1. (a) The algorithm written in C:

```c
int T(int n) {
    int sum = 0;
    if (n==0 || n==1)
        return 2;
    for(int j=1;j<=n-1;j++)
        sum+=T(j)*T(j-1);
    return sum;
}
```

Let \( N(n) \) denote the number of operations required to calculate \( T(n) \).

Note that \( N(2) = 2 \). So if \( n \) is even, from the algorithm above we have:

\[
N(n) = \sum_{i=1}^{n-1} 2 \times N(i) - N(n - 1) + N(0) + 2 \times (n - 1)
\]

\[
\geq N(n - 1) + N(n - 2) \geq 2 \times N(n - 2)
\]

\[
\geq 2 \times 2 \times N(n - 4) \geq \cdots \geq 2^{\frac{n}{2}}.
\]

When \( n \) is odd, it’s the same idea. Hence it’s exponential.

(b) An algorithm in C:

```c
T[0]=T[1]=2;
for(j=2;j<=n;j++)
    { T[j] = 0;
        for(k=1;k<j;k++)
            T[j]+=T[k]*T[k-1];
    }
return T[n];
```

(c) An algorithm in C:

```c
T[0]=T[1]=2;
T[2]=T[0]*T[1];
for(j=3;j<=n;j++)
return T[n];
```

2. The solution to the problem of the shortest common subsequence for three strings is essentially identical to the solution with 2 strings. By treating the \( T(i,j,k) \)'s as array entries and updating the table in the appropriate way we can get the following \( O(n^3) \) time algorithm. Let \( a \) be the length of \( A \), \( b \) be the length of \( B \), and \( c \) be the length of \( C \).
for j = 0 to b
  for k = 0 to c
    T(0,j,k)=0

for k = 0 to c
  for i = 0 to a
    T(i,0,k)=0

for i = 0 to a
  for j = 0 to b
    T(i,j,0)=0

for i = 1 to a do
  for j = 1 to b do
    for k = 1 to c do
      if a(i) = b(j) = c(k) then
        T(i,j,k)=T(i-1,j-1,k-1) + 1
      else
        T(i,j,k)= MAX(T(i, j-1, k), T(i-1, j, k), T(i, j, k-1))

3. [This solution was adapted from Brian Wongchaowart’s homework writeup.]

   (a) The path taken by the algorithm through the table to reconstruct
   the shortest common super-sequence is highlighted in bold.
   Let M be the name of our table, let A = zxyyzz and B = zzyxzy.

<table>
<thead>
<tr>
<th>M</th>
<th>j=</th>
<th>i=0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>z</td>
<td>z</td>
<td>y</td>
<td>x</td>
<td>z</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i=0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>z</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>y</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>z</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>z</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

   Following this path from M[6,6] to M[0,0], the shortest common
   super-sequence of A and B is zzyxzyyzz. (For details see part c.)

   (b) If the strings A and B have length m and n, respectively, the table en-
   try M[m, n] gives the length of the shortest common super-sequence
   of A and B. In this example, the bottom-right entry in the table is 9, which
   is the length of the shortest common super-sequence of zxyyzz and zzyxzy.
(c) The letters of string A label the rows of the table, and the letters of string B label the columns. The shortest common super-sequence is constructed in reverse order starting from the bottom-right entry. A letter is added to the beginning of the partial super-sequence constructed so far as one moves past its row or column, so moving left adds the letter at the top of the current column, and moving up adds the letter at the head of the current row. A diagonal move is only allowed when the current row and column are marked with the same letter; that letter is added once to the common super-sequence but is accounted for (moved over) in both words.

The idea is to “reverse” the code that built the table, eventually reaching the upper-left corner (the entry containing the 0) in as few moves as possible, at which point every letter of both words will have been added to the super-sequence. Note that in the first row and column there is obviously only one possible direction in which to move.

Some pseudocode for the trace back:

\[
i = m, \quad j = n, \quad S = \varepsilon
\]

while \( M[i, j] > 0 \) //do until we’ve reached upper left corner of M

\[
\text{if } i == 0 \quad //\text{we’re in the first row}
\]

\[
S = b_j + S; \quad //\text{add the letter from B}
\]

\[
j - -; \quad //\text{move left one column}
\]

\[
\text{else if } j == 0 \quad //\text{we’re in the first column}
\]

\[
S = a_i + S; \quad //\text{add letter from A}
\]

\[
i - -; \quad //\text{move up one row}
\]

\[
\text{else if } a_i == b_j \text{ then}
\]

\[
S = a_i + S; \quad //\text{add this letter (it’s in both A and B)}
\]

\[
i - -, \quad j - -; \quad //\text{move diagonally up and left}
\]

\[
\text{else if } M(i - 1, j) \leq M(i, j - 1) \text{ then}
\]

\[
S = a_i + S; \quad //\text{add letter from A}
\]

\[
i - -; \quad //\text{move up one row}
\]

\[
\text{else}
\]

\[
S = b_j + S; \quad //\text{add letter from B}
\]

\[
j - -; \quad //\text{move left one column}
\]

end if

end while

return S

4. We present an algorithm to compute the minimum edit distance of two strings. Note that:

(a) If it were possible to convert \( a_1, \ldots, a_{m-1} \) into \( b_1, \ldots, b_n \), one could complete the transformation of \( A \) into \( B \) by deleting \( a_m \).

(b) If it were possible to convert \( a_1, \ldots, a_m \) into \( b_1, \ldots, b_{n-1} \), one could
complete the transformation by adding $b_n$ to $A$.

(c) If it were possible to convert $a_1, \ldots, a_{m-1}$ into $b_1, \ldots, b_{n-1}$, one could complete the transformation by replacing $a_m$ with $b_n$.

(d) If string $A$ is empty and $B$ is not empty then the conversion can only be done by inserting all remaining characters of $B$ into $A$.

(e) If string $B$ is empty and $A$ is not empty then the conversion can only be done by deleting all remaining characters in $A$ (assuming we can’t replace a character with the empty character).

Now let $A[i, j]$ be the minimum cost of transforming $a_1, \ldots, a_i$ into $b_1, \ldots, b_j$. The algorithm is:

\[
\text{MinimumEditDistance}(A, B) \n\]
\[
A[0, 0] = 0 
\]
\[
\text{for } i = 1 \text{ to } m \\
A[i, 0] = i \times 3 
\]
\[
\text{for } j = 1 \text{ to } n \\
A[0, j] = j \times 4 
\]
\[
\text{for } i = 1 \text{ to } m \\
\quad \text{for } j = 1 \text{ to } n \\
\quad \quad \text{if } a_i = b_j \text{ then } \\
\quad \quad \quad A[i, j] = A[i - 1, j - 1] \\
\quad \quad \quad \text{else } A[i, j] = \min(A[i - 1, j] + 3, A[i, j - 1] + 4, A[i - 1, j - 1] + 5) 
\]

Starting from $A[m, n]$, we can trace backwards through the table to determine which operations were performed at each step.

5. [This solution is courtesy Matthias Grabmair (in collaboration with Ian Wong).]

For $K_1, K_2, K_3, K_4, K_5$, our algorithm produces the following table of expected access times

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 0 & .5 & .6 & .85 & 1.4 & 2.15 \\
2 & 0 & .05 & .2 & .55 & 1.05 \\
3 & 0 & .1 & .4 & .9 \\
4 & 0 & .2 & .65 \\
5 & 0 & & 0.25 \\
\end{array}
\]
Given this table, we can produce the table of roots that correspond to the above access times:

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 1 & 1 & 1 \\
2 & 0 & 2 & 3 & 4 & 4 \\
3 & 0 & 3 & 4 & 4 \\
4 & 0 & 4 & 5 \\
5 & 0 & 5 \\
\end{array}
\]

Hence, the tree looks as follows:

\[
\begin{array}{c}
1 \\
\backslash \\
4 \\
/ \backslash \\
3 & 5 \\
/ \\
2
\end{array}
\]

The expected access time of the tree is found in the upper right corner of the table, 2.15. The tree can be constructed from the table above as follows. Start with the upper right corner (entry [2,5]) in the table of roots to see that node 1 is at the root of the tree. Hence the remaining nodes 2 through 4 are all in one subtree that is the right child of the root. The subtree has minimal weight when the root is 4, given in position [2,5] of the table. Since node 4 is the new root node, 5 inevitably becomes 4’s right child node. For the subtree that is the left child of node 4, we need to figure out which of 2 or 3 is the root. According to table cell [2,3] it is node 3.

6. We use dynamic programming to formulate an algorithm to compute the minimum polygon triangulation. Note that after the first “cut” is made, the original polygon \( P \) will be divided into two new polygons \( P' \) and \( P'' \). The triangulations \( P' \) and \( P'' \) are completely independent of each other. The cheapest way to triangulate \( P \) is the minimum of:

(a) For \( 3 \leq k \leq n-2 \), the length of \( (v_1, v_k) \) plus the length of \( (v_k, v_n) \) plus the cheapest way to triangulate \( P' = \{v_1, \ldots, v_k\} \) plus the cheapest way to triangulate \( P'' = \{v_k, \ldots, v_n\} \).
(b) The length of \((v_2, v_n)\) plus the cheapest way to triangulate \(P' = \{v_2, \ldots, v_n\}\).

(c) The length of \((v_1, v_{n-1})\) plus the cheapest way to triangulate \(P' = \{v_1, \ldots, v_{n-1}\}\).

Create a two-dimensional array \(A\). \(A[i, j] =\) minimum weight of triangulation of the polygon defined by \(v_i, \ldots, v_j\). Note we can initialize the diagonal (where \(i = j\)) to zeroes and will not use any entries below the diagonal (where \(i > j\)).

\[
\begin{align*}
&\text{for } i = n \text{ downto } 1 \text{ do} \\
&\quad \text{for } j = i + 1 \text{ to } n \text{ do} \\
&\quad \quad T[i, j] = \min( \\
&\quad \quad \quad \min_{i+2 \leq k \leq j-2} (d(i, k) + d(k, j) + A[i, k] + A[k, j]), \\
&\quad \quad \quad d(j, i + 1) + A[i + 1, j], \\
&\quad \quad \quad d(i, j - 1) + A[i, j - 1])
\end{align*}
\]

7. We present an algorithm to compute the maximum consecutive sum of \(n\) integers \(x_1, \ldots, x_n\). We define two functions:

- \(\text{MCS}(i) =\) Maximum Consecutive Sum of the first \(i\) integers.
- \(\text{MSS}(i) =\) Maximum Consecutive Sum of the first \(i\) integers that uses \(x_i\).

\(\text{MSS}(i)\) is computed by:

\[
\text{MSS}(i) = \max(0, \text{MSS}(i-1) + x_i)
\]

And \(\text{MCS}(n)\) is computed by:

\[
\text{MCS}(i) = \max(\text{MCS}(i-1), \text{MSS}(i-1) + x_i)
\]

By placing the two assignment statements inside a loop where \(i\) runs from 1 to \(n\), we get a linear time algorithm.

8. The input to this problem is a tree \(T\) with weights on the edges. The goal is to find the path in \(T\) with minimum aggregate weight. An path is a collection of adjacent vertices, where no vertex is used more than once.

Root the tree at an arbitrary node \(r\), and process the tree in postorder. We generalize the induction hypothesis to compute not only the shortest path in each subtree, but also the shortest path with one endpoint being the root of the subtree. Consider an arbitrary node \(v\) with branches to \(k\) descendants \(w_1, w_2, \ldots, w_k\). For each such node \(v\) the algorithm computes the following information:
best(v) = the minimum weight of a path for the subtree rooted at v.
root-best(v) = the minimum weight of a path ending at v in the
subtree rooted v.

At node v, the algorithm first recursively computes best(w_i) and root-
best(w_i) for each descendant subtree. It then computes best(v) and root-
best(v) using the following recurrence relations that correspond to the two
cases identified above:

rootbest(v) = \min(0, \min_{i=1}^{k}(d(v, w_i) + rootbest(w_i)))

best(v) = \min(rootbest(v), \min_{i}(best(w_i)), \min_{i,j}(rootbest(w_i)+rootbest(w_j)+d(v, w_i)+d(v, w_j)))

Its not too hard (but not completely trivial) to see that this can be im-
plemented in linear time.

9. We present the following dynamic programming algorithm to compute
the AVL tree with minimum expected access time. The main idea is
to strengthen the inductive hypothesis to compute the AVL tree of each
height with the minimum expected access time. Define \( A[i, j, h] \) as the
minimized expected access time for a tree with height \( h \) on the keys
\( K_i, K_{i+1}, \cdots, K_j \). We then get the following recurrence:

\[
A[i, j, h] = \min_{i \leq r \leq j} \left( A[i, r - 1, h - 1] + A[r + 1, j, h - 2] + \sum_{a=i}^{j} p_a, \\
A[i, r - 1, h - 1] + A[r + 1, j, h - 1] + \sum_{a=i}^{j} p_a, \\
A[i, r - 1, h - 2] + A[r + 1, j, h - 1] + \sum_{a=i}^{j} p_a \right)
\]

Here \( r \) is the root of the new tree, and \( p_a \) is the probability of accessing
key \( K_a \). Note that if the tree is going to be of height \( h \) then one of its two
subtrees must be of height \( h - 1 \) and the other can be of height at most
\( h - 1 \). Since the tree is an AVL tree the heights of the two subtrees can
differ by at most 1. One can then get code by wrapping this assignment
statements in a loop for \( i \) from \( n \) to 1, a loop for \( j \) from \( i \) to \( n \), and a loop for
\( h \) from 0 to \( n \) (actually you can replace \( n \) here by something like \( \sqrt{2} \log n \)
if you know that AVL trees are balanced). The final minimum expected
depth can be found by taking the minimum over all \( h \) of \( A[1, n, h] \), and
the tree can be recovered the same way that it was for the problem on
computing the binary search tree with minimum expected access time.
10. We give dynamic programming algorithm to compute the non-overlapping collection of intervals $I_1, \ldots, I_n$ with maximum measure. We consider the intervals by increasing order of their left endpoints. The main idea is to strengthen the inductive hypothesis to compute the maximum measure non-overlapping collection of intervals with a particular rightmost interval, for all possible choices of rightmost interval. Define $S[i, j]$ as the total length of the longest non-overlapping interval set among the first $i$ intervals such that the right most interval in this set is the $j$th one. Then $S[i, j]$ is the maximum over all intervals $I_k$, such that the right endpoint of $I_k$ is to the left of the left endpoint of $I_j$, of $S[k, k]$ plus the length of $I_j$. By wrapping this in a loop for $i$ from 1 to $n$, and for $j$ from 1 to $i$, you get the algorithm. The maximum measure of a non-overlapping collection of intervals can then be found by taking the maximum $S[n, j]$ for all $j$.

11. No solution given.

12. No solution given.

13. We present a solution for the problem of determining if the values of $n$ items can be added and subtracted in such a way that $\sum_{k=1}^{n} (-1)^{x_k} v_k = 0$. We use dynamic programming. As pruning rule we remark that if we have two solutions with the same value we only need to keep one. Here we cannot discard solutions with negative values or with values larger than $L$ since subtraction is allowed. However, if we define $L = \sum_{i=1}^{n} v_i$, we know that any solution will have its value in $\{-L \ldots L\}$, and thus we will have at most $2L$ solutions to keep.

For $i = 1 \ldots n$ and $j = -L \ldots L$ let $A(i, j)$ be a boolean variable that indicates whether or not there is a solution with the first $i$ objects that has a value equal to $j$. That is, $A(i, j)$ is true if there is a solution to $\sum_{k=1}^{i} (-1)^{x_k} v_k = j$

where each $x_k$ is either 1 or 0.

With the initialization $A(0, 0) = T$ and $A(0, j) = F$ for $j \neq 0$, we can fill the table row by row using the following rule:

if $A(i, j)$ then $A(i+1, j + v_{i+1}) = T$ and $A(i+1, j - v_{i+1}) = T$.

That is, if there is a solution for $j$ at level $i$ then there is a solution for $j + v_{i+1}$ and $j - v_{i+1}$ at level $i + 1$.

At the end, the entry $A(n, 0)$ will indicate whether there is a solution for the problem or not. If you want you can then construct a solution by tracing a “path” back in the table starting at $A(n, 0)$. 

8
The size of the table is \( n \times 2L \). The time to fill an entry is constant. Therefore the total time is \( O(nL) \) which is polynomial in \( n + L \).

14. We present an algorithm for determining if there is a subset of \( n \) items whose total value is \( L \mod n \). We use dynamic programming. As pruning rule we remark that if we have two solutions with the same value modulo \( n \) we only need to keep one. Therefore there are at most \( n \) solutions to keep at each level.

For \( i = 1 \ldots n \) and \( j = 0 \ldots n - 1 \) let \( A(i, j) \) be a boolean variable that indicate whether or not there is a solution with the first \( i \) objects that has a value equal to \( j \mod n \). That is, \( A(i, j) \) is true if there is a solution to

\[
A(i, j) = \left( \sum_{k=1}^{i} x_k v_k \right) \mod n = j
\]

where each \( x_k \) is either 1 or 0.

With the initialization \( A(0, 0) = T \) and \( A(0, j) = F \) for \( j = 1 \ldots n - 1 \), we can fill the table row by row using the following recurrence relation:

\[
A(i, j) = A(i - 1, j) \text{ or } A(i - 1, (j - v_i) \mod n).
\]

That is, there is a solution modulo \( j \) at level \( i \) if there is a solution for \( j \) at level \( i - 1 \) or if you can add \( v_i \) to a solution at level \( i - 1 \) and get \( j \mod n \).

At the end, the entry \( A(n, L \mod n) \) will indicate whether there is a solution for the problem or not. If you want you can then construct a solution by tracing a “path” back in the table starting at \( A(n, L \mod n) \).

The size of the table is \( n^2 \). The time to fill an entry is constant. Therefore the total time is \( O(n^2) \) which is polynomial in \( n + L \).

15. We use dynamic programming for the problem of obtaining the maximum value from a subcollection (allowing repetition) of \( n \) items, subject to the restriction that the total weight of the set cannot exceed \( W \). As pruning rule we remark that if we have two solutions with the same weight we only need to keep the one with the highest value and that we only need to keep solutions with weight no greater than \( W \).

For \( i = 1 \ldots n \) and \( j = 0 \ldots W \) we compute \( A(i, j) \) = the maximum value of an assignment of the \( i \) first objects with weight bounded by \( j \). That is,

\[
A(i, j) = \max \sum_{k=1}^{i} x_k v_k \text{ subject to } \sum_{k=1}^{i} x_k w_k \leq j
\]

where each \( x_k \) is a non-negative integer.
Assuming \( A(0,j) = 0 \) for \( j = 0 \ldots W \), we can fill the table row by row using the following recurrence relation:

\[
A(i,j) = \max_{x_i=0 \ldots j/w_i} x_i v_i + A(i-1,j-x_i w_i)
\]

We only need to consider values of \( x_i \) in the range 0 \ldots j/w_i since any higher value would make \( j - x_i w_i \) negative. The final solution will be in \( A(n,W) \).

The size of the table is \( n \times W \). The time to fill an entry is \( O(W) \). Therefore the total time is \( O(n \times W^2) \) which is polynomial in \( n + W \).

16. [This solution is courtesy of Brian Wongchaowart.]

Let \( A \) be an array of size \((n+1) \times (L+1) \times (L+1)\) indexed from 0 and assign \( A[i][j][k] \) true if there is a subset \( S \) of \( v_1, \ldots, v_i \) such that

\[
\sum_{v \in S} v^3 = j \quad \text{and} \quad \prod_{v \in S} v = k.
\]

```c
for (j = 0; j <= L; j++)
    for (k = 0; k <= L; k++)
        A[0][j][k] = false;
A[0][0][1] = true;
for (i = 1; i <= n; i++) {
    for (j = 0; j <= L; j++)
        for (k = 0; k <= L; k++)
            A[i][j][k] = A[i - 1][j][k];
    if ((j - v[i]^3 >= 0) && (k % v[i] == 0) && A[i - 1][j - v[i]^3][k / v[i]])
        A[i][j][k] = true;
}
```

If \( A[n][j][k] \) is true for some \( j = k \), then a solution exists and can be reconstructed from the table as follows. (Below, \( x[i] \) is assigned 1 if \( v_i \) is included in \( S \), and 0 otherwise.)

```c
for (j = 0; j <= L; j++)
    if (A[n][j][j])
        break; // solution found
if (j <= L) {
    k = j;
    for (i = n; i >= 1; i--)
        if ((j - v[i]^3 >= 0) && (k % v[i] == 0))
            x[i] = 1;
```

11
Given $A[i - 1][j - v[i]^3][k / v[i]]$,

\[
\begin{align*}
&j = j - v[i]^3; \\
&k = k / v[i]; \\
&x[i] = 1; \quad \text{// include } v[i] \text{ in the subset} \\
&\text{else} \{ \\
&\quad x[i] = 0; \quad \text{// } v[i] \text{ not included} \\
&\}
\end{align*}
\]

Since the size of array $A$ is $O(nL^2)$, the algorithm runs in $O(nL^2)$ time.

17. We present a dynamic programming algorithm for the problem of determining if there are two subsets of $n$ rubies and emeralds that contain the same number of rubies, the same number of emeralds, and the same total value. Let $n_e$ and $n_r$ be the number of emeralds and rubies among the $n = n_e + n_r$ gems. If any of $n_e$, $n_r$ or $L$ is odd then clearly the problem has no solution. Otherwise we can determine whether there is a solution using dynamic programming.

For $i = 0 \ldots n$, $j = 0 \ldots n_e/2$, $k = 0 \ldots n_r/2$, and $\ell = 0 \ldots L$ let $A(i, j, k, \ell)$ be a boolean variable that indicate whether or not there is a subset of the first $i$ gems that contains exactly $j$ emeralds and $k$ rubies and has value $\ell$. The table is initialized to all False, except for $A(0, 0, 0, 0)$ which is initialized to True. We can fill the table row by row using the following code:

\[
\begin{align*}
&\text{for } i = 1 \ldots n \text{ do} \\
&\quad \text{for } j = 1 \ldots n_e/2 \text{ do} \\
&\quad \quad \text{for } k = 1 \ldots n_r/2 \text{ do} \\
&\quad \quad \quad \text{for } \ell = 0 \ldots L/2 \text{ do} \\
&\quad \quad \quad \quad \text{if } A(i - 1, j, k, \ell) \text{ then} \\
&\quad \quad \quad \quad \quad A(i, j, k, \ell) = \text{True} \\
&\quad \quad \quad \quad \quad \text{if } \text{gem}_i = \text{emerald} \text{ then} \\
&\quad \quad \quad \quad \quad \quad A(i, j + 1, k, \ell + v_i) \leftarrow \text{True} \\
&\quad \quad \quad \quad \quad \text{else} \\
&\quad \quad \quad \quad \quad \quad A(i, j, k + 1, \ell + v_i) \leftarrow \text{True}
\end{align*}
\]

There is a solution to the problem if at the end $A(n, n_e/2, n_r/2, L/2)$ is True. The time taken by the algorithm is $O(n^3 \times L)$ which is polynomial in $n + L$.

18. Problem not assigned

19. We use dynamic programming to find a subsequence of maximum aggregate cost (note that there is more than one reasonable way to construct a
dynamic programming algorithm here). We use the pruning method. Let the level \( l \) of the tree consist of all subsequences of \( T \) that match the first \( l \) letters in the pattern \( P \). The pruning rule is that if you have two solutions on the same level are of the same length and end at the same letter in \( T \), then you can prune the one of lesser cost. So let \( A[l, s] \) be the maximum cost of a subsequence of \( T \) equal to \( p_1, \ldots, p_l \), where the last letter of this subsequence is \( t_s \). We then get the following code

\[
\text{for } l = 1 \ldots k - 1 \text{ do} \\
\quad \text{for } s = 1 \ldots n \text{ do} \\
\quad \quad \text{if } A(l, s) \text{ is defined then} \\
\quad \quad \quad \text{for } r = s + 1 \ldots n \text{ do} \\
\quad \quad \quad \quad \text{if } t_r = p_{l+1} \text{ then} \\
\quad \quad \quad \quad \quad A[l + 1, r] = \max(A[l + 1, r], A[l, s] + c_r)
\]

20. We use dynamic programming to compute the optimal solution to the two taxi cab problem. We first note that when \( p_i \) is serviced one of the taxi is in \( p_i \) while the other taxi is in one of the location \( p_0 \ldots p_{i-1} \), where \( p_0 \) denotes the origin. Therefore at stage \( i \) we only need to keep one solution (the best one) for each possible location \( p_0 \ldots p_{i-1} \) of the second taxi.

For \( i = 1 \ldots n \) and \( j = 0 \ldots i - 1 \) let \( A(i, j) \) be the minimum cost of a routing that serves the points \( p_1 \ldots p_i \) and leaves one taxi in \( p_i \) and the other in \( p_j \). Given the best solutions for stage \( i \) we can compute the solutions for stage \( i + 1 \) by considering the following two possibilities:

(a) the taxi that was in \( p_i \) is used to serves \( p_{i+1} \) and the other taxi remains where it was. That is, for \( j = 0 \ldots i - 1 \),

\[ A(i + 1, j) = |p_ip_{i+1}| + A(i, j). \]

(b) the taxi in \( p_i \) remains at \( p_i \), and the other taxi serves \( p_{i+1} \). Only the solution with minimum cost is kept.

\[ A(i + 1, i) = \min_{k=0 \ldots i-1} |p_kp_{i+1}| + A(i, k), \]

Using the above relations it is easy to fill the table. At the end the minimum routing cost is given by the minimum of \( A(n, i) \), \( i = 0 \ldots n - 1 \). The actual routing can be reconstructed by tracing the “path” back in the table starting from the final cell.

Since there are \( n \) stages and each stage takes \( O(n) \) time, the total time is \( O(n^2) \) which is polynomial in \( n \).

21. [This solution is adapted from solutions given by Brian Wongchaowart and Daniel Cole.] Number the points on the line from left to right, so that \( x_1 \) is the leftmost point and \( x_n \) is the rightmost point. Without loss of generality, assume
that the origin is one of these points. Every feasible path begins at the origin and visits the points one by one. The set of all points visited so far must expand by one point to the left or right at a time until every point has been added. Within this set, the point that was added last may be either the leftmost point or the rightmost point. If the last visited point is to the right of the origin, then this point is the rightmost of the visited points. If the last visited point is to the left of the origin, then this point is the leftmost of the visited points. Thus we can think of a tree of feasible solutions where the up to $2^\ell$ solutions at level $\ell$ are all solutions for visiting exactly $\ell$ points, and each solution has two children depending on whether the next visited point is to the right or left of the currently visited points.

We now give the pruning rule. Consider two solutions at the same level that have the same last visited point. We know from the reasoning above that these two solutions have visited exactly the same points. The two characteristics that we care about these solutions are the total response times of the points that these two solutions have visited, and the response time of the last visited point (which is the length of the path/solution). We keep the solution that minimizes the total response time to visit the points so far plus $(n - \ell)$ times the length of the path. If you have two solutions at the same level with the same last point, where the difference in their paths lengths is $L$, then the term $(n - \ell)L$ represents the savings that the solution with the shorter path will incur for the response times of points that will be added in the future.

The outer loop of the code will iterate over levels. Define the array $Cost[\ell, k]$ be the the total response time of the path that visits exactly $\ell$ points and ends at $p_k$ that minimizes the total response time plus $n - \ell$ times the path length. Define the array $Length[\ell, k]$ be the the length of the path that visits exactly $\ell$ points and ends at $p_k$ that minimizes the total response time plus $n - \ell$ times the path length. The next loop loops over the possible values for $k$. If $Cost[\ell, k]$ is defined and $p_k$ is to the right of the origin, then the next two points that might be visited are $p_{k+1}$ and $p_{k-\ell}$. If

$$Cost[\ell, k] + d(p_k, p_{k+1}) + Length[\ell, k] + (n - \ell - 1)(d(p_k, p_{k+1}) + Length[\ell, k])$$

is less than

$$Cost[\ell + 1, k + 1] + (n - \ell - 1)Length[\ell + 1, k + 1]$$

then $Cost[\ell + 1, k + 1]$ is updated to $Cost[\ell, k] + d(p_k, p_{k+1}) + Length[\ell, k]$ and $Length[\ell + 1, k + 1]$ is updated to $Length[\ell, k] + d(p_k, p_{k+1})$.

If

$$Cost[\ell, k] + d(p_k, p_{k-\ell}) + Length[\ell, k] + (n - \ell - 1)(d(p_k, p_{k-\ell}) + Length[\ell, k])$$
is less than
\[ Cost[\ell + 1, k - \ell] + (n - \ell - 1)Length[\ell + 1, k - \ell] \]
then \( Cost[\ell + 1, k - \ell] \) is updated to \( Cost[\ell, k] + d(p_k, p_{k-\ell}) + Length[\ell, k] \)
and \( Length[\ell + 1, k - \ell] \) is updated to \( Length[\ell, k] + d(p_k, p_{k-\ell}) \). If \( p_k \) to the left of the origin (and hence the leftmost visited point) then the next two possible visited points are \( p_{k-1} \) and \( p_{k+\ell} \), and the code is similar to the above. So this gives use time \( O(n^2) \).

22. [This solution is courtesy of Brian Wongchaowart.] Number the trips in increasing order by date, so that \( d_1 < \ldots < d_n \). Let \( A \) be an array of size \((n + 1) \times (n + 1)\) indexed from 0. Assign to \( A[i][j] \) the minimum possible total cost of taking the first \( i \) trips when the last Bahncard was purchased just before trip \( j \) (if \( j = 0 \), no Bahncard is ever purchased). Assume for simplicity that a Bahncard is always purchased on the day of a trip. It is obvious that buying one on a day between two trips cannot result in a lower total cost.

```c
for (j = 0; j <= n; j++)
    A[0][j] = 0;
for (i = 1; i <= n; i++) {
    A[i][0] = A[i - 1][0] + f[i];
    for (j = 1; j <= i; j++) {
        if (i == j) {
            // buy Bahncard
            A[i][j] = INFINITY;
            for (k = 0; k < i; k++) {
                if (A[i - 1][k] + f[i] / 2 + BAHNCARD_COST < A[i][j])
            }
        } else if (i > j && d[i] - d[j] < ONE_YEAR) {
            // use discounted fare
        } else {
            // use full fare
            A[i][j] = A[i - 1][j] + f[i];
        }
    }
}
```

The optimal cost of all \( n \) trips is the minimum entry in row \( n \) of \( A \). The dates on which a Bahncard was purchased can be determined from \( A \) as below, where \( x[i] \) is assigned 1 if a Bahncard was purchased just before (on the day of) trip \( i \).
i = 0;  // last time Bahncard was bought
for (j = 1; j <= n; j++) {
    if (A[n][j] < A[n][i])
        i = j;
}
while (i > 0) {
    x[i] = 1;  // buy Bahncard just before trip i
    for (k = 0; k < i; k++) {
        if (A[i][i] == A[i - 1][k] + f[i] / 2 + BAHNCARD_COST)
            i = k;  // last time Bahncard was bought
    }
}

The algorithm runs in $O(n^2)$ time, since the “for $k$” loop runs only once for each $i$.

23. [This solution is courtesy of Brian Wongchaowart.] Assume for simplicity that $R_n > 0$ (discard any trailing $R_i$ such that $R_i = 0$). For consistency with previous problems let the first integer in sequence $R$ be numbered $R_1$, not $R_0$ as in the problem statement. Let $k = \min(k, n)$, since more server broadcasts than requests are unnecessary. Let $A$ be an array of size $(n + 1) \times (k + 1)$ indexed from 0. Assign to $A[i][j]$ the minimum total waiting time obtainable for the sub-problem consisting of satisfying requests $R_1, \ldots, R_i$ using $j$ server broadcasts, so that the optimal solution to the original problem has a total waiting time of $A[n][k]$.

for (j = 1; j <= k; j++)
    A[0][j] = 0;
for (i = 1; i <= n; i++) {
    for (j = 1; j <= k && j <= i; j++) {
        if (j == 1) {
            // broadcast at time $i + 1$
            A[i][j] = 0;
            for (m = 1; m <= i; m++)
                A[i][j] += R[m] * (i - m + 1);
        } else {
            A[i][j] = INFINITY;
            // find best time for broadcast $j - 1$
            for (b = j - 1; b < i; b++) {
                temp = A[b][j - 1];
                // also broadcast at time $i + 1$
                for (m = b + 1; m <= i; m++)
                    temp += R[m] * (i - m + 1);
            }
        }
    }
}
if (temp < A[i][j])
    A[i][j] = temp;
}
}
}

The optimal broadcast times can be determined from A as follows, where
x[i] is assigned 1 if there is a broadcast at time i.

j = k; // number of broadcasts remaining
i = n;
while (j > 0) {
    x[i + 1] = 1; // broadcast j
    if (j > 1) {
        // find time of broadcast j - 1
        for (b = j - 1; b < i; b++) {
            temp = A[b][j - 1];
            for (m = b + 1; m <= i; m++)
                temp += R[m] * (i - m + 1);
            if (temp == A[i][j])
                i = b;
        }
        j--;
    }
}

An examination of the loops shows that the algorithm runs in $O(n^2k)$
time.

24. [This solution is courtesy of Pengfei Li.]

Our goal is to develop a dynamic programming algorithm using the pruning
method for the problem of finding whether there are two partitions of
equal weight.

Let $W(B_i)$ donate the weight of box $B_i$, and $W(C)$ donate the weight of
sub-collection $C$.

Consider the tree where level $i$ consists of all possible sub collections of
$B_1, \ldots, B_i$.

Pruning Rule:

(a) If the node of some sub-collection is heavier (use the balance) than
the remaining sub-collection, delete this node
(b) If two nodes at same level have equal weight (use the balance), prune
out one of them.
Then we keep at most $\frac{1}{2}nL$ nodes in each tree level since the sum of the $n$ boxes is no more than $nL$.

Let $Q(i)$ be the ordered queue by weight of all available sub-collections in level $i$. Let $\bar{C}$ be the complement sub-collection of $C$.

If $n$ is 1 or 2, we can use the balance at most once to get the result.

We now give pseudocode for the algorithm for $n > 2$.

\begin{verbatim}
Use balance to test $\{B_1\}$ and $\{\bar{B}_1\}$,
If $W(\{B_1\}) > W(\{\bar{B}_1\})$ Return No Partition
If $W(\{B_1\}) = W(\{\bar{B}_1\})$ Return Partition Found

$Q(1) = (\{B_1\})$

For $i = 2$ to $n$
do
  $FirstQ = Q(i-1)$ //Assign a copy of $Q(i-1)$
  $SecondQ = ()$ //Define the second queue
  //Fetch each element from $FirstQ$ from light to heavy
  For each $C = Dequeue(FirstQ)$ do
    Use balance to test $C \cup \{B_i\}$ and $\bar{C} \cup \{\bar{B}_i\}$,
    //Add $C \cup \{B_i\}$ into $SecondQ$ if weight of $C \cup \{B_i\}$ is less than half
    If $W(C \cup \{B_i\}) < W(\bar{C} \cup \{\bar{B}_i\})$ Then Enqueue($SecondQ$, $C \cup \{B_i\}$)
    //If weight equals we already find the partition $C \cup \{B_i\}$
    Else If $W(C \cup \{B_i\}) = W(\bar{C} \cup \{\bar{B}_i\})$ return Partition Found

  //Now $FirstQ$ and $SecondQ$ are both ordered queue by weight,
  //Then merge the two queues and prune out queues of equal weight.
  $C_1 = Dequeue(FirstQ)$
  $C_2 = Dequeue(SecondQ)$
  While $FirstQ$ is not empty do
    Use balance to test $C_1$ and $C_2$,
    If $W(C_1) < W(C_2)$ Then
      Enqueue($Q(i)$, $C_1$)
      $C_1 = Dequeue(FirstQ)$
    Else If $W(C_1) > W(C_2)$ Then
      Enqueue($Q(i)$, $C_2$)
      $C_2 = Dequeue(SecondQ)$
    Else
      Enqueue($Q(i)$, $C_1$)
    $C_1 = Dequeue(FirstQ)$
    $C_2 = Dequeue(SecondQ)$

Return No Partition
\end{verbatim}

In this algorithm, once $W(C \cup \{B_i\}) = W(\bar{C} \cup \{\bar{B}_i\})$, we find the desired partition.
Now to compute the use of pan balance.

Inside the loop, each time we dequeue an element from FirstQ, we use the pan balance once, hence the number of uses of balance is $|FirstQ| = |Q(i-1)|$. Merging FirstQ and SecondQ need traversing all elements of FirstQ and SecondQ, which takes $|FirstQ| + |SecondQ| \leq 2|Q(i-1)|$ uses of balance. So in the loop we at most use $3|Q(i-1)|$ times of balance. Notice $|Q(i-1)| \leq \frac{1}{2} nL$. The total number of uses is $O(\sum_{i=1}^{n} 3|Q(i-1)|) = O(n * 3 * \frac{1}{2} nL) = O(n^2 L)$.

25. Problem not assigned

26. [This solution is courtesy of Pengfei Li.]

Assume the array $R$ consists of $n$ rows and $m$ columns, so the ending time is $n$ and number of pages is $m$. Here $m \leq k$.

Let $W(p, l_p)$ be the minimal total waiting time for page $p$ with at most $l_p$ broadcasts of page $p$ under the requests of $R_{1,p}, \ldots, R_{n,p}$. Here $l_1 + l_2 + \cdots + l_m = k$ and $l_i \geq 1$.

In the solution of problem 23, we have solved each $W(p, l_p)$ with time complexity $O(n^2 l_p)$.

Now we are to determine how to assign $l_i (1 \leq i \leq m)$ so that we can schedule minimum total waiting time. This is like Unbounded Knapsack Problem and we use tree to solve it.

Consider the tree where level $i$ consists of all assignments of $l_1, \ldots, l_i$.

The Pruning Rule:

(a) Eliminate the subtree rooted at any assignment of $l_1, \ldots, l_i$ with sum greater than $k$.

(b) If there are two subsets $A$ and $B$ at the same depth $i$, with the same sum of assignment $l_1, \ldots, l_i$, then eliminate the one of higher total waiting time. Here the total waiting time is $W(1, l_1) + \cdots + W(i, l_i)$.

Note that these two pruning rules mean that there are at most $k+1$ nodes left unpruned at any level.

Let $T(p, j)$ be minimum total waiting time we can obtain from an assignment of $l_1, \ldots, l_p$ with sum $l_1 + \cdots + l_p = j$. Notice we need to ensure $p \leq j$. We compute $T(p, j)$ as follows.
\(T(0, 0) = 0\)
For \(j = 1\) to \(k\) do \(T(0, j) = 0\)
For \(p = 1\) to \(m\) do
  For \(j = p\) to \(k\) do
    \(T(p, j) = \text{max } \text{infinity}\)
    For \(l = 1\) to \(j - p + 1\) do
      Compute \(W(p, l)\) by calling the solution of problem 23 under the requests of \(R_{1,p}, \ldots, R_{n,p}\)
      If \(T(p, j) > T(p - 1, j - l) + W(p, l)\)
        \(T(p, j) = T(p - 1, j - l) + W(p, l)\) (Find the minimum waiting time)
Output \(T(n, k)\)

To reconstruct the optimal broadcast arrangement, we should use another array \(R(p, j)\) to record the \(l\) in the inner loop where the current solution comes from. Finally trace-back from optimal \(R(n, k)\) to find the best assignment of \(l_1, \ldots, l_m\) and refer to solution of problem 23 to find the broadcast schedule of page \(p\) with at most \(l_p\) broadcasts.

Now we compute the time complexity. Calling each \(W(p, l)\) takes time \(O(n^2l)\), so traversing all possible \(l\) from 1 to \(j - p + 1\) takes \(\sum_{l=1}^{j-p} O(n^2l) + O(1)] = O(n^2(j-p)^2]\).

Then consider \(j\) traversing from \(p\) to \(k\). It takes time \(\sum_{j=p}^{k} O[n^2(j-p)^2] = O[n^2(k-p)^3]\). (By \(\sum_{i=1}^{h} i^2 = \frac{h(h+1)(2h+1)}{6}\))

At last, consider \(p\) traversing from 1 to \(m\). It takes time \(\sum_{p=1}^{m} O(n^2(k-p)^3)] = O[n^2(k^4-(k-m)^4)]\). (By \(\sum_{i=1}^{h} i^3 = \frac{1}{4}i^2(i+1)^2\))

We know \(k \geq m\), Hence \(O[n^2(k^4-(k-m)^4)] = O[n^2(k^2+(k-m)^2)(k^2-(k-m)^2)] = O[n^2k^2m(2k-m)] = O(n^2k^3m)\), when \(m\) is the number of different pages.