2|23

Assignment - 3

2. Given 'n' number of customers and let the time taken for each customer be $t_1, t_2 \ldots t_n$. An i^{th} customer, $1 \leq i \leq n$ needs a service time t_i

Using greedy algorithm, we can compute the optimal order and the total waiting time for customers. In the following algorithm, we ~~compute the~~ sort the given time array in ascending order such that the shortest time comes first. Later, we compute the optimal time (waiting time) of all customers.

Merge Sort is used for sorting and then sum is calculated. The time complexity of merge sort is $O(n \lg n)$ and for sum time complexity is $O(n^2)$. Thus, the total time complexity is $O(n^2)$.

Algorithm:

```
Merge(A, l, m, r)
{
    n1 = m - l + 1;
    n2 = r - m;
    declare temp arrays L, R.
    for i = 0 to n1
        L[i] = A[l + i]
    for j = 0 to n2
        R[i] = A[m + 1 + j]

    i = 0, k = l, j = 0
    while i < n1 and j < n2
    {
        if L[i] <= R[j]
        {
            A[k] = L[i]
            i++;
        }
        else {
            A[k] = R[j]
            j++
        }
    }
    while i < n1 {
        A[k] = L[i]
        i++ ; k++
    }
```

```
while j < n2 {
    A[k] = R[j]
    k++, j++
  }
}

MergeSort(A, l, r)
{
  if {l < r
  {
    m = l + (r-1)/2
    MergeSort(A, l, m)
    MergeSort(A, m+1, r)
    merge(A, l, m, r)
  }
}

OptimalWaitingTime(A)
{
  sum = 0
  for i = 0 to A-length-1
    for j = 0 to i
      sum = sum + A[j]
}
```

Scanned by CamScanner

Proof: Consider Time $T = \{5, 20, 15, 7, 3, 1, 12, 10\}$

Using the algorithm, we sort T into ascending order, i.e.,

$$T = \{1, 3, 5, 7, 10, 12, 15, 20\}$$

Now, the optimal sum is

$$Sum = 1 + (1+3) + (1+3+5) + \dots + (1+3+5+7+10+12+15+20)$$

$$Sum = 290$$

Let us take a contradicting array, i.e., which is not sorted.

$$T = \{5, 7, 15, 3, 20, 1, 10, 12\}$$

$$Sum = 5 + (5+7) + (5+7+15) + \dots + (5+7+15+3+20+1+10+12)$$

$$Sum = 309.$$

Let Time(T) has $t_1, t_2, t_3 \dots t_n$. The time taken for an optimal order, has an i^{th} element t_i at place i, and time taken for other order at i^{th} position be t_j such that $t_i < t_j$. Then the sum of waiting time for i^{th} position t_i will always be less than t_j. Hence it is proved that the optimal solution should be in ascending order.

3. 0-1 Knapsack problem:

The knapsack problem can be solved by using greedy technique and dynamic programming. In greedy approach, the time complexity is $O(n \lg n)$ and using dynamic programming, we can sort it in $O(nW)$ where n is number of items and W is the capacity of knapsack.

Algorithm:

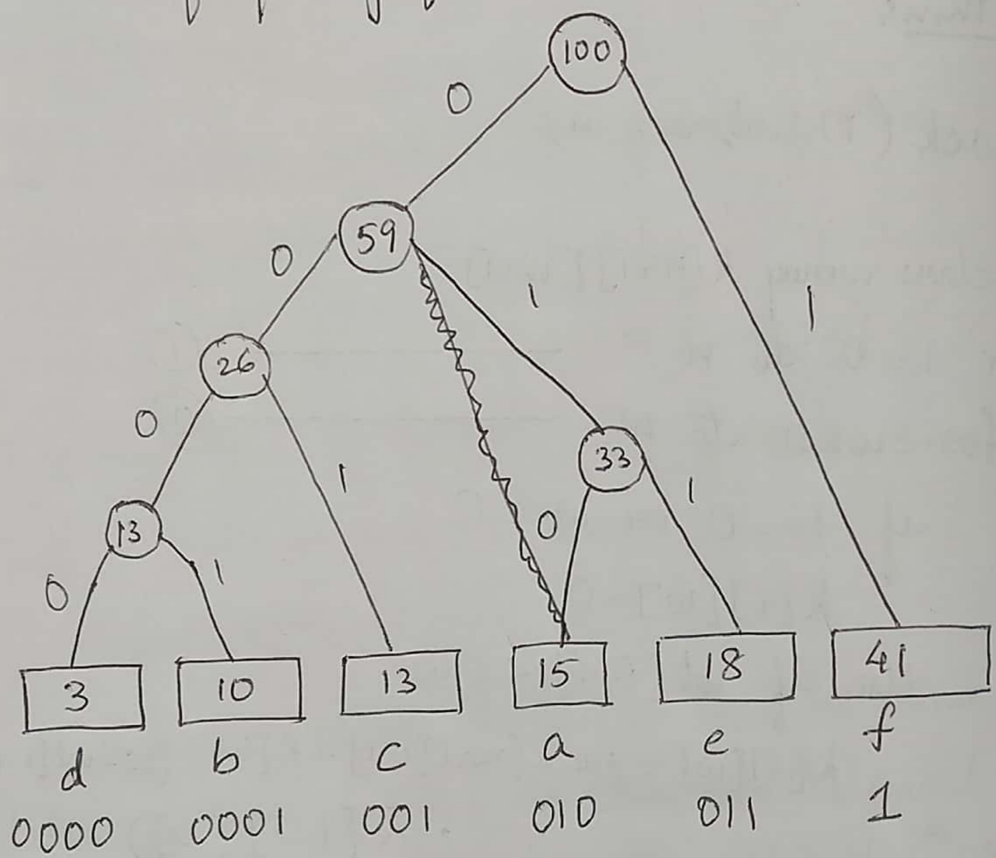
```
Knapsack (W, wt, val, n)
{
  declare array k[n+1][W+1]
  for i = 0 to n  ———————— (1)
   for w = 0 to W ———————— (2)
     if i == 0 or w == 0
        k[i][w] = 0
     else if wt[i-1] <= w
        k[i][w] = max(val[i-1] + k[i-1][w-wt[i-1]]
                      , k[i-1][w])
     else
        k[i][w] = k[i-1][w]

  return k[n][w]
}
```

In the above algorithm, the line (1) runs for 'n' times and point (2) runs for $n \times W$ times. Thus the worst case scenario, the time complexity for knapsack problem using dynamic programming is $O(nW)$.

4. Huffman Code:

Given letters are a, b, c, d, e, f.
and frequency of 15, 10, 13, 3, 18, 41



Huffman tree for given file

Codewords:

a: 010 d: 0000

b: 0001 e: 011

c: 001 f: 1

Size of compressed file:

Size of message: $(45+40+39+12+54+41 = 231)$

a: $15 \times 3 = 45$ d: $3 \times 4 = 12$

b: $10 \times 4 = 40$ e: $18 \times 3 = 54$

c: $13 \times 3 = 39$ f: $41 \times 1 = 41$

Size of encoded bits: $4+4+3+3+3+1 = 18$

Size of each ASCII character: $6 \times 8 = 48$.

Total size of compressed file:

size of (message + encoded bits + ASCII characters)

$= 231 + 18 + 48 = \boxed{297}$

5. Binomial Coefficient:

A binomial coefficient $\binom{n}{k} = \dfrac{n!}{k!(n-k)!}$

It can also be represented as,

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad \text{—— } (1)$$

Algorithm can be designed using ① by using dynamic programming $\{ {n \atop k} \}$

<u>Algorithm:</u>

```
BinomialCoeff(n, k)
{
  if (k==0 OR k==n)
     return 1
  return BinomialCoeff(n-1, k-1)+BinomialCoeff(n-1, k);
}
```

6. <u>Rod Cutting:</u>

Consider the bottom up approach for algorithm for rod cutting. It is described as follows.

```
Bottom-Up-Cut-Rod (p, n)
   let r[0..n] be new array
   r[0] = 0
   for j = 1 to n
      q = -∞
      for i = 1 to j  ———————— ①
         q = max(q, p[i]+r[j-i]) ← ②
```

$r[j] = q$
return $r[n]$

The maximum total cost of pieces is computed using the lines ① and ② which takes a time complexity of $O(n^2)$. In the given solution, it is mentioned that the maximum total cost is already calculated. Therefore, the time complexity of algorithm reduces to $O(n)$ as the complexity of lines ① and ② in the algorithm are now $O(1)$ since additional memory is required to save the maximum values for each piece.

1. Given S be a set of 'n' activities. For a given activity 'i' it has time interval (s_i, f_i)

* <u>Condition</u>: Two activities can be assigned to same room if their time intervals do not overlap.

* <u>Task</u>: Find an assignment of all activities to rooms using minimum number of rooms.

(a) Algorithm A: (Greedy by room)

In this algorithm, an activity can be assig till when the finish slot time is available for a given room.

Eg: There are 3 rooms: R_1, R_2, R_3

and 3 tasks: T_1 (10 am - 11 am)

T_2 (10:30 am - 12:30 pm)

T_3 (12:00 pm - 1:00 pm)

According to this algorithm, the greedy technique uses only room R_1, and places all tasks in the same room. Even if the task is accomplished by using minimum number of rooms. The given ~~cone~~ overlapping condition is violated. Hence algorithm A doesn't work for the scheduling solution.

(b) Algorithm B: (Greedy by time)

In this algorithm, an activity is placed by using the finish time f_i. The finish time of task t_j should be less than the start time of task t_i. Then t_i and t_j can be placed in the same room.

Eg: There are 3 rooms: R_1, R_2, R_3

and 3 Tasks: T_1, T_2, T_3

T_1 (10am - 11am), T_2 (10:30am - 12:30pm)
T_3 (12:00pm - 1:00pm)

The above algorithm uses rooms R_1 and R_2 where Task T_1 and T_3 can be placed in R_1 and task T_2 is placed in R_2. Even though this algorithm uses more rooms compared to algorithm A, the algorithm B doesn't over rule any condition and accomplishes the task. Thus Algorithm B works for the given scheduling problem.