1: Probability of k heads

Approach: To get the solution for this problem we can use dynamic programming approach. Let X_i = random variable for the event of getting head on tossing ith coin, where $1 <= i <= n$. Here Probability($\sum_{(i=1)}^{n} X_i = k$) = Probability($\sum_{(i=1)}^{(n-1)} X_i = k-1$ AND $X_n = 1$) + Probability($\sum_{(i=1)}^{(n-1)} X_i = k$ AND $X_n = 0$) = p_nProbability($\sum_{(i=1)}^{(n-1)} X_i = k-1$ + q_nProbability($\sum_{(i=1)}^{(n-1)} X_i = k$)

Algorithm: The step by step implementation of this problem can be done as:
Assume: An array 'Prob' exists that has the probability of heads of each coin. Let 'k' has no. of desired head and 'n' has total coins.
$PROB\_FUNC(Prob, k, n)$
1. If Prob is NULL or length of Prob == 0, stop and return null.
2. Let a array be MAIN with rows = n+1, columns = k+1. Initialize the array with 1, as we will start with probability of atleast a single coin. Thus MAIN[0][0]=1.
3. Loop i for (n+1) times and nested loop j for (k+1) times.
4. Stop if j is greater than i, else calculate value as decsribed in above approach.
5. Stop if iterations end. return MAIN[n][k].

As we can observe the algo runs in $O(n^2)$ time complexity as 2 'for' loops exist that run [n]*[k] times. Worst Case $\implies$ k=n

Running time: Here Time Complexity is $O(n^2)$.

2: Count Number of Parsings

Assume: We have a collection of all words stored in dictionary. 'count' variable is initialized to zero.

Algorithm: We can create a function that performs tasks according to the given pseudocode:
MEMCOUNT_PARSER(string-input, len-input, dictionary, count):
1. Check if dictionary contains the string-input, stop and return string-input.
2. Check if mem contains string input, stop and return string-input.
3. For i <= len-input :
4. Find substring of string-input from 0 to ith index and prefix = input subtring 0 to i
5. Check if dictionary contains prefix, Call MEMCOUNT_PARSER with input = string input i to len-input.
6. String suffix = input substring i to length of word
7. If suffix is not Null, count ++
8. Put (string-input as key and prefix and suffix as value in mem array) **Running time:** The time complexity of the algorithm is $O(NL)$

3: The Ill - Prepared Burglar

(a) Example when strategy is not optimal
When the size of expensive item is equal to the size of the sack, the burglar will be able to take only one item with him.
Otherwise he could have taken several other small size items whose indivisual value would have been small but sum of total value would have been larger than the value of the most expensive item.
Example:
Item A,B,C have values 10,9,8 and let sizes be 10,5,5 respectively. Let the size of sack be 10.
Thus Case 1:Burglar takes most expensive item A. Total value he takes is 10.Sack is also full
Case 2: Burglar takes item B and C.Total value = 17 which is more than 10. Sack is full.
Thus this strategy fails.

(b) Let the following items exist:

Size	**Value**	**Value/Size**
A- 10	60	6
B- 20	100	5
C- 30	120	4

Let the sack size be 50. Now burglar starts with Item in decreasing order of Value/Size.
He picks item A then B, now the space is not left for item C as he has already picked 30/50 and C has 30 weight. Thus total value of taken items is 160.
If he would have taken Item B and then item C then the total value of goods would have been 220 (greater than 160 as per previous case) and sack would have been full as 20+30=50.
Thus it shows that the burglar's strategy is not optimal.

(c) We can use the following step by step algorithm:
Assumption: let S=total size, v=value of each item, s=size of each item, n=total no. of items.
Algorithm: Most_Valued_Collection (S, v_array, s_array, n)

(a) Let MAIN be an array. Dimensions = n + 1 x S + 1. Initialize the array by 0.
(b) For i in range (n+1) and another nested loop for s in S+1
(c) If i is equal to 0 OR s is equal to 0, initialize MAIN[i][s]=0
(d) else if size [i-1]<=s, set MAIN[i][s] as max value of (v_array[i-1] + MAIN[i-1][s-s_array[i-1]], MAIN[i-1][s])
(e) MAIN[i][s]=MAIN[i-1][s]
(f) Stop after completing loops, return MAIN[n,S] for max of total value.

Running time: The runtime of the algorithm as observed above is $O(nS)$.

(d)

NO. No such algorithm exists for this particular situation that runs in time polynomial in the length of input $nlogS$. However, the runtime of the algorithm as observed above is $O(nS)$.

4: Central nodes in trees

Step by step algorithm may be used for this particular problem:
Goal: Our primary goal is to minimize $cost_s(v)$ for all $v \epsilon T$.

Algorithm:
PART 1
We need to make a dynamic state assignment with three variables as follows:
let there be a function_MAIN
Some variables
index_given_node=index
no._of_nodes_that_can_be_colored_in_subtree=k
distance_to_closest_marked_ancestor=d
PART 2
Calculation of the optimal result for subsequent loops can be done as:
1. If current node is the left node
2. If k is greater than 0. (If marking of the node in above step is possible then the distance will be equal to zero.)
3. stop and return 0.
4. stop and return d.
5. result is equal to infinity.
6. if the current node is not marked till now, and can be marked, mark it.
7. if (d is not equal to 0) and (k is greater than 0)
8. result = minimum (result, function (index,k-1,0)
9. for a variable t equal to 0 to k:
10. allocate t mark nodes to left_child and k-t mark nodes to right_child
11. result = minimum $(result, function(indexofleft\_child), t, d+1)$ PLUS
$fuction(indexofright\_child, k, d+1) + d)$
12. return result
Thus we will get the total of distances of node plus its descendents to the closest marked node.
PART 3
Function(0, // root index of tree, k, infinity)
This gives the best optimal results.

Running time: The result of each iteration can be saved in a memoized table to get the program running with time complexity $O(n^2k^2)$ using dynamic programming method.

5: Faster LIS

(a) While updating the new array, we can follow three conditions to ensure that the array has strictly decreasing sequence:
Take the righmost element $A[i]$ in original array and append it in a new array B.
1. For element A[i] in original array, if it is greater than the largest element in new array B, then replace it with element at index 0 of the new array B. This step makes sure that B has the largest element in the beginning.
2. Take $A[i]$ and check if it is smaller than the smallest element in new array B, then add that element in B.Thus this step helps to ensure that decreasing order is present in elements of B.

3. Take $A[i]$ and check if it is in between any 2 numbers (neither largest nor smallest) in the new array B. Find the largest element in B that is smaller than A[i]. Replace the element in new array B.
Thus we can find out that the largest number is placed at the 0th position in new array B and array B maintains a strictly decreasing sequence.

(b) Algorithm:

$FAST\_LIS$(A[], Array_A_size)
1. Check Array_A_size is equal to 1. Stop and return 1.
2. Let B[0]=A[0]. Let L_length=1. L_length is the size/length of Longest Increasing Subsequence. Current LIS =1, So L_length=1.
3. Loop i=Array_A_size, loop till i is greater than 0.
4. If A[i] is greater than B[0], then make B[0]=A[i]
5. If A[i] is less than B[L_length - 1], then make B[L_length ++] = A[i]
6. Find the index of value that is largest in B, but smaller than new value. Replace new value at that position.
7. Stop after loops ends, return L_length.

Running time: The time complexity of the algorithm is $O(nlogn)$.
Loop iterates through items in Array A, that makes the time complexity n, Step 6 uses binary search to find index of element. Hence $O(nlogn)$.

6: Maximizing Happiness

(a)

EXAMPLE SETTING:
Lets take the following example to analyse the case:

___	**giftA**	**giftB**	**giftC**	**giftD**
C 0	21	4	7	10
C 1	60000	8	7	5
C 2	6000	6000	140	12
C 3	10000	21000	400000	17

Best assignment:
C0 = giftD = 10
C1 = giftA = 60000
C2 = giftB = 6000
C3 = giftC = 400000
Happiness = 10 + 60000 + 6000 + 400000 = 466010.
Greedy distribution:
C0 = giftA = 21
C1 = giftB = 8
C2 = giftC = 140

C3 = giftD = 17
Happiness = 21 + 8 + 140 + 17 = 186
Ratio of greedy to best assignment = 466010/186 = 2505 times worst than best assignment.

(b) and **BONUS QUESTION** (Total 16 points)
General Working of Local Search Algorithm:
The Local Search Algorithm swaps two values if the happiness parameter of children is less than maximum. It stops its swapping function when there is no room for further increase in the happiness parameter of two children or when the children have been assigned correct gifts according to their wish and summation of happiness is maximum as expected. Thus swapping stops at this point.
OPTIMAL SOLUTION: We want to reach the stage when each child has received the gift as per his/her wish.
Assumption: Optimal assignment is acheived = $Child_{ij}$.
Now, If we do a local search and we dont get this particular optimal solution, then that particular search will have some assignments such that Child_i will get other gift m and the second child Child_j will get some n gift.
Thus here in this case it is possible that addition of $Child_{im}$ and $Child_{jn}$ is ¿ than optimal $Child_{ij}$.
In worst case scenario, it is double times or twice as bad as that optimal assignment and the similar case can occur for more than one arrangements. Thus we can conclude that it is atleast $\frac{1}{2}$ of the optimal solution. The method applied for triplet case is similar to previous one.
Let us assume an optimal assignments = $Child_{ik}$ and $Child_{jl}$ are 2 assignments. Suppose local search doesn't has this assignment then it will contain in the worst case, $Child_{il}$,$Child_{jz}$ and $Child_{hl}$.
This can happen in $3C_2$ or total of six combinations. At this point, by applying a similar logic, one can conclude that it is atleast $\frac{2}{3}$ the value of optimum solution.