1A) Have to send the symbols in the alphabet, their initial ordering, the layout of the numbers (for arithmetic coding) and the final output of the arithmetic coding. Some or all of {symbols in the alphabet, initial ordering, layout for arithmetic coding} can be skipped of they are already shared by the encoder and decoder.

1B) The string being sent would be 11222 repeated infinitely, which has an entropy of .97, so asymptotically requires .97 bits per symbol when using arithmetic encoding (if the arithmetic encoding is customized to the string).

1C) The string being sent would just be 4 repeated, which has an entropy of 0, so asymptotically requires 0 bits per symbol when using arithmetic encoding (obviously some constant number of bits would need to be sent, but its bits-per-character are zero when divided by an infinite sequence).

1D) It would adapt well, as compared to a static distribution of character probabilities, since English, C and postscript files have different frequencies of occurrence for different characters.

1E) This would help get better compression for long messages, since the probability distribution of pairs of characters in English is much more heavily skewed than the probability distribution of single characters, so many pairs (such as qz) would be commonly at the tail of the list, and frequent pairs (such as th) would be commonly near the front of the list.
2A) We have the message \( w = xG' + e = xSGP + e \), which is a vector of length \( n \), and we have \( S, G \) and \( P \).

We first reverse the effects of \( P \) by reordering the variables into their original positions according to how they were permuted by \( P \). The addition of \( e \) does not matter in this case, because \( e \) is random noise added to the data, so the noise will be shifted into new positions, but its magnitude will be unchanged.

We then have \( w' = xSG + e \). We now apply the Goppa decoding algorithm to this vector to calculate the message \( xS \). This is possible because the addition of \( e \) introduced \( t \) bits of noise into the data, which is the number of errors the Goppa code can undo, since there are no two different messages \( x \) such that \( xG \) has fewer than \( 2t + 1 \) different bits.

We then have \( w'' = xS \), which is decoded by multiplying by the inverted \( S \), so \( m = w''S^{-1} = xSS^{-1} = x \), and we then have the final message.

2B) Generate a table of all "clean" ciphers and their corresponding messages by running \( w = xG' \) on every possible \( x \). Then the task is, given \( w \), to find the nearest cipher and return that word. To do this quickly, I'd construct a lookup table of every possible \( X \)-tuple (where \( X \) is the \( n/(t+1) \), rounded down) with each table cell containing pointers to the "clean" ciphers which contain those tuples along with the position of the tuple, and use \( w \) to index into this array, searching for a match. From the matches, I would then calculate the one with the least difference and return that.

The lookup table has \( 2^{n/(t+1)} \) entries, each of which has about \( 1 + (n - n/(t+1))/(2^{n/(t+1)}) \) constant size pointers of size \( \log_2 n \) and numbers of size \( \log_2 n \). The target of the pointers are the ciphers, which is an additional \( (n - t\log_2 n) \) entries of size \( n \) (for the cipher) + \( (n - t\log_2 n) \) (for the corresponding message).

The total is:
\[
2^{n/(t+1)} \times \left[ 1 + (n - n/(t+1))/(2^{n/(t+1)}) \right] \times 2 \times \log_2 n + (n - t\log_2 n) \times [(n - t\log_2 n) + n] \text{ bits.}
\]

2C) The weight of \( e \) would simply be \( t-10 \) rather than \( t \), which allows 10 more bits to get scrambled en route. This makes the message easier to crack (e.g. easier to decode via the method outlined in 2B) or compare to another message, if someone happened to be listening on the noiseless end of the transmission.

2D) Advantages:
- Randomness is inherent in the encryption algorithm, so the same message sent over and over again would generate different ciphers. (This could be done in RSA by appending noise to each message, but is not part of the system.) This helps keep anyone from listening for message recurrences or a specific message.
- Not based on factoring, so you don't have an infinite number of programmers on an infinite number of laptops trying to break it.

<next page>
2D, cont)
Disadvantages:
- The length of the cipher is almost twice the length of the message, which is a storage/bandwidth issue when compared to RSA, where $|c| = |m|$.
- You have to be able to generate a random number $e$ of a specified weight for each message sent, which could be a weakness if the random number generator is slow or not sufficiently random.
- The length of the keys is greater ($k * n$ bits for the public key, $k^2 + n^2 + n^2$ for the private key).

3A) The corresponding submatrix is invertible (it is inverted as part of the Simplex algorithm). Since there are two non-zero values per column, if there was a cycle in the non-zero values, it would correspond to an insolveable system of equations, which it would not be possible to invert. Therefore, since the submatrix is invertible, there must not be such a cycle.

3B) No. That kind of situation would represent a non-corner of the feasible region, which is not visited by Simplex.

3C) One step corresponds to changing the flow along one edge from 0 to some positive value until it compensates for all of the flow along another edge (so the other flow becomes zero), and adjusting the rest of the nonzero flows in the graph to compensate.
4) Working on a graph with V vertices and E edges. I will also refer to C as the maximum number of colors. \( C = V \), but it is easier to understand the algorithm by referring to them as separate values.

Variables:

- VC variables \( A \), one for every possible vertex-color. \( A_{v,c} \) is 1 if the color of vertex \( v \) is \( c \), and 0 otherwise.

- C variables \( O \), one for each possible color. \( O_c \) is 1 if color \( C \) is used, and 0 otherwise.

Equations:

- 2VC equations to force the values of \( A_{V,C} \) to 0 or 1
  For all \( v,c \): \( A_{v,c} \geq 0; A_{v,c} \leq 1 \)

- V equations, one for every vertex, to keep its number of colors to 1.
  For all \( v \): \( \sum_{c=1}^{C} A_{v,c} = 1 \)

- CE equations, C for every edge, to keep the colors of the vertices different if they are connected by an edge.
  For each edge between vertex \( i \) and vertex \( j \):
  For all \( c \): \( A_{i,c} + A_{j,c} < 2 \)

- 2C equations to force the values of the \( O \) variables to 0 or 1 (actually, \( O_c \leq 1 \) is not necessary, given the next set of equations and the minimization, but it would probably help the integer program narrow down its search)
  For all \( c \): \( O_c \geq 0; O_c \leq 1 \)

- VC equations to map each variable to a color, to connect the minimization of the sum over \( O \) to the color distinction requirement of variables \( A \).
  For all \( c, v \): \( O_c - A_{v,c} \geq 0 \)

Goal:
Finally, we minimize the following, which is just the number of colors used:
\( \sum_{c=1}^{C} O_c \)

The total number of variables is \( V^2 + V \), and the total number of equations is \( 3V^2 + 3V + VE \), plus the equation to be minimized.

The coloring is represented by the 1s in the \( A \) variables, where the color of vertex \( i \) is \( j \) where \( A_{i,j} = 1 \).
5) We're trying to find a state assignment which maximizes the probability of the observed sequence of nucleotides. At any given time $t$, we can be in one of two states, $X^+$ or $X^-$, where $X$ is the nucleotide being observed.

For the purposes of describing the filling in algorithm, I assume that:
- The sequence of nucleotides is accessible as a zero-indexed one dimensional array $S$ of length $N$.
- The transition probabilities are accessible as an array $M$, where $M[u,v,x,y]$ is the probability of transitioning from nucleotide $u$ to nucleotide $v$ given that the old CG state is $x$ and new CG state is $y$. ($x=0$ means not in a CG island, $x=1$ means in a CG island). So $N[A,G,0,1]$ is the probability of transitioning from $A^-$ to $G^+$.

The algorithm runs on a $2 \times N$ array $A$. Each cell $A[x,n]$ refers to the maximum probability of being in state $x$ at nucleotide $n$. If $x=0$, we are currently not in a CG island, if $x=1$, we are in a CG island.

Algorithm:
$A[x,0] = .5$. This is equivalent to performing the algorithm twice: once assuming we start in a CG island, the other time assuming we don't, and choosing the most likely. Since we do not know the probability of being in a given state at the start, we assume they are equally likely.

At each step, we take the maximum of the probability of ending in this state given the probabilities of being in the two states at the previous nucleotide, and the probability of transitioning from the previous nucleotide in a given state to the current nucleotide at a given state.

$$
	ext{for } n = 1 \text{ to } N-1 \text{ for } x = 0 \text{ to } 1
\quad A[n,x] = \max(A[0,n-1]*M[S[n-1],S[n],0,x], A[1,n-1]*M[S[n-1],S[n],1,x])
$$

To calculate the sequence, we pick whichever state has the larger probability at $n = N-1$, and trace backward by calculating which previous state the current state came from. (we can do this by recalculating $A[0,n-1]*M[S[n-1],S[n],0,x]$ and $A[1,n-1]*M[S[n-1],S[n],1,x]$, and comparing it with $A[n,x]$, or we can store the previous state in each array cell.

The numerical problem with the algorithm is that in practice, the probabilities involved would get quite small. To solve this problem, we can either normalize the probabilities to sum to 1 at each step ($A[0,n]+A[1,n] = 1$) by dividing by the sum of the two cells, or calculate the sum of the logs of the probabilities rather than the product of the probabilities.
6) Paying $\beta$ for each gap, and $\alpha+\beta$ to complete a gap. For the purposes of the recurrence relation, $a_i$ is the $i$th character of string $A$, $b_i$ is the $i$th character of string $B$, and $\sigma(a,b)$ is the cost of replacing character $a$ with character $b$. The goal is to find the alignment of minimum cost.

Each cell stores three values: $E =$ cost of optimal alignment of form $A$, $B$, _; $F =$ cost of optimal alignment of form $A_$, $B$; $W =$ cost of optimal alignment.

$$
E_{0j} = \infty \\
E_{ij} = \min\{ E_{i-1,j} + \beta, \ W_{i-1,j} + \beta \} \\
F_{i0} = \infty \\
F_{ij} = \min\{ F_{ij-1} + \beta, \ W_{ij-1} + \beta \} \\
W_{00} = 0 \\
W_{ij} = \min\{ W_{i-1,j-1} + \sigma(a_i,b_j), \ E_{ij} + \alpha, \ F_{ij} + \alpha \}
$$

This is just a way of moving the calculations around – instead of paying $\alpha$ at the start of a gap by placing it in the $E$, $F$ calculations, we pay it at the end of a gap by placing it in the $W$ calculations. The rest of the algorithm is performed as before, using the new recurrence relations. Since the relations do not require any more access to previous data than before, the algorithm still runs in $O(nm)$ time.
7A) Off-line, we would perform Latent Semantic Indexing using SVD, but instead of using the terms in the documents, we use the outgoing links in the document. This is essentially indexing and searching, except we use outgoing links as keywords rather than arbitrary terms in the document. We would also have to store all the outgoing links of all the documents.

7B) To perform a query, we would collect the outgoing links in the set S presented by the user, then transform this query to the lower-dimensional space and find the documents most similar to the query in the lower-dimensional space. The program would then return the outgoing links (the destination pages) of the most-similar documents.

7C) There are two steps to this – first we have to convert the query into the lower-dimensional space, and then we have to retrieve the nearest matching documents.

- Converting the query is multiplying an nx1 matrix by a nxk matrix, which can be done in $O(n^2k)$ time (assuming constant time per multiply).
- Retrieving the nearest set of queries can take up to $O(nk)$ time to calculate the distance (dot-product or otherwise) between the query and each document (assuming k time for distance calculation), though that could probably be optimized with a spatial data structure.

The total time for a query is therefore $O(n^2k)$.

7D) The preprocessing in 7A would be insufficient for two reasons. First, the LSI was performed by using the outgoing links, not the incoming links, so the similarity metric would not be the same, and second, we would need to store the incoming links to each document in order to retrieve the target dataset.

- Conceptually, the easiest way to perform the task is to reverse the direction of all the links and run on the resulting graph rather than the original graph.

7E) Given m documents and n terms, we would run SVD on a matrix of size $(2m + n)$ x m, where the m columns are one for each document, m rows are for incoming links, n rows are for outgoing links, and n rows are for terms.

- Instead of just setting the contents of the matrix to 1 or 0, we set them to whatever values are required by the linear combination (e.g. if outgoing links are 2x as important as incoming links, and 3x times as important as terms, we could set the values to 6 for outgoing links, 3 for incoming links, and 2 for terms) and run SVD on that.