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 for selection sort are taken from an 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
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
Selection sort: pseudocode

**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: \( \text{min\_index} \leftarrow i \)
5: **for** \( j \leftarrow i + 1 \) **to** \( N - 1 \) **do**
6: \[ \text{if } *sequence*[j] \leq *sequence*[\text{min\_index}] \text{ then} \]
7: \( \text{min\_index} \leftarrow j \)
8: **end if**
9: **end for**
10: **SWAP**(\( *sequence*[i],*sequence*[\text{min\_index}] \))
11: **end for**
12: **end procedure**
def selection_sort(sequence):
    N = len(sequence)
    for i in range(0,N):
        min_index = i
        for j in range(i+1,N):
            if sequence[j] <= sequence[min_index]:
                min_index = j
        sequence[i],sequence[min_index] = \
        sequence[min_index],sequence[i]
    return sequence
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
Insertion sort #2

Let’s reuse our unsorted list from before and sort it in ascending order:

\[
\begin{array}{cccccccc}
14 & 33 & 27 & 10 & 35 & 19 & 42 & 44 \\
\end{array}
\]

Start by finding out where to insert 33:

33

14 > 33? no

\[
\begin{array}{cccccccc}
14 & & 27 & 10 & 35 & 19 & 42 & 44 \\
\end{array}
\]

put 33 back
Insertion sort #3

27

33 > 27? yes

14 33 10 35 19 42 44

14 > 27? no

14 33 10 35 19 42 44

insert 27 at index 1

14 27 33 10 35 19 42 44

Sorted so far
Insertion sort #4

10

33 > 10? yes

14 27 33 35 19 42 44

27 > 10? yes

14 27 33 35 19 42 44

14 > 10? yes

14 27 33 35 19 42 44

insert 10 at position 0

10 14 27 33 35 19 42 44
Insertion sort #5

19

35 > 19? yes

33 > 19? yes

27 > 19? yes

14 > 19? no

insert 19 at position 2

10 14 19 27 33 35 42 44
Insertion sort #5

Sorted so far

10 14 19 27 33 35 42 44

35 > 42? no

10 14 19 27 33 35 44

42

42 > 44? no

10 14 19 27 33 35 42 44

put back 42

Sorted so far

10 14 19 27 33 35 44

44

42 > 44? no

10 14 19 27 33 35 42 44

put back 42

Completely
Insertion sort algorithm

i ← 3, key ← 10
j ← i - 1

sequence[j] > key? True

sequence[j+1] ← sequence[j]

j ← j - 1

sequence[j] > key? True

sequence[j+1] ← sequence[j]
Insertion sort: pseudocode

Algorithm 2 Insertion sort

1: procedure INSERTION_SORT(sequence)
2: for $i \leftarrow 1$ to $N$ do
3:  $key \leftarrow sequence[i]$
4:  // inset key into the sorted sub-list
5:  $j \leftarrow i - 1$
6:  while $j \geq 0$ and $sequence[j] > key$ do
7:    $sequence[j + 1] \leftarrow sequence[j]$
8:    $j \leftarrow j - 1$
9:  end while
10: $sequence[j + 1] \leftarrow key$
11: end for
12: end procedure
def insertion_sort(sequence):
    N = len(sequence)
    for i in range(1, N):
        j = i - 1
        key = sequence[i]
        while (j >= 0 and sequence[j] > key):
            sequence[j + 1] = sequence[j]
            j -= 1
        sequence[j + 1] = key
    return sequence
Which sorting algorithm is faster?

```python
def selection_sort(sequence):
    N = len(sequence)
    for i in range(0,N):
        min_index = i
        for j in range(i+1,N):
            if sequence[j] <= sequence[min_index]:
                min_index = j
        sequence[i],sequence[min_index] = sequence[min_index],sequence[i]
    return sequence
```

```python
def insertion_sort(sequence):
    N = len(sequence)
    for i in range(1,N):
        j = i-1
        key = sequence[i]
        while(j >= 0 and sequence[j] > key):
            sequence[j+1] = sequence[j]
            j -= 1
        sequence[j+1] = key
    return sequence
```
Why learn both selection and insertion sort?

- Insertion sort is expected to be faster in practice.
- Selection sort makes $n$ passes to scan each element.
  - Insertion sort only scans each element (being sorted) once.
- Insertion sort does more swapping (i.e., memory usage) than selection sort at the worst case.
- There are faster sorting algorithms: e.g., mergesort, quicksort (beyond the scope of this class).