Algorithms and data structures

Computing Methods for Experimental Physics and Data Analysis

L. Baldini

luca.baldini@pi.infn.it

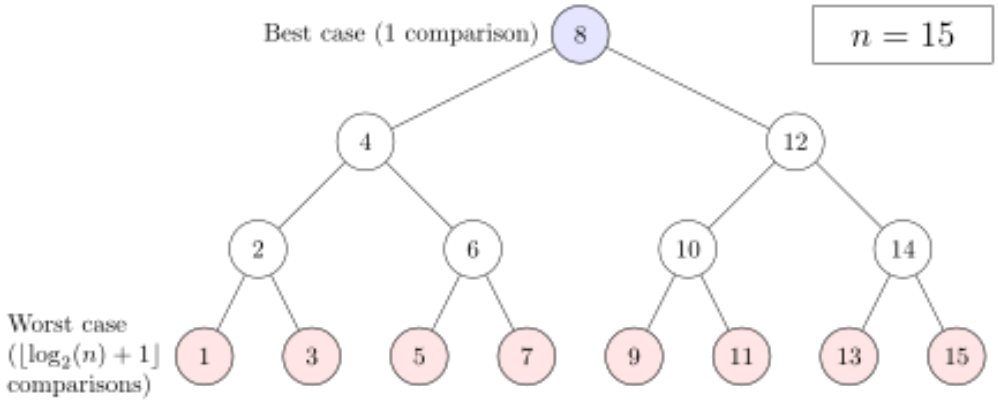
Università and INFN–Pisa

Compiled on October 17, 2019

What is an algorithm?

- Algorithm: a sequence of instructions to perform a computation
- Algorithms can be expressed in several different ways
  - Flowcarts
  - Pseudo-code
  - Working code snippets
- The sequence of operation must be expressed unambigously
- Using the right algorithm sometimes makes all the difference between being able to attack a problem in a practical time or not

Example: sequential vs. binary search



- Problem: find an element in a sorted list
- Brute force: loop over the list until you find (or not) the target
- Binary search
  - Start from the middle (if that's the target you're done)
  - If the target is smaller (larger) than the element in the center, bisect the half-list on the left (right)
  - Iterate until you've found the target (or exhausted the list)
- Which algorithm is more efficient?
- (This problem is germane to that of finding a person on the phonebook)

Complexity of an algorithm

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/find_max.py

```
1  def find_maximum(list_):
2      """Find the biggest element in a list.
3      """
4      maximum = list_[0]
5      for value in list_[1:]:
6          if value > maximum:
7              maximum = value
8      return maximum
9
10 l = [1, 2, 5, 98, 3, 1672, 6, 34, 651]
11 print(find_maximum(l))
12
13 [Output]
14 1672
```

- Example: find the largest element in a list of length n
- How many fundamental instructions is this code executing?
  - (You should realize this is an ill-posed question)
  - One assignment and one list lookup at the beginning: 2
  - One lookup, one assignment and one comparison for each iteration in the for loop: $3(n-1)$
  - A variable number of assignments: between 0 and $(n-1)$
  - One final return instruction: 1
- Answer: anything between $3n$ and $4n-1$
  - (Depending on the input list)

Complexity of an algorithm

Continued

- Message #1: the exact number of fundamental instructions that an algorithm performs is not determined a priori
  - It depends on the input data, instead
  - And so does the running time
- There's a few questions that you can legitimetely ask, anyway
  - How many instructions in the worst case?
  - How many instructions in the best case?
  - How many instructions on average?
- Message #2: the *exact* number of operations doesn't really matter, does it?
  - Different machines have different executions speed
  - Different languages have different meaning of *fundamental instruction*
- Message #3: still, the running time is related to the number of fundamental instructions

Asymptotic behavior and big-O notation

- Say you have an algorithm operating on an input of lenght n
  - e.g., a list with n elements
  - or a string with n characters
- How many fundamental instructions N does it take to for your algorithm to run?

$$N = f(n)$$

- Asymptotic behavior: drop all the terms that grow slowly with n and only keep the one that grows faster

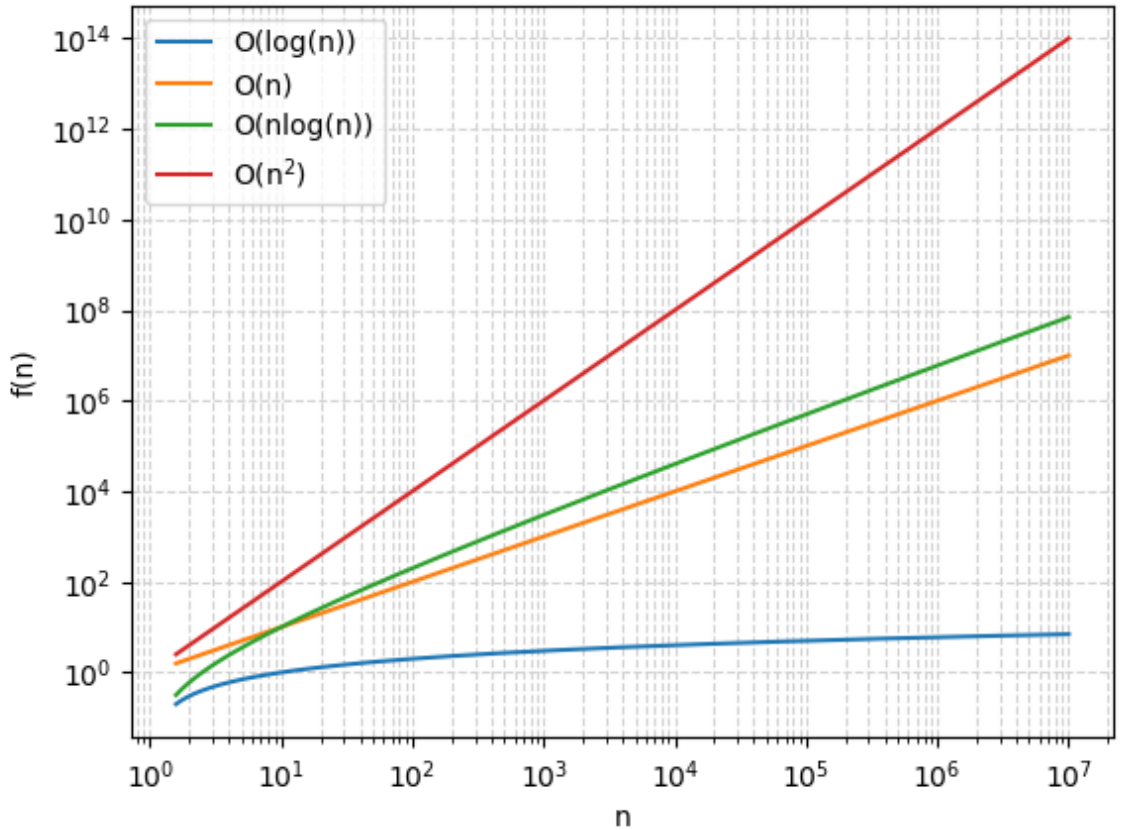
$$4n - 1 \approx 4n \quad \text{and} \quad 2n^2 + 6n + 3 \approx 2n^2$$

(for large n)

- Let's go one step further, and say that we neglect the multiplicative factor in front of the leading term

$$4n - 1 \approx n \quad \text{and} \quad 2n^2 + 6n + 3 \approx n^2$$

- big-O notation: the two algorithms are $O(n)$ and $O(n^2)$



▷ Yes! If $n = 10^6$ and you can beat down the complexity from n^2 to $n \log(n)$ you are cutting down the execution time by one million!

Asymptotic behavior

How do I measure it?

- By brute force
  - Implement the algorithm
  - Run it on input data of different size and time the run
  - (Be careful: results may vary from run to run)
  - Plot the running time vs. input size
- By analysis
  - Go ahead and count the instructions
  - Evaluate the best, worst and average case
  - (This can be difficult for complex programs, and subject to the hidiosyncrasis of the language)
- By eye
  - One loop: $O(n)$
  - Two sequential loops: $O(n)$
  - Two nested loops: $O(n^2)$

SPECIAL SECTION

Guest Editor
Peter Neumann

The Complexity of Songs

DONALD E. KNUTH

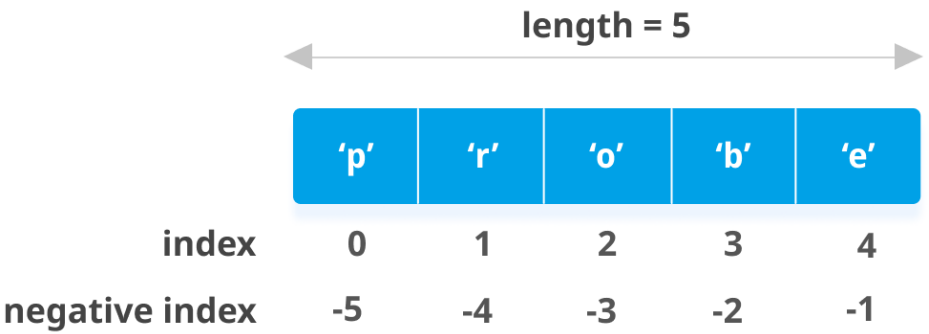
Every day brings new evidence that the concepts of computer science are applicable to areas of life which have little or nothing to do with computers. The purpose of this survey paper is to demonstrate that important aspects of *popular songs* are best understood in terms of modern complexity theory.

It is known [3] that almost all songs of length n require a text of length $\sim n$. But this puts a considerable space requirement on one's memory if many songs are

By the Distributive Law and the Commutative Law [4], we have

$$\begin{aligned} c &= n - (V + R)m + mV \\ &= n - Vm - Rm + Vm \\ &= n - Rm. \end{aligned} \tag{3}$$

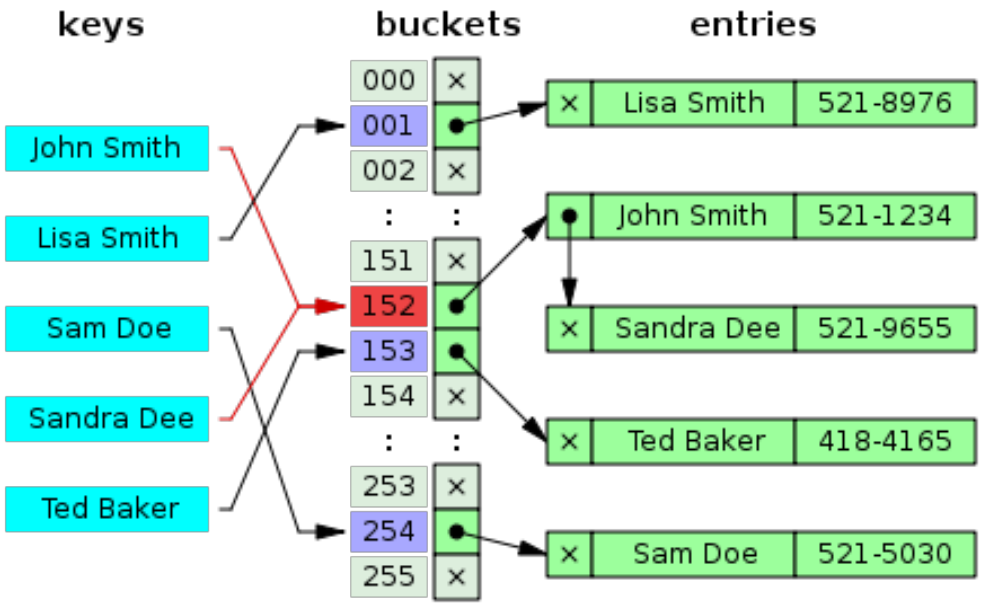
The lemma follows. □



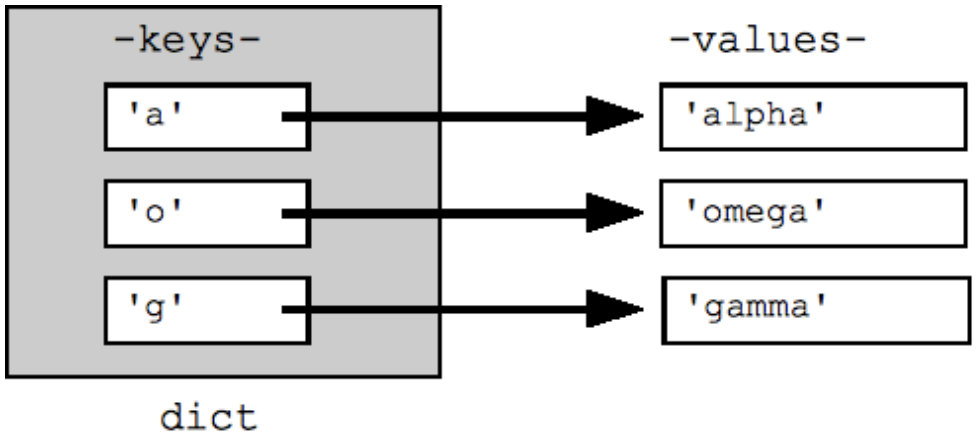
Operation	Average case	Worst case
Copy	O(n)	O(n)
Append	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

Hash tables

Why do I even bring this up? Python dictionaries are hash tables



- Associative array mapping keys to values
- Basic idea:
  - Pre-allocate some space (which might grow or shrink)
  - Keys are mapped to indices via a hash function
  - This is about it, except that you have to be able to handle collisions
- Hash tables (aka dictionaries) are highly optimized in Python



Operation	Average case	Worst case
Copy	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration	O(n)	O(n)

An odd corner of Python dictionaries

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/dict_hashing.py

```
1  print(hash(3))
2  print(hash(3.))
3
4  d = {}
5
6  d[3] = 'Hi there!'
7  print(d)
8
9  d[3.] = 'How are you?'
10 print(d)
11
12 [Output]
13 3
14 3
15 3: 'Hi there!'
16 3: 'How are you?'
```

- When a float corresponds to an integer, its hash is the same as that of the integer
- The hash is used to map keys into indices
- Therefore: 3 and 3. are the same key to a dictionary

A real-life example: sorting

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/sloppy_sort.py

```
1  def sloppy_sort(list_):
2      """Poor man's implementation of a sorting algorithm.
3      """
4      sorted_list = []
5      for item in list_:
6          if len(sorted_list) == 0:
7              sorted_list.append(item)
8          else:
9              if item < sorted_list[0]:
10                 sorted_list.insert(0, item)
11             else:
12                 for i, sorted_item in enumerate(sorted_list):
13                     if item <= sorted_item:
14                         sorted_list.insert(i, item)
15                         break
16     return sorted_list
17
18 l = [10, 1, 5, 2, 7, 3, 9, 4]
19 print(l)
20 print(sloppy_sort(l))
21
22 [Output]
23 [10, 1, 5, 2, 7, 3, 9, 4]
24 [1, 2, 3, 4, 5, 7, 9, 10]
```

- Say you are asked to implement an algorithm to sort a list
  - Chances are that you would come up with two nested loops, $O(n^2)$

Sorting algorithms on the market

aka Don't try this at home

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$ variation is n	$n \log n$	n^2	$\log n$ on average, worst case space complexity is n; Sedgewick variation is $\log n$ worst case.	Typical in-place sort is not stable; stable versions exist.	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n A hybrid block merge sort is $O(1)$ mem.	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm[7] or, more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place merge sort	—	—	$n \log^2 n$ See above, for hybrid, that is $n \log n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Heapsort	n If all keys are distinct, $n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[9]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n comparisons when the data is already sorted or reverse sorted.
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n comparisons when the data is already sorted or reverse sorted.
Shell sort	$n \log n$	Depends on gap sequence	Depends on gap sequence; best known is $n^{4/3}$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications. There is a worst case $O(n(\log n)^2)$ gap sequence but it loses $O(n \log n)$ best case time.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Binary tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.
Library sort	n	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	n	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	

What is Python using? Timsort

The name comes from Tim Peters

https://github.com/lucabaldini/cmepda/tree/master/slides/latex/snippets/timsort.py

```
1  l = [10, 1, 5, 2, 7, 3, 9, 4]
2  print(l)
3  l.sort()
4  print(l)
5
6  [Output]
7  [10, 1, 5, 2, 7, 3, 9, 4]
8  [1, 2, 3, 4, 5, 7, 9, 10]
```

- Hybrid algorithm, derived from merge sort and insertion sort
  - Find subsequences of the data that are already ordered
  - Use that knowledge to sort the remainder more efficiently
- "Although practicality beats purity" (The Zen of Python)

References

- https://en.wikipedia.org/wiki/Algorithm
- https://discrete.gr/complexity/
- https://wiki.python.org/moin/TimeComplexity
- https://en.wikipedia.org/wiki/Sorting_algorithm
- https://bugs.python.org/file4451/timsort.txt
- https://en.wikipedia.org/wiki/Timsort