Lecture 4 : Introduction to Dynamic Programming

Justin Pearson

Today's Topics

- Overview of different types of algorithm design.
- Introduction to Dynamic programming:
  - Dynamic programming as recursion optimised with caching.
  - Dynamic programming as dividing a problem into smaller sub problems.

Dynamic programming is a common solution technique, and is often a quick way to get an efficient algorithm.

Dynamic programming is very often the solution to various difficult coding interview questions. Often because it is the only thing of substance the interviewer remembers from their algorithms course.

Three algorithmic paradigms

Greedy Process the input in some order making irreversible decisions hoping that you get a solution and if you do hoping that it is a good solution. Later on we'll see when greed is good.

Divide and Conquer Divide the input into smaller parts, solve on the smaller parts and then combine the solutions.

Dynamic Programming Breaking a problem into smaller sub-problems (not the same as dividing the input smaller parts) and combine them.

Dynamic Programming — Sub-problems

Overlapping Sub-problems This is a rather confusing turn of phrase, but it means we divide it into sub-problems that we use more than once. If we only use the results of the sub-problems once, then it is just divide and conquer.

Optimal Substructure Property A problem has the Optimal Substructure Property if if an optimal solution can be constructed from optimal solutions of its sub-problems.

Dynamic Programming — History and Etymology

Invented in the 1950's by Richard Bellman. It is well worth reading the original paper.[1]

- Programming — making a schedule not computer programming.
- Dynamic — Because Richard Bellman thought it sounded cool. He used it to disguise that fact that he was doing mathematics from his Washington paymasters.

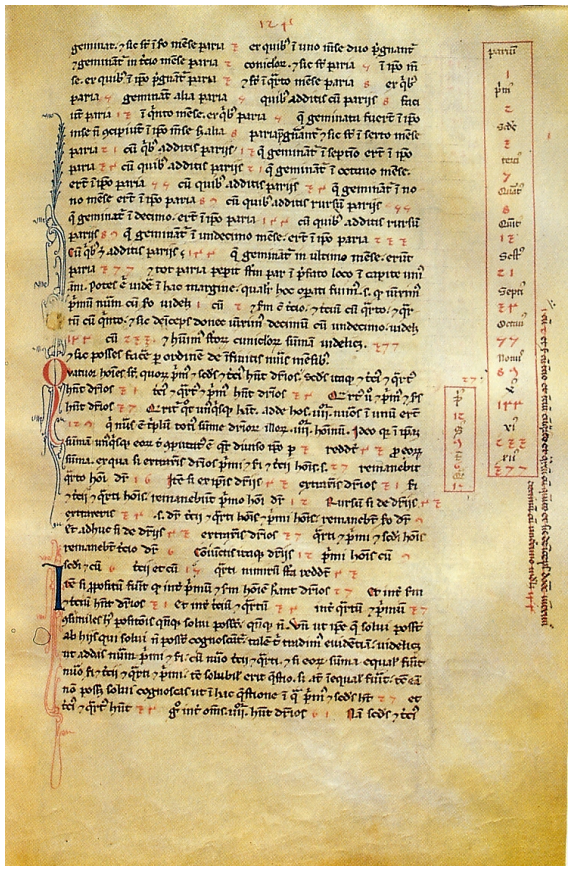
Making schedules over time, or solving multi-stage problems over time.

It is best not to get too hung up on terminology.

[1] Bellman R. The theory of dynamic programming. Bulletin of the American Mathematical Society. 1954;60(6):503-15.

Fibonacci numbers

Long history. First appearance around 200BC in Indian mathematics. Introduced to the west by Leonardo of Pisa, a.k.a Fibonacci in 1202. In the same book he introduced the numeral zero and positional notation for numbers.



Fibonacci Numbers

Fibonacci numbers appear in lots of areas of mathematics, art and even music (Bartok was obsessed with them). They are connected with the Golden section.

Definition $F_0 = F_1 = 1$ and for $n > 1$

$$F_n = F_{n-1} + F_{n-2}$$

This gives the sequence:

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

Some people start from 1, but starting from 0 makes some of the code examples easier later.

Closed-form expression

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

$$F_n \approx \frac{\phi^n}{\sqrt{5}}$$

where

$$\phi = \frac{1+\sqrt{5}}{2}$$

So $F_n \in O(2^n)$ since $\phi \approx 1.6180339887\ldots$.

Computing Fibonacci numbers

```
def fib(n) :
    if n == 0 :
        return 1
    if n == 1 :
        return 1
    return fib(n-1) + fib(n-2)
```

Let T_{fib} be the number of steps to calculate F_n. Question: What is a answer to

$$T_{\text{fib}}(n) \in O(???)$$

T_{fib}

Ignoring the base case:

$$T_{\text{fib}}(n) = T_{\text{fib}}(n-1) + T_{\text{fib}}(n-2) = F_n$$

Computing a Fibonacci number with recursion has the same complexity as the value of the Fibonacci number. So

$$T_{\text{fib}}(n) \in \Theta\left(\frac{\phi^n}{\sqrt{5}}\right)$$

Doing things more efficiently

Why is this code so inefficient?

```
def fib(n) :
    if n == 0 :
        return 1
    if n == 1 :
        return 1
    return fib(n-1) + fib(n-2)
```

What are two strategies for making this faster?

Doing things more efficiently

It is slow because we keep repeating the same work over and over again. There are two strategies for making things go faster:

- Use caching (often called memoisation or memoization)
- Use a loop.

Fibonacci with Caching — without using functools

```
cache = {}
def fib_cache(n) :
    if n not in cache.keys() :
        cache[n] = _fib_cache(n)
    return cache[n]

def _fib_cache(n) :
    if (n == 0) or (n == 1):
        return 1
    else:
        return fib_cache(n-1) + fib_cache(n-2)
```

```
>>> fib_cache(5)
8
>>> print(cache)
{1: 1, 0: 1, 2: 2, 3: 3, 4: 5, 5: 8}
```

The complexity of the Cached Version

Remember I had two functions

- `fib_cache(n)` that checks if we have already computed the number
- `_fib_cache(n)` that does the recursion on `fib_cache(n)`

Thus to compute say `fib_cache(4)` for the first time it would call `_fib_cache(4)` that calls `fib_cache(3)` and `fib_cache(2)` and so on.

When you call `fib_cache` it is either a constant time lookup or a call to `_fib_cache(n)`

For each n we only call `_fib_cache(n)` once. So the complexity of `fib_cache(n)` is linear in n.

Computing Fibonacci numbers with a loop

```
def fib_loop(n):
    loop_cache = [1]*(n+1)
    for i in range(2,n+1):
     loop_cache[i] =
         loop_cache[i-1] + loop_cache[i-2]
    return loop_cache[n]
```

The statement `[1]*(n+1)` creates a list with $(n+1)$ elements all with the value 1.

Again the complexity of this function is linear in n.

We are assuming that array/list lookup is constant time, and it sort of is in Python.

Computing Fibonacci numbers with a loop

The function on the previous slide uses an array to store the intermediate values. So it has a space requirement linear in n.
We can optimise away the space by only storing the numbers we need:

```
def fib_loop_optimised(n) :
    if (n ==0) or (n == 1) :
        return 1
    f_n_minus_1 = 1
    f_n_minus_2 = 1
    for i in range(2,n+1) :
        current = f_n_minus_1 + f_n_minus_2
        f_n_minus_2 = f_n_minus_1
        f_n_minus_1 = current
    return current
```

Dynamic Programming — Approaches

You now know almost everything there is to know about dynamic programming. There are two approaches:

Bottom Up Compute the smaller sub-problems first and combine the answers to get the answer to what you are looking for.

Top Down Essentially implement a recursive solution with caching to reduce the time complexity.

Both bottom up and top down have the same asymptotic time complexity. The same space complexity, unless you to some more optimisation with bottom up.

Dynamic Programming — Top Down

Top down dynamic programming, implement the recursive solution but using caching.

You either to do this automatically or with some explicit data structure.

Top down uses the stack to do the recursion. In some programming (not all) languages this can be expensive.

Dynamic Programming — Bottom Up

With bottom up there are two strategies:

- Be very clever about what order you compute things, and pass the values on to the next value that needs to be computed. Fibonacci with a loop and a table or some other data structure. This is often quite hard to do.
- Use some data structure to store intermediate values. Often this is the same data structure that you would have used in a top down approach. You still have to compute things in the right order, but you don't have to be as careful as when you don't use a table.

You can think of the first approach as a highly optimised version of the second approach. Throw away the data that you do not need. So try the second approach first, and then optimise.

Dynamic Programming is not just recursion

So far we have made it look like dynamic programming is just a fancy way of optimising recursion. In fact there are libraries out there, that with a minimal amount of effort will do automatic caching of your recursive functions. You can do it yourself with Python decorators or using the `functools` package.
Remember from before:

Overlapping Sub-problems Solve your problem by using sub-problems.

Optimal Substructure Property A problem has the Optimal Substructure Property if if an optimal solution can be constructed from optimal solutions of its sub-problems.

The key thing to remember is that you split a problem into sub-problems. How you do this depends on the problem.

According to Riksbank the currently valid coins are: 1-krona, 2-krona, 5-krona and 10-krona.[2]

Back in the stone age (when I was young) people used coins and notes. If you bought something for 87 krona and used a 100 krona note. You expected to get 13 krona change with the minimal number of coins. So $10 + 1 + 2$ rather than 13 1-krona coins.

[2] Pictures taken from Riksbankens website

Coin Change Example

We are going to write a function, $C_{\min}$, that tells you the minimum number of coins that you need to give change. So for example with Swedish money we should get

$$C_{\min}(13) = 3$$

If we trying to compute $C_{\min}(n)$ and we have the information for $C_{\min}(n-1)$ $C_{\min}(n-2)$ $C_{\min}(n-5)$ and $C_{\min}(n-10)$ then you should be able to work out $C_{\min}(n)$.

Coin Change Example

Set $C_{\min}(1) = C_{\min}(2) = C_{\min}(5) = C_{\min}(10) = 1$ and $C_{\min}(0) = 0$. Then we get

$$C_{\min}(n) = \min\{1 + C_{\min}(n-k) \mid n - k \geq 0, k \in \{1, 2, 5, 10\}\}$$

Note we are using set comprehension here.

Coin Change Example

Same formula from the previous slide without the base cases.

$$C_{\min}(n) = \min\{1 + C_{\min}(n-k) \mid n-k \geq 0, k \in \{1,2,5,10\}\}$$

So if we want to work out $C_{\min}(13)$ we look at $C_{\min}(13-1)$, $C_{\min}(13-2)$, $C_{\min}(13-5)$ and $C_{\min}(13-10)$ pick the one that is smallest, but we remember to add 1 because we use one more coin.

Coin Change : Possible Python implementation

Before you do top-down or bottom up you can come up with an inefficient recursive solution.

```
def min_coin(n) :
if (n == 0) :
    return 0
if n in [1,2,5,10] :
    return 1
best = n
for k in [1,2,5,10] :
    if (n - k ) >= 0 :
        current = 1 + min_coin (n - k)
        best = min(current,best)
return best
```

Coin change — Optimal Substructure property

You have to argue that the expression

$$C_{\min}(n) = \min\{1 + C_{\min}(n-k) \mid n-k \geq 0, k \in \{1, 2, 5, 10\}\}$$

is correct.

Proving the correctness of the recursive equation means that you are showing that the solution to $C_{\min}(n)$ can be computed just by looking at the sub-problems.

We are trying compute the optimal number of coins, this is why it is called the optimal sub-structure property. Look at the demo report for an example and how we expect you to argue the correctness.

Coin change — Optimal Substructure property

We can prove correctness by contradiction. Assume that:

$$C_{\min}(n) < \min\{1 + C_{\min}(n-k) \mid n-k \geq 0, k \in \{1,2,5,10\}\}$$

This means that the optimal solution, $C_{\min}(n)$, is not found by the minimum expression. The optimal solution must use some coin c, and we can remove this coin to get a (possibly) non-optimal solution of $C_{\min}(n-c)$. So

$$C_{\min}(n-c) \leq C_{\min}(n) - 1$$

Coin change — Optimal Substructure property

Using out assumption (that we are trying to get a contradiction from) and our coin c from our previous slide:

$$C_{\min}(n) < 1 + C_{\min}(n - c)$$

But

$$C_{\min}(n - c) \leq C_{\min}(n) - 1 < (1 + C_{\min}(n - c)) - 1$$

Which gives

$$C_{\min}(n - c) < C_{\min}(n - c)$$

Giving our contradiction.

Coin change — Bottom up implementation

```
def min_coin_bottom_up(n):
if (n == 0):
    return 0
Change =  [sys.maxsize]*(n+1)
Change[0] = 0
for i in range(1,n+1):
    for k in [1,2,5,10]:
        if k <= i :
            current = Change[i - k] + 1
            if current < Change[i] :
                Change[i] = current
return(Change[n])
```

Bottom up complexity

This is going to be an important theme in dynamic programming.
What is the complexity of the bottom up approach?

- There are two nested loops. The inner loop goes through the coins and the previous entries. This takes constant time (or it takes the same time for every loop).
- The outer loop depends on n the value that we are trying to compute. So the algorithm has complexity $O(n)$.

Pseudo Polynomial

This value n is not the input size of the problem, but an integer value. The algorithm is what is known as pseudo-polynomial (more on this later).

The number 256 only needs 8-bits to be represented, so the real input size for 256 would be 8, if added an extra bit I double the number size.

So the algorithm is sort of exponential in the input size, but for small values of n the algorithm is sort of polynomial.

More on this later.

What coins do I use?

- Can you give me an algorithm using dynamic programming to tell me what coins I actually used?
- At first sight this seems a hard problem, but we have solved the easier problem of computing the minimum number of coins needs.
- Once we have computed this number, we can work back from our array and work out how the value was computed.

What coins do I use?

So

- `Change[13]` is 3 because `Change[13-1]` is 2
- `Change[12]` is 2 because `Change[12-2]` is 1
- `Change[10]` is 1 because `Change[10-10]` is 0.

This gives

$$13 = 10 + 2 + 1$$

.

You can either add another data structure to keep track of your choices, or just work backwards from the answer redoing the work.

Coin Change

Somethings that I've not be totally honest about :

- There is more than one solution. Which solution we get depends a bit on how we implement it.
- If you want to generalise to arbitrary coin sets $\{2, 3, 5\}$ what happens if you have a number for which you cannot give change say $n = 1$?

Summary

- Dynamic programming is recursion with caching to avoid recomputing the same solutions over and over again.
- Optimal substructure property is the fact that you can compute your solution by looking at the sub-problems (this is not always true).
- Unlike straightforward recursion that you have already met, in dynamic programming you can split up the problem into smaller sub-problems indexed by something else. This is often the hardest part in coming up with a dynamic programming solution.