

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 ($A[0]$ - $A[\text{lastExchangeIndex}]$) is unordered
 - ◆ Rear of array ($A[\text{lastExchangeIndex}]$ - $A[n-1]$) is ordered
 - Subsequent passes:
 - ◆ Compare adjacent terms in sublist of ($A[0]$ - $A[\text{lastExchangeIndex}]$)
 - Terminates when `lastExchangeIndex = 0`
- Code:

def bubbleSort(array):

n = len(array)
isSorted = False

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

for i in range(n - 1):
wrong order

```

    if array[i] > array[i + 1]:
        array = swap( array, i, i+1 )
        isSorted = False

```

```

    n -= 1
    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]
        j = i

```

```

        # locate insertion point
        while j > 0 and array[j] < array[j-1]:

```

```

            # free up space to insert

```

```

    array = swap( array, j, j-1 )
    j -= 1

```

```

    array[j] = target

```

```

    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):

```

```

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

```

```

    else:

```

```

        # select & remove pivot value (any index value)
        pivot_value = array.pop(0)

```

```

less = []
greater = []

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

    # array has more than one element
    if low < high:
        pos = partition( A, low, high ) # splits array into two
        sublists, pos is final position of pivot

        # quick sort lower sublist
        quickSort( A, low, pos-1 )

        # quick sort higher sublist
        quickSort( A, pos+1, high )

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

    # infinite loop
    while True:

        # increment left pointer
        while A[left] < pivot:

```

```

    left = left + 1

# decrement right pointer
while A[right] > pivot:
    right = right - 1

# swap pointers
if left < right:
    swap( A, left, right )

# left and right pointer meets
else:
    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
- Code:

```

def selectionSort ( A, n ):
    for i in range( 0, n ):
        startIndex = i
        minIndex = i
        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:

Assume sorted arrays A, B

def mergeSort (A, B):

res = []

i = 0

j = 0

loop when end of array A and B are not reached

while (i < len(A)) and (j < len(B)):

if A[i] < B[j]:

res.append (A[i])

i = i + 1

else:

res.append (B[j])

j = j + 1

end of array B reached

```

while i < len(A):
    # append remaining A into resultant
    res.append ( A[i] )
    i = i + 1

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