Searching and Sorting

• Linear Search
• Binary Search
• Bubble Sort

• Selection Sort
• Insertion Sort
• Working with Sorted Lists
Searching

- The process of selecting particular information from a collection of data based on specific criteria.
  - Can be performed on different data structures.
  - **sequence search** – search within a sequence.
  - **search key** (or key) – identifies a specific item.
  - **compound key** – consists of multiple parts.
Linear Search

- Iterates over the sequence, item by item, until the specific item is found or the list is exhausted.
- The simplest solution to sequence search problem.
- Python's `in` operator: find a specific item.

```python
if key in theArray :
    print( 'The key is in the array.' )
else :
    print( 'The key is not in the array.' )
```
Linear Search Examples

- Searching for 31

- Searching for 8
Linear Search Code

def linearSearch( theValues, target ) :  
n = len( theValues )  
for i in range( n ) :  
    if theValues[i] == target  
        return True
Linear Search: Sorted Sequence

- A linear search can be performed on a sorted sequence.
- Example: searching for 8.
Linear Search: Sorted Sequence

- Similar to the unsorted sequence, with one major difference.

```python
def sortedLinearSearch( theValues, target ) :
    n = len( theValues )
    for i in range( n ) :
        if theValues[i] == target :
            return True
        elif theValues[i] > target :
            return False
    return False
```
Linear Search: Smallest Value

- We can search for an item based on certain criteria.
- Example: find the smallest value.

```python
def findSmallest( theValues ):
    n = len( theValues )
    smallest = theValues[0]
    for i in range( 1, n ) :
        if theList[i] < smallest :
            smallest = theValues[i]

    return smallest
```
Binary Search

- The linear search has a linear time-complexity.
  - We can improve the search time if we modify the search technique itself.
  - Use a **divide and conquer** strategy.
  - Requires a sorted sequence.

![Sorted sequence diagram](image.png)
Binary Search Algorithm

- Examine the middle item (searching for 10):

![Binary Search Diagram]

- One of three possible conditions:
  - target is found in the middle item.
  - target is less than the middle item.
  - target is greater than the middle item.
Binary Search Algorithm

- Since the sequence is sorted, we can eliminate half the values from further consideration.
Binary Search Algorithm

- Repeat the process until either the target is found or all items have been eliminated.
def binarySearch( theValues, target ) :
    low = 0
    high = len(theValues) - 1

    while low <= high :
        mid = (high + low) // 2
        if theValues[mid] == target :
            return True
        elif target < theValues[mid] :
            high = mid - 1
        else :
            low = mid + 1

    return False
Binary Search Implementation

Diagram showing the process of binary search with a list of numbers and indices labeled as low, mid, and high.
Sorting

- The process of arranging a collection of items such that each item and its successor satisfy a prescribed relationship.
  - items can be simple types or complex types
  - **sort key** – values on which items are ordered.
  - items arranged in ascending or descending order.
Bubble Sort

- A simple solution to the sorting problem.
- Arranges the items by
  - iterating over the sequence multiple times.
  - larger values bubble to the top (or end).
def bubbleSort( theSeq ):
    n = len( theSeq )
    for i in range( n - 1 ) :
        for j in range( n - 1 - i ) :
            if theSeq[j] > theSeq[j + 1] :
                tmp = theSeq[j]
                theSeq[j] = theSeq[j + 1]
                theSeq[j + 1] = tmp
Bubble Sort Example

- First complete iteration of the inner loop.
Bubble Sort Example
Bubble Sort Example

- Results after each iteration of the outer loop.

```
Results after each iteration of the outer loop.
```

```
10 2 18 4 31 13 5 23 51 29 64
```

```
2 10 4 18 13 5 23 31 29 51 64
```

```
2 4 10 13 5 18 23 29 31 51 64
```

```
2 4 10 5 13 18 23 29 31 51 64
```

```
2 4 5 10 13 18 23 29 31 51 64
```

```
2 4 5 10 13 18 23 29 31 51 64
```
Bubble Sort Example
Bubble Sort

• One of the most inefficient sorting algorithm.
• Always performs $n^2$ iterations of the inner loop.
• Consider:
  • an array of keys in reverse order
  • a sorted sequence
Selection Sort

• Improves on the bubble sort.

• Works in a fashion similar to what a human may use to sort a sequence.

• Instead of swapping many items,
  • requires a search to select the smallest item.
  • makes a single swap after each pass
def selectionSort( theSeq ):
    n = len( theSeq )
    for i in range( n - 1 ):
        smallNdx = i
        for j in range( i + 1, n ):
            if theSeq[j] < theSeq[smallNdx] :
                smallNdx = j

    if smallNdx != i :
        tmp = theSeq[i]
        theSeq[i] = theSeq[smallNdx]
        theSeq[smallNdx] = tmp
Selection Sort Example
Selection Sort Example
Insertion Sort

- Another commonly studied algorithm.
- Arranges the items by
  - iterating over the sequence one complete time.
  - inserts each unsorted item into its proper place.
- To position an item:
  - find the correct spot within the sorted sequence
  - open the slot by shifting the items down one position
Insertion Sort Code

```python
def insertionSort( theSeq ):
    n = len( theSeq )
    for i in range( 1, n ) :
        value = theSeq[i]

        pos = i
        while pos > 0 and value < theSeq[pos - 1] :
            theSeq[pos] = theSeq[pos - 1]
            pos -= 1

        theSeq[pos] = value
```
Insertion Sort Example
Insertion Sort Example
Working With Sorted Lists

- The efficiency of some algorithms can be improved when working with sorted sequences. eg. Binary Search
  - For non-static collections, it would be inefficient to re-sort a sequence for each add/remove.
  - Better to maintain a sorted sequence.

Static collection: no items will be added/removed to the sequence
Maintaining a Sorted List

- To maintain a sorted list, new items must be inserted into their proper position.
  - Can not simply be appended at the end.
  - Must locate the proper position and use `insert()`.
Modified Binary Search

A modified version of the binary search can be used to find the proper location of an item.

```python
def findSortedPosition(theList, target):
    low = 0
    high = len(theList) - 1
    while low <= high:
        mid = (high + low) // 2
        if theList[mid] == target:
            return mid  # Index of the target.
        elif target < theList[mid]:
            high = mid - 1
        else:
            low = mid + 1

    # Index where the target value should be.
    return low
```
Merging Sorted Lists

- Sometimes it may be necessary to merge two sorted lists into a new list.
- For example

```python
listA = [ 2, 8, 15, 23, 37 ]
listB = [ 4, 6, 15, 20 ]
newList = mergeSortedLists( listA, listB )
print( newList )
```

creates a new merged list

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```
Merging Algorithm

- We can use an approach similar to what you might do, if merging two stacks of cards by hand.
  - Examine the value of the top card.
  - Select the smallest of the two.
  - Flip it over and place it on top of a third stack.
  - Repeat until one of the two original stacks is empty.
  - Finally, flip all of the cards in the remaining stack onto the top of the new stack.
Merging Example

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]

\[\text{ListA} \begin{array}{cccc} 2 & 8 & 15 & 23 \end{array} \quad \text{ListB} \begin{array}{cccc} 4 & 6 & 15 & 20 \end{array} \quad \text{newList} \begin{array}{c} 2 \end{array}\]
Merging Example

ListA

2 8 15 23 37

ListB

4 6 15 20

newList

2 4 6 8 15
Merging Implementation

```python
def mergeSortedLists( listA, listB ) :
    newList = list()
    a = 0
    b = 0
    while a < len( listA ) and b < len( listB ) :
        if listA[a] < listB[b] :
            newList.append( listA[a] )
            a += 1
        else :
            newList.append( listB[b] )
            b += 1
    while a < len( listA ) :
        newList.append( listA[a] )
        a += 1
    while b < len( listB ) :
        newList.append( listB[b] )
        b += 1
    return newList
```
Merging Analysis

- Assume \texttt{listA} and \texttt{listB} each contain \textit{n} items.
- The worst case depends on the number of iterations performed by all three loops combined.
  - Max iterations of the 1st loop:
    - occurs when the selection alternates between the two lists.
    - $2n - 1$ iterations + 1 iteration of either of the next two loops.
  - Min iterations of the 1st loop:
    - occurs when all values of one list are selected in order.
    - $n$ iterations + $n$ iterations of either of the next two loops.