1. We can do the sorting in the following way:
(a) Make $n/p$ groups of the $n$ numbers and sort them in $O(p)$ time each using
the priority queue hence using $O(p \times n/p = n)$ time.
(b) Again make groups of $p$ lists each from those $n/p$ sorted lists. In other
words, if $a_1, a_2, ... a_{n/p}$ are the lists after the first pass, we again divide them
in to $n/p^2$ sets of $p$ lists or $p^2$ elements.
(c) Now take the first element of each list in the first group and insert in the
queue. Extract the minimum one and now insert the next element from that
particular list to the queue. Keep going on in this way until all $p^2$ are sorted
which will take $O(p^2)$. Do the same for all $n/p^2$ numbers making it $O(n)$
time.
(d) Continuing in this fashion, we have to divide the list $O(log_p n)$ times and
each division takes $O(n)$ time to sort. Hence we get a time complexity of
$O(n log_p n)$.

2.
(a) We make two cases. One when $X_i = Y_j$ where the cost will be the same
as it was with the previous $Y$ character and the other when they are equal
which is when we need to add the previous cost to the cost of matching the
last (i-1)th character of $X$ with the last (j-1)th character of $Y$.
Hence we have
$$C(X_i, Y_j) = C(X_i, Y_j) \text{ if } X_i \neq Y_j$$
$$= C(X_{i-1}, Y_{j-1}) + C(X_{i-1}, Y_{j-1}) \text{ if } X_i = Y_j$$

Here we need to initialise the values of the matrix to ensure correctness
in the following way:
$$C(X_0, Y_j) = 1 \forall j \geq 0 \text{ (implying that the a zero length string will occur}
\text{ once trivially})$$
and
$$C(X_i, 0) = 0 \forall i > 0 \text{ (implying that a finite length sequence can’t be a sub-}
\text{ sequence of a zero length sequence)}$$

(b) We can calculate the matrix as:
\textbf{numberOfOccurences}(X, Y)
1. initialize $C(0, j)$ and $C(i, 0)$ as mentioned in (a).
2. for $i = 1$ to $m$
3. for $j = 1$ to $n$
4. if $X_i = Y_j$ then
5. \[ C(i, j) = C(i, j - 1) + C(i - 1, j - 1) \]

6. else
7. \[ C(i, j) = C(i, j - 1) \]
8. return $C(m, n)$

(c) Since the if statement (step 4) is executed $mn$ times hence the running time will be $O(mn)$.

3.[CLRS 15.4-4] Since $c[i, j]$ depends only on $c[i - 1, j - 1], c[i, j - 1]$ and $c[i - 1, j]$, we see that it needs only the previous row or previous column. Hence we can calculate the new row/column by using values from the previously calculated row/column. We can decide on whether to use the row or column depending on the length of the sequences. If its an $m \times n$ matrix and $m < n$ then use the column else use the row. After calculating the new row/column the old row/column can be reused to find the next values.

Since this involves storing 2 rows/columns, the space requirement will be $2 \text{min}(m, n)$.

We can also do this using a single row (let us assume without loss of generality that $n > m$). When we calculate $c[i, j]$ using the previous row, instead of storing $c[i, j]$ in a new row, we can store it in place of $c[i - 1, j - 1]$ since that entry won’t be used again anyway. And then to compute the next entry $c[i, j + 1]$ in place of $c[i, j]$ we use $c[i - 1, j - 1]$. The only problem will be while calculating the first value of every row. For that we will need to use $O(1)$ space and then shift all values by 1 to the right when the whole row is computed.

4.

(a) We can write the recurrence as:
\[ T(n) = T(n - 1) + T(n - 2) + T(n - 3) + 3 \]

Since this is a inhomogeneous recurrence relation, we can obtain its roots and write the solution as the sum of the exponents of those roots along with a constant term. The only positive root of this equation comes out to be around 1.84 (I did it on a calculator using the bisection method and checking for the signs), the other 2 are either negative or complex. We get this result because of the nature of the curve of $a^3 - a^2 - a - 1 = 0$ which is the homogeneous part of the recurrence relation. It will be increasing beyond
the third root because the biggest power is positive. Also, since there is only one root between 0 and 2, (because if we find its derivative we get two maxima/minima: one +ve and one -ve) we know that 1.84 is the biggest one and also the only positive one. Hence we get the time complexity as \(O(2^n)\) or \(O(1.84^n)\) to be more accurate. That is, the time is exponential.

(b) We can form a recurrence using dynamic programming. Instead of using recursion, we fill in the array \(\text{Cheapest}[1:n]\) from the other end as follows:

\[
\text{Cheapest}[i] = \min \{ \text{Cheapest}[i+1], \text{Cheapest}[i+2], \text{Cheapest}[i+3] \} + B[i, 1]
\]

if \(B[i, 2] = 3\)

Or

\[
\text{Cheapest}[i] = \min \{ \text{Cheapest}[i+1], \text{Cheapest}[i+2] \} + B[i, 1]
\]

if \(B[i, 2] = 2\)

Or

\[
\text{Cheapest}[i] = \text{Cheapest}[i + 1]
\]

if \(B[1, 2] = 1\)

We can find all the values of \(\text{Cheapest}\) going from \(i = n - 1\) to 1. Only we have to initialize \(\text{Cheapest}[n] = B[n, 1]\) and \(\text{Cheapest}[n+1]\) and \(\text{Cheapest}[n+2]\) to 0 to ensure that \(\text{Cheapest}[n - 1]\) and \(\text{Cheapest}[n - 2]\) are determined correctly. The algorithm will be a simple decreasing \(for\) loop.

Hence the cheapest cost would be the value of \(\text{Cheapest}[1]\). Also, since evaluation of each slot involves only 2 if statements and a \(O(1)\) operation, calculation will take \(O(n)\) time in all.

Please note that I took Sudheer’s help in doing the b part of above 4th problem.

5. Let \(c[1:n]\) be an array where \(c[i]\) will store the minimum cost of aligning the last \(n-i+1\) words properly on a number of lines. Hence we can build up a recurrence for the minimum cost as:

\[
c[i] = \min_{i \leq j \leq n} \text{cube term} + ve \{ (M - j + i + \sum_{k=i}^{j} t_k)^3 + c[j+1] \}
\]

This means we start in the opposite direction from \(n\). \(c[n]\) will be 0 since we are not to include the spaces at the end of the paragraph as cost. The next few \(c[i]\)s will continue to be zero until the cube term stops being positive. Next the recurrence will be followed.

Hence the algorithm will first find the point where the cost becomes nonzero. Next it will be a straight forward iterative function based on above recurrence. It will examine all \(j\)s from \(i\) to \(n\) until the cube term becomes negative and determine the minimum. Thus if the average length of the words is \(l_{avg}\), we need to examine around \(M/l_{avg}\) possibilities while finding the minimum. Thus the average time complexity will be \(\Theta(nM/l_{avg})\) to fill up all the entries in \(c\) and in the worst case \(O(n^2)\). \(c[1]\) will contain the minimum cost of printing the \(n\) words. In order to store the actual order of the words, the \(j\)
at which minimum is attained can be stored along with the min cost value.

Space required will be $O(n)$ to store the cost for each word and the $j$s.

6. We can keep two arrays $INV[1 : n]$ and $NOTINV[1 : n]$. $INV[i]$ will store the sum of the convivialities of all the guests invited within the subtree rooted at $i$ assuming that $i$ is invited, while $NOTINV[i]$ will store the sum of convivialities assuming that $i$ is not invited. Also let $conv[n]$ hold the convivility ratings for each prospective guest.

We can build up a recursive function to calculate both arrays as follows using a bottom-up approach:

$INV[i] = conv[i] + \sum_{j \in \{immsubordinateofi\}} NOTINV[j] \ldots (1)$

$NOTINV[i] = \sum_{j \in \{immsubordinateofi\}} \max\{\text{NONINV}[j], INV[j]\} \ldots (2)$

Finally, the sum of the convivialities of all the guests in the optimum list will be $\max\{INV[\text{president}], NONINV[\text{president}]\}$

The actual guest list can be constructed by storing an extra field with the two arrays to determine which array was used to compute the $NONINV[parent]$. That is when we calculate the value if $NONINV[i]$ using equation (2) we keep a boolean array $FLAG[1 : n]$ and store a 1 if the $\max$ function returns $INV[j]$ and 0 otherwise. Hence finally scanning the boolean array will return the list in $O(n)$ time.

To actually find the optimal guest list, we will build a tree with identical structure, but three fields: The algorithm will be as follows:

$party(employee)$
1. $inv[employee] = conv[employee]$
2. $noninv[employee] = 0$
3. $temp = left\_child(employee)$
4. while $temp \neq null$
5. \quad $party(temp)$
6. \quad $inv[employee] = inv[employee] + noninv[temp]$
7. \quad $noninv[employee] = noninv[employee] + \max(inv[temp], noninv[temp])$
8. \quad if(inv[employee] > noninv[employee])
9. \quad \quad flag[temp] = 1
10. \quad $temp = right\_sibling(temp)$
11. end while

The argument to the above function will be the president node. Maximum
of \textit{inv[president]}, \textit{noninv[president]} is the sum. The guest list can be determined by scanning the tree once and looking at the flag values for each node. Hence $O(n)$ as mentioned earlier.

Equations (1) and (2) take $O(n)$ time to execute. This is because each node is examined exactly once during the summation in (1) for its parent’s $INV$ calculation and $n$ additions for adding $conv$. Hence $2n$ or $O(n)$ for (1). For (2) again each node is examined only once which results in a time complexity of $O(n)$. 