1: Probability of K heads

inputs from the question: n = number of coins

k = number of heads

P_i = set of probabilities for n coins to flip head at i^{th} index

$+^{ion}$ and X^{ion} of two numbers in the interval [0,1] takes O(1) time

assumptions:

$1 \leq i \leq n$, $0 \leq k \leq n$.

goal: setup a recurrence equation to compute number of heads at every stage (tosses) leading to precisely k heads at O(n^2) complexity.

Algorithm: :

The basis of algorithm is derived from the below recurrence principle -

1.if $n-1$ coins have k heads, then n^{th} coin should be a tail. given that the probability of i^{th} coin having a head is P_i, the probability for tails can be given as $1 - P_i$.

2.if $n-1$ coins have k-1 heads, then n^{th} coin should be a head. we already know that the probability of i^{th} coin having a head is P_i.

from above two principles we can have the total probability equaation as -

$$P(n,k) = P_n.P(n-1,k-1) + (1-P_n).P(n-1,k)$$

here $P(n,k)$ will return probability of k heads for n coin flips.

the base case let's way when reduced to 1 coin and no heads (as k can be 0) will return us a tail on single flip and we can build the case for any heads more than 0 after that.

Correctness: : The algorithm starts with max values of n and k, and backtracks the number of heads required based on the requirement. every single iteration adds a new recurrence term for computing based on whether a head or tail is required at the n^{th} flip. The proof can be done by induction when let's say for 1 coin and we need 1 head, gives us a 100% probability as againsst 1 tail gives us a 0 probability. similarly with max value of k=n, we will have every flip pushing for a head while let's say k = n-1 number of heads, will give us 1 tail in first iteration and all heads for rest of n-1 iterations.

Running time: : The recurrence essentially requires to fill up memory cells with n rows and k columns. This transforms to a nxk cell with worst case being $k = n$ heads, meaning all flips returning a head. Hence, the complexity amounts to nxk computational loops and with $k = n$, it gives us $O(n^2)$

2: Count Number of Parsings

Algorithm: : Given an input String Str of length N, idea is to create a new empty array of N elements, say $A[0...n-1]$. We will traverse the input string recursively to come up with different sub$_stringsthatcanbegeneratedwithdifferentsub_stringlengthsandcountthenumberofpossibilities.$

string_identification(Str)

$count = 0$

for i in $(0 \ldots N)$ //length of string

$S = Str[0:i]$

if S belongs to dictionary

if $str - S$ is not $null$ // checking if the string is empty

$count += 1$

else

$count +=$string_identification($str - S$)

end //if

end //if

end //for

end proc //end procedure

The procedure string_identification tries to find the possible ways to break up the string Str. count is a variable which keeps track of all possibilities. Since we know that the maximum length of a word can be L, we try all the possible combination break ups of this string from the beginning of Str. If the sub-string S is present in the dictionary, there can be two further cases. $strs$ is removed from the original string. Unless we reach the end of the string, this will count as one possibility to this chain of recursions. Otherwise, we need to apply the recursive possibilities for $StrS$ and add it to the current value of count.

Correctness: : If sub-string S is not present in the dictionary, we dont need to consider any further partitions with S removed from Str because if we did, this break-up will result in a word S which was not part of the input string. Therefore, we only need to move forward as at least S is valid. Also $StrS = null$ means that we have hit the end of the string with the last valid word S. This will contribute 1 to the possibility counter $count$. In the other case, when $StrS$ is non-empty, we try to find the possible break-ups of the remaining part of Str -¿ $Str - S$ and add it to the current $count$. The final value of the $count$ becomes the total possible allowed break-ups for the given Str.

Running time: : The worst case is when dictionary has split for all the sub-strings within given Str. In that case, the initial call with $len(str) = N$, will recursively call itself with the the sub-string. In addition, every call there is a constant time operation of finding the sub-string involved. The recurrence relation for the algorithm can be written as,

$$T(N) = T(N1) + T(N2) + \; + T(NL) + 1$$

$$T(N)LT(N1) + 1$$

Solving this recurrence relation gives the complexity $O(L^N)$.

3: the ill prepared burglar

let's say Burglar's sack can hold items upto size of 180 and consider a set of Item Sizes and their respective Values:

item- A size- 60 value- 180 value/size- 3

item- B size- 36 value- 270 value/size- 7.5

item- C size- 60 value- 360 value/size- 6

item- D size- 72 value- 540 value/size- 7.5

item- E size- 108 value- 648 value/size- 6

item- F size- 120 value- 720 value/size- 6

(a) as a counter example for first case, he first picks up "F" as it has the highest value. Now, the burglar's sack can hold item of size upto $180 - 120 = 60$, so, the next best valued item of size less than or equal to 60 is "C". Now the burglar has no space left in the sack and he was able to collect items worth $720 + 360 = 1080$. Instead, if the burglar was well prepared, he could have collected "D" and "E" which are of size 180 and worth $540 + 648 = 1188$ which is more than what he collected.

(b) For the same example mentioned above, picking Items in decreasing ratio will give the burglar the "B","D" and "C" which will be a total of size $36 + 72 + 60 = 168$ and worth $270 + 540 + 360 = 1170$. However, this is not the optimum solution for this example. Instead, the burglar could have collected "D" and "E" which are of size 180 and worth $540 + 648 = 1188$ which is more than what he collected in the first place.

(c) Algorithm: :

Variables :

v$_i$: $arrayofitemvalues$

s$_i$: $arrayofitemsize$

n : total number of items

S : size of the locker

The algorithm involves recursively inserting item values into a n x S matrix:

```
for q from 0 to S do:
   Initialize matrix[0, q] := 0
end //for
for p from 1 to n do:
   for q from 0 to S do:
      if a[p] > q then:
         matrix[p, q] := matrix[p-1, q]
      else
         matrix[p, q] := max(matrix[p-1, q], matrix[p-1, q-a[p]] + v[p])
     end //end if
   end //for
```

end //for

matrix[a] will return the list of optimal values.

Correctness: : In every loop, we check if the item is to be added to the locker or not. The max value for all the n items can be either the maximum value so far, without considering the current item i, or the maximum value of current item plus value of item for the remaining size and value so far, without considering the current item i.If the size is greater than the locker size, we do not consider it. Hence, the algorithm will always give the maximum colletion value.

Running time: : For each item in the collection of size n, the inner loop runs for each possible value S. Therefore, the complexity of this algorithm is $O(nS)$

(d) the answer is NO as there is no n term involved in the space complexity.

4: Central nodes in a tree

Algorithm: :

1. This is a tree clustering problem as we want to divide the tree into k parts in such a way that we minimize the $cost_S(v)$ for every $v \in T$ with a Set of marked vertices m.

2. By above intuition, we can say that the max size MAX of a cluster can be $\frac{n}{k}$ and min size MIN can be 1 i.e. $1 <= size(k) <= \frac{n}{k}$

3. We have a function $mark(node, removeCount, nodeCount)$ which returns true if we can remove $removeCount$ elements from subtree of $node$ keeping $nodeCount$ vertexes connected to it (to satisfy the condition mentioned in the step 2).

4. Base Case of this algorithm will be for leaves.

Base Case :

$mark(leafNode, 1, 0)$ is true only if $MAX >= 1$ and $MIN <= 1$

$mark(leafNode, 0, 1)$ is always true

5. The function $mark(node, removeCount, nodeCount)$ is defined as

for leftSubtreeClusters from 0 *to* k do:
    for rightSubtreeClusters from 0 *to* K do:
        for nodeCountLeft from 0 *to* $leftSubtreeSize$ do:
            for nodeCountRight from 0 *to* $rightSubtreeSize$ do:(
                if $mark(leftChild,\ leftSubtreeClusters,\ nodeCountLeft) == true\ and$
                    $mark(rightChild,\ rightSubtreeClusters,\ nodeCountRight) == true\ then:$

$mark(node, leftSubtreeClusters + rightSubtreeClusters, nodeCountLeft + nodeCountRight) =$

true)

endAll // end for all loops

for marks from 0 *to* $K - 1$ do:

for count from 0 *to* $nodeSubtreeSize$ do

if $mark(node, marks, nodeSubtreeSize) == trueandMIN <= count <= MAXthen$

$mark(node, marks + 1, 0)$

endAll //end for all loops

Correctness: The observation about this algorithm is that if it works for MIN and MAX then it will also work for MIN and $MAX + 1$. In first set of for loops, combines all the states of child nodes to compute state of a parent node. Then in the next set of loops, it separates the node with its parents to form the cluster. If mark(rootNode, k, 0) is true then it means it is possible to find k clusters that satisfy the step 2 condition.

Running time: In the first set of for loops, the algorithm loops n*n*k*k times which is the highest complexity component of this algorithm. Thus complexity of the algorithm is $O(n^2K^2)$.

Note : discussed and referred solution with Madhur from stackoverflow.

5: faster LIS

(a) Lets take an example A = [6, 4, 12, 7, 5, 8, 11, 10]. The index of array B will start from 1 as the minimum LIS can be 1. We start with i = 7 with A[i] = 10 so B[1] = 10 as the only LIS of length 1 is A[7] as of now. Next i = 6 with A[i] = 11. Since LIS[6] is also 1, we overwrite the value in B[1] as 11. In essence, I can only replace the value in B[1], when the current A[i] is larger than B[1] otherwise I would have had a LIS of length 2. Therefore, for each index j of array B we find LIS[i] of length j and place the maximum of A[i]s. In this way, the first element of B will be the largest value of A as it has LIS of length 1 and is largest among any other such LIS. The next element has to be smaller than B[1] otherwise it cannot make an LIS of length 2. The second element of B is largest element that has the LIS length 2. The third element of B cannot be larger than B[2] otherwise we will not have a LIS of length 3. In this manner, we can say that the B[j + 1] cannot be larger than B[j]. Since the entries in A are all distinct and B is made up from the elements in A, B is a strictly decreasing array. For our example, the array will be B = [12, 8, 7, 6]

(b)

Algorithm:
1. Start with last element of A.
2. Loop over i from $n - 1$ *to* 0 - set j = $min(A[i])$; min function finds out the minimum element of A closest to A[i].

check that if $j = -1$ then

put $A[i]$ in the end of B

else

set $B[j] = A[i]$ (i.e. A[i] is starting point of a larger LIS)

3. Return the length of array B

Correctness: Every element in B is biggest of LIS of $length$ = index of B, which means that there is an LIS with that length. So $B[last]$ = biggest LIS of $A[]$. Thus, length of B is the largest LIS length in A.

Running time: The algorithm loops over n and for each cycle performs the binary search which is $logn$. So, the final complexity turns out to be $O(nlogn)$.

6: maximizing happiness

(a)

	Gift A	**Gift B**	**Gift C**	**Gift D**
Child 1	20	15	10	1
Child 2	500000	30	10	20
Child 3	600000	700000	40	10
Child 4	600000	800000	900000	10

In the greedy strategy, Santa will give $Child$ 1 his most desired gift i.e $Gift\ A$. Subsequently, he will give $Child$ 2 -> $Gift\ B$, $Child$ 3 -> $Gift\ C$ and $Child$ 4 -> $Gift\ D$ based on their most desired gifts.
This allocation gives a total happiness factor of $20 + 30 + 40 + 10 = 100 \ldots (1)$
However, this is not the best way to maximize happiness. Consider the below allocation of gifts :
$Child$ 1 -> $Gift\ D$, $Child$ 2 -> $Gift\ A$, $Child$ 3 -> $Gift\ B$ and $Child$ 4 -> $Gift\ C$
In this case, the total happiness is $1 + 500000 + 700000 + 900000 = 2100001 \ldots (2)$
Compariing $Case$ (1) $and\ Case$ (2), we can see that $Case$ (2) is approximately 21000 better than $Case$ (1) i.e $Greedy\ Method$

(b)

discussed solution with Sagar.