Jahongir Hayitov – CS-01

Homework-1, Theoritical Part

3.1 Asymptotics (32 points)

1. Prove or disprove the following statements. You must provide a formal proof. You may use the definitions of the asymptotic notations as well as their properties and properties of common functions, as long as you properly reference them (e.g. by specifying the exact place in Cormen where this property is introduced)!

(a) $n^3 \log n = \Omega(3n \log n)$

(b) $n^{9/2} + n^4 \log n + n^2 = O(n^4 \log n)$

(c) $6^{n+1} + 6(n + 1)! + 24n^4\ 2 = O(n!)$

(d) There exists a constant $\varepsilon > 0$ such that $n / \log n = O(n^{1-\varepsilon})$.

Solution:

a) Proof: according to Cormen 3.2

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$$0 \leq c*g(n) \leq f(n) \text{ for all } n \geq n_0 \}$$

Now let's show that $n^3 \log n = \Omega(3n \log n)$. We need to find positive constants c and n_0 such that $n^3 \log n \geq c*(3n \log n)$, for all $n \geq n_0$. As before, we divide both sides by $n*(\log n)$ giving $n^2 \geq c*3$. Regardless of what value we choose for the constant c, this inequality **holds** for any value of $n^2 \geq c*3$. Q.E.D.

b) Disproof: according to Cormen 3.2

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c*g(n) \text{ for all } n \geq n_0 \}$$

Now let's show that $n^{9/2} + n^4 \log n + n^2 = O(n^4 \log n)$. We need to find positive constants c and n_0 such that $n^{9/2} + n^4 \log n + n^2 \leq c*(n^4 \log n)$, for all $n \geq n_0$. As before, we divide both sides by $n^4 (\log n)$ giving $(n^{1/2} * \log^{-1} n) + 1 + n^{-2} * \log^{-1} n \leq c$. Because of faster increase of square root function compared to logarithm function, regardless of what value we choose for the constant c, this inequality **does not hold** for any value of $(n^{1/2} * \log^{-1} n) + 1 + n^{-2} * \log^{-1} n \leq c$ as the same as $\text{sqrt}(n) / \log n \leq c$ Q.E.D.

c) Disproof: according to Cormen 3.2

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c*g(n) \text{ for all } n \geq n_0 \}$$

Now let's show that $6^{n+1} + 6(n + 1)! + 24n^4\ 2 = O(n!)$. We need to find positive constants c and n_0 such that $6^{n+1} + 6(n + 1)! + 24n^4\ 2 \leq c*(n!)$, for all $n \geq n_0$. As before, we divide both sides by n! giving $6^{n+1}/\ n! + 6(n + 1)! / n! + 24n^4\ 2 / n! \leq c$. Cormen 3.3 We already now that for extremely greater values of n, the factorial function grows faster than exponential. $6^{n+1}/\ n! + 6(\mathbf{n! * (n+1)}) / \mathbf{n!} + 24n^4\ 2 / n! \leq c$ gives $6^{n+1}/\ n! + 6\mathbf{(n+1)} + 24n^4\ 2 / n! \leq c$, regardless of what value we choose for the constant c, this inequality **does not hold** for any value of $6^{n+1}/\ n! + 6(n+1) + 24n^4\ 2 / n! \leq c$ as the same as $\text{sqrt}(n) / n \leq c$ Q.E.D.

d) Disproof: according to Cormen 3.2

$O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that

$$0 \leq f(n) \leq c*g(n) \text{ for all } n \geq n_0 \}$$

Now let's show that there exists a constant $\varepsilon > 0$ such that $n / \log n = O(n^{1-\varepsilon})$. We need to find positive constants c and n_0 such that $n / \log n \leq c*(n^{1-\varepsilon})$, for all $n \geq n_0$. We can write this expression as $n / \log n \leq c*(n / n^{\varepsilon})$. As before, we divide both sides by n giving, $\log^{-1} n \leq c*n^{-\varepsilon}$. In Cormen 3.24 Logarithmic functions grows slower than n^{ε} where $\varepsilon > 0$, but their reciprocal is inverse. Reciprocal of logarithmic function grows faster than reciprocal of n^{ε}. Therefore regardless of what value we choose for the constant c, this inequality **does not hold** for any value of $\log^{-1} n \leq c*n^{-\varepsilon}$ Q.E.D

2. For each of the following recurrences, apply the master theorem yielding a closed form formula. You must specify which case is applied, explicitly check the necessary conditions, and provide final answer using Θ-notation. If the theorem cannot be applied you must provide justification.

(a) T (n) = 2T (n/3) + log n

(b) T (n) = 3T (n/9) + sqrt(n)

(c) T (n) = 4T (n/9) + sqrt(n) log n

(d) T (n) = 5T (n/5) + n / 1000

Solution:

according to Cormen 4.5

*The Master Theorem: T(n) = a * (T(n / b) + f(n)*

where a > 0 and b > 1 are constants. We call f(n) driving function, and we call a recurrence of this general form a master recurrence

1. If there exists a constant $\epsilon > 0$ such that $f(n) = O(n^{\log_b a - \epsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
2. If there exists a constant $k \geq 0$ such that $f(n) = \Theta(n^{\log_b a} \lg^k n)$, then $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
3. If there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$, and if $f(n)$ additionally satisfies the ***regularity condition*** $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

a) T (n) = 2T (n/3) + log n

1) a = 2, b = 3, f(n) = log n a>0, b>1

2) we need to find $\log_b a = \log_3 2 < 1$

. 3) **first condition holds**: there exists a constant $\varepsilon > 0$ | log n = $O(n^{\log_3(2) - \varepsilon})$, therefore the time complexity **$T(n) = \Theta(n^{\log_3(2)})$**

b) T (n) = 3T (n/9) + sqrt(n)

1) a = 3, b = 9, f(n) = $n^{1/2}$ a>0, b>1

2) we need to find $\log_b a = \log_9 3 = 1/2$

. 3) **second condition holds**: there exists a constant k = 0 | $n^{1/2} = \Theta(n^{1/2} * \lg^k n)$, therefore the time complexity **$T(n) = \Theta(n^{1/2} * \lg n)$**

c) T (n) = 4T (n/9) + sqrt(n) log n

1) a = 4, b = 9, $f(n) = n^{1/2} \log n$ a>0, b>1

2) we need to find $\log_b a = \log_9 4 < 1$ and $\log_9 4 > 1/2$

. 3) **second condition holds**: there exists a constant $1<k<2$ | $n^{1/2} \log n = \Theta(n^{\log_9(4)} * \lg^k n)$, therefore the time complexity **$T(n) = \Theta(n^{\log_9(4)} * \lg^{k+1} n)$** where $2<k+1<3$

d) T (n) = 5T (n/5) + n / 1000

1) a = 5, b = 5, f(n) = n / 1000 a>0, b>1

2) we need to find $\log_b a = \log_5 5 = 1$

. 3) **second condition holds**: there exists a constant k=0 | $n / 1000 = \Theta(n * \lg^k n)$, therefore the time complexity **$T(n) = \Theta(n * \lg n)$**

3.2 Segmented List (18 points)

Recall the methods of the List ADT:

```
public interface List<E> {
      int size();
      boolean isEmpty();
      void add(int position, E element);
      E remove (int position);
      E get(int position);
      E set(int position, E element);
}
```

Consider a SegmentedList implementation, that consists of a doubly linked list of arrays (segments) of fixed capacity k:

1. all segments, except the last one, must contain at least floor(k/2) elements;
2. when adding at the end of SegmentedList, the element is added in the last segment, if it is not full; otherwise, a new segment of capacity k is added at the end of the linked list;

3. when adding x at any position, the doubly linked list is scanned from left to right (or right to left) to find the corresponding segment; if the segment is full, it is split into two segments with k/2 elements each, and x is inserted into one of them, depending on the insertion position.
4. when removing an element, only elements in its segment are shifted; if the segment size becomes less than floor(k/2) , then we rebalance:
(a) if it is the last segment, do nothing;
(b) if it is a middle segment s_i and the sum m = size(s_{i-1}) + size(s_i) + size(s_{i+1}) < 2k, then replace these three segments with two segments of size m/2
(c) if it is a middle segment si and the sum m = size(s_{i-1}) + size(s_i) + size(s_{i+1}) ≥ 2k, then replace these three segments with tree segments of size m/3
(d) if it is the first segment, and there exist two segments after, then proceed similarly to previous two cases;
(e) if it is the first segment and only one segment after it exists, move as many elements from the last segment to the first one as possible; if the last segment becomes empty, remove it.
Perform the following analysis for the SegmentedList:
1. Argue that the worst case time complexity of add(i, e) is O(n/k + k).
2. Argue that the worst case time complexity of remove(i) is O(n/k + k).
3. What value of k should be chosen in practice? Why?

Solution:

1) `add(i, e)` - Since in Doubly Linked List, it takes O(1) constant time to go to the first and the last element. The worst case time complexity is when the position of the segment is in the middle. In average the number of all segments is at least `(2*n/k)` Therefore it takes `O(2*n/(k*2)) = O(n/k)` to go to the middle position. We know that elements inside the middle segment array is at least `k/2,` we can consider this array full with k elements **`O(k)`**. To split this array into two segments `k/2` elements each and add them to the list also takes **`O(k).`** Therefore total time complexity is `O(n/k + 2*k) = O(n/k + k)` **Q.E.D**

2) `remove(i)` - Let's again consider our target position is in the middle, because in the worst case it takes **`O(n/k)`** to get this position. We can also consider the number of elements in the middle segment is `k/2` and and its previous and next neighbors with full k elements both. Therefore, when we remove the element from the middle segment array **`O(k/2)`**, it becomes less than `[k/2]` and the sum m is `(k/2-1)+k+k = 2*k+k/2-1` is greater than `2*k`. We choose (c) case: "`if it is a middle segment` s_i `and the sum m = size(`s_{i-1}`) + size(`s_i`) + size(`s_{i+1}`) ≥ 2k, then`

replace these three segments with three segments of size m/3 ". Replacing all elements takes **O(m)**. Therefore total time complexity is O(n/k + k/2 + m) , since m = 2*k+k/2-1, O(n/k + k/2 + 2*k + k/2 – 1) the same as **O(n/k + k)** **Q.E.D**

3) What value of k should be chosen in practice?

In practice, the most efficient value of k – the length of the segment array is nearly equal to the l - length of the List. In this case k*l = n is the maximum probability of the number of elements or capacity of the list. Giving, $k^2 = n$, Therefore for better implementation, we can choose **k = sqrt(n) = √n** in practice.

3.3 Segmented Queue (+1% extra credit)

Consider SegmentedList used as a Queue:

1. we enqueue (offer(e)) elements by adding them to the end of the segmented list;

2. we dequeue (poll()) elements by removing the first element of the segmented list;

3. in an attempt to make remove more efficient, we do not perfom shifts when removing an element in a segment (so there can be a gap on the left of the cells where values are stored); add will create a new segment when it reaches the right end of the last segment, regardless of whether there is empty space to the left;

4. rebalancing part of# remove remains, to keep the invariant that each segment has at least k/2 elements.

Perform amortised analysis for arbitrary sequences of offer(e) and poll() operations applied to an initially empty queue, implemented using SegmentedList in a way described above. Show that amortized cost for any such sequence of length N is O(N) (does not depend on the value of k). You must use the accounting method or the potential method

Solution:

Since we are given empty queue using SegmentList and N operations, we need to fill the queue with using N/2 number of offer(e) operations takes **O(N/2 + N/k)**. Then, N/2 number of poll() operations left. Now, We can make analysis recursively. For case (a): if it is the last segment, do nothing; which takes constant **O(k/2),** then case (e) if it is the first segment and only one segment after it exists, move as many elements from the last segment to the first one as possible; if the last segment becomes empty, remove it. Which takes **O(2k)**. Then we can consider case c as

the worst case (c) if it is a middle segment si and the sum m = size(s_{i-1}) + size(s_i) + size(s_{i+1}) ≥ 2k, then replace these three segments with tree segments of size m/3 it - **O(2*k+k/2-1)** and may occur N/k-2 times, therefore it takes ~ **O(5*N/2).** And finally, it takes O(k/2+1) to make half the front segment of the full queue. Therefore total time complexity is O(N/2 + N/k + k/2 + 2k + 5N/2) = O((6N – 5k)/2 + N/k)) = O(3N) = O(N) Q.E.D