Problem 1: Sybil's Bday

1. (a) This strategy, of greedily picking workshops in increasing order of start time, is not optimal. Let's assume we have a number of workshops $w_i : s_i \to t_i$ sorted in increasing order:

w_1: 10am $\to$ 2pm	w_2: 11am $\to$ 12pm	w_3: 12pm $\to$ 1pm	w_4: 1pm $\to$ 4pm

Notice how if we pick workshop w_1 from 10am $\to$ 2pm we miss all other workshops. An optimal strategy would have Sybil attending 3 workshops as opposed to 1, thus, this is not optimal.

(b) If Sybil wants to attend the most number of workshops without either leaving early or arriving late to a workshop, then the strategy of greedily picking workshops in decreasing order of start time is optimal. Let's prove this by assuming for the sake of contradiction that given some set of choices made according to the above strategy, $\{c_1, c_2, ..., c_k\}$, the result is *not* optimal.

Let c_i be the first choice we make s.t. the result is not optimal i.e. making choice c_i makes it impossible for Sybil to attend the maximum number of workshops. If we let OPT be an optimal solution that uses $\{c_1, c_2, ..., c_{i-1}\}$, which we know exists since c_i is the *first* choice making our solution incompatible with OPT, we want to show that swapping choice c_i selected by our strategy into OPT won't decrease the number of workshops OPT has Sybil attending.

Key to this is realizing that in order to maximize the number of workshops Sybil attends, we must choose those workshops with the least overlap. If we have a total of k workshops, then each time Sybil attends a workshop, w, the number of remaining workshops becomes:

$$k = k - 1 - w_{overlap} \mid w_{overlap} = \text{number of workshops overlapping } w.$$

If we consistently choose w's with minimum $w_{overlap}$, $k \to 0$ at a slower rate and we'll be able to pick more workshops.

Let's assume choice c_i of our strategy picks workshop w_i. Regardless what choices OPT makes, it has to either eventually choose w_i, in which case our strategy is optimal, or choose a workshop that overlaps with w_i. But note there will be no workshops that start after w_i. Thus, the only possibility for overlap comes from workshops starting before w_i.

Let's say OPT chooses a workshop $w_p \neq w_i$, but that overlaps with w_i. w_p will have an earlier start time, thus increasing the number of workshops it can potentially overlap with. Formally, if W_p is the set of workshops overlapping w_p and W_i

is the set of workshops overlapping w_i, $|W_p| \geq |W_i|$.

Thus, if we swap our choice c_i into OPT, it'll either maintain or decrease the number of overlapping workshops, which will only increase the available remaining number of workshops Sybil can attend.

(c) Like *part 1*, this strategy is also not optimal. Assume we have 3 workshops:

w_1: 11am → 2pm	w_2: 2pm → 5pm

w_3: 1pm → 3pm

If we were to first choose the workshop with the smallest duration, w_3, this'll overlap with both w_1 and w_2. In this case Sybil will only attend a single workshop instead of 2, making this strategy sub-optimal.

2. **Algorithm** Let's greedily pick times to be at the party in decreasing order of guest start time. Concretely, let's first sort the times guests will be at the party in decreasing order of start time, s_i, then repeatedly pick the first start time in the list, deleting all guest intervals that overlap with this.

Proof Let's assume that our algorithm makes a series of choices $\{c_1, c_2, ..., c_k\}$, but for the sake of contradiction let's assume the result of these choices is not optimal. Let c_i be the first of these choices incompatible with an optimal solution.

Let OPT be an optimal solution that uses choices $\{c_1, c_2, ..., c_{i-1}\}$. In our solution, choice c_i allows Sybil to meet guest i who's at the party from $(s_i,\ t_i)$. If OPT doesn't make the same choice as c_i, in order to meet guest i it must eventually pick some time, t_p, within this interval i.e. it must choose some time $s_i < t_p \leq t_i$.

Since there are no start times $> s_i$, since all have already been handled through choices $\{c_1, c_2, ..., c_{i-1}\}$, the only intervals that'll overlap $(s_i,\ t_i)$ must start before s_i. Since $t_p > s_i$, any interval that overlaps t_p will also overlap s_i, but not vice versa.

Therefore, if we swap c_i's choice s_i with t_p then the number of guest intervals overlapping with our choice, i.e. the number of guests Sybil will be able to meet at a particular time, can only increase — when Sybil meets with guest i our algorithm maximizes the number of additional guests she'll be able to visit.

A key difference from the previous proofs is that Sybil isn't staying for the entirety of the interval, but only making an appearance. Our choice c_i is sure not to divide overlapping intervals into distinct, non-overlapping groups that would require additional visits.

Thus, if swapping c_i with t_p only increases the number of guests Sybil meets at a particular time, without possibly generating non-overlapping groups that would require additional visits, this can only decrease the number of total appearances she needs to make to meet everyone. Thus, this contradicts our original assumption that we can't make an optimal solution with choices $\{c_1, c_2, ..., c_i\}$ making our algorithm optimal.

Problem 2: Flying

1. **Algorithm** In order for Sybil to finish her trip in the least number of days she should try to maximize the distance she travels each day. Therefore, when choosing which hotel to stay in each night she should greedily choose the hotel whose distance is furthest from her current location (but still within the 200 mile range limit). Let's prove that greedily choosing hotels whose distance is furthest from our current location is an optimal strategy.

   Proof Our proposition is that the above algorithm maximizes the distance, d_i, Sybil can travel after i days. The faster Sybil is able to travel, the sooner she'll reach her destination and the fewer hotels she'll have to stay at. Thus, if after i days she's traveled as far as possible, she'll have minimized the number of nights spent at hotels while traveling.

   Let's prove this proposition via induction on i, where c_i represents the i^{th} choice made about which hotel to stay at. In the base case, at $i = 0$, we haven't traveled for any days and therefore have yet to make a choice about where to stay. Thus, $d_0 = 0$. Let's assume that for some $0 \leq k \leq i$, our strategy has picked hotels in such a way as to maximize the distance traveled. Let's now show that our algorithm works for c_{k+1}.

   When making our $(k+1)^{th}$ choice we'll choose the hotel furthest from our current location. When we add this to the distance traveled after choice c_k, which represents the max distance traveled after k choices, we'll get the maximum possible distance traveled after $(k+1)$ choices of hotels.

   Thus, our algorithm successfully maximizes the distance traveled after each day, meaning we'll reach our destination quicker and will therefore have to spend fewer nights in hotels.

2. Let's say the total distance Sybil has to travel to get to Oregon is 300 miles and there are 3 hotels along the path. Let's denote each hotel using a pair of values: {distance from origin; cost}. Using this, let's describe the available hotels with the set:

   $$\{\{h_1 : 50;\ \$50\}, \{h_2 : 100;\ \$150\}, \{h_3 : 250;\ \$200\}\}$$

   If we follow Sybil's greedy strategy, we'll first stay at hotel h_1, since the ratio of cost to distance traveled is $\frac{1}{1}$. Sybil will then have a remaining 250 miles to go, meaning she'll have to stay at another hotel. Of these, h_3 has the best ratio, so in total Sybil will have to stay 2 nights at hotels at a total cost of \$250.

Note, however, that if Sybil first stays at h_2 she'll be able to make it to her destination staying only one night in a hotel at a cost of \$150, which is cheaper than the greedy strategy above.

3. Let's instead solve this dynamically. Given a list of hotels $\{h_1, h_2, ..., h_i\}$, each with a *distance* from the origin and *cost* field, let's first sort this list in increasing order of *distance*. Next, let's create an $i \times j$ table where the i^{th} value represents a hotel and the j^{th} value represents a distance. Our recurrence relation will look something like:

$$T(i,j) = \begin{cases} T(i-1,j) & \text{if } j < i_{distance} \\ \min \begin{cases} i_{cost} + T(i-1, j - i_{distance}) \\ T(i-1,j) \end{cases} & \text{if } i_{distance} \leq j \leq (i_{distance} + 200) \\ \infty & \text{if } j > (i_{distance} + 200) \end{cases}$$

Each cell $(i,\ j)$ in our table will represent the minimum cost, using only i hotels, to travel j miles. We'll fill in our table from left to right, top to bottom. In the base case, when $i = 0$ we can only travel 200 miles at a cost of \$0. After that, it's beyond our range, something we'll represent with ∞. When $j = 0$ we never need to stay in a hotel so the cost is always \$0. Our initial table will look something like:

	0	1	2	...	200	...	j
h_0	0	0	0	...	0	∞	∞
h_1	0						
...	0						
...	0						
h_i	0						

Let's prove the above algorithm works. Assume there is some optimal solution that has already filled in $T(i, j-1)$ with the minimum amount of money needed to go $j-1$ miles using only i hotels. We want to show that following the recursion above, our entry into $T(i,j)$ will be just as good as the optimal solution.

If j is out of our range i.e. $j > (i_{distance} + 200)$, then there's no way we can reach j, so we'll set $T(i,j) = \infty$. If, on the other hand, $j < i_{distance}$, we've yet to get to h_i and therefore can't stay there. In this case, we'll set $T(i,j) = T(i-1,j)$, which represents the minimum amount of money needed to go j miles using only $i-1$ hotels. If $i_{distance} \leq j \leq (i_{distance} + 200)$ then let's take the minimum of staying at h_i to go j miles and not staying at h_i to go j miles.

Thus, we set $T(i,j)$ equal to the minimum cost of all possible ways to go j miles, making our solution just as good as the optimal solution.

Problem 3: Failure

This modified algorithm fails mainly due to the swapping step. In the proof of this algorithm we want to make $p(i)$ and T siblings so we can merge them. In order to do this, we swap $p(i)$ and T with the two elements at the lowest level of the tree. The problem with this, however, is we're moving higher probability elements into lower positions in the tree.

As our algorithm states, we take successively larger elements and place them into our tree. Thus, $p(i)$ will be greater than or equal to all elements placed in the tree before it. Looking back at the original proof of Huffman encoding, when we swap two elements into the lowest part of the tree we know these elements have lower probability than the siblings they are replacing. Since we're moving smaller probability items lower in the tree, we're sure to not hurt the average bits per symbol of our encoding.

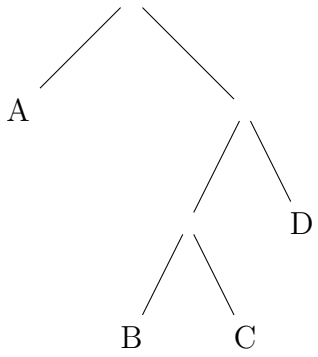
In this modified algorithm, however, we're moving higher probability items lower in the tree. Formally, let's say we're swapping $p(i)$ and T with elements X and Y, respectively. Using similar language from the notes, we're moving X j levels closer to the root and $p(i)$ j levels further from the root. If we use k analogously to j when describing T and Y, the overall change in average bits per symbol is:

$$j(|p(i)| - |X|) + k(|T| - |Y|) \tag{1}$$

Since we're moving $p(i)$, a higher probability element, lower in the tree equation (1) could be positive, which would signify an increase in the average bits per symbol of our encoding, thus creating a sub-optimal solution.

Problem 4: Compression

1. As a binary tree, let's construct:



which corresponds to symbols in a probability distribution:

$$[(A : \tfrac{1}{2}), (D : \tfrac{1}{4}), (B : \tfrac{1}{8}), (C : \tfrac{1}{8})].$$

Let's say our original message is:

$$\text{f} = 1101010101101 \rightarrow \text{DACACC}$$

If we change the first bit to a 0, however, this now decodes to:

$$\text{g} = 0101010101101 \rightarrow \text{ACACADA...}$$

Notice how, for any index $0 \leq k \leq 5$ where $k = \min(\text{len}(f), \text{len}(g))$ in our decoded message, $f(k) \neq g(k)$.

2. At each level i we insert all elements with probability 2^{-i}. Elements are inserted as leafs, therefore each insertion reduces the number of nodes we can insert into at lower levels.

   We consider the number of nodes at each level n and the total probability left to insert l. If the l is 0 we have successfully constructed the tree.

   If we have an element with probability 1 we insert it at level 0 and are done. Otherwise level 1 has 2 nodes that can be inserted and a maximum of 2 elements which can be inserted on this level. For the first two levels we can always insert all elements of the corresponding probability.

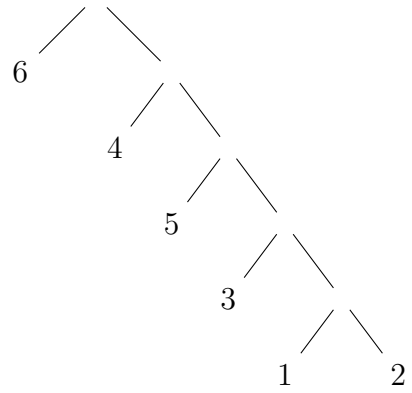
We induct on the level i. At each level i we assume there are n nodes, and l is at most $n \times 2^{-i}$. With this assumption we can insert all remaining elements on this level (if they all have probability 2^i). We show this implies level $i+1$ will also be able to insert all elements of probability $2^{-(i+1)}$.

Let us insert an arbitrary number of elements s into our tree, at level i and assume we can insert all elements of size 2^{-i}. Level $i+1$ has $2s$ fewer nodes that we can insert elements into, but total number of elements we could possibly insert on level $i+1$ is $2s$ smaller because we have reduced the total probability to insert by $(2^{-i})s$ and we are inserting elements of probability $2^{-(i+1)}$. As a result, if all remaining elements were of probability $2^{-(i+1)}$ they could fit on level $i+1$. This shows that for any arbitrary level we can insert all elements required and therefore we can build the tree.

3. (a) With this lucky die, the probability of each value becomes:

$$[(1 : \tfrac{1}{100}), (2 : \tfrac{3}{100}), (3 : \tfrac{6}{100}), (4 : \tfrac{3}{10}), (5 : \tfrac{1}{10}), , (6 : \tfrac{1}{2})].$$

If we build our tree according to the above algorithm, we get:



such that we have an encoding:

1	11110
2	11111
3	1110
4	10
5	110
6	0

(b) Encoding: 001100001110001000. We use a total of 18 bits to encode these 12 symbols, therefore our average bits per symbol $= \frac{3}{2} = 1.5$.

(c) Encoding: 11110111101111011111111101101110. We use a total of 32 bits to encode these 7 symbols, therefore our average bits per symbol $= \frac{32}{7} \approx 4.6$.

(d) The expected number of bits per symbol, using our table from (a) as well as the distribution p equals:

$$\frac{1}{100} * 5 + \frac{3}{100} * 5 + \frac{6}{100} * 4 + \frac{3}{10} * 2 + \frac{1}{10} * 3 + \frac{1}{2} * 1 = \frac{46}{25} = 1.84$$

(e) Note, using the equation for entropy, $\sum_i p(i)|log_2(p(i))|$ we get something very close to the above, namely ≈ 1.815. This means our Huffman encoding scheme uses roughly only .025 bits per symbol more than the optimal solution. As expected, this is ≤ 1.

4. Huffman encoding is a greedy algorithm since it continuously picks elements with the **lowest probabilities to merge first**. Since we're building the tree from the bottom up, though, this makes sense. Merging smaller probability elements **puts them lower in the tree**. This is good, since we're trying to get an average bits per symbol close to the distribution's entropy, which we'll only be able to do if, in general, lower probability elements are at depths greater than higher probability ones.

5. Let's use the changes we make in *part 6* as a hint to how we might construct a set of symbols and probabilities that contradict the claim that for each symbol, α, its depth is at most 1 more than $|\log_2 p(\alpha)|$.

   Let's first start by finding a probability, β, whose $|\log_2 p(\beta)|$ is close to an integer value. For example, let $\beta = .126$ such that $|\log_2 p(\beta)| \approx 2.9885$. Next, let's try to find a distribution that, when we use the Shannon-Fano algorithm, ends up placing β at a depth of 4 in our tree. If we can do this, we'll have contradicted our previous claim.

   Let's create a probability distribution for a set of symbols as follows:

$$[(A : .30), (B : .20), (C : .18), (D : .174), (E : .126), (F : .02)].$$

   Using Shannon-Fano, we'll first divide into two groups:

$$1\text{: } [(A : .30), (B : .20)]$$

$$2\text{: } [C : .18), (D : .174), (E : .126), (F : .02)].$$

   The group on the left will eventually lead to encodings for A and B of [00] and [01], respectively. More interesting, though, is what happens in the group on the right. Following our algorithm, we next want to divide our list s.t. the frequency counts on

the left and right are as close as possible. If we place C and D in one group and E and F in another, the difference between their total probabilities is .208. If we divide the list placing D, E, and F in a group, then the difference in the total probabilities is only .14.

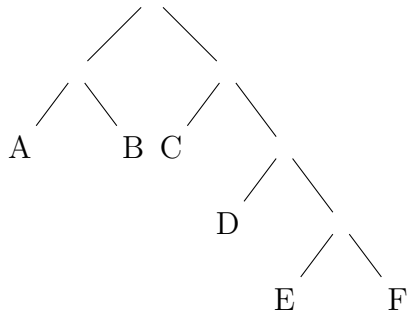
$$[C : .18)]$$

$$[(D : .174), (E : .126), (F : .02)]$$

If we continue, we'll eventually get encodings for each symbol in the distribution of:

A	00
B	01
C	10
D	110
E	1110
F	1111

which maps to a graph that looks like:



Notice, E is at a depth of 4 in our graph, which is greater than $1 + |\log_2(.126)|$ contradicting the above claim.

6. We want to prove that modifying the Shannon-Fano algorithm such that, if instead of dividing a list of probabilities to make the halves as close as possible we instead divide the list into two parts at the first point which makes the left side's probability greater than $(T - s)/2$ where T is the total probability of the list and s is the smallest probability in the list, we can substantiate the claim that *each* symbol, α, in the alphabet will be at depth no greater than $1 + |\log_2 p(\alpha)|$

In order to do this, let's restate our claim by stating that if a symbol α ends up at depth d then its probability can be at most $\frac{2}{2^d}$ i.e. its depth should only vary by at most 1 from $|\log_2 p(\alpha)|$.

Let's define a function $f(v)$ for any internal node v (not a leaf) that computes the sum of the probabilities of all descendants of v except the descendant with the smallest probability. Given some list of probabilities, A, sorted in decreasing order with length n and total probability T, we subdivide our list to place L_T of the total probability into the left list and R_T into the right list.

According to our modified algorithm, we pick the *first* point that makes $L_T > (T-s)/2$, where s is the smallest probability in A. Let's assume this point is right after element i such that L_T is the total probability of $A[1:i]$.

$$L_T = \sum_{k=1}^{i} p(A[k]) > (T-s)/2 \tag{1}$$

Since the probabilities are sorted in decreasing order, and L_T includes the least number of elements needed to make its probability greater than $(T-s)/2$, if we exclude the smallest probability in $A[1:i]$, i.e. only sum the probabilities in $A[1:i-1]$, we'll have a probability less than or equal to $(T-s)/2$. This is exactly what $f(v)$ does. Thus, the left child resultant from splitting A, $A[1:i]$, satisfies:

$$f(A[1:i]) \leq (T-s)/2 \leq \frac{1}{2} \tag{2}$$

The right side resultant from splitting A, $A[i+1:n]$, will have total probability:

$$T - (T-s)/2 = (T+s)/2 \tag{3}$$

If we apply $f(v)$ to this, we eliminate the smallest probability element when computing its probability. The smallest probability element here is simply s. This yields:

$$(T-s)/2 < \frac{1}{2} \tag{4}$$

Thus, we've shown that for some parent node p and child v:

$$f(v) \leq \frac{1}{2} f(p) \tag{5}$$

Notice, then, that as some symbol increases its depth, its total probability (minus its descendant with the least probability) *at least halves*. For some symbol at depth d, then, if we let $f(v) \leq \frac{1}{2^d}$ we can use (5) to rewrite things as:

$$\frac{1}{2} f(p) \leq \frac{1}{2^d} \tag{6}$$

$$f(p) \leq \frac{2}{2^d} \tag{7}$$

Thus, an upper bound for the parent node p is $\frac{2}{2^d}$, proving our claim.

7. (a) entropy $= .9|\log_2(.9)| + .1|\log_2(.1)| \approx .469$

   (b) The expected bits per symbol is 1, so more than twice the entropy.

   (c) Let's construct a table of triples along with their probabilities:

aaa: .729	aab: .081	aba: .081	baa: .081
abb: .009	bba: .009	bab: .009	bbb: .001

This'll give us a Huffman encoding that looks like:

aaa	0
aba	100
aab	101
baa	110
abb	11100
bba	11101
bab	11110
bbb	11111

The expected bits per supersymbol is 1.598. Compared to the above's entropy, which is ≈ 1.407, the ratio of our expected bits per symbol to the entropy is much smaller than in b, which is good.

Problem 5: Data Storage

1. For each integer we encode a 6 bit binary string representing the smallest integer value of k such that the integer we are coding is less than 2^k. We follow the first 6 bits with the integer encoded as a 64-bit binary string, without leading zeros, and without a leading 1.

   We can infer the number of leading zeros because the smallest value of 2^k is also the index of the lead 1 digit. This also allows us to infer the kth index in the binary string is 1, so we do not represent the leading 1 bit for any integer.

   The largest number of bits to represent $2^{64-1} - 1$ is 64 bits, and $2^6 = 64$, so we can represent the length of the binary coding of an integer with 6 bits.

   Each $i < 2^k$ requires $k - 1$ bits to represent, plus 6 bits to encoding its length. We use at most $k + 5$ bits to encoding integer i.

2. The entropy is approximately 0.1 per symbol.

   We divide the sequence into sections of length 100. For each section of length 100, we record the position of the events that are not A. It requires 7 bits to record 100 possible positions. We also use an 8th bit to represent whether this is the last non-A event in this subset of length 100.

   The expected number of non-A events is 1 per 100 symbols.

   We use the following scheme to code the non-A symbols.

   $P(D) = .005 \mapsto 0$
   $P(E) = .002 \mapsto 10$
   $P(B) = .001 \mapsto 1110$
   $P(C) = .001 \mapsto 1101$
   $P(F) = .001 \mapsto 1111$

   This encoding scheme has an expected length of 2.1 bits for each symbol.

   We have 8 + 2.1 bits per 100 symbols we encode. 10.1 is very close to the optimal solution.

3. We compute this sequence by starting with the binary string of the first number. This takes 14 bits because we have 10,000 equally likely integers. Each subsequent integer is encoded with 4-bits representing the unsigned-difference between the character encoded and the character before. We use a fifth bit to encode the sign of the difference. This gives us $\frac{9{,}999*5+14}{10{,}000} = 5.001$ bits per integer.

This encoding scheme is close to optimal because after the first integer, the randomness is confined to 32 equally probable symbols. These 32 symbols are the 32 integers within a distance of 16 from the preceding integer. The entropy of 32 equally likely events is 5, and our solution is very close to this optimal solution.

4. We can compress 500 words with 4000 bits, if we have a representation of whether the article is sarcastic or sad. To represent the ordering of sarcastic and sad sections we use 8, 9 bit strings to represent the start and end of the four sarcastic sections. We can also use this information to infer the start and end of the sad sections.

   This gives us $\frac{4072}{500} = 8.144$ bits per symbol.

   This is close to optimal because the arrangement of the sections is random and 9 bits is close to the optimal solution of 8.97 bits to represent 500 equally likely positions. Once we determine which type of section we are in, we use the 8 bit entropy per word.

   If the optimal scheme for encoding words isn't given, we use Huffman encoding to get close to the 8 bit entropy of words.

5. This file is in sorted order by person number, and as a result we do not need to store which person we are encoding because it is not random. If the person doesn't like quizzes, we store a 0. Otherwise we store a 1, and then the 8 bit sequence representing the binary representation of which quiz number is their favorite.

   This solution is close to optimal because it takes at least 1 bit to store the information representing if the person like quizzes. Both choices are equally probable, and equally likely.

   To optimally encode 243 equally likely messages at least 7.925 bits are required. Our solution uses 8 bits which is a factor of less than 0.1 from the optimal solution. We use a total of 9 bits per symbol.

6. We encode a, b as 10 bit strings, followed by 1 bit representing if the result of addition is correct. If this bit indicates that the result of addition is incorrect, this is followed by a 10 bit string representing the value of c. On average we use 26 bits.

   We argue this solution is close to optimal. If the result of addition equation is correct, the c term has no randomness, and 0 entropy. If the addition is incorrect we need to store c because the value of c is random. Encoding whether the result of addition is correct requires 1 bit, because it is equally likely the result of addition is incorrect. a, b, c are between 1 and 1000, giving an entropy of approximately 9.966. We use 10 bits to encode a, b, c which is close to optimal. We do not encode c if it is not random

which is optimal, and we represent whether the result of addition is correct using 1 indicator bit which is optimal.