Khuyen Le Thi Minh s5128 – ASD 3

TASK DESCRIPTION:

Find an algorithm which determines an element of an input n---element array of natural numbers, that occurs mostly.
Prove algorithm's correctness and estimate its complexity.

ALGORITHM:

Since there can be situation that more than 1 number appear with the same number of times and be the most frequent one, we will return an array of int.

```
Int[] Buckets (int [] A, int n)
{
    //if the length of an array is higher than 1 then do the calculation. if (n >
    1) then
      //initialising the maximum and minimum numbers in array to the first one in array.
      Int maxNum = A[0], minNum = A[0];
      //first loop going through the original array to find maximum and minimum numbers. . With each
      iteration value I is increasing by 1, once it reach n the loop will break. The worst case time complexity
      = O(n) for i:=0 to n---1 do
      //if A[i] is smaller than the current smallest number then we assign A[i] to
          minNum. if(A[i] < minNum) then  minNum = A[i];
        //if A[i] is bigger than the current biggest number then we assign A[i] to
          maxNum. else if (A[i] > maxNum) then minNum = A[i];
          fi; od;


      //initialising the array where we will fill with the frequency number, we find the length of the array
      by taking out from maximum number minimum number and increase the result by 1.
      Int diff = maxNum – minNum + 1;
      Int[] countedArray [diff];
      //second loop going through the original array to count how many time each number repeats. With
      each iteration value I is increasing by 1, once it reach n the loop will break. The worst case time
      complexity = O(n) for i:=0 to n---1 do
      //finding the position of number in counting array by taking out from the number the minNum. Int
          numberOfB = A[i] --- minNum;
        //increasing the count value in the finded position by 1. countedArray[numberOfB] =
          countedArray[numberOfB] + 1;
      od;


      //initialising the highest count value to the first element of the counting array.
      Int mostOccured = countedArray[0];
      //first loop through count array to find the highest value. With each iteration value I is increasing by
      1, once it reach diff the loop will break. The worst case time complexity = O(k) where k is the
      different between max and min, depend on the input values the time complexity will be different. If the
      array is uniform then k will be rather small, in the case when the different between max and min is
      big then the k will be very big. for i:=0 to diff---1 do
      //if countedArray[i] is smaller than the current highest count then we assign
      countedArray[i] to mostOccured. if(countedArray[i] > mostOccured) then  mostOccured =
      countedArray [i];  fi; od;


      //initialising the size for the array with elements that appears mostly.
      Int counter = 0;
      //second loop through count array to find how many time the highest count appears. With each
      iteration value i is increasing by 1, once it reach diff the loop will break. The worst case time
      complexity = O(k) where k is the different between max and min, depend on the input values the time
      complexity will be different. If the array is uniform then k will be rather small, in the case when the
      different between max and min is big then the k will be very big. for i:=0 to diff---1 do
      //if countedArray[i] is equals to the highest count, we increase the size of the array by 1.
      if(countedArray[i]== mostOccured) then  counter = counter + 1;
          fi; od;


      //initialising the result array fint number of elements equals to found earlier counter.
      Int[] resultArray [counter];
```

```
        //initialising the number of positioning the element in result array to 0.
        Int k = 0;
        //third loop through count array to find the positions of highest number of appearance. With each
        iteration value i is increasing by 1, once it reach diff the loop will break. The worst case time
        complexity = O(k) where k is the different between max and min, depend on the input values the time
        complexity will be different. If the array is uniform then k will be rather small, in the case when the
        different between max and min is big then the k will be very big. for i:=0 to diff---1 do
            //comparing if countedArray[i] is equals to number most occur elements appear.. if(countedArray[i]
            == mostOccured) then
                //we change the number back to its original by adding the minimum value from original array
                to the position value.
                resultArray[k] = i + minNum;
                //we increase the pointer in result array by 1 until it reaches counter to break. if(k <
                counter) then
                    k = k + 1;
                fi;
            fi; od;


        //returning the result. return
        resultArray;
    else then
        //if n <=1, then return original array return
        A;
    fi;
}
```

Justification of an algorithm's correctness:

The specification of the algorithm Buckets becomes a pair <WP, WK>, where WP = (A is a array of natural numbers, n is a natural number and it's the size of the A) and WK = (the numbers that appeared mostly in the array).

Stop condition of the algorithm – We have 5 loops in the algorithm, each of them start at 0 and 2 loops that go through original array with size n will stop once the value reach n. Another 3 loops go through the finite array of natural number with size diff and will stop once the value reach diff. There is 1 if condition that starts from 0 and will break once it reach the number counter which is a natural number. Thus the algorithm stops for any data satisfying the initial condition WP.

Partial correctness of the algorithm – for the initial condition WP = (A is a array of natural numbers, n is a natural number and it's the size of the A) and WK = (the numbers that appeared mostly in the array:

With the Array A = {12, 14, 13, 16, 18, 20, 18, 17, 19, 18, 19, 18, 15, 19, 19, 11, 159, 159, 159, 159} n= 20

```
Int[] Buckets (int [] A, int n)
{ if (n > 1) then
        //n>1, we start the if condition.
        Int maxNum = A[0], minNum = A[0];
        //maxNum = 12, minNum = 12.
        for i:=0 to n---1 do if(A[i] < minNum)
            then  minNum = A[i];
            else if (A[i] > maxNum) then minNum = A[i];
            fi;
        //when i = 0, A[i] = 12, since 12 = 12 then nothing happen;  //when i = 1, A[i] =
        14, since 14 > maxNum then maxNum = 14;
        //when i = 2, A[i] = 13, since 13 > minNum and 13 < maxNum then nothing
        happen;
        //when i = 3, A[i] = 16, since 16 > maxNum then maxNum =16;
        //when i = 4, A[i] = 18, since 18 > maxNum then maxNum =18;
        //when i = 5, A[i] = 20, since 20 > maxNum then maxNum =20;
        //when i = 6, A[i] = 18, since 18 > minNum and 18 < maxNum then nothing
        happen;
```

```
//when i = 7, A[i] = 17, since 17 > minNum and 17 < maxNum then nothing
happen;
//when i = 8, A[i] = 19, since 19 > minNum and 19 < maxNum then nothing
happen;
//when i = 9, A[i] = 18, since 18 > minNum and 18 < maxNum then nothing
happen;
//when i = 10, A[i] = 19, since 19 > minNum and 19 < maxNum then nothing
happen;
//when i = 11, A[i] = 18, since 18 > minNum and 18 < maxNum then nothing
happen;
//when i = 12, A[i] = 15, since 15 > minNum and 15 < maxNum then nothing
happen;
//when i = 13, A[i] = 19, since 19 > minNum and 19 < maxNum then nothing
happen;
//when i = 14, A[i] = 19, since 19 > minNum and 19 < maxNum then nothing
happen;
//when i = 15, A[i] = 11, since 11 < minNum then minNum = 11;
//when i = 16, A[i] = 159, since 159 > maxNum then maxNum = 159; //when i =
17, A[i] = 159, since 159 = maxNum then nothing happen;
//when i = 18, A[i] = 159, since 159 = maxNum then nothing happen;
//when i = 19, A[i] = 159, since 159 = maxNum then nothing happen;
od;

Int diff = maxNum – minNum + 1;
//159 – 11 + 1 = 149;

Int[] countedArray [diff]; //Int[]
countedArray[149];
for i:=0 to n---1 do
    Int numberOfB = A[i] --- minNum;
    countedArray[numberOfB] = countedArray[numberOfB] + 1;
    //when i = 0, A[i] = 12, numberOfB = 12 – 11 = 1, countedArray[1] = 1;
    //when i = 1, A[i] = 14, numberOfB = 14 – 11 = 3, countedArray[3] = 1;
    //when i = 2, A[i] = 13, numberOfB = 13 – 11 = 2, countedArray[2] = 1;
    //when i = 3, A[i] = 16, numberOfB = 16 – 11 = 5, countedArray[5] = 1;
    //when i = 4, A[i] = 18, numberOfB = 18 – 11 = 7, countedArray[7] = 1;
    //when i = 5, A[i] = 20, numberOfB = 20 – 11 = 9, countedArray[9] = 1;
    //when i = 6, A[i] = 18, numberOfB = 18 – 11 = 7, countedArray[7] = 2;
    //when i = 7, A[i] = 17, numberOfB = 17 – 11 = 6, countedArray[6] = 1;
    //when i = 8, A[i] = 19, numberOfB = 19 – 11 = 8, countedArray[8] = 1;
    //when i = 9, A[i] = 18, numberOfB = 18 – 11 = 7, countedArray[7] = 3;
    //when i = 10, A[i] = 19, numberOfB = 19 – 11 = 8, countedArray[8] = 2;
    //when i = 11, A[i] = 18, numberOfB = 18 – 11 = 7, countedArray[7] = 4;
    //when i = 12, A[i] = 15, numberOfB = 15 – 11 = 4, countedArray[4] = 1;
    //when i = 13, A[i] = 19, numberOfB = 19 – 11 = 8, countedArray[8] = 3;
    //when i = 14, A[i] = 19, numberOfB = 19 – 11 = 8, countedArray[8] = 4;
    //when i = 15, A[i] = 11, numberOfB = 11 – 11 = 0, countedArray[0] = 1;
    //when i = 16, A[i] = 159, numberOfB = 159 – 11 = 148, countedArray[148] =
    1;
    //when i = 17, A[i] = 159, numberOfB = 159 – 11 = 148, countedArray[148] =
    2;
    //when i = 18, A[i] = 159, numberOfB = 159 – 11 = 148, countedArray[148] =
    3;
    //when i = 19, A[i] = 159, numberOfB = 159 – 11 = 148, countedArray[148] =
4; od;

Int mostOccured = countedArray[0];
//mostOccured = 1;
for i:=0 to diff---1 do
    if(countedArray[i] > mostOccured) then
    mostOccured = countedArray [i];  fi;
```

```
        //after iteration through countedArray, mostOccured = 4;
        od;

        Int counter = 0; //counter =
        0;
        for i:=0 to diff---1 do.
            if(countedArray[i]== mostOccured) then
            counter = counter + 1;  fi; //since
            number 4 appears in countedArray[7],
            countedArray[8] and countedArray[148]
            then counter = 3 ; od;

        Int[] resultArray [counter];
        //resultArray[3];
        Int k = 0; //k =
        0;
        for i:=0 to diff---1 do
            if(countedArray[i] == mostOccured) then
                resultArray[k] = i + minNum; if(k < counter)
                then
                    k = k + 1;
                fi;
            fi;
            // countedArray[7] = 4, then resultArray[0] = 7 + 11 = 18, k = 1;
            // countedArray[8] = 4, then resultArray[1] = 8 + 11 = 19, k = 2;
            // countedArray[148] = 4, then resultArray[2] = 148 + 11 = 159, k =
        3;  od;

        Return resultArray;
        // the returned result are: 18, 19, 159 which appeared 4
    times; else then return A;
    fi;
}
```

With the Array B = {12} n=
1

```
Int[] Buckets (int [] A, int n)
{ if (n > 1) then
    // n = 1 so we jump to the else condition;
          ...
    else then return A;
        // we return 12 which appeared once;
    fi;
}
```

Hence the result returning the elements that appear most frequently in the input array is true.

Because the Buckets algorithm is partially correct in terms of specification <WP,WK> and we have shown the stop condition for this algorithm, it is also absolutely correct with respect to the considered specification.

Estimation of algorithm's complexity:

Time complexity – assuming that the dominant operation in the Buckets is the natural numbers arithmetic operations and comparison, then $T(n) = n + n + k + k + k = 2n + 3k = n + k \Rightarrow T(n) = n$

Space complexity – the Buckets algorithm uses 7 auxiliary variables that are independently of the value of the argument n, 1 count array with length of k (depend on the input value the k will be determined, it can be even larger than n) and 1 consider small result array, therefore S should be $S(n) = k + 1 = k \Rightarrow S(n) = k$.