1: Probability of k heads

Algorithm: Let's denote the probabilty of finding j heads in i tosses as $M[i,j]$.In the general setting, for the i^{th} toss, there can be two cases: we don't want the i^{th} to be a head (i.e. it's a tail) or we want it to be a head. For the first case, we can get the probability $M[i,j]$, as $(1-p_i)\cdot M[i-1,j]$ and for the second case we can get the same as $p_i \cdot M[i-1,j-1]$. Therefore, the general can written as

$$M[i,j] = (1-p_i)\cdot M[i-1,j] + p_i \cdot M[i-1,j-1]$$

Algorithm 1 Finding probability of k heads

```
procedure $find\_prob(n,k)$
    if $n < k$ then
        return 0
    end if
    Declare Array $M[0\dots n, 0\dots n]$
    for $i = 0,1,\dots n$ do
        for $j = 0,1,\dots n$ do
            if $i = 0$ then
                $M[i,j] = 0$
            else if $i < j$ then
                $M[i,j] = 0$
            else if $j = 0$ then
                if i=1 then
                    $M[i,j] = (1-p_1)$
                else
                    $M[i,j] = M[i-1,j]\cdot(1-p_i)$
                end if
            else if $i = 1$ && $j = 1$ then
                $M[i,j] = p_1$
            else
                $M[i,j] = (1-p_i)\cdot M[i-1,j] + p_i \cdot M[i-1,j-1]$
            end if
        end for
    end for
    return $M[n,k]$
end procedure
```

Let's assume that there is a two dimensional matrix M with rows $0,1,\dots,n$ and columns $0,1,\dots,n$ where each element denoting the value $M[i,j]$ described as above. In addition to the general case, we need to have some special cases for our evaluations. The first row of the matrix ($i = 0$) is 0 as we are not tossing any coin. For the case where $j > i$, the entry needs to be zero as number of heads cannot exceed number of tosses. When $i = 1, j = 1$, the entry in the matrix is p_1. Finally, all the entries with $j = 0$ will look like $((1-p_1)\cdot(1-p_2)\cdot(1-p_3)\cdots(1-p_i))$ except when $i = 0$. This entry can therefore be written as $(1-p_i)\cdot M[i-1,0]$.

All other entries can be evaluated using the general case. The pseudocode of the algorithm is given in Algorithm 1.

Correctness: For the case, when $j = 0, i = 1$, we are explicitly assigning the value of the probability, for other cases $j = 0, i = 1, \dots, n$, the probability is the product of the previously calculated such probability and the current value of $(1 - p_i)$. The expression of the general case represents that the current toss can be a head or not. If it is head, the current probability is the product of p_i and the probability of $j - 1$ heads in $i - 1$ tosses. If it is tail, the current probability is the product of $(1 - p_i)$ and the probability of j heads in $i - 1$ tosses. Since the algorithm fills up the array one row at a time and the values to be reused are picked up from the previous array only, we will never have a invalid value of a probability.

Running time: The algorithm effectively runs for two nested loops of size $n + 1$ each. In each iteration, we are either assigning the element to a fixed value or performing a constant time operation (product). Also, each iteration only reuses a previously calculated value along with the given probabilities. In essence, the algorithm runs for the nested loops while performing constant time operation in each iteration. Therefore the runtime of the algorithm is $O((n + 1)^2) = O(n^2)$.

2: Count number of parsings

Algorithm:

Algorithm 2 Finding words in a string

```
 1: procedure $break\_str(str)$
 2:     $pc = 0$
 3:     for $i = 0, \dots, min(L, len(str))$ do
 4:         $s = str[0 : i]$
 5:         if $s \in dictionary$ then
 6:             if $str - s = \phi$ then
 7:                 $pc \mathrel{+}= 1$
 8:             else
 9:                 $pc \mathrel{+}= break\_str(str - s)$
10:             end if
11:         end if
12:     end for
13:     return $pc$
14: end procedure
```

The pseudocode for the algorithm is given in Algorithm 2. The procedure *break_str* tries to find the possible ways to break up the string str. pc is a counter that counts these possibilities. Since we know that the maximum length of a word can be L we try all the possible combinations break up of this string starting with the first character of str. This is the purpose of the for loop and these break-ups are stored in s. If s is present in the dictionary, there can be two further cases. $str - s$ represents removing the s from str in the same order. For example, if $s = abc$ and $str = abcdef$, then $str - s = def$. If $str - s$ is empty, we have reached the end of the string and this will count as one possiblity to this chain of recursions. Otherwise, we need to find the number of break-up possibilities for $str - s$ and add it to the current value of pc. That's why we recursively call $break\_str$ with $str - s$. The value of pc is returned in the end, which denotes the number of possiblities for the current str given the dictionary.

Correctness: If s is not in the dictionary, we don't 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 the intention of the typer. Therefore, we only need to move forward when atleast s is valid. Also $str - s = \phi$ means that we have hit the end of the string with the last valid word s. This possible and valid break-up will contribute 1 to the pc. In the other case, when $str - s$ is non-empty, we try to find the possible break-ups of the remaining part of str and add it to the current pc. The final value of the pc becomes the total possible allowed break-ups for str.

Running time: The worst case for this algorithm will be encountered when all poosible break-ups are allowed by the dictionary. In that case, the intial call with $len(str) = N$, will recursively call itself with the the sub-strings of length $N-1, N-2, \ldots, N-L$. In addition in every call there is a constant time operation of finding the sub-string is involved. The recurrence relation for the algorithm can be written as,

$$T(N) = T(N-1) + T(N-2) + \cdots + T(N-L) + 1$$
$$T(N) \leq L \cdot T(N-1) + 1$$

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

3: The Ill-prepared Burglar

(a) Let's say the size of the burglar's bag is 15. Next, assume that there are three items in the house, $S_1 : 10, 12$, $S_2 : 12, 15$, and $S_3 : 5, 4$, where the first number denotes the size of the item and second number denotes its value. Being greedy, the burglar will pick up the item S_2 as it can fit his bag and has the highest value. So greedy algorithm will let him get away with value 15. But if he would have chosen item S_1 and S_3 as they both can fit in his bag, he could have taken items of total value $12 + 4 = 16$.

(b) Let's say that the size of the burglar's bag is 50 and there are three items in the house, $S_1 : 30, 30$, $S_2 : 25, 24$, and $S_3 : 25, 23$, where the first number denotes the size of the item and second number denotes its value. The value to size ratio for the three items is $S_1 : 30/30 = 1; S_2 : 24/25 = 0.96; S_3 : 23/25 = 0.92$. The burglar will pick S_1 (as it has highest value/size ratio) and will see that there is no space left for any more item. So the value he takes away is 30. But if he would have picked items S_1 and S_2 (both can fit in his bag), he would have taken items of total value 47.

(c) Algorithm: : Let's call the optimal solution for the problem as $COL(n, S)$. Let's say at some step, I select the n^{th} item. The problem will now become $COL(n-1, S-s_n)$. On the other hand, if I reject the n^{th} item, the problem becomes $COL(n-1, S)$. This gives the insight about the subproblem. At each step, I need to know the best possible solution with the first $n-1$ items with the available size of $S - s_n$ or S. Let's call this variable size s.

An item will be rejected at some step i because it's size is bigger than the available weight. Otherwise, there will be two cases: we don't select that item and check for the value of

$COL(i-1, S)$ or we select that item and check for the value of $v_i + COL(i-1, S-s_i)$. This gives the idea about a two-dimensional matrix, M, with rows from $n, n-1, \ldots, 0$ and columns $0, 1, \ldots, S$, where each element $M(x, y)$ will denote the most valuable collection ($COL(x, y)$) made from the subset of items $0, 1, \ldots, x$ and their size is bounded by y. The bottom-most row and the left-most column of the matrix can only have zeros, as either $n = 0$ or $S = 0$. The algorithm will have two nested loops with the outer loop iterating over $i = 1, 2, \ldots, n$ and the inner loop iterating over $0, 1, \ldots, S$. At each set of iteration, we will populate some entry in the matrix using the previously calculated values and the result will be the entry $M(n, S)$. The pseudocode for the algorithm is presented in Algorithm 3

Algorithm 3 Most Valuable Items

```
procedure $find\_items(n, S)$
    Declare Array $M[n \ldots 0, 0 \ldots S]$
    Initialize $M[0, :] = 0$ and $M[:, 0] = 0$
    for $i = 1, 2, \ldots, n$ do
        for $s = 0, 1, \ldots, S$ do
            if $s < s_i$ then
                $M[i, s] = M[i-1, s]$
            else
                $M[i, s] = max(M[i-1, s], v_i + M[i-1, s-s_i])$
            end if
        end for
    end for
    return $M[n, S]$
end procedure
```

Correctness: : The initial values in the array have been explained above. The array M is filled up row by row from bottom to the top. In doing so, we are finding the optimum sum when we have i items available and for spaces from $0, \ldots, S$. For a particular i, if the size of this item is greater than s, the optimum solution remains $M[i-1, s]$. But if the size is less than s, we are finding the maximum of the two cases of selecting and rejecting this item. Since s_i is an integer, we will have all the possible values of $s - s_i$ available beforehand and we can simply look for optimum sum when there are $i-1$ items and space available is $s - s_i$. The value of $M[n, S]$ is thus guaranteed to be optimum.

Running time: : The algorithm initializes the array and then starts filling up the array using the previous values. In essence, there is no recursive call but only resuse of the previous values. The algorithm just have the nested loop to consider for the complexity which is $O(nS)$.

(d) We need to store nS entries in the matrix. The space required to store these entries is $nS \log V$ where V is bounded by $\sum_{i=1}^{n} v_i$. The nS term output size can be obtained when the $\log S$ is the exponent of 2. Therefore, the algorithm is not polynomial in the size of input.

4: Central nodes in trees

Algorithm: The algorithm maintains a 3-dimensional array with indices as u, i, v where $u, v \in V$(set of vertices) and $i = 0, \ldots, k$. Each entry denotes the optimum cost of the configuration when there are i dots and the node under consideration is u with as possible ancestors. $|v-u|$ means hop distance of u from it ancestor v. If it's a invalid ancestor we can store inf there.

Algorithm 4 Central node

```
procedure $k\_mark(T)$
    Store all the nodes in a stack $S$ in BFS fashion with leaf nodes on top
    for $i = 0, \dots, k$ do
        for each $u = pop(S)$ do
            if $u$ is leaf then
                for each $v$ in $V$ do
                    if i = 0 then
                        $M(u,i,v) = |v-u|$
                    else
                        $M(u,i,v) = 0$
                    end if
                end for
            else
                for each $v$ in $V$ do
                    if i=0 then
                        $M(u,i,v) = \min_{x+y=i}(max(M(u_r,x,v), M(u_l,y,v), |v-u|)$
                    else
                        $M(u,i,v) = min(\min_{x+y=i}(max(M(u_r,x,v), M(u_l,y,v), |v-u|)),$
                        $, M(u,i-1,v))$
                    end if
                end for
            end if
        end for
    end for
    for each $u, v \in V$ and i = 0,...,k do
        if M(u,i,v) = 0 then
            $marking = marking \cup u$
        end if
    end for
    return $marking$
end procedure
```

Correctness: Each entry provides us the cost of the node u from it's marked ancestor. If the cost is zero, it means that this u is marked. After running all the iterations, there will be exactly k markings. We can go through the entire matrix and find out those k entries; u index of those entries will be the solution.

Running time: Each iteration is just accessing some pre-computed values. There are kn outer iterations and for each iteration we have another n iteration (for v) and k iterations for each combination of $x+y=k$. Therefore, the complexity of the algorithm is $O(n^2k^2)$

5: Faster LIS

(a) Let's take an example $A = [4, 2, 10, 5, 3, 6, 9, 8]$. The index of array B will start from 1 as the minimum LIS can be 1. We start with $i = 7$ with $A[i] = 8$ so $B[1] = 8$ as the only LIS of length 1 is $A[7]$ as of now. Next $i = 6$ with $A[i] = 9$. Since $LIS[6]$ is also 1, we overwrite the value in $B[1]$ as 9. 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 $B = [10, 6, 5, 4]$

(b)

Algorithm: The pseudocode for the algorithm is given in Algorithm 5.

Algorithm 5 Faster LIS

```
1: procedure $faster\_LIS(A)$
2:     for $i = n - 1, \ldots, 0$ do
3:         $j = modified\_BS(A[i])$
4:         if k = -1 then
5:             Add $A[i]$ to the end of $B$
6:         else
7:             $B[j] = A[i]$
8:         end if
9:     end for
10:    return len(B)
11: end procedure
```

We start from the right hand side of array A. The function $modified\_BS(a)$ is a modified version of Binary Search which returns the smallest value (we will never have equal as all the elements in A are distinct) closest to $A[i]$. When the search returns some index, it means that the LIS that has a length equal to that index can still be formed if the first element in that is replaced by the current $A[i]$. The else part just indicates that the $A[i]$ is the starting point of a LIS with length 1 greater than the LIS found till now and hence we append it to B. The complexity of this search is still same as that of a normal binary search. After the iterations are complete, we just return the length of the array B as this is largest subsequence present in A. In our example, since the length of B is 4, therefore LIS for A is 4.

Correctness: Since each index of array B is the largest starting element of LIS of length equal to that index, this implies that a LIS of that length exists. Therefore, the last index of B will denote the largest LIS that could be found in A and hence $len(B)$ gives the largest LIS possible in A.

Running time: The algorithm iterates over n and in the worst case (when A is sorted) will result in a $\log n$ binary search. Therefore, the complexity is $O(n \log n)$

6: Maximizing Happiness

(a) Let's take a setting when there are two children and two gifts. Let the values of A_{ij} be, $A_{11} = 5$, $A_{12} = 4$, $A_{21} = 7000$ and $A_{22} = 2$. If Santa is using the greedy technique, he will start with the child 1 and give him gift 1 as gift 1 is regarded highest by child 1. He only has gift 2 left for child which gives us the total value $5 + 2 = 7$. But the optimum solution for this setting is by giving gift 1 to child 2 and gift 2 to child 1 and getting the total value as 7004. Therefore, in this setting, the greedy technique can be as bad as 1000 times. $(7 < 7 \times 1000)$

(b) Let us the the pairwise algorithm assigns gifts as $A_{1\pi(1)}, A_{2\pi(2)}, \ldots, A_{n\pi(n)}$. If we consider all the possible pairs of these values then we have the following inequalities,

$$A_{1\pi(1)} + A_{2\pi(2)} \geq A_{1\pi(2)} + A_{2\pi(1)}$$
$$A_{1\pi(1)} + A_{3\pi(3)} \geq A_{1\pi(3)} + A_{3\pi(1)} \ \cdots$$

These inequalities simply imply that by swapping gifts for any two childs,we are not making any improvement to the total value. Summing these inequalities result in the following relation:

$$(n-1) \cdot (A_{1\pi(1)} + A_{2\pi(2)} + \cdots + A_{n\pi(n)})$$
$$\geq (A_{1\pi(2)} + A_{1\pi(3)} + \cdots A_{1\pi(n)}) + \cdots + (A_{n\pi(1)} + A_{n\pi 2} + \cdots A_{n\pi(n-1)})$$

Adding $A_{1\pi(1)} + A_{2\pi(2)} + \cdots + A_{n\pi(n)}$ on both sides result in

$$n \cdot (A_{1\pi(1)} + A_{2\pi(2)} + \cdots + A_{n\pi(n)})$$
$$\geq (A_{1\pi(1)} + A_{1\pi(2)} + \cdots A_{1\pi(n)}) + \cdots + (A_{n\pi(1)} + A_{n\pi 2} + \cdots A_{n\pi(n)})$$

The left side of the inequality can be written as $n \cdot v$, where v is the total value of pairwise solution.

$$n \cdot v \geq (A_{1\pi(1)} + A_{1\pi(2)} + \cdots A_{1\pi(n)}) + \cdots + (A_{n\pi(1)} + A_{n\pi 2} + \cdots A_{n\pi(n)})$$

The right side of the inequality is all possible values of $A_{i\pi(j)}$ and the global optimum sum is a part of it. The remaining part of the proof will require that expression on the R.H.S. is greater than $\frac{n \cdot OPT}{2}$.