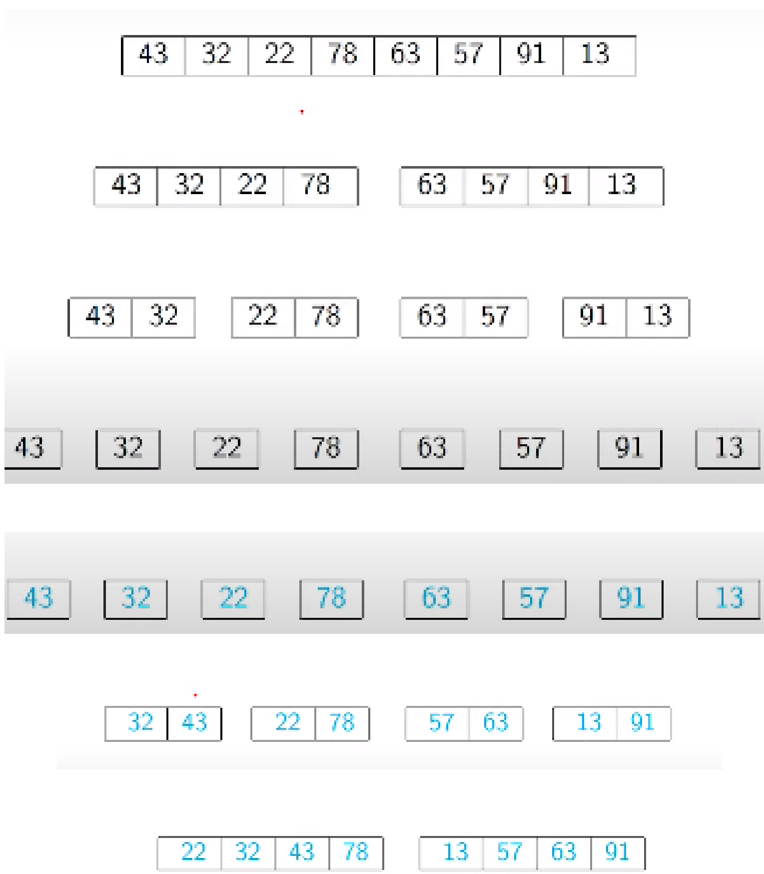
merge sort

Both Selection sort and Insert sort takes time `O(n^2)`

This is `infeasible for n > 10000`

strategy 3: Merge Sort

```
graph TD
  list --> Divide_in_two_halves --> List1
  Divide_in_two_halves --> List2
  List1 --> Compare_Sort --> Merge_list3
  List2 --> Compare_Sort --> Merge_list3
```

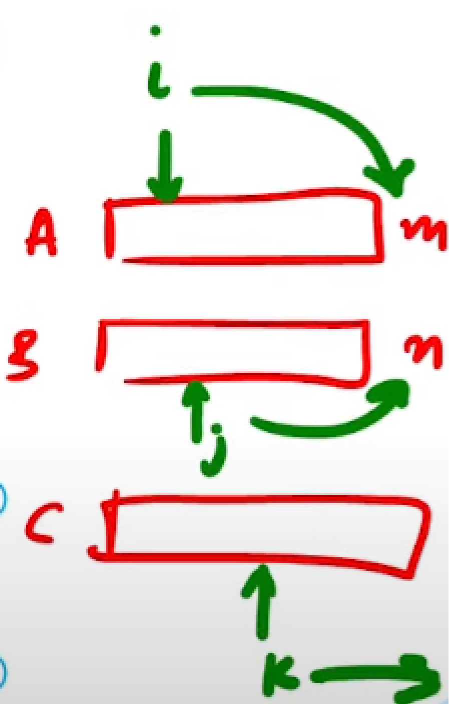


13	22	32	43	57	63	78	91

- Combine two sorted lists A and B into C
  - If A is empty copy B to C
  - if B is empty copy A to C
  - Otherwise compare first element of A and B
    - Move smaller of two to C
  - Repeat till all the elements are moved

```
def merge(A,B):
  (m,n) = (len(A),len(B))
  (C,i,j,k) = ([],0,0,0)
  while k < m+n:
    if i == m:
      C.extend(B[j:])
      k = k + (n-j)
    elif j == n:
      C.extend(A[i:])
      k = k + (n-i)
    elif A[i] < B[j]:
      C.append(A[i])
      (i,k) = (i+1,k+1)
    else:
      C.append(B[j])
      (j,k) = (j+1,k+1)
  return(C)
```

A empty



analysis:

- Merge A of `length m` and B of `length n`
- Output list C has length `m + n`
- In each iteration add one element to `C`
- Hence `merge` takes `O(m+n)`
- Recall that `m + n ≤ 2(max(m,n))`
- If `m~n` then merge takes time `O(n)`

```
def mergesort(A);
  n = len(A)

  if n <=1:
    return (A)

  L = mergesort(A[:n//2])
  R = mergesort(A[n//2:])

  B = merge(L,R)

  return(B)
```

- Recurrence
  - $T(0) = T(1) = 1$
  - $T(n) = 2T(n/2) + n$
- $$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\left[2T(n/4) + n/2\right] + n = 2^2 T(n/2^2) + 2n \\ &= 2^2\left[2T(n/2^3) + n/2^2\right] + 2n = 2^3 T(n/2^3) + 3n \\ &\vdots \\ &= 2^k T(n/2^k) + kn \end{aligned}$$
- When $k = \log n$, $T(n/2^k) = T(1) = 1$
- $T(n) = 2^{\log n} T(1) + (\log n)n = n + n \log n$
- Hence $T(n)$ is $O(n \log n)$

Drawbacks:

1. Merge needs to create a new list to hold the merged elements
2. No obvious way to efficiently merge two lists in place
3. Extra storage can be costly
4. Inherently recursive
    a. Recursive calls and returns are expensive