PROBLEM SET 2 – Manav Bilakhia

CSc 250, Spring 2014
Assigned: Tuesday, Week 2
Due: Tuesday, Week 3

Aaron G. Cass
Department of Computer Science
Union College

O MY

1. [From [2] pg. 25, #1-3]
   (a) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?
   (b) If I prove that an algorithm takes $O(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?
   (c) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on some inputs?
   (d) If I prove that an algorithm takes $\Theta(n^2)$ worst-case time, is it possible that it takes $O(n)$ on all inputs?

   You must explain your answers.

   Answer:

   (a) Yes. Since the worst case is bound from above by n^2. Therefore some cases could be $O(n)$ as $O(n)$ is always below.
   (b) Yes. Since the worst case is bound from above by n^2. Therefore all cases could be $O(n)$ as $O(n)$ is always below.
   (c) Yes. Since the worst case is $\Theta(n^2)$, all the cases do not have to be $\Theta(n^2)$ and it is possible to take $O(n)$ on some inputs.
   (d) No, The worst case is $\Theta(n^2)$ hence all cases cannot be $O(n)$.

2. [From [1] §2.2, #7]. Prove or disprove the following:
   (a) $t(n) \in O(g(n)) \rightarrow g(n) \in \Omega(t(n))$
   (b) $\forall \alpha > 0 : \Theta(\alpha g(n)) = \Theta(g(n))$. Remember that to prove two sets A and B equal, you must prove that any element in A is also in B **and** vice versa.
   (c) $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

   Answer:

   (a) *Proof.* Assume $\exists n_1 \geq 0$, $c_1 > 0$: $\forall n \geq n_1$: $t(n) \leq c_1 g(n)$. We want to show that $\exists n_2 \geq 0$, $c_2 > 0$: $\forall n \geq n_2$: $g(n) \geq c_2 t(n)$. We know that $\frac{1}{c_1} t(n) \leq g(n)$ where $\frac{1}{c_1}$ and c_2 are both constants. Hence we can write $c_2 t(n) \leq g(n)$ as desired. □
   (b) *Proof.* Assume $f(x) \in \Theta(\alpha g(n))$. We want to show that $f(x) \in \Theta(g(n))$. We know $\exists n_0 \geq 0, c_1 > 0$. Therefore $\forall n \geq n_0, f(n) \leq c_1 \alpha g(n)$. We can now say that $f(n) \in \Theta(g(n))$. Concluding that $\Theta(\alpha g(n)) \subset \Theta(g(n))$.
   Now let us assume $f(n) \in \Theta g(n)$. We want to show that $f(n) \in \Theta(\alpha g(n))$ for $\alpha > 0$. Therefore $\exists n_1 \geq 0, c_2 > 0 : \forall n \geq n_2, f(n) \leq c_2 g(n)$. Therefore $f(n) \leq \frac{c_2}{\alpha} \alpha g(n) = c_1 \alpha g(n), \forall n \geq n_0$ where $c_1 = \frac{c}{\alpha} > 0$
   Therefore $f(n) \in \theta(\alpha g(n))$ which we can write as $\Theta(\alpha g(n)) \supset \Theta(g(n))$. Hence we can say $\Theta(\alpha g(n)) = \Theta(g(n))$ □

(c) *Proof.* Case 1: $\Theta(g(n)) \subset O(g(n)) \cap \Omega(g(n))$
Assume $f(n) \in \Theta(g(n))$. Therefore we know $\exists n_0 \geq 0, c_1 > 0, c_2 > 0 : \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$. According to the definition of $f(n) \in \Omega(g(n))$. Therefore we can conclude $\Theta(g(n)) \subset O(g(n)) \cap \Omega(g(n))$.

Case 2: $\Theta(g(n)) \supset O(g(n)) \cap \Omega(g(n))$
Assume $f(n) \in \Omega(g(n))$. Therefore we know that $\exists n_1 > 0$ and $c_1 > 0 : \forall n \geq n_1, f(n) \geq c_1 g(n)$. Assume $f(n) \in O(g(n))$. We know that $\exists n_2 \geq 0$ and $c_2 > 0 : \forall n > n_2, f(n) \geq c_2 g(n)$. Therefore $\exists n_1 \geq 0, n_2 \geq 0, c_1 > 0$ and $c_2 > 0 : \forall n > MAX(n_1, n_2), c_1 g(n) \leq f(n) \leq c_2 g(n)$ which can be written as $f(n) \in \Theta(g(n))$. Therefore we know that $\Theta(g(n)) \supset O(g(n)) \cap \Omega(g(n))$.
From case 1 and case 2, we can conclude that $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$ $\square$

ANALYSIS

1. [From [1] §2.4, #3]. Consider the following recursive algorithm for computing the sum of the first n cubes: $S(n) = 1^3 + 2^3 + ... + n^3$.

   S(n)
   Input: A positive integer n
   Output: The sum of the first n cubes

   ```
   if n = 1 then
       return 1
   else
       return S(n − 1) + n × n × n
   ```

   (a) Set up and solve a recurrence relation for the exact number of times the algorithm's basic operation is executed.

   (b) How does this algorithm compare with the straightforward non-recursive algorithm for computing this function?

Answer:

(a)

$$
\begin{aligned}
c(n) &= c(n-1) + 2 \text{ from the base case we know } c(1) = 0 \\
&= [c(n-2) + 2] + 2 \\
&= [[c(n-3) + 2] + 2] + 2 \\
&= [[[c(n-4) + 2] + 2] + 2] + 2 \\
&. \\
&. \\
&. \\
&= c(n-i) + 2i \\
&. \\
&. \\
&= c(1) + 2(n-1) \\
&= 2(n-1)
\end{aligned}
$$

(b) Let us first write down a simple straight forward algorithm. S(n)
Input: A positive integer n
Output: The sum of the first n cubes

$sum \leftarrow 1$
for $i \leftarrow 2$ to n **do**
    $sum \leftarrow sum + (i * i * i)$
return sum

Comparing the two algorithms,

$$\begin{aligned}
&= \sum_{i=2}^{n} 2 \\
&= 2\sum_{i=2}^{n} 1 \\
&= 2(n-1)
\end{aligned}$$

They have the same runtime hence are equally time efficient.

2. [From [1] §2.4, #8]. Consider the following recursive algorithm:

   MIN1($A[0..n-1]$)
   Input: Array $A[0..n-1]$ of real numbers

   if $n = 1$ **then**
       return A[0]
   else
       temp $\leftarrow$MIN1($A[0..n-2]$)
       if temp $\leq A[n-1]$ **then**
           return temp
       else
           return $A[n-1]$

   (a) What does this algorithm compute?
   (b) Set up a recurrence relation for the algorithm's basic operation count and solve it.

Answer:

(a) The item finds the minimum value from the given set of values
(b)

$$\begin{aligned}
c(n) &= c(n-1) + 1 \text{ from the base case we know } c(1) = 0 \\
&= [c(n-2)+1]+1 \\
&= [[c(n-3)+1]+1]+1 \\
&= [[[c(n-4)+1]+1]+1]+1 \\
&. \\
&. \\
&. \\
&= c(n-i)+i \\
&. \\
&. \\
&= c(1)+(n-1) \\
&= (n-1)
\end{aligned}$$

3. [From [1] §2.4, #9]. Consider another algorithm for solving the problem from the previous exercise. This version recursively divides an array into two halves. It is called by calling MIN2($A[0..n-1], 0, n-1$).

   MIN2($A[0..n-1], l, r$)
   Input: Array $A[0..n-1]$ of real numbers, and indices l and r

```
if l = r then
    return A[l]
else
    temp1 ← MIN2(A, l, ⌊(l + r)/2⌋)
    temp2 ← MIN2(A, ⌊(l + r)/2⌋ + 1, r)
    if temp1 ≤ temp2 then
        return temp1
    else
        return temp2
```

(a) Set up and solve a recurrence for the exact count of the number of times the algorithm's basic operation is executed.

(b) Which of MIN1 or MIN2 is more efficient?

(c) Is it possible to solve the same problem more efficiently than both of them?

Answer:

(a)

$$
\begin{aligned}
c(n) = c\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor + 1 \text{ from the base case we know } c(1) = 0 \\
&\text{solving for } n = 2^m \text{ inputs} \\
&= 2c(2^{m-1}) + 1 \\
&= 2[2c(2^{m-1-1}) + 1] + 1 \\
&= 2[2[2c(2^{m-1-1-1}) + 1] + 1] + 1 = 2^3 c(2^{m-3}) + 2^2 + 2 + 1 \\
&. \\
&. \\
&= 2^i c(2^{m-i}) + 2^{i-1} + 2^{i-2} + ... + 1 \\
&. \\
&. \\
&= 2^m c(2^{m-m}) + 2^{m-1} + 2^{m-2} + ... + 1 \\
&= 2^m - 1 \\
&= n - 1
\end{aligned}
$$

(b) They both have the same time complexity. Although MIN1 utilizes less memory than MIN2 as MIN2 has two recursive calls whereas MIN1 has one recursive call.

(c) We cannot solve the problem any faster but we could utilize less memory by using loops instead of recursion.

Honor Code Affirmation

I affirm that I have carried out my academic endeavors with full academic honesty
Manav Bilakhia

References

[1] Anany Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, 2003.

[2] Steven S. Skiena. *The Algorithm Design Manual*. Springer TELOS, 1998.