CSCI 406: Algorithms
Solution to Assignment 5

8-12 – [10] First, observe that the problem is asking for an optimal sequence of breaks. It is not asking us to actually *perform* the breaks, nor is it concerned with the contents of the string. With this in mind, let k be the number of breaks specified and let's assume that these are distinct; i.e., we don't have two breaks in the same location. Note the string has n characters and so the number of breaks $k \leq n - 1$. Let break i occur after character b_i for $1 \leq i \leq k$. In the example in the book, $k = 3$ and $b_1 = 3$, $b_2 = 8$ and $b_3 = 10$. In addition, we add dummy breaks b_0 at the start of the string and b_{k+1} at the end of the string (to simplify the description of the algorithm).

Notation: Next, we define notation. This will be key to formulating the recurrence relation and the table. Let minCost(i, j) represent the optimal cost of breaking a string segment consisting of characters $b_i + 1$ to b_j.

Recurrence Relation We have the following recurrence relation:

minCost(i,j) $= (b_j - b_i) + \min_{i<a<j}(minCost(i, a) + minCost(a, j))$, when $j - i \geq 2$.

The base case occurs when $j - i = 1$ and results in a cost of 0. Note this corresponds to a string segment that does not contain any breaks.

We are seeking the solution to minCost(0,$k + 1$).

Recursive algorithm: It should be straighforward to write a recursive algorithm that implements the recurrence relation above. This will require a for loop indexed on a. (We omit this.) A naive implementation of the recurrence relation results in repeated computations of identical subproblems. We seek to improve on this by using a table as below.

– [10 pts] *Table*: Next we define a two-dimensional $(k + 2) \times (k + 2)$ table with both rows and columns indexed from 0 to $k + 1$, but it's important to recognize that most of the table will remain unused. Only (i, j) locations with $0 \leq i < j \leq k + 1$ (i.e., the upper triangle) will get used.

The dynamic programming algorithm follows:

```
for i = 0 to k TABLE[i][i+1] = 0; // base case, diagonal above principal diagonal

for g = 2 to k+1 do // process diagonal g away from the principal diagonal
    for i = 0 to k+1-g do // process each entry in the diagonal
        j = i + g;
        min = infinity;
        minA = null
        for a = i+1  to j-1 do // apply recurrence relation
            costA = b_j - b_i + TABLE[i][a] + TABLE[a][j];
            if (costA < min)
               min = costA;
               minA = a;
        TABLE[i][j] = min;
        TRACEBACK_TABLE[i][j] = minA;
```

Note that the last entry computed occurs when $g = k + 1$ and $i = k + 1 - g$ giving $i = 0$ and $j = k + 1$ or TABLE[0][k+1] which is the value we are after.

– [5 pts] Traceback step.

```
TRACEBACK(i,j)
    if (i+1 == j) return; // base case, string cannot be broken further
    // else, we are in main case
    Output ''Break at TRACEBACK_TABLE[i][j]";
    TRACEBACK(i, TRACEBACK_TABLE[i][j]); // recursive call on left substring
    TRACEBACK(TRACEBACK_TABLE[i][j], j); //recursive call on right substring
```

As before we would start by calling TRACEBACK$(0, k+1)$.

– [5 pts] The complexity of the DP algorithm is $O(n^3)$ (three nested for loops, none of which require more than n iterations).

8-14 – [10] I am expressing the recurrence as a recursive program. It could also be written as a mathematical expression.

```
ProbabilityChampRetainsTitle(g, i)

if (g == 0 && i > 0) return 0; champ needs to win i games, but no games left
if (g >= 0 && i <= 0) return 1; champ does not need to win any more games

if (g % 2 == 0) // g is even, champ plays with WHITE, use B probabilities
    return  W_w * ProbabilityChampRetainsTitle(g-1,i-1)  // Champ wins
          + W_d * ProbabilityChampRetainsTitle(g-1,i-0.5) // Draw
          + W_l * ProbabilityChampRetainsTitle(g-1,i); // Champ Loses

else // g is odd, champ plays with BLACK, use B probabilities
    return  B_w * ProbabilityChampRetainsTitle(g-1,i-1)  // Champ wins
          + B_d * ProbabilityChampRetainsTitle(g-1,i-0.5) // Draw
          + B_l * ProbabilityChampRetainsTitle(g-1,i); // Champ Loses
```

– [10] We need a two-dimensional table with entries for all combinations of g and i. There are 24 games in the match, so g the number of games left will be $0 \leq g \leq 24$. The number of points the Champion needs to win the title is i. Since the Champion wins if the match is tied, he/she needs to get 12 points. So $0 \leq i \leq 12$. However, i is in increments of 0.5, so possible values are [0, 0.5, 1.0, ..., 11.5, 12.0]. We will need a 25×25 table. These entries are populated using the following:

Beware: The following is pseudocode that assumes tables can be indexed in increments of 0.5. It would need to be modified to work in an actual programming language!

```
for g = 0 to 24 Prob[g][0] = 1.0; // g >= 0, i = 0, Champ retains
for g = 0 to 24 Prob[g][-0.5] = 1.0; // g >= 0, i < 0, Champ retains

for i = 0.5 to 12 by 0.5 Prob[i][0] = 0.0 //

for g = 1 to 24
    for i = 0.5 to 12.0 in increments of 0.5
```

```
if (g % 2 == 0) // g even
   Prob[g][i] =  W_w * Prob[g-1,i-1]
               + W_d * Prob[g-1,i-0.5]
               + W_l * Prob[g-1,i];
else // g odd
   Prob[g][i] =  B_w * Prob[g-1,i-1]
               + B_d * Prob[g-1,i-0.5]
               + B_l * Prob[g-1,i];
```

– [5] If n games are played and the champion is required to get $n/2$ points, the table has size $\Theta(n^2)$. each entry takes $\Theta(1)$ time to compute, so the time complexity is $\Theta(n^2)$.