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 3.

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. Finally, 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.

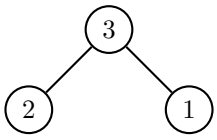
```
FISHER-YATES(A)
1  i = A.length
2
3  while − − i > 0
4        temp = ⌊Math.random ∗ (i + 1)⌋
5        A[temp], A[i] = A[i], A[temp]
6
7  return A
```

```
COUNTING-SORT(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
```

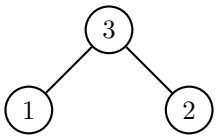
There are three loops in COUNTING-SORT and one in FISHER-YATES, all of which iterate over each element in A exactly once. This makes the total time complexity of the algorithm $O(n) + O(n) + O(n) + O(n) = O(n)$.

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 is $O(log(n))$. Since we are doing it n times, we get a bound on the runtime of $O(nlog(n))$.

Problem 7. Solving the array of integers problem: Radix sort takes an array of n integers, in base b, where each number as 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 then, 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 8.

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.