<2T>

# Description

We now turn our attention to using a divide and conquer strategy as a way to improve the performance of sorting algorithms. The first algorithm we will study is the **merge sort**. Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.

**Merge sort** (sometimes spelled mergesort) is an efficient sorting algorithm that uses a divide-and-conquer approach to order elements in an array. Sorting is a key tool for many problems in computer science. For example, inputting a list of names to a sorting algorithm can return them in alphabetical order, or a sorting algorithm can order a list of basketball players by how many points they each scored. Running time is an important thing to consider when selecting a sorting algorithm since efficiency is often thought of in terms of speed. Mergesort runs in a guaranteed time, which is significantly faster than the average and worst-case running times of several other sorting algorithms.

Sorting may seem like a simple concept, but efficient sorting is critical when dealing with large amounts of data. Sorting is the basis for more complex computer programs such as searching for files on a computer, finding the shortest route to a destination, and compressing data.

# Pseudocode

**function** merge\_sort(*node* head)

// return if empty list

**if** (head == nil)

**return** nil

**var** *node* array[32]; initially all nil

**var** *node* result

**var** *node* next

**var** *int* i

result = head

// merge nodes into array

**while** (result != nil)

next = result.next;

result.next = nil

**for**(i = 0; (i < 32) && (array[i] != nil); i += 1)

result = merge(array[i], result)

array[i] = nil

// do not go past end of array

**if** (i == 32)

i -= 1

array[i] = result

result = next

// merge array into single list

result = nil

**for** (i = 0; i < 32; i += 1)

result = merge(array[i], result)

**return** result

# Code

# Merges two subarrays of arr[].

# First subarray is arr[l..m]

# Second subarray is arr[m+1..r]

def merge(arr, l, m, r):

    n1 = m - l + 1

    n2 = r- m

    # create temp arrays

    L = [0] \* (n1)

    R = [0] \* (n2)

    # Copy data to temp arrays L[] and R[]

    for i in range(0 , n1):

        L[i] = arr[l + i]

    for j in range(0 , n2):

        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]

    i = 0     # Initial index of first subarray

    j = 0     # Initial index of second subarray

    k = l     # Initial index of merged subarray

    while i < n1 and j < n2 :

        if L[i] <= R[j]:

            arr[k] = L[i]

            i += 1

        else:

            arr[k] = R[j]

            j += 1

        k += 1

    # Copy the remaining elements of L[], if there

    # are any

    while i < n1:

        arr[k] = L[i]

        i += 1

        k += 1

    # Copy the remaining elements of R[], if there

    # are any

    while j < n2:

        arr[k] = R[j]

        j += 1

        k += 1

# l is for left index and r is right index of the

# sub-array of arr to be sorted

def mergeSort(arr,l,r):

    if l < r:

        # Same as (l+r)/2, but avoids overflow for

        # large l and h

        m = (l+(r-1))/2

        # Sort first and second halves

        mergeSort(arr, l, m)

        mergeSort(arr, m+1, r)

        merge(arr, l, m, r)