UWI

Design and Analysis of Algorithms

Tutorial 5- Sample Solutions

Divide and Conquer Approach

It is important that we learn the theory behind the divide and conquer approach and its algorithms.

1. Explain the theory behind the divide and conquer approach
   (a) What is meant by the divide and conquer approach?
   The divide and conquer approach follows three steps:
      1. Divide instance of problem into two or more smaller instances
      2. Solve the smaller instances recursively
      3. If necessary, solutions obtained for smaller instances are combined to get solution to the original instance

   (b) Is divide and conquer more efficient than brute force?
   Generally, yes. Like decrease and conquer, the divide and conquer algorithms are usually more efficient than their brute force counterparts because dividing up the work can reduce the total amount of work to do. Since they may have less work to do, they typically take less time, hence more efficient. Note that there are some limited applications where brute force problems may prove more effective.

2. Master's theorem can be used to calculate the time complexities of recurrence relations.
   (a) State Master's Theorem and its uses.
   Master's theorem states that the recurrence for running time $T(n)$ is

$$T(n) = aT(n/b) + f(n), \tag{1}$$

   where f(n) is time spent on dividing and combining. If $f(n) \in \Theta(n^d)$ with $d \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d logn) & \text{if } a = b^d \\ \Theta(n^{log_b^a}) & \text{if } a > b^d \end{cases} \tag{2}$$

   (b) Calculate the time complexity of the Quick Hull algorithm (which has the following recurrence relation) using Master's Theorem.

$$T(n) = 2T(\frac{n}{2}) + n \tag{3}$$

   For master's theorem, we observe that $a = 2$ and $b = 2$ from our recurrence relation. We must solve for d, where $b^d = f(n)$ and $b = 2$ and $f(n) = n \implies f(n) \in \Theta(n^1)$. In this case $d = 1$.
   Now we check to see how a stacks up against b^d. We observe that $a = b^d$ because $2 = 2^1$. This means that, according to Master's theorem, $T(n) \in \Theta(n^1 logn)$ or $T(n) \in \Theta(nlogn)$.

   (c) Calculate the time complexity of the Merge Sort algorithm (which has the following recurrence relation) using Master's Theorem.

$$C_w(n) = 2C_w(\frac{n}{2}) + C_{merge}(n), \text{ for } n > 1 \tag{4}$$
$$C_{merge}(n) = n - 1 \tag{5}$$
$$C_w(1) = 0 \tag{6}$$

For master's theorem, we observe that $a = 2$ and $b = 2$ from our recurrence relation. We must solve for d, where $b^d = f(n)$ and $b = 2$ and $f(n) = n - 1 \implies f(n) \in \Theta(n^1)$. In this case $d = 1$.
Now we check to see how a stacks up against b^d. We observe that $a = b^d$ because $2 = 2^1$. This means that, according to Master's theorem, $C_w(n) \in \Theta(n^1 logn)$ or $C_w(n) \in \Theta(nlogn)$.

3. Solve the following recurrence relation for the Quick Sort algorithm using the backward substitution method.

$$C_b(n) = 2C_b(\frac{n}{2}) + n, \text{ for } n > 1 \tag{7}$$
$$C_b(1) = 0 \tag{8}$$

We proceed by using the backward substitution method, where $n = 2^k$.

$$\begin{aligned}
C_b(n) = C_b(2^k) &= 2C_b(\frac{2^k}{2}) + 2^k = 2C_b(2^{k-1}) + 2^k && (9)\\
\text{sub } 2^{k-1} \text{ into } C_b \quad &= 2\left[2C_b(\frac{2^{k-1}}{2}) + 2^{k-1}\right] + 2^k = 2^2C_b(2^{k-2}) + 2 \cdot 2^k && (10)\\
\text{sub } 2^{k-2} \text{ into } C_b \quad &= 2^2\left[2C_b(\frac{2^{k-2}}{2}) + 2^{k-2}\right] + 2 \cdot 2^k = 2^3C_b(2^{k-3}) + 3 \cdot 2^k && (11)\\
&\vdots && (12)\\
\text{we notice a pattern} \quad &= 2^iC_b(2^{k-i}) + i2^i && (13)\\
&\vdots && (14)\\
\text{we generalize for k} \quad &= 2^kC_b(2^{k-k}) + k2^k && (15)\\
&= 2^kC_b(2^0) + k2^k && (16)\\
&= 2^kC_b(1) + k2^k && (17)\\
\text{recall } C_b(1) = 0 \quad &= k2^k && (18)
\end{aligned}$$

We know that $n = 2^k$, which means that if we solve for k, we will get $k = log_2(n)$. This means that $C_b(n) = nlog_2n$, which implies that $C_b(n) \in \Theta(nlogn)$.

4. Solve the following recurrence relation for the n-digit multiplication algorithm using the backward substitution method.

$$M(n) = 3M(\frac{n}{2}), \text{ for } n > 1 \tag{19}$$
$$M(1) = 1 \tag{20}$$

We proceed by using the backward substitution method, where $n = 2^k$.

$$\begin{aligned}
M(n) = M(2^k) = 3M(\frac{2^k}{2}) &= 3M(2^{k-1}) && (21)\\
\text{sub } 2^{k-1} \text{ into } M \qquad &= 3\left[3M(\frac{2^{k-1}}{2})\right] = 3^2M(2^{k-2}) && (22)\\
\text{sub } 2^{k-2} \text{ into } M \qquad &= 3^2\left[3M(\frac{2^{k-2}}{2})\right] = 3^3M(2^{k-3}) && (23)\\
&\vdots && (24)\\
\text{we notice a pattern} \qquad &= 3^iM(2^{k-i}) && (25)\\
&\vdots && (26)\\
\text{we generalize for k} \qquad &= 3^kM(2^{k-k}) && (27)\\
&= 3^kM(2^0) && (28)\\
&= 3^kM(1) && (29)\\
\text{recall } M(1) = 1 \qquad &= 3^k && (30)
\end{aligned}$$

We know that $n = 2^k$, which means that if we solve for k, we will get $k = log_2(n)$. This means that $M(n) = 3^{log_2 n}$, which can be simplified to $M(n) = n^{log_2 3} = n^{1.585}$. This implies that $M(n) \in O(n^2)$.

5. Solve the following recurrence relation using the backward substitution method.

$$T(n) = T(\sqrt{n}) + 1, \text{ for } n > 2, \text{ where } T(n) \text{ is constant for } n \leq 2 \qquad (31)$$

We proceed by using the backward substitution method, where $n = 2^{2^k}$.

$$\begin{aligned}
T(n) = T(2^{2^k}) = T(\sqrt{2^{2^k}}) + 1 &= T(2^{2^{k-1}}) + 1 && (32)\\
\text{sub } 2^{2^{k-1}} \text{ into } T \qquad &= \left[T(\sqrt{2^{2^{k-1}}}) + 1\right] + 1 = T(2^{2^{k-2}}) + 2 && (33)\\
\text{sub } 2^{2^{k-2}} \text{ into } T \qquad &= \left[T(\sqrt{2^{2^{k-2}}}) + 1\right] + 2 = T(2^{2^{k-3}}) + 3 && (34)\\
&\vdots && (35)\\
\text{we notice a pattern} \qquad &= T(2^{2^{k-i}}) + i && (36)\\
&\vdots && (37)\\
\text{we generalize for k} \qquad &= T(2^{2^{k-k}}) + k && (38)\\
&= T(2^{2^0}) + k && (39)\\
&= T(2^1) + k && (40)\\
\text{recall } T(2) = c \qquad &= c + k && (41)
\end{aligned}$$

We know that $n = 2^{2^k}$, which means that if we solve for k, we will get $k = log_2(log_2(n))$. This means that $T(n) = c + log_2(log_2(n))$, which implies that $T(n) \in \Theta(loglogn)$.