Algorithmic patterns
Data Structures and Algorithms for Computational Linguistics III (ISCL-BA-17)

## Çağ̛n Çoltekin

ccoltekinasfs.uni-tuebingen.de
Univeraty of Tubingse
Seminar fur Spractivisensectaft
Winter Semester 2002/23

## Overview

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


## Recursion

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

|  | the complete code |
| :---: | :---: |
| ```If val - saq(01: zeturm 1 else: return r1_search(segg [1:1. nal, i+1)``` | ```1 def r1_search(seq. val, i=0): if not seq: return None if val -- seq[0]:``` |
| And we need a base case: | return 1 |
| 14 not aege | ( return rl_search(seq[1:], val, 1+1) |

Can we improve this?

Mindersemaramis 21

Recursion: practical issues
recursion depth and tall 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)

How does this recursion work
recursion trace/graph



Another recursive example
every algoerthm course is required to introduce Fibonace numbers
Fibonacci numbers are defined as:
$F_{0}=0$
$F_{1}=1$
$F_{n}=F_{n-1}+F_{n-2}$ for $n>1$

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

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

- 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
- How can we enumerate all possible segmentations of a string?


## Segmentation

a recursive solution
Segmentation
example/analysts


- Can you think 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 sdea


Divide and conquer
an example: nearest neighbors (only a sketch)

- Task: find the closest two points
- Direct solution:
$20 \times 20=400$ comparisons $^{1}$
- 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


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



## 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 $c<=s$
2. set $s=s-c$
3. repeat $1 \& 2$ until $\mathrm{s}=0$

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 eariler 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: Filonacci

```
def memofib(n, memo - {0:0, 1:1})
    if n not in nemo:
    meno[n] - memofib(n-1) + nemofib(n-2)
    return meno [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 - $\quad-\quad$.


## 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
- Backtracking, Branch-and
- Randomized algorithms
- Randomized algorithms - Distributed algorithms (sometime called swarm optimization)
- Drstributed algor
- Designing algorithms is difficult (possibly, not as difficult as analyzing them) Next:
- Sorting
- Reading: goodrich2013

Complexity of Fibonacci algorithm with dynamic pogramming


## Nearest neighbors

an exercase

- 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)

```
    Mof fl_marc&(deq, val, 100):
    it pot aqeq
```



```
    14 val-mag[0]
    My.
        Nenturn r1-
```

    Which one is faster, and why?
    

```
dat r1-aurck2(avq, va1, 1~0):
    14, i> 1es(asq)
    if velumn haq[ [i]
    if val-2nq(i)
```


Segmentation
with yield
1. def segnent_r(seq):
if len(seq) - 1
yield [seq]
else:
for seg in segnent_r (seq[1:]):
yield [seq[0]] + seg
yield $[s e q[0]+\operatorname{seg}[0]]+\operatorname{seg}[1:]$


## Acknowledgments, credits, references

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

Better solutions for Fibonacci numbers


Which one is faster/better?

|  |
| :---: |
|  |
|  |
|  |
|  |


|  |
| :--- |
|  |
|  |
|  |
|  |



|  |  |
| :---: | :---: |
|  |  |



