

Sorted is to sort the array first, then check

Hashset is to create a hashset, then check

Mine

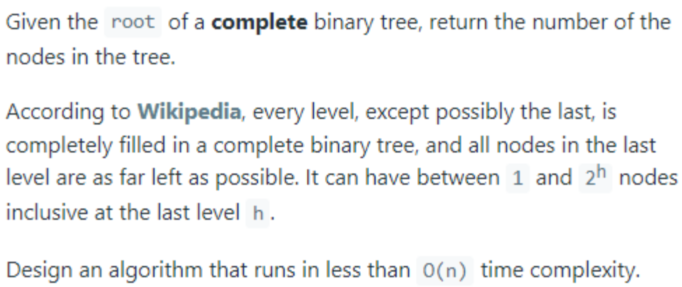
```
checkset = set()
for x in range(len(nums)):
    checkset.add(nums[x])
    if len(checkset)!=(x+1):
        return True
return False
```

Neetcode

```
hashset = set()
for n in nums:
    if n in hashset:
        return True
    hashset.add(n)
return False
```

Leetcode 222 - Count Complete Tree Nodes

Recursive



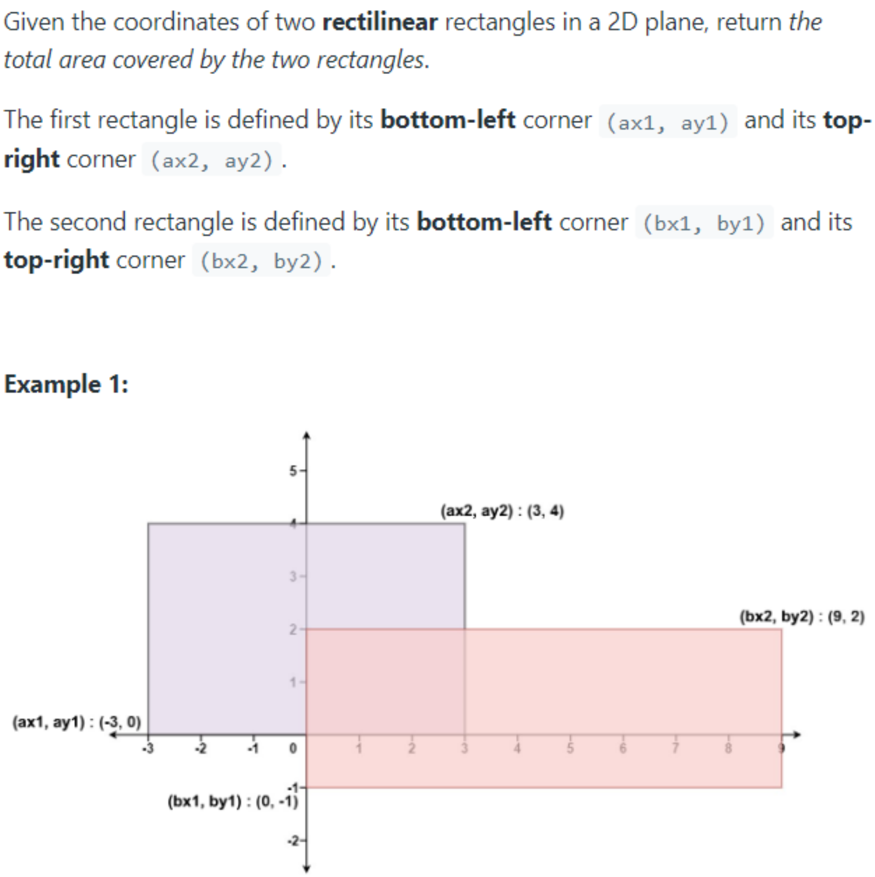
Given the `root` of a **complete** binary tree, return the number of the nodes in the tree.

According to **Wikipedia**, every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between `1` and `2`^{h} nodes inclusive at the last level `h`.

Design an algorithm that runs in less than `O(n)` time complexity.

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
ans = 0

class Solution:
    def countNodes(self, root: Optional[TreeNode]) -> int:
        global ans
        ans = 0
        if root==None:
            return 0
        recursiveCount(root)
        return ans

def recursiveCount(root):
    global ans
    ans += 1
    if root==None:
        return;

    if root.left != None:
        recursiveCount(root.left)

    if root.right != None:
        recursiveCount(root.right)

    return
```

Leetcode 223 - Rectangle Area



Given the coordinates of two **rectilinear** rectangles in a 2D plane, return *the total area covered by the two rectangles.*

The first rectangle is defined by its **bottom-left** corner `(ax1, ay1)` and its **top-right** corner `(ax2, ay2)`.

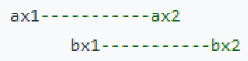
The second rectangle is defined by its **bottom-left** corner `(bx1, by1)` and its **top-right** corner `(bx2, by2)`.

Example 1:

1. Calculate the total area covered by the two rectangles
2. Calculate the area of the overlap between the two rectangles
3. Substract the overlap area from the total area

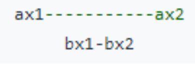
Overlapped area have been seperated into 3 cases.

case. 1



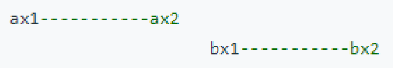
```
ax1-----------ax2
       bx1-----------bx2
```

case. 2



```
ax1-----------ax2
      bx1-bx2
```

case. 3



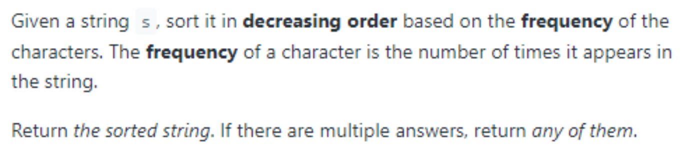
```
ax1-----------ax2
                        bx1-----------bx2
```

So, we only need to find the smallest end point and the greatest start point. If the smallest end point - the greatest start point < 0, that means it is case 3, no overlap occured. The rest of the case only indicates that there is overlap.

```
class Solution:
    def computeArea(self, ax1:int, ay1:int, ax2:int, ay2:int,
                    bx1:int, by1:int, bx2:int, by2:int) -> int:
        #rectangleA area + rectanleB Area - overlap Area
        return (ax2-ax1)*(ay2-ay1)+(bx2-bx1)*(by2-by1)
                -max(min(ax2,bx2)-max(ax1,bx1),0)*max(min(ay2,by2)-max(ay1,by1),0)
```

▼ Leetcode 304 - Range Sum Query 2D (Immutable)

▼ Leetcode 451 - Sort Characters By Frequency



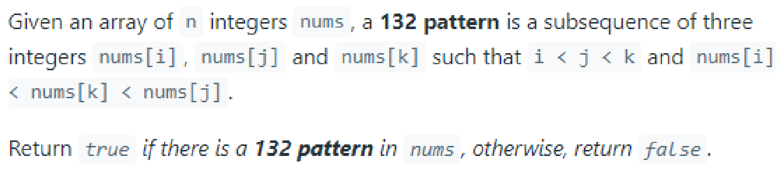
Given a string `s`, sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

```
dict_s = {}
# counts the frequency of the elements, can be simplified into Counters.mostCommon
for x in s:
    if x not in dict_s:
        dict_s[x] = 1
    else:
        dict_s[x] += 1

ans = ""
max = ""
max_num = 0
for x in range(len(dict_s)):
    for y in dict_s:
        if dict_s[y]>max_num:
            max = y
            max_num = dict_s[y]
    for y in range(max_num):
        ans += max
    del dict_s[max]
    max_num = 0

return ans
```

▼ Leetcode 456 - 132 Patterns

Decrease Stack (monotonically decreasing order)



Given an array of `n` integers `nums`, a **132 pattern** is a subsequence of three integers `nums[i]`, `nums[j]` and `nums[k]` such that `i < j < k` and `nums[i] < nums[k] < nums[j]`.

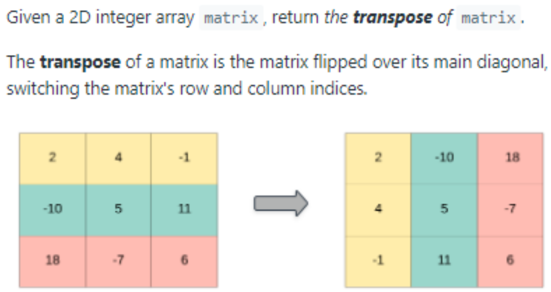
Return `true` *if there is a* ***132 pattern*** *in* `nums`*, otherwise, return* `false`.

Ex: [3,1,4,2]

n	3	1	4	2
min	3	3	1	1

```
stack = [] #pair [num, min], mono decreasing
curMin = nums[0]

for n in nums[1:]:
    while stack and n>=stack[-1][0]:
        stack.pop()
    if stack and n>stack[-1][1]:
        return True

    stack.append([n, curMin])
    curMin = min(curMin. n)
return False
```

▼ Leetcode 867 - Tramspose Matrix



Given a 2D integer array `matrix`, return *the* ***transpose*** *of* `matrix`.

The **transpose** of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.

2	4	-1
-10	5	11
18	-7	6

2	-10	18
4	5	-7
-1	11	6

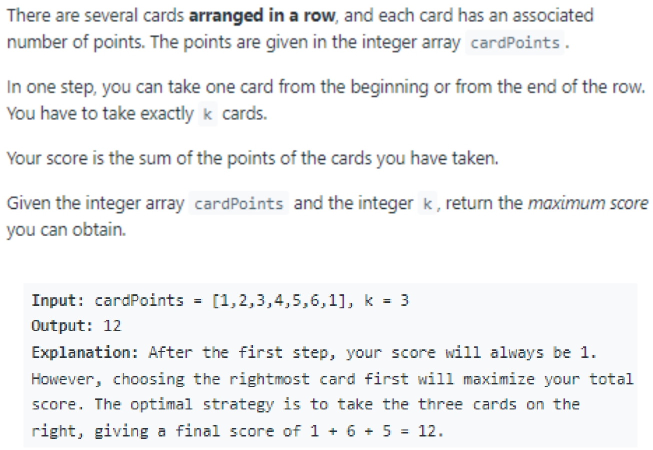
```
def transpose(self, matrix: List[List[int]]) -> List[List[int]]:
        ans=[]
        for x in range(len(matrix[0])):
            tmp=[]
            for y in range(len(matrix)):
                tmp.append(matrix[y][x])
            ans.append(tmp)
        return ans
```

```
def transpose(self, matrix: List[List[int]]) -> List[List[int
    return list(zip(*matrix))
```

“*” can turn matrix into 1 dimensional, such as turn [[1,2,3],[4,5,6]] ⇒ [1,2,3],[4,5,6]

zip() can pack lists into a matrix ⇒ zip([1,2,3],[4,5,6]) ⇒ [[1,4],[2,5],[3,6]]

Leetcode 1423 - Maximum Points You Can Obtain from Cards



There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array `cardPoints`.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly `k` cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array `cardPoints` and the integer `k`, return the *maximum score* you can obtain.

```
Input: cardPoints = [1,2,3,4,5,6,1], k = 3
Output: 12
Explanation: After the first step, your score will always be 1.
However, choosing the rightmost card first will maximize your total
score. The optimal strategy is to take the three cards on the
right, giving a final score of 1 + 6 + 5 = 12.
```

Think this question by moving the window to remove, rather than add each possibilities.

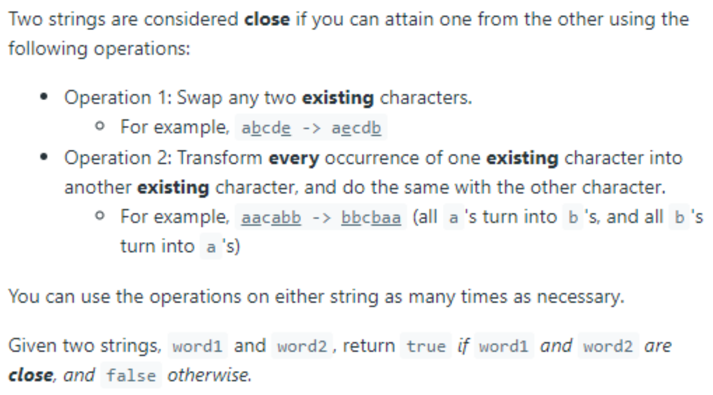
Mine

```
#limit time exceeded
if k==len(cardPoints):
    return sum(cardPoints)
ans=sum(cardPoints[0:k])
if sum(cardPoints[len(cardPoints)-k::])>ans:
    ans=sum(cardPoints[len(cardPoints)-k::])
tmp2=sum(cardPoints[0:k])
for x in range(1,k):
    tmp=sum(cardPoints[0:x])+sum(cardPoints[len(cardPoints)-k+x::])
    if tmp>ans:
        ans=tmp
return ans
```

Discussion

```
n=len(cardPoints)
#start the remains from 0~n-k-1
remain=sum(cardPoints[:n-k])
minRemain=remain
#shift the window from 0~n-k-1 to k~n-1
for x in range(n-k,n):
    remain+=cardPoints[x]
    remain-=cardPoints[x-n+k]
    minRemain=min(minRemain, remain)

return sum(cardPoints)-minRemain
```

▼ Leetcode 1657 - Determine if Two Strings Are Close



Two strings are considered **close** if you can attain one from the other using the following operations:

- Operation 1: Swap any two **existing** characters.
  - For example, `abcde -> aecdb`
- Operation 2: Transform **every** occurrence of one **existing** character into another **existing** character, and do the same with the other character.
  - For example, `aacabb -> bbcbaa` (all `a`'s turn into `b`'s, and all `b`'s turn into `a`'s)

You can use the operations on either string as many times as necessary.

Given two strings, `word1` and `word2`, return `true` *if* `word1` *and* `word2` *are* ***close****, and* `false` *otherwise.*

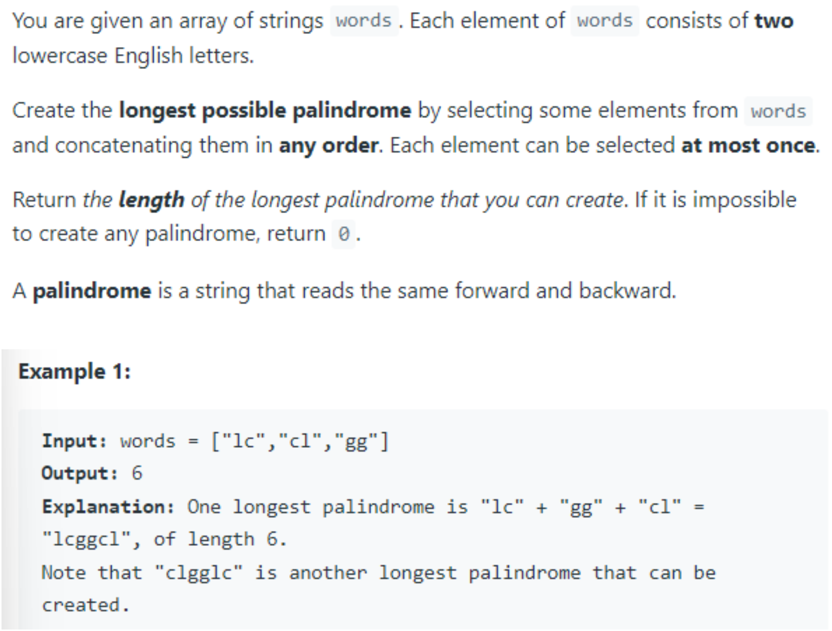
Thoughs: If the individual elements in each strings have the same amount, then the two strings can be the same through the two operations above.

```
# if the two strings don't have the same length, the two strings can't be similar
if len(word1)!=len(word2):
    return False

# create the dict to store the amount of elements in word1
dict_word1 = {}
for x in word1:
    if x not in dict_word1:
        dict_word1[x] = 1
    else:
        dict_word1[x] += 1

# create the dict to store the amount of elements in word2
dict_word2 = {}
for x in word2:
    if x not in dict_word2:
        # this checks the new element in word2, to make sure the new element is in word1
        if x not in dict_word1:
            return False
        dict_word2[x] = 1
    else:
        dict_word2[x] += 1

# find same appear times of the element and delete it
for x in dict_word1:
    found = False
    for y in dict_word2:
        if dict_word1[x]==dict_word2[y]:
            found = True
            del dict_word2[y]
            break
    if found == False:
        return False
return True
```

faster way

```
# collections.Counter counts the individual elements of word1
count_1 = collections.Counter(word1)
count_2 = collections.Counter(word2)
# num unique chars in both should be equal
if len(word1) != len(word2) or len(count_1) != len(count_2):
    return False
# set returns different elements in word
# collections.Counter(count_1.values()) returns the times of the individual elements appear, simillar to the last step of my code
return set(word1) == set(word2) and collections.Counter(count_1.values()) == collections.Counter(count_2.values())
```

▼ Leetcode 2131 - Longest Palindrome by Concatenating Two Letter Words



You are given an array of strings `words`. Each element of `words` consists of **two** lowercase English letters.

Create the **longest possible palindrome** by selecting some elements from `words` and concatenating them in **any order**. Each element can be selected **at most once**.

Return *the* ***length*** *of the longest palindrome that you can create*. If it is impossible to create any palindrome, return `0`.

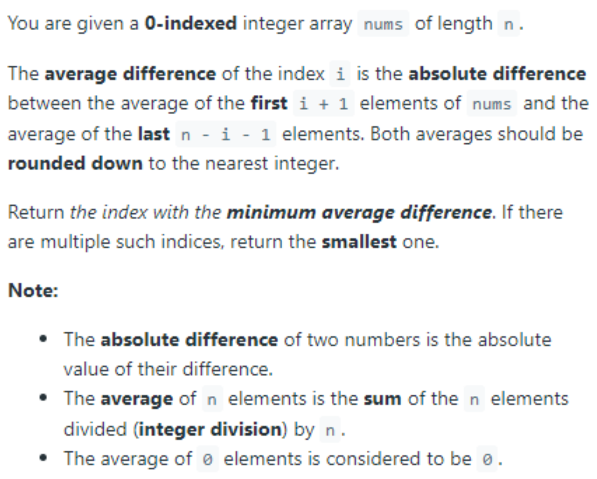
A **palindrome** is a string that reads the same forward and backward.

Example 1:

```
Input: words = ["lc","cl","gg"]
Output: 6
Explanation: One longest palindrome is "lc" + "gg" + "cl" =
"lcggcl", of length 6.
Note that "clgglc" is another longest palindrome that can be
created.
```

```
class Solution:
    def longestPalindrome(self, words: List[str]) -> int:
        ans = 0
        # create the map to store the times that a two letter
        # word appeared
        count = [[0]*26 for _ in range(26)]

        for a, b in words:
            # turn the two letters into the map
            i = ord(a)-ord('a')
            j = ord(b)-ord('a')

            # if ba is in the map ans+4 and remove the used element
            if count[j][i]:
                ans+=4
                count[j][i]-=1
            # if ba is not in the map add the times that ab has appeared
            else:
                count[i][j]+=1

        # last check whether there exist elements that is aa
        # if there is return ans+2
        for x in range(26):
            if count[x][x]:
                return ans+2
        return ans
```

▼ Leetcode 2256 - Minimum Average Difference



You are given a **0-indexed** integer array `nums` of length `n`.

The **average difference** of the index `i` is the **absolute difference** between the average of the **first** `i + 1` elements of `nums` and the average of the **last** `n - i - 1` elements. Both averages should be **rounded down** to the nearest integer.

Return *the index with the* ***minimum average difference***. If there are multiple such indices, return the **smallest** one.

Note:

- The **absolute difference** of two numbers is the absolute value of their difference.
- The **average** of `n` elements is the **sum** of the `n` elements divided (**integer division**) by `n`.
- The average of `0` elements is considered to be `0`.

```
# if nums = [0], the smallest will only be the first one
if len(nums)==1:
    return 0

# calculate the first range
total_back = sum(nums)-nums[0]
total_front = nums[0]

min_index = [0, abs(total_front-int(total_back/(len(nums)-1)))]
# calculate the rest of the range, but not the last index
for x in range(1, len(nums)-1):
    total_front += nums[x]
    total_back -= nums[x]
    if abs(int(total_front/(x+1))-int(total_back/(len(nums)-x-1))) < min_index[1]:
        min_index = [x, abs(int(total_front/(x+1))-int(total_back/(len(nums)-x-1)))]

# calculate the last index
total_back = 0
total_front = sum(nums)
if abs(total_front/len(nums)-total_back) < min_index[1]:
    return len(nums)-1

return min_index[0]
```

▼ Something to know

▼ 1.

In python, if we try to refrence a variable, which is not a object, it is going to assume it is a local variable for the function

▼ 2. shallow copy v.s. deep copy

In python, if we run the code bellow, we might find something weird

```
a=[[1,1],[2,2]]
b=a
a.append([3,3])
print(b)
```

You will find out, instead of getting [[1,1],[2,2]], we'll get [[1,1],[2,2],[3,3]]. That is we have linked a and b together. In order to not get this result, we can change b=a into b=copy.copy(a), this will avoid the event happened above.

However, if we run the code bellow, we might still find something weird

```
a=[[1,1],[2,2]]
b=copy.copy(a)
a[0][0]=0
print(b)
```

You will find out, instead of getting [[1,1],[2,2]], we’ll get [[0,1],[2,2]]. In order to prevent this from happening, we’ll have to use deepcopy()

```
a=[[1,1],[2,2]]
b=copy.deepcopy(a)
a[0][0]=0
print(b)
```

This time you will find out b=[[1,1],[2,2]], so by using deepcopy, we can totally seperate a and b

- 3. Data Structure
  - Array

    Values are stored in continueos spots.

Operation	Insert End	Delete End	Insert Mid	Delete Mid	Get to a specific node
Time Complexity	O(1)	O(1)	O(N)	O(N)	O(1)

  - Linked List (Pointers)

    Values are stored in different spots.

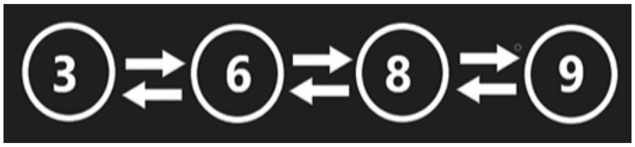
Operation	Insert End	Delete End	Insert Mid	Delete Mid	Get to a specific node
Time Complexity	O(1)	O(1)	O(1)	O(1)	O(N)
Explain	We’ll just have to add a extra path from the end to another node.	We’ll just have to point the node before the last node to null, to delete the last node.	We’ll just change the path and point the path to the node we want, then we can insert from the middle.	Simillar to insert mid. but instead of pointing it to a new node, we just point the node to the next node we deleted.	We’ll have to go pass the node one by one, to get to the node we want.

  - HashMap

    Values are stored in a arbitrary key (index). This is an unordered data structure, so it doesn’t have the concept of begin or end.

Operation	Insert	Delete	Search
Time Complexity	O(1)	O(1)	O(1)

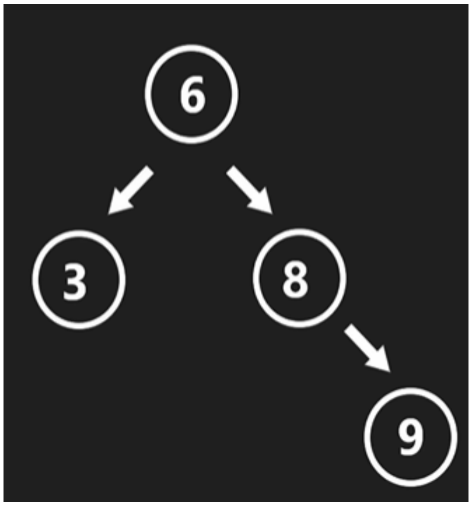
  - Queue



Between two nodes, we have two pointers.

Operation	Push Front	Pop Front	Push Back	Pop Back
Time Complexity	O(1)	O(1)	O(1)	O(1)

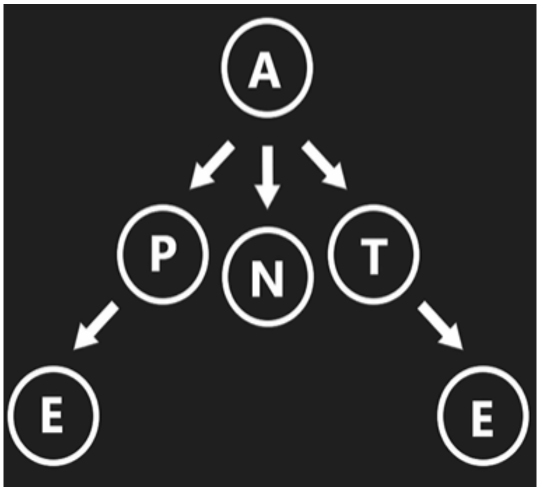
▼ Binary Tree (Tree Map)



The advantage of binary tree over Hashmap is the values are ordered, so that if you do a DFS over the tree, you'll get a ordered set.

Operation	Insert	Remove	Search
Time Complexity	O(logn)	O(logn)	O(logn)

▼ Trie / Prefix Tree



Each node represents a single character, and each node can have up to 26 childrens (26 alphabets). If we want to find all words which starts with A, we can do it by searching over the graph, and it is alse very convenient for auto type function.

Operation	Insert	Search
Time Complexity	O(n)	O(n)

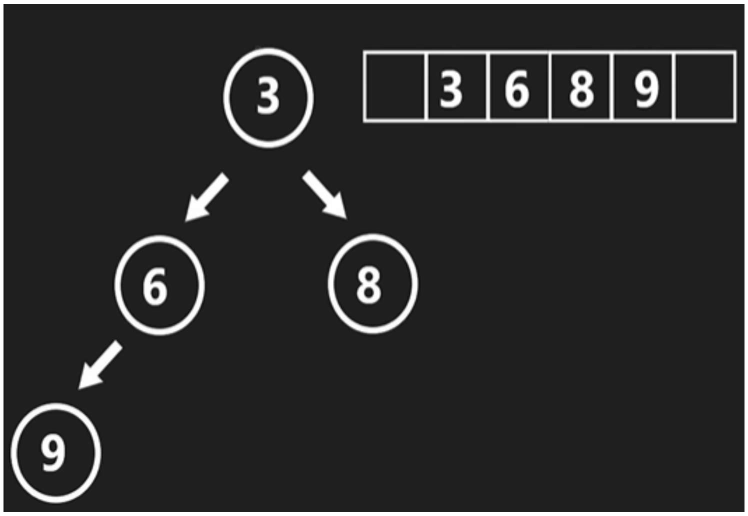
p.s. n is the length of the word

▼ Heap

Typically a heap will be a min heap or a max heap, for min heap the minimum value of the tree will be the root of the tree, and its children will always be greater, also the tree will be a complete tree, which means all levels will be full, instead of the last level.

Heaps are also associated with array. It is implemented this way, because we can get its left children by multiplying by 2 and get its right children by multiplying 2 and plus 1

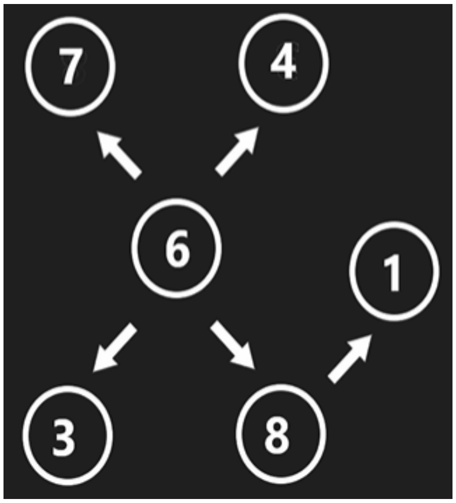
Ex: The below heap tree is associated with the array next to it. We can see that, the left children for 3 is 6, which its order in the array is [2] = [1]*2. And the right children for 3 is 8, which its order in the array is [3] = [1]*2+1. All elements in the tree follows this rule.

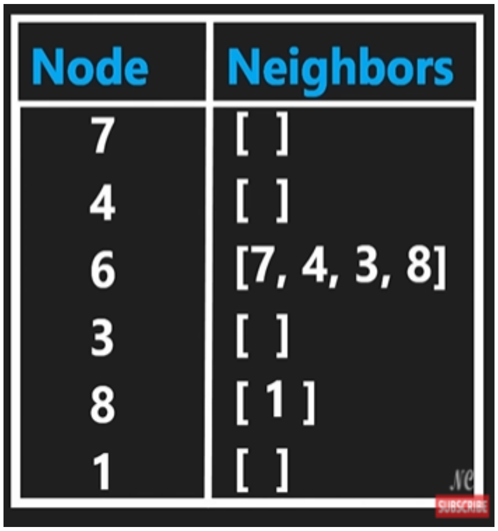


Operation	Insert	Pop	Min / Max
Time Complexity	O(logn)	O(logn)	O(1)

▼ Graphs

Some of the hardest data type to work with. All the above, instead of array are also called graphs, but they are resricted graphs, which has certain laws to follow. Since it is complicated (due to it can have arbitrary numbers of neighbors), it is more easy to represent it by a json list.





Node	Neighbors
7	[ ]
4	[ ]
6	[7, 4, 3, 8]
3	[ ]
8	[1]
1	[ ]

▼ 4. (&, |) v.s. (and, or)

and, or compares the value of the variable

&, | compares the bit of the variable

a and b: Returns 0 if a or b equals 0, if neither a nor b equals 0, it returns the last value

a or b: Returns 0 if a and b equals 0, if a or b has a value, it returns the first value that isn’t 0

▼ List Comprehension

▼ self

__init__ first variable will always be self, after that you can connect each thing to self. After using init, you can’t import empty variables.

```
class Example(object):
    def __init__(self, name, age, height):
        self.name = name
        self.age = age
        self.__height = height
    def print_score(self):
        print "%s: %s" % (self.name, self.age)
    def get_height(self):
        return self.__height
'''
Henry=Example("Henry", 22, 167)
print(Henry.name)
print(Henry.age)
print(Henry.__height)
Henry.get_height()
Henry.print_score()

output:
Henry
22
error
167
Henry:22
```

▼ Set

Sets will only contain one of each elements (no duplicates)

Create a set by: NAME=set()

Remove an element in the set by: NAME.remove(?)

Add an element in the set by: NAME.add(?)

▼ heapq

The heapq module is heap queue algorithm aka. priority queue. Heap queue is a binary tree, which its parents are always less or equal to its children’s value. Heap has an intersting property, the element which has the smallest value is always at the root.

Functions for heapq

1. heapq.heappush(heap, item)

   Put the item into heap, and keep heap’s properties unchanged

2. heapq.heappop(heap)

   Return the smallest value in the heap and take it out the heap, keep the properties unchanged at the same time. If we just want the smallest value of the heap, we can use heap[0]. (If the heap is empty, it will generate IndexError)

3. heapq.heappushpop(heap, item)

   Put the item into the heap. then return the smallest value of the heap. This will consider item also.

4. heapq.heapify(x)

   Turn list x into heap, during the process it won't need eccess memory.

5. heapq.heapreplace(heap, item)

   Return the smallest value of the heap, then put item into the heap. (This won't consider item)

6. heapq.merge(*iterables, key=None, reverse=False)

   Merge multiple sorted iterables. The variable keys, decide which to compare.

7. heapq.nlargest(n, iterable, key=None)

   Return the n largest value of the iterable. ⇒ It will generate a n elements iterable.

8. heapq.nsmallest(n, iterable, key=None

   Return the n smallest value of the iterable. ⇒ It will generate a n elements iterable.

▼ python: join

```
# words = ['bb','a','c']
ans = ''.join(words)
# ans = 'bbac'
```

▼ python: Counter

```
from collections import Counter
```

1 1