CS 3510 Honors Algorithms
Solutions : Homework 5

Problem 1. [**2-Universal Hash Functions**]
We can manually inspect the sets and find out the probabilities of collision. For every pair of inputs $x \neq y$, we can check when they get mapped to the same hash value.

x	y	Colliding functions
1	2	h_3
1	3	h_2
1	4	h_1
2	3	h_1
2	4	h_2
3	4	h_3

For every pair of inputs $x \neq y$ there is only one hashing function from the family H which results in collision. Assuming each $h_i \in H$ is chosen uniformly, the probability of collision is $1/3$, which is less than $1/2$. Hence this is family is 2-Universal.

Problem 2. [**More 2-Universal Hash Functions**]

(a) Yes, this is a 2-Universal set of hash functions. Every $h \in H_a$ is determined by the choice of $(a_1, a_2) \in [m]^2$. So we need $2 \log m$ random bits to sample from the family. Let $(x_1, x_2), (y_1, y_2) \in [m]^2$ such that $(x_1, x_2) \neq (y_1, y_2)$. Let us see the cases when they collide ie., for what values of (a_1, a_2) do we get $h(x_1, x_2) = h(y_1, y_2)$.

Let $h(x_1, x_2) = h(y_1, y_2)$. That means $a_1x_1 + a_2x_2 = a_1y_1 + a_2y_2 \bmod m$. Equivalently, $a_1(x_1 - y_1) = a_2(x_2 - y_2) \bmod m$. We can consider two cases.

Case 1: $x_1 \neq y_1$. Then $x_1 - y_1 \neq 0$ and will have an inverse modulo m. So $a_1 = a_2(x_2 - y_2)/(x_1 - y_1) \bmod m$. So for every $k \in [m]$ where $a_2 = k$, we have exactly one colliding value of a_1. We can pick a_2 in m ways and that fixes the colliding a_1. So number of hash functions that will collide is m.

Case 2: $x_1 = y_1$. Then $x_2 \neq y_2$ and we can make a similar argument to that in Case 1 and conclude that there are m colliding hash functions.

Since the number of colliding hash functions is m in either case, and there are m ways each to pick a_1 and a_2 (independently), there are a total of m^2 hash functions in H_a.

$$Pr[h(x_1, x_2) = h(y_1, y_2)] = m^2/m = 1/m$$

So the set is 2-Universal.

(b) This is same as the previous problem, but here we have m a power of 2, and not a prime number. The number of random bits needed to represent the hash function is $2 \log m = 2k$.

This is not a 2-Universal hash family. As a counter example, look at the pair of inputs $(0, 0)$ and $(0, 2^{k-1})$.

Clearly $h(0, 0) = 0$ for all pairs (a_1, a_2). Also $h(0, 2^{k-1}) = a_2 2^{k-1} \bmod 2^k$, which is equal to zero if and only if a_2 is even. So there are $m/2$ hash functions for which $h(0, 0) = h(0, 2^{k-1})$. So $Pr[h(0, 0) = h(0, 2^{k-1})] = 1/2 > 1/m$. So this is not a 2-Universal family.

(c) Let us count the number of all functions from $[m]$ to $[m-1]$. Every $x \in [m]$ can choose any image in $[m-1]$, that is $m-1$ possible options. Each of the m elements in $[m]$ have to choose an image to specify the function. This can be done in $(m-1)^m$ ways. So we require $\log(m-1)^m = m \log(m-1)$ random bits to specify the hash function.

This is a 2-Universal function. Given any pair $x, y \in [m]$, let us see what the probability of collision is. Let $f(x) = k \in [m-1]$ for some k. Then for choosing $f(y)$, we have $m-1$ options, one of which is k. So $Pr[f(x) = f(y)] = 1/(m-1)$ for any pair $x, y \in [m]$. Hence the family is 2-Universal.

Problem 3. [**Hashing by Adversary**]

(a) Here we can easily attack by picking $O(m)$ elements, all of which are mapped to the same hash value. Since size of U is large compared to m, we can always do this. So all the elements would result in a collision, and we can query one of those elements which we inserted. One would have to look through $O(m)$ elements on average to retrieve the queried element.

(b) If the universe is big compared to m, there would always be elements for which the k hash values would be some $t_1, t_2, \ldots, t_k$, where $t_i \in \{1, 2, \ldots, m\}$ according to our choice. Then we could pick elements in such a way that each element covers a different set of k values in $\{1, 2, \ldots, m\}$. Thus, after $\lceil m/k \rceil$ insertions, all the m entries would be 1, and every query would return YES. So probability of false positive would be 1.

(c) By an attack of this sort, the data structure gets far more heavily loaded than it should be. The analysis of hashing/bloom filters assume random inputs. We have shown in the preceding parts that if the inputs are adversarial, things can be much worse.

Using these attacks, the attacker could aim for rendering some server out of service by loading it by planned attacks. If I run my own server, and I want more people to use my server, I can launch these Denial of Service attacks to render my competitor out of service.

Problem 4. [**Increasing Subsequences**]

(a) Consider the sequence $A = \{6, 7, 3, 4\}$. Clearly the longest increasing subsequence is of length 2 here. But the algorithm will compute as follows. $M[1] = 1, M[2] = 2, M[3] = 2, M[4] = 3$. That is, according to the algorithm, the length of the longest increasing subsequence is 3.

(b) Instead of using the naive algorithm in part (a), we can think of a more sophisticated technique. Let entry $M[i]$ contain the longest increasing subsequence that has $A[i]$ as its last element (note that this subsequence **must** have $A[i]$ in it). The array M is of length n. At the end of the algorithm, $\max_i M[i]$ will be the length of longest increasing subsequence.

As a base case, we have $M[1] = 1$. We can fill up M in the increasing order of indices, using the recursive equation below.

$$M[i] = \max_{1 \leq k < i : A[k] \leq A[i]} M[k] + 1$$

This is because the longest increasing subsequence containing $A[i]$ will have as its predecessor, an element $A[k]$ which is not bigger than itself. (**Note:** We assume that the max function returns 0 in the case when the set is empty, ie., all the elements prior to $A[i]$ are bigger than $A[i]$)

To find the longest subsequence itself, we can have another array P, where $P[i]$ contains the index to the predecessor of $A[i]$ in the longest increasing subsequence containing $A[i]$ (in the case when all the previous elements are bigger than $A[i]$, $P[i] = 0$. This P array can be used to back track and print out the longest increasing subsequence.

```
Longestsub{
    for i = 1 to n   {
         M[i] = 1
         P[i] = 0
         for k = 1 to i-1 {
              if (A[k] ≤ A[i]) {
                   if (M[i] < M[k] + 1) {
                       M[i] = M[k] + 1
                       P[i] = k
                   }
              }
         }
    }
}

Printlongest (A,M,P,n) {
    t = argmax (M) // Takes O(n) time and returns the index of max in M
    Printlongest (A,M,P,t-1)
    Print number A[t]
}
```

The two for loops in Longestsub causes it to take $O(n^2)$ time. The argmax in Printlongest will take $O(n)$ time, and it recurses called atmost $O(n)$ times, giving a maximum running time of $O(n^2)$.

Problem 5. [**Gasoline Refilling**]
Let $M^i(g)$ be the minimum gas bill when leaving the gas station i with g gallons of fuel in the tank, possibly **after** purchasing gas at staion i. The variables range as $1 \leq i \leq n$ and $0 \leq g \leq C$.

Let there be h gallons of fuel in the tank when leaving station $i-1$. Since the car needs to have sufficient fuel to reach station i, $(d_i - d_{i-1})/m \leq h \leq C$. Also $h \leq (d_i - d_{i-1})/m + g$ because the fuel at station i can be atmost g, because we cannot purchase negative quantity of fuel. So the recursive equation is

$$M^i(g) = \min_h [M^{i-1}(h) + (g + (d_i - d_{i-1})/m - h)c_i]$$

where h runs from $(d_i - d_{i-1})/m$ to $\min(C, (d_i - d_{i-1})/m + g)$. The base case is given by

$$M^1(g) = c_1 g \text{where } 0 \leq g \leq C$$

The answer will be $\min_{g=0}^{C} M^n(g)$. It is easy to see that the cheapest solution will involve arriving at station n with 0 gallons, so the answer is simply $M^n(0)$. We can compute the matrices M^i in the increasing order of indices i. Since we need only M^{i-1} to compute M^i, we can reuse the space. The following pseudo code uses only two arrays M and N, and reuses the space effectively, instead of a two dimensional array.

```
GasolineRefilling(n, d[], c[]) {
    for g = 0 to C
        M[g] = c[1] * g  // base case
    for i = 2 to n {
        for g = 0 to C {
            N[g] = infinity     // N is temp array
            for h = (d[i] - d[i-1])/m to min(C, (d[i] - d[i-1])/m + g) {
                cost = M[h] + (g + (d[i] - d[i-1])/m - h) * c[i]
                if (cost < N[g])
                    N[g] = cost
            }
            M[g] = N[g]   //copy entries from temp array
        }
    }
    return M[0]
}
```

There are 3 nested for loops, one ranging over $n-1$ values, one ranging over $C+1$ values and another one ranging over atmost C values. The running time is $O(nC^2)$.

Note: Because the distances are divisible by m, one can argue that we cannot achieve better by purchasing fractions of gallons, so the integer solution found is indeed the best possible.

Problem 6. [**Card Game**]

(a) Consider the sequence of 4 cards 3, 10, 2, 1. If the first player takes the biggest card available, ie. 3, then the opponent can take the card of value 10 and win. Whereas if player 1 had taken card with value 1, the opponent would have taken either 3 or 2, leaving the 10 open for the first player and assuring him a win. So choosing the biggest card on turn 1 is not the optimal strategy.

(b) Let $M(i,j)$ be the maximal total value of the cards that can be chosen, assuming one has the first move, in a subgame with cards s_i to s_j, where $1 \leq i \leq j \leq n$. We assume that the opponent uses the best strategy and try to find the best that we can do in that case. The recursive equation simply takes the maximum of the scores after the two possible moves of the player with the turn, ie., taking card s_i or card s_j.

$$M(i,j) = \max\left(s_i + \sum_{k=i+1}^{j} s_k - M(i+1,j), s_j + \sum_{k=i}^{j-1} s_k - M(i,j-1)\right)$$

which simplifies to

$$M(i,j) = \sum_{k=i}^{j} s_k - \min(M(i+1,j), M(i,j-1))$$

The base case is $M(i,i) = s_i$.

In order to meet the $O(n^2)$ requirement, we have to perform the following trick, we can precompute the partial sums $S_i = \sum_{k=1}^{i} s_k$ for i from 0 to n. This enables us to perform the summation $\sum_{k=i}^{j} s_k$ in constant time because $\sum_{k=i}^{j} s_k = S_j - S_{i-1}$. We can evaluate the matrix M in the increasing order of $d = j - i$.

```
Cardgame(n,s[]) {
    S[0] = 0
    for i = 1 to n
        S[i] = S[i-1] + s[i]     // compute partial sums
    for i = 1 to n
        M[i,i] = s[i]     // base case
    for d = 1 to n-1 {
        for i = 1 to n-d {
            j = i + d
            M[i,j] = S[j] - S[i-1] - min (M[i+1,j], M[i,j-1])
        }
    }
}
```

During the game, the optimal move can be determined by looking at which of the two parameters to `min` was actually the minimum. More precisely, if $i = j$, there is only one move, else if $i < j$, then compare $M(i+1, j)$ and $M(i, j-1)$. If $M(i+1, j)$ is smaller, then pick s_i, else pick s_j.

There are two nested for loops each ranging over at most n values, so the running time is $O(n^2)$. As explained above, each move can be calculated in $O(1)$ time from the matrix M.

(c) Look at all the even numbered cards and the odd numbered cards at the beginning. If you sum both of them, one of the two sums will be greater than or equal to the other. Pick that set. If it is the even set, player 1 can pick s_n, leaving player 2 with two odd numbered cards. Whatever player 2 picks, player 1 can pick the even numbered card and leave player 2 with only odd numbered cards at each turn. By this strategy, player 1 ensures that he picks the even numbered half and player 2 gets the odd numbered half. This ensures that player 1 has a sum atleast as much as player 2, if not more.

If it is the odd numbered set which is bigger, we can similarly argue that player 1 has a strategy which ensure that he does not lose.