Programming Interview Notes

Michael Li

Contents

1 DFS **2**

2 BFS **2**

3 Tree Traversal (DFS on Trees) **2**

3.1 Preorder Traversal 2

3.2 Inorder Traversal 3

4 Sliding Window **3**

5 Backtracking **4**

6 Dynamic Programming **4**

7 Greedy Algorithm **5**

1 DFS

```
stack.push(element);
while(!stack.isEmpty()) {
  T curr = stack.pop();
  for (e : adj(curr)) {
    if(!visited[e]) {
      visited[e] = true;      //Keep track of unvisited using array
      stack.push(e);
    }
  }
}
```

For graphs, need to figure out to use visited array to match requirements
323. Number of Connected Components in an Undirected Graph   200. Number of Islands   547. Friend Circles

2 BFS

```
queue.add(element);
while(!queue.isEmpty()) {
  T curr = queue.poll();
  for (e : adj(curr)) {
    if(!visited[e]) {
      visited[e] = true;    //keep track of unvisited using array
      queue.add(e);
    }
  }
}
```

Key issue to recognize when to use DFS (look at deep as possible first) or BFS (process neighbors first because going deep first might be inefficient)
102. Binary Tree Level Order Traversal   542. 01 Matrix

3 Tree Traversal (DFS on Trees)

3.1 Preorder Traversal

root, Left, Right

```
public List<Integer> preorderTraversal(TreeNode root) {
  List<Integer> list = new ArrayList();
  if(root == null) return list;
  Stack<TreeNode> stack = new Stack();
  stack.push(root);

  while(!stack.isEmpty()) {
    TreeNode curr = stack.pop();
    list.add(curr.val);
    if(curr.right != null) stack.push(curr.right);
    if(curr.left != null) stack.push(curr.left);
  }
  return list;
}
```

144. Binary Tree Preorder Traversal

3.2 Inorder Traversal

Left, root, Right
Useful when you need to iterate through a tree inorder or retrieve the kth element

```
public List<Integer>> inorderTraversal(TreeNode root) {
  List<Integer> list = new ArrayList();
  if(root == null) return list;

  Stack<TreeNode> stack = new Stack();  //Use stack for DFS
  while(root != null && !stack.isEmpty()) {
    while(root != null) {               //Find left-most element
      stack.push(root);
      root = root.left;
    }
    root = stack.pop();
    list.add(root.val);
    root = root.right;                  //Look at left-most element's right child
  }
}
```

94. Binary Tree Inorder Traversal   98. Validate Binary Search Tree   230. Kth Smallest Element in a BST

4 Sliding Window

Use two pointers and keep sliding the right pointer, examining the new character and updating the count table. Once a condition (usually duplicate) is found, start removing elements from the left pointer, shortening the window.
76. Minimum Window Substring

```
public String minWindow(String s, String t) {
  Map<Character, Integer> map = new HashMap();
  for(char c : t.toCharArray()) {
    if(map.containsKey(c)) {
      map.put(c, map.get(c) + 1);
    } else {
      map.put(c, 1);
    }
  }
  int left = 0;
  int right = 0;
  int slen = s.length();
  int count = map.size();
  int len = slen;
  String ans = s;

  while(end < slen) {
    char rightChar = s.charAt(right);
    if(map.containsKey(rightChar)) {
      map.put(rightChar, map.get(rightChar) - 1);
      if(map.get(rightChar) == 0) count--;
    }
    right++;

    while(count == 0) {
      if(right - left < len) {     //right++ happened above so indexing is correct
        len = right - left;
        ans = s.substring(left, right);
      }
      char leftChar = s.charAt(left);
```

```java
      if(map.containsKey(leftChar)) {
        map.put(leftChar, map.get(leftChar) + 1);
        if(map.get(leftChar) > 0) count++;
      }
      left++;
    }
  }
}
```

5 Backtracking

1. Iterate through all possible configurations of search space (permutations or subsets)
2. Generate each configuration once by defining a generation order
3. At each step, try to extend a partial solution by adding an element. Teest if it's a solution
   - if yes then process it
   - if no then recurse down and check if it is a partial solution
4. Pruning: cut off search if we know it is not part of the solution (e.g. cost > curr min)

```java
public List<List<Integer>> subsets(int[] nums) {
    List<List<Integer>> list = new ArrayList(); //solution array
    backtrack(list, new ArrayList(), 0, nums);
    return list;
}

public void backtrack(List<List<Integer>> list, List<Integer> sublist, int k, int[] nums) {
    if (isSolution(sublist)) {
      process(sublist);                    //if sublist is a valid solution, process it
    }
    for (int i = k; i < nums.length; i++) {
        sublist.add(nums[i]);              //extend partial solution
        backtrack(list, sublist, i + 1, nums); //recurse down
        sublist.remove(sublist.size() - 1); //remove extension
    }
}
```

6 Dynamic Programming

Solves by combining optimal solutions to subproblems that overlap. Divide and conquer only work for disjoint sets otherwise there's a lot of repetitive recursive calls. Howeever, dynamic programming uses additional memory to save subproblem solutions so there is a time-memory tradeoff.

1. Characterize structure of optimal solution
2. Recursively define value of optimal solution
3. Compute value of optimal solution (usually bottom-up approach)
4. Construct optimal solution from computed info

Two approaches to dynamic programming:

- Top-down with memoization: write the procedure recurisvely but saves result of each subproblem in an array. The procedure will first check to see if the array has the solution to the subproblem; if not then the value is computed normally.
- Bottom-up: intuition is to sort by subproblem size and solve them in size order (smallest first). When solving a particular subproblem, we have already saved the solutions to the smaller subproblems.

Look for an optimal substructure: optimal solution requires optimal solution to subproblems

- Solution consists of making a choice
- Assume given choice leads to optimal solutino
- determine which subproblems occur
- show solutions to subproblems used within optimal solution must also be optimal

Dynamic Programming makes choice at each step based on solutions to sub problems so we do bottom-up approach, using best subproblems to solve later problems or top-down with memoization. In both cases, we always end up solving the smaller subproblems then solving bigger ones.

7 Greedy Algorithm

Greedy Choice Property: Idea is to make locally optimal choices in hopes that it will lead to a globally optimal solution.

- Determine optimal substructure
- Develop recursive solution
- Show if we make greedy choice, only one subproblem remains
- Prove it's safe to make greedy choice
- Develop recursive algorithm for greedy choice
- Convert recursive greedy to iterative

We want to make greedy choice then solve the subproblems that remain. This is usually a top-down approach where one greedy choice is made after another, reducing the size of the problem