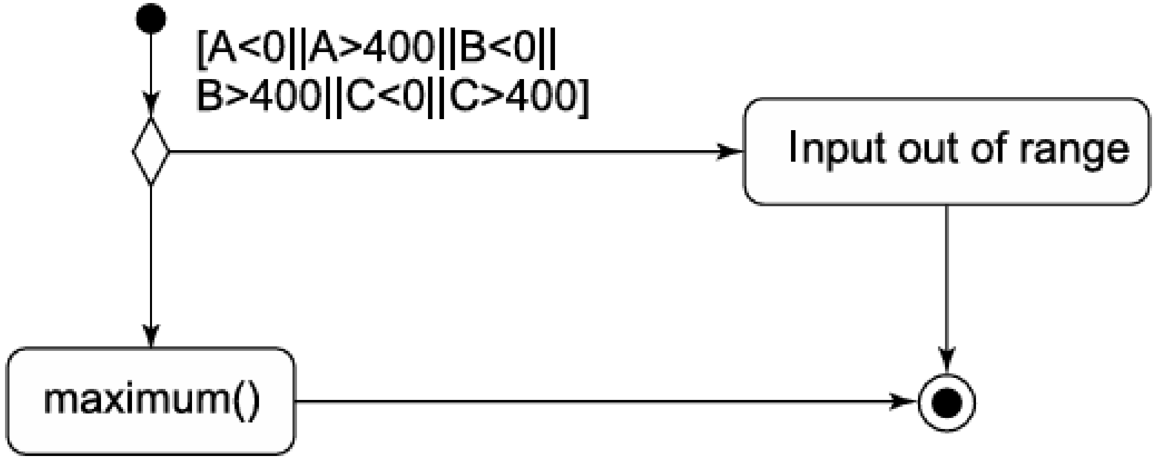
Calculation of Cyclomatic Complexity

- V(G)=e-n+2P
- Where e: number of edges of a graph G
    - n: number of nodes of a graph G
    - P: number of connected components
-

```
#include<iostream.h>
#include<conio.h>
class greatest
{
float A;
float B;
float C;
public:
void getdata()
{
cout<<”Enter number 1:\n”;
cin>>A;
cout<<”Enter number 2:\n”;
cin>>B;
cout<<”Enter number 3:\n”;
cin>>C;
}
void validate()
{
if(A<0||A>400||B<0||B>400||C<0||C>400){
        cout<<”Input out of range”;
}
else{
maximum();
}
}
void greatest::maximum()
{
/*Check for greatest of three numbers*/
if(A>B) {
if(A>C) {
        cout<<A;
        }
else  {
        cout<<C;
        }
}
else {
if(C>B) {
        cout<<C;
        }
else  {
        cout<<B;
        }
}
}
```



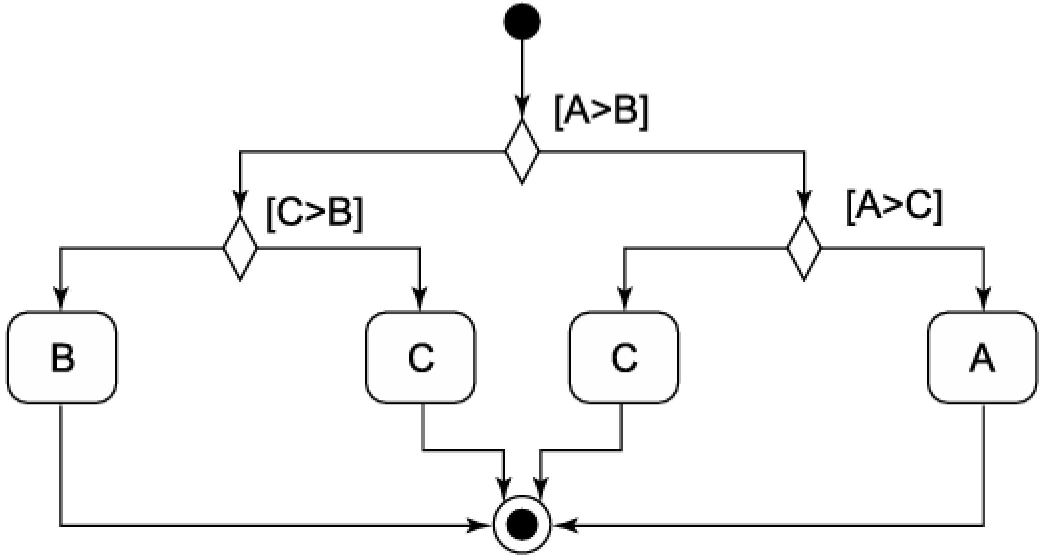
Cyclomatic complexity = e-n+2P =transitions – activities/branches +2P

$$= 5 - 5 + 2$$

$$= 2$$

Table 9.4. Test cases of activity diagram in Figure 9.10

Test case	A	B	C	Path
1.	500	40	50	Input out of range
2.	90	75	75	maximum()



Cyclomatic complexity = e – n + 2P = transitions – activities/branches +2P

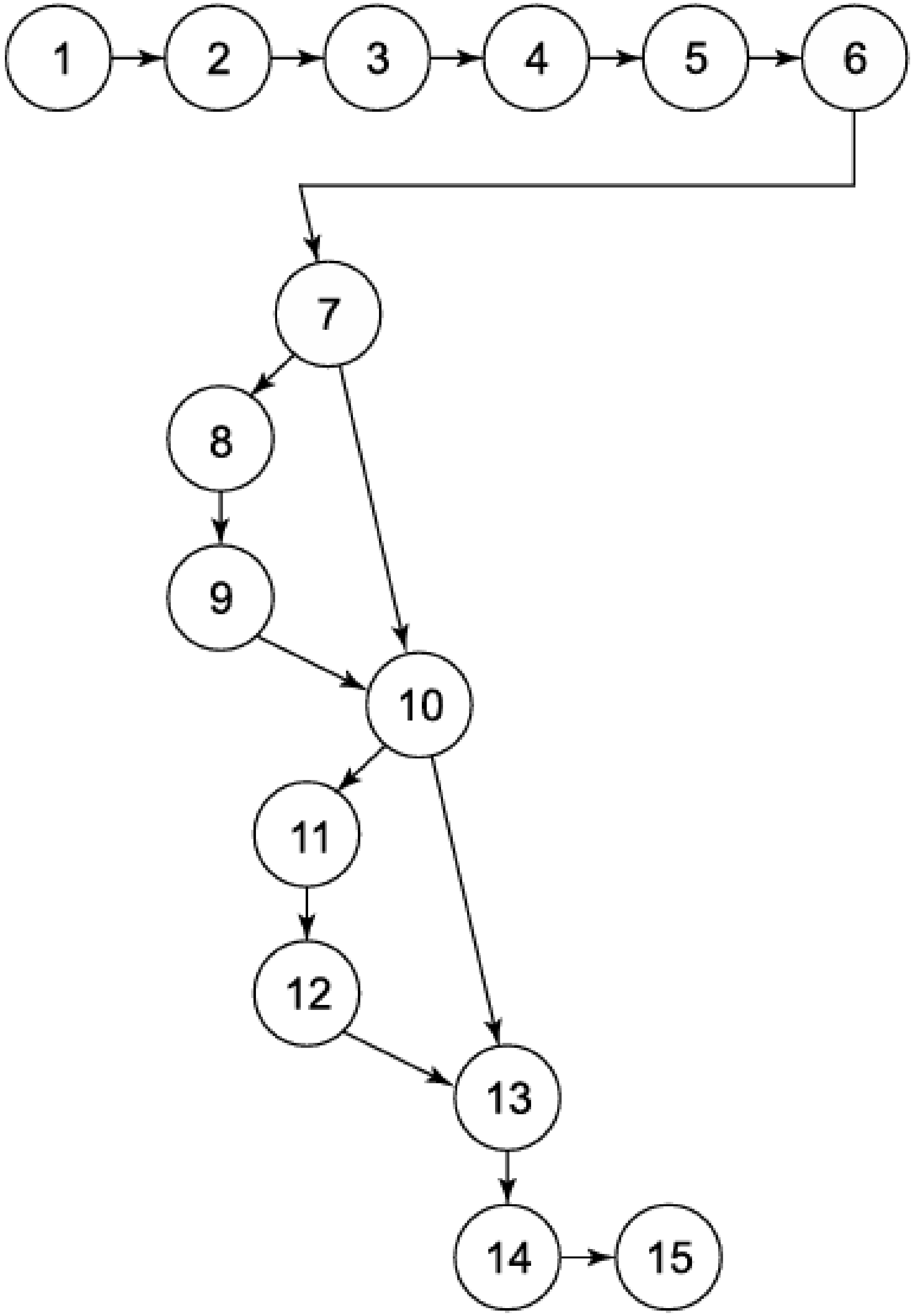
11 – 9 + 2 = 4

Table 9.5. Test cases of activity diagram in Figure 9.11

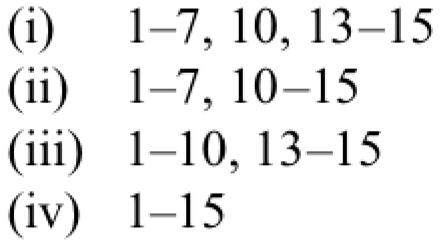
Test case	A	B	C	Expected output
1.	100	87	56	100
2.	87	56	100	100
3.	56	87	100	100
4.	87	100	56	100

```
        #include<stdio.h>
        #include<conio.h>

1.      void main()
2.      {
3.      int a,b,c,x=0,y=0;
4.      clrscr();
5.      printf("Enter three numbers:");
6.      scanf("%d %d %d",&a,&b,&c);
7.      if((a>b)&&(a>c)){
8.              x=a*a+b*b;
9.      }
10.     if(b>c){
11.             y=a*a-b*b;
12.     }
13.     printf("x= %d y= %d",x,y);
14.     getch();
15.     }
```



- All paths



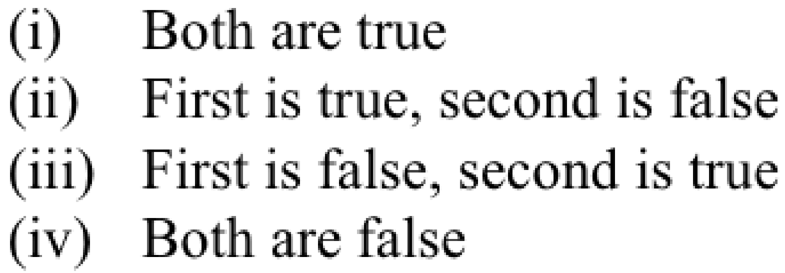
(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

- Statement 1-15
- Branch 1-15

1-7, 10, 13-15

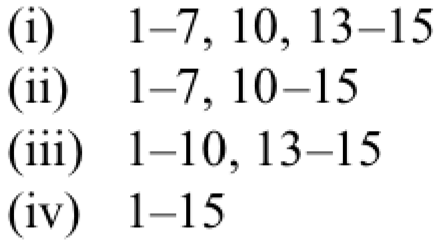
- Condition 1-15

1-10, 13-15
1-7, 10-15
1-7, 10, 13-15

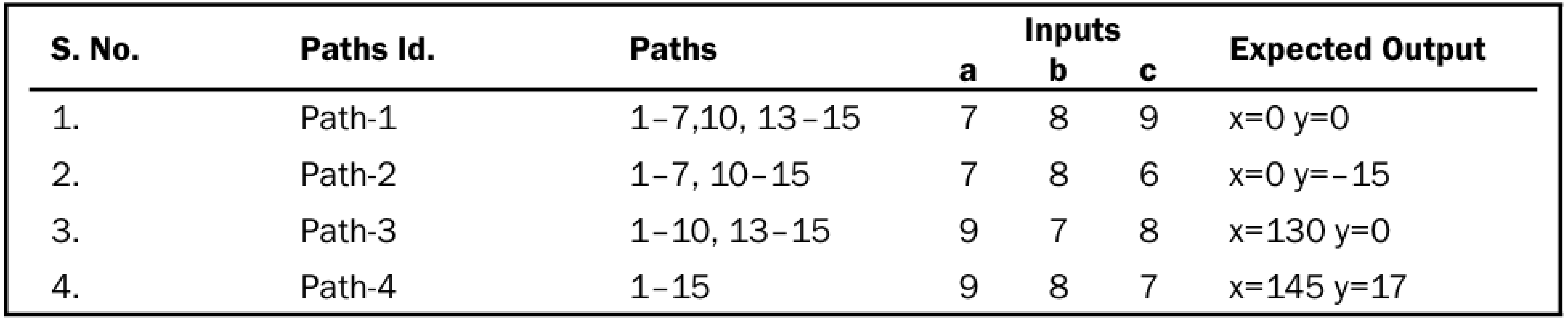


(i) Both are true
(ii) First is true, second is false
(iii) First is false, second is true
(iv) Both are false

- All paths

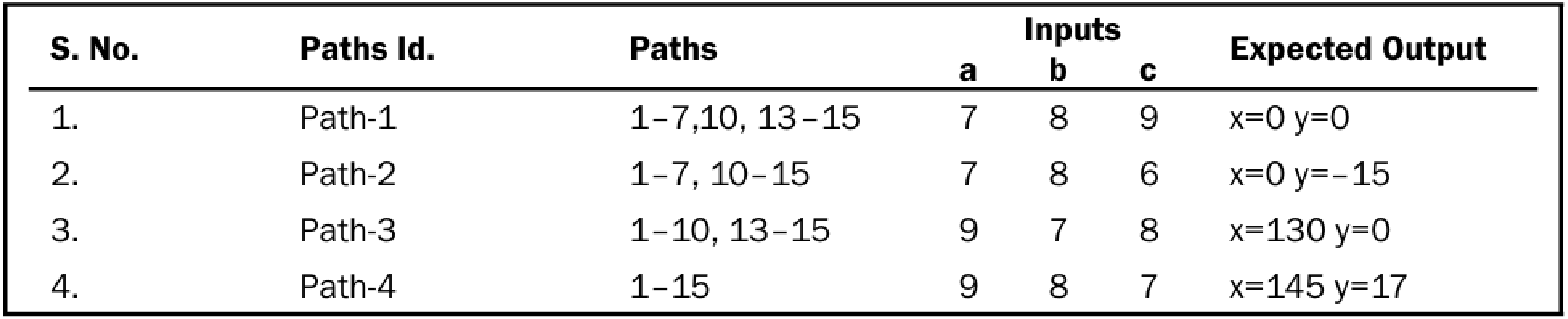


(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

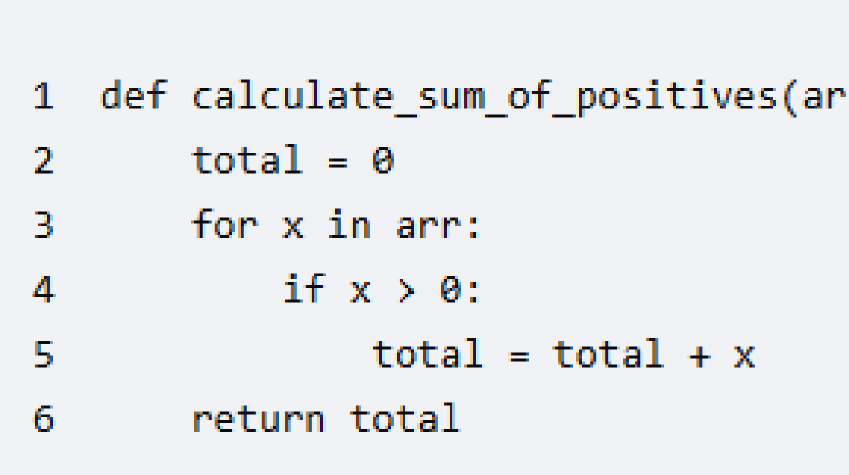


S. No.	Paths Id.	Paths	Inputs			Expected Output
			a	b	c	
1.	Path-1	1–7,10, 13–15	7	8	9	x=0 y=0
2.	Path-2	1–7, 10–15	7	8	6	x=0 y=–15
3.	Path-3	1–10, 13–15	9	7	8	x=130 y=0
4.	Path-4	1–15	9	8	7	x=145 y=17

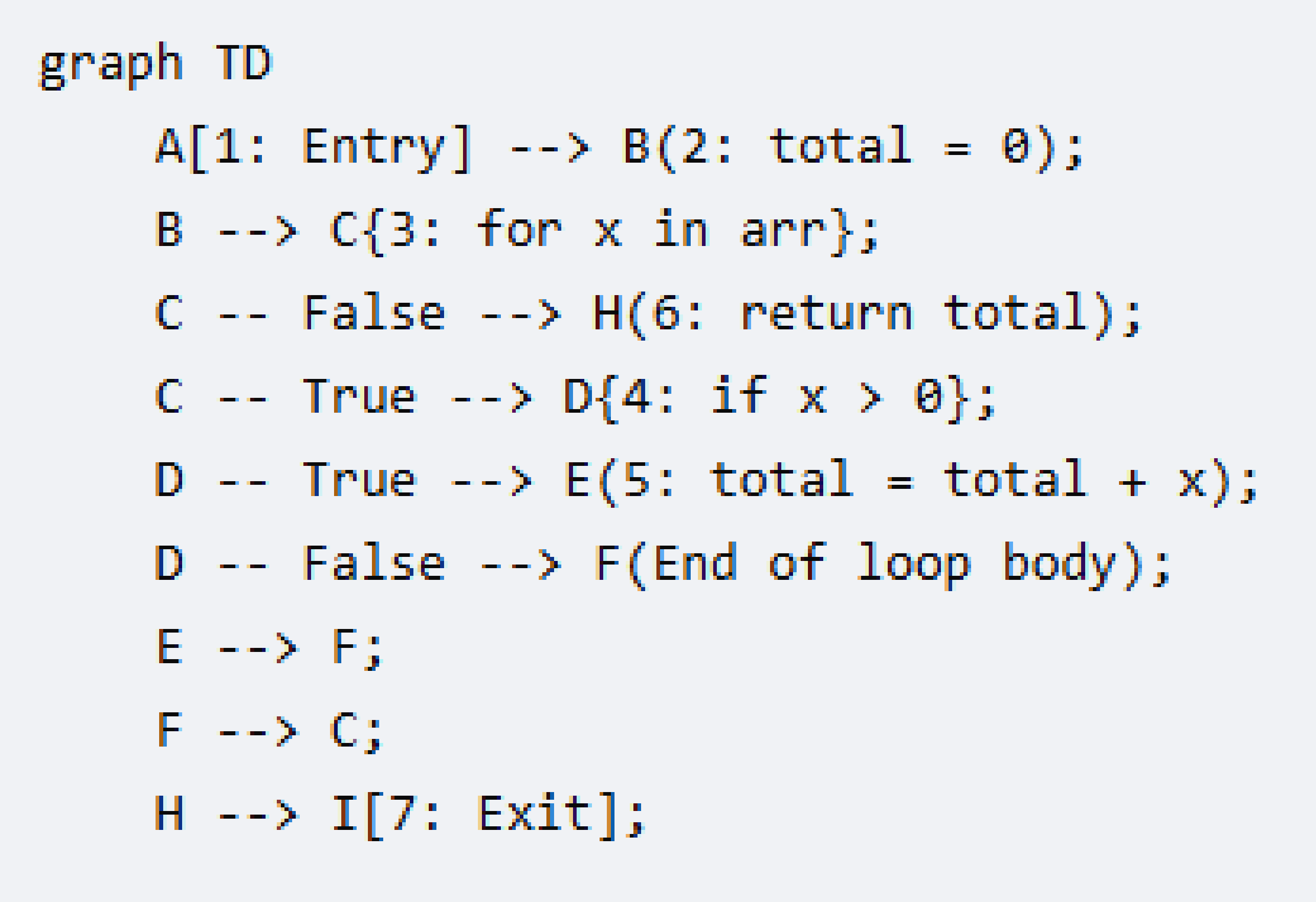
Basis Paths?



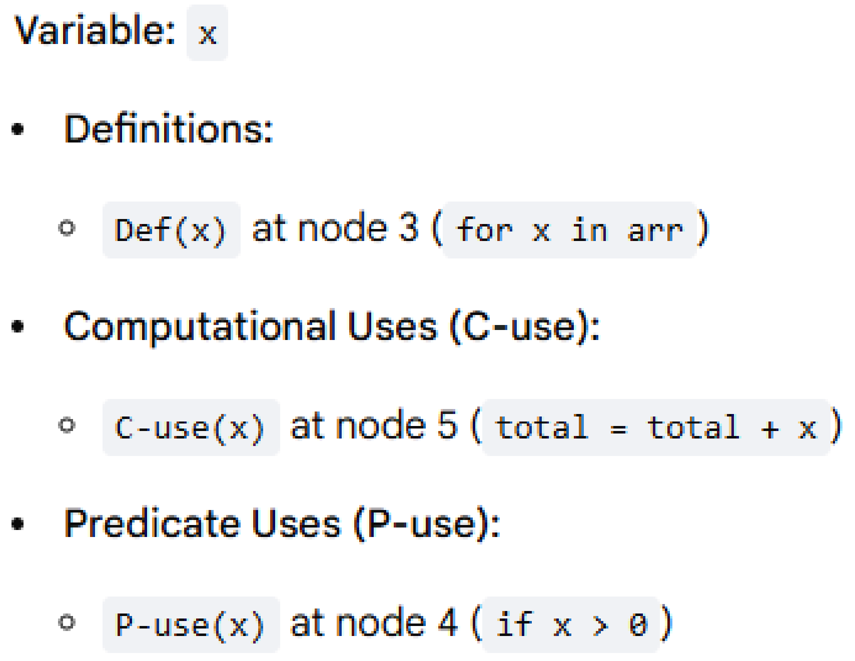
S. No.	Paths Id.	Paths	Inputs			Expected Output
			a	b	c	
1.	Path-1	1–7,10, 13–15	7	8	9	x=0 y=0
2.	Path-2	1–7, 10–15	7	8	6	x=0 y=–15
3.	Path-3	1–10, 13–15	9	7	8	x=130 y=0
4.	Path-4	1–15	9	8	7	x=145 y=17



```
1  def calculate_sum_of_positives(arr):
2      total = 0
3      for x in arr:
4          if x > 0:
5              total = total + x
6      return total
```

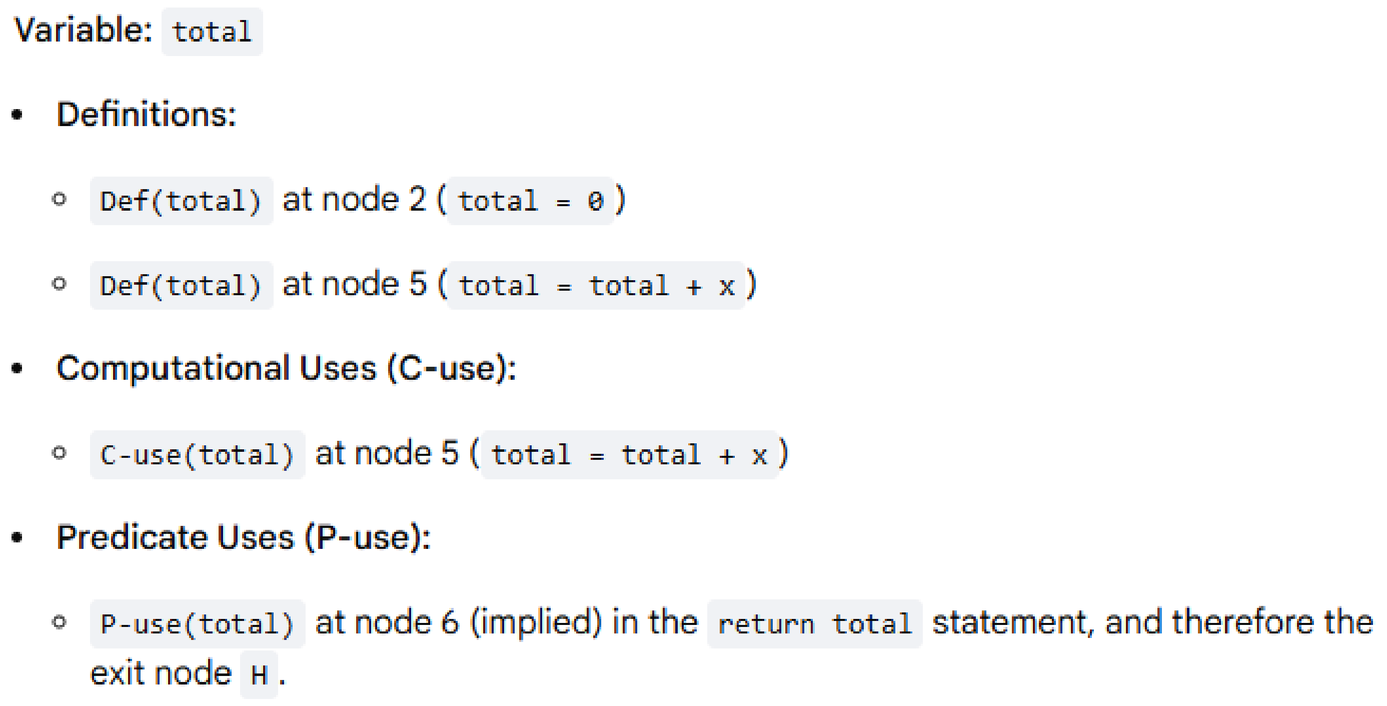


```
graph TD
    A[1: Entry] --> B(2: total = 0);
    B --> C{3: for x in arr};
    C -- False --> H(6: return total);
    C -- True --> D{4: if x > 0};
    D -- True --> E(5: total = total + x);
    D -- False --> F(End of loop body);
    E --> F;
    F --> C;
    H --> I[7: Exit];
```



Variable: `x`

- **Definitions:**
  - `Def(x)` at node 3 (`for x in arr`)
- **Computational Uses (C-use):**
  - `C-use(x)` at node 5 (`total = total + x`)
- **Predicate Uses (P-use):**
  - `P-use(x)` at node 4 (`if x > 0`)

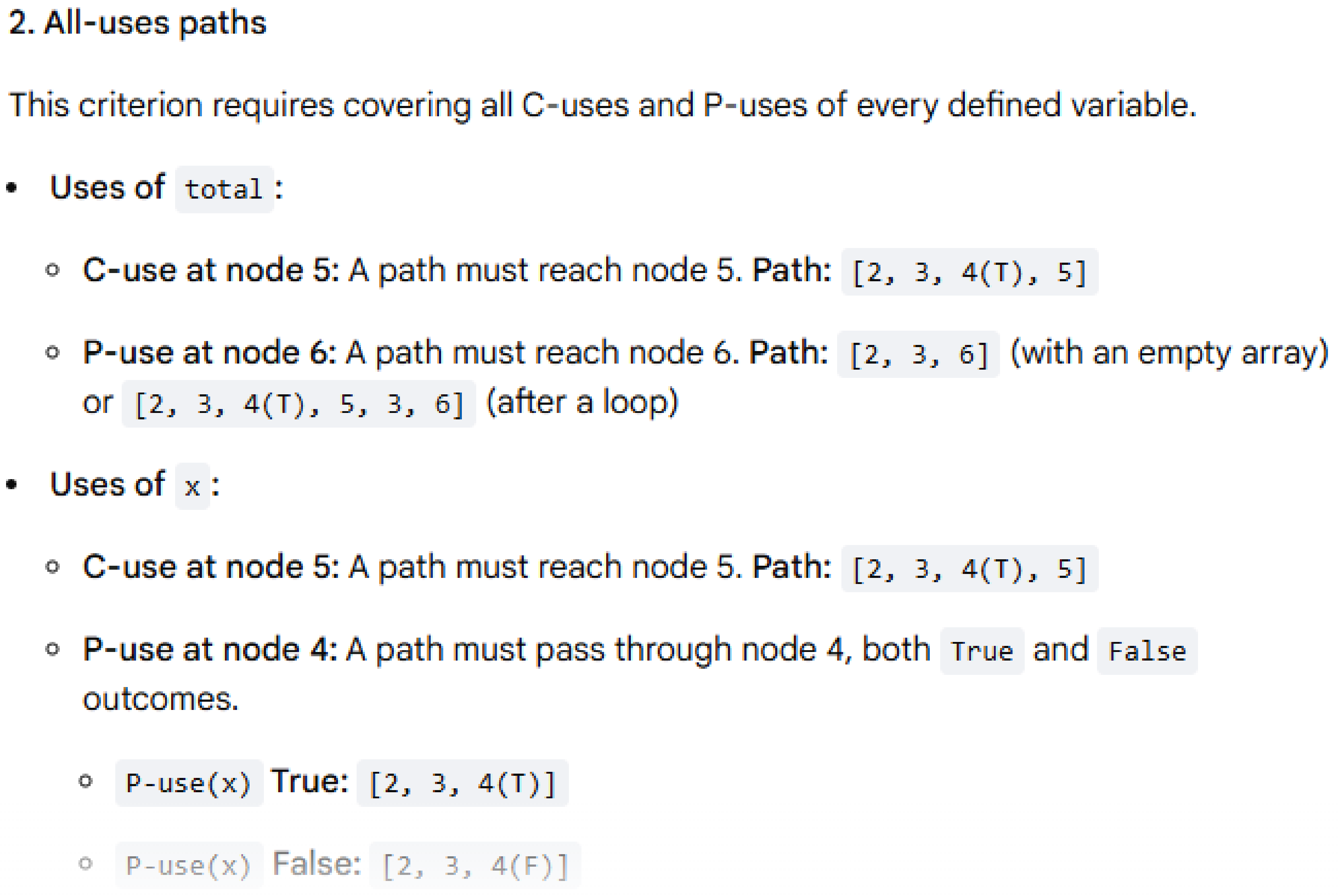


Variable: `total`

- **Definitions:**
  - `Def(total)` at node 2 (`total = 0`)
  - `Def(total)` at node 5 (`total = total + x`)
- **Computational Uses (C-use):**
  - `C-use(total)` at node 5 (`total = total + x`)
- **Predicate Uses (P-use):**
  - `P-use(total)` at node 6 (implied) in the `return total` statement, and therefore the exit node `H` .

- du-paths for `total`:
  - `(def: 2, c-use: 5)`: Path: `[2, 3, 4(T), 5]` (where `x > 0`)
  - `(def: 2, p-use: 6)`: Path: `[2, 3, 6]` (where `arr` is empty)
  - `(def: 5, c-use: 5)`: Path: `[2, 3, 4(T), 5, 3, 4(T), 5]` (multiple loop iterations)
  - `(def: 5, p-use: 6)`: Path: `[2, 3, 4(T), 5, 3, 6]` (loop finishes after one or more iterations)

- du-paths for `x`:
  - `(def: 3, c-use: 5)`: Path: `[2, 3, 4(T), 5]`
  - `(def: 3, p-use: 4)`: Paths: `[2, 3, 4(T)]` and `[2, 3, 4(F)]`



2. All-uses paths

This criterion requires covering all C-uses and P-uses of every defined variable.

- Uses of `total`:
  - **C-use at node 5:** A path must reach node 5. **Path:** `[2, 3, 4(T), 5]`
  - **P-use at node 6:** A path must reach node 6. **Path:** `[2, 3, 6]` (with an empty array) or `[2, 3, 4(T), 5, 3, 6]` (after a loop)
- Uses of `x`:
  - **C-use at node 5:** A path must reach node 5. **Path:** `[2, 3, 4(T), 5]`
  - **P-use at node 4:** A path must pass through node 4, both `True` and `False` outcomes.
    - `P-use(x)` **True:** `[2, 3, 4(T)]`
    - `P-use(x)` False: `[2, 3, 4(F)]`

3. All-definitions paths

This criterion requires covering at least one use for every defined variable.

- **Definition of `total` at node 2:** Can be covered by either a C-use at node 5 or a P-use at node 6. **Path:** `[2, 3, 4(T), 5]` covers the C-use, and `[2, 3, 6]` covers the P-use.
- **Definition of `total` at node 5:** Can be covered by the C-use at node 5 in the next loop iteration or the P-use at node 6. **Path:** `[2, 3, 4(T), 5, 3, 6]` covers both.
- **Definition of `x` at node 3:** Can be covered by the P-use at node 4 or the C-use at node 5. **Path:** `[2, 3, 4(T), 5]` covers both.