COMP3711 Assignment 1

TANUWIJAYA, Randy Stefan
(20582731)
rstanuwijaya@connect.ust.hk

Department of Physics - HKUST
Department of Computer Science and Engineering - HKUST

April 11, 2021

Problem 1: Proof of Recurrence Simplification

Recall the recurrence:

$$\forall n > 1, \quad T(n) \le \left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + n \quad \text{and} \quad T(1) = 1 \tag{1}$$

(a) Let $c > 0$ be some constant integer, Let $T(n)$ be a function satisfying:

$$\forall n > 2, \quad T(n) \le T\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c \quad \text{and} \quad T(1) = T(2) = 1 \tag{2}$$

(i) Prove using the expansion method that $T(3^k) = O(k)$

$$\begin{aligned}
T(3^k) &\le T(3^{k-1}) + c \quad \le T(3^{k-2}) + 2c \\
&\dots \\
&\le T(3^{k-k}) + kc \quad = T(1) + kc \\
&= 1 + kc \quad \le 1 + kc' \quad (\text{let } c' = c + 1, \text{ then for } k \ge 1) \\
&\le k + kc = O(k) \quad (\text{Q.E.D})
\end{aligned}$$

(ii) Let $S(n)$ be some nondecreasing function of n. You are told that $S(n)$ also satisfies $S(3^k) = O(k)$. Prove that $S(n) = O(\log_3 n)$.

$S(3^k) = O(k)$ implies $\exists c > 0 \land k_0$ s.t for $k > k_0$, $S(3^k) \le ck$. Since $S(n)$ is nondecreasing, let $k_1 = \log_3 n + 1 \ge k_0$, then $T(n) \le T(3^{k_1}) \le ck_1 = c(\log_3 n + 1) \le 2c \log_3 n$. The last inequality requires $n \ge 3$. Thus, for $n > \max(3, 3^{k_0 - 1})$ and $c_1 = 2c$, $S(n) \le c_1 \log_3 n = O(\log_3 n)$

(iii) For all $n \ge 1$, set $R(n) = \max_{1 \le i \le n} T(i)$. Prove that

$$\forall n > 1, \quad R(n) \le R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c \tag{S1.1}$$

Since $R(n) = \max_{1 \le i \le n} T(i)$ we have $T(n) \le R(n)$ for all n. Combining with equation (2), we have:

$$T(i) \le T\left(\left\lfloor \frac{i}{3} \right\rfloor\right) + c \le R\left(\left\lfloor \frac{i}{3} \right\rfloor\right) + c$$

$$R(n) = \max_{1 \le i \le n} T(i) \le \max_{1 \le i \le n} R\left(\left\lfloor \frac{i}{3} \right\rfloor\right) + c \le R\left(\left\lfloor \frac{n}{3} \right\rfloor\right) + c$$

note that the last inequality holds due to the fact that $R(i)$ is a nondecreasing function.

(iv) Using (i), (ii), and (iii), prove that if $T(n)$ satisfies Equation (2) then $T(n) = O(\log n)$

First, note that $\forall n \geq 2, T(n) \leq R(n)$. Then, from (iii) and (i), we know that the upper bound of $T(n)$, which is $R(n)$ is a nondecreasing function and also upperbounded by $R(3^k) = O(k)$. Combining with the result from (ii). we proved that $R(n) = O(\log_3 n)$. Since $\forall n \geq 2, T(n) \leq R(n)$, $T(n) = O(\log_2 n)$. (QED)

(b) If $T(n)$ satisfies Equation (1) then

$$T(2^K) = O(k2^k) \tag{3}$$

(i) Let $S(n)$ be some nondecreasing function of n. You are told that $S(n)$ also satisfies $S(2^k) = O(k2^k)$. Prove that $S(n) = O(n \log_2 n)$.

$S(2^k) = O(k2^k)$ implies $\exists c > 0 \wedge k_0$ s.t for $k > k_0$, $S(2^k) \leq ck2^k$. Since $S(n)$ is nondecreasing, let $k_1 = \log n + 1 \geq k_0$, then $T(n) \leq T(2^{k_1}) \leq ck_1 2^{k_1} = cn(\log n + 1) \leq 2cn \log n$. The last inequality requires $n \geq 2$. Thus, for $n > \max(2, 2^{k_0 - 1})$ and $c_1 = 2c$, $S(n) \leq c_1 n \log n = O(n \log n)$.

(ii) For all $n \geq 1$, set $R(n) = \max_{1 \leq i \leq n} T(i)$. Prove that

$$\forall n > 1, \quad R(n) \leq R\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + R\left(\left\lceil \frac{n}{2} \right\rceil\right) + c$$

Since $R(n) = \max_{1 \leq i \leq n} T(i)$ we have $T(n) \leq R(n)$ for all n. Combining with equation (2), we have:

$$T(i) \leq T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + T\left(\left\lceil \frac{i}{2} \right\rceil\right) + c \leq R\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + R\left(\left\lceil \frac{i}{2} \right\rceil\right) + c$$

$$R(n) = \max_{1 \leq i \leq n} T(i) \leq \max_{1 \leq i \leq n} R\left(\left\lfloor \frac{i}{2} \right\rfloor\right) + R\left(\left\lceil \frac{i}{2} \right\rceil\right) + c \leq R\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + R\left(\left\lceil \frac{n}{2} \right\rceil\right) + c$$

note that the last inequality holds due to the fact that $R(i)$ is a nondecreasing function.

(iii) Using Equation (3), (v), and (vi), prove that $T(n)$ defined by Equation (1) satisfies $T(n) = O(n \log_2 n)$.

First, note that $\forall n \geq 1, T(n) \leq R(n)$. Then, from (iii) and (i), we know that the upper bound of $T(n)$, which is $R(n)$ is a nondecreasing function and also upperbounded by $R(2^k) = O(k2^k)$. Combining with the result from (ii). we proved that $R(n) = O(n \log_2 n)$. Since $\forall n \geq 1, T(n) \leq R(n)$, $T(n) = O(n \log_2 n)$. (QED)

Problem 2: Right-flipped Array Divide and Conquer

Let $A[1 \dots n]$ be an array with n items. A' is *array A right-flipped at k* with $1 \leq k \leq n$ if

$$A'[i] = \begin{cases} A[i] & \text{if } i < k \\ A[n+k-i] & \text{if } i \geq k \end{cases} \tag{4}$$

(a) (i) Documented pseudocode with $O(\log n)$ time to return the value of k

Algorithm 1: Find-k(p, r)

```
if r − p = 1 ;                                   /* cover base case n = 2 */
then
   if A[p] > A[p + 1] then
      return p
   return p + 1
else
   q ← ⌊(p + r)/2⌋ ;                                    /* find midpoint */
   if A[q] > A[q + 1] then                /* if the midpoint is decreasing */
      Find-k(p, q) ;                                /* recurse to left-half */
   else
      Find-k(q, r) ;                  /* otherwise, recurse to right-half */
```

(ii) For input right-flipped array $A[1 \dots n]$ at k and $n \geq 2$. For the base case, $n = 2$, the flipping is either at 1 or 2. If $A[p] < A[p+1]$, then the flipping is at $k = p+1$. On the other hand, if $A[p] > A[p+1]$, then the flipping is at $k = p$.

For general case $n > 2$, we find the flipping by divide and conquer method. First, define the midpoint as $q = \lfloor (p+r)/2 \rfloor$. Then, if the midpoint is decreasing $A[q] > A[q+1]$, we recurse to the left-half of the array (call Find-k(p, q)). Otherwise, if midpoint is increasing $A[q] < A[q+1]$, we recurse to the right-half of the array (call Find-k(q, r)). The algorithm eventually will terminate and return the flipping point k when it reached the base case.

(b) Prove that the algorithm correctly returns the flipping point k.

<u>**Base Cases:**</u> $n = 2$

(i) $\mathbf{A[p] > A[p+1]}$: the algorithm correctly returns $k = p$

(ii) $\mathbf{A[p] < A[p+1]}$: the algorithm correctly returns $k = p+1$

<u>**General Cases:**</u> $n > 2$, assume $I(n')$ is true for all $1 \leq n' < n$ and suppose $r - p + 1 = n$. Let $q = \lfloor (p+r)/2 \rfloor$ as the midpoint of the array in the current recursive call.

(i) $\mathbf{A[q] > A[q+1]}$: Algorithm will call Find-k(p, q), which has size of $n' = \lfloor (r-p)/2 \rfloor + 1 < n$ and $n' \geq 2$. By the induction hypothesis it returns the correct answer.

(ii) $\mathbf{A[q] < A[q+1]}$: Algorithm will call Find-k(q, r), which has size of $n' = \lceil (r-p)/2 \rceil + 1 < n$ and $n' \geq 2$. By the induction hypothesis it returns the correct answer.

Thus, Find-k(p, q) always returns the correct answer and $I(m)$ is true for all $n \geq 2$.

(c) Derive recurrence relation of $T(n)$, the worst case number of comparisons performed by your algorithm on an array of size n, and explain why this recurrence relation implies $T(n) = O(\log n)$.

The recurrence relation of Find-k(p, r) can be expressed as:

$$\forall n \geq 2, T(n) = \begin{cases} T\left(\left\lfloor \dfrac{n+1}{2} \right\rfloor\right) + 1 & \text{if } A[q] > A[q+1] \\ T\left(\left\lceil \dfrac{n+1}{2} \right\rceil\right) + 1 & \text{if } A[q] < A[q+1] \end{cases}, \quad T(2) = 1 \tag{S2.1}$$

The Find-k the reccurence relation follows $T(n) \leq T(n/2) + c$ for $c = 1$. Therefore, the worst case number of comparisons $T(n)$ is $O(\log n)$.

Problem 3: Heavy Element in Array

Let A be an array of n elements. A *heavy element* of A is any element that appears more than $2n/5$ times. Design an $O(n \log n)$ divide-and-conquer algorithm for finding the heavy items in an array.

(a) (i) Documented pseudocode for a procedure *Heavy(i,j)* that returns the set of heavy items in subarray $A[i \ldots j]$

Algorithm 2: Heavy(i, j)

```
if j − i + 1 = 1 then
    return {A[i]} ;                                   /* Base case n=1 */
else if j − i + 1 = 2 then
    return {A[i], A[i + 1]} ;                         /* Base case n=2 */
else
    m ← ⌊(i + j)/2⌋
    S = Heavy(i,m) ∪ Heavy(m+1, j) ;                  /* S is possible heavy elements */
    R = ∅
    forall x ∈ S do
        count = CountHeavy(x, i, j) ;                 /* count the number of x in A[i..j] */
        if count > 2(j − i + 1)/5 then                /* determine if x is Heavy in A[i..j] */
            R ← R ∪ {x}
    end
    return R
```

Algorithm 3: CountHeavy(x, i, j)

```
count ← 0
for k ← i to j do
    if A[k] = x then
        count ← count + 1
end
return count ;                                        /* return the number of x in A[i..j] */
```

(ii) Begin with the fact that the heavy element in A must be the heavy element in either left-half subarray, right-half subarray, or both.

The base case of the algorithm is when $n \in \{1, 2\}$, where all of the element is heavy element.

The general case is when $n > 2$, where we will apply the recursion. First, we identify the midpoint $m = \lfloor (i+j)/2 \rfloor$ of the array $A[i \ldots j]$. Then using the fact we know before, the candidate of the heavy element in A is the union of heavy element in left-half and right-half subarray, and store it as the set S. Then, check for every element of S, whether the count of that particular element in the array $A[i \ldots j]$ exceed 40% by calling CountHeavy(x, i, j). If so, store the element into R. After checking for all the element in S, the algorithm return R, which is the set containing the heavy element in $A[i \ldots j$

The second algorithm, CountHeavy(x, i, j) simply count the number of x in the array $A[i \ldots j]$

(b) Prove the correctness of the algorithm

<u>**Base Cases:**</u> $n \in \{1, 2\}$

(i) $\mathbf{n = 1}$: The algorithm correctly returns $\{A[i]\}$

(ii) $\mathbf{n = 2}$: The algorithm correctly returns $\{A[i], A[i+1]\}$

<u>**General Cases:**</u> $n > 2$, assume $I(n')$ is true for all $1 \leq n' < n$ and suppose $j - i + 1 = n$. Let $m = \lfloor (i+j)/2 \rfloor$ as the midpoint of the list in the $I(n)$ recursive call. From the induction, the algorithm will return $S = \text{Heavy(i, m)} \cup \text{Heavy(m+1, j)}$ as the possible heavy element in $A[i \ldots j]$. Since all array may only have zero, one, or two heavy elements, it follows that $|S| \leq 4$. Then check for every element

x in S, whether it is a heavy element in $A[i \dots j]$, by counting the number of x in $A[i \dots j]$ by calling the CountHeavy(x,i,j). If the count is greater than $2n/5$, then x is the heavy element in $A[i \dots j]$ and append it to R. After checking for all x, return the R, which is the heavy element of $A[i \dots j]$. It also follows that $|R| \leq 2$.

Therefore, Heavy(i, j) always correctly returns the heavy element in $A[i \dots j]$ and $I(n)$ is true for all $n \geq 1$.

(c) Let T(n) be the worst case of total operations of all types performed by your algorithm. Derive a recurrence relation for $T(n)$. Show that $T(n) = O(n \log n)$

(i) For the base case $n \in \{1, 2\}$, $T(1) = T(2) = 1$ since the only operation is checking the length of the array.

(ii) For the general case $n > 2$, determining S takes $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 4$ number of operations, for the recursive call and the set operation (recall the $|\text{Heavy(i,j)}| \leq 2$). To check the count of all x in S, it takes at most $4n$ operation, as $|S| \leq 4$. Therefore, the recurrence relation of the algorithm is:

$$\forall n \geq 1, \quad T(n) \leq T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + 4n; \quad T(1) = T(2) = 1; \tag{S3.1}$$

This recurrence relation implies $T(n) = O(n \log n)$ running time.

Problem 4: Running time of Algorithm

For each pair of expressions (A, B), indicate the most appropriate relation, whether A is O, Ω, Θ of B.

(a) $A = n^2 + n^2 \log n, \quad B = 5n - 7n^3 + 2n^4, \quad \mathbf{A = O(B)}$

(b) $A = \log_{100}((n+100)!), \quad B = \ln n^n; \quad \mathbf{A = \Theta(B)}$

(c) $A = ({}^3\sqrt{2})^{\frac{1}{\log_n 10}}, \quad B = 2^{\sqrt{3 \log_2 n}}; \quad \mathbf{A = O(B)}$

(d) $A = \sum_{i=1}^{n} k^3, \quad B = 12 \times \binom{n}{4}; \quad \mathbf{A = \Theta(B)}$

(e) $A = n^6 2^{n(\log n)^3}, \quad B = n^8 + 2021^{2020^{2019}}; \quad \mathbf{A = \Omega(B)}$

(f) $A = \sum_{k=1}^{n} \frac{1}{k(k+1)}, \quad B = ln \sum_{k=1}^{n} \frac{1}{k} \quad \mathbf{A = O(B)}$

(g) $A = n(1 + (-1)^n), \quad B = n(1 + (-1)^{n+1}); \quad$ **none of the relations is satisfied**

Problem 5: Asymptotic Upper Bounds

Give asymptotic upper bounds for $T(n)$ satisfying the following recurrences.

(a) $T(1) = 1; \quad T(n) = 6T(n/4) + n^2 \quad \text{for } n > 1$

$$\begin{aligned}
T(n) &= 6T(n/4) + n^2 \\
&= 6(6T(n/4^2) + (n/4)^2) + n^2 \\
&= 6^2T(n/4^2) + n^2(1 + 6/4^2) \\
&= 6^2(6T(n/4^3) + (n/4^2)^2) + n^2(1 + 6/4^2) \\
&= 6^3T(n/4^3) + n^2(1 + 6/4^2 + (6/4^2)^2) \\
&= 6^kT(n/4^k) + n^2\sum_{i=0}^{k-1}\left((6/16)^i\right)
\end{aligned}$$

Assume n is a power of 4 and let $k = \log_4 n$, then $T(n/4^k) = T(1) = 1$.

$$\begin{aligned}
&= 6^{\log_4 n} + n^2\sum_{i=0}^{k-1}\left((6/16)^i\right) \\
&= n^{\log_4 6} + n^2 \times \frac{8}{5}\left(1 - \left(\frac{3}{8}\right)^{k-1}\right) \\
&\le n^2 + n^2 \times \frac{8}{5} = \frac{13}{5}n \quad \text{for } n > 1 \\
&= \mathbf{O(n^2)}
\end{aligned}$$

(b) $T(1) = 1; \quad T(n) = 9T(n/3) + n \quad \text{for } n > 1$

$$\begin{aligned}
T(n) &= 9T(n/3) + n^2 \\
&= 9(9T(n/3^2) + n/3) + n \\
&= 9^2T(n/3^2) + n(1 + 1/3) \\
&= 9^2(9T(n/3^3) + n/3^2) + n(1 + 1/3 + 1/3^2) \\
&= 9^3T(n/3^3) + n(1 + 1/3 + 1/3^2) \\
&= 9^kT(n/3^k) + n\sum_{i=0}^{k-1}\left((1/3)^i\right)
\end{aligned}$$

Assume n is a power of 3 and let $k = \log_3 n$, then $T(n/3^k) = T(1) = 1$.

$$\begin{aligned}
&= 9^{\log_3 n} + n\sum_{i=0}^{k-1}\left((1/3)^i\right) \\
&= n^{\log_3 9} + n \times \frac{3}{2}\left(1 - \left(\frac{1}{3}\right)^{k-1}\right) \\
&= n^2 + n \times \frac{3}{2}\left(1 - \left(\frac{1}{3}\right)^{k-1}\right) \\
&\le n^2 + n^2 \times \frac{3}{2} = \frac{5n^2}{2} \quad \text{for } n > 1 \\
&= \mathbf{O(n^2)}
\end{aligned}$$