Question 1

Knapsack under given condition

December 2016

Objective : Given a set of N objects of positive weights $\{w_i\}$, find the subset of these objects that maximizes their sum, subject to :

$$\sum_{i=1}^{m} w_{k_i} <= W_0 - \ln m$$

for given m>=0 and W_0 >=0

Approach 1 : Dynamic Programming

A part of the code used has been given below as well as further explanation. Please note that K[i][j] denotes the j-ln m sum acquired using the first i weights.
Code :

```
/*********************************START*************************************/
        int K[n+1][W+1],count[n+1][W+1];     //K stores
            cumulative sum. K[i][j]=sum <= j-log(m) using the
            first i weights
        int keep[n+1][W+1]; //to keep track of the weights
            used to obtain K[][].


        for(j=0;j<=W;j++)   //first row of zeros for the
            K[][] matrix
        {
             K[0][j] = 0;
             count[0][j]=0;
        }
        for(i=1;i<=n;i++)   //first column of zeros for the
            K[][] matrix
        {
             K[i][0]=0;
```

```
        }

/***************Main Algorithm******************/

        for(i=1;i<=n;i++)    //filling the rest of the K
            matrix
        {
             for(j=0;j<=W;j++)
             {
                  if(w[i-1]<=j && (K[i-1][j-w[i-1]] + w[i-1] >
                     K[i-1][j]) && K[i-1][j-w[i-1]] +
                     w[i-1]<=j-log(count[i-1][j-w[i-1]]+1)) //
                     if condition holds when weight added, add
                     weight
                  {
                       K[i][j] = K[i-1][j-w[i-1]] + w[i-1];
                          //w[i-1] considered
                       count[i][j]=count[i-1][j-w[i-1]]+1;
                          //increment the no. of used weights
                          at position(i,j)
                       keep[i][j]=1;    //note down that weight
                          has been used
                  }
                  else //if by adding the weight condition
                     fails
                  {
                       K[i][j] = K[i-1][j];     //w[i-1] not
                          considered
                       count[i][j]=count[i-1][j];  //count is
                          same as without w[i-1]
                       keep[i][j]=0;    //note down that weight
                          has not been used
                  }

             }
        }
/*********************************END**********************************/
```

And hence the table for computing the maximum sum of weights possible is formulated. This table is written on to the output file and this has been done for each test case. The maximum possible weight is stored in K[n][W].

Now, inorder to find the weights used in getting the maximum sum subject to the given condition, the below given code has been used :

```
/*********************************START*********************************/
        j=W;    //initialise max_weight
        for(i=n;i>0;i--)    //printing out the used weghts
            by traversing through the keep matrix
             if(keep[i][j]==1)   //weight used
             {
                 printf("   %d",w[i-1]);
                 j=j-w[i-1]; //subtract the used weight from
                     the cumulative sum
             }
/*********************************END*********************************/
```

And hence, the weights are calculated by using the keep[][] matrix that had been updated while formulating our weight table K[][]. Each time a weight was used, keep[][] was updated to 1 from 0. And hence I have used somewhat like a path retracing algorithm (a sort of backtracking) to find the weights used. (Note that j denotes the cumulative sum and i denotes the i^{th} weight.)

Approach 2 : Greedy Algorithm

Here, two answers have been formulated for each test case. For greedy algorithm, first, the weights have been sorted in ascending order.

```
/*********************************START*********************************/

int cmpfunc (const void * a, const void * b)    //a utility
   function for using inbuilt function for quicksort from
   library
{
   return ( *(int*)a - *(int*)b );
}

qsort(w, n, sizeof(int), cmpfunc);  //sort weights in
   ascending order

/*********************************END*********************************/
```

The first answer comes from filling the knapsack with the maximum number of i weights starting from the lowest weights as well as heighest weights and taking the maximum value thus obtained under the condition that sum $<= W - \ln i$. (where i from 1 to n) (This had been interpreted from wikipedia)

```
/**********************************START*****************************/

   /************Traverse till a particular weight can't be
      added*********/
     for(i1=0;i1<n;i1++)      //front approach, ie,
        maximum no.of weights are added to the sack
     {
         sum1+=w[i1];     //add weight to the cumulative
            sum of weights
         if(sum1+log(i1+1)>W)     //if condition is not
            satisfied, then limit reached. subtract the
            last added weight
         {
             sum1-=w[i1];
             break;
         }
     }

     for(i2=n-1;i2>=0;i2--)       //bag approach ie,
        minimum no.of weights are added to the sack
     {
         sum2+=w[i2];     //add weight to the cumulative
            sum of weights
         if(sum2+log(n-i2)>W)      //if condition is not
            satisfied, then limit reached. subtract the
            last added weight
         {
             sum2-=w[i2];
             break;
         }
     }


     if(sum1>sum2)         //If method 1a gives larger sum
     {
         for(j=0;j<i1;j++)
             printf("   %d",w[j]);   //print out the used
                weights
         printf("\n\tObtained sum = %d",sum1);   //print
            max sum of weights
```

```
        }
        else         //if method 1b gives larger sum
        {
           for(j=n-1;j>i2;j--)
                printf("   %d",w[j]);   //print out the used
                    weights
            printf("\n\tObtained sum = %d",sum2);   //print
                out the maximum sum of weights
        }

/**********************************END*******************************/
```

The second answer is an extension of the first wherein the first i weights + later weights have been put into the knapsack such that all items upto a certain p may or may not have been used up to form the maximum sum under the condition that sum $<= W - \ln i$ (where i from 1 to n). This has been given a front approach as well as a back approach where both ascending as well as descending weight order has been considered (as in the first method) and the maximum weight taken into account and displayed. This method is found to be more accurate than the first greedy approach.

```
/**********************************START*****************************/

        /***********Traversing even if a weight in between
            cannot be added***********/
         for(i1=0;i1<n;i1++)      //front approach, ie,
             maximum no.of weights are added to the sack
        {
            if(sum1+w[i1]+log(count1+1)<=W) //as long as
                condition is satisfied
            {
                index1[count1]=i1;  //store index of the
                    accepted weight
                count1++;   //increment the no. of weights
                    accepted
                sum1+=w[i1];    //increment the sum
            }

        }

        for(i2=n-1;i2>=0;i2--)      //bag approach ie,
            minimum no.of weights are added to the sack
```

```
        {
            if(sum2+w[i2]+log(count2+1)<=W) //as long as
                condition is satisfied
            {
                index2[count2]=i2;  //store index of the
                    accepted weight
                count2++;          //increment the no. of
                    weights accepted
                sum2+=w[i2];      //increment the sum
            }
        }

         if(sum1>sum2)           //If method 1a gives larger
            sum
        {
            for(j=0;j<count1;j++)   //print out the accepted
                weights using method 2a
                 printf("   %d",w[index1[j]]);
            printf("\n\tObtained sum = %d",sum1);   //print
                the cumulative sum in method 2a
        }
        else         //If method 1a gives larger sum
        {
            for(j=0;j<count2;j++)
                 printf("   %d",w[index2[j]]);    //print out
                    the accepted weights using method 2b
            printf("\n\tObtained sum = %d",sum2);    //print
                the cumulative sum in method 2b
        }
/*************************************END*******************************/
```

Qn : How does the factor, $\ln m$, which penalises the use of larger number of items, affect your algorithm?

Ans. The factor -ln(m) affects both dynamic programming as well as greedy algorithm in such a way that the algorithm makes use of minimum number of weights in formulating the maximum sum of weights $<= W - \ln i$.

Dynamic Programming : Here, the algorithm tries to make use of the maximum weights to get the maximum sum under the given condition. Hence, when we print out the weights used, larger weights seem to be used while formulating the resultant sum.

Greedy Algorithm : Here, we had followed a front approach as well as a back approach, ie, using the weights arranged in descending order and ascending order in formulating the sum under the given condition, and then taking the maximum of the above two sums. It was found that in most of the cases , the descending order sum was preferred over the ascending order sum. This is because we are penalised with the quantity -ln(m) which penalises the use of larger number of weights and hence, since descending order approach makes use of lesser number of weights (since larger weights are put into the knapsack first), it is preferred more.

Qn : Estimate the number of operations required using direct recursion and for using the two methods above.

Ans. . It is quite obvious that the time complexity of the above two algorithms is much less when compared recursion which has an exponential time complexity

Dynamic Programming : O(nW)
Here n is the number of weights and W is the maximum weight as specified in the question.
This comes from the two for() loops used while making up the table K[][].

Greedy Algorithm : O(n+log(n)) which is more or less O(n)
O(log(n)) is for quicksort and O(n) is for traversing the array once in the for() loop for finding the weights used in getting the maximum possible sum

Recursion : When using recursion, the time complexity turns out to be exponential O(|E|). It also involves the overheads of repeated function calls.