1 Bubble Sort

Recall the pseudocode for bubble sort.

```
function bubbleSort(L)
  sorted = false
  while not sorted
    sorted = true
    for i from 1 to length(L)-1
      if L[i-1] > L[i]
        temp = L[i-1]
        L[i-1] = L[i]
        L[i] = temp
        sorted = false
  return L
```

Intuitively, bubble sort works its way from the beginning of the list to the end swapping adjacent elements when they are out of order. Repeat this process until one full pass is made over the list without making any swaps.

Let's walk through how this algorithm sorts the list $[2, 5, 4, 3, 1]$. Instead of following the pseudocode line by line, we can use the intuition to see how the list changes over time.

$[2, 5, 4, 3, 1]$ (Our initial list)

$[2, 4, 5, 3, 1]$ ($5 > 4$ so we swap these values)

$[2, 4, 3, 5, 1]$ ($5 > 3$ so we swap)

$[2, 4, 3, 1, 5]$ ($5 > 1$ so we swap. Notice how the largest value in the list has "bubbled" up to the end)

$[2, 3, 4, 1, 5]$ (Starting back at the beginning of the list, we see $4 > 3$ and swap)

$[2, 3, 1, 4, 5]$ (And $4 > 1$ so we swap. Do we need to compare 4 and 5 here?)

$[2, 1, 3, 4, 5]$ (Compare 3 and 1 and swap. Do we need to compare 3 and 4?)

$[1, 2, 3, 4, 5]$ (Swap 1 and 2)

$[1, 2, 3, 4, 5]$ (Check that everything is in order)

Very importantly, bubble sort requires one more additional pass through the whole list to verify that it is sorted.

2 Insertion Sort

Let's look at insertion sort next.

```
function insertionSort(L)
  i = 1
  while i < length(L)
    j = i
    while j > 0 and L[j] < L[j-1]
      temp = L[j-1]
      L[j-1] = L[j]
      L[j] = temp
      j = j - 1
    i = i + 1
  return L
```

Insertion sort keeps track of a sorted and unsorted portion of the list. At each step, the first element of the unsorted portion is moved to the sorted portion and placed in the correct location. This is repeated until the unsorted portion of the list is empty.

Let's walk through how insertion sort works on $[2, 5, 4, 3, 1]$, the same list as above.

$[2, 5, 4, 3, 1]$ (The first element is sorted on its own)

$[2, 5, 4, 3, 1]$ (Insert 5. Since $5 > 2$ it's already in the right spot)

$[2, 4, 5, 3, 1]$ (Insert 4. $4 < 5$ so we swap these values. Now the first 3 elements are sorted)

$[2, 4, 3, 5, 1]$ (Insert 3. First, $3 < 5$ so we swap these values)

$[2, 3, 4, 5, 1]$ (Next, swap 3 and 4 since $3 < 4$. $3 > 2$ so we stop swapping here)

$[2, 3, 4, 1, 5]$ (Finally, inserting 1, we start by swapping 1 and 5 since $1 < 5$)

$[2, 3, 1, 4, 5]$ (Swap again since $1 < 4$)

$[2, 1, 3, 4, 5]$ (And again since $1 < 3$)

$[1, 2, 3, 4, 5]$ (And one more time since $1 < 2$)

3 Selection Sort

Selection sort is next.

```
function selectionSort(L)
  for i from 0 to length(L)-2
    uMin = i
    for j from i+1 to length(L)-1
      if L[j] < L[uMin]
        uMin = j
    temp = L[i]
    L[i] = L[uMin]
    L[uMin] = temp
  return L
```

Similar to insertion sort, we can think of selection sort as keeping track of sorted and unsorted sections of the list. Selection sort looks through all elements of the unsorted portion, finds the minimum, and moves it to the end of the sorted portion.

$[2, 5, 4, 3, 1]$ (Our original ordering)

$[1, 5, 4, 3, 2]$ (Looking through the whole list, 1 is the minimum. We swap this with 2)

$[1, 2, 4, 3, 5]$ (Looking through $[5, 4, 3, 2]$, we find 2 to be the minimum and swap it with 5)

$[1, 2, 3, 4, 5]$ (The minimum of $[4, 3, 5]$ is 3. We swap this with 4 so it is at the end of the sorted portion)

$[1, 2, 3, 4, 5]$ (Selection sort doesn't notice that the list is sorted yet. It finds the minimum of $[4, 5]$ and swaps with 4)

Notice that we do not need to look for the minimum of [5] at the very end. This is why the outer loop can stop at `length(L)-2` instead of going to `length(L)-1`.

It is easy to confuse insertion and selection sort. Remember that insertion sort **inserts** one element at a time into a sorted portion of the list while selection sort looks at all remaining elements and **selects** the one that should come next.

4 Quicksort

The sorts above are all iterative. The next sort uses recursion.

```
function quicksort(L)
  if length(L) <= 1
    return L
  pivot = L[0]
  A = []
  B = []
  for e in rest(L)
    if e <= pivot
      append(A, e)
    else
      append(B, e)
  return quicksort(A) + [pivot] + quicksort(B)
```

Quicksort uses the following intuition. If the given list is of length 0 or 1, it is already sorted (the base case). Otherwise, select of pivot value from the list L (we choose the first element here). Place all values less than or equal to the pivot (excluding the pivot itself) in one list and all other values in another. Recursively sort these two lists. Concatenate these now sorted lists together with the pivot in the middle.

Since quicksort is recursive, it is natural to visualize how it sorts a list as a kind of tree as it breaks the problem down (see Figure 1).

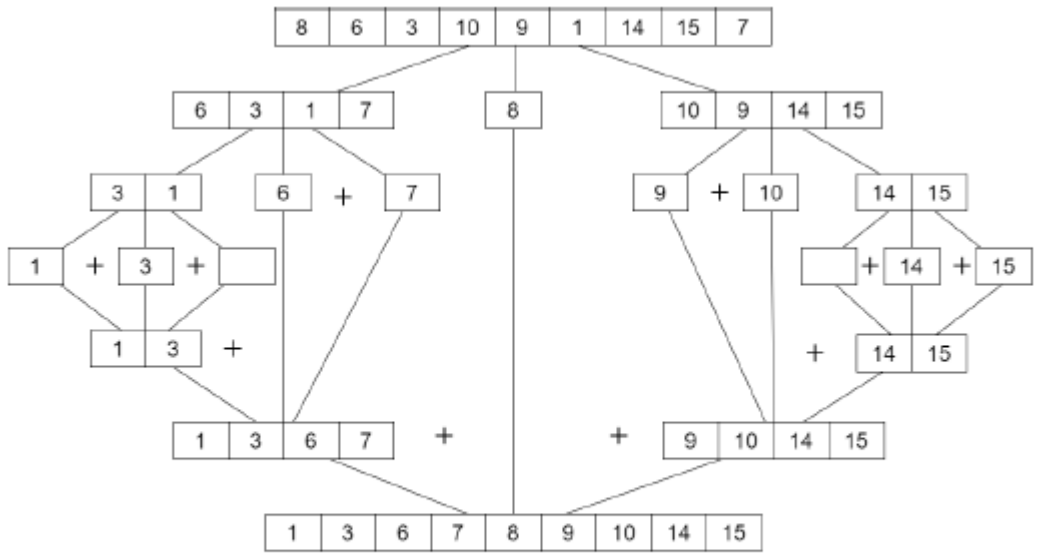

Figure 1: Quicksort applied to the list $[8, 6, 3, 10, 9, 1, 14, 15, 7]$. The + operators here represent concatenation after sublists have been sorted recursively.

5 Merge Sort

Another recursive sort!

```
function mergeSort(L)
  if length(L) <= 1
    return L
  A = mergeSort(L[:length(L)/2])
  B = mergeSort(L[length(L)/2:])
  return merge(A, B)

function merge(A, B)
  C = []
  while length(A)>0 and length(B)>0
    if A[0] < B[0]
      append(C, A[0])
      A = rest(A)
    else
      append(C, B[0])
      B = rest(B)
  if length(A)>0
    C = C + A
  else if length(B)>0
    C = C + B
  return C
```

Similar to quicksort, merge sort works by recursively sorting smaller and smaller lists. The key difference is that merge sort does not use a pivot. Instead, a little more work is done to combine lists back together (this is not a simple concatenation as in quicksort).

Intuitively, merge sort says that a list of length 0 or 1 is already sorted (the base case). If the list has more than a single elements, split it in half. Sort these two halves recursively and then merge them back together. Can you figure out how the merge process works?

Similar to quicksort, it is natural to visualize how merge sort divides the problem up as a tree (see Figure 2).

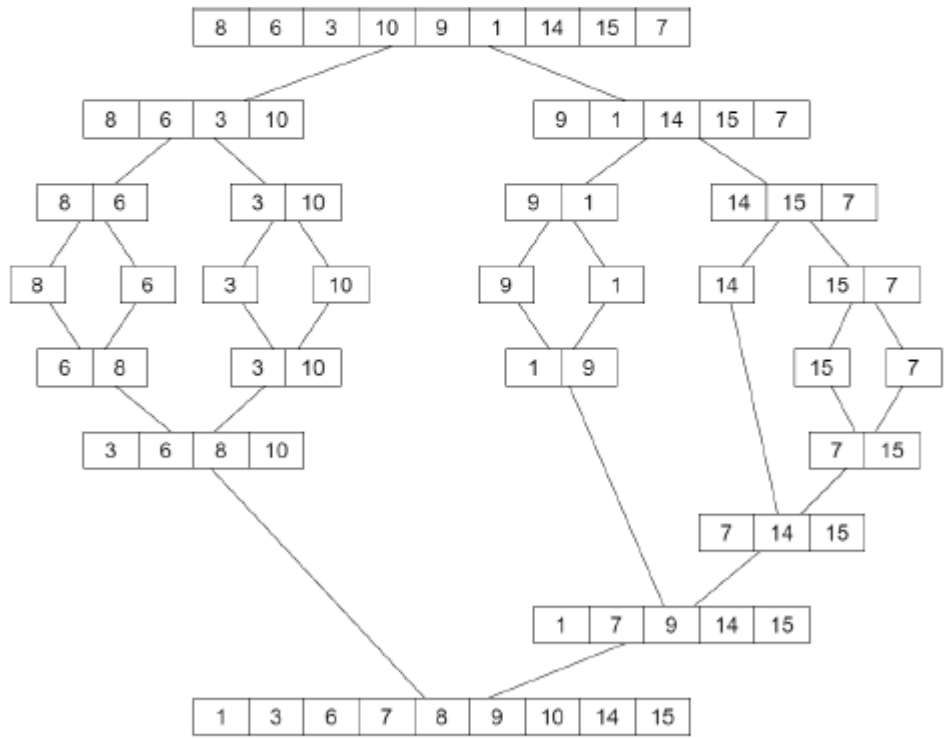

Figure 2: Merge sort applied to the list $[8, 6, 3, 10, 9, 1, 14, 15, 7]$.