Date: Feb 16, 2006, Thursday
Time: 10:40am – 12:40pm
Professor: Alper Üngör (Office CSE 430)

This is a closed book exam. No collaborations are allowed. Your solutions should be concise, but complete, and handwritten clearly. Use only the space provided in this booklet, including the even numbered pages. Write your initials on each sheet. You should answer all the questions to get full credit.

GOOD LUCK!

Your name: ________________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Credit</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
1. [20 = 10 + 10 points] **Binary Search Trees**

Recall that inserting a new element into a binary search tree involves a top-down search and an append operation at the leaf level. Suppose we construct a binary search tree by successively inserting \( n \) distinct items into an initially empty tree, without ever rebalancing the tree.

(a) How many different trees can you get for \( n = 3 \) items? Draw the trees.

(b) Is it true that if you pick a random sequence of the \( n \) items then each of the possible trees is equally likely? Justify your answer.

(a) For \( 3! = 6 \) different sequences, we have 5 different trees.

\[
\begin{align*}
\text{1, 2, 3} & \quad 1, 3, 2 & \quad 2, 1, 3 \quad \text{and} \quad 2, 3, 1 \\
\end{align*}
\]

(b) No. Consider \( n = 3 \) in part a. The third tree is more likely to be constructed than the others.
2. [20=5+5+10 points] TREE-DEPTH AND RECURRENCE

A binary tree is **full** if each node has exactly zero or two children. Consider a full binary tree $T$ with $n$ leaves. Define the **right-depth** of a node $v$ as the number of right edges from root to $v$. Let $R_T$ denote the sum of right depths over all leaves of $T$.

(a) Which tree with $n$ leaves minimizes $R_T$? What is $R_T$ for this tree?
(b) Which tree with $n$ leaves maximizes $R_T$? What is $R_T$ for this tree?
(c) Prove that if every internal node has at least as many leaves in the left as in the right subtree then $R_T \leq n \log_2 n$.

\[
R_T = n - 1
\]

\[
R_T = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}
\]

\[
R_T(n) \leq 2 R_T(n/2) + \frac{n}{2}
\]

At most $\frac{n}{2}$ leaves in the subtree of $T$ uses the right edge of the root.

\[
R_T(n) \leq 2 \frac{n \log n}{2} + \frac{n}{2}
\]

\[
\leq n(\log n - \log 2) + \frac{n}{2}
\]

\[
\leq n \log n - n + \frac{n}{2}
\]

\[
\leq n \log n
\]
3. [20 = 10+10 points] DYNAMIC PROGRAMMING

Given a sequence $S$ of $n$ integers (not necessarily positive), MAXIMUM SUM CONSECUTIVE SUBSEQUENCE PROBLEM asks to find a consecutive subsequence of $S$ whose summation is maximized. For example, for $S = \langle -6, 12, -7, 0, 14, -7, -3 \rangle$, the maximum sum of 19 is achieved for the subsequence $\langle 12, -7, 0, 14 \rangle$.

Note that it is straight-forward to design a $O(n^3)$-time algorithm for this problem by computing the sum for all possible consecutive subsequences. However, your goal is to design a dynamic programming algorithm with better running time, i.e., that runs in $o(n^3)$ time.

(a) Write and describe a recurrence to structure a dynamic programming algorithm for solving the MAXIMUM SUM CONSECUTIVE SUBSEQUENCE PROBLEM.

(b) Describe and analyze your algorithm based on the recurrence you constructed in (a).

[Note/Hint: While there is a solution with $\Theta(n)$ running time, $\Theta(n^2)$ time solution is also worth full credit.]

(a) Let $M[i]$ denote the maximum sum consecutive subsequence that ends at location $i$.

The observe the optimal substructure given by the recurrence

$$
M[i] = \begin{cases} 
0 & \text{if } i = 0 \\
\max\{M[i-1] + S[i], S[i]\} & \text{if } i > 1 
\end{cases}
$$

Note that we either append a new number to the substructure solution or start a new subsequence. We cannot include $M[i-1]$ in the max operator be it would result in non-consecutive subsequences.

(b) A simple linear time algorithm follows from (a):

Scan $S$ while computing all $M[i]$ using the values, maintaining the prev value of $M$ and max value of $M$.

Running time $O(n)$.
Extra space $O(1)$. 

7
4. [20 = 10+10 points] Greedy Algorithms

(a) You are asked to tile an $m \times n$ room, using any square tiles, minimizing the number of tiles used ($m \geq n$ are integers). Consider a greedy strategy which uses the largest fitting square tile first. Does this algorithm give an optimal solution? Justify your answer.

No. Consider $m=7$, $n=6$.
For this case we give a solution better than the output of greedy.

\begin{align*}
\text{7 tiles} & \quad \text{5 tiles} \\
6x6 & \quad 3x3 \quad 4x4 \\
1x1 & \quad 3x3 \quad 2x2 \quad 2x2
\end{align*}

(b) Consider the following version of the 0/1 Knapsack Problem. Given a set of $n$ objects with weights $w_1 \geq w_2 \geq \ldots \geq w_n$, and profits $p_1 \leq p_2 \leq \ldots \leq p_n$, find a subset of objects such that the total weight is bounded by a given capacity $W$ and the total profit is maximized. Describe the best greedy algorithm you can for solving this problem. Does your algorithm result in an optimal solution? Justify your answer.

Greedy Algorithm: Start with the $n$th item and iteratively put the remaining highest index item in the knapsack as long as there is capacity.

Yes, this algorithm results in an optimal solution.

Proof Sketch: Given any other solution we can transform it into our solution without decreasing the total profit as follows: Consider the highest index item that is not include in the optimal solution. Replacing another item with this item iteratively gives the transformation.
5. [20 points] **Amortized Analysis**

Recall that a **Queue** is a first-in-first-out data structure with two basic operations *enqueue* and *dequeue*, and a **Stack** is a first-in-last-out data structure with operations *push* and *pop*. Describe how to implement a queue with two stacks so that the amortized cost of each *enqueue* and *dequeue* operation is constant.

### ENQUEUE($\alpha, x$)

- Push(STACK-1, $x$)

### DEQUEUE($\alpha$)

- If STACK-2.empty() then
  - While !STACK-1.empty() do
    - $t \leftarrow$ POP(STACK-1)
    - Push(STACK-2, $t$)
  - $x \leftarrow$ POP(STACK-2)
- Return $x$

Note that a worst-case analysis would highlight $O(n)$ time of a single DEQUEUE operation.

Instead we can use taxation method and charge

- $\$4$ when we enqueue an item, $\$1$ covering the push into STACK-1,
- $\$2$ covering the transfer induced by a later DEQUEUE operation, and $\$1$ covering the cost of POP from STACK-2.