There are 55 regular points available on this problem set.

1. (10 pts) *You are given two arrays of integers A and B, both of which are sorted in ascending order. Consider the following algorithm for checking whether or not A and B have an element in common.*

```
findCommonElement(A, B) :
    # assume A,B are both sorted in ascending order
    for i = 0 to length(A) {                    # iterate through A
        for j = 0 to length(B) {                # iterate through B
            if (A[i] == B[j]) { return TRUE }  # check pairwise
        }
    }
    return FALSE
```

   (a) *If arrays A and B have size n, what is the worst case running time of the procedure* `findCommonElement`*? Provide a Θ bound.*

   $\Theta(n^2)$.

   (b) *For $n = 5$, describe input arrays A_1, B_1 that will be the best case, and arrays A_2, B_2 that will be the worst case for* `findCommonElement`*.*

   The best case is when the very first element of the arrays are the same: $A_1 : [0, 1, 2, 3, 4]$ and $B_1 : [0, 2, 4, 5, 6]$. The worst case is when no elements are in common: $A_2 : [1, 2, 5, 6, 10]$ and $B_2 : [0, 3, 4, 7, 9]$.

   (c) *Write pseudocode for an algorithm that runs in $\Theta(n)$ time for solving the problem. Your algorithm should use the fact that A and B are sorted arrays.*
   (Hint: repurpose the `merge` *procedure from MergeSort.)*

```
findCommonElement(A, B) :
    # assume A, B are both sorted in ascending order
    i = 0
    j = 0
    while (i < length(A) and j < length(B)) {
        if (A[i] == B[j]) {
            return True
        }
        if (A[i] < B[j]) {
            i = i + 1
        }
        else if (A[i] > B[j]) {
            j = j + 1
        }
    }
    return FALSE
```

2. (15 pts) *Suppose we are given an array A of historical stock prices for a particular stock. We are asked to buy stock at some time i and sell it at a future time $j > i$, such that both $A[j] > A[i]$ and the corresponding profit of $A[j] - A[i]$ is as large as possible.*

   For example, let $A = [7, 3, 4, 2, 15, 11, 16, 7, 18, 9, 11, 10]$. If we buy stock at time $i = 3$ with $A[i] = 2$ and sell at time $j = 8$ with $A[j] = 18$, we make the maximum profit of $18 - 2 = 16$ megabucks. (Note that "short positions," where we sell stock before buying it back, i.e., where $j < i$, is not allowed here.)

   (a) Consider the pseudocode below that takes as input an array A of size n:

```
makeMaxProfitInHindsight(A) :
    maxProfitSoFar = 0
    for i = 0 to length(A)-1 {
        for j = i+1 to length(A) {
            profit = A[j] - A[i]
            if (profit > maxProfitSoFar) { maxProfitSoFar = profit }
    }}
    return maxProfitSoFar
```

What is the running time complexity of the procedure above? Write your answer as a Θ bound in terms of n.

The procedure has two nested loops with running time given by

$$\sum_{i=0}^{n-1}(n-i) = \frac{n(n+1)}{2} = \Theta(n^2).$$

(b) *Explain under what circumstances the algorithm in* (2a) *will return a profit of 0. Two sentences should suffice in your answer.*

The algorithm returns 0 whenever the array A is non-increasing, i.e. $A[i] \geq A[i+1]$ for all indices i of the array, wherein $0 \leq i < n-1$.

(c) *Write pseudocode that would calculate a new array B of size n such that*

$$B[i] = \min_{0 \leq j \leq i} A[j] \ .$$

In other words, $B[i]$ should store the minimum element in the subarray of A with indices from 0 to i, inclusive.
What is the running time complexity of the pseudocode to create the array B? Write your answer as a Θ bound in terms of n.

```
def createArrayB(A):
    n = length(A)
    B = fill(0, n)  # Create an array of all 0s of length n
    minSoFar = A[0]
    for i = 0 to n:
        minSoFar = min(A[i], minSoFar)
        B[i] = minSoFar
```

The time complexity is $\Theta(n)$.

(d) *Use the array B computed from* (2c) *to compute the maximum profit in time $\Theta(n)$.*

```
def computeMaxProfit(A, B):
    maxProfitSoFar = 0
    for i = 0 to n:
        profitAtI = A[i] - B[i]
        maxProfitSoFar = max(maxProfitSoFar, profitAtI)
    return maxProfitSoFar
```

(e) *Rewrite the algorithm above by combining parts* (2b)–(2d) *to avoid creating a new array* B.

The full pseudocode for the combined method is

```
def makeMaxProfitInHindsight(A):
    maxProfitSoFar = 0
    minSoFar = A[0]
    for i = 0 to n:
        minSoFar = min(minSoFar, A[i])
        profitAtI = A[i] - minSoFar
        maxProfitSoFar = max(maxProfitSoFar , profitAtI)
    return maxProfitSoFar
```

We have combined the previous two parts to provide a $\Theta(n)$ procedure that avoids creating a new array.

3. (15 pts) *Consider the problem of linear search. The input is a sequence of* n *numbers* $A = \langle a_1, a_2, \ldots, a_n \rangle$ *and a target value* v. *The output is an index* i *such that* $v = A[i]$ *or the special value NIL if* v *does not appear in* A.

(a) *Write pseudocode for a simple linear search algorithm, which will scan through the input sequence* A, *looking for* v.

```
linear-search(A, v)
for i = 1 to length[A]
    if (A[i] == v) then { return i }
return NIL
```

(b) *Using a loop invariant, prove that your algorithm is correct. Be sure that your loop invariant and proof covers the initialization, maintenance, and termination conditions.*

To prove the correctness of `linear-search`, we use a loop invariant.

Loop invariant: At the start of the ith pass through the `for` loop, for all $j < i$, it is true that `A[j]` $\neq$ `v`.

- **Initialization**: When `i=1`, before the first pass through the loop, the loop invariant holds because the set of elements `A[j]` for $j < i$ is an empty set and thus cannot contain `v`.
- **Maintenance**: (Proof by contradiction) Assume that the loop invariant breaks on the ith pass through the `for` loop. This would imply that there exists some $j < i$ such that `A[j]=v`. If this were true, then on the jth pass of the loop, the inner conditional would have been true, and the algorithm would have returned j, preventing us from reaching the $i > j$ iteration. Thus, if we have not returned yet, the loop invariant must still hold.
- **Termination**: The loop may terminate for one of two reasons. (i) It terminates after `i` $\leq$ `length(A)` iterations, and returns `i`; in this case, the `if` conditional must have been true, and `A[i] == v`. Or (ii) `i` exceeds `length(A)`; in this case, by the loop invariant, we have that for all `j` $\leq$ `length(A)`, it is true that `A[j]` $\neq$ `v`; thus, returning `NIL` is correct.

Hence, `linear-search` is correct. □

4. (15 pts total) *Crabbe and Goyle are arguing about binary search. Goyle writes the following pseudocode on the board, which he claims implements a binary search for a target value* `v` *within input array* `A` *containing* n *elements.*

```
bSearch(A, v) {
   return binarySearch(A, 0, n, v)
}

binarySearch(A, l, r, v) {
   if l >= r then return -1
   p = floor( (l + r)/2 )
   if A[p] == v then return m
```

```
   if A[m] < v then
     return binarySearch(A, m+1, r, v)
     else return binarySearch(A, l, m-1, v)
}
```

(a) *Help Crabbe determine whether this code performs a correct binary search. If it does, prove to Goyle that the algorithm is correct. If it is not, state the bug(s), give line(s) of code that are correct, and then prove to Goyle that your fixed algorithm is correct.*

There are **three** errors in the code given.

The first error is `n` $\rightarrow$ `n-1`, within the `bSearch` function, which correctly gives the length of the `A` array as n elements instead of $n + 1$.

The second error is the use of the variable `p` instead of `m`; thus, we replace all instances of `p` with `m`.

The third error is in the base case; the original code said `if l >= r then return -1`, which would have returned `-1` even if the target was in the last element of the array, i.e., when `l == r`.

The corrected code is as follows:

```
bSearch(A, v) {
   return binarySearch(A, 0, n-1 , v)
}

binarySearch(A, l, r, v) {
   if l > r then return -1
   p = floor( (l + r)/2 )
   if A[p] == v then return p
   if A[p] < v then
     return binarySearch(A, p+1, r, v)
   else return binarySearch(A, l, p-1, v)
}
```

We'll prove this algorithm's correctness using *strong induction*, i.e., we will show that `binarySearch()` is correct when `(r-l+1)=n`, for all `n`.

The base cases are `n=0` and `n=1`. In the former situation, we have `l > r`, and the algorithm halts on the first conditional with a `return -1`, which is correct. If the

list is empty, then the target cannot be a member of it. In the latter situation, `r == l` and the list contains one element. Here, `p=0`, and if the list contains the target, we return `p`, the correct behavior. If the list does not contain the target, then we make a recursive call, both of which terminate in the `n=0` base case, which yields the correct behavior.

Now we assume that `binarySearch()` is correct when the length of the input subarray is `(r-l+1)=k` (the inductive hypothesis), and we will show that it is also correct for a slightly larger subarray, with length `(r-l+1)=k+1`. There are three cases to handle: `A[p]==v`, `A[p]>v`, and `A[p]<v`.

Case `A[p]==v`: The algorithm has the correct behavior, and returns `p`.

Case `A[p]>v`: Because `A` is sorted, it must be true that the target is either within the subarray `A[l..p-1]` or is not in `A` at all. Thus, if the recursive call behaves correctly, then so too will this call. All that remains is to show that the recursive call is to a subarray that has smaller size than our current one (and then invoke the inductive hypothesis). The size of `A[l..p-1]` is `p-l=floor( (l + r)/2 )-l`. If `l + r` is even, then the size is `(r -l)/2`, and if `l + r` is odd, then the size is `(r-l-2)/2`, both of which are strictly smaller than our current array length of `r-l+1`.

Case `A[p]<v`: The logic here is symmetric to the `A[p]>v` case, except that the target is either within the subarray `A[p+1..r]` or is not in `A` at all. The calculations are also symmetric, and the conclusions the same.

Thus, by strong induction, we know that our corrected binary search algorithm is indeed correct. □

NB: There are other ways to prove this algorithm is correct. All work on some kind of *invariant*, a property that is true each time we make a recursive call. The complementary invariant to the one used above would be to observe that because `A` is sorted, each time we make a recursive call, we are increasing the size of the set of elements in which we know the target cannot lay within.

(b) *Goyle tells Crabbe that binary search is efficient because, at worst, it divides the remaining problem size in half at each step. In response Crabbe claims that trinary search, which would divide the remaining array A into thirds at each step, would be even more efficient. Explain who is correct and why.*

The exact number of comparisons performed during binary search is at most $\log_2 n$, while the exact number performed using "ternary" search is at most $2\log_3 n$, because it takes 2 comparisons to determine which of the three sub-arrays the target value must lie within. (This counting doesn't account for other, lower-order terms in the running time of the algorithms.)

And, $2\log_3 n = 2\left(\frac{\log 2}{\log 3}\right)\log_2 n \approx 1.262\log_2 n > \log_2 n$.

This implies that the exact number of comparisons done using ternary search exceeds that of binary search. However, in algorithms, we don't care about exact counts, only asymptotic comparisons, and $\log_3 n = \Theta(\log_2 n)$. Hence, binary search and ternary search are equivalent, and Crabbe is incorrect.