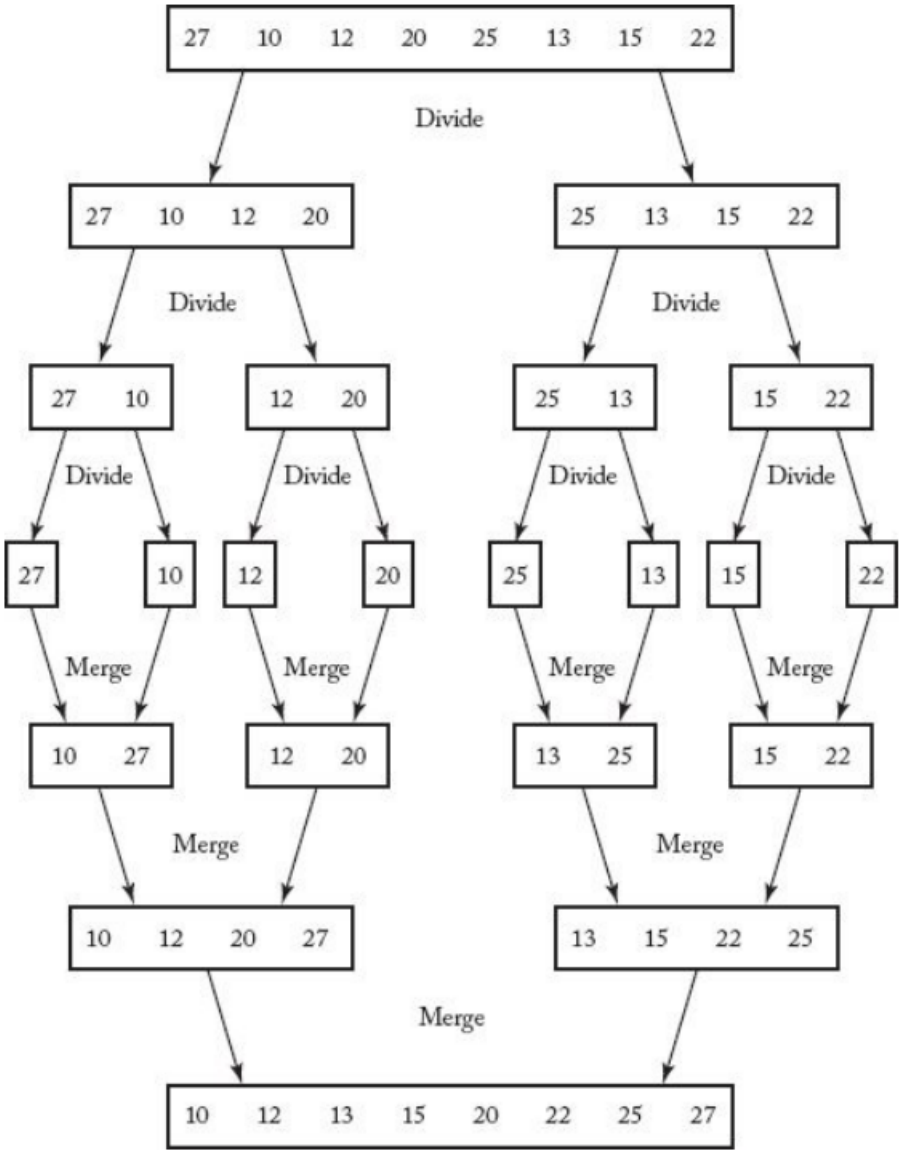
Algorithms

---

1 Merge Sort

The Merge Sort algorithm is a divide and conquer algorithm whose three basic parts are the following: 1) Divide - split lists into smaller list of size $\frac{n}{2}$, 2) Conquer - Sort the smaller lists, 3) Combine - Merge the sorted lists. The following diagram is a visual representation of how this algorithm sorts lists taken from the textbook.



1.1 The Merge Process

The process of merging two smaller lists into a larger one has 3 major steps. First, compare the first items of each of the smaller lists to each other. Second, the smaller of the two is appended onto the larger list. Third, repeat steps 1 and 2, but for subsequent repititions use the next item in the list of smaller item that was compared previously.

For example, the last merge in the diagram above between the lists $[10, 12, 20, 27]$ and $[13, 15, 22, 25]$ starts by comparing 10 and 13. $10 < 13$ so 10 goes to the final list and the next comparison is between 12 and 13. It continues to compare 20 and 13, 20 and 15, 20 and 22 etc. until the final list is completed sorted in nondecreasing order $[10, 12, 13, 15, 20, 22, 25, 27]$.

1.2 mergeSort Pseudocode

```
function MERGESORT(A)                          ▷ A is the list to be sorted
    if n ≤ 1 then                              ▷ n is the length of A
        return
    end if
    x = n//2                                   ▷ Integer division
    y = n − x
    B = new array of length x
    C = new array of length y
    B[0 ... x − 1] = A[0 ... x − 1]
    C[0 ... y − 1] = A[x ... n − 1]
    MERGESORT(B)          ▷ Recursively sort smaller list B, same for C below
    MERGESORT(C)
    MERGE(B, C, A)                             ▷ Combines all smaller lists
end function
```

1.3 merge Pseudocode

```
function MERGE(B, C, A)                        ▷ combines B and C into A
                          ▷ Assumes length(B) + length(C) == length(A)
    i = j = k = 0
    while i < length(B) && j < length(C) do
        if B[i] ≤ C[j] then                    ▷ item in B smaller than item in C
            A[k] = B[i]                        ▷ item in B put into A
            i++                                ▷ i set to next item in B
        else                                   ▷ item in C smaller than item in B
            A[k] = C[j]                        ▷ item in C put into A
            j++                                ▷ j set to next item in C
        end if
        k++                                    ▷ k set to next index in A
    end while
                ▷ One list has ended, determine which and copy the rest of the other
    if i == length(B) then                     ▷ B ended first, copy remaining C to A
        A[k ... length(A) − 1] = C[j ... length(C) − 1]
    else                                       ▷ C ended first, copy remaining B to A
```

$A[k \dots length(A) - 1] = B[i \dots length(B) - 1]$
end if
end function

2 Merge Sort Analysis

For the worst-case analysis for merge and mergeSort, $x : length(B) = \frac{n}{2}$ and $y : length(C) = n - x$. These values for x and y were established in the mergeSort pseudo code above.

2.1 merge worst-case analysis

Basic Operation: comparison of $B[i]$ & $C[j]$
Input size: x & y which represent the sizes of the 2 lists being merged

The worst case for merge occurs when one list finishes being put into A and the other list only has 1 item left. $[\dots\dots i]$ and $[\dots\dots]$ j represents this situation as j has moved through the entirety of its list and i only has one more item to be moved to A to be finished in its list. Thus, the worst-case time complexity is

$$W(x, y) = x + y - 1$$

where the -1 correlates to the 1 item left to get copied over to A.

2.2 mergeSort worst-case analysis

Basic operation: the number of comparisons done in merge
Input size: n, the length of the list to be sorted

For this analysis it is assumed that n is a power of 2 which means $x = \frac{n}{2}$ and $y = \frac{n}{2}$.

The total number of comparisons done in mergeSort is the number of comparisons done in $mergeSort(B) + mergeSort(C) + merge(B\&C)$. Thus, $W(n)$ is

$$\begin{aligned} W(n) &= W(x) + W(y) + (x + y - 1) \\ &= W\left(\frac{n}{2}\right) + W\left(\frac{n}{2}\right) + n - 1 \\ &= 2W\left(\frac{n}{2}\right) + n - 1 \end{aligned}$$

In the first equation, $W(x)$ is the number of comparisons done in $mergeSort(B)$ as x and B both represent the lower half of items in the list, $W(y)$ relates to $mergeSort(C)$ for the same reason and $(x + y - 1)$ correlates to $merge(B\&C)$ as that was the worst-case for merge calculated above. In the second line, x and y are substituted for $\frac{n}{2}$ because of the assumption stated at the beginning of the analysis and $n = x + y$ for the same reason.

When $n = 1$, $W(1) = 0$ as there are no comparisons to be done with a list of length 1 which gives an initial term for the recurrence relation:

$$W(n) = \begin{cases} 0 & \text{for } n = 1 \\ 2W(\frac{n}{2}) + n - 1 & \text{for } n > 1 \end{cases}$$

The Master Method will be used to solve this recurrence where $a = 2$, $b = 2$, $f(n) = n - 1$ and $d = 1$ because $f(n) \in O(n)$. Using these values, $b^d = 2^1 = 2$ which means $a = b^d \equiv 2 = 2^1$ and the second case of the Master Method applies. Therefore,

$$W(n) \in \Theta(n^d log(n)) \equiv\in \Theta(n \lg(n))$$

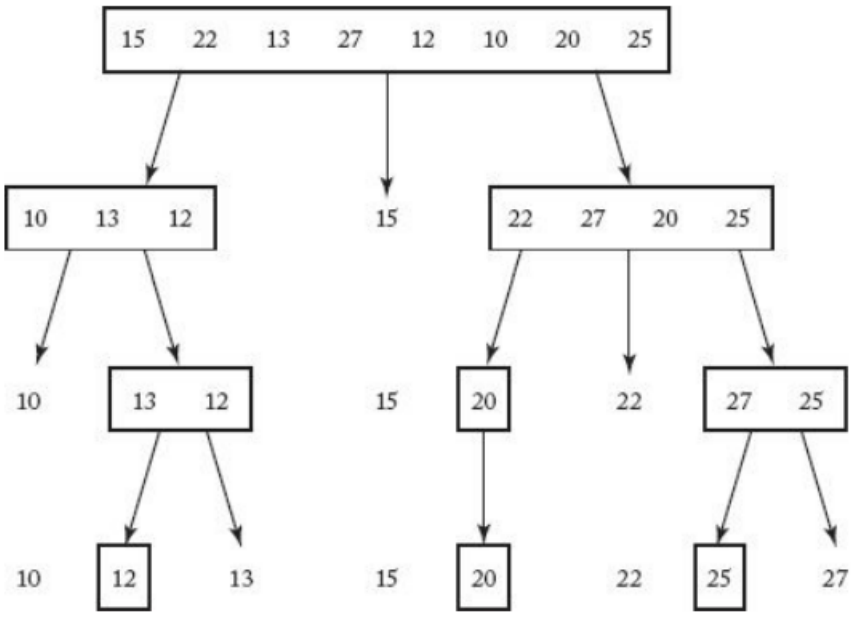
3 Quicksort

The main components of the QuickSort algorithm are: 1) Divide - list into two parts called partitions which are not guaranteed to be size of $\frac{n}{2}$ like MergeSort and 2) Conquer - sort each part/partition recursively. For this algorithm the third step of recombining is not necessary.

3.1 Partition selection

In order to split a list using the Quicksort algorithm there are two steps. First, pick a pivot item from the list and second, move items $<$ the pivot to its left and items $>$ the pivot to its right.

For example, given the list $[15, 22, 13, 27, 12, 10, 20, 25]$ if 15 is selected as the pivot item, the partitions of the list would be $[10, 13, 12]$ and $[22, 27, 20, 25]$. The pivot item, 15, is not included in the partitions because it is already sorted in the right place (lower values to the left and larger to the right). This is why there is no recombining component necessary for this algorithm to sort a list and results in the length of the two partitions being 1 less than the length of the original list. Below is a diagram of how the algorithm would recursively sort the rest of the list taken from the textbook.



3.2 quickSort pseudocode

```
function QUICKSORT(A, low, high)
    if low < high then
        pivotPoint = PARTITION(A, low, high)
        QUICKSORT(A, low, pivotPoint − 1)          ▷ Recursively sorting
        QUICKSORT(A, pivotpoint + 1, high)
    end if
end function
```

3.3 partition pseudocode

```
function PARTITION(A, low, high)                   ▷ return an int
    pivotItem = A[low                              ▷ First item/element selected pivot
    j = low
    for i from low + 1 to high do
        if A[i] < pivotItem then
            j + +
            swap A[i] and A[j]
        end if
    end for
    pivotPoint = j
    swap A[low] and A[pivotPoint]
    return pivotPoint
end function
```