Homework #4
Solution Manual

Problem 1 Money (programming assignment)

```
#include <stdio.h>
#define N 110
#define M 100010
long long dp[N][M],que[M];
int main()
{
    int m,n,z,b,s;
    scanf("%d%d",&m,&n);
    for ( int i=1; i<=n; i++ ) {
        scanf("%d%d%d",&z,&b,&s);
        if ( b>m ) continue;
        for ( int j=0; j<b; j++ ) for ( int k=0,ql=0,qr=0; j+k*b<=m; k++ ) {
            long long now=dp[i-1][j+k*b]-1LL*k*s;
            while ( ql<qr && dp[i-1][j+que[qr-1]*b]-que[qr-1]*s<now ) qr--;
            que[qr++]=k;
            while ( ql<qr && que[ql]+z<k ) ql++;
            dp[i][j+k*b]=dp[i-1][j+que[ql]*b]+(k-que[ql])*s;
        }
    }
    long long ans=0;
    for ( int i=0; i<=m; i++ ) if ( dp[n][i]>ans ) ans=dp[n][i];
    printf("%lld\n",ans);
    return 0;
}
```

It is easy to derive a dynamic programming solution where $f(i,j)$ means the he maximum money you can get with j coupons when only exchange the first i kinds of items. and the recursion is

$$f(i,j) = \max_{0 \le k \le n_i} f(i-1, j-k \times b_i) + k \times s_i$$

We can maintain a double ended queue which stores the candidates, and make the cost for transition in a fixed i be $O(M)$.

The key observation is that in that recursion, the value of j mod b_i we considered is fixed, so we can solve then independently, and for each subproblem, we can solve it just like a moving window problem.

Problem 2 Queue

1) As follows.

```
def ENQUEUE(x):
    stack1.push(x)
def DEQUEUE():
    if stack2.empty():
        while not stack1.empty():
            stack2.push(stack1.pop())
    return stack2.top()
```

2) As follows.

Operation	Stack1	Stack2	Return
ENQUEUE(a)	a	(empty)	(null)
ENQUEUE(d)	ad	(empty)	(null)
ENQUEUE(a)	ada	(empty)	(null)
DEQUEUE()	(empty)	ad	a
DEQUEUE()	(empty)	a	d

3) A) As the pseudo code above, we only run `stack1.push(x)` when enqueuing, which take exactly one push.

B) There are at most n elements in stack1, so the code in while loop when dequeuing will be executed at most n times, and there is one additional pop need taken be return. Thus there are at most $n+1$ pops and n pushes.

C) Performing n enqueue takes n pushes, and n dequeue takes $n \times O(n) = O(n^2)$ pops and pushes, so there are totally $O(n^2)$ pushes and $O(n^2)$ pops.

4) A) Let the amortized cost enqueue and dequeue be 4 and 0 respectively. Since for any element x, the actual cost of it when enqueue is 1 by `stack1.push()`, and the cost for dequeue will be no more than 3 since it may be operated as `stack1.pop()`, `stack2.push()`, `stack2.pop()`, each at most 1 time. We would not be out of money, and the amortized cost for each operation is $O(1)$, so the total cost is $O(n)$.

B) Let $\Phi(D) = 2 \times$ `stack1.size()`, and s is the size of stack1 before the operator for convenience.
It is easy to see that $\Phi(D_0) = 0$.
If the i-th operator is enqueue, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2(s+1) - 2s = 3$
If the i-th operator is dequeue and stack2 is empty, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 2s + 1 + 0 - 2s = 1$
If the i-th operator is dequeue and stack2 is not empty, we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2s - 2s = 1$
So we have $\Phi(D_i) \geq 0$ and $\hat{c}_i = O(1)$, which implies the total cost is $O(n)$.

Problem 3 Counter

1) Consider the operation sequence as $m = 2^n$ increments, m decrements/increments alternatively. which costs $\Theta(m) + m \times \Theta(n) = \Theta(m \lg m) \neq O(m)$.

2) Let $\Phi(D) = \#$non-zero in A[]. It is easy to see that $\Phi(D_0) = 0$.

If the i-th operation is increment, which makes $0, 1, \ldots, p-1$-th bit be zero, and p-th bit from 0 to 1. we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (p+1) + (-p) + 1 = 2$

If the i-th operation is increment, which makes $0, 1, \ldots, p-1$-th bit be zero, and p-th bit from -1 to 0. we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (p+1) + (-p) - 1 = 0$

If the i-th operation is decrement, which makes $0, 1, \ldots, p-1$-th bit be zero, and p-th bit from 0 to -1. we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (p+1) + (-p) + 1 = 2$

If the i-th operation is decrement, which makes $0, 1, \ldots, p-1$-th bit be zero, and p-th bit from 1 to 0. we have $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = (p+1) + (-p) - 1 = 0$

So the amortized cost of each operation is $O(1)$.

Problem 4 TABLE-DELETE

1) Since $\alpha_{i-1} \geq \frac{1}{2}$, the case $\alpha_i < \frac{1}{4}$ won't occur when num_i is sufficient large, and the cost of that case can be treat as constant, so we can make the assumption that $\alpha_i \geq \frac{1}{4}$, which means it would not trigger the contraction.

If $\alpha_i \geq \frac{1}{2}$, we have

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 1 + (2\text{num}_i - \text{size}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\
&= 1 + (2(\text{num}_{i-1} - 1) - \text{size}_{i-1}) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\
&= -1
\end{aligned}$$

If $\alpha_i < \frac{1}{2}$, we have

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\
&= 1 + (\text{size}_i/2 - \text{num}_i) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\
&= 1 + (\text{size}_{i-1}/2 - (\text{num}_{i-1} - 1)) - (2\text{num}_{i-1} - \text{size}_{i-1}) \\
&= 2 + \frac{3}{2}\text{size}_{i-1} - 3\text{num}_{i-1} \\
&= 2 + \frac{3}{2}\text{size}_{i-1} - 3\alpha_{i-1}\text{size}_{i-1} \\
&\leq 2
\end{aligned}$$

So the amortized cost of the operation with respect to the potential function is bounded by a constant.

Problem 5 Making Binary Search Dynamic

1) A) Simply perform a binary search on each A_i where $n_i = 1$.

   B) If $n = 2^m - 1$, all $n_i = 1$, so we need to perform binary search on all A_i, which takes $\sum_{i=0}^{k-1} \lg 2^i = \sum_{i=0}^{k-1} i = k(k-1)/2 = O(k^2) = O(\lg^2 n)$.

2) A) Note that just like merge sort, we can merge two ordered list in linear time.

```
1  def INSERT(x):
2      new_list = [x]
3      p = 0
4      while A[p].full():
5          new_list.merge(A[p])
6          A[p].clear()
7          p += 1
8      A[p] = new_list
```

   We will need to merge the p-th array for each 2^p insertions, so it costs $\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor 2^i \leq \sum_{i=0}^{k-1} n = nk = O(n \lg n)$, which implies the amortized cost for each insertion is $O(\lg n)$.

   B) If $n = 2^m - 1$, all $n_i = 1$, we need to merge all A_i, which costs $O(n)$ time.

3) A) Simply merge all A_i into a list, clear the whole structure, remove that element, and then insert all element again. It costs $O(n) + O(n) + O(n) + O(n \lg n) = O(n \lg n)$.