

파이썬으로 배우는 데이터 구조



한동대학교
전산전자공학부
김영섭 교수



학습 목표

삽입(Insertion) 정렬 알고리즘을 이해하고
시간복잡도를 계산할 수 있다

Data Structures in Python

Chapter 5 - 1

- Binary Search
- Recursive Binary Search
- Bubble sort
- Selection sort
- **Insertion sort**

Agenda & Readings

- Agenda
 - Insertion sort algorithm
 - Time complexity
 - Summary - sorting basics
 - Empirical Analysis
- Reference:
 - Problem Solving with Algorithms and Data Structures
 - Chapter 5 Search, Sorting and Hashing

Insertion Sort Algorithm

- Given is a list L of n value $\{L[0], \dots, L[n-1]\}$
 - Divide list into sorted (left - initially only one element) and sorted part (right):
Sorted: $\{L[0]\}$ **Unsorted:** $\{L[1], \dots, L[n-1]\}$
 - In each pass, **take left most element from unsorted part** and **place it into correct position of sorted part**.
 - Reduce size of unsorted part by one and increase size of sorted part by one.
After i-th pass:
Sorted: $\{L[0], \dots, L[i]\}$ **Unsorted:** $\{L[i+1], \dots, L[n-1-i]\}$
 - Repeat until unsorted part is an empty list - then all elements are sorted.

← Unsorted →				
29	10	14	13	18
10	29	14	13	18
10	14	29	13	18
10	13	14	29	18
10	13	14	18	29

List to sort

PASS 1 (1 Comp, 1 Shift)

PASS 2 (2 Comp, 1 Shift)

PASS 3 (3 Comp, 2 Shift)

PASS 4 (2 Comp, 1 Shift)

Insertion Sort Algorithm

- Given is a list L of n value $\{L[0], \dots, L[n-1]\}$
 - Divide list into sorted (left - initially only one element) and sorted part (right):
Sorted: $\{L[0]\}$ **Unsorted:** $\{L[1], \dots, L[n-1]\}$
 - In each pass, **take left most element from unsorted part** and **place it into correct position of sorted part**.
 - Reduce size of unsorted part by one and increase size of sorted part by one.
After i-th pass:
Sorted: $\{L[0], \dots, L[i]\}$ **Unsorted:** $\{L[i+1], \dots, L[n-1-i]\}$
 - Repeat until unsorted part is an empty list - then all elements are sorted.

← Unsorted →				
29	10	14	13	18
10	29	14	13	18
10	14	29	13	18
10	13	14	29	18
10	13	14	18	29

List to sort

PASS 1 (1 Comp, 1 Shift)

PASS 2 (2 Comp, 1 Shift)

PASS 3 (3 Comp, 2 Shift)

PASS 4 (2 Comp, 1 Shift)

29	10	14	13	18
10	29	14	13	18
10	14	29	13	18
10	13	14	29	18
10	13	14	18	29

→ pick 10 & inserted

→ pick 14 & inserted

→ pick 13 & inserted

→ pick 18 & inserted

Insertion Sort – Exercise

→

pick 26 & inserted

pick 93 & inserted

pick 17 & inserted

pick 77 & inserted

pick 31 & inserted

pick 44 & inserted

pick 55 & inserted

pick 20 & inserted

54	26	93	17	77	31	44	55	20
26	54	93						

List to sort

PASS 1 (Comp, Shift)

PASS 2 (Comp, Shift)

PASS 3 (Comp, Shift)

PASS 4 (Comp, Shift)

PASS 5 (Comp, Shift)

PASS 6 (Comp, Shift)

PASS 7 (Comp, Shift)

PASS 8 (Comp, Shift)

Total PASS 8 (26 Comp, 20 Shift)

Insertion Sort – Exercise

	35	34	26	90	37	28	10	27	36	List to sort
→ pick 34 & inserted	34	35								PASS 1 (1 Comp, 1 Shift)
pick 26 & inserted										PASS 2 (Comp, Shift)
pick 90 & inserted										PASS 3 (Comp, Shift)
pick 37 & inserted										PASS 4 (Comp, Shift)
pick 28 & inserted										PASS 5 (Comp, Shift)
pick 10 & inserted										PASS 6 (Comp, Shift)
pick 27 & inserted										PASS 7 (Comp, Shift)
pick 36 & inserted										PASS 8 (Comp, Shift)

Insertion Sort - making room for the element to be inserted

6	28	34	35	37	90	10	27	36
---	----	----	----	----	----	----	----	----



- For example, to insert 10 into the sorted part of the list we need to store 10 into a temporary variable and move all the elements which are bigger than 10 up one position, then insert 10 into the empty slot.

temp = 10

6	28	34	35	37	90	—	27	36
---	----	----	----	----	----	---	----	----

Shift 5 list values

6	—	28	34	35	37	90	27	36
---	---	----	----	----	----	----	----	----

6	10	28	34	35	37	90	27	36
---	----	----	----	----	----	----	----	----

Insertion Sort Code

```
def insertion_sort(a):
    for i in range(1, len(a)):
        ivalue = a[i]
        while i > 0 and a[i - 1] > ivalue:
            a[i] = a[i - 1]          # this is "shift", not swap
            i = i - 1
        a[i] = ivalue
        #print(i, "-", a) # enable to see each pass

if __name__ == '__main__':
    a = [54, 26, 93, 17, 77, 31, 44, 55, 20]
    print("before: ", a)
    insertion_sort(a)
    print(" after: ", a)
```

```
before: [54, 26, 93, 17, 77, 31, 44, 55, 20]
after:  [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

```
def swap(a, i, j):
    temp = a[i]
    a[i] = a[j]
    a[j] = temp
```

```
def swap(a, i, j):
    a[i], a[j] = a[j], a[i]
```

Insertion Sort - Big O

- For a list with n elements:
 - The number of comparisons in the WORST CASE?
 - | | | | | |
|--------|--------|--------|-----|-----------|
| pass 1 | pass 2 | pass 3 | ... | last pass |
| 1 | 2 | 3 | ... | $n-1$ |
- $$1 + 2 + \dots + (n-3) + (n-2) + (n-1) = \frac{1}{2}(n^2 - n)$$
- Big O of the insertion sort is $O(n^2)$
 - The number of data increases 10 times, then it takes a 100 times longer.
- Note 1: Best case **$O(n)$... when does this occur?**
- Note 2: The number of shifts is equal or one smaller than the number of comparisons, so same order of magnitude.

Insertion Sort - Big O

- What if the data is **already sorted**?
 - Move elements?
 - Comparisons?

29	10	14	13	18
10	29	14	13	18
10	14	29	13	18
10	13	14	29	18
10	13	14	18	29

List to sort

PASS 1 (1 Comp, 1 Shift)

PASS 2 (2 Comp, 1 Shift)

PASS 3 (3 Comp, 2 Shift)

PASS 4 (2 Comp, 1 Shift)



pick 10 & inserted

pick 14 & inserted

pick 32 & inserted

pick 35 & inserted

5	10	14	32	35
5	10	14	32	35
5	10	14	32	35
5	10	14	32	35

List to sort

PASS 1 (Comp, Shift)

PASS 2 (Comp, Shift)

PASS 3 (Comp, Shift)

PASS 4 (Comp, Shift)

Insertion Sort - Big O

- What if the data is in **reverse order**?
 - Move elements?
 - Comparisons?

29	10	14	13	18
10	29	14	13	18
10	14	29	13	18
10	13	14	29	18
10	13	14	18	29

List to sort

PASS 1 (1 Comp, 1 Shift)

PASS 2 (2 Comp, 1 Shift)

PASS 3 (3 Comp, 2 Shift)

PASS 4 (2 Comp, 1 Shift)



pick 10 & inserted

pick 14 & inserted

pick 32 & inserted

pick 35 & inserted

35	32	14	10	5
32	35	14	10	5
14	32	35	32	35
10	14	32	35	35
5	10	14	32	35

List to sort

PASS 1 (Comp, Shift)

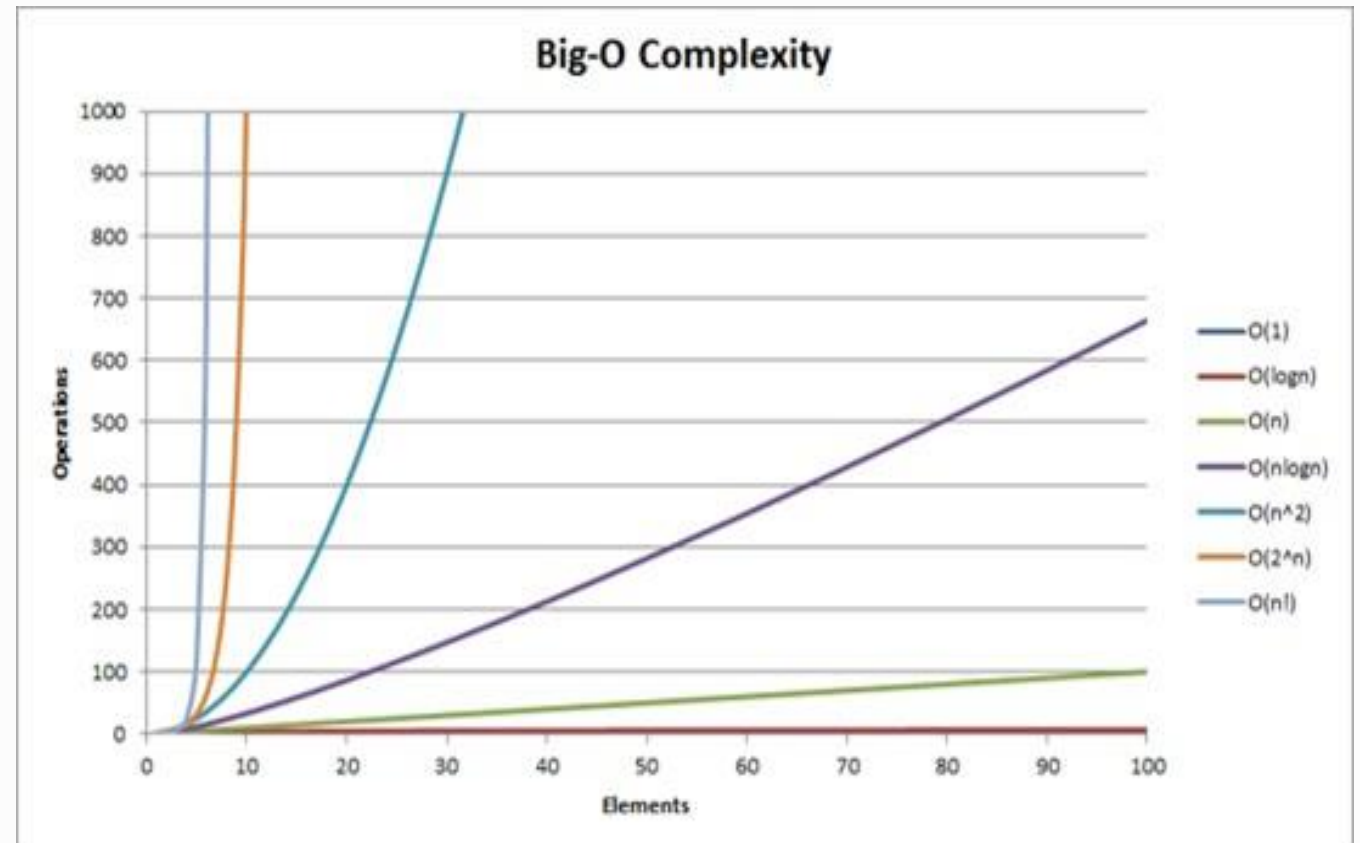
PASS 2 (Comp, Shift)

PASS 3 (Comp, Shift)

PASS 4 (Comp, Shift)

Running Time Matters

- The usefulness of an algorithm in practice depends on the data size n and the complexity (Big O) of the algorithm (time and memory).
 - In general algorithms with linear, logarithmic or low polynomial running time are acceptable.
 - $O(\log n)$
 - $O(n)$
 - $O(n^k)$ where k is a small constant, (in many cases $k < 2$ is ok)
 - Algorithms with exponential or high polynomial running time are often of limited use.
 - $O(n^k)$ where k is a large constant, say > 3
 - $O(2^n), O(n^n)$
-
- | n | $O(\log n)$ | $O(n)$ | $O(n^2)$ | $O(n^3)$ | $O(2^n)$ | $O(n^4)$ |
|----|-------------|--------|----------|----------|------------|----------|
| 10 | 3.3 | 10 | 100 | 1000 | 1024 | 10000 |
| 20 | 4.3 | 20 | 400 | 8000 | 1048576 | 160000 |
| 30 | 4.8 | 30 | 900 | 27000 | 1073741824 | 810000 |



Summary

- Insertion sort is a good middle-of-the-road choice for sorting lists of a few thousand items or less.
- **Insertion sort is known faster than selection sort** on average.
- For small lists, the insertion sort is appropriate due to its simplicity.
For **almost sorted** lists, the insertion sort is **a good choice**.
- For large lists, all $O(n^2)$ algorithms, including the insertion sort, are prohibitively inefficient.

Summary - Simple Sorting Algorithms

- All sorting algorithms (bubble, selection, insertion sorts) discussed so far had an $O(n^2)$ average and the worst-case complexity
→ In practice for large lists it is **too slow**.
- The **Timsort** algorithm (written in C - not using the Python interpreter) used by Python combines elements from **Merge sort** and **Insertion sort**.
 - Worst case and average case complexity $O(n \log n)$
 - Very fast for almost sorted lists
- All comparison-based sorting algorithms require at least $O(n \log n)$ time in the worst and average case.

학습 정리

- 1) 교환(swap)과 이동(shift)를 비교했을 때, shift는 값을 단순히 덮어쓰는 것이기 때문에 swap보다 더 빠르다
- 2) 삽입 정렬은 요소가 적을 때에는 효율적이지만 요소가 많아질수록 $O(n^2)$ 의 시간복잡도를 가진다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다