ETH zürich

Departement Informatik

Programming and Problem-Solving

Dynamic Programming

Dennis Komm

Spring 2021 – Mai 6, 2021

Recursive Sorting and Searching

$\mathcal{O}(n \log_2 n)$ Sorting Algorithms

Iterative Mergesort

8	3	11	5	66	2	4	47	7

3	8		1	5		2	6		4	7

1	3	5	8		2	4	6	7

1	2	3	4	5	6	7	8

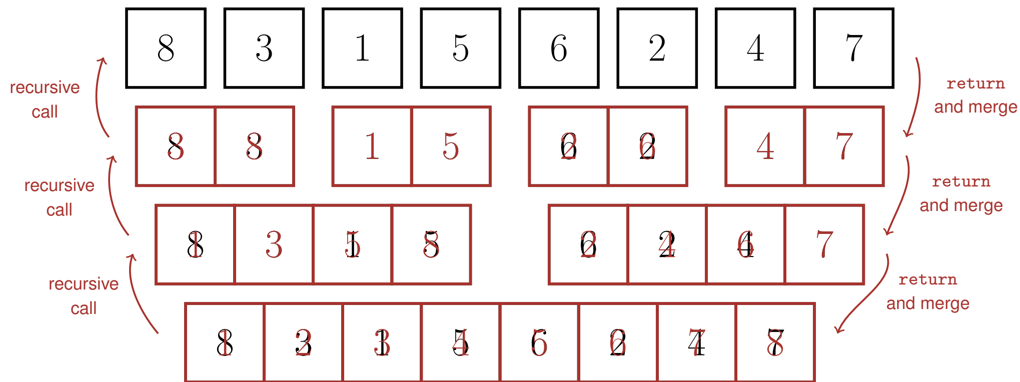
```
[[1, 2, 3, 4, 5, 6, 7, 8]]
```


Iterative Mergesort

Centerpiece is the function `merge()` which merges two sorted lists

```
def merge(leftdata, rightdata):
    result = []
    while len(leftdata) > 0 and len(rightdata) > 0:
        if leftdata[0] > rightdata[0]:
            result.append(rightdata.pop(0))
        else:
            result.append(leftdata.pop(0))
    return result + leftdata + rightdata
```

Recursive Mergesort



Recursive Mergesort

Mergesort as a recursive Python function

- that takes a list as parameter
- splits it in the middle into two lists
- calls the algorithm recursively on these lists
- merges the lists that are sorted this way, and returns them

Recursive Mergesort

```
def mergesort(data):
    if len(data) <= 1:
        return data
    mid = len(data) // 2
    leftdata = mergesort(data[:mid])
    rightdata = mergesort(data[mid:])
    result = []
    while len(leftdata) > 0 and len(rightdata) > 0:
        if leftdata[0] > rightdata[0]:
            result.append(rightdata.pop(0))
        else:
            result.append(leftdata.pop(0))

    return result + leftdata + rightdata
```


Recursive Sorting and Searching

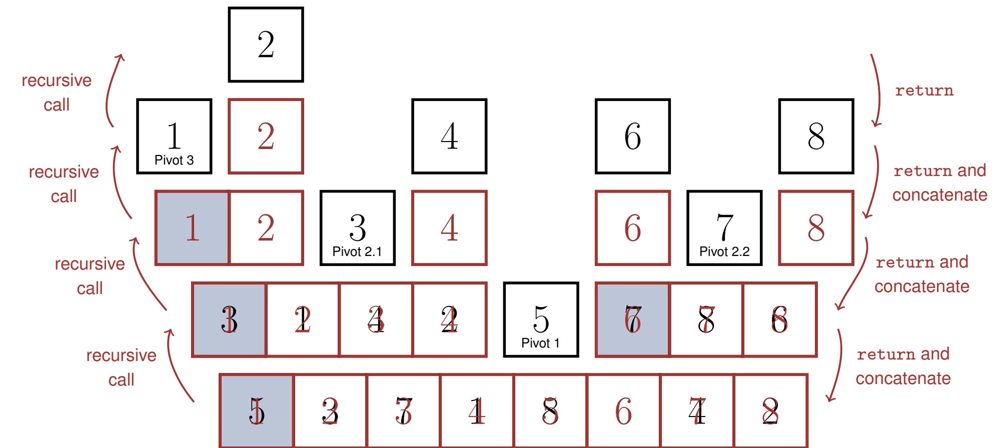
$\mathcal{O}(n \log_2 n)$ Sorting Algorithms – Quicksort

Recursive Quicksort

- One of the best-known sorting algorithms
- Worst-case time complexity in $\mathcal{O}(n^2)$
- But can be **randomized** at a specific place
- Expected time complexity in $\mathcal{O}(n \log_2 n)$
- Very good time complexity in practice

> - Pick arbitrary **pivot element** (we always take the first one)
> - Create a list with smaller and one with larger elements
> - Call algorithm recursively on these lists
> - Concatenate lists that are sorted this way

Recursive Quicksort



Recursive Quicksort

Quicksort as a recursive Python function

- that takes a list `data`
- chooses the first element of `data` as pivot element
- creates a list with smaller and a list with larger elements
- calls the algorithm recursively on these lists
- concatenates and returns the lists that are sorted this way and the pivot element

Recursive Quicksort

```
def quicksort(data):
    if len(data) <= 1:
        return data
    else:
        pivot = data[0]
        leftdata = [i for i in data[1:] if i < pivot]
        rightdata = [i for i in data[1:] if i >= pivot]
        return quicksort(leftdata) + [pivot] + quicksort(rightdata)
```

Dictionaries

Python Lists

- Access position `i` with `[i]`
- Add element `x` at the end using `append(x)`
- Remove element at beginning or end with `pop(0)` respectively `pop()`
- Add or remove element at position `i` with `insert(i,x)` or `pop(i)`
- List Comprehensions
- Test whether element is in list with `in`
- Iterate over all elements with `for` loop
- Generate sublist from position `i` to `j` with `[i:j+1]`
- ...

Restriction through access via index, e.g., if all data are associated with indices, but there is not data for all possible indices

Python Dictionaries

Key-Value pairs
Access not through index, but self-defined "key"

- Partially similar functionality as lists
- ... but data is not sorted by indices
- Initialization with curly brackets

```
data = {}
```

- Keys and values are separated by colon

```
data = {10 : "Wert 1", 16 : "Wert 2", 39 : "Wert 3" }
data = {"eins" : "Wert 1", "zwei" : "Wert 2", "pi" : 3.14 }
```

- Access value with key `key` with `[key]`

```
print(data[16])
```


Fibonacci Numbers

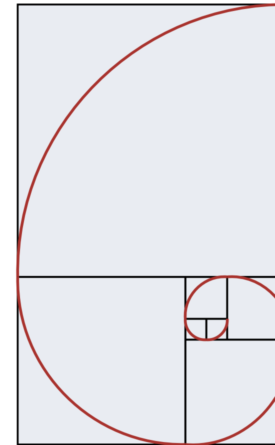
Fibonacci Numbers

- The sequence of Fibonacci numbers is defined as

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \dots,$$

- A number of the sequence is given by the sum of its two predecessors; the first two numbers are both 1
- Can be found in many natural phenomena… or at Zurich main station

Fibonacci Numbers



- Computation can be carried out iteratively or recursively
- Recursively defined as

$$\text{fib}(1) = 1, \text{fib}(2) = 1$$

and

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

- Can be implemented directly in Python

Exercise – Computing Fibonacci Numbers Recursively

Implement a recursive function that

- takes a parameter `n`
- and returns the `n`th Fibonacci number

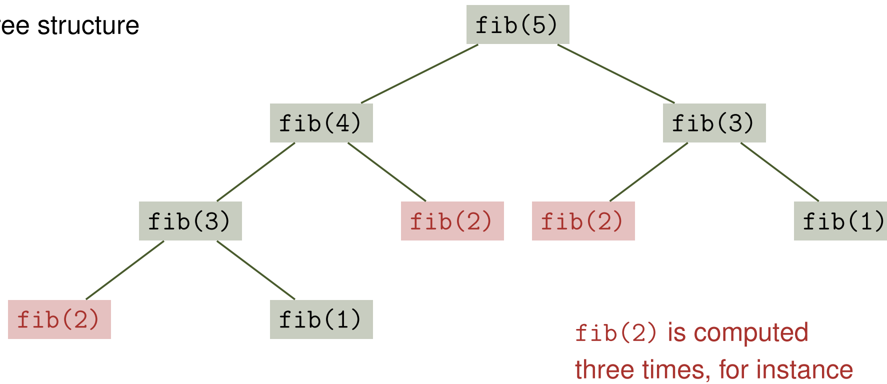
Then output the first 20 Fibonacci numbers

Computing Fibonacci Numbers Recursively

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

for i in range(1, 21):
    print(fib(i), end=" ")
```

Computing Fibonacci Numbers Recursively



Memoization

Memoization

- Function `fib` is called repeatedly with identical parameter values
- Recursion is a lot slower than iteration
- This problem did not appear with binary search or computing the factorial, as the calls were linear
- With Mergesort and Quicksort we also had a tree structure, but disjoint calls

- `fib(n)` calls `fib(n-1)` and `fib(n-2)`
- `fib(n-1)` again calls `fib(n-2)`
- `fib(n-2)` calls the whole subtree both times

Memoization

- Instead of computing values multiple times, store and reuse them
- Every function call first checks whether value has already been calculated
  - If it is, the value is not computed again
  - If it is not, the value is newly computed and stored
- Apart from that, principle of the algorithm stays the same
- Store values in dictionary
- This can be a global variable or passed as parameter

Exercise – Fibonacci Numbers with Memoization

Implement a recursive function that

- takes a parameter `n`
- and returns the `n`th Fibonacci number
- while using a dictionary to implement memoization, looking up the given value using `in`

Then output the first 200 Fibonacci numbers

Fibonacci Numbers with Memoization

```
memo = {1: 1, 2: 1}

def fib(n):
    if n in memo:
        return memo[n]
    else:
        memo[n] = fib(n-1) + fib(n-2)
        return memo[n]


for i in range(1, 201):
    print(fib(i))
```


Similarity of DNA

Similarity of DNA

Find method to compare different DNA molecules

- Search in gene (or protein) database
- Creation of phylogenetic trees
- Problem appearing in DNA sequencing

- Find data structure for molecules
- Define a reasonable similarity measure
- Design an algorithm to compute the similarity with respect to the measure efficiently

Modelling the Data – Molecules as Strings

Representation as strings
DNA are chainlike molecules that consist of repeated building blocks (cytosine, guanine, adenine, and thymine)

Alignments – Similarity Measure

Similarity measure should reflect **common changes** in DNA sequences

- Exchange of single bases of amino acids
- Insertion of removal of short subsequences

Alignments: Write both strings below each other, insert gaps at arbitrary positions

Input: Strings $s =$ GACGATTATG and $t =$ GATCGAATAG

Possible alignments

$s' =$ GA–CGATTATG $s'' =$ GAC–GATTATG $s''' =$ GACGAT——TA–TG

$t' =$ GATCGAATA–G $t'' =$ GATCGAATAG– $t''' =$ ——GATCGAATAG——

Two consecutive gaps do not make sense and are therefore assumed not appear

Alignments – Similarity Measure

Idea for penalties

- Evaluate alignment column by column, then sum over all columns
- Column with gap induces penalty g
- Column with letters a and b induces penalty $p(a, b)$
- $p(a, b)$ is zero for $a = b$ and large for $a \neq b$
- **Goal:** Minimize penalty
- Example for penalties: **edit distance**

Edit Distance

Levenshtein, 1966

Count mismatches and gaps, i.e.,

- $g = 1$
- $p(a, a) = 0$, and
- $p(a, b) = 1$ for $a \neq b$

Input: Strings $s =$ GACGATTATG and $t =$ GATCGAATAG

Possible alignments

$s' =$ GA–CGATTATG $s'' =$ GAC–GATTATG $s''' =$ GACGAT——TA–TG

$t' =$ GATCGAATA–G $t'' =$ GATCGAATAG– $t''' =$ ——GATCGAATAG——

Edit distance: $d_{\text{edit}}(s', t') = 3$ $d_{\text{edit}}(s'', t'') = 5$ $d_{\text{edit}}(s''', t''') = 10$

Exhaustive Search

- **Question:** How to find an optimal alignment?
- **Idea:** Try out all possible alignments
- **Problem:** These are too many

> Let s and t be two strings of length n.
> Then there are **more than 3^n** possible alignments for s and t.

- Alignment is uniquely defined by the positions of inserted gaps
- **Example for $n = 3$**

$$\begin{array}{cc|cc|c} s_1 & - & - & s_2 & s_3 \\ - & t_1 & t_2 & - & t_3 \end{array} \quad \text{or} \quad \begin{array}{c|cc|cc} s_1 & s_2 & - & - & s_3 \\ t_1 & - & t_2 & t_3 & - \end{array}$$

- This already leads to 3^n alignments

Exponential Time Complexity

n	10	50	100	300	10 000
$10n$	100	500	1 000	3 000	100 000
$3n^2$	300	7 500	30 000	270 000	300 000 000
n^3	1 000	125 000	1 000 000	27 000 000	13 digits
3^n	59 049	24 digits	48 digits	143 digits	4 772 digits

- Exhaustive search too slow
- **Dynamic programming**

Dynamic Programming

The Algorithm of Needleman and Wunsch

Dynamic Programming

> Solution for input can be computed from subsolutions to subproblems, starting with the smallest subproblem

- Subsolutions are stored and reused (repeatedly)
- Use table
- Memoization is closely related to dynamic programming
- Similar approach to divide-and-conquer, but tries to avoid recursion
- Bottom-Up instead of Top-Down
- **Bellman equation:** Optimal solution can be computed from optimal solutions to subproblems

Dynamic Programming

Whether DP can be applied depends on whether subproblems can be defined for which the Bellman equation works

The crucial point is thus to cleverly define subproblems

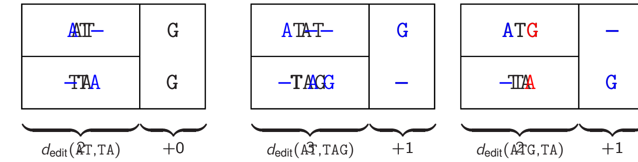
Needleman and Wunsch, 1970

- All pairs of prefixes of the given strings are subproblems
- Compute alignments of longer prefixes from optimal alignments of shorter prefixes

Example for the Alignment of Prefixes

Compute optimal alignment of $s = \texttt{ATG}$ and $t = \texttt{TAG}$

Distinguish **three cases** with respect to the last column



Computation of edit distance reduced to computation of edit distance of three pairs of prefixes

Optimal alignment with $d_{\text{edit}}(\texttt{ATG}, \texttt{TAG}) = 2$:

$$s' = \texttt{AT-G}$$
$$t' = \texttt{-TAG}$$

Initialization the Penalty Table

- The **empty string** is a string of length 0
- It is prefix of ever string

Initialization

Alignment of a non-empty prefix with λ is unique

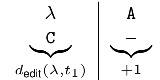
$$\begin{matrix} s_1\, s_2 \ldots s_i \\ -\,- \ldots - \end{matrix} \quad \text{or} \quad \begin{matrix} -\,- \ldots - \\ t_1\, t_2 \ldots t_i \end{matrix}$$

Penalty: $d_{\text{edit}}(s_1 \ldots s_i, \lambda) = d_{\text{edit}}(\lambda, t_1 \ldots t_i) = i$

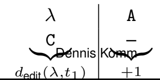
Filling out the Penalty Table

Initialization

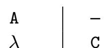
Insert gap in t



Insert gap in t



Insert gap in s



Filling out the Penalty Table

s \ t	0	1	...	j−1	j	...	n
0							
1							
2							
⋮							
i − 1							
i							
⋮							
m							

- ↓ Gap in t — Case 1
- → Gap in s — Case 2
- ↘ Match resp. mismatch — Case 3

$$d_{\text{edit}}(s_1 \dots s_i, t_1 \dots t_j) = \min\{d_{\text{edit}}(s_1 \dots s_{i-1}, t_1 \dots t_j) + 1,\; d_{\text{edit}}(s_1 \dots s_i, t_1 \dots t_{j-1}) + 1,\; d_{\text{edit}}(s_1 \dots s_{i-1}, t_1 \dots t_{j-1}) + p(s_i, t_j)\}$$

Dynamic Programming

Implementation in `numpy`

Alignment Algorithm – Initialization

- Use the module `numpy` that allows fast computations using matrices (tables)

```
import numpy as np
```

- Input is given as two strings `seq1` and `seq2`

```
seq1 = "ACTAC"
seq2 = "AACTGATGA"
m = len(seq1)
n = len(seq2)
```

- Initialize penalty table

```
penal = np.zeros((m+1, n+1))
for j in range(0, n+1):
    penal[0][j] = j                First row (alignment with λ)
for i in range(0, m+1):
    penal[i][0] = i                First column (alignment with λ)
```


Alignment Algorithm – Filling out the Penalty Table

```
for i in range(1, m+1):                                                For every row
  for j in range(1, n+1):                                              For every column
    if seq1[i-1] == seq2[j-1]:                                         Same letter in current cell?
      pij = 0                                                          Then no penalty
    else:                                                              Otherwise
      pij = 1                                                          There is a penalty
    x = [penal[i-1][j] + 1, penal[i][j-1] + 1, penal[i-1][j-1] + pij]
    penal_min = np.amin(x)  Consider three possibilities  Compute minimum of these
possibilities
    penal[i][j] = penal_min Insert gap in second string Insert gap in first string Insert
match respectively mismatch                                  Store minimum value in penalty table
```

Computing the Optimal Alignment

s \ t	0	C 1	C 2	T 3	G 4
0	0	1	2	3	4
A 1	1	1			
C 2	2				
T 3	3				
T 4	4				
G 5	5				

s \ t	0	C 1	C 2	T 3	G 4
0	0	1	2	3	4
A 1	1	1			
C 2	2	1			
T 3	3				
T 4	4				
G 5	5				

$s' =$ ACTTG

$t' =$ CCT–G

s \ t	0	C 1	C 2	T 3	G 4
0	0	1	2	3	4
A 1	1	1	2	3	4
C 2	2	1	1	2	3
T 3	3	2	2	1	2
T 4	4	3	3	2	2
G 5	5	4	4	3	2

Alignment Algorithm – Initialization of the Tracing Table

Tracing (Arrows in the penalty table)

Additionally store the index of the minimum element

Create table to store the way through the penalty table...

```
penal = np.zeros((m+1, n+1))
trace = np.zeros((m+1, n+1))                    Same size as penalty table
```

```
for j in range(1, n+1):
  trace[0][j] = 1                               Only steps from left in first row
for i in range(1, m+1):
  trace[i][0] = 0                               Only steps from above in first column
```


Alignment Algorithm – Filling out the Tracing Table

Store index of minimum with `numpy` function `argmin`...

```
penal_min = np.amin(x)                          Compute minimum penalty
index_min = np.argmin(x)                        Compute index of the minimum
penal[i][j] = penal_min                         Store value
trace[i][j] = index_min                         Store index
```

- Run backwards through table `trace`
- Value gives index of the minimum
- Insert gap or match respectively mismatch accordingly
- Continue with previous column and row of `trace`
- For match respectively mismatch, this is the cell above-left
- `i = i-1` and `j = j-1`
- Otherwise, only decrease `i` or `j`

Alignment Algorithm – Print the Result

- Result of algorithm is 2-dimensional list `result` with
  - one list for the first string
  - another list for the second string
- Start in the lower-right corner of the table
- Fill `result` with reversed alignment
- Result will be formatted more readable afterwards

```
result = [[], []]
i = m
j = n
```

Alignment Algorithm – Print the Result

```
while i > 0 or j > 0:                  As long as we are not yet at the upper-left corner
  if trace[i][j] == 0:                                              If step from above
    result[0].append(seq1[i-1])                              Insert gap in second string
    result[1].append("-")
    i -= 1                                                    And continue in row above
  elif trace[i][j] == 1:                                              If step from left
    result[0].append("-")                                     Insert gap in first string
    result[1].append(seq2[j-1])
    j -= 1                                              And continue in column to the left
  else:                                                         If step from above-left
    result[0].append(seq1[i-1])                    Insert match respectively mismatch
    result[1].append(seq2[j-1])                                     (Print both letters)
    i -= 1                                               And continue in cell above-left
    j -= 1

for k in range(len(result[0])-1, -1, -1):
   print(result[0][k], " <-> ", result[1][k])
```


Alignment Algorithm – Time Complexity

- There are two parts
  1. Filling out the tables
  2. Output of the result
- Filling out the matrices takes more time since every cell is considered

```
for i in range(1, m+1):                                                   For every row
  for j in range(1, n+1):                                              For every column
    if seq1[i-1] == seq2[j-1]                                          One comparison
    ...
    x = [penal[i-1][j] + 1, penal[i][j-1] + 1, penal[i-1][j-1] + pij]
    penalty_min = amin(x)                                             Two comparisons
    ...
```

$3 \cdot m \cdot n$ comparisons

Alignment Algorithm – Time Complexity

Time complexity: Roughly $3n^2$ for two strings of equal length n

n	10	50	100	300	10 000
$10n$	100	500	1 000	3 000	100 000
$3n^2$	300	7 500	30 000	270 000	300 000 000
n^3	1 000	125 000	1 000 000	27 000 000	13 digits
3^n	59 049	24 digits	48 digits	143 digits	4 772 digits

Comparison of two genes (size of $n \approx 10\,000$) takes

- **100 MB** of space and
- **less than 1 minute** of time

Alignment Algorithm – Time Complexity