

DDA 6050: Homework #2

Due on Oct 28, 2021

Professor Yixiang Fang

Haoyu Kang

Problem 1

LCS

The state transition equation is as below:

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1, & x[i] = y[j] \\ \max(c[i-1, j], c[i, j-1]), & \text{otherwise.} \end{cases} \quad (1)$$

For this question, we adopt method of state compression instead of a traditional approach that takes $O(n^2)$ memory. In details, we use **scrolling array** which is a one-dimension array to save each states. For each inside loop, $dp[j]$ represents length of LCS between i-length of prefix A and the j-length of prefix B. The variable *temp* will save the value of LCS between i-1 prefix of A and the j-1 prefix of B.

Hence the optimal method of LCS just takes $O(n)$ space complexity and $O(n^2)$ time complexity.

The cpp code is attached as follow:

```

1  #include<iostream>
2  #include<vector>
3  #include<cstdio>
4  using namespace std;
5  int main(){
6      int num;
7      vector<int> seq1, seq2;
8      while( scanf( "%d", &num)) {
9          seq1.push_back(num);
10         if( cin.get() == '\n' ) break;
11     }
12     while( scanf( "%d", &num)) {
13         seq2.push_back(num);
14         if( cin.get() == '\n' ) break;
15     }
16     int n=seq1.size();
17     vector<int> dp(n+1, 0);
18     int pre;
19     for( int i=1; i<n+1; i++){
20         for( int j=1; j<n+1; j++){
21             int temp=dp[j];
22             if( i==1 && j==1 ) dp[j]=seq1[i-1]==seq2[j-1]?1:0;
23             else if( i==1 ) dp[j]=seq1[i-1]==seq2[j-1]?1:dp[j-1];
24             else if( j==1 ) dp[j]=seq1[i-1]==seq2[j-1]?1:dp[j];
25             else {
26                 if( seq1[i-1]==seq2[j-1] ) dp[j]=pre+1;
27                 else dp[j]=max(dp[j-1], dp[j]);
28             }
29             pre=temp;
30         }
31     }
32     cout<<dp[n]<<endl;
33 }
```

Problem 2

0-1 Knapsack Problem

The state transition equation is as below:

$$B[k, w] = \begin{cases} B[k-1, w] + 1, & w_k > w \\ \max(B[k-1, w], B[k-1, w_k] + b_k), & \text{else.} \end{cases} \quad (2)$$

We also adopt method of state compression instead of a traditional approach that takes $O(n^2)$ memory. In details, we use **scrolling array** which is a one-dimension array to save each states. For each inside loop, $dp[j]$ represents maximum value the bag is able to attain if the volume of bag is limited within j respected to subset of S_i . Different from the problem 1, in this question we create additional array to save states of the last once loop. e.g. $pre[j - weight[i-1]]$ represents maximum value the bag is able to attain if the volume of bag is limited within $j - weight[i-1]$ respected to subset of S_{i-1} .

Hence the optimal method just takes $O(n)$ space complexity and $O(n^2)$ time complexity.

The cpp code is attached as follow:

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4  int main(){
5      int n,w;
6      cin>>n>>w;
7      vector<int> weight,value;
8      int w_i,v_i;
9      for(int i=0;i<n;i++){
10         cin>>w_i>>v_i;
11         weight.push_back(w_i);
12         value.push_back(v_i);
13     }
14     vector<int> dp(w+1,0);
15     vector<int> pre(w+1,0);
16     for(int i=1;i<n+1;i++){
17         if(i==n){
18             if(weight[i-1]<=w)
19                 dp[w]=max(dp[w],pre[w-weight[i-1]]+value[i-1]);
20             break;
21         }
22         for(int j=1;j<w+1;j++){
23             if(weight[i-1]<=j)
24                 dp[j]=max(dp[j],pre[j-weight[i-1]]+value[i-1]);
25         }
26         pre=dp;
27     }
28     cout<< dp[w] <<endl;
29 }
```

Problem 3

The Shortest Path Problem

The state transition equation is as below:

$$dst[u] = \max \{dst[u], dst[v] + w\} \quad (3)$$

Since the question description tells that it contains negative edge, we adopt Bellmanford to solve this question. We use two-dimension array *edges* to save edge with weight, and one-dimension array *dst* to represent the shortest path from start point to each station. Bellmanford claims that the shortest path will not cover than $n-1$ edges. Therefore in the i -th loop, $dst[u]$ represent the shortest distances to u if only go through i edges.

Hence the optimal method just takes $O(n^2)$ space complexity and $O(n^2)$ time complexity.

The cpp code is attached as follow:

```

1  #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<climits>
5  using namespace std;
6  int main(){
7      int n,m,s,t;
8      cin>>n>>m>>s>>t;
9      vector<vector<int>>> edges;
10     for (int i=0;i<m;i++){
11         int u,v,w;
12         cin>>u>>v>>w;
13         edges.push_back({u,v,w});
14     }
15     vector<int> dst(n+1,INT_MAX);
16     dst[s]=0;
17     for (int i=0;i<n-1;i++){
18         int flag=0;
19         for (const auto& e:edges){
20             int u=e[0];
21             int v=e[1];
22             int w=e[2];
23             if (dst[u]!=INT_MAX && dst[v]>dst[u]+w){
24                 dst[v]=dst[u]+w;
25                 flag=1;
26             }
27         }
28         if (!flag) break;
29     }
30     cout<<dst[t]<<endl;
31 }

```

Problem 4

LIS

Different from what we learn in class, I optimize the algorithm with less memory and time cost. In this algorithm, we just create one-dimensional array to save increasing sequence. In the i -th loop, if $nums[i] > \text{back of } dp$ put the $nums[i]$ into the end of the array. Otherwise find the one which is larger than $nums[i]$ from left, and replace it.

Hence the optimal method just takes $O(n)$ space complexity and $O(n \log n)$ time complexity.

The cpp code is attached as follow:

```

1  #include<iostream>
2  #include<vector>
3  using namespace std;
4  int main(){
5      int n;
6      cin>>n;
7      vector<int> nums;
8      int num;
9      for (int i=0;i<n;i++) {
10         cin>>num;
11         nums.push_back(num);
12     }
13     if (n <= 1) return n;
14     vector<int> dp;
15     dp.push_back(nums[0]);
16     for (int i = 1; i < n; ++i) {
17         if (dp.back() < nums[i]) {
18             dp.push_back(nums[i]);
19         } else {
20             /*      nums[i]      */
21             auto itr = lower_bound(dp.begin(), dp.end(), nums[i]);
22             *itr = nums[i];
23         }
24     }
25     cout<<(int)dp.size()<<endl;
26 }
```

Problem 5

Max M Sum Subsequences Problem

The state transition equation is as below:

$$DP[i, j] = \begin{cases} mk[i-1, j-1] + nums[j], & i == j \\ \max(DP[i, j-1] + nums[j], mk[i-1, j-1] + nums[j]), & \text{else.} \end{cases} \quad (4)$$

In above equation, $DP[i, j]$ represents max i sum subsequences of j prefix of sequence when the j -th number is in the i -th subsequence, and $mk[i, j]$ represents max i sum subsequences of j prefix of the sequence. In order to compress states, we also adopt scrolling arrays to replace above two-dimensional arrays.

Hence the optimal method just takes $O(n)$ space complexity and $O(n \log n)$ time complexity.

The cpp code is attached as follow:

```

1  #include<iostream>
2  #include<vector>
3  #include<cmath>
4  #include<cstring>
5  using namespace std;
6  int main(){
7      int n,m;
8      cin>>n>>m;
9      int dp[n];
10     int nums[n];
11     // vector<int> nums(n,0);
12     // vector<int> dp(n,0);
13     for (int i=0;i<n;i++){
14         int num;
15         cin>>num;
16         nums[i]=num;
17     }
18     int res;
19     int max_sum;
20     // vector<int> mk(n,0);
21     int mk[n];
22     memset(dp,0,sizeof(int)*n);
23     memset(mk,0,sizeof(int)*n);
24     for (int i=0;i<m;i++){
25         max_sum=INT_MIN;
26         for (int j=i;j<n;j++){
27             if (j==i) {
28                 if (j==0) dp[j]=nums[j];
29                 else dp[j]=mk[j-1]+nums[j];
30             }
31             else {
32                 dp[j]=max(dp[j-1]+nums[j],mk[j-1]+nums[j]);
33                 mk[j-1]=max_sum;
34             }
35             max_sum=max(max_sum,dp[j]);
36         }
37     }
38     cout<<max_sum<<endl;
39
40 }
```