Problem 1

a. The maximum sum is $3 + 7 + 4 + 9 = 23$.

b. Let $T$ be a triangle of numbers with $r$ rows, where $T[i, j]$ is the number in the $ith$ row of $T$ that is $j$ from the left, starting from 1 (e.g. $T[3, 2]$ in the example triangle is 4). We define an algorithm defined by the following recurrence relation:

$$S[i, j] = \begin{cases} T[i, j] & \text{if } i = r \\ T[i, j] + \max(T[i + 1, j], T[i + 1, j + 1]) & \text{if } i < r \end{cases}$$

where $1 \leq i \leq r, 1 \leq j \leq i$ and $S[i, j]$ is the maximum sum from row $r$ of $T$ to $T[i, j]$.

$S[i, j]$ is an $r \times r$ table. We will fill only the bottom half of the table (we only need $i \geq j$), starting at the bottom row and working across rows to the top. The maximum sum in $T$ will be stored at $S[1, 1]$.

Algorithm:

\begin{algorithm}
\caption{Max-Sum($T$)}
1 $r \leftarrow T.numRows$
2 $S \leftarrow n \times n$ matrix
3 for $j \leftarrow 1$ to $r$
4 \hspace{1em} do
5 \hspace{2em} $S[r, j] \leftarrow T[r, j]$
6 \hspace{1em} for $i \leftarrow r - 1$ to 1
7 \hspace{2em} do
8 \hspace{3em} for $j \leftarrow 1$ to $i$
9 \hspace{4em} do
10 \hspace{5em} $S[i, j] \leftarrow T[i, j] + \max(T[i + 1, j], T[i + 1, j + 1])$
11 return $S[1, 1]$
\end{algorithm}

c. We can prove that the algorithm is correct by showing that the recurrence relation works and each $S[i, j]$ is the maximum sum from row $r$
of $T$ to $T[i, j]$.

Inductive hypothesis: The algorithm returns the maximum sum for a triangle with $r = k$ rows.

First, the base case ($r = 1$) is true. If we have a triangle with only one row, then it can only have one entry, and it is clear that the maximum sum in the tree is exactly the value of that entry.

General case: We want to show that the algorithm returns the correct answer when $r = k + 1$. Starting from the top of the triangle, the path followed to obtain the maximum sum must pass through one of the numbers in the second row of $T$. Both of these taken individually are the top row of triangles with $k$ rows, so we know from the induction hypothesis that $S[2, j]$ stores the maximum sum from the base of $T$ to the numbers in the second row. Since the path to the top row is obtained by adding this sum to the value stored in the first row of $T$, it is clear that the maximum sum for $T$ will be obtained by adding $T[1, 1]$ to the maximum of the sums stored in the second row of $T$, which is exactly what is calculated by the recurrence relation. Therefore, our algorithm is correct for $r = k + 1$ and therefore for all $r$.

d. During the execution of this algorithm, we consider each number a constant number of times. Therefore, the algorithm runs in $O(n)$.

Problem 2

a. The sequence of cuts to minimize the cost of cutting a piece of wood of length 10 at positions \{a_1, a_2, a_3, a_4\} = \{1, 4, 5, 8\} is to cut it at $a_3$ (cost = 10), $a_1$ or $a_2$ (cost = 5) and $a_4$ (cost = 5) and then $a_2$ or $a_1$ (whichever was not already cut) (cost = 4), for a total cost of 24.

b. Input: a sequence of cuts $A = \{a_1, a_2, \ldots, a_n\}$, where $a_i$ is the distance from the left edge of the piece of wood, and $L$, the length of the piece of wood.

For the recurrence relation, redefine $A = \{a_1, a_2, \ldots, a_n, a_{n+1}\} = \{a_1, a_2, \ldots, a_n, L\}$ such that in the recurrence relation, $a_{n+1} = L$. The recurrence relation for this problem can be described as

$$C[i, j] = \begin{cases} 0 & \text{if } i = j \text{ or } i = j - 1 \\ a_j - a_i + \min_{i < k < j} C[i, k] + C[k, j] & \text{if } i \neq j \text{ and } i \neq j - 1 \end{cases}$$
where \( C[i, j] \) is the minimum cost of cutting the wood between cuts \( a_i \) and \( a_j \) and \( i \leftarrow 0 \) to \( n \) and \( j \leftarrow 1 \) to \( n + 1 \).

\( C[i, j] \) is a table in which only the upper half is filled (we only need \( i \leq j \) since we are looking at the cost of cutting between positions \( a_i \) and \( a_j \), so all other entries would be redundant.) Each entry \( C[i, j] \) holds the minimum cost of cutting the wood between positions \( a_i \) and \( a_j \), and the table is filled diagonally, from the center diagonal to the upper-right corner.

Algorithm:

\[
\text{Min-Cost}(\{a_1, a_2, \ldots, a_n\}, L)
\]

1. \( A \leftarrow \{a_1, a_2, \ldots, a_n, L\} \)
2. \( \text{for } i \leftarrow 1 \) to \( n \) \( \text{do} \)
3. \hphantom{2.} \( \text{for } j \leftarrow 0 \) to \( n + 1 \) \( \text{do} \)
4. \hphantom{3.} \hphantom{2.} \hphantom{3.} \hphantom{4.} \hphantom{5.} \hphantom{6.} \hphantom{7.} \hphantom{8.} \hphantom{9.} \text{if } i = 1 \)
5. \hphantom{3.} \hphantom{2.} \hphantom{3.} \hphantom{4.} \hphantom{5.} \hphantom{6.} \hphantom{7.} \hphantom{8.} \hphantom{9.} \hphantom{10.} \hphantom{11.} \hphantom{12.} \text{then } C[j, j + 1] \leftarrow 0 \)
6. \hphantom{3.} \hphantom{2.} \hphantom{3.} \hphantom{4.} \hphantom{5.} \hphantom{6.} \hphantom{7.} \hphantom{8.} \hphantom{9.} \hphantom{10.} \hphantom{11.} \hphantom{12.} \text{else } C[j, j + 1] \leftarrow a_{j+1} - a_j + \min_{j < k < j + 1} C[j, k] + C[k, j + 1] \)
7. \( \text{return } C[0, n + 1] \)

We can prove that the algorithm is correct by showing that the recurrence relation works and each \( C[i, j] \) is the minimum cost of cutting between positions \( a_i \) and \( a_j \), since the algorithm follows the recurrence relation defined above.

First, the base cases are true. In the base case where \( i = j \), it is obvious that there is no cut between a position and itself. In the base case where \( i = j - 1 \), there are no cuts between positions \( a_i \) and \( a_j \), hence the cost is again 0.

Now take all other cases in which there is at least one cut between positions \( a_i \) and \( a_j \). Suppose the minimum cost \( C[i, j] \) for making all cuts between \( a_i \) and \( a_j \) is obtained by making the next cut at some position \( a_{k'} \), \( i < k' < j \). Then \( C[i, j] = a_j - a_i + C[i, k'] + C[k', j] \). Because \( a_j - a_i \) is constant for any given \( i, j \) (the length of a piece of wood does not change regardless of how it will be cut), \( C[i, k'] + C[k', j] \) is the
minimum of all \( C[i, k] + C[k, j] \) \( \forall i < k < j \), where \( a_k \) is the next cut, and the recurrence relation (and hence the algorithm) holds.

d. The recurrence relation fills in the upper triangle of a table with dimensions \( n \) by \( n \). There are thus \( n^2/2 \) or \( O(n^2) \) entries in the table. For every entry \( C[i, j] \), we perform on the order of \( O(n) \) operations by comparing the sum of the entries \( C[i, k] + C[k, j] \) for all \( i < k < j \). This is a linear number of operations on the order of \( n \) since in the worst case, we have \( 0 < k < n + 1 \) and have to sum the entries \( C[0, 1] + C[1, n + 1] \) through \( C[0, n] + C[n, n + 1] \). The total runtime is thus \( O(n^2) \times O(n) = O(n^3) \).

Problem 3

a. First, we move to the middle of \( (x_1, x_2 \ldots x_n) \), arriving at element \( x_{n/2} \). We know that the optimal solution to the \textit{BigMiddleWin} problem includes this element, and so must be composed of the two maximum-sum substrings that start at \( x_{n/2} \) and moves towards each end of the input list. With this knowledge in hand, we can simply walk the list in either direction, finding \( i \) such that the cumulative sum \( x_{n/2} + x_{n/2+1} \ldots x_i \) is maximized, and \( j \) such that the sum \( x_{n/2+1} + x_{n/2+2} \ldots x_j \) is maximized. When we are done, return the sum of the two cumulative sums - this is the sum \( x_i + x_{i+1} \ldots + x_j \).

b. The algorithm above can be adapted to solve the general \textit{BigWin} problem. We can recursively apply the \textit{BigMiddleWin} algorithm as follows:

- First, apply the \textit{BigMiddleWin} algorithm to the input list. This gives us the maximum-sum substring that includes the middle element of this list.
- Recursively apply this \textit{BigWin} algorithm to the sublists \( (x_1, x_2 \ldots x_{n/2}) \) and \( (x_{n/2+1}, x_{n/2+2} \ldots x_n) \). This gives us the maximum-sum substrings that are entirely contained in each half of the input list.
- Finally, return the maximum value among these three sums.

c.

d.
e. For a more efficient solution to the *BigWin* problem, we need to utilize the intuition that the maximal-sum substring of the input list cannot begin or end with a negative number - after all, we can chop off any negative values on the end of any such substring to produce a higher-sum substring. More generally, an negative-sum substring at the beginning or end of a substring can be removed to increase the total sum of the elements in the substring. With this in mind, consider the following algorithm:

- Create a variable `maxSum` to record the maximum sum we encounter.
- Initialize `i`, `j`, and `sum` - these represent the start, end, and total sum of the substring we’re considering at any point.
- Start at the beginning of the input list, set `i`, `j`, and `sum` to 0.
- In each iteration, increment `j` by 1 and add `x_j` to `sum`. This includes `x_j` in the substring we’re currently considering.
- After adding `x_j`, consider `sum` - if it is greater than `maxSum`, update `maxSum` to the value of `sum`. If `sum` is negative, we want to drop the negative-sum string from `x_i` to `x_j`, so set the value of `i` to the value of `j`. Set `sum` to 0.
- Once we reach the end of the list, at element `x_n`, return `maxSum`.

f.

g.