学号:　201618013229011
姓名:　李坚松

Assignment 2

Problem-1

Solution

The problem is to let us to find the largest divisible subset, we can take use of the idea of dynamic programming to solve it. Before we start, we know the fact that if a%b==0&&b%c==0, then a%c==0. We define an array to store the numbers, which is named by A. To reduce search space, we suppose that A is sorted. We define dp[i], which means length of the largest divisible subset of A's sub-array that is formed by the previous i elements of A. Obviously, dp[i] is initialized by 1. By enumerating the i elements, we can know that if A[i]%A[j]==0, then dp[i]=dp[j]+1, where 0<j<i, we update dp[i] to the max value meanwhile. We can get the following **DP equation**:

$$dp[i]=\begin{cases}1, initial\ value \\ \max\{dp[i], dp[j]+1\}, where \text{ j<i, and A[i]\%A[j]==0}\end{cases}$$

To get the subset, we need some extra variables to record the position of the largest divisible subset. Following is pseudo code of the algorithm:

Algorithm: GetLargestDivisibleSubset (int A[])

Input: An array A that consists of distinct positive integers
Output: the largest divisible subset of array A

Begin
```
  sort(A);//sort A in increase order
  int maxIndex=0,curMax=0;//maxIndex records the end of the largest divisible subset
  int previous[A.length];//extra array to record position of the largest divisible subset
  previous[0] = -1;
  int dp[A.length];
  for( int i=0; i<A.length; ++i )
  {
    dp[i] = 1;
    previous[i] = -1;
    for( int j=0; j<i; ++j)
    {
      if( A[i]%A[j]==0 )
```

```
            {
                dp[i] = dp[j] + 1;
                previous[i] = j;
            }
        }
        if( curMax < dp[i] )
        {
            curMax = dp[i];
            maxIndex = i;
        }
    }
    //get the subset
    int index=maxIndex;
    int i=0, result[];//record the subset
    while(index!=-1)
    {
        result[i++] = A[index];
        index = previous[index];
    }
    return result;
End
```

Proof

Since the array is sorted by increase order, we can initialize dp[i]=1,which means that the subset contains one element. If A[i] % A[j]==0, then we can know that A[j] can be joint into the subset, therefore dp[i]=dp[j]+1. After the loop, we can get the answer. However, to get the subsequence of the array, we need some extra variables to record the position of each max-index of A[i]. By backward scanning, eventually we will can get the largest divisible subset.

Time Complexity

For the sort operation, it takes O(nlogn) time, while the loop takes O(n^2). Therefore, the time complexity is O(n^2).

Problem-2

Solution

The problem is to let us to find the maximum amount of money, we can take use of the idea of dynamic programming to solve it. We define an array dp, dp[i] means for the i-th house, the maximum amount of money he can get. Besides, we define the array v, which v[i] means that the i-th house contains v[i] money. We initialize dp[i]=v[i], which means that he rob the i-th house and get the money it contains. For the i-th house, if he rob it, he can get dp[i-2]+v[i], else he will get dp[i-1], then dp[i]=max(dp[i-2]+v[i],dp[i-1]). Therefore, we can get the following **DP equation**:

$$dp[i] = \max\{dp[i-2] + v[i], dp[i-1]\}$$

The pseudo code of the algorithm is shown below:

Algorithm: GetMaxMoney (int v[])

Input: An array v which means the money that each house contains
Output: the maximum amount of money that the robber can get

```
Begin
    int dp[ v.length];
    dp[0]=v[0],dp[1]=max(v[0],v[1]);
    for( int i=2; i<n; ++i)
    {
        dp[i] = max( dp[i-2] + v[i], dp[i-1] );
    }
    return dp[ A.length – 1 ];
End
```

Proof

Suppose for the i-th house, there is a better value dp[i]' that is larger than dp[i], without loss of generality, dp[i] chooses to rob at the i-th house, then dp[i]= dp[i-2]+v[i]. Since dp[i]' is a different choice, it should be dp[i-1]. So dp[i-1] > dp[i-2]+v[i], which means there is a contradiction to the definition of dp. So the suppose is wrong.

Time Complexity

It takes one pass loop to get the answer, therefore its time complexity is O(n).

For the problem 2, if all houses are arranged in a circle, the first house is adjacent to the last house. We can divide the original problem into two sub-problem: including the first house and not including the first house. We can get the solution of the two sub-problem, then answer is the bigger one. The pseudo code of the algorithm is shown below:

Algorithm: GetMaxMoney (int v[])

Input: An array v which means the money that each house contains
Output: the maximum amount of money that the robber can get

```
Begin
```

```
    int dp1[ v.length - 1],dp2[v.length - 1];
    dp1[0]=v[0],dp1[1]=max(v[0],v[1]);
    for( int i=2; i < v.length-1 ; ++i )
    {
        dp1[i] = max( dp1[i-2] + v[i], dp1[i-1] );
    }
    dp2[1]=v[1],dp2[2]=max(v[1],v[2]);
    for( int i=3; i < v.length ; ++i )
    {
        dp2[i] = max( dp2[i-2] + v[i], dp2[i-1] );
    }

    return max(dp1[v.length-2], dp2[v.length-2]);
End
```

The proof is similar to problem 1, each sub-problem takes O(n), therefore its time complexity is O(n).

Problem-4

Solution

We define an array dp, where dp[i] is the total ways to decode the number sequence until the i-th digit. We suppose the number sequence is valid and it is marked by s. If s[i]='0',then s[i-1] must be '1' or '2',dp[i]=dp[i-2], which means we can get the sequence by appending s[i-1]s[i] at the end of dp[i-2] ways of decoding sequence, so the total decoding ways is dp[i-2]. If '3'<=s[i-1]<='9', then s[i] cannot combine with s[i-1], so the total decoding ways is dp[i-1]. If (s[i-1]='1'&'1'<=s[i]<='9') or (s[i-1]='2'&'1'<=s[i]<='6'), that means s[i-1] and s[i] can combine with each other; if s[i-1] combine with s[i], we have dp[i-2] ways to decode the sequence, if we split s[i-1] and s[i], we have dp[i-1] ways to decode it, so dp[i]=dp[i-1]+dp[i-2]. Therefore, we can get the following **DP equation:**(we assume that **all the input is valid**)

$$dp[i] = \begin{cases} dp[i-2], if \ \ (s[i] = '0') \,\&\, (s[i-1] = '1' \text{ or } '2') \\ dp[i-1], if \ \ ('3' <= s[i-1] <= '9') \\ dp[i-1] + dp[i-2], if \ \ (\text{s[i-1]='1' \& '1'<=s[i]<='9') or (s[i-1]='2' \& '1'<=}s[i] <= '6') \end{cases}$$

The pseudo code of the algorithm is shown below:

Algorithm: GetDecodingWays (char s[])

Input: An character sequence s which consists of digits

Output: the total ways to decode the sequence

```
Begin
    int dp[ v.length ];
```

```
    dp[0] = s[0] > ‘0’? 1:0;
    int k = s[0] > '0' && s[1] > '0'? 1:0;
    dp[1] = k + (s[0] == '1' || s[0] == '2' && s[1] <= '6' ? 1:0);
    for( int i = 2; i < s.length; ++i )
    {
       if(s[i] > ‘0’)
          dp[i] += dp[i-1];
       if( s[i-1]==’1’ || (s[i-1]==’2’ && s[i] <= ‘6’) )
          dp[i] += dp[i-2];
    }
    return dp[ s.length - 1];
End
```

Proof

The method to get the total ways to decode the sequence is very similar to Fibonacci sequence. Firstly, if the input is valid, then dp[0]=1; then dp[1] can be 1 or 2 under the condition that the input is also valid. To get dp[i], we can follow this rules. If s[i]=’0’,then s[i-1] must be ‘1’ or ‘2’,dp[i]=dp[i-2], which means we can get the sequence by appending s[i-1]s[i] at the end of dp[i-2] ways of decoding sequence, so the total decoding ways is dp[i-2]. If ‘3’<=s[i-1]<=’9’, then s[i] cannot combine with s[i-1], so the total decoding ways is dp[i-1]. If (s[i-1]=’1’&’1’<=s[i]<=’9’) or (s[i-1]=’2’&’1’<=s[i]<=’6’), that means s[i-1] and s[i] can combine with each other; if s[i-1] combine with s[i], we have dp[i-2] ways to decode the sequence, if we split s[i-1] and s[i], we have dp[i-1] ways to decode it, so dp[i]=dp[i-1]+dp[i-2].

If we have a better way to decode the sequence, we mark it by dp[i]’, according to definition of dp[i], dp[i]=dp[i]’.

Time Complexity

It takes one pass loop to get the answer, therefore its time complexity is O(n).

Problem-7

Solution

To determine a strictly increasing subsequence, we can take use of the idea of dynamic programming. We define an array nums to store the input sequence, and an array dp that means the length of best sequence at the position i, then we can get the following dp equation:

$$dp[i] = \begin{cases} 1, initial\ value \\ \max\left\{dp[i], dp[j]+1\right\}, where \text{ j<i, and nums[i]>nums[j]} \end{cases}$$

Based on the above equation, it is easy for us to implement the algorithm. Beside, to get the subsequence, we need some extra variables to record relevant position.

We define the input format like this: the first line int an integer n, which means the length of the input sequence, the second line follows n integers represents nums[i].

Following is the implement:

```
#include<iostream>
#include<algorithm>
#include<vector>
#include<cstring>
using namespace std;
const int max_n = 0x1fff;
int nums[max_n];//the input sequence
int maxIndex = 0, curMax = 1;//current maximum length
int dp[max_n];//dp array
int previous[max_n];//extra array to record the backward index
int n;//the length of input sequence
int ans[max_n];//store the answer;
void solve()
{
    //initialize dp[0] to 1, it means we only get nums[0]
    dp[0] = 1;
    previous[0] = -1;
    for(int i=0;i<n;++i)
    {
        dp[i] = 1;
    previous[i] = -1;
    }
    for (int i = 1; i < n; ++i)
    {
        //initialization
        //dp[i] = 1;
        //previous[i] = -1;
        for (int j = 0; j<i; ++j)
        {
            //dp equation
            if (nums[i]>nums[j]&&dp[i]<dp[j]+1)
            {
                dp[i] = dp[j] + 1;
                if (dp[i]>=curMax)
                {
```

```
                    //update current max index
                    curMax = dp[i];
                    previous[i] = j;
                    maxIndex = i;
                }
            }
        }
    }
    for(int i=0;i<n;++i)
    {
        cout<<dp[i]<<" ";
    }
    cout<<endl;
    for(int i=0;i<n;++i)
    {
        cout<<previous[i]<<" ";
    }
    cout<<endl;
    //output the answer
    //cout<<curMax<<endl;
    //backward to get the sequence
    ans[0] = nums[maxIndex];
    int index = previous[maxIndex];
    int i = 1;
    while (index != -1)
    {
        ans[i] = nums[index];
        index = previous[index];
        ++i;
    }
    for (int j = i - 1; j >= 0; --j)
        cout << ans[j] << " ";
    cout << endl;
    //clear the buff
    memset(dp, 0, max_n);
    memset(previous,-1,max_n);
    memset(ans,0,max_n);
}
int main()
{
    freopen("in.txt", "r", stdin);
    //freopen("out.txt","w",stdout);
    while (cin >> n)
    {
```

```
        //get the input
        for (int i = 0; i < n; ++i)
            cin >> nums[i];
        solve();
    }
    fclose(stdin);
    //fclose(stdout);
    return 0;
}
```