**⁕ Code templates :~**

**1.Two pointers: one input, opposite ends**

*int fn(vector<int>& arr) {*

*int left = 0;*

*int right = int(arr.size()) - 1;*

*int ans = 0;*

*while (left < right) {*

*// do some logic here with left and right*

*if (CONDITION) {*

*left++;*

*} else {*

*right--;*

*}*

*}*

*return ans;*

*}*

**2.Two pointers: two inputs, exhaust both**

*int fn(vector<int>& arr1, vector<int>& arr2) {*

*int i = 0, j = 0, ans = 0;*

*while (i < arr1.size() && j < arr2.size()) {*

*// do some logic here*

*if (CONDITION) {*

*i++;*

*} else {*

*j++;*

*}*

*}*

*while (i < arr1.size()) {*

*// do logic*

*i++;*

*}*

*while (j < arr2.size()) {*

*// do logic*

*j++;*

*}*

*return ans;*

*}*

**3.Sliding window**

*int fn(vector<int>& arr) {*

*int left = 0, ans = 0, curr = 0;*

*for (int right = 0; right < arr.size(); right++) {*

*// do logic here to add arr[right] to curr*

*while (WINDOW\_CONDITION\_BROKEN) {*

*// remove arr[left] from curr*

*left++;*

*}*

*// update ans*

*}*

*return ans;*

*}*

**4.Build a prefix sum**

*vector<int> fn(vector<int>& arr) {*

*vector<int> prefix(arr.size());*

*prefix[0] = arr[0];*

*for (int i = 1; i < arr.size(); i++) {*

*prefix[i] = prefix[i - 1] + arr[i];*

*}*

*return prefix;*

*}*

**5.Efficient string building**

*string fn(vector<char>& arr) {*

*return string(arr.begin(), arr.end())*

*}*

**6.Linked list: fast and slow pointer**

*int fn(ListNode\* head) {*

*ListNode\* slow = head;*

*ListNode\* fast = head;*

*int ans = 0;*

*while (fast != nullptr && fast->next != nullptr) {*

*// do logic*

*slow = slow->next;*

*fast = fast->next->next;*

*}*

*return ans;*

*}*

**7.Reversing a linked list**

*ListNode\* fn(ListNode\* head) {*

*ListNode\* curr = head;*

*ListNode\* prev = nullptr;*

*while (curr != nullptr) {*

*ListNode\* nextNode = curr->next;*

*curr->next = prev;*

*prev = curr;*

*curr = nextNode;*

*}*

*return prev;*

*}*

**8.Find number of subarrays that fit an exact criteria**

*int fn(vector<int>& arr, int k) {*

*unordered\_map<int, int> counts;*

*counts[0] = 1;*

*int ans = 0, curr = 0;*

*for (int num: arr) {*

*// do logic to change curr*

*ans += counts[curr - k];*

*counts[curr]++;*

*}*

*return ans;*

*}*

**9.Monotonic increasing stack**

*int fn(vector<int>& arr) {*

*stack<integer> stack;*

*int ans = 0;*

*for (int num: arr) {*

*// for monotonic decreasing, just flip the > to <*

*while (!stack.empty() && stack.top() > num) {*

*// do logic*

*stack.pop();*

*}*

*stack.push(num);*

*}*

*}*

**10.Binary tree: DFS (recursive)**

*int dfs(TreeNode\* root) {*

*if (root == nullptr) {*

*return 0;*

*}*

*int ans = 0;*

*// do logic*

*dfs(root.left);*

*dfs(root.right);*

*return ans;*

*}*

**11.Binary tree: DFS (iterative)**

*int dfs(TreeNode\* root) {*

*stack<TreeNode\*> stack;*

*stack.push(root);*

*int ans = 0;*

*while (!stack.empty()) {*

*TreeNode\* node = stack.top();*

*stack.pop();*

*// do logic*

*if (node->left != nullptr) {*

*stack.push(node->left);*

*}*

*if (node->right != nullptr) {*

*stack.push(node->right);*

*}*

*}*

*return ans;*

*}*

**12.Binary tree: BFS**

*int fn(TreeNode\* root) {*

*queue<TreeNode\*> queue;*

*queue.push(root);*

*int ans = 0;*

*while (!queue.empty()) {*

*int currentLength = queue.size();*

*// do logic for current level*

*for (int i = 0; i < currentLength; i++) {*

*TreeNode\* node = queue.front();*

*queue.pop();*

*// do logic*

*if (node->left != nullptr) {*

*queue.push(node->left);*

*}*

*if (node->right != nullptr) {*

*queue.push(node->right);*

*}*

*}*

*}*

*return ans;*

*}*

**13.Graph: DFS (recursive)**

For the graph templates, assume the nodes are numbered from 0 to n - 1 and the graph is given as an adjacency list. Depending on the problem, you may need to convert the input into an equivalent adjacency list before using the templates.

*unordered\_set<int> seen;*

*int fn(vector<vector<int>>& graph) {*

*seen.insert(START\_NODE);*

*return dfs(START\_NODE, graph);*

*}*

*int dfs(int node, vector<vector<int>>& graph) {*

*int ans = 0;*

*// do some logic*

*for (int neighbor: graph[node]) {*

*if (!seen.contains(neighbor)) {*

*seen.insert(neighbor);*

*ans += dfs(neighbor, graph);*

*}*

*}*

*return ans;*

*}*

**14.Graph: DFS (iterative)**

*int fn(vector<vector<int>>& graph) {*

*stack<int> stack;*

*unordered\_set<int> seen;*

*stack.push(START\_NODE);*

*seen.insert(START\_NODE);*

*int ans = 0;*

*while (!stack.empty()) {*

*int node = stack.top();*

*stack.pop();*

*// do some logic*

*for (int neighbor: graph[node]) {*

*if (!seen.contains(neighbor)) {*

*seen.insert(neighbor);*

*stack.push(neighbor);*

*}*

*}*

*}*

*}*

**15.Graph: BFS**

*int fn(vector<vector<int>>& graph) {*

*queue<int> queue;*

*unordered\_set<int> seen;*

*queue.push(START\_NODE);*

*seen.insert(START\_NODE);*

*int ans = 0;*

*while (!queue.empty()) {*

*int node = queue.front();*

*queue.pop();*

*// do some logic*

*for (int neighbor: graph[node]) {*

*if (!seen.contains(neighbor)) {*

*seen.insert(neighbor);*

*queue.push(neighbor);*

*}*

*}*

*}*

*}*

**16.Find top k elements with heap**

*vector<int> fn(vector<int>& arr, int k) {*

*priority\_queue<int, CRITERIA> heap;*

*for (int num: arr) {*

*heap.push(num);*

*if (heap.size() > k) {*

*heap.pop();*

*}*

*}*

*vector<int> ans;*

*while (heap.size() > 0) {*

*ans.push\_back(heap.top());*

*heap.pop();*

*}*

*return ans;*

*}*

**17.Binary search**

*int binarySearch(vector<int>& arr, int target) {*

*int left = 0;*

*int right = int(arr.size()) - 1;*

*while (left <= right) {*

*int mid = left + (right - left) / 2;*

*if (arr[mid] == target) {*

*// do something;*

*return mid;*

*}*

*if (arr[mid] > target) {*

*right = mid - 1;*

*} else {*

*left = mid + 1;*

*}*

*}*

*// left is the insertion point*

*return left;*

*}*

**18.Binary search: duplicate elements, left-most insertion point**

*int binarySearch(vector<int>& arr, int target) {*

*int left = 0;*

*int right = arr.size();*

*while (left < right) {*

*int mid = left + (right - left) / 2;*

*if (arr[mid] >= target) {*

*right = mid;*

*} else {*

*left = mid + 1;*

*}*

*}*

*return left;*

*}*

**19.Binary search: duplicate elements, right-most insertion point**

*int binarySearch(vector<int>& arr, int target) {*

*int left = 0;*

*int right = arr.size();*

*while (left < right) {*

*int mid = left + (right - left) / 2;*

*if (arr[mid] > target) {*

*right = mid;*

*} else {*

*left = mid + 1;*

*}*

*}*

*return left;*

*}*

**20.Binary search: for greedy problems**

If looking for a minimum:

*int fn(vector<int>& arr) {*

*int left = MINIMUM\_POSSIBLE\_ANSWER;*

*int right = MAXIMUM\_POSSIBLE\_ANSWER;*

*while (left <= right) {*

*int mid = left + (right - left) / 2;*

*if (check(mid)) {*

*right = mid - 1;*

*} else {*

*left = mid + 1;*

*}*

*}*

*return left;*

*}*

*bool check(int x) {*

*// this function is implemented depending on the problem*

*return BOOLEAN;*

*}*

If looking for a maximum:

*int fn(vector<int>& arr) {*

*int left = MINIMUM\_POSSIBLE\_ANSWER;*

*int right = MAXIMUM\_POSSIBLE\_ANSWER;*

*while (left <= right) {*

*int mid = left + (right - left) / 2;*

*if (check(mid)) {*

*left = mid + 1;*

*} else {*

*right = mid - 1;*

*}*

*}*

*return right;*

*}*

*bool check(int x) {*

*// this function is implemented depending on the problem*

*return BOOLEAN;*

*}*

**21.Backtracking**

*int backtrack(STATE curr, OTHER\_ARGUMENTS...) {*

*if (BASE\_CASE) {*

*// modify the answer*

*return 0;*

*}*

*int ans = 0;*

*for (ITERATE\_OVER\_INPUT) {*

*// modify the current state*

*ans += backtrack(curr, OTHER\_ARGUMENTS...)*

*// undo the modification of the current state*

*}*

*return ans;*

*}*

***22.Dynamic programming: top-down memorization***

*unordered\_map<STATE, int> memo;*

*int fn(vector<int>& arr) {*

*return dp(STATE\_FOR\_WHOLE\_INPUT, arr);*

*}*

*int dp(STATE, vector<int>& arr) {*

*if (BASE\_CASE) {*

*return 0;*

*}*

*if (memo.contains(STATE)) {*

*return memo[STATE];*

*}*

*int ans = RECURRENCE\_RELATION(STATE);*

*memo[STATE] = ans;*

*return ans;*

*}*

To convert a top-down solution to a bottom-up one:

1. Initialize an array dpdp that is sized according to the state variables. For example, let's say the input to the problem was an array numsnums and an integer kk that represents the maximum number of actions allowed. Your array dpdp would be 2D with one dimension of length nums.lengthnums.length and the other of length kk. In the top-down approach, we had a function dp. We want these two to be equivalent. For example, the value of dp(4, 6) can now be found in dp[4][6].
2. Set your base cases, same as the ones you are using in your top-down function. In the example we just looked at, we had dp(0) = dp(1) = 0. We can initialize our dp array values to 0 to implicitly set this base case. As you'll see soon, other problems will have more complicated base cases.
3. Write a for-loop(s) that iterate over your state variables. If you have multiple state variables, you will need nested for-loops. These loops should **start iterating from the base cases and end at the answer state**.
4. Now, each iteration of the inner-most loop represents a given state, and is equivalent to a function call to the same state in top-down. Copy-paste the logic from your function into the for-loop and change the function calls to accessing your array. All dp(...)dp(...) changes into dp[...]dp[...].
5. We're done! dpdp is now an array populated with the answer to the original problem for all possible states. Return the answer to the original problem, by changing return dp(...)return dp(...) to return dp[...]return dp[...].

**23.Build a trie**

*// note: using a class is only necessary if you want to store data at each node.*

*// otherwise, you can implement a trie using only hash maps.*

*struct TrieNode {*

*int data;*

*unordered\_map<char, TrieNode\*> children;*

*TrieNode() : data(0), children(unordered\_map<char, TrieNode\*>()) {}*

*};*

*TrieNode\* buildTrie(vector<string> words) {*

*TrieNode\* root = new TrieNode();*

*for (string word: words) {*

*TrieNode\* curr = root;*

*for (char c: word) {*

*if (!curr->children.contains(c)) {*

*curr->children[c] = new TrieNode();*

*}*

*curr = curr->children[c];*

*}*

*// at this point, you have a full word at curr*

*// you can perform more logic here to give curr an attribute if you want*

*}*

*return root;*

*}*

**24.Dijkstra's algorithm**

*vector<int> distances(n, INT\_MAX);*

*distances[source] = 0;*

*priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> heap;*

*heap.push({0, source});*

*while (!heap.empty()) {*

*int currDist = heap.top().first;*

*int node = heap.top().second;*

*heap.pop();*

*if (currDist > distances[node]) {*

*continue;*

*}*

*for (pair<int, int> edge: graph[node]) {*

*int nei = edge.first;*

*int weight = edge.second;*

*int dist = currDist + weight;*

*if (dist < distances[nei]) {*

*distances[nei] = dist;*

*heap.push({dist, nei});*

*}*

*}*

*}*