Homework 2.
Due: Thursday, June 3, 2021 before 11:59PM via Gradescope. Late submission with no penalty by Friday, June 4, 2021 before 11:59AM.

Problem 1 (Repeated element).

Design:
We will use a binary search approach. The base case is $| A | = 1$ or $| A | = 2$. If $| A | = 1$, simply return 0. If $| A | = 2$, check if the first and second elements are equal. If they are equal, return the element, otherwise return 0.
For the general case, we split the array from the middle, dividing it at index $| A | /2$. Then we check the element $A_{|A|/2}$ and see if it equals to the element before or after it. If it does equal to one of them, we return this element and the duplicate is found. Otherwise proceed to the next step.
Then we substract the last element in the first half of the array by the first element in the first half of the array and add it by 1.
If this does not equal to the length of the array then we know that the duplicate is in there. So we recursively solve the problem on this subarray. Otherwise we recursively solve the problem on the second half subarray.
Correctness:
The base case stops the design from an infinite loop. Our first step ensures that we won't miss the duplicate if it's in the middle.
Then our next step check to see if the duplicate exists in the first half subarray. If there is no duplicate of any number, the length of the array will equal to the last element - the first element + 1 since the array is a continous and increasing collection of natural numbers.
For exmaple:

$$A = \{2, 3, 4, 5, 6\}$$
$$| A |= 6 - 2 + 1 = 5$$

If a duplicate exists:

$$A = \{2, 3, 4, 5, 5, 6\}$$
$$| A |= 7 \neq 6 = 7 - 2 + 1$$

This step determines if we need to recursively move to the first or second half, as we determine which half does the duplicate exists.
Runtime analysis:
We reduce the input size by half on each recursive call, and comparing two elements and substracting two element are constant time so we do constant amount of work each call. So we got recurrence relation:

$$T(n) = T(n/2) + O(1)$$

By Master Theorem this gives us $O(logn)$ runtime complexity.

Problem 2 (Limited supply of coins).

(a)
Let $T[i, j, K = \{k_1, k_2, ..., k_n\}]$ be equal to 1 if you can make change for value $0 \leq j \leq V$, using the first $0 \leq i \leq n$ coins with a total of at most $\sum_{i=1}^{n} k_i, k \geq 0$ coins, or 0 otherwise.
(b)
The recurrence is

$$T[i, j, K] = max\{T[i-1, j, K], T[i, j-c_i, K_i - 1]\}$$

Which can be intrepreted as follows:
To make change of the given value j we either use the coin of denomination D[i] or we jump to the next denomination by substracting i by 1. If we do so, we substract j by c_i indicating that we use the coin for that much value. Also, we substract K by 1 at entry i indicating that we use the coin and now the coin of that denomination has one less coin.
Here are the base cases:
When $i = 0$, **return 0** because that means we tried all the denominations and no more denominations are avaliable.
When $j < 0$ **return 0** because after substracting the denomination it exceed the value and can't be used.
When $K_i = -1$, **return** 0 because it has pass the limitation of coins and we don't have that many coins.
When $j = 0$ and $K_i \neq -1$, **return 1** because we know we successfully make change out of that value and have not exceed the limitation.
(c)
Pseudocode:
Coin Change(i,V,$F = \{f_1, f_2, ..., f_n\}$)
IF Helper$(i, V, F) = 1$
RETURN TRUE
ELSE
RETURN FALSE
Helper(i, V, F)
IF $i = 0$ RETURN 0
IF $V < 0$ RETURN 0
IF $F[i] = -1$ RETURN 0
IF $V = 0$ AND $F[i] \neq -1$ RETURN 1
RETURN MAX(Helper$(i-1, V, F)$,Helper$(i, V - D[i], F = \{..., f_i - 1, ...\})$)
(d)
This table has a size of $n \cdot V$ and for each entry we compute $F[i]$ time at maximum.
Since the worst case in runtime to comput each entry is $max_i F[i]$ our runtime is:

$$O(n \cdot V \cdot max_i F[i])$$