CSCI 570 Spring 2020 Xinrui Ying (USCID 1092350440)
Homework 3 Problem

Solution:

1. We loop through all edges, for each directed edge (x, y), we assign weight of node y to the edge (x, y). Afterward, we simply run Dijkstra algorithm twice, once from A to B and once from B to A. We add weight of node A to the path selected from A to B and we add weight of node B to the path selected from B to A. We then selected the least cost path among the two. The total time complexity will be $|E| + 2 \cdot (|E| \log |E|) = O(|E| \log |E|) = O(|E| \log |V|)$.

2.
   - $T(n) = 3T(n/2) + n^2$
     By Master Theorem, $a = 3$ and $b = 2 \implies c = \log_2 3 \leq 2$. Satisfy the case 3 condition, $T(n) = \Theta(n^2)$
   - $T(n) = 4T(n/2) + n^2$
     By Master Theorem, $a = 4$ and $b = 2 \implies c = 2 = 2$. Satisfy the case 2 condition, $T(n) = \Theta(n^2 \log n)$
   - $T(n) = T(n/2) + 2^n - n$
     By Master Theorem, $n^{\log_2 1} = n = O(2^n)$. Satisfy the case 3 condition, $T(n) = \Theta(2^n)$
   - $T(n) = 2^n T(n/2) + n^n$
     Master theorem not apply.
   - $T(n) = 16T(n/4) + n + 10$
     By Master Theorem, $a = 16$ and $b = 4 \implies c = 2 \geq 1$. Satisfy the case 1 condition, $T(n) = \Theta(n^2)$
   - $T(n) = 2T(n/2) + n \log n$
     By Master Theorem, $a = 2$ and $b = 2 \implies c = 1 \geq 1$. Satisfy the case 2 condition, $T(n) = \Theta(n \log^2 n)$
   - $T(n) = 2T(n/4) + n^{0.51}$
     By Master Theorem, $a = 2$ and $b = 4 \implies c = \frac{1}{2} \leq 0.51$. Satisfy the case 3 condition, $T(n) = \Theta(n^{0.51})$
   - $T(n) = 0.5T(n/2) + 1/n$
     Master theorem not apply.
   - $T(n) = 16T(n/4) + n!$
     By Master Theorem, $n^{\log_4 16} = n^2 = O(n!)$. Satisfy the case 3 condition, $T(n) = \Theta(n!)$
   - $T(n) = 10T(n/3) + n^2$
     By Master Theorem, $a = 10$ and $b = 3 \implies c = \log_3 10 \geq 2$. Satisfy the case 1 condition, $T(n) = \Theta(n^{\log_3 10})$

3. Initialize $l = 1$ and $r = n$ for $A[1...n]$. The algorithm is as following:

   - Check if $l > r$, if it is then return $False$; else, we define $mid = (l + r)/2$.
   - If $A[mid] == mid$, we return $True$; else if $A[mid] > mid$, we set $r = mid - 1$; else we set $l = mid + 1$.
   - Iterate the process with new l and r.

   $T(n) = T(n/2) + O(1)$. The total time complexity will be $\Theta(\log n)$.

4. Set the current element as the first element in the linkedList. The algorithm is as following:

   - Given linkedList of size k, if k is 0, we return $False$; if k is 1, we check the value of head of the linkedList, if it matches the target, we return $True$; if it doesn't match, we return $False$.
   - If $k \geq 2$, we keep going to the next element until we reached at the $k/2$ index element. We will have the $k/2 + 1$ index element be the new second linkedList's head, we also will have the $k/2$ index element points to $NULL$. Thus, we will have 2 equal sized linkedLists.
   - We check if the value of second LinkedList's head is greater than our target, if it's greater than our targer, we iterate the process using the first linkedList with $k/2$ elements; if it's less than our target, we iterate the above process using the second linkedList with $k - k/2$ elements. If it's equal to the target, return $True$.

   $T(n) = T(n/2) + O(n)$. $c = \log_b a = \log_2 1 = 0$, it's case 3, the total time complexity will be $\Theta(n)$.

5. Let n be the length of $nums$, create a $2d$ array A of size $(n+2) \times (n+2)$. Initialize all entries to 0. For $A[i][j]$ represent the maximum coins collected by adding balloons through index i to j not include index i and j.

   We reverse the idea by reverse the process, instead of burst balloons, we choose to add balloons to empty list. Each process, we will have the left right index corresponding to the original $nums$, indicates the choice of balloons we can add to the list. We iterate all choice and once we add the balloon, the instant coins we get is $nums$[leftindex] $\times$ $nums$[rightindex] $\times nums$[selectingballoon], the list is split into two parts: left index to the choosing balloon index and choosing balloon index to the right index. We then continue on the two subproblems until there is no balloons to add between the left and right index.

   The pseudo-code is as following as we input $maximumCoins(nums, 0, sizeOf(nums)+2)$

Algorithm 1 maximumCoins($nums$,$left$, $right$)

```
Let $n$ be size of $nums$, and $A$ be array of size $(n+2) \times (n+2)$ with all 0.
Add 1 at the front and the end of $nums$.
if $left + 1 = right$ then
    return 0
end if
for $i = left+1, left+2 \ldots, right-1$ do
    if $A[left, i]$ is 0 then
        $A[left, i] = maximumCoins(nums, left, i)$
    end if
    if $A[i, right]$ is 0 then
        $A[i, right] = maximumCoins(nums, i, right)$
    end if
    $A[0, n+1] = max\{A[0, n+1], nums[i-1] \times nums[i] \times nums[i+1] + A[left, i] + A[i, right]\}$
end for
Return $A[0, n+1]$
```

The time complexity will be $O(n^3)$ as space complexity is $O(n^2)$ and each entry is need linear time to compute.

6. Create a $1d$ array A of size $(n+1)$ where n is the size of the str, and initialize all entry from 0 to $n-1$ to 0, and the last entry to 1. Each $A[i]$ for $0 \le i \le n-1$, $A[i]$ represent if str from i to $n-1$ is a sequence of dictionary words.

   Based on our data structure, we start from second last row and going up. For each row i, we loop through i to $n-1$, if str from i to j is in dictionary and $A[j+1]$ is 1, we set $A[i]=1$ and continue with row $i-1$. Once all rows are finished, we return $True$ if $A[0] == 1$; else we return $False$.

Algorithm 2 sequenceOfDict(str, $Dictionary$)

```
Let $n$ be size of $str$, and $A$ be array of size $n+1$ with $A[0],\dots A[n-1]$ to 0, $A[n]=1$.
for $i = n-1,\dots,0$ do
    for $j = i, i+1,\dots,n-1$ do
        $condition1$ = Check if $str$ from $i$ to $j$ is in $Dictionary$
        if $condition1$ is $True$ then
            $condition2$ = Check if $A[j+1]$ is 1
            if $condition2$ is $True$ then
                Set $A[i]$ to 1 and exit the inner loop
            end if
        end if
    end for
end for
Return $True$ if $A[0]$ is 1, else $False$.
```

Time complexity will be $O(n^2)$ as we loop through n rows and we have linear operation for each loop.

7. We create a $1d$ array A of size n, initialize A to 0 and start by putting a_0 to $A[0]$. We loop from 1 to $n-1$, $A[i] = max\{A[i-1]+a_i, a_i\}$. Each time step, if we get a new global maximum, we will update startIndex and endIndex accordingly. We keep track of global maximum entire time.

   The pseudo-code is as following as we input $maxSequence(a)$

Algorithm 3 maxSequence(a)

```
Initialize $A$ as a $1d$ array of size $n$ with $A[0] = a_i$, and $A[i] = 0$ for $i > 0$
$startIndex = 0$, $endIndex = 0$, $currentStartIndex = 0$, and $maximum = A[0]$

for $i = 1, \dots, n-1$ do
    if $A[i-1] + a_i \geq a_i$ then
        $A[i] = A[i-1] + a_i$
        if $A[i] > maximum$ then
            $maximum = A[i]$
            $startIndex = currentStartIndex$
            $endIndex = i$
        end if
    end if
    if $A[i-1] + a_i < a_i$ then
        $A[i] = a_i$
        $currentStartIndex = i$
        if $A[i] > maximum$ then
            $maximum = A[i]$
            $startIndex = currentStartIndex$
            $endIndex = i$
        end if
    end if
end for
return $(startIndex, endIndex)$
```

The time complexity will be $T(n) = T(n-1) + O(1)$, $\Theta(n)$.

■