CSci 4041
Homework 1 Solutions

1) **Bubble Sort Problem**

(Tips - Write and run the code in your favorite language for a better understanding)

a) **i) Loop invariant** : In the for loop of lines 2-4, the starting point of each iteration consists of

a) The subarray A[j...n] that consists of actual elements that were originally in A[j...n] before entering the inner loop in different order and

b) A[j] is the smallest of those elements.

ii) Proof:

Initialization: It holds trivially, because A[j...n] = A[n] i.e. consists of only one element and it is the last element of A when execution starts the inner loop.

Maintenance: On each step, we replace A[j] with A[j−1] if it is smaller. Because we're only adding the previous element and possibly swapping two values (a) holds. Since A[j−1] becomes the smallest of A[j] and A[j−1] and the loop invariant states that A[j] is the smallest one in A[j...n], we know that (b holds.

Termination: After the loop terminates, we will get j=i. This implies that A[i] is the smallest element of the subarray A[i...n] and array contains the original elements that in a sorted manner.

b) **Loop invariant:** At the start of each iteration, A[1..i−1] consists of sorted elements, all of which are smaller or equal to the ones in A[i...n].

Initialization: Initially i = 1 so A[1 ... i-1] = A[0], which is an empty array. This is clearly consistent with the loop invariant.

Maintenance: The invariant of the inner loop tells us that on each iteration, A[i] becomes the smallest element of A[i...n] while the rest get shuffled. Note that the loop invariant is true till A[1...i-1] and we need that for maintenance. This implies that at the end of the loop:

$$A[i]<A[k], \text{ for } i<k$$

Termination: The loop terminates with i=n, where n is the length of the array. Substituting the n for i in the invariant, we have that the subarray A[1..n] consists of the original elements, but in sorted order with A[n] > A[n-1] in the final iteration. This is the entire array which is sorted.

2. Growth Function Problems

a. f(n) = O(g(n)) implies g(n) = O(f(n)).

Answer: False

Proof: Let f(n) = 1 and g(n) = n for all natural numbers. Then, there exist positive constants c and n0 such that $0 \leq f(n) \leq c * g(n)$ for all n >= n0 from definition. However, suppose g(n) = O(f(n)), then there are a natural number n0 and constant c > 0 such that

$n = g(n) \leq c * f(n) = c$, for all n >= n0. This means we need to get a c, n0 such that $n \geq n0$ and $n \leq c$. This is not possible for n > c. This is a contradiction.

b. f(n) = O(g(n)) implies log(f(n)) = O(log(g(n))), where log g(n) ≥ 1 and f(n) ≥ 1 for all sufficiently large n.

Answer: True

Proof:

f(n) = O(g(n)) denotes

$0 \leq f(n) \leq c * g(n)$ for some c > 0 and all n > n0

=> $\log(f(n)) \leq \log(c * g(n))$

=> $\log(f(n)) \leq logc + \log(g(n)) \leq c2 * \log(g(n))$ for some constant c2 > 1

Therefore $\log(f(n)) = O(\log(g(n))$

This only holds if $(\log(g(n))$ is not approaching 0 as n grows.

c. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$

Answer: False

Proof: Let f(n) = 2n and g(n) = n for all natural numbers n, then $f(n) = 2 * \ g(n))$ for all n. There exist positive constants c and n0 such that 0 <= f(n) <= cg(n) for all n >= n0 from definition. However, suppose $2^{f(n)} = O(2^{g(n)})$, then there are natural number n0 and a constant c > 0 such that $4^n = 2^{2n} = 2^{f(n)} = c * \ 2^{g(n)} = c * \ 2^n$

i.e. $(4/2)^n < c$ for all n >= n0. This means we need to get a c, n0 such that $n \geq \ n0$ and $n \leq c$. This is a contradiction.

d. $f(n) = O(f(n)^2)$

Answer: False

Proof: Let f(n) = 1/n for all natural numbers n. Suppose that $f(n) = O(f(n)^2)$

Then there are a natural number n0 and c > 0 such that

$\frac{1}{n} = f(n) = c * f(n)^2 = c * \frac{1}{n^2}$ This results in $n = \frac{n^2}{n} \leq c$ for all n >= n0. This means we need to get a c, n0 such that n >= n0 and n <= c. This is not possible for n > c. This is a contradiction.

e. $f(n) = O(g(n))$ implies $f(n) = \Theta(g(n))$

Answer: True

Proof: There exist positive constants c and n0 such that $0 \leq f(n) \leq c * g(n)$ for all n >= n0 from definition. Assuming $c1 = \frac{1}{c} > 0$ then $g(n) \geq c1 * f(n)$ for all n>=n0.

So, g(n) = Ω(f(n)).

f. $f(n) = \Theta(f\left(\frac{n}{2}\right))$

Answer: False

Proof: Let f (n) = 2^n for all natural numbers n. Suppose $f(n) = \Theta(f\left(\frac{n}{2}\right))$. Then there are a natural number n0 and c2 > 0 such that $2^n = f(n) \leq c2 * f\left(\frac{n}{2}\right) = c2 * 2^{\frac{n}{2}}$ i.e. $2^{(1/2)^n}$. Therefore equating both sides we get $(\frac{2}{2^{1/2}})^n \leq c2$ for all n >= n0. This means we get a c, n0 such that $n \geq n0$ and $n \leq c$. This is a contradiction.

g. $f(n) + o(f(n) = \Theta\big(f(n)\big)$.

Answer: True

Proof: We know,

$o(f(n) = \{ g(n) : 0 \leq g(n) \leq c * f(n)$, for any c > 0, there exists a n0 > 0 from definition}

Now, At most, f(n) + o(f(n)) can be $f(n) + c * f(n) = (1 + c) * f(n)$

At least, f(n) + o(f(n)) can be $f(n) + 0 = f(n)$

Thus, $0 < f(n) < f(n) + o(f(n) < (1 + c) * f(n) = c2 * f(n)$, $c2 \; being \; some \; constant > 0$

Which is the same as saying $f(n) = \Theta(f\left(\frac{n}{2}\right))$

3)

a)

Using master method, case 3:

a=4

b=2

f(n)=$n^{2.5}$

$n^{log_b a}$= n^2

ϵ=0.1

$a * f(n/b) <= c * f(n)$

$c >= a * f(n/b) / f(n)$

$c >= 4 * n^{2.5} / (2^{2.5} * n^{2.5})$

$c >= 1/\sqrt{2}$

$1/\sqrt{2} <= c < 1$

T(n) = Θ($n^{2.5}$)

=> T(n) = O($n^{2.5}$)

=> T(n) = **Ω**($n^{2.5}$)

b)

Using master method, case 3:

a=1

b=10/7

f(n)=n

$n^{log_b a}$ = 1

ϵ=0.1

$a * f(n/b) <= c * f(n)$

$c >= a * f(n/b) / f(n)$

$c >= 1 * n * 7 / (10 * n)$

$c >= 0.7$

$0.7 <= c < 1$

T(n) = Θ(n)

=> T(n) = O(n)

=> T(n) = $\boldsymbol{\Omega}$(n)

c)

Solving recurrence:

$$T(n) = T(n-2) + n^2$$

$$= T(n-4) + (n-2)^2 + n^2$$

$$= k + n^2 + (n-2)^2 + (n-4)^2 + \ldots (n/2) times$$

Using equation A.3 from the book, we have sum of squares of the form $n(n+1)(2n+1)/6$

T(n) = Θ(n^3)

=> T(n) = O(n^3)

=> T(n) = $\boldsymbol{\Omega}$(n^3)

d)

Solving recurrence:

$$T(n) = T(n-1) + 1/n$$

$$= T(n-2) + 1/n + 1/(n-1)$$

$$= k + 1/n + 1/(n-1) + 1/(n-2) + \ldots (n-1) times$$

It's a Harmonic Series

Using equation A.7 from the book

T(n) = Θ($\log n$)

=> T(n) = O($\log n$)

=> T(n) = $\boldsymbol{\Omega}$($\log n$)

e)

Solving recurrence:

$T(n) = T(n-1) + \log n$

$= T(n-2) + \log n + \log(n-1)$

$= k + \log n + \log(n-1) + \log(n-2) ... (n-1) times$

$= k + \log[n(n-1)(n-2) ... (n-1) times]$

$= k + \log(n!)$

Using equation 3.19 from book.

T(n) = Θ($n \log n$)

=> T(n) = O($n \log n$)

=> T(n) = **Ω**($n \log n$)

f)

Solving recurrence:

$T(n) = T(n-1) + 1/\log n$

$= T(n-2) + 1/\log(n) + 1/\log(n-1)$

$= k + 1/\log(n) + 1/\log(n-1) + 1/\log(n-2) ... (n-1) times$

$$= k + \sum_{i=\log 2}^{\log n} \frac{1}{i}$$

T(n) = Θ($\log \log(n)$)

=> T(n) = O($\log \log(n)$)

=> T(n) = **Ω**($\log \log(n)$)

g)

Cannot use master method.

Building recursion tree we see that there are logn levels.

Cost of one node at ith level: $(n/2^i)/(\log n - i)$

2^i nodes at ith level

Total cost at ith level = $n/(\log n - i)$

Overall cost =

$$\sum_{i=0}^{\log n - 1} \frac{n}{\log n - i}$$

$$= n \sum_{i=1}^{\log n} \frac{1}{i}$$

T(n) = Θ($n \log \log n$)

We have harmonic series for the summation above.

=> T(n) = O($n \log \log n$)

=> T(n) = **Ω**($n \log \log n$)

h)

$$T(n) = \sqrt{n}T(\sqrt{n}) + n$$

$$T(n) = \sqrt{n}(n^{1/4} * T(n^{1/4}) + \sqrt{n}) + n$$

$$= n^{1/2+1/4} * T(n^{1/4}) + n^{1/2+1/2} + n$$

$$= n^{1/2+1/4+1/8+...} * T(n^{1/8}) + n + n + ...log(n)times$$

We have geometric series in the power of n. See equation A.5 in the book.

$$= n + n * \log(n)$$

Replacing value of m,

T(n) = O($n \log(n)$)

=> T(n) = O($n \log(n)$)

=> T(n) = **Ω**($n \log(n)$)

4)

Algorithm keeps track of two variables:

1. maxSum: The maximum subarray in A[1...j], i.e. when we have looked at first j elements. This maximum subarray need not end in j.
2. currentMaxSum: The maximum subarray in A[1...j] ending at j. Start index of this maximum subarray may not be equal to 1.

We start traversing the array from its start and keep track of the "current sum" (currentMaxSum) and corresponding indices. currentMaxSum is the sum of all elements with indices between i and j. Suppose at a particular instant, currentMaxSum has value C and it lies between indices i and j. Now, when we move from index j to j+1, we are looking at a bigger subarray lying between indices i and j+1. If adding the (j+1)th element to C gives us a sum bigger than the maxSum corresponding to subarray 1 to j, we update the maxSum, else we just update C and move on. We update C even if it is less than maxSum in an anticipation that in future C might overcome maxSum.

In any case, value of current sum becomes negative, we make the currentMaxSum as zero, which is equivalent to a sum when we don't choose any element from the array.

Pseudocode:

maximum_subarray(array[1..n])

begin

// Initialization

maxSum = -INFINITY

maxStartIndex = 0

```
    maxEndIndex = 0

    currentMaxSum = 0
    currentStartIndex = 1

    for currentEndIndex = 1 to n do
     // Update currentMaxSum with the new element
    currentMaxSum = currentMaxSum + array[currentEndIndex]

    // Update maxSum if we have found a bigger value
    if currentMaxSum > maxSum then
    maxSum = currentMaxSum
    maxStartIndex = currentStartIndex
    maxEndIndex = currentEndIndex
    endif

    // If currentMaxSum becomes negative,
    // start afresh from the next element
    if currentMaxSum < 0 then
    currentMaxSum = 0
    currentStartIndex = currentEndIndex + 1
    endif
    endfor

    return (maxSum, maxStartIndex, maxEndIndex)
end
```

This algorithm is commonly known as Kadane's algorithm.