The ***Incremental*** design technique was employed in Insertion Sort, the elements to be sorted were inserted one at a time.

The ***Divide and Conquer*** design technique is recursive. To solve a problem, the algorithm ***recurses***, or calls itself, to solve a similar, but smaller, problem. Once the problem is small enough and reaches the ***base case***, the problem is solved directly, without recursion. Otherwise, in the ***recursive case***, there are three steps:

Divide the problem into one or more subproblems that are smaller instances of the same problem.

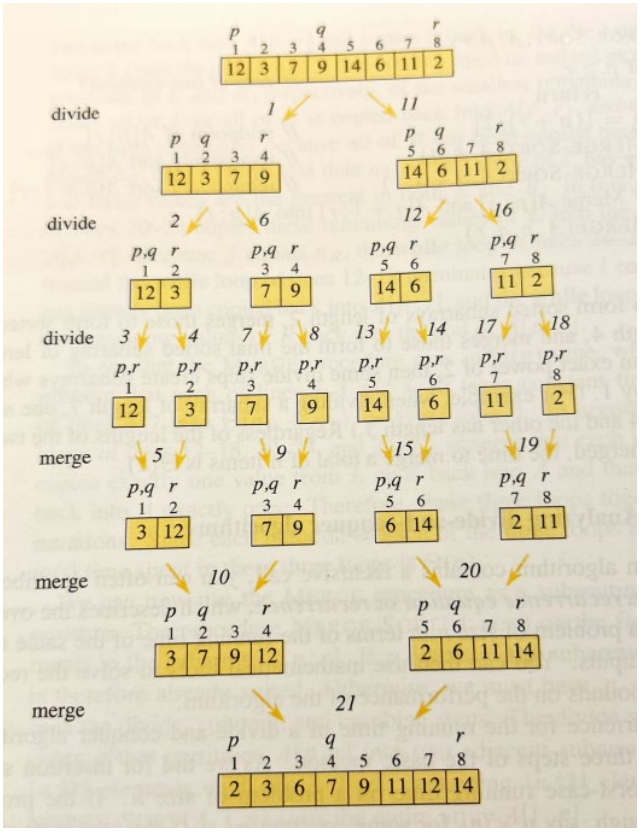
Conquer the problems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

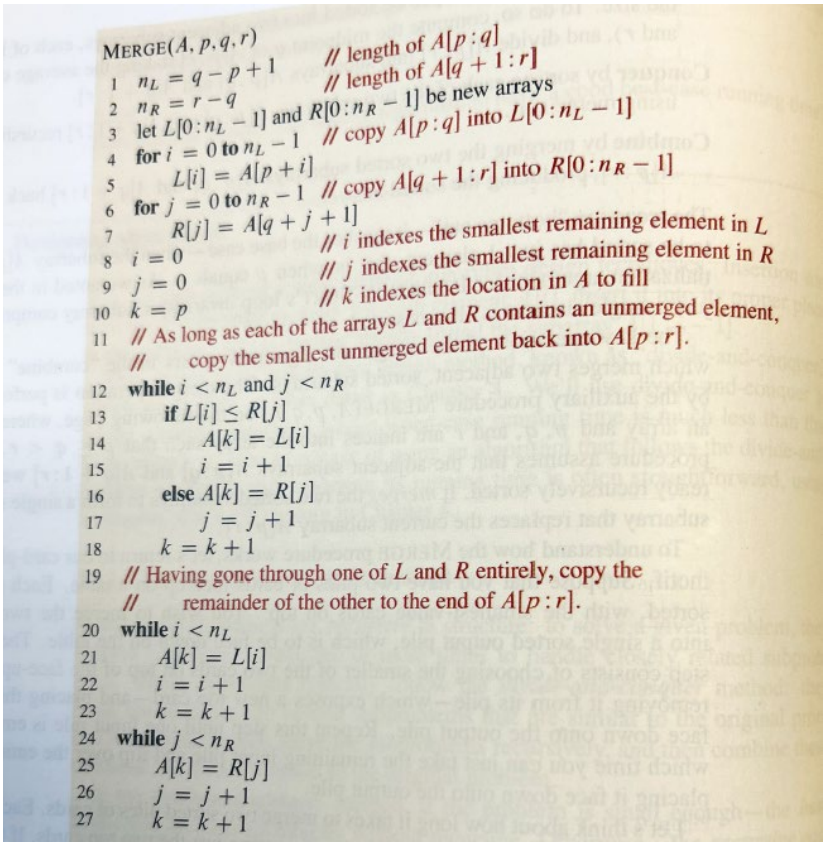
Merge sort is actually quite simple and elegant:

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Here is a sample execution of Merge sort, with the divide/conquer and combine (merge) steps of the recursion:



Merge does all the work! Note that it ***combines two sorted subarrays***.



```
MERGE(A, p, q, r)
1  n_L = q − p + 1          // length of A[p : q]
2  n_R = r − q              // length of A[q + 1 : r]
3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7      R[j] = A[q + j + 1]
8  i = 0                    // i indexes the smallest remaining element in L
9  j = 0                    // j indexes the smallest remaining element in R
10 k = p                    // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
   //     copy the smallest unmerged element back into A[p : r].
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //     remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. First, take two sorted lists and merge them. Try this for various sized lists.
4. Try the Mergesort algorithm on $N = 8$ unsorted cards.
5. What happens if N is not a power of 2?
6. What is Merge's (***not*** Mergesort) expected run time?

   No nested loops, and all the loops are on the order of the problem size ***n***.
7. Discuss Merge's (***not*** Merge sort) invariant?

   At the start of the while loop (lines 12-18), A[p , (k-1)] contains the k-p smallest elements of L[0 , n_L-1] and R[0 , n_R-1] in sorted order. In addition, R[j] and L[i] are the smallest elements not yet copied back into A.

To analyze Merge sort, and since it is recursive, we will use a recurrence relation. We start with the general recurrence relation for a divide and conquer algorithm:

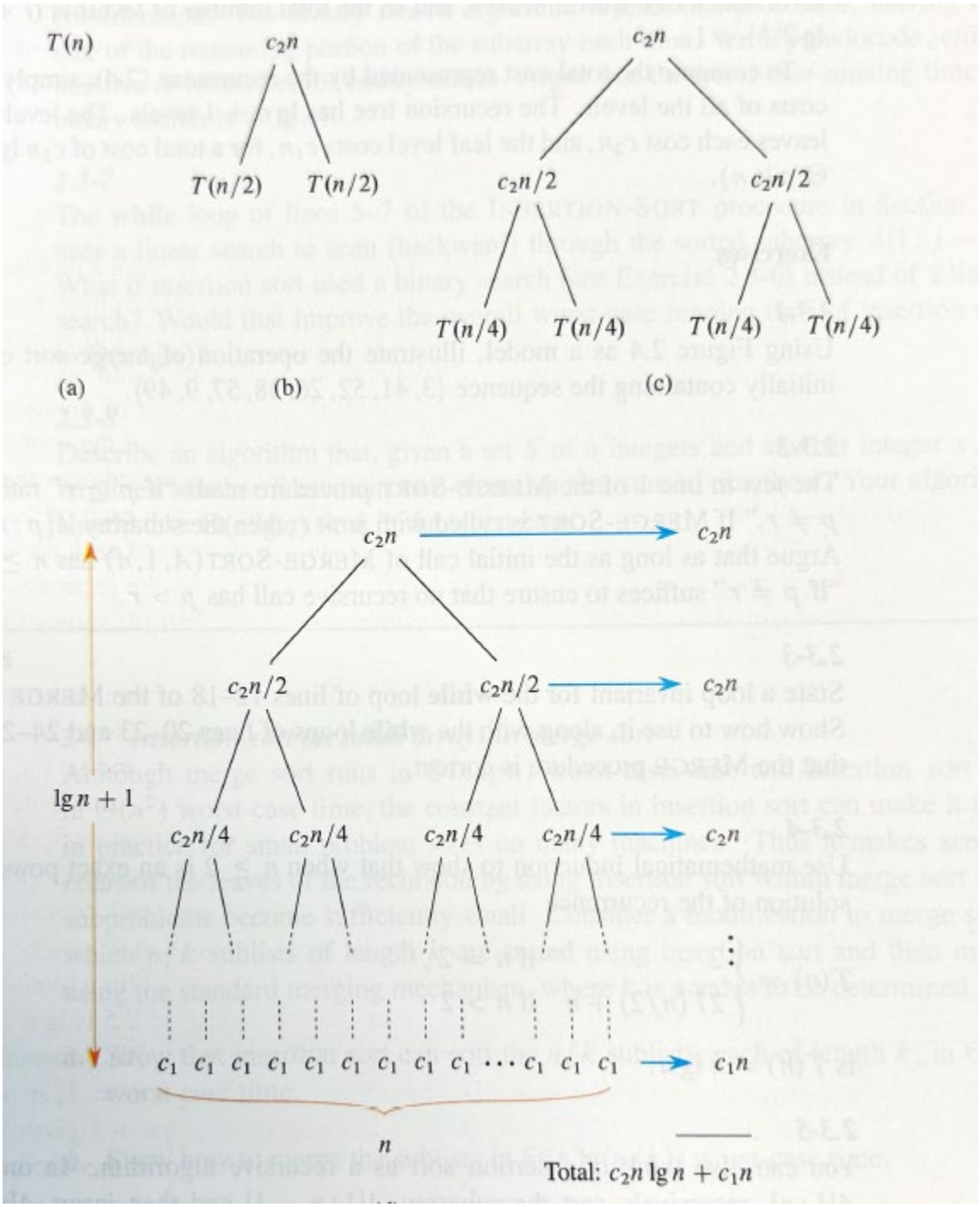
$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 , \\ D(n) + aT(n/b) + C(n) & \text{otherwise} . \end{cases}$$

The stopping criteria, when the problem size ***n***, is less than the criteria n_0 is constant amount of work, denoted by θ(1) or the constant ***c***. The divide, ***D(n)*** and conquer, ***C(n)***, are typically of size ***n***, and the problem is divided into ***a*** subproblems of size ***n/b***.

For Merge sort in particular, the divide step is just lines 3-5 in the above algorithm, and of constant time, Merge is on an ***n*** element subarray taking θ(***n***) time, and the problem is divided in half into to two subproblems, the resulting recurrence is:

$$T(n) = \begin{cases} c_1 & \text{if } n = 1, \\ 2T(n/2) + c_2 n & \text{if } n > 1, \end{cases}$$

Once we have the recurrence for an algorithm, we can create a sequence of recurrence trees to "guess" the solution.



To ***guess*** a solution, create enough recurrence trees showing the pattern, which helps you determine the amount of work on each level, and the number of levels. Since there are two subproblems, each node has two children, and each new subproblem is half the original size. Above, you can see the total amount of work on each level is ***n***.

To calculate the height, or the number of levels of the recurrence tree, a simple approach is to consider how many times the original problem size can be divided by 2 until it equals 1. The simplifying assumption here is that the problem size ***n*** is a power of 2, but this analysis still holds:

$$\frac{n}{2^h} = 1, n = 2^h, \lg(n) = \lg(2^h), \lg(n) = h$$

To wrap this up, ***n*** amount of work on ***lg(n)*** levels gives θ(***n*lg(n)***) amount of work for Merge sort. Note that this is only a guess. You will have a homework problem where you take this guess, and prove it using induction.

Starting with `MergesortPoC.ipynb`, which includes the pseudocode above, complete the `Mergesort` function (you only need to modify three lines!). Step through the code and output below:

The output is inside Mergesort, after the three calls:

```
    Mergesort()
    Mergesort()
    Merge()

    if print_flag:
      print(A)
```

```
N=8
A = list(range(N))
random.shuffle(A)
print(A)
Mergesort(A,0,N-1,True)
print(A)
```

```
[2, 3, 4, 0, 6, 5, 7, 1]
[2, 3, 4, 0, 6, 5, 7, 1]
[2, 3, 0, 4, 6, 5, 7, 1]
[0, 2, 3, 4, 6, 5, 7, 1]
[0, 2, 3, 4, 5, 6, 7, 1]
[0, 2, 3, 4, 5, 6, 1, 7]
[0, 2, 3, 4, 1, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 2, 3, 4, 5, 6, 7]
```