CS4311 Design and Analysis of Algorithms

Homework 3 (Suggested Solution)

1. For $k > 3$, we can walk k steps in exactly one of the following ways:

   - First reaching the $(n-1)$th step, and walk one step in the last move;
   - First reaching the $(n-2)$th step, and walk two steps in the last move;
   - First reaching the $(n-3)$th step, and walk three steps in the last move.

   Based on the above, the recurrence of F_k can be expressed as follows:

   - $F_1 = 1$, $F_2 = 2$, $F_3 = 4$;
   - For $k > 3$, $F_k = F_{k-3} + F_{k-2} + F_{k-1}$.

   If we compute F_k in a bottom-up manner (i.e., for $k = 1, 2, \ldots, n$), each F_k can be computed in $O(1)$ time. Thus, F_n can be found in $O(n)$ time.[1]

2. A combination is losing if every move leads to a winning combination (for our opponent). Conversely, a combination is winning if at least one of the moves leads to a losing combination, since we can take such a move and force our opponent to lose.

   Based on this observation, we can derive the recurrence for $A(i, j)$ as follows:

   - $A(0, 0) = L$;
   - If every move of (i, j) leads to a winning combination, then $A(i, j) = L$;
     Precisely, $A(i, j) = L$ if
     - $A(i', j) = W$ for all $0 \le i' < i$,
     - $A(i, j') = W$ for all $0 \le j' < j$, and
     - $A(i-k, j-k) = W$ for all $1 \le k \le \min\{i, j\}$;
   - Otherwise, $A(i, j) = W$.

   Since the value of $A(i, j)$ depends only on the values of $A(k, \ell)$ with $(k + \ell) < (i + j)$, we can compute $A(i, j)$ according to the increasing order of $i + j$. For instance, we can first compute $A(1, 0), A(0, 1)$, then $A(2, 0), A(1, 1), A(0, 2)$, then $A(3, 0), A(2, 1), A(1, 2), A(0, 3)$, and so on. The time to compute each $A(i, j)$ is $O(i + j)$, which is $O(n)$.

   As x and y are at most n, to obtain $A(x, y)$, there are at most $O(n^2)$ entries of A to be computed. The total time to compute all such entries is thus $O(n^3)$.

3. Let s be the rightmost (farthest) gas station within the first n km from SF. Then, we can show the following:

   Lemma 1. *There exists an optimal solution (using the fewest number of gas stations) whose first station is s.*

[1] Based the recurrence, we can in fact compute F_n in $O(\log n)$ time using matrix multiplication.

Proof. (By cut-and-paste argument.) Consider an optimal solution $OPT = (s_1, s_2, \ldots, s_k)$, with stations ordered from left to right. If $s_1 = s$, then the lemma is correct. Otherwise, we replace s_1 by s, and remove redundant gas stations (if any). It is easy to see that s_1 must be on the left of s, since if not, the distance of s_1 and SF is more than n km. Thus, after the replacement, we obtain another solution that can allow us to travel from SF to Seattle. The new solution uses at most the same number of gas stations as OPT, which must be optimal. Thus, the proof completes. □

Let OPT be an optimal solution that contains s. Suppose OPT has k gas stations.

Lemma 2. *By removing s from OPT, the remaining $k-1$ gas stations must form an optimal solution to travel from s to Seattle, starting with a full-tank at s.*

Proof. (By contradiction.) Let OPT' be an optimal solution to travel from s to Seattle. Suppose on the contrary that the lemma is incorrect. Then, OPT' must use fewer than $k-1$ gas stations, since $OPT - \{s\}$ is a feasible solution to travel from s to Seattle. Consequently, $OPT' \cup \{s\}$ has at most $k-1$ gas stations, which implies a solution to travel from SF to Seattle with fewer gas stations than OPT, leading to a contradiction. □

The previous lemmas suggest the following algorithm to find the optimal set of gas stations:

```
Choose s_1 = rightmost gas station from SF within first n km ;
k = 1 ;
while ( distance(s_k, Seattle) > n ) {
    Choose s_{k+1} = rightmost gas station from s_k within first n km ;
    k = k + 1 ;
}
```

4. We give two different potential functions Φ that can show the desired amortized bounds.

   Solution 1: For each node u in the heap, $\Phi(u)$ = size of the subtree rooted at u. Then, for a heap H, $\Phi(H) = \sum_{u \in H} \Phi(u)$ = sum of potentials of all the nodes. It is easy to check that at any time, $\Phi(H) \geq 0$.

   When `Insert` is performed, at most $\log n$ nodes will increase each of their potential by 1, so that the potential of H will be increased by at most $\log n$. Since the actual cost of insertion is also $\log n$, the amortized cost of `Insert` is $O(\log n)$.

   When `Extract-Min` is performed, each ancestor of the last node of the heap will decrease their potential by 1, which is the same as the actual cost of `Extract-Min`. Thus, its amortized cost is $O(1)$.

   Solution 2: For each node u in the heap, $\Phi(u)$ = node-depth of u. Then, for a heap H, $\Phi(H) = \sum_{u \in H} \Phi(u)$ = sum of potentials of all the nodes. It is easy to check that at any time, $\Phi(H) \geq 0$.

   When `Insert` is performed, a new node is created with node-depth at most $\log n$. Consequently, the potential of H will be increased by at most $\log n$. Since the actual cost of insertion is also $\log n$, the amortized cost of `Insert` is $O(\log n)$.

   When `Extract-Min` is performed, the last node of the heap is removed. This causes a drop in the potential by its node-depth, which is the same as the actual cost of `Extract-Min`. Thus, the amortized cost of `Extract-Min` is $O(1)$.

5. Let b be the binary representation of the number of elements before an insertion, and let r be the number of consecutive `1`'s at the rightmost of b. Here, r ranges from 0 to $\lfloor \log n \rfloor$.

   After an insertion, the binary representation is changed such that the $(r+1)$th rightmost bit changes from `0` to `1`, while the last r bits all become `0`.

   Consequently, we need update the set of sorted arrays. One way to do so is to merge the r arrays corresponding to the r `1`'s, and merge them together with the newly inserted element. We can do so by merging the newly inserted element with the 1-element array, forming a 2-element array; then, merge this with the original 2-element array to form a 4-element array; and so on. This process can be done in $O(2^r)$ time.

   When m insertions (with $m \leq n$) are performed, the number of times that a binary representation has exactly r rightmost consecutive `1`'s is $O(m/2^r)$. Thus, the total time for m insertions is at most:

$$\sum_{r=0}^{\lfloor \log n \rfloor} O(m/2^r) \times O(2^r) = O(m \log n).$$

   Thus, the amortized cost of insertion is $O(\log n)$.

6. (a) Suppose on the contrary that there exist distinct $(x, x+k)$ and $(y, y+k)$ which are both losing. WLOG, let $y > x$. Now, by removing $y - x$ coins from both piles in $(y, y+k)$, we obtain a losing combination $(x, x+k)$. Thus, contradiction occurs.

   (b) Suppose on the contrary that there exist distinct (x, r) and (x, r') which are both losing. WLOG, let $r' > r$. Now, by removing $r' - r$ coins from the second pile in (x, r'), we obtain a losing combination (x, r). Thus, contradiction occurs.

   (c) We shall prove the statement "L_k is a losing combination" by induction.

   Base case: Since L_0 is losing, the statement is correct for $k = 0$.

   Inductive case: Suppose the statement is true for $k = 0, 1, \ldots, t-1$. Then, consider $L_t = (v, v+t)$. We shall prove L_t is losing by showing each move of L_t leads to a winning combination. There are three cases for a move:

   i. *Take from 1st pile:*
   We obtain a combination $(v', v+t)$ with $v' < v$. By the choice of v, v' appears in L_k for some $k \leq t-1$. This implies there is a losing combination (v', x) or (x, v') with $|x - v'|$ at most $t - 1$. By part (b), we see that $(v', v+t)$ cannot be losing.

   ii. *Take from both piles:*
   We obtain a combination $(v', v'+t)$ with $v' < v$. Similarly, by part (b), we see that $(v', v'+t)$ cannot be losing.

   iii. *Taking from 2nd pile:*
   If taking at least t coins, we get a combination (v, v') with $v' < v$. By our choice of v, there is some losing combination (v', x) or (x, v') with $x < v$. Then, by part (b), we see that (v, v') cannot be losing.
   If taking less than t coins, we get a combination $(v, v+t')$ with $t' < t$. By our choice of v, there is some losing combination $(v', v'+t')$ with $v' < v$. Then, by part (a), we see that $(v, v+t')$ cannot be losing.

   In conclusion, every move of $(v, v+t)$ leads to a winning combination. Thus, L_t must be losing, and this completes the proof of the inductive case.

   (d) The v-value of L_k is strictly increasing. Thus, the v-value of L_x, v_x, is at least x. This implies that $L_0, L_1, \ldots, L_x$ must contain all values at most v_x, and thus containing x.

(e) The most time-consuming step is to find the smallest unseen number. We can make this efficient by using an auxiliary array A to record whether a number is seen or not. At any time, we maintain a pointer P pointing at the smallest unseen number, so that each time v can be found in $O(1)$ time. Once L_k is computed, we update A accordingly, and move the pointer P rightwards, one entry after another, until it locates the next unseen number.

Since $x \leq n$, it is easy to check that A is updated at most $O(n)$ times, and the pointer P is moved at most $O(n)$ times. The total time spent is $O(n)$.

(f) We compute $L_0, L_1, \ldots, L_x$ in $O(n)$ time and find the losing combination that contains x. If that combination is (x, y) or (y, x), we conclude immediately that (x, y) is losing. Otherwise, by part (b), we can also conclude immediately that (x, y) is winning.