Homework #2
Solution Manual

1 Open!! (programming assignment)

```
#include <stdio.h>
#define N 500
void chkmin( int *a, int b ) {
    if ( b<*a ) *a=b;
}
int before[N],after[N],diff[N],dp[N][N][10];
int main()
{
    int i,j,k,l,m,n;
    scanf("%d",&n);
    for ( i=0; i<n; i++ ) scanf("%1d",before+i);
    for ( i=0; i<n; i++ ) scanf("%1d",after+i);
    for ( i=0; i<n; i++ ) diff[i]=(after[i]-before[i]+10)%10;
    for ( i=0; i<n; i++ )
        for ( j=0; j<10; j++ )
            dp[1][i][j]=(diff[i]-j+10)%10;
    for ( l=2; l<=n; l++ )
        for ( i=0; i+l<=n; i++ ) {
            for ( j=0; j<10; j++ ) {
                dp[l][i][j]=10*N;
                for ( m=1; m<l; m++ )
                    chkmin(&dp[l][i][j],dp[m][i][j]+dp[l-m][i+m][j]);
            }
            for ( j=0; j<10; j++ )
                for ( k=0; k<10; k++ )
                    chkmin(&dp[l][i][j],dp[l][i][k]+(k-j+10)%10);
        }
    printf("%d\n",dp[n][0][0]);
    return 0;
}
```

We use $a\%b$ to denote a mod b for simplicity.
Let $S[0..n-1]$ be the starting state and $T[0..n-1]$ be the target state which can open the lock. What we want to do is same as rotate the lock from 00...0 to $D[0..n-1]$, where $D[i] = (S[i] - T[i])\%10$.
Let $c(l, i, j)$ be the minimum cost to rotate $jj...j$ to $D[l..l+i-1]$, we have the recurrence

$$c(l,i,j) = \begin{cases} w(j, D[i]) & \text{if } l = 1; \\ \min_{\substack{1 \le m < l \\ 0 \le k < 10}} \{c(m,i,k) + c(l-m, i+m, k) + w(j,k)\} & \text{if } l > 1. \end{cases}$$
$$w(a,b) = (b-a)\%10$$

, where $w(a, b)$ denotes the cost to rotate a single digit from a to b. Straightforwardly apply the above recurrence will lead an $O(100N^3)$ algorithm, but if we notice that when l, i are fixed, there must be a j such that the optimal parameter $k = j$ for $c(l, i, j)$, which implies $w(j, k) = 0$ in that situation. Thus we can divide the parameter selection of m, k into two steps instead of one step, and make an $O(10N^3)$ algorithm which can get full score!

2 Televisions

Note that it is easy to observe that each TV would serve a consecutive segment on the street, and the optimal way to serve some families by 1 TV is installing TV for the median family. Let the non-decreasing sequence $p_1, p_2, \ldots, p_n$ be the positions of families on the street, we also use p_i to denote that family for simplicity.

1) Suppose that in the optimal installation for $p_1, p_2, \ldots, p_m$ by t TVs, we serve $p_i, p_{i+1}, \ldots, p_m$ by 1 TV, and $p_1, p_2, \ldots, p_{i-1}$ by $t-1$ TVs. The way we serve the last $m-i+1$ families by 1 TV must be optimal, since if there is a less costly way, we could substitute that way in installation for $p_1, p_2, \ldots, p_m$, and produce another installation whose cost was lower than optimum: a contradiction. A similar statement also holds for the first $i-1$ families by same reason, thus this problem exhibits optimal substructure.

2) Let $c(m,t)$ be the cost function denote the cost in optimal installation for first m families by t TVs, we have the recurrence

$$c(m,t) = \begin{cases} 0 & \text{if } t \geq m; \\ w(1,m) & \text{if } t = 1; \\ \min_{1<i\leq m} \{c(i-1,t-1) + w(i,m)\} & \text{if } m > t > 1. \end{cases}$$
$$w(l,r) = \sum_{i=l}^{r} \left| p_i - p_{\lfloor \frac{l+r}{2} \rfloor} \right|$$

, where $w(l,r)$ denotes the cost to serve $p_l, p_{l+1}, \ldots, p_r$ by 1 TV in optimal way.

3) As following table.

$t \backslash m$	1	2	3	4	5	6
1	0	3	7	12	15	20
2	0	0	3	4	6	10
3	0	0	0	1	3	6

3 Breaking a String

Let $S[1..n]$ be that n-character string and $L[0] = 1, L[m+1] = n$ for simplicity.

1) Suppose that in the optimal solution for breaking $S[L[l-1]..L[r+1]]$ at $L[l..r]$, we break the string at $L[i]$ first. The way we break $S[L[l-1]..L[i]]$ at $L[l..i-1]$ must be optimal, since if there is a less costly way, we could substitute that way in solution for breaking $S[L[l-1]..L[r+1]]$ at $L[l..r]$, and produce another solution whose cost was lower than optimum: a contradiction. A similar statement also holds for breaking $S[L[i]..L[r+1]]$ at $L[i+1..r]$ by same reason, thus this problem exhibits optimal substructure.

2) Let $c(l,r)$ be the cost function denote the cost to breaking $S[L[l-1]..L[r+1]]$ at $L[l..r]$, we have the recurrence

$$c(l,r) = \begin{cases} 0 & \text{if } l > r; \\ \min_{l\leq i\leq r} \{c(l,i-1) + c(i+1,r)\} + w(l,r) & \text{if } l \leq r. \end{cases}$$
$$w(l,r) = L[r+1] - L[l-1] + 1$$

, where $w(l,r)$ denotes the length of that string segment when we want to break at $L[l..r]$.

3)
```
1: function BREAKSTRING(S[1..n], L[1..m])
2:     L[0] ← 1
3:     L[m + 1] ← n
4:     c ← new int[0..m + 1][0..m + 1]
5:     for l ← 0 to m + 1 do
6:         for r ← 0 to m + 1 do
7:             if l ≤ r then
8:                 c[l][r] ← ∞
9:             else
10:                c[l][r] ← 0
11:    for d ← 0 to m − 1 do
12:        for l ← 1 to m − d do
13:            r ← l + d
14:            for i ← l to r do
15:                c[l][r] ← min(c[l][r], c[l][i − 1] + c[i + 1][r])
16:            c[l][r] ← c[l][r] + L[r + 1] − L[l − 1] + 1
17:    return c[1][m]
```

4 Diamond Pile

1) No, we can not. Since we need to know how many diamonds we can take at most in this turn, and the array c does not contain this information.

2) Here is an $O(N^3)$ algorithm.
```
1: function PILE1(N)
2:     d ← new boolean[1..N][1..N]
3:     for n ← 1 to N do
4:         for k ← 1 to n do
5:             if k ≥ n then
6:                 d[n][k] ← True
7:             else
8:                 d[n][k] ← False
9:                 for i ← 1 to k do
10:                    if d[n − i][min(2i, n − i)] = False then
11:                        d[n][k] ← True
12:    return d[N][N − 1]
```

3) Let $f(n) = \min\{k \mid d[n][k] = \textbf{True}\}$, $f(n)$ always exists since $d[n][n] = \textbf{True}$. By definition of minimum, we have $d[n][k] = \textbf{False}$ for any $k < f(n)$, thus the "only if" part is finished. For the "if" part, consider the winning move such that $d[n][f(n)] = \textbf{True}$, we can also take this move for any $k \geq f(n)$, which implies for any $k \geq f(n)$, $d[n][k] = \textbf{True}$.

4) Note that the minimum winning move of n is taking $f(n)$ diamonds if exists.

```
1: function PILE2(N)
2:     e ← new int[1..N]
3:     for n ← 1 to N do
4:         e[n] ← 1
5:         while e[n] < n and 2e[n] ≥ e[n − e[n]] do
6:             e[n] ← e[n] + 1
7:     return n − 1 ≥ e[n]
```

5) $c[1..15] =$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F	F	F	T	F	T	T	F	T	T	T	T	F	T	T

We claim that the first player would win if and only if N is not a Fibonacci number.

```
1: function PILE3(N)
2:     a, b ← 1, 1
3:     while b < n do
4:         a, b ← b, a + b
5:     return b ≠ n
```

Because Fibonacci number $F_n = \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$ grow exponentially, this algorithm runs in $O(\lg N)$ time.

The proof idea of our claim is that we can represent N in Fibonacci base ($1 = 1, 2 = 10, 3 = 100, 4 = 3 + 1 = 101, 5 = 10000, 6 = 5 + 1 = 10001, 7 = 5 + 2 = 10010, \ldots$), such that there will no two consecutive one since $F_n = F_{n-1} + F_{n-2}$. If N is not a Fibonacci number, the first player always can take the last one in Fibonacci representation for each turn, and then the second player can not take any one out, since $F_n > 2F_{n-2}$, thus the first player will win. If N is a Fibonacci number, assume that the first player take k diamonds, the second player can take the last one, assume that is F_m, since $F_m \leq 2F_{m-1} \leq 2k$. and become the same situation as the case discussed above, thus the first player will lose.