Search-Based Software Engineering

Search-based Testing

Gordon Fraser
Lehrstuhl für Software Engineering II

Alternating Variable Method

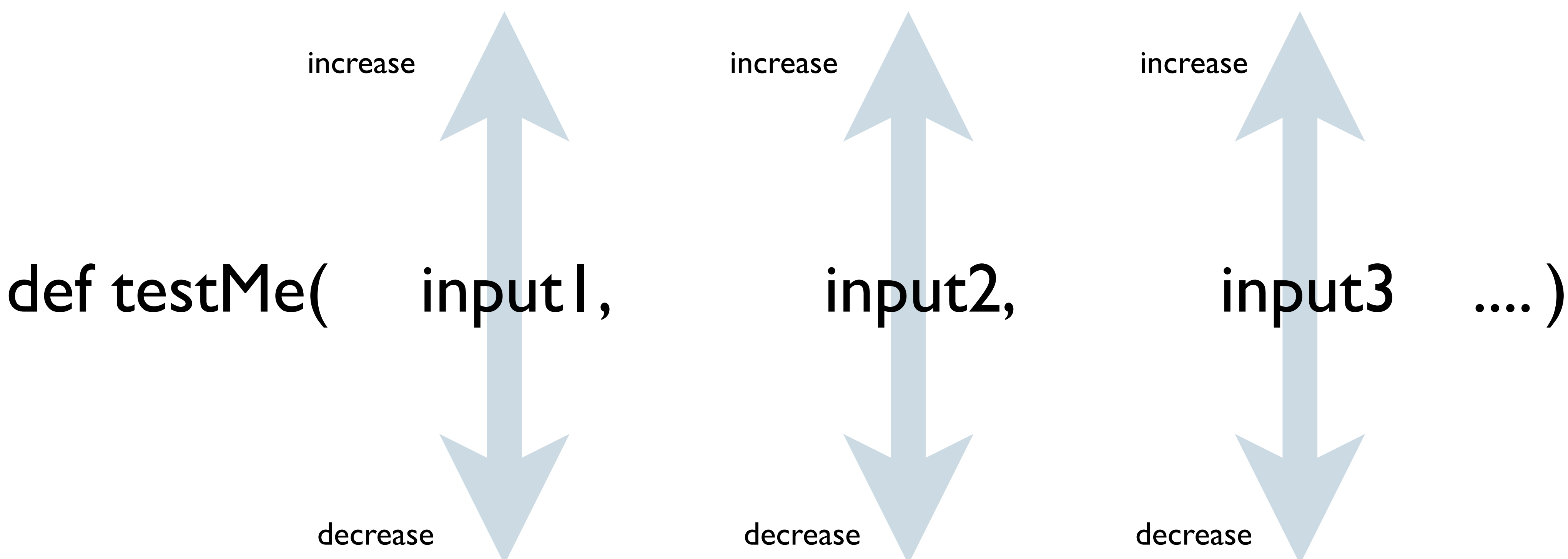
Alternating Variable Method

```
def testMe(x, y, z):
    if x * z == 2 * (y + 1):
        return True
    else:
        return False
```

Example: Hill climbing applied to the testMe function

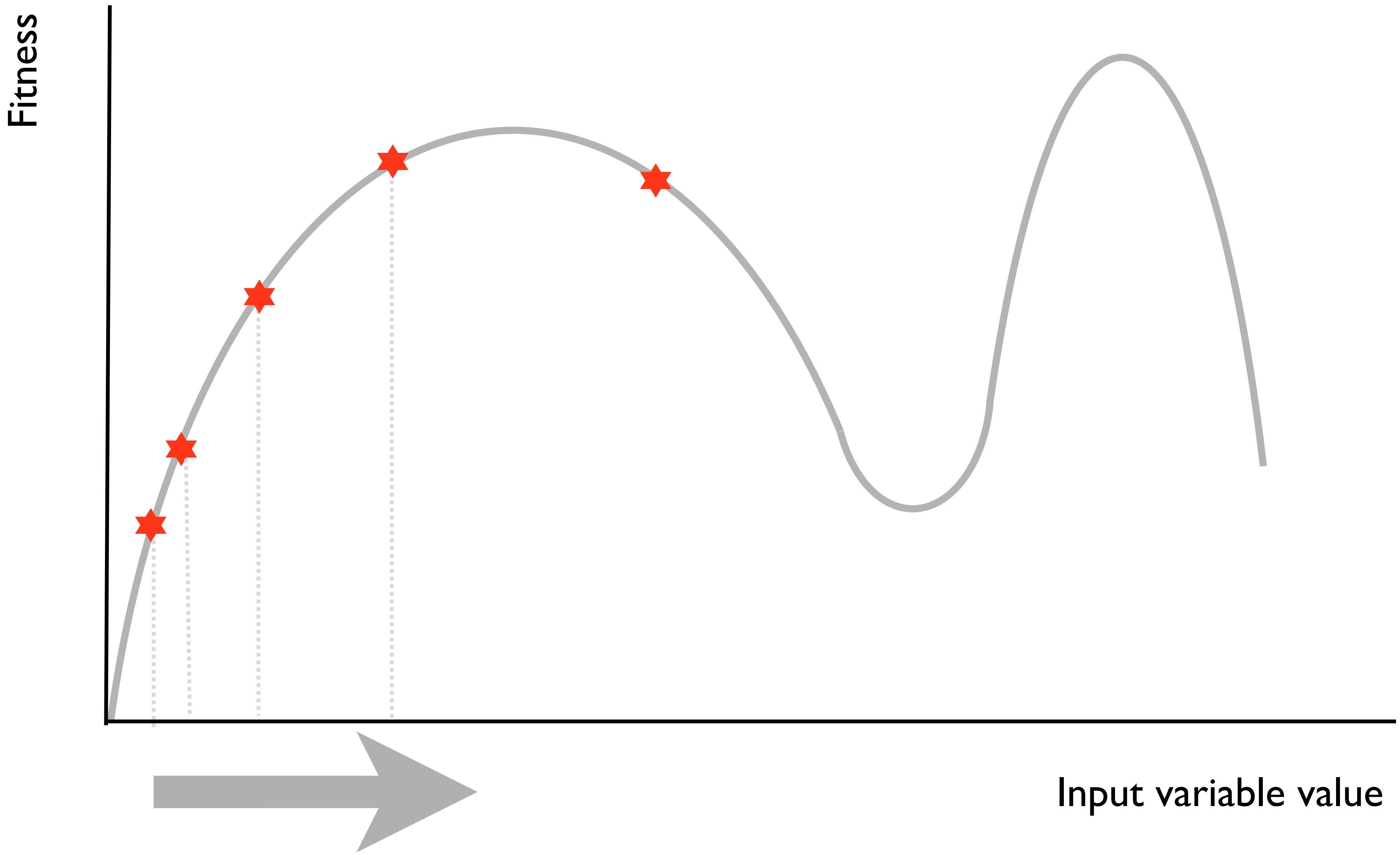
Alternating Variable Method

‘Probe’ moves



Alternating Variable Method

Accelerated hill climb



Alternating Variable Method

1. Randomly generate start point

   a=10, b=20, c=30

2. 'Probe' moves on a

   a=9, b=20, c=30

   a=11, b=20, c=30

   no effect

3. 'Probe' moves on b

   a=10, b=19, c=30

   improved fitness

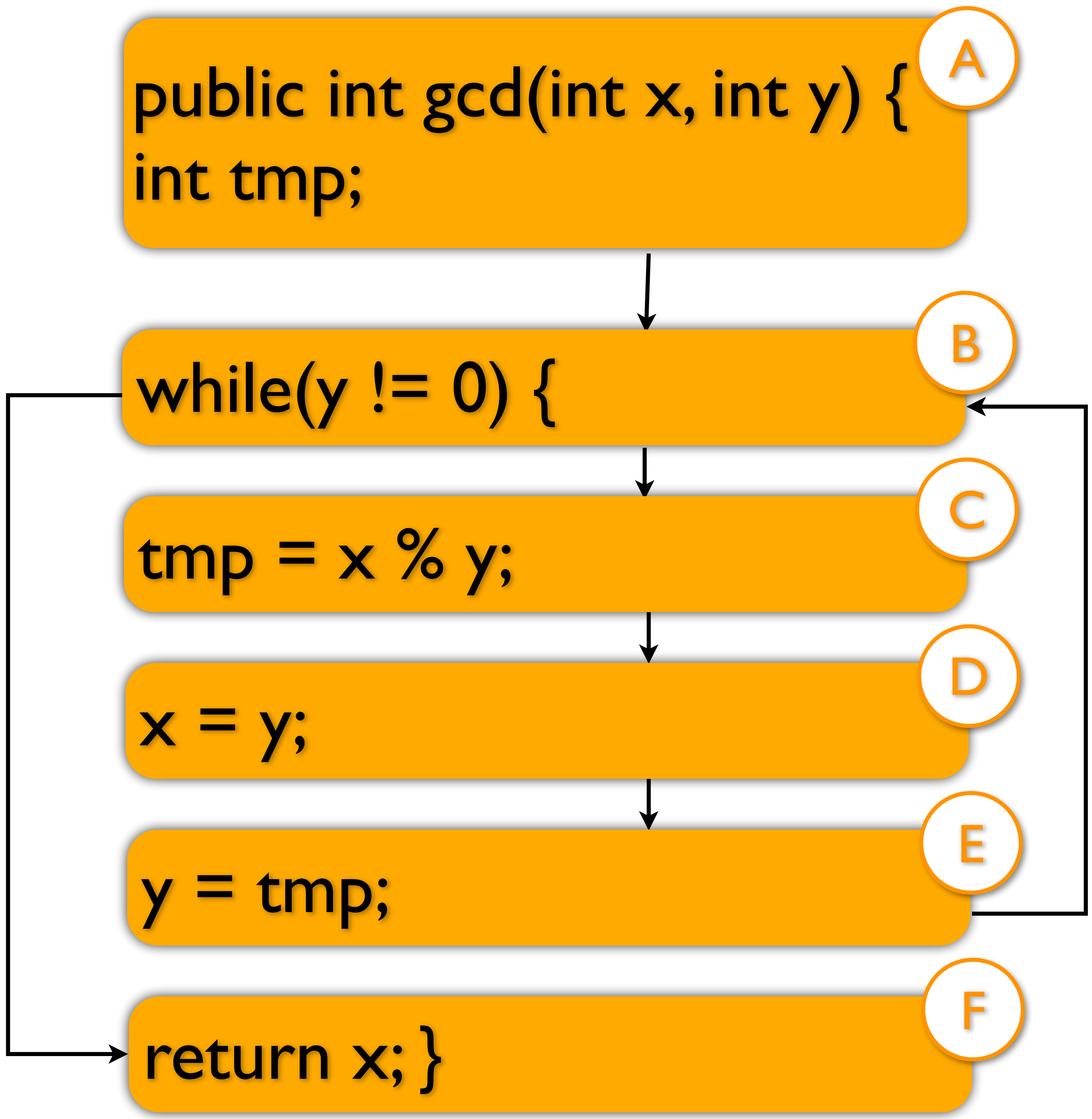
4. Accelerated moves in direction of improvement

```
void example(int a, int b, ...) {

    if (a == 0) {
        ...
    }

    if (b == 0) {
        // target
    }

    ...
}
```

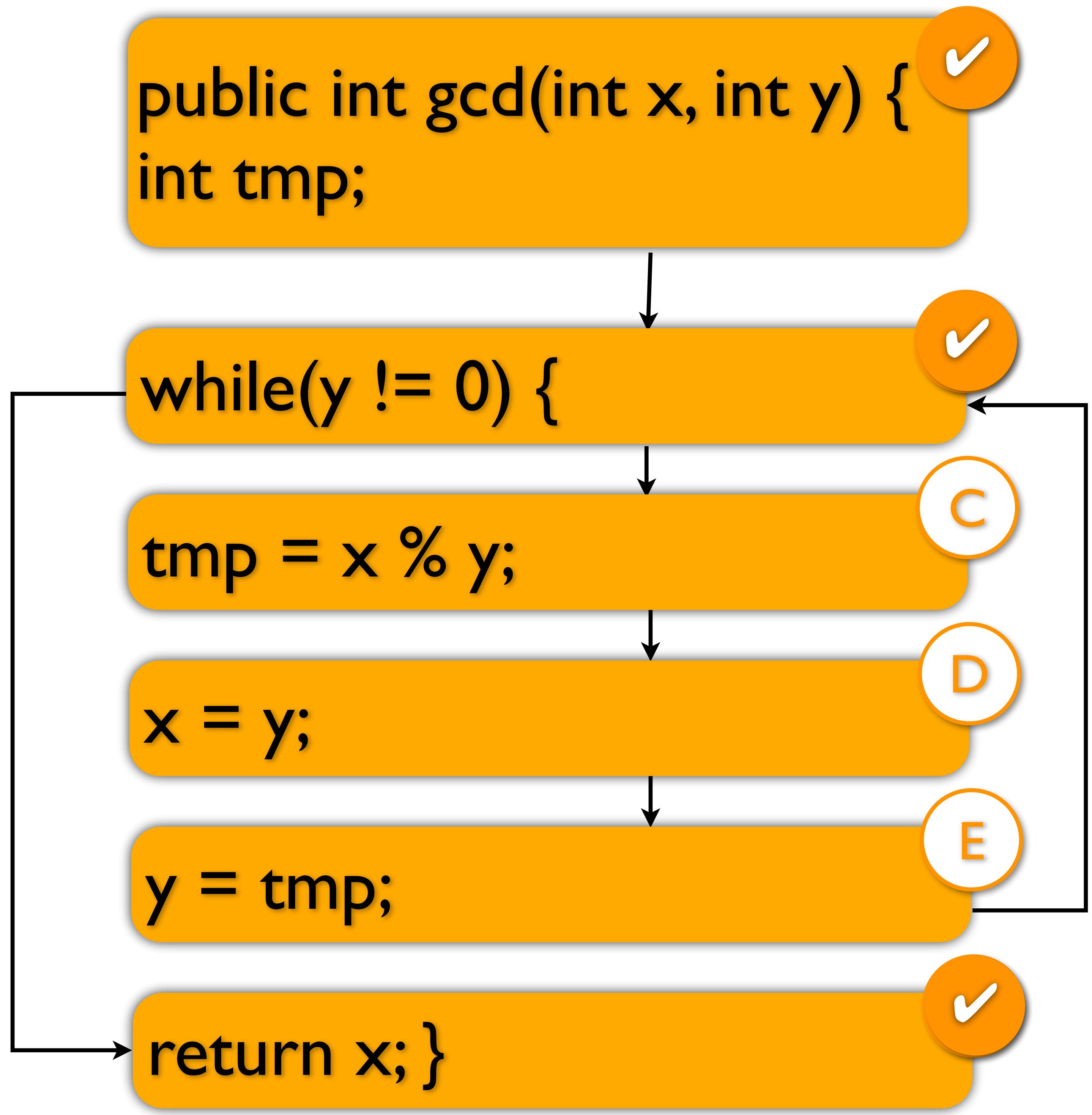
```python
def testMe(x, y, z):
    if x * z == 2 * (y + 1):
        return True
    else:
        return False
```

Example: AVM applied to the testMe function

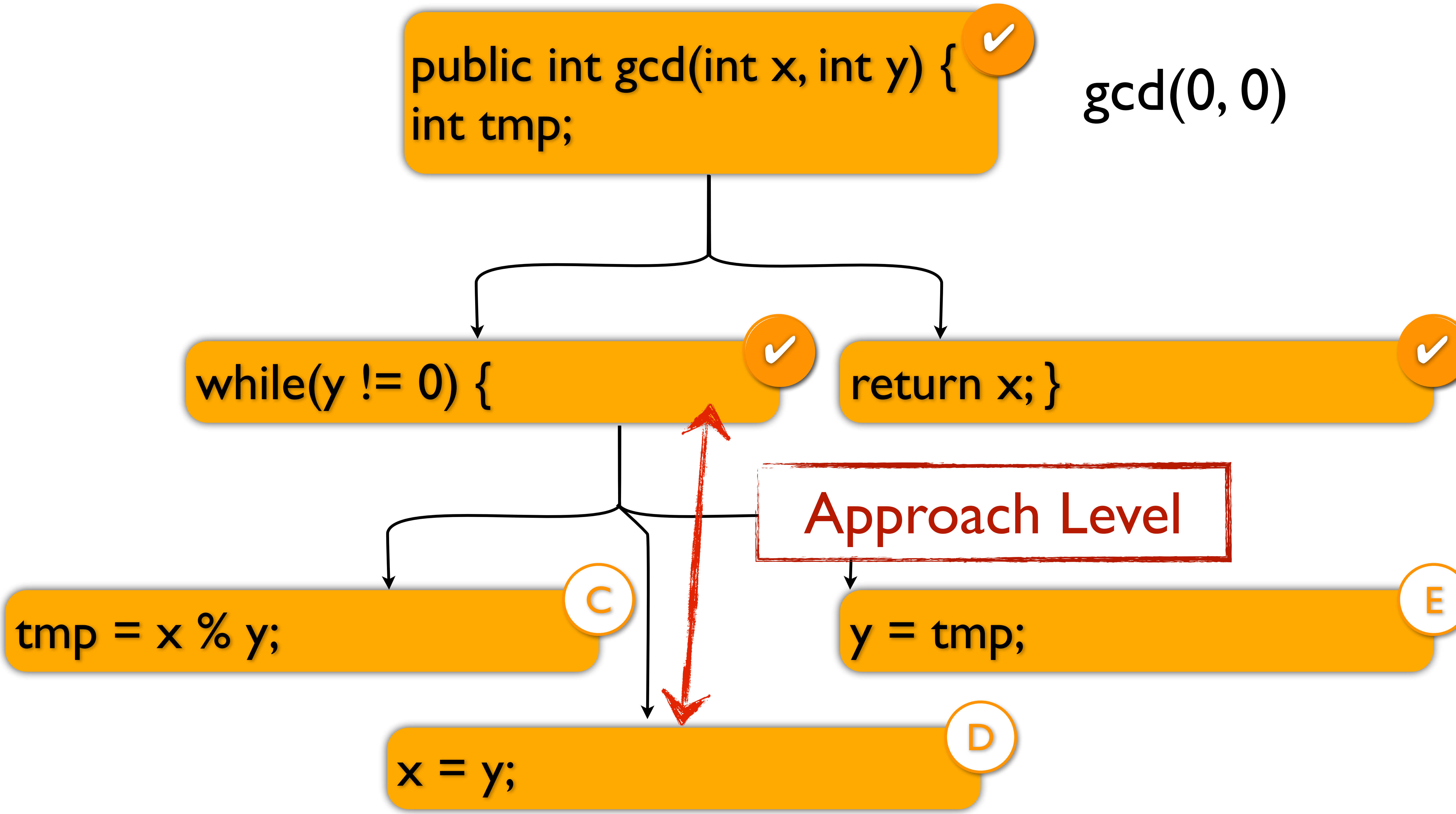
```
public int gcd(int x, int y) {
  int tmp;
  while (y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }
  return x;
}
```


public int gcd(int x, int y) {
int tmp;
A
while(y != 0) {
B
tmp = x % y;
C
x = y;
D
y = tmp;
E
return x; }
F

gcd(0, 0)

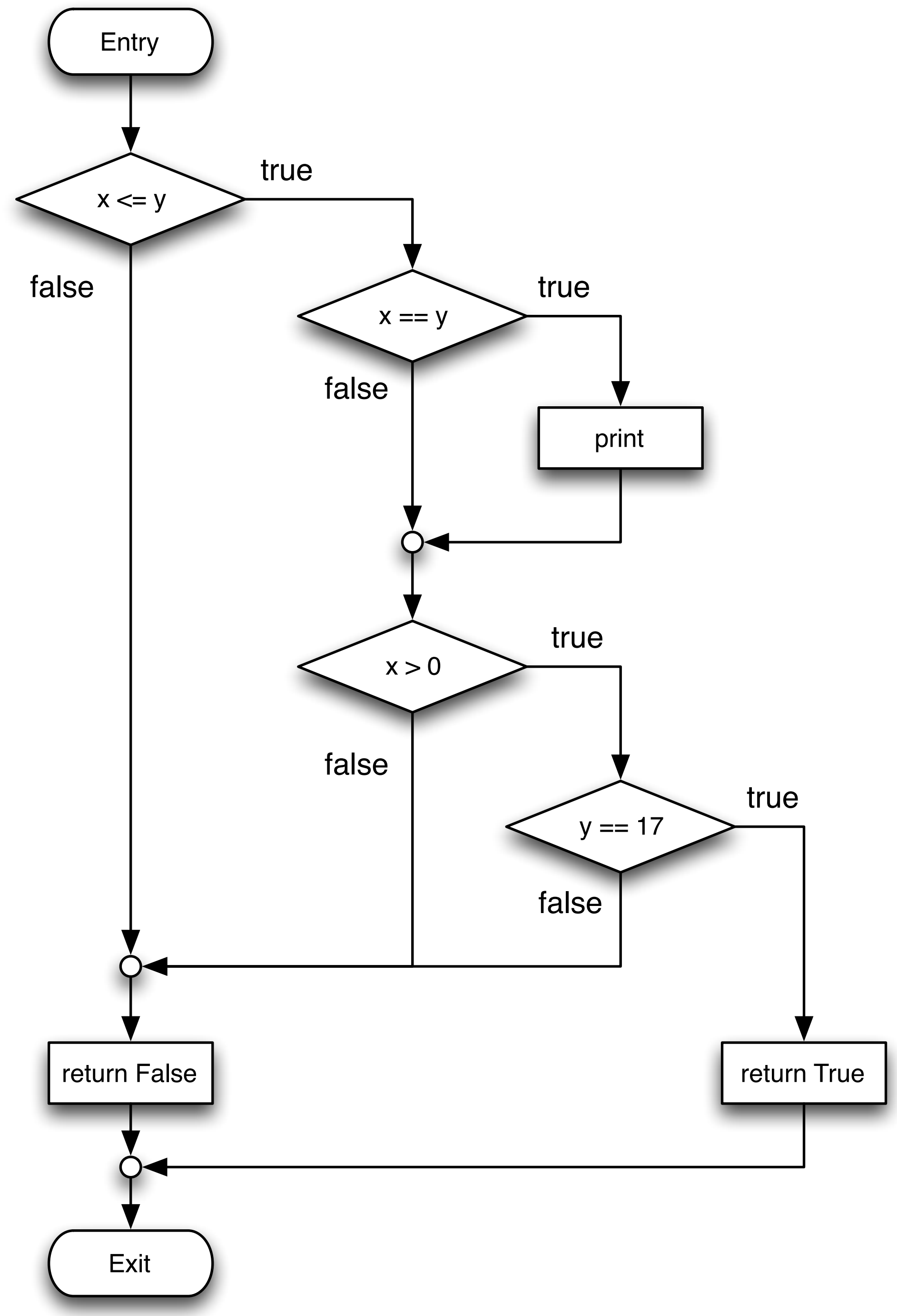


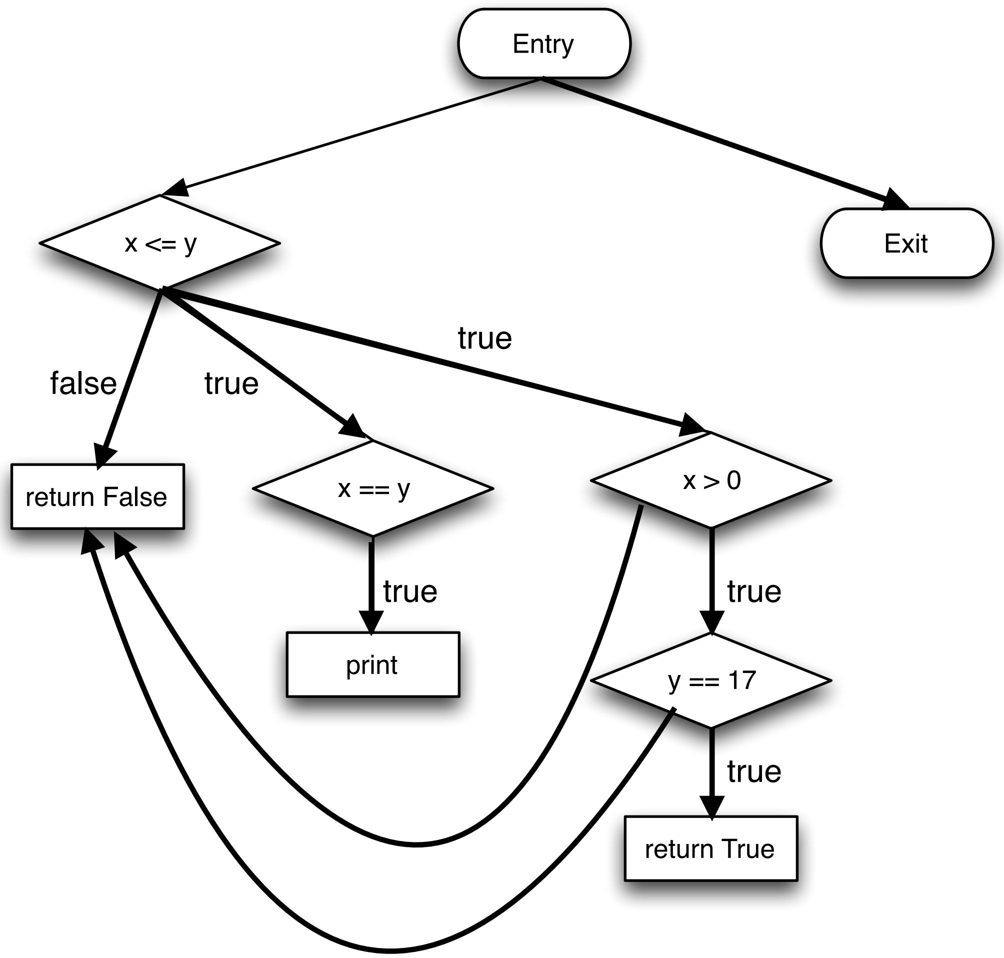
public int gcd(int x, int y) {
int tmp;
while(y != 0) {
tmp = x % y;
C
x = y;
D
y = tmp;
E
return x; }

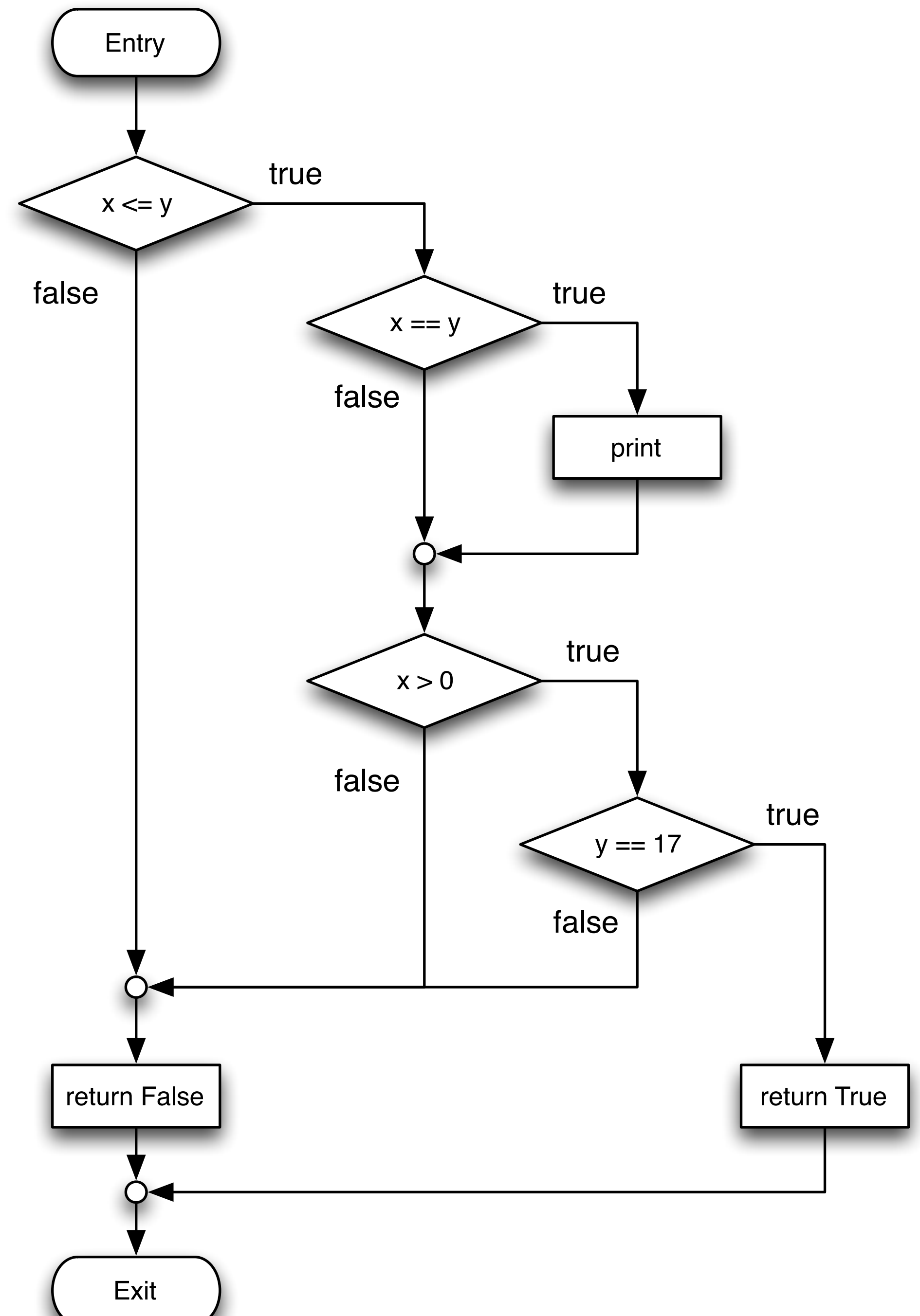

public int gcd(int x, int y) {
int tmp;
gcd(0, 0)
while(y != 0) {
return x; }
Approach Level
tmp = x % y;
C
y = tmp;
E
x = y;
D

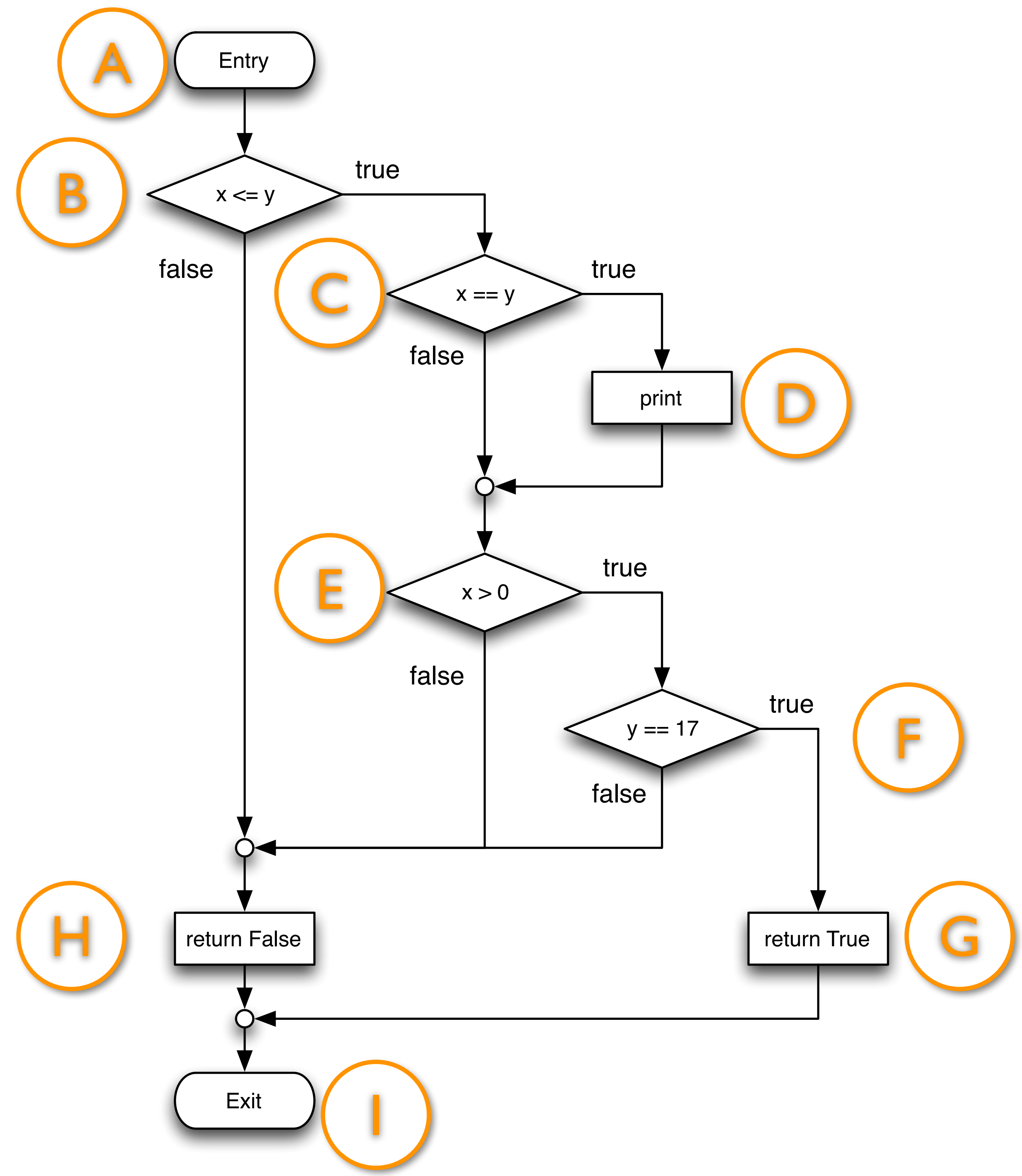
```python
def testMe(x, y):
    if x <= y:
        if x == y:
            print("Some output")
        if x > 0:
            if y == 17:
                # Target Branch
                return True
    return False
```

```python
def testMe(x, y):
    if x <= y:
        if x == y:
            print("Some output")
        if x > 0:
            if y == 17:
                # Target Branch
                return True
    return False
```



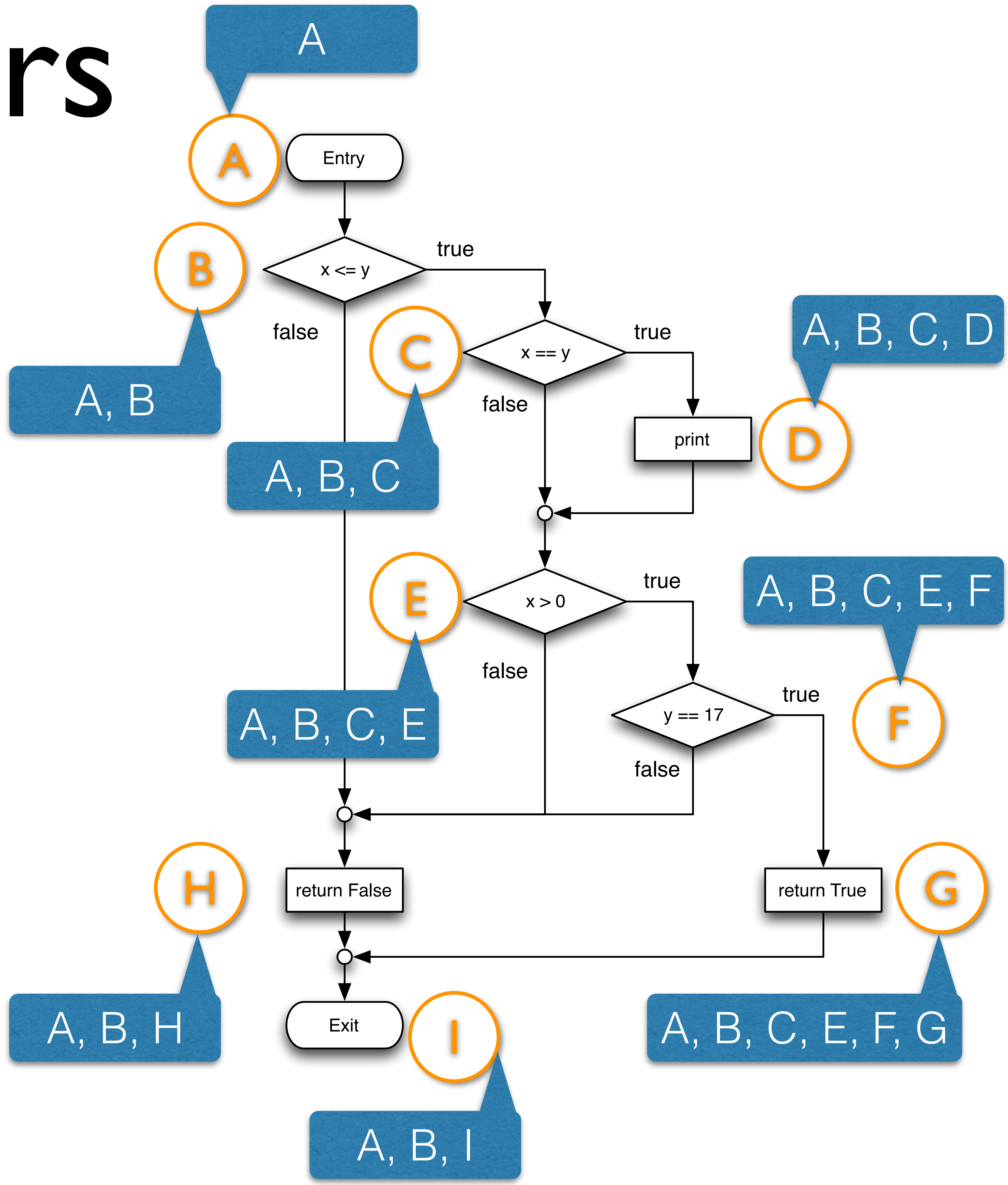

Entry
Exit
x <= y
false
true
true
return False
x == y
x > 0
true
true
print
y == 17
true
return True


Entry
x <= y
true
false
x == y
true
false
print
x > 0
true
false
y == 17
true
false
return False
return True
Exit


A
Entry
B
x <= y
true
false
C
x == y
true
false
print
D
E
x > 0
true
false
y == 17
true
false
F
H
return False
return True
G
Exit
I

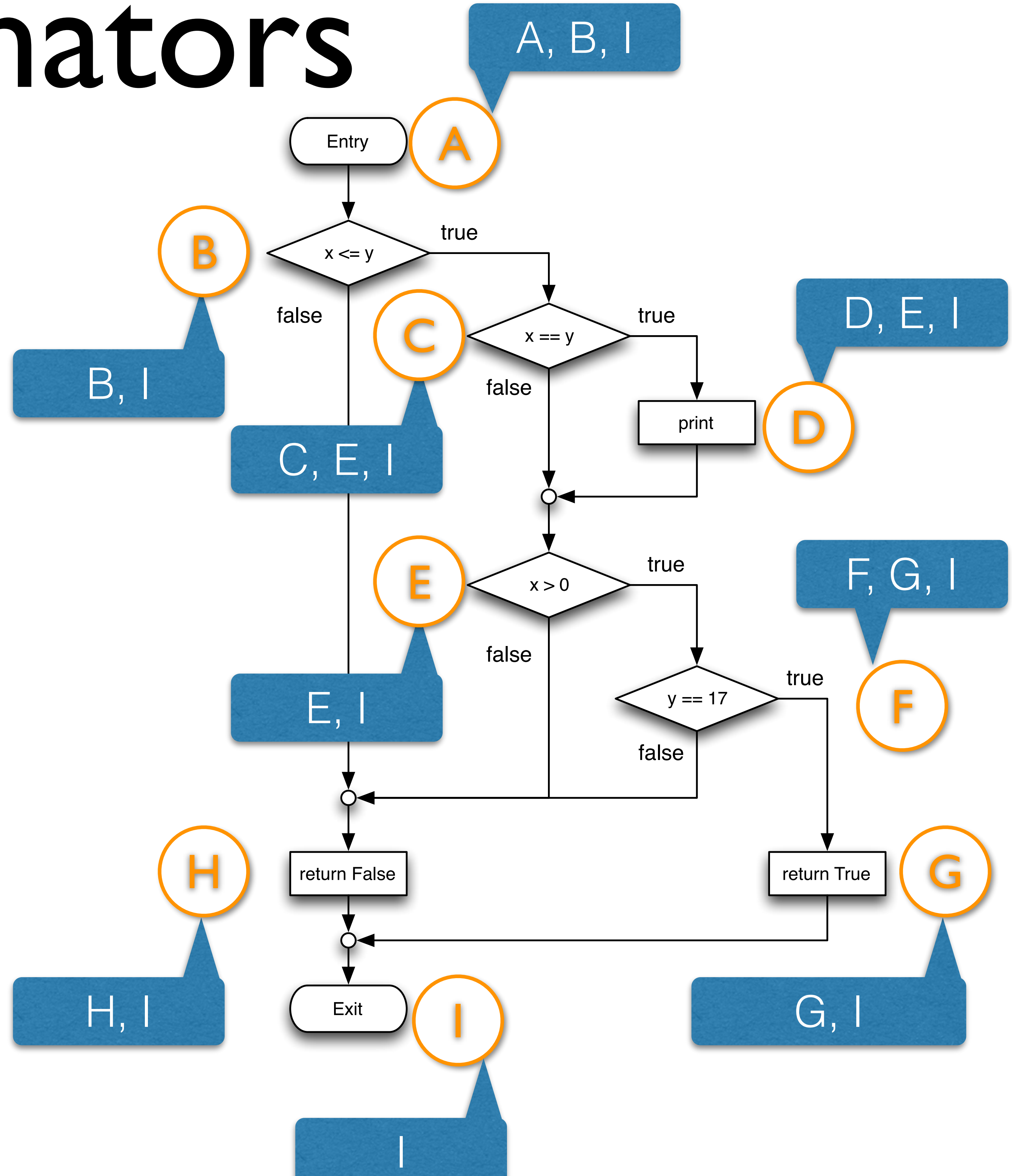
Dominators

- Node A dominates B if every path to B goes through A



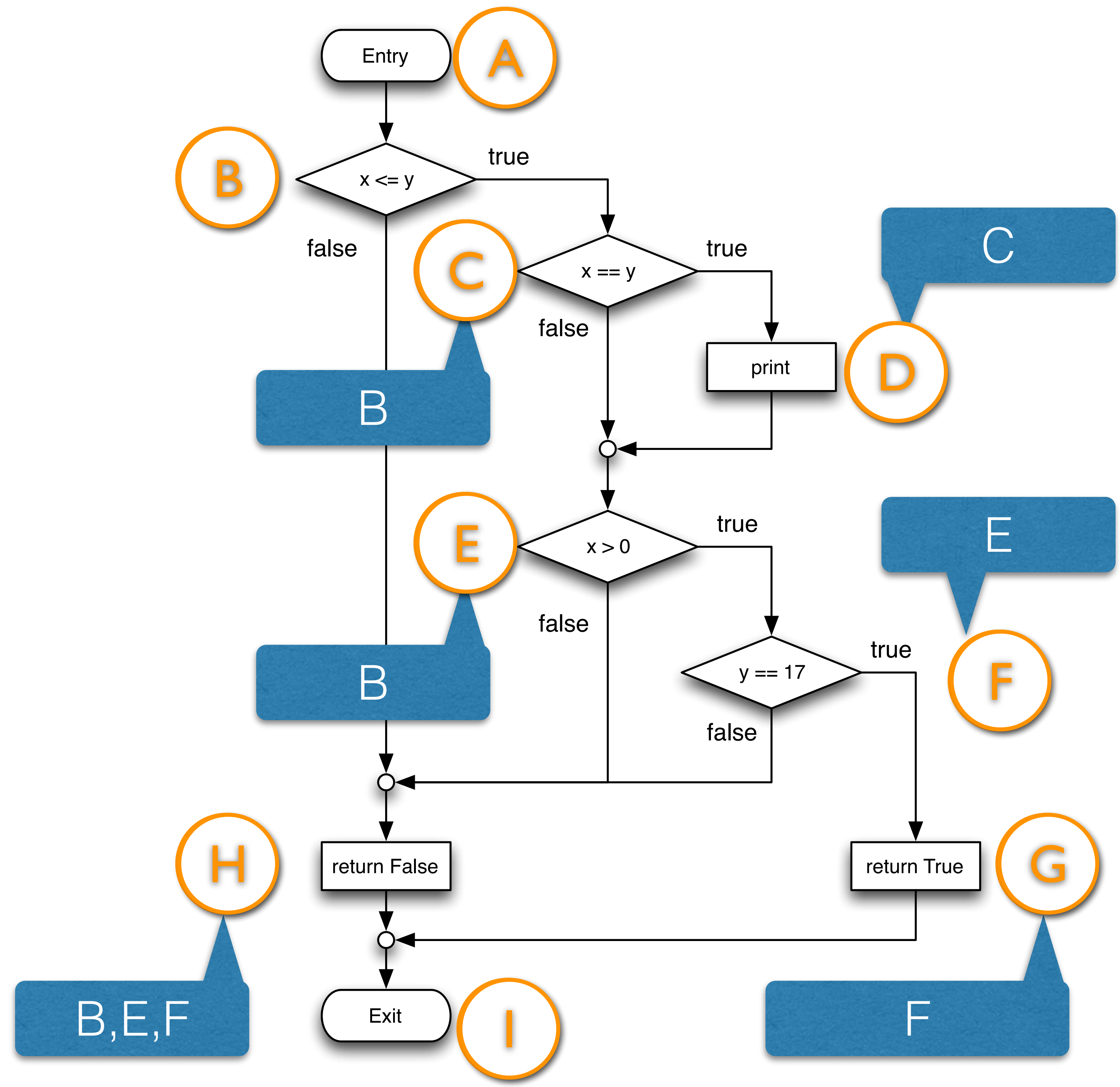
Post-Dominators

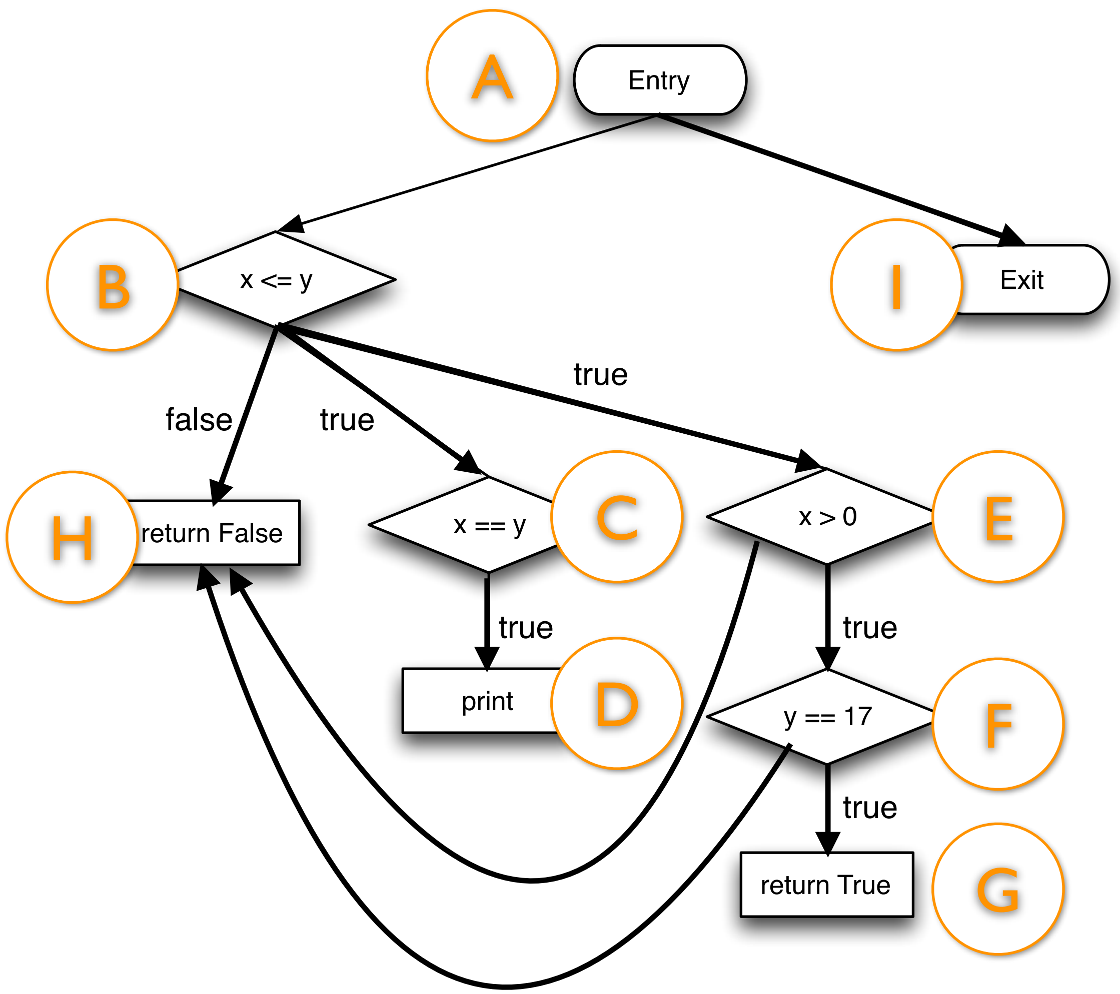
- Dominators viewed in reverse (paths from exit node)

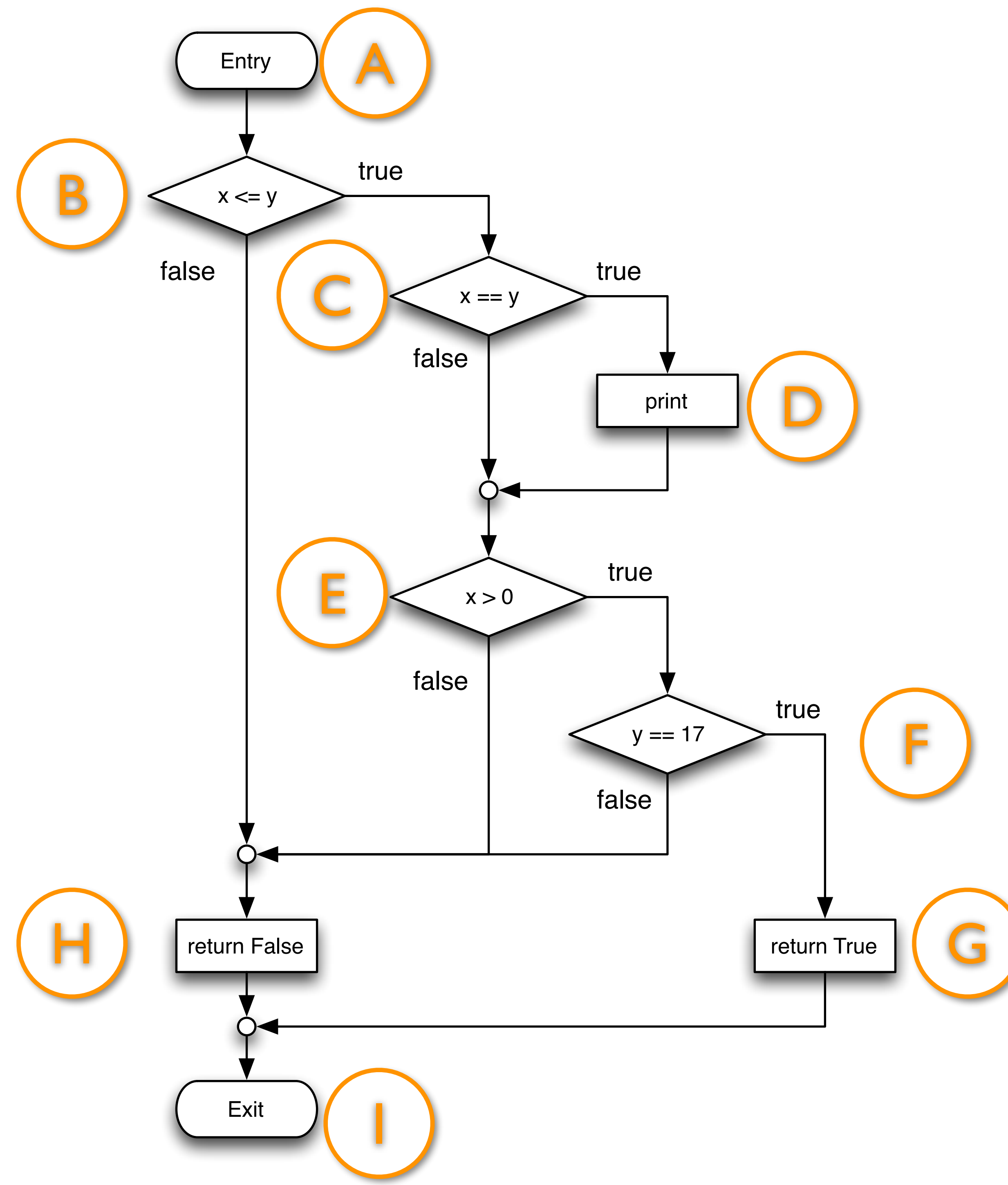


Control Dependence

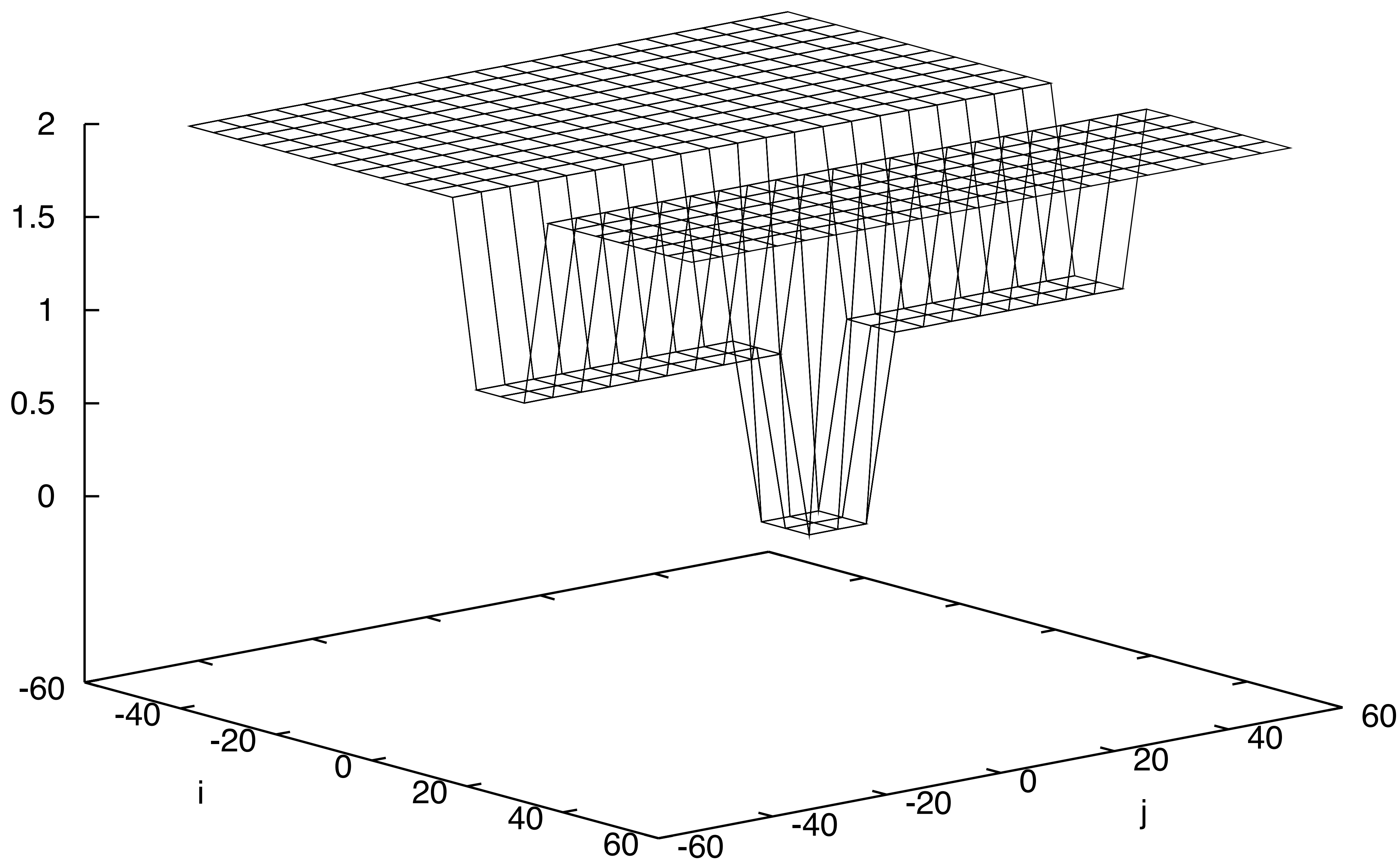
- A is control dependent on B if:
- B has at least two successors in the CFG
- B dominates A
- B is not post-dominated by A
- There is a successor of B that is post-dominated by A



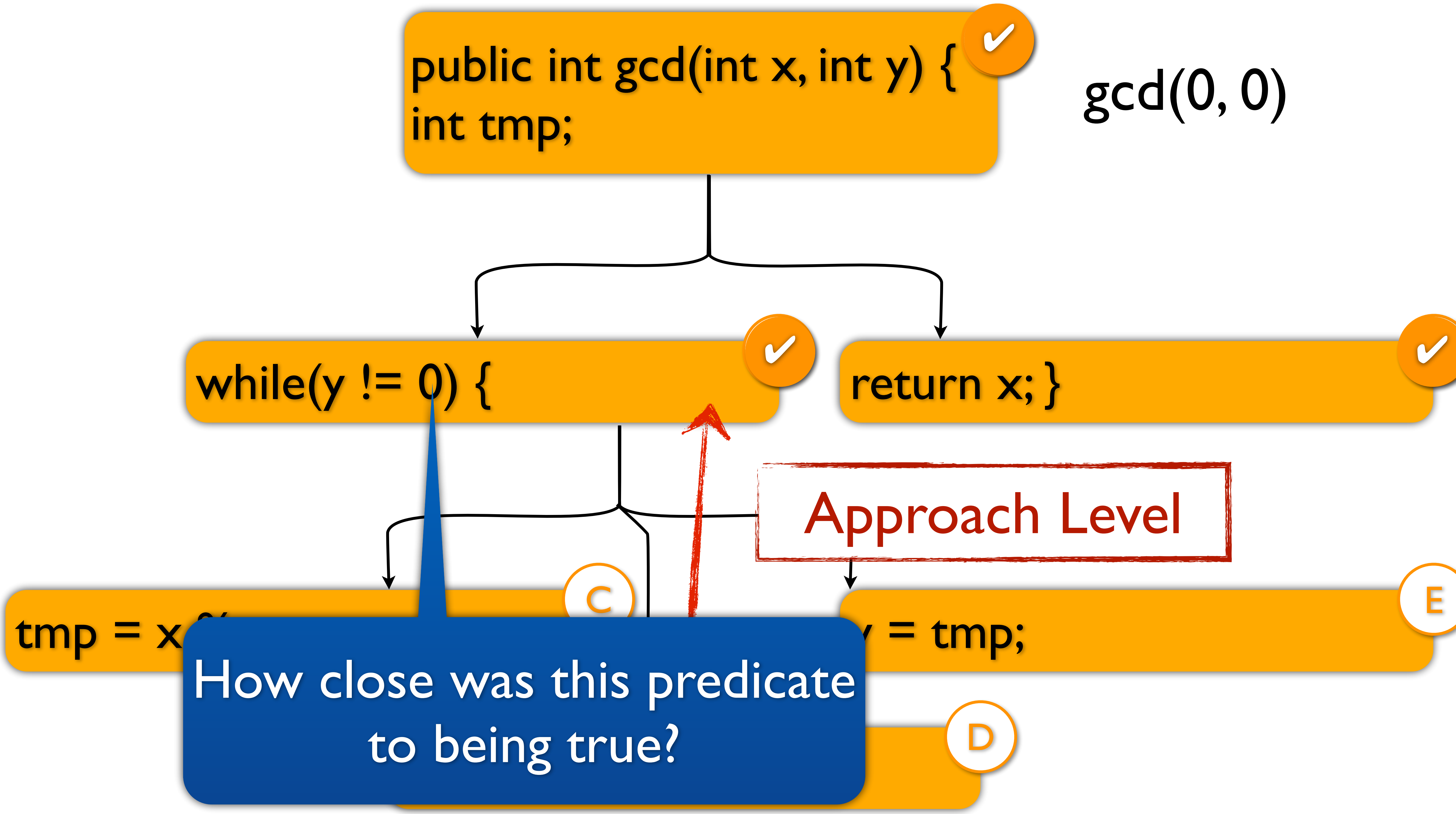

A
Entry
B
x <= y
I
Exit
false
true
true
H
return False
x == y
C
x > 0
E
true
true
print
D
y == 17
F
true
return True
G


Entry
A
B
x <= y
true
false
C
x == y
true
false
print
D
E
x > 0
true
false
y == 17
true
F
false
H
return False
return True
G
Exit
I

```
void landscape_example(int i, int j) {
  if (i >= 10 && i <= 20) {
    if (j >= 0 && j <= 10) {
      // target statement
      // ...
    }
  }
}
```



2
1.5
1
0.5
0
-60
-40
-20
0
20
40
60
i
-60
-40
-20
0
20
40
60
j


public int gcd(int x, int y) {
int tmp;
gcd(0, 0)
while(y != 0) {
return x; }
Approach Level
tmp = x
C
y = tmp;
E
How close was this predicate to being true?
D

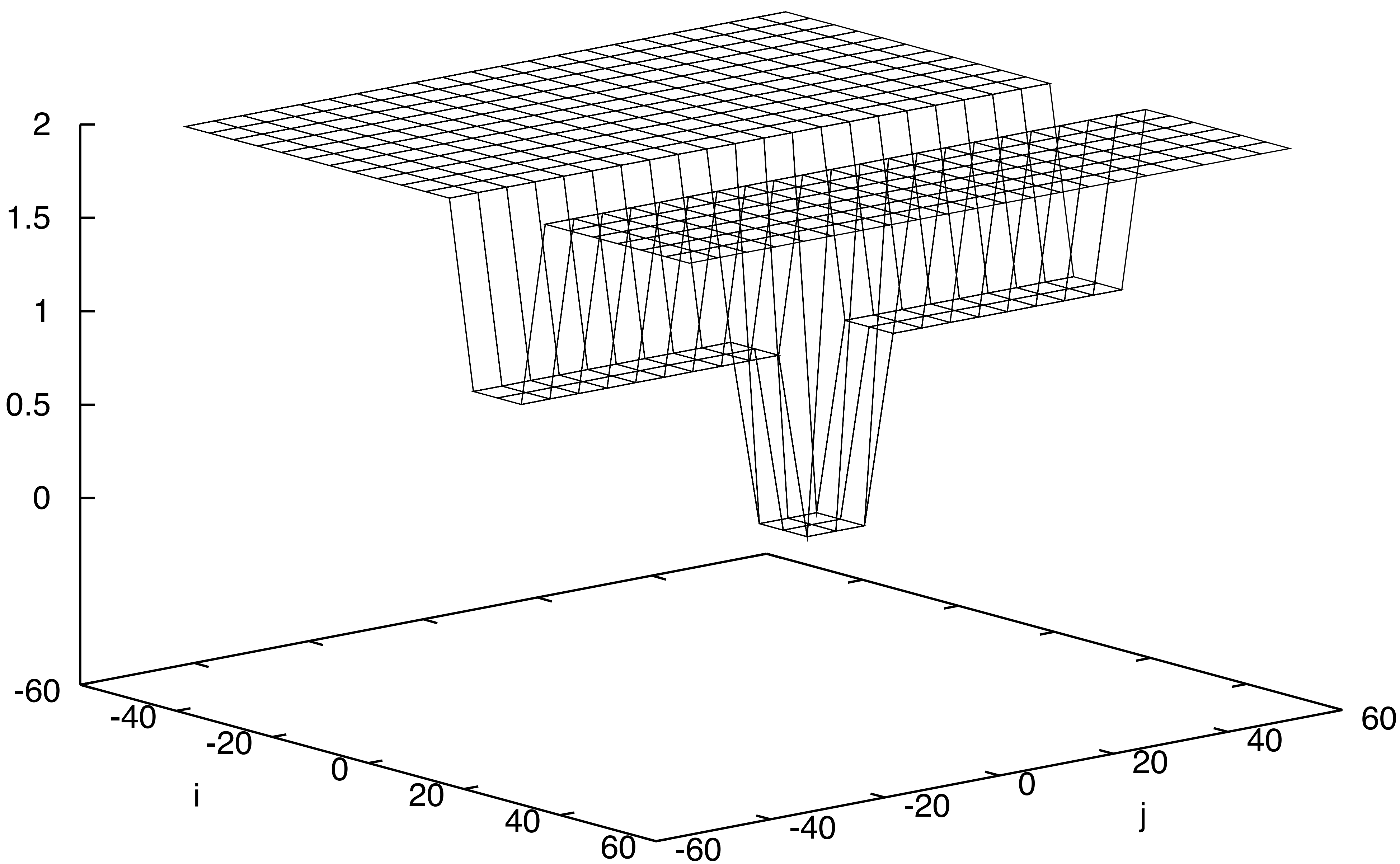
```java
public int gcd(int x, int y) {
  int tmp;
  while (y != 0) {
    tmp = x % y;
    x = y;
    y = tmp;
  }
  return x;
}
```

Branch Distance

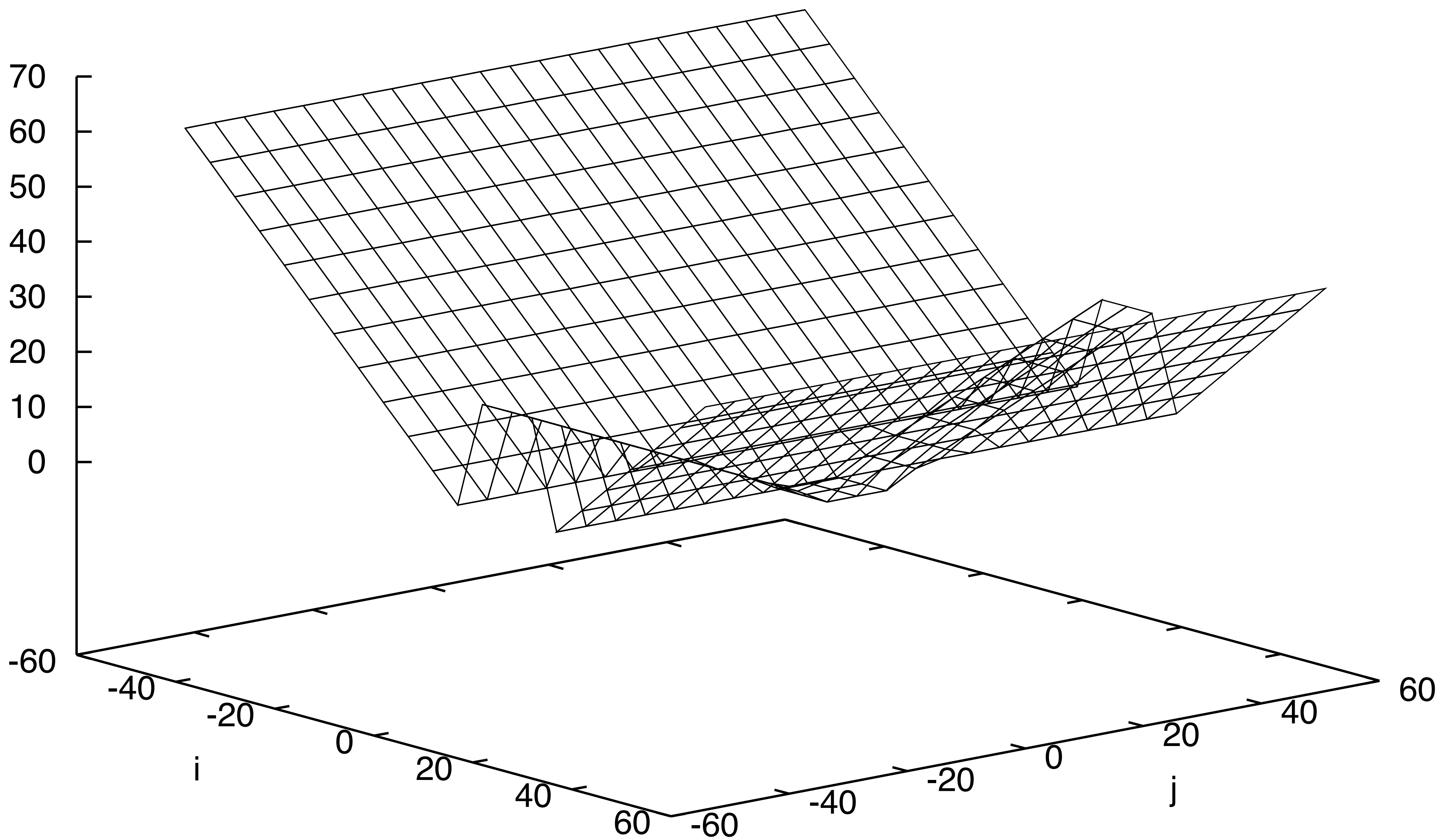
Expression	Distance True	Distance False
x == y	$\lvert x - y \rvert$	1
x != y	1	$\lvert x - y \rvert$
x > y	y - x + 1	x - y
x >= y	y - x	x - y + 1
x < y	x - y + 1	y - x
x <= y	x - y	y - x + 1

```
void landscape_example(int i, int j) {
  if (i >= 10 && i <= 20) {
    if (j >= 0 && j <= 10) {
      // target statement
      // ...
    }
  }
}
```

P. McMinn, Search-based Software Testing: A Survey

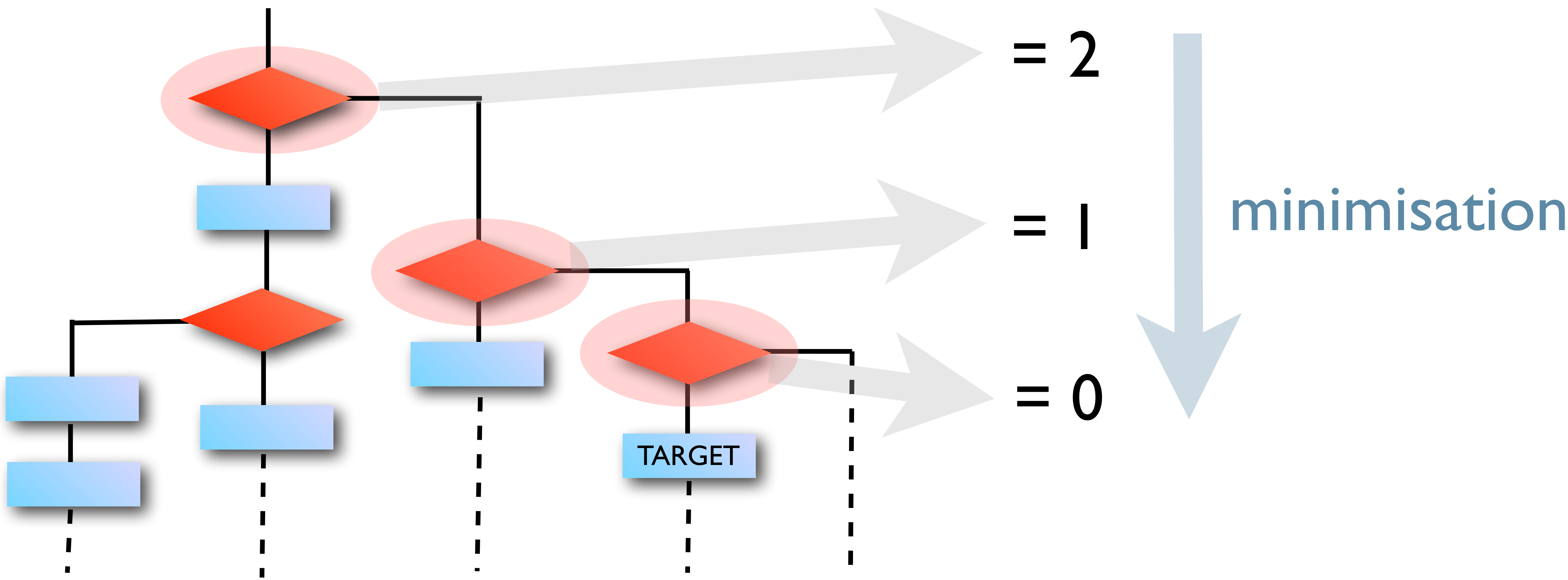


P. McMinn, Search-based Software Testing: A Survey



P. McMinn, Search-based Software Testing: A Survey

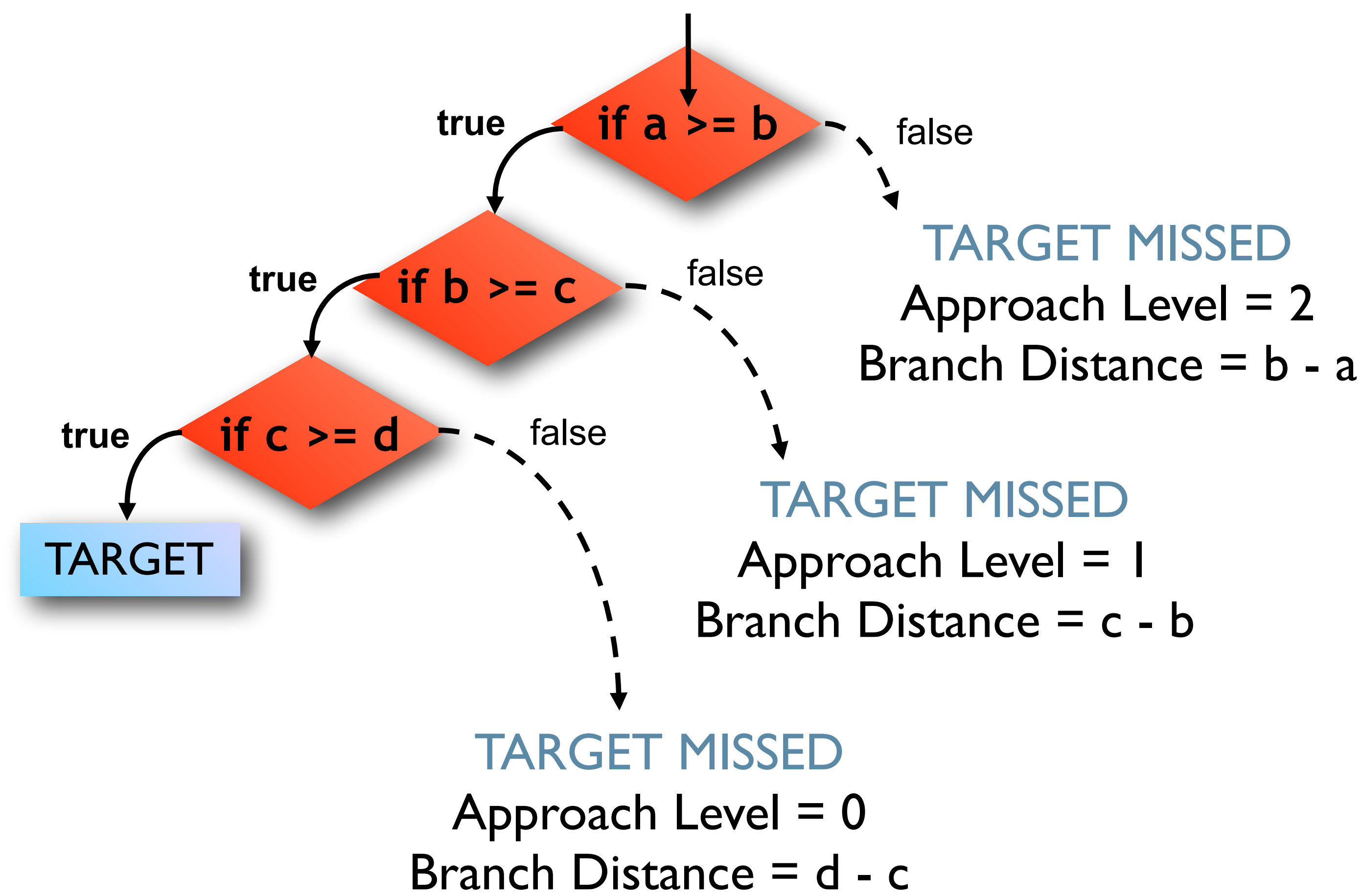
Approach Level



Putting it all together

Fitness = approach Level + *normalised* branch distance

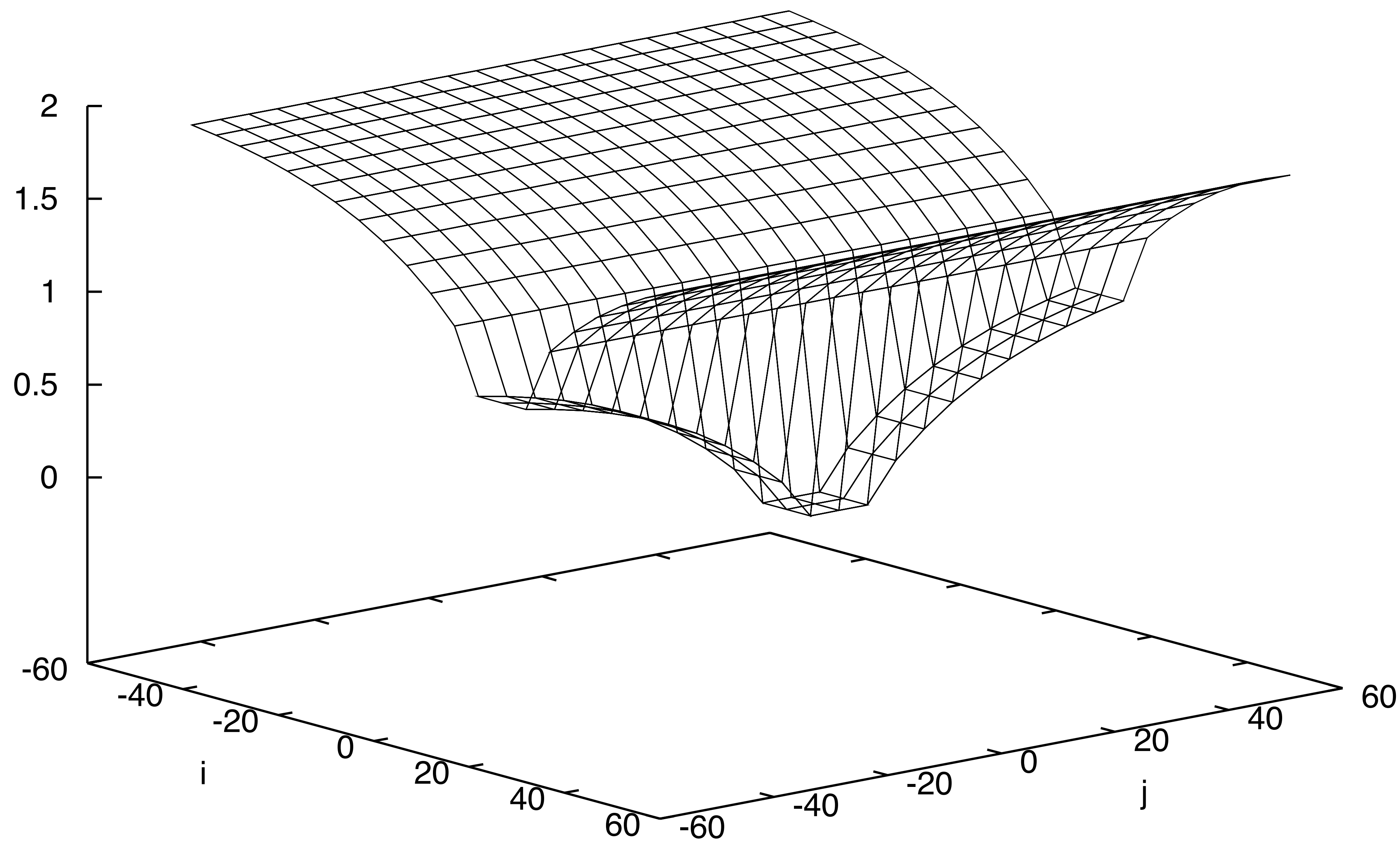
```
void f1(int a, int b, int c, int d)
{
    if (a > b)
    {
        if (b > c)
        {
            if (c > d)
            {
                // target
. . .
```



normalised branch distance between 0 and 1

indicates how close approach level is to being penetrated

Approach level + normalised(Branch distance)



P. McMinn, Search-based Software Testing: A Survey

Many-Objective Optimisation

- Single Objective Formulation: Minimise sum of branch distances for all branches
- Many-Objective Formulation: For each branch, find test that minimises branch distance
- Problem: The proportion of non-dominated solutions increases exponentially with the number of objectives, i.e., all or most of the individuals are non-dominated.
- Domain specific knowledge is needed to impose an order of preference over test cases that are non-dominated according to traditional nondominance
  - For branch coverage, this means focusing the search effort on the test cases that are closer to one or more uncovered branches of a program

Dominance vs. Preference

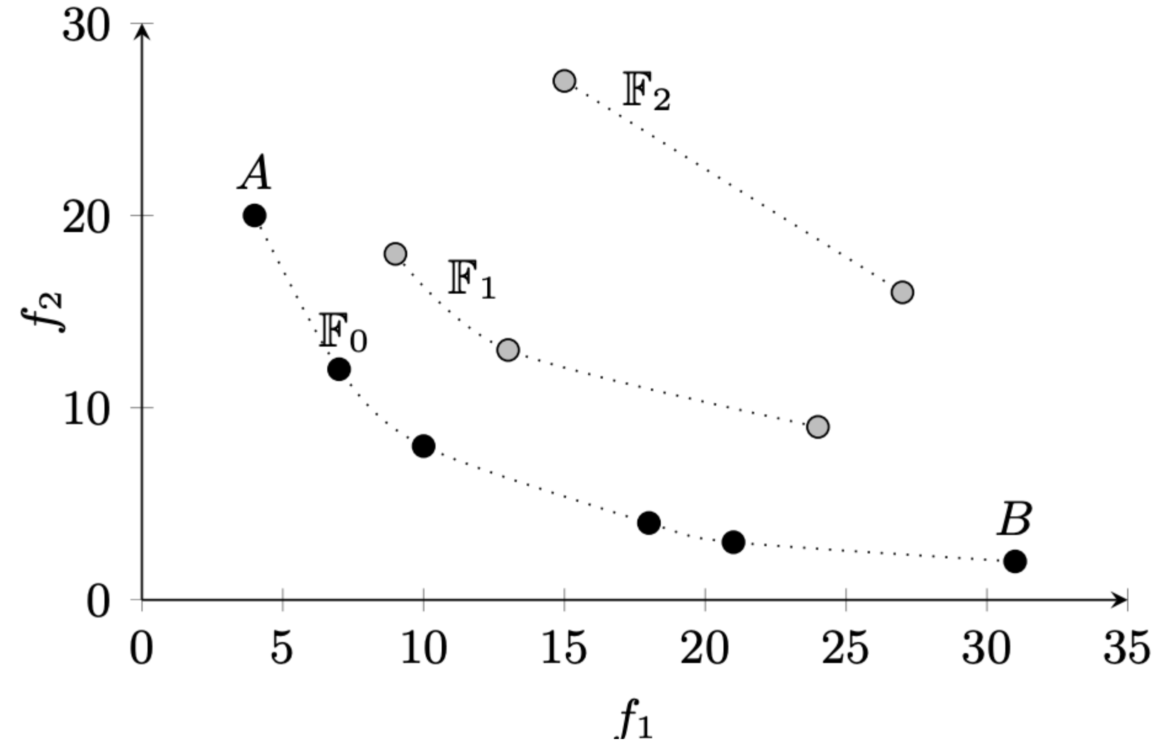
- Dominance: A test case x dominates another test case y if and only if the values of the objective functions satisfy the following conditions:
  - $\forall i \in \{1, \ldots, m\}$ $f_i(x) \leq f_i(y)$ and $\exists j \in \{1, \ldots, m\}$ such that $f_j(x) < f_j(y)$
- Preference: Given a branch b_i, a test case x is preferred over another test case y if and only if the values of the objective function for b_i satisfy the following condition:
  - $f_i(x) < f_i(y)$ where $f_i(x)$ denotes the objective score of test case x for branch b_i
- The best test case for a branch b_i is the one preferred over all other tests for b_i
- The set of best test cases across all uncovered branches defines a subset of the Pareto front that is given priority over the other non-dominated test cases.
- When there are multiple test cases with the same minimum fitness value for a given branch b_i, we use the test case length as a secondary preference criterion

Many-Objective Sorting Algorithm (MOSA)

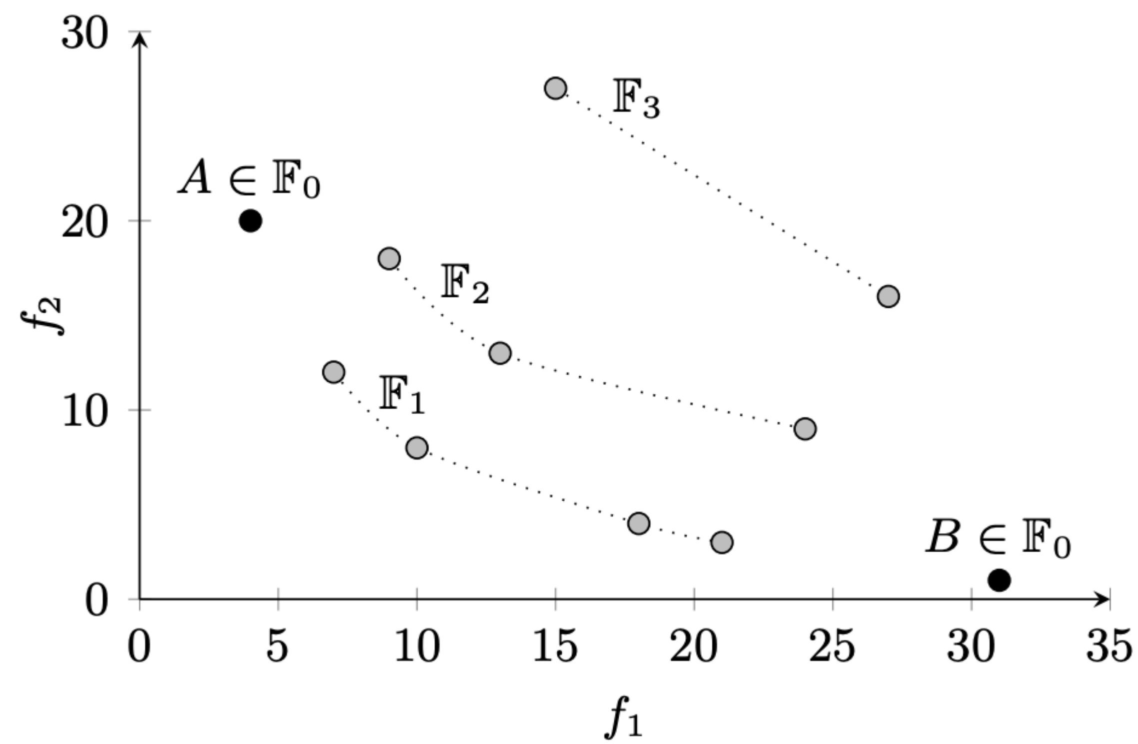
- Based on NSGA-II
- Rank 0 (First non-dominated front): For each uncovered branch b_i the test case that is closest to covering b_i
- The remaining test cases are ranked according to the traditional non-dominated sorting algorithm used by the NSGA-II, starting with a rank equal to 1
  - Ranks are calculated by considering only the non-dominance relation for the *uncovered* branches, i.e., by focusing the search toward the interesting sub-region of the search space
- Once a rank is assigned to all candidate test cases, the crowding distance is used in order to make a decision about which test case to select
- MOSA uses a second population (archive), to keep track of the best test cases
  - After each generation MOSA stores every test case that covers previously uncovered branches in the archive as a candidate test case to form the final test suite
  - This function considers both the covered branches and the length of test cases when updating the archive: for each covered branch b_i it stores the shortest test case covering b_i in the archive.

MOSA: Example

```
  int example(int a, int b, int c) {
1   if (a == b)
2     return 1;
3   if (b == c)
4     return -1;
5   return 0;
  }
```



- Assume that the uncovered goals are the true branches of nodes 1 and 3, whose branch predicates are (a == b) and (b == c) respectively.



- I.e. there are two residual optimisation goals, which are:
  - $f_1 = al(b_1) + bd(b_1) = abs(a - b)$
  - $f_2 = al(b_2) + bd(b_2) = abs(b - c)$

Input: Stopping condition C, Fitness function δ, Population size p_s, Crossover function c_f, Crossover probability c_p, Mutation probability m_p

Output: Archive of optimised individuals A

```
p ← 0
N_p ← GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, N_p)
A ← { }
while ¬C do
    N_o ← GenerateOffspring(c_f, c_p, m_p, N_p)
    R_t ← N_p ∪ N_o
    r ← 0
    F_r ← PreferenceSorting(R_t)
    N_{p+1} ← { }
    while |N_{p+1}| + |F_r| ≤ p_s do
        CalculateCrowdingDistance(F_r)
        N_{p+1} ← N_{p+1} ∪ F_r
        r ← r + 1
    end while
    DistanceCrowdingSort(F_r)
    N_{p+1} ← N_{p+1} ∪ F_r with size p_s − |N_{p+1}|
    UpdateArchive(A, N_{p+1})
    p ← p + 1
end while
return A
```

MOSA

- Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Reformulating branch coverage as a many-objective optimization problem." IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015.