

Algorithmic patterns

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

Çağrı Çöltekin
ccolt@infsa.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2023/24

© Çöltekin, 2023

Overview

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

© Çöltekin, 2023

Winter Semester 2023/24

Recursion

linear search again

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

- Recursion is relatively easy:


```
if val == seq[0]:
    return i
else:
    return r1_search(seq[1:], val, i+1)
```
- And we need a base case:


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

```
the complete code
def r1_search(seq, val, i=0):
    if not seq:
        return None
    if seq[i] == val:
        return i
    else:
        return r1_search(seq[1:], val, i+1)
```

Can we improve this?

© Çöltekin, 2023

Winter Semester 2023/24

How does this recursion work

recursion trace/graph



© Çöltekin, 2023

Winter Semester 2023/24

© Çöltekin, 2023

Winter Semester 2023/24

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 are likely to cause 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)

© Çöltekin, 2023

Winter Semester 2023/24

Another recursive example

every algorithm course is required to introduce Fibonacci numbers

Fibonacci numbers are defined as:

$$F_0 = 0$$

$$F_1 = 1$$

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

```
def fib(n):
    if n <= 1:
        return n
    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?

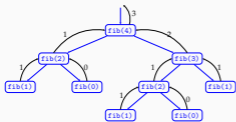
© Çöltekin, 2023

Winter Semester 2023/24

© Çöltekin, 2023

Winter Semester 2023/24

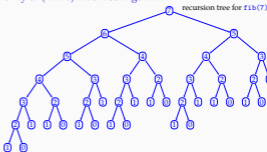
Visualizing binary recursion



© Çöltekin, 2023

Winter Semester 2023/24

Complexity of (naive) Fibonacci algorithm



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

© Çöltekin, 2023

Winter Semester 2023/24

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

```
def segment_r(seq):
    segs = []
    if len(seq) == 1:
        return [[seq]]
    for seg in segment_r(seq[1:]):
        segs.append([seq[0]] + seg)
        segs.append([seq[0]] + seg[0:] + seg[1:])
    return segs
```

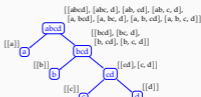
• Can you think of a non-recursive solution?

© Çöltekin, 2023

Winter Semester 2023/24

Segmentation

example/analysis



© Çöltekin, 2023

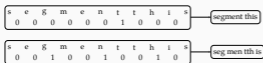
Winter Semester 2023/24

© Çöltekin, 2023

Winter Semester 2023/24

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 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

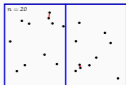
General idea



Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution: $20 \times 20 = 400$ comparisons¹
- Divide
- Solve separately (conquer): $10 \times 10 + 10 \times 10 = 200$ comparisons
- Combine: pick the minimum of the individual solutions



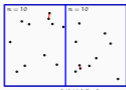
- Gain is higher when n is larger, and we divide further

¹Presumably, $(20 \times 19) / 2 = 190$. In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gains by divisions is the same.

Divide and conquer

an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution: $20 \times 20 = 400$ comparisons¹
- Divide
- Solve separately (conquer): $10 \times 10 + 10 \times 10 = 200$ comparisons
- Combine: pick the minimum of the individual solutions



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

¹Presumably, $(20 \times 19) / 2 = 190$. In this class we focus on 'order' of operations, rather than the exact numbers. And, the order of gains by divisions is the same.

Divide and conquer

summary

- This is probably the most common pattern
- Divide and conquer does not always yield good results, the cost of merging should be less than the gain from the division(s)
- Many of the important algorithms fall into this category:
 - merge sort and quick sort (coming soon)
 - integer multiplication
 - matrix multiplication
 - fast Fourier 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 $s = 0$
- 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

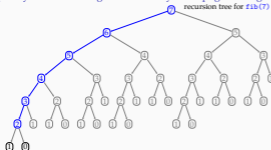
example: Fibonacci

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

Complexity of Fibonacci algorithm with dynamic programming

recursion tree for fib(7)



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 (sometimes called swarm optimization)
 - Transformation
- Designing algorithms is difficult (possibly, not as difficult as analyzing them)

Next:

- Sorting
- Reading: [goodrich2013](#)

Nearest neighbors

an exercise

- Define and implement a divide-and-conquer algorithm for nearest neighbor problem, which divides the input into two until the solution becomes trivial
- Analyze your algorithm and compare to the naive version sketched above (an implementation was provided in the previous lecture)

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,
8                          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
8                            + 1)
```

Which one is faster, and why?

Better solutions for Fibonacci numbers

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

```
1 def fibo(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

with yield

```
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:]
```

Acknowledgments, credits, references