1: Probability of k Heads

Given n different coins, we want to calculate the probability that exactly k of them turn up heads. If we want a polynomial time algorithm, we will have be clever because there are $\frac{n!}{k!(n-k)!}$ possible outcomes. There would potentially be an exponential number of multiplication operations to find the product of the independent probabilities for all the successful events.

Algorithm: An observation is we have two outcomes for a coin toss: heads and tails. To have precisely k heads on flip n, then we need $k-1$ heads on the flip $n-1$ (and flip n is heads) or k heads on flip $n-1$ (and flip n is tails).

Starting at $cur\_n = 1$, for $0 \leq cur\_k \leq cur\_n$ calculate the probability of getting exactly cur_k heads with cur_n coins. Increase cur_n and repeat for $1 \leq cur\_n \leq n$. The probability is cached and we use previous probabilities to calculate the probability for cur_n and cur_k. This is repeated until $cur\_n = n$ and $cur\_k = k$ at which point we have the desired probability.

$$cache[(n,k)] = p_n * cache[(n-1,k-1)] + (1-p_n) * cache[(n-1,k)]$$

We initialize the cache with `cache[(0, 0)] = 1`.

Correctness: A sequence of coin flips are independent events. To find the joint probability of two independent events, we take their product. In our algorithm, we take the joint probability of the events that can lead to getting exactly k heads on n coin flips. Thus we have correctness.

Running time: We multiply at most n pairs of numbers in interval (0, 1) for n coins. We know k will be at most n due to the nature of the problem. Multiplication happens in O(1) time. Giving $n(n * O(1))$.

$\boxed{O(n^2)}$

2: Count Number of Parsings

Collaboration with Maks Cegielski-Johnson.

Algorithm: The following is an algorithm for counting the number of ways a string can be parsed, given a language of valid words.

Algorithm 1 Count number of parsings

```
Global Variables
    cache = dictionary of parsings counts for a given word
    language = collection of words in the language
    L = length of the language
end Global Variables

procedure PARSING_COUNT(s)
    if len(s) == 0 then
        return 1
    end if

    if s in cache then
        return cache[s]
    end if

    for index in range(min(N , L) + 1) do
        substring = the first index characters of s
        if substring in language then
            remaining = s with substring removed from the front of s
            count = count + parsing_count(remaining)
        end if
    end for

    cache[w] = count
    return count
end procedure
```

The recursive algorithm will try every possible parse of s, using the words in the language. At each call of the recursive algorithm, s is scanned left to right for matches with words in the language. If a match is found, a substring is formed (removing the match from the beginning of s) and passed to *parsing_count* and the process is repeated on the substring. The scan of s is continued to find additional parses. Scanning continues the minimum of N and L. A parse is successful if the string passed to *parsing_count* is empty. The total count of parses is returned once all possibilities are tested.

Correctness: The algorithm will terminate as the input string s gets smaller with each recursive call to *parsing_count* (moving towards the base case) or no parsing is found and no recursive call is made.

We can observe that we will find every possible parsing as we exhaustively try every possible parsing of s given the language. Thus the algorithm is always correct.

Running time: There are 2^{N-1} possible parsings in the worst-case. This worst-case is when s is a string consisting of all the same letter (ex. aaaaa) and the language is a growing sequence of length N (ex. a, aa, aaa, aaaa, aaaaa).

We combat this exponential runtime by caching the parse counts of the substrings of s. In the worst-case, there are N entries in the language that need to be parsed and cached. The recursion tree with have N levels, and there will be $N - k$ lookups on each level where k is the recursion level.

$$\sum_{i=1}^{n} n - i = \frac{n(n-1)}{2}$$

This is making the assumption that the language is stored in a hash set structure that provides constant time lookup.

$\boxed{O(n^2) \text{ time complexity}}$

3: The Ill-prepared Burglar

(a) Consider the scenario where the ill-prepared burglar can carry a weight of 10 and the house has the following items.

Item	Value	Weight
T.V.	8	9
Laptop	5	5
Fine Art	7	5

His strategy of picking the most valuable items first won't be most optimal here. He will end up with the T.V. in his bag with a value of 9. The most optimal strategy would be taking the laptop and fine art, putting his loot value at 12.

(b) Again, the ill-prepared burglar can carry a weight of 10, this time picking the items with the best ratio.

Item	Value	Weight	Ratio
Finer Art	9	6	1.5
iPad	5	5	1
jewelry	5	5	1

The burglar's new ratio strategy will lead him to pick the finer art, with no room for anything else and a value of 9. The optimal choice would of been to take the iPad and jewelry for a total value of 10.

Algorithm: The following is an algorithm for finding the most valuable collection of items that can fit in a bank locker if size S.

Algorithm 2 Most Valuable Collection of Items

```
Global Variables
    V = collection of the values for each item
    S = collection of the weights for each item
    cache = the max value of items for a given size
    cache[0] = 0
end Global Variables

procedure Max_Value(S)
    for cur_size in range(1, S+1) do
        for index in range(len(V)) do
            if cur_size ≥ W[index] then
                if V[index] + cache[cur_size - W[index]] > cache[cur_size] then
                    cache[cur_size] = V[index] + cache[cur_size - W[index]]
                end if
            end if
        end for
    end for

    return cache[size]
end procedure
```

The strategy is to calculate the most valuable collection of items for each size of locker s (where $0 \leq s \leq S$) and cache the result. If the best value of each size is calculated in ascending order, the previous results can be used for the next locker until we arrive at a locker of size S.

Using the previous sub-problems, we can find the most valuable collection of items for the current size.

$$cache(s) = MAX\{cache(s - s_i) + v_i\}\ for\ 1 \leq i \leq N$$

We try each item v_i and add it to an already calculated optimal subproblem. We find the previous optimal subproblem by subtracting the size s_i from the current size. Each item is tried and we take the maximum for a locker of size s. This is repeated until a size S is reached, at which point we have found the max total value.

Correctness: We build our final solution using optimal sub-problems. Including item i is the optimal choice for $cache(s)$, if not including item i will still provide an optimal solution to $cache(s - s_i)$. So we know that we should include item i in the optimal solution. This process is continued until $cache(S)$ at which point we have a max total value and a total size $\leq S$.

Running time: For S different sizes, we must consider at most n possible items that will provide the max total value for that size. Simply put, we will do n work, S times.

$\boxed{\text{O(nS)}}$

(d) The algorithm is not polynomial in the input size. The number of bits required to represent S is $log_2(S)$ bits. The length of the input S is proportional to the number of bits required to represent it. So it's not polynomial in the input size.

$\boxed{\text{NO}}$

4: Central Node in Trees

5: Faster LIS

(a) The element at $B[1]$ is the beginning element of the longest increasing subsequence of length 1. Element $B[1]$ is also the largest element in the array seen so far with an LIS of length 1. All the elements are distinct.

We scan the array right to left, so to have an LIS of length 2, we can see that $B[2]$ must be strictly less than $B[1]$. We know this because if $B[1]$ isn't less than $B[2]$ then we can't "build" an LIS starting with element at $B[2]$.

For $1 \leq j \leq S$ then $B[j] < B[j-1]$ will also be satisfied, where S is the best longest increasing subsequence.

(b)

Algorithm:

Correctness:

Running time:

6: Maximizing Happiness

(a) Consider the following setting of children and gifts.

	gift 1	gift 2
child 1	2	1
child 2	2,999	1

If Santa is in a hurry and assigns gifts greedily, child 1 will receive present 1 and child 2 will receive present 2. This will result in a total value of 3. The best assignment that could take place is giving child 2 gift 1 and giving child 1 gift 2. This will result in a total value of 3,000.

(b)