Data Structures and Algorithms with Python

Heikki Peura

h.peura@imperial.ac.uk

Lecture 2

Last time

Some algorithms were **faster** than others

- Bisection vs. Heron
- Recursive vs. iterative Fibonacci

Plan for today:

- How to think about **algorithm complexity** more formally
- Designing **search** and **sorting** algorithms

Why is recursive Fibonacci slow?

```python
def fiboRec(n):
    if n == 0 or n == 1:
        return n
    else:
        return fiboRec(n-1) + fiboRec(n-2)
```

```python
def fiboIter(n):
    fibNumbers = [0,1]
    for i in range(2,n+1):
        fibNumbers.append(fibNumbers[-1] + fibNumbers[-2])
    return fibNumbers[n]
```

Goals in designing algorithms

1. **Correctness** — returns the correct answer for any input
2. **Efficiency** — returns the answer quickly

- It is important to understand both: think of airplane software, Uber or algorithmic trading...

Efficiency (recall what computers were good at):

- How much **time** will our computation take?
- How much **memory** will it need?

How much time will it take?

Simple: run and time it? But time depends on

1. Speed of computer
2. Specifics of implementation
3. Value of input

We can avoid 1 and 2 by measuring time in the **number of basic steps executed**

- Step: **constant-time computer operation** for any input
  - Assignment (eg `x=2`)
  - Comparison (eg `x>y`)
  - Arithmetic operations (eg `x*y`) (for not too large numbers)
  - Accessing memory

For 3, we can **measure time depending on the size of input**

- **Time ≈ complexity** (# operations given input size)

Complexity and input

Searching for an item in a list?

```python
def linearSearch(A,x):
    # A is a list of length n
    for elem in A:
        if elem == x:
            return True
    return False
```

- x could be the first element of A
- x could not be in A
- **How to give a general complexity measure?**

Complexity cases

Cases for given input size (length of A):

- **Best case** — minimum time
- **Worst case** — maximum time
- **Average case** — average or expected time over all possible inputs

Principle: focus on worst-case analysis

- Upper bound on running time
- Bonus: usually easier to analyze

Example

```python
def factIter(n):
    result = 1                 # 1 step
    while n > 1:               # 1 step
        result = result * n    # 2 steps
        n = n - 1              # 2 steps
    return result              # 1 step
```

Total: $5n + 2$ steps

- As n gets large, 2 is irrelevant
- Arguably, so is 5
  - It's the size of the problem that matters

Principle 2: ignore constant factors and lower-order terms

- These depend on computer and program implementation
- We lose little predictive power (coming up!)
- It makes our life easier...

Example

```
def f(x):                    # x integer
    ans = 0                  # 1 step
    for i in range(100)
        ans += 1             # 200 steps
    for i in range(x):
        ans += 1             # 2*x
    for i in range(x)
        for j in range(x)
            ans -= 1         # 2*x^2
    return ans               # 1 step for return
```

Complexity is $202 + 2x + 2x^2$

- x small -> first loop dominates ($x = 3$)
- x large -> last loop dominates ($x = 10^6$)
- Only need to consider last (nested) loop for large x
- Does the 2 in $2x^2$ matter? For large x, order of growth much more important

Asymptotic analysis

Principle 0: measure number of basic operations as function of input size

Principle 1: focus on worst-case analysis

Principle 2: ignore constant factors and lower-order terms

Principle 3: only care about large inputs

- Only large problems are interesting
- What happens when size gets very large?

Formal way to describe this approach:

- Big-Oh notation: upper bound on worst-case running time

Big-Oh definition

Let $T(n)$ be a function on $n = 1, 2, 3, ...$, for example:
$T(n) = 202 + 2n + 2n^2$

Informally: $T(n) = O(f(n))$ if for all sufficiently large n, $T(n)$ is bounded above by a constant multiple of $f(n)$

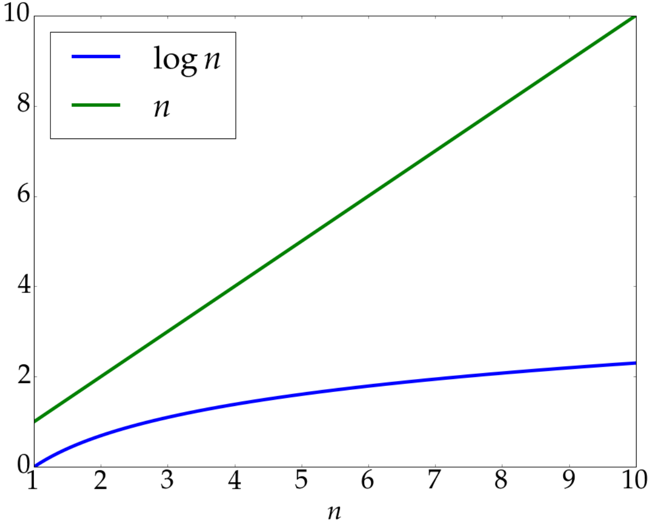
Formally: $T(n) = O(f(n))$ if and only if there exist constants $c, n_0 > 0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$.

Complexity classes

Fast algorithm: worst-case running time grows slowly with input size

- $O(1)$: constant running time — primitive operations
- $O(\log n)$: logarithmic running time
- $O(n)$: linear running time
- $O(n \log n)$: log-linear time
- $O(n^c)$: polynomial running time
- $O(c^n)$: exponential running time

Complexity matters with large inputs



Search algorithms

Finding an item in a list?

```
def linearSearch(A,x):
    for elem in A:
        if elem == x:
            return True
    return False
```

Linear search on unsorted list with n items: worst-case $O(n)$ operations

Can we do better?

- No...
- But what if the list is sorted?

Recall bisection search

- Start with a guess g as average of search range $low = 0$ and $high = \max\{1.0, x\}$
- If $g * g$ is close to x, stop and return g as the answer
- Otherwise, if $g * g < x$, update search range: $low = g$
- Otherwise, if $g * g >= x$, update search range: $high = g$
- Make new guess as average of updated search range
- Repeat process using new guess until close enough

Binary search on sorted list

Algorithm for finding item x in sorted list L:

- Pick an index i roughly dividing L in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, recursively search left half of L
  - Otherwise recursively search right half

Binary search example

Algorithm for finding x in sorted list L:

- Pick an index i roughly dividing L in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, recursively search left half of L
  - Otherwise recursively search right half

Find number 24 in a list $L = [9, 24, 32, 56, 57, 59, 61, 99]$

First iteration

9	24	32	**56**	57	59	61	99

9	24	32

56	57	59	61	99

$L[i] = 56 > 24$ —> discard right half and recursively call binary search on left half

Second iteration

9	**24**	32

56	57	59	61	99

$L[i] = 24$ —> return True

Binary search complexity

Algorithm for finding x in list L:

- Pick an index i roughly dividing L in half
- If $L[i] == x$, return True (if nothing left to search return False)
- If not:
  - If $L[i] > x$, recursively search left half of L
  - Otherwise recursively search right half

Complexity = **# of recursive calls** * **Constant time per call**

But **how many** recursive calls?

- How many times can you split n items in half?
- $\log_2(n)$ (but base of logarithm does not matter for big-Oh)
- Complexity $O(\log n)$ – much better than $O(n)$!

Sorting algorithms

So if we have an unsorted list, should we sort it first?

- Suppose complexity $O(sort(n))$
- Is $sort(n) + \log(n) < n$?
- No...

But what if we need to search repeatedly, say k times?

- Is $sort(n) + k\log(n) < kn$?
- Depends on k...

Sorting algorithms

How would you intuitively sort a list L?

56	24	99	32	9	61	57	79
9	24	99	32	56	61	57	79
9	24	99	32	56	61	57	79
9	24	32	99	56	61	57	79
9	24	32	56	99	61	57	79
9	24	32	56	57	61	99	79
9	24	32	56	57	61	99	79
9	24	32	56	57	61	79	99

In words: Start with an (empty) “prefix” and a “suffix” (equal to L) and iteratively move the smallest element of suffix into prefix

Selection sort algorithm

Selection sort list L of length n:

- Initialization step: divide the list into a “prefix” P and a “suffix” S with initially P empty and $S = L$
- Main loop:
  - Search for the smallest element of S and move it to the end of P (in other words, swap its position with the first element of S)
  - Repeat until S is empty

Correctness (for those into math): by induction

- (exercise: convince yourself!)

Selection sort complexity

Complexity:

- Repeat until suffix S empty: $O(n)$ passes of main loop
- Each pass: search for the smallest element in $O(n)$
- Total $O(n^2)$

Can we do better?

- Yes! Merge sort is $O(n \log n)$ — challenge problem
- You can't do any better than that...

Complexity classes

Fast algorithm: worst-case running time grows slowly with input size

- $O(1)$: constant running time — primitive operations
- $O(\log n)$: logarithmic running time — binary search
- $O(n)$: linear running time — linear search
- $O(n \log n)$: log-linear time — merge sort
- $O(n^c)$: polynomial running time — selection sort
- $O(c^n)$: exponential running time — ??

Workshop

After the break...

Search and sort implementations

Complexity analysis and more Python practice