CS240 Homework 2

Shen Linfeng Student ID: 2018E8018461028

March 20, 2019

Problem 1

(a) Before you buy your ith toy, if you have exactly j toy types, the probability of the i toy is the type you already have is $\frac{j}{n}$; before you buy your ith toy, if you have exactly $j-1$ toy types, the probability of the ith toy is the type you have not had is $\frac{n-(j-1)}{n} = \frac{n-j+1}{n}$.

So, we have

$$p_{i,j} = \frac{j}{n} p_{i-1,j} + \frac{n-j+1}{n} p_{i-1,j-1}$$

(b) Suppose you only have 0 toys before the first purchase. Then we have some base case:

$$p_{1,1} = 1$$
$$p_{i,j} = 0 \ , i > j$$

Direct use of recursion from will result in $O(2^n)$, but with saving past p values we can calculate $p_{i,j}$ in $O(ij)$. The algorithm is shown below:

TOY(i, j, n)

Initialize a $i \times j$ array p with 0, let $p[1][1] = 1$
if $i > j$
    return 0
for $a = 1$ **to** i
    for $b = 1$ **to** i
        $p[a][b] = \frac{b}{n} p[a-1][b] + \frac{n-b+1}{n} p[a-1][b-1]$
        if $b \geqslant j$
            break
return $p[i][j]$

Problem 2

Let v, w be the arrays store value and weight, and W is the capacity.

KNAPSACK(n, v, w, W)

$v\text{-}sum = \sum_{i=1}^{n} v[i]$
Initialize a $(n+1) \times (v\text{-}sum + 1)$ array opt with ∞ (for clearly, let the index begin from 0 here).
for $i = 1$ **to** n
    for $j = 1$ **to** $v\text{-}sum$
        if $j \geqslant v[i]$
            $opt[i][j] = \min\{opt[i-1][j], opt[i-1][j - v[i]] + w[i]\}$
        else
            $opt[i][j] = opt[i-1][j]$

Initialize an array $c[n]$
using Binary Search by the index j of $opt[i][j]$ for every i, to find $c[i] = \max\{j \mid opt[i][j] \leqslant W\}$
return $\max\{c[i]\}$

The two for loop will cost $O(nv\text{-}sum)$, and n Binary Search will cost $O(n \log v\text{-}sum)$. From $v_i < n$ we can know $v\text{-}sum < n^2$, so this algorithm will run in $O(n^3)$.

$opt[i][j]$ is the min weight subset of items $1, \cdots, i$ with value j. There exist two case:

- *opt* does not select item i.
  opt selects best of $\{1, 2, \cdots, i-1\}$ using value limit j.

- *opt* selects item i.
  new value limit $= j - v[i]$.
  opt selects best of $\{1, 2, \cdots, i1\}$ using this new value limit.

The rest of the analysis is similar to the original Knapsack problem, which can prove the correctness.

Problem 3

Let's create a data structure with: $S[i].x = x_i$, $S[i].y = y_i$.

```
COUNTING-FRIENDS(S, n)
Sort S in ascending order by S[i].x, it will cost O(n log n).
return COUNT(S, n)
```

```
COUNT(S, n)
if n == 1
    return 0

mid = ⌊n/2⌋
count = 0, i = j = k = 1
S-left = S[1 : mid]
S-right = S[mid + 1 : n]
left = COUNT(S-left, mid)
right = COUNT(S-right, n - mid)

while i ⩽ mid and j ⩽ n - mid
    if S-left[i].y > S-right[j].y
        count+ = mid - i + 1
        S[k] = S-right[j]
        j + +
    else
        S[k] = S-left[i]
        i + +
    k + +
if i ⩽ mid
    S[k : n] = S[i : mid]

return left + right + count
```

This problem can be convert to: when S be sorted by $S.x$, then the number of Reverse Pairs is the answer. So it can be solved by Merge Sort.

This algorithm modifies from Merge Sort, still run in $O(n \log n)$.

Problem 4

Let array C store the cards, Equ be the equivalence tester.

EQUIVALENT-DETECTION(C, n)

```
tmp = C[1]
num = 1
for i = 2 to n
    if Equ(tmp, C[i])
        num++
    else
        num--
        if num == 0
            tmp = C[i]
            num = 1
num = 0
for i = 1 to n
    if Equ(tmp, C[i])
        num++
if num > n/2
    return True
else
    return False
```

Both two for loop run in $O(n)$, so this algorithm will run in $O(n)$.

If there exist than $n/2$ cards that are all equivalent to one another, let the number of those cards be n_1. When in the first for loop and *tmp* equivalent to those cards, assume it past n_2 other type cards, then we have $n_2 \leqslant n - n_1 < n/1$. So we have $n_1 - n_2 > 0$, which mean after the first for loop end, if there exist than $n/2$ cards that are all equivalent to one another the *tmp* is what we want(If exist). And the second for loop will check if it is true.

Problem 5

Let A,B,C be the three sequences, and m, n be the length of A and B.

```
SEQUENCE-MERGING(A, B, C, m, n)
Initialize a (m + 1) × (n + 1) bool array T
for i = 0 to m
    for j = 0 to n
        if i == 0 and j == 0
            T[i][j] = True
        elseif i == 0 and B[j] == C[j]
            T[i][j] = T[i][j - 1]
        elseif j == 0 and A[i] == C[j]
            T[i][j] = T[i - 1][j]
        elseif A[i] == B[j] == C[i + j]
            T[i][j] = T[i - 1][j] || T[i][j - 1]
        elseif A[i] == C[i + j]
            T[i][j] = T[i - 1][j]
        elseif B[i] == C[i + j]
            T[i][j] = T[i][j - 1]
        else
            T[i][j] = False
return T[m][n]
```

Two for loop will cost $O(mn)$, so this algorithm will run in $O(mn)$.

$T[i][j]$ store the possibility of if the first i of A and the first j of B can compose the first $i+j$ of C. All situations are considered in the for loop, so this algorithm will run well.

Problem 6

POLYNOMIAL MULTIPLICATION(A, B, n)

Initialize a 2^k array C with $0(k = \min\{k|\ 2^k > n, k \in \mathbb{Z}\})$.
for $i = 0$ **to** $2^k - 1$
    for $p = 0$ **to** $n - 1$
        $C[i] = C[i] + A[p] \cdot B[p \oplus i]$
return C

The correctness can be proved by the definition and the fact of $p \oplus (p \oplus i) = i$. Unfortunately this is a brute force, will cost $O(n^2)$.