091M4041H - Assignment 2
Algorithm Design and Analysis

Song Qige 2017E8018661044

2017 年 10 月 31 日

1 decode(4)

1.1 optimal substructure and DP equation

$$OPT[i] = \begin{cases} OPT[i-1] & msg[i]\ can't\ decode\ with\ msg[i-1] \\ OPT[i-2] + OPT[i-1] & msg[i]\ can\ decode\ with\ msg[i-1] \end{cases}$$

OPT[i] means the number of ways to decode from msg[1] to msg[i]
$0 < i < the\ length\ of\ the\ message$.

1.2 algorithm describe and pseudo-code

when we decide how many ways to decode 1 to i,there is two possible situations.Fisrt is msg[i] can only decode by itself,then the ways to decode 1 to i equal to the way to decode 1 to i-1.The other situation is msg[i] can decode with msg[i-1],which means the combination of msg[i-1] and msg[i] is in the range of 1 to 26,then the ways to decode 1 to i is equal to the way to decode 1 to i-1 plus the way to decode 1 to i-2.

```
function WaysToDecode(msg[], len)
    OPT[1] = 1
    if msg[2] can't decode with msg[1] then
        OPT[2] = 1
    else
        OPT[2] = 2
    end if
    for i = 3 → len do
        if msg[i] can't decode with msg[i − 1] then
            OPT[i] = OPT[i − 1]
        else OPT[i] = OPT[i − 1] + OPT[i − 2]
        end if
    end for
    return OPT[len]
end function
```

1.3 Prove the correctness

Suppose the ways to decode msg from 1 to i is OPT[i],it's obvious that OPT[1] = 1,when calculating OPT[2],we have to transform msg[1] connect with msg[2] to a number,if it's in the range of 1 to 26,OPT[2] = 2 because we can decode msg from 1 to 2 by separately decode msg[1] and msg[2] or decode by the number msg[1] connect with msg[2].When i is bigger than 3,we calculate OPT[i] by the value of OPT[1] to OPT[i-1],if the msg[i] can't decode with msg[i-1],the ways to decode the string won't increase so OP-T[i] = OPT[i-1],but if msg[i] can decode with msg[i-1],the result should plus OPT[i-2] to contain the situation which treats msg[i] and msg[i-1] as a w-hole to decode.

1.4 time complexity

$T(n) = T(n-1) + c = O(n)$

2 Largest Divisible Subset(1)

2.1 optimal substructure and DP equation

$$OPT[i] = max \begin{cases} OPT[i] \\ \\ OPT[j] + 1 \qquad 0 < j < i < len \end{cases}$$

2.2 Algorithm Description and pseudo-code

To reduce the number of subproblems,we can sort the elements in the set by descending order.Suppose OPT[i] means the number of the elements of largest divisible subset from 0 to i,when calculating OPT[i],we have to check ever j smaller than i, if nums[j]%nums[i]=0,all elements that can divisible nums[j] surely can also divisible nums[i] and the ith element can join into the subset ending by jth element.And if OPT[j]+1 is larger than OPT[i],we should instead OPT[i] by OPT[j]+1.Then we get the optimal structure:
$OPT[i] = max\{OPT[i], OPT[j] + 1\}$.
To get the elements of the largest divisible subset,we need an array to store the index of the elements in largest divisible subset for backtracking after this process.

```
function LargeDivisibleSubset(nums[], len)
    state = 0
    sort(nums)
    for i = 0 → len do
        part[i] = i
    end for
    for i = 0 → len do
        for j = 0 → i do
```

```
            if nums[j]%nums[i] == 0 then
                OPT[i] = max{OPT[i], OPT[j] + 1}
                state = 1
            end if
            if state == 1 then
                part[i] = j
            end if
        end for
    end for
    k = 0
    for i = 0 → len do
        if part[i]! = i then
            res[k + +] = nums[part[i]]
        end if
    end for
    return res
end function
```

2.3 Prove the correctness

After sorting the set by descending order,we can solve this problem by considering the ith element.For each j in range of 0 to i-1,when we find a jth element can divisible the ith element,the ith element should join into the subset ending by jth element,and OPT[i] should be replaced by the length of this subset (OPT[j]) plus 1 if it's smaller than it,since the ith element has joined in this subset.

2.4 time complexity

Since we sort the set before starting dp process,the number of subproblems reduce to n^2 so $T(n) = O(n^2)$

3 Money robbing(2)

3.1 2-1 optimal substructure and DP equation

$$OPT[i] = max \begin{cases} OPT[i-1] \\ \\ OPT[i-2] + Money[i] \end{cases}$$

OPT[i] means the maximum amount of money we can rob after meeting the ith house.Money[i] means the amount of money ith house stashed.
$0 \ < \ i \ < the\ total\ number\ of\ houses\ along\ the\ street.$

3.2 2-1 Algorithm Description and pseudo-code

When solving the problem of robbing along a street,we should decide whether the ith house should be robbed.There are two situations,the first one is robbing the ith house simultaneously.To avoid alerting the police,we can't rob the i-1th house,and the money we get should plus Money[i].The other one is not robbing the ith house.Then the money we get at ith house won't increase. We can get the optimal structure:$OPT[i] = max\{OPT[i-1], OPT[i-2] + Money[i]\}$.

```
function RobAlongStreet(Money[], n)
    OPT[0] = 0
    OPT[1] = Money[1]
    for i = 2 → n do
        OPT[i] = max{OPT[i − 1], OPT[i − 2] + Money[i]}
    end for
    return OPT[n]
end function
```

3.3 2-1 Prove the correctness

For ith house,if we rob it,we can get money[i],and it means we didn't rob i-1th house,so OPT[i] = OPT[i-2] + Money[i].If we don't rob ith house,the money we get is equal to the money we get from 1 to i-1,so OPT[i] = OPT[i-1].

3.4 2-1 time complexity

$T(n) = T(n-1) + c = O(n)$

3.5 2-2 Algorithm Description and pseudo-code

When all houses are arranged in a circle,the difference is that the first house and the last house become adjacent so they can't be robbed together.So we can separate the problem into robbing the first house and not robbing the first house.Both subproblem can be solved same as robbing the houses along the streets.

function ROBFIRSTHOUSE($Money[], n$)
    $OPT[0] = 0$
    $OPT[1] = Money[1]$
    for $i = 2 \rightarrow n$ **do**
        $OPT[i] = max\{OPT[i-1], OPT[i-2] + Money[i]\}$
    end for
    return $OPT[n]$
end function
function NOTROBFIRSTHOUSE($Money[], n$)
    $OPT[1] = 0$
    $OPT[2] = Money[2]$

```
    for i = 3 → n do
        OPT[i] = max{OPT[i − 1], OPT[i − 2] + Money[i]}
    end for
    return OPT[n]
end function
function RobCircleHouse(Money[], n)
    return
max{RobFirstHouse(Money[], n−1), NotRobFirstHouse(Money[], n)}
end function
```

3.6 2-2 Prove the correctness

The second problem can be separated into two situations both similar to the first problem,so we can use the same way to calculate maximum money.When robbing the fist house,OPT[1] = Money[1],and when not robbing the first house,OPT[1] = 0,OPT[2] = Money[2].The optimal structure of both situation is $OPT[i] = max\{OPT[i-1], OPT[i-2] + Money[i]\}$.Then we compare the result of the two situations,choose the bigger one as the maximum amount of money we can rob.

3.7 2-2 time complexity

$T(n) = 2T(n-1) + c = O(n)$

4 Maximum length(7)

4.1 Source code

```
#include <iostream>

using namespace std;

```

```
const int N = 1000;
//store the given sequence
double a[N];
//opt[i] means the LIS length from a0 to ai
double opt[N];
//for LIS backtracking
int path[N];

int max_a;//the length of LIS
int max_b;//the index of the last element of LIS
void LongestIncreasingSub(int n)
{
    int i, j;
    for(i = 0; i < n; i++)
        opt[i] = 1;
    //opt[i] = max{opt[i],opt[j] + 1}
    for(i = 1;i < n;i++)
    {
     for(j = 0;j < i;j++)
     {
         if(a[i] > a[j] && opt[j] + 1 > opt[i])
         {
             opt[i] = opt[j] + 1;
             //the last step tp i is j
             path[i] = j;
         }
     }
     }
    for(i = 0;i < n;i++)
    {
```

```
        if(max_a < opt[i]){
            max_a = opt[i];
            max_b = i;
        }
    }
}

int main()
{
    int n;
    cin>>n;//the length of input sequence
    int i,j;
    for(i = 0; i < n; i++)
            cin>>a[i];
    LongestIncreasingSub(n);
    cout<<"the maximum length of the longest";
    cout<<" increasing subsequence is:";
    cout<<max_a<<endl;
    cout<<"the longest increasing subsequence:";

    //LIS backtracking
    for(j = 0,i = max_b;i >= 0;){
        opt[max_a - 1 - j] = a[i];
        i = path[i];
        j++;
        if(j == max_a) break;
    }
    for(i = 0;i < max_a;i++)
        cout<<opt[i]<<' ';

```

```
    cout<<endl;
    return 0;
}
```

4.2 result