Review

- 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 <u>**sorted**</u> list

Selection Sort

Lecture 10-1

Hyung-Sin Kim

SNU Graduate School of Data Science

Why Sorting?

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

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

- Sorting helps searching
  - Binary search

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*

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**

Summary

- Selection sort – A basic sorting algorithm
  - Find the minimum value of the unsorted list and swap it with the leftmost entry
  - Time complexity ~ N**2

Insertion Sort

Lecture 10-2

Hyung-Sin Kim

SNU Graduate School of Data Science

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 *Current target*

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

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 + … + (N-1)/2 + N/2 (When N = len(L))
    - = (1 + 2 + 3 + … + (N-1) + N)/2 – ½ = **N(N+1)/4 – ½**
  - Swap: 0.5 + 1 + 1.5 + … + (N-1)/2
    - = (1 + 2 + 3 + … + (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

Summary

- Insertion sort
  - Insert the leftmost item of the unsorted list to the proper location of the sorted list
  - Time complexity ~ N**2 (a bit slower than selection sort)
  - Nice when a list is almost sorted already

Computing Bootcamp

Big O

Lecture 10-3

Hyung-Sin Kim

SNU Graduate School of Data Science

Two Types of Program Cost

- Execution cost (our focus while learning algorithms)
  - Time complexity of a program (how much time?)
  - Memory complexity of a program (how much memory?)

- Programming cost (very important in practice, but not a focus in this course)
  - Development time
    - What if you develop a very nice program a year later than your competitor?
  - Readability, modifiability, and maintainability
    - Super important for real-world products (majority of cost actually…)

Measuring Time Complexity

- Measure execution time in seconds using a client program (e.g., time module)
  - **Pros**: Easy to measure. Gives actual time
  - **Cons**: large amounts of time might be required. Results depend on lots of factors (machine, compiler, data…)

- Count possible operations in terms of input list size **N**
  - **Pros**: Machine independent. Gives algorithm's scalability
  - **Cons**: Tedious to compute… Does not give actual time

⇒ Fortunately, we usually care only about asymptotic behavior
(with a very large **N** – Big Data!)

Count Possible Operations

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

- Assume that input list size is **N**

Operation	Count
==	1 to N

```
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]
```

Operation	Count
Smallest = i	
<	
Smallest = j	
Swapping	

Count Possible Operations

Operation	Count
Smallest = i	N
$<$	$(N^2 - N)/2$
Smallest = j	0 to $(N^2 - N)/2$
Swapping	N

```python
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]
```

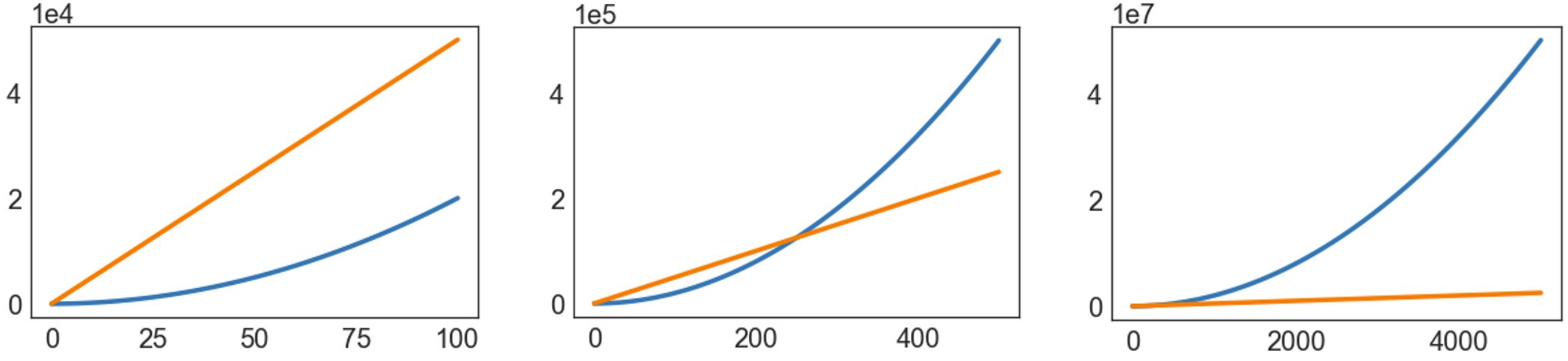
- Assume that input list size is **N**

Operation	Count
==	1 to N

Operation	Count
Smallest = i	N
$<$	$(N^2 - N)/2$
Smallest = j	0 to $(N^2 - N)/2$
Swapping	N

What is Important for Asymptotic Analysis?

- Compare the two algorithms below:
  - Algorithm 1 requires $2N^2$ operations
  - Algorithm 2 requires 500N operations
- Algorithm 1 is faster than Algorithm 2 for a small N, but becomes much slower for a very large N
  - What is important?: Not a specific value but a function **shape**! (parabola vs. line)
  - **Order of growth**



The figures are from 61B course material at UC Berkeley

How can we characterize an algorithm's time complexity more ***formally*** *and* ***simply****?*

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance

Operation	Count
Smallest = i	N
<	$(N^2 - N)/2$
Smallest = j	~~0 to~~ $(N^2 - N)/2$
Swapping	N

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance
- 2. Focus on only one operation that has the highest order of growth
  - There could be multiple good choices. Then, just choose any of them.

Operation	Count
Smallest = i	~~N~~
<	~~$(N^2 - N)/2$~~
Smallest = j	$(N^2 - N)/2$
Swapping	~~N~~

Simplification to Find “Order of Growth”

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance
- 2. Focus on only one operation that has the highest order of growth
  - There could be multiple good choices. Then, just choose any of them.
- 3. Remove lower order terms

Operation	Count
Smallest = j	$(N^2 \cancel{- N})/2$

Simplification to Find "Order of Growth"

- 1. Consider only the worst case
  - When comparing algorithms, we usually care only about the worst case performance
- 2. Focus on only one operation that has the highest order of growth
  - There could be multiple good choices. Then, just choose any of them.
- 3. Remove lower order terms
- 4. Remove constants
  - We have already thrown away information at step 2. At this stage, constants are not meaningful

- Worst-case order of growth of **selection sort**
  - N^2

Operation	Count
Smallest = j	$N^2\cancel{/2}$

Formal Definition

- If a function $T(N)$ has its order of growth less than or equal to $f(N)$,
- we write this as $\boldsymbol{T(N) \in O(f(N))}$
- where $\boldsymbol{O}$ is called **Big-O** notation

- More mathematically, $\boldsymbol{T(N) \in O(f(N))}$ means that
- there exists positive constants k such that
- $T(N) \leq k \cdot f(N)$ for all values of N greater than some N_0 (i.e., very large N)

Examples

- Simplify T(N) to find f(N) and use the Big-O notation

Function T(N)	Order of Growth in terms of Big-O
$\mathrm{N}^2 + 5\mathrm{N}^5$	
$1/N + 100$	
$100\cos(N) + \mathrm{N}^2/50$	
$Ne^{2N} + 100\mathrm{N}^2$	

Examples

- Simplify T(N) to find f(N) and use the Big-O notation

Function T(N)	Order of Growth in terms of Big-O
$\mathrm{N}^2 + 5\mathrm{N}^5$	$O(N^5)$
$1/N + 100$	$O(1)$
$100\cos(N) + \mathrm{N}^2/50$	$O(N^2)$
$Ne^{2N} + 100\mathrm{N}^2$	$O(Ne^{2N})$

Summary

- Big O
  - A simple and formal way to represent complexity
  - Focusing on asymptotic behavior
  - No need to run an algorithm on a machine

Thanks!