Algorithms & Data Structures I Week 17 Lecture Note

Notebook:	Algorithms & Data Structures I		
Created:	2020-10-21 4:14 PM	**Updated:**	2021-01-22 4:35 PM
Author:	SUKHJIT MANN		

Cornell Notes	**Topic:** Sorting Data III, Part 1	Course: BSc Computer Science Class: CM1035 Algorithms & Data Structures I [Lecture] Date: January 20, 2021

Essential Question:

What are divide and conquer algorithms?

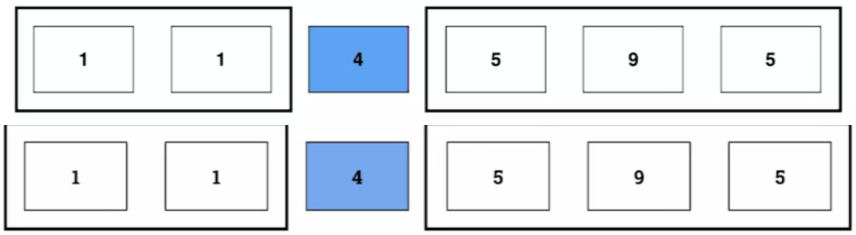
Questions/Cues:

- What is a divide and conquer algorithm?
- What is Quicksort?
- How can we implement Quicksort without creating new vectors and instead using the Swap operations?
- What is the proof that Quicksort is indeed correct?
- What is the pseudocode for QuickSort?
- What is Mergesort?
- What is the pseudocode for merge procedure?
- What is the pseudocode for Mergesort?

Notes

- Divide and conquer algorithm = in D&C algorithm, we take a problem, divide it up into two or more smaller problems, solve the smaller problems and then combine the solutions to give a solution to the original problem; this is simplistically achieved through recursion. This a general approach to algorithm design.
- Quicksort = In Quicksort, we reduce our input vector to one or two smaller vectors and then call Quicksort individually on these smaller vectors using recursion
    - We generate the smaller vectors by partitioning elements according to how their values compare with the value at a particular element called the pivot. The pivot is just a particular element of the vector, how it is chosen can very much depend on a particular version of Quicksort.
        - The version of Quicksort taught in this week chooses the midpoint of the vector to be the pivot. In versions of Quicksort, the rightmost element is often chosen as the pivot
    - Once the pivot has been chosen and its value inspected, we partition the vector into two smaller vectors where the elements with values smaller than the value of the pivot element are on the left and elements with larger than that of the pivot go to the right. Once we have finished partitioning the vector into smaller vectors, we run Quicksort on the smaller vectors
    - The base case is a vector of one or zero elements where the input is already sorted and just needs to be returned

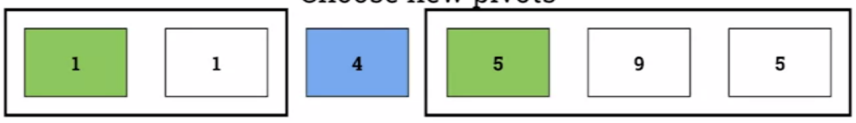
left = 1, right = 6
Pick a pivot = floor((left + right)/2)



Left vector Right vector

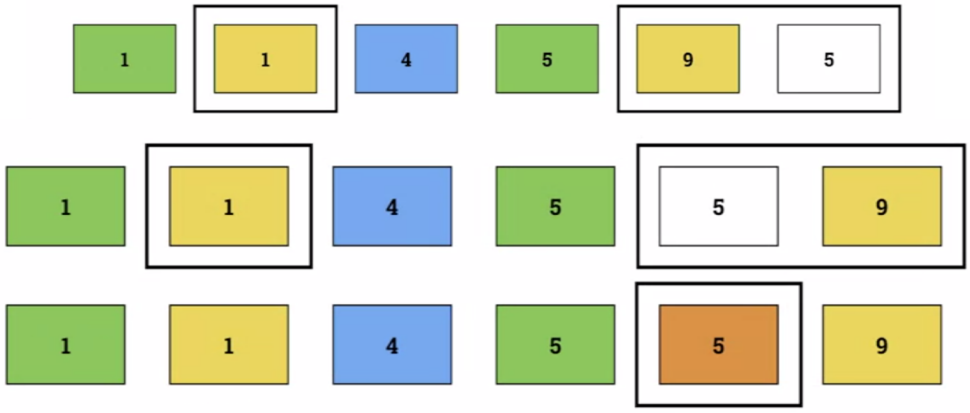
Apply Quicksort to these individual vectors

Choose new pivots



Generate new smaller vectors, apply Quicksort

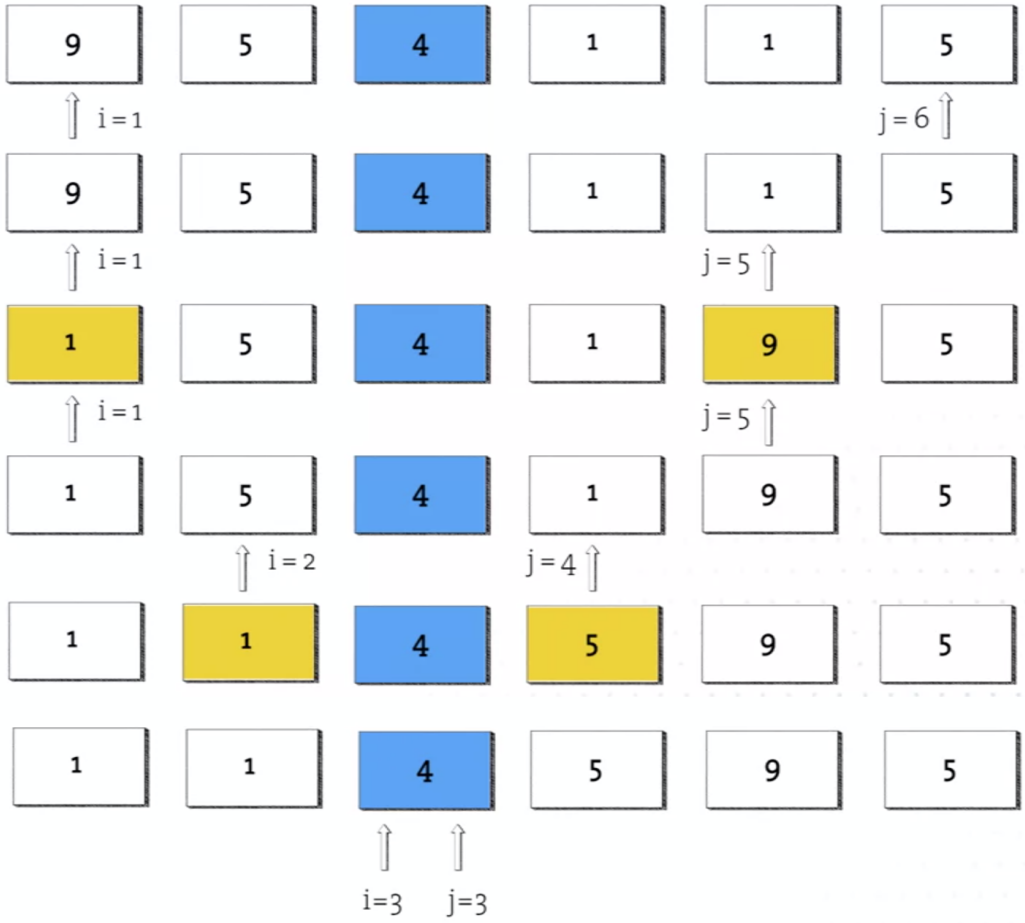
Choose pivots...



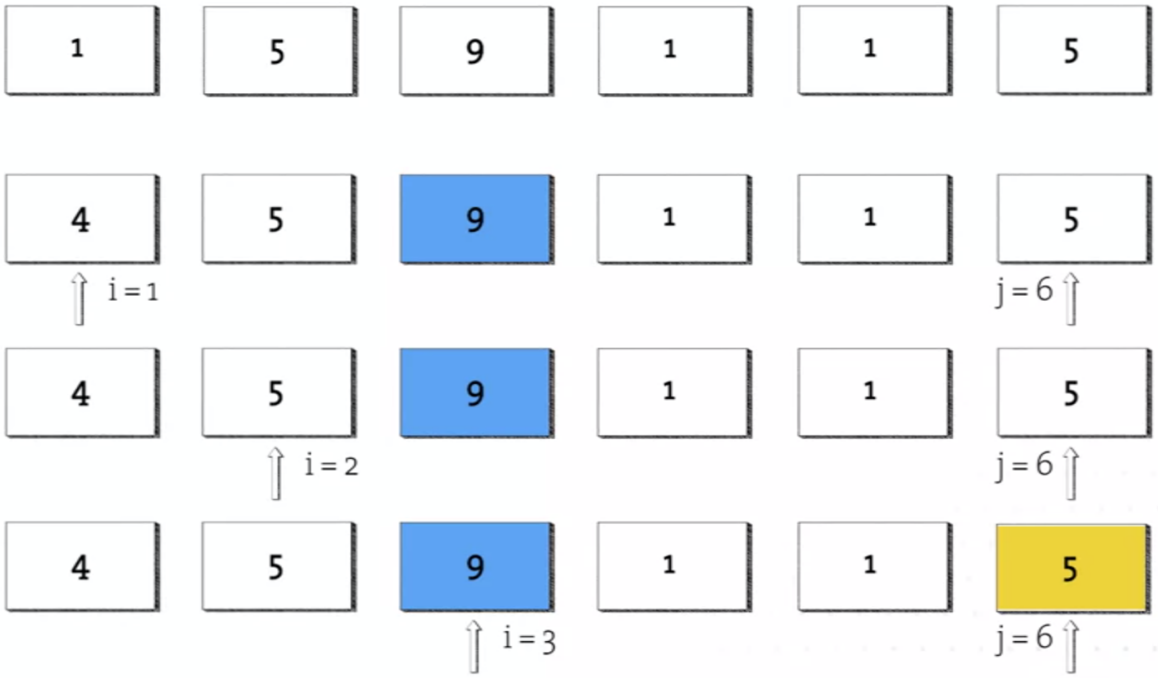
When we hit base case of vector of 1 element

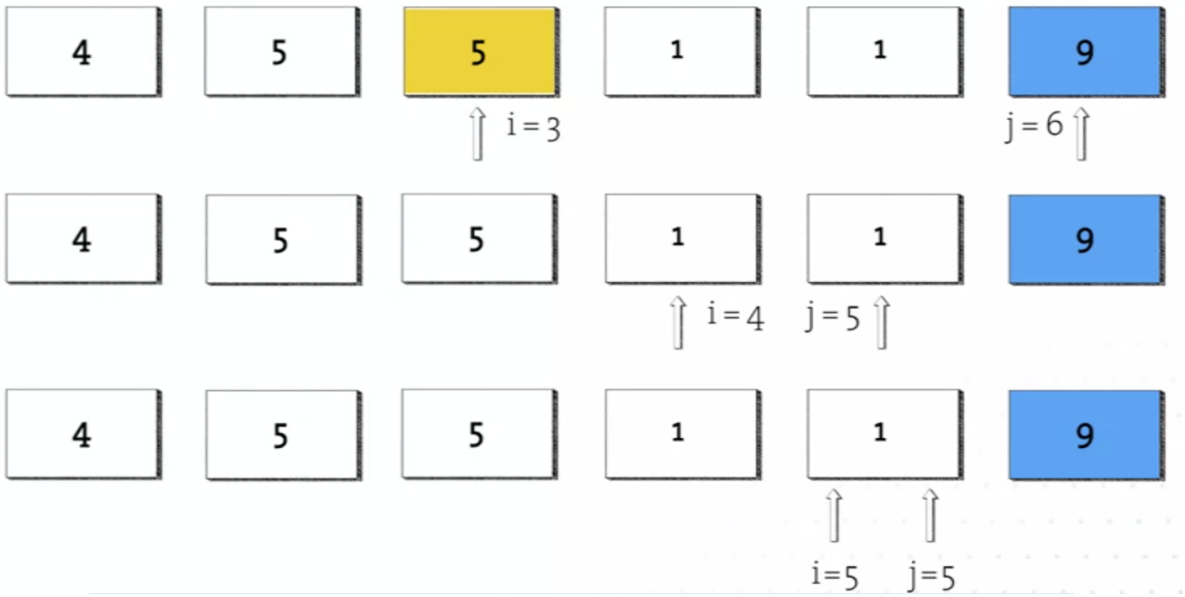
Return vector

- To perform Quicksort with Swap operation, we first create two new variables i and j.
  - i starts at the leftmost element and j starts at the rightmost element. From here, we going to compare the values indexed by i and j in order to partition the vector into two smaller vectors. So we start with i and i is going to checking whether the value stored at i is smaller than the value at the pivot. For j, we are going be looking at values to the right of the pivot and if they are smaller than the value of the pivot, we going to move it to the left-hand side and if it's greater or equal to the pivot then it just stays where it is.



Finish partitioning when i=j





Quicksort

Partitioning divides vectors into smaller vectors - this works correctly

Proof that the recursive component of algorithm works by **induction**

First prove base case is correct

Then prove that **if** *n*th case is correct, **then** *(n+1)*th case is also correct

Recursion reduces to base case: we have “correctness all the way down”

Quicksort

Base case: vectors of length 0 or 1

Quicksort does nothing, which is correct
Inductive step: assume Quicksort sorts correctly all vectors of length n or less

If we have a vector of length $n+1$

Partitioned into at most three vectors: pivot, left vector, right vector

Largest of three vectors will have at most n elements — Quicksort can correctly sort vectors of length at most n

Whole vectors will be sorted

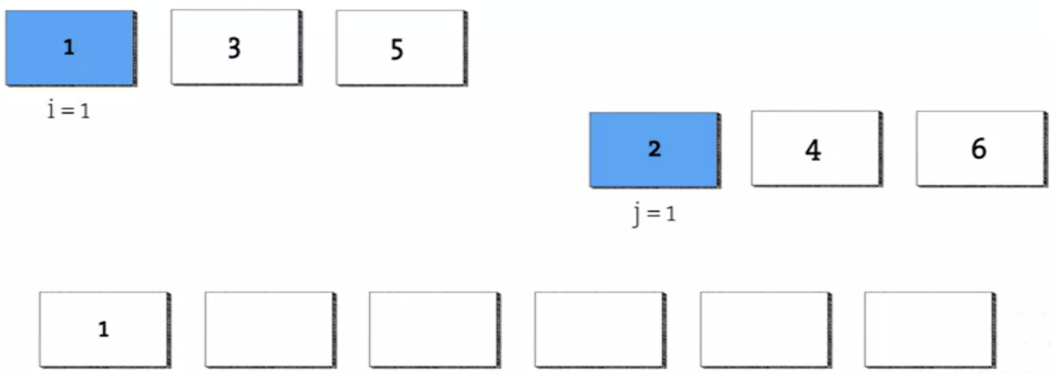
The partition we present in this module is called the Hoare partition, named after the inventor of Quicksort, Tony Hoare. There are other methods of partitioning; one such method described in module textbook Introduction to Algorithms (Third Edition) by Cormen *et al* (in section 7.1) is called the Lomuto partition, attributed to Nico Lomuto. In your studies you can use whichever partition function you wish as long as it works.

We will assume that the SWAP(*vector*, i, j) function is already defined, which swaps the values in elements i and j in a vector called *vector*. Let's define the partition function based on the Hoare partition:

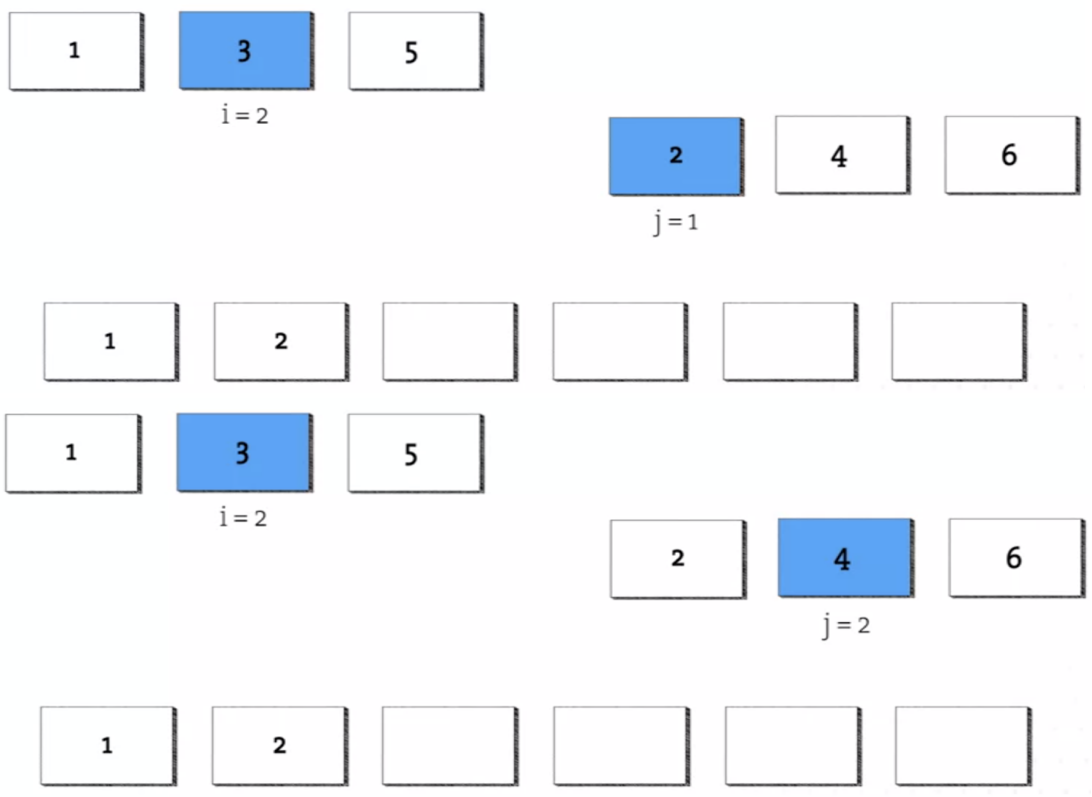
```
1:  function PARTITION(vector, i, j)
2:      m ← ⌊LENGTH(vector/2)⌋                          ▷ this is the mid-point in the vector
3:      pivot ← vector[m]                               ▷ this is the pivot value stored at the mid-point
4:      final ← m                                       ▷ this will be the final location in the vector of the pivot value
5:      while i < j do
6:          while vector[i] < pivot do
7:              i ← i + 1                               ▷ this will increase the index on the left until a value should be swapped
8:          end while
9:          while vector[j] > pivot do
10:             right ← j − 1                           ▷ this will decrease the index on the right until a value should be swapped
11:         end while
12:         if i < j then
13:             SWAP(vector, i, j)                      ▷ two values swapped at i and j
14:             if i = final then
15:                 final ← j                           ▷ updates the location of the pivot value in the vector if it is being swapped
16:                 i ← i + 1
17:             else if j = final then
18:                 final ← i                           ▷ updates the location of the pivot value in the vector if it is being swapped
19:                 j ← j − 1
20:             else
21:                 i← i + 1
22:                 j ← j − 1
23:             end if
24:         end if
25:     end while
26:     return final                                    ▷ final location of the pivot
27: end function
```

We have some comments on the right to help us follow the logic of the function. The important thing to observe is that the variable final keeps track of the location of the pivot value because it might be swapped with other values. So if we call the index f the value returned by the function PARTITION(*vector*, 1, LENGTH[*vector*]) there are now at most two smaller sub-vectors from element 1 to f − 1, and another from element f + 1 to LENGTH[*vector*]. We can then recursively apply the Quicksort algorithm to each of these sub-vectors, thereby partitioning them further.

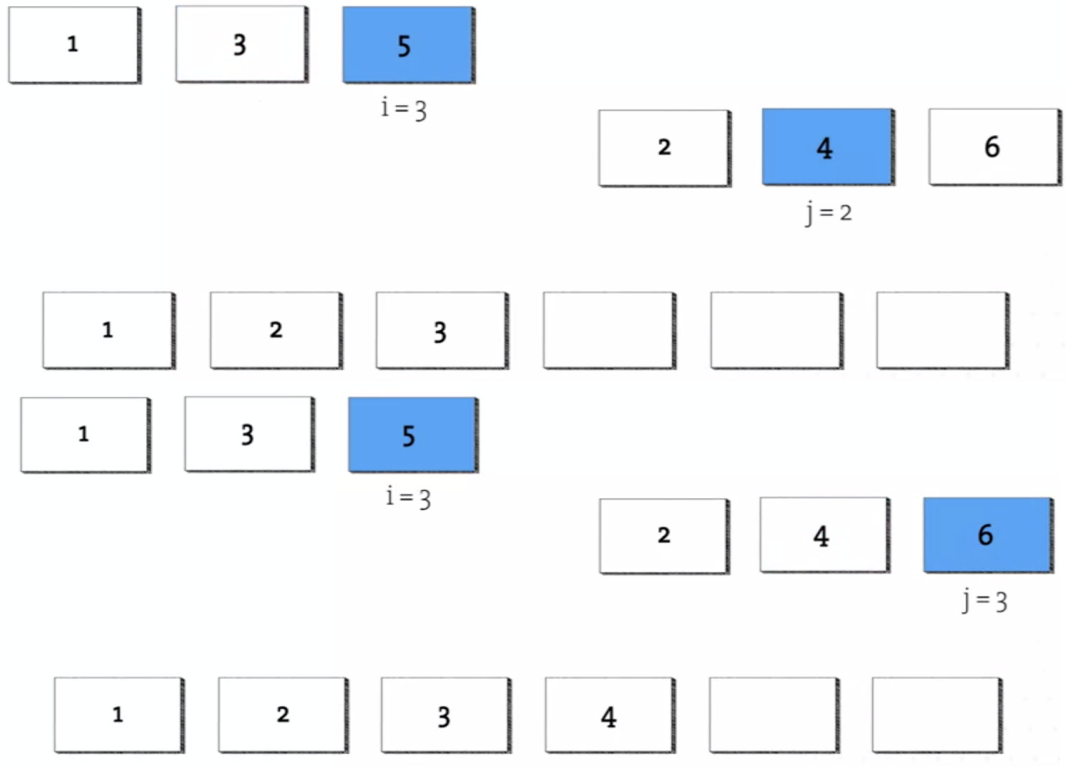
- Mergesort = already sorted vectors are merged. for example two smaller sorted vectors are merged to produce a larger sorted vector that stores the same values as the original two vectors.
    - First we create two variables i and j which are going to compare the elements of each vector so that we end up with a longer merged sorted vector. So for any sorted vector, we want to initially put the smallest value inside the larger vector

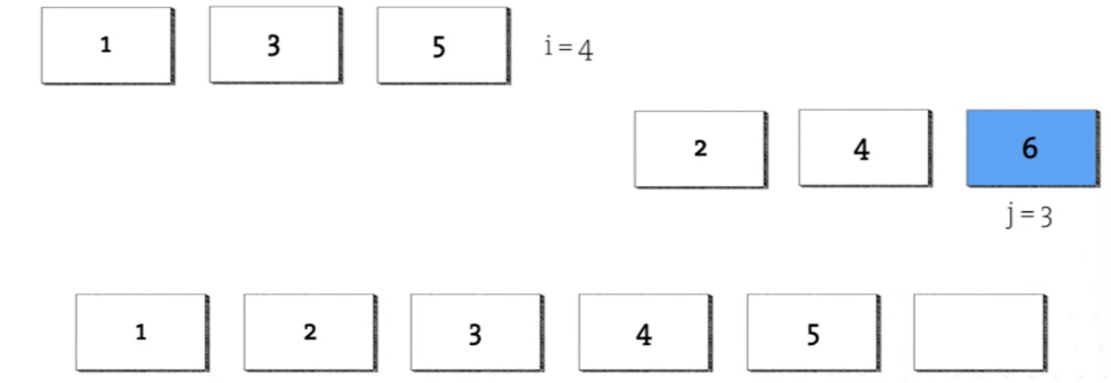


- Here we see that when comparing the two first values in both smaller vector, 1 is less than two, so we place 1 in the first element of the larger merged sorted vector and increment i by 1 so i = 2 but j is still at j = 1

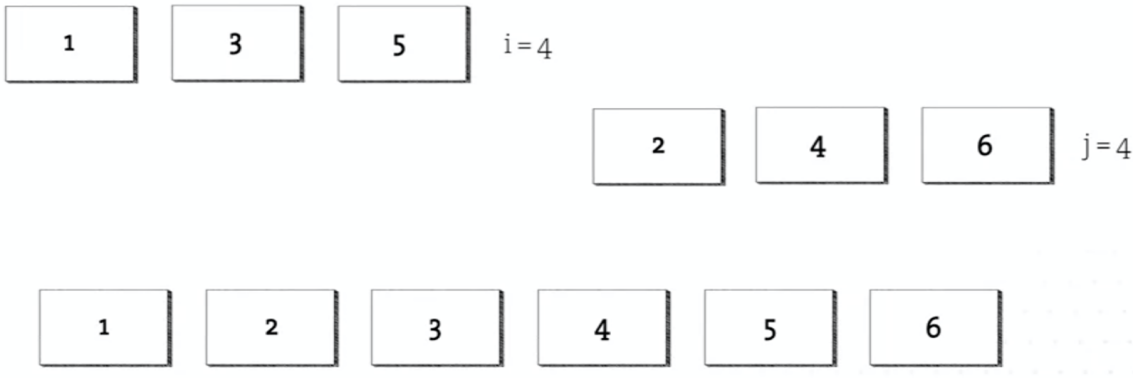


- So as we can see, we are going from left to right in the two vectors and comparing the elements one by one





- Here we see that i = 4 after incrementing which is larger than the size or length of the vector. So i has gone beyond the dimensions of the vector but j still hasn't



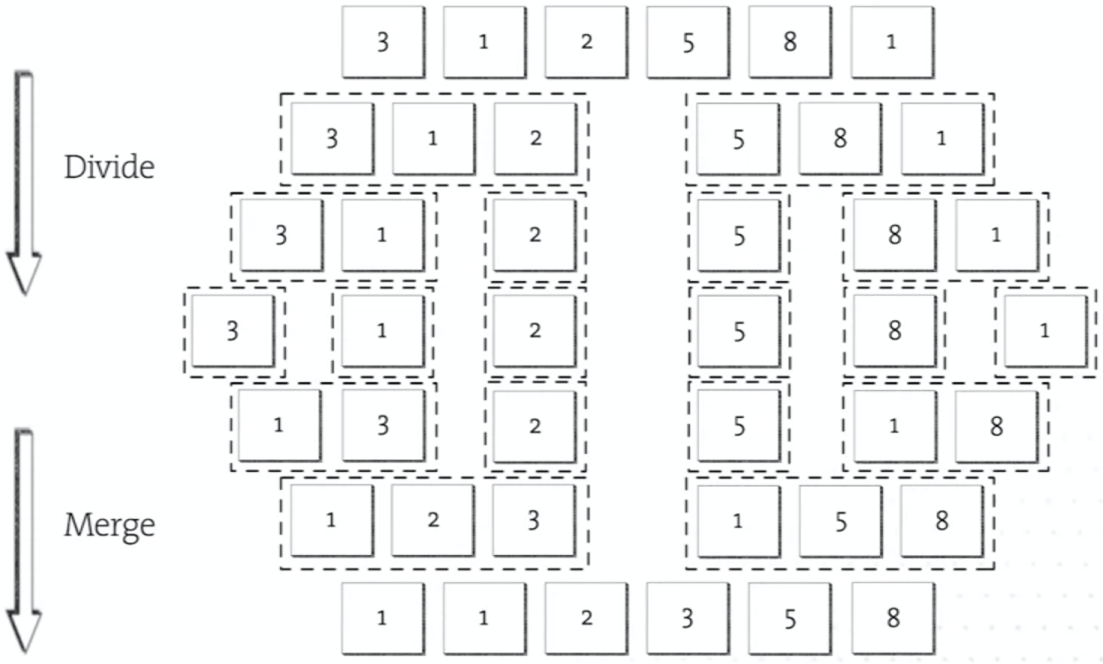
- So after placing the value 6 in the larger vector and incrementing j by one. We can see that we've exceeded the length of both smaller vector and we're done
- So to merge the two vectors in this way, we needed to create a new vector of the appropriate length and then scan from left to right of each vector comparing the values of both of them. We only needed to scan once from left to right to both, since the two vectors were already sorted

```
function Merge(w, v)
        m ← LENGTH[w]      n ← LENGTH[v]
        new Vector s(m + n)
        i ← 1    j ← 1    k ← 1
        while (i ≤ m)AND(j ≤ n) do
                if w[i] < v[j] then
                        s[k] ← w[i]
                        i ← i + 1
                else
                        s[k] ← v[j]
                        j ← j + 1
                end if
                k ← k + 1
        end while
        while i ≤ m do
                s[k] ← w[i]
                i ← i + 1      k ← k + 1
        end while
        while j ≤ n do
                s[k] ← v[j]
                j ← j + 1    k ← k + 1
        end while
        return s
end function
```

- Once we've established the merge procedure, the algorithm for mergesort works by dividing the initial vector into two by looking at the midpoint. A thing to notice here is that the division doesn't depend on the values stored in the vector (like QS or BSA)

```
function MergeSort(vector)
    n ← LENGTH[vector]
    if n = 1 then
        return vector
    end if
    m ← floor((n + 1)/2)
    new Vector L(m)
    new Vector R(n − m)
    L ← vector[1 : m]
    R ← vector[m + 1 : n]
    return Merge(MergeSort(L), MergeSort(R))
end function
```

- As shown by the function above, mergesort is applied to two smaller vectors and thus once sorted, they can be merged. The base case is when the input vector to the mergesort algorithm consists of one element. In this case, the sub-vector is already sorted and so we can proceed to merging it with other one element vectors



- As we see above from the picture, Mergesort can be shown as the procedure of dividing up our vector into other smaller vectors by halving them and then we merge them into a larger sorted vector

Summary

In this week, we learned about Quicksort and Mergesort.