ICC4101 - Algorithms and Competitive Programing

Javier Correa

Dynamic Programing

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

Types of parameters for Dynamic Programing problems

Most DP sub-problems will have parameters (or a combination) of these types:

1. Index `i` of an array `[a_0, a_1, ..., a_i, ...]`.

The steps of guessing and relating sub-problmes implies including more elements of the array. For example, the Longest Increasing Subsequence (LIS) problem. 2. Indices `(i, j)` of two arrays `[a_0, a_1, ..., a_i, ...]` and `[b_0, b_1, ..., b_j, ...]`. The steps of guessing and relating sub-problmes implies incresing the index `i`, `j` or both. For example, calculating the edit distance.

3. Sub-indices `(i, j)` of an array `[a_0, a_1, ..., a_i, a_{i+1}, ..., a_j, ...]`.

The steps of guessing and relating sub-problmes implies dividing the problem such as `(i, i + k)` and `(i + k, j)`. 4. Vertex position in a DAG (usually an implicit graph). Process the neighbors (next or previous vertex).

5. Knapsack-stype parameter.

The step of guessing and realting sub-problem implies incresing (or decreasing) a value (or weight?) until the maximum value or threshold. 6. Small subset, usually modeled with bitmask (for example if only element 1 and 3 are part of the set, this can me modeled with the bitmask `010100...`). The step of guessing and relating sub-problems implies changing one or more 0s to 1s.

Previously...

1. Undirectional TSP

Technically a Graph problem. Solution similar to Bellmand-Ford or Floy-Warshall but since it is a DAG (Direct **Acyclic** Graph) a simple solution exists:

- `path(i, j) = (c, p)` minimum cost `c` and path `p` to reach the last row from row `i` and column `j`.
- Guess if the next step (`j+1`) goes through th row `i-1`, `i` or `i+1`.
- Minimize!

```
import sys
from itertools import chain
from functools import lru_cache

def numbers():
    yield from chain.from_iterable(map(int, line.split()) for line
in sys.stdin.readlines())

def main():
    nums = numbers()
    while True:
        try:
            n, m = next(nums), next(nums)
            M = dict()
            for i in range(n):
                for j in range(m):
                    M[i, j] = next(nums)
        except StopIteration:
            return
```

```
@lru_cache(None)
        def path(i, j):
            if i < 0:
                i = n + i
            if i == n:
                i = 0
            if j == m:
                return 0, []

            min_cost, min_path = min(path(i-1, j+1), path(i, j+1),
path(i+1, j+1))
            return M[i,j] + min_cost, [i+1] +  min_path

        min_cost, min_path = min(path(i, 0) for i in range(n))
        print(" ".join(map(str, min_path)))
        print(min_cost)

main()
```

2. Testing the CATCHER

Longest **Decreasing** Sub-sequence

- `LIS(i)` longest decreasing sub-sequence starting at index `i`.
- Guess: the next step `k` of the longest sub-sequence starting at index `i`

```
from sys import stdin
from itertools import takewhile, count


for k in count():
    heights = list(takewhile(
        lambda x: x != -1,
        map(int, (stdin.readline() for _ in count())))
    ))
    N = len(heights)
    if N == 0:
        break
```

```
capture = N*[1]
    for i in range(N-2, -1, -1):
        capture[i] = max(
            (1 + capture[j] for j in range(i+1, N) if heights[i] >=
heights[j]),
            default=1
        )

    intercepts = max(capture)
    if k > 0:
        print()
    print(f"Test #{k+1}:")
    print(f"  maximum possible interceptions: {intercepts}")
```

3. Jill Rides Again

Maximum sum between an interval.

1. `nice(i, j)` nice route value between indices `i` and `j`
2. No Guessing
3. `nice(i, j) = nice(i, j-1) + route[j-1]`

4. Compromise

Longest **Common** Sub-sequence

1. `LCS(i, j)` is the longest common subsequence starting at index `i` at the first sequence and index `j` at the second sub-sequence
2. Guess the next word of the LCS

```
@lru_cache(None)
def lcs(i, j):
    if i == len(words1) or j == len(words2):
        return []
    if words1[i] == words2[j]:
        return [words1[i]] + lcs(i+1, j+1)
    return max([lcs(i+1, j), lcs(i, j+1), lcs(i+1, j+1)], key=len)
```

5. Coin Change

1. `ways(v, i)` ways of giving change to `v` cents using coins with index `i` or above (to avoid repetitions)
2. No guessing
3. `ways(v, i) = sum(ways(v - coins[k], k) for k in range(i, N))`

Why use index `i`?

```
COINS = [50, 25, 10, 5, 1]

@lru_cache(maxsize=None)
def ways(value, i=0):
    if value == 0:
        return 1
    elif value < 0:
        return None

    ways_ = 0
    for k, c in enumerate(COINS):
        if k < i:
            continue
        v = ways(value - c, k)
        if v is None:
            continue
        ways_ += v
    return ways_
```

6. Diving for gold

1. `value(t, i)` maximum value of treasures to get with `t` remainding seconds and only considering treasures with index `i` and above.
2. Guess which is the optimal treasure to get with `t` remaining seconds
3. `value(t, i) = max(value(t, id + 1), VAL[id] + value(t - 3 times[id], id + 1))`

```
@lru_cache(maxsize=None)
def value(time, treasure):
    if time == 0 or treasure == N:
        return 0, [], []

    # Ignore
    skip_value, skip_times, skip_times = value(
        time,
        treasure + 1
    )
```

```
        # Take item treasure
            if TIME[treasure] <= time:
                take_value, take_times, take_treasures = value(
                    time - TIME[treasure],
                    treasure + 1
                )
                take_value += GOLD[treasure]
                if take_value > skip_value:
                    take_times_ = [time] + take_times
                    take_treasures_ = [treasure] + take_treasures
                    return take_value, take_times_, take_treasures_
            return skip_value, skip_times, skip_times

        while True:
            value.cache_clear()
            T, w = map(int, stdin.readline().split())
            N = int(stdin.readline())
            TIME = N*[0]
            GOLD = N*[0]
            for i in range(N):
                d, GOLD[i] = map(int, stdin.readline().split())
                TIME[i] = 3*w*d
            val, times, treasures = value(T, 0)
            print(val)
            print(len(treasures))
            for t, T in zip(times, treasures):
                print(t, GOLD[t])
            print()
```

For this week!

Explain with comments (one or two sentences) the optimal sub-stucture of your dynamic programming solution!

In [ ]: