Übungen zu Algorithmen und Programmentwicklung für die Biologische Chemie

Michael T. Wolfinger
(based on slides by Sven Findeiß)

Research Group Bioinformatics and Computational Biology
Department for Theoretical Chemistry
Währingerstrasse 17, 1090 Vienna, Austria

Sommersemester 2021

Dynamic Progamming

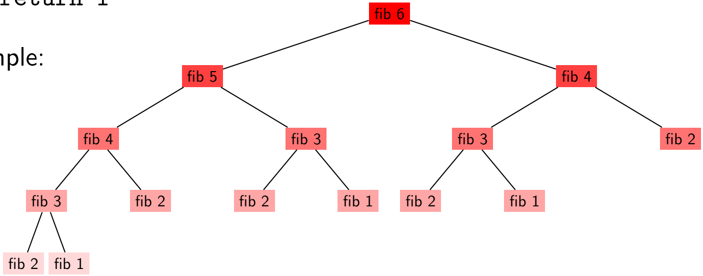
- Solving recursion equations (with overlapping subproblems) efficiently
- Optimization employing 'optimal substructure'

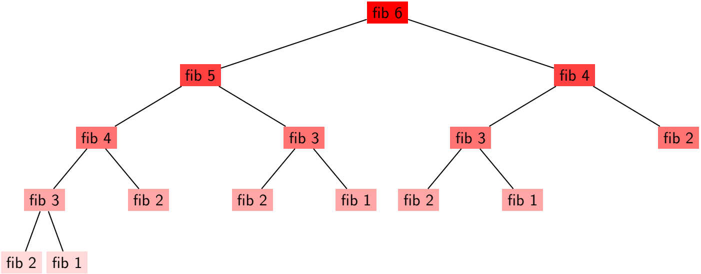
Example 1: Fibonacci Series

The Fibonacci Series of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, ... can be defined recursively:

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

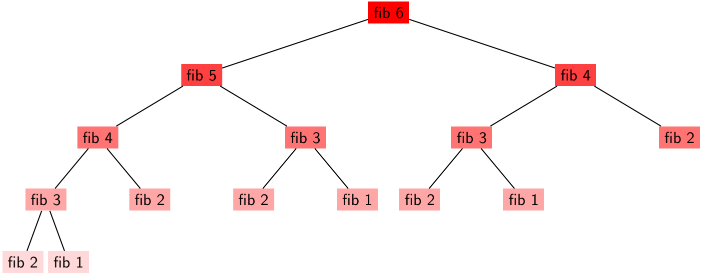
Example:





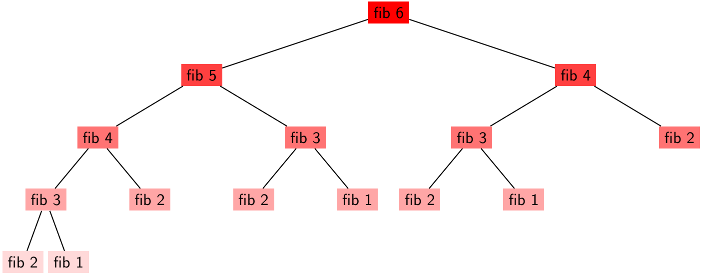
Building the solution space we realize:

1. The run-time of the recursion scales exponentially ($\sim 2^n$) with the input size.
2. Many (sub)solutions are calculated over and over again ($5 \times$ `fib 2`, $3 \times$ `fib 3`, $2 \times$ `fib 4`).
3. Avoid redundancy: tabulate subsolutions!



Building the solution space we realize:

1. The run-time of the recursion scales exponentially ($\sim 2^n$) with the input size.
2. Many (sub)solutions are calculated over and over again ($5 \times$ `fib 2`, $3 \times$ `fib 3`, $2 \times$ `fib 4`).
3. Avoid redundancy: tabulate subsolutions!



Building the solution space we realize:

1. The run-time of the recursion scales exponentially ($\sim 2^n$) with the input size.
2. Many (sub)solutions are calculated over and over again ($5 \times$ `fib 2`, $3 \times$ `fib 3`, $2 \times$ `fib 4`).
3. Avoid redundancy: tabulate subsolutions!

Dynamic Programming = Recursion + Memoization

Dynamic Programming = Recursion + Memoization

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

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    elif n <= 2:
        f = 1
    else:
        f = fib(n-1)
          + fib(n-2)
    memo[n] = f
    return f
```

Stop and think:

- How much faster can one compute fib using DP?
- How much space do we need for DP-fib?

Dynamic Programming = Recursion + Memoization

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

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    elif n <= 2:
        f = 1
    else:
        f = fib(n-1)
            + fib(n-2)
    memo[n] = f
    return f
```

Stop and think:

- How much faster can one compute fib using DP?
- How much space do we need for DP-fib?

Recursive (top-down) vs. iterative (bottom-up) Fibonacci

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    elif n <= 2:
        f = 1
    else:
        f = fib(n-1)
            + fib(n-2)
    memo[n] = f
    return f
```

```
memo = {}
def fib(n):

    for i in range(1,n+1):
        if i <= 2:
            f = 1
        else:
            f = memo[i-1]
                + memo[i-2]
        memo[i] = f
    return memo[n]
```

DP = Order + Tabulation (Reuse)

Recursive (top-down) vs. iterative (bottom-up) Fibonacci

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    elif n <= 2:
        f = 1
    else:
        f = fib(n-1)
            + fib(n-2)
    memo[n] = f
    return f
```

```
memo = {}
def fib(n):

    for i in range(1,n+1):
        if i <= 2:
            f = 1
        else:
            f = memo[i-1]
                + memo[i-2]
        memo[i] = f
    return memo[n]
```

DP = Order + Tabulation (Reuse)

Optimization by DP: Levenshtein Distance

Definition: *Levenshtein distance of strings a and b* := minimal cost of transforming *a* into *b* by edit operations "replace", "insert", "delete" each of cost 1.

Example: `AUTO` ⇒ `RAD`, `MOTORRAD` ⇒ `FAHRRAD`

As edit sequence:

`AUTO` → `AUT` → `AUD` → `AD` → `RAD` (4)

`MOTORRAD` → `MOTHRRAD` → `MOAHRRAD` → `MFAHRRAD` → `FAHRRAD` (4)

or as alignment:

```
-AUTO        MOTORRAD
RA-D-        -FAHRRAD
```

NOTE: "Levenshtein Distance" has *optimal substructure*:
If `-AUTO` / `RA-D-` is optimal, then it's subsolution `-AUT` / `RA-D` (of subproblem 'AUT' vs. 'RAD') must be optimal.

Optimization by DP: Levenshtein Distance

Definition: *Levenshtein distance of strings a and b* := minimal cost of transforming *a* into *b* by edit operations "replace", "insert", "delete" each of cost 1.

Example: `AUTO` ⇒ `RAD`, `MOTORRAD` ⇒ `FAHRRAD`

As edit sequence:

`AUTO` → `AUT` → `AUD` → `AD` → `RAD` (4)

`MOTORRAD` → `MOTHRRAD` → `MOAHRRAD` → `MFAHRRAD` → `FAHRRAD` (4)

or as alignment:

```
-AUTO        MOTORRAD
RA-D-        -FAHRRAD
```

NOTE: "Levenshtein Distance" has *optimal substructure*:
If

```
-AUTO
RA-D-
```

is optimal, then it's subsolution

```
-AUT
RA-D
```

(of subproblem 'AUT' vs. 'RAD') must be optimal.

Calculating the Levenshtein distance of a and b

- build $n \times m$-matrix D
  where: $D_{ij} :=$ distance of $a_1, \ldots, a_i$ and $b_1, \ldots, b_j$
- calculate each D_{ij} from optimal partial solutions

Recursion:

$$D_{0,0} = 0; D_{0,j} = j; D_{i,0} = i$$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{otherwise} \end{cases} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

Recursion:

$$D_{0,0} = 0; D_{0,j} = j; D_{i,0} = i$$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{otherwise} \end{cases} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

	j	0	1	2	3
i		-	R	A	D
0	-	0	1	2	3
1	A	1			
2	U	2			
3	T	3			
4	O	4			

Recursion:

$$D_{0,0} = 0; D_{0,j} = j; D_{i,0} = i$$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{otherwise} \end{cases} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

	j	0	1	2	3
i		-	R	A	D
0	-	0	1	2	3
1	A	1			
2	U	2			
3	T	3			
4	O	4			

	j	0	1	2	3
i		-	R	A	D
0	-	**0**	**1**	2	3
1	A	1	1	**1**	2
2	U	2	2	2	**2**
3	T	3	3	3	**3**
4	O	4	4	4	**4**

Recursion:

$$D_{0,0} = 0; D_{0,j} = j; D_{i,0} = i$$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{otherwise} \end{cases} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

	j	0	1	2	3
i		-	R	A	D
0	-	0	1	2	3
1	A	1			
2	U	2			
3	T	3			
4	O	4			

	j	0	1	2	3
i		-	R	A	D
0	-	**0**	**1**	2	3
1	A	1	1	**1**	2
2	U	2	2	2	**2**
3	T	3	3	3	**3**
4	O	4	4	4	**4**

$D_{n,m}$ contains the distance of a and b.

Recursion:

$$D_{0,0} = 0;\ D_{0,j} = j;\ D_{i,0} = i$$

$$D_{i,j} = min \begin{cases} D_{i-1,j-1} + \begin{cases} 1 & \text{if } a_i \neq b_j \\ 0 & \text{otherwise} \end{cases} \\ D_{i-1,j} + 1 \\ D_{i,j-1} + 1 \end{cases}$$

	j	0	1	2	3
i		-	R	A	D
0	-	0	1	2	3
1	A	1			
2	U	2			
3	T	3			
4	O	4			

	j	0	1	2	3
i		-	R	A	D
0	-	**0**	**1**	2	3
1	A	1	1	**1**	2
2	U	2	2	2	**2**
3	T	3	3	3	**3**
4	O	4	4	4	**4**

$D_{n,m}$ contains the distance of a and b.

Tracing back the optimal choices from $D_{n,m}$ to $D_{0,0}$ yields *some* optimum alignment of a and b.

Assignment A3: Optimization by DP

Go to:

`https://github.com/TBIAPBC/APBC2021/tree/master/A3`

Happy hacking!