Q1:

Given:

• There is an algorithm for multiplying two numbers with n digits each in O($n^{1.6}$) time. --- (1)

• The product of k numbers each with n digits has at most O(kn) digits. --(2)

a)

We know that the product of k numbers each with n digits has at most O(kn) digits. Now we have k/2 digits from A[1] to A[mid] and A[mid+1] to A[k]. The product from A[1] to A[mid] will have at most O(kn) digits, same will apply for A[mid+1] to A[k].

Now

$$A[1] \times \ldots \times A[k] = (A[1]x \ldots xA[mid])\, x\, (A[mid+1]x \ldots xA[k])$$

Let

$(A[1]x \ldots xA[mid])$ be - (3)

$(A[mid+1]x \ldots xA[k])$ be -(4)

By (1) we have (3) and (4) each with at max k*n digits, so the complexity to multiply them will be

$$O((n*k)^{1.6}) = O(n^{1.6} * k^{1.6})$$

Hence, proved

b)

We can use the divide and conquer approach to solve this problem.

We shall divide the task of $A[1] \times \ldots \times A[k]$ into two subtasks

$(A[1]x \ldots xA[mid])$ *and* $(A[mid+1]x \ldots xA[k])$

Then we compute the above two halves by dividing again recursively.

The base case will be when k is equal to 2, then we simply multiply the two numbers and return.

The recurrence relation we will get is

$$T(k) = 2 * T(k/2) + O(n^{1.6} * k^{1.6})$$

Using masters theorem: As the recurrence is over k, $n^{1.6}$ is a constant

a=2, b=2, k>>n

$log_2 2 = 1$ *and* $c = 1.6$

$c > 1$

So we have time complexity as $O(n^{1.6} * k^{1.6})$

Algorithm:

```
multiply(Arr,1,k){
        if (k == 2)
        {
                return Arr[1] * Arr[0]
        }
        left  = multiply(Arr,1,k/2)
        right = multiply(Arr,k/2+1,k)

        return left*right
}
```

Q2)
a)
We will take the arrayA[low to high] and find the median in the range low..high range by: ceil ((low+high)/2). Then if the median is not in the array, it is the missing element. Otherwise, we divide the array into two halves. One half, say P, will have elements less than or equal to the median and the other half, say Q, will have elements greater than the median. Now we're only going to go to one of these two arrays: if the size of P is less than median+1, then we go into P otherwise Q.
This solution is correct because every time we are dividing the array(even though it's unsorted) by the median. So effectively, we have checked for all possible values in the range 0 to n. Since only one number is missing from 0 to n and we are checking for all the values, we will eventually end up with a median value that is not in the array.
The recursion for this algorithm will be:
$T(n) = T(n/2) + O(n)$

Using master theorem we can prove that the running time of T(n) will be O(n)

Pseudocode:

```
findmissingnumber(array A, int i, int j):
        find median m using ceil ( (low+high)/2 )
        if m not in A: return m
        else divide A into P and Q
                P: elements <= m
                Q: elements > m
        if size(P) < m+1:
                findmissingnumber(P,startP,endP)
        else:
                findmissingnumber(Q,startQ,endQ)
```

b)We first take the array, have two variables of count_0 and count_1.
Now we list down all the elements in their binary representation and start from the 0th bit till the kth bit.
At the ith bit.
Let's have two arrays to keep elements with that bit 0 and with bit 1.
Now the one with lesser elements is the group that has the missing bit.
At last, we will collect all missing bits and get the missing number.

```
k = logn
A <- array
zero_array
one_array
missingnum[len(A)]
from j = 0 to k:
        for i=0 to len(A):
                set = BIT-LOOKUP(i,j)
                if (set == 0):
                        zero_array.append(A[i])
                else:
                        one_array.append(A[i])

        if len(zero_array) < len(one_array):
```

```
            A = zero_array
            missingnum[j] = 0
        else:
            A = one_array
            missingnum[j] = 0

missing_number = conv_to_num(missingnum)
```

Runtime analysis:
T(n) = T(n/2) + O(n)
The above can be visualized as every time we calculate the word count we reject one half of the array and now do processing on the other half.

By masters theorem, we get the time complexity of T(n) as O(n)

Q3. DP
L[]-> array of length of words
Max chars in a line = P
Spaces at the end of line if words i to j are in same line = P- {∑ k=i to j (L[k])} - (j-i) = s(i,j). We need to minimise the square of these spaces for each line.
S[i,j] -> n x n array of square of spaces if words from L[i] to L[j] are put in one line. It is made -1 in case those words can't be accommodated in a single line.
C[] -> array of length n whose k'th element has the number of total minimised spaces for words 1 to k. Our answer will be in C[n-1].

```
for i=0 to n-1:
        S[i][i]=(P-L[i-1])^2
        for j=i+1 to n-1:
            s=-1;
            if (S[i][j-1]>=0) s=sqrt(S[i][j-1])-L[j-1]-1
            if(s>=0)
                S[i,j]= s^2
            else
                S[i,j]=-1
        endfor
endfor

C[0]=0;
for k=1 to n-1:
        smallest=inf;
        flag=0;
        for z=1 to k:
            if(S[z,k]!=-1 && C[z-1]!=-1)
                smallest= min(C[z-1]+S[z,k],min)
                flag=1;
        if(flag ==1) C[k]=smallest
        else C[k]=-1;
```

return C[n-1]

The complexity is $O(n^2)$ because the first loop has a nested loop within an outer loop, and inner runs for n-i-1 iterations and outer runs for n iterations. In the inner loop, we do a constant number of operations. Hence, order for first loop = $O(n^2)$. Similarly, for the second loop, it has a loop nested within another, with the outer loop running for n iteration and inner loop running for k iterations. In the inner loop, we do a constant number of operations. Hence the order of second loop = $O(n^2)$.
Therefore, the order of the algorithm is $O(n^2)$.

Q4. Given: Questions 1...n with each having pi points and if i attempt i'th question, then I can't attempt the next fi question.
P[] -> array of points
F[]-> array of questions to skip
array[]-> array initialized to zero to get the max points gotten till i'th ques.
Path[]-> array to tell which questions were attempted to get the number of points in the array above.

```
array[0]=p[0]
array[f0+1]=array[0]+p[f0+1]
for i=1:n-1
        if(array[i-1]-p[i-1]==0)
                path[i]=i
        else if(array[i-1]-p[i-1]+p[i]>array[i])
                path[i]=i-1
        array[i] = max(array[i-1]-p[i-1]+p[i],array[i])
        if(i+fi+1<n)
                if(array[i]+p[i+fi+1]> array[i+fi+1])
                        path[i+fi+1]=i
                array[i+fi+1]=max(array[i]+p[i+fi+1], array[i+fi+1])
endfor
larg=-1
largid=0;
for i=0:n-1
        if(larg<array[i]) largid=i;
        larg=max(larg,array[i])
endfor
quesToAttempt(largid);

def quesToAttempt(int id):
        if(id==path[id])
                print(id+1)                     //adding 1 as ques i corresponds to array index i-1
        else
                print(id+1)                     //adding 1 as ques i corresponds to array index i-1
                quesToAttempt(path[id]);
```

First loop = O(n) as n iterations and each iteration has constant number of operations

finding max of array "array" -> O(n) by loop
find whole path from path[i]-> worst case O(n) as traversing all questions
So algo = O(n) + O(n) + O(n) = O(n)

Q5) I couldn't come up with O(mk) dp, what is written below is $O(m^2k)$

a)
DP[i][j] = optimum answer for books (1 … i) and j scribes.

The required answer = dp[m][k]
Where m is the number of books we have.

b)

Let book[i] be the number of pages in i th book.

Let sumBooks[i] = sum of all books from 1 to i

DP[i][1] = sumBooks[i]

DP[1][i] = book[1]

DP[i][j] = 0;
For p = (1 ... n):
    DP[i][j] = min(DP[i][j], max (DP[k][j - 1], sumBooks[i] - sumBooks[p]))

c)
sumBooks[0] = 0
For i = 1...m
    sumBooks[i] = sumBooks[i - 1] + book[i]
for i = 1 ... k
    DP[1][i] = Books[1]

for i = 1...m
    DP[i][1] = sumBooks[i]
    for j = 2...k
        for p = 1 … m
            DP[i][j] = min(DP[i][j], max (DP[k][j - 1], sumBooks[i] - sumBooks[p]))

return DP[m][k]

d)

The first 2 loops are O(m) and O(k) respectively, the final loop is $O(m * k * m) = O(m^2k)$

Thus $T(n) = O(m) + O(k) + O(m^2k) = O(m^2k)$

e)

Assume that all subproblems less than DP[i][j], have already been solved with an optimal solution.

By iterating over p as mentioned above, we are effectively iterating over the possible assignments to the j^{th} scribe. We then assume that the rest of the books were allocated to the remaining (j - 1) scribes, and that since we had already found the optimum solution for this.

Assume that we haven't found the best solution.

Consider the optimal assignment, let j^{th} scribe has books from q to i.
Considering the case when p = q, in our algorithm.

So only 2 Cases are possible

1) j^{th} scribe had most work, but the algorithm found more for 1 of the other scribes, this is not possible as we had assumed that we had found the optimal answer to the subproblem.
2) 1 of the scribes in the first (j - 1) scribes had most work, but our algorithm said j^{th} scribe had most work, this is not possible as both our algorithm and the optimal solution have given the same books to the j^{th}, hence his workload must be same in both cases.

Since we reach a contradiction, hence our assumption that we haven't found the best solution is false.
Hence proved.