1. **Permutations**

Given a string S, find all permutations of the string that contain all characters of the original string.

**Example:**

|  |
| --- |
| Input: S = “ABC”  Output: {“ACB”, “CAB”, “CBA”, “BCA”, “BAC”} |

Think about an algorithm that can perform this task. You may implement it in Java, but you can also write it down in pseudocode or, at least, try to visualize the execution of an algorithm.

ANSWER:

|  |
| --- |
| **algorithm** permutations(string S)  N <- length(S)  permute(S, 0, N-1)  **def** permute(S, start, end):  **if** start = end **then**  **output** S  **else**  **for** i <- start **to** end **do**  swap(S, start, i)  permute(S, start+1, end)  swap(S, start, i)  **def** swap(S, start, target):  temp <- S[start]  S[start] <- S[target]  S[target] <- temp |
|  |

What is the complexity of this algorithm?

ANSWER:

🡪 the number of permutations varies with the factorial of the size of the string.

Now let’s change the algorithm a little bit.

Given a string S, find all *k* sized permutations of the string that contain all characters of the original string. Where *k* is a positive integer and that represents the length of each resulting permutation.

**Example:**

|  |
| --- |
| Input: S = “ACB”  k = 2  Output: {“AB”, “AC”, “BA”, “BC”, “CB”, “CA”} |

Seeing that the size of the output changed, surely the complexity must also have changed. What do you think?

ANSWER:

Each step generates permutations, where . Depending on the relationship between and , this might, in the worst case, be if .

What is the complexity of this algorithm?

ANSWER:

1. **Minimum binary subarray**

Given binary array find the length of a subarray with the *minimum* number of 1s.

Note: There is at least a single 1 present in the array.

**Example:**

|  |
| --- |
| Input : arr[] = {1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1}  Output : 3  Minimum length subarray of 1s is {1, 1, 1}.  Input : arr[] = {0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1}  Output : 2  Minimum length subarray of 1s is {1, 1}. |

Think about an algorithm that can perform this task. You may implement it in Java, but you can also write it down in pseudocode or, at least, try to visualize the execution of an algorithm.What is the complexity of the solution you found? Can you think of a way to improve the complexity?

ANSWER:

A “trivial” approach would be to split the array into all possible substrings and check each of them individually. This takes operations. A much faster approach is given below:

|  |
| --- |
| **def** getMinSubarrayLength(int arr[])  N <- length(arr)  count <- 0  result <- N  **for** i <- 0 **to** N **do**  **if** arr[i] = 1 **then**  count <- count + 1  **else**  **if** count >= 2 **then**  result <- min(result, count)  count <- 0  **output** result |

By traversing the array 1-by-1, this algorithm takes comparisons to converge to a solution.

1. **List comparison**

Below you find the pseudo code of four algorithms to compare two lists (each element is unique in the list). Have a look the implementations.

Analyse the run time of each of the algorithms? Consider different implementations of the list (e.g. Linked List, Array List) and describe the run time for each of these implementations.

|  |
| --- |
| compare1 (data a, data b):  for each i in a  found <-false  for each j in b  if i = j  found <- true  if not found  return false  return true |

|  |
| --- |
| compare2 (data a, data b):  for each i in a  found <-false  for each j in b  if i = j  found <- true  break;  if not found  return false  return true |

|  |
| --- |
| compare3 (data a, data b):  for each i in a  found <-false  for each j in b  if i = j  found <- true  remove j from b;  if not found  return false  return true |

|  |
| --- |
| compare4 (list a, list b):  a <- sort(a)  b <- sort(b)  for (int i = 0; i < len(a) && i < len(b);i++)  if (a[i] != b[i])  return false  return true |

|  |
| --- |
| compare5 (list a, list b):  b <- sort(b)  for (int i = 0; i < len(a) && i < len(b);i++)  if(not b.find(a[i]))  return false  return true |

ANSWER:

For the sake of simplicity, let’s assume that

Compare1 – for both an ArrayList and a LinkedList as it traverses all elements of both lists

Compare2 – for both an ArrayList and a LinkedList, same as above, but the traversal of the list is halted prematurely if the element is found which would result in half as many operations.

Compare3 – for a LinkedList, as the removal is constant time. For an ArrayList, as the removal would require time each iteration of the inner loop if b is found.

Compare4 – as sorting an array takes in the average case. The traversal is for both an ArrayList and a LinkedList.

Compare5 – for either an ArrayList or LinkedList, as sorting is complexity and finding an element by value is .

1. **Graph algorithms**

In a graph, let the value of a neighbourhood be the sum of the values of the node itself and each of its neighbors. The task is now to find the nth largest neighbourhood in a graph.

Below you find the pseudo code of 3 different algorithms to calculate this value.

On which parameters does the run time depend?

What is the run-time in big O notation?

What type of list() should be used in find1 and find2? How does it affect the run time?

How does the memory consumption differ in the three algorithms?

|  |
| --- |
| find1(n,graph):  values = list()  for each node node:  sum = node.getValue()  for e in graph.getNeighbours(node):  sum += e.getNode().getValue()  values.add(sum)  values.sort()  return values[n] |

|  |
| --- |
| find2(n,graph):  values = list()  for each node node:  sum = node.getValue()  for e in graph.getNeighbours(node):  sum += e.getNode().getValue()  values.add(sum)  i = 0  max = 0  while (i < n):  max = 0  for each e in values:  if (e > max):  max = e  remove max from values  return max |

|  |
| --- |
| find3(n,graph):  i = 0  max = 0  while (i < n):  max = 0  for each node node not marked:  sum = node.getValue()  for e in graph.getNeighbours(node):  sum += e.getNode().getValue()  if (sum > max):  max = sum  mark max  return max |

ANSWER:

The runtime for all algorithms depends on the number of vertices as well as the average degree (number of adjacent vertices) for each vertex. In the worst case, this would be and in the best case (fully disconnected graph), this would be . For the sake of simplicity, let’s assume an average case where we have .

The first algorithm has a sort, while the complexity of creating the unsorted list is as it iterates through each vertex, and for each vertex it iterates through all of its adjacent edges. In the worst case, a vertex is adjacent to other vertices. Any list would be most suitable for this, but utilizing a partially sorted or fully sorted structure instead lead to slightly faster runtime without improving the complexity.

The second algorithm requires operations to calculate the values of each neighbourhood, but then requires an additional operations to calculate the maximum. While it is similar to the first algorithm, a linked list would be considerably better than an array-based list as linked lists have constant time removal. Using an ArrayList would increase the runtime complexity by a factor of .

The third algorithm requires only operations to return an answer.