Sorting

Bubble Sort

- Maximum number of passes: number of elements - 1
- For each pass, compare adjacent elements, exchange their places if out of place
- Rear of array is sorted first
    - Largest element put to last place
    - Second largest element put to second last place
    - …
    - Smallest element put to first place
- Process:
    - Pass 0:
        - Compares adjacent elements (A[0], A[1]), (A[1], A[2]) … (A[n-2], A[n-1])
        - For each pair of (A[j], A[j+1]):
            - Exchange their values if A[j] > A[j+1]
            - Set lastExchangeIndex = j
        - Largest element is A[n-1]
        - Front of array (A[0] - A[lastExchangeIndex]) is unordered
        - Rear of array (A[lastExchangeIndex] - A[n-1]) is ordered
    - Subsequent passes:
        - Compare adjacent terms in sublist of (A[0] - A[lastExchangeIndex])
    - Terminates when lastExchangeIndex = 0
- Code:

```
def bubbleSort(array):
    n = len( array )
    isSorted = False                         # create flag

    while not isSorted and n > 0:
        isSorted = True                      # assume sorted

        for i in range( n - 1 ):
            if array[i] > array[i + 1]:      # wrong order
                array = swap( array, i, i+1 )
                isSorted = False
```

```
        n -= 1                              # last element in
place after each pass
    return arr
```

- Benefits:
  - Stable
  - Memory efficient
- Drawback: Inefficient

Insertion Sort

- Process:
  - Pass 0:
    - First element stays at position 0
    - Compare second element A[1] with first A[0]
    - Swap A[0] and A[1] if A[0] > A[1]
  - Subsequent passes:
    - For every target element A[i], compare element down the list of elements (A[i-1], A[i-2] ... A[0])
    - Stop comparison at first element A[j] if (A[j] <= A[i]), or beginning of array (A[0]) is reached
    - Shift every element to the right after comparing (A[j] = A[j-1])
    - Insert target element at correct position (j) after sliding other elements
    - Sublist (A[0] - A[j]) is ordered
- Code:

```
def insertionSort(array):
    for i in range( 1,  len(array) ):
        target = array[i]                       # scans down list
        j = i

        while j > 0 and target < array[j-1]:    # locate insertion
point
            array = swap( array, j, j-1 )       # free up space
to insert
            j -= 1

        array[j] = target                       # insertion
```

return array

- Benefits:
  - Efficient for small sets of data
  - Easily implemented
- Drawbacks: Inefficient on large lists and arrays
- Sample:

Array Index	A[0]	A[1]	A[2]	A[3]	A[4]
Original	50	20	40	75	35
PASS 0	20	50	40	75	35
PASS 1	20	40	50	75	35
PASS 2	20	40	50	75	35
PASS 3	20	35	40	50	75

Quick Sort

- Fastest sorting algorithm
- Uses partition approach to sort array
- Process:
  - Array is sorted
- Simple quick sort code:

```
def quickSort(array):
    if len( array ) <= 1:                # list of 0 or 1 element is already sorted
        return array

    else:
        pivot_value = array.pop(0)       # select & remove pivot value (any index value)
        less = []
        greater = []

        for item in array:               # append each item into appropriate array
```

```
                        if item < pivot_value:
                              less.append( item )
                        else:
                              greater.append( item )

                  return quickSort( less ) + [pivot_value] +
quickSort( greater )
                                                            # note: [pivot_value] is a
list
```

- Hoare's partition code:

```
      def quickSort(A,  low, high):
            if low < high:                              # array has more than
one element
                  pos = partition( A, low, high )  # splits array into two
sublists, pos is final position of pivot
                  quickSort( A, low, pos-1 )        # quick sort lower sublist
                  quickSort( A, pos+1, high )       # quick sort higher sublist

      def partition(A, low, high):
            pivot = A[low]                              # set pivot value
            left, right = low, high -1

            while True:                                 # infinite loop
                  while A[left] < pivot:                # increment left pointer
                        left = left + 1
                  while A[right] > pivot:               # decrement right pointer
                        right = right - 1

                  if left < right:                      # swap pointers
                        swap( A, left, right )
                  else:                                 # left and right pointer
meets
                        return right
```

- Benefit: Efficient for any array
- Drawback: Unstable

Selection Sort (Not In Syllabus)

- Total number of passes: number of elements - 1
- For each pass, find smallest element to exchange with first element in selected group
- Ignore first element of selected group after every pass to get next selected group
- Process:
    - Assume n elements in array A, index starts from 0
    - Pass 0:
        - Select smallest element, exchange with A[0], placing it at beginning of array A
        - Front of array (A[0]) is ordered
        - Rear of array (A[1] - A[n-1]) is unordered
    - Pass 1:
        - Select smallest element from A[1] - A[n-1], exchange with A[1], placing it at front of sublist (A[1] - A[n-1])
        - Front of array (A[0] - A[1]) is ordered
        - Rear of array (A[2] - A[n-1]) is unordered
    - Pass 2:
        - Select smallest element from A[2] - A[n-1], exchange with A[2], placing it at front of sublist (A[2] - A[n-1])
        - Front of array (A[0] - A[2]) is ordered
        - Rear of array (A[3] - A[n-1]) is unordered
    - ...
- Code:

```
def selectionSort ( A, n ):        # array A[0...n-1] of n integers
    for i in range( 0, n ):
        startIndex = i                 # start of sub-array
        minIndex = i               # check for smallest value in sub-array
        for j in range( i+1, n ):
            if A[j] < A[minIndex]:
                minIndex = j
        swap(A, minIndex, startIndex)
```

- Benefit: Simple
- Drawback: Inefficient

Merge Sort (Not in Syllabus)

- **Merging:**
  - Combines 2 arrays that are already sorted
  - Outputs 3rd arrays that is sorted
  - Process:ku
    - Assume sorted arrays array A and array B
    - Initialise empty array C
    - Read first element x from A
    - Read first element y from B
      - If x < y:
        - Write x into C
        - Read new x from A
      - Else (y < x):
        - Write y into C
        - Read new y from B
    - If end of A is reached:
      - Copy all remaining elements from B into C
    - If end of B is reached:
      - Copy all remaining elements from A into C
  - Code:

```
def mergeSort (A, B):                  # Assume sorted arrays A, B
    res = [ ]                          # Resultant array
    i = 0
    j = 0

    while ( i < len(A) ) and ( j < len(B) ):     # loop when end of array A and B are not reached
        if A[i] < B[j]:
            res.append ( A[i] )
            i = i + 1
        else:
            res.append ( B[j] )
            j = j +1

        while i < len(A):                        # end of array B reached
            res.append ( A[i] )                  # append
```

remaining A into resultant
i = i + 1

while j < len(B):   # end of array A reached
res.append (B[j])   # append remaining B into resultant
j = j + 1

- **Straight Merge Sort:**
  - Process:
    - Start with array A = [6, 5, 3, 1, 8, 7, 2, 4]
    - Separate all elements into single-element subarrays A[0] - A[n]:
      - First pass: [6], [5], [3], [1], [8], [7], [2], [4]
    - Compare and sort adjacent subarrays, write into resultant array:
      - Second pass: [5, 6], [1, 3], [7, 8], [2, 4]
      - Third pass: [1, 3, 5, 6], [2, 4, 7, 8]
      - Fourth pass: [1, 2, 3, 4, 5, 6, 7, 8]
  - Efficiency increases as length of subarrays increases