Sorting Algorithms
Objectives

- Examine different sorting algorithms that can be implemented in-place (without the use of auxiliary collections) and using auxiliary collections.
Sorting Problem

• Consider an unordered list of $n$ objects that we wish to have sorted into ascending order.

• We will study the following sorting algorithms:
  • *Insertion sort* using stacks and in-place
  • *Selection sort* using queues and in-place
  • *Quick Sort*
Insertion Sort

• **Insertion Sort** orders a sequence of values by repeatedly taking each value and inserting it in its proper position within a *sorted subset* of the sequence.

• More specifically:
  
  • Consider the first item to be a *sorted subsequence* of length 1
  
  • Insert the second item into the *sorted subsequence*, now of length 2
  
  • Repeat the process for each item, always inserting it into the current *sorted subsequence*, until the entire sequence is in order
Value **5** is to be inserted in the sorted sequence to its left. Since 5 is smaller than 8, then 8 needs to be shifted one position to the right and then 5 can be inserted on the first position of the array.
2 will be inserted here

6 will be inserted here
9 is already in its correct position

4 will be inserted here
6 will be inserted here

And we’re done!
Insertion Sort using Stacks

- Use two temporary stacks called sorted and temp, both of which are initially empty.
- The contents of sorted will always be in order, with the smallest item on the top of the stack.
  - This will be the "sorted subsequence".
- temp will temporarily hold items that need to be "shifted" out in order to insert the new item in the proper place in stack sorted.
**Algorithm** insertionSort (A, n)

**In:** Array A storing n elements

**Out:** Sorted array

sorted = empty stack

temp = empty stack

for i = 0 to n-1 do {
    while (sorted is not empty) and (sorted.peek() < A[i]) do
        temp.push (sorted.pop())
    sorted.push (A[i])
    while temp is not empty do
        sorted.push (temp.pop())
}

for i = 0 to n-1 do
    A[i] = sorted.pop()

return A
Insertion Sort

8 5 2 6 9 4 6

sorted temp
Insertion Sort

sorted 8

temp
Insertion Sort

sorted temp

8 5 2 6 9 4 6

sorted

5 8

temp
Insertion Sort
Insertion Sort

8 5 2 6

sorted

2 5 8

temp
Insertion Sort

8 5 2 6 9 4 6

Sorted: 8 5 2 6
Temp: 5 8 2
Insertion Sort
Insertion Sort

sorted

6
8

temp

5
2
Insertion Sort

sorted:

8 5 2 6

temp:

5 6 8

2
Insertion Sort

sorted

8 5 2 6

temp

2 5 6 8

13-20
Insertion Sort

8 5 2 6 9 4 6

sorted

temp

2 5 6 8
Insertion Sort
Insertion Sort

sorted: 8 5 2 6 9 4 6

temp: 6 8 5 2
Insertion Sort

sorted: 8, 5, 2, 6, 9, 4, 6

temp: 6, 5, 2
Insertion Sort

![Diagram of Insertion Sort]

Sorted: 8, 5, 2, 6, 9, 4, 6

Temp: 8, 6, 5, 2
Insertion Sort

sorted: 9

temp: 8, 6, 5, 2
Insertion Sort

8 5 2 6 9 4 6

sorted

8 9

temp

6 5 2
Insertion Sort

sorted: 8 5 2 6 9

temp: 6 8 9

temp: 5 2
Insertion Sort

8 5 2 6 9

sorted

5 6 8 9
temp

2
Insertion Sort

8 5 2 6 9 4 6

sorted temp
Insertion Sort

... and so on until all values are stored and ordered in stack sorted.
Insertion Sort

Now, copy the values back into the array…
Insertion Sort

Now, copy the values back into the array…
Insertion Sort

Now, copy the values back into the array…
In-Place Insertion Sort

**In-Place:** the algorithm does not use any auxiliary data structures.

Consider the next value: 5
In-Place Insertion Sort

Shift 8 to make room for 5
In-Place Insertion Sort

Consider the next value: 2
In-Place Insertion Sort

Shift 8 and 5 to the right
In-Place Insertion Sort

Consider the next value: 6
In-Place Insertion Sort

Shift 8 to the right
In-Place Insertion Sort

9 is already in its correct position
In-Place Insertion Sort

Consider the next value: 4
In-Place Insertion Sort

Shift 5, 6, 8, 9 to the right and insert 4 in the second position
In-Place Insertion Sort

Finally, consider the last value: 6
Shift 8 and 9 to the right and insert 6 in the fifth position. The array is sorted!
Algorithm *insertionSort* \((A,n)\)

**In:** Array \(A\) storing \(n\) values  
**Out:** \{Sort \(A\) in increasing order\}

```plaintext
for \(i = 1\) to \(n-1\) do {
    // Insert \(A[i]\) in the sorted sub-array \(A[0..i-1]\)
    temp = A[i]
    j = i - 1
    while \((j \geq 0)\) and \((A[j] > temp)\) do {
        j = j - 1
    }
    A[j+1] = temp
}
```
Selection Sort

- **Selection Sort** orders a sequence of values by repetitively putting a particular value into its *final* position.

- More specifically:
  - Find the *smallest value* in the sequence.
  - Switch it with the value in the *first position*.
  - Find the *next smallest value* in the sequence.
  - Switch it with the value in the *second position*.
  - Repeat until all values are in their proper places.
Selection Sort Algorithm

Initially, the entire array is the “unsorted portion”

The sorted portion is in red.

Find the smallest element in the unsorted portion of the array

Interchange the smallest element with the one at the first position of the array

Find the smallest element in the unsorted portion of the array
Interchange the smallest element with the one at the second position

Find the smallest element in the unsorted portion

Interchange the smallest element with the one at the third position
Find the smallest element in the unsorted portion

Interchange the smallest element with the one at the fourth position

After $n-1$ repetitions of this process, the last item has automatically fallen into place!
Selection Sort Using a Queue

• Create a queue called sorted, initially empty, to hold the items that have been sorted so far
• The contents of sorted will always be in order, with new items added at the end of the queue
Selection Sort Using Queue Algorithm

• While the unordered list list is not empty:
  • *remove* the smallest item from list and *enqueue* it to the end of sorted

• At the end of the while loop the list is empty, and sorted contains the items in ascending order, from front to rear

• To restore the original list, *dequeue* the items one at a time from sorted, and *add them to* list
Algorithm selectionSort(list)
In: Unsorted list
Out: Sorted list

sorted = empty queue

\( n = \text{number of data items in } list \)

while list is not empty do {
    \( \text{smallestSoFar} = \text{get first item in } list \)
    for \( i = 1 \) to \( n - 1 \) do {
        item = get item in the \( i \)-th position of list
        if item < smallestSoFar then smallestSoFar = item
    }
    sorted.enqueue(smallestSoFar)
    remove smallestSoFar from list
    \( n = n - 1 \)
}

for \( i = 0 \) to \( n - 1 \) do
    insert sorted.dequeue() in the \( i \)-th position of list

return list
In-Place SelectionSort

Selection sort without using any additional data structures. Assume that the values to sort are stored in an array.
In-Place SelectionSort

First, find the smallest value
In-Place SelectionSort

Swap it with the element in the first position of the array.
In-Place SelectionSort

Swap it with the element in the first position of the array.
In-Place Selection Sort
In-Place SelectionSort

Now consider the rest of the array and again find the smallest value.
In-Place SelectionSort

Swap it with the element in the second position of the array, and so on.
In-Place Selection Sort
In-Place Selection Sort

sorted

smallest value
In-Place SelectionSort

sorted

swap

smallest value
In-Place SelectionSort
In-Place SelectionSort

sorted

smallest value
In-Place SelectionSort
**Algorithm selectionSort** \((A, n)\)

**In:** Array \(A\) storing \(n\) values

**Out:** {Sort \(A\) in increasing order}

for \(i = 0\) to \(n-2\) do {

    // Find the smallest value in unsorted subarray \(A[i..n-1]\)
    \(\text{smallest} = i\)

    for \(j = i + 1\) to \(n - 1\)do {

        if \(A[j] < A[\text{smallest}]\) then
            \(\text{smallest} = j\)

    }

    // Swap \(A[\text{smallest}]\) and \(A[i]\)
    \(\text{temp} = A[\text{smallest}]\)
    \(A[\text{smallest}] = A[i]\)
    \(A[i] = \text{temp}\)

}
Quick Sort

- *Quick Sort* orders a sequence of values by partitioning the list around one element (called the *pivot* or *partition element*), then sorting each partition
- More specifically:
  - Choose one element in the sequence to be the *pivot*
  - Organize the remaining elements into three groups (*partitions*): those *greater than* the *pivot*, those *less than* the *pivot*, and those *equal* to the *pivot*
  - Then sort each of the first two partitions (recursively)
Quick Sort

*Partition element* or *pivot*:

- The choice of the *pivot* is arbitrary
- For efficiency, it would be nice if the pivot divided the sequence roughly in half
  - However, the algorithm will work in any case
Quick Sort

• We put all the items to be sorted into a container (e.g. an array)
• We choose the pivot (partition element) as the first element from the container
• We use a container called smaller to hold the items that are smaller than the pivot, a container called larger to hold the items that are larger than the pivot, and a container called equal to hold the items of the same value as the pivot
• We then recursively sort the items in the containers smaller and larger
• Finally, copy the elements from smaller back to the original container, followed by the elements from equal, and finally the ones from larger
QuickSort

6 3 2 6 9 4 8
QuickSort

pivot or partition element

smaller

larger

equal
QuickSort

pivot or partition element

6 3 2 6 9 4 8

smaller

Put 6 in this container

equal

larger

6
QuickSort

scan the array and place the values in the proper container

pivot or partition element

smaller

larger

equal
QuickSort

scan the array and place the values in the proper container

pivot or partition element

smaller

larger

equal

6 3 2 6 9 4 8

3 2

6
QuickSort

scan the array and place the values in the proper container

pivot or partition element

smaller

larger

equal
QuickSort

scan the array and place the values in the proper container

pivot or partition element

smaller

larger

equal
QuickSort

scan the array and place the values in the proper container

pivot or partition element

smaller

larger

equal
QuickSort

Now sort this list
QuickSort

```
| 6 | 3 | 2 | 6 | 9 | 4 | 8 |
```

- **smaller**
  - 2
  - 3
  - 4

- **larger**
  - 9
  - 8

- Sorted!

Sorted!
QuickSort

Next sort this list
QuickSort

6 3 2 6 9 4 8

smaller
2 3 4

larger
8 9

equal
6 6

Sorted!
QuickSort

Copy data back to original list

smaller

2 3 4

larger

8 9

equal

6 6
QuickSort

Copy data back to original list

smaller

larger

equal
QuickSort

Copy data back to original list

smaller

larger

equal
QuickSort

Copy data back to original list

smaller

larger

2 3 4

8 9

6 6

6 6
QuickSort

Sorted!

smaller

larger

equal
QuickSort

How to sort this list?
QuickSort

select a pivot

3 2 4

smaller

equal

larger
QuickSort

Scan array and put the values in the containers

smaller

larger

equal
QuickSort

sort the lists

smaller
default.png

equal
default.png

larger
default.png
QuickSort

smaller

larger

copy data back

equal
Algorithm quicksort(A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
If n > 1 then {
    smaller, equal, larger = new arrays of size n
    \( n_s = n_e = n_l = 0 \)
    pivot = A[0]
}

\[ \begin{array}{ccc}
    & 3 & 2 & 4 \\
    A
    \end{array} \]

pivot = 3

\[ \begin{array}{ccc}
    & & \\
    smaller
    \end{array} \]
\( n_s = 0 \)

\[ \begin{array}{ccc}
    & & \\
    equal
    \end{array} \]
\( n_e = 0 \)

\[ \begin{array}{ccc}
    & & \\
    larger
    \end{array} \]
\( n_l = 0 \)
Algorithm quicksort(A,n)

In: Array A storing n values

Out: {Sort A in increasing order}

If n > 1 then {
    smaller, equal, larger = new arrays of size n
    $n_s = n_e = n_l = 0$
    pivot = A[0]
    for i = 0 to n-1 do  // Partition the values
        // Partition the values
**Algorithm** quicksort(A,n)

**In:** Array A storing n values

**Out:** {Sort A in increasing order}

**If** n > 1 **then** {

smaller, equal, larger = new arrays of size n

n_s = n_e = n_l = 0

pivot = A[0]

**for** i = 0 **to** n-1 **do** // Partition the values

  **if** A[i] = pivot **then** equal[n_e++] = A[i]

}

**Diagram:**

```
   A  3  2  4

   i

   pivot = 3

smaller

   n_s=0

equal

   3

   n_e=1

larger

   n_l=0
```
Algorithm quicksort(A,n)

In: Array A storing n values

Out: {Sort A in increasing order}

If n > 1 then {
    smaller, equal, larger = new arrays of size n
    \( n_s = n_e = n_l = 0 \)
    pivot = A[0]
    for i = 0 to n-1 do // Partition the values
        if A[i] = pivot then equal[\( n_e++ \)] = A[i]
        else if A[i] < pivot then smaller[\( n_s++ \)] = A[i]
}

A

\[ \begin{array}{c}
\text{3} & \text{2} & \text{4} \\
\end{array} \]

smaller
\[ \begin{array}{c}
\text{2} \\
\end{array} \]

\( n_s = 1 \)

equal
\[ \begin{array}{c}
\text{3} \\
\end{array} \]

\( n_e = 1 \)

larger
\[ \begin{array}{c}
\text{} \\
\end{array} \]

\( n_l = 0 \)
Algorithm quicksort(A,n)
In: Array A storing n values
Out: {Sort A in increasing order}
If n > 1 then {
    smaller, equal, larger = new arrays of size n
    n_s = n_e = n_l = 0
    pivot = A[0]
    for i = 0 to n-1 do  // Partition the values
        if A[i] = pivot then equal[n_e++] = A[i]
        else if A[i] < pivot then smaller[n_s++] = A[i]
        else larger[n_l++] = A[i]
}
**Algorithm** quicksort(A,n)

**In:** Array A storing n values

**Out:** {Sort A in increasing order}

**If** n > 1 **then** {
    smaller, equal, larger = new arrays of size n
    \( n_s = n_e = n_l = 0 \)
    pivot = A[0]
    for i = 0 to n-1 do  // Partition the values
        if A[i] = pivot then equal[\( n_e \)+1] = A[i]
        else if A[i] < pivot then smaller[\( n_s \)+1] = A[i]
        else larger[\( n_l \)+1] = A[i]
    quicksort(smaller,n_s)
}

\[A = \begin{array}{ccc}
3 & 2 & 4 \\
\end{array}\]

\[\begin{array}{ccc}
\text{smaller} & 2 & n_s = 1 \\
\text{equal} & 3 & n_e = 1 \\
\text{larger} & 4 & n_l = 1 \\
\end{array}\]
**Algorithm** quicksort(A,n)

**In**: Array A storing n values

**Out**: {Sort A in increasing order}

If n > 1 then {

smaller, equal, larger = new arrays of size n

n_s = n_e = n_l = 0

pivot = A[0]

for i = 0 to n-1 do // Partition the values

if A[i] = pivot then equal[n_e++] = A[i]

else if A[i] < pivot then smaller[n_s++] = A[i]

else larger[n_l++] = A[i]

quicksort(smaller, n_s)

quicksort(larger, n_l)

}

A

| 3 | 2 | 4 |

smaller

| 2 |

n_s = 1

equal

| 3 |

n_e = 1

larger

| 4 |

n_l = 1

Sort
**Algorithm** quicksort(A,n)

**In:** Array A storing n values  
**Out:** {Sort A in increasing order}

If n > 1 then {
    smaller, equal, larger = new arrays of size n  
    \( n_s = n_e = n_l = 0 \)
    pivot = A[0]
    for i = 0 to n-1 do  // Partition the values
        if A[i] = pivot then equal[n_e++] = A[i]
        else if A[i] < pivot then smaller[n_s++] = A[i]
        else larger[n_l++] = A[i]
    quicksort(smaller,n_s)
    quicksort(larger,n_l)
    i = 0
    for j = 0 to n_s do A[i++] = smaller[j]
    for j = 0 to n_e do A[i++] = equal[j]
    for j = 0 to n_l do A[i++] = larger[j]
}