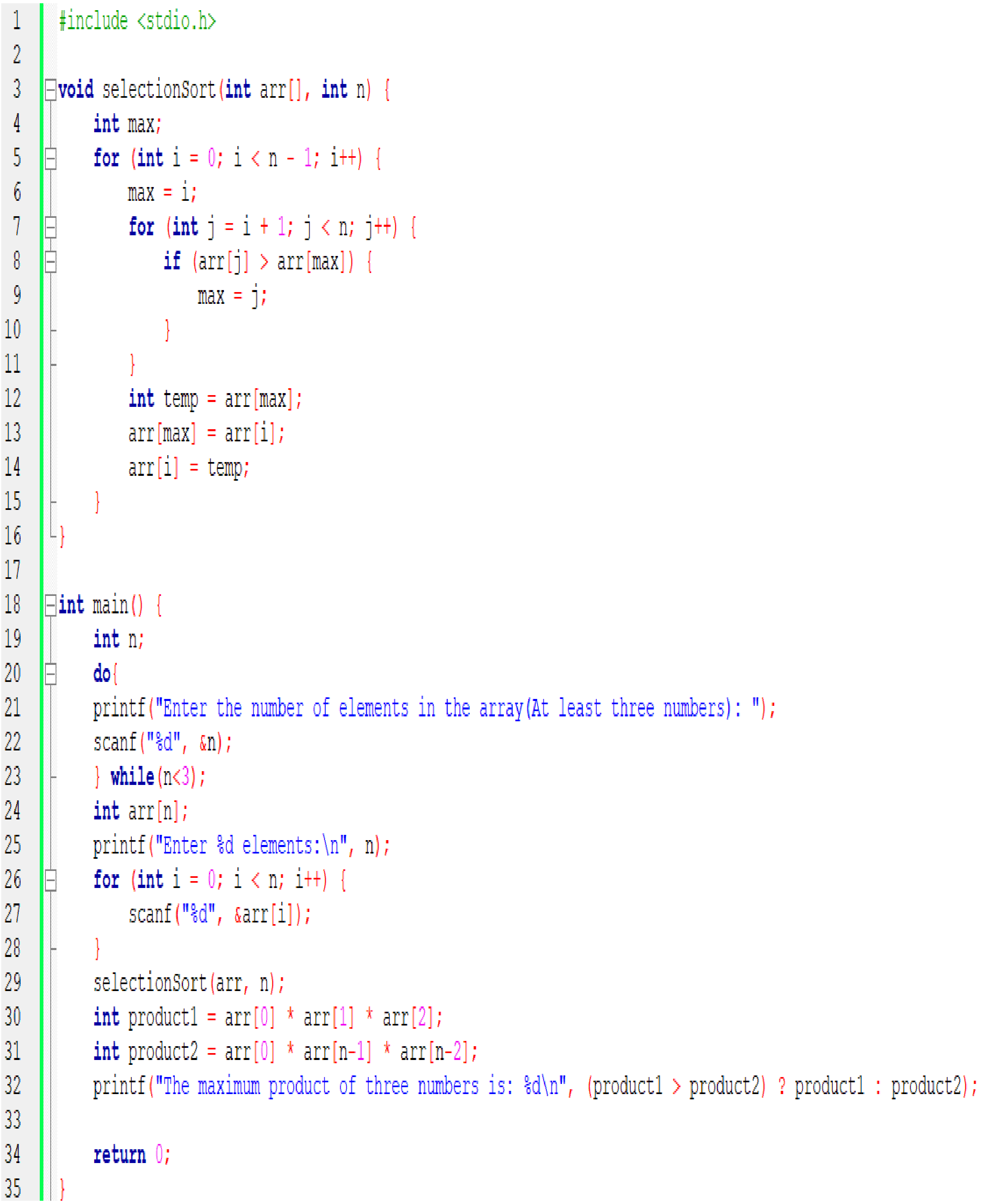
Implementation Non-recursive Algorithm of Max Product Of Three



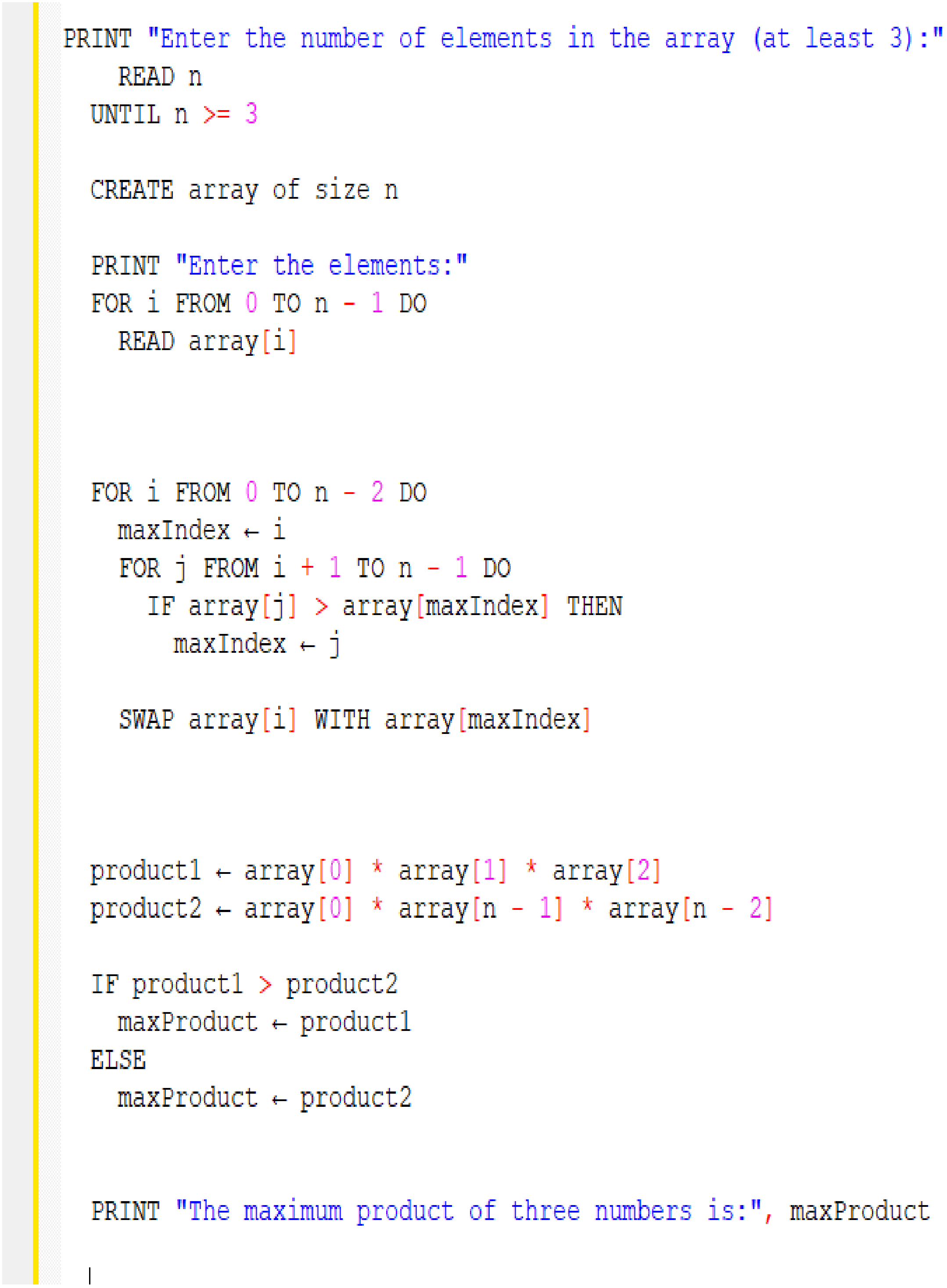
```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    int max;
    for (int i = 0; i < n - 1; i++) {
        max = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] > arr[max]) {
                max = j;
            }
        }
        int temp = arr[max];
        arr[max] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int n;
    do{
    printf("Enter the number of elements in the array(At least three numbers): ");
    scanf("%d", &n);
    } while(n<3);
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    selectionSort(arr, n);
    int product1 = arr[0] * arr[1] * arr[2];
    int product2 = arr[0] * arr[n-1] * arr[n-2];
    printf("The maximum product of three numbers is: %d\n", (product1 > product2) ? product1 : product2);

    return 0;
}
```

Pseudo code Non-recursive Algorithm of Max Product Of Three



```
PRINT "Enter the number of elements in the array (at least 3):"
    READ n
  UNTIL n >= 3

  CREATE array of size n

  PRINT "Enter the elements:"
  FOR i FROM 0 TO n - 1 DO
    READ array[i]



  FOR i FROM 0 TO n - 2 DO
    maxIndex ← i
    FOR j FROM i + 1 TO n - 1 DO
      IF array[j] > array[maxIndex] THEN
        maxIndex ← j

    SWAP array[i] WITH array[maxIndex]



  product1 ← array[0] * array[1] * array[2]
  product2 ← array[0] * array[n - 1] * array[n - 2]

  IF product1 > product2
    maxProduct ← product1
  ELSE
    maxProduct ← product2


  PRINT "The maximum product of three numbers is:", maxProduct
```

Analysis & complexity Non-recursive Algorithm Max Product Of Three

---

1. Input-validation loop

```
do {
 printf(...);      // A
 scanf("%d", &n); // B
} while (n < 3);   // C
```

- Let k = number of times the user enters an invalid n (< 3) before finally entering a valid one.
- **Total iterations** of the body = k + 1
- **printf calls (A):** k + 1 → O(1)
- **scanf calls (B):** k + 1 → O(1)
- **n<3 tests (C):** one per loop check, including the final false check → k + 2 → O(1)
- **Overall cost:** A constant number of O(1) ops, independent of n.

---

2. Reading the array

```
for (i = 0; i < n; i++) {
 scanf("%d", &arr[i]);
}
```

- **Loop-entry tests (i<n):** n + 1
- **Increments (i++):** n
- **scanf calls:** n
- **Total:**
  - Tests: n + 1
  - Increments: n

    - I/O: n
      ⇒ Θ(n) operations

---

3. Selection sort (descending version)

```
for (i = 0; i < n - 1; i++) {
 max = i;
 for (j = i + 1; j < n; j++) {
  if (arr[j] > arr[max]) max = j;
 }
 // swap arr[i] and arr[max]:
 temp     = arr[max];
 arr[max]  = arr[i];
 arr[i]    = temp;
}
```

Operation	**Count**	**Total**
Outer-loop tests (i < n-1)	$(n-1) + 1$	n
Outer-loop increments (i++)	$n-1$	$n-1$
max = i; assignments	once per outer iteration	$n-1$
Inner-loop tests (j < n)	$\sum_{i=0}^{n-2} [(n - (i+1)) + 1] = \sum_{k=1}^{n-1} (k + 1) = n(n-1)/2 + (n-1)$	$(n^2-n)/2 + n-1$
Inner-loop increments (j++)	$\sum_{i=0}^{n-2} (n - i - 1) = n(n-1)/2$	$(n^2-n)/2$

Operation	**Count**	**Total**
Comparisons (arr[j] > arr[max])	$\sum_{i=0}^{n-2}$ (n − i − 1) = n(n−1)/2	$(n^2-n)/2$
max = j; assignments	at most once per comparison; worst-case every comparison succeeds ⇒ $(n^2-n)/2$	$(n^2-n)/2$
Swaps (3 assignments each)	one per outer iteration	3·(n−1)

- **Total comparisons:**

n(n−1)2=n2−n2. \frac{n(n-1)}{2} = \frac{n^2 - n}{2}.

- **Total assignments (excluding loop counters/tests):**

(n−1) [max⁡=i]+n(n−1)2 [max⁡=j]+3(n−1) [swap]=n(n−1)2+4(n−1). (n-1)\;[\max=i] + \frac{n(n-1)}{2}\;[\max=j] + 3(n-1)\;[\text{swap}] = \frac{n(n-1)}{2} + 4(n-1).

- **Loop-overhead (tests + increments):** two nested loops contribute $\Theta(n^2)$ tests/increments as well.

- ⇒ **Selection sort is $\Theta(n^2)$** overall.

---

4. Final product computation & output

```
int product1 = arr[0]     * arr[1]     * arr[2];

int product2 = arr[0]     * arr[n-1]    * arr[n-2];

if (product1 > product2)

  printf("%d", product1);

else

  printf("%d", product2);
```

- **Multiplications:** 6

- **Array-accesses:** 6

- **Comparison (product1>product2):** 1

- **printf call:** 1

- **⇒ Constant time, O(1).**

Grand total

Phase	**Dominant term**
1. Input-validation	O(1)
2. Reading array	Θ(n)
3. Selection sort	Θ(n^2)
4. Product + print	O(1)
Overall	**Θ(n^2)**

- **Exact comparison count:**

$\underbrace{n(n-1)2}_{\text{selection-sort}}$(plus a handful of constant comparisons in loops/input.) \underbrace{\frac{n(n-1)}{2}}_{\text{selection-sort}} \quad\text{(plus a handful of constant comparisons in loops/input.)}

- **Exact swap-assignment count:**

3(n−1)(array-element assignments in swaps). 3(n-1)\quad\text{(array-element assignments in swaps).}

Conclusion: Every major cost term beyond reading input is dominated by the Θ(n^2) comparisons and assignments of selection-sort.