Lecture 10: Complexity, Dynamic Programming

PCL II, CL, UZH
May 4, 2016

Universität Zürich[UZH]

Outline

- Part 1: Complexity
  - Time Complexity
  - Space Complexity
  - Big O Notation

- Part 2: Dynamic Programming

Part 1: Complexity

For a program/an algorithm that handles input of varying length

- e.g. string, text, file, sequence of numbers

- **How long** will it take?
    - *Time complexity*
- **How much memory** will it use?
    - *Space complexity*

(Relative to the size of the input)

Profiling

Universität Zürich[UZH]

- understand why the code is running slow
- compare different approaches/methods
- optimize solutions
- Tools:
  - `timeit`
  - `cProfile`

→ **Monitoring and "debugging" complexity**

Time complexity

- **Hapax legomena**: words (word forms) that occur in a text only once
- Useful why?
  - hapax legomena usually form around half of the vocabulary
    - 44.2% of the Brown corpus vocabulary
  - hapax legomena usually constitute a small portion of the running tokens
    - 1.9% of the Brown corpus
  - a good model for unknown words

Counting hapax legomena

```
for token in tokenList:
    tokenCount = 0

    for token2 in tokenList:
        if token2 == token:
            tokenCount += 1

    if tokenCount == 1:
        hapaxCount += 1
```

- python hapax.py data.txt 300: 0.186s
- python hapax.py data.txt 400: 0.316s
- python hapax.py data.txt 500: 0.545s
- python hapax.py data.txt 600: 0.816s

Counting hapax legomena

```
frequencies = defaultdict(int)

for token in tokenList:
    frequencies[token] += 1

for token in frequencies:
    if frequencies[token] == 1:
        hapaxCount += 1
```

- python fhapax.py data.txt 1000: 0.033s
- python fhapax.py data.txt 2000: 0.039s
- python fhapax.py data.txt 4000: 0.049s
- python fhapax.py data.txt 8000: 0.077s

Counting hapax legomena

```
hapaxCount = 0

for token in tokenList:
    tokenCount = 0

    for token2 in tokenList:
        if token2 == token:
            tokenCount += 1

    if tokenCount == 1:
        hapaxCount += 1
```

```
frequencies = defaultdict(int)

for token in tokenList:
    frequencies[token] += 1

for token in frequencies:
    if frequencies[token] == 1:
        hapaxCount += 1
```

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = 0
    for word2 in text:
        if word2 == word:
            c += 1
    if c == 1:
        print(word)      # is, only
```

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = 0
    for word2 in text:
        if word2 == word:
            c += 1
    if c == 1:
        print(word)        # is, only
```

How many times are the various blocks of the program executed?

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = 0                     # 6 times (text length)
    for word2 in text:
        if word2 == word:     # 6*6 times
            c += 1
    if c == 1:                # 6 times
        print(word)           # is, only
```

How many times are the various blocks of the program executed?

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = 0                    # 6 times (text length)
    for word2 in text:
        if word2 == word:        # 6*6 times
            c += 1
    if c == 1:               # 6 times
        print(word)          # is, only
```

- outer loop: n times (n = size of input)
- inner loop: n^2 times

Time complexity

Universität Zürich[UZH]

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = 0                      # 6 times (text length)
    for word2 in text:
        if word2 == word:      # 6*6 times
            c += 1
    if c == 1:                 # 6 times
        print(word)            # is, only
```

- Basically: count iterations per loop
- Nested loops: multiply

Time complexity

Universität Zürich[UZH]

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = mycount(text, word)
    if c == 1:
        print(word)
```

```
def mycount(txt, w):
    c = 0
    for w2 in txt:
        if w2 == w:
            c += 1
    return c
```

- Basically: count iterations per loop
- Nested loops: multiply
- Loops can "hide" in functions

Time complexity

Universität Zürich[UZH]

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
for word in text:
    c = text.count(word)
    if c == 1:
        print(word)
```

- Basically: count iterations per loop
- Nested loops: multiply
- Loops can "hide" in functions
  - also applies to built-in and standard functions
  - http://wiki.python.org/moin/TimeComplexity

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
frequencies = defaultdict( int)
for word in text:
    frequencies[word] += 1
for word in frequencies:
    if frequencies[word] == 1
        print(word)
```

- Faster Solution: how many iterations?

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
frequencies = defaultdict( int)
for word in text:
     frequencies[word] += 1
for word in frequencies:
     if frequencies[word] == 1
          print(word)
```

- Faster solution: $n + v$ iterations
- n: text length, v: vocabulary size

Time complexity

- Finding hapax legomena:

```
text = ["a", "duck", "is", "only", "a", "duck"]
frequencies = defaultdict( int)
for word in text:
     frequencies[word] += 1          # 6 times (text length n )
for word in frequencies:
     if frequencies[word] == 1      # 4 times (vocabulary length v )
          print(word)
```

- Faster solution: $n + v$ iterations
- n: text length, v: vocabulary size

Time complexity

- Why does this matter?
- Let's assume that
    - 1 operation takes 1 microsecond
    - = 1 mln operations take 1 second
- Finding hapax legomena in the Brown corpus:
    - n = 1161192 (words)
    - v = 44815 (types)
    - **slow solution**: $n^2 + 2n$ operations
        - **374.5 hours**
    - **faster solution**: $n + v$ operations
        - **1.2 seconds**

Algorithmic complexity

Universität Zürich[UZH]

- For a program/an algorithm that handles input of varying length
  - e.g. string, text, file, sequence of numbers

- **How long** will it take?
  - *Time complexity*
- **How much memory** will it use?
  - *Space complexity*

(Relative to the size of the input)

Space complexity

```
fileHandle = open(fileName, 'r')

lines = fileHandle.readlines()
for line in lines:
    ...
```

Space complexity

```
fileHandle = open(fileName, 'r')

lines = fileHandle.readlines()  #['line1', 'line2',...]
for line in lines:
    ...
```

- used memory = size of the whole file

Space complexity

Universität Zürich[UZH]

```
fileHandle = open(fileName, 'r')

lines = fileHandle.readlines()  #['line1', 'line2',...]
for line in lines:
    ...
```

- used memory = size of the **whole file**

```
fileHandle = open(fileName, 'r')

for line in fileHandle:
    ...
```

- used memory = size of the longest line

Algorithmic complexity

Universität Zürich[UZH]

- For a program/an algorithm that handles input of varying length
  - e.g. string, text, file, sequence of numbers
- **How long** will it take?
  - *Time complexity*
- **How much memory** will it use?
  - *Space complexity*

(Relative to the size of the input)

Big O notation

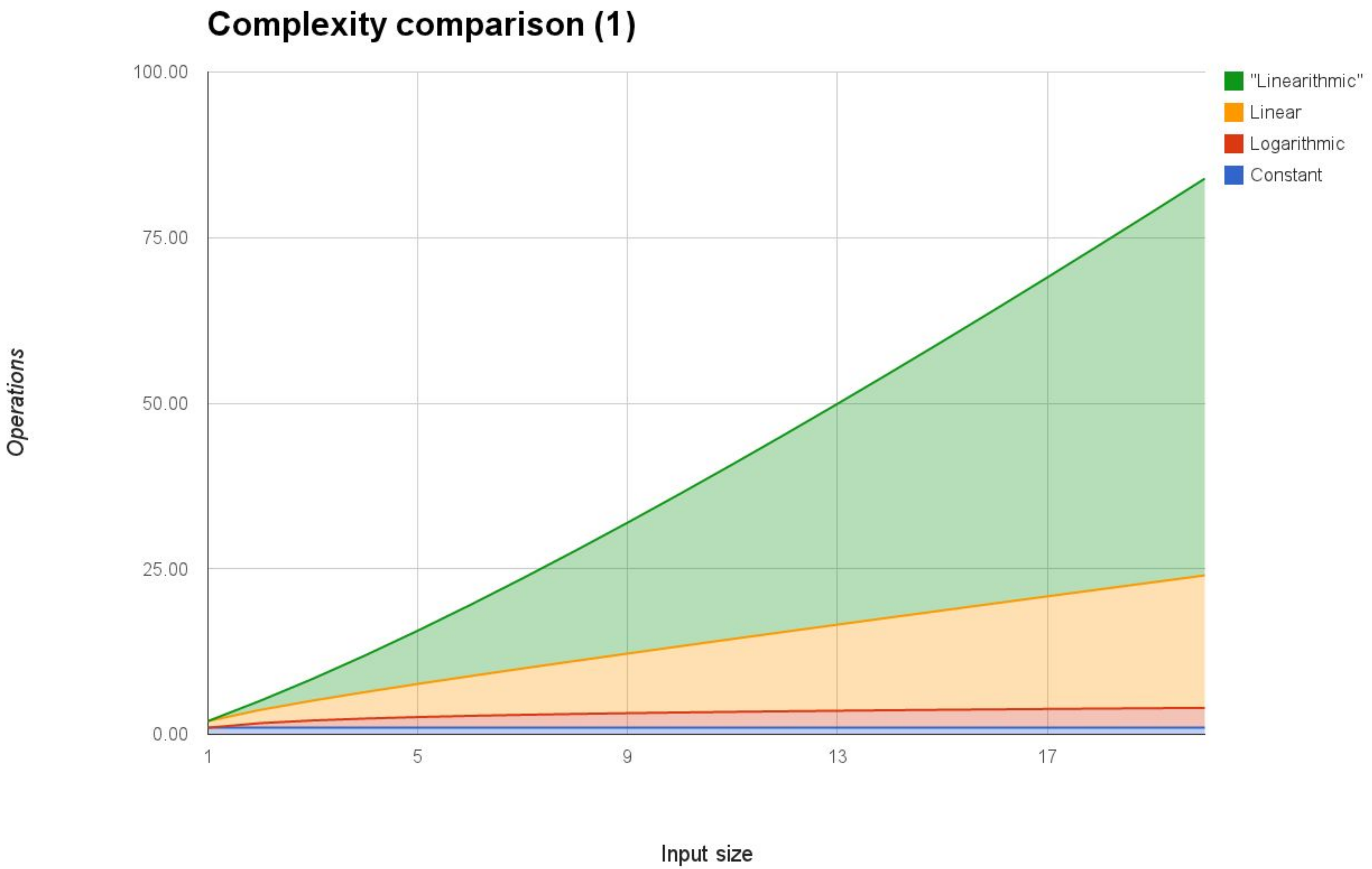
- O: upper bound of a function
  - "Worst-case scenario"
- hapax legomena:
  - naive: $O(n^2)$
  - good: $O(n + v)$
- **does not** describe the exact number of operations, instead -- shows how quickly a function grows
  - constant factors shortened,
    $n + n + n = 3n \in O(n)$
  - for a combo the fastest growing part counts:
    $2n + 3n^2 + n^3 \in O(n^3)$

Big O notation

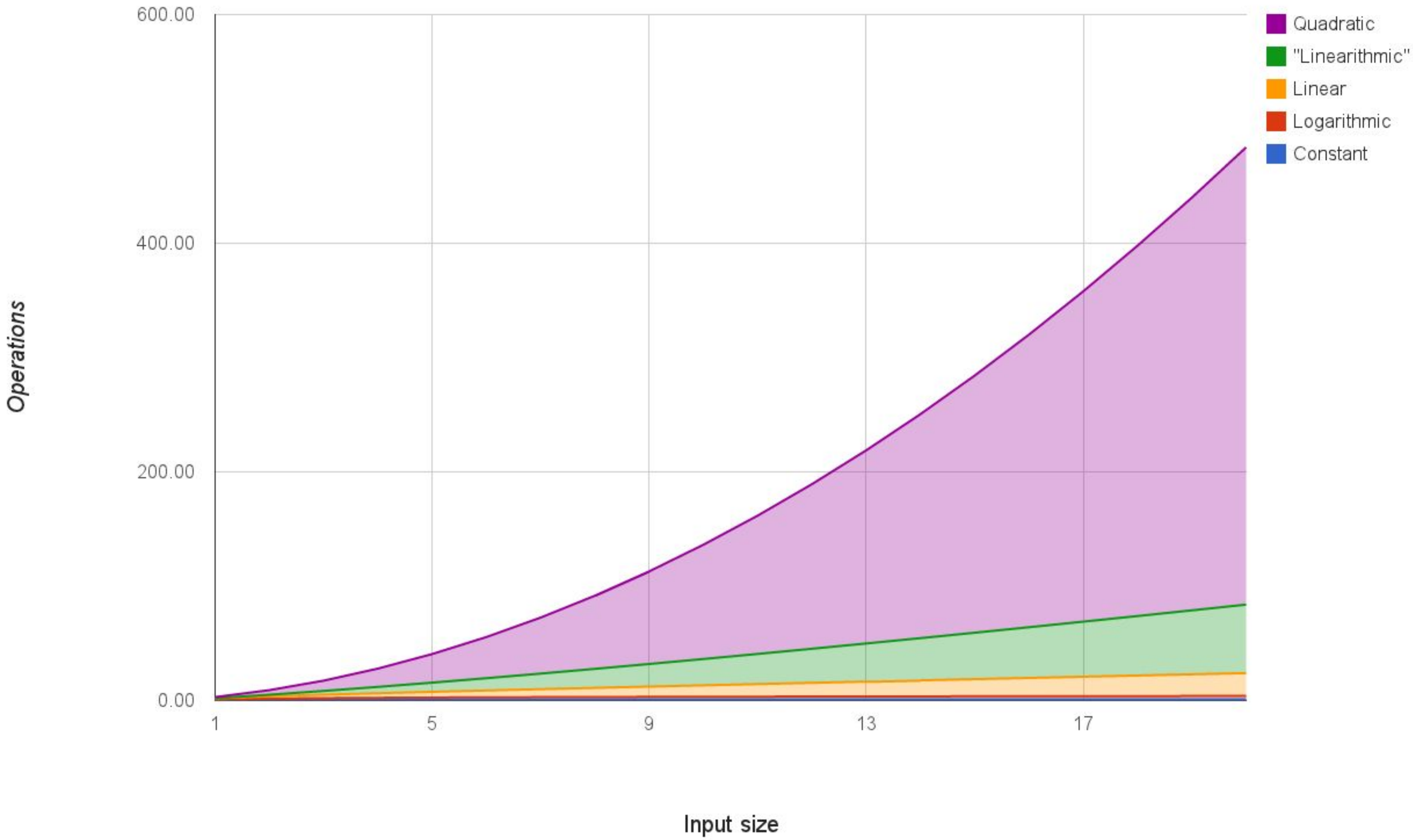
- $O(1)$: constant
  - e.g. access to a list element at a known index
- $O(log(n))$: logarithmic
  - e.g. search in a sorted list
- $O(n)$: linear
  - e.g. search in a list, adding/deleting elements
- $O(n \cdot log(n))$: n log n/"linearithmic"/loglinear
  - e.g. sorting a list

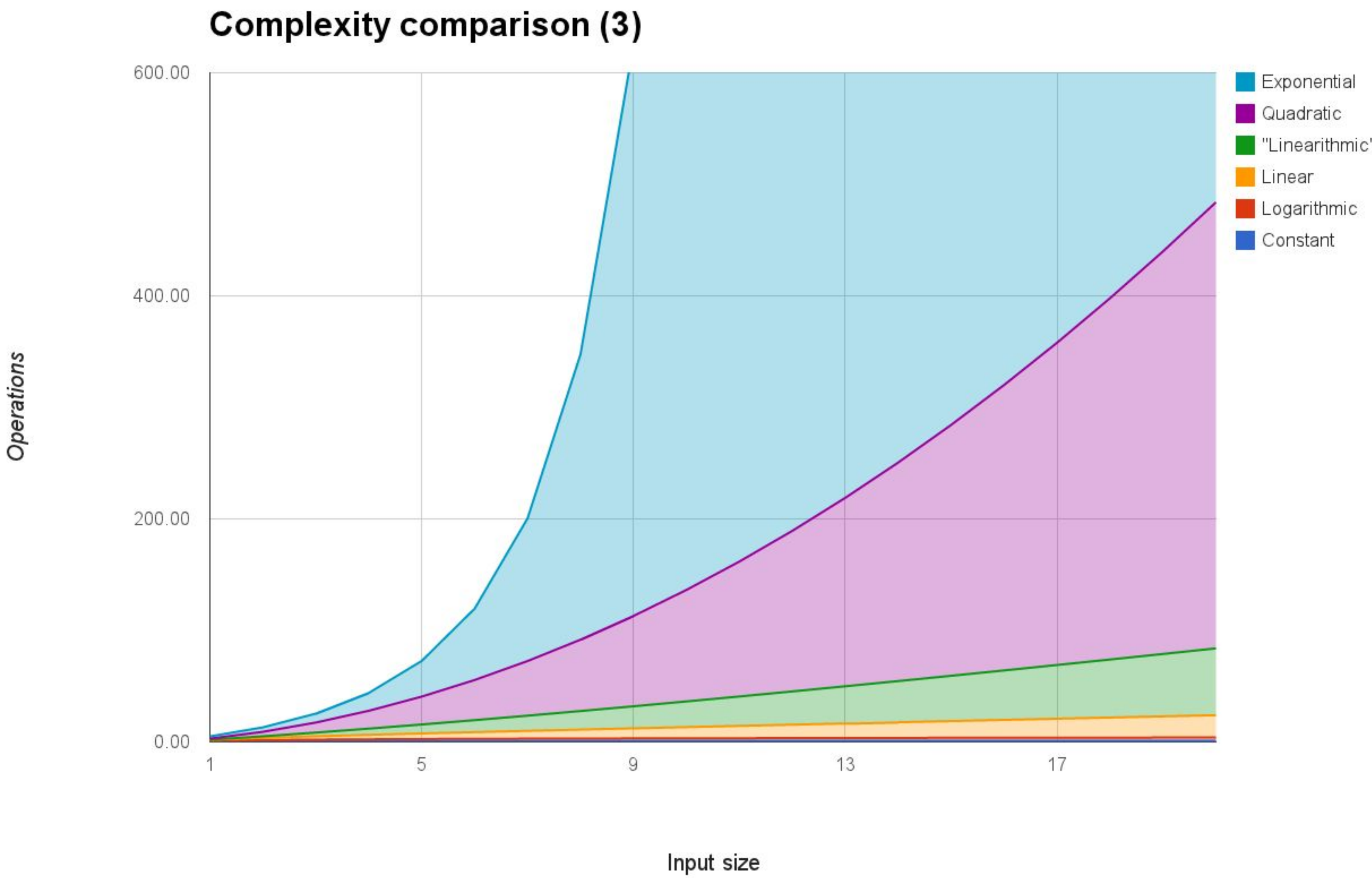
Big O notation

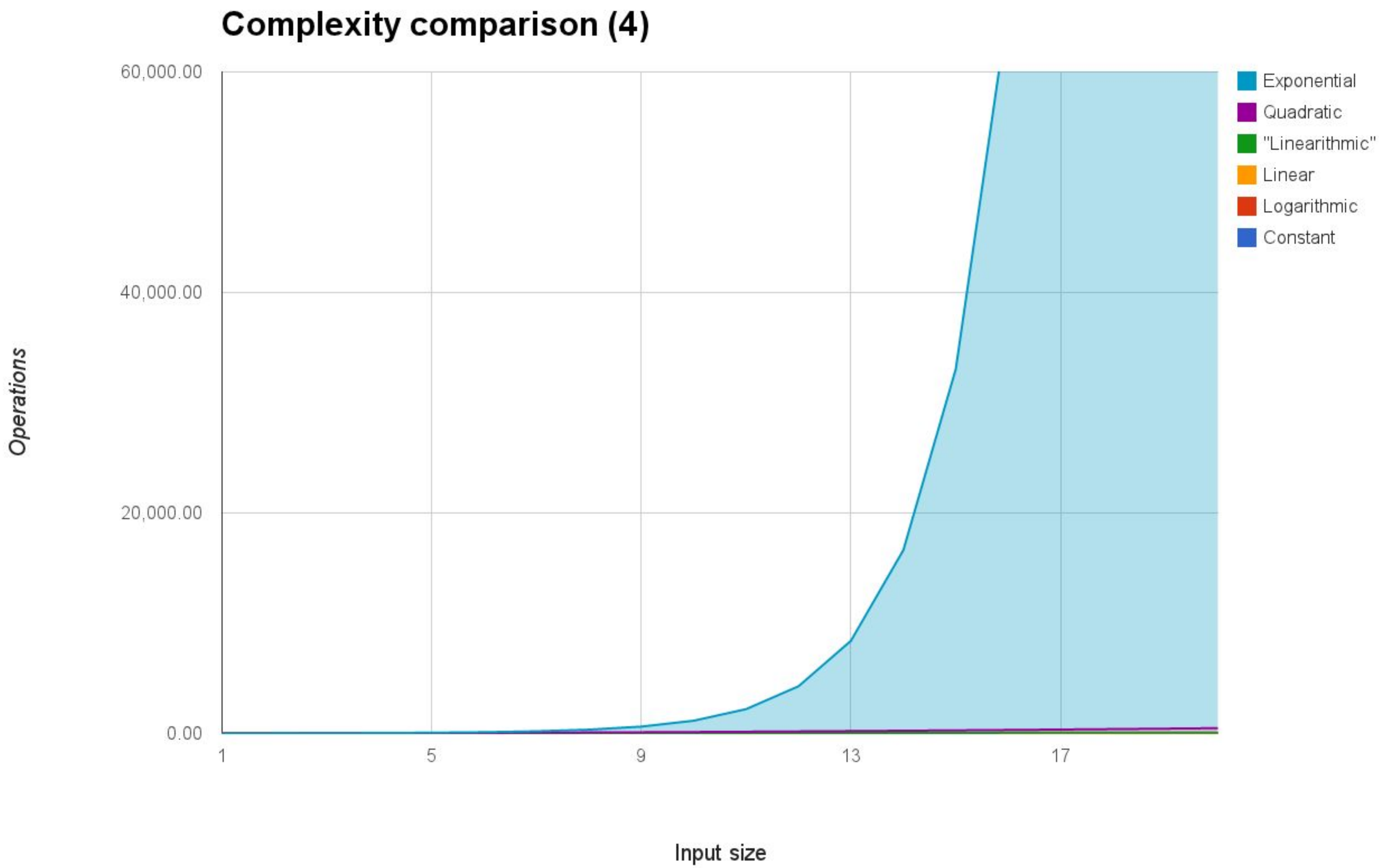
- $O(n^2)$: quadratic
  - e.g. naive bubble sort
  - any 2 nested loops over the same data
- $O(n^c), c > 1$: polynomial
- $O(c^n)$: exponential
- http://wiki.python.org/moin/TimeComplexity


Complexity comparison (1)
100.00
75.00
50.00
25.00
0.00
1
5
9
13
17
"Linearithmic"
Linear
Logarithmic
Constant
Operations
Input size

Complexity comparison (2)




Complexity comparison (3)
600.00
400.00
200.00
0.00
Operations
1
5
9
13
17
Input size
Exponential
Quadratic
"Linearithmic"
Linear
Logarithmic
Constant



Complexity comparison (4)
60,000.00
40,000.00
20,000.00
0.00
Operations
1
5
9
13
17
Input size
Exponential
Quadratic
"Linearithmic"
Linear
Logarithmic
Constant

Processing time difference

Assuming 1 mln operations per second:

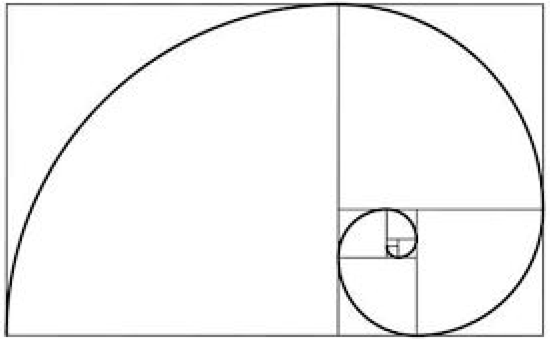
n=	10	20	30
$O(1)$	1µs	1µs	1µs
$O(log(n))$	2.3µs	3.0µs	3.4µs
$O(n)$	10µs	20µs	30µs
$O(n^2)$	0.1ms	0.4ms	0.9ms
$O(n^4)$	10ms	160ms	810ms
$O(2^n)$	1ms	1 sec	18 min

Fibonacci series

- $Fib(1) = 1$
- $Fib(2) = 1$
- $Fib(n) = Fib(n\text{-}1) + Fib(n\text{-}2)$, for $n > 2$

1 1 2 3 5 8 13 21 34 55 89 144...

Fibonacci series



Fibonacci series

Recursive implementation:

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

Fibonacci series

Universität Zürich[UZH]

Recursive implementation:

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

- `fib(3)`: 2 steps
- `fib(4)`: 3 steps
- `fib(5)`: 5 steps
- `fib(6)`: 8 steps
- `fib(7)`: 13 steps
- `fib(8)`: 21 steps

Fibonacci series, recursive implementation

- `fib(3)`: 2 steps
- `fib(4)`: 3 steps
- `fib(5)`: 5 steps
- `fib(6)`: 8 steps
- `fib(7)`: 13 steps
- `fib(8)`: 21 steps

What is the complexity?

Fibonacci series, recursive implementation

- `fib(3)`: 2 steps
- `fib(4)`: 3 steps
- `fib(5)`: 5 steps
- `fib(6)`: 8 steps
- `fib(7)`: 13 steps
- `fib(8)`: 21 steps
- `fib(9)`: 34 steps
- `fib(10)`: 55 steps

- `fib(20)`: 6 765
- `fib(30)`: 832 040

- `fib(40)`: 102 334 155 steps

What is the complexity?

Fibonacci series, recursive implementation

- `fib(3)`: 2 steps
- `fib(4)`: 3 steps
- `fib(5)`: 5 steps
- `fib(6)`: 8 steps
- `fib(7)`: 13 steps
- `fib(8)`: 21 steps
- `fib(9)`: 34 steps
- `fib(10)`: 55 steps

- `fib(20)`: 6 765
- `fib(30)`: 832 040

- `fib(40)`: 102 334 155 steps

What is the complexity?

$O(c^x)$, $1 < c < 2$; exponential on input value x

Tagging

- Sentence with n words given
- One of m tags can be assigned to each word
- Task: find the most likely tag sequence
- Naive solution:
  - generate all possible tag sequences
  - select the one with the highest probability
- Complexity?

Tagging

- Sentence with n words given
- One of m tags can be assigned to each word
- Task: find the most likely tag sequence
- Naive solution:
  - generate all possible tag sequences
  - select the one with the highest probability
- Complexity:
  - there is m^n possible tag sequences
  - $\in O(m^n)$, exponential on sentence length

Tagging

- $n = 20$ words in a sentence
- each word has one of $m = 50$ tags
- number of different ways of tagging the sentence: $m^n = 9.537 \times 10^{33}$
    - 3.2×10^{20} years
    - earth is only $\sim 4.5 \times 10^9$ years old

Outline

- Part 1: Complexity
- Part 2: Dynamic Programming

Dynamic programming

A way of optimizing complex tasks and avoiding bad complexity

Applications:

- Syntax parsing
- Sentence alignment
- Tagging

Dynamic programming

Basic idea:

- split the task into smaller sub-tasks
- solve the sub-tasks, **saving the intermediate results**
- the solution to the final task might use only some intermediate results
  - unless it's possible to tell which will or will not be used, all sub-tasks are solved

Fibonacci series
Up-down

```
def fib(a):




    if (a == 1 or a == 2):
        return 1
    else:

        return fib(a-1) + fib(a-2)

print(fib(40))
```

Fibonacci series
Up-down, Memoization

Memoization (NOT Memorization)

```
def fib(a):
    memory = {}            # using a dict as a memory

    if a in memory:        # if already computed
        return memory[a] # then retrieve solution from memory
    if (a == 1 or a == 2):
        return 1
    else:
        memory[a] = fib(a-1) + fib(a-2)  # new sub-solution into memory
        return memory[a]

print(fib(40))
```

Fibonacci series Bottom-up

Universität Zürich[UZH]

```
def fib(a):
    if a < 3:
        return 1
    else:
        pprev = 1
        prev = 1
        for i in range(3, a + 1):
            curr = pprev + prev
            pprev = prev
            prev = curr
        return curr
print(fib(40))
```

- no recursion

Fibonacci series
Bottom-up

```
def fib(a):
    memory = []
    for i in range(a + 1):
        if i < 3:
            memory.append(1)
        else:
            memory.append(memory[-1] + memory[-2])

    return memory[-1]

print(fib(40))
```

- no recursion

Fibonacci series

- Naive solution: $O(c^n)$
- Dynamic programming solutions: $O(n)$
  - the sub-tasks overlap
  - naive algorithm solves the same sub-tasks over and over again
- Strategies:
  - solve each sub-task once and store in memory
    solve task based on memorised sub-task solutions

    OR

  - bottom-up: start with the smallest sub-tasks and reach the full task "at the top", to replace recursion

Longest Common Subsequence (LCS)

- Given two strings, finds **<u>a</u>** longest common sequence of characters
  - the subsequence can have gaps (≠substring)

Longest Common Subsequence (LCS)

Universität Zürich[UZH]

- Given two strings, finds **a** longest common sequence of characters
  - the subsequence can have gaps (≠substring)
- For example

  `lcs`("börsenstraße", "boersenstrasse") = ["b", "r", "s", "e", "n", "s", "t", "r", "a", "e"]

  `lcs`("jumper", "jumps") = ["j", "u", "m", "p"]

- Applications:
  - highlight/correct differences between two texts
    - diff command
  - bioinformatics (gene sequence comparison)
  - …

LCS via Dyn. Prog.

Given x[1..m]and y[1..n]

Simplification:

- Look at length of LCS (x, y)
- Extend the algorithm to find LCS itself.

How to split lcs(x, y) into sub-tasks?

LCS via Dyn. Prog.

Universität Zürich[UZH]

Given x[1..m]and y[1..n]

Simplification:

- Look at length of LCS (x, y)
- Extend the algorithm to find LCS itself.

How to split lcs(x, y) into sub-tasks?

- Assume we know the C[i-1, j-1] = | LCS(x[1..i-1], y[1..j-1] |
    - If x[i] = y[j] → the last character of x and y is the same?

      C[i, j] = C[i-1, j-1] +1
    - the last character of xs and ys is different?

      C[i, j] = max{ C[i, j-1], C[i-1, j]}, otherwise

LCS
Up-down, recursive

```
#Recursive
#Simplification: Length


def lcs_len(x, y, i, j):



      if (x and y): #for both non-empty strings
         if x[i] == y[j]: #equal last characters
            return lcs_len(x, y, i-1, j-1) + 1
         else:                         #different last characters
            return max(lcs_len(x, y, i-1, j), lcs_len(x, y, i, j-1))

       else:                #for strings, one of which is empty
            return 0        #if one of the strings is empty, the LCS


print lcs_len("jumper", "jumps", 6, 5)  # LCS length = 4
```

LCS
Up-down, recursive

Universität Zürich[UZH]

```
#Recursive, Memoization (NOT Memorization)
#Simplification: Length

from collections import defaultdict

def lcs_len(x, y, i, j):
   global matrix

   if not matrix[i][j]: #check if solution is already memorized
      if (x and y): #for non-empty strings
         if x[i] == y[j]: #equal last characters
            matrix[i][j] = lcs_len(x, y, i-1, j-1) + 1
         else:              #different last characters
            matrix[i][j] = max(lcs_len(x, y, i-1, j), lcs_len(x, y, i, j-1))


   return matrix[i][j]

matrix = defaultdict(lambda: defaultdict(list)) #initialize memory
print lcs_len("jumper", "jumps", 6, 5)  # LCS length = 4
```

LCS(“fear”, “fair”) bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅					
f					
f**a**					
fa**i**					
fai**r**					

LCS("fear", "fair")
bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0				
f					
f**a**					
fa**i**					
fai**r**					

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f					
f**a**					
fa**i**					
fai**r**					

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0				
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f			
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f			
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1			
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair")
bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f		
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f		
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1		
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**<u>f</u>**	f**<u>e</u>**	fe**<u>a</u>**	fea**<u>r</u>**
∅	0	0	0	0	0
<u>f</u>	0	f = f 1	e ≠ f 1	a ≠ f 1	
f**<u>a</u>**	0				
fa**<u>i</u>**	0				
fai**<u>r</u>**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0				
fa**i**	0				
fai**r**	0				

LCS(“fear”, “fair”) bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1			
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1		
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a	
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0				
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1			
fai**r**	0				

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1		
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	
fai**r**	0				

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0				

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1			

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1	1		

LCS(“fear”, “fair”) bottom-up

	∅	**<u>f</u>**	f**<u>e</u>**	fe**<u>a</u>**	fea**<u>r</u>**
∅	0	0	0	0	0
<u>f</u>	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**<u>a</u>**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**<u>i</u>**	0	1	1	2	2
fai**<u>r</u>**	0	1	1	2	

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1	1	2	3

LCS("fear", "fair") bottom-up

	Ø	**f**	f**e**	fe**a**	fea**r**
Ø	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1	1	2	3

r

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1	1	2	3

a r

LCS("fear", "fair") bottom-up

Universität Zürich[UZH]

	Ø	**f**	f**e**	fe**a**	fea**r**
Ø	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1	1	2	3

f a r

LCS("fear", "fair") bottom-up

	∅	**f**	f**e**	fe**a**	fea**r**
∅	0	0	0	0	0
f	0	f = f 1	e ≠ f 1	a ≠ f 1	r ≠ f 1
f**a**	0	f ≠ a 1	e ≠ a 1	a = a 2	r ≠ a 2
fa**i**	0	1	1	2	2
fai**r**	0	1	1	2	3

f a r

Plan for next lecture:

More dynamic programming:

- Levenshtein distance
- Sentence alignment
- Tagging: Viterbi algorithm

Lecture 10: Complexity, Dynamic Programming

PCL II, CL, UZH
May 4, 2016

Universität Zürich[UZH]