
Studies SDE Interview

Bruna Zamith Santos

05/2020

<http://bzamith.github.io>

Contents

1 Behavioral Interview	1
2 Coding	2
2.1 Big-O	2
2.2 Tree Traversals	3
2.3 Divide and Conquer	5
2.4 Breadth-first search	8
2.5 Depth-first search	9
2.6 Quicksort	9
2.7 Mergesort	10
2.8 Dynamic Programming	11
3 Python	12
3.1 Set	12
3.2 List	12
3.3 Dict	13
3.4 Sort	14
3.5 Join	15
3.6 Strip	15
4 Data Structures	16
4.1 Linked List	16
4.2 Hash Table	16
4.3 Binary Search Tree	16
4.4 Min-heap	16
4.5 Stack	16
4.6 Queue	16
5 Object-oriented Programming	17
5.1 Concrete Class, Abstract Class and Interface	17
5.2 Static	17
5.3 Inheritance	17
5.4 Aggregation, Association and Composition	18
5.5 Coupling and Cohesion	18
5.6 Polymorphism	18
5.7 Function overloading	19
6 Design Patterns	20
6.1 Template Method	20
6.2 Strategy	20
6.3 State	20
6.4 Composite	20
6.5 Singleton	20
6.6 Builder	20
6.7 Factory	20

7	Architecture	21
7.1	API	21
7.2	Remote Procedure Call	21
7.3	Caching	21
7.4	Database Partitioning (Sharding)	21
7.5	Non Relational Databases	21

1 Behavioral Interview

- Study Amazon's Leadership Principles (LPs)¹
- Try to think of examples from previous works that fit each LP
 - Having more than one example per LP is desirable
 - The more diverse the examples are, the better it is
- Structure each example in Amazon's STAR Method²
- Prepare questions to ask the interviewers

¹<https://www.aboutamazon.com/working-at-amazon/our-leadership-principles>

²https://www.amazon.jobs/en/landing_pages/in-person-interview

2 Coding

2.1 Big-O

- Big-O Cheatsheet

- <https://github.com/bzamith/BigO-Cheatsheet>

- Asymptotic analysis

- Best case, worst case, expected case

- Space complexity and Time complexity

- Stack space (from recursion) counts as space complexity as well

```
1 int sum(int n) {  
2     if (n <= 0)  
3         return 0;  
4     return (n + sum (n-1));  
5 }  
6  
7 Stack memory:  
8 - sum(4)  
9 - sum(3)  
10 - sum(2)  
11 - sum(1)  
12 - sum(0)
```

- Drop constants and non-dominant terms

- $O(2N) = O(N)$
 - $O(N^2 + N) = O(N^2)$
 - $O(N + \log(N)) = O(M)$

- Add vs Multiply

```
1 > Sequential loops: O(A+B)  
2 for(int a: arrA) {  
3     print(a);  
4 }  
5 for(int b: arrB) {  
6     print(b);  
7 }  
8  
9 > Nested loops: O(A*B)  
10 for(int a: arrA) {  
11     print(a);  
12     for(int b: arrB) {  
13         print(a + "." + b);  
14     }  
15 }
```

- Comparison

- $O(1) < O(\log(N)) < O(N) < O(N * \log(N)) < O(N^2) < O(N^3) < O(2^N) < O(N!)$

- Amortized Time

- ArrayList example: When inserting, if not full, $O(1)$. If full, $O(N)$, because it will create an array with double the current size and copy the previous N elements.

- $O(\log(N))$

- Example: Binary Search

```

1 N = 16 /* divide by 2 */
2 N = 8  /* divide by 2 */
3 N = 4  /* divide by 2 */
4 N = 2  /* divide by 2 */
5 N = 1
6
7 2^k = N
8 k = log(N) base 2

```

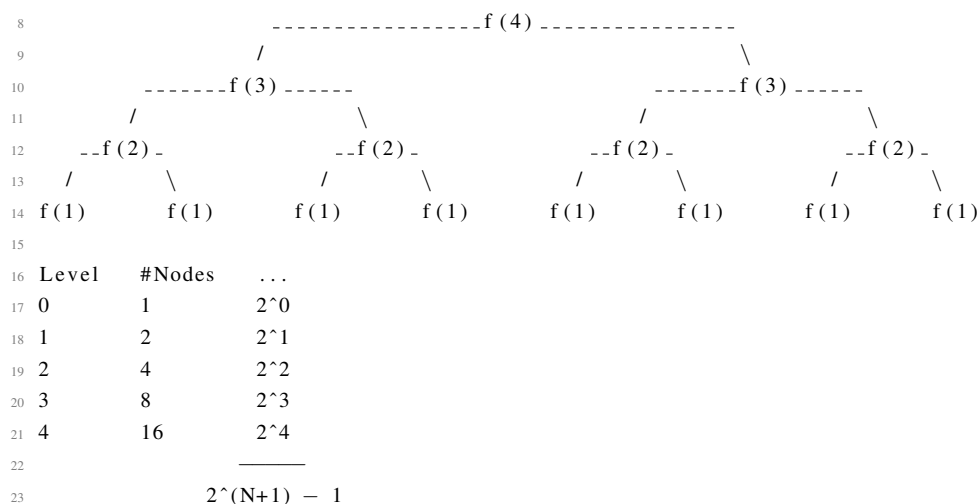
- When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a $O(\log(N))$ runtime

- Recursive runtime

```

1 int f(int n) {
2     if (n <= 1) {
3         return (1);
4     }
5     return (f(n-1) + f(n+1));
6 }

```



- $O(2^N)$ time complexity
 - Although we have $O(2^N)$ in the tree total, only $O(N)$ exist at any given time. So $O(N)$ space complexity.

2.2 Tree Traversals

- Binary Tree: Tree data structure in which each node has at most two children

- Binary Search Tree:

- The left subtree of a node contains only nodes with keys lesser than the node's key
 - The right subtree of a node contains only nodes with keys greater than the node's key
 - The left and the right subtree each must also be a binary search tree

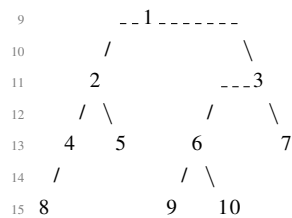
- Inorder Traversal

- Example:

```

1 public void inorderTraversal(TreeNode root) {
2     if(root != null) {
3         inorderTraversal(root.left);
4         System.out.println(root.data + " ");
5         inorderTraversal(root.right);
6     }
7 }

```



8 > 4 > 2 > 5 > 1 > 9 > 6 > 10 > 3 > 7

– Inorder Traversal of a Binary Search Tree will always give you nodes in a sorted manner

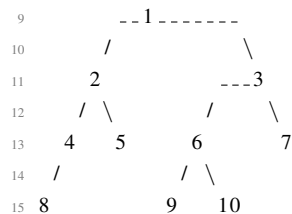
- Preorder Traversal

– Example:

```

1 public void preorderTraversal(TreeNode root) {
2     if(root != null) {
3         System.out.println(root.data + " ");
4         preorderTraversal(root.left);
5         preorderTraversal(root.right);
6     }
7 }

```



1 > 2 > 4 > 8 > 5 > 3 > 6 > 9 > 10 > 7

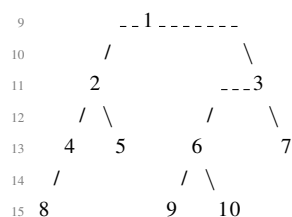
- Postorder Traversal

– Example:

```

1 public void postorderTraversal(TreeNode root) {
2     if(root != null) {
3         postorderTraversal(root.left);
4         postorderTraversal(root.right);
5         System.out.println(root.data + " ");
6     }
7 }

```



17 8 > 4 > 5 > 2 > 9 > 10 > 6 > 7 > 3 > 1

- Level Order Traversal

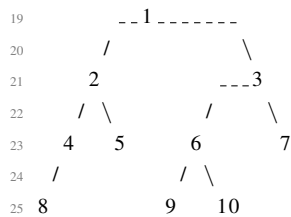
- Breadth-first search (Section 2.4):

- Example:

```

1 public void levelorderTraversal(TreeNode root) {
2     if (root != null) {
3         return;
4     }
5     Queue<TreeNode> queue = new LinkedList<>();
6     queue.add(root);
7     while (!queue.isEmpty()) {
8         TreeNode node = queue.remove();
9         System.out.println(node.data + " ");
10        if (node.left != null) {
11            queue.add(node.left);
12        }
13        if (node.right != null) {
14            queue.add(node.right);
15        }
16    }
17 }

```



27 1 > 2 > 3 > 4 > 5 > 6 > 7 > 8 > 9 > 10

2.3 Divide and Conquer

- Binary Search

- $O(\log(N))$

- It is a divide and conquer algorithm. It divides a large array into two smaller sub-arrays and the recursively or iteratively operate the subarrays. But instead of operating on both subarrays, it discards one subarray and continues on the other one. Needs to be sorted³.

- Case 1: if target == A[mid], return mid

- Case 2: if target < A[mid], right = mid - 1

- Case 3: if target > A[mid], left = mid + 1

- Iterative solution

```

1 public int binarySearch(int[] A, int x) {
2     int left = 0;
3     int right = A.length - 1;
4     while (left <= right) {
5         int mid = (left+right)/2;
6         if (x == A[mid]) {
7             return mid;

```

³<https://www.techiedelight.com/binary-search/>


```

8         }
9         else if (x < A[mid]) {
10             right = mid - 1;
11         }
12         else {
13             left = mid + 1;
14         }
15     }
16 }

```

– Recursive solution

```

1 public int binarySearch(int[] A, int left, int right, int x) {
2     if (left > right) {
3         return -1;
4     }
5     int mid = (left+right)/2;
6     if (x == A[mid]) {
7         return mid;
8     }
9     else if (x < A[mid]) {
10         return binarySearch(A, left, mid-1, x);
11     }
12     else {
13         return binarySearch(A, mid+1, right, x);
14     }
15 }

```

• Maximum Sum Subarray

- Given an array of integers, find maximum sum subarray among all possible subarrays
- Example: [2, -4, (1, 9, -6, 7), -3]
- Use divide and conquer:

- * $O(N * \log(N))$
- * Divide the array into two equal subarrays
- * Recursively calculate the max subarray sum for left subarray
- * Recursively calculate the max subarray sum for right subarray
- * Find the max subarray sum that crosses mid element
- * Return the max of above three sums

```

1 public int maxSum(int[] A, int left, int right) {
2     if (right == left) {
3         return A[left];
4     }
5     int mid = (left+right)/2;
6     int leftMax = Integer.MIN_VALUE;
7     int sum = 0;
8     for (int i = mid; i >= left; i--) {
9         sum += A[i];
10        if (sum > leftMax) {
11            leftMax = sum;
12        }
13    }
14    int rightMax = Integer.MIN_VALUE;
15    sum = 0;
16    for (int i = mid+1; i <= right; i++) {
17        sum += A[i];
18        if (sum > rightMax) {

```

```

19         rightMax = sum;
20     }
21 }
22 int maxLeftRight = Integer.max(maxSum(A, left, mid), maxSum(A, mid+1, right
    ));
23 return Integer.max(maxLeftRight, leftMax+rightMax);
24 }

```

- Power function

- Implement pow(x,n)
- Use divide and conquer:

* $O(N)$

```

1 public int pow(int x, int n) {
2     if(n==0) {
3         return 1;
4     }
5     if((n&1)==1) { // odd
6         return x * pow(x, n/2) * pow(x, n/2);
7     }
8     else { // even
9         return pow(x, n/2) * pow(x, n/2);
10    }
11 }

```

* Optimizing ($O(\log(N))$):

```

1 public int pow(int x, int n) {
2     if(n==0) {
3         return 1;
4     }
5     int pow = pow(x, n/2);
6     if((n&1)==1) { // odd
7         return x * pow * pow;
8     }
9     else { // even
10        return pow * pow;
11    }
12 }

```

- Find frequency of each element in a sorted array

- Split the array to two equal halves and recur for both of the halves. The base condition checks if the last element of the subarray is the same as its first element. If they are equal, then that implies that all elements in the subarray are similar (since the array is sorted) and we update the element count by number of elements in the subarray⁴.
- $O(m * \log(n))$, with m the number of distinct elements in the array and n the size of the input.

```

1 public void findFrequency(int[] A, int left, int right, Map<Integer, Integer>
    freq) {
2     if(left > right) {
3         return;
4     }
5     if(A[left] == A[right]) {
6         Integer count = freq.get(A[left]);
7         if(count == null) {
8             count = 0;
9         }

```

⁴<https://www.techiedelight.com/find-frequency-element-sorted-array-containing-duplicates/>

```

10         freq.put(A[left], count+(right-left+1));
11         return;
12     }
13     int mid = (left+right)/2;
14     findFrequency(A, left, right, freq);
15     findFrequency(A, mid+1, right, freq);
16 }

```

2.4 Breadth-first search

BFS is an algorithm for traversing (or searching) trees or graph data structures. It starts at the root and explores the neighbor nodes first, before moving on to the next level neighbors⁵.

- Applications of BFS:

- Finding the shortest path between two nodes u and v , with path length measured by number of edges (advantage over depth first search)
- Testing a graph bipartiteness
- Minimum Spanning Tree for unweighted graph
- Finding nodes in any connected component of a graph
- Ford-Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree

- Implementation:

- Queue
- Finds the shortest path
- Requires more memory than a DFS

- In matrix:

```

1 from collections import deque
2 def bfs(matrix):
3     if not matrix:
4         return []
5     rows, cols = len(matrix), len(matrix[0])
6     visited = set()
7     directions = ((0,1),(0,-1),(1,0),(-1,0))
8     for i in range(rows):
9         for j in range(cols):
10             traverse(i,j)
11
12 def traverse(i,j):
13     queue = deque([(i,j)])
14     while queue:
15         curr_i, curr_j = queue.popleft()
16         if (curr_i, curr_j) not in visited:
17             visited.add((curr_i, curr_j))
18             for direction in directions:
19                 next_i = curr_i + direction[0]
20                 next_j = curr_j + direction[1]
21                 if 0 <= next_i < rows and 0 <= next_j < cols:
22                     queue.append((next_i, next_j))

```

⁵<https://www.techiedelight.com/breadth-first-search/>

2.5 Depth-first search

We start at the root (or another arbitrarily select node) and explore each branch completely before moving on to the next branch. DFS is often preferred if we want to visit every node in the graph. In DFS, we visit a node a and then iterate through each of a 's neighbors. When visiting a node b that is a neighbor of a , we visit all of b 's neighbors before going to a 's neighbors. That is, a exhaustively searches b 's branch before any of its other neighbors. Note that pre-order and other forms of tree traversal are a form of DFS. The key difference is that when implementing this algorithm for a graph, we must check if the node has been visited [1].

```
1 void search(Node root) {
2     if(root == null) {
3         return;
4     }
5     visit(root);
6     root.visited = true;
7     for each(Node n in root.adjacent) {
8         if(n.visited == false) {
9             search(n);
10        }
11    }
12 }
```

2.6 Quicksort

Uses the divide and conquer technique (Section 2.3).

- Implementation:

- Find a pivot p (usually the first element)
- If the elements of array x are rearranged so that p is put in position j and that the following conditions are taken into account:
 - * All the elements in between positions 0 and $j - 1$ are smaller or equal to p
 - * All the elements in between positions $j + 1$ and $n - 1$ are higher than p
- Then p will be kept in position j at the end of sorting
- Repeat the process to subarrays $x[0..j - 1]$ and $x[j + 1..n - 1]$

```
1 def swap(i, j):
2     aux = elements[i]
3     elements[i] = elements[j]
4     elements[j] = aux
5
6 def partition(start, end):
7     i = start
8     for j in range(start, end):
9         if elements[j] <= elements[end]:
10            swap(i, j)
11            i += 1
12     swap(i, end)
13     return i
14
15 def quickSort(start, end):
16     if start >= end:
17         return
18     pivot = partition(start, end)
19     quickSort(start, pivot - 1)
```

```

20     quickSort(pivot+1, end)
21
22 if __name__ == "__main__":
23     quickSort(0, len(elements)-1)
24     print(elements)

```

- Big O:
 - Worst case: $O(N^2)$
 - Average case: $O(N * \log(N))$
 - Space complexity: $O(\log(N))$
- Performance depends largely on pivot selection

2.7 Mergesort

Uses the divide and conquer technique (Section 2.3)⁶.

- Divide
 - If q is the half-array point between p and r , then we can split the subarray $A[p..n]$ into two arrays $A[p..q]$ and $A[q + 1..n]$
- Conquer
 - We try to sort both subarrays $A[p..q]$ and $A[q + 1..r]$. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.
- Combine
 - When the conquer step reaches the base step and we get two sorted subarrays $A[p..q]$ and $A[q + 1..r]$ for array $A[p..r]$, we combine the results by creating a sorted array $A[p..r]$ from two sorted subarrays $A[p..q]$ and $A[q + 1..r]$

```

1 def merge(array, p, q, r):
2     arrayA = array[p:q+1]
3     arrayB = array[q+1:r+1]
4     total = r - p + 1
5     result = [0] * total
6     pA = pB = pR = 0
7     while pR < total:
8         if pA == len(arrayA) - 1:
9             result[pR:total] = arrayB[pB:len(arrayB)]
10            return result
11        if pB == len(arrayB) - 1:
12            result[pR:total] = arrayA[pA:len(arrayA)]
13            return result
14        if arrayA[pA] <= arrayB[pB]:
15            result[pR] = arrayA[pA]
16            pR += 1
17            pA += 1
18        else:
19            result[pR] = arrayB[pB]
20            pR += 1
21            pB += 1
22    return result

```

⁶<https://medium.com/@paulsoham/merge-sort-63d75df76388>

```

23
24 def mergeSort(array , p, r):
25     if p >= r:
26         return
27     q = (p+r)//2
28     mergeSort(array , p, q)
29     mergeSort(array , q+1, r)
30     merge(array , p, q ,r)

```

- Big O:

- Worst case: $O(N * \log(N))$
- Average case: $O(N * \log(N))$
- Space complexity: $O(N)$

2.8 Dynamic Programming

Taking a recursive algorithm and finding the overlapping subproblems. Use memoization!

- Fibonacci without DP

```

1 int f(int i) {
2     if(i==0) return 0;
3     if(i==1) return 1;
4     return f(i-1) + f(i-2);
5 }

```

```

7           -----f (4) -----
8           /                      \
9         -----f (3) -----      -----f (3) -----
10        /      \                /      \
11      --f (2) --  --f (2) --    --f (2) --  --f (2) --
12     /    \      /    \      /    \      /    \
13   f(1)  f(1)  f(1)  f(1)  f(1)  f(1)  f(1)  f(1)

```

- DP with top-down approach

```

1 int f(int n) {
2     return f(n, new int[n+1]);
3 }
4
5 int f(int i, int[] memo) {
6     if(i==0||i==1) return i;
7     if(memo[i]==0) {
8         memo[i] = f(i-1, memo) + f(i-2, memo);
9     }
10    return memo[i];
11 }

```

- DP with bottom-up approach

```

1 int f(int n) {
2     if(n==0) return 0;
3     int a = 0;
4     int b = 1;
5     for (int i=2; i < n; i++) {
6         int c = a + b;
7         a = b;
8         b = c;
9     }
10    return a + b;
11 }

```

3 Python

3.1 Set

Collection of items. Uses a hash.

- Creating a set

```
1 set()
```

- Checking if item is in set

- Time complexity: $O(1)$ on average
- Worst case: $O(N)$

- Adding elements

```
1 set.add()
```

- Union

- Two sets can be merged using `union()` function
- $O(\text{len}(s1) + \text{len}(s2))$

- Intersection

- $O(\min(\text{len}(s1), \text{len}(s2)))$

- Difference

```
1 difference()
```

- Examples

```
1 a = set()
2 a.add('a')
3 a.add('b')
4 a.add('a')
5 'a' in a #return True
6 a.remove('b')
```

3.2 List

- `[]` vs `list()`

- `[]` is literal and `list()` is function call
- `[]` is faster because it doesn't involve loading and calling a separate function
- `[(a, b, c)]` returns `[(a, b, c)]` and `list((a, b, c))` returns `[a, b, c]`
- `list()` accepts only iterables as argument

- Examples

```
1 sea_creatures = ['shark', 'fish', 'squid', 'mantis', 'anemone']
2 sea_creatures[1:4] # returns ['fish', 'squid', 'mantis']
3 sea_creatures[:3] # returns ['shark', 'fish', 'squid']
4 sea_creatures[2:] # returns ['squid', 'mantis', 'anemone']
5
6 numbers = [0,1,2,3,4,5,6,7,8,9,10,11,12]
7 numbers[1:11:2] # returns [1,3,5,7,9]
8 numbers[::3] # returns [0,3,6,9,12]
```

```

9 numbers + [13,14] # O(1). Worst case: Double the size, O(N)
10
11 letters = ['a','b','c']
12 letters*2 # returns ['a','b','c','a','b','c']
13
14 del sea_creatures[1:3] # returns ['shark', 'mantis', 'anemone']. O(N)
15 sea_names = [['shark', 'octopus', 'squid'],['Sammy', 'Jesse', 'Drew']]
16 sea_names[0][0] = 'shark'
17 sea_names[1][2] = 'Drew'

```

- `+=` vs `append()`

- `append()` is less time and space expensive because it doesn't have to create a new list every time

- `min` and `max`

- Both costs $O(N)$

- `in` vs `find()`

- “`x in s`” costs $O(N)$ and returns *true* or *false*
- `find()` returns the index. `l.find(':')`

- `insert()`

- `list.insert(index, element)`
- $O(N)$

3.3 Dict

- Examples

```

1 thisDict = {"brand": "Ford",
2             "model": "Mustang",
3             "year": 1964}
4 x = thisDict["model"] # or
5 x = thisDict.get("model")
6 thisDict["year"] = 2018
7
8 for x in thisDict:
9     print(x) # prints key name
10 for x in thisDict:
11     print(thisDict[x]) # prints values
12 for x in thisDict.values():
13     print(x)
14 for x,y in thisDict.items():
15     print(x,y)
16
17 # Checks if "model" is present in the dictionary
18 "model" in thisDict
19
20 # Adds item
21 thisDict["color"] = "red"
22
23 # Removes item
24 thisDict.pop("model")
25
26 # Clear dict
27 thisDict.clear()

```


- How Python dict works⁷

- Implemented as hash tables
- Uses open addressing to resolve hash collisions
- Contiguous block of memory
- Each slot in the table can store one and only one entry
- Each entry in the table is actually a combination of three values $\langle hash, key, value \rangle$
- When a new dict is initialized it starts with 8 slots
- When adding entries to the table, we start with a slot that is based on the hash of the key
- If that slot is empty, the entry is added to the slot
- If that slot is occupied, compares the hash and the key of the entry in the slot against the hash and the key of the current entry to be inserted. If both match, then it thinks the entry already exists and gives up. If they don't match, it starts probing.
- Probing means it searches slot by slot to find an empty slot
- The dict will be resized if it is two-thirds full

3.4 Sort

Python's "`list.sort()`" and "`sorted()`" methods use timesort, a stable mergesort and insertionsort hybrid. The first sorts in-place and the second returns a new sorted list. The first is a bit more efficient if you don't need to keep the original list. They both take a parameter "key", which is a key function (i.e. if you want to sort by a specific property).

- Time complexity

- Worst case: $O(N * \log(N))$
- Best case: $O(N)$
- Average case: $O(N * \log(N))$

- Space complexity

- $O(N)$

- Examples

```
1 sorted([5,2,3,1,4])
2
3 a = [5,2,3,1,4]
4 a.sort()
5
6 student_tuples = [('joe', 'A', 15),
7                   ('jane', 'B', 12),
8                   ('dave', 'B', 10)]
9 sorted(student_tuples, key = lambda student: student[2])
10
11 class Student:
12     def __init__(self, name, grade, age):
13         self.name = name
14         self.grade = grade
15         self.age = age
```

⁷<https://stackoverflow.com/questions/21595048/how-python-dict-stores-key-value-when-collision-occurs>

```

16 student_objects = [Student('joe', 'A', 15),
17                     Student('jane', 'B', 12),
18                     Student('dave', 'B', 10)]
19 sorted(student_objects, key = lambda student: student.age)

```

3.5 Join

- Join all items in a tuple into a string, using a hash character as separator.

```

1 x = myseparator.join(myDict) # myDict is any iterable

```

- Equivalent of C's stringBuilder. Imagine you are concatenating a list of strings as shown below:

```

1 our_str = ''
2 for num in range(loop_count):
3     our_str += 'num'
4 return our_str

```

- On each concatenation, a new copy of the string is created and the two strings are copied over, character by character. Join can help you avoid this problem.

```

1 str_list = []
2 for num in range(loop_count):
3     str_list.append('num')
4 return ''.join(str_list)

```

- Big O

– $O(N)$ where N is the total length of the output

- *join* vs *split*

```

1 txt = "hello , my name is Peter , I am 26 years old"
2 x = txt.split(", ")
3 print(x)
4 # ['hello ', 'my name is Peter ', 'I am 26 years old']

```

3.6 Strip

- $O(N)$

- Example

```

1 str = "ooooo hi!! ooo"
2 str = str.strip('o')
3 print(str) # " hi!! "

```

4 Data Structures

4.1 Linked List

4.2 Hash Table

4.3 Binary Search Tree

4.4 Min-heap

4.5 Stack

Last-in, first-out (LIFO)

```
1 stack = []
2 stack.append(1) # means stack.push(1)
3 stack.pop()
4 stack[-1] # means stack.peak()
5 if stack: # means stack.isEmpty()
6     ...
```

4.6 Queue

First-in, first-out (FIFO)

```
1 queue = []
2 queue.append(1) # means queue.add(1)
3 queue.pop(0) # means queue.remove()
4 queue[0] # means queue.peak()
5 if queue: # means queue.isEmpty()
6     ...
7
8 # or
9 from collections import deque
10 q = deque()
11 q.append('c')
12 q.popleft()
```

5 Object-oriented Programming

5.1 Concrete Class, Abstract Class and Interface

Concrete class can be instantiated because it provides (or inherits) the implementation for all of its methods. Abstract class cannot be instantiated because at least one method has not been implemented. Interface must contain only method signatures and static members.

- In abstract class, keyword "abstract" is mandatory to declare a method as an abstract. In interface, "abstract" is optional.
- Interface can only have public methods
- One concrete class extends only one abstract class, but can implement many interfaces

```
1 interface Readable {  
2     public void open();  
3     public void read();  
4 }  
5  
6 abstract class Animal() {  
7     String species;  
8     public void eat();  
9     abstract void walk();  
10 }
```

- Abstract classes are used when we require classes to share a similar behavior (or methods). However, if we need classes to share method signatures, and not the method themselves, we should use interfaces
- Interfaces can add to the existing functionality of a class. They are not necessarily integral to the identity of the classes that reference them

5.2 Static

The keyword "static" is used to refer to common property of all objects. Can be applied to variables, methods and nested classes. If a variable is static, it is assigned memory once and all objects of the class access the same variable⁸.

```
1 class Citizen {  
2     static String country();  
3     Citizen() {  
4         country = "Brazil";  
5     }  
6     public static void setCountry(String c) {  
7         country = c;  
8     }  
9 }
```

5.3 Inheritance

- Super class: The class whose features are inherited is known as super class
- Sub class: The class that inherits the other
- Reusability: Inheritance supports reusability
- In java, use *@override* and *extends*

⁸<https://www.educative.io/edpresso/how-to-use-static-keyword-in-java>

5.4 Aggregation, Association and Composition

- Aggregation

- Relationship in which an object contains one or more other subordinate objects as part as of its state. The subordinate has independent existence separate from their containing object. Has-a relationship⁹.
- "I have an object in which I've borrowed from someone else". When Foo dies, Bar may live on¹⁰.

```
1 public class Foo {  
2     private Bar bar;  
3     Foo(Bar bar) {  
4         this.bar = bar;  
5     }  
6 }
```

- Composition

- "I own an object and I am responsible for its lifetime". When Foo dies, so does Bar.

```
1 public class Foo {  
2     private Bar bar = new Bar();  
3 }
```

- Association

- "I have a relationship with an object". Foo uses Bar.

```
1 public class Foo {  
2     void Baz(Bar bar);  
3 }
```

- Dependency

- One object is dependent on another object for its specification or implementation.
- Dependency injection is a technique whereby one object supplies the dependencies of another object¹¹.

5.5 Coupling and Cohesion

- Coupling: Degree of interdependence between software modules
- Degree of which the elements of a module belong together
- Desirable: Less coupling, more cohesion

5.6 Polymorphism

The ability of an object reference to be used as if it referred to an object with different forms. Examples: Class inheritance and interface inheritance. The most common use occurs when a parent class reference is used to refer to a child class object¹².

⁹<https://www.cs.kent.ac.uk/people/staff/djb/oop/glossary.html>

¹⁰<https://tinyurl.com/y2wjx3f9>

¹¹<https://medium.com/@harivigneshjayapalan/dagger-2-for-android-beginners-di-part-i-f5cc4e5ad878>

¹²https://www.tutorialspoint.com/java/java_polymorphism.htm

5.7 Function overloading

Example:

```
1 int Volume(int s);  
2 int Volume(double r, int h);  
3 int Volume(long l, int b, int h);
```

6 Design Patterns

6.1 Template Method

6.2 Strategy

6.3 State

6.4 Composite

6.5 Singleton

6.6 Builder

6.7 Factory

7 Architecture

7.1 API

- Application Programming Interface
- API allows various software applications to communicate with one another over the internet.
- The most common methods seen in APIs are:
 - GET (asks to retrieve a resource)
 - POST (asks to create a new resource)
 - PUT (asks to edit/update an existing resource)
 - DELETE (asks to delete a resource)
- API requests are made using HTTP
- Endpoint tell the server which resources that the client wants to interact with

7.2 Remote Procedure Call

RPC is a protocol that one program can use to request a service from another program located in another computer on a network. It uses client-server model. RPC is a synchronous operation. Examples: REST, SOAP.

7.3 Caching

7.4 Database Partitioning (Sharding)

7.5 Non Relational Databases

References

- [1] G. L. McDowell. *Cracking the coding interview: 189 programming questions and solutions*. 2015.