CSCI 3104 Spring 2017 Problem Set 1
Mohamed Al-Rasbi (07/25)

---

1. *For each of the following claims, determine whether they are true or false. Justify your determination (show your work). If the claim is false, state the correct asymptotic relationship as O, Θ, or Ω.*

   (a) $n + 1 = O(n^4)$

   True.
   $\lim_{n\to\infty} \frac{n+1}{n^4} = \lim_{n\to\infty} \frac{1}{4n^3} = 0$

   (b) $2^{2n} = O(2^n)$

   False. The right answer is $2^{2n} = \Omega(2^n)$
   $\lim_{n\to\infty} \frac{2^{2n}}{2^n} = \lim_{n\to\infty} 2^{2n-n} = \lim_{n\to\infty} 2^n = \infty$

   (c) $2^n = \Theta(2^{n+7})$

   True.
   $\lim_{n\to\infty} \frac{2^n}{2^{n+7}} = \lim_{n\to\infty} 2^{n-n-7} = 2^{-7} = c$

   (d) $1 = O(\frac{1}{n})$

   False. The right answer is $1 = \Omega(\frac{1}{n})$
   $\lim_{n\to\infty} \frac{1}{\frac{1}{n}} = \lim_{n\to\infty} n = \infty$

   (e) $\ln^2 n = \Theta(\log_2^2 n)$

   True.

$$\lim_{n\to\infty} \frac{\ln^2 n}{\log_2^2 n} = \lim_{n\to\infty} \frac{\frac{\log^2 n}{\log^2 e}}{\frac{\log^2 n}{log^2 2}} = \frac{\log^2 2}{\log^2 e} = c$$

(f) $n^2 + 2n - 4 = \Omega(n^2)$

False. The right answer is $\Theta(n^2)$

$$\lim_{n\to\infty} \frac{n^2 + 2n - 4}{n^2} = \lim_{n\to\infty} \frac{2n + 2}{2n} = \lim_{n\to\infty} \frac{2}{2} = 1 = c$$

(g) $3^{3n} = \Theta(9^n)$

False. The right answer is $3^{3n} = \Omega(9^n)$

$$\lim_{n\to\infty} \frac{3^{3n}}{3^{2n}} = \lim_{n\to\infty} 3^{3n-2n} = \lim_{n\to\infty} 3^n = \infty$$

(h) $2^{n+1} = \Theta(2^{n \log_2 n})$

False. The right answer is $2^{n+1} = O(2^{n \log_2 n})$

$$\lim_{n\to\infty} \frac{2^{n+1}}{2^{n \log_2 n}} = \lim_{n\to\infty} 2^{n+1-n \log_2 n} = 0$$

Since $n \log_2 n > n$, then the limit equals zero.

(i) $\sqrt{n} = O(\log_2 n)$

False. The right answer is $\sqrt{n} = \Omega(\log_2 n)$

$$\lim_{n\to\infty} \frac{n^{1/2}}{\log_2 n} = \lim_{n\to\infty} \frac{\frac{1}{2\sqrt{n}}}{\frac{1}{n \ln 2}} = \lim_{n\to\infty} \frac{n \ln 2}{2\sqrt{n}} = \infty$$

(j) $10^{100} = \Theta(1)$

True.

$$\lim_{n\to\infty} \frac{10^{100}}{1} = 10^{100} = c$$

2. *Professor Dumbledore needs your help optimizing the Hogwarts budget. You'll be given an array A of exchange rates for muggle money and wizard coins, expressed at integers. Your task is help Dumbledore maximize the payoff by buying at some time i and selling at a future time $j > i$, such that both $A[j] > A[i]$ and the corresponding difference of $A[j] - A[i]$ is as large as possible. For example, let A = [8, 9, 3, 4, 14, 12, 15, 19, 7, 8, 12, 11]. If we buy stock at time i = 2 with A[i] = 3 and sell at time j = 7 with A[j] = 19, Hogwarts gets in income of 19 - 3 = 16 coins.*

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

```
0      makeWizardMoney(A) :
1        maxCoinsSoFar = 0
2        for i = 0 to length(A)-1 {
3           for j = i+1 to length(A) {
4           coins = A[j] - A[i]
5           if (coins > maxCoinsSoFar) { maxCoinsSoFar = coins }
6        }}
7        return maxCoinsSoFar
```

   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 run time complexity:
   $makeWizardMoney(n) = \sum_{i=0}^{n-1}(n-i) = \frac{n(n-1)}{2} = \Theta(n^2)$

   (b) *Explain (1-2 sentences) under what conditions on the contents of A the $makeWizardMoney$ algorithm will return 0 coins.*

   If the array is empty, then the loop will be skipped and the algorithm will return 0.
   Also, if the elements of the array A have the same value, then the algorithm will return 0.

   (c) *Dumbledore knows you know that makeWizardMoney is wildly inefficient. With a wink, he suggests writing a function to make a new array M of size n such that $M[i] = a_{0 \le j \le i} A[j]$*
   That is, M[i] gives the minimum value in the subarray of A[0..i]. What is the

running time complexity of the pseudocode to create the array M? Write your answer as a Θ bound in terms of n

```
createArrayM(A):
    n = length(A)
    M = fill(0, n) \\fill M with 0's
    currentMin = A[0]
    for i = 0 to length(A)-1 {
        currentMin = min(A[i], currentMin) \\compare
        M[i] = currentMin
    }
```

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

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

```
makeWizardMoney(A, M) :
    maxCoinsSoFar = 0
       for i = 0 to length(A)-1 {
            coins = A[i] - M[i]
            if (coins > maxCoinsSoFar) { maxCoinsSoFar = coins }
     }
     return maxCoinsSoFar
```

(e) *Give Dumbledore what he wants: rewrite the original algorithm in a way that combine parts (2b)-(2d) to avoid creating a new array M*

```
makeWizardMoney(A) :
    if(A == []) { \\empty
        return 0 }
    else {
        maxCoinsSoFar = 0
        minCoin = A[0]
        for i = 1 to length(A)-1 {
            if(A[i] < minCoin)
                minCoin = A[i]
            currentMax = A[i] - minCoin
```

```
            if(currentMax > maxCoinsSoFar)
                maxCoinsSoFar = currentMax
        }
    }
    return maxCoinsSoFar
```

The run time complexity for this algorithm is $\Theta(n)$

3. *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.*

```
linSearch(A, v):
    for i = 0 to length(A)-1 {
         if(v == A[i])
             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.*

   Using loop invariant:
   Initialization : Before the the first iteration, the loop invariant holds because $A[j]$ for $j < i (i = 0)$ is an empty set and thus cannot contain v.
   Maintenance : For every iteration, if $v = A[i]$, then it is impossible that $v = A[j]$ for $j < i$ because if $v = A[j]$ it will return j and exit the loop before reaching i. So, the loop invariant still hold.
   Termination : The loop terminates in two cases:
   (i) After $i < length(A) - 1$ iterations. This means the if condition is true and the returned value is i. Thus, $v = A[i]$.
   (ii) Exceeds $length(A) - 1$ and exit the loop. This means that the if condition is false, thus $v \neq A[j]$ and the returned value is NIL.
   Hence, $linSearch(A, v)$ is correct.

4. *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.*

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

Goyle's code is incorrect. There are many bugs:
1) In line 1, l should equal 0 not 1, since the starting index of any array is 0.
2) In line 1, n is not defined. It is the index of the last element. It can be defined as length(A)-1.
3) In line 5, $\leq$ should be $=$, since l is always less than r. This line means the value v is not found in the array.
4) The algorithm doesn't check the first and last element. If $v = A[0]$ or $v = A[length(A) - 1]$, -1 will be returned. We need to add if statements to check that.
A correct version is as follows:

```
0    bSearch(A, v) {
1        return binarySearch(A, 0, length(A)-1, v)
2    }
3
4    binarySearch(A, l, r, v) {
5        if l == r then return -1
6        if A[l] == v then return l
```

```
7        if A[r] == r then return r
8        p = floor( (l + r)/2 )
9        if A[p] == v then return p
10        if A[p] < v then
11           return binarySearch(A, p+1, r, v)
12       else return binarySearch(A, l, p-1, v)
```

We can prove this version of the algorithm is correct by showing that it fulfills all three requirements of a loop invariant:
Initialization : Every time binraySearch is called, v is somewhere in A[l...r] or doesn not exist in A.
Maintenance : On every call of binarySearch, v is either the item under consideration, to the left of the item under consideration, or to the right of the item under consideration.
First case: v is returned.
Second case: binarySearch is called on the portion of A to the left of p.
Third case: binarySearch is called on the portion of A to the right of p.
We can see that the initialization case holds true.
Termination : The function either terminates from finding v, or from exhausting all the elements of A. When l = r the portion of the array being searched is only the item at p, so the function will return -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 four-nary search, which would divide the remaining array A into fourths at each step, would be way more efficient. Explain who is correct and why.*

Goyle is correct. Binary search is bounded by $O(log_2 n)$. The base 2 is given because of how binary search divides whatever it is searching by two at each step, which means four-nary would be bounded by $O(log_4 n)$.
The base of a logarithm has no effect in its asymptotic behavior, which means four-nary search is asymptotically the same as binary search.

5. *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 if A_1, B_2 has the same first element. For example, $A_1 = [1, 2, 3, 4, 5]$ and $B_1 = [1, 3, 9, 20, 21]$
The worst case is if A_2, B_2 have no common element. For example, $A_2 = [10, 20, 30, 40, 50]$ and $B_2 = [11, 12, 13, 14, 15]$

(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.*

```
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
```

```
    else if(A[i] < B[j])
        i = i + 1
    else if(A[i] > B[j])
        j = j + 1
}

return FALSE
```