Discussion about iterative and recursive algorithm:

(1) Read the section in textbook starting from P. 316
(2) Implement a lightweight recursive algorithm. Observe the error message shown when the program collapses. Keep it for future reference.
(3) If you have taken enough mathematics course, you are encouraged to study discrete Fourier transform (DFT). See how initially it is implemented using a recursive algorithm and how is turned into iterative control. Search keywords like butterfly algorithm.

Other comments:

(1) How to show numerically the complexity of the algorithm?
(2) If you have Matlab, run a few experiments with curve fitting. Another thing close to curve fitting is interpolation, which can be considered as a motivation for discrete Fourier transform.
(3) What is the break-even point of the algorithm?
(4) Constant associated with the complexities.

Counting sort assumes that each of the \( n \) input elements is an integer in the range 1 to \( k \), for some integer \( k \). When \( k = O(n) \), the sort runs in \( O(n) \) time.

The basic idea of counting sort is to determine, for each input element \( x \), the number of elements less than \( x \). This information can be used to place element \( x \) directly into its position in the output array.

We assume that the input is an array \( A[1, \ldots, n] \), and thus \( \text{length}(A) = n \). We require two other arrays: the array \( B[1, \ldots, n] \) holds the sorted output, and the array \( C[1, \ldots, k] \) provides temporary working storage.

**Counting Sort**:

```plaintext
1 for i ← 1, . . . , k
2    do C[i] ← 0
3 for j ← 1, . . . , length(A)
4    do C[A[j]] ← C[A[j]] + 1
5 for i ← 2, . . . , k
6    do C[i] ← C[i] + C[i - 1]
7 for j ← length(A), . . . , 1
8    B[C[A[j]]] ← A[j]
9    C[A[j]] ← C[A[j]] - 1
```

After the initialization in lines 1–2, we inspect each input elements in lines 3–4. If the value of an input element is \( i \), we increment \( C[i] \). Thus, after lines 3–4, \( C[i] \) holds the number of input elements equal to \( i \) for each integer \( i = 1, 2, \ldots, k \). In lines 5–6, we determine for each \( i = 1, 2, \ldots, k \), how many input elements are less than or equal to \( i \); Finally, in lines 7–9, we place each element \( A[j] \) in its correct sorted position in the output array \( B \). If all \( n \) elements are distinct, then when we first enter line 7, for each \( A[j] \), the value \( C[A[j]] \) is the correct final position of \( A[j] \) in the output.
array, since there are \( C[A[j]] \) elements less than or equal to \( A[j] \). Because the elements might not be distinct, we decrement \( C[A[j]] \) each time we place a value \( A[j] \) into the \( B \) array; this causes the next input element with a value equal to \( A[j] \), if one exists, to go to the position immediately before \( A[j] \) in the output array.

The complexity of counting sort. The for loop of lines 1–2 takes time \( O(k) \), the for loop for lines 3–4 takes time \( O(n) \), the for loop of lines 5–6 takes time \( O(k) \), and the for loop of lines 7–9 takes time \( O(n) \). Thus, the overall time is \( O(k + n) \). When we have \( k = O(n) \), the complexity is \( O(n) \).

Another important property of counting sort it that it is stable: numbers with the same value appear in the output array in the same order as they do in the input array.

The fundamental difference between counting sort and the other sorting algorithms discussed in the class is that counting sort does not rely on comparison. Can a sorting algorithm based on comparison achieve \( O(n) \) complexity?

Comparison sorts can be viewed abstractly in terms of decision trees. A decision tree represents the comparison performed by a sorting algorithm when it operates on an input of a given size. Control, data movement, and all other aspects of the algorithm are ignored.

The above figure shows the decision tree corresponding to the insertion sort algorithm operating on an input sequence of three elements \( a_1, a_2, a_3 \). In a decision tree, each internal node is annotated by \( a_i : a_j \) for some \( i \) and \( j \) in the range \( 1 \leq i, j \leq n \), where \( n \) is the number of elements in the input sequence. Each leaf node is annotated by a permutation of the input. The execution of the sorting algorithm correspond to tracing a path from the root of the decision tree to a leaf. At each node, a comparison \( a_i < a_j \) is made. The left subtree dictates subsequent comparisons for \( a_i < a_j \), and the right subtree dictates subsequent comparisons for \( a_i > a_j \). When we reach to the leaf, the sorting algorithm has established the ordering.
The length of the longest path from the root a decision tree to any of its leaves represents the worst-case number of comparisons the sorting algorithm performs. Consequently, the worst-case number of comparisons for a comparison sort corresponds to the height of its decision tree.

**Theorem:** Any decision tree that sorts $n$ elements has height $O(n \log n)$.

**Proof:** Consider a decision tree of height $h$ that sorts $n$ elements. Since there are $n!$ permutations of $n$ elements, each permutation representing a distinct sorted order, the tree must at have at least $n!$ leaves. Since a binary tree of height $h$ has no more than $2^h$ leaves, we have

$$n! \leq 2^h \Rightarrow h \geq \log n!.$$  

We introduce Stirling's approximation

$$n! > \left(\frac{n}{e}\right)^n,$$

thus

$$h \geq n \log n - n \log e = \Omega(n \log n).$$