# Sorting

Fall 2020

School of Software Engineering

South China University of Technology

# contents

- Introduction
- $\Theta(n^2)$ Sorting Algorithms
- $O(n\log n)$ Sorting Algorithms
- Linear-Time Sorts
- External sorting

# Sorting

- Sorting is a central problem in computer science
  - It has been studied intensively
  - Many algorithms have been designed
  - New algorithms are still being developed

# Terminology and Notation

- Sorting problem
  - Given a set of records $r_1$ , $r_2$ , … , $r_n$ with key values $k_1$ , $k_2$ ,… , $k_n$,
  - Arrange the records into any order *s* such that records $r_{s1}$, $r_{s2}$, …, $r_{sn}$ have keys obeying the property $k_{s1} \leq k_{s2} \leq \ldots \leq k_{sn}$.
    - Key value can or cannot have duplicate values depending on the application requirements

# Terminology and Notation

- comparison-based sorting
  - Inputs:
    - A collection of records stored in an array A
    - Each record has a key field
    - a comparison function which imposes a consistent ordering on the keys
  - Output
    - reorganize the elements of A such that
      - For any i and j, if $i < j$ then $A[i] < A[j]$

# Terminology and Notation

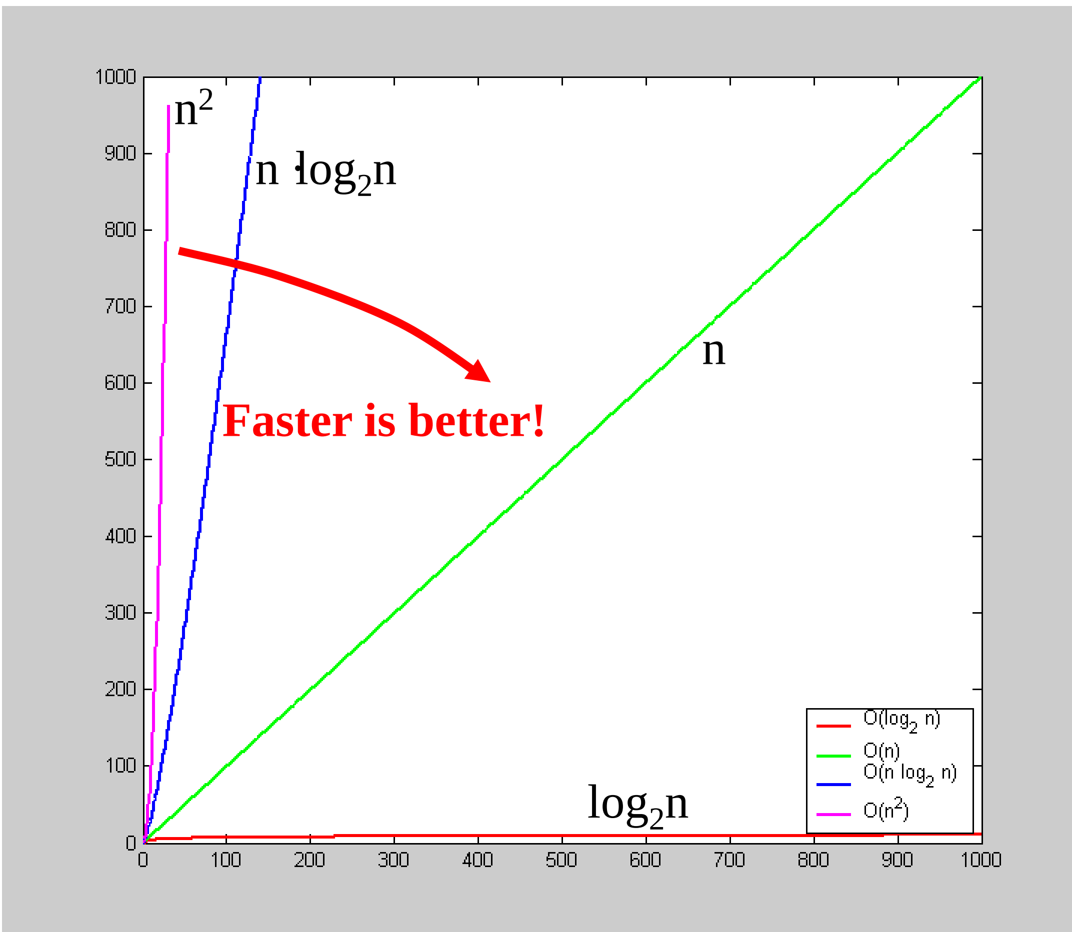
- Internal sorting
- External sorting

# Space

- How much space does the sorting algorithm require in order to sort the collection of items?
  - Is copying needed? O(n) additional space
  - In-place sorting – no copying – O(1) additional space
  - Somewhere in between for “temporary”, e.g. O(logn) space
  - External memory sorting – data so large that does not fit in memory

# Time

- How fast is the algorithm?
  - The definition of a sorted array A says that for any $i<j$, $A[i] < A[j]$
  - This means that you need to at least check on each element at the very minimum, i.e., at least $O(N)$
  - And you could end up checking each element against every other element, which is $O(N^2)$
  - The big question is: How close to $O(N)$ can you get?



n2
n log2n
n
log2n
Faster is better!
O(log2 n)
O(n)
O(n log2 n)
O(n2)
0
100
200
300
400
500
600
700
800
900
1000

# Stability

- Stability: Does it rearrange the order of input data records which have the same key value (duplicates)?
  - e.g. Phone book sorted by name. Now sort by county – is the list still sorted by name within each county?
  - Extremely important property for databases
  - A stable sorting algorithm is one which does not rearrange the order of duplicate keys

- Given 5 records $r_1(7)$, $r_2(5)$, $r_3(8)$, $r_4(3)$, and $r_5(7)$ to be sorted, which of the following output is generated by a stable sorting algorithm?
  - Case 1: $r_4(3)$ $r_2(5)$ $r_1(7)$ $r_5(7)$ $r_3(8)$
  - Case 2: $r_4(3)$ $r_2(5)$ $r_5(7)$ $r_1(7)$ $r_3(8)$

# $\Theta(n^2)$ Sorting Algorithms

# Bubble Sort

- "Bubble" elements to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
  - Bubble every element towards its correct position
    - last position has the largest element
    - then bubble every element except the last one towards its correct position
    - then repeat until done

# Bubble Sort

```
/* Bubblesort
 * At the ith iteration, it keeps bubbling up the ith
 *  smallest value to position i in the array.
*/
template <typename E, typename Comp>
void bubsort(E A[], int n) {
  for (int i=0; i<n-1; i++)
    for (int j=n-1; j>i; j--)
      if (Comp::prior(A[j], A[j-1])) //compare
        swap(A, j, j-1); //swap
}
```

# Bubblesort

n=8

| | |
|---|---|
| 0 | **42** |
| 1 | **20** |
| 2 | **17** |
| 3 | **13** |
| 4 | **28** |
| 5 | **14** |
| 6 | **23** |
| 7 | **15** |

i=0

| | | | | | | |
|---|---|---|---|---|---|---|
| i=1 | j=7 | j=6 | j=5 | j=4 | j=3 | j=2 |
| i=2 | j=7 | j=6 | j=5 | j=4 | j=3 | |
| i=3 | j=7 | j=6 | j=5 | j=4 | | |
| i=4 | j=7 | j=6 | j=5 | | | |
| i=5 | j=7 | j=6 | | | | |
| i=6 | j=7 | | | | | |

# Bubble Sort

- "Bubble" elements to to their proper place in the array by comparing elements i and i+1, and swapping if A[i] > A[i+1]
- We bubblize for i=1 to n (i.e, n times)
- Each bubblization is a loop that makes n-i comparisons.
  - The total number of comparisons will be

$$\sum_{i=2}^{n} i \approx n^2/2 = \Theta(n^2).$$

roughly the same in the best, average, and worst cases.

- The number of swaps can be expected about half the comparisons in the average case, leading to O ($n^2$), in the average and worst cases. 0 swaps in the best case.

- Bubble Sort is O($n^2$)

# Selection Sort

- In the *i*th pass of selection sort, the *i*th smallest key in the array is selected and placed into position i.
  - It searches through the entire unsorted portion to find the next smallest key value;
  - Only require one swap to put the record in place
    - The total number of swaps required will be n-1.

- It is similar to Bubble Sort but requires much fewer swaps.

# Selection Sort

```
/*
*  Selection Sort
*  In the ith pass, the ith smallest key in the array is
  selected * and placed into position i
*/
template <typename E, typename Comp>
void selsort(E A[], int n) {
  for (int i=0; i<n-1; i++) { //select ith record
    int lowindex = i; //Remember its index
    for (int j=n-1; j>i; j--) //Find least value
      if (Comp::lt(A[j], A[lowindex]))
        lowindex = j; // Put it in place
    swap(A, i, lowindex);
  }
}
```

# Selection Sort

| | i=0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 42 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 20 | 20 | 14 | 14 | 14 | 14 | 14 | 14 |
| 17 | 17 | 17 | 15 | 15 | 15 | 15 | 15 |
| 13 | 42 | 42 | 42 | 17 | 17 | 17 | 17 |
| 28 | 28 | 28 | 28 | 28 | 20 | 20 | 20 |
| 14 | 14 | 20 | 20 | 20 | 28 | 23 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 28 | 28 |
| 15 | 15 | 15 | 17 | 42 | 42 | 42 | 42 |

# Selection Sort

- Time complexity analysis
  - The number of comparisons is $\boldsymbol{\Theta(n^2)}$ in the best, worst, and average cases

  - The number of swaps is
    - 0 in the best case
    - n-1 in the worst case
    - $\boldsymbol{\Theta(n)}$ in the average case

# Insertion Sort

- What if first *k* elements of array are already sorted?
  - 4, 7, 12, 5, 19, 16
- We can shift the tail of the sorted elements list down and then *insert* next element into proper position and we get k+1 sorted elements
  - 4, 5, 7, 12, 19, 16

# Insertion Sort

```
/*
* Insertion sort
* Each record is inserted in turn at the correct position
* within a sorted list composed of those records already
* processed
*/
template <typename E, typename Comp>
void insertionSort(E A[], int n) {
  for (int i=1; i<n; i++) //insert i'th record
    for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
      swap(A, j, j-1);
}
```

# Insertion Sort

- Input: an array of 8 records with key values
  42 20 17 13 28 14 23 15

| | i=1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 42 | 20 | 17 | 13 | 13 | 13 | 13 | 13 |
| 20 | 42 | 20 | 17 | 17 | 14 | 14 | 14 |
| 17 | 17 | 42 | 20 | 20 | 17 | 17 | 15 |
| 13 | 13 | 13 | 42 | 28 | 20 | 20 | 17 |
| 28 | 28 | 28 | 28 | 42 | 28 | 23 | 20 |
| 14 | 14 | 14 | 14 | 14 | 42 | 28 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 42 | 28 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 42 |

# Insertion Sort

- Time complexity analysis
  - Two nested **`for`** loops
    - Outer **`for`** loop executes n-1 times
    - Inner **`for`** loop: depends on the number of keys in positions 1 to i-1 that are smaller than the key in position i
  - Worst case: the input records are initially arranged in the reverse of the sorted order
    - The number comparisons is $\Sigma_{i=2}^{n} i \approx n^2/2 = \Theta(n^2)$
  - Best case: the input records are already in sorted order
    - Every pass through the inner for loop fails immediately
    - The number comparisons is $n\text{-}1 = \Theta(n)$

# Insertion Sort

- The number of comparisons and swaps is determined by the number of inversions in the input records
  - Inversion: a value is greater than a given value and also occurs prior to it in the array
- We expect on average that half of the keys in the first i-1 array positions will have a value greater than that of the key at position i
  - The average-case cost is about half of the worst-case cost, i.e., around $\mathbf{n^2/4=\Theta(\mathit{n}^2)}$

# Insertion Sort

- # swaps vs. # comparisons
  - Every time through the inner for loop yields both a comparison and a swap, except that last which has no swap.
  - #swaps = #comparisons – (n-1)

- #swaps is
  - **0** in the best case
  - **$\Theta(n^2)$** in the average and worst cases

# Insertion Sort Characteristics

- In place and Stable
- Running time
  - Worst case is $O(N^2)$
    - reverse order input
    - must copy every element every time
- Good sorting algorithm for almost sorted data
  - Each item is close to where it belongs in sorted order.

# The Cost of Exchange Sorting

- Comparison of the three algorithms

| | Insertion | Bubble | Selection |
|---|---|---|---|
| **Comparisons:** | | | |
| Best Case | $\Theta(n)$ | Q($n^2$) | Q($n^2$) |
| Average Case | Q($n^2$) | Q($n^2$) | Q($n^2$) |
| Worst Case | Q($n^2$) | Q($n^2$) | Q($n^2$) |
| **Swaps:** | | | |
| Best Case | 0 | 0 | $\Theta(n)$ |
| Average Case | Q($n^2$) | Q($n^2$) | $\Theta(n)$ |
| Worst Case | Q($n^2$) | Q($n^2$) | $\Theta(n)$ |

# The Cost of Exchange Sorting

- The crucial bottleneck of the three algorithm is that only adjacent records are compared.
  - Comparisons and moves are by single steps.
  - Swapping happens between adjacent records – exchange sorts.
- What is the average number of exchanges required when sorting a list L?
  - An inversion in an array of numbers is any ordered pair (i, j) having the property that $i < j$ but $a[i] > a[j]$.
  - The average number of inversions in an array of N distinct elements is $n(n-1)/4$.
    - Define $L_R$ to be the inverse of L. There are **$n(n-1)/2$** distinct pairs of values in L ($L_R$)
    - For each pair, it must either be an inversion in L or in $L_R$.
    - The total number of inversions in L and $L_R$ is **$n(n-1)/2$** for an average of **$n(n-1)/4$** per list.
- Any algorithm that sorts by exchanging adjacent elements requires $\Omega(n^2)$ time on average.

# Shellsort

- Shellsort makes comparisons and swaps between non-adjacent elements.

- It tries to make the list “mostly sorted” so that a final insertion sort can finish the job.
  - Better performance than $\Theta(n^2)$ in the worst case.

- Central idea: divide and conquer
  - Break the list into sublists
  - Sort sublists individually
  - Recombine the sublists

# Shellsort

- Shellsort is also calld as diminishing increment sort.
- Increment sequence $h_1, h_2, \ldots, h_t$
  - After a phase, using some increment $h_k$, for every i, we have $a[i] \leq a[i + h_k]$
  - All elements spaced *hk* apart are sorted, $\boldsymbol{h_k}$**-sorted**
  - An $h_k$-sorted file that is then $h_{k-1}$-sorted remains $h_k$-sorted

| Original | 81 | 94 | 11 | 96 | 12 | 35 | 17 | 95 | 28 | 58 | 41 | 75 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| After 5-sort | 35 | 17 | 11 | 28 | 12 | 41 | 75 | 15 | 96 | 58 | 81 | 94 | 95 |
| After 3-sort | 28 | 12 | 11 | 35 | 15 | 41 | 58 | 17 | 94 | 75 | 81 | 96 | 95 |
| After 1-sort | 11 | 12 | 15 | 17 | 28 | 35 | 41 | 58 | 75 | 81 | 94 | 95 | 96 |

# Shellsort

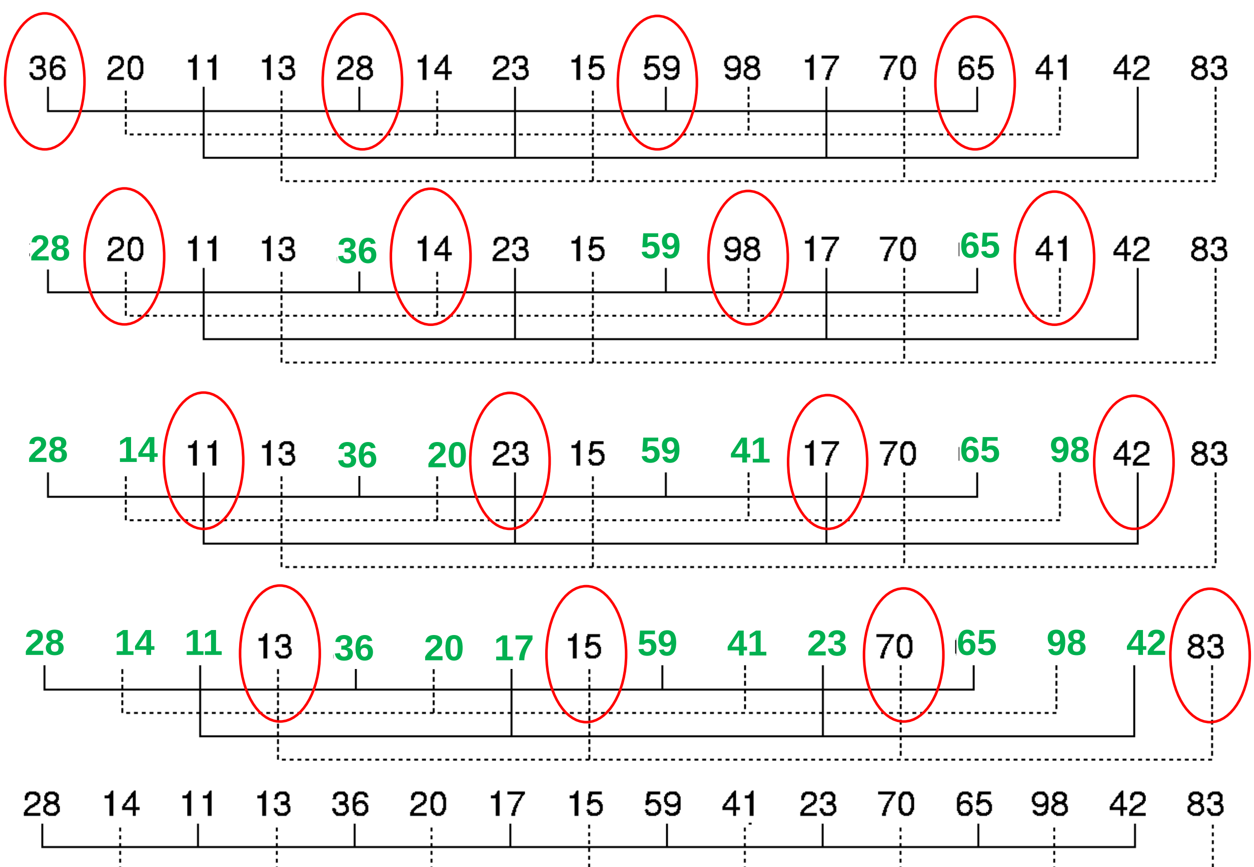
- Modified version of Insertion Sort for varying increments
  - Insertion Sort among a set of elements with gapped positions.

```
// Modified version of Insertion Sort
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
 for (int i=incr; i<n; i+=incr)
  for (int j=i; (j>=incr) && (Comp::prior(A[j], A[j-incr]));
       j-=incr)
   swap(A, j, j-incr);
}

//Shellsort
template <typename E, typename Comp>
void shellsort(E A[], int n) {
 for (int i=n/2; i>2; i/=2) //For each increment
  for (int j=0; j<i; j++) //Sort each sublist
   inssort2<E,Comp>(&A[j], n-j, i);
 //Normal insertion sort
 inssort2<E,Comp>(A, n, 1);
}
```

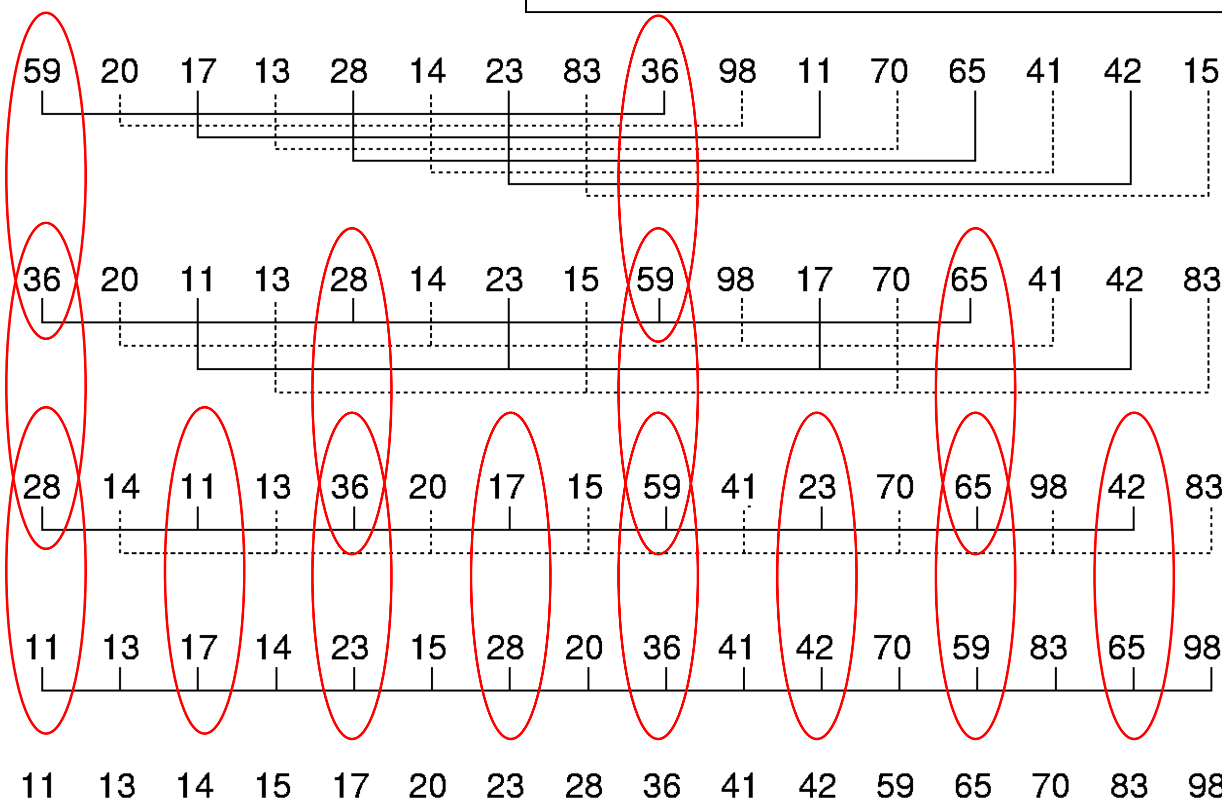
# Shellsort

```
for (int j=0; j<i; j++) //for sublist
      inssort2<E,Comp>(&A[j], n-j, i);
```



# Shellsort (VI)

```
for (int i=n/2; i>2; i/=2)
  for (int j=0; j<i; j++)
     inssort2<E,Comp>(&A[j], n-j,
i);
inssort2<E,Comp>(A, n, 1);
```



# Shellsort

- Shellsort conclude with a regular Insertion Sort
  - The complexity will be at least the complexity of Insertion Sort?
  - Each of the sublist sorts will make the list "more sorted" than it was before, which leads to a relatively cheap final Insertion Sort pass.
- Choice of increments also influences the efficiency

# Analysis of Shellsort

- Shellsort is a very simple algorithm with an extremely complex analysis.

- The worst-case running time of Shellsort using Shell's increments is $\Theta(N^2)$.
  - Shell's increments: $h_t$=floor(N/2), $h_k$=floor( $h_k$+1/2)

- The worst-case running time of Shellsort using Hibbard's increments is $\Theta$ ($N^{3/2}$).
  - Hibbard's increments: 1, 3, 7, . . . , $2^k - 1$
  - The key difference is that consecutive increments have no common factors
  - The average-case running time of Shellsort, using Hibbard's increments, is thought to be $O(N^{5/4})$.(not proven)

- The worst-case running time of Shellsort using Sedgewick's increments is $O(N^{4/3})$, $O(N^{7/6})$ for the average-case.
  - Sedgewick's increments: 1, 5, 19, 41, 109, . . ., ($9 \cdot 4^i - 9 \cdot 2^i + 1$ or $4^i - 3 \cdot 2^i + 1$). (best known in practice)

# Homework

- Coming soon
- Deadline: to be confirmed.