1. Insertion Sort
1. Selection Sort
2. Bubble Sort
2. Merge Sort
4. Sort a list of lists
5. Heapsort
6. Quicksort
9. Counting Sort
10. Radix Sort
11. Bucket Sort
12. Binary Search
13. Kadane's Algorithm
14. Strassen's Algorithm
15. Priority queue
16. Find Min / Max simultaneously
17. Kth largest element / Order statistics
18. Boyer-Moore majority voting Algorithm
19. Stack
20. Queue
22. Linked list (single / Double / Circular)
25. Hash table
25. Direct address table
26. Hashing
26. Collision handling (chaining)
29. Hash functions
30. Division method
30. Multiplication method
31. Universal hashing
31. Collision handling (open addressing)
32. Linear probing
32. Quadratic probing
33. Double hashing
35. Perfect hashing
35. Binary Search Tree
35. Interview tips
35. Why use recursion
36. Tail recursion
36. Tree Traversal (inorder, preorder, postorder, morris traversal)
39. Searching
39. Minimum, Maximum
39. Successor (without parent also)
41. Predecessor (without parent also)
42. Insertion
43. Deletion
45. Self-balancing Binary Search Tree
45. AVL Trees
46. Rotations
47. Height
48. Insertion
51. Deletion
54. Count BST
55. Red-Black Trees
56. AA Tree
56. Splay Tree
57. Common Operations
58. Amortized Analysis
58. B-trees
60. Fibonacci Heaps
60. Van Emde Boas Trees
61. LUP decomposition for linear equations
62. Linear programming
63. Polynomials
64. NP-completness
66. Approximation algorithms
67. Vertex Cover problem
68. Traveling Salesman problem
72. Set counting problem
72. Max SAT problem
73. Strings
73. String Searching algorithms
73. Naive algorithm (single pat)
73. Rabin-Karp algorithm (single pat)
75. Knuth-Morris-Pratt algo (single pat)
78. Boyer-Moore algo (single pat)
80. Two-way algorithm (single pat)

## Insertion Sort  Time = O(N²)  Space = O(1), Stable

Takes maximum time if elements are sorted in reverse order. It is used when the size of array is small and array is almost sorted.

It works by splitting array into sorted and unsorted array. e.g.

→ 7 | 8 5 2 4 6 3
Sorted | Unsorted

→ In next step, place 8 at correct position
7 8 | 5 2 4 6 3
Sorted | Unsorted

→ Move 5 to the correct position
5 7 8 | 2 4 6 3
Sorted | Unsorted

Note: For the searching part i.e. where to insert the element you can also use binary search as left array is always sorted. But time complexity remains same as you need to swap the elements also.

```
def insertion_sort(x):
    if len(x)==0: return None
    for i in range(1, len(x)):
        key = x[i], j = i-1
        while j>=0 and x[j] > key:
            x[j+1] = x[j]
            j = j-1
        x[j+1] = key
    return x
```

Swap the elements till you find the correct position to place the element

If the input is singly linked list, you can use the same algorithm. For some element at index 'i', traverse from 0 → i in the sorted array and find the correct position i.e. where to insert an element. Then simply change the pointers.

## Selection Sort  Time = O(N²)  Space = O(1), Unstable

Can be made stable by inserting the element in correct position than swapping

It makes only O(N) swaps and can be useful when memory write is expensive. It works by placing the smallest element on the left, and then finding the second smallest element and placing it at index = 1, .... Thus it also divides the array into sorted and unsorted arrays.

Insertion sort only scans as many elements as it needs to place the k+1 st element, while Selection Sort must scan all the remaining elements to find the k+1 st element. (insertion almost does half as many comparisons)

this however becomes inefficient as you end up spending more time on disk I/O.

It works by recursively sorting the left and right subarray and then merging them.

```
def mergesort (x, left, right):
    if left < right:
        middle = left + (right - left) // 2
        mergesort (x, left, middle)
        mergesort (x, middle+1, right)
        merge (x, left, right)
```

left_arr = x [left : middle+1]
right_arr = x [ middle+1 : right+1]

In python if you pass a list to a function then it will also be modified as the argument merely acts as a pointer.

```
def merge (x, left, right):
    m = left + (right - left) // 2
    l = left
    r = middle + 1
    out = [ ]
    while l ≤ middle and r ≤ right:
        if x[l] ≤ x[r]:
            out.append (x[l])
            l += 1
        else:
            out.append (x[r])
            r += 1
    while l ≤ middle:
        out.append (x[l])
        l += 1
    while r ≤ right:
        out.append (x[r])
        r += 1
    for o, i in enumerate (range (left, right+1)):
        x[i] = out[o]
```

Merge arrays

Add remaining elements

Copy elements back to 'x'

You can also implement mergesort iteratively. In this case, you start from bottom up i.e. merge blocks of size 2, then merge blocks of size 4, and so on

```
def Sort(x):
    insertion-Sort(x, 0, len(x)-1, 0)
    inderes = get_inderes(x, 0, len(x)-1, 1)
    while len(inderes) != 0:
        s, e, k = inderes.pop()
        insertion-Sort(x, s, e, k)
        if k < len(x[0]):
            new_inderes = get_inderes(x, s, e, k+1)
            for index in new_inderes:
                inderes.append(index)
```

} Base information

→ Logic is same as DFS/BFS implemen

- Sort the given range
- Find the consecutive integers in the above range and repeat

**Heapsort** Time = $O(n \log n)$, Space = $O(1)$, Not stable (can be made stable by also taking into account the position of the element)

~~To build heap = O(n)~~

It can be used to sort a nearly sorted array or find the k-th largest element in an array. Quicksort / Mergesort preferred in practice.

Heaps - these are balanced binary trees and can be max-heap / min-heap. In max-heap the parent element value is larger than children.

Max-heap = used for heapsort

Min-heap = priority queue

Times: To convert unordered array to max-heap = $O(n)$

Insert a value / increase it / extract-max = $O(\log n)$

To build the heap, we start from the end of the array. Now we basically assume the children are already max-heaps and we want to insert the parent element, which we basically do by comparing its value to children and repeating it till the leaf node.

There is also d-ary heaps which generalize binary heap to have d-children. With this the decrease operation can be performed quickly at the expense of slower delete minimum operations. This can improve Dijkstra's algo. Also, they have better memory cache, ~~allowing~~ behaviour, allowing them to run more quickly in practice despite having a theoretically larger worst-case running time.

The following is an iterative implementation of heapsort.

To partition the array, there are two methods
- Lomuto partition → the one where you move pivot to right end and then have pointers to maintain end of left array
- Hoare partition → It is more efficient than above (requiring less than 1/3rd comparisons) but also hard to implement

```
def lomuto_partition(x, start, end):
    import random
    pivot_index = random.randint(start, end)
    x[end], x[pivot_index] = x[pivot_index], x[end]

    left_end = start - 1
    for right_start in range(start, end):
        if x[right_start] < x[end]:
            left_end = left_end + 1
            x[left_end], x[right_start] = x[right_start], x[left_end]
    x[left_end + 1], x[end] = x[end], x[left_end + 1]
    return left_end + 1
```

Many ways to choose the pivot
- first/last element
- random index
- randomly get three values and use their median

```
def quicksort(x, start, end):
    if start < end:
        pivot_index = lomuto_partition(x, start, end)
        quicksort(x, start, pivot_index - 1)
        quicksort(x, pivot_index + 1, end)
```

Hoare may not be correct, prefer lomuto in interview

```
def hoare_partition(x, start, end):
    pivot = x[start]
    left, right = start - 1, end + 1
    while True:
        left += 1
        while x[left] < pivot: left += 1
        right -= 1
        while x[right] > pivot: right -= 1

        if left >= right: return right

        x[left], x[right] = x[right], x[left]
```

you have two pointers from each end. Find the value from left which is greater than pivot and find the value which is smaller than pivot from the right. Swap these two and repeat till the pointers meet.

In def quicksort
quicksort(x, start, pivot_index)

## Counting Sort Time = O(N), Space = O(max - min), Stable →

the steps are as follows

1. Make a count array to store the count of each unique object
2. Take cumulative sum of the count array (i.e. each element stores the sum of previous counts)
3. Rotate array clockwise one time (i.e. remove the last value and add 0 in the first place)

- If we only needed to sort numbers, we could output the numbers directly from count array
- The problem with above approach is data comes from count array than the input array. If you had a (value, key) pair then above approach will fail;
- Step 2 addresses this as the cummulative sum gives us the index at which that element should end up in the sorted array
- Step 3 is necessary in case of duplicate values. It makes the sort stable.

Use it when the input range is not significantly greater than the number of elements. It is often used alongside radix sort.

```
def counting_sort(x):                                   # O(n+k) time & space where k = len(count_arr)
    min_value = min(x)                                  # Works for -ve numbers also
    max_value = max(x)                                  # Works for >= 0 values and [min, max] optimized. It can also be done for a string as strings are ASCII (0 <= x <= 255)
    count_arr = [0] * (max_value - min_value + 1)

    for i in range(len(x)):                             # Step 1
        count_arr[x[i] - min_value] += 1

    for i in range(1, len(count_arr)):                  # Step 2: Cumulative Sum
        count_arr[i] += count_arr[i-1]

    for i in range(len(count_arr)-1, 0, -1):            # Step 3: Rotate right
        count_arr[i] = count_arr[i-1]
    count_arr[0] = 0

    out = [0] * len(x)
    for i in range(len(x)):
        val = x[i]                                      # The count_arr[i] corresponds to the index at which index should 'i' be inserted. Then add +1 to it
        val_index = count_arr[val - min_value]
        out[val_index] = val
        count_arr[val - min_value] += 1
    return out
```

## Bucket Sort  Time = O(n)  Space = O(n+k) where k = # buckets

Useful when the input is uniformly distributed over a range. It works for elements in the range [0,1). But if you are given another range like [min, max], then you can calculate the ~~number of buckets as~~ range of each bucket as

range = (max - min) / n  where n = number of buckets

Bucket index for each element can be found using

BucketIndex = (arr[i] - min) / range

Steps: 1. Create n empty buckets
2. Insert each element of the array into a bucket.
3. Sort individual buckets using insertion sort
4. Concatenate all the buckets.

→ Bucket Sort can thus work for floats also, while Counting Sort cannot.

```
def bucket_sort(x, num_buckets):
    min_value = min(x)
    max_value = max(x)
    bucket_range = (max_value - min_value) / num_buckets

    buckets = [[] for _ in range(num_buckets)]   → Do not use [[]]*10 as all the sublists will store the same memory

    for val in X:
        float_index = (val - min_value) / bucket_range
        diff = float_index - int(float_index)   → If this value is 0, then it means it is a boundary element and we add it to the previous bucket

        if diff == 0 and val != min_value:
            buckets[int(float_index()) - 1].append(val)
        else:
            buckets[int(float_index()).append(val)

    for i in range(len(buckets)):                 } Sort the buckets individually
        if len(buckets[i]) != 0:
            buckets[i].sort()

    i = 0                                         } Copy result back to the input
    for bucket in buckets:
        for val in bucket:
            x[i] = val
            i = i+1
```

Similarly if we get F F F, then we should look on right.

=> So this is the intuition. Make this T/F array, and just think about all the possible cases that binary search can end up in. Then your goal is to get to the 'i' that satisfies your requirements (which is simply done by modifying the left/right pointer)

→ Find the smallest element in a rotated array using binary search

6 7 9 15 19 23
F F F F F T T

Here I am looking for the first True. This problem can be solved by comparing it with the first or last element.

```
def get_min(x, start, end):
    while left < right:        -> No <= as middle will be ans
        middle = left + (right-left)//2
        if x[middle] > x[right]:
            left = middle + 1
        else: right = middle
    return x[right]
```

→ Find Sqrt with binary search

$\sqrt{2}$ → check [0, 2]

if $1^2 < 2$, then check [1, 2]
if $1.5^2 > 2$, then check [1, 1.5]
$1.25^2 < 2$, then check [1.25, 1.5]

Repeat this till the difference between height is low is epsilon.

Kadane's Algorithm  Time = O(N), Space = O(1)

goal is to find the largest sum of a contiguous subarray. the idea is as follows

-2 1 -3 4 -1 2 1 -5 4

no point in having this towards sum

no point in having this also

If this is your current ans, then you need to check if adding -5 to it is worth it or not

```
ans = -1e9
current_sum = 0
for x in nums:
    current_sum += x
    ans = max(ans, current_sum)
    current_sum = max(current_sum, 0)   -> If sum becomes <0, then no point in carrying it
return ans
```

This code also handles the case when all the numbers are negative.

## Priority Queue

- Every element has an associated key value
- In case of max-priority queue, element with highest value is removed
- if elements have same value, then the order in which they came to the queue is used.

Priority queue supports the following operations

- Insert - an element
- Extract Maximum
- Increase key (in case of max priority queue, it is assumed the value is increased to a larger value and vice versa for min-queue)

Max priority queue:

- use to schedule jobs on a computer
- when job is finished or interrupted next job is called

Min-priority queue:

- used in event driven simulator. Each element is associated with a time value and the events must be simulated in order of occurrence
- Dijkstra's & Prim's algorithm can also make use of this

(Implementation is similar to Heapsort, but you need to maintain 'heap size' also which is not included in that implementation)

time = O(log n)

```
def extract_max(x):
    if heap_size < 1: return "underflow"
    max = x[0]
    x[0] = x[heap_size - 1]
    heap_size -= 1
    max_heapify(x, heap_size, 0)
```

It is same as last lines of 'heapsort', all we are doing is maintaining heap_size, rest everything is same

```
def increase_key(x, i, val):
    if x[i] > val: return "val should be larger"
    x[i] = key
    while i > 0 and x[i] > x[(i-1)//2]:
        x[i], x[(i-1)//2] = x[(i-1)//2], x[i]
        i = (i-1)//2
```

use (i-1) > 1

check if value > parent, and then swap and repeat till you get to root or parent is larger

To insert an element into the queue, you can increase the heap size and insert the element as the last index and then repeat 'increase_key' to set it's value from -∞ to the given key

## Find Min & Max simultaneously

## $k^{th}$ Largest element / Order statistics

① Bubble sort : the outer loop can run for k times . time = $O(nk)$

② Max heap : Build max heap in $O(n)$ and use extract max to get k maximum elements Time = $O(n + K \log n)$

③ Quickselect : this is most often used in practice. Worst time = $O(n^2)$, Average time = $O(n)$

Approach : - Make a random pivot and move it to the last
- Do Lumoto / Hoare partition
- Now you either need to go to left partition or right partition

```
def partition (x, start, end):
    pivot_index = random.randint (start, end)
    x[end], x[pivot_index] = x[pivot_index], x[end]

    left_end = start - 1
    for right_start in range (start, end):
        if x[right_start] < x[end]:
            left_end += 1
            x[left_end], x[right_start] = x[right_start], x[left_end]
    x[left_end+1], x[end] = x[end], x[left_end+1]
    return left_end + 1

def quickselect (x, k):
    start = 0
    end = len(x) - 1

    while start <= end :
        pivot_index = ~~lumoto~~ partition (x, start, end)
        if pivot_index == k: return x[k]
        if k < pivot_index :
            end = pivot_index - 1
        else :
            start = pivot_index + 1
```

→ Hoare partition was giving an error for some reason

→ think it should k-1 in all the conditions i.e. call the function with k-1

# Stacks

(Last in First out) i.e. Delete the most recently inserted element.

Applications
- Balancing of symbols
- infix to postfix
- Redo/undo features in text editors
- Forward/backward feature in web browsers
- Backtracking / Topological Sort / Strongly connected components
- Memory management to manage running processes

Stack can be implemented using array and linked list.

## Stack with Array

```
class Stack:
    def __init__(self, size):
        self.top = -1          → -1
        self.size = size - 1   → size - 1
        self.arr = [0 for _ in range(size)]

    def push(self, x):
        if self.top == self.size:
            return "Overflow"
        self.top += 1
        self.arr[self.top] = x

    def pop(self):
        if self.top == -1:
            return "Underflow"
        val = self.arr[self.top]
        self.top -= 1
        return val

    def isempty(self):
        if self.top == -1:
            return True
        return False
```

Pros: Memory is saved
Cons: Not dynamic

## Stack with linked list


```
class Node:
    def __init__(self, val, next):
        self.val = val
        self.next = ~~None~~ next

class Stack:
    def __init__(self):
        self.root = None

    def push(self, x):
        node = Node(x, self.root)
        self.root = node

    def pop(self):
        if self.root is None:
            return "Underflow"
        val = self.root.val
        self.root = self.root.next
        return val

    def isempty(self):
        if self.root is None:
            return True
        return False
```

```
class Node:
    def __init__(self, val):
        self.val = val
        self.next = next

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def enqueue(self, x):
        node = Node(x)
        if self.tail is None:
            self.head = node
            self.tail = node
        else:
            self.tail.next = node
            self.tail = node

    def dequeue(self):
        if self.head is None:
            return "Underflow"
        val = self.head.val
        self.head = self.head.next

        if self.head is None:
            self.tail = None
        return val

    def isempty():
        if self.head is None:
            return True
        return False
```

Need to handle two cases
- In case queue is empty, then you need to set both head / tail to the new node
- Otherwise it is same as Stack

If queue becomes empty, then you need to set 'tail' also to None, otherwise 'enqueue' would not work

Doubly linked list (without sentinal)

```
class Node:
    def __init__(self, val):
        self.val = val
        self.prev = None
        self.next = None

class list:
    def __init__(self):
        self.head = None

    def insert(self, x):
        new_node = Node(x)
        if self.head is None:
            self.head = new_node
        else:
            node = self.head
            while node.next is not None:
                node = node.next
            node.next = new_node
            new_node.prev = node

    def delete(self, x):
        if self.head is None:
            return "Not in list"
        if self.head.val == x:
            self.head = self.head.next
            if self.head is not None:        # these checks handle boundary conditions
                self.head.prev = None
        else:
            node = self.head
            while node is not None:
                if node.val == x:
                    if node.prev is not None:
                        node.prev.next = node.next
                    if node.next is not None:
                        node.next.prev = node.prev
                    return
                node = node.next
            return "not in list"
```

```
def search(self, x):
    node = self.head
    while node is not None:
        if node.val == x:
            return True
        node = node.next
    return False
```

```
def search(self, x):
    node = self.head
    if node is None:
        return False
    if node.val == x:
        return True
    node = node.next
    while node is not self.head:
        if node.val == x:
            return True
        node = node.next
    return False
```

Need to check for head node separately

```
def delete(self, x):
    if self.head is None:
        return
    if self.head.val == x:
        if self.head.next is self.head:
            self.head = None
        else:
            self.head.next.prev = self.head.prev
            self.head.prev.next = self.head.next
            self.head = self.head.next
        return
    node = self.head.next
    while node is not self.head:
        if node.val == x:
            node.next.prev = node.prev
            node.prev.next = node.next
            return
        node = node.next
    return "Not in list"
```

If list only contains head

Atleast list with 2 nodes

## Hash table

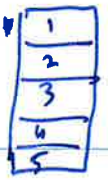
- Insert, Search, Delete.
- Used by compiler to maintain a Symbol table where keys of elements are arbitrary character strings corresponding to identifiers in the language
- For search worst time = O(n) but on average O(1)
- Uses concept of direct addressing. (where we ~~take advantage~~ can afford to allocate an array that has one position for every possible key)
- Instead of using the 'key' value directly as ~~address~~ index, the index is computed from the key

→ **Direct-address table**

If universe of keys = {0, 1, ..., m-1} where m is not too large, we can just use an array for this, where slot 'k' points to an element in the set with key 'k'

The main limitation of this approach is it takes a large amount of space.

e.g. Key range = [0, 5]

then 

→ 2, 12, 22, ... → we ~~map all~~ Store all the <key, value> pairs

Insertion = O(Search) (as we need to check if a key is already present)
Search = O(Search) (e.g. For index 2, we need to check if key 32 is present)
Deletion = O(Search) (Here we can either use a single/doubly linked list, the complexity would remain the same)

→ In CLRS, the bounds are not practical. Like for insertion = O(1), assuming there are no duplicate keys. For deletion, O(1) if we are given the node directly.

O(Search)
Hash table with 'm' slots that stores 'n' values (in total).
(α) Load factor = $\frac{n}{m}$ (avg. number of elements stored in a chain)

Worst case, all elements map to same slot = Θ(n)
For average case, we assume 'simple uniform hashing' i.e. all elements are equally likely to hash into any of the m slots
Average time = Θ(1 + α), where Θ(1) is time taken by hash function

A modification can be made to use balanced BST instead of linked list, then where n = number of elements in that slot
Search = O(log(n))
Delete = O(log(n))
Insert = O(n)

```
class Dict:
    def __init__(self, num_slots):
        self.m = num_slots
        self.arr = [list() for _ in range(num_slots)]

    def hash_function(self, k):
        return k % self.m

    def insert(self, k, v):
        hashed_k = self.hash_function(k)
        self.arr[hashed_k].insert(k, v)

    def search(self, k):
        hashed_k = self.hash_function(k)
        return self.arr[hashed_k].search(k)

    def delete(self, k):
        hashed_k = self.hash_function(k)
        return self.arr[hashed_k].delete(k)
```

Hash functions

Good hash function approximately satisfies 'simple uniform hashing' assumption (each key is equally likely to hash to any of the m slots, independently of where other key has hashed to).

- In practice, often we cannot know the distribution of the input keys or even if they are independent. So in general we develop hash functions using heuristic techniques that perform well.
- In case we know the distribution of keys, we can use this info in the design process. Eg. in case of a compiler's symbol table, we know the keys, so we can for closely related symbols such as 'pt', 'pts' we can minimize the chance they hash to the same slot (if they are frequent symbols)

Assuming keys are natural numbers (if they are not, find a way to convert them to natural numbers)

## 3. Universal hashing

Problem with fixed hash function is that an adversary can choose 'n' keys that map to the same slot, yielding an average retrieval time of $\Theta(n)$.

The only way to solve this issue is to choose hash function randomly in a way that is independent of the keys that are actually going to be stored. (called universal hashing)

In this, we are given a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m-1\}$.
Perfect hashing is an example of universal hashing.

We can design such a function as follows

p = prime (to hold range of keys)
$Z_p = \{0, 1, \ldots, p-1\}$
$Z_p^* = \{1, 2, \ldots, p-1\}$
m = number of slots ($p > m$)

$h_{ab}$, where $a \in Z_p^*$, $b \in Z_p$
$h_{ab}(k) = ((ak+b) \bmod p) \bmod m$ is universal

→ Collision handling (open addressing)

- all elements occupy the hash table (so you cannot store keys greater than the number of slots i.e. load factor cannot exceed 1)
- every entry in hash table either contains 'key' or 'NIL' (if the slot is empty)

To insert we get a 'probe sequence'

$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$

$h(k,0)$ → where to insert key for the first time
$h(k,1)$ → if $h(k,0)$ is occupied, where to insert it now
...

This is just a permutation of $\langle 0, 1, \ldots, m-1 \rangle$ so that every hash-table position is eventually considered as a slot for a new key as the table fills up.

- The insert procedure is thus very simple. We run a loop from $[0, m-1)$ and find a 'NIL' value where we can insert, otherwise return the table is full
- To search, we can use the same procedure till we find NIL. But there is a problem

## 3. Double hashing

- Best method for open addressing

$$h(k,i) = (h_1(k) + i\,h_2(k)) \bmod m$$

let ~~m = 2^p~~ m = prime, then $h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

where $m' = m - 1$ (slightly less than m)

Reason is for m = prime we want $h_2$ to return a +ve integer less than m

- Here $m^2$ probe sequences are used as two different values $h_1(k)$ & $h_2(k)$ determine initial position
- If m is a power of 2, then $h_2(k)$ should produce an odd number

Search time for open addressing = $O(\frac{1}{1-\alpha})$

### Separate chaining

- Chaining is simpler to implement
- Hash table never fills up
- less sensitive to hash function
  - used when it is unknown how many and how frequently keys may be inserted or deleted
- cache performance is poor as linked list is used (but can use vector)

### Open addressing

- requires more computation
- table may become full
- requires extra care to avoid clustering
- when frequency and number of keys is known
- Better cache performance

```
class Dict:
    def __init__(self, num_slots):
        self.m = num_slots
        self.arr = [None for _ in range(self.m)]
        self.deleted = [False for _ in range(self.m)]

    def hash_function(self, k):
        return math.floor(self.m * (k*A)%1)

    def h(self, k, i):
        return (self.hash_function(k) + i) % self.m
            (self.hash_function(k) + self.c1*i + self.c2*i*i) % self.m
```

Normal hash function division / Multiplication method

New hash function linear probing

Quadratic probing

→ Perfect hashing (FKS Hashing)

For excellent worst case performance when set of keys is static i.e. set of keys never change. This occurs for set of reserved words in a programming language or set of file names on a CD-ROM.

O(1) search in worst case

To design perfect hashing, two levels of hashing is used

- First level is same as hashing with chaining. Hash n keys into m slots using a hash function h carefully selected for a family of universal hash functions

$$h_{ab}(k) = ((ak+b) \bmod p) \bmod m$$

- Instead of linked list use a secondary hash table $S_j$ with hash function $h_j$. To guarantee O(1) search size of $S_j$ would be square of the number of keys that map to that slot.

Overall memory would still be O(n)

~~Size of primary hash table i.e. m = n. In this case memory would be same~~

$m = 2(n-1)$, then for each slot with number of entries = n, second hash table of size $n^2$ is constructed.

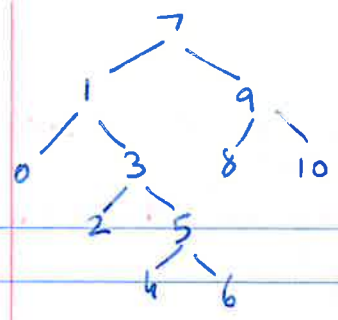
→ Binary Search Tree

- Support Search, Minimum, Maximum, Predecessor, Successor, Insert, Delete
- Can be used as dictionary and a priority queue.
- Height = O(log n) if randomly built. But it cannot be guaranteed
- Red-Black Trees have height = O(log n)
- B-Trees are good for maintaining databases on secondary storage.

Interview tips

- Use DP if it makes code easier to read
- Ask about the size of the input data and suggest that, if it's very large and the language you are using doesn't handle tail recursion, in practice you would use a loop instead.
- Python does not support tail recursion optimization

- Why use recursive function in some cases?
  - When working with recursive data structures and you don't know the size. Although it can often be done iteratively, it's way easier to do it recursively
  - For working with asynchronous code. If you have 10 jobs and you need to run them in specific order, but asynchronously (e.g. on the same server)




Preorder = 7, 1, 0, 3, 2, 5, 4, 6, 9, 8, 10
Inorder = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Postorder = 0, 2, 4, 6, 5, 3, 1, 8, 10, 9, 7

Look at the order of the nodes printed and based on that decide what traversal algorithm to use.

- If you need to explore roots before inspecting any leaves, use pre-order as roots often encountered before leaves
- If you need to explore leaves before any nodes, use post-order
- If you know tree has an inherent sequence in the nodes, and you want to flatten the tree back to its original sequence, use in-order traversal.

Postorder traversal used to delete the tree.

```
class Node:
    def __init__(self, val=None, left=None, right=None, parent=None):
        self.val = val
        self.left = left
        self.right = right
        self.parent = parent

class BST:
    def __init__(self):
        self.root = None

    def inorder_traversal(self, node):
        if node is not None:
            self.inorder_traversal(node.left)
            print(node.val)
            self.inorder_traversal(node.right)

    def preorder_traversal(self, node):
        ...

    def postorder_traversal(self, node):
        ...
```

you can create a helper function that calls traversal function by passing the root node. Also, if you want to do anything with the node value, you can create an array in the helper function and add elements to that work or you can directly work on the node val instead of printing it.

Here memory requirement is stack.
Time = O(n)
Space = O(h) height

For postorder, you can use Morris (by swapping left and right in the preorder implementation) but the result would be in reverse order. So you will have to initially store all the nodes in a list / stack and then reverse them. This is because postorder traversal cannot be made tail recursive.

## Searching

Time = O(h)
Space = O(1)

```
def search (self, val):
    node = self.root
    while node is not None and val != node.val:
        if val < node.val:
            node = node.left
        else:
            node = node.right
    return node
```

This checks for value but you can do this for a node also.

None is returned if not found, else the node is returned.

## Minimum / Maximum

Minimum - left most node
Maximum - right most node

Time = O(h), Space = O(1)

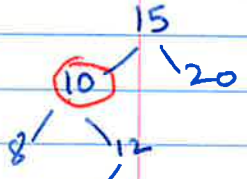
```
def minimum (self, node):
    while node.left is not None:
        node = node.left
    return node.val
```

```
def maximum (self, node):
    while node.right is not None:
        node = node.right
    return node.val
```

The only problem with above code is when the 'input' node is None i.e. we try to find minimum / maximum of an empty tree. This can be checked by an if condition before while loop.

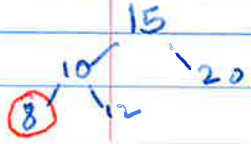
## Successor

Case 1: There is a right subtree




To find successor, we are doing inorder traversal. So for this case, the node that would be visited after 10 in the inorder traversal is the ans. Which would be the minimum of it's right node.

Case 2: No subtree, but left subchild




In this case, after 8 it's parent would be visited. So 10 is the ans.

```
if node.right is not None:
    return self.minimum(node.right)
```

Case 1

```
node = self.root
successor = None

while node is not None:
    if val < node.val:
        successor = node
        node = node.left
    elif val > node.val:
        node = node.right
    else:
        if successor is None:
            return "No successor of maximum"
        return successor.val
```

Store successor if going left

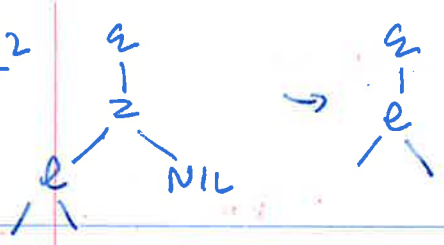
Predecessor

```
def predecessor(self, val):
    node = self.search(val)
    if node is None:
        return "val not in tree"

    if node.left is not None:
        return self.maximum(node.left)

    parent = node.parent
    while parent is not None and node == parent.left:
        node = parent
        parent = parent.parent

    if parent is None:
        return "Minimum value"
    return parent.val
```

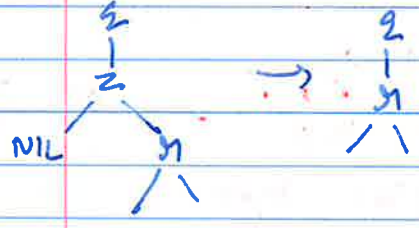
Code is same as successor, except right is replaced with left

Predecessor without parent logic is also the same as successor. The only change is 'predecessor' is updated in 'elif val > node.val' instead of 'if val < node.val'.

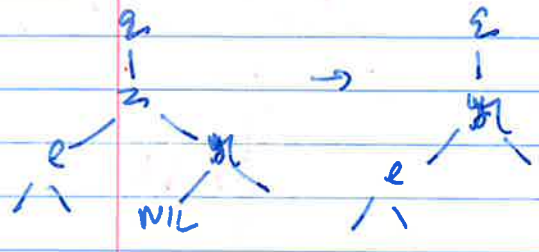
Case 2 (Single child on left)




(Single child on right)

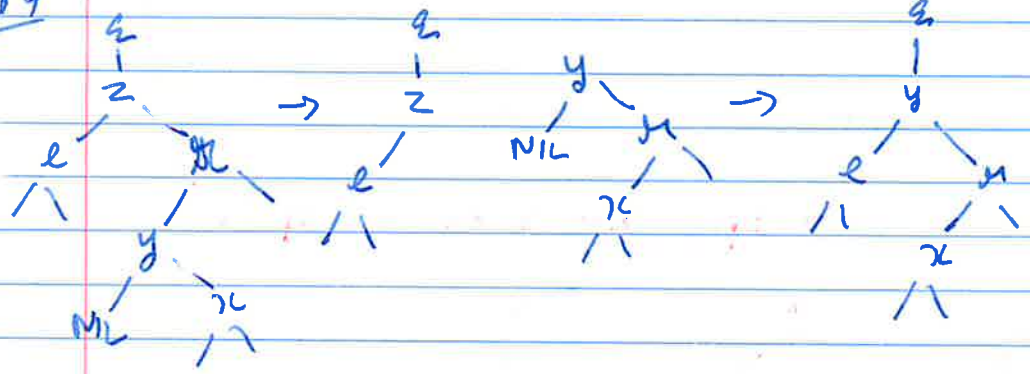



Case 3



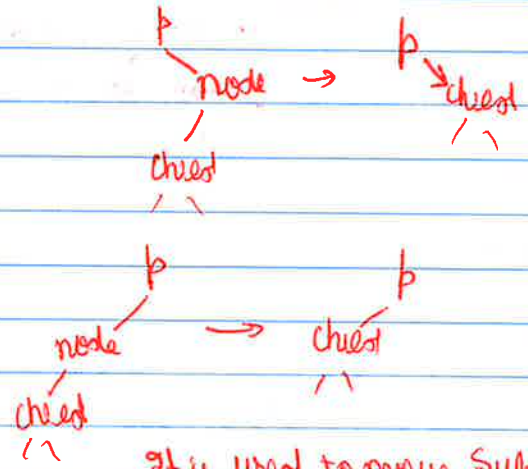

(two child, and on right y is the successor, so we can directly replace it)

Case 4




(two child, and successor is not ~~on~~ ~~right~~ right child. In this case we move y to z and ~~the~~ and ~~the~~ right becomes left on x y's

```
def transplant (self, node, child):
    if node.parent is None:
        self.root = child
    elif node == node.parent.left:
        node.parent.left = child
    else:
        node.parent.right = child

    if child is not None:
        child.parent = node.parent
```




It is used to move subtrees as shown (i.e. move child one level up)

## Self balancing binary search tree

node-based BST that automatically keeps its height in the face of insertion and deletion.

Height = $O(\log n)$

also called symmetric binary B-trees

It consists of AVL Tree, Red-Black Tree, B-tree, (and many more)
↳ generalization of 2-3 trees

These BST's can be used to construct and maintain ordered lists (like priority queue), associative arrays (key-value pairs, key only appears once).

Advantage of Self-balancing BST over hash tables is that they allow fast (indeed, asymptotically optimal) enumeration of the items in 'key order', which hash tables do not provide.

Disadvantage is that the lookup algorithm gets more complicated when there may be multiple items with the same key.

BST have better worst-case lookup performance $O(\log n)$ vs $O(n)$

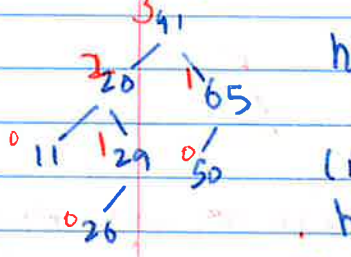
BST have worse average-case performance $O(\log n)$ vs $O(1)$

AVL-trees are more strictly balanced (faster retrieval i.e. search), while RB trees don't have to maintain perfect balance (meaning insertion / deletion are faster)

B-trees seem to be useful when you have a ton of items to the point where they can't be allocated to memory simultaneously.

Depth = distance from the node to the root

Height = length of longest path from it down to a leaf



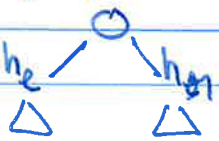

height(node) = max(height(node.left), height(node.right)) + 1

(the ~~base~~)

(if no left-child exists, then the 'None' node can be given a height of -1 but for my implementation this would not work, as I am using 'None' for left-child rather than creating a new 'None node'.

## → AVL Trees

require height of left and right children of every node to differ by at most ±1




$|h_e - h_{or}| \leq 1$

```
node.right = right.left
if node.right is not None:
    node.right.parent = node
right.left = node
node.height = self.get_height_of_node(node)
right.height = self.get_height_of_node(right)
```

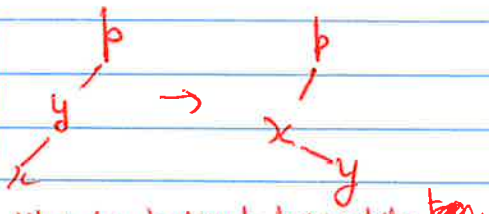
Main rotation

For AVL, we need to maintain height of every node also (see next section)

For right rotate, code is same just swap ~~left~~ with left

```
def right_rotate(self, node):
    if node is None:
        return "None Node"
    if node.left is None:
        return "Left child cannot be None"

    if node.parent is not None:
        if node == node.parent.left:
            node.parent.left = node.left
        else:
            node.parent.right = node.left
    else:
        self.root = node.left
    node.left.parent = node.parent
    node.parent = node.left

    left = node.left

    node.left = left.right
    node.left.parent = node
    left.right = node
    node.height = self.get_height_of_node(node)
    left.height = self.get_height_of_node(left)
```

Error checking




Update parent pointers too

Update parent of y & x

Main rotation

Update height of modified nodes

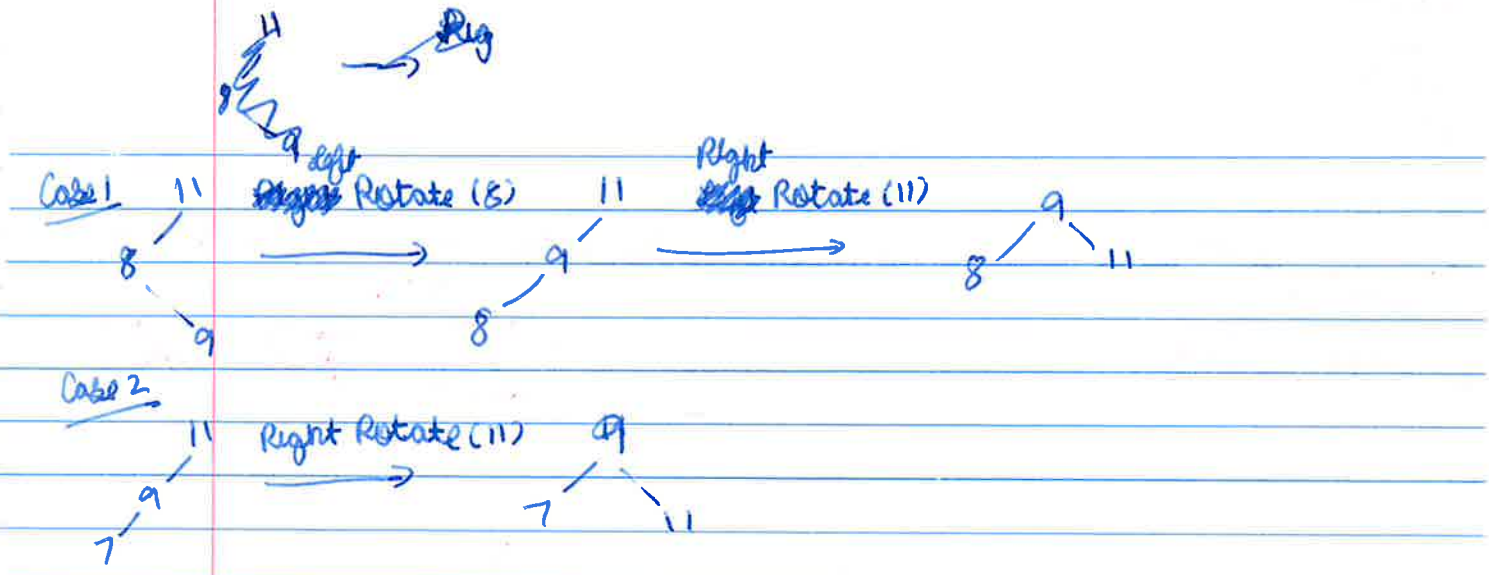
~~Insertion~~ Height

Step 1: Insert like a normal BST

Step 2: Fix the AVL property using rotations

Before insertion, we also need to store the height of all nodes. You can do this by augmenting the Node class ~~(or in python just dynamically creating a new value)~~ i.e. add 'self.height = 0')

For insertion, we have 2 cases

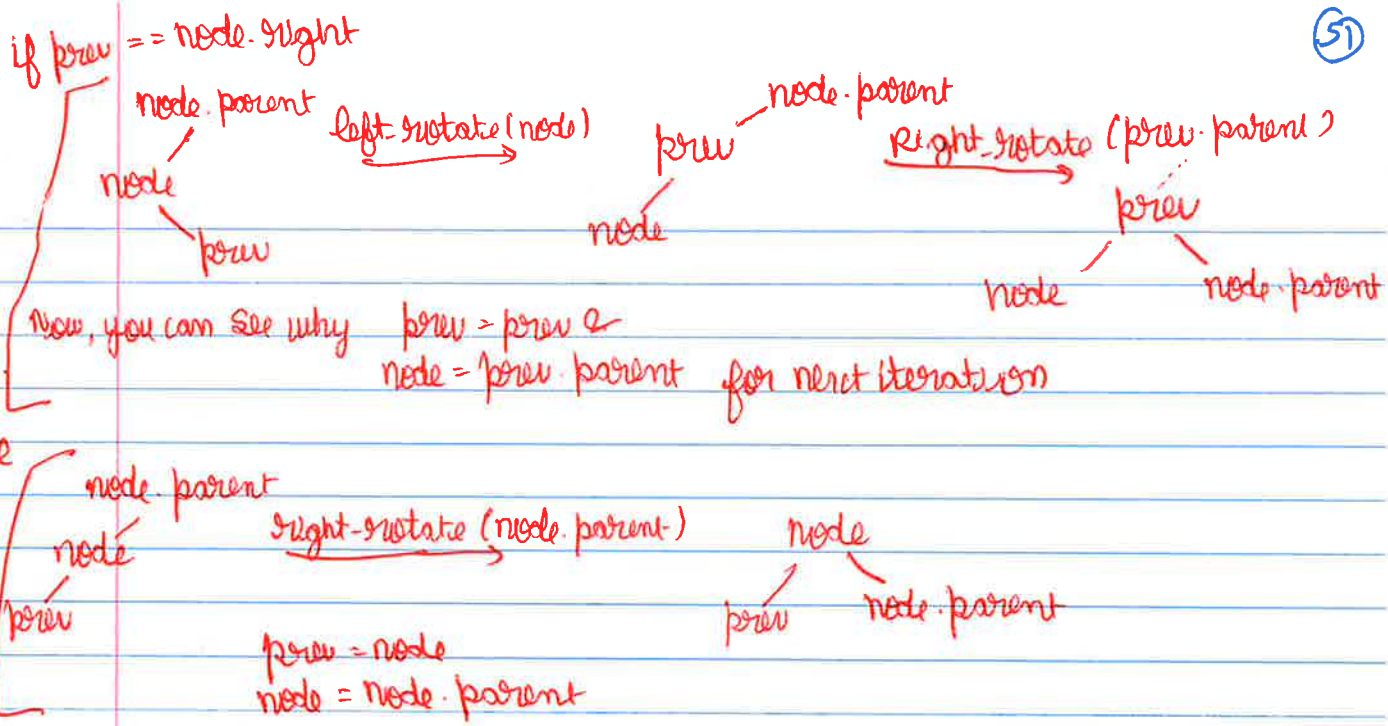



And the same follows for Right-Left Rotate, Left Rotate

```
def insert(self, val):
    if self.root is None:
        self.root = node(val)
        return

    parent = None
    node = self.root
    while node is not None:
        parent = node
        if val < node.val:
            node = node.left
        else:
            node = node.right

    new_node = Node(val)
    new_node.parent = parent
    if val < parent.val:
        parent.val = new_node
    else:
        parent.right = new_node
```

Base case of empty tree. Here we do not have to set 'self.root.height = 0' as we modified the 'class Node' to include this default value

Same code as normal BST




Deletion

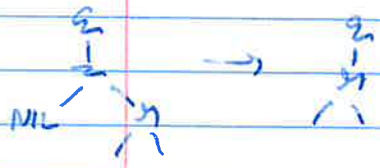
Implementation is same as the regular BST. But now after 'transplant' we have to rebalance the tree (like insert)

Case1

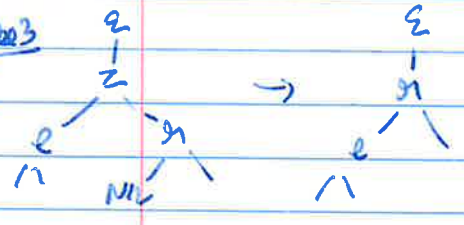

(rebalance from z to root)

Case2

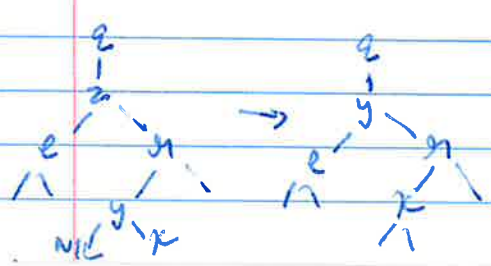

(rebalance from z to root)




Case3




(rebalance from x to root)




z. parent

(rebalance from z to root)

```
prev = child
node = parent
node.height = self.get_height_of_node(node)
balance_factor = self.get_balance_factor(node)

if balance_factor > 1:
    left_child = node.left
    self.right_rotate(node)
    prev = node
    node = left_child
elif balance_factor < -1:
    right_child = node.right
    self.left_rotate(node)
    prev = node
    node = right child
```

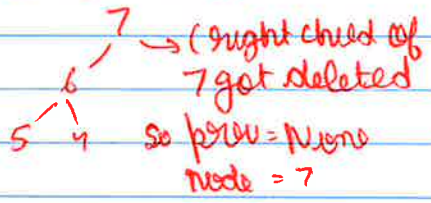
In the 'insert' code we worked with 3 nodes

node.parent

node

prev

Here, ~~due to delete~~ we may stand with 2 node i.e.




→ (right child of 7 got deleted so prev = None node = 7

Due to this reason, we need to have this code

```
while node is not None and node.parent is not None:
    node.height = self.get_height_of_node(node)
    parent_balance_factor = self.get_balance_factor(node.parent)

    if parent.balance_factor > 1:
        if prev is None or prev == node.left:
            self.right_rotate(node.parent)
            prev = node
            node = node.parent
        else:
            self.left_rotate(node)
            self.right_rotate(prev.parent)
            node = prev.parent
    elif parent_balance_factor < -1:
        if prev is None or prev == node.right:
            self.left_rotate(node.parent)
            prev = node
            node = node.parent
        else:
            self.right_rotate(node)
            self.left_rotate(prev.parent)
            node = prev.parent
```

Same logic as before, only addition is

if prev is None:

(In this case we can only do one rotation case)

```
print(' ' + str(root.val)) → single space
if prev:
    prev.str = prev.str
trunk.str = ' ' → 3 spaces
printTree(root.left, trunk, False)
```

## → Red-Black Trees

- Every node has an extra bit, specifying its color has RED or BLACK
- Root is BLACK
- NIL (leaves) are BLACK → i.e. default color of a node should be BLACK
- By constraining the node colors on a path from root to leaf, RB trees ensure that no such path is more than twice as long as the other (so tree is approximately balanced)

Internal nodes = Nodes with a key value
External nodes = NIL nodes (i.e. leaves)

~~Red-Black Tree~~

Red-Black property

1. Every node is either red or black
2. Root is black
3. Every leaf (NIL) is black
4. If a node is red, then both the children are black
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

To save space for leaf nodes, you can specify a sentinal node with val = None, left, right, parent = None, color = Black
The root's parent can also take this sentinal node.

Black height of node = Number of black nodes on any simple path from node's children to leaves (i.e. do not count the 'node' during the calculation).
By property 5, all paths should have same number of black nodes
Black height of RB tree = Black height of root
Height of RB tree $\leq 2 \log_2(n+1)$ → for AVL it was $1.44 \log_2 n$

- whenever you do any operation (I don't know if it includes insert, but search for sure), you move that node to the root by using rotations (called Splaying)

Advantages
- It is self-optimizing, in that frequently accessed nodes will move nearer to the root where they can be accessed more ~~accurately~~ quickly.
- It does not include self-balancing, so in worst case it can be O(n)

## Common Operations

```
class Node:
    def __init__(self, val=None, ~~color="black"~~ black=True):
        self.val = val
        self.~~color~~ left = None
        self.right = None
        self.parent = None
        self.~~color~~ black = ~~color~~ black
```

```
class RBTree:
    def __init__(self):
        self.root = None
        self.sentinal = Node()

    def search(self, val):
        node = self.root
        if node is None:                    # Empty tree
            return node

        while node.val is not None and      # only change
              val != node.val:
            if val < node.~~left~~ val:
                node = node.left
            else:
                node = node.right
        return node

    def minimum(self, node):
        while node.left.val is not None:
            node = node.~~right~~ left
        return node
```

```
    def maximum(self, node):
        while node.right.val is not None:
            node = node.~~left~~ right
        return node

    def inorder_traversal(self, node):
        if node is not None and node.val
                is not None:
            self.inorder_traversal(node.left)
            print(node.val)
            self.inorder_traversal(node.right)
```

```
    def successor(self, node):
        if node is None or node.val is None:
            return "Node not found in tree"

        if node.right.val is not None:
            return self.minimum(node.right)
        Rest is Same

    def predecessor(self, node):
        if node is None or node.val is None:
            return "Node not found in tree"

        if node.left.val is not None:
            return self.maximum(node.left)
        Rest is Same
```

* All the operations are done on blocks i.e. you write a block or read a block.

Consider a database, with columns of size (10, 50, 10, 8, 50) i.e. 128 bytes are required to store a row.

$\frac{512 \text{ bytes}}{128 \text{ bytes}} = 4$ i.e. one block can store 4 rows of the database table.

Suppose we have 100 rows, so we need 25 blocks to store the table.

If you want to search for a query, you would at most need to access 25 blocks as the row can be present anywhere.
To reduce this search time we will maintain an index. In index we will store a key and a pointer to row

Index

| id | pointer |
|---|---|
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |

This index is also stored on disk. Index has 2 cols
- id (assume 10 bytes)
- pointer (assume 6 bytes)

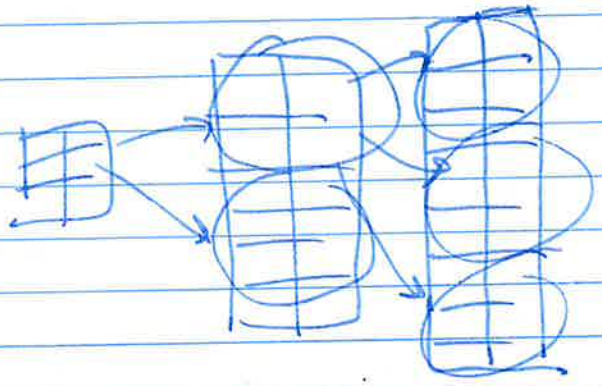
= 16 bytes

So every block can have $\frac{512 \text{ bytes}}{16 \text{ bytes}} = 32$. (rows of index table)

$\therefore$ Total blocks required $= \frac{100}{32} = 3.2 = 4$

So we reduced 25 blocks to 4 blocks.

Now, we generalize the above idea. What if the index table gets large, we maintain another index (where every pointer points to a block)



i.e Search, Insert, Delete, Min, Max, Successor, Predecessor in $O(\log n)$ time. (61)

→ Matrix operations / Solve a system of linear equations

To solve a system of linear equations

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1$$
$$\vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n$$

Assuming n equations and n unknowns,

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & & & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_{12} \\ x_{13} \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

i.e. $Ax = b$

$x = A^{-1}b$ (assuming A is non-singular)

If num of equations < number unknowns, then the system is undetermined (with infinite solutions)

We solve the above matrix by LUP decomposition

$PA = LU$ P = permutation matrix, L = lower triangle

$Ax = b$

$PAx = Pb$

$LUx = Pb$

Let $y = Ux$

$Ly = Pb$

Now we solve it using "forward substitution" and then use the value of y to get x, using $Ux = y$

Forward substitution takes $\theta(n^2)$ time

$$y_1 = b_{1}$$
$$l_{21}y_1 + y_2 = b_{2}$$
$$\vdots$$
$$l_{n1}y_1 + l_{n2}y_2 + \dots + y_n = b_{n}$$

Now we start substituting from $y_1$ to $y_n$ (i.e. forward direction)

Then we solve for x using "backward substitution" as U is upper-triangle

→ Polynomials and FFT

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

Addition and subtraction of polynomials is easy and can be done in $O(n)$

Multiplication can be done in $O(n^2)$. This can be reduced to $O(n \lg n)$ by using FFT.

$$C(x) = A(x)\, B(x) \quad \text{degree of } C(x) = 2n-1$$

$$C(x) = \sum_{j=0}^{2n-2} c_j x^j \quad \text{where} \quad c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

Two ways to represent a polynomial

- Coefficient representation: Here coefficients $a = (a_0, a_1, \dots, a_{n-1})$. This is useful when evaluating a polynomial at a given point $x_0$, using Horner's rule

$$A(x_0) = a_0 + x_0 (a_1 + x_0 (a_2 + \dots + x_0 (a_{n-2} + x_0 (a_{n-1})) \dots))$$

It is also useful for addition, as we just add the two coefficient vectors.

- Point-value representation: It is a set of n distinct points

$$\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\} \text{ where } y_k = A(x_k)$$

We can convert coefficient representation to point value repn by choosing n different points and evaluating them using Horner's rule to get a total of $O(n^2)$ time. This can be reduced to $O(n \lg n)$ by cleverly choosing the points.

Interpolation is used to convert point-value repn to coefficient repn

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

and can be done in $O(n^2)$ time

Addition is still $O(n)$ in point value repn as

$$A(x) = \{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$$

$$B(x) = \{(x_0, y_0'), (x_1, y_1'), \dots, (x_{n-1}, y_{n-1}')\}$$

$$C(x) = A(x) + B(x)$$

$$= \{(x_0, y_0 + y_0'), (x_1, y_1 + y_1'), \dots, (x_{n-1}, y_{n-1} + y_{n-1}')\}$$

To multiply two polynomials in this repn takes $O(n)$ time. First we use $2n$ points instead of n as the result of multiplication has degree $2n-1$

∴ For the exponential algorithms we will write polynomial time nondeterministic algos and hope somebody in future can make them deterministic.

We define two classes
- P : set of polynomial time deterministic algorithms
- NP : set of polynomial time non deterministic algorithms


E.g. In the past, maybe merge sort was not known, but then we found it and put it into 'P'

We said we want to make all the algorithms similar. To relate all the problems, we define a base problem called 'satisfiability problem'

CNF - Satisfiability

$x_i = \{x_1, x_2, x_3\}$

$CNF = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$

$C_1$ $C_2$

Prob: For what values of $x_1, x_2, x_3$ is CNF true.

Brute-force is $2^n$, where we try all the boolean combinations of $x_1, x_2, x_3$

E.g. 0/1 - Knapsack, we can either choose the first item, or not & so on which is same as CNF where for every variable we have 2 options. Thus, we reduced 0/1 - Knapsack to CNF - Satisfiability.

Reduction

We can take some interpretation of satisfiability problem and reduce it to 0/1 knap

Sat $\propto$ 0/1 Knapsack

$I_1$ $I_2$

Now if $I_1$ is solved in polynomial, then the same algo can be used to solve $I_2$ in polynomial time and vice versa.

The conversion from $I_1$ to $I_2$ is taking polynomial time also

Assume Satisfiability is NP-hard, then all the algo's which are reduced from 'Sat' are also NP-hard

i.e. if Sat $\propto L$, then L is also NP-hard

Performance ratio of approx. algo.
- If each potential soln has positive cost, then the near-optimal solution will have the maximum or minimum cost, depending upon the problem

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$$

$C$ = approx algo
$C^*$ = optimal solution
$P(n)$ = approx ratio

Some algos may just provide an approximate answer, while other algos might provide better solution as the number of iterations increase.

① Vertex Cover problem
In an undirected graph, a 'vertex cover' is a subset of vertices such that for every edge $(u, v)$ of the graph either $u$ is in vertex cover or $v$

e.g. 

vertex cover = {2, 4} or {0, 4}

Brute-force: Consider subset of all vertices e.g. with 3 vertices sets are {0,1,2}, {0,1,3}, {0,2,3}, {1,2,3} and then check each set for a valid solution and print the subset with minimum number of vertices.

Approx algo:

```
res = {}
while set of edges != 0:
    pick an arbitrary edge
    add the edge in 'res'
    remove all edges incident on the nodes of selected edge
return res
```

This algo never produces a vertex cover whose size is more than twice the minimum possible vertex cover size.
Time = O(V + E)

```
visited = [False] * num_nodes
for u in range(num_nodes):
    if not visited[u]:          → if edge not visited then source is not visited
        for v in adjacent_nodes(u):
            if not visited[v]:  → pick the first non-visited node
                visited[u], visited[v] = True, True
                break
```

Print all visited nodes as ans

Approx algo: If triangle inequality is valid i.e.


$(u,v) + (v+w) \geq w$

then we can use this algo

- Choose a vertex at most
- Compute minimum spanning tree from 'root' using PRIM
- Do a preorder walk of the resulting tree and the result is the ans

The cost would not be more than twice of MST's weight. Time = $O(V^2)$

If triangle inequality is not valid then no good approximate tours can be found in polynomial time.

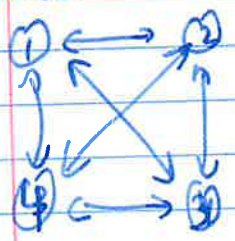
Dynamic programming soln in $O(n^2 2^n)$

Intro: To get the lower bound of cost, consider the weight matrix

$$\begin{bmatrix} \infty & 10 & 17 & 3 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

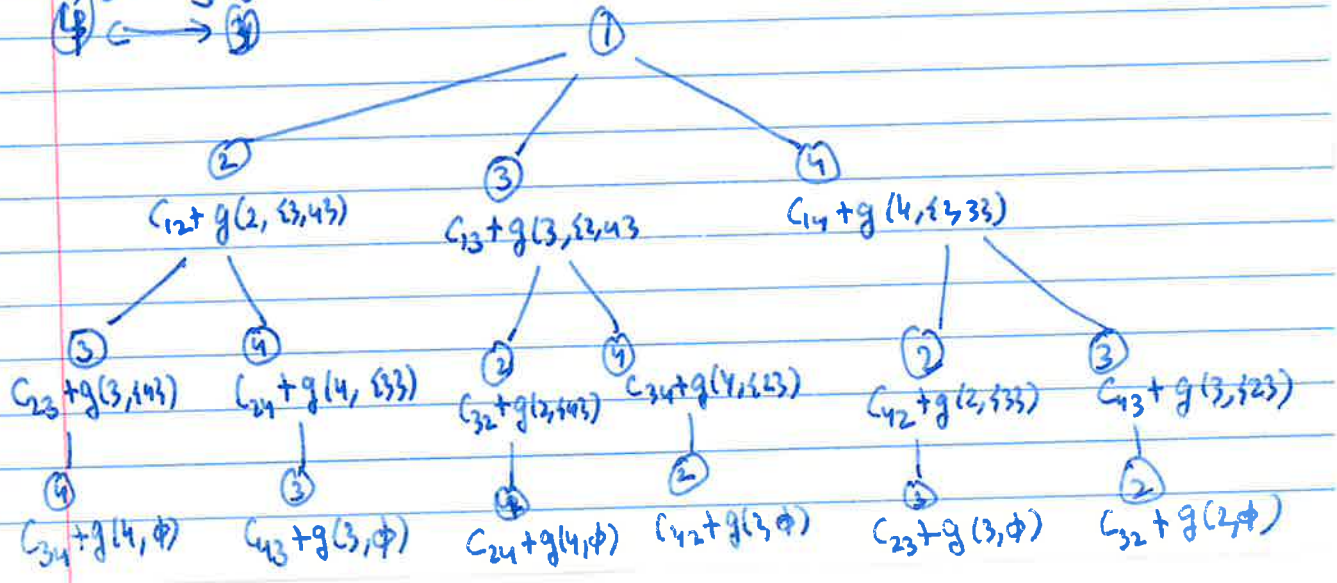
- Find the minimum in each row
- Substract the minimum from each row
- Add all the minimum's
- Repeat for columns and add the min's to that of row's and you got the lower bound

DP formula




$$g(1, \{2,3,4\}) = \min_{k \in \{2,3,4\}} \{ C_{1k} + g(k, \{2,3,4\} - \{k\}) \}$$

generating the recursive tree of above



Compute the reduced cost matrix for each neighboring node

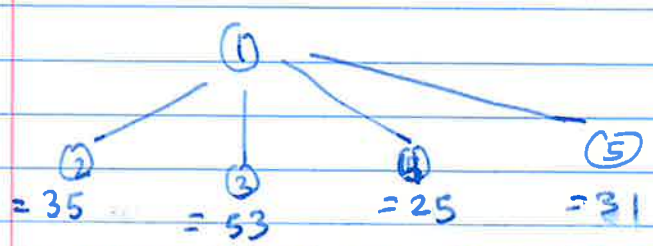
For ②, we make the row 1 and col 2 infinity in the reduced cost matrix of ① and generate a new reduced matrix same as above.

If the edge is ① → ② then we make row 1 as inf and col 2 as inf + [2,1] = ∞ (as we cannot go from 2 to 1)

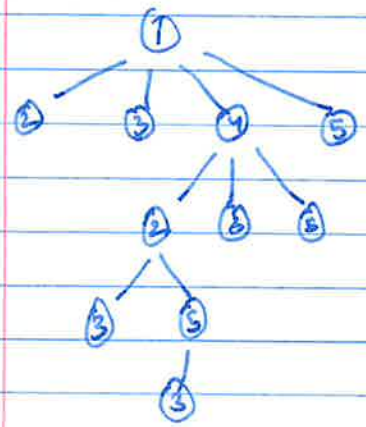
$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix} \longrightarrow \begin{bmatrix} \quad \end{bmatrix}$$

The new cost = cost(1,2) + prev_cost + curr_cost

- cost(1,2): get this cost from the reduced matrix of ① i.e. [1,2] = 10
- prev_cost: cost to reduce ①
- curr_cost: cost to reduce ②




Now only expand the minimum one and keep repeating.




Time = 2n worst case same as brute force, as we may never get a chance to prune a node if cost is same.

# ⇒ Strings

## → String Searching Algorithms / String matching Algorithms

Find where 'pattern' is in 'text'
↓ needle ↓ haystack

m = pattern length
n = text length

### → Naive algorithm (Single pattern)

Time = O(mn), Space = O(1), Preprocessing = None

Brute-force method where you ~~can~~ check for the pattern starting at each index.

```
for i in range(len(text) - len(pattern) + 1):
    if text[i : i + len(pattern)] == pattern:
        return i
return -1
```

### → Rabin-Karp algorithm (Single pattern)

Better for multiple pattern search. Single pattern perf is poor as compared to other options

Preprocessing = O(m), Time = O(mn) average = O(n), Space = O(1)

It works by computing a hash of the pattern and then performs approximate check for each position, and then only perform an exact comparison ~~when~~ that pass that approximate check.

It is used for plagarism detection. In this we want to find text overlap between two documents i.e. 'text' and a set of 'patterns'. KMP cannot do anything here, as we will have to apply it for each word separately. But in Rabin-Karp we can ~~put these~~ compute hash of each word and put those in a set for easy query.
(In this case Rhio-Corasick makes more sense, I think)

Steps of the algo
- Compute hash of the pattern
- Start at index 0 and find hash of ~~for~~ characters of length equal to pattern and repeat
- When match is found, do an exact comparison to check if the string is same as pattern

Hash function should be 'rolling hash' i.e. its value can be quickly updated from each position of the text to the next. (as recomputing hash from scratch would be slow).

⇒ Knuth-Morris-Bratt algorithm // KMP (Single pattern)

Preprocessing = $\Theta(m)$, time = $\Theta(n)$, space = $\Theta(m)$

The main idea is we can use previous mismatch to find how many characters we should skip.

E.g. text = a b c d a b c a b c d f

↑ start directly here

pattern = a b c d f

The naive algorithm would first match a b c d and then there would be a mismatch. But now we can start directly from this index as there is no way for pattern to be before that.

KMP

Given the pattern = a b c d a b c

Find the prefix = a, ab, abc, abcd, abcda, abcdab, ~~abcdabc~~

Find the suffix = c, bc, abc, dabc, cdabc, bcdabc, ~~abcdabc~~

Now we want to find if there are any common prefix & suffix

In our example it is 'a b c'

pattern = a b c d a b c

To prepare the table (prefix table)

P₁ : a b c d a b e a b f

0 0 0 0 1 2 0 1 2 0

Start from left & see where the first 'a' is. It occurs after 'd' so give it the original index (starting from 1). Now b also matches with starting 'a b' So give it index 2 and repeat

P₂ : a a b c a d a a b e

0 1 0 0 1 0 1 2 3 0

Eg:

```
        else:
            prefix_end_index = prefix_table[prefix_end_index-1]

    pat_index, text_index = 0, 0
    while text_index < len(text):
        if pattern[pat_index] == text[text_index]:
            pat_index += 1
            text_index += 1

            if pat_index == len(pattern):
                return text_index - pat_index      # in case you want to return all the items
        else:
            if pat_index == 0:
                text_index += 1
            else:
                pat_index = prefix_table[pat_index-1]
```

pat_index = prefix_table[pat_index-1]

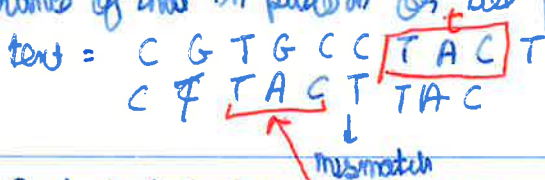
## Real-time KMP

- One thing to notice in the above implementation is that the when a mismatch occurs, we might not update the 'text-index'. This can be optimized using real-time KMP
- In this we build a table for each alphabet
  i.e. Preprocessing = $O(k \cdot M)$, Time = $\theta(n)$, Space = $O(k \cdot m)$
  if input space is all lowercase characters then $k = 26$
- I don't know how it works, but it saves only constant time, while significantly increasing the memory.

## Lexicographically minimal string rotation

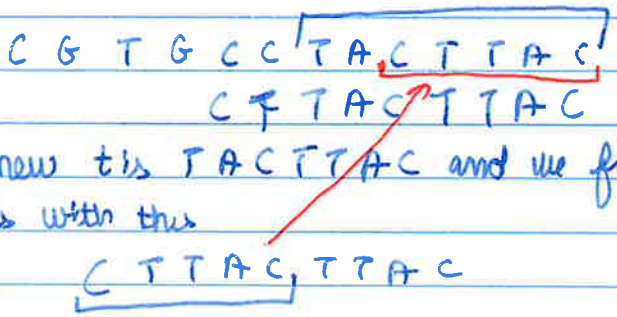
- KMP can be used to find this.
- E.g. "bbaaccaadd". The lowest lexicographical order is "aaccaaddbb" obtained by rotating left 2 times
- Used to normalize strings, if strings contain isomorphic structures then you can check for equality
- Concatenating the strings is a useful technique when dealing with circular strings to avoid modular arithmetic

- let t = substring matched in text before mismatch, skip until the first occurance of that in pattern or till the pattern moves past t




good suffix heuristic

find t in pattern and move it




Now our new t is T A C T T A C and we find a suffix in pattern that aligns with this



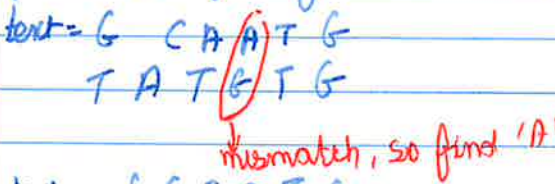

~~At every step~~

So in short, you preprocess the pattern

- Create 2 arrays (one for 'bad character heuristic', one for 'good suffix heuristic')
- at each step, slide the pattern by maximum of the two heuristics

Bad character heuristic

- If mismatch is at char 'A' in text then find the next occurrence of 'A' in pattern and if not found slide the whole pattern




Worst case for this is O(mn), when all characters of the text & pattern are same
text = AAAA, pat = AA

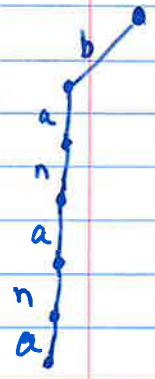
Best case = O(n/m) when all chars in text & pattern are different

```
NUM_CHARS = 26
bad_char = [-1] * NUM_CHARS
for i in range (len (pattern)):
    bad_char [ord (pattern [i]) - ord ('a')] = i
```

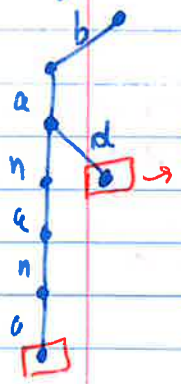
→ Trie

Also called prefix tree is a type of k-ary search tree used for locating specific keys from within a set.

We start from empty node 'root'. If first string is 'banana' then we add it as

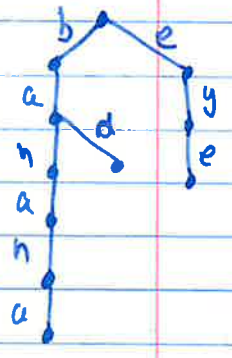



Next if we get the word 'bad', we start from root and ~~so if we~~ continue till the starting letters are already in the tree




Terminal node to represent end of string

Ily if we add 'eye




This is useful to check if ~~a suffix~~ there is any

This is useful to check suffix i.e. among your words {banana, bad, eye}, is there any word that has ~~suffix~~ prefix 'bab'

We start from root and try to ~~the~~ traverse 'bab' and then we can know if the word exist or not

→ This is used for suggesting words when you are typing in your phone. So when the user is typing, you keep traversing the 'Trie' and at each node you suggest words, with the entered suffix. Then you can use heuristics to decide which word to show (like frequency, length)

```
for c in word:
    index = ord(c) - ord('a')
    if node.children[index] is None:
        return "word not present"
    parent = node
    node = node.children[index]

if not node.word_end:
    return "Word not present"

can_delete_node = True
for child in node.children:
    if child is not None:
        can_delete_node = False
        break

if can_delete:

if can_delete_node:
    parent.children[index] = None
else:
    node.word_end = False
```

if there are children node, then we only need to set 'word_end' to False, otherwise delete the whole node

Applications of Trie

① Count number of strings with given prefix
→ Change 'word_end' to 'prefix_count' and during 'insert' update 'node.prefix_count' in each step of 'for loop'

② Autocomplete
→ Traverse the prefix and then do DFS to find the words to suggest (may be top-k alphabetical words)
(implementation later)

③ Spell Checker
→ An example of "approximation string matching". (implementation later)

④ Longest prefix matching
→ Given a word, find the longest ? Used by routers in IP networking to select an entry from a routing table

## Word game / Boggle

Given an m×n board of characters and a list of dictionary words, return all the words on the board.

Rules
- Can consider 4-neighbor or 8-neighbor (in this example 4-neighbor is used)
- Same cell cannot be used again for the same word

e.g.

| | | | |
|---|---|---|---|
| o | a | a | |
| e | t | a | . |
| i | h | k | |

you can form 'taaa' like this

The problem is a simplified 'crossword puzzle game'

```
true = {}
for word in words:
    node = true
    for c in word:
        if c not in node:
            node[c] = {}
        node = node[c]
    node["WORD_END"] = word

R, C = len(board), len(board[0])
self.solutions = []
self.board = board
self.R, self.C = R, C

for r in range(R):
    for c in range(C):
        self.dfs(r, c, self.true)
```

Alternate true implementation using dict. The code logic is same as True. Insert

~~We are not storing "char"~~
The "char" is actually being stored as the key of dict

We need to mark end node as, so as to indicate end of word. If we used a boolean here, then in the "dfs" function we would have to pass 'prefix' string. This trick of storing the entire word at the end node in a special key avoids that and provides some speedup as the call stack has less parameters now

Backtracking logic where we try all the positions

```
def dfs(self, row_ind, col_ind, node):
    if row_ind < 0 or row_ind >= self.R or col_ind < 0 or col_ind >= self.C:
        return
    letter = self.board[row_ind][col_ind]
    if letter not in node:
        return
    child_node = node[letter]
```

As we are considering the 4 neighbors, this trick avoids writing multiple if conditions when calling this func

Same as backtracking. Check if position is valid before continuing

the above time complexity does not factor into the pruning of leaf nodes.

Space : $O(N)$, $N$ = total number of letters in the dict, as only true space taken by recursion stack not factored here

~~Prefix and Suffix~~

Sort an array of strings → Pretty standard code

```
class Node:
    def __init__(self):
        self.child = [None for _ in range(26)]
        self.word_end = None
class Trie:
    def __init__(self):
        self.root = Node()
    def insert(self, word):
        node = self.root
        for c in word:
            index = ord(c) - ord('a')
            if node.child[index] is None:
                node.child[index] = Node()
            node = node.child[index]
        node.word_end = word

true = Trie()
for word in words:
    trie.insert(word)

out = []
def dfs(node):
    if node is None:
        return
    if node.word_end is not None:
        out.append(node.word_end)
        node.word_end = None
    for i in range(26):
        dfs(node.child[i])
dfs(trie.root)  <- to start recursion
```

node ["NUM-END"] = nums[i] → Store the num in the last node. Using this we can easily get the number and not have to store all the intermediate results.

Also, duplicate values would not be thrown away. For this reason, initialize the answer = 0, as $x \wedge x = 0$

```
out = 0
for i, bits in enumerate(num_bits):
    node = trie
    for bit in bits:
        if bit == 1: opposite_bit = 0
        else:        opposite_bit = 1

        if opposite bit in node:
            node = node[opposite_bit]
        else:
            node = node[bit]
    x = nums[i]
    y = node["NUMS_END"]
    out = max(out, x ^ y)
return out
```

## Autocomplete

① The main autocomplete system used by many popular websites is based on ElasticSearch (open-source for Node.js applications (probably))
There are multiple ways to implement an autocomplete feature
- Index time: just prefix completion or infix comp. The trie implementation after this section does that.
- Query time: All the words are separately searched for in the data (using KMP like algo). Thus the results contain the data where the tokens can be separated.

② In a google interview, the question was to autocomplete using only the last 7 days of data.

```
        node = node.child[index]

    if not prefix_exist:
        print("No word found")
    else:
        dfs(node)


def dfs(node):
    if node is None:
        return
    if node.word_end is not None:
        out.append(node.word_end)
        node.word_end = None
    for i in range(26):
        if node.child[i] is not None:
            dfs(node.child[i])
```

Depending upon your problem, filter the results. This implementation by default produces a sorted output. But if lexicographic order is not desired, a dict can be used for this implementation to save space.

## Aho-Corasick algorithm

Preprocessing = $\Theta(m)$, Time = $\Theta(n+o)$, Space = $\Theta(m)$

$m$ = total length of pattern (if 4 words of 10 chars each, then $m = 40$)

$n$ = length of text

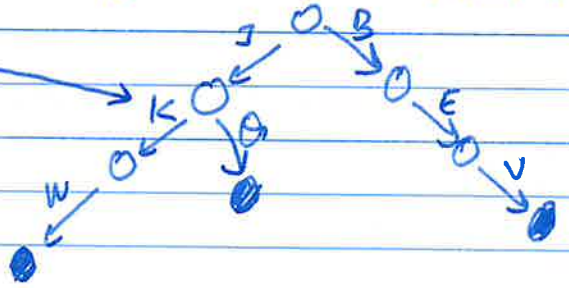
$o$ = number of occurrences of word in text

Use this algo, over KMP when given a list of words to search instead of a single word.

E.g. words = {JKW, JA, BEV}

There are three steps

1. Construct trie
2. Suffix link
3. Output link



Suffix link is the same thing as in KMP, where you find the longest prefix string
e.g. GC. After traversing C, we can use this info to directly go to the next character after C

For AC, the proper suffix are C, ""
For ACT, the proper suffix are C, CT, ""

Output links are also needed. Say we traverse ab, then for the next char c, we would move to the suffix link and print bc. But what if 'c' was also in the set of words. For this reason, every node also maintains an output links that points to the words that points to the words also in the dict while traversing using suffix links.

```
words = ["...", "...", "..."]

class Node:
    def __init__(self):
        self.child = {}
        self.suffix_node = None
        self.output_node = None
        self.word_end_index = -1   → Here instead of storing a bool as the entire word, we store the index of the word in "words"

class AhoCorasick:
    def __init__(self, words):
        self.root = Node()                          ┐ Same as regular Trie
        for i in range(len(words)):                 │
            self.insert(words[i], i)                ┘

        self.insert_suffix_output_nodes()

    def insert(self, word, index):                  ┐ Same as Trie
        node = self.root                            │
        for c in word:                              │
            if c not in node.child: node.child[c] = Node()
            node = node.child[c]                    │
        node.word_end_index = index                 ┘
```

To extend the implementation of ① for this, you can get a list of all the indexes at which a pattern is found i.e.

[1-3, 2-7, ....]

- Then you can sort these indexes based on first-index, then-second and find the difference b/w two consecutive pairs. The diff of length L would give the substring that does not contain any of the given patterns.

③ Find the shortest string containing all given strings

- Similar to ②. Get a sorted list of indexes [1-3, 2-7, ...]
- Iterate through the list, add the word to a set. If you haven't found all the words, remove the
- Then something like DP / Kadane's algo, through a 2-pointer solution is possible. - Use a 2-pointer soln then

Keep minimizing the range till there are duplicates of a word present inside that range

[1-3, _____, 10-11]

↓

If this range contains the word in 1-3, then you can remove 1-3 from the ans (and same for 10-11)

④ Find the lexicographical smallest string of length L containing K strings
Extension of ③ but use DP instead

Pattern match code

```
out = [[] for _ in range(len(words))]
node = aho.root
text_index = 0
while text_index < len(text):
    c = text[text_index]

    if c in node.child:
        node = node.child[c]

    if node.word_end_index != -1:
        out[node.word_end_index].append(
            text_index - len(words[node.word_end_index])+1)
```

Add all the words that end at that node i.e.
abcd. If cd, abcd are in words, then abcd is added using above and cd is added using output-nodes

⇒ Suffix tree

Tree with all the suffixes of a string as keys and positions in text as values

xyzx$ , then suffixes are
xyzx$
yzx$
zx$
x$
$

(it is a compressed tree)
↳ this means merge all the chains of single nodes

The string is padded with a terminal char like '$' which is a unique char. This ensures no suffix is a prefix of another and there are n leaf nodes.

Applications

- In pattern matching, we can preprocess the text by constructing a suffix tree and then search time for each pattern would be $\Theta(m)$ where m is the length of the pattern
  Because construction of the suffix tree on a large amount of text can be costly, use this approach when text is fixed or is changed infrequently.

- Find the longest repeated substring
  Will be done in $\Theta(n)$ time and space. Construct the suffix tree, find the deepest internal node with more than one child. The string from root to that node is the longest repeated substring.
- For longest repeated substring with k-occurrences
  Construct suffix tree. For each internal node count the number of leaf descendents (just post order traversal). Then same as above, find the deepest internal node with atleast k leaf descendants

- Longest common substring problem can also be solved using this (the usual approach is DP)
  DP takes quadratic time while suffix tree takes linear time

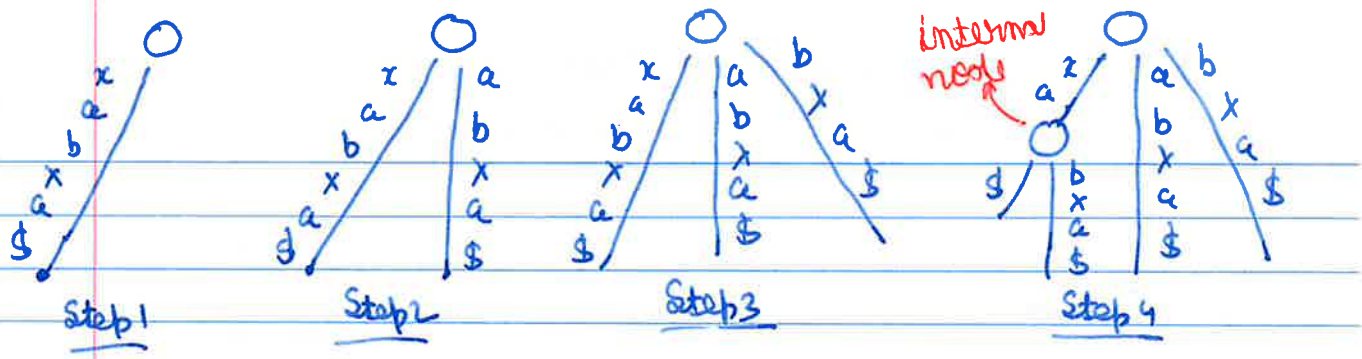
- Longest palindrome substring. In a string, find a substring that is a palindrome and maximize its length
  The usual approach is to find palindromes of length 2 & 3 and check all of them to find the longest palindrome. time = $O(n^2)$

  Manacher's algo can do this in $O(n)$ time.

  Reverse the string and use LCS, with the condition that LCS in string & reversed string should come from same position in

- Suffix trees can also be used for approximate string matching with string mismatches

Eg. $S = xabxa\$$




An implicit suffix tree
- Remove all terminal $ from edge labels of the tree
- Remove any node that has only one edge going out of it and merge the edges

Ukkonen's algorithm constructs this tree in $O(m)$ time.
- Build an implicit suffix tree using $S[0]$
- Extent previous suffix tree with $S[1]$ and so on
- True suffix tree built by adding $ char in tree

Because the tree is being built from first to last index, this means it can be used on-line also.

So in general we are doing 'suffix extensions' by adding the next char in suffix tree. There are 3 rules
1. If $S[i]$ is the last char on leaf edge, then $S[i+1]$ is just added to the end of the label on that leaf edge
2. If $S[i]$ ends in middle of edge and the next char on edge is not $S[i+1]$, then add a new edge with label = $S[i+1]$
3. If $S[i]$ ends in middle of edge and next char on edge = $S[i+1]$, do nothing

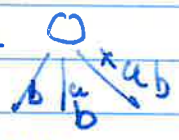
Eg. For $S = xabxa\$$

$T_1$ = 

$T_2$ = 

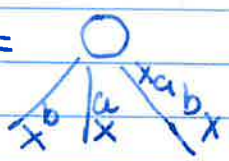
Rule 1 + Rule 2

(as 'x' was last char on leaf edge (rule 1) and there is no next char which is equal to 'a' (rule 2))

$T_3$ = 

Rule 1 + 2

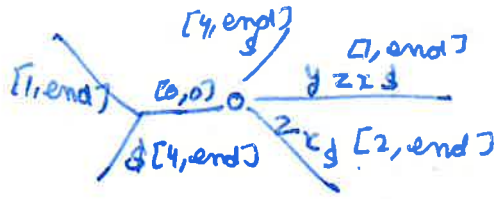
Rule 1: Extend path label in existing edges
Rule 2: Add new leaf

$T_4$ = 

Rule 1: Extend path label in existing leaf edge
Rule 3: Path with label x already exists

Summary of trees:

1. Skip / count
   Edge label compression
2. Rule 3 extension is show stopper
3. Global end for leaves

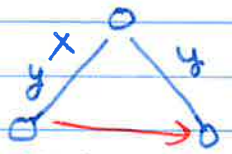



4. Suffix link

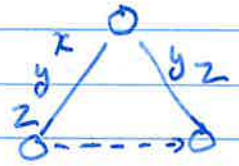
single char → 0 or more chars

For every internal node 'v' with path 'tα', there is another internal node 'sv' with 'α' which is suffix link of v.

e.g.




,




as x y, and α = y is in tree

t α

5. Active points : point at which next extension / phase will start

activeNode = root

activeEdge = -1 → index of char in string

activeLength = 0

- If activeLength = 0, start from root
- when rule 3 applies, increment activeLength
- if an internal node is visited, set it to activeNode

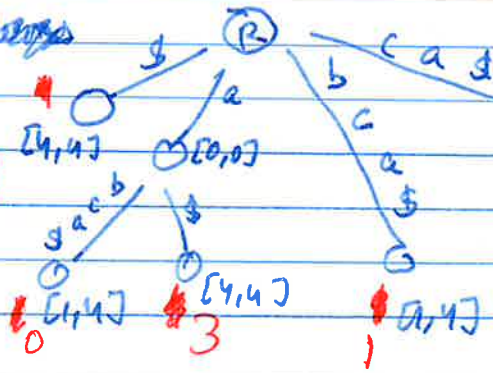
---

Implementation is very long and complex, so not worth it.

Application

1. Search for a pattern in O(pattern) time
   - Start from root and traverse the characters of pattern
   - if we are still in the tree after traversal, then pattern is found.

~~Searching all patterns~~




S = a b c a $
0 1 2 3 4

→ index of suffix ending at this leaf from root

```
while len(stack) != 0:
    top_node = stack.pop()
    for child in top_node.child.values():
        if child.index != -1:
            out.append(child.index)
        else:
            stack.append(child)
    out.sort()
    return out
pat_index += 1
```

```
node = child_node
child_node = node.child.get(pat[pat_index])
return []  → ran out of tree to search
```

```
tree = SuffixTree("abcabxabcd")
print(tree.search("abc")
```

② Longest repeated substring

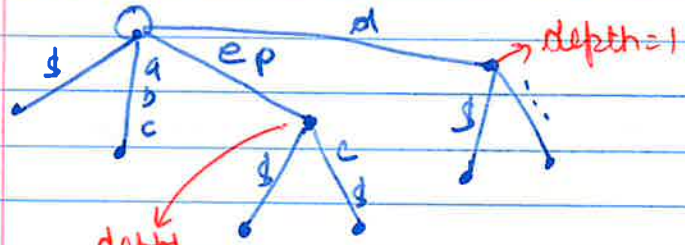
E.g. AAA → AA

pepapep → pep

Find the deepest internal node, and the string from root to that node is ans.

There can be multiple nodes of same max depth, in that case multiple answers are there.

E.g.




```
def longest_repeated_substring(self):
    self.max_depth = 0
    self.out = []

    for child in self.root.child.values():
        self.dfs(child, 0, child.start, child.end)
    for (start, end) in self.out:
        print(self.s[start:end+1])
```

Further, if you are given more than two strings, then you can extend this approach of concatenating strings as $S_1 \# S_2 \# S_3 \$$. The only problem that will arise is the number of available unique terminal symbols.
There is an approach which uses only one terminal symbol also.

The simplest way to solve LCS (even for multiple strings)
- Concatenate all strings with '#'
- Build a suffix tree
- Use 'Longest Repeated Substring' algorithm, where we use the following constraints
  - Because '#' is a unique symbol, the substring would always belong to one string.
  - ~~When searching~~ When a substring is found, visit the leaf nodes and check if there is an 'index' that belongs to each of the input strings
  e.g. abab # cd$. In this case 'ab', # should not be the ans as it belongs to the same string.

To do this, maintain a list $(S1_{start}, S1_{end}), (S2_{start}, S2_{end}), \ldots$
then for every leaf node, find its range and increment the count of that. In the end, if every range has a count $\geq 0$, it means you found the ans

For 2-strings, just hardcode the logic.

this can be done by maintaining a sorted list of $[S1_{start}, S2_{start}, \ldots]$ and doing a binary search to find appropriate list.

LCS code with 2 strings shown below

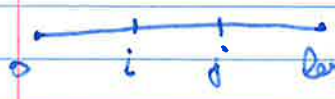
Code same as before

```
def longest_common_substring(self):
    self.max_depth = 0
    self.out = []

    self.first_end = self.s.find('#') - 1
    self.second_end = len(s) - 2

    for child in self.root.child.values():
        self.dfs(child, 0, child.start, child.end)

    for (start, end) in self.out:
        print(self.s[start: end+1])
```

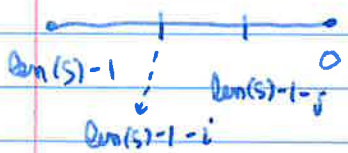
$S = a \# b \$$
For K-string it would be a list
For $K = 2$, let $S = "a\,b\,C\,\#\,d\,e\,f\,\$"$
(first_end: C; second_end: f)

(4) Longest palindromic substring

- Reverse the string S and add it to S as 'S#R$'
- Longest common substring can now be applied but we have to add one additional constraint
- LCS should be from the same position in S




Here S = 'S#R$'




So Substring in S should start at $S_i$

Substring in R should start at $(len(S)-1)-(S_i+L-1)$

↑ length of substring

To implement this, you can do a pre-DFS traversal where you add the forward and reverse indices to each node

i.e. forward_start = start

reverse_start = $(len(S)-2)-(forward\_start+L-1)$

↑ end - start + 1

Then you only consider the indices for which this constraint is being met.

→ Suffix Array

| Input String | Get all suffixes | Sort the suffixes | Take the indices |
|---|---|---|---|
| BANANA | 1 BANANA | 6 A | [6, 4, 2, 1, 5, 3] |
| | 2 ANANA | 4 ANA | |
| | 3 NANA | 2 ANANA | |
| | 4 ANA | 1 BANANA | |
| | 5 NA | 5 NA | |
| | 6 A | 3 NANA | |

- Space = O(n)
- Same computational power as suffix trie
- Construction time = O(n) but hard to implement
  O(n) using suffix tree
- ~~O($n^2 + n \log n$)~~ approach is the simplest. Generate all the suffixes in $n^2$ time and then sort them in ~~n log n~~ $S_i \log S_i$ time using trie where $S_i$ is the number of nodes in the trie
- O($n \log^2 n$) approach also exists

s = '2+3*4*5+(1-6)*(7*9+1)'

~~def process_op(op, num_stack~~

Evaluate expression + generate reverse polish

```
def priority(op_stack_top):
    op, unary = op_stack_top[0], op_stack_top[1]
    if unary:
        return 3
    else:
        if op in ('+', '-'):
            return 1
        if op in ('*', '/'):
            return 2
        return -1
```

Unary gets highest priority

Unary ops can occur
- At beginning
- After ( opening parenthesis
- After another unary op

*, / higher than +, - These are evaluated right to left (including exponential)

Normal ops evaluated left to right

num_stack = [2, 3, 4, ...]

op_stack = [[op, is_unary], ... ]

```
def process_op(op_stack_top, num_stack, reverse_polish):
    op, unary = op_stack_top[0], op_stack_top[1]
    reverse_polish.append(op)
    if unary:
        num = num_stack.pop()
        if op == '+':
            num_stack.append(num)
        elif op == '-':
            num_stack.append(-num)
    else:
        num2 = num_stack.pop()
        num1 = num_stack.pop()
        if op == '+':
            num_stack.append(num1 + num2)
        elif op == '-':
            num_stack.append(num1 - num2)
        elif op == '*':
            num_stack.append(num1 * num2)
        elif op == '/':
            num_stack.append(num1 // num2)
```

In case of 'else', just pop another val. In that case nothing

```
while len(op_stack) != 0:
    process_op(op_stack.pop(), num_stack, reverse_polish)
return num_stack[0], "".join(reverse_polish)
```

## → Regular expression matching

* → 0 or more of previous char

. → matches any single char

Using DP

```
T[i][j] = T[i-1][j-1] → if text[i] == pattern[j] || pattern[j] == '.'
                        if pattern[j] == '*':
          T[i][j-2]   → 0 occurrence
          T[i-1][j]   if str[i] == pattern[j-1] || pattern[j-1] == '.'
          False       (in all other cases)
```

i = text index
j = pat index

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | T | F | F | F | F |
| 1 | F | | | | → |
| 2 | F | | | | |
| 3 | F | | | | |
| 4 | F | | | | |

(Assume text & pattern start from index 1

```
dp = [[False for _ in range(len(p)+1)] for _ in range(len(s)+1)]
dp[0][0] = True          → empty text and pattern
for i in range(len(p)):
    if p[i] == '*' and dp[0][i-1]:
        dp[0][i+1] = True
```

→ ~~Empty text~~ For 'p = c*'. In this case we can ignore c, so if text is empty then c* would match it

```
s, p = '$'+s, '$'+p   → Makes the indexing easier to understand

for i in range(1, len(s)):
    for j in range(1, len(p)):
        if s[i] == p[j] or p[j] == '.':
            dp[i][j] = dp[i-1][j-1]
        elif p[j] == '*':
            if dp[i][j-2]:
                dp[i][j] = True
            elif s[i] == p[j-1] or p[j-1] == '.':
                dp[i][j] = dp[i-1][j]

return dp[len(s)-1][len(p)-1]
```

Just ignore c* i.e. check the match before 2 chars of pat

Char matches in str and pattern. '.' match any char.

~~S=abc~~ S = abcd
P = def[d]
→ After a match we can remove just 'd' and check S, P without them

→ In this case we have atleast one occurrance, so just check ~~with~~ by removing that one char from S

# Levenshtein distance / Edit distance

~~lev(a,b) = { len(a) if |b|=0 ; len(b) if |a|=0 ; lev(a[1:], b[1:]) if a[0]==b[0] ; 1 + min{ lev(a[1:], b) ; lev(a, b[1:]) ; lev(a[1:], b[1:]) } otherwise }~~

- Insert a char
- Delete a char
- Replace a char

For strings a, b

$$lev(a,b) = \begin{cases} len(a) & \text{if } len(b) == 0 \\ len(b) & \text{if } len(a) == 0 \\ lev(a[1:], b[1:]) & \text{if } a[0] == b[0] \\ 1 + \min \begin{cases} lev(a[1:], b) \\ lev(a, b[1:]) \\ lev(a[1:], b[1:]) \end{cases} & \text{otherwise} \end{cases}$$

Some useful bounds

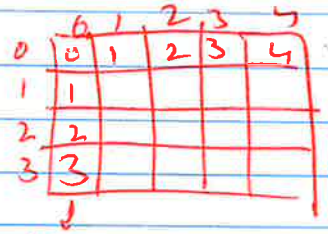
- ~~It is ≥ diff~~
- It is $\geq |len(a) - len(b)|$
- It is $\leq \max(len(a), len(b))$
- It is 0 iff $a == b$
- If the strings are of same size, then Hamming distance is the upper bound
- Triangle inequality b/w 3 strings is followed

Applications

- In approximate string matching, find matches for short strings in many longer texts (short strings can come from a dict)
- The above is useful for spell checkers, correction system for OCR
- The general guidance is one string should be small to keep the overall time small.

Edit distance b/w 2 strings word1 & word2 code

```
R = len(word1) + 1
C = len(word2) + 1

dp = [[0 for _ in range(C)] for _ in range(R)]
for c in range(C): dp[0][c] = c
for r in range(R): dp[r][0] = r
```




These base cases are easy to understand as dist b/w empty string and non-empty is length of non-empty

```
text = 'TATTGGCTATACGGTT'
pat = 'GCGTATGC'
dp = [[[0,'diag']] for _ in range(len(text)+1)] for _ in range(len(pat)+1)]   # to get the result
for i in range(len(text)+1):
    dp[0][i][0] = None
for i in range(len(pat)+1):
    dp[i][0][0] = i

for i in range(1,len(pat)+1):
    for j in range(1,len(text)+1):
        up = dp[i-1][j][0]
        left = dp[i][j-1][0]
        diag = dp[i-1][j-1][0]

        if pat[i-1] == text[j-1]:
            dp[i][j][0] = diag
        else:
            if up <= left and up <= diag:
                dp[i][j][1] = "up"
            elif left <= up and left <= diag:
                dp[i][j][1] = "left"
            dp[i][j][0] = 1 + min(up, left, diag)

K = 2
for i in range(len(text)+1):
    if dp[-1][i][0] <= K:
        r, c = len(pat), i
        cell = dp[r][c]
        while cell[1] is not None:
            if cell[1] == 'diag':
                r, c = r-1, c-1
            elif cell[1] == 'left':
                c = c-1
            elif cell[1] == 'up':
                r = r-1
            cell = dp[r][c]
        substr = text[c-1:i]
```

None is the stopping condition of backtrack

By default, 'diag' is assigned

→ Iterate over last row and the cells with value <= K are the answer

start_index = c-1
end_index = i
substr = text[start_index : end_index]

## Search word in dict with '.'

words = [ cat, bat, rat, drat, drab ]  '.' can match any char

cat → true
.at → true
.a. → true

Two ways to solve this

① Create a hashmap with key = length of word and value = list of all words with that length. Then for a given query, just iterate through the list of words of that length till a match is found

Time = $O(MN)$, where M = length of word and N = number of words

② Trie can be used. And for query '.' just do a DFS.

Time = ~~O(?) on average if word~~

$= O(N \cdot 26^M)$ where N = number of chars in query that are not '.'

M = number of dots in query

③ You can combine the two approaches and based of number of dots in the query, decide which approach to use

For trie the code is

```
node = self.root
for c in word:
    if c not in node:
        node[c] = {}
    node = node[c]
node['$'] = True
```

Add word to self.root = {}

```
def search(word):
    node = self.root
    self.res = False
    self.dfs(node, word, 0)
    return self.res

def dfs(self, node, word, index):
    if node is None: return
    if index == len(word):
        if '$' in node:
            self.res = True
        return
    if word[index] == '.':
        for child in node.values():
            if type(child) != bool:
                self.dfs(child, word, index+1)
    else:
        if word[index] not in node:
            return
        node = node[word[index]]
        self.dfs(node, word, index+1)
```

to skip the '$' key

Total size of message
$= A + B + C + D + E$
$= 3\times3 + 5\times2 + 6\times2 + 4\times2 + 2\times3$
$= 45$

Along with the message, the tree / codes should also be sent
For chars we have $5\times8 = 40$ bits
For codes we have = 12 bit
Total = $45 + 40 + 12 = 97$ bits

Message = BCCABB...
Encoded message = 1011 11001 1010...
For decoding
- Traverse the tree from root following 1 for right edge, 0 for left edge.
- When a leaf is reached, it means it is that character. Then move back to root and repeat

If better compression ratio is required use
- Arithmetic coding
- Asymmetric numeral systems

Time = $O(n \log n)$ where $n$ = number of unique characters. As there are $n$ leaf nodes and $n-1$ non-leaf nodes. $O(\log n)$ for every insertion in min heap. This time does not include time to get the frequency array.

Applications:
- Used for transmitting fax and text
- Used by GZIP, ..
- Multimedia codecs like JPEG, PNG, MP3

Variations
- n-ary Huffman coding - Pick n-smallest nums and build n-ary tree
- Adaptive huffman coding - Calculate the freq dynamically based on recent actual freqs, and update the tree. Used rarely in practice because the tree update is slow and adaptive arithmetic coding is better
- ...

```
for i in range (len(heap)-1, -1, -1):
    min_heapify (len(heap)-1, i)
```

Step 2 (internals)

```
while len(heap) != 1:
    left_child = heap.pop(0)
    min_heapify (len(heap)-1, 0)

    right_child = heap.pop(0)
    min_heapify (len(heap)-1, 0)

    new_node = node (left_child.val + right_child.val)
    new_node.left = left_child
    new_node.right = right_child

    heap.insert (0, new_node)
    min_heapify (len(heap)-1, 0)
root = heap[0]
```

Get minimum 2 values from the array

Add their sum back to the array

First min is left child
Second min is right child

```
codes = {}
stack = [(root, "")]
while len(stack) != 0:
    node, curr_code = stack.pop()
    if node is None:
        continue
    if node.c is not None:
        codes[node.c] = curr_code
    else:
        stack.append ((node.left, curr_code + "0"))
        stack.append ((node.right, curr_code + "1"))
```

Step 3: Get the codes for each character from the tree

Add 0 for left child and 1 for right child

```
out = []
for c in text:
    out.append (codes[c])
return "".join(out), codes
```

Step 4: Generate the encoded text and return the codes to be used for decoding.

```
heap = []
for c, f in freq.items():
    heap.append(Node(f, c, {c}))
```

Add the 'char' as leave for all leaf nodes

```
# run-heapify the 'heap' array
```

```
codes = {}
for c in freq.keys():
    codes[c] = []
```

Create the 'codes' dict for mapping char → code

```
while len(heap) != 1:
    left_child = heap.pop(0)
    min_heapify(len(heap)-1, 0)
    for c in left_child.leaves:
        codes[c].append('0')
```

Add '0' to left leaves

```
    right_child = heap.pop(0)
    min_heapify(len(heap)-1, 0)
    for c in right_child.leaves:
        codes[c].append('1')
```

Add '1' to right leaves

```
    new_node = Node(left_child.val + right_child.val,
                    leaves = left_child.leaves + right_child.leaves)
    new_node.left = left_child
    new_node.right = right_child
```

Concatenate the left and right leaves

```
    heap.insert(0, new_node)
    min_heapify(len(heap)-1, 0)

for k,v in codes.items():
    codes[k] = "".join(v[::-1])
```

Reverse the codes

# Combinatorics

## Find powers of factorial divisor (Legendres formula)

given n and k, find the largest power of k which divides n! i.e. i.e. find largest x such that n! is divisible by $k^x$.

If k is prime:

$n! = 1 \cdot 2 \cdot 3, \ldots n$

then every $k^{th}$ element in the above product would divide by k and the total number of such elements is $\lfloor \frac{n}{k} \rfloor$

e.g. $10! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10$

$k = 3$, then only 3, 6, 9 are part of ans

We can extend the same logic to $k^i$

So the largest power of k is

$$\lfloor \frac{n}{k} \rfloor + \lfloor \frac{n}{k^2} \rfloor + \ldots + \lfloor \frac{n}{k^i} \rfloor + \ldots$$

i.e. we want to find the number of numbers which are divisible by $k, k^2, k^3, \ldots$

```
def fact_pow (n, k):
    x = 0
    while n > 0:
        n = n // k
        x = x + n
    return x
```

Time $= O(\log_k n)$

If k is composite:

We first need to factor k into $k = k_1^{p_1} \ldots k_m^{p_m}$ i.e. product of primes. For each $k_i$, we find it is present in n!. So the final ans becomes

$$\min_{i=1 \ldots m} \frac{q_i}{p_i}$$

## Binomial Coefficients

$\binom{n}{k}$ are the number of ways to select a set of k elements without taking into account the order of arrangement of these elements.

these also occur in

$$(a+b)^n = \binom{n}{0} a^n + \binom{n}{1} a^{n-1} b + \ldots + \binom{n}{k} a^{n-k} b^k + \ldots + \binom{n}{n} b^n$$

$\binom{n}{k} = \frac{n!}{(n-k)! \, k!}$. To choose k elements from n elements the number of ways are $n \cdot n-1 \cdot n-2 \ldots = \frac{n!}{(n-k)!}$

You can also build a table in $O(n^2)$ to get the values for all the binomial coefficients using $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

```
def f(n, k):
    res = [[0 for _ in range(n+1)] for _ in range(n+1)]
    res[0][0] = 1
    for i in range(1, n+1):
        res[i][0] = 1
        res[i][i] = 1
        for j in range(1, i):
            res[i][j] = res[i-1][j-1] + res[i-1][j]
    return res[n][k]
```

## Catalan Numbers

- For combinatorial problems, often involving recursively defined objects

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

$$C_0 = C_1 = 1$$

$$C_n = C_0 C_{n-1} + C_1 C_{n-2} + \dots + C_{n-1} C_0$$

or

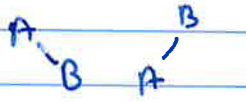
$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

## Applications

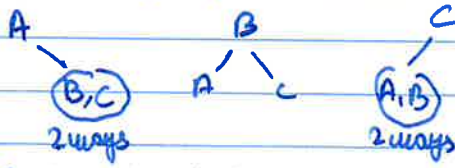
- Number of BSTs with N nodes $= C_n$

For 0 node → 1 way

For 1 node → 1 way

For 2 nodes → 

(assuming A < B)

For 3 nodes 

= 5 ways (A < B < C)

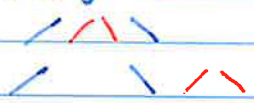
$C_0 C_2 + C_1 C_1 + C_2 C_0$

Main idea is you can take any node as root node and then you have to place nodes on left and right

0 X4 → 1 way
1 X4 → 1 way (X4)
2 X4 → X X4 Y
X Y X4

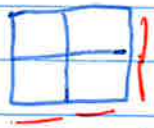
The logic is same. You put one X4 then and then decide how many can we put inside and outside.

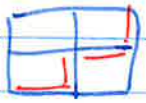
→ Mountain Ranges → $C_n$
Given N upstrokes and N downstrokes, such that we always stay above a horizontal line.
Or calculate the number of mountains and valleys such that you always stay above sea level.

N = 0 → 1 way
N = 1 → 1 way 
N = 2 → 

Again same logic. Put 1 pair of upstroke, downstroke and then decide how many to put inside and outside.

→ Path on grid → $C_n$
Given N X N grid, you have to go from bottom left to top right corner while staying below the main diagonal (or equal)
To stay below diagonal take more left than up (which is same as X4 question)

2 →  HHVV (always below the diagonal)

 HVHV (also (equal to the diagonal)

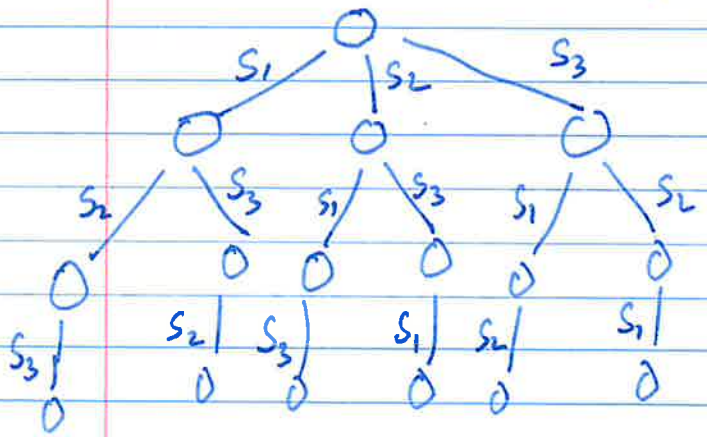
→ Convex polygon → $C_{n-2}$
Number of ways of triangulation of a convex polygon. Polygon of N sides, find number of ways to convert it into triangles.

## → Backtracking

It is a brute force approach where we try out all the possibilities. But unlike dynamic programming where we are also brute forcing and optimizing. In backtracking we do not do optimization.

Backtracking is used when there are multiple solutions and you want all of them.

E.g. Given 3 students $S_1, S_2, S_3$ in how many ways can we sit them on 3 chairs. It is $3!$ and to get generate all these solutions backtracking is used. For this, we need to consider the "state space tree"




Generally for these problems, we also have some constraints like $S_2$ should not sit in middle. Again, draw the state space tree and prune it when the constraint is being violated.

~~Backtrack~~ Another similar approach to backtracking is 'branch and bound' which is used in traveling salesman. The diff is how the state space tree is traversed

Backtracking → DFS
Branch and bound → BFS

## N-Queens problem

Given a n×n chess board, place n queens such that no two queens attack. Return all the distinct solutions. (in the leetcode problem the output format is of a ~~a list of n~~ single solution is a list of n strings, each of length n with 'Q' for queen and '.' for empty)

n = 4

[[".Q..", "...Q", "Q...", "..Q."], ["..Q.", "Q...", "...Q", ".Q.."]]

Checking for constraints

- Same column: No need to check as we reduced the problem from NxN matrix to a list of 'N' size and as each entry in this list corresponds to a different column, we do not need to check
- Same row: If state is [2, 3, 1, -1], and index = 2. This means we have only placed first 3 queens and we have already verified that queens in position 0,1 are valid. And we only need to verify queen at position 'index', which is simple

```
for i in range(index):
    if State[i] == State[index]:
        return False
```

- Same diagonal: There are two diagonals, and there are 2 cases

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | -1 | -2 | -3 |
| 1 | 1 | 0 | -1 | -2 |
| 2 | 2 | 1 | 0 | -1 |
| 3 | 3 | 2 | 1 | 0 |

For ↘ diagonal, the difference i.e. row - col is constant along the diagonal

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 |
| 1 | 1 | 2 | 3 | 4 |
| 2 | 2 | 3 | 4 | 5 |
| 3 | 3 | 4 | 5 | 6 |

For ↗ diagonal, the sum i.e. row + col would be constant along the anti-diagonal

```
for i in range(index):
    if i == row, State[i] == col
```

For N-queen, brute force would be trying out all the combinations i.e. $^{N^2}C_N$ $= O(N^{2N})$

With backtracking we reduce it to ~~O(N!) time~~

- $O(N!)$ time as we have a vector of size N and we are trying all possible outcomes with pruning e.g.

  [1 1 X X]. We pruned the "x x" positions as "1 1" can never occur.

So for first position we have N places to place the queen, for second pos there are N-1 positions (as we don't count diagonal and same row), and then it is N-4, and so on

Although it takes $O(N^2)$ to board each valid position, as we prune if an invalid state is reached. So total = $O(N^2 + N!) = O(N!)$

## Sudoku

Write a program to solve a sudoku puzzle by filling the empty cells.
Empty cells represented by "."

dfs(0,0) → the main recursion call

```
n = 9
def dfs(row_ind, col_ind):
    if row_ind == n-1 and col_ind == n:        # Base condition, when we reach end of board
        return True
    if col_ind == n:
        row_ind += 1
        col_ind = 0
    if board[row_ind][col_ind] != ".":         # if number already present, then skip it as we cannot modify it
        return dfs(row_ind, col_ind+1)

    for val in range(1, 10):                   # → try all possible values and prune
        if is_valid(row_ind, col_ind, str(val)):
            board[row_ind][col_ind] = str(val)
            if dfs(row_ind, col_ind+1):        # Here we will only get one solution and then can return it.
                return True
            board[row_ind][col_ind] = "."      # Imp. → in this situation, when we backtrack, we should set the future states back to default
    return False

def is_valid(row_ind, col_ind, val):
    for i in range(n):
        if board[i][col_ind] == val:           # → check column
            return False
        if board[row_ind][i] == val:           # → check row
            return False

    row_start, col_start = (row_ind//3)*3, (col_ind//3)*3   # → check 3x3 box
    for r in range(row_start, row_start+3):
        for c in range(col_start, col_start+3):
            if board[r][c] == val:
                return False

    return True
```

```
def adjacency_matrix(num_nodes, edges):
    adj_matrix = [[0 for _ in range(num_nodes)] for _ in range(num_nodes)]
    for u, v, w in edges:
        adj_matrix[u][v] = w
    return adj_matrix

def adjacency_list(num_nodes, edges):
    adj_list = [[] for _ in range(num_nodes)]
    for u, v, w in edges:
        adj_list[u].append((v, w))
    return adj_list

def adjacency_set(num_nodes, edges):
    adj_set = [set() for _ in range(num_nodes)]
    for u, v in edges:
        adj_set[u].add(v)
    return adj_set
```

Cannot store weight alongside 'v' as you will not be able to search.

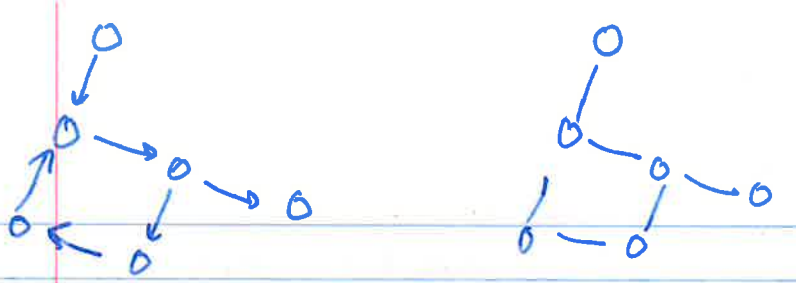
## Transpose a graph

- For adjacency matrix there are two options
  - ~~Invert~~ Invert the coordinates i.e. use A[j][i] instead of A[i][j]. You can create a wrapper function for this. This approach has the benefit of not moving the data. So if you have a large matrix, you can use this technique provided you do not have to do too many operations with the transposed matrix
  - Swap the values. Look ~~through~~ i & j and ~~sw~~ as

```
for i in range(1, N-1):
    for j in range(i, N):
        A[i][j], A[j][i] = A[j][i], A[i][j]
```

But this technique modifies the original matrix and this may not be allowed in real life.
  - Create a new matrix and add the transposed values to it.

For adjacency list
- Create a new list and as you traverse the original list of node 'u', Add ~~to the~~ 'u' to the list of all v's.



3 Strongly connected components

1 weakly connected component

Code for connected components (undirected graph)

```
def dfs (adjacency_list, num_nodes):
    visited = [False] * num_nodes
    cc_count = 0
    for node in range (num_nodes):
        if not visited [node]:
            cc_count += 1
            self. dfs_helper (node, adjacency_list, visited)
    return cc_count
```

—

Check if graph is tree

A tree has
- A single connected component
- V-1 edges

So first check if graph has 1 connected component and then check if there V-1 edges.

Bipartite graph

A graph where you can divide the vertex set into 2 disjoint sets such that
- Each vertex belongs to exactly one of the two sets
- Each edge connects vertices of 2 different sets.

To solve this
- On a high level we visit one vertex set node, then take an edge and visit the other set node and so on
- To simulate this without a set, you can color the graph using 2 colors. So if you visit color 0, then all its neighbors should have color 1 and so on.

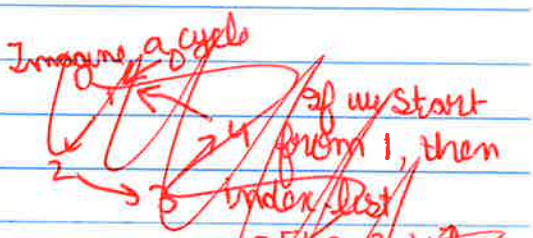
```
def dfs(node, adj_list, visited, parent):
    visited[node] = True
    for v in adj_list[node]:
        if visited[v]:
            if v != parent:
                return True
        else:
            out = dfs(v, adj_list, visited, node)
            if out:
                return True
    return False
```

## Longest cycle in a graph

You are given a directed graph, where each node has atmost one outgoing edge, represented as edges = [3, 3, 4, 2, 3, -1]

node 0 has an edge to node 3

```
max_length = -1
visited = [False] * len(edges)
        ↳ None = never visit again
          True = can visit for current cycle
          False = never visited before
for node in range(len(edges)):
    if visited[node] is None:
        continue

    length = 0
    index = i
    index_list = [index]
    while edges[index] != -1:
        if visited[index]:
            for j in range(len(index_list)):
                if index_list[j] == index:
                    offset = j
                    break
            length = length - offset
            max_length = max(max_length, length)
            break
```




Explanation #1 (See next page)

Compute in/out time

```
in_time = [0]* num_nodes
out_time = [0]*num_nodes
visited = [False]* num_nodes
timer = 1
for node in range(num_nodes):
    if not visited[node]:
        dfs(node, adj_list, visited)

def dfs(node, adj_list, visited):
    visited[node] = True
    in_time[node] = timer
    timer += 1
    for v in adj_list[node]:
        if not visited[v]:
            dfs(v)
    out_time[node] = timer
    timer += 1
```

Diameter of tree

Longest path between any 2 nodes in the tree
- Take any node as root and run DFS and find the farthest node. Let it be X
- Run DFS from X and find the farthest node. The distance between them is the diameter.

The graph might have to be a MST for this to work
- In an unweighted graph, you can apply MST as normal
- In weighted graph, you have to constrain the MST so as to get the biggest diameter. This makes it an NP-hard problem. (Why can't you just treat it as unweighted?)

```
class graph:
    def __init__(self):
        self.max_dist = -1
        self.max_node = None
        self
```

# Disjoint Sets / Union - Find

- We can count number of connected components in a graph
- Minimum Spanning Tree
- Set representation e.g. {1, 2, 3, 4, 5} can be represented as disjoint sets

Operations on disjoint set

- Find (n) : Find the set to which 'n' belongs to
  e.g. A = {1, 4, 7}, B = {10, 14}
  find (4) returns A

Instead of naming the set by characters, we choose one of the elements in the set as parent. So
find (4) returns 1

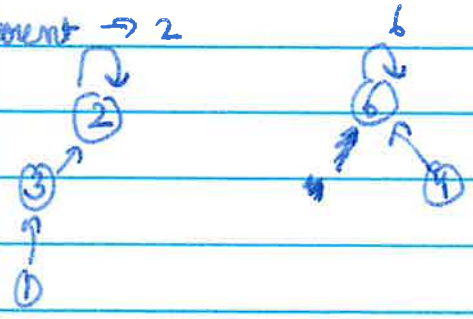
(Also the parent id would be unique among all the sets)

- Union (a, b) : merge 2 sets
  A = {1, 4, 7}, B = {10, 14}
  Union (1, 4) → does nothing as they belong to same set
  Union (4, 10) → will merge A & B

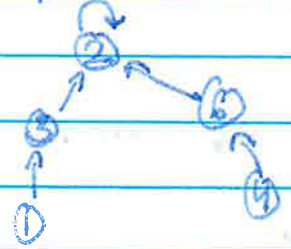
Representation

{2, 3, 1}, {4, 6}

Parent → 2




So in 'union' operation all we have to do is change the parent pointer of the parent. So union of A & B would be



We start with parent array with each element as parent of itself

parent = | 1 | 2 | 3 | 4 | 5 |

index 1 2 3 4 5

Then for each of the 'm' relations we apply the union operation

For the given question, we want to get the size of each group, so we initialize the parent as

parent = | -1 | -1 | -1 | -1 | -1 |

Here -1 at index 1 means, '1' is a parent and it's set contains 1 elements.

For the relation (1, 2), parent is updated as

parent = | -2 | 1 | -1 | -1 | -1 |

Here -2 means '1' is parent & it's set contains 2 elements

After this we can loop through the 'parent' array and get product of all negative numbers

## Path compression

It is used to optimize the 'find' operation. The problem with regular 'find' is that the chain can get quite long.

The main idea is all the nodes should point to 'parent' rather than other nodes

E.g.: (1) ← (2) ← (3) should be (1) ← (2), (3)

this can also be made constant memory by doing 2 passes starting from 4 to parent

① First pass find the parent

② Second pass update all the pointers

```
find(n):
    V = []                          → List of all nodes whose parent needs to be updated
    while parent[n] > 0:
        V.push_back(n)
        n = parent[n]
    for i in range(len(v)):
        parent[V[i]] = n
    return n
```

(1) ← (2) ← (3) ← (4)

find(4)

So V would be [4, 3, 2]

And then we update parent of these elements to 1

This trick is called "merging smaller into bigger" and is also used in Tree DP questions.

In Union by Rank, maximum height = $O(\log n)$ and thus find takes $O(\log n)$ time

Path compression and Union by Rank can also be combined.

Union by Rank code

```
parent = [i for i in range(N)]   → it can be -1 also,
rank = [1 for _ in range(N)]

def merge(a, b):
    a = find(a)
    b = find(b)
    if a == b:
        return
    if rank[a] > rank[b]:
        parent[b] = parent[a]
        rank[a] += rank[b]
    else:
        parent[a] = parent[b]
        rank[b] += rank[a]
```

Problem: String of 0's & 1's. You can replace a character to '1' and for each query return the maximum number of consecutive 1's

Solution:

parent = 


rank = 


str = 1 0 1 1 0

For each index which is "1", if previous element is also "1", then merge them. And maintain a global val for max length

During the query steps, we merge both with the previous and next index.

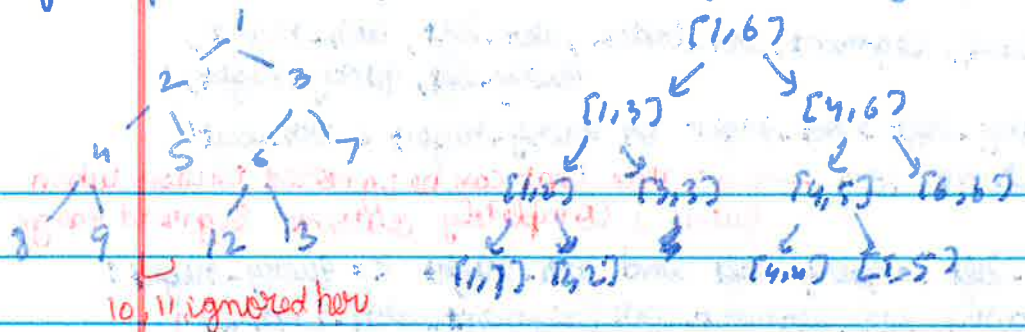
Time complexity = $O(n)$ in case path compression or union by rank is not used

= $O(m \log n)$ using only union by rank for 'm' operations of any kind

= $\Theta(m\,\alpha(n))$ using both path compression & union by rank

(Inverse Ackermann func) → this is ~~almost~~ so small that it is constant

If we replace the ranges with index, for visualization




As you can see from above, we don't need to build a tree, we can just create an array to store the values ~~of all the~~ for all the ranges.

Also, to get the value of [1,6], we just take the min. of [1,3], [4,6] i.e. merge the result of children. So we are building the array from right to left (i.e. leaf to root).

```
import math
class SegmentTree:
    def __init__(self, arr):
        self.arr = arr
        len_segment_arr = 2 * 2**(math.ceil(math.log2(len(arr))))
        self.segment_arr = [0] * (len_segment_arr)

    def buildTree(self, segment_arr_index, arr_left, arr_right):
        if arr_left == arr_right:                                   # leaf node
            self.segment_arr[segment_arr_index] = self.arr[arr_left]
            return

        middle = arr_left + (arr_right - arr_left) // 2
        left_child_index = 2 * segment_arr_index
        right_child_index = left_child_index + 1

        self.buildTree(left_child_index, arr_left, middle)
        self.buildTree(right_child_index, middle+1, arr_right)

        self.segment_arr[segment_arr_index] = min(                  # Main computation
            self.segment_arr[left_child_index],
            self.segment_arr[right_child_index])

# e.g. arr = [1,2,3]
#      t = SegmentTree(arr)
#      t.buildTree(1, 0, len(arr)-1)
```

In this array, index 0 would be wasted, as we are doing index-based computations

```
def update(self, query_index, query_value):
    self.arr[query_index] = query_value
    self.update_func(1, 0, len(self.arr)-1, query_index)
def update_func(self, segment_arr_index, arr_left, arr_right, query_index):
    if arr_left == arr_right:
        self.segment_arr[segment_arr_index] = arr[query_index]
        return
    middle = arr_left + (arr_right - arr_left)//2
    left_child_index = 2 * segment_arr_index
    right_child_index = left_child_index + 1
    if query_index <= middle:
        self.update_func(left_child_index, arr_left, middle, query_index)
    else:
        self.update_func(right_child_index, middle+1, arr_right, query_index)
    self.segment_arr[segment_arr_index] = min(
        self.segment_arr[left_child_index],
        self.segment_arr[right_child_index])
```

Leaf node. Code is same as 'build tree' leaf node

main computation, also same as 'build tree' main computation

Lazy propagation → when updates are made to all elements in a range

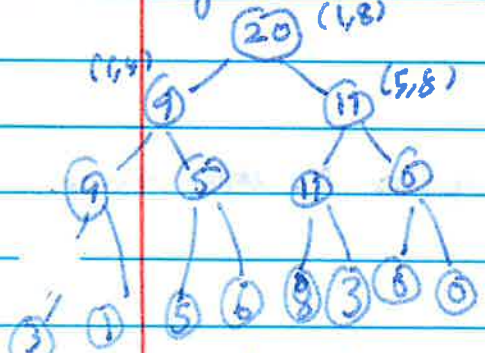
Problem with 'Point update' is that it will take $O(N \log N)$ to update $N$ elements i.e. when the range is $[1, N]$. Using lazy propagation this time can be reduced to $O(\log n)$.

The main idea is update only when needed.

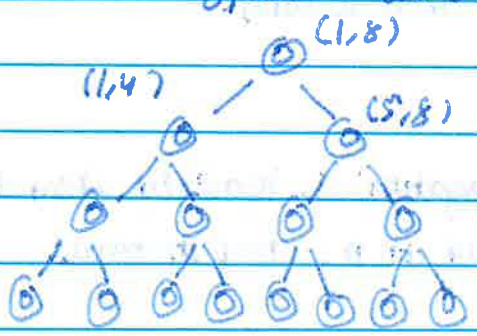
So we maintain a separate array ('lazy tree') where we keep all the pending updates. The lazy tree is exactly same as segment tree (only the value of node is changed)

e.g. 'Range sum' query

Segment tree




Lazy tree → initially all values are 0




Now when we get update query "Add 2 to [4,8]" i.e. add 2 to all these values

Problem: Find the multiples of 3 in an array of size n, where arr is initialized with 0's and we update values in a range say l

Solution: In this case, the node would contain info about 3 values

| Modulo %3 | Count |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | |

So ans would be 2 in this case. When we update all the values in this range say l, the modulo would just get right shifted i.e.

| Modulo %3 | Count |
|---|---|
| 0 | 3 |
| 1 | 2 |
| 2 | |

```
class SegmentTree:
    def __init__(self, n, d=3):
        self.d = d
        len_segment_arr = 2 * 2**(math.ceil(math.log2(n)))
        self.segment_arr = [[0 for _ in range(3)] for _ in range(len_segment_arr)]
        self.lazy_arr = [0]*len_segment_arr

        self.buildTree(1, 0, n-1)

    def buildTree(self, segment_arr_index, arr_left, arr_right):
        if arr_left == arr_right:
            self.segment_arr[segment_arr_index][0] = 1
            for i in range(1, self.d):
                self.segment_arr[segment_arr_index][i] = 0
        middle = ...
        ...
        ...
        self.buildTree(left_child_index, arr_left, middle)
        self.buildTree(right_child_index, middle+1, arr_right)

        for i in range(self.d):
            self.segment_arr[segment_arr_index][i] =
                self.segment_arr[left_child_index][i] +
                self.segment_arr[right_child_index][i]

    def shift(self, segment_arr_index):
        last_val = self.segment_arr[segment_arr_index][-1]
        for i in range(self.d-2, -1, -1):
            self.segment_arr[segment_arr_index][i+1] = self.segment_arr[segment_arr_index][i]
        self.segment_arr[segment_arr_index][0] = last_val
```

Same as before. We did not create an array in this. But if 'range sum' is asked for a given array, then you would need to store the array also

as 0 divisible by 3

How to initialize the leaf

Same as before. Just leaf node & final is changed

```
def query(self, segment_arr: Index, arr_left, arr_right, query_left, query_right):
    self.check_lazy(segment_arr_index, arr_left, arr_right)
    if query..
    ...
```

Same as before, just need to perform the 'lazy update'.

**Merge Sort Tree** → for L R K problems i.e. given a range and some changing thresholds

Segment tree cannot be used for

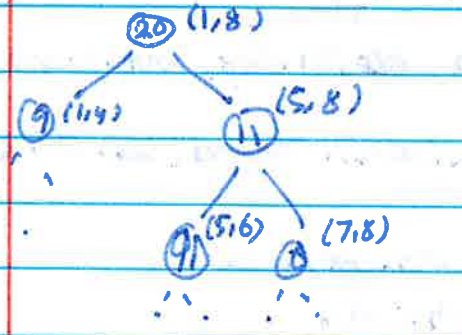
① Problem1: Find the elements in the range L&R that are < k (or the question can be > k)

② Problem2: Find the number of unique elements in range L to R.

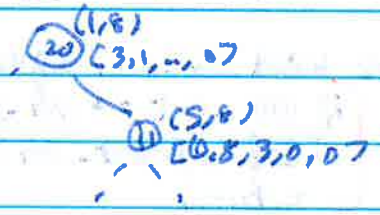
In Merge Sort Tree, instead of storing the result in each node, we store the entire vector of numbers.

e.g. arr = [3|1|5|0|8|3|0|0]

Segment tree (sum)




Merge Sort Tree




The reason we are doing this is because this now allows us to answer queries like find the sum of all numbers in the range [L,R] that are less than k.

Why problem 1 can't be solved with segment tree? If there are no updates, then we can use a segment tree. But when we include updates, then because we are not storing the elements, so there is no way which elements are smaller. (I still don't know why we can't use segment tree. Like during the update we can compare it to the original value. I guess the problem arises when the value of 'k' can change itself)

The tree takes $O(N \log N)$ space as there are $\log N$ levels and at each level there are $N$ values.

Space complexity of segment tree is roughly $O(N)$

How to query? Same as before. If some overlap make recursive call to children. If complete overlap, return the count

```
def query(self, segment_arr_index, arr_left, arr_right, query_left, query_right, query_k):
    # query outside
    if query_left > arr_right or query_right < arr_left:
        return 0
    # segment inside
    if arr_left >= query_left and arr_right <= query_right:
        arr = self.segment_arr[segment_arr_index]
        left, right = 0, len(arr)-1
        while left <= right:
            middle = left + (right-left)//2
            if query_k > arr[middle]:        # if query_k >= arr[middle]:
                left = middle+1
            else:
                right = middle-1
        return left                          # return len(arr) - left
    middle = arr_left + (arr_right - arr_left)//2
    left_child_index = 2*segment_arr_index
    right_child_index = left_child_index + 1
    left_val = self.query(left_child_index, arr_left, middle, query_left,
                          query_right, query_k)
    right_val = self.query(right_child_index, middle+1, arr_right, query_left,
                           query_right, query_k)
    return left_val + right_val
```

If question was values greater than k

```
import math
class SQRT:
    def __init__(self, arr):
        self.arr = arr
        self.block_size = math.floor(math.sqrt(len(arr)))
        self.block_arr = []
        for i in range(0, len(arr), self.block_size):
            min_val = self.arr[i]
            for j in range(i+1, min(i+self.block_size, len(self.arr))):
                if self.arr[j] < min_val:
                    min_val = self.arr[j]
            self.block_arr.append(min_val)

    def query(self, arr_left, arr_right):
        block_left = arr_left // self.block_size
        block_right = arr_right // self.block_size

        if block_left == block_right:
            if arr_right - arr_left + 1 == self.block_size:
                return self.block_arr[block_left]

        min_val = self.arr[arr_left]
        for i in range(arr_left+1, arr_right+1):
            min_val = min(min_val, self.arr[i])
        return min_val

        min_val = self.arr[arr_left]
        for i in range(arr_left, self.block_size * (block_left+1)):
            min_val = min(min_val, self.arr[i])

        for i in range(block_left+1, block_right):
            min_val = min(min_val, self.block_arr[i])
        for i in range(self.block_size * block_right, arr_right+1):
            min_val = min(min_val, self.arr[i])
        return min_val


t = SQRT(arr)
t.query(0,0)
t.query(0,3)
```

Initialize the block array

→ inclusive of both arr_left & arr_right

if first indices of the same block

otherwise traverse the original array

traverse array, then block array, then array

# Stars and Bars

(1) How many non-negative integer solution exist for $x+y=20$ & $(x \geq 0, y \geq 0)$

(2) Select n marbles. Marbles are of k different colors and you need to pick atleast one marble of each color

(3) How many n digit numbers can be formed, where the digits are in non-decreasing order

(4) Given integer n & m. Calculate number of pairs of arrays (a,b) both of length m, $a_i < b_i$, a is sorted non-decreasing, b is sorted non-ascending

**Prob 1** $x+y=5$

Take 5 Stars/circles

○ ○ ○ ○ ○

Now we want to divide these into 1 variables (i.e. bars)

○ ○ | ○ ○ ○ → We only needed 1 bar, as it divides the circles into 2 groups

Now we have 5 circles and 1 bar and we want to place these 5 circles in 6 slots, so the ans would be ${}^6C_5$

**Prob** $x+y+z=3$

○ | ○ | ○ 3 circles, 2 bars

Ans = ${}^5C_3$ (as we have 5 blocks and we want to place 3 circles in that)

So for Stars = N, bars = K-1, ans is ${}^{N+K-1}C_N$ or ${}^{N+K-1}C_{K-1}$

**Prob 2** Choose n marbles, k unique color marbles and pick atleast one of each

$A_1 + A_2 + \dots + A_k = N$ where $A_i \geq 1$

To apply Stars and bars we need $A_i$ to be non-ve i.e. be $\geq 0$

This can be done by subtracting 1 i.e. $A_i - 1 \geq 0$

So we get $a_1 + a_2 + \dots + a_k = N-k$, where $a_i \geq 0$

Stars = N-k
bars = k-1
${}^{N-k+k-1}C$

And now Stars = N-k,
bars = k-1

$${}^{(N-k)+(k-1)-1}C_{k-1} = {}^{N-1}C_{k-1}$$

# Bit Manipulation

## Left and Right Shift

N = 22


Left shift (multiply by 2) = Move bits left, using <<.

So N << 1, means $N * 2^1 = 44$

N << 2, means $N * 2^2 = 88$

Right shift (divide by 2) - Move bits left, using >>

N >> 1, means $N/2^1 = 11$

N >> 2, means $(N/2)/2 = 5$

$= N/2^2$

## Checking for $i^{th}$ Set bit

E.g. N = 12 (1100), i = 0 ans = false

i = 3 ans = true

Steps
- Create a new number = 1
- Left shift new number, so that the '1' is moved to $i^{th}$ bit
- Take AND between this num & orig num
- If result = 0, then bit was not set.

```
def check_bit(n,i):
    f = 1 << i
    res = f & n
    if res == 0:
        return False
    return True
```

⇒

```
if n & (1<<i):
    return True
return False
```

## Count number of Set bits

E.g. N = 12 (1100), out = 2

Steps
- Do right shifts and in each step do AND with 1
- If res is 1, then increment
- Continue till num becomes 0

```
def count_set_bits(n):
    count = 0
    while n > 0:
        if n & 1:
            count += 1
        n = n >> 1
    return count
```

Time = O(log n)

Prob: Calculate pair sum XOR of all elements of array i.e.

$(A_1+A_1)\wedge(A_1+A_2)\wedge\ldots\wedge(A_1+A_n)\wedge$
$(A_2+A_1)\wedge(A_2+A_2)\wedge\ldots\wedge(A_2+A_n)\wedge$
$\vdots$
$(A_n+A_1)\wedge(A_n+A_2)\wedge\ldots\wedge(A_n+A_n)$

Sol: We can ignore lower triangle as $(A_2+A_1)\wedge(A_1+A_2)$ would be 0.

e.g. arr = [4, 3, 9, 1]

Sum =

| | | | |
|---|---|---|---|
| 8 | 7 | 13 | 5 |
| 7 | 6 | 12 | 4 |
| 13 | 12 | 18 | 10 |
| 5 | 4 | 10 | 2 |

→ this is the only thing we need to calculate

So ans is XOR of every element in array twice.

Prob: Total sum of all pair XOR

$(A_1\wedge A_2)+(A_1\wedge A_3)+\ldots+(A_1\wedge A_n)+$
$(A_2\wedge A_3)+\ldots+(A_2\wedge A_n)+$
$\vdots$
$(A_{n-1}\wedge A_n)$

Sol: e.g. arr = [5, 3, 9]

| 5 | 3 | 9 |
|---|---|---|
| 0101 | 0011 | 1001 |

$5\wedge 3 = 0110$

$5\wedge 9 = 1100$

$3\wedge 9 = 1010$

Total Sum = $0*2^0 + 2*2^1 + 2*2^2 + 2*2^3$

1 → there are 0 numbers with 0 bit set

2 → 2 numbers with 1st bit set

Next, we need to know how many pairs there are with $i^{th}$ set bit. To do this we just calculate the numbers have 0 at position i & 1 at position i.

So number of pairs with $i^{th}$ set bit would be = $x * y$ (x: num 0's, y: num 1's)

So this solution is $O(n \log n)$.

Implementation: Maintain an array for count of 1's at each index. Traverse the numbers and update this array. Then you just need to calculate total sum from this array (as num 0's = n - arr[i])

**Prob** find missing and repeating number in an array, where all other integers are unique

e.g. [1,3,2,5,5,6,7] missing = 4, repeating = 5

To do in O(1) space & O(n) time

Step 1: XOR all elements of array with index range from 1 to n

Res = (1^3^2^5^5^6^7)^(1^2^3^4^5^6^7)

= 4^5

You get XOR of missing and repeating

Step 2: Next we need to separate out the missing and repeating number from above XOR

4 100

5 101

4^5 001

↳ find a set bit. We do this because for this bit the bit of missing and repeat would be different

Step 3: Next, we partition the input array based on that bit (i.e. 0th bit in this case)

L = 1^3^5^5^7 R = 2^6

Step 4: Next we XOR 'L' with all the numbers from 1 to N where the 0th bit is set and repeat it for R

L = (1^3^5^5^7)^(1^3^5^7) = 5

R = (2^6)^(2^4^6) = 4

Step 5: Next, identify if 'L = 5' is missing or the repeating value by finding its count in the array.

~~LSB~~ Step 2 Implementation

i.e. find LSB

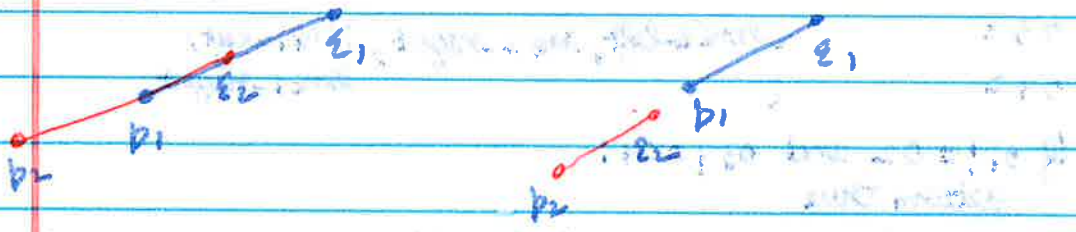
setbit = res & ~(res - 1)

## Find if two lines intersect

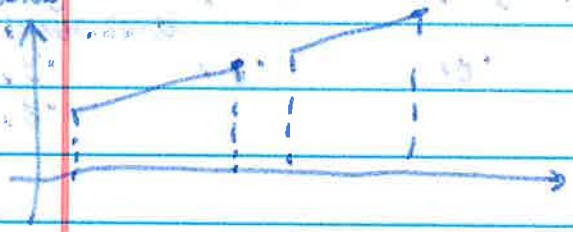
Given 2 line segments, each represented by 2 points $(a_1, b_1)$ and $(a_2, b_2)$.
Find if these intersect


- Find orientation of $p_1$ & $q_1$ w.r.t. point line i.e. $p_2, q_2$. If they are opposite then it means the lines intersect
- If one point is collinear, then it is 'case 2'

  i.e. orientation $(p_2, q_2, p_1)$ and orientation $(p_2, q_2, q_1)$.
- If both the points are collinear then we have 2 cases




In the above case check for their projection on x-axis is 'x' values of all points



```
def orientation(point1, point2, point3):
    res = (point2.y - point1.y)*(point3.x - point2.x) -
          (point3.y - point2.y)*(point2.x - point1.x)
    if res < 0:
        return "anticlockwise"
    if res > 0:
        return "clockwise"
    return "collinear"
```

# Searching Algorithms

## Linear Search

Time = $O(n)$

i.e. run a loop from left to right

works for both sorted and unsorted arrays

## Jump Search

Time = $O(\sqrt{n})$ works for sorted arrays

- Have a block size of $\sqrt{n}$, and you maintain low & high pointer
- At step 0, low = 0, high = $\lfloor \sqrt{n} \rfloor - 1$
- Compare the value at low & high index. ~~If it falls~~
  - ~~If it is inside, then do a linear search~~
  - If it is outside, then low = $\lfloor \sqrt{n} \rfloor$, high = $2\lfloor \sqrt{n} \rfloor - 2$ i.e. move to the next block and repeat

Why $\sqrt{n}$ block size?

Let $n$ = array size

$m$ = block size

~~In worst case we do~~ ~~$\frac{n}{m}$~~ jump there are $\frac{n}{m}$ blocks and ~~$\frac{1}{m}$~~ each block we there are $m-1$ elements

So worst case time = $O\left(\frac{n}{m} + m - 1\right)$. Next we differentiate it to find optimal $m$

$$\frac{d}{dm}\left(\frac{n}{m} + m - 1\right) = -\frac{n}{m^2} + 1 = 0$$

$$n = m^2$$

$$m = \sqrt{n}$$

## Interpolation Search

Given sorted array of 'n uniformly distributed values

- $pos = low + \frac{x - a[l]}{a[h] - a[l]} \times (h - l)$

  $l = 0, h = n - 1$
- If element not found at this position, then move low, high same as binary search

If array uniform (i.e. gap b/w elements is fairly same), then interpolation search faster than binary search.

so we check if substring of length k is valid and if it is, then we can reduce the size of substring
(note: this will take O(n log n) time. this can be done in O(n) using Rodeno's algorithm) :

```
ans = leq
left = 3
right = len(s)
while left <= right:
    middle = left + (right - left) // 2
    if is_valid(S, middle):
        ans = min(ans, middle)
        right = middle - 1
    else:
        left = middle + 1
if ans == leq:
    return 0
return ans
```

~~sorted numbers~~

- given N sorted numbers, you want to select K numbers such that the minimum distance b/w any two of them is maximum

eg. 1, 2, 4, 8, 9 K = 3

way 1: 1, 2, 4 (min = 1)
way 2: 1, 4, 8 (min = 3)
:

Define the monotonic function
f(dist) : it is possible to find k numbers such that min distance b/w is ≥ dist

f(1) = T
f(2) = T
f(3) = T → we want to find the largest dist for which we can select k numbers for which dist ≥ dist 'x' for f(x)
f(4) = F
:
f(∞) = F

it is monotonic because if f(x) = true then f(x-1) is also true.

So given are ~~are~~ permutation 'a' & 'b'. You want to find the number of shifts that will make a & b closest i.e. maximum number of matched integers

e.g. 1 3 2 4
4 2 3 1

Ans = 2 (after 1 right shift on b it is 4 2 3 1 with 2 matched ints

```
pos = {}
for i in range(len(a)):
    pos[a[i]] = i+1
```

Store position of every element

```
freq = {}
for i in range(len(b)):
    current_pos = i+1
    final_pos = pos[b[i]]
    dist = (final_pos - current_pos + n) % n

    if dist not in freq:
        freq[dist] = 1
    else:
        freq[dist] += 1

return max(freq.values())
```

thus +n is used to handle the -ve case i.e. final = 1, current = 3

# Aditya Kumar (Dynamic Programming)

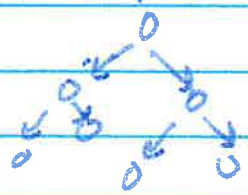
DP = enhanced recursion

Generally in recursion / DP there will

- choice (like can you include an element or not). This is recursion.

Now if only 1 recursive call is being made then DP is not possible

If there are 2 calls then DP is possible, so as to memorize some past outputs



- second is something optimal is being asked like find min, max, largest

So use DP when there is

- choice → can be recursive if only 1 call
- optimal

To solve DP prob

1. First write recursive function
2. Do memorization (i.e. ~~bottom-up~~ top-down)
3. If needed, change to ~~top-down~~ approach i.e. make the ~~bottom-up~~

Recursion → Memorization → Tabulation

Main problems → # subproblems

1. 0-1 knapsack (6)
2. Unbounded knapsack (5)
3. Fibonacci (7)
4. ~~Least~~ LCS
5. LIS (Longest increasing
6. Kadane's algo (6)
7. Matrix chain mul (7)
8. DP on trees (4)
9. DP on grid (14)
10. Others (5)

It is optimal because we are asked for maximum
It has choice because we can either use an item or not
So it is DP.

And now we start by writing Recursive Solution. And then we do memoization and tabulation

0/1 Knapsack Recursive

Input
wt[] : [1|3|4|5]
val[] : [1|4|5|7]
W : 7

Find max profit

Now we can either choose item i or not. Now before recursive code, make a Choice diagram

~~Item~~
Choice diagram

Item i ($w_i$)

$w_i \leq W$ $\quad$ $w_i > W$

[✓] [X] [X]

(We only have 2 choices, as if the ~~it~~ weight of item greater than capacity, then we cannot put it in the bag in the first place when weight is less, we have an option to either use it or not

size of array

int (int wt[], int val[], int W, int n) {

For recursive func
→ Base condition
→ Choice diagram

~~For~~
Base condition
→ Never run the recursion i.e:
$f(n) \rightarrow f(n-1) \rightarrow \dots \rightarrow f(1)$
And see where the recursion ends

→ Instead think of the ~~valid~~ smallest valid input
For input we have
wt[] → smallest of array can be a zero size array
wt[]
W → ~~Sma~~ Smallest weight comb. : 0

Now we only need to include these variables in the dp table.

dp [n+1] [w+1] = -1 (we initialized everything to -1)

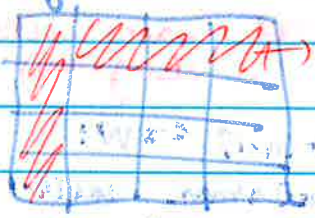
And now in the recursive function whenever we make a recursive call, we check if that value is already present in the dp table.

```
dp = [[-1 for _ in range(W+1)] for _ in range(n+1)]
def knapsack(weight, val, W, n):
    if n==0 or W==0:
        return 0
    if dp[n][W] != -1:
        return dp[n][W]
    if weight[n-1] <= W:
        dp[n][W] = max(val[n-1] + knapsack(...),
                       knapsack(...))
        return dp[n][W]
    dp[n][W] = knapsack(weight, val, W, n-1)
    return dp[n][W]
```

The only drawback of memoization is that the stack space can get full.

## Bottom-up approach

In this we have the same matrix and first fill some base cases and then start filling the other values




In the last two memoization, we will convert recursion to bottom-up

Recursion → Memoization (top-down)

→ Tabulation (bottom-up)

Again, we first identify the size of the dp table by checking how many variables change in the recursive calls.

# 0-1 Knapsack

```
dp = [[0 for _ in range(W+1)] for _ in range(n+1)]
for i in range(1, n+1):
    for w in range(1, W+1):
        if wt[i-1] > w:
            dp[i][w] = dp[i-1][w]
        else:
            dp[i][w] = max(val[i-1] + dp[i-1][w-wt[i-1]],
                           dp[i-1][w])
return dp[n][W]
```

i-1 = current item

→ if weight of current element greater than capacity

Time = O(nW), Space = O(nW)

```
dp = [0 for _ in range(W+1)]
for i in range(1, n+1):
    curr_dp = [0 for _ in range(W+1)]
    for w in range(1, W+1):
        :
    for i in range(len(dp)):
        dp[i] = curr_dp[i]
return dp[W]
```

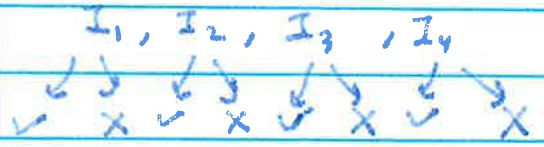
Time = O(nW), Space = O(W)

## Identification of knapsack problem

Input: wt[]:
val[]:
W: capacity

wt[], val[] → are arrays of the same item. So if we are given a single array then it should be wt array.

We were given items $I_1, I_2, I_3, I_4$ and we were to choose if we should select an item or not




This is the knapsack pattern

if arr[i-1] <= j : OR
dp[i][j] = max(val[i-1] + t[i-1][j - val[i-1]], dp[i-1][j])

else:
dp[i][j] = dp[i-1][j]

val does not exist here

Subset sum

```
dp = [False for _ in range(sum+1)]
dp[0] = True
for i in range(1, N+1):  curr_dp = [True for _ in range(sum+1)]
    for s in range(1, sum+1):
        if arr[i-1] > s:
            curr_dp[s] = dp[s]
        else:
            curr_dp[s] = dp[s-arr[i-1]] or dp[s]
    for k in range(len(dp)):
        dp[k] = curr_dp[k]
return dp[sum]
```

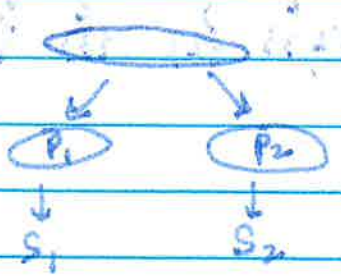
Time = O(N * sum)
Space = O(sum)

Equal Sum partition

Given an arr, divide the arr into 2 subsets whose sum is equal

arr[]: {1, 5, 11, 5}
output: T/F

E.g. {1, 5, 5}, {11}




$S_1 + S_2 = S$

As $S_1 = S_2$, then

$S_1 + S_2 = 2S$ i.e. Sum of array should be even

Count subsets with given sum

```
dp = [0 for _ in range(sum+1)]
dp[0] = 1
for i in range(1, n+1):  curr_dp = [0 for _ in range(sum+1)]
    for s in range(0, sum+1):
        if arr[i-1] > s:
            curr_dp[s] = dp[s]
        else:
            curr_dp[s] = dp[s - arr[i-1]] + dp[s]
    for k in range(sum+1):
        dp[k] = curr_dp[k]
return dp[sum]
```

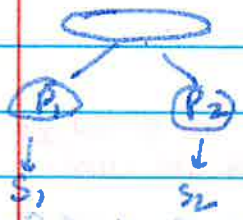
as sum can be zero also ~~if array has element~~

take modulo of this sum if needed

## Minimum subset sum diff

Input arr[]: 1, 6, 11, 5

Output = 1




Here $S_1 - S_2$ should be minimum

- What can be the range of $S_1, S_2$? We can take empty set and set with all elements as the extreme to get range of $S_1, S_2$ as [0, sum(arr)]

So the sums can be


Consider arr = {1, 2, 7}

Range = [0, 10]


→ 4, 5, 6 can never occur as sum of any subset

→ ~~because~~ So our range of $S_1 | S_2$ is {0, 1, 2, 3, 7, 8, 9, 10}

→ Further because $S_1, S_2$ occur in pairs, ~~(a+b)~~ (0, 10) i.e. if $S_1 = 0$, $S_2$ has to be 10, we can further reduce the range as

$S_1$ = {0, 1, 2, 3}, $S_2$ = sum(arr) − $S_1$

Now $S_1 - S_2 = diff$

Also, $S_1 + S_2 = sum(arr)$

$2S_1 = diff + sum(arr)$

$S_1 = \frac{diff + sum(arr)}{2}$

## Target Sum

arr : 1, 1, 2, 3

Sum : 1

Assign + or - in front of each number to make the sum of array equal to "sum"

1, 1, 2, 3

+1, +3 (1, 3) | -1, -2 (1, 2)

We divided the array into two subsets whose sum difference is the given sum.

i.e. $S_1 - S_2 = sum$

## 2. Unbounded Knapsack

- Rod cutting
- Coin change ( number of ways or number of coins)
- Maximum ribbon cut

### 0/1 Knapsack

wt: 0 0 0 0 0 0

wt[i] → val[i]

We consider every item only once

X ✓ ✓ X

~~In unbounded~~ Knapsack, multiple occurrences of the same item are allowed

The general idea if you have an item, you can either have it or not. In case you do not have it, then there is no need to consider this item again as the ans would always be to not have it. If you decide to have it, then you further have a choice to either have it for the second time or not.

# Stack (Aditya Verma)

Same problem
- Nearest greater to left
- Nearest greater to right / Next largest element
- Nearest smaller to left / Nearest smaller element
- Nearest smaller to right
- Stock pair problem
- Maximum area of histogram
- Max area of rectangle in binary matrix

Problems
- Rain water trapping
- Implementing a min Stack (extra and no-extra space)
- Implement Stack using heap
- The celebrity problem
- Longest valid parenthesis
- Iterative tower of hanoi

## Identification

- Have an array (if you get a feeling for sort then use heap)
- The brute force $O(n^2)$ would look like

```
for i in range(0, n):
    for j in range(i, n):      # i.e. a dependent for-loop
        ...
```

## Nearest greater to right

arr = [1, 3, 2, 4]

out = [3, 4, 4, -1]

Brute force:

```
for i in range(0, n-1):
    for j in range(i+1, n):
        ...
```

Now to implement it using Stack, consider the example,

1, 3, 0, 0, 1, 2, 4

Now for 3 you need to check 0, 0, 1, 2, 4 from left to right. So if you want to use a Stack then 0 should be the first element that should be popped. ∴ We should add elements in Stack from right such that 0 is the first element to be popped.

~~for i in range(n):~~

Now we want i-1 index to be top of stack. So start adding elements to stack from index 0.

Conditions:
- Stack empty → append
- stack.top() >= arr[i] → stack.pop() till empty
- stack.top() <= arr[i] → stack.pop() till → top > arr[i]

Only change in code is
- for i in range(n):
- return ans

→ i.e. start loop from left to right and no need to reverse now

## Stock Span problem

arr: 100, 80, 60, 70, 60, 75, 85 → These are prices of the stock for each day.

For each index output consecutively
→ number of smaller or equal values on left i.e. previous days

e.g. ~~100~~ 80 60 70 60 75 85

70 → This does not count

for 85, there are 60, 70, 60, 75

Brute force:
```
for i in range(n):
    for j in range(i-1, -1, -1):
```

For stack, we store building it from index 0, as i-1 should be on top. And we are essentially finding the index of the nearest greater to left and then subtract that from the current index

~~s.empty()~~ → -1

~~s.top~~

Stack conditions:
- s.empty() → -1
- s.top()[0] > arr[i] → i - s.top()[1]
- s.top()[0] <= arr[i] → s.pop() till → s.empty() or s.top()[0] > arr[i]

So we breake the 2D array into 4 histograms and we find the maximums over all the 4 histograms and that is the ans.

## Sort a K sorted array / nearly sorted array

arr[]: 6, 5, 3, 2, 8, 10, 9

K sorted means the value at index 'i' can be present in the range $[i-k, i+k]$ in the sorted array.

For index 0, 3 only need to consider 6, 5, 3, 2

For index 1, 3 only need to consider 6, 5, 3, 8 (because we already placed 2)

So we are only considering 4 elements at a time i.e. k+1

- So we add k+1 elements to a min-heap
- Pop the min value
- Add the value of next index to min-heap
- Repeat

## K closest numbers

arr: 5, 6, 7, 8, 9

K = 3, X = 7

i.e. find K values from 'arr' closest to X

Algo

- Subtract X from all elements of arr and take absolute value of that.

  5, 6, 7, 8, 9

  − 7

  2, 1, 0, 1, 2

- Next find the K-smallest numbers from (2, 1, 0, 1, 2), which gives 1, 0, 1
- Then we get the numbers 6, 7, 8 from these

In heap implementation, we store a pair now.

# Sliding window

Problem: arr: 2, 3, 5, 2, 9, 7, 1 and a number = 3 i.e. size of sub-array (continuous part)

Sub arrays for above are

2 3 5
3 5 2
5 2 9
2 9 7
9 7 1

→ find their sum and return max

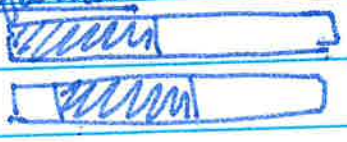
Brute force is

```
for i in range (0, N):
    for j in range (i, i+k):
```

Next, you have to check

- if you are doing repetitive work

Identification



We are given ~~array + Sliding~~

- array) String
- Subarray) Substring
- Smallest / largest
- k i.e. window size

- you can also be asked to find the largest and smallest window

Sliding window

→ fixed size

→ variable size (will have to use list / map also)

```
need = Set(pat)
count = len(need)

freq = {}
for c in pat:
    if c not in freq:
        freq[c] = 0
    freq[c] += 1
out = 0
count = len(freq)
for i in range(len(pat)):
    if s[i] in freq:
        freq[s[i]] -= 1
        if freq[s[i]] == 0:
            count -= 1

if count == 0:
    out += 1
for i in range(len(pat), len(s)):
    if s[i - len(pat)] in freq:
        freq[s[i - len(pat)]] -= 1
        if freq[s[i - len(pat)]] == 0:
            count += 1
    if s[i] in freq:
        freq[s[i]] += 1
        if freq[s[i]] == 0:
            count -= 1
    if count == 0:
        out += 1
return out
```

Get frequencies of chars in pattern

Initialize Sliding window

Remove [i - len(pat)] from window

Add [i] to window

The logic for count += 1 might be incorrect

# Recursion

## Introduction

- Goal is to make input smaller. We do not do this directly, but instead we ~~do a prism~~ take some decisions and based on these the input becomes small.


- Next, we need a decision space. Based on the choices and decisions, we can identify if the problem is recursive.
- Create the recursive tree (which is the solution to the problem)

e.g. Subset problem "abc" → "", a, b, c, ab, bc, ac, abc

- First we write all the possible solutions for all inputs in a table

| | a | b | c |
|---|---|---|---|
| "" | 0 | 0 | 0 |
| a | 1 | 0 | 0 |
| b | 0 | 1 | 0 |
| c | 0 | 0 | 1 |
| ab | 1 | 1 | 0 |
| bc | 0 | 1 | 1 |
| ca | 1 | 0 | 1 |
| abc | 1 | 1 | 1 |

Using this table, we find out what choices and decisions we want to take

~~For~~ For each input here we have a binary choice to either take it or not and use this info to fill the table

~~Now~~ Now we use a recursive tree to convert the table into a meaningful information.

## If-DP method

To generate recursive tree

## Multithreaded Algorithms (CLRS)

A simple example of multi-threaded program can be illustrated using Fibonacci numbers

```
Fib(n):
    if n ≤ 1
        return n
    else
        x = spawn Fib(n-1)
        y = Fib(n-2)
        sync
        return x+y
```

In this example we created a new thread to do the computation as the computation of x is independent from y.
And then we wait to get the results from all threads by syncying

Note :- It is upto the scheduler to decide which thread to run next and using spawn here only means 'logical parallelism'.

We can have shared memory or distributed memory. In the above fibonacci example there is explicit synchronization by using 'sync' and then there is implicit parallelism when every procedure returns, thus ensuring that all its children terminate before it does.

Multithreaded computation can be thought of as a directed acyclic graph (called a computation dag)

### Measuring performance

Work and Span

Work = Total time to execute the entire computation on a single processor
(for a DAG where each strand takes unit time, the work is the number of vertices in the dag)

Span = Longest time to execute the strands along any path in the dag
(for a DAG where each DAG takes unit time, the Span equals the number of vertices on a longest or critical path in the dag)

### Notation

P = number of available processors
$T_p$ = running time on P processors
Work = $T_1$
Span = $T_\infty$
} provide lower bound on $T_p$

Race conditions occur when two logically parallel instructions access the same memory location and atleast one of the instruction performs a write.

## Analysis Brute force Matrix Multiplication

```
Multiply-Matrix(A,B):
    n = num_rows
    parallel for i = 1 to n
        parallel for j = 1 to n
            for k = 1 to n
                c_ij = c_ij + a_ik · b_kj
```

$T_1 = \Theta(n^3)$ (treat it as a single processor)

$T_\infty = \Theta(n)$ Each thread has a loop from 1 to n

$\frac{T_1}{T_\infty} = \Theta(n^2)$

If you try to do this on the divide and conquer approach then parallelism would be $\Theta\left(\frac{n^3}{\log^2 n}\right)$ which is worse than above

## Analysis Merge Sort

$T_1 = (n \log n)$

$T_\infty(n) = T_\infty(\frac{n}{2}) + \underbrace{O(n)}_{\text{for merge step}} = O(n)$

$\underbrace{\quad}_{\text{we split into half size}}$

So parallelism = $O(\log n)$ which is really bad. (To sort 10 million numbers it might achieve linear speedup of a few processors)

## Distributed algorithm to determine median of arrays located on different computers

- Let there be a master node.
- First query all servers to know the size of dataset, then we know we are looking for $k = n/2$ largest element.
- Master selects a random server and queries it for a random element. Then broadcast it to all the servers and each server partitions its elements into larger than or equal to and those smaller than broadcasted element
- Each server tells the master the size of its array. If this value $> k$ then master instructs servers to discard smaller array. If it is $< k$, then master indicates to disregard the larger than set

  If $= k$, then pivot element is median

  $O\left(\frac{n}{S} + S \log \frac{n}{S}\right)$ where S: num servers,

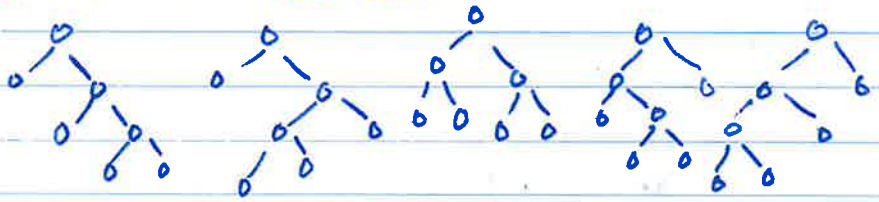
  when $S << \text{sqrt}(n)$ then it is $O\left(\frac{n}{S}\right)$

# Recursion

## 894. Print All Possible Full Binary Trees

Given an integer 'n', print a list of all full binary trees with n nodes.
A full binary tree has 0 or 2 children.

Sol n = 7, out =



```
dp = {0:[], 1:[Node()]}
def backtrack(n):
    if n in dp:
        return dp[n]

    res = []
    for l in range(n):
        r = n-l-1
        leftTrees = backtrack(l)
        rightTrees = backtrack(r)
        for t1 in leftTrees:
            for t2 in rightTrees:
                res.append(Node(0, t1, t2))
    dp[n] = res
    return res
return backtrack(n)
```

You place 'l' nodes on left and 'r' nodes on right

After you got left and right trees
[l1, l2, l3, ...]
[r1, r2, r3, ...]
you can add them in any combination
(l1, r1), (l1, r2), ..., (l2, r1), ...

There is one problem with this solution, that the nodes are being shared being shared between trees. This can cause memory leaks and deleting nodes can become messy. So to avoid this, you have to make a deepcopy of 't1', 't2' before adding them to 'res'. And same when you return 'dp[n]'

## 1823. Find the Winner of the Circular Game

n friends from 1 to n. Start from '1' and count the next k friends (including the start) and remove him. Start from the next person after removed and repeat till only 1 friend is left.

Sol If you know who won (i-1) rounds, then you can compute the person who won the $i^{th}$ rounds as new = (old + k) % i

```
out = 0
for i in range(1, n+1):
    out = (out + k) % i
return out + 1
```

0-index to 1-index

```
dp = memo[i]
if dp[amount_left][coupons_left] != -1:
    return dp[amount_left][coupons_left]

if coupons_left > 0:
    # do not apply coupon
    if price[i] <= amount_left:
        val1 = max(
            tastiness[i] + recursion(i+1, amount_left - price[i], coupons_left-1),
            recursion(i+1, amount_left, coupons_left)
        )
    else:
        val2 = recursion(i+1, amount_left, coupons_left)

    # apply coupon
    new_price = price[i] // 2
    if new_price <= amount_left:
        val2 = max(
            tastiness[i] + recursion(i+1, amount_left - new_price, coupons_left-1),
            recursion(i+1, amount_left, coupons_left)
        )
    else:
        val2 = recursion(i+1, amount_left, coupons_left)

    dp[amount_left][coupons_left] = max(val1, val2)
    return dp[amount_left][coupons_left]
else:
    # do not apply coupon
    if price[i] <= amount_left:
        dp[amount_left][coupons_left] = max(
            tastiness[i] + recursion(i+1, amount_left - price[i], coupons_left-1),
            recursion(i+1, amount_left, coupons_left)
        )
        return dp[amount_left][coupons_left]

    dp[amount_left][coupons_left] = recursion(i+1, amount_left, coupons_left)
    return dp[amount_left][coupons_left]
return recursion(0, maxAmount, maxCoupons)
```

Recursion

```
def reverse(left, right, level):
    if left is None and right is None: return
    if level % 2 == 0: swap(left.val, right.val)
    reverse(left.left, right.right, level+1)
    reverse(left.right, right.left, level+1)
```

## 2434. Using a Robot to Print the Lexicographically Smallest String

given 's', you can remove first char and add it to 't' or you can remove the last char from 't' and add it to 'p'. Return the lexicographically smallest 'p' that can be formed.

Logic s = bcaadaz , t = '' , p = ''
s = caadaz , t = b , p = ''
s = aadaz , t = bc , p = ''

Our objective is to add the smallest characters first i.e. all the a's
s = adaz , t = bc , p = a
s = daz , t = bc , p = aa
s = az , t = bcd , p = aa
s = z , t = bcd , p = aaa

Now we see that d < z , so we can add it to p and repeat
s = z , t = bc , p = aaad
s = z , t = b , p = aaadc
s = z , t = '' , p = aaadcb
s = '' , t = '' , p = aaadcbz

```
count = Counter(s) , lo = 'a', p = [], t = []
for c in s:
    t += c
    count[c] -= 1
    while lo < 'z' and count[lo] == 0:
        lo = chr(ord(lo)+1)
    while t and t[-1] <= lo:
        p += t.pop()
return "".join(p)
```

] find the next smallest available character in 's' on right of given index

Binary Search Trees (BST) and Binary Indexed Tree (BIT) can help in this situation.

In BST, we define 'count' on Node which is equal to the number of elements in the subtree that are greater than or equal to current node val.

```
class Node:
    def __init__(self, val):
        self.val = val
        self.count = 1
        self.left, self.right = None, None

def search(root, val):
    if root == None: return 0
    if val == root.val: return root.count
    if val < root.val: return root.count + search(root.left, val)
    return search(root.right, val)

def insert(root, val):
    if root == None:
        root = Node(val)
    elif val == root.val:
        root.count += 1
    elif val < root.val:
        root.left = insert(root.left, val)
    else:
        root.count += 1
        root.right = insert(root.right, val)
    return root

def reverse_pairs(nums):
    root = None
    res = 0

    for x in nums:
        res += search(root, 2*x+1)
        root = insert(root, x)
    return res
```

we want to find the number of nodes with value ≥ current value ~~the elements~~

we add this because right subtrees contain values ≥ current val (and this is stored in root.count)

nums[i] > 2* nums[j]
Search for all elements no less than twice the current element plus 1
i.e. we are finding ≥ 2* nums[x] + 1

## 2406. Divide intervals into minimum number of groups

Given a 2D array intervals[i] = [left_i, right_i] (inclusive). You have to divide the intervals into one or more groups such that each interval is in exactly one group, and no two intervals that are in the same group intersect each other.

Return the minimum number of groups needed.

Eg. [5,10], [6,8], [1,5], [2,3], [1,10]

↳ 3 i.e. [1,5], [6,8]
[2,3], [5,10]
[1,10]

Two Solutions
- Use priority queue
- Order interval (discussed)

- Split the interval into start and end arrays and sort them
- You lost the original order, but that does not matter as you are only concerned with finding overlaps.

```
used_rooms = 0
start = sorted([i[0] for i in intervals])
end = sorted([i[1] for i in intervals])
start_pointer, end_pointer = 0, 0

while start_pointer < len(intervals):
    if start[start_pointer] >= end[end_pointer]:
        used_rooms -= 1
        end_pointer += 1
    used_rooms += 1
    start_pointer += 1
return used_rooms
```

If there is a meeting that has ended by the time the meeting at 'start_pointer' starts then it means they can run in the same room i.e. we do not need to change 'used_rooms'

If start and end are inclusive then remove =

## 2439. Minimize Maximum of Array

Given a non-negative array. In one operation, you do
- If nums[i] > 0, then nums[i] -- and nums[i-1] ++

Return the minimum possible value of the maximum integers after performing any number of operations

**Prob** Return index of two numbers such that they add to 'target' from an array

**Sol** As only one unique pair is ans, go from left to right and find if 'target - a' is in your mounted dict

```
dict = { }
for i in range(len(nums)):
    if nums[i] in dict: return [dict[nums[i]], i]
    else: dict[target - nums[i]] = i
```

**Prob**: Given a string, find the length of the longest substring with repeating chars

**Sol** Iterate from left to right where to maintain a hash map of all the characters seen and the index at which they last occured. Now you can maintain a left pointer 'start' that you can use to filter out values that are not repeated

```
start = 0
max_length = 0
map = { }
for i in range(len(s)):
    if s[i] in map and start <= map[s[i]]:        # this is key
        start = map[s[i]] + 1
    else:
        max_length = max(max_length, i - start + 1)
    map[s[i]] = i
```

You can also use an array instead of hash map. As ASCII chars ~~only take~~ are only 128 chars

**Prob** Median of two sorted arrays in $O(\log(\min(N,M)))$

**Sol** Consider 2 arrays

$x \rightarrow x_1\ x_2 \mid x_3\ x_4\ x_5\ x_6$

$y \rightarrow y_1\ y_2\ y_3\ y_4\ y_5 \mid y_6\ y_7\ y_8$

Do a partition in x b/w $x_2$ & $x_3$. Now because we want to find median, then both the left and right sides should be of equal length. So, we have to partition y b/w $y_5$ & $y_6$ so that left & right arrays have length = 7

For the median, we observe if

$x_2 \le y_6$ & $y_5 \le x_3$ then all elements in left array are smaller than all elements in right array. So the median is among $x_2, y_5, x_3, y_6$

if even length then median = avg(max($x_2, y_5$), min($x_3, y_6$))

else = avg($x_2, y_5$)

Do binary search on the shorter array. If the above condition meets you found median, else if max on left > min on right, then move left in binary search

else move right

```
digit = ord(num[i]) - ord("0")
if check_next and digit > max_digit:        → Number overflowed
    return MAX if is_positive else MIN
if digit < max_digit:
    check_next = False
int_num = 10 * int_num + digit
return int_num if is_positive else -int_num
```

The same solution can be used to reverse a 32-bit signed integers. The only thing that changes is the calculation of 'digit' as follows

```
digit = x % 10
x = x // 10
```

Also, check the length of given num, if length is less than max_length then simply reverse it, no need for any checks

**Prob** Merge two sorted arrays

**Sol**

```
if len(nums1) > len(nums2):                 → first array
    nums1, nums2 = nums2, nums1
out = []
left
```

```
out = 0
first = 0, second = 0        → index of given 2 arrays
while True:
    if first == len(nums1):
        for i in range(right, len(nums2)):
            out.append(nums2[i])
    if second == len(nums2):
        ....
    if nums1[first] <= nums2[second]:
        out.append(nums1[first])
        first += 1
    else: out.append(nums2[second])
        second += 1
```

If you reach end of one array then add all elements of next array

**Prob** Find longest palindrome substring

**Sol** For base case, the answer is always 1. Then you need to have two separate cases for odd and even length palindrome. First get the indexes of all length = 2 palindrome and length = 3 palindrome. Then you can check for palindromes at only these indexes.

An optimization would be to do inorder traversal on both trees simultaneously. When we want to add a value to the final list, add the smallest value and only go up the tree of the smallest values. This ~~combines~~ sort of combines the inorder traversal and merge step.

Prob: Get maximum & second maximum of an array

Sol:
```
max_1, max_2 = -1e9, -1e9        (assume max_1 ≥ max_2)
for i in range(len(nums)):
    if nums[i] ≥ max_1:
        max_2 = max_1
        max_1 = nums[i]
    elif nums[i] > max_2:
        max_2 = nums[i]
```

Prob: Given unsorted array, print the indexes of a target value in the sorted array

Sol: E.g. [1,2,5,2,3], target = 2, Out = [1,2]
Just count the number of values smaller than given value. And then the target value should be the next indexes.

Prob: Given a list [id, score], find the average of top-5 maximum score of each id.

Sol: The naive solution is to sort and then average the top-5 values in O(n log n)
You can also use a priority queue / min-heap to construct the heap and then get the 5 maximum values in O(n)

~~Prob: use binary search to get first last occ~~

Prob: Construct a min-heap / priority queue, where the first priority is given to lower value and if values are equal priority is given to lower index i.e. [[2,1], [2,3]], Here 2 is the value and we want to return the minimum value. If there is a tie, look in the next dimension and return the '2' with the lower value in second dim

Sol: For this solution, we add keys as they are given i.e. we do not convert an array to min-heap ~~too~~

Prob: given a list, return a subsequence such that the sum of elements in the subsequence is greater than the sum of elements not included in the list.

Sol: inp = [4,3,10,9,8] , Out = [10,9]

you can sort the input and run a for loop from the end. Or you can use a priority queue to keep getting the largest value, till the sum becomes greater than the sum of other elements.

Prob: given a m x n matrix which is sorted in non-increasing order both row-wise and column-wise, return the number of negative numbers.

Sol: It forms a stair

```
+ + + + -
+ + - - -
+ - - - -
```

. So you can start from bottom left and count in O(m+n)

```
for i in range (len(grid), -1, -1):
    for j in range (left_start, len(grid[0])):
        if grid[i][j] < 0:
            count += len(grid[0]) - j
            left_start = j
            break
```

Prob: Find the peak of mountain in an array [0,1,2,3,4,3,2,1] (ans: 4)

Sol:

```
left = 0, right = len(arr) - 1
while left <= right:
    middle = left + (right - left) // 2
    if arr[middle] > arr[middle+1] and arr[middle] > arr[middle-1]:
        return middle
    if arr[middle] < arr[middle+1]:
        left = middle + 1
    else:
        right = middle - 1
```

Prob: given sorted array, find the smallest 'i' that satisfies arr[i] == i

Sol: For the smallest index, we need to continue the search towards left

```
potential_ans = -1
while left <= right:
    middle = ...
    if arr[middle] == middle:
        potential_ans = middle
        right = middle - 1
    elif arr[middle] > middle:
        right = middle - 1
    else: left = middle + 1
return potential_ans
```

```
left = 0, right = len(arr) - 1
while left <= right:
    middle = left + (right - left) // 2
    missing = arr[middle] - middle - 1
    if missing < k:  left = middle + 1
    else:            right = middle - 1
return left + k
```

For return: right → values missing before arr[right] − right − 1 & left = right + 1. So value to return in arr[right] + k − (arr[right] − right − 1)

**Prob:** Reverse bits of a given 32 bits unsigned number

**Sol:** E.g. 1000 → 0001 (this is for 4 bit number)

The standard O(32) soln is

```
out = 0
for i in range(32):
    bit = n & 1
    n = n >> 1
    if bit:
        out = out + 2**(31 - i)
return out
```

An O(1) optimized solution is as follows

```
n = ((n & 0xffff0000) >> 16) | ((n & 0x0000ffff) << 16)
n = ((n & 0xff00ff00) >> 8) | ((n & 0x00ff00ff) << 8)
n = ((n & 0xf0f0f0f0) >> 4) | ((n & 0x0f0f0f0f) << 4)
n = ((n & 0xcccccccc) >> 2) | ((n & 0x33333333) << 2)
n = ((n & 0xaaaaaaaa) >> 1) | ((n & 0x55555555) << 1)
return n
```

The idea is pretty simple, exchange bits in powers of two

n & 0xffff0000 → selects first 16 bits → Shift these to left by 16

n & 0x0000ffff → selects last 16 bits → Shift these to right by 16

Then take the 'or' to combine them.

E.g. 1010 0000 → 0000 1010

(first 16 bits → shift to)

And then you repeat it for smaller value i.e. exchange first 8 bits by next 8 bits and so on.

Prob: Find the missing number in an array containing numbers in range [0,n]
i.e. [0,3,2] → 1 is missing

Sol: Sum

```
c = len(nums) * len(nums+1) // 2
c = c - sum(nums)
return c
```

the problem with sum approach is that it will overflow

XOR

```
res = len(nums)
for i in range(len(nums)):
    res = res ^ i
    res = res ^ nums[i]
return res
```

e.g. 0 1 2 3 4 → [0,n]

0 2 3 4 → nums array

XOR(0,0) = 0, XOR(2,2) = 0, ...

So only 1 will remain

Binary Search

```
left = 0, right = len(nums) - 1
while left <= right:
    middle = left + (right - left) // 2
    if nums[middle] > middle:
        right = middle - 1
    else: left = middle + 1
return left
```

Preferred over XOR when the array is sorted.

Prob: Given an integer array of digits (with duplicates). Find all the unique 3 digit numbers that can be formed with these digits (number must be even) & output should be sorted and contain unique elements

Sol digits = [2,1,3,0]

out = [102, 120, 130, 132, 210, 230, 302, 310, 312, 320]

```
count = [0 for _ in range(10)]
for d in digits: count[d] += 1
out = []
for i in range(1,10):
    for j in range(10):
        for k in range(0,10,2)
            if count[i] > 0 and
               count[j] > (i == j) and
               count[k] > (k == i) and (k == j):
                num = 100*i + 10*j + k
                out.append(num)
```

get count of every digit

→ First digit cannot be 0

→ We need even numbers

→ Count of first digit should be > 0

→ For second digit the count should be > 0, but when the first & second digits are same the count should be > 1

Prob: Find ~~product of~~ 3 numbers whose product is maximum. Nums can be negative

Sol: prod = max ~~(max1, max2, max3, max1,~~
(max1 * max2 * max3 , max1 * min1 * min2)

max1 >, max2 ≥ max3
min1 ≤ min2

Generalizing the solution to k elements

nums.sort() → can use minheap / maxheap to make it O(n log k)

if nums[0] * nums[-1] >= 0: → if there are only negative nums or positive nums in the array
    return prod of last k elements

l = 0, r = len(nums) - 1, prod = 1

while k > 0:
    if k % 2 == 1: → if k = odd, then only option is to take highest element
        prod = ~~nums~~ prod * nums[r]
        r -= 1
        k = k - 1
    else:
        if nums[r] * nums[r-1] > nums[l] * nums[l+1]: → if k = even, then compare 2 elements and take the ~~two which~~ two whose product is greater.
            prod = prod * nums[r] * nums[r-1] ; r = r - 2
        else: prod = prod * nums[l] * nums[l+1] ; l = l + 2
    & k = k - 2

Prob: Find the length of the largest subsequence that has the difference of maximum value and minimum value = 1

Sol: The diff of max - min = 1 implies the subsequence only contains x & x+1. So you can get a count of all values in the input array. And then find the two elements which are k, k+1 with maximum (count[k] + count[k+1])

Prob: Find the sum of digits of a number, & repeat it for the sum, till the sum is a single digit

Sol: num = 38
3 + 8 = 11
1 + 1 = 2 (ans)

Use the observation
179 = 1 + 7 + 9
= 8 + 9 → we need to ignore these 9 i.e. the ans becomes modulo 9 the sum of all digits
= 17
17 = 1 + 7 = 8

if digit_sum == 0: return 0
if digit_sum % 9 == 0: return 9
return digit_sum % 9

Prob: Monotone Stack (used only to find the next greater element in an array)

Sol [2, 3, 5, 1, 0, 7, 3]

[3, 5, 7, 7, 7, -1, -1] → next greater value

This can be done using stack. If the current value > top value, then it means we found the next greater value of top. So pop top and repeat.

```
greater_map = {}
stack = []
for x in nums:
    while len(stack) and stack[-1] < x:
        greater_map.append[stack.pop()] = x
    stack.append(x)
while len(stack):
    greater_map[stack.pop()] = -1
```

This problem can be extended to circular array also. In that case you can duplicate the array and pretend it is a single array. Then repeat this algorithm for the first half.

Or you can use % operator to index, if you don't want to duplicate.

You can also do above for lesser_than case

Eg. j > i and prices[j] <= prices[i]. In this case, suppose '8' is on top and the next value is '6', then by above condition it is the ans. Ily value of '8' is also the ans

```
out = [0 for _ in range(len(prices))]
Stack = []
for i in range(len(prices)):
    while len(stack) and stack[-1][0] >= prices[i]:
        price, index = Stack.pop()
        out[index] = price
    stack.append((prices[i], i))
while len(stack):
    price, index = stack.pop()
    out[index] = price
```

Logic to decide inequality

```
def max_depth (root):
    if root is None:
        return 0
    left_h = self.max_depth (root.left)
    right_h = self.max_depth (root.right)
    return max (left_h, right_h) + 1
```

Iterative version

```
stack = [(root, 1)]
max_height = 1
while len (stack) != 0:
    node, height = stack.pop ()
    max_height = max (height, max_height)
    if node.left is not None:
        stack.append ((node.left, height+1))
    if node.right is not None:
        stack.append ((node.right, height+1))
return max_height
```

Prob: Given an array, you buy a stock on $i^{th}$ day and sell on $j^{th}$ day $j > i$. Find max profit

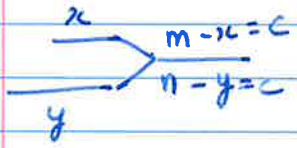
Sol: This is an application of Kadane's algo

```
maximum_profit = 0
current_profit = 0
left_min = prices[0]
for i in range (1, len(prices)):
    if prices[i] < left_min:
        maximum_profit = max (maximum_profit, current_profit)
        current_profit = 0
        left_min = prices[i]
    else:
        if prices[i] - left_min > current_profit:
            current_sum = prices[i] - left_min
return max (maximum_profit, current_profit)
```

Prob: intersection of 2 linked lists in O(m+n) time & O(1) space

Sol: Eg. 4 → 1 → 8 → 4 → 5
5 → 6 → 1 ↗
Ans

If length of first list = m, length of second list = n




traverse first list & when it ends traverse second

$x + c + y$ ~~$= m + n$~~

traverse second list & when it ends traverse first

$y + c + x$

You reach the same node

```
if headA is None or headB is None:
    return None
node1, node2 = headA, headB
while node1 is not node2:
    if node1 is None: node1 = headB
    else            : node1 = node1.next

    if node2 is None: node2 = headA
    else            : node2 = node2.next
return node1
```

Prob: Convert number to excel column number

Sol: 701 → ZY, 1 → A, 27 → AA

```
chars = [chr(x) for x in range(ord('A'), ord('Z')+1)]
out = ""
while num != 0:
    rem = (num-1) % 26
    num = (num-1) // 26
    out += chars[rem]
return out[::-1]
```

Prob: Starting with 'n', make it equal to sum of squares of digits. Repeat till you get 1 or there is a cycle

Soln: With time = O(log n), space = O(log n), you can use the standard hash table approach

```
            if second is not None:
                min_dist = min(min_dist, abs(first - second))
        elif word == word2:
            second = i
            if first is not None:
                min_dist = min(min_dist, abs(first - second))
    return min_dist
```

**Prob** Check if number is a power of 3

**Sol** - the standard solution is to check the bits of number in $O(\log n)$ time

- Find the maximum power of 3 and see if the given number is a multiple of that number $O(1)$ time (only works for prime base)

- Use log method

```
log = math.log10(n) / math.log10(3)
if math.isclose(log % 1, 0):
    return True
return False
```

Here we have to use base 10, as natural log causes rounding error issues

- Similar to above but we check for power, so don't have to worry about rounding error (maybe $O(\log n)$, I don't know time complexity of power)
  ↳ $O(\log \log n)$ with binary exponentiation

```
if n == Math.pow(3, Math.round(Math.log(n) / Math.log(3))):
    return True
return False
```

**Prob**: Find if a stream of strings $s_1, s_2, \ldots, s_k$ are a subsequence of string t

**Sol** e.g. $s$ = "abc" $t$ = "ahbgdc" (the order must be preserved)

Method 1 is to use two-pointers, but for a stream of strings that can be optimized.

We can build a hashmap for t where we store the index at which the character occurs dict = { 'a': [0, 3], 'b': [2], ... }

now for each incoming strings we know where to look for. We pick the minimum index in each list that satisfies the property and keep repeating. So the solution becomes kind of independent of length of 't' (binary search for further optimizing the lookup of index)

- Even if there is only a single thread, or the algorithm has exclusive access to the array while running, the array might need to be reused later or by another thread once the lock has been released.

**Prob** Given an integer array, return all the triplets [nums[i], nums[j], nums[k]] where their sum = 0 (i != j != k and soln should not contain duplicates)

**Sol** First, consider the problem when only 2 numbers need to be found
- Use hash map to get O(n) time and space
- Use 2 pointers on sorted array to get O(n log n) time & O(1) space

Now for 3 values, we can try following approaches
- Brute force in $O(n^3)$ time and O(1) space
- ~~Split the array into +ve & -ve numbers and run 2 for-loops if Use 2 for-loops~~
- Sort the array and use 2-for loops for the first two values and binary search for the third in $O(n \log n + n^2 \log n)$ time, O(1) space
  (sort; first 2 vals; binary search for 3rd val)
- Extend the 2-pointer approach to get $O(n^2)$ time

  Steps:
  1. Sort the array
  2. Loop through array and skip the current val if it is same as previous val (to avoid duplicates)
  3. Use the 2-pointer approach to find the 2 elements whose sum is equal to target - current val

So for interview
① Do the 2-pointer approach
② Follow up might be, how to solve without sorting? For that ~~instead~~ we run the main loop and now use the hash map approach
   The "No-Sort" approach would be efficient when we have a large input with many duplicates.
③ Another follow up, is find the ~~value closest to~~ 3 numbers whose value is closest to the target or smaller than the target
   For this, extend 2-pointer approach

- Create another set to skip duplicates of the ~~the~~ in the outer loop
- Create a dictionary instead of set and set "d[nums[j]] = i" after we are done with the second val. Now we can use this as a check

```
out = set()
first_vals_seen = set()
d = {}

for i in range(len(nums)):
    if nums[i] not in first_vals_seen:
        first_vals_seen.add(nums[i])

    for j in range(i+1, len(nums)):
        target = -(nums[i] + nums[j])
        if target in d and d[target] == i:
            out.add(tuple(sorted((nums[i], nums[j], target))))
        d[nums[j]] = i
return out
```

just an optimization to skip outer loop duplicates

Only changes. By setting d[nums[j]] = index of current iteration, we are saying that we can use this value from the hash map

## 2pointer for Closest sum

Find 3 numbers whose sum is closest to the target (return their sum)

```
nums.sort()
min_diff, closest_sum = 1e9, 0

for i in range(len(nums)):
    if i == 0 or nums[i] != nums[i-1]:
        left, right = i+1, len(nums)-1

        while left < right:
            s = nums[i] + nums[left] + nums[right]
            diff = abs(target - s)
            if diff < min_diff:
                min_diff = diff
                closest_sum = s

            if s == target:
                return s
```

Avoid duplicates for optimization

need to check for every pair

~~value~~

```
def loop(self, start, target, depth, nums):
    if depth == 2:
        left, right = start, len(self.nums)-1
        while left < right:
            if self.nums[left] + self.nums[right] == target:
                self.out.append([*nums, self.nums[left], self.nums[right]])
                left += 1
                right -= 1
                while left < right and self.nums[left] == self.nums[left-1]:
                    left += 1
            elif self.nums[left] + self.nums[right] > target:
                right -= 1
            else:
                left += 1
        return
    for i in range(start, len(self.nums)):
        if i == start or self.nums[i] != self.nums[i-1]:
            self.loop(i+1, target - self.nums[i], depth-1, [*nums, self.nums[i]])
```

~~Prob~~ ~~ps to get depth~~

Prob: get depth of tree

Sol

```
if root is None:
    return 0
max_depth = 1
nodes = [None, root]
while len(nodes) != 1:
    node = nodes.pop()
    if node is None:
        max_depth += 1
        nodes.insert(0, None)
    else:
        for child in node.children:
            nodes.insert(0, child)
return max_depth
```

thunk if root has no children

as nodes = [None] is stopping const

top → O(1)

peekMax → If you use heap then O(1), for binary tree it is O(log n). But for binary tree you can further maintain a pointer to max value and when deleting the node, you can update this pointer

popMax → O(log n)

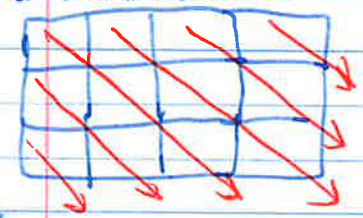
Prob: Given a BST find minimum difference b/w values of any 2 nodes

Sol. Inorder traversal can be used. The space complexity can be reduced to O(h) by storing the previous node visited by inorder traversal

```
self.pre = -1e9
self.min_diff = 1e9
self.traversal(root)
return self.min_diff

def traversal(self, node):
    if node.left:
        self.traversal(node.left)
    self.min_diff = min(self.min_diff, node.val - self.pre)
    self.pre = node.val
    if node.right:
        self.traversal(node.right)
```

Prob: Check if matrix is Toeplitz i.e. every diagonal from top-left to bottom-right has the same elements

Sol



Just check if current value is same as top-left neighbour (as you can get access to all the diagonal elements using top-left approach)

```
for i in range(len(m)):
    for j in range(1, len(m[0])):
        if m[i][j] != m[i-1][j-1]:
            return False
return True
```

~~First second place & Sequence~~

To find out the value in the second place

(1) Find value in the first place. Say you get 'index = 2' of array.

(2) Store that index in a 'Set' (this will be useful for higher order places)

(3) Now the problem is reduced, as ~~the~~ you can imagine the array size being reduced by 1

(4) Find the value in the second place, same as you got for the first place

E.g. For first place

$K = 15$ x ___

~~the~~ then $\frac{15}{3!} = 3$

i.e index = 2 (0-based)

Now, for second place

- Remove the ~~first~~ sequences that came before 3 i.e. 1___, 2___ ~~to~~ i.e. $2 \times 3!$.

$\therefore K = 15 - 2 \times 3!$

$= 3$

The sequence now becomes 3 x ___

then $\frac{3}{2!} = 1$ i.e. index = 1

(5) ~~Now~~ Now you want to find the second element in the starting seq, i.e. 1, 2, 3, 4 that is not already taken

E.g. 1, 2, 3, 4

For first place we got 3

1, 2, ~~3~~, 4

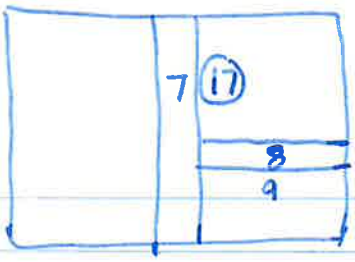
For second place, the second element from left is '2'

1, ~~2~~, ~~3~~, 4

(6) Repeat the above process for third, fourth place

Space = O(n) if you use a set

O(1) if you ~~can allow~~ modify the ~~seq~~ starting sequence to store a special value to mark the indexes already used (will have to reset the starting seq, before each query)




Keep track of 17 and then compare 9 to 17 to finally decide to move above instead of below

Prob Given string and a char in string, return an array with distance from index i to the closest occurrence of char c

Sol s = "loveleetcode", c = "e"

out = [3, 2, 1, 0, 1, 0, 0, 1, 2, 2, 1, 0]

Find left_dist and right_dist separately. Return their min as ans

Prob Find median of a linked list. In even case, return the 'right' value

Sol Use slow and fast pointer.

```
slow, fast = head, head
while fast is not None and fast.next is not None:
    slow = slow.next
    fast = fast.next.next
return slow
```

Prob Find if two rectangles intersect

Sol For 2 lines we have (left1, right1), (left2, right). For them to intersect


① $left1 < x < right1$ && $left2 < x < right2$

② $left1 < x < right2$ && $left2 < x < right1$

From ① & ②

$left1 < right2$ && $left2 < right$

bottom left upper right

For rect, we have two segment for x & y $[x_1, y_1, x_2, y_2]$

```
if rect1[0] < rect2[2] and rect2[0] < rect1[2] and
   rect1[1] < rect2[3] and rect2[1] < rect1[3]:
    return True
return False
```

this can be implemented as an array, where you maintain an indegree and outgree array

```
if len(trust) < N-1:
    return -1
indegree = [0]*(N+1)
outdegree = [0]*(N+1)

for a,b in trust:
    indegree[b] += 1
    outdegree[a] += 1

for i in range(1,N+1):
    if indegree[i] == N-1 and outdegree[i] == 0:
        return i
return -1
```

you can further extend the solution to use only one array.

As indegree - outdegree = -(N-1) or the opposite.

*

Prob: Given a string of words, return an array of all the common chars

Sol words = ['abc', 'bc'] , out = [b, c]

= ['abbc'] , out = [a, b, b, c]

```
common_count = [0]*26
for c in words[0]:
    common_count[ord(c) - ord('a')] += 1

for i in range(1, len(words)):
    temp_count = [0]*26
    for c in words[i]:
        temp_count[ord(c) - ord('a')] += 1

    for i in range(26):
        common_count[i] = min(common_count[i], temp_count[i])

out = []
for i in range(26):
    for j in range(common_count[i]):
        out.append(chr(i + ord('a')))
return out
```

Prob given a 2D grid, rotate elements by k places

$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ for k=1 $\begin{bmatrix} 6 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$

Sol
```
R, C = len(grid), len(grid[0])
out = [[0 for _ in range(C)] for _ in range(R)]

for r in range(R):
    for c in range(C):
        new_c = (c+k) % C
        new_r = (r + (C+k)//C) % R
        out[new_r][new_c] = grid[r][c]
return out
```

$1 \le a, b, c, d \le 9$

Prob given a list of 2D points, two points are equivalent if x = (a, b), y = (c, d) (a == c & b == d) or (a == d and b == c). Find the number of pairs that are equivalent

Sol if instead we are given a list of nums = [1, 1, 3, 4, 3]. Then the number of pairs is related to the count of each unique value.

e.g. for [1, 1, 1] the num pairs are 3 which is $n*(n-1)/2$ which is ${}^nC_2$

So for the given prob we can reduce them to a single num as

min( ) * 10 + max( ) or

multiply with primes as primes[a] * primes[b]

or $|c < a| + c < b$

```
count = 0
d = {}
for x in arr:
    val = min(x[0], x[1]) * 10 + max(x[0], x[1])
    if val not in d:
        d[val] = 0
    d[val] += 1
for k, v in d.items():
    count += v*(v-1)//2
return count
```

Qn: Given an array of size 2n in the form $[x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n]$, return $[x_1, y_1, x_2, y_2, \ldots, x_n, y_n]$

Sol: To do it in O(n) time and O(1) space

- the numbers should be small. So that when the ~~binary~~ numbers are concatenated they still fit in 32-bits
- e.g. → 1000, 1736 → 10001736 ~~(assuming 4 digits)~~ (in practice, every number should take 16 bits, so that concatenation is 32 bits
- Now, we can store pairs of numbers in a single num

Left shift by 16
OR the second num to concatenate
Use AND to get the second num
Use right shift to get the first num

Qn: Given an array of words, return all words which is substring of another word

Sol: words = [mass, as, hero, superhero]

output = [as, hero]

Brute-force

$O(N^2)$ time where we compare all the pairs of ~~s~~ words. The python 'in' ~~to~~ uses KMP like algo to match strings

~~Total time in this case O(S+S)~~

~~In most scenarios, the N is large but~~

Total time for one pair = O(S+S) to build KMP table and search where S = max length of word

For $N^2$ pairs, total time = $O(N^2 S)$, space = ~~O(S)~~ O(NS)

In practice, N is large and S is small. So this is a tradeoff that you have to check.

Solution using suffix tree $O(N \log N + NS^2)$

- Sort words with length first $(N \log N)$
- Add all suffixes for each word to the tree (do not add duplicate) $(NS^2$ to build suffix tree)

Space = $(NS^2)$

# Mastering Regular Expressions (book)

Do not
`|^cat|` → matches a line with 'cat' at the beginning.

Do this
`|^cat|` → matches if you have @ the beginning of a line,
followed immediately by 'c',
followed immediately by 'a',
followed immediately by 't'

^ → Start of line
$ → end of line
[...] → any of these characters can match [ea] either 'e' or 'a'
Use '-' to provide a range [1-9], [a-z], [0-9a-zA-Z]
Use [^...] to do 'not' operation.
. → match any character
| → Same as ~~'OR'~~ 'or' Bob|Robert matches either 'Bob' or 'Robert'

gr[eay]? ⇔ grey|gray ⇔ gr(e|a)y

~~Case~~ your regular expression library also provides
- Case insensitive search
- matching word boundaries

Quantifiers
? → the last character is optional
[colou?r] ⇔ [colour|color]
[4(th)?] ⇔ [4|4th]
+ → one or more of the last character
* → zero or more of the last character

{m,n} → ≥m and ≤n of the last character
[a-zA-Z]{1,5} → match 1 to 5 letters
? = {0,1}

Backreferencing → parentheses remember what was matched, and that can be used later.
~~@~~[([A-Za-z]+) +\1]
~~expression that~~
\1 to get value of first parentheses

reg = /^abc/m
Str = test\nabc\ndef
out = abc

s = becomes useful when searching in a multi-line string
reg = /a.b/s
Str = a\nb
out = a\nb

u = unicode support in regular exp
/\u{1F680}/u
v = add on unicode property escapes and set notation in char classes
/\p{Emoji}/uv

y = start the search from last index. Useful where the order and position of matches are critical
reg = /ab/y
reg.lastIndex = 1;
Str = "cababc"
↑ start search from here

Input boundary
^ = start of string
$ = end of string, or \n if 'm' is provided

Lookahead
(?= pattern)
(?! pattern)
If subsequent input does not match the input, the input is not consumed. But the current position in input remains the same.

Useful if you want to validate the matched string is followed by something without returning that as the result.

/^.*?(?=[,.])/ "Hello, World!" → "Hello"
"Thank you." → "Thank you"

| Greedy | Non-greedy |
|---|---|
| atom? | atom?? |
| atom* | atom*? |
| atom+ | atom+? |
| atom{count} | atom{count}? |
| atom{min,} | atom{min,}? |
| atom{min,max} | atom{min,max}? |

Do not put spaces in between

? = {0,1}
* = {0,}
+ = {1,}

Ch-4

Two engines

① NFA (Nondeterministic Finite Automaton)

It would try to match one regex component at a time. If a match is found, then it will test the next component and so on. If there are possibilities like a|b|c, then it will try them one at a time, until a match is found.

∴ This is a regex-driven engine, and the person writing regex has more control.

The same char in String, might be checked multiple times.

② DFA (Deterministic Finite Automaton)

It scans the string, and keeps track of all the matches "currently in the works".

∴ This is a text-directed engine.

Each char in String, is only checked atmost once.

NFA = implemented using backtracking, and thus can get extremely slow (as in the Cloudfare IP regex hiccup)

In NFA, how to choose b/w various possible options? Greedy quantifiers are evaluated first.

cont 162