Dynamic Programming

Rossano Venturini[1]

1 **Computer Science Department, University of Pisa, Italy**
rossano.venturini@unipi.it

Notes for the course *"Competitive Programming and Contests"* at Department of Computer Science, University of Pisa.
Web page: `https://github.com/rossanoventurini/CompetitiveProgramming`

These notes sketch the content of 16th and 17th lectures.

1 Minimum Cost Path

▶ **Problem 1** (Minimum Cost Path). *We are given a matrix M of $n \times m$ integers. The goal is to find the minimum cost path to move from the top-left corner to the bottom-right corner by moving only down or to right.*

Consider the matrix below.

1	2	6	9
0	0	3	1
1	7	7	2

It is easy to see that the following recurrence solves the problem.

$$W(i,j) = M[i,j] + \begin{cases} 0 & \text{if } i = j = 1 \\ W(i, j-1) & \text{if } i = 1 \text{ and } j > 1 \\ W(i-1, j) & \text{if } i > 1 \text{ and } j = 1 \\ \min(W(i-1,j), W(i,j-1)) & \text{otherwise} \end{cases}$$

Thus, the problem is solved by filling a matrix W. Matrix W for the example above is as follows.

1	3	9	18
1	1	4	5
2	8	11	7

2 Longest Bitonic Subsequence

▶ **Problem 2** (Longest Bitonic Subsequence). *Given a sequence $S[1,n]$, find the length of its longest bitonic subsequence.*

A bitonic sequence is a sequence that first increases and then decreases.

Consider for example the sequence $S = 2, -1, 4, 3, 5, -1, 3, 2$. The subsequence $-1, 4, 3, -1$ is bitonic. A longest bitonic subsequence is $2, 3, 5, 3, 2$.

The idea is to compute the longest increasing subsequence from left to right and the longest increasing subsequence from right to left by using the $\Theta(n \log n)$ time solution of the previous lecture.

Then, we combine these two solutions to find the longest bitonic subsequence. This is done by taking the values in a column, adding them and subtracting one. This is correct because LIS$[i]$ computes the longest increasing subsequence of $S[1, i]$ and ends with $S[i]$.

S	2	-1	4	3	5	-1	3	2
LIS	1	1	2	2	3	1	2	2
LDS	2	1	3	2	3	1	2	1
LBS	2	1	4	3	5	1	3	2

3 Subset sum

▶ **Problem 3** (Subset sum). *Given a set S of n non-negative integers, and a value v, determine if there is a subset of the given set with sum equal to given v.*

The problem has a solution which is almost the same as 0/1 knapsack problem.

As in the 0/1 knapsack problem, we construct a matrix W with $n + 1$ rows and $v + 1$ columns. Here the matrix contains booleans.

Entry $W[i][j]$ is true if and only if there exists a subset of the first i items with sum j.

The entries of the first row $W[0][]$ are set to false while entries of the first column $W[][0]$ are set to true.

Entry $W[i][j]$ is true either if $W[i-1][j]$ is true or $W[i-1][j-S[i]]$ is true.

As an example, consider the set $S = \{3, 2, 5, 1\}$ and value $v = 6$. Below the matrix W for this example.

	0	1	2	3	4	5	6
$\emptyset$	T	F	F	F	F	F	F
3	T	F	F	T	F	F	F
2	T	F	T	T	F	T	F
5	T	F	T	T	F	T	F
1	T	T	T	T	T	T	T

4 Coin change

▶ **Problem 4** (Coin change). *We have n types of coins available in infinite quantities where the value of each coin is given in the array $C = [c_1, c_2, \ldots, c_n]$. The goal is find out how many ways we can make the change of the amount K using the coins given.*

For example, if $C = [1, 2, 3, 8]$, there are 3 ways to make $K = 3$, i.e., 1, 1, 1, 1, 2 and 3.

The solution is similar to the one for subset sum.

The goal is to build a $(n + 1) \times (K + 1)$ matrix W, i.e., we compute the number of ways to change any amount smaller than or equal to K by using coins in any prefix of C.

The easy cases are when $K = 0$. We have just one way to change this amount. Thus, the first column of W contains 1 but $W[0,0] = 0$ which is the number of ways to change 0 with no coin.

Now, for every coin we have an option to include it in the solution or exclude it.

If we decide to include the ith coin, we reduce the amount by coin value and use the subproblem solution $(K - c[i])$.

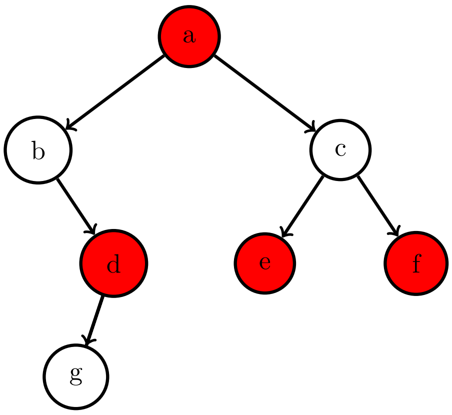
If we decide to exclude the ith coin, the solution for the same amount without considering that coin is on the entry above.

5 Largest independent set on trees

▶ **Problem 5** (Largest independent sets on trees)**.** *Given a tree T with n nodes, find one of its largest independent sets.*

An independent set is a set of nodes I such that there is no edge connecting any pair of nodes in I.

Example below shows a tree whose largest independent set consists of the red nodes a, d, e, and f.



In general the largest independent set is not unique. For example, we can obtain a different largest independent set by replacing nodes a and d with nodes b and g.

Consider a bottom up traversal of the tree T. For any node u, we have two possibilities: either add or not add the node u to the independent set.

In the former case, u's children cannot be part of the independent set but its grandchildren could. In the latter case, u's children could be part of the independent set.

Let $\mathsf{LIST}(u)$ be the size of the independent set of the subtree rooted at u and let C_u and G_u be the set of children and the set of grandchildren of u, respectively.

Thus, we have the following recurrence.

$$\mathsf{LIST}(u) = \begin{cases} 1 & \text{if u is a leaf} \\ \max(1 + \sum_{v \in G_u} \mathsf{LIST}(v), \sum_{v \in C_u} \mathsf{LIST}(v)) & \text{otherwise} \end{cases}$$

The problem is, thus, solved with a post-order visit of T in linear time.

Observe that the same problem on general graphs is NP-Hard.

6 Edit distance

▶ **Problem 6** (Edit distance). *Given two strings $S_1[1, n]$ and $S_2[1, m]$, find the minimum number of* edit operations *required to transform S_1 into S_2. There are three edit operations:*
- *Insert a symbol at a given position*
- *Replace a symbol at a given position*
- *Delete a symbol at a given position*

For example, if the two strings are $S_1 =$ `hello` and $S_2 =$ `hallo`, the edit distance is 1 as we can replace symbol `e` with `a` in second positions of S_1.

Let $\mathsf{ED}(i, j)$ be the edit distance of prefixes $S_1[1, i]$ and $S_2[1, j]$. The idea is to compute $\mathsf{ED}(i, j)$ by using shorter prefixes as subproblems. Ideed, if $S[i] = S[j]$, $\mathsf{ED}(i, j) = \mathsf{ED}(i - 1, j - 1)$ We can compute $\mathsf{ED}(i, j)$

Thus, the recurrence is as follows.

$$\mathsf{ED}(i, j) = \begin{cases} i & \text{if j = 0} \\ j & \text{if i = 0} \\ \mathsf{ED}(i-1, j-1) & \text{if S[i] = S[j]} \\ 1 + \min(\mathsf{ED}(i, j-1), \mathsf{ED}(i-1, j), \mathsf{ED}(i-1, j-1)) & \text{otherwise} \end{cases}$$

Here we report the matrix obtained for the strings $S_1 =$ `agced` and $S_2 =$ `abcdef`.

		a	b	c	d	e	f
	0	1	2	3	4	5	6
a	1	0	1	2	3	4	5
g	2	1	1	2	3	4	5
c	3	2	2	1	2	3	4
e	4	3	3	2	2	2	3
d	5	4	4	3	2	3	3

7 Longest palindromic subsequence

▶ **Problem 7** (Longest palindromic subsequence). *Given a sequence $S[1, n]$, find the length of its longest palindromic subsequence.*

Given sequence is $S =$ `bbabcbcab`, then the output should be 7 as `babcbab` is the longest palindromic subsequence in it. `bbbbb` and `bbcbb` are also palindromic subsequences of S, but not the longest ones.

Let $\mathsf{LPS}(i, j)$ be the length of the longest palindromic subsequence of $S[i, j]$.

The idea is that of computing $\mathsf{LPS}(i, j)$ by using three subproblems. If $S[i]$ and $S[j]$ equal, then we can extend the longest palindromic subsequence of $S[i + 1, j - 1]$ by prepending and appending character $S[i]$. If this is not the case, then the longest palindromic subsequence is the longest palindromic subsequence of either $S[i + 1, j]$ or $S[i, j - 1]$.

Thus, the recurrence is as follows.

$$\mathsf{LPS}(i, j) = \begin{cases} 0 & \text{if i > j} \\ 1 & \text{if i = j} \\ 2 + \mathsf{LPS}(i+1, j-1) & \text{if S[i] = S[j]} \\ \max(\mathsf{LPS}(i+1, j), \mathsf{LPS}(i, j-1)) & \text{otherwise} \end{cases}$$

Here we report the matrix obtained for the example above. Observe that we have to fill the matrix starting from its diagonal

	b	b	a	b	c	b	c	a	b
b	1	2	2	3	3	5	5	5	7
b	0	1	1	3	3	3	3	5	7
a	0	0	1	1	1	3	3	5	5
b	0	0	0	1	1	3	3	3	5
c	0	0	0	0	1	1	3	3	3
b	0	0	0	0	0	1	1	1	3
c	0	0	0	0	0	0	1	1	1
a	0	0	0	0	0	0	0	1	1
b	0	0	0	0	0	0	0	0	1

8 Weighted job scheduling

▶ **Problem 8** (Weighted job scheduling)**.** *There are n jobs and three arrays $S[1 \ldots n]$ and $F[1 \ldots n]$ listing the start and finish times of each job, and $P[1 \ldots n]$ reporting the profit of completing each job. The task is to choose a subset $X \subseteq \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$, which maximizes the total profit.*

We already solved this problem (called activity selection) under the hypothesis that all the jobs have the same profit. For that easier problem, a greedy algorithm finds the optimal schedule in $\Theta(n \log n)$ time.

Consider the following example. Arrays are $S = [1, 2, 4, 6, 5, 7]$, $F = [3, 5, 6, 7, 8, 9]$, and $P = [5, 6, 5, 4, 11, 2]$.

The solution is as follows. We first sort the jobs by finish time.

Let $\mathsf{WJS}(i)$ be the maximum profit by selecting among the first i jobs (ordered by finish time).

The value of $\mathsf{WJS}(i)$ is computed by either excluding or including ith job. If we exclude ith job, then $\mathsf{WJS}(i) = \mathsf{WJS}(i-1)$. Instead, if it is included, then we have to find the largest index of j such that $F[j] \leq S[i]$ so that $\mathsf{WJS}(i) = \mathsf{WJS}(j) + P[i]$. Obviously, among these two possibilities, we take the one that give the largest profit.

We assume we have a fake job with index 0 whose finish time is $-\infty$. We also let $\mathsf{WJS}(0)$ to be 0. This serves as sentinel to avoid special cases.

Observe that sorting by finish time guarantees that if we current job i does not overlap with jth job, then it does not overlap with any other job with a smaller index. This means that we can safely use the value $\mathsf{WJS}(j)$.

Notice that the index j can be found in $\Theta(\log n)$ time with binary search. Thus, the algorithm runs in $\Theta(n \log n)$ time.

S	-	1	2	4	6	5	7
F	$-\infty$	3	5	6	7	8	9
P	0	5	6	5	4	11	2
R	0	5	6	10	14	17	17