(1) Exercise 8.2-3 on Page 196 [20 points]

COUNTING-SORT(A, B, k)
1 let C[0..k] be a new array
2 for i = 0 to k
3 C[i] = 0
4 for j = 1 to A.length
5 C[A[j]] = C[A[j]] + 1
6 // C[i] now contains the number of elements equal to i.
7 for i = 1 to k
8 C[i] = C[i] + C[i - 1]
9 // C[i] now contains the number of elements less than or equal to i.
10 for j = A.length downto 1
12 C[A[j]] = C[A[j]] - 1

8.2-3
Suppose that we were to rewrite the for loop header in line 10 of the COUNTING-SORT as

10 for j = 1 to A.length

Show that the algorithm still works properly. Is the modified algorithm stable?

We show this by an example. Suppose that there are two elements a1 and a2 with the same value and a1 appears before a2 in the input array A. In the original algorithm, if a2 is put into the output array B at position x, i.e., put a2 into B[x], then a1 will be put into B[x-1]. But in the modified algorithm, a2 is in B[x-1] and a1 is in B[x]. They are reversed in the output. For the purpose of sorting, the reversing is ok because a1 = a2. So the modified algorithm works properly.

The modified algorithm is not stable. Equal elements will appear in reverse order in the sorted array.

(2) Exercise 8.3-4 on Page 200 [20 points]
Clue: first convert each integer to base n, then radix sort them

8.3-4
Show how to sort n integers in the range 0 to \( n^3 - 1 \) in \( O(n) \) time.
Treat the numbers as 2-digit numbers in radix $n$. Each digit ranges from 0 to $n−1$. Sort these 2-digit numbers with radix sort.

There are 2 calls to counting sort, each taking $\Theta(n + n) = \Theta(n)$ time, so that the total time is $\Theta(n)$.

(3) Exercise 9.1-1 on Page 215 [20 points]

9.1-1
Show that the second smallest of $n$ elements can be found with $n + \lceil \lg n \rceil − 2$ comparisons in the worst case. (Hint: Also find the smallest element.)

The smallest of $n$ numbers can be found with $n − 1$ comparisons by conducting a tournament as follows: Compare all the numbers in pairs. Only the smaller of each pair could possibly be the smallest of all $n$, so the problem has been reduced to that of finding the smallest of $\lceil n/2 \rceil$ numbers. Compare those numbers in pairs, and so on, until there is just one number left, which is the answer.

To see that this algorithm does exactly $n − 1$ comparisons, notice that each number except the smallest loses exactly once. To show this more formally, draw a binary tree of the comparisons the algorithm does. The $n$ numbers are the leaves, and each number that came out smaller in a comparison is the parent of the two numbers that were compared. Each non-leaf node of the tree represents a comparison, and there are $n − 1$ internal nodes in an $n$-leaf full binary tree (see Exercise (B.5-3)), so exactly $n − 1$ comparisons are made.

In the search for the smallest number, the second smallest number must have come out smallest in every comparison made with it until it was eventually compared with the smallest. So the second smallest is among the elements that were compared with the smallest during the tournament. To find it, conduct another tournament (as above) to find the smallest of these numbers. At most $\lceil \lg n \rceil$ (the height of the tree of comparisons) elements were compared with the smallest, so finding the smallest of these takes $\lceil \lg n \rceil − 1$ comparisons in the worst case.

The total number of comparisons made in the two tournaments was $n − 1 + \lceil \lg n \rceil − 1 = n + \lceil \lg n \rceil − 2$ in the worst case.

(4) Exercise 9.3-1 on Page 223 [20 points]
Groups of 7
The algorithm will work if the elements are divided in groups of 7. On each partitioning, the minimum number of elements that are less than (or greater than) $x$ will be:

$$4 \left( \left\lfloor \frac{n}{7} \right\rfloor - 2 \right) \geq \frac{2n}{7} - 8$$

The partitioning will reduce the subproblem to size at most $5n/7 + 8$. This yields the following recurrence:

$$T(n) = O(1) \text{ if } n < n_0$$
$$T(n) = T\left( \left\lceil \frac{n}{7} \right\rceil \right) + T\left( \frac{5n}{7} + 8 \right) + O(n) \text{ if } n \geq n_0$$

We guess $T(n) \leq cn$ and bound the non-recursive term with $an$:

$$T(n) \leq cn/7 + c(5n/7 + 8) + an$$
$$\leq cn/7 + 5cn/7 + 8c + an$$
$$= 6cn/7 + 9c + an$$
$$\leq cn + (-cn/7 + 9c + an)$$
$$\leq cn = O(n)$$

The last step holds when $(-cn/7 + 9c + an) \leq 0$. That is:

$$-cn/7 + 9c + an \leq 0$$
$$\Rightarrow$$
$$c(n/7 - 9) \geq an$$
$$\Rightarrow$$
$$c(n-63)/7 \geq an$$
$$\Rightarrow$$
$$c \geq 7an/(n-63)$$

By picking $n_0 = 126$ and $n \leq n_0$, we get that $n/(n-63) \leq 2$.

Then we just need $c \geq 14a$.

Groups of 3
The algorithm will not work for groups of three. The number of elements that are less than (or greater than) the median-of-medians is:
The recurrence is thus:

\[ T(n) = T\left(\lceil \frac{n}{3} \rceil \right) + T\left(\frac{2n}{3} + 4\right) + O(n) \]

We're going to prove that \( T(n) = \Omega(n) \) using the substitution method. We guess that \( T(n) > cn \) and bound the non-recursive term with \( an \).

\[
T(n) > cn/3 + c(2n/3 + 4)c + an = cn + 5c + an > cn = \Omega(n) \quad (c > 0, a > 0, n > 0)
\]

The calculation above holds for any \( c > 0 \).

Another way to prove group of 3 is not work is:

If we use \( k \) elements as a group, the number of elements less than the median is:

\[
\left\lfloor \frac{k}{2} \right\rfloor \left( \left\lfloor \frac{1}{2} \left\lceil \frac{n}{k} \right\rceil \right\rfloor - 2 \right) \geq \frac{n}{3} - k
\]

In the worst case, we need to recursively call Select for \( n-(n/4-k)=3n/4+k \) \( k \) times. Thus we have:

\[
T(n) = T\left(\lceil n/k \rceil \right) + T\left(3n/4 + k\right) + O(n) \\
\leq c(\lceil n/k \rceil) + c(3n/4 + k) + O(n) \\
\leq c(n/k + 1) + 3cn/4 + ck + O(n) \\
= cn/k + 3cn/4 + ck + c + O(n) \\
\leq cn \\
If 1/k + 3/4 < 1
\]

Therefore, \( k \) must be greater than 4. Hence, the group size must be larger than 4 to make it linear.

(5) Exercise 9.3-8 on Page 223 [20 points]
Let’s start out by supposing that the median (the lower median, since we know we have an even number of elements) is in X. Let’s call the median value m, and let’s suppose that it is in X[k]. Then k elements of X are less than or equal to m and n − k elements of X are greater than or equal to m. We know that in the two arrays combined, there must be n elements less than or equal to m and n elements greater than or equal to m, and so there must be n − k elements of Y that are less than or equal to m and n−(n−k)=k elements of Y that are greater than or equal to m.

Thus, we can check that X[k] is the lower median by checking whether Y[n−k] ≤ X[k] ≤ Y[n−k+1]. A boundary case occurs for k = n. Then n − k = 0, and there is no array entry Y[0]; we only need to check that X[n] ≤ Y[1].

Now, if the median is in X but is not in X[k], then the above condition will not hold. If the median is in X[k′], where k′ < k, then X[k] is above the median, and Y[n−k+1] < X[k]. Conversely, if the median is in X[k′], where k′ > k, then X[k] is below the median, and X[k] < Y[n−k].

Thus, we can use a binary search to determine whether there is an X[k] such that either k < n and Y[n−k] ≤ X[k] ≤ Y[n−k+1] or k = n and X[k] ≤ Y[n−k+1]; if we find such an X[k], then it is the median. Otherwise, we know that the median is in Y, and we use a binary search to find a Y[k] such that either k < n and X[n−k] ≤ Y[k] ≤ X[n−k+1] or k = n and Y[k] ≤ X[n−k+1]; such a Y[k] is the median. Since each binary search takes \( O(\lg n) \) time, we spend a total of \( O(\lg n) \) time. Here is how we write the algorithm in pseudocode:

\[
\text{TWO-ARRAY-MEDIAN}(X, Y) \\
n \leftarrow \text{length}[X] \triangleright n \text{ also equals length}[Y] \\
\text{median} \leftarrow \text{FIND-MEDIAN}(X, Y, n, 1, n) \\
\text{if } \text{median} = \text{NOT-FOUND} \\
\hspace{1em} \text{then } \text{median} \leftarrow \text{FIND-MEDIAN}(Y, X, n, 1, n) \\
\text{return median} \\
\]

\[
\text{FIND-MEDIAN}(A, B, n, \text{low, high}) \\
\text{if } \text{low} > \text{high} \\
\hspace{1em} \text{then return NOT-FOUND} \\
\hspace{1em} \text{else } k \leftarrow \lfloor (\text{low} + \text{high})/2 \rfloor \\
\hspace{1em} \text{if } k = n \text{ and } A[n] \leq B[1] \\
\hspace{2em} \text{then return } A[n] \\
\hspace{1em} \text{elseif } k < n \text{ and } B[n−k] \leq A[k] \leq B[n−k+1] \\
\]
then return A[k]
else if A[k] > B[n − k + 1]
    then return FIND-MEDIAN(A, B, n, low, k − 1)
else return FIND-MEDIAN(A, B, n, k + 1, high)

source code:

```cpp
#include <iostream>
#include <assert.h>
using namespace std;

int MidNum(int *A, int l1, int r1, int *B, int l2, int r2)
{
    int mid1, mid2;
    assert(A != NULL && B != NULL && l1 <= r1 && l2 <= r2 && r1 - l1 == r2 - l2);
    // when two arrays both left one element
    if (l1 == r1 && l2 == r2)
    // when two arrays both left two elements
    if ((r1 - l1 + 1) % 2 == 0)
    {
        mid1 = (r1 + l1) / 2 + 1;
        mid2 = (r2 + l2) / 2;
    }
    else
    {
        mid1 = (r1 + l1) / 2;
        mid2 = (r2 + l2) / 2;
    }
    if (A[mid1] == B[mid2])
        return A[mid1];
    else if (A[mid1] > B[mid2])
        return MidNum(A, l1, mid1, B, mid2, r2);
    else
        return MidNum(A, mid1, r1, B, l2, mid2);
}

int main()
{
    int A[6] = {1, 3, 5, 6, 8};
}
int B[6]={2, 4, 6, 9, 11};

int m2=MidNum(A, 0, 4, B, 0, 4);
cout<<m2<<endl;

int A2[10]={17, 18, 28, 37, 42, 54, 63, 72, 89, 96};
int B2[10]={3, 51, 71, 72, 91, 111, 121, 131, 141, 1000};

int m=MidNum(A2, 0, 9, B2, 0, 9);
cout<<m<<endl;

return 0;
}