Problem 1)

The code for this would look like:

```
for (int i = 0; i < n; i++) {
    for (int j = i; j <= i; j++) {
        System.out.println("*");
    }
}
```

a. The basic operations performed are:

print a star, print a new line, print a blank space

b. For the specific output shown above:

- 15 stars are printed
- 5 lines are created
- 10 lines are printed

c. For a general case,

let's take $n = 6$ here:

*
**
***
****
*****
******

stars = 21
lines = 6

we notice a pattern here:

- number of stars $\frac{n \cdot (n+1)}{2}$
- number of lines n

d) Each for loop is linear, so it takes $O(n)$ time. Having nested for loops makes it $n * n = n^2$ amount of operations we need to do. Hence, nested for loop make this $O(n^2)$ time.

Problem 2)

a) The basic operations would be:

- addition of all the student scores to a single value
- division of that value by number of students to get average value

b) The worst case running time is:

- c exams and r students → c * r additions
- c exams → c divisions

c) big O running time is O(rc)

d) This is linear because rc is the input size, and the run time is linearly proportional to that.

Problem 3)

For this question, 4 arrays should be created with length 13 each.

each suit is assigned an array, and the card value number corresponds to an index

ace → index 0

2 → index 1

3 → index 2

⋮

joker → index 10

queen → index 11

king → index 12

⟶ from here, what we have to do is go through the cards and read them and put them in their proper array and index

· worst case big O run time is O(1) → linear time

Linear time because we are only reading the value and assigning it to a certain index

· basic operations are reading the value and sending it to it's array index

· since it is constant, average big O running time is same as worst case

Problem 4)

1.

a) the worst-case big O running time is $O(nm)$. This happens when there are no songs in common so you need to go through both lists entirely.

- the basic operations are:

b) the best case big O run time is:

maximum common songs → $\min(m, n)$

hence the big O would be $O(\min(m,n)^2)$

2. using merge-sort,

the worst case would be: $O(m\log m) + O(n\log n) + O(m+n) = O(m\log m + n\log n)$

the best case would be: $O(m\log m) + O(n\log n) + O(m+n) = O(m\log m + n\log n)$

these are actually the same

Problem 5)

to make it as fast algorithm as possible, we should:

- make another array and sum the distance from the start to that specific exit

for example, the distances are represented as:

$\{5, 3, 4, 2, 8, 6\}$

summing the distances to the exit give us:

$S = \{5, 8, 12, 14, 22, 28\}$

in this array, index 0 corresponds to distance from entrance to exit 1
index 1 corresponds to distance from entrance to exit 2
index 2 corresponds to distance from entrance to exit 3
index 3 corresponds to distance from entrance to exit 4
index 4 corresponds to distance from entrance to exit 5
index 5 corresponds to distance from entrance to exit 6

finding the distance from exit j to exit i is just one operation:

$S[j] - S[i]$ → this will take constant time

we notice that:

to create the S array, it takes $O(n)$ time because each operation takes $O(1)$ time and we must do it n times

once we have the S array, finding the distances is just a subtraction so it takes $O(1)$ time