Merging

- *Merging* means combining two sets of data which have already been sorted in such a way that the result is also sorted. For example, when you merge the lists 2 6 9 10 15 and 4 8 12, the result should be 2 4 6 8 9 10 12 15.

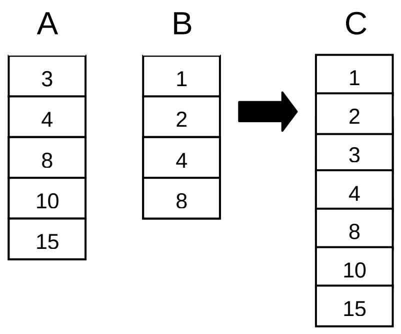
- When you are merging two lists, you should decide whether the duplicates should be eliminated or not. For instance, if you are trying to merge students of two sections according to their grades, duplicates must be repeated, because we don't want to lose students on the way, just because they got the same grade.

- There are two ways of merging two arrays. One is to use a separate array to store the merged list. The other way is to store the merged list in one of the two arrays. Now, you will learn the algorithms for both merging methods, starting with the first one.

Merging into a Separate Array

- This algorithm merges the elements of two sorted arrays and uses another array to store the merged values.

- For instance, assume that the arrays to be sorted are A and B arrays, and the result will be put into the C array as below, thus without duplicates.



```
void merge_into_C(int A[], int sizeA, // first list
                  int B[], int sizeB, // second list
                  int C[], int *sizeC) {   // result
     int kA = 0, kB = 0, kC = 0;
     /* until the end of A or B is reached */
     while (kA < sizeA && kB < sizeB)
         if (A[kA] < B[kB])
             C[kC++] = A[kA++];
         else if (B[kB] < A[kA])
             C[kC++] = B[kB++];
         else {
             C[kC++] = A[kA++];
             kB++;
         }
```

```
        /* if the end of A is reached, copy the remaining
           elements of B to the end of C */
        if (kA == sizeA)
            while (kB < sizeB)
                C[kC++] = B[kB++];
        /* if the end of B is reached, copy the remaining
           elements of A to the end of C */
        else
            while (kA < sizeA)
                C[kC++] = A[kA++];
        *sizeC = kC;
    }
```

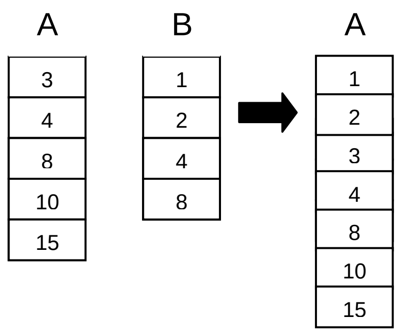
- How can we change our function so that the array `C` will contain the duplicates? If the same value is matched in both arrays, both of them should be put into the `C` array.

```
            {
                C[kC++] = A[kA++];
                C[kC++] = B[kB++];
            }
```

- Notice that, we don't need `sizeC` as an output parameter any more, because the size of the resultant array will be `sizeA + sizeB`.

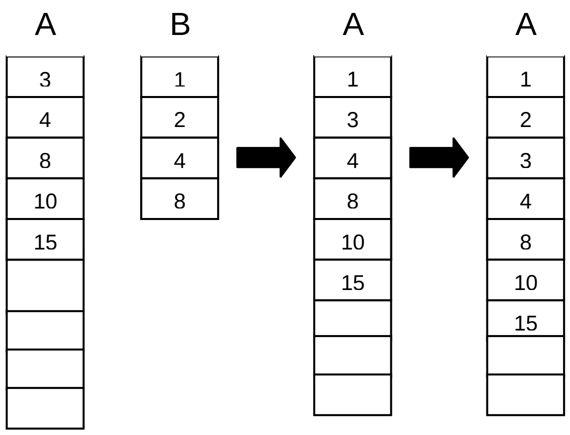
Merging into One of the Arrays

- This algorithm merges the elements of two sorted arrays and uses one of those arrays to store the merged values.
- For instance, assume that the arrays to be sorted are A and B arrays, and the result will be put into the A array as below, thus without duplicates.



- The hint here is to declare one of the arrays (e.g. array `A`) extra large in the main, so that there will be some empty space at the bottom of it to be used as free space to insert the elements of the other array (`B`).

- We will again compare the elements in both arrays as we did in the previous merge algorithm. However, when we decide that an element of `B` should be inserted into `A` at a certain position, we have to move all of the elements in the following positions of `A` one element down to make room for the element coming from `B`. This operation is named as *shift* operation.



```
void shift_down(int ar[], int size, int pos) {
     int k;
     for (k = size - 1; k >= pos; k--)
         ar[k+1] = ar[k];
}

void merge_into_A(int A[], int *sizeA,  //first(result) list
                  int B[], int sizeB) { // second list
     int kA = 0, kB = 0;
     /* until the end of A or B is reached */
     while (kA < *sizeA && kB < sizeB) {
         if (B[kB] < A[kA]) {
             shift_down(A, *sizeA, kA);
             A[kA] = B[kB++];
             *sizeA = *sizeA + 1;
         }
         else if (B[kB] == A[kA])
             kB++;
         kA++;
     }
     /* if the end of A is reached, copy the remaining
        elements of B to the end of A */
     if (kA == *sizeA)
         while (kB < sizeB)
         {
             A[kA++] = B[kB++];
             *sizeA = *sizeA + 1;
         }
}
```

- How can we change our function so that the array `A` will contain the duplicates? If the same value is matched in both arrays, it should also be inserted in the `A` array. Thus, we should add an equality in the first condition of the if statement, and remove the else part.

```
while (kA < *sizeA && kB < sizeB)
{
    if (B[kB] <= A[kA])
    {
        shift_down(A, *sizeA, kA);
        A[kA] = B[kB++];
        *sizeA = *sizeA + 1;
    }
    kA++;
}
```

- Again notice that, `sizeA` does not need to be a pointer any more, because it will become `sizeA + sizeB`.

- When we compare the two methods, the second method is more complex than the first one. However, the first one uses more memory, since we have to declare a third array with a size that is equal to the sum of the sizes of the two arrays. On the other hand, it does not destroy either of the original arrays, however the second one destroys one of the arrays.

- Additionally, the first one is faster than the second way. The second one is rather time consuming since it is necessary to shift the elements during the insertion operation.