DP template:

1. High level idea on recursion
2. Base case
3. Implementation with iteration

DP complexity:

1. if time complexity is O(n), space complexity can be optimized to O(1)
2. if time complexity is O(n^2), space complexity is O(n)
3. if time complexity is O(m*n) for a 2d grid, space complexity can be optimized to O(n) or O(m)

Knapsack template, O(n^2) time, O(n) space:

```
int[] dp = new int[target+1];   // int array or boolean array
dp[0] =        // initialization
for(num : nums) {
        for(; ; ;) {// find the condition that meets the requirement
                        // elements can be repeated infinite times, write for loop from the front to the end
                        // elements can only be use for one time, write for loop from the end to the front
           // dp implementation
        }
}
return dp[target]
```

Possible variations:

1. Elements can be repeated or can only be used once
2. For grid problem, there might be variations that there are some obstacle in certain grids

DFS template:

1. check base case, return
2. dfs(left)
3. function(root) // process root element
4. dfs(right)

```
public void dfs_traverse(TreeNode root) {
	check if root is null, for base case
	dfs_traverse(root.left);
	process(root);
	dfs_traverse(root.right);
}
```

possible variations:

1. order of doing dfs on left, right and on processing root elements
2. different functions on processing root element

Backtrack template:

```
backtrack() {
1.      check base case, return

for(all possible options) {
                1.      check if it is a valid option. If not, break;
                2.      choose that optoin
                3.      Backtrack()
                4.      undo choosing that option
        }
}

private List<List<Integer>> res = new ArrayList<>();
private List<Integer> current_list = new ArrayList<>();
res.add(new ArrayList(current_list));
```

BFS template:

1. initialize a priorityQueue
2. Initialize a HashSet, to store visited elements
3. put the first element into the set and into priorityQueue
4. **while** (the queue is not empty) {

```
        take out one element from the queue
        for(all neighboring elements) {
                if(this element is not yet visited) {
                        add this element into the queue
                }
        }
}
```

Possible variation:

1. Count the number of nodes in each level
2. Add the left element first into the queue, or the right element first

Binary search template:

Binary search, O(logn):

1. left = 0, right = len -1
2. **while**(left <= right)
3. mid = left + (right- left) / 2
4. **if** too small, left = mid +1
5. **if** too big, right = mid – 1
6. **if** equal, **return**
7. **return** -1 when no match

Binary search left bound:

1. left = 0, right = len -1
2. **while**(left <= right)
3. mid = left + (right – left) . 2
4. **if** too small, left = mid +1
5. **if** too big, right = mid – 1
6. **if** equal, right = mid – 1
7. **if**(left >=len || nums[left] != target), **return** -1
8. **return** left

Binary search right bound:

1. left = 0, right = len -1
2. **while**(left <= right)
3. mid = left + (right – left) . 2
4. **if** too small, left = mid +1
5. **if** too big, right = mid – 1
6. **if** equal, left = mid + 1
7. **if**(right <0|| nums[right] != target), **return** -1
8. **return** right

Sliding window minimum template:

```
left = 0, right;
for(right = 0; right < len; right++) {
	add the element pointed by right into the window

	while(window valid) {
		use Math.min to update target function
		remove the element pointed by left from the window
		left++; // increment left pointer
	}
}
```

Sliding window maximum template:

```
left = 0, right;
for(right = 0; right < len; r++) {
	add the element pointed by right into the window

	while(window invalid) {
		remove the element pointed by left from the window
		left++; // increment left pointer
	}
	// after the this while loop, the window is valid again.
	use Math.max to update max
}
```

Greedy, activity selection template:

1. sort the elements using ending time
2. choose the one that ends the earliest, eliminate those that are in conflict with it
3. repeat step 2 until there is no more available elements

```
Arrays.sort(intervals, Comparator.comparingInt(o -> o[1]));
int end = intervals[0][1];
for (int i = 1; i < intervals.length; i++) {
   if (intervals[i][0] < end) {
      continue;
   }
   end = intervals[i][1];
}
```

Binary tree template:

function(root):

1. check base **case**.
2. process the root
3. recursive calls: function(root.left), function(root.right)

LinkedList template:

function(node) :

1. check base **case**.
2. process current node, and possibly process node.next or node.next.next
3. recursive calls: function(node.next) or function(node.next.next)

Next greater number template, using stack:

```
Stack<Integer> stack = new Stack<>();
for(int i = 0; i < len; i++) {
   while(!stack.isEmpty() && stack.peek() < nums[i]) {
      int current = stack.pop();
   }
   stack.add(nums[i]);
}
```

Double pointer template:

Double pointer: left, right

Segmentation: 0, left, right, len - 1

```
left = 0; right = len - 1;
while(left < right) {
	int temp = nums[left] + nums[right];
	if(temp == target) {
		return;
	} else if(temp > target) {
		right--;
	} else if(temp < target) {
		left++;
	}
}
```

Triple pointer template:

triple pointer: left, current, right

Segementation: 0, left, current, right, len - 1;

0 to left: small numbers
Left to current: middle numbers
Current to right: unknown numbers (could be small, middle or large)
Right to Len - 1: big numbers

```
int len = nums.length;
int left = 0; // this is to store 0
int right = len - 1; // this is to store 2
int current = 0; // this is to store 1

while(current <= right) {
   if(nums[current] == 0) {
      swap(nums, current, left);
      eft++;
      current++;
   } else if(nums[current] == 1) {
      current++; // do nothing
   } else {
               // don't do current++;
               // the one on the right may still need to be swapped again
      swap(nums, current, right);
       right--;
   }
}
```

LinkedList cycle template:

```
// detect cycle
slow = head;
fast = head.next;
while(slow != fast) {
    slow = slow.next;
    fast = fast.next;
}

// find entry in the cycle. Move either slow or fast to the head
fast = head;
        while(slow.next != fast) {
            slow = slow.next;
            fast = fast.next;
        }
return fast;

// find the length of the cycle
fast = slow.next;
length = 1;
while(slow != fast) {
    fast = fast.next;
    length++;
}
```

Topological sort template, DFS:

1. Run DFS
2. output vertices in decreasing order of finish time

```
dfs(Node current) {

    if(visiting) {
        // this means that there is no topological sort
    }

    if(visited) {
        break;
    }

    set status to visiting
    for(Node child : current.children) {
        dfs(child);
    }
    set status to visited

    result.add(current);
}

in main function:
 for all nodes: dfs(node)
Collections.reverse(result)
```

Bellman-Ford Algorithm template:

This algorithm is to calculate single source, shortest path. It is good with negative weights. Initially, the distance to the starting node is 0 and the distance to any other node is infinite. The algorithm then reduces the distance by finding edges that shorten the paths until it is not possible to reduce any distance.

The requirement is that there should not be any cycles with negative weights.

It can be used to detected cycles with negative weights. Time complexity is O(mn)

```
for(int i = 0; i <= n; i++) {
distance[i] = Integer.MAX_VALUE; // initialize the distance of all other
node to be infinity
        }
distance[x] = 0; // this is the starting node

for(int i = 1; i <= n-1; i++) {  // for each node
        for(edge e : edges) {   // for each edge
                (start, end, weight) = e;
                // try to reduce the distance by finding edges that shortes the
paths
                distance[end] = Math.min(distance[end], distance[start] +
weight);
        }
}
```

Dijkstra’s algorithm template:

This algorithm is to calculate single source, shortest path.

The requirement is that there is no negative weight.

Initially, the distance to the starting node is 0 and the distance to any other node is infinite. At each stop, **this** algorithm selects a node that has not been processed yet, and whose distance is as small as possible. Then, the goes through all the edges start at that node and reduces the distances using them.

Time complexity: O(n + m logm)

```
priority_queue<pair<int,int>> q;
for (int i = 1; i <= n; i++) {
distance[i] = INF;
}
distance[x] = 0;
q.push({0,x});
while (!q.empty()) {
	int a = q.top().second; q.pop(); if (processed[a]) continue; processed[a] = true;
	for (auto u : adj[a]) {
		int b = u.first, w = u.second;
		if (distance[a]+w < distance[b]) {
				distance[b] = distance[a]+w;
				q.push({-distance[b],b});
		}
	}
}
```

Floyd-Warshall Algorithm template:

This algorithm is to calculate all pairs shortest path.

The algorithm maintains a matrix that contains distances between the nodes. The initial matrix is directly constructed based on the adjacency matrix of the graph. Then, the algorithm consists of consecutive rounds, and on each round, it selects a **new** node that can act as an intermediate node in paths from now on, and reduces distances using **this** node.

Time complexity is O(n^3)

```
for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
                if (i == j) dist[i][j] = 0;
                else if (adj[i][j]) dist[i][j] = adj[i][j]; else dist[i][j] = INF;
        }
}

for (int k = 1; k <= n; k++) {
        for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= n; j++) {
                        dist[i][j] = min(dist[i][j],dist[i][k]+dist[k][j]);
                }
        }
}
```