ICCS313 - Algorithm Analysis, T1 2019/20

Lecture 8-9: Dynamic Programming

3, 9 November 2019

Lecturer: Chaya Hiruncharoenvate *Scribe: -*

```
* Go over Shortest Path algorithms
* Assignment 4 is out, due in 2 weeks
```

1 Dynamic Programming

Source: Competitive Programmers Handbook by Antti Laaksonen

Dynamic programming is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of *subproblems* and solving them one by one, smallest first, using the answers to small problems to help figure out solutions for larger subproblems, until the original problem is solved.

This sounds a lot like divide and conquer strategy. However, when subproblems are overlapped, we can be more efficient by not solving the same problem more than once. To do so, we need to "remember" solutions so that when we see the same subproblem again, we can just solve it from our "memory". This is the key characteristic of dynamic programming technique.

```
P[n] -> P[n-1] ---> P[n-2] -> ... -> P[0]
  |        \------> P[n-3] -> ... -> P[0]
  \---> P[n-2] ---> P[n-3] -> ... -> P[0]
           \------> P[n-4] -> ... -> P[0]
```

The main difference between divide-and-conquer and dynamic programming is how the organization of the subproblems. In dynamic programming, subproblems are organized into a DAG while, in divide-and-conquer, they are organized into a tree.

[Give an example of fibonacci sequence vs factorial]

There are two styles of dynamic programming: top-down and bottom-up.

1. Top-down: Say we want to solve $P[n]$ but to solve $P[n]$ we need to $P[n-1]$ and $P[n-2]$ recursively. Once we have a solution for $P[n-2]$, we write it down so that when the recursive call to solve for $P[n-1]$ we just return the solution from our note right away. The name 'top-down' comes from the fact that we start from the original $P[n]$ and work our way down to $P[0]$. The implementation of top-down DP is recursive.

2. Bottom-up: Again, say we want to solve $P[n]$. Instead of start from $P[n]$, we can build up solutions from the bottom, namely $P[0], P[1], P[2]$ and so on. The implementation of bottom-up DP is iterative.

2 Coin Change (Revisit)

Recall the coin change problem that we have already seen in a previous lecture. Given a set of coin values $coins = c_1, c_2, \ldots, c_k$ and a target sum of money n, our task is to form the sum n using as few coins as possible.

In the previous lecture, we solved the problem using a greedy algorithm that always chooses the largest possible coin. The greedy algorithm works, for example, when the coins are the euro coins, but in the general case the greedy algorithm does not necessarily produce an optimal solution.

Now is time to solve the problem efficiently using dynamic programming, so that the algorithm works for any coin set. The dynamic programming algorithm is based on a recursive function that goes through all possibilities how to form the sum, like a brute force algorithm. However, the dynamic programming algorithm is efficient because it uses "memoization" and calculates the answer to each subproblem only once.

Recursive formulation The idea in dynamic programming is to formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum x?

Let $solve(x)$ denote the minimum number of coins required for a sum x. The values of the function depend on the values of the coins. For example, if $coins = 1, 3, 4$, the first values of the function are as follows:

```
solve(0) = 0
solve(1) = 1
solve(2) = 2
solve(3) = 1
solve(4) = 1
solve(5) = 2
solve(6) = 2
solve(7) = 2
solve(8) = 2
solve(9) = 3
solve(10) = 3
```

For example, $solve(10) = 3$, because at least 3 coins are needed to form the sum 10. The optimal solution is $3 + 3 + 4 = 10$.

The essential property of $solve()$ is that its values can be recursively calculated from its smaller values. The idea is to focus on the first coin that we choose for the sum. For example, in the above scenario, the first coin can be either 1, 3 or 4. If we first choose coin 1, the remaining task is to form the sum 9 using the minimum number of coins, which is a subproblem of the original problem. Of course, the same applies to coins 3 and 4. Thus, we can use the following recursive formula to calculate the minimum number of coins:

```
solve(x) = min(solve(x-1)+1,
               solve(x-3)+1,
               solve(x-4)+1)
```

The base case of the recursion is $solve(0) = 0$, because no coins are needed to form an empty sum. For example, $solve(10) = solve(7) + 1 = solve(4) + 2 = solve(0) + 3 = 3$.

Now we are ready to give a general recursive function that calculates the minimum number of coins needed to form a sum x:

$$solve(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ min_{c \in coins} solve(x - c) + 1 & x > 0 \end{cases}$$

First, if $x < 0$, the value is ∞, because it is impossible to form a negative sum of money. Then, if $x = 0$, the value is 0, because no coins are needed to form an empty sum. Finally, if $x > 0$, the variable c goes through all possibilities how to choose the first coin of the sum.

Once a recursive function that solves the problem has been found, we can directly implement a solution as follows:

```
def solve(x):
    if x < 0: return INF
    if x == 0: return 0
    best = INF
    for c in coins:
        best = min(best, solve(x-c)+1)
    return best
```

Still, this function is not efficient, because there may be an exponential number of ways to construct the sum. However, next we will see how to make the function efficient using a technique called memoization.

Memoization The idea of dynamic programming is to use memoization to efficiently calculate values of a recursive function. This means that the values of the function are stored in an array or a lookup table after calculating them. For each parameter, the value of the function is calculated recursively only once, and after this, the value can be directly retrieved from the array or the lookup table.

In this problem, we use a dictionary or a hash table called "known" to keep track of which value of x that have already been computed.

```
def solve(x):
    if x < 0: return INF
    if x == 0: return 0
    if x in known: return known[x]
    best = INF
    for c in coins:
       best = min(best, solve(x-c)+1)
    known[x] = best
    return best
}
```

The function handles the base cases $x < 0$ and $x = 0$ as previously. Then the function checks from $known$ if $solve(x)$ has already been stored in $known[x]$, and if it is, the function directly returns it. Otherwise the function calculates the value of $solve(x)$ recursively and stores it in $known[x]$.

This function works efficiently, because the answer for each parameter x is calculated recursively only once. After a value of $solve(x)$ has been stored in $known[x]$, it can be efficiently retrieved whenever the function will be called again with the parameter x. The time complexity of the algorithm is $O(nk)$, where n is the target sum and k is the number of coins.

The solution presented above is considered top-down. However, the bottom-up version can be done as well.

```
value = [0]*(n+1)
```

```
for x in range(1,n+1):
    value[x] = INF
    for c in coins:
       if x-c >= 0:
           value[x] = min(value[x], value[x-c]+1)
```

Constructing a solution Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. In the coin problem, for example, we can declare another array that indicates for each sum of money the first coin in an optimal solution:

```
choice = [-1]*(n+1)
value = [0]*(n+1)
for x in range(1,n+1):
    value[x] = INF
    for c in coins:
       if x-c >= 0 and value[x] > value[x-c]+1:
           value[x] = value[x-c]+1
           choice[x] = c
```

We use $choice[x]$ to indicate which coin to choose when optimally changing value of x. So to construct the solution for sum n, we do:

```
solution = []
while n > 0:
    solution.append(choice[n])
    n -= choice[n]
```

Counting number of solutions Let us now consider another version of the coin problem where our task is to calculate the total number of ways to produce a sum x using the coins. For example, if $coins = 1, 3, 4$ and $x = 5$, there are a total of 6 ways:

```
1+1+1+1+1
1+1+3
1+3+1
3+1+1
1+4
4+1
```

Again, we can solve the problem recursively. Let $ways(x)$ denote the number of ways we can form the sum x. For example, if $coins = 1, 3, 4$, then $ways(5) = 6$ and the recursive formula is:

```
ways(x) = ways(x-1) + ways(x-3) + ways(x-4)
```

Then, the general recursive function is as follows:

$$ways(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in coins} ways(x-c) & x > 0 \end{cases}$$

If $x < 0$, the value is 0, because there are no solutions. If $x = 0$, the value is 1, because there is only one way to form an empty sum. Otherwise we calculate the sum of all values of the form $ways(xc)$ where c is in coins.

Rod cutting Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

```
length    | 1   2   3   4   5   6   7   8
--------------------------------------------
price     | 1   5   8   9  10  17  17  20
```

3 Longest Increasing Subsequence (LIS)

Here our problem is to find the longest increasing subsequence in an array of n elements. This is a maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, in the array:

```
index | 0 1 2 3 4 5 6 7
-------------------------
value | 6 2 5 1 7 4 8 3
```

The longest increasing subsequence contains 4 elements: `2 5 7 8`.

Let $length(k)$ denote the length of the longest increasing subsequence that ends at position k. Thus, if we calculate all values of $length(k)$ where $0 \leq k \leq n-1$, we will find out the length of the longest increasing subsequence. For example, the values of the function for the above array are as follows:

```
length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2
```

For example, $length(6) = 4$, because the longest increasing subsequence that ends at position 6 consists of 4 elements.

To calculate a value of $length(k)$, we should find a position $i < k$ for which $array[i] < array[k]$ and $length(i)$ is as large as possible. Then we know that $length(k) = length(i) + 1$, because this is an optimal way to add $array[k]$ to a subsequence. However, if there is no such position i, then $length(k) = 1$, which means that the subsequence only contains $array[k]$.

Since all values of the function can be calculated from its smaller values, we can use dynamic programming. In the following code, the values of the function will be stored in an array length.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
       if (array[i] < array[k]) {
           length[k] = max(length[k],length[i]+1);
```

```
            }
        }
}
```

Running time: $O(n^2)$
Question: How do you find the sequence itself?

4 Knapsack

The term knapsack refers to problems where a set of objects is given, and subsets with some properties have to be found. Knapsack problems can often be solved using dynamic programming.

In this section, we focus on the following problem: Given a list of weights $[w_1, w_2, \ldots, w_n]$, determine all sums that can be constructed using the weights. For example, if the weights are $[1, 3, 3, 5]$, the following sums are possible:

```
0  1  2  3  4  5  6  7  8  9 10 11 12
X  X  -  X  X  X  X  X  X  X  -  X  X
```

In this case, all sums between $0, \ldots, 12$ are possible, except 2 and 10. For example, the sum 7 is possible because we can select the weights $[1, 3, 3]$.

To solve the problem, we focus on subproblems where we only use the first k weights to construct sums. Let $possible(x, k) = true$ if we can construct a sum x using the first k weights, and otherwise $possible(x, k) = false$. The values of the function can be recursively calculated as follows:

$$possible(x, k) = possible(x - w_k, k - 1) \vee possible(x, k - 1)$$

The formula is based on the fact that we can either *use or not use* the weight w_k in the sum. If we use w_k, the remaining task is to form the sum $x - w_k$ using the first $k - 1$ weights, and if we do not use w_k, the remaining task is to form the sum x using the first $k - 1$ weights. As the base cases,

$$possible(x, 0) = \begin{cases} true & x = 0 \\ flase & x \neq 0 \end{cases}$$

because if no weights are used, we can only form the sum 0.

The following table shows all values of the function for the weights $[1, 3, 3, 5]$ (the symbol X indicates the true values):

```
  | 0  1  2  3  4  5  6  7  8  9 10 11 12
--------------------------------------------
0 | X
1 | X  X
2 | X  X     X  X
3 | X  X     X  X     X  X
4 | X  X     X  X  X  X  X  X  X     X  X
```

After calculating those values, $possible(x, n)$ tells us whether we can con- struct a sum x using all weights.

Let W denote the total sum of the weights. The following $O(nW)$ time dynamic programming solution corresponds to the recursive function:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

However, here is a better implementation that only uses a one-dimensional array $possible[x]$ that indicates whether we can construct a subset with sum x. The trick is to update the array from right to left for each new weight:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Note that the general idea presented here can be used in many knapsack problems. For example, if we are given objects with weights and values, we can determine for each weight sum the maximum value sum of a subset.

5 Edit Distance

The edit distance or Levenshtein distance is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

- insert a character (e.g. ABC $\to$ ABCA)
- remove a character (e.g. ABC $\to$ AC)
- modify a character (e.g. ABC $\to$ ADC)

For example, the edit distance between LOVE and MOVIE is 2, because we can first perform the operation LOVE $\to$ MOVE (modify) and then the operation MOVE $\to$ MOVIE (insert). This is the smallest possible number of operations, because it is clear that only one operation is not enough.

Suppose that we are given a string x of length n and a string y of length m, and we want to calculate the edit distance between x and y. To solve the problem, we define a function $distance(a, b)$ that gives the edit distance between prefixes $x[0...a]$ and $y[0...b]$. Thus, using this function, the edit distance between x and y equals $distance(n-1, m-1)$.

We can calculate values of $distance$ as follows:

$$\begin{aligned} distance(a,b) = min(&distance(a,b-1)+1,\\ &distance(a-1,b)+1,\\ &distance(a-1,b-1)+cost(a,b)). \end{aligned}$$

where $cost(a, b) = 0$ if $x[a] = y[b]$, and otherwise $cost(a, b) = 1$. The formula considers the following ways to edit the string x:

- $distance(a, b-1)$: insert a character at the end of x
- $distance(a-1, b)$: remove the last character from x
- $distance(a-1, b-1)$: match or modify the last character of x

In the two first cases, one editing operation is needed (insert or remove). In the last case, if $x[a] = y[b]$, we can match the last characters without editing, and otherwise one editing operation is needed (modify).

The following table shows the values of distance in the example case:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	1	2	3	4	5
O	2	2	1	2	3	4
V	3	3	2	1	2	3
E	4	4	3	2	2	2

The lower-right corner of the table tells us that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

		M	O	V	I	E
	0	1	2	3	4	5
L	1	↖	2	3	4	5
O	2	2	↖	2	3	4
V	3	3	2	↖	←	3
E	4	4	3	2	2	↖

The last characters of LOVE and MOVIE are equal, so the edit distance between them equals the edit distance between LOV and MOVI. We can use one editing operation to remove the character I from MOVI. Thus, the edit distance is one larger than the edit distance between LOV and MOV, etc.

The bottom-up implementation is given below:

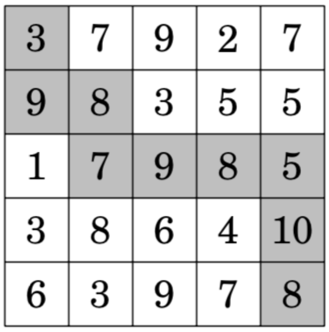
```
def EditDistance(X,Y)
for i = 0 to n:
  distance[i][0] = i
for j = 0 to m:
  distance[0][j] = j
for i = 1 to n:
  for j = 1 to m:
    distance[i][j] = min(distance[i-1][j] + 1,
                         distance[i][j-1] + 1,
                         distance[i-1][j-1] + (X[i]==Y[j]?0:1))
return distance[n][m]
```

Running time: $O(mn)$

6 Path in Grid

Our next problem is to find a path from the upper-left corner to the lower-right corner of an $n \times n$ grid, such that we only move down and right. Each square contains a positive integer, and the path should be constructed so that the sum of the values along the path is as large as possible.

The following picture shows an optimal path in a grid:



3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

Assume that the rows and columns of the grid are numbered from 1 to n, and $value[y][x]$ equals the value of square (y, x). Let $sum(y, x)$ denote the maximum sum on a path from the upper-left corner to square (y, x). Now $sum(n, n)$ tells us the maximum sum from the upper-left corner to the lower-right corner. For example, in the above grid, $sum(5, 5) = 67$.

We can recursively calculate the sums as follows:

$$sum(y, x) = max(sum(y, x - 1), sum(y - 1, x)) + value[y][x]$$

The recursive formula is based on the observation that a path that ends at square (y, x) can come either from square $(y, x - 1)$ or square $(y - 1, x)$.

[Draw a picture on the board]

Thus, we select the direction that maximizes the sum. We assume that $sum(y, x) = 0$ if $y = 0$ or $x = 0$ (because no such paths exist), so the recursive formula also works when $y = 1$ or $x = 1$. This is a useful implmentation trick to avoid excessive out-of-bound checking.

Here's an implementation:

```
int sum[N+1][N+1];
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1],sum[y-1][x])+value[y][x];
    }
}
```

Running time: $O(n^2)$

Question: How to recover the path?

7 Exercise: Min Cost Path

Given a cost matrix $cost[][]$ and a position (m, n) in $cost[][]$, write a function that returns cost of minimum cost path to reach (m, n) from $(0, 0)$. Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach (m, n) is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell (i, j), cells (i+1, j), (i, j+1) and (i+1, j+1) can be traversed. You may assume that all costs are positive integers.

8 Chain matrix multiplication

Suppose that we want to multiply four matrices, $A \times B \times C \times D$, of dimensions 50×20, 20×1, 1×10, and 10×100, respectively. This will involve iteratively multiplying two matrices at a time. Matrix multiplication is not commutative (in general, $A \times B \neq B \times A$), but it is associative, which means for instance that $A \times (B \times C) = (A \times B) \times C$. Thus we can compute our product of four matrices in many different ways, depending on how we parenthesize it. Are some of these better than others?

Multiplying an $m \times n$ matrix by an $n \times p$ matrix takes mnp multiplications, to a good enough approximation. Using this formula, lets compare several different ways of evaluating $A \times B \times C \times D$:

1. $A \times ((B \times C) \times D)$ costs 120200 ops
2. $(A \times B) \times (C \times D)$ costs 700 ops

As you can see, the order of multiplications make a hugh difference in the final running time. Greedy is not gauranteed to work here.

Suppose we want to find the optimal order of $A_1 \times A_2 \times \ldots \times A_n$ where A_i's are the matices with dimensions $m_0 \times m_1$, $m_1 \times m_2$, respectively.

Let $C(i, j)$ be the minimum number of multiplications needed to multiply $A_i \times A_{i+1} \times \ldots \times A_j$.

The smallest subproblems in this case are $C(1, 1)$, $C(2, 2)$ where 0 multiplications is needed. What about $C(1, 2)$? We can see that $C(1, 2) = C(1, 1) + C(2, 2) +$[number of multiplications needed for $A_1 \times$
or $C(1, 2) = C(1, 1) + C(2, 2) + m_0 m_1 m_2$

In general,

$$C(i, j) = \min_{i \leq k < j} (C(i, k) + C(k + 1, j) + m_{i-1} m_k m_j)$$

The top-down implementation is below:

```
def C(i,j):
    if i==j: return 0
    if (i,j) in known: return known[(i,j)]
    best = INF
    for k in range(i,j):
        best = min(best, C(i,k) + C(k+1,j) + m[i-1] * m[k] * m[j])
    known[(i,j)] = best
    return best
```

Here is the button-up version:

```
for i=1 to n: C[i][i]=0
for s=1 to n1:
  for i=1 to ns:
    j=i+s
    best = INF
    for k=i to j-1:
      best = min(best,C[i][k]+C[k+1][j]+m[i-1]*m[k]*m[j])
    C[i][j] = best
# return C[1][n]
```

The subproblems constitute a two-dimensional table, each of whose entries takes $O(n)$ time to compute. The overall running time is thus $O(n^3)$.

9 All-pair shortest path

What if we want to find the shortest path not just between s and t but between all pairs of vertices? One approach would be to execute Bellman-Ford algorithm (since there may be negative edges) $|V|$ times, once for each starting node. The total running time would then be $O(|V|^2|E|)$. Well now see a better alternative, the $O(|V|^3)$ dynamic programming-based known as *Floyd-Warshall* algorithm.

Observe that the shortest path $u \to w_1 \to \ldots \to w_l \to v$ between u and v uses one or more intermediate nodes. Suppose we disallow intermediate nodes altogether. Then we can solve all-pairs shortest paths at once: the shortest path from u to v is simply the direct edge (u, v), if it exists. What if we now gradually expand the set of permissible intermediate nodes? We can do this one node at a time, updating the shortest path lengths at each stage. Eventually this set grows to all of V , at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph!

More concretely, number the vertices in V as $\{1, 2, \ldots, n\}$, and let $dist(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates. Initially, $dist(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

What happens when we expand the intermediate set to include an extra node k? We must reexamine all pairs i, j and check whether using k as an intermediate point gives us a shorter path from i to j. But this is easy: a shortest path from i to j that uses k along with possibly other lower-numbered intermediate nodes goes through k just once (why? because we assume that there are no negative cycles). And we have already calculated the length of the shortest path from i to k and from k to j using only lower-numbered vertices:

$$\begin{aligned} dist(i, j, k) = min(&dist(i, j, k-1), \\ &dist(i, k, k-1) + dist(k, j, k-1)) \end{aligned}$$

Here is the Floyd-Warshall algorithmand as you can see, it takes $O(|V|^3)$ time.

```
# suppose G[i][j] = w(i,j) or infty
for i=1 to n:
  for j = 1 to n:
    dist[i][j][0] = G[i][j]
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
      dist[i][j][k] = min(dist[i][k][k1]+dist[k][j][k1],
                          dist[i][j][k1])
```

Or, we can use just $O(|V|^2)$ space:

```
# suppose G[i][j] = w(i,j) or infty
for i=1 to n:
  for j = 1 to n:
    dist[i][j] = G[i][j]
for k = 1 to n:
  for i = 1 to n:
    for j = 1 to n:
      dist[i][j] = min(dist[i][k]+dist[k][j], dist[i][j])
```