BUBBLE SORT

BEST $O(n)$ ← array already sorted

AVG $O(n^2)$

WORST $O(n^2)$

- compares adjacent pairs of elements
- swaps if they are in wrong order
- with every iteration it puts the greatest left in correct position

```
def bubble_sort(arr):
    def swap(i, j):
        arr[i], arr[j] = arr[j], arr[i]

    n = len(arr)
    swapped = True
    x = -1
    while swapped:          # stop if no more swaps are made in last iteration
        swapped = False
        x = x + 1
        for i in range(1, n-x):   # decrease by 1 because end of array is sorted
```

```
        if arr[i-1] > arr[i]:
            swap(i-1, i)
            swapped = True
```

new swaps were made

return arr.

↳ in place sorting

SELECTION SORT

BEST	AVG	WORST
$O(n^2)$	$O(n^2)$	$O(n^2)$

- select the smallest element from unsorted part
- place it in sorted part

```
def selectionSort(arr):
    for i in range(len(arr)):
        minimum = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[minimum]:
                minimum = j
        arr[minimum], arr[i] = arr[i], arr[minimum]
return arr
```

select the smallest value

↳ place it at the front of the array i.e. in the sorted part

INSERTION SORT

BEST	AVG	WORST
$O(n)$	$O(n^2)$	$O(n^2)$

- begin from first element and traverse the array.
- keep sorting the array from the beginning to i'th element by shifting the elements

```
def insertion_sort(arr):
    for i in range(len(arr)):
        cursor = arr[i]
        pos = i
        while pos > 0 and arr[pos-1] > cursor:
            arr[pos] = arr[pos-1]
            pos = pos-1
        arr[pos] = cursor
    return arr
```

↗ shift the number to its right position

↘ place it in the right position in the sorted array

MERGE SORT

Divide & Conquer
O(n) space complexity

BEST	AVG	WORST
$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

- continuously divide the array into halves until it cannot be further divided.
- merge the halves continuously into a sorted array

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left, right = merge_sort(arr[:mid],
                             arr[mid:])
    return merge(left, right, arr.copy())
def merge(left, right, merged):
    left_cursor, right_cursor = 0,0
```

↗ cost array split

↗ sort halves

MINISO

compare right & left halves

```
while left_cursor < len(left) and right_cursor<len(right):
    if left[left_cursor]<=right[right_cursor]:
        merged[left_cursor+right_cursor]
            = left[left_cursor].
        left_cursor +=1
    else:
        merged[left_cursor+right_cursor] =
                right[right_cursor]
        right_cursor +=1

for left_cursor in range(left_cursor, len(left)):
    merged[left_cursor+right_cursor] =
    left[left_cursor].
```

put remaining elements

```
for right_cursor in range(right_cursor, len(right)):
    merged[left_cursor+right] =right[right_cursor]

return merged
```

FREE NOTE

MINISO

Divide & Conquer

QUICK SORT

O(log(n)) space complexity

↳ built in functions use a randomized quick sort

BEST	AVG	WORST
O(nlogn)	O(nlogn)	O(n^2) (rarely)

- select an element as the pivot (first/last/random)
- move all elements that are smaller than the pivot to its left and all elements that are larger to its right.
- recussively apply the above 2 steps to each subarray to the left & right of the pivot. (in-place sorting)

```
def partition(arr, low, high):
    i = low-1
    pivot = arr[high]

    for j in range(low, high):
        if arr[j] <= pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
```

pivot

```
    arr[i+1], arr[high] = arr[high], arr[i+1]
return (i+1)
```
→ return partition index

```
def quickSort(arr, low, high):
    if low < high:
        pI = partition(arr, low, high)
        quickSort(arr, low, pI-1)
        quickSort(arr, pI+1, high)
```
left / right

RADIX SORT

→ uses counting sort as subroutine

BEST	AVG	WORST
O(nk)	O(nk)	O(nk)

- take the least significant digits of the values to be sorted
- sort the list of elements based on the digit
- keep repeating 'k' times
  ↳ max. no. of digits

```
def countingSort(nums, exp):
    count = [0 for i in range(10)]
    res = nums.copy()

    for i in nums:
        pos = int((i/exp) % 10)
        count[pos] += 1
```
→ find the digit

```
for i in range(1,10):
    count[i] += count[i-1]

for i in range(len(nums)-1, -1, -1):
    pos = int((nums[i]/exp)%10)
    res[count[pos]-1] = nums[i]
    count[pos] -= 1

return res

def radixSort(nums):
    high = max(nums)
    exp = 1
    while high/exp > 0:        # find number of max digits
        nums = countingSort(nums, exp)
        exp *= 10
    return nums
```


stable sort

COUNTING SORT

BEST	AVG	WORST
O(n+k)	O(n+k)	O(n+k)

k is the range of values in the array

- if the values of elements are in range 0.....k
- count occurences of these values in the array
- find cummulative sums
- thus now you have indexes

```
def countingSort(nums):
    high = max(nums)
    low = min(nums)            # to map negative integers
    r = high - low + 1
    res = nums.copy()
    count = [0 for i in range(r)]
    for i in nums:
        count[i-low] += 1      # count occurrences
```

```
for i in range(1, r):          ← cummulative sum
    count[i] += count[i-1]

for i in range(len(nums)-1, -1, -1):
    res[count[nums[i]-low]-1] = nums[i]
    count[nums[i]-low] -= 1

return res
```

place element at right position then decrement count.

HEAPSORT

uses heap data structure

similar to selection sort first finding the max element, but more does it more efficiently

BEST	AVG	WORST
O(nlogn)	O(nlogn)	O(nlogn)

- since binary heap is a complete binary tree, it can be easily represented as array.
  i → 2i+1, 2i+2 children

- build a max heap from input data
- swap the largest item with last element of heap, reduce size of heap by 1.
- heapify root.
- repeat until size of heap is greater than 1.

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
```

trickle down

```
if l < n and nums[l] > nums[i]:
    largest = l

if r < n and nums[r] > nums[i]:
    largest = r

if i != largest:
    nums[i], nums[largest] = nums[largest], nums[i]
    heapify(largest)

def heapSort(nums):
    n = len(nums)

    for i in range(n, -1, -1):        # build max heap
        heapify(nums, n, i)

    for i in range(n-1, 0, -1):
        nums[i], nums[0] = nums[0], nums[i]   # swap root with last element and reduce heap size
        heapify(nums, i, 0)
```


RUNNING MEDIAN

FIND THE MEDIAN OF AN INCOMING STREAM OF INTEGERS (ONLINE)

Naive →

- sort the integers seen so far, calculate and return the median

time complexity $O(n \times n\log n)$ ↑ sorting

Optimized Solution →

- use two heaps – one max heap to store number less than the median, a min heap to store number greater than the median
- balance sizes of the heaps

time complexity $O(n\log n)$ ↑ insertion

heapq → multiply values by -1 to store as a max heap.

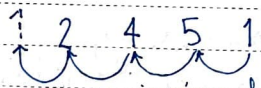
→ implementation in python

COUNT INVERSIONS

↳ also count the no. of smaller elements to the right side after self

- use merge sort to count the number of inversions

- inversions = left inversions
  inversions += right inversions
  inversions += merge inversions



inversions for 1 is 4

while merging, (right)

arr[j] < arr[i]

inv += mid - ~~xxxx~~ i

SORTING PROBLEMS

MEDIAN OF TWO SORTED ARRAYS OF DIFFERENT (SAME) SIZES

- use medians of each array keep dividing the array — O(log n)
- merge till n/2 iterations O(n)

ROTATED SORTED ARRAY SEARCH

- find pivot, then binary search
  or
- find mid, if arr[1 ... mid] is sorted or arr[mid ... 1] is sorted — O(log n)

3 SUM ZERO

sort and find 2 sum using 2 ptrs for every element

SORT LINKED LIST IN NLOGN

merge sort
slow & fast ptr to find mid

INSERTION SORT SINGLY LINKED LIST

create separate sorted list

MINISO

FREE NOTE

MINISO

ARDEN DERTAT

1. ARRAY PAIR SUM
   sort - 2 pointers / map indices
2. MATRIX REGION SUM
   store area of rectangle with (0,0) as corners and other opposite corner can be changing.. ABCD = OD - OB - OC + OA
3. LARGEST CONTINUOUS SUM
   Kadane's algorithm
4. FIND MISSING ELEMENT
   XOR all / find sums / sort / hashmap
5. LINKED LIST REMOVE NODES
   consider all corner cases
6. COMBINE TWO STRINGS
   recursion with DP - O(nm) only recursion exp.
7. BINARY SEARCH TREE
   range based solution / inorder traversal.
8. TRANSFORM WORD
   breadth first search
9. CONVERT ARRAY
   tricky O(n^3) super linear, recursively swapIndex

10. K^{th} LARGEST ELEMENT IN ARRAY
sorting / priority queue / selection sort / median of medians

11. ALL PERMUTATIONS OF STRING
Backtracking.

12. REVERSE WORDS IN A STRING
common string manipulation

13. MEDIAN OF INTEGER STREAM
using two heaps - highers & lowers

14. CHECK BALANCED PARANTHESIS
stack

15. FIRST NON REPEATED CHARACTER IN A STRING
hashtable

16. ANAGRAM STRINGS
hashtable O(n) / sorting

17. SEARCH UNKNOWN LENGTH ARRAY
linear search / reverse / modified binary search - 1, 2, 4, 8.... 2^k
O(logn)

18. FIND EVEN OCCURRING ELEMENT
hashtable / sorting / set + XOR.

MINISO

used for range queries (with updates) } largest element in range (L,R) / sum of elements in (L,R) etc

SEGMENT TREES / INTERVAL TREES

↳ query can be reduced from O(n) to O(logn)

Let us consider the following problem to understand Segment Trees:
We have an array arr[0...n-1]. We should be able to:

① find the sum of the elements from index l to r where $0 <= l <= r <= n-1$

② Change value of a specified elements of the array to a new value x. We need to do arr[i] = x where $0 <= i <= n-1$.

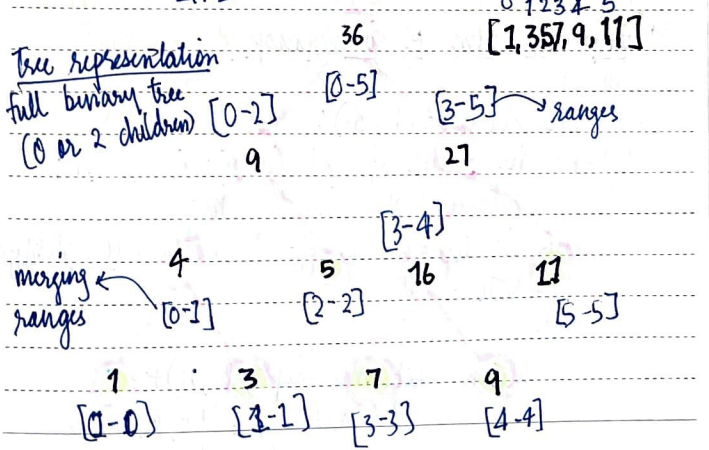
Solution 1: calculate sum in O(n)
update in O(1)

Solution 2: cache sum from 0 to i^{th} index DP
calculate sum is O(1) now
but update is O(n)

Solution 3: Using segment trees
update in O(logn).
calculate sum in O(logn).

Representation of segment trees
- leaf nodes → elements of input array
- internal nodes → some merging of leaf nodes.
- array representation is used.

i → left child 2i+1
i → right child 2i+2
i → parent (i-1)/2



Tree representation
full binary tree (0 or 2 children)

array representation

[36, 9, 27, 4, 5, 16, 11, 1, 3, Dummy, Dummy, 7, 9, Dummy, Dummy].

size of array → if n power of 2 no dummy nodes
→ else 2n−1
(smallest n which is next power of 2.

Construction of segment tree

- divide current segment in two halves & repeat
- each half store sum.

Query for sum of given range.

```
int getSum(node, l, r):
    if the range of node is within l & r:
        return value in the node
    else if the range of the node is completely outside
        return 0
    else
        return getSum(left, l, r) +
               getSum(right, l, r).
```


Update

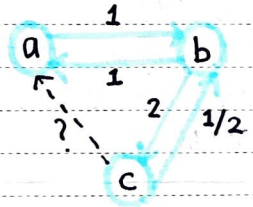
- start from root, add diff to all nodes with given index in their range.

ORDER RECONSTRUCTION BY HEIGHT

- all permutations $O(N!)$
- sort acc. to height, place acc. to infronts $O(N^2)$
- segment/interval trees to find the i^{th} empty position. $O(n \log n)$

EVALUATE DIVISION

a/b = 1 b/c = 2
find c/a.

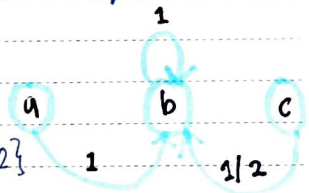


Solution 1
- create graph from given equations O(e)
- do graph traversal dfs for query O(qe).

Solution 2 - Union Find

a/b = 1 → parents[a] = {b, 1}
parents[b] = {b, 1}

b/c = 2 → parents[c] = {b, 1/2}



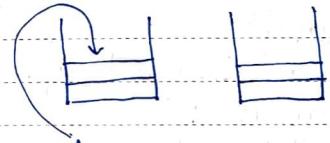
a/c ? p[a].key == p[b].key == B
= p[A].val / p[c].val = 1/1/2 = 2

Time complexity: O(e+q).
Space complexity: O(e).

ITERATIVE POSTORDER

USING TWO STACKS

1. Create stack 1 & stack 2



2. Push root
3. While stack is not empty:
   - pop from stack 1 → push to stack 2
   - push its children into stack 1.
4. Pop from stack 2 and push to result array of postorder traversal.

```
def postOrder(root):
    if root is None:
        return []
    stack_1 = [root]
    stack_2 = []
```

```
while len(stack_1) > 0:
    top = stack_1[-1]
    stack_1.pop(-1)
    stack_2.append(top.val)

    if top.left:
        stack_1.append(top.left)
    if top.right:
        stack_2.append(top.right)
return stack_2[::-1]
```


ITERATIVE INORDER

USING A STACK

- set current to root
- push current to stack
- set current to current → left
- repeat until current is NULL
- when current is NULL, print the top from stack
- push its right child in stack, set it to current
- keep pushing its left and repeat

```
def inorderTraversal(self, A):
    if A is None:
        return []
    stack = []
    res = []
    current = A
    while current:
        stack.append(current)
```

```
        current = current.left
    while current is None and len(stack)>0:
        top = stack[-1]
        stack.pop(-1)
        res.append(top.val)
        current = top.right
return res
```


ITERATIVE PREORDER

USING A STACK

- create an empty stack, push root to it
- while stack is not empty -
  - → pop top & print
  - → push right
  - → push left.

```
def preorderTraversal(A):
    if A is None:
        return []
    stack = [A]
    res = []
    while len(stack)>0:
        top = stack[-1]
        stack.pop(-1)
        res.append(top.val)
```

```
        if top.right:
            stack.append(top.right)
        if top.left:
            stack.append(top.left)
    return res
```

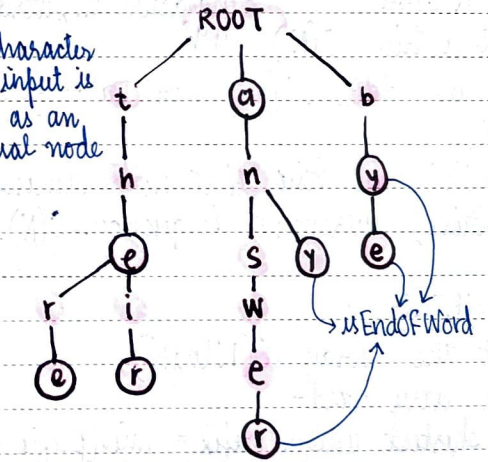

TRIES

- stores strings
- prefix trees
- size equal to size of alphabet

Search	Insert	Delete
O(l)	O(l)	O(l)

l → length of string

every characters of the input is inserted as an individual node




```
class TrieNode:
    def __init__(self):
        self.children = [None]*26
        self.isEndOfWord = False
```

SHORTEST UNIQUE PREFIX

USING A TRIE

- create a trie $O(N)$ → characters in all words. (pq.)
- add an addition 'frequency' value for each node
- everytime you visit that node during insertion, increment frequency. $O(N)$.

USING SORTING

- sort the array $O(N \log N)$
- for every word -
  shortest unique prefix = min(prefix with left neighbours, prefix with right neighbours)

STRONGLY CONNECTED COMPONENTS - undirected

1. DFS
   - perform DFS on the entire graph
   - initially set all nodes as unvisited
   - if after a component is visited, you will need to iterate to next node
   - increment count. $O(V+E)$
2. BFS.
   - similarly perform BFS on all components. $O(V+E)$.

TOPOLOGICAL SORT

- sorting of vertices of a directed acyclic graph
- ordering of vertices in such a way that if there is an edge directed towards v from u, the u comes before v.
- multiple topological sorting possible for a graph
- can be done using BFS or DFS
  - BFS → requires (KAHN'S ALGORITHM) in-degree array
  - DFS → recursion stack

★ for lexographically smallest topological sort
- sort adj list values of keys decreasingly } DFS
- start from last node

- use priority queue in Kahn's algorithm

MINISO

DIJKSTRA'S ALGORITHM

(similar to Prim's algorithm)

single - source shortest path

↳ greedy algorithm

- create a shortest path tree set
- initialise all distance values as ∞ except src.
- while shortest path set doesn't include all vertices -
  - pick a vertex which is not in shortest path with min dist value
  - include it to shortest path
  - update its adjacent vertices

$O((V+E)\log V)$

doesn't work for negative weight edges

can be used for both directed & undirected graph

subgraph (i.e. a tree) of a graph that connects all vertices together

MINIMUM SPANNING TREE

PRIM'S ALGORITHM

Similar to Dijkstra's Algorithm $O(ElogV)$

KRUSKAL'S ALGORITHM

(using Union-Find Algorithm)

1. Sort all edges in ascending order acc. to weight
2. Pick smallest edge, check if it forms a cycle with the spanning tree formed so far. and include it if not
3. Repeat until spanning tree contains all edges.

$O(ElogE + ElogV)$.

USEFUL PYTHON FUNCTIONS

COPYING A LIST

```
copy_of_arr = arr[:]
```

CREATING A CLASS

```
class Dog():
    def __init__(self,name):
        self.name = name
    def sit(self):
        print(self.name)
```

INHERITENCE

```
class Student(Person):
    def __init__(self, name):
        Person.__init__(self,name)
        self.graduate = 2019
```

LOOPING THROUGH DICT

```
for key,value in d.items():
```

LOOPING THROUGH KEYS:

```
for key in d.keys():
```

LOOPING THROUGH VALUES:

```
for value in d.values():
```

INSERT ELEMENT AT INDEX

```
usess.insert(0,"joe")
```

DELETE ELEMENT AT INDEX

```
del usess[-1]
```

REMOVE ELEMENT BY VALUE

```
usess.remove('mia')
```

POPPING ELEMENTS

```
usess.pop(0)
```

MINISO

START OF LOOP IN LINKED LIST

HARE & TORTOISE:

- use Two pointers - one moves one node at a time other moves ~~other~~ two nodes at a time
- if any pointer is null → no loop
- if they meet at any point → loop found

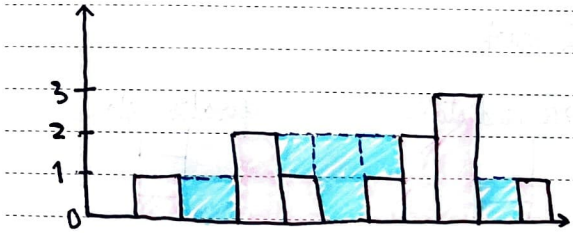
Detecting start of loop:

- use another two pointers - one from beginning of linked list other at the point where the previous pointers met
- ~~two~~ iterate both one node at a time
- where they meet is the starting point

FREE NOTE

MINISO

TRAPPING RAIN WATER



INTUITION

collect the rainwater on top of each building and add it

NAIVE

for building a[i], find left tallest building right tallest building..

waterAbove = min(a[left], a[right]) - a[i]

$O(n^2)$ → time

DYNAMIC PROGRAMMING

store left largest so far and right largest so far

$O(n)$ $O(n)$ → space.

EVALUATE INFIX EXP

using 2 stacks

operand stack

operator stack

① push numbers in operand stack
② '(' into operator stack
③ for ')' →
    while s.top() != '(':
        pop operator
        pop 2 operands
        apply
        push result in operand
    pop '('
④ for an operator while reading input (thisOP)
    while s.top() has same or greater precedence:
        pop operator

        pop operands
        apply
        push result.
⑤ push thisOP in operator
⑥ while operator stack is not empty:
    pop operator
    pop operands
    apply push
⑦ pop result from operand stack

EVALUATE POSTFIX EXP

Using single stack
- if number → push
- if operator → pop two operands
  apply operator
  push value

EVALUATE PREFIX EXP

- iterate from end of the expression
- evaluate as postfix

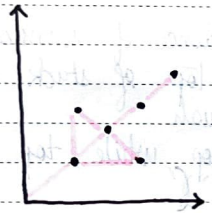
INFIX → POSTFIX

① if operand ~~push to stack~~ output it
② if operator -
  → if precedence of scanned operator > on the top of stack or '(' then push
  → else pop while top is greater or '('
③ if '(' push
④ if ')', pop until '(' and output

INFIX → PREFIX

① reverse infix, '(' → ')' and ')' → '('
② get postfix
③ reverse postfix

MAX POINTS ON SAME LINE



$O(n^2)$ → time
$O(n)$ → space

- calculate slope of every point with every other point
- store slope in map
  (to avoid round off errors store as $((x_1-x_2),(y_1-y_2))$ pairs reduced by their gcds
- slope with max points is the answer
- avoid duplicate points
- take care of horizontal & vertical lines
- take care of negative slope.
- take care of parallel lines

wildcard matching

USING DYNAMIC PROGRAMMING:

string = "baaabab"
pattern = "ba*a?" → matches a single character.
matches any sequence of characters (can be empty)

`s[i] == p[j]` or `p[j] == '?'`
`dp[i][j] = dp[i-1][j-1]` → if current char match, it depends on previous chars

`s[i] != p[j]` and `p[j] != '*'`
`dp[i][j] = False` → curr char don't match

`p[j] == '*'`
`dp[i][j] = dp[i-1][j] || dp[i][j-1]`
- `dp[i-1][j]` → matching with *
- `dp[i][j-1]` → matching with zero chars

regex matching

USING DYNAMIC PROGRAMMING

String = xaabyc pattern = xa* b.c

'.' → matches one character

'*' → matches same preceeding characters

① if str[i] == pattern[j] || pattern[j] == '.'
matches current characters
dp[i][j] = dp[i-1][j-1]

② if pattern[j] == '*'
dp[i][j] = dp[i][j-2] || dp[i-1][j]
one or more occurence
zero occurence of char before *

0-1 KNAPSACK

Given weights and values of n items, put these in a knapsack of capacity W to get the maximum total value.

Recursion → generate all subsets with weight ≤ W and pick one with max value
O(2^n)

Dynamic Programming →
if wt[i] <= W
don't include item
dp[i][w] = max(dp[i-1][w], dp[i-1][w-wt[i]] + val[i])
include item

else → can't include because Weight exceeds
dp[i][w] = dp[i-1][w]

COIN CHANGE PROBLEM

Given coins of values $S_1, S_2, S_3 \ldots S_m$ and given total value N, we want to know how many possible ways to form N using the change.

Minimum coins required.

① set min[i] = ∞ for all 0<i<N

② min[0] = 0 for =1

③ for 0<j<m:

- if Sj <= i and min[Sj+i]+1 < min[i]
  min[i] = min[i-Sj] + 1

④ return min[N]

Total number of ways

① dp[i][j] = dp[i-1][j] + dp[i][j-Si]

result is dp[n][m]

3-DIMENSIONAL BFS

THE PROBLEM

Given a mxn matrix representing a maze. X is the starting point, Y is the destination, '.' is walkable, # is a wall. You can walk in 4 directions: u, d, r, l. You are allowed to break atmost k walks. What is the minimum steps to walk from X to Y?

```
. X . # . .
. # . . # .
. # . . . .      K=3
. # # . . .
. Y . . . .
. . . . . #
```

If we couldn't break walls, we do a simple BFS to all walkable states

If we could break atmost 1 wall, we could do BFS with extra state added to it if wall broken so far or not. i, j, k → 0 walls broken
→ 1 wall broken

If we could break atmost k walls
visited [i][j][k]
and in queue keep state for no. of walls broken so far.

PROBABILITY OF KNIGHT TO REMAIN ON CHESSBOARD

Given a NXN chessboard and a knight at position (x,y). The knight has to take exactly k steps in any 8 directions. What is the probability that the knight remains in the chessboard after taking K steps, it cannot enter the board once it leaves.

dp[x][y][steps] → stores the probability of reaching (x,y) after 'steps' number of moves.

if steps = 0, p = 1. ← base case.

TOP DOWN DP → keep a visited matrix too

① Initialize dp[x][y][0] = 1 for all x, y.

② From x, y : generate neighbours.
dp[x+dx][y+dx][steps+1] = $\frac{1}{8}$ × dp[x][y][steps] } BFS

③ Return $\sum$ dp[x][y][k] for all x, y.

MINISO

LONGEST INCREASING SUBSEQUENCE

	1	4	3	5	6	2	8
dp[i]	1	2	2	3	4	2	5

```
for i in range(n):
    for j in range(i-1, -1, -1): if a[j]<a[i]:
        dp[i] = max(dp[i], dp[j]+1)
return max(dp)
```

Time $O(n^2)$

Space $O(n)$

FREE NOTE

MINISO

LONGEST COMMON SUBSEQUENCE

S1 = AGGTAB

S2 = GXTXAY

```
if s1[i] == s2[j]:
    dp[i][j] = dp[i-1][j-1]
else:
    dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

Linear Diophantine Equations

A diophantine equation is a polynomial equation with two or more unknowns, such that only integral solutions are required.

$$ax + by = c$$

For linear diophantine equations, solutions integral exist if and only if

```
c % gcd(a, b) == 0
```


FIBONACCI SEQUENCE

using matrix exponentiation

$$\begin{bmatrix} f(n) \\ f(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-1) \\ f(n-2) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-2) \\ f(n-3) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} f(n-3) \\ f(n-4) \end{bmatrix}$$

$$\vdots$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} f(1) \\ f(0) \end{bmatrix}$$

solve this using binary exponentiation

Time Complexity $O(\log n)$

Space Complexity $O(1)$

ODD EVEN JUMP

APPROACH 1 Brute force.

Simulate the deterministic jumping process on each element.

$O(n^2)$.

APPROACH 2 DP + Binary Search (TreeMap)

(up) Odd jump → smallest value greater than self

(down) Even jump → largest value less than self

map → keep sorted indexes $O(n \log n)$

Start from $(n-2)^{th}$ element, (reverse order $O(n)$)

find odd jump index j

find even jump index k.

$dp[i][0] = dp[j][1]$ (can reach from j to end with even jump)

$dp[i][1] = dp[k][0]$ (next jump is odd; can reach from k to end with odd jump)

APPROACH 3 Monotone Stack (used in largest rectangle area in a histogram.)

1. get indexes of elements when the array is sorted by increasing value
2. get indexes of elements when the array is sorted in decreasing value.
3. use a monotonically decreasing stack
4. find next jump index for each index in odd jumps
5. find next jump index for each index in even jumps
6. construct odd & even arrays i.e. if we can reach from i to end in odd / even jump.
7. return sum(odd)

LRU CACHE

Using
- **queue**
  Implemented using a doubly linked list
  to represent the current cache
- **hash**
  With page numbers as key and address of corresponding queue node as value.

When a new page arrives –
- if cache is not full, attach at front
- if already in cache, move it to front
- if cache full, remove the last node and add new page at front.

STRING SEARCHING

NAIVE

for every index of str1, search 'i' as starting point for str2.
O(nxm)

KNUTH MORRIS PRATT

S : a b c d a b c d a f
P : a b c d a f
↑ ↑

S : a b a d a b a d a f
P : a b a d a f

S : a a a a a a a a a b
P : a a a b

The basic idea is whenever we detect a mismatch, we already know some of the characters in S which will match any and avoid them.

Preprocessing

- KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m which is used to skip characters while matching.
- lps → longest proper prefix which is also a suffix.

lps[i] = the longest proper prefix of pat[0...i] which is also a suffix of pattern[0...i]

Eg.

pat : AAAA
lps : 0123

pat : AABAACAABAA
0 1 0 1 2 0 1 2 3 4 5

Searching

i → 0 . . . . n

- we start comparison of pat[j] with j=0.
- increment i & j if matched
- In case of mismatch -
  → use lps to avoid matching the prefix again which is also a common suffix
  → move j to lps[j-1].

Code O(n+m)

```
def preprocess(p, lps, m):
    len = 0
    i = 1
    while i<m:
        if p[i] == p[len]:
            len+=1
            lps[i] = len
            i+=1
        else:
```

```
        if i > 0:
            len = lps[len-1]
        else:
            i += 1

def kmp(s, p):
    n = len(s)
    m = len(p)
    for i in
    lps, j = [0]*m, 0
    preprocess(p, lps, m)
    for i in range(n): while i < n:
        if s[i] == p[j]:
            j += 1
            i += 1
        else: if j > 0:
            j = lps[j-1]
            else
                j = 0, i += 1
        if j >= m:
            return True
    return False
```


RABIN KARP

Matches hash value of current window of text with the hash value of the pattern

Preprocessing

- calculate hash value of pattern
- calculate hash value of all substrings of text of length m.

* hash value of next shift must be efficiently computable from current hash value and next character

Hashing

$$p = (p[0] * d^{m-1} + p[1] * d^{m-2} + p[2] * d^{m-3}) \% m$$

(d^{m-1}: total no. of characters; m: prime number)

$$h(s[m]) = (h(s[m-1]) / d^{m-1} + s[m] * d^{m-3} - s[0]) * d$$

manacher's Algorithm

LONGEST PALINDROMIC SUBSTRING

Naive → for every substring, check if it's a palindrome or not
$O(n^3)$

Optimisation → assume every character or space b/w two characters in the string as the center for a palindrome and find the longest possible
$O(n^2)$

Efficient →

- add a special character such as a '#' in between every character of the string as well as beginning and end of array.
- generate LPS array.

for eg.

#	a	#	b	#	a	#	a	#	b	#	a	#
0	1	0	3	0	1	6	1	0	3	0	1	0

LPS :

notice that the lps mirrors itself after the center of the longest palindrome found so far.

Calculating LPS array :

HAMILTONIAN CYCLE

Undirected graph - it is a path that visits each vertex exactly once.

Hamiltonian cycle - if there is an edge between the first and last vertex of hamiltonian path.

Q Determine whether a given graph contains Hamiltonian cycle or not.

NAIVE

- Generate all possible configurations of vertices
- check for if any path follows through any configuration
  $O(n!)$

BACKTRACKING

- create an empty path array.
- add different starting vertexes
- before adding, next vertex, check if

it is already added and if there's a path or not.

- return true if all vertexes added
- return false and backtrack if the current path is not possible

NP-Hard

TRAVELLING SALESMAN

Given a set of cities and distance between every pair of cities, find the shortest possible route that visits every city exactly once and returns to starting point.

↳ find all possible hamiltonian cycles and return the one with lowest cost?

NAIVE

- consider city 1 as the starting and ending point.
- generate all $(n-1)!$ permutations of cities
- calculate cost of each and keep track of minimum.

$O(n!)$

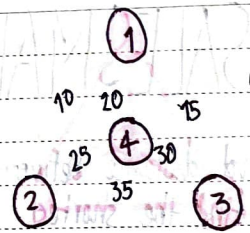
DYNAMIC PROGRAMMING

top down

~~bottom up~~

$f(i, s) = \min(C_{ik} + f(k, s))$

MINISO



set of all vertices.

$C(S,i) = \min\{C(S-\{i\},j) + dis(j,i)\}$

where j belongs to S,
$j != i$ and $j != 1$

$O(n^2 * 2^n)$

FREE NOTE

MINISO

WARNSDORFF'S ALGORITHM knight's tour

NAIVE (backtracking)

- check all possible paths using backtracking
  $O(n!)$

WARNSDORFF (linear)

- set P to be a random initial position on the board.
- mark the board at P as 1
- do ~~ear~~ following for next moves -
  → let S be the set of positions accessible from P.
  → set P to the position in S with minimum accessibility → min. no. of unvisited adjacent
  → mark the board at P with next move

MAX PROFIT WITH K TRANSACTIONS

DYNAMIC PROGRAMMING

- $profit[i][j] = max\begin{cases} profit[i][j-1] & \text{not selling on } j^{th} \text{ day} \\ prices[j] - prices[k] + profit[i-1][k] & \text{for } 0 \le k < j \end{cases}$

  (i^{th} transaction; j^{th} day)

  → buying on k^{th} day and selling on j^{th} day plus profit till k^{th} day with one less transaction

Time Complexity - $O(n^2k)$.
Space Complexity - $O(nk)$

OPTIMIZATION.

$max(price[j] - price[k] + profit[i-1][k]$

$= price[j] + max(profit[i-1][k] - price[k])$
$0 \le k < j$

$= price[j] + max(prevDiff, profit[i-1][k-1] - price[k-1])$.

where $prevDiff = max(profit[i-1][j] - price[j])$
$0 \le j \le i-2$

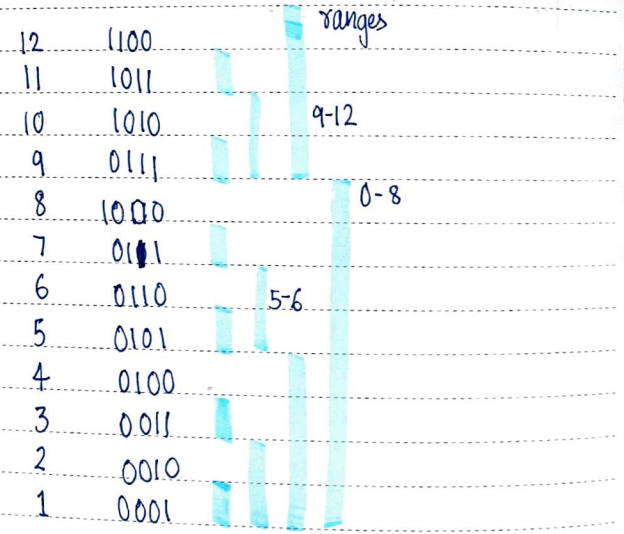
Time Complexity - $O(nk)$
Space Complexity - $O(nk)$

FENWICK TREES

MINISO

BINARY INDEXED TREES

We have an array a[0:.....n].
1) Compute sum of first i elements.
2) Modify the value of arr[i] = x.



FREE NOTE

MINISO

The size of BITree[] is equal to the size of the input array.

```
def getSum(x):
    sum = 0
    x = x+1
    while x>0:
        sum += BITree[x]
        x = x-(x&-x)        # find parent
    return sum

def update(i,x):
    x = x+1
    while x<=n:
        BITree[x] += x
        x = x+(x&-x)        # update all ranges.
```

RANGE UPDATES

```
update(l,r,val):
    arr[l] = arr[l]+val
    arr[r+1] = arr[r+1]-val

getElement(x):
    sum(x)
```

FIND LOCAL MINIMA IN AN ARRAY

Given an array arr[0....n-1] of distinct integers, the task is to find a local minima. A local minimum is less than or equal to both its neighbours.

→ for corner elements, consider only one neighbour

→ there can be more than one local minima, find any one.

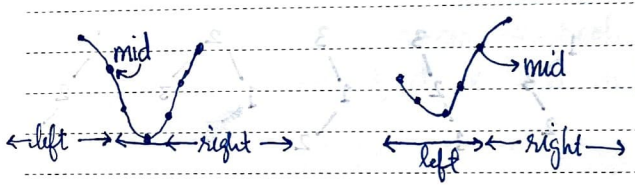
LINEAR SEARCH

Do a linear scan on the array and as soon as we find local minima, we return it.

T.C. - O(n)

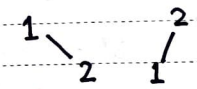
BINARY SEARCH

- if middle element is not greater than any of its neighbours, return it.
- if middle element is greater than left neighbour, then there will always be a local minima in left half.
- if middle element is greater than its right neighbours, then there will always be a local minima in right half.

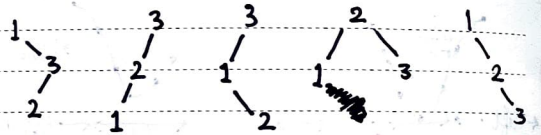


CONSTRUCT ALL POSSIBLE BSTs FOR KEYS 1 - N.

N=2

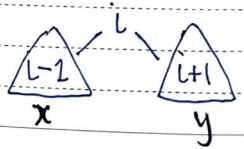


N=3



- We know that all nodes in left subtree are smaller than root and in right subtree are larger than root.

- if we have i^{th} number as root, all numbers from 1 to i-1 will be in left subtree and i+1 to N in right subtree.



- if 1 to i-1 can form x different trees and i+1 to N can form y different trees, then we will have x*y different trees with i^{th} root.

- we han 1 to N choices for roots, so for each root, find x and y.

- These form N^{th} CATALAN NUMBER

```
def constructTrees(start, end):
    res = [ ]
    if start > end:
        res.append(None)
        return res.

    for i in range (start, end+1):
        leftSubTree = constructTrees(start, i-1)
        rightSubTree = constructTrees(i+1, end)
        for j in range(len(leftSubTree)):
            for k in range(len(rightSubTree)):
```

```
node = newNode(i)
node.left = leftSubTree[j]
node.right = rightSubTree[k]
res.append(node)
return res
```


CATALAN NUMBER

Catalan numbers are a sequence of natural numbers that occur in many interesting counting problems like following:

1) Count number of expressions containing n pairs of parantheis which are correctly matched
2) Count the number of possible BST with n keys
3) Count number of full binary trees with n+1 leaves.

0 1 2 3 4 5 6 ...

4. 1, 1, 2, 5, 14, 42, 132, ...

RECURSIVE FORMULA

$C_0 = 1$

$$C_{n+1} = \sum_{i=0}^{n} C_i C_{n-i} \quad \text{for } n \geq 0$$

DYNAMIC PROGRAMMING

```
for i in range(2, n+1):
    catalan[i] = 0
    for j in range(i):
        catalan[i] += catalan[j] * catalan[i-j-1]
```

T.C- $O(n^2)$

BINOMIAL COEFFICIENT

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

T.C- $O(n)$

OR

$$C_n = \frac{(2n)!}{(n+1)!\,n!} = \prod_{k=2}^{n} \frac{n+k}{k} \quad \cdot n \geq 0$$

LONGEST COMMON SUBSTRING

BRUTE FORCE

1. generate all substrings of one string
2. do string search on second string

TC- $O(m^2 n)$

DYNAMIC PROGRAMMING

$$dp[i][j] = \begin{cases} \text{if } s[i] == t[j],\ 1 + dp[i-1][j-1] \\ \text{else } 0 \end{cases}$$

TC- $O(mn)$

LONGEST PALINDROMIC SUBSEQUENCE

```
for cl in range(2, n+1):
    for i in range(0, n-cl+1):
        j = i+cl-1
        if str[i] == str[j] and cl==2:
            L[i][j] = 2
        elif str[i] == str[j]:
            L[i][j] = 2+ L[i+1][j-1]
        else
            L[i][j] = max(L[i][j-1],L[i+1][j])
```

LONGEST PALINDROMIC SUBSTRING

```
for cl in range(n):
  dp[i][i] = 1
for cl in range(2, n+1):
    for i in range(0, n-cl+1):
        j = i+cl-1
        if s[i] == s[j] and cl==2:
            dp[i][j] = true
        elif s[i] == s[j]:
            dp[i][j] = dp[i+1][j-1]
```


Another $O(n^2)$ Solution

- iterate through all palindromes characters
- use it as center for palindrome on either sides.

Manacher's Algorithm $O(n)$