Advanced Sorting

• Merge Sort
• Quick Sort
• Radix Sort
Sorting Algorithms

- Can be divided into two categories:
  - comparison sorts
    - items are arranged by performing pairwise logical comparisons between two sort keys.
  - distribution sorts
    - distributes the sort keys into intermediate groups based on individual key values.
Merge Sort

- Uses a divide and conquer strategy to sort the keys stored in a sequence.
  - Keys are recursively divided into smaller and smaller subsequences.
  - Subsequences are merged back together.
Merge Sort – Divide

- Starts by splitting the original sequence in the middle to create two subsequences of approximately equal size.
Merge Sort – Divide

- The two subsequences are then split in the middle.
Merge Sort – Divide

- The subdivision continues until there is a single item in the sequence.

![Diagram of Merge Sort – Divide process with examples of numbers being divided and recombined.](image)
Merge Sort – Conquer

- After the sequences are split, they are merge back together, two at a time to create sorted sequences.
**Merge Sort Code #1**

- A simple implementation for sorting a Python list.

```python
def pythonMergeSort( theList ):  
    # Check the base case.  
    if len(theList) <= 1 :  
        return theList  
    else :  
        # Compute the midpoint.  
        mid = len(theList) // 2  

        # Split the list and perform the recursive step.  
        leftHalf = pythonMergeSort( theList[ :mid ] )  
        rightHalf = pythonMergeSort( theList[ mid: ] )  

        # Merge the two ordered sublists.  
        newList = mergeSortedLists( leftHalf, rightHalf )  
        return newList
```
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
Merge Sort – Improved Version

- The previous version:
  - only works with Python lists.
  - the splitting creates new physical lists.
  - uses the splice operation which is time consuming.
Merge Sort – Improved Version

- We can improve the implementation:
  - using virtual subsequences.
  - that works with any sequence.
An improved version of the merge sort.

```python
def recMergeSort( theSeq, first, last, tmpArray ):
    # Check the base case.
    if first == last :
        return;
    else :
        # Compute the mid point.
        mid = (first + last) // 2
        # Split the sequence and perform the recursive step.
        recMergeSort( theSeq, first, mid, tmpArray )
        recMergeSort( theSeq, mid+1, last, tmpArray )
        # Merge the two ordered subsequences.
        mergeVirtualSeq( theSeq, first, mid+1, last+1, tmpArray )
```
def mergeVirtualSeq( theSeq, left, right, end, tmpArray ):
a = left
b = right
m = 0

while a < right and b < end :
    if theSeq[a] < theSeq[b] :
        tmpArray[m] = theSeq[a]
a += 1
    else :
        tmpArray[m] = theSeq[b]
b += 1
m += 1
Merging Sorted Sequences

```python
while a < right :
    tmpArray[m] = theSeq[a]
    a += 1
    m += 1

while b < end :
    tmpArray[m] = theSeq[b]
    b += 1
    m += 1

for i in range( end - left ) :
    theSeq[i+left] = tmpArray[i]
```
Merge Sort – Temporary Array

- A temporary array is used to merge two virtual subsequences.

![Diagram of Merge Sort with Temporary Array](image)

The two virtual subsequences are merged into the temporary array.

The elements are copied from the temporary array back into the original sequence.
Wrapper Functions

- A function that provides a simpler and cleaner interface for another function.
  - Provides little or no additional functionality.
  - Commonly used with recursive functions that require additional arguments.

```python
def mergeSort( theSeq ):
    n = len( theSeq )
    tmpArray = Array( n )
    recMergeSort( theSeq, 0, n-1, tmpArray )
```
Merge Sort – Efficiency

- We need to determine the number of invocations and the time required by each function.
Merge Sort – Efficiency

- Consider a sequence of $n$ items.
**Quick Sort**

- Uses a divide and conquer strategy to sort the keys stored in a sequence.
  - Partitions the sequence by dividing it into two segments based on a **pivot key**.
  - Uses virtual subsequences without the need for temporary storage.

- Quick sort is a recursive algorithm.
Quick Sort – Description

- Select the first key as the pivot (p)
- Partition the sequence into segments L and G.
  - L contains all keys less than p
  - G contains all keys greater than or equal to p.
- Recursively apply the same operation on L & G.
  - Continues until the sequence contains 0 or 1 key.
- Merge the pivot and two segments back together.
Quick Sort – Divide
Quick Sort – Merge
Quick Sort – Implementation

- An efficient solution can be designed.

```python
def quickSort(theSeq):
    n = len(theSeq)
    recQuickSort(theSeq, 0, n-1)

def recQuickSort(theSeq, first, last):
    if first >= last:
        return
    else:
        # Partition the sequence and obtain the pivot position.
        pos = partitionSeq(theSeq, first, last)

        # Repeat the process on the two subsequences.
        recQuickSort(theSeq, first, pos - 1)
        recQuickSort(theSeq, pos + 1, last)
```
Quick Sort – Partition

- The partitioning step can be done without having to use temporary storage.
  - Rearranges the keys within the sequence structure.
  - The pivot will be in its correct position within the sequence.
  - Position of the pivot indicates the position where the split occurred.
Quick Sort – Partition

- For illustration, we step through the first complete partitioning.
  - Pivot value is the first key in the segment.
  - Two markers *(left and right)* are initialized.

- The markers will be shifted left and right until they cross each other.
Quick Sort – Partition

- The **left** marker is shifted right until a key value larger than the pivot is found.

- The **right** marker is then shifted left until a key value less than the pivot is found.
Quick Sort – Partition

- The two keys at the positions of the left and right markers are swapped.
Quick Sort – Partition

- The two markers are again shifted starting where they left off.
Quick Sort – Partition

- After the markers are shifted, the corresponding keys are swapped as before.
Quick Sort – Partition

- The shifting and swapping continues until the two markers cross each other.
Quick Sort – Partition

- When the two markers cross, the right marker indicates the final position of the pivot value.
- The pivot value and the value at the right marker have to be swapped.
Quick Sort – Partition

def partitionSeq( theSeq, first, last ):
    pivot = theSeq[first]
    left = first + 1
    right = last
    while left <= right :
        while left <= right and theSeq[left] < pivot :
            left += 1
        while right >= left and theSeq[right] >= pivot :
            right -= 1
        if left < right :
            tmp = theSeq[left]
            theSeq[left] = theSeq[right]
            theSeq[right] = tmp
        if right != first :
            theSeq[first] = theSeq[right]
            theSeq[right] = pivot
    return right
Pivot Key

- We are not limited to selecting the first key within the sequence as the pivot.
  - Using the first or last key is a poor choice in practice.
  - Choosing a key near the middle is a better choice.
Quick Sort – Efficiency

- The quick sort algorithm:
  - has an average case time of $O(n \log n)$
- It does not require additional storage.
- Commonly used in language libraries.
  - Earlier versions of Python used quick sort.
  - Current versions use a hybrid that combines the insertion and merge sort algorithms.
Radix Sort

- A fast distribution sorting algorithm.
  - Special purpose sorting algorithm.
  - Orders keys by examining individual key components instead of comparing them.
  - When used with integers, individual key digits are compared from least to most significant.
  - *aka* bin sort
    - dates back to the days of card readers.
Radix Sort Example

- Bins are used to store various keys based on individual column values.
  - Consider an array of positive integers.
  - We will need 10 bins, one for each digit.

![Array of positive integers](image)
The process starts by distributing the keys among the various bins.
- Based on the digits in the ones column.
- Stored in the order they occur in the sequence.

<table>
<thead>
<tr>
<th>Bin</th>
<th>Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>51, 31</td>
</tr>
<tr>
<td>2</td>
<td>62</td>
</tr>
<tr>
<td>3</td>
<td>23, 13</td>
</tr>
<tr>
<td>4</td>
<td>54</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>18, 48, 8</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
</tr>
</tbody>
</table>
Gather the Keys

- The keys are then gathered back into the array, one bin at a time.
  - Start with bin 0 and continue in bin order.
  - Gathered without rearranging.
Repeat the Process

- Repeat the process, but this time based on the 10's column.
Radix Sort – Iterations

- The iterative process has to be repeated for the number of digits in the largest value.

*What if we wanted to sort the values in descending order?*
Radix Sort – Implementation

- How should we represent the storage bins?
  - The bins store groups of keys based on the individual digits.
  - Keys with duplicate digits are stored in FIFO in the same bin.
  - When gathered, the keys are stored back into the original sequence structure.
Radix Sort – Implementation

- How do we compare individual integer digits?

\[
digit = (\text{key} \div \text{column}) \mod 10
\]

where \text{column} is the value \((1, 10, 100, \ldots)\) of the column being processed.
def radixSort(intList, numDigits):
    column = 1
    for d in range(numDigits):
        binList = list()
        for k in range(10):
            binList.append(list())

        for key in intList:
            digit = (key // column) % 10
            binList[digit].append(key)

        i = 0
        for bin in binList:
            for key in bin:
                intList[i] = key
                i += 1

        column *= 10
Radix Sort – Analysis

- Assume:
  - a sequence of $n$ keys
  - each key consists of $d$ components
  - each component contains a value between $0$ and $k$. 
Radix Sort – Analysis

- The creation of the array/list: $O(kd)$
- The distribution and gathering: $O(dn)$
  - distribute the $n$ keys across $k$ bins: $O(n)$
  - gathering the $n$ keys back into the sequence: $O(n)$

- Total time: $O(kd + dn)$
  - In practice, $k$ and $d$ are constants.
  - When sorting integers, the time only depends on the number of keys: $O(n)$
Sorting Linked List

- What if we need to sort keys stored in an unsorted linked list?
  - Many of the algorithms used with sequences can be used.
  - Instead of swapping values, nodes are unlinked and relinked as necessary.
Insertion Sort – Linked List

- A simple approach for sorting a linked list:
  - unlink each node, one at a time, from the original unsorted linked list
  - insert each node into a new sorted linked list.

```python
def llistInsertionSort( origList ):
    if origList is None :
        return None

    newList = None
    while origList is not None :
        curNode = origList
        origList = origList.next
        curNode.next = None
        newList = addToSortedList( newList, curNode )

    return newList
```
Insertion Sort – Linked List

- The unlinking/relinking is done in 4 steps:
Insertion Sort – Linked List

- The unlinking/relinking is done in 4 steps:
Insertion Sort – Linked List
Insertion Sort – Linked List

![Insertion Sort Diagram](image.png)
Merge Sort – Linked List

- Is an excellent choice for sorting a linked list.
  - No additional storage space is needed.
  - Efficient in both time and space.
Merge Sort – Linked List

- To split a linked list the node located at the midpoint.
  - Can not directly access this node.
  - Must traverse the list to find the midpoint.
  - Use two temporary external references.
    - Advance one reference, one node at a time.
    - Advance the other, two nodes at a time.
  - Done when the latter falls off the list.
Merge Sort – Linked List
Merge Sort – Linked List

- After the midpoint is located, remove the link between two sublists.
Merging the two sub linked list can be done in a similar fashion as that used with sequences.
- Remove each node from the unsorted lists.
- Insert them into a new sorted linked list.
- Use a tail reference with the new list.