CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)

Homework 6 Problem 1

Solution: (a) Let *MinCost*(i) be the value of the minimum cost L-split of the substring $A[1...i]$ of string $A[1...n]$. If we know *MinCost*(k) for every $k \in [1, i-1]$, then we can calculate *MinCost*(i) by first finding a non-empty suffix of length j that is in L. Then *MinCost*$(i-j) +$ *cost*(j) is the minimum cost if we accept $A[i-j...i]$ in the L-split. If no such suffix exists, then we let *MinCost*$(i) = \infty$ because there is no valid L-split and any valid L-split will cost less. We then take the minimum cost of all possible splits to be *MinCost*(i). The base case is when $i = 0$ and $A[1...0] = \epsilon \in L^*$, *MinCost*$(i) =$ *cost*$(|\epsilon|) = 0$. We can write the following recurrences.

$$\mathit{MinCost}(i) = \begin{cases} 0 & i = 0 \\ \min(\{\mathit{MinCost}(i-j) + \mathit{cost}(j) \mid j \in [1, i], A[(i-j+1)...i] \in L\} \cup \{\infty\}) & i > 0 \end{cases}$$

The result should be *MinCost*(n). If *MinCost*$(n) = \infty$, then we know there is no valid L-split.

```
MINLSPLITCOST(A[1...n]):
  integer MinCost[n + 1]
  MinCost[0] ← 0
  for i ← 1 to n
      currMin ← ∞
      for j ← 1 to i
          if IsStringInL(A[(i − j + 1)...i])
              if MinCost[i − j] + cost(j) < currMin
                  currMin ← MinCost[i − j] + cost(j)
      MinCost(i) ← currMin
  if MinCost[n] = ∞
      return "w ∉ L*!"
  return MinCost[n]
```

Since there are $\sum_{k=1}^{n} k = O(n^2)$ iterations, and in each iteration IsStringInL( ) takes $O(n)$ time, the time complexity of this algorithm is $O(n^3)$.

(b) Consider the following induction steps of regular languages.

- If $r = (s)^*$, then we check if each non-empty suffix w_i of w matches s, by recursively calling the algorithm with (w_i, s). If the suffix matches s, then we delete that suffix from w to get w_{trim} and recursively call the algorithm with (w_{trim}, r). If any recursive call returns true, then the algorithm returns true.
- If $r = r_1 + r_2$, then we recursively call the algorithm with (w, r_1) and (w, r_2). If w can be matched by any of r_1, r_2, then the algorithm should return true.
- If $r = r_1 r_2$, then we divide w into two sections, w_1, w_2, and recursively call with (w_1, r_1) and (w_2, r_2). If both recursive calls return true, then the algorithm should return true. There are $|w| + 1$ such divisions thus $2|w| + 2$ recursive calls taking ϵ into account.

Then consider the base cases:

- If $w = \epsilon$ and $r = (s)^*$, then the algorithm returns true.
- If $w = a$ where $a \in \Sigma$ and $r = a$, then the algorithm returns true.

```
IsStringInRegExp(w, r):
    if w = ϵ and r = (s)*
        return True
    if |w| = 1 and w = r
        return True
    inRegExp ← False
    if r = (s)*
        for each non-empty suffix w_i of w
            if IsStringInRegExp(w_i, s)
                w_trim ← w without w_i at the end
                inRegExp ← inRegExp ∨ IsStringInRegExp(w_trim, r)
    else if r = r_1 + r_2
        inRegExp ← IsStringInRegExp(w, r_1) ∨ IsStringInRegExp(w, r_2)
    else if r = r_1 r_2
        for i ← 0 to |w|
            w_1 ← w[1 : i]              // w[1 : 0] = ϵ
            w_2 ← w[i + 1 : |w|]        // w[|w| + 1 : |w|] = ϵ
            if IsStringInRegExp(w_1, r_1) and IsStringInRegExp(w_2, r_2)
                inRegExp ← True
    return inRegExp
```

■

CS/ECE 374 Spring 2023 William Cheng (shihuac2@illinois.edu)

Homework 6 Problem 2

Solution: Suppose Mr.Fox is now at booth i. To account for the possibility that any future value in $A[i+1...n]$ will change the optimal sequence of "Ring!" and "Ding!", we keep track of the number of consecutive "Ring!"s and "Ding!"s.

Let *maxChicken*(i,j,k) be the maximum number of chickens that Mr.Fox can get when he is at booth i, with j consecutive "Ring!"s and k consecutive "Ding!"s before Mr.Fox reaches booth i. Note that exactly one of j and k must be 0 because there cannot be consecutive "Ring!"s and "Ding!"s at the same time. The final result should be the maximum in the set $\{\textit{maxChicken}(n,j,k)\}$ since it contains all possible values when Mr.Fox reaches booth n. Consider the following cases:

- Mr.Fox is at booth 1. There can be at most one consecutive "Ring!" or "Ding!", and the maximum numbers of chickens are $A[i]$ and $-A[i]$ respectively.
- Mr.Fox is at booth i where $i > 1$, and he has shouted j consecutive "Ring!"s before he reaches booth i.
  - This time he shouts "Ding!". Since Mr.Fox could have shouted 1, 2, or 3 consecutive "Ring!"s before, the maximum number of chickens in this situation is the maximum of $\{\textit{maxChicken}(i-1,m,0) - A[i]\}$ where $1 \leq m \leq 3$.
  - This time he shouts "Ring!". By the rules, $j \leq 2$ because otherwise the farmer would shoot Mr.Fox. Then the maximum number of chickens is just the maximum number of chickens at booth $i-1$ where Mr.Fox had shouted $j-1$ consecutive "Ring!"s plus $A[i]$.
- Mr.Fox is at booth i where $i > 1$, and he has shouted k consecutive "Ding!"s before he reaches booth i.
  - This time he shouts "Ring!". Since Mr.Fox could have shouted 1, 2, or 3 consecutive "Ding!"s before, the maximum number of chickens in this situation is the maximum of $\{\textit{maxChicken}(i-1,0,m) + A[i]\}$ where $1 \leq m \leq 3$.
  - This time he shouts "Ding!". By the rules, $k \leq 2$ because otherwise the farmer would shoot Mr.Fox. Then the maximum number of chickens is just the maximum number of chickens at booth $i-1$ where Mr.Fox had shouted $k-1$ consecutive "Ding!"s minus $A[i]$.

The recurrences are as follows:

$$\textit{maxChicken}(i,j,k) = \begin{cases} A[i] & i=1, j=1, k=0 \\ -A[i] & i=1, j=0, k=1 \\ \max(\{\textit{maxChicken}(i-1,0,m) + A[i] \mid 1 \leq m \leq 3\}) & i>1, j=1, k=0 \\ \max(\{\textit{maxChicken}(i-1,m,0) - A[i] \mid 1 \leq m \leq 3\}) & i>1, j=0, k=1 \\ \textit{maxChicken}(i-1,j-1,0) + A[i] & i>1, j>1, k=0 \\ \textit{maxChicken}(i-1,0,k-1) - A[i] & i>1, j=0, k>1 \\ -\infty & \text{otherwise} \end{cases}$$

Note that only j and k confined to $[0, 3]$ should be considered in the above recurrence relations according to the rules. Invalid values are initialized to $-\infty$ because any valid value would be larger and would therefore be taken. Following is the pseudocode for the algorithm. *Note: To be consistent with the recurrence relation above, the array **maxChicken**'s first index is* 1*-based, while the second and third indices are* 0*-based.*

```
LargestNumOfChickens(A[1...n]):
  integer maxChicken[n][4][4]
  initialize maxChicken with −∞
  maxChicken[1][1][0] ← A[1]
  maxChicken[1][0][1] ← −A[1]
  for i ← 2 to n
      maxChicken[i][0][1] ← max(maxChicken[i − 1][m][0] for m ← 1 to 3) − A[i]
      maxChicken[i][1][0] ← max(maxChicken[i − 1][0][m] for m ← 1 to 3) + A[i]
      for j ← 2 to 3
          maxChicken[i][j][0] ← maxChicken[i − 1][j − 1][0] + A[i]
      for k ← 2 to 3
          maxChicken[i][0][k] ← maxChicken[i − 1][0][k − 1] − A[i]
  largestNumOfChickens ← largest element in the 2-d array maxChicken[n]
  if largestNumOfChickens < 0
      just shoot Mr.Fox before the obstable course since he would get shot anyway
  return largestNumOfChickens
```

Since only constant-time operations are involved in each iteration of i and there are $n - 1$ such iterations, the time complexity of the algorithm is $O(n)$. ■