Brian Loughran
Johns Hopkins
Algorithms
4/12/20

1. [20 points] Write a Θ(m + n) algorithm that prints the in-degree and the out-degree of every vertex in an m-edge, n-vertex directed graph where the directed graph is represented using adjacency lists. (Hint: See Figure 2.2 on page 590 of CLRS for a diagram and description of adjacency list representations of graphs.)

```
list_in_out_degree(adj_list):
 in_degree = [0] * len(adj_list)              // create an array of 0's to represent the in-degree
 out_degree = [0] * len(adj_list)             // create an array of 0's to represent the out-degree
 for index, node in enumerate(adj_list):      // iterate over each vertex in the graph
  while node.pointer != null:                 // check that the current pointer is not null
   out_degree[index] += 1                     // add one to the out degree of the current node
   in_degree[node.pointer.value] += 1         // add one to the in degree of the destination node
   node = node.pointer                        // go to the next node in the adjacency list

 index = 0
 while index < len(adj_list):                 // starting @ index 0 print each in-degree & out_degree
  print("For node " + index+1 +
        " the in_degree is " + in_degree[index] +
        " & the out_degree is " + out_degree[index])
  index += 1                                  // move to the next index until all degrees are printed
  return
```

The algorithm here is relatively simple. The in-degree and out-degree are stored in lists where each index of the list represents the in-degree and out-degree of a particular vertex in the graph. Iterating over each vertex in the graph, we check each element in the adjacency list. For each item in the adjacency list, we add one to the out-degree corresponding to the current node number, and we add one to the in-degree corresponding to the vertex number found in the adjacency list. Thus, once we iterate over each of the vertexes in the graph, we must have a populated list of both the in-degree and out-degree for each node. Then we simply print the values as requested and return.

For the adjacency list, we assume that the adjacency list is referenced by the root item, and each item in the adjacency list has item.value which returns the vertex number that the current vertex is adjacent to, and item.pointer which returns the next item in the adjacency list. This is a pretty standard adjacency list implementation, and matches with figure 2.22 on page 590 of the book.

The runtime depends on the number of vertexes and edges in the graph. To compute the in-degree and out-degree, we iterate once over each vertex in the graph in the for() loop, and we iterate once over each edge in the graph in the while() loop. Everything otherwise contained in the loops are a finite set of instructions, thus take O(1) time. Therefore to compute the in-degree and out-degree, we take O(m+n)

time. To print the values, we iterate once over each vertex in the graph, thus the runtime is O(n) time. O(m+n) + O(n) = O(m+n), thus we have an efficient algorithm that meets the problem specifications.

2. [10 points] CLRS 9.3-9: Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2 in the textbook. Given the x- and y-coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time, and prove the correctness and linear bound of the algorithm.

To design the algorithm, we must first determine the ideal place to put the pipeline in a field of n wells. We can start out with a pipeline at a particular y coordinate and see what happens as the y coordinate changes. We note that if we are between two wells, moving the pipeline vertically will not change the total spur length of a two well system. If we have three wells, moving the pipeline toward the majority of the wells will reduce the total length of the spurs, while moving the pipeline away from the majority will increase the total length of the spurs. This leads us to the proposition that moving the main pipeline toward the median y coordinate of n well system will minimize the total length of the spurs. And as we saw in our two-well system, if we have an even number of wells, anywhere between the median two y-coordinates (y-coordinates inclusive) will produce the minimum total spur length.

Now that we know we just have to find the median of the y coordinates, we can search for an algorithm that gives us the median in linear time. Fortunately section 9.3 of the textbook provides us with an algorithm select(array, i) which returns the ith smallest value of an input array of n > 1 elements in O(n) time. We will use select() in our pseudocode below:

```
// given x, y coordinates, find optimal y position for main pipeline
main_pipeline_loc(x_coords, y_coords):
 if len(y_coords) == 0:
  return 0                                  // return 0 if there are no y coordinates
 elif len(y_coords) == 1:
  return y_coords[0]                        // return the y coordinate if there is just 1
 else:
  med_index = len(y_coords) / 2             // get the index of the array that we want (see below)
  // use select() from section 9.3 of textbook to get median value
  median = select(y_coords, med_index)
  return median
```

Brian Loughran
Johns Hopkins
Algorithms
4/12/20

We note that for an array with an odd number of elements len(array)/2 will give the middle index of the array by integer division (e.g. 5/2 = 2). We also note that for an array with an even number of elements, len(array)/2 will give the index of the upper median (e.g. 6/2 = 3). As discussed, placing the main pipeline at any location between the median two y-coordinates (y-coordinates inclusive) will produce the median total spur length. Doing the arithmetic this way will save us the trouble of checking if our y_coords array is even or odd in length while still producing an optimal result.

If the y_coords is of length 0 or 1, we of course have O(1) runtime. If the y_coords is greater than that length, we use select() from section 9.3 of the textbook which has been shown to have linear runtime and our runtime is O(n).

3. [40 points] ***Collaborative Problem***–CLRS 5-1: With a b-bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's *probabilistic counting*, we can count up to a much larger value at the expense of some loss of precision.

   We let a counter value i represent a count of n_i for $i = 0, 1, ..., 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The **Increment** operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operator reports an overflow error. Otherwise, the **Increment** operator increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it remains unchanged with probability $1 - 1/(n_{i+1} - n_i)$.

   If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the ith Fibonacci number–See Section 3.2 in the text).

   For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

   (a) [20 points] Prove that the expected value represented by the counter after n **Increment** operations have been performed is exactly n.

   (b) [20 points] The analysis of the variance of the count represented by the counter depends on the sequence of the n_i. Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Determine the variance in the value represented by the register after n **Increment** operations have been performed.

a) We know from the problem statement that the probability that the counter increments is given by:

$$\frac{1}{n_{i+1} - n_i}$$

For a binomial distribution, the expected counter value change for one increment is equal to the probability the counter remains unchanged * the counter value change + the probability the counter is incremented * the counter value change. In this case, the expected counter value change for one increment operation is:

$$\left[1 - \frac{1}{n_{i+1} - n_i}\right] * 0 + \left[\frac{1}{n_{i+1} - n_i}\right] * n_{i+1} - n_i = 1$$

Thus the expected value change for one increment is 1. For the case with n increments, it follows that the expected value change is 1 * n = n. Thus the expected value represented by the counter after n increment operations has been performed is n, as expected.

Brian Loughran
Johns Hopkins
Algorithms
4/12/20

b) Given ni = 100i, the expected value change for the increment operation is 100, and that increment operation will happen once out of every 100 times. The variance for a distribution such as the one above is given by the expression:

$$var(x) = E(X^2) - (E(X))^2$$

We already know the expected value of E(x) from part (a) = 1. $E(X^2)$ is given by the following equation, which the probability that the counter is not incremented * the square of the value added to the counter when it is not incremented + the probability that the counter is incremented * the square of the value added to the counter when it is incremented.

$$E(X^2) = \left[\frac{99}{100} * 0^2\right] + \left[\frac{1}{100} * 100^2\right] = 100$$

Thus the variance of the distribution for a single increment operation is:

$$var(x) = 100 - (1)^2 = 99$$

And it follows that for n increment operations, a variance of 99 * n operations will give a variance of 99n after n increment operations have been performed.

4. [30 points] ***Collaborative Problem***– In this problem we consider two stacks, A and B, manipulated using the following operations (n denotes the size of A and m the size of B):

   - $PushA(x)$: Push element x on stack A.
   - $PushB(x)$: Push element x on stack B.
   - $MultiPopA(k)$: Pop $min(k, n)$ elements from A.
   - $MultiPopB(k)$: Pop $min(k, m)$ elements from B.
   - $Transfer(k)$: Repeatedly pop an element from A and push it on B, until either k elements have been moved or A is empty.

   Assume that A and B are implemented using doubly-linked lists such that $PushA$ and $PushB$, as well as a single pop from A or B, can be performed in $O(1)$ time worst-case.

   (a) [5 points] What is the worst-case running time of the operations $MultiPopA$, $MultiPopB$, and $Transfer$?
   (b) [25 points] Define a potential function $\Phi(n, m)$ and use it to prove that the operations have amortized running time $O(1)$.

a) The worst case runtime for MultiPopA and MultiPopB would be when either have to pop the entire stack. Since Pop() from either list is defined as being O(1), doing this O(n) times would result in O(n) runtime for MultiPopA, and doing this O(m) times would result in O(m) runtime for MultiPopB.

Transfer() is defined as performing Pop() on A and Push() on B. Both Pop() and Push() are defined as O(1) runtime, thus the runtime for each element is O(1). The worst case for Transfer()

is if we transfer the whole stack from A to B, thus the worst case runtime for a stack in A of size n is O(n).

b) We define a potential function to be linear to the size of the two stacks, or:

$$\Phi(\mathrm{n}, \mathrm{m}) = 3\mathrm{n} + \mathrm{m}$$

Initially the function is positive, and for a stack of non-negative size (which is invalid) the potential is always positive. Thus this potential function meets the criteria required for a potential function. According to equation 17.2 in the textbook, the amortized cost of each of the functions (PushA, PushB, MultiPopA, MultiPopB, and Transfer) is of the form:

$$\acute{c}_i = c_i + \Phi(\mathrm{D_i}) - \Phi(\mathrm{D_{i-1}})$$

Thus we can solve for the amortized costs for each of the five functions requested. The amortized cost is given by the equation 17.2 in the textbook. For PushA we have:

$$\begin{aligned}\acute{c}_i &= 1 + [3(n+1) + m] - [3n + m] \\ &= 4\end{aligned}$$

For PushB the amortized cost is:

$$\begin{aligned}\acute{c}_i &= 1 + [3n + m + 1] - [3n + m] \\ &= 2\end{aligned}$$

For MultiPopA(k) we have the amortized cost

$$\begin{aligned}\acute{c}_i &= k + [3(n-k) + m] - [3n + m] \\ &= -2k\end{aligned}$$

For MultiPopB(k) we have the amortized cost:

$$\begin{aligned}\acute{c}_i &= k + [3n + (m-k)] - [3n + m] \\ &= 0\end{aligned}$$

Next we can take a look at Transfer(k). The amortized cost for Transfer(k) is:

$$\begin{aligned}\acute{c}_i &= 2k + [3(n-k) + (m+k)] - [3n + m] \\ &= 0\end{aligned}$$

Now we see why we chose a constant of 3 for the n term. Using 3n we are able to apply an upper bound the amortized cost for each of the functions by a constant. And since the potential function does not break any of the rules required for a valid potential function, and the amortized cost for each of the five functions is upper bounded by a constant (4), the amortized cost must be O(1) for each.