Question 1

(i) The shortcomings of measuring an algorithm's time complexity by finding put the time elapsed between start and end time of running the algorithm is:

* It is difficult to directly compare with each other unless the experiments are performed using the same environment which has the same hardware and software.
* By using this way, it can be only done on a limited set of test inputs and might cause some of the important running times of input not included in this experiment.
* By using this way, the algorithms must be fully implemented in order to study it’s running time.

(ii) Big-O notation allows the developer to evaluate the relative efficiency of any two algorithms independent of the hardware and software. This is because this Big O notation is performed by studying the high level description of the algorithm without the need for implementation. Hence, it allows the developer to take into account all possible inputs.

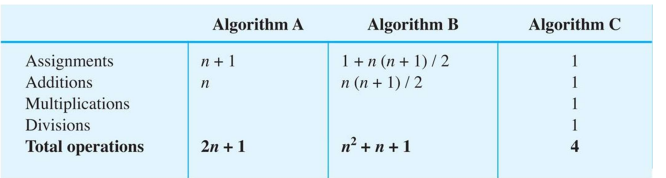
There are 2 algorithms to be analysed by counting the primitive operation:

Algorithm 1:

| sum=0  for i = 1 to n  sum = sum + i |
| --- |

Algorithm 2:

| sum = n \* (n + 1) / 2 |
| --- |



In order to derive the big O notation for the algorithms, we will ignore the smaller growing term in which:

Algorithm 1:

O(2n + 1)

O(2n) by ignoring the smaller term

O(n) by ignoring the multiplier

Algorithm 2:

O( 1)

Therefore, we can know that algorithm 2 is more efficient than algorithm1

(iii)

| **n** | **O(n)** | **O(n ^ 2)** |
| --- | --- | --- |
| 1000 | 10s | 10s |
| 2000 | 20s | 40s |

n = 1000 =10s

= 2000 =20s

n^2 = 1000000 = 10

= 4000000 = 40

4000000/1000000\*10 = 40s

b) Queue ADT is suitable when the objects need to be processed in First-In-First-Out (FIFO) principle

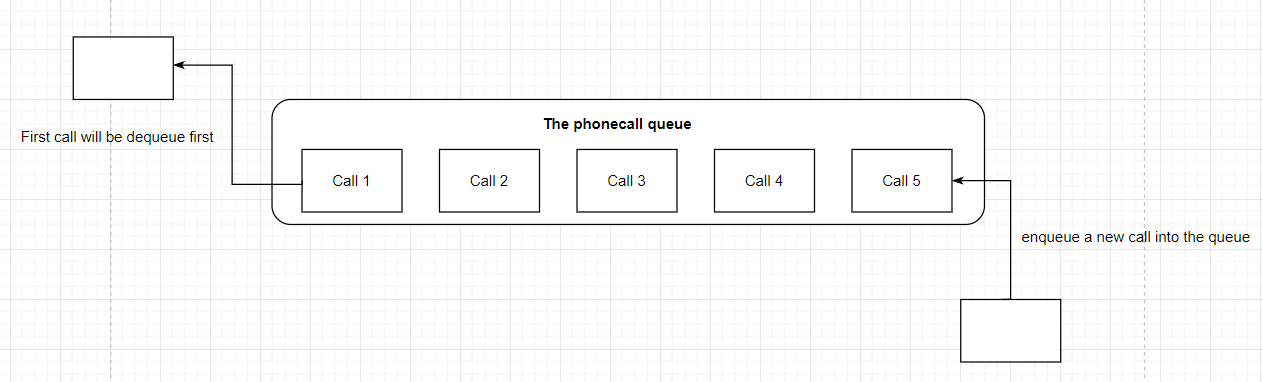
**Real-life computer application :** **Phone answering system**

**Object stored :** calls

**Reason used :**

* The reason why queue has been used for this system is because the process of answering calls will be carried out in the **first in first one (FIFO)** strategy. For example, the person who first calls will get the response first from the phone answering system followed by the call 2, call 3 and the person who calls last will get the response last.

**Example with diagram :**



c) Benefits of using abstraction:

d)

(i)

| public static void printPattern(int n) {  if (n > 0) {  printPattern(n - 1);  System.out.print("\*");  }  }  public static void drawPattern(int n) {  if (n > 0) {  drawPattern(n - 1);  }  printPattern(n);  System.out.println("");  } |
| --- |

(ii) Recursive solution is better than the iterative solution because the big O notation of **iterative solution** is **O(n ^ 2)** as it involves a nested loop where the inner loop performs the linear operation and the outer loop performs linear number of times. While the big O notation of a **recursive solution** is just **O(n)** which is more efficient than the iterative solution.

**Question 2**

a)

**ADT :** Sorted List

This is because the sorted list will be maintained in sorted order in which we can control which data member can be sorted. In order to sort the object, the sorted list must be Comparable and implement the compareTo() method. To enforce these requirements, we need to write <T extends Comparable<T>> . In the Book class, we should override the compareTo() method by the following code

public int compareTo(Book b) {

return (int) (this.title - b.title);

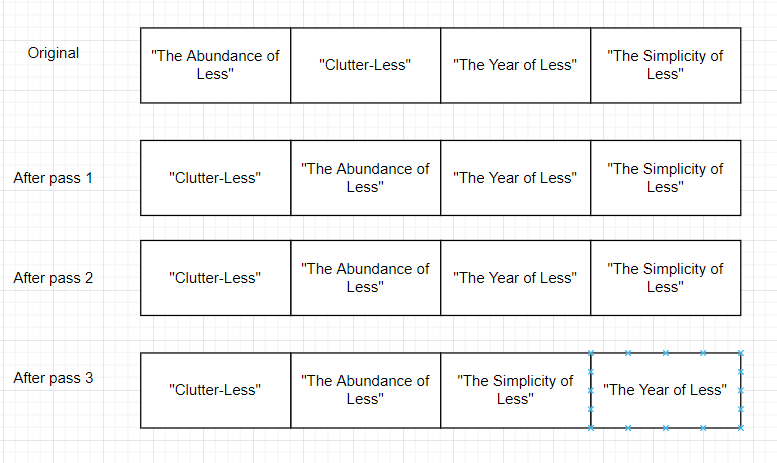
}

so that it will compare the new entry with the current entry in order to determine the correct location which is in ascending order (A-Z) by the title.

b)

(i) Insertion sort

(ii) Yes, it is suitable as insertion sort is more suitable for small array size. In this case, the usual number of book results returned is 1 - 20 which is still considered a small size. If the book is almost sorted, it will be more efficient as it requires less work to shift the elements.

(iii) 

c)

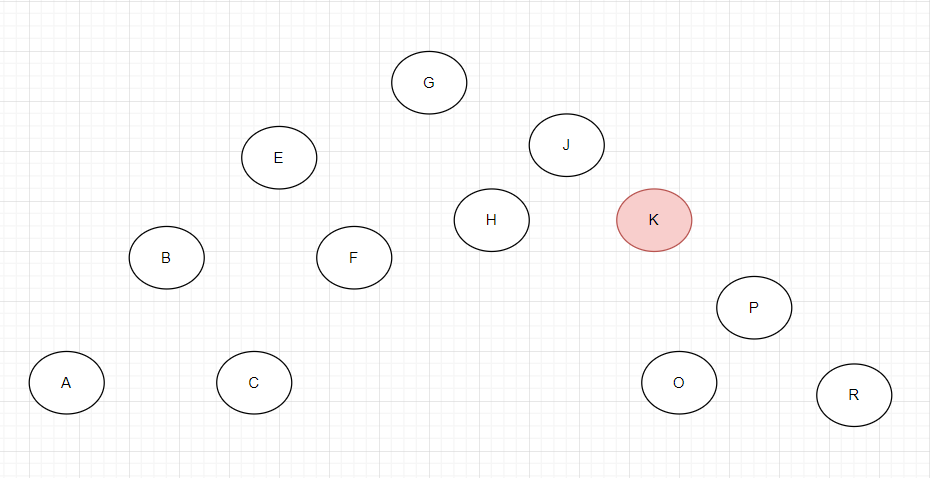
**In order**

A, B, C, E, F, G, H, J, K, M, O, P, R

**Post order**

A, C, B, F, E, H, K, O, R, P, M, J, G

(ii)



Because the “M” node has 2 children, therefore we need to replace the M node’s data with the biggest value (rightmost node) in the left subtree. In this case, the K is the one who meets the requirements so we replace “M” with “K”.

d) Hash function using Student ID

1. Compute hash code of student ID
2. Compress the hash code into the range of indices of hash table (size = 83)

Algorithms:

| Algorithm getHashIndex(studentID)  i = last 4 digits of studentID  Return i % 83 |
| --- |

Notes: As every last 4 digits of the student ID will be unique, so we no need to worry about the collision

April 2021

a)

**ADT List**

A list is a linear collection of entries of a type T which allows duplicate elements. An entry may be added at a specified position or at the end of the list. The list position starts from 1.

**add(T newEntry)**

Description : Adds newEntry to the end of the list.

Postcondition : newEntry has been added to the end of the list

**T remove(Integer givenPosition)**

Description : Removes the entry at position givenPosition within the list.

Precondition : givenPosition must be between 1 to total entries.

Postcondition : The entry at position givenPosition has been removed from the list.

Returns : The entry that was removed from the list.

**boolean contains(T anEntry)**

Description : Determines whether the list contains anEntry.

Postcondition : The list remains unchanged.

Returns : true if anEntry is in the list, or false if not.

**boolean isEmpty()**

Description : Determines whether the list is empty.

Postcondition : The list remains unchanged.

Returns : true if the list is empty, or false if not.

b)

public Interface ListInterface<T>{

public void add(T newEntry);

public T remove(Integer givenPosition);

public boolean contains(T anEntry);

public boolean isEmpty();

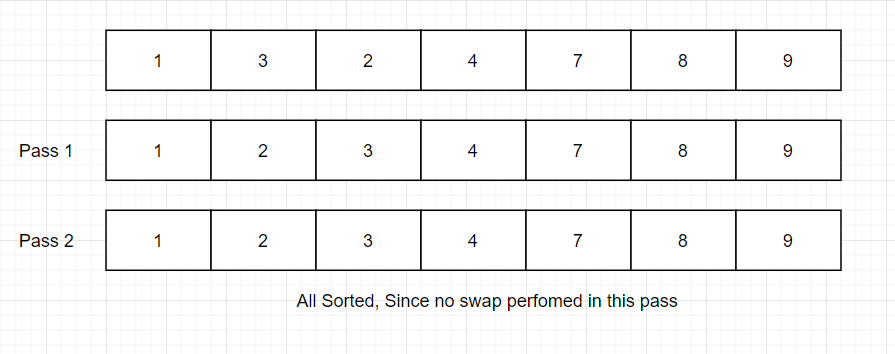
}

Question 2 answer can be found on Chapter 4 source code

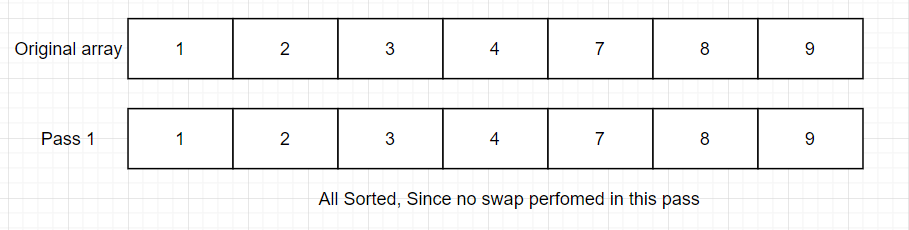
Question 3

a)

(i) The big-O of the bubble sort in the best case is O(n2). The best case for bubble sort occurs when the list is already sorted or nearly sorted. In the case where the list is already sorted, bubble sort will terminate after the first iteration, since no swaps were made. Any time that a pass is made through the list and no swaps were made, it is certain that the list is sorted.



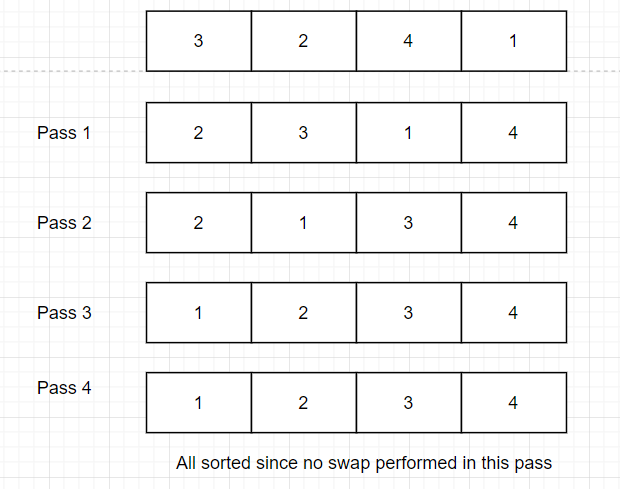
Nearly sorted array



A sorted array

In this example, we only need 2 pass which only perform swap for one time because this array is nearly sorted

(ii) The big-O of the bubble sort in the worst case is O(n^2). The absolute worst case for bubble sort is when the smallest element of the list is at the large end. Because in each iteration only the largest unsorted element gets put in its proper location, when the smallest element is at the end, it will have to be swapped each time through the list, and it wont get to the front of the list until all n iterations have occurred.



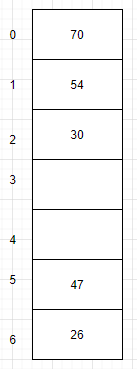
It needs 4 passes to let the array become sorted.

b) Write a recursive method that implements the Selection Sort algorithm for an array of integers. Include any auxiliary methods that you need.

| private static <T extends Comparable<T>> void selectionSort(T[] a, int n) {  for (int index = 0; index < n - 1; index++) {  int indexOfNextSmallest = getIndexSmallest(a, index, n - 1);  swap(a, index, indexOfNextSmallest);  }  }  //returns the index of the smallest entry in the subarray first...last  private static <T extends Comparable<T>> int getIndexSmallest(T[] a, int first, int last) {  T min = a[first];  int indexOfMin = first;  for (int index = first + 1; index <= last; index++) {  if (a[index].compareTo(min) < 0) {  min = a[index];  indexOfMin = index;  }  }  return indexOfMin;  }  private static <T> void swap(T[] a, int i, int j) {  T temp = a[i];  a[i] = a[j];  a[j] = temp;  } |
| --- |

Question 4

| **Key** | **Hash Index (key % 7)** |
| --- | --- |
| 47 | 5 |
| 30 | 2 |
| 26 | 5 |
| 70 | 0 |
| 54 | 5 |



b)

**Preorder**

63, 20, 11, 54, 38, 70

**Inorder**

11, 20, 38, 54, 63, 70

**Postorder**

11, 38, 54, 20, 70, 63

c)

| public boolean contains(T anEntry) {  boolean found = false;  Node tempNode = firstNode;  while (!found && (tempNode != null)) {  if (anEntry.compareTo(tempNode.data) <= 0) {  found = true;  } else {  tempNode = tempNode.next;  }  }  if (tempNode != null && tempNode.data.equals(anEntry)) {  return true;  } else {  return false;  }  } |
| --- |