1. A clear and obvious formulation is that we need to make a series of
choices of edges of the path. In particular, we could make a choice for
the first edge out of vertex \( s \) taking us to say \( \text{succ}(s) \), and then, it is
evident that the optimal substructure property holds for the residual
path from \( \text{succ}(s) \) to \( t \).

The corresponding recurrence relation is

\[
L_{s,t} = \max_{x \in N^+(s)} (w(s, x) + L_{x,t})
\]

Here, \( L \) with subscripts stands for the longest path from the first
mentioned vertex to the second and \( w \) is the weight of the edge within
parentheses. \( N^+ \) stands for the set of out-neighbours of the vertex
in parentheses, while \( N^- \) stands for the corresponding in-neighbours.
These terms clearly apply only for \textit{directed} graphs.

It follows immediately that there are several overlapping subproblems
because each of these out-neighbours of \( s \) over which we are maximising
represent subproblems, optimal solutions to which potentially use
common paths. And solutions to subproblems on which the current
subproblem depend involve paths from the out-neighbours and one edge
less. Thus to solve the problem bottom up, we need to compute back-
ward paths on exactly one edge (the in-neighbours of \( t \)) and then work
backwards. We add all these in-neighbours into a queue and update
the longest distances to them from \( t \) (in backward direction) according
to the order in which we visit them from \( t \). After this is done, sub-
sequent steps involve alternation between dequeueing the elements
and enqueueing \textbf{ALL} their in-neighbours and repeating this until we
exhaust all vertices in the graph.
The backward distances of all vertices other than \( t \) are initialised to \(-\infty\). Everytime, we use the above recurrence relation to update an increase in the longest weighted path to a corresponding vertex. We also save a parent field which is initialised to \( \text{nil} \) and updated to the out-neighbour from which we reach this vertex (along a longest path) backward from \( t \). This is basically the same as the standard Breath First Search (BFS) Algorithm, modified to work backwards and to take into account edge weights. Unlike in the standard BFS, we are finding here longest paths, rather than shortest ones. Also, unlike in BFS, we do not colour code the vertices to keep track of which have been visited and which have not. The acyclicity property of the graph (remember that it is a directed acyclic graph) ensures that the procedure terminates despite not using colour coding. Whenever we reach a source vertex (one with no in-neighbours) the procedure does not enqueue any vertices, and so we are guaranteed termination.

The pseudocode could be something like this:

\[
\text{LENGTH}-\text{LONGEST}-\text{PATH} \ (G, s, t) \\
1. \text{for each } v \in V \setminus t \\
2. \quad d[v] \leftarrow -\infty \\
3. \quad \pi[v] \leftarrow \text{nil} \\
4. \quad d[t] \leftarrow 0 \\
5. \quad \pi[t] \leftarrow \text{nil} \\
6. \quad Q \leftarrow \{t\}. \\
7. \text{while } (Q \neq \emptyset) \\
8. \quad \text{do current} \leftarrow \text{dequeue}(Q). \\
9. \quad \text{for each } v \in N^-(\text{current}), \\
10. \quad \quad \text{do if } (d[\text{current}] + w(v, \text{current}) > d[v]) \\
11. \quad \quad \quad \quad \text{then } d[v] \leftarrow d[\text{current}] + w(v, \text{current}) \\
12. \quad \quad \quad \quad \pi[v] \leftarrow \text{current} \\
13. \quad \quad \quad \quad \text{enqueue}(Q, v) \\
\]

In the end, \( d[s] \) holds the length of the longest path from \( s \) to \( t \). To get an actual longest path, we obtain the first edge as \( s, \pi[s] \) and then concatenate successive applications of the parent function. The routine to obtain the actual longest path can be written in a simple recursive form as follows.
LONGEST-PATH\((s, t)\)

1. if \((s \neq t)\)
2. then return \(s\).LONGEST-PATH\((\pi[s], t)\)
3. else return \(\varepsilon\)

Here, \(\varepsilon\) stands for the empty string which is zero characters long and the \((.\) stands for the concatenate operator. The required field of \(\pi\) is assumed to be available from the previous procedure as a global array variable. The subproblem graph is a directed tree with all arrows oriented in the direction of vertex \(t\). The number of the vertices increases and eventually includes all vertices in the graph from which \(t\) is reachable. The complexity of the algorithm is \(O(V + E)\), just like the standard BFS procedure. The problem could also have been solved by probing in the forward direction starting from the vertex \(s\) and the solution is very similar.

2. Here, one can solve it directly by the usual dynamic programming methods or reduce it to a problem already solved in the lecture (as well as in the textbook) and then applying the corresponding algorithm.

(a) First we present the solution by reduction. Note that any palindrome subsequence of a string \(S\) is also a subsequence of the reverse string \(S^R\). Suppose the indices of the original string which constitutes a palindrome subsequence are \(i_1, \ldots, i_k\), with \(i_1 < \cdots < i_k\), where the whole string has \(n\) characters. These indices are \(n - i_k + 1, \ldots, n - i_1 + 1\) respectively in \(S^R\), with \(n - i_k + 1 < \cdots < n - i_1 + 1\) and thus generate the same subsequence.

Now if we show that any common subsequence of \(S\) and \(S^R\) which is not a palindrome cannot be a longest common subsequence it implies that a maximum common subsequence of \(S\) and \(S^R\) is a maximum palindrome subsequence.

Let \(T\) be a longest common subsequence of \(S\) and \(S^R\). We assume \(T\) is not a palindrome subsequence and arrive at a contradiction. Since \(T\) is not a palindrome, \(|T| > 1\). Let the indices of \(T\) in \(S\) and \(S^R\) be \(i_1, \ldots, i_k\) and \(j_1, \ldots, j_k\). The indices of the subsequence of \(S^R\) when translated to indices in \(S\) are \(n - j_1 + 1, \ldots, n - j_k + 1\) and are in backward (or decreasing) order. Now, let \(I = \{i_1, \ldots, i_k\}\) and \(J = \{n - j_k + 1, \ldots, n - j_1 + 1\}\). We know that, \(i_1 < \cdots < i_k\)
and \( n - j_k + 1 < \cdots < n - j_1 + 1 \). \(|\mathcal{I}| = |\mathcal{J}| = k\). Now, \( S_{i_1} = S_{n-j_1+1} \), since each represents the first character of \( T \). Similarly, \( S_{i_2} = S_{n-j_2+1}, \ldots, S_{i_k} = S_{n-j_k+1} \).

If \( i_1 = n - j_1 + 1 \), then we know from the inequalities above that \( n - j_k + 1 < \cdots < n - j_1 + 1 = i_1 < \cdots < i_k \). The subsequence of \( S \) with these indices is a palindrome subsequence of length \( 2k - 1 \). As seen previously, any palindrome subsequence is also a subsequence of the reversed string. However, \( 2k - 1 > k \), for \( k > 1 \), and this contradicts the assumption that \( T \) is a longest common subsequence of \( S \) and \( S^R \).

If \( i_1 > n - j_1 + 1 \), we similarly generate the length \( 2k \) subsequence of \( S \), with indices \( n - j_k + 1 < \cdots < n - j_1 + 1 < i_1 < \cdots < i_k \), which is a palindrome, again leading to a contradiction. Finally, if \( i_1 < n - j_1 + 1 \) then consider the turning-point \( t \) (1 \( \leq \ t \leq k \)), such that, \( i_1 < n - j_1 + 1, \ldots, i_t < n - j_t + 1 \) and \( i_{t+1} \geq n - j_{t+1} + 1 \). Each of the subsequences of \( S \), with indices \( i_1, \ldots, i_t, n - j_t + 1, \ldots, n - j_1 + 1, n - j_k + 1, \ldots, n - j_{t+1} + 1, i_{t+1}, \ldots, i_k \) are palindrome subsequences. They are thus also subsequences of \( S^R \). The first subsequence has length \( 2t \) while the second has length either \( 2(k - t) \) or \( 2(k - t) - 1 \). For any value of \( t > \frac{k}{2} \), the first is of length \( > k \). If \( t < \frac{k}{2} \), and the second subsequence has length \( 2(k - t) \) then it is of length \( > k \).

If the second subsequence is of length \( 2(k - t) - 1 \) and if it is less than \( k \), then we obtain a modified first subsequence by including the extra index \( i_{t+1} \) to obtain a longer palindrome subsequence of length \( > k \). Hence, in all cases we have a common subsequence of \( S \) and \( S^R \) of length \( > k \), a contradiction.

Thus, we can run the standard dynamic programming routine for the longest common subsequence between \( S \) and \( S^R \) to obtain the longest palindrome subsequence in \( O(n^2) \) time.

(b) The direct method involves, as usual, writing a recurrence and then devising a bottom-up algorithm exploiting the optimal substructure and overlapping subproblems properties. In this case, we can consider the first and last characters and if they are identical we include both and then recurse on the subsequence obtained by deleting them. If the first and last characters are non-identical, then we have to eliminate at least one of them, and so we eliminate each by turn and solve the two resultant sub-problems and take the bigger of the two solutions. Thus,

\[
LPS(S, 1, n) = \max\{LPS(S, 1, n-1), LPS(S, 2, n)\}
\]
if \( S_1 \neq S_n \) and

\[
LPS(S, 1, n) = 2 + LPS(S, 2, n - 1)
\]

if \( S_1 = S_n \).

The pseudocode might be something like this (written in recursive form):

\[
\text{LPS}(S, i, j)
\]

1. \text{if} \((j < i)\)
2. \text{then return} 0
3. \text{if} \((i = j)\)
4. \text{then return} 1
5. \text{if} \((S_i = S_j)\)
6. \text{then return} \((2+ \text{LPS}(S, i + 1, j - 1))\)
7. \text{else return} \max\{\text{LPS}(S, i + 1, j), \text{LPS}(S, i, j - 1)\}

The code to get the actual longest palindrome sequence merely requires that we use a few concatenate operations and eventually return a string instead of a number. In the above code, in Line 2, we would return the empty string (zero characters long) denoted by \( \epsilon \), in Line 4, we would return \( S_i \), in Line 6, we would return \( S_j \), \text{LPS (modified)} \( . \) \text{S}_j. \) Here the dot (.) denotes the string concatenation operator. In Line 7, we would simply return \text{LPS (modified)} of the larger of the two. The word \text{modified} signifies the algorithm to construct the actual optimal solution and not just the value of the optimum.

The algorithm if implemented as above has exponential complexity. We need to do a bottom-up implementation and we use the maximum palindrome subsequences of shorter substrings to compute the longer ones. Thus we need to proceed in increasing order of string length.
LPS($S$)
1. for $i \leftarrow 1$ to $n$
2. \hspace{1em} $OPT(i, i) \leftarrow 1$
3. for $i \leftarrow 2$ to $n$ (STRING LENGTH)
4. \hspace{1em} for $j \leftarrow 1$ to $n - i + 1$ (START POSITION)
5. \hspace{2em} if ($S_i = S_j$)
6. \hspace{3em} then $OPT(j, j + i - 1) \leftarrow 2 + OPT(j + 1, j + i - 2)$
7. \hspace{3em} else $OPT(j, j + i - 1) \leftarrow \max\{OPT(j, j + i - 2), OPT(j + 1, j + i - 1)\}$.

Here, the value of $OPT(x, y) = 0$ if $x > y$. To find the actual subsequence, we just modify the pseudocode minimally. The running time is a computation of $\theta(n^2)$ terms, and each computation takes $O(1)$ time, since we read stored values of previous computations from the table $OPT$. Thus the overall running time is $\theta(n^2)$.

3.

4. In this problem, once we have decided what words to print on the first line, we must print the remaining words on the later lines in an optimal fashion (here, minimising the sum of the cubes of the residual spaces at the end of each line). We can decide to break the first line at any word $i$, such that $\sum_{t=1}^{i} l_t + i - 1 \leq M$. In each case, we need to sum the cube of the remaining space on line 1, and the optimal solution to the subproblem on the remaining words (already obtained) and take the minimum of these. Clearly, we have overlapping subproblems, and we need (in order to avoid duplicate computations) the optimal solutions to all suffixes of the given paragraph. That is we need to know the neatest ways of printing the paragraphs $l_{i+1}, \ldots, l_n; l_{i+2}, \ldots, l_n; \ldots; l_n$ in order to determine the neatest way of printing the paragraph $l_i, \ldots, l_n$. 

6
The algorithm in high-level pseudocode could be written as follows.

1. for \( i \leftarrow n \) downto 1
2. \( OPT \leftarrow \infty \)
3. linelength \( \leftarrow l_i \)
4. lastword \( \leftarrow i \)
5. \( \text{breakpoint}[i] \leftarrow i \)
6. while (linelength < M)
7. \quad if \((((M - \text{linelength})^3 + OPT_{\text{lastword}+1,i}) < OPT)\)
8. \quad \quad then \( OPT \leftarrow ((M - \text{linelength})^3 + OPT_{\text{lastword}+1,i}) \)
9. \quad \quad \text{breakpoint}[i] \leftarrow \text{lastword}
10. \quad lastword \leftarrow lastword + 1
11. \quad linelength \leftarrow \text{linelength} + 1 + l_{\text{lastword}}

Here, the array \( \text{breakpoint}[i] \) of size \( n \) gives the optimal breakpoint for the first line, for the suffix paragraph beginning with word \( i \). Once we get the first breakpoint, we must find the second breakpoint by looking for the solution corresponding to the suffix paragraph starting after the last word of line 1. Thus building up the actual solution is routine. The while loop within each iteration of the for loop runs as many times as the number of words which fit into a line starting from word \( i \). This number os \( O(M) \). All the lines within the while loop and prior to the while loop take \( \theta(1) \) time. Thus the overall running time of the algorithm is \( O(Mn) \). As for the extra space used, it is \( \theta(n) \), since we use the two arrays \( OPT_{i,j} \) and \( \text{breakpoint}[\]\). Note that the first is a subscripted variable, but can be implemented as an array. The other variables we use take \( O(1) \) space.

5.
6. Here, whenever we select a node, we cannot select its parent or its children. Since there can be variable number of children but only one parent, we decide the selection or nonselection of a node after deciding for the children. By extension, we take the decision on the inclusion of a node only after we have decided on all the other nodes in the subtree rooted at the node. Thus, we solve the problem for subtrees. The tree, for compactness is stored using the left-child, right-sibling scheme. At any stage, using the optimal substructure property, we
include or exclude a node depending on which case leads to greater total conviviality.

The solution can be described in a more natural and logical manner if we view the original tree, rather than the left-child right-sibling version. In that case, we perform the computations for the subtree rooted at a node only after finding optimal solutions to the subproblems at subtrees rooted at its children. Thus we start at the leaves and progress upwards, eventually reaching the root. The solution procedure is identical at any node.

At each node, we maintain three computed values. The first is \( OPT_{INC}v \) the optimal total conviviality of the subtree rooted at the node \( v \) where the node \( v \) is included in the solution. The second is \( OPT_{EX}v \) the optimal total conviviality of the subtree rooted at the node \( v \) where the node \( v \) is excluded from the solution. The value \( OPTv \) is simply the maximum of \( OPT_{INC}v \) and \( OPT_{EX}v \). The algorithm eventually returns \( OPT_{ROOT} \) as the maximum conviviality achievable. In order to generate the actual guest list, we need to store along with each of \( OPT_{EX}v \), which of its children are also included in the corresponding optimal solution. Note, this is not necessary for \( OPT_{INC}v \) because when we include a node we cannot include any of its children. This is a small overhead whose space requirement is \( O(n) \) where \( n \) is the number of employees. This is because the node corresponding to any employee has exactly one parent in the tree and thus the data that needs to be maintained is \( O(1) \) for each edge in the tree.

To compute the three parameters defined above at a node \( v \), we use the following notation. \( CONVv \) represents the conviviality rating of an employee (here, the node representing the employee), which is given as an input to the algorithm. For leaves \( OPT_{INC} = CONV, OPT_{EX} = 0 \) and \( OPT = OPT_{INC} \) (assuming no employee has a negative conviviality rating). If this assumption is not in place then we need to use the value 0, whenever the corresponding employee has a negative conviviality rating. We denote the children of \( v \) by \( c_{1v}^{r}, \ldots, c_{d(v)}^{r} \).

For an internal node, we have the following formulas.

\[
OPT_{INC}v = OPT_{EX}c_{1v}^{r} + \ldots + OPT_{EX}c_{d(v)}^{r} + CONVv
\]

\[
OPT_{EX}v = OPT_{EX}c_{1v}^{r} + \ldots + OPT_{EX}c_{d(v)}^{r}
\]

Now, in order to implement this in the left-child right-sibling implementation, we merely need to decode how to get the children of a node.
in the direct representation, from the left-child right-sibling representation. It is easy to see that the set of all children of a node in a tree is given by the set of nodes encountered while traversing downwards first to the left-child and then following a maximal path of right-children till we reach a node which has no right-child in the corresponding left-child right-sibling representation. In other words, given a tree $T$, let us denote its left-child right-sibling by $T_{LCRS}$. For a node $v$ in in $T$, the set of all its children in $T$ can be obtained from $T_{LCRS}$ by the following simple routine.

$\text{CHILDREN}(T_{LCRS}, v)$

1. $x \leftarrow v$
2. $children \leftarrow \emptyset$
3. $x \leftarrow \text{left}(x)$
4. if $x \neq \text{null}$ then $children \leftarrow children \cup x$
5. while $(x \neq \text{null})$
6. $children \leftarrow children \cup x$
7. $x \leftarrow \text{right}(x)$
8. return $children$

Hence, we can solve the problem starting at the leaves of the hierarchy tree and proceeding according to the recurrence relations provided above. Note that a node without a leftchild in $T_{LCRS}$ is a leaf in $T$. We can follow the tree walking algorithms to systematically traverse the nodes of $T_{LCRS}$ and use the recurrences defined above to compute the $OPT$ at the root and also encode the information required for computing the guest list.

At each node, we compute the three ratings in time proportional to the number of its children (in the original tree), using the optimal solutions to the subproblems for the subtrees rooted at its children, which is already computed. Thus, the running time to compute the subproblem at a node $v$ is $O(d_v)$. The total running time is therefore $\sum_{v \in T} cd_v = O(n)$. Here $c$ is a positive constant and $d_v$ is the degree of node $v$. The sum of degrees in a tree is asymptotically the same as the number of nodes, and the running time is therefore $O(n)$. 
7. (a) We can use a $|V(G)| \times (|s| + 1)$ matrix to keep track of every vertex reachable by following the symbol sequence on edges starting from $v_0$. We refer to the adjacency matrix $M$ of the given graph (note the adjacency matrix is not the normal 0, 1 matrix, but rather has either 0 or one of the symbols $\sigma_1, \ldots, \sigma_k$ representing the label of the corresponding edge). The column in our datastructure (the auxiliary $|V(G)| \times (|s| + 1)$ matrix) will have a 1 in column zero against the row corresponding to vertex $v_0$, and the rest are all zeros. The next column will have a 1 against every row corresponding to a vertex reachable from $v_0$ by an edge labelled $\sigma_1$. Using this column, we will compute the set of all vertices reachable from the vertices having 1, by an edge labelled $\sigma_2$ and label the corresponding entries 1 in the next column. In this way, we can iterate and eventually see if there is at least a single 1 in the last column numbered $|s|$. If there is a 1 in that column then such a path exists, and otherwise none exists.

The pseudocode for this can be written as follows.

1. Initialise the auxiliary matrix $Aux$ to all zeros except for a 1 against vertex $v_0$ in the first column (column numbered 0).

2. $\textbf{for } i_1 \leftarrow 1 \textbf{ to } |s|$
3. $\quad \textbf{for } i_2 \leftarrow 1 \textbf{ to } |V(G)|$
4. $\quad \quad \textbf{for } i_3 \leftarrow 1 \textbf{ to } |V(G)|$
5. $\quad \quad \quad \textbf{if } (Aux[i_1 - 1, i_2] = 1 \text{ AND } M[i_2, i_3] = \sigma_i) $
6. $\quad \quad \quad \quad \textbf{then } Aux[i_1, i_2] \leftarrow 1$

Note that if in a particular column, if there is more than one way to reach a vertex, the vertex by which we reached is not reflected because we merely save a 1. Alternatively we can modify the pseudocode so that, instead of merely saving a 1 in the entry, we can save the name of the vertex from which we reach it. If there is more than one vertex from which we can reach it, we save the first such vertex. We can keep track of this using a flag variable. If we want to avoid the use of a flag variable, we can directly write the vertex name itself. In this case, we will get the last vertex (in the ordering of the vertices by label) from which we can reach the given vertex following the correct symbol edge.
The modified pseudocode is as follows.

1. Initialise the auxiliary matrix $Aux$ to all zeros except for a 1 against vertex $v_0$ in the first column (column numbered 0).
2. for $i_1 ← 1$ to $|s|$ 
3. for $i_2 ← 1$ to $|V(G)|$
4. for $i_3 ← 1$ to $|V(G)|$
5. if ($Aux[i_1 - 1, i_3] = 1$ AND $M[i_3, i_2] = \sigma_i$)
6. then $Aux[i_1, i_2] ← v_{i_3}$

We have only computed the matrix from which to get the solution. The required output is to give the actual path or a declaration that none exists. To do this, after we have computed the matrix according to the procedure above, we scan the last column until we see a nonzero entry. The row corresponding to that nonzero entry is the last vertex of the path. That entry therein will be a vertex label. We print that symbol as the second last vertex in the path and goto the row corresponding to it in the previous column and read the symbol and print it and go to the previous column etc.

(b) In order to modify the above code to find the most probable path, we merely compute at each stage the weight of the subpath reaching any vertex with symbol $\sigma_i$. Among all subpaths reaching a given vertex we merely take the one with greatest weight. Note that we must maintain one path to each vertex reachable at an
intermediate stage because there is no guarantee that any of these paths will continue with the labelling left over. Some of them could get truncated, so we need to maintain one for each vertex. The input adjacency matrix now has the symbol labels and also the weights of the edges which are probabilities in the interval $(0, 1]$. Our auxiliary datastructure now has two fields for each entry. One is the symbol of the vertex from which we reached the current vertex. The other is the total weight of the path followed so far. The weight of a path is the product of the weight of the constituent edges. The pseudocode is as follows.

1. Initialise the vertex field of the auxiliary matrix $Aux$ to all zeros except for a 1 against vertex $v_0$ in the first column (column numbered 0).
2. Initialise the weight field of all entries to zero except the first column (column numbered 0) against the row for vertex $v_0$ which is initialised to 1.
3. for $i_1 \leftarrow 1$ to $|s|$  
4. for $i_2 \leftarrow 1$ to $|V(G)|$
5. for $i_3 \leftarrow 1$ to $|V(G)|$
6. if $(Aux^w[i_1 - 1, i_3] > 0$ and $M[i_3, i_2] = \sigma_i)$  
7. then if $Aux^w[i_1 - 1, i_3] \times M[i_3, i_2] > Aux^w[i_1, i_2]$  
8. then $Aux^v[i_1, i_2] \leftarrow v_{i_3}$  
9. $M^w[i_3, i_2] \leftarrow Aux^w[i_1 - 1, i_3] \times M^w[i_3, i_2]$

In order to print this path, we scan the last column before proceeding like in the previous case. Here, however instead of starting at the first nonzero entry in the last column, we scan the entire column till we find the maximum element and then use that position to obtain the path with greatest probability. The running time of this, and the previous algorithm is $\theta(nk)$, where $n$ is the number of vertices in the graph and $k$ is the number of symbols in the string ($s = \sigma_1, \ldots, \sigma_k$).

8. (a) If $n \geq 2$, then for any choice of pixel in the first row, there is always at least two choices in transition to the second row. In fact, from any row to the next, there are at least 2 choices since we can always either go down or to at least one of diagonally left or diagonally right, since we cannot be at both extreme columns, simultaneously. For any choice of pixels of a seam of the first
There are at least 2 ways to extend it to include the 
\((k + 1)th\) row. Thus the number of seams is at least \(2^{m+1}\), which is exponential in \(m\).

(b) Clearly, we need to make a series of choices for the pixel of each row, leading to an optimal solution. It is straightforward to show that, given a particular pixel selection for the first row, an optimal solution using this pixel, contains within it an optimal way of solving the problem on the remaining rows, restricted to the solutions compatible with the selection for the first row. In other words, among the three (or two) legal selections, we will pick the optimal one. The problem, thus, has the optimal substructure property. Also, while solving the problem for various rows starting from some row number up to the last, the subproblems arise repeatedly and thus we also have the overlapping subproblem property.

The recurrence relation, which gives the pixel number for the first row, in terms of the selection of all the remaining rows is

\[
OPT_{1,m} = \min_{1 \leq i \leq n} \left\{ d[1,i] + \min_{i-1 \leq l \leq i+1} OPT_{2,m} \right\}
\]

For the \(j\)th row in general, where \(1 \leq j \leq m\), we get the similar-looking relation

\[
OPT_{j,m} = \min_{1 \leq i \leq n} \left\{ d[j,i] + \min_{i-1 \leq l \leq i+1} OPT_{j+1,m} \right\}
\]

The notation is very natural, and thus clear. The \(OPT_{j,m}\) stands for the optimal solution to the subpicture starting at row \(j\) and ending at the last row (row \(m\)). When there is a superscript \(l\), it means the selection for the \(j\)th row is position \(l\).

The solution method is as usual to find the optimal solution including the rows one by one starting from the last. In addition to the optimal solution, from row \(j\) downwards, we also need optimal solutions for each choice of pixel from row \(j\), in order to perform the computation correctly. The base case is the picture consisting of just the last row. The values are simply the entries of \(d[m,i]\), where \(1 \leq i \leq n\).

Let us store the optimal compression in an array \(C\) of size \(m\). For subsequent values, we calculate according to the recurrence relation stated above, but in a bottom-up fashion. The array,
has one entry corresponding to the selection of that pixel and the subpicture on the rows including and below the row of the pixel, where the selected pixel is dropped. Thus the running time of the algorithm is the number of entries $\theta(mn)$ times the time taken to minimise, i.e. $O(n)$. Thus, the algorithm has complexity $O(mn^2)$. The pseudocode could be as follows:

1. for $i \leftarrow 1$ to $n$
   2. do $OPT_{m,m}^i \leftarrow d[m, i]$.
3. for $i \leftarrow m - 1$ downto 1
   4. do for $j \leftarrow 1$ to $n$
       5. $OPT_{i,m}^j \leftarrow \{d[i, j] + \min_{i-1 \leq l \leq i+1} OPT_{j+1,m}^l\}$.
6. $OPT \leftarrow \infty$
7. for $i \leftarrow 1$ to $n$
   8. do if ($OPT_{i,m}^i < OPT$)
       9. then $C[1] \leftarrow i$
10. for $i \leftarrow 2$ to $m$
   11. do $OPT \leftarrow \infty$
   12. for $j \leftarrow C[i - 1] - 1$ to $C[i - 1] + 1$
       13. do if ($OPT_{i,m}^j < OPT$)
           14. then $C[i] \leftarrow j$

9. To compute the optimal cost of the first break, we need the optimal costs for further breaking the two pieces generated. This goes on till we have atomic substrings requiring no further breaks. Also, given that we make the first break at some particular position, it can lead to an optimum solution only if we further break the two fragments also optimally. Thus there is an optimal substructure to the solutions. Further, we will have repeated occurrences of each fragment in several branches of the computation and thus the problem also exhibits the overlapping subproblem property. The recurrence relation to compute the optimal solution is as follows.

$$OPT_{S, L} = \min_{1 \leq t \leq m} \{n + OPT_{S_1, L_1} + OPT_{S_2, L_2}\}$$

Here, the term $OPT_{S, L}$ represents the optimal cost of breaking string $S$, with indices range given, according to positions of break given in array $L$. The terms on the right-hand side represent analogous parameters for the subproblems. We compute it bottom-up. The cost of
breaking a string is zero when it has zero break points and the cost is the total string length when it has one break point. We compute the solution for increasing value of the number of break points. The set of break points considered must be a (contiguous) subarray of \( L \), and the string must start after the previous breakpoint and before the succeeding breakpoint at the ends of the subarray.

Given the string \( S \) and the array \( L \), with \(|S| = n \) and \(|L| = m\), we can use the notation \( S_1, \ldots, S_{m+1} \), to represent the \( m + 1 \) maximal substrings containing no breakpoints. We can likewise use \( S_{i,j} \) to represent the concatenation of all the strings from \( S_i \) to \( S_j \).

The pseudocode might appear like this:

String-breaking\((S, L)\)

1. \textbf{for} \( i \leftarrow 1 \) to \( m \) (number of breakpoints)
2. \textbf{for} \( j \leftarrow 1 \) to \( m - i + 1 \) (index of first segment)
3. \( \text{OPT}_{S_{j,j+i}} \leftarrow \min_{1 \leq t \leq i} \{ \text{OPT}_{S_{j,j+t}} + \text{OPT}_{S_{j+t+1,i}} + |S_{j,j+i}| \} \)

Here, it is assumed that \( \text{OPT}_{S_{i,i}} = 0 \), since it costs nothing to break a string with no breakpoints. The algorithm is like a triple nested for loop. The innermost loop is implicit, using the min notation. Thus the running time is \( O(m^3) \). Getting the actual breakpoints, merely requires us to store the value of \( t \) achieving the minimum in line 3 of the pseudocode.

10. (a) We will show that there is an optimal investment strategy in which there is a single investment in the first year, and then extend the result to show that there is an optimal investment strategy where there is a single investment every year. Suppose there is an optimal strategy with multiple investments in the first year.

(b)

(c)

(d)

11.

12. Since the players can be signed for multiples of $100,000 only, we may assume without loss of generality, that the budget \( X \) available is a multiple of 100,000 as any excess remaining cannot be spent. We can
pick at most one player for each of the \( N \) positions. There are \( P \) candidate players available for each of the positions. Thus the total number of choices is \((P + 1)^N\), which would be the cost of a naive algorithm which cycles through all these possibilities looking for the one which maximises the VORP while staying within the budgetary constraints. This includes the possibility of picking no player for some positions. We can use dynamic programming to reduce this exponential time algorithm to one which runs in polynomial time.

Optimal substructure exists, in the sense that, once we decide the choices for the first \( N - 1 \) positions, the choice for the \( N^{th} \) position is the one which maximises the VORP within the remaining money dictated by the budget. Thus, for each \( i \), we calculate the maximum VORP we can achieve for all possible values of money spent and use this to obtain corresponding values for the first \( i + 1 \) positions. We maintain a table \( TabSel \) in a two dimensional \((X \times N)\) array, and the entries in the position \((i, j)\) indicates the choice of player for position \( j \), which maximises the VORP of the first \( j \) positions using a total budget of at most \( i \times 100000 \) dollars. The corresponding total VORP is stored in a table \( TabVorp \) of the same size. The corresponding entry is \((i, j)\) is computed by having a loop variable \( k \) which runs from 0 to \( i \) and picking the maximum of the VORP totals achievable with a budget of \((i - k) \times 100000\) for the first \((j - 1)\) positions and \( k \times 100000\) for the \( j^{th} \) position. The values of the optimal VORP totals of the first \((j - 1)\) columns for a fixed budget can be read off the table, while the maximum VORP for a given row \( j \) within a fixed budgetary constraint can be read off the corresponding input data.

The number of table entries computed is \( \frac{NX}{100000} \). Each table entry is a maximum over \( \frac{X}{100000} \) entries each of which takes \( O(P) \) time to compute. For each position, the candidate players can be stored in a maximum binary treap (the heap is keyed on the player’s VORP and the binary search tree is keyed on the player’s price). These treaps can be built initially at the cost of \( O(P) \) each, costing a total of \( O(NP) \). We find the maximum valued player within a given budgetary constraint for a position by a corresponding search routine on the treap. The running time of the entire algorithm is thus \( O\left(\frac{NX^2P}{10^{15}} + NP\right) = O(NPX^2)\), which is polynomial in the input size.
The pseudocode could be as follows.

1. Build a Treap for each of the $N$ positions of the candidate $P$ players with heap keyed on VORP and binary search tree keyed on price.
2. Initialise the auxiliary table $TabSel$ to all nil
3. Initialise the auxiliary table $TabVorp$ to all $-\infty$
4. for $j \leftarrow 1$ to $N$
5. for $i \leftarrow 0$ to $\frac{X}{100000}$
6. for $k \leftarrow 0$ to $i$
7. if $(\text{TabVorp}[i-k, j-1] + \text{MAXVORP}_j(k)) > \text{TabVorp}[i, j])$
8. then $\text{TabVorp}[i, j] \leftarrow \text{VORP}(\text{TabSel}[i-k, j-1]) + \text{MAXVORP}_j(k)$
9. The corresponding player is entered in $\text{TabSel}[i, j]$

Here, the $\text{MAXVORP}_j(k)$ is the maximum VORP achievable for a player for position $j$ using a budget of $k \times 100000$. This is taken from the corresponding treap. The value of the