**GOOGLE** Onsite Prep

# Topics

## Binary Search

* ~~Lowest index search~~
* ~~Highest index search~~
* ~~Search a peak~~
* ~~Search array of booleans~~
* ~~Search rotated sorted array~~
* Variations
  + ~~Find a ^ (1/n)~~

## BFS/DFS

* BFS
  + ~~Multi-source BFS~~
  + ~~BFS lowest distance~~
  + ~~BFS on a tree / graph~~
* DFS
  + ~~Reachability~~
  + ~~DFS on a tree / graph~~
  + ~~DFS connected components~~
  + ~~Cycle detection~~
  + ~~Schedule~~
  + ~~Topological sort~~
  + Articulation Points

## Arrays / Lists

* + ~~Dutch national flag problem , sort an array of 3 types of elements~~
    - ~~Binary 0/1~~
    - ~~3 colors~~
    - 4 colors ?
  + ~~Quicksort Pivot~~
  + Ways to partition
    - <= , > (Into less than or equal to pivot , greater than pivot)
    - <, = , > (Into less than, equal to pivot, greater than pivot)
  + ~~Sorting colors in place~~
  + ~~Two pointers on sorted array~~
    - 2sum / 3 sum / 4sum
  + ~~Sorted array problems~~
  + ~~Random sampling~~
  + ~~Rotate array~~
  + ~~Zigzag~~
  + ~~Spiral~~

## 

## Hash Tables

* + Some common problems leetcode / Interviewbit

## Graphs

* ~~BFS~~
  + ~~Multi source BFS~~
  + ~~BFS on grids w/wo obstacles~~
* ~~DFS~~
  + ~~Reachability~~
  + ~~Cycle Detection~~
  + ~~DFS + DP (Finding shortest/ longest path based on a graph that is created on some prerequisite)~~
    - ~~Longest string chain transformation~~
  + ~~Connected components ~> double DFS problems~~
* ~~Dijkstra~~
  + ~~Basic~~
  + ~~Board game state reachability~~
* ~~Bellman Ford~~
  + Detect Arbitrage
  + ~~Negative cycle detection~~
* Floyd Warshall

## Linked Lists

* ~~Construction~~
* ~~Cycle Detection~~
  + ~~Simple~~
  + ~~Overlapping lists~~
* ~~Kth from last~~
* ~~Reverse~~
* Reverse in groups of k
* ~~Reverse a sublist~~
* Remove duplicates
  + ~~Completely remove dups in a sorted list~~
  + ~~Make frequency of all elements min(f(element), 0)~~
* ~~Merge sorted lists~~
* ~~DLL Construction~~
* ~~CLL Implementation~~

## Trees

* ~~Left view / top view / right view~~
* ~~Find LCA in Binary tree~~
  + ~~LCA in Binary Tree~~
  + ~~LCA in n-ary tree~~
* ~~Next pointers populate~~
* ~~Find if BT is BST~~
* ~~Construction using preorder and inorder~~
* ~~BT Construction~~
  + ~~Insertion~~
  + ~~Deletion~~
  + For a balanced BT
* Some general problems involving subtree sizes.
* ~~Flattening a tree~~
* ~~Convert a BT to DLL~~

## DP (Using the DP card on Leetcode)

* ~~Knapsack~~
* ~~Coin change~~
  + ~~Minimum no of coins~~
  + ~~Can a change be made with given denominations ?~~
  + Constrained?
    - Number of coins is limited
* ~~LCS~~
  + ~~Print LCS~~
    - ~~# of LCS~~
  + O(nlogn) way
* ~~Min difficulty of a job~~
* ~~Counting problem with LCS~~
* ~~Max height of stack of boxes that can be made, if lower box can be placed only on bigger box~~
* K profit loss transactions
  + ~~K is 1~~
  + ~~K is 2~~
  + K < N
* ~~House Robber~~
* ~~Multiply transactions from both ends on an array, find max cost~~
  + ~~When all transactions have cost 1 ~> O(K) check~~
  + ~~When transactions have different costs f(start, end) ~> max(f(start + 1, end) + T[start], f(start, end - 1) + T[end])~~
* ~~DP with 2 ends on array~~
* ~~String cuts / Min string cuts~~
* ~~Staircase, number of ways to reach using upto k steps in a jump~~
* No of ways to paint with k colors
* Rod cutting

## Queues / Stacks

* Parsing expressions
  + ~~Reverse Polish Notation~~
  + ~~Polish Notation~~
* ~~Max Queue API~~
  + ~~Using Deque~~
    - ~~Max~~
    - ~~Min~~
* ~~Many variations can be solved with next greater / next lesser template~~
* ~~Queue using 2 stacks~~
* ~~Min Stack~~

## Heaps

* ~~Construction~~
* ~~Heap Sort~~
* 2 Heaps
  + Running Median
  + ~~K closest~~

## Tries

* ~~Construction~~
  + ~~Seach~~
  + ~~Insert~~
  + ~~Keeping track of # endings at a point~~
  + Removing a string ? ~> decrease counts
* Some general problems involving tries
  + ~~String search in suffix~~

## Sorting

* Wiggle Sort
  + ~~Make alternate peaks and valleys~~
* ~~QS, RQS~~
* ~~MS~~
* ~~Heap Sort~~
  + ~~Heap construction~~
    - ~~Sift up~~
    - ~~Sift Down~~
    - ~~Heapify~~
* ~~Intervals~~
  + ~~Merging Intervals & its variant problems~~
  + ~~Meeting rooms~~
  + ~~When to sort by start point vs end point ?~~
* We sort by start times, because meetings are scheduled by start times. We can process one by one according to end times in a heap, as long as there are meetings that conflict (start < smallest end) they will conflict.

Very similar problems

* + Max number of people alive in a given year (given birth, death tuples)
* Merging intervals
  + Merging calendars of two people
    - Find free time given two calendars?
    - With k calendars?
    - Find free time for k people
  + ~~Can a person attend all meetings~~
  + ~~Minimum arrows to shoot balloons~~
* ~~Search in rotated sorted array~~

### Search in Sorted Rotated Array

* A rotated array sorted is always of form inc – dec – inc
* Use the modified BS
  + >= and <= for comparing elements with bounds
* Find Origin
  + Comparing with last element of the entire array always works
* Find origin, and do second BS

**Modified BS:**

class Solution:

def search(self, nums: List[int], target: int) -> int:

"""

H > h > h > h < l < l < l < l

1 2 3 4 5 6 7 8 9 10

8 9 10 1 2 3 4 5 6 7

5 6 7 8 9 10 1 2 3 4

mid ->

if left is sorted and nums is in there -> go left

elif nums is less than mid -> go left

else: go right

"""

low = 0

high = len(nums) - 1

while low <= high:

mid = (low + high) // 2

print("Searching in ", nums[low : high + 1])

if nums[mid] == target:

return mid

if nums[mid] >= nums[low]:

if target < nums[mid] and target >= nums[low]:

high = mid -1

else:

low = mid + 1

else:

if target <= nums[high] and target > nums[mid]:

low = mid + 1

else:

high = mid - 1

return -1

### Search in Sorted Rotated Array (With Dups)

* Here we do the same thing as above
* If mid is equal to target, return target
* If mid is greater than low # you are in the sorted part of array
  + If target falls in low to mid go left
  + Else go right
* Elif mid is less than low # you are not in sorted part of array
  + If target falls in mid to high -> go right
  + Else go to the left
* Else now mid is equal to low
  + Try increasing low += 1 until the end
* If you break out of the loop, return -1

### Find origin / index 0 position in sorted rotated array (No dups)

* If mid is greater than high, go to the right
* If mid is less than high, go to the left but include mid
  + High = mid
  + And termination is then low < high

low = 0

high = len(nums) - 1

while low < high:

mid = (low + high) // 2

if nums[mid] > nums[high]:

low = mid + 1

elif nums[mid] < nums[high]:

high = mid

return nums[low]

### Find origin / index 0 position in sorted rotated array (With dups)

* The same thing here as above
* If mid > high: go to the right
* If mid < high: [mid: high] is a sorted array, and mid could be a candidate for the minimum element
  + So we need to modify high as high = mid and NOT high = mid - 1
  + Because of this the break condition is low < high
* Else if they are equal, then try to do high -= 1

low = 0

high = len(nums) - 1

while low < high:

mid = (low + high) // 2

if nums[mid] > nums[high]:

low = mid + 1

elif nums[mid] < nums[high]:

high = mid

else:

high -= 1

return nums[low]

## 

## Recursion & Backtracking / + Memoization

* ~~Partition to K equal Sum subsets~~
* ~~N - queens~~
* ~~Word break~~

## Disjoint Sets

* ~~Construction~~
  + Simple construction and querying
  + ~~Path compression ?~~

## Segment Trees / Fenwick Trees

* ~~Construction~~
  + ~~R-max~~
  + ~~R-min~~
  + ~~R-sum~~
  + Lazy Propagation - not required
* Fenwick construction
  + Update / Range query

## Bit Masking

* ~~Elementary bit operations~~
* ~~Finding all subsets~~
* [~~Campus Bikes~~](https://leetcode.com/problems/campus-bikes-ii/) ~~~> Bitmasking + DP Not understood~~

# 

# Mock Assessments

## 3/15

Set 1 (1:30 hrs)

* <https://leetcode.com/problems/sentence-similarity/>
  + Solved ~5 minutes
* <https://leetcode.com/problems/sentence-similarity-ii/>
  + Follow up on first one, solved using DFS ~ 10 minutes
* <https://leetcode.com/problems/android-unlock-patterns/>
  + With backtracking
  + Spent more time on deciding how to validate a pattern
  + Code ready, but failed on some cases
* <https://leetcode.com/problems/design-search-autocomplete-system/>
  + No time left
* Score: 2.23/10

## 3/ 17

Set 2 (1hr)

* <https://leetcode.com/problems/number-of-good-pairs/>
  + Solved ~ 5 minutes
* <https://leetcode.com/problems/minimum-time-to-make-rope-colorful/>
  + Track total sum in consecutive elements and local maximum color
  + Solved ~ 7 minutes
* <https://leetcode.com/problems/count-unique-characters-of-all-substrings-of-a-given-string/>
  + Almost figured out optimal approach
* Score: 4.4 / 10

## 3/19

Set 3 (1hr)

* <https://leetcode.com/problems/peak-index-in-a-mountain-array/>
  + Solved using BS ~ 7 minutes
* <https://leetcode.com/problems/find-in-mountain-array/>
  + BS on left leg and on right leg ~ 10 minutes
* <https://leetcode.com/problems/delete-nodes-and-return-forest/>
  + Solved using DFS, when returning back, remove pointers and add root to forest
  + ~15 minutes
* Score: 3.78 / 10

## 3/21

Set 4 (1:15 hr)

* <https://leetcode.com/problems/single-row-keyboard/>
  + Easy ~ 5 minutes
* <https://leetcode.com/problems/maximum-level-sum-of-a-binary-tree/>
  + Level traversal ~ 5 minutes
* <https://leetcode.com/problems/odd-even-jump/>
  + Got a brute force approach only
    - Code did not pass
* Score 3.05 / 10

## 3/25

Set 5 (1:30 hr)

* <https://leetcode.com/problems/toeplitz-matrix/>
  + Solved ~ 6 minutes
* <https://leetcode.com/problems/bulls-and-cows/>
  + Use 2 hashsets, Solved ~ 10 minutes
* <https://leetcode.com/problems/evaluate-division/>
  + Construct a graph a ~> b if a divides b
  + Answer query using DFS
  + Solved ~15 minutes
* <https://leetcode.com/problems/minimum-cost-to-hire-k-workers/>
  + Could not solve
* Score: 4.09 / 10

## 3/27

* <https://leetcode.com/problems/alphabet-board-path/>
  + Graph problem, use BFS to get shortest path
  + Solved ~ 13 minutes
* <https://leetcode.com/problems/shortest-distance-to-target-color/>
  + Using a prefix and suffix combination, find the shortest score
  + Solved ~ 15 minutes
* <https://leetcode.com/problems/sliding-puzzle/>
  + Use dijkstra, game states are 1 unit apart after a swap
  + Find if destination can be reached
  + Solved ~20 minutes
* Score: 5.10 / 10

## 4/16 G onsite set

* Solved all 4
  + Easy
  + Medium, find if binary reps of all numbers from 1 to n is a substring of a given string S
    - Avoid O(n^3) to find all present numbers in S, calculate i …j + 1 val using i .. j result
    - Match in O(n ^ 2) while computing, subtract sum of first n numbers from every new number being added to the set
    - In the end size of set >= n and sum == 0
  + Knapsack variation, but can be solved using greedy
  + Bookshelf arrangement
    - Solved using DP, very nice problem .. similar to minimum job difficulty scheduling in d days.
    - Choice of starting on a new shelf, or continue on same shelf keep max height side by side
    - State requires only the current book index and remaining shelf width to fill

* Cherry Pickup
  + Needed to remove one extra dimension since row of both robots is going to be same
    - f(row, col1, col2) instead of f(row1, col1, row2, col2)

Problems

(Mostly Google tagged leetcode / Interviewbit)

## 3/19

* Range module
  + Add a range
    - BS and add the range N
  + Remove a range
    - Go through all intervals
    - If an interval overlaps completely - remove it
    - If it overlaps partially modify it
    - If it encloses removing range completely - split it in to two
    - If no intersection keep it
  + Query a range
    - BS
* Summarize as intervals a stream of numbers
  + Easy way out, add [v, v] everytime and sort and merge - NlgN
  + Better is to insert in O(n) ~> Very tricky
* Robot Room cleaner
  + Something like a DFS - keep track of directions where you came from

## 3/22

* First missing positive entry
* 1 < a + b + c < 2
* BS in peak array - BS on left and right and finding peak

## 3/24

* Remove duplicates from a sorted LinkedList, retain min(1, f(x)) for each x present
  + Simple, two pointers, outer gives a start, inner goes forward and points to the end of the last duplicate. Then we can point the outer next to where the inner stopped (if inner is not equal to outer, if there were no duplicates).
* Remove duplicates, retain no occurrence of duplicate. min(0, f(x)) if x is a duplicate.
  + A very good example of sentinel/ placeholder usage.
* Remove dups in unsorted, O(n) + O(n)
* Sort a linked list of 0s, 1s / Sort like QS partition
  + A better idea is always to create separate lists and then join them.

## 3/26

* Step by Step
  + Every step add i or -i, minimum steps to reach target
  + Add steps as long as less than or equal to target
  + 1 + 2 + …n >= target
  + If target -n is even , n is the answer, because the difference d means
  + We need a step reverse of d/2 in the entire sequence to reach the target, which we can always do since we have numbers from 1 to n added.
  + If the difference is odd, add n + 1, try n + 2 adding etc.
* AIO sub array sort, find minimum subarray that if sorted sorts the entire array
  + Find leftmost and rightmost out of order elements
  + Find leftmost and rightmost places where they need to be placed - that is the answer
* ZigZag traversal, zigzag traverse the array from top left
  + Directions are changed when row hits 0 or n -1 or when column hits 0 or m - 1
  + Separate out going up and going down
  + Going down you can hit column 0 or row n - 1, going up you can stop at row 0 or column m - 1
  + Based on above conditions, manipulate the row / col values
* Read: Max visible points ~~TODO not understood~~
* Gassup problem TODO O(n) not understood
  + Brute force, n^2, check starting from each city.
  + If the total sum of gas required is <= available gas, a trip is possible.
* Solve N queens
* Solve a sudoku
* String chain, find max string chain ~> Graph DFS
* Longest balanced substring
  + Solve it using merging intervals, every matching brace start end can be tracked, longest merged will be the answer.
  + A better approach is to keep a stack, initialize with -1, and push a new index onto the stack when a closing brace comes, take stack’s top and cur\_index - top is max length, if an unmatched closing brace comes pop the stack and put this on.
    - (())())

stack = [-1]

ans = 0

for i in range(len(string)):

x = string[i]

if x == '(':

stack.append(i)

else:

stack.pop()

if len(stack) == 0:

stack.append(i)

else:

ans = max(ans, i - stack[-1])

* + A surprisingly better approach can be done with just count of opening and closing brace. If closing > opening we instantly invalidate current counts to zero.

## 3/27

* Hard: Minimum number of squares required to fit in a rectangle
  + Naturally feels DP, L, W ~> MIN Over(1 + (L - x, W) + (W, x)) for x in [1, MIN(L, W)] if we were to place the square at the bottom, leaving two new smaller rectangles to fit.
* Coin Change, ..
* Hard: Number of ways to parenthesize a bool expression consisting of |&^ that evaluates to true or false
  + DP, count for left and right for true and false, left\_false, left\_true, right\_false, right\_true
  + If we want to find ways to true, if ^, then left\_false\*right\_true + left\_true\*right\_false
  + If expr is & then left\_true \* right\_true etc
  + Every odd character will be an operator, rest will be numbers.
* Longest stack of boxes that can be arranged, O(n^2) DP
  + Sort boxes, so that Bi, Bj : Bi comes before Bj has possibility Bi can be placed on Bj
  + cache[index] == 0 then compute, last return max of this included and with not including index
  + Return max(cache[index], f(index + 1, bottom)) bottom is the parent, on top of whom we need to put the current box.
* Generate valid parenthesis,
  + If we keep a count of open and closing parens we can use
    - When can we add a left paren ?
      * If we have left > 0 we can always insert a left paren
    - When can we add a right paren?
      * If right > left, means left parens are used more currently, so we can insert a right paren.
* Multiply without using \* or /
  + f(small, big) , if small is 1 then ans is 1\* big = big
  + f(1, 50) = 50, f(2, 50) = 100, f(4, 50) = 400 and so on
  + O(logs) recursion
* Towers of hanoi
  + Move top n -1 to 1 using buffer = 2
  + Move bottom to 3
  + Move n - 1 disks on 2 to 3 using buffer = 1
  + Base cases
    - N = 0 do nothing return
    - N = 1, simply move
* Find redundant braces
  + Pop in only operands and open braces
  + When ), if a ( is on immediate top , we have encountered a redundant bracket.
  + Otherwise remove everything in between this ) and its corresponding opening brace.

## 3/28

* Max area histogram / water area / skyline problems.
  + Water area ~> left max and right max O(n), O(n)
    - O(n), O(1) not understood
  + Skyline - not understood
  + Histogram ~> not understood
* Multi source BFS ~> Hotel service
* Completed all stacks and queues questions
  + Max Queue
  + Max area of histogram
    - Variant ~> Hollow pillars
  + First non repeating character in a stream using count + MAX Queue like structure
  + Normalize directory path names
  + Trapped rain water
    - Variant ~> Hollow pillars
  + Reverse polish notation - scan forwards, pop 2 operands push back res with one operator
    - Variant: Introduce brackets?

## 

## 3/29

* Min jumps to reach end
  + Intuitive O(n^2) dp
  + Unintuitive greedy solution choose next from max of from [i, i + A[i]]
* Mark X as Os that are surrounded by X in a grid
  + Mark as X all O’s that are not connected with Os on border
* Next permutation / Previous permutation
  + find longest decreasing suffix as its already decreasing so we cannot improve it
  + find an element max which is just greater than the decreasing sufix,
  + ex: 4, 2, 6, 5, 3, 1 => 4, 2, 1, 3, 5, 6
  + 2 breaks the decreasing suffix
  + now we can't swap 2 with a number lesser than 2 since that would take us
  + back lexicographically, but we want to go greater lexicographically
  + so we swap suffix breaker with an element just greater than 2
  + which is 3 in this case so now we have 4, 3, 6, 5, 2, 1
  + But now since we are ahead lexicographically due to introduction of 3 in place of 2
  + we can further go to a more less (but lexico greater seq) by reversing our decreasing
  + suffix, so we are thrown to a more lexico lesser sequence
  + so now we have 4, 3, 6, 5, 2, 1 -> 4, 3, 1, 2, 5, 6
* Find kth permutation
  + O(kn) ? next permutation k times
* Generate permutations in sorted order ?
  + O(n! . n) , append the next permutation until there is no next permutation.

## 3/30

* Longest palindrome ~> LCS of A and A.reverse()
  + N ^ 2 way to expand from each index as a center
* LCS such that index is different, A[i] = B[j] and i != j
  + Tricky base case
  + Careful on adding 1 + to the recursive call on matching characters, already covered in the base case?
* Count distinct subsequences
  + Did with recursion similar to LCS
  + O(n) space solution with keeping track of just the previous row.
* Minimum Edits needed to make both strings same
  + The final string will be the LCS
  + So answer is L(A) + L(B) - 2\*LCS
* Are strings interleaved ?
  + Similar to LCS, try matching a character from either of string
  + Base case: return 1 when all pointers cross 0
* Number of ways to fit a domino and triomino in a 2 X N grid
  + Wrong answer to go by Length X Width recursion
  + Generalize recursion by fitting 1 / 2 domino , 1 triomino
* Paint House - minimum paint required to paint houses in a row, with 3 colors. No two houses should have the same color.
  + Top down approach, find maximum by excluding one color and including other two.

## 3/31

* Combination sum
  + Recursion ~> f(remaining, index, partial) , use index or not use index
  + No duplicates
    - Keep a running counter of currently used elements in partial
* Hard: Find server that served most requests
  + Similar to keeping track of least request end time
  + O(n^2) because of searching next free server

# 4

## 4/1

* Find longest zigzag path length in a binary tree
  + If current node is left child return right + 1, else left + 1
  + If its root, return max both +1
  + Special case for only single node. ~> answer is 0
* All nodes at distance K
  + Find through DFS, the subtree which has given node, if found in left at distance x,
    - Explore right branch with x
  + Same for right branch, otherwise return -1
  + One func that explores nodes and adds nodes at a certain distance from given node find\_node(node, d), find all descendants of node that are at distance d from node
* Inorder successor of a node p in Binary Search Tree (with root ~> tree)
  + If it was a BT, the answer is leftmost node in right subtree
  + otherwise , the first parent in descendants such that we move from its left child
  + For BST, any value higher than p is a potential candidate.
  + If tree.val < p.val, we can discard tree.left, and we move to tree.right
  + If tree.val >= p we go to its left, but keep the potential candidate updated to tree
* Inorder successor of BST when parent pointer is present.(and you don't know the root)
  + Same as BT solution

## 4/2

* EPI Recursion
* TOH, need to involve P3 in all movements.
  + N = 1, if start, target has P3 move start to target
  + Else start - P3 - target
  + N = some K
  + Move K - 1 disks from start to other
  + If target, start in P3, move Kth disk to target, and K -1 to target
  + If target, start not in P3
    - move k - 1 disks to target
    - Move kth disk to P3
    - Move k - 1 disk to start
    - Move kth disk to target
    - Move k - 1 disk to target
  + Move k - 1
* TOH, must go P1-P2, P2-P3, P3-P1, cyclic moves allowed only
  + N = 1, if valid move start - target
  + If not valid move, start - int - int - target
  + N = K
    - Move k -1 disks from start to target (if valid)
      * If not valid, move k - 1 to - int - int - target
    - Move kth from start - target
      * If not valid, move k - 1 disks to other

## 4/3

* Merge elements
  + If not consecutive, push all elements into heap, take the top 2 and push back the sum until heap is of size 1
  + Very unintuitive, f(i, j) will be the sum from i …j plus resolving f(i, k), f(k, j) for all k in i…j
    - f(i, j) = min(f(i, j), sum(i, j) + f(i, k) + f(k, j)) V k E [i, j]
* Largest area of rectangles with column swaps allowed.
  + Compute top to down sum of consecutive ones, and then sort each row descending
    - So now the answer would be maximum of A[i][j] \* (j + 1)
    - 0 1 1

1 1 1

1 0 0

* 0 1 1 1 1 0

1 2 2 ~> 2 2 1 max is 2 \* (2) ~> 4 size of rectangle

2 0 0 2 2 1

* Find all solutions to N-queens
  + Backtracking, track queen positions in a column array, row[k] = c means (k, c) is a placement
  + How to check if a conflicting position IS in O(1) ?
* Edit distance classic
  + Similar to LCS, if we edit given state (i, j) we can go to (i - 1, j - 1) if chars are same
  + Or try editing
    - If we delete A[i], state is (i + 1, j) # Match B[j] with next char, A[i] is ignored since it is deleted
    - If we replace A[i], state is (i + 1, j + 1) # B[j] is matched, so is A[i]
    - If we insert a char at A[i] to match B[j], state is (i, j + 1) # B[j] is matched
  + f(i, j) ~>
    - f(i - 1, j - 1 (if A[i] and B[j] are same)
    - min (f(i, j + 1), f(i + 1, j + 1), f(i + 1, j))
  + Will be similar if all operations had a cost incurred of type of operation
  + Iterative ?
* Sequence of first k numbers that are a product of given primes, in increasing order
  + Keep index of next number that hasn't been multiplied by the prime no yet
  + When adding a number, if it is equal of any prime \* index[num[prime]], increment the index.

## 4/4

* Flip array, flip minimum elements( change to -ve) so that the sum of the array is as close to 0 as possible.
  + Knapsack variation, here value of each element is 1, and weights are the array
  + Find the knapsack for weight of S // 2, and then moving backwards from S // 2
  + Find first non zero f(sum, index) , that will be the answer.
* Tushar’s Birthday
  + When occurrences are unlimited, knapsack is not needed, something like coin change suits the problem.
    - f(amount) ~> min(f(amount), cost(X) + f(amount) - X) where is X is coin denominations/ values / capacities of the subject that needs to be minimized / maximised.
  + Compute the minimum cost for friend\_capacityMAX

def solve\_for\_capacity(capacity, dishes, cost):

dp = [float("inf") for \_ in range(capacity + 1)]

dp[0] = 0

for cap in range(1, capacity + 1):

for idx in range(len(dishes)):

dish\_cost, dist\_capacity = cost[idx], dishes[idx]

if dish\_capacity > cap:

continue

dp[cap] = min(dp[cap], dish\_cost + dp[cap - dish\_capacity])

return dp

to\_pay = 0

dp = solve\_for\_capacity(max(A), B, C)

for friend\_capacity in A:

to\_pay += dp[friend\_capacity]

return to\_pay

* N Bit gray code

"""

000 - 001 - 011 - 111 - 101 - 100 - 110

0 ~> 00, 10

000, 010, 110, 100 + reverse this sequence and append with 1

0000 0010 0110, 0100, 1100, 1110, 1010, 1000

"""

def find\_sequence(num\_bits):

if num\_bits == 0:

return [0]

sequence\_bits\_minus\_1 = find\_sequence(num\_bits - 1)

sequence = sequence\_bits\_minus\_1 + [2 \*\* (num\_bits - 1) + k for k in reversed(sequence\_bits\_minus\_1)]

return sequence

return find\_sequence(A)

* Number of unique BSTs of size n
  + f(i) = f(n - 1 - k) \* f(k) [n - k - 1 because 1 count is the root itself]

def findBST(size, cache) -> int:

if size in cache:

return cache[size]

if size < 0:

return 0

if size <= 0:

return 1

ans = 0

for k in range(size):

ans += findBST(size - k - 1, cache) \* findBST(k, cache)

cache[size] = ans

return ans

* Palindrome partition with minimum cuts
  + Similar to finding if a partition exists

def pal(i, j): # is palindrome

sl = A[i: j + 1]

return sl == sl[:: -1]

self.cache = {}

def min\_cuts(index):

if index >= len(A): # success, return cuts - 1

return -1

if index in self.cache:

return self.cache[index]

ans = float("inf")

for i in range(index, len(A)):

if pal(index, i):

ans = min(ans, 1 + min\_cuts(i + 1))

self.cache[index] = ans

return ans

return min\_cuts(0)

## 4/5

* Coins in a Line
* K Manhattan Distance
  + Recurrence , dp(i, j, k) ~> max(dp(nx, ny, k - 1)) (nx, ny) are neighbors
* Number of ways to Parenthesize an expression to get true / false
  + f(start, end, boolean) ~> num ways to pa^ize S{start:end} to make it evaluate to boolean
  + Base case is if string length is zero, return boolean
  + If String length is 1, return if it matches the required boolean
* Longest balanced parenthesis with dp
* Egg dropping
  + If eggs == 1, floors steps are required
  + Otherwise we can check from every floor between 1 to floors
    - If on dropping from Kth floor out of floors
      * If egg breaks, ~> f(eggs - 1, floors - K)
      * If it does not break ~> 1 + f(eggs, K - 1)

def find\_min\_trials(eggs, floors):

if floors <= 1:

return floors

if eggs == 1:

return floors

if (eggs, floors) in self.cache:

return self.cache[(eggs, floors)]

mn = float("inf")

for floor in range(1, floors + 1):

mn = min(mn, max(find\_min\_trials(eggs, floors - floor), find\_min\_trials(eggs - 1, floor - 1)))

self.cache[(eggs, floors)] = mn + 1

return self.cache[(eggs, floors)]

* Profits with k transactions
  + 1 transaction
  + Any number of transactions
  + K transactions
    - If at index = i, you can sell at i and buy at some index k < i
    - f(i, j) ~> max(f(i, j), A[i] - A[k] + f(k - 1, j - 1)) V k E [0, i - 1]

## 4/6

* Minimum subset sum difference

Knapsack template for storing the actual selection with minimum/maximum property.

In this case also find the indices which the knapsack algorithm chooses along with the sum value.

def solve(self, A):

S = sum(A)

INF = float("inf")

dp = [[INF for \_ in range(len(A))] for \_ in range(S // 2 + 1)]

indices = [[[] for \_ in range(len(A))] for \_ in range(S // 2 + 1)]

dp[0] = [0] \* (S // 2 + 1)

for w in range(1, S // 2 + 1):

for j in range(1, len(A)):

dp[w][j] = dp[w][j - 1]

indices[w][j] = indices[w][j - 1]

if A[j] > w:

continue

if dp[w][j] > A[j] + dp[w - A[j]][j - 1]:

dp[w][j] = A[j] + dp[w - A[j]][j - 1]

indices[w][j] = indices[w - A[j]][j - 1][:] + [j]

for possible\_sum in range(S // 2, -1, -1):

for j in range(len(A) - 1, -1, -1):

index = indices[possible\_sum][j]

if dp[possible\_sum][j] == INF:

continue

sum\_made = sum([A[k] for k in index])

#print(index)

return abs(S - 2 \* sum\_made)

return 0

* Rod Cutting
  + If we cut at an index j between start to end, we find minimum over all j between start and end

def rodCutHelper(self, B, A):

self.cache = {}

def f(start, end, length):

if start > end or length <= 0:

return (0, [])

tup = ",".join([str(i) for i in (start, end, length)])

if start == end:

if start in (0, B):

return (0, [])

return (length, [start])

if tup in self.cache:

return self.cache[tup]

ans = float("inf")

sequence = []

for j in range(start + 1, end):

left = f(start, j - 1, A[j])

right = f(j + 1, end, length - A[j])

if left[0] + right[0] + length < ans:

ans = left[0] + right[0] + length

sequence = [j] + left[1][:] + right[1][:]

self.cache[tup] = (ans, sequence)

return (ans, sequence)

x = f(0, len(A) - 1, B)

return [A[idx] for idx in x[1]]

## 4/9

* Number of monotonic sequences using DDP, covers the second variation pg 372
* Divide spoils fairly
* ***dp(knapsack, index) = dp(knapsack, index - 1) + dp(knapsack - w[index], index - 1)*** if a weight can be used only once, index - 1 makes a subtle difference in the last term
* *dp(knapsack, index) = dp(knapsack, index - 1) + dp(knapsack - w[index], index)* if a weight can be used more than once, and the sequence of weights doesn't matter.
* <https://leetcode.com/problems/maximum-score-of-a-good-subarray/>
  + Yay solved in O(n)

## 4/12

* Find alphabetically letter sequence from a list of sorted strings in a different language
  + abc, ab ~> a valid order cannot exist if len(A) > len(B) and A precedes B
  + Rest is top sort with 3 colors
* Word Ladder ||
  + BFS without stopping at endWord, and skip marking the endWord visited.
* <https://leetcode.com/problems/maximum-score-of-a-good-subarray/>
  + Two pointers, whenever a new minimum is hit, re-calculate the answer and expand until the new minimum is no longer the minimum.
* Number of subarrays with some Sum / XOR / property
  + Store count of properties for each suffix, and on the way count frequency of *current - required*
* Number of subarrays with B odd elements
  + Not understood
* Problems of type , find quads a + b = c + d / a + b + c + d = target
  + N^ 2, find a pair, and find pairs in required - pair\_sum, caution: invalid pair counts
* Fraction
  + If divisible , or 0 or 1 return the answer, else mod the number and begin with remainder like school maths - ~> rem = rem \* 10, rem = rem % B
  + Find fraction is recurring, keep track of if the remainder has occurred before, in case yes .. stop and return the answer
  + Some tricky string manipulation for concatenating the required answer
* <https://leetcode.com/problems/number-of-ways-to-arrive-at-destination/>

if ways[i] is the number of ways to arrive at i optimally and j is connected to i the number of ways to arrive at j is ways[j] += ways[i] for all such i such that the distance to j is least slight modification of dijkstra

## 4/13

* Sentence screen filtering, find number of times a sentence fits the screen
  + BF is good, but does not pass O(row \* col)
  + Instead cache the places - how many next words are going to fit If I start filling a row with word = word\_idx from column 0, in case this is cached, just jump through
* Less than O(mn) algo to find the minimum rectangle enough to fit a spread of black pixels
  + Find the coords of rectangle as far as they go
* Remove Boxes
  + Made a O(2 ^ n) from O(n!) algorithm, but still it didn't pass.
  + The correct state is dp(L, R, count), answer for L to R if you assume that you have already counts head start contiguous before L , that’s equal to Arr[L]
  + dp(l, r, count) ~> max((count + 1 ) \*\* 2 + dp(m, r, count + 1) for m in l + 1 to r)
* Typing with two fingers
  + L, R, index was the correct state
  + I used Dijkstra on cost, index, L, R ~> TLE, when removing the cost it passed,

Usually the objective (value) which is to be maximized or minimized should never be part of the state that is cached, because that is exactly why a cache is needed.

* Count inversions, with a[i] > 2\* a[j] similar to count inversions problem

4/14

LL questions

## 4/17

* Check bipartite-ness of a graph
  + Can Wired connections be arranged in 2 groups such that there is a connection only between left and right half
  + Wired connections can be disjoint
  + Every bipartite graph is 2 - colorable, check with BFS shortest path, child color can be 1 - color\_of\_parent, if child is already visited check if color is still 1 - color\_of\_parent, if not graph is not bipartite.
  + Check with all the components of the graph.
* Find minimum height of arrangement of books on a bookshelf with given height
  + Very intuitive DP solution, O(shelfwidth \* number\_books)
  + Choices:
    - If shelf width is 0, place a book
    - If shelf width is non-zero (some part is used)
      * Place book on the same shelf
      * Place book on a new shelf
* Solving some random Design questions

## 4/21

Problem statement

* 10 mins on understanding
* 15 mins to get DFS
* 10 mins on impn
* 10 on last discussion

I think I should keep a timer in phone side by side

Understand fast

Clarify fast

And start thinking by writing thoughts on the docs

## 