1. (2.4) (a) The recurrent relation is $T(n) = 5T(n/2) + O(n)$. Using the master theorem, the complexity of the algorithm is $O(n^{\log_2 5})$.

(b) The recurrent relation is $T(n) = 2T(n - 1) + O(n)$. We cannot use the master theorem to solve this problem. Let’s solve it from scratch. We use the recursion tree method. Based on the recurrent relation, the tree has a branching factor of 2 and the height of the tree is $n$. At depth $k$, there are $2^k$ nodes and each subproblem has size $n - k$. The complexity for depth $k$ is $O(2^k(n - k))$. The overall complexity is $O(\sum_{k=0}^{n} 2^k(n - k)) = O(n(2^{n+1} - 1) - B)$. Ignoring the negative components, the result is $O(n2^n)$.

(c) The recurrent relation is $T(n) = 9T(n/3) + O(n^2)$. Using the master theorem, the complexity is $O(n^{2 \log 3})$.

We can see that the complexity of algorithm (c) is the lowest and (b) is the highest.

2. (2.14) We assume that the elements can be sorted in order. We first sort the $n$ elements in $O(n \log(n))$ and then we scan the sorted array to remove the duplicated elements in time $O(n)$.

3. (2.16) We first need to estimate $n$, the length of the sequence. We start from the guess of 1 and we double the guess until the last element is infinity. Apparently, we need no more than $\lceil \log(n) \rceil$ guesses to find out the (rough) length of the sequence. Now we can run the binary search on the first $n'$ elements of the array, where $n'$ is the estimated length of the array and $n'$ is not greater than $2n$. The binary search can be done in $O(\log(n))$. The overall complexity is therefore also $O(\log(n))$.

4. (2.17) The observation is that if $A(m) > m$ then it is impossible to have $A(i) = i$ for $i \geq m$. This is due to the assumption that the array is composed of sorted distinct integers. Using the divide and conquer scheme, we can split the array in half and use the relation of the middle element with its index to guide the search. The algorithm is as follows:

```python
function search(A, l, u)
    """INPUT: An array A with sorted distinct integers, l and u are the lower and upper bound of the indexes""
    OUTPUT: the index i such that A(i)=i, otherwise -1
    """
    if l > u
        return -1
    m = floor((l+u)/2)
    if A(m) = m
        return m
    else
        if A(m) > m
            return search(A, l, m)
        else
            return search(A, m, u)
```

1
5. (2.19) The merge algorithm has complexity $O(n + m)$, where $n$ and $m$ are the lengths of the two input sequences. If we merge the $k$ arrays sequentially, the complexity will be $O(2n + 3n + \ldots + kn)$ which is $O(k^2 n)$.

A better scheme is to merge in pairs in length $n$ and then in length $2n$ and then $4n$, ..., until we hit $kn$. We assume $k$ is a power of 2. The merge complexity becomes $O(2n + 4n + 8n + \ldots + mn)$, where $k = 2^m$. After some simplification, the complexity is $O(kn)$.

6. (2.20) We can pre-allocate an array $A$ whose length is $M + 1$ and initialize the array with Infinity. Note that $M$ can be obtained in time $O(n)$ even if we do not know the max and min values. Then we scan the array from 1 to $n$ and put integer $x[i]$ into $A[x[i]]$. We then go through $A$ again to output the sorted array.

Here, for small $M$, we have an algorithm better than $n \log(n)$. We in fact use more than the ordering of the integers. The advantage here is that we know the integers in the array are finite and can always be restricted in some bounds.

7. (2.21)

(a) If array $A$ has a majority element, then after we split it in half, at least one of them has a majority element. It takes linear time to check whether the majority element are the same in each half. We thus have a algorithm that recursively divides $A$ into halves and if both of the sub-arrays have the same majority element it returns true; if both are not, it returns false; otherwise we need to check whether the total number of the elements in $A$ that are the same as the majority element in one of the sub-arrays. We have a divide and conquer scheme that has complexity $n \log(n)$.

(b) There are at most half elements left since we discard at least one element in each pair (there may be single element “pair”).

To survive in each round, the non-majority element should try to pair with majority elements. Since there are more majority elements, after each round there will still be more majority elements left. Therefore, the decimated array has a majority element if $A$ does. It is also true the other way round. We now have an algorithm that repeats the procedure until we have 1 element left iff $A$ has a majority element or 0 element iff $A$ is not. It is easy to show that the overall complexity is linear.