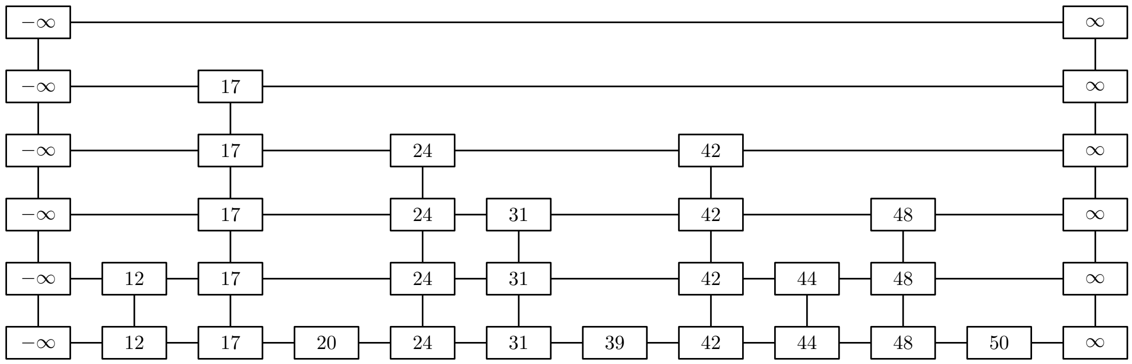
1. After the specified sequence of operation we will have the structure in the figure below.



2. The idea is to do a simultaneous binary search on both lists, where we keep the two indices summing to $k - 1$. The details are given in the python code below.

```
def k_small(A, B, k):
    n = len(A)
    m = len(B)

    # Handle base cases.
    if n == 0:
        return B[k - 1]
    if m == 0:
        return A[k - 1]
    if n + m == k:
        if A[n - 1] < B[m - 1]:
            return B[m - 1]
        else:
            return A[n - 1]

    # Set starting indices
    i = min((n - 1) / 2, k - 1)
    j = k - 1 - i
    # Cap j if too big
    if j >= m:
        j = m - 1
        i = k - 1 - j

    # Recurse
    if A[i] <= B[j]:
        if j == 0:
            return A[i]
        elif B[j - 1] <= A[i]:
            return A[i]
        else:
            return k_small(A[i:n], B[0:j], k - i)
    else:
        if i == 0:
            return B[j]
        elif A[i - 1] <= B[j]:
            return B[j]
        else:
            return k_small(A[0:i], B[j:m], k - j)
```

3. Construct a sequence of lists L_n by $L_0 = [0]$, $L_n = [n] + L_{n-1}$ when n is odd, and $L_n = L_{n-1} + [n]$ when n is even. So for example $L_5 = [5, 3, 1, 0, 2, 4]$. For this family of lists the pivots will be 1,2,3,4,... in order. This will give the $\Theta(n^2)$ runtime desired.

4. If we represent sets with bit vectors as proposed in the problem, then bitwise `and` computes the intersection, bitwise `or` computes the union, and bitwise `not` computes the complement of a sets. For set difference we use the formula $A \setminus B = A \cap \bar{B}$ where $\bar{B}$ is the complement of B. If we use a single word to store the bit vector, then all operations are $O(1)$. However, a more precise run time is $O(n/w)$ where w is the number of bits that can fit in a single word.

5. First we modify the `merge(A, B)` function in the merge sort algorithm. Every time that we take an element from `B` when `A` is nonempty there is an inversion. So we will count this event, and we will have the `merge` function return the number of inversions along with the merged list. Now we modify the `sort` function so that it returns the number of inversions along with the sorted list. Like before we will split the list into even halves, call `sort` on each half and merge the two halves. When the base case of a single element list or empty list happens we return zero inversions. To return the inversion count we sum the values returned from the two recursive calls to `sort` together with the value returned by `merge`.