Sorting Algorithms

Sorting

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. A collection of records called a list where every record has one or more fields. The fields which contain a unique value for each record is termed as the key field.

Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion. Sorting is performed according to some key value of each record.

The records are either sorted either numerically or alphanumerically. The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

Quick Sort

Quick Sort first partitions the array into two parts by picking a pivot. The left part contains all elements less than the pivot, while the right part contains all elements greater than the pivot. After the array is partitioned, quicksort recurses into the left and right parts. When a part contains only a single element, recursion stops.

The partition operation makes a single pass over the active part of the array. As each element is visited, if it is less than the pivot it is swapped into the lesser part; if it is greater than the pivot the partition operation moves on to the next element.

Pseudo Code

```
function quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end function
```

Code

```
# python implementation of quicksort
def partition(start, end, array):

    # Initializing pivot's index to start
    pivot_index = start
    pivot = array[pivot_index]

    # This loop runs till start pointer crosses
    # end pointer, and when it does we swap the
    # pivot with element on end pointer
    while start < end:

        # Increment the start pointer till it finds an
        # element greater than  pivot
        while start < len(array) and array[start] <= pivot:
            start += 1

        # Decrement the end pointer till it finds an
        # element less than pivot
        while array[end] > pivot:
            end -= 1

        # If start and end have not crossed each other,
        # swap the numbers on start and end
        if(start < end):
            array[start], array[end] = array[end], array[start]

    # Swap pivot element with element on end pointer.
    # This puts pivot on its correct sorted place.
    array[end], array[pivot_index] = array[pivot_index], array[end]
```

```

    # Returning end pointer to divide the array into 2
    return end

# The main function that implements QuickSort
def quick_sort(start, end, array):

    if (start < end):

        # p is partitioning index, array[p]
        # is at right place
        p = partition(start, end, array)

        # Sort elements before partition
        # and after partition
        quick_sort(start, p - 1, array)
        quick_sort(p + 1, end, array)
```

There are many variations of quicksort. The one shown above is one of the simplest and slowest. This variation is useful for teaching, but in practice more elaborate implementations are used for better performance.

Merge Sort

As you've likely surmised from either the code, mergesort takes a very different approach to sorting than quick sort. Unlike quick sort, which operates in-place by performing swaps, merges ort requires an extra copy of the array. This extra space is used to merge sorted subarrays, combining the elements from pairs of subarrays while preserving order. Since merge sort performs copies instead of swaps.

Merge Sort works from the bottom-up. Initially, it merges subarrays of size one, since these are trivially sorted. Each adjacent subarray — at first, just a pair of elements — is merged into a sorted subarray of size two using the extra array. Then, each adjacent sorted subarray of size two is merged into a sorted subarray of size four. After each pass over the whole array, merge sort doubles the size of the sorted subarrays: eight, sixteen, and so on. Eventually, this doubling merges the entire array and the algorithm terminates.

Because mergesort performs repeated passes over the array rather than recursing like quick sort, and because each pass doubles the size of sorted subarrays regardless of input, it is easier to design a static display. We simply show the state of the array after each pass.

Pseudo Code

```
function mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while

   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

```

```
end function
```

Code

```python
# python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

         # finding the mid of the array
        mid = len(arr)//2

        # dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # sorting the first half
        mergeSort(L)

        # sorting the second half
        mergeSort(R)

        i = j = k = 0

        # copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

# code to print the list
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()
```

Bubble Sort

Bubble Sort Algorithm is used to arrange N elements in ascending order, and for that, you have to begin with 0th element and compare it with the first element. If the 0^{th} element is found greater than the 1^{st} element, then the swapping operation will be performed, that is the two values will get interchanged.

In this way, all the elements of the array get compared.

Pseudo Code

```
function bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:
         if list[j] > list[j+1] then
            swap( list[j], list[j+1] )
            swapped = true
         end if

      end for

    /*if no number was swapped that means array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

    end for

end function
    return list
```

Code

```
# python program for implementation of BubbleSort

def bubbleSort(array):

  # loop to access each array element
  for i in range(len(array)):

    # loop to compare array elements
    for j in range(0, len(array) - i - 1):
      # compare two adjacent elements
      # change > to < to sort in descending order

      if array[j] > array[j + 1]:
        # swapping elements if elements
        # are not in the intended order

        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp
```

Optimized Bubble Sort

In the above algorithm, all the comparisons are made even if the array is already sorted.

This increases the execution time.

To solve this, we can introduce an extra variable swapped. The value of swapped is set **true** if there occurs swapping of elements. Otherwise, it is set **false**.

After an iteration, if there is no swapping, the value of swapped will be **false**. This means elements are already sorted and there is no need to perform further iterations.

This will reduce the execution time and helps to optimize the bubble sort.

Pseudo Code

```
function bubbleSort(array)
    swapped <- false
        for i <- 1 to indexOfLastUnsortedElement-1
            if leftElement > rightElement
                swap leftElement and rightElement
                swapped <- true
    end bubbleSort
```

Code

```python
# optimized Bubble sort in python

def bubbleSort(array):

  # loop through each element of array
  for i in range(len(array)):

    # keep track of swapping
    swapped = False

    # loop to compare array elements
    for j in range(0, len(array) - i - 1):

      # compare two adjacent elements
      # change > to < to sort in descending order
      if array[j] > array[j + 1]:

        # swapping occurs if elements
        # are not in the intended order
        temp = array[j]
        array[j] = array[j+1]
        array[j+1] = temp

        swapped = True

    # no swapping means the array is already sorted
    # so no need for further comparison
    if not swapped:
      break
```

Insertion Sort

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game. We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left.

In the same way, other unsorted cards are taken and put in their right place.

Pseudo Code

```
function insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while

      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

   end function
```

Code

```
# insertion sort in python

def insertionSort(array):

    for step in range(1, len(array)):
        key = array[step]
        j = step - 1
        # compare key with each element on the left
        # of it until an element smaller than it is found
        # For descending order, change key<array[j] to key>array[j].
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1

        # place key at after the element just smaller than it.
        array[j + 1] = key
```

Selection Sort

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Pseudo Code

```
function selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end function
```

Code

```python
# selection sort in python

def selectionSort(array, size):

    for step in range(size):
        min_idx = step

        for i in range(step + 1, size):

            # to sort in descending order, change > to < in this line
            # select the minimum element in each loop
            if array[i] < array[min_idx]:
                min_idx = i

        # put min at the correct position
        (array[step], array[min_idx]) = (array[min_idx], array[step])
```

Shell Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as interval.

This interval is calculated based on Knuth's formula as,

Knuth's Formula

```
h = (h * 3) + 1
where : h is interval with initial value 1
```

Pseudo Code

```
function shellSort()
   A : array of items

   /* calculate interval*/
   while interval < A.length /3 do:
      interval = interval * 3 + 1
   end while

   while interval > 0 do:

      for outer = interval; outer < A.length; outer ++ do:

      /* select value to be inserted */
      valueToInsert = A[outer]
      inner = outer;

         /*shift element towards right*/
         while inner > interval -1 && A[inner - interval] >= valueToInsert do:
            A[inner] = A[inner - interval]
            inner = inner - interval

         end while

      /* insert the number at hole position */
      A[inner] = valueToInsert

      end for

   /* calculate interval*/
   interval = (interval -1) /3;

   end while

end function
```

Code

```python
# python program for implementation of shell Sort

def shellSort(arr):

    # Start with a big gap, then reduce the gap
    n = len(arr)
    gap = n/2

    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped
    # order keep adding one more element until the entire array
    # is gap sorted
    while gap > 0:

        for i in range(gap,n):

            # add a[i] to the elements that have been gap sorted
            # save a[i] in temp and make a hole at position i
            temp = arr[i]

            # shift earlier gap-sorted elements up until the correct
            # location for a[i] is found
            j = i
            while  j >= gap and arr[j-gap] >temp:
                arr[j] = arr[j-gap]
                j -= gap

            # put temp (the original a[i]) in its correct location
            arr[j] = temp
        gap /= 2
```