Algorithmic complexity: Speed of algorithms

Ruth Anderson
UW CSE 160
Bonus Material - Winter 2020
How fast does your program run?

• Usually, this *does not matter*
• Correctness is more important than speed

• Computer time is much cheaper than human time
• The cost of your program depends on:
  – Time to write and verify it
    • High cost: salaries
  – Time to run it
    • Low cost: electricity
• An inefficient program may give you results faster
Sometimes, speed does matter

• Programs that need to run in real time
  – e.g. will my car crash into the car in front of me?

• Very large datasets
  – Even inefficient algorithms usually run quickly enough on a small dataset
  – Example large data set:
    Google:
    67 billion pages indexed (2014)
    5.7 billion searches per day (2014)
    Number of pages searched per day??
Program Performance

We’ll discuss two things a programmer can do to improve program performance:

• Good Coding Practices – covered 2/28/2020
• Good Algorithm Choice
Good Algorithm Choice

• Good choice of algorithm can have a much bigger impact on performance than the good coding practices mentioned.
• However good coding practices can be applied fairly easily
• Trying to come up with a better algorithm can be a (fun!) challenge
• Remember: Correctness is more important than speed!!
How to compare two algorithms?

• Implement them both in Python
• Run them and time them
A Better Way to Compare Two Algorithms

• Hardware?
  – Count number of “operations” something independent of speed of processor

• Properties of data set? (e.g. almost sorted, all one value, reverse sorted order)
  – Pick the worst possible data set: gives you an upper bound on how long the algorithm will take
  – Also it can be hard to decide on what is and “average” data set

• Size of data set?
  – Describe running time of algorithm as a function of data set size
Asymptotic Analysis

• Comparing “Orders of Growth”
• This approach works when problem size is large
  – When problem size is small, “constant factors” matter
• A few common Orders of Growth:

<table>
<thead>
<tr>
<th>Order</th>
<th>Notation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>O(1)</td>
<td>integer + integer</td>
</tr>
<tr>
<td>Linear</td>
<td>O(n)</td>
<td>iterating through a list</td>
</tr>
<tr>
<td>Quadratic</td>
<td>O(n^2)</td>
<td>iterating through a grid</td>
</tr>
</tbody>
</table>

Example:
Which Function Grows Faster?

\[ O(n^3) \quad \text{vs.} \quad O(n^2) \]

\[ n^3 + 2n^2 \quad \text{vs.} \quad 100n^2 + 1000 \]
Running Times of Python Operations

**Constant Time operations: O(1)**
- Basic Math on numbers (+ - * /)
- Indexing into a sequence (eg. list, string, tuple) or dictionary
  - E.g. `myList[3] = 25`
- List operations: `append`, `pop` (at end of list)
- Sequence operation: `len`
- Dictionary operation: `in`
- Set operations: `in`, `add`, `remove`, `len`

**Linear Time operations: O(n)**
- `for` loop traversing an entire sequence or dictionary
- Built in functions: `sum`, `min`, `max`, slicing a sequence
- Sequence operations: `in`, `index`, `count`
- Dictionary operations: `keys()`, `values()`, `items()`
- Set operations: `&`, `|`, `-`
- String concatenation (linear in length of strings)

**Note:** These are general guidelines, may vary, or may have a more costly worst case. Built-in functions (e.g. `sum`, `max`, `min`, `sort`) are often faster than implementing them yourself.
Example: Processing pairs

def make_pairs(list1, list2):
    """Return a list of pairs.
    Each pair is made of corresponding elements of list1 and list2.
    list1 and list2 must be of the same length."""

    ...

assert make_pairs([100, 200, 300], [101, 201, 301]) == [[100, 101], [200, 201], [300, 301]]

• 2 nested loops vs. 1 loop
• Quadratic ($n^2$) vs. linear (n) time
def search(value, lst):
    """Return index of value in list lst.
The value must be in the list."""
    ...

• Any list vs. a sorted list
• Linear (n) vs. logarithmic (log n) time
Example: Sorting

def sort(lst):
    """Return a sorted version of the list lst. The input list is not modified.""
    ...

assert sort([3, 1, 4, 1, 5, 9, 2, 6, 5]) == [1, 1, 2, 3, 4, 5, 5, 6, 9]

• selection sort vs. quicksort
• 2 nested loops vs. recursive decomposition
• time: quadratic (n²) vs. log-linear (n log n) time

Note: Calling built in sorting methods sort or sorted in Python has O(n log n) time