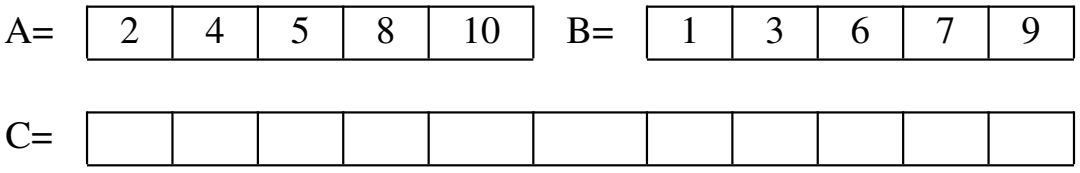
Merge Sort

- The idea behind **Mergesort** is the following:
  - **Divide** the array in half.
  - **Recursively** sort each half.
  - **Merge** the two sorted halves.
- The algorithm is as follows:

```
Mergesort(int *A,int left,int right) {
   if (left >= right) return
   int mid = (left + right)/2;
   Mergesort(a, left, mid);
   Mergesort(a, mid + 1, right);
   Merge(a, left, mid, right);
}
```

- So, how do we do the merge?
- **Example:**



Merge: Try 1

- **Merge** takes two sorted subarrays and produces a single sorted array.
- Assume we have 2 sorted arrays, and we want to combine them into a third array. **Merge** works as follows:
  - We have a pointer to the beginning of each subarray.
  - Put the smaller of the elements pointed to in the new array.
  - Move the appropriate pointer.
  - Repeat until new array is full.
- Here is some C++ code that merges two sorted arrays, A and B, into a third array C.

```
int p1=0, p2=0, index=0;
int n=sizeA+sizeB;
while(index<n) {
   if(A[p]) < B[p2]) {
        C[index]=A[p1];
        p1++; index++; }
   else {
        C[index]=B[p2];
        p2++; index++; }
   }
```

Merge: Try 2

- The problem with that code is that it assumes we have 3 different arrays.
- For mergesort, our two input arrays are part of a bigger array, and together they form the output array.
- We can use an auxiliary arrays (But read on–this is not quite correct):

```
void Merge(int *A,int L,int M,int R) {
   int B1=new int[M-L+1];
   int B2=new int[R-M];

   for (int i=0;i<M-L+1;i++)
      B1[i] = A[i+L];
   for (int i=0;i<R-M;i++)
      B2[i] = A[i+M+1];

   int b1=0; int b2=0;
   for(int k=L;k<=R;k++)
     if(B1[b1] < B2[b2])
         A[k]=B1[b1++];
     else
         A[k]=B2[b2++];
   }
}
```

Merge Problems and Solutions

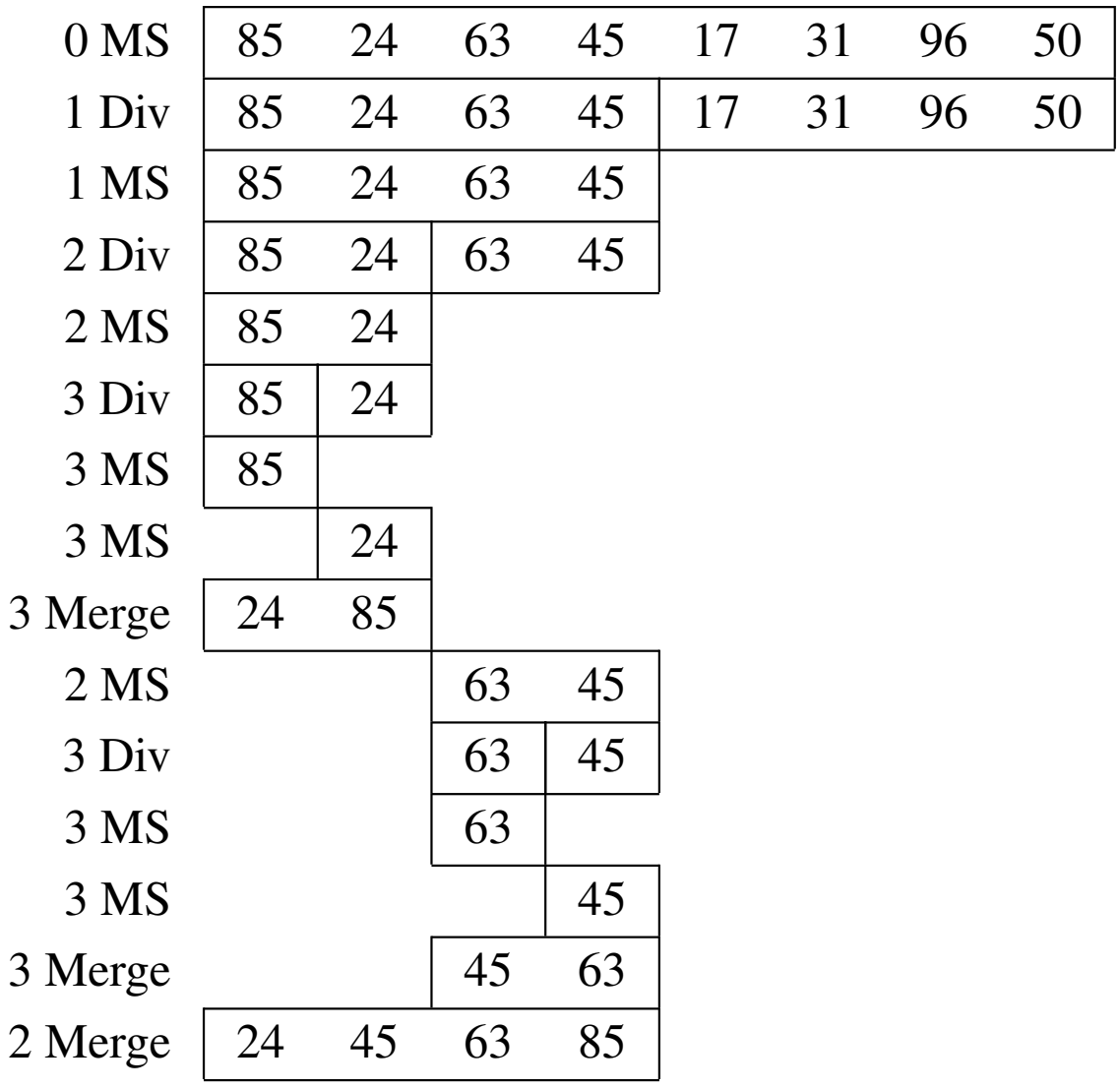
- The second attempt is more suited for our purpose, but both versions still have one major flaw. What is it?
- Neither of these does bound checking. There are 3 solutions that come to mind:
  - Do explicit bound checking.
  - Add a large element to the end of each input list (Sometimes called a sentinel).
  - Do something more clever.
- Here two ideas about how to be more clever:
  - Use a single auxiliary array. Copy the first half normally, and the second half of the array backwards. If you think about it, there is no need for bound checking if done this way. See the next page for implementation.
  - Use a single auxiliary array throughout the entire algorithm, swapping the subproblems back and forth. Although more complicated to implement, this can actually result in making the algorithm about twice as fast.

Merge: Try 3

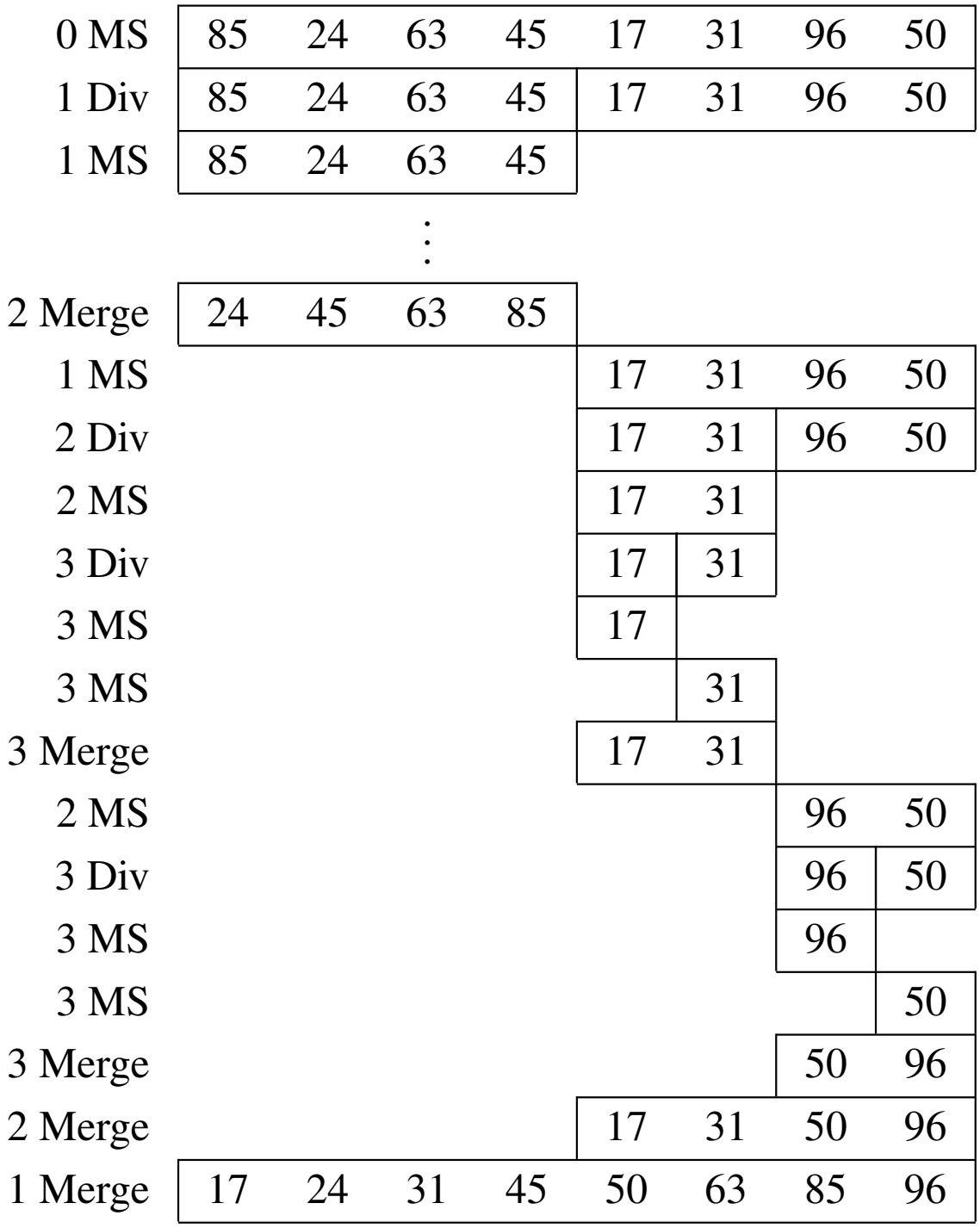
Here is an implementation of `Merge` that uses uses a single auxiliary array, with half copied normally, and half copied backwards.

```
void Merge(int A[],int L,int m,int R) {
    int size=R-L+1;
    int mid=m-L+1;

    // Copy the array
    int *B=new int[size];
    for(int i=0;i<mid;i++)
        B[i]=A[L+i]; // 1st half forward
    for(int j=mid;j<size;j++)
        B[j]=A[R-j+mid];  // 2nd half reversed

    // Now merge
    int i=0;
    int j=size-1;
    for(int k=L;k<=R;k++)
        if(B[i]<B[j]) // Notice: no bound check
           A[k]=B[i++];
        else
           A[k]=B[j--];

    delete []B; // In some implementations
}
```

Mergesort Example



Mergesort Example Continued



Complexity of Mergesort

- Let $T(n)$ be the time complexity of **Mergesort**.
- **Mergesort** always splits the list evenly, so the recursive depth of mergesort is always $O(\log n)$
- The amount of work done at each level is $O(n)$.
- In light of this, one would expect that $T(n) = O(n \log n)$.
- Indeed, $T(n) = O(n \log n)$:
  - It is no hard to see that

    $$T(n) = 2T(n/2) + \Theta(n),$$

    where $T(1) = \Theta(1)$.
  - We have seen many times before that the solution to this recurrence is $T(n) = \Theta(n \log n)$.
- The amount of extra memory used $O(n)$, which is acceptable.
- An important property of **Mergesort** is that it is stable.