DVP 6.1

we can just use the thought of DP,the $maxvalue[s_j] = max(maxvlue[s_(j-1)] + a[j], a[j])$, so the max sum of the subsequence is $max(s_j)$

```
int maxSeq(vector<int>a[])
{
    int maxvalue = 0, tmpvalue = 0;
    for (int i = 0; i < a.size(); i++)
    {
        tmpvalue = tmpvalue + a[i];
        if (tmpvalue < 0) tmpvalue = 0;
        if (maxvalue < tmpvalue) maxvalue = tmpvalue;
    }
    return maxvalue;
}
```

DVP 6.2

we can use p_i to represent the minimize value of the penalty until this point, so the $p_j = min(p_i + (200 - (a_i - a_j))^2)$ · When we check all the hotel on the way, we can get the minimize of the penalty

```
int mindistance(vector<int>a[])
{
    vector<int>dp(0,a.size());
    dp[0]=0;

    for(int i=0;i<a.size()-1;i++){
        for(int j=i+1;j<a.size();j++){
            int x = hotels[j]-hotels[i];
            if(x<=200) dp[j] = min(dp[j],dp[i]+Pow((200-x),2);
        }
    }
    return dp[i-1]
}
```

DVP 6.5

1. considering for an isolated col, we can place 1 or 2 pebbles on one col. we can assume that the index of an isolated col is {1,2,3,4}

Solutions: {1},{2},{3},{4},{1,3},{1,4},{2,4}

1. Now we have a compare function C(i,j) to compare that if type i and type j is compatible. we can get a s(i,t) was the optimal solution of the final col when the previous i cols is all have stone. v(i,t) was the value of col i which was put the stone followed type T. So we have $s(i,t) = max_{c(t,s)=true}(s(i-1,s) + v(i,t))$. Finally, we can just return

$max_{0<=t<8}(s(n,t))$.

DVP6.24

1. just like the figure 6.4 , when we need to calculate and compute the value of the edit distance, we only need to consider three status which is insert, delete and replace so if we want to compute one grid in the 2-way matrix, we only need to use the right grid, bottom grid and lower right grid. To sum up, we only need to store two row(the current status and previous status) on the memory to compute the value of distance, so we can use a “sliding array” to make the space compliexity to O(n)

2. we can just use m to represent row, and n to represent col. so we can just use a sliding array which have a col size of $n/2 + 1$ to store the current row and previous row and add such code into edit distance

```
K(i,0) = i
if j > n/2
    if E(i,j) == E(i-1,j) + 1
        K(i,j-n/2) = K(i-1,j-n/2)
    else if E(i,j) == E(i,j-1)+1
        K(i,j-n/2) = K(i,j-n/2-1)
    else if E(i,j) == E(i-1,j-1) + diff(i,j)
        K(i,j-n/2) = K(i-1,j-n/2-1)
```

3. the time conpliexity is $T(m,n) = O(mn) + 0.5 * O(mn) + 0.25 * O(mn) + \cdots$ which equal to O(mn)

CLRS 15.1

Since directed acyclic graphs can be linearized, we can sort it in topological order. And then the problem can be solved by Dynamic Programming.

LongestPath (G)
- Input: weight Dag
- Output: Largest Path Cost
Topologically sort G
For each vertex $v \in V$ in linearized order
    do dist(v) = $max_{(u,v)\in E}\{dist(u) + weight(u, v)\}$
return $max_{v\in V}\{dist(v)\}$

The above algorithm describes the subproblems as: The largest path cost from s to v is the maximum of the predecessor u's longest path cost plus the step cost from u to v. Topologically sorting the graph takes $O(V + E)$ time. Then the algorithm updates each vertice's longest path cost by running a loop on all of the adjacent vertices. And there are $O(E)$ number of vertices in the graph. So the inner loop is executed $O(V + E)$ times. Therefore, the complexity of the algorithm is $O(V + E)$;

CLRS 15.3

- First, sorting the input points by x-coordinate.
- Then, we defined subproblem min_dist(i, j).

$$i- > \ldots - > 1- > \ldots - > j$$

  This min_dist is the same with min_dist(j, i). Only different in direction. But from 1->i and j->1 same as 1->j, i->1.

- So, there are different transition fucntions:
  - When i == 1 and j == 2, min_dist(i,j) = cost_(i, j) = cost(1,2).

    In this case, back and forth has the same path.1->2 and 2->1, we cannot touch 3 because the property of bitonic problem.

  - When i < j - 1, min_dist(i, j) = min_dist(i, j - 1) + cost(j-1,j)

    In this case, when k = j - 1 has the min cost(k,j). So, we can put j - 1 to the 1->j path directly.

  - When i = j - 1

$$min_{1<=k<i}(l(k,i)+dist(k,j))$$

In this case, we have to get from i to 1, but didn't know how to back. Then for every 1<=k<i, we check wheter the l(k,i)(which is computed before) + dist(k,j) is smaller.

- Complexity:
  - sort the array is $O(nlogn)$.
  - For every i we need to calculate every j one time. which is $O(n^2)$
  - For i == j-1, need $O(n)$ time to check every k, so plus $O(n)$, which is dominated by $O(n^2)$.
  - So, the total complexity is $O(n)$.
  - spcace complexity is $O(n^2)$