Evan Edelstein
EN.605.621.84.FA25
HW 4
The work in this exercise is mine alone without un-cited help. No AI was used to answer these questions.

1) Pseudocode:

```
Function print_in_out_degree(G):            // G is a graph with .adj member
    vertices = G.adj.length()
    in_degree = []
    out_degree = []
    // initialize degree counters
    for i=1 to vertices:
        in_degree[i] = 0
        out_degree[i] = 0
    // iterate adjacency list to get in/out degree of each vertex
    For i=1 to vertices:                    // for vertex i in G
        od = 0                              // counter for out-degree of vertex i
        For j in G.adj[i]:                  // for edges out of i -> (i,j)
            od++                            //increment
            in_degree[j]++                  //increment in degree of neighbor j
        out_degree[i] = od                  // assign out degree of vertex i
    for i=1 to vertices:
        print("vertex: ",  i)
        print("in-degree:"    , in_degree[i])
        print("out-degree:" , out_degree[i])
```

The input to the function is a directed graph *G*. The graph is represented by an adjacency list *adj*, that is a member of *G*. Each index in the adjacency list represents a vertex in the graph and points to another list of vertices that it forms an edge with. A visual example can be in figure 1. We assume that the nodes in the graph are labeled as integers.

The goal of the pseudocode is to implement a function that computes the in- and out- degree of each vertex in G. The in-degree of a vertex is the number of edges that terminate at the vertex. Conversely, the out-degree is the number of edges that start at the vertex. The in-degree of each vertex can be computed by iterating through all the edges in the graph and tallying the number of times a vertex is an end of an edge. In code, this translates to a nested for loop that iterates through each starting vertex *i* in *G.adj* and every ending vertex *j* in *G.adj[i]*. We can use an array to hold the count of how many times each *j* is seen. The out-degree of a vertex *i* is the length of the array stored at *G.adj[i]*.

This algorithm begins by initializes two secondary arrays to store the in- and out-degree for each vertex. This loop (line 6) iterates over all the vertices in *G* and does constant work within the loop (initialized index to 0). This step contributes a factor of θ (n) to the runtime. The next loop (line 10) performs a nested for loop. The outer loop operates over all the vertices in *G* and over the course of the program execution the inner for loop will iterate over all the edges in the graph. Since only constant time operations (addition/assignment) are performed in the nested for loop, the outer loop contributes a factor of θ (n) to the runtime and the inner loop contributes a factor of θ (m) to the runtime. A final for loop is performed over the number of vertices to print the in- and out- degree of each vertex, contributed another θ (n) factor to the runtime. In total the runtime is then $3\theta(n) + \theta(m) \rightarrow \theta(n + m)$

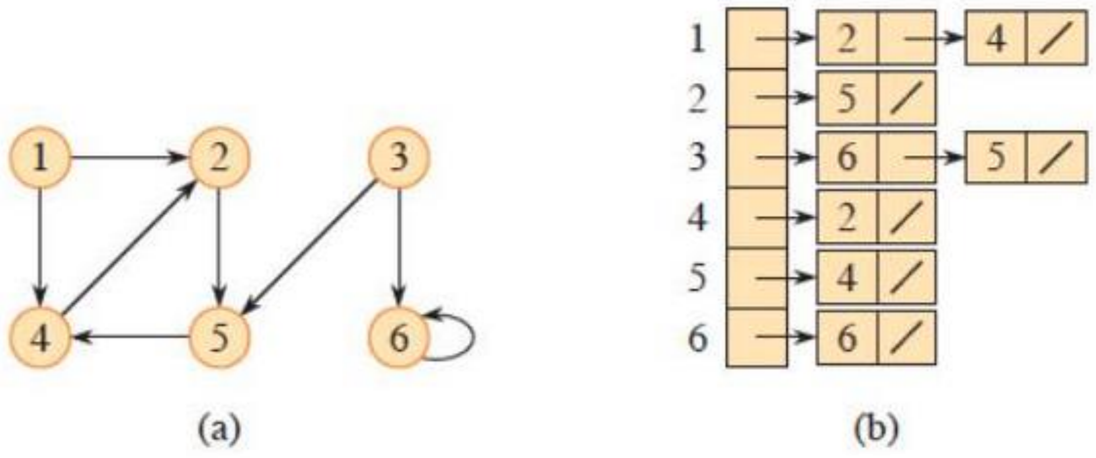

Figure 1: a) graphical representation of directed graph. b) adjacency list representation of directed graph. Image taken from [1]

2)

a. Since we are looking for a straight line that connects the wells (Figure 1), we only need to consider the difference in y-value between the wells to find an optimal pipe location. In this way we can think of all the wells as being on single vertical line (Figure 2) and reframe the problem as searching for a point on this line that has the minimum distance to all the points. We can use the median of all the wells y axis value to determine this point, because, by definition, the median is the point that is closest to all the wells. We can show that this is valid solution by an inductive argument. For a single well, the pipeline should obviously be placed through the y value of the well. For two wells we need to find a point halfway in between the two wells. For three wells we can find a midpoint between all three wells, which is the median of their y values. For any more wells, if the number of wells is even, we need to look for the halfway point between the two middle wells, and if the number if wells is odd, the median of the wells y-values will minimize the pipeline spur length.

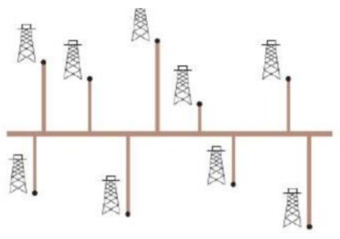

Figure 1: Problem construction from textbook

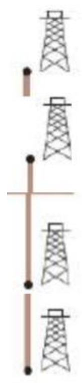

Figure 2: Mock projection of 4 wells onto singe x-axis.

b. The Select algorithm from the textbook can be used to find the median of an array of the wells y-coordinates in O(n) time. For an array with an odd number of wells (n), we would select for the (n+1)/2 order statistic and for an even number of wells, we select either the (n/2) or (n+1)/2 order statistic (or we can put it halfway between the two). This is accomplished using the median-of-median construction, similar to the pivot selection in quicksort, but without sorting the well coordinates (which wouldn't be linear).

3)

a. The worst-case running time of the operation MultiPopA is O(n), since at worst the entire stack A is popped (when k >= n) and each pop operations is O(1). The same applies to MultiPopB, which has the worst-case runtime of O(m). The worst case for Transfer is when the entire Stack A is popped and then pushed onto stack B, which results in n pushes and n pops, each of which are O(1). In this case the runtime would be O(2n) = O(n).

b. We need to define a potential function potential function that can account for the Transfer operation which performs multiple constant time sub-operations per item, one pop from StackA and one push into StackB. We also want to ensure the Transfer operation has no potential change, as the total elements is not changing. Therefore, we can define our potential function to be:

$$\Phi(n, m) = an + m$$

for StackA and StackB and define n = |StackA| and m |StackB|. The amortized cost for each function after i operations can be defined by:

$$\hat{c_i} = c_i + \Phi(n, m)_{After} - \Phi(n, m)_{Before}$$

The following table shows the amortized cost of the Transfer operation with different values of *a*. Note we use x to be equal to the minimum of n and k, and y to be the minimum of k and m.

a	Actual Cost (c_i)	$\Phi(n,m)_{After}$	$\hat{c_i}$
1	2 *min(n,k) = 2x	$(n-x)+m+x)$	$2x + \big((n-x)+(m+x)\big) - (n+m)$ $\rightarrow 2x + n - x + m + x - n - m \rightarrow 2x$
2	2 *min(n,k) = 2x	$2(n-x)+(m+x)$	$2x + \big((2n-2x)+(m+x)\big) - (2n+m)$ $\rightarrow 2x + 2n - 2x + m + x - 2n - m$ $\rightarrow x$
3	2 *min(n,k) = 2x	$3(n-x)+(m+x)$	$2x + \big((3n-3x)+(m+x)\big) - (3n+m)$ $\rightarrow 2x + 3n - 3x + m + x - 3n - m$ $\rightarrow 0$

We see that $a = 3$ satisfies the invariant that the Transfer operation has no change in potential. The following table shows how the rest of the operation have an amortized cost of O(1).

Function	Actual Cost (c_i)	$\Phi(n,m)_{After}$	$\hat{c_i}$
PushA	1	$3(n+1)+m$	$1 + (3n+3+m) - (n+m)$ $\rightarrow 1 + 3n + 3 + m - 3n - m \rightarrow 4$

PushB	1	$3n + (m + 1)$	$1 + (3n + m + 1) - (3n + m) \rightarrow 1 + 3n + m + 1 - 3n - m \rightarrow 2$
MultiPopA	min(n,k) = x	$3(n - x) + m$	$x + \left((3n - 3x) + m\right) - (3n + m) \rightarrow x + 3n - 3x + m - 3n - m \rightarrow -2x$
MulitPopB	min(m,k) = y	$3n + (m - y)$	$y + \left(3n + (m - y)\right) - (3n + m) \rightarrow y + 3n + m - y - 3n - m \rightarrow 0$

This table shows that the potential function is never negative, which means our potential function is valid. We also see that our potential function is zero when both stacks are empty. We also see that the amortized cost of MultiPopA is negative which translates in an expenditure of potential.

Citations:

[1] Cormen, Thomas H., et al. Introduction to Algorithms, Fourth Edition, MIT Press, 2022. ProQuest Ebook Central, https://ebookcentral-proquest-com.proxy1.library.jhu.edu/lib/jhu/detail.action?docID=6925615.