UNIVERSITY OF TORONTO
Faculty of Arts and Science

Midterm 1, Version 2
CSC263H1F

October 14 2016, 1:10-2:00pm (**50 min.**)
Examination Aids: No aids allowed

Name:

Student Number:

Please read the following guidelines carefully!

- Please write your name on the front **and back** of the exam.
- This examination has **4** questions. There are a total of **11 pages, DOUBLE-SIDED**.
- Answer questions clearly and completely. Give complete justifications for all answers unless explicitly asked not to. You may use any claim/result from class, unless you are being asked to prove that claim/result, or explicitly told not to.

Take a deep breath.

This is your chance to show us

How much you've learned.

We **WANT** to give you the credit

That you've earned.

A number does not define you.

Good luck!

1. Consider the following algorithm, which checks whether an array has duplicates.

```
def has_duplicate(array):
  i = 0
  while i < array.length:
    j = i + 1
    while j < array.length:
      if array[i] == array[j]:
        return True
      j = j + 1
    i = i + 1
  return False
```

(a) **[2 marks]** Let n be the length of `array`. Prove that the worst-case running time of this algorithm is $\Omega(n^2)$. (Note the Big-**Omega**.)

Solution

Input family: array of length n that contains no duplicates.

In this case, the check `array[i] == array[j]` will always be false, so the inner loop never stops early.

For each i, the number of iterations of the inner loop is $n-1-i$, so the total number of repetitions of the inner loop code (which takes constant time) is $(n-1)+(n-2)+(n-3)+\cdots+2+1 = \Omega(n^2)$, so the overall running time is $\Omega(n^2)$.

[Comment: in fact the running time for this input family is $\Theta(n^2)$. But since we're only proving a lower bound on the worst-case running time, we only need a lower bound on the running time for this input family.

Some students were penalized for over-counting the number of iterations of the inner loop (n instead of $n-1-i$).]

(b) **[2 marks]** Consider running `has_duplicate` on an array of length n, where the each item in `array` is independently chosen uniformly at random from the range 1 to n, inclusive. For each $0 \leq k < n$, let A_k be the following event:

(i) The array elements `array[0]`, `array[1]`, ..., `array[k]` are all distinct.

(ii) For all $j > k$, `array[j]` is not equal to any of the array elements between indices 0 and k, inclusive.

Determine, with justification, the probability that A_k occurs (in terms of k). You may write your answer in terms of factorials, binomial coefficients $\binom{a}{b}$, and/or summations or products; no need to simplify fully.

Hint: duplicates *are* allowed in the indices after k.

Solution

In terms of binomial coefficients, there are $\binom{n}{k+1}$ choices for the first $k+1$ array elements, which can be arranged in $(k+1)!$ ways. With these $k+1$ elements chosen, there are $n-k-1$ possibilities for the remaining $n-k-1$ array elements, for a total of $\binom{n}{k+1}(k+1)!(n-k-1)^{n-k-1}$. Dividing this by the total number of lists yields the probability:

$$\frac{\binom{n}{k+1}(k+1)!(n-k-1)^{n-k-1}}{n^n}.$$

[Comment: it's possible to express the probability of the first part as $\frac{n}{n} \cdot \frac{n-1}{n} \cdot \cdots \cdot \frac{n-k}{n}$.]

(c) **[2 marks]** Let $0 \leq k < n$. Assuming that A_k occurs, find a good lower bound on the total number of comparisons `array[i] == array[j]` made by the algorithm.

Like part (b), it is fine to not fully simplify your expression, and leave summations in your final answer. We are looking more for English justification of your expression here.

Solution

If A_k occurs, this means that for the first $k+1$ values of i (from 0 to k), the inner loop must repeat all $n-i-1$ times, because the comparison `array[i] == array[j]` will always be false.

So then the number of comparisons is at least $\sum_{i=0}^{k} n-i-1$.

2. Suppose we want to implement the following operation:

   - MergeHeaps($heap1, heap2$): return a new heap containing all the elements of $heap1$ and $heap2$, including duplicates. Do *not* modify the two input heaps.

   (a) **[2 marks]** Show how to implement `MergeHeaps` in $\mathcal{O}(n_1 + n_2)$ time in the worst case, where n_1 and n_2 are the number of items in $heap1$ and $heap2$, respectively.

   Only give the algorithm in this part; do not do the running time analysis here.

   Hint: recall Heapsort from lecture.

Solution

The key idea is that it is possible to take any list and rearrange its elements to satisfy the heap property in time linear in the length of the list.

```
1 def MergeHeaps(heap1, heap2):
2   new_heap = new heap of size (heap1.size + heap2.size)
3   copy all elements of heap1 and heap2 into new_heap
4   BuildHeap(new_heap)
5   return new_heap
```

   (b) **[1 mark]** Show that your algorithm in part (a) has worst-case running time $\Omega(n_1 + n_2)$.

Solution

Calling BuildHeap both take time linear in the total number of elements, which is $n_1 + n_2$, for any input.

[Comment: so in fact the algorithm's *best*-case running time is also $\Omega(n_1 + n_2)$, although we didn't need to show that here.]

(c) **[2 marks]** Show how to implement MERGEHEAPS in $\mathcal{O}(n_1 \log(n_1 + n_2))$ time in the worst case. (Do not do the running time analysis in this part.)

Solution

The running time gives a hint about how to implement this. We can think about performing n_1 different INSERT operations into a heap of size at most $n_1 + n_2$.

```
1 def MergeHeaps(heap1, heap2):
2   new_heap = copy of heap2
3   for i from 1 to heap1.size:
4     Insert(new_heap, heap1[i])
5   return new_heap
```

(d) **[2 marks]** Show that your algorithm in part (c) has worst-case running time $\mathcal{O}(n_1 \log(n_1 + n_2))$.

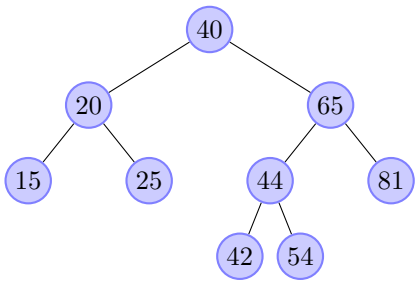
Solution

There are n_1 INSERT operations, and each one is performed on a heap of size at most $n_1 + n_2$, leading to a $\mathcal{O}(n_1 \log(n_1 + n_2))$ running time.

Note: Only after the exam, however, did we realize that we assumed the copying of `heap2` takes constant time (some sort of block-copying mechanism, but this is not correct asymptotically). Essentially, this algorithm works in the correct time if we allow `heap1` to be inserted *into* `heap2`, since in this case we do not need to copy the elements of `heap2`. So we accepted for part (c) any algorithm that ran in the (slower) time $\mathcal{O}((n_1 + n_2) \log(n_1 + n_2))$, and for part (d) any analysis that was consistent with the algorithm provided in part (c). We certainly regret this error.

3. Here are some questions about binary search trees and AVL trees.

   (a) **[2 marks]** Consider the following AVL tree:



   Suppose we insert a value chosen uniformly at random between 1 and 100 inclusive, using the AVL Tree INSERT algorithm from lecture. Calculate the expected number of **rotations** that will occur for this insertion. You do not need to simplify nor add fractions in your final answer.

   You may make the following two assumptions in your work:

   - If the chosen value is already in the AVL Tree, no rotations occur.
   - At most one node will have an imbalance fixed. That is, the possible number of rotations performed is always between 0 and 2.

Solution

One nice observation is an imbalance only occurs when inserting into the subtree rooted at 44.

- If an item is inserted under 42, 1 right rotation occurs (around 65).
- If an item is inserted under 54, 1 left rotation occurs (around 44), then 1 right rotation occurs (around 65).

So the expected number of rotations is

$$1 \cdot \frac{1+1}{100} + 2 \cdot \frac{9+10}{100}.$$

   (b) **[3 marks]** Suppose we augment binary search trees so that each node stores the size of the subtree rooted at that node (i.e., 1 plus the number of its descendants).

   Write the pseudocode for a modified BST INSERT algorithm that correctly maintains the `size` attribute after the insertion occurs. This algorithm does not need to perform rotations or keep the AVL property. Your algorithm must run in $\mathcal{O}(h)$ time, where h is the height of the BST.

   Also, briefly justify why your algorithm does *not* need to update every node. You do not need to analyse the running time of your algorithm.

Solution

The key idea is that when we insert a new node into a tree, we need to ensure that the root's size gets incremented by 1. Because we take a recursive approach, the nodes whose size attribute gets incremented will be the ancestors of the new node. Only these nodes need to be updated, since none of the other nodes have their subtrees modified, and so their sizes don't change. (The size of a node depends only on the subtree rooted at that node.)

```
1 def Insert(D, key, value):
2   if D is empty:
3     D.root.key = key
4     D.root.value = value
5     D.root.size = 1
```

```
6     else if D.root.key == key:
7       # you can do whatever you want here
8     else if D.root.key < key:
9       Insert(D.right, key, value)
10      D.root.size += 1
11    else if D.root.key > key:
12      Insert(D.left, key, value)
13      D.root.size += 1
```

(c) **[3 marks]** Now suppose we have BSTs augmented with `size` attributes for each node, as described in the previous part. Show how to use this to support the following operation in $\mathcal{O}(h)$ time, where h is the height of the BST:

- FindKthLargest(D, k): return the k-th largest key stored in D. FindKthLargest$(D, 1)$ is equivalent to finding the maximum key in D.

You may **assume** the binary search tree has no duplicate keys.

In addition to the pseudocode, briefly justify why your solution is correct. You do not need to analyse the running time of your algorithm.

Hint: think recursively. The size of the right subtree tells you whether the k-th largest key is at the root, left subtree, or right subtree.

Solution

If the right subtree has size $\geq k$, the k-th largest is in the right subtree. If the right subtree has size $k-1$, the k-th largest is the root. Otherwise, the k-th largest is in the left subtree, and we can search for it recursively, offsetting by the size of the right subtree plus 1 (for the root), since those keys are all bigger than the keys in the left subtree.

```
1 def FindKthLargest(D, k):
2   if k > D.size:
3     return null
4   else if D.right.size <= k:
5     return FindKthLargest(D.right, k)
6   else if D.right.size == k - 1:
7     return D.root.key
8   else:
9     return FindKthLargest(D.left, k - 1 - D.right.size)
```

4. Let n be a positive integer. Suppose we have an empty binary search tree, and insert n distinct numbers into it, in some order.

   If we use the naïve INSERT algorithm for BSTs, the maximum height of a BST we could get is n. Recall that there are 2^{n-1} permutations of the n distinct numbers that result in a BST of height n when the items are inserted according to this permutation. You may use this fact, without proof, in this question.

   Let C_n be the number of permutations of the n distinct numbers that result in getting a BST of height exactly $n-1$. Note that C_1 and C_2 are both 0.

   (a) **[1 mark]** Determine the value of C_3.

Solution

We can think of this as looking for orders of $\{1, 2, 3\}$ that yield a BST of height 2. There's two of them: $[2, 1, 3]$ and $[2, 3, 1]$.

   (b) **[3 marks]** Find a formula relating C_n and C_{n-1} that is valid for all $n \geq 4$.
   Hint: a permutation that results in getting a BST of height $n-1$ has exactly four possibilities for its first number.

Solution

There are two kinds of orders that are counted by C_n:

- Orders that start with the smallest/largest value.
- Orders that start with the second smallest/second largest value.

From the first group, the remaining $n-1$ values must be inserted in an order that results in a tree of height $n-2$; there are C_{n-1} such orders.

From the second group, the remaining items are divided into two groups: the single element that is smaller/greater than the starting value (when it is the second smallest or second largest, respectively), and the remaining $n-2$ items. These $n-2$ items must be inserted in an order that results in a BST of height $n-2$. There are 2^{n-3} such permutations, and the single element can be inserted into each order in $n-1$ different spots, for a total of

$$C_n = 2C_{n-1} + 2 \cdot 2^{n-3} \cdot (n-1).$$

Use this page for rough work. If you want work on this page to be marked, please indicate this clearly *at the location of the original question.*

Name:

	Q1	Q2	Q3	Q4	Total
Grade					
Out Of	6	7	8	4	25