Algorithm – Search and Sort (1)

Hyung-Sin Kim

Computing Foundations for Data Science

Graduate School of Data Science, Seoul National University

Review

- Class vs. Class object
- Method vs. Function
- Object-oriented programming
  - Encapsulation
  - Abstraction
  - Inheritance
  - Polymorphism
  - This is a CS course dedicated for OOP…

Why Search?

- Searching is a fundamental part of programming, especially in data science

- There are **massive** amount of data in the world and you want to find data you are interested

- You should find data that you want, **efficiently**

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 4

Found!
Return 6

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	**4**	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear** **search**)
- If the target value exists, return the index where the value **first occurs**
- Otherwise, return -1

target = 4

Found!
Return 6

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	**4**	9	-7	50	4	3

Ignored…

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear** **search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 1

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Linear Search – Algorithm

- Find if a target value exists in a list
- To do this, search from the first item to the last item sequentially (**linear search**)
- If the target value exists, return the index where the value first occurs
- Otherwise, return -1

target = 1

No target!
Return -1

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Algorithm *vs.* ***Programming***

Algorithm*: A recipe for computers to follow (logical steps)*
Program*: An instruction set in programming languages for a computer to understand and put an algorithm to practice*

There can be ***many different*** *ways to implement (program)*
a ***single*** *algorithm!*

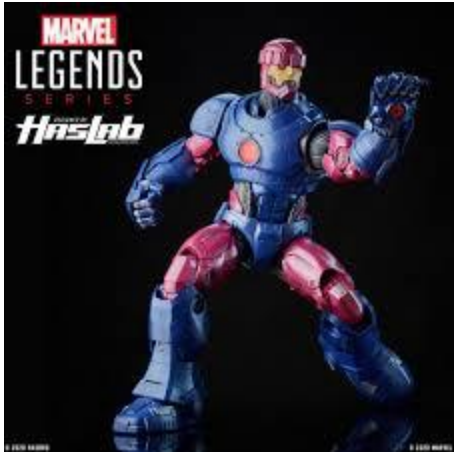
Linear Search – Impl (1): While Loop

```
def linear_search_while(L: list, value: Any) -> int:
        i = 0
        while i < len(L) and L[i] != value:
                i = i + 1
        if i == len(L):
                return -1
        else:
                return i
```

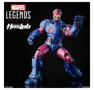
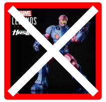
Linear Search – Impl (2): While Loop with Sentinel

- The while loop version needs to do (i < len(L)) every time
  - Because we need to know when the loop reaches the end of the list

- How can we remove this? – by using **sentinel** at the end of the list!



Linear Search – Impl (2): While Loop with Sentinel

- def linear_search_sentinel(L: list, value: Any) -> int:
-     **L.append(value)** # <u>Add the sentinel</u> 
-     i = 0
-     **while L[i] != value**: # <u>This condition is enough!</u>
-         i = i + 1
-     **L.pop()** # <u>Remove the sentinel</u> 
-     if i == len(L):
-         return -1
-     else:
-         return i

Caveat
Some people do not like modifying the input list because it could be dangerous and possibly incur errors

Linear Search – Impl (3): For Loop

- def linear_search_for(L: list, value: Any) -> int:
-         for i in range(len(L)):
-                 if L[i] == value:
-                         return i
-         return -1

- Simple code, no complex conditions
- But some people dislike returning in the middle of a loop
- We have learnt three types of linear search, among which you can choose according to your taste ☺

Linear Search – Time Complexity

- How to measure time spent for an algorithm?
  - import time
  - t_start = time.perf_counter()
  - **<<Your Algorithm>>**
  - t_end = time.perf_counter()
  - return (t_end – t_start) * 1000.0 # the unit becomes milliseconds

Linear Search – Time Complexity (10 M items)

- When the value is located at the end of the list, it takes more time (**linear increase**)
  - This is why the algorithm is called **linear** search!
- Built-in list.index is the **fastest**
  - Python program is notoriously slow since every line of code needs to pass through the Python **interpreter** at run time

Case	while	sentinel	for	list.index
First	0.01	0.01	0.01	0.01
Middle	1261	697	515	106
Last	2673	1394	1029	212

*What if the list is **sorted**?*
Can we do anything better?

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

*What if the list is **sorted**?*
Can we do anything better?

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Motivation

- Linear search does work for a sorted list, but does **NOT** take advantage of the fact that it is sorted

target = 4

Return 5
(Takes similar)

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Motivation

- Linear search does work for a sorted list, but does **NOT** take advantage of the fact that it is sorted

target = 100

Return 11
Takes very long

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search - Idea

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = 0
end = 11

mid = (0+11)//2
4 < target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = ***6***
end = *11*

mid = (0+11)//2
4 < target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = 6
end = 11

mid = (6+11)//2
7 < target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = ***9***
end = 11

mid = (6+11)//2
7 < target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = 9
end = 11

mid = (9+11)//2
50 >= target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = 9
end = ***9***

mid = (9+11)//2
50 >= target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = 9
end = 9

mid = (9+9)//2
9 >= target

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

Binary Search – Algorithm

- Idea: Evaluate the **middle** of the sorted list and removes half of candidate entries
- Linear search: one evaluation removes **one** candidate entry
- Binary search: one evaluation removes **half** of candidate entries

target = 9

start = 9
*end = **8***

Cannot proceed further!

If there is target in L,
L[start] must be the target!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	**9**	50	100

Binary Search – Code

```python
def binary_search(L: list, v: Any) -> int:
    start, end = 0, len(L) – 1
    while start != end + 1:
        mid = (start+end) // 2
        if L[mid] < v:
            start = mid + 1
        else:
            end = mid - 1
    if start < len(L) and L[start] == v:
        return start
    else:
        return -1
```

Binary Search – Time Complexity (10 M items)

- Linear search
  - Time delay is proportional to **len(L)**
- Binary search
  - Time delay is proportional to $\boldsymbol{log_2^{len(L)}}$
- A good example why **<u>sorting</u>** is useful!
  - But remember that sorting is **NOT** free either. It also takes non-negligible time…

Case	list.index	binary_search
First	0.007	0.02
Middle	105	0.02
Last	211	0.02 (WoW!)

Summary

- Linear search
  - Evaluate the **first** item and cut the **one** evaluated item
  - Time proportional to **len(L)**
  - Applicable to **any** list

- Binary search
  - Evaluate the **middle** item and cut the **half**
  - Time proportional to $\boldsymbol{log_2^{len(L)}}$
  - Applicable to a **sorted** list

Let’s move onto sorting

Why Sorting?

- People often want to see numerous items sorted!
  - Midterm score, sports…
  - Dictionary

- Sorting helps searching
  - Binary search

투수 순위 | 타자 순위

순위	선수	타율	경기수	타수	안타	2루타	3루타	홈런	타점	득점	도루	볼넷	삼진	출루율	장타율
1	페르난데스 (두산)	0.369	92	374	138	21	0	16	76	72	0	37	30	0.431	0.553
2	로하스 (KT)	0.354	89	359	127	28	1	31	84	76	0	32	89	0.409	0.696
3	김현수 (LG)	0.350	93	369	129	26	1	20	80	67	0	37	32	0.407	0.588
4	이정후 (키움)	0.349	95	373	130	36	4	14	74	63	8	35	29	0.404	0.579
5	손아섭 (롯데)	0.346	86	327	113	25	0	6	57	64	3	43	38	0.418	0.477
6	강진성 (NC)	0.339	74	251	85	17	0	12	54	40	5	11	28	0.371	0.550
7	이명기 (NC)	0.337	84	297	100	14	2	1	34	52	8	26	49	0.391	0.407
8	오재일 (두산)	0.336	75	289	97	20	0	11	55	40	0	29	56	0.395	0.519
9	최형우 (KIA)	0.333	87	318	106	20	1	11	57	56	0	46	58	0.423	0.506
10	박민우 (NC)	0.326	70	264	86	17	2	4	28	47	8	22	24	0.375	0.451
11	나성범 (NC)	0.323	83	341	110	25	2	25	78	74	0	35	101	0.396	0.628
12	배정대 (KT)	0.321	89	327	105	22	3	9	41	55	16	42	85	0.401	0.489
13	김상수 (삼성)	0.320	73	259	83	17	1	3	25	48	8	39	36	0.422	0.429
14	정훈 (롯데)	0.319	59	235	75	14	1	7	39	48	6	30	52	0.400	0.477
15	조용호 (KT)	0.315	81	254	80	11	0	0	17	49	8	36	51	0.403	0.358
16	구자욱 (삼성)	0.313	69	262	82	15	0	8	43	38	12	30	58	0.390	0.462
17	박해민 (삼성)	0.310	81	294	91	12	3	7	33	49	15	17	48	0.349	0.442
18	강백호 (KT)	0.309	74	285	88	17	1	15	51	53	3	34	57	0.383	0.533
19	마차도 (롯데)	0.309	87	311	96	22	1	7	47	46	10	30	38	0.370	0.453
20	정수빈 (두산)	0.305	89	311	95	11	5	2	35	54	9	28	37	0.365	0.392

Then, how can we sort a list?

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Selection Sort – Idea

- Find the minimum value of the unsorted list and swap it with the leftmost entry

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

1-st iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

unsorted

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

1-st iteration

Minimum in [0:11]

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	**-7**	50	4	3

unsorted

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

1-st iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	**-7**	-2	0	100	-6	7	4	9	**5**	50	4	3

unsorted

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

1-st iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	**-7**	-2	0	100	-6	7	4	9	**5**	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

2-nd iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-2	0	100	-6	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

2-nd iteration

Minimum in [1:11]

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-2	0	100	**-6**	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

2-nd iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	**-6**	0	100	**-2**	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

2-nd iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	**-6**	0	100	**-2**	7	4	9	5	50	4	3

sorted | *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

3-rd iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	0	100	-2	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

3-rd iteration

Minimum in [2:11]

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	0	100	**-2**	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

3-rd iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	**-2**	100	**0**	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

3-rd iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	**-2**	100	**0**	7	4	9	5	50	4	3

sorted | *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

4-th iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	100	0	7	4	9	5	50	4	3

sorted *unsorted*

SNU Data Science
서울대학교 데이터사이언스대학원

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

4-th iteration

Minimum in [3:11]

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	100	**0**	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

4-th iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	**0**	**100**	7	4	9	5	50	4	3

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

4-th iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	**0**	**100**	7	4	9	5	50	4	3

sorted | *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

5-th iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	100	7	4	9	5	50	4	3

sorted | *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

5-th iteration

Minimum in [4:11]

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	100	7	4	9	5	50	4	**3**

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

5-th iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	**3**	7	4	9	5	50	4	**100**

sorted | *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

5-th iteration

Swap!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	**3**	7	4	9	5	50	4	**100**

sorted *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

Repeat the procedure 12 times!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	7	4	9	5	50	4	100

sorted | *unsorted*

Selection Sort – Algorithm

- Find the minimum value of the unsorted list and swap it with the leftmost entry

Repeat the procedure 12 times!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

sorted

Selection Sort – Code

- def selection_sort(L: list) -> None:
- for i in range(len(L)):
- *# Find the index of the smallest item in L[i:]: **smallest***
- L[i], L[smallest] = L[smallest], L[i] *# swap*

Selection Sort – Code

- def selection_sort(L: list) -> None:
- for i in range(len(L)):
- smallest = **find_min**(L, i)
- L[i], L[smallest] = L[smallest], L[i] *# swap*

Selection Sort – Code

- def find_min(L: list, start_idx: int) -> int:
- smallest = start_idx # *(1) Initialize smallest*
- for i in range(start_idx+1, len(L)): # *(2) Update smallest*
- if L[i] < L[smallest]:
- smallest = i
- return smallest # *(3) Return the final value*

Selection Sort – Code (in one function)

```
def selection_sort(L: list) -> None:
    for i in range(len(L)):
        smallest = i
        for j in range(i+1, len(L)):
            if L[j] < L[smallest]:
                smallest = j
        L[i], L[smallest] = L[smallest], L[i]    # swap
```

Selection Sort – Time Complexity

- At i-th iteration, its inner loop (func **find_min**) needs to look up (N+1-i) items
  - When N = len(L)

- N + (N-1) + (N-2) + … + 1 = **N(N+1)/2**

Let’s see another sorting algorithm called
Insertion Sort

Insertion Sort – Idea

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

1-st iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	-2	0	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

1-st iteration

Insert location *Current target*

index	0	1	2	3	4	5	6	7	8	9	10	11
values	5	**-2**	0	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

1-st iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	**-2**	5	0	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

1-st iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	**-2**	5	0	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

2-nd iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	5	0	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

2-nd iteration

Insert location *Current target*

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	5	**0**	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

2-nd iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	**0**	5	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

2-nd iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	**0**	5	100	-6	7	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

3-rd iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	0	5	100	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

3-rd iteration

Insert location *Current target*

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	0	5	**100**	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

3-rd iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	0	5	**100**	-6	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

3-rd iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	0	5	**100**	-6	7	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

4-th iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	0	5	100	-6	7	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

4-th iteration

Insert location *Current target*

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-2	0	5	100	**-6**	7	4	9	-7	50	4	3

sorted *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

4-th iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	**-6**	-2	0	5	100	7	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

4-th iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	**-6**	-2	0	5	100	7	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

5-th iteration

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-6	-2	0	5	100	7	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

5-th iteration

Insert location ↓ (index 4) *Current target* ↓ (index 5)

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-6	-2	0	5	100	**7**	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

5-th iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-6	-2	0	5	**7**	100	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

5-th iteration

Insert

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-6	-2	0	5	**7**	100	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

Repeat the procedure 11 times!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-6	-2	0	5	7	100	4	9	-7	50	4	3

sorted | *unsorted*

Insertion Sort – Algorithm

- Insert the leftmost item of the unsorted list to the proper location of the sorted list

Repeat the procedure 11 times!

index	0	1	2	3	4	5	6	7	8	9	10	11
values	-7	-6	-2	0	3	4	4	5	7	9	50	100

sorted

SNU Data Science
서울대학교 데이터사이언스대학원

Insertion Sort – Code

- def insertion_sort(L: list) -> None:
-         for i in range(1, len(L)):
-                 *# insert L[i] to the proper location of L[:i]*

- def insertion_sort(L: list) -> None:
-         for i in range(1, len(L)):
-                 **insert(L, i)**

Insertion Sort – Code

```
def insert(L: list, last_idx: int) -> None:
    for i in range(last_idx,0,-1):          # (1) Go backwards
        if L[i-1] > L[i]:                   # (2) Check stopping condition
            L[i-1], L[i] = L[i], L[i-1]     # (3) Swap
        else:
            break
```

Insertion Sort – Code

```
def insertion_sort(L: list) -> None:
    for i in range(1, len(L)):
        for j in range(i,0,-1):            # (1) Go backwards
            if L[j-1] > L[j]:               # (2) Check stopping condition
                L[j-1], L[j] = L[j], L[j-1] # (3) Swap
            else:
                break
```

Insertion Sort – Time Complexity

- At i-th iteration, its inner loop (**func insert**) needs to look up $(i+1)/2$ items and swap $i/2$ times on average
  - Look up: $1 + 1.5 + 2 + 2.5 + \ldots + (N-1)/2 + N/2$ (When N = len(L))
    - $= (1 + 2 + 3 + \ldots + (N-1) + N)/2 - ½ =$ **$N(N+1)/4 - ½$**
  - Swap: $0.5 + 1 + 1.5 + \ldots + (N-1)/2$
    - $= (1 + 2 + 3 + \ldots + (N-1))/2 =$ **$(N-1)N/4$**

- **A bit slower** than Selection sort
  - find_min() needs to look up the **whole** list
  - Insert() needs to look up only **half** on average but also need to swap!

- When a list is almost sorted, insertion sort needs to look up only **kN** items

Yes, there are better sorting algorithms,
*which you will see next time ^0^*

Thanks!