Competitive Programming I

Dynamic Programming

October 31, 2024

Dynamic Programming

Dynamic programming:

- Breaking down a problem into smaller sub-problems by figuring out the problem states
- Determining the transitions between the current problem and its sub-problems
- Overlapping sub-problems, i.e., sub-problems are repeated in several problems

Dynamic Programming

Dynamic programming:

- Breaking down a problem into smaller sub-problems by figuring out the problem states
- Determining the transitions between the current problem and its sub-problems
- Overlapping sub-problems, i.e., sub-problems are repeated in several problems

Dynamic programming oftens shows up in

- Optimization problems — minimize/maximize a function
- Counting problems — count the number of feasible/optimal solutions

There are two main approaches:

- Top-down
- Bottom-up

For now we will focus on top-down

Dynamic Programming — Top-down

For a top-down approach we usually need to identify:

- A recursive formulation for the problem
- The overlapping sub-problems

Then, we need to memoize the solutions of sub-problems

Fibonacci Sequence

$$
\begin{aligned}
f(0) &= 1 \\
f(1) &= 1 \\
f(n) &= f(n-1) + f(n-2)
\end{aligned}
$$

We have our recursive formulation.

We also have overlapping sub-problems, for example:

- $f(5)$ is a sub-problem for both $f(6)$ and $f(7)$

How to memoize solutions to the sub-problems?

Basic recursive implementation

Code

```
int64_t fib(int64_t n) {
  if (n <= 1) return 1;
  return fib(n-1) + fib(n-2);
}

int main() {
  cout << fib(45) << "\n";
}
```

Result

```
1836311903
```

Took 2.8 seconds

Dynamic Programming — Fibonacci

Add memoization using a global `vector`

Code

```
vector<int64_t> mem;

int64_t fib(int64_t n) {
  if (n <= 1) return 1;
  if (mem[n] > 0) return mem[n];
  mem[n] = fib(n-1) + fib(n-2);
  return mem[n];
}

int main() {
  mem.assign(100, 0); // assumes f(99) is the max call
  cout << fib(45) << "\n";
}
```

Dynamic Programming — Fibonacci

Result

`1836311903`

Took 0.005 seconds

Complexity?

Result

```
1836311903
```

Took 0.005 seconds

Complexity? $O(N)$ since there are at most N states, and each state takes constant time to be computed (after the sub-problems are computed).

Alternative we can also use an `unordered_map`

Code

```
unordered_map<int64_t, int64_t> mem;

int64_t fib(int64_t n) {
  if (n <= 1) return 1;
  if (auto it = mem.find(n); it != mem.end())
    return it->second;
  auto [it, _] = mem.emplace(n, fib(n-1) + fib(n-2));
  return it->second;
}

int main() {
  cout << fib(45) << "\n";
}
```

Dynamic Programming — Fibonacci

Result

```
1836311903
```

Took 0.005 seconds

Dynamic Programming — Nikola (kattis)

Let's look at another exercise:

https://open.kattis.com/problems/nikola

We start by finding a recursive formula:

Dynamic Programming — Nikola

We start by finding a recursive formula:

$$
\begin{aligned}
f(p,j) &= c_p && \text{if } p = N \\
f(p,j) &= \infty && \text{if } p < 1 \vee p > N \\
f(p,j) &= c_p + \min\{f(p-j,j), f(p+j+1,j+1)\}
\end{aligned}
$$

Where p is our current position, j is the last jump length, and c_p is the cost on position p. To solve the problem we need to call $f(2,1)$.

Do we have overlapping sub-problems?

Dynamic Programming — Nikola

We start by finding a recursive formula:

$$
\begin{aligned}
f(p, j) &= c_p && \text{if } p = N \\
f(p, j) &= \infty && \text{if } p < 1 \vee p > N \\
f(p, j) &= c_p + \min\{f(p - j, j), f(p + j + 1, j + 1)\}
\end{aligned}
$$

Where p is our current position, j is the last jump length, and c_p is the cost on position p. To solve the problem we need to call $f(2, 1)$.

Do we have overlapping sub-problems?

Yes, for example $f(4, 3)$ is a sub-problem for $f(1, 3)$ and for $f(8, 4)$.

Okay, then the next step is to memoize $f(p, j)$. What is the size of the table we need?

We start by finding a recursive formula:

$$
\begin{aligned}
f(p,j) &= c_p && \text{if } p = N \\
f(p,j) &= \infty && \text{if } p < 1 \vee p > N \\
f(p,j) &= c_p + \min\{f(p-j,j), f(p+j+1,j+1)\}
\end{aligned}
$$

Where p is our current position, j is the last jump length, and c_p is the cost on position p. To solve the problem we need to call $f(2,1)$.

Do we have overlapping sub-problems?

Yes, for example $f(4,3)$ is a sub-problem for $f(1,3)$ and for $f(8,4)$.

Okay, then the next step is to memoize $f(p,j)$. What is the size of the table we need? $N \times N$ since jump cannot be bigger than N in a valid solution.

Pseudo-code for a possible DP top-down solution

```
mem[N+1][N+1] = {-1} // initialize mem table with -1

f(p, j):
  if p < 1 or p > N:
    return infinity
  if p == N:
    return cost[p]
  if mem[p][j] == -1:
    mem[p][j] = min(f(p-j, j), f(p+j+1, j+1))
    if mem[p][j] != infinity:
      mem[p][j] += cost[p]
  return mem[p][j]
```

Complexity?

Pseudo-code for a possible DP top-down solution

```
mem[N+1][N+1] = {-1} // initialize mem table with -1

f(p, j):
  if p < 1 or p > N:
    return infinity
  if p == N:
    return cost[p]
  if mem[p][j] == -1:
    mem[p][j] = min(f(p-j, j), f(p+j+1, j+1))
    if mem[p][j] != infinity:
      mem[p][j] += cost[p]
  return mem[p][j]
```

Complexity? $O(N^2)$ since there are at most N^2 states, and each state takes constant time to compute (after computing the sub-problems).

Tips for implementing the previous pseudo-code on `C++`:

- Infinity can be `numeric_limits<int64_t>::max()`. However, be careful with overflow. The `if aux != infinity` conditional prevents overflow.
- Use a `vector` or `array` for `mem`. Alternatively, you may also use a `map` with `log` complexity on each operation. To use `unordered_map` you would need to define an hash function.

What if the problem was to count the number of solutions? Different recursive formula, same memoization logic:

$$
\begin{aligned}
f(p,j) &= 1 && \text{if } p = N \\
f(p,j) &= 0 && \text{if } p < 1 \lor p > N \\
f(p,j) &= f(p-j,j) + f(p+j+1,j+1)
\end{aligned}
$$

Dynamic Programming — Nikola variations

What if the problem was to count the number of optimal solutions? This is a bit more tricky. One possibility would be to first compute the optimal solution *opt*, and then define the following recursion:

$$f(p,j,c) = 1 \quad \text{if } p = N \wedge c = opt$$
$$f(p,j,c) = 0 \quad \text{if } p = N \wedge c \neq opt$$
$$f(p,j,c) = 0 \quad \text{if } p < 1 \vee p > N \vee c > opt$$
$$f(p,j,c) = f(p-j,j,c+c_{p-j}) + f(p+j+1,j+1,c+c_{p+j+1})$$

However, since *opt* (and consequently c) can be quite large, this can be computationally expensive. A more efficient solution would be to solve the optimization problem and count the number of solutions from the `mem` table (we will look at this next class).

Dynamic Programming — Other tips

Regarding data structure for memoization table:

- Using a `vector` is often more efficient than an `unordered_map`.
- An `unordered_map` can be more efficient for sparse data (e.g. if not all states are possible)

Regarding state:

- If the state is a string, use an `unordered_map` possibly with a custom hashing function. Rolling hashing functions can be efficient if the operations on the string are limited.
- If the state is a binary vector, you can either encode it with a `bitset` or `vector<bool>` and use an `unordered_map` for the memoization table, or if the number of bits is small you can encode it as an integer and either use a `vector` or `unordered_map` depending on the integer size.