Sorting Algorithms

Bubble Sort Algorithm

The Bubble Sort algorithm is a simple algorithm to sort a list of N numbers in ascending order. Bubble sort works by iterating through a list and checking whether the current element is larger or smaller than the next element.

This algorithm consists of an outer iteration and an inner iteration. In the inner iteration, the first and second elements are first compared and swapped so that the second element has a higher value than the first. This is repeated for the subsequent second and third element pairs and so forth until the last pair of (N-2, N-1) elements is compared. At the end of the inner iteration, the largest element appears last. This is repeated for all elements of the list in the outer iteration.

Bubble Sort Big-O Runtime

The Bubble Sort algorithm utilizes two loops: an outer loop to iterate over each element in the input list, and an inner loop to iterate, compare and exchange a pair of values in the list. The inner loop takes (N-1) iterations while the outer loop takes N iterations. Hence, the Big-O runtime for the algorithm is the product of O(N) and O(N-1), which is O(N^2).

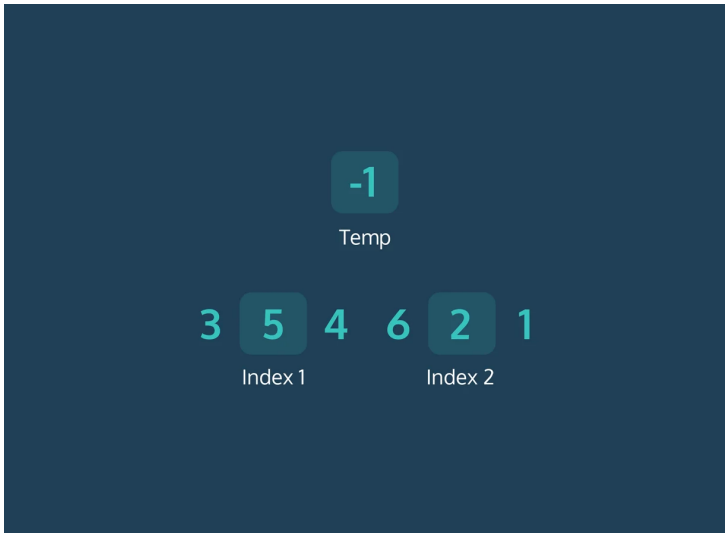
Bubble Sort Swapping Variables

The Bubble Sort algorithm requires swapping of variables in order to sort them. The swapping algorithm is dependent on the programming language. For most languages, a temporary variable is needed to hold one of the values being swapped:

```
temp_variable = number_1
number_1 = number_2
number_2 = temp_variable
```

For others, the swapping can be done in a single assignment:

```
number_1, number_2 = number_2, number_1
```



Swapping Variables in Bubble Sort

In the Bubble Sort algorithm, the swap function that swaps two elements in a `list` can be called in a Bubble Sort function to iteratively swap an element with its adjacent neighbor whose value is smaller until all the elements are sorted in ascending order.

```
def swap(arr, left_pos, right_pos):
  temp = arr[left_pos]
  arr[left_pos] = arr[right_pos]
  arr[right_pos] = temp

def bubble_sort(arr):
  for itm in arr:
    for idx in range(len(arr) - 1):
      if arr[idx] > arr[idx + 1]:
        swap(arr, idx, idx + 1)
```

Python Swap Function

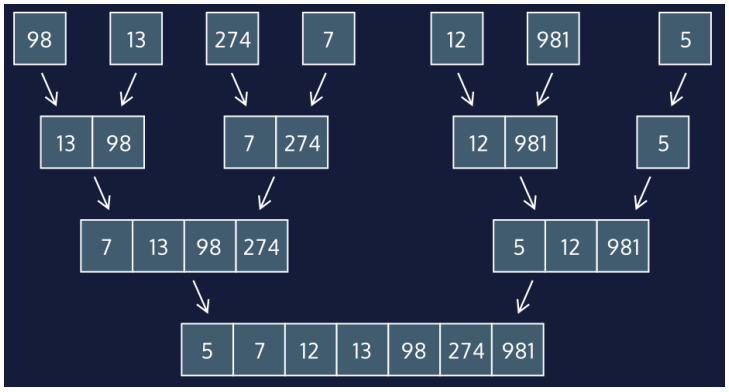
A Python function that swaps two adjacent values in a list can be written as follows:

```
def swap(arr, pos_1, pos_2):
  tmp = arr[pos_1]
  arr[pos_1] = arr[pos_2]
  arr[pos_2] = tmp
```

Merge Sort Merging

Merge Sort is a divide and conquer algorithm. It consists of two parts:

```
1) splitting the original list into smaller sor
2) merging back the presorted 1-element lists i
```



The merging portion is iterative and takes 2 sublists. The first element of the left sublist is compared to the first element of the right sublist. If it is smaller, it is added to a new sorted list, and removed from the left sublist. If it is bigger, the first element of the right sublist is added instead to the sorted list and then removed from the right sublist. This is repeated until either the left or right sublist is empty. The remaining non-empty sublist is appended to the sorted list.

Big-O Runtime for Merge Sort

The Merge Sort algorithm is divided into two parts. The first part repeatedly splits the input list into smaller lists to eventually produce single-element lists. The best, worst and average runtime for this part is Θ(log N). The second part repeatedly merges and sorts the single-element lists to twice its size until the original input size is achieved. The best, worst and average runtime for this part is Θ(N). Therefore, the combined runtime is Θ(N log N).

Merge Sort Implementation in Python

We can implement the Merge Sort algorithm in Python using two functions, `merge_sort(lst)`, the main function and `merge(left, right)`, a helper function.

```
def merge_sort(lst):
  if len(lst) <= 1:
    return lst
  middle = len(lst) // 2
  left = lst[:middle]
  right = lst[middle:]
  sleft = merge_sort(left)
  sright = merge_sort(right)
  return merge(sleft, sright)

def merge(left, right):
  result = []
  while (left and right):
    if left[0] < right[0]:
      result.append(left[0])
      left.pop(0)
    else:
      result.append(right[0])
      right.pop(0)
  if left:
    result += left
  if right:
    result += right
  return result
```

Quick Sort Performance

Quicksort's performance can be inefficient when the algorithm encounters imbalanced partitions. The worst case scenario is if the first or last element is always the partition point for an array or sub-array. In this case, one side of the partition will contain all the elements. This makes the recursive stack deeper, resulting in `O(N^2)` runtime.

Quick Sort General

Quicksort is a method for sorting an array by repeatedly partitioning it into sub-arrays by:

1. Selecting an element from the current array. This element is called the pivot element, and in our implementation we used the mid element.
2. Comparing every element in the array to the pivot element, swap the elements into sides greater than and less than. The partition point in the array is where we guarantee everything before is less and everything after is greater than.
3. Repeating this process on the sub-arrays separated by the partition point. Do this until a sub-array contains a single element. When the partitioning and swapping are done, the arrays are sorted from smallest to largest.

The worst case runtime for quicksort is $O(N^2)$ and the average runtime for quicksort is $O(N \log N)$. The worst case runtime is so unusual that the quicksort algorithm is typically referred to as $O(N \log N)$ "

Print Share