Algorithms and Data Structures (ECS529)

Nikos Tzevelekos

Lecture 5

Dynamic Programming

Approaches to algorithm design

There are different approaches in designing an algorithm.

So far, we have seen:

- iterative algorithms (e.g. for-loops on arrays)
- divide-and-conquer algorithms (e.g. quicksort, merge sort)
- recursive algorithms (as above)

Today we are going to look at **Dynamic Programming**

this is a recursive way to build fast algorithms re-using results that we have already computed

Example: Fibonacci

The Fibonacci sequence is a sequence of natural numbers given by:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

-> i.e. it starts with 0, 1,

-> and then every next element is the sum of the last two.

It is important in mathematics, but also found in nature!

In computer science it is the classic example of a problem solved with recursion. We can compute `fib(n)` by:

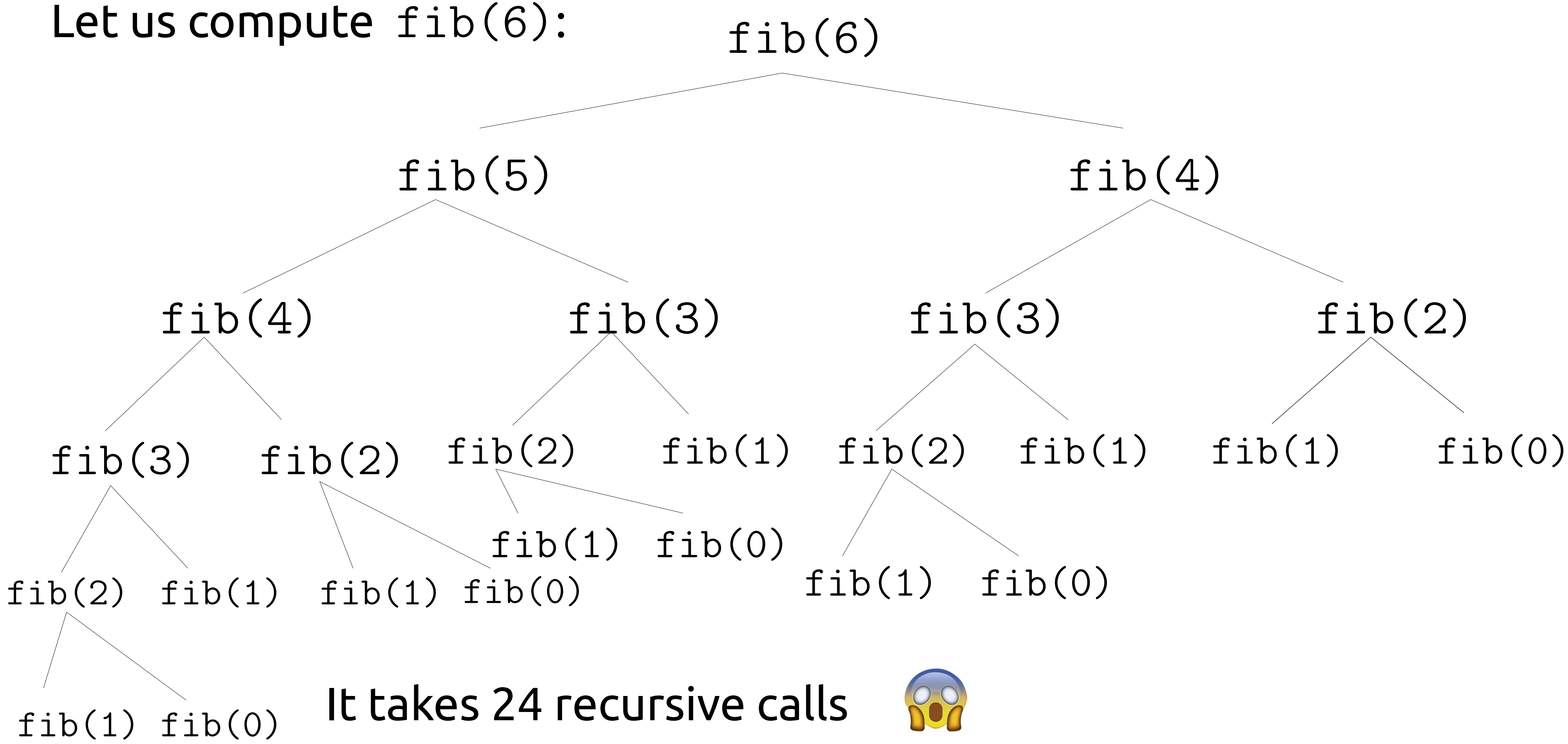
```
def fib(n):
   if n <= 1:
      return n
   return fib(n-1)+fib(n-2)
```

Strength of recursion: algorithm is neat!

Is it efficient?

Though neat, the recursive solution is hugely inefficient.

Let us compute `fib(6)`:



It takes 24 recursive calls 😱

The reason is: we compute the same thing very many times!

Running time is **exponential**: $\Theta(2^{0.694n})$ – worse than any polynomial of n

Memoisation

Idea: avoid repeated computation of intermediate `fib(n)`'s by storing and reusing them.

This is called **memoisation** and can be done by using a storage array (here called `memo`):

```
def fibDP(n):
    memo = [-1 for i in range(n+1)]
    return fibMem(n,memo)

def fibMem(n, memo):
    if memo[n] != -1:
        return memo[n]
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fibMem(n-1,memo)+fibMem(n-2,memo)
    return memo[n]
```

Dynamic programming: recursion + memoisation

So, `fibDP(n)`:

- first creates a storage array `memo` of length `n` and initialises it to -1
- the value -1 simply means: this value has not been stored yet
- we then call `fibMem(n,memo)`

Then, `fibMem(n,memo)`:

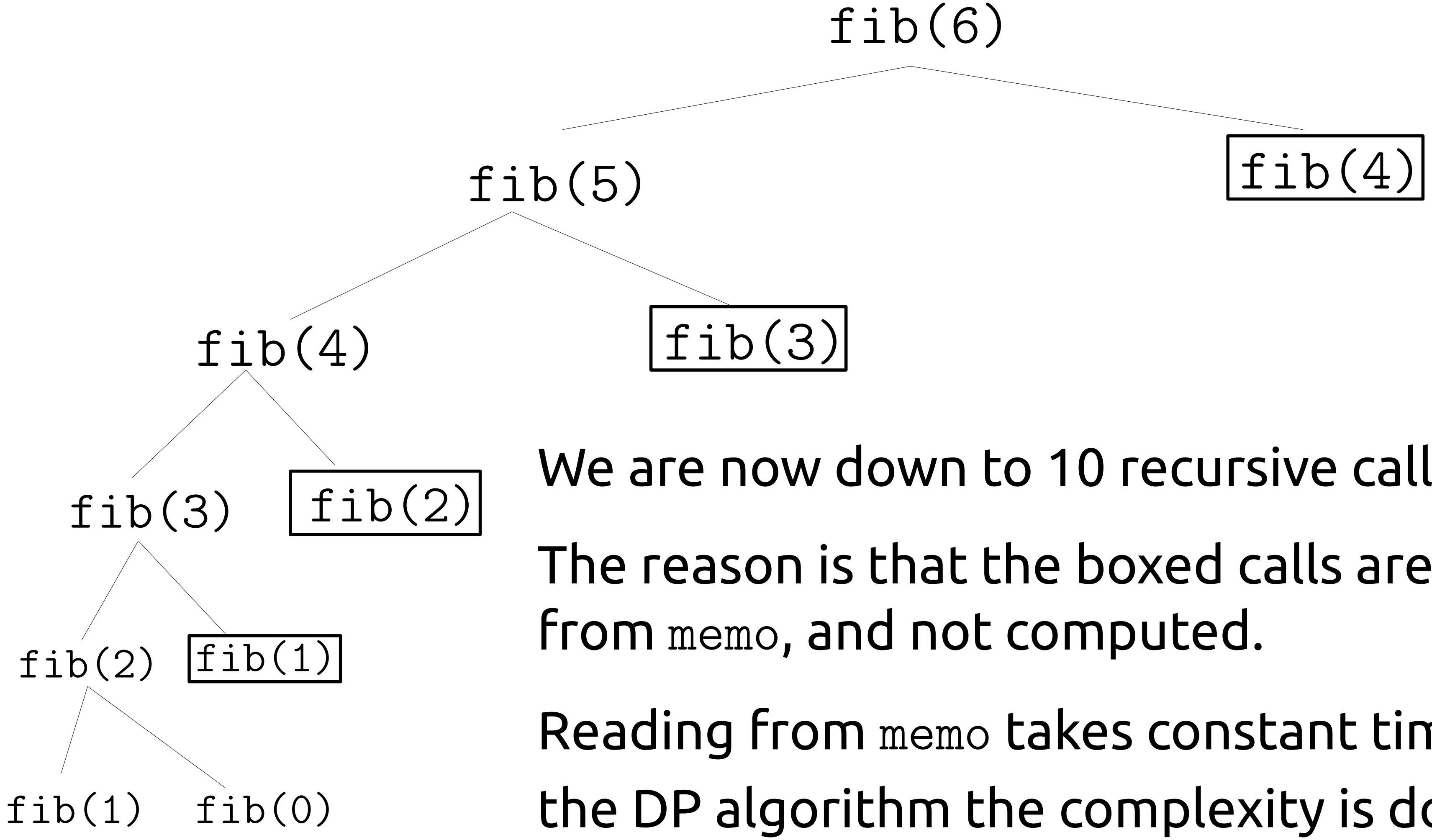
- first checks if `memo[n]` has been already stored – if so it just returns it
- otherwise, it needs to compute `memo[n]` – there are 2 cases:
  - in case `n` is 0 or 1 (base case) we know that `fib(n) = n`
  - otherwise, `fib(n)` is computed by recursively calling `fibMem`

Thus, we use the array to store the results of all recursive calls to `fibMem` so that we only really compute each `fibMem(n)` once!

use recursion but memoise results of recursive calls

Efficiency

Let us compute with the dynamic programming algorithm, i.e. each `fib(n)` is in fact `fibMem(n,memo)`:



We are now down to 10 recursive calls.

The reason is that the boxed calls are fetched from `memo`, and not computed.

Reading from `memo` takes constant time. So with the DP algorithm the complexity is down to $O(n)$.

Exponential speed-up – magic.

A final ingredient

We can make a further optimisation to the algorithm:

- since it uses an array to store recursive call results, we can turn recursion to iteration
- this follows the general idea that iteration is more efficient than recursion as it needs less memory

  (and does not throw runtime errors because of reaching the stack limit for recursive calls)

In practice, we turn the solution from top-down to bottom-up!

Dynamic Programming Bottom-Up

```
def fibDPBU(n):
    memo = [-1 for i in range(n+1)]
    memo[0] = 0
    memo[1] = 1
    for i in range(2,n+1):
        memo[i] = memo[i-1]+memo[i-2]
    return memo[n]
```

Dynamic Programming Bottom-Up

```
def fibDPBU(n):
    memo = [-1 for i in range(n+1)]
    memo[0] = 0
    memo[1] = 1
    for i in range(2,n+1):
        memo[i] = memo[i-1]+memo[i-2]
    return memo[n]
```

we initialise `memo` as before

the base cases of `fibMem` are for `n` <= 1, so we can set these values of `memo` straight away.

for the other cases, we simply use the formula:
`memo[n] = fibMem(n-1,memo)+fibMem(n-2,memo)`
where e.g. instead of `fibMem(i-1,memo)` we use `memo[i-1]`

Bottom-up: In order for the assignment `memo[i] = memo[i-1]+memo[i-2]` to make sense we need to make sure that `memo[i-1]` and `memo[i-2]` have been already been computed!

This is why we need to go bottom-up: from the base cases up to the general ones.

Optimisation problems

Dynamic programming is typically used for optimisation problems that can be solved by recursion, i.e. problems where:

- we try to find a solution
- that is moreover optimal with respect to a given criterion
- and where solutions can be computed recursively.

For example:

- how to give change with least number of coins
- how to find the longest palindromic substring of a string
- how to find the shortest path between two points in a graph

Example: split in least coins

Least Coin Split: given an amount m of money, find the minimum number of coins whose value adds up to m.

Coins are taken from a given array, e.g:

```
coin = [200, 100, 50, 20, 10, 5, 2, 1]
```

You can think e.g. of a vending machine designed to give change using as few coins as possible.

So, here:

- a solution is a number k so that we can pick k coins from above that sum up to m
- an optimal solution is one that has the smallest possible k
- solution can be calculated recursively (how?)

Recursive solution

Suppose we want to split `m` in coins, using coins from the `i`-th coin on:

- there are two cases that we can resolve straight away (base cases):
  1. `m = 0` (i.e. there is nothing to split) -> just return `0`
  2. `i = len(coin)` (we are at the last coin, i.e.1p) -> just return `m`
- otherwise, we argue as follows. We have two options:
  1. we can leave coin `i` out of our split. So, our problem becomes: split `m` in coins, using coins from the `i+1`-th coin on
  2. if `m >= coin[i]`, then we can include coin `i` in our split. We now have: split `m-coin[i]` in coins, using coins from the `i`-th coin on

  We go and solve these new problems **recursively**, and get solutions `withoutIt` and `withIt`. We return the min of `withoutIt` and `withIt`.

Recursive solution in code

```
def coinSplit(m):
    return coinSplitRec(m,0)

def coinSplitRec(m, startCoin):
    if m == 0:
        return 0
    if startCoin == len(coin)-1:
        return m
    withoutIt = coinSplitRec(m,startCoin+1)
    if coin[startCoin] <= m:
        withIt = 1 + coinSplitRec(m-coin[startCoin],startCoin)
        if withIt < withOutIt:
            return withIt
    return withoutIt
```

Recursive solution in code

```
def coinSplit(m):
    return coinSplitRec(m,0)

def coinSplitRec(m, startCoin):
    if m == 0:
        return 0
    if startCoin == len(coin)-1:
        return m
    withoutIt = coinSplitRec(m,startCoin+1)
    if coin[startCoin] <= m:
        withIt = 1 + coinSplitRec(m-coin[startCoin],startCoin)
        if withIt < withOutIt:
            return withIt
    return withoutIt
```

we call the recursive method specifying the amount to split (i.e. `m`) and the coin to start from (the first one, i.e. coin 0)

if the amount to split is 0 then return 0

if we start splitting from the last coin (which we assume is 1) then return `m`

otherwise, we first check the case where we split `m` without using `startCoin`, so we call `coinSplitRec` again on the same `m` and with `startCoin` increased by 1

then we check the case where we use startCoin in the split, so we first check that this can be done

and call `coinSplitRec` on `m` decreased by the value of `startCoin` and with the same `startCoin`. We add 1 to the result to account for this coin being used in the split. We just return the min of the two cases.

Dynamic Programming: add memoisation

To add memoisation, we store the results of recursive calls of `coinSplitRec` so that we don't compute them more than once.

Since these calls have 2 arguments that vary (`m` and `startCoin`), we need an array `memo` of 2 dimensions so that e.g:

`memo[42][5]` stores the result of `coinSplitRec(42,5)`.

We need `memo[i][j]` with `i` from 0 to `m`, and with `j` from 0 to `len(coin)-1`.

```
def coinSplitDP(m):
    memo = [[-1 for j in range(len(coin))] for i in range(m+1)]
    return coinSplitMem(m,0,memo)


def coinSplitMem(m, startCoin, memo):
    # to be filled in
```

Dynamic Programming: add memoisation

```
def coinSplitMem(m, startCoin, memo):
    if memo[m][startCoin] != -1:
        return memo[m][startCoin]
    if m == 0:
        memo[m][startCoin] = 0
    elif startCoin == len(coin)-1:
        memo[m][startCoin] = m
    else:
        withoutIt = coinSplitMem(m,startCoin+1,memo)
        memo[m][startCoin] = withoutIt
        if coin[startCoin] <= m:
            withIt = 1 + coinSplitMem(m-coin[startCoin],startCoin,memo)
            if withIt < withoutIt:
                memo[m][startCoin] = withIt
    return memo[m][startCoin]
```

Dynamic Programming: add memoisation

```
def coinSplitMem(m, startCoin, memo):
    if memo[m][startCoin] != -1:
        return memo[m][startCoin]
    if m == 0:
        memo[m][startCoin] = 0
    elif startCoin == len(coin)-1:
        memo[m][startCoin] = m
    else:
        withoutIt = coinSplitMem(m,start
        memo[m][startCoin] = withoutIt
        if coin[startCoin] <= m:
            withIt = 1 + coinSplitMem(m-
            if withIt < withoutIt:
                memo[m][startCoin] = wit
    return memo[m][startCoin]
```

code is the same as in the recursive solution, with some additions:

- the array `memo` is initialised with 'undefined' (-1) and passed to each recursive call (i.e. call of `coinSplitMem`)
- each call of `coinSplitMem(m,startCoin,memo)` first checks whether `memo[m][startCoin]` is defined, i.e. whether this call has already been computed and stored. If so, it immediately returns.
- the rest of the code of `coinSplitMem` is the same as that of `coinSplitRec`, with the modification that:
  - instead of returning values, store them in `memo`
  - at the end, return the value you computed and stored in `memo`

Dynamic Programming Bottom-Up solution

To make our previous solution bottom-up :

- we start by building an initially empty array `memo`
- we fill in the base values of `memo`:
  - `memo[m][startCoin] = 0` when `m = 0`
  - `memo[m][startCoin] = m` when `startCoin = len(coin)-1`
- we then fill in the rest of `memo` using iteration, i.e. for `memo[m][startCoin]`:
  - we first look up the value of `memo[m][startCoin+1]` and store it in a variable `withoutIt`
  - if `coin[startCoin] <=m` then we look up the value of `memo[m-coin[StartCoin]][startCoin]`, add 1 to it and store it in a variable `withIt`
  - we finally let `memo[m][startCoin]` be the minimum of `withIt` and `withoutIt`.
- We finally return `memo[m][0]` for the initial value of `m`.

Bottom up

To compute `memo[m][startCoin]` we need to have first computed:

`memo[m][startCoin+..]`

`memo[m-..][startCoin]`

i.e. we need to have computed `memo` for all larger values of `startCoin` and for all smaller values of `m`.

So, iteration should go:

- from smaller to larger values of `m`
- from larger to smaller values of `startCoin`

Dynamic Programming Bottom-Up solution

```
def coinSplitDPBU(mInit):
    memo = [[-1 for j in range(len(coin))] for i in range(mInit+1)]
    for i in range(len(coin)):
        memo[0][i] = 0
    for m in range(mInit+1):
        memo[m][len(coin)-1] = m
    for m in range(1,mInit+1):
        for startCoin in range(len(coin)-2,-1,-1):
            withoutIt = memo[m][startCoin+1]
            memo[m][startCoin] = withoutIt
            if coin[startCoin] <= m:
                withIt = 1 + memo[m-coin[startCoin]][startCoin]
                if withIt < withoutIt:
                    memo[m][startCoin] = withIt
    return memo[mInit][0]
```

Complexity analysis for least coin split (naive rec.)

```
def coinSplit(m):
    return coinSplitRec(m,0)


def coinSplitRec(m, startCoin):
    if m == 0:
        return 0
    if startCoin == len(coin)-1:
        return m
    withoutIt = coinSplitRec(m,startCoin+1)
    if coin[startCoin] <= m:
        withIt = 1 + coinSplitRec(m-coin[startCoin],
                                  startCoin)
        if withIt < withOutIt:
            return withIt
    return withoutIt
```

The inputs here are `m` and `coin` (though we left `coin` implicit). Suppose `coin` has length n.

Each recursive call has a fixed max number of steps, so it is enough to simply count the number of recursive calls.

We can show that the number of recursive calls is $\Theta(\mathtt{m}^{n-1})$.

This means that:

- if `coin` is part of the input (so, can be very long) then the time complexity of `coinSplit` is exponential
- if `coin` is fixed, then the time complexity of `coinSplit` is polynomial.

  E.g. if `coin` = `[200,100,50,20,10,5,2,1]` then the time complexity is $\Theta(\mathtt{m}^7)$.

Complexity analysis for least coin split (DP)

```
def coinSplitDP(m):
    memo = [[-1 for j in range(len(coin))] for i in range(m+1)]
    return coinSplitMem(m,0,memo)


def coinSplitMem(m, startCoin, memo):
    if memo[m][startCoin] != -1:
        return memo[m][startCoin]
    if m == 0:
        memo[m][startCoin] = 0
    elif startCoin == len(coin)-1:
        memo[m][startCoin] = m
    else:
        withoutIt = coinSplitMem(m,startCoin+1,memo)
        memo[m][startCoin] = withoutIt
        if coin[startCoin] <= m:
            withIt = 1 + coinSplitMem(m-
coin[startCoin],startCoin,memo)
            if withIt < withoutIt:
                memo[m][startCoin] = withIt
    return memo[m][startCoin]
```

The inputs here are `m` and `coin` (though we left `coin` implicit). Suppose `coin` has length n.

Each recursive call has a fixed max number of steps, so it is enough to simply count the number of recursive calls.

Because of memoisation, we can assume that `coinSplitMem(i,startCoin,memo)` **will be called at most once**, for each different value of `i` and `startCoin`.

how many different values can `i` take? `m+1`

how many can `startCoin` take? n

---

so, max number of recursive calls = $n\,(\mathtt{m+1})$

Therefore, the time complexity is $\Theta(\mathtt{m}\,n)$.

Longest palindromic substring

Longest Palindrome: given a string `s`, find the longest, not necessarily contiguous, palindromic substring contained in `s`.

For example:

- on input `010` it should return `010`
- on input `01000` it should return `0000`
- on empty string input it should return the empty string
- on input `12312323321` it should return ?

Longest palindromic substring

Longest Palindrome: given a string `s`, find the longest, not necessarily contiguous, palindromic substring contained in `s`.

Recursive solution

- if the string `s` has length no more than 1 $\Longrightarrow$ return `s`
- otherwise, there are two cases:
  - `s` begins-ends with same character, e.g. `s = c + sMid + c`
    $\Longrightarrow$ return `c` + longest palindrome of `sMid` + `c`
  - `s` begins-ends with different characters, e.g. `s = c1 + sMid + c2`
    $\Longrightarrow$ return the longest of these strings:
    - the longest palindrome of `s1 = c1 + sMid`
    - the longest palindrome of `s2 = sMid + c2`

Longest palindrome recursive solution

Longest Palindrome: given a string `s`, find the longest, not necessarily contiguous, palindromic substring contained in `s`.

```
def longestPalin(s):
    return palinRec(s,0,len(s))

def palinRec(s, lo, hi):
    if hi-lo <= 1:
        return s[lo:hi]
    if s[lo] == s[hi-1]:
        return s[lo]+palinRec(s,lo+1,hi-1)+s[hi-1]
    s1 = palinRec(s,lo,hi-1)
    s2 = palinRec(s,lo+1,hi)
    if len(s1) < len(s2):
        return s2
    return s1
```

Add memoisation

To add memoisation, we store the results of recursive calls of `palinRec` so that we don't compute them more than once.

Since these calls have 2 arguments that vary (`lo` and `hi`), we need an array `memo` of 2 dimensions so that e.g:

`memo[2][9]` stores the result of `palinRec(s,2,9)`.

```
def longestPalinDP(s):
    if len(s) == 0:
        return s
    memo = [[None for i in range(len(s)+1)] for j in range(len(s))]
    return palinMem(s,0,len(s),memo)

def palinMem(s, lo, hi, memo):
    # to be filled in
```

Add memoisation

```
def palinMem(s, lo, hi, memo):

    if memo[lo][hi] != None:
        return memo[lo][hi]
    if hi-lo <= 1:
        memo[lo][hi] = s[lo:hi]
    else:
        if s[lo] == s[hi-1]:
            memo[lo][hi] = s[lo]+palinMem(s,lo+1,hi-1,memo)+s[hi-1]
        else:
            s1 = palinMem(s,lo,hi-1,memo)
            s2 = palinMem(s,lo+1,hi,memo)
            if len(s1) < len(s2):
                memo[lo][hi] = s2
            else:
                memo[lo][hi] = s1
    return memo[lo][hi]
```

Memoisation + bottom-up

```
def longestPalinDPBU(s):
    if len(s) == 0:
        return s
    memo = [[None for i in range(len(s)+1)] for j in range(len(s))]
    for lo in range(len(s)-1,-1,-1):
        for hi in range(lo,len(s)+1):
            if hi-lo <= 1:
                memo[lo][hi] = s[lo:hi]
            elif s[lo] == s[hi-1]:
                memo[lo][hi] = s[lo]+memo[lo+1][hi-1]+s[hi-1]
            else:
                s1 = memo[lo][hi-1]
                s2 = memo[lo+1][hi]
                if len(s1) < len(s2):
                    memo[lo][hi] = s2
                else:
                    memo[lo][hi] = s1
    return memo[lo][hi]
```

An old example

This is not the first time you see dynamic programming…

Recall Context-Free Grammars in Chomsky Normal Form:

$$
\begin{aligned}
S &\rightarrow U_0 X \mid \varepsilon \\
X &\rightarrow Y U_1 \mid 1 \\
Y &\rightarrow U_0 X \\
U_0 &\rightarrow 0 \\
U_1 &\rightarrow 1
\end{aligned}
$$

(CNF form of $S \rightarrow 0S1 \mid \varepsilon$)

In order to check whether a given word is produced by the grammar, we can use the CYK algorithm.

The algorithm builds a table where it stores variables that can produce subwords of the word we want to check.

It (re-)uses them to finally find variables producing the whole word.

Recursive parsing

Suppose we are given a grammar G in CNF and a non-empty string s.

To check G that accepts s we can compute the variables which produce all substrings of s. That is, for each non-empty substring s' of s:

- if s' is a single character, say c, return the set of all variables X for which we have a rule $X \to c$
- otherwise, for all possible binary splittings of , i.e. $s' = s_1 s_2$,
  - compute the sets Set_1 and Set_2 producing s_1 and s_2 respectively,
  - return the set of all variables X for which we have a rule $X \to X_1 X_2$ such that X_1 is in Set_1 and X_2 is in Set_2.

This algorithm works, but can lead to computing several times the variables producing the same substring.

It can be optimised using dynamic programming: we can use a table in order to remember intermediate sets we compute. This is what the CYK algorithm does! (recap example in next slides)

CYK parsing example

Consider the CNF grammar on the left below and input word 0011:

$$
\begin{aligned}
S &\rightarrow U_0 X \mid \varepsilon \\
X &\rightarrow Y\,U_1 \mid 1 \\
Y &\rightarrow U_0 X \\
U_0 &\rightarrow 0 \\
U_1 &\rightarrow 1
\end{aligned}
$$

In the first step we decide the first level.

Once we compute the box above 0, we do not need to compute it again, and similarly for that above 1.

Above 0 we put U_0, because of the rule $U_0 \rightarrow 0.$

Above 1 we put U_1 and X, because of $U_1 \rightarrow 1$ and $X \rightarrow 1$

4th				
3rd				
2nd				
1st	U_0	U_0	U_1, X	U_1, X
	0	0	1	1

CYK parsing example

Consider the CNF grammar on the left below and input word 0011:

$$
\begin{aligned}
S &\rightarrow U_0X \mid \varepsilon \\
X &\rightarrow YU_1 \mid 1 \\
Y &\rightarrow U_0X \\
U_0 &\rightarrow 0 \\
U_1 &\rightarrow 1
\end{aligned}
$$

We fill in the cells of the 2nd row with the variables producing pairs of variables that are immediately below and to the right.

I.e. we put in this cell Y (because of $Y \rightarrow U_0X$) and S (because of $S \rightarrow U_0X$).

And there are no other cells to fill in the second row.

4th				
3rd				
2nd	-	Y, S	-	
1st	U_0	U_0	U_1, X	U_1, X
	0	0	1	1

CYK parsing example

Consider the CNF grammar on the left below and input word 0011:

$$
\begin{aligned}
S &\to U_0X \mid \varepsilon \\
X &\to YU_1 \mid 1 \\
Y &\to U_0X \\
U_0 &\to 0 \\
U_1 &\to 1
\end{aligned}
$$

We fill in the cells of the 3nd row and above with the variables producing pairs of variables that are below and to the right, but taking into account the length of the words that these produce.

4^{th}				
3^{rd}	-	X		
2^{nd}	-	Y, S	-	
1^{st}	U_0	U_0	U_1, X	U_1, X
	0	0	1	1

I.e. we put in this cell X (because of $X \to YU_1$), and there are no other cells to fill in the third row.

CYK parsing example

Consider the CNF grammar on the left below and input word 0011:

$$
\begin{aligned}
S &\to U_0 X \mid \varepsilon \\
X &\to Y U_1 \mid 1 \\
Y &\to U_0 X \\
U_0 &\to 0 \\
U_1 &\to 1
\end{aligned}
$$

We fill in the cells of the 3nd row and above with the variables producing pairs of variables that are below and to the right, but taking into account the length of the words that these produce.

4th	Y, S			
3rd	-	X		
2nd	-	Y, S	-	
1st	U_0	U_0	U_1, X	U_1, X
	0	0	1	1

I.e. we put in this cell Y (because of $Y \to U_0 X$) and S (because of $S \to U_0 X$).

So, in this case, the grammar accepts 0011.

Summary

Dynamic programming is a great technique for writing efficient algorithms starting from inneficient recursive solutions

When it is applicable, it can lead to exponential speedup

It is applicable when there are opportunities for re-using intermediate/recursive results

It is typically used for optimisation problems: e.g. the longest substring with a specific property

Apart from speeding up recursive algorithms, it also allows one to transform them to iterative ones (by doing recursion ‘bottom-up’)

Exercises

1. Let us call Pythagorean the following sequence:

   0, 1, 1, 2, 5, 29, 866, ...

   where each number in the sequence is the sum of squares of the two numbers preceding it in the sequence. That is, we can compute its elements by:

$$pyth(n) = \begin{cases} n & \text{if } n \leq 1 \\ pyth(n\text{-}1)^2 + pyth(n\text{-}2)^2 & \text{otherwise} \end{cases}$$

   Write a recursive function `pyth(n)` that, given `n`, returns the `n`-th number in the sequence.

2. Change your function `pyth(n)` to a dynamic programming one, by using memoisation.
3. Change your function again to one that uses DP in its bottom-up version, i.e. using iteration.
4. We have a bag and we want to fill it with books. The bag can take at most `w` kilos of weight, while the weights of our books are given by an array `bkWeight` (e.g. `bkWeight[0]` is the weight of the first book, etc.). Each book has a value, given by an array `bkVal` (e.g. `bkVal[0]` is the value of the first book, etc.). Write a DP function

   `def maxBooksValDP(w, bkWeight, bkVal)`

   which returns the maximum value of books that we can fill our bag with. Assume `bkWeight` is sorted.

   Start from this recursive solution:

```
def maxBooksVal(w, bkWeight, bkVal):
    return maxBooksRec(w,bkWeight,bkVal,0)

def maxBooksRec(w, bkWeight, bkVal, startBk):
    if startBk == len(bkWeight) or w < bkWeight[startBk]:
        return 0
    withIt = bkVal[startBk]+maxBooksRec(w-bkWeight[startBk],
                                        bkWeight,bkVal,startBk+1)
    withoutIt = maxBooksRec(w,bkWeight,bkVal,startBk+1)
    if withIt > withoutIt:
        return withIt
    return withoutIt
```

5. Change your solution to question 4 and make it bottom-up.
6. Change your solution to question 4 so that, instead of returning just the maximum value, it returns the maximum value and an array containing all books that can be included in the bag and add up to this value. That is, it should return an array `[maxVal,inBag]` where `maxVal` is the maximum book value and `inBag` is the array of the books included in the bag in order to achieve `maxVal`.

   You can use the function `append(A,k)` that returns a new array with all elements of `A` and with `k` added at the end.

Exercises – solutions

1. This is a variation of the Fibonacci example. Here is a recursive solution:

```
def pyth(n):
    if n <= 1:
        return n
    return pyth(n-1)**2 + pyth(n-2)**2
```

2. Here is a DP solution:

```
def pythDP(n):
    memo = [-1 for i in range(n+1)]
    return pythMem(n,memo)

def pythMem(n,memo):
    if memo[n] != -1:
        return memo[n]
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = pythMem(n-1,memo)**2 +
pythMem(n-2,memo)**2
    return memo[n]
```

3. Here is a DP botton-up solution:

```
def pythDPBU(n):
    memo = [-1 for i in range(n+1)]
    memo[0] = 0
    memo[1] = 1
    for i in range(2,n+1):
        memo[i] = memo[i-1]**2 + memo[i-2]**2
    return memo[n]
```

4. Here is a DP solution:

```
def maxBooksValDP(w, bkWeight, bkVal):
    memo = [[-1 for in range(len(bkWeight)+1)] for j in range(w+1)]
    return maxBooksRec(w,bkWeight,bkVal,0,memo)

def maxBooksMem(w, bkWeight, bkVal, startBk, memo):
    if memo[w][startBk] != -1:
        return memo[w][startBk]
    if startBk == len(bkWeight) or w < bkWeight[startBk]:
        memo[w][startBk] = 0
    else:
        withIt = bkVal[startBk] + maxBooksMem(w-
bkWeight[startBk],bkWeight,bkVal,startBk+1,memo)
        withoutIt = maxBooksMem(w,bkWeight,bkVal,startBk+1,memo)
        if withIt > withoutIt:
            memo[w][startBk] = withIt
        else:
            memo[w][startBk] = withoutIt
    return memo[w][startBk]
```

5. Here is a DP bottom-up solution:

```
def maxBooksValDPBU(wInit, bkWeight, bkVal):
    memo = [[-1 for j in range(len(bkWeight)+1)] for i in range(wInit+1)]
    for w in range(wInit+1):
        for startBk in range(len(bkWeight),-1,-1):
            if startBk == len(bkWeight) or w < bkWeight[startBk]:
                memo[w][startBk] = 0
            else:
                withIt = bkVal[startBk] + memo[w-bkWeight[startBk]][startBk+1]
                withoutIt = memo[w][startBk+1]
                if withIt > withoutIt:
                    memo[w][startBk] = withIt
                else:
                    memo[w][startBk] = withoutIt
    return memo[wInit][0]
```