ICC4101 - Algorithms and Competitive Programing

Javier Correa

Dynamic Programing

Dynamic programing is a general technique to solve some optimization and/or search algorithm that allows to reduce the complexity from (usually) an exponential time complexity to a polynomial complexity.

The key insight is that the main problem can be dividing into sub-problmes which can be recursively solved. These problems have an *Optimal Substructure*.

In a sense, Dynamic Programming is a smart backtracking.

Many of the Dynamic Programming problems are optimization or counting problems.

A generic approach to solve dynamic programing problems is as follows:

1. Find sub-problems
2. "Guess" (partial) solution
3. Relate subproblems
4. Recursion and memoization, or use a bottom-up approach to build solutions (avoid repeating yourself)
5. Find solution to the original problem

Dynamic Programing

Description step-by-step.

1. Find sub-problems

Divide the original problem in a number of related sub-problems.

For example: Let $S[n]$ be a solution to sub-problems using only the first n constraints.

2. Guess partial solution

Construct a partial solution by *guessing* part of it.

For example: If we knew the optimal $n-1$-th sub-problem solution, we can construct the n-th sub-problem solution as:

$$S[n] = S[n-1] \oplus c^*(n).$$

3. Relate sub-problems

Find a general recursion for $S[n]$.

For example:

$$S[n] = \max_i S[n-1] \oplus c_i(n),$$

and

$$S[1] = k.$$

4. Recursion + memoization or bottom-up solution

Use recursion with memoization to solve the equation for $S[n]$ or build a solution $S[n]$ from the smallest partial solutions.

For example (memoization):

```
memoization = dict()
def S(n):
    if n in memoization:
        return memoization[n]

    Sn = max(S(n-1) + c(i) for i in I)
    memoization[n] = Sn
    return memoization[n]
```

Pro tip: `python` standard library provides a decorator to memoize function calls, see `functools.lru_cache` (or `functools.cache` in pythopn > 3.9) for more information.

Or with `functools.lru_cache`:

```
@functools.lru_cache(None)
def S(n):
    return max(S(n-1) + c(i) for i in I)
```

Example (bottom-up)

```
def S(n):
    partial_S = [CONST]
    for k in range(1, n+1):
        Sk = max(partial_S[k-1] + c(i) for i in I)
        partial_S.append(Sk)
    return partial_S[n]
```

5. Find the solution to the original problem

Find which partial solution(s) is the solution to the original problem.

Usually $S[n]$ with n the size of the problem is a sub-problem equivalent to the original problem.

Example: Fibonacci

1, 2 and 3: In this case, both steps are part of the definition of a Fibonacci number

$$F_n = 0, \text{ for } n = 0$$

$$F_n = 1, \text{ for } n = 1$$

$$F_n = F_{n-1} + F_{n-2}, \text{ in other cases}$$

4.1 Memoization solution

In [ ]:
```
memory = {0: 0, 1: 1}
def fib_memoization(n):
    if n in memory:
        return memory[n]
    fib_n = fib_memoization(n-1) + fib_memoization(n-2)
    memory[n] = fib_n
    return fib_n
fib_memoization(100)
```

In [ ]:
```
import functools
```

In [ ]:
```
@functools.lru_cache(None)
def fib_memoization2(n):
    if n <= 1:
        return n
    return fib_memoization2(n-1) + fib_memoization2(n-2)

fib_memoization2(100)
```

4.2 Bottom-up solution

fib(100) = fib(99) + fib(98) = (fib(98) + fib(97)) + fib(98)

In [ ]:
```
def fib_bottomup(n):
    if n <= 1:
        return n
    fn1 = 1
    fn2 = 0
    for i in range(2, n+1):
        fn = fn1 + fn2
        fn1, fn2 = fn, fn1
    return fn

fib_bottomup(100)
```

Example: Shortest path (Bellman-Ford algorithm)

Find the shortest path a node to all other nodes.

1. Sub-problem definition

Define the sub-problem $S[u, v]$ as distance of the shortest path between nodes u and v.

2. Guessing partial-solution

If we know the optimal first step, we can divide the shortest path between u and v as a path between u and w, and a single optimal path between w and u:

$$S[u, v] = S[u, w] + c(w, v)$$

3. Relate sub-problems

Since we don't know the optimal node w of for the last step, we can write the optimal partial solution as a minimization problem as follows:

$$S[u, v] = \min_{w} S[u, w] + c(w, v)$$

Problem!! This formulation fails for graphs with cycles!

1. Sub-problem definition

Define the sub-problem $S[u, v, k]$ as distance of the shortest path between nodes u and v using at most k nodes.

2. Guessing partial-solution

Make explicit the length of the path:

$$S[u, v, k] = S[u, w, k-1] + c(w, v)$$

3. Relate sub-problems

Relate sub-problems of increasing path-length. Note that now we must compare the paths of that use k nodes and the paths using $k-1$ nodes:

$$S[u, v, k] = \min\left[S[u, v, k-1], \left(\min_{w} S[u, w, k-1] + c(w, v)\right)\right],$$

with

$$S[u, v, 1] = c(u, v)$$

4.1 Memoization solution

In [ ]:
```
G = {
    0: {1: 1, {2: 1}, ...},
    ...
}

memory = dict()
def short_path_memory(G, u, v, k):
    if k == 1:
        return G[u][v]
    if (u, v, k) in memory:
        return memory[(u, v, k)]
    Sk = min(short_path_memory(G, u, v, k-1),
             min(short_path_memory(G, u, w, k-1) + G[w][v] for w in G))
    memory[(u, v, k)] = Sk
    return Sk
```

4.2 Bottom-up solution

In [ ]:
```python
INF = 1E6
def short_path_bottomup(G, u):
    D1 = defaultdict(lambda: math.inf)
    for v in G[u]:
        D1[v] = G[u][v]
    for k in range(1, len(G)):
        D2 = defaultdict(lambda: math.inf)
        for v in G:
            D2[v] = min(D1[v], min(D1[w] + G[w][v] for w in G))
        D1 = D2
    return D1
```

Edit distance

Given two strings s_1 and s_2, what is the minimum number of insertion or deletions to make both strings equal?

- Sub-problem: calculate edit distance between $s_1[i:]$ and $s_2[j:]$

- Guess solution. If $s_1[i] \neq s_2[j]$:
  - Remove $s_1[i]$ (or prepend $s_1[i]$ to $s_2[j:]$) and calculate edit distance between $s_1[i+1:]$ and $s_2[j:]$
  - Remove $s_2[j]$ (or prepend $s_2[j]$ to $s_1[i:]$) and calculate edit distance between $s_1[i:]$ and $s_2[j+1:]$

- Relate sub-problems:

$$d(s_1, s_2) = \begin{cases} \max |s_1|, |s_2|, & |s_1| = 0 \vee |s_2| = 0 \\ d(s_1[1:], s_2[1:]) & s_1[0] = s_2[0] \\ 1 + \min d(s_1[1:], s_2), d(s_1, s_2[1:]), & \text{in other cases.} \end{cases}$$

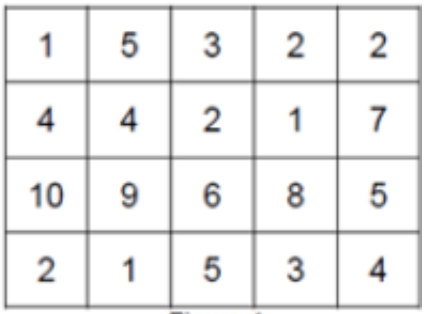
- Recursion with memoization

In [ ]:
```python
@functools.lru_cache(None)
def string_distance(s1, s2):
    if len(s1) == 0 or len(s2) == 0:
        return max(len(s1), len(s2))
    if s1[0] == s2[0]:
        return string_distance(s1[1:], s2[1:])
    else:
        return 1 + min(string_distance(s1[1:], s2),
                       string_distance(s1, s2[1:]))
```

```
In [ ]: string_distance('electroencefalografista', 'esternocleidomastoideo')
```

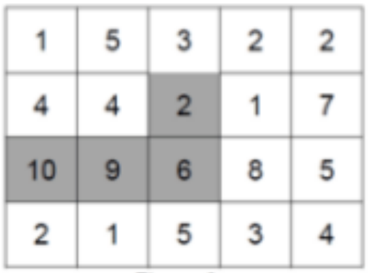
Problem 0 (Exercise)

In spite of his old age, Grandpa Pierre likes to take a walk. He believes that by taking more exercises, he will be healthy and live a longer life. The area which Grandpa used to explore can be represented as a N ×M map, where each cell contains the height information of that area. Grandpa can start from any cell and walk from a cell to its adjacent cells (north, south, west, east) if and only if the destination cell's height is strictly lower than his current cell's height. For example, consider a 4 × 5 map below.



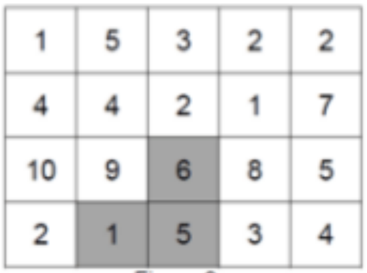
1	5	3	2	2
4	4	2	1	7
10	9	6	8	5
2	1	5	3	4

Figure 1



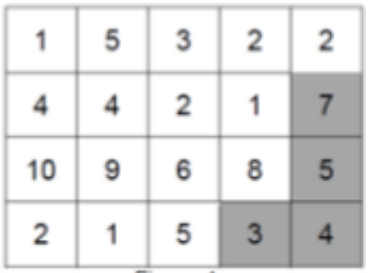
1	5	3	2	2
4	4	2	1	7
10	9	6	8	5
2	1	5	3	4

Figure 2



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

Figure 3



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

Figure 4

There are many possible routes, for example:

- Figure 2 shows a route of length 4 which starts from a cell with height 10 and ends at a cell with height 2 (10 - 9 - 6 - 2).
- Figure 3 shows a route of length 3 which starts from a cell with height 6 and ends at a cell with height 1 (6 - 5 - 1).
- Figure 4 shows a route of length 4 which starts from a cell with height 7 and ends at a cell with height 3 (7 - 5 - 4 - 3)

There are many other routes we can get from Figure 1 which are not shown.

A route is considered maximal if and only if one cannot extend the route at the beginning or at the end to make it a longer route.

- Figure 2 is not a maximal route. We can extend the end of the route from cell with height 2 to its east, a cell with height 1, such that the route is 10 - 9 - 6 - 2 - 1 with length of 5.

- Figure 3 is not a maximal route. We can extend the beginning of the route from cell with height 6 to its east, a cell with height 8, such that the route is 8 - 6 - 5 - 1 with length of 4.
- Figure 4 corresponds to a maximal route because we can extend neither the beginning nor the end of the route.

Your task is to help Grandpa Pierre to count how many maximal routes there are in the given map.

Input

The first line of input contains an integer T (T ≤ 100) denoting the number of cases. Each case begins with two integers N and M (1 ≤ N, M ≤ 50) denoting the number of rows and columns in the map respectively. The next N lines each contains M integers Aij (1 ≤ Aij ≤ 100) denoting the height of a cell in i-th row and j-th column.

Output

For each case, output ‘Case #X: Y ’, where X is case number starts from 1 and Y is the number of maximal route we can find in the given map such that the route visits cells in strictly decreasing height order. Notes:

- Explanation for 2nd sample input. All cells have the same height, so there are only routes with length 1, and there are 12 of them.
- Explanation for 3rd sample input.

List of visited cells’ height of all maximal routes are: 1. 7 - 1 2. 7 - 1 3. 7 - 6 - 2 - 1 4. 8 - 5 - 4 - 1 5. 8 - 7 - 4 - 1 6. 8 - 7 - 6 - 1 7. 8 - 7 - 6 - 1

Notice there are two (7 - 1) above and both are different routes. The first one corresponds to a route from cell (1, 1) to cell (0, 1), while the second one is from cell (1, 1) to cell (1, 2). A same explanation applies for (8 - 7 - 6 - 1).

1. Sub-problems

$R[i,j]$ are the number of routes starting at cell (i,j) and is cell (i,j) a start cell.

2. Guess

No need to guess. Explore all solutions.

3. Relate sub-problems

$$\begin{aligned} R[i,j] = & R[i-1,j]\cdot\delta[H[i-1,j] < H[i,j]] & (1)\\ & + R[i+1,j]\cdot\delta[H[i+1,j] < H[i,j]] & (2)\\ & + R[i,j-1]\cdot\delta[H[i,j-1] < H[i,j]] & (3)\text{'}\\ & + R[i,j+1]\cdot\delta[H[i,j+1] < H[i,j]] & (4)\end{aligned}$$

where:

$$\delta[F] = \begin{cases} 1, & \text{if } F \text{ is true} \\ 0, & \text{in other cases.}\end{cases} .$$

If no neighbors with lower height, then $R[i,j] = 1$.

4. Recursion and memoization

```
In [ ]: import functools
        import collections
```

```
In [ ]: # Map
        n,m = 4,5
        map_ = [[1, 5, 3, 2, 2],
                [4, 4, 2, 1, 7],
                [10, 9, 6, 8, 5],
                [2, 1, 5, 3, 4]]
        # Initial cell
        I = collections.defaultdict(bool)
```

```
In [ ]: @functools.lru_cache(None)
        def max_path_length(i, j):
            Hij = map_[i][j]
            Rij = 0
            for di, dj in [(i+1, j), (i-1, j), (i, j+1), (i, j-1)]:
                if di >= 0 and di < n and dj >= 0 and dj < m:
                    if Hij > map_[di][dj]:
                        Rij += max_path_length(di, dj)
                        I[di, dj] = True
            if Rij == 0:
                return 1
            return Rij
```

```
In [ ]: from itertools import product

        for i,j in product(range(n), range(m)):
            max_path_length(i, j)

        total = 0
        for i,j in product(range(n), range(m)):
            r = max_path_length(i, j)
            if not I[i, j]:
                total += r

        total
```

Classical Dynamic Programing

In both previous examples we have had to come up with the recursion ourselves. However, there are a number of *classical* DP problems for which the recursion (and solution) are well-known:

I. Max 1/2D sums

Finding the maximum sum of a 1D array can be computed by brute force in $O(n^3)$ (calculate all possible sums of all possible intervals and find the maximum one). Using `MAXSUM(i)` as the value for the maximum sum ending at index `i` we can calculate it in $O(n)$:

```
MAXSUM(0) = V[0]
MAXSUM(i) = max(MAXSUM(i-1) + V[i], V[i])
```

And the maximum sub-sum is `max(MAXSUM(i))`.

For 2D arrays, the brute force approach can be achieved in $O(n^6)$, but by pre-computing the accumulated array, finding the maximum sub-rectangle sum can be achieved in $O(n^4)$.

II. Longest increasing subsequence

The problem is to find a sun-sequence of numbers (not necessarily contiguous) such as the values are in an increasing order. A brute force algorithm that lists/enumerates all sub-sequences and then finds the longest one, has complexity $O(2^n)$. A better approach is to defined the function `LIS(i)` as the longest sub-sequence up to index `i`, and then the recursion:

```
LIS(0) = 1
LIS(i) = max(LIS(j) + 1 for j in range(i) if seq[j] < seq[i])
```

Exercise: Analyze this recursion, what's its complexity?

III. 0-1 Knapsack (Subset Sum)

This is a known NP problem, and as such, hard to solve. Given `n` items with values `v[i]` and weights `w[i]` and a knapsack size `S`, find the maximum value you can carry by either taking or ignoring an item.

To solve this problem, we define `value(id, weight)` as the value of using items labeled `id` or above having remaining weight `weight`. It is possible to write the recursion as follows:

```
val(_, 0) = 0
val(N, _) = 0
val(id, weight) = val(id + 1, weight) if w[id] > weight else \
                  max(val(id + 1, weight), v[id] + val(id + 1, weight - w[id]))
```

Exercise: Analyze this recursion, what's its complexity? Have we solved $P \stackrel{?}{=} NP$

IV. Coin change

Given a value `V` and coin values `values[c]` for coin `c`, what is the minimum number of coins we must use to represent `V`? We can represent this problem by the simple function:

```
change(0) = 0
change(v) = -inf if v < 0 else \
            1 + min(change(v-c) for c in values)
```

```
In [ ]:
```