Adam Ten Hoeve
adam.tenhoeve@colorado.edu

CSCI 3104 Spring 2017 Problem Set 1
Adam Ten Hoeve (09/11)

---

1. *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])                   # check pairwise
            {
                return TRUE
            }
        }
    }
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.*

   SOLUTION: The worst case would be if there is no matching element in the two arrays. This means that the worst case is $O(n^2)$ and a close bound of $\Theta(n^2)$.

   (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`*.*

   SOLUTION: The best cases would be if the first element in array A_1 is the same as the first element in B_1. The worst case would be if the last element in A_2 was the same as the last element in B_2, or if there were no matching elements between the arrays.

(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.)*

SOLUTION:

```
findCommonElement(A, B)
{
    int i = 0, j = 0
    for index = 0 to legnth(A)
    {
        if (A[i] == B[j])
        {
            return TRUE
        }
        else if (A[i] < B[j])
        {
            i++;
        }
        else
        {
            j++;
        }
    }
    return FALSE
}
```

2. *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.

   SOLUTION: The complexity would be $\Theta(n^2)$ because there is a nested loop.

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

   SOLUTION: The function would return a profit of 0 if all of the values in the array are the same or are sorted in descending order. If these were true, then maxProfitSoFar would always be greater than profit and would never be reassigned a value other than 0.

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

$$B[i] = \min_{0 \le j \le i} A[j] \tag{1}$$

   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.

   SOLUTION:

```
makeMinSubArray(A)
{
    B[length(A)]
    B[0] = A[0]
    for (i = 1 to length(A)-1)
    {
        min = A[i]
        if (min < B[i-1])
        {
```

```
                B[i] = min
            }
            else
            {
                B[i] = B[i-1]
            }
        }
        return B
    }
```

The running time complexity of this pseudocode is $\Theta(n)$.

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

```
maxProfitWithB(A, B)
{
    max = 0
    for i = 0 to length(A)
    {
        temp = A[i] - B[i]
        if (temp > max)
        {
            max = temp
        }
    }
    return max
}
```

The best buy time would be at A[3] = 2, the lowest point in the array. After i=3, every position in B is 2. The best sell point would be at A[8] = 18. The max profit would be A[8] - B[8]= 18 - 2 = 16.

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

```
maxProfit(A)
{
    min = A[0]
    max = 0
    for i = 0 to length(A)
    {
        if (min < A[i])
        {
            min = A[i]
        }
        temp = A[i] - min
        if (temp > max)
        {
            max = temp
        }
```

```
        }
        return max
}
```

Instead of using a second array, the algorithm tracks the lowest value seen in the array in the variable min and then compares that to the following values in the array to find the max profit.

3. *(15 pts) Consider the problem of linear search. The input is a sequence of n numbers A = <a_1, a_2, . . . , a_n> 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.*
   SOLUTION:

```
linearSearch(A, v)
{
    for (i = 0 to length(A))
    {
        if (A[i] == v)
        {
            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.*
   SOLUTION: The loop invariant would be that for k < i, A[k] != v assuming v is in A.

   Initialization: i = 0 so everything less than i is an empty set, so A[k] != v.

   Maintenance: Let k < i < n. 2 cases:

   i. If A[i] != v, then A[k] != v because i must have passed through k and if it did not equal v at the time, it could not equal v afterwards. Loop invariant holds.
   ii. If A[i] == v, then A[k] != v because i != k. Loop invariant holds.

   Termination: If loop terminates, either:

   i. A[i] == v so i returned.
   ii. i > length(A) and v is not found. If not found, algorithm returns NIL, which is intended behavior.

4. *(15 pts) 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.*

```
1.  bSearch(A, v) {
2.      return binarySearch(A, 0, n, v)
3.  }
4.
5.  binarySearch(A, l, r, v) {
6.      if l >= r then return -1
7.      p = floor( (l + r)/2 )
8.      if A[p] == v then return m
9.      if A[m] < v
10.         return binarySearch(A, m+1, r, v)
11.     else
12.         return binarySearch(A, l, m-1, v)
13. }
```

(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.*

SOLUTION: The binary search is not correct. There are three problems.

i. The algorithm switches the variable p to m on line 8 and uses m instead of p for the rest of the function.

ii. On line 6, l ¿= r is incorrect. It should be l ¿ r because if l l = r then the program is still functional and can find v.

iii. On line 2, the first iteration should not pass in n but rather n-1. This is because n is the number of elements in the array, and because arrays start at 0 rather than 1, means n is 1 greater than the length of the array.

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

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

To prove that the function is correct, we must use induction.

Base Cases:

- If n = 0, then l ¿ r and the program returns -1.

- If n = 1, then there are two options, either v does not exist in A or v is the only element.

Induction Step: Assume the algorithm is true for 2 <= k < n and that v exists in A. There are 3 possible cases.

- m is the midpoint of the subarray. If A[m] = v then the index position m is returned. Intended behavior.
- If A[m] < v, then v is located later in the array than position m. The function recursively calls the function with a new lower bound of m+1. Then v will be searched for in the subarray A[m+1 ... r].
- If A[m] > v, then v is located earlier in the array than position m. The function recursively calls the function with a new upper bound of m-1. Then v will be searched for in the subarray A[l ... m-1].

(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 tri-nary search, which would divide the remaining array A into thirds at each step, would be even more efficient. Explain who is correct and why.*

SOLUTION: Because binary search is similar to a binary search tree, it can be shown that the process is $\Theta(log_2(n))$. It then can be shown that binary search would be $\Theta(log_2(n))$ and tri-nary search would be $\Theta(log_3(n))$. Using the change of base formula, we can rewrite these orders as $\frac{ln(n)}{ln(2)}$ and $\frac{ln(n)}{ln(3)}$. As n $\rightarrow \infty$, the denominators become insignificant so both functions would be $\Theta(logn)$, so both functions are equally as efficient and neither Goyle or Crabbe is correct.