Sample Solutions to Homework #3

1. (15)

```
FIBONACCI(n):
    let fib[0...n] be a new array
    fib[0] = fib[1] = 1
    for i = 2 to n
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

"FIBONACCI" directly implements the recurrence relation of the Fibonacci sequence. Each number in the sequence is the sum of the two previous numbers in the sequence. The running time is clearly \(O(n)\).

The subproblem graph consists of \(n+1\) vertices, \(v_0, v_1, ..., v_n\). For \(i = 2, 3, ..., n\), vertex \(i\) has two leaving edges: to vertex \(v_{i-1}\) and to vertex \(v_{i-2}\). No edges leave vertices \(v_0\) or \(v_1\). Thus, the subproblem graph has \(2n - 2\) edges.

2. (15)

(a) The \(m\) and \(s\) tables computed by MATRIX-CHAIN-ORDER for \(n = 6\) and the sequence of dimensions < 5, 10, 3, 12, 5, 50, 6 > are shown in Figure 1. The minimum number of scalar multiplications to multiply the six matrices is \(m[1, 6] = 2010\) and its corresponding parenthesization is \(((A_1A_2)(A_3A_4)(A_5A_6))\).

![Figure 1: The m and s tables computed by MATRIX-CHAIN-ORDER for n = 6 and the sequence of dimensions < 5, 10, 3, 12, 5, 50, 6 >.](image)

(b) The \(c\) and \(b\) table computed by LCS-LENGTH on the sequence \(X = < 1, 0, 1, 0, 1, 0, 1 >\) and \(Y = < 0, 1, 0, 1, 1, 0, 1, 0 >\) is shown in Figure 2. The longest common subsequence of \(X\) and \(Y\) is \(< 1, 0, 1, 0, 1, 0 >\).

![Figure 2: The c and b table computed by LCS-LENGTH.](image)

(c) Tables \(e\), \(w\), and \(root\) computed by OPTIMAL-BST for the given probabilities are shown in Figure 3(a), (b) and (c). The lowest expected search cost of any binary search tree for the given probabilities is 3.12 and the corresponding structure is shown in Figure 3(d).
Figure 2: The \( a \) and \( b \) table computed by LCS-LENGTH on the sequence \( X = < 1, 0, 0, 1, 0, 1, 0 > \) and \( Y = < 0, 1, 0, 1, 0, 1, 0 > \).

3. (10)

Let the best currency exchange rate from currency \( i \) to currency \( j \) be \( s_{ij} \), and if \( c_k = 0 \) for all \( k = 1, 2, ..., n \), then the optimal substructure is:

\[
s_{ij} = \max_{m} s_{im} s_{mj}
\]

where \( s_{im} = \max_{p} s_{ip} s_{pm} \), \( s_{mj} = \max_{q} s_{mq} s_{qj} \) and so on, which means that we can trade currency \( j \) by making a sequence of trades through currency \( m \) and the exchange rate is also the best. However, if \( c_k \) are arbitrary values, then the commissions may increase during the sequence of trades. So the problem of finding the best sequence of exchanges from currency \( 1 \) to currency \( n \) does not necessarily exhibit optimal substructure.

4. (10)

Let the array \( X[n] \) to record the input sequence of length \( n \). During the algorithm, we maintain two arrays as we process each element in \( X[n] \):

- \( \text{BestEnd}[j] \): stores the position \( k \) of the smallest value \( X[k] \) such that there is an increasing subsequence of length \( j \) ending at \( X[k] \) on the range \( j \leq k \leq i \).

- \( P[k] \): stores the position of the predecessor of \( X[k] \) in the longest increasing subsequence ending at \( X[k] \).

Then, we can derive the longest monotonically increasing subsequence by using the following algorithm.

- input: \( X[n] \)

```latex
\begin{verbatim}
Longest-Monotonically-Increasing-Subsequence(X)
L = 0
for i = 1 to n
    binary search for the largest positive j \leq L such that X[BestEnd[j]] \leq X[i]
    (or set j = 0 if no such value exists)
    P[i] = BestEnd[j]
if j == L or X[i] < X[BestEnd[j + 1]]
    BestEnd[j + 1] = i
    L = max(L, j + 1)
\end{verbatim}
```
As the end of the algorithm, the longest monotonically increasing subsequence can be generated with the form \( \ldots, X^{[P[P[BestEnd[L]]]], X[P[BestEnd[L]]], X[BestEnd[L]]} \). Since the algorithm performs a binary sort in each iteration (for each element in the input sequence), this algorithm can be done by \( O(n\log n) \) time.

5. (20)

(a) The full table is shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

(b) The recurrence relation of \( LCSuff(m, n) \) is shown in Equation 1.

\[
LCSuff(m, n) = \begin{cases} 
 LCSuff(m - 1, n - 1), & \text{if } m > 0, n > 0, \text{ and } x_m = y_n, \\
0, & \text{otherwise}.
\end{cases} \tag{1}
\]

(c) As shown in Equation 2, \( LCSubStr(m, n) \) can be defined in terms of \( LCSuff(i, j) \).

\[
LCSubStr(m, n) = \max_{1 \leq i \leq m, 1 \leq j \leq n} LCSuff(i, j) \tag{2}
\]

(d) Derive \( LCSuff(i, j) \forall 1 \leq i \leq m, 1 \leq j \leq n \), and then derive \( \max_{1 \leq i \leq m, 1 \leq j \leq n} LCSuff(i, j) \) \( (LCSubStr(m, n) \text{ by (c)}) \). Time complexity: \( O(mn) \). Space complexity: \( O(mn) \).

6. (15)

(a) The optimal substructure:

\[
P(i, j) = \begin{cases} 
 1, & \text{if } i = 0, 0 < j < n, \\
 0, & \text{if } j = 0, 0 < i < n, \\
 NIL, & \text{if } i = 0, j = 0, \\
p \times P(i - 1, j) + (1 - p) \times P(i, j - 1), & \text{if } 0 < i < n, 0 < j < n.
\end{cases}
\]
(b) The probability: 
\[ P(2, 2) = 0.4 \times P(1, 2) + (1 - 0.4) \times P(2, 1) = 0.4 \times 0.64 + 0.6 \times 0.16 = 0.352. \]

(c) An algorithm is given:

```
CHAMPION-PREDICTION(C, n, p):
    t = n - 1
    for k = 1 to t
        C[0, k] = 1
        C[k, 0] = 0
    for j = 1 to t
        for i = 1 to t
            C[i, j] = p \times C[i - 1, j] + (1 - p) \times C[i, j - 1]
    return C[t, t]
```

Because \( C \) is an \( n \times n \) matrix, it takes \( O(n^2) \) time and space complexities to fill out \( C \).

7. (10)

The counterexample is shown in Table 2. Let the given rod length be 4. According to a greedy strategy,

<table>
<thead>
<tr>
<th>Table 2: Full table</th>
<th>length i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>price ( p_i )</td>
<td>1</td>
<td>20</td>
<td>33</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>( p_i/i )</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

we first cut out a rod of length 3 for a price of 33, which leaves us with a rod of length 1 of price 1. The total price for the rod is 34. The optimal way is to cut it into two rods of length 2 each fetching us 40 dollars.

8. (10)

The solution is based on the optimal substructure observation in the text: Let \( i \) be the highest-numbered item in an optimal solution \( S \) for \( W \) pounds and item \( 1, \ldots, n \). Then \( S' = S - i \) is an optimal solution for \( W - w_i \) pounds and item \( 1, \ldots, i - 1 \), and the value of the solution \( S \) is \( v_i \) plus the value of the subproblem solution \( S' \).

We can express this relationship in the following formula: define \( c[i, w] \) to be the value of the solution for item \( 1, \ldots, i \) and maximum weight \( w \). Then

\[
    c[i, w] = \begin{cases} 
    0 & \text{if } i = 0 \text{ or } w = 0, \\
    c[i - 1, w] & \text{if } w_i > w, \\
    \max(v_i + c[i - 1, w - w_i], c[i - 1, w]) & \text{if } i > 0 \text{ and } w \geq w_i 
    \end{cases}
\]

Notice that the last case determines whether or not the \( i \)th element should be included in an optimal solution. The inputs of the algorithm are the maximum weight \( W \), the number of items \( n \), and the two sequences \( v = < v_1, v_2, \ldots, v_n > \) and \( w = < w_1, w_2, \ldots, w_n > \). Table \( c[0..n, 0..W] \) stores the \( c[i, j] \) values whose entries are computed in row-major order. (That is, the first row of \( c \) is filled in from left to right, then the second row, and so on.) At the end of the computation, \( c[n, W] \) contains the maximum value the thief can take. We can use the recursion to create a straightforward dynamic programming algorithm:
The set of items involved in an optimal solution can be derived by tracing the $c$ table starting at $c[n, W]$. If $c[i, w] = c[i - 1, w]$, then item $i$ is included in the solution, and we continue the tracing process with $c[i - 1, w]$. Otherwise, item $i$ is included in the solution, and we continue the tracing process with $c[i - 1, w - w_i]$. The above algorithm takes $\Theta(nW)$ time in total:

- $\Theta(nW)$ time to fill in the $c$ table: $(n + 1) \cdot (W + 1)$ entries, and each requires $\Theta(1)$ time to compute.
- $\Theta(n)$ time to trace the solution (since it starts in row $n$ of the table and moves up one row at each step).

9. (10) The optimal Huffman code based on the first 8 Fibonacci numbers is shown in Figure 4.

The optimal Huffman code for the first $n$ Fibonacci numbers is generalized as,

\[
\begin{cases}
\begin{array}{l}
\text{1...1}, \text{1st number,} \\
\text{1...10}, \text{i}^{th} \text{ number where } 2 \leq i \leq n
\end{array}
\end{cases}
\]

or,

\[
\begin{cases}
\begin{array}{l}
\text{0...0}, \text{1st number,} \\
\text{0...01}, \text{i}^{th} \text{ number where } 2 \leq i \leq n
\end{array}
\end{cases}
\]
10. (20)

We first describe the greedy algorithm for the case where coins in denominations \(w_0, w_1, \ldots, w_k\) are available. Suppose \(w_0 < w_1 < \ldots < w_k\). The greedy algorithm is as follows:

\[
\text{CHANGE}(n) \\
1 \quad N \leftarrow n \\
2 \quad i \leftarrow k \\
3 \quad \text{while } N > 0 \text{ do} \\
4 \quad x \leftarrow \lfloor N/w_i \rfloor \\
5 \quad \text{if } x > 0 \\
6 \quad \text{print "} x \text{ coins of } w_i \text{"} \\
7 \quad N \leftarrow N - xw_i \\
8 \quad i \leftarrow i - 1
\]

We first make some general observations. If the algorithm gets a solution with \(\alpha_0, \alpha_1, \ldots, \alpha_k\) coins of \(w_0, w_1, \ldots, w_k\), then \(n = \alpha_0 w_0 + \ldots + \alpha_k w_k\). Clearly, \(\alpha_0, \alpha_1, \ldots, \alpha_k\) should also be an optimal solution of coins for \(n - \alpha_k w_k\) if \(\alpha_0, \alpha_1, \ldots, \alpha_k\) is an optimal choice for \(n\). Moreover, \(n - \alpha_k w_k < w_k\) due to the algorithm above.

In general, \(n - \sum_{i=1}^{k} \alpha_i w_i < w_1\). If the optimal solution, say \(\beta_0, \beta_1, \ldots, \beta_k\), differs from \(\alpha_0, \alpha_1, \ldots, \alpha_k\) first in \(l\)th coefficient, i.e., \(\alpha_i = \beta_i\) for \(i > l\), then \(\alpha_i > \beta_i\) and \(n - \sum_{i=1}^{k} \alpha_i w_i \geq w_l\). It is easy to see that there will be a set of coins adding up to \(p\) where \(w_l \leq p < w_l + w_{l-1}\) after the coins of \(w_l, \ldots, w_k\) have been deleted. (One can construct such a \(p\) by randomly adding one coin at a time to a pile till the value of the pile is at least \(p\). Since all coins are of value less or equal to \(w_{l-1}\), the result follows.)

Denote \(n - \sum_{i=1}^{k} \beta_i w_i\) by \(r_l\) - the residue after removing all coins of value more than \(w_{l-1}\).

(a) (5) Let \(w_0 = 1, w_1 = 5, w_2 = 10, w_3 = 25\).

It is easy to see that CHANGE yields the right solution if \(w_0\) is the only denomination. Suppose the algorithm finds a solution \(\alpha_0, \ldots, \alpha_3\) which is different from the optimal solution \(\beta_0, \ldots, \beta_3\). Let \(l\) be the first position where the solutions differ, by the earlier terminology. \(\alpha_l > \beta_l\) as the solution is optimal if \(w_0\) is the only denomination. If \(l = 1\), then \(\beta_0, \beta_1\) is a better solution than \(\alpha_0, \alpha_1\) for \(r_2\). By the above observation, there will be a set of coins adding up to \(p\), where \(w_l \leq p < w_l + w_{l-1}\) in the solution for \(r_1\), i.e, \(5 \leq p < 6 \Rightarrow p = 5\). But this can be replaced by one coin of denomination 5. Thus, \(\beta_0, \beta_1\) is not optimal. Similar arguments are used to find contradictions for \(k = 2, 3\).

(b) (5) Let \(w_i = c^i\) for \(0 \leq i \leq k\).

We prove the optimality of CHANGE by induction on \(k\).

If \(k = 0\), then only coins of denomination 1 are available. The algorithm clearly find the optimal solution for this situation.

Suppose the algorithm is optimal for some \(k\). Add the new denomination \(w_{k+1} = c^{k+1}\). Suppose the algorithm finds the solution \(\alpha_0, \ldots, \alpha_{k+1}\) and the optimal is \(\beta_0, \ldots, \beta_{k+1}\). Let the first position where they differs be \(l\). If \(l < k + 1\) then the algorithm does not find the optimal solution for \(r_l + 1\), which contradicts the assumption. If \(l = k + 1\) (i.e \(\beta_{k+1} < \alpha_{k+1}\)), then \(r_{k+1}\) has the optimal solution \(\beta_0, \ldots, \beta_k\). Since \(r_{k+1} \geq c^{k+1}\), we must have \(\beta_k \geq c\). But then the \(\beta_k\) coins of denomination \(c^k\) can be replaced by \(\beta_k - c\) coins of \(c^k\) and one of \(c^{k+1}\) reducing the numbers of coins by \(c - 1 > 0\). Thus \(\beta_0, \ldots, \beta_k\) cannot be an optimal solution for \(r_{k+1}\).

(c) (5) Let \(w_0 = 1, w_1 = 5, w_2 = 7, n = 25\). CHANGE finds the solution \(\alpha_2 = 3, \alpha_1 = 0, \alpha_0 = 4\), which are in total 7 coins. However, the optimal solution is \(\beta_2 = 0, \beta_1 = 5, \beta_0 = 0\), which are in total only 5 coins.

(d) (5) Assume the \(k\) kinds of coins of denominators are \(w_0, w_1, \ldots, w_{k-1}\). Then solving CHANGE(n-1) will give the solution.

11. (10)
(a) (5) The greedy choice is that the thief always takes as valuable a load as possible. For any optimal solution which is not the result of greedy choices, if we replace as many items per pound as possible with the most valuable items being left, the new total value must be larger than or equal to the original total value. Thus, this problem has the greedy-choice property.

(b) (5) Consider that we remove a weight \( w \) of item \( j \) from an optimal load. The remaining load must be the most valuable load weighing at most \( W - w \) that the thief can take from the \( n - 1 \) original items plus item \( j \) with \( w_j - w \) pounds. Otherwise, there were a more valuable solution to the subproblem. We could use this better solution to get a better solution containing item \( j \) of weight \( w_j \) for the full problem, contradicting our supposition of optimality. That is, the fractional knapsack problem exhibit the optimal-substructure property.

12. (20) We first describe the greedy algorithm for the case where the convenience store from Taipei to Kaohsiung are denoted as a set \( S = \{c_1, c_2, \ldots, c_n\} \) and the distance between each pair of convenience stores are also denotes as \( \{d_1, d_2, \ldots, d_n\} \) and \( d_i = |c_i - c_{i-1}|. \) The greedy algorithm is as follows:

\[
\begin{align*}
\text{FIND STOP}(n) \\
1 \quad D &\leftarrow 0 \\
2 \quad A &\leftarrow \emptyset \\
3 \quad \text{for } i \leftarrow 0 \text{ to } n \text{ do} \\
4 \quad D &\leftarrow D + d_{i+1} \\
5 \quad \text{if } D > 20 \\
6 \quad A &\leftarrow S \cup \{i\} \\
7 \quad D &\leftarrow 0 \\
8 \quad \text{return } A
\end{align*}
\]

Greedy-choice property:
Suppose \( A \subseteq S \) is an optimal solution by the greedy algorithm, and the first stop which is picked is \( c_i. \) Suppose here exists another optimal solution \( B \subseteq S \) and its first stop is not \( c_i. \) Since \( A \) is greedy, the first stop \( c_i \) of \( A \) is farther than \( c_j \) which is the first stop of \( B. \) Also, both \( A \) and \( B \) are optimal, the number of convenience stores are the same. Therefore, we can exchange \( c_j \) with \( c_i, B \) is also an optimal solution.

Optimal substructure:
Suppose \( A \subseteq S \) is an optimal solution which needs \( k \) stops by the greedy algorithm, then \( A' = A \setminus \{c_i\} \) is also an optimal solution to \( S' = \{c_j \in S : j > i\} \) and \( |A'| = k - 1. \) If \( A' \) is not an optimal solution to \( S', \) we can find a better solution \( A'' \) which contains less stops \( k - 2 \) than \( A'. \) Then, \( A'' \cup c_i \) would be a better solution which only needs to take \( k - 1 \) stops than \( A' \cup c_i = A \) to \( S, \) contradicting to the original claim that \( A \) is an optimal solution to \( S. \)

13. (15) Let \( \tilde{c}(D_i) \) be the amortized cost of performing one operation of the sequence.

The three conditions below conclude that \( \tilde{c}(D_i) \) is constant, where \( 1 \leq i \leq n. \)

- For a perfect square case (i.e., \( \sqrt{i} = t \in N): \)

\[
\begin{align*}
\tilde{c}(D_i) &= c(D_i) + \Phi(D_{i+1}) - \Phi(D_i) \\
&= 2\sqrt{i} + (i + 1 - (\lfloor \sqrt{i} \rfloor)^2) - (i - (\lfloor \sqrt{i-1} \rfloor)^2) \\
&= 2\sqrt{i} + 1 + [i - (\sqrt{i} - 1)^2] \\
&= 2\sqrt{i} + 1 - (2\sqrt{i} - 1) \\
&= 2t + 1 - (2t - 1) \\
&= 2.
\end{align*}
\]
• For $\sqrt{i-1} = t \in N$:

$$
\hat{c}(D_i) = c(D_i) + \Phi(D_{i+1}) - \Phi(D_i) \\
= 2 + (i + 1 - (\lceil \sqrt{i} \rceil)^2) - (i - (\lceil \sqrt{i-1} \rceil)^2) \\
= 2 + (i + 1 - (\lceil \sqrt{i} \rceil)^2) - (i - (\lceil \sqrt{i-1} \rceil)^2) \\
= 3.
$$

• Neither of $i$ and $i - 1$ is a perfect square:

Therefore, $\exists t \in N, s.t. [\sqrt{i}] = [\sqrt{i-1}] = t$.

$$
\hat{c}(D_i) = c(D_i) + \Phi(D_{i+1}) - \Phi(D_i) \\
= 2 + (i + 1 - (\lceil \sqrt{i} \rceil)^2) - (i - (\lceil \sqrt{i-1} \rceil)^2) \\
= 2 + (i + 1 - t^2) - (i - t^2) \\
= 3.
$$

14. (10)

The potential analysis has 5 different cases:

case 1: $\alpha_{i-1} < \frac{1}{2}$ and $\alpha_i > \frac{1}{2} \Rightarrow 2num_{i-1} < size_{i-1}$ \& $size_{i-1} = size_i$

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= 1 + [2num_i - size_i] - [2num_{i-1} - size_{i-1}] \\
= 1 + (size_{i-1} - 2num_{i-1}) + 2 - (size_{i-1} - 2num_{i-1}) \\
= 3.
$$

case 2: $\alpha_{i-1} < \frac{1}{2}$ and $\alpha_i < \frac{1}{2} \Rightarrow size_i = \frac{3}{2}size_{i-1}, 2num_{i-1} < size_{i-1}$ \& $3num_i < size_{i-1} \Rightarrow 2num_i < size_i$

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= (num_i + 1) + (size_i - 2num_i) - (size_{i-1} - 2num_{i-1}) \\
= num_i + 1 + size_i - 2num_i - \frac{3}{2}size_i + 2num_i + 2 \\
= 3 + num_i - \frac{1}{2}size_i \\
< 3.
$$

case 3: $\alpha_{i-1} \geq \frac{1}{2}$ and $\frac{1}{2} < \alpha_i < \frac{1}{2} \Rightarrow size_i = size_{i-1}$ \& $2num_{i-1} \geq size_{i-1}$

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= 1 + [2num_i - size_i] - [2num_{i-1} - size_{i-1}] \\
\leq 1 + [2num_{i-1} - size_{i-1}] + 2 - [2num_{i-1} - size_{i-1}] \\
= 3.
$$

case 4: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i \geq \frac{1}{2} \Rightarrow 2num_i \geq size_i$ \& $2num_{i-1} \geq size_{i-1}$

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= 1 + 2num_{i-1} - 2 - size_{i-1} - 2num_{i-1} + size_{i-1} \\
= -1.
$$

case 5: $\alpha_{i-1} \geq \frac{1}{2}$ and $\alpha_i < \frac{1}{2} \Rightarrow 2num_{i-1} \geq size_{i-1}, 3num_i < size_{i-1}$ \& $size_i = \frac{3}{2}size_{i-1} \Rightarrow \frac{3}{2}size_i - 2 \leq 2num_i < size_i$ \& $size_i < 4$

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}
$$
Thus, when the \( i \)th operation is a TABLE-DELETE, the amortized cost is bounded by a constant.

15. (15)

(a) (5) The SEARCH operation can be performed by searching each of the individually sorted arrays. Since all the individual arrays are sorted, searching one of them using a binary search algorithm takes \( O(n \log m) \) time, where \( m \) is the size of the array. In an unsuccessful search, the time is \( \Theta(\log m) \).

In the worst case, we may assume that all the arrays \( A_0, A_1, \ldots, A_{k-1} \) are full, \( k = \lceil \log (n + 1) \rceil \), and we perform an unsuccessful search. The total time taken is

\[
T(n) = \Theta((2 + 1) \sum_{i=0}^{k} 2^i) = \Theta(2^k) = \Theta(n)
\]

Thus, the worst-case running time is \( \Theta(\log^2(n)) \).

(b) (5) We create a new sorted array of size 1 containing the new element to be inserted. If array \( A_0 \) (which has size 1) is empty, then we replace \( A_0 \) with the new sorted array. Otherwise, we merge sort the two arrays into another sorted array of size 2. If \( A_1 \) is empty, then we replace \( A_1 \) with the new array; otherwise we merge sort the arrays as before and continue. Since array \( A_i \) is of size \( 2^i \), if we merge sort two arrays of size \( 2^i \) each, we obtain one of size \( 2^{i+1} \), which is the size of \( A_{i+1} \). Thus, this method will result in another list of arrays in the same structure that we had before.

Let us analyze its worst-case running time. We will assume that merge sort takes \( 2m \) time to merge two sorted lists of size \( m \) each. If all the arrays \( A_0, A_1, \ldots, A_{k-2} \) are full, then the running time to fill array \( A_{k-1} \) would be

\[
T(n) = 2(2^0 + 2^1 + \ldots + 2^{k-2}) = \Theta(n)
\]

Therefore, the worst-case time to insert an element into this data structure is \( \Theta(n) \).

However, let us now analyze the amortized running time. Using the aggregate method, we compute the total cost of a sequence of \( n \) inserts, starting with the empty data structure. Let \( r \) be the position of the rightmost 0 in the binary representation \( (n_{k-1}, n_{k-2}, \ldots, n_0) \) of \( n \), so that \( n_j = 1 \) for \( j = 0, 1, \ldots, r - 1 \). The cost of an insertion when \( n \) items have already been inserted is

\[
\sum_{j=0}^{r-1} 2 \cdot 2^j = O(2^r).
\]
Furthermore, $r = 0$ half the time, $r = 1$ a quarter of the time, and so on. There are at most $[n/2^r]$ insertions for each value of $r$. The total cost of the $n$ operation is therefore bounded by

$$O \left( \sum_{r=0}^{\lfloor \lg(n+1) \rfloor} \left( \left\lfloor \frac{n}{2^r} \right\rfloor \right) 2^r \right) = O(n \lg n).$$

The amortized cost per INSERT operation, therefore is $O(\lg n)$.

We can also use the accounting method to analyze the running time. We can charge $\$k$ to insert an element. $\$1$ pays for the insertion, and we put $\$(k - 1)$ on the inserted item to pay for it being involved in merges later on. Each time it is merged, it moves to a higher-indexed array, i.e., from $A_i$ to $A_{i+1}$. It can move to a higher-indexed array at most $k - 1$ times, and so the $\$(k - 1)$ on the item suffices to pay for all the times it will ever be involved in merges. Since $k = \Theta(\lg n)$, we have an amortized cost of $\Theta(\lg n)$ per insertion.

(c) (5) DELETE($x$) will be implemented as follows:

1. Find the smallest $j$ for which the array $A_j$ with $2^j$ elements is full. Let $y$ be the last element of $A_j$.
2. Let $x$ be in the array $A_i$. If necessary, find which array this is by using the search procedure.
3. Remove $x$ from $A_i$ and put $y$ into $A_i$. Then move $y$ to its correct place in $A_i$.
4. Divide $A_j$ (which now has $2^j - 1$ elements left): The first element goes into array $A_0$, the next 2 elements go into array $A_1$, the next 4 elements go into array $A_2$, and so forth. Mark array $A_j$ as empty. The new arrays are created already sorted.

The cost of DELETE is $\Theta(n)$ in the worst case, where $i = k - 1$ and $j = k - 2$: $\Theta(\lg n)$ to find $A_j$, $\Theta(\lg^2 n)$ to find $A_i$, $\Theta(2^i) = \Theta(n)$ to put $y$ in its correct place in array $A_i$, and $\Theta(2^j) = \Theta(n)$ to divide array $A_j$.

The following sequence of $n$ operations, where $n/3$ is a power of 2, yield an amortized cost that is no better: perform $n/3$ INSERT operations, followed by $n/3$ pairs of DELETE and INSERT. It costs $O(n \lg n)$ to do the first $n/3$ INSERT operations. This creates a single full array. Each subsequent DELETE/INSERT pair costs $\Theta(n)$ for the DELETE to divide the full array and another $\Theta(n)$ for the INSERT to recombine it. The total is then $\Theta(n^2)$, or $\Theta(n)$ per operation.

16. (0)

See Problem 1.

17. (15) (HW2 Exercise 15.2-1 (page 378).)

The $m$ and $s$ tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the sequence of dimensions $< 5, 10, 3, 12, 5, 50, 6 >$ are shown in Figure 5. The minimum number of scalar multiplications to multiply the six matrices is $m[1, 6] = 2010$ and its corresponding parenthesization is $((A_1.A_2)((A_3.A_4)(A_5.A_6)))$.

![Figure 5: The $m$ and $s$ tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the sequence of dimensions $< 5, 10, 3, 12, 5, 50, 6 >$.](image)

18. (40) DIY.