Homework 3

Joni Vrapi

10/09/2022

Statement of Integrity: I, Joni Vrapi, attempted to answer each question honestly and to the best of my abilities. I cited any, and all, help that I received in completing this assignment.

Problem 1. We are asked to use indicator random variables to compute the expected value [1] of the sum of n dice. First, we figure out the expected value of rolling just one die:

$$E[X] = \sum_{i=1}^{6} iP(X_i) = \frac{1}{6}\sum_{i=1}^{6} i \tag{1}$$

From [2] we know that

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2} \tag{2}$$

Then, the sum of n dice has the probability

$$E[nX] = nE[X] = \frac{n}{6}\sum_{i=1}^{6} i = (\frac{n}{6})\frac{m(m+1)}{2} = \frac{n6(6+1)}{12} = 3.5n \tag{3}$$

Problem 2. Intuitively, PERMUTE-WITH-ALL results in n^n possible combinations. Mathematically, however, we know that there are only $n!$ possible combinations. This discrepancy tells us that some permutations must be overrepresented. If we consider the simplest case where $n = 3$, we can see that PERMUTE-WITH-ALL will produce $3^3 = 27$ orderings, while the correct number of combinations should be $3! = 6$. 6 does not divide 27, therefore there will not be a random uniform permutation. In fact, $n!$ in general does not divide n^n.

Problem 3a. For this problem, I was not sure what exactly the "basic linear in-place algorithm on the internet" was, nor was I able to figure out a way to improve what I assume are optimal solutions that I did find on the internet. Something I noticed, however, was that all the solutions on the internet use pointers to organize the colors, which I personally find difficult to comprehend. After reading the Bucket Sort section of our Module content, however, I believe a much simpler solution, without pointers, to this problem can be achieved.

If we choose to represent our variables like so (for simplicity of the algorithm) $R \to 0$, $W \to 1$, $B \to 2$, we can construct a solution to this problem with bucket sort like so:

```
BUCKET-SORT-DNF(A)
 1  counts = [0, 0, 0]
 2
 3  for item in A
 4       counts[item]++
 5
 6  for (item, index) in A
 7       if counts[0]−− > 0
 8            A[index] = 0
 9       elseif counts[1]−− > 0
10            A[index] = 1
11       else
12            A[index] = 2
13  return A
```

An *in-place* algorithm is an algorithm that does not need extra memory to perform whatever action is required of it, except for any constant amount of memory that is required for its operation, irrespective of input size. For example, some programming languages allow you to swap values in an array without the use of a temporary variable. Other languages do not allow you to swap values in an array, mandating the use of a temporary swap variable. With even those languages, however, the compiler is smart enough to recognize that you are doing a swap, and will write the optimal code that does so for you. This bucket sort implementation is therefore an in-place algorithm. It requires only the 'counts' array on line 1 of the program, which is of constant size, to perform the sort.

Problem 3b. Asymptotically, there are two loops in this pseudocode which both iterate through every item in $\boldsymbol{A}$ exactly once. This results in a time complexity of $O(n) + O(n) = O(n)$, keeping this solution to *DNF* linear, as well as in-place.

Problem 4. My algorithm describes this as a two-step process. First, assuming we have a 2 dimensional array, filled with k sub-arrays, we flatten that array to produce an output array of 1 dimension. While each k sub-array is sorted, the flattened array is not guaranteed to be sorted. Therefore, we use heap sort [3] with a max-heap to sort the remaining list in descending order.

```
HEAPIFY(A, n, i)
 1  largest = i
 2  left = 2 * i + 1
 3  right = 2 * i + 2
 4
 5  if left < n and A[left] > A[largest]
 6       largest = left;
 7
 8  if right < n and A[right] > A[largest]
 9       largest = right;
10
11  if largest ≠ i
12       A[largest], A[i] = A[i], A[largest]
13       HEAPIFY(A, n, largest)
```

```
SORT(A)
1  n = FLATTEN(A).length
2  for i = ⌊n/2⌋ − 1; i ≥ 0; i++
3        HEAPIFY(A, n, i)
4
5  for i = n − 1; i > 0; i − −
6        A[0], A[i] = A[i], A[0]
7        HEAPIFY(A, i, 0)
8
9  return A
```

```
FLATTEN(A)
1  return A.flat(1)
```

To drive this code you would just pass in a 2d array to the SORT(A) method.

For the first step, flattening the input array, each element in the 2d array is touched only once, therefore it is an $O(n)$ operation. Heap sort is known from [4] to be an $O(nlog(n))$ algorithm, and because we have k lists and n total elements, for us it is $O(nlog(k))$. We therefore have $O(n) + O(nlog(k)) = O(nlog(k))$.

Problem 5. My algorithm will combine a Fisher-Yates Shuffle [5] with a counting sort [6] to produce a nondeterministic linear time sorting algorithm over a set of integers. It will accept an array and input n which signifies the maximum integer value to be found in the array, apply a Fisher-Yates shuffle to the input array, and then, since the maximum integer value of the array is known, will apply a counting sort to it.

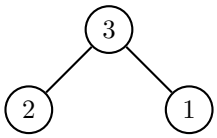
```
FISHER-YATES(A)
1  i = A.length
2
3  while − − i > 0
4        temp = ⌊Math.random ∗ (i + 1)⌋
5        A[temp], A[i] = A[i], A[temp]
6
7  return A
```

```
NSort(A, n)
 1  A = Fisher-Yates(A)
 2
 3  counts = new Array(n + 1).fill(0)
 4  for item in A
 5        counts[item]++
 6
 7  numItemsBefore = 0
 8  for i = 0; i < n; i++
 9        temp = counts[i]
10        counts[i] = numItemsBefore
11        numItemsBefore += temp
12
13  sortedArray = new Array(A.length).fill(0)
14  for item in A
15        sortedArray[counts[item]] = item
16        counts[item] += 1
17
18  return sortedArray
```

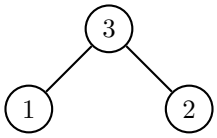
There are three loops in NSort and one in Fisher-Yates, all of which iterate over each element in A exactly once. This makes the total time complexity of the algorithm $O(n) + O(n) + O(n) + O(n) = O(n)$.

Problem 6. Do BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array? No. For example, if $\boldsymbol{A} = [1, 2, 3]$, then

BUILD-MAX-HEAP will produce:



While BUILD-MAX-HEAP' will produce:



From CLRS 6.1 [4] we know that the insertion operation into a heap is $\Theta(log(n))$. Since we are inserting n times, we get a bound on the runtime of $\Theta(nlog(n))$.

Problem 7. Solving the array of integers problem: Radix sort takes an array of n integers, in base b, where each number has at most d digits and $d = \lfloor log_b(k) + 1 \rfloor$ where k is the largest number in the array. From [7] we know that, when b and n are of the same size magnitude, radix sort will run in $O(d(n + b)) = O(n)$ time.

First, we go through the array, putting each number in n buckets by number of digits (e.g. [1, 2, 3], [12, 23, 34]...). This takes $O(n)$ steps. The only thing that remains is to sort the numbers within the n buckets. If we assume there are k_i numbers with i digits, then radix will sort these numbers digit by digit in $O(ik_i)$ with i iterations of counting sort over k_i numbers. To sort all the buckets we would sum this for $O(\sum ik_i)$. Since we know from the problem that the total number of digits is n, we know that $\sum ik_i = n$. Therefore, the total number of steps for the whole algorithm would be $O(n) + O(\sum ik_i) = O(n) + O(n) = O(n)$.

Problem 8a. Given this random process, the expected number of incoming links to node v_j is as follows

$$\sum_{x=j}^{n-1} \frac{1}{x} \tag{4}$$

When $n = 1$, indicating you have just 1 node, this summation does not occur, and the result is 0 incoming links. When $n = 2$, indicating you have 2 nodes, the expression evaluates to 1, indicating that node v_j (the first node) has 1 incoming link. When $n = 3$, indicating you have 3 nodes, the expression evaluates to 1.5 indicating that node v_j (either node 1 or 2 in this case) has an expected 1.5 incoming links.

In order to express this asymptotically via a tight bound without summations, we can take the indefinite integral of $\frac{1}{x}$.

$$\int \frac{1}{x} = ln(x) + c = \Theta(ln(x)) \tag{5}$$

Problem 8b. Intuitively, the worst possible organization of this network (or rather, a subsection of it as it is peer-to-peer and a node may not have to traverse the whole network to get whatever content it needed) would be a "linked list" of nodes, where each node points to the previous node. As this type of network grows, the nodes that came later would experience slower and slower connection speeds because they would be proxying through every node that came before them. Likewise, the best possible configuration would be to arrange nodes in a tree-like structure, keeping the overall depth of the network as shallow as possible while not giving any single node too many incoming connections so as to overload them. In keeping with this intuition, we can write an expression to formulate the expected number of nodes with no incoming connections like so:

$$\sum_{j=1}^{n} \prod_{x=j}^{n-1} (1 - \frac{1}{x}) = \frac{n}{2} \tag{6}$$

Here we add a term in front of the $\frac{1}{x}$ to signify that you will always have at least one node with no incoming connections – in the case where you only have one node. Likewise, we need to sum a series of products which correspond to the probabilities that nodes have incoming connections in order to evaluate this properly. Courtesy of Wolfram Alpha [8], we were able to show that this goes to $\frac{n}{2}$.

References

[1] "Random variables." https://www.cs.princeton.edu/courses/archive/fall02/cs341/lec22.pdf. Accessed on 2022-10-09.

[2] "List of mathematical series." https://en.wikipedia.org/wiki/List_of_mathematical_series. Accessed on 2022-10-09.

[3] "Heap sort." https://www.geeksforgeeks.org/heap-sort/. Accessed on 2022-10-09.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. The MIT Press, 4 ed., 2022.

[5] "Fisher yates shuffle." https://www.tutorialspoint.com/what-is-fisher-yates-shuffle-in-javascript. Accessed on 2022-10-09.

[6] "Counting sort." https://www.interviewcake.com/concept/javascript/counting-sort. Accessed on 2022-10-09.

[7] "Radix sort." https://brilliant.org/wiki/radix-sort/. Accessed on 2022-10-09.

[8] "Wolfram alpha." https://www.wolframalpha.com/input?i=sum+from+j%3D1+to+n+of+%28product+from+x%3Dj+to+%28n-1%29+of+%281-1%2Fx%29%29. Accessed on 2022-10-09.