Homework #2

(Due: Oct 25)

Task 1. [**10 Points**] **There's a calm under the waves... So I choose to sink... (Maria Mena)**

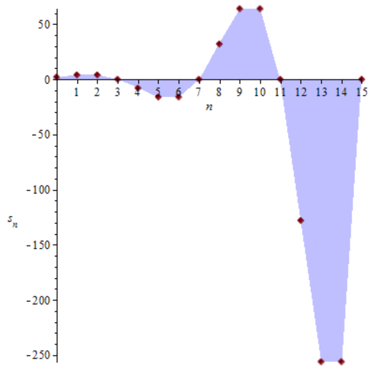

Figure 1: Plot of $s_n = 2^{\frac{n+3}{2}} \cos \frac{(n-1)\pi}{4}$.

Consider the following recurrence defined for nonnegative integral values of n.

$$s_n = \begin{cases} 2 & \text{if } n = 0, \\ 4 & \text{if } n = 1, \\ 2\left(s_{n-1} - s_{n-2}\right) & \text{otherwise.} \end{cases}$$

Show that s_n represents the wave shown in Figure 1, i.e, $s_n = 2^{\frac{n+3}{2}} \cos \frac{(n-1)\pi}{4}$.

Task 2. [**20 Points**] **When you try your best, but don't succeed... (ColdPlay)**

When the Master Theorem fails, try the Akra-Bazzi method to solve the following recurrences.

(*a*) [**10 Points**] The following recurrance arises during the worst-case complexity analysis of a deterministic selection algorithm (given in Section 9.3 of the textbook[1]).

$$T(n) \le \begin{cases} \mathcal{O}(1) & \text{if } n < 140, \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 6\right) + \mathcal{O}(n) & \text{if } n \ge 140. \end{cases}$$

Dropping the ceiling for simplicity and observing that $\frac{7n}{10} + 6 \le \frac{8n}{10} = \frac{4n}{5}$ holds when $n \ge 60$, we can obtain an upper bound $T_1(n)$ on $T(n)$ using the following recurrence.

$$T_1(n) = \begin{cases} \Theta(1) & \text{if } n < 140, \\ T_1\left(\frac{n}{5}\right) + T_1\left(\frac{4n}{5}\right) + \Theta(n) & \text{if } n \ge 140. \end{cases}$$

[1]Chapter 9 (Medians and Order Statistics), Introduction to Algorithms (3rd Edition) by Cormen et al.

Solve the recurrence for $T_1(n)$.

Also observe that $\frac{7n}{10} + 6 \leq \frac{7.5n}{10} = \frac{3n}{4}$ holds when $n \geq 120$, and so we obtain the following upper bound on $T(n)$.

$$T_2(n) = \begin{cases} \Theta(1) & \text{if } n < 140, \\ T_2\left(\frac{n}{5}\right) + T_2\left(\frac{3n}{4}\right) + \Theta(n) & \text{if } n \geq 140. \end{cases}$$

Solve for $T_2(n)$.

(*b*) [**5 Points**] A random binary search tree is formed by inserting the keys into the tree one at a time according to a random permutation. It turns out that in at least half the random binary search trees formed in this way both subtrees of the root have between $\frac{n}{4}$ and $\frac{3n}{4}$ keys, where n is the total number of keys in the tree. Then the expected search time in such a tree can be upper bounded using the following recurrence relation:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ \frac{1}{2}T(\frac{3n}{4}) + \frac{1}{2}T(n) + \Theta(1) & \text{otherwise.} \end{cases}$$

Solve for $T(n)$.

(*c*) [**5 Points**] The following recurrence gives the time $Q(n)$ needed to answer a line query in a *ham-sandwich tree*[2] for n points in a plane (excluding the time used to report the points).

$$Q(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ Q(\frac{n}{2}) + Q(\frac{n}{4}) + \Theta(\log n) & \text{otherwise.} \end{cases}$$

Solve for $Q(n)$.

Task 3. [20 Points] Averaging: getting as close to the stars as one is to the earth

We will analyze the average performance of the Randomized-Quicksort algorithm given in Section 7.3 (page 179) of the textbook[3]. Let $n = r - p + 1$ and $k = q - p + 1$. We also assume that the Randomized-Partition algorithm (also on page 179) is *stable* in the sense that if two keys x and y end up in the same partition and x appears before y in the input, then x must also appear before y in the resulting partition.

We will average the running times of Randomized-Quicksort on all possible arrangements of the keys in the original input array. If $T(n)$ is the average running time of the algorithm on n keys, we can describe $T(n)$ using the following recurrence.

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ n - 1 + \frac{1}{n}\sum_{k=1}^{n}\{T(k-1) + T(n-k)\} & \text{otherwise.} \end{cases}$$

Use generating functions to show that $T(n) = \mathcal{O}(n \log n)$ without removing the full history from the recurrence.

[2]This is a serious data structure in computational geometry, invented by H. Edelsbrunner and E. Welzl in 1983!

[3]Chapter 7 (Quicksort), Introduction to Algorithms (3rd Edition) by Cormen et al.

Task 4. [30 Points] (Encoded title) A barbarian cheers decently![4]

Please refresh your knowledge on *binary search trees* before you proceed. You may want to review Sections 12.1, 12.2 and 12.3 of the textbook[5]. Recall that such trees support SEARCH, INSERT and DELETE operations in $\mathcal{O}(h)$ time each, where h is the height of the tree. But h can be as large as $n-1$ in the worst-case, where n is the number of keys in the tree. We will explore one way of improving these bounds.

Any node x in a binary search tree $\mathcal{T}$ is called *α-balanced* provided neither subtree of x contains more than $\alpha \cdot (x.n)$ keys, where $\alpha \in \left[\frac{1}{2}, 1\right)$ is a constant and $x.n$ is the number of keys stored in the subtree rooted at x. The tree $\mathcal{T}$ is *α-balanced* as a whole if every node in $\mathcal{T}$ is α-balanced.

(*a*) [**5 Points**] The algorithm in Figure 2 makes the subtree rooted at any arbitrary node x of $\mathcal{T}$ $\frac{1}{2}$-balanced. Show that it runs in time $\Theta(x.n)$.

(*b*) [**5 Points**] Show that an α-balanced tree containing n keys supports search operations in $\mathcal{O}(\log n)$ worst-case time.

(*c*) [**5 Points**] Consider an α-balanced tree with $\alpha > \frac{1}{2}$. INSERT and DELETE operations on such a tree are implemented in exactly the same way as in an ordinary binary search tree (see Section 12.3 of the textbook), except that after every such operation, if any node in the tree is no longer α-balanced, we make the highest such node $\frac{1}{2}$-balanced by calling the REBUILD-TREE routine given in Figure 2.

In order to analyze the amortized performance of an α-balanced tree $\mathcal{T}$, we first define an *imbalance function* $\Delta(\ x\ )$ for each node x in $\mathcal{T}$ as follows:

$$\Delta(\ x\ ) = |\ x.left.n - x.right.n\ |.$$

We then define the potential of $\mathcal{T}$ as

$$\Phi(\ \mathcal{T}\ ) = c \sum_{x \in \mathcal{T} : \Delta(\ x\ ) > 1} \Delta(\ x\ ),$$

where c is a sufficiently large constant that depends only on α.

Argue that Φ is always nonnegative, and is 0 for $\frac{1}{2}$-balanced trees.

(*d*) [**5 Points**] You proved in part (*a*) that the actual cost of rebuidling the subtree rooted at x is $\Theta(n.x)$. Suppose $n.x$ units of potential can pay for that cost when using the potential function defined in part (*c*). Then show that REBUILD-TREE will run in $\mathcal{O}(1)$ amortized time provided $c \geq \frac{1}{2\alpha - 1}$.

(*e*) [**10 Points**] Show that an INSERT operation on an α-balanced tree containing n keys costs $\mathcal{O}(\log n)$ amortized time. Prove the same for DELETE.

[4]If you rearrange the letters of "A barbarian cheers decently" you get "Balanced binary search tree"!

[5]Chapter 12 (Binary Search Trees), Introduction to Algorithms (3rd Edition) by Cormen et al.

```
REBUILD-TREE( $x$ )
(Input is a node $x$ in an arbitrary binary search tree. This function makes the subtree rooted at $x$ $\frac{1}{2}$-balanced.)
 1. $A \leftarrow$ CREATE-ARRAY( $x.n$ )                       {create a temporary array $A[1 \ldots x.n]$}
 2. $k \leftarrow 0$                                            {global variable used by GET-IN-SORTED-ORDER}
 3. GET-IN-SORTED-ORDER( $x$, $A$ )  {put the keys in the subtree rooted at $x$ in nondecreasing order of value in $A$}
 4. FREE-TREE( $x$ )                                          {free the memory occupied by the subtree rooted at $x$}
 5. $x \leftarrow$ BUILD-HALF-BALANCED-TREE( 1, $x.n$, $A$ )   {build a $\frac{1}{2}$-balanced search tree using the keys in $A$}
 6. FREE-ARRAY( $A$ )                                          {free the memory occupied by $A$}
```

```
GET-IN-SORTED-ORDER( $x$, $A$ )
(Traverses the subtree rooted at $x$ and puts the keys stored in that subtree in sorted order in array $A$. Uses a global index $k$ that keeps track of where to put the next key value in $A$.)
 1. if $x \neq nil$ then                                       {if the subtree is not empty}
 2.     GET-IN-SORTED-ORDER( $x.left$, $A$ )                   {get the keys in the left subtree in sorted order}
 3.     $k \leftarrow k + 1$                                   {the next empty location in $A$}
 4.     $A[k] \leftarrow x.key$                                {put the root key right after the keys in the left subtree}
 5.     GET-IN-SORTED-ORDER( $x.right$, $A$ )  {put the keys in the right subtree in sorted order after the root key}
```

```
BUILD-HALF-BALANCED-TREE( $l$, $r$, $A$ )
(Builds a $\frac{1}{2}$-balanced search tree using the sorted key values stored in $A[l \ldots r]$. It puts the median of $A[l \ldots r]$ at the root, and then recursively builds the left and right subtrees using the values stored to the left and right of median in $A$, respectively. Returns a pointer to the root.)
 1. if $l > r$ then                                            {no keys in $A$}
 2.     return nil                                             {return an nil pointer}
 3. else                                                       {$A$ has one or more keys}
 4.     $x \leftarrow$ CREATE-NODE( )                          {allocate a new tree node}
 5.     $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$           {get the midpoint of $A[l \ldots r]$}
 6.     $x.key \leftarrow A[m]$                                {key of $x$ is set to the median of $A[l \ldots r]$}
 7.     $x.n \leftarrow r - l + 1$                             {set the size of the subtree rooted at $x$}
 8.     $x.left \leftarrow$ BUILD-HALF-BALANCED-TREE( $l$, $m-1$, $A$ )  {put the keys in $A[l \ldots m-1]$ in the left subtree}
 9.     $x.right \leftarrow$ BUILD-HALF-BALANCED-TREE( $m$+1, $r$, $A$ ) {put the keys in $A[m+1 \ldots r]$ in the right subtree}
10.     return $x$
```

Figure 2: REBUILD-TREE(x) makes the subtree rooted at any node x of a binary search tree $\frac{1}{2}$-balanced.