Sample Solutions to Assignment 4

1. You are given a boolean expression consisting of a string of the symbols *true*, *false*, separated by operators AND, OR, NAND and NOR but without any parentheses. Count the number of ways one can put parentheses in the expression such that it will evaluate to *true*. (20 pts)

   Solution: Let the number of symbols in the expression be n.

   Subproblem: Let $T(i,j)$ be the number of ways to parenthesise the subexpression between symbols i and j (inclusive) such that it evaluates to *true*, and let $F(i,j)$ be the number of ways to parenthesise the subexpression such that it evaluates to *false* ($1 \leq i \leq j \leq n$). The base cases are:

$$T(i,i) = \begin{cases} 1 & \text{if symbol } i \text{ is } \textit{true} \\ 0 & \text{if symbol } i \text{ is } \textit{false} \end{cases} \qquad F(i,i) = \begin{cases} 0 & \text{if symbol } i \text{ is } \textit{true} \\ 1 & \text{if symbol } i \text{ is } \textit{false} \end{cases}$$

   The recursion is:

$$T(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k) \times T(k+1,j) \\ \qquad \text{if operator } k \text{ is AND} \\ T(i,k) \times T(k+1,j) + T(i,k) \times F(k+1,j) + F(i,k) \times T(k+1,j) \\ \qquad \text{if operator } k \text{ is OR} \\ T(i,k) \times F(k+1,j) + F(i,k) \times T(k+1,j) + F(i,k) \times F(k+1,j) \\ \qquad \text{if operator } k \text{ is NAND} \\ F(i,k) \times F(k+1,j) \\ \qquad \text{if operator } k \text{ is NOR} \end{cases}$$

$$F(i,j) = \sum_{k=i}^{j-1} \begin{cases} T(i,k) \times F(k+1,j) + F(i,k) \times T(k+1,j) + F(i,k) \times F(k+1,j) \\ \qquad \text{if operator } k \text{ is AND} \\ F(i,k) \times F(k+1,j) \\ \qquad \text{if operator } k \text{ is OR} \\ T(i,k) \times T(k+1,j) \\ \qquad \text{if operator } k \text{ is NAND} \\ T(i,k) \times T(k+1,j) + T(i,k) \times F(k+1,j) + F(i,k) \times T(k+1,j) \\ \qquad \text{if operator } k \text{ is NOR} \end{cases}$$

   Algorithm: We fill an $n \times n$ triangular matrix in a bottom-up manner by first computing $T(i,j)$ and $F(i,j)$ for all i and j such that $j - i = 0$, then $T(i,j)$

and $F(i, j)$ for all i and j such that $j - i = 1$, and so on until $j - i = n - 1$. The final answer to the problem is $T(1, n)$.

Time complexity: There are $O(n^2)$ subproblems, and for each subproblem, we perform an $O(n)$ summation over previously computed results. Therefore, the time complexity of the algorithm is $O(n^3)$.

2. You are given a 2D map consisting of an $R \times C$ grid of squares; in each square there is a number representing the elevation of the terrain at that square. Find a path going from square $(1, R)$ which is the top left corner of the map to square $(C, 1)$ in the lower right corner which from every square goes only to the square immediately below or to the square immediately to the right so that the number of moves from lower elevation to higher elevation along such a path is as small as possible. (20 pts)

Solution: Let $E(c, r)$ be the elevation at square (c, r).

Subproblem: Let $T(c, r)$ be the minimum number of moves from lower elevation to higher elevation needed to go from square $(1, R)$ to square (c, r) $(1 \le c \le C,\ 1 \le r \le R)$. The base case is $T(1, R) = 0$. The recursion is:

$$T(c,r) = \begin{cases} T(c-1,r) & \text{if } r = R \text{ and } E(c-1,r) \ge E(c,r) \\ T(c-1,r)+1 & \text{if } r = R \text{ and } E(c-1,r) < E(c,r) \\ T(c,r+1) & \text{if } c = 1 \text{ and } E(c,r+1) \ge E(c,r) \\ T(c,r+1)+1 & \text{if } c = 1 \text{ and } E(c,r+1) < E(c,r) \\ \min(T(c-1,r), T(c,r+1)) & \\ & \text{if } E(c-1,r) \ge E(c,r) \text{ and } E(c,r+1) \ge E(c,r) \\ \min(T(c-1,r)+1, T(c,r+1)) & \\ & \text{if } E(c-1,r) < E(c,r) \text{ and } E(c,r+1) \ge E(c,r) \\ \min(T(c-1,r), T(c,r+1)+1) & \\ & \text{if } E(c-1,r) \ge E(c,r) \text{ and } E(c,r+1) < E(c,r) \\ \min(T(c-1,r)+1, T(c,r+1)+1) & \\ & \text{if } E(c-1,r) < E(c,r) \text{ and } E(c,r+1) < E(c,r) \end{cases}$$

Algorithm: We fill an $R \times C$ table row by row (or column by column). After we have filled the table, we can traverse the table starting from cell $(C, 1)$, moving left or up in the direction of smaller value to get the path in reverse.

Time complexity: The table has $R \times C$ entries (i.e., there are $R \times C$ subproblems), and each entry can be computed in $O(1)$ time. After the entire table has been computed, the path can be determined in $O(R + C)$ time. Hence, the overall time complexity is $O(RC)$.

3. You are on vacation for N days at a resort that has three possible activities. For each day, for each activity, you've determined how much enjoyment you will get out of that activity if you do it on that particular day (the same activity might give you a different amount at different days). However, you are not allowed to do the same activity two days in a row. What is the maximum total enjoyment possible? (30 pts)

   Solution: Let $E(i, j)$ represent the amount of enjoyment you will get from doing activity j on day i.

   Subproblem: Let $T(i, j)$ be the maximum total enjoyment that can be obtained up to day i, assuming that you did activity j on day i ($1 \leq i \leq N, 1 \leq j \leq 3$). The base case is $T(1, j) = E(1, j)$ for all j. The recursion is:

$$\begin{aligned} T(i,1) &= \max(T(i-1,2),\ T(i-1,3)) + E(i,1) \\ T(i,2) &= \max(T(i-1,1),\ T(i-1,3)) + E(i,2) \\ T(i,3) &= \max(T(i-1,1),\ T(i-1,2)) + E(i,3) \end{aligned}$$

   Algorithm: We fill an $N \times 3$ table row by row using the recursion above. The maximum total enjoyment possible is $\max(T(N,1), T(N,2), T(N,3))$.

   Time complexity: The table has $3N$ entries, and each entry can be computed in $O(1)$ time. Hence, the time complexity of the algorithm is $O(N)$.

4. Given a weighted **directed** graph $G(V, E)$, find a path in G (possibly self-intersecting) of length exactly K that has the maximum total weight. The path can visit a vertex multiple times and can traverse an edge also multiple times. It can also start and end at arbitrary vertices or even start and end at the same vertex. (30 pts)

   Solution: Let $\text{weight}(j, i)$ represent the weight of an edge from j to i.

   Subproblem: Let $\text{opt}(k, i)$ be the maximum total weight of a path of length k that ends at node i ($0 \leq k \leq K$, $1 \leq i \leq |V|$). The base case is $\text{opt}(0, i) = 0$ for all i. The recursion is:

$$\text{opt}(k, i) = \max\{\text{opt}(k-1, j) + \text{weight}(j, i) : (j, i) \in E\}$$

   Algorithm: We fill a $(K+1) \times |V|$ table. In each cell (k, i) of our table, we store two values: (1) $\text{opt}(k, i)$, and (2) the value of j for which $\text{opt}(k-1, j) + \text{weight}(j, i)$ is maximum (i.e., the predecessor of node i on the maximum weight path of length k that ends at i). Once we have computed the entire table, $\arg\max_i \text{opt}(K, i)$ is the final node in the maximum weight path, and we can traverse the table by following the appropriate predecessor values to obtain the path.

Time complexity: For each value of k, we consider each node once, and for each node, we consider each incoming edge. Hence, for each value of k, we perform an $O(|V| + |E|)$ operation, and therefore the time complexity of the algorithm is $O(K(|V| + |E|))$.