Data Structures - 67109
Exercise 5

Due: 1/5/2019

Question 1

In order to better understand the last part of the analysis of the expected running time for the probabilistic quick-sort algorithm we consider the following question:
We would like to calculate the n'th element in the sequence defined by the recurrence relation $a_n := \sum_{i=1}^{n-1} a_i$ and the first element $a_1 = 1$. The following algorithm returns this element, for a given n:

```
Calc_a(n):
    if n == 1:
        return 1
    sum = 0
    for i = 1 to n - 1
        sum = sum + calc_a(i)
    return sum
```

1. Write down a recurrence relation for the above pseudocode's running time per input (instead of per input's length). Use constants rather than asymptotic notation. (Note that there is no "worst-case" or "average-case" running time as the running time is calculated for a specific input).
2. Find a tight asymptotic bound for the relation from the previous subquestion. (Hint: look both at $T(n)$ and at $T(n+1)$ at the same time).

Question 2

Recall that a p-coin is a coin that has p probability to fall on T. Define X as a random variable that given n flip results, gives the number of Ts that we got ($X(x_1 x_2 \ldots x_n) = |\{i : x_i = T\}|$). In exercise 3 we saw that $\mathbb{E}[X] = np$.
Note: Markov's inequality may be expressed as $P(X \geq a) \leq \frac{\mathbb{E}[X]}{a}$.

1. Using Markov's inequality, what is the upper bound for the probability of the event $A = (X \geq \frac{n}{2})$ (i.e. $\{x_1x_2\dots x_n : X(x_1\dots x_n) \geq \frac{n}{2}\}$)? For what values of p does this bound give us some information about this probability?

2. What is the probability of event A from the previous subquestion?

3. For every $n \in \mathbb{N} \setminus \{0\}$ we define a random variable X_n that gets the values $\frac{1}{k}$ for $1 \leq k \leq n$ in a uniform distribution (i.e. there is an implicit probability space over n elements with uniform distribution and X_n maps them to these values). Using Markov's inequality prove that for any $\epsilon > 0$, the following holds: $\lim_{n\to\infty} P(X_n \geq \epsilon) = 0$.
   Guidance: Use Markov's inequality to show that it suffices to prove that $\lim_{n\to\infty} \mathbb{E}[X_n] = 0$. Then, in order to prove this use the definition of a limit and the fact that the expectation for a uniform distribution is just the average.

Question 3

Recall radix sort from the recitation.

1. Use induction to prove that Radix Sort works. Where does your proof need the assumption that the per-digit sort is stable?

Recall bucket sort from the recitation. Analyze the average running time for bucket sort, use n_i a random variable for the number of elements in the i bucket and write it as $n_i = \sum_{j=1}^{n} X_{i,j}$, where $X_{i,j} = \begin{cases} 1 & arr[j] \text{ is in } buckets\_arr[i] \\ 0 & otherwise \end{cases}$.

Hint: We assume that the input is uniformly distributed, this means that $P(X_{i,j} = 1) = \frac{1}{k}$ (the probability of element j in the array to be assigned to bucket i is $\frac{1}{k}$). You may assume that the running time of the inner sort we use for each bucket is $\Theta(n^2)$.

2. Explain why the expression for the running time for sorting the buckets is $\sum_{i=1}^{k} \mathbb{E}[n_i^2]$.

3. Resolve $\sum_{i=1}^{k} \mathbb{E}[n_i^2]$ and give an upper asymptotic bound for the total running time of the function.
   Hint: find $\mathbb{E}[n_i^2]$, decompose n_i^2 to indicator variables, and then use $\mathbb{E}[X^2] = \sum_{x\in image(X)} x^2 P(X = x)$.

Question 4

In this question we will prove that a binary tree of height h has at most 2^h leaves.

Definition: A COMPLETE BINARY TREE is a binary tree in which all the leaves are in the same depth (which is the height of the tree) and all internal nodes have 2 children.

Define A_h as the set of all binary trees of height h.

For every tree $a \in A_h$ denote the number of leaves in a by $L(a)$.

Note that there is exactly one tree $a_h \in A_h$which is a complete binary tree (make sure you understand why).

1. Show that for any tree $a \in A_h$ that is not a_h ($a \neq a_h$), there is another tree $b \in A_h$ such that $L(a) < L(b)$.

2. Conclude that a_h is the tree with the most leaves amongst A_h ($a_h = argmax_{a \in A} L(a)$).
   Hint: You can use the fact that there is only a finite number of binary trees of heigh h ($|A_h| < \infty$).
   Hint: Assume for the sake of contradiction that there is a tree $a \in A_h$ with $L(a_h) < L(a)$.

3. Show by induction that $L(a_h) = 2^h$.
   Hint: Note that a_h is just 2 copies of a_{h-1} attached as sub-trees to a root. Meaning that the copies' roots are the children of a_h's root. Use this fact to write down a recurrence formula in h for $L(a_h)$, solve it (you can guess the solution by the iterative method) and prove the solution by induction.

Question 5

Let $Merge_k$ be the following problem: Given k sorted arrays, $A_1, ..., A_k$, each of size $\frac{n}{k}$, return a single sorted array A containing all n elements from $A_1, ..., A_k$. Consider this algorithm for $Merge_k$:

```
Merge_k(A_1,...,A_k):
    B_1 = A_1
    for i=2 to k:
        B_i= Merge(B_{i-1}, A_i)
    return B_n
```

1. Give a tight asymptotic bound for the Merge_k algorithm, assuming the running time of Merge(A,B) is $\theta(|A| + |B|)$.

2. Give a comparison-based algorithm that solves $Merge_k$ in $O(nlogk)$. No need to formally prove correctness or running time, just give a simple explanation. You may assume k is a power of 2

3. Prove: $\Omega(nlogk)$ is a lower-bound for the number of queries performed by a comparison-based algorithm that solves $Merge_k$.
   Guidance: Think about how to construct an input for $Merge_k$ that has enough permutations, i.e. enough leaves in the algorithms decision tree, that will bound the height of the tree by $\Omega(nlogk)$ (Notice that each array A_i is sorted, so a simple permutation of it is not a legal input to the algorithm). You may also want to use the claim from question 4, that a binary tree of height h has at most 2^h leaves.

Question 6

In class you will see a new data structure called BST (binary search tree), and the insert and successor operations on it. In the following questions assume all elements are distinct.

1. Consider the following array, $[11, 9, 16, 12, 5, 6, 2]$, generate its respective BST (meaning perform the insert operation given below, element by element, from left to right to generate the BST). Sketch the BST that was formed.

2. Run the given Inorder function (pseudocode below) on the tree from the previous subquestion and write down the output. What does Inorder do?

3. In this subquestion we find an asymptotic upper bound for the given Inorder algorithm:
   (a) Prove that in the run of Inorder on a tree T, after the run leaves a sub-tree of T it will never come back to it during the run.
   (b) Conclude that in the run of the given Inorder function on a tree $T = < V, E >$ (V nodes and E edges), every edge $e \in E$ is visited at most twice.
   (c) Conclude an upper asymptotic bound for the given Inorder function (the tightest you can find).

4. Explain how can the upper asymptotic bound for this algorithm be less than $n \log(n)$ when we showed that any general sorting algorithm that is comparisson-based has to have a lower bound of $n \log(n)$ for its running time? (Note that we don't have any assumptions for the inputs, as we had for the linear sorting algorithms we saw).

```
Insert(tree, value):

    parent = null
    node = tree.root
    while node is not null
```

```
        parent = node
        if value < node.val
            node = x.left
        else
            node = node.right
    new_node = create a new node
    new_node.val = value
    new_node.parent = parent
    if parent == null
        tree.root = new_node
    else if value < parent.value
        parent.left = new_node
    else
        parent.right = new_node
```

```
Inorder(tree):
    node = Min(tree.root)
    while node is not null
        print node.value
        node = Successor(node)
```

```
Min(tree):
    if tree.root is null
        return null
    node = tree.root
    while node.left is not null
        node = node.left
    return node
```

```
Successor(node):
    if node.right is not null
        return Min(node.right)
    parent = node.parent
    while (parent is not null) and (node == parent.right)
        node = parent
        parent = node.parent
    return parent
```