homework 1

Exercise 1

```
Input: A (array of integers), left (start index), right (end index)
Output: Index of a peak element
Algorithm PeakFinder(A, left, right)
mid = (left + right) // 2
if (mid == left or A[mid] >= A[mid - 1]) and (mid == right or A[mid]
>= A[mid + 1]) then
        return mid
else if mid > left and A[mid - 1] > A[mid] then
        return PeakFinder(A, left, mid - 1)
else
        return PeakFinder(A, mid + 1, right)
```

base case:
for an array of size n = 1:

- the algorithm defines `mid = (left + right) // 2 = 0`. it immediately returns the `mid` value as it satisfies the condition `mid == left and mid == right`
- the algorithm correctly identifies the single element (which is the peak element).

strong induction hypothesis: we assume that the array can find the correct peak element in an array of size up to k (i.e. $n \leq k$)

inductive step: we try to prove that the array can correctly identify the peak element in the array of size $k + 1$.
in a full array, `mid` is defined as `(k + 1) // 2` in the worst case. so, the greatest length after an array is split will always be less than k.

- case 1: the `(mid == left or A[mid] >= A[mid - 1]) and (mid == right or A[mid] >= A[mid + 1])` is reached, and `mid` is returned. this case is reached

when:

- **the algorithm reached the edge** and it is larger than the element immediately to its left/right
- or the element at the index is greater than both elements adjacent to it

- case 2: if `A[mid - 1] > A[mid]`, so the element to the left of `mid` is greater than the element at `mid`:
  - the algorithm binary searches the subarray `A[left:mid - 1]` for the peak, which always has length *less than* k (as the algorithm splits the array into two parts).
  - according to the induction hypothesis, the algorithm is always able to find the peak element for any array of size less than k, so the peak element can be identified.
- case 3: if `A[mid - 1] <= A[mid]`, so the element to the left of `mid` is lesser than the element at `mid`:
  - the algorithm binary searches the subarray `A[mid + 1:right]` for the peak, which always has length less than k.
  - according to the induction hypothesis, the algorithm is always able to find the peak element for any array of size less than k, so the peak element can be identified.

in all cases, the algorithm is able to correctly identify the peak element in one of the two halves of the array, by repeatedly halving the search space. therefore, we have proved that the algorithm is correct.

Exercise 2

part a.

base case: $b_1 = 3$, which is less than 4
inductive hypothesis: assume that the inequality holds for the k^{th}: $b_k \leq 4^k$
inductive step: we try to prove $b_{k+1} \leq 4^{k+1}$ (1)

$b_k \leq 4^k$
$\iff 3b_k + 2 \leq 3 \times 4^k + 2$ (2)

$b_{k+1} = 3b_k + 2$ (3)
$4^{k+1} = 4 \times 4^k$ (4)

substituting (2) and (4) into (1):
$3 \times 4^k + 2 \leq 4 \times 4^k$
$\iff 2 \leq 4 \times 4^k - 3 \times 4^k$
$\iff 2 \leq 4^k$ which is always true for $k \geq 1$

thus, we prove that the inequality will always hold for all $n \geq 1$.

part b.

```
input: n, where n >= 1
output: the value for b_n

algorithm B(n):
        if n <= 1: return 3
        else:
                return 3 * B(n - 1) + 2
```

proof for algorithm correctness:

1. **base case**: the first item is correctly computed (3)
2. **inductive hypothesis**: assume that for an arbitrary k, the algorithm correctly computes the value for `B(k)`
3. prove for $k+1$: `B(k + 1) = 3 * B(k) + 2`, and since the algorithm correctly calculates the value for `B(k)`, the correct value of `B(k + 1)` is reached.

thus, with induction, we have proven the algorithm's correctness.

Exercise 3

use the loop invariant technique to prove Gnome sort.

1. pick a loop invariant:

at any given position k, the elements in `A[0:k - 1]` is sorted.

2. **initialization**: for k = 1, the array with one element `A[0:0]` is trivially sorted.
3. **maintenance**:
   - while in the loop, Gnome sort:
     - if `pos == 0 or A[pos - 1] <= A[pos]`: the elements are in the correct sorted order, so the `pos` index advances by 1. so, now `pos = pos + 1`, and `A[0:pos]` is sorted.
     - if the condition is broken, that means the element is out of order. Gnome sort will move the element back from `pos` to `pos - 1`, and so on. since all elements from `A[0:pos - 1]` has been sorted, as `pos` decreases and the out-of-order element is swapped back, the subarray `A[0:pos - 1]` still remains sorted.

     $\rightarrow$ at any position k in the array, all previous elements of the array has been sorted.
4. **termination**: because the algorithm terminates when `pos == n`, the loop is guaranteed to terminate correctly.

thus, we have proven the algorithm's correctness.