15-750 Graduate Algorithms, Spring 2017
Homework 1 (110 pts) Due: Wednesday Feb 15
Gary Miller, David Witmer TAs: Amir, Jason

Instructions. Collaboration is permitted in groups of size at most three. You must write the names of your collaborators on your solutions and must write your solutions alone.

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td></td>
<td>110</td>
</tr>
</tbody>
</table>

(30) 1. **Winograd’s Variant of Strassen’s Algorithm**

In class, we presented the classic version of Strassen’s Algorithm for matrix multiplication. The following variant due to Winograd also uses 7 multiplications but only uses 15 additions (rather than 18) in order to multiply $2 \times 2$ matrices.

$$
\begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}
= 
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}
$$

The 22 computations in Winograd’s variant are as follows.

$$
\begin{align*}
C_{11} &= U_1 = P_1 + P_2 & P_1 &= A_{11}B_{11} & S_1 &= A_{21} + A_{22} & T_1 &= B_{12} - B_{11} \\
U_2 &= P_1 + P_4 & P_2 &= A_{12}B_{21} & S_2 &= S_1 + A_{11} & T_2 &= B_{22} - T_1 \\
U_3 &= U_2 + P_5 & P_3 &= S_1T_1 & S_3 &= A_{11} - A_{21} & T_3 &= B_{22} - B_{11} \\
C_{21} &= U_4 = U_3 + P_7 & P_4 &= S_2T_2 & S_4 &= A_{12} - S_2 & T_4 &= B_{21} - T_2 \\
C_{22} &= U_5 = U_3 + P_3 & P_5 &= S_3T_3 & & & \\
U_6 &= U_2 + P_3 & P_6 &= S_4B_{22} & & & \\
C_{12} &= U_7 = U_6 + P_6 & P_7 &= A_{22}T_4 & & & 
\end{align*}
$$

a) Compute the smallest $n = 2^k$ such that this new method does fewer arithmetic operations (additions and multiplies) than the naive matrix multiplications on matrices of size $2^k$ by $2^k$. 

b) A naive implementation of the computation described in the previous question would simply compute all the $S$'s and the $T$'s in order followed by the $P$'s and then the $U$'s. Keeping all those intermediate results would require as many as 22 submatrices at each level of the recursion. This assumes that the $A$’s and $B$’s are provided as parameters from the calling function. In this problem, you will lay out the code and the memory for a single recursive call so that not too much memory is used.

Write the 22 computations of Winograd’s algorithm in the order that they will be executed in your implementation. You cannot repeat any computations and the data dependencies must all be satisfied. That is, for example, you cannot compute $P_6$ before computing $S_4$.

c) Choose a number $k$ of submatrices that your implementation will use for scratch space. Next to each line of code, write down which variables ($P, U, S, T$’s) are stored in your scratch space immediately after that computation. For example, two lines might read

\[
P_1 = A_{11}B_{11} \quad (P_1) \quad (1)
\]
\[
P_2 = A_{12}B_{21} \quad (P_1, P_2) \quad (2)
\]

Every computation must have all of its operands stored in the scratch space before it can be executed. The goal is to minimize $k$. You should be able to get $k = 8$. Can you get $k = 7$? Remember that you don’t have to store the $A$’s and $B$’s in the scratch space.

d) For the $k$ you computed in part b, compute the maximum amount of space used at any time by the algorithm for matrices of size $n \times n$ where $n$ is a power of 2.

(20) 2. Fibonacci Numbers

Suppose instead of using powers of two, we now represent integers as the sum of Fibonacci numbers. That is, rather than representing a number as an array of bits, we keep an array of “fibbits” so that $(x_k x_{k-1} \ldots x_1)_F$ denotes the number $\sum_{i=1}^{k} x_i F_i$. As an example, the Fibonacci number $(1101)_F = F_4 + F_3 + F_1 = 1 + 2 + 3 = 6$. Recall that the Fibonacci numbers satisfy the recurrence $F_0 = 0, F_1 = 1$ and $F_{n+2} = F_{n+1} + F_n$.

a) Show that every positive integer $n$ can be represented as a Fibonacci number.

b) Give an algorithm to increment a Fibonacci number in constant amortized time.

(20) 3. Maintaining a list with reversals

Consider a data structure that represents an ordered list of elements under the following three types of operations:

- **access**($k$): Return the $k$th element of the list (in its current order).
- **insert**($k, x$): Insert $x$ (a new element) after the $k$th element in the current version of the list.
• reverse\((i, j)\): Reverse the order of the \(i\)th through \(j\)th elements.

For example, if the initial list is \([a, b, c, d, e]\), then access(2) returns \(b\). After reverse(2,4), the represented list becomes \([a, d, c, b, e]\), and then access(2) returns \(d\).

Show how to modify splay tree construction so that each operation runs in \(O(\log n)\) amortized time, where \(n\) is the (current) number of elements in the list. The list starts out empty.

Hint: First consider how to implement access and insert using splay trees. Then think about a special case of reverse in which the \([i, j]\) range is represented by a whole subtree. Use these ideas to solve the real problem. Remember, if you store extra information in the tree, you must state how this information can be maintained under various restructuring operations.

(40) 4. Binomial heaps

In this problem, we are going to discuss an alternative representation of binomial trees and binomial heaps with missing children, Fibonacci Heaps. We will then design a priority queue based on that new representation.

a) Let \(B_k\) be a binomial tree of rank \(k\). As mentioned in the class, nodes of \(B_k\) store a set of pointers that can be essentially pictured as follows:

```
       *  
      /\  
     /  \  
    *  *  *  
   / \ / \ /  
  *  *  *  *  *  
```

show that if we remove all pointers pointing to parents, the resulting graph is a single node attached to a perfect binary tree. This structure is called a half-tree.

b) Justify that by keeping this set of pointers in binomial heaps, all heap operations other than decrease-key can be performed in the exact same way as before.

c) Show that this half-tree is half-ordered, i.e., for every node \(v\), the key of the left child is guaranteed to be larger than the key of \(v\). (In fact, this representation of binomial trees provides the additional binary property at the cost of losing the full order).

d) From now on, we look at this half-tree as a rooted ordered binary tree with the children ordered such that the child discussed in part (c) be the left child. The general idea to design a priority queue is to implement decrease_key(x) by simply cutting \(x\) and its left subtree as follows:
However, careless utilization of this **decrease_key** method results into slow **delete_min** and **decrease_key** operations. To obtain a reasonably fast priority queue, we assign a non-negative integer $r[v]$—referred to as rank—to any node $v$. We enforce $r$ to satisfy the following properties the entire time:

- If node $v$ has two children: Let $R(v)$ and $L(v)$ respectively denote $v$’s right and left children. Then \{r[v] - r[R(v)], r[v] - r[L(v)]\} has to be either \{1, 1\}, \{1, 2\}, or \{0, i\} for some $i > 1$. We refer to this set as the rank difference set of $v$.
- For non-root nodes with less than two children, the rank of missing children is considered to be $-1$ and the above-mentioned rule has to hold.
- If node $v$ is the root of one of the trees in the collection: The rank of $v$ has to be exactly one unit larger than its child.
- If node $v$ is a single node: $r[v] = 0$

show that if there are $h$ trees in the collection after a **delete_min**, the **delete_min** operation can be done in $O(h)$; having the above-mentioned key rules satisfied.

e) Show that every node of rank $k$ has at least $F_{k+2}$ descendants including itself.

f) (10 extra points) Propose a procedure to fix rank rules by modifying rank values after a **decrease_key**. (Hint: Recursively float up to the ancestors from the point of probable violation)

g) (30 extra points) Show that **decrease_key** and **delete_min** procedures will perform in $O(1)$ and $O(\log n)$ time respectively; using the following potential function: We define potentials for each vertex and the potential of the heap is defined as the summation of its nodes’ potentials. The potential of a non-root node with rank difference set \{i, j\} is $i + j - 1$ unless $i = j = 1$. In \{1, 1\} case, the potential is defined to be zero. Also, the potential of the root is defined as one. (Hint: the recursive procedure in part (f) does not encounter more than two nodes with rank difference set \{1, 1\}. [10 points will be awarded to the proof of the hint])