UWI

Design and Analysis of Algorithms

Tutorial 4 - Sample Solutions

Decrease and Conquer Approach

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

1. Explain the theory behind the decrease and conquer approach
   (a) What is meant by the decrease and conquer approach?
   The decrease and conquer approach follows three steps:
      1. Reduce the initial problem instance to smaller instance of the same problem
      2. Solve this smaller instance of the problem
      3. Extend solution of smaller instance to obtain a solution to initial instance

   (b) Is decrease and conquer more efficient than brute force?
   Generally, yes. Decrease 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 for efficient.

   (c) What are the three types of decrease and conquer algorithms?
   The three approaches are as follows:
      1. Decrease by a constant value e.g. insertion sort, generating permutations etc.
      2. Decrease by a variable value e.g. interpolation search
      3. Decrease by a constant factor e.g. binary search etc.

2. Consider the following decrease by a constant algorithms
   (a) Use the minimal change algorithm to generate permutations for the set {6,7,8}.
   The minimal change algorithm seeks to take a previously generated permutation and insert a new element from right to left or left to right. For our example, we start with a subset containing only 6. Then we will insert the element 7, then 8 in the right to left or left to right patterns.

Action	Result		
start	6		
insert 7 into 6 right to left	67	76	
insert 8 into 67 right to left	678	687	867
insert 8 into 76 left to right	876	786	768

Table 1: A small instance of the minimal change algorithm operation

   From the above table we observe that it produced the subsets {6,7}, {7,6}, {6,7,8}, {6,8,7}, {8,6,7}, {8,7,6}, {7,8,6} and {7,6,8}. Subsets with less than 2 items are ignored for this operation.

   (b) Use the binary reflected grey code algorithm to generate combinations for the set {1,2,3}.
   Binary reflected grey code seeks to generate binary numbers with the condition that it differs from its predecessor by a single bit. In this case, our set contains 3 elements. Therefore, we must generate all binary numbers that can be represented by 3 bits according to binary reflected grey code. We will end up with: 000, 001, 011, 010, 110, 111, 101 and 100. To apply this to subset generation, we interpret a 1 as present and a 0 as absent directly corresponding to the order of the elements in our set.

Code	Subset
000	$\emptyset$
001	{3}
011	{2,3}
010	{2}
110	{1,2}
111	{1,2,3}
101	{1,3}
100	{1}

Table 2: A small instance of the binary reflected grey code algorithm operation

3. Analyse the following recurrence relation for the worst case binary search algorithm and answer the following questions.

$$C_w(n) = C_w(\lfloor \frac{n}{2} \rfloor) + 1, \text{ for } n > 1 \tag{1}$$
$$C_w(1) = 1 \tag{2}$$

(a) Calculate the time complexity using the backward substitution method.
We proceed by using the backward substitution method, where $n = 2^k$.

$$\begin{aligned}
C_w(n) = C_w(2^k) &= C_w(\frac{2^k}{2}) + 1 = C_w(2^{k-1}) + 1 && (3)\\
\text{sub } 2^{k-1} \text{ into } C_w) &= \left[C_w(\frac{2^{k-1}}{2}) + 1\right] + 1 = C_w(2^{k-2}) + 2 && (4)\\
\text{sub } 2^{k-2} \text{ into } C_w) &= \left[C_w(\frac{2^{k-2}}{2}) + 1\right] + 2 = C_w(2^{k-3}) + 3 && (5)\\
&\vdots && (6)\\
\text{we notice a pattern} &= C_w(2^{k-i}) + i && (7)\\
&\vdots && (8)\\
\text{we generalize for k} &= C_w(2^{k-k}) + k && (9)\\
&= C_w(2^0) + k && (10)\\
&= C_w(1) + k && (11)\\
\text{recall } C_w(1) = 1 &= 1 + k && (12)
\end{aligned}$$

We know that $n = 2^k$, which means that if we solve for k, we will get $k = log_2(n)$. This means that $C_w(n) = log_2(n) + 1$, which if we ignore the constants implies that $C_w(n) \in \Theta(logn)$. This is more efficient than the worst case for a simple linear search, which would have an efficiency of $\Theta(n)$

(b) Calculate the time complexity using master's theorem.
Master's theorem states that the recurrence for running time $T(n)$ is

$$T(n) = aT(n/b) + f(n), \tag{13}$$

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

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d logn) & \text{if } a = b^d \\ \Theta(n^{log_b^a}) & \text{if } a > b^d \end{cases} \tag{14}$$

For master's theorem, we observe that $a = 1$ and $b = 2$ from our recurrence relation. We must solve for d, where $b^d = f(n)$ and $b = 2$ and $f(n) = 1$.

$$\begin{aligned}
2^d &= 1 && (15)\\
log_2(2^d) &= log_2(1) && (16)\\
dlog_2(2) &= 0 && (17)\\
d &= 0 && (18)
\end{aligned}$$

Now we check to see how a stacks up against b^d. We observe that $a = b^d$ because $1 = 2^0$. This means that, according to Master's theorem, $C_w(n) \in \Theta n^0 logn$ or $C_w(n) \in \Theta logn$

Note that master's theorem only works with divide and conquer algorithms and decrease and conquer algorithms that reduce by a factor.