1. $T(n) = aT(n/b) + O(n^d)$

$a\, n/b$ = number of subproblems

$n/b$ = size of subproblem

$n^d$ = time needed to conquer

when $d = \log_b a$ then $T(n)$ is $O(n^d \log_b n)$

" $d > \log_b a$ then $T(n) = O(n^d)$

" $d < \log_b a$ then $T(n) = O(n^{\log_b a})$

(a)

$a = 5$

$n/b = n/2$

$n^d = n$

$T(n) = 5T(n/2) + O(n)$

$d = 1$

$\log_2 5 = 2.32$

~~$T(n) = O(n^{2.32} \log_2 n)$~~

$d < \log_b a$

$\therefore T(n) = O(n^{\log_2 5})$

$= O(n^{2.32})$

(b)

$a = 2$

$n/b = n-1$

$n^d = 1$

$T(n) = 2T(n-1) + O(1$

say,

$T(1) = 2T(1-1) + O(1)$

$= 0$

$T(2) = 2T(1)$

$= 2 \times T(1)$

$T(3) = 2T(2)$

$= 2 \times 2T(1)$

$\cdot\, T(n) = 2^n T(1)$

(c)

$a = 9$

$n/b = n/3$

$O(n^d) = O(n^2)$

$T(n) = 9T(n/3) + O(n^2)$

$b = 3$

$d = 2$

$\log_b a = \log_3 9 = 2$

$\cancel{b > \log_b a}$

$\cancel{\therefore T(n) = \theta(n^d)}$

$\cancel{= O(n^2}$

$d = \log_b a$

$\therefore T(n) = O(n^d \log_b n)$

$= O(n^2 \log_3 n)$

$\therefore$ I would choose algorithm B, because its time complexity is minimum.

2. (a) We split array A into 2 subproblems of half of the subarrays. $A_1$ and $A_2$ of equality size. After that we do a linear time operation to decide whether it is possible to find a majority element. The recurrence therefore given by $T(n) = 2T(n/2) + O(n)$.

The complexity of algorithm comes to $O(n \log n)$

Get majorelement(a[1 .... n])

Input = Array a of objects.

Output = Majority element of a

If n = 1 : return a[1]

$k = n/2$

$elem_{lsub}$ = Getmajorelement(a[1....k])

$elem_{rsub}$ = Getmajorelement(a[k+1....n])

If $elem_{lsub} = elem_{rsub}$:

  return $elem_{lsub}$

lcount = GetFrequency (a[1... n], $elem_{lsub}$)

rcount = Get Freqeuency [a[1 ... n], $elem_{rsub}$

If lcount > k + 1

  return $elem_{lsub}$

else if rcount > k+1 :
 return $elem_{rsub}$
else return Nomajorelement :

(b) Recurrence relation for the algorithm is given belo

$T(n) = T(n/2) + O(n)$

The processing of array in the recursive functio is done is O(n) time.

Getmajorityelement (a[1 .... n]

Input : Array a of objects

Output / Majority element of a

If n = 2 :

 If a[1] = a[2] return a[1]

else

 return No-majority-element;

For i = 1 to n :

 If (a[i] = a[i+1]) :

  Insert a[i] into temp

 i = i+1 ;

return GetMajorityelement (temp)

Checkcan (a[1 .... n])

Input : Array a of objects

Output : Majority of a element of a

m = Getmajorelement (a[1 ....n])

freq = Getfrequency (a[1... n], m)

If freq > [n/2] +1 :

  return m;

else

  return no-majority-Element.

3.

a = 68

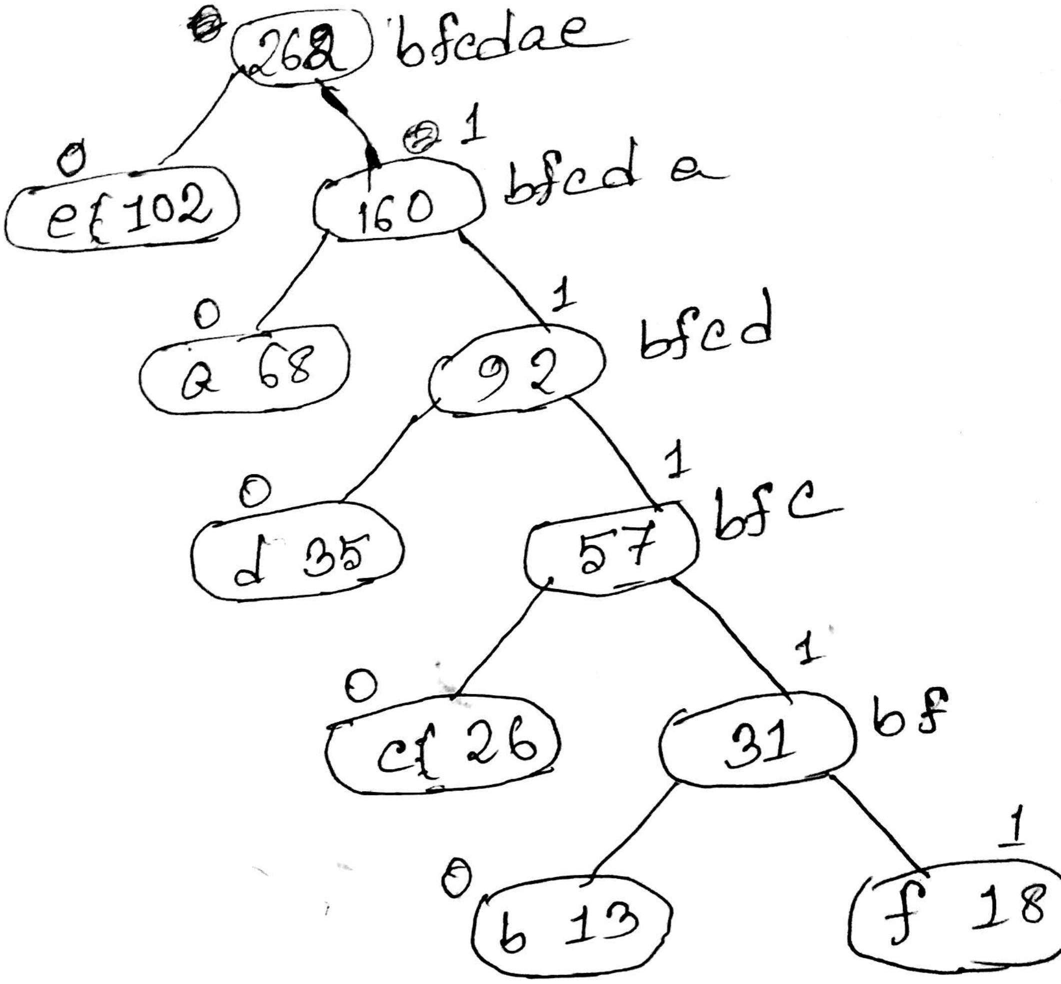
b = 13

c = 26

d = 35

e = 102

f = 18




a = 10

b = 11110

c = 1110

d = 110

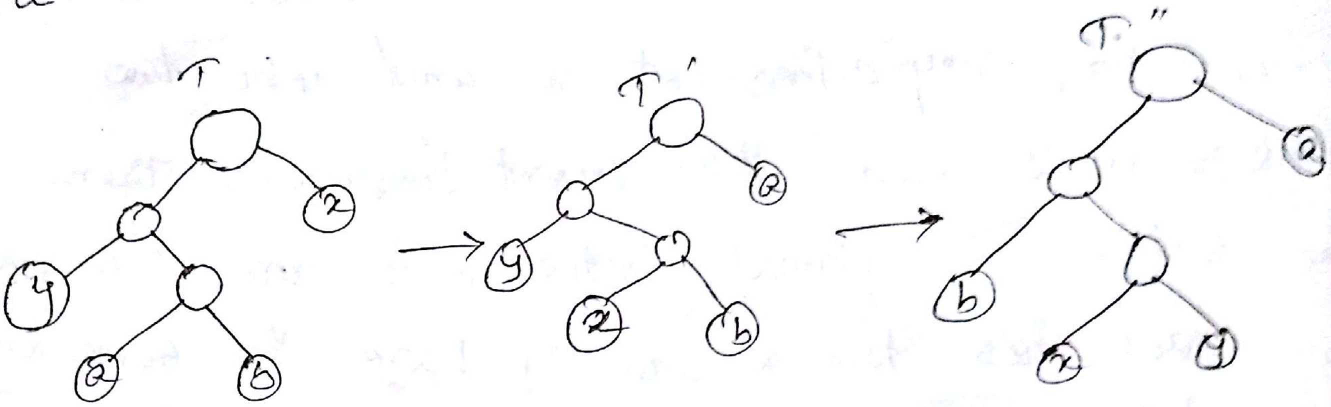
e = 0

f = 11111

Scanned by CamScanner

Let $c$ be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let $x$ and $y$ be two characters in $C$ have the lowest frequencies. Then there exists an optimal prefix code for $C$ in which the codewords for $x$ and $y$ have the same length and differs only in the last bit.

Proof: The idea of the proof is to the tree $T$ representing an arbitrary optimal prefix code such that the characters $x$ and $y$ appear as sibling leaves of maximum depth in the new tree. Let $a$ and $b$ be two characters that are sibling leaves of maximum depth in $T$. In the remainder of the proof, it is possible that we could have $x.freq = a.freq$ or $y.freq = b.freq$. However, if we had $x.freq = b.freq$ then we would also have $a.freq = b.freq = x.freq = y.freq$. At fig 1 shows, we exchange the positions in ~~#~~ $T$ of $a$ and $x$ to produce a tree $T'$. and then we exchange the positions in $T'$ of $b$ and $y$ to produce a tree $T''$.




$B(T) - B(T')$

$$= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T}(c)$$

$$= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a)$$

$$= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x)$$

$$= (a.freq - x.freq)(d_T(a) - d_T(x))$$

$$\geq 0$$

Because both $a.freq - x.freq$ and $d_T(a) - d_T(x)$ are nonnegative. More specifically, $a.freq - x.freq$ is nonnegative because $x$ is a minimum-frequency leaf, and $d_T(a) - d_T(x)$

is nonnegative because $a$ is a leaf of maximum depth in $T$. Similiarly, exchanging $y$ and $b$ does not increase the cost, and so $B(T') - B(T'')$ is nonnegative. Therefore, $B(T'') \leq B(T)$, and since $T$ is optimal, we have $B(T) \leq B(T'')$, which implies $B(T'') = B(T)$. Thus $T''$ is an optimal tree in which $x$ and $y$ appear as sibling leaves of maximum depth, from which the lemma follows.

We first show how to express the cost $B(T)$ of tree $T$ in terms of the cost $B(T')$ of tree $T'$, by considering the component costs in equation. For each character $c \in C - \{x, y\}$, we have that $d_T(c) = d_{T'}(c)$, and hence $c.freq \cdot d_T(c) = c.freq \cdot d_{T'}(c)$. Since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have that, ~~x.freq.dx y~~

$$x.freq \cdot d_T(x) + y.freq \cdot d_T(y) = (x.freq + y.freq)(d_{T'}(z) + 1)$$

$$= z.freq \cdot d_{T'}(z) + (x.freq + y.freq)$$

from which we conclude that

$$B(T) = B(T') + x.freq + y.freq$$

or, eventually

$$B(T') = B(T) - x.freq - y.freq$$

Scanned by CamScanner

we now prove the lemma by contradiction suppose that T does not represent an optimal prefix code for C. Without lose the generality, T'' has x and y as siblings. let T''' be the tree T'' with common parent of x and y replaced by a leaf z with frequency z.freq = x.freq + y.freq. Then

$$B(T''') = B(T'') - x.freq - y.freq$$

$$< B(T) - x.freq - y.freq$$

$$= B(T')$$

yielding a contradiction to the assumption T' represents an optimal code for C'. Thus T must represent an optimal code for alphabet C.

4. 0/1 knapsack problem

Algorithm:

// Input
// values (stored in array v)
// weights (stored in array w)
// Numbers of distinct items (n)
// knapsack capacity (w)

for j from 0 to w do:
  m[0, j] := 0
for i from 1 to n do:
  for j from 0 to w do:
    if w[i] > j then:
      m[i, j] := m[i-1, j]
    else:
      m[i, j] := max(m[i-1, j],
                m[i-1, j - w[i]] + v[i]

(*) The running time = $O(1) O(nw)$

$= O(nw)$

6. The max reliability for constructing system with stage 1....j with budget b. Redundancy(b,j)

```
for j=1 to n
    Pr(0,j) = 0
for b=1 to B
    Pr(b,1) = r-1
for b=1 to B
    for j=1 to n
        for k=1 to b/c - j
            Pr(b,j) = max{Pr(b-c, j-1)* (1-(1-r)^k)}
        Red [B][j] = k
print (Red [][])
```