

Spencer Shimko
CMSC 441
HW 3

1) 5.2-4

Use indicator random variables to solve the following problem, which is known as the hat-check problem. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers that get back their own hat?

Let: X_1, X_2, \dots, X_n be the r.v. s.t.:

$$X_i = 1 \quad \text{If person gets right hat}$$
$$X_i = 0 \quad \text{else}$$

Y = total persons get right hat
 $Y = \sum X_i$

$$E[Y] = E[\sum X_i] = \sum E[X_i]$$

$P(X_i) = \text{prob. that person receives right hat}$
 $E[X_i] = P(X_i=1) = (1/n)$
 $E[Y] = \sum E[X_i] = \sum (1/n) = (1/n) \sum 1 = (1/n) * n = 1$

2) 6.5-7

The operation **HEAP-DELETE(A, i)** deletes item in node i from heap A . Give an implementation of **HEAP-DELETE** that runs in $O(\lg n)$ time for an n -element max-heap.

Given: Max-Heap A ; item i
Return: Max-Heap A w/ i removed

```
-Switch  $A[i]$  and  $A[\text{last}]$  ( $\text{last}$  = last index of heap)
-Remove that last element or  $A$  (now  $i$ ) by shrinking heap
size by 1
-If  $i$  was the last element return (prevent the following
operations on non-existent element)
-Store element  $A[i]$  in  $\text{key}$ 
-If  $\text{key} \leq A[\text{parent}]$  ( $\text{parent}$  = index of parent node in heap)
    -Recursively MAX-HEAPIFY( $A, i$ )
-Else
    - while  $i \neq 0$  and  $A[\text{parent}(i)] < \text{key}$ 
        -Switch  $A[i]$  and  $A[\text{parent}]$  ( $\text{parent}$  = index of
parent node in heap)
        - $i = \text{Parent}(i)$ 
    -End
```

-End

Quick explanation: switches item i with the last item in heap/array and then removes it by shrinking the array size; test array size to make sure i is still a valid index; set node to key; then either "heapify up" or "heapify down" depending on updated heap subtree.

3) The running time of quicksort can be improved in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the array. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should k be picked, both in theory and practice?

$T(n)$ = run time of quicksort + running time of insertion sort
 $= n(\log n - \log k) + n/k(k^2) = n \log(n/k) + nk$
note: insertion sort runtime = n^2 ($n = k$ for this scenario)

Practice: k needs to be the maximized s.t.:
-still allows insertion sort to out-perform quicksort

Theory: k needs to be maximized s.t.:
 $-nk + n \log(n/k) = n \log n = k = \log n$

4) 5.2-2

In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the possibility that you will hire exactly twice?