General Strategies

Vivek Kulkarni

(1) Don't try to solve the problem from scratch. To solve a problem it is sufficient to do two things:

(a) Show how to solve the problem for smaller inputs (eg: n = 1, empty list, string with 1 character, BST with 1 node)

(b) How to extend / or form the solution for larger problem by reducing it to smaller inputs solution already assumed to be known in (a)

(2) You can break a problem int smaller chunks into 2 ways:

(a) Reduce size of input: "n" node BST into Left & right subtrees or "n character string into (n-1) character string + last character" etc

(b) Reduce search space: Commonly reduce the search space eg in search problems. Eg: left / right half of an array etc, or top half or bottom half of matrix

The above can yield many different commonly used patterns which are outlined below:

Sliding Window Pattern

Keywords: max/min subarray / substring sum/length, O(n) time

The problem typically asks one to find the max/min subarray /substring under some other constraints and usually might expect a O(n) time solution

Brute Force: The brute force solution is to just check every possible subarray &/or substring $\{[i, j] \mid j > i\}$ and pick the best. This will take atleast $O(n^2)$ time.

However if the following property is true, then we can use the below sliding window pattern to ensure we dont need to check all $O(n^2)$ pairs and thus obtain a significantly more efficient algorithm (like $O(n)$).

Sliding Window property : If the property of a candidate subarray $A[i \cdots j]$ can be computed in constant time $O(1)$ from $A[i \cdots j-1]$ and if we can establish that $A[i \cdots j]$ cannot be a solution then it is also guaranteed that $A[i \cdots j+k] \ \forall k \geq 0$ can never be solutions then sliding window pattern can be used and we can eleminate all pairs $[i, j+k] \ \forall k \geq 0$

Template

```
best-soln = None
window-start = 0
current-ix = 0
While current-ix < len(A):
    current-ele = A[current-ix]
    if A[window-start ... current-ix] is not a valid soln:   # check in O(1) time or efficiently
        # update window-start to index which can potentially
          be valid
        window-start += 1

    else:
        best-soln = max/min(best-soln, A[window-start ... current-ix])
        current-ix += 1
return best-soln
```

Representative problems : ① Longest substring without repeating characters. ② Container with most water ③ Longest substring with almost ④ K-distinct characters, ⑤ Longest substring with atleast K repeating characters

Prefix Sum :

This is useful when we need to compute max/sum of every prefix / suffix of an array A. Basically 'if $f(A[0 \cdots i])$ can be computed very easily from $f(A[0 \cdots i-1])$ then we just memoize the computed results and store them. so that values for every prefix can be computed and looked up later. For computing "f" (sum/max) for every suffix, just reverse A and use the same function used for prefixes

Template

```
PREFIX [0] = <value>
~~for ix, e~~ in ~~range(1,~~
for ix, e in enumerate (A):
    if ix == 0:
        continue
    else:
        PREFIX [ix] = PREFIX [ix-1] ⊙ e
return PREFIX
```

Problems : ① Product except self ② Trapping rain water

Binary Search :

We will use this template when searching for an element in an array where it is possible to eliminate half of the search space at each step

Template :

```
low = 0
high = len(A) - 1
while (high > low):
    mid = (low+high)//2
    if A[mid] is your soln:
        return mid, A[mid] # success
    else:
        #using low, high, mid ⇒ identify which half of array
        can be eliminated
        if left can be eliminated:
            low = mid+1
        else
            high = mid - 1   # right is eliminated
return None
```

Keywords : sorted array, rotated array, O(lg n) time

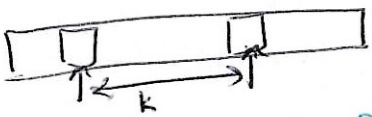
Representative problems : ① Find first & last position of element in sorted array ② Search in rotated sorted array ③ Minimum element in rotated sorted array ④ Peak element in array.

2-pointer pattern :

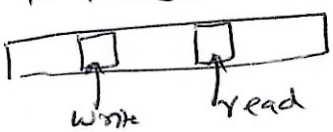
This is useful when dealing with arrays / string / or linked list problems where we might want to process it and compute soln in a single pass.

Idea is to maintain 2 pointers and advance them in different ways.

① Fixed offset : Initialize 2 pointers K-steps apart and advance them in lock-step manner



② Read & Write pointers: Read pointer points to element being currently read while Write pointer points to location to write. Useful for inplace read / write



③ Different Speeds (Hare & Tortoise) : One pointer (Hare) advances at twice the speed of the other (Tortoise)

④ Start & End pointers: Pointers at both ends of array. Start moves to end (right) & end moves to the left

Representative Problems ① Two sum ② 3-sum ③ Merge sorted array ④ Remove duplicates from sorted array ⑤ Move zeros ⑥ Dutch National Flag problem ⑦ Valid palindrome ⑧ Reverse string ⑨ Cycle in linked list ⑩ Kth node from end of LL

Backtracking: Use this when you want to explore all possible solutions (brute force). Typically yields $O(2^n)$ algo.

Template

```
Backtrack ( state, current-soln)
    if is-soln ( state, current-soln):
        process (current-soln)
        return

    else
        candidates = get-candidates ( state, current-soln)
        for candidate in candidates:
            add candidate to current-solution
            backtrack ( state, current-solution)
            un-add candidate from current solution
```

Keywords: all possible strings, solutions, paths etc

Representative problems ① Word search ② word-break II ③ word-break ④ Palindrome Partitioning ⑤ N-queens

BFS: Use when you have to find connected components, traverse a graph or find shortest path from a source in an un-weighted setting

Keywords: Matrix of 0's/1's, # of islands, level order, connected components

Template

```
doBFS ( G, u )
    assert ( u.color == WHITE)
    u.color = GRAY
    d[u] = 0
    parent[u] = None
    Q = [ ]
    Q.append (u)
    While Q
        head = Q.pop(0)
        for v in head.neighbors:
            if v.color == WHITE:
                d[v] = d[u]+1
                parent[v] = u
                v.color = GRAY
                Q.append (v)
    u.color = BLACK

BFS ( G ):
    num-components = 0
    for u in G
        if u.color == WHITE
            doBFS (G, u)
            num-components +=1
```

Representative problems: ① Number of islands ② Number of connected components ③ Level order traversal of tree ④ Zig zag traversal of tree

DFS & Topological Sort : Use this when there is an underlying graph and you need to detect cycles or you need a valid ordering of vertices of a directed graph (acyclic) that typically requires ordering defined by edges to be respected

```
doDFSVisit(G, u)
    assert(u.color == WHITE)
    u.start = global-time + 1
    u.color = GREY
    for v in u.neighbors:
        if v.color == WHITE:
            doDFSVISIT(G, v)
        elif v.color == GREY
            #cycle detected.
             handle as required
    u.end = global-time + 1
    u.color = BLACK

doDFS(G):
    for u in G:
        if u.color == WHITE:
            doDFS(G, u)

TopologicalSort(G):
    ① Do DFS on G (G is a DAG)
    ② Sort vertices in descending order of finish
       time & output it
```

Problems : ① Jump game ② Alien Dictionary ③ Course Schedule ④ Course Schedule-II

Array as Hashmap:

Use on problems where all numbers are in range $[1 \ldots N]$ & requirement is to use $O(1)$ space.

Idea: Rather than hash element "e" to ~~index "i" in~~ a seprate hashmap, use the given array itself. Basically assume that "e" hashes to index "e". Key idea works because all elements are bounded eg $[0 \ldots N-1)$ or $[1 \ldots N]$ etc.

Representative problems: ① Contains Duplicates ② Find duplicate numbers ③ First missing positive

Heap for maintaining ~~top~~ K (max/min) among K-elements / OR Top K elements in a stream

① Maintain a heap of K-elements (overall) assuming a stream
- → top K-max: maintain a min-heap. If new element > min remove min & add new element
- → top K-min: maintain a max-heap. If new-element < max remove max & add new element

② K-way merge: To find minimum element among K sorted arrays, use a min-heap. & store (e, array source) in heap.

Representative problems ① Merge K-sorted lists ② Top K frequent elements ③ Meeting Rooms - II.

Interval Patterns:

There are 2 main sub-problems when dealing with time interval data.

① Do two time intervals overlap?

```
doIntervalsOverlap (u, v)
    t-start = max(u.start, v.start)
    t-end = min(u.end, v.end)
    return t-start <= t-end
```

② Merge 2 time overlapping intervals:

```
Merge(u, v): #assumes u & v overlap)
    t-start = min(u.start, v.start)
    t-end = max(u.end, v.end)
    return (t-start, t-end) # merged interval
```

It is generally a good idea to sort an array of intervals by start time.

Representative Problems

① Meeting Rooms

② Merge intervals

③ Meeting Rooms - II

Bit Operations:

They are used to deal with problems like ① find # of set bits or where binary representation of a number is revealing of some property.

```
num-bits (x)
    c=0
    while(x)
        c+=1
        x = x & (x-1)
    return x
```

```
power-of-two(n)
    return (n & (n-1) == 0)

is-bit-pos-set(n, bit-pos)
    return n & (1 << bit-pos)
```

Representative Problems:

① All elements occur twice except one ② Number of bits set ③ Power of 2 ④ Is bit set

Recursion (+ Memoization)

Steps: ① Set up base case soln (eg empty tree, string, n=0 etc)

② Start with an arbitrary instance of size 'n'

- → Show how to solve it by reducing size & assume magic of recursion for smaller cases
- → Memoize if same problem is solved multiple times in different branches

Example: Roman to integer

```
def roman-to-int(s):
    if len(s) == 1 or len(s) == 2:  #Base case
        return 'n' from stored map
    if s[-2:] in ['IV', 'IX'...]:  ← Recursion  ← 2-character to INT
        return: roman-to-int(s[:-2] + soln[s[-2:]])
    else:  ← Recursion
        return: roman-to-int(s[:-1] + soln(s[-1]))  ← 1 Roman char to INT
```

Representative Problems: ① Roman to Integer ② Longest Palindromic Substring ③ Longest common prefix ④ letter combinations of phone number ⑤ Spiral matrix ⑥ Word search ⑦ Decode ways ⑧ Count & say ⑨ Coin change ⑩ Pascal's triangle ⑪ Climbing Stairs ⑫ Pow(x,n) ⑬ House Robber ⑭ Permutation ⑮ Subsets ⑯ Validate BST ⑰ Generate parentheses ⑱ Combinations ⑲ Generate all permutations ⑳ Knapsack.