Q1.a. [b = ta is not valid , hence it is fair to say it does not create any references]

No of objects of A = 1 (Created when B is constructed)

reference count = 0 (The only reference is ta which does not exist on return from B's constructor.)

No of objects of B = 1 (A single new B created in main.)

reference count = 1 (Only b2 refers to it.)

No of objects of C = 0 (There is no objects of C created in main.)

reference count = 0

Q1.b. In derived class B, m() is redeclared with the same signature as in the parent class A. This overrides the base class's method m().

In derived class B, an additional method m() is declared, which has a signature different from the other available m(). This is overloading.

v is hidden: A child class can define the variable with the same name present in the superclass. This hides the otherwise directly usable variable v declared in the parent's class (which is now accessible only through the super qualifier).

Q1.c. If we perform the given modification, instead of setting v associated with class B, v associated with the Parent class A will be set to 5. The value of v associated with B would remain unmodified.

Q1.d. When a class extends another class, it inherits methods and types from the parent, which is well defined. However, the parent's behaviour must be independent of any child for proper abstraction. (And there may be more than one, so which would it even refer to. In fact, a child need not even exist.)

Q1.e. Instead of building all classes from scratch, one may build upon an abstraction, reusing behaviour, and modifying it where necessary. Thus, multiple similar classes may all share the parent's behaviour, modifying it incrementally as per their own specifications. This modularity also allows the subclass to remain unconcerned with many details of parent's implementation (which it may hide).

Q1.f. This ensures that when a constructor of a child class is invoked, it can rely on all the fields in its superclass being initialised. It separates a child's concerns from its parent's. Recall that only a parent is even aware of private data, and only the parent may initialize that part.

Q2 $f(n) = n * (\log n)^2 + n * (\log n) + n^3$

We know that for $n > 1$,

$\log(n) < n$ ...(1)

$\log(n) > 0$ ...(2)

Proving O(n): Using (1),

$$n * \log(n)^2 < n^3 \quad \text{for } n > 1 \qquad ...(3)$$

$$n^2 * \log(n) < n^3 \quad \text{for } n > 1 \qquad ...(4)$$

Combining (3) and (4) we get, for all n > 1

$$f(n) \leq 3 * n^3 \qquad ...(5)$$

Thus using the definition of big-O, we can say that $f(n) = O(n^3)$ (with k = 3 and $n_0 = 1$).

Proving Ω(n): Using (2),

we know that all terms in f(n) will be positive for n > 1.

Thus we can drop the first two terms and say that for all n >1,

$$f(n) \geq n^3 \qquad ...(6)$$

Thus using the definition of Ω we can say that f(n) = Ω(n) (with k = 1 and $n_0 = 1$), . As we have proved $f(n) = O(n^3)$ and $f(n) = \Omega(n^3)$, we can say that $f(n) = \Theta(n^3)$

Q3 The postcondition is abs=|x|. This is true for every value of x. (The input precondition "True" implies no restriction on the input value. (In this mathematical exercise we are not concerned with the variable's type.)

Q4.a

Contrapositive stmt:

If n is a perfect square, n is a positive integer such that n%4 is neither 2 nor 3.

Alternate allowable:

If n is a perfect square, either n is not a positive integer or n%4 is neither 2 nor 3.

Proof:

If n is a perfect square integer, $n = m^2$ (where m is an integer)

Let's say m%2 = k, meaning k = 0 or 1. m is (2p + k) for some integer p. $m^2 = 4p^2 + 4pk + k^2 = n$

The first two terms are divisible by 4, meaning $n\%4 = k^2$, which is 0 or 1 as k is 0 or 1. Hence it is neither 2 nor 3. QED.

Q4.b

Contrapositive stmt:

If x and y are two integers such that none of them is even, then x * y is not even.

Proof:

Since both x and y are odd.

x = 2d + 1 (for some integer d)

y = 2e + 1 (for some integer e)

x * y = (2d + 1)(2e + 1)

= 4de + 2d + 2e + 1

= 2(2de + d + e) + 1 = 2p + 1 (letting 2de + d + e = p)

Hence x*y is odd. QED.

Q5 Proof

Loop Invariant:

Let j denote the jth iteration of for loop.

At the start of each iteration 'j' of the for loop, “min” contains smallest among A[0 … j-1]

Basis (1 mark):

According to algorithm, min_0 is initialized to A[0] (that is the 0th element of array)

Indeed A[0] is minimum element from 0 to 0 index. Basis holds.

Induction Hypothesis:

Assume that the algorithm correctly finds “min” that is the minimum element of array from 0 to kth index.

Induction Step::

Prove correctness for min for (k+1)th iteration. According to the algorithm, if we are in the loop for (k+1)th iteration, we have the minimum of the 0 to kth element with us. (Induction Hypothesis)

if(A[k+1] < min_k)

min_(k+1) = A[k+1]

So, min_(k+1) = smaller(min_k, A[k+1]) = smaller(min(A[0..k]), A[k+1])) = min(A[0..k+1]). (In words, if A[k+1] is smaller than the minimum of A[0..K]), A[k+1] is the minimum of A[0..k+1], Otherwise the minimum of A[0..k] is also the minimum of A[0..k+1]).

At the end of the loop, k = n-1, meaning that min = min(A).

Analysis

Since the loop runs from 1 until n-1, there are always n-1 iterations. Each iteration takes constant time, whether the if condition passes or fails. Thus the number of steps are always kn. As a result, both upper and lower bounds are **O(n)** and **Ω(n)**. Hence the algorithm take $\Theta(n)$.

Q6 Correctness means find if v exists AND fail if it does not.

Base Cases:-

If(lo > hi), n = 0; find correctly fails, because there are 0 elements to search.

If(lo == hi), n = 1; find is correct: hi = lo = mid. f A[mid] = v then return mid, otherwise fail. Correct.

Induction Hypothesis:-
Let our algorithm works correctly for all k <= n.

Inductive Step:-
Prove correctness for k = n+1. (n+1 > 1)

Divide n+1 elements into 3 disjoint sections: mid, lo to mid-1, and mid+1 to hi. The algorithm is correct if all three sections are correctly processed, and it fails if all three sections show failure and it succeeds if any section succeeds (this part directly follows from the code).

Now, to prove that all three sections are correct. The first section is correctly answered by the base case of n = 1. The other two section are each of the form lo > hi or lo < hi. If lo > hi, they are correctly processed by the base case. If lo < hi, there are n or fewer items, as the first section always has one item and the three section are disjoint. and it is correct by inductive hypothesis.

Hence it is proved by induction that our algorithm works correctly for all cases.

Time Complexity Analysis:-
Recurrence Relation :- T(n) = T(n/2) + k, where k is some constant.
Solving recurrence relation:-
T(n) = T(n/2) + k = T(n/4) + 2k = ... = T(n/2^i) + ik
The loop will terminate in our base case, i.e. array has just 1 element or T(n/2^i) = T(1).

or, n/2^i = 1

or, i = log n

Putting in original equation we have,

T(n) = T(1) + klogn = O(log n)

Q7 New objects are created on the heap, meaning all threads can access them (if they are made visible by the code). Heaps also contain static variables and functions, which are shared. Call stacks are independent in each thread, and hence the items on the stack are separate, meaning each thread manages its function calls separately from others. The call stack includes locally declared variables (like, int, float, references etc.), function parameters and return locations.

Q8 The given program has two major problems:

- The keyword volatile is missing for shared variable turn. Without it, there is no guarantee that the updates to the variable turn would be visible to other threads.
- Progress condition - if a thread doesn't execute, the other should not stall - is not met here. Threads t1 and t2 must execute in strict alternation for them to execute exclude. Further, t2 cannot execute exclude until t1 executes properexclude.

Here is a solution:

```
class Twoway {

    static volatile int turn = 0;
    static volatile boolean wantsToEnter[2] = {false, false}; private
    int id;
    Twoway(int id) { this.id = id; }
    void properexclude() {

        wantsToEnter[id] = true;
        int others_id = (id==1) ? 0:1; // Other's id turn =
        others_id;
        // Wait if other thread wants to enter and it's his turn while
        (wantsToEnter[others_id] && turn == others_id) {}
        // Execute the exclusive
        code exclude();
        // Set wantsToEnter[id] as false
        wants_to_enter[id] = false;

    }

}
```

Q9

To ensure that the list is never corrupted, we need to make sure that all the read operations are done on a consistent list and all the write operations leave the list consistent after they are done. (Also see serializability and linearizability.)

```
1 class Node <T> {
2       Node <T> next ;
3       T value ;
4       Node (Node <T> n, T v) {
5               next = n; value = v;
6       }
```

```
7 }
8 class LinkedNode <T> {
9       private Node <T> sentinel = new Node <T >( null , null );
10      Node <T> addafter (Node <T> node , T val) {
11              Node <T> newnode ;
12              /**
13              * In this case we are locking two nodes , between whom the new node has to be inserted .
14              * [A]-->[B*]----[C]--->[D]
15              * To be inserted between B and C
16              * [*] denotes the lock
17              */
18              ⇒synchronized ( node ) {
19                      newnode = new Node <T >( node .next , val);
20                      node . next = newnode ;
21              }
22              return newnode ;
23
24      }
25
26      Node <T> addfrst (T val ) {
27              return addafter ( sentinel , val );
28      }
29
30      Node <T> deleteafter (Node <T> node ) {
31              /**
32              * In this case we are locking two nodes between whom the node will be deleted .
33              * [A]-->[B*]----[C*]--->[D]
34              * To be deleted between B and D
35              * [*] denotes the lock
36              */
37              ⇒synchronized ( node ) {
38                      ⇒synchronized ( node . next ) {
39                              node . next = node . next . next ;
40                      }
41              }
42              return null ;
43      }
44 }
```

Please note that sanity tests (null checks and deleted node checks) are omitted for brevity.
One synchronized call on the node passed as the parameter for addafter and deleteafter will not be enough.Please consider following example:
A->B->C->D->E->F
If two threads concurrently call addafter(B,val) and deleteafter(A) and we only have a single synchronized call on the node passed as parameter, then both the operations will be allowed to proceed as one will lock B and other will lock A. Consider following order of execution:

Thread 1: addafter Line 20: newnode = new Node<T>(B.next, val)
//newnode points to C.

Thread 2: deleteafter Line 41: A.next = A.next.next
//A.next points to C. B is deleted

Thread 1: addafter Line 21: node.next = newnode
//B.next points to newnode
Final, state:
Main list: A->C->D->E->F
Side list: B->newnode

Add operation is lost on the main list. This will not be allowed if we lock the current node and the next node using two synchronized keywords in delete after as in the example delete after (A) will need to lock both A and B.
Please note, there might be a case where a thread is waiting on a node, synchronized(n), but n was deleted by some other thread. In that case we need to keep a flag with the nodes, indicating whether it has been deleted or not. We are not showing this in the code to keep things simple.