

# Algorithmic patterns

Data Structures and Algorithms for Computational Linguistics III  
(ISCL-BA-07)

Çağrı Çöltekin

`ccoltekin@sfs.uni-tuebingen.de`

University of Tübingen  
Seminar für Sprachwissenschaft

Winter Semester 2020/21

# Overview

- Some common approaches to algorithm design
  - Revisiting recursion
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming

# Recursion - again

## linear search again

Your task from the last lecture: writing a recursive linear search.

# Recursion - again

## linear search again

Your task from the last lecture: writing a recursive linear search.

- Recursion is relatively easy:

```
if val == seq[0]:  
    return i  
else:  
    return rl_search(seq[1:], val, i+1)
```

# Recursion - again

## linear search again

Your task from the last lecture: writing a recursive linear search.

- Recursion is relatively easy:

```
if val == seq[0]:  
    return i  
else:  
    return rl_search(seq[1:], val, i+1)
```

- And we need a base case:

```
if not seq: # empty sequence  
    return None
```

# Recursion - again

## linear search again

Your task from the last lecture: writing a recursive linear search.

- Recursion is relatively easy:

```
if val == seq[0]:  
    return i  
else:  
    return rl_search(seq[1:], val, i+1)
```

- And we need a base case:

```
if not seq: # empty sequence  
    return None
```

# Recursion - again

## linear search again

Your task from the last lecture: writing a recursive linear search.

- Recursion is relatively easy:

```
if val == seq[0]:  
    return i  
else:  
    return rl_search(seq[1:], val, i+1)
```

- And we need a base case:

```
if not seq: # empty sequence  
    return None
```

### the complete code

```
1 def rl_search(seq, val, i=0):  
2     if not seq:  
3         return None  
4     if val == seq[0]:  
5         return i  
6     return rl_search(seq[1:], val, i+1)
```

# Recursion - again

## linear search again

Your task from the last lecture: writing a recursive linear search.

- Recursion is relatively easy:

```
if val == seq[0]:
    return i
else:
    return rl_search(seq[1:], val, i+1)
```

- And we need a base case:

```
if not seq: # empty sequence
    return None
```

### the complete code

```
1 def rl_search(seq, val, i=0):
2     if not seq:
3         return None
4     if val == seq[0]:
5         return i
6     return rl_search(seq[1:], val, i+1)
```

Can we improve this?



# How does this recursion work

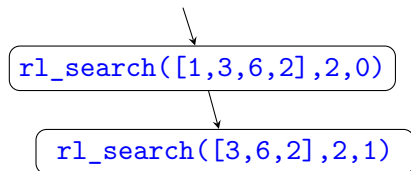
recursion trace/graph



```
rl_search([1,3,6,2],2,0)
```

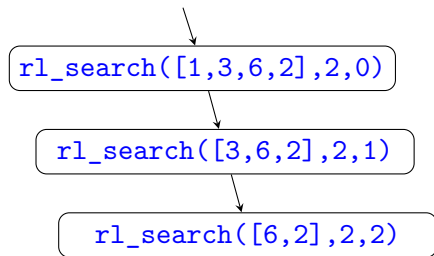
# How does this recursion work

recursion trace/graph



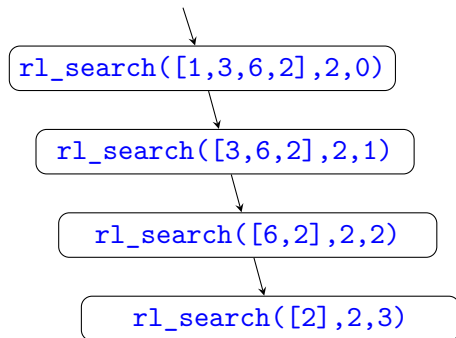
# How does this recursion work

recursion trace/graph



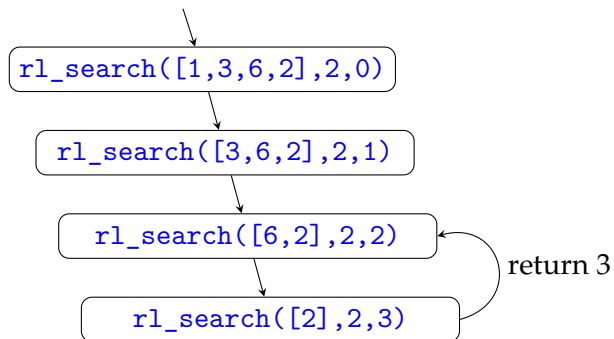
# How does this recursion work

recursion trace/graph



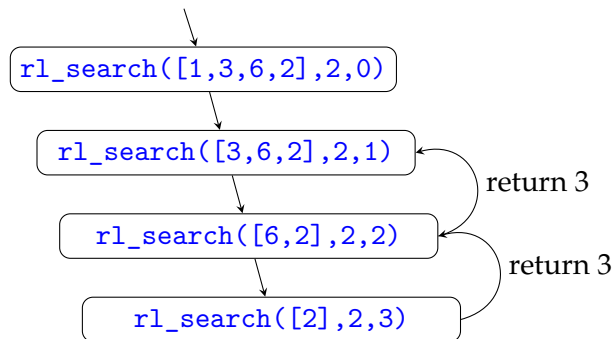
# How does this recursion work

recursion trace/graph



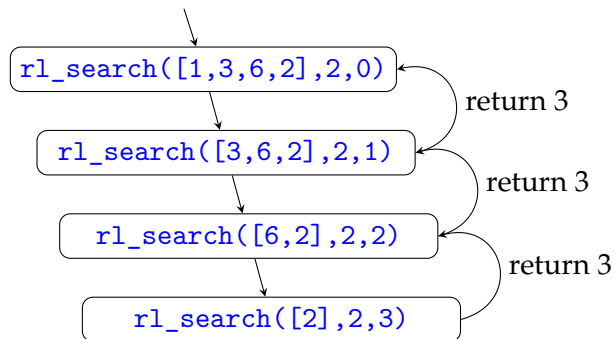
# How does this recursion work

recursion trace/graph



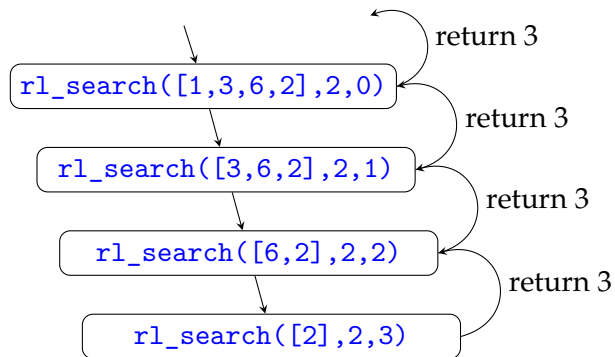
# How does this recursion work

recursion trace/graph



# How does this recursion work

recursion trace/graph





# Recursion: practical issues

## recursion depth and tail recursion

- Each function call requires some bookkeeping
- Compilers/interpreters allocate space on a stack for the bookkeeping for each function call
- Most environments limit the number of recursive calls: long chains of recursion is likely to be errors
- *Tail recursion* (e.g., our recursive search example) is easy to convert to iteration
- It is also easy to optimize, and optimized by many compilers (not by the Python interpreter)

# Another recursive example

an algorithm course is required to introduce Fibonacci numbers

Fibonacci numbers are defined as:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1$$

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

- Recursion is common in math, and maps well to the recursive algorithms

## Another recursive example

an algorithm course is required to introduce Fibonacci numbers

Fibonacci numbers are defined as:

$$F_0 = 0$$

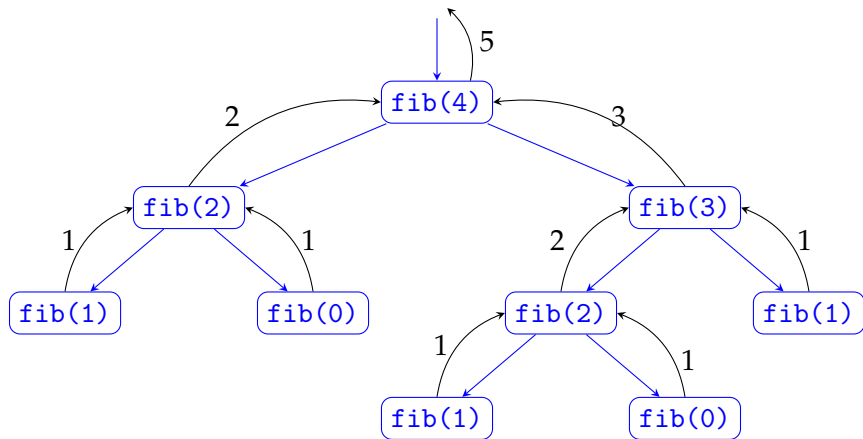
$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1$$

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

- Recursion is common in math, and maps well to the recursive algorithms
- Note that we now have binary recursion, each function call creates two calls to self
- We follow the math exactly, but is this code efficient?

# Visualizing binary recursion



# Brute force

- In some cases, we may need to enumerate all possible cases (e.g., to find the best solution)
- Common in combinatorial problems
- Often intractable, practical only for small input sizes
- It is also typically the beginning of finding a more efficient approach

# Brute force

example: finding all possible ways to segment a string

- Segmentation is prevalent in CL
  - Examples include finding words: tokenization (particularly for writing systems that do not use white space)
  - Finding sub-word units (e.g., morphemes, or more specialized application: compound splitting)
  - Psycholinguistics: how do people extract words from continuous speech?
- We consider the following problem:
  - Given a metric or score to determine the "best" segmentation
  - We enumerate all possible ways to segment, pick the one with the best score

# Brute force

example: finding all possible ways to segment a string

- Segmentation is prevalent in CL
  - Examples include finding words: tokenization (particularly for writing systems that do not use white space)
  - Finding sub-word units (e.g., morphemes, or more specialized application: compound splitting)
  - Psycholinguistics: how do people extract words from continuous speech?
- We consider the following problem:
  - Given a metric or score to determine the "best" segmentation
  - We enumerate all possible ways to segment, pick the one with the best score
- How can we enumerate all possible segmentations of a string?

# Segmentation

a recursive solution

```
1 def segment_r(seq):  
2     if len(seq) == 1:  
3         yield [seq]  
4     else:  
5         for seg in segment_r(seq[1:]):  
6             yield [seq[0]] + seg  
7             yield [seq[0] + seg[0]] + seg[1:]
```



# Segmentation

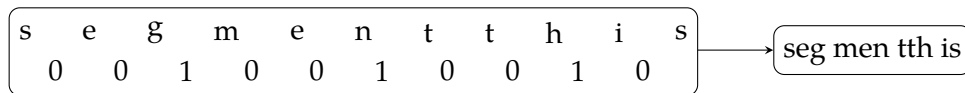
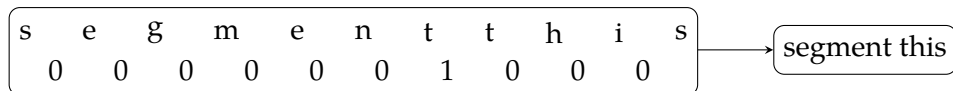
a recursive solution

```
1 def segment_r(seq):  
2     if len(seq) == 1:  
3         yield [seq]  
4     else:  
5         for seg in segment_r(seq[1:]):  
6             yield [seq[0]] + seg  
7             yield [seq[0] + seg[0]] + seg[1:]
```

- Can you think of a non-recursive solution?

# Enumerating segmentations

sketch of a non-recursive solution



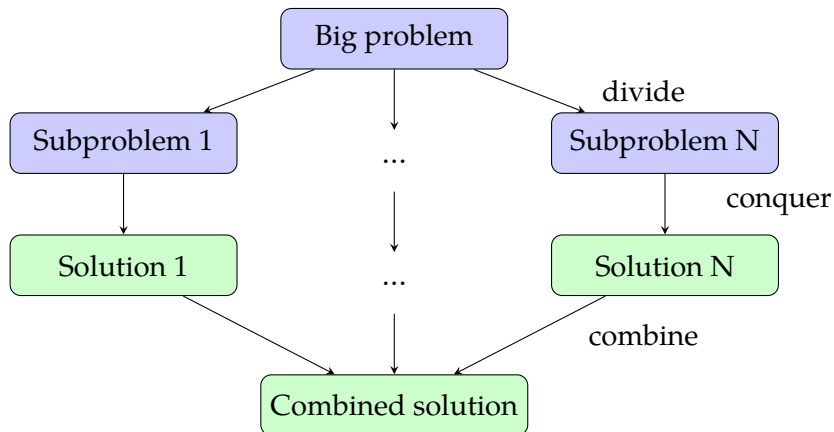
- '1' means there is a boundary at this position
- Problem is now enumerating all possible binary strings of length  $n - 1$  (this is binary counting)

# Divide and conquer

- The general idea is dividing the problem into smaller parts until it becomes trivial to solve
- Once small parts are solved, the results are combined
- Goes very well with recursion
- We have already seen a particular flavor: binary search
- The algorithms like binary search are sometimes called *decrease and conquer*

# Divide and conquer

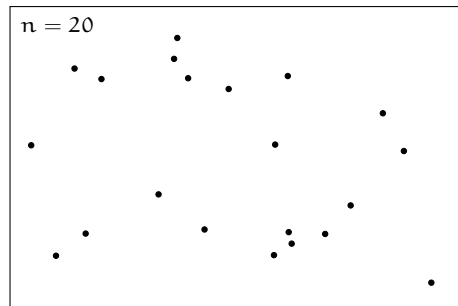
## General idea



# Divide and conquer

an example: nearest neighbors (only a sketch)

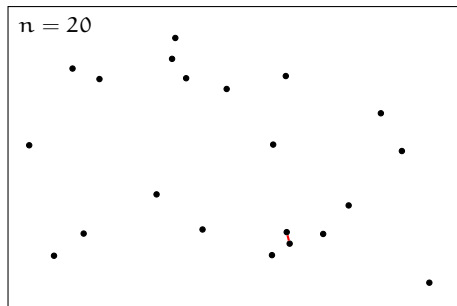
- Task: find the closest two points



# Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution:  
 $20 \times 20 = 400$  comparisons<sup>1</sup>

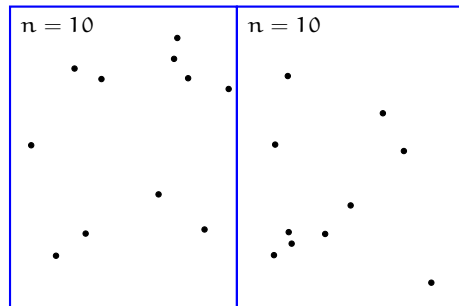


<sup>1</sup>Precisely,  $(20 \times 19) / 2 = 190$ . In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gain by division is the same.

# Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution:  
 $20 \times 20 = 400$  comparisons<sup>1</sup>
- Divide



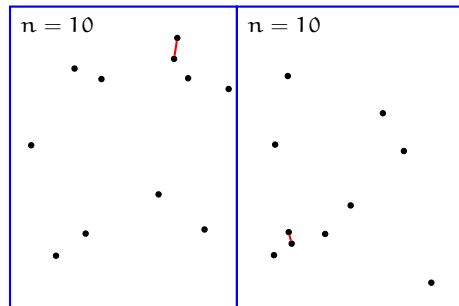
assume we can divide into half easily

<sup>1</sup>Precisely,  $(20 \times 19) / 2 = 190$ . In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gain by division is the same.

# Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution:  
 $20 \times 20 = 400$  comparisons<sup>1</sup>
- Divide
- Solve separately (conquer):  
 $10 \times 10 + 10 \times 10 = 200$  comparisons



assume we can divide into half easily  
 overlooking the comparisons across the division

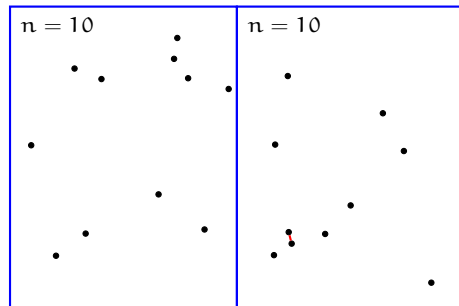
<sup>1</sup>Precisely,  $(20 \times 19) / 2 = 190$ . In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gain by division is the same.



# Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution:  
 $20 \times 20 = 400$  comparisons<sup>1</sup>
- Divide
- Solve separately (conquer):  
 $10 \times 10 + 10 \times 10 = 200$  comparisons
- Combine: pick the minimum of the individual solutions



assume we can divide into half easily  
overlooking the comparisons across the division

- Gain is higher when  $n$  is larger, and we divide further

<sup>1</sup>Precisely,  $(20 \times 19) / 2 = 190$ . In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gain by division is the same.

# Divide and conquer

## summary

- This is probably the most common example
- Divide and conquer does not always yield good results, the cost of merging should be less than the gain from division
- Many of the important algorithms fall into this category:
  - merge sort and quick sort (coming soon)
  - integer multiplication
  - matrix multiplication
  - fast Furrier transform (FFT)

# Greedy algorithms

- An algorithm is greedy if it optimizes a local constraint
- For some problems, greedy algorithms result in correct solutions
- In others they may result in 'good enough' solutions
- If they work, they are efficient
- An important class of graph algorithms fall into this category (e.g., finding shortest paths, scheduling)

# Greedy algorithms

a simple example: 'change making'

- We want to produce minimum number of coins for a particular sum  $s$ 
  1. Pick the largest coin  $c \leq s$
  2. set  $s = s - c$
  3. repeat 1 & 2 until  $s = 0$

# Greedy algorithms

a simple example: 'change making'

- We want to produce minimum number of coins for a particular sum  $s$ 
  1. Pick the largest coin  $c \leq s$
  2. set  $s = s - c$
  3. repeat 1 & 2 until  $s = 0$
- Is this algorithm correct?

# Greedy algorithms

a simple example: 'change making'

- We want to produce minimum number of coins for a particular sum  $s$ 
  1. Pick the largest coin  $c \leq s$
  2. set  $s = s - c$
  3. repeat 1 & 2 until  $s = 0$
- Is this algorithm correct?
- Think about coins of 10, 30, 40 and apply the algorithm for the sum value of 60

# Greedy algorithms

a simple example: 'change making'

- We want to produce minimum number of coins for a particular sum  $s$ 
  1. Pick the largest coin  $c \leq s$
  2. set  $s = s - c$
  3. repeat 1 & 2 until  $s = 0$
- Is this algorithm correct?
- Think about coins of 10, 30, 40 and apply the algorithm for the sum value of 60
- Is it correct if the coin values were limited Euro coins?

# Dynamic programming

- Dynamic programming is a method to save earlier results to reduce computation
- It is sometimes called memoization (it is not a typo)
- Again, a large number of algorithms we use fall into this category, including common parsing algorithms



# Dynamic programming

example: Fibonacci

```
1 def memofib(n, memo = {0: 0, 1:1}):  
2     if n not in memo:  
3         memo[n] = memofib(n-1) + memofib(n-2)  
4     return memo[n]
```

- We save the results calculated in a dictionary,
- if the result is already in the dictionary, we return without recursion
- Otherwise we calculate recursively as before
- The difference is big, but there is also a 'neater' solution without (explicit) memoization

# Summary

- We saw a few general approaches to (efficient) algorithm design

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound
  - Randomized algorithms

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound
  - Randomized algorithms
  - Distributed algorithms (sometime called swarm optimization)

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound
  - Randomized algorithms
  - Distributed algorithms (sometime called swarm optimization)
  - Transformation



# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound
  - Randomized algorithms
  - Distributed algorithms (sometime called swarm optimization)
  - Transformation
- Designing algorithms is difficult but analyzing them is even more difficult (next topic)

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound
  - Randomized algorithms
  - Distributed algorithms (sometime called swarm optimization)
  - Transformation
- Designing algorithms is difficult but analyzing them is even more difficult (next topic)

# Summary

- We saw a few general approaches to (efficient) algorithm design
- Designing algorithms is not a mechanical procedure: it requires creativity
- There are other common patterns, including
  - Backtracking, Branch-and-bound
  - Randomized algorithms
  - Distributed algorithms (sometime called swarm optimization)
  - Transformation
- Designing algorithms is difficult but analyzing them is even more difficult (next topic)

Next:

- Analysis of algorithms
- Reading: textbook (Goodrich, Tamassia, and Goldwasser 2013) chapter 3

# Linear search

a little bit of optimization

```
1 def rl_search(seq, val, i=0):  
2     if not seq:  
3         return None  
4     if val == seq[0]:  
5         return i  
6     else:  
7         return rl_search(seq[1:], val,  
                           ↪ i+1)
```

```
1 def rl_search2(seq, val, i=0):  
2     if i >= len(seq):  
3         return None  
4     if val == seq[i]:  
5         return i  
6     else:  
7         return rl_search2(seq, val, i  
                           ↪ + 1)
```

Which one is faster, and why?

## Better solutions for Fibonacci numbers

```
1 def fib2(n):  
2     if n <= 1:  
3         return (n, 0)  
4     a, b = fib2(n - 1)  
5     return (a+b, a)
```

```
1 def fib3(n):  
2     if n <= 1:  
3         return n  
4     a, b = 0, 1  
5     for i in range(0, n):  
6         a, b = b, a + b  
7     return a
```

Which one is faster/better?

# Segmentation

without yield

```
1 def segment_r(seq):  
2     segs = []  
3     if len(seq) == 1:  
4         return [seq]  
5     for seg in segment_r(seq[1:]):  
6         segs.append([seq[0]] + seg)  
7         segs.append([seq[0] + seg[0]] + seg[1:])  
8     return segs
```

# Acknowledgments, credits, references

- Some of the slides are based on the previous year's course by Corina Dima.



Goodrich, Michael T., Roberto Tamassia, and Michael H. Goldwasser (2013). *Data Structures and Algorithms in Python*. John Wiley & Sons, Incorporated. ISBN: 9781118476734.







