CS 517 Midterm 2

(Tripp) Milton Lamb

2025 April, 13th

A#:25002371

Disclaimer: I often think more naturally in psuedocode/code than algorithm so I've included the psuedocode for many problems even though it isn't required. I hope that is alright.

Problem 1

Question 1

Design an algorithm to find all the common elements in two sorted lists of numbers. For example, for the lists [2, 5, 5, 5] and [2, 2, 3, 5, 5, 7] the output should be 2, 5, 5 .What is the maximum number of comparisons your algorithm makes if the lengths of the two given lists are m and n, respectively? [15 Points]

Algorithm

1. record the length of both list arguments ($list_a,\ list_b$) as n_a and n_b
2. declare and set index integer variables i_a and i_b to 0
3. declare an empty $list_{result}$
4. loop while $i_a < n_a$ and $i_b < n_b$, and perform the indented steps below repeated until this condition is met

   a. if the value of $list_a[i_a]$ equals $list_b[i_b]$ then add the integer $list_a[i_a]$ to $list_{result}$ and increment both i_a and i_b by 1 and repeat to step 4.a. else go to 4.b.

   b. else if the value of $list_a[i_a]$ is less than $list_b[i_b]$ then increment i_a by 1 and go to step 4.a. else go to step 4.c

   c. else if the value of $list_a[i_a]$ is greater than $list_b[i_b]$ then increment i_b by 1. Go to step 4.a

Answers

The maximum number of comparisons if list lengths are recorded m and n is $m * n$.

Pseudocode

```
#for zero indexed sorted lists
algorithm find_common_elements(list[int]: list_a:, list[int]: list_b) return(list[int]:result)

    int: i_a = 0
    int: i_b = 0
    int: n_a = length(list_a)
    int: n_b = length(list_b)

    result = []

    while (i_a < n_a and i_b < n_b)

        int: a = list_a[i_a]
        int: b = list_b[i_b]

        if a == b
            result.push(a)
            i_a += 1
            i_b += 1
```

```
        else if a < b
            i_a += 1
        else
            i_b += 1
        end if

    end while

    return

end algorithm
```

Question 2

Design an algorithm for computing ⌊ √ n⌋ for any positive integer n. Besides assignment and comparison, your algorithm may only use the four basic arithmetical operations. [10 Points]

Algorithm

1. set $result$ to $\lfloor n/2+1 \rfloor$
2. set $delta$ to $\lfloor result/2 \rfloor$
3. loop forever until explicitly exited by the sub steps
   a. if $result^2 <= n$ and $(result+1)^2 > n$ exit the algorithm $result$ is the answer. otherwise proceed to 3b
   b. if $result^2 <= n$ and $(result+1)^2 <= n$ the increment $result$ by $delta$ and proceed to 3d. otherwise proceed to 3c
   c. if $result^2 > n$ decrement $result$ by $delta$. proceed to 3d
   d. set $delta$ equal to the max value of either 1 or $\lfloor delta/2 \rfloor$. proceed to 3a

Pseudocode

```
# for n > 0
# integer division truncates remainder
algorithm square_root(n:int) return(real:result)

    int: result = n / 2 + 1
    int: delta = result / 2

    while True:

        int: r2 = result**2

        if r2 <= n:
            if (result+1)**2 > n:
                return
            else:
                result += delta
        else:
            result -= delta

        delta = max(1,delta/2)

    end while

    return
```

Problem 2

Question 1

Write pseudocode for an algorithm for finding real roots of equation ax**2 + bx + c = 0 for arbitrary real coefficients a, b, and c. (You may assume the availability of the square root function sqrt(x)). [15 Points]

Pseudocode

```
# implementation of the quadratic formula roots = (-b +- sqrt(b**2 - 4ac)/2a
# an empty list means no real roots exist
# a single value list indicates a double root
algorithm solve_quadratic(real:a, real:b, real:c) return(list[real]:roots)

    roots = []

    if a == 0 # dividing by zero is undefined
        return

    real: discriminant = b**2 - 4*a*c

    if discriminant < 0 # only real solutions are allowed
        return

    real: d = 2*a
    real: z = sqrt(discriminant)

    root_1 = (-b + z)/d

    roots.push(root_1)

    if discriminant == 0 # roots are equal
        return

    root_2 = (-b - z)/d
    roots.push(root_2)

    return

```

Question 2

Consider the following algorithm for finding the distance between the two closest elements in an array of numbers.

```
ALGORITHM MinDistance(A[0.. . . . ..n − 1])
// Input : Array A[0.. . . . ..n − 1] of numbers
// Output : Minimum distance between two of its elements

dmin = ∞
for i = 0 to n − 1 do
    for j = 0 to n − 1 do
        if i ≠ j and !|A[i] − A[j]| < dmin
        dmin = |A[i] − A[j]|
return dmin
```

Make as many improvements as you can in this algorithmic solution to the problem. If you need to, you may change the algorithm altogether; if not, improve the implementation given. [10 Points]

Pseudocode

Starts off at $O(n^2)$ time complexity

Take 1

```
ALGORITHM MinDistance(A[0.. . . . ..n − 1])
// Input : Array A[0.. . . . ..n − 1] of numbers
// Output : Minimum distance between two of its elements

dmin = ∞
for i = 0 to n − 1 do
    for j = (i+1) to n − 1 do
        if |A[i] − A[j]| < dmin
        dmin = |A[i] − A[j]|
return dmin
```

Cuts basic operations in about half and removes an unnecessary comparison operation. Still at $O(n^2)$ time complexity

Take 2

```
ALGORITHM MinDistance(A[0.. . . . ..n − 1])
// Input : Array A[0.. . . . ..n − 1] of numbers
// Output : Minimum distance between two of its elements

B = sort(A) #quick sort and merge sort both have O(n * log(n)) time complexity
dmin = |B[1] - B[0]|
for i = 2 to n − 1 do
    m = |B[i] - B[i-1]|
    if m < dmin
        dmin = m
return dmin
```

By using an $O(n * log(n))$ time complexity sorting algorithm we can parse the min distance from a single pass of the list of numbers which is O(n) time complexity. This means the algorithm's time complexity is $O(n * log(n)) + O(n)$ which simplifies to $O(n * log(n))$. This reduces the overall time complexity. This also removed a single assignment but that doesn't matter.

Take 3

```
ALGORITHM MinDistance(A[0.. . . . ..n − 1])
// Input : Array A[0.. . . . ..n − 1] of numbers
// Output : Minimum distance between two of its elements

if n < 2
    return 0 #not enough numbers in the list to have a difference

B = sort(A) #quick sort and merge sort both have O(n * log(n)) time complexity
dmin = |B[1] - B[0]|
for i = 2 to n − 1 do
    m = |B[i] - B[i-1]|
    if m < dmin
        dmin = m
        if dmin == 0
            return 0
return dmin
```

Added validation checking. If you reach a `dmin` of 0 you can early terminate.

Extra

If we happened to need to make this call many times, it would make more sense to presort the list, and then call the `MinDistance` algorithm at $O(n)$ time complexity, but in that case you should just store the result instead of calling multiple times

There are probably some edge cases where you shave off some time if you start with a long list and insert or delete a small number of new indices. In this case it might make more sense to presort the list upon addition of any new values to the list. This can be done in $O(n)$ time complexity whether in a sorted array list or sorted linked list.

- Array list is $O(log(n))$ to binary search and $O(n)$ to shift.
- Linked list is $O(n)$ to search and $O(1)$ to insert.

If it is likely there may be duplicate numbers it might be worth checking that first because that would make it probable that time complexity would be $O(n)$

But these would probably be fairly situationally specific. So Take 3. Is probably the best place to leave it.

If you wanted to you could modify merge sort or quick to to also check for duplicates and distance while sorting and you could do everything in a single pass, but it wouldn't reduce the overall time complexity and is unlikely to be much if at all faster due to the extra comparisons in the sort algorithm.

Problem 3

Question 1

Write an algorithm to find the “magic index” as defined below in a sorted array A of distinct integers.

Given a sorted array A, we say that index j is the magic index if A[j] = j. For this problem, assume that the arrays are “0” indexed, that is the array elements start at A[0] and go up to A[n − 1]. Clearly write the algorithm and provide a pseudo code for your algorithm. [5 + 5 = 10 Points]

Notes

I'm not sure this algorithm can be written as stated unless I assume that returning any magic index is allowed. As stated the algorithm requests **the** magic index implying a single value; however, without an additional constraint past distinct and sorted you can easily have multiple magic indices (for example [-10, -2, 0, 3, 4, 5, 10] zero indexed contains distinct and sorted integers, but contains 3 magic indices.

For my algorithm I'm going to assume returning any magic index is valid. My algorithm should have worst case $O(log(n))$ time complexity.

Finding all magic indices would require in worst case $O(n)$ time complexity, but allows for early termination as well once the element value exceeds the element index. You would still use binary search to find a magic index, but once found you would have to search in both directions until no more magic indices were found.

Algorithm

1. record the length of the given list a as n_a
2. set i_{left} index to 0
3. set i_{right} index to $n_a - 1$
4. loop the following steps while $i_{left} <= i_{right}$
   a. set i_{mid} equal to the average of i_{left} and i_{right}.
   b. if $a[i_{mid}] == i_{mid}$ return i_{mid} as magic index. exit algorithm. otherwise proceed to step 4b
   c. if $a[i_{mid}] > i_{mid}$ set i_{right} to $i_{mid} - 1$ and proceed to 4a. otherwise proceed to 4c
   d. if $a[i_{mid}] < i_{mid}$ set i_{left} to $i_{mid} + 1$. proceed to 4a
5. If the loop conditional fails before returning a magic index return -1. no magic index was found.

Pseudocode

```
# return any magic index for any given list of sorted distinct integers
# a return of a negative number indicates no magic index exists in list
# integer division truncates remainder

algorithm find_magic_index(list[int]: a) return(int:idx)

    int: n_a = length(a)

    int: i_left = 0
    int: i_right = n_a - 1

    while i_left <= i_right

        int: i_mid = (i_left + i_right)/2

        if a[i_mid] == i_mid
```

```
            idx = i_mid
            return
        else if a[i_mid] > i_mid
            i_right = i_mid - 1
        else
            i_left = i_mid + 1
        end if

    end while

    idx = -1
    return

```

Question 2

Consider the definition-based algorithm for adding two n × n matrices. What is its basic operation? How many times is it performed as a function of the matrix order n? As a function of the total number of elements in the input matrices? [7 Points]

Answer

the basic operation is floating point addition. The basic operation is performed n^2 times as a function of matrix order n, but as a function of the total number of elements (in a single matrix i.e. $m = n^2$) its time complexity is linear at $O(m)$. As a function of the total elements it would be $O(p/2)$ where $p = 2n^2$ which also reduces $O(p)$.

Question 3

Answer the same questions for the definition based algorithm for matrix multiplication. [8 Points

Answer

the basic operation is multiplication. the basic operation is performed n^3 times as a function of matrix order n, but as a function of the total number of elements (in a single matrix i.e. $m = n^2$) its time complexity is $O(m^{3/2})$.

Problem 4

Question 1

Prove the following statements using definitions of O, Ω, Θ. Note that this is a question that is on the harder side and hence I will allow proofs that are argumentative but I need them to be intuitively correct. [5+5+5 = 15 Points]

(a) $3n^2 + 7n + 3 \in O(n^2)$

Big O defines a set of functions whose upper bound is within a multiplicative constant. The smaller terms become insignificant as n grows large. So the equation simplifies to $3n^2 \in O(n^2)$. $3n^2$ is of the same order as n^2 and is clearly within a multiplicative constant of n^2 since 3 is the multiplicative constant. Thus the above statement is true.

(b) $n^2 + n + 9 \notin O(n)$

Big O defines a set of functions whose upper bound is within a multiplicative constant. The smaller terms become insignificant as n grows large. So the equation simplifies to n^2. n^2 is of a higher order than n which means it is not upwardly bound by n therefore. $n^2 + n + 9$ is not in the set $O(n)$. Thus the statement above is true.

(c) $25n^3 - n^2 \in \Theta(n^3)$

Big Θ defines a set of functions who are tightly bound within a multiplicative constant. meaning it must show the same order of growth. Smaller terms become insignificant as n grows sufficiently large meaning the function simplifies to $25n^3 \in \Theta(n^3)$. $25n^3$ is clearly within a multiplicative constant of n^3 since 25 is the multiplicative constant. Thus the above statement is true.

Question 2

Prove or disprove the following statements. [5+5 = 10 Points]

(a) $f \in O(g) \rightarrow f \in \Theta(g)$

Big O defines an upward bound while Big Θ defines a tight bound. In plain English the above statement says if f upwardly bound by the order of g (within a multiplicative constant) then that implies f is also of the same order of g (within a multiplicative constant). $O(g)$ is less restrictive than $\Theta(g)$ so intuitively this statement cannot be correct. But further we can liken that statement to the statement if $x <\approx y$ then $x \approx y$ which is clearly not always true as x could be any number less that y. Similarly f may be of any order less than the order of g, and that does not mean f is of the same order as g. The original statement is false.

(b) $n^{0.01} \in O((log(n))^2)$

The above statement must be true if $n^{0.01}$ is upwardly bound $log(n)^2$ within a multiplicative constant. Or the math equation $\lim_{n\to\infty} n^{0.01} <= \lim_{n\to\infty}[c * log(n)^2]$ must be true. Note: $\lim_{n\to\infty}$ is left off of the below equations to aide in readability. Their presence is implied.

$$\begin{aligned}
n^{0.01} &<= c * log(n)^2 \\
ln(n^{0.01}) &<= ln(c) + ln(log(n)^2) && \text{(take natural log)} \\
0.01 * ln(n) &<= ln(c) + 2 * ln(log(n)) && \text{(logarithm power rule)} \\
0.01 * ln(n) &<= 2 * ln(log(n)) && \text{(ln(c) becomes insignificant as n}\rightarrow\infty\text{)} \\
ln(n) &<= ln(log(n)) && \text{(multiplicative constants can be ignored)} \\
n &<= log(n) && \text{(exponentiation with base e)}
\end{aligned}$$

the growth rate of n is not less than or equal to the growth rate of log(n) or $\lim_{n\to\infty} n \nleq \lim_{n\to\infty} log(n)$ therefore the original statement is false.