1 Q1 (25 pts) O.S. : Universal Hash Families


• Prove or provide a counter example:

1. [6 pts] Given $H$, a U.H.F of functions $h : U \rightarrow K$, we identify $K$ with the set 
   \{0, 1, 2, \ldots, k - 1\}. For any $h \in H$ we define $h + 3$ as the function that maps $x \in U$ to 
   $h(x) + 3 \mod k$. We denote $H + 3$ as the set of functions \{h + 3 ; h \in H\}. 
   Then $H + 3$ is a U.H.F.

2. [6 pts] Given $H$, a U.H.F of functions $h : U \rightarrow K$, we identify $K$ with the set 
   \{0, 1, 2, \ldots, k - 1\}. For any $h \in H$ we define $3 \cdot h$ as the function that maps $x \in U$ to 
   $3h(x) \mod k$. We denote $3H$ as the set of functions \{3 \cdot h ; h \in H\}. 
   Then $3 \cdot H$ is a U.H.F.

3. [6 pts] Given $H$, a U.H.F of functions $h : U \rightarrow K$, and for any $f : U \rightarrow K$, the set 
   $H \cup \{f\}$ is a U.H.F.

4. [6 pts] Given $H$, a U.H.F of functions $h : U \rightarrow K$, and for any $f : U \rightarrow K$ which 
   equi-partitions $U$ (That is, for any $i$ and $j$ it holds that the sizes $|f^{-1}(i)|$ and $|f^{-1}(j)|$ 
   are identical up to a factor of $\pm 1$). Then the set $H \cup \{f\}$ is a U.H.F.

2 Q2 (25 pts) T.S. : Merging BSTs

Binary search trees allow low-cost (up to logarithmic) inserts and lookups in maintaining a sorted 
collection of elements. However these costs are bounded by the height of the tree, and in imbalanced 
trees can become quite high (up to linear).

We attempt to create a self-balancing tree in the following way: we insert elements in a tree 
as long as the insertions don’t make it imbalanced. If an insertion will make the tree imbalanced, 
we simply create a new tree and continue our insertions on the new tree. Thus when we run a 
lookup, we must check a forest of trees. In order to keep the cost of lookup low, we merge these
trees frequently. In this problem we analyze the cost of merging binary trees to assess whether our proposition for self-balancing trees is practical.

You will prove upper and lower bounds on merging two BSTs \( T_1 \) and \( T_2 \) into a BST \( T \) of minimal height. You may assume all values in the BSTs are unique.

1. Give an \textbf{exact lower bound} for the number of comparisons required to construct a BST of \( 2n \) nodes.

2. Give an \textbf{asymptotic lower bound} for the number of comparisons required to merge \( T_1 \) and \( T_2 \) into a BST of \( 2n \) nodes. You may assume the cost of each insert into \( T_1 \) and \( T_2 \) was bounded from above by \( \log n \). [Hint: use part (a)]

3. Give an \textbf{upper bound} on merging \( T_1 \) and \( T_2 \) into some BST of \( 2n \) nodes. Can you asymptotically match the lower bound you have given?

4. Give a \textbf{lower bound} \( h_L \) on the height of \( T \).

5. Give an \textbf{upper bound} on the number of comparisons required to merge \( T_1 \) and \( T_2 \) into some \( T \) of height \( O(h_L) \). You may use arrays or linked lists to temporarily store values if necessary. Can you asymptotically match the lower bound you have given?

3 Q3 (20 pts) S.T.: Dynamic Programming

You are placed at the lower left hand corner, \((1, 1)\), of an \( n \times n \) grid of non-negative integers. Your goal is to reach the upper right hand corner of the grid, \((n, n)\). You are allowed to move one position at a time either up or to the right, i.e. you are only allowed to take Manhattan paths. We now define the cost of a path from \((1, 1)\) to \((n, n)\) as the sum of the values in each grid position visited along the path. Your goal is to find a path of maximal cost; we call such a path an optimal path. For convenience let \( \omega(i, j) \) denote the \((i, j)\)th entry of the grid.

1. Give a dynamic programming algorithm for finding the cost of an optimal path from \((1, 1)\) to \((n, n)\).

2. Give a dynamic programming algorithm for finding an optimal path from \((1, 1)\) to \((n, n)\).
   (\textit{Hint:} you can reconstruct an optimal path from the data used to compute its cost.)

\[
\begin{array}{cccc}
4 & 9 & 9 & 9 & 1 \\
3 & 0 & 0 & 0 & 1 \\
2 & 0 & 0 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Figure 1: An \( n = 4 \) instance of this problem.

\textbf{Note.} A recursive algorithm that uses memoization is \textit{not} a dynamic programming. A dynamic programming algorithm is bottom up by definition, so please give a bottom up algorithm.
4 Q4 (25 pts) A.V.G. : Upper and Lower Bounds

You have \( n \) machines, some are faulty and some are working properly. You know that more than half of your machines are working properly. Your objective is to find one machine that is working properly. Each machine can test each other machine.

If \( A \) tests \( B \) then \( A \) will output whether it thinks \( B \) is working properly or not. If \( A \) is working properly, then it will always output correctly whether \( B \) is working properly or not. However, if \( A \) is not working properly, then it may output either answer (no assumptions!).

Our goal is to find a single working machine using as few tests as possible.

- [1 pt] In general, what do you need to do in order to prove an upper bound? (we are looking for a general technique, not specifically for this problem)
- [1 pt] In general, what do you need to do in order to prove a lower bound? (we are looking for a general technique, not specifically for this problem)
- Find upper [11 pts] and lower [12 pts] bounds on the number of tests required to find a working machine; make your bounds as tight as you can (So you should not use an asymptotic analysis!).
  [Hint: try placing the machines in a row]

5 Q5 (5 pts) D.S.

Come up with three different modifications of problem 3. Points are proportional to originality. Extra credit if you come up with applications for the modified problems / solve them.