6.006	Lecture 3	Sept. 13, 2018

TODAY: Sorting (intro.)

6.006 sorting algorithms:	technique	time
– permutation sort	brute force	$O((n+1)!)$
– insertion sort	decrease & conquer	$O(n^2)$
– merge sort	divide & conquer	$O(n \lg n)$
– heap sort [L5]	data structure	$O(n \lg n)$
– AVL sort [L7]	data structure	$O(n \lg n)$
– radix sort [L8]	decr. & conquer on integers	

TODAY

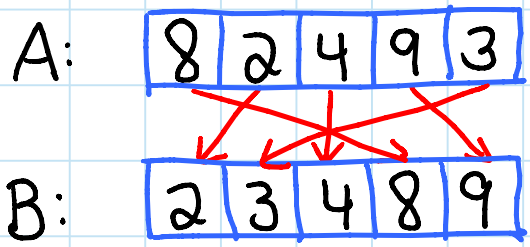
Sorting problem:

- input: array $A[0 \,..\, n-1]$ of numbers
- output: permutation B of A such that
  - ↳ array with same elements, re-ordered

$$B[0] \leq B[1] \leq \cdots \leq B[n-1]$$

- some algorithms produce new array B; others change A directly & return $B = A$
- in place if $B = A$ & use $O(1)$ extra space

Python sorted vs. list.sort



Permutation sort: brute force

(a.k.a. deterministic bogosort, slowsort, stupid sort, ...)

- for each permutation B of A:
  - if B is sorted:
    - return B

← n! choices

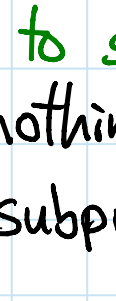
] O(n) time (also to build next perm.)

- $O(n! \cdot n) = O((n+1)!)$ time

- subproblem graph: (recursion graph)

Insertion sort: decrease & conquer

- recursively sort all but last element A[n-1]
- insert A[n-1] where it fits
  - could find location with binary search
  - but need linear time to shift elements
- base case n = 1: do nothing
- subproblem graph:



Bottom-up implementation:


```
for i in range(1, n):
    while A[i] < A[i-1]:
        swap A[i] ↔ A[i-1]
        i -= 1
```

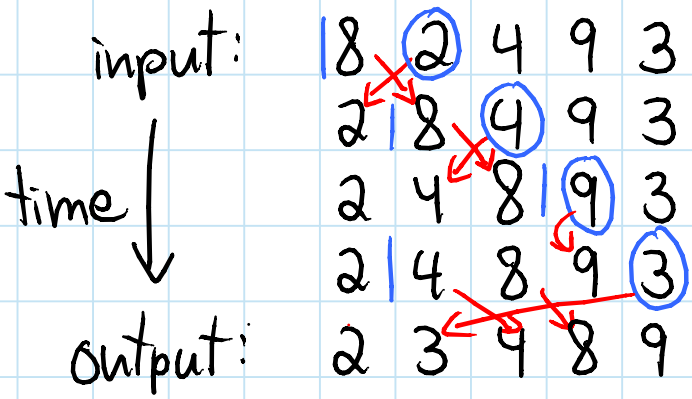
where A[-1] = -∞

Python:

```
A[i], A[i-1]
= A[i-1], A[i]
```

- in place

Example:



$T(n)$ = worst-case running time on input of size n
$= \max\{T(\text{input}) : |\text{input}| = n\}$

run-time guarantee, no assumption

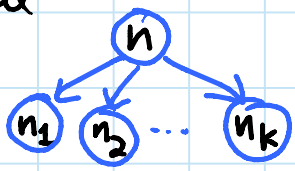
- $O(n^2)$: two nested loops of $\le n$ iterations
- $\Omega(n^2)$: e.g. reverse sorted $n-1, \ldots, 1, 0$
  $\Rightarrow i$ comparisons & swaps in step i
  $\Rightarrow \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$ total

$\exists$ case taking $\Omega(n^2)$

$\Rightarrow T(n) = \Theta(n^2)$

Divide & conquer algorithm:

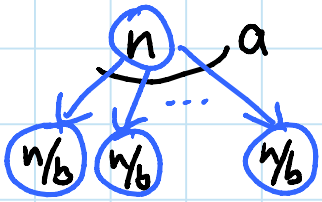
1. divide input into subproblems
2. conquer (solve) each part recursively
3. combine result(s) to solve original



Divide & conquer recurrence:

- if size-n input divided into subproblems of sizes $n_1, n_2, \ldots, n_k$
  then $T(n)$ = divide time
  $+ T(n_1) + T(n_2) + \cdots + T(n_k)$
  $+$ combine time

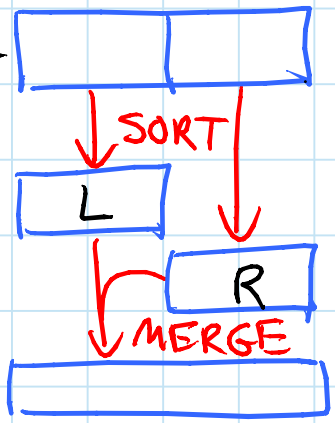
running time



- if size-n input divided into a subproblems each of size n/b:
  $T(n) = a\,T(n/b)$ + divide & combine time
  – exactly what the Master Theorem covers!

Merge sort(A): (not in place)

- if $n = 1$: return A
- recursively sort $A[:n/2] \to L$
- recursively sort $A[n/2:] \to R$
- merge L & R → output

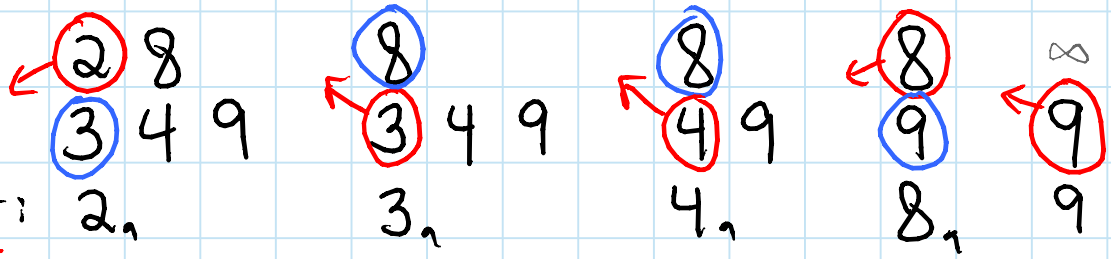


Merge(L, R): assuming L sorted & R sorted

- define $L[\text{len}(L)] = R[\text{len}(R)] = \infty$
- $l = r = 0$
- if $L[l] \le R[r]$: output $L[l]$
  $l \mathrel{+}= 1$
  else: output $R[r]$
  $r \mathrel{+}= 1$
- repeat until would output ∞

yield, or output.append

Example:



- output: 2, 3, 4, 8, 9

Time:

① divide: $\Theta(n)$ ($\Theta(1)$ via implicit splice)

② recursion: $n_1 = n_2 = n/2$
$\Rightarrow T(n_1) + T(n_2) = 2T(n/2)$

③ merge: $\Theta(1)$ time per output element
$\Theta(n)$ total

$\Rightarrow T(n) = 2T(n/2) + \Theta(n)$ MERGE SORT RECURRENCE

(same as L2's divide & conquer MSS)

Building a recursion tree: another perspective

- $T(n) = 2T(n/2) + cn$

tree represents sum of nodes

$= cn$

$T(n/2) \quad T(n/2)$

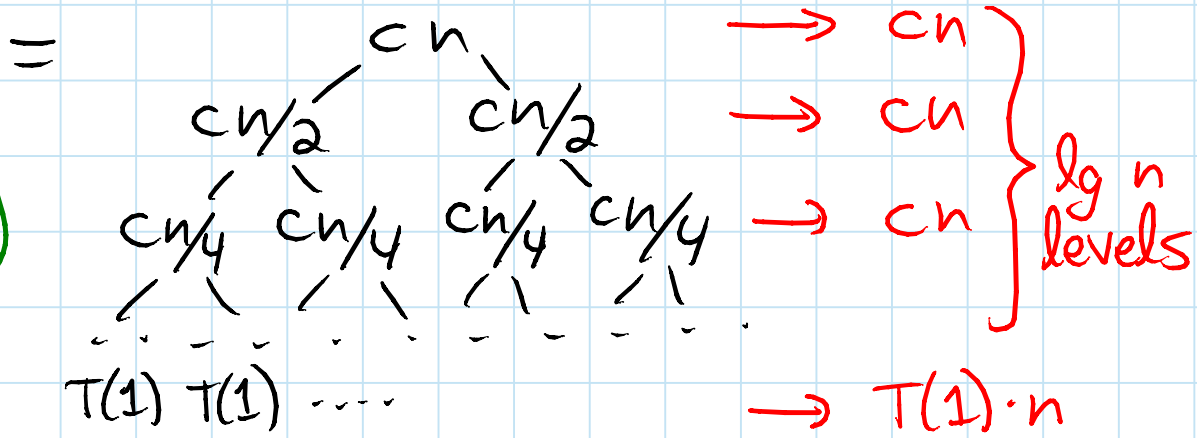
$= cn$

$cn/2 \quad cn/2$

$T(n/4) \; T(n/4) \; T(n/4) \; T(n/4)$

$= \cdots$

$=$



(same as subproblem graph)

- cool idea: sum each level, then sum these sums (as in L2/R2)

$\Rightarrow T(n) = cn \lg n + T(1) \cdot n$

$= \Theta(n \lg n)$

FUN FACT: for most divide & conquer recurrences, level sums either

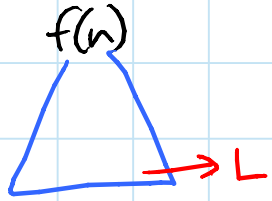
- $\Theta(\text{same}) \Rightarrow \text{total} = \Theta(\text{level sum} \cdot \#\text{levels})$ — typically a log factor
- geometrically decreasing $\Rightarrow$ total $= \Theta(\text{top level}) = \Theta(\text{root node})$
- geometrically increasing $\Rightarrow$ total $= \Theta(\text{bottom level}) = \Theta(\#\text{leaves})$ — usually

Master Theorem: [R2] formalizes some of these cases

- consider $T(n) = a\,T(n/b) + f(n)$

$\Rightarrow h = \#\text{levels} = \log_b n = \Theta(\lg n)$

& $L = \#\text{leaves} = a^h = a^{\log_b n} = n^{\log_b a}$



① $f(n) = O(L^{1-\varepsilon}) = O(n^{\log_b a - \varepsilon})$ (geometrically increasing)

$\Rightarrow T(n) = \Theta(L) = \Theta(n^{\log_b a})$

② $f(n) = \Theta(L) = \Theta(n^{\log_b a})$ (equal levels)

$\Rightarrow T(n) = \Theta(L \lg n) = \Theta(n^{\log_b a} \lg n)$

②' $f(n) = \Theta(L \lg^k n)$ (still roughly equal levels ~ extra lg factor)

$\Rightarrow T(n) = \Theta(L \lg^{k+1} n)$

③ $f(n) = \Omega(L^{1+\varepsilon}) = \Omega(n^{\log_b a + \varepsilon})$ (geometrically decreasing)

& $a f(n/b) \le (1-\delta) f(n)$ ← second level

$\Rightarrow T(n) = \Theta(f(n))$

see R2, or textbook for proof

```
import itertools

def permutation_sort(A):
    for B in itertools.permutations(A):
        if is_sorted(B):
            return B

def is_sorted(B):
    for i in range(len(B) - 1):
        if B[i] > B[i+1]:
            return False
    return True
```

```
def insertion_sort_rec(A, n = None):
    if n is None: n = len(A)
    if n == 1: return
    insertion_sort_rec(A, n-1)
    i = n-1
    while i > 0 and A[i] < A[i-1]:
        A[i], A[i-1] = A[i-1], A[i]
        i -= 1
    return A
```

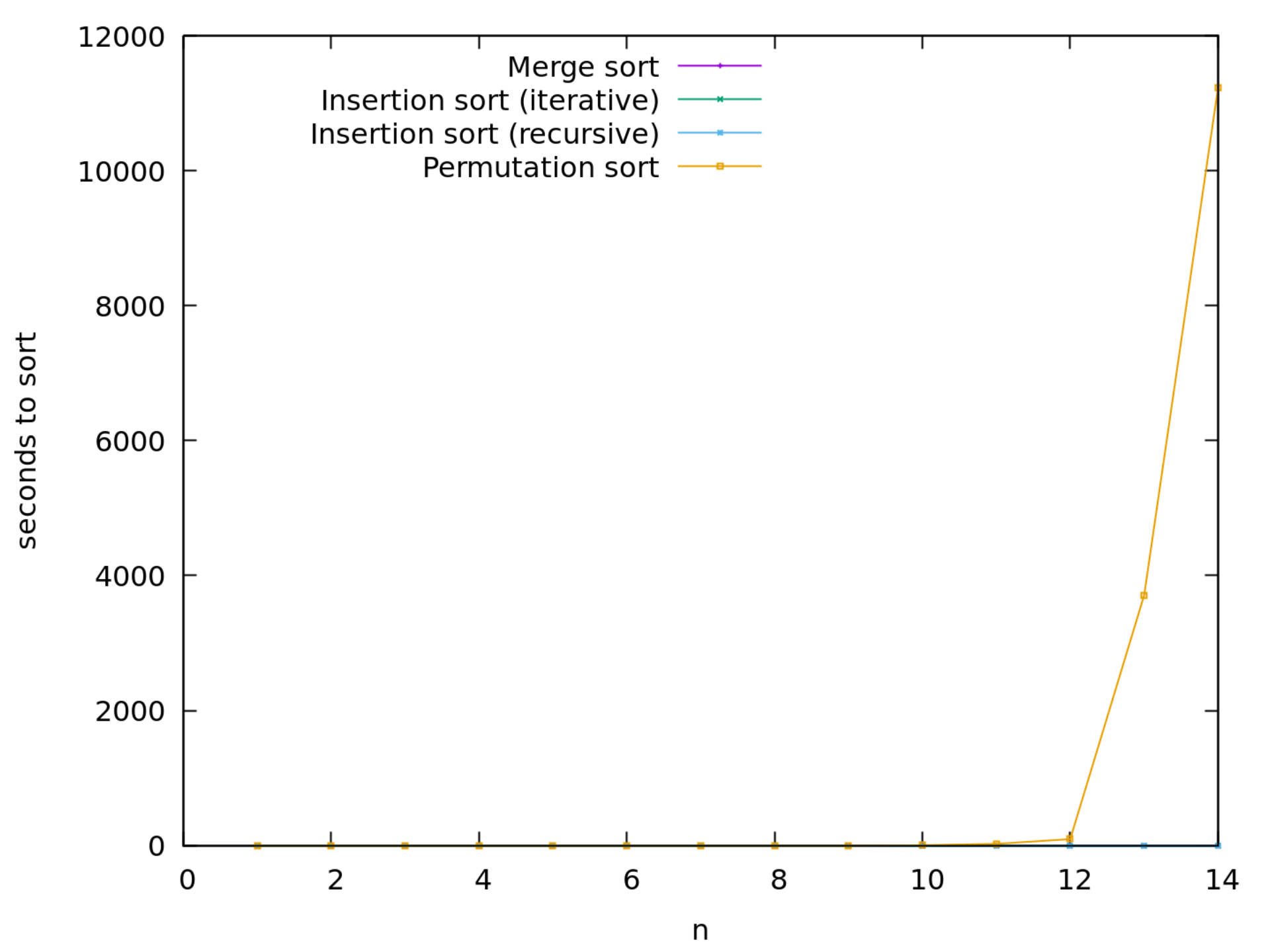
```
def insertion_sort_rec(A, n = None):
    if n is None: n = len(A)
    if n == 1: return
    insertion_sort_rec(A, n-1)
    i = n-1
    while i > 0 and A[i] < A[i-1]:
        A[i], A[i-1] = A[i-1], A[i]
        i -= 1
    return A
```

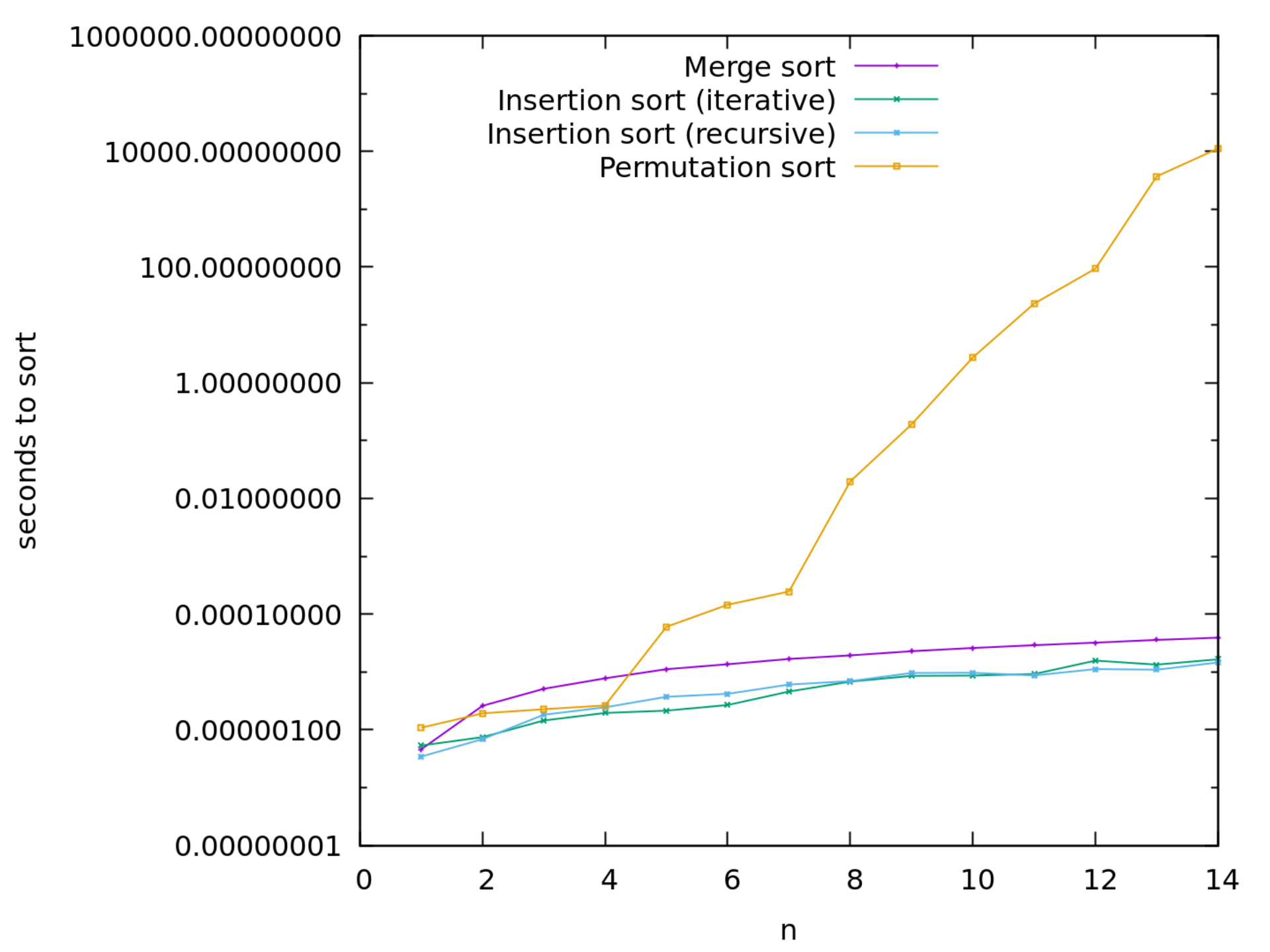
```
def insertion_sort(A):
    for i in range(1, len(A)):
        while i > 0 and A[i] < A[i-1]:
            A[i], A[i-1] = A[i-1], A[i]
            i -= 1
    return A
```

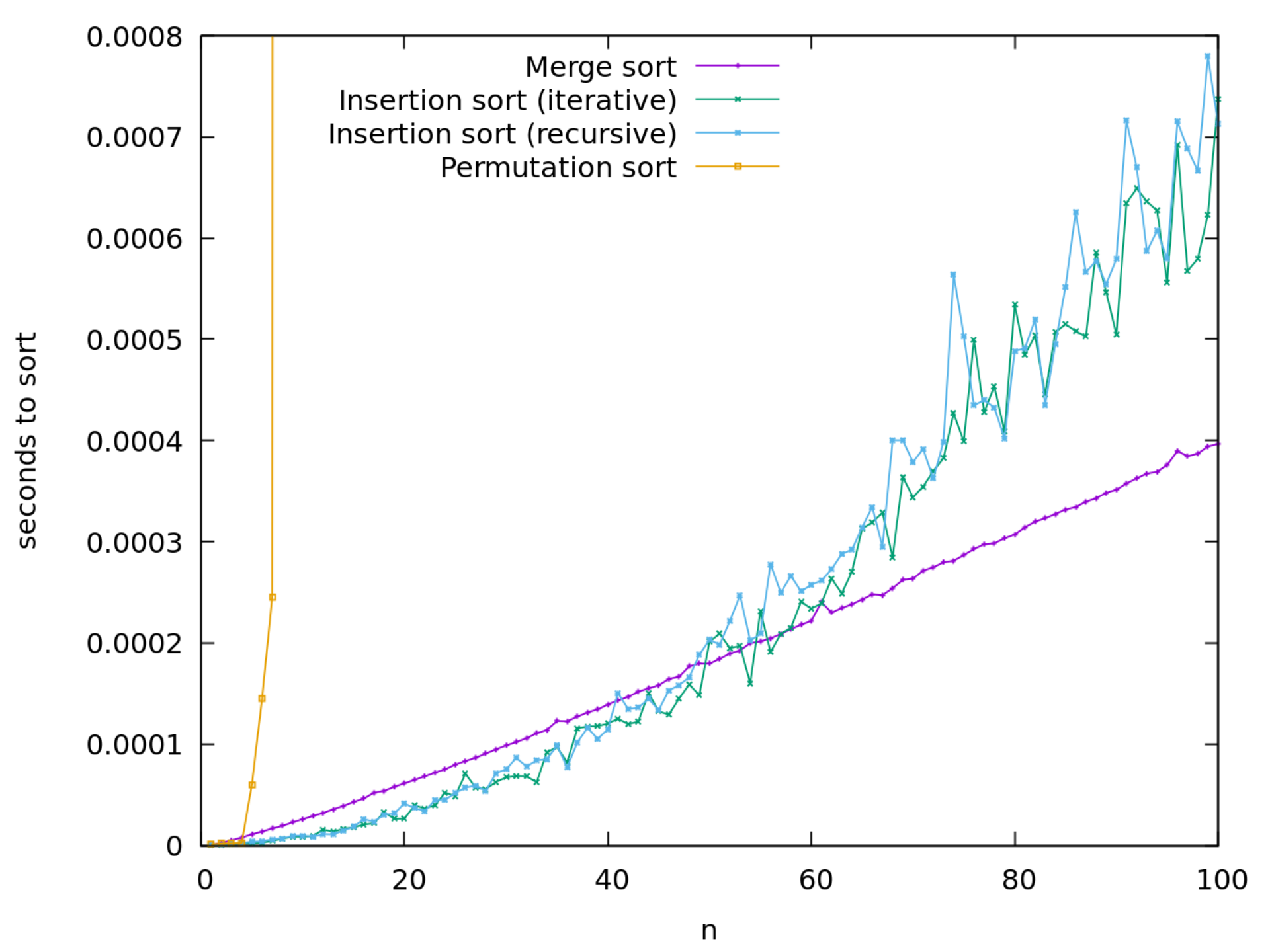
```
def merge_sort_slice(A):
    m = len(A) // 2
    L = merge_sort_slice(A[:m])
    R = merge_sort_slice(A[m:])
    return list(merge(L, R))
```

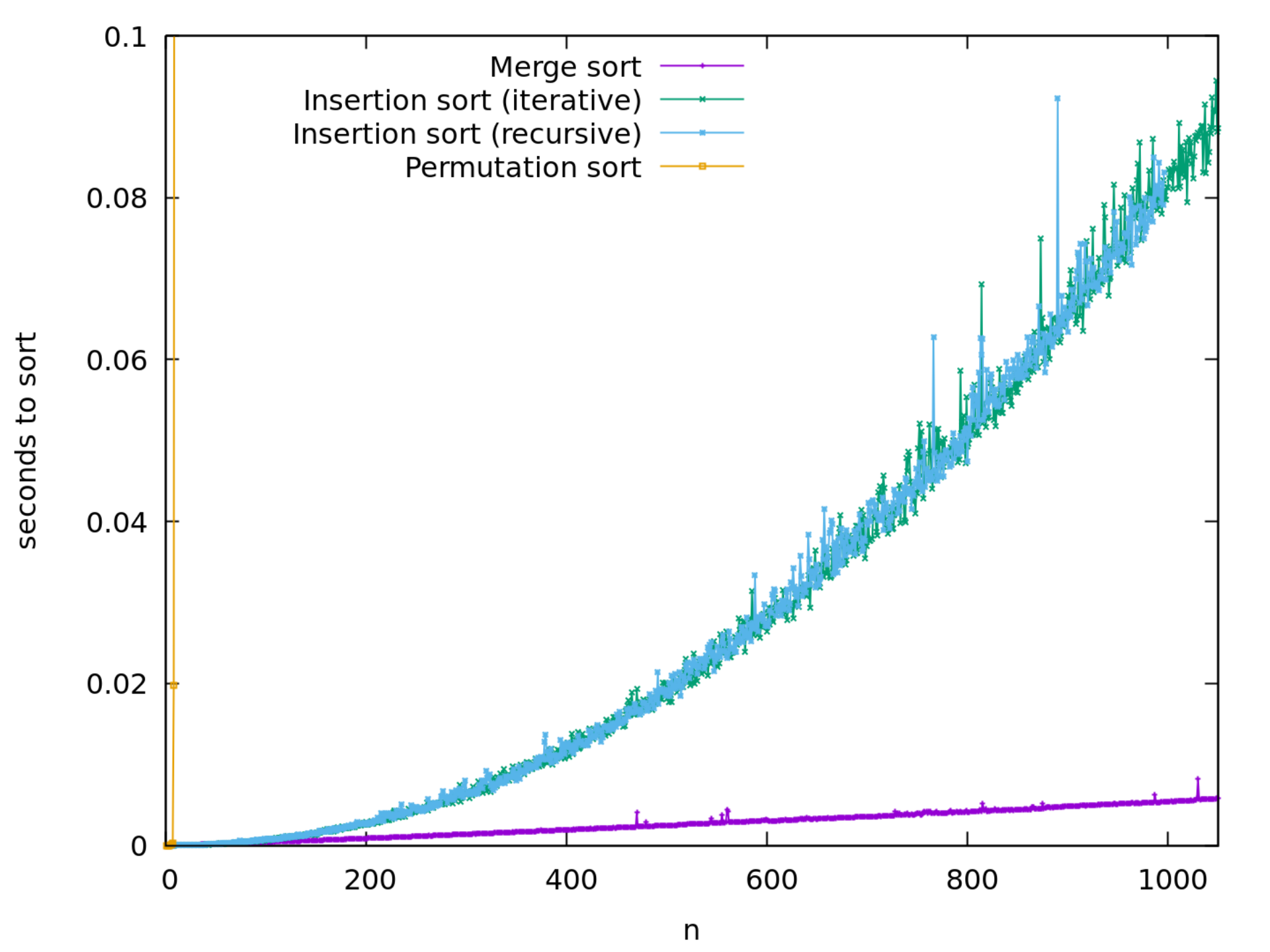
```
def merge_sort(A, i = 0, j = None):
    if j is None: j = len(A)
    if j - i == 1: return [A[i]]
    m = (i + j) // 2
    L = merge_sort(A, i, m)
    R = merge_sort(A, m, j)
    return list(merge(L, R))
```

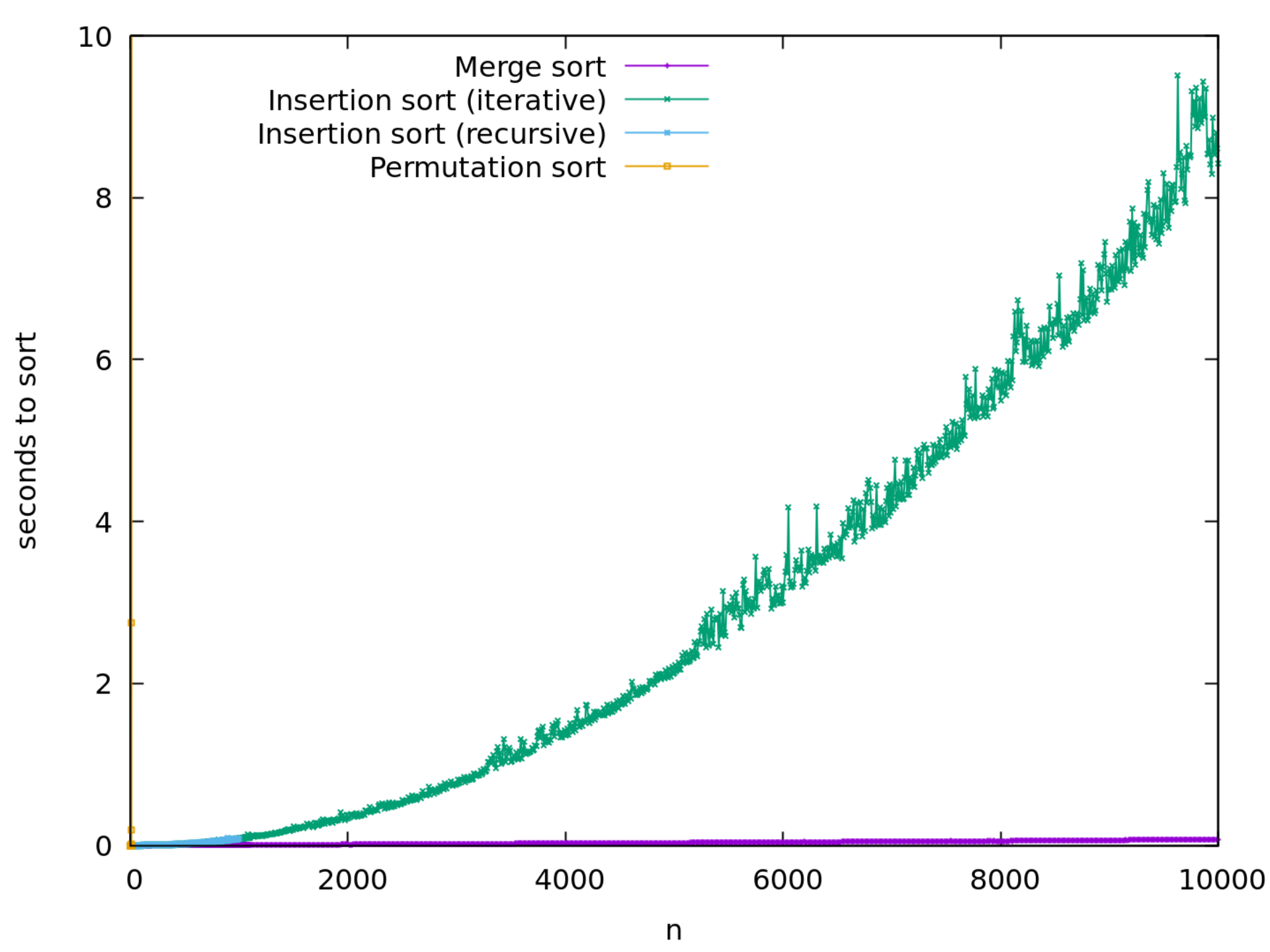
```
def merge_sort_slice(A):
    m = len(A) // 2
    L = merge_sort_slice(A[:m])
    R = merge_sort_slice(A[m:])
    return list(merge(L, R))

def merge(L, R):
   l = r = 0
   while l < len(L) or r < len(R):
      if l < len(L) and (r >= len(R) or L[l] <= R[r]):
          yield L[l]
          l += 1
      else:
          yield R[r]
          r += 1
```

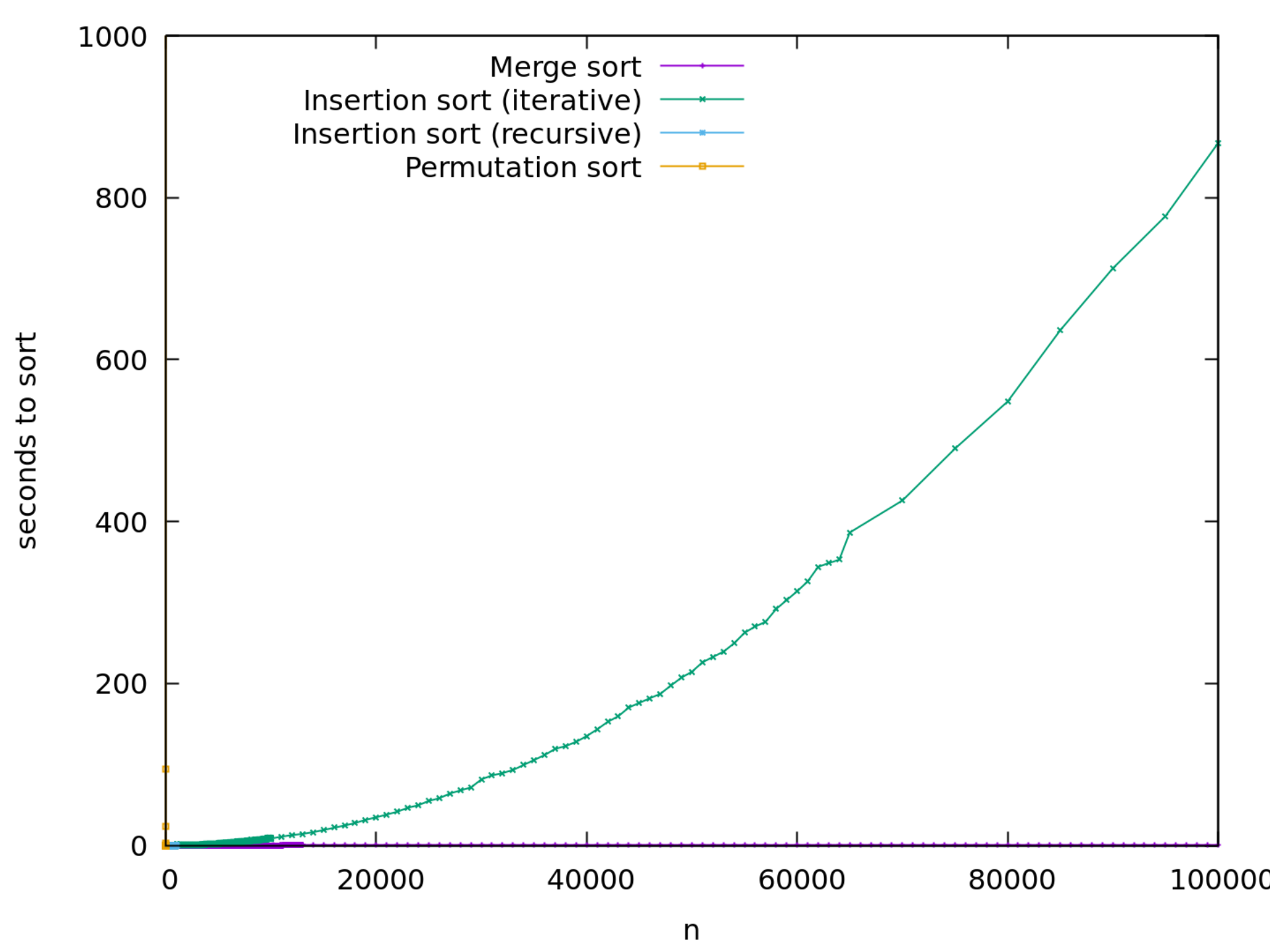

Merge sort
Insertion sort (iterative)
Insertion sort (recursive)
Permutation sort
seconds to sort
n
0
2000
4000
6000
8000
10000
12000
0
2
4
6
8
10
12
14


Merge sort
Insertion sort (iterative)
Insertion sort (recursive)
Permutation sort
1000000.00000000
10000.00000000
100.00000000
1.00000000
0.01000000
0.00010000
0.00000100
0.00000001
seconds to sort
0
2
4
6
8
10
12
14
n


Merge sort
Insertion sort (iterative)
Insertion sort (recursive)
Permutation sort
seconds to sort
n
0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0
0
20
40
60
80
100


Merge sort
Insertion sort (iterative)
Insertion sort (recursive)
Permutation sort
seconds to sort
n
0.1
0.08
0.06
0.04
0.02
0
0
200
400
600
800
1000


Merge sort
Insertion sort (iterative)
Insertion sort (recursive)
Permutation sort
seconds to sort
n
0
2
4
6
8
10
0
2000
4000
6000
8000
10000


Merge sort
Insertion sort (iterative)
Insertion sort (recursive)
Permutation sort
seconds to sort
n
0
200
400
600
800
1000
0
20000
40000
60000
80000
100000