CS 515: Written Assignment #3

Due on Tue, 11/15/16

Amir Nayyeri
6:00-7:20pm, T,Th

Chen Hang, 932-727-711
Kaibo Liu, 932-976-427

11/11/2016

Contents

Problem 1

[**40 pts**] Suppose you are given n bolts and n nuts. The bolts have distinct sizes. Also, the nuts have distinct sizes. Each bolt match exactly one nut.

Consider the following randomized algorithm for choosing the **largest** bolt. Draw a bolt uniformly at random from the set of n bolts, and draw a nut uniformly at random from the set of n nuts. If the bolt is smaller than the nut, discard the bolt, draw a new bolt uniformly at random from the unchosen bolts, and repeat. Otherwise, discard the nut, draw a new nut uniformly at random from the unchosen nuts, and repeat. Stop either when every nut has been discarded, or every bolt except the one in your hand has been discarded.

(a) What is the probability that the algorithm discards NO bolts?

- Lemma 1.1:
  No bolts are discarded if and only if when the first chosen bolt is the biggest bolt.
- Proof of Lemma 1.1:
  Name the first chosen bolt as b1.
  Suppose if b1 is not the biggest bolt, to find the largest bolt by using this algorithm, b1 must be discarded in the end and finally be replaced by the biggest bolt. Thus, if no bolts are allowed to be discarded, b1 must be the biggest bolt.
  On the other hand, if b1 is the biggest bolt. Then the algorithm would just discard all the nuts left and finally return the biggest bolt. Therefore, in this case, no bolts are discarded.

As stated in lemma 1.1, the problem is reduced to find the probability of first chosen bolt is the biggest bolt. The answer is $\frac{1}{n}$.

(b) What is the exact expected number of discarded bolts when the algorithm terminates?
Suppose a bolt bolts[i] in all bolts bolts[1..n] means the i^{th} bolt randomly selected by the algorithm. Then we have the lemma 1.2.

- Lemma 1.2:
  k$(0 \leq k \leq n-1)$ bolts are discarded if and only if bolts[k+1] is the biggest bolt.
- Proof of Lemma 1.2:
  k is always smaller than n because the worst case is that only the biggest bolt left. That means at least one bolt should be left.
  As stated in lemma 1.1, no bolts are discarded happens if and only if when bolts[1] is the biggest bolt. We can also say that the first bolt is discarded if and only if when the first bolt is not the biggest bolt.
  Suppose the lemma 1.2 works for discarding k-1 bolts.
  For any i in [1..k], if bolts[1..i-1] are discarded by using lemma 1.2, bolts[k] is discarded if and only if first selected but not the biggest in bolts[k..n]. Then for bolts[k+1..n], bolts[k+1] is kept if and only if it is first selected bolt and also the biggest one. Thus, the lemma 1.2 is correct.

By using lemma 1.2, we know that discarding k bolts means bolts[k+1] is the biggest bolt. Then we need to find each probability of k in [0,1,..,n-1].
Set Pr[k=i] means the probability of k equals to i.
Pr[k=0]=$\frac{1}{n}$ is from problem (a).
If k is larger than 0, at least 1 bolt would be discarded. In the first step, for bolts[1], its probability of being discarded is $\frac{n-1}{n}$. Then in the second step, for bolts[2], its probability of being discarded is $\frac{n-2}{n-1}$ because there are only n-1 bolts left. Similarly, in the $(i)^{th}$ step, for bolts[i], probability for being discarded is $\frac{n-i}{n-i+1}$. However, in $(i+1)^{th}$ step, we need to keep bolts[i+1], so the probability should be

$\frac{1}{n-i}$. Therefore, we have the equality that $Pr[k = i](1 \le i \le n-1)$=$\frac{n-1}{n} * \frac{n-2}{n-1} * .. * \frac{n-i}{n-i+1} * \frac{1}{n-i}$=$\frac{1}{n}$. Then, we have:

$$E(k) = \sum_{i=0}^{n-1} i * Pr(k = i) = \frac{1}{n} * \sum_{i=0}^{n-1} i = \frac{1}{n} * \frac{n*(n-1)}{2} = \frac{n-1}{2}$$

.

(c) What is the probability that the algorithm discards exactly one nut?

- Lemma 1.3:
  Nuts are not all discarded if and only if bolts[n] is the biggest bolt.
- Proof of Lemma 1.3:
  If bolts[n] is not the biggest bolt, then the biggest bolt is before bolts[n]. Suppose bolts[i] is the biggest bolt, then bolts[i..n] could be all kept and nuts would all be discarded. So if nuts are not all discarded, bolts[n] must be bolts[i].
  On the other hand, if bolts[n] if the biggest bolt, then bolts[1..n-1] must be smaller than the biggest nut. So bolts[1..n-1] must be discarded because of the biggest nut. Then only bolts[n] left. So, in this case, algorithm would return bolts[n] and keep all nuts from biggest nut.

Let's set nuts[j] in nuts[1..n] means the j^{th} selected nut. We can also set x is the number of discarded nuts. Then, we have lemma 1.4.

- Lemma 1.4:
  x nuts($0 \le x \le n-1$) are discarded if and only if both (nuts[x+1] is the biggest nut) and (bolts[n] is the biggest bolt).
- Proof of Lemma 1.4:
  By lemma 1.3, we know that not all nuts are discarded if and only if bolts[n] is the biggest bolt. Then, this case is similar to lemma 1.2.

Similar to problem b, probability for x in each valid number (from 0 to n-1) is $\frac{1}{n}$. However, we still need to consider the probability of bolts[n] is the biggest bolt. Then Pr(x=j)($0 \le j \le n-1$)=$\frac{1}{n^2}$. Therefore, Pr(x=1)=$\frac{1}{n^2}$.

(d) What is the exact expected number of discarded nuts when the algorithm terminates?
Pr(x=n)=1-Pr(k=n)=$\frac{n-1}{n}$.(we use the k in problem c here) Then, we have the equality:

$$\begin{aligned} E(x) &= \sum_{j=0}^{n} j * Pr(x = j) = (\frac{1}{n^2} * \sum_{j=0}^{n-1} j) + n * \frac{n-1}{n} \\ &= (\frac{1}{n^2} * \frac{n*(n-1)}{2}) + n - 1 = \frac{(n-1)}{2n} + n - 1 \\ &= \frac{2n^2 - n - 1}{2n} \end{aligned}$$

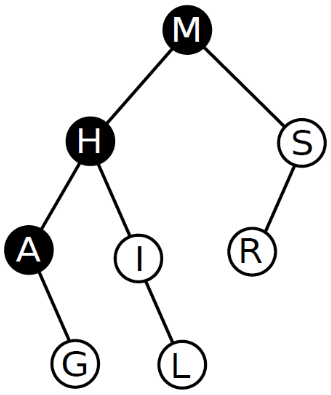
(e) What is the exact expected number of nut-bolt tests performed by this algorithm?
In each comparison, there is either a bolt or a nut being discarded, so we assume t is the totally number of comparisons. As mentioned before, k is the number of discarding bolts, x is the number of discarding nuts. Then, $t = x + k$. We have:

$$E(t) = E(x + k) = E(x) + E(k) = \frac{3n^2 - 2n - 1}{2n}$$

Problem 2

[**40 pts**] Consider a randomized treap T with n vertices.

(a) The left spine of a binary tree is a path starting at the root and following left-child pointers down to the rst node with no left child pointer. What is the expected number of nodes in the left spine of T? For example, the number of nodes on the left spine of the following tree is 3, all black nodes. [Hint: What is the probability that a node is on the left spine.]



Let's traverse the treap by using inorder sequence. Then, the sequence is sorted by keys increasingly. Set the sequence as nodes[1..n]. Then, we know that nodes[i].key<nodes[j].key if and only if i<j. In Jeff Ericksons lecture 10, we know that if nodes[i] is parent or ancestor of nodes[j](i!=j), nodes[i] should have the smallest priority among nodes[i..j](if $i < j$) or nodes[j..i](if $i > j$). This lemma is not proved in this problem.
By definition of left spine and binary tree's inorder sequence, we can find that nodes[1] must be in left spine.
Then, all ancestors of nodes[1] should be in left spine.
Thus, the problem is reduced to find the expected depth of nodes[1].
We set a function LS(i) to return whether the node is in left spine. We can also say whether the node is nodes[1] or ancestor of nodes[1]. If true, return 1. Else, return 0.
We set "depth" to be the depth of nodes[1], it is the number of nodes in the left spine as well, denoted as N_{LS}. Then, have the equality:

$$E(N_{LS}) = E(depth) = \sum_{i=1}^{n} Pr[LS(i) = 1]$$

Then, we need to compute Pr[LS(i)=1] for each i in [1..n].
If i is 1, Pr[LS(i)=1]=1.
If i is not 1, Pr[LS(i)=1] if and only if nodes[i] is ancestor of nodes[1]. In this case, nodes[1].priority is the smallest priority in nodes[1..i]. Then, the probability is $\frac{1}{i}$.
Therefore, we can say:

$$E(depth) = \sum_{i=1}^{n} Pr[LS(i) = 1] = \sum_{i=1}^{n} \frac{1}{i} = H(n)$$

$$E(N_{LS}) = E(depth) = H(n) \geq \sum_{i=1}^{n} ln(1 + \frac{1}{i}) = ln(\prod_{i=1}^{n} \frac{i+1}{i}) = ln(1+n)$$

(b) What is the expected number of leaves in T? [Hint: What is the probability that a node is a leaf?]
Define adjacent nodes of nodes[i] as nodes[i-1] and node[i+1]. If i is 1, its only adjacent node is nodes[2]. If i is n, its only adjacent node is nodes[n-1]. If n is 1, there is no adjacent node for nodes[1].
Then, we have the lemma 2.1.

– Lemma 2.1
nodes[i] is a leaf if and only if nodes[i] has no adjacent nodes or has a bigger priority than all its adjacent nodes.

– Proof of Lemma 2.1:
If nodes[i] has no adjacent nodes, there is only one node in the treap. The root is a leaf.
If there are more than 1 nodes in the treap, each node has at least one and at most two adjacent nodes. If nodes[i] has right adjacent node, then i is smaller than n-1. In this case, if nodes[i].priority<nodes[i+1].priority, then nodes[i] is ancestor of nodes[i+1]. To ensure nodes[i] has no right branch, nodes[i+1] must be smaller than nodes[i]. On the other hand, if nodes[i+1]<nodes[i], nodes[i].priority must not be the smallest in all priorities among nodes[i..j] if $i < j$, then nodes[i] would never have right branch. It's the same that nodes[i]($i > 1$) doesn't have left branch if and only if nodes[i].priority > nodes[i-1].priority.
We know a node is a leaf if and only if it has neither left branch nor right branch(definition of a leaf in a binary tree). Thus, lemma 2.1 is correct.

For n=1, the number of leaves is always 1, so E(leaves)=1;
For n=2, each node has $\frac{1}{2}$ probability to be leaf, therefore, E(leaves)=$\frac{1}{2}+\frac{1}{2}$=1;
For n≥3, there exists at least one node containing two adjacent nodes. For nodes[1] it has $\frac{1}{2}$ probability that nodes[1].priority > nodes[2].priority(nodes[1] is a leaf). The same as nodes[n], nodes[n] also has $\frac{1}{2}$ probability to be a leaf. For i($1 < i < n$), it could be a leaf if and only if nodes[i].priority=max{nodes[i-1].priority, nodes[i].priority, nodes[i+1].priority}. In this case, nodes[i] has $\frac{1}{3}$ probability to be a leaf. So we can conclude that:

$$E(leaves) = 2 * (\frac{1}{2}) + (n-2) * \frac{1}{3} = \frac{n+1}{3}, \qquad n \geq 3$$

So the result is:

$$E(leaves) = \begin{cases} 0 & , n = 0 \\ 1 & , n = 1 \\ \dfrac{n+1}{3} & , n \geq 2 \end{cases}$$

Problem 3

[**20 pts**] Suppose that the worst case expected running time of an algorithm on an input of size n is given by $T(n)$. Find an upper bound on the probability that the algorithm takes more than $10 \cdot T(n)$ on any given input. [Hint: study Markov inequality]

Let t_n denotes the worst case running time of the algorithm with an n-size-input. We have

$$E[t_n] = T(n)$$

According to Markov inequality, we have

$$Pr(t_n \geq 10T(n)) \leq \frac{E[t_n]}{10T(n)} = \frac{T(n)}{10T(n)} = \frac{1}{10}$$