CSC 225 - Summer 2019

Algorithm Analysis II

Bill Bird

Department of Computer Science
University of Victoria

May 10, 2019

PairSum225 Again (1)

```
1:  procedure PairSum225(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = 0, 1, 2, ..., n − 1 do
5:              if A[i] + A[j] = 225 then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

A best case input for the above algorithm for PairSum225 contains a valid pair in the first two positions of the array. For example,

$$A = \quad \mathbf{1}, \quad \mathbf{224}, \quad 6, \quad 10, \quad \ldots$$

PairSum225 Again (2)

```
 1: procedure PairSum225(A)
 2:     n ← Length of A
 3:     for i = 0, 1, 2, ..., n − 1 do
 4:         for j = 0, 1, 2, ..., n − 1 do
 5:             if A[i] + A[j] = 225 then
 6:                 return True
 7:             end if
 8:         end for
 9:     end for
10:     return False
11: end procedure
```

On a best case input, the return statement on line 6 will be executed when $i = 0$ and $j = 1$.

PairSum225 Again (3)

```
 1: procedure PairSum225(A)
 2:     n ← Length of A
 3:     for i = 0, 1, 2, ..., n − 1 do
 4:         for j = 0, 1, 2, ..., n − 1 do
 5:             if A[i] + A[j] = 225 then
 6:                 return True
 7:             end if
 8:         end for
 9:     end for
10:     return False
11: end procedure
```

(Similar to LinearSearch, this algorithm always requires the same amount of time in the best case, regardless of input size)

PairSum225 Again (4)

```
 1: procedure PairSum225(A)
 2:     n ← Length of A
 3:     for i = 0, 1, 2, ..., n − 1 do
 4:         for j = 0, 1, 2, ..., n − 1 do
 5:             if A[i] + A[j] = 225 then
 6:                 return True
 7:             end if
 8:         end for
 9:     end for
10:     return False
11: end procedure
```

A worst case input for PairSum225 does not contain a pair which sums to 225.

PairSum225 Again (5)

```
1:  procedure PairSum225(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = 0, 1, 2, ..., n − 1 do
5:              if A[i] + A[j] = 225 then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

In the worst case, the outer loop must run for n iterations, and for each iteration of the outer loop, the inner loop runs for n iterations.

PairSum225 Again (6)

```
 1: procedure PairSum225(A)
 2:     n ← Length of A
 3:     for i = 0, 1, 2, ..., n − 1 do
 4:         for j = 0, 1, 2, ..., n − 1 do
 5:             if A[i] + A[j] = 225 then
 6:                 return True
 7:             end if
 8:         end for
 9:     end for
10:     return False
11: end procedure
```

Combined, the two loops run for n^2 iterations.

PairSum225 Again (7)

```
 1: procedure PairSum225(A)
 2:     n ← Length of A
 3:     for i = 0, 1, 2, ..., n − 1 do
 4:         for j = 0, 1, 2, ..., n − 1 do
 5:             if A[i] + A[j] = 225 then
 6:                 return True
 7:             end if
 8:         end for
 9:     end for
10:     return False
11: end procedure
```

We say that the running time of PairSum225 is **quadratic** in the worst case, or that the algorithm is $O(n^2)$.

Comparing Algorithms (1)

First Issue: What does 'best' mean?

- Smallest running time?
- Smallest memory requirement?
- Fewest cache misses?
- Lowest power consumption?

For most of this course, we will compare algorithms based on their **worst case running time** on large input sizes. However, the other metrics are also important in practice.

Comparing Algorithms (2)

Second Issue: How big is a 'large' input?

The constraints of the problem should dictate the meaning of 'large'. In a problem like ContainsDuplicate, there are no inherent restrictions on the input size (only the restrictions imposed by the available memory and time on the machine executing the algorithm). In this course, we will use **asymptotic analysis** to consider the worst case behavior of algorithms as the input size becomes arbitrarily large.

Comparing Algorithms (3)

Third Issue: How should worst case running time be measured?

- Timings?
- Counting clock cycles?
- Counting machine instructions?
  - On the x64 architecture?
  - On an ARM architecture?
  - On the AVR architecture?

If a worst case input can be constructed, timing an implementation can be effective to gauge the worst case performance.

However, implementing the algorithm and running it is time consuming, and as usual, timings do not necessarily generalize beyond a single machine.

Comparing Algorithms (4)

Third Issue: How should worst case running time be measured?

- Timings?
- Counting clock cycles?
- Counting machine instructions?
  - On the x64 architecture?
  - On an ARM architecture?
  - On the AVR architecture?

Counting machine instructions gives a reasonably accurate measurement of the algorithm's performance, although it is only an approximation of the real running time (since different processors can execute the same machine instruction differently).

On the other hand, machine code (and assembly language) is not well suited to understanding algorithms, and every architecture uses different instructions.

Comparing Algorithms (5)

Third Issue: How should worst case running time be measured?

- Timings?
- Counting clock cycles?
- Counting machine instructions?
  - On the x64 architecture?
  - On an ARM architecture?
  - On the AVR architecture?

Assumption: If the same algorithm is compiled for architectures A, B and C, the general performance trend will be the same across all three architectures.

Even though certain instructions on architecture A may be faster or slower than architectures B and C, the overall growth rate of the running time of the algorithm is architecture-independent.

Comparing Algorithms (6)

Third Issue: How should worst case running time be measured?

- Timings?
- Counting clock cycles?
- Counting machine instructions?
  - On the x64 architecture?
  - On an ARM architecture?
  - On the AVR architecture?

Therefore, if an algorithm requires about n^2 operations on architecture A, we can assume it will require about n^2 operations on architectures B and C.

Still, none of the architectures above are very pleasant for algorithm design.

Comparing Algorithms (7)

Idea: If the choice of architecture does not affect the overall trend of performance, why not use a hypothetical architecture which is more convenient for algorithm analysis?

Operation Counts (1)

Primitive Operations

Each primitive operation takes one unit of time.

- Arithmetic (add, subtract, multiply, divide)
- Comparisons ($<$, $\leq$, $=$, $\neq$, $\geq$, $>$)
- Boolean logic (and, or, xor)
- Assignment

Data Access

Accessing variables or fields in an object (including the length of an array) is considered to be free (and requires 0 units of time). This is because we can assume that the operation using the data (e.g. add or subtract) includes the cost of accessing it.

Operation Counts (2)

Array Allocation

Allocating an array of length k requires k operations.

Loops

The number of operations required to execute one iteration of a loop (for, while or do-while), is the sum of the costs of the condition and the loop body. The cost of a for-loop condition of the form "$i = 1, 2, \ldots, n$" is one operation.

If-Statements

Similar to loops, the cost of an if-statement is the sum of the costs of the condition and body.

Function Calls

The number of operations required to call a function F is equal to the number of operations required to run F on the provided input. Return statements inside a function require one operation.

SumArray Analysis

```
1: procedure SumArray(A)
2:     n ← Length of A
3:     sum ← 0
4:     for i = 0, 1, 2, ..., n − 1 do
5:         sum ← sum + A[i]
6:     end for
7:     return sum
8: end procedure
```

- Lines 2, 3 and 7 require one operation each.
- The loop condition on line 4 requires one operation and the loop body (line 5) requires two operations.
- The loop runs for n iterations in the worst case.
- Therefore, the algorithm requires $3n + 3$ operations.

ContainsDuplicate Analysis - Nested Loops (1)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = 0, 1, 2, ..., n − 1 do
5:              if i ≠ j and A[i] = A[j] then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

- This is the original version of the nested loop algorithm for ContainsDuplicate.
- Lines 2 and 10 each require one operation.
- In the worst case (outlined in the last lecture), the outer loop runs for n iterations.

ContainsDuplicate Analysis - Nested Loops (2)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = 0, 1, 2, ..., n − 1 do
5:              if i ≠ j and A[i] = A[j] then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

- During each iteration of the outer loop, the inner loop runs for n iterations.
- The loop body (lines 5 - 7) requires 3 operations.
- The return statement on line 6 is never executed for a worst case input.

ContainsDuplicate Analysis - Nested Loops (3)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = 0, 1, 2, ..., n − 1 do
5:              if i ≠ j and A[i] = A[j] then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

- Cost of each instance of the inner loop: $n(1 + 3) = 4n$.
- Cost of the outer loop: $n(1 + 4n) = n + 4n^2$.
- Total operation count: $4n^2 + n + 2$.

ContainsDuplicate Analysis - Nested Loops (Improved) (1)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = i + 1, i + 2, ..., n − 1 do
5:              if A[i] = A[j] then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

- This is the improved version of the nested loop algorithm.
- The body of the inner loop (lines 5 - 7) now requires 1 operation.
- How do we count the operations required by the inner loop?

ContainsDuplicate Analysis - Nested Loops (Improved) (2)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      for i = 0, 1, 2, ..., n − 1 do
4:          for j = i + 1, i + 2, ..., n − 1 do
5:              if A[i] = A[j] then
6:                  return True
7:              end if
8:          end for
9:      end for
10:     return False
11: end procedure
```

- One way to count the total operations is to add up the number of iterations required by the inner loop for each value of i in the outer loop and multiply by the total number of operations per iteration.

ContainsDuplicate Analysis - Nested Loops (Improved) (3)

Value of i	**Inner loop iterations**
0	$n-1$
1	$n-2$
2	$n-3$
$\vdots$	$\vdots$
$n-3$	2
$n-2$	1
$n-1$	0

Total iterations of inner loop:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1+0=\sum_{k=0}^{n-1} k$$

ContainsDuplicate Analysis - Nested Loops (Improved) (4)

Total iterations of inner loop:

$$(n-1)+(n-2)+(n-3)+\ldots+2+1+0=\sum_{k=0}^{n-1} k$$

Total operations for all iterations of the inner loop:

$$(1+1)\cdot\sum_{k=0}^{n-1} k = 2\sum_{k=0}^{n-1} k$$

Total operations for the entire algorithm:

$$\underbrace{2+n+2\overbrace{\sum_{k=0}^{n-1} k}^{\text{Inner loop}}}_{\text{Outer loop}}$$

(The extra n in the outer loop counts the 1 operation per iteration needed to evaluate the for-loop condition)

ContainsDuplicate Analysis - Nested Loops (Improved) (5)

Cumbersome sums may be unavoidable when analysing algorithms. However, it is often possible to find a closed form for a sum, allowing further simplification.

Theorem: For all integers $n \geq 0$,

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}.$$

Proof: By induction.

ContainsDuplicate Analysis - Nested Loops (Improved) (6)

Applying the identity from the previous slide to the operation count gives a total operation count of

$$\begin{aligned} 2 + n + 2\sum_{k=0}^{n-1} k &= 2 + n + 2\frac{(n-1)n}{2} \\ &= 2 + n + n^2 - n \\ &= n^2 + 2 \end{aligned}$$

Proofs by Induction

Theorem: For all integers $n \geq 0$,

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1.$$

Theorem: For all integers $n \geq 0$,

$$\sum_{i=1}^{n} 2i - 1 = n^2.$$

- A set of sample induction proofs on summation identities has been posted to conneX.
- We will use induction to prove other types of results later in the course (not just summation identities).
- The ability to write concise induction proofs is **critically important** in CSC 225.

ContainsDuplicate Analysis - Sort and Scan (1)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      Sort A
4:      for i = 0, 1, 2, ..., n − 2 do
5:          if A[i] = A[i + 1] then
6:              return True
7:          end if
8:      end for
9:      return False
10: end procedure
```

- This is the 'sort and scan' variant of the ContainsDuplicate algorithm.
- As before, lines 2 and 9 each require one operation.

ContainsDuplicate Analysis - Sort and Scan (2)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      Sort A
4:      for i = 0, 1, 2, ..., n − 2 do
5:          if A[i] = A[i + 1] then
6:              return True
7:          end if
8:      end for
9:      return False
10: end procedure
```

- The loop on lines 4 - 8 requires $2(n-1)$ operations in the worst case.

ContainsDuplicate Analysis - Sort and Scan (3)

```
1:  procedure ContainsDuplicate(A)
2:      n ← Length of A
3:      Sort A
4:      for i = 0, 1, 2, ..., n − 2 do
5:          if A[i] = A[i + 1] then
6:              return True
7:          end if
8:      end for
9:      return False
10: end procedure
```

- What is the cost of sorting the array A?
- In pseudocode, can write 'Sort A' without being specific, since sorting is a known and clearly defined problem and we know it can be done.

ContainsDuplicate Analysis - Sort and Scan (4)

Theorem: An array A of n integers can be sorted with at most

$$10n \log_2 n$$

operations.

- When writing code, arrays are normally sorted by calling a standard library function.
- We do not need to write a sorting algorithm to finish our analysis. For now, we will use an existing result (above) without proving it.

ContainsDuplicate Analysis - Sort and Scan (5)

Theorem: An array A of n integers can be sorted with at most

$$10n \log_2 n$$

operations.

We will study several sorting algorithms which achieve running times similar to the above later in the course.

ContainsDuplicate Analysis - Sort and Scan (6)

```
 1: procedure ContainsDuplicate(A)
 2:     n ← Length of A
 3:     Sort A
 4:     for i = 0, 1, 2, ..., n − 2 do
 5:         if A[i] = A[i + 1] then
 6:             return True
 7:         end if
 8:     end for
 9:     return False
10: end procedure
```

The total number of operations required for the sort and scan algorithm is

$$2 + 2(n-1) + 10n\log_2 n = 10n\log_2 n + 2n$$

in the worst case.

Comparing Worst Case Performance (1)

We have now derived analytic expressions for the worst case operation counts of each algorithm.

Nested Loops (Original):

$$F_1(n) = 4n^2 + n + 2$$

Nested Loops (Improved):

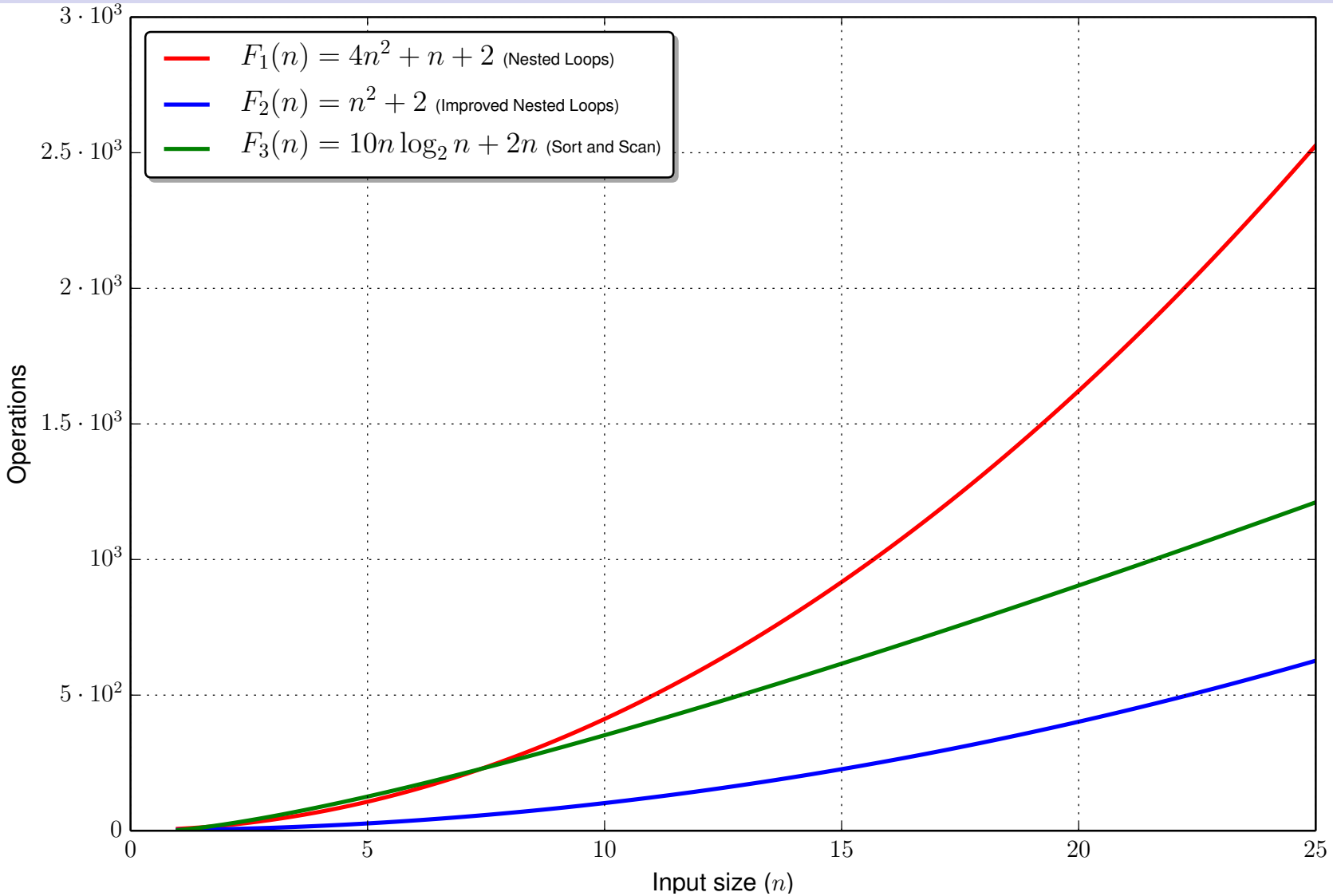
$$F_2(n) = n^2 + 2$$

Sort and Scan:

$$F_3(n) = 10n \log_2 n + 2n$$

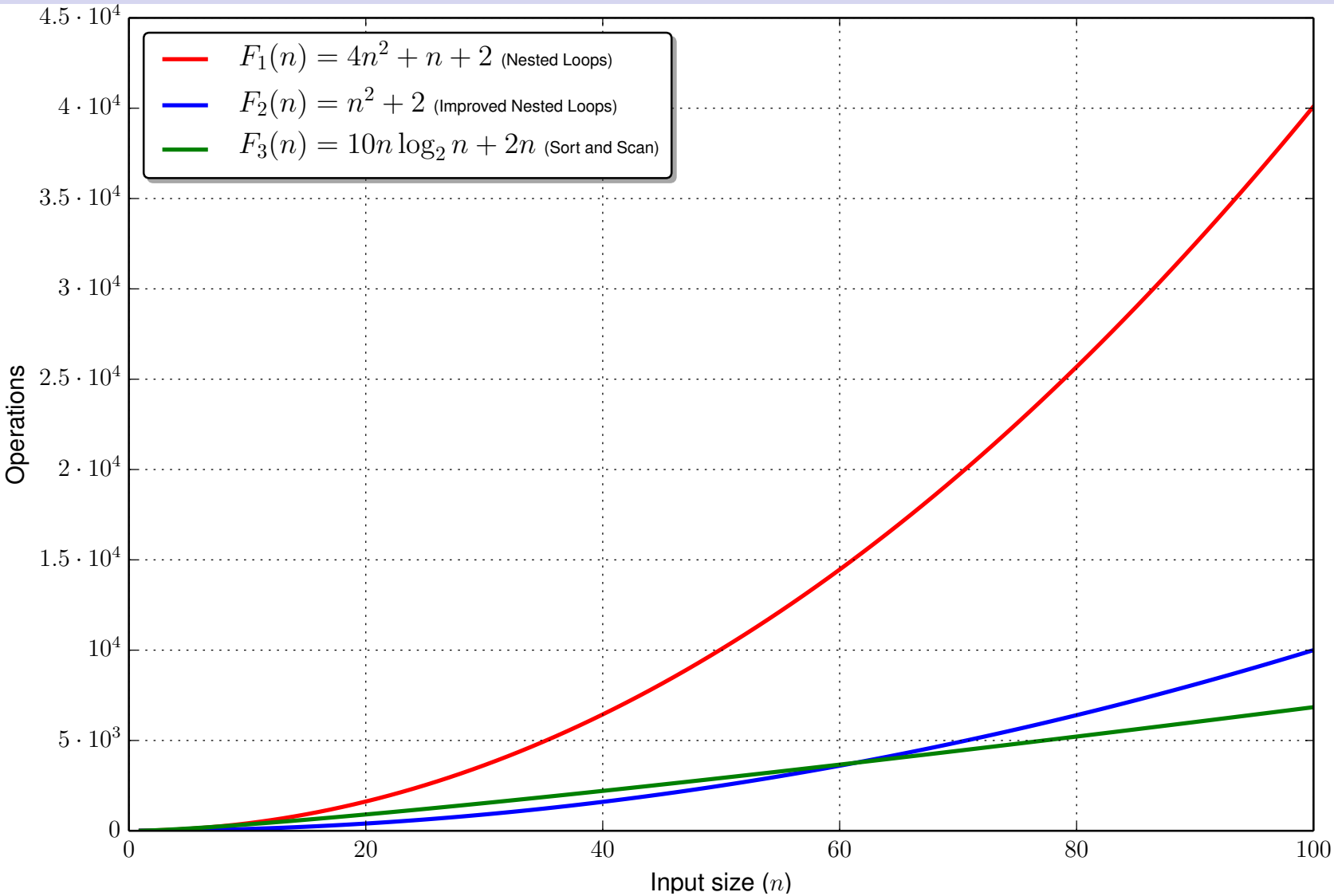
Now, at last, we can choose the 'best' algorithm based on worst-case running time.

Comparing Worst Case Performance (2)



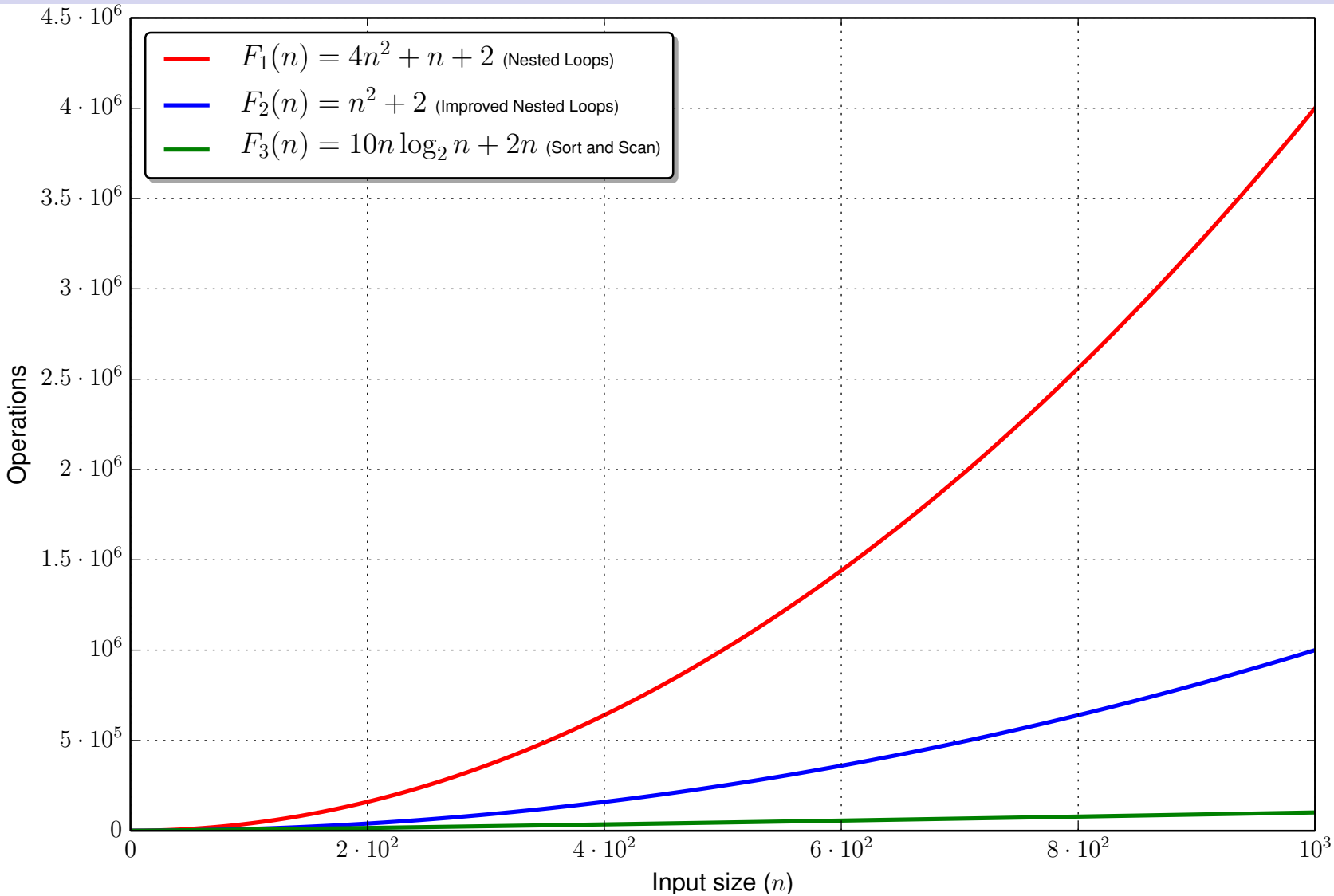
On input sizes up to $n = 25$, the improved nested loop algorithm requires fewer operations.

Comparing Worst Case Performance (3)



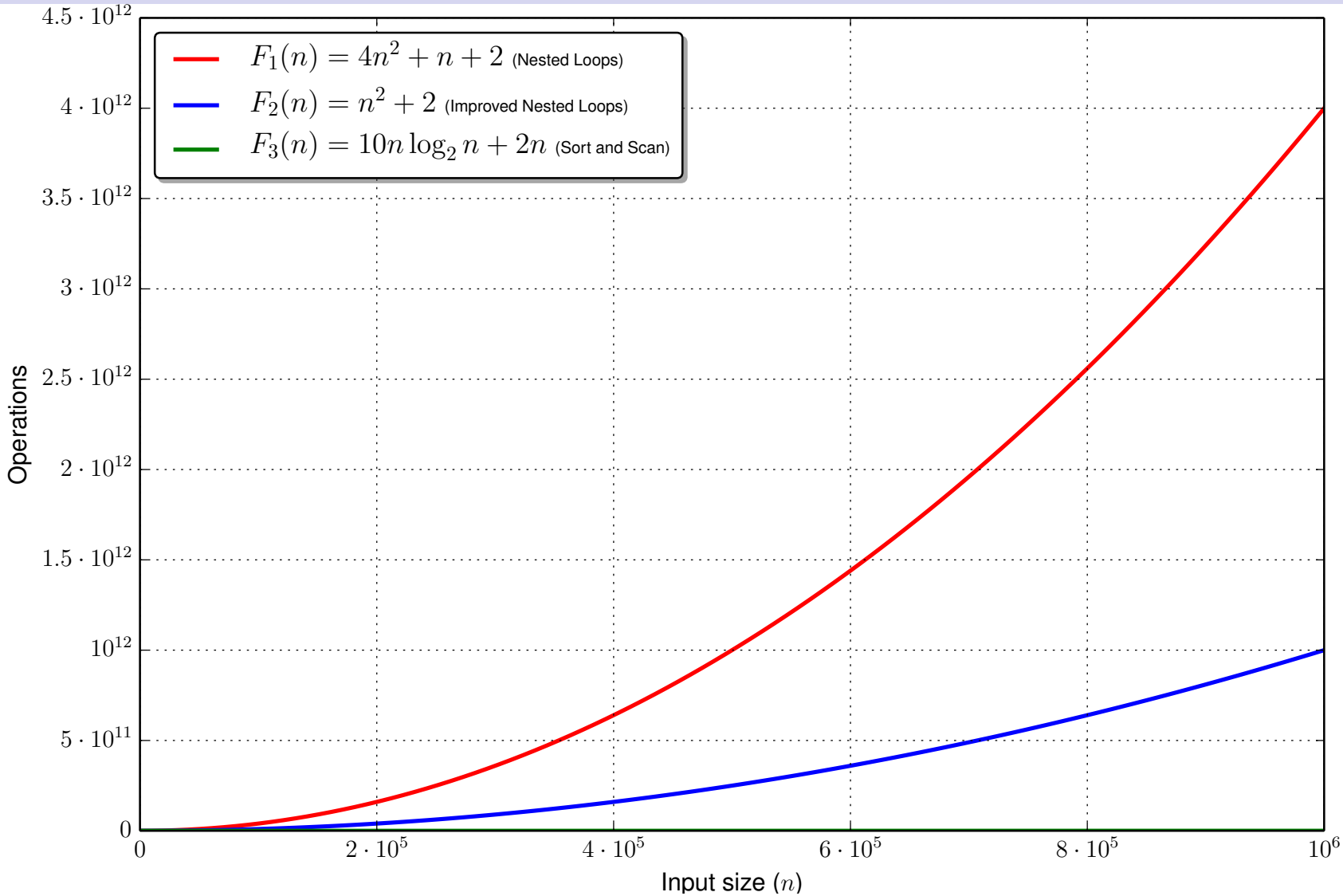
On larger input sizes (e.g. $n = 100$), the sort and scan algorithm requires fewer operations.

Comparing Worst Case Performance (4)



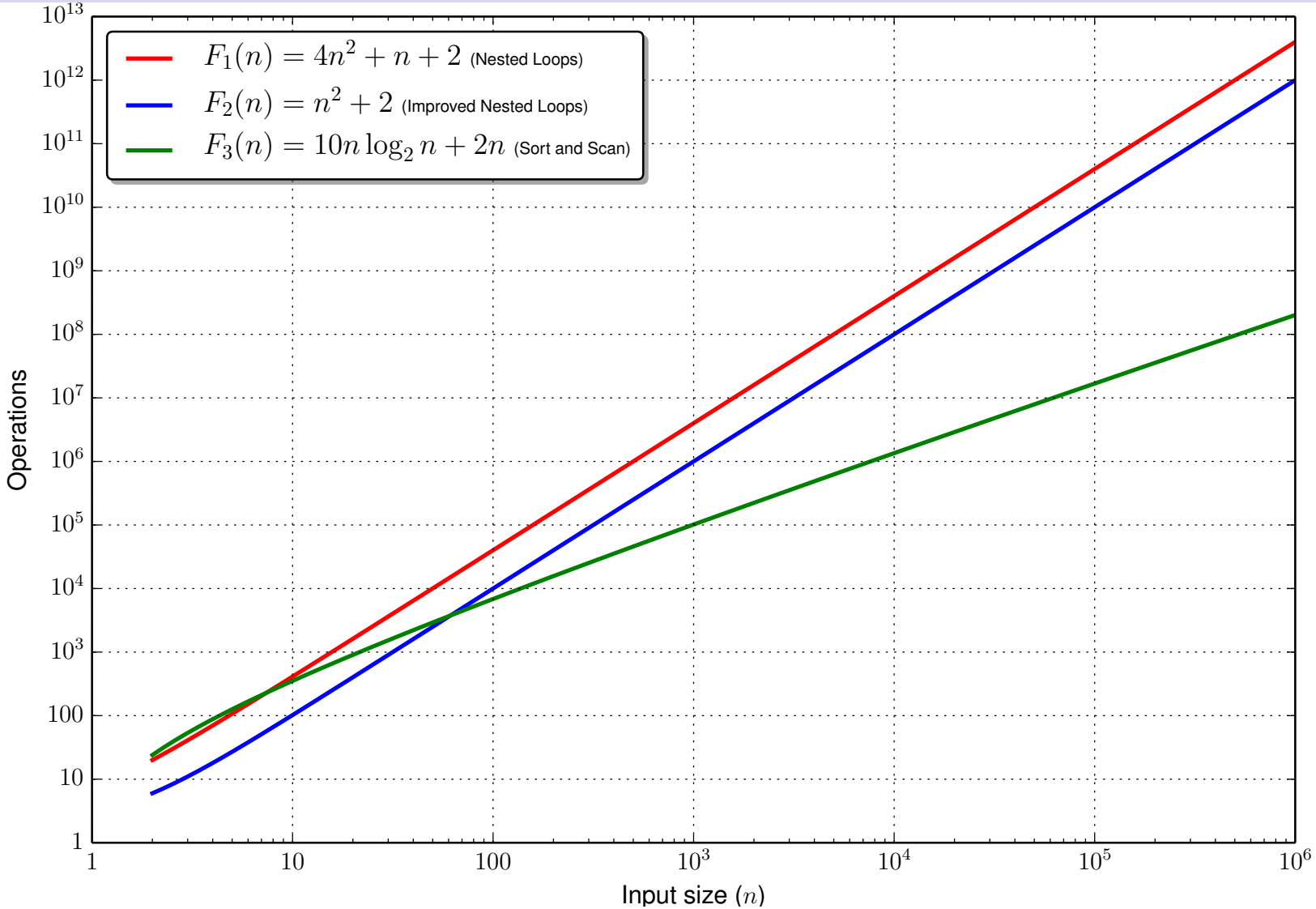
As the input size grows, the advantage of the sort and scan algorithm increases.

Comparing Worst Case Performance (5)



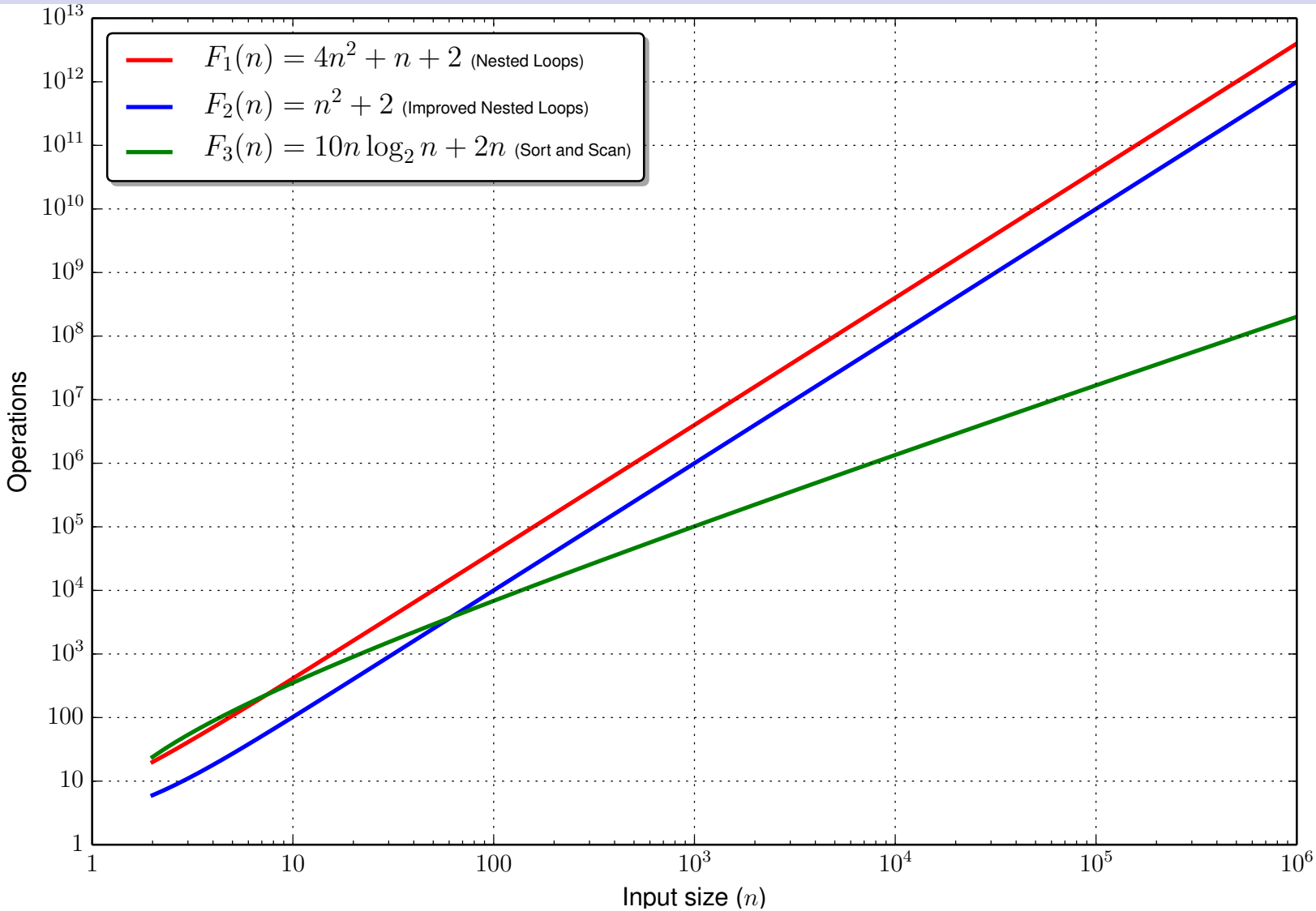
When $n = 10^6$, the curve for F_3 is no longer visible with a normal plot.

Comparing Worst Case Performance (6)



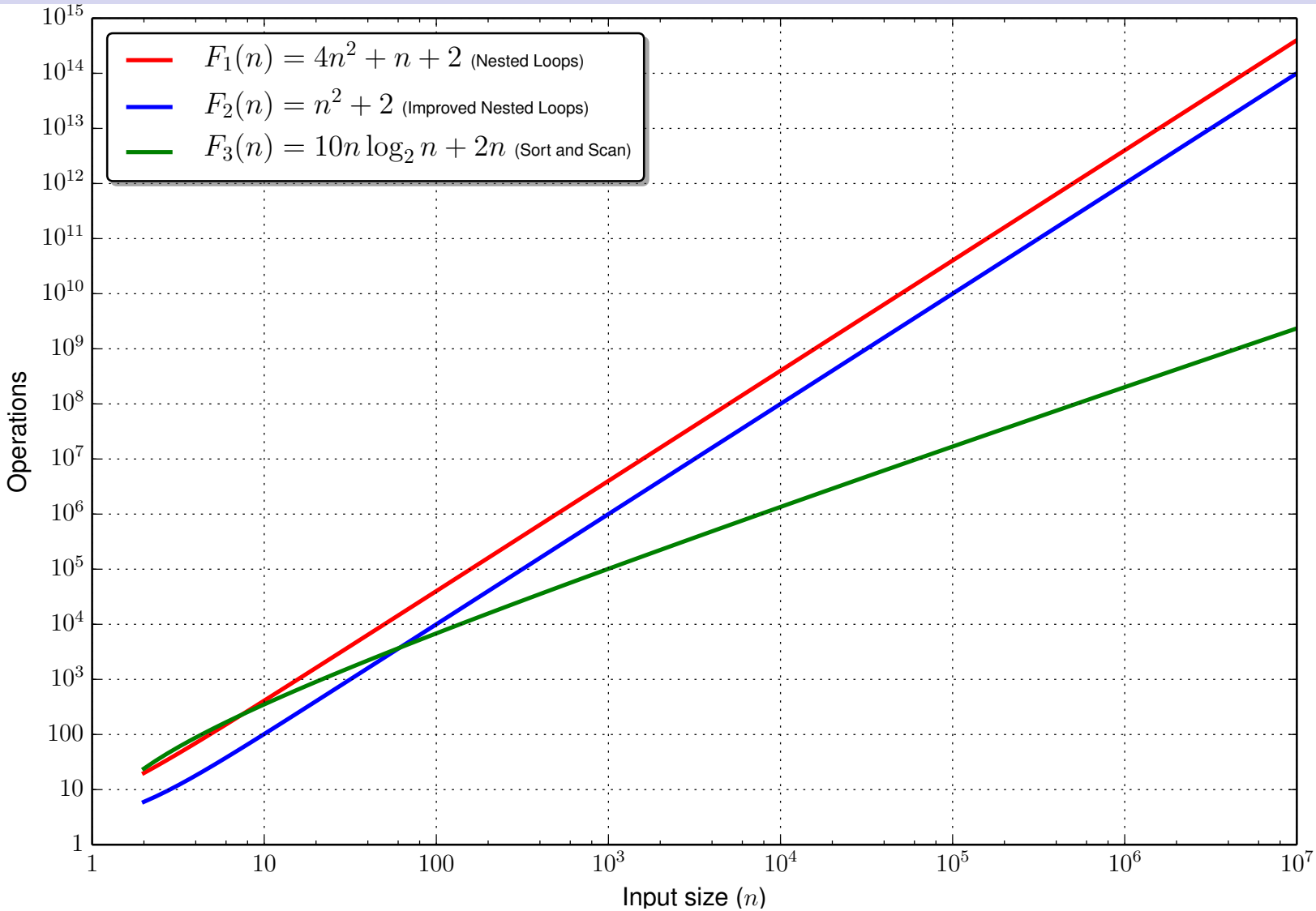
At this scale, a logarithmic plot is more illuminating.

Comparing Worst Case Performance (7)



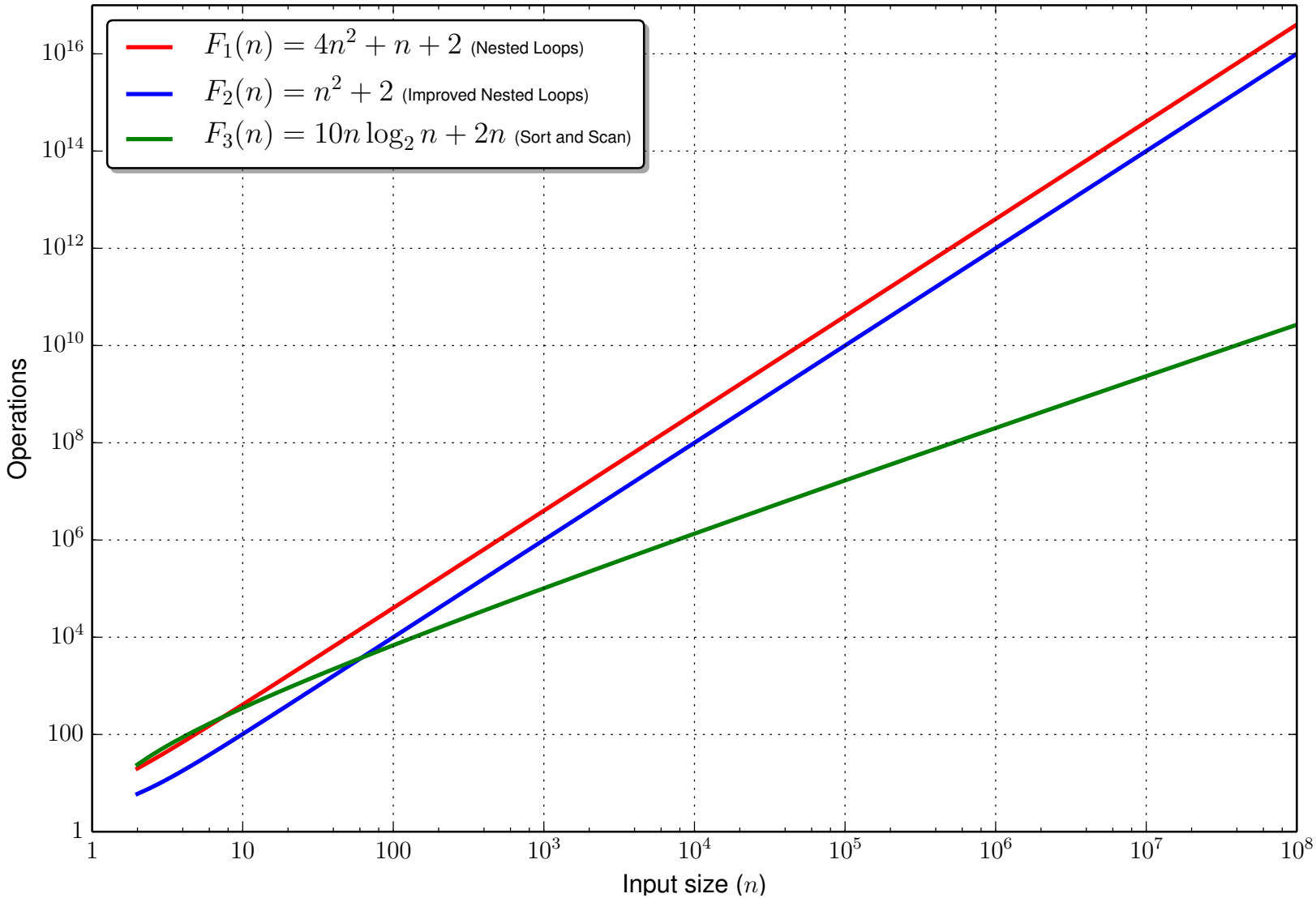
For inputs of size $n = 10^6$ (a 4mb array of type `int`), the algorithms differ by *trillions* of operations.

Comparing Worst Case Performance (8)



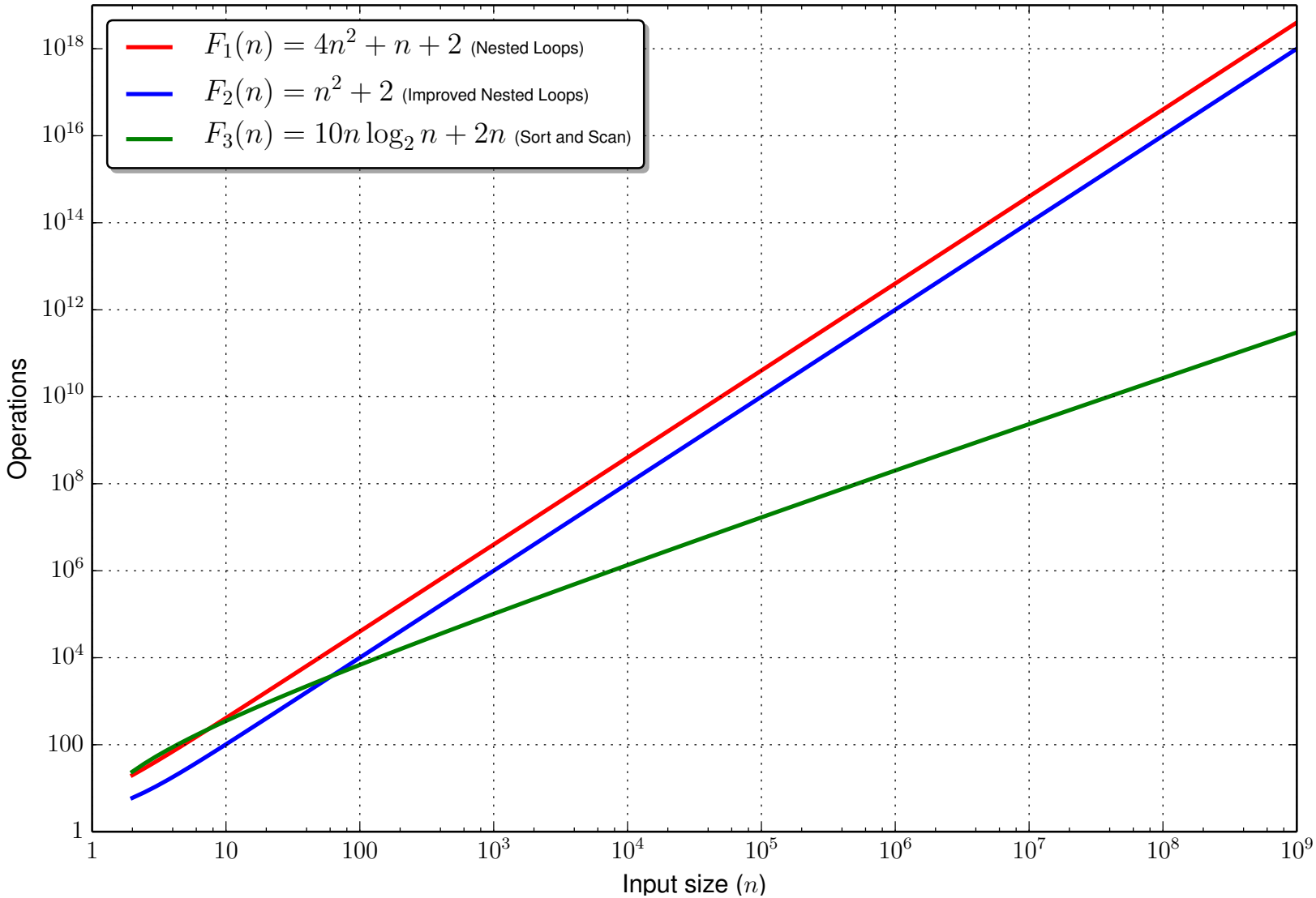
Observation: The performance difference between the two nested loop algorithms becomes less significant as n increases.

Comparing Worst Case Performance (9)



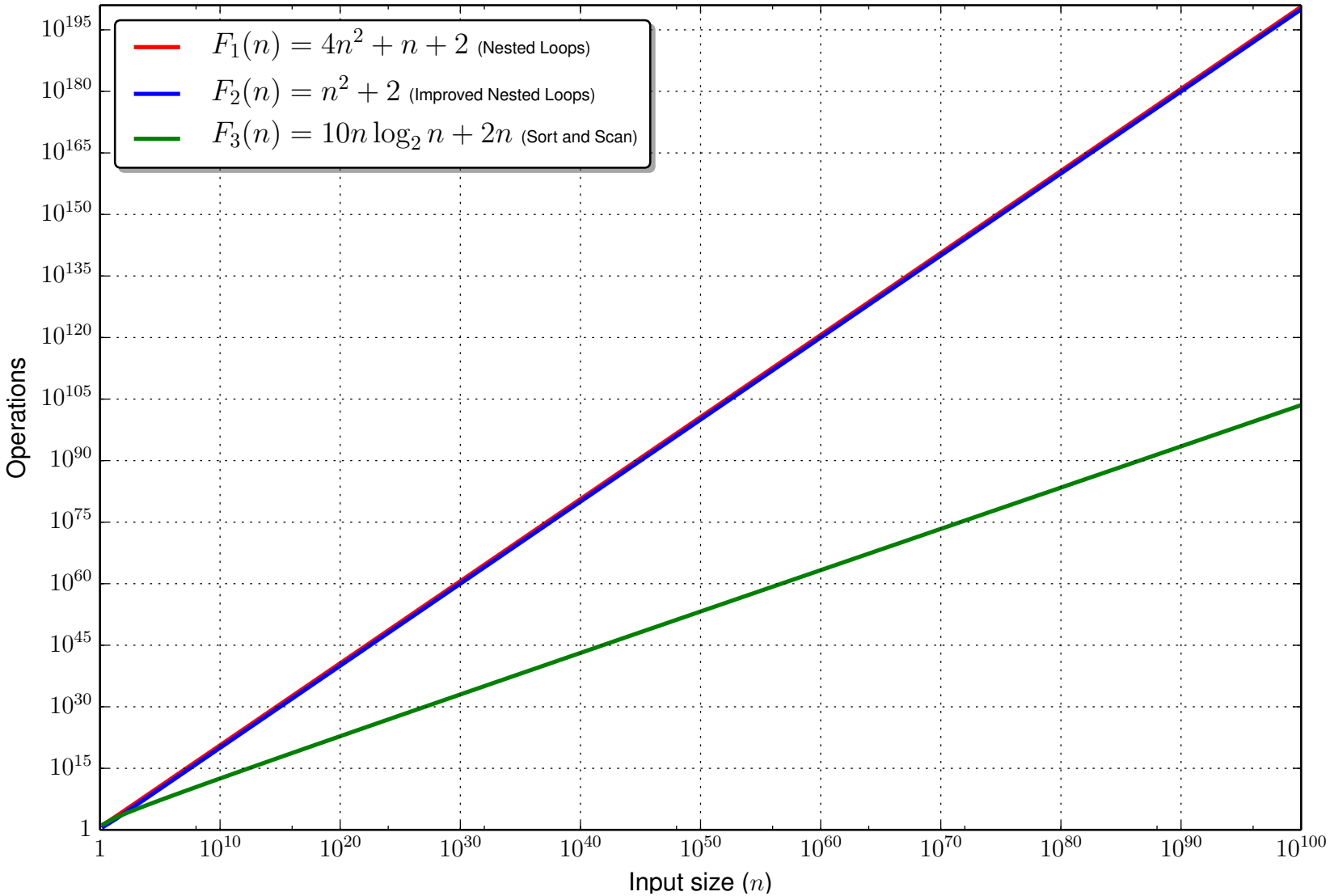
Observation: The performance difference between the two nested loop algorithms becomes less significant as n increases.

Comparing Worst Case Performance (10)



Observation: The performance difference between the two nested loop algorithms becomes less significant as n increases.

Comparing Worst Case Performance (11)



Observation: The performance difference between the two nested loop algorithms becomes less significant as n increases.