COMP 204
Algorithm design: Selection and Insertion Sort

Yue Li
based on material from Mathieu Blanchette, Christopher J.F. Cameron and Carlos G. Oliver
A **sorting algorithm** is an algorithm that takes

- a list/array as input
- performs specified operations on the list/array
- outputs a sorted list/array

For example:

- \([a, c, d, b]\) could be sorted alphabetically to \([a, b, c, d]\)
- \([1, 3, 2, 0]\) could be sorted:
  - increasing order: \([0, 1, 2, 3]\)
  - or decreasing order: \([3, 2, 1, 0]\)
Why is it useful to sort data?

Sorted data searching can be optimized to a very high level
▶ also used to represent data in more readable formats

Contacts
▶ your mobile phone stores the telephone numbers of contacts by names
▶ names can easily be searched to find a desired number

Dictionary
▶ dictionaries store words in alphabetical order to allow for easy searching of any word

Remember **binary search**?
Adding more algorithms to your toolbox

In the last lecture, we covered searching algorithms, specifically:

▶ linear search
▶ binary search

Today, we will cover the following sorting algorithms:

▶ selection sort
▶ insertion sort

Images are taken from the following online tutorial: https://www.tutorialspoint.com/data_structures_algorithms/
Selection sort

Conceptually the most simple of all the sorting algorithms

Start be selecting the smallest (or largest) item in a list
▶ then place this item at the start of the list
▶ repeat for the remaining items in the list
   ▶ move next smallest/largest item to the second position
   ▶ then the next
   ▶ and so on and so on...
   ▶ until the list is sorted

Let’s consider the following unsorted list:

```
14  33  27  10  35  19  42  44
```
Selection sort #2

For the first position in the resulting sorted list
  ▶ the whole list is scanned sequentially
  ▶ the first position is where 14 is currently stored

We search the whole list
  ▶ to find that 10 is the lowest value in the list
Selection sort #3

We then replace 14 with 10

After one iteration

- 10, which happens to be the minimum value in the list
- appears in the first position of the sorted list

For the second position

- where 33 is residing
- we start scanning the rest of the list in a linear manner
Selection sort #4

14 is found to be the second lowest value in the list
▶ and should appear at the second place
▶ we swap these values.

After two iterations
▶ the two items with the least values
▶ are positioned at the beginning in a sorted manner
Selection sort #5

The same process is applied to the rest of the items in the list
Selection sort #6

Until the list is sorted
Selection sort algorithm

**Selection sort** *(sequence)*

**Step 1** - find the item with the smallest value in *sequence*
**Step 2** - swap it with the first item in *sequence*
**Step 3** - find the item with the second smallest value in *sequence*
**Step 4** - swap it with the second item in *sequence*
**Step 5** - find the item with the third smallest value in *sequence*
**Step 6** - swap it with the third item in *sequence*
**Step 7** - repeat finding the item with the next smallest value
**Step 8** - then swap it with the correct item until *sequence* is sorted
Algorithm 1 Selection sort

1: procedure SELECTION_SORT(sequence)
2: \( N \leftarrow \text{length of sequence} \)
3: for \( i \leftarrow 0 \) to \( N - 1 \) do
4: \( min \leftarrow i \)
5: for \( j \leftarrow i + 1 \) to \( N - 1 \) do
6: if \( \text{sequence}[j] \leq \text{sequence}[min] \) then
7: \( min \leftarrow j \)
8: end if
9: end for
10: SWAP(\( \text{sequence}[i], \text{sequence}[min] \))
11: end for
12: end procedure
Selection sort: Python implementation

```python
import random

sequence = list(range(0,10))
random.shuffle(sequence)  # shuffles items
N = len(sequence)
for i in range(0,N,1):  # why not N-1?
    min = i
    for j in range(i+1,N,1):
        if sequence[j] <= sequence[min]:
            min = j
    sequence[i],sequence[min] = sequence[min],sequence[i]
print(sequence)  # prints ???
```
Insertion sort

Insertion sort does what you might expect
▶ inserts each item of the list into its proper position
▶ resulting in progressively larger sequences of a sorted list

Start with a sorted list of 1 element on the left and \( N-1 \) unsorted items on the right
▶ take the first unsorted item
▶ insert it into the sorted list, moving elements as necessary
▶ now have a sorted list of size 2, and \( N -2 \) unsorted elements
▶ repeat for all items
Let’s reuse our unsorted list from before and sort it in ascending order:

Start by comparing the first two items
Insertion sort #3

We find that both 14 and 33 are already in ascending order.
▶ for now, 14 is in sorted sub-list

Insertion sort then moves ahead and compares 33 with 27
Insertion sort #4

33 is not in the correct position

Swap 33 with 27
  ▶ also check that all the elements of sorted sub-list
  ▶ we see that the sorted sub-list has only one element 14
  ▶ 27 is greater than 14
  ▶ therefore, the sorted sub-list remains sorted after swapping
Now that we have 14 and 27 in the sorted sub-list

- compare 33 with 10

Values are not in a sorted order

So we swap them
Insertion sort #6

However, swapping makes 27 and 10 unsorted

We swap them too

We find 14 and 10 in an unsorted order
Insertion sort #7

We swap them again
- by the end of third iteration, we have a sorted sub-list of 4 items

This process goes on until all the unsorted values are covered in a sorted sub-list.
Insertion sort algorithm

**Insertion sort** (*sequence*)

- **Step 1** - If it is the first element, item is already sorted
- **Step 2** - select next item
- **Step 3** - compare against all other items in the sorted sub-list
- **Step 4** - shift all the elements in the sorted sub-list that are greater than the value to be sorted
- **Step 5** - Insert the value in the sorted sub-list
- **Step 6** - repeat until list is sorted
Algorithm 2 Insertion sort

1: procedure INSERTION_SORT(sequence)
2:     for i ← 1 to N − 1 do
3:         key ← sequence[i]
4:         // inset key into the sorted sub-list
5:         j ← i − 1
6:         while j ≥ 0 and sequence[j] > key do
7:             sequence[j + 1] ← sequence[j]
8:             j ← j − 1
9:         end while
10:     sequence[j + 1] ← key
11: end for
12: end procedure
insertion sort: python implementation

```python
import random

sequence = list(range(0,10))
random.shuffle(sequence)
N = len(sequence)
for i in range(1,N,1):
    j = i-1
    key = sequence[i]
    while(j >= 0 and sequence[j] > key):
        sequence[j+1] = sequence[j]
        j -= 1
    sequence[j+1] = key
print(sequence)  # prints ???
```
Summary

Why learn both selection and insertion sort?

▶ insertion sort is expected to be faster
▶ selection sort makes more comparisons than movements
  ▶ insertion sort is the opposite
▶ if less movement is needed
  ▶ e.g., list is almost sorted
  ▶ then selection sort is the better choice

Question: based on the algorithms you have already learned, how could you further improve insertion sort?