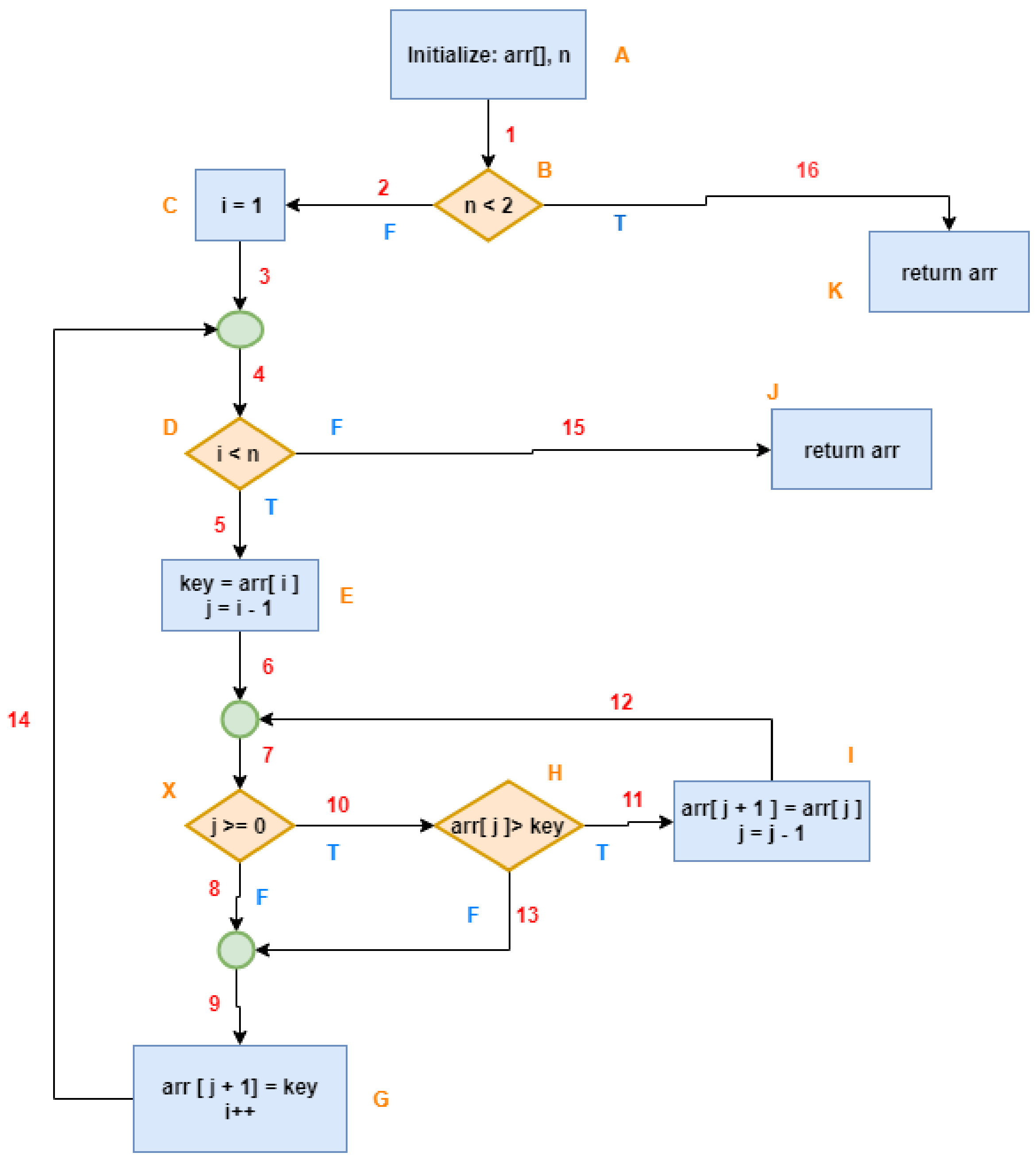
CENG 437 HOMEWORK 2

1) **Draw a CFG for `insertionSort ()`. Tag each statement with unique letters and tag each branch with unique numbers.**



2) From the CFG, identify a set of entry–exit paths to satisfy the complete statement coverage criterion.

SCPath1 = A –> 1 –> B(T) –> 16 -> K

SCPath2 = A -> 1 -> B(F) -> 2 -> C -> 3 -> 4 -> D(T) -> 5 -> E -> 6 -> 7 -> X(T) -> 10 -> H(T) -> 11 -> I -> 12 -> 7 -> X(F) -> 8 -> 9 -> G -> 4 -> D(F) -> 15 -> J

We have visited all nodes from A to K, and complete statement coverage is satisfied.

3) Identify additional paths, if necessary, to satisfy the complete branch coverage criterion.

Addition to SCPath1 and SCPath2, to satisfy the complete branch coverage, we can add the following path:

BCPath3 = A -> 1 -> B(F) -> 2 -> C -> 3 -> 4 -> D(T) -> 5 -> E -> 6 -> 7 -> X(T) -> 10 -> H(F) -> 13 -> 9 -> 14 -> 4 -> D(F) -> 15 -> J

We have visited all edges from 1 to 16, and complete branch coverage is satisfied.

4) For each path identified above, derive their path predicate expressions.

- **The path SCPath1 indicates that node B is the decision node. Table1**

Path Predicate for SCPath1	Path Predicate Expression for SCPath1
n < 2 = True	n < 2 = True

- **The path SCPath2 indicates that nodes B, D, X, H are decision nodes. Table2**

	Path Predicate for SCPath1	Path Predicate Expression for SCPath1
1)	n < 2 = False	n < 2 = False
2)	i < n = True	1 < n = True
3)	j >= 0 = True	0 >= 0 = True
4)	arr[j] > key = True	arr[0] > arr[1] = True
5)	i < n = False	n < 2 = False

- **The path BCPath1 indicates that nodes B, D, X, H are decision nodes. Table3**

	Path Predicate for BCPath1	Path Predicate Expression for BCPath1
1)	n < 2 = False	n < 2 = False
2)	i < n = True	1 < n = True
3)	j >= 0 = True	0 >= 0 = True
4)	arr[j] > key = False	arr[0] > arr[1] = False
5)	i < n = False	n < 2 = False

5) Solve the path predicate expressions to generate test input and compute the corresponding expected outcomes.

- **For table 1**

Input data:	Expected Output:
arr = [15]	[15]
n = 1	

- **For Table 2**

--Constraint 3 (j>=0) is always satisfied.

--Constraint 1, 2 and 5 must be solved togetger:

1) n < 2 = false,
2) 1 < n = true,
3) 2 < n = false

We can conclude that n should 2 according to SCPath2.

Input data:	Expected Output:
arr = [100, 30]	[30, 100]
n = 2	

- **For Table 3**

--Constraint 3 (j>=0) is always satisfied.

--Constraint 1, 2 and 5 must be solved togetger:

1) n < 2 = false,
2) 1 < n = true,
3) 2 < n = false

We can conclude that n should 2 according to BCPath1.

Input data:	Expected Output:
arr = [65, 110]	[65, 110]
n = 2	

6) Are all the selected paths feasible? If not, select and show that a path is infeasible, if it exists.

Yes, SCPath1, SCPath2 and BCPath1 are all feasible.

If we can add a path like:

A -> 1 -> B(F) –> 2 -> C -> 3 -> 4 -> D(F) -> 15 -> J, that path is infeasible.

Because path predicate is:

$n < 2$ = False
$i < n$ = False

And path predicate expression is:

$n < 2$ = False
$1 < n$ = False

Which yields with the unsolved equation $1 < n < 2$, so the path is infeasible.

7) Can you introduce faults in the routine so that these go undetected by your test cases designed for complete branch coverage?

If we replace the correct statement:

```
while( j >= 0  && arr[j] > key){
            ...
}
```

with the faulty statement:

```
while( j >= 0  && arr[j] >= key){
            ...

}
```

In first case, arr[j] > key does not changes the order of the same elements

After faulty statement(arr[j] >= key), that method changes the equal elements' order, and test cases cannot detect that error.

8) Suggest a general way to detect the kinds of faults introduced in the previous step.

The general way can be understanding related code correctly. We can understand that fault with testing step count of the algorithm. Faulty statement cause that algorithm is completed in more steps than expected and we can catch that fault if we control it.

Briefly, if we understood the code correctly, we can find a way to detect these kind of semantic errors, however if we do not understand the code exactly, to detect these faults become very hard.