CS302 - Problem Set 4

Due: Monday, Oct 9. Must be uploaded to Canvas before the beginning of class.

I'm introducing a new rating system. (Apologies in advance if this does not correspond to your experience. After all, what is challenging for each person is different. My goal in doing this is to help you budget your time/energy better on the problem sets.)

- * Straightforward, should be pretty easy.
- ** Less straightforward, but no out-of-the-box thinking required.
- *** Challenging. Requires creativity/out of the box thinking to solve, and usually more complicated mathematical analysis.

1. *Indicator Variables* Recall that an indicator random variable X_A takes value 1 if event A happens, and takes value 0 if event A does not happen. Suppose you need to run J jobs, and you have Q processors to which you can assign jobs. So for example, you could assign job 1 to processor 10, job 2 to processor 5, job 3 to processor 10 (again), etc. Now suppose your assignment strategy is to assign each job to a processor uniformly at random (i.e. with equal probability).

   (a) [**3 points**] * What is the sample space in this problem, and what is the size of the sample space?

   (b) [**6 points**] *(*) Use indicator random variables and linearity of expectation to argue that the expected number of jobs each processor will run is J/Q.

2. *Randomized Search* Consider an algorithm which searches an array A of length n for a value x. In each round of the algorithm, an index of the array is chosen uniformly at random, with equal probability of choosing any of the n indices. The algorithm then checks if that index in the array contains x. The algorithm is not particularly smart, because it could repeatedly check indices that it has already previously verified do not contain x. (This is called *sampling with replacement.*) Once an index i such that $A[i] = x$ is found, the algorithm terminates. Otherwise, if the algorithm does not contain x, once the algorithm has looked at all indices it terminates.

   (a) **6 points** ** Use conditional probability to determine the probability that $K + 1$ rounds of the algorithm occur, for $K \in \mathbb{Z} : K \geq 1$, if the algorithm does contain x.

   (b) [**11 points**] ** Suppose there is exactly one index i in the array, such that $A[i] = x$. Use the solution to the previous part, and indicator random variables, to prove the value of the expected number of rounds before x is found.

(c) **[11 points]** *** In the case that the array does not contain the value x, prove an upper bound on the expected number of rounds before the algorithm terminates. You may find the following bound useful:

$$\sum_{i=1}^{n-1} \frac{1}{g-i} \leq \ln(n-1)+1 \tag{1}$$

3. *Randomized Selection* `Selection` is an algorithm that is similar to QuickSort. Instead of sorting the array, it finds the k^{th} smallest element of an unsorted array. (Also called the kth order statistic) Here is psuedocode:

```
Algorithm 1: Selection($A, s, f, k$)
Input  : A global array $A$ of integers (not necessarily unique), start index $s$, final
         index $f$, and an integer $k \in \{1, 2, \ldots, f-s+1\}$
Output: The $k$th order statistic of $A[s:f]$ (the subarray of $A$ from $s$ to $f$
         inclusive).
1  $piv$ = ChoosePivot($s, f$);
   // Chooses a pivot between $s$ and $f$ inclusive using some procedure
2  Run Partition on elements of $A$ from $s$ to $f$ (inclusive) using $piv$ as pivot;
3  Let $p$ be the index of the pivot after Partition;
4  if $p = k$ then
5      return $A[p]$;
6  end
7  if $p < k$ then
8      return Selection($A, p+1, f, k-p$);
9  else
10     return Selection($A, s, p-1, k$);
11 end
```

(a) **[3 points]** * Use the master method to determine the runtime of this algorithm if `ChoosePivot` somehow always chooses the pivot to be the middle order statistic (i.e. the $(n/2)^{\text{th}}$ largest element in an array of size n).

(b) **[11 points]** *** Now consider the case that `ChoosePivot` chooses a pivot uniformly at randomly from s to f inclusive. Consider the indicator variable X_{ij}, for $i < j$, which takes value 1 if the i^{th} and j^{th} largest elements of A are compared at some point in the algorithm. Prove that the average number of comparisons over the course of the algorithm is the same as what you found in part (a). (Your strategy should be similar to what we did in class/video, but the analysis is more complicated.)