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 operations to find the sum 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 $0 \leq cur\_k \leq cur\_n$ we 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 sum. 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

```
1: Global Variables
2:     cache = dictionary of parsings counts for a given word
3:     language = collection of words in the language
4:     L = length of the language
5: end Global Variables
6:
7: procedure PARSING_COUNT(s)
8:     if len(s) == 0 then
9:         return 1
10:    end if
11:
12:    if s in cache then
13:        return cache[s]
14:    end if
15:
16:    for index in range(min(N , L) + 1) do
17:        substring = the first index characters of s
18:        if substring in language then
19:            remaining = s with substring removed from the front of s
20:            count = count + parsing_count(remaining)
21:        end if
22:    end for
23:
24:    cache[w] = count
25:    return count
26: 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.

$O(n^2)$ 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

```
1: Global Variables
2:     V = collection of the values for each item
3:     S = collection of the weights for each item
4:     cache = the max value of items for a given size
5:     cache[0] = 0
6: end Global Variables
7:
8: procedure MAX_VALUE(S)
9:     for cur_size in range(1, S+1) do
10:        for index in range(len(V)) do
11:            if cur_size ≥ W[index] then
12:                if V[index] + cache[cur_size - W[index]] > cache[cur_size] then
13:                    cache[cur_size] = V[index] + cache[cur_size - W[index]]
14:                end if
15:            end if
16:        end for
17:    end for
18:
19:    return cache[size]
20: 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.

NO

4: Central Node in Trees

Collaboration with Maks Cegielski-Johnson and Dietrich Geisler.

Algorithm:

Algorithm 3 Central nodes in trees

```
1: procedure DRIVER(HEAD, K)
2:     for 0 ≥ i ≥ k − 1 find the min do
3:         MAX(cost(head.l, i, 0), cost(head.r, k-i-1, 0))
4:     end for
5:     return found minimum
6: end procedure
7:
8: procedure COST(NODE, K, CURCOST)
9:     if node.l is null and node.r is null then
10:        if k > 0 then
11:            return ∞
12:        else if k == 1
13:            return curCost
14:        end if
15:        return curCost + 1
16:    end if
17:
18:    return curCost + the min from below for loop
19:    for 0 ≥ i ≥ k − 1 find the min do
20:        for 0 ≥ j ≥ k − 1 find the min do # mark the current node
21:            MAX(cost(node.l, i, 0), cost(node.r, k-i-1, 0))
22:        end for
23:        for 0 ≥ k ≥ k − 1 find the min do # don't mark the current node
24:            MAX(cost(node.l, i, curCost + 1), cost(node.r, k-i-1, curCost + 1))
25:        end for
26:    end for
27: end procedure
```

In the *driver* method, we first mark the root node. This node always must be marked or it would be ∞. Next, we try all possible combinations of passing the remaining k marks down both the left and right side. Consider the maximum of the left and right option for each distribution of k and take the minimum.

In *cost*, first we check if the current node is a child node. If it is and we still have k left over, we know we have made a mistake. If we only have one k left we mark the child node. Otherwise if we are out of k, we just add 1 to the current cost.

In the return statement, we have to check four possible options: mark the current node and go left, mark the current node and go right, don't mark the current node and go left, and don't mark the current node and go right. The maximum of the two pairs of choices is taken, then a minimum is taken across the two maximum choices.

Correctness: We try every possible assignment of marking k of the n vertices. Of all the possible assignments, we take the maximum. Thus by exhaustive search we find the k nodes that minimize the loss and have correctness.

Running time: There are k levels of recursions called from the *Driver* method. For each of those k calls to *cost*, each also calls *cost* k times. A total of n nodes is considered in each recursive call to *cost*. Giving $k * n * k * n$.

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

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: The following is an algorithm for finding the longest increasing subsequence of integers in an array.

Algorithm 4 Find Longest Increasing Subsequence

```
1:  Global Variables
2:      L = length of the longest increasing subsequence starting at a given position
3:      B = jth element is the value of the largest A[i] in array where L[i] ≥ j
4:      best = best subsequence found so far
5:
6:      L[|A| − 1] = 1
7:      B[1] = A[|A| − 1]
8:      best = 1
9:  end Global Variables
10:
11: procedure LIS(A)
12:     for i in range(|A| − 1, 0) do # right-to-left
13:         pos = BinarySearch(B, A[i], 1, best)
14:         if A[i] > B[pos] then
15:             B[pos] = A[i]
16:             L[i] = pos
17:         else if A[i] < B[best] then
18:             best = best + 1
19:             B[best] = A[i]
20:             L[i] = best
21:         end if
22:     end for
23:
24:     return best
25: end procedure
```

With the use of an additional array B, we can improve the LIS algorithm from class.

We move through A from right-to-left. For each element, we use binary search to determine where it might belong in B. If A[i] is greater than B[pos], then we replace B[pos] with A[i] and set L[i] to pos. This means we haven't found a better LIS, but we have found a larger entry with the same subsequence length.

If the value A[i] is less than B[best] (the first element of the current best LIS), then we have found a new LIS. This is because we can prepend the old best LIS with A[i] to form a longer LIS. We update B with A[i] and L[i] with *best* to track this.

Once all elements in A are considered, *best* is returned which is the LIS.

Correctness: Given i that scans A from right-to-left, LIS[j] is correct for all $j > i$. The base case is the right most element has a LIS of 1. B is used to find the largest entry that begins the current best LIS. If a value smaller than B[best] is found that is to the left of B[best], we know we have found a new best LIS. We are adding one element to an existing optimum LIS.

Running time: A is of length n and is scanned from right-to-left. For each element in A, we perform a binary search over B which is also of size n. Binary search is $O(logn)$ in all cases and is performed n times.

$\boxed{O(n\ logn)}$

6: Maximizing Happiness

Collaboration with Maks Cegielski-Johnson and Dietrich Geisler.

(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) Starting at any given setting A_{ij}, we must prove we achieve at least 1/2 the optimum value for the given setting by applying Santa's algorithm. The algorithm sees if swapping the gifts of any pair of children increases the total value. This is repeated until no such swap improves the total value. Call this the *converged setting* and call the total value C. Say we are given the *optimum setting* and call the total value O. We must prove that $C \geq O/2$.

Let π denote the *converged setting* and π^* denote the *optimum setting*. For any $A_{i,\pi(i)}$ and $A_{j,\pi(j)}$ (two children and their gifts from the *converged setting*), we swap the gifts of the children strategically. The goal is to get the value of a child from the *optimum setting* and the other child will have whatever gift was swapped to them.

$$\forall i,\ \exists j\ s.t.\ \pi^*(i) = \pi(j)\ and\ A_{i,\pi(i)} + A_{j,\pi(j)} \geq A_{i,\pi^*(i)} + A_{j,\pi(i)}$$

The LHS of the equality $(A_{i,\pi(i)} + A_{j,\pi(j)})$ will always be equal to or greater to the RHS $(A_{i,\pi^*(i)} + A_{j,\pi(i)})$. This is because Santa's algorithm would of already made this swap if the right hand side resulted in a greater total value.

We know there are n children and n gifts, so we sum the values.

$$\sum_{i=1}^{n} A_{i,\pi(i)} + \sum_{j=1}^{n} A_{j,\pi(j)} \geq \sum_{i=1}^{n} A_{i,\pi^*(i)} + \sum_{i=1}^{n} A_{j_i,\pi(i)}$$

An observation is after summing, the left hand side has a total value equal to $2C$. This is because we have each kid's value from the *converged setting* twice. The right hand side will sum to O + k. We have each kid's value from the *optimum setting* plus some other positive amount of happiness. Since k is positive, we can say $O + k \geq O$.

$$2C \geq O + k \geq O$$
$$C \geq O/2 + k/2 \geq O/2$$
$$C \geq O/2$$

Proving that $C \geq O/2$.

(Bonus) Now Santa's algorithm picks every triple of children and sees if there is a reassignment of gifts that will make them happy. Using the same notation as part b, I will prove that with this new algorithm we can achieve $C \geq 2O/3$.

Given the option of a three way swap, we can make strategic swaps such that on the RHS we have two values of children from the *optimum setting* and another child with the leftover gift.

Let $\pi^*(i) = \pi(j)$ *and* $\pi^*(j) = \pi(k)$.

$$\forall i,\ \exists j, k\ s.t.\ A_{i,\pi(i)} + A_{j,\pi(j)} + A_{k,\pi(k)} \geq A_{i,\pi^*(i)} + A_{j,\pi^*(j)} + A_{k,\pi(i)}$$

The LHS will always be greater than the RHS for the same reasons outlined in part b.

$$\sum_{i=1}^{n} A_{i,\pi(i)} + \sum_{j=1}^{n} A_{j,\pi(j)} + \sum_{k=1}^{n} A_{k,\pi(k)} \geq \sum_{i=1}^{n} A_{i,\pi^*(i)} + \sum_{j=1}^{n} A_{j_i,\pi^*(j)} + \sum_{k=1}^{n} A_{k,\pi(i_k)}$$

After summing,the left hand side has a total value equal to $3C$. This is because we have each kid's value from the *converged setting* three times. The right hand side will sum to 2O + k. We have each kid's value from the *optimum setting* twice plus some other positive amount of happiness. Since k is positive, we can say $2O + k \geq 2O$.

$$3C \geq 2O + k \geq 2O$$
$$C \geq 2O/3 + k/3 \geq 2O/3$$
$$C \geq 2O/3$$

Proving that $C \geq 2O/3$.