1
(A)

```
input: array a of n elements and array b of m elements
output: Number of distinct elements

Algorithm: distinct(a, n, b, m)
    result <- 0

    for i <-1 to n
        if (NOT(BinarySearch(b, m, a[i])))
            result <- result + 1;

    for j <- 1 to m
        if (NOT(BinarySearch(a, n, b[j])))
            result <- result + 1;

    return result


input: array a of n elements and array b of m elements
output: Number of distinct elements in array a from b

Algorithm: distinct_in_first(a, n, b, m)
    result <- 0

    for i <- 1 to n
        if (NOT(BinarySearch(b, m, a[i])))
            result <- result + 1

    return result;
```

(B)

```
input: array a of n elements and array b of m elements
output: Number of unique elements in the intersection

Algorithm: fixed_intersection(a, n, b, m)
    intersect <- new int[1]
    count <- 0
    for i <- 1 to n
        if search(b,m,a[i])
            //intersect is of size count
            if NOT search(intersect, count, a[i] )
                if count == intersect.length
                    intersect2 <- new int[intersect.length*2]
                    for j <- 1 to intersect.length
                        intersect2[j] <- intersect[j]
                    intersect <-intersect2
                count <- count + 1
                intersect[count] <- a[i]
    return count
```

2
(a)

```
input: array a of n elements and natural number k
```

```
    output: array of elements occuring k times in array a

    Algorithm: k_times_elements(a, n, k)
        result <- new int[1]
        count <- 0

        for i <- 1 to n
            if occurences(a, n, a[i]) <= k AND occurences(result, count, a[i]) > 0
                //This is to mimic a list by increasing the size of the array
every time it is full
                if count == result.length
                    result2 <-new int[result.length+1]
                    for j <-1 to result.length
                        result2[j] <- result[j]
                    result <-result2
                count <- count + 1
                result[count] <-a[i]
        return result

    subroutine:
    input: array a of n elements and value b
    output: number of occurences of b in array a
    Algorithm: occurences(a, n, b)
        result <- 0
        for i <- 1 to n
            if a[i] == b
                result <-result + 1
        return result
```

(b)

The occurences subroutine has a running time of O(n)
The k_times_elements algorithm has a runnig time of O(n^2)
In the worse case running time: each iteration is bounded by O(2n) = O(n),
because the occurences subroutine is called twice,
and i is at most n.

3
(e)

For the multiplyConstant method, the worse case running time is O(n). This is because only the inner loop will interate at most n times and the outer loop iterates 1 time only inside the multiplyPolys method.

For the multiplyPolys method, the worse case running time is O(n^2)

for the clean method, the worse case running time is O(n^2)

For the derive method, the worse case running time is O(n)

4
(c)

The factorization method has a worse case running time of O(1). This is because there is no loop and no matter the input the running time will remain constant and does not grow.